โครงสร้างข้อมูล

v2.0.1

ฉบับวาจาจาวา

สมชาย ประสิทธิ์จูตระกูล

https://www.cp.eng.chula.ac.th/books/ds-vjjv

คำนำ
พิมพ์ครั้งที่ 2

คอมพิวเตอร์แบบอิเล็กทรอนิกส์ถือกำเนิดมาเกือบ 70 ปีแล้ว (มีอายุพอๆ กับเครื่องรับโทรทัศน์สี) จากอดีตจนถึงปัจจุบัน โครงสร้างข้อมูลได้รับการบรรจุเป็นองค์ความรู้สำคัญพื้นฐานที่ต้องศึกษากันในศาสตร์ทางวิทยาการคอมพิวเตอร์และสาขาอื่น ๆ ที่เกี่ยวข้อง ถือได้ว่า เป็นวิชาที่สองที่ผู้เรียนต้องฝึกทักษะการเขียนโปรแกรม ต่อจากวิชาการเขียนโปรแกรมคอมพิวเตอร์ โดยทั่วไปเนื้อหาแกนทางโครงสร้างข้อมูลอาจมีการเปลี่ยนแปลงบ้าง แต่ก็ไม่มากนัก ส่วนที่เปลี่ยนไปตามยุคตามสมัย เห็นจะเป็นภาษาโปรแกรมคอมพิวเตอร์ที่ใช้บรรยายการจัดเก็บและจัดการข้อมูล ซึ่งเปลี่ยนแปลงไปตามความนิยมในภาคอุตสาหกรรมซอฟต์แวร์

หนังสือ “โครงสร้างข้อมูล : ฉบับวาจาจาวา” เล่มนี้เป็นหนึ่งในหนังสือทางโครงสร้างข้อมูลที่เลือกใช้ภาษาจาวาเป็นสื่อในการนำเสนอแนวคิดในการจัดเก็บข้อมูลอย่างมีระเบียบและจัดการข้อมูลอย่างมีระบบ ผู้เขียนมีความยินดีเป็นอย่างยิ่งที่หนังสือเล่มนี้ได้รับการเผยแพร่และนำไปใช้ในหลากหลายสถาบันการศึกษา สำหรับการพิมพ์ครั้งที่ 2 ของหนังสือเล่มนี้ ยังคงเนื้อหาเดิม แต่ได้แก้ไขเนื้อหาของครั้งแรกที่พิมพ์ผิด

ผู้เขียนขอขอบคุณผู้อ่านทั้งหลายที่ได้ส่งจดหมาย (อิเล็กทรอนิกส์) พร้อมข้อเสนอแนะต่าง ๆ ซึ่งผู้เขียนจะรับไปปรับปรุงเพิ่มเติมในโอกาสต่อไป

สมชาย ประสิทธิ์จูตระกูล

ภาควิชาวิศวกรรมคอมพิวเตอร์

จุฬาลงกรณ์มหาวิทยาลัย

somchaip@chula.ac.th

๒๕/๕/๒๕๕๒

จ. ๒.๐

คำนำ

พิมพ์ครั้งที่ 1

โครงสร้างข้อมูลเป็นหนึ่งในองค์ความรู้ขั้นพื้นฐานของการศึกษาทางวิทยาการคอมพิวเตอร์ วิศวกรรมคอมพิวเตอร์ และอื่น ๆ อีกหลากหลายสาขา ที่ว่าด้วยการจัดเก็บข้อมูลอย่างมีระเบียบ และการจัดการข้อมูลอย่างมีระบบ เพื่อให้ตรงตามความต้องการในการประมวลผลข้อมูลอย่างมีประสิทธิภาพ การศึกษาโครงสร้างข้อมูลจึงต้องอาศัยความรู้และความชำนาญทางการเขียนโปรแกรม (จากวิชาการเขียนโปรแกรมคอมพิวเตอร์เบื้องต้น) เพื่อพัฒนาโครงสร้างการจัดเก็บข้อมูลที่ออกแบบไว้ให้เห็นจริง และอาศัยความสามารถทางตรรก การนับ การวิเคราะห์ และการจำลอง (จากวิชาคณิตศาสตร์ดีสครีตหรือบางทีเรียกว่าวิยุตคณิตศาสตร์) เพื่อเป็นเครื่องมือในการออกแบบและวิเคราะห์วิธีการจัดการข้อมูลว่ามีประสิทธิภาพเพียงใด

วัตถุประสงค์หลักของการศึกษาเรื่องโครงสร้างข้อมูลพื้นฐานคือเพื่อให้ "เลือกเป็น, ใช้เป็น, และสร้างเป็น" เนื่องจากไม่มีโครงสร้างข้อมูลใดที่สนองความต้องการได้ทุกการใช้งานได้รวดเร็ว เราต้องรู้ว่าเมื่อไรควรเลือกใช้โครงสร้างข้อมูลแบบใดกับงานประเภทใด ด้วยประสิทธิภาพที่ต่างกันอย่างไรทั้งทางทฤษฎีและทางปฏิบัติ ดังนั้นจึงต้องรู้ความแตกต่างของโครงสร้างข้อมูลที่มีหลากหลายแบบที่สนองความต้องการเดียวกัน เมื่อเลือกได้แล้วก็ต้องเรียกใช้บริการต่าง ๆ ที่ตัวข้อมูลมีให้ได้อย่างถูกต้อง รู้เงื่อนไขที่จำเป็นต้องมีก่อนใช้บริการ รู้ว่าอะไรคือผลที่ได้หลังการใช้บริการ และต้องจำชื่อหรือมีทักษะในการค้นหาชื่อของบริการต่าง ๆ ให้ได้ [†] และสุดท้ายคือต้องรู้วิธีการแปลงแนวคิดการจัดเก็บและจัดการข้อมูล เขียนออกมาเป็นโปรแกรมที่ใช้งานจริงได้อย่างมีประสิทธิภาพ ถึงแม้ว่าในปัจจุบันจะมีคลังคลาสมาตรฐานมากมายให้ใช้ได้เลยโดยไม่ต้องเขียนเอง แต่การเขียนโครงสร้างข้อมูลได้เองย่อมเป็นหลักฐานยืนยันความเข้าใจในเนื้อหาโดยไม่ต้องพิสูจน์เรื่องอื่นใดอีก

[†] ชื่อต่าง ๆ ที่ใช้ในหนังสือนี้จะพยายามล้อเลียนการตั้งชื่อตามมาตรฐานของคลังคลาสจาวา หรือไม่ก็ใช้ชื่อที่เป็นที่ยอมรับกันในวงการ

แล้วจะทำอย่างไรจึงจะบรรลุวัตถุประสงค์ ? ต้องเข้าใจด้วยว่าการศึกษาองค์ความรู้ใดในวิทยาการคอมพิวเตอร์ คงไม่สามารถทำได้ด้วยการอ่านและจำแต่เพียงอย่างเดียว เราต้องอ่านไป คิดไป และเถียงไปด้วยว่าน่าจะมีอะไรผิดหรือมีอะไรที่ดีกว่า เมื่อใดที่อ่านเสร็จหนึ่งเรื่อง คิดว่าตัวเองเข้าใจแล้ว ก็ลองลงรหัสเขียนเป็นโปรแกรมจริงด้วยตนเองโดยไม่เปิดหนังสือ เพราะเชื่อเถิดว่าเขียนโปรแกรมแล้วก็ต้องผิด ผิดก็ต้องแก้ไข จะแก้ไขได้ก็ต้องรู้สาเหตุ จะหาสาเหตุได้ก็ต้องทำความเข้าใจแนวคิดการออกแบบกับตัวโปรแกรมที่เขียน การเขียนโปรแกรมจริงจึงเป็นตัวตอกย้ำความเข้าใจและทำให้เกิดความเข้าใจมากขึ้นด้วย

หนังสือเล่มนี้นำเสนอเนื้อหาโครงสร้างข้อมูลขั้นพื้นฐาน โดยเน้นการเขียนโปรแกรมจริงที่พยายามเขียนให้สั้นและสวย เสริมด้วยการวิเคราะห์ประสิทธิภาพ และตามด้วยตัวอย่างการประยุกต์ในงานหลายหลาก ด้วยความต้องการที่จะนำเสนอเป็นโปรแกรมจริง จึงต้องเลือกภาษาการโปรแกรมแล้วเขียนให้เห็นจริง ผู้เขียนขอเลือกภาษาจาวา (Java) ด้วยความทันสมัย, ความนิยมทั้งจากภาคการศึกษาและภาคอุตสาหกรรม, ความสามารถของการทำงานเชิงวัตถุ, และเป็นภาษาที่เรียนกันมาในวิชาการเขียนโปรแกรมเบื้องต้น จึงเรียกหนังสือนี้ว่าเป็น “ฉบับวาจาจาวา” ‡

นอกจากจะนำเสนอเนื้อหาทางโครงสร้างข้อมูลแล้ว หนังสือเล่มนี้ยังแทรกเสริมแนวคิดการออกแบบเชิงวัตถุ (object-oriented design) และแบบอย่างการออกแบบ (design patterns) ในตัวโปรแกรมที่นำเสนอด้วย เพราะการออกแบบโครงสร้างข้อมูลต่าง ๆ ล้วนอาศัยคุณสมบัติของการออกแบบเชิงวัตถุและแบบอย่างการออกแบบ ไม่ว่าจะเป็นการห่อหุ้มปกปิดข้อมูลภายในโครงสร้าง (encapsulation) การมองที่เก็บข้อมูลในหลากหลายบทบาท (polymorphism) การสร้างประเภทข้อมูลใหม่จากประเภทเดิมด้วยการสร้างคลาสลูก (subclassing) หรือสร้างโดยนำอ็อบเจกต์เก่ามาประกอบเป็นของใหม่แล้วสั่งทำงานแทน (composition & delegation) การใช้แนวคิดการทำฟังก์ชันให้เป็นอ็อบเจกต์ (functor) การใช้ตัวแจงย้ำ (iterator) เพื่อแจกแจงข้อมูลภายในให้ผู้อื่นใช้ และอื่น ๆ อีกมากมาย ซึ่งล้วนเป็นโอกาสอันเหมาะสมที่จะนำเสนอแนวคิดเหล่านี้ในขณะที่นำเสนอเรื่องโครงสร้างข้อมูล โดยไม่จำเป็นต้องลงในรายละเอียดมาก

ผู้เขียนสอนวิชาโครงสร้างข้อมูลมาตั้งแต่ปี พ.ศ. 2534 แต่เพิ่งมาเขียนเป็นตำราครบเล่มได้หลังจากสอนไป 15 ปี ผู้เขียนใคร่ขอขอบคุณโครงการสนับสนุนการเขียนตำรา/หนังสือ ของคณาจารย์คณะวิศวกรรมศาสตร์ ที่ให้การสนับสนุน ขอขอบคุณภาควิชาวิศวกรรมคอมพิวเตอร์ จุฬาลงกรณ์

---

‡ เมื่อใดที่มีการใช้เรื่องจุกจิกของตัวภาษาจาวาในตัวโปรแกรม จะอธิบายเสริมเพิ่มเติมในกรอบที่มีตัวการ์ตูนถือถ้วยกาแฟกำกับคำอธิบาย

มหาวิทยาลัย ที่สนับสนุนวัสดุ ครุภัณฑ์ และโอกาส ขอขอบคุณสำนักพิมพ์แห่งจุฬาฯ ที่รับจัดพิมพ์และเผยแพร่ได้อย่างราบรื่น และขอขอบคุณสำนักนโยบายและแผนการอุดมศึกษา สำนักงานคณะกรรมการการอุดมศึกษา กระทรวงศึกษาธิการ ที่ได้สนับสนุนการผลิตสื่อบรรยายบทเรียน บทเรียนนี้เหมาะสำหรับการศึกษาด้วยตนเอง ครอบคลุมเนื้อหาหลักที่นำเสนอในหนังสือ พร้อมแบบทดสอบประเมินตนเองระหว่างการเรียน [§] [**]

สำหรับผู้สนใจเอกสารอื่น ๆ เพิ่มเติม (เช่น แผ่นใส, โปรแกรม, รายการแก้ไขข้อผิดพลาด, เนื้อหาเพิ่มเติมอื่น ๆ เป็นต้น) สามารถหาได้ที่

http://www.cp.eng.chula.ac.th/~somchai/books

สมชาย ประสิทธิ์จูตระกูล
ภาควิชาวิศวกรรมคอมพิวเตอร์
จุฬาลงกรณ์มหาวิทยาลัย
somchaip@chula.ac.th

๕ ธันวาคม ๒๕๔๘

---

§ ผู้สนใจสามารถเข้าชมบทเรียนวิชาโครงสร้างข้อมูลได้ที่ http://www.cp.eng.chula.ac.th/~somchai/ULearn

** หนังสือเล่มนี้ใช้ชุดแบบอักษร Angsana New, Browallia New, Cordia New, Tahoma, Times New Roman, และ Courier New นอกจากนี้ยังใช้ชุดแบบอักษร "**เลย์อิจิมหานิยม**" (www.f0nt.com) สำหรับคำอธิบายประกอบโปรแกรมต่าง ๆ ซึ่งผู้เขียนขอขอบคุณผู้ออกแบบ ณ ที่นี้ด้วย

สารบัญ

13 การเรียงลำดับข้อมูล 301

บรรณานุกรม 337

ดัชนี 339

1

บทนำ

บทนี้จะมาให้คำตอบว่า โครงสร้างข้อมูลคืออะไร มีประโยชน์อย่างไร ทำไมถึงสำคัญมากขนาดเป็นองค์ความรู้บังคับที่ต้องศึกษาและเข้าใจในวิทยาการคอมพิวเตอร์

การจัดเก็บและจัดการข้อมูล

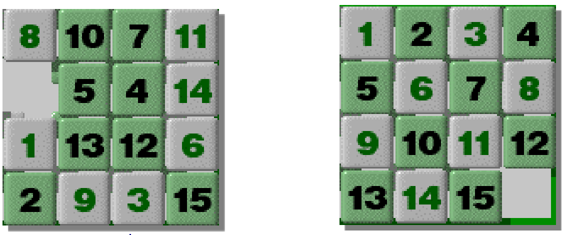
หน้าที่หลักของเครื่องคอมพิวเตอร์ก็คือการประมวลผลข้อมูล ข้อมูลอาจจะเป็นทะเบียนประชากรของประเทศ ประวัติคนไข้ในโรงพยาบาล รูปภาพหรือภาพยนตร์จากกล้องดิจิทัล เสียงคำสนทนาทางโทรศัพท์ และอื่น ๆ อีกมาก ลองคิดดูว่า ถ้าต้องการเขียนเกมคอมพิวเตอร์ประเภทมีพระเอกวิ่งผจญภัยไปตามด่านต่าง ๆ เพื่อช่วยเจ้าหญิง (ดูรูปที่ 1-1) ตัวละครต่าง ๆ ในเกมก็เป็นข้อมูล ฉากหลังก็เป็นข้อมูล แผนที่การผจญภัยก็เป็นข้อมูล สิ่งของและศัตรูต่าง ๆ ก็เป็นข้อมูล คะแนนที่สะสมและพลังของตัวละครในเกมระหว่างการผจญภัยก็เป็นข้อมูล เราจะจัดเก็บข้อมูลเหล่านี้อย่างไรในโปรแกรม เพื่อให้สามารถนำข้อมูลเหล่านี้ไปจัดการได้อย่างมีประสิทธิภาพ

โครงสร้างข้อมูลเป็นองค์ความรู้หนึ่งทางวิทยาการคอมพิวเตอร์ที่ว่าด้วยการจัดเก็บและจัดการข้อมูลให้ตรงตามวัตถุประสงค์ของการใช้งาน จัดเก็บข้อมูลให้ประหยัดปริมาณหน่วยความจำ และจัดการข้อมูลให้ได้อย่างรวดเร็ว นอกจากนี้การจัดเก็บและจัดการข้อมูลควรเป็นกระบวนการที่ทำความเข้าใจได้ง่ายไม่ซับซ้อนจนเกินไป ขอนำเสนอด้วยตัวอย่างเพื่อแสดงให้เห็นจริงว่า โครงสร้างข้อมูลมีผลต่อประสิทธิภาพการทำงานของโปรแกรมมากน้อยเพียงไร

รูปที่ 1-1 ตัวอย่างเกมคอมพิวเตอร์

ปริศนา 15

หวังว่าผู้อ่านคงเคยเล่นเกมปริศนา 15 (15 puzzle) กันมาก่อน (รูปที่ 1-2) เครื่องมือที่ใช้เล่นเป็นแผ่นพลาสติกรูปสี่เหลี่ยมจัตุรัส ภายในประกอบด้วยแผ่นพลาสติกสี่เหลี่ยมจัตุรัสย่อยเล็ก ๆ จำนวน 15 แผ่น วางเรียงกันเป็นตาราง 4×4 โดยมีช่องว่างหนึ่งช่องอยู่ภายใน แผ่นสี่เหลี่ยมเล็กแต่ละแผ่นมีตัวเลขกำกับตั้งแต่ 1 ถึง 15 จุดประสงค์ของเกมก็คือ ให้เลื่อนแผ่นสี่เหลี่ยมภายในไปมา (ตามแนวนอนหรือแนวตั้ง) เพื่อให้ได้แผ่นสี่เหลี่ยมเหล่านี้ เรียงเป็นระเบียบไล่ตั้งแต่ 1 ถึง 15 (จากซ้ายไปขวา จากบนลงล่าง) ดังตัวอย่างในรูปที่ 1-2 เริ่มจากรูปทางซ้าย ให้หาวิธีเลื่อนจนได้ดังรูปขวา



รูปที่ 1-2 แผ่นตารางของเกมปริศนา 15

เราจะมาเขียนโปรแกรมให้เครื่องคอมพิวเตอร์ช่วยบอกวิธีการเลื่อนแผ่นพลาสติกเล็ก ๆ เพื่อไปสู่จุดหมายที่ต้องการ โดยจะขอใช้กลวิธีพื้นฐานเพื่อแก้ปริศนาในลักษณะนี้ที่เรียกง่าย ๆ ว่า "วิธีการลุยทุกรูปแบบ" คือเนื่องจากเครื่องคอมพิวเตอร์ทำงานเร็วและมีหน่วยความจำมาก จึงอาศัยคุณสมบัติข้อนี้ในการให้เครื่องลองเลื่อนแผ่นพลาสติกในทุก ๆ หนทาง พร้อมทั้งจำผลลัพธ์ไว้ด้วย หากพบว่าผลลัพธ์ที่เลื่อนได้มีหมายเลขเรียงสวยงามตามต้องการ ก็หยุดลุยได้ เพราะได้พบวิธีการแก้ปริศนาที่ได้รับแล้ว

ขอเริ่มด้วยคลาส PuzzleBoard สิ่งที่ผลิตได้จากคลาสนี้คือตารางหมายเลข (ดังตัวอย่างในรูปที่ 1-2 จากนี้ไปขอเรียกสั้น ๆ ว่า *ตาราง*) มีสมาชิกที่เป็นข้อมูลและบริการที่สำคัญ ๆ ดังแสดงในรหัสที่ 1-1

```
class PuzzleBoard {
  private byte table[][];
  private byte rowB, colB;
  private PuzzleBoard prev;

  public PuzzleBoard moveBlank(int dir) {...}
  public boolean isAnswer() {...}
  public void show() {...}
  ...
}
```

รหัสที่ 1-1 โครงของคลาส PuzzleBoard แสดงข้อมูลภายในและเมท็อดที่ให้บริการ

PuzzleBoard มีเมท็อดหลักคือ moveBlank(int) ซึ่งมีหน้าที่ผลิตตารางใหม่ที่เป็นผลจากการเลื่อนช่องว่างไปตามทิศทางที่กำหนดให้ ขอบอกก่อนว่า แทนที่เราจะให้บริการเลื่อนหมายเลขใดไปทางซ้าย หรือขวา หรือบน หรือล่าง เราจะกลับความคิดเล็กน้อยแล้วบอกว่า ต้องการเลื่อนช่องว่าง (ที่มีเพียงช่องเดียวในตาราง) ไปในทิศทางตรงข้ามที่ทำให้หมายเลขที่เกี่ยวข้องต้องถูกเลื่อนมาแทนช่องว่างนั้น นอกจากนี้ PuzzleBoard ยังมี isAnswer() ซึ่งตรวจสอบว่า ตัวตารางมีหมายเลขเรียงกันตามที่ต้องการแล้วหรือยัง และมี show() ไว้แสดงลักษณะของตารางออกทางจอภาพ

PuzzleBoard มีสมาชิกข้อมูลที่เป็นแถวลำดับสองมิติ (table) เก็บหมายเลขต่าง ๆ ของตาราง มีตำแหน่งแถว (rowB) และคอลัมน์ (colB) ของช่องว่างในตาราง และมีสมาชิกที่สำคัญอีกตัวหนึ่งชื่อ prev มีไว้เก็บตารางก่อนหน้าที่ได้เรียกให้ moveBlank เพื่อทำให้เกิดตารางนี้ เช่น คำสั่ง b2 = b1.moveBlank(1) จะสร้างตารางใหม่ที่ได้มาจากการเลื่อนช่องว่างของ b1 ลงด้านล่าง (กำหนดให้ 1 แทนการเลื่อนลง) โดยตัวตาราง b1 ยังเหมือนเดิม จะได้ว่า b2.prev คือ b1 เพราะ b2 ได้มาจากการสั่งเลื่อนช่องว่างใน b1

คราวนี้เรามาสนใจกระบวนการเลื่อนทุกรูปแบบเพื่อหาคำตอบว่าทำอย่างไร ? รหัสที่ 1-2 แสดงเมท็อด solve ซึ่งรับตารางปริศนาตั้งต้นเป็นพารามิเตอร์ชื่อ b เริ่มด้วยการขอที่เก็บข้อมูลชื่อว่า queue เสมือนเป็น “แถวคอย” มีไว้เก็บตารางต่าง ๆ ที่จะถูกผลิตออกมาระหว่างการทำงาน ArrayQueue เป็นคลาสของโครงสร้างข้อมูลชนิดหนึ่ง ให้บริการเพิ่มข้อมูลที่ท้ายแถว และลบข้อมูลตัวที่อยู่หัวแถว จึงมีลักษณะการเก็บข้อมูลเหมือนการเข้าแถว หลังจากสร้าง queue เสร็จก็นำตารางเริ่มต้นที่ได้รับเข้าเก็บในแถวเป็นตัวแรก (ด้วยคำสั่ง enqueue) แล้วเข้าวงวนที่ทำงานตราบเท่าที่

queue ยังมีข้อมูลอยู่ การทำงานในวงวนเริ่มด้วยการดึงตารางจากหัวแถวของ queue ออกมา (ด้วยคำสั่ง dequeue) แล้วสั่งให้เลื่อนช่องว่างด้วย b.moveBlank(d) ไปในแต่ละทิศทาง (อาศัยคำสั่ง for แจกแจง d = 0, 1, 2, 3 แทนทิศทั้งสี่) ถ้าเลื่อนได้ moveBlank จะคืนตารางใหม่ ซึ่งมีค่าไม่ใช่ null ก็ตรวจสอบเล็กน้อยว่า ตารางที่ได้มาเป็นคำตอบหรือไม่ (โดยเรียก isAnswer()) ถ้าใช่ ก็คืนตารางคำตอบกลับไปเลย ถ้ายังไม่ใช่ ก็เพิ่มตารางใหม่ใน queue แล้ววนกลับไปทำงานในรอบต่อไป เมื่อใดการทำงานหลุดจากวงวนเพราะ queue ไม่มีข้อมูลเหลือ ก็แสดงว่า ไม่มีวิธีเลื่อนไปเป็นคำตอบที่ต้องการได้

```
public static PuzzleBoard solve(PuzzleBoard b) {
  ArrayQueue queue = new ArrayQueue();
  queue.enqueue(b);
  while ( !queue.isEmpty() ) {
    b = queue.dequeue();
    for (int d = 0; d < 4; d++) {
      PuzzleBoard b2 = b.moveBlank(d);
      if (b2 != null) {
        if ( b2.isAnswer() ) return b2;
        queue.add(b2);
      }
    }
  }
  return null;
}
```

ขอแถวคอยไว้เก็บตารางที่รอการเลื่อนช่องว่าง

ลองเลื่อนทั้งสี่ทิศ

เลื่อนแล้วพบคำตอบก็จบ

เลื่อนแล้วไม่ใช่คำตอบก็เก็บเข้าแถวคอย รอไปเลื่อนช่องว่างต่อไป

รหัสที่ 1-2 เมท็อด solve ที่ใช้แก้ปริศนา 15

เมื่อได้ตารางคำตอบแล้ว เราสามารถหาขั้นตอนการเลื่อนจากตารางตั้งต้นที่ได้รับ ไปสู่คำตอบได้ด้วยการวิ่งไล่ย้อนกลับทางข้อมูล prev ที่เก็บในแต่ละตาราง ดังรหัสที่ 1-3

```
public static void showSolution(PuzzleBoard b) {
  if ( b != null) {
    showSolution(b.prev);
    b.show();
  }
}
```

ต้องการแสดงตารางตั้งต้นถึง b

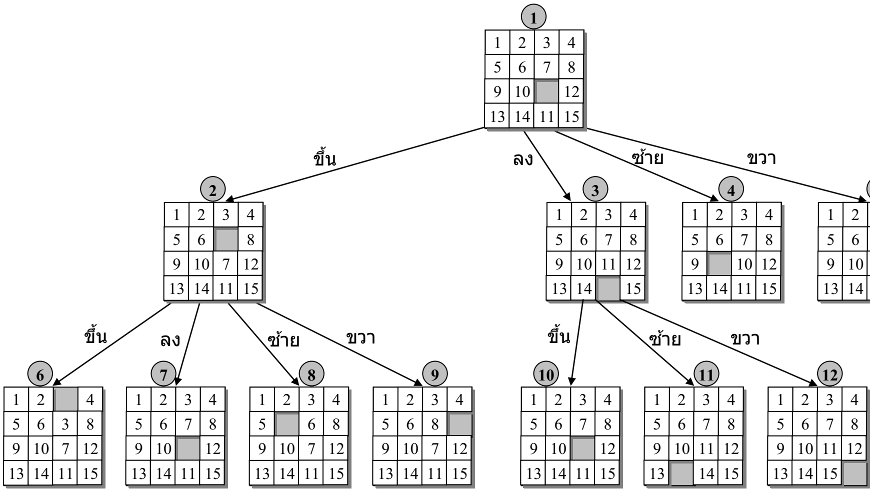
ไปแสดงตารางตั้งต้นถึงก่อนหน้า b ก่อน

แล้วค่อยมาแสดงตาราง b

รหัสที่ 1-3 เมท็อด showSolution มีหน้าที่แสดงการเปลี่ยนตารางจนได้คำตอบ

ขอแสดงตัวอย่างการทำงานในกรณีง่าย ๆ ดังรูปที่ 1-3 เริ่มจากการนำตารางเริ่มต้นที่รับ (ตารางบนสุดหมายเลข 1) เก็บเข้าแถวคอยเป็นตัวเดียวตัวแรก เมื่อเข้าวงวนการลบตารางจากแถวคอยจะได้ตารางหมายเลข 1 ลองเลื่อนช่องว่างของตารางหมายเลข 1 ได้ทั้งสี่ทิศ ผลิตได้สี่ตาราง (หมายเลข 2, 3, 4, และ 5) ตารางทั้งสี่ก็ถูกนำใส่แถวคอย เข้าแถวเรียงกันไป 2, 3, 4, และ 5 แล้ววนกลับไปทำงานที่ while คราวนี้การดึงตารางจากแถวคอยย่อมได้ตารางหมายเลข 2 ก็ผลิตได้อีก 4 ตาราง (6, 7, 8, และ

9) นำไปเพิ่มในแถวคอย ตอนนี้แถวคอยจะมีตาราง 3, 4, 5, 6, 7, 8, 9 เข้าแถวโดยตาราง 3 อยู่ที่หัวแถว จากนั้นวนกลับไปทำงานที่ `while` ถึงรอบนี้การลบข้อมูลที่หัวแถวจะได้ตารางหมายเลข 3 ซึ่งเมื่อผลิตตารางใหม่ด้วยการเลื่อนช่องว่าง จะพบว่า ตารางหมายเลข 12 ที่ได้จากการเลื่อนช่องว่างของตารางหมายเลข 3 ในทิศทางขวา คือคำตอบ ก็เป็นอันจบการค้นคำตอบ



รูปที่ 1-3 ตัวอย่างการลุยแก้ปัญหาปริศนา 15

เมื่อได้ตารางหมายเลข 12 เป็นคำตอบ ก็ย่อมรู้วิธีการเลื่อน โดยวิ่งตารางย้อนกลับไปด้านบน ได้ลำดับของตารางที่นำไปสู่คำตอบคือเริ่มตาราง 1 เลื่อนช่องว่างลง ได้เป็นตาราง 3 ตามด้วยการเลื่อนช่องว่างไปทางขวา ก็จะได้คำตอบ

อยากให้ผู้อ่านสังเกตรูปที่ 1-3 ว่า มีอะไรที่แสดงให้เห็นถึงความซ้ำซ้อนบ้าง ลองดูที่ตารางหมายเลข 1, 7 และ 10 พบว่า ทั้งสามตารางเหมือนกัน ตาราง 7 ได้มาจากการเลื่อนช่องว่างของตาราง 1 ขึ้นแล้วก็ลง จึงได้ตารางเดิม ส่วนตาราง 10 ได้จากการเลื่อนช่องว่างของตาราง 1 ลงแล้วก็ขึ้น ในกรณีที่ตารางเริ่มต้นซับซ้อนกว่าที่แสดงในรูป ความซ้ำซ้อนในการผลิตตารางในลักษณะที่กล่าวมาจะทำให้เสียเวลามากเกินความจำเป็นในการหาคำตอบ เราสามารถหลีกเลี่ยงเหตุการณ์ดังกล่าวได้ ด้วยการจำตารางทั้งหมดที่เคยผลิตมาตั้งแต่เริ่มทำงาน แล้วตรวจสอบความซ้ำซ้อนก่อนเพิ่มตารางใหม่ที่ผลิตเข้าแถวคอย

เราอาศัยที่เก็บข้อมูลอีกตัวชื่อ `set` (ของคลาส `ArraySet`) ทำหน้าที่เสมือนเซตเก็บข้อมูล (ดูรหัสที่ 1-4) ที่มีเมท็อด `add` ให้เพิ่มข้อมูล และ `contains` เพื่อค้นหาว่า มีข้อมูลที่กำหนดให้เก็บอยู่หรือไม่ ทุกครั้งที่มีการเพิ่มตารางใหม่เข้า `queue` ก็เพิ่มตารางนั้นเข้า `set` ด้วย โดยก่อนจะเพิ่ม

ตาราง b2 ก็ต้องถาม set.contains(b2) ซึ่งหมายความว่า set เก็บตาราง b2 อยู่หรือไม่ ถ้าไม่มี แสดงว่า b2 คือตารางใหม่ที่ไม่เคยพบมาก่อน จึงต้องเพิ่มใน queue และต้องเพิ่มใน set ด้วย

```
public static PuzzleBoard solve(PuzzleBoard b) {
  ArraySet set = new ArraySet();
  ArrayQueue queue = new ArrayQueue();
  queue.enqueue(b);  set.add(b);
  while ( !queue.isEmpty() ) {
    b = queue.dequeue();
    for (int d = 0; d < 4; d++) {
      PuzzleBoard b2 = b.moveBlank(d);
      if (b2 != null) {
        if ( b2.isAnswer() ) return b2;
        if ( ! set.contains(b2) )  {
          queue.enqueue(b2);  set.add(b2);
        }
      }
    }
  }
  return null;
}
```

ขอที่เก็บข้อมูลแบบเซตเพื่อป้องกันการผลิตตารางซ้ำซ้อน

จะเก็บตารางที่ผลิตใหม่ใส่แถวคอย ก็เมื่อเป็นตารางที่ไม่เคยพบมาก่อน

รหัสที่ 1-4 เมท็อด solve ที่หลีกเลี่ยงการผลิตตารางซ้ำซ้อนด้วยเซต

จากกระบวนการค้นคำตอบของปริศนา 15 ที่ได้นำเสนอมา มีข้อมูลที่เกี่ยวข้องหลายประเภท เริ่มตั้งแต่ข้อมูลขาเข้าของปัญหา (พารามิเตอร์ของ solve) คือตัวตาราง ที่เราแทนด้วยคลาส PuzzleBoard (และก็ใช้เป็นผลลัพธ์ด้วย) ตัวตารางมีโครงสร้างที่จัดเก็บข้อมูลภายในเพื่อรองรับการจัดการที่เรากำหนดไว้คือ moveBlank, isAnswer, และ show นอกจากนี้เราต้องมีข้อมูลเสริมที่ใช้ประกอบการแก้ปัญหาซึ่งคือตัวแถวคอยและเซต แถวคอยมีโครงสร้างที่จัดเก็บและจัดการข้อมูลภายในเพื่อให้บริการเพิ่มข้อมูลที่ท้ายแถวและลบข้อมูลที่หัวแถว ในขณะที่เซตมีโครงสร้างที่จัดเก็บและจัดการข้อมูลภายในเพื่อให้บริการเพิ่มและค้นข้อมูล เราไม่ได้นำเสนอรายละเอียดของโครงสร้างของแถวคอยและเซต ซึ่งคือคลาส ArrayQueue และ ArraySet เลย เพราะนั่นคือรายละเอียดที่จะต้องอธิบายกันในบทถัด ๆ ไป แต่จะขอบอกไว้ก่อนตอนนี้ว่า มีวิธีสร้างแถวคอยและเซตได้หลากหลายวิธี แต่ละแบบมีข้อดีข้อเสียแตกต่างกันไป ผู้ออกแบบต้องเลือกใช้ให้เหมาะสม ตัวอย่างเช่น ผู้เขียนได้ลองเปรียบเทียบเวลาการทำงานของ solve เมื่อสร้างเซตด้วยโครงสร้างข้อมูล 4 ประเภท (ที่จะได้ศึกษากัน) คือ ArraySet, BSTSet, AVLSet, และ HashSet เพื่อหาคำตอบของปริศนา 15 ที่มีตารางเริ่มต้นต่างกันสามแบบ (มีความซับซ้อนของตารางเริ่มต้นต่างกัน) ได้ผลการทดลองดังแสดงในตารางที่ 1-1

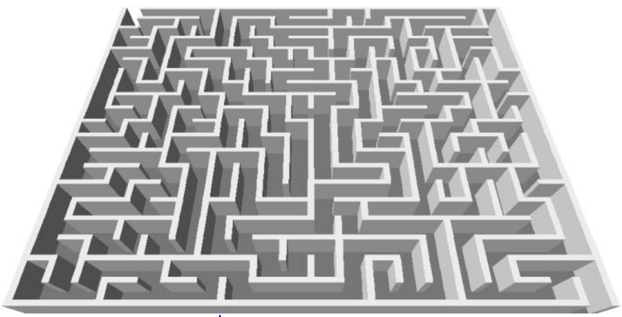
ตารางที่ 1-1 ผลการทดลองเพื่อวัดผลของการใช้โครงสร้างข้อมูลกับเวลาการทำงาน

ตารางเริ่มต้น	จำนวนตารางที่ผลิต	เวลาการทำงาน (วินาที)			
		ArraySet	BSTSet	AVLSet	HashSet
แบบที่ 1	552	0.03	0.02	0.04	0.05
แบบที่ 2	5242	1.94	0.22	0.18	0.12
แบบที่ 3	132049	1819.6	7.08	5.71	2.56

ผลการทดลองข้างบนนี้แสดงให้เห็นว่า การสร้างเซตด้วยโครงสร้างข้อมูลที่ต่างกันจะส่งผลต่อเวลาการทำงานอย่างเด่นชัด ผู้อ่านอาจสงสัยว่า `HashSet` ดีที่สุด แล้วมีเหตุผลอะไรที่ต้องไปสนใจแบบอื่นเล่า ต้องขอบอกว่า `HashSet` มันเหมาะมากกับการใช้งานใน `solve` ก็จริง แต่ก็อาจไม่เหมาะกับงานในลักษณะอื่น เพราะโครงสร้างข้อมูลแต่ละแบบนั้นมีจุดเด่นจุดด้อยขึ้นกับลักษณะการใช้งาน จึงเป็นหน้าที่ของผู้ออกแบบระบบและนักเขียนโปรแกรมที่ต้องเลือกใช้ หรือออกแบบโครงสร้างข้อมูลให้เหมาะสมกับลักษณะของงาน

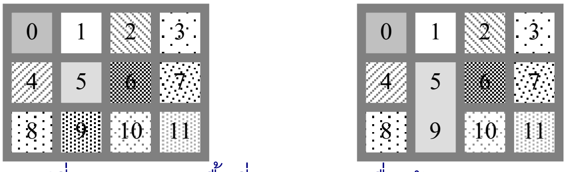
การสร้างเขาวงกต

มาดูกันอีกสักตัวอย่าง ถ้าเราต้องการเขียนเกมที่เกี่ยวกับการวิ่งไล่อะไรสักอย่างในเขาวงกต (รูปที่ 1-4) และเราอยากได้โปรแกรมที่สร้างเขาวงกตแบบสุ่ม ๆ ที่ประกันว่า มีเส้นทางเดินจากทุก ๆ บริเวณในเขาวงกตถึงที่อื่น ๆ จะเขียนโปรแกรมนี้อย่างไร ?



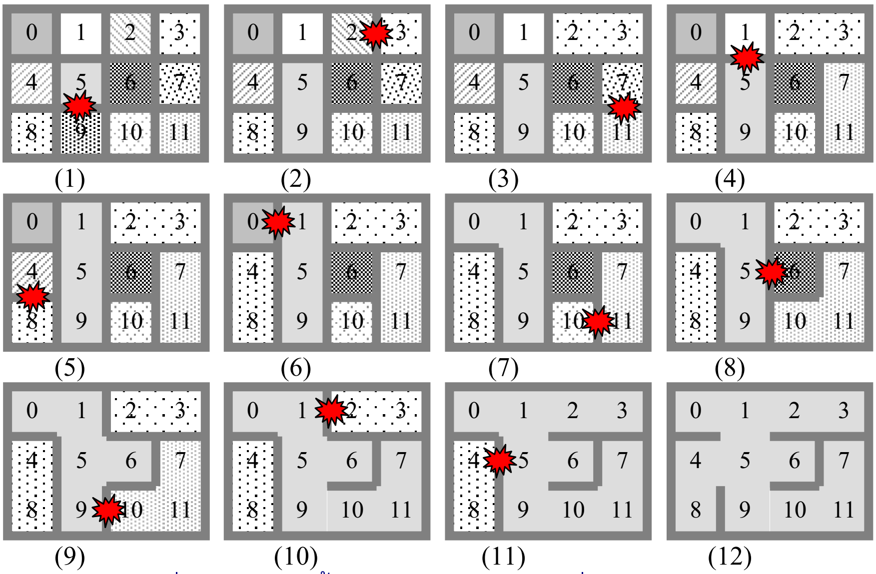
รูปที่ 1-4 ตัวอย่างเขาวงกต

เริ่มด้วยพื้นที่ขนาด $p \times q$ มีกำแพงแบ่งพื้นที่ออกเป็นบริเวณปิด จำนวน pq บริเวณ แต่ละบริเวณมีหมายเลขกำกับและมีสีพื้นต่างกัน (ดูรูปซ้ายของรูปที่ 1-5 เป็นตัวอย่าง) สุ่มเลือกกำแพงออกมาทุบทิ้งไปเรื่อย ๆ เมื่อทุบกำแพงใดทิ้ง ก็ต้องทำให้บริเวณที่ติดกับกำแพงทั้งสองมีสีพื้นเดียวกัน เพื่อแสดงให้เห็นว่า ทั้งสองบริเวณนั้นเชื่อมถึงกันแล้ว ตัวอย่างเช่น การทุบกำแพงที่กั้นบริเวณหมายเลข 5 และ 9 ในรูปที่ 1-5 ทางซ้ายทิ้ง ทำให้บริเวณทั้งสองเชื่อมถึงกัน จึงมีสีพื้นเหมือนกัน ดังรูปขวา



รูปที่ 1-5 ตัวอย่างพื้นที่ขนาด 3x4 เพื่อสร้างเขาวงกต

เพื่อให้เราเลือกทุบกำแพงเท่าที่จำเป็นเท่านั้น จะทุบเฉพาะกำแพงที่บริเวณสองด้านของกำแพงมีสีพื้นต่างกัน เพราะเป็นการเชื่อมบริเวณที่ยังต่อไม่ถึงกันเข้าด้วยกัน ขอตั้งข้อสังเกตว่า ถ้าเริ่มต้นมีบริเวณปิดอยู่ n บริเวณ เราต้องทุบกำแพงออกเป็นจำนวนอย่างน้อย $n - 1$ ครั้ง เมื่อทุบกำแพงได้ครบจำนวนดังกล่าวแล้ว จะได้ทุก ๆ บริเวณมีสีพื้นเดียวกันหมด แสดงว่า มีทางเดินจากทุกบริเวณถึงกันหมด รูปที่ 1-6 แสดงตัวอย่างขั้นตอนการทุบกำแพงเพื่อสร้างเขาวงกต

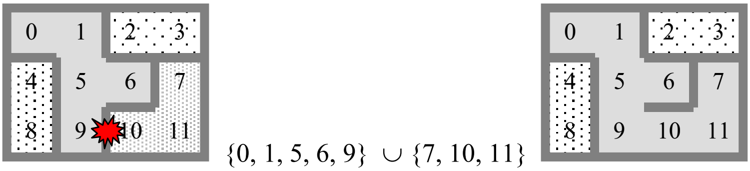


รูปที่ 1-6 ตัวอย่างขั้นตอนการสุ่มทุบกำแพงเพื่อสร้างเขาวงกต

กลุ่มเซตไร้ตัวร่วม

ขอละรายละเอียดของการแทนพื้นที่ กำแพง และบริเวณให้เป็นแบบฝึกหัดของผู้อ่าน เพราะสิ่งที่ต้องการเน้นในหัวข้อนี้ก็คือโครงสร้างข้อมูลตัวหนึ่งที่เรียกว่า *กลุ่มเซตไร้ตัวร่วม* (disjoint sets) ซึ่งใช้ในกระบวนการทุบกำแพง เพื่อหลีกเลี่ยงการทุบกำแพงที่มีบริเวณทั้งสองด้านของกำแพงเชื่อมต่อกัน กำหนดให้หมายเลขของบริเวณที่มีสีพื้นเดียวกันให้อยู่ในเซตเดียวกัน ดังนั้นตอนเริ่มต้นมีบริเวณที่แยกจากกัน n บริเวณ ก็มีทั้งหมด n เซต $\{0\}, \{1\}, \{2\}, ..., \{n - 1\}$ เมื่อใดมีการทุบกำแพง ก็จะนำเซตที่แทนบริเวณทั้งสองด้านของกำแพงนั้นมายูเนี่ยนกัน เป็นการรวมบริเวณทั้งสองเข้าด้วยกัน ดังตัวอย่าง

ในรูปที่ 1-7 เราต้องการทุบกำแพงที่กั้นบริเวณ 9 และ 10 ออก ก็คือการนำเซตที่ 9 เป็นสมาชิก มายูเนียนกับเซตที่ 10 เป็นสมาชิก



รูปที่ 1-7 ตัวอย่างการยูเนี่ยนของเซตของบริเวณทั้งสองด้านของกำแพงที่ถูกทุบ

```
void removeWalls(Wall[] walls, int numRooms) {
  DisjointSets sets = new DisjointSets(numRooms);
  shuffle(walls);
  for (int i=0; i<walls.length; i++) {
    int s1 = sets.find(walls[i].getRoom1());
    int s2 = sets.find(walls[i].getRoom2());
    if (s1 != s2) {
      sets.union(s1, s2);
      walls[i].setEnabled(false);
    }
  }
}
```

หาเซตของบริเวณทั้งสองด้านของกำแพง

ถ้าสองด้านเป็นคนละพวก ก็รวมเข้าด้วยกัน

แทนการทำลายกำแพง

รหัสที่ 1-5 เมท็อด `removeWalls` มีหน้าที่ทุบกำแพงให้เหลือเป็นเขาวงกต

รหัสที่ 1-5 แสดงเมท็อด `removeWalls` ซึ่งทำหน้าที่ทำลายกำแพงแบบสุ่มเพื่อให้ทุกบริเวณมีทางเดินถึงกันหมด เมท็อดนี้รับ `walls` ที่เป็นแถวลำดับของข้อมูลประเภท `Wall` แทนกำแพงทั้งหมด และ `numRooms` ที่ระบุจำนวนบริเวณทั้งหมด เริ่มด้วยการสร้างกลุ่มเซตไร้ตัวร่วม `sets` จากคลาส `DisjointSets` ที่ภายในเก็บ {0}, {1}, ... {`numRooms` – 1} เพื่อแทนบริเวณต่าง ๆ ตอนเริ่มต้นมีสีพื้นต่างกันหมด (อย่าลืมว่า เราใช้เซตแทนสีพื้น หมายความว่า ถ้า a และ b อยู่ในเซตเดียวกัน เป็นการแทนว่า บริเวณ a และ b มีสีพื้นเดียวกัน แสดงว่า มีทางเชื่อมต่อจาก a กับ b) หลังสร้างกลุ่มเซตเสร็จ ก็จัดการสุ่มสลับลำดับของกำแพงใน `walls` ด้วย `shuffle(walls)` (เพื่อจะได้จำลองการเลือกทุบกำแพงแบบสุ่ม) แล้วเข้าสู่วงวนหยิบหมายเลขของบริเวณทั้งสองของกำแพงด้วย `walls[i].getRoom1()` และ `walls[i].getRoom2()` เพื่อส่งไปหาว่า บริเวณนี้อยู่ในเซตหมายเลขอะไรด้วย `sets.find` ถ้าหมายเลขเซตทั้งสอง (`s1` และ `s2`) มีค่าไม่เท่ากัน แสดงว่าบริเวณทั้งสองไม่ต่อกัน ก็ยูเนียนเซตทั้งสองด้วย `sets.union(s1, s2)` จากนั้นทุบกำแพง `wall[i]` ทิ้งด้วยคำสั่ง `walls[i].setEnabled(false)`

จากนี้ไปจะนำเสนอวิธีการสร้างคลาส `DisjointSets` ซึ่งแทนการจัดเก็บกลุ่มเซตไร้ตัวร่วม ให้สมาชิกของกลุ่มเซตนี้เป็นเลขจำนวนเต็มตั้งแต่ 0 ถึง $n - 1$ โดยที่ n เป็นจำนวนที่ต้องกำหนด

ขณะที่สร้างกลุ่มเซต บริการที่ต้องมีคือ ตัวสร้าง (constructor) ที่เริ่มให้สมาชิกแต่ละตัวอยู่ในเซตที่แตกต่างกันหมด มีเมท็อด `find(e)` เพื่อหาว่า `e` อยู่ในเซตหมายเลขอะไร และ `union(s1, s2)` เพื่อยูเนียนเซตหมายเลข `s1` และ `s2` เข้าด้วยกัน ดังแสดงในรหัสที่ 1-6

```
public class DisjointSets {
  public DisjointSets(int n) { ... }
  public int find(int e) { ... }
  public void union(int s1, int s2) { ... }
}
```

รหัสที่ 1-6 คลาส `DisjointSets` ที่มีบริการ `find` และ `union`

หลังจากกำหนดความต้องการแล้ว ภาระถัดไปก็คือการออกแบบการจัดเก็บกลุ่มเซต และการจัดการในเมท็อดต่าง ๆ อย่างมีประสิทธิภาพ ขอนำเสนอให้เห็นจริงสองแบบเพื่อการเปรียบเทียบ เริ่มด้วยแบบแรกก่อน เราใช้แถวลำดับของจำนวนเต็มชื่อว่า `s` โดยที่ `s[e]` เก็บหมายเลขเซตที่ `e` เป็นสมาชิกอยู่ รูปที่ 1-8 แสดงตัวอย่างของแถวลำดับ `s` สองลักษณะที่แทนกลุ่มของเซต {0}, {1,2,3,5,6}, และ {4,7,8} ให้สังเกตว่า เราแทนกลุ่มเซตได้หลายลักษณะ ตราบเท่าที่ `a` และ `b` อยู่ในเซตเดียวกันก็ต่อเมื่อ `s[a]` เท่ากับ `s[b]`

	0	1	2	3	4	5	6	7	8
s	0	2	2	2	7	2	2	7	7

	0	1	2	3	4	5	6	7	8
s	0	3	3	3	4	3	3	4	4

รูปที่ 1-8 ตัวอย่างการแทนกลุ่มเซตไร้ตัวร่วม {0}, {1, 2, 3, 5, 6}, {4, 7, 8}

```
public class DisjointSets {
  private int[] s;

  public DisjointSets(int n) {
    s = new int[n];
    for (int e = 0; e < n; e++)
      s[e] = e;
  }
  public int find(int e) {
    return s[e];
  }
  public void union(int s1, int s2) {
    for (int e = 0; e < s.length; e++) {
      if (s[e] == s1) s[e] = s2;
    }
  }
}
```

s[e] เก็บหมายเลขเซตของ e

เริ่มต้นให้เป็น {0},{1},...,{n-1}
คือให้แต่ละ e อยู่ในเซตหมายเลข e

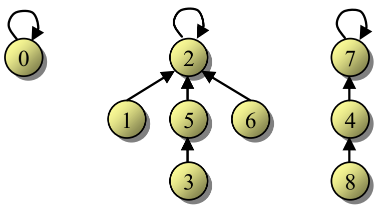
หมายเลขเซตของ e ก็คือ s[e]

เปลี่ยนทุกตัวใน s ที่มีค่าเป็น s1 ให้เป็น s2

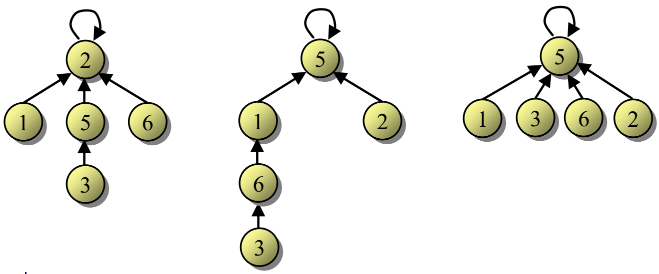
รหัสที่ 1-7 คลาส `DisjointSets` สร้างโดยใช้แถวลำดับเก็บหมายเลขเซตของสมาชิก

ด้วยวิธีนี้ find(e) ทำงานง่ายมาก คือเพียงแค่คืน s[e] ก็เป็นอันเสร็จ ส่วนการ union(s1,s2) ต้องเปลี่ยนช่องต่าง ๆ ใน s ที่มีค่า s1 ให้เป็น s2 (หรือจะเปลี่ยนในทางกลับกันก็ได้ คือเปลี่ยนทุกช่องใน s ที่มีค่า s2 ให้เป็น s1) สำหรับตัวสร้างนั้นก็เพียงแต่ตั้งค่าให้แต่ละ e มี s[e] เก็บ e ก็ทำให้สมาชิกทุกตัวอยู่ในเซตต่างกัน สรุปได้เป็นคลาสที่สมบูรณ์ดังรหัสที่ 1-7 ที่สั้นกะทัดรัด ไม่ซับซ้อน ประหยัดเนื้อที่ ใช้เพียงแค่แถวลำดับเดียวที่มีจำนวนช่องเท่ากับจำนวนสมาชิก ตัวสร้างใช้เวลาการทำงานซึ่งแปรตามจำนวนสมาชิก find ใช้เวลาการทำงานคงตัวเสมอ ไม่ขึ้นกับว่าถามสมาชิกตัวไหน หรือมีจำนวนสมาชิกมากเพียงใด ส่วน union ใช้เวลาแปรตามจำนวนสมาชิกทั้งหมด เพราะต้องไล่ดูทีละช่องในแถวลำดับเพื่อเปลี่ยนหมายเลขเซต สิ่งที่น่าท้าทายสำหรับผู้ออกแบบโครงสร้างข้อมูลก็คือ มีวิธีจัดเก็บและจัดการแบบอื่นไหม ที่ใช้ปริมาณเนื้อที่ไม่มาก แต่ทำงานได้เร็วกว่า

มาออกแบบใหม่กันอีกวิธี คราวนี้เราแทนแต่ละเซตด้วยต้นไม้ หนึ่งเซตหนึ่งต้น สมาชิกแต่ละตัวในเซตคือปมของต้นไม้ รูปที่ 1-9 แสดงตัวอย่างของต้นไม้ 3 ต้นที่แทนเซต {0}, {1,2,3,5,6}, และ {4,7,8} แต่ละปมมีตัวระบุตำแหน่งของปมพ่อ (เส้นในรูปมีลูกศรชี้จากปมลูกไปยังปมพ่อ) ปมที่เป็นรากของต้นไม้ก็คือปมที่มีปมพ่อคือปมตัวเอง รูปที่ 1-9 มีต้นไม้ 3 ต้น แทนเซต 3 เซต ปม 0, 2, และ 7 คือรากของต้นไม้แต่ละต้น ขอใช้หมายเลขของปมรากเป็นหมายเลขเซตที่ต้นไม้นั้นแทน เนื่องจากไม่มีข้อจำกัดในเรื่องรูปร่างของต้นไม้ ดังนั้นเซตหนึ่งเซตสามารถแทนได้ด้วยต้นไม้หลากหลายลักษณะดังตัวอย่างในรูปที่ 1-10

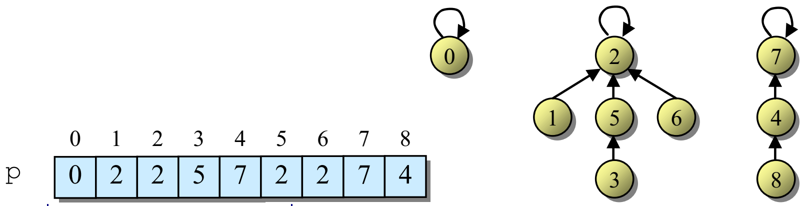


รูปที่ 1-9 ตัวอย่างการแทนกลุ่มเซตไร้ตัวร่วม {0}, {1, 2, 3, 5, 6}, {4, 7, 8}



รูปที่ 1-10 ตัวอย่างการแทนเซต {1, 2, 3, 5, 6} ด้วยต้นไม้สามลักษณะ

ด้วยวิธีการแทนเซตด้วยต้นไม้ ตัวสร้างของกลุ่มเซตไร้ตัวร่วมที่มีสมาชิก n ตัวก็คือการสร้างต้นไม้ n ต้น แต่ละต้นมีแต่ราก (ต้นหมายเลข 0 ถึง $n-1$) การ `find(e)` ก็คือการหารากของต้นไม้ที่มี `e` เป็นปมภายใน และการ `union(s1,s2)` คือการนำต้นไม้ที่มี `s1` และ `s2` เป็นรากมารวมเป็นต้นเดียว ถึงตอนนี้อาจยังไม่ค่อยชัดเจนเท่าไรนักว่า จะทำ `find` และ `union` อย่างไร ขอเริ่มด้วยวิธีจัดเก็บต้นไม้ต่าง ๆ ก่อน เราใช้แถวลำดับของจำนวนเต็ม `p` เก็บต้นไม้ต่าง ๆ ให้ `p[e]` เก็บหมายเลขปมพ่อของปม `e` ดังตัวอย่างในรูปที่ 1-11 ดูที่เซต {4, 7, 8} พ่อของปม 8 คือ 4 (`p[8]=4`) พ่อของ 4 คือ 7 (`p[4]=7`) และพ่อของ 7 คือ 7 (`p[7]=7`) ซึ่งคือราก ดังนั้น 7 แทนหมายเลขของเซตนี้



รูปที่ 1-11 การจัดเก็บต้นไม้ที่แทน {0}, {1, 2, 3, 5, 6}, {4, 7, 8} ด้วยแถวลำดับ

จากวิธีการจัดเก็บต้นไม้ด้วยแถวลำดับทำให้การ `union` กระทำได้ง่ายมาก คือถ้าเราต้องการ `union(s1,s2)` ก็เพียงจับรากของต้นไม้ต้นหนึ่งไปเป็นลูกของรากของต้นไม้อีกต้น เช่น จับราก `s1` ไปเป็นลูกของ `s2` ด้วยคำสั่ง `p[s1]=s2` หรือจะทำในทางกลับกันคือจับราก `s2` ไปเป็นลูกของราก `s1` ซึ่งก็คือ `p[s2]=s1` สำหรับการ `find(e)` นั้นก็ต้องเข้าวงวนเลื้อยจากปม `e` ขึ้นไปยังปมพ่อ ขึ้นไปเรื่อย ๆ จนพบปมราก ก็จะได้หมายเลขเซต สรุปเป็นคลาสที่สมบูรณ์ดังรหัสที่ 1-8

```
public class DisjointSets {
  private int[] p;

  public DisjointSets(int n) {
    p = new int[n];
    for(int e = 0; e < n; e++)
      p[e] = e;
  }
  public int find(int e) {
    while( p[e] != e) {
      e = p[e];
    }
    return e;
  }
  public void union(int s1, int s2) {
    p[s1] = s2;
  }
}
```

p[e] คือหมายเลขปมพ่อของ e

เริ่มต้นให้เป็น {0}, {1}, ..., {n-1} หนึ่งต้นหนึ่งปม

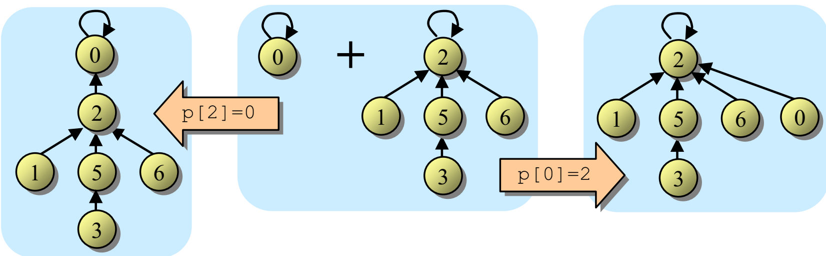
จาก e วิ่งขึ้นไปหาปมพ่อ เรื่อย ๆ จนพบราก

ให้รากของเซต s1 ไปเป็นลูกของรากของเซต s2

รหัสที่ 1-8 คลาส `DisjointSets` สร้างโดยใช้แถวลำดับเก็บต้นไม้ที่แทนเซต

คลาสนี้มีตัวสร้างที่ใช้เวลาการทำงานแปรตาม *n* ซึ่งคือจำนวนสมาชิกทั้งหมด union ใช้เวลาการทำงานคงตัวไม่ขึ้นกับจำนวนเซต หรือจำนวนสมาชิก ส่วน find(e) นั้นมีการทำงานเป็นวงวนซึ่งหมุนเป็นจำนวนรอบที่แปรตามจำนวนเส้นจาก e ขึ้นไปจนถึงรากของต้นไม้ที่ e อยู่ ประสิทธิภาพการทำงานจึงขึ้นอยู่ลักษณะของต้นไม้ ลองกลับไปดูรูปที่ 1-10 ซึ่งแสดงให้เห็นว่า เราแทนเซตหนึ่งเซตด้วยต้นไม้ได้หลากหลายแบบ ในรูปนี้ต้นไม้ทางขวาสุดย่อมเป็นลักษณะของต้นไม้ที่ทำให้ find ใช้เวลาน้อยสุด (เพราะเป็นต้นไม้ที่เตี้ยสุด) คำถามที่น่าสนใจก็คือ เราจะต้องทำอย่างไรจึงได้ต้นไม้ที่เตี้ย ๆ

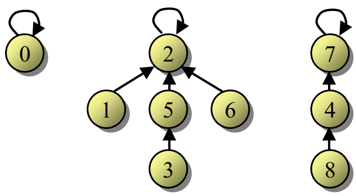
ต้นไม้จะมีรูปร่างอย่างไร ก็ขึ้นอยู่กับการทำงานใน union เพราะต้นไม้ค่อย ๆ โตขึ้นจากการ union จากที่ได้อธิบายมาว่า union(s1,s2) ทำได้ง่าย ๆ ด้วยการจับรากของต้นหนึ่งไปเป็นลูกของรากอีกต้นหนึ่ง ซึ่งกระทำได้สองแบบคือ p[s1]=s2 หรือ p[s2]=s1 แล้วเราควรทำแบบใดแน่นอนว่า เราควรทำแบบที่ทำให้ได้ต้นไม้เตี้ย ๆ นั่นคือควรนำรากของต้นไม้ต้นเตี้ยไปเป็นลูกของรากของต้นไม้ต้นสูง รูปที่ 1-12 แสดงตัวอย่างการรวมต้นไม้สองต้นไม้ซึ่งทำได้สองวิธี แต่ได้ความสูงของต้นไม้ต่างกัน ซึ่งเราควรเลือกวิธีการรวมที่ได้ต้นที่เตี้ยกว่า เพราะย่อมส่งผลให้ find ทำงานได้เร็วขึ้น



รูปที่ 1-12 การรวมต้นไม้ทางซ้ายได้ต้นไม้ที่สูงกว่าทางขวา

แล้วจะรู้ความสูงของต้นไม้ได้อย่างไร ? เราอาจใช้วิธีวิเคราะห์จากแถวลำดับ p ก็ได้ ซึ่งต้องเสียเวลาพอควร หรืออีกวิธีหนึ่งซึ่งทำได้ง่ายและเร็ว คือการจองแถวลำดับอีกแถวให้ชื่อว่า h มี *n* ช่อง โดยที่ h[e] เก็บความสูงของต้นไม้ที่รากคือปม e (ดูรูปที่ 1-13) เริ่มต้นในตัวสร้างก็ให้ความสูงของต้นไม้ทุกต้นเป็นศูนย์ (เพราะมีปมเดียว) ใน union ที่มีการรวมต้นไม้ก็นำ h มาช่วยตัดสินใจและปรับความสูงด้วยหลังการรวม ดังแสดงด้วยโปรแกรมที่ปรับปรุงใหม่ดังรหัสที่ 1-9

	0	1	2	3	4	5	6	7	8
p	0	2	2	5	7	2	2	7	4
h	0		2					2	



รูปที่ 1-13 การจัดเก็บต้นไม้ต่าง ๆ ด้วยแถวลำดับ ที่มีการจำความสูงด้วย

```
public class DisjointSets {
  private int[] p;
  private int[] h;

  public DisjointSets(int n) {
    p = new int[n]; h = new int[n];
    for(int e = 0; e < n; e++) {
      p[e] = e; h[e] = 0;
    }
  }
  public int find(int e) {
    while( p[e] != e)
      e = p[e];
    return e;
  }
  public void union(int s1, int s2) {
    if (h[s1] < h[s2])
      p[s1] = s2;
    else {
      p[s2] = s1;
      if (h[s1] == h[s2]) h[s1]++;
    }
  }
}
```

เพิ่มแถวลำดับ h เก็บความสูง
h[e] เก็บความสูงของต้นไม้ที่มี e เป็นราก

เริ่มต้นทุกต้นมีปมเดียว ความสูงจึงเป็น 0

find ไม่ใช้ h เลย

ถ้าต้น s1 เตี้ยกว่า นำ s1 ไปเป็นลูกของ s2

ถ้าต้น s1 ไม่เตี้ยกว่า นำ s2 ไปเป็นลูกของ s1

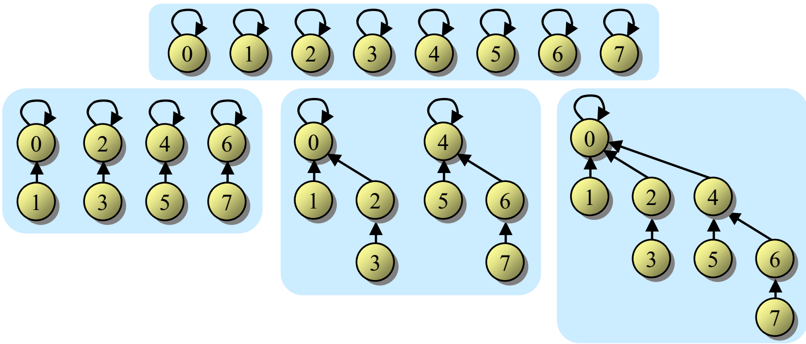
ถ้าต้นทั้งสองสูงเท่ากัน ให้ s2 เป็นลูกของ s1 และเพิ่มความสูงของ s1

รหัสที่ 1-9 คลาส `DisjointSets` ที่นำต้นไม้เตี้ยไปเป็นลูกของต้นไม้สูงในการยูเนียน

`union` อาศัยการเปรียบเทียบ `h[s1]` กับ `h[s2]` ถ้าต้น `s1` เตี้ยกว่า ก็เปลี่ยนพ่อของ `s1` ไปเป็น `s2` ไม่เช่นนั้นก็เปลี่ยนพ่อของ `s2` เป็น `s1` ในกรณีหลังนี้รวมกรณีที่ต้นไม้สูงเท่ากันด้วย ถ้าต้นไม้ทั้งสองสูงเท่ากัน การรวมต้นไม้ย่อมทำให้ต้นไม้ใหม่มีความสูงเพิ่มขึ้นอีก 1 ดังนั้นจึงต้องทำ `h[s1]++` ด้วย เพียงเท่านี้การรวมต้นไม้แบบใหม่ก็จะได้ต้นไม้ที่ไม่สูงผิดปกติ อีกทั้งการทำงานใน `union` ยังคงใช้เวลาคงตัว ไม่ขึ้นกับขนาดของเซต หรือจำนวนสมาชิกทั้งหมดแต่อย่างใด

ถึงตรงนี้ผู้อ่านอาจสงสัยว่า รู้ได้อย่างไรว่า จะได้ต้นไม้เตี้ย ขอนำเสนอลำดับการรวมเซตต่าง ๆ แบบแกล้งให้การ `union` ในรหัสที่ 1-9 ได้ต้นไม้สูงสุด ๆ วิธีแกล้งคือต้องให้การยูเนียนทุกครั้งเกิดการรวมต้นไม้ที่มีความสูงเท่ากัน ต้นใหม่จะได้สูงขึ้น ดูรูปที่ 1-14 ประกอบ เริ่มจากกลุ่มเซต {0}, {1}, ..., {7} เลือกต้นไม้สองต้นที่มีความสูงเท่ากันมารวมได้ด้วย `union(0,1)`, `union(2,3)`, `union(4,5)`, และ `union(6,7)` จากนั้นก็ใช้กลวิธีเดิมคือเลือกต้นไม้ที่สูงเท่ากันมารวมกันด้วย `union(0,2)`, `union(4,6)` ถึงตอนนี้จะเหลือเพียงสองต้นซึ่งก็สูงเท่ากันอีก ก็นำมา `union(0,4)` ครั้งสุดท้ายเหลือเพียงต้นเดียวซึ่งมีความสูงเป็น 3 สรุปจากตัวอย่างนี้คือเมื่อมีสมาชิกทั้งหมด 8 ตัว การยูเนียนที่ทำให้ได้ต้นไม้สูงสุดจะมีความสูงเป็น 3 ถ้าเพิ่มจำนวนสมาชิกเป็น 16 แล้วลองทำดูในลักษณะเดียวกันก็จะได้ความสูงของต้นไม้ท้ายสุดเป็น 4 ซึ่งสามารถพิสูจน์ได้ว่า ด้วยการ

รวมต้นไม้แบบอาศัยความสูงในรหัสที่ 1-9 จะได้ต้นไม้สูงไม่เกิน $\lfloor \log_2 n \rfloor$ เมื่อ n คือจำนวนสมาชิกทั้งหมดของกลุ่มเซต สรุปได้ว่า การ `find(e)` ย่อมหมุนในวงวนไม่เกิน $\lfloor \log_2 n \rfloor$ ครั้ง เพราะจำนวนรอบในการวิ่งจากปมจนถึงรากถูกกำหนดโดยความสูงของต้นไม้ สมมติว่า n = 1,000,000 จะได้ว่า การหมุนวนเพื่อเลื้อยขึ้นไปยังรากของ `find` นั้นจะทำไม่เกิน $\lfloor \log_2 1{,}000{,}000 \rfloor$ = 19 รอบเท่านั้น จึงถือได้ว่า เป็นโครงสร้างข้อมูลที่มีประสิทธิภาพดีมาก



รูปที่ 1-14 ตัวอย่างการยูเนียนที่ได้ต้นไม้สูงสุด

ถ้าเรายังไม่พอใจกับประสิทธิภาพที่ได้ ก็ต้องหาวิธีทำให้ต้นไม้เตี้ยลงไปอีก ได้มีผู้ออกแบบกลวิธีที่เรียกว่า *การบีบอัดวิถี* (path compression) ที่ทำให้ต้นไม้เตี้ยลงอีกระหว่างการ `find` ก่อนอื่นขอนำเสนอการเขียน `find` ในอีกแบบหนึ่งคือในรูปแบบเวียนเกิดดังรหัสที่ 1-10

```
    public int find(int e) {
      if (p[e] == e) return e;
      return find(p[e]);
    }
```

หมายเลขเซตของพ่อ ก็คือ หมายเลขเซตของเรา

รหัสที่ 1-10 เมท็อด `find` เขียนแบบ recursive

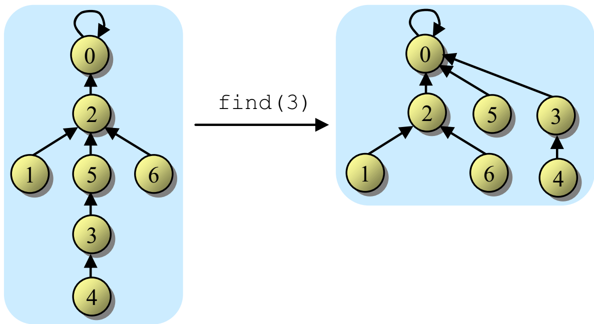
อ่านตีความได้ว่า เมื่อถามว่า `e` อยู่ในเซตหมายเลขอะไร ก็ตรวจสอบก่อนว่า `e` เป็นรากหรือไม่ ถ้าใช่ (ซึ่งคือกรณีที่ `p[e]` มีค่าเท่ากับ `e`) ก็คืนค่า `e` แต่ถ้าไม่ใช่ ก็ให้ถาม `find(p[e])` ซึ่งคือถามว่า พ่อของ `e` อยู่เซตหมายเลขอะไร `e` ก็อยู่หมายเลขเซตนั้นแหละ แล้วก็คืนค่ากลับไป เขียนแบบนี้ได้ผลเดียวกับการเขียนแบบใช้วงวน `while` จุดที่น่าสนใจคือถ้าเราเปลี่ยนคำสั่งท้ายสุดเป็น `return p[e]=find(p[e])` ได้ดังรหัสที่ 1-11

```
    public int find(int e) {
      if (p[e] == e) return e;
      return p[e] = find(p[e]);
    }
```

เปลี่ยนพ่อให้เป็นหมายเลขเซต ซึ่งคือรากนั่นเอง

รหัสที่ 1-11 เมท็อด `find` ที่มีการบีบอัดวิถี

ซึ่งหมายความว่า เมื่อไปถาม `find(p[e])` แล้วได้ผลเป็นหมายเลขรากของต้นไม้ การเติม `p[e]=` ก็หมายถึงให้เปลี่ยนพ่อของ `e` ไปเป็นหมายเลขรากของต้นไม้ที่หาได้แทน ทำให้ปมต่าง ๆ บนวิถีการเลื้อยขึ้นจาก `e` ถึงราก ถูกเปลี่ยนพ่อใหม่ ให้เป็นลูกของรากหมดทุกปม ดังตัวอย่างรูปที่ 1-15 เมื่อเรียก `find(3)` ในต้นไม้ทางซ้าย จะได้ว่า ปม 3, 5, และ 2 ล้วนเปลี่ยนพ่อเป็น 0 หมดได้เป็นรูปทางขวา ทำให้ต้นไม้เตี้ยลง เวลาการทำงานของ `find` ในอนาคต ก็จะลดลง



รูปที่ 1-15 ตัวอย่างการบีบอัดวิถีหลังจากเรียก `find(3)`

สรุป

ระบบงานที่มีขนาดใหญ่ย่อมต้องยุ่งเกี่ยวกับปริมาณข้อมูลที่มีจำนวนมาก การจัดโครงสร้างข้อมูลที่ดีส่งผลให้ระบบงานมีระเบียบ บำรุงรักษาและปรับปรุงง่าย ปัญหาก็มีน้อย ประสิทธิภาพการทำงานก็สูง การเขียนโปรแกรมต้องประมวลผลข้อมูล ควรถามตัวเองเสมอว่า ข้อมูลแต่ละประเภทที่เราต้องการนั้นเราต้องการไปทำไม นั่นคือต้องนิยามตัวดำเนินการต่าง ๆ ของข้อมูลด้วยการเขียนข้อกำหนดของข้อมูลแต่ละประเภทให้ชัดเจน จากนั้นถามตัวเองต่อว่า ประเภทข้อมูลที่เราต้องการนั้นมีคนอื่นเขาออกแบบไว้แล้วหรือไม่ ถ้ามี มีกี่แบบที่เราเลือกใช้ได้ แล้วควรเลือกใช้แบบใด ถ้าไม่มี หรือว่า มีแต่ไม่เหมาะเพราะอาจขาดบางคุณสมบัติ หรือเพราะอาจมีคุณสมบัติมากเกินไป ก็ต้องปรับปรุง หรือออกแบบกันใหม่ให้เหมาะสม การออกแบบโครงสร้างข้อมูลจึงประกอบด้วย

- การออกแบบโครงสร้างการจัดเก็บข้อมูล ที่ใช้ปริมาณหน่วยความจำน้อย ๆ
- การออกแบบวิธีการจัดการข้อมูลตามข้อกำหนดการใช้งาน ที่ทำงานอย่างรวดเร็ว

โดยทั่วไปภาระทั้งสองข้อข้างบนนี้อาจขัดแย้งกัน คือถ้าต้องการเก็บให้ประหยัดเนื้อที่ อาจใช้เวลาจัดการมาก แต่ถ้าเพิ่มเนื้อที่เพื่อเก็บข้อมูลเสริมอื่น ๆ ก็อาจลดเวลาการจัดการลง แต่ในบางครั้งก็พบว่าเราสามารถออกแบบโครงสร้างข้อมูลที่ทั้งประหยัดเนื้อที่จัดเก็บและลดเวลาจัดการได้ด้วย จึงเป็นหน้าที่ของผู้ออกแบบที่ต้องออกแบบและวิเคราะห์ผลไปพร้อมกันเพื่อให้ได้โครงสร้างที่เหมาะสมที่สุด

แบบฝึกหัด

1. จงเขียนคลาส `DisjointSets` ด้วยตนเอง โดยไม่ดูรายละเอียดในหนังสือ

2. จงตอบคำถามต่อไปนี้ที่เกี่ยวกับคลาส `PuzzleBoard` ของปริศนา 15

   2.1. `prev` ของตาราง `b` ซึ่งเป็นตารางเริ่มต้นของเมท็อด `solve` ต้องมีค่าเป็นอะไร

   2.2. ทำไมจึงให้แต่ละช่องของ `table` และตัวแปร `rowB` กับ `colB` เป็นแบบ `byte`

   2.3. `PuzzleBoard` ควรมีเมท็อดอะไรให้บริการเพิ่มอีกบ้าง

3. จงตอบคำถามต่อไปนี้ที่เกี่ยวกับคลาส `DisjointSets` ในรหัสที่ 1-9

   3.1. ถ้าเราเปลี่ยนแถวลำดับ `h` และ `p` จากที่เป็น `int[]` ให้เป็น `short[]` เพื่อประหยัดเนื้อที่จากแต่ละช่อง 4 ไบต์เหลือแค่ช่องละ 2 ไบต์ (และคงต้องเปลี่ยนที่อื่น ๆ ในคลาสด้วยอีกเล็กน้อยเพื่อเปลี่ยนจาก `int` เป็น `short`) จะทำให้คุณสมบัติของกลุ่มเซตไร้ตัวร่วมของเราด้อยลงมากไหม เมื่อเทียบกับของเดิม

   3.2. จากข้อที่แล้ว ถ้าเราเปลี่ยนเฉพาะ `int[] h` เป็น `byte[] h` (ไม่เปลี่ยนของ `p`) จะดีไหมในทางปฏิบัติ

   3.3. เขียนคลาสนี้ใหม่ โดยยุบแถวลำดับ `h` และ `p` ให้เหลือเพียงแถวเดียว (ข้อแนะนำ : ของเดิม `h` และ `p` เป็นแถวลำดับที่แต่ละช่องเป็นจำนวนเต็มไม่ติดลบ จึงน่าเสียดายที่ต้องจอง `int` ที่เก็บจำนวนลบได้แต่ไม่เก็บ)

   3.4. คลาสนี้ใช้ความสูงของต้นไม้เป็นตัวตัดสินทางเลือกในการรวมต้นไม้เมื่อต้องการยูเนี่ยน จงเขียนคลาสนี้ใหม่โดยใช้จำนวนปมในต้นไม้ (แทนความสูง) เป็นตัวตัดสินทางเลือกในการรวมต้นไม้ เพื่อให้ได้ต้นไม้ที่เตี้ย (ข้อแนะนำ : ลองคิดดูว่า ควรนำรากของต้นใหญ่ไปเป็นลูกของรากของต้นเล็ก หรือควรนำรากของต้นเล็กไปเป็นลูกของรากของต้นใหญ่)

   3.5. จะเกิดอะไรขึ้น ถ้าเราเปลี่ยนวงวน `while` ในเมท็อด `find` ให้เป็น

   ```
   while (p[e] != e) {
     p[e] = p[p[e]];
     e = p[e];
   }
   ```

   3.6. จงพิสูจน์ว่า การนำรากของต้นเตี้ยไปเป็นลูกของรากของต้นสูงจะทำให้ต้นไม้มีความสูงไม่เกิน $\lfloor \log_2 n \rfloor$

3.7. จงพิสูจน์ว่า การนำรากของต้นเล็กไปเป็นลูกของรากของต้นใหญ่จะทำให้ต้นไม้มีความสูงไม่เกิน $\lfloor \log_2 n \rfloor$

4. กำหนดให้ r คือตารางขนาด $n \times n$ ที่แสดงความสัมพันธ์การเป็นญาติกัน โดยถ้า $r[i][j]$ มีค่าเป็น 1 แสดงว่า i เป็นญาติกับ j จงนำเสนอการใช้กลุ่มเซตไร้ตัวร่วมเพื่อช่วยในการหาว่า มีอยู่กี่ตระกูลในตาราง r ที่ได้รับ (หมายเหตุ : คนในตระกูลต่างกันต้องไม่เป็นญาติกัน)

5. กำหนดให้มีเนื้อร้องเพลงไทยอยู่ 10,000 เพลง จงเขียนโปรแกรมที่ใช้ `Set` เพื่อหาว่า ในเนื้อร้องเหล่านี้มีคำที่แตกต่างกันอยู่กี่คำ (หมายเหตุ: เนื้อเพลงไทยสามารถหาได้จากแผ่นซีดีคาราโอเกะที่จำหน่ายในท้องตลาด ซึ่งภายในบรรจุเนื้อเพลง พร้อมทำนองในรูปแบบ midi จำนวนมาก นอกจากนี้ให้ศึกษาอินเทอร์เฟซ `Set` และคลาส `TreeSet` ในชุด `java.util` เพื่อเป็นที่เก็บคำ และศึกษา `BreakIterator` ในคลาส `java.text` เพื่อใช้ในการแยกข้อความไทยออกเป็นคำ)

6. ลองสืบค้นว่า ในคลังคลาสของจาวามีคลาสและอินเทอร์เฟซอะไรบ้างที่ใช้เก็บกลุ่มข้อมูล (ดูในชุด `java.util`)

2 การเก็บข้อมูลด้วยแถวลำดับ

คอลเล็กชัน (collection) คือลักษณะการจัดเก็บกลุ่มข้อมูล รองรับการจัดการเพียงแค่การเพิ่ม ลบ และค้นข้อมูล เสมือนเป็นถัง หรือเป็นถุงเก็บข้อมูล ไม่มีข้อกำหนดเรื่องระเบียบการจัดเก็บแต่อย่างไร สามารถเก็บข้อมูลซ้ำ ๆ กันได้ เป็นลักษณะการจัดเก็บที่ธรรมดา ง่าย แต่ใช้กันแพร่หลาย บทนี้จะอธิบายวิธีการสร้างคอลเล็กชันด้วยแถวลำดับ (`ArrayCollection`) ซึ่งเป็นโครงสร้างพื้นฐานที่นำไปสู่การออกแบบโครงสร้างข้อมูลประเภทอื่น ๆ

อินเทอร์เฟซ Collection

ขอเริ่มด้วยการอธิบายข้อกำหนดของคอลเล็กชันกันก่อน ซึ่งเขียนบรรยายด้วยอินเทอร์เฟซ (interface) ในภาษาจาวา ภายในอินเทอร์เฟซประกอบด้วยรายการของเมท็อด ที่ข้อมูลประเภทนั้นให้บริการ เป็นการบอกความสามารถของตัวข้อมูล โดยเขียนเฉพาะหัวของเมท็อด ไม่เขียนอธิบายรายละเอียดอื่น ๆ ดังแสดงในรหัสที่ 2-1

```
public interface Collection {
  public void add(Object e);
  public void remove(Object e);
  public boolean contains(Object e);
  public boolean isEmpty();
  public int size();
}
```

interface มีแต่หัวเมท็อดเท่านั้น

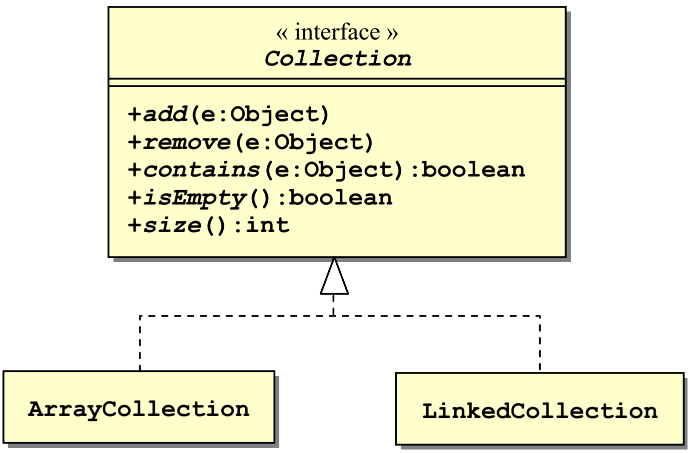
รหัสที่ 2-1 อินเทอร์เฟซ `Collection`

ตารางที่ 2-1 อธิบายหน้าที่การทำงานของแต่ละเมท็อดในอินเทอร์เฟซ `Collection` คลาสอะไรจะเป็น `Collection` ได้ (หรือที่เราเขียนในจาวาว่า `implements Collection`) ต้องมีรายละเอียดการทำงานของทั้ง 5 เมท็อดที่แสดง บทนี้และบทที่ 4 จะนำเสนอรายละเอียดการออกแบบ

คลาส `ArrayCollection` และ `LinkedCollection` ซึ่งมีบริการและพฤติกรรมตามตารางที่ 2-1 เราสามารถวาดความสัมพันธ์ของคลาสและอินเทอร์เฟซเหล่านี้ได้ดังแผนภาพในรูปที่ 2-1 เรียกแผนภาพนี้ว่า *แผนภาพคลาส* (class diagram) สี่เหลี่ยมแต่ละก้อนแทนคลาสหรืออินเทอร์เฟซ (ถ้าเป็นอินเทอร์เฟซก็มีคำว่า «interface» อยู่ที่หัว) เส้นประหัวสามเหลี่ยมขาวดังในรูปแสดงการที่คลาส implements หรือมีรายละเอียดของทุกเมท็อดที่ปรากฏในอินเทอร์เฟซ

ตารางที่ 2-1 หน้าที่ของเมท็อดต่าง ๆ ของอินเทอร์เฟซ Collection

เมท็อด	หน้าที่
`void add(Object e)`	เพิ่มอ็อบเจกต์ e ไว้ในคอลเล็กชัน
`void remove(Object e)`	หาอ็อบเจกต์ e แล้วลบออกจากคอลเล็กชัน (ถ้ามีซ้ำกันหลายตัวก็ลบออกเพียงตัวเดียว)
`boolean contains(Object e)`	ค้นว่า คอลเล็กชันนี้เก็บอ็อบเจกต์ e ไว้หรือไม่
`boolean isEmpty()`	ถามว่า คอลเล็กชันนี้ว่าง (คือไม่มีข้อมูล) หรือไม่
`int size()`	คืนจำนวนข้อมูลที่เก็บในคอลเล็กชันนี้



รูปที่ 2-1 แผนภาพคลาส `ArrayCollection` และ `LinkedCollection`

เพื่อให้เข้าใจวิธีใช้งานคอลเล็กชันมากขึ้น ขอนำเสนอโปรแกรมที่ตรวจสอบว่า แต่ละบรรทัดในแฟ้มข้อมูลชื่อ “data.txt” มีบรรทัดที่มีข้อความซ้ำกันหรือไม่ เช่น แฟ้มทางซ้ายในรูปที่ 2-2 มีข้อความ “สวัสดี” ซ้ำกันสองบรรทัด ส่วนแฟ้มทางขวาไม่มีการซ้ำกัน

```
Hello
สวัสดี
ลาก่อน
good bye
สวัสดี
```

```
Hello
สวัสดี
ลาก่อน
good bye
สวัสดีครับ
```

รูปที่ 2-2 แฟ้มข้อมูลตัวอย่าง ทางซ้ายมีบรรทัดที่มีข้อความซ้ำกัน ส่วนทางขวาไม่มี

```
01 import java.io.*;
02
03 public class Test {
04   public static void main(String[] args)
05                          throws IOException {
06     FileReader fr = new FileReader("data.txt");
07     BufferedReader br = new BufferedReader(fr);
08     Collection c = new ArrayCollection();
09     String line;
10     while ((line = br.readLine()) != null) {
11       if (c.contains(line))
12         System.out.println("ข้อความซ้ำ : " + line);
13       else
14         c.add(line);
15     }
16     System.out.println(c.size());
17   }
18 }
```

เปิดแฟ้มด้วย FileReader แล้วห่อด้วย BufferedReader ทำให้อ่านทีละบรรทัดได้

ขอคอลเล็กชันไว้เก็บข้อมูล

ถ้า c.contains(line) เป็นจริง แสดงว่าเคยอ่านสตริงนี้มาแล้ว

ถ้าเป็นข้อความใหม่ก็เพิ่มใส่ c

แสดงจำนวนสตริงใน c

รหัสที่ 2-2 โปรแกรมตรวจสอบบรรทัดที่มีข้อความซ้ำกันในแฟ้มด้วย `Collection`

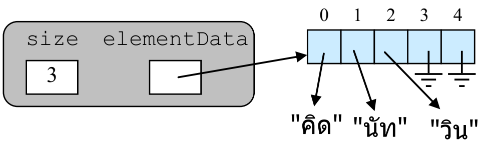
รหัสที่ 2-2 แสดงคลาส `Test` ซึ่งมีเมท็อด `main` เปิดแฟ้มชื่อ “data.txt” ด้วยการสร้าง `FileReader` ในบรรทัดที่ 6 แล้วส่งไปสร้าง `BufferedReader` ทำให้สามารถอ่านแฟ้มข้อความได้ทีละบรรทัดด้วยเมท็อด `readLine` ในบรรทัดที่ 10 ตามด้วยการสร้างคอลเล็กชันแบบ `ArrayCollection` ในบรรทัดที่ 8 ให้ชื่อว่า `c` แล้วเข้าวงวนเพื่ออ่านแฟ้มเข้ามาทีละบรรทัดเก็บในสตริง `line` เมื่อใดอ่านได้ `null` แสดงว่าหมดแฟ้ม เงื่อนไขในบรรทัดที่ 10 เป็นเท็จ ก็หลุดจากวงวน ภายในวงวนเรานำ `line` ไปตรวจสอบว่า มีเก็บอยู่ใน `c` หรือไม่ด้วย `c.contains(line)` ในบรรทัดที่ 11 ถ้าเป็นจริง แสดงว่ามีซ้ำ ก็แสดงข้อความให้ผู้ใช้ทราบ ถ้าหาไม่พบใน `c` ก็เพิ่ม `line` เข้าไปใน `c` ด้วย `c.add(line)` ในบรรทัดที่ 14 แล้วกลับขึ้นไปอ่านบรรทัดถัดไป เมื่อใดอ่านหมดแฟ้มแล้วก็ไปทำบรรทัดที่ 16 เพื่อแสดงขนาดของคอลเล็กชันด้วย `c.size()` จะเห็นได้ว่า การมีคอลเล็กชันให้ใช้ทำให้เขียนโปรแกรมและทำความเข้าใจตัวโปรแกรมได้ง่ายขึ้น

ArrayCollection

ขอนำเสนอวิธีการสร้างคอลเล็กชันด้วยแถวลำดับ ให้ชื่อคลาสว่า `ArrayCollection` คลาสนี้มีสมาชิกภายในสองตัวคือ `elementData` กับ `size` (ดูรหัสที่ 2-3) `elementData` เป็นแถวลำดับเก็บอ็อบเจกต์ในช่องที่ 0 ไปจนถึงช่องที่ `size-1` ข้อมูลใหม่ที่เพิ่มเข้ามาก็ถูกเก็บใส่ช่องถัดจากตัวหลังสุดไปเรื่อย ๆ ดังนั้น ตัวแปร `size` จึงทำหน้าที่เป็นทั้งตัวนับจำนวนอ็อบเจกต์ในคอลเล็กชันและตัวมันเองยังใช้เป็นเลขชี้ตำแหน่งของ `elementData` ที่จะใส่ข้อมูลใหม่อีกด้วย รูปที่ 2-3 แสดงตัวอย่างแถวลำดับที่มี 5 ช่อง เก็บข้อมูลอยู่จริง 3 ตัว (ช่องที่ 0 ถึง 2) ค่าของ `size` จึงมีค่าเป็น 3

```
public class ArrayCollection implements Collection {
  private Object[] elementData;
  private int      size;
  ...
```

รหัสที่ 2-3 `ArrayCollection` มีสมาชิก 2 ตัวคือ `elementData` กับ `size`



รูปที่ 2-3 ตัวอย่างแสดงสมาชิกของ ArrayCollection เมื่อตัวคอลเล็กชันเก็บข้อมูล 3 ตัว

สิ่งที่ได้อธิบายในย่อหน้าที่แล้ว คือการออกแบบระเบียบการจัดเก็บข้อมูล เราได้กำหนดประเภทของสมาชิกที่เป็นข้อมูล ความหมายและความสัมพันธ์ของสมาชิกต่าง ๆ หลังจากนี้ก็เริ่มออกแบบการจัดการ ซึ่งคือการทำงานของแต่ละเมท็อดที่ให้บริการ เริ่มที่ตัวสร้าง (constructor) ก่อนเพื่อกำหนดค่าต่าง ๆ ตอนเริ่มต้น แน่นอนว่า เมื่อเริ่มสร้างคอลเล็กชันใหม่ ก็ย่อมเป็นคอลเล็กชันว่าง ๆ ไม่มีข้อมูลเก็บอยู่เลย `size` มีค่าเป็น 0 ตามด้วยการจองแถวลำดับไว้สำหรับเก็บข้อมูล ดังแสดงในบรรทัดที่ 6 และ 7 ของรหัสที่ 2-4

ดูต่อกันที่เมท็อด `add` ก็เพียงนำอ็อบเจกต์ที่ได้รับไปใส่ไว้ในช่องที่ `size` ของแถวลำดับ แล้วก็เพิ่มขนาดไปอีกหนึ่ง และเพื่อให้ง่ายต่อการศึกษา เราจะไม่อนุญาตให้เก็บข้อมูลที่มีค่าเป็น `null` ในคอลเล็กชัน ดังนั้นหากมีการส่ง `null` มาเพิ่ม จะทำให้เกิดสิ่งผิดปกติ (exception) ที่บรรทัดที่ 10

ผู้อ่านลองนึกดูสักครู่หนึ่งว่า `add` ที่เขียนนี้มีข้อจำกัดอะไร ? บอกเลยก็ได้ว่า `add` นี้มีปัญหาตอนที่ `size` มีค่าเท่ากับ `elementData.length` คือถ้า `add` ถูกเรียกตอนที่คอลเล็กชันเก็บข้อมูลเป็นจำนวนเท่ากับขนาดของแถวลำดับที่จองไว้ ก็ย่อมเพิ่มข้อมูลไม่ได้ แล้วจะทำอย่างไร ? ขอเก็บปัญหานี้ไว้ก่อน แล้วจะบอกวิธีแก้ไขในภายหลัง

```
01 public class ArrayCollection implements Collection {
02   private Object[] elementData;
03   private int      size;
04
05   public ArrayCollection() {
06     elementData = new Object[100];
07     size = 0;
08   }
09   public void add(Object e) {
10     if(e == null) throw new IllegalArgumentException();
11     elementData[size++] = e;
12   }
13   public int size() {
14     return size;
15   }
16   public boolean isEmpty() {
17     return size == 0;
18   }
..   ...
```

เตรียมแถวลำดับไว้เก็บข้อมูล และให้ตอนนี้ยังไม่มีข้อมูลใด ๆ เก็บเลย

คอลเล็กชันแบบนี้ไม่เก็บ null ถ้าไม่ใช่ null ก็เก็บต่อท้าย

size เป็นตัวแปรเก็บจำนวนข้อมูล

คอลเล็กชันไม่มีข้อมูลเมื่อ size มีค่าเป็น 0

รหัสที่ 2-4 เมท็อดต่าง ๆ ของคลาส ArrayCollection

ต่อด้วยเมท็อด size และ isEmpty ซึ่งง่ายมาก เนื่องจากเรามีตัวแปร size ที่เก็บจำนวนข้อมูล ดังนั้นเมท็อด size เพียงแค่คืนค่าของตัวแปร size ส่วน isEmpty ก็คืนค่าจริง ถ้า size มีค่าเป็น 0 ไม่เช่นนั้นก็คืนค่าเท็จ นั่นคือการคืนผลของการเปรียบเทียบว่า size == 0 หรือไม่

```
19   public boolean contains(Object e) {
20     return indexOf(e) != -1;
21   }
22   public void remove(Object e) {
23     int i = indexOf(e);
24     if (i != -1) {
25       elementData[i] = elementData[--size];
26       elementData[size] = null;
27     }
28   }
29   private int indexOf(Object e) {
30     for (int i=0; i<size; i++)
31       if (elementData[i].equals(e)) return i;
32     return -1;
33   }
34 }
```

indexOf คืน -1 แสดงว่าหาไม่พบ

ใช้ indexOf หาตำแหน่งก่อน ถ้าหาพบก็ลบ โดยนำตัวท้ายมาทับตัวที่จะถูกลบ

เราย้ายตัวท้ายไปอยู่ตำแหน่งอื่นแล้ว ช่องท้ายก็ไม่ควรมีการอ้างอิงข้อมูล จึงใส่ null เพื่อตัดการอ้างอิงซะ

วิ่งหาตำแหน่งในแถวลำดับที่เก็บ e ไปเรื่อย ๆ ถ้าไม่พบก็คืน -1

รหัสที่ 2-5 เมท็อด contains และ remove อาศัยเมท็อด indexOf ในการค้น

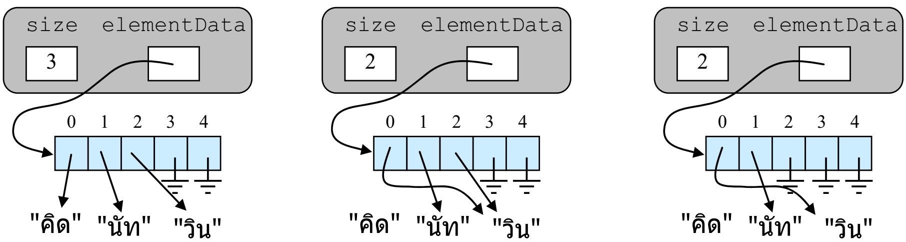
เหลืออีกสองเมท็อดที่ยังไม่นำเสนอคือ contains และ remove ทั้งคู่ต้องอาศัยการค้นหาข้อมูลที่ได้รับว่าอยู่ที่ใดในแถวลำดับ (รหัสที่ 2-5) ขอเขียนเมท็อดเสริมที่ใช้เฉพาะภายในคลาส (เป็น private) ชื่อ indexOf (บรรทัดที่ 29 ถึง 33) ทำงานโดยใช้วงวน for ค่อย ๆ เปรียบเทียบข้อมูล

ใน `elementData` กับ `e` ซึ่งคืออ็อบเจกต์ที่ได้รับ เริ่มตั้งแต่ช่องที่ 0 ไปจนถึงช่องที่ `size-1` พบว่าเท่ากันเมื่อใด ก็คืนหมายเลขช่องกลับทันที ถ้าหมุนในวงวนจนครบ หลุดจากวงวน แสดงว่า ค้นไม่พบ ก็คืน –1 การค้นในลักษณะเช่นนี้เรียกว่า *การค้นหาแบบลำดับ* (sequential search)

ขอให้สังเกตบรรทัดที่ 31 ซึ่งเปรียบเทียบอ็อบเจกต์สองตัว (`elementData[i]` กับ `e`) ว่ามีค่าเท่ากันหรือไม่นั้น เราใช้เมท็อด `equals` ในการเปรียบเทียบ เราไม่ใช้ `==` เพราะนั่นจะเป็นการตรวจสอบว่า `elementData[i]` และ `e` เป็นอ็อบเจกต์ตัวเดียวกันหรือไม่ `equals` เป็นเมท็อดมาตรฐานของคลาส `Object` ซึ่งเป็นบรรพบุรุษของทุก ๆ คลาสในจาวา ดังนั้นทุก ๆ อ็อบเจกต์ในจาวาจะมี `equals` ให้เรียกใช้ จึงเป็นข้อแนะนำของผู้ที่จะเขียนคลาสใหม่ว่า ควร `override` เมท็อด `equals` เพื่อใช้เปรียบเทียบความเท่ากันของอ็อบเจกต์ตามลักษณะการจัดเก็บข้อมูลของคลาสนั้น

ข้อสังเกตอีกข้อหนึ่งที่จุกจิกนิดหน่อยคือ แทนที่เราจะเขียน `elementData[i].equals(e)` ทำไมเราไม่เขียน `e.equals(elementData[i])` ซึ่งควรให้ผลเหมือนกัน ขอให้ดูว่า `elementData` กับ `e` ตัวไหนเราควบคุมค่าของมันได้ `elementData` เป็นข้อมูล `private` ของอ็อบเจกต์ การเปลี่ยนแปลงค่าต่าง ๆ ในแถวลำดับนี้เกิดขึ้นใน `add` และ `remove` ที่เราเป็นผู้เขียนเอง ส่วน `e` นั้นเป็นพารามิเตอร์ที่รับต่อมาจาก `contains` และ `remove` ตัวแปร `e` ที่เรารับมาก็อาจมีค่าเป็น `null` ได้ (ซึ่งอาจเป็นความตั้งใจของผู้เรียกเมท็อด หรืออาจเป็นเพราะข้อผิดพลาดในการเขียนโปรแกรม) ถ้า `e` เป็น `null` คำสั่ง `e.equals(elementData[i])` จะทำให้เกิด `exception` ในขณะที่ถ้าเป็นคำสั่ง `elementData[i].equals(e)` จะได้ผลเป็น `false` จึงต้องจำไว้ตอนนี้ว่า ช่องต่าง ๆ ที่มีข้อมูลใน `elementData` นั้นต้องควบคุมไม่ให้เป็น `null`

เมื่อมี `indexOf` ก็สามารถเขียน `contains` ได้ง่าย ๆ โดยการเรียก `indexOf` แล้วตรวจสอบผลที่ได้คืนมา ถ้ามีค่า –1 ก็แสดงว่า หาไม่พบ ให้คืน `false` ถ้าไม่ใช่ –1 ก็คืนค่า `true` (บรรทัดที่ 20 ในรหัสที่ 2-5) สำหรับการลบในเมท็อด `remove` ก็อาศัย `indexOf` ค้นหาช่องของแถวลำดับที่เก็บข้อมูลที่จะลบ ถ้า `indexOf` ค้นพบ นั่นคือคืนค่าที่ไม่ใช่ –1 ก็จัดการลบทิ้ง โดยนำข้อมูลตัวท้ายสุด คือข้อมูลในช่อง `elementData[size-1]` ย้ายมาวางแทนที่ตัวที่ต้องการลบ คือช่อง `elementData[i]` แล้วก็ลดขนาดของ `size` ลงหนึ่ง ดังแสดงในบรรทัดที่ 25 (ซึ่งเขียนแบบกะทัดรัดด้วยลดค่า `size` ลงหนึ่งก่อน ได้ค่าของตำแหน่งท้ายสุดนำไปใช้ได้พอดี) ตามด้วยการนำ `null` ไปใส่ในช่องท้าย รูปที่ 2-4 แสดงตัวอย่างการเปลี่ยนข้อมูลเมื่อมีการลบ



รูปที่ 2-4 ผลการทำงานเมื่อเรียก `remove("คิด")`

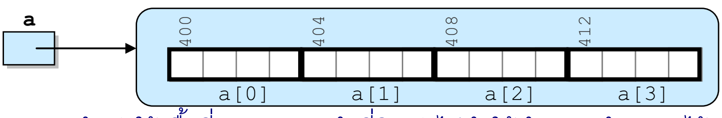
เหตุผลที่ต้องนำ null ไปใส่ในช่องของตัวท้าย ก็เพื่อตัดการอ้างอิงอ็อบเจกต์ที่ไม่จำเป็นออก ขอใช้ตัวอย่างในรูปที่ 2-4 เดิมรูปทางซ้ายเก็บ 3 อ็อบเจกต์ เมื่อเรียก remove("คิด") แทนการลบข้อมูลในช่องที่ 0 ทำได้ด้วยคำสั่ง elementData[0]=elementData[--size] ได้ดังรูปกลาง ให้สังเกตว่า สตริง "วิน" ตอนนี้มีทั้งช่อง 0 และ 2 อ้างอิงมันอยู่ ทั้ง ๆ ที่ช่อง 2 ไม่จำเป็นต้องอ้างอิงมันก็ได้ ก็เลยนำ null ไปใส่ในช่อง 2 ได้ดังรูปขวา

เมื่อใดอ็อบเจกต์ที่เราสร้างขึ้นในระบบ แล้วไม่มีตัวแปรใดอ้างอิงมันเลย ระบบจะถือว่า อ็อบเจกต์นั้นเป็น "ขยะ" หมายความว่า ระบบสามารถเรียกเนื้อที่นั้นคืนกลับมาใช้ประโยชน์ใหม่ได้ ถ้าสังเกตในรูปขวาของรูปที่ 2-4 จะพบว่า สตริง "คิด" ไม่มีตัวแปรใดอ้างอิงแล้ว ก็ถือได้ว่าเป็นขยะ การตัดการอ้างอิงอ็อบเจกต์ที่ไม่จำเป็นด้วย null ตามที่อธิบายมา จึงช่วยให้เราใช้หน่วยความจำได้อย่างคุ้มค่า

ผู้อ่านอาจสงสัยว่า ก็เมื่อกี้เพิ่งบอกเองว่า คอลเล็กชันไม่อนุญาตให้เก็บ null และเราต้องควบคุมไม่ให้ช่องใน elementData มีค่าเป็น null ไม่เช่นนั้นจะมีปัญหาในเมท็อด indexOf ที่มีการเรียก element[i].equals ขอย้ำอีกครั้งว่า ที่เราไม่อนุญาตให้เก็บ null นั้น เราไม่อนุญาตให้ผู้ใช้ส่ง null มาให้เก็บ แต่เราเองจะเก็บเอง ก็เป็นการภายใน สิ่งที่ต้องระวังคือป้องกันไม่ให้ข้อมูลของ elementData ตั้งแต่ช่อง 0 ถึงช่อง size-1 เป็น null

แถวลำดับขยายขนาดได้

แถวลำดับใช้เนื้อที่หน่วยความจำที่ติดกันไปตั้งแต่ช่องที่ 0 จนถึงช่องสุดท้าย จึงทำให้สามารถใช้ข้อมูลในช่องใด ๆ ของแถวลำดับได้อย่างรวดเร็ว ด้วยการคำนวณจากตำแหน่งเริ่มต้นของแถวลำดับ หมายเลขช่อง และประเภทข้อมูล เช่น ถ้าสร้างแถวลำดับจำนวนเต็ม 4 ช่อง ด้วยคำสั่ง a=new int[4] (ดูรูปที่ 2-5) ตัวแปร a ก็จะอ้างอิงไปยังหน่วยความจำที่เก็บแถวลำดับ ที่มีขนาดเท่ากับ int สี่ตัวติดกันไป (int หนึ่งตัวในจาวากินที่ 4 ไบต์) ถ้าตำแหน่งไบต์เริ่มต้นของแถวลำดับนี้อยู่ที่หมายเลข 400 การอ้างอิง a[2] ก็ย่อมเป็นตำแหน่งไบต์ที่ $400 + 2 \times 4 = 408$ ถึง 411 ของหน่วยความจำ การอ้างอิงช่องของแถวลำดับได้อย่างรวดเร็ว นับเป็นข้อดีของการใช้แถวลำดับเก็บข้อมูล



รูปที่ 2-5 แถวลำดับใช้เนื้อที่หน่วยความจำที่ติดกันไปทำให้คำนวณตำแหน่งได้รวดเร็ว

เมื่อเราเก็บข้อมูลครบทุกช่องในแถวลำดับแล้ว ก็ไม่สามารถเพิ่มข้อมูลได้อีก จาวาไม่มีคำสั่งขยายแถวลำดับ เราจึงต้องขยายขนาดเอง โดยจองแถวลำดับใหม่ให้ใหญ่กว่าเดิม ทำสำเนาข้อมูลจากชุดเก่าไปยังชุดใหม่ แล้วเปลี่ยนตัวแปรที่อ้างอิงแถวลำดับเดิมให้มาอ้างอิงแถวลำดับใหม่ ดังแสดงในเมท็อด ensureCapacity ของรหัสที่ 2-6 ซึ่งประกันว่า elementData จะมีขนาดอย่างน้อยเท่ากับขนาดที่ให้มา ถ้าขนาดใหม่ที่ให้มาไม่ใหญ่กว่าขนาดปัจจุบัน ก็ไม่ต้องทำอะไร แต่ถ้าขนาดใหม่

ใหญ่กว่า ก็ขยายแถวลำดับให้มีขนาดเป็นอย่างน้อยสองเท่าของของเดิม เมื่อมี ensureCapacity ก็สามารถปรับปรุง add ให้สมบูรณ์ได้ โดยก่อนจะเพิ่ม ให้เรียก ensureCapacity(size+1) เพื่อสร้างความมั่นใจว่า หลังเรียกแล้วต้องมีช่องหมายเลขที่ size ให้ใส่ข้อมูลใหม่ได้แน่ ๆ

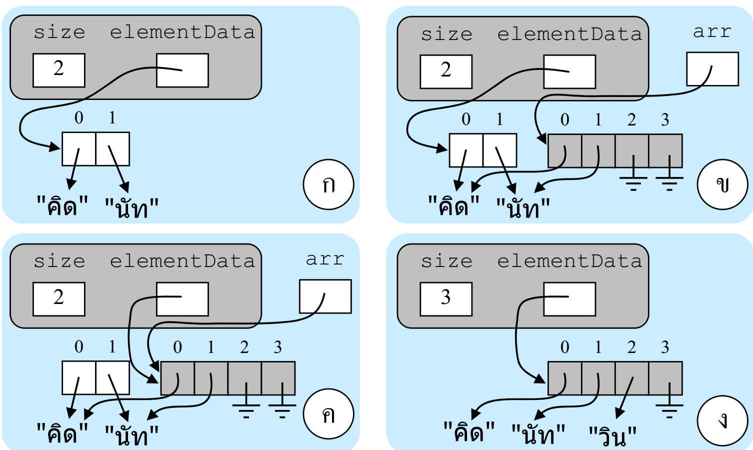
```
private void ensureCapacity(int capacity) {
  if (capacity > elementData.length) {
    int s = Math.max(capacity, 2*elementData.length);
    Object[] arr = new Object[s];
    for(int i = 0; i < size; i++)
      arr[i] = elementData[i];
    elementData = arr;
  }
}
public void add(Object e) {
  if(e == null) throw new IllegalArgumentException();
  ensureCapacity(size + 1);
  elementData[size++] = e;
}
```

ขยายแถวลำดับด้วยการจองแถวลำดับใหม่ใหญ่กว่าเดิม ทำสำเนาจากของเก่าไปของใหม่ ปิดท้ายด้วยการเปลี่ยน elementData มาอ้างอิงแถวลำดับใหม่

ให้ขยายแถวลำดับถ้าจำเป็น

รหัสที่ 2-6 การขยายขนาดของแถวลำดับในเมท็อด ensureCapacity

รูปที่ 2-6 แสดงการเพิ่มข้อมูลขณะที่แถวลำดับมีข้อมูลเต็ม รูป ก เป็นคอลเล็กชันที่มีข้อมูลสองตัว ใช้แถวลำดับสองช่อง เมื่อต้องการเพิ่ม "วิน" จะไปทำงานที่ ensureCapacity โดยต้องการขนาด 3 จึงจองแถวลำดับใหม่ (ขนาด 4) และทำสำเนา ได้ดังรูป ข เปลี่ยนให้ elementData อ้างอิงแถวลำดับใหม่ได้ดังรูป ค (ไม่มีใครอ้างอิงแถวลำดับเก่าอีกแล้ว ก็จะกลายเป็นขยะ) เป็นอันเสร็จการทำงานของ ensureCapacity กลับมายัง add เพื่อนำ "วิน" ใส่ในช่องที่ 2 ได้ดังรูป ง



รูปที่ 2-6 ตัวอย่างการเพิ่มข้อมูลเมื่อแถวลำดับมีข้อมูลเต็มทุกช่อง

สรุปได้คลาส ArrayCollection ที่สมบูรณ์ดังรหัสที่ 2-7

```
public class ArrayCollection implements Collection {
  private Object[] elementData;
  private int      size;

  public ArrayCollection() {
    elementData = new Object[1];
    size = 0;
  }
  public void add(Object e) {
    if(e == null) throw new IllegalArgumentException();
    ensureCapacity(size + 1);
    elementData[size++] = e;
  }
  public int size() {
    return size;
  }
  public boolean isEmpty() {
    return size == 0;
  }
  public boolean contains(Object e) {
    return indexOf(e) != -1;
  }
  public void remove(Object e) {
    int i = indexOf(e);
    if (i != -1) {
      elementData[i] = elementData[--size];
      elementData[size] = null;
    }
  }
  private int indexOf(Object e) {
    for (int i = 0; i < size; i++)
      if (elementData[i].equals(e)) return i;
    return -1;
  }
  private void ensureCapacity(int capacity) {
    if (capacity > elementData.length) {
      int s = Math.max(capacity, 2*elementData.length);
      Object[] arr = new Object[s];
      for(int i = 0; i < size; i++)
        arr[i] = elementData[i];
      elementData = arr;
    }
  }
}
```

ใช้แถวลำดับกับอีกหนึ่ง int เป็นข้อมูลภายใน

สร้างด้วยการจองตัวแถวลำดับและตั้งค่า size ให้เป็นศูนย์ เนื่องจากขยายขนาดได้ จึงจองแค่ 1 ช่องก็พอ

เพิ่มข้อมูล (ที่ไม่ใช่ null) ด้วยการนำข้อมูลไปต่อท้ายตัวหลังสุด

คืนตัวแปร size เพื่อบอกจำนวนข้อมูลในคอลเล็กชัน

คอลเล็กชันไม่มีข้อมูลเมื่อ size มีค่าเป็น 0

ใช้ indexOf หา ถ้าคืนกลับมาเป็น –1 แสดงว่าหาไม่พบ

ใช้ indexOf หาตำแหน่ง ถ้าพบก็ย้ายข้อมูลตัวท้ายมาทับข้อมูลที่จะถูกลบ อย่าลืมตัดการอ้างอิงอ็อบเจกต์ด้วย

เมท็อดใช้ภายในเพื่อค้นหา index ที่เก็บอ็อบเจกต์ e ถ้าหาไม่พบให้คืน –1

เมท็อดที่ขยายขนาดแถวลำดับอย่างน้อย 2 เท่าของขนาดเดิม ถ้าแถวลำดับเดิมมีขนาดไม่พอ

รหัสที่ 2-7 คลาส ArrayCollection

ArrayCollection แบบไม่จำขนาด

จากคลาส ArrayCollection ที่ได้นำเสนอมา ขอถามว่า จำเป็นต้องมีตัวแปร size ไว้ประจำตัวคอลเล็กชันหรือไม่ ถ้าขอไม่เก็บ size เพื่อประหยัดเนื้อที่ จะได้หรือไม่ การไม่เก็บ size ย่อมส่งผลถึงระเบียบการจัดเก็บข้อมูลและวิธีการจัดการในเมท็อดต่าง ๆ ที่ต้องเปลี่ยนไป คำถามที่ตามมาคือประหยัดไปหนึ่ง int (สี่ไบต์) คุ้มหรือไม่ แน่นอนว่า การเก็บข้อมูลเสริมเพิ่มในอ็อบเจกต์ ก็เพื่อให้การทำงานของเมท็อดต่าง ๆ ง่ายขึ้น แต่ในอีกมุมหนึ่ง เมื่อเพิ่มข้อมูลเสริมเข้าไปแล้ว ก็ต้องปรับปรุงเปลี่ยนแปลงค่าของมันให้ตรงตามความหมายของข้อมูลนั้น ๆ เช่น ในกรณีของ size เราก็ต้องคอยเพิ่มหรือลดค่าของ size ให้มีค่าเท่ากับจำนวนข้อมูลในคอลเล็กชันเสมอ ผู้ออกแบบจึงต้องตัดสินใจให้ดีว่า ภาระที่เพิ่มขึ้นเพื่อปรับปรุงค่าของข้อมูลเสริม กับภาระที่ลดลงที่ได้ใช้ค่าของข้อมูลเสริมนี้ให้เป็นประโยชน์นั้น คุ้มค่ากันไหม

กลับมาที่คำถามว่า จะไม่เก็บ size ได้หรือไม่ คำตอบคือได้ แล้วเราจะเขียนเมท็อด size() ได้อย่างไร จากการเก็บข้อมูลในแถวลำดับ elementData ที่ได้ทำมา ถ้ามีข้อมูล n ตัว ก็จะเริ่มเก็บที่ช่อง 0 ไปถึง n–1 โดยช่องหลังจากนั้นไปมีค่าเป็น null หมด ดังนั้นการวิ่งไล่ในแถวลำดับ elementData จากช่องที่ 0 ไปเรื่อย ๆ จนเจอ null ตัวแรก ก็ย่อมรู้จำนวนข้อมูล นอกจากนี้การถาม isEmpty() ก็ตรวจสอบเพียงแค่ว่า ช่องที่ 0 เป็น null หรือไม่ ถ้าใช่ ย่อมหมายความว่าไม่มีข้อมูล ดังแสดงในรหัสที่ 2-8

```
public int size() {
  for (int i=0; i<elementData.length; i++) {
    if (elementData[i] == null)
      return i;
  }
  return elementData.length;
}
public boolean isEmpty() {
  return elementData[0] == null;
}
```

รหัสที่ 2-8 การทำงานของ size() และ isEmpty() ที่ไม่ต้องใช้ตัวแปร size

สำหรับเมท็อดอื่น ๆ ที่ของเดิมมีการใช้ตัวแปร size ก็ต้องปรับไปใช้ size() แทน และตรงไหนที่มีการเปลี่ยนค่าของ size ก็ไม่ต้องทำ เพราะไม่เก็บ size แล้ว ดังแสดงคลาสที่สมบูรณ์ในรหัสที่ 2-9

```
public class ArrayCollection implements Collection {
  private Object[] elementData;

  public ArrayCollection() {
    elementData = new Object[1];
  }
  public int size() {
    for(int i=0; i<elementData.length; i++)
      if (elementData[i] == null) return i;
    return elementData.length;
  }
  public boolean isEmpty() {
    return elementData[0] == null;
  }
  public void add(Object e) {
    if(e == null) throw new IllegalArgumentException();
    int sz = size();
    if (sz + 1 > elementData.length) {
      int s = Math.max(capacity, 2*elementData.length);
      Object[] arr = new Object[s];
      for(int i = 0; i < sz; i++)
        arr[i] = elementData[i];
      elementData = arr;
    }
    elementData[sz] = e;
  }
  public boolean contains(Object e) {
    return indexOf(e) != -1;
  }
  public void remove(Object e) {
    int i = indexOf(e);
    if (i != -1) {
      int sz = size() - 1;
      elementData[i] = elementData[sz];
      elementData[sz] = null;
    }
  }
  private int indexOf(Object e) {
    for (int i=0; i<elementData.length; i++) {
      if (elementData[i] == null) break;
      if (elementData[i].equals(e)) return i;
    }
    return -1;
  }
}
```

มีแค่แถวลำดับเก็บข้อมูล

ต้องวิ่งไล่นับจากซ้ายไปขวา
จนกว่าจะพบ null

ว่างเมื่อตัวแรกสุดเป็น null

ย้ายการขยายแถวลำดับมาไว้ที่ add เลย
จะได้ไม่ต้องเรียก size() หลายครั้ง

ต้องเรียก size() เพราะ
ต้องการตำแหน่งตัวหลังสุด

วิ่งไล่เปรียบเทียบจากซ้ายไปขวา ถ้าพบ e ก็คืนตำแหน่งนั้น
แต่ถ้าพบ null หรือเมื่อหมดขอบขวาแถวลำดับ ก็คืน -1

รหัสที่ 2-9 คลาส ArrayCollection แบบไม่ต้องจำขนาด

จากการปรับปรุงที่ได้ทำมา เห็นได้ว่าการไม่เก็บ size นั้นไม่คุ้มเลย มันทำให้โปรแกรมอ่านเข้าใจยาก และเมท็อดหลัก ๆ ทำงานช้าลง เพราะต้องการจำนวนข้อมูล การไม่เก็บ size มีข้อดีตรงที่ไม่ต้องคอยเพิ่มหรือลด size ตามการเปลี่ยนแปลง แต่ภาระที่ลดลงตรงนี้ไม่คุ้มเลยกับภาระที่ต้องหาจำนวนข้อมูลในเกือบทุกเมท็อด นี่จึงเป็นตัวอย่างของการเพิ่มข้อมูลเสริมภายในโครงสร้างข้อมูล (ซึ่งคือ size) แล้วได้ผลดี

คอลเล็กชันเก็บได้แต่อ็อบเจกต์

คอลเล็กชันที่ได้นำเสนอมาเก็บอ็อบเจกต์ได้ทุกประเภท แต่มีข้อจำกัดว่า ไม่สามารถเก็บ null ได้ นอกจากนี้ยังไม่สามารถเก็บข้อมูลพื้นฐาน เช่น int, double, char, boolean, ... เพราะเมท็อด add รับพารามิเตอร์แบบ Object เท่านั้น แล้วถ้าเราต้องการใช้คอลเล็กชันที่มีไว้เก็บจำนวนเต็ม จะทำอย่างไร ? ทางออกก็คือการสร้างคลาสที่ห่อข้อมูลพื้นฐานให้เป็นอ็อบเจกต์ก่อน (เรียกว่า wrapper class) ตัวอย่างเช่น คลาส Int (ขอเน้นตรงนี้ว่า ชื่อของคลาสนี้ขึ้นต้นด้วย I ตัวใหญ่) ในรหัสที่ 2-10 คลาสเล็ก ๆ นี้มีข้อมูลภายในเป็น int หนึ่งตัว มีตัวสร้างที่รับ int มาเก็บในอ็อบเจกต์ มีเมท็อด intValue ไว้ให้หยิบค่า int ที่เก็บภายใน และอีกสองเมท็อดที่มักเขียนกำกับคลาสต่าง ๆ คือ equals มีไว้เปรียบเทียบความเท่ากัน และ toString มีไว้เปลี่ยนอ็อบเจกต์เป็นสตริง (ต้องขอบอกว่า เราจะไม่ใช้ Int นี้หรอก เพราะจาวามีคลาส Integer ที่ทำหน้าที่เดียวกันให้ใช้อยู่แล้ว)

```
public class Int {
  private int value;
  public Int(int v) {
    value = v;
  }
  public int intValue() {
   return value;
  }
  public boolean equals(Object x) {
    if (!(x instanceof Int)) return false;
    return value == ((Int)x).value;
  }
  public String toString() { return "" + value; }
}
```

ค่า int ที่เก็บไว้ภายใน

รับ int มาเก็บกำกับอ็อบเจกต์

คืน int ที่เก็บไว้

เปรียบเทียบ x กับอ็อบเจกต์นี้ว่าเท่ากันหรือไม่ ไม่เท่าแน่ ถ้า x ไม่ใช่ Int ถ้าเป็น Int ก็เปรียบเทียบ value ของทั้งคู่

เปลี่ยน value เป็นสตริง

รหัสที่ 2-10 wrapper class ที่ใช้ห่อข้อมูลพื้นฐานแบบ int

ระบบจาวามี wrapper class ในชุด java.lang ชื่อ Character, Integer, Short, Long, Float, Double, และ Boolean สำหรับไว้ห่อข้อมูลพื้นฐาน char, int, short, long, float, double, และ boolean ตามลำดับ

ถ้าเราจะเขียนเมท็อดที่รับแถวลำดับของ int แล้วนำไปสร้างคอลเล็กชันที่เก็บจำนวนเต็มต่าง ๆ ในแถวลำดับที่ได้รับ ก็สามารถทำได้ดังตัวอย่างในรหัสที่ 2-11

```
public static Collection toCollection(int[] a) {
  Collection c = new ArrayCollection();
  for(int i=0; i<a.length; i++) {
    c.add(new Integer(a[i]));
  }
  return c;
}
```

สร้างอ็อบเจกต์ของ Integer ที่เก็บค่า a[i]

รหัสที่ 2-11 toCollection สร้างคอลเล็กชันจากพารามิเตอร์ที่เป็น int[]

autoboxing : ด้วยกระบวนการห่อข้อมูลพื้นฐานให้เป็นออกเจกต์นั้นเกิดขึ้นบ่อยมาก ภาษาจาวาตั้งแต่รุ่น 5.0 เป็นต้นไป จึงเพิ่มคุณสมบัติ autoboxing คือเราสามารถเขียน c.add(25) ได้ ทั้ง ๆ ที่เมท็อด add รับพารามิเตอร์เป็น Object โดยที่ตัวแปลภาษาจะเปลี่ยนคำสั่งดังกล่าวให้เป็น c.add(new Integer(25)) โดยอัตโนมัติ จึงเรียกขั้นตอนนี้ว่า "autobox" ซึ่งหมายถึงการจับข้อมูลพื้นฐาน "ใส่กล่องเป็นอ็อปเจกต์ให้อัตโนมัติ" ตัวแปลภาษาจะเลือก wrapper class ที่ตรงกับข้อมูลพื้นฐานที่เขียนในโปรแกรม เช่นโปรแกรมข้างล่างนี้

```
public class TestAutobox {
  public static void main(String[] args) {
    System.out.println( toObject(25).getClass().getName() );
    System.out.println( toObject(25.0).getClass().getName() );
    System.out.println( toObject(25L).getClass().getName() );
    System.out.println( toObject('c').getClass().getName() );
  }
  static Object toObject(Object x) {
    return x;
  }
}
```

การเรียก x.getClass().getName() จะคืนชื่อของคลาสของอ็อบเจกต์ x โดย getClass() เป็นเมท็อดประจำคลาสบรพบุรุษ Object นั่นคือ ทุกอ็อบเจกต์ในจาวาเรียก getClass() ได้หมด

เมื่อสั่งทำงานจะได้ผลลัพธ์คือ

```
java.lang.Integer
java.lang.Double
java.lang.Long
java.lang.Character
```

การใช้ autoboxing ทำให้เขียนโปรแกรมได้สั้น กะทัดรัด สะอาดดี แต่บ่อยครั้งก็ทำให้หาที่ผิดได้ลำบากขึ้น เพราะตัวแปลภาษาทำอะไรอัตโนมัติให้มาก จนผู้เขียนโปรแกรมลืมไปว่า ตัวแปลภาษาทำให้

บริการอื่น ๆ

ขอนำเสนอบริการอื่น ๆ ที่ไม่ใช่ข้อกำหนดของอินเทอร์เฟซ Collection แต่ก็ควรจะมีเอาไว้เพื่อประโยชน์ในการจัดการข้อมูล และใช้เป็นกรณีศึกษา โดยจะขอนำเสนอ 3 เมท็อดคือ toString, toArray, และ equals

String toString( )

toString เป็นเมท็อดมาตรฐานของคลาสบรรพบุรุษ Object การเขียน toString ไว้ที่คลาสของเรา จึงเป็นการเปลี่ยนพฤติกรรมที่ได้รับจากบรรพบุรุษ toString มีหน้าที่เปลี่ยนอ็อบเจกต์ให้เป็นสตริงที่สื่อความหมายกับ "คน" โดยทั่วไปมีไว้สื่อความหมายกับนักเขียนโปรแกรม ที่ใช้กันมากก็คือ ใช้ระหว่างการหาที่ผิดพลาดของโปรแกรม เนื่องจากระหว่างการหาที่ผิดพลาด นักเขียนโปรแกรมต้องดูเนื้อในของอ็อบเจกต์ โดยแสดงอ็อบเจกต์ที่อ่านได้ความหมาย แล้วตรวจสอบว่า สภาพของอ็อบเจกต์เป็นไปตามพฤติกรรมของโปรแกรมที่ถูกต้องหรือไม่

เนื้อในของคอลเล็กชันก็คือข้อมูลที่เก็บอยู่ในคอลเล็กชัน ขอกำหนดรูปแบบของผลลัพธ์ให้เป็น [e_0, e_1, e_2, ..., e_{n-1}] โดยที่ e_i คือข้อมูลแต่ละตัวในคอลเล็กชัน จะแสดงลำดับของข้อมูลอย่างไรก็ได้ เช่น คอลเล็กชันหนึ่งเก็บข้อมูล 3 ตัวคือ "A", "B", และ "C" เมื่อเรียก toString แล้วอาจจะได้ ["A", "B", "C"] หรือจะได้ ["B", "C", "A"] ก็ไม่ว่ากัน เพราะถือว่า แทนการเก็บข้อมูลที่มีความหมายเหมือนกัน วิธีทำงานง่าย ๆ ของ toString ใน ArrayCollection ก็น่าจะเป็นการนำข้อมูลใน elementData ตั้งแต่ช่องที่ 0 ถึง size-1 มาสร้างเป็นสตริง ดังรหัสที่ 2-12

```
public class ArrayCollection implements Collection {
  ...
  public String toString() {
    String out = "[";
    for(int i=0; i<size; i++) {
      out = out + elementData[i].toString();
      if (i+1 < size) out = out + ",";
    }
    return out + "]";
  }
  ...
```

เปลี่ยนอ็อบเจกต์ในแถวลำดับแต่ละตัว เป็นสตริง ด้วยการเรียก toString

รหัสที่ 2-12 toString ของคลาส ArrayCollection

การนำสตริงมา + กันในจาวานั้น เป็นการสร้างสตริงใหม่ รหัสที่ 2-12 นั้นมีการ + สตริงมาก จึงมีการสร้างสตริงใหม่ (และทิ้งสตริงเก่า) เป็นจำนวนมาก วิธีหลีกเลี่ยงเหตุการณ์เช่นนี้ ทำได้โดยใช้คลาส StringBuffer แทน ซึ่งมีเมท็อด append ใช้ต่อข้อความ ที่ลดจำนวนการสร้างอ็อบเจกต์ใหม่ ดังรหัสข้างล่างนี้

```
public String toString() {
  StringBuffer out = new StringBuffer("[");
  for(int i=0; i<size; i++) {
    out.append(elementData[i].toString());
    if (i + 1 < size) out.append(",");
  }
  return out.append("]").toString();
}
```

หมายเหตุ : กรณีที่ใช้จาวารุ่นที่ 5 เป็นต้นไป สามารถใช้คลาส StringBuilder แทน StringBuffer ที่ให้ประสิทธิภาพการทำงานที่ดีกว่า

Object[ ] toArray( )

ผู้อ่านอาจสงสัยว่า คอลเล็กชันทำตัวเป็นที่เก็บข้อมูลเพื่อให้ค้นเท่านั้น ไม่มีบริการดึงข้อมูลที่เก็บไว้ออกมาใช้งานเลย เช่น ถ้าต้องการการหาผลรวมของคอลเล็กชันที่เก็บจำนวนเต็ม จะทำอย่างไร ? ขอเสนอวิธีง่าย ๆ ที่ให้บริการผ่านเมท็อด `toArray` ซึ่งคืนแถวลำดับที่แต่ละช่องเก็บข้อมูลแต่ละตัวในคอลเล็กชัน ผลลัพธ์จึงเป็นแถวลำดับที่มีขนาดเท่ากับจำนวนข้อมูลในคอลเล็กชัน ถ้า `c` คือคอลเล็กชันที่เก็บอ็อบเจกต์ของ `Integer` เราก็สามารถหาผลรวมของข้อมูลใน `c` ได้ด้วยรหัสที่ 2-13

```
public int sum(ArrayCollection c) {
  Object [] a = c.toArray();
  int s = 0;
  for(int i=0; i<a.length; i++) {
    Integer x = (Integer) a[i];
    s += x.intValue();
  }
  return s;
}
```

ดึงข้อมูลใน c ออกมาในรูปของแถวลำดับ

ต้อง cast ให้เป็น Integer

ดึง int ออกจาก x ด้วย intValue

รหัสที่ 2-13 ตัวอย่างการใช้ `toArray`

บางคนคิดว่า เขียน `toArray` ง่ายมาก เพียงแค่คืน `elementData` เป็นผลลัพธ์กลับไปก็จบ ต้องบอกว่า ทำไม่ได้ เพราะไม่ถูกต้องตามข้อกำหนด เรากำหนดให้ต้องคืนแถวลำดับที่มีขนาดเท่ากับจำนวนข้อมูลเพื่อความสะดวกของผู้ใช้ในการประมวลผล แต่ `elementData` อาจมีจำนวนช่องมากกว่าจำนวนข้อมูลได้ นอกจากนี้การคืน `elementData` ให้ผู้เรียกก็ไม่เป็นเรื่องดี เพราะเป็นการเปิดโอกาสให้ผู้เรียกสามารถเปลี่ยนแปลงช่องต่าง ๆ ใน `elementData` ส่งผลให้คอลเล็กชันผิดรูปแบบไปได้

ดังนั้น `toArray` จึงต้องสร้างแถวลำดับใหม่ให้มีขนาดเท่ากับจำนวนข้อมูล แล้วทำสำเนาข้อมูลจาก `elementData` ไปยังแถวลำดับใหม่ ดังรหัสที่ 2-14

```
public class ArrayCollection implements Collection {
  ...
  public Object[] toArray() {
    Object [] a = new Object[size];
    for(int i=0; i<size; i++) {
      a[i] = elementData[i];
    }
    return a;
  }
  ...
```

จองแถวลำดับใหม่

ให้แถวลำดับใหม่อ้างอิงไปยังข้อมูลทุกตัวในคอลเล็กชัน

รหัสที่ 2-14 `toArray` ของคลาส `ArrayCollection`

boolean equals(Object x)

equals ให้บริการเปรียบเทียบคอลเล็กชันสองชุดว่า มีข้อมูลภายในเหมือนกันหรือไม่ ตัวอย่างเช่น คอลเล็กชัน [A, B, C, C] เท่ากับ [C, B, C, A] แต่ไม่เท่ากับ [A, B, C] รหัสที่ 2-15 แสดงการทำงานของ equals ที่เรียก toArray ของทั้งสองคอลเล็กชัน ได้แถวลำดับมาสองชุด นำไปเรียงลำดับด้วยบริการ Arrays.sort (Arrays เป็นคลาสในชุด java.util) เมื่อได้แถวลำดับที่เรียงลำดับแล้ว ก็นำไปเปรียบเทียบข้อมูลแบบตัวต่อตัวตำแหน่งต่อตำแหน่งด้วยบริการ Arrays.equals

```
public class ArrayCollection implements Collection {
  ...
  public boolean equals(Object x) {
    if (!(x instanceof ArrayCollection)) return false;
    Object[] a1 = ((ArrayCollection) x).toArray();
    Object[] a2 = this.toArray();
    Arrays.sort(a1);
    Arrays.sort(a2);
    return Arrays.equals(a1, a2);
  }
```

ขอข้อมูลทั้งหมดเป็นแถวลำดับ

นำไปเรียงลำดับจากน้อยไปมาก

เปรียบเทียบแถวลำดับแบบตัวต่อตัว

รหัสที่ 2-15 equals ของ ArrayCollection ที่เก็บอ็อบเจกต์แบบ Comparable

equals ในรหัสที่ 2-15 จะใช้ได้ ก็เมื่อเราสามารถเรียงลำดับข้อมูลได้ แต่สมมติฐานนี้อาจไม่เป็นจริงเสมอไป เพราะถึงแม้เราจะสามารถเปรียบเทียบว่า อ็อบเจกต์สองตัวเท่ากันหรือไม่ได้ด้วย equals ก็ตาม แต่การเรียงลำดับนั้นอาศัยการเปรียบเทียบความน้อยกว่ามากกว่าของอ็อบเจกต์ ซึ่งไม่จำเป็นว่า อ็อบเจกต์ทุกแบบจะทำการเปรียบเทียบความน้อยกว่ามากกว่าได้ จึงต้องเขียน equals ให้กับ ArrayCollection ใหม่ที่ใช้ equals ของตัวอ็อบเจกต์เท่านั้น ดังแสดงในรหัสที่ 2-16

```
  public boolean equals(Object x) {
    if (!(x instanceof ArrayCollection)) return false;
    Object[] a1 = ((ArrayCollection) x).toArray();
    nextElement:
    for (int i=0; i<size; i++) {
      for (int j=0; j<a1.length; j++) {
        if (elementData[i].equals(a1[j])) {
          a1[j] = null; continue nextElement;
        }
      }
      return false;
    }
    for (int j=0; j<a1.length; j++)
      if (a1[j] != null) return false;
    return true;
  }
```

หยิบข้อมูลใน elementData ทีละตัวออกมาค้นใน a1

พบที่ช่องใดใน a1 ก็เปลี่ยนช่องนั้นให้เป็น null แล้วไปทำ elementData ตัวถัดไป

ค้นไม่พบใน a1 แสดงว่าไม่เท่า

เหลือบางตัวใน a1 ที่ไม่เป็น null ก็แสดงว่าไม่เท่า

รหัสที่ 2-16 equals ของ ArrayCollection ที่เก็บอ็อบเจกต์ทั่วไป

equals ในรหัสที่ 2-16 ใช้ toArray ดึงอ็อบเจกต์ต่าง ๆ ในคอลเล็กชัน x ที่ได้รับออกมาเป็นแถวลำดับ a1 แล้วเริ่มหยิบข้อมูลใน elementData ทีละตัวเพื่อนำมาค้นในแถวลำดับ a1 เมื่อพบ ก็เปลี่ยนช่องที่พบให้ a1 นั้นให้เป็น null ซึ่งเป็นการตัดข้อมูลใน a1 ตัวนั้นทิ้งออกจากแถวลำดับ (การทำเช่นนี้ไม่ได้ลบข้อมูลตัวนั้นออกจากคอลเล็กชันเดิมแต่อย่างใด เพราะเป็นการตัดแค่การอ้างอิงเท่านั้น) แล้วก็วนกลับไปพิจารณาข้อมูลตัวถัดไปใน elementData เมื่อใดค้นไม่พบก็แสดงว่าไม่เท่ากัน เมื่อวนค้นจนครบทุกตัวใน elementData แล้ว แต่กลับยังมีข้อมูลเหลือใน a1 (คือมีบางช่องใน a1 ที่ไม่เป็น null) ก็แสดงว่าไม่เท่าเช่นกัน ถ้าเป็น null หมด ก็แสดงว่าเท่ากัน

ในจาวา คลาสใดที่ตัวอ็อบเจกต์สามารถเปรียบเทียบความน้อยกว่ามากกว่าได้ จะต้องเป็นคลาสที่ implements อินเทอร์เฟซ Comparable โดยอินเทอร์เฟซนี้บังคับว่า ต้องมีเมท็อด compareTo ที่มีหัวดังนี้

```
public int compareTo(Object x)
```

compareTo มีหน้าที่เปรียบเทียบอ็อบเจกต์ที่ถูกเรียกกับ x ถ้ามากกว่า x ให้คืนจำนวนเต็มบวก ถ้าเท่ากันให้คืน 0 ถ้าน้อยกว่า x ให้คืนจำนวนเต็มลบ รหัสข้างล่างนี้แสดงตัวอย่างของ compareTo ในคลาส Integer โดยการนำค่าที่เป็นจำนวนเต็มภายในอ็อบเจกต์ออกมาเปรียบเทียบ (wrapper classes ของข้อมูลพื้นฐาน ล้วนเป็น Comparable ทั้งสิ้น)

```
public class Integer implements Comparable {
  ...
  public int compareTo(Object x) {
    Integer that = (Integer) x;
    int thisVal = this.intValue();
    int thatVal = that.intValue();
    return (thisVal < thatVal ? -1 : thisVal == thatVal ? 0 : 1);
  }
  ...
}
```

อีกสักตัวอย่าง รหัสข้างล่างนี้แสดงคลาส RationalNumber คลาสนี้ถูกออกแบบมาเพื่อสร้างจำนวนตรรกยะ ซึ่งคือจำนวนจริงที่แทนได้ด้วยจำนวนเต็มที่เรียกว่าเศษ (numerator) หารด้วยจำนวนเต็มที่เรียกว่าส่วน (denominator) เช่น 5/7, 1/3, 0/1 เป็นต้น compareTo อาศัยการนำเศษมาหารส่วนได้จำนวนจริงเพื่อนำมาเปรียบเทียบความมากกว่าน้อยกว่าได้ง่าย ๆ

```
public class RationalNumber implements Comparable {
  private int numerator;
  private int denominator;
  ...
  public int compareTo(Object x) {
    RationalNumber that = (RationalNumber) x;
    double thisVal = (double) this.numerator / this.denominator;
    double thatVal = (double) that.numerator / that.denominator;
    return (thisVal < thatVal ? -1 : thisVal == thatVal ? 0 : 1);
  }
  ...
}
```

แบบฝึกหัด

1. จงเขียนคลาส `ArrayCollection` ด้วยตนเอง โดยไม่ดูรายละเอียดในหนังสือ

2. ถ้าเราเปลี่ยนบรรทัดที่ 6 ของรหัสที่ 2-7 ให้จองแถวลำดับตอนเริ่มต้น 0 ตัวด้วย `new Object[0]` จะมีปัญหาหรือไม่ ที่ใด อย่างไร

3. คลังคลาสมาตรฐานของจาวา ก็มีอินเทอร์เฟซ `Collection` (อยู่ในชุด `java.util`) แต่ไม่มีคลาสที่สร้างให้เป็นคอลเล็กชันโดยตรง (เหมือน `ArrayCollection` ที่เราได้นำเสนอมา) อย่างไรก็ตาม ก็มีคลาสใน `java.util` ที่เป็น `Collection` ซึ่งถูกสร้างให้ implements อินเทอร์เฟซบางตัวที่เป็นอินเทอร์เฟซลูกของ `Collection` อีกทีหนึ่ง จงอธิบายบริการต่าง ๆ ของ `java.util.Collection` และค้นหาว่า มีคลาสอะไรบ้างที่ใช้เป็นคอลเล็กชันได้ในชุด `java.util`

4. เพื่อให้แถวลำดับ `elementData` เก็บข้อมูลได้คุ้มกับเนื้อที่ที่จอง ก็ควรให้แถวลำดับหดตัวได้ หากพบว่า หลังการลบ สัดส่วนของข้อมูลที่เก็บกับขนาดของแถวลำดับมีค่าต่ำกว่าค่าที่กำหนดไว้ เช่น ถ้าต่ำกว่า 25% ก็ให้หด `elementData` ลงครึ่งหนึ่ง จงปรับปรุง `ArrayCollection` ให้มีความสามารถดังกล่าว

5. `ArrayCollection` ที่เขียนมาเก็บอ็อบเจกต์ได้ทุกประเภท ส่วนของโปรแกรมข้างล่างนี้มีปัญหาที่ใด หรือไม่ อย่างไร

```
ArrayCollection c1 = new ArrayCollection();
ArrayCollection c2 = new ArrayCollection();
c1.add("A"); c2.add("B");
c1.add(c2);  System.out.println(c1);
c1.add(c1);  System.out.println(c1);
```

6. `ArrayCollection` ที่เขียนมาเก็บอ็อบเจกต์ได้ทุกประเภท ถ้าจะนำไปเก็บ `int` ก็ต้องใช้คลาส `Integer` ช่วย ซึ่งสิ้นเปลือง จงเขียนคลาส `IntArrayCollection` ซึ่งเป็นคอลเล็กชันที่มีไว้เก็บเฉพาะจำนวนเต็ม

7. จงเขียนคลาส `ArraySet` โดยเขียนให้เป็นคลาสลูกของ `ArrayCollection` (รหัสที่ 2-7) ที่ไม่อนุญาตให้เก็บข้อมูลซ้ำ ถ้ามีการสั่งเพิ่มตัวซ้ำ จะไม่เพิ่มให้ คืนการทำงานกลับไปเลย

8. หากเราใช้ `ArrayCollection` ในรหัสที่ 2-7 เพื่อสร้างคอลเล็กชันแล้วเก็บข้อมูลสักล้านตัวด้วยส่วนของโปรแกรมข้างล่างนี้ ถามว่า จะมีการขยายแถวลำดับใน `ensureCapacity` กี่ครั้ง

```
Collection c = new ArrayCollection();
for (int i=0; i<1000000; i++) c.add("A");
```

9. ศึกษาเมท็อด `arraycopy` ของคลาส `System` (เป็นคลาสมาตรฐานของจาวา) เพื่อนำไปใช้ทำสำเนาแถวลำดับแทนบรรทัดที่ 39-40 ในเมท็อด `ensureCapacity` ของรหัสที่ 2-7

10. เขียนโปรแกรมจับเวลาการสร้างคอลเล็กชันที่มีข้อมูลสักสิบล้านตัวด้วย `ArrayCollection` (รหัสที่ 2-7) เปรียบเทียบกับ `ArrayCollection` แบบไม่จำขนาด (รหัสที่ 2-9) และเปรียบเทียบกับการปรับปรุงในแบบฝึกหัดข้อ 9 ว่า ใช้เวลาแตกต่างกันหรือไม่ อย่างไร

    (หมายเหตุ : เนื่องจากเราต้องการเก็บข้อมูลจำนวนมาก ต้องบอกให้ตัว jvm ทราบด้วยว่า ต้องจองเนื้อที่มากตอนเริ่มทำงาน เช่น สมมติว่า โปรแกรมที่เขียนชื่อ `Test.java` เมื่อเราจะสั่งทำงานก็ใช้คำสั่ง `java Test` แต่ถ้าใช้ `java -Xms200M -Xmx200M Test` จะหมายความว่าขอหน่วยความจำ 200MB ไว้ใช้งาน ถ้าใช้ Eclipse เป็นซอฟต์แวร์ในการเขียนโปรแกรมก็ต้องตั้งที่เมนู `Run->Run...` ถ้าใช้ JLab ก็ต้องเพิ่มที่เมนู `Tools->Options->JDK Tools->java options` นอกจากนี้แนะนำให้ใช้ `System.currentTimeMillis()` หรือ `System.nanoTime()` จับเวลา ดังตัวอย่างข้างล่างนี้ (อย่าใช้นาฬิกาข้อมือในการจับเวลา ☺)

```
Collection c = new ArrayCollection();
long t = System.nanoTime();
for(int i=0; i<10000000; i++) c.add("A");
System.out.println( System.nanoTime() - t );
```

11. จงเขียนเมท็อดต่อไปนี้ (ที่ทำงานเร็ว ๆ) เพิ่มให้กับ `ArrayCollection` (รหัสที่ 2-7)

    11.1. `boolean containsDup()` เพื่อตรวจสอบว่า มีข้อมูลซ้ำกันหรือไม่ในคอลเล็กชัน

    11.2. `void clear()` เพื่อล้างคอลเล็กชันให้ไม่มีข้อมูลเหลืออยู่เลย

    11.3. `int frequency(Object e)` เพื่อนับว่า มี `e` ปรากฏกี่ตัวในคอลเล็กชัน

    11.4. `void removeAll(Object e)` เพื่อลบ `e` ทุกตัวในคอลเล็กชันออกให้หมด

    11.5. `void removeDup()` เพื่อลบข้อมูลที่ซ้ำกันให้เหลือแค่ตัวเดียว เช่นเดิมเป็น [A,B,A,B,B] เมื่อลบแล้วจะได้ [A,B]

    11.6. `void trimToSize()` เพื่อปรับขนาดของ `elementData` ใหม่ให้มีขนาดพอดีกับจำนวนข้อมูลในคอลเล็กชัน

    11.7. `boolean containsAll(ArrayCollection c)` เพื่อตรวจว่า คอลเล็กชันนี้มีข้อมูลทุกตัวที่ `c` มีหรือไม่ เช่น ถ้า `c1` เก็บ [A, B, C, A, D] และ `c2` เก็บ [A, B, B] จะได้ `c1.containsAll(c2)` เป็นจริง แต่ `c2.containsAll(c1)` เป็นเท็จ

11.8. `Object mode()` คืนฐานนิยม (mode) ของคอลเล็กชัน ฐานนิยมคือข้อมูลที่ปรากฏในคอลเล็กชันเป็นจำนวนมากสุด เช่น `c1` เก็บ [A, B, A, B, A, C] เมื่อเรียก `c1.mode()` จะคืน A ในกรณีที่มีฐานนิยมมากกว่าหนึ่งตัว คืนตัวไหนก็ได้

11.9. เพิ่มตัวสร้าง `ArrayCollection(ArrayCollection c)` เพื่อสร้างคอลเล็กชันที่มีข้อมูลเริ่มต้นเหมือนของ `c` หมายเหตุ : ตัวสร้างในลักษณะนี้เรียกว่า *ตัวสร้างทำสำเนา* (copy constructor)

11.10. เพิ่มตัวสร้าง `ArrayCollection(int initialCapacity)` เพื่อกำหนดขนาดเริ่มต้นของแถวลำดับ หมายเหตุ : ตัวสร้างแบบนี้มีประโยชน์ในกรณีที่ผู้ใช้ทราบจำนวนของข้อมูลที่จะเก็บ จะได้ไม่ต้องเสียเวลาขยายแถวลำดับ

12. `Random` เป็นคลาสมาตรฐานของจาวา ถ้าเราต้องการสุ่มจำนวนเต็มตั้งแต่ `0` ถึง `n-1` ก็ให้สร้างอ็อบเจกต์ของ `Random` แล้วเรียกเมท็อด `nextInt(n)` ส่วนของโปรแกรมข้างล่างนี้ผลิตจำนวนเต็มสุ่มตั้งแต่ 0 ถึง 99 จำนวน 5 ตัว

```
Random r = new Random(123);
for (int i=0; i<5; i++)
  System.out.println(r.nextInt(100));
```

โดยตัวสร้างของ `Random` จะรับจำนวนเต็ม (เรียกว่า seed ของตัวสุ่ม) ที่ถูกนำไปกำหนดพฤติกรรมการสุ่ม (ตัวอย่างข้างบนนี้ก็คือเลข 123) จงเขียน

```
int testRandom(long seed, int n)
```

ซึ่งตรวจสอบว่า อ็อบเจกต์ `Random` ที่สร้างด้วย seed ที่ส่งให้ แล้ววนเรียก `nextInt(n)` จะผลิตจำนวนเต็มแบบสุ่มที่ต่างกันออกมากี่ตัว ถึงจะเริ่มซ้ำกับตัวที่เคยผลิตออกมาก่อน

3 การวิเคราะห์เชิงเส้นกำกับ

เราทราบได้อย่างไรว่า เมท็อดที่ได้เขียน ๆ มาในคลาส `ArrayCollection` มีประสิทธิภาพมากน้อยเพียงใด เมท็อดหนึ่งอาจเขียนขั้นตอนการทำงานได้หลายรูปแบบ แล้วจะมีวิธีเปรียบเทียบประสิทธิภาพได้อย่างไร บทนี้จะนำเสนอวิธีการวิเคราะห์เชิงเส้นกำกับ (asymptotic analysis) ที่ช่วยประเมินประสิทธิภาพทางด้านเวลาการทำงานของเมท็อด วิธีนี้ใช้ได้ดีกับกรณีที่ข้อมูลมีปริมาณมาก เป็นวิธีวิเคราะห์ที่ไม่ซับซ้อน ละเลยเรื่องจุกจิกที่ไม่ควรสนใจ พุ่งเป้าเฉพาะจุดที่มีอิทธิพลต่อเวลาการทำงานมากที่สุด แสดงให้เห็นภาพรวมของการเติบโตของฟังก์ชันเวลาการทำงานกับปริมาณข้อมูล ซึ่งสามารถนำผลไปใช้เปรียบเทียบได้อย่างดี

เวลาการทำงาน

เราจะวัดเวลาการทำงานของเมท็อดได้อย่างไร ? คำตอบสั้น ๆ ก็คือเขียนเป็นโปรแกรม แปล สั่งทำงาน แล้วจับเวลา แต่นั่นไม่ใช่วิธีที่ต้องการในทางปฏิบัติ สิ่งที่ต้องการคือมาตรวัดอะไรบางอย่างที่เราสามารถใช้วัดเวลาการทำงานของเมท็อด โดยไม่ต้องลองสั่งทำงานจริง คำถามที่ตามมาก็คือจะเป็นไปได้อย่างไร ในเมื่อเรายังไม่รู้เลยว่า จะใช้งานบนเครื่องคอมพิวเตอร์เครื่องไหน ใช้ตัวแปลภาษา และตัวสั่งทำงานของบริษัทใด ถูกต้องแล้วที่บอกว่า เป็นไปไม่ได้หรอกที่จะวัดเวลาการทำงานจริง สิ่งที่เราต้องการคือฟังก์ชันที่แสดงความสัมพันธ์ของจำนวนครั้งที่คำสั่งพื้นฐานในเมท็อดทำงาน กับปริมาณข้อมูล เนื่องจากจำนวนครั้งที่คำสั่งพื้นฐานในเมท็อดทำงานนั้นมีความสัมพันธ์โดยตรงกับเวลาการทำงานจริง เช่น กำหนดให้ n แทนปริมาณข้อมูลที่จัดเก็บด้วยวิธีที่เราได้ออกแบบ ถ้าพบว่า บริการการค้นข้อมูลต้องใช้งานคำสั่งพื้นฐานเป็นจำนวน $n + 9$ ครั้ง ก็สรุปได้ว่า เวลาการทำงานของการค้นข้อมูลนั้นเป็นฟังก์ชันเชิงเส้นกับปริมาณข้อมูล ตีความได้ว่า หากปริมาณข้อมูลเพิ่มขึ้น 16 เท่า การค้น

ข้อมูลก็จะใช้เวลาเพิ่มขึ้น 16 เท่าเช่นกัน หากเราพบวิธีการจัดเก็บข้อมูลอีกแบบที่การค้นข้อมูลต้องสั่งทำงานคำสั่งพื้นฐานเป็นจำนวน 2 $\log_2 n$ ครั้ง ก็สรุปได้ว่า เวลาการทำงานของการค้นข้อมูลในรูปแบบการจัดเก็บวิธีใหม่นี้เป็นฟังก์ชันแบบ $\log_2$ ถ้าปริมาณข้อมูลเพิ่มขึ้น 16 เท่า การค้นข้อมูลจะใช้เวลาเพิ่มขึ้น $\log_2 16 = 4$ เท่า ถ้าปริมาณข้อมูลมีมาก วิธีการจัดเก็บแบบที่สอง ย่อมใช้เวลาการค้นข้อมูลที่น้อยกว่าแบบแรก ด้วยเหตุนี้ภาระการวิเคราะห์ประสิทธิภาพเชิงเวลาก็คือการหาฟังก์ชันที่แสดงความสัมพันธ์ของจำนวนครั้งที่คำสั่งพื้นฐานถูกใช้งานกับปริมาณข้อมูล

คำสั่งพื้นฐาน

ผู้อ่านคงสงสัยต่อว่า คำสั่งต่าง ๆ ในโปรแกรมใช้เวลาการทำงานแตกต่างกัน แล้วอะไรที่เราถือได้ว่าเป็นคำสั่งพื้นฐาน กำหนดให้คำสั่งพื้นฐานคือคำสั่งที่ใช้เวลาการทำงานคงตัวไม่ขึ้นกับปริมาณข้อมูลที่จัดเก็บ และก็ไม่ขึ้นกับค่าของตัวถูกดำเนินการหรือพารามิเตอร์ คำสั่งและตัวดำเนินการต่าง ๆ ของภาษาการเขียนโปรแกรมส่วนใหญ่เป็นคำสั่งพื้นฐานเช่น `= + - * / % == != < > return` เป็นต้น บางเมท็อดใช้เวลาคงตัวไม่ขึ้นกับค่าของพารามิเตอร์ที่ได้รับหรือปริมาณข้อมูลที่จัดเก็บ ก็ถือได้ว่า เป็นคำสั่งพื้นฐาน เช่น `Math.max(a, b)` ซึ่งเป็นเมท็อดที่คืนตัวที่มีค่ามากระหว่างตัวแปร `a` และ `b` ภายใน `max` ประกอบด้วยคำสั่งพื้นฐานเป็นจำนวนคงตัว เรียกใช้ `max` เมื่อใด ก็ใช้งานคำสั่งพื้นฐานภายในเป็นจำนวนคงตัว

แต่ถ้าเป็นคำสั่งที่ใช้เวลาไม่คงตัว ขึ้นกับค่าของพารามิเตอร์หรือปริมาณข้อมูล ก็ถือว่าไม่ใช่คำสั่งพื้นฐาน เช่น `int[] x = new int[n]` การสร้างแถวลำดับแบบนี้ในจาวา จะตั้งค่าของแต่ละช่องเป็น 0 แสดงว่า ต้องใช้เวลาใส่ค่า 0 จำนวน `n` ช่อง จึงไม่เป็นคำสั่งพื้นฐาน หรือการเรียกเมท็อด `Arrays.sort(x)` ซึ่งมีหน้าที่เรียงลำดับข้อมูลในแถวลำดับ `x` จะใช้เวลามากหรือน้อยก็ขึ้นกับปริมาณข้อมูลที่ส่งไปเรียงลำดับ จึงไม่ใช่คำสั่งพื้นฐาน แล้ว `sort` นี้มีฟังก์ชันของเวลาการทำงานเป็นเท่าใด จะตอบได้ก็ต้องวิเคราะห์ละเอียดเข้าไปในตัวเมท็อด `sort` ซึ่งจะได้นำเสนอในรายละเอียดในบทหลัง ๆ

การนับจำนวนคำสั่งที่ถูกใช้งาน

มาลองนับจำนวนคำสั่งพื้นฐานกัน ขอใช้เมท็อด `remove` ในคลาส `ArrayCollection` ที่แสดงในรหัสที่ 3-1 เป็นตัวอย่าง เริ่มบรรทัดที่ 24 เรียก `indexOf` แล้วตามด้วย = หนึ่งครั้ง บรรทัดต่อมาทำ `i!=1` หนึ่งครั้ง ถ้าเป็นจริงทำ `--size` หนึ่งครั้ง = อีกครั้งในบรรทัดที่ 26 ตามด้วย = อีกครั้งในบรรทัดที่ 27 เป็นคำสั่งสุดท้าย ภาระของ `indexOf` เริ่มที่บรรทัดที่ 31 ใช้คำสั่ง `for` ทำ `i=0`

หนึ่งครั้ง แล้วเริ่มวงวน จะหลุดออกจากวงวนก็เมื่อ `i` มีค่าเป็น `size` หรือว่า เงื่อนไขของ `if` เป็นจริง ถ้าเป็นเท็จก็จะทำ `i++` อีกหนึ่งครั้งแล้ววนต่อ จะรู้ได้อย่างไรว่า หมุนในวงวนกี่รอบ เราไม่รู้ เพราะมันขึ้นกับ `e` ว่า มีเก็บอยู่ใน `elementData` หรือไม่ แต่เรารู้อยู่อย่างหนึ่งว่า วงวนนี้จะหมุนเป็นจำนวนรอบมากสุดเมื่อไม่มี `e` อยู่ใน `elementData` ซึ่งคือกรณีหาไม่พบ การทดสอบ `equals` ก็เป็นเท็จตลอด ทำให้หมุนเป็นจำนวนรอบเท่ากับค่าในตัวแปร `size` (จึงต้องเปรียบเทียบ `i<size` เป็นจำนวน `size+1` ครั้ง) และทำ `return` ที่บรรทัดที่ 33 อีกหนึ่งครั้ง ถ้า `equals` เป็นคำสั่งพื้นฐาน จะได้ว่า `indexOf` ทำอย่างมาก `1+(size+1)+2(size)+1` คำสั่ง ดังนั้น `remove` สั่งทำคำสั่งทั้งหมดอย่างมาก `5+1+(size+1)+2(size)+1 = 8+3(size)` ครั้ง เนื่องจาก `size` เก็บจำนวนข้อมูลในคอลเล็กชัน สรุปได้ว่า ในกรณีที่ทำงานช้าสุด `remove` ใช้เวลาการทำงานเป็นฟังก์ชันเชิงเส้นกับจำนวนข้อมูลในคอลเล็กชัน

```
01 public class ArrayCollection implements Collection {
..   ...
23   public void remove(Object e) {
24     int i = indexOf(e);
25     if (i != -1) {
26       elementData[i] = elementData[--size];
27       elementData[size] = null;
28     }
29   }
30   private int indexOf(Object e) {
31     for (int i=0; i<size; i++)
32       if (elementData[i].equals(e)) return i;
33     return -1;
34   }
     ...
```

1 ครั้ง

1 ครั้ง

size+1 ครั้ง

size ครั้ง

รหัสที่ 3-1 จำนวนครั้งที่คำสั่งต่าง ๆ ในเมท็อด `remove` ทำงาน ในกรณีทำงานช้าสุด

การนับคำสั่งที่ได้ทำมานั้น ถ้าจะนับกันอย่างละเอียดยิบ อาจจะนับยังไม่ครบ บางคนอาจบอกว่า แค่เขียน `elementData[i]` ก็ถือว่า เป็นการทำงานอย่างหนึ่ง เพราะต้องใช้เวลาอ่านข้อมูลจากช่องที่ `i` ในแถวลำดับ ถ้าเราจะนับคำสั่งแบบนี้ด้วย ผลที่ได้ก็จะเปลี่ยนไปเป็น `11+4(size)` คำสั่ง แต่สิ่งที่ไม่เปลี่ยนก็คือการทำงานยังคงใช้เวลาเป็นฟังก์ชันเชิงเส้นกับปริมาณข้อมูล ขอให้เข้าใจด้วยว่า ถ้า n คือจำนวนข้อมูล $t(n)$ คือเวลาการทำงานจริง $c(n)$ คือจำนวนครั้งที่คำสั่งพื้นฐานทำงาน จะได้ว่า $t(n) \le k\ c(n)$ โดยที่ k เป็นค่าคงตัวค่าหนึ่งที่แทนเวลาการทำงานของคำสั่งพื้นฐานที่ทำงานนานสุด ถ้าเราจะนับจำนวนครั้งที่คำสั่งพื้นฐานทำงานให้ละเอียดแบบสุด ๆ ก็ต้องนับคำสั่งระดับรหัสเครื่องเลย (ถ้าเป็น jvm ก็คือการนับจำนวนครั้งที่ byte code ทำงาน) ก็อาจได้เป็นฟังก์ชัน $c_0(n)$ โดยที่ $t(n) \le k_0c_0(n)$ และ k_0 คือค่าคงตัวอีกค่าหนึ่ง นั่นหมายความว่า การนับแบบละเอียดจะเป็นจำนวนเท่าที่คง

ตัวของการนับแบบหยาบ ลักษณะการเติบโตของฟังก์ชันไม่ว่าจะนับแบบหยาบ แบบละเอียด หรือแบบจับเวลาจริง ๆ จะยังคงคล้าย ๆ กัน ดังนั้นถ้าต้องการจะลดภาระการนับ เราสามารถนับแบบหยาบ เลือกเฉพาะคำสั่งพื้นฐานที่สำคัญ ๆ เป็นคำสั่งตัวแทน โดยจะต้องเป็นคำสั่งตัวแทนที่เมื่อนับแล้วทำให้เวลาการทำงานจริงแปรผันตรง ๆ กับจำนวนครั้งที่คำสั่งตัวแทนนี้ทำงาน ก็จะลดภาระไปได้มาก เช่น ถ้าเราดู `remove` ในรหัสที่ 3-1 ให้ดีก็จะพบว่า เราเลือกนับเฉพาะคำสั่ง `equals` ในบรรทัดที่ 32 ก็พอ ในกรณีค้นไม่พบข้อมูล หรือกรณีที่พบข้อมูลแต่เป็นตัวท้ายสุด จะหมุนในวงวนเป็นจำนวนรอบมากที่สุด `equals` ก็จะถูกเรียกใช้เป็นจำนวน `size` ครั้ง แสดงให้เห็นว่า การทำงานของ `remove` ใช้เวลาแปรตาม `size` แบบเชิงเส้น

ดูกันอีกสักตัวอย่างในรหัสที่ 3-2 `dummy` (ที่ทำงานไร้สาระ) มีการทำงานเป็นวงวนสองวนซ้อนกัน เห็นได้ว่า เราสามารถใช้คำสั่งในบรรทัดที่ 5 เป็นคำสั่งตัวแทนของเวลาการทำงานได้ เพราะเป็นคำสั่งที่ทำงานในวงวนในสุด ถ้านับเฉพาะ `for` วงในของบรรทัดที่ 4 ถึง 5 พบว่า ทำบรรทัดที่ 5 ทั้งสิ้น $\sum_{j=0}^{i-1} 1$ ครั้ง ดังนั้นรวม ๆ แล้วบรรทัดที่ 5 ที่ทำงานภายในวงวน `for` ตั้งแต่บรรทัดที่ 3 ถึง 5 ย่อมทำเป็นจำนวน $\sum_{i=1}^{n-1}\left(\sum_{j=0}^{i-1} 1\right) = \sum_{i=1}^{n-1}(i) = n(n-1)/2$ ครั้ง

```
01 public void dummy(int n) {
02   int c = 0;
03   for (int i=1; i<n; i++)
04     for (int j=0; j<i; j++)
05       c += i + j;
06 }
```

รหัสที่ 3-2 เมท็อด `dummy` ใช้ประกอบตัวอย่างการวิเคราะห์

สัญกรณ์เชิงเส้นกำกับ

หากเราสามารถหาวิธีการทำงานในเมท็อดเดียวกันได้มากกว่าหนึ่งวิธี หรือสามารถหาวิธีการจัดเก็บข้อมูลที่ต่างกันหลายวิธีที่ส่งผลให้การทำงานในเมท็อดเดียวกันแตกต่างกันไป สิ่งที่น่าสนใจก็คือวิธีใดดีกว่ากัน เราสามารถนำฟังก์ชันของเวลาการทำงานที่หามาได้ของแต่ละวิธีมาเปรียบเทียบกัน สิ่งที่เราสนใจเปรียบเทียบคืออัตราการเติบโตของฟังก์ชันว่า ตัวใดโตเร็วกว่ากันเมื่อข้อมูลมีปริมาณมาก โตเร็วกว่าหมายความว่า เมื่อเพิ่มจำนวนข้อมูลในปริมาณเท่ากันจะส่งผลให้ใช้เวลาการทำงานมากกว่า ฟังก์ชันที่โตช้ากว่าก็น่าจะเป็นสิ่งที่ต้องการ

กำหนดให้ $f(n)$ และ $g(n)$ คือ สองฟังก์ชันที่ต้องการนำมาเปรียบเทียบ เราเขียน $f(n) \prec g(n)$ เพื่อแทนว่า $f(n)$ โตช้ากว่า $g(n)$ โดยมีนิยามดังนี้

$$f(n) \prec g(n) \quad iff \quad \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

ในทางกลับกัน ถ้าค่าลิมิตข้างบนนี้มีค่าเป็นอนันต์ แสดงว่า $f(n)$ โตเร็วกว่า $g(n)$ แต่ถ้าค่าลิมิตนี้เป็นค่าคงตัวอื่นที่ไม่ใช่ 0 และ อนันต์ ก็เรียกว่า ทั้ง $f(n)$ และ $g(n)$ โตในอัตราพอกัน

ตัวอย่างที่ 3-1 จงเปรียบเทียบฟังก์ชัน 0.5^n, 1, $\log n$, n และ 10^n

- $0.5^n \prec 1 \prec \log n$ เพราะว่า 0.5^n เป็นฟังก์ชันซึ่งนอกจากจะไม่โตแล้ว ยังมีค่าลดลงเรื่อย ๆ แต่ 1 นั้นเป็นฟังก์ชันนิ่ง ๆ ไม่เพิ่มไม่ลด ในขณะที่ $\log n$ เป็นฟังก์ชันที่โต
- มาดู $\log n$ กับ n จะได้ว่า $\lim_{n\to\infty} \frac{\log n}{n} = \lim_{n\to\infty} \frac{(1/\ln 10)(1/n)}{1} = 0$ ดังนั้น $\log n \prec n$
- จากข้อบนย่อมได้ว่า $10^{\log n} \prec 10^n$ ดังนั้น $n \prec 10^n$

ดังนั้น $0.5^n \prec 1 \prec \log n \prec n \prec 10^n$

เราสามารถเขียนความสัมพันธ์ของฟังก์ชันในแง่ของอัตราการเติบโตได้ โดยการใช้สัญกรณ์เชิงเส้นกำกับ (asymptotic notations) ซึ่งจะช่วยให้เขียนบรรยายฟังก์ชันได้ง่าย และทำให้วิเคราะห์เวลาการทำงานได้ง่ายขึ้นด้วย จะขอนำเสนอสัญกรณ์ 5 ตัวคือ o, ω, Θ, O และ Ω โดยเราจะใช้ตัว O และ Θ กันมากในหนังสือเล่มนี้ ส่วนตัวอื่นเป็นตัวประกอบ และเพื่อความง่ายในการนำเสนอ กำหนดให้ฟังก์ชันต่าง ๆ ที่กล่าวถึงเป็นฟังก์ชันที่ให้ค่าไม่ติดลบ เพราะกำลังสนใจฟังก์ชันที่แทนเวลาการทำงาน

โอเล็ก

เริ่มด้วยสัญกรณ์โอเล็ก เขียนแทนด้วย o ซึ่งมีนิยามดังนี้

$$o(g(n)) = \left\{ f(n) \mid \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \right\}$$

นั่นคือ $o(g(n))$ เป็นเซตของฟังก์ชันทั้งหลายที่โตช้ากว่า $g(n)$ ดังนั้นจากตัวอย่างก่อนหน้านี้ จะได้ว่า $\log n \in o(n)$, $n \in o(10^n)$ เป็นต้น

โอเมกาเล็ก

ในทางกลับกัน นิยามให้ $\omega(g(n))$ คือ เซตของฟังก์ชันทั้งหลายที่โตเร็วกว่า $g(n)$ ดังนี้

$$\omega(g(n)) = \left\{ f(n) \mid \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \right\}$$

ซึ่งก็เห็นได้ชัดเจนว่า $f(n) \in \omega(g(n))$ ก็ต่อเมื่อ $g(n) \in o(f(n))$

ทีตาใหญ่

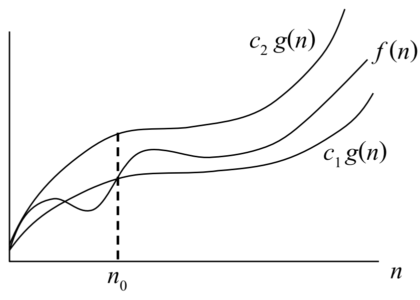
สำหรับกรณีที่ฟังก์ชันโตด้วยอัตราเดียวกัน มีสัญกรณ์ให้คือ Θ ซึ่งมีนิยามดังนี้

$$\Theta(g(n)) = \left\{ f(n) \mid \lim_{n\to\infty} \frac{f(n)}{g(n)} = c, c \neq 0, c \neq \infty \right\}$$

หรือจะเขียนนิยามแบบไม่ต้องยุ่งกับลิมิตก็จะได้แบบนี้

$\Theta(g(n)) = \{ f(n) \mid$ มีค่าคงตัวบวก c_1, c_2 และ n_0 ที่ทำให้ $c_1g(n) \leq f(n) \leq c_2g(n)$ เมื่อ $n \geq n_0$ $\}$

เขียนซะยืดยาวอย่างนี้ ก็เพราะนิยามแบบนี้มองเห็นภาพได้ง่ายกว่า พิจารณารูปที่ 3–1 ถ้า $f(n) \in \Theta(g(n))$ ก็แสดงว่า $g(n)$ เป็นฟังก์ชันที่กำหนดขอบเขตการเติบโตของ $f(n)$ ทั้งขอบเขตบนและขอบเขตล่าง เมื่อ n มีค่ามากพอ (คือเมื่อมีค่าตั้งแต่ n_0 เป็นต้นไป) ค่า c_1 และ c_2 เป็นแค่ตัวคูณ $g(n)$ เพื่อให้ขอบเขตล่างและบนมีรูปแบบการเติบโตคล้าย $g(n)$ เพียงแต่เอียงลงและขึ้นเล็กน้อย รูปที่ 3–1 แสดงให้เห็นว่า $f(n)$ จะไม่หลุดออกนอกขอบเขตล่างและบนนี้เลย เมื่อ $n \geq n_0$ เราเรียก $g(n)$ ว่าเป็นฟังก์ชันกำหนดขอบเขตที่กระชับของ $f(n)$



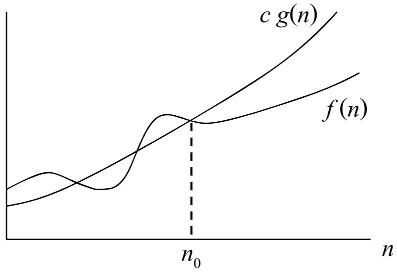
รูปที่ 3–1 $f(n) \in \Theta(g(n))$

โอใหญ่

โอใหญ่ไม่ได้มีไว้เพื่อให้ตรงข้ามกับโอเล็ก เราเขียน $f(n) \in O(g(n))$ เพื่อบอกว่า $f(n)$ เป็นฟังก์ชันที่โต<u>ไม่เร็วกว่า</u> $g(n)$ นั่นคือ $O(g(n)) = o(g(n)) \cup \Theta(g(n))$ เป็นเซตที่รวมฟังก์ชันที่โตช้ากว่าและที่โตเท่ากับ $g(n)$ หรือเขียนเป็นนิยามได้อีกแบบหนึ่งดังนี้

$\mathrm{O}(g(n)) = \{ f(n) \mid$ มีค่าคงตัวบวก c และ n_0 ที่ทำให้ $f(n) \le cg(n)$ เมื่อ $n \ge n_0 \}$

หมายความว่า ถ้า $f(n) \in \mathrm{O}(g(n))$ แสดงว่า การเติบโตของ $f(n)$ ถูกกำหนดขอบเขตด้านบนด้วยลักษณะการเติบโตของ $g(n)$ นั่นคือมีค่าคงตัวบวก c ที่ $f(n) \le cg(n)$ แสดงเป็นตัวอย่างได้ดังรูปที่ 3–2



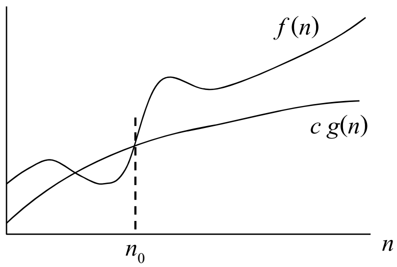
รูปที่ 3–2 $f(n) \in \mathrm{O}(g(n))$

โอเมกาใหญ่

เรามีแบบโตช้ากว่า (o) โตเร็วกว่า (ω) โตเท่ากัน (Θ) และโตไม่เร็วกว่า (O) ก็ต้องปิดท้ายด้วยแบบโตไม่ช้ากว่า เราเขียน $f(n) \in \Omega(g(n))$ เพื่อบอกว่า $f(n)$ เป็นฟังก์ชันที่โตไม่ช้ากว่า $g(n)$ นั่นคือ $\Omega(g(n)) = \omega(g(n)) \cup \Theta(g(n))$ เป็นเซตที่รวมฟังก์ชันที่โตเร็วกว่าและที่โตเท่ากับ $g(n)$ หรือเขียนเป็นนิยามได้อีกแบบหนึ่งดังนี้

$\Omega(g(n)) = \{ f(n) \mid$ มีค่าคงตัวบวก c และ n_0 ที่ทำให้ $cg(n) \le f(n)$ เมื่อ $n \ge n_0 \}$

หมายความว่า ถ้า $f(n) \in \Omega(g(n))$ แสดงว่า การเติบโตของ $f(n)$ ถูกกำหนดขอบเขตด้านล่างด้วยลักษณะการเติบโตของ $g(n)$ นั่นคือมีค่าคงตัวบวก c ที่ $cg(n) \le f(n)$ แสดงเป็นตัวอย่างได้ดังรูปที่ 3–3



รูปที่ 3–3 $f(n) \in \Omega(g(n))$

ตัวอย่างที่ 3-2 จงแสดงให้เห็นจริงว่า $2n^2 + 500n + 1000\log n = O(n^2)$

ต้องหาค่า c และ n_0 ที่ทำให้ $2n^2 + 500n + 1000\log n \le cn^2$ เป็นจริงเสมอเมื่อ $n \ge n_0$ ถ้าให้ $c = 1502$ ก็สบายใจได้เลยว่า อสมการนี้เป็นจริงแน่เมื่อ $n \ge 1$

ตัวอย่างที่ 3-3 จงแสดงให้เห็นจริงว่า $2n^2 + 500n + 1000\log n = O(n^{200})$

จากผลของตัวอย่างที่ 3-2 $2n^2 + 500n + 1000\log n = O(n^2)$ และความจริงที่แทบไม่ต้องแสดงให้เห็นว่า $n^2 \le n^{200} = O(n^{200})$ ดังนั้น $2n^2 + 500n + 1000\log n = O(n^{200})$

ตัวอย่างที่ 3-4 จงแสดงให้เห็นจริงว่า $2n^2 + 500n + 1000\log n = \Omega(n^2)$

ต้องหาค่า c และ n_0 ที่ทำให้ $cn^2 \le 2n^2 + 500n + 1000\log n$ เมื่อ $n \ge n_0$ ให้ $c = 1$ และ $n_0 = 1$ ก็จะได้ $n^2 \le 2n^2 + 500n + 1000\log n$ เมื่อ $n \ge 1$ ดังนั้น $2n^2 + 500n + 1000\log n = \Omega(n^2)$

ตัวอย่างที่ 3-5 จงแสดงให้เห็นจริงว่า $2n^2 + 500n + 1000\log n = \Theta(n^2)$

ตัวอย่างที่ 3-2 แสดงให้เห็นขอบเขตบนว่า $2n^2 + 500n + 1000\log n = O(n^2)$ ส่วนตัวอย่างที่ 3-4 แสดงให้เห็นขอบเขตล่างเป็น $\Omega(n^2)$ จึงสรุปได้ว่า $2n^2 + 500n + 1000\log n = \Theta(n^2)$

ตัวอย่างที่ 3-6 จงแสดงให้เห็นจริงว่า $\sum_{i=1}^{n} i^k = \Theta\left(n^{k+1}\right)$ โดยที่ k เป็นค่าคงตัว

เริ่มด้วยขอบเขตบนก่อน เนื่องจากภายในผลบวก i มีค่าตั้งแต่ 1 ถึง n แสดงว่า $i \le n$ สรุปได้ว่า $i^k \le n^k$ ดังนั้นเมื่อรวมทุก ๆ i ตั้งแต่ 1 ถึง n ย่อมได้ว่า $\sum_{i=1}^{n} i^k \le \sum_{i=1}^{n} n^k = n^{k+1} = O(n^{k+1})$ สำหรับขอบเขตล่าง ถ้าเราหาผลบวกของ i^k สำหรับทุก ๆ i ตั้งแต่ $\lceil n/2 \rceil$ ถึง n ย่อมได้ค่าไม่มากกว่าผลบวกที่ต้องการหา นั่นคือ $\sum_{i=1}^{n} i^k \ge \sum_{i=\lceil n/2 \rceil}^{n} i^k$ แทน i^k ในผลบวกทางขวาด้วย $(n/2)^k$ จะได้

$$\sum_{i=\lceil n/2 \rceil}^{n} i^k \ge \sum_{i=\lceil n/2 \rceil}^{n} (n/2)^k \ge (n/2)^{k+1} = (1/2)^{k+1} n^{k+1} = \Omega(n^{k+1})$$

จากที่แสดงให้เห็นว่า $\sum_{i=1}^{n} i^k = O\left(n^{k+1}\right)$ และ $\sum_{i=1}^{n} i^k = \Omega\left(n^{k+1}\right)$ ดังนั้น $\sum_{i=1}^{n} i^k = \Theta\left(n^{k+1}\right)$

ตัวอย่างที่ 3-7 จงแสดงให้เห็นจริงว่า $\log n! = \Theta(n \log n)$

ขอแสดงให้เห็นจริงว่า $\log n! = O(n \log n)$ จากนิยามของแฟกทอเรียล $n! = n \cdot (n-1) \ldots 2 \cdot 1$ ถ้าแทนทุก ๆ พจน์ทางขวาด้วย n จะได้ว่า $n! \le n^n$ หาค่า log ได้ $\log n! \le n \log n = O(n \log n)$

คราวนี้จะแสดงว่า $\log n! = \Omega(n \log n)$ จากนิยามของแฟกทอเรียล $n! = n\cdot(n-1) \ldots 2\cdot 1$ ถ้าแทนพจน์ $n, (n-1), \ldots, \lfloor n/2\rfloor+1$ ด้วย $(n/2)$ และแทนพจน์ $\lfloor n/2\rfloor, (\lfloor n/2\rfloor-1), \ldots, 2, 1$ ด้วย 1 จะได้ว่า $n! \geq (n/2)^{n/2}$ หาค่า log จะได้ $\log n! \geq (n/2)\log(n/2) = \Omega(n \log n)$ เนื่องจากขอบเขตบนและขอบเขตล่างเหมือนกัน แสดงว่า $\log n! = \Theta(n \log n)$

ตัวอย่างที่ 3-8 จงแสดงให้เห็นจริงว่า $\log_a n = \Theta(\log_b n)$ สำหรับค่าคงตัว $a, b > 1$

เนื่องจากเราสามารถแปลงฐานของ log จาก a เป็น b ได้ จากสูตร $\log_a n = (\log_b n) / (\log_b a)$ ดังนั้น สรุปได้ว่า $\log_a n = \Theta(\log_b n)$ เพราะ $\log_b a$ เป็นค่าคงตัว

บางคนอาจสังเกตและเกิดความสงสัยมาตั้งแต่อ่านตัวอย่างที่ผ่านมาแล้วว่า ตอนแรกก็นิยามให้สารพัด o, ω, Θ, O และ Ω เป็นเซต แล้วก็เขียนสวย ๆ มาตลอด เช่น $f(n) \in \Theta(g(n))$ แล้วอยู่ดี ๆ ก็มาใช้เครื่องหมาย = แทน ∈ เหตุผลก็คือคนในวงการเขาเขียนกันแบบนี้ มันชินตาและสะดวกดี (ไม่ต้องเปลี่ยนฟอนต์บ่อย)

การวิเคราะห์เชิงเส้นกำกับ

การวิเคราะห์เวลาการทำงานเชิงเส้นกำกับของเมท็อดคือการหาฟังก์ชันแสดงจำนวนครั้งที่คำสั่งพื้นฐานในเมท็อดทำงานในรูปแบบของสัญกรณ์เชิงเส้นกำกับ ตัวอย่างเช่น เราได้วิเคราะห์กันแล้วว่าเมท็อด `remove` ของ `ArrayCollection` สั่งงานคำสั่งพื้นฐานเป็นจำนวนไม่เกิน $8 + 3n$ ครั้ง โดยที่ n คือจำนวนข้อมูลในคอลเล็กชัน ซึ่งเราสามารถเขียนแบบเชิงเส้นกำกับได้ว่า "`remove` ใช้เวลาการทำงานเป็น $O(n)$" เขียนแค่นี้ก็แสดงให้เห็นว่า เวลาการทำงานมีอัตราการเติบโตไม่เร็วกว่าฟังก์ชันเชิงเส้น "ไม่เร็วกว่า" นี้มาจากนิยามของ O ซึ่งระบุถึงขอบเขตด้านบนของอัตราการเติบโต ให้สังเกตว่า เราจะสรุปว่า `remove` ใช้เวลาการทำงานเป็น $\Theta(n)$ ไม่ได้ เพราะตัว Θ หมายความว่าขอบเขตบนและล่างต้องเท่ากัน แต่ `remove` นั้นไม่ได้เป็นเชิงเส้นตลอด ทั้งนี้เพราะขึ้นกับตำแหน่งของข้อมูลตัวที่ถูกลบด้วย การวิเคราะห์ที่ได้ทำมานั้นเป็นกรณีที่ `remove` ทำงานช้าสุด คือกรณีหาไม่พบ หรือกรณีที่หาพบตัวสุดท้าย จึงต้องสรุปว่า `remove` ใช้เวลาการทำงานกรณีช้าสุดเป็น $\Theta(n)$ ซึ่งก็คือการสรุปว่า ใช้เวลาการทำงานเป็น $O(n)$

ตัวอย่างที่ 3-9 เราได้วิเคราะห์เมท็อด `dummy` ในรหัสที่ 3-2 (หน้าที่ 42) ว่าสั่งงานคำสั่งในบรรทัดที่ 5 ซึ่งคือคำสั่งตัวแทนเป็นจำนวน $n(n-1)/2$ ครั้ง จึงสรุปได้ว่า `dummy` ใช้เวลาทำงานเป็น $\Theta(n^2)$ หรือจะเขียนเป็น $O(n^2)$ ก็ไม่ผิด

<u>ตัวอย่างที่ 3-10</u> รหัสที่ 3-3 แสดงวิธีการค้นหาข้อมูลในแถวลำดับที่เรียกว่า *การค้นหาแบบทวิภาค* (binary search) ซึ่งเป็นการค้นหาข้อมูลที่รวดเร็ว โดยมีกฎเกณฑ์ว่า ข้อมูลที่เก็บ ต้องเรียงลำดับจากน้อยไปมาก (จากมากมาน้อยก็ได้ แต่รายละเอียดบางอย่างก็ต้องเปลี่ยนไป) เพื่อให้ง่ายต่อการนำเสนอ เราจะค้นข้อมูลในแถวลำดับที่เก็บ `int` ซึ่งสามารถนำข้อมูลมาเปรียบเทียบได้ด้วย < == และ >

การค้นหาแบบทวิภาคอาศัยความสามารถในการขจัดข้อมูลที่ไม่ใช่ข้อมูลที่ต้องการ ออกจากการพิจารณาได้ทีละครึ่ง หลังจากเปรียบเทียบกับข้อมูลตัวตรงกลางของช่วงที่กำลังค้น ช่วงของข้อมูลที่เราสนใจค้นถูกกำหนดโดยตัวแปรสองตัวคือ `left` และ `right` ซึ่งระบุตำแหน่งเริ่มต้น และตำแหน่งสุดท้ายของช่วงตามลำดับ ตอนเริ่มต้น (บรรทัดที่ 2) ให้ `left` มีค่า 0 และ `right` มีค่า `data.length-1` ซึ่งแทนช่องสุดท้าย จากนั้นเข้าวงวนค้นไปเรื่อย ๆ ตราบเท่าที่ `left<=right` หรือตีความได้ว่า ตราบเท่าที่ยังมีข้อมูลเหลือให้ค้นในช่วง ภายในวงวน `while` เริ่มด้วยการคำนวณตำแหน่งตรงกลางของช่วงข้อมูลที่สนใจ (บรรทัดที่ 4) แล้วหยิบตัวตรงกลางนั้นมาเปรียบเทียบกับ `e` ถ้าเท่ากัน แสดงว่าพบแล้ว ก็คืนตำแหน่งตรงกลางนั้น ถ้า `e` มีค่ามากกว่า แสดงว่า ทุก ๆ ตัวทางซ้ายของตัวตรงกลางไม่มีทางเท่ากับ `e` แน่ ๆ ต้องค้นต่อในช่วงขวา ซึ่งทำได้โดยเลื่อน `left` ให้มาอยู่ถัดจากตัวตรงกลางไปหนึ่งตำแหน่ง (บรรทัดที่ 7) แล้วกลับไปค้นต่อ แต่ถ้า `e` มีค่าน้อยกว่าตัวตรงกลาง ก็ต้องไปค้นต่อในช่วงซ้ายของตัวตรงกลางโดยการเปลี่ยนค่า `right` (บรรทัดที่ 9) กระทำการค้นในวงวนไปจนกว่าจะพบ (บรรทัดที่ 5) หรือหลุดจากวงวนซึ่งแสดงว่า หา `e` ไม่พบ ก็ให้คืน –1 กลับไป

```
01 static int binarySearch(int[] data, int e) {
02   int left = 0, right = data.length - 1;
03   while( left <= right ) {
04     int mid = (left + right) / 2;
05     if (e == data[mid]) return mid;
06     if (e > data[mid])
07       left = mid + 1;
08     else
09       right = mid - 1;
10   }
11   return -1;
12 }
```

คำนวณตำแหน่งกลางของช่วง

พบข้อมูลแล้ว

e มากกว่า ค้นต่อช่วงขวา

e น้อยกว่า ค้นต่อช่วงซ้าย

หลุดจากวงวน แสดงว่าหาไม่พบ

รหัสที่ 3-3 การค้นหาแบบทวิภาค

ก่อนจะวิเคราะห์เวลาการทำงานก็ต้องเลือกคำสั่งตัวแทนที่เราจะนับ ดูที่รหัสที่ 3-3 แล้วถามตัวเองว่า บรรทัดใดที่ทำงานมากที่สุด ตัดบรรทัดที่ 2 กับ 11 ทิ้งได้ เพราะทำแค่ครั้งเดียว บรรทัดที่ 6 ทำน้อยกว่าบรรทัดที่ 5 บรรทัดที่ 5 และ 4 ทำเท่ากัน ถ้าหมุนอยู่ในวงวนจนหลุดออกมาเพราะหาไม่เจอ บรรทัดที่ 3 ก็ทำมากกว่าบรรทัดที่ 4 หนึ่งครั้ง แต่ถ้าค้นพบแล้ว `return` ตรงบรรทัดที่ 5 บรรทัดที่ 3 ก็ทำเป็นจำนวนเท่ากับบรรทัดที่ 4 จึงพอสรุปได้ว่า การเปรียบเทียบ `left<=right` ใน

บรรทัดที่ 3 เป็นคำสั่งตัวแทนได้ และเนื่องจากเวลาการทำงานขึ้นกับว่า ค้นพบข้อมูลที่ตำแหน่งใดด้วย ดังนั้นเราจะวิเคราะห์เวลาการทำงานกรณีที่ทำงานช้าสุด ซึ่งในการค้นหาแบบทวิภาคก็คือกรณีที่ค้นไม่พบข้อมูล

กลับมานับคำสั่ง `left<=right` กัน ให้สังเกตว่า ค่าของ `right-left+1` คือจำนวนข้อมูลในช่วงของแถวลำดับที่เราสนใจค้น ผ่านไปหนึ่งรอบจำนวนข้อมูลในช่วงจะลดลงครึ่งหนึ่ง เมื่อการทำงานหมุนไปเรื่อย ๆ จนถึงรอบสุดท้าย การเปรียบเทียบครั้งสุดท้ายเกิดขึ้นตอนที่ `right-left+1<1` แล้วก็หลุดจากวงวน ถ้าให้ n คือจำนวนข้อมูล รอบแรก `right-left+1` จะมีค่าเท่ากับ n เพื่อให้ง่ายกับการวิเคราะห์ขอกำหนดให้ $n = 2^k$ โดยที่ k เป็นจำนวนเต็ม

รอบที่	1	`right-left+1`	มีค่าเท่ากับ	n	
"	2	"	"	$n / 2^1$	
"	3	"	"	$n / 2^2$	
		. . .			
"	k	"	"	$n / 2^{k-1}$	
"	$k+1$	"	"	$n / 2^k$	$= 1$
"	$k+2$	"	"	$n / 2^{k+1}$	$= 0$

ดังนั้นคำสั่ง `left<=right` ทำงานในกรณีช้าสุดเป็นจำนวน $2 + k = 2 + \log_2 n$ ครั้ง สรุปได้ว่าการค้นหาแบบทวิภาคใช้เวลาเป็น $O(\log n)$

กลับมาวิเคราะห์เมท็อดต่าง ๆ ใน `ArrayCollection` กันดีกว่า (รหัสที่ 3-4) เราวิเคราะห์ `remove` ไปแล้วว่า ใช้เวลา $O(n)$ ซึ่งก็เท่ากับของ `contains` เพราะทั้งคู่เรียก `indexOf` ที่ใช้เวลา $O(n)$ เมท็อด `size` `isEmpty` และ constructor ทำงานด้วยเวลาคงตัวไม่ขึ้นกับปริมาณข้อมูลเลย แบบนี้เขียนว่า ใช้เวลาเป็น $\Theta(1)$ หรือจะเขียน $O(1)$ ก็ได้

มาดูที่ `add` บรรทัดที่ 10 และ 12 ทำงานด้วยเวลาคงตัว ส่วน `ensureCapacity` นั้นใช้เวลา $\Theta(1)$ ถ้าไม่ต้องขยายแถวลำดับ แต่ถ้าต้องขยายก็จะใช้เวลา $\Theta(n)$ เพราะการ `new` ที่บรรทัดที่ 38 ต้องจองแถวลำดับใหม่ขนาด $2n$ ช่อง และต้องย้ายข้อมูลจากแถวเก่ามาแถวใหม่ในบรรทัดที่ 40 เป็นจำนวน n ครั้ง สรุปได้ว่า `add` ใช้เวลา $O(n)$

หลายคนอาจสงสัยว่า การขยายแถวลำดับของ `add` นาน ๆ ทำที ถ้าเราเริ่มด้วยขนาด 1 ช่อง แล้วเพิ่มข้อมูลไปเรื่อย ๆ สัก 9 ครั้ง จะเกิดการขยายในการเพิ่มครั้งที่ 2, 3, 5 และ 9 (ดูรูปที่ 3–4) และถ้าเพิ่มต่อไปกว่าจะขยายอีกทีก็เป็นการเพิ่มตัวที่ 17, 33, หรือพอสรุปได้ว่า การเพิ่มครั้งที่ $2^k + 1$ จะมีการขยายแถวลำดับ โดยที่ $k = 1, 2, 3, 4, \ldots$

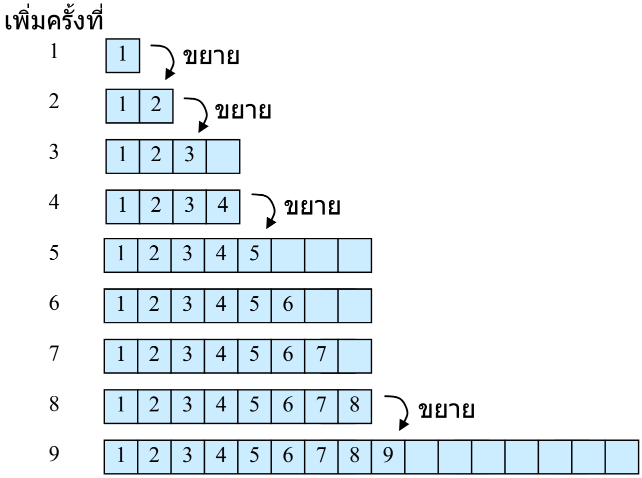
```
01 public class ArrayCollection implements Collection {
```

```
02   private Object[] elementData;
03   private int      size;
04
05   public ArrayCollection() {
06     elementData = new Object[1];        Θ(1)
07     size = 0;
08   }
09   public void add(Object e) {
10     if(e == null) throw new IllegalArgumentException();
11     ensureCapacity(size + 1);         O(n)
12     elementData[size++] = e;
13   }
14   public int size() {
15     return size;           Θ(1)
16   }
17   public boolean isEmpty() {
18     return size == 0;          Θ(1)
19   }
20   public boolean contains(Object e) {
21     return indexOf(e) != -1;         O(n)
22   }
..   ...
35   private void ensureCapacity(int capacity) {
36     if (capacity > elementData.length) {
37       int s = Math.max(capacity, 2*elementData.length);
38       Object[] arr = new Object[s];
39       for(int i = 0; i < size; i++)
40         arr[i] = elementData[i];         ถ้าไม่ขยาย Θ(1)
41       elementData = arr;                 แต่ถ้าต้องขยาย Θ(n)
42     }                                    สรุปแล้วเป็น O(n)
43   }
44 }
```

รหัสที่ 3-4 คลาส `ArrayCollection`

หากเราลองสมมติสถานการณ์การเพิ่มข้อมูลเป็นจำนวน n ตัว คำถามคือเวลาการทำงานสะสมที่ใช้ทั้งหมดจะเป็นเท่าไร ขอใช้คำสั่งการนำข้อมูลใส่แถวลำดับเป็นคำสั่งตัวแทนเพื่อการวิเคราะห์ (คือบรรทัดที่ 12 กับบรรทัดที่ 40 ในรหัสที่ 3-4) ข้อสังเกตที่ว่า การเพิ่มครั้งที่ $2^k + 1$ ต้องมีการขยายแถวลำดับ จากขนาด 2^k เป็น 2^{k+1} (รูปที่ 3–4) ทำให้รู้ว่า ต้องเกิดภาระการย้ายข้อมูลจากแถวลำดับเก่าไปแถวใหม่จำนวน 2^k ตัว คือทำบรรทัดที่ 40 จำนวน 2^k ครั้งในการเพิ่มครั้งที่ $2^k + 1$ ดังนั้นเฉพาะภาระการขยาย จะทำบรรทัดที่ 40 ทั้งสิ้น $2^0+2^1+2^2+2^3+\ldots+2^{\lfloor \lg n \rfloor} = 2^{\lfloor \lg n \rfloor + 1} - 1 \le 2n-1$ ครั้ง เมื่อนำไปรวมกับการทำงานที่บรรทัดที่ 12 ซึ่งคือการนำข้อมูลใหม่เข้าไปต่อตัวท้าย (1 ครั้งต่อการเพิ่ม 1 ตัว) อีก n ครั้ง เป็นภาระรวมเท่ากับ $3n - 1$ สรุปได้ว่า การเพิ่มข้อมูลจำนวน n ตัวใช้เวลาการทำงานเป็น $O(n)$ ถ้าเรานำมาหารด้วยจำนวนครั้งของการเรียก `add` ซึ่งคือ n ครั้ง ก็ย่อมตีความได้ว่า ถัวเฉลี่ยแล้วการเรียก `add` หนึ่งครั้งใช้เวลา $O(1)$ นี่เป็นตัวอย่างของการวิเคราะห์ที่แสดงให้เห็นว่า บางครั้งเรา

วิเคราะห์การเรียกเมท็อดหนึ่งครั้ง โดยพฤติกรรมของเมท็อดไม่แน่นอนบางครั้งเร็วบางครั้งช้า ก็ต้องเลือกวิเคราะห์กรณีทำงานช้าสุด ๆ แต่กลับพบว่า ความจริงแล้วกรณีช้าสุด ๆ นั้น นาน ๆ เกิดที ดังนั้นหากวิเคราะห์แบบถัวเฉลี่ยดังที่ทำมา ซึ่งสะท้อนความเป็นจริงของการใช้โครงสร้างข้อมูลที่สร้างขึ้นเพื่อเรียกใช้บริการหลาย ๆ ครั้ง จะพบว่า มีประสิทธิภาพการทำงานแบบถัวเฉลี่ยที่ดี



รูปที่ 3–4 ตัวอย่างการขยายแถวลำดับเมื่อเพิ่มข้อมูลจำนวน 9 ตัว

แบบฝึกหัด

1. จงแสดงว่า $f(n)$ มีความสัมพันธ์เชิงเส้นกำกับอย่างไรกับ $g(n)$ ในตารางข้างล่างนี้

$f(n)$	$g(n)$
10^{100}	n
$\log_2 n$	$\sqrt{n}$
$3n^5 + 50\sqrt{n}$	$100n^5$
$n!$	2^n
$4^{\lfloor n/2 \rfloor}$	2^n
2^{n+1}	3^n
$2^{\log_2 n}$	n

2. ให้ A, B, และ C แทนขั้นตอนวิธีการค้นหาข้อมูลซึ่งใช้เวลาการทำงานเป็น $O(n)$, $\Theta(n)$, และ $\Omega(n)$ ตามลำดับ อยากทราบว่า ถ้า n มีค่ามาก เราควรค้นหาข้อมูลด้วยวิธีใด

3. อธิบายความหมายของ $O(1)$, $\Theta(1)$, $\Omega(1)$, $o(1)$, และ $\omega(1)$

4. $O(1)$ ต่างกับ $O(10^6)$ หรือไม่ อย่างไร

5. ให้ $f(n) = 100n \log_2 n$ และ $g(n) = n^2$ อยากทราบว่า $f(n)$ มีค่าน้อยกว่า $g(n)$ เมื่อ n มีค่าเท่าใด

6. มะลิรู้แล้วว่า $\log n! \geq (n/2) \log (n/2)$ จงช่วยมะลิพิสูจน์ต่อว่า $\log n! = \Omega(n \log n)$

7. หากเราเปลี่ยนให้ `remove` ใน `ArrayCollection` เป็นดังที่แสดงข้างล่างนี้ ซึ่งค้นข้อมูลที่จะลบด้วย `indexOf` ถ้าพบได้ตำแหน่ง `i` กลับมาแล้ว ให้ลบด้วยการย้ายข้อมูลทุกตัวทางขวาของ `i` มาทางซ้ายตัวละหนึ่งตำแหน่ง จงวิเคราะห์เวลาการทำงานเชิงเส้นกำกับ

```
public class ArrayCollection implements Collection {
  public void remove(Object e) {
    int i = indexOf(e);
    if (i != -1) {
      while (i < size-1) elementData[i] = elementData[i+1];
    }
  }
```

8. พิจารณาการขยายแถวลำดับ `elementData` ภายในเมท็อด `ensureCapacity` ของคลาส `ArrayCollection` (รหัสที่ 3-4) ถ้าเราเปลี่ยนขนาดของแถวลำดับใหม่ที่จองจากเดิมที่ให้ขยายเป็นสองเท่า เปลี่ยนเป็นสามเท่า จะมีผลอย่างไรต่อการเพิ่มข้อมูลในกรณีถัวเฉลี่ย ถ้าเปลี่ยนเป็น 1.1 เท่า จะมีผลอย่างไร และถ้าทุกครั้งที่ขยายแทนที่จะเพิ่มเป็นเท่า ๆ ให้เพิ่มอีก 100 ช่อง จะมีผลอย่างไร จงวิเคราะห์ในแต่ละกรณี

9. ศึกษารายละเอียดการทำงานของเมท็อดต่าง ๆ ในคลังคลาสมาตรฐานของจาวาข้างล่างนี้ จากคู่มือการใช้งาน Java API help file แล้วลองวิเคราะห์ว่า แต่ละเมท็อดน่าจะใช้เวลาเชิงเส้นกำกับเท่าไร

   9.1. `System.arraycopy`
   9.2. `Arrays.equals`
   9.3. `Arrays.fill`
   9.4. `Rectangle.contains`
   9.5. `Math.abs`
   9.6. `Math.toDegrees`
   9.7. `Math.pow`
   9.8. `Color.equals`
   9.9. `BigIntger.multiply`

10. จงวิเคราะห์เวลาการทำงานของเมท็อด `equals` ข้างล่างนี้ ของคลาส `ArrayCollection` ที่ได้นำเสนอในบทที่แล้ว

```
public boolean equals(Object x) {
  if (!(x instanceof ArrayCollection)) return false;
  Object[] a1 = ((ArrayCollection) x).toArray();
```

```
  nextElement:
  for (int i=0; i<size; i++) {
    for (int j=0; j<a1.length; j++) {
      if (elementData[i].equals(a1[j])) {
        a1[j] = null; continue nextElement;
      }
    }
    return false;
  }
  for (int j=0; j<a1.length; j++)
    if (a1[j] != null) return false;
  return true;
}
```

11. จงวิเคราะห์เวลาการทำงานของเมท็อดต่อไปนี้

10.1
```
static int log10(int n) {
  int log = 0;
  for (; n > 0; log++, n /= 10) ;
  return log;
}
```

10.2
```
static int log2(int n) {
  int log = 0;
  for (int k=1; k < n; log++, k += k) ;
  return log;
}
```

10.3
```
static void bubble(int[] d) {
   for (int k = d.length; k > 1; k--) {
     for (int j = 1; j < k; j++) {
       if (d[j] < d[j-1]) swap(d, j-1, j);
     }
   }
}
```

10.4
```
static void merge(Object[] a, Object[] b, Object[] c) {
   int i = 0, j = 0, k = 0;
   while (i<a.length && j<b.length)
     t[k++] = a[i] < b[j] ? a[i++] : b[j++];
   while (i<a.length) t[k++] = a[i++];
   while (j<b.length) t[k++] = b[j++];
}
```

10.5
```
static double[][] add(double[][] a, double[][] b) {
   double[][] c = new double[a.length][a[0].length];
   for (int i=0; i<a.length; i++)
```

```
    for (int j=0; j<a[i].length; j++)
      c[i][j] += a[i][j] + b[i][j];
}
```

10.6
```
static double[][] mult(double[][] a, double[][] b) {
  double[][] c = new double[a.length][b[0].length];
  for (int i=0; i<a.length; i++)
    for (int j=0; j<b[0].length; j++)
      for (int k=0; k<b.length; k++)
        c[i][j] += a[i][k] * b[k][j];
}
```

4 การเก็บข้อมูลแบบโยง

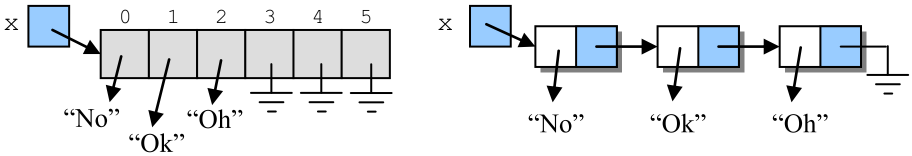
การเก็บข้อมูลแบบโยงเป็นลักษณะการจัดเก็บความสัมพันธ์ของข้อมูลอีกรูปแบบหนึ่ง ที่รองรับการเปลี่ยนแปลงความสัมพันธ์ของข้อมูลได้ดี บทนี้นำเสนอ `LinkedCollection` ซึ่งเป็นวิธีสร้างคอลเล็กชันที่อาศัยการโยงข้อมูลเข้าด้วยกัน เป็นตัวอย่างการจัดเก็บข้อมูลแบบโยงอย่างง่าย และเป็นพื้นฐานสำคัญในการจัดเก็บโครงสร้างข้อมูลที่ซับซ้อนมากขึ้นในบทถัดไป

การโยงข้อมูล

การสร้างคอลเล็กชันด้วยแถวลำดับที่ได้ศึกษามาใน `ArrayCollection` นั้นอาศัยการจัดเก็บข้อมูลในแถวลำดับแบบติดกัน ตั้งแต่ช่องที่ 0 ไปถึงช่องที่ `size-1` แถวลำดับมีข้อดีตรงที่เราสามารถเข้าถึงช่องใด ๆ ได้ในเวลาอันรวดเร็ว ไม่ว่าจะเป็นช่องที่ 0 ช่องที่ 1000 หรือช่องใด ๆ หากเราสนใจข้อมูลในช่องที่ k และอยากได้ข้อมูลตัวถัดไปก็<u>คำนวณ</u>ค่าของ $k + 1$ ก็จะรู้ตำแหน่งของข้อมูลช่องถัดไปทันที อย่างไรก็ตาม กฎที่ให้เก็บข้อมูลติดกันในแถวนั้น ก็เป็นข้อเสียของการเก็บข้อมูลด้วยแถวลำดับ เพราะหากต้องการแทรกข้อมูลหนึ่งตัวระหว่างช่องที่ k กับช่องที่ $k + 1$ ย่อมต้องย้ายข้อมูลในช่องที่ $k + 1$ ถึงข้อมูลตัวสุดท้ายไปทางขวาหนึ่งตำแหน่ง ซึ่งใช้เวลาเป็น $O(n)$ เมื่อ n เป็นจำนวนข้อมูล (อนึ่ง ความต้องการแทรกข้อมูลในลักษณะนี้ยังไม่เกิดขึ้นกับ `ArrayCollection` ที่ได้ศึกษามา แต่อาจเกิดขึ้นกับโครงสร้างข้อมูลแบบอื่น)

เราสามารถจัดเก็บข้อมูลในอีกรูปแบบหนึ่ง โดยให้ข้อมูลแต่ละตัวมีข้อมูลเสริมกำกับไว้<u>จำตำแหน่ง</u>ของข้อมูลตัวถัดไป การจำตำแหน่งของข้อมูลตัวถัดไปนี้เรียกว่า *การโยงข้อมูล* รูปที่ 4-1 ทางซ้ายแสดงตัวอย่างการใช้แถวลำดับ `x` เก็บสตริงสามตัวในช่องที่ 0, 1, และ 2 ส่วนรูปขวาแสดงการเก็บแบบโยง เราสร้าง “ปมข้อมูล” (เรียกว่าเป็น “ก้อนข้อมูล” ก็ได้ แต่จะขอใช้คำว่า “ปม” แทน

เพราะจะตรงกับคำว่า node ซึ่งใช้กันทั่วไปในตำราภาษาอังกฤษ) หนึ่งปมเก็บตัวอ้างอิงสองตัว ตัวหนึ่งไว้อ้างอิงข้อมูลที่เราต้องการเก็บ อีกตัวมีหน้าที่อ้างอิง (ซึ่งก็คือการจำตำแหน่ง) ปมข้อมูลปมถัดไป ในกรณีเป็นปมสุดท้าย ไม่มีปมถัดไปให้อ้างอิง ก็ให้เก็บ `null` แทน ส่วนปมแรกก็ต้องมีตัวแปรตัวหนึ่งอ้างอิง ซึ่งก็เหมือนกับแถวลำดับที่ต้องมีตัวแปรอ้างอิงเช่นกัน (ตัวแปร `x` ในรูปที่ 4-1)



รูปที่ 4-1 การจัดเก็บข้อมูลแบบใช้แถวลำดับ กับแบบโยง

ปมข้อมูล

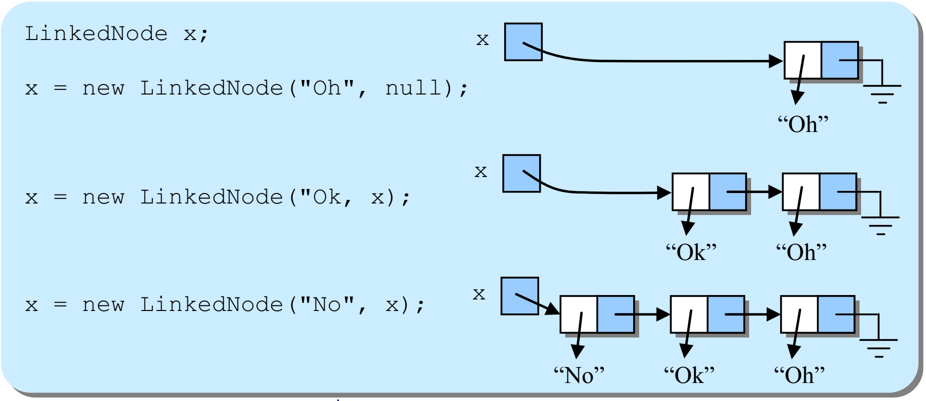
ก่อนจะสร้างคอลเล็กชันด้วยการเก็บข้อมูลแบบโยง เราต้องออกแบบปมข้อมูลก่อน ขอตั้งชื่อคลาสว่า `LinkedNode` ซึ่งมีตัวอ้างอิงสองตัวเป็นสมาชิก ตัวแรกชื่อ `element` มีไว้อ้างอิงข้อมูลแบบ `Object` อีกตัวชื่อ `next` อ้างอิง `LinkedNode` ปมถัดไป มีตัวสร้างหนึ่งตัวทำหน้าที่ตั้งค่าเริ่มต้นให้กับสมาชิกทั้งสอง ดังแสดงในรหัสที่ 4-1

```
class LinkedNode {
  private Object element;
  private LinkedNode next;

  LinkedNode(Object e, LinkedNode n) {
    this.element = e;
    this.next = n;
  }
}
```

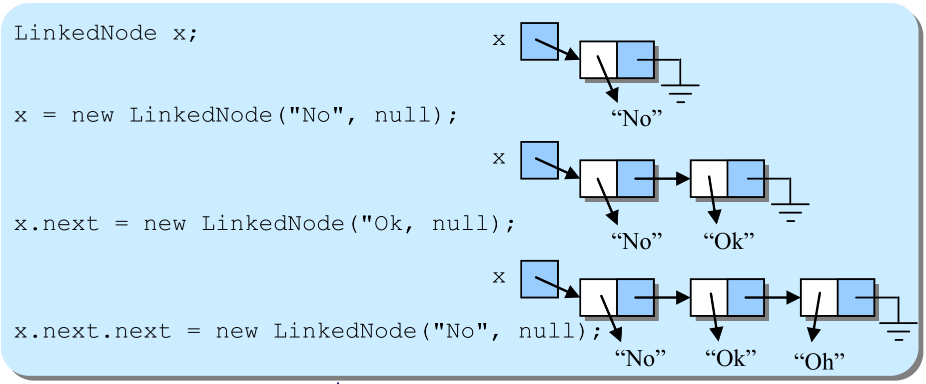
รหัสที่ 4-1 รายละเอียดของคลาส `LinkedNode`

รูปที่ 4-2 และ รูปที่ 4-3 แสดงตัวอย่างวิธีการสร้างปมข้อมูลที่มีการโยงดังรูปที่ 4-1 รูปที่ 4-2 สร้างอ็อบเจกต์ของปมข้อมูลจากด้านขวามาซ้าย ตัวแปร `x` เปลี่ยนการอ้างอิงไปยังปมที่เพิ่งสร้างใหม่ โดย `next` ของปมใหม่จะชี้ไปยังปมที่ `x` เคยชี้ (การตั้งค่าของ `next` กระทำที่ตัวสร้าง)

รูปที่ 4-3 สร้างอ็อบเจกต์ของปมข้อมูลจากด้านซ้ายไปขวา ตัวแปร `x` ชี้ปมทางซ้ายสุดตลอดการสร้าง เมื่อสร้างปมที่สอง ก็ให้ `x.next` ชี้ พอสร้างปมที่สามก็ให้ `x.next.next` ชี้ อย่าลืมว่า `x.next.next` อยู่ทางซ้ายของเครื่องหมาย = ย่อมหมายถึงตัวแปร `next` ของปมที่ `x.next` ชี้ นั่นคือให้ตัวแปร `next` ของปมที่สอง ชี้ไปยังปมตัวขวาสุด



รูปที่ 4-2 ตัวอย่างการสร้างข้อมูลแบบโยง



รูปที่ 4-3 ตัวอย่างการสร้างข้อมูลแบบโยง

คลาส LinkedCollection

`LinkedCollection` ที่นำเสนอในบทนี้คือคลาสที่สร้างคอลเล็กชันด้วยการจัดเก็บข้อมูลแบบโยง ลองคิดดูว่า คลาสนี้ต้องมีสมาชิกประจำอ็อบเจกต์อะไรบ้าง ก็คล้ายกับ `ArrayCollection` คือมีตัวแปร `size` เพื่อเก็บจำนวนข้อมูล และมีตัวแปร `first` ไว้อ้างอิงหรือชี้ปมข้อมูลแรกของการโยง เมื่อออกแบบข้อมูลที่จำเป็นแล้ว ก็ถามตัวเองว่า หลังสร้างอ็อบเจกต์ `LinkedCollection` ตัวอ็อบเจกต์จะมีลักษณะอย่างไร แน่นอนว่า `size` ต้องมีค่าเป็น 0 เพราะยังไม่มีข้อมูลใด ๆ เก็บอยู่เลยในคอลเล็กชัน `first` ก็ต้องไม่ชี้อะไรเลย จึงให้ `first` มีค่าเป็น `null` เพื่อบอกว่าไม่ชี้อะไร รหัสที่ 4-2 แสดงส่วนหนึ่งของคลาส รวมถึงเมท็อด `size()` และ `isEmpty()` ซึ่งใช้ตัวแปร `size` ประกอบการทำงาน

```
01 public class LinkedCollection implements Collection {
02   private static class LinkedNode {
03     private Object element;
04     private LinkedNode next;
05     LinkedNode(Object e, LinkedNode n) {
06       this.element = e;
07       this.next = n;
08     }
09   }
10   private int size;
11   private LinkedNode first;
12
13   public LinkedCollection() {
14     size = 0; first = null;
15   }
16   public int size() {
17     return size;
18   }
19   public boolean isEmpty() {
20     return size == 0;
21   }
..   ...
```

ย้าย LinkedNode มา ซ่อนในคลาสนี้

เก็บจำนวนข้อมูลในคอลเล็กชัน

ชี้ปมข้อมูลแรกของการโยง

คอลเล็กชันว่าง size เป็น 0 และ first ไม่ชี้อะไร

รหัสที่ 4-2 ส่วนหนึ่งของคลาส `LinkedCollection`

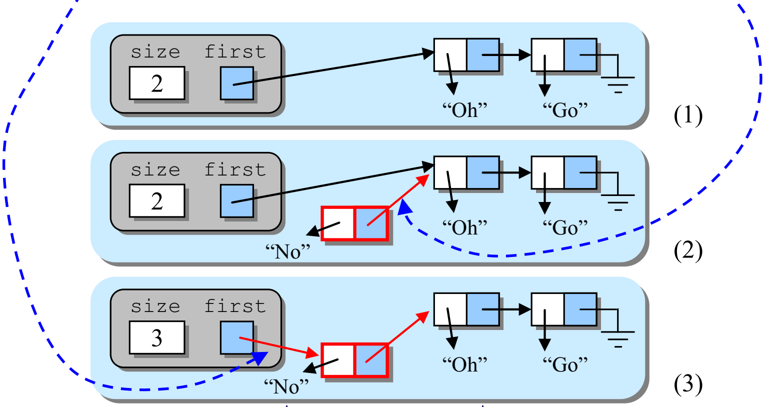
ให้สังเกตว่า เรานำคลาส `LinkedNode` ที่เขียนไว้ก่อนหน้านี้มาแทรกเป็นคลาสภายในแบบ `private` (บรรทัดที่ 2 ถึง 9) เพราะไม่มีการใช้ `LinkedNode` จากนอกคลาสนี้ อีกทั้งยังเป็นการซ่อนรายละเอียดการออกแบบโครงสร้างข้อมูลเพื่อไม่ให้ผู้อื่นเข้าใช้ได้อีกด้วย

เมท็อด `add` (รหัสที่ 4-3) รับข้อมูลมาสร้าง `LinkedNode` แล้วเพิ่มทางด้านหน้าของการโยง บรรทัดที่ 24 รวบรัดทำสองงานในหนึ่งบรรทัด ดูตัวอย่างในรูปที่ 4-4 (1) คือสภาพเดิมก่อนเพิ่ม เมื่อสั่งเพิ่ม "No" บรรทัดที่ 24 จะสร้างปมใหม่ด้วย `new LinkedNode` โดยส่ง "No" ให้เก็บใน `element` และส่งสิ่งที่ `first` ชี้ ซึ่งคือตำแหน่งของปมแรก ไปให้ `next` ชี้ จะได้ดังรูปที่ 4-4 (2) และเมื่อตัวสร้างทำงานเสร็จ ก็คืนตำแหน่งของอ็อบเจกต์ใหม่มาเก็บใน `first` ได้ดังรูปที่ 4-4 (3) ปิดท้ายด้วยการเพิ่มขนาดของคอลเล็กชันในบรรทัดที่ 25 ความจริงแล้วเราจะแทรกปมข้อมูลใหม่ ณ ที่ใดก็ได้ในการโยง แต่การเลือกเพิ่มไว้ด้านหน้านั้นสะดวกและรวดเร็วสุด

มาถึงเมท็อด `contains` เพื่อค้นหาข้อมูล (รหัสที่ 4-4) เริ่มที่บรรทัดที่ 28 ให้ตัวแปร `node` ชี้ปมแรก แล้วเข้าสู่วงวนการค้น โดยจะหลุดจากวงวนเมื่อวิ่งค้นหมดแล้ว (`node` เป็น `null`) หรือเมื่อพบข้อมูล (`node.element.equals(e)` เป็นจริง) แต่ถ้ายังมีข้อมูล และไม่ใช่ตัวที่ต้องการ ก็ให้ `node` ชี้ปมถัดไปโดยใช้ `node=node.next` (บรรทัดที่ 31) ถ้าหลุดจากวงวนโดยที่ `node == null` แสดงว่าไม่พบข้อมูล แต่ถ้าไม่เท่า แสดงว่าพบแล้ว

```
  public void add(Object e) {
    if (e == null) throw new IllegalArgumentException();
    first = new LinkedNode(e, first);
    ++size;
  }
```

รหัสที่ 4-3 add เพิ่มปมข้อมูลไว้ด้านหน้า



รูปที่ 4-4 ตัวอย่างการเพิ่มข้อมูล

```
  public boolean contains(Object e) {
    LinkedNode node = first;

    while (node != null && !node.element.equals(e) ) {
      node = node.next;
    }
    return node != null;
  }
```

ยังไม่หมด

ข้อมูลที่เก็บไม่เท่ากับ e

เลื่อนไปชี้ปมถัดไป

หาพบ ถ้า node ไม่เป็น null

รหัสที่ 4-4 contains ของ LinkedCollection

ขอย้ำว่า คำสั่ง node=node.next เปลี่ยนค่าในตัวแปร node จากเดิมชี้ปมข้อมูลหนึ่ง ให้เลื่อนไปชี้ปมถัดไป เพื่อฝึกความเข้าใจการใช้ตัวแปรชี้ปมข้อมูล อยากให้ลองคิดตอนนี้ว่า คำสั่ง node = node.next.next ทำอะไร ?[1] และ node.next = node ทำอะไร ?[2]

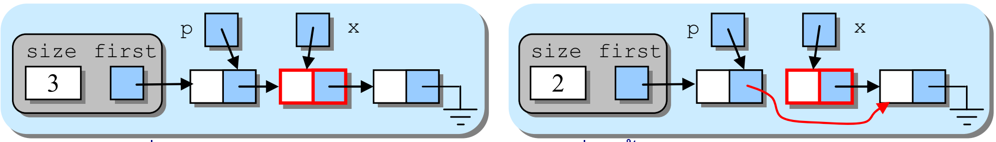
เมท็อด remove มีภาระค้นข้อมูลให้พบแล้วค่อยลบ การลบข้อมูล e ในคอลเล็กชัน ทำได้ด้วยการลบปมข้อมูลที่เก็บ e ออกจากการโยง ถ้า x ชี้ปมข้อมูลที่เราต้องการลบออกจากการโยง เรา

---

[1] node = node.next.next ก็คือการเปลี่ยน node ที่ชี้ปมหนึ่ง เลื่อนไปชี้ตัวถัดจากตัวถัดไป นั่นคือเปลี่ยนไปชี้สองตัวถัดไป ข้อควรระวังก็คือจะเกิด NullPointerException ถ้าไม่มีตัวถัดไป เพราะ node.next เป็น null เรียก .next อีกทีก็เกิด exception

[2] node.next = node ก็คือการเปลี่ยนให้ปมข้อมูลถัดไปเป็นปมตัวเอง เขียนคำสั่งแบบนี้ท่าทางจะทำให้โครงสร้างผิดแปลกไปมาก

ต้องรู้ตำแหน่งของปมก่อนหน้าปม x เพื่อจะได้เปลี่ยน next ของปมนั้น ให้ชี้ข้ามไปชี้ปมถัดจากปม x ถ้าให้ p ชี้ปมก่อนหน้าปมที่ต้องการลบออก คำสั่ง p.next = p.next.next ก็จะทำหน้าที่ดังกล่าว (ดูรูปที่ 4-5) ให้สังเกตว่า p.next ทางซ้ายของเครื่องหมาย = หมายถึงตัวแปร next ของปมที่ p ชี้ ส่วน p.next ทางขวาของเครื่องหมาย = หมายถึงปมที่จะถูกลบ ดังนั้น p.next.next จึงแทนตำแหน่งของปมข้อมูลที่ถัดจากปมที่เราอยากลบ



รูปที่ 4-5 p.next=p.next.next ลบปมที่ x ชี้ จากรูปซ้ายไปเป็นรูปขวา

การลบแบบข้างบนนี้มีข้อจำกัดว่า ถ้าอยากลบปมไหน ปมนั้นต้องมีปมก่อนหน้า ดังนั้น การลบปมแรกจึงต้องจัดการเป็นกรณีพิเศษ (เพราะไม่มีปมก่อนหน้า) รหัสที่ 4-5 เริ่มตรวจสอบว่า ถ้า first เป็น null ก็ไม่ต้องทำอะไร บรรทัดที่ 37 ตรวจว่า ถ้าปมแรกเก็บข้อมูลที่ต้องการลบ ก็เปลี่ยน first ที่ชี้ปมแรก ให้ไปชี้ปมถัดไป แล้วก็ลดค่าของ size แต่ถ้าไม่ใช่ปมแรกที่อยากลบ ก็เริ่มค้นข้อมูล โดยใช้ตัวแปร p ที่เราตั้งใจให้ชี้ปมก่อนปมที่เราสนใจ (p ชี้ปมไหน ก็สนใจปมถัดไป) เนื่องจากเราเริ่มค้นตั้งแต่ตัวที่สอง จึงเริ่มให้ p=first (บรรทัดที่ 41) แล้วเข้าสู่วงวน ที่เลื่อน p ไปชี้ตัวถัดไปตราบเท่าที่ยังมีปมถัดจากที่ p ชี้ และปมนั้นเก็บข้อมูลที่ไม่ใช่ตัวที่อยากลบ เมื่อหลุดจากวงวนแสดงว่าไม่มีปมถัดไป หรือไม่ก็พบตัวที่ต้องลบ เงื่อนไข p.next != null (บรรทัดที่ 45) บอกว่า p.next คือปมที่ต้องลบ จึงลบด้วย p.next=p.next.next แล้วลดค่าของ size (รหัสที่ 4-6 สรุปรายละเอียดของคลาส LinkedCollection)

```
35   public void remove(Object e) {
36     if (first == null) return;
37     if (first.element.equals(e)) {
38       first = first.next;
39       --size;
40     } else {
41       LinkedNode p = first;
42       while (p.next != null && !p.next.element.equals(e)) {
43         p = p.next;
44       }
45       if (p.next != null) {
46         p.next = p.next.next;
47         --size;
48       }
49     }
50   }
```

ไม่ต้องทำอะไรเมื่อไม่มีข้อมูล

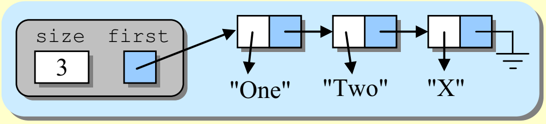
เปลี่ยนค่าของ first กรณีลบปมแรก

เลื่อน p ไปทีละปม เมื่อยังมีปมถัดไป และข้อมูลในปมถัดไปไม่ใช่ e

ถ้าค้นพบ ก็ลบปม p.next ออกจากการโยง

รหัสที่ 4-5 remove ของ LinkedCollection

```
public class LinkedCollection implements Collection {
  private static class LinkedNode {
    private Object element;
    private LinkedNode next;
    LinkedNode(Object e, LinkedNode n) {
      this.element = e;
      this.next = n;
    }
  }
  private int size;
  private LinkedNode first;

  public LinkedCollection() {
    size = 0; first = null;
  }
  public int size() {
    return size;
  }
  public boolean isEmpty() {
    return size == 0;
  }
  public void add(Object e) {
    if (e == null) throw new IllegalArgumentException();
    first = new LinkedNode(e, first);
    ++size;
  }
  public boolean contains(Object e) {
    LinkedNode node = first;

    while (node != null && !node.element.equals(e) ) {
      node = node.next;
    }
    return node != null;
  }
  public void remove(Object e) {
    if (first == null) return;
    if (first.element.equals(e)) {
      first = first.next;
      --size;
    } else {
      LinkedNode p = first;
      while (p.next != null && !p.next.element.equals(e)) {
        p = p.next;
      }
      if (p.next != null) {
        p.next = p.next.next;
        --size;
      }
    }
  }
}
```

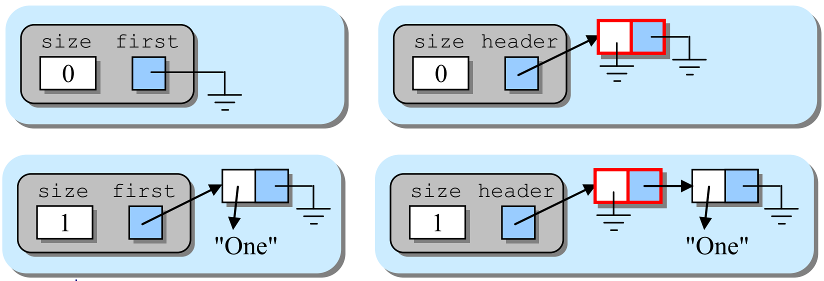


รหัสที่ 4-6 คลาส LinkedCollection

LinkedCollection แบบมีปมหัว

เมท็อด remove ที่ได้อธิบายมามีรายละเอียดการทำงานที่ค่อนข้างจุกจิก กรณีคอลเล็กชันว่างทำแบบหนึ่ง กรณีที่ลบข้อมูลที่ปรากฏในปมแรกทำแบบหนึ่ง และกรณีที่ลบข้อมูลตัวอื่นทำอีกแบบหนึ่ง ที่ต้องตรวจกรณีแรกก็เพราะจะเกิด NullPointerException ถ้าปล่อยให้ทำให้ต่อ ส่วนที่ต้องตรวจกรณีที่สองเพราะปมแรกไม่มีปมก่อนหน้า

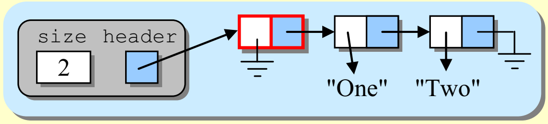
สองกรณีนี้จะไม่มีทางเกิดขึ้นแน่ ถ้าเรามั่นใจว่า มีปมข้อมูลอย่างน้อยหนึ่งปม และมั่นใจว่า จะไม่ลบปมแรกออกจากการโยง คำถามก็คือ เราจะสร้างความมั่นใจแบบนั้นได้อย่างไร เราทำได้เพียงแค่เติมปมพิเศษเป็นปมแรกไว้หนึ่งปม โดยปมนี้ไม่เก็บข้อมูลอะไรทั้งสิ้น เรียกปมพิเศษนี้ว่า *ปมหัว* (header node) ภายในตัวสร้างก็สร้างปมหัวให้เลย ปมหัวนี้จะต้องไม่ถูกลบ และไม่ถูกเปลี่ยนแปลงตลอดอายุการใช้งาน รูปที่ 4-6 เปรียบเทียบการจัดเก็บข้อมูลแบบโยงไม่มีปมหัว (รูปทางซ้าย) และมีปมหัว (รูปทางขวา) ให้สังเกตในรูปว่า เราเปลี่ยนชื่อตัวแปรจาก first เป็น header กรณีจัดเก็บแบบมีปมหัว header.next จึงชี้ปมที่เก็บข้อมูลตัวแรก



รูปที่ 4-6 การเก็บข้อมูลแบบโยงไม่มีปมหัว (ทางซ้าย) และมีปมหัว (ทางขวา)

รหัสที่ 4-7 แสดงรายละเอียดของคลาส LinkedCollection แบบมีปมหัว ถ้าเปรียบเทียบกับแบบเดิมในรหัสที่ 4-6 ที่บรรทัดที่ 11 เราเปลี่ยนชื่อ first เป็น header ภายในตัวสร้างมีการสร้างปมหัวให้ header อ้างอิง (บรรทัดที่ 14) ที่ add จากที่เคยสร้างปมข้อมูลใหม่แล้วแทรกไว้ด้านหน้าของการโยง ก็ให้สร้างปมข้อมูลใหม่แล้วเพิ่มไว้หลังปมหัว (บรรทัดที่ 24) ที่ contains จากเดิมเริ่มค้นที่ first ก็เปลี่ยนไปเริ่มค้นที่ header.next (บรรทัดที่ 28) และที่น่าสนใจที่สุดก็คือ remove ซึ่งสามารถตัดการตรวจสอบสองกรณีพิเศษออก กลายเป็นรหัสที่ซับซ้อนน้อยลง

```java
public class LinkedCollection implements Collection {
  private static class LinkedNode {
    private Object element;
    private LinkedNode next;
    LinkedNode(Object e, LinkedNode n) {
      this.element = e;
      this.next = n;
    }
  }
  private int size;
  private LinkedNode header;

  public LinkedCollection() {
    size = 0; header = new LinkedNode(null, null);
  }
  public int size() {
    return size;
  }
  public boolean isEmpty() {
    return size == 0;
  }
  public void add(Object e) {
    if (e == null) throw new IllegalArgumentException();
    header.next = new LinkedNode(e, header.next);
    ++size;
  }
  public boolean contains(Object e) {
    LinkedNode node = header.next;

    while (node != null && !node.element.equals(e) ) {
      node = node.next;
    }
    return node != null;
  }
  public void remove(Object e) {
    LinkedNode p = header;
    while (p.next != null && !p.next.element.equals(e)) {
      p = p.next;
    }
    if (p.next != null) {
      p.next = p.next.next;
      --size;
    }
  }
}
```



สร้างปมหัวให้เลยที่นี่

สร้างปมใหม่ แล้วเพิ่มไว้หลัง header

header.next ชี้ปมที่เก็บข้อมูลตัวแรก

มีปมหัว การทำงานง่ายขึ้น p ไม่เป็น null และ ไม่เคยต้องการลบ header

รหัสที่ 4-7 คลาส LinkedCollection แบบมีปมหัว

ประสิทธิภาพการทำงาน

LinkedCollection ไม่ว่าจะเป็นแบบมีปมหัว หรือไม่มีก็ตาม มีประสิทธิภาพการทำงานเชิงเวลาเหมือนกัน ตัวสร้าง เมท็อด size และ isEmpty ใช้เวลาคงตัว $\Theta(1)$ add เพิ่มข้อมูลด้านหน้าการโยงซึ่งก็เป็น $\Theta(1)$ เช่นกัน การค้นข้อมูลอาศัยการไล่เปรียบเทียบไปเรื่อย ๆ ทีละตัว ซึ่งใช้เวลามากสุดเมื่อต้องวิ่งไล่ทุก ๆ ปมข้อมูล ถ้าให้ n คือจำนวนข้อมูลในคอลเล็กชัน ดังนั้น contains ใช้เวลา $O(n)$ และการลบอาศัยการวิ่งค้นเช่นเดียวกับ contains จึงใช้เวลา $O(n)$ เมื่อค้นพบก็ลบปมข้อมูลออกจากการโยงใช้เวลาอีก $\Theta(1)$ ดังนั้น remove ใช้เวลา $O(n)$ ตารางที่ 4-1 สรุปประสิทธิภาพเชิงเวลาของการสร้างคอลเล็กชันทั้งสองแบบที่ได้ศึกษามาซึ่งเหมือนกันทุกเมท็อด ยกเว้นการเพิ่มข้อมูลซึ่งช้ากว่า ถ้ามีการขยายแถวลำดับสำหรับ ArrayCollection (แต่ก็ได้นำเสนอในบทที่แล้วว่า ถ้ามีการเพิ่มมาก ๆ หลาย ๆ ครั้ง แล้ววิเคราะห์เวลาถัวเฉลี่ยจะใช้เวลาคงตัว $O(1)$)

ตารางที่ 4-1 ประสิทธิภาพเชิงเวลาของ ArrayCollection และ LinkedCollection

	ArrayCollection	LinkedCollection
constructor	$\Theta(1)$	$\Theta(1)$
size	$\Theta(1)$	$\Theta(1)$
isEmpty	$\Theta(1)$	$\Theta(1)$
contains	$O(n)$	$O(n)$
add(e)	$O(n)$ *	$\Theta(1)$
remove(e)	$O(n)$	$O(n)$

* $O(1)$ ถ้าเป็นกรณีถัวเฉลี่ย

แบบฝึกหัด

1. จงเขียนคลาส LinkedCollection ด้วยตนเอง โดยไม่ดูรายละเอียดในหนังสือ

2. LinkedCollection ที่ได้ศึกษากันมานั้นเก็บอ็อบเจกต์ได้ทุกประเภท ถ้าจะนำไปเก็บ int ก็ต้องใช้คลาส Integer ช่วย ซึ่งสิ้นเปลือง จงเขียนคลาส IntLinkedCollection ซึ่งเป็นคอลเล็กชันที่ไว้เก็บจำนวนเต็มโดยเฉพาะ

3. จงเขียนคลาส LinkedSet ให้เป็นคลาสลูกของ LinkedCollection (รหัสที่ 4-7) โดยไม่อนุญาตให้เก็บข้อมูลซ้ำ ถ้ามีการสั่งเพิ่มตัวซ้ำ จะไม่เพิ่มให้ คืนการทำงานกลับไปเลย

4. การลบข้อมูลในเมท็อด remove อาศัยการเปลี่ยนตัวโยงของปมก่อนหน้าปมที่ถูกลบให้ชี้ข้ามปมที่ต้องการลบ การค้นหาปมจึงต้องใช้ตัวแปรที่ชี้ปมก่อนหน้าปมที่สนใจตรวจสอบ (ดังแสดงใน

รหัสที่ 4-7) ขอเสนอการลบอีกแบบ คือค้นหาปมที่ต้องการลบ ได้ p ชี้ปมที่ต้องการลบ ก็ให้นำข้อมูลของปมถัดจาก p มาเก็บใส่ที่ปม p ตามด้วยการลบปมที่ถัดจาก p ดังแสดงข้างล่างนี้ อยากทราบว่า วิธีนี้ทำงานถูกต้องหรือไม่ อย่างไร

```
public void remove(Object e) {
  LinkedNode p = header.next;
    while (p != null && !p.element.equals(e)) {
      p = p.next;
    }
    if (p != null) {
      p.element = p.next.element;
      p.next = p.next.next;
      --size;
    }
  }
}
```

นำข้อมูลของปมถัดจาก pมา
เก็บใส่ปม p

ลบปมถัดจาก p ทิ้ง

5. ขอปรับปรุงการจัดเก็บใน `LinkedCollection` ที่ทำให้การค้นหาข้อมูลกระทำได้เร็วขึ้น (การลบก็เร็วขึ้นด้วยเพราะการลบก็ต้องค้น) ของเดิมปมท้ายสุดไม่มีปมถัดไป เราก็เลยให้ `next` มีค่าเป็น `null` จะได้รู้สถานะว่าเป็นปมท้ายสุด ขอเปลี่ยนใหม่ให้ `next` ของปมท้ายสุดชี้ไปที่ปมพิเศษที่เราสร้างไว้ก่อนแล้ว โดยให้ชื่อว่า `NULL` (ไม่ใช่ `null`) เมื่อถึงตอนจะค้นหาข้อมูลก่อนจะเข้าวงวนค้นก็ให้นำข้อมูลที่เราจะค้นไปเก็บใส่ `element` ของปม `NULL` การทำเช่นนี้ทำให้เราไม่ต้องพะวงว่า ถ้าถึงปม `NULL` ให้เลิกค้น เพราะอย่างไรก็ตามการค้นต้องพบข้อมูลที่ต้องการแน่ ๆ เงื่อนไขการตรวจสอบของวงวนก็น้อยลง การทำงานก็เร็วขึ้น อยากทราบว่า วิธีนี้มีปัญหาอะไร ต้องแก้ไขอย่างไร

```
public class LinkedCollection implements Collection {
  private static final NULL = new LinkedNode(null, null);
  public LinkedCollection() {
    size = 0; header = new LinkedNode(null, NULL);
  }
  public boolean contains(Object e) {
    LinkedNode node = header.next;
    NULL.element = e;
    while (!node.element.equals(e) ) {
      node = node.next;
    }
    return node != NULL;
  }
```

เริ่มต้นให้ปมหัวชี้ NULL

ใส่ e ไว้ที่ NULL ก่อนค้น

วงวนนี้ต้องค้นพบ e แน่

พบที่ NULL แสดงว่าไม่มี e ในคอลเล็กชัน

6. เราสามารถปรับปรุงวิธีที่นำเสนอการใช้ปม `NULL` ในข้อที่แล้ว ให้ประหยัดขึ้นโดยไม่ต้องใช้ปม `NULL` หรอก เพียงแค่คิดเสียว่า ปมหัวก็คือปม `NULL` ซึ่งหมายความว่า ปมท้ายสุดจะโยงกลับมาที่ปมหัว จงเขียนคลาสใหม่ด้วยแนวคิดนี้ให้สมบูรณ์

7. การเก็บกลุ่มข้อมูลที่รู้ก่อนล่วงหน้าว่ามีข้อมูลซ้ำกันมาก ๆ ใน `LinkedCollection` ก็คงสิ้นเปลืองมาก เราสามารถปรับปรุงโครงสร้างการจัดเก็บให้ประหยัดเนื้อที่ โดยนิยามให้แต่ละปมมีตัวแปรชื่อ `count` กำกับเพื่อระบุว่า ข้อมูลของปมนี้เก็บในคอลเล็กชันซ้ำกันกี่ตัว จงปรับปรุงคลาส `LinkedCollection` ใหม่ให้จัดเก็บกลุ่มข้อมูลในลักษณะดังกล่าว

8. สมมติว่า เรามีเมท็อดที่รับอ็อบเจกต์ของ `LinkedCollection` มาประมวลผล แต่ไม่มั่นใจว่าสิ่งที่ได้รับนั้นมีโครงสร้างถูกต้องหรือไม่ โครงสร้างการโยงที่สร้างปัญหามากคือกรณีที่โยงข้อมูลไปเรื่อย ๆ แบบวงวน หากเราเดินตามการโยงก็ย่อมเดินวนไม่สิ้นสุด จงเขียนเมท็อดที่รับอ็อบเจกต์แบบ `LinkedCollection` มาตรวจสอบว่ามีการโยงเป็นวงวนหรือไม่

   8.1. อนุญาตให้ใช้เนื้อที่เสริมที่มีขนาดพอ ๆ กับปริมาณข้อมูลที่ได้รับ เพื่อช่วยตรวจสอบ

   8.2. อนุญาตให้ใช้เนื้อที่เสริมเป็นปริมาณคงตัว ในการตรวจสอบ

9. จงเขียนโปรแกรมจับเวลาการสร้าง และค้นหาข้อมูลในคอลเล็กชันที่มีข้อมูลสักสิบล้านตัวซึ่งสร้างด้วย `ArrayCollection` (บทที่ 2) เพื่อเปรียบเทียบเวลากับ `LinkedCollection` (ขอให้อ่านรายละเอียดวิธีการจับเวลาในแบบฝึกหัดข้อ 10 ของบทที่ 2)

10. จงเขียนเมท็อดต่อไปนี้ (ที่ทำงานเร็ว ๆ) เพิ่มให้กับ `LinkedCollection` (รหัสที่ 4-7)

    10.1. `Object toArray()` เพื่อคืนแถวลำดับที่มีขนาดเท่ากับจำนวนข้อมูลในคอลเล็กชันและเก็บข้อมูลชุดเดียวกับของคอลเล็กชัน

    10.2. `String toString()` เพื่อคืนสตริงที่บรรยายข้อมูลต่าง ๆ ที่เก็บในคอลเล็กชัน

    10.3. `boolean equals(Object c)` เพื่อคืนผลการเปรียบเทียบว่า คอลเล็กชันนี้กับคอลเลกชัน `c` มีข้อมูลเหมือนกันหรือไม่

    10.4. `boolean containsDup()` เพื่อตรวจสอบว่า มีข้อมูลซ้ำกันหรือไม่ในคอลเล็กชัน

    10.5. `void clear()` เพื่อล้างคอลเล็กชันให้ไม่มีข้อมูลเหลืออยู่เลย

    10.6. `int frequency(Object e)` เพื่อนับว่า มี `e` ปรากฏกี่ตัวในคอลเล็กชัน

    10.7. `void removeAll(Object e)` เพื่อลบ `e` ทุกตัวในคอลเล็กชันออกให้หมด

    10.8. `void removeDup()` เพื่อลบข้อมูลที่ซ้ำกันให้เหลือแค่ตัวเดียว เช่นเดิมเป็น [A,B,A,B] เมื่อลบแล้วจะได้ [A,B]

10.9. `boolean containsAll(LinkedCollection c)` เพื่อตรวจว่าคอลเล็กชันนี้มีข้อมูลทุกตัวที่ `c` มีหรือไม่ เช่น ถ้า `c1` เก็บ [A, B, C, A, D] และ `c2` เก็บ [A, B, B, A] จะได้ `c1.containsAll(c2)` คืนค่า `true` แต่ `c2.containsAll(c1)` คืนค่า `false`

10.10. `Object mode()` คืนฐานนิยม (mode) ของคอลเล็กชัน ฐานนิยมคือข้อมูลที่ปรากฏในคอลเล็กชันเป็นจำนวนมากสุด เช่น `c1` เก็บ [A, B, A, B, A, C] เมื่อเรียก `c1.mode()` จะคืน A ในกรณีที่มีฐานนิยมมากกว่าหนึ่งตัว คืนตัวไหนก็ได้

10.11. เพิ่มตัวสร้าง `LinkedCollection(LinkedCollection c)` ที่ใส่ข้อมูลเริ่มต้นในคอลเล็กชันให้เหมือนของ `c` หมายเหตุ : ตัวสร้างในลักษณะนี้เรียกว่าตัวสร้างทำสำเนา (copy constructor)

5 รายการ

รายการ (list) คือลักษณะการจัดเก็บกลุ่มข้อมูล โดยที่ข้อมูลแต่ละตัวมีเลขลำดับกำกับ ในมุมมองของผู้ใช้งาน ข้อมูลที่เก็บจึงมีลำดับ เช่น ข้อมูลตัวแรก ตัวถัดไป ตัวที่ 20 ตัวสุดท้ายเป็นต้น ที่เก็บข้อมูลแบบรายการจึงมีเมท็อดให้บริการที่เกี่ยวข้องกับลำดับของข้อมูลด้วย รายการเป็นลักษณะการจัดเก็บข้อมูลที่สร้างง่าย ใช้งานง่าย ได้รับการประยุกต์ในงานสารพัดชนิด บทนี้จะอธิบายการสร้างรายการด้วยแถวลำดับ และสร้างด้วยการโยงข้อมูล

อินเทอร์เฟซ List

ผู้ใช้ที่เก็บข้อมูลในคอลเล็กชันจะไม่สนใจเรื่องลำดับก่อนหลังของข้อมูล เปรียบเสมือนเก็บข้อมูลในถุง แต่ถ้าผู้ใช้เก็บข้อมูลในรายการ แสดงว่าต้องการเก็บข้อมูลอย่างมีระเบียบ โดยลำดับของข้อมูลมีความหมาย เช่น ⟨31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31⟩ เป็นรายการของจำนวนวันในแต่ละเดือน เป็นต้น เพื่อให้มีวิธีใช้งานเหมือนกับเมท็อดที่มีเลขลำดับในคลังคลาสมาตรฐานของระบบจาวา ขอกำหนดให้เลขลำดับของรายการเริ่มที่ 0 เราเขียนข้อกำหนดของรายการในรูปของอินเทอร์เฟซ `List` (รหัสที่ 5-1) มีรายละเอียดของเมท็อดต่าง ๆ แสดงในตารางที่ 5-1 บทนี้จะนำเสนอรายการสามรูปแบบคือ `ArrayList`, `SinglyLinkedList`, และ `LinkedList` (รูปที่ 5-1)

```
public interface List extends Collection {
  public void add(int i, Object e);
  public void remove(int i);
  public Object get(int i);
  public void set(int i, Object e);
}
```

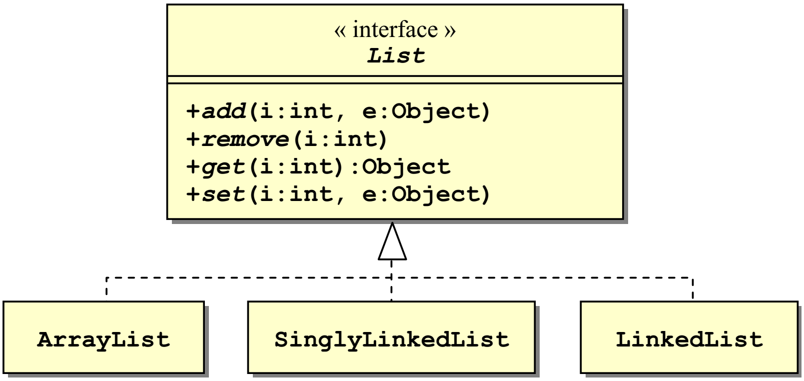
List ก็เป็น Collection

รหัสที่ 5-1 อินเทอร์เฟซ `List`

ตารางที่ 5-1 หน้าที่ของเมท็อดต่าง ๆ ของอินเทอร์เฟซ List

เมท็อด	หน้าที่
void add(int i, Object e)	เพิ่มอ็อบเจกต์ e ไว้ที่เลขลำดับ i ในรายการ
void remove(int i)	ลบข้อมูลซึ่งเก็บที่เลขลำดับ i ในรายการ
Object get(int i)	ขอข้อมูลซึ่งเก็บที่เลขลำดับ i ในรายการ
void set(int i, Object e)	เปลี่ยนข้อมูลซึ่งเก็บที่เลขลำดับ i ในรายการให้เป็น e
void add(Object e)	เพิ่มอ็อบเจกต์ e ต่อท้ายในรายการ (เมท็อดนี้เป็นของอินเทอร์เฟซ Collection)

หมายเหตุ : List นั้น extends Collection จึงมีทุกเมท็อดที่ Collection มีด้วย สำหรับ add(Object e) ที่รับมาจาก Collection นั้น เราขอกำหนดให้เป็นการเพิ่มต่อท้ายรายการ



รูปที่ 5-1 แผนภาพคลาส ArrayList SinglyLinkedList และ LinkedList

ถ้ารายการหนึ่งเก็บข้อมูล n ตัว เราสามารถกำหนดเลขลำดับให้กับเมท็อด remove, get และ set ของรายการนั้นได้ตั้งแต่ค่า 0 ถึง $n-1$ ส่วนเลขลำดับที่ให้กับเมท็อด add มีค่าได้ตั้งแต่ 0 ถึง n โดยการ add ข้อมูลที่ตำแหน่ง n ก็คือการเพิ่มข้อมูลต่อท้ายรายการนั่นเอง ดังนั้นคำสั่ง x.add(e) จึงความหมายเหมือนกับ x.add(x.size(),e)

เพื่อให้เข้าใจวิธีใช้งาน List ให้มากขึ้น รหัสที่ 5-2 แสดงตัวอย่างการสร้างรายการ x ด้วยคลาส ArrayList ที่บรรทัดที่ 1 แล้วเพิ่ม "A" ที่ลำดับ 0, ต่อท้ายรายการด้วย "B", เพิ่ม "C" ที่ลำดับ 0 ซึ่งจะดันข้อมูลเดิมที่ลำดับ 0 เป็นต้นไปตัวละหนึ่งตำแหน่ง ได้รายการเป็น ⟨ "C", "A", "B" ⟩, บรรทัดที่ 8 เปลี่ยนข้อมูลลำดับ 2 (ซึ่งตอนนี้คือ "B") ให้เป็น "D" แล้วลบข้อมูลลำดับที่ 1 ออกในบรรทัดที่ 6 ได้รายการเป็น ⟨ "C", "D" ⟩ วงวน for ใช้เมท็อด get (บรรทัดที่ 8) หยิบข้อมูลออกมาแสดงจากตัวลำดับ 0 จนถึงตัวท้ายของรายการ

```
    List x = new ArrayList();
    x.add(0,"A");                  // <"A">
    x.add("B");                    // <"A", "B">
    x.add(0,"C");                  // <"C", "A", "B">
    x.set(2,"D");                  // <"C", "A", "D">
    x.remove(1);                   // <"C", "D">
    for(int i=0; i<x.size(); i++)
      System.out.println(x.get(i));
```

รหัสที่ 5-2 ตัวอย่างการทำงานของ List

การสร้างรายการด้วยแถวลำดับ

การสร้างรายการทำได้คล้ายกับการสร้างคอลเล็กชัน นั่นคือ สร้างด้วยแถวลำดับ และด้วยการโยง คลาส ArrayList ใช้แถวลำดับสร้างรายการ ภายในประกอบด้วย elementData ซึ่งคือแถวลำดับไว้เก็บข้อมูล และ size เก็บจำนวนข้อมูลเหมือน ArrayCollection ทุกประการ เมท็อด size, isEmpty, contains, และ add ของ ArrayCollection นำมาใช้กับ ArrayList ได้ทั้งสิ้น (จะมีก็ remove ที่นำมาใช้ไม่ได้ ซึ่งจะได้อธิบายในรายละเอียดต่อไป) รหัสที่ 5-3 แสดงคลาส ArrayList ที่มีเมท็อดต่าง ๆ ที่นำมาจาก ArrayCollection (ขอเขียน add(e) ใหม่ให้เรียก add(size,e) เพื่อไม่ให้เกิดความซ้ำซ้อน)

```
public class ArrayList implements List {
  private Object[] elementData = new Object[1];
  private int      size = 0;
  public int size()                    { return size; }
  public boolean isEmpty()             { return size == 0; }
  public boolean contains(Object e)    { return indexOf(e) != -1; }
  public void add(Object e)            { add(size, e); }
  private int indexOf(Object e) {
    for (int i=0; i<size; i++)
      if (elementData[i].equals(e)) return i;
    return -1;
  }
  private void ensureCapacity(int capacity) {
    if (capacity > elementData.length) {
      int s = Math.max(capacity, 2*elementData.length);
      Object[] arr = new Object[s];
      for(int i = 0; i < size; i++) arr[i] = elementData[i];
      elementData = arr;
    }
  }
  ...
```

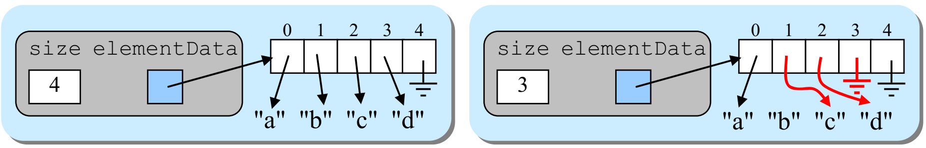
สมาชิกประกอบด้วยตัวอาเรย์กับจำนวนข้อมูล

add แบบนี้คือการเพิ่มข้อมูลใหม่ต่อท้ายรายการ

คืนเลขลำดับของ e ในรายการ หาไม่พบคืน –1

รหัสที่ 5-3 เมท็อดของ ArrayList ที่เหมือนกับของ ArrayCollection

เรานำเมท็อด `remove` ของ `ArrayCollection` มาใช้ไม่ได้กับ `ArrayList` เนื่องจากการลบที่เคยทำมา ใช้วิธีการนำตัวท้ายมาแทนตัวที่ถูกลบ ซึ่งในกรณีที่เป็นรายการจะทำให้ลำดับของข้อมูลผิดไป ดังนั้นเมื่อต้องการลบตัวใด ก็ต้องเลื่อนตัวถัดจากตัวนั้นเป็นต้นไป มาด้านหน้าตัวละหนึ่งตำแหน่ง รหัสที่ 5-4 แสดงการทำงานของ `remove` ทั้งสองแบบคือแบบที่รับอ็อบเจกต์มาค้นแล้วลบกับแบบที่ระบุเลขลำดับที่จะลบ แบบแรก (บรรทัดที่ 21) อาศัย `indexOf` ในการค้นข้อมูลได้เลขลำดับกลับคืนมาเพื่อส่งต่อไปลบด้วย `remove` แบบที่สอง (บรรทัดที่ 25) โดยแบบหลังนี้เริ่มด้วยการตรวจสอบก่อนว่า เลขลำดับ `i` ที่ได้รับต้องอยู่ในช่วงของตำแหน่งที่ลบได้ คือตั้งแต่ 0 ถึง `size-1` ด้วยเมท็อด `assertInRange` หากอยู่นอกช่วง จะเกิด exception ถ้าลบได้ ก็เริ่มย้ายข้อมูลจากช่องที่ `i+1` จนถึงช่องที่ `size-1` มาทางซ้ายตัวละหนึ่งตำแหน่ง (ด้วยวงวน `for` ในบรรทัดที่ 27 ถึง 29) ปิดท้ายด้วยการลดจำนวนข้อมูล และตัดการอ้างอิงที่ไม่จำเป็นออก (บรรทัดที่ 30) รูปที่ 5-2 แสดงตัวอย่างการเรียก `remove(1)` ของรายการที่มีข้อมูล 4 ตัว จึงต้องย้ายข้อมูลในช่องที่ 2 และ 3 มาไว้ช่องที่ 1 และ 2 ตามลำดับ และให้ช่องที่ 3 เป็น `null`



รูปที่ 5-2 ตัวอย่างการลบ `remove(1)` ของรายการในรูปซ้าย ได้ผลในรูปขวา

```
21    public void remove(Object e) {
22      int i = indexOf(e);
23      if (i >= 0) remove(i);
24    }
25    public void remove(int i) {
26      assertInRange(i, size-1);
27      for(int j=i+1; j<size; j++) {
28        elementData[j-1] = elementData[j];
29      }
30      elementData[--size] = null;
31    }
32    private static void assertInRange(int i, int max) {
33      if (i < 0 || i > max)
34        throw new IllegalArgumentException();
35    }
..    ...
```

หาเลขลำดับของ e
ถ้าหาพบก็ลบตัวที่ตำแหน่งนั้น

ตรวจสอบให้มั่นใจว่า $0 \le i \le size - 1$

เลื่อนตั้งแต่ตัวที่ i + 1 จนถึงตัวสุดท้าย
มาทางซ้ายตัวละหนึ่งตำแหน่ง

รหัสที่ 5-4 เมท็อด `remove` ของคลาส `ArrayList`

รหัสที่ 5-5 แสดงเมท็อด `get`, `set` และ `add` ที่มีเลขลำดับเป็นพารามิเตอร์ `get(i)` เพียงตรวจสอบให้มั่นใจว่า `i` อยู่ในช่วงของข้อมูล ถ้าใช่ก็คืน `elementData[i]` ส่วน `set(i,e)` ก็

ทำงานคล้ายกัน คือตรวจสอบว่า i อยู่ในช่วงของข้อมูล และ e มีค่าไม่เป็น null (ด้วยเมท็อด assertNonNull) ถ้าถูกต้องก็เปลี่ยน elementData[i] ให้เป็น e สำหรับ add(i,e) ในบรรทัดที่ 45 จะมีขั้นตอนมากหน่อย เริ่มด้วยการตรวจสอบทั้ง i และ e ให้ถูกต้องตามกฎก่อน ถ้าถูกต้อง ก็เรียก ensureCapacity เพื่อตรวจสอบขนาดแถวลำดับว่า ถ้าไม่พอก็ต้องขยาย จากนั้นย้ายข้อมูลตั้งแต่ช่องที่ size-1 ถอยมาจนถึง i ไปทางขวาตัวละหนึ่งตำแหน่ง (ด้วยวงวน for ในบรรทัดที่ 49 ถึง 51) ใส่ข้อมูลใหม่ในช่องที่ i ปิดท้ายด้วยการเพิ่มจำนวนข้อมูล (บรรทัดที่ 53)

```
36    public Object get(int i) {
37      assertInRange(i, size - 1);
38      return elementData[i];
39    }
40    public void set(int i, Object e) {
41      assertNonNull(e);
42      assertInRange(i, size - 1);
43      elementData[i] = e;
44    }
45    public void add(int i, Object e) {
46      assertNonNull(e);
47      assertInRange(i, size);
48      ensureCapacity(size+1);
49      for(int j=size-1; j>=i; j--) {
50        elementData[j+1] = elementData[j];
51      }
52      elementData[i] = e;
53      size++;
54    }
55    private void assertNonNull(Object e) {
56      if(e == null) throw new IllegalArgumentException();
57    }
58  }
```

คืนข้อมูลลำดับที่ i ของรายการ

เปลี่ยนข้อมูลลำดับที่ i ของรายการ

ย้ายข้อมูลตั้งแต่ตัวที่ i เป็นต้นไป ไปทางขวาตัวละตำแหน่ง

นำข้อมูลใหม่ใส่ช่องที่ i

ตรวจสอบให้มั่นใจว่า e ไม่เป็น null

รหัสที่ 5-5 เมท็อด get, set และ add ของคลาส ArrayList

boolean equals(Object x)

การเปรียบเทียบว่า ArrayList สองตัวเท่ากันหรือไม่ ทำได้ง่ายกว่าการเปรียบเทียบคอลเล็กชัน เนื่องจากสองรายการจะเท่ากันก็ต่อเมื่อมีจำนวนข้อมูลเท่ากัน และข้อมูลในแต่ละตำแหน่งเท่ากัน ดังนั้นก็เพียงแต่ค่อย ๆ เปรียบเทียบไปทีละตัวของทั้งสองรายการ ถ้ามีตำแหน่งใดที่ข้อมูลไม่เท่ากัน ก็สรุปได้ทันทีว่า รายการไม่เท่ากัน รหัสที่ 5-6 แสดงรายละเอียดการทำงาน เราเริ่มด้วยการตรวจสอบประเภทของข้อมูลที่รับมา (ตามมาตรฐานการเขียน equals) ว่าเป็นประเภท ArrayList หรือไม่ ถ้าใช่ จึงทำต่อด้วยการเปรียบเทียบขนาดของทั้งสองรายการว่าต้องเท่ากัน ถ้าเท่ากัน จึงไล่เปรียบเทียบ

ทีละตัว ถ้ามีตัวใดตัวหนึ่งไม่เท่า ก็สรุปว่า สองรายการไม่เท่ากัน ถ้าเปรียบเทียบแล้วเท่ากันทุกตัว ก็คืนค่าจริง

```
public class ArrayList implements List {
  ...
  public boolean equals(Object x) {
    if (!(x instanceof ArrayList)) return false;
    ArrayList that = (ArrayList) x;
    if (size != that.size) return false;        ไม่เท่าถ้าจำนวนไม่เท่า
    for (int i=0; i<size; i++) {
      if (!elementData[i].equals(that.elementData[i]))
        return false;                           ไม่เท่าถ้าตัวที่ i ไม่เท่า
    }
    return true;
  }
  ...
}
```

รหัสที่ 5-6 `equals` ของ `ArrayList`

ประสิทธิภาพการทำงาน

เมท็อด `size`, `isEmpty`, `get`, `set` และตัวสร้างใช้เวลาการทำงานเป็น $\Theta(1)$ เนื่องจากในเมท็อดเหล่านี้ใช้งานคำสั่งพื้นฐานเป็นจำนวนคงตัว ไม่มีวงวนใด ๆ ส่วนการเพิ่มทั้งสองแบบคือ `add(e)` และ `add(i,e)` ใช้เวลาเป็น $O(n)$ เนื่องจากอาจมีการขยายแถวลำดับ และถึงแม้ไม่มีการขยายขนาดก็ต้องมีการย้ายข้อมูล จำนวนการย้ายข้อมูลจะมากหรือน้อยก็ขึ้นกับตำแหน่งของข้อมูลใหม่ การเพิ่มที่ช่อง 0 จะใช้เวลาการทำงานนานสุดเพราะต้องย้ายข้อมูลเดิมทุกตัวไปทางขวา ในขณะที่การเพิ่มข้อมูลใหม่ต่อท้ายรายการจะไม่มีการย้ายข้อมูลใด ๆ เลย สำหรับการลบ ในกรณีของ `remove(e)` ต้องเสียเวลาค้นข้อมูลบวกกับการย้ายข้อมูลจากตำแหน่งที่พบมาทางซ้ายตัวละตำแหน่ง ถ้าค้นพบเร็วแสดงว่า พบที่ตำแหน่งแรก ๆ ก็ต้องย้ายข้อมูลจำนวนมาก แต่ถ้าพบช้าแสดงว่า พบที่ตำแหน่งท้าย ๆ ก็ย้ายข้อมูลจำนวนน้อย โดยสรุปคือค้นพบที่ช่องที่ k ก็ต้องย้ายข้อมูลเป็นจำนวน $n - k$ ช่อง ภาระรวมก็แปรตาม n ซึ่งคือจำนวนข้อมูล จึงใช้เวลาเป็น $\Theta(n)$ ส่วนการลบแบบระบุเลขลำดับนั้น จะเสียเวลาแค่การย้ายข้อมูลอย่างเดียวจึงใช้เวลาซึ่งขึ้นกับตำแหน่งที่ลบ ถ้าลบตัวท้ายรายการก็เร็วสุด และถ้าลบตัวต้นรายการก็ต้องย้ายข้อมูลจำนวนมากสุด ดังนั้นการลบแบบนี้ใช้เวลาเป็น $O(n)$ ส่วนเมท็อด `equals` นั้นเห็นได้ชัดเจนว่า มีการวนตรวจสอบอย่างมาก n ครั้ง จึงใช้เวลาเป็น $O(n)$

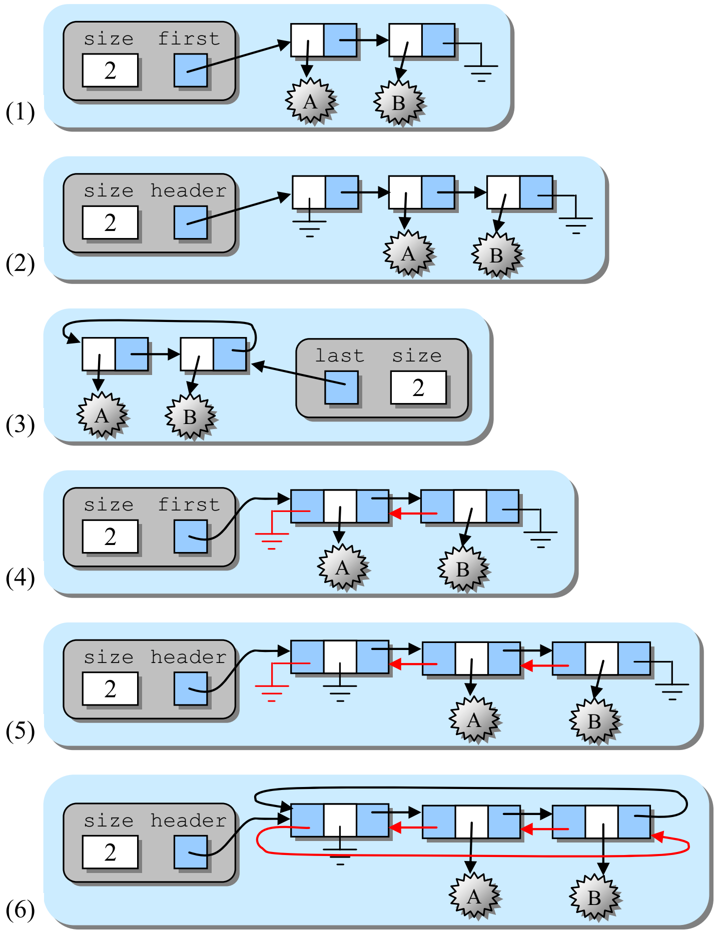
การสร้างรายการโยง

รายการโยง (linked list) คือการจัดเก็บรายการด้วยการโยงข้อมูลต่าง ๆ ตามลำดับในรายการ การจัดเก็บรายการโยงมีลักษณะเหมือนกับคลาส `LinkedCollection` ทุกประการ ส่วนวิธีจัดการนั้นจะต้องคำนึงถึงลำดับของข้อมูลในรายการด้วย ที่ผ่านมาเราได้นำเสนอการโยงข้อมูลแบบมีกับแบบไม่มีปมหัว ยังมีแบบอื่น ๆ อีกหลายประเภทที่ปรับปรุงให้โยงข้อมูลในลักษณะที่ทำให้จัดการข้อมูลได้อย่างมีประสิทธิภาพมากขึ้น รูปที่ 5-3 แสดงตัวอย่างรายการโยงแบบต่าง ๆ พอจำแนกรูปแบบได้ 3 ลักษณะ ดังนี้

- *รายการโยงเดี่ยวหรือโยงคู่* (singly or doubly linked list) โยงเดี่ยวคือแบบที่มีตัวชี้ปมถัดไปอย่างเดียว เช่น รูปที่ 5-3 (1) (2) และ (3) ส่วนโยงคู่นั้นเป็นแบบที่มีทั้งตัวชี้ปมถัดไป และตัวชี้ปมก่อนหน้าด้วย เช่น รูปที่ 5-3 (4) (5) และ (6)
- *รายการโยงแบบมีปมหัว* (header) *หรือไม่มี* เช่น รูปที่ 5-3 (1) (3) และ (4) ไม่มี แต่รูปที่ 5-3 (2) (5) และ (6) มีปมหัว
- *รายการโยงแบบวน* (circular) *หรือไม่วน* กรณีรายการโยงเดี่ยวแบบวน ปมสุดท้ายจะชี้วนกลับมาหาปมแรกในรายการ เช่น รูปที่ 5-3 (3) และถ้าเป็นกรณีรายการโยงคู่แบบวน นอกจากปมสุดท้ายจะชี้ถัดไปยังปมแรกแล้ว ปมแรกก็จะชี้กลับมายังปมสุดท้ายด้วย เช่น รูปที่ 5-3 (6)

แน่นอนว่า รายการโยงคู่ใช้เนื้อที่ในหนึ่งปมข้อมูลมากกว่าแบบโยงเดี่ยว แต่ก็สามารถให้บริการหาปมก่อนหน้าได้ในเวลาคงตัว ซึ่งถ้าเราถามปมก่อนหน้าในรายการโยงเดี่ยวย่อมต้องเสียเวลาเป็น O(n) เพราะต้องเริ่มไล่ตั้งแต่ปมแรก รายการแบบมีปมหัวจะช่วยขจัดการที่ต้องตรวจสอบกรณีพิเศษ ๆ ออกไป ทำให้เขียนโปรแกรมได้สะอาด ส่วนกรณีของรายการโยงแบบวนจะช่วยให้เราเข้าถึงปมข้อมูลปมแรก และปมท้ายได้อย่างรวดเร็ว เหมาะสำหรับรายการที่ต้องการให้บริการที่ปลายสองด้านของรายการอยู่เป็นประจำ

หัวข้อย่อยต่อไปนี้จะนำเสนอการสร้างรายการสองวิธี คือรายการโยงเดี่ยวแบบไม่วนที่มีปมหัว (รูปที่ 5-3 (2)) และรายการโยงคู่แบบวนที่มีปมหัว (รูปที่ 5-3 (6))

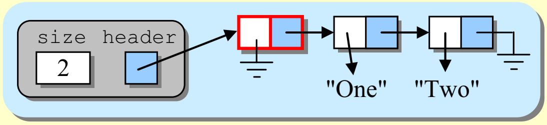


รูปที่ 5-3 รายการโยงแบบต่าง ๆ

รายการโยงเดี่ยวแบบไม่วนที่มีปมหัว

จากคลาส `LinkedCollection` ที่ได้ศึกษากันมา เราสามารถนำมาปรับปรุงเพิ่มให้เป็นรายการได้ไม่ยาก ให้ชื่อคลาสใหม่นี้ว่า `SinglyLinkedList` เป็นรายการโยงเดี่ยวแบบไม่วนที่มีปมหัว (รูปที่ 5-3(2)) เมท็อด `size`, `isEmpty`, `contains`, `remove` และตัวสร้างล้วนมีขั้นตอนการทำงานเหมือนกับที่ปรากฏใน `LinkedCollection` ทุกประการ ดังแสดงในรหัสที่ 5-7 (ขอสร้างเมท็อดส่วนตัว เพื่อใช้เฉพาะภายในคลาสนี้ชื่อว่า `removeAfter` ซึ่งจะลบปมข้อมูลที่อยู่หลัง `node` ที่เป็นพารามิเตอร์ เมท็อดนี้ใช้ใน `remove(e)` และจะใช้ใน `remove(i)` ด้วย)

```
public class SinglyLinkedList implements List {
  private static class LinkedNode {
    private Object element;
    private LinkedNode next;
    LinkedNode(Object e, LinkedNode n) {
      this.element = e;
      this.next = n;
    }
  }
  private int size;
  private LinkedNode header;

  public SinglyLinkedList() {
    size = 0; header = new LinkedNode(null, null);
  }
  public int size() {
    return size;
  }
  public boolean isEmpty() {
    return size == 0;
  }
  public boolean contains(Object e) {
    LinkedNode node = header.next;
    while (node != null && !node.element.equals(e) ) {
      node = node.next;
    }
    return node != null;
  }
  public void add(Object e) {
    add(size, e);
  }
  public void remove(Object e) {
    LinkedNode p = header;
    while(p.next != null && !p.next.element.equals(e)){
      p = p.next;
    }
    removeAfter(p);
  }
  private void removeAfter(LinkedNode p) {
    if (p.next != null) {
      p.next = p.next.next;
      --size;
    }
  }
  ...
```



เมท็อดนี้คือการเพิ่มต่อท้ายรายการ

ลบปมข้อมูลที่อยู่ถัดจาก node ออกจากการโยง

เขียนแยกเป็นเมท็อด เพราะเดี๋ยวจะมีเมท็อดอื่นเรียกใช้บริการนี้

รหัสที่ 5-7 คลาส SinglyLinkedList ซึ่งนำเมท็อดมาจาก LinkedCollection

นอกจาก removeAfter ที่เป็นเมท็อดส่วนตัวแล้ว ยังมีอีกสามเมท็อดที่ใช้เฉพาะในคลาสนี้คือ assertNonNull, assertInRange, และ nodeAt สองเมท็อดแรกเคยอธิบายมาแล้วใน

`ArrayList` มีหน้าที่ตรวจสอบความถูกต้องของข้อมูลที่ต้องไม่เป็น `null` และเลขลำดับที่ต้องอยู่ในช่วงที่เป็นไปได้ ส่วนเมท็อด `nodeAt(i)` มีหน้าที่คืนปมที่เก็บข้อมูลลำดับที่ `i` โดยมีกรณีพิเศษคือถ้าให้ `i` เป็น –1 จะคืนปมหัวของรายการโยง การทำงานของ `nodeAt` เริ่มด้วยการตั้งตัวแปร `p` ให้ชี้ที่ปมหัว แล้วเข้าสู่วงวนเลื่อน `p` ไปยังปมถัดไปเรื่อย ๆ จนครบ `i+1` ครั้ง คำสั่ง `for` ที่บรรทัดที่ 54 เริ่ม `j=-1` แล้วไปจบที่ `i-1` จึงทำคำสั่ง `p = p.next` ทั้งสิ้น `i+1` ครั้ง เหตุที่ทำเช่นนี้เพราะเราเริ่ม `p` ที่ปมหัว ถ้า `i` มีค่า –1 ก็ไม่มีการเลื่อน ถ้าเท่ากับ 0 ก็เลื่อน 1 ครั้งไปยังปมถัดจากปมหัว ซึ่งก็คือปมที่เก็บข้อมูลลำดับ 0 ดังนั้นถ้าต้องการปมลำดับที่ `i` ต้องเลื่อน `i+1` ครั้งจากปมหัว

```
45    private static void assertNonNull(Object e) {
46      if(e == null) throw new IllegalArgumentException();
47    }
48    private static void assertInRange(int i, int max) {
49      if (i < 0 || i > max)
50        throw new IllegalArgumentException();
51    }
52    private LinkedNode nodeAt(int i) {
53      LinkedNode p = header;
54      for (int j=-1; j<i; j++) p = p.next;
55      return p;
56    }
..    ...
```

คืนปมข้อมูลที่มีเลขลำดับ i ถ้า i มีค่า -1 ให้คืน header

รหัสที่ 5-8 เมท็อดส่วนตัวใช้ภายใน `SinglyLinkedList`

รหัสที่ 5-9 แสดงเมท็อดที่เหลือซึ่งเกี่ยวข้องกับเลขลำดับ คือ `get`, `set`, `add(i,e)` และ `remove(i)` ทุก ๆ เมท็อดอาศัย `nodeAt` ในการค้นปมข้อมูล `get` และ `set` จะค้นปมข้อมูลของเลขลำดับที่ต้องการเพื่อมาหยิบ (บรรทัดที่ 59) หรือเปลี่ยน (บรรทัดที่ 64) ตัวแปร `element` ซึ่งคือตัวข้อมูลของปมที่ค้นได้ ส่วน `add` และ `remove` ใช้ `nodeAt` เพื่อค้นปมก่อนหน้าปมที่มีเลขลำดับที่ต้องการ (จึงใช้ `nodeAt(i-1)` ที่บรรทัดที่ 69 และ 75) เพื่อเพิ่ม (บรรทัดที่ 70) และลบ (บรรทัดที่ 76) ปมข้อมูลหลังปมที่หาได้

ประสิทธิภาพเชิงเวลาของเมท็อดใน `SinglyLinkedList` เป็นดังนี้ ตัวสร้าง, `isEmpty` และ `size` ใช้เวลา $\Theta(1)$ `remove(e)` ใช้เวลา $O(n)$ เพราะเสียเวลาค้น $O(n)$ บวกกับการลบปมข้อมูลจากการโยงอีก $\Theta(1)$ เนื่องจากเมท็อด `nodeAt` ต้องทำงานเป็นวงวนโดยเริ่มจากปมหัว วิ่งตามการโยงไปจนถึงเลขลำดับที่ได้รับจึงใช้เวลาเป็น $O(n)$ ดังนั้นเมท็อดที่เกี่ยวข้องกับเลขลำดับอันได้แก่ `get`, `set`, `remove`, และ `add` ทั้งสองแบบล้วนใช้เวลาการทำงานเป็น $O(n)$ ทั้งหมด

```
  public Object get(int i) {
    assertInRange(i, size-1);
    return nodeAt(i).element;
  }
  public void set(int i, Object e) {
    assertNonNull(e);
    assertInRange(i, size-1);
    nodeAt(i).element = e;
  }
  public void add(int i, Object e) {
    assertNonNull(e);
    assertInRange(i, size);
    LinkedNode p = nodeAt(i-1);
    p.next = new LinkedNode(e, p.next);
    ++size;
  }
  public void remove(int i) {
    assertInRange(i, size-1);
    LinkedNode p = nodeAt(i-1);
    removeAfter(p);
  }

}
```

คืนข้อมูลของปมข้อมูล ที่มีเลขลำดับ i

เปลี่ยนข้อมูลของปมข้อมูลที่มีเลข ลำดับ i ให้เป็น e

เพิ่มปมข้อมูลหลังปมที่มี เลขลำดับ i – 1

ลบปมข้อมูลหลังปมที่มี เลขลำดับ i – 1

รหัสที่ 5-9 เมท็อดต่าง ๆ ที่เกี่ยวข้องกับเลขลำดับใน `SinglyLinkedList`

รายการโยงคู่แบบวนที่มีปมหัว

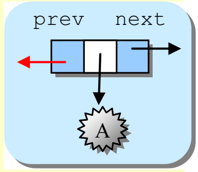
รายการโยงแบบนี้เสริมคุณลักษณะครบถ้วน (รูปที่ 5-3(6)) ทั้งโยงไปโยงกลับ ทำให้กระเถิบไปลำดับถัดไป หรือถอยกลับลำดับได้รวดเร็ว โยงวนจากท้ายสุดมาหน้าสุด และจากหน้าสุดกลับไปท้ายสุด ทำให้พุ่งไปปมสุดท้ายได้อย่างรวดเร็ว และมีปมหัวเพื่อให้เขียนโปรแกรมได้สวย ขจัดการตรวจสอบกรณีจุกจิก ให้ชื่อคลาสของรายการโยงแบบนี้ว่า `LinkedList` [1]

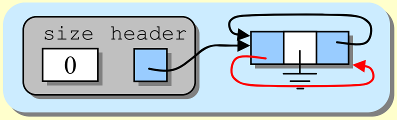
รหัสที่ 5-10 แสดงส่วนต้นของ `LinkedList` ประกอบด้วยคลาสภายใน `LinkedNode` ซึ่งเพิ่มตัวโยงกลับหลัง เพราะเป็นรายการแบบโยงคู่ โดยตั้งชื่อตัวโยงกลับนี้ว่า `prev` (บรรทัดที่ 4) อ็อบเจกต์ของ `LinkedList` หนึ่งตัวประกอบด้วยตัวแปร `size` เก็บจำนวนข้อมูล และ `header` อ้างอิงปมหัวของรายการโยง (เช่นเดียวกับ `LinkedCollection`) มีตัวสร้างหนึ่งตัวสำหรับสร้างรายการโยงใหม่ เนื่องจากเป็นรายการโยงแบบมีปมหัว เราจึงต้องสร้างปมหัวให้ `header` อ้างอิงเลยตั้งแต่เกิด และเนื่องจากเป็นรายการแบบวน ตอนเริ่มต้นมีแค่ปมหัวปมเดียว ดังนั้นปมก่อนหน้าและปมถัดไปก็คือตัวเอง (บรรทัดที่ 18)

[1] ในคลังคลาสมาตรฐานของจาวา ก็มีคลาสชื่อ java.util.LinkedList ซึ่งเป็นรายการโยงในลักษณะเดียวกัน

```
01 public class LinkedList implements List {
02   private static class LinkedNode {
03     private Object element;
04     private LinkedNode prev;
05     private LinkedNode next;
06     LinkedNode(Object e, LinkedNode p, LinkedNode n) {
07       this.element = e;
08       this.prev = p;
09       this.next = n;
10     }
11   }
12   private int size;
13   private LinkedNode header;
14
15   public LinkedList() {
16     size = 0;
17     header = new LinkedNode(null, null, null);
18     header.prev = header.next = header;
19   }
..   ...
```

เพิ่มตัวโยงย้อนกลับ





ปมก่อนหน้า และปมถัดไป ก็คือตัวเอง

รหัสที่ 5-10 คลาสภายใน `LinkedNode` และตัวสร้างของคลาส `LinkedList`

รหัสที่ 5-11 แสดงเมท็อด `size`, `isEmpty`, และ `contains` สองเมท็อดแรกก็เหมือนที่ผ่าน ๆ มาคือใช้ตัวแปร `size` ให้เป็นประโยชน์ ส่วน `contains` นั้นเรียกใช้ `nodeOf(e)` ซึ่งค้นหาปมข้อมูลที่เก็บ `e` ถ้าผลที่ได้คือ `header` แสดงว่าค้นไม่พบ ถ้าเป็นค่าอื่น ก็สรุปว่าค้นพบ (บรรทัดที่ 24) การทำงานของ `nodeOf` เริ่มค้นที่ `header.next` วิ่งตามตัวโยงเปรียบเทียบข้อมูลไปเรื่อย ๆ ที่ต้องสังเกตก็คือว่า การวิ่งตามตัวโยงจะจบลงก็เมื่อวนกลับมาหา `header` (บรรทัดที่ 28) เพราะนี่เป็นรายการโยงแบบวน (ที่เราต้องแยกเขียนเป็นเมท็อดส่วนตัวชื่อ `nodeOf` นี้ก็เพราะจะมีการใช้บริการนี้อีกในเมท็อด `remove`)

```
20   public int size()        { return size; }
21   public boolean isEmpty() { return size == 0; }
22
23   public boolean contains(Object e) {
24     return nodeOf(e) != header;
25   }
26   private LinkedNode nodeOf(Object e) {
27     LinkedNode node = header.next;
28     while(node != header && !node.element.equals(e) ) {
29       node = node.next;
30     }
31     return node;
32   }
..   ...
```

เหมือนของเดิม ๆ

เป็นรายการแบบวน ดังนั้นจะหมดรายการ เมื่อโยงกลับมาหา header

รหัสที่ 5-11 เมท็อด `size` `isEmpty` และ `contains` ของ `LinkedList`

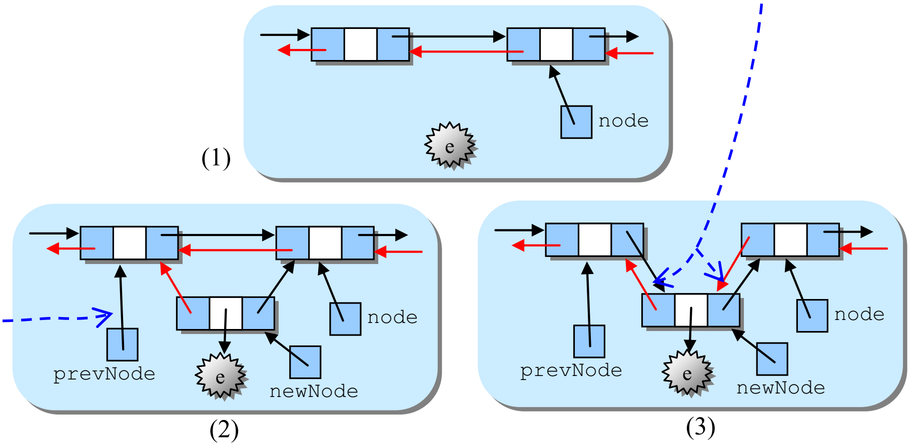
รหัสที่ 5-12 แสดงเมท็อดการเพิ่มข้อมูล มีทั้งการเพิ่มต่อท้าย (บรรทัดที่ 33) และการเพิ่มแบบระบุเลขลำดับ (บรรทัดที่ 36) ทั้งสองเมท็อดนี้อาศัย `addBefore(node,e)` ซึ่งมีหน้าที่สร้างปมข้อมูลใหม่ให้เก็บ `e` และเพิ่มปมใหม่นี้ไว้ด้านหน้าของปม `node` ที่ได้รับ ดังนั้น `add(e)` ซึ่งเป็นการเพิ่มต่อท้าย ก็ใช้ `addBefore(header,e)` เพื่อเพิ่มไว้หน้าปมหัว ซึ่งคือการเพิ่มท้ายรายการนั่นเองเพราะปมท้ายของรายการจะวนกลับมาหาปมหัว ส่วนการเพิ่มแบบระบุเลขลำดับ `add(i,e)` เราต้องหาปมข้อมูลลำดับที่ `i` โดยใช้บริการ `nodeAt(i)` ซึ่งเป็นเมท็อดเดียวกับที่เคยนำเสนอในคลาส `SinglyLinkedList` เมื่อได้ปมลำดับที่ `i` แล้วก็สั่งเพิ่มปมใหม่ไว้หน้าปมที่ `i` ด้วย `addBefore` (บรรทัดที่ 38)

```
33   public void add(Object e) {
34     addBefore(header, e);
35   }
36   public void add(int i, Object e) {
37     assertInRange(i, size);
38     addBefore(nodeAt(i), e);
39   }
40   private void addBefore(LinkedNode node, Object e) {
41     assertNonNull(e);
42     LinkedNode prevNode = node.prev;
43     LinkedNode newNode = new LinkedNode(e, prevNode, node);
44     prevNode.next = node.prev = newNode;
45     ++size;
46   }
..   ...
```

เพิ่มต่อท้าย คือเพิ่มไว้ด้านหน้าของปมหัว

เพิ่มที่ลำดับ i คือเพิ่มไว้ด้านหน้าของปมเก็บข้อมูลลำดับ i

รหัสที่ 5-12 เมท็อดการเพิ่มข้อมูลของ `LinkedList`



รูปที่ 5-4 การทำงานของ `addBefore` ใน `LinkedList`

รูปที่ 5-4 (1) แสดงตัวอย่างการทำงานของ addBefore บรรทัดที่ 42 ให้ prevNode ชี้ปมข้อมูลที่อยู่ก่อนหน้า node บรรทัดที่ 43 สร้างปมข้อมูลใหม่โดยส่งตัวข้อมูล ตัวโยงหลังกลับไปยัง prevNode และตัวโยงหน้าไปยังปม node เป็นการแทรกปมใหม่นี้ไว้ระหว่างปม prevNode และปม node ดังรูปที่ 5-4 (2) ภาระที่เหลือก็คือการโยงเส้น next ของ prevNode และโยงเส้น prev ของ node มายังปมใหม่ในบรรทัดที่ 44 แสดงดังรูปที่ 5-4 (3) ปิดท้ายด้วยการเพิ่มจำนวนข้อมูล (บรรทัดที่ 45)

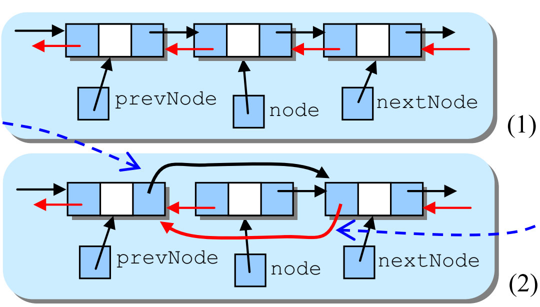
รหัสที่ 5-13 แสดงเมท็อดการลบข้อมูล มีทั้งการลบข้อมูลที่กำหนดให้ (บรรทัดที่ 47) และการลบแบบระบุเลขลำดับ (บรรทัดที่ 51) ทั้งสองเมท็อดนี้อาศัย removeNode(node) ซึ่งมีหน้าที่ลบปมข้อมูล node ออกจากการโยง เมท็อด remove(e) ค้นปมข้อมูลที่เก็บ e ด้วย nodeOf แล้วส่งไปให้ removeNode ลบออก ส่วน remove(i) หยิบปมข้อมูลลำดับที่ i ด้วยเมท็อด nodeAt แล้วก็ส่งให้ removeNode ลบเช่นกัน

```
47    public void remove(Object e) {
48      LinkedNode node = nodeOf(e);
49      if (node != header) removeNode(node);
50    }
51    public void remove(int i) {
52      assertInRange(i, size);
53      removeNode(nodeAt(i));
54    }
55    private void removeNode(LinkedNode node) {
56      LinkedNode prevNode = node.prev;
57      LinkedNode nextNode = node.next;
58      prevNode.next = nextNode;
59      nextNode.prev = prevNode;
60      --size;
61    }
..    ...
```

ใช้ nodeOf ค้นปมที่เก็บ e

ใช้ nodeAt ค้นปมลำดับที่ i

รหัสที่ 5-13 เมท็อดการลบข้อมูลของ LinkedList



รูปที่ 5-5 การทำงานของ removeNode ใน LinkedList removeNode(node) ลบ node ออกจากการโยง เริ่มด้วยการให้ prevNode ชี้ปมข้อมูลก่อนหน้า node (บรรทัดที่ 56) และ ให้ nextNode ชี้ปมข้อมูลถัดจาก node (บรรทัดที่ 57) แสดงดังรูปที่ 5-5 (1) จากนั้นให้ปมข้อมูลทั้ง prevNode และ nextNode โยงไปโยงกลับข้ามปม node โดยโยงให้ next ของ prevNode ชี้ไปยัง nextNode (บรรทัดที่ 58) และโยงให้ prev ของ nextNode ชี้กลับมายัง prevNode (บรรทัดที่ 59) ทำให้ปม node ถูกลบออกจากโยง ดังรูปที่ 5-5 (2) ปิดท้ายด้วยการลดจำนวนข้อมูล (บรรทัดที่ 60) เพื่อความสมบูรณ์ รหัสที่ 5-14 แสดงส่วนที่เหลือของคลาส LinkedList

ประสิทธิภาพเชิงเวลาของเมท็อดต่าง ๆ ของ LinkedList เป็นดังนี้ ตัวสร้าง, isEmpty และ size ใช้เวลา $\Theta(1)$ เมท็อด nodeOf และ nodeAt ล้วนต้องวิ่งตามการโยงจนถึงปมที่ต้องการ ซึ่งใช้เวลา $O(n)$ ดังนั้น contains, get, set, remove(e) remove(i) และ add(i,e) จึงใช้เวลา $O(n)$ เหมือนกับของ SinglyLinkedList ทั้งสิ้น ยกเว้นเฉพาะ add(e) ซึ่งเป็นการเพิ่มต่อท้ายรายการ (ซึ่งคือเพิ่มหน้าปมหัว)ใช้เวลา $\Theta(1)$ เท่านั้น

```
  public Object get(int i) {
    assertInRange(i, size-1);
    return nodeAt(i).element;
  }
  public void set(int i, Object e) {
    assertNonNull(e);
    assertInRange(i, size-1);
    nodeAt(i).element = e;
  }
  private static void assertNonNull(Object e) {
    if(e == null) throw new IllegalArgumentException();
  }
  private static void assertInRange(int i, int max) {
    if (i < 0 || i > max)
      throw new IllegalArgumentException();
  }
  private LinkedNode nodeAt(int i) {
    LinkedNode p = header;
    for (int j=-1; j<i; j++) p = p.next;
    return p;
  }
}
```

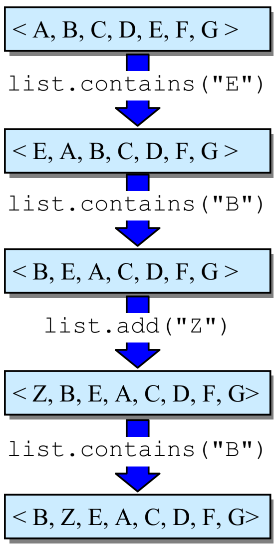
รหัสที่ 5-14 เมท็อด get set และเมท็อดส่วนตัวอื่น ๆ ของ LinkedList

ตัวอย่างการใช้งานรายการ

หัวข้อนี้นำเสนอตัวอย่างการใช้โครงสร้างแบบรายการจัดเก็บข้อมูล ได้แก่ รายการที่ปรับตัวเอง ฟังก์ชันพหุนามตัวแปรเดียว เวกเตอร์มากเลขศูนย์ และเมทริกซ์มากเลขศูนย์

รายการที่ปรับตัวเอง

รายการที่ปรับตัวเอง (self-adjusting list) เป็นรายการที่เก็บข้อมูลในลักษณะซึ่งสามารถเปลี่ยนตำแหน่งของข้อมูลได้เอง ทั้งนี้เพื่อให้การค้นข้อมูลในรายการครั้งต่อ ๆ ไป กระทำได้รวดเร็วขึ้น โดยอาศัยสมมติฐานที่ว่า ความถี่ในการค้นข้อมูลแต่ละตัวในรายการมีไม่เท่ากัน บางตัวถูกใช้บ่อย บางตัวถูกใช้น้อย ข้อมูลตัวที่เพิ่งถูกใช้มีโอกาสสูงที่จะถูกใช้อีกในอนาคตอันใกล้ ดังนั้นจึงควรจัดเก็บข้อมูลให้ตัวที่ถูกใช้บ่อยอยู่ด้านหน้ารายการ เนื่องจากเราค้นข้อมูลตามลำดับเริ่มจากด้านหน้าไปท้าย จะได้ค้นพบข้อมูลที่ใช้บ่อยได้เร็ว ๆ ปัญหาอยู่ตรงที่ว่า เราไม่รู้ความถี่ในการใช้ข้อมูลแต่ละตัวก่อนล่วงหน้า จึงต้องมีวิธีการอะไรบางอย่างที่จะปรับข้อมูลรายการให้อยู่ในสภาพที่กล่าวไว้ข้างต้น วิธีการหนึ่งที่ได้ผลดีคือกลยุทธ "*การย้ายไปด้านหน้า*" (move-to-front) หมายความว่า หากมีใครมาค้นข้อมูล x แล้วพบในรายการ ก็ให้ย้ายข้อมูล x มาไว้ด้านหน้ารายการเลย ครั้งต่อ ๆ ไปถ้าค้น x อีกจะได้พบ x แบบเร็วสุด ๆ และการเพิ่มข้อมูลใหม่ในรายการก็ให้เพิ่มไว้ที่ด้านหน้ารายการด้วย รูปที่ 5-6 แสดงตัวอย่างการเปลี่ยนแปลงรายการที่ปรับตัวเองด้วยการย้ายไปด้านหน้า



รูปที่ 5-6 ตัวอย่างการเปลี่ยนแปลงของรายการที่ปรับตัวเอง

ถ้าเราสร้างรายการด้วยแถวลำดับ การย้ายข้อมูลไปด้านหน้าจะต้องดันข้อมูลทั้งหมดทุกครั้งที่ค้นหรือเพิ่มข้อมูล แต่ถ้าใช้รายการโยง เราสามารถย้ายข้อมูลได้ในเวลาคงตัว รหัสที่ 5-15 แสดงรายละเอียดของ `contains` และ `add` ของคลาส `SelfAdjustingList` ซึ่งเป็นรายการโยงที่ปรับตัวเอง

```
public class SelfAdjustingList implements List {
  ...
  // รายละเอียดอื่น ๆ เหมือนของ LinkedList ทุกประการ
  ...
  public boolean contains(Object e) {
    LinkedNode node = nodeOf(e);
    if (node == header) return false;          หาไม่พบ ก็ไม่ต้องทำอะไร

    node.prev.next = node.next;                ลบออกจากตำแหน่งเดิม
    node.next.prev = node.prev;

    node.prev = header; node.next = header.next;
    node.prev.next = node.next.prev = node;

    return true;                               ย้ายไปไว้หน้ารายการ
  }
  public void add(Object e) {
    addBefore(header.next, e);                 เพิ่มข้อมูลใหม่ไว้หน้ารายการ
  }
}
```

รหัสที่ 5-15 เมท็อด `contains` และ `add` ของ คลาส `SelfAdjustingList`

ฟังก์ชันพหุนามตัวแปรเดียว

ฟังก์ชันพหุนาม (polynomial) แบบตัวแปรเดียว คือฟังก์ชันที่เขียนได้ในรูปแบบ $f(x) = a_0x^0 + a_1x^1 + \dots + a_{n-1}x^{n-1}$ โดยสัมประสิทธิ์ a_k ทั้งหลายเป็นค่าคงตัว จะขอสร้างคลาสชื่อ `Polynomial` ที่ให้บริการต่าง ๆ ดังแสดงในตารางที่ 5-2

ตารางที่ 5-2 หน้าที่ของเมท็อดต่าง ๆ ในคลาส `Polynomial`

เมท็อด	หน้าที่
`Polynomial add(Polynomial that)`	คืนผลบวกของฟังก์พหุนาม `this` กับ `that`
`Polynomial multiply(Polynomial that)`	คืนผลคูณของฟังก์พหุนาม `this` กับ `that`
`void addTerm(double c, int e)`	เพิ่มพจน์ใหม่ cx^e
`Polynomial diff()`	คืนอนุพันธ์ของฟังก์ชันพหุนาม `this`

```
Polynomial p = new Polynomial();
p.addTerm(-3, 2);
p.addTerm(1, 0);
System.out.println(p);
Polynomial q = p.add(p);
System.out.println(q.diff());
System.out.println(q.multiply(p));
```

$p(x) = 1 - 3x^2$

$q(x) = p(x) + p(x)$

$q'(x)$

$q(x) \times p(x)$

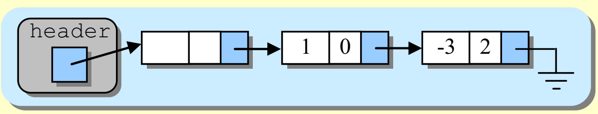
รหัสที่ 5-16 ตัวอย่างการใช้งาน `Polynomial`

รหัสที่ 5-16 แสดงตัวอย่างการใช้งานคลาส `Polynomial` เริ่มด้วยการสร้างฟังก์ชันพหุนามว่าง ๆ ชื่อ `p` เพิ่มพจน์ $-3x^2$ กับ 1 จะได้ $p(x) = 1 - 3x^2$ จากนั้นทำ `q=p.add(p)` ได้ $q(x) = 2 - 6x^2$ ตามด้วยการทำ `q.diff()` จะได้ผลเป็น $-6x$ (ให้สังเกตว่า `q.diff` ไม่ได้ทำให้ `q` เปลี่ยนแปลง) และท้ายสุดทำ `q.multiply(p)` จะได้ผลเป็น $(1 - 3x^2)(2 - 6x^2) = (2 - 12x^2 + 18x^4)$

เราสามารถจัดเก็บฟังก์ชันพหุนามนี้ด้วยรายการคู่ลำดับของเลขชี้กำลังและสัมประสิทธิ์ เช่น $p(x) = 2 + 3x - 4.5x^3$ ก็จัดเก็บเป็น $\langle$ (2,0), (3,1), (–4.5, 3) $\rangle$ เป็นต้น โดยจัดเก็บคู่ลำดับเหล่านี้ให้เรียงลำดับตามเลขชี้กำลังจากน้อยไปมากเพื่อความสะดวกในการจัดการ จะขอสร้างฟังก์ชันพหุนามนี้ด้วยรายการโยงเดี่ยวที่มีปมหัว ที่เลือกรายการโยงก็เพราะการแทรกปมข้อมูลใหม่ในรายการใช้เวลาคงตัวสำหรับรายการโยง ที่เลือกโยงเดี่ยวก็เพราะไม่จำเป็นต้องโยงกลับ (การจัดการในเมท็อดต่าง ๆ ไม่มีการวิ่งย้อนกลับ) และที่เลือกให้มีปมหัวก็เพราะจะทำให้โปรแกรมสั้นและสะอาด

คลาส `LinkedNode` แทนลักษณะของแต่ละปมข้อมูล มีสมาชิก 3 ตัวคือ สัมประสิทธิ์ (`coef`) เลขชี้กำลัง (`exp`) และตัวอ้างอิงปมข้อมูลถัดไป (`next`) ดังแสดงในรหัสที่ 5-17 ส่วนคลาส `Polynomial` มีเฉพาะ `header` ไว้อ้างอิงปมหัวก็พอ (ไม่จำเป็นต้องมี `size` เหมือนที่เคยมีมา เพราะเราไม่มีบริการให้ถามจำนวนพจน์ของฟังก์ชัน) โดยเมื่อมีการสร้างอ็อบเจกต์ใหม่จะสร้างปมหัวให้ `header` เก็บ (บรรทัดที่ 12)

```
01 public class Polynomial {
02   private static class LinkedNode {
03     private double coef;
04     private int exp;
05     private LinkedNode next;
06     LinkedNode(double c, int e, LinkedNode n) {
07       this.coef = c;
08       this.exp = e;
09       this.next = n;
10     }
11   }
12   private LinkedNode header = new LinkedNode(0,0,null);
..   ...
```



รหัสที่ 5-17 คลาส `LinkedNode` ภายใน `Polynomial`

ต่อด้วยเมท็อด `addTerm` (รหัสที่ 5-18) ซึ่งรับสัมประสิทธิ์และเลขชี้กำลังมาเพื่อสร้างพจน์ใหม่เพิ่มให้กับฟังก์ชัน ขอย้ำตรงนี้อีกครั้งว่า เราจัดเก็บรายการให้เก็บปมข้อมูลเรียงลำดับจากน้อยไปมากตามเลขชี้กำลัง ดังนั้นจึงต้องเริ่มหาตำแหน่งที่จะแทรกปมใหม่ ตัวแปร `node` ใน `addTerm` มีหน้าที่อ้างอิงปมข้อมูลก่อนปมที่สนใจ ให้เริ่มต้นที่ `header` แล้วเข้าวงวน (บรรทัดที่ 15 ถึง 17) เลื่อน `node` ไปทีละปมเรื่อย ๆ ตราบเท่าที่ยังมีปมถัดไปให้สนใจ และปมถัดไปนั้นมีเลขชี้กำลังน้อยกว่า `e` ซึ่งคือเลขชี้กำลังตัวใหม่ที่อยากเพิ่ม ดังนั้นเมื่อหลุดออกจากวงวน ก็สามารถเพิ่มปมใหม่ไว้หลังปมที่ `node` อ้างอิง (บรรทัดที่ 18) รูปที่ 5-7 แสดงตัวอย่างการเพิ่มพจน์ $7x$ เข้าใน $1x^0 - 3x^2$ ซึ่งต้องเพิ่มไว้หลังพจน์ $1x^0$ เพื่อให้รายการที่เก็บเรียงลำดับตามเลขชี้กำลัง

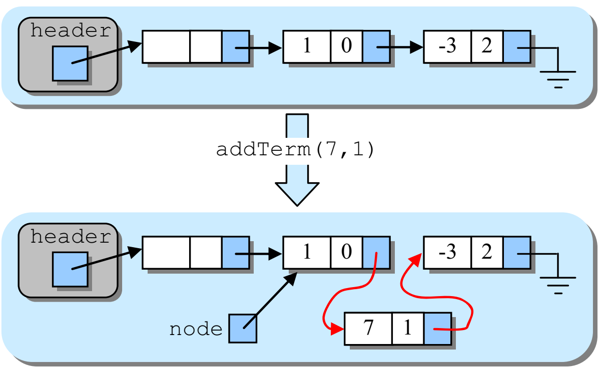
```
13    public void addTerm(double c, int e) {
14      LinkedNode node = header;
15      while( node.next != null && node.next.exp < e) {
16        node = node.next;
17      }
18      node.next = new LinkedNode(c, e, node.next);
19    }
..    ...
```

หาตำแหน่งที่จะแทรก

เพิ่มหลังปมที่หาได้

รหัสที่ 5-18 เมท็อด `addTerm` เพิ่มปมใหม่ในรายการให้เรียงตามเลขชี้กำลัง

มาเขียนเมท็อด `diff` ที่คืนอนุพันธ์ของฟังก์ชันพหุนามที่เก็บอยู่ วิธีการง่าย ๆ ก็คือการผลิตฟังก์ชันพหุนามตัวใหม่ แล้วค่อย ๆ เพิ่มพจน์ซึ่งเป็นอนุพันธ์ของแต่ละพจน์ของที่เก็บ แสดงได้ดังรหัสที่ 5-19 เริ่มด้วยการเตรียมผลลัพธ์ `r` จากนั้นให้ `node` ชี้พจน์แรก (ซึ่งเก็บในปมถัดจาก `header`) แล้วเข้าวงวนทำไปเรื่อย ๆ ตราบเท่าที่ยังมีปมข้อมูลเหลือ ภายในวงวนก็เพียงแต่เพิ่มพจน์ $ce(x^{e-1})$ ซึ่งเป็นอนุพันธ์ของ cx^e เข้าไปใน `r` แล้วก็เลื่อน `node` ไปยังปมถัดไป ทำเสร็จครบทุกพจน์ก็คืน `r` กลับไปเป็นผลลัพธ์



รูปที่ 5-7 ตัวอย่างการเพิ่มพจน์ $7x$ เข้าใน $1 - 3x^2$ กลายเป็น $1 + 7x - 3x^2$

```
public Polynomial diff() {
  Polynomial r = new Polynomial();
  LinkedNode node = this.header.next;
  while( node != null ) {
    r.addTerm(node.coef * node.exp, node.exp - 1);
    node = node.next;
  }
  return r;
}  ...
```

สร้าง r ไว้เก็บผลลัพธ์

อนุพันธ์ของแต่ละพจน์

รหัสที่ 5-19 เมท็อด `diff` แบบช้า

`diff` ในรหัสที่ 5-19 ทำงานช้า เนื่องจากการเรียก `r.addTerm` แต่ละครั้งเกิดการเพิ่มพจน์ใหม่ต่อท้ายรายการตลอดทุก ๆ รอบของวงวน (เพราะพจน์ที่ได้จาก `node` มีเลขชี้กำลังเพิ่มขึ้นเรื่อย ๆ) ดังนั้นถ้าฟังก์ชันมี n พจน์ การต่อท้าย `r` ในรอบที่ 1, 2, 3, ..., n ต้องวิ่งผ่าน 0, 1, 2, ..., $n - 1$ ปม ตามลำดับ ทำให้ใช้เวลารวมเป็น $0 + 1 + 2 + ... + (n - 1) = \Theta(n^2)$ เมื่อทราบอยู่แล้วว่า ต้องนำพจน์ใหม่ต่อท้าย `r` เสมอ ก็อย่าใช้บริการ `addTerm` เลย ควรหันมาจำปมท้ายของ `r` ไว้ แล้วเขียนรหัสการเพิ่มเอง ก็ไม่ต้องเสียเวลาวิ่งไปยังตัวท้ายของ `r` ดังแสดงในรหัสที่ 5-20 เริ่มด้วยการสร้างผลลัพธ์ `r` (บรรทัดที่ 21) มีตัวแปร `rnode` ไว้จำปมท้ายของรายการ `r` เริ่มต้นปมท้ายก็คือ `r.header` นั่นเอง (บรรทัดที่ 22) จากนั้นเตรียมตัวแปร `node` และเข้าสู่วงวนในลักษณะเดียวกับที่ทำในรหัสที่ 5-19 เพียงแต่ภายในวงวนจะสร้างปมข้อมูลใหม่ที่แทนอนุพันธ์ของพจน์ใน `node` แล้วเพิ่มไว้หลังปมที่ `rnode` ชี้ (บรรทัดที่ 25 และ 26) จากนั้นเลื่อนทั้ง `rnode` และ `node` ไปหนึ่งตำแหน่งแล้ววนกลับไปทำรอบถัดไปจนกว่าจะหมด ด้วยวิธีนี้ แต่ละรอบของวงวนทำงานในเวลาคงตัว เนื่องจากเราต้องหมุนในวงวนเท่ากับจำนวนพจน์ จึงใช้เวลาเป็น $\Theta(n)$ (สำหรับเมท็อด `add` และ `multiply` นั้นขอละไว้เป็นแบบฝึกหัดให้ผู้อ่านทำเอง)

```
20   public Polynomial diff() {
21     Polynomial r = new Polynomial();
22     LinkedNode rnode = r.header;
23     LinkedNode node = header.next;
24     while( node != null ) {
25       rnode.next = new LinkedNode(node.coef * node.exp,
26                                   node.exp - 1, null);
27       rnode = rnode.next;
28       node = node.next;
29     }
30     return r;
31   }
..   ...
```

หาอนุพันธ์ของแต่ละพจน์แล้วเพิ่มต่อท้ายไปเรื่อย ๆ

รหัสที่ 5-20 เมท็อด `diff` แบบเร็ว

เวกเตอร์มากเลขศูนย์

เมื่อใดที่เราต้องการใช้เวกเตอร์ในโปรแกรม ก็ย่อมนึกถึงแถวลำดับ เช่น เวกเตอร์ (0,3,0,5) ก็แทนด้วยแถวลำดับ `double[] v = {0,3,0,5}` เวกเตอร์ที่ยาว *n* ก็ใช้แถวลำดับ *n* ช่อง แต่ถ้าเราทราบก่อนล่วงหน้าว่า ข้อมูลส่วนใหญ่ในเวกเตอร์มีค่าเป็นศูนย์ ซึ่งเรียกกันว่า *เวกเตอร์มากเลขศูนย์* (sparse vector) เราสามารถจัดเก็บเวกเตอร์แบบนี้ด้วยรายการที่เก็บเฉพาะข้อมูลที่ไม่ใช่ศูนย์ ซึ่งประหยัดเนื้อที่ได้มาก แถมอาจทำให้การประมวลผลเวกเตอร์บางอย่างเร็วขึ้นด้วย โดยต้องเก็บหมายเลขดัชนีกำกับข้อมูลแต่ละตัวด้วย เช่น เวกเตอร์ (0, 0, 7, 0, 3, 0) ก็เก็บเป็นรายการ 〈 (2,7), (4,3) 〉 เพราะช่องที่ 2 และ 4 ของเวกเตอร์มีค่า 7 และ 3 ตามลำดับ นอกนั้นเป็นศูนย์หมด คลาสที่จะเขียนต่อไปนี้ชื่อ `SparseVector` แทนการจัดเก็บเวกเตอร์มากเลขศูนย์ด้วยรายการ ให้บริการต่าง ๆ ดังแสดงในตารางที่ 5-3

รหัสที่ 5-21 แสดงส่วนต้น ๆ ของคลาส `SparseVector` ที่สร้างรายการด้วยแถวลำดับภายในประกอบด้วย `elementData` และ `size` (เหมือน `ArrayList`) มีตัวแปร `length` เก็บขนาดของเวกเตอร์ ซึ่งผู้ใช้ส่งให้ตอนสร้าง เช่น คำสั่ง `v = new SparseVector(100)` แทนการสร้างเวกเตอร์ขนาด 100 ช่อง ทุกช่องมีค่าศูนย์หมด ภายในตัวสร้างให้ `elementData` มีขนาดศูนย์ช่องเพราะเราจะเขียนโปรแกรมให้ขยายขนาดได้เมื่อจำเป็น ดังเช่นที่ได้ทำมาใน `ArrayList` แต่ละช่องเก็บอ็อบเจกต์ของคลาส `Element` ซึ่งแทนข้อมูลในลักษณะคู่ลำดับ `(index,value)` โดย `value` คือข้อมูลที่ไม่ใช่ศูนย์ ณ ตำแหน่ง `index` ของเวกเตอร์

ตารางที่ 5-3 หน้าที่ของเมท็อดต่าง ๆ ในคลาส `SparseVector`

เมท็อด	หน้าที่
`int length()`	คืนความยาวของเวกเตอร์
`double get(int index)`	คืนค่าในช่องที่ `index` ของเวกเตอร์
`void set(int index, double value)`	ตั้งค่า `value` ให้ช่องที่ `index`
`SparseVector add(SparseVector v2)`	คืนผลบวกของเวกเตอร์นี้กับ `v2`
`double dot(SparseVector v2)`	คืนผลคูณจุดของเวกเตอร์นี้กับ `v2`
`SparseVector multiply(double c)`	คืนผลคูณของเวกเตอร์นี้กับค่าคงตัว `c`
`SparseVector multiply(SparseMatrix m)*`	คืนผลคูณของเวกเตอร์นี้กับเมทริกซ์ `m`

* เราจะนำเสนอคลาส `SparseMatrix` ในหัวข้อย่อยถัดไป

```
01 public class SparseVector {
02   private static class Element {
03     int index;
04     double value;
05     Element(int index, double value) {
06       this.index = index;
07       this.value = value;
08     }
09   }
10   private Element[] elementData;
11   private int size;
12   private int length;
13
14   public SparseVector(int length) {
15     this.elementData = new Element[0];
16     this.size = 0;
17     this.length = length;
18   }
19   public int length() {
20     return length;
21   }
..   ...
```

elementData และ size แทนตัวรายการ

length แทนความยาวของเวกเตอร์

รหัสที่ 5-21 SparseVector แทนเวกเตอร์มากเลขศูนย์ด้วยรายการของข้อมูลที่ไม่ใช่ศูนย์

คู่ลำดับต่าง ๆ ในรายการถูกจัดเก็บให้เรียงตามดัชนีของเวกเตอร์จากน้อยไปมาก เพราะทำให้การประมวลผลเวกเตอร์รวดเร็วขึ้น รหัสที่ 5-22 แสดงเมท็อด get ที่คืนข้อมูลตำแหน่ง index ในเวกเตอร์ โดยค้นดัชนีของคู่ลำดับใน elementData พบที่ใด ก็คืน value ของคู่ลำดับนั้น แต่จะเลิกค้นเมื่อค้นหมดแถว หรือพบดัชนีที่มีค่ามากกว่าตัวที่ต้องการ (เพราะเราเก็บแบบเรียงตามดัชนีจากน้อยไปมาก) หลุดจากวงวน แสดงว่าหาไม่พบ ก็คืน 0 (เมท็อด assertInRange ที่ถูกเรียกในบรรทัดที่ 23 ทำหน้าที่ตรวจสอบให้มั่นใจว่า ค่าของ index ที่ได้รับอยู่ในช่วงของเวกเตอร์ที่ถูกต้อง)

```
22   public double get(int index) {
23     assertInRange(index);
24     for (int i=0; i<size && elementData[i].index <= index; i++) {
25       if (elementData[i].index == index)
26         return elementData[i].value;
27     }
28     return 0.0;
29   }
30   private void assertInRange(int index) {
31     if (index < 0 || index >= length)
32       throw new IndexOutOfBoundsException("" + index);
33   }
..   ...
```

ค้นหา index ใน elementData

ค้นไม่หา คืน 0

รหัสที่ 5-22 เมท็อด get คืนค่าในช่องที่ index ของเวกเตอร์

รหัสที่ 5-23 แสดงการทำงานของเมท็อด `set` เริ่มด้วยการตรวจสอบค่า `index` ว่าต้องอยู่ในช่วง (บรรทัดที่ 35) จากนั้นตรวจสอบ `value` ว่าต้องไม่เป็นศูนย์จึงจะทำต่อ (เพราะเราสนใจเก็บเฉพาะค่าที่ไม่ใช่ศูนย์) เนื่องจากเราจัดเก็บข้อมูลให้เรียงตามดัชนีของเวกเตอร์ จึงต้องวิ่งหาที่แทรกให้เหมาะสมด้วยวงวนในบรรทัดที่ 37 ถึง 39 ถ้าพบเลขดัชนีในรายการซ้ำกับที่จะตั้งค่า ก็เพียงเปลี่ยน `value` ให้เป็นค่าใหม่ที่ได้รับ (บรรทัดที่ 41) แต่ถ้าเป็น `index` ใหม่ ก็แทรก ณ ตำแหน่งที่พบ (บรรทัดที่ 43) โดยใช้ `add(i,index,value)` ซึ่งรับผิดชอบการสร้าง `Element` ที่มี `index` และ `value` แล้วแทรกใน `elementData` ตรงตำแหน่ง `i` (ถ้า `elementData` มีขนาดไม่พอ ก็ขยายด้วย) การแทรกอาศัยการเลื่อนข้อมูลจาก `i` จนถึงช่อง `size-1` ไปทางขวาหนึ่งตำแหน่ง (บรรทัดที่ 49 ถึง 51)

```
34    public void set(int index, double value) {
35      assertInRange(index);
36      int i = 0;
37      while (i < size && elementData[i].index < index) {
38        i++;
39      }
40      if (i < size && elementData[i].index == index) {
41        elementData[i].value = value;
42      } else {
43        add(i, index, value);
44      }
45    }
46    void add(int i, int index, double value) {
47      if (value != 0) {
48        ensureCapacity(size + 1);
49        for(int k=size; k>i; k--) {
50          elementData[k] = elementData[k-1];
51        }
52        elementData[i] = new Element(index, value);
53        ++size;
54      }
55    }
..    ...
```

หาที่แทรกข้อมูลใน elementData

มี index อยู่แล้ว ก็แค่เปลี่ยนค่า

ไม่มี index เก็บอยู่ ก็สร้างและเพิ่มเข้าไป

ขยายขนาดถ้าจำเป็น

ย้ายข้อมูลไปทางขวาเพื่อสร้างช่องว่าง

เพิ่มค่าใหม่ในช่องที่ i

รหัสที่ 5-23 เมท็อด `set` ตั้งค่า `value` ให้กับช่องที่ `index` ของเวกเตอร์

รหัสที่ 5-24 แสดงการทำงานของเมท็อด `add(v2)` เพื่อหาผลรวมของเวกเตอร์ `this` กับ `v2` เพื่อให้การนำเสนอเข้าใจง่ายขึ้น ขอให้ชื่อเวกเตอร์ `this` ว่า `v1` (บรรทัดที่ 58) ดังนั้นเมท็อดนี้มีหน้าที่คำนวณให้ `v3=v1+v2` เช่น `v1` = [0,5,0,2] แทนด้วยรายการ ⟨ (1,5), (3,2) ⟩ และ `v2` = [9,3,0,0] แทนด้วยรายการ ⟨ (0,9), (1,3) ⟩ จะได้ `v3` = [9,8,0,2] แทนด้วยรายการ ⟨(0,9),(1,8),(3,2)⟩ เริ่มด้วยการเตรียมเวกเตอร์ `v3` ในบรรทัดที่ 59 แล้วเข้าวงวนหาผลรวม เนื่องจากพจน์ต่าง ๆ ในเวกเตอร์เรียงลำดับตามดัชนี การเทียบดัชนีของแต่ละพจน์ใน `v1` และ `v2` จึงไล่เปรียบเทียบทีละตัว

จากซ้ายไปขวา (ใช้ตัวแปร i1 และ i2 อ้างอิงพจน์ต่าง ๆ ใน elementData ของ v1 และ v2 ตามลำดับ) ถ้าพจน์ใดมีดัชนีน้อยกว่า ก็นำพจน์นั้นไปเพิ่มต่อท้าย v3 แล้วกระเถิบไปพิจารณาพจน์ถัดไปของเวกเตอร์ (บรรทัดที่ 64 ถึง 67) แต่ถ้าดัชนีเท่ากันก็รวม value ของทั้งสองพจน์เพื่อเพิ่มต่อท้ายใน v3 แล้วเลื่อนพจน์ไปหนึ่งตำแหน่งของทั้งสองเวกเตอร์ (บรรทัดที่ 69 และ 70) วงวนนี้จะหมุนทำงานตราบเท่าที่ทั้งสองเวกเตอร์ยังมีพจน์เหลือให้พิจารณา เมื่อหลุดออกจากวงวนแรกแล้ว ก็นำพจน์ที่เหลือในเวกเตอร์ไปเพิ่มต่อท้ายใน v3 ให้หมด (ซึ่งคือวงวนอีกสองวงในบรรทัดที่ 73 ถึง 80) การเพิ่มพจน์ใน v3 ทั้งหมดเป็นการเพิ่มต่อท้าย (อาศัยการเรียกเมท็อดภายในชื่อ append บรรทัดที่ 83) เนื่องจากพจน์ที่นำมาพิจารณาเรียงลำดับตามดัชนีจากน้อยไปมาก

```
56   public SparseVector add(SparseVector v2) {
57     assertEqualLength(v2);
58     SparseVector v1 = this;
59     SparseVector v3 = new SparseVector(v1.length);
60     int i1 = 0, i2 = 0;
61     while (i1 < v1.size && i2 < v2.size) {
62       Element e1 = v1.elementData[i1];
63       Element e2 = v2.elementData[i2];
64       if (e1.index < e2.index) {
65         v3.append(e1.index, e1.value); i1++;
66       } else if (e1.index > e2.index) {
67         v3.append(e2.index, e2.value); i2++;
68       } else {
69         v3.append(e1.index, e1.value + e2.value);
70         i1++; i2++;
71       }
72     }
73     while (i1 < v1.size) {
74       Element e1 = v1.elementData[i1++];
75       v3.append(e1.index, e1.value);
76     }
77     while (i2 < v2.size) {
78       Element e2 = v2.elementData[i2++];
79       v3.append(e2.index, e2.value);
80     }
81     return v3;
82   }
83   void append(int index, double value) {
84     add(size, index, value);
85   }
86   private void assertEqualLength(SparseVector v) {
87     if (this.length() != v.length())
88       throw new IllegalArgumentException();
89   }
..   ...
```

e1 และ e2 ตัวใดมีดัชนีน้อยกว่า นำไปไว้ที่ผลรวมก่อน

ถ้า e1 และ e2 มีดัชนีเดียวกัน ให้รวม value เป็นผลลัพธ์

นำพจน์ที่เหลือใน v1 ไปต่อท้ายผลลัพธ์

นำพจน์ที่เหลือใน v2 ไปต่อท้ายผลลัพธ์

นี่คือการต่อท้ายรายการ

รหัสที่ 5-24 เมท็อด add คืนผลบวกของเวกเตอร์ this กับ v2

รหัสที่ 5-25 แสดงเมท็อด `dot` ซึ่งคำนวณผลคูณจุด (dot product) ของเวกเตอร์ที่ถูกเรียกกับ `v2` ผลคูณจุดของเวกเตอร์คำนวณด้วยสูตร $a \cdot b = \sum_{i=0}^{n-1} a_i b_i$ นั่นคือการนำค่าที่เก็บในดัชนีที่เหมือนกันมาคูณแล้วรวมทุก ๆ ผลคูณเข้าด้วยกัน การทำงานจึงคล้ายกับเมท็อด `add` โดยอาศัยการนำพจน์ต่าง ๆ ของเวกเตอร์ทั้งสองจากซ้ายไปขวา หาคู่ที่มีดัชนีเหมือนกันมาคูณกัน แล้วรวมเข้าในตัวแปร `r` ซึ่งมีไว้เก็บผลลัพธ์ (บรรทัดที่ 103) เมื่อใดที่เปรียบเทียบดัชนีแล้วไม่เท่ากัน ก็ให้เลื่อนตำแหน่งในเวกเตอร์ของพจน์ที่มีดัชนีค่าน้อยกว่า (บรรทัดที่ 98 ถึง 102) วนการทำงานในลักษณะเช่นนี้จนกระทั่งมีสักหนึ่งเวกเตอร์ไม่มีพจน์เหลือให้พิจารณา (เราไม่ต้องพิจารณาพจน์ที่เหลือของอีกเวกเตอร์เพราะพจน์เหล่านั้นคูณด้วยศูนย์หมด)

```
 90    public double dot(SparseVector v2) {
 91      assertEqualLength(v2);
 92      SparseVector v1 = this;
 93      double r = 0;
 94      int i1 = 0, i2 = 0;
 95      while (i1 < v1.size && i2 < v2.size) {
 96        Element e1 = v1.elementData[i1];
 97        Element e2 = v2.elementData[i2];
 98        if (e1.index < e2.index) {
 99          i1++;
100        } else if (e1.index > e2.index) {
101          i2++;
102        } else {
103          r += e1.value * e2.value;
104          i1++;  i2++;
105        }
106      }
107      return r;
108    }
 ..    ...
```

ถ้าดัชนีไม่เท่ากัน ให้กระเถิบตัวน้อยไปหนึ่งตำแหน่ง

ดัชนีเท่ากัน หาผลคูณ แล้วรวมเข้าในผลลัพธ์

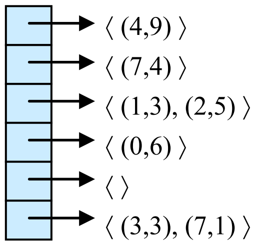
รหัสที่ 5-25 เมท็อด `dot` คืนค่าผลคูณจุดของเวกเตอร์ `this` กับ `v2`

ด้วยการแทนเวกเตอร์มากเลขศูนย์ในลักษณะที่นำเสนอมา การทำงานของเมท็อดต่าง ๆ จึงแปรผันโดยตรงกับจำนวนพจน์ที่ไม่ใช่ศูนย์ในเวกเตอร์ แทนที่จะแปรผันโดยตรงกับความยาวของเวกเตอร์ถ้าเราแทนเวกเตอร์ด้วยแถวลำดับแบบดัชนีละช่อง (สำหรับ `multiply(c)` ซึ่งคือการคูณด้วยค่าคงตัว ขอละไว้เป็นแบบฝึกหัด ส่วนเมท็อด `multiply(m)` ซึ่งคือการคูณเวกเตอร์ด้วยเมทริกซ์ จะได้อธิบายรายละเอียดในหัวข้อถัดไป)

เมทริกซ์มากเลขศูนย์

เราสามารถขยายแนวคิดของการสร้างเวกเตอร์มากเลขศูนย์มาสร้าง*เมทริกซ์มากเลขศูนย์* (sparse matrix) โดยจัดเก็บเมทริกซ์ด้วยแถวลำดับมิติเดียวของเวกเตอร์มากเลขศูนย์หลาย ๆ แถว (ดูตัวอย่างในรูปที่ 5-8) ตารางที่ 5-4 สรุปหน้าที่ของเมท็อดต่าง ๆ ใน `SparseMatrix`

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 9 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 4 \\ 0 & 3 & 5 & 0 & 0 & 0 & 0 & 0 \\ 6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 & 0 & 0 & 1 \end{bmatrix}$$



รูปที่ 5-8 การเก็บเมทริกซ์มากเลขศูนย์ด้วยแถวลำดับของเวกเตอร์มากเลขศูนย์

ตารางที่ 5-4 หน้าที่ของเมท็อดต่าง ๆ ในคลาส `SparseMatrix`

เมท็อด	หน้าที่
`int numRows()`	คืนจำนวนแถวแนวนอนของเมทริกซ์
`int numCols()`	คืนจำนวนแถวแนวตั้งของเมทริกซ์
`double get(int r, int c)`	คืนค่าในเมทริกซ์ที่ช่อง `(r,c)`
`void set(int r, int x, double v)`	ตั้งค่า `v` ให้ช่อง `(r,c)` ในเมทริกซ์
`SparseMatrix add(SparseMatrix m)`	คืนผลบวกของเมทริกซ์นี้กับเมทริกซ์ `m`
`SparseVector multiply(SparseVector v)`	คืนผลคูณของเมทริกซ์นี้กับเวกเตอร์ `v`
`SparseMatrix multiply(SparseMatrix m)`	คืนผลคูณของเมทริกซ์นี้กับเมทริกซ์ `m`

รหัสที่ 5-26 แสดงคลาส `SparseMatrix` ที่แทนเมทริกซ์ที่สร้างด้วยแถวลำดับของเวกเตอร์ (บรรทัดที่ 2) ตัวสร้างรับขนาดของเมทริกซ์ `r×c` เพื่อจองแถวลำดับจำนวน `r` ช่อง และสร้างเวกเตอร์ขนาดความยาว `c` เก็บตามช่องต่าง ๆ (บรรทัดที่ 5 ถึง 7)

```
01 public class SparseMatrix {
02   SparseVector [] rows;
03
04   public SparseMatrix(int r, int c) {
05     rows = new SparseVector[r];
06     for(int i=0; i<r; i++)
07       rows[i] = new SparseVector(c);
08   }
09   public int numRows() { return rows.length; }
10   public int numCols() { return rows[0].length(); }
..   ...
```

บริการคืนจำนวนแถวแนวนอน และแนวตั้ง

รหัสที่ 5-26 `SparseMatrix` เก็บเมทริกซ์ด้วยแถวลำดับของเวกเตอร์

รหัสที่ 5-27 แสดงการตั้งค่าและการหยิบค่าในเมทริกซ์ ซึ่งเรียกใช้บริการของเวกเตอร์อีกทอดหนึ่ง นั่นคือ เมท็อด `set(r,c,v)` เรียกบริการ `set(c,v)` ของเวกเตอร์ใน `rows[r]` และทำนองเดียว `get(r,c)` ก็คืนค่าในช่องที่ `c` ของเวกเตอร์ที่เก็บใน `rows[r]`

```
  public void set(int r, int c, double v) {
    assertInRange(r, c);
    rows[r].set(c,v);
  }
  public double get(int r, int c) {
    assertInRange(r, c);
    return rows[r].get(c);
  }
  private void assertInRange(int r, int c) {
    if (r < 0 || r >= numRows() || c < 0 || c >= numCols())
      throw new IndexOutOfBoundsException(r + "," + c);
  }
  ...
```

บริการตรวจสอบว่าเป็นตำแหน่งที่ถูกต้องของเมทริกซ์

รหัสที่ 5-27 เมท็อด set และ get ของ SparseMatrix

รหัสที่ 5-28 แสดงการบวกเมทริกซ์สองตัวโดยการบวกเวกเตอร์ทีละแถวของทั้งสองเมทริกซ์

```
  public SparseMatrix add(SparseMatrix m2) {
    SparseMatrix m1 = this;
    if (m1.numRows()!=m2.numRows() || m1.numCols()!=m2.numCols())
      throw new IllegalArgumentException();
    SparseMatrix m3 = new SparseMatrix(
                          m1.numRows(), m1.numCols());
    for(int i=0; i<m3.numRows(); i++) {
      m3.rows[i] = m1.rows[i].add(m2.rows[i]);
    }
    return m3;
  }
  ...
```

หาผลรวมของเวกเตอร์ในแถวเดียวกันของเมทริกซ์ทั้งสอง

รหัสที่ 5-28 เมท็อด add ของ SparseMatrix

กำหนดให้ `m` คือเมทริกซ์ และ `v` คือเวกเตอร์ `m.multiply(v)` แทน m×v ในขณะที่ `v.multiply(m)` แทน v×m ซึ่งไม่เหมือนกัน รหัสที่ 5-29 แสดงบริการของการคูณแบบแรก ส่วนการคูณอีกแบบนั้นอยู่ในคลาส `SparseVector` (ซึ่งยังไม่ได้นำเสนอเลย) การหาผลคูณ m×v ทำได้ง่ายกว่า ดังตัวอย่างที่แสดงข้างล่างนี้ ด้วยการสร้างเวกเตอร์ที่แต่ละช่องคือผลคูณจุดของ `v` กับแต่ละแถวใน `m` ดังแสดงในรหัสที่ 5-29

$$m = \begin{bmatrix} 4 & 0 \\ 5 & 1 \\ 0 & 3 \end{bmatrix}, v = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \quad m \times v = \begin{bmatrix} 4 & 0 \\ 5 & 1 \\ 0 & 3 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} [4 \quad 0] \cdot [1 \quad 2] \\ [5 \quad 1] \cdot [1 \quad 2] \\ [0 \quad 3] \cdot [1 \quad 2] \end{bmatrix} = \begin{bmatrix} 4 \\ 7 \\ 6 \end{bmatrix}$$

```
  public SparseVector multiply(SparseVector v) {
    if (v.length() != numCols())
      throw new IllegalArgumentException();
    SparseVector r = new SparseVector(numRows());
    for(int i=0; i<numRows(); i++) {
      r.append(i, rows[i].dot(v));
    }
    return r;
  }
  ...
```

แต่ละค่าของเวกเตอร์ผลลัพธ์คือ ผลคูณจุดของแต่ละแถวนอนของเมทริกซ์กับ v

รหัสที่ 5-29 เมท็อด `multiply(v)` คืนผลการคูณเมทริกซ์นี้ด้วยเวกเตอร์ `v`

สำหรับการคูณเวกเตอร์ด้วยเมทริกซ์ ที่เราจะเขียนให้กับคลาส `SparseVector` นั้น จะมีความซับซ้อนเล็กน้อย ดังตัวอย่างที่แสดงข้างล่างนี้

$$v=\begin{bmatrix}2 & 1\end{bmatrix},\ m=\begin{bmatrix}2 & 0 & 3\\ 0 & 4 & 5\end{bmatrix},\quad v\times m=\begin{bmatrix}2 & 1\end{bmatrix}\begin{bmatrix}2 & 0 & 3\\ 0 & 4 & 5\end{bmatrix}=\begin{bmatrix}4 & 4 & 11\end{bmatrix}$$

จะเห็นว่า เราต้องหาผลคูณจุดของ `v` ซึ่งคือ [2 1] กับสามเวกเตอร์ตามแนวตั้งของ `m` คือ [2 0] [0 4] และ [3 5] ซึ่งได้ผลเป็น 4 4 และ 11 ตามลำดับ การหยิบเวกเตอร์ทั้งสามในแนวตั้งของ `m` นั้นทำลำบาก เพราะเราเก็บแยกแถวแนวนอนละเวกเตอร์ อย่างไรก็ตามมีวิธีการคูณอีกแบบที่สะดวกกว่า คือนำแต่ละค่าที่ `i` ของ `v` ไปคูณกับแต่ละเวกเตอร์แนวนอนแถวที่ `i` ของ `m` ตามลำดับ แล้วรวมผลลัพธ์ทั้งหมดเข้าด้วยกัน จากตัวอย่างก็คือ $2 \cdot [2\ 0\ 3]\ +\ 1 \cdot [0\ 4\ 5] = [4\ 0\ 6] + [0\ 4\ 5]\ = [4\ \ 4\ \ 11]$ วิธีนี้ง่ายกว่าเพราะยุ่งกับเวกเตอร์แนวนอนของเมทริกซ์ซึ่งหยิบใช้ง่าย เขียนเป็นเมท็อด `multiply(m)` ให้กับ `SparseVector` ได้ดังรหัสที่ 5-30

```
public class SparseVector {
  ...
  public SparseVector multiply(SparseMatrix m) {
    if (this.length != m.numRows())
      throw new IllegalArgumentException();
    SparseVector r = new SparseVector(m.numCols());
    for(int i=0; i<this.length(); i++) {
      r = r.add(m.rows[i].multiply(this.get(i)));
    }
    return r;
  }
```

รหัสที่ 5-30 เมท็อด `multiply(m)` ของคลาส `SparseVector`

รหัสที่ 5-30 เริ่มตรวจสอบความยาวของเวกเตอร์ `this` ว่าต้องเท่ากับจำนวนแถวของเมทริกซ์ `m` จึงจะคูณกันได้ ต่อด้วยการเตรียมผลลัพธ์ `r` ที่มีความยาวเท่ากับจำนวนแถวแนวตั้งของเมทริกซ์ `m` แล้วเริ่มเข้าวงวนนำค่าที่ดัชนี `i` ของเวกเตอร์ไปคูณกับเวกเตอร์ของแถวที่ `i` ใน `m` ได้ผลลัพธ์นำไป

บวกสะสมเข้าใน `r` ด้วยคำสั่ง `r=r.add(m.rows[i].multiply(this.get(i)))` หมุนจนครบทุกตัวในเวกเตอร์ ได้ผลลัพธ์เก็บใน `r`

กลับมาที่คลาส `SparseMatrix` รหัสที่ 5-31 แสดงการคูณเมทริกซ์ด้วยเมทริกซ์ การเรียก `m1.multiply(m2)` จะนำเวกเตอร์แต่ละแถวแนวนอนที่ `i` ของ `m1` ไปคูณกับเมทริกซ์ `m2` (โดยใช้เมท็อด `multiply(m2)` ของคลาส `SparseVector` ที่ได้อธิบายในหน้าที่แล้ว) ได้ผลเป็นเวกเตอร์ซึ่งก็คือแถวแนวนอนที่ `i` ของเมทริกซ์ผลลัพธ์ เช่น ในตัวอย่างข้างล่างนี้คือการนำแถวที่ 0 และ 1 ของ `m1` ซึ่งคือ [2 0 3] และ [0 4 1] ตามลำดับ ไปคูณกับ `m2` ได้ผลเป็น [5 10] และ [9 6] ซึ่งคือแถวที่ 0 และ 1 ของเมทริกซ์ผลลัพธ์

$$m1 \times m2 = \begin{bmatrix} 2 & 0 & 3 \\ 0 & 4 & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 2 & 1 \\ 1 & 2 \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} 2 & 0 & 3 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 2 & 1 \\ 1 & 2 \end{bmatrix} \\ \begin{bmatrix} 0 & 4 & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 2 & 1 \\ 1 & 2 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} 5 & 10 \\ 9 & 6 \end{bmatrix}$$

```
  public SparseMatrix multiply(SparseMatrix m2) {
    SparseMatrix m1 = this;    //  m3 = m1 x m2
    if (m1.numCols() != m2.numRows())
      throw new IllegalArgumentException();
    SparseMatrix m3 = new SparseMatrix(m1.numRows(),m2.numCols());
    for(int i=0; i<m1.numRows(); i++) {
      m3.rows[i] = m1.rows[i].multiply(m2);
    }
    return m3;
  }
  ...
```

รหัสที่ 5-31 เมท็อด `multiply(m)` คืนผลการคูณเมทริกซ์นี้ด้วยเมทริกซ์ `m2`

แบบฝึกหัด

1. จงเขียนคลาส `ArrayList` และ `LinkedList` ด้วยตนเอง โดยไม่ดูรายละเอียดในหนังสือ
2. จงเขียนคลาส `ArrayList` ให้เป็นคลาสลูกของ `ArrayCollection`
3. จงเขียนเมท็อด `swap(int i, int j)` เพื่อสลับปมที่ `i` และ `j` ในรายการโยงแบบ `LinkedList` (ขอเน้นว่าให้สลับตัวปม ไม่ใช่ให้สลับข้อมูลที่เก็บในปม)

4. คลังคลาสมาตรฐานของจาวา ก็มีอินเทอร์เฟซ `List`, คลาส `ArrayList` และ `LinkedList` เหมือนกัน (อยู่ในชุด `java.util`) โดยมีโครงสร้างภายในคล้ายกับที่เราได้นำเสนอ จงอธิบายบริการต่าง ๆ ที่คลาสทั้งสองมีให้

5. จงสร้างคลาสให้กับรายการโยง ที่มีลักษณะดังนี้

   5.1. รายการโยงเดี่ยวแบบไม่วนไม่มีปมหัว (รูปที่ 5-3 (1))

   5.2. รายการโยงเดี่ยวแบบวนไม่มีปมหัว (รูปที่ 5-3 (3))

   5.3. รายการโยงคู่แบบไม่วนไม่มีปมหัว (รูปที่ 5-3 (4))

   5.4. รายการโยงคู่แบบไม่วนมีปมหัว (รูปที่ 5-3 (5))

6. จงเขียนเมท็อดต่อไปนี้ให้กับ `ArrayList`, `SinglyLinkedList`, และ `LinkedList` (โดยใช้เนื้อที่เสริมในการทำงานน้อย ๆ และทำงานได้อย่างรวดเร็ว)

   6.1. `getFirst()` เพื่อคืนข้อมูลตัวแรกสุดของรายการ

   6.2. `getLast()` เพื่อคืนข้อมูลตัวท้ายสุดของรายการ

   6.3. `removeFirst()` เพื่อลบข้อมูลตัวแรกสุดของรายการ

   6.4. `removeLast()` เพื่อลบข้อมูลตัวท้ายสุดของรายการ

   6.5. `indexOf(Object e)` เพื่อคืนเลขลำดับในรายการที่พบ `e` เป็นตัวแรกสุด

   6.6. `lastIndexOf(Object e)` เพื่อคืนเลขลำดับในรายการที่พบ `e` เป็นตัวหลังสุด

   6.7. `removeRange(int from, int to)` เพื่อลบข้อมูลตั้งแต่เลขลำดับที่ `from` ถึง `to-1` ในรายการ

   6.8. `shuffle()` เพื่อสลับลำดับของข้อมูลในรายการอย่างสุ่ม

   6.9. `reverse()` เพื่อกลับลำดับของข้อมูลในรายการ เช่น เดิมเป็น $\langle 5, 6, 8, 2 \rangle$ ก็เปลี่ยนให้เป็น $\langle 2, 8, 6, 5 \rangle$ โดยใช้เนื้อที่เสริมจำนวนคงตัวในการกลับลำดับ

7. จงเขียนเมท็อด

```
void cutPaste(LinkedList a, int i, int j, LinkedList b, int k)
```

   เพื่อตัดปมของข้อมูลตัวที่ `i` ถึง `j` ของรายการ `a` ไปใส่เพิ่มไว้ที่หน้าปมของข้อมูลตัวที่ `k` ของรายการ `b`

8. จงเขียนเมท็อด `zip(LinkedList b)` ซึ่งนำปมข้อมูลของรายการ `b` มาผสมกับของรายการโยงของตัวที่ถูกเรียก (ลักษณะเหมือนกับการรูดซิป) เช่น ถ้า `a` คือรายการ ⟨1, 2, 3, 4, 5⟩ และ `b` คือรายการ ⟨A, B, C⟩ เมื่อเรียก `a.zip(b)` จะได้ `a` เป็น ⟨1, A, 2, B, 3, C, 4, 5⟩ และ `b` เป็นรายการว่าง

9. เราได้นำเสนอวิธีการย้ายข้อมูลไปด้านหน้าของรายการที่ปรับตัวเอง ยังมีอีกวิธีในการปรับลำดับข้อมูลในรายการ โดยให้นำข้อมูลที่ถูกค้นพบสลับกับตัวก่อนหน้ามันในรายการ (ตัวใดถูกค้นบ่อยก็จะค่อย ๆ ขยับมาด้านหน้า) จงเขียนคลาสของรายการที่มีพฤติกรรมแบบนี้

10. หากส่งพจน์ cx^e ให้เมท็อด `addTerm` ของคลาส `Polynomial` ในรหัสที่ 5-18 โดยที่ตัวฟังก์ชันพหุนามมีพจน์ที่มีกำลังเป็น e อยู่แล้ว จะเกิดปัญหา จงนำเสนอวิธีแก้ไขให้ถูกต้อง

11. จงเขียนเมท็อดต่อไปนี้เพิ่มให้กับคลาส `Polynomial`

    11.1. `add(Polynomial p)` เพื่อคืนผลบวกของฟังก์ชันพหุนามนี้กับ `p`

    11.2. `multiply(Polynomial p)` เพื่อคืนผลคูณของฟังก์ชันพหุนามนี้กับ `p`

    11.3. `eval(double c)` เพื่อคืนค่าของฟังก์ชันพหุนามเมื่อ x มีค่าเป็น c

    11.4. `order()` เพื่อคืนอันดับของฟังก์ชันพหุนามซึ่งคือเลขชี้กำลังสูงสุดของฟังก์ชัน

12. หากเราส่ง 0.0 ไปให้เมท็อด `set` ของคลาส `SparseVector` และ `SparseMatrix` ตั้งค่าโดยหลักการแล้ว ก็ไม่น่าตั้งค่าให้ เพราะเราไม่ควรเก็บค่าศูนย์ซึ่งสิ้นเปลือง จึงควรเขียนเพิ่มในเมท็อดนี้ให้ตรวจสอบว่า ถ้าค่าที่ให้มาเป็น 0 ก็คืนการทำงานทันที อยากทราบว่า ทำเช่นนี้มีปัญหาหรือไม่ อย่างไร และต้องแก้ไขอย่างไร

13. จงเขียนเมท็อด `multiply(double c)` ให้กับ `SparseVector`

14. จงเขียนเมท็อด `power(int c)` ให้กับ `SparseMatrix` เพื่อคำนวณ m^c โดยเมทริกซ์ที่เรียกได้ต้องเป็นเมทริกซ์จัตุรัส (มีจำนวนแถวแนวนอนเท่ากับแถวแนวตั้ง) ข้อแนะนำ : อย่าใช้วิธีการนำ m มาคูณกัน c ครั้ง ซึ่งช้า ลองคิดหาวิธีที่เร็วกว่า

6 กองซ้อน

กองซ้อน (stack) คือที่เก็บข้อมูลประเภทหนึ่ง มีลักษณะคล้ายรายการ ให้บริการเพิ่ม ลบ และเข้าถึงข้อมูลที่ค่อนข้างจำกัดมาก ๆ คือเป็นเสมือนรายการที่อนุญาตให้เพิ่ม ลบ และดูข้อมูลที่ปลายด้านเดียวของรายการเท่านั้น ทำให้ข้อมูลที่เพิ่มเข้ากองซ้อนก่อน จะถูกลบออกทีหลัง หรือในทางกลับกัน ข้อมูลที่เพิ่มเข้าทีหลัง จะถูกลบออกมาก่อน จึงมีชื่อเรียกกองซ้อนกันว่า LIFO ย่อมาจากคำว่า Last-In-First-Out ถึงแม้จะมีบริการที่ค่อนข้างจำกัด แต่กองซ้อนกลับเป็นโครงสร้างข้อมูลที่ได้รับการประยุกต์ในการแก้ปัญหาต่าง ๆ มากมาย กองซ้อนมีความสำคัญมากถึงขนาดที่หน่วยประมวลผลทั่วไป มีบริการกองซ้อนให้ใช้ในระดับคำสั่งเครื่อง

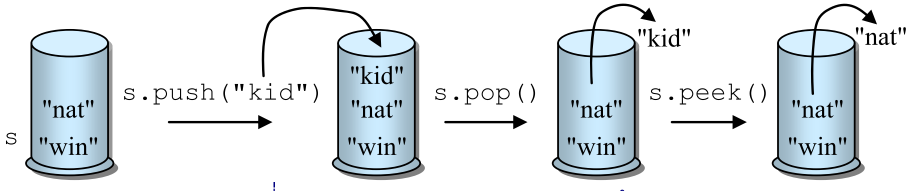
ข้อกำหนดของกองซ้อน

กองซ้อนมีบริการสั้น ๆ ง่าย ๆ บรรยายด้วยอินเทอร์เฟซ `Stack` ดังแสดงในรหัสที่ 6-1 เราเรียกการเพิ่มในกองซ้อนว่า `push` และเรียกการลบข้อมูลที่เพิ่มตัวล่าสุดออกจากกองซ้อนว่า `pop`

```
public interface Stack {
  public boolean isEmpty();    // ถามกองซ้อนว่าว่างหรือไม่
  public int size();           // ถามจำนวนข้อมูลที่เก็บในกองซ้อน
  public void push(Object e); // เพิ่มอ็อบเจกต์ e เข้าเก็บในกองซ้อน
  public Object pop();         // ลบข้อมูลตัวที่เพิ่มตัวล่าสุด
  public Object peek();        // ขอดูข้อมูลตัวที่เพิ่มตัวล่าสุด
}
```

รหัสที่ 6-1 อินเทอร์เฟซ `Stack`

บางคนมองกองซ้อนเสมือนเป็นถังไม่มีฝา ที่รับข้อมูลใหม่ ใส่ลงถังทับข้อมูลเก่าที่เก็บซ้อน ๆ กันในถัง การนำข้อมูลออก ก็คือการนำข้อมูลที่อยู่ตำแหน่งบนสุดในถังออก ดังตัวอย่างการใช้งานกองซ้อนในรูปที่ 6-1



รูปที่ 6-1 การ push pop และ peek ข้อมูล

รหัสที่ 6-2 แสดงการใช้งานกองซ้อนตามตัวอย่างในรูปที่ 6-1 เริ่มด้วยการสร้างกองซ้อนในบรรทัดที่ 3 แล้ว push ข้อมูลเข้าไปสามตัว ตามด้วย pop และ peek มีการถามขนาดของกองซ้อนเพื่อแสดงให้เห็นว่า peek เป็นแค่การขอดูข้อมูลที่อยู่บนกองซ้อน ไม่ได้ลบออกจากกองซ้อน

```
01 public class TestStack {
02   public static void main(String[] args) {
03     Stack s = new ArrayStack();
04     s.push("win");
05     s.push("nat");
06     s.push("kid");
07     System.out.println(s.size());
08     System.out.println(s.pop());
09     System.out.println(s.size());
10     System.out.println(s.peek());
11     System.out.println(s.size());
12   }
13 }
```

```
3
"kid"
2
"nat"
2
```

รหัสที่ 6-2 ตัวอย่างการทำงานของ Stack ในรูปที่ 6-1

การสร้างกองซ้อนด้วยรายการ

เราสามารถสร้างกองซ้อนได้ง่าย ๆ ด้วยการนำรายการมาเก็บซ่อนไว้ในคลาส แล้วให้รายการนี้ทำหน้าที่จัดเก็บและจัดการข้อมูลแทน รหัสที่ 6-3 แสดงตัวอย่างการสร้างด้วยวิธีดังกล่าว มีตัวแปร list เก็บอ็อบเจกต์ของ ArrayList บริการ isEmpty และ size ก็ส่งต่อไปถามที่ตัว list ที่มีหน้าที่เก็บข้อมูล, push ก็คือการเพิ่มข้อมูลต่อท้ายรายการ, peek ก็คือการขอข้อมูลที่ตำแหน่งท้ายของรายการ ส่วน pop ก็คือการลบข้อมูลตัวท้ายของรายการ list

```
public class ArrayListStack implements Stack {
  private ArrayList list = new ArrayList();

  public boolean isEmpty() {
    return list.isEmpty();
  }
  public int size() {
    return list.size();
  }
  public void push(Object e) {
    list.add(list.size(),e);
  }
  public Object peek() {
    if (isEmpty())
      throw new IllegalStateException();
    return list.get(list.size()-1);
  }
  public Object pop() {
    Object e = peek();
    list.remove(list.size()-1);
    return e;
  }
}
```

กองซ้อนนี้ใช้ ArrayList เป็นที่เก็บข้อมูล

ทุกคำสั่งของกองซ้อน สั่งงานต่อไปใช้บริการของ ArrayList

นำข้อมูลต่อท้ายรายการ

ไม่มีข้อมูลให้ดู เกิด exception

หยิบข้อมูลท้ายรายการ

ลบข้อมูลท้ายรายการ

รหัสที่ 6-3 การสร้างกองซ้อนด้วยการใช้รายการจัดเก็บและจัดการข้อมูลแทน

เนื่องจากเราใช้ ArrayList เก็บข้อมูล การ push และ pop จึงกระทำที่ท้ายรายการ เพราะการเพิ่มและลบข้อมูลท้ายรายการที่สร้างด้วยแถวลำดับใช้เวลาคงตัว ไม่ต้องย้ายข้อมูลในแถวลำดับเลย แต่ถ้าเราเลือกที่จะให้ push และ pop เพิ่มและลบข้อมูลที่ต้นรายการแบบ ArrayList จะทำให้ประสิทธิภาพของกองซ้อนแย่มาก ๆ

การสร้างกองซ้อนด้วยแถวลำดับ

การสร้างกองซ้อนด้วยรายการนั้นง่าย นำของที่มีอยู่แล้วมาสร้างของใหม่ แต่เปลือง คืออ็อบเจกต์กองซ้อนมีอ็อบเจกต์ของรายการเก็บอยู่ภายใน การสั่งงานกองซ้อนก็ต้องเสียเวลาสั่งงาน ArrayList ต่ออีกทอดหนึ่ง จึงขอนำแถวลำดับมาสร้างกองซ้อนตรง ๆ เลย (คล้ายกับ ArrayCollection) ให้ช่องที่ 0 (ทางซ้าย) ของแถวลำดับคือก้นกองซ้อน การเพิ่มข้อมูลในกองซ้อนก็คือการนำข้อมูลนั้นไปเก็บต่อทางขวาของตัวท้ายสุดในแถว ในกรณีที่เก็บข้อมูลเต็มแถวแล้ว ก็ขยายขนาดด้วยวิธีที่ได้นำเสนอในบทก่อน ๆ รายละเอียดดังแสดงในรหัสที่ 6-4 ทุกเมท็อดใช้เวลาเป็น $\Theta(1)$ ทั้งสิ้น ยกเว้นเมื่อเพิ่มข้อมูลแล้วแถวลำดับเต็ม

```
public class ArrayStack implements Stack {
  private Object[] elementData = new Object[1];
  private int size;

  public boolean isEmpty() { return size == 0; }
  public int size() { return size; }

  public void push(Object e) {
    if (size == elementData.length) {
      Object[] arr = new Object[2*elementData.length];
      for(int i=0; i<size; i++)
        arr[i] = elementData[i];
      elementData = arr;
    }
    elementData[size++] = e;
  }
  public Object peek() {
    if (isEmpty())
      throw new IllegalStateException();
    return elementData[size-1];
  }
  public Object pop() {
    Object e = peek();
    elementData[--size] = null;
    return e;
  }
}
```

ใช้อาเรย์เก็บข้อมูล

ถ้าเต็ม ก็ขยายขนาด

เพิ่มต่อท้ายอาเรย์

ห้าม peek ถ้ากองซ้อนไม่มีข้อมูล

หยิบตัวท้ายอาเรย์

ลบ reference ออก

รหัสที่ 6-4 การสร้างกองซ้อนด้วยแถวลำดับ

ตัวอย่างการใช้งานกองซ้อน

ด้วยลักษณะของการเข้าหลังออกก่อนของข้อมูลในกองซ้อน ทำให้มีการนำกองซ้อนไปใช้แก้ปัญหาที่มีโครงสร้างในลักษณะที่ซ้อน ๆ กัน ดังตัวอย่างที่จะนำเสนอกันในหัวข้อนี้ คือ การใช้กองซ้อนช่วยตรวจสอบการใส่วงเล็บ ช่วยจัดการทำงานของเครื่องเสมือนจาวา (java virtual machine) และช่วยแปลงนิพจน์เติมกลางไปเป็นนิพจน์เติมหลัง

การตรวจสอบการใส่วงเล็บ

กำหนดให้มีข้อความที่ภายในมีการใช้วงเล็บในหลายรูปแบบ เช่น ( ) { } [ ] การเขียนวงเล็บซ้อนกันต้องจับคู่กันให้ถูกต้อง เช่น `{x=b[5*(x+c[i+2])+8]+(3*d)}` การเขียนวงเล็บที่ผิดแบ่งได้เป็น 3 กรณีคือ กรณีที่มีวงเล็บเปิดมากเกินกว่าปิด เช่น `(b[2+5]` กรณีที่มีวงเล็บเปิดน้อยกว่าปิด เช่น `b[2+5])` และกรณีที่วงเล็บเปิดจับคู่กับวงเล็บปิดคนละประเภทกันเช่น `b[2+5}` เราสามารถใช้กองซ้อนช่วยตรวจสอบความถูกต้องของการใส่วงเล็บ ด้วยขั้นตอนการทำงานดังนี้

1. สร้างกองซ้อน *s*
2. อ่านตัวอักษรในข้อความออกมาทีละตัว
    2.1. ถ้าเป็นวงเล็บเปิด ให้ push ลง *s*
    2.2. ถ้าเป็นวงเล็บปิด ให้ pop วงเล็บเปิดจาก *s* มาตรวจสอบ ถ้าเป็นคนละประเภท ก็แสดงว่าผิด เพราะวงเล็บไม่ตรงประเภทกัน
3. เมื่อใดต้องการ pop จาก *s* แต่ *s* ไม่มีข้อมูลเหลือ ก็แสดงว่าผิด เพราะวงเล็บปิดมีมากไป
4. ถ้าอ่านข้อความจนครบแล้วยังมีวงเล็บเหลือใน *s* แสดงว่าผิด เพราะมีวงเล็บเปิดมากไป

```
01 public static boolean checkParentheses(String t) {
02   String open = "{([", close = "})]";
03   Stack s = new ArrayStack();
04   for (int i = 0; i < t.length(); i++) {
05     String token = t.substring(i, i + 1);
06     if (open.indexOf(token) >= 0) {
07       s.push(token);
08     } else {
09       int k = close.indexOf(token);
10       if (k >= 0)
11         if (s.isEmpty() ||
12             !s.pop().equals(open.substring(k, k + 1)))
13           return false;
14     }
15   }
16   return s.isEmpty();
17 }
```

a.indexOf(b) คืนตำแหน่งใน สตริง a ที่มี b ปรากฏอยู่ ถ้าหาไม่พบคืน –1

พบวงเล็บเปิดให้ push ลงกองซ้อน

พบวงเล็บปิดให้ pop มาตรวจสอบ

ถ้าวงเล็บปิดตัวที่พบเป็นตัวที่ k ใน close วงเล็บเปิดที่คู่กับมันใน open ก็ต้องเป็นตัวที่ k ด้วย ถึงถูกต้อง

รหัสที่ 6-5 การใช้กองซ้อนในการตรวจสอบการใส่วงเล็บ

รหัสที่ 6-5 แสดงเมท็อดการตรวจสอบการใส่วงเล็บ เริ่มด้วยการตั้งสตริงของวงเล็บเปิด และปิดทั้งหลายในตัวแปร `open` และ `close` ตามลำดับ จากนั้นสร้างกองซ้อน `s` ไว้ใช้ในการตรวจสอบ แล้วเข้าวงวนหยิบตัวอักษรออกมาทีละตัว นำไปค้นใน `open` ถ้าพบ แสดงว่าเป็นวงเล็บเปิด ก็ให้เพิ่มลง `s` (บรรทัดที่ 6 และ 7) ถ้าไม่ใช่ก็ลองไปค้นใน `close` ถ้าพบ แสดงว่าเป็นวงเล็บปิด (บรรทัดที่ 10) ก็ให้ตรวจสอบต่อว่า `s` มีข้อมูลหรือไม่ เพราะถ้า `s` ไม่มีข้อมูลให้ลบ แสดงว่าเขียนวงเล็บผิด (บรรทัดที่ 11) หรือว่าถ้า `s` มีข้อมูลแต่เมื่อลบออกมาแล้วเป็นตัวที่ไม่ตรงกับวงเล็บปิดที่ได้รับ ก็ผิดเช่นกัน (บรรทัด 12) การตรวจสอบดำเนินไปกับทุก ๆ ตัวอักษรที่ได้รับ หลังจากตรวจสอบครบหมดแล้ว เหลือการตรวจสอบด่านสุดท้าย ซึ่งจะสรุปได้ว่าถูกต้องก็เมื่อ `s` ไม่มีข้อมูลเหลืออยู่ (บรรทัดที่ 16) การทำงานทั้งหมดนี้วิ่งในวงวนเป็นจำนวนรอบเท่ากับความยาวของข้อความที่ได้รับ ในแต่ละรอบมีการตรวจสอบต่าง ๆ รวมกับการเพิ่มหรือลบที่ใช้เวลาคงตัว ดังนั้นใช้เวลารวมทั้งสิ้นเป็น O(n) โดยที่ n คือความยาวของสตริงที่รับมาตรวจสอบ

กองซ้อนภายในเครื่องเสมือนจาวา

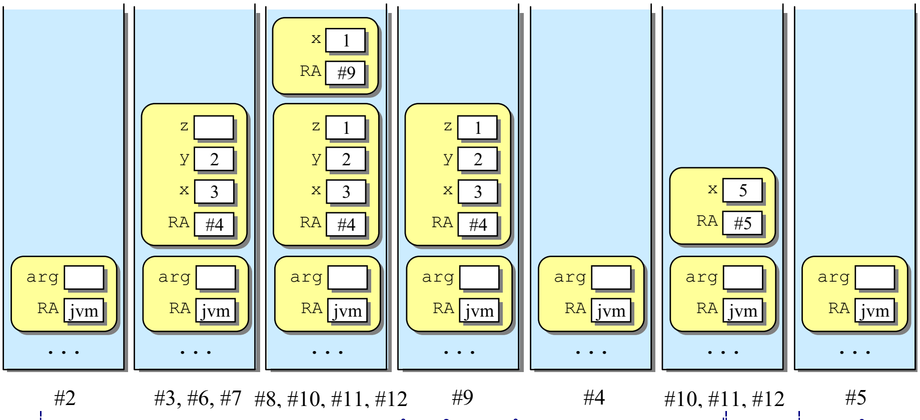
ถ้าผู้อ่านเคยลองเขียนและสั่งทำงานรหัสที่ 6-6 ก็จะพบว่าเกิด `StackOverflowError` ที่เป็นเช่นนี้เพราะการทำงานของเครื่องเสมือนจาวา (เรียกว่า jvm) เกิดปัญหาว่า กองซ้อนภายในระบบมีเนื้อที่ไม่พอ jvm ใช้กองซ้อนเป็นโครงสร้างข้อมูลภายในเพื่อจัดการและจัดเก็บข้อมูลเสริมต่าง ๆ ในการเรียกเมท็อด รหัสที่ 6-6 เรียกเมท็อดตลอดเวลา เกิดการเพิ่มข้อมูลในกองซ้อนระบบจนมีเนื้อที่ไม่พอ

```
public class Jeng3 {
  public static void main(String[] args) {
    main(args);       // เรียกเมท็อดตัวเองอย่างไม่สิ้นสุด
  }
}
```

รหัสที่ 6-6 โปรแกรมสาธิตการเรียกเมท็อดจนเกิด `StackOverflowError`

ทุกครั้งที่มีการเรียกเมท็อด jvm จะเพิ่มสิ่งที่เรียกว่า *กรอบกองซ้อน* (stack frame หรือในบางตำราเรียกว่า activation record) ลงในกองซ้อนของระบบที่ชื่อว่า Java stack ระบบเตรียมกรอบกองซ้อนไว้เป็นที่เก็บค่าของพารามิเตอร์ของเมท็อด, ตัวแปรเฉพาะที่ (local variables) ภายในเมท็อด, เลขที่อยู่กลับ (return address) ซึ่งเป็นตำแหน่งของรหัสที่ต้องทำต่อเมื่อพบคำสั่ง return, และเนื้อที่เสริมอื่น ๆ เพื่อการทำงานของเมท็อด กรอบกองซ้อนของเมท็อดหนึ่งจะอยู่ในกองซ้อนระบบระหว่างที่เมท็อดนั้นยังทำงาน เมื่อทำงานเสร็จกรอบกองซ้อนนี้จะถูกลบออก นี่เองจึงเป็นเหตุผลว่าพารามิเตอร์และตัวแปรเฉพาะที่ของเมท็อดจึงมีให้ใช้เมื่อเมท็อดเริ่มทำงาน และตัวแปรทั้งหมดนี้จะหายไปเมื่อเมท็อดทำงานเสร็จ

รูปที่ 6-2 แสดงขั้นตอนการเปลี่ยนแปลงกรอบกองซ้อนใน jvm เมื่อโปรแกรมในรหัสที่ 6-7 ทำงาน เริ่มจาก jvm เรียก `main` (ที่บรรทัด 2 ซึ่งเขียนแทนด้วย #2 ที่ด้านล่างกองซ้อนในรูป) จะเพิ่มกรอบกองซ้อนซึ่งภายในเก็บพารามิเตอร์ `arg` และเลขที่อยู่กลับ (เขียนแสดงย่อด้วย `RA` ในรูป) เมื่อทำถึงบรรทัดที่ 3 ซึ่งคือ `a(3,2)` กรอบกองซ้อนใหม่ของเมท็อด `a` จะถูกเพิ่ม กรอบนี้จะเป็นที่เก็บของตัวแปร `x,y,z` ของเมท็อด `a` แล้วให้ค่า 3 และ 2 กับ `x` และ `y` ตามลำดับ พร้อมทั้งให้ `RA=#4` เพื่อแสดงว่าเมื่อเมท็อด `a` ทำงานเสร็จ ให้กลับไปทำที่บรรทัดที่ 4 เมื่อสร้างและเพิ่มกรอบกองซ้อนเสร็จ jvm ก็ย้ายไปทำงานที่เมท็อด `a` เมื่อทำถึงบรรทัดที่ 8 ซึ่งคือ `b(z)` ระบบเพิ่มกรอบกองซ้อนของ `b` โดยตัวแปร `x` ของ `b` มีค่าเริ่มต้นตามค่า `z` ของ `a` (ซึ่งมีค่าเท่ากับ 1) พร้อมทั้งให้ `RA=#9` แล้วย้ายไปทำที่ `b` เมื่อ `b` ทำเสร็จที่บรรทัดที่ 12 ระบบจะลบกรอบกองซ้อนนี้ออก แล้วกลับไปทำต่อ ณ บรรทัดที่เก็บใน `RA` ของกรอบกองซ้อนที่เพิ่งถูกลบ (บรรทัดที่ 9) ซึ่งคือคำสั่งจบการทำงานของ `a` ระบบลบกรอบกองซ้อนแล้วกลับไปต่อ ณ ที่ ๆ เก็บใน `RA` (บรรทัดที่ 4 ใน `main`) ซึ่งคือการเรียกเมท็อด `b(5)` ที่มีกระบวนการเรียกเมท็อดในทำนองเดียวกับที่บรรยายมา



รูปที่ 6-2 การ push และ pop กรอบกองซ้อนในกองซ้อนระบบของ jvm เมื่อรหัสที่ 6-7 ทำงาน

```
01 public class StackFrame {
02   public static void main(String[] args) {
03     a(3, 2);
04     b(5);
05   }
06   static void a(int x, int y) {
07     int z = x / y;
08     b(z);
09   }
10   static void b(int x) {
11     ++x;
12   }
13 }
```

ลำดับของหมายเลขบรรทัดที่ทำงานเป็นดังนี้
2, 3, 6, 7, 8, 10, 11, 12, 9, 4, 10, 11, 12, 5

รหัสที่ 6-7 ตัวอย่างการเรียกเมท็อด

ณ ขณะใดขณะหนึ่ง jvm จะทำงานกับข้อมูลภายในกรอบกองซ้อนที่อยู่บนสุดในกองซ้อนเท่านั้น เพราะนี่คือกรอบกองซ้อนของเมท็อดที่ jvm กำลังทำงานอยู่ การจัดหน่วยความจำของพารามิเตอร์และตัวแปรเฉพาะที่จึงทำได้ง่าย อีกทั้งยังช่วยจำเลขที่อยู่กลับเมื่อเมท็อดทำงานเสร็จ นอกจากนี้การใช้กรอบกองซ้อนยังรองรับการทำงานแบบเวียนเกิด (recursive) ได้อีกด้วย เมท็อดแบบเวียนเกิดคือเมท็อดที่มีการเรียกเมท็อดตัวเอง รหัสที่ 6-6 มี `main` ซึ่งเป็นแบบเวียนเกิด แต่เป็นการเขียนที่ไม่ถูกต้องนัก เนื่องจากการเรียกตัวเองกระทำไม่สิ้นสุด การเขียนเมท็อดแบบเวียนเกิดที่ถูกต้องจะต้องมั่นใจว่า มีการเรียกตัวเองไปจนถึงสภาวะซึ่งตัดสินใจไม่เรียกตัวเองต่อ ก็จะเริ่มคืนการทำงานกลับ แต่ละครั้งที่เรียกเมท็อดตัวเอง จะเกิดกรอบกองซ้อนใหม่ในกองซ้อนระบบ นั่นแสดงว่ามีพารามิเตอร์และตัวแปรเฉพาะที่ชุดใหม่ของเมท็อดเกิดขึ้น ถึงแม้ชื่อตัวแปรจะเหมือนกับของเมท็อดที่เรียกครั้งที่แล้ว ๆ แต่ใช้ที่เก็บคนละตำแหน่งกัน

ขอยกตัวอย่างการคำนวณค่ายกกำลัง x^k วิธีคำนวณค่านี้แบบง่าย ๆ กระทำได้โดยการใช้วงวนหมุนคำนวณค่า `c=c*x` เป็นจำนวน `k` ครั้งโดยเริ่มให้ `c=1` ก่อนเข้าวงวน เมื่อหมุนเสร็จจะได้คำตอบเก็บในตัวแปร `c` วิธีนี้ใช้เวลาการทำงานเป็น $\Theta(k)$ ยังมีอีกวิธีหนึ่งซึ่งคำนวณค่านี้ในเวลา $\Theta(\log k)$ โดยอาศัยความสัมพันธ์ข้างล่างนี้ เขียนเป็นเมท็อดดังแสดงในรหัสที่ 6-8

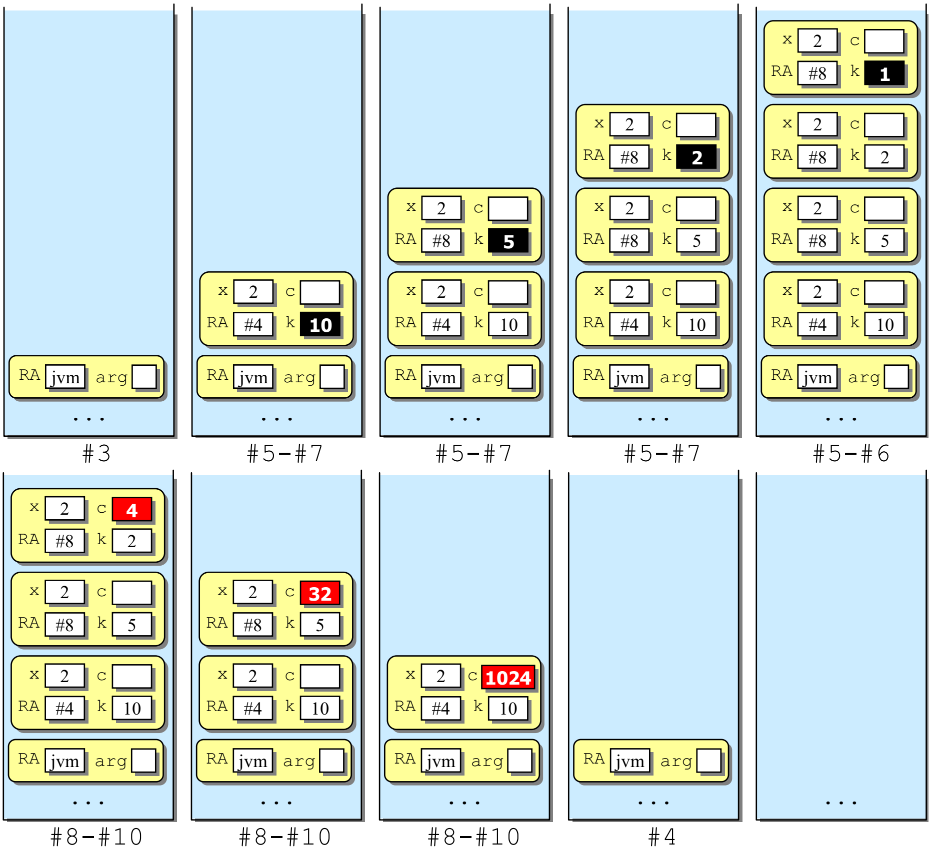
$$x^k = \begin{cases} \left(x^{\lfloor k/2 \rfloor}\right)^2 & \text{ถ้า } k \text{ เป็นจำนวนคู่} \\ x\left(x^{\lfloor k/2 \rfloor}\right)^2 & \text{ถ้า } k \text{ เป็นจำนวนคี่} \end{cases}$$

```
01 public class Recursive {
02   public static void main(String[] args) {
03     System.out.println(pow(2,10));
04   }
05   static long pow(int x, int k) {
06     if (k == 1) return x;
07     long c = pow(x, k / 2);
08     c = c * c;
09     if (k % 2 == 1) c = c * x;
10     return c;
11   }
12 }
```

ถ้า k เป็นจำนวนคี่ คูณด้วย x อีกครั้ง

รหัสที่ 6-8 โปรแกรมการคำนวณ x^k ที่เขียนแบบเวียนเกิด

รูปที่ 6-3 แสดงการเปลี่ยนแปลงกรอบกองซ้อนเมื่อโปรแกรมในรหัสที่ 6-8 ทำงาน เริ่มการเรียก `pow(2,10)` ที่บรรทัดที่ 3 จึงข้ามไปทำที่บรรทัดที่ 5 ทำไปจนถึงบรรทัดที่ 7 มีการเรียกตัวเองด้วย `pow(2,5)` ให้สังเกตว่าการเรียกตัวเองที่บรรทัดที่ 7 นี้ค่าที่ส่งไปให้ `k` ในการเรียกครั้งต่อไปมีค่าเป็นครึ่งหนึ่ง (ปัดเศษทิ้ง) ของ `k` ปัจจุบัน ค่าของ `k` ในการเรียกแต่ละครั้งจึงลดลงเรื่อย ๆ จนมีค่าเป็น 1 ดังนั้นลำดับการเรียกเมท็อด `pow` จึงเป็น `pow(2,10)`, `pow(2,5)`, `pow(2,2)` และ `pow(2,1)` ทำให้เกิดการเพิ่มกรอบกองซ้อนจากการเรียก `pow` เป็นดังแสดงในแถวบนของรูปที่ 6-3 ส่วนแถวล่างของรูปที่ 6-3 แสดงการลบกรอบกองซ้อนเมื่อเริ่มคืนทำงานย้อนกลับไปทีละระดับ การคืนจาก `pow(2,1)` ได้ค่า `c` เป็น 2 กลับมายังบรรทัดที่ 8 ของ `pow(2,2)` คำนวณ `c=c*c` ได้ค่า 4 คืนกลับไปบรรทัดที่ 8 ของ `pow(2,5)` คำนวณ `c=c*c` ได้ค่า 16 แล้วทำ `c=c*x` ได้ 32 (เพราะ `k` เป็นจำนวนคี่) คืนกลับไปบรรทัดที่ 8 ของ `pow(2,10)` คำนวณ `c=c*c` ได้ค่า 1024 คืนกลับไปบรรทัดที่ 4 ของ `main` นำไปให้ `println` แล้วจบการทำงาน



รูปที่ 6-3 การ push และ pop กรอบกองซ้อนในกองซ้อนระบบของ jvm เมื่อรหัสที่ 6-8 ทำงาน

การใช้กองซ้อนจัดเก็บกรอบกองซ้อนทำให้ระบบจัดสรรหน่วยความจำให้กับพารามิเตอร์และตัวแปรของเมท็อดได้ง่าย อีกทั้งเหมาะสำหรับจำตำแหน่งที่จะกลับไปทำต่อ เมื่อเมท็อดทำงานเสร็จ เนื่องจากกองซ้อนมีลักษณะการนำข้อมูลเข้าและออกในลักษณะเข้าหลังออกก่อน จึงตรงกับลักษณะการเรียกเมท็อดที่มีลักษณะถูกเรียกทีหลังคืนการทำงานก่อน

นิพจน์เติมหลัง

นิพจน์คณิตศาสตร์ที่เราเขียนในภาษาโปรแกรม (และเขียนกันทั่ว ๆ ไป) เช่น `a+b*c` เป็นแบบที่เรียกว่า*นิพจน์เติมกลาง* (infix expression) หมายความว่าตัวดำเนินการเขียนอยู่ตรงกลางระหว่างตัวถูกดำเนินการ นิพจน์เติมกลางอาศัยกฎการกำหนดลำดับการทำงานของตัวดำเนินการว่าตัวใดทำก่อนตัวใด ตัวใดทำจากซ้ายไปขวา หรือจากขวามาซ้าย หรือไม่ก็ใช้วงเล็บช่วย เช่น เขียน `a+b*c` หมายถึง `a+(b*c)` ถ้าต้องการทำ + ก่อน * ก็ต้องเขียน `(a+b)*c` มีนิพจน์อีกแบบหนึ่งคือ*นิพจน์เติมหลัง*

(postfix expression) ซึ่งมีลำดับการคำนวณแน่นอนไม่ต้องอาศัยกฎหรือวงเล็บ เช่น `a b + c *` แทน `(a+b)*c` และ `a b c * +` แทน `a+(b*c)`

ในกรณีของภาษาจาวา ตัวแปลภาษาจาวาจะเปลี่ยนนิพจน์ทางคณิตศาสตร์ที่เราเขียนแบบเติมกลางให้กลายเป็นรหัสเครื่องของ jvm ซึ่งเป็นแบบเติมหลัง รหัสที่ 6-9 แสดงตัวอย่างนิพจน์พร้อมหมายเหตุแสดงรหัสเครื่องของจาวาซึ่งได้จากตัวแปลภาษาจาวา ในที่นี้ตัวแปร `a`, `b`, และ `c` ซึ่งเป็นพารามิเตอร์ของเมท็อด `goo` ถูกตัวแปลภาษากำหนดให้เก็บอยู่ในตัวแปรหมายเลข 0, 1, และ 2 ตามลำดับในกรอบกองซ้อน `iload_0`, `iload_1`, และ `iload_2` เป็นรหัสเครื่องแทนการ push ตัวแปรหมายเลข 0, 1, และ 2 ตามลำดับ ในขณะที่ `istore_0` แทนการ pop ออกไปเก็บในตัวแปรหมายเลข 0 ส่วนคำสั่ง `iadd` และ `imul` คือรหัสเครื่องแทนการลบข้อมูลจากกองซ้อนมาสองตัวมาบวกกัน (`iadd`) และคูณกัน (`imul`) ได้ผลลัพธ์แล้วเพิ่มกลับลงกองซ้อน การเพิ่มและลบที่กล่าวถึงนี้กระทำกับกองซ้อนของระบบอีกตัวชื่อว่า operand stack (กองซ้อนนี้ jvm เตรียมที่เก็บให้ภายในกรอบกองซ้อนเมื่อเรียกเมท็อด) รหัสที่ 6-9 แสดงให้เห็นว่านิพจน์ `(a+b)*c` ถูกแปลงเป็น `a b + c *` ในขณะที่ `a+(b*c)` ถูกแปลงเป็น `a b c * +` ด้วยลักษณะการทำงานเช่นนี้เราจึงเรียก jvm ว่าเป็น stack-based machine ซึ่งคือเครื่องที่ทำงานโดยอาศัยกองซ้อนเป็นหลัก

```
static void goo(int a, int b, int c) {
    a = (a + b) * c;
//    iload_0     ;  s.push(a)
//    iload_1     ;  s.push(b)
//    iadd        ;  s.push(s.pop()+s.pop())
//    iload_2     ;  s.push(c)
//    imul        ;  s.push(s.pop()*s.pop())
//    istore_0    ;  a = s.pop()

    a = a + (b * c);
//    iload_0     ;  s.push(a)
//    iload_1     ;  s.push(b)
//    iload_2     ;  s.push(c)
//    imul        ;  s.push(s.pop()*s.pop())
//    iadd        ;  s.push(s.pop()+s.pop())
//    istore_0    ;  a = s.pop()
```

a b + c *

s คือ operand stack ที่ระบบจัดให้

a b c * +

รหัสที่ 6-9 ตัวอย่างรหัสเครื่องที่ได้จากการแปลงนิพจน์เติมกลางเป็นเติมหลัง

วิธีการแปลงนิพจน์เติมกลางให้เป็นแบบเติมหลังวิธีหนึ่ง กระทำด้วยการเขียนนิพจน์เติมกลางที่ใส่วงเล็บกำหนดลำดับการคำนวณให้ครบถ้วน จากนั้นย้ายตัวดำเนินการแต่ละตัวไปไว้หลังวงเล็บปิดที่กำกับตัวดำเนินการนั้น เมื่อย้ายครบก็ลบวงเล็บออกให้หมด จะได้นิพจน์เติมหลัง ตัวอย่างเช่น `a+b*(c-d)/d` เติมวงเล็บให้ครบจะได้ `(a+((b*(c-d))/d))` ย้ายตัวดำเนินการไปไว้ด้านหลังวงเล็บปิดได้ `(a(b(c d)-)*)/)+` ลบวงเล็บออกหมดได้เป็น `a b c d - * / +`

ภาพรวมของวิธีนี้อาจแลดูง่าย แต่จะยุ่งในขั้นตอนการใส่วงเล็บให้ครบ (ขอให้ผู้อ่านลองเขียนโปรแกรมดู)

ยังมีอีกวิธีหนึ่งในการแปลงนิพจน์เติมกลางให้เป็นแบบเติมหลัง ที่อาศัยกองซ้อนช่วยในการแปลง ใช้เวลา $O(n)$ โดยที่ n คือจำนวนพจน์ รหัสที่ 6-10 แสดงเมท็อดที่รับ `infix` ซึ่งคือนิพจน์เติมกลางที่จัดเก็บเป็นรายการของสตริง โดยที่สตริงแต่ละตัวคือตัวดำเนินการและตัวถูกดำเนินการของนิพจน์ เช่น ⟨"a","+","b","*","c"⟩ ได้ผลเป็นรายการของสตริงในลำดับแบบเติมหลังเก็บในตัวแปร `postfix` การทำงานอาศัยกองซ้อนที่มีไว้เก็บตัวดำเนินการ มีวงวนหลัก (วงวน `for` บรรทัดที่ 5 ถึง 16) พิจารณาสตริงแต่ละตัวใน `infix` จากตัวที่ 0 ไปถึงตัวสุดท้าย ถ้าสตริงตัวที่พิจารณา (ตัวแปร `token`) ไม่ใช่ตัวดำเนินการ จะนำไปต่อท้ายรายการ `postfix` ทันที (บรรทัดที่ 8) แต่ถ้าเป็นตัวดำเนินการ จะเพิ่มลงกองซ้อน (บรรทัดที่ 14)โดยก่อนจะเพิ่มอาจมีการลบตัวดำเนินการเดิมในกองซ้อนออกไปต่อท้าย `postfix` (บรรทัดที่ 10 ถึง 13 ซึ่งจะขออธิบายในรายละเอียดในภายหลัง) เมื่อพิจารณาครบทุกตัว จะปิดท้ายด้วยการลบตัวดำเนินการที่เหลือในกองซ้อนออกไปต่อท้าย `postfix` จนหมด (บรรทัดที่ 17) ด้วยการทำงานหลัก ๆ เช่นนี้ลำดับของตัวถูกดำเนินการในนิพจน์เติมกลางและเติมหลังย่อมเหมือนกัน ในขณะที่ลำดับของตัวดำเนินการอาจไม่เหมือน

```
01 public class Expression {
02   public static List infix2Postfix(List infix) {
03     ArrayList postfix = new ArrayList();
04     Stack s = new ArrayStack();
05     for (int i = 0; i < infix.size(); ++i) {
06       String token = (String) infix.get(i);
07       if (!isOperator(token)) {
08         postfix.add(token);
09       } else {
10         while (!s.isEmpty() && priority(token)
11                 <= priority((String) s.peek())) {
12           postfix.add(s.pop());
13         }
14         s.push(token);
15       }
16     }
17     while (!s.isEmpty()) postfix.add(s.pop());
18     return postfix;
19   }
..   ...
```

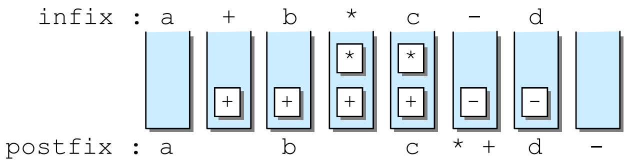
ต่อท้ายผลลัพธ์ถ้าเป็น operand

ก่อนจะ push operator ตัวใหม่ ให้ pop operator ตัวก่อน ๆ ที่ต้องทำก่อน operator ใหม่ ไปต่อท้ายผลลัพธ์

รหัสที่ 6-10 เมท็อดแปลงนิพจน์เติมกลางเป็นเติมหลัง

เหตุผลที่เราต้องใช้กองซ้อนเก็บตัวดำเนินการ ไม่ยอมปล่อยให้ออกไปก่อน ก็เผื่อว่าตัวดำเนินการที่ตามหลังมาทางขวาของนิพจน์ อาจต้องทำก่อน เช่น `a+b*c-d` (ดูรูปที่ 6-4) เมื่อพบ +

ก็เพิ่ม +, พอพบ * ก็เพิ่ม * ทับ + เป็นการแสดงว่า * ต้องถูกลบออกมาก่อน + (นั่นคือ * ต้องทำก่อน +) และเมื่อพบ – ต้องไม่เพิ่ม – ทับคูณ * แต่ต้องลบ * ออก เพราะ * ต้องทำก่อน – และต้องเพิ่ม + ด้วยเพราะ + มาก่อน – (อยู่ทางซ้าย) แล้วจึงค่อยเพิ่ม – ลงกองซ้อน



รูปที่ 6-4 การใช้กองซ้อนช่วยแปลงนิพจน์เติมกลางเป็นเติมหลัง

การเปรียบเทียบความสำคัญของตัวดำเนินการเพื่อให้ทราบลำดับการทำงาน อาศัยสตริง `operators` เก็บตัวดำเนินการทุกตัว และแถวลำดับ `priority` (ดูรหัสที่ 6-11) โดยตัวที่ k ของ `priority` เก็บความสำคัญของตัวดำเนินการตัวที่ k ในสตริง `operators` ค่าความสำคัญนี้เป็นค่าสัมพัทธ์ไว้ใช้ตัดสินว่าตัวดำเนินการสองตัว ตัวใดต้องทำก่อน เช่น + กับ * อยู่ที่ตำแหน่ง 0 และ 2 ใน `operators` มีค่า `priority[0]=3` และ `priority[2]=5` แสดงว่า + ต้องทำทีหลัง * เพราะมีค่าความสำคัญน้อยกว่า เรามี `operatorIndex(x)` ไว้ค้นหาตำแหน่งของตัวดำเนินการ x มี `priority(x)` ไว้หาค่าลำดับการทำงานของตัวดำเนินการ x และมี `isOperator(x)` ไว้ตรวจสอบว่า x คือตัวดำเนินการหรือไม่

```
   static String operators = "+-*/^";   // ^ is power op
   static int priority[] = {3, 3, 5, 5, 7};

   private static boolean isOperator(String x) {
     return operatorIndex(x) >= 0;
   }
   private static int priority(String x) {
     return priority[operatorIndex(x)];
   }
   private static int operatorIndex(String x) {
     return operators.indexOf(x);
   }
   public static void main(String[] args) {
     String[] d = {"a", "+", "b", "*", "c"};
     List e = java.util.Arrays.asList(d);
     System.out.println(Expression.infix2Postfix(e));
   }
}
```

บริการสร้าง list จากแถวลำดับ

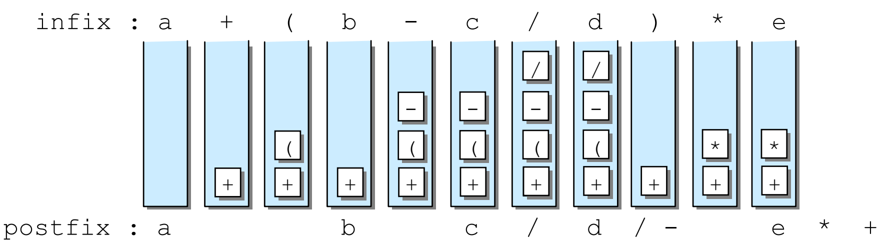
รหัสที่ 6-11 เมท็อดแปลงนิพจน์เติมกลางเป็นเติมหลัง (ต่อจากรหัสที่ 6-10)

ด้วยเมท็อดเสริมต่าง ๆ ที่ได้อธิบายมา ก็คงเข้าใจการทำงานของบรรทัดที่ 10 ถึง 13 ในรหัสที่ 6-10 ซึ่งคือวงวนที่ลบตัวดำเนินการจากกองซ้อนที่สำคัญกว่าซึ่งต้องทำก่อนตัวดำเนินการ `token` ที่เพิ่งนำมาพิจารณา

โปรแกรมแปลงนิพจน์ที่เขียนมายังไม่รองรับเครื่องหมายวงเล็บเปิดและปิด กำหนดให้วงเล็บเป็นตัวดำเนินการ และเปลี่ยนให้ความสำคัญของตัวดำเนินการต่าง ๆ มีสองสภาวะ คือความสำคัญตอนที่ตัวดำเนินการอยู่นอก และตอนอยู่ในกองซ้อน ตอนที่อยู่นอกกองซ้อนก็คือตอนที่หยิบมาจาก `infix` เพื่อพิจารณานั่นเอง การพบเครื่องหมายวงเล็บเปิดใน `infix` เสมือนกับการพบนิพจน์ย่อยที่เราต้องทำนิพจน์ภายในวงเล็บนั้นให้เสร็จ ก่อนกลับมาทำนอกวงเล็บต่อ ดังนั้นตัวดำเนินการต่าง ๆ ในกองซ้อนก็ต้องรอนิพจน์ภายในวงเล็บที่จะตามมาให้ทำเสร็จก่อน เมื่อพบวงเล็บเปิด จึงต้องเพิ่มลงกองซ้อนทันที (เป็นการกดทับตัวดำเนินการอื่น ๆ ทั้งหมดในกองซ้อนให้รอ) ดังนั้นต้องให้ความสำคัญของวงเล็บเปิดตอนอยู่นอกกองซ้อนมีค่าสูงสุด แต่ขณะที่วงเล็บเปิดอยู่ในกองซ้อน กลับต้องมีความสำคัญต่ำสุด เพื่อที่ให้ตัวดำเนินการอื่น ๆ ที่ตามมาใน `infix` กดทับมัน เพราะเราจะลบวงเล็บเปิดก็เฉพาะเมื่อพบวงเล็บปิดเท่านั้น

ส่วนวงเล็บปิดเป็นตัวดำเนินการที่แปลกอีกตัวหนึ่ง คือเมื่อพบ ก็เป็นการสิ้นสุดนิพจน์ย่อย จึงต้องลบตัวดำเนินการทุกตัวในกองซ้อนออกสู่ `postfix` จนพบวงเล็บเปิดจึงหยุดลบ ดังนั้นความสำคัญของวงเล็บปิดตอนอยู่นอกกองซ้อนต้องมีค่าต่ำกว่าตัวดำเนินการอื่น ๆ ในกองซ้อน เพื่อให้ตัวดำเนินการในกองซ้อนถูกลบออก และเพื่อให้หยุดการลบเมื่อพบวงเล็บเปิด ความสำคัญของวงเล็บปิดนอกกองซ้อนต้องมากกว่าวงเล็บเปิดในกองซ้อน

ดูตัวอย่างในรูปที่ 6-5 เมื่อพบ + ก็เพิ่ม, พบ (ก็เพิ่ม, เพราะ (สำคัญมากสุดตอนอยู่นอกกองซ้อน แต่พอเข้าไปในกองซ้อนกลับมีความสำคัญต่ำสุด ทำให้ต่อมาเมื่อพบ – ก็เพิ่ม พบ / ก็เพิ่มอีก เพราะ / สำคัญกว่า – คราวนี้พอพบ) ที่มีความสำคัญต่ำมากแต่สูงกว่า (จึงลุยลบจากกองซ้อนจนกว่าจะพบ (ทำให้เราลบ / และ – ออก แล้วค่อยลบ (แต่ไม่เพิ่ม) จึงถือเป็นกรณีพิเศษ สำหรับพจน์ที่เหลือขอให้ผู้อ่านลองพิจารณาต่อเอง



รูปที่ 6-5 การใช้กองซ้อนช่วยแปลงนิพจน์เติมกลางแบบมีวงเล็บเป็นเติมหลัง

รหัสที่ 6-12 แสดงโปรแกรมแปลงนิพจน์เติมกลางเป็นเติมหลังที่รองรับวงเล็บ การกำหนดความสำคัญแบบสองสภาวะอาศัยแถวลำดับ `inPriority` กับ `outPriority` เก็บค่าความสำคัญของตัวดำเนินการขณะอยู่ในและอยู่นอกกองซ้อน มี `inPriority(x)` กับ `outPriority(x)` ที่คืนความสำคัญของตัวดำเนินการ `x` ให้สังเกตของวงเล็บเปิด `inPriority[5]=0` มีค่าน้อยสุด ในขณะที่ `outPriority[5]=9` มีค่ามากสุด ส่วนของวงเล็บปิดนั้นมี `outPriority[6]=1` ซึ่งมีค่าน้อยกว่าของตัวอื่น ๆ ยกเว้นก็เฉพาะของวงเล็บเปิดในกองซ้อน ทั้งนี้เพื่อให้เมื่อพบวงเล็บปิดใน `infix` การทำงานของวงวนในบรรทัดที่ 11 ถึง 13 จะลบตัวดำเนินการไปเรื่อย ๆ จนพบวงเล็บเปิด ให้สังเกตว่าไม่มี `inPriority[6]` ของวงเล็บปิด เพราะเราจะไม่เพิ่มวงเล็บปิดลงกองซ้อน แต่จะลบวงเล็บเปิดที่คู่กันออกแทน (บรรทัดที่ 14 และ 15)

```
01 public class Expression {
02   public static List infix2Postfix(List infix) {
03     ArrayList postfix = new ArrayList();
04     Stack s = new ArrayStack();
05     for (int i = 0; i < infix.size(); ++i) {
06       String token = (String) infix.get(i);
07       if (!isOperator(token)) {
08         postfix.add(token);
09       } else {
10         int p = outPriority(token);
11         while (!s.isEmpty() && inPriority((String) s.peek())>=p) {
12           postfix.add(s.pop());
13         }
14         if (token.equals(")")) s.pop();
15         else                   s.push(token);
16       }
17     }
18     while (!s.isEmpty()) postfix.add(s.pop());
19     return postfix;
20   }
21
22   static String operators = "+-*/^()";
23   static int inPriority[]  = {3, 3, 5, 5, 7, 0};
24   static int outPriority[] = {3, 3, 5, 5, 7, 9, 1};
25
26   private static int inPriority(String x) {
27     return inPriority[operatorIndex(x)];
28   }
29   private static int outPriority(String x) {
30     return outPriority[operatorIndex(x)];
31   }
32   // isOperator และ operatorIndex เหมือนในรหัสที่ 6-11
..   ...
```

พบวงเล็บปิดไม่ push แต่จะ pop วงเล็บเปิดทิ้งไป

รหัสที่ 6-12 เมท็อดแปลงนิพจน์เติมกลางเป็นเติมหลังที่รองรับวงเล็บ

แบบฝึกหัด

1. จงเขียนคลาส ArrayListStack (ซึ่งคือกองซ้อนที่ใช้ ArrayList ช่วยเก็บข้อมูล) และ ArrayStack (ซึ่งสร้างกองซ้อนด้วยแถวลำดับ) ด้วยตนเอง โดยไม่ดูรายละเอียดในหนังสือ
2. คลังคลาสมาตรฐานของจาวาไม่มีอินเทอร์เฟซ Stack แต่มีคลาสที่ชื่อว่า Stack อยู่ในชุด java.util ซึ่งเป็นคลาสลูกของคลาส Vector ที่มีลักษณะคล้ายกับ ArrayList จงอธิบายบริการต่าง ๆ ของคลาส java.util.Stack
3. ลองสั่งงานเมท็อด main ข้างล่างนี้ แล้วหาคำอธิบายว่าทำไมจึงได้ผลเช่นนั้น

```
public static void main(String[] args) {
  int n = 1000000;
  testStack(new ArrayStack(), n);
  testStack(new ArrayListStack(), n);
}
static void testStack(Stack s, int n) {
   long t = System.nanoTime();
   for (int i=0; i<n; i++) s.push("A");
   System.out.println(s.getClass().getName() + ".push : " +
                        ((System.nanoTime() - t)/1000000.0));
}
```

4. จงเขียนคลาส LinkedListStack เพื่อสร้างกองซ้อนด้วยรายการโยง
5. จงเขียนคลาส StackX ซึ่งสร้างกองซ้อนด้วยแถวลำดับ มีตัวสร้างรับขนาดมากสุดของกองซ้อน มีเมท็อด isFull ไว้ตรวจสอบว่าเต็มหรือไม่ และการ push จะไม่ขยายขนาดของที่เก็บเมื่อเต็ม แต่จะโยน IllegalStateException แทน
6. จงหาข้อมูล (จากห้องสมุดหรือในอินเทอร์เน็ต) ของรหัสเครื่องที่เกี่ยวกับกองซ้อนและการเรียกโปรแกรมย่อยของไมโครโปรเซสเซอร์เพนเทียม (Pentium)
7. จงแสดงการเปลี่ยนแปลงข้อมูลภายในกองซ้อนที่ใช้ ระหว่างการแปลงนิพจน์แบบเติมกลางต่อไปนี้ให้เป็นแบบเติมหลัง `a * b / c - d ^ 3 - e + f / g`
8. จงเขียนเมท็อดเพื่อการหาค่าของนิพจน์เติมหลังที่ประกอบด้วยจำนวนที่เป็นค่าคงตัว กับตัวดำเนินการ + - * / ^ % เช่น 3 2 ^ 6 4 - * มีค่าเท่ากับ 18
9. การยกกำลังในรหัสที่ 6-12 ทำแบบซ้ายไปขวา เช่น 2^3^4 คือ (2^3)^4 แต่โดยทั่วไปเขามักให้การยกกำลังทำจากขวามาซ้าย นั่นคือ 2^3^4 คือ 2^(3^4) จงปรับปรุงให้ทำการยกกำลังแบบขวามาซ้าย (ข้อแนะนำ : การปรับปรุงเพียงแค่ข้อมูลบางตัวในรหัสที่ 6-12 เท่านั้น)

10. จงเขียนเมท็อดรับสตริงของนิพจน์เติมกลางแล้วแปลงเป็นรายการ `infix` ของตัวดำเนินการและตัวถูกดำเนินการที่พร้อมส่งให้เมท็อด `infix2Postfix` ของรหัสที่ 6-12

11. จงวิเคราะห์เวลาการทำงานของการหาค่า x^k ด้วยเมท็อด `pow` ในรหัสที่ 6-8

12. จงเขียนเมท็อดเพื่อตรวจสอบว่านิพจน์เติมหลังที่ได้รับมาถูกต้องหรือไม่

13. จงเขียนเมท็อดที่เปลี่ยนนิพจน์เติมหลังให้เป็นนิพจน์เติมกลาง

14. จงปรับปรุงเมท็อดที่เปลี่ยนนิพจน์เติมกลางให้เป็นนิพจน์เติมหลัง ซึ่งรองรับการใช้เครื่องหมาย – ที่แทนการติดลบ เช่น `1 + -3 * -(3 + 1)`

15. ถ้าสั่งเมท็อด `main` ข้างล่างนี้ทำงาน จะได้ผลอะไร จงเขียนการเปลี่ยนแปลงกรอบกองซ้อนระหว่างการสั่งงานเมท็อด `main` ข้างล่างนี้

```
  public static void main(String[] args) {
    $$$(new int[3], 0);
  }
  static void $$$(int[] x, int k) {
    if (k == x.length) {
      System.out.println(Arrays.toString(x));
    } else {
      x[k] = 0; $$$(x, k + 1);
      x[k] = 1; $$$(x, k + 1);
    }
  }
```

16. ข้างล่างนี้แสดงเมท็อด `contains` ของคลาส `LinkedList` (ที่ได้นำเสนอในบทที่แล้ว) ซึ่งเขียนแบบเวียนเกิด โดยมี `containsR(p,e)` ทำหน้าที่ค้นหาว่ามี `e` อยู่ตั้งแต่ปม `p` เป็นต้นไปหรือไม่ อยากทราบว่าส่วนของโปรแกรมข้างล่างนี้ทำงานถูกต้องหรือไม่ มีข้อเสียอะไร

```
public class LinkedList implements List {
  private LinkedNode header;
  ...
  public boolean contains(Object e) {
    return containsR(header.next, e);
  }
  private boolean containsR(LinkedNode p, Object e) {
    if (p == header) return false;
    if (p.element.equals(e)) return true;
    return containsR(p.next, e);
  }
```

7 แถวคอย

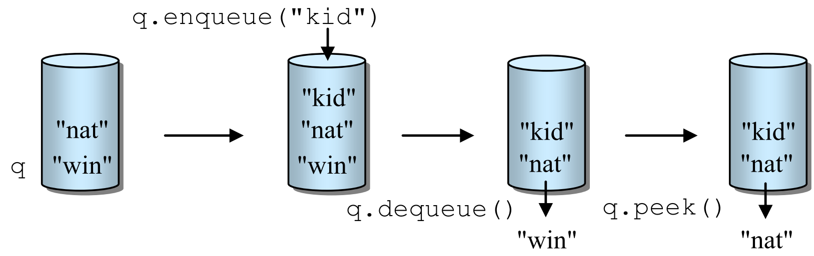
แถวคอย (queue) คือที่เก็บข้อมูลอีกประเภทหนึ่งที่มีบริการเพิ่ม ลบ และเข้าถึงข้อมูลที่ค่อนข้างจำกัดเช่นเดียวกับกองซ้อน แถวคอยเป็นเสมือนรายการที่อนุญาตให้เพิ่มข้อมูลที่ปลายด้านหนึ่งของรายการ ให้ดู และลบข้อมูลที่ปลายอีกด้านหนึ่งของรายการ ข้อมูลภายในแถวคอยเรียงเข้าแถวกันอย่างเป็นระเบียบ ข้อมูลตัวใดเข้าอยู่ในแถวก่อน ก็มีสิทธิ์ออกจากแถวก่อน จึงมีชื่อเรียกแถวคอยกันว่า FIFO ย่อมาจากคำว่า First-In-First-Out แถวคอยเป็นโครงสร้างข้อมูลที่ได้รับการประยุกต์ในการแก้ปัญหาต่าง ๆ มากมาย

ข้อกำหนดของแถวคอย

แถวคอยมีบริการสั้น ๆ ง่าย ๆ บรรยายด้วยอินเทอร์เฟซ `Queue` ดังแสดงในรหัสที่ 7-1 เราเรียกการเพิ่มข้อมูลต่อท้ายแถวคอยว่า `enqueue` และการลบข้อมูลตัวที่หัวแถวออกว่า `dequeue` บางคนมองแถวคอยเสมือนเป็นท่อ ที่รับข้อมูลที่ปลายด้านหนึ่ง เข้าต่อแถวในท่อ การลบข้อมูลกระทำที่ปลายอีกด้าน ดังตัวอย่างการใช้งานแถวคอยในรูปที่ 7-1

```
public interface Queue {
  public boolean isEmpty();         // ถามแถวคอยว่าว่างหรือไม่
  public int size();                // ถามจำนวนข้อมูลในแถวคอย
  public void enqueue(Object e);  // เพิ่ม e เข้าต่อท้ายแถว
  public Object dequeue();          // ลบข้อมูลตัวที่อยู่หัวแถว
  public Object peek();             // ขอดูข้อมูลตัวที่อยู่หัวแถว
}
```

รหัสที่ 7-1 อินเทอร์เฟซ `Queue`



รูปที่ 7-1 การ enqueue dequeue และ peek ข้อมูล

รหัสที่ 7-2 แสดงการใช้งานแถวคอย ตามตัวอย่างในรูปที่ 7-1 เริ่มด้วยการสร้างแถวคอยในบรรทัดที่ 3 แล้ว enqueue ข้อมูลเข้าไปสามตัว ตามด้วย dequeue และ peek มีการถามขนาดของแถวคอย เพื่อแสดงให้เห็นว่า peek เป็นแค่การขอดูข้อมูลที่อยู่บนแถวคอย ไม่ได้ลบออกจากแถวคอย

```
01 public class TestQueue {
02   public static void main(String[] args) {
03     Queue q = new ArrayQueue();
04     q.enqueue("win");
05     q.enqueue("nat");
06     q.enqueue("kid");
07     System.out.println(q.size());
08     System.out.println(q.dequeue());
09     System.out.println(q.size());
10     System.out.println(q.peek());
11     System.out.println(q.size());
12   }
13 }
```

```
3
"win"
2
"nat"
2
```

รหัสที่ 7-2 ตัวอย่างการทำงานของแถวคอยในรูปที่ 7-1

การสร้างแถวคอยด้วยรายการ

เราสามารถสร้างแถวคอยได้ง่าย ๆ ด้วยการนำรายการมาเก็บซ่อนไว้ในคลาส แล้วให้รายการนี้ทำหน้าที่จัดเก็บและจัดการข้อมูลแทน รหัสที่ 7-3 แสดงตัวอย่างการสร้างด้วยวิธีดังกล่าว มีตัวแปร list เก็บอ็อบเจกต์ของ ArrayList บริการ isEmpty และ size ส่งต่อไปถามที่ตัว list ที่มีหน้าที่เก็บข้อมูล, enqueue ก็คือการเพิ่มข้อมูลต่อท้ายรายการ, peek ก็คือการขอข้อมูลที่ตำแหน่งแรกของรายการ ส่วน dequeue ก็คือการลบข้อมูลตัวแรกของรายการ ให้สังเกตว่า เราเลือกให้ enqueue เพิ่มข้อมูลต่อท้ายรายการ จึงใช้เวลา $\Theta(1)$ แต่ทำให้การ dequeue ต้องลบข้อมูลที่ต้นรายการ จึงใช้เวลา $\Theta(n)$ แต่ในทางกลับกัน ถ้าเราให้ enqueue เพิ่มที่ต้นรายการ จะใช้เวลา $\Theta(n)$ ส่งผลให้ dequeue ต้องทำที่ท้ายรายการ ใช้เวลา $\Theta(1)$

```
public class ArrayListQueue implements Queue {
  private ArrayList list = new ArrayList();

  public boolean isEmpty() {
    return list.isEmpty();
  }
  public int size() {
    return list.size();
  }
  public void enqueue(Object e) {
    list.add(list.size(),e);
  }
  public Object peek() {
    if (isEmpty()) throw new IllegalStateException();
    return list.get(0);
  }
  public Object dequeue() {
    Object e = peek();
    list.remove(0);
    return e;
  }
}
```

แถวคอยนี้ใช้ ArrayList เป็นที่เก็บข้อมูล

ทุกคำสั่งของแถวคอย สั่งงานต่อให้ไปใช้บริการของ ArrayList

ต่อท้าย ArrayList เร็วสุด

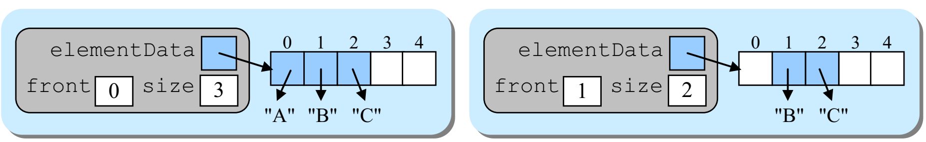
ไม่มีข้อมูลให้ดู เกิด exception

ลบตัวแรกของ ArrayList ย่อมช้าสุด

รหัสที่ 7-3 การสร้างแถวคอยด้วยการใช้รายการจัดเก็บและจัดการข้อมูลแทน

การสร้างแถวคอยด้วยแถวลำดับวงวน

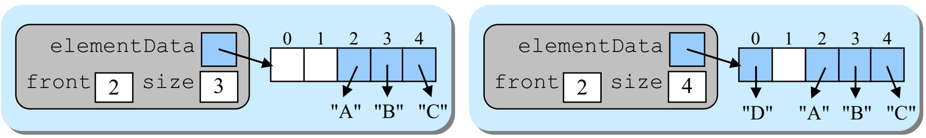
การสร้างแถวคอยด้วยรายการนั้นง่ายดี แต่ประสิทธิภาพไม่ค่อยดี เราน่าจะทำได้เหมือนกับกรณีของกองซ้อนที่ทั้งการเพิ่มและลบข้อมูลใช้เวลาคงตัวหมด การเก็บข้อมูลของ ArrayList บังคับให้ข้อมูลเริ่มที่ช่องหมายเลข 0 ของแถวลำดับ ทำให้ต้องย้ายข้อมูลเมื่อต้องการลบข้อมูลที่ช่องหมายเลข 0 หากเราเพิ่มตัวแปร front ไว้เก็บตำแหน่งเริ่มต้นของชุดข้อมูล จะทำให้ลบตัวแรกของชุดข้อมูลได้ง่าย ๆ ด้วยการเพิ่มค่าของ front ไปอีก 1 และลดจำนวนข้อมูลก็เสร็จ ดูตัวอย่างในรูปที่ 7-2 รูปทางซ้ายเป็นสภาพเมื่อแถวคอยมีข้อมูลสามตัว เก็บเริ่มที่ช่องที่ 0 ไป 3 ช่อง นั่นคือช่องที่ 0, 1, และ 2 (front=0, size=3) การ dequeue คือการลบตัวแรกของแถวคอยทิ้ง ก็ทำได้ง่าย ๆ ด้วย front++ และ size-- จะได้ดังรูปทางขวา ซึ่งแสดงสภาพเมื่อแถวคอยมีข้อมูลสองตัว เริ่มเก็บที่ช่องที่ 1 ไป 2 ช่อง



รูปที่ 7-2 การสร้างแถวคอยด้วยแถวลำดับ front เก็บตำแหน่งแรกของชุดข้อมูลในแถว

สำหรับการเพิ่มข้อมูลใหม่ต่อท้ายแถว จากค่าของ front และ size เราสามารถสรุปได้ว่า front+size คือหมายเลขช่องถัดไปที่สามารถนำข้อมูลใหม่ไปเพิ่มได้ แต่เป็นวิธีที่ไม่ค่อยดีนัก เนื่องจากเราไม่สามารถนำช่องว่างด้านซ้ายของ front กลับมาใช้ใหม่ได้ ตัวอย่างเช่น รูปซ้ายของรูปที่ 7-3 มี front=2 และ size=3 ช่องที่ 2, 3, และ 4 เก็บข้อมูลของแถวคอย ในขณะที่ช่องที่ 0 และ 1 ว่าง แต่เรากลับไม่สามารถนำมาใช้เก็บข้อมูลได้ เพราะการเพิ่มข้อมูลต่อท้ายจะถูกนำไปเพิ่มที่ช่อง front+size = 5 ซึ่งล้นแถวลำดับ (ซึ่งทำให้ต้องขยายขนาดของแถว) เราสามารถนำช่องทางซ้ายที่ว่างของ front มาใช้ใหม่ได้ด้วยการมองแถวลำดับที่จองไว้เป็นแบบวงวน หมายความว่า ช่องถัดจากช่องสุดท้ายของแถวก็คือช่องที่ 0 ของแถว ดังนั้นการเพิ่มข้อมูลใหม่ต่อท้ายรูปทางซ้ายของรูปที่ 7-3 จะได้ดังรูปขวา ได้เป็นสูตรคำนวณช่องถัดจากตัวท้ายในแถวคอยคือ

```
(front + size) % elementData.length
```



รูปที่ 7-3 การสร้างแถวคอยด้วยแถวลำดับแบบวงวน

เมื่อมองแถวลำดับในลักษณะวงวนเช่นนี้ การลบข้อมูลจากหัวแถวที่ตำแหน่ง front ด้วย front++ ก็ต้องเปลี่ยนให้เป็น front=(front+1)%elementData.length เพื่อให้การลบตัวท้ายสุดของแถวลำดับ เปลี่ยนค่าของ front เป็น 0

สรุปได้เป็นคลาส ArrayQueue ในรหัสที่ 7-4 มีเมท็อด inc(i) ในบรรทัดที่ 36-38 ให้บริการคำนวณตำแหน่งถัดจาก i ไปหนึ่งตำแหน่งในแถวลำดับแบบวงวน การ dequeue อาศัยการ peek แล้วตามด้วย front=inc(front) ซึ่งแทนการเลื่อนค่าของ front ไปหนึ่งตำแหน่งแบบวงวน ส่วนการ enqueue คำนวณตำแหน่งถัดจากตัวท้ายในแถวคอย แล้วนำข้อมูลใหม่ใส่ในช่องนั้น (บรรทัดที่ 32 และ 33) ถ้าแถวลำดับเต็ม (ด้วยการทดสอบจาก size และขนาดของแถวลำดับในบรรทัดที่ 25) ก็ให้ขยายขนาดและหยิบข้อมูลจากหัวแถวไปท้ายแถวไปใส่ในแถวลำดับตัวใหม่เริ่มจากช่องที่ 0 เป็นต้นไป

```
public class ArrayQueue implements Queue{
  private Object[] elementData;
  private int front, size;

  public ArrayQueue() {
    elementData = new Object[1];
  }
  public int size() {
    return size;
  }
  public boolean isEmpty() {
    return size == 0;
  }
  public Object peek() {
    if (isEmpty()) throw new IllegalStateException();
    return elementData[front];
  }
  public Object dequeue() {
    Object element = peek();
    front = inc(front);
    --size;
    return element;
  }
  public void enqueue(Object element) {
    if (size == elementData.length) {
      Object[] arr = new Object[2*elementData.length];
      for (int i=0, j=front; i<size; i++, j=inc(j))
        arr[i] = elementData[j];
      front = 0;
      elementData = arr;
    }
    int b = (front + size) % elementData.length;
    elementData[b] = element;
    ++size;
  }
  private int inc(int i) {
    return (i + 1) % elementData.length;
  }
}
```

ตัวหัวแถว ชี้โดย front

ลบตัวหัวแถว โดยการเลื่อน front ไปหนึ่งตำแหน่ง

ขยายขนาดได้ ถ้าเต็ม

คำนวณตำแหน่งของช่องที่ให้ต่อท้าย จากค่าของ front และ size

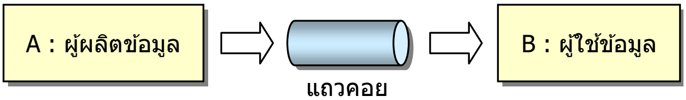
เพิ่มค่า i แบบวนกลับ 0 ได้

รหัสที่ 7-4 การสร้างแถวคอยด้วยแถวลำดับแบบวงวน

ทุก ๆ เมท็อดของ ArrayQueue ซึ่งสร้างด้วยแถวลำดับแบบวงวนใช้เวลาเป็น $\Theta(1)$ ทั้งสิ้น (ยกเว้นก็เฉพาะกรณี enqueue แล้วเกิดการขยายขนาดเมื่อเต็ม) จะเห็นได้ว่า เป็นวิธีที่จัดเก็บและจัดการข้อมูลอย่างง่าย และได้ประสิทธิภาพการทำงานที่ดี

ตัวอย่างการใช้งานแถวคอย

แนวคิดของแถวคอยมีให้เห็นทั่วไป ตามสถานบริการต่าง ๆ ที่มีผู้คนเข้าแถวรอรับบริการ เราใช้แถวคอยเพื่อจัดระเบียบ และเหตุที่มีแถวคอยให้พักรอก็เพราะไม่สามารถบริการได้ทันกับผู้มารับบริการ ในมุมมองของการจัดการข้อมูล หาก A ต้องการส่งข้อมูลให้ B ไปประมวลผลเป็นระยะ ๆ โดยที่อัตราการส่งข้อมูลจาก A เท่ากับอัตราการประมวลผลของ B เข้าจังหวะกันพอดี ก็สามารถให้ A ส่งข้อมูลไปยัง B ได้โดยตรง A ไม่ต้องรอ B (เพราะ B ทำเสร็จทัน) และ B ก็ไม่ต้องรอ A (เพราะ A ผลิตข้อมูลให้พอดี) แต่ถ้าบางช่วงเวลา A ผลิตข้อมูลถี่มาก เร็วเกินกว่าช่วงเวลาที่ B ทำงาน ก็ต้องใช้แถวคอยคั่นกลางระหว่าง A กับ B



รูปที่ 7-4 การใช้แถวคอยเป็นที่พักข้อมูลระหว่างผู้ผลิตและผู้ใช้ที่ทำงานไม่ประสานกัน

รูปที่ 7-4 แสดงการใช้แถวคอยเป็น "ที่พัก" (buffer) ของข้อมูลที่ได้จากผู้ผลิต เพื่อรอให้ผู้ใช้รับไปประมวลผล เนื่องจากทั้งสองฝ่ายทำงานด้วยจังหวะที่ไม่ตรงกัน ตัวอย่างเช่น

- ข้อมูลที่เรากดทางแป้นพิมพ์ รวมถึงการเลื่อนหรือกดปุ่มเมาส์ จะถูกเก็บเข้าแถวคอยของระบบเพื่อรอให้โปรแกรมระบบนำไปประมวลผล เราอาจรู้สึกว่า การกดแป้นพิมพ์หรือเลื่อนเมาส์น่าจะทำได้ช้ากว่าการประมวลผลของเครื่องคอมพิวเตอร์ แต่ในบางช่วงผู้ใช้อาจกดแป้นหรือเลื่อนเมาส์ในช่วงที่ระบบกำลังประมวลผลอย่างอื่น เช่น กำลังประมวลผลภาพที่แสดงบนจอ การใช้แถวคอยจึงทำให้ระบบไม่พลาดการประมวลผลเหตุการณ์ที่ผู้ใช้โต้ตอบกับระบบ
- เมื่อผู้ใช้สั่งพิมพ์งานจากเครื่องคอมพิวเตอร์ของตนสู่เครื่องพิมพ์กลางของเครือข่าย งานพิมพ์ที่สั่งไปจะถูกเก็บเข้าแถวคอยของระบบจัดการพิมพ์ ซึ่งจะป้อนงานพิมพ์ให้กับเครื่องพิมพ์ตามลำดับงานที่ได้รับ

นอกจากการใช้แถวคอยเป็นที่พักข้อมูลแล้ว ยังมีการใช้แถวคอยในงานอีกหลากหลาย หัวข้อนี้นำเสนอตัวอย่างแถวคอยที่ใช้เป็นที่พักแบบพิเศษเรียกว่า แถวคอยให้หยุดรอ (blocking queue) ตามด้วยการใช้แถวคอยเป็นที่เก็บข้อมูลระหว่างการเรียงลำดับข้อมูลแบบฐาน (radix sort) และการใช้แถวคอยเก็บข้อมูลเสริมระหว่างการค้นตามแนวกว้าง (breadth-first search)

แถวคอยให้หยุดรอ [1]

ในกรณีที่เราใช้แถวคอยเป็นที่พักของข้อมูลระหว่างผู้ผลิตข้อมูลกับผู้ใช้ข้อมูล เมื่อใดที่ผู้ผลิตต้องการเพิ่มข้อมูลใหม่ ถ้าแถวคอยที่ใช้สามารถขยายขนาดได้อัตโนมัติ ก็เพียงแต่เรียกบริการ enqueue แต่สำหรับผู้ใช้ข้อมูล เมื่อต้องการเรียกใช้บริการ dequeue ผู้ใช้ต้องตรวจสอบด้วยว่า แถวคอยมีข้อมูลให้ลบออกหรือไม่ ถ้าไม่มีก็ต้องรอ วิธีรอทำได้หลายวิธี วิธีหนึ่งคือใช้วงวนตรวจสอบแถวคอยจนกว่าแถวคอยจะไม่ว่าง แล้วจึง dequeue ข้อมูลไปประมวลได้ แต่วิธีนี้มีข้อเสียตรงที่วงวนการตรวจสอบนั้นสิ้นเปลืองการทำงานของหน่วยประมวลผลอย่างมาก (ต้องอย่าลืมว่า เครื่องคอมพิวเตอร์ไม่ได้มีโปรแกรมของเราทำงานเพียงโปรแกรมเดียว แต่มีหลาย ๆ โปรแกรมแบ่งหน่วยประมวลผลกันทำงานจนเรารู้สึกว่า โปรแกรมเหล่านี้ทำงานไปพร้อม ๆ กัน ถ้าโปรแกรมหนึ่งทำงานติดในวงวน ก็พลอยทำให้โปรแกรมอื่นทำงานช้าลง) นอกจากนี้ยังมีปัญหาอย่างอื่นที่ต้องคำนึงถึง คือในกรณีที่ระบบมีผู้ผลิตข้อมูลหลายหน่วย หรือผู้ใช้ข้อมูลหลายหน่วย (แต่ละหน่วยที่กล่าวถึงนี้คือหน่วยทำงานที่เรียกว่า thread) แย่งกัน enqueue หรือ dequeue จากแถวคอยเดียวกัน ซึ่งหากทำงานไปแค่ครึ่ง ๆ กลาง ๆ เมท็อดที่กำลังเปลี่ยนข้อมูลภายในแถวคอย แล้ว jvm ขัดจังหวะ สลับให้หน่วยทำงานอื่นทำบ้าง สลับกันทำไปทำมา ก็ย่อมเกิดปัญหาได้ว่า ข้อมูลที่เพิ่มอาจหาย หรือข้อมูลอาจถูกลบออกไปใช้ซ้ำกัน

```
public class BlockingQueue extends ArrayQueue{
  public synchronized Object dequeue() {
    while (isEmpty());
    return super.dequeue();
  }
  public synchronized void enqueue(Object element) {
    super.enqueue(element);
  }
}
```

วงวนหมุนรอข้อมูล

รหัสที่ 7-5 BlockingQueue ที่ทำงานค้างใน dequeue ถ้าแถวคอยว่าง

รหัสที่ 7-5 แสดงคลาส BlockingQueue ที่สร้างจาก ArrayQueue โดยที่เมท็อด dequeue มีวงวนที่รอจนกว่าแถวคอยมีข้อมูล จึงจะลบข้อมูลคืนกลับให้ผู้เรียก ให้สังเกตว่า เมท็อด enqueue และ dequeue มีคำว่า synchronized กำกับหัวเมท็อด ก็เพื่อป้องกันไม่ให้หน่วยทำงานหนึ่งเข้าทำ enqueue หรือ dequeue ของแถวคอยหนึ่ง เมื่อมีอีกหน่วยทำงานหนึ่งกำลังทำงานใน enqueue หรือ dequeue ของแถวคอยนั้น จึงประกันได้ว่า จะไม่เกิดการเปลี่ยนแปลงแถวคอยเดียวกันจากหน่วยทำงานหลาย ๆ ตัวพร้อม ๆ กัน แต่รหัสที่ 7-5 ยังใช้ไม่ได้ เพราะถ้าแถวคอยว่าง และหน่วยทำงาน A เรียก dequeue จะเกิดการวนรอจนกว่าแถวคอยจะมีข้อมูล แต่

[1] หัวข้อนี้เกี่ยวกับ multithread ซึ่งใช้คำสั่ง synchronized, wait และ notify อันเป็นคุณสมบัติเฉพาะของจาวา

เนื่องจากระบบไม่อนุญาตให้หน่วยทำงานอื่นใดทำ enqueue เพราะ A ยังทำ dequeue ไม่เสร็จ (มัวหมุนรออยู่) จึงเกิดปัญหาการทำงานค้างและทำต่อไปไม่ได้

รหัสที่ 7-6 แสดงวิธีแก้ปัญหาข้างต้น โดยอาศัยการรอด้วยคำสั่ง this.wait (บรรทัดที่ 4) ซึ่งทำให้หน่วยทำงานหยุดรอจนกว่าจะมีหน่วยทำงานอื่นเรียก notifyAll กับอ็อบเจกต์ this (ที่ถูกเรียกให้ wait ก่อนหน้านี้) โดย wait มีคุณสมบัติพิเศษตรงที่จะปลดการป้องกันการเรียกเมท็อดที่มี synchronized ออกชั่วคราว ทำให้ระบบอนุญาตให้หน่วยทำงานอื่นเรียกเมท็อดที่มี synchronized ได้ ในที่นี้ก็คือการอนุญาตให้หน่วยทำงานอื่นเรียก enqueue เพื่อเพิ่มข้อมูล โดยหลังเพิ่มเสร็จก็จะเรียก this.notifyAll (บรรทัดที่ 12) เมื่อ enqueue ทำงานเสร็จ หน่วยทำงานที่รอด้วย wait ก็จะทำงานต่อได้

```
01 public class BlockingQueue extends ArrayQueue {
02   public synchronized Object dequeue() {
03     try {
04       while (isEmpty()) this.wait();
05       return super.dequeue();
06     } catch (InterruptedException e) {
07       return null;
08     }
09   }
10   public synchronized void enqueue(Object element) {
11     super.enqueue(element);
12     this.notifyAll();
13   }
14 }
```

รอให้ thread อื่น notify

ถ้า thread ที่ wait ถูกขัดจังหวะ จะเกิด InterruptedException

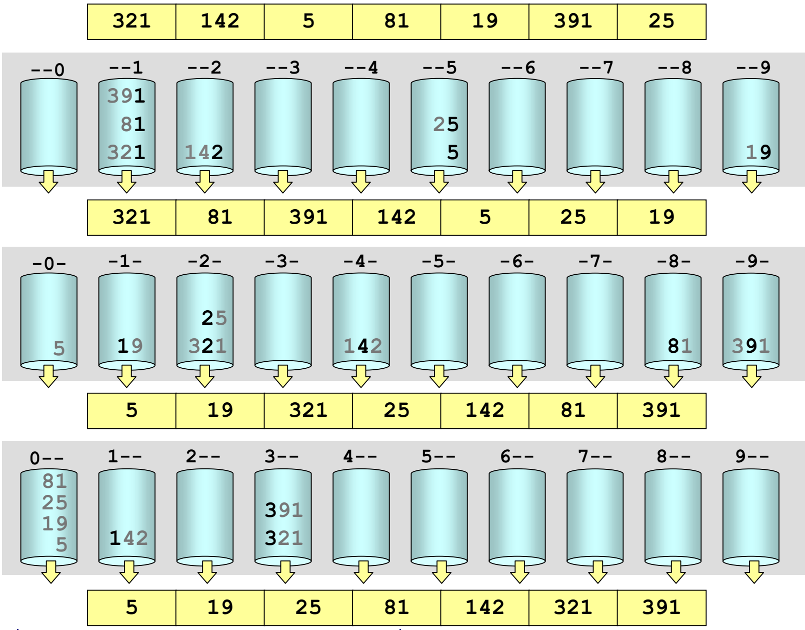
notify thread ที่เรียก this.wait()

รหัสที่ 7-6 BlockingQueue ที่รอโดยใช้ wait กับ notify

ต้องขอบอกตรงนี้ว่า คลาสต่าง ๆ ที่เราได้เขียนโครงสร้างข้อมูลหลากหลายชนิดกันมาตั้งแต่ต้น ล้วนมีปัญหาเมื่อมีหน่วยทำงานหลายตัวใช้งานกับที่เก็บข้อมูลตัวเดียวกันพร้อม ๆ กันทั้งสิ้น แต่โดยทั่วไปการใช้งานมักอยู่ในระบบแบบหน่วยทำงานเดี่ยว ซึ่งต่างกับแถวคอยให้หยุดรอ ที่ได้รับการประยุกต์ใช้เป็นที่พักข้อมูล ซึ่งส่งจากผู้ผลิตไปยังผู้ใช้ข้อมูลในระบบหลายหน่วยทำงานอยู่เป็นประจำ

การเรียงลำดับข้อมูลแบบฐาน

การเรียงลำดับข้อมูลคือการจัดลำดับข้อมูลในรายการใหม่ (ซึ่งโดยทั่วไปเก็บไว้ในแถวลำดับ) ให้ข้อมูลภายในเรียงลำดับจากน้อยไปมาก เช่น เมื่อนำข้อมูล ⟨ 10, 20, 5, 8, 9, 4 ⟩ ไปเรียงลำดับแล้วจะได้ ⟨ 4, 5, 8, 9, 10, 20 ⟩ การเรียงลำดับมีมากมายหลากหลายวิธี ที่เราจะมานำเสนอในหัวข้อย่อยนี้คือ*การเรียงลำดับแบบฐาน* (radix sort) ที่อาศัยการมองข้อมูลเป็นเลขฐาน แล้วแยกแต่ละเลขโดดในข้อมูลออกมาพิจารณา โดยใช้แถวคอยช่วยจัดเก็บข้อมูลระหว่างการเรียงลำดับ



รูปที่ 7-5 ตัวอย่างการเรียงลำดับแบบฐานที่ใช้แถวคอยจัดเก็บข้อมูลระหว่างการทำงาน

ขออธิบายการทำงานของการเรียงลำดับแบบฐานด้วยการใช้ตัวอย่างประกอบ (ดูรูปที่ 7-5) เนื่องจากข้อมูลขาเข้าเป็นจำนวนเต็มฐานสิบ ก็เริ่มด้วยการสร้างแถวคอย 10 แถว ให้หมายเลข 0 ถึง 9 กำกับแถวคอยแต่ละแถว การทำงานจะเป็นวงวนพิจารณาเลขโดดแต่ละหลักของข้อมูล รอบละหลัก เริ่มจากหลักที่มีนัยสำคัญต่ำสุด (ขอเรียกว่าหลักที่ 0) ไปจนถึงหลักที่มีนัยสำคัญสูงสุด รอบแรกเริ่มหลักที่ 0 นำข้อมูลที่หลักที่ 0 มีค่าเป็น *c* ไปใส่ในแถวคอยที่ *c* เช่น พิจารณาแถวบนสุดของรูปที่ 7-5 จะเห็นแถวคอยที่ 1 มีข้อมูลคือ 321, 81, และ 391 เพราะหลักที่ 0 ของข้อมูลสามตัวนี้มีค่าเป็น 1 เมื่อใส่ครบแล้ว ก็ลบข้อมูลออกจากแถวคอยไล่ตั้งแต่แถวที่ 0 ไปจนถึงแถวที่ 9 โดยนำข้อมูลที่ลบออกใส่กลับในแถวลำดับขาเข้า เรียงจากซ้ายไปขวา แล้วเริ่มทำรอบถัดไป เพื่อพิจารณาหลักถัดไป คือหลักที่ 1 ก็ทำเหมือนเดิมอีกคือ นำข้อมูลที่หลักที่ 1 มีค่าเป็น *c* ไปใส่ในแถวคอยที่ *c* เช่น พิจารณาแถวกลางของรูปที่ 7-5 จะเห็นแถวคอยที่ 2 มีข้อมูล 321 และ 25 เพราะหลักที่ 1 ของข้อมูลทั้งสองมีค่าเป็น 2 ใส่ครบทุกตัวแล้ว ก็ลบข้อมูลจากแถวคอยทุกตัวเพื่อใส่กลับในแถวลำดับเดิม แล้วเริ่มรอบที่ 3 เพื่อนำข้อมูลที่หลักที่ 2 มีค่าเป็น *c* ไปใส่ในแถวคอยที่ *c* เช่น พิจารณาแถวล่างของรูปที่ 7-5 จะเห็นแถวคอยที่ 0 มีข้อมูล 5, 19, 25, และ 81 เพราะหลักที่ 2 ของข้อมูลทั้งหมดมีค่าเป็น 0 จากตัวอย่างข้อมูลมีเพียง 3 หลักจึงทำ 3 รอบ จะได้ข้อมูลเก็บกลับในแถวลำดับเรียงจากน้อยไปมาก

รหัสที่ 7-7 แสดงโปรแกรมการเรียงลำดับแบบฐานสำหรับจำนวนเต็มฐานสิบ โดยรับ `data` เป็นแถวลำดับของอ็อบเจกต์แบบ `Integer` และ `d` เป็นจำนวนหลักมากสุดของข้อมูล เริ่มด้วยการ

จองแถวลำดับ 10 ช่องไว้เก็บแถวคอย 10 แถว (เพราะข้อมูลเราเป็นฐานสิบ) ตามด้วยวงวนสร้างแถวคอย 10 แถวเก็บในแต่ละช่อง (บรรทัดที่ 3 ถึง 5) จากนั้นเข้าวงวนหลัก (บรรทัดที่ 6) เพื่อพิจารณาแต่ละหลักของข้อมูลเริ่มจากหลักที่ 0 ถึง `d-1` วงวนในบรรทัดที่ 7 ถึง 8 นำข้อมูลแต่ละตัวในแถวลำดับ ที่หลักที่ `k` มีค่าเป็น c ไปใส่ในแถวคอยที่ c โดยใช้ `getDigit` ดึงหลักที่ `k` ของข้อมูลออกมาใช้ ตามด้วยวงวนในบรรทัดที่ 9 ถึง 12 ลบข้อมูลออกจากแถวคอยไล่ตั้งแต่แถวที่ 0 จนถึงที่ 9 ใส่กลับในแถวลำดับขาเข้าเรียงจากซ้ายไปขวา

```
01 public class ArrayUtil {
02   public static void radixSort(Integer[] data, int d) {
03     Queue[] q = new ArrayQueue[10];
04     for (int i = 0; i < q.length; i++)
05       q[i] = new ArrayQueue();
06     for (int k = 0; k < d; k++) {
07       for (int i = 0; i < data.length; i++)
08         q[getDigit(data[i],k)].enqueue(data[i]);
09       for (int i = 0, j = 0; i < q.length; i++) {
10         while(!q[i].isEmpty())
11           data[j++] = (Integer) q[i].dequeue();
12       }
13     }
14   }
15   private static int getDigit(Integer v, int k) {
16     int n = v.intValue();
17     for (int i = 0; i < k; i++) n /= 10;
18     return n % 10;
19   }
```

ที่ต้องเป็น Integer เพราะแถวคอยที่เราออกแบบมาเก็บอ็อบเจกต์

คืนเลขโดดหลักที่ k ของ v k = 0 คือหลักหน่วย

รหัสที่ 7-7 `Radix sort` สำหรับการเรียงลำดับจำนวนเต็มฐานสิบ

```
20   public static void radixSort(String[] data, int d) {
21     Queue[] q = new ArrayQueue[27];
22     for (int i = 0; i < q.length; i++)
23       q[i] = new ArrayQueue();
24     for (int k = d - 1; k >= 0; k--) {
25       for (int i = 0; i < data.length; i++)
26         q[getDigit(data[i], k)].enqueue(data[i]);
27       for (int i = 0, j = 0; i < q.length; i++) {
28         while (!q[i].isEmpty())
29           data[j++] = (String) q[i].dequeue();
30       }
31     }
32   }
33   private static int getDigit(String v, int k) {
34     if (k >= v.length()) return 0;
35     return v.charAt(k) - 'A' + 1;
36   }
```

ต้องการแถวคอย 27 ตัว

ตัวที่ 0 ของสตริงคือตัวซ้ายสุด

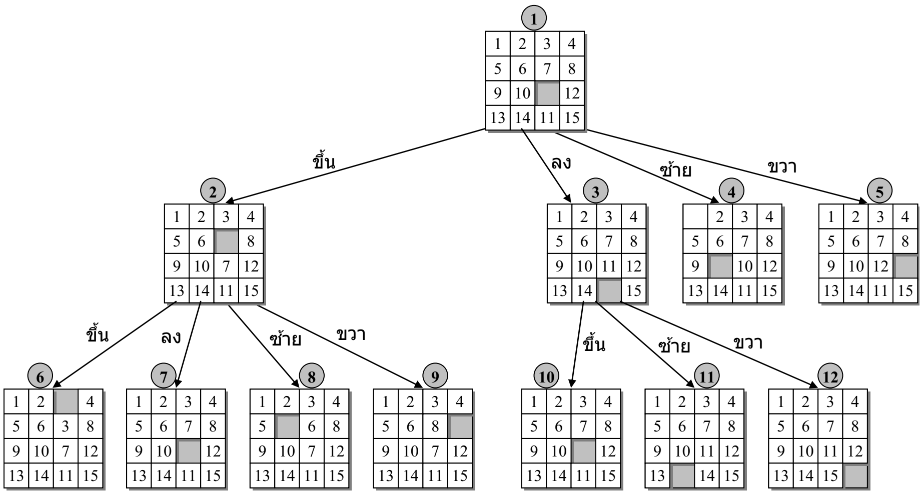
A คือ 1, B คือ 2, ... Z คือ 26 ว่าง ๆ คือ 0

รหัสที่ 7-8 `Radix sort` สำหรับการเรียงลำดับสตริงของตัวอักษรอังกฤษตัวใหญ่

รหัสที่ 7-8 แสดงโปรแกรมการเรียงลำดับแบบฐานสำหรับข้อมูลที่เป็นสตริงของตัวอักษรภาษาอังกฤษตัวใหญ่ ในกรณีนี้เราต้องมองข้อมูลเป็นฐาน 27 เพราะว่า ตัวอักษรภาษาอังกฤษตัวใหญ่มี 26 ตัว กับตัวว่างเปล่าอีกตัว รวมเป็น 27 ตัว สิ่งที่ต้องคำนึงถึงอีกประการหนึ่งคือเราเปรียบเทียบสตริงแบบชิดซ้าย เช่น "ABC" กับ "B" ต้องถือว่า "B" มากกว่า "ABC" ซึ่งไม่เหมือนกับกรณีของจำนวนเต็มซึ่งเป็นแบบชิดขวา เช่น 123 กับ 2 ดังนั้นเมท็อด `getDigit` จะไม่เหมือนกับกรณีของรหัสที่ 7-7 ขอให้ผู้อ่านลองศึกษาการทำงานของรหัสที่ 7-8 ดูเอง

การค้นตามแนวกว้าง

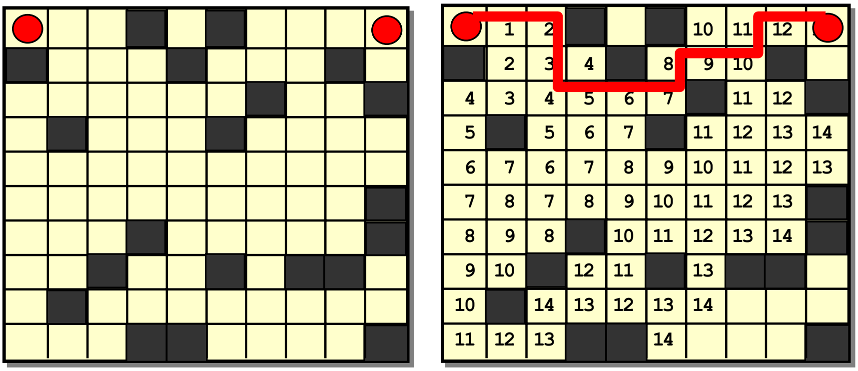
ถ้ายังจำกันได้ เราได้นำเสนอการใช้แถวคอยในการแก้ไขปริศนา 15 กันในบทที่ 1 ซึ่งในตอนนั้นแถวคอยถูกนำมาใช้เก็บตารางต่าง ๆ ระหว่างการลองเลื่อนช่องว่างของตารางหนึ่งในสี่ทิศทางเพื่อผลิตตารางใหม่ ๆ ในรูปลักษณะอื่น ๆ จนกว่าจะพบตารางที่เป็นคำตอบ เราเรียกกระบวนการค้นคำตอบโดยใช้วิธีการแจงแจงผลเฉลยนี้ว่า *การค้นในปริภูมิสถานะ* (state space search) การค้นคำตอบมีลำดับที่เป็นมาตรฐานหลากหลายแบบ เราได้ใช้แถวคอยเป็นตัวช่วยจัดเก็บและจัดลำดับตารางที่นำมาผลิตตารางใหม่ นั่นคือตารางใดเกิดก่อน ก็จะถูกนำไปผลิตตารางใหม่ ๆ ก่อน รูปที่ 7-6 แสดงผังการผลิตตารางต่าง ๆ ที่นำมาพิจารณาระหว่างการหาคำตอบของปริศนา 15 โดยตัวเลขภายในวงกลมสีเทาที่กำกับแต่ละตารางนั้นคือลำดับของตารางที่ถูกผลิตขึ้นระหว่างการค้นคำตอบ ให้สังเกตลำดับการผลิต เกิดการผลิตตามแนวกว้างลงไปทีละระดับ ๆ จึงเรียกว่า *การค้นตามแนวกว้าง* (breadth-first search) จะขอยกตัวอย่างการค้นตามแนวกว้างกับอีกสองปัญหาคือ การหาวิถีสั้นสุดในตาราง และปริศนาคูณสามหารสอง



รูปที่ 7-6 ลำดับการผลิตตารางในการค้นตามแนวกว้างของปริศนา 15

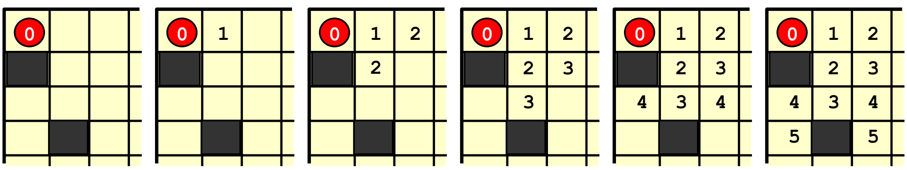
การหาวิถีสั้นสุดในตาราง

กำหนดให้มีตารางที่ภายในแบ่งเป็นช่องย่อย ๆ ขนาดเท่า ๆ กัน บางช่องเป็นสิ่งกีดขวาง บางช่องเป็นช่องว่าง ดังตัวอย่างในรูปที่ 7-7 ซ้าย ถ้าเราต้องการหาวิถีสั้นสุดจากมุมซ้ายบนถึงมุมขวาบน ห้ามทะลุช่องที่เป็นสิ่งกีดขวาง จะได้วิถีสั้นสุดทางแสดงในรูปที่ 7-7 ขวา



รูปที่ 7-7 การหาวิถีสั้นสุดในตารางที่มีบางช่องเป็นสิ่งกีดขวาง

การหาวิถีสั้นสุดตามแนวกว้าง มีลักษณะการทำงานดังนี้ (ดูรูปที่ 7-8 ประกอบ) เริ่มด้วยการใส่ค่า 0 ในช่องเริ่มต้น จากนั้นใส่ค่า 1 ในทุกช่องที่ติดกับช่องที่มีค่า 0 แล้วก็ใส่ค่า 2 ในทุกช่องที่ติดกับช่องที่มีค่า 1 ดำเนินการใส่ค่า $k + 1$ ในทุกช่องที่ติดกับช่องที่มีค่า k (ช่องใดที่เป็นสิ่งกีดขวางก็ไม่ต้องใส่ค่า) ทำเช่นนี้ไปเรื่อย ๆ จนชนช่องที่เป็นเป้าหมายก็เป็นอันว่าพบวิถีสั้นสุด จะสังเกตได้ว่า จำนวน k ที่เติมในช่องก็คือระยะทางสั้นสุดจากช่องเริ่มต้นถึงช่องนั้น ๆ โดยลักษณะการเติมจำนวนนี้ เสมือนการแผ่อาณาเขตของการใส่ระยะทางสั้นสุดในวงที่กว้างขึ้นเรื่อย ๆ จนชนเป้าหมาย (ในกรณีที่ไม่มีวิถีเดินไปถึงเป้าหมายเนื่องจากมีสิ่งกีดขวางปิดล้อมเป้าหมายไว้ การทำงานก็จะสิ้นสุดลงเมื่อเราเติมจนไม่มีอะไรจะเติมแล้ว) หลังการเติม การหาวิถีทำได้โดยเริ่มจากช่องเป้าหมายวิ่งตามจำนวนที่ลดลงทีละหนึ่งในช่องที่ติดกัน ก็จะพบวิถีสั้นสุดกลับไปยังช่องเริ่มต้น (ดูตัวอย่างในรูปที่ 7-7 ขวา)



รูปที่ 7-8 การแผ่อาณาเขตการค้นตามแนวกว้าง

```
public class Lee {
  private static final int SPACE = -1;
  private static final int BLOCK = -9;
  private static int[][] map = new int[10][10];

  private static class Pos {
    int row, col;
    Pos(int r, int c) {row = r; col = c;}
  }
  public static void main(String[] args) {
    for (int i = 0; i < map.length; i++)
      for (int j = 0; j < map[i].length; j++)
        map[i][j] = Math.random()<0.2 ? BLOCK : SPACE;
    findPath(new Pos(0,0), new Pos(0,map[0].length-1));
    for (int i = 0; i < map.length; i++) {
      for (int j = 0; j < map[i].length; j++)
        System.out.printf("%4d", map[i][j]);
      System.out.println();
    }
  }
  static void findPath(Pos source, Pos target) {
    map[source.row][source.col] = 0;
    map[target.row][target.col] = SPACE;
    Queue q = new ArrayQueue();  q.enqueue(source);
    while (!q.isEmpty()) {
      Pos p = (Pos) q.dequeue();
      if (p.row == target.row && p.col == target.col) break;
      expand(q, p.row + 1, p.col, map[p.row][p.col]);
      expand(q, p.row - 1, p.col, map[p.row][p.col]);
      expand(q, p.row, p.col + 1, map[p.row][p.col]);
      expand(q, p.row, p.col - 1, map[p.row][p.col]);
    }
  }
  static void expand(Queue q, int r, int c, int k) {
    if (r < 0 || r >= map.length ||
        c < 0 || c >= map[r].length ||
        map[r][c] != SPACE) return;
    map[r][c] = k + 1;
    q.enqueue(new Pos(r, c));
  }
}
```

คลาสภายในใช้เก็บตำแหน่งของช่องในตาราง

สุ่มใส่สิ่งกีดขวาง

หาวิถีสั้นสุด

แสดงตารางทางจอภาพ

เลิกค้นเมื่อพบตำแหน่งเป้าหมาย

แผ่อาณาเขตการค้นตามแนวกว้างไปทั้งสี่ทิศ

ถ้าตกขอบหรือไม่ใช่ช่องว่างก็ไม่เติม

เพิ่มระยะสั้นสุดขึ้นอีก 1 ให้กับช่องนี้และนำตำแหน่งของช่องนี้ใส่แถวคอย

รหัสที่ 7-9 โปรแกรมหาวิถีสั้นสุดในตาราง โดยใช้การค้นตามแนวกว้าง

สิ่งสำคัญของการหาวิถีสั้นสุดด้วยการค้นตามแนวกว้างคือ ต้องพิจารณาช่องที่มีค่า k เพื่อใส่ค่าในช่องที่ติดกับมันด้วยค่า k+1 โดยที่ k ต้องเป็นไปตามลำดับ 0, 1, 2, ... เรารับประกันเหตุการณ์นี้ได้ด้วยการใช้แถวคอยเก็บตำแหน่งของช่องที่เราใส่ระยะทางสั้นสุดแล้ว แต่ยังไม่ได้พิจารณาใส่ให้กับช่องที่ติดกับมัน รหัสที่ 7-9 แสดงโปรแกรมที่สุ่มสร้างตาราง หาวิถีสั้นสุด แล้วแสดงตารางออกทาง

จอภาพ (เมท็อด main) ภายในมีคลาส Pos ไว้แทนตำแหน่งของช่องในตาราง เราแทนตารางด้วยแถวลำดับสองมิติชื่อ map ค่าของช่องต่าง ๆ ใน map มี 3 ประเภท SPACE แทนช่องว่าง BLOCK แทนสิ่งกีดขวาง และจำนวนไม่ติดลบแทนระยะทางสั้นสุดที่ถูกเติมระหว่างการหาวิถี เมท็อด findPath ทำหน้าที่หาวิถีสั้นสุดเริ่มจากตำแหน่ง source ไปยังตำแหน่ง target เริ่มด้วยการเติมช่องของ source ให้มีค่าเป็น 0 และเปลี่ยนช่องของ target ให้เป็นช่องว่าง (บรรทัดที่ 22, 23) ตามด้วยการสร้างแถวคอย q พร้อมทั้งใส่ source เข้าใน q จากนั้นเข้าวงวน เริ่มด้วยการลบข้อมูลจาก q แล้วตรวจสอบ ถ้าเป็นตำแหน่งเป้าหมาย ให้ออกจากวงวนได้ ถ้าไม่ใช่ ให้เรียกเมท็อด expand เพื่อเติมระยะทางสั้นสุดในช่องที่ติดกันทั้งสี่ทิศ ใส่ระยะทางสั้นสุดที่เพิ่มขึ้นอีกหนึ่งไว้ในช่องที่ติดกัน และเพิ่มตำแหน่งของช่องเหล่านั้นเข้าในแถวคอย ทำครบสี่ทิศ ก็กลับไปทำต้นวงวนต่อ จะหลุดจากวงวนก็เมื่อพบ target (บรรทัดที่ 27) หรือ q ไม่มีข้อมูลเหลือ (บรรทัดที่ 25) ซึ่งแสดงว่า หาวิถีสั้นสุดไม่ได้ (เพราะมีสิ่งกีดขวางล้อมรอบ target ไว้)

ปริศนาคูณสามหารสอง

ว่ากันว่า จำนวนเต็มใด ๆ สามารถคำนวณได้จากการนำเลข 1 มาคูณสาม และ/หรือ หารสอง (ปัดเศษทิ้ง) ไปเรื่อย ๆ เช่น $10 = 1\times3\times3\times3/2/2/2$ จงหาวิธีคำนวณจำนวนเต็ม n ใด ๆ ในลักษณะนี้

รหัสที่ 7-10 แสดงโปรแกรมที่ค้นคำตอบของปริศนานี้ตามแนวกว้าง เริ่มด้วยการเพิ่ม 1 เข้าแถวคอย แล้วเข้าวงวน ภายในมีการดึงข้อมูลจากแถวคอย (เก็บในตัวแปร v) ถ้า v มีค่าเท่ากับจำนวนเป้าหมาย (ในตัวแปร target) ก็เลิกค้น ถ้าไม่เท่าก็เพิ่มค่า v/2 และ v×3 ใส่เข้าแถวคอย แล้วกลับไปทำต่อ ทำเช่นนี้ไปจนกว่าจะพบจำนวนเป้าหมาย หรือหลุดจากวงวนเพราะแถวคอยว่างซึ่งแสดงว่า target คำนวณแบบนี้ไม่ได้

```
public static void m3d2(int target) {
  Queue q = new ArrayQueue();
  q.enqueue(new Integer(1));                      เริ่มที่ 1
  while (!q.isEmpty()) {
    Integer v = (Integer) q.dequeue();
    if (v.intValue() == target) break;            พบคำตอบแล้ว
    q.enqueue(new Integer(v.intValue()/2));       ลองแบบ /2
    q.enqueue(new Integer(v.intValue()*3));       ลองแบบ ×3
  }
}
```

รหัสที่ 7-10 การค้นคำตอบของปริศนาคูณสามหารสองตามแนวกว้าง (ยังไม่สมบูรณ์)

รหัสที่ 7-10 ยังไม่ค่อยดีเพราะเราอาจผลิตจำนวนซ้ำแล้วซ้ำเล่าใส่แถวคอย เช่น 1 / 2 ได้ 0 พอนำ 0 ออกมาผลิต 0/2 กับ 0×3 ก็ได้ 0 ใส่แถวคอยอีก รหัสที่ 7-11 ขจัดปัญหานี้ได้ด้วยการใช้เซตเก็บ

จำนวนต่าง ๆ ที่เคยผลิตมาทั้งหมดตั้งแต่เริ่มทำงาน เพื่อนำมาตรวจสอบก่อนว่า ถ้าซ้ำกับที่เก็บในเซตนี้ก็ไม่ต้องเพิ่มในแถวคอย

```
Queue q = new ArrayQueue(); Set s = new ArraySet();
q.enqueue(new Integer(1)); s.add(new Integer(1));
while (!q.isEmpty()) {
  Integer v = (Integer) q.dequeue();
  if (v.intValue() == target) break;
  Integer v1 = new Integer(v.intValue()/2);
  Integer v2 = new Integer(v.intValue()*3);
  if (!s.contains(v1)) {q.enqueue(v1); s.add(v1);}
  if (!s.contains(v2)) {q.enqueue(v2); s.add(v2);}
}
```

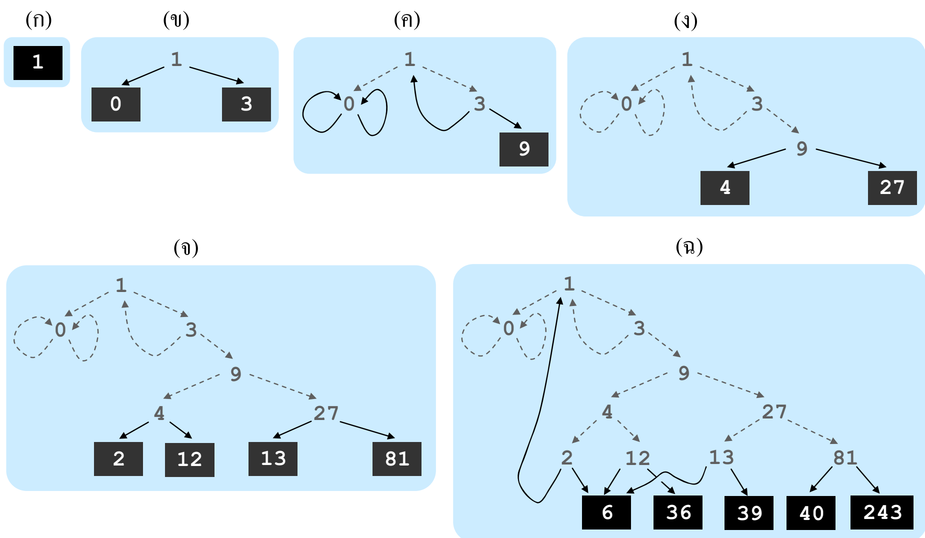
เซต s มีไว้เก็บจำนวนที่เคยผลิตมา

เพิ่มในเซตด้วย

ถ้าไม่เคยพบมาก่อน จึงเพิ่มเข้าแถวคอย

รหัสที่ 7-11 การใช้เซตเก็บจำนวนเต็มทั้งหมดที่เคยผลิตมาไว้ตรวจสอบความซ้ำซ้อน

รูปที่ 7-9 แสดงการผลิตจำนวนระหว่างการค้นตามแนวกว้าง โดยตัวเลขขาวพื้นทึบคือจำนวนที่เก็บในแถวคอย เริ่มที่ 1 ใส่เข้าแถวคอย (รูป ก) เข้าวงวนลบได้ 1 ออกแล้วผลิต 0 กับ 3 (รูป ข เส้นที่พุ่งไปทางซ้ายแทนการ /2 ส่วนเส้นที่พุ่งไปทางขวาแทนการ ×3) จากนั้นลบได้ 0 ออกแต่ผลิต 0 ซ้ำ ลบได้ 3 ออกผลิต 1 (ซึ่งซ้ำ) กับ 9 (รูป ค) ลบได้ 9 แล้วผลิต 4 กับ 27 (รูป ง) ลบได้ 4 แล้วผลิต 2 กับ 12 และ ลบได้ 27 แล้วผลิต 13 กับ 81 (รูป จ) ถึงตอนนี้แถวคอยมี 2, 12, 13, และ 81 รูป ฉ แสดงผลที่ได้จากการลบและผลิตข้อมูลในระดับถัดไป เมื่อใดข้อมูลที่ลบได้คือเป้าหมายที่ต้องการก็จบ



รูปที่ 7-9 การแก้ปริศนาคูณสามหารสองด้วยการค้นตามแนวกว้าง

แต่หลังจากรหัสที่ 7-11 ค้นพบเป้าหมายแล้ว จะรู้ว่า ต้องคูณสามและหารสองอย่างไรจึงจะได้เป้าหมาย เพราะเราไม่ได้จำลำดับการคูณสามและหารสองเลย จึงต้องปรับปรุงให้ข้อมูลแต่ละตัวจำ

ด้วยว่า ถูกผลิตจากข้อมูลตัวใด รหัสที่ 7-12 แสดงโปรแกรมที่สมบูรณ์ มีคลาสภายในชื่อ Node ซึ่งมาแทน Integer ที่เราเคยใช้ ภายในคลาส Node เก็บจำนวนเต็ม (value) และ Node ก่อนหน้า (prev) ที่เป็นตัวผลิต Node ปัจจุบัน สำหรับเลข 1 ซึ่งเป็นตัวตั้งต้นนั้นไม่มีตัวก่อนหน้า ก็ให้ค่า prev เป็น null บรรทัดที่ 20 และ 21 เป็นขั้นตอนการผลิตข้อมูลใหม่ v1 และ v2 ที่ได้มาจาก v จึงส่งค่า v ไปยังตัวสร้างของ Node เพื่อให้เป็น prev ของข้อมูลตัวใหม่ (นั่นคือ v1.prev และ v2.prev มีค่าเป็น v) ดังนั้นเมื่อพบเป้าหมายแล้ว ก็สามารถวิ่งกลับจากตัวเป้าหมายผ่าน prev กลับไปยังเลข 1 ก็จะได้ลำดับของการคูณสามหารสองที่ได้ผลเป็นจำนวนเป้าหมาย ด้วยเมท็อด solution ในรหัสที่ 7-12

```
01 public class M3D2 {
02   public static void main(String[] args) {
03     System.out.println("31 = " + m3d2(31));
04   }
05   private static class Node {
06     int value;
07     Node prev;
08     Node(int v, Node p) {value = v; prev = p;}
09     public boolean equals(Object o) {
10       return this.value == ((Node) o).value;
11     }
12   }
13   public static String m3d2(int target) {
14     Queue q = new ArrayQueue(); Set s = new ArraySet();
15     Node v = new Node(1, null);
16     q.enqueue(v); s.add(v);
17     while (!q.isEmpty()) {
18       v = (Node) q.dequeue();
19       if (v.value == target) break;
20       Node v1 = new Node(v.value / 2, v);
21       Node v2 = new Node(v.value * 3, v);
22       if (!s.contains(v1)) {q.enqueue(v1); s.add(v1);}
23       if (!s.contains(v2)) {q.enqueue(v2); s.add(v2);}
24     }
25     return v.value == target ? solution(v) : "???";
26   }
27   private static String solution(Node v) {
28     if (v.prev == null) return "1";
29     return solution(v.prev) +
30            (v.prev.value/2 == v.value ? "/2" : "x3");
31   }
32 }
```

ต้องมี equals ไว้ให้ ArraySet ใช้ใน contains

1 ไม่มีตัวก่อนหน้า ให้ prev = null

v1 และ v2 ถูกผลิตจาก v

รหัสที่ 7-12 โปรแกรมการค้นคำตอบของปริศนาคูณสามหารสองตามแนวกว้าง

ผู้อ่านอาจสงสัยว่า ทำไมเราต้องใช้แถวคอยด้วย ใช้คอลเล็กชัน ใช้กองซ้อนก็น่าจะได้ เพราะต่างก็เก็บข้อมูลทุกตัว ลองทุกแบบ การค้นก็ต้องพบเป้าหมายเข้าสักวัน ต้องขอบอกว่า การใช้แถวคอยทำให้เกิดการค้นตามแนวกว้าง ถ้าดูรูปที่ 7-9 อีกครั้ง จะพบว่า การผลิตจำนวนใหม่ ๆ มาพิจารณานั้นทำแบบมีระเบียบเป็นระดับ ๆ ดังนั้นเมื่อพบเป้าหมาย จะประกันได้ว่า เป้าหมายอยู่ในระดับที่ใกล้เลข 1 ที่สุด หมายความว่า การคูณสามหารสองที่ได้เป็นแบบที่มีจำนวนตัวดำเนินการน้อยสุด

แบบฝึกหัด

1. จงเขียนคลาส `ArrayListQueue` (ซึ่งสร้างแถวคอยโดยใช้รายการแบบ `ArrayList` ช่วยเก็บข้อมูล) และ `ArrayQueue` (ซึ่งสร้างแถวคอยด้วยแถวลำดับแบบวงวน) ด้วยตนเอง โดยไม่ดูรายละเอียดในหนังสือ

2. `ArrayQueue` ที่ได้เขียนมามีตัวแปร `front` และ `size` กำกับแถวคอย โดยเราใช้ `front` และ `size` เพื่อคำนวณตำแหน่งของช่องที่จะใส่ข้อมูลใหม่ จงเขียน `ArrayQueue` ใหม่ที่มีตัวแปร `front` และ `back` โดย `front` เก็บตำแหน่งหัวแถว ส่วน `back` เก็บตำแหน่งท้ายแถว โดยไม่ต้องเก็บตัวแปร `size`

3. คลังคลาสมาตรฐานของจาวา (ตั้งแต่รุ่น 5 เป็นต้นไป) ก็มีอินเทอร์เฟซ `Queue` แต่ใช้ชื่อเมท็อดของบริการต่าง ๆ ไม่เหมือนกับที่เราได้นำเสนอมา จงเปรียบเทียบชื่อเมท็อดที่ใช้ พร้อมหาว่า มีคลาสอะไรบ้างในคลังคลาสมาตรฐานของจาวาที่นำมาสร้างแถวคอยได้

4. จงปรับปรุงให้คลาส `LinkedList implements Queue` ด้วย

5. `inc` ในรหัสที่ 7-4 ควรเขียนเป็น `return (++i == elementData.length ? 0 : i)` เห็นด้วยไหม ? ให้ลองทำการทดลอง เรียกเมท็อด `testQueue` ข้างล่างนี้โดยส่งแถวคอยที่ใช้ `inc` แบบเดิม เปรียบเทียบเวลาการทำงานเมื่อเปลี่ยน `inc` เป็นแบบใหม่ (ให้ `n` มีค่าสักหนึ่งล้าน)

```
static void testQueue(Queue q, int n) {
  for (int i=0; i<n; i++) q.enqueue("A");
  long t = System.nanoTime();
  for (int i=0; i<n; i++) q.dequeue();
  System.out.println(q.getClass().getName() + ".dequeue : " +
      ((System.nanoTime() - t)/1000000.0));
}
```

6. จงเขียนคลาส `QueueX` ซึ่งสร้างแถวคอยด้วยแถวลำดับแบบวงวน มีตัวสร้างรับขนาดมากสุดของแถวคอย มีเมท็อด `isFull` ไว้ตรวจสอบว่า เต็มหรือไม่ และการ `enqueue` จะไม่ขยายขนาดของที่เก็บเมื่อเต็ม แต่จะโยน `IllegalStateException` แทน

7. จงเปลี่ยนการใช้แถวคอยในรหัสที่ 7-12 มาเป็นกองซ้อน จากนั้นลงเปรียบเทียบผลลัพธ์ของคำตอบที่ได้เมื่อใช้แถวคอย กับใช้กองซ้อน แตกต่างกันอย่างไร

8. รหัสที่ 7-12 ใช้กับปริศนาคูณสามหารสอง จงเปลี่ยนให้ใช้หาคำตอบของปริศนาคูณสองหารสาม

9. การหาวิถีสั้นสุดของรหัสที่ 7-9 ใช้เวลาการทำงานเท่าไร (วิเคราะห์ให้เป็นฟังก์ชันของจำนวนช่องทั้งหมดของตาราง)

10. จงออกแบบคลาสใหม่สำหรับสร้างที่เก็บข้อมูลที่เรียกกันว่า *แถวคอยสองด้าน* (double-ended queue หรือเรียกสั้น ๆ ว่า *เดก* deque) ซึ่งเป็นที่เก็บที่ข้อมูลที่มีบริการเพิ่มและลบข้อมูลได้ทั้งด้านหน้าและด้านหลัง

8 แถวคอยบุริมภาพ

แถวคอยที่ได้ศึกษามามีระเบียบ ใครเข้าแถวก่อน ก็ออกจากแถวก่อน แต่บางครั้งเราก็อยากให้มีการลัดแถว ใครสำคัญสุดก็ให้ไปรอที่หัวแถว แถวคอยที่จัดแถวตามลำดับความสำคัญเช่นนี้เรียกว่า *แถวคอยบุริมภาพ* (priority queue) บทนี้นำเสนอโครงสร้างข้อมูลสำหรับแถวคอยบุริมภาพทั้งแบบง่ายแต่ประสิทธิภาพไม่ค่อยดี กับแบบซับซ้อนแต่ทำงานเร็วกว่า พร้อมทั้งการนำแถวคอยบุริมภาพไปใช้ในงานประยุกต์หลากหลาย

ข้อกำหนดของแถวคอยบุริมภาพ

แถวคอยบุริมภาพมีบริการเหมือนกับแถวคอยคือ `size`, `isEmpty`, `enqueue`, `peek` และ `dequeue` โดย `peek` และ `dequeue` มีความหมายต่างกับของแถวคอย คือให้บริการขอดู และลบข้อมูลที่สำคัญสูงสุดตามลำดับ จึงนิยามอินเทอร์เฟซ `PriorityQueue` (รหัสที่ 8-1) ที่ขยายจากอินเทอร์เฟซ `Queue` แต่ไม่มีเมท็อดอะไรเพิ่ม เพียงแต่เปลี่ยนคำอธิบายความหมายเท่านั้นของ `peek` และ `dequeue` เพื่อเป็นข้อตกลงร่วมกัน ระหว่างผู้ออกแบบและผู้ใช้

```
public interface PriorityQueue extends Queue{
  public Object dequeue();         // ลบข้อมูลตัวสำคัญที่สุด
  public Object peek();            // ขอดูข้อมูลตัวสำคัญที่สุด
}
```

รหัสที่ 8-1 อินเทอร์เฟซ `PriorityQueue`

ในระบบจาวา มีอินเทอร์เฟซมาตรฐานตัวหนึ่งชื่อ `Comparable` (รหัสที่ 8-2) บังคับให้มีเมท็อด `compareTo` โดยผลของการเรียก `a.compareTo(b)` เป็นจำนวนเต็ม ถ้าเป็นจำนวนลบแสดงว่า a น้อยกว่า b ถ้าเป็นศูนย์แสดงว่า a เท่ากับ b และถ้าเป็นจำนวนบวกแสดงว่า a มากกว่า b เมื่อใดที่เรามีงานซึ่งต้องเปรียบเทียบความน้อยกว่ามากกว่าของอ็อบเจกต์ในระบบ เช่น การเรียงลำดับ

ข้อมูล ก็จะบังคับว่า อ็อบเจกต์ที่นำมาประมวลผลต้องเป็นของคลาสที่ implements Comparable วิธีเปรียบเทียบอ็อบเจกต์จึงเป็นภาระของผู้ออกแบบคลาสว่า เมท็อด compareTo จะเปรียบเทียบอะไร อย่างไร รหัสที่ 8-3 แสดงตัวอย่างคลาส Rectangle ที่เป็น Comparable โดยใช้พื้นที่ของสี่เหลี่ยมเป็นตัวเปรียบเทียบ

```
public interface Comparable {
  // คืนค่าลบ      ถ้า this น้อยกว่า obj
  // คืน 0         ถ้า this เท่ากับ obj
  // คืนค่าบวก     ถ้า this มากกว่า obj
  public int compareTo(Object obj);
}
```

ค่าลบและค่าบวกนี้ จะบวกหรือลบเท่าใดก็ได้ ผู้ใช้จะสนใจเฉพาะเครื่องหมาย

รหัสที่ 8-2 อินเทอร์เฟซ Comparable ไว้ใช้เปรียบเทียบข้อมูล

```
public class Rectangle implements Comparable {
  private int width, height;
  ...
  public int compareTo(Object obj) {
    Rectangle that = (Rectangle) obj;
    int thisArea = width * height;
    int thatArea = that.width * that.height;
    return thisArea - thatArea;
  }
}
```

อย่าลืม ต้องเขียน implements ...

ใช้พื้นที่เปรียบเทียบ

รหัสที่ 8-3 การเปรียบเทียบสี่เหลี่ยมด้วยการเปรียบเทียบพื้นที่

เรานำอินเทอร์เฟซ Comparable มาใช้กับแถวคอยบุริมภาพ โดยกำหนดให้อ็อบเจกต์ที่นำมาเก็บในแถวคอยต้องเป็นแบบ Comparable ที่เมท็อด compareTo ของอ็อบเจกต์มีไว้เปรียบเทียบความสำคัญ รหัสที่ 8-4 แสดงตัวอย่างการใช้งานแถวคอยบุริมภาพ ที่เก็บอ็อบเจกต์ของคลาส Integer ซึ่งก็เป็น Comparable ที่อาศัยจำนวนเต็มภายในเป็นตัวเปรียบเทียบ หลังจากเพิ่ม 5 และ 7 ตามด้วย dequeue จะได้ 7 เพราะมากสุด จากนั้นเพิ่ม 8 และ 3 แล้วเข้าวงวนลบข้อมูลจนหมด ก็ย่อมได้ 8, 5, และ 3 ออกมาตามลำดับ

```
PriorityQueue q = new ArrayListPQ();
q.enqueue(new Integer(5));
q.enqueue(new Integer(7));
System.out.println(q.dequeue());
q.enqueue(new Integer(8));
q.enqueue(new Integer(3));
while (!q.isEmpty()) {
  System.out.println(q.dequeue());
}
```

ลบได้ 7

ได้ 8, 5, และ 3 ตามลำดับ

รหัสที่ 8-4 ตัวอย่างการใช้แถวคอยบุริมภาพ

การสร้างด้วยรายการ

เราสามารถสร้างแถวคอยบุริมภาพได้ง่าย ๆ ด้วยการนำรายการมาเก็บซ่อนไว้ในคลาส แล้วให้รายการนี้ทำหน้าที่จัดเก็บและจัดการข้อมูลแทน รหัสที่ 8-5 แสดงตัวอย่างวิธีดังกล่าว มีตัวแปร `list` ไว้เก็บอ็อบเจกต์ของ `ArrayList` ให้บริการ `isEmpty` และ `size` ที่ส่งต่อไปถามที่ตัว `list` เมท็อด `enqueue` ก็เพียงแต่เพิ่มข้อมูลต่อท้ายรายการเพราะเร็วสุด `peek` เรียกใช้เมท็อด `maxIndex` วิ่งเปรียบเทียบหาตำแหน่งของตัวมากสุดในรายการ ส่วน `dequeue` นั้นอาศัย `maxIndex` ได้ข้อมูลตัวมากสุด แล้วสั่งให้รายการลบตัวนั้นออก

```java
public class ArrayListPQ implements PriorityQueue {
  private ArrayList list = new ArrayList();
  public boolean isEmpty() { return list.isEmpty();}
  public int size() { return list.size(); }
  public Object peek() {
    return list.get(maxIndex());
  }
  public void enqueue(Object e) {
    list.add(list.size(), e);
  }
  public Object dequeue() {
    int max = maxIndex();
    Object result = list.get(max);
    list.remove(max);
    return result;
  }
  private int maxIndex() {
    if (isEmpty())
      throw new IllegalStateException();
    int max = 0;
    for (int i = 1; i < list.size(); i++) {
      Comparable d = (Comparable) list.get(i);
      if (d.compareTo(list.get(max)) > 0) max = i;
    }
    return max;
  }
}
```

ใช้ maxIndex เพื่อหาตำแหน่งตัวมากสุด

เพิ่มต่อท้ายรายการเพราะเร็วสุด

คืนตำแหน่งในรายการ
ที่เก็บตัวมากสุด

get คืน Object ต้อง cast เป็น Comparable ก่อน จึงจะเรียกใช้ compareTo ได้

ถ้า d มากกว่าตัวที่ max ใน list ก็เปลี่ยนค่า max

รหัสที่ 8-5 การสร้างแถวคอยบุริมภาพด้วยการใช้รายการจัดเก็บและจัดการข้อมูลแทน

เมท็อด `isEmpty` และ `size` ในรหัสที่ 8-5 ใช้เวลา $\Theta(1)$, `enqueue` ก็ใช้เวลา $\Theta(1)$ เพราะเป็นการเพิ่มท้ายรายการแบบ `ArrayList` ส่วน `peek` ใช้เวลา $\Theta(n)$ เพราะต้องวิ่งหาตัวมากสุด ซึ่งก็เหมือนกับ `dequeue` ที่ทั้งต้องวิ่งหาตัวมากสุด และลบตัวนั้นทิ้ง

รหัสที่ 8-6 แสดงการปรับปรุงให้ peek ทำงานเร็วขึ้นเป็น $\Theta(1)$ โดยมีตัวแปร max จำตำแหน่งของตัวมากสุดไว้ตลอดเวลา โดยก่อนจะเพิ่มข้อมูลใหม่ต่อท้ายรายการ ก็เปรียบเทียบก่อนว่า มีค่ามากกว่าตัวที่ max ของรายการหรือไม่ ถ้ามากกว่าก็เปลี่ยนค่า max (บรรทัดที่ 16) ด้วยวิธีนี้ peek ก็เพียงแค่คืนข้อมูลตัวที่ max ของรายการ (บรรทัดที่ 9) และที่ต้องปรับอีกที่ก็คือ dequeue ที่หลังจากลบตัวที่ max ออกแล้ว ก็ต้องวิ่งเปรียบเทียบหาตำแหน่งของตัวมากสุด แล้วตั้งค่าให้กับ max เพื่อใช้เป็นประโยชน์ในอนาคต (บรรทัดที่ 24 ถึง 28)

```
01 public class ArrayListPQ implements PriorityQueue {
02   private ArrayList list = new ArrayList();
03   private int max;
04
05   public boolean isEmpty() { return list.isEmpty(); }
06   public int size() { return list.size(); }
07   public Object peek() {
08     if (isEmpty()) throw new IllegalStateException();
09     return list.get(max);
10   }
11   public void enqueue(Object e) {
12     if (isEmpty()) {
13       max = 0;
14     } else {
15       Comparable m = (Comparable) list.get(max);
16       if (m.compareTo(e) < 0) max = list.size();
17     }
18     list.add(list.size(), e);
19   }
20   public Object dequeue() {
21     Object result = peek();
22     list.remove(max);
23     if (!isEmpty()) {
24       max = 0;
25       for (int i = 1; i < list.size(); i++) {
26         Comparable d = (Comparable) list.get(i);
27         if (d.compareTo(list.get(max)) > 0) max = i;
28       }
29     }
30     return result;
31   }
32 }
```

จำตำแหน่งของตัวมากสุด peek จะได้เป็น $\Theta(1)$

คืนตัวที่ max ของรายการ ใช้ $\Theta(1)$

ถ้า list ว่าง เดี๋ยวต้องเติมช่อง 0 และเป็นตัวมากสุดแน่

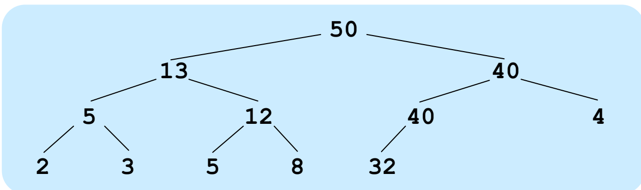
ถ้าตัวมากสุดน้อยกว่าตัวใหม่ ก็เปลี่ยน max ให้เป็นตำแหน่งท้ายรายการ เพราะเป็นที่ที่เราจะเพิ่มในรายการ

ไล่เปรียบเทียบหาตำแหน่งตัวมากสุดใน รายการเพื่อเก็บใน max

รหัสที่ 8-6 การปรับรหัสที่ 8-5 ให้จำตำแหน่งข้อมูลที่สำคัญสุดไว้ ทำให้ peek เป็น $\Theta(1)$

การสร้างด้วยฮีปแบบทวิภาค

หัวข้อนี้นำเสนอการสร้างแถวคอยบุริมภาพด้วยโครงสร้างข้อมูลที่ทั้งประหยัดเนื้อที่และมีประสิทธิภาพ โดยเมท็อด `isEmpty`, `size`, `peek` ใช้เวลา $\Theta(1)$ ส่วน `enqueue` และ `dequeue` ใช้เวลา $O(\log n)$ โครงสร้างข้อมูลนี้มีชื่อว่า *ฮีปแบบทวิภาค* (binary heap) มีโครงสร้างการจัดเก็บในลักษณะของต้นไม้ได้ดุล (รูปที่ 8-1) จึงเป็นต้นไม้ที่สูงเท่ากับ $\lfloor \log_2 n \rfloor$ ให้สังเกตว่า ต้นไม้มีโครงสร้างแบบทวิภาค มีปมเต็มในทุก ๆ ระดับ ยกเว้นระดับล่างสุด ปมทั้งหลายจะถูกวางเรียงจากซ้ายไปขวา นอกจากนี้ข้อมูลในต้นไม้มีค่าความสำคัญในลักษณะที่ว่า ความสำคัญของปมพ่อต้องไม่น้อยกว่าของลูกทั้งสอง เรียกว่า มี*อันดับแบบฮีป* (heap-order) (เพื่อความง่ายในการนำเสนอ จำนวนที่แสดงตามปมแทนค่าความสำคัญของข้อมูลที่เก็บที่ปมนั้น ขอไม่แสดงตัวข้อมูล แสดงแต่ค่าความสำคัญเท่านั้น) ด้วยเงื่อนไขอันดับแบบฮีปนี้ ประกันได้ว่า ข้อมูลที่มีความสำคัญมากสุดต้องอยู่ที่รากของต้นไม้

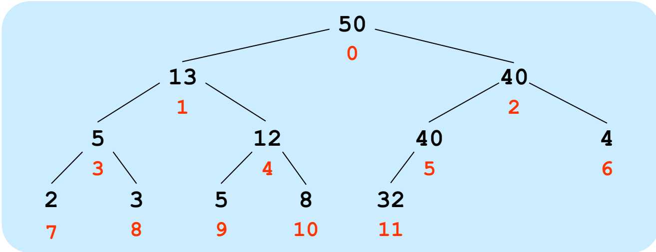


รูปที่ 8-1 ต้นไม้ฮีปแบบทวิภาค (จำนวนที่แสดงแทนค่าความสำคัญของข้อมูล)

การแทนฮีปแบบทวิภาคด้วยแถวลำดับ

ด้วยลักษณะของโครงสร้างต้นไม้แบบทวิภาคได้ดุล มีปมเต็มทุกระดับ ยกเว้นระดับล่างสุดที่วางปมจากซ้ายไปขวา ทำให้เราสามารถแทนฮีปแบบทวิภาคด้วยแถวลำดับเพียงหนึ่งแถวกับตัวแปรเก็บจำนวนข้อมูลอีกหนึ่งตัวเท่านั้น ดูรูปที่ 8-2 เป็นตัวอย่าง ขอกำกับหมายเลขให้กับปมต่าง ๆ (เขียนไว้ด้านล่างของปม) โดยเริ่มที่รากให้หมายเลข 0 จากนั้นไล่จากซ้ายไปขวา บนลงล่าง ไปทีละระดับ ๆ ให้หมายเลขเพิ่มขึ้นทีละหนึ่ง หมายเลขเหล่านี้ก็คือเลขที่ช่องของแถวลำดับที่เก็บข้อมูล การเก็บในลักษณะนี้ทำให้สามารถคำนวณหาเลขที่ของปมพ่อ ลูกซ้าย และลูกขวาดังนี้

- ลูกซ้ายของปมที่เก็บในช่องที่ k ถูกเก็บในช่องที่ $2k + 1$
- ลูกขวาของปมที่เก็บในช่องที่ k ถูกเก็บในช่องที่ $2k + 2$
- พ่อของปมที่เก็บในช่องที่ k ถูกเก็บในช่องที่ $(k - 1) / 2$



size		0	1	2	3	4	5	6	7	8	9	10	11	12	13
12	elementData	50	13	40	5	12	40	4	2	3	5	8	32		

รูปที่ 8-2 ตัวอย่างการแทนฮีปแบบทวิภาคด้วยแถวลำดับ

เนื่องจากข้อมูลทุกตัวถูกเก็บในแถวลำดับติด ๆ กัน ตั้งแต่ช่องเลขที่ 0 เป็นต้นไป ดังนั้นเลขที่ของปมที่ถูกต้อง คือตั้งแต่ 0 ถึง `size-1` ถ้าอยู่นอกช่วงนี้ แสดงว่า ไม่มีปมนั้น เช่น ในรูปที่ 8-2 จะรู้ได้โดยพิจารณาตัวแปร `size` ว่าปมในช่องเลขที่ 5 มีแต่ลูกซ้ายไม่มีลูกขวา เพราะลูกซ้ายอยู่ช่องเลขที่ $2\times5+1=11$ ซึ่งน้อยกว่า `size` ในขณะที่ลูกขวาอยู่ช่องที่ $2\times5+2=12$ ไม่น้อยกว่า `size` แสดงว่าไม่มีลูกขวา รหัสที่ 8-7 แสดงส่วนต้น ๆ ของคลาส `BinaryHeap` ที่เราจะเขียนในหัวข้อนี้

```
public class BinaryHeap implements PriorityQueue {
  Object[] elementData = new Object[10];
  int size;

  public BinaryHeap() {}
  public boolean isEmpty() { return size == 0; }
  public int size() { return size; }
  public Object peek() {
    if (isEmpty()) throw new IllegalStateException();
    return elementData[0];
  }
  ...
```

แถวลำดับที่เก็บข้อมูล

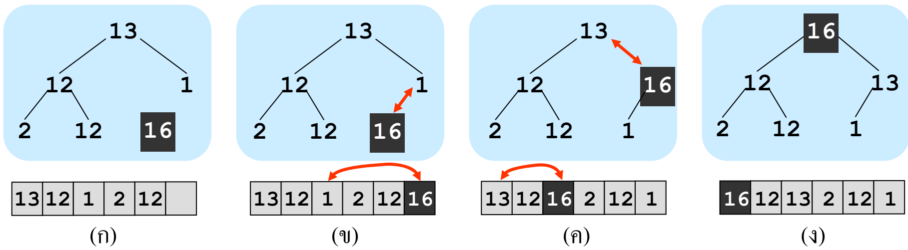
ข้อมูลตัวมากสุด (รากของฮีป) อยู่ช่องเลขที่ 0

รหัสที่ 8-7 การจัดเก็บข้อมูลภายในคลาส `BinaryHeap`

void enqueue(Object e)

เนื่องจากข้อมูลถูกจัดเก็บในแถวลำดับติด ๆ กัน ตั้งแต่ช่องเลขที่ 0 จนถึง `size-1` ดังนั้นการเพิ่มข้อมูลตัวใหม่ให้เร็วสุด ต้องเพิ่มในช่องเลขที่ `size` (ในกรณีที่แถวลำดับมีขนาดไม่พอ ก็ให้ขยายก่อนเพิ่ม) การเพิ่มต่อท้ายในลักษณะนี้ อาจทำให้ความสัมพันธ์ของข้อมูลใหม่กับปมพ่อไม่ตรงตามอันดับแบบฮีป จึงต้องแก้ไขลำดับข้อมูลในแถวลำดับบ้าง เพื่อให้ค่าความสำคัญของปมพ่อไม่น้อยกว่าของปมลูก รูปที่ 8-3 แสดงตัวอย่างการเพิ่มข้อมูลใหม่ที่มีค่าความสำคัญเป็น 16 ในรูป (ก) ต่อท้ายได้ดังรูป (ข) ถึงขณะนี้ 16 มีความสำคัญมากกว่าพ่อ (ซึ่งคือ 1) ก็ให้สลับกับปมพ่อได้ดังรูป (ค) ซึ่งก็ต้องสลับกับ

ปมพ่อต่ออีก เพราะ 16 ยังคงสำคัญกว่า 13 ได้ดังรูป (ง) รหัสที่ 8-8 แสดงรายละเอียดของเมท็อด `enqueue` เริ่มด้วยการขยายขนาดของแถวลำดับถ้าจำเป็น (บรรทัดที่ 13) นำข้อมูลใหม่ใส่ด้านท้าย แล้วเริ่มกระบวนการปรับลำดับของข้อมูลในแถวด้วยเมท็อด `fixUp` โดยเมื่อเทียบกับตัวอย่างในรูปที่ 8-3 ตัวแปร `k` คือตำแหน่งของข้อมูลสีดำในรูป และตัวแปร `p` คือตำแหน่งของพ่อของข้อมูลสีดำ ทำงานในวงวนเพื่อสลับข้อมูลขึ้นไปเรื่อย ๆ จนกระทั่งถึงราก (เมื่อเงื่อนไขของ `while` ที่บรรทัดที่ 18 เป็นเท็จ) หรือจนกระทั่งไม่สำคัญกว่าพ่อแล้ว (บรรทัดที่ 20) โดยสรุป `enqueue` เพิ่มข้อมูลที่ใบ เกิดการสลับข้อมูลขึ้นตามความสูงไม่เกิน $\lfloor \log_2 n \rfloor$ ครั้ง เมื่อมีข้อมูล n ตัว จึงใช้เวลาเป็น $O(\log n)$



รูปที่ 8-3 ตัวอย่างแสดงขั้นตอนการเพิ่ม 16 เข้าในฮีปแบบทวิภาค

```
12    public void enqueue(Object e) {
13      ensureCapacity(size+1);
14      elementData[size] = e;
15      fixUp(size++);
16    }
17    private void fixUp(int k) {
18      while (k > 0) {
19        int p = (k-1)/2;
20        if (!greaterThan(k, p)) break;
21        swap(k, p);
22        k = p;
23      }
24    }
25    protected boolean greaterThan(int i, int j) {
26      Comparable e = (Comparable) elementData[i];
27      return e.compareTo(elementData[j]) > 0;
28    }
..    ...
```

ขยายอาเรย์ถ้าจำเป็น

size คือตำแหน่งของข้อมูลใหม่ และต้องเพิ่มค่า size ด้วย

สลับข้อมูลกับปมพ่อขึ้นไปเรื่อย ๆ จนกว่าจะไม่สำคัญกว่าของพ่อ

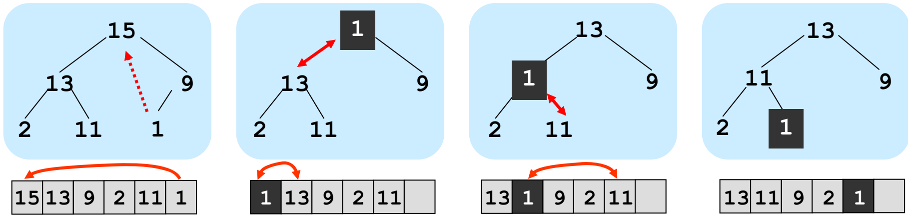
เป็น Object ต้อง cast เป็น Comparable ก่อน

รหัสที่ 8-8 การเพิ่มข้อมูลของ `BinaryHeap`

Object dequeue()

การลบตัวสำคัญที่สุด ก็คือการลบราก วิธีลบรากทิ้งอย่างรวดเร็วอาศัยการย้ายข้อมูลตัวระดับล่างสุดขวาสุด (ถ้าดูในแถวลำดับก็คือ ตัวท้ายสุดช่องที่ `size-1`) มาแทนราก (ซึ่งก็คือช่องที่ 0) รูปที่ 8-4 แสดงการลบตัวสำคัญสุดออกจากฮีปในรูป (ก) ทำได้โดยนำ 1 ซึ่งคือตัวท้ายแถวมาแทนช่องที่ 0 ซึ่งคือตัว

สำคัญสุดที่จะลบทิ้ง ได้ดังรูป (ข) การย้ายเช่นนี้ย่อมทำให้ความสัมพันธ์ของรากกับลูก ๆ อาจไม่ตรงตามอันดับแบบฮีป เช่น ในรูป (ข) พบว่า 1 น้อยกว่า 13 และ 9 ซึ่งผิดกฎ วิธีแก้ไขอาศัยการสลับข้อมูลกับลูกตัวมาก เช่น ในรูป (ข) สลับ 1 กับ 13 (ซึ่งเป็นตัวมากของ 13 กับ 9) ได้ดังรูป (ค) ซึ่งอาจทำให้ผิดอันดับแบบฮีปในระดับล่างลงมา ก็สลับข้อมูลลงไปยังลูกตัวมากต่อลงไปอีก เช่น ในรูป (ค) ก็สลับ 1 กับ 11 ได้ดังรูป (ง) กระทำการสลับกับลูกลงไปเรื่อย ๆ จนกว่าจะไม่มีลูกให้สลับแล้ว หรือจนกว่าจะไม่ผิดกฎ



รูปที่ 8-4 ตัวอย่างแสดงขั้นตอนการลบรากออกจากฮีปแบบทวิภาค

```
29    public Object dequeue() {
30      Object max = peek();
31      elementData[0] = elementData[--size];
32      elementData[size] = null;
33      if (size > 1) fixDown(0);
34      return max;
35    }
36    private void fixDown(int k) {
37      int c;
38      while ((c = 2 * k + 1) < size) {
39        if (c + 1 < size && greaterThan(c+1, c)) c++;
40        if (!greaterThan(c, k)) break;
41        swap(k, c);
42        k = c;
43      }
44    }
..    ...
```

ย้ายตัวท้ายไปที่ช่อง 0 แล้วลบ reference ที่ไม่จำเป็นออก

ตราบเท่าที่ยังมีลูกซ้าย

ถ้ามีลูกขวาและขวามากกว่าซ้าย

เลิก เมื่อลูกไม่สำคัญกว่าพ่อ

สลับพ่อกับลูกตัวมาก แล้วลงไปทำต่อ

รหัสที่ 8-9 การลบข้อมูลตัวสำคัญที่สุดของ BinaryHeap

รหัสที่ 8-9 แสดงเมท็อด dequeue เพื่อลบตัวสำคัญที่สุดออกจากฮีปแบบทวิภาค เริ่มด้วยการเรียก peek หยิบตัวมากสุดมาเก็บไว้ก่อน (บรรทัดที่ 30) เพื่อคืนกลับให้ผู้เรียกในบรรทัดที่ 34 ต่อด้วยการย้ายข้อมูลตัวท้ายไปแทนตัวมากสุดซึ่งอยู่ช่องเลขที่ 0 แล้วเข้ากระบวนการปรับลำดับข้อมูลเพื่อให้มีอันดับแบบฮีป ด้วยเมท็อด fixDown เมท็อดนี้รับ k แทนเลขที่ช่องที่มีการเปลี่ยนค่า ภายในทำงานเป็นวงวนทำตราบเท่าที่ตัวที่ช่อง k ยังมีลูก (ด้วยเงื่อนไขว่ามีลูกซ้าย ซึ่งก็คือเมื่อช่องที่ 2k+1 < size) ถ้ามี ให้ c เก็บเลขที่ช่องของลูกซ้าย จากนั้นบรรทัดที่ 39 อาจเพิ่ม c อีกหนึ่ง (ซึ่งหมายความ

ว่า ให้ c ไปเก็บเลขที่ช่องของลูกขวา) ก็เมื่อช่องที่ k มีลูกขวาและลูกขวาสำคัญกว่าลูกซ้าย เมื่อถึงบรรทัดที่ 40 จึงสรุปได้ว่า c เก็บเลขที่ช่องของลูกตัวมาก ถึงตอนนี้ถ้าที่ช่อง c สำคัญไม่มากกว่าที่ช่อง k ก็เลิกการทำงานเพราะถูกกฎแล้ว (เพราะลูกไม่สำคัญกว่าพ่อ) แต่ถ้าที่ลูกสำคัญกว่าพ่อก็สลับช่องที่ k กับ c (บรรทัดที่ 41) ตามด้วยให้ k=c เพราะ k คือช่องที่มีการเปลี่ยนค่า แล้ววนกลับขึ้นไปพิจารณาเพื่อสลับข้อมูลลงไปจนกว่าจะถูกกฎ โดยสรุปเมื่อมีข้อมูล n ตัว dequeue ลบข้อมูลที่รากแล้วดันข้อมูลลงตามความสูง เกิดการสลับข้อมูลไม่เกิน $\lfloor \log_2 (n-1) \rfloor$ จึงใช้เวลาเป็น $O(\log n)$

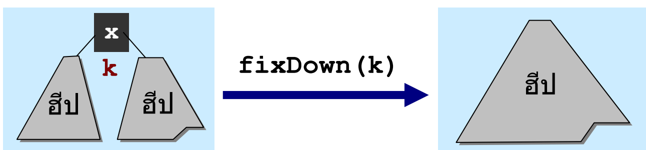
การสร้างฮีปจากข้อมูลในแถวลำดับ

ขอแถมบริการให้อีกตัวหนึ่ง บริการนี้คือตัวสร้างของ BinaryHeap ซึ่งรับพารามิเตอร์เป็นแถวลำดับของอ็อบเจกต์ มีหน้าที่นำอ็อบเจกต์ของทุกช่องไปสร้างฮีป รหัสที่ 8-10 แสดงวิธีสร้างอย่างง่ายด้วยการนำข้อมูลมาเพิ่มในฮีปให้ครบทุกตัว มาลองวิเคราะห์เวลาการทำงาน การเพิ่มข้อมูลตัวใหม่เข้าฮีปแล้วมีข้อมูล k ตัว จะเกิดการสลับข้อมูลไม่เกิน $\lfloor \log_2 k \rfloor$ ครั้ง (เพราะต้นไม้ได้ดุลที่มี k ตัวมีความสูง $\lfloor \log_2 k \rfloor$) ดังนั้นการเพิ่มตั้งแต่ตัวที่ 1 ถึง n จะเกิดการสลับข้อมูลไม่เกิน $\lfloor \log_2 1 \rfloor + \lfloor \log_2 2 \rfloor + \ldots + \lfloor \log_2 n \rfloor \le \sum_{k=1}^{n} \log_2 k = \log n! = O(n \log n)$ (จากตัวอย่างที่ 3-7 ในบทที่ 3)

```
public BinaryHeap(Object[] data) {
  for(int i=0; i<data.length; i++) enqueue(data[i]);
}
```

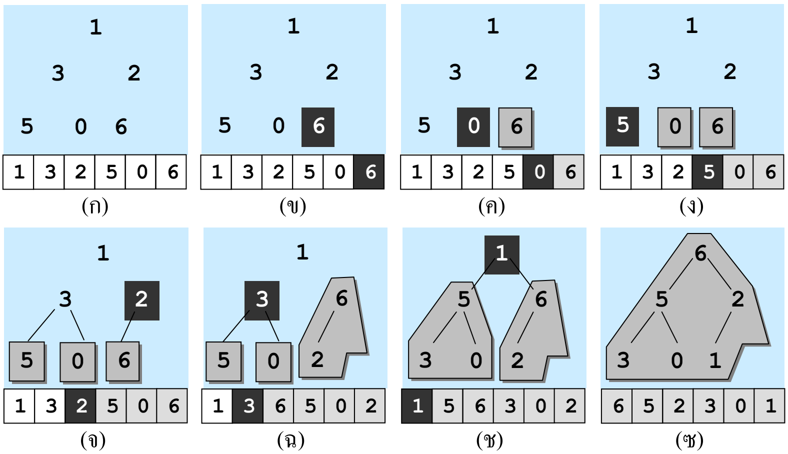
รหัสที่ 8-10 การสร้าง BinaryHeap จากข้อมูลในแถวลำดับด้วยการเพิ่มทีละตัว

ยังมีอีกวิธีหนึ่งในการสร้างฮีปจากข้อมูลในแถวลำดับขนาด n ตัว ซึ่งใช้เวลาการทำงานเป็น $O(n)$ วิธีนี้อาศัยเมท็อด fixDown ในการปรับข้อมูลในแถวลำดับจนกลายเป็นฮีป ถ้ายังจำได้ เราเรียก fixDown(k) ก็เพราะว่า ข้อมูลในช่องเลขที่ k เปลี่ยนค่าลดลง (ซึ่งต่างกับการเรียก fixUp(k) อันเนื่องมาจากข้อมูลในช่องเลขที่ k เปลี่ยนค่าเพิ่มขึ้น) อย่างในกรณีที่เราเรียก fixDown(0) ใน dequeue ก็เพราะเกิดการย้ายข้อมูลตัวท้าย (ที่มีค่าน้อย) มาแทนที่ช่อง 0 ทำให้อันดับแบบฮีปผิดไป โดยสรุป (ดูรูปที่ 8-5) fixDown(k) ทำหน้าที่ปรับต้นไม้ย่อยที่มีข้อมูลที่ช่อง k เป็นราก ให้มีอันดับแบบฮีป ภายใต้เงื่อนไขว่า ต้นไม้ย่อยทางซ้ายและทางขวาของข้อมูลที่ช่อง k ต้องมีอันดับแบบฮีปอย่างถูกต้องอยู่แล้ว



รูปที่ 8-5 fixDown(k) ปรับต้นไม้ที่ข้อมูลที่ช่อง k เป็นรากให้เป็นฮีป

ดังนั้นเราสามารถปรับข้อมูลในแถวลำดับให้เป็นฮีป ได้ด้วยการเรียก `fixDown` ตามลำดับ `fixDown(size-1)`, `fixDown(size-2)` ไปจนถึง `fixDown(0)` ซึ่งก็คือการปรับต้นไม้ย่อยขวาสุดระดับล่างสุด ย้อนกลับจากขวามาซ้าย และจากล่างขึ้นบน ทุกครั้งที่เรียก `fixDown` ที่ข้อมูลใด จึงประกันได้ว่า ต้นไม้ย่อยทางซ้ายและทางขวาต้องเป็นฮีปแน่ เพราะเราเรียก `fixDown` จากระดับล่างขึ้นบน ลูก ๆ จึงต้องถูกปรับให้เป็นฮีปก่อนจะปรับพ่อ ดังตัวอย่างในรูปที่ 8-6 ข้อมูลพื้นดำแทนการเรียก `fixDown` ณ ตำแหน่งของข้อมูลนั้น ต้นไม้พื้นเทาแทนต้นไม้ที่เป็นฮีปแล้ว ดูรูปไล่จาก (ก) ถึง (ซ) จะเห็นการปรับต้นไม้ย่อยจากขวาไปซ้ายจากล่างขึ้นบน จนในที่สุดได้ข้อมูลทั้งต้นมีอันดับแบบฮีป



รูปที่ 8-6 ตัวอย่างแสดงการใช้ `fixDown` เพื่อปรับแถวลำดับให้เป็นฮีป

รหัสที่ 8-11 แสดงการทำงานของตัวสร้างที่รับแถวลำดับมาสร้างเป็นฮีป เริ่มด้วยการให้ `elementData` ของฮีปอ้างอิงแถวลำดับตัวเดียวกับที่ได้รับ จากนั้นตั้งค่า `size` ให้เท่ากับจำนวนข้อมูล ซึ่งคือความยาวของแถว แล้วเริ่มเรียก `fixDown` ไล่ตั้งแต่ตัวขวาสุดกลับมายังตัวซ้ายสุด ทำเสร็จก็จะได้ข้อมูลในแถวลำดับเป็นฮีปตามต้องการ

```
  public BinaryHeap(Object[] data) {
    elementData = data;
    size = data.length;
    for (int k = size-1; k >= 0; k--) fixDown(k);
  }
  ...
```

เปลี่ยนเป็น size/2 – 1 ก็ได้นะ เร็วขึ้น ลองคิดดู

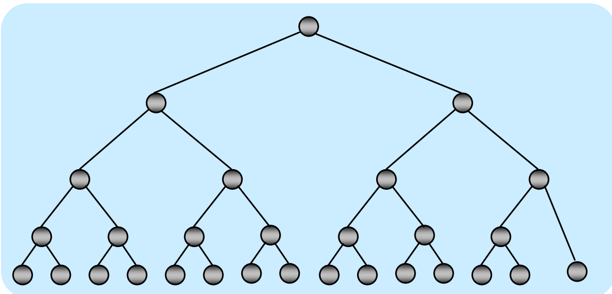
รหัสที่ 8-11 การสร้าง `BinaryHeap` จากข้อมูลในแถวลำดับ

การสร้างฮีปด้วยวิธีข้างบนนี้ใช้เวลาเท่าใด ฮีปแบบทวิภาคที่มีข้อมูล n ตัว มีความสูง $\lfloor \log_2 n \rfloor$ ให้สังเกตว่า ต้นไม้นี้จะมีต้นไม้ย่อยที่สูง h อยู่จำนวนไม่เกิน $\lceil n/2^{h+1} \rceil$ ต้น โดยที่ $h = 0, 1, \ldots, \lfloor \log_2 n \rfloor$

(ดูรูปที่ 8-7) `fixDown` แต่ละครั้งมีจำนวนการสลับข้อมูลแปรตามความสูงของต้นไม้ย่อย ดังนั้นการสร้างฮีปจากแถวลำดับมีการสลับข้อมูลเป็นจำนวนทั้งสิ้นเท่ากับ

$$\sum_{h=0}^{\lfloor \log_2 n \rfloor}\left(\left\lceil \frac{n}{2^{h+1}} \right\rceil h\right) \le n\sum_{h=0}^{\log_2 n}\left(\frac{h}{2^h}\right) < n\sum_{h=0}^{\infty}\left(\frac{h}{2^h}\right) = 2n = O(n)$$

โดย $\sum_{h=0}^{\infty}\left(\frac{h}{2^h}\right) = 2$ หาได้จาก $f(x) = \sum_{h=0}^{\infty} x^h = \frac{1}{1-x}$ หาอนุพันธ์ $f'(x) = \sum_{h=0}^{\infty} hx^{h-1} = \frac{1}{(1-x)^2}$ คูณด้วย x ได้ $xf'(x) = \sum_{h=0}^{\infty} hx^h = \frac{x}{(1-x)^2}$ แทน x ด้วย ½ จะได้ $\sum_{h=0}^{\infty} h\left(\frac{1}{2}\right)^h = \frac{1/2}{(1-1/2)^2} = 2$

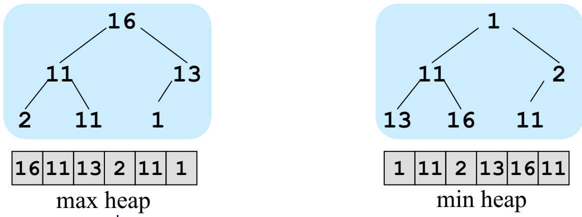


สูง	จำนวนต้น
4	1
3	2
2	4
1	7
0	15

รูปที่ 8-7 ฮีปแบบทวิภาคที่มีข้อมูล n ตัว มีต้นไม้ย่อยที่สูง h จำนวนไม่เกิน $\lceil n / 2^{h+1} \rceil$ ต้น

ฮีปมากสุดและฮีปน้อยสุด

ที่ได้ศึกษากันมาเรากำกับข้อมูลแต่ละตัวด้วยค่าความสำคัญ แล้วกำหนดให้ตัวที่สำคัญมากลัดแถวได้ ทำให้ต้องกำหนดอันดับแบบฮีปในลักษณะที่ว่า ความสำคัญของปมพ่อต้องไม่น้อยกว่าปมลูก หากเราขอไม่ใช้คำว่าความสำคัญ แต่ใช้ค่าของข้อมูลที่เก็บแทนการเปรียบเทียบเลย ก็จะตีความว่า ฮีปที่ได้ศึกษามาเป็นแบบที่รากเก็บข้อมูลที่มีค่ามากสุด รองรับการขอดูและลบค่ามากสุดได้เร็ว เราเรียกฮีปแบบนี้ว่า *ฮีปมากสุด* (max heap) แต่ถ้าเรากลับความคิด โดยตั้งกฎอันดับแบบฮีปเป็นแบบที่ปมพ่อต้องมีค่าไม่มากกว่าปมลูก ก็จะได้ว่า รากเก็บข้อมูลน้อยสุด ให้บริการขอดูและลบค่าน้อยสุดได้รวดเร็ว เรียกฮีปแบบนี้ว่า *ฮีปน้อยสุด* (min heap) (ดูตัวอย่างประกอบในรูปที่ 8-8) เมท็อด `fixUp` และ `fixDown` ที่เคยเรียก `greaterThan` ในการเปรียบเทียบข้อมูลของฮีปมากสุดก็ต้องเปลี่ยนไปเรียก `lessThan` สำหรับกรณีฮีปน้อยสุด



รูปที่ 8-8 ตัวอย่างฮีปมากสุดและฮีปน้อยสุด

อีกวิธีง่าย ๆ ในการสร้างฮีปน้อยสุด ทำได้โดยอาศัยการสร้างคลาสใหม่ที่เป็นคลาสลูกของ `BinaryHeap` แล้วเปลี่ยนเมท็อด `greaterThan` ให้มีความหมายว่า ข้อมูลค่าน้อยมีความสำคัญมาก ดังแสดงในรหัสที่ 8-12 ให้ความหมายว่าตัวที่ `i` สำคัญกว่าตัวที่ `j` เมื่อมีค่าน้อยกว่าตัวที่ `j`

```
public class BinaryMinHeap extends BinaryHeap {
  public BinaryMinHeap() {}
  public BinaryMinHeap(Object[] data) {super(data);}
  protected boolean greaterThan(int i, int j) {
    Comparable e = (Comparable) elementData[i];
    return e.compareTo(elementData[j]) < 0;
  }
}
```

ชื่อ greaterThan แต่เปรียบเทียบน้อยกว่า

รหัสที่ 8-12 การสร้าง min heap แบบง่าย

ตัวอย่างการใช้งานแถวคอยบุริมภาพ

การใช้งานแถวคอยบุริมภาพมีมากมาย ที่เห็นกันชัด ๆ ก็คืองานที่ต้องใช้แถวคอย แต่อนุญาตให้ลัดแถวได้ เช่น ระบบปฏิบัติการของเครื่องคอมพิวเตอร์จะต้องจัดลำดับการทำงานให้กับโปรแกรมต่าง ๆ ที่แบ่งกันใช้หน่วยประมวลผลตัวเดียวกัน ระบบปฏิบัติการจะเปลี่ยนให้แต่ละโปรแกรมทำงานกันคนละนิดละหน่อย จนผู้ใช้เครื่องคอมพิวเตอร์รู้สึกว่า เขาสามารถใช้งานโปรแกรมหลาย ๆ ตัวได้พร้อม ๆ กัน ซึ่งก็เห็นชัดว่า ต้องใช้แถวคอยจัดเก็บงานต่าง ๆ แต่เนื่องจากความสำคัญของงานไม่เท่ากัน เช่น ถ้าเป็นเครื่องให้บริการข้อมูลกับเครือข่าย ก็ต้องให้ความสำคัญกับโปรแกรมติดต่อกับเครือข่ายหรือฐานข้อมูลมากกว่าโปรแกรมที่แสดงผลภาพกราฟิก โปรแกรมที่สำคัญกว่าจึงมีสิทธิ์ลัดแถวได้สิทธิ์ทำงานก่อน

นอกจากการใช้แถวคอยบุริมภาพเพื่องานที่ใช้แถวคอยแบบลัดแถวได้แล้ว ยังมีงานอื่นอีกที่เหมาะกับแถวคอยแบบนี้ หัวข้อนี้นำเสนอตัวอย่างการประยุกต์ ได้แก่ การเรียงลำดับแบบฮีป (heap sort) การเลือกข้อมูลตามอันดับ (selection problem) การค้นในปริภูมิสถานะตามต้นทุนน้อยสุด (least-cost search) และตัวจำลองวงจรตรรก

การเรียงลำดับแบบฮีป

จะว่าไปแล้วฮีปแบบทวิภาคถูกออกแบบขึ้นมาครั้งแรกไม่ได้เพื่อใช้เป็นแถวคอยบุริมภาพ แต่เพื่อเป็นโครงสร้างข้อมูลหลักของ*การเรียงลำดับแบบฮีป* (heap sort) ที่ใช้เวลาเป็น $O(n \log n)$ เมื่อ n คือจำนวนข้อมูล การเรียงลำดับแบบฮีปอาศัยการสร้างฮีปมากสุดจากชุดข้อมูลในแถวลำดับ จากนั้นเข้าวงวนลบตัวมากสุดทีละตัว ๆ เพื่อนำไปเก็บจากตำแหน่งขวาไล่มาซ้าย ดังแสดงในรหัสที่ 8-13 การสร้างฮีปจากแถวลำดับใช้เวลา $O(n)$ ในขณะที่วงวนหมุนจำนวน n รอบ แต่ละรอบเรียก `dequeue` ใช้เวลา $O(\log n)$ รวมเวลาทั้งหมดเป็น $O(n) + O(n \log n)$ ซึ่งคือ $O(n \log n)$ อันเป็นเวลาเชิงโอใหญ่ที่ดีที่สุดของวิธีเรียงลำดับที่อาศัยการเปรียบเทียบเป็นหลัก (จะนำเสนอการเรียงลำดับวิธีต่าง ๆ ในบทที่ 13)

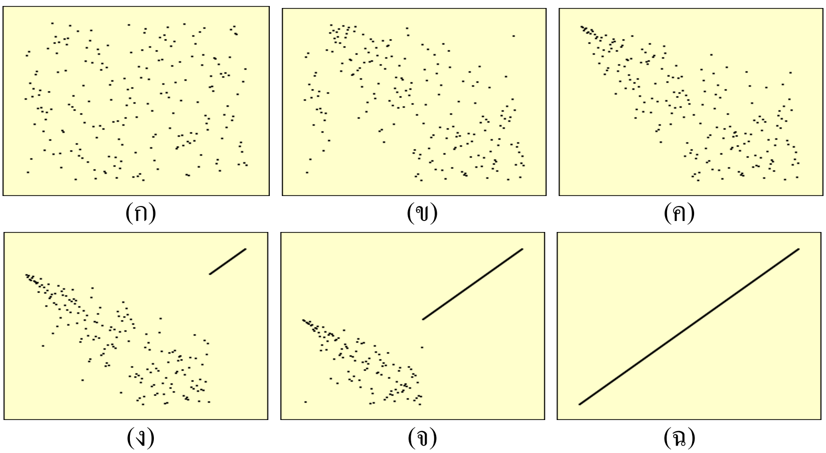
```
  public static void heapSort(Object[] data) {
    BinaryHeap h = new BinaryHeap(data);
    for (int k = h.size() - 1; k > 0; k--)
      data[k] = h.dequeue();
  }
```

สร้างฮีปมากสุดจากอาเรย์

นำข้อมูลมากไปน้อยเก็บที่ช่องขวามาซ้าย

รหัสที่ 8-13 การเรียงลำดับแบบฮีป

รูปที่ 8-9 แสดงการจินตทัศน์ข้อมูลในแถวลำดับระหว่างการเรียงลำดับแบบฮีป โดยข้อมูลเริ่มต้นมีลักษณะสุ่ม จุดในภาพหนึ่งจุดแทนข้อมูลหนึ่งตัว พิกัด x คือตำแหน่งของข้อมูลในแถวลำดับ และพิกัด y คือค่าของข้อมูล ค่ามากอยู่สูง ค่าน้อยอยู่ต่ำ ดังนั้นเมื่อข้อมูลเรียงจากน้อยไปมากจะได้จุดทั้งหลายเรียงกันเป็นเส้นทแยงมุมจากซ้ายล่างไปขวาบน เริ่มจากข้อมูลเรียงแบบสุ่มในรูป (ก) กำลังสร้างฮีปในรูป (ข) สร้างฮีปเสร็จได้รูป (ค) ให้สังเกตว่า จุดซ้ายสุด (แทนข้อมูลในช่องที่ 0) อยู่สูงสุด จากนั้นข้อมูลมากสุดแต่ละตัวจะถูกลบไปอยู่ด้านท้ายของแถวลำดับ ทำให้เห็นเป็นแนวทแยงยาวขึ้นเรื่อย ๆ จากมุมขวาบน แสดงให้ดูสองช่วงในรูป (ง) และ (จ) จนทำเสร็จในรูป (ฉ)



รูปที่ 8-9 ภาพระหว่างการเรียงลำดับข้อมูลแบบสุ่มด้วย heap sort

การเลือกข้อมูลตามอันดับ

ปัญหามีอยู่ว่า ข้อมูลที่น้อยสุดเป็นอันดับที่ k ของชุดข้อมูลที่เก็บในแถวลำดับคือตัวใด ? วิธีหาก็ไม่ยาก เพียงแต่นำแถวลำดับไปเรียงลำดับ แล้วก็หยิบข้อมูลในช่องที่ $k - 1$ ก็ได้คำตอบ วิธีนี้ใช้เวลาเรียงลำดับ (สมมติว่าใช้การเรียงลำดับแบบฮีป) เป็น $O(n \log n)$ บวกเวลาการหยิบข้อมูลซึ่งใช้เวลาคงตัว จึงรวมเป็น $O(n \log n)$ แต่ถ้ามาคิดดูดี ๆ ถ้าเราต้องการตัวน้อยที่สุด (ซึ่งคือตัวน้อยสุดอันดับที่ 1) เราสามารถหาได้ด้วยการวิ่งไล่เปรียบเทียบในแถวลำดับเป็นจำนวน $n - 1$ ครั้งก็ย่อมได้ตัวน้อยที่สุด ซึ่งก็เป็นกระบวนการคล้ายกับการหาตัวมากที่สุด (ซึ่งคือตัวน้อยสุดอันดับที่ n) ซึ่งใช้เวลาเป็น $O(n)$ เช่นกัน แล้วการหาตัวน้อยสุดอันดับที่ k ทำไมกลับต้องใช้เวลาเป็น $O(n \log n)$ ด้วยเล่า ? น่าจะมีวิธีอื่นที่ดีกว่าการเรียงลำดับข้อมูล

การเรียงลำดับให้ข้อมูลในแถวลำดับมีระเบียบเรียงจากน้อยไปมาก แล้วหยิบข้อมูลเพียงตัวเดียวเพื่อเป็นผลลัพธ์ คงให้ความรู้สึกว่า ไม่ค่อยคุ้มเท่าไหร่ มาลองหาวิธีใหม่ที่ใช้ฮีปช่วย เริ่มด้วยการสร้างฮีปน้อยสุดจากชุดข้อมูลที่ได้รับ จากนั้นลบตัวน้อยสุดจากฮีปออกมา k ครั้ง การลบครั้งที่ k ก็ย่อมได้ข้อมูลตัวน้อยสุดอันดับที่ k (ดูรหัสที่ 8-14) การสร้างฮีปจากแถวลำดับใช้เวลา $O(n)$ ลบจากฮีป k ครั้งใช้ $O(k \log n)$ รวมเวลาทั้งหมดเป็น $O(n) + O(k \log n) = O(n + k \log n)$

```
public static Object select(Object[] data, int k) {
  BinaryMinHeap h = new BinaryMinHeap(data);
  Object x = null;
  for (int i=0; i<k; i++) x = h.dequeue();
  return x;
}
```

สร้างฮีปน้อยสุดจากอาเรย์

ลบออก k ครั้ง

ตัวสุดท้ายคือตัวน้อยสุดอันดับ k

รหัสที่ 8-14 การเลือกข้อมูลน้อยสุดอันดับที่ k ในแถวลำดับ data

บางครั้งข้อมูลที่เราได้รับมา ไม่ได้มาในรูปของแถวลำดับ เช่น อาจจะได้มาจากแฟ้มข้อมูล หรือมาจากอ็อบเจกต์ประเภทที่เรียกกันว่า *ตัวแจงย้ำ* (iterator) โดยทั่วไปตัวแจงย้ำมีหน้าที่ให้บริการแจกแจงข้อมูล (ที่เก็บไว้ หรือผลิตเอง หรือด้วยวิธีใดก็แล้วแต่) ออกมาให้ใช้ทีละตัว เพื่อนำไปประมวลผลย้ำ ๆ ด้วยกระบวนการเดียวกัน เช่น แจกแจงข้อมูลทั้งหมดเพื่อนำไปแสดงออกจอภาพเป็นต้น ในจาวา ตัวแจงย้ำคืออ็อบเจกต์ของคลาสที่ implements อินเทอร์เฟซ `Iterator` ซึ่งมีบริการ `next` และ `hasNext` ให้เรียกใช้ (ความจริงยังบังคับเมท็อด `remove` ด้วย แต่จะไม่ขอกล่าวถึงในที่นี้) เมท็อด `next` มีหน้าที่ผลิตข้อมูลตัวถัดไป ส่วน `hasNext` คืนสถานะว่า ยังมีข้อมูลเหลือให้ผลิตด้วย `next` หรือไม่ รหัสที่ 8-15 แสดงตัวอย่างการใช้ตัวแจงย้ำเพื่อแจกแจงข้อมูลทุกตัวแสดงออกจอภาพจนหมด (จะได้นำเสนอรายละเอียดการเขียนตัวแจงย้ำให้กับที่เก็บข้อมูลในบทที่ 12)

```
public static void printAll(Iterator itr) {
  while(itr.hasNext())
    System.out.println(itr.next());
}
```

itr.hasNext() = ยังมีข้อมูล ?
itr.next() = คืนข้อมูลตัวถัดไป

รหัสที่ 8-15 ตัวอย่างการแจกแจงข้อมูลที่ผลิตจากอ็อบเจกต์แบบ `Iterator`

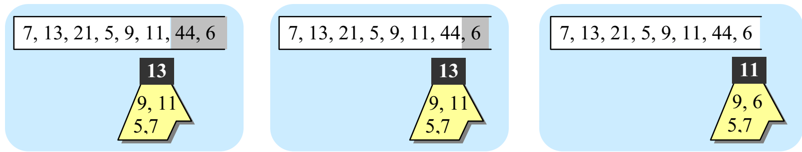
หากเราต้องเลือกข้อมูลตัวน้อยที่สุดอันดับ k จากข้อมูลทั้งหมดที่ผลิตจากตัวแจงย้ำด้วยวิธีที่ได้กล่าวมา ก็คงต้องจองแถวลำดับขนาดเท่ากับจำนวนข้อมูล แจงข้อมูลออกมาเก็บในแถวให้ครบ แล้วค่อยนำไปสร้างฮีป ตามด้วยวงวนลบตัวน้อยสุดจากฮีป k ครั้ง ดังแสดงในรหัสที่ 8-16 ถ้าการผลิตข้อมูลจากตัวแจกย้ำแต่ละตัวใช้เวลา $\Theta(1)$ วิธีดังกล่าวยังคงใช้เวลา $O(n + k \log n)$ แต่มีข้อเสียที่ต้องใช้แถวลำดับและฮีปขนาด n ช่อง ถ้า $n = 10^6$, $k = 100$ ก็ต้องเสียเนื้อที่มาก

```
public static Object select(Iterator itr, int n, int k) {
  Object[] data = new Object[n];
  for(int i=0; itr.hasNext(); i++) data[i] = itr.next();
  BinaryMinHeap h = new BinaryMinHeap(data);
  Object x = null;
  for (int i=0; i<k; i++) x = h.dequeue();
  return x;
}
```

อ่านทุกตัวมาเก็บในอาเรย์

รหัสที่ 8-16 การเลือกข้อมูลน้อยสุดอันดับที่ k จากข้อมูลที่แจกแจงจากตัวแจงย้ำ

เราสามารถปรับปรุงการใช้ฮีปเพื่อหาตัวน้อยสุดอันดับที่ k โดยใช้ฮีปขนาด k และไม่ต้องใช้แถวลำดับเพื่อเก็บข้อมูลที่แจกแจงเพิ่มเสริมแต่อย่างใด ในรหัสที่ 8-16 เราใช้ฮีปน้อยสุดขนาด n ช่อง เพื่อหาค่าน้อยสุดอันดับที่ k แต่วิธีใหม่นี้เราใช้ฮีปมากสุดขนาด k ผู้อ่านอาจแปลกใจว่า จะหาค่าน้อยแต่กลับใช้ฮีปมากสุด หลักการทำงานคือ "การใช้ฮีปมากสุดขนาด k เพื่อเก็บตัวน้อยสุด k ตัวจากข้อมูลทั้งหมดที่ได้ผลิตมา" ณ ขณะใดขณะหนึ่งรากของฮีปก็คือตัวมากสุดของข้อมูลน้อยสุด k ตัว เมื่อผลิตข้อมูลจนครบ รากของฮีปก็ต้องเป็นตัวน้อยสุดอันดับที่ k รูปที่ 8-10 แสดงตัวอย่างการใช้ฮีปมากสุดขนาด 5 ตัวเพื่อเก็บตัวน้อยสุด 5 ตัวของข้อมูลที่ได้รับมา รายการของตัวเลขในรูปที่มีพื้นสีขาวคือชุดข้อมูลที่ได้ผลิตมาพิจารณาแล้ว ส่วนที่มีพื้นสีเทาคือชุดที่ยังไม่ได้ผลิตออกมา รูป (ก) แสดงค่าในฮีปประกอบด้วย 13, 7, 11, 9, และ 5 ซึ่งเป็นข้อมูล 5 ตัวน้อยสุดจากชุดข้อมูล 7, 13, 21, 5, 9 และ 11 สังเกตว่า รากของฮีปคือตัวมากสุดในกลุ่มตัวน้อย เมื่ออ่านข้อมูลตัวถัดไปในรูป (ข) ได้ 44 มีค่ามากกว่ารากของฮีป ก็ไม่ต้องสนใจอะไร พออ่านตัวถัดไปได้ 6 มีค่าน้อยกว่า 13 รากของฮีป จะลบรากทิ้ง แล้วเพิ่ม 6 ใส่ในฮีปได้ดังรูป (ค) ถึงตอนนี้ข้อมูลหมดแล้ว จะได้ 11 ซึ่งคือรากของฮีป เป็นตัวมากสุดในกลุ่มตัวน้อยสุด 5 ตัว สรุปได้ว่า 11 คือตัวน้อยสุดอันดับที่ 5



(ก) (ข) (ค)

รูปที่ 8-10 ฮีปมากสุดขนาด 5 เก็บตัวน้อยสุด 5 ตัวของข้อมูลที่ได้รับมา

รหัสที่ 8-17 แสดงวิธีการนี้ เริ่มด้วยการสร้างฮีปมากสุด จากนั้นให้ตัวแจงย้ำผลิตข้อมูลแล้วเพิ่มเข้าฮีป (บรรทัดที่ 2 ถึง 4) แล้วเข้าวงวนผลิตข้อมูลมาเพิ่มใส่ฮีป แต่การเพิ่มใส่ฮีปมีเงื่อนไขว่า จะทำก็เมื่อข้อมูลที่ได้มาใหม่มีค่าน้อยกว่าข้อมูลที่รากของฮีป ณ ขณะนั้น (บรรทัดที่ 7) และก่อนจะเพิ่มต้องลบตัวมากสุดในฮีปออกก่อน วนทำงานจนข้อมูลหมด ก็คืนข้อมูลที่อยู่ที่รากของฮีปกลับไปเป็นตัวน้อยสุดอันดับที่ k ในกรณีช้าสุดจะเกิดการ `enqueue` ที่บรรทัดที่ 9 ทุกครั้ง สรุปว่า ต้องเพิ่มใส่ฮีปขนาด k จำนวน n ครั้ง ใช้เวลารวมเป็น $O(n \log k)$ โดยใช้ฮีปขนาด k เป็นเนื้อที่เสริม

```
01   public static Object select(Iterator itr, int k) {
02     BinaryHeap h = new BinaryHeap();
03     for (int i = 0; i < k && itr.hasNext(); i++)
04       h.enqueue(itr.next());
05     while (itr.hasNext()) {
06       Comparable x = (Comparable) itr.next();
07       if (x.compareTo(h.peek()) < 0) {
08         h.dequeue();
09         h.enqueue(x);
10       }
11     }
12     return h.peek();
13   }
```

นี่คือฮีปมากสุด

เพิ่มอย่างมาก k ตัว

ถ้าตัวใหม่น้อยกว่าตัวมากสุดในฮีป ให้ลบตัวมากสุดนั้นและเพิ่มตัวใหม่

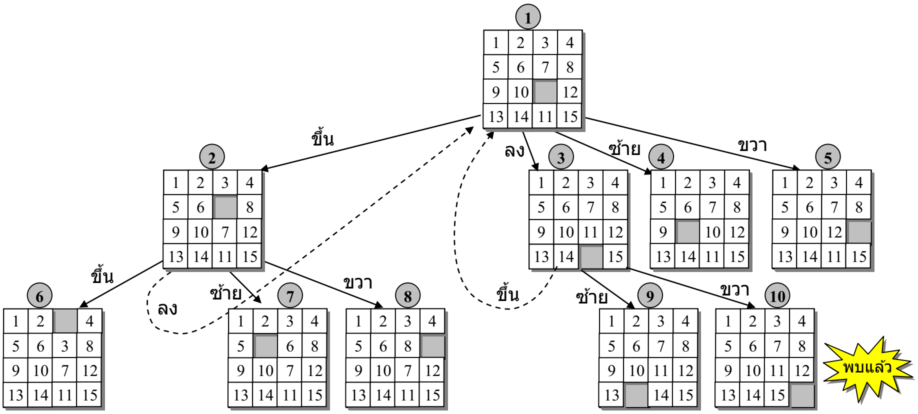
รหัสที่ 8-17 การเลือกข้อมูลน้อยสุดอันดับที่ k ด้วยฮีปมากสุดขนาด k

การค้นตามต้นทุนน้อยสุด

เราได้นำเสนอการใช้แถวคอยในการแก้ไขปริศนา 15 กันในบทที่ 1 ซึ่งเป็นการค้นตามแนวกว้าง โดยใช้แถวคอยจัดเก็บและจัดลำดับตารางต่าง ๆ ในลักษณะที่ตารางใดเกิดก่อน ก็จะถูกนำไปผลิตตารางใหม่ก่อน รูปที่ 8-11 แสดงตัวอย่างการค้นตามแนวกว้างโดยตัวเลขข้างบนตารางแสดงลำดับที่ตารางถูกลบออกจากแถวคอย

การค้นตามแนวกว้างมีระเบียบดี แต่ช้า ตารางถูกผลิตไปทีละระดับ เนื่องจากตารางในแต่ละระดับมีมากขึ้น ๆ ย่อมกินที่ในแถวคอยมากขึ้น ๆ และทำงานช้าลง ๆ เราสามารถค้นให้เร็วขึ้นและใช้เนื้อที่น้อยลงได้ ด้วยการค้นอีกแบบหนึ่งที่เรียกว่า *การค้นตามต้นทุนน้อยสุด* (least-cost search) กำหนดให้ตาราง a มีต้นทุนน้อยกว่าตาราง b ก็เมื่อ a เป็นตารางที่ "น่าจะ" นำไปสู่เป้าหมายได้ "ง่าย

กว่า" ตาราง b ดังนั้นระหว่างการค้นคำตอบ แทนที่จะเลือกตารางในลำดับแบบ "เกิดก่อน ถูกเลือกก่อน" ก็ควรเป็น "เลือกตารางที่มีต้นทุนน้อยสุด" มาผลิตตารางใหม่ เพราะตารางที่มีต้นทุนน้อยสุดย่อมหมายถึงตารางที่น่าจะนำไปสู่เป้าหมายได้ง่ายสุดในบรรดาตารางที่เก็บอยู่ จึงควรใช้แถวคอยบุริมภาพเก็บตารางแทนแถวคอยธรรมดาที่เคยใช้ และกำหนดให้ตารางที่มีต้นทุนยิ่งน้อยยิ่งสำคัญ (นั่นคือการสร้างแถวคอยบุริมภาพด้วยฮีปน้อยสุด)



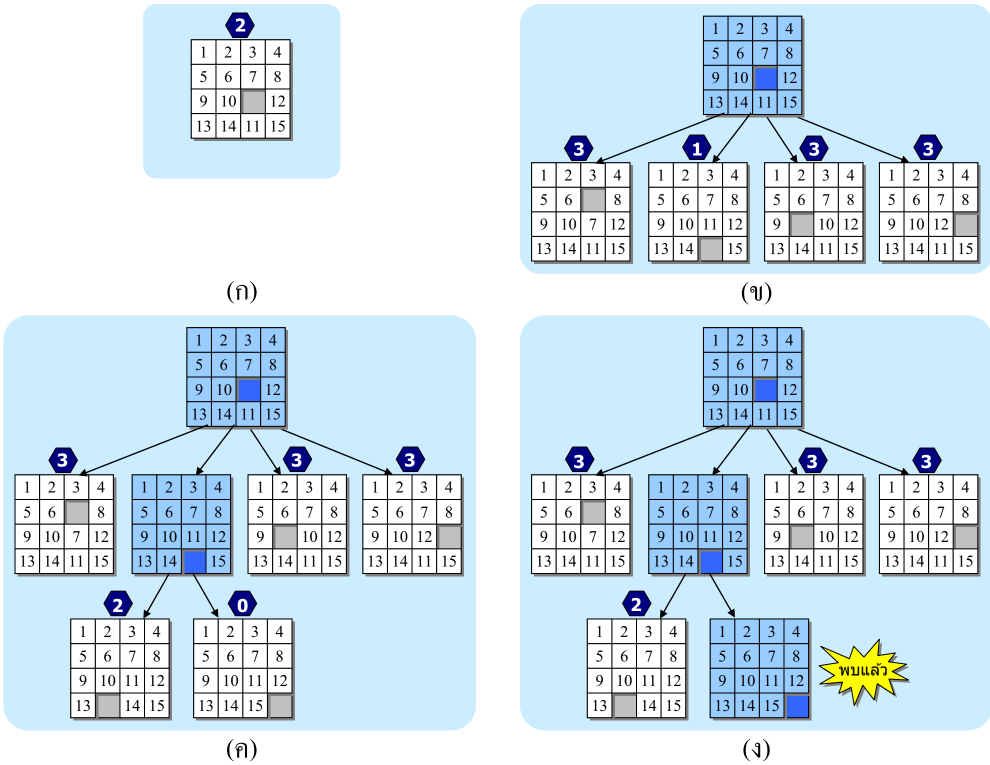
รูปที่ 8-11 ลำดับการผลิตตารางในการค้นตามแนวกว้างของปริศนา 15

ดูตัวอย่างในรูปที่ 8-12 ตัวเลขในรูปหกเหลี่ยมด้านบนตารางแทนต้นทุนของตาราง (ขอยังไม่บอกตอนนี้ว่า ต้นทุนเหล่านี้หามาได้อย่างไร) ตารางสีขาวคือตารางที่เก็บในฮีป ส่วนตารางสีเข้มคือตารางที่ถูกลบออกจากฮีป เริ่มการทำงานด้วยการนำตารางเริ่มต้นเพิ่มใส่ฮีป แล้วเข้าวงวนเพื่อลบตารางที่มีต้นทุนน้อยสุด นำไปเลื่อนช่องว่างทั้งสี่ทิศได้ตารางใหม่ ๆ เพิ่มใส่ฮีป เริ่มที่รูป (ก) ใส่ตารางเริ่มต้นในฮีป จากนั้นในรูป (ข) ลบตารางจากฮีปเพื่อผลิตอีกสี่ตาราง ได้ตารางที่มีต้นทุน 1 เป็นตารางที่มีต้นทุนน้อยสุด จึงถูกลบออกมาผลิตตารางใหม่ได้อีกสองตารางดังรูป (ค) ถึงตอนนี้มีตารางในฮีปอยู่ห้าตาราง ก็เลือกตารางที่มีต้นทุนน้อยสุดซึ่งมีค่าเป็น 0 ได้ดังรูป (ง) ซึ่งปรากฏว่า เป็นตารางเป้าหมายที่ต้องการ เป็นอันสิ้นสุดการค้น การเลือกตารางที่มีต้นทุนน้อยสุด จึงเป็นการตัดสินใจที่ดี นำไปสู่เป้าหมายได้รวดเร็วกว่าการค้นตามแนวกว้าง

คำถามที่คาใจคือต้นทุนของตารางหามาได้อย่างไร ? คำตอบก็คือหาอย่างไรก็ได้ที่ตรงตามความประสงค์ที่บอกไว้ตอนต้นคือ ตารางที่มีต้นทุนน้อยน่าจะนำไปสู่เป้าหมายได้เร็วกว่าตารางที่มีต้นทุนมาก ตัวอย่างในรูปที่ 8-12 นั้นเรากำหนดให้ต้นทุนของตารางคือ “จำนวนหมายเลขในตารางที่อยู่ผิดตำแหน่ง” เช่น ตารางเริ่มต้นในรูป (ก) มีหมายเลข 11 และ 15 อยู่ผิดตำแหน่งจึงมีต้นทุนเป็น 2 ด้วยวิธีนี้ตารางที่มีต้นทุนเป็น 0 ก็ย่อมเป็นเป้าหมายที่ต้องการเพราะไม่มีหมายเลขใดอยู่ผิดตำแหน่งเลย

แต่ก็ต้องบอกก่อนว่า การกำหนดต้นทุนแบบนี้ก็ยังไม่ค่อยดี เช่น ในรูปที่ 8-13 ตารางทั้งสองมีจำนวนหมายเลขที่อยู่ผิดตำแหน่งเท่ากันคือ 7 แต่ดูด้วยตาก็รู้ว่า ตาราง (ก) เลื่อนยากกว่าตาราง (ข)

วิธีกำหนดต้นทุนให้ตารางอย่างง่าย ๆ อีกวิธีหนึ่งที่ได้ผลดีกว่า (ผลดีกว่าคือโดยทั่วไปมักนำไปสู่เป้าหมายได้เร็วกว่า) คือคำนวณผลรวมระยะทางฉากของแต่ละหมายเลขจากตำแหน่งที่อยู่ในตารางกับตำแหน่งที่ต้องอยู่ (ระยะทางฉากของหมายเลข *k* ก็คือจำนวนครั้งในการเลื่อน *k* จากตำแหน่งที่อยู่ในตารางไปยังตำแหน่งที่ต้องอยู่ โดยสมมติว่า ไม่มีหมายเลขอื่นใดขวางอยู่เลย) เช่น หมายเลข 13 ของตาราง (ก) ในรูปที่ 8-13 มีระยะทางฉากที่กล่าวถึงเป็น 4 ดังนั้นตาราง (ก) มีระยะทางฉากของหมายเลข 9, 10, 11, 12, 13, 14, และ 15 เป็น 3, 1, 2, 2, 4, 2, และ 2 ตามลำดับ จึงมีต้นทุนเป็น 16 ในขณะที่ต้นทุนของตาราง (ข) มีค่าเป็น 8



รูปที่ 8-12 การค้นตามต้นทุนน้อยสุดของปริศนา 15

รหัสที่ 8-18 แสดงโปรแกรมที่เคยเขียนไว้คร่าว ๆ ในบทที่ 1 ซึ่งใช้แถวคอย เราเพียงแต่เปลี่ยนบรรทัดที่สร้าง `ArrayQueue` มาเป็นการสร้าง `BinaryMinHeap` และปรับปรุงให้คลาส `PuzzleBoard` ที่แทนตารางให้เป็น `Comparable` และเพิ่มเมท็อด `compareTo` เพื่อเปรียบเทียบต้นทุนของตารางดังแสดงในรหัสที่ 8-19

1	2	3	4
5	6	7	8
14	12	10	13
15	11	9	

(ก)

1	2	3	4
5	6	7	8
13	9	10	11
14	15	12	

(ข)

รูปที่ 8-13 ตาราง (ก) เลื่อนยากกว่าตาราง (ข)

```
public static PuzzleBoard solve(PuzzleBoard b) {
  Set c = new ArraySet();
  Queue queue = new ArrayQueue();
  PriorityQueue queue = new BinaryMinHeap();
  queue.enqueue(b);  c.add(b);
  while ( !queue.isEmpty() ) {
    b = queue.dequeue();
    for (int d = 0; d < 4; d++) {
      PuzzleBoard b2 = b.moveBlank(d);
      if (b2 != null) {
        if (b2.isAnswer()) return b2;
        if (!c.contains(b2)) {queue.enqueue(b2); c.add(b2);}
      }
    }
  }
  return null;
}
```

ไม่ใช้แถวคอย หันมาใช้ฮีปน้อยสุดแทน

รหัสที่ 8-18 การปรับปรุงโปรแกรมแก้ปริศนา 15 จากที่ใช้แถวคอยเป็นฮีปน้อยสุด

```
class PuzzleBoard implements Comparable {
  private byte table[][];
  private byte rowB, colB;
  private PuzzleBoard prev;

  public int compareTo(Object obj) {
    PuzzleBoard that = (PuzzleBoard) obj;
    return this.cost() - that.cost();
  }
  public int cost() {
    int cost = 0, n = table.length;
    for (int r = 0; r < n; r++)
      for (int c = 0; c < n; c++)
        if (table[r][c] != BLANK)
          cost += Math.abs((table[r][c] - 1) / n - r) +
                  Math.abs((table[r][c] - 1) % n - c);
    return cost;
  }
  ...
```

ให้ตารางเปรียบเทียบได้

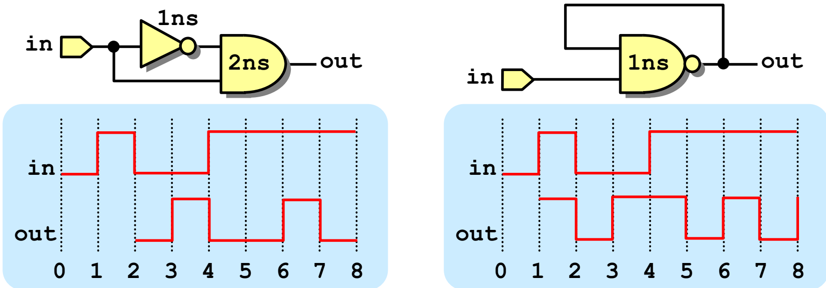
เปรียบเทียบต้นทุน

หาผลรวมระยะทางฉากของแต่ละหมายเลขจากตำแหน่งที่อยู่ในตารางกับตำแหน่งที่ต้องอยู่

รหัสที่ 8-19 การปรับปรุงตารางเป็น Comparable เพื่อใช้เปรียบเทียบต้นทุน

ตัวจำลองวงจรตรรก

ขอปิดท้ายบทนี้ด้วยตัวอย่างการสร้าง*ตัวจำลองวงจรตรรก* (logic circuit simulator) โดยใช้กลวิธี*การจำลองตามเหตุการณ์* (event-driven simulation) ที่ใช้แถวคอยบุริมภาพเพื่อเก็บเหตุการณ์การเปลี่ยนแปลงสัญญาณในวงจร แต่ละเหตุการณ์ถูกกำกับด้วยเวลาที่สัญญาณนั้นจะเกิด โดยใช้เวลานี้เป็นตัวกำหนดความสำคัญของเหตุการณ์ เนื่องจากเหตุการณ์ต่าง ๆ ถูกผลิตออกมาไม่ได้เรียงตามเวลาที่สัญญาณนั้นเกิด แต่ตัวจำลองวงจรต้องนำเหตุการณ์ที่สัญญาณเกิดก่อนไปจำลองก่อนให้เรียงตามเวลา ดังนั้นจึงต้องใช้ฮีปน้อยสุดในการสร้างแถวคอยบุริมภาพ



รูปที่ 8-14 ตัวอย่างการจำลองวงจรตรรก

ตัวจำลองที่จะเขียนนี้รองรับวงจรเชิงผสมรวมทั้งแบบวงวน โดยแต่ละเกตมีค่าหน่วงเวลากำกับดังตัวอย่างในรูปที่ 8-14 ตัวจำลองนี้ประกอบด้วยคลาสหลักคือคลาส `Value`, `Gate` และ `Event` เริ่มที่คลาส `Value` ก่อน คลาสนี้มีไว้ใช้แทนค่าของสัญญาณต่าง ๆ ในการจำลอง ขอใช้แบบง่ายสุดคือมีเพียงสองระดับ 0 และ 1 (แบบที่ละเอียดขึ้นจะมีหลายระดับสัญญาณระหว่าง 0 ถึง 1) คำถามที่น่าสนใจคือ ถ้ามีเพียงสองระดับ ทำไมไม่ใช้ `boolean` เลย ทำไมต้องสร้างเป็นคลาสด้วย ต้องขอบอกว่า มีสองระดับสัญญาณ แต่มีอีกหนึ่งสภาวะคือสภาวะที่ยังไม่รู้ระดับสัญญาณ ซึ่งจะเห็นได้ชัดคือช่วงเริ่มต้นของการจำลอง เราต้องกำหนดให้ทุก ๆ สัญญาณในวงจรอยู่ในสภาวะไม่รู้ระดับ แล้วปล่อยให้ระบบค่อย ๆ ตั้งค่าเข้าสู่สภาวะที่ควรเป็นจากการเปลี่ยนแปลงที่เกิดขึ้นที่ช่องสัญญาณขาเข้า ซึ่งจะได้เห็นจริงต่อไป

รหัสที่ 8-20 แสดงรายละเอียดของ `Value` ภายในมีการนิยามอ็อบเจกต์ที่เป็นค่าคงตัวไว้ใช้งานสามค่าคือ `ONE`, `ZERO` และ `UNDEFINED` แทนสภาวะทั้งสาม โดยให้บริการการนอต แอนด์ ออร์ และอื่น ๆ ที่รองรับสภาวะ `UNDEFINED` ด้วย เช่น การแอนด์ `ZERO` กับ `UNDEFINED` ย่อมได้ `ZERO` (บรรทัดที่ 11) ในขณะที่การแอนด์ `ONE` กับ `UNDEFINED` ย่อมได้ `UNDEFINED` เพราะผลลัพธ์มีสิทธิ์เป็นได้หลายแบบ (บรรทัดที่ 13)

```
public class Value {
  public static final Value ONE = new Value();
  public static final Value ZERO = new Value();
  public static final Value UNDEFINED = new Value();
  private Value() {}
  public Value not() {
    if (this == UNDEFINED) return UNDEFINED;
    else return this == ONE ? ZERO : ONE;
  }
  public Value and(Value v) {
    if (this == ZERO || v == ZERO) return ZERO;
    else if (this == ONE && v == ONE) return ONE;
    else return UNDEFINED;
  }
  public Value or(Value v) {
    if (this == ONE || v == ONE) return ONE;
    else if (this == ZERO && v == ZERO) return ZERO;
    else return UNDEFINED;
  }
  public String toString() {
    return this == ONE ? "1" : (this == ZERO ? "0" : "?");
  }
  ...
```

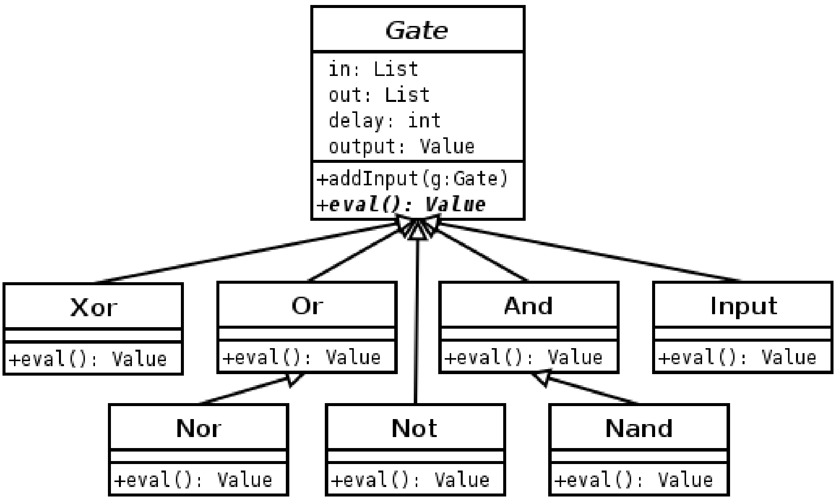
คลาสนี้มีแค่ 3 อ็อบเจกต์ให้คนอื่นใช้ร่วมกัน

toString เอาไว้ใช้ตอน print

เขียนการดำเนินการแบบอื่นเอง (เช่น xor)

รหัสที่ 8-20 คลาส Value ใช้แทนระดับสัญญาณ ONE, ZERO และ UNDEFINED

คลาส Gate เป็นคลาสแม่ของเกตต่าง ๆ หลากหลายประเภท เช่น And, Or, Not, Nand, ... แสดงด้วยแผนภาพคลาสดังรูปที่ 8-15



รูปที่ 8-15 Gate เป็นคลาสแม่เพื่อสร้างเกตประเภทต่างๆ

```
01 public abstract class Gate {
02   List in = new ArrayList();
03   List out = new ArrayList();
04   int delay = 0;
05   Value output = Value.UNDEFINED;
06   protected Gate(int delay) { this.delay = delay; }
07   public final void addInput(Gate g) {
08     in.add(g);
09     g.out.add(this);
10   }
11   public abstract Value eval();
12 }
```

รายการของเกตขาเข้า (in) และเกตขาออก (out)

เริ่มต้นยังไม่รู้ระดับสัญญาณ

เพิ่ม g ใน in ของเกตเรา และเพิ่มเกตเราใน out ของ g

คลาสลูกต้อง override eval()

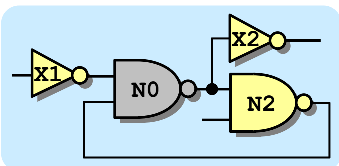
```
01 public class And extends Gate {
02   public And(int d) { super(d); }
03   public Value eval() {
04     Value result = Value.ONE;
05     for(int k=0; k<in.size(); k++)
06       result = result.and(((Gate)in.get(k)).output);
07     return result;
08   }
09 }
```

นำสัญญาณของขาเข้าทั้งหมดมา and กัน

```
01 public class Nand extends And {
02   public Nand(int d) { super(d); }
03   public Value eval() { return super.eval().not(); }
04 }
```

เรียกของ And แล้วมา not

รหัสที่ 8-21 คลาส Gate ซึ่งเป็นคลาสแม่ของเกตต่าง ๆ เช่น And และ Nand



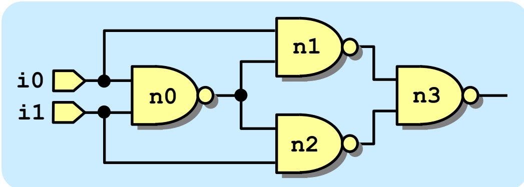
in ของ N0 คือ ⟨ X1, N2 ⟩

out ของ N0 คือ ⟨ X2, N2 ⟩

รูปที่ 8-16 ตัวอย่างรายการ in และ out ของเกต N0

รหัสที่ 8-21 แสดงคลาส Gate แต่ละเกตประกอบด้วยรายการ in และ out ซึ่งเป็นรายการของเกตที่ต่อมายังขาเข้า และรายการของเกตที่รับสัญญาณขาออก (ดูตัวอย่างในรูปที่ 8-16) นอกจากนี้มี delay เก็บเวลาหน่วง และ output เก็บระดับสัญญาณที่ขาออก มีบริการ addInput(g) เพื่อเพิ่มเกต g ที่รายการ in ของเรา (บรรทัดที่ 8) และในขณะเดียวกันก็เพิ่มเกตเราที่รายการ out ของ g ด้วย (บรรทัดที่ 9) มีเมท็อด eval เพื่อประเมินสัญญาณที่ขาออกจากสัญญาณที่ขาเข้า eval เป็น abstract แสดงว่า คลาสลูกของ Gate ต้องเป็นผู้กำหนดพฤติกรรม เช่น คลาส And ต้องเขียน eval ให้นำค่าของขาเข้าทั้งหมดมาแอนด์กัน ส่วนตัวสร้างของ Gate ที่มีให้คลาสลูกเรียกนั้นรับพารามิเตอร์ที่กำหนดเวลาหน่วงของเกต

รหัสที่ 8-22 แสดงตัวอย่างการสร้างวงจรตรรกในรูปที่ 8-17 ในที่นี้เราสร้างแนนด์สี่ตัว และช่องขาเข้าสองช่อง จากนั้นใช้เมท็อด `addInput` เพื่อสร้างการเชื่อมโยงให้ได้เป็นวงจร ขอย้ำอีกครั้งว่า `addInput` จะเพิ่มเกตขาเข้าของเรา และเพิ่มเกตขาออกของเกตอื่นอย่างอัตโนมัติ เช่น คำสั่ง `n1.addInput(i0)` จะเพิ่ม `i0` ให้กับรายการ `in` ของ `n1` และเพิ่ม `n1` ให้กับรายการ `out` ของ `i0` ด้วย



รูปที่ 8-17 ตัวอย่างวงจรการสร้าง exclusive-or ด้วย Nand

```
Input i0 = new Input(6), i1 = new Input(6);
Gate n0 = new Nand(2), n1 = new Nand(2);
Gate n2 = new Nand(2), n3 = new Nand(2);
n0.addInput(i0); n0.addInput(i1);
n1.addInput(i0); n1.addInput(n0);
n2.addInput(i1); n2.addInput(n0);
n3.addInput(n1); n3.addInput(n2);
```

สร้างเกตและอินพุต

เชื่อมต่อกันเป็นวงจร

รหัสที่ 8-22 ตัวอย่างการสร้างวงจรในรูปที่ 8-17

```
public class Input extends Gate {
  int[] inputSignal = new int[1];
  int nextSignal;

  public Input(int d) {super(d);}
  public void setSignal(int[] v) {
    inputSignal = v;
    nextSignal = 0;
  }
  public Value eval() {
    int v = inputSignal[nextSignal];
    nextSignal = (nextSignal + 1) % inputSignal.length;
    return v == 0 ? Value.ZERO : Value.ONE;
  }
}
```

แถวลำดับของสัญญาณ

คืนสัญญาณถัดจากที่เรียกครั้งที่แล้ว

รหัสที่ 8-23 คลาส Input เพื่อสร้างช่องสัญญาณขาเข้า

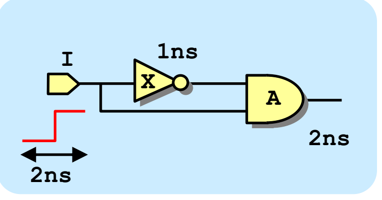
วัตถุประสงค์ของการใช้ตัวจำลองวงจร ก็คือต้องการดูพฤติกรรมการทำงานของวงจร โดยการป้อนสัญญาณทดสอบให้กับช่องขาเข้าแล้วดูการเปลี่ยนแปลงของขาสัญญาณที่สนใจ จึงเป็นหน้าที่ของคลาส `Input` (รหัสที่ 8-23) ในการผลิตสัญญาณทดสอบดังกล่าว เพื่อให้เกิดความกลมกลืนกันในการออกแบบ เราถือว่า `Input` ก็เป็น `Gate` แบบหนึ่ง (นั่นคือให้ `Input extends Gate`)

มีเมท็อด setSignal ซึ่งรับแถวลำดับของเลข 0, 1 ที่บรรยายการเปลี่ยนแปลงสัญญาณ เมื่อเรียก eval หนึ่งครั้งก็จะคืนค่าในแถวลำดับสัญญาณนี้กลับไปทีละช่องจากซ้ายไปขวา (และวนกลับมาช่องศูนย์ใหม่) ตัวอย่างเช่น การทดสอบวงจรในรูปที่ 8-17 ก็ควรให้ขา i0 และ i1 ป้อนสัญญาณ 4 รูปแบบคือ 00, 01, 10, 11 จึงต้องให้ขา i0 ผลิตสัญญาณ 0, 0, 1, 1 ส่วนขา i1 ผลิต 0, 1, 0, 1 เขียนได้ด้วยคำสั่งดังนี้ (ระยะเวลาระหว่างสัญญาณแต่ละตัวนั้นกำหนดไว้เป็นเวลาหน่วงตอนสร้าง Input)

```
i0.setSignal(new int[]{0, 0, 1, 1});
i1.setSignal(new int[]{0, 1, 0, 1});
```

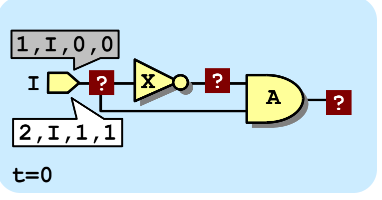
คราวนี้มาดูคลาส Event ซึ่งเป็นคลาสหลักที่แทนเหตุการณ์การเปลี่ยนแปลงสัญญาณระหว่างการจำลองการทำงาน เหตุการณ์การเปลี่ยนแปลงต่าง ๆ เริ่มจากการเปลี่ยนแปลงสัญญาณที่ช่องขาเข้าแล้วส่งผลไปยังเกตต่าง ๆ แต่ละเหตุการณ์ประกอบด้วยสี่สิ่งอันดับ ได้แก่ หมายเลขเหตุการณ์, ชื่อเกต, ระดับสัญญาณ, และเวลาที่ระดับสัญญาณจะเกิดที่ขาออกของเกต เช่น [4,A,1,5] แทนเหตุการณ์หมายเลข 4 ที่ให้ขาออกของเกต A เป็น 1 เมื่อเวลา 5 หน่วย

จะขอยกตัวอย่างการจำลองในรูปที่ 8-18 (1) ตัววงจรประกอบด้วยนอตเกต (X) ที่มีเวลาหน่วง 1 หน่วย ต่อกับแอนด์เกต (A) ที่มีเวลาหน่วง 2 หน่วย สัญญาณขาเข้ามีค่า 0 นาน 1 หน่วยเวลา แล้วเปลี่ยนเป็น 1 อีก 1 หน่วยเวลา โดยเราต้องการดูว่า สัญญาณขาออกของเกต A จะเป็นอย่างไร ?

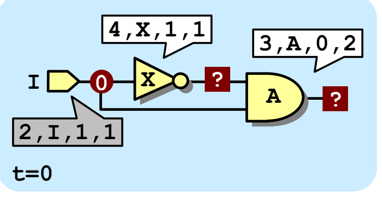


(1)

เริ่มด้วยการสร้างเหตุการณ์ [1,I,0,0] และ [2,I,1,1] แทนการเปลี่ยนแปลงของสัญญาณขาเข้า เหตุการณ์ทั้งสองถูกผลิตและเก็บในฮีปน้อยสุด โดยใช้เวลาของเหตุการณ์เป็นตัวกำหนดอันดับแบบฮีป เหตุการณ์ในฮีปจึงถูกลบตามลำดับเวลาของเหตุการณ์ กรณีที่เหตุการณ์มีเวลาน้อยสุดเท่ากัน จะเลือกเหตุการณ์ที่ถูกผลิตก่อนออกมาก่อน (ซึ่งดูได้จากหมายเลขเหตุการณ์ เพราะเหตุการณ์ที่ถูกผลิตก่อนจะมีหมายเลขน้อยกว่า) จากรูป (2) ลบได้เหตุการณ์ [1,I,0,0] แล้วตั้งสัญญาณออกของ I เป็น 0 จากนั้นผลิตเหตุการณ์ให้กับเกตต่าง ๆ ที่มีขาเข้าต่อกับ I ในรูป (3) ได้แก่ [3,A,0,2] เพราะขาเข้าของ A เป็น 0 และ UNDEFINED (แสดงในรูปด้วย



(2)

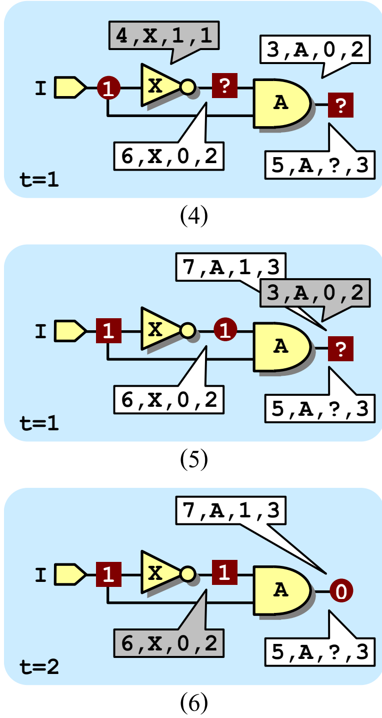


(3)

รูปที่ 8-18 ตัวอย่างการจำลอง

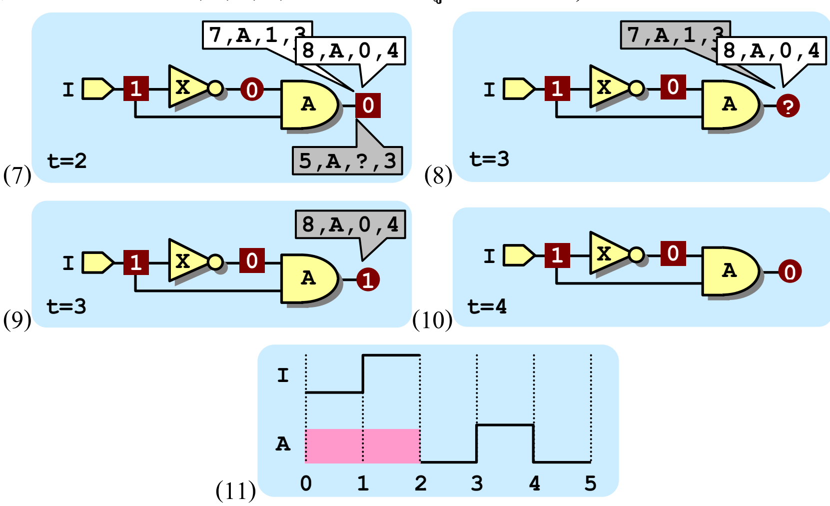
เครื่องหมาย ?) แอนด์แล้วได้ 0 ในอีก 2 หน่วยเวลา (เวลา 2 ของ [3,A,0,2] มาจากเวลา 0 ของ [1,I,0,0] บวกกับเวลาหน่วงของ A ซึ่งคือ 2) และเหตุการณ์ [4,X,1,1] เพราะตอนนี้ขาเข้าของ X เป็น 0 ต้องได้ผลเป็น 1 ในอีก 1 หน่วยเวลาต่อไป

ตอนนี้ [2,I,1,1] และ [4,X,1,1] มีเวลาน้อยสุดเท่ากัน จึงลบตัวที่ถูกผลิตก่อน(โดยดูจากหมายเลขเหตุการณ์) ได้ [2,I,1,1]ก่อให้เกิด [5,A,?,3] และ [6,X,0,2] ได้ดังรูป (4) ลบเหตุการณ์ต่อได้ [4,X,1,1] เพื่อตั้งค่า 1 ให้นอตเกต X ส่งผลให้อีก 2 หน่วยเวลาต่อจากนี้ A จะเป็น 1 จึงผลิต [7,A,1,3] ได้ดังรูป (5) ถึงตอนนี้มีสองเหตุการณ์ที่มีเวลาน้อยสุดเป็น 2 ลบได้ [3,A,0,2] เพราะเกิดก่อน แล้วตั้งค่าขาออกของ A เป็น 0 ซึ่งไม่มีการผลิตเหตุการณ์อื่นแต่อย่างใด เพราะสัญญาณขาออกของ A ไม่ได้ต่อกับเกตใด ๆ ได้ดังรูป (6)



รูปที่ 8-18 ตัวอย่างการจำลอง

ต่อด้วยการลบเหตุการณ์ออก และผลิตเหตุการณ์เข้าไปในฮีปไปเรื่อย ๆ โดยจะหยุดการทำงานก็เมื่อไม่มีเหตุการณ์เหลือในฮีป หรือไม่ก็เหตุการณ์ที่ลบออกมามีเวลาที่เกินเวลาที่ผู้ใช้ต้องการดู (รูปที่ 8-18 (7) ถึง (10)) จากการจำลองการทำงานในตัวอย่าง สรุปแล้วสัญญาณขาออกของเกต A จะมีค่าเป็น ?,?,0,1,0 ตามลำดับเวลา 0, 1, 2, 3, 4 ดังแสดงในรูปที่ 8-18 (11)



รูปที่ 8-18 ตัวอย่างการจำลองวงจรตามเหตุการณ์

```
01 class Event implements Comparable {
02   private static int idCounter = 0;
03   final int id;
04   final Gate gate;
05   final Value output;
06   final int time;
07   Event(Gate gate, int time) {
08     id = idCounter++;
09     this.gate = gate;
10     this.time = time;
11     output = gate.eval();
12   }
13   public int compareTo(Object o) {
14     Event that = (Event) o;
15     int cmp = this.time - that.time;
16     return cmp != 0 ? cmp : (this.id - that.id);
17   }
18 }
```

ตัวนับหมายเลขเหตุการณ์ เพิ่มขึ้นหนึ่งทุกครั้งที่สร้างอ็อบเจกต์ใหม่

ขาออกของ gate จะมีค่าเป็น output เมื่อถึงเวลา time

เหตุการณ์ที่น้อยกว่าคือตัวที่มี time น้อยกว่า ในกรณีที่ time เท่ากัน เหตุการณ์ที่น้อยกว่าก็คือตัวที่ถูกผลิตก่อน คือมี id น้อยกว่า

รหัสที่ 8-24 คลาส `Event` ที่แทนเหตุการณ์การเปลี่ยนแปลงสัญญาณ

รหัสที่ 8-24 แสดงคลาส `Event` ภายในมีตัวแปร `id`, `gate`, `output` และ `time` ซึ่งคือข้อมูลที่ได้อธิบายมา มี `idCounter` เป็นตัวนับภายในที่มีค่าเพิ่มขึ้นหนึ่งทุกครั้งที่มีการเรียกตัวสร้างแล้วให้แทนเป็นหมายเลขเหตุการณ์ ให้สังเกตว่า ค่า `output` ของ `gate` ได้มาจากการเรียก `gate.eval()` (บรรทัดที่ 11) ซึ่งเป็นค่าของขาออกเมื่อถึงเวลา `time` เนื่องจากเรานำเหตุการณ์ไปเก็บในฮีป คลาส `Event` จึงต้องเป็น `Comparable` มีเมท็อด `compareTo` จัดอันดับแบบฮีปด้วยเวลาของเหตุการณ์ และในกรณีที่เวลาเท่ากัน ก็ให้จัดอันดับตามลำดับที่เหตุการณ์ถูกผลิต นั่นคือ `compareTo` เปรียบเทียบด้วย `time` และ `id` ถ้า `time` ไม่เท่ากัน ก็เปรียบเทียบด้วย `time` แต่ถ้า `time` เท่ากัน ก็เปรียบเทียบด้วย `id` (บรรทัดที่ 15 และ 16)

และก็มาถึงรหัสที่ 8-25 ซึ่งแสดงรายละเอียดทั้งหมดของการจำลองวงจร เมท็อด `simulate` รับ `gates` ซึ่งคือรายการของเกตต่าง ๆ (ซึ่งรวมทั้งช่องขาเข้าด้วย), `watch` ซึ่งคือเกตที่ผู้ใช้สนใจดูสัญญาณขาออก และ `fin` ซึ่งคือเวลาที่ให้หยุดการจำลองวงจร ผลที่ได้จากการจำลองเป็นรายการของเหตุการณ์ที่เกต `watch` ตามลำดับเวลา การทำงานเริ่มด้วยการเตรียมรายการผลลัพธ์และแถวคอยบุริมภาพแบบฮีปน้อยสุด (บรรทัดที่ 3 และ 4) จากนั้นเรียก `addInputEvents` เพื่อผลิตเหตุการณ์การเปลี่ยนแปลงสัญญาณขาเข้าของทุกช่องสัญญาณเก็บลงฮีป แล้วเริ่มวงวนการจำลอง หมุนทำไปจนกว่าฮีปจะว่าง ภายในวงวนเริ่มด้วยการลบเหตุการณ์ออกจากฮีป ให้ `e` คือเหตุการณที่ลบออกมานี้ ถ้าเวลาที่ `e` เกิดซึ่งคือ `e.time` อยู่หลังเวลาที่สนใจ ก็เลิกการทำงาน (บรรทัดที่ 8) และถ้า `e` เป็นเหตุการณ์ของเกตที่สนใจ ก็เก็บ `e` ไว้ในรายการผลลัพธ์ (บรรทัดที่ 9) จากนั้นตั้งระดับสัญญาณที่เก็บใน

e.output ให้กับเกต e.gate (บรรทัดที่ 10) แล้วเริ่มผลิตเหตุการณ์ใหม่ ๆ ให้กับเกตที่ต่อจากขาออกของ e.gate โดยเกตต่าง ๆ ที่มีผลกระทบเหล่านี้ก็คือเกตในรายการ out ของ e.gate ถ้า x คือเกตที่เกิดผลกระทบจาก e ก็ต้องผลิตเหตุการณ์ที่จะเกิดขึ้นหลังจาก e.time ไปอีกเท่ากับเวลาหน่วงของเกต x (บรรทัดที่ 14)

```
01 public class LogicSimulator {
02   public static List simulate(List gates,Gate watch,int fin){
03     List result = new ArrayList();
04     PriorityQueue q = new BinaryMinHeap();
05     addInputEvents(q, gates, fin);
06     while (!q.isEmpty()) {
07       Event e = (Event) q.dequeue();
08       if (e.time > fin) break;
09       if (e.gate == watch) result.add(e);
10       e.gate.output = e.output;
11       List out = e.gate.out;
12       for (int i = 0; i < out.size(); i++) {
13         Gate x = (Gate) out.get(i);
14         q.enqueue(new Event(x, e.time + x.delay));
15       }
16     }
17     return result;
18   }
19   static void addInputEvents(PriorityQueue q,
20                                            List gates, int fin ) {
21     for(int i=0; i<gates.size(); i++) {
22       if (gates.get(i) instanceof Input) {
23         Input inp = (Input) gates.get(i);
24         for(int t=0; t<=fin; t+=inp.delay)
25           q.enqueue(new Event(inp, t));
26       }
27     }
28   }
29   public static void main(String[] args){
30     Input I = new Input(1);
31     Gate X = new Not(1);
32     Gate A = new And(2);
33     I.setSignal(new int[]{0,1});
34     X.addInput(I);
35     A.addInput(I); A.addInput(X);
36     List gates = new ArrayList();
37     gates.add(I); gates.add(X); gates.add(A);
38     List result = simulate(gates, A, 4);
39     reportResult(result);
40   }
..   ...
```

เพิ่มเหตุการณ์การเปลี่ยนแปลงสัญญาณขาเข้า

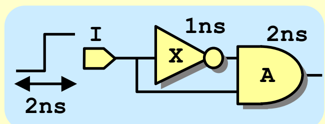
ลบเหตุการณ์ถัดไป เรียงตามเวลา ถ้าเลยเวลาที่สนใจ ก็เลิก

ถ้าเป็นเหตุการณ์ของเกตที่สนใจ ก็เก็บเหตุการณ์นี้ไว้

ผลิตเหตุการณ์ใหม่ให้กับเกตทุกตัวที่รับสัญญาณจากเกตเรา

ดูทีละเกต สนใจเฉพาะ Input

เพิ่มเหตุการณ์การเปลี่ยนแปลงสัญญาณ ให้ครบช่วงเวลาที่สนใจจำลองการทำงาน



สนใจเกต A จำลองการทำงาน 4 หน่วยเวลา

รายงานผล

รหัสที่ 8-25 เมท็อด simulate สำหรับการจำลองวงจรตรรก

หลัง `simulate` ทำงานเสร็จ จะนำรายการของเหตุการณ์ที่เกิด ณ เกตที่สนใจ ไปรายงานให้ผู้ใช้ทราบด้วยเมท็อด `reportResult` (บรรทัดที่ 39) จะขอเขียนแบบง่ายสุด คือแสดงผลการจำลองเป็นข้อความ ขอให้กลับไปดูตัวอย่างการจำลองในรูปที่ 8-18 ถ้าเราสนใจที่ขาออกของเกต `A` จะพบเหตุการณ์ `[3,A,0,2]`, `[5,A,?,3]`, `[7,A,1,3]`, และ `[8,A,0,4]` ขอให้ดูเฉพาะสองจำนวนทางขวาของแต่ละเหตุการณ์ จะได้ว่า ขาออกของเกต `A` มีค่า 0, ?, 1, 0 เมื่อเวลา 2, 3, 3, 4 ที่น่าสงสัยก็คือตรงเวลา 3 มีสองค่าที่ต่างกัน เราต้องเลือกค่าหลังเพราะมาจากเหตุการณ์ที่ผลิตทีหลัง (ดูจาก `id` ของเหตุการณ์) ดังนั้นการรายงานต้องตัดเหตุการณ์ที่ซ้ำกันในลักษณะนี้โดยรายงานเฉพาะตัวหลังสุดที่มีเวลาซ้ำกัน อีกจุดหนึ่งคือเราเริ่มจำลองที่เวลาเป็น 0 แต่เหตุการณ์สนใจที่ได้รับอาจเริ่มหลังจาก 0 ดังนั้นจากเวลา 0 จนถึงเวลาของเหตุการณ์ที่ได้รับระดับสัญญาณของเกตที่สนใจจึงต้องเป็นแบบ `UNDEFINED` ได้ผลการจำลองเป็นข้อความ `??010` แสดงออกจอภาพ สรุปการรายงานผลการจำลองแบบง่ายด้วยข้อความทางจอภาพ มีการทำงานดังแสดงในรหัสที่ 8-26

```
41    private static void reportResult(List result) {
42      int t = 0;
43      Value v = Value.UNDEFINED;
44      Event e0 = (Event) result.get(0);
45      for(; t<e0.time; t++) System.out.print(v);
46      for(int i=1; i<result.size(); i++) {
47        Event e1 = (Event) result.get(i);
48        if (e0.time != e1.time) {
49          for(; t<e0.time; t++) System.out.print(v);
50          System.out.print(v=e0.output);
51          t++;
52        }
53        e0 = e1;
54      }
55      for(; t<e0.time; t++) System.out.print(v);
56      System.out.print(e0.output);
57    }
58  }
```

แสดงผลหนึ่งตัวต่อหนึ่งหน่วยเวลา

แสดงเมื่อพบตัวใหม่มีเวลาต่างกับตัวก่อน

รหัสที่ 8-26 การรายงานผลการจำลองแบบง่ายด้วยข้อความทางจอภาพ

ตัวจำลองวงจรตรรกที่ได้นำเสนอมานี้ อาศัยแถวคอยบุริมภาพเป็นตัวเก็บอ็อบเจกต์ที่แทนเหตุการณ์การเปลี่ยนแปลงสัญญาณระหว่างการจำลองการทำงาน เหตุการณ์หนึ่งนำไปสู่การผลิตเหตุการณ์ใหม่ ๆ โดยเหตุการณ์เริ่มต้นมาจากการเปลี่ยนสัญญาณของช่องขาเข้า เหตุการณ์ต่าง ๆ ถูกผลิตแบบไม่ได้เรียงตามเวลาที่สัญญาณจะเปลี่ยน เพราะเวลาหน่วงของเกตต่าง ๆ ไม่เท่ากัน แต่เราสนใจพิจารณาเหตุการณ์ในลำดับของเวลาที่เกิดการเปลี่ยนแปลงสัญญาณ แถวคอยบุริมภาพจึงเป็นตัวหลักในการจัดลำดับการพิจารณา โดยอาศัยฮีปน้อยสุด เก็บเหตุการณ์ที่มีเมท็อด `compareTo` เพื่อ

เปรียบเทียบสองเหตุการณ์ว่า ตัวใดน้อยกว่า อันเป็นกลไกหลักในการควบคุมลำดับการพิจารณาเหตุการณ์ระหว่างการจำลองวงจร

แบบฝึกหัด

1. จงเขียนคลาส `BinaryHeap` (เป็นฮีปมากสุด) ด้วยตนเอง โดยไม่ดูรายละเอียดในหนังสือ

2. คลังคลาสมาตรฐานของจาวาไม่มีอินเทอร์เฟซ `PriorityQueue` แต่ในชุด `java.util` มีคลาสชื่อ `PriorityQueue` ซึ่งก็เป็นฮีปแบบทวิภาค จงอธิบายบริการต่าง ๆ ของคลาสมาตรฐานนี้

3. จงปรับปรุงเมท็อด `fixUp` และ `fixDown` โดยหลีกเลี่ยงไม่ใช้การสลับข้อมูล แต่ใช้การย้ายข้อมูลเพื่อลดจำนวนการย้ายข้อมูลที่ต้องทำ

4. จงเพิ่มเมท็อด `public void merge(BinaryHeap h)` ให้กับคลาส `BinaryHeap` ที่ทำงานอย่างรวดเร็วเพื่อรวมฮีป `h` เข้ากับฮีปเรา โดยหลังการทำงานจะไม่สนใจใช้ฮีป `h` อีกต่อไป

5. จงเพิ่มเมท็อด `public BinaryHeap getLessThan(Object x)` ให้กับคลาส `BinaryHeap` ที่ทำงานดีกว่าที่แสดงข้างล่างนี้ เพื่อคืนฮีปใหม่ที่ประกอบด้วยข้อมูลในฮีปเราทุกตัวที่มีค่าน้อยกว่า `x` (ฮีปเราไม่เปลี่ยนแปลง)

```
public BinaryHeap getLessThan(Object x) {
  BinaryHeap h = new BinaryHeap();
  for (int i=0; i<size; i++) {
    if (((Comparable)elementData[i]).compareTo(x) < 0)
      h.enqueue(elementData[i]);
  return h;
}
```

6. จงเพิ่มเมท็อด `public static boolean isMaxHeap(Object[] d)` ให้กับคลาส `BinaryHeap` เพื่อตรวจสอบว่า ข้อมูลที่เก็บใน `d` ทั้งหมดมีอันดับแบบฮีปมากสุดหรือไม่

7. ถ้ารากของฮีปเปลี่ยนจากที่เก็บในแถวลำดับช่องที่ 0 มาเก็บในช่องที่ r จะต้องเปลี่ยนสูตรการคำนวณช่องของลูกซ้าย ลูกขวาและพ่อเป็นอะไร

8. จงแสดงขั้นตอนการเปลี่ยนแปลงของข้อมูลในแถวลำดับระหว่างการสร้างฮีปจากแถวลำดับ ด้วยรหัสที่ 8-11 โดยที่แถวลำดับเริ่มต้นที่ส่งให้คือ 9, 0, 8, 3, 1, 7, 2, 5, 6, 7, 4

9. การสร้างฮีปจากแถวลำดับที่แสดงในรหัสที่ 8-11 อาศัยการ `fixDown` จากช่องที่ `size-1` ย้อน กลับมาจนถึงช่องที่ 0 ซึ่งสามารถเปลี่ยนเป็นเริ่มจากช่องที่ `size/2-1` ก็ได้ เพราะเหตุใด

10. คุณคณิตเสนอวิธีการสร้างฮีปมากสุดจากแถวลำดับ 5 วิธีข้างล่างนี้ อยากทราบว่า วิธีใดใช้ได้ วิธีใดใช้ไม่ได้ วิธีใดน่าใช้

    10.1. เรียก `fixDown` เริ่มจากช่องที่ 0 ไปจนถึงช่องที่ `size-1`

    10.2. เรียก `fixUp` เริ่มจากช่องที่ 0 ไปจนถึงช่องที่ `size-1`

    10.3. เรียก `fixUp` เริ่มจากช่องที่ `size-1` กลับมาที่ช่องที่ 0

    10.4. เรียงลำดับข้อมูลในแถวลำดับจากมากไปน้อย

    10.5. เรียงลำดับข้อมูลในแถวลำดับจากน้อยไปมาก

11. การทำงานของรหัสที่ 8-9 ในกรณีช้าสุดจะต้องเปรียบเทียบข้อมูล (ด้วย `greaterThan`) กี่ครั้ง

12. จงวิเคราะห์จำนวนการสลับข้อมูลระหว่างการสร้างฮีปจากแถวลำดับด้วยการค่อยเพิ่มข้อมูลทีละตัว ๆ จนหมด (รหัสที่ 8-10) แบบละเอียดว่า ต้องทำกี่ครั้งในกรณีช้าสุด

13. จงวิเคราะห์จำนวนการสลับข้อมูลระหว่างการสร้างฮีปจากแถวลำดับด้วยการ `fixDown` จากช่อง `size-1` ถึง 0 (รหัสที่ 8-11) แบบละเอียดว่า ต้องทำกี่ครั้งในกรณีช้าสุด

14. เราสามารถทำให้ต้นไม้ฮีปเตี้ยลงได้ โดยปรับปรุงฮีปแบบทวิภาคที่หนึ่งปมมีสองลูก ให้กลายเป็นหนึ่งปมมี d ลูก เรียกว่า *ฮีปแบบดี* (d-heap) โดย 2-heap ก็คือฮีปแบบทวิภาค 1-heap ก็คือรายการที่เก็บข้อมูลเรียงลำดับ จงเขียนคลาส `DHeap` เพื่อสร้างแถวคอยเชิงบุริมภาพ

15. คุณมั่นบอกว่า เราสามารถหาค่ามากที่สุดในฮีปน้อยสุดได้ง่าย ๆ โดยการวิ่งไล่เปรียบเทียบหาตัวมากสุดเฉพาะครึ่งขวาของแถวลำดับ (ตั้งแต่ตัวที่ $\lfloor$`size/2`$\rfloor$ ถึง `size-1`) ไม่ต้องไปสนใจครึ่งซ้าย นอกจากนี้คุณมั่นยังบอกว่า นี่เป็นวิธีที่ดีที่สุดแล้ว จงให้เหตุผลสนับสนุนหรือคัดค้านคำกล่าวอ้างข้างต้น (ถ้าคัดค้านให้นำเสนอวิธีที่ดีกว่าด้วย)

16. จงใช้ตัวจำลองวงจรที่ได้นำเสนอมาเพื่อจำลองวงจรในรูปที่ 8-14 ทางขวา และรูปที่ 8-17

ต้นไม้แบบทวิภาค

9

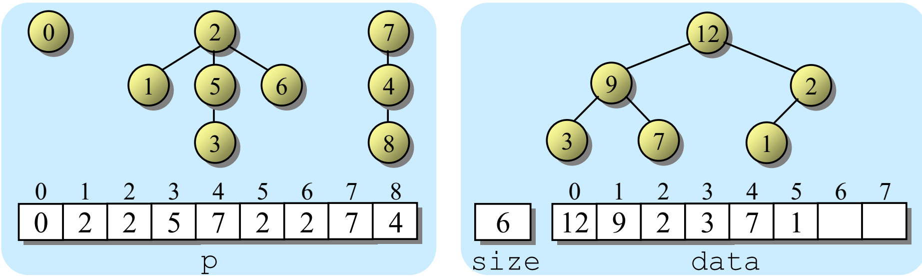
ต้นไม้เป็นโครงสร้างข้อมูลพื้นฐานอีกตัวหนึ่งที่ได้รับการประยุกต์มากมาย ด้วยลักษณะที่ข้อมูลแต่ละตัวมีความสัมพันธ์กับตัวอื่น ๆ ได้หลาย ๆ ตัว เป็นระดับ ๆ เมื่อเปรียบกับลักษณะที่ข้อมูลเรียงกันไปเป็นแถว เช่น รายการ กองซ้อน หรือแถวคอย ที่เรียกกันว่าโครงสร้างเชิงเส้น ต้นไม้จะมีลักษณะไม่เชิงเส้น ซับซ้อนมากกว่า แต่ก็มีความยืดหยุ่นและประสิทธิภาพสูงกว่า จะว่าไปแล้วเราได้เห็นการใช้ต้นไม้เก็บข้อมูลกันมาสองแบบ คือการใช้ป่าไม้แทนกลุ่มเซตไร้ตัวร่วมในบทที่ 1 โดยที่ต้นไม้หนึ่งต้นแทนเซตของจำนวนเต็มหนึ่งเซต และการใช้ต้นไม้แทนแถวคอยบุริมภาพในบทที่ 8 ต้นไม้มีหลายประเภท ต้นไม้แบบพื้นฐานและยอดนิยมสุด ๆ ที่ใช้เป็นโครงสร้างข้อมูลเห็นจะเป็น*ต้นไม้แบบทวิภาค* (binary tree) บทนี้จะได้กล่าวถึงการสร้าง บริการพื้นฐาน และตัวอย่างการประยุกต์ต้นไม้แบบทวิภาค

การสร้างต้นไม้

การเก็บข้อมูลให้มีโครงสร้างแบบต้นไม้นั้น เราเก็บข้อมูลตามปมต่าง ๆ แล้วกำหนดความสัมพันธ์ของปมต่าง ๆ ให้เชื่อมโยงกันเป็นต้นไม้ มาทบทวนความจำกันด้วยตัวอย่างในรูปที่ 9-1 รูปซ้ายเป็นการแทนกลุ่มเซตไร้ตัวร่วมด้วยป่าไม้ ต้นไม้หนึ่งต้นแทนหนึ่งเซต โดยมีข้อจำกัดว่า ข้อมูลในเซตต่าง ๆ เป็นจำนวนเต็มตั้งแต่ 0 ถึง $n-1$ เราสร้างทั้งป่าไม้ด้วยแถวลำดับหนึ่งแถวที่มี n ช่อง โดย `p[k]` เก็บข้อมูลที่เป็นปมพ่อของ `k` ในกรณีที่ `p[k]` มีค่าเป็น `k` แสดงว่า `k` เป็นรากของต้นไม้ เช่น `p[4]=7` แสดงว่า ปมพ่อของ 4 คือ 7 เป็นต้น ด้วยวิธีนี้การหาปมพ่อของ `k` ก็เพียงดู `p[k]` จึงทำได้รวดเร็วในเวลาคงตัว ในขณะที่การหาปมลูกของ `k` ย่อมต้องวิ่งหาทั้งแถวลำดับว่า ช่อง `m` ใดที่มี `p[m]=k` ซึ่งช้า ส่วนรูปขวา เป็นการสร้างต้นไม้แบบทวิภาคหนึ่งต้นด้วยแถวลำดับ โดยอาศัยสูตรในการคำนวณ

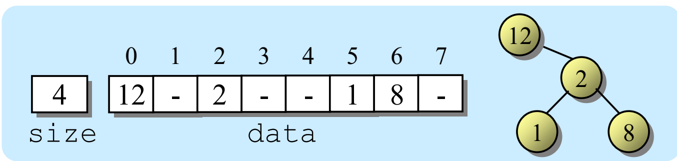
ตำแหน่งของปมพ่อ-ลูก กำหนดให้ปมรากอยู่ที่ช่องเลขที่ 0 การคำนวณเลขที่ของปมพ่อ, ลูกซ้าย และลูกขวาทำได้ดังนี้

- พ่อของปมที่เก็บในช่องที่ k ถูกเก็บในช่องที่ $\lfloor (k-1)/2 \rfloor$
- ลูกซ้ายของปมที่เก็บในช่องที่ k ถูกเก็บในช่องที่ $2k+1$
- ลูกขวาของปมที่เก็บในช่องที่ k ถูกเก็บในช่องที่ $2k+2$



รูปที่ 9-1 การสร้างต้นไม้ด้วยแถวลำดับสองรูปแบบ

การสร้างต้นไม้แบบทวิภาคด้วยแถวลำดับและสูตรคำนวณแบบนี้ ใช้ได้ดี หาตำแหน่งของปมพ่อ ปมลูกได้รวดเร็ว แต่มีข้อจำกัดว่า จะใช้เนื้อที่น้อยสุด (ต้นไม้ที่มี n ปมสร้างได้ด้วยแถวลำดับขนาด n ช่อง) ก็เฉพาะกับลักษณะของต้นไม้ที่มีโครงสร้างได้ดุล มีปมเต็มทุกระดับ ยกเว้นก็ระดับล่างสุด ไม่เต็มก็ได้แต่ต้องมีปมเรียงจากซ้ายไปขวา ถ้ามีโครงสร้างที่ไม่เป็นดังข้อกำหนดจะสิ้นเปลืองเนื้อที่บางช่องดังตัวอย่างในรูปที่ 9-2



รูปที่ 9-2 การสร้างต้นไม้ด้วยแถวลำดับที่สิ้นเปลืองเนื้อที่

สำหรับกรณีที่เราต้องการสร้างต้นไม้ทั่วไปที่ไม่มีข้อจำกัดว่าต้องเก็บจำนวนเต็ม ไม่จำเป็นต้องเป็นต้นไม้ได้ดุล การไปยังปมพ่อและลูกทำได้รวดเร็ว และมีความคล่องตัวในการเพิ่มและลบปม เราสามารถทำได้โดยเก็บตัวโยงระหว่างปมพ่อ-ลูกไว้ตามปมต่าง ๆ รูปที่ 9-3 แสดงตัวอย่างการสร้างต้นไม้ด้วยการโยงปมต่าง ๆ รูป (ก) เป็นวิธีการสร้างต้นไม้แบบทวิภาคด้วยการโยงที่ง่ายสุด แต่ละปมเก็บตัวโยงไปยังข้อมูล ตัวโยงไปยังปมของลูกซ้าย และตัวโยงไปยังปมของลูกขวา หากไม่มีลูกก็ให้ตัวโยงมีค่าเป็น `null` โดยมีตัวแปร `root` ชี้ไปยังปมรากของต้นไม้ รหัสที่ 9-1 แสดงคลาส `BinaryTree` ซึ่งเป็นคลาสหลักที่เราจะใช้ในการสร้างคลาสอื่น ๆ ที่มีโครงสร้างต้นไม้แบบทวิภาค มี `Node` เป็นคลาสภายในใช้แทนปมของต้นไม้ แถมเมท็อด `isLeaf` ที่ตรวจสอบว่า ปมที่เรียกเป็น

ใบหรือไม่ ซึ่งจะเป็นใบเมื่อทั้งลูกซ้ายและขวาเป็น `null` รูปที่ 9-3 (ก) ให้ความสะดวกในการหาปมลูก แต่ถ้าจะกลับไปหาปมพ่อ คงต้องมีชุดตัวแปรจำสภาวะหรือวิถีการเข้าถึงปมเพื่อจะได้กลับไปหาปมพ่อได้ ถ้าไม่อยากยุ่งยาก ก็เพิ่มอีกตัวแปรตามปมให้โยงกลับไปยังปมพ่อดังแสดงในรูป (ข) และในรหัสที่ 9-2 (อย่างไรก็ตาม เราหลีกเลี่ยงการเก็บปมพ่อได้ถ้าเขียนเมท็อดจัดการต่าง ๆ แบบเวียนเกิด ซึ่งเป็นการยืมกองซ้อนของระบบช่วยจำสภาวะและตำแหน่งของปมพ่อ ระหว่างการเรียกเมท็อด ซึ่งจะได้เห็นตัวอย่างต่อไป)

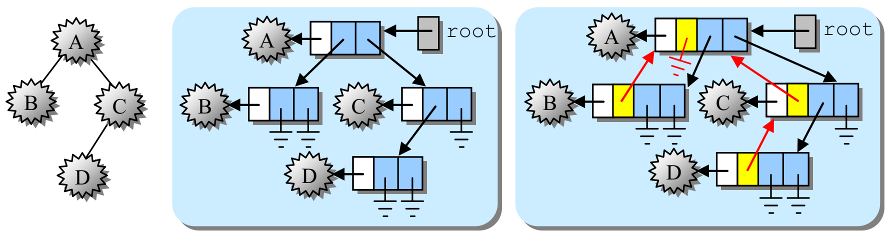
```
public class BinaryTree {
  Node root;
  static class Node {
    Object element;
    Node left;
    Node right;
    Node(Object e, Node l, Node r) {
      element = e; left = l; right = r;
    }
    boolean isLeaf() {return left == null && right == null;}
  }
  ...
```

เก็บรากของต้นไม้

Node บรรยายปมของต้นไม้แบบทวิภาค ประกอบด้วยข้อมูล ลูกซ้าย และลูกขวา

ใบคือปมที่ลูกทั้งสองเป็น null

รหัสที่ 9-1 `BinaryTree` คือคลาสแม่ของสารพัดคลาสที่มีโครงสร้างเป็นต้นไม้แบบทวิภาค



(ก) (ข)

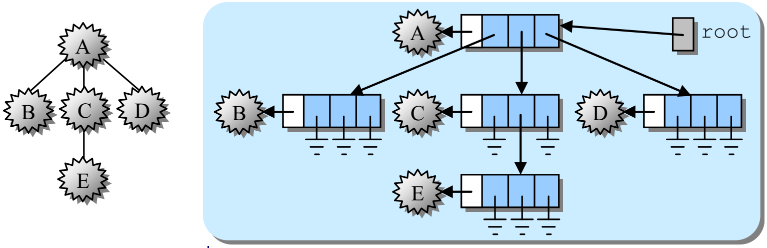
รูปที่ 9-3 การสร้างต้นไม้แบบทวิภาคด้วยการโยง

```
  static class Node {
    Object element;
    Node parent;
    Node left;
    Node right;
```

เพิ่มตัวโยงไปยังปมพ่อ

รหัสที่ 9-2 คลาส Node แบบมีตัวโยงกลับไปยังปมพ่อของต้นไม้ในรูปที่ 9-3 (ข)

ถ้าแต่ละปมมีได้สามลูก ก็เพิ่มจำนวนตัวโยงไปยังปมลูกแต่ละปมให้เป็นสาม ดังรูปที่ 9-4 เรียกว่า *ต้นไม้ 3-ภาค* (3-ary tree) หรือจะสร้างแถวลำดับกำกับปมเพื่อโยงไปยังลูก ๆ (เรียกว่า *m*-ary tree) ซึ่งมีข้อดีตรงที่เราสามารถพุ่งไปยังลูกปมที่ k ได้อย่างรวดเร็วดังแสดงในรหัสที่ 9-3 แต่วิธีนี้เปลือง เพราะตัวโยงส่วนใหญ่จะมีค่าเป็น `null`



รูปที่ 9-4 การสร้างต้นไม้ 3 ภาคด้วยการโยง

```
static class Node {
  Object element;
  Node child1;
  Node child2;
  Node child3;
```

มีได้ 3 ลูกก็มี 3 ตัวโยง

```
static class Node {
  Object element;
  Node[] children = new Node[3];
  ...
```

ใช้แถวลำดับ 3 ช่องเก็บลูก ๆ

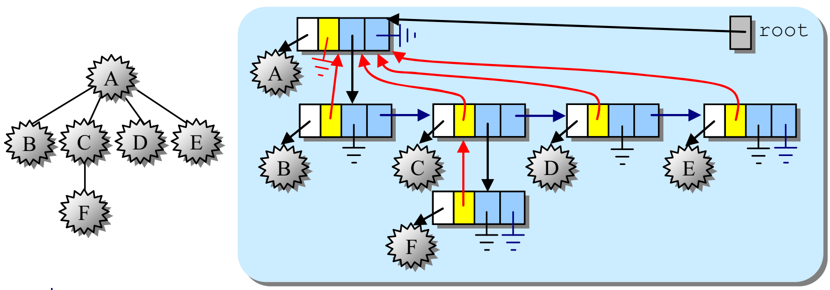
รหัสที่ 9-3 คลาส Node แทนปมของต้นไม้ที่มีได้ 3 ลูกในรูปที่ 9-4

ถ้าต้องการเพิ่มความยืดหยุ่นให้แต่ละปมมีกี่ลูกก็ได้ โดยใช้เนื้อที่ตามจำนวนลูกที่มีอยู่จริง ก็เก็บลูก ๆ ในรายการที่สร้างด้วยรายการโยง เช่น LinkedList ดังแสดงในรหัสที่ 9-4

```
static class Node {
  Object element;
  List children = new LinkedList();
```

ใช้ LinkedList เก็บลูก ๆ

รหัสที่ 9-4 คลาส Node แทนปมของต้นไม้ที่มีกี่ลูกก็ได้



รูปที่ 9-5 การสร้างต้นไม้ด้วยรายการของตัวโยงแบบโยงลูกคนโตและน้องคนถัดไป

หรือจะนำโครงสร้างของรายการโยงมารวมเข้ากับปมของต้นไม้ เพื่อเก็บเป็นรายการโยงของลูก ๆ ก็นิยามให้แต่ละปมของต้นไม้เก็บตัวโยงไปยังลูกคนโต กับเก็บตัวโยงไปยังน้องคนถัดไป ดังแสดงในรูปที่ 9-5 (ในรูปนี้มีการเก็บตัวโยงไปยังปมพ่อด้วย) และในรหัสที่ 9-5

```
static class Node {
  Object element;
  Node parent;
  Node leftChild;
  Node nextSibling;
```

โยงไปยังพ่อ

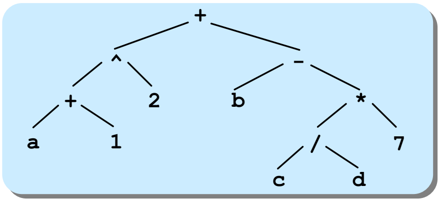
โยงไปยังลูกคนโต

โยงไปยังน้องคนถัดไป

รหัสที่ 9-5 คลาส Node แทนปมของต้นไม้แบบโยงลูกคนโตและน้องคนถัดไปในรูปที่ 9-5

ต้นไม้นิพจน์

ขอยกตัวอย่างการใช้ต้นไม้แบบทวิภาคมาแทนนิพจน์คณิตศาสตร์ ที่เรียกว่า *ต้นไม้นิพจน์* (expression tree) ซึ่งสามารถนำไปใช้ประมวลผลนิพจน์ เช่น การลดรูป การหาอนุพันธ์ ที่จะได้กล่าวในหัวข้อถัดไป ขอเสนอวิธีการสร้างต้นไม้นิพจน์ก่อน เพื่อความง่าย กำหนดให้นิพจน์ประกอบด้วยค่าคงตัว ตัวแปร และ*ตัวดำเนินการ* (operator) +, – , *, / และ ^ (ในจาวา ^ แทนการออร์เฉพาะ แต่จะขอใช้เครื่องหมาย ^ แทนการยกกำลัง) ซึ่งเป็นแบบที่ต้องการ*ตัวถูกดำเนินการ* (operand) สองตัว เราแทนนิพจน์ด้วยต้นไม้ที่ปมภายในเก็บตัวดำเนินการ และใบเก็บค่าคงตัวหรือตัวแปร ดังตัวอย่างในรูปที่ 9-6



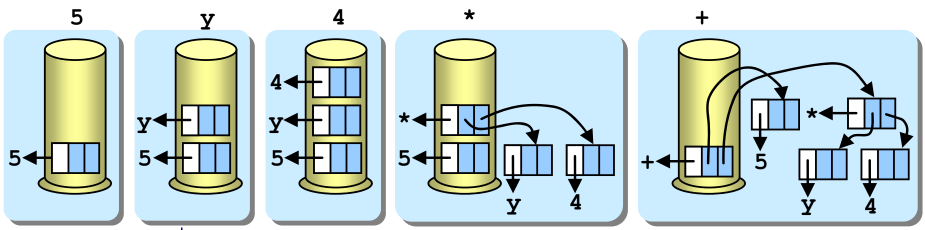
รูปที่ 9-6 ต้นไม้นิพจน์ของ `(a+1)^2 + b - c/d * 7`

ด้วยการแทนนิพจน์แบบนี้ ทำให้สามารถเข้าถึงโครงสร้างของนิพจน์ได้ดี ตัวดำเนินการที่ปมลูกต้องทำให้เสร็จก่อนที่ปมพ่อ จึงไม่ต้องมีเครื่องหมายวงเล็บ และหากต้องการประมวลผลเชิงขนาน ถ้ามีหน่วยประมวลผลกลางหลายตัว ก็สามารถวิเคราะห์ได้ว่า ตัวดำเนินการใดทำพร้อมกันได้บ้าง

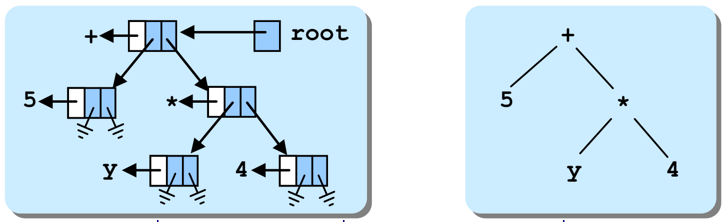
แล้วต้นไม้นิพจน์สร้างได้อย่างไร ? เราอาศัยการแปลงนิพจน์เติมกลางเป็นเติมหลังที่ได้นำเสนอมาในบทที่ 6 (เรื่องกองซ้อน) จากนั้นใช้กองซ้อนอีกตัวช่วยในการสร้างต้นไม้นิพจน์จากนิพจน์เติมหลังได้อย่างง่ายดายดังนี้

- หยิบทีละตัวจากรายการของนิพจน์เติมหลัง (จากตัวแรกไปตัวสุดท้าย) เก็บใส่ `x`
  - ถ้า `x` เป็นตัวถูกดำเนินการ ให้สร้างปมที่มีข้อมูลเป็น `x` แล้วเพิ่มลงกองซ้อน
  - ถ้า `x` เป็นตัวดำเนินการ ให้ลบสองปมจากกองซ้อน มาโยงเป็นปมลูกทางซ้ายและขวาของปมใหม่ที่มี `x` เป็นข้อมูล จากนั้นเพิ่มปมใหม่นี้ลงกองซ้อน
- เมื่อพิจารณาหมดแล้ว ปมที่ถูกลบจากกองซ้อนคือรากของต้นไม้นิพจน์

ดูตัวอย่างการสร้างต้นไม้นิพจน์จากนิพจน์ `5 y 4 * +` ในรูปที่ 9-7 เริ่มด้วยการพบว่า 5 เป็นตัวถูกดำเนินการ ก็สร้างปมใหม่แล้วใส่ลงกองซ้อน โดยให้ลูกของปมที่เป็นตัวถูกดำเนินการมีค่าเป็น `null` (ไม่ได้แสดงในรูป) ต่อมาพบ `y` และ `4` ก็ทำเช่นเดียวกัน คราวนี้พบ `*` ให้ลบสองปมออกมาเป็นลูกซ้ายและขวาของปมใหม่ของ `*` แล้วใส่กลับลงกองซ้อน เมื่อพบ + ก็ทำทำนองเดียวกัน สุดท้ายได้ปมบนกองซ้อนเป็นรากของต้นไม้นิพจน์ (ลบมาเก็บในตัวแปร `root` ดังรูปที่ 9-8)



รูปที่ 9-7 การใช้กองซ้อนช่วยสร้างต้นไม้นิพจน์จากนิพจน์เติมหลัง



รูปที่ 9-8 ต้นไม้นิพจน์ที่ได้จากตัวอย่างในรูปที่ 9-7

รหัสที่ 9-6 แสดงตัวสร้างของ `Expression` ซึ่งสร้างต้นไม้นิพจน์เก็บไว้ภายในอ็อบเจกต์ ตัวสร้างนี้รับรายการของพจน์ต่าง ๆ ที่แทนนิพจน์แบบเติมกลาง (บรรทัดที่ 2) เริ่มทำงานด้วยการแปลงนิพจน์ที่ได้รับให้เป็นนิพจน์แบบเติมหลัง (บรรทัดที่ 3) จากนั้นสร้างกองซ้อน แล้วเข้าวงวน (บรรทัดที่ 5) หยิบแต่ละพจน์มาพิจารณา (ตัวแปร `token`) ถ้า `token` เป็นตัวถูกดำเนินการก็ให้สร้างใบ (ซึ่งคือปมที่มีลูกซ้ายและขวาเป็น `null`) และเพิ่มใบนั้นลงกองซ้อนในบรรทัดที่ 8 แต่ถ้าเป็นตัวดำเนินการก็ให้ลบปมออกจากกองซ้อนตามจำนวนที่ตัวถูกดำเนินการต้องการ ในที่นี้เรามีแต่ตัวดำเนินการที่ต้องการตัวถูกดำเนินการสองตัว จึงลบมาต่อเป็นลูกซ้ายและขวาของปมใหม่ เมื่อประมวลผลทุกพจน์แล้วหลุดจากวงวน ก็ลบปมบนกองซ้อนมาตั้งเป็นรากของต้นไม้ในบรรทัดที่ 15

```
01 public class Expression extends BinaryTree {
02   public Expression(List infix) {
03     List postfix = infix2Postfix(infix);
04     Stack s = new ArrayStack();
05     for (int i = 0; i < postfix.size(); i++) {
06       String token = (String)postfix.get(i);
07       if (!isOperator(token)) {
08         s.push(new Node(token, null, null));
09       } else {
10         Node right = (Node) s.pop();
11         Node left = (Node) s.pop();
12         s.push(new Node(token, left, right));
13       }
14     }
15     root = (Node) s.pop();
16   }
```

เปลี่ยนเป็นนิพจน์เติมหลัง

push ถ้าพบ operand

พบ operator ให้ pop ออกมาสองปม มาผูกเป็นลูกของปมใหม่ แล้ว push กลับ

รากของต้นไม้นิพจน์อยู่ในกองซ้อน

รหัสที่ 9-6 ตัวสร้างของคลาส `Expression` ที่สร้างต้นไม้นิพจน์เก็บไว้ภายใน

บริการพื้นฐานของต้นไม้

บริการพื้นฐานที่น่าจะใช้ประโยชน์ได้ของต้นไม้ทั่วไป เห็นจะเป็นการหาลักษณะสมบัติของตัวต้นไม้ เช่น ความสูง จำนวนปม จำนวนใบ หรือบริการทำสำเนาต้นไม้ บริการคืนแถวลำดับที่เก็บข้อมูลของต้นไม้ บริการแวะผ่านต้นไม้ เป็นต้น หัวข้อนี้อธิบายรายละเอียดของบริการเหล่านี้กับต้นไม้แบบทวิภาคที่สร้างแบบง่ายสุด มีตัวโยงไปยังลูกซ้ายและขวา ไม่มีตัวโยงไปยังปมพ่อ

ก่อนจะนำเสนอบริการต่าง ๆ ของคลาส `BinaryTree` ขอเขียนรายละเอียดของตัวคลาสอีกครั้งในรหัสที่ 9-7 ให้สังเกตว่า ต่างกับที่เขียนไว้ในรหัสที่ 9-1 (เพราะเรายังไม่อยากลงรายละเอียดมากนักในตอนแรก) ต้องขอชี้แจงก่อนว่า `BinaryTree` เป็นคลาสที่เราตั้งใจให้เป็นคลาสแม่ของคลาสที่เก็บข้อมูลโดยใช้โครงสร้างต้นไม้แบบทวิภาค (เช่น คลาส `Expression`, `HuffmanTree`, `BSTree`, `AVLTree` เป็นต้น ที่จะได้นำเสนอต่อไป) เมท็อดส่วนใหญ่รวมถึงข้อมูลและคลาส `Node` จึงเป็นแบบ `protected` ซึ่งอนุญาตให้เฉพาะคลาสลูกเรียกใช้ได้ เราไม่ได้ประสงค์จะให้ใครมาสร้างอ็อบเจกต์ของ `BinaryTree` โดยตรง จึงเขียนให้ตัวสร้างเป็นแบบ `protected` เพื่อให้เรียกได้จากตัวสร้างของคลาสลูกเท่านั้น (บรรทัดที่ 13) เพราะตัวต้นไม้เองใช้การอะไรไม่ได้ จนกว่าจะตั้งกฎระเบียบการจัดเก็บข้อมูลและความสัมพันธ์ของปมพ่อ-ลูกว่าเป็นอย่างไร ซึ่งเป็นหน้าที่ของคลาสลูก (จะได้เห็นตัวอย่างต่อไป) และถ้าสังเกตที่คลาส `Node` จะพบว่า สมาชิกภายในเป็น `public` หมด ทั้งนี้ก็เพื่อให้คลาสลูกของ `BinaryTree` มีสิทธิ์ใช้สมาชิกต่าง ๆ ของ `Node` ได้

```
01 public class BinaryTree {
02   protected Node root;
03
04   protected static class Node {
05     public Object element;
06     public Node left;
07     public Node right;
08     public Node(Object e, Node l, Node r) {
09       element = e; left = l; right = r;
10     }
11     public boolean isLeaf(){return left==null&&right==null;}
12   }
13   protected BinaryTree() {}
..   protected int numNodes(Node r) { ... }
..   ...
     public Object[] toArray() { ... }
```

เป็น protected เพราะให้คลาสลูกเท่านั้นที่ใช้และ extends Node ได้

เป็น public เพราะคลาสลูกของ BinaryTree จะได้เรียกใช้ได้

เป็น public เพราะคลาสลูกใช้เป็นบริการสาธารณะได้ด้วย

รหัสที่ 9-7 `BinaryTree` คือคลาสแม่ของสารพัดคลาสที่มีโครงสร้างเป็นต้นไม้แบบทวิภาค

จาวาควบคุมการเข้าถึงสมาชิกต่าง ๆ ของคลาสอยู่สี่ระดับคือ

1. แบบ private ใช้ภายในคลาสตัวเองเท่านั้น
2. แบบ protected อนุญาตให้เฉพาะคลาสลูกเรียกใช้ได้ด้วย
3. แบบที่ให้เฉพาะคลาสเพื่อน ๆ ในแพกเกจเดียวกันใช้ได้ด้วย (แบบนี้ไม่ต้องใช้คำสำคัญใดกำกับตัวสมาชิก)
4. แบบ public อนุญาตให้คลาสใด ๆ ใช้ได้หมด

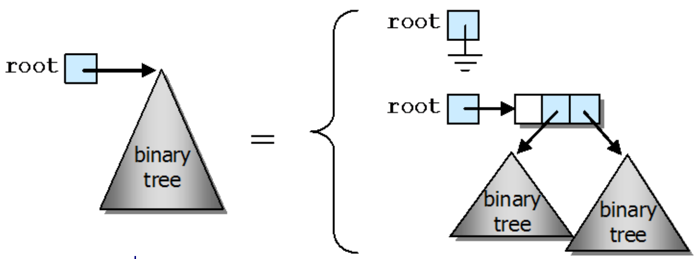
คลาส Node ในรหัสที่ 9-7 เป็นแบบ protected ดังนั้นเฉพาะคลาสลูกของ BinaryTree เท่านั้นที่ใช้คลาส Node ได้ ส่วนการที่ให้สมาชิกของ Node เป็น public นั้น เพื่อให้คลาสลูกของ BinaryTree ใช้สมาชิกของ Node ได้ เพราะถ้าเราให้สมาชิกภายใน Node เป็น protected จะหมายความว่า เฉพาะคลาสลูกของ Node จึงจะใช้ได้ แสดงว่า คลาสลูกของ BinaryTree ใช้สมาชิกภายใน Node ไม่ได้เพราะไม่ใช่ลูกของ Node และถ้าให้เป็นแบบเพื่อน ก็แสดงว่า คลาสลูกของ BinaryTree ที่ไม่ได้อยู่ในแพกเกจเดียวกันก็ใช้ไม่ได้อีก ดังนั้นจึงต้องให้สมาชิกของ Node เป็น public

ส่วนการให้ตัวสร้าง (constructor) เป็น protected นั้นก็เพื่อให้เฉพาะคลาสลูกของ BinaryTree เรียกด้วยคำสั่ง super ในตัวสร้างของคลาสลูกเท่านั้น จึงเป็นการป้องกันการสร้างอ็อบเจกต์ของ BinaryTree (นอกจากนี้เราสามารถป้องกันการสร้างอ็อบเจกต์ของคลาส ด้วยการให้คลาสเป็น abstract หรือไม่ก็ให้ตัวสร้างเป็นแบบ private แต่เราไม่เลือกใช้วิธีทั้งสองนี้เพราะไม่ตรงวัตถุประสงค์)

โครงสร้างเวียนเกิดของต้นไม้แบบทวิภาค

ปกติเรานิยามต้นไม้แบบทวิภาคว่า เป็นโครงสร้างซึ่งประกอบด้วยปม โดยที่แต่ละปมมีได้ไม่เกินสองลูก ทุกปมมีปมพ่อหนึ่งปม จะยกเว้นก็มีเพียงปมรากเท่านั้นที่ไม่มีปมพ่อ เราถือว่า รากของต้นไม้คือตัวแทนต้นไม้ ดังจะเห็นได้จากคลาส BinaryTree ที่เขียนในหัวข้อที่แล้ว มีเพียงตัวแปร root เท่านั้นที่เป็นตัวโยงไปสู่ปมรากของต้นไม้ เราไม่ได้เก็บตำแหน่งปมอื่นใดเลยของต้นไม้นอกจากราก ถ้าเรารู้ว่า รากอยู่ที่ใด เราก็สามารถเข้าถึงปมอื่น ๆ ของต้นไม้นั้นได้

จะขอนิยามโครงสร้างต้นไม้แบบทวิภาคในอีกลักษณะที่เรียกว่า แบบเวียนเกิด คือนิยามต้นไม้แบบทวิภาคต้นใหญ่ด้วยต้นไม้แบบทวิภาคต้นย่อย (ดูรูปที่ 9-9) จะได้ว่า ต้นไม้แบบทวิภาคคือต้นไม้ว่าง (ซึ่งคือ null) หรือไม่ก็คือโครงสร้างที่ประกอบด้วยปมหนึ่งปมเรียกว่า ราก ซึ่งมีลูกซ้าย และลูกขวาเป็นต้นไม้ย่อยแบบทวิภาคทั้งสองต้น



รูปที่ 9-9 โครงสร้างเวียนเกิดของต้นไม้แบบทวิภาค

int numNodes(Node r)

ด้วยนิยามของต้นไม้แบบทวิภาคในลักษณะเวียนเกิด จำนวนปมของต้นไม้ย่อมเป็น 0 ถ้าเป็นต้นไม้ว่าง แต่ถ้าไม่เป็น ก็ย่อมเท่ากับ 1 + จำนวนปมของต้นไม้ย่อยทางซ้ายของราก + จำนวนปมของต้นไม้ย่อยทางขวาของราก เขียนได้ดังนี้

$$n(r) = \begin{cases} 0 & r = null \\ 1 + n\big(L(r)\big) + n\big(R(r)\big) & r \neq null \end{cases}$$

โดยที่ $n(r)$ แทนจำนวนปมของต้นไม้ที่มี r เป็นราก $L(r)$ คือปมลูกซ้ายของ r และ $R(r)$ คือปมลูกขวาของ r จากนิยามข้างต้นนี้เขียนได้เป็นเมท็อด `numNodes(Node r)` เพื่อคืนจำนวนปมของต้นไม้แบบทวิภาคที่มี `r` เป็นรากดังรหัสที่ 9-8

```
public class BinaryTree {
  ...
  protected int numNodes(Node r) {
    if (r == null) return 0;
    return 1 + numNodes(r.left) + numNodes(r.right);
  }
```

รหัสที่ 9-8 เมท็อด `numNodes(r)` คืนจำนวนปมของต้นไม้ที่มี `r` เป็นราก

เมท็อด `numNodes(Node r)` เป็นเมท็อดที่เรียกตัวเองแบบเวียนเกิด ซึ่งเขียนตามนิยามของ $n(r)$ คือถ้า `r` เป็น `null` ก็ให้คืน 0, ถ้าไม่เป็น `null` ก็เรียก `numNodes(r.left)` เพื่อหาจำนวนปมของลูกต้นซ้าย และเรียก `numNodes(r.right)` เพื่อหาของลูกต้นขวา นำผลของทั้งสองต้นย่อยมารวมกันแล้วบวกเพิ่มอีกหนึ่งซึ่งแทนปม `r` เป็นจำนวนปมรวมของต้น `r`

int height(Node r)

ความสูงของต้นไม้ก็คือความยาวของวิถีจากรากถึงใบที่อยู่ลึกสุด อ่านนิยามความสูงแบบนี้อาจทำให้หลงประเด็นไปเขียนแจกแจงวิถีจากรากถึงทุก ๆ ใบ แล้วหาค่ามากสุด แต่ถ้าเราคิดถึงโครงสร้างต้นไม้ในลักษณะแบบเวียนเกิด จะได้ว่า ความสูงของต้นไม้ย่อมเท่ากับความสูงของต้นย่อยที่เป็นลูกทางซ้ายหรือไม่ก็ความสูงของต้นย่อยที่เป็นลูกทางขวา เลือกต้นที่สูงกว่าแล้วบวกเพิ่มอีกหนึ่ง ต้นไม้ที่มีปมเดียวก็สูงเป็นศูนย์ ดังนั้นต้นไม้ว่างก็ต้องสูงเป็น –1 เขียนเป็นความสัมพันธ์เวียนเกิดได้ดังนี้

$$h(r) = \begin{cases} -1 & r = null \\ 1 + \max\Big(h\big(L(r)\big), h\big(R(r)\big)\Big) & r \neq null \end{cases}$$

โดยที่ $h(r)$ แทนความสูงของต้นไม้ที่มี r เป็นราก เขียนเป็นเมท็อด `height` ได้ดังรหัสที่ 9-9

```
protected int height(Node r) {
  if (r == null) return -1;
  return 1 + Math.max(height(r.left), height(r.right));
}
```

รหัสที่ 9-9 เมท็อด `height(r)` คืนความสูงของต้นไม้ที่มี `r` เป็นราก

เมท็อด `height(Node r)` เป็นเมท็อดที่เรียกตัวเองแบบเวียนเกิด ซึ่งเขียนตามนิยามของ $h(r)$ คือถ้า `r` เป็น `null` ก็ให้คืน –1, ถ้าไม่เป็น `null` ก็เรียก `height(r.left)` เพื่อหาความสูงของลูกต้นซ้าย และเรียก `height(r.right)` เพื่อหาของลูกต้นขวา จากนั้นเลือกค่ามาก นำมาบวกเพิ่มอีกหนึ่งเป็นความสูงของต้น `r`

int numLeaves(Node r)

ใบก็คือปมที่ไม่มีลูก หรือกล่าวได้ว่า คือปมในต้นไม้ที่ตัวโยงไปยังลูกทั้งสองเป็น `null` ดังนั้นจำนวนใบในต้นไม้ก็ย่อมเท่ากับจำนวนใบในลูกต้นซ้ายรวมกับจำนวนใบในลูกต้นขวา ต้นไม้ว่างไม่มีใบ และต้นไม้ที่มีปมเดียวมีหนึ่งใบ เขียนนิยามได้ดังนี้

$$l(r) = \begin{cases} 0 & r = null \\ 1 & L(r) = null \ \ and \ \ R(r) = null \\ l\big(L(r)\big) + l\big(R(r)\big) & otherwise \end{cases}$$

โดยที่ $l(r)$ แทนความสูงของต้นไม้ที่มี r เป็นราก เขียนเป็นเมท็อดได้ดังรหัสที่ 9-10

```
protected int numLeaves(Node r) {
  if (r == null) return 0;
  if (r.isLeaf()) return 1;
  return numLeaves(r.left) + numLeaves(r.right);
}
```

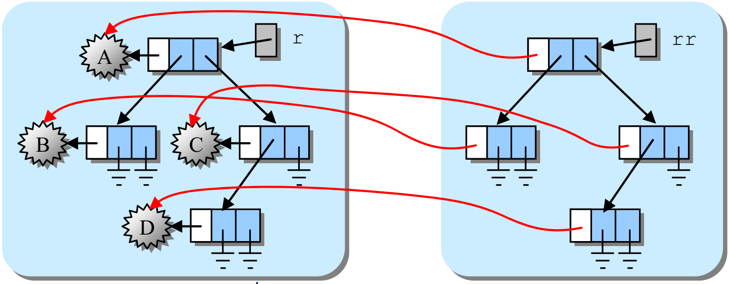
isLeaf เป็นเมท็อดของคลาส Node

รหัสที่ 9-10 เมท็อด `numLeaves(r)` คืนจำนวนใบของต้นไม้ที่มี `r` เป็นราก

เมท็อด `numLeaves(Node r)` เป็นเมท็อดซึ่งเขียนตามนิยามของ $l(r)$ คือถ้า `r` เป็น `null` ก็ให้คืน 0, ถ้าเป็นใบ ให้คืน 1, ถ้าไม่ใช่ใบ ก็ให้หาผลรวมของจำนวนใบของลูกต้นซ้ายและต้นขวาด้วย `numLeaves(r.left)` และ `numLeaves(r.right)` เป็นจำนวนใบของต้น `r`

Node copy(Node r)

ในบางครั้งการประมวลผลต้นไม้อาจมีการเปลี่ยนแปลงตัวต้น แต่เราก็ต้องการต้นเดิมไว้ด้วย จึงจำเป็นต้องทำสำเนาทั้งต้นไว้ แล้วค่อยนำอีกต้นไปประมวลผล การทำสำเนาต้นไม้นี้หมายถึงการทำสำเนาปมทุกปม มีตัวโยงไปยังข้อมูลตามปมร่วมกับต้นเดิม แต่ตัวโยงไปยังปมพ่อ-ลูกให้โยงไปยังปมใหม่โดยให้โครงสร้างเหมือนของต้นเดิมทุกประการ ดังตัวอย่างในรูปที่ 9-10



รูปที่ 9-10 การทำสำเนาต้นไม้

รหัสที่ 9-11 แสดงเมท็อด `copy(Node r)` มีหน้าที่สร้างและคืนต้นไม้ต้นใหม่ให้เหมือนกับต้นไม้ที่มีปม `r` เป็นราก โดยจะคืน `null` ซึ่งแทนต้นไม้ว่าง ถ้า `r` มีค่าเป็น `null` ถ้าไม่ใช่ `null` ก็จะสร้างปมใหม่ซึ่งมีข้อมูลคือ `r.element` ส่วนลูกซ้ายและลูกขวาก็คือผลจากการทำสำเนาลูกต้นซ้ายและลูกต้นขวาของ `r` ด้วย `copy(r.left)` และ `copy(r.right)` แล้วคืนปมใหม่ที่สร้างนี้เป็นรากของต้นไม้ที่สำเนาจากต้น `r`

```
  protected Node copy(Node r) {
    if (r == null) return null;
    return new Node(r.element, copy(r.left), copy(r.right));
  }
```

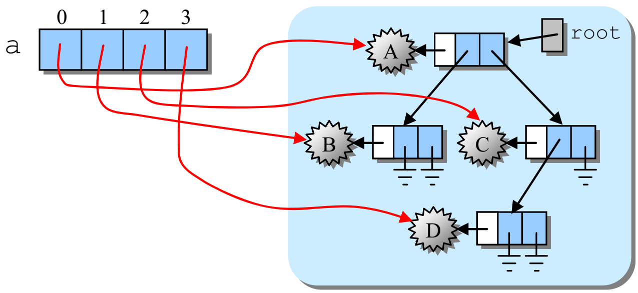
copy ทำสำเนาเฉพาะปม

รหัสที่ 9-11 เมท็อด `copy(r)` คืนสำเนาของต้นไม้ที่มี `r` เป็นราก

Object[ ] toArray()

เมื่อใดที่ผู้ใช้ต้องการประมวลผลข้อมูลต่าง ๆ ที่เก็บในโครงสร้างข้อมูล เช่น คอลเล็กชัน หรือเซต `toArray` มักเป็นบริการที่ ถ้ามีให้ใช้ จะใช้ง่าย เพราะเราคุ้นเคยกับการประมวลผลด้วยแถวลำดับ `toArray` คือบริการที่คืนแถวลำดับที่มีขนาดเท่ากับจำนวนข้อมูล โดยแต่ละช่องอ้างอิงข้อมูลแต่ละตัวในกลุ่มข้อมูลที่จัดเก็บ ตัวอย่างเช่น ในรูปที่ 9-11 แสดงแถวลำดับ `a` ที่ได้จาก `toArray` ของต้นไม้ทางขวา ข้อมูลในต้นไม้มี 4 ตัวก็ได้แถวลำดับ 4 ช่อง แต่ละช่องอ้างอิงไปยังตัวข้อมูลที่เก็บในต้นไม้ ปัญหาก็คือจะเขียน `toArray` ให้กับโครงสร้างต้นไม้ได้อย่างไร ?

เราต้องการให้ผู้ใช้เรียกใช้บริการได้ง่าย ๆ ถ้าเขามีต้นไม้แบบทวิภาค `t` ก็เพียงแค่เรียก `t.toArray()` ก็จะได้แถวลำดับคืนกลับไป ดังนั้นเราจะเขียนเมท็อด `toArray()` ที่เตรียมแถวลำดับผลลัพธ์ แล้วส่งไปให้อีกเมท็อดชื่อ `toArray(r,a,k)` ที่นำข้อมูลจากต้นไม้ที่มี `r` เป็นปมราก เติมลงในแถวลำดับ `a` เริ่มที่ช่อง `k` เมื่อเติมเสร็จจะคืนเลขที่ช่องถัดไปในแถวที่ยังไม่ได้เติมข้อมูล เช่น ถ้าต้นไม้ที่มี `r` เป็นราก มีข้อมูลอยู่ 20 ตัว เมื่อเรียก `toArray(r,a,k)` จะเติมข้อมูล 20 ตัวนี้ไว้ที่ `a[k]`, `a[k+1]` ไปจนถึง `a[k+19]` แล้วคืนค่าของ `k+20` กลับให้ผู้เรียก



รูปที่ 9-11 การสร้างแถวลำดับที่เก็บข้อมูลทุกตัวในต้นไม้

รหัสที่ 9-12 แสดงรายละเอียด `toArray()` เริ่มทำงานด้วยการสร้างแถวลำดับ `a` ให้มีขนาดเท่ากับจำนวนปม จากนั้นเรียก `toArray(root,a,0)` โดย `toArray(r,a,k)` ทำงานแบบเวียนเกิด คือจะคืน `k` กลับไปทันทีถ้า `r` เป็น `null` เพราะไม่มีอะไรต้องเติม แต่ถ้า `r` ไม่เป็น `null` ก็นำ `r.element` ใส่ในช่อง `a[k]` แล้วเพิ่มค่า `k` อีกหนึ่ง (บรรทัดที่ 38) จากนั้นก็นำข้อมูลในลูกต้นซ้ายมาเติมใส่ในแถวลำดับด้วย `toArray(r.left,a,k)` ได้ผลกลับมาเก็บใส่ `k` แล้วทำเช่นเดียวกันกับข้อมูลในลูกต้นขวาด้วย `toArray(r.right,a,k)` แล้วคืนค่าที่ได้รับกลับไปเป็นอันเสร็จภารกิจ ให้สังเกตว่า ไม่มีข้อกำหนดใด ๆ ที่ระบุให้ต้องนำ `r.element` ไปเติม ก่อนเติมข้อมูลในต้นซ้าย แล้วค่อยเติมข้อมูลในต้นขวา จะสลับลำดับแบบใดก็ได้

```
01 public class BinaryTree {
02   protected Node root;
..   ...
31   public Object[] toArray() {
32     Object[] a = new Object[numNodes(root)];
33     toArray(root, a, 0);
34     return a;
35   }
36   private int toArray(Node r, Object[] a, int k) {
37     if (r == null) return k;
38     a[k++] = r.element;
39     k = toArray(r.left, a, k);
40     return toArray(r.right, a, k);
41   }
..   ...
```

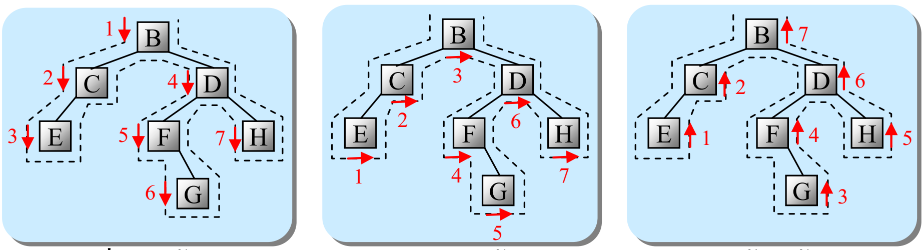
k คือช่องของ a ที่เริ่มเก็บข้อมูล

คืนเลขที่ช่องถัดไปในแถวลำดับที่ยังไม่ได้เติม

รหัสที่ 9-12 เมท็อด `toArray` เพื่อสร้างแถวลำดับที่เก็บข้อมูลทุกตัวในต้นไม้

การแวะผ่านต้นไม้

การแวะผ่านต้นไม้ (tree traversal) เป็นกระบวนการเข้าถึงปมต่าง ๆ ในต้นไม้ ปมละหนึ่งครั้งอย่างมีระเบียบ ทำให้สามารถประมวลผลต้นไม้ได้อย่างมีระบบ หัวข้อนี้นำเสนอการแวะผ่านแบบมาตรฐานสามแบบ ที่เขียนง่ายและใช้เวลาการทำงานแปรตามจำนวนปมคือ การแวะผ่านแบบ*ก่อนลำดับ* (preorder) *ตามลำดับ* (inorder) และ*หลังลำดับ* (postorder) รูปที่ 9-12 แสดงลำดับของปมที่ได้จากการแวะผ่านทั้งสามแบบในต้นไม้ต้นหนึ่ง การหาลำดับการแวะผ่านทำได้ง่ายด้วยการลากเส้นเริ่มที่ราก (เส้นประที่แสดงในรูปที่ 9-12) ลากเส้นทางซ้ายของกิ่งลงไปเรื่อย ๆ เมื่อถึงใบก็ลากเส้นอ้อมตามปมกลับขึ้นทางขวาของกิ่ง ลากเส้นในลักษณะนี้จนกลับขึ้นไปทางขวาของราก ลำดับของปมที่ถูกแวะผ่านแบบก่อนลำดับคือลำดับการลากผ่านด้านซ้ายของปม ของแบบตามลำดับคือลำดับการลากผ่านด้านล่างของปม และของแบบหลังลำดับคือลำดับการลากผ่านด้านขวาของปม ดังตัวอย่างในรูปที่ 9-12



ก่อนลำดับ B, C, E, D, F, G, H

ตามลำดับ E, C, B, F, G, D, H

หลังลำดับ E, C, G, F, H, D, B

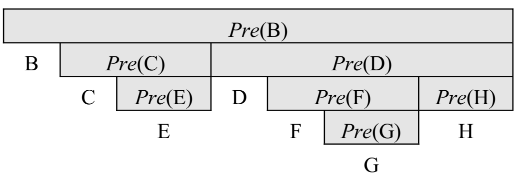
รูปที่ 9-12 ตัวอย่างการแวะผ่านต้นไม้

การแวะผ่านแบบก่อนลำดับ

แล้วจะเขียนเป็นโปรแกรมได้อย่างไร ? ขอเขียนเป็นความสัมพันธ์เวียนเกิดก่อนนำไปสู่การเขียนตัวเมท็อด คือถ้าต้องการแวะผ่านต้น *r* แบบ<u>ก่อน</u>ลำดับ ก็ให้แวะปม *r* <u>ก่อน</u> จากนั้นแวะผ่านลูกต้นซ้ายของ *r* แบบก่อนลำดับ แล้วแวะผ่านลูกต้นขวาของ *r* แบบก่อนลำดับ กำหนดให้ $Pre(r)$ คือลำดับของปมที่ได้จากการแวะผ่านต้น *r* แบบก่อนลำดับ จะได้ว่า

$$Pre(r) = \begin{cases} empty\ sequence & if\ \ r = null \\ r,\ Pre(L(r)),\ Pre(R(r)) & if\ \ r \neq null \end{cases}$$

การแวะผ่านแบบก่อนลำดับในรูปที่ 9-12 แสดงได้ผังข้างล่างนี้ ผลที่ได้คือลำดับ B, C, E, D, F, G, H

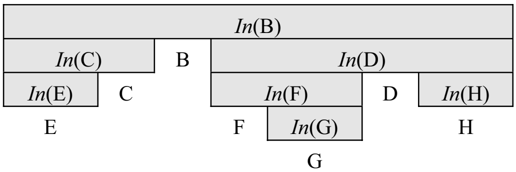


การแวะผ่านแบบตามลำดับ

การแวะผ่านแบบตามลำดับ มีขั้นตอนการทำงานคล้ายกับแบบก่อนลำดับ จะต่างกันก็ตรงลำดับที่เราแวะปมราก คือถ้าทำกับต้นไม้ที่มี r เป็นราก จะเริ่มด้วยการแวะผ่านลูกต้นซ้ายของ r แบบตามลำดับ ตามด้วยการแวะปม r แล้วจึงค่อยแวะผ่านลูกต้นขวาของ r แบบตามลำดับ กำหนดให้ $In(r)$ คือลำดับของปมในการแวะผ่านต้นไม้ที่มี r เป็นรากแบบตามลำดับจะได้ว่า

$$In(r) = \begin{cases} empty\ sequence & if\ \ r = null \\ In(L(r)),\ r,\ In(R(r)) & if\ \ r \neq null \end{cases}$$

การแวะผ่านแบบตามลำดับในรูปที่ 9-12 แสดงได้ผังข้างล่างนี้ ผลที่ได้คือลำดับ E, C, B, F, G, D, H

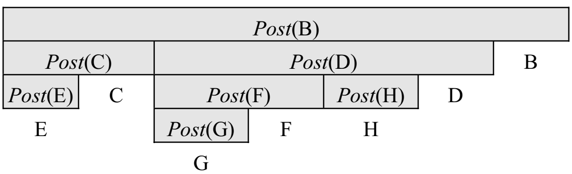


การแวะผ่านแบบหลังลำดับ

ก็คงพอเดากันได้ว่า การแวะผ่านแบบหลังลำดับ จะเป็นเช่นใด ให้ $Post(r)$ คือลำดับของปมในการแวะผ่านในต้นไม้ที่มี r เป็นรากแบบหลังลำดับ จะได้ว่า

$$Post(r) = \begin{cases} empty\ sequence & if\ \ r = null \\ Post(L(r)),\ Post(R(r)), r & if\ \ r \neq null \end{cases}$$

การแวะผ่านแบบหลังลำดับในรูปที่ 9-12 แสดงได้ผังข้างล่างนี้ ผลที่ได้คือลำดับ E, C, G, F, H, D, B



การใช้ตัวเยี่ยมชมในการแวะผ่านต้นไม้

รหัสที่ 9-13 แสดงเมท็อดการแวะผ่านต้นไม้แบบก่อนลำดับ เมท็อด `preOrder(r)` จะคืนการทำงานทันทีถ้า `r` เป็น `null` ถ้าไม่เป็น `null` ก็จะแวะข้อมูลที่ปม `r` ด้วย `visit(r.element)` ตามด้วยการแวะผ่านลูกต้นซ้ายแบบก่อนลำดับด้วย `preOrder(r.left)` และแวะลูกต้นขวาแบบก่อนลำดับด้วย `preOrder(r.right)`

```
  protected void preOrder(Node r) {
```

```
      if (r == null) return;
      visit(r.element);
      preorder(r.left);
      preorder(r.right);
    }
```

pre แปลว่า ก่อน หมายถึงให้ visit(r.element) ก่อนไปแวะผ่านลูก ๆ

รหัสที่ 9-13 เมท็อด preOrder แวะผ่านต้นไม้แบบก่อนลำดับ

preOrder ในรหัสที่ 9-13 ดูแปลก ๆ ไม่รู้ว่าทำอะไร จะทำอะไรก็ขึ้นกับว่า visit ทำอะไร และถ้า visit เขียนให้ทำอะไร ก็คงทำแบบนั้นตลอด น่าจะมีวิธีเขียนให้เมท็อด visit เปลี่ยนพฤติกรรมได้ตามที่ผู้ใช้ต้องการ ขอนำเสนอกลวิธีหนึ่งที่เมท็อด visit ไม่ได้เขียนตายตัวไว้ในคลาสของต้นไม้ แต่ย้ายไปเขียนไว้กับอ็อบเจกต์ชนิดพิเศษที่เรียกว่า "ตัวเยี่ยมชม" (visitor) โดยผู้ใช้เป็นผู้สร้างแล้วส่งมาให้ preOrder เรียกต่ออีกทีทุกครั้งที่แวะปมในต้นไม้

ขอนิยามคลาส Visitor (รหัสที่ 9-14) ไว้สร้างตัวเยี่ยมชม คลาสนี้บังคับคลาสลูกให้เขียนเมท็อด visit(Object e) มีบริการ done ให้เรียกเพื่อแจ้งความจำนงว่า ต้องการยุติการแวะผ่าน และ isDone ให้ตรวจสอบว่า ต้องการยุติการแวะผ่านหรือไม่ จากนั้นเขียนให้ preOrder รับตัวเยี่ยมชม ดังแสดงในรหัสที่ 9-15 เมื่อใดพบปม r ที่ต้องแวะก็เรียก v.visit(r.element) เพื่อไปทำงานที่เมท็อด visit ของตัวเยี่ยมชมที่รับมา และให้คืนการทำงานทันทีที่ isDone เป็นจริง

```
public abstract class Visitor {
  private boolean done = false;
  public void done() {done = true;}
  public boolean isDone() {return done;}
  public abstract void visit(Object e);
}
```

e คือข้อมูลที่ปมของต้นไม้ที่ถูกแวะ

รหัสที่ 9-14 คลาส Visitor สำหรับสร้างตัวเยี่ยมชม

```
  protected void preOrder(Node r, Visitor v) {
    if (r == null || v.isDone()) return;
    v.visit(r.element);
    preOrder(r.left, v);
    preOrder(r.right, v);
  }
```

รับตัวเยี่ยมชมที่มีเมท็อด visit ให้เรียก

เรียก visit ของตัวเยี่ยมชม ส่งข้อมูลที่ r ไปประมวลผล

รหัสที่ 9-15 การใช้ตัวเยี่ยมชมไว้เรียก visit ตอนแวะผ่านปมของต้นไม้

เพื่อให้เห็นเป็นรูปธรรมถึงการใช้งานตัวเยี่ยมชม รหัสที่ 9-16 แสดงวิธีการเขียน toArray ที่ต่างจากที่ได้นำเสนอมา เริ่มด้วยการสร้างแถวลำดับ a เตรียมไว้ก่อน จากนั้นสร้างตัวเยี่ยมชมซึ่งมีเมท็อด visit ที่นำข้อมูลของปมที่ถูกแวะมาใส่ในแถวลำดับ โดยมีตัวแปร k เก็บตำแหน่งช่องที่จะเติมข้อมูลตัวถัดไป เมื่อสร้างตัวเยี่ยมชม v เสร็จก็เรียก preOrder(root,v) ให้เริ่มแวะผ่านต้นไม้เริ่มที่รากบนสุด หลังทำเสร็จก็คืน a ที่เก็บข้อมูลระหว่างการแวะผ่านต้นไม้

```
public Object[] toArray() {
  final Object[] a = new Object[numNodes()];
  Visitor v = new Visitor() {
                int k = 0;
                public void visit(Object e) {
                  a[k++] = e;
                }};
  preOrder(root, v);
  return a;
}
```

k เก็บเลขช่องของแถวลำดับที่พร้อมใส่ข้อมูลตัวถัดไป

แวะปมใหม่ก็นำข้อมูลในปมนั้นไปใส่ในแถวลำดับ

สั่งให้แวะผ่าน แล้วก็จะมาทำที่ visit ทุกครั้งที่พบปมใหม่

รหัสที่ 9-16 เมท็อด `toArray` ใช้ตัวเยี่ยมชมปมเป็นตัวนำข้อมูลใส่แถวลำดับ

การเขียนคลาสในจาวากระทำได้หลายแบบ แบบที่เราเขียนให้กับตัวเยี่ยมชมในรหัสที่ 9-16 นั้นเป็นแบบที่เรียกว่า *คลาสภายในนิรนาม* (anonymous inner class) คือเป็นแบบที่เรานิยามคลาสไว้ภายในเมท็อด ไม่ต้องตั้งชื่อคลาส เขียนเสร็จก็สร้างอ็อบเจกต์ทันที โดยมีกฎว่า การเขียนคลาสแบบนี้ ต้องระบุชื่อคลาสพ่อของคลาสใหม่ที่จะเขียน หรือไม่ก็ต้องระบุชื่ออินเทอร์เฟซที่คลาสใหม่นี้ implements ในกรณีของรหัสที่ 9-16 เราเขียน new Visitor() { ... } หมายความว่า ต้องการสร้างอ็อบเจกต์ของคลาสใหม่ (ที่ไม่อยากตั้งชื่อ) ซึ่ง extends Visitor ภายในเครื่องหมาย { และ } จึงต้องเขียนรายละเอียดของเมท็อด visit เราเขียนอะไรก็ได้ภายในคลาสนิรนามนี้เหมือนคลาสทั่วไป เช่น ในรหัสที่ 9-16 มีตัวแปร k กำกับอ็อบเจกต์ เพื่อเก็บเลขช่องของแถวที่เราพร้อมเติมข้อมูลใหม่ จะมีอยู่สิ่งเดียวที่เขียนไม่ได้ในคลาสนิรนามก็คือ constructor เพราะจะเขียนตัวสร้างก็ต้องรู้ชื่อคลาส แต่คลาสแบบนี้ไม่มีชื่อ เลยเขียนไม่ได้ ให้สังเกตด้วยว่า คลาสนี้นิยามอยู่ภายในเมท็อด มีสิทธิ์ใช้ตัวแปรของเมท็อดได้ด้วย แต่มีข้อจำกัดเล็กน้อยในจาวาว่า จะใช้ได้เฉพาะกับตัวแปรที่เป็นแบบ final เท่านั้น เพราะว่า ตัวแปรของเมท็อดจะหายเมื่อเมท็อดเลิกทำงาน แต่อ็อบเจกต์ของคลาสภายในที่ใช้ตัวแปรอาจยังอยู่แล้วจะทำอย่างไร ? จาวาแก้ปัญหานี้ด้วยการบังคับให้ตัวแปรนั้นต้องเป็น final แล้วระบบจะทำสำเนาค่าของตัวแปรนั้นเก็บใส่อ็อบเจกต์ของคลาสภายในไว้ใช้แทนการใช้ตัวแปรของเมท็อดนั้น จึงต้องเขียนให้แถวลำดับ a ใน toArray เป็นแบบ final ทำให้เราเปลี่ยน a ไม่ได้ แต่ยังเปลี่ยนช่องต่าง ๆ ใน a ได้

รหัสที่ 9-17 แสดงรายละเอียดของเมท็อด inOrder และ postOrder มีการทำงานคล้ายกับของ preOrder ต่างกันก็เพียงตำแหน่งของคำสั่งการแวะปม v.vist(r.element)

```
protected void inOrder(Node r, Visitor v) {
  if (r == null || v.isDone()) return;
  inOrder(r.left, v);
  v.visit(r.element);
  inOrder(r.right, v);
}
protected void postOrder(Node r, Visitor v) {
  if (r == null || v.isDone()) return;
  postOrder(r.left, v);
  postOrder(r.right, v);
  v.visit(r.element);
}
```

แวะปม หลังแวะผ่านต้นซ้ายเสร็จ แต่ก่อนแวะต้นขวา

แวะปม หลังจากแวะผ่านต้นลูก ๆ แล้ว

รหัสที่ 9-17 การแวะผ่านต้นไม้ด้วยเมท็อด `inOrder` และ `postOrder` โดยใช้ตัวเยี่ยมชม

เมื่อมีเมท็อดสำหรับแวะผ่านที่ต้นไม้ย่อยใด ๆ แล้ว ก็สามารถให้บริการสาธารณะเพื่อแวะผ่านทั้งต้น ด้วยเมท็อดที่แสดงในรหัสที่ 9-18 ที่ผู้ใช้สามารถเรียกใช้กับตัวต้นไม้ได้เลย

```
  public void preOrder(Visitor v) {
    preOrder(root, v);
  }
  public void inOrder(Visitor v) {
    inOrder(root, v);
  }
  public void postOrder(Visitor v) {
    postOrder(root, v);
  }
```

รหัสที่ 9-18 เมท็อดสาธารณะสำหรับการแวะผ่านต้นไม้ทั้งต้น

มาดูกันอีกสักตัวอย่าง เรากำลังเขียนโปรแกรมหนึ่งในรหัสที่ 9-19 มีการใช้ต้นไม้นิพจน์ และภายในเมท็อดหนึ่งต้องตรวจสอบว่า นิพจน์นี้มี 0 อยู่หรือไม่ แต่มาพบว่า คลาส `Expression` ไม่มีบริการค้นหาข้อมูลในนิพจน์เลย จึงจำเป็นต้องเขียนเอง เนื่องจาก `Expression` เป็นคลาสลูกของ `BinaryTree` ซึ่งให้บริการแวะผ่านต้นไม้ด้วย จึงขอยืมการแวะผ่านนี้มาค้นข้อมูล โดยสร้างตัวเยี่ยมชม ที่มี `visit` ตรวจสอบว่า ข้อมูลของปมที่ส่งมาให้คือ "0" หรือไม่ ถ้าใช่ ก็จำผลลัพธ์ว่าได้พบแล้ว (โดยเก็บในแถวลำดับขนาดช่องเดียวแบบ `final`) พร้อมฝากบอกตัวเยี่ยมชมว่า ต้องการให้ยุติการค้นด้วยคำสั่ง `done()` (จำได้ไหม ? `done` เป็นเมท็อดในคลาส `Visitor`) เมื่อสร้างตัวเยี่ยมชมแล้ว ก็สั่งให้เริ่มแวะผ่านต้นไม้ แวะเสร็จก็นำผลลัพธ์มาใช้

```
public class SomeApplication {
  ...
  public void doSomething(Expression expr) {
    final boolean[] found = new boolean[1];
    Visitor v = new Visitor() {
                public void visit(Object e) {
                  if (e.equals("0")) {found[0]=true; done();}
                }};
    expr.preOrder(v);
    if (found[0]) ...
```

ใช้แถวลำดับเพราะ inner class จะได้เปลี่ยนแปลงข้อมูลได้

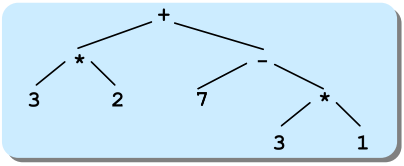
บอกให้ยุติการค้นได้

ความจริงไม่ต้องใช้ found เลยก็ได้ ใช้ v.isDone() แทนเพื่อตรวจสอบว่าค้นพบหรือไม่ก็ได้

รหัสที่ 9-19 การใช้การแวะผ่านเพื่อค้นหาข้อมูลในต้นไม้

การใช้การแวะผ่านต้นไม้มิได้ถูกจำกัดอยู่เพียงแค่กับตัวเยี่ยมชมเท่านั้น บางครั้งเราเขียนการดำเนินการบางอย่างโดยยืมโครงร่างการทำงานของการแวะผ่าน ตัวอย่างเช่น ถ้าเรามีต้นไม้นิพจน์ที่ตัวถูกดำเนินการเป็นค่าคงตัวทั้งหมด แล้วอยากทราบว่า นิพจน์นี้คำนวณแล้วจะได้ค่าเท่าใด เช่นถ้าต้องการคำนวณค่าของต้นไม้นิพจน์ในรูปที่ 9-13 (ซึ่งจะได้ค่าเป็น 10) จะคำนวณอย่างไร ? ให้สังเกตว่า เราจะคำนวณ + ที่รากของรูปที่ 9-13 ได้ก็ต้องคำนวณค่าของต้นไม้ย่อยทางซ้ายกับของทางขวาให้เสร็จเสียก่อน จึงตรงกับการแวะผ่านแบบหลังลำดับ เขียนได้เป็นเมท็อด `eval` ดังรหัสที่ 9-20 ซึ่งมี

`eval(Node r)` คำนวณค่าของต้นไม้ที่มี `r` เป็นราก ถ้า `r` เป็น `null` ก็คืน 0 ถ้า `r` เป็นใบ ก็แสดงว่าต้องเป็นค่าคงตัว (ตามสมมติฐาน) ก็ให้เปลี่ยน `r.element` จาก `String` เป็น `double` แล้วคืนค่ากลับไป แต่ถ้าไม่ใช่ทั้งสองกรณีข้างต้น ก็ย่อมต้องเป็นตัวดำเนินการ ก็ให้ไปคำนวณหาค่าของต้นไม้ย่อยทางซ้าย และทางขวาเก็บผลในตัวแปร `vLeft` และ `vRight` เพื่อนำมาคำนวณผลตามตัวดำเนินการที่เก็บในปม `r`



รูปที่ 9-13 ต้นไม้นิพจน์แทน (3*2)+(7-(3*1)) = 10

```
public class Expression extends BinaryTree {
  ...
  public double eval() {
    return eval(root);
  }
  private double eval(Node r) {
    if (r == null) return 0;
    if (r.isLeaf())
      return Double.parseDouble((String) r.element);
    double vLeft = eval(r.left);
    double vRight = eval(r.right);
    if (r.element.equals("+")) return vLeft + vRight;
    if (r.element.equals("-")) return vLeft - vRight;
    if (r.element.equals("*")) return vLeft * vRight;
    if (r.element.equals("/")) return vLeft / vRight;
    if (r.element.equals("^")) return Math.pow(vLeft,vRight);
    throw new IllegalStateException();
  }
```

ข้อมูลที่เก็บตามปมในต้นไม้เป็นสตริง ต้องเปลี่ยนเป็น double จะได้คำนวณได้

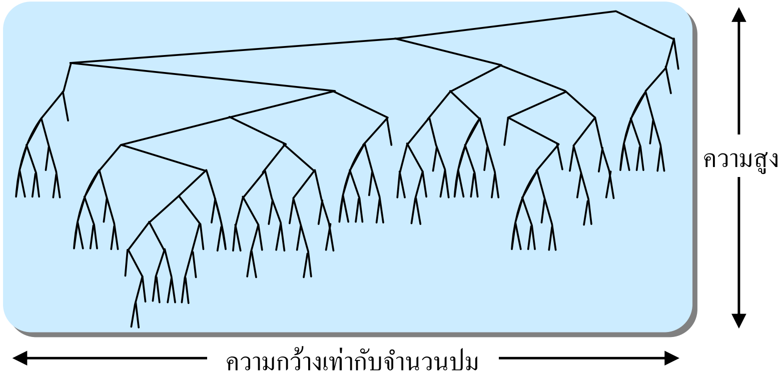
ต้องคำนวณค่าของลูก ๆ ก่อนคำนวณของพ่อ

รหัสที่ 9-20 การใช้การแวะผ่านแบบหลังลำดับเพื่อคำนวณต้นไม้นิพจน์

การวาดรูปต้นไม้

รูปที่ 9-14 แสดงตัวอย่างการวาดต้นไม้ สังเกตได้ว่า การวาดมีลักษณะดังนี้

- ปมของต้นไม้ถูกวางเรียงเป็นระดับ ๆ ในแนวตั้ง ตามความลึกของปม
- ต้นไม้มีความกว้างเท่ากับจำนวนปมในต้นไม้ โดยถ้าพิจารณาที่ปม r ใด ๆ จะพบว่า ปมในลูกต้นซ้ายของ r ทุกปมต้องอยู่ทางซ้ายของ r และในทำนองเดียวกันปมในลูกต้นขวาของ r ทุกปมต้องอยู่ทางขวาของ r
- ด้วยวิธีการวาดข้างต้น ไม่มีกิ่งใดในต้นไม้ตัดขวางกันและกัน

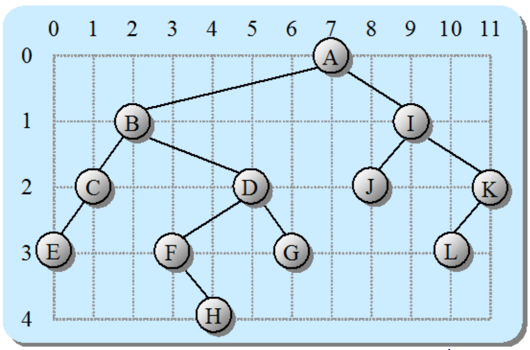


รูปที่ 9-14 ตัวอย่างการวาดต้นไม้

หากเราแบ่งพื้นที่การวาดออกเป็นตาราง สิ่งที่ต้องหาคือพิกัดของปมในต้นไม้ กำหนดให้ (x_i, y_i) คือพิกัดของปม i ในต้นไม้ (โดยกำหนดให้มุมซ้ายบนมีพิกัดเป็น (0,0) พิกัด x และ y เพิ่มขึ้นเมื่อไปทางขวาและลงล่างตามลำดับ) เราสามารถหาค่าของ x_i และ y_i ให้เป็นไปตามข้อสังเกตข้างต้น ดังนี้

- y_i มีค่าเท่ากับเลขระดับ (ซึ่งก็คือความลึก) ของปม i ในต้นไม้
- x_i มีค่าเท่ากับเลขลำดับของปม i ในการแวะผ่านต้นไม้แบบตามลำดับ

รูปที่ 9-15 แสดงตัวอย่างพิกัดของปมต่าง ๆ ที่หาด้วยวิธีข้างบนนี้ การแวะผ่านต้นไม้นี้แบบตามลำดับจะได้ E, C, B, F, H, D, G, A, J, I, L, K ลำดับที่ของปมต่าง ๆ (เริ่มลำดับที่ 0) ก็คือพิกัด x ของปม ตัวอย่างเช่น E เป็นปมแรกในของการแวะผ่าน และมีความลึก 3 จึงอยู่ที่พิกัด (0, 3) ในขณะที่ K เป็นปมลำดับสุดท้าย (เลขลำดับที่ 11) และมีความลึก 2 จึงอยู่ที่พิกัด (11, 2)



รูปที่ 9-15 พิกัด *x* ของปมคือความลึก พิกัด y ของปมคือลำดับที่ในการแวะผ่านตามลำดับ

แล้วเราจะให้ผู้ใช้เรียกบริการวาดรูปได้อย่างไร ? รหัสที่ 9-21 แสดงเมท็อด `toCanvas` ที่คืนอ็อบเจกต์ `Canvas` ให้ผู้ใช้นำไปวางใน `Frame` หรือ `Panel` เพื่อแสดงรูปต้นไม้ (คลาสเหล่านี้อยู่ในชุด `java.awt` ของจาวา) เช่น ในรหัสที่ 9-22 เราสร้างอ็อบเจกต์ของ `Expression` ด้วยการ

ส่งรายการของสตริงที่บรรยายนิพจน์เติมกลาง (รายการนี้สร้างโดยใช้ tokenize2List ที่แยกสตริงออกเป็นรายการของสตริง ที่ขอไม่แสดงรายละเอียด) จะได้ต้นไม้นิพจน์เก็บอยู่ภายใน จากนั้นสร้าง Frame แล้วเรียก exp.toCanvas() ได้อ็อบเจกต์ Canvas กลับมา นำไปใส่ใน Frame แล้วสั่งแสดงผล Frame จะได้รูปต้นไม้นิพจน์ ดังรูปทางขวา

```
public class BinaryTree {
  ...
  public Canvas toCanvas() {
    return new TreeCanvas();
  }
  private class TreeCanvas extends Canvas {
    int ppx, ppy;

    public void paint(Graphics g) {
      ppx = this.getWidth() / (2 + numNodes(root));
      ppy = this.getHeight() / (2 + height(root));
      drawTree(g, root, 1, 1);
    }
    int drawTree(Graphics g, Node r, int x0, int y0) {
      int xr = x0;
      if (r != null) {
        xr += numNodes(r.left);
        int lx = drawTree(g, r.left, x0, y0+1);
        int rx = drawTree(g, r.right, xr+1, y0+1);
        drawNode(g, r, xr, y0);
        if (r.left  != null) drawEdge(g, xr, y0, lx, y0+1);
        if (r.right != null) drawEdge(g, xr, y0, rx, y0+1);
      }
      return xr;
    }
    void drawNode(Graphics g, Node r, int x, int y) {
      int dy = r.isLeaf() ? 15 : 0;
      g.drawString(r.element.toString(), x*ppx, y*ppy+dy);
    }
    void drawEdge(Graphics g, int x1,int y1,int x2,int y2) {
      g.drawLine(x1*ppx, y1*ppy, x2*ppx, y2*ppy);
    }
  }
```

Canvas เป็นคลาสของ java.awt ที่มีให้ใช้วาดรูป และนำไปใส่ใน AWT Container ได้

มีไว้เปลี่ยนพิกัดของตารางเป็นพิกัดของจอภาพ

ระบบแสดงผลของจาวาจะเรียก paint อัตโนมัติ

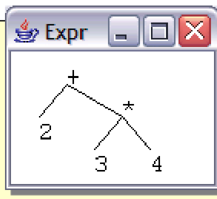
xr เก็บพิกัด x ของปม r

มีลูกซ้ายวาดกิ่งซ้าย มีลูกขวาวาดกิ่งขวา

ถ้าเป็นใบแสดงให้ต่ำนิดหน่อย

รหัสที่ 9-21 คลาสเพื่อสร้าง Canvas สำหรับแสดงรูปต้นไม้

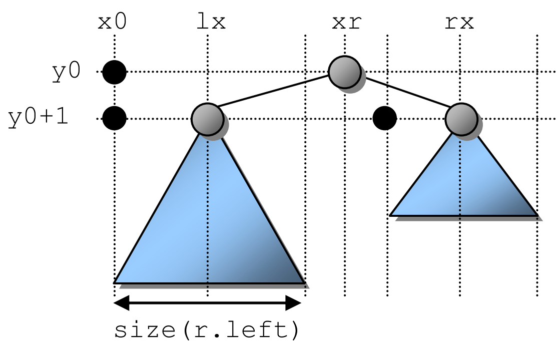
```
List list = tokenize2List("2 + 3 * 4");
Expression exp = new Expression(list);
Frame frame = new Frame("Expr");
frame.add(exp.toCanvas());
frame.setSize(200, 200);
frame.setVisible(true);
```



รหัสที่ 9-22 ตัวอย่างโปรแกรมแสดงต้นไม้นิพจน์ด้วย toCanvas ของ BinaryTree

เมท็อด `toCanvas` (รหัสที่ 9-21) เป็นของคลาส `BinaryTree` เพื่อให้คลาสลูก ๆ สามารถใช้ได้ด้วย `toCanvas` มีการทำงานง่าย ๆ ที่สร้างอ็อบเจกต์ของคลาส `TreeCanvas` แล้วคืนให้ผู้ใช้ ภายใน `TreeCanvas` มีเมท็อด `paint` ซึ่งจะถูกเรียกเมื่อระบบจัดการจอภาพของจาวาต้องการแสดงผล เรามีพิกัดที่ยุ่งด้วยอยู่สองระบบ หนึ่งคือพิกัดของตารางวาดรูปที่ได้นำเสนอในรูปที่ 9-15 และสองคือพิกัดของจุดภาพบน `Canvas` เริ่มด้วยการคำนวณหาจำนวนจุดภาพต่อหน่วยตามแกน x และแกน y ของตารางด้วยการนำจำนวนจุดภาพของความกว้างและความสูงของ `Canvas` มาหารด้วยจำนวนคอลัมน์และจำนวนแถวของตาราง เก็บไว้ในตัวแปร `ppx` และ `ppy` เพื่อนำไปใช้เปลี่ยนพิกัดของตารางไปเป็นพิกัดของจุดภาพ (บรรทัดที่ 76 และ 77) ความกว้างและความสูงของตารางคือจำนวนปมและความสูงของต้นไม้ แต่เราเพิ่มขนาดของตารางไปอีก 2 หน่วยทั้งแนวสูงและกว้างเพื่อให้การวาดต้นไม้ไม่ชิดขอบ จากนั้นสั่งวาดรูปต้นไม้จริง ๆ จัง ๆ ในบรรทัดที่ 78

`drawTree(g,r,x0,y0)` เป็นเมท็อดแสดงต้นไม้ที่มีปม `r` เป็นราก เริ่มวาดในบริเวณที่มุมซ้ายบนอยู่ที่พิกัด `(x0,y0)` ของตาราง (ดูรูปที่ 9-16) วาดเสร็จจะคืน `xr` ซึ่งคือตำแหน่ง x ของปม `r` บนตาราง ที่มีค่าเท่ากับ `x0` บวกกับจำนวนปมของลูกต้นซ้าย (บรรทัดที่ 83 ของรหัสที่ 9-21) แล้ววาดต้นซ้ายด้วย `drawTree(g,r.left,x0,y0+1)` ที่ต้องเป็น `y0+1` เพราะเราต้องวาดลูกในระดับที่ต่ำกว่าพ่อหนึ่งระดับ วาดเสร็จได้ตำแหน่ง x ของรากของต้นซ้าย (`r.left`) เก็บใส่ `lx` แล้วเรียก `drawTree(g,r.right,xr+1,y0+1)` ในทำนองเดียวกันเพื่อวาดต้นขวา ได้ตำแหน่ง x ของรากของ `r.right` เก็บใส่ `rx` จากนั้นวาดปม `r` โดยเรียก `drawNode(g,r,xr,y0)` แล้วลากกิ่งจาก `r` ไปยัง `r.left` ด้วย `drawEdge(g,xr,y0,lx,y0+1)` ปิดท้ายด้วยการลากกิ่งจาก `r` ไปยัง `r.right` ด้วย `drawEdge(g,xr,y0,rx,y0+1)`



รูปที่ 9-16 พิกัดบนตารางของการวาดต้นไม้

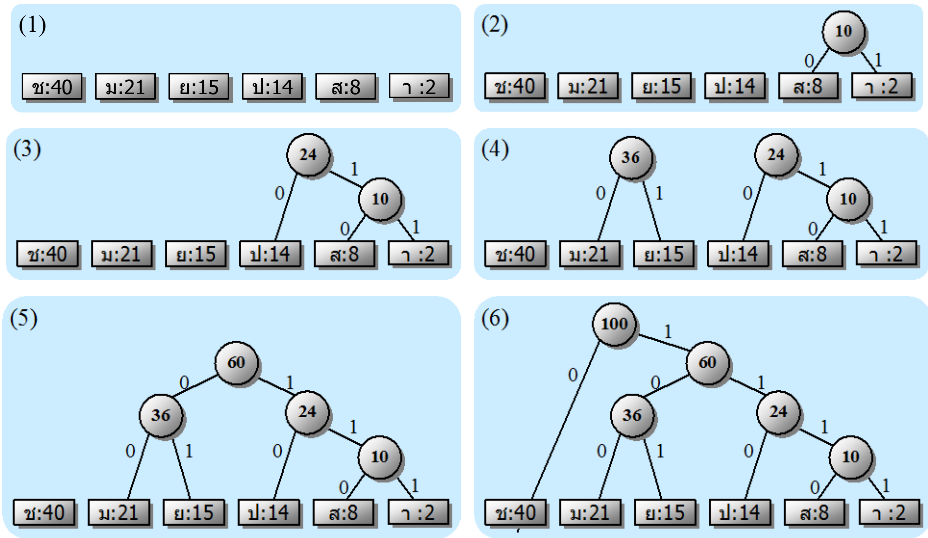
รหัสฮัฟฟ์แมน

ขอยกตัวอย่างการใช้ต้นไม้แบบทวิภาคเป็นโครงสร้างข้อมูลเสริมเพื่อ*การลงรหัสข้อมูลแบบฮัฟฟ์แมน* (Huffman coding) ผู้อ่านคงรู้จักรหัส ASCII ที่ใช้ในการแทนตัวอักษรด้วยจำนวนฐานสองกันมาแล้ว เช่น 'A' แทนด้วย $(01000001)_2$ 'b' แทนด้วย $(01100010)_2$ เป็นต้น ASCII เป็นลักษณะการลงรหัสข้อมูลที่เรียกว่า *แบบความยาวคงที่* (fixed–length) หมายความว่า ข้อมูลทุกตัวใช้เนื้อที่เก็บเท่ากันหมด ยังมีวิธีการลงรหัสข้อมูลอีกแบบหนึ่งที่เรียกว่า *แบบความยาวแปรได้* (variable–length) ซึ่งจำนวนบิตของรหัสที่แทนข้อมูลแต่ละตัวอาจไม่เท่ากัน จุดเด่นของรหัสแบบความยาวแปรได้นี้คือ เราสามารถกำหนดรหัสสั้น ๆ ให้กับข้อมูลที่มีความถี่สูง และกำหนดรหัสยาวให้กับข้อมูลที่มีความถี่ต่ำ ซึ่งอาจใช้ปริมาณเนื้อที่เก็บข้อมูลโดยรวมน้อยกว่าแบบใช้รหัสความยาวคงที่ เพื่อความง่ายในการนำเสนอ ขอสนใจลงรหัสตัวอักษรแต่ละตัวที่มีอยู่ในแฟ้มข้อความ ตัวอย่างเช่น แฟ้มข้อความหนึ่งประกอบด้วยตัวอักษร 'ช', 'ม', 'ย', 'ป', 'ส', และ 'า' เป็นจำนวน 40, 21, 15, 14, 8 และ 2 ตัวตามลำดับ ตารางที่ 9–1 แสดงตัวอย่างการลงรหัสแบบความยาวคงที่ กับแบบความยาวแปรได้ จะเห็นว่า แบบความยาวคงที่นั้นใช้ปริมาณหน่วยความจำรวมเป็น $(40+21+15+14+8+2)\cdot 3 = 300$ บิต ในขณะที่แบบความยาวแปรได้ใช้เพียง $40\cdot 1+21\cdot 3+15\cdot 3+ 14\cdot 3+8\cdot 4+2\cdot 4 = 230$ บิต

ตารางที่ 9–1 ตัวอย่างการเข้ารหัสแบบความยาวคงที่และแบบความยาวแปรได้

	'ช'	'ม'	'ย'	'ป'	'ส'	'า'
จำนวน	40	21	15	14	8	2
รหัสแบบความยาวคงที่	000	001	010	011	100	101
รหัสแบบความยาวแปรได้	0	100	101	110	1110	1111

รหัสฮัฟฟ์แมนเป็นรหัสแบบความยาวแปรได้ ที่เมื่อใช้ลงรหัสชุดข้อมูล จะใช้จำนวนบิตโดยรวมน้อยที่สุด การสร้างรหัสฮัฟฟ์แมนให้กับชุดข้อมูลทำได้ง่าย ๆ เริ่มจากป่าไม้ที่มีแต่ต้นไม้เล็ก ๆ ต้นละปม หนึ่งปมแทนข้อมูลหนึ่งตัว จากนั้นเลือกต้นไม้ในป่ามาสองต้นที่รากมีความถี่น้อยสุดมาผูกเป็นลูกของปมรากใหม่กำกับด้วยผลรวมความถี่ของต้นย่อย กลายเป็นต้นใหม่ เพิ่มเข้าในป่า กระทำการเลือกรากที่มีความถี่น้อยสุดแล้วรวมในลักษณะเช่นนี้เป็นจำนวน n–1 ครั้ง (n คือจำนวนข้อมูลที่ต่างกัน) ก็จะเหลือเพียงต้นเดียวในป่า ซึ่งเป็นต้นไม้แบบทวิภาคที่ใช้ลงรหัสข้อมูล รูปที่ 9–17 แสดงขั้นตอนการเลือกแล้วรวมต้นไม้ของชุดตัวอักษรที่แสดงในตารางที่ 9–1 (จำนวนที่กำกับปมต่าง ๆ ในรูปคือความถี่) จนได้ต้นไม้รหัสในรูป (6) ให้สังเกตว่า เรากำกับทุก ๆ กิ่งซ้ายด้วยเลข 0 และกิ่งขวาด้วยเลข 1 ทำให้รหัสของข้อมูลที่ใบใดก็คือลำดับของเลขที่กำกับกิ่งจากรากถึงใบนั้น เช่น รหัสของตัว 'ส' ได้มาจากรากพุ่งลงกิ่งขวาสามกิ่งแล้วลงกิ่งซ้าย ซึ่งคือรหัส 1110 เป็นต้น รหัสของข้อมูลแต่ละตัวที่ได้ในรูป (6) เหมือนกับแถวล่างสุดที่แสดงในตารางที่ 9–1



รูปที่ 9–17 ขั้นตอนการเลือกแล้วรวมต้นไม้เพื่อสร้างต้นไม้รหัสฮัฟฟ์แมน

รหัสที่ 9-23 แสดงคลาส `HuffmanTree` มีเมท็อด `coding` ให้บริการสร้างต้นไม้รหัสฮัฟฟ์แมนจาก `freq` ซึ่งเป็นแถวลำดับเก็บความถี่ของข้อมูลต่าง ๆ (บรรทัดที่ 13 ถึง 25) เริ่มด้วยการสร้างฮีปน้อยสุดเอาไว้เก็บต้นไม้ระหว่างการหารหัส (ซึ่งก็คืออ็อบเจกต์ของ `HuffmanTree`) โดยใช้ความถี่ที่เก็บกำกับรากของต้นไม้เป็นตัวจัดอันดับแบบฮีป ดังนั้น `HuffmanTree` จึงต้อง `implements Comparable` แล้วเขียนเมท็อด `compareTo` ที่นำความถี่ที่รากมาเปรียบเทียบ (บรรทัดที่ 10 ถึง 12) เราให้ `HuffmanTree` เป็นคลาสลูกของ `BinaryTree` ออกแบบให้ปมของต้นไม้เก็บอ็อบเจกต์ของ `Integer` ซึ่งคือความถี่ แล้วมีเมท็อด `freq` คืนความถี่ที่รากของต้นไม้ (เนื่องจากเราเก็บเป็นอ็อบเจกต์ของ `Integer` จึงต้องใช้เมท็อด `intValue()` ดึงจำนวนเต็มแบบ `int` คืนกลับไป) หลังจากสร้างฮีปเสร็จ ก็นำความถี่ต่าง ๆ ที่ได้รับไปสร้างต้นไม้ต้นเล็ก ๆ ต้นละปมแล้วเพิ่มใส่ฮีป (บรรทัดที่ 15 ถึง 17) จากนั้นเข้าวงวนที่วน n–1 รอบ โดยที่ n คือจำนวนข้อมูลที่หารหัส (อย่าลืมว่า เราไม่สนใจตัวข้อมูล เราสนใจเฉพาะความถี่ของข้อมูล ซึ่งรับมาเป็นพารามิเตอร์ชื่อ `f`) ภายในวงวน ลบต้นไม้สองต้นออกจากฮีป (ซึ่งก็คือต้นไม้ที่รากมีความถี่น้อยสุด) นำไปสร้างต้นใหม่โดยมีรากของต้นทั้งสองที่ถูกลบออกมาจากฮีปเป็นลูก และมีความถี่ที่รากของต้นใหม่เท่ากับผลรวมของความถี่ที่รากของลูกทั้งสอง แล้วนำต้นใหม่เพิ่มใส่ฮีป (บรรทัดที่ 19 ถึง 22) เมื่อวนทำครบแล้ว จะเหลือต้นไม้รหัสที่สมบูรณ์ต้นเดียวในฮีป จึงลบออกจากฮีปแล้วคืนกลับไปผู้เรียก นำต้นไม้รหัสไปใช้ลงรหัสข้อมูลต่อไป จะไม่ขอนำเสนอรายละเอียดของการลงรหัสข้อมูลด้วยต้นไม้รหัส แต่จะแสดงเมท็อด `printCodes` (บรรทัดที่ 26 ถึง 38) ซึ่งทำหน้าที่แสดงรหัสต่าง ๆ ที่หาได้จากต้นไม้รหัสเพื่อเป็นแนวทางในการใช้ต้นไม้รหัส (ผู้อ่านลองทำความเข้าใจเมท็อด `printCodes` ดูเอง) รหัสที่

9-24 แสดงตัวอย่างการสร้างต้นไม้รหัสฮัฟฟ์แมนจากแถวลำดับของความถี่ แสดงเป็นต้นไม้ให้เห็นจริง และเรียก printCodes เพื่อแสดงรหัสต่าง ๆ ของข้อมูลแต่ละตัวในรูปที่ 9–18

```
public class HuffmanTree extends BinaryTree
                             implements Comparable {

  private HuffmanTree(int freq, Node left, Node right) {
    root = new Node(new Integer(freq), left, right);
  }
  private int freq() {
    return ((Integer) root.element).intValue();
  }
  public int compareTo(Object obj) {
    return freq() - ((HuffmanTree) obj).freq();
  }
  public static HuffmanTree coding(int[] f) {
    BinaryMinHeap h = new BinaryMinHeap();
    for (int i = 0; i < f.length; i++) {
      h.enqueue(new HuffmanTree(f[i], null, null));
    }
    for (int i = 0; i < f.length - 1; i++) {
      HuffmanTree t1 = (HuffmanTree) h.dequeue();
      HuffmanTree t2 = (HuffmanTree) h.dequeue();
      h.enqueue(new HuffmanTree(t1.freq() + t2.freq(),
                                  t1.root, t2.root));
    }
    return (HuffmanTree) h.dequeue();
  }
  public void printCodes() {
    printCodes(root, new int[numLeaves(root)], 0);
  }
  private void printCodes(Node r, int[] c, int k) {
    if (r.isLeaf()) {
      System.out.print(r.element + " \t: ");
      for (int i = 0; i < k; i++) System.out.print(c[i]);
      System.out.println();
    } else {
      c[k] = 0; printCodes(r.left, c, k + 1);
      c[k] = 1; printCodes(r.right, c, k + 1);
    }
  }
}
```

ความถี่ที่รากเป็น Integer ต้องใช้ intValue ดึง int

เปรียบเทียบต้นไม้ด้วยความถี่ที่ราก

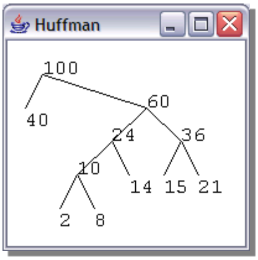
ลบสองต้นที่มีความถี่น้อยสุดมาผูกเป็นลูกของรากใหม่ แล้วเพิ่มใส่ฮีป

ให้กิ่งซ้ายเป็น 0 กิ่งขวาเป็น 1

รหัสที่ 9-23 คลาส Huffman ใช้หารหัสฮัฟฟ์แมน

```
    int[] f = { 40, 21, 15, 14, 8, 2 };
    HuffmanTree t = HuffmanTree.coding(f);
    Frame frame = new Frame("Huffman");
    frame.add(t.toCanvas());
    frame.setSize(250, 250);
    frame.setVisible(true);
    t.printCodes();
```

รหัสที่ 9-24 ตัวอย่างการสร้างต้นไม้รหัสฮัฟฟ์แมน




```
40 : 0
2  : 1000
8  : 1001
14 : 101
15 : 110
21 : 111
```

รูปที่ 9–18 ผลลัพธ์ที่ได้จากการทำงานของรหัสที่ 9-24

การหาอนุพันธ์ด้วยต้นไม้นิพจน์

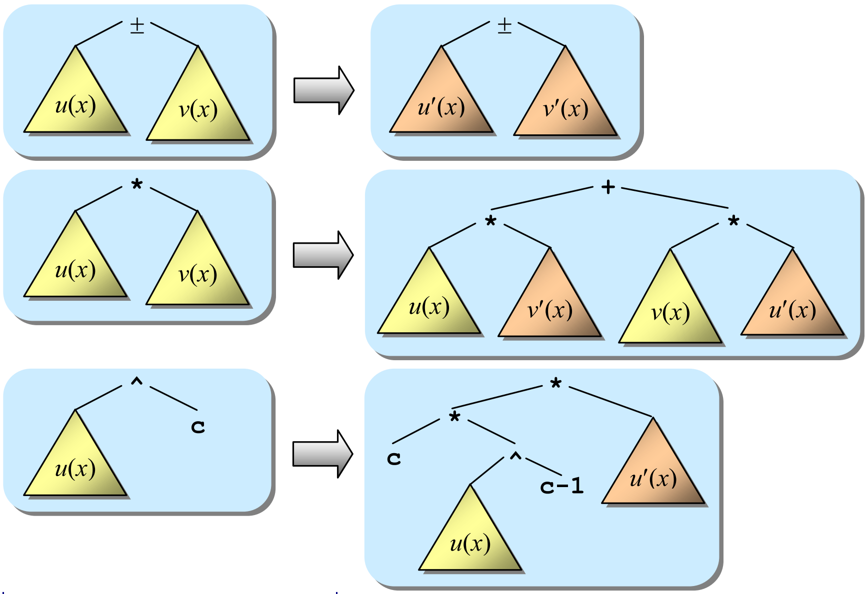
ในกรณีที่เราแทนนิพจน์ตัวแปรเดียวด้วยต้นไม้นิพจน์ เราสามารถหาอนุพันธ์ของนิพจน์ได้อย่างง่ายดายด้วยการเปลี่ยนแปลงต้นไม้ให้มีลักษณะตามกฎการหาอนุพันธ์ เนื่องจากการหาอนุพันธ์ของฟังก์ชันแบบง่ายมีรูปแบบที่สามารถหาได้จากการนำอนุพันธ์ของส่วนย่อย ๆ ในฟังก์ชันมารวมกันดังแสดงในตารางที่ 9-2 ดังนั้นการหาอนุพันธ์กับต้นไม้นิพจน์ก็สามารถใช้กับการสร้างต้นไม้นิพจน์ที่มีโครงสร้างดังแสดงในรูปที่ 9–19 จึงเห็นได้ชัดว่า เราต้องหาอนุพันธ์ของต้นไม้ย่อย ๆ ให้เสร็จก่อนจึงนำต้นไม้นิพจน์ของอนุพันธ์ย่อยทั้งหลายมาประกอบกันเป็นต้นไม้ใหม่ตามสูตร คล้ายกับการแวะผ่านแบบหลังลำดับ

ตารางที่ 9-2 สูตรการหาอนุพันธ์ของฟังก์ชันแบบง่าย

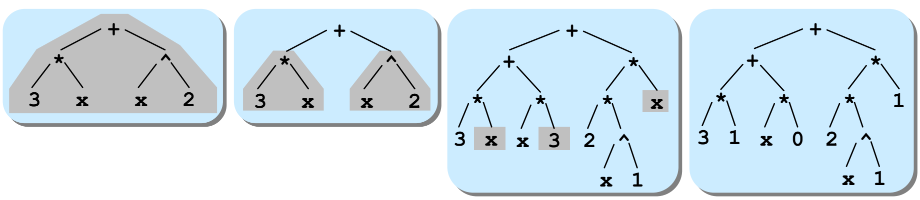
$f(x)$	$f'(x)$
$u(x) \pm v(x)$	$u'(x) \pm v'(x)$
$u(x) \cdot v(x)$	$v(x) \cdot u'(x) + u(x) \cdot v'(x)$
$u(x) / v(x)$	$(\, v(x) \cdot u'(x) - u(x) \cdot v'(x) \,) / v(x)^2$
$u(x)^c$	$c \cdot u(x)^{c-1} \cdot u'(x)$

ตัวอย่างในรูปที่ 9–20 ต้องการหาอนุพันธ์ของ $f(x) = 3x + x^2$ ในรูปนี้ ต้นไม้ย่อยใดที่มีพื้นสีเทาเข้มแสดงว่า เราต้องการหาอนุพันธ์ของต้นไม้ย่อยนั้น เริ่มต้นที่รูป (ก) ต้องการหา $(3x + x^2)'$ มีค่าเป็น $(3x)' + (x^2)'$ ดังรูป (ข) พจน์ย่อย $(3x)'$ มีค่าเป็น $3(x)' + x(3)'$ ส่วน $(x^2)'$ มีค่าเป็น $2x^1(x)'$ ได้ดังรูป (ค)

ซึ่งหาอนุพันธ์ของพจน์ย่อย ๆ ต่อก็จะได้ดังรูป (ง) สรุปว่า อนุพันธ์ของ $3x + x^2$ คือ $(3{\cdot}1 + x{\cdot}0) + (2x^1{\cdot}1)$ ถึงตรงนี้อาจรู้สึกว่า ได้ผลลัพธ์ที่ถึงแม้จะถูกต้องแต่ไม่ค่อยดีเลย ใจเย็น ๆ ก่อน เดี๋ยวเราค่อยนำเสนอวิธีลดรูปต้นไม้นิพจน์ให้เหลือแค่ $3 + 2x$



รูปที่ 9–19 การหาอนุพันธ์ด้วยการเปลี่ยนต้นไม้นิพจน์ (ผู้อ่านลองวาดแบบ $u(x)$ / $v(x)$ เอง)



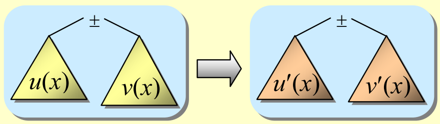
(ก) (ข) (ค) (ง)

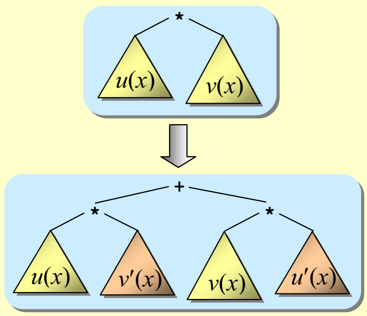
รูปที่ 9–20 อนุพันธ์ของ f(x) = 3*x + x^2 คือ f '(x) = ((3*1)+(x*0)) + 2*(x^1)*1

รหัสที่ 9-25 แสดงเมท็อด `diff` เพื่อใช้เปลี่ยนนิพจน์ที่ถูกเรียก ให้เป็นอนุพันธ์ของนิพจน์นั้น เริ่มเรียก `root=diff(root)` เพื่อหาอนุพันธ์ที่รากบนสุด ได้ผลกลับมาก็เปลี่ยนรากเป็นของผลลัพธ์ ตัวเมท็อด `diff(Node r)` มีหน้าที่คืนต้นไม้ที่เป็นอนุพันธ์ของต้นไม้ที่มี `r` เป็นราก โดยจะคืน `null` ถ้า `r` เป็น `null` ถ้า `r` เป็นใบก็แสดงว่า เป็นตัวถูกดำเนินการที่ต้องเป็นค่าคงตัว, หรือไม่ก็เป็นตัวแปร x, ถ้าเป็น x หาอนุพันธ์ก็ย่อมได้ 1, แต่ถ้าเป็นค่าคงตัว หาอนุพันธ์ก็ย่อมได้ 0 นำผลที่ได้เก็บในปม `r` (บรรทัดที่ 40) แต่ถ้า `r` ไม่ใช่ใบก็ต้องเป็นตัวดำเนินการ ก็ไล่เปรียบเทียบว่า เป็นตัวดำเนินการใด แล้วหาอนุพันธ์ตามกฎของตัวดำเนินการนั้น (บรรทัดที่ 42 ถึง 46) ได้ผลคืนกลับมาเป็นรากของต้นไม้ที่แทนอนุพันธ์ของต้นเดิม ทำเสร็จก็คืนปมรากใหม่กลับคืนไป

```
01 public class Expression extends BinaryTree {
..   ...
33   public void diff() {
34     root = diff(root);
35   }
36   private Node diff(Node r) {
37     if (r == null) return null;
38     String s = (String) r.element;
39     if ( r.isLeaf() ) {
40       r.element = (s.equals("x") ? "1" : "0");
41     } else {
42       if (s.equals("+")) r = diffSum(r);
43       else if (s.equals("-")) r = diffSum(r);
44       else if (s.equals("^")) r = diffExpo(r);
45       else if (s.equals("*")) r = diffMult(r);
46       else if (s.equals("/")) r = diffDiv(r);
47     }
48     return r;
49   }
50   private Node diffSum(Node r) {
51     r.left = diff(r.left);
52     r.right = diff(r.right);
53     return r;
54   }
55   private Node diffMult(Node r) {
56     Node u = copy(r.left);
57     Node v = copy(r.right);
58     Node du = diff(r.left);
59     Node dv = diff(r.right);
60     Node t1 = new Node("*", u, dv);
61     Node t2 = new Node("*", v, du);
62     return new Node("+", t1, t2);
63   }
..   ...
```

dx/dx = 1, dc/dx = 0





รหัสที่ 9-25 เมท็อด `diff` เปลี่ยนนิพจน์ที่เก็บอยู่ให้เป็นอนุพันธ์

เมท็อด `diffSum` และ `diffMult` แสดงการหาอนุพันธ์ผลบวกและผลคูณ `diffSum` เปลี่ยนลูกต้นซ้ายเป็นต้นไม้อนุพันธ์ของลูกต้นซ้าย และก็ทำทำนองเดียวกันกับลูกต้นขวา (บรรทัดที่ 51 และ 52) โดยไม่ต้องเปลี่ยนปม `r` แต่อย่างใด เพราะเราต้องการให้เป็น + เหมือนเดิม ส่วน `diffMult` นั้นซับซ้อนกว่าตรงที่เราต้องทำสำเนาของลูกต้นซ้ายและต้นขวาเก็บไว้ใน `u` และ `v` (บรรทัดที่ 56, 57) ก่อนนำ `r.left` และ `r.right` ไปเปลี่ยนเป็นอนุพันธ์ได้ผลเก็บไว้ใน `du` และ `dv` (บรรทัดที่ 58, 59) แล้วค่อยนำมาสร้างให้เป็น `udv+vdu` (บรรทัดที่ 60 ถึง 62) ได้เป็นผลลัพธ์ (ผู้อ่านลองเขียน `diffExpo` และ `diffDiv` เอง)

การลดรูปต้นไม้นิพจน์

การลดรูปต้นไม้นิพจน์คือการเปลี่ยนต้นไม้นิพจน์ให้มีขนาดเล็กลงโดยไม่เปลี่ยนค่าของนิพจน์ ดังตัวอย่างต้นไม้ทางขวาสุดของรูปที่ 9–20 ซึ่งแทนนิพจน์ $(3{\cdot}1 + x{\cdot}0) + (2x^1{\cdot}1)$ เป็นต้นไม้ที่สามารถลดรูปให้เหลือเพียง 5 ปมเท่านั้น ซึ่งแทนนิพจน์ $3 + 2x$ ที่มีค่าเหมือนกัน

การลดรูปอาศัยการตรวจสอบรูปแบบของต้นไม้ที่มั่นใจว่า เปลี่ยนให้เล็กลงได้โดยไม่เปลี่ยนค่าของนิพจน์ เช่น ถ้าปมใดมีลูกทั้งคู่เป็นค่าคงตัว ย่อมสามารถคำนวณให้เป็นค่าคงตัว ถ้าพบการบวกด้วย 0 การคูณด้วย 0 การยกกำลังด้วย 0 และรูปแบบอื่น ๆ ย่อมลดรูปต้นไม้ให้เล็กลงได้ รหัสที่ 9-26 แสดง `simplify` ที่เป็นเมท็อดลดรูปแบบง่าย ๆ เรียก `simplify(root)` เพื่อลดรูปต้นไม้จากปมรากบนสุด เมท็อด `simplify(Node r)` ลดรูปต้นไม้ที่มี `r` เป็นราก ได้ผลคืนกลับไปเป็นรากของต้นไม้หลังลดรูปแล้ว ถ้า `r` เป็น `null` หรือเป็นใบก็ไม่มีอะไรต้องลด คืน `r` กลับได้เลย แต่ถ้า `r` มีลูก ก็ให้ไปลดรูปลูกทั้งสองก่อน (บรรทัดที่ 69 และ 70) เตรียมตัวแปรสามตัวเพื่อให้เขียนโปรแกรมต่อไปได้กระชับ ได้แก่ `s` เก็บสตริงที่ปม `r` ซึ่งต้องเป็นตัวดำเนินการ (เพราะ `r` ไม่ใช่ใบ) ในกรณีที่ปมลูกเป็นค่าคงตัว `vLeft` หรือ `vRight` จะเก็บค่าคงตัวนั้น แต่ถ้าปมลูกเป็นตัวดำเนินการ เราจะเก็บค่า `Double.NaN` แทน (ในจาวา `NaN` เป็นค่าคงตัวของคลาส `Double` อ่านว่า "Not-a-Number" เอาไว้แทนผลการคำนวณที่แทนด้วย `double` ไม่ได้ เช่น 0/0 เราจึงนำมาใช้แทนสภาพเมื่อนำปมที่เป็นตัวดำเนินการมาแปลงเป็นจำนวน ซึ่งย่อมแปลงไม่ได้ โดยมีเมท็อด `isNumber(x)` ในบรรทัดที่ 105 ถึง 107 ตรวจสอบสภาพดังกล่าว) สรุปรูปแบบที่ใช้ลดรูปต้นไม้ในรหัสที่ 9-26 ดังนี้

1. (บรรทัดที่ 75 และ 76) ตรวจสอบว่า ถ้าลูกของ `r` เป็นค่าคงตัว ก็ให้คำนวณค่าของต้น `r` โดยใช้ `eval` ที่เคยเขียนไว้ในรหัสที่ 9-20 คำนวณเสร็จก็สร้างใบที่เก็บผลลัพธ์แทนรากเก่า
2. (บรรทัดที่ 78 ถึง 80) ไม่ได้ลดรูปต้นไม้ แต่ตรวจสอบว่า ถ้าลูกขวาเป็นค่าคงตัว และ `s` คือ `+` หรือ `*` จะสลับลูกทั้งสอง การสลับนี้ไม่เปลี่ยนค่าของนิพจน์ (เช่น $f(x)+3 = 3+f(x)$) แต่จะทำให้การตรวจสอบรูปแบบอื่น ๆ ที่ตามมามีน้อยกรณีลง
3. (บรรทัดที่ 84) แทนกรณี $1 * f(x)$ เปลี่ยนเป็น $f(x)$
4. (บรรทัดที่ 86) แทนกรณี $0 * f(x)$, $0 / f(x)$ และ $0 \wedge f(x)$ เปลี่ยนเป็น 0
5. (บรรทัดที่ 87) แทนกรณี $0 + f(x)$ เปลี่ยนเป็น $f(x)$
6. (บรรทัดที่ 91) แทนกรณี $f(x) * 1$, $f(x) / 1$ และ $f(x) \wedge 1$ เปลี่ยนเป็น $f(x)$
7. (บรรทัดที่ 93) แทนกรณี $f(x) \wedge 0$ เปลี่ยนเป็น 1
8. (บรรทัดที่ 94) แทนกรณี $f(x) - 0$ เปลี่ยนเป็น $f(x)$

```
public class Expression extends BinaryTree {
  ...
  public void simplify() {
    root = simplify(root);
  }
  private Node simplify(Node r) {
    if (r == null || r.isLeaf()) return r;
    r.left = simplify(r.left);
    r.right = simplify(r.right);

    String s = (String) r.element;
    double vLeft = toDouble(r.left.element);
    double vRight = toDouble(r.right.element);
    if (isNumber(vLeft) && isNumber(vRight)) {
      r = new Node(Double.toString(eval(r)), null, null);
    } else {
      if (isNumber(vRight) && "*+".indexOf(s) >= 0) {
        Node t1 = r.left; r.left = r.right; r.right = t1;
        double t2 = vLeft; vLeft = vRight; vRight = t2;
      }
      if (isNumber(vLeft)) {
        if (vLeft == 1) {
          if (s.equals("*")) r = r.right;
        } else if (vLeft == 0) {
          if ("*/^".indexOf(s)>=0) r=new Node("0",null,null);
          if (s.equals("+")) r = r.right;
        }
      } else if (isNumber(vRight)) {
        if (vRight == 1) {
          if ("*/^".indexOf(s) >= 0) r = r.left;
        } else if (vRight == 0) {
          if (s.equals("^")) r = new Node("1", null, null);
          if (s.equals("-")) r = r.left;
        }
      }
    }
    return r;
  }
  private static double toDouble(Object s) {
    try { return Double.parseDouble((String) s); }
    catch (NumberFormatException e) { }
    return Double.NaN;
  }
  private static boolean isNumber(double x) {
    return !Double.isNaN(x);
  }
  ...
```

ลดรูป ลูก ๆ ก่อนลดรูปพ่อ

เขียน `"*/^".indexOf(s) >= 0` **เหมือนกับเขียน** `s.equals("*")||s.equals("/")||s.equals("^")`

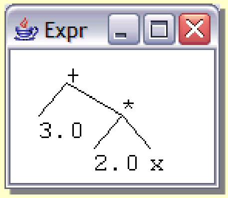
ถ้า s เป็น operator, parseDouble เกิดปัญหา

ต้องใช้ Double.isNaN(x) ใช้ x != Double.NaN ไม่ได้

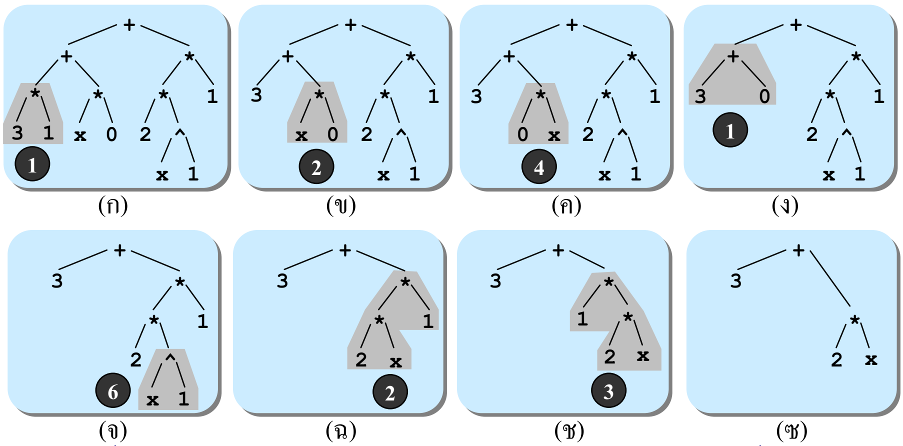
รหัสที่ 9-26 เมท็อด `simplify` ลดรูปต้นไม้นิพจน์ให้เล็กลง (เลขในวงกลมทึบคือรูปแบบ)

รหัสที่ 9-27 แสดงตัวอย่างการใช้งานคลาส `Expression` ที่ได้เขียนกันมาเพื่อสร้างนิพจน์ หาอนุพันธ์ ลดรูป พร้อมทั้งแสดงต้นไม้นิพจน์ภายในคลาสให้ชมด้วย การหาอนุพันธ์นั้นได้แสดงขั้นตอนการทำงานไว้แล้วในรูปที่ 9–20 ส่วนขั้นตอนการลดรูปนั้นแสดงในรูปที่ 9–21 (เลขในวงกลมทึบคือรูปแบบที่ใช้กับต้นไม้ย่อยพื้นสีเทา)

```
List list = tokenize2List("3*x + x^2");
Expression exp = new Expression(list);
exp.diff();
exp.simplify();
Frame frame = new Frame("Expr");
frame.add(e.toCanvas());
frame.setSize(140, 120);
frame.setVisible(true);
```



รหัสที่ 9-27 ตัวอย่างการใช้งานคลาส Expression ในการหาอนุพันธ์และลดรูป



รูปที่ 9–21 การลดรูปต้นไม้นิพจน์โดยใช้เมท็อด `simplify` ในรหัสที่ 9-26

แบบฝึกหัด

1. จงวิเคราะห์ว่า มีตัวโยงไปยังลูกที่มีค่า `null` ร้อยละเท่าไร ในการสร้างต้นไม้ 3-ภาค ด้วยการจองตัวโยงสามตัวเพื่ออ้างอิงลูกทั้งสามดังแสดงในรูปที่ 9-4 และถ้าเป็นกรณีของการสร้างต้นไม้ m-ภาค ด้วยการจองตัวโยงลูก m ตัวต่อหนึ่งปม จะมีตัวโยงร้อยละเท่าไรที่มีค่า `null` (ข้อแนะนำ : ต้นไม้ที่มี n ปม จะมี $n - 1$ กิ่ง)

2. จงวิเคราะห์เวลาการทำงานของเมท็อด `numNodes`, `numLeaves`, `height`, `copy`, และ `toArray` ในรหัสที่ 9-8 ถึงรหัสที่ 9-12

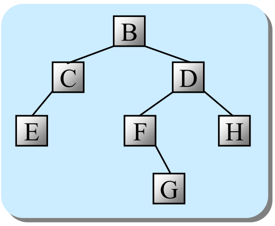
3. จงเขียนเมท็อด `numNodes`, `numLeaves`, และ `height` ให้กับคลาส `BinaryTree` โดยใช้ตัวเยี่ยมชม

4. จงวาดต้นไม้แบบทวิภาคต้นหนึ่งที่เมื่อแวะผ่านแบบก่อนลำดับได้ A, B, C, E, D, F, G, I, H และเมื่อแวะผ่านแบบหลังลำดับจะได้ C, D, E, B, G, H, I, F, A

5. รหัสที่ 9-19 ในหน้าที่ 181 ใช้ `preOrder` ในการแวะผ่าน หากเราเปลี่ยนเป็น `inOrder` หรือ `postOrder` จะได้ผลเหมือนกัน ต่างกันอย่างไร

6. เพื่อให้ตัวเยี่ยมชมสามารถเก็บหรือสะสมข้อมูลอะไรบางอย่างระหว่างการแวะผ่าน เมื่อแวะเสร็จแล้วผู้ใช้สามารถขอข้อมูลนั้นจากตัวเยี่ยมชมได้ เช่น ถ้าเราจะเขียนเมท็อด `numNodes` โดยใช้ตัวเยี่ยมชม ก็คงต้องจองแถวลำดับของ `int` หนึ่งช่องแบบ `final` ไว้ให้เมท็อด `visit` เพิ่มทุกครั้งที่แวะปมใหม่ แต่ถ้าตัวเยี่ยมชมมีเมท็อด `setResult` ให้การทำงานใน `visit` ตั้งผลลัพธ์ และมีเมท็อด `getResult` ให้ผู้ใช้หยิบผลลัพธ์ การเขียนตัวเยี่ยมชมก็จะสะดวกขึ้น จงนำเสนอวิธีการปรับปรุงคลาส `Visitor` เพื่อให้บริการดังกล่าว

7. ถ้าแต่ละปมในต้นไม้แบบทวิภาคมีตัวโยง `parent` ไปยังปมพ่อด้วย (โดยปมพ่อของรากคือ `null`) จงเขียนเมท็อด `Node successor(Node x)` ให้กับคลาส `BinaryTree` เพื่อคืนปมที่เป็นปมถัดจากปม `x` ในการแวะผ่านแบบตามลำดับ (เช่น `successor` ของปม `B` ในรูปข้างล่างนี้คือปม F)



8. จงเขียนเมท็อด `levelOrder` เพื่อแวะผ่านต้นไม้ในอีกลักษณะที่เรียกว่า *การแวะผ่านตามระดับ* คือจะเยี่ยมปมทีละระดับ ๆ จากบนลงล่าง โดยในแต่ละระดับจะเยี่ยมปมจากซ้ายไปขวา เช่น การแวะผ่านตามระดับในต้นไม้ข้างบนนี้จะได้ B, C, D, E, F, H, G

9. เราเรียก `null` ในต้นไม้ว่า ปมภายนอก และปมอื่น ๆ ว่า ปมภายใน นิยามให้*ความยาววิถี* (path length) จากรากถึงปม k คือจำนวนกิ่งจากรากลงมาจนถึงปม k, *ความยาวรวมของวิถีภายใน* (internal path length) คือผลรวมของความยาววิถีจากรากถึงปมภายในทุกปม, และ*ความยาวรวมของวิถีภายนอก* (external path length) คือผลรวมของความยาววิถีจากรากถึงปมภายนอกทุกปม

9.1. ให้ I(n) และ E(n) คือความยาวรวมของวิถีภายในและภายนอก (ตามลำดับ) ของต้นไม้แบบทวิภาคที่มีปมภายใน n ปม จงพิสูจน์ว่า E(n) = I(n) + 2n

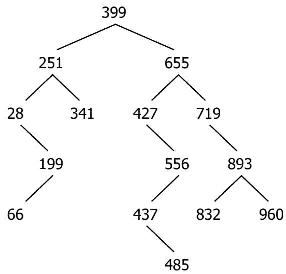
9.2. จงพิสูจน์ว่า $n \log_2(n/4) \le \mathrm{I}(n) \le n(n-1)/2$

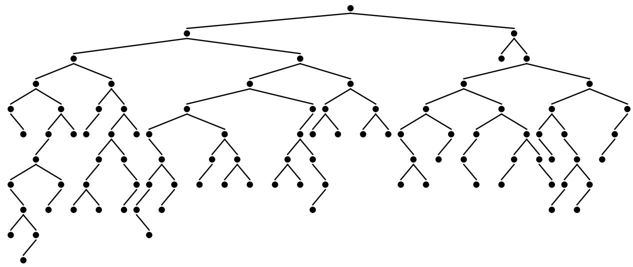
9.3. จงเขียนเมท็อด

```
public int internalPathLength()
public int externalPathLength()
```

ให้กับคลาส `BinaryTree` เพื่อคืนความยาวรวมของวิถีภายในและภายนอกของต้นไม้

10. จงออกแบบและเขียนส่วนของโปรแกรมเพื่อวาดรูปต้นไม้ที่ “น่าจะ” สวยกว่าแบบที่ได้นำเสนอมาในรหัสที่ 9-21 ดังแสดงตัวอย่างข้างล่างนี้





11. จงวาดต้นไม้ของการหาอนุพันธ์แบบ $u(x)/v(x)$ ให้กับรูปที่ 9–19

12. จงเขียนเมท็อด `diffExpo` และ `diffDiv` ให้กับรหัสที่ 9-25

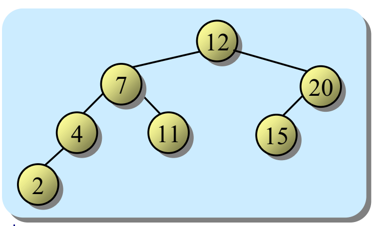
13. จงเพิ่มรูปแบบการลดรูป และปรับปรุงเมท็อด `simplify` ในรหัสที่ 9-26 ให้ลดรูปได้ดีกว่าที่ได้นำเสนอ

10 ต้นไม้ค้นหาแบบทวิภาค

ต้นไม้ค้นหาแบบทวิภาค (binary search tree) เป็นต้นไม้แบบทวิภาคชนิดหนึ่งที่มีกฎการจัดเก็บข้อมูลที่เหมาะกับการให้บริการค้นหา เพิ่ม และลบข้อมูล เป็นโครงสร้างพื้นฐานที่ได้รับการประยุกต์และปรับปรุงให้จัดเก็บข้อมูลทั้งในลักษณะง่าย ๆ เพื่อเก็บข้อมูลในหน่วยความจำหลัก และซับซ้อนเพื่อเก็บข้อมูลจำนวนมากในระบบฐานข้อมูล บทนี้นำเสนอการสร้างต้นไม้ค้นแบบทวิภาคแบบพื้นฐาน และต้นไม้เอวีแอลซึ่งเป็นต้นไม้ค้นหาแบบทวิภาคชนิดพิเศษที่ประกันความสูง ทำให้ใช้เวลาการทำงานของการค้น เพิ่ม และลบข้อมูลเป็น $O(\log n)$ นอกจากนี้ยังนำเสนอการนำต้นไม้ค้นหามาสร้างที่เก็บข้อมูลแบบเซตกับแมป และแนะนำต้นไม้ค้นหาแบบอื่น ๆ

ลักษณะของต้นไม้ค้นหาแบบทวิภาค

ต้นไม้ค้นหาแบบทวิภาคมีโครงสร้างเหมือนต้นไม้แบบทวิภาคทุกประการ สิ่งที่เพิ่มเติมคือกฎการจัดเก็บข้อมูลตามปมต่าง ๆ โดยต้องให้ข้อมูลที่ปมมีค่ามากกว่าข้อมูลทุกตัวในลูกต้นซ้าย และมีค่าน้อยกว่าข้อมูลทุกตัวในลูกต้นขวา เช่น ในรูปที่ 10-1 พิจารณาต้นไม้ย่อยที่มี 7 เป็นราก ที่ต้นซ้ายของ 7 เก็บ 4 และ 2 ซึ่งน้อยกว่า 7 ในขณะที่ต้นขวาของ 7 เก็บ 11 ซึ่งมากกว่า 7 และไม่ว่าจะพิจารณาที่ปมใดในต้นไม้ ก็เป็นไปตามกฎการจัดเก็บดังกล่าวทั้งสิ้น



รูปที่ 10-1 ตัวอย่างต้นไม้ค้นหาแบบทวิภาค

ผู้อ่านอาจสงสัยว่า ถ้ามีข้อมูลซ้ำกับตัวที่ราก จะนำไปเก็บทางต้นซ้ายหรือต้นขวา ต้องขอบอกตอนนี้ก่อนว่า ในช่วงแรกจะขอนำเสนอเฉพาะกรณีไม่มีตัวซ้ำ นั่นคือสนใจเก็บข้อมูลในลักษณะของเซต แล้วค่อยนำเสนอในภายหลังกรณีเก็บในลักษณะคอลเล็กชันที่อนุญาตให้มีข้อมูลซ้ำได้

```
public class BinaryTree {

  protected Node root;

  protected static class Node {
    public Object element;
    public Node left;
    public Node right;
    public Node(Object e, Node l, Node r) {
      element = e; left = l; right = r;
    }
    public boolean isLeaf() {return left==null && right==null;}
  }
  ...
```

เก็บรากของต้นไม้

แต่ละปมเก็บข้อมูล 1 ตัว และตัวโยงไปหาลูกทั้งสอง

บริการตรวจสอบว่าปมที่เรียกเป็นใบ (ไม่มีทั้งลูกซ้ายและขวา) หรือไม่

รหัสที่ 10-1 `BinaryTree` คือคลาสแม่ของสารพัดคลาสที่มีโครงสร้างเป็นต้นไม้แบบทวิภาค

เราจะสร้างต้นไม้ค้นหาแบบทวิภาคจากต้นไม้แบบทวิภาคที่ได้ศึกษากันมาในบทที่แล้ว รหัสที่ 10-1 สรุปโครงสร้างของคลาส `BinaryTree` ที่ได้นำเสนอในบทที่ 9 รหัสที่ 10-2 แสดงคลาส `BSTree` ที่สร้างต้นไม้ค้นหาแบบทวิภาค โดยให้เป็นคลาสลูกของ `BinaryTree` มีตัวแปร `size` เก็บจำนวนข้อมูลในต้นไม้ ที่สามารถใช้หาผลลัพธ์ของเมท็อด `size()` และ `isEmpty()` ตามที่เคยปฏิบัติกันมา รหัสที่ 10-2 ยังนำเสนอหัวเมท็อดที่เป็นบริการของ `BSTree` ได้แก่ `add` และ `remove` เพื่อการเพิ่มและลบข้อมูล, `get` ค้นหาและคืนข้อมูลที่ต้องการ, `getMin` และ `getMax` คืนข้อมูลที่มีค่าน้อยสุดและมากสุด, และ `treeSort` เป็นบริการเสริมเรียงลำดับข้อมูลในแถวลำดับที่ได้รับ

```
public class BSTree extends BinaryTree {
  protected int size;

  public BSTree() {}
  public int size() {return size;}
  public boolean isEmpty() {return size == 0;}
  protected int compare(Object a, Object b) {
    return ((Comparable)a).compareTo(b);
  }
  public Object get(Object e) {...}
  public Object getMin() {...}
  public Object getMax() {...}
  public void add(Object e) {...}
  public void remove(Object e) {...}
  public static treeSort(Object[] data) {...}
```

เก็บจำนวนข้อมูล เพื่อความรวดเร็วในการให้บริการ size()

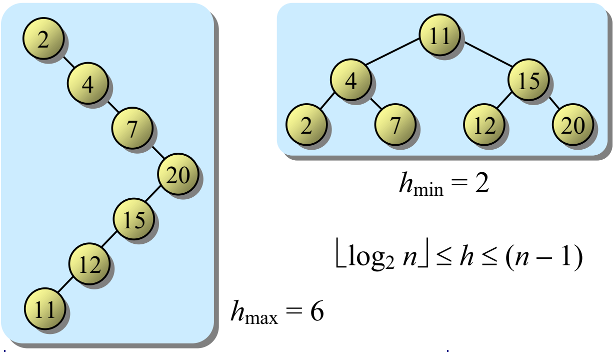
คืน 0 เมื่อ a เท่ากับ b
คืนจำนวนลบ เมื่อ a น้อยกว่า b
คืนจำนวนบวก เมื่อ a มากกว่า b

รหัสที่ 10-2 `BSTree` คือคลาสของต้นไม้ค้นหาแบบทวิภาค

ขอตั้งข้อสังเกตตรงนี้ก่อนว่า การเก็บข้อมูลชุดหนึ่งด้วยต้นไม้ค้นหาแบบทวิภาคนั้น สามารถเก็บในต้นไม้ได้หลากหลายแบบ ดังตัวอย่างในรูปที่ 10-2 แสดงการเก็บข้อมูลชุดเดียวกับในรูปที่ 10-1 แต่มีลักษณะและความสูงต่างกัน ต้นไม้ในรูปที่ 10-2 แสดงกรณีที่สูงสุด และเตี้ยสุด ในที่นี้มีข้อมูล 7 ตัว กรณีสูงสุดจะเป็นต้นไม้สูง $7 - 1 = 6$ แต่ถ้าเป็นกรณีเตี้ยสุด จะสูง $\lfloor \log_2 7 \rfloor = 2$ ในกรณีทั่วไป ถ้า h แทนความสูงของต้นไม้ ต้นไม้ค้นหาแบบทวิภาคที่มี n ปมจะมีความสูงในช่วง

$$\lfloor \log_2 n \rfloor \le h \le (n-1)$$

ซึ่งเป็นช่วงที่กว้างมาก เช่น $n = 10^6$ พบว่า h มีค่าได้ตั้งแต่ 19 ถึง 999,999 เนื่องจากความสูงมีผลโดยตรงกับประสิทธิภาพการให้บริการค้น เพิ่ม และลบข้อมูล (จะได้ศึกษากันต่อในประเด็นนี้) ดังนั้น หากเราสามารถจัดเก็บให้ได้ต้นไม้ที่เตี้ย ย่อมทำให้การจัดการข้อมูลมีประสิทธิภาพดี



รูปที่ 10-2 ต้นไม้ค้นหาแบบทวิภาคสองลักษณะที่เก็บข้อมูลเหมือนกัน

อีกข้อสังเกตหนึ่งที่ต้องแจ้งให้ทราบก่อนตอนนี้ก็คือ การเก็บข้อมูลในต้นไม้ค้นหาแบบทวิภาคต้องใช้การเปรียบเทียบข้อมูลว่า น้อยกว่าไปอยู่ทางซ้าย มากกว่าไปอยู่ทางขวา เนื่องจากเราจะเก็บข้อมูลที่เป็นอ็อบเจกต์ตามปมต่าง ๆ (ไม่ได้เก็บข้อมูลพื้นฐาน เช่น `int`, `double`, `char`) ดังนั้น เพื่อให้มั่นใจว่า อ็อบเจกต์ต่าง ๆ ที่เก็บในต้นไม้ "เปรียบเทียบกันได้" จึงต้องบังคับกันว่า อ็อบเจกต์เหล่านี้ต้องเป็นของคลาสที่ `implements Comparable` ซึ่งบังคับให้อ็อบเจกต์มีเมท็อด `compareTo` และเนื่องจากก่อนจะเรียก `compareTo` ต้องเปลี่ยนหรือที่เรียกว่า cast อ็อบเจกต์ของคลาส `Object` ให้เป็น `Comparable` ก่อน ดังนั้นเพื่อให้โปรแกรมที่จะเขียนต่อไปกะทัดรัด อ่านง่าย จึงขอเขียนเมท็อด `compare(a,b)` ดังแสดงในบรรทัดที่ 7 ถึง 9 ของรหัสที่ 10-2 เพื่อเปรียบเทียบ `a` กับ `b` ผลที่ได้เป็นจำนวนเต็ม ถ้าได้ 0 แสดงว่า `a` เท่ากับ `b` ถ้าได้จำนวนลบแสดงว่า `a` น้อยกว่า `b` และถ้าได้จำนวนบวกแสดงว่า `a` มากกว่า `b`

การค้นหาข้อมูล

แค่ชื่อก็บอกแล้วว่า ต้นไม้ค้นหาแบบทวิภาคเป็นโครงสร้างเพื่อการค้นหา แล้วจะค้นหาอย่างไรได้เร็ว ? บางคนอาจคิดถึงการแวะผ่านต้นไม้ที่เป็นบริการซึ่งเรียกใช้ได้ของ `BinaryTree` เพื่อเปรียบเทียบข้อมูลตามปมต่าง ๆ ไปเรื่อย ๆ ว่าพบหรือยัง ดังแสดงในรหัสที่ 10-3 เมท็อด `get` อาศัยการสร้างตัวเยี่ยมชมที่มีเมท็อด `visit` ไว้ตรวจสอบว่า ข้อมูลของปมที่แวะมีค่าเท่ากับ `x` ที่ต้องการหรือไม่ ถ้าเท่าก็จำผลนี้ไว้ (ใช้แถวลำดับ `result` ที่มีขนาดหนึ่งช่องในเมท็อดไว้ให้ `get` เก็บผล) และสั่งให้ยุติการค้น (ด้วยเมท็อด `done` ของตัวเยี่ยมชม) หลังจากแวะผ่านเสร็จแล้ว ก็คืนผลที่ได้

```
public Object get(final Object e) {
  final Object[] result = new Object[1];
  preOrder(root,
    new Visitor() {
      public void visit(Object x) {
        if (x.equals(e)) {result[0] = x; done();}
      }});
  return result[0];
}
```

แถวลำดับหนึ่งช่องนี้มีไว้เก็บผลการค้น

เมื่อพบ e เก็บผลไว้ แล้วบอก visitor ให้ยุติการแวะผ่านเสร็จแล้ว

รหัสที่ 10-3 การค้นหาข้อมูลด้วยการแวะผ่านต้นไม้

วิธีแวะผ่านต้นไม้ง่ายดี ได้ใช้บริการของ `BinaryTree` ที่มีอยู่แล้ว (เพราะ `BSTree` เป็นคลาสลูก) แต่ช้า ใช้เวลาเป็น $O(n)$ โดยที่ n คือจำนวนปมในต้นไม้ เขียน `get` แบบนี้เสียชื่อการเป็นต้นไม้ค้นหา การแวะผ่านต้นไม้ไม่ได้ใช้ความสัมพันธ์ของการจัดเก็บให้ลูกต้นซ้ายเก็บข้อมูลน้อยกว่าที่ราก และลูกต้นขวาเก็บข้อมูลที่มากกว่าเลย จึงน่าจะมีวิธีที่ดีกว่าการแวะผ่านต้นไม้ แต่ก่อนอื่น ขอแยกเขียนการค้นออกเป็นเมท็อดย่อยชื่อ `getNode(r,e)` ซึ่งคืนปมที่เก็บ `e` ในต้นที่มี `r` เป็นราก ถ้าหาไม่พบให้คืน `null` เราให้เมท็อดนี้เป็น `protected` เพื่อให้คลาสลูกใช้ประโยชน์ได้ เมื่อมี `getNode(r,e)` จึงเขียนเมท็อด `get` ให้เรียก `getNode(root,e)` ได้ดังรหัสที่ 10-4

```
public Object get(Object e) {
  Node node = getNode(root, e);
  return node == null ? null : node.element;
}
protected Node getNode(Node r, Object e) {
  ...
```

getNode คืนปมที่มีข้อมูลเท่ากับ e

get คืนข้อมูลที่เก็บในปมที่พบ

รหัสที่ 10-4 เมท็อดสาธารณะ `get` เรียกใช้เมท็อด `getNode` เพื่อค้นหาปมที่มีข้อมูลที่ต้องการ

ขอนำเสนอ `getNode(r,e)` ในรหัสที่ 10-5 ซึ่งคืนปมที่เก็บ `e` ในต้นที่มี `r` เป็นราก ซึ่งทำงานได้รวดเร็วกว่ารหัสที่ 10-3 หลักการทำงานคือเริ่มต้นที่ปม `r` แล้วเลื้อยลงมาตามกิ่งไปในทิศทางที่นำไปสู่เป้าหมาย โดยใช้ผลการเปรียบเทียบความน้อยกว่ามากกว่าของตัวที่ต้องการกับข้อมูลในปม

การทำงานอาศัยวงวน `while` ในบรรทัดที่ 15 ที่วนค้นหาไปเรื่อย ๆ ตราบเท่าที่ปมที่สนใจยังไม่เป็น `null` ภายในวงวนเริ่มเปรียบเทียบ `e` กับข้อมูลที่ `r` โดยอาศัยเมท็อด `compare` ที่เขียนไว้ก่อนหน้านี้ (ในรหัสที่ 10-2) ถ้า `e` เท่ากับข้อมูลที่ `r` (บรรทัดที่ 16) ก็แสดงว่าพบแล้ว หาก `e` น้อยกว่าข้อมูลที่ `r` ก็เลี้ยวซ้ายไปค้นหาต่อในลูกต้นซ้ายของ `r` (เพราะถ้า `e` น้อยกว่าที่ `r` ก็ย่อมต้องน้อยกว่าข้อมูลทุกตัวในลูกต้นขวา จึงควรเลี้ยวซ้าย ไม่ต้องสนใจลูกต้นขวา) และในทางกลับกัน ถ้า `e` มากกว่าข้อมูลที่ `r` ก็เลี้ยวขวาไปค้นหาต่อในลูกต้นขวา (บรรทัดที่ 17) เมื่อใดที่ `r` เป็น `null` ก็แสดงว่าหาไม่พบ การทำงานหลุดจากวงวน จึงคืนค่า `null`

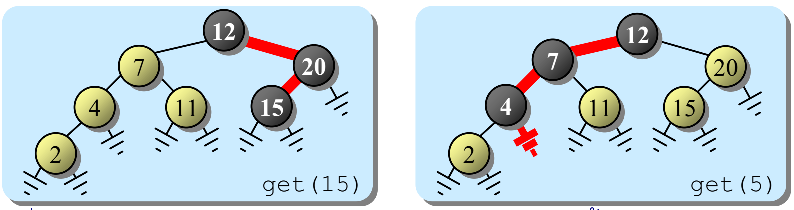
```
14    protected Node getNode(Node r, Object e) {
15      while (r != null) {
16        if (compare(e, r.element) == 0) return r;
17        r = compare(e, r.element) < 0 ? r.left : r.right;
18      }
19      return null;
20    }
```

ไม่พบคืน null

เปลี่ยน r ไปทางซ้ายเมื่อ e น้อยกว่าข้อมูลที่ r ถ้ามากกว่าเปลี่ยนไปทางขวา

รหัสที่ 10-5 เมท็อด `getNode` เพื่อการค้นหาข้อมูล

รูปที่ 10-3 แสดงตัวอย่างการค้นหาข้อมูล รูปซ้ายต้องการค้นหา 15 ต้องเปรียบเทียบกับ 12, 20, และ 15 จึงสรุปได้ว่า พบ 15 ในต้นไม้ ในขณะที่รูปขวาต้องการค้นหา 5 ต้องเปรียบเทียบกับ 12, 7, 4 และจบด้วยการพบต้นไม้ว่าง จึงสรุปได้ว่า ไม่พบ 5 ในต้นไม้



รูปที่ 10-3 ตัวอย่างการค้นหาข้อมูลต้นไม้ค้นหาแบบทวิภาคทั้งพบและไม่พบข้อมูล

เห็นได้ชัดว่า ในกรณีโชคร้ายสุด ๆ จะเปลี่ยนค่าของ `r` ลงมาเรื่อย ๆ ทีละระดับ ๆ จนถึงระดับล่างสุด นั่นคือทำงานในวงวนเป็นจำนวนรอบเท่ากับความสูงของต้นไม้ ดังนั้น `getNode` (ซึ่งก็หมายถึง `get` ด้วย) ใช้เวลาเป็น O(h)

ในกรณีที่เราจุกจิกเรื่องกลวิธีการเขียนโปรแกรม ขอให้กลับไปดูรหัสที่ 10-5 พบว่า ไม่ควรตรวจสอบเงื่อนไขเท่ากันก่อน ในบรรทัดที่ 16 เพราะโอกาสที่เราจะพบปมที่มีข้อมูลที่ต้องการ น่าจะน้อยกว่ากรณีไม่พบ (ลองคิดดูว่า ถ้ามีข้อมูลสัก 1,000 ตัวในต้นไม้ เรามีโอกาสเพียงหนึ่งในพันที่จะหยิบมาหนึ่งปมแล้วพบตัวที่ต้องการ) จึงควรตรวจสอบแบบไม่เท่าก่อน ดังแสดงในรหัสที่ 10-6

```
protected Node getNode(Node r, Object e) {
  while (r != null) {
    if      (compare(e, r.element) < 0) r = r.left;
    else if (compare(e, r.element) > 0) r = r.right;
    else return r;
  }
  return null;
}
```

โอกาสเท่ามีน้อยกว่าไม่เท่า ย้ายมาทำทีหลัง

ปรับปรุงได้อีก ถ้าเก็บผลของ compare ในตัวแปร จะได้ไม่ต้องเรียกสองครั้ง

รหัสที่ 10-6 เมท็อด getNode เพื่อการค้นหาฉบับปรับปรุง

ที่ผ่านมาการค้นหาอาศัยการทำงานแบบวงวน เราสามารถบรรยายขั้นตอนการค้นหาเป็นแบบเวียนเกิด (รหัสที่ 10-7) เริ่มด้วยการตรวจสอบว่า ถ้า r เป็น null ก็แสดงว่าหาไม่พบ ถ้า e มีค่าน้อยกว่าที่ r ก็เรียก getNode(r.left,e) เพื่อไปค้นในลูกต้นซ้ายของ r แต่ถ้า e มีค่ามากกว่าที่ r ก็เรียก getNode(r.right,e) เพื่อไปค้นในลูกต้นขวา ถ้าไม่น้อยไม่มาก สรุปได้แน่นอนว่าพบแล้ว ก็คืน r

```
protected Node getNode(Node r, Object e) {
  if (r == null) return null;
  if (compare(e, r.element) < 0) return getNode(r.left, e);
  if (compare(e, r.element) > 0) return getNode(r.right, e);
  return r;
}
```

พบก็คืน ไม่พบก็ค้นต่อทางซ้ายหรือขวา

รหัสที่ 10-7 เมท็อด getNode (เขียนแบบเวียนเกิด) เพื่อการค้นหาข้อมูล

ต้องขอบอกว่า การค้นหาแบบเวียนเกิดในรหัสที่ 10-7 ใช้เวลาการทำงานเหมือนกับของรหัสที่ 10-5 และรหัสที่ 10-6 คือเป็น $O(h)$ แต่ในกรณีที่เราเขียนแบบเวียนเกิด ซึ่งอาศัยการเรียกเมท็อด จะเกิดการสร้างกรอบกองซ้อนในระบบ กรณีช้าสุดต้องเรียกแบบเวียนเกิด h ครั้ง จึงใช้หน่วยความจำเสริมเป็น $O(h)$ ด้วย ดังนั้นหากต้นไม้สูง ก็เป็นเรื่องไม่ดี ทั้งเวลาและเนื้อที่หน่วยความจำ

ขอย้ำอีกครั้งว่า getNode(r,e) คืนปมที่พบ e ส่วน get(e) คืนข้อมูลที่ปมซึ่งมีค่าเท่ากับ e ผู้อ่านบางคนอาจตีความว่า "get(e) คืน e ถ้าต้นไม้มี e เก็บอยู่" ต้องขอเน้นว่าไม่ถูกต้อง ที่ถูกต้องเป็น "get(e) คืนข้อมูลของปมซึ่งมีค่าเท่ากับ e" กรณีที่เราเก็บข้อมูลที่ซับซ้อนตามปมของต้นไม้ เช่น ใช้ต้นไม้ค้นหาแบบทวิภาคเก็บสินค้าประเภทต่าง ๆ โดยสินค้าแต่ละชนิดมีรหัส ราคา และจำนวนกำกับตัวสินค้า ตัวสินค้ามี compareTo ที่ใช้เฉพาะรหัสสินค้าเป็นตัวเปรียบเทียบความน้อยกว่ามากกว่า และเท่ากัน การค้นหาด้วย get จึงสามารถระบุเฉพาะรหัสสินค้า ถ้าพบ จะคืนตัวสินค้าที่เก็บในต้นไม้ที่มีข้อมูลกำกับสินค้าอย่างสมบูรณ์กลับไปใช้งานได้ เราจะพบการใช้งานในลักษณะนี้ที่เรียกว่า *แมป* (map) ในหัวข้อถัด ๆ ไป

การค้นหาข้อมูลน้อยสุดและมากสุด

ข้อมูลที่เก็บในต้นไม้ค้นหาแบบทวิภาคมีการจัดอันดับกันดีพอควร จัดตัวน้อยอยู่ทางซ้าย ตัวมากอยู่ทางขวา ถ้าเราจะขอข้อมูลตัวน้อยสุด หรือตัวมากสุด จะหาได้เร็วเพียงใด ? ขอเริ่มที่การหาข้อมูลน้อยสุดกันก่อน ด้วยลักษณะที่เราเก็บตัวน้อยกว่าไว้ด้านซ้าย เมื่ออยู่ที่ปมใด ปมซ้ายของปมนั้นก็ต้องมีข้อมูลน้อยกว่าปมนั้น ดังนั้นถ้าเราเริ่มที่ราก ข้อมูลที่ปมซ้ายของรากก็ต้องน้อยกว่าที่ราก ข้อมูลที่ปมซ้ายของปมซ้ายของรากก็ต้องน้อยลง ข้อมูลที่ปมซ้ายของปมซ้ายของปมซ้ายของรากก็ต้องน้อยลงไปอีก เป็นเช่นนี้ไปเรื่อย ๆ ดังนั้นการหาปมที่มีข้อมูลตัวน้อยสุด ทำได้โดยเริ่มที่ราก วิ่งลงทางปมซ้ายไปเรื่อย ๆ จนกว่าจะพบปมที่ไม่มีลูกซ้าย (แสดงว่าไม่มีปมที่มีค่าน้อยกว่าแล้ว) ก็ย่อมเป็นปมที่มีข้อมูลน้อยสุด ดังแสดงในรหัสที่ 10-8 เมท็อด `getMin` เริ่ม `r` ที่ราก เข้าวงวนหมุนตราบเท่าที่ยังมีลูกซ้ายด้วยเงื่อนไข `r.left != null` ถ้ายังมีปมซ้ายก็เปลี่ยน `r` เป็นปมซ้ายด้วย `r = r.left` เมื่อหลุดจากวงวน แสดงว่าปม `r` ไม่มีลูกซ้าย เป็นปมที่มีข้อมูลน้อยสุด จึงคืน `r.element` กลับไป

```
  public Object getMin() {
    Node r = root;
    if (r == null) return null;
    while (r.left != null) {
      r = r.left;
    }
    return r.element;
  }
```

ลงไปยังตัวน้อยกว่าตราบเท่าที่ยังมีตัวน้อยกว่า

ไม่มีตัวน้อยกว่า แสดงว่าเป็นตัวน้อยสุด

รหัสที่ 10-8 เมท็อด `getMin` คืนข้อมูลน้อยสุดในต้นไม้

ต้องการหาตัวน้อยสุด ให้วิ่งดิ่งลงทางซ้าย ดังนั้นถ้าต้องการหาตัวมากสุด ก็วิ่งดิ่งลงทางขวา ดังแสดงในรหัสที่ 10-9 อ้อลืมบอกไปว่า ขอกำหนดให้ `getMin` และ `getMax` คืน `null` เมื่อเรียกกับต้นไม้ที่ไม่มีข้อมูลเลย

```
  public Object getMax() {
    Node r = root;
    if (r == null) return null;
    while (r.right != null) {
      r = r.right;
    }
    return r.element;
  }
```

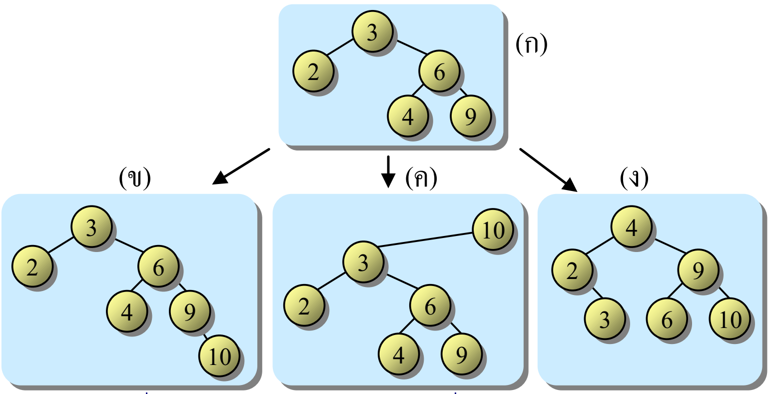
ลงไปยังตัวมากกว่าตราบเท่าที่ยังมีตัวมากกว่า

ไม่มีตัวมากกว่า แสดงว่าเป็นตัวมากสุด

รหัสที่ 10-9 เมท็อด `getMax` คืนข้อมูลมากสุดในต้นไม้

การเพิ่มข้อมูล

จุดประสงค์ของการเพิ่มข้อมูลก็คือ หลังจากเพิ่มข้อมูลแล้ว มีการเพิ่มปมใหม่ซึ่งเก็บข้อมูลใหม่เข้าไปเป็นส่วนหนึ่งของต้นไม้ ต้นไม้มีจำนวนปมเพิ่มขึ้นอีกหนึ่ง โดยที่การจัดเก็บข้อมูลต้องเป็นไปตามกฎเกณฑ์ของต้นไม้ค้นหาแบบทวิภาค สิ่งที่เราต้องพิจารณาคือ ปมใหม่หนึ่งปมที่เก็บข้อมูลใหม่นั้นจะไปอยู่ที่ใดในต้นไม้ ต้นไม้จะเปลี่ยนโครงสร้างไปมากน้อยแค่ไหนหลังการเพิ่ม และใช้เวลาในการเพิ่มเท่าใด พิจารณาต้นไม้ (ก) ในรูปที่ 10-4 ถ้าเราเพิ่ม 10 เข้าไปในต้นนี้ จะได้ผลอย่างไร ต้นไม้ด้านล่างทั้งสามต้นในรูป ต่างก็เป็นผลลัพธ์ที่ถูกต้องตามกฎทั้งสิ้น เนื่องจากต้นไม้ใหม่มี 10 เป็นข้อมูลใหม่ที่เพิ่มขึ้น และต้นไม้ทั้งสามนี้ก็ยังรักษาความสัมพันธ์ของข้อมูลตามกฎของต้นไม้ค้นหาแบบทวิภาค



รูปที่ 10-4 ตัวอย่างต้นไม้หลังการเพิ่ม 10 ในต้นไม้ต้นบน

หากเราพิจารณาโดยคำนึงว่า ต้นไม้หลังการเพิ่มน่าจะมีรูปร่างที่เป็นประโยชน์ต่อการค้นหา เราก็คงอยากได้ต้นไม้หลังการเพิ่มเป็นแบบต้น (ง) เนื่องจากเป็นต้นไม้ที่เตี้ยสุด ย่อมทำให้ประสิทธิภาพการค้นหาดีตามไปด้วย (เพราะว่า เวลาการค้นหาเป็น $O(h)$) ปัญหาที่ตามมาก็คือ ต้นไม้ก่อนและหลังเพิ่มนั้นมีตำแหน่งของข้อมูลต่างกันโดยสิ้นเชิง ต้องมีการปรับตัวชี้มากมาย อีกทั้งยังไม่รู้เลยด้วยว่าจะออกแบบขั้นตอนการเพิ่มอย่างไรเพื่อเปลี่ยนต้นไม้ให้เป็นไปตามที่ต้องการได้รวดเร็ว

แต่ถ้าเราพิจารณาต้น (ข) และ (ค) จะพบว่า ข้อมูลเดิมในต้นไม้หลังการเพิ่มมีความสัมพันธ์เหมือนเดิมทุกประการ ปมใหม่ถูกเพิ่มเข้าไปในสองลักษณะที่แตกต่างกัน ต้น (ข) 10 ถูกเพิ่มเป็นใบใหม่ ส่วนต้น (ค) 10 ถูกเพิ่มเป็นรากใหม่ ในงานจัดเก็บข้อมูลบางงาน ผู้ออกแบบรู้ว่า ข้อมูลที่เพิ่มใหม่จะเป็นข้อมูลที่ได้รับการอ้างอิงบ่อยในอนาคต ถ้าเป็นเช่นนี้การเพิ่มข้อมูลใหม่ให้เป็นรากของต้นไม้ย่อมเป็นผลดี แต่เราคงไม่สามารถนำข้อมูลใหม่ใส่ปมแล้วต่อเป็นรากได้ง่าย ๆ ดังในรูป (ค) (ใน

ตัวอย่างเราโชคดีที่ 10 มีค่ามากกว่าข้อมูลทุกตัวในต้นไม้เดิม จึงต่อเป็นรากใหม่ได้) วิธีที่ให้ได้ปมใหม่เป็นรากคงต้องมีกระบวนการที่ซับซ้อน

มาดูกรณีการเพิ่มปมใหม่ให้เป็นใบในต้น (ข) ตามทฤษฎี ต้นไม้ที่มี n ปม จะมีตำแหน่งที่ให้เพิ่มเป็นใบใหม่ได้ n+1 (ลองคิดดูสักครู่ หรือจะใช้อุปนัยทางคณิตศาสตร์พิสูจน์ก็ได้) ปัญหาจึงอยู่ที่ว่าจะนำปมใหม่ไปต่อเป็นใบที่ตำแหน่งใด แนวคิดง่าย ๆ ก็คือถ้าจะเพิ่ม `e` ก็ให้ลองค้นหา `e` ในต้นไม้ ซึ่งย่อมต้องค้นหาไม่พบ (เพราะเราจะเก็บแบบเซต ถ้าพบ `e` ในต้นไม้ ก็จะไม่เพิ่ม) การค้นหาไม่พบนี้จะต้องจบที่ต้นไม้ว่าง (ซึ่งคือ `null`) ซึ่งต้องเป็นลูกของปมใดปมหนึ่งในต้น เราก็นำใบใหม่ที่เก็บ `e` ต่อเป็นลูกแทน `null` ตัวนั้น (จะมีข้อยกเว้นก็เฉพาะการเพิ่มข้อมูลตัวแรกในต้นไม้ว่าง ปมใหม่จะเป็นรากของต้นใหม่ด้วย) จากตัวอย่างการเพิ่ม 10 ในต้น (ก) ของรูปที่ 10-4 ถ้าค้นหา 10 จะผ่านปม 3, 6, 9, และจบที่ลูกขวาของปม 9 ซึ่งเป็น `null` ก็สร้างใบใหม่เก็บ 10 แล้วต่อให้เป็นลูกขวาของปม 9

```
37    public void add(Object e) {
38      if (e == null) throw new IllegalArgumentException();
39      Node newNode = new Node(e, null, null);
40      if (root == null) root = newNode;
41      else {
42        Node p = null, r = root;
43        while( r != null ) {
44          if      (compare(e, r.element) < 0) {p = r; r = r.left;}
45          else if (compare(e, r.element) > 0) {p = r; r = r.right;}
46          else return;
47        }
48        if (compare(e, p.element) < 0)
49          p.left = newNode;
50        else
51          p.right = newNode;
52      }
53      ++size;
54    }
```

ไม่อนุญาตให้เพิ่ม null

ต้องการให้ p เก็บปมพ่อของ r

ไม่ทำอะไร ถ้าเพิ่มตัวซ้ำ

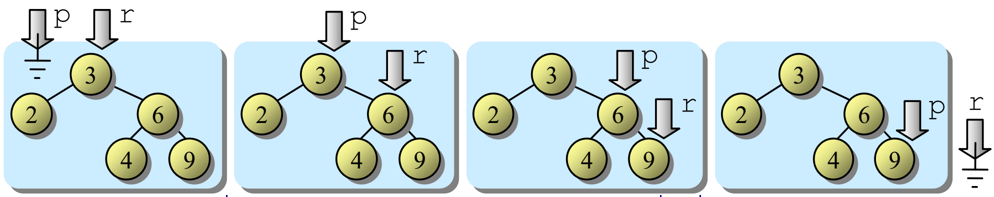
จะเปลี่ยน r ก็ให้เปลี่ยน p ตาม

เพิ่มใบใหม่เป็นลูกของปมสุดท้ายที่ผ่าน

รหัสที่ 10-10 เมท็อด add เพิ่มข้อมูลใหม่ โดยสร้างใบใหม่เพิ่มใส่ต้นไม้

รหัสที่ 10-10 แสดงเมท็อด `add` ที่เพิ่มข้อมูลใหม่ให้เป็นใบใหม่ด้วยวิธีข้างต้น เริ่มด้วยการสร้างปมใหม่ เก็บข้อมูลที่ได้รับ และทำให้เป็นใบ (บรรทัดที่ 39) แล้วเริ่มตรวจสอบว่า เป็นต้นไม้ว่างหรือไม่ ถ้าใช่ ก็ตั้งรากของต้นไม้เป็นปมที่เพิ่งสร้าง แต่ถ้าไม่ว่าง ก็เริ่มค้น โดยระหว่างวงวนการค้นมีตัวแปร `p` และ `r` ซึ่งตั้งใจให้ `p` เป็นปมพ่อของ `r` เริ่มให้ `r` เป็น `root` (รากไม่มีปมพ่อก็เลยตั้งให้ `p` เป็น `null` ตอนเริ่มต้น) ทำวงวนตราบเท่าที่ `r` ไม่เป็น `null` เปรียบเทียบข้อมูลใหม่กับที่ปม `r` ถ้าข้อมูลใหม่น้อยกว่าก็เลี้ยวซ้าย โดยให้ `p=r` ก่อนที่จะเปลี่ยน `r` ไปชี้ปมซ้าย (บรรทัดที่ 44) ถ้าข้อมูลใหม่มากกว่าก็เลี้ยวขวา (บรรทัดที่ 45) ในกรณีที่ข้อมูลใหม่เท่ากับที่ปม `r` ก็ `return` เลย เพราะซ้ำ

(บรรทัดที่ 46) หมุนทำในวงวนจน r เป็น null หลุดออกมา มี p เป็นปมสุดท้ายที่ผ่าน เป็นพ่อของ null ถ้าข้อมูลใหม่น้อยกว่าของที่ p ก็เปลี่ยนลูกซ้ายของ p เป็นปมใหม่ ไม่เช่นนั้นก็เปลี่ยนลูกขวาของ p (บรรทัดที่ 48 ถึง 51) ปิดท้ายด้วยการเพิ่มตัวแปร size เพื่อบอกว่า จำนวนข้อมูลเพิ่มอีกหนึ่ง รูปที่ 10-5 แสดงตัวอย่างการเปลี่ยนแปลง p และ r ระหว่างการเพิ่ม 10 ในต้นไม้



รูปที่ 10-5 ตัวอย่างการหาตำแหน่งของใบใหม่เพื่อเพิ่มข้อมูล 10

คราวนี้ขอเขียนเมท็อด add ในลักษณะการทำงานแบบเวียนเกิด (เพราะเป็นหลักการทำงานของเมท็อดอื่นที่จะตามมา) กำหนดให้มีเมท็อดภายในชื่อ add(Node r,Object e) ซึ่งมีหน้าที่เพิ่ม e ให้กับต้นไม้ที่มีปม r เป็นราก โดยผลที่ได้จากเมท็อดนี้คือรากของต้นไม้หลังเพิ่มเสร็จ (รากของต้นไม้หลังการเพิ่มอาจเปลี่ยนได้) รหัสที่ 10-11 แสดงรายละเอียดการเพิ่มข้อมูล เมท็อด add(r,e) แบ่งการทำงานเป็นสองกรณี ถ้า r เป็น null ก็สร้างปมใหม่ให้ r, เพิ่มจำนวนข้อมูล, แล้วคืน r เป็นผลลัพธ์ (นี่เองคือกรณีที่หลังเพิ่มแล้วรากเปลี่ยน) ในกรณีที่ r ไม่เป็น null ก็เทียบ e กับ r.element แบ่งเป็นอีกสามกรณีย่อย

1. ถ้า e น้อยกว่า r.element ให้เพิ่ม e ในลูกซ้ายของ r ด้วย add(r.left,e) เนื่องจากรากของลูกต้นซ้ายหลังการเพิ่มอาจมีการเปลี่ยนแปลง จึงต้องตั้งให้ r.left มีค่าเท่ากับรากของลูกต้นซ้ายหลังเพิ่ม e ด้วย (บรรทัดที่ 48)
2. ถ้า e มากกว่า r.element ให้เพิ่ม e ในลูกขวาของ r ด้วย add(r.right,e) เนื่องจากรากของลูกต้นขวาหลังการเพิ่มอาจมีการเปลี่ยนแปลง จึงต้องตั้งให้ r.right มีค่าเท่ากับรากของลูกต้นขวาหลังเพิ่ม e ด้วย (บรรทัดที่ 50)
3. ถ้า e มีค่าเท่ากับ r.element แสดงว่าข้อมูลซ้ำ ไม่ต้องทำอะไร

รูปที่ 10-6 แสดงการเปลี่ยนแปลงตัวแปร r ระหว่างการเพิ่ม 10 ในต้นไม้ โดยมีการเรียก add(r.right,e) ที่บรรทัดที่ 50 ของรหัสที่ 10-11 ซึ่งแทนการเพิ่ม e ในต้นขวาลงไปเรื่อย ๆ จากรูป (1) ถึง จนถึงรูป (4) ได้ r เป็น null จึงสร้างปมใหม่ให้ r ในรูป (5) แล้วคืน r กลับไปตั้งเป็นค่าของ r.right ที่บรรทัดที่ 50 ได้รูป (6) (ตอนนี้ r ชี้ที่ปม 9) แล้วก็คืน r กลับไป ตั้งเป็นค่าของ r.right เช่นนี้ไปเรื่อย ๆ จนถึงรูป (8) สรุปได้ว่า การเพิ่มใช้เวลาแปรตามความสูง เป็น O(h)

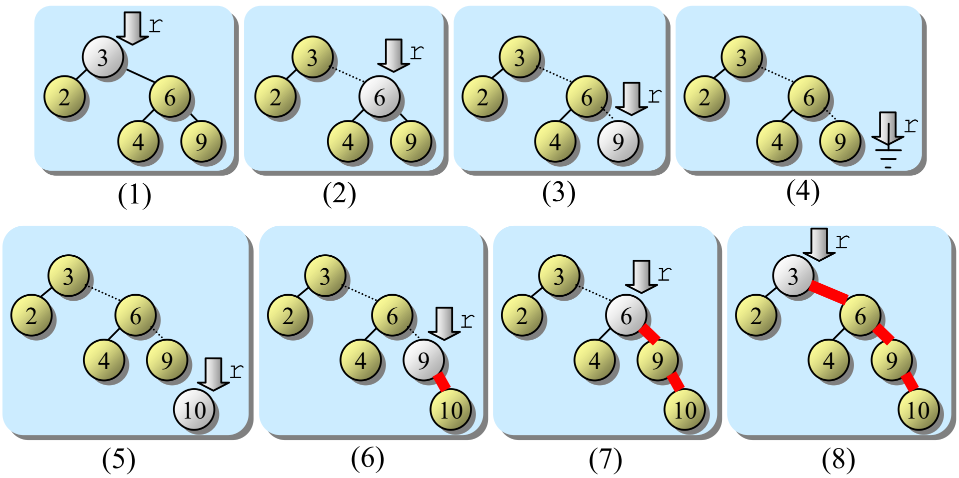
```
  public void add(Object e) {
    if (e == null) throw new IllegalArgumentException();
    root = add(root, e);
  }

  protected Node add(Node r, Object e) {
    if (r == null) {
      r = new Node(e, null, null);
      ++size;
    } else {
      if (compare(e, r.element) < 0) {
        r.left = add(r.left, e);
      } else if (compare(e, r.element) > 0) {
        r.right = add(r.right, e);
      }
    }
    return r;
  }
```

เพิ่ม e ในต้น r ได้ผลเป็นรากของต้นไม้หลังการเพิ่ม

สร้างใบใหม่

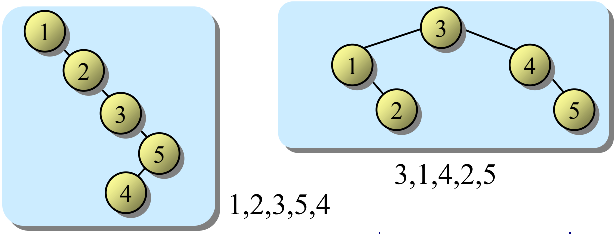
e น้อยกว่าที่ r ไปเพิ่ม e ในลูกต้นซ้าย ถ้ามากกว่าไปเพิ่มในลูกต้นขวา

รหัสที่ 10-11 เมท็อด `add` เพิ่มข้อมูลใหม่ โดยสร้างใบใหม่เพิ่มใส่ต้นไม้

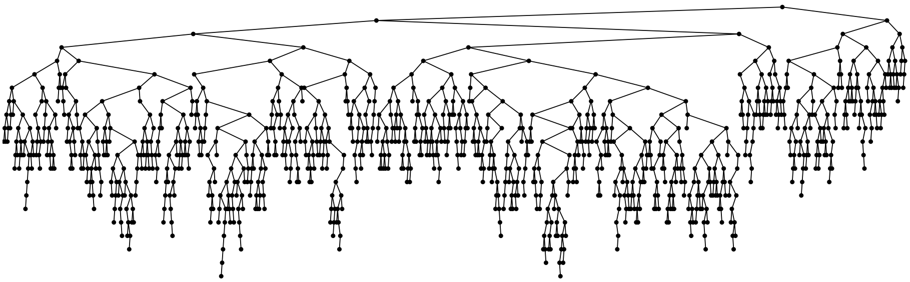


รูปที่ 10-6 ตัวอย่างการเปลี่ยน `r` และ `r.right` ระหว่างการเพิ่มข้อมูล 10

ด้วยวิธีการเพิ่มที่นำเสนอมา ลำดับของข้อมูลจะเป็นตัวกำหนดลักษณะของต้นไม้ รูปที่ 10-7 แสดงต้นไม้สองลักษณะซึ่งได้จากการลำดับการเพิ่มข้อมูลที่ต่างกัน ต้นหนึ่งสูงสุด อีกต้นเตี้ยสุด ได้มีผู้ทำการวิเคราะห์ว่า หากเรานำข้อมูลสุ่มมาเพิ่มใส่ต้นไม้ค้นหาแบบทวิภาคจำนวน n ตัว (n มีค่ามาก ๆ) จะได้ต้นไม้สูงประมาณ $4.31107 \ln n - 1.953 \ln \ln n \approx 2.99 \log_2 n$ ตีความได้ว่า ถึงแม้ว่าความสูงของต้นไม้ค้นหาแบบทวิภาคจะอยู่ในช่วง $\lfloor \log_2 n \rfloor \le h \le (n-1)$ ก็ตาม แต่ถ้าเรานำข้อมูลที่มีลักษณะสุ่ม มาสร้างต้นไม้ จะได้ต้นไม้ที่สูงประมาณเพียงสามเท่าของต้นที่เตี้ยสุด รูปที่ 10-8 แสดงตัวอย่างต้นไม้ค้นหาแบบทวิภาคที่สร้างจากข้อมูลสุ่มจำนวน 1,000 ตัว



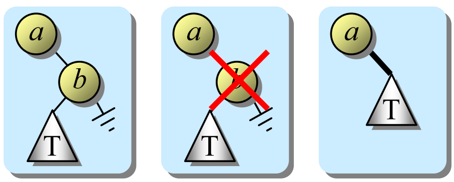
รูปที่ 10-7 ต้นไม้ค้นหาแบบทวิภาคสองลักษณะที่ได้จากลำดับการเพิ่มข้อมูลต่างกัน



รูปที่ 10-8 ตัวอย่างต้นไม้ค้นหาแบบทวิภาคที่สร้างจากข้อมูลสุ่มจำนวน 1,000 ตัว

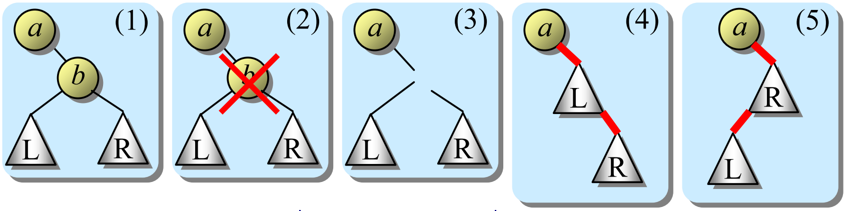
การลบข้อมูล

การลบปมออกจากต้นไม้จะง่าย ถ้าปมนั้นมีลูกหนึ่งเป็น `null` รูปที่ 10-9 แสดงการลบปม b โดยที่ b มีลูกข้างหนึ่งเป็น `null` เราเพียงแค่ยกลูกอีกข้างมาแทนตัวเอง นั่นคือให้ a ซึ่งคือปมพ่อของ b เปลี่ยนจากที่เคยชี้ b มาชี้ลูกของ b แทน ก็เสมือนเป็นการลบ b ออกไป



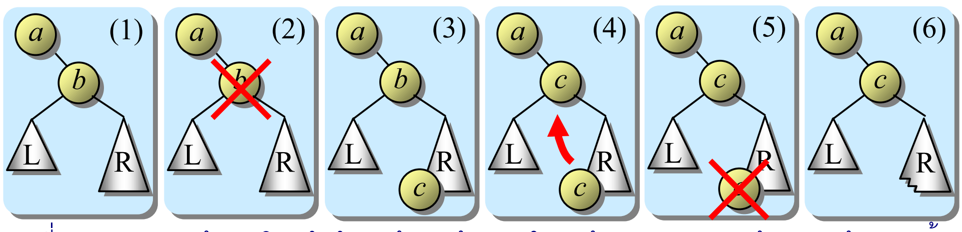
รูปที่ 10-9 การลบปมที่มีลูกข้างหนึ่งเป็น `null` (อีกข้างเป็น `null` ด้วยก็ได้)

กรณีที่ต้องลบปมที่มีสองลูกจะซับซ้อนกว่า พิจารณาการลบปม b ออกในรูปที่ 10-10 (1) จะแบ่งต้นเดิมออกเป็นสามส่วน (รูป (3)) สิ่งที่เราต้องทำก็คือ ประกบต้นย่อยทั้งสองกลับเป็นต้นเดียว วิธีแรกคือยกให้ต้นย่อยซ้าย L ของ b ขึ้นมา ซึ่งคือให้ขึ้นมาเป็นลูกของ a แทน b จากนั้น นำรากของต้นย่อยขวา R มาต่อเป็นลูกขวาของปมที่มีค่ามากสุดในต้น L (รูป (4)) หรือจะทำอีกแบบคือยกให้ต้นย่อยขวา R ของ b ขึ้นมา แล้วนำรากของต้นย่อยซ้าย L มาต่อเป็นลูกซ้ายของปมที่มีค่าน้อยสุดในต้น R (รูป (5)) ต้นไม้ผลลัพธ์ที่ได้ (ไม่ว่าจะเป็นดังรูป (4) หรือ (5)) ยังคงเป็นต้นไม้ค้นหาแบบทวิภาค



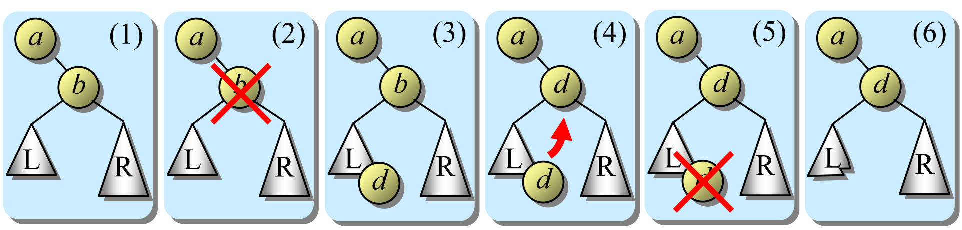
รูปที่ 10-10 การลบปมที่มีสองลูก

ผู้อ่านดูรูปที่ 10-10 ก็จะเห็นข้อเสียของวิธีลบปมข้างต้น ให้สังเกตว่า ต้นไม้อะไรกันลบปมแล้วสูงขึ้น ! การลบปมออกจากต้นไม้ควรทำให้ต้นไม้เตี้ยลง หรือไม่ก็สูงเท่าเดิม จะขอเสนอวิธีลบอีกแบบที่เราจะใช้กัน (ดูรูปที่ 10-11 ประกอบ) ถ้าเราต้องการลบข้อมูลในปม *b* ซึ่งมีทั้งลูกซ้ายและขวาแน่นอนว่า ลูกต้นขวาของ *b* ต้องมีปมที่มีข้อมูลน้อยสุด ซึ่งคือปม *c* แสดงในรูป (3) เราสรุปได้ว่า ปม *c* เก็บข้อมูลที่ต้องมีค่ามากกว่าทุก ๆ ปมในต้น L และมีค่าน้อยกว่าทุกปมในต้น R ดังนั้นการลบข้อมูลในปม *b* ออกจากต้นไม้ ทำได้โดยไม่ต้องลบปม *b* หรอก แต่ทำได้โดยทำสำเนาข้อมูลในปม *c* มาเก็บแทนที่ข้อมูลในปม *b* ดังรูป (4) เพียงเท่านี้ข้อมูลในปม *b* ก็เสมือนถูกลบออกจากต้นไม้ จากนั้นภาระที่เหลือคือการลบปม *c* ในต้น R ออกไป ซึ่งลบได้ง่าย เพราะเนื่องจากปม *c* เก็บค่าน้อยสุดใน R ย่อมเป็นปมที่ไม่มีลูกซ้ายแน่นอน นั่นคือมีลูกข้างซ้ายเป็น `null` ซึ่งลบได้ง่ายด้วยวิธีที่กล่าวไว้ก่อนหน้านี้



รูปที่ 10-11 การลบข้อมูลโดยทำสำเนาข้อมูลน้อยสุดในลูกต้นขวามาแทน แล้วลบตัวน้อยสุดนั้น

หรือจะลบในอีกรูปแบบหนึ่ง (ดูรูปที่ 10-12 ประกอบ) คือถ้าต้องการลบข้อมูลในปม *b* ออก ก็ให้นำข้อมูลตัวมากสุดในลูกต้นซ้ายของ *b* มาแทนข้อมูลใน *b* แล้วไปลบปมที่เก็บตัวมากสุดในลูกต้นซ้ายนั้นทิ้งไป



รูปที่ 10-12 การลบข้อมูลโดยทำสำเนาข้อมูลมากสุดในลูกต้นซ้ายมาแทน แล้วลบตัวมากสุดนั้น

```
55    public void remove(Object e) {
56      root = remove(root, e);
57    }
58    protected Node remove(Node r, Object e) {
59      if (r == null) return r;
60      if (compare(e, r.element) < 0) {
61        r.left = remove(r.left, e);
62      } else if (compare(e, r.element) > 0) {
63        r.right = remove(r.right, e);
64      } else {
65        if (r.left == null || r.right == null) {
66          r = (r.left == null ? r.right : r.left);
67          --size;
68        } else {
69          Node m = r.right;
70          while (m.left != null) m = m.left;
71          r.element = m.element;
72          r.right = remove(r.right, m.element);
73        }
74      }
75      return r;
76    }
```

ลบ e ในต้น r ได้ผลเป็นราก ของต้นไม้หลังการลบ

e น้อยกว่าที่ r ไปลบในลูกต้นซ้าย ถ้ามากกว่าไปลบในลูกต้นขวา

อยากลบ r ที่ไม่มีลูก หรือมีลูกเดียว

อยากลบ r ที่มี 2 ลูก

หาปมน้อยสุดในลูกต้นขวาของ r

สำเนาข้อมูลน้อยสุดที่หาพบไว้ที่ r

ลบปมน้อยสุดในลูกต้นขวาของ r

รหัสที่ 10-12 เมท็อด `remove` ลบข้อมูล

รหัสที่ 10-12 แสดงรายละเอียดเมท็อด `remove(e)` ทำหน้าที่ค้นหาและลบ `e` ออกจากต้นไม้ (ถ้าหาไม่พบก็ไม่ต้องทำอะไร) โดยอาศัยเมท็อด `remove(r,e)` ซึ่งค้นหาและลบ `e` ออกจากต้นไม้ที่มี `r` เป็นราก ผลลัพธ์ของเมท็อดเป็นรากใหม่ของต้นหลังการลบ เมท็อดนี้มีโครงการทำงานคล้ายกับการเพิ่ม ใน `add(r,e)` เราค้นไปเรื่อย ๆ จนพบ `null` แล้วจึงเพิ่ม ถ้าพบค่าซ้ำจะไม่ทำอะไร แต่ใน `remove(r,e)` เราค้นไปเรื่อย ๆ จนพบค่า `e` แล้วจึงลบ หากพบ `null` จะไม่ทำอะไร เริ่มการทำงานที่บรรทัดที่ 59 ตรวจสอบว่า ถ้า `r` เป็น `null` ก็คืน `r` กลับไปทันที ถ้าไม่ใช่ `null` และ `e` น้อยกว่า `r.element` ก็ต้องไปค้นและลบในลูกต้นซ้ายด้วย `remove(r.left,e)` ได้ผลเป็นรากใหม่หลังการลบ ตั้งเป็นค่าให้กับ `r.left` (บรรทัดที่ 61) ถ้า `e` มากกว่า `r.element` ก็ให้ไปค้นและลบในลูกต้นขวา (บรรทัดที่ 63) แต่ถ้าไม่น้อยกว่าและไม่มากกว่า นั่นคือพบข้อมูลที่ต้องการลบแล้ว (บรรทัดที่ 64) ก็ตรวจสอบก่อนว่า ถ้าเป็นกรณีง่ายที่มีลูกใดข้างหนึ่ง (หรือทั้งสองข้างก็ได้) เป็น `null` (บรรทัดที่ 65) ก็ให้นำลูกอีกข้างมาเป็นค่าใหม่ใน `r` (บรรทัดที่ 66) เพื่อคืนกลับไปเป็นผลของการลบ ตามด้วยการลดจำนวนข้อมูลลงหนึ่ง แต่ถ้าเป็นกรณีที่ทั้งสองลูกไม่เป็น `null` (บรรทัดที่ 68) ก็ลงไปที่ลูกต้นขวาเพื่อหาค่าน้อยสุด (โดยให้ `m` ชี้ลูกต้นขวา แล้วลงไปทางซ้ายด้วย `m=m.left` ตราบเท่าที่ยังมีลูกซ้าย หลุดจากวงวนในบรรทัดที่ 70 จะได้ `m.element` เป็นข้อมูลน้อยสุดที่ต้องการ) ทำสำเนาใส่ `r.element` เป็นการลบ `r.element` ของเดิมที่มีค่าเท่ากับ `e` ออก ปิดท้าย

ด้วยการสั่งให้ไปลบ m.element ในลูกต้นขวาในบรรทัดที่ 72 และไม่ว่าจะทำงานในกรณีใด สุดท้ายก็คืนค่าของ r กลับไปเป็นรากของต้นไม้หลังการลบ (บรรทัดที่ 75)

รหัสที่ 10-12 อาศัยแนวคิดในการลบกรณีพบข้อมูลที่มีปมซึ่งมีสองลูกดังแสดงในรูปที่ 10-11 แต่ถ้าจะอาศัยแนวคิดของรูปที่ 10-12 ก็เพียงแต่เปลี่ยนบรรทัดที่ 69 ถึง 72 จาก left เป็น right จาก right เป็น left

การลบที่ซับซ้อนสุดคือ กรณีที่ลบข้อมูลที่อยู่ในปมที่มีสองลูก ก็ต้องค้นให้พบ เลี้ยวขวา วิ่งไปทางซ้ายจนตันเพื่อหาตัวน้อยสุด ใช้เวลา O(h) บวกกับเวลาที่ไปลบตัวน้อยสุดอีก O(h) รวมแล้วก็เป็น O(h) เช่นกัน

ความลึกเฉลี่ยของปม

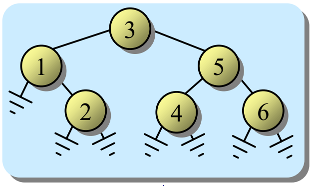
ประสิทธิภาพของบริการต่าง ๆ ของต้นไม้ค้นหาแบบทวิภาคที่ได้นำเสนอมา ไม่ว่าจะเป็น getMin, getMax, get, add, และ remove ล้วนใช้เวลาแปรตามความสูงของต้นไม้ทั้งสิ้น เราได้หยิบผลการวิเคราะห์ที่นักวิจัยได้ทำกันมาว่า ต้นไม้ค้นหาแบบทวิภาคที่สร้างจากข้อมูลสุ่มจะมีความสูงเป็น $4.31107 \ln n - 1.953 \ln \ln n + O(1)$ (วิธีวิเคราะห์ความสูงเฉลี่ยนี้ยุ่งยากเกินกว่าที่จะนำเสนอในหนังสือเล่มนี้) ยังมีลักษณะสมบัติอีกอย่างหนึ่งของต้นไม้ที่น่าสนใจคือความลึกเฉลี่ยของปมในต้นไม้ ถึงแม้ว่า ประสิทธิภาพของบริการต่าง ๆ แปรตามความสูงของต้นไม้ แต่นั่นเป็นขอบเขตกรณีโชคร้ายสุด เช่น ค้นแล้วไปจบที่ใบที่อยู่ระดับล่างสุดของต้นไม้ การค้นหาในกรณีที่พบข้อมูลที่ต้องการ จะสิ้นสุดที่ปมใดปมหนึ่งในต้นไม้ แต่ในกรณีที่ไม่พบข้อมูลที่ต้องการ การค้นหาจะไปสิ้นสุดที่ null ตัวใดตัวหนึ่งในต้นไม้ นิยามให้*ปมภายใน*คือปมต่าง ๆ ในต้นไม้ที่มีข้อมูลเก็บ และ*ปมภายนอก*คือ null ต่าง ๆ ของต้นไม้ ความลึกเฉลี่ยของปมภายใน และความลึกเฉลี่ยของปมภายนอกในต้นไม้ จึงเป็นตัวสะท้อนประสิทธิภาพการค้นหาข้อมูลในกรณีที่พบและไม่พบข้อมูลตามลำดับ

ให้ n แทนจำนวนปมภายในของต้นไม้ จะได้ว่า ต้นไม้นี้มีปมภายนอกจำนวน n+1 ปม เพราะว่า ปมภายในทุกปมมีตัวโยงสองตัว ทุก ๆ ปมต้องมีปมพ่อหนึ่งปม โดยมีหนึ่งในตัวโยงของปมพ่อชี้มาหา ยกเว้นก็เฉพาะปมรากที่ไม่มีพ่อ สรุปได้ว่า มีตัวโยงทั้งสิ้น 2n ในตัวโยงทั้งหมดนี้มีที่ไม่เป็น null เป็นจำนวน n–1 ดังนั้นมีตัวโยงที่เป็น null ซึ่งคือปมภายนอกจำนวน $2n - (n-1) = n+1$ ขอให้คำนิยามต่อไปนี้เพื่อความง่ายในการวิเคราะห์ (ดูตัวอย่างในรูปที่ 10-13 ประกอบ)

- *ความยาววิถี*จากรากถึงปม k คือจำนวนกิ่งจากรากลงมาจนถึงปม k
- *ความยาวรวมของวิถีภายใน*คือผลรวมของความยาววิถีจากรากถึงปมภายในทุกปม

- *ความยาวรวมของวิถีภายนอก*คือผลรวมของความยาววิถีจากรากถึงปมภายนอกทุกปม
- $I(n)$ คือค่าเฉลี่ยของความยาวรวมของวิถีภายใน
- $E(n)$ คือค่าเฉลี่ยของความยาวรวมของวิถีภายนอก
- $D_I(n)$ คือความลึกเฉลี่ยของปมภายใน, $D_I(n) = \dfrac{I(n)}{n}$
- $D_E(n)$ คือความลึกเฉลี่ยของปมภายนอก, $D_E(n) = \dfrac{E(n)}{n+1}$

n ที่เขียนใน $I(n)$, $E(n)$, $D_I(n)$, และ $D_E(n)$ เป็นการระบุลักษณะสมบัติของต้นไม้แบบทวิภาคที่มีปมภายใน n ปม



ความยาวรวมของวิถีภายใน = 0 + (1+1) + (2+2+2) = 8

ความยาวรวมของวิถีภายนอก = 2 + (3+3+3+3+3+3) = 20

ความลึกเฉลี่ยของปมภายใน = 8 / 6 = 1.33

ความลึกเฉลี่ยของปมภายนอก = 20 / 7 = 2.86

รูปที่ 10-13 ตัวอย่างของความยาวรวมของวิถี และความลึกเฉลี่ยของปม

ความจริงแล้วเราสามารถหา $E(n)$ ได้จาก $I(n)$ จากความสัมพันธ์ $E(n) = I(n) + 2n$ ซึ่งสามารถพิสูจน์ด้วยอุปนัยทางคณิตศาสตร์ดังนี้

ขั้นตอนฐานหลัก : $n = 0$ คือกรณีต้นไม้ว่าง ไม่มีปมภายใน มีแต่ปมภายนอก `null` หนึ่งตัว ทำให้ $E(n) = 0$, $I(n) = 0$ ตรงตามความสัมพันธ์

ขั้นตอนอุปนัย : ให้ $E(n) = I(n) + 2n$ เป็นจริงเมื่อมีปมภายใน n ปม ถ้าเราเพิ่มปมภายในอีก 1 ปม แสดงว่า ต้องมีตัวโยง `null` หนึ่งตัวในอดีตที่เปลี่ยนมาเป็นปมภายในปมใหม่ ถ้าให้ d คือความยาววิถีจากรากถึง `null` ตัวนั้น ย่อมทำให้ความยาวรวมของวิถีภายนอกลดจากเดิม d และความยาวรวมของวิถีภายในเพิ่มขึ้นจากเดิม d นอกจากนี้ปมภายในใหม่จะมี `null` เพิ่มอีกสองตัวทำให้ความยาวรวมของวิถีภายนอกเพิ่มขึ้นอีก $2(d + 1)$ รวม ๆ แล้ว $E(n+1) = E(n) - d + 2(d + 1)$ และ $I(n+1) = I(n) + d$ จะได้ $E(n+1) - I(n+1) = (E(n) - d + 2(d+1)) - (I(n) + d) = E(n) - I(n) + 2 = 2n + 2 = 2(n+1)$ สรุปได้ว่า $E(n+1) - I(n+1) = 2(n+1)$ ดังนั้นความสัมพันธ์ $E(n) = I(n) + 2n$ เป็นจริง สำหรับจำนวนเต็ม $n \geq 0$ ♦

ถึงตรงนี้เรารู้แล้วว่า เพียงแค่หา $I(n)$ ตัวเดียว ก็สามารถคำนวณ $E(n)$, $D_I(n)$, และ $D_E(n)$ ได้จากสูตรข้างต้น ภาระที่เหลือก็คือการหา $I(n)$ ซึ่งเป็นค่าเฉลี่ยของความยาวรวมของวิถีภายในของต้นไม้แบบทวิภาคที่มีปมภายใน n ปม ให้ต้นไม้ L และ R มีปมภายในจำนวน k และ $n - k - 1$ ปมตามลำดับ ต้น L และ R ย่อมมีค่าเฉลี่ยของความยาวรวมของวิถีภายในเป็น $I(k)$ และ $I(n - k - 1)$

ตามลำดับด้วย หากนำ L มาต่อเป็นลูกต้นซ้ายของปมใหม่ปมหนึ่ง และนำ R มาต่อเป็นลูกต้นขวาของปมใหม่นั้น จะได้ต้นไม้ที่มีปมภายในรวมเป็น n ปม โดยทุกปมในต้น L และ R เดิมจะอยู่ระดับต่ำลงหนึ่งระดับ ส่งผลให้ทุกปมในต้น L และ R มีความยาววิถีจากรากใหม่ถึงแต่ละปมในต้นเพิ่มขึ้นอีกหนึ่ง ดังนั้นความยาวรวมของวิถีภายในของต้นใหม่คือ $I(k) + k + I(n - k - 1) + (n - k - 1) = I(k)+ I(n - k - 1) + (n - 1)$

ความยาวรวมของวิถีภายในขึ้นกับรูปร่างของต้นไม้ โดยรูปร่างของต้นไม้ค้นหาแบบทวิภาคจะเป็นเช่นไรขึ้นกับลำดับข้อมูลที่ส่งไปเพิ่ม กำหนดให้มีข้อมูล n ตัวที่มีค่าต่างกันหมด หากข้อมูลตัวน้อยสุดอันดับที่ k+1 ถูกเลือกมาเป็นราก เมื่อนำข้อมูลที่เหลือเพิ่มในต้นจนครบ ย่อมมีข้อมูลที่น้อยกว่าราก k ตัว แสดงว่า ลูกต้นซ้ายของรากมีปมภายใน k ปม ที่เหลืออีก $n - k - 1$ ตัวก็ต้องอยู่ในลูกต้นขวาของราก ถ้าให้ข้อมูลทุกตัวมีโอกาสเท่ากันที่จะเป็นราก การสร้างต้นไม้แบบทวิภาคที่มีปมภายใน n ปม จะได้ลูกต้นซ้ายของรากมีจำนวนปมภายในได้ตั้งแต่ 0 ถึง $n - 1$ ปม ดังนั้นค่าเฉลี่ยของความยาวรวมของวิถีภายในจึงเท่ากับผลรวมของ $I(k)+ I(n - k - 1) + (n - 1)$ โดยที่ $k = 0, 1, ..., (n - 1)$ หารด้วย n ได้ $I(n)=\frac{1}{n}\sum_{k=0}^{n-1}\big(I(k)+I(n-k-1)+n-1\big) = \frac{2}{n}\sum_{k=0}^{n-1}I(k)+(n-1)$ หาผลเฉลย ได้ดังนี้

$$nI(n)=2\sum_{k=0}^{n-1}I(k)+n(n-1)$$ คูณ n ตลอด

$$(n-1)I(n-1)=2\sum_{k=0}^{n-2}I(k)+(n-1)(n-2)$$ เปลี่ยน n เป็น n–1

$$nI(n)=(n+1)I(n-1)+2(n-1)$$ นำสองความสัมพันธ์ข้างต้นมาลบกัน ย้าย $(n–1)I(n–1)$ มาทางขวา

$$\begin{aligned}\frac{I(n)}{n+1}&=\frac{I(n-1)}{n}+\frac{2(n-1)}{n(n+1)}\\&=\frac{I(1)}{2}+2\sum_{i=2}^{n}\frac{(i-1)}{i(i+1)}\\&\approx 2\sum_{i=2}^{n}\frac{1}{(i+2)}\\&=2\left(\frac{1}{4}+\frac{1}{5}+\cdots+\frac{1}{n}+\frac{1}{n+1}+\frac{1}{n+2}\right)\\&=2\left(\left(H_n-1-\frac{1}{2}-\frac{1}{3}\right)+\frac{1}{n+1}+\frac{1}{n+2}\right)\end{aligned}$$

หาร $n(n+1)$ ตลอด แล้วคลี่ความสัมพันธ์เวียนเกิด โดยที่ $I(1) = 0$ จากนั้นใช้การประมาณ $\frac{(i-1)}{i(i+1)}\approx\frac{1}{i+2}$ เพื่อเปลี่ยนพจน์ในผลรวมให้ง่ายขึ้น ซึ่งเมื่อเขียนแจกแจงออกมาพบว่า สามารถเขียนในรูปของจำนวนฮาร์มอนิก H_n ซึ่งเท่ากับ $1+\frac{1}{2}+\frac{1}{3}+...+\frac{1}{n}$ มีค่าประมาณ $\ln n + \gamma$ โดยที่ γ คือค่าคงตัวของออย์เลอร์ มีค่าประมาณ 0.577...

$$I(n) \approx 2(n+1)\left(H_n - \frac{11}{6} + \frac{1}{n+1} + \frac{1}{n+2}\right)$$ คูณด้วย $(n+1)$ ตลอด แล้วประมาณให้ $(n+1)/(n+2)$ มีค่าประมาณ 1 เมื่อ n มีค่ามาก

$$\approx 2nH_n + 2H_n - \frac{11n}{3} + \frac{1}{3}$$ ประมาณค่าของ H_n ด้วย $\ln n + \gamma$

$$\approx 2n(\ln n + \gamma) + 2(\ln n + \gamma) - \frac{11n}{3} + \frac{1}{3}$$ สรุปได้ว่า $I(n) = O(n \log n)$

$$= O(n \log n)$$

$$D_I(n) = \frac{I(n)}{n}$$ นำ $I(n)$ ที่ได้มาหา $D_I(n)$

$$\approx 2(\ln n + \gamma) + \frac{2(\ln n + \gamma)}{n} - \frac{11}{3} + \frac{1}{3n}$$ ตัดพจน์ $1/3n$ ที่โตช้าสุดออก จากนั้นประมาณค่าของ H_n ด้วย $\ln n + \gamma$

$$\approx 2\ln n + 2\gamma - \frac{11}{3}$$ แล้วแปลง ln เป็น $\log_2$

$$\approx 1.386 \log_2 n - 2.513 = O(\log n)$$ สรุปได้ว่า $D_I(n) = O(\log n)$

เมื่อรู้ $D_I(n)$ ก็สามารถคำนวณ $D_E(n) = \frac{E(n)}{(n+1)} = \frac{I(n)+2n}{(n+1)} = \frac{nD_I(n)+2n}{(n+1)} \approx D_I(n) + 2$ สรุปได้ว่าความลึกเฉลี่ยของทั้งปมภายในและภายนอกของต้นไม้ค้นหาแบบทวิภาคที่สร้างแบบสุ่มเป็น $O(\log n)$ หมายความว่า การค้นข้อมูลในต้นไม้ค้นหาแบบทวิภาคที่สร้างจากข้อมูลที่มีลักษณะสุ่ม ไม่ว่าจะเป็นกรณีหาข้อมูลพบ หรือไม่พบ จะใช้เวลาเป็น $O(\log n)$

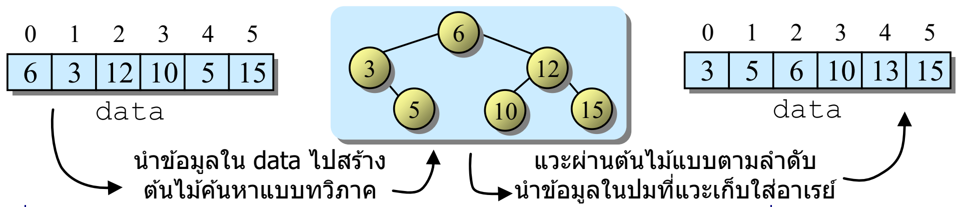
เพื่อยืนยันความแม่นยำของการวิเคราะห์ความสูงและความลึกเฉลี่ยของปมในต้นไม้ค้นหาแบบทวิภาคที่สร้างจากข้อมูลสุ่ม ผู้เขียนได้ลองเขียนโปรแกรมสร้างต้นไม้ค้นหาแบบทวิภาคจากจำนวนข้อมูลสุ่มขนาดต่าง ๆ ตั้งแต่หนึ่งพันถึงสองล้านตัว ในแต่ละกรณีสร้างต้นไม้ 40 ต้นที่มีจำนวนข้อมูลเท่ากัน วัดความสูงและความลึกปมภายใน แล้วนำมาเฉลี่ย ได้ค่าที่เปรียบเทียบกับค่าที่ได้จากการคำนวณ ดังตารางที่ 10-1 คอลัมน์ $\%\Delta h$ และ $\%\Delta$D แสดงเปอร์เซ็นต์ความคลาดเคลื่อนจากที่วัดได้จริงกับที่คำนวณ พบว่า ความสูงมีเปอร์เซ็นต์ความคลาดเคลื่อนมากกว่า ในขณะที่ความลึกนั้นใกล้เคียงมาก แต่ทั้งคู่ก็แสดงแนวโน้มของความคลาดเคลื่อนที่ลดลงเมื่อจำนวนข้อมูลมากขึ้น ๆ

ตารางที่ 10-1 ผลการวัดความสูงและความลึกปมของต้นไม้ค้นหาแบบทวิภาคที่สร้างจากข้อมูลสุ่ม

n	ความสูงเฉลี่ยที่วัดได้	4.31107 ln n – 1.953 ln ln n	%Δh	ความลึกเฉลี่ยของปมภายในที่วัดได้	2ln n – 2.513	%ΔD
1,000	20.65	26.01	25.93%	10.97	11.30	3.03%
5,000	27.80	32.53	17.03%	14.25	14.52	1.90%
10,000	30.40	35.37	16.35%	15.62	15.91	1.84%
50,000	36.75	41.99	14.27%	18.82	19.13	1.63%
100,000	40.23	44.86	11.52%	20.29	20.51	1.10%
500,000	46.20	51.54	11.57%	23.29	23.73	1.90%
1,000,000	48.70	54.43	11.77%	24.77	25.12	1.41%
2,000,000	52.13	57.32	9.97%	26.24	26.50	1.01%

การเรียงลำดับแบบต้นไม้

ด้วยลักษณะการจัดเก็บข้อมูลในต้นไม้ค้นหาแบบทวิภาคที่มีระเบียบ น้อยอยู่ทางซ้าย มากอยู่ทางขวา ประกอบกับลักษณะการแวะผ่านแบบตามลำดับที่แวะลูกต้นซ้ายก่อน แวะราก แล้วค่อยแวะลูกต้นขวา จึงสามารถนำมาใช้เรียงลำดับข้อมูลได้ เริ่มด้วยการนำข้อมูลในแถวลำดับที่ต้องการเรียงลำดับมาสร้างต้นไม้ค้นหาแบบทวิภาค จากนั้นแวะผ่านต้นไม้นี้แบบตามลำดับ โดยนำข้อมูลในปมที่ถูกแวะใส่กลับในแถวลำดับเรียงจากซ้ายไปขวา ดังตัวอย่างในรูปที่ 10-14 เขียนเป็นเมท็อด `treeSort` ได้ดังรหัสที่ 10-13 ซึ่งอาศัยการสร้างตัวเยี่ยมชมที่ทำงานเหมือนกับเมท็อด `toArray` ที่นำเสนอในบทที่ 9 แต่ใช้การแวะผ่านแบบตามลำดับ



รูปที่ 10-14 การแวะผ่านแบบตามลำดับในต้นไม้ค้นหาแบบทวิภาคจะได้ลำดับที่เรียงจากน้อยไปมาก

```
  public static void treeSort(final Object[] data) {
    BSTree t = new BSTree();
    for (int i=0; i<data.length; i++) t.add(data[i]);
    t.inOrder(new Visitor() {
                int k = 0;
                public void visit(Object e) {
                  data[k++] = e;
                } } );
  }
```

สร้างต้น BSTree ด้วยข้อมูลที่ได้รับ

แวะผ่านตามลำดับ ได้ข้อมูลเรียงลำดับเก็บใส่แถวลำดับ

รหัสที่ 10-13 การเรียงลำดับข้อมูลโดยการเพิ่มข้อมูลในต้นไม้ค้นหาแล้วแวะผ่านแบบตามลำดับ

การเรียงลำดับแบบนี้เร็วแค่ไหน ? เวลาการทำงานส่วนใหญ่อยู่ที่การนำข้อมูลในแถวลำดับไปสร้างต้นไม้ พิจารณาตัวอย่างในรูปที่ 10-14 ข้อมูล 6 ซึ่งเป็นราก เป็นตัวแรกที่ถูกเพิ่ม ไม่ต้องเปรียบเทียบกับตัวใดเลย ดูที่ข้อมูล 5 กว่าจะมาเพิ่มเป็นลูกขวาของ 3 นั้น ต้องผ่านสองปมที่เก็บ 6 และ 3 พอสรุปได้ว่า กว่าจะเพิ่มปม k ได้ ต้องวิ่งผ่านปมต่าง ๆ จากรากเป็นจำนวนเท่ากับความยาววิถีจากรากถึงปม k นั้น ภาระการเพิ่มข้อมูลให้ครบทั้งต้นก็ย่อมเท่ากับผลรวมของความยาววิถีจากรากถึงปมภายในทุกปม ซึ่งก็คือความยาวรวมของวิถีภายในที่เราได้ศึกษากันมานั่นเอง จึงสรุปได้ว่า กรณีข้อมูลในแถวลำดับที่ต้องการเรียงลำดับมีลักษณะสุ่ม เวลาการสร้างต้นไม้แปรตาม $I(n) \approx 2n \ln n$ การแวะผ่านใช้เวลา $\Theta(n)$ ดังนั้นการเรียงลำดับข้อมูลแบบต้นไม้นี้จึงใช้เวลาเป็น $O(n \log n)$

ต้องขอเน้นว่า treeSort จะใช้เวลาเป็น $O(n \log n)$ ก็เมื่อข้อมูลที่อยู่ในแถวลำดับเรียงแบบสุ่ม ๆ หากต้นไม้ที่สร้างได้สูง $n - 1$ ต้นไม้นี้จะมีความยาวรวมของวิถีภายใน $= 0+1+\ldots+(n-1) = n(n-1)/2 = \Theta(n^2)$ ทำให้ใช้เวลา $\Theta(n^2)$ หากเราต้องการ treeSort ที่มีประสิทธิภาพดีไม่ขึ้นกับลำดับของข้อมูลในแถวลำดับที่รับมา ก็สามารถทำได้โดยการสลับข้อมูลในแถวอย่างสุ่ม ๆ ก่อนที่จะนำไปสร้างต้นไม้ ดังแสดงในรหัสที่ 10-14 ซึ่งใช้เวลาเป็น $O(n \log n)$ ด้วยความน่าจะเป็นสูง

```
public static void randomizedTreeSort(final Object[] data) {
  for (int i=data.length-1; i>=0; i--) {
    int j = (int)(i*Math.random());
    Object t = data[j]; data[j] = data[i]; data[i] = t;
  }
  treeSort(data);
}
```

สุ่มสลับข้อมูลให้ยุ่งเหยิง ก่อนนำไปเรียงลำดับ

รหัสที่ 10-14 การเรียงลำดับข้อมูลโดยสุ่มสลับข้อมูลก่อนทำงาน

การสร้างเซต คอลเล็กชัน และแมป

ต้นไม้ค้นหาแบบทวิภาคเก็บข้อมูลไม่ซ้ำกัน จึงเหมาะมาก ๆ สำหรับเก็บข้อมูลแบบเซต แต่ถ้าหากจะเปลี่ยนให้เก็บข้อมูลซ้ำกันได้ ก็สามารถปรับปรุงการจัดเก็บเพียงเล็กน้อย นอกจากนี้ยังเหมาะกับการนำมาเก็บในลักษณะแมป (map) ซึ่งแต่ละข้อมูลเป็นคู่อันดับ (key, value) โดยที่ key เป็นข้อมูลที่ไม่ซ้ำกัน ตอนค้นหาจะใช้ key เป็นตัวค้นหา แล้วได้ value ที่คู่กับ key เป็นผลลัพธ์คืนกลับไป เสมือนเป็น*ฟังก์ชันการส่ง* (mapping function) จาก key ไปเป็น value แต่ใช้วิธีจำ แทนที่จะเป็นการคำนวณ

การสร้างเซตด้วยต้นไม้ค้นหาแบบทวิภาค

เนื่องจากตัวต้นไม้ค้นหาแบบทวิภาคเก็บข้อมูลไม่ซ้ำ จึงนำมาสร้างเซตได้ทันที รหัสที่ 10-15 แสดงคลาส BSTSet ภายในมีต้นไม้ค้นหาแบบทวิภาคเก็บข้อมูล ทุกเมท็อดส่งต่อไปให้ตัวต้นไม้จัดการต่อ

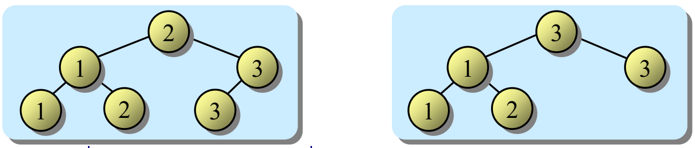
```
public class BSTSet implements Set {
  protected BSTree tree = new BSTree();

  public int size() {return tree.size();}
  public boolean isEmpty() {return tree.size() == 0;}
  public boolean contains(Object e) {return tree.get(e) != null;}
  public void add(Object e) {tree.add(e);}
  public void remove(Object e) {tree.remove(e);}
}
```

เก็บเซตด้วยต้นไม้ BSTree

รหัสที่ 10-15 การสร้างเซตด้วยต้นไม้ค้นหาแบบทวิภาค

การสร้างคอลเล็กชันด้วยต้นไม้ค้นหาแบบทวิภาค

ที่ผ่านมาการเพิ่มข้อมูล `add(r,e)` ในต้นไม้ค้นหาแบบทวิภาคที่ราก `r` จะไปเพิ่มต้นซ้ายเมื่อ `e` น้อยกว่าที่ `r` และจะไปเพิ่มต้นขวาเมื่อ `e` มากกว่า แต่ถ้าเท่ากันก็เพิกเฉยไม่สนใจ คืนการทำงาน ในกรณีที่เราต้องการให้เก็บข้อมูลซ้ำได้ ก็เปลี่ยนกฎการเพิ่มข้อมูลเล็กน้อย ซึ่งทำได้หลายวิธี เช่น ถ้า `e` เท่ากับข้อมูลที่ `r` ก็ให้ไปเพิ่มต้นซ้าย แต่นี่ก็ไม่ได้หมายความว่า ถ้าข้อมูลที่รากมีตัวซ้ำต้องอยู่ทางต้นซ้ายเสมอ เพราะการลบอาจมีการย้ายข้อมูลดังแสดงในรูปที่ 10-15 ซึ่งลบ 2 ที่รากของรูปซ้าย เกิดการย้าย 3 ขึ้นมาแทนได้ดังรูปขวา ทำให้มี 3 อีกตัวที่ทางขวาของ 3 ที่ราก อย่างไรก็ตามเหตุการณ์เช่นนี้ก็ไม่ได้ทำให้การค้นหาข้อมูลผิดพลาดแต่อย่างใด ทุกเมท็อดของคอลเล็กชันจึงยังทำงานถูกต้อง



รูปที่ 10-15 ตัวอย่างการเพิ่มข้อมูล 2, 1, 3, 1, 2, 3 แล้วลบ 2

รหัสที่ 10-16 แสดงคลาส `BSTCollection` ที่เป็นคลาสลูกของ `BSTSet` มีการทำงานเหมือนกันทุกประการ แต่เราต้องการให้เก็บข้อมูลซ้ำได้ จึงต้องสร้างต้นไม้ต้นใหม่ที่มีพฤติกรรมเก็บของซ้ำได้ แทนของเดิม โดยสร้างคลาสลูกของ `BSTree` แล้วเปลี่ยนพฤติกรรมการเพิ่ม `add(r,e)` ให้เพิ่ม `e` ที่ซ้ำไว้ต้นซ้าย จากนั้นสร้างต้นไม้ใหม่นี้แทนต้นเดิมไว้ในตัวสร้างของคลาส

```
public class BSTCollection extends BSTSet implements Collection {
  private static class BSTreeWithDup extends BSTree {
      protected Node add(Node r, Object e) {
        if (r == null) {
          r = new Node(e, null, null);
          ++size;
        } else {
          if (compare(e, r.element) <= 0) {
            r.left = add(r.left, e);
          } else {
            r.right = add(r.right, e);
          }
        }
        return r;
      }
  }
  public BSTCollection() {tree = new BSTreeWithDup();}
}
```

นิยามคลาสใหม่ เปลี่ยนพฤติกรรม ของ add ให้ยอมเพิ่มตัวซ้ำ

ข้อมูลใหม่น้อยกว่าหรือเท่ากับที่ r ก็ให้ไปเพิ่มในลูกต้นซ้าย

เปลี่ยนต้นไม้ที่เป็น BSTree ของคลาสพ่อ มาเป็นต้นไม้ที่ยอมเก็บตัวซ้ำ

รหัสที่ 10-16 การสร้างคอลเล็กชันด้วยต้นไม้ค้นหาแบบทวิภาคที่ปรับให้เก็บตัวซ้ำได้

การสร้างแมปด้วยต้นไม้ค้นหาแบบทวิภาค

แมปคือที่เก็บข้อมูลที่มีลักษณะเป็นคู่อันดับ (key, value) โดย key คือส่วนของข้อมูลที่ใช้เป็นตัวหลักในการค้นหา มีค่าไม่ซ้ำกัน (แต่ value ซ้ำได้) บางคนเรียกลักษณะการจัดเก็บเช่นนี้ว่า แบบ*พจนานุกรม* (dictionary) เทียบ key เป็นเช่นคำศัพท์ และ value เป็นความหมายของคำศัพท์นั้น นิยามข้อกำหนดของแมปด้วยอินเทอร์เฟซ Map ดังแสดงในรหัสที่ 10-17 และคำอธิบายเมท็อดต่าง ๆ ในตารางที่ 10-2

```
public interface Map {
  public int size();
  public boolean isEmpty();
  public boolean containsKey(Object key);
  public Object get(Object key);
  public Object put(Object key, Object value);
  public void remove(Object key);
}
```

รหัสที่ 10-17 อินเทอร์เฟซ Map

ตารางที่ 10-2 หน้าที่ของเมท็อดต่าง ๆ ของอินเทอร์เฟซ Map

เมท็อด	หน้าที่
int size()	คืนจำนวนคู่อันดับในแมป
boolean isEmpty()	ตรวจสอบว่า แมปว่างหรือไม่
containsKey(key)	ตรวจสอบว่า แมปเก็บคู่อันดับที่มี key ที่ให้มาหรือไม่
Object get(key)	คืน value ของคู่อันดับที่มี key ที่ให้มา ถ้าไม่มีคืน null
put(key, value)	เพิ่มคู่อันดับ (key, value) ใส่ในแมป
remove(key)	ลบคู่อันดับที่มี key ที่ให้มา

เพื่อสร้างความเข้าใจในการใช้งานแมป ขอยกตัวอย่างรหัสที่ 10-18 ซึ่งเป็นโปรแกรมการนับคำที่แตกต่างกันในแฟ้มข้อมูลว่า แต่ละคำมีปรากฏกี่ครั้งในแฟ้ม เริ่มด้วยการเปิดแฟ้มในบรรทัดที่ 3 และ 4 ตามด้วยการสร้างแมป m ที่เราออกแบบไว้ให้เก็บคู่อันดับ (คำ, จำนวนครั้งที่ปรากฏ) โดยที่คำเป็นสตริง ส่วนจำนวนครั้งที่ปรากฏเป็นอ็อบเจกต์ของ Integer จากนั้นเข้าวงวนอ่านทีละบรรทัดจนหมดแฟ้ม (บรรทัดที่ 8) อ่านมาหนึ่งบรรทัดก็จัดการแยกเป็นคำ ๆ ด้วย StringTokenizer ที่จะแยกคำโดยอาศัยตัวอักษรคั่นดังที่ปรากฏในตัวแปร delim ประกอบด้วย ช่องว่าง รหัสตั้งระยะ tab เครื่องหมายมหัพภาคและจุลภาค , . วงเล็บเปิดและปิด (บรรทัดที่ 7) ดึงออกมาทีละคำเก็บในตัวแปร token ในบรรทัดที่ 11 (ขอให้อ่านรายละเอียดของ StringTokenizer กันเอง) นำไปค้นด้วย m.get(token) ถ้าได้ null แสดงว่าไม่มีคำนี้ ให้ m.put(token, new Integer(1)) (บรรทัดที่ 14) เพื่อบอกว่า คำนี้พบแล้ว 1 ตัว แต่ถ้า v=m.get(token) ไม่เป็น null แสดงว่า v เก็บจำนวนครั้งที่พบ ก็ให้เพิ่มอีกหนึ่งแล้ว put กลับ เนื่องจากคลาส Integer ไม่มีเมท็อดเพื่อ

เปลี่ยนค่าภายใน จึงต้องใช้ intValue() หยิบจำนวนเต็มที่เก็บใน v ออกมา เพิ่มอีกหนึ่ง แล้วสร้างอ็อบเจกต์ใหม่ของ Integer ใส่กลับเข้าไปในแมป (บรรทัดที่ 16) เมื่ออ่านและประมวลผลจนครบก็ให้แสดงแมป m ออกทางจอภาพ (บรรทัดที่ 19) โดยเรียกเมท็อด toString ของแมป m

```
01 public class LineCount {
02   public static void main(String[] args) throws IOException {
03     FileReader fr = new FileReader("data.txt");
04     BufferedReader br = new BufferedReader(fr);
05     Map m = new BSTMap();
06     String line;
07     String delim = " \t,.()";
08     while ((line = br.readLine()) != null) {
09       StringTokenizer str = new StringTokenizer(line, delim);
10       while (str.hasMoreTokens()) {
11         String token = str.nextToken().toLowerCase();
12         Object v = m.get(token);
13         if (v == null)
14           m.put(token, new Integer(1));
15         else
16           m.put(token, new Integer(((Integer)v).intValue()+1));
17       }
18     }
19     System.out.println(m.toString());
20   }
21 }
```

เปิดแฟ้ม

แมปนี้เก็บ (คำ,จำนวนครั้งที่พบ)

อ่านจากแฟ้มหนึ่งบรรทัด

ดึงมาหนึ่งคำ

หาว่าปรากฏกี่ครั้งแล้ว

ถ้าไม่เคยพบ ก็เก็บว่าพบแล้ว 1

ถ้าเคยพบ ก็เก็บว่าพบเพิ่มขึ้นอีกแล้ว 1

รหัสที่ 10-18 โปรแกรมนับคำที่แตกต่างกันในแฟ้ม

ก็มาถึงขั้นตอนการสร้างแมป แบบง่ายสุดคือการสร้างแมปด้วยรายการ การค้นหาคงต้องไล่เปรียบเทียบกันทั้งรายการ ประสิทธิภาพการทำงานไม่ดี จึงไม่ขออธิบายการสร้างแบบนี้ ขอนำเสนอการสร้างแมปด้วยต้นไม้ค้นหาแบบทวิภาคให้ชื่อว่า BSTMap สองรูปแบบ แบบแรกสร้าง BSTMap ให้เป็นคลาสลูกของ BSTree อีกแบบสร้าง BSTMap ที่ภายในมีต้นไม้ BSTree เป็นส่วนประกอบเพื่อเก็บข้อมูลแทน

เริ่มด้วยแบบแรก (รหัสที่ 10-19) เป็นการสร้าง BSTMap ให้เป็นคลาสลูกของ BSTree วิธีนี้ขยายปมเดิมแบบ Node ของต้นไม้ จากเดิมที่เคยเก็บเฉพาะ element ซึ่งเราให้เป็น key ก็ให้เก็บ value เพิ่มที่ปมด้วย จึงนิยามปมใหม่ให้ชื่อ MNode (บรรทัดที่ 2 ถึง 8) เป็นคลาสลูกของ Node (ซึ่งเราได้นิยามไว้ในคลาสปู่ BinaryTree) แล้วเพิ่มตัวแปร value และตัวสร้างของ MNode ส่งค่า key ไปเก็บใน element และ value เก็บที่ตัวแปรใหม่ที่เพิ่ม ดังนั้นเมท็อดที่เคยสร้างปมแบบ Node ก็ต้องเปลี่ยนให้มาสร้างปมแบบ MNode แทน ด้วยวิธีนี้เราไม่ต้องเขียนเมท็อด size และ isEmpty ใช้ของคลาสพ่อ BSTree ได้เลย เช่นเดียวกับ remove(key) ก็ไม่ต้องเขียน เพราะ key ที่เก็บใน element เป็นตัวหลักในการค้นหาปมเพื่อลบ มาถึงเมท็อดที่ต้องเขียนเพิ่ม เมท็อด

containsKey (บรรทัดที่ 9 ถึง 11) เรียก getNode(root,key) ของ BSTree เพื่อค้นปมที่เก็บ key ถ้าคืนมาไม่เป็น null ก็แสดงว่า มี key นี้อยู่ในแมป (ดูเมท็อด getNode ในรหัสที่ 10-5 หน้าที่ 201) เมท็อด get (บรรทัดที่ 12 ถึง 15) คืน value ของ key ที่ให้มา โดยเรียกใช้ getNode ของ BSTree เพื่อค้นหาปมที่มีค่า key นั้น ถ้าค้นพบ (ได้ปมที่ไม่เป็น null) ก็คืน value ของปมที่พบ ส่วนเมท็อด put(key,value) ขอละไว้ให้ผู้อ่านเขียนเอง

```
01 public class BSTMap extends BSTree implements Map {
02   private static class MNode extends Node {
03     Object value;
04     MNode (Object key, Object value) {
05       super(key, null, null);
06       this.value = value;
07     }
08   }
09   public boolean containsKey(Object key) {
10     return super.getNode(root, key) != null;
11   }
12   public Object get(Object key) {
13     MNode node = (MNode) super.getNode(root, key);
14     return node == null ? null : node.value;
15   }
16   public Object put(Object key, Object value) {
..     ...
```

สร้างปมประเภทใหม่ที่เก็บ value เพิ่มจากของเก่า

key ไปเก็บในตัวแปร element ของคลาส Node

ลองเขียน put เอง

รหัสที่ 10-19 คลาส BSTMap ที่สร้างเป็นคลาสลูกของ BSTree

มาดูการสร้างแมปอีกแบบ (รหัสที่ 10-20) แบบนี้สร้าง BSTMap ที่ภายในมีต้นไม้ BSTree (ให้ชื่อว่า tree ในบรรทัดที่ 11) เป็นตัวเก็บข้อมูลแทน เนื่องจาก BSTree เก็บข้อมูลตัวเดียวต่อปม แต่เราต้องเก็บ key และ value จึงนิยามคลาสใหม่ชื่อ Entry (บรรทัดที่ 2 ถึง 10) ภายในมีตัวแปร key และ value พร้อมตัวสร้าง เนื่องจากอ็อบเจกต์ของ Entry นี้เป็นข้อมูลที่เก็บในต้นไม้ค้นหาแบบทวิภาค จึงต้องเป็นอ็อบเจกต์ประเภทที่เปรียบเทียบได้ นั่นคือต้องให้ Entry implements Comparable และเขียนเมท็อด compareTo (บรรทัดที่ 7 ถึง 9) โดยใช้ compareTo ของ key ในการเปรียบเทียบต่ออีกทีหนึ่ง

เมท็อดส่วนใหญ่เรียกใช้บริการสาธารณะของ tree ต่ออีกทอดหนึ่ง size และ isEmpty ก็เรียก tree.size() และ tree.isEmpty() (บรรทัดที่ 13 และ 16) เมท็อด get(key) เรียกใช้ tree.get แต่ต้องสร้างอ็อบเจกต์ของ Entry ที่มี key แต่ไม่ต้องใส่ value (บรรทัดที่ 19) เพราะเราใช้เฉพาะ key เป็นตัวค้น เมื่อได้ข้อมูลที่ไม่ใช่ null กลับคืน นั่นคือข้อมูลในปมที่มี key เหมือนกับที่ต้องการ จึงสามารถดึง value จากข้อมูลที่ได้กลับไปเป็นผลลัพธ์ (ขอให้กลับไปอ่านคำอธิบายในย่อหน้าสุดท้ายของหน้าที่ 202 อีกครั้ง) remove ก็ทำในลักษณะเดียวกันคือเรียก

tree.remove แล้วส่งอ็อบเจกต์ Entry ที่มี key ที่จะลบทิ้ง เมท็อดcontainsKey ก็ทำในลักษณะเดียวกัน สำหรับเมท็อด put นั้น ขอให้ผู้อ่านลองทำความเข้าใจเอง

```
public class BSTMap implements Map {
  protected static class Entry implements Comparable {
    public Object key, value;
    public Entry(Object k, Object v) {
      key = k; value = v;
    }
    public int compareTo(Object obj) {
      return ((Comparable)key).compareTo(((Entry)obj).key);
    }
  }
  private BSTree tree = new BSTree();
  public int size() {
    return tree.size();
  }
  public boolean isEmpty() {
    return tree.isEmpty();
  }
  public Object get(Object key) {
    Entry e = (Entry) tree.get(new Entry(key, null));
    return e == null ? null : e.value;
  }
  public void remove(Object key) {
    tree.remove(new Entry(key, null));
  }
  public boolean containsKey(Object key) {
    return tree.get(new Entry(key, null)) != null;
  }
  public Object put(Object key, Object value) {
    Entry newEntry = new Entry(key, value);
    Entry e = (Entry) tree.get(newEntry);
    Object oldValue = (e == null ? null : e.value);
    if (e == null) {
      tree.add(newEntry);
    } else {
      e.value = value;
    }
    return oldValue;
  }
}
```

สร้างข้อมูลประเภทใหม่แทนคู่อันดับ key,value เพื่อเก็บที่ปมของต้นไม้

ข้อมูลที่ปมของ BSTree ต้องเปรียบเทียบได้ ในที่นี้ใช้การเปรียบเทียบ key

สร้างอ็อบเจกต์ของ Entry ที่มีแต่ key เพื่อการค้นข้อมูลในปมที่มี key เหมือนกัน

ต้องคืนค่าเก่ากลับคืน ถ้ามี key อยู่แล้ว

ถ้าไม่ key อยู่ ก็เพิ่ม (key, value) ใหม่

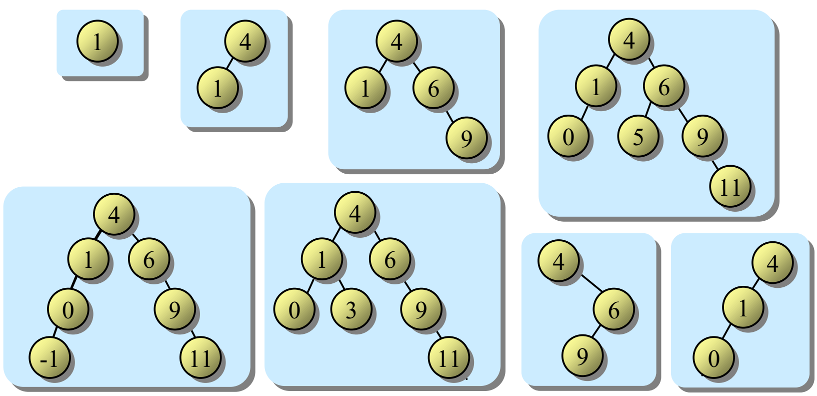
ถ้ามี key อยู่ ก็เพียงแค่เปลี่ยน value

รหัสที่ 10-20 คลาส BSTMap ที่สร้างโดยเก็บต้นไม้ BSTree ไว้ภายใน

ต้นไม้เอวีแอล

จากลักษณะของต้นไม้ค้นหาแบบทวิภาคที่ได้นำเสนอมา ซึ่งมีประสิทธิภาพการทำงานเชิงเวลาของบริการต่าง ๆ ทั้งการค้นหา เพิ่ม และลบ ล้วนเป็น O(h) ทั้งสิ้น ใครจะใช้ต้นไม้ค้นหาแบบทวิภาคก็อาจจะเป็นห่วงว่า การทำงานจะเร็วจะช้าขึ้นกับลักษณะของต้นไม้ ความสูงของต้นไม้ที่มีข้อมูล n ตัวอยู่ในช่วงกว้างมากคือ $\lfloor \log_2 n \rfloor \leq h \leq (n-1)$ แสดงว่า ถ้าโชคร้าย การค้นหา เพิ่ม ลบก็เป็น O(n) โชคดีก็เป็น O(log n) โดยเราได้แสดงให้เห็นว่า หากข้อมูลที่ได้รับมาสร้างต้นไม้มีลักษณะสุ่ม ก็จะได้ประสิทธิภาพเป็น O(log n) ถ้าเรายอมรับกับสภาพเช่นนี้ไม่ได้ ต้องการประกันประสิทธิภาพให้เป็น O(log n) ตลอด ก็ต้องประกันความสูงของต้นไม้ให้ h = O(log n) ตลอดเวลา หัวข้อนี้นำเสนอ*ต้นไม้เอวีแอล* (AVL tree [1]) ซึ่งเป็นต้นไม้ค้นหาแบบทวิภาคที่ประกันความสูงได้ตามที่ต้องการ

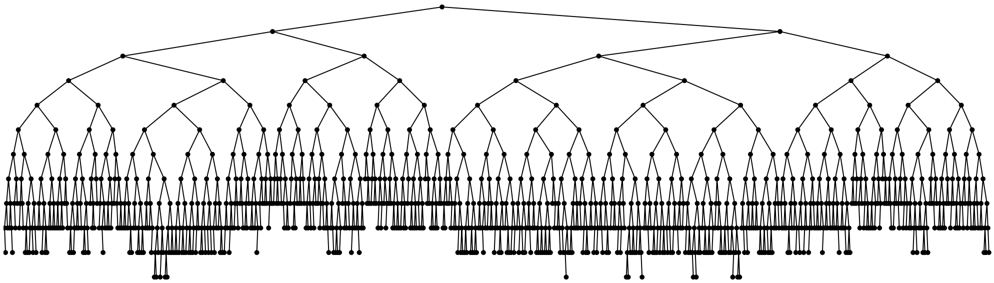
ต้นไม้เอวีแอลคือต้นไม้ค้นหาแบบทวิภาคที่เพิ่มกฎบังคับความสูงของต้นไม้ โดยตั้งกฎไว้ว่า ที่ปม r ใด ๆ ในต้นไม้เอวีแอล ความสูงของลูกต้นซ้าย และความสูงของลูกต้นขวาของ r ต่างกันได้ไม่เกิน 1 และทั้งลูกต้นซ้ายกับลูกต้นขวาต้องเป็นต้นไม้เอวีแอลด้วย รูปที่ 10-16 แสดงตัวอย่างต้นไม้เอวีแอล (สี่ต้นด้านบน) และที่ไม่ใช่เอวีแอล (สี่ต้นด้านล่าง) ขออธิบายเฉพาะกรณีที่ไม่ใช่ต้นไม้เอวีแอลเริ่มที่สองต้นทางขวาล่าง ซึ่งผิดกฎเมื่อพิจารณาที่ปม 4 มีลูกข้างหนึ่งสูง 1 แต่ลูกอีกข้างไม่มี (`null` ถือว่าสูง -1) จึงมีความสูงต่างกันเป็น 2 ซึ่งผิดกฎ มาดูสองต้นซ้ายล่าง พิจารณาที่ปม 4 ไม่ผิดกฎ (ลูกทั้งสองสูงต่างกันไม่เกินหนึ่ง) แต่มีลูกที่ไม่ใช่เอวีแอล (มีโครงสร้างที่ผิดกฎคล้ายต้นทางขวาสุด) ตัวเองจึงไม่ใช่เอวีแอลด้วย



รูปที่ 10-16 ตัวอย่างต้นไม้เอวีแอล (สี่ต้นด้านบน) และไม่ใช่เอวีแอล (สี่ต้นด้านล่าง)

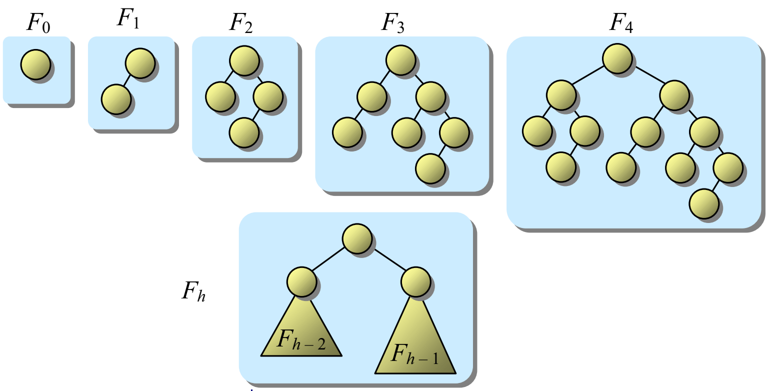
[1] AVL เป็นชื่อย่อของชาวรัสเซียสองคนชื่อ Adelson-Velskii และ Landis ที่ออกแบบต้นไม้ประเภทนี้

เพื่อให้เห็นกันก่อนด้วยตาว่า ต้นไม้เอวีแอลไม่สูง รูปที่ 10-17 แสดงต้นไม้เอวีแอลที่สร้างด้วยข้อมูลสุ่มจำนวน 1,000 ตัว เห็นได้ว่า ต้นไม้มีปมแน่นเกือบทุกระดับ ได้ดุลดี และถึงแม้ว่า จะสร้างต้นไม้เอวีแอลด้วยข้อมูลมีระเบียบเรียงจากน้อยไปมาก ก็จะได้ต้นไม้ที่ได้ดุลดีเช่นกัน ขอบอกตรงนี้ก่อนว่า ต้นไม้เอวีแอลไม่ใช่ต้นไม้ได้ดุลแบบเตี้ยสุด นั่นคือ $h \neq \lfloor \log_2 n \rfloor$ แต่มีความสูง $h = O(\log n)$ และถ้าวิเคราะห์อย่างละเอียดแล้วจะได้ว่า ความสูงของต้นไม้เอวีแอลเป็น $\lfloor \log_2 n \rfloor \leq h \leq 1.44 \log_2 n$ ซึ่งถือได้ว่าเตี้ยมาก ๆ ซึ่งเราจะแสดงให้เห็นจริงต่อไป



รูปที่ 10-17 ตัวอย่างต้นไม้เอวีแอลที่สร้างจากข้อมูลสุ่มจำนวน 1000 ตัว

กำหนดให้ F_h คือต้นไม้เอวีแอลที่สูง h ซึ่งมีจำนวนปมน้อยสุด[2] รูปที่ 10-18 แสดงตัวอย่างของ $F_0, F_1, F_2, F_3,$ และ F_4 ลองพิจารณา F_3 มี 7 ปม ถ้าเราลบปมใดใน 7 ปมนี้ออก จะทำให้ต้นไม้ไม่สูง 3 หรือไม่ก็ทำให้ไม่เป็นเอวีแอล การสร้าง F_h กระทำได้ง่าย ๆ โดยนำ F_{h-1} และ F_{h-2} มาเป็นลูกของรากใหม่ ทำให้ต้นไม้นี้สูง h เป็นไปตามกฎ (เพราะความสูงของลูกทั้งสองต่างกันหนึ่งพอดี) และมีจำนวนปมน้อยสุด (เพราะประกอบด้วยต้นย่อยที่มีจำนวนปมน้อยสุด)



รูปที่ 10-18 ตัวอย่างต้นไม้ฟิโบนักชี

---

2 เราเรียกต้นไม้แบบนี้ว่า *ต้นไม้ฟิโบนักชี* (Fibonacci tree) เนื่องจากจำนวนปมของต้นไม้นี้เขียนบรรยายได้คล้ายกับจำนวนฟิโบนักชี (คงจำกันได้ว่าจำนวนฟิโบนักชีเขียนได้ด้วยความสัมพันธ์เวียนเกิด $f_n = f_{n-1} + f_{n-2}$ เมื่อ $n > 2$ โดยที่ $f_0 = 0, f_1 = 1$)

กำหนดให้ n_h แทนจำนวนปมของต้นไม้ F_h เราเขียนความสัมพันธ์เวียนเกิดของ n_h ได้เป็น

$$n_h = 1 + n_{h-1} + n_{h-2} \text{ สำหรับ } h \geq 2, n_0 = 1, n_1 = 2$$

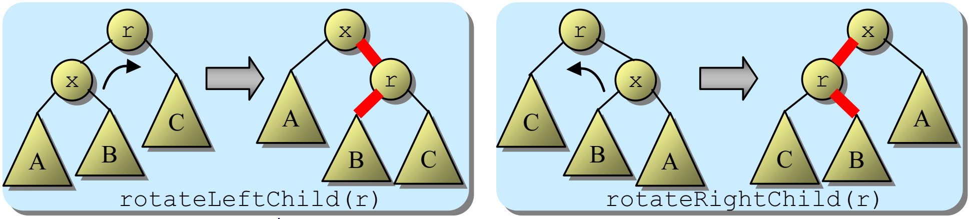
หาผลเฉลยได้ $n_h = c_1\phi^h + c_2\hat{\phi}^h - 1$, $\phi = \frac{1+\sqrt{5}}{2}$, $\hat{\phi} = \frac{1-\sqrt{5}}{2}$, $c_1 = 1+\frac{\sqrt{5}}{2}$, $c_2 = 1-\frac{\sqrt{5}}{2}$ สังเกตได้ว่า ϕ เป็นค่าคงตัวที่เรียกว่า *สัดส่วนทอง* (golden ratio) ตัวเดียวกับผลเฉลยของจำนวนฟีโบนักชีที่คงเคยได้กันเรียนในวิชาวิยุตคณิตศาสตร์ (discrete mathematics) เนื่องจาก $|\hat{\phi}| < 1$ ทำให้พจน์ที่สองของ n_h มีค่าเข้าใกล้ศูนย์เมื่อ n_h มีค่าเพิ่มขึ้น ดังนั้น $n_h + 1 \approx c_1\phi^h$ หาลอการิทึมและจัดการได้ดังนี้

$$\begin{aligned} \log_2(n_h+1) &\approx h\log_2\phi + \log_2 c_1 \\ h &\approx \frac{\log_2 n_h}{\log_2\phi} - \frac{\log_2 c_1}{\log_2\phi} \\ &\approx 1.44\log_2 n_h - 1.328 \end{aligned}$$

เนื่องจากต้นไม้ฟีโบนักชีที่เราใช้วิเคราะห์ข้างต้นนี้เป็นต้นไม้เอวีแอล n ปมที่สูงสุด ดังนั้นค่าของ h ที่ได้จากการวิเคราะห์จึงเป็นความสูงมากสุดของต้นไม้เอวีแอลที่มี n ปม สรุปได้ว่า ด้วยกฎที่เพิ่มขึ้นในต้นไม้เอวีแอลเป็นการประกันว่า ต้นไม้เอวีแอลที่มี n ปม สูงไม่เกิน $1.44 \log_2 n$

การหมุนลูก

ปัญหาที่ตามมาคือจะเพิ่มและลบข้อมูลอย่างไร ที่ยังคงรักษาสภาพของต้นไม้เอวีแอลให้ตรงตามกฎที่ตั้งไว้ และใช้เวลาการทำงานเป็น O(log n) ก่อนอื่นขอนำเสนอการดำเนินการพื้นฐานตัวหนึ่งของต้นไม้ค้นหาแบบทวิภาค ที่ใช้ปรับตัวต้นไม้ เรียกว่า *การหมุนลูก* การหมุนมีสองแบบคือหมุนลูกซ้ายกับหมุนลูกขวา ตรงกับเมท็อดชื่อ `rotateLeftChild(r)` และ `rotateRightChild(r)` รูปที่ 10-19 ซ้ายแสดงการหมุนลูกซ้ายของ `r` ทำให้ลูกซ้ายของ `r` ขึ้นมาเป็นรากแทน `r` โดยต้นไม้ยังคงสภาพเป็นต้นไม้ค้นหา (คือข้อมูลในต้นไม้ย่อยซ้ายมีค่าน้อยกว่าราก และข้อมูลในต้นไม้ย่อยขวามีค่ามากกว่าราก) สิ่งที่ต้องทำคือให้ `x` ที่เคยเป็นลูกซ้ายของ `r` เปลี่ยนมาเป็นพ่อของ `r` แล้วให้ `r` ไปเป็นลูกขวาของ `x` และต้นย่อย B ที่เคยมากกว่า `x` และน้อยกว่า `r` ถูกย้ายไปเป็นลูกซ้ายของ `r` ซึ่งมากกว่า `x` และน้อยกว่า `r` เหมือนเดิม ผลลัพธ์ที่ต้องคืนให้ผู้เรียกการหมุนก็คือรากของต้นไม้หลังการหมุน ซึ่งก็คือปม `x` นั่นเอง ให้สังเกตว่าภาระจริง ๆ ของการหมุนคือการเปลี่ยนตัวโยงซ้ายของ `r` กับตัวโยงขวาของ `x` ให้เป็นไปตามที่แสดงในรูปเท่านั้นเอง จึงใช้เวลาคงตัว สำหรับกรณีการหมุนลูกขวาก็ทำได้ในลักษณะคล้าย ๆ กัน รหัสที่ 10-21 แสดงขั้นตอนการทำงานของการหมุนลูกทั้งสองแบบ (เมท็อดทั้งสองนี้อยู่ในคลาส `BSTree` เพราะต้นไม้ค้นหาแบบทวิภาคหลาย ๆ ชนิดสามารถเรียกใช้ให้เป็นประโยชน์ได้)



รูปที่ 10-19 การหมุนลูกซ้ายกับการหมุนลูกขวา

```
public class BSTree extends BinaryTree {
  ...
  protected Node rotateLeftChild(Node r) {
    Node newRoot = r.left;
    r.left = newRoot.right;
    newRoot.right = r;
    return newRoot;
  }
  protected Node rotateRightChild(Node r) {
    Node newRoot = r.right;
    r.right = newRoot.left;
    newRoot.left = r;
    return newRoot;
  }
```

การหมุนลูกอยู่ใน BSTree เพราะมีต้นไม้ค้นหาแบบทวิภาคหลายชนิดใช้การหมุนลูก

รหัสที่ 10-21 เมท็อดหมุนลูกซ้ายและลูกขวา

โครงสร้างปมของต้นไม้เอวีแอล

เนื่องจากต้นไม้เอวีแอลต้องคอยรักษากฎที่ว่า ลูกต้นซ้ายและลูกต้นขวาห้ามสูงต่างกันเกินหนึ่ง ดังนั้น การรู้ความสูงของต้นไม้ย่อยใด ๆ ในต้นไม้เอวีแอลได้อย่างรวดเร็วจึงจำเป็นอย่างยิ่ง ถึงแม้จะมีเมท็อด `height` ในคลาส `BinaryTree` ให้ใช้ ก็ไม่ควรใช้เพราะช้า ใช้เวลาแปรตามจำนวนปมในต้นไม้ย่อย จึงต้องใช้การจำความสูงตามปมต่าง ๆ แทน แล้วคอยปรับความสูงให้ตรงความจริงเสมอเมื่อต้นไม้เปลี่ยนแปลง รหัสที่ 10-22 แสดงคลาส `AVLTree` ที่เขียนให้เป็นคลาสลูกของ `BSTree` ภายในนิยามคลาสของปม `AVLNode` เป็นคลาสลูกของ `Node` (ซึ่งนิยามในคลาส `BinaryTree` ที่เป็นคลาสปู่) ภายใน `AVLNode` มีตัวแปร `height` เอาไว้เก็บความสูงของต้นไม้ย่อยที่มีปมนี้เป็นราก มีตัวสร้าง มีเมท็อด `setHeight` ให้เรียกเพื่อตั้งความสูงของปม โดยจะขอความสูงของลูกทั้งสองมาคำนวณความสูงของตัวเอง ดังนั้น `setHeight` จะทำงานถูกต้องเมื่อความสูงของลูกทั้งสองถูกต้อง การขอความสูงนี้ใช้เมท็อด `height(Node r)` ที่รับ `r` มาคิด ถ้า `r` เป็น `null` ให้คืน –1 ถ้าไม่ใช่ `null` ให้คืนตัวแปร `height` ของปม `r` นอกจากนี้ยังมีเมท็อด `balanceValue` ซึ่งคืนผลต่างของความสูงของลูกขวาและลูกซ้าย ถ้าได้ผลเป็น –1, 0, หรือ 1 แสดงว่า ต้นไม้ที่มีปมนี้เป็นรากไม่ผิดกฎของเอวีแอล แต่ถ้าได้ค่าอื่นแสดงว่าผิดกฎ ต้องรีบปรับปรุงที่จะได้อธิบายกันต่อไป

```
public class AVLTree extends BSTree {
  private static class AVLNode extends Node {
    private int height;
    AVLNode (Object e, Node l, Node r) {
      super(e, l, r);
    }
    void setHeight() {
      height = 1 + Math.max(height(left), height(right));
    }
    int balanceValue() {
      return height(right) - height(left);
    }
    private static int height(Node n) {
      return (n == null ? -1 : ((AVLNode) n).height);
    }
  }
```

สร้างปมประเภทใหม่ที่ขยายจากปมแบบ Node ของ BinaryTree ให้มี height ด้วย

-1, 0, +1 คือถูกกฎ
-2 คือเอียงซ้ายผิดปกติ
+2 คือเอียงขวาผิดปกติ

รหัสที่ 10-22 AVLTree คือคลาสของต้นไม้เอวีแอลมีปมที่เก็บความสูงของต้นไม้ย่อยที่ปมนี้เป็นราก

การเพิ่มและลบข้อมูล

การเพิ่มและลบข้อมูลของต้นไม้เอวีแอลมีหลักการทำงานเหมือนของต้นไม้ค้นหาแบบทวิภาคทุกประการ แต่หลังเพิ่มหรือลบเสร็จ จะต้องตรวจสอบและปรับปรุงต้นไม้ให้ถูกกฎ ดังแสดงในรหัสที่ 10-23 เมท็อด remove เรียก super.remove เพื่อให้ลบแบบเดียวกับของ BSTree ตามด้วยการปรับต้นไม้ในถูกกฎด้วยเมท็อด rebalance (ซึ่งจะอธิบายในหัวข้อถัดไป) แล้วคืน r ซึ่งเป็นรากของต้นไม้หลังลบและปรับ ส่วนเมท็อด add ก็ทำงานในลักษณะเดียวกัน ต่างกันเล็กน้อยตรงที่ต้องจัดการกรณีเพิ่มข้อมูลในต้นไม้ว่างเอง คือถ้า r เป็น null จะสร้างปมที่เป็นใบใหม่คืนกลับไปเป็นรากของต้นไม้หลังเพิ่ม ความจริงการทำงานแบบนี้ก็มีใน super.add ซึ่งคือ add ที่เขียนใน BSTree อยู่แล้ว แต่ที่ต้องทำที่นี่เองเพราะการสร้างปมใน BSTree เป็นการสร้างปมแบบ Node ที่ไม่มีความสูงกำกับปม เราต้องสร้างปมแบบ AVLNode ที่มี height อยู่ภายในด้วย จึงต้องสร้างเองที่นี่ อนึ่ง AVLTree ไม่ต้องเขียนเมท็อดสาธารณะ add(Object e) และ remove(Object e) (รวมทั้งเมท็อด get(Object e)) เพราะใช้ของที่เขียนในคลาสพ่อ BSTree ได้

การปรับต้นไม้ให้ถูกกฎ

หลังการเพิ่มหรือลบข้อมูลที่ต้นไม้ย่อยใดเสร็จแล้ว สิ่งที่ได้คืนมาคือรากใหม่ของต้นไม้หลังการเพิ่มหรือลบ เราต้องตรวจสอบต้นไม้ย่อยที่รากนั้นว่า ถูกต้องตามกฎของเอวีแอลหรือไม่ ถ้าผิดกฎ ผิดแบบใด จะได้จัดการปรับปรุงให้ถูก โดยอาศัยการหมุนเป็นเครื่องมือหลักในการปรับ

```
  protected Node add(Node r, Object e) {
    if (r == null) {
      r = new AVLNode(e, null, null);
      ++size;
    } else {
      r = super.add(r,e);
      r = rebalance(r);
    }
    return r;
  }
  protected Node remove(Node r, Object e) {
    r = super.remove(r,e);
    r = rebalance(r);
    return r;
  }
```

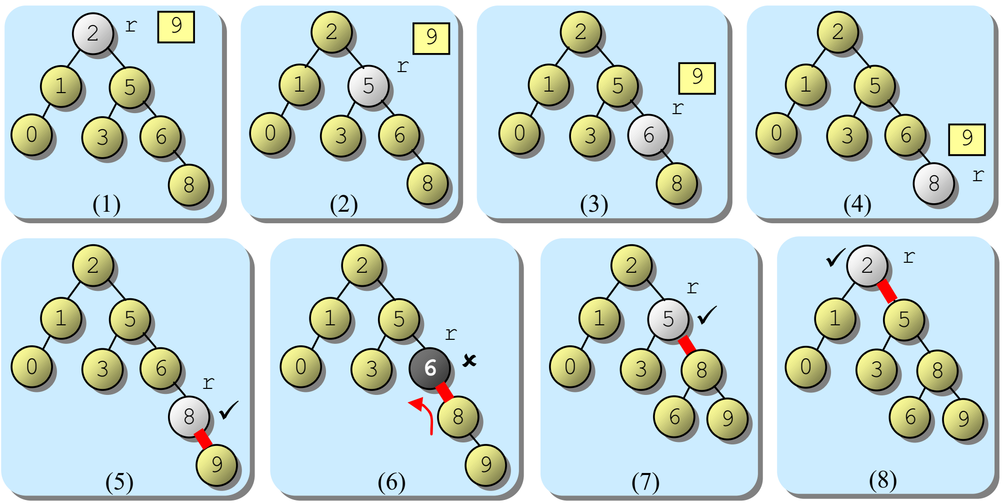
ต้องสร้างปมแบบ AVLNode เลยต้องเขียนเองที่นี่ ใช้ของ BSTree ไม่ได้

เพิ่มแบบเดียวกับของ BSTree แล้วค่อยปรับต้นไม้หลังเพิ่ม

ลบแบบเดียวกับของ BSTree แล้วค่อยปรับต้นไม้หลังลบ

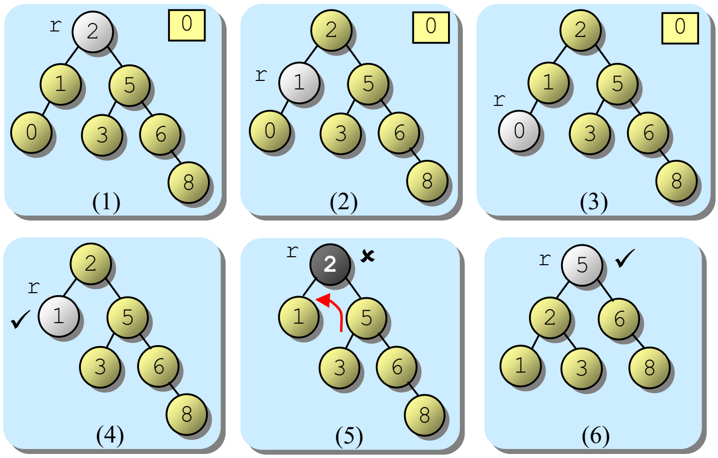
รหัสที่ 10-23 เมท็อด `add` และ `remove` ซึ่งเพิ่มและลบปม แล้วปรับต้นไม้ให้ถูกกฎ

เพื่อให้เข้าใจหลักการคร่าว ๆ ก่อน พิจารณารูปที่ 10-20 (1) ต้องการเพิ่ม 9 ในต้นไม้ที่รากบนสุด มีกระบวนการเพิ่มเช่นเดียวกับของ `BSTree` รูป (1) ถึง (4) แสดงการเลื่อน `r` ลงมาทางขวาเรื่อย ๆ เพราะ 9 มีค่ามากกว่าตลอดวิถีจากราก จนท้ายสุด `r` เป็น `null` จึงสร้างใบใหม่เติมเป็นลูกขวาของ 8 ได้รูป (5) จากนั้นถอย `r` ขึ้นตามวิถีเดิมที่ลงมา ทุกครั้งที่ถอยจะต้องตรวจสอบว่าผิดกฎหรือไม่ เช่น ถอยถึงรูป (6) พบว่า ซ้ายของ `r` สูง –1 ขวาของ `r` สูง 1 ผิดกฎ ก็หมุนลูกขวา 8 ขึ้นมาเป็นรากแทน 6 ทำให้ถูกกฎ ถอยกลับขึ้นไปได้รูป (7) ถูกกฎ ถอยขึ้นอีกได้รูป (8) ถูกกฎ กลับมาที่รากบนสุด เป็นอันสิ้นสุดการเพิ่ม (ขอบอกว่า เราเขียนเมท็อด `add` โดยเรียกตัวเองแบบเวียนเกิด ก็คือการเปลี่ยน `r` จากรากลงไป และเมื่อ `add` คืนการทำงาน ก็คือการให้ `r` ถอยกลับไปยังปมก่อนหน้าอย่างอัตโนมัติ)



รูปที่ 10-20 ตัวอย่างการปรับต้นไม้หลังการเพิ่ม

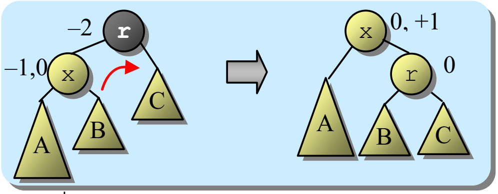
รูปที่ 10-21 แสดงตัวอย่างการปรับต้นไม้หลังการลบ รูป (1) ต้องการลบ 0 ออกจากต้นไม้ การลบต้องค้นข้อมูล ทำให้เกิดการเลื่อนค่าของ r ลงซ้ายลงเรื่อย ๆ เพราะ 0 น้อยกว่าข้อมูลตามปม จนถึงรูป (3) พบปมที่เก็บ 0 ก็ลบด้วยวิธีปกติ ลบเสร็จคืนการทำงานถอยกลับขึ้นไปตามวิถีที่ได้ลงมา พร้อมกับตรวจสอบว่า ผิดกฎหรือไม่ ถ้าผิดก็ปรับ รูป (4) ไม่ผิดกฎ แต่พอถอยขึ้นมาถึงรูป (5) พบว่า ปม 2 ผิดกฎ (ลูกซ้ายสูง 0 แต่ลูกขวาสูง 2) เกิดการหมุนลูกขวา ได้รูป (6) ซึ่งถูกกฎ



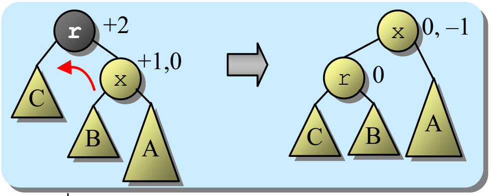
รูปที่ 10-21 ตัวอย่างการปรับต้นไม้หลังการลบ

แล้วจะตรวจสอบอย่างไร ? จะปรับอย่างไร ? จะหมุนอย่างไร ? ลักษณะการปรับต้นไม้มีอยู่ 4 กรณีดังรูปที่ 10-22 ในรูปแสดงจำนวนกำกับข้าง ๆ ปมที่ได้มาจาก `balanceValue` ของปมนั้น ซึ่งคือความสูงของลูกขวาลบด้วยความสูงของลูกซ้าย ถ้า `r.balanceValue() = -1` เราเรียก `r` ว่าเอียงซ้าย แต่ถ้าเป็น `-2` เรียกว่า เอียงซ้ายผิดปกติ และถ้า `r.balanceValue() = +1` เราเรียก `r` ว่า เอียงขวา แต่ถ้าเป็น `+2` เรียกว่า เอียงขวาผิดปกติ การตรวจสอบและการปรับของทั้งสี่กรณีมีดังนี้

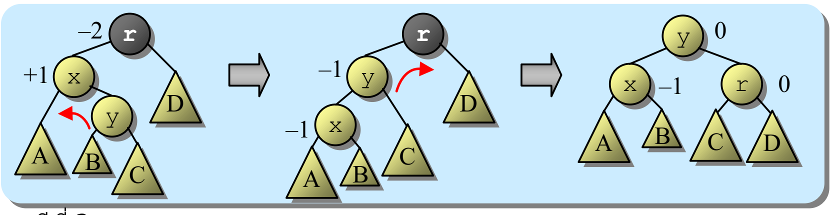
1. ถ้า `r` เอียงซ้ายผิดปกติ และลูกซ้ายไม่เอียงขวา ให้หมุนลูกซ้ายของ `r` ขึ้นเป็นรากด้วย
   `r = rotateLeftChild(r);`
2. ถ้า `r` เอียงขวาผิดปกติ และลูกขวาไม่เอียงซ้าย ให้หมุนลูกขวาของ `r` ขึ้นเป็นรากด้วย
   `r = rotateRightChild(r);`
3. ถ้า `r` เอียงซ้ายผิดปกติ และลูกซ้ายเอียงขวา ให้หมุนลูกขวาของลูกซ้ายของ `r` ขึ้นเป็นรากด้วย
   `r.left = rotateRightChild(r.left); r = rotateLeftChild(r);`
4. ถ้า `r` เอียงขวาผิดปกติ และลูกขวาเอียงซ้าย ให้หมุนลูกซ้ายของลูกขวาของ `r` ขึ้นเป็นรากด้วย
   `r.right = rotateLeftChild(r.right); r = rotateRightChild(r)`



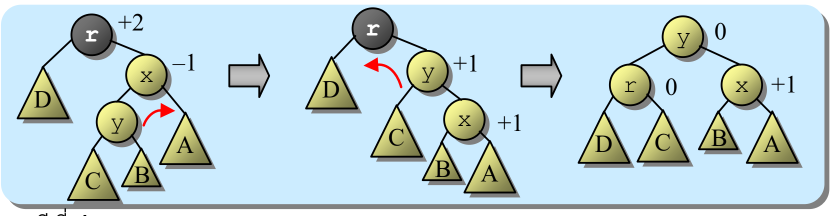
กรณีที่ 1 : `rotateLeftChild(r)`



กรณีที่ 2 : `rotateRightChild(r)`



กรณีที่ 3 : `rotateRightChild(r.left); rotateLeftChild(r)`

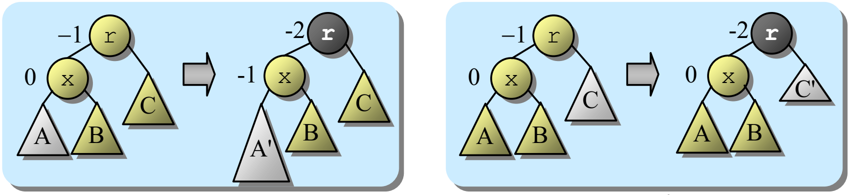


กรณีที่ 4 : `rotateLeftChild(r.right); rotateRightChild(r)`

รูปที่ 10-22 การปรับต้นไม้ทั้งสี่กรณี

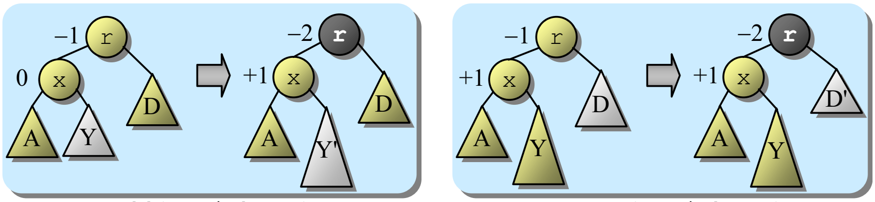
ให้สังเกตว่า ต้นไม้หลังการหมุนในทั้งสี่กรณี จะเตี้ยลงจากตอนที่มีปัญหา และยังขจัดปัญหาของ `balanceValue` ที่รากด้วย ขออธิบายรูปที่ 10-22 เพิ่มเติมเล็กน้อย ค่า `balanceValue` ของบางปมในรูปเขียนไว้สองค่า (ในกรณีที่ 1 และ 2) นั่นหมายความว่า เป็นค่าใดก็ได้ ใช้การปรับเหมือนกัน ได้ `balanceValue` ต่างไปบ้าง แต่ก็ยังถูกกฎ ในกรณีที่ 3 และ 4 ต้น B และ C ในรูปอาจมีความสูงสลับกันได้ ซึ่งจะให้ผลของ `balanceValue` ต่างไปบ้าง แต่ก็ยังถูกกฎเช่นกัน

ต้องลองคิดย้อนไปด้วยว่า เราไปทำอะไรกับต้นไม้ ถึงจะเกิดกรณีผิดกฎดังกล่าว รูปที่ 10-23 แสดงตัวอย่างการเพิ่มและลบข้อมูลที่นำไปสู่กรณีที่ 1 ราก `r` เดิมมีลูกซ้ายสูงกว่าลูกขวาอยู่ 1 การทำให้ `r` มีลูกซ้ายสูงกว่าลูกขวา 2 ก็เกิดมาจากการที่เราไปเพิ่มข้อมูลในลูกซ้าย แล้วทำให้ลูกซ้ายสูงขึ้น (รูปที่ 10-23 ซ้าย) หรือไม่ก็เราไปลบปมในลูกขวาจนทำให้มันเตี้ยลง (รูปที่ 10-23 ขวา) สำหรับตัวอย่างการเพิ่มและลบข้อมูลที่นำไปสู่การผิดกฎกรณี 3 ขอให้ผู้อ่านศึกษาจากรูปที่ 10-24



add(x.left,e) remove(r.right,e)

รูปที่ 10-23 ตัวอย่างการเพิ่มและลบข้อมูลที่ทำให้เกิดการผิดกฎกรณีที่ 1



add(x.right,e) remove(r.right,e)

รูปที่ 10-24 ตัวอย่างการเพิ่มและลบข้อมูลที่ทำให้เกิดการผิดกฎกรณีที่ 3

ก็มาถึงภาระสุดท้ายคือเขียนแนวคิดการปรับต้นไม้ให้ถูกกฎตามที่เสนอมา ให้เป็นเมท็อด `rebalance(r)` ดังแสดงในรหัสที่ 10-24 ถ้า `r` เป็น `null` ก็ไม่ต้องปรับ ถ้าไม่เป็น ให้ดึงค่า `balanceValue` มา ถ้าเท่ากับ –2 (บรรทัดที่ 35) แสดงว่า เป็นกรณีที่ 1 หรือ 3 ถ้ากลับไปดูรูปที่ 10-22 จะเห็นได้ว่า ทั้งสองกรณีนี้ต้องทำ `rotateLeftChild(r)` (บรรทัดที่ 38) โดยถ้าเป็นกรณีที่ 3 ก็ต้องทำ `rotateRightChild(r.left)` ก่อน โดยจะเป็นกรณีที่ 3 ก็เมื่อ `balanceValue` ของลูกซ้ายของ `r` มีค่าเป็น 1 (บรรทัดที่ 36) สำหรับกรณีที่ 2 และ 4 ก็ทำงานในทำนองคล้ายกัน หลังจากนั้นต้องอย่าลืมเรียก `setHeight` เพื่อตั้งความสูงให้กับปม `r` ให้ถูกต้อง ก่อนคืน `r` กลับไปเป็นรากของต้นไม้หลังการปรับ

```
32   private Node rebalance(Node r) {
33     if (r == null) return r;
34     int balance = ((AVLNode)r).balanceValue();
35     if (balance == -2) {
36       if (((AVLNode)r.left).balanceValue() == 1)
37         r.left = rotateRightChild(r.left);      กรณีที่ 3
38       r = rotateLeftChild(r);      กรณีที่ 1 และ 3
39     } else if (balance == 2) {
40       if (((AVLNode)r.right).balanceValue() == -1)
41         r.right = rotateLeftChild(r.right);      กรณีที่ 4
42       r = rotateRightChild(r);      กรณีที่ 2 และ 4
43     }
44     ((AVLNode)r).setHeight();
45     return r;
46   }
```

รหัสที่ 10-24 เมท็อด `rebalance` เพื่อตรวจสอบและปรับต้นไม้ในถูกกฎ

อ้อเกือบลืม เมท็อดการหมุนต่าง ๆ ที่เราเรียกใช้นั้นเป็นบริการของ BSTree ซึ่งไม่รู้เรื่องตัวแปร height กำกับปมแบบ AVLNode ที่เราใช้ในต้นไม้เอวีแอล ดังนั้นจึงต้องเขียนเมท็อดการหมุนใหม่ให้ไปเรียกการหมุนแบบเดิมของคลาสพ่อ แล้วจึงตามด้วยการตั้งค่าความสูงของปมที่เปลี่ยนแปลงด้วย ดังแสดงในรหัสที่ 10-25 ให้สังเกตว่า เราต้องตั้งค่าความสูงให้กับปมลูกก่อนตั้งให้ตัวเอง เช่น ในบรรทัดที่ 49 ลูกทางขวาของ r ถูกตั้งความสูงก่อนตั้งของ r ในบรรทัดถัดไป

```
47   protected Node rotateLeftChild(Node r) {
48     r = super.rotateLeftChild(r);
49     ((AVLNode) r.right).setHeight();
50     ((AVLNode) r).setHeight();
51     return r;
52   }
53   protected Node rotateRightChild(Node r) {
54     r = super.rotateRightChild(r);
55     ((AVLNode) r.left).setHeight();
56     ((AVLNode) r).setHeight();
57     return r;
58   }
59 }
```

หมุนแล้วต้องตั้งความสูงของรากเก่าและรากใหม่ด้วย

รหัสที่ 10-25 การปรับค่าความสูงกำกับปมหลังการหมุน

เราเลือกสร้าง AVLTree ให้เป็นคลาสลูกของ BSTree ทำให้เราเขียน AVLTree ได้สั้น ได้ใช้หลาย ๆ เมท็อดของคลาสพ่อและคลาสปู่ให้เป็นประโยชน์ได้ แต่ก็ต้องอย่าลืมว่า สิ่งที่ทำให้ AVLTree ทำงานได้อยู่ที่ตัวแปร height ตามปมต่าง ๆ ที่ต้องมีค่าที่ถูกต้องตลอดเวลา หากเราเพิ่มเมท็อดที่มีการเปลี่ยนแปลงต้นไม้ใน BSTree ก็ต้องอย่าลืม override เมท็อดเหล่านั้นให้ตั้งค่าความสูงกำกับปมให้ถูกต้องด้วย

อีกเรื่องหนึ่งที่ผู้อ่านอาจสงสัยว่า ภายในคลาส AVLTree ที่เขียนกันมามีการ cast จาก Node เป็น AVLNode เช่น บรรทัดที่ 49, 50, 55, 56 ของรหัสที่ 10-25 ข้างบนนี้ หลายคนอาจสงสัยว่า ทำไมไม่นิยามปมที่รับเป็นพารามิเตอร์ต่าง ๆ ใน AVLTree ให้เป็น AVLNode ให้หมด จะได้ไม่ต้องทำการเปลี่ยนประเภทข้อมูล เราทำเช่นนั้นไม่ได้ เพราะ AVLTree พึ่งพาเมท็อดต่าง ๆ ของ BSTree และตัวแปรในคลาส Node โดยเมท็อดใน BSTree ใช้ปมแบบ Node ทั้งสิ้น จึงจำเป็นต้องเขียนหัวเมท็อดที่รับและคืนปมแบบ Node แล้วค่อย cast เป็น AVLNode เมื่อยามต้องการ

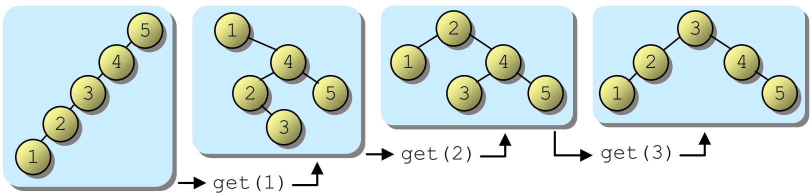
ต้นไม้ค้นหาแบบอื่น ๆ

ในหัวข้อนี้ ขอนำเสนอต้นไม้ค้นหาแบบอื่น ๆ ที่มีลักษณะการจัดเก็บและจัดการที่แปลกออกไป เพื่อให้ผู้อ่านได้เห็นแนวทางการออกแบบโครงสร้างข้อมูลที่มีหลากหลายแบบ (จะขอนำเสนอเพียงแนวคิด ไม่ได้ลงรายละเอียดของตัวโปรแกรม) เริ่มจากต้นไม้บานอาศัยแนวคิดการปรับตัวต้นไม้ในแทบทุกการดำเนินการทำให้ได้ประสิทธิภาพโดยรวมที่ดี, ต้นไม้ 2-3-4 อนุญาตให้หนึ่งปมเก็บข้อมูลได้ 1, 2,

หรือ 3 ตัว ทำให้เกิดความยืดหยุ่นในการปรับต้นไม้ให้ได้ดุลเสมอ, ต้นไม้แดงดำมีไว้สร้างต้นไม้ได้ดุล 2-3-4 ที่มีประสิทธิภาพทั้งการจัดเก็บและการจัดการข้อมูล, ต้นไม้ทรีปซึ่งผนวกแนวคิดของฮีป และการทำงานเชิงสุ่ม และปิดท้ายด้วยรายการก้าวกระโดดซึ่งมองได้ว่า เป็นรายการแบบพิเศษ หรือจะมองเป็นต้นไม้ค้นหาแบบหลายทางที่อาศัยพฤติกรรมสุ่ม

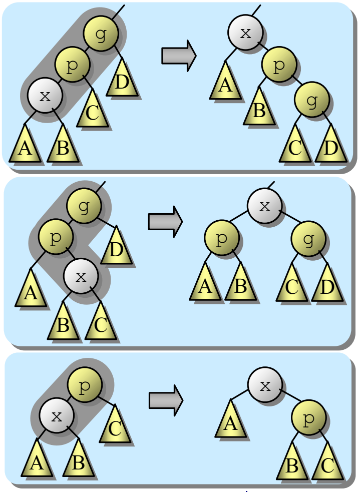
ต้นไม้บาน

ต้นไม้บาน (splay tree) เป็นต้นไม้ค้นหาแบบทวิภาคชนิดหนึ่งที่อาศัยการหมุนปรับตัวเอง ในขณะที่ต้นไม้เอวีแอลจะปรับต้นไม้ก็เมื่อตรวจพบปัญหาหลังการเพิ่มและลบ แต่ต้นไม้บานไม่มีกฎใด ๆ ให้ต้องตรวจสอบ แต่ปรับต้นไม้ทุกครั้งเมื่อเพิ่ม ลบ รวมทั้งเมื่อค้นหาข้อมูลด้วย โดยจะหมุนให้ปมท้ายสุดที่พบของการดำเนินการขึ้นไปเป็นรากของต้นไม้ รูปที่ 10-25 แสดงตัวอย่างการปรับต้นไม้เมื่อมีการค้นหา รูปซ้ายสุดมาจากการเพิ่มข้อมูล 1,2,3,4,5 (ถ้าเพิ่มตามลำดับนี้ในต้นไม้แบบทวิภาคจะได้ต้นไม้เอียงไปทางขวา แต่พอเป็นต้นไม้บานจะเอียงมาทางซ้าย) จากนั้นเรียก `get(1)` ได้รูปถัดมา ให้สังเกตว่า ต้นไม้ถูกปรับจน 1 กลายเป็นราก ต่อมา `get(2)` ได้ 2 ขึ้นเป็นราก และสุดท้าย `get(3)` ก็ได้ 3 เป็นราก นอกจากรากเปลี่ยนแล้ว ต้นไม้ยังอาจเตี้ยลงอีกด้วย



รูปที่ 10-25 ตัวอย่างการค้นหาแล้วเกิดการปรับต้นไม้

การปรับต้นไม้นี้อาศัยการหมุน หลักการคือหมุนปมท้ายสุดในการดำเนินการให้ขึ้นไปจนเป็นราก การหมุนมี 6 รูปแบบ ขึ้นกับว่า ปมที่ถูกหมุนขึ้นเป็นรากมีความสัมพันธ์กับปมพ่อปมปู่อย่างไร รูปที่ 10-26 แสดงการหมุนสามรูปแบบ (อีกสามรูปแบบคล้ายกัน เพียงแต่เปลี่ยนจากขวาเป็นซ้าย จากซ้ายเป็นขวา) ให้ x คือปมที่ต้องการหมุนขึ้นไปเป็นราก สองรูปแบบบนต่างกันตรงที่กิ่งระหว่าง x กับพ่อและพ่อกับปู่ เป็นในทิศเดียวกันหรือต่างกัน แบบที่สองเหมือนกับการหมุนสองครั้งของต้นไม้เอวีแอล ในขณะที่แบบที่หนึ่งคือการหมุนลูกซ้ายของรากสองครั้ง ก็ได้หลาน x ขึ้นมา ต้องระวังอย่าใช้วิธีหมุนลูกซ้ายของ `p` ตามด้วยหมุนลูกซ้ายของ `g` ซึ่งได้ x ขึ้นมาเหมือนกัน แต่ได้รูปร่างต่างกัน และประสิทธิภาพโดยรวมที่ด้อยกว่า (ขอไม่พิสูจน์ ดูตัวอย่างเปรียบเทียบในรูปที่ 10-27 ทางซ้ายใช้การหมุนที่ถูกต้อง ในขณะที่รูปขวาหมุนกรณีที่หนึ่งผิด) รูปที่ 10-28 แสดงการหมุนปม 3 จนเป็นรากเมื่อค้นหาปม 3 ถ้าเราเขียนการค้นหาแบบเวียนเกิด การกำหนดวิธีการปรับจะกระทำจากบนลงล่าง เมื่อพบปมแล้วก็จะเริ่มหมุนขึ้นตามรูปแบบที่พบจนได้ดังรูปที่ 10-28 (6)

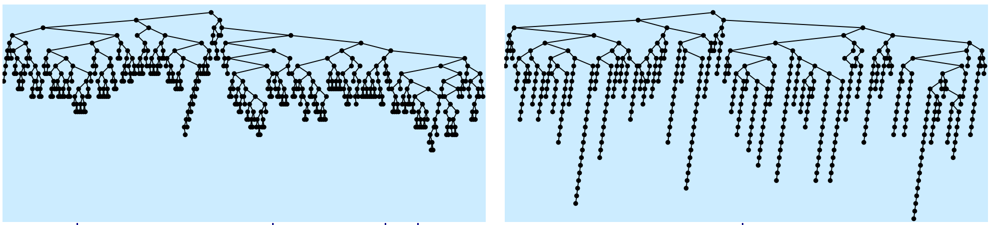


```
r = rotateLeftChild(g);
r = rotateLeftChild(r);
return r;
```

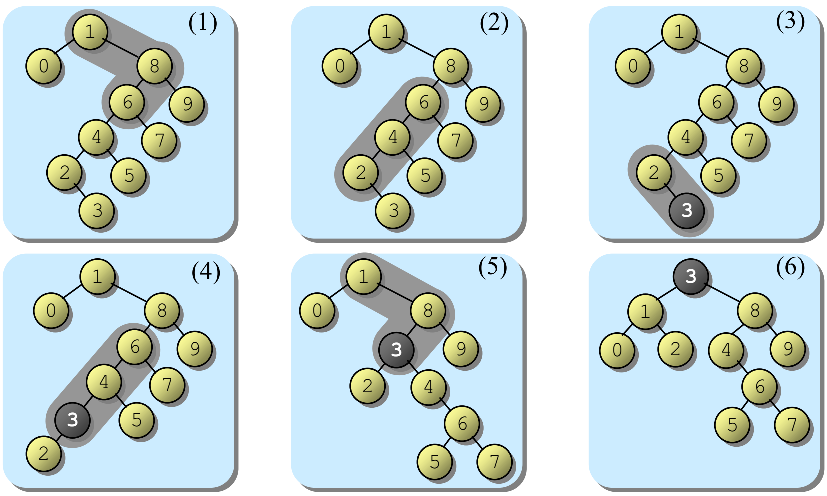
```
g.left = rotateRightChild(g.left);
r = rotateLeftChild(g);
return r;
```

```
r = rotateLeftChild(p);
return r;
```

รูปที่ 10-26 สามในหกรูปแบบการหมุนของต้นไม้บาน



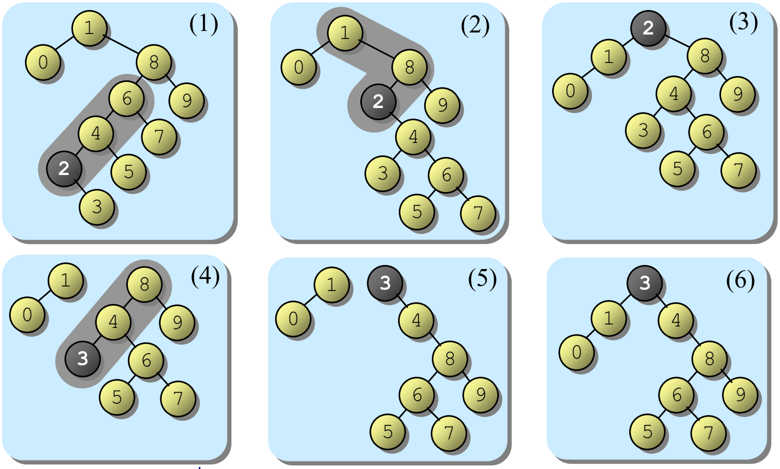
รูปที่ 10-27 ต้นไม้บานที่หมุนกรณีที่หนึ่งถูกต้อง (ซ้าย) เปรียบเทียบกับที่หมุนไม่ถูกต้อง (ขวา)



รูปที่ 10-28 รูป (1)-(3) ค้นหา 3 พร้อมกำหนดรูปแบบการปรับ ตามด้วยรูป (4)-(6) ปรับหลังค้นพบ

การเพิ่มข้อมูลใหม่ในต้นไม้บาน ทำเหมือนกับต้นไม้ค้นหาแบบทวิภาค ตามด้วยการหมุนปม ใหม่ขึ้นเป็นรากตามรูปแบบที่นำเสนอมา สำหรับการลบข้อมูลจะซับซ้อนบ้าง เมื่อต้องการลบ x ก็ต้อง

หา x ให้พบ หมุนปม x ขึ้นเป็นราก แล้วลบ x ทิ้ง จะได้ต้นไม้ย่อยซ้ายและขวาให้ชื่อว่า T_L และ T_R ไปหาตัวน้อยสุดของ T_R หมุนปมน้อยสุดนี้ขึ้นเป็นราก แล้วนำรากของ T_L ไปต่อเป็นลูกทางซ้ายของรากใหม่ (ซึ่งคือปมที่มีค่าน้อยสุด) ของต้นขวา รูปที่ 10-29 แสดงตัวอย่างการลบ 2 เริ่มค้น 2 พอพบในรูป (1) ก็หมุน 2 ขึ้นได้รูป (2) และ (3) ลบ 2 ทิ้งได้รูป (4), ค้นตัวน้อยสุดในต้นขวา ได้ค่า 3, หมุน 3 ขึ้นได้รูป (5), แล้วนำรากของต้นซ้ายมาต่อเป็นลูกซ้ายของ 3 ได้รูป (6)



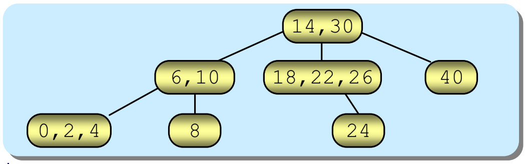
รูปที่ 10-29 ตัวอย่างการลบปม 2 ในต้นไม้บาน

แล้วต้นไม้บานมีดีตรงไหน ? ต้นไม้บานไม่ได้ประกันว่า จะมีความสูงเป็น O(log n) เราสามารถสร้างต้นไม้บานที่สูง n–1 ด้วยซ้ำ (เริ่มด้วยต้นไม้ว่าง แล้วเพิ่ม 1,2,3,4,5 ก็จะได้ต้นไม้เอียงซ้ายสูง 4 มี 5 เป็นราก) จะเห็นข้อดีของต้นไม้บานได้ต้องเรียกบริการของต้นไม้หลาย ๆ ครั้ง (ซึ่งก็ต้องเป็นเช่นนั้นอยู่แล้ว คงไม่มีใครสร้างที่เก็บข้อมูลเพื่อเก็บข้อมูลสองสามตัว ค้นหาสักครั้งสองครั้งแล้วเลิก) เราสามารถวิเคราะห์ให้เห็นจริงได้ว่า การเรียกใช้บริการเพิ่ม ลบ และค้นหาข้อมูลกับต้นไม้บานที่เก็บข้อมูล n ตัวเป็นจำนวน m ครั้งจะใช้เวลาโดยรวมเป็น O(m log n) บางการดำเนินการอาจช้าในบางขณะ แต่บางการดำเนินการจะเร็วได้ในบางขณะเช่นกัน รวมแล้วก็เหมือนกับการใช้ต้นไม้เอวีแอลที่ประกันว่า เพิ่ม ลบ และค้นหาเป็น O(log n) ทำ m ครั้งก็ใช้เวลารวมเป็น O(m log n)

นอกจากนี้ด้วยการหมุนข้อมูลตัวที่ถูกค้นหรือถูกเพิ่มตัวล่าสุดขึ้นมาเป็นราก จะตรงกับพฤติกรรมการใช้ข้อมูลในหลาย ๆ งานที่ว่า ข้อมูลแต่ละตัวที่เก็บไว้ มีความถี่ในการเรียกใช้แตกต่างกัน ตัวที่ถูกเรียกใช้บ่อย ๆ ก็มักถูกเรียกใช้ในอนาคตบ่อยด้วย การหมุนขึ้นมาทำให้การค้นหาในอนาคตรวดเร็วขึ้นกว่าปล่อยทิ้งไว้ให้อยู่ลึก ๆ ดังนั้นการกล่าวหาว่า ต้นไม้ค้นหาที่สูงทำให้ทำงานช้านั้น อาจไม่จริง ถ้าเรามีต้นไม้สูงที่เก็บข้อมูลที่มีความถี่ของการถูกเรียกใช้สูงอยู่บน ๆ แล้วตัวที่ไม่ค่อยถูกเรียกใช้อยู่ลึก ๆ ก็อาจจะดีกว่าต้นไม้ได้ดุลด้วยซ้ำไป

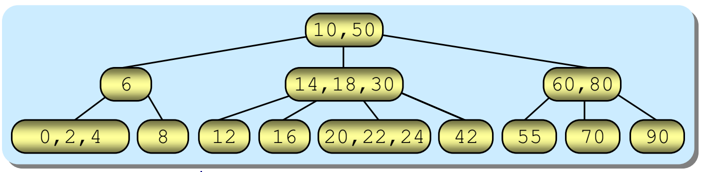
ต้นไม้ได้ดุล 2-3-4

ต้นไม้ค้นหาแบบทวิภาคเก็บข้อมูลหนึ่งตัวต่อหนึ่งปม ก็เลยมีลูกได้สองต้นสำหรับเก็บข้อมูลกลุ่มที่น้อยกว่า และกลุ่มที่มากกว่า ถ้าเราให้เก็บปมละสักสองตัว x_1 และ x_2 โดยที่ $x_1 < x_2$ ก็ย่อมมีลูกได้สามต้น คือต้นที่เก็บข้อมูลที่น้อยกว่า x_1 ต้นที่เก็บข้อมูลที่มากกว่า x_1 แต่น้อยกว่า x_2 และอีกต้นที่เก็บข้อมูลที่มากกว่า x_2 นิยามปมแบบ k คือปมที่เก็บข้อมูลได้ $k-1$ ตัว มีลูกได้ k ต้น รูปที่ 10-30 แสดงตัวอย่างต้นไม้ที่มีปมแบบ 2 แบบ 3 และแบบ 4



รูปที่ 10-30 ตัวอย่างต้นไม้ทีมีปมแบบ 2 แบบ 3 และ แบบ 4

ต้นไม้ 2-3-4 คือต้นไม้ค้นหาที่มีปมได้ทั้งแบบ 2 แบบ 3 และแบบ 4 ต้นไม้ได้ดุล 2-3-4 คือต้นไม้ 2-3-4 ที่มีใบทุกใบอยู่ในระดับเดียวกันหมด ดังตัวอย่างในรูปที่ 10-31 ด้วยการที่มีปมได้สามแบบ ทำให้ง่ายต่อการปรับต้นไม้ให้ได้ดุลเสมอ และได้ต้นไม้ที่สูงเป็น O(log n) ตลอดเวลา

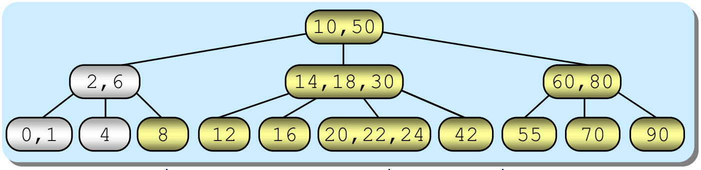


รูปที่ 10-31 ตัวอย่างต้นไม้ได้ดุล 2-3-4

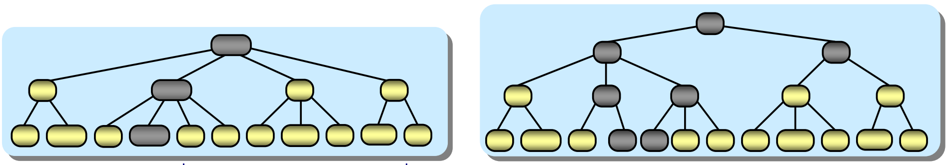
การเพิ่มข้อมูลในต้นไม้ได้ดุล 2-3-4 ก็คล้าย ๆ กับที่ผ่านมา คือไล่เปรียบเทียบจากรากลงมาตามผลการเปรียบเทียบจนถึงใบ ถ้าใบนั้นเป็นแบบ 2 หรือ 3 ก็เพิ่มข้อมูลลงในใบ เปลี่ยนให้เป็นแบบ 3 หรือ แบบ 4 เช่น การเพิ่ม 9 ในรูปที่ 10-31 ก็เพียงแค่เพิ่ม 9 ในใบที่มี 8 อยู่ เป็นต้น ปัญหาเกิดขึ้นก็เมื่อใบที่พบเป็นแบบ 4 ซึ่งมีข้อมูล 3 ตัวในใบ ไม่มีที่ให้เพิ่มข้อมูลใหม่ จะใช้วิธีสร้างใบเก็บข้อมูลใหม่แล้วต่อเป็นลูกก็ไม่ได้ เพราะจะทำให้ใบทุกใบไม่อยู่ในระดับเดียวกันตามข้อกำหนด เช่น การเพิ่ม 1 จะจบการค้นที่ใบซ้ายสุดที่เป็นปมแบบ 4 วิธีแก้ปัญหาคือ ดันข้อมูลตัวตรงกลางขึ้นไปให้พ่อ ทำให้มีที่เพิ่มข้อมูลใหม่ จากนั้นแตกใบนี้ออกเป็นสองใบ พ่อจะได้มีลูกครบ เช่น การเพิ่ม 1 ในใบที่มี 0,2,4 ทำได้โดยดัน 2 ขึ้นไปให้พ่อที่มีแต่ 6 ให้เป็น 2,6 จากนั้นเพิ่ม 1 ในใบ 0,4 กลายเป็น 0,1,4 จากนั้นแตกใบนี้เป็น 0,1 กับ 4 ให้เป็นลูกของปม 2,6 ได้ดังรูปที่ 10-32

ถ้าเพิ่มแล้วต้องแตกใบ ผลักตัวกลางไปให้ปมพ่อ แต่ปมพ่อก็เป็นแบบ 4 แล้วจะทำอย่างไร ? ก็คงทำในลักษณะเดียวกัน คือแตกปมพ่อ ผลักตัวกลางให้ปมปู่ เช่น การเพิ่ม 21 ในรูปที่ 10-32 ทำให้

ต้องแตกปม 20,22,24 ย้าย 22 ไปเพิ่มให้ปม 14,18,30 ที่ต้องแตกปมอีก ส่ง 18 ขึ้นไปเพิ่มในปมปู่ 10,50 ซึ่งมีที่ว่าง และถ้าปมปู่เป็นแบบ 4 ด้วย ก็คงทำแบบเดิมอีกนั่นแหละ ทำไปเรื่อย ๆ จนถึงราก ซึ่งถ้ายังเป็นแบบ 4 อีก ก็แตกรากเป็นสองปม แล้วสร้างรากใหม่ ดังตัวอย่างในรูปที่ 10-33 ทางซ้าย ถ้าเราเพิ่มข้อมูลที่ต้องจบลงที่ปมทึบด้านล่าง โดยปมต่าง ๆ ตั้งแต่รากลงมาเป็นปมแบบ 4 หมด ก็ต้องแตกปมและผลักข้อมูลขึ้นตลอด จนต้องสร้างรากใหม่ ได้รูปทางขวา



รูปที่ 10-32 ตัวอย่างการเพิ่ม 1 ในรูปที่ 10-31


รูปที่ 10-33 ตัวอย่างการเพิ่มข้อมูลแล้วเกิดการแตกปมไปจนถึงราก

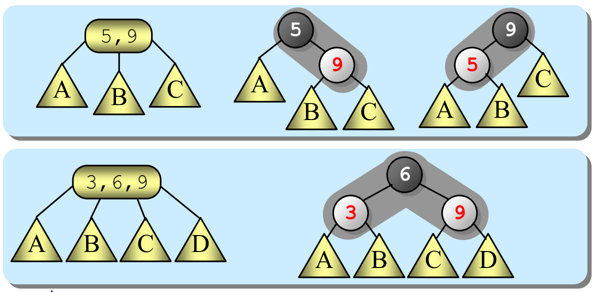
กระบวนการเพิ่มข้อมูลที่อธิบายมาข้างต้นนี้ เกิดการแตกปมและส่งข้อมูลขึ้นในทิศทางจากใบไปสู่ราก (แบบ bottom up) โดยจะแตกปมและส่งข้อมูลขึ้นเมื่อจำเป็นเท่านั้น ยังมีอีกวิธีหนึ่งที่แตกปมแบบ 4 ทุกครั้งที่พบตามวิถีการค้นจากรากลงมา (แบบ top-down) เมื่อลงถึงใบแล้วเริ่มเพิ่มจะได้มีช่องว่างทันที วิธีนี้ทำให้มีปมแบบ 4 น้อย เพราะถ้าพบก็จะแตกปมเตรียมไว้เลย ทำให้ตอนเพิ่มเกิดการแตกปมน้อย

สำหรับการลบข้อมูลออกจากต้นไม้ได้ดุล 2-3-4 ถ้าลบข้อมูลของปมภายใน ก็ให้นำข้อมูลตัวน้อยสุดของลูกถัดไปมาแทนตัวที่จะลบ แล้วเปลี่ยนปัญหาไปลบตัวน้อยสุดตัวนั้น เช่น ในรูปที่ 10-32 อยากลบ 10 ก็ให้นำ 12 มาแทน อยากลบ 50 ก็ให้นำ 55 มาแทน อยากลบ 18 ก็ให้นำ 20 มาแทน เป็นต้น แล้วจะลบข้อมูลที่ใบอย่างไร ? ในกรณีที่ลบแล้ว ยังเหลือข้อมูลในปม ก็ไม่มีปัญหาอะไร แต่ถ้าลบแล้วไม่เหลือ เช่น อยากลบ 16 ในรูปที่ 10-32 ก็จะเกิดกระบวนการย้าย 14 ในปมพ่อโอนให้พี่ทางซ้ายซึ่งคือ 12 หรือถ้าอยากลบ 42 ในรูปที่ 10-32 เราย้าย 30 ในปมพ่อให้พี่ทางซ้ายไม่ได้เพราะเต็ม ก็ให้ย้าย 30 ของพ่อลงมา และโอนของพี่คือ 24 ขึ้นไปแทนที่พ่อ กระบวนการลบนี้มีเรื่องจุกจิกขอละเป็นแบบฝึกหัดให้ผู้อ่านคิดต่อในรายละเอียด

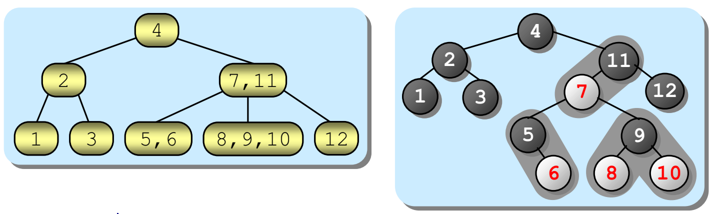
ต้นไม้ได้ดุล 2-3-4 ประกันความสูงว่า เป็น O(log n) อีกทั้งการเพิ่มและลบล้วนใช้เวลาเป็น O(log n) ด้วย จึงเป็นต้นไม้ค้นหาที่มีประสิทธิภาพในการให้บริการที่ดี

ต้นไม้แดงดำ

แนวคิดของต้นไม้ได้ดุล 2-3-4 น่าสนใจมาก แต่การเขียนโปรแกรมในทางปฏิบัตินั้นคงไม่สร้างปมสามแบบ หรือถ้าสร้างเฉพาะแบบ 4 แล้วใช้แทนอีกสองแบบขึ้นกับจำนวนข้อมูลที่เก็บ ก็จะเปลืองมาก มีวิธีสร้างต้นไม้ได้ดุล 2-3-4 วิธีหนึ่งที่น่าสนใจมาก เรียกว่า *ต้นไม้แดงดำ* (red-black tree) ซึ่งใช้ปมแบบ 2 อย่างเดียว โดยจัดกลุ่มของปมแบบ 2 หลายปม เพื่อแทนปมแบบ 3 และแบบ 4 ดังรูปที่ 10-34 รูปบนใช้ปมแบบ 2 สองปมต่อกันแทนปมแบบ 3 ซึ่งสามารถจัดได้สองแบบ ส่วนรูปล่างใช้ปมแบบ 2 สามปมต่อกันเพื่อแทนปมแบบ 4 ปมในต้นไม้แดงดำที่ใช้มีสองชนิดคือปมดำกับปมแดง (ในรูปปมดำมีสีทึบ ส่วนปมแดงมีสีอ่อนกว่า) รูปที่ 10-35 แสดงตัวอย่างการแทนต้นไม้ได้ดุล 2-3-4 ด้วยต้นไม้แดงดำ ซึ่งเพียงแต่เปลี่ยนปมแบบ 3 และแบบ 4 ด้วยการแทนในรูปที่ 10-34 (กรณีของปมแบบ 3 แทนได้สองแบบ เลือกแบบใดก็ได้) โดยปมแบบ 2 ในต้นไม้ได้ดุล 2-3-4 ให้แทนด้วยปมดำ สิ่งที่เห็นได้ชัดอย่างหนึ่งคือ ต้นไม้แดงดำที่ได้ก็คือต้นไม้ค้นหาแบบทวิภาค ต้นไม้แดงดำสูงอย่างมากเป็นสองเท่าของต้นไม้ได้ดุล 2-3-4 ดังนั้นต้นไม้แดงดำก็มีความสูงเป็น O(log n)



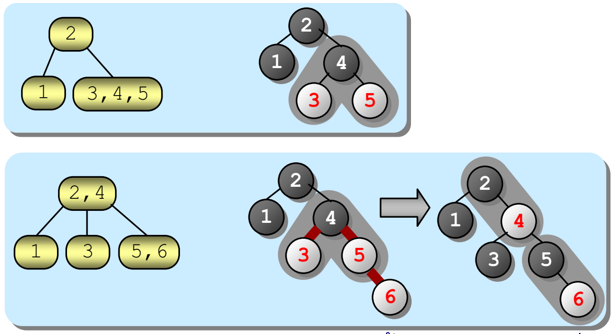
รูปที่ 10-34 การแทนปมแบบ 3 และแบบ 4 ด้วยปมแบบ 2



รูปที่ 10-35 การแทนต้นไม้ได้ดุล 2-3-4 ด้วยต้นไม้แดงดำ

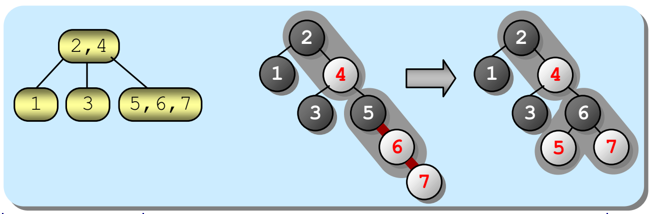
ความเท่ของต้นไม้แดงดำก็คือ การแตกปมแบบ 4 หนึ่งปมไปเป็นแบบ 2 สองปม สามารถทำได้โดยเพียงแค่เปลี่ยนสีปม ซึ่งรวมถึงการเพิ่มข้อมูลในปมก็กระทำได้ง่ายเช่นเดียวกัน รูปที่ 10-36 แสดงการเพิ่ม 6 เข้าในต้นไม้แดงดำ เราก็เพียงเพิ่มใบ 6 ตามปกติ ให้ปมใหม่มีสีแดง แล้วเริ่มตรวจสอบ หากพบว่า มีปมแดงสองปมใดเป็นพ่อลูกกัน แสดงว่า ผิดปกติ (เพราะแทนกลับไปเป็นปมแบบ 3 ก็ไม่ได้แบบ 4 ก็ไม่ได้ ดังรูปที่ 10-34) ในตัวอย่างนี้คือ 5 กับ 6 ซึ่งแก้ได้ด้วยการเปลี่ยนสีปม 3, 4, 5 จากดำ

เป็นแดง จากแดงเป็นดำ ซึ่งเทียบได้กับการแตกปมและผลักข้อมูลตัวกลางขึ้นไปพร้อมกัน ไปผลดังต้นไม้ในรูปที่ 10-36 ล่างขวา

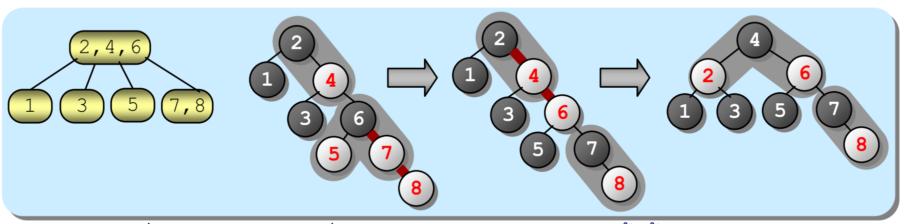


รูปที่ 10-36 การแตกปมและการผลักข้อมูลขึ้นทำได้ด้วยการเปลี่ยนสีปม

แต่ในบางกรณีเมื่อพบปมแดงเป็นพ่อลูกกัน ก็ต้องเปลี่ยนสีปมพร้อมกับหมุนปมด้วย ดังตัวอย่าง เราเพิ่ม 7 จากผลในรูปที่ 10-36 ได้ดังรูปที่ 10-37 หลังเพิ่มใบ 7 แล้วพบว่า พ่อลูก 6 และ 7 เป็นปมแดงทั้งคู่ ต้องทั้งหมุนและเปลี่ยนสีดังแสดงในรูป ถ้าเราเพิ่ม 8 ต่อจะได้ดังรูปที่ 10-38 เป็นการเพิ่มใบ 8 เกิดการแตกปมและผลักข้อมูลขึ้น (ด้วยการเปลี่ยนสีปม 5, 6, 7) ตามด้วยการเพิ่มข้อมูลในปมแบบ 3 ให้เป็นแบบ 4 ด้วยการหมุนและเปลี่ยนสี ได้ดังต้นไม้ขวาสุดในรูป



รูปที่ 10-37 การเพิ่มข้อมูลแล้วผิดปกติต้องอาศัยการหมุนและเปลี่ยนสีปม

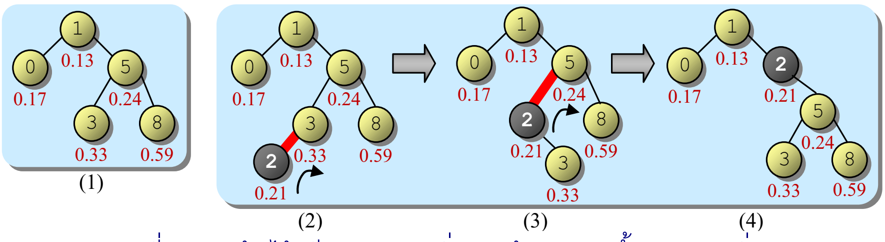


รูปที่ 10-38 การแก้ไขสิ่งผิดปกติ อาจต้องทำหลายครั้งขึ้นไปจนถึงราก

แต่ละปมของต้นไม้แดงดำต้องเก็บสถานะภายในปมเพื่อระบุว่า เป็นปมแดงหรือปมดำ (ซึ่งใช้เพียงแค่ 1 บิตต่อปมเท่านั้น) ขอไม่ลงรายละเอียดเกี่ยวกับการหมุนและเปลี่ยนสีปมให้ครบทุกกรณี เพียงแต่ต้องการนำเสนอแนวคิดการออกแบบต้นไม้ที่แทนต้นไม้ได้ดุล 2-3-4 ซึ่งมีประสิทธิภาพทั้งในแง่ของการจัดเก็บและจัดการ โครงสร้างข้อมูล

ต้นไม้ทรีป

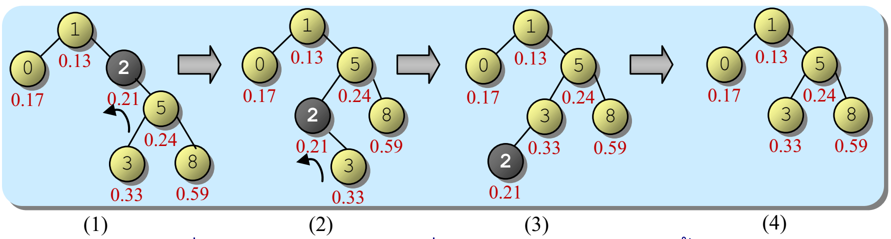
ทรีป (Treap) มาจากคำว่า tree + heap เป็นต้นไม้ค้นหาแบบทวิภาคชนิดหนึ่งที่แต่ละปมนอกจากจะเก็บข้อมูลแล้ว ยังเก็บจำนวนจริง p กำกับโดยมีข้อบังคับเพิ่มว่า ค่า p ที่กำกับปมต่าง ๆ ต้องมีอันดับแบบฮีป (คือ p ของปมพ่อต้องมีค่าไม่มากกว่าของปมลูก เสมือนเป็นฮีปน้อยสุด) รูปที่ 10-39 (1) แสดงตัวอย่างทรีป โดย p ของปมเขียนอยู่ใต้ปม



รูปที่ 10-39 ต้นไม้ทรีป และการเพิ่ม 2 แล้วหมุน 2 ขึ้นหลังการเพิ่ม

การเพิ่มข้อมูลใหม่ในทรีป มีการทำงานเหมือนกับต้นไม้ค้นหาแบบทวิภาคทุกประการ แต่หลังจากเพิ่มเป็นใบเสร็จแล้ว จะตรวจสอบว่า ค่า p ใหม่ผิดอันดับแบบฮีปหรือไม่ ถ้าผิด สามารถแก้ไขได้ด้วยการหมุน ดังตัวอย่างเราเพิ่ม 2 ในรูปที่ 10-39 (1) ได้ต้นไม้ในรูป (2) แต่เนื่องจาก p ของ 2 น้อยกว่าของ 3 จึงหมุน 2 ขึ้นได้รูป (3) พบว่า p ของ 2 ก็ยังน้อยกว่าของปมพ่อ ก็หมุนต่อได้รูป (4) ได้อันดับแบบฮีปที่ถูกต้อง ก็ได้ทรีปที่ถูกต้อง

สำหรับการลบ จะทำงานในทิศทางที่กลับกับการเพิ่ม คือเมื่อพบปมที่จะลบ ให้หมุนปมนั้นลงไปจนเป็นใบ แล้วก็ลบใบนั้นทิ้ง เช่น การลบ 2 ออกจากทรีปในรูปที่ 10-40 (1) ทำได้ด้วยการหมุน 2 ลงไปจนเป็นใบได้ดังรูป (3) แล้วก็ลบ 2 ทิ้งได้ง่าย ๆ

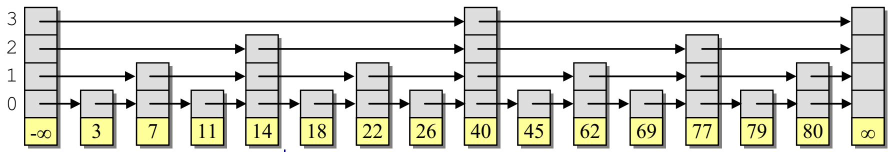


รูปที่ 10-40 การหมุนปม 2 ที่จะลบลงเป็นใบ แล้วลบทิ้ง

แล้วค่า p นี้ได้มาจากผู้ใด ? ไม่มีใครให้ค่า p มาหรอก ทรีปผลิตค่า p เอง และผลิตแบบสุ่ม เป็นจำนวนจริงระหว่าง 0 ถึง 1 ตอนสร้างปม ซึ่งสามารถวิเคราะห์ได้ว่า ด้วยค่าสุ่ม ประกอบกับการหมุนง่าย ๆ ดังที่นำเสนอมานี้ทำให้โดยเฉลี่ยแล้วทรีปสามารถเพิ่ม ลบ และค้นหาข้อมูลได้ในเวลา $O(\log n)$

รายการก้าวกระโดด

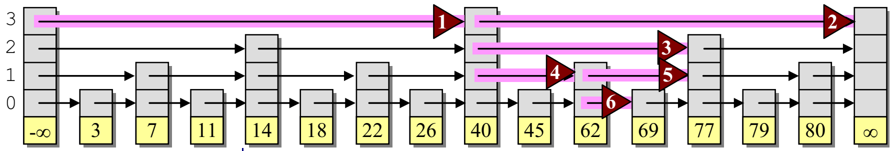
ขอปิดท้ายบท ด้วยโครงสร้างข้อมูลที่ดูแล้วไม่เหมือนต้นไม้ ชื่อว่า *รายการก้าวกระโดด* (skip list) ต้นไม้ค้นหาที่ได้อธิบายกันมา ถ้าเป็นแบบทวิภาคธรรมดาพื้น ๆ ก็เสี่ยงว่า อาจสูงมากได้ เพราะลำดับของข้อมูลที่นำมาจัดเก็บมีผลต่อรูปร่างต้นไม้ ถ้าเปลี่ยนมาใช้ต้นไม้เอวีแอล ต้นไม้แดงดำ ก็ประกันความสูง แต่ต้องจัดเก็บข้อมูลเสริมประจำปม (ความสูงหรือสีปม) มีการหมุนมากมายหลากหลายกรณี หรือจะใช้ต้นไม้บาน แต่ละปมไม่ต้องจัดเก็บข้อมูลเสริม ให้ผลตอบแทนในระยะยาวที่ดี แต่ก็มีบางจังหวะที่อาจทำงานช้าได้ หรือถ้าต้องการเขียนโปรแกรมง่าย ประกันประสิทธิภาพด้วยความน่าจะเป็นสูง ก็ต้องต้นไม้ทรีป แต่ก็อย่าลืมว่า แต่ละปมต้องเก็บข้อมูลเสริม (คือค่า p เพื่อการจัดอันดับแบบฮีปภายในต้น) นอกจากนี้ทุก ๆ แบบที่กล่าวมาล้วนเป็นต้นไม้แบบทวิภาค หมายความว่า ต่อหนึ่งปมต้องมีตัวโยงไปหาลูกสองตัวแน่ ๆ มาดูรายการก้าวกระโดดกันดีกว่า ที่เรียกว่า ก้าวกระโดดก็เพราะมีการเพิ่มตัวโยงในรายการโยงให้โยงข้ามไปชี้ตัวไกล ๆ ในรายการได้ บางปมมีตัวโยงหนึ่งตัว บางตัวมีสอง บางตัวมีสาม ... แต่ทั้งนี้สามารถปรับจำนวนตัวโยงได้ ซึ่งสามารถวิเคราะห์ได้ว่า มีตัวโยงโดยเฉลี่ยน้อยกว่าสอง ไม่ต้องเก็บข้อมูลเสริมใด ๆ กำกับปมข้อมูล (นอกจากตัวโยง) ประสิทธิภาพการทำงานไม่ขึ้นกับลำดับการจัดเก็บของข้อมูล มีพฤติกรรมเชิงสุ่ม และใช้เวลาการทำงานเป็น O(log n)



รูปที่ 10-41 ตัวอย่างรายการก้าวกระโดด

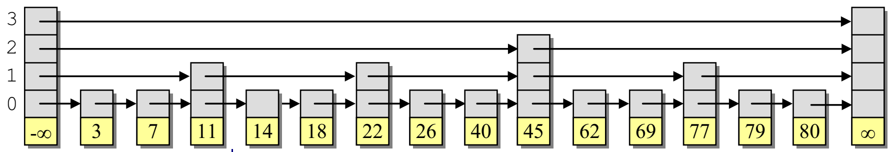
รูปที่ 10-41 แสดงตัวอย่างของรายการก้าวกระโดด ปมต่าง ๆ เก็บข้อมูลแล้วโยงกันให้ข้อมูลเรียงจากน้อยไปมาก มีปมหัวเก็บข้อมูลน้อยสุด (–∞) และมีปมท้ายเก็บข้อมูลมากสุด (+∞) ปิดหัวปิดท้ายรายการ แต่ละปมมีตัวโยงได้หลายตัว เราเรียกปมที่มีตัวโยง m ตัวว่า *ปมระดับ* m โดยตัวโยงล่างสุดของปมคือตัวโยงที่ 0 ตัวโยงบนสุดของปมระดับ m คือตัวโยงที่ m–1 ให้ปมหัวและปมท้ายเป็นปมระดับ M (โดย M เป็นค่ามากสุดของระดับปมซึ่งจะอธิบายต่อไป) ทุกปมมีตัวโยงที่ 0 ทุกปมเว้นปมมีตัวโยงที่ 1 ทุกปมเว้นสี่ปมมีตัวโยงที่ 2 กล่าวโดยทั่วไปคือทุกปมเว้น 2^i ปมมีตัวโยงที่ i ให้สังเกตในรูปว่า ตัวโยงที่ i ของปมจะชี้ไปยังปมถัดไปอีก 2^i ปมข้างหน้า ด้วยการจัดวางปมและตัวโยงในลักษณะนี้ การค้นหาข้อมูล x เริ่มที่ปมหัวในระดับบนสุด โยงไปปมใดก็นำ x ไปเทียบกับข้อมูลในปมนั้น ถ้าเท่ากันก็จบ ถ้า x มากกว่า ก็ค้นหาตามตัวโยงของปมในระดับเดียวกันต่อ ถ้า x น้อยกว่าก็ถอยกลับไปปมเดิม ลงไปที่ตัวโยงตัวที่ระดับต่ำกว่าหนึ่งระดับ แล้วค้นหาตามตัวโยงนั้นต่อ สมมติว่าต้องการค้น 69 ในรูปที่ 10-42 หมายเลขลูกศรในรูปแสดงลำดับการวิ่งตามตัวโยงระหว่างการค้นหา

เริ่มที่ตัวโยงที่ 3 บนสุดของปมหัว ไปที่ปม 40, พบว่า 69 มากกว่าก็โยงต่อ ไปที่ปมท้าย, พบว่า 69 น้อยกว่า ∞ ก็ถอยกลับมาที่ปม 40 ลงมาตัวโยงที่ 2 ไปที่ปม 77, พบว่า 69 น้อยกว่าเช่นกัน ก็ถอยกลับมาที่ปม 40 อีกครั้ง ลงมาตัวโยงที่ 1 ไปที่ปม 62, พบว่า 69 มากกว่า ก็โยงต่อ ไปที่ปม 77, พบว่า 69 น้อยกว่า ถอยกลับมาที่ปม 62, แล้วลงมาตัวโยงที่ 0 วิ่งตามตัวโยงก็พบปม 69 ที่ต้องการ ในกรณีที่ตามตัวโยงที่ 0 แล้วพบปมที่มีค่ามากกว่า x ก็สรุปได้ว่า หา x ไม่พบ



รูปที่ 10-42 การค้นหา 69 ในรายการก้าวกระโดด

เราสามารถเปลี่ยนกฎการโยงจาก "ทุกปมเว้น 2^i ปมมีตัวโยงที่ i ซึ่งชี้ไปยังปมถัดไปอีก 2^i ปมข้างหน้า" ให้เป็น "ทุกปมเว้น k^i ปมมีตัวโยงที่ i ซึ่งชี้ไปยังปมถัดไปอีก k^i ปมข้างหน้า" สำหรับกรณี k=3 ในรูปที่ 10-43 จะได้ว่า ใช้เวลาการค้นแปรตาม $k \log_k n$ (ขอไม่วิเคราะห์)



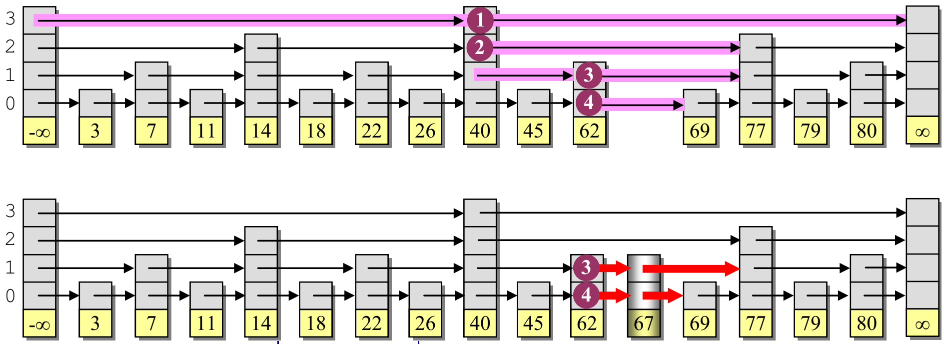
รูปที่ 10-43 ตัวอย่างรายการก้าวกระโดดแบบ $t = 3$

แล้วจะเพิ่มหรือลบข้อมูลอย่างไร ? ถ้าเราจะรักษากฎระเบียบของการโยงที่ว่า ทุกปมเว้น k^i ปมต้องมีตัวโยงที่ i ซึ่งชี้ไปยังปมถัดไปอีก k^i ปมข้างหน้า ก็คงจะลำบาก เพราะการเพิ่มเข้าหรือลบออกสักปม ต้องเปลี่ยนตัวโยงของปมในรายการมากมาย มีผู้เสนอว่า เราไม่ต้องเคร่งครัดกฎการโยงมากนัก จากการสังเกตพบว่า ทุกปมต้องมีตัวโยงอย่างน้อย 1 ตัว ทุก k ปมจะมีหนึ่งปมที่มีตัวโยงอย่างน้อย 2 ตัวโยง, ..., สรุปได้ว่า ทุก k^i ปมจะมีหนึ่งปมที่มีตัวโยงอย่างน้อย i+1 ตัว ดังนั้นทุกครั้งที่สร้างปมใหม่ก็พยายามสร้างให้ได้ตามการกระจายนี้ ดังรหัสที่ 10-26 ซึ่งสุ่มจำนวนจริงในช่วง [0..1) ไปเรื่อย ๆ ถ้าสุ่มได้ค่าน้อยกว่า $1/k$ ติดกัน i ครั้งก็ได้ปมระดับ i+1

```
int randomLevel(int k, int M) { // M คือระดับมากสุด
  double p = 1.0 / t;
  int i = 1;
  while( Math.random() < p && i < M) i++;
  return i;
}
```

รหัสที่ 10-26 การสุ่มระดับของปมใหม่ในรายการก้าวกระโดด

การเพิ่มข้อมูลเริ่มด้วยการค้นหาข้อมูลจนจบ ณ ปมที่สรุปได้ว่าไม่พบ โดยต้องจำตัวโยงตัวล่าสุดในระดับต่าง ๆ ระหว่างการค้นไว้ เช่น ถ้าต้องการเพิ่ม 67 ในรูปที่ 10-44 (บน) การค้นจบที่ตัวโยงที่ 0 ของปม 62 ตัวโยงที่ต้องจำไว้คือตัวโยงหมายเลข 1, 2, 3, และ 4 จากนั้นสุ่มระดับของปมใหม่ สมมติว่า ได้ปมระดับ 2 ก็ให้วางปมใหม่ไว้หลังปมของตัวโยงสุดท้ายของการค้น ดังรูปที่ 10-44 (ล่าง) แล้วเปลี่ยนค่าของตัวโยงต่าง ๆ ให้เหมาะสม โดยพิจารณาว่า ปมใหม่ไปขวางเส้นโยงเก่าเส้นใดก็เปลี่ยนเส้นนั้น ซึ่งทำได้ด้วยการนำค่าของตัวโยงหมายเลข 3 และ 4 มาใส่ที่ตัวโยงที่ 1 และ 0 ของปมใหม่ และเปลี่ยนตัวโยงหมายเลข 3 และ 4 มาชี้ที่ปมใหม่ที่เพิ่ม ได้ผลดังรูปที่ 10-44 (ล่าง) สำหรับการลบก็ทำง่าย ๆ ด้วยการลบปมที่ต้องการออก แล้วขยายตัวโยงจากที่เคยชี้มาที่ปมที่ถูกลบ ให้ชี้เลยไป



รูปที่ 10-44 การเพิ่ม 67 ในรายการก้าวกระโดด

แล้ว k และ M ควรมีค่าเท่าใด M คือค่ามากสุดของระดับปม ให้ n คือปริมาณข้อมูลที่คาดว่าจะเก็บ ตัวโยงบนสุดที่ M–1 ชี้ไปยัง k^{M-1} ปมถัดไปควรมีค่า n/k ดังนั้น $M = \log_k n$ ส่วนค่าของ k มีผลต่อเวลาการค้นซึ่งแปรตาม $k \log_k n$ (ขอไม่วิเคราะห์ให้ดู) และมีผลต่อจำนวนตัวโยงเฉลี่ยต่อปม ซึ่งสามารถวิเคราะห์ได้ง่าย ๆ ดังนี้ ให้สังเกตว่า มี n ปมที่มีตัวโยงที่ 0, มี n/k ปมที่มีตัวโยงที่ 1, มี n/k^2 ปมที่มีตัวโยงที่ 2, ... รวมแล้วมีตัวโยงทั้งหมด $n + n/k + n/k^2 + n/k^3 + \ldots = n/(1 - 1/k)$ ดังนั้นเฉลี่ยต่อปมคือ $k/(k - 1)$ สรุปได้ว่า ถ้า k มีค่ามาก จำนวนตัวโยงเฉลี่ยต่อหนึ่งปม ก็มีค่าน้อย เป็นการประหยัดเนื้อที่ แต่ก็ค้นช้าลง ดังตัวอย่างในตารางที่ 10-3 ค่าของ k จึงเป็นค่าที่ผู้ใช้มีไว้ปรับเพื่อเพิ่มเวลาแลกกับการลดเนื้อที่ (ซึ่งไม่สามารถทำได้ในต้นไม้ค้นหาแบบอื่น ๆ)

ตารางที่ 10-3 ผลของค่า k ต่อเวลาในการค้นและจำนวนตัวโยงเฉลี่ยต่อปม

k	ค่าสัมพัทธ์ของเวลาในการค้น $(k \log_k n) / (2 \log_2 n)$	จำนวนตัวโยงเฉลี่ยต่อปม $k / (k-1)$
2	1	2
3	0.946...	1.5
4	1	1.333...
5	1.077...	1.25
6	1.161...	1.2
7	1.247...	1.167...

แบบฝึกหัด

1. จงเขียนคลาส `BSTree` และ `AVLTree` ด้วยตนเอง โดยไม่ดูรายละเอียดในหนังสือ

2. จงเขียนเมท็อด `removeMin` และ `removeMax` เพื่อลบตัวที่มีค่าน้อยสุด และมากสุดตามลำดับ ให้กับคลาส `BSTree`

3. จงเขียนเมท็อดเสริมให้กับคลาส `BSTree` เพื่อทดลองวัดความลึกเฉลี่ยของปมภายในและของปมภายนอก และยืนยันผลการวิเคราะห์ว่า ต้นไม้ค้นหาแบบทวิภาคมี $D_E(n) = D_I(n) + 2$

4. จงเขียนคลาส `BSTPriorityQueue implements PriorityQueue` ที่สร้างแถวคอยบุริมภาพด้วยต้นไม้ค้นหาแบบทวิภาค

5. จงเขียน `BSTree(Object[] a)` ซึ่งคือตัวสร้าง ให้กับคลาส `BSTree` ที่รับข้อมูลซึ่งเก็บในแถวลำดับ `a` มาสร้างต้นไม้ค้นหาแบบทวิภาคที่ได้ดุล

6. จงเขียนเมท็อดมาตรฐาน `equals` ให้กับคลาส `BSTree` (โดยถือว่า ต้นไม้ต้องมีโครงสร้างและข้อมูลเหมือนกันจึงเท่ากัน)

7. จงเขียนเมท็อดมาตรฐาน `equals` ให้กับคลาส `BSTSet` รหัสที่ 10-15 (เนื่องจากเป็นเซตความเท่ากันจึงไม่ขึ้นกับโครงสร้างของต้นไม้ ต้นไม้รูปร่างไม่เหมือนกัน แต่เก็บข้อมูลเหมือนกันก็ถือได้ว่าเท่ากัน)

8. ถ้าเพิ่มเมท็อด `goo` ข้างล่างนี้ให้กับคลาส `BSTree` อยากทราบว่า `goo` ทำอะไร ?

```
public void goo(Object e) {
  return goo(root, e);
}
private void goo(Node r, Object e) {
  if (r == null) r = new Node(e, null, null);
  else {
    int cmp = ((Comparable) r.element).compareTo(e);
    if (cmp > 0) {
      r.left  = goo(r.left, e);
      r = rotateLeftChild(r);
    } else if (cmp < 0) {
      r.right = goo(r.right, e);
      r = rotateRightChild(r);
    }
  }
  return r;
}
```

9. ถ้าเพิ่มเมท็อด foo ข้างล่างนี้ให้กับคลาส BSTree อยากทราบว่าทำอะไร (foo มีการเรียก goo ของแบบฝึกหัดข้อที่แล้วด้วย)

```
public void foo(BSTree t) {
  root = foo(root, t.root);
}
private Node foo(Node a, Node b) {
  if (a == null) return b;
  if (b == null) return a;
  goo(b, a.element);    // goo เป็นเมท็อดในแบบฝึกหัดข้อที่ 8
  b.left  = foo(a.left, b.left);
  b.right = foo(a.right, b.right);
  return b;
}
```

10. ถ้าเขียนเมท็อด contains ข้างล่างนี้ให้กับคลาส BSTree จะทำงานได้ถูกต้องหรือไม่ และใช้เวลาเท่าใด

```
public boolean contains(Object x) {
  return contains(root, x);
}
private boolean contains(Node r, Object x) {
  if (r == null) return false;
  if (x.equals(r.element)) return true;
  if (contains(r.left, x)) return true;
  return contains(r.right, x);
}
```

11. จงวาดการเปลี่ยนแปลงของต้นไม้ เริ่มจากต้นไม้เอวีแอลว่าง ๆ ต้นหนึ่ง แล้วเพิ่มข้อมูลตามลำดับดังนี้ 1, 2, 3, 4, 5, 6, 7, 15, 14, 13, 12, 11, 10, 9, 8

12. รหัสที่ 10-22 แสดงให้เห็นการประกาศตัวแปร height แบบ int กำกับตามปมต่าง ๆ ในต้นไม้เอวีแอล น้องนัทเสนอว่า ในทางปฏิบัติเราไม่จำเป็นต้องใช้ int ซึ่งในเนื้อที่ตั้ง 4 ไบต์ต่อหนึ่งปม ทำไมไม่ใช้แค่ byte ก็พอ เก็บจำนวนเต็มได้เหมือนกัน กินที่แค่ 1 ไบต์ต่อปม อยากทราบว่า น่าเชื่อน้องนัทไหม จะมีปัญหาอะไรหรือไม่ อย่างไร

13. จงเขียนเมท็อด Object select(int k) ให้กับคลาส BSTree ที่คืนข้อมูลตัวน้อยสุดอันดับ *k* ในต้นไม้ค้นหาแบบทวิภาค

14. จงเขียนคลาสใหม่ชื่อ BSTreeX ที่เพิ่มตัวแปร size ไว้ตามปมต่าง ๆ เพื่อเก็บจำนวนปมของต้นไม้ย่อย ทั้งนี้ก็เพื่อจะเขียนเมท็อด Object select(int k) ที่ทำงานได้เร็วกว่าที่เขียนในแบบฝึกหัดข้อที่แล้ว

15. จงเขียนเมท็อด put และ toString ให้กับ BSTMap ในรหัสที่ 10-19

16. จงเขียนคลาส `BSTCollection` ใหม่ โดยแทนที่ข้อมูลที่ซ้ำกันต้องอยู่คนละปมกันในต้นไม้ ก็เก็บข้อมูลที่ซ้ำเพียงตัวเดียวที่ปม แล้วให้แต่ละปมมีตัวแปร `count` เก็บจำนวนการซ้ำกันของข้อมูลนั้นในคอลเล็กชัน

17. จงวิจารณ์ข้อดีข้อเสียของการสร้าง `BSTMap` ทั้งสองแบบในรหัสที่ 10-19 และรหัสที่ 10-20

18. ทำไมต้นไม้บาน หรือ splay tree จึงได้รับการตั้งชื่อเช่นนี้

19. จงออกแบบกระบวนการลบข้อมูลในต้นไม้ได้ดุล 2-3-4 (ไม่ต้องเขียนเป็นเมท็อด)

20. จงหาเหตุผลประกอบว่า ทำไมเราถึงไม่ออกแบบต้นไม้ได้ดุล 2-3-4-5-6-7-8-9-10-11-12 ซึ่งก็จะได้ต้นไม้ที่เตี้ยกว่าที่ได้นำเสนอมา

21. จงหาเหตุผลประกอบว่า ทำไมผู้ออกแบบต้นไม้เอวีแอลต้องตั้งกฎแค่ว่า ต้นซ้ายและต้นขวาห้ามสูงต่างกันเกินหนึ่ง ทำไมไม่ทะเยอทะยานตั้งกฎว่า ต้นไม้ต้องได้ดุลไปเลย รับรองว่า ต้องเป็นต้นไม้เตี้ยสุด ๆ

22. จงพิสูจน์ว่า วิถีจากรากถึงใบใด ๆ ในต้นไม้แดงดำ ประกอบด้วยปมดำเป็นจำนวนเท่ากันหมด

23. จงเขียนคลาส `SplayTreeSet implements Set` ที่สร้างเซตด้วยต้นไม้บาน

24. จงเขียนคลาส `TreapSet implements Set` ที่สร้างเซตด้วยทรีป

25. จงเขียนคลาส `SkipListMap implements Map` ที่สร้างแมปด้วยรายการก้าวกระโดด

11 ตารางแฮช

จะว่าไปแล้วต้นไม้ค้นหาแบบทวิภาคเป็นโครงสร้างข้อมูลที่รองรับความต้องการสารพัดรูปแบบได้อย่างมีประสิทธิภาพ ไม่ว่าจะเป็นการเพิ่ม ลบ ค้นหา ค้นหาตัวน้อยสุด ตัวมากสุด ตัวถัดไป หรือแม้กระทั่งไล่เรียงข้อมูลจากน้อยไปมาก แต่ถ้าเราจำกัดความต้องการเหลือแค่การเพิ่ม ลบ และค้นหาโดยไม่มีการดำเนินการที่เกี่ยวข้องกับอันดับของข้อมูลเลย เราสามารถใช้โครงสร้างข้อมูลอีกแบบหนึ่งที่เรียกว่า *ตารางแฮช* (hash table) [1] ซึ่งมีประสิทธิภาพที่เหนือกว่า ที่ผ่านมาโครงสร้างข้อมูลต่าง ๆ ใช้การจำตำแหน่งข้อมูลในโครงสร้าง การดำเนินการต่าง ๆ อาศัยตำแหน่งที่จำไว้อย่างมีระเบียบเพื่อค้นหาข้อมูลได้รวดเร็ว ในขณะที่ตารางแฮชใช้การคำนวณตำแหน่ง โดยนำตัวข้อมูลไปผ่านกระบวนการคำนวณ ได้เป็นตำแหน่งที่เก็บของข้อมูลนั้นในเวลาอันรวดเร็ว นอกจากนี้ตารางแฮชยังเอื้ออำนวยให้ผู้ใช้สามารถปรับเนื้อที่ในการจัดเก็บเพื่อแลกกับเวลาการทำงานได้ด้วย

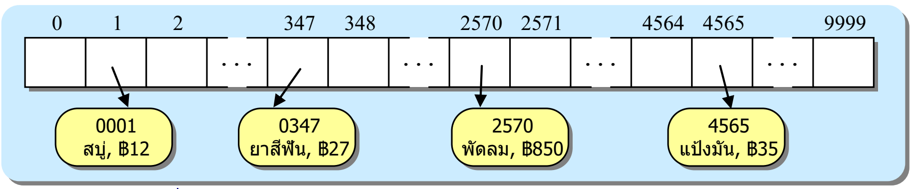
ตารางเก็บข้อมูล

สมมติว่า เราต้องการเก็บสินค้า 500 ประเภทไว้เพื่อการสืบค้น ถ้าเราเก็บในแถวลำดับเรียงจากน้อยไปมาก แล้วใช้การค้นหาแบบทวิภาค จะได้ว่า กรณีช้าสุดต้องพิจารณาข้อมูล $1 + \lfloor \log_2 500 \rfloor = 9$ ตัว แต่การเก็บในแถวลำดับเรียงจากน้อยไปมากเช่นนี้ ทำให้การเพิ่มและลบข้อมูลใช้เวลา $O(n)$ หากเราหันไปเก็บข้อมูลในต้นไม้เอวีแอล ซึ่งสูงไม่เกิน $1.44 \log_2 500 - 1.328$ (จากบทที่แล้ว) แสดงว่า กรณีช้าสุดต้องพิจารณาข้อมูล 12 ตัว แต่การเพิ่มและลบข้อมูลทำได้ในเวลา $O(\log n)$ ถ้าสินค้า 500 ประเภท

---

[1] Dumey เป็นผู้นำเสนอแนวคิดของตารางแฮชในปี ค.ศ. 1956 สำหรับใช้สร้างตารางสัญลักษณ์ (symbol table) เพื่อเก็บชื่อต่าง ๆ ของโปรแกรม เช่นชื่อตัวแปร ชื่อฟังก์ชัน ระหว่างการแปลโปรแกรมคอมพิวเตอร์

นี้ ใช้รหัสสินค้าเป็นคีย์[2] โดยรหัสสินค้าเป็นจำนวนเต็มขนาด 4 หลัก เราสามารถใช้แถวลำดับขนาด 10^4 = 10,000 ช่องเป็นตารางเก็บสินค้า แล้วให้สินค้าที่มีรหัส x ไปเก็บในช่องที่ x ดังตัวอย่างในรูปที่ 11-1 เช่น สินค้าที่มีรหัส 347 ถูกนำไปเก็บไว้ในช่องที่ 347 เป็นต้น การใช้ตารางเก็บข้อมูลนี้เป็นวิธีที่ง่าย และใช้เวลาน้อยมากในการเพิ่ม ลบ และค้นหา รหัสที่ 11-1 แสดงคลาส `AbstractTable` แทนตารางเก็บข้อมูลแบบแมป ซึ่งอาศัยเมท็อด `f` ในการแปลงคีย์ x ไปเป็นดัชนีหรือเลขที่ช่องของแถวลำดับ เรียกว่า *ฟังก์ชันดัชนี* (index function) ภายในมีแถวลำดับ `table` เก็บข้อมูล มี `size` ไว้นับจำนวนข้อมูล ผู้ใช้กำหนดขนาดของตารางตอนเรียกตัวสร้าง เมท็อด `containsKey` อาศัยการเรียก `f(key)` จะได้เลขที่ช่อง หยิบข้อมูลในช่องนั้นมาดู ถ้าไม่เป็น `null` แสดงว่า นี่คือข้อมูลของ `key` นี้ (บรรทัดที่ 15) ซึ่งก็เหมือนกับเมท็อด `get` ที่คืน `table[f(key)]` ได้เลย ส่วน `put` ต้องเก็บข้อมูลเก่าไว้ก่อน แล้วค่อยนำข้อมูลใหม่ใส่กลับเข้าไป ในกรณีที่ของเก่าไม่มี ก็ต้องเพิ่มค่าให้ `size` ด้วย และสุดท้ายคือ `remove` เพียงแค่นำค่า `null` ไปใส่ใน `table[f(key)]` และลดค่า `size` ลง ก็เป็นการลบข้อมูลนั้นออกจากตาราง ใครต้องการเก็บข้อมูลในลักษณะนี้ก็สร้างคลาสลูกของ `AbstractTable` แล้วเขียนเมท็อด `f` ให้ครบ เช่น รหัสที่ 11-2 แสดงคลาส `ProductTable` ซึ่งออกแบบให้เป็นตารางเก็บสินค้าโดยใช้รหัสสินค้าเป็นดัชนี



รูปที่ 11-1 ตัวอย่างการเก็บข้อมูลในตาราง ใช้รหัสสินค้าเป็นดัชนี

วิธีที่กล่าวมาข้างต้นรวดเร็วมาก แต่มีปัญหาตรงที่ แถวลำดับที่ใช้จะมีขนาดใหญ่หรือเล็กขึ้นกับลักษณะของข้อมูลที่ใช้ในการค้นหา การเก็บสินค้าจำนวน 500 ประเภทในแถวลำดับขนาด 10,000 ช่อง ใช้เนื้อที่เพียง 5% จากขนาดที่จองไว้ ถือว่าเป็นการใช้เนื้อที่ที่ไม่คุ้มเลย เราสามารถลดขนาดของแถวลำดับได้ด้วยการหาฟังก์ชันแบบหนึ่งต่อหนึ่ง (หมายความว่า ถ้า $x \neq y$, $f(x)$ ต้องไม่เท่ากับ $f(y)$) โดยแปลงคีย์ให้เป็นเลขที่ช่องในช่วง $[0, m-1]$ ซึ่งค่า m นี้ก็คือขนาดของแถวลำดับที่เราพอยอมรับได้ สมมติว่า พอไปดูความหมายของรหัสสินค้าที่เป็นจำนวนเต็ม 4 หลักแล้วพบว่า หลักหน่วยของรหัสเป็นเลขตรวจสอบที่ผู้ออกแบบรหัสใส่เพิ่มไว้ตรวจสอบและป้องกันความผิดพลาดของการป้อนรหัสสินค้า เมื่อรู้เช่นนี้ ก็สามารถตัดหลักหน่วยออกจากการนำมาคำนวณเลขที่อยู่ ได้ตัวอย่างการจัดเก็บใน

[2] คีย์ (key) คือส่วนของข้อมูลที่ข้อมูลแต่ละตัวมีค่าต่างกัน เช่นใช้เลขประจำประชาชนเป็นคีย์ของคนไทย ใช้รหัสสินค้าเป็นคีย์ของสินค้า เป็นต้น เรามักใช้คีย์เป็นตัวค้นหาข้อมูล

รูปที่ 11-2 เขียนเป็นคลาสเก็บสินค้าที่ได้นำเสนอมาในรหัสที่ 11-3 ทำให้แถวลำดับที่ใช้เหลือแค่ 1,000 ช่อง ใช้เนื้อที่ได้คุ้มถึง 50% ของแถวลำดับที่จอง

```
public abstract class AbstractTable implements Map {
  private Object[] table;
  private int size = 0;

  protected AbstractTable(int m) {
    table = new Object[m];
  }
  public boolean isEmpty() {
    return size == 0;
  }
  public int size() {
    return size;
  }
  public boolean containsKey(Object key) {
    return table[f(key)] != null;
  }
  public Object get(Object key) {
    return table[f(key)];
  }
  public Object put(Object key, Object value) {
    Object oldValue = get(key);
    table[f(key)] = value;
    if (oldValue == null) ++size;
    return oldValue;
  }
  public void remove(Object x) {
    if (table[f(x)] != null) --size;
    table[f(x)] = null;
  }
  protected abstract int f(Object key);
}
```

ตารางเก็บ value

คืน value ที่คู่กับ key ที่ให้มา

คืน value เก่า ถ้าเดิมมีเก็บใน table

คืนเลขที่ช่องของ table ที่เก็บ value ของ key ที่ให้มา

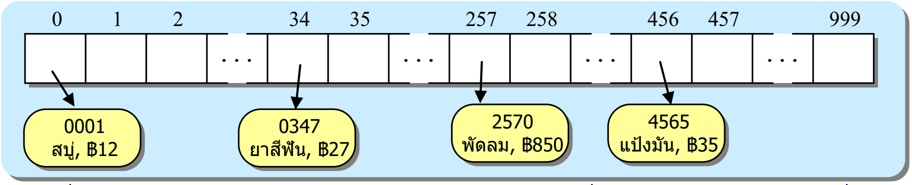
รหัสที่ 11-1 ตารางเก็บข้อมูลซึ่งใช้ฟังก์ชันดัชนีคำนวนช่องที่เก็บข้อมูล

```
public ProductTable extends AbstractTable {
  public ProductTable() {
    super(10000);
  }
  protected int f(Object x) {
    return Integer.parseInt((String)x);
  }
}
```

คีย์ 4 หลักต้องจองหมื่นช่อง

เช่น “123” เปลี่ยนเป็น 123 หมายความว่า value ของคีย์ “123” เก็บในช่องที่ 123

รหัสที่ 11-2 คลาสตารางเก็บสินค้า โดยใช้รหัสสินค้าเป็นดัชนี



รูปที่ 11-2 ตัวอย่างการเก็บสินค้าในตาราง ใช้รหัสสินค้าที่ไม่คิดหลักหน่วยเป็นเลขที่อยู่

```
public ProductTable extends AbstractTable {
  public ProductTable() {
    super(1000);
  }
  protected int f(Object x) {
    return Integer.parseInt((String)x) / 10;
  }
}
```

คีย์มี 3 หลัก จึงจองพันช่อง

ตัดหลักหน่วยทิ้ง

รหัสที่ 11-3 คลาสตารางเก็บสินค้า โดยใช้รหัสสินค้าที่ไม่คิดหลักหน่วยเป็นเลขที่อยู่ของข้อมูล

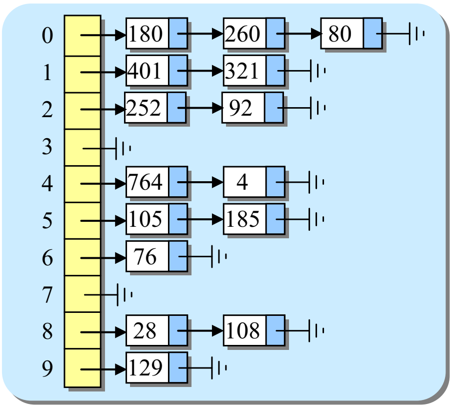
จากตัวอย่างการจัดเก็บสินค้าในตารางที่ได้นำเสนอมา ถ้ารหัสสินค้าเป็นเลข 8 หลัก แล้วเราใช้ฟังก์ชันดัชนี $f(x) = x$ คงเห็นได้ชัดว่า ต้องสร้างแถวลำดับขนาด 10^8 ช่อง ซึ่งใหญ่เกินไปในทางปฏิบัติ แต่ถ้าเรารู้ก่อนว่า สินค้าที่จะจัดเก็บมีรหัสดังนี้

10293451, 18763481, 54129823, 78766712, 87228124, 09829321, 78910114, 98984523

ก็อาจใช้ฟังก์ชันดัชนี $f(x) = (\lfloor x/100 \rfloor \,\%\, 10) + (\lfloor x/10 \rfloor \,\%\, 10)$ กับข้อมูลชุดบนนี้ ได้ผลลัพธ์เป็น 9, 12, 10, 8, 3, 5, 2, 7 ตามลำดับ ซึ่งไม่มีเลขที่ช่องใดซ้ำกันเลย และใช้ตารางขนาดเพียง 13 ช่องเท่านั้น (เพราะดัชนีตัวมากสุดที่ได้คือ 12) (ฟังก์ชัน $f(x)$ นี้อาจดูรู้สึกซับซ้อน ความจริงคือการนำเลขหลักร้อยของ x บวกกับเลขหลักสิบของ x) แต่ถ้าเปลี่ยนคีย์ตัวแรกในชุดจาก 10293451 เป็น 10293415 จะได้ดัชนีเป็น 5 ซึ่งไปซ้ำกับของ 09829321 ก็แสดงว่า $f(x)$ ใช้ไม่ได้กับข้อมูลชุดใหม่ กล่าวโดยสรุป การเก็บข้อมูลในตารางโดยอาศัยฟังก์ชันดัชนีนั้น ใช้ได้ดีเมื่อเราสามารถหาฟังก์ชันที่เป็นแบบหนึ่งต่อหนึ่ง (เพราะข้อมูลต่างกันจะได้เก็บในตารางคนละช่องกัน) และเป็นฟังก์ชันที่มีพิสัยไม่กว้างนัก (เพราะพิสัยเป็นตัวกำหนดขนาดของตารางที่ต้องสร้าง) จากตัวอย่างข้างต้น ทำให้เห็นว่าการออกแบบฟังก์ชันดัชนีให้ได้ตามเงื่อนไขทั้งสองนี้เป็นเรื่องยาก ถ้าเราไม่รู้ลักษณะของข้อมูลเลย หรือถึงแม้จะรู้ แต่ไม่รู้ชุดข้อมูลจริงก็คงทำไม่ได้ เพราะโดยปกติช่วงของคีย์ที่เป็นไปได้จะกว้างกว่าขนาดของตาราง ก็ย่อมต้องมีคีย์บางคู่ที่เมื่อผ่านฟังก์ชันดัชนีแล้วต้องเก็บในช่องเดียวกัน เราเรียกสภาพเช่นนี้ว่า *การชน* (collision) ทางออกของปัญหานี้ก็คือคงต้องยอมใช้ฟังก์ชันที่เกิดการชนได้ แต่ขอให้ชนน้อย ๆ, ใช้ตารางขนาดไม่ใหญ่นัก, แล้วค่อยหาวิธีแก้ปัญหาการชนว่าจะจัดการอย่างไรดี

ตารางแฮชแบบแยกกันโยง

ก่อนอื่นขอทวนเพื่อความเข้าใจร่วมกัน เราต้องการใช้ตารางขนาด m ช่องเก็บข้อมูล มีฟังก์ชัน $h(x)$ [3] ซึ่งแปลงคีย์ x ไปเป็นเลขที่ช่องของตารางในช่วง 0 ถึง $m-1$ โดยอนุญาตให้คีย์ชนกันได้ (นั่นคือคีย์ $x \neq y$ แต่ $h(x) = h(y)$) ขอเสนอการจัดเก็บตารางที่เรียกว่า *ตารางแฮชแบบแยกกันโยง* (separate chaining hash table) ซึ่งจัดเก็บข้อมูลต่าง ๆ ที่ชนที่ช่องเดียวกันให้อยู่ในรายการโยงเดียวกันที่เก็บในช่องนั้น รูปที่ 11-3 แสดงตัวอย่างของตารางแฮชแบบแยกกันโยง มีตารางขนาด 10 ช่อง ใช้ฟังก์ชัน $h(x) = x \% 10$ ในการคำนวณเลขที่ช่องของคีย์ x เมื่อต้องการค้นหา x ให้นำ x ไปผ่านฟังก์ชัน h แล้วไปค้นหาในรายการโยงที่ช่อง $h(x)$ ของตาราง ถ้าต้องการเพิ่มข้อมูลที่มีคีย์เป็น x ก็ให้นำข้อมูลไปเพิ่มในรายการโยงที่ช่อง $h(x)$ ของตาราง (เพื่อให้เพิ่มได้เร็วก็น่าจะเพิ่มที่ต้นรายการ) การลบข้อมูลที่มีคีย์เป็น x ออกจากตารางก็เช่นเดียวกัน คือการลบข้อมูลนั้นออกจากรายการโยงในช่อง $h(x)$ ของตาราง



รูปที่ 11-3 ตัวอย่างตารางแฮชแบบแยกกันโยงโดยใช้ $h(x) = x \% 10$

รหัสที่ 11-4 แสดงส่วนข้อมูลภายในคลาส `SeparateChainingHashMap` ซึ่งคือที่เก็บข้อมูลแบบแมป สร้างด้วยตารางแฮชแบบแยกกันโยง โดยแต่ละรายการโยงเป็นแบบโยงเดี่ยวไม่มีปมหัว เหมือนกับที่แสดงในรูปที่ 11-3 มีคลาส `LinkedNode` อยู่ภายในที่นิยามแต่ละปมข้อมูลให้ประกอบด้วยข้อมูลส่วนที่เป็น `key` ส่วนที่เป็น `value` และส่วนที่เป็นตัวโยงปมถัดไป (ชื่อว่า `next`) ตัวตารางชื่อ `table` ซึ่งแต่ละช่องเก็บตัวอ้างอิงไปยังปมแรกของรายการโยง ถ้าไม่มีรายการโยงก็ให้ช่องนั้นเป็น `null` มีตัวแปร `size` คอยนับจำนวนข้อมูลในแมป เหมือนกับที่เคยทำมา ขนาดของตารางกำหนดโดยผู้ใช้ซึ่งให้มาทางตัวสร้าง (บรรทัดที่ 11) แล้วจองแถวลำดับตามขนาดที่กำหนด

[3] ขอเปลี่ยนจากฟังก์ชันดัชนี $f(x)$ ที่ไม่อนุญาตให้ชนกัน มาใช้สัญลักษณ์ใหม่ $h(x)$ ที่เราคาดไว้ว่าเกิดการชนได้

```
public class SeparateChainingHashMap implements Map {
  private static class LinkedNode {
    Object key, value;
    LinkedNode next;
    LinkedNode(Object k, Object v, LinkedNode n) {
      key = k; value = v; next = n;
    }
  }
  private int size;
  private LinkedNode[] table;
  public SeparateChainingHashMap(int cap) {
    table = new LinkedNode[cap];
  }
  public int size() { return size; }
  public boolean isEmpty() { return size == 0; }
```

ในปมเก็บทั้ง key, value และตัวโยงปมถัดไป

แต่ละช่องเก็บตัวโยงไปยังปมแรกของรายการ

รหัสที่ 11-4 คลาส `SeparateChainingHashMap` ที่สร้างแมปด้วยตารางแฮชแบบแยกกันโยง

รหัสที่ 11-5 แสดงเมท็อด `get` ซึ่งนำคีย์ไปค้นด้วยเมท็อด `getNode` หากผลที่ได้เป็น `null` แสดงว่าหาไม่พบ ถ้าไม่เป็น แสดงว่าได้ปมข้อมูลกลับมา ก็คืน `value` ของปมนั้น `containsKey` ตรวจสอบว่า คีย์ที่ได้รับเก็บอยู่ในแมปนี้หรือไม่ ซึ่งก็อาศัย `getNode` เช่นกัน เมท็อด `getNode` นำคีย์ที่ได้ ไปผ่านเมท็อด `h` ซึ่งเรียกเมท็อด `hashCode` ของตัวคีย์ (ขอชี้แจงก่อนว่า `hashCode` เป็นเมท็อดที่คืนจำนวนเต็มออกมาหนึ่งค่า รายละเอียดของเมท็อดนี้จะนำเสนอในภายหลัง) ได้ผลกลับมาทำให้เป็นค่าไม่ติดลบแล้วมอดุโลด้วยขนาดของตาราง (บรรทัดที่ 31) ได้จำนวนเต็มซึ่งมีค่าระหว่าง `0` ถึง `table.length-1` นำผลนี้ไปเป็นเลขที่ช่องเพื่อหยิบปมแรกของรายการมาค้น (บรรทัดที่ 24) ด้วยวงวนการเปรียบเทียบคีย์ในบรรทัดที่ 25 ถึง 27 ถ้าพบก็คืนปมกลับไป ถ้าไม่พบก็คืน `null`

```
16   public Object get(Object key) {
17     LinkedNode node = getNode(key);
18     return node == null ? null : node.value;
19   }
20   public boolean containsKey(Object key) {
21     return getNode(key) != null;
22   }
23   private LinkedNode getNode(Object key) {
24     LinkedNode cur = table[h(key)];
25     while (cur != null && !cur.key.equals(key)) {
26       cur = cur.next;
27     }
28     return cur;
29   }
30   private int h(Object x) {
31     return Math.abs(x.hashCode()) % table.length;
32   }
```

คืน value ของปมที่เก็บ key

ค้นหาไม่พบ คืน null

ค้นหาในรายการที่ปมแรกเก็บใน ช่องที่ h(key) ของ table

hashCode คืนจำนวนเต็ม

รหัสที่ 11-5 เมท็อด `get` และ `containsKey` ของตารางแฮชแบบแยกกันโยง

```
33   public Object put(Object key, Object value) {
34     LinkedNode node = getNode(key);
35     Object oldValue = null;
36     if (node != null) {
37       oldValue = node.value;
38       node.value = value;
39     } else {
40       int h = h(key);
41       table[h] = new LinkedNode(key, value, table[h]);
42       ++size;
43     }
44     return oldValue;
45   }
46   public void remove(Object key) {
47     int h = h(key);
48     if (table[h] == null) return;
49     if (table[h].key.equals(key)) {
50       table[h] = table[h].next; --size;
51     } else {
52       LinkedNode prev = table[h];
53       while (prev.next != null && !prev.next.key.equals(key)) {
54         prev = prev.next;
55       }
56       if (prev.next != null) {
57         prev.next = prev.next.next; --size;
58       }
59     }
60   }
61 }
```

พบปมที่เก็บ key ก็ใส่ value ใหม่ในปม

ไม่พบ ก็สร้างปมใหม่เพิ่มใส่ข้างหน้ารายการ

กรณีลบปมแรกของรายการ

วงวนค้นปมก่อนหน้าปมที่เก็บ key

ถ้าพบ ก็ลบปมนั้นออก

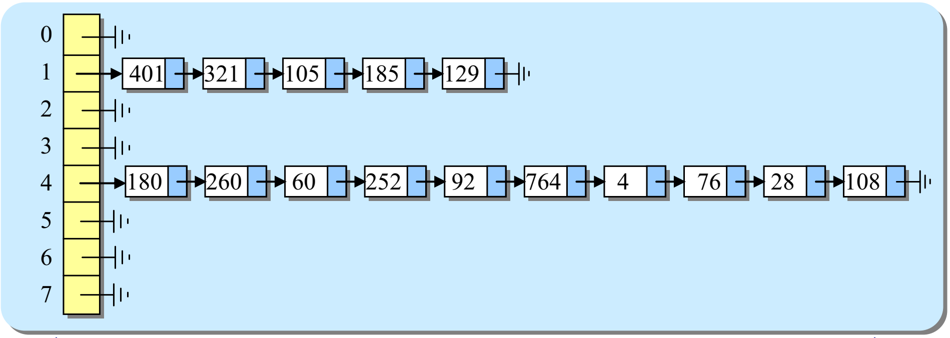
รหัสที่ 11-6 เมท็อด put และ remove สำหรับการเพิ่มและลบข้อมูล

ปิดท้ายด้วยเมท็อด put และ remove เพื่อเพิ่มและลบข้อมูล ดังแสดงในรหัสที่ 11-6 เมท็อด put ต้องตรวจสอบก่อนว่า มี key เก็บอยู่หรือไม่ ถ้ามีก็เปลี่ยน value ของเก่าเป็นของใหม่ (บรรทัดที่ 38) แต่ต้องจำของเก่า (บรรทัดที่ 37) เพื่อคืนกลับด้วย ถ้าค้นไม่พบ ก็สร้างปมใหม่แล้วเพิ่มไว้หน้ารายการโยงที่ช่อง table[h(key)] (บรรทัดที่ 41) ส่วนเมท็อด remove ลบข้อมูลในรายการโยงที่ช่อง table[h(key)] เนื่องจากเราใช้รายการโยงแบบไม่มีปมหัว ดังนั้นจึงต้องตรวจสอบเป็นกรณีพิเศษว่า ปมแรกเก็บ key ที่ต้องการลบหรือไม่ ถ้าใช่ก็ลบปมแรกทิ้ง (บรรทัดที่ 50) ถ้าไม่ใช่ปมแรกที่ต้องการลบ ก็ไล่เปรียบเทียบในรายการจนพบ หรือจนหมดรายการ โดยใช้ตัวแปร prev ชี้ปมก่อนปมที่สนใจ (prev ชี้ปมใด เราสนใจปมถัดจากที่ prev ชี้) เมื่อพบ ก็ลบปม prev.next (บรรทัดที่ 57)

เมท็อดต่าง ๆ ต้องค้นข้อมูล จึงใช้เวลาเท่ากับเวลาของเมท็อด h และเวลาในการค้นในรายการโยงที่ช่อง table[h(key)] ดังนั้นสิ่งที่ต้องการคือให้ h ทำงานเร็ว ๆ และค้นในรายการโยงสั้น ๆ

ฟังก์ชันแฮช

ข้อมูลใดจะไปเก็บอยู่ในรายการที่ช่องไหน ขึ้นกับเมท็อด h และตัวข้อมูลเอง นั่นหมายความว่า ในกรณีของการเก็บข้อมูลในตารางแฮชแบบแยกกันโยงนั้น ความยาวของรายการโยงจึงขึ้นกับลักษณะของข้อมูลและเมท็อด h ถ้าเราเปลี่ยน $h(x) = x \% 10$ ที่ใช้ในรูปที่ 11-3 มาเป็น $h(x) = x \% 8$ ด้วยชุดข้อมูลเดียวกัน จะได้ผลการเก็บดังรูปที่ 11-4 ซึ่งมีรายการโยงที่ยาวกว่าแบบเดิมในรูปที่ 11-3 เพราะชุดข้อมูลนี้เมื่อใช้กับ $h(x)$ ตัวใหม่เกิดการชนมากกว่า



รูปที่ 11-4 ตารางแฮชแบบแยกกันโยงโดยใช้ $h(x) = x \% 8$ กับข้อมูลชุดเดียวกับในรูปที่ 11-3

ดังนั้นฟังก์ชัน $h(x)$ จึงมีผลโดยตรงกับประสิทธิภาพการทำงาน $h(x)$ ที่ดีควรเป็นฟังก์ชันที่คำนวณได้รวดเร็ว โดยทั่วไปใช้เวลาแปรตามขนาดของคีย์ ไม่ใช่จำนวนคีย์ และ $h(x)$ ควรทำให้เกิดการชนน้อย ๆ แต่เนื่องจากจำนวนการชนจะมีมากมีน้อยขึ้นกับชุดข้อมูลที่ถูกเลือกมาเก็บ จึงตีความแบบรวม ๆ ว่า $h(x)$ ที่ดีควรทำให้คีย์แต่ละตัวไปตกอยู่ในช่องใดในตารางด้วยโอกาสเท่า ๆ กัน เรียกลักษณะเช่นนี้ว่า simple uniform hashing กำหนดให้คีย์ต่าง ๆ เป็นจำนวนเต็มในช่วง $[0, U-1]$ และ m คือขนาดของตาราง $h(x)$ ทำหน้าที่แปลงจำนวนเต็มในช่วง $[0, U-1]$ เป็นจำนวนเต็มในช่วง $[0, m-1]$ ถ้า $p(x)$ คือความน่าจะเป็นที่คีย์ x จะถูกเลือกมาจากเซตของคีย์ที่เป็นไปได้ทั้งหมด และ S_j^h คือเซตของคีย์ซึ่งชนกันที่ช่อง j เมื่อใช้ $h(x)$ จะได้ว่า ฟังก์ชัน $h(x)$ เป็น simple uniform hashing เมื่อ

$$\sum_{x \in S_j^h} p(x) = \frac{1}{m} \qquad \text{สำหรับ } j = 0, 1, \ldots, m-1, \; S_j^h = \{ x \mid h(x) = j \}$$

สมมติว่า ใช้เลข 10 หลักเป็นคีย์ ค่าของคีย์ที่เป็นไปได้ก็คือ 0000000000 ถึง 9999999999 (นั่นคือ $U = 10^{10}$) กำหนดให้ทุก ๆ ค่าของคีย์เหล่านี้มีสิทธิ์ถูกเลือกมาเก็บได้เท่า ๆ กัน หมายความว่า $p(x)$ มีค่าเท่ากัน (ซึ่งก็คือเท่ากับ $1/10^{10}$) ถ้าเรานำคีย์ที่เป็นไปได้ทุก ๆ คีย์ (คือตั้งแต่คีย์ 0000000000 ถึง 9999999999) มาผ่าน $h(x)$ ได้ผลเป็นเลขที่ช่องของตาราง จะได้ว่า แต่ละช่องในตารางจะมีคีย์ที่ชนกันเป็นจำนวนเท่า ๆ กัน (เพราะเราสมมติให้แต่ละคีย์มีโอกาสถูกเลือกเท่า ๆ กัน) หมายความว่า การ

"โปรย" คีย์ต่าง ๆ ลงตารางด้วย $h(x)$ นั้นกระจายอย่างสม่ำเสมอดีมาก สมมติต่อว่า ถ้าชุดของคีย์ที่มาเก็บมี n ตัว โดยที่ $n < m$ และ $n << U$ (อ่านเครื่องหมาย << นี้ว่าน้อยกว่ามาก ๆ) เช่น $m = 1000$, $n = 700$ คีย์ 700 ตัวที่ถูกเลือกมาจาก 10^{10} ค่าที่เป็นไปได้นั้น เมื่อนำมาโปรยลงตารางแฮชโดยใช้ $h(x)$ ก็น่าจะกระจาย ๆ ในตารางเช่นกัน

ข่าวร้ายก็คือว่า ในทางปฏิบัติ $p(x)$ ของทุกคีย์ไม่เท่ากัน และเราก็ไม่รู้ค่า $p(x)$ จนกว่าจะรู้ชุดข้อมูลทั้งหมด (ผู้เขียนได้ลองเขียนโปรแกรมหาความถี่ของคำต่าง ๆ ในเนื้อเพลงไทยจำนวน 1,435 เพลง ได้ความถี่ของคำที่ใช้มากสุด 18 อันดับแรกดังแสดงในตารางที่ 11-1 ซึ่งตรงกับสามัญสำนึกที่ว่าการใช้คำต่าง ๆ มีโอกาสไม่เท่ากัน) จึงทำให้การออกแบบฟังก์ชันที่ให้ได้ผลกระจายอย่างสม่ำเสมอตามที่ต้องการนั้น เห็นจะเป็นไปได้ยาก

ตารางที่ 11-1 ความถี่ของคำที่ใช้มากสุด 18 คำแรกในเนื้อเพลงไทยจำนวน 1435 เพลง

คำ	ความถี่	คำ	ความถี่	คำ	ความถี่
ไม่	11130	ฉัน	5998	คน	4441
เธอ	10260	ที่	5673	เป็น	4057
จะ	8089	ใจ	5219	กัน	3796
ไป	6596	มี	5207	มัน	3629
ก็	6558	มา	5069	ได้	3091
ให้	6250	รัก	4993	ว่า	3007

แล้วจะออกแบบ $h(x)$ อย่างไรดี ? เราไม่รู้ลักษณะของข้อมูล แต่ต้องการให้เกิดการชนน้อย ๆ คือต้องการ $h(x)$ รับชุดของคีย์ แล้วส่งไปเก็บได้ "กระจาย ๆ" ทั่วตาราง ลองคิดแบบนี้ดู สมมติว่า คีย์ของข้อมูลเป็นจำนวนเต็มสุ่ม ถ้าตารางข้อมูลมีขนาด 100 ช่อง แล้วใช้ $h(x) = x \% 100$ (ซึ่งคือการใช้เฉพาะหลักสิบและหลักหน่วยแทนเลขที่ช่องของตาราง) ก็ย่อมได้การกระจายของข้อมูลในตารางที่ดี เพราะตัวคีย์เป็นจำนวนเต็มสุ่มโดยตัวมันเอง แต่ในทางปฏิบัติคีย์ไม่ใช่จำนวนเต็มสุ่ม ดังนั้นเราก็ต้องหาวิธีทำให้คีย์ที่อาจแลดูเป็นระเบียบ ไม่ใช่จำนวนสุ่ม แปลงให้มันดู "สุ่ม ๆ" ก่อน แล้วค่อยมามอดุโลด้วยขนาดของตาราง นั่นคือนำคีย์ที่ได้รับไป "สับให้เละ" "บดให้แหลกเหลว" จนไม่เห็นซากเดิม แต่ยังเป็นจำนวนเต็ม ที่นำไปใช้คำนวณเป็นเลขที่ช่องได้ จึงเรียก $h(x)$ ว่า *ฟังก์ชันแฮช* (hash function) [4] และเรียกตารางเก็บข้อมูลซึ่งใช้ฟังก์ชันแฮชว่า *ตารางแฮช* (hash table)

การแปลงคีย์ให้เป็นจำนวนเต็ม

ก่อนที่จะนำเสนอกลวิธีการออกแบบฟังก์ชันแฮช ต้องขอบอกก่อนว่า คีย์ของข้อมูลไม่จำเป็นต้องเป็นจำนวนเต็มเพียงประเภทเดียว อาจเป็นจำนวนจริง ตัวอักษร สตริง หรืออ็อบเจกต์ประเภทอื่น ๆ ก็ได้

[4] ผู้อ่านที่ยังไม่รู้ว่าคำว่า "hash" แปลว่าอะไรก็อยากให้เปิดพจนานุกรมตอนนี้เลย

ถ้าเราเขียนโปรแกรมด้วยรหัสคำสั่งระดับเครื่อง ข้อมูลทุกประเภทล้วนถูกเข้ารหัสด้วยเลขฐานสอง ทั้งสิ้น เช่น ค่า 0.5 แบบ `float` แทนด้วยรหัสฐานสอง 00111111000000000000000000000000 หรือสตริง **"ไท"** แทนด้วยรหัส 00001110010001000000111000010111 เป็นต้น ดังนั้นข้อมูลทุกประเภทล้วนมองเป็นจำนวนเต็มได้ทั้งสิ้น แต่ต้องเข้าใจด้วยว่า ถ้าคีย์มีลักษณะสุ่ม ไม่ได้หมายความว่า จำนวนเต็มที่ตีความจากรหัสฐานสอง จะเป็นจำนวนสุ่มด้วย เพราะข้อมูลทุก ๆ บิตในการเข้ารหัสไม่ได้ถูกใช้พอ ๆ กัน จึงควรมีกระบวนอื่นประกอบด้วย ในการแปลงคีย์ให้เป็นจำนวนเต็ม ดังนี้

- จำนวนจริง : ถ้าต้องการแปลงจำนวนจริง x ให้เป็นจำนวนเต็มในช่วง $[0, m-1]$ และคีย์ x อยู่ในช่วง $[0, 1)$ ก็เพียงแค่คูณ x ด้วย m แต่ถ้า x อยู่ในช่วง $[a, b)$ ก็สามารถปรับให้อยู่ในช่วง $[0, 1)$ ก่อนด้วย $(x - a) / (b - a)$ แล้วค่อยนำไปคูณด้วย m

- สตริง : ใช้การตีความสตริงเสมือนเป็นเลขฐาน k โดยทั่วไปให้ k มีค่าอย่างน้อยเท่ากับจำนวนตัวอักษรที่เป็นไปได้ในสตริง เช่น ถ้าเรารู้แน่ชัดว่า คีย์ที่ใช้เป็นสตริงของตัวอักษรภาษาอังกฤษเฉพาะตัวใหญ่เท่านั้น ก็มองคีย์นี้คล้ายเลขฐาน 26 เช่น `"JAVA"` ก็แปลงเป็น $74\times26^3 + 65\times26^2 + 86\times26^1 + 65\times26^0 = 1346865$ (รหัสของ `'J'` คือ 01001010 ฐานสองซึ่งเท่ากับ 74, รหัสของ `'A'` คือ 01000001 ฐานสองซึ่งเท่ากับ 65) เขียนเป็นเมท็อดดังแสดงในรหัสที่ 11-7

```
static int toInt(String x) {
  int h = 0;
  for (int i=0; i<x.length(); i++)
    h = 26 * h + x.charAt(i);
  return h & 0x7FFFFFFF;
}
```

มองตัวอักษรแต่ละตัวเป็นเลขฐาน 26

ทำให้เป็นจำนวนไม่ติดลบ โดยให้บิตขวาสุดเป็น 0

รหัสที่ 11-7 การแปลงสตริงของภาษาอังกฤษตัวใหญ่ให้เป็นจำนวนเต็ม

จริง ๆ แล้วก็ไม่จำเป็นต้องใช้ฐาน 26, ใช้ฐาน 32 ก็ได้ ถ้าใช้ฐาน 32 จะเขียนโปรแกรมให้ทำงานเร็วขึ้นได้ ด้วยการเลื่อนบิตไปทางซ้าย 5 บิต แทนการคูณ 32 (เพราะ $32 = 2^5$ และการเลื่อนซ้าย 1 บิตคือการคูณด้วย 2) นั่นคือเปลี่ยน `26*h` ในรหัสที่ 11-7 เป็น `h<<5` แต่บางคนก็พบว่า ใช้เลขฐานที่เป็นจำนวนเฉพาะจะได้ผลดีกว่า (ในเชิงของกระจายคีย์) เช่น คลาส `String` ของจาวาแปลงสตริงเป็นจำนวนเต็มโดยตีความสตริงเป็นเลขฐาน 31

- อ็อบเจกต์ : นำสมาชิกต่าง ๆ ที่กำกับอ็อบเจกต์มาแปลงให้เป็นจำนวนเต็ม (ถ้าเป็นแถวลำดับก็นำข้อมูลในแต่ละช่องมาแปลงเป็นจำนวนเต็ม) แล้วนำมา “รวม” กัน กลวิธีการรวมจำนวนเต็มของข้อมูลย่อยแต่ละตัวนั้นมีหลากหลายวิธี เช่น การนำสมาชิกของอ็อปเจกต์มาบวกกัน หรือออร์เฉพาะกัน (exclusive or) โดยทั่วไปเราเลือกเฉพาะข้อมูลที่สำคัญในอ็อบเจกต์มารวมกัน ไม่จำเป็นต้องทั้งหมด ข้อมูลตัวใดที่สามารถหาได้จากข้อมูลตัวอื่น ๆ ก็ไม่ต้องนำมาคิด

กลวิธีการเขียนฟังก์ชันแฮช

ก่อนอื่นต้องบอกก่อนว่า ฟังก์ชันแฮชที่ดีเขียนยาก เพราะเรายังไม่รู้เลยว่า ข้อมูลที่จะมาผ่านฟังก์ชันมีลักษณะอย่างไร โดยทั่วไปเราจะแปลง บด และสับคีย์ให้เป็นจำนวนเต็มที่มีขนาดไม่เกิน w โดยที่ w คือขนาดของจำนวนเต็มที่หน่วยประมวลผลสามารถจัดการแบบพื้นฐาน (เช่น w=32 ในจาวาเพราะ `int` มีขนาด 32 บิต) แล้วจึงปิดท้ายการแปลงให้เป็นจำนวนเต็มในช่วง $[0, m-1]$ โดยที่ m คือขนาดของตาราง ซึ่งทำได้โดยมอดุโลด้วย m เราได้นำเสนอกระบวนการแปลงคีย์เป็นจำนวนเต็ม ต่อไปนี้จะนำกระบวนการบดและสับจำนวนเต็ม (เมื่ออ้างถึงคีย์ ให้ถือว่า คีย์เป็นจำนวนเต็มไม่ติดลบ)

วิธีแรกอาศัยการมอดุโล นั่นคือใช้สูตร $h(x) = x \% p$ โดยทั่วไปมักหลีกเลี่ยง p ที่มีค่าเป็น 2^k เพราะผลที่ได้คือ k บิตทางขวา หรือในกรณีที่คีย์มีความหมายเป็นเลขฐานสิบ ก็ไม่ควรให้ $p = 10^k$ เพราะผลที่ได้คือ k หลักทางขวาเช่นกัน การใช้ p ในรูปแบบดังกล่าวทำให้คีย์ที่ต่างกันที่บิตซ้าย ๆ หรือหลักซ้าย จะไม่มีผลต่อค่า $h(x)$ ซึ่งไม่ตรงกับจุดประสงค์ของฟังก์ชันแฮช การเปลี่ยนบิตใด ๆ ของคีย์ควรให้ผลต่างกัน (และโดยทั่วไปต้องการให้ได้ผลต่างกันมาก ๆ ด้วย) นอกจากนี้ถ้าเราแปลงสตริงเป็นจำนวนเต็มด้วยการมองสตริงเป็นเลขฐาน 2^k (เช่น เลขฐาน 32, 64, ...) การให้ $p = 2^k - 1$ จะทำให้คำต่าง ๆ ที่เกิดจากการสลับอักษร (เช่น "**สมชาย**" กับ "**สายชม**") จะได้ผลลัพธ์ $h(x)$ ที่เหมือนกัน ซึ่งก็ไม่เป็นผลดี (ให้ผู้อ่านลองพิสูจน์ หรือลองเขียนโปรแกรมทดสอบดู) และจากความจริงที่ว่า ถ้า x และ p มี c เป็นตัวประกอบร่วม จะได้ $x \% p$ มีค่าเป็นจำนวนเท่าของ c ด้วยเหตุนี้ p จึงไม่ควรมีตัวประกอบที่มีค่าน้อย ๆ เพราะจะทำให้มี x จำนวนมากที่ได้ $x \% p$ มีค่าเป็นจำนวนเท่าของตัวประกอบนั้น ซึ่งไม่กระจาย ดังนั้นค่า p ที่นิยมใช้กันจะเป็นจำนวนเฉพาะ

วิธีที่สองอาศัยสูตร $h(x) = \lfloor m(xA - \lfloor xA \rfloor) \rfloor$ ซึ่งคือการคูณคีย์ด้วยจำนวนจริง A ที่มีค่าระหว่าง (0,1) แล้วนำเฉพาะเศษมาคูณด้วย m (m คือขนาดตาราง) เพื่อปรับให้เป็นจำนวนเต็มในช่วง $[0, m-1]$ ได้มีผู้ศึกษาพบว่า ค่า A ที่ดีคือ $(\sqrt{5} - 1)/2 \approx 0.618$ เขียนเป็นเมท็อดได้ดังรหัสที่ 11-8 (เมื่อ $m = 2^p$) ดูที่รหัสอาจไม่ค่อยเหมือนกับสูตรที่เขียนนัก แต่ความจริงทำงานเหมือนกัน บรรทัดแรกคำนวณค่าของ $(\sqrt{5} - 1)/2 \times 2^{32} = 2654435769$ คูณด้วยคีย์ `x` แล้วเหลือไว้แค่ 32 บิตทางขวา บรรทัดต่อมาเลื่อนบิตไปทางขวา `32-p` บิต (ซึ่งคือ `p` บิตทางซ้ายของ 32 บิตทางขวา) ก็ได้ผลลัพธ์ ลองใช้สูตรนี้กับ `x` = 1, 2, ..., 8 ด้วย `p` = 10 ได้ผลลัพธ์คือ 632,241,874,483,92,725,334,966 พบว่า ได้ค่าที่กระจายดีมาก

```
static int multHash(int x, int p) {
  long hash = (2654435769L * x) & 0xFFFFFFFFL;
  return (int) (hash >> (32-p));
}
```

รหัสที่ 11-8 เมท็อดคำนวณ $h(x) = \lfloor 2^p (x\phi - \lfloor x\phi \rfloor) \rfloor$ โดยที่ $\phi = (\sqrt{5} - 1)/2$

วิธีที่สามอาศัยการสับเปลี่ยนบิต ในกรณีที่คีย์มีขนาดยาว เช่น สตริง แถวลำดับ หรืออ็อบเจกต์ที่ประกอบด้วยข้อมูลภายในหลายตัว ก็ให้นำส่วนต่าง ๆ มารวมกัน โดยทั่วไปใช้การบวกหรือการออร์เฉพาะ (exclusive or) ซึ่งมีผลต่อการเปลี่ยนแปลงข้อมูลที่ดี แต่ถ้าต้องการให้ได้ผลลัพธ์ที่เกิดการเปลี่ยนแปลงของค่าแฮชมากขึ้น เราควรเรียงสับเปลี่ยนบิตภายในข้อมูลไปพร้อมกับการรวมด้วย เช่น รหัสที่ 11-9 หาค่าแฮชให้กับสตริงโดยนำแต่ละตัวอักษรมารวมกัน มีการหมุนบิตไปทางซ้าย 5 ตำแหน่งระหว่างการรวม รหัสที่ 11-10 แสดงตัวอย่างการเรียงสับเปลี่ยนที่ซับซ้อน แต่ได้ผลลัพธ์ที่ "เละ" ดังผลการทดลองที่ได้ในตารางที่ 11-2

```
static int rotatingHash(String x) {
  int hash = x.length();
  for (int i = 0; i < x.length(); i++)
    hash = ((hash << 5)^(hash >> 27))^x.charAt(i);
  return (hash & 0x7FFFFFFF);
}
```

รหัสที่ 11-9 ตัวอย่างการรวมข้อมูลและเรียงสับเปลี่ยนบิตระหว่างการหาค่าแฮช

```
static int mix(int x) {
  x = ~x + (x << 15);
  x ^= (x >>> 11);
  x += (x << 3);
  x ^= (x >>> 5);
  x += (x << 10);
  x ^= (x >>> 16);
  return x;
}
```

รหัสที่ 11-10 ตัวอย่างการเรียงสับเปลี่ยนบิตในคีย์

ตารางที่ 11-2 ตัวอย่างผลลัพธ์ของการเรียงสับเปลี่ยนคีย์ในรหัสที่ 11-10 (แสดงเป็นฐานสอง)

คีย์	ผลลัพธ์จาก mix ในรหัสที่ 11-10
00000000	01010101000101110101010100010111
00000001	00010001011101100111010111110100
00000010	00100010111011001110111111101001
00000011	00110100011000110100000111111011
00000100	01000101110110011101101111010010
00000101	01011011010100011111000111100110
00000110	01101000110001101000001111110110
00000111	01111110001111100100010110010011
00001000	10001011101100111011011110100101

วิธีที่สี่อาศัยแนวคิดการสุ่ม เพื่อลดโอกาสที่เราจะโชคร้ายได้ชุดข้อมูลซึ่งไม่เหมาะกับฟังก์ชันแฮชที่เลือกใช้ ต้องยอมรับว่า ไม่ว่าเราจะใช้กลวิธีใด, จะมอดุโลด้วยจำนวนเฉพาะ, คูณด้วย $(\sqrt{5}-1)/2$ เรียงสับเปลี่ยน, หรือผสมกับการออร์เฉพาะ ถ้าสิ่งที่ทำนั้นมีขั้นตอนที่แน่นอน ด้วยค่าคงตัวที่ไม่เปลี่ยนแปลง เราก็สามารถออกแบบชุดข้อมูลที่ทำให้ฟังก์ชันแฮชที่ใช้ เกิดการชนมากมายได้ วิธีหนึ่งที่

หลีกเลี่ยงเหตุการณ์ดังกล่าว ก็ด้วยการให้ฟังก์ชันแฮชที่ใช้มีพฤติกรรมไม่แน่นอน ! มีอยู่รูปแบบหนึ่งเรียกว่า *การแฮชเชิงเอกภพ* (universal hashing) ซึ่งมีสูตรดังนี้

$$h(x) = (\,(ax + b)\ \%\ p\,)\ \%\ m$$

โดยที่ x คือคีย์เป็นจำนวนเต็มในช่วง $[0, U\,)$, m เป็นขนาดของตารางแฮช, p เป็นจำนวนเฉพาะที่มีค่าในช่วง $[U, 2U\,)$ สิ่งที่ทำให้ฟังก์ชันนี้มีพฤติกรรมไม่แน่นอนก็คือ ให้ a และ b เป็นจำนวนสุ่ม โดยที่ $0 < a < p$ และ $0 \le b < p$ (ซึ่งสามารถพิสูจน์ให้เห็นจริงว่า จะได้การแฮชที่กระจายดี) โดยตอนเริ่มใช้งานตารางแฮช ก็สุ่มค่าทั้งสองเก็บไว้ จากนั้นก็ใช้ค่านั้นไปตลอดการทำงานของตารางแฮชนั้น ด้วยกลวิธีแบบนี้ ใครก็ตามที่ต้องการป้อนข้อมูลที่แกล้งให้การชนมาก ๆ ย่อมกระทำไม่ได้ เพราะไม่ทราบค่าของ a และ b

คลาส `Object` ในจาวามีเมท็อดชื่อว่า `hashCode` มีหน้าที่คืนจำนวนเต็มซึ่งได้มาจากข้อมูลที่เก็บอยู่ภายในอ็อบเจกต์ โดยตามมาตรฐานของจาวากำหนดไว้ว่า หากอ็อบเจกต์ `x` และ `y` มีค่าเท่ากัน คือ `x.equals(y)` เป็นจริง จะได้ว่า `x.hashCode()` ต้องมีค่าเท่ากับ `y.hashCode()` เมท็อด `hashCode` นอกจากมีหน้าที่คืนจำนวนเต็มแล้ว โดยทั่วไปยังผสมการนำข้อมูลภายในมา "บด" ให้แหลกเพื่อนำมาสร้างผลลัพธ์ที่เป็นจำนวนเต็ม ซึ่งสามารถนำไปใช้เป็นฟังก์ชันแฮชได้ `hashCode` ของอ็อบเจกต์นี้เองจึงถูกนำไปใช้ในคลาสมาตรฐานของจาวาที่ใช้ตารางแฮชเป็นโครงสร้างข้อมูล เช่น `HashMap`, `HashSet` เป็นต้น ดังตัวอย่าง คลาส `Point2D` ของจาวามีไว้แทนจุดในระนาบสองมิติ ภายในมีข้อมูลพิกัด `x` กับ `y` เก็บแบบ `double` มี `hashCode` ซึ่งนำค่า `x` และ `y` มาแปลงเป็นจำนวนเต็ม โดยมองรหัสที่แทน `double` แบบบิตให้เป็นแบบ `long` (โดยใช้เมท็อด `doubleToLongBits`) นำจำนวนเต็มที่แปลงจาก `y` คูณด้วย 31 แล้วออร์เฉพาะกับจำนวนเต็มที่แปลงจาก `x` เนื่องจากผลลัพธ์เป็น `long` ยาวกว่า `int` ที่ต้องคืน จึงนำเฉพาะ 32 บิตขวามาออร์เฉพาะกับ 32 บิตซ้ายได้เป็นผลลัพธ์ 32 บิต ดังแสดงด้วยรหัสข้างล่างนี้

```
public class Point2D {
  ...
  public int hashCode() {
    long bits = Double.doubleToLongBits(getX());
    bits ^= Double.doubleToLongBits(getY()) * 31;
    return (((int) bits) ^ ((int) (bits >> 32)));
  }
  ...
}
```

`hashCode` เป็นเมท็อดของคลาส `Object` ซึ่งเป็นคลาสบรรพบุรุษของทุก ๆ คลาสในจาวา ดังนั้นทุกอ็อบเจกต์จึงมี `hashCode` ให้เรียก โดยทั่วไปเราควรเขียน `hashCode` ของคลาสเราเอง นั่นคือให้ `override` ของคลาสบรรพบุรุษ เพราะ `hashCode` ของคลาส `Object` คืนเลขที่อยู่ในหน่วยความจำของอ็อบเจกต์ที่เรียก ดังนั้น `hashCode` ของอ็อบเจกต์ต่าง ๆ จะต่างกันหมด ซึ่งก็ใช้ได้กับ `equals` ของคลาส `Object` ที่เขียนไว้ว่าอ็อบเจกต์เท่ากัน ต้องเป็นอ็อบเจกต์เดียวกัน

ดังนั้นหากเราเขียนคลาสใหม่ โดยอ็อบเจกต์คนละตัวของคลาสนี้มีค่า "เท่ากัน" ได้ เราก็ควรเขียน `equals` เปรียบเทียบเอง และเขียน `hashCode` ให้ด้วย โดยนำข้อมูลภายในที่ใช้เปรียบเทียบใน `equals` มาใช้ในการคำนวณค่าแฮช

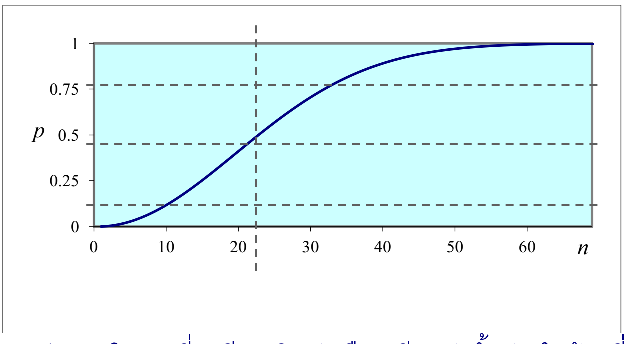
ปฏิทรรศน์วันเกิด

สิ่งที่ต้องเตรียมตัวเตรียมใจในการเก็บข้อมูลในตารางแฮช ไม่ว่าจะออกแบบฟังก์ชันแฮชซับซ้อนเพียงใด ก็อาจเกิดการชนได้ จะชนมาก ชนน้อย ขึ้นกับตัวฟังก์ชันแฮชที่ใช้ จำนวนข้อมูล และข้อมูลที่ได้รับ เราอาจรู้สึกว่า เก็บข้อมูลน้อย ๆ ในตารางขนาดใหญ่ ก็อาจไม่ชน เพื่อแสดงให้เห็นว่า โอกาสเกิดการชนมีสูง ถึงแม้ว่า เราใช้ฟังก์ชันแฮชที่ดีมาก ๆ กับจำนวนข้อมูลน้อย ๆ ลองมาหาคำตอบของปริศนาที่ว่า "ต้องมีสักกี่คนในห้อง ๆ หนึ่ง จึงจะมีโอกาสเกินครึ่งที่คนในห้องนี้มีวันเดือนเกิดซ้ำกัน" แน่นอนว่า ถ้าในห้องมีเกิน 366 คน ต้องมีคนเกิดวันเดือนซ้ำกันแน่ ๆ (ด้วยความน่าจะเป็น 1) แต่ที่เราต้องการคือ มีกี่คนจึงจะมีคนเกิดวันเดือนซ้ำด้วยความน่าจะเป็นเกิน 0.5 ผู้เขียนเคยถามวันเกิดนักเรียนในห้องเรียนจำนวน 40 คน ขณะสอนเรื่องตารางแฮชในปี พ.ศ. 2543 (รวมวันเดือนเกิดของผู้เขียนด้วย) พบว่า มีคนเกิดวันเดือนซ้ำกัน 2 คู่ ซึ่งอาจรู้สึกขัดกับความรู้สึกบ้างว่า แค่ 41 คน ก็ซ้ำกันตั้งสองคู่ (หรือว่าจะเป็นความบังเอิญ !) ลองมาวิเคราะห์ทางคณิตศาสตร์กัน กำหนดให้วันเดือนเกิดของคนนั้นเป็นฟังก์ชันที่กระจายดีมาก ๆ นั่นหมายความว่า คน ๆ หนึ่งมีโอกาส 1/366 ที่จะเกิดวันใดวันหนึ่งของปี ถ้าในห้องมี

- 1 คน : ความน่าจะเป็นที่มีคนเกิดวันเดือนซ้ำกันย่อมเป็น 1 – (366/366)
- 2 คน : ความน่าจะเป็นที่มีคนเกิดวันเดือนซ้ำกันย่อมเป็น 1 – (366/366)(365/366)
- ...
- n คน : ความน่าจะเป็นที่มีคนเกิดวันเดือนซ้ำกันย่อมเป็น

$$p = 1 - \left(\frac{366}{366} \cdot \frac{365}{366} \cdot \frac{364}{366} \cdot \ldots \cdot \frac{(366-n+1)}{366} \right)$$

ลองเขียนโปรแกรมเปลี่ยนค่า n แล้วคำนวณค่า p จะได้ผลดังรูปที่ 11-5 พบว่า n ต้องมีค่าอย่างน้อย 23 จึงทำให้ $p > 0.5$ เปรียบเสมือนการมีตาราง 366 ช่อง แล้วนำข้อมูลแฮชลงตารางแบบสุ่ม ๆ มีโอกาสเกินครึ่งที่จะเกิดการชน หลังจากแฮชตัวที่ 23 เป็นต้นไป



รูปที่ 11-5 กราฟแสดงโอกาสที่จะมีคนเกิดวันเดือนเดียวกันซ้ำกัน ในห้องที่มีคน n คน

ตารางแฮชแบบกำหนดเลขที่อยู่เปิด

ปฏิทรรศน์วันเกิดบอกเราว่า ถึงแม้ฟังก์ชันแฮชที่ใช้จะดี และข้อมูลมีไม่มาก ก็มีโอกาสที่จะเกิดการชน ดังนั้นจึงจำเป็นต้องหาวิธีแก้ไขปัญหาการชน เราได้นำเสนอตารางแฮชแบบแยกกันโยง ซึ่งนำข้อมูลที่ชนกันที่ช่องเดียวกัน มาผูกเป็นรายการโยงเดียวกันที่ช่องนั้น รายการโยงมีข้อเสียตรงที่มีตัวโยง ซึ่งเปลืองเนื้อที่ และเนื่องจากใช้การโยง ข้อมูลก็อาจจะโยงกันไปโยงกันมาในหน่วยความจำ การเข้าถึงข้อมูลแต่ละตัวจึงไม่ได้ใช้ความสามารถของระบบจัดการหน่วยความจำที่เรียกว่า *แคช* (cache) ที่ช่วยให้โปรแกรมทำงานเร็วขึ้น ถ้าอ่านข้อมูลจากหน่วยความจำที่อยู่ใกล้ ๆ กันบ่อย ๆ ในกรณีที่เก็บข้อมูลในแถวลำดับทั้งหมด การอ่านข้อมูลจากช่องใกล้ ๆ กัน ติด ๆ กัน บ่อย ๆ จะเร็วกว่าเมื่ออ่านจากช่องห่าง ๆ กันในแถวลำดับ รหัสที่ 11-11 เป็นโปรแกรมลองอ่านข้อมูลในแถวช่องที่ 0, m, $2m$, ... (ถ้าเกินตัวแถวก็วนกลับ) โดยทดสอบให้ $m = 1, 2, 2^2, 2^3, ..., 2^{13}$ เมื่อสั่งทำงานได้ผลดังตารางที่ 11-3 (ใช้ Java 5 บนเครื่อง Pentium M 1.3GHz) จะเห็นได้ว่า รูปแบบการอ้างอิงข้อมูลในหน่วยความจำแบบห่างกัน ช้ากว่าแบบไล่เรียงใกล้ ๆ กัน

```
public class Cache {
  public static void main(String[] args) {
    for (int i = 0; i < 14; i++) test((int) Math.pow(2, i));
  }
  static void test(int m) {
    int[]  d = new int[10000019];
    long s = System.currentTimeMillis();
    for (int k = 0; k < 10; k++)
      for (int a, j = 0, i = 0; i < d.length; i++) {
        a = d[j];
        j = (j + m) % d.length;
      }
    System.out.println(System.currentTimeMillis() - s);
  }
}
```

ให้ตารางมีขนาดเป็นจำนวนเฉพาะ d[j] ในวงวนข้างล่างนี้จะได้ไม่เคยซ้ำตัวกัน

รหัสที่ 11-11 โปรแกรมสาธิตผลกระทบของหน่วยความจำแคชต่อเวลาการทำงาน

ตารางที่ 11-3 ผลลัพธ์เมื่อสั่งโปรแกรมให้เมท็อด `test(m)` ในรหัสที่ 11-11 ทำงาน

m	1	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192
วินาที	3.9	4.0	4.0	4.0	4.8	5.6	5.6	5.6	5.8	6.1	6.7	6.7	6.9	7.0

ดังนั้นแทนที่จะใช้ตารางแฮชแบบแยกกันโยง จะขอเก็บข้อมูลต่าง ๆ ในช่องของตารางแฮชเลย เมื่อใดเกิดการชน ก็พยายามหาช่องอื่นที่ว่างในตารางเพื่อเก็บตัวที่ชน แนวคิดนี้เรียกว่า *การแฮชแบบปิด* (closed hashing) หรือบางทีเรียกว่า *การกำหนดเลขที่อยู่เปิด* (open addressing) คำถามที่ตามมาคือ

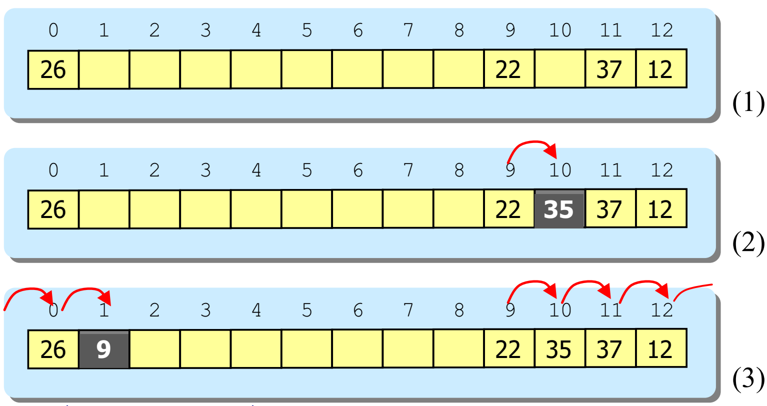
มีวิธีหาช่องอื่นที่ว่างในตารางอย่างไร ขอนำเสนอ 3 วิธีคือการตรวจเชิงเส้น (linear probing) การตรวจกำลังสอง (quadratic probing) และการแฮชสองชั้น (double hashing)

การตรวจเชิงเส้น

เมื่อ x ถูกแฮชไปที่ช่อง $h(x)$ แล้วชน ก็เพียงแต่ไป*ตรวจ* (probe) ช่องถัดไป ถ้าชนอีกก็ไปตรวจช่องถัดไปอีก ทำเช่นนี้ไปเรื่อย ๆ จนกว่าจะพบช่องว่าง (ถ้าตารางไม่เต็ม ก็ต้องพบช่องว่างแน่) ดังนั้นลำดับของเลขช่องที่ตรวจในตารางคือ $h(x)$, $h(x)+1$, $h(x)+2$, $h(x)+3$, $h(x)+4$, ... กำหนดให้ $h_j(x)$ คือเลขช่องที่ตรวจหลังการชนครั้งที่ j จะได้ว่า

$$h_j(x) = (h(x) + j) \% m$$

ซึ่งสามารถเขียน $h_j(x)$ ได้จากช่องที่ตรวจครั้งก่อน $h_{j-1}(x)$ เป็น $h_j(x) = (h_{j-1}(x) + 1) \% m$ เรียกการตรวจแบบนี้ว่า *การตรวจเชิงเส้น* (linear probing) มาลองดูกันสักตัวอย่าง เราใช้ฟังก์ชันแฮชง่าย ๆ $h(x) = x \% 13$ กับตารางแฮชขนาด 13 ช่อง แล้วเพิ่ม คีย์ 22, 37, 12, และ 26 (เพื่อความง่ายขอแสดงเฉพาะคีย์ที่เป็นจำนวนเต็ม) ข้อมูลทั้ง 4 ตัวนี้แฮชแล้วไม่ชนกันเลยได้ผลดังรูปที่ 11-6 (1) จากนั้นเพิ่ม 35 แฮชลงช่อง 9 ชนกับ 22 ก็ตรวจช่องถัดไปซึ่งว่าง จึงเพิ่มคีย์ 35 ลงช่อง 10 ดังรูป (2) ถ้าเพิ่มต่อด้วยคีย์ 9 แฮชลงช่อง 9 อีกแล้ว ต้องตรวจช่องถัดไปต่ออีก 5 ช่องถึงช่อง 1 จึงว่าง เพิ่มคีย์ 9 ได้ ดังรูป (3) เมื่อการเพิ่มมีขั้นตอนดังกล่าว การค้นหาก็อาศัยแนวคิดเดียวกัน คือนำคีย์ไปแฮชได้เลขช่องเริ่มต้น แล้วเริ่มไล่เปรียบเทียบคีย์ตามช่องต่าง ๆ ไปทีละช่อง จนกว่าจะพบ หรือจนกว่าจะพบช่องว่างซึ่งแสดงว่าหาไม่พบ



รูปที่ 11-6 ตารางแฮชที่ใช้การตรวจเชิงเส้น ใช้ฟังก์ชันแฮช $h(x) = x \% 13$

มาเริ่มเขียนคลาสให้เห็นจริง รหัสที่ 11-12 แสดงคลาส `LinearProbingHashMap` สร้างตารางแฮชที่ใช้การตรวจเชิงเส้นไว้เก็บข้อมูลแบบแมป ภายในนิยามคลาสเล็ก ๆ ชื่อ `Entry` สำหรับ

คู่ลำดับ (key,value) ของแมป มี table เป็นตารางแฮช และมี size เก็บจำนวนข้อมูล มีตัวสร้างรับขนาดของตารางเพื่อสร้างแถวลำดับ และเมท็อดมาตรฐาน size และ isEmpty

```
01 public class LinearProbingHashMap implements Map {
02   private static class Entry {
03     Object key, value;
04     Entry(Object k, Object v) {
05       key = k; value = v;
06     }
07   }
08   private Entry[] table;
09   private int size;
10
11   public LinearProbingHashMap(int m) {
12     table = new Entry[m];
13   }
14   public int size() {
15     return size;
16   }
17   public boolean isEmpty() {
18     return size == 0;
19   }
20   public boolean containsKey(Object key) {
21     return table[indexOf(key)] != null;
22   }
23   private int indexOf(Object key) {
24     int h = h(key);
25     for(int j=0; j<table.length; j++) {
26       if (table[h] == null) return h;
27       if (table[h].key.equals(key)) return h;
28       h = (h + 1) % table.length;
29     }
30     throw new AssertionError("ตารางเต็มได้ไง !");
31   }
32   private int h(Object key) {
33     return (key.hashCode() & 0x7FFFFFFF) % table.length;
34   }
```

แถวลำดับของตัวอ้างอิงไปยังข้อมูลของคลาส Entry

คืนเลขที่ช่องของ table ที่ key อยู่

นี่เองที่เรียกว่าการตรวจเชิงเส้น

รหัสที่ 11-12 คลาส LinearProbingHashMap เป็นตารางแฮชที่ใช้การตรวจเชิงเส้น

ต่อด้วยเมท็อด containsKey (บรรทัดที่ 20) ซึ่งอาศัยเมท็อด indexOf (บรรทัดที่ 23) ซึ่งรับคีย์ไปผ่านเมท็อด h เพื่อแฮชได้เลขช่องของตาราง แล้วเริ่มค้นหาในวงวน (บรรทัดที่ 25) ถ้าพบช่องว่าง (คือในช่องมีค่าเป็น null) ก็หยุดค้น ถ้าไม่ใช่ null และคีย์ที่ช่องนั้นเท่ากับคีย์ที่ต้องการ ก็หยุดค้นได้ (บรรทัดที่ 27) ถ้าไม่เท่า ให้เลื่อนไปช่องถัดไป (บรรทัดที่ 28) แล้วกลับไปค้นต่อ ให้สังเกตที่บรรทัดที่ 30 มีคำสั่ง throw ให้เกิดสิ่งผิดปกติ ที่ทำเช่นนี้ก็เพราะว่า วงวน for ในบรรทัดที่ 25 ถูกจำกัดให้ทำงานเป็นจำนวนรอบอย่างมากเท่ากับจำนวนช่องในตาราง หากหลุดจากวงวนมาถึง

บรรทัดที่ 30 แสดงว่า ได้ตรวจครบทุกช่อง แล้วไม่พบช่องว่าง และไม่พบคีย์ที่ต้องการ หากเราประกันว่า ตารางไม่เคยเต็ม เหตุการณ์นี้ก็จะไม่เกิด การทำงานของ `indexOf` ต้องจบด้วยการค้นพบคีย์ หรือไม่ก็พบ `null` จะได้แสดงให้เห็นต่อไปว่า เราจะควบคุมไม่ให้เก็บข้อมูลจนเต็มตาราง

เมท็อด `indexOf` ถูกใช้เป็นกลไกหลักในการ `get`, `put`, และ `remove` ด้วย รหัสที่ 11-13 แสดงเมท็อด `get` ซึ่งใช้ `indexOf` กับคีย์ที่ได้รับ ถ้าช่องที่พบเป็น `null` ก็แสดงว่า ไม่พบคีย์ที่ต้องการ ก็คืน `null` แต่ถ้าช่องนั้นไม่ใช่ `null` ก็ต้องเป็นตัวที่ต้องการ ให้คืน `value` ของข้อมูลในช่องนั้น สำหรับเมท็อด `put` ก็เช่นกันใช้ `indexOf` กับคีย์ที่ได้รับ ถ้าช่องที่พบเป็น `null` แสดงว่าต้องเพิ่ม `(key,value)` ไว้ที่ช่องนั้น (บรรทัดที่ 43) แล้วเพิ่ม `size` อีกหนึ่ง แต่ถ้าช่องที่พบไม่ใช่ `null` แสดงว่า มีคีย์นี้อยู่ ก็ให้เปลี่ยน `value` ของข้อมูลนั้นเป็น `value` ตัวใหม่ แต่ก็ต้องอย่าลืมจำ `value` ของเก่าไว้คืนเป็นผลลัพธ์ของ `put` ด้วย

```
35   public Object get(Object key) {
36     Entry e = table[indexOf(key)];
37     return e == null ? null : e.value;
38   }
39   public Object put(Object key, Object value) {
40     Object oldValue = null;
41     int i = indexOf(key);
42     if (table[i] == null) {
43       table[i] = new Entry(key, value);
44       ++size;
45     } else {
46       oldValue = table[i].value;
47       table[i].value = value;
48     }
49     return oldValue;
50   }
```

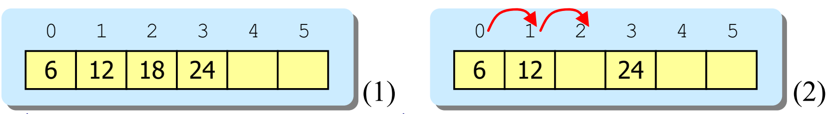
หา key ในตาราง ถ้าพบให้คืน value หาไม่พบ คืน null

หาไม่พบ สร้างและเพิ่ม Entry ใหม่

หาพบ เปลี่ยนเป็น value ใหม่

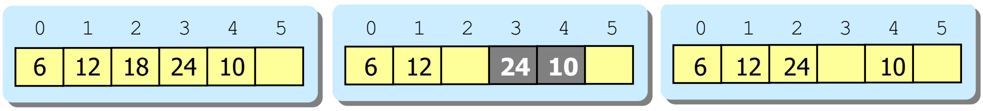
รหัสที่ 11-13 เมท็อด `get` และ `put` ของคลาส `LinearProbingHashMap`

สำหรับการลบจะซับซ้อนเล็กน้อย การลบข้อมูลที่พบในช่องที่ i จะทำเพียงแค่นำ `null` ไปใส่ในช่องนั้นไม่ได้ เพราะคีย์ที่ถูกเพิ่มก่อนหน้านี้ ที่เคยต้องตรวจผ่านช่อง i จะหาไม่พบในอนาคต ตัวอย่างรูปที่ 11-7 (1) แสดงตารางแฮชเก็บคีย์สี่ตัวที่ล้วนแฮชผ่าน $h(x) = x \% 6$ ลงไปที่ช่อง 0 หมด ถ้าเราลบ 18 ทิ้ง โดยใส่ `null` ในช่อง 2 จะทำให้ค้นคีย์ 24 ไม่พบ ดังรูป (2)



รูปที่ 11-7 การลบ 18 ออก โดยทำให้ช่องที่ 2 ว่าง ยังไม่พอ (ฟังก์ชันแฮช $h(x) = x \% 6$)

ข้อมูลตัวที่อาจได้รับผลกระทบจากการลบช่องที่ i คือตั้งแต่ช่องที่ i+1 ไปเรื่อย ๆ จนถึงช่อง `null` ถัดไป ดังนั้นต้องนำข้อมูลที่อาจได้รับผลกระทบเหล่านี้มาแฮชใหม่ซึ่งทำได้โดยการลบข้อมูลเหล่านั้นชั่วคราวแล้วเพิ่มกลับเข้าไปในตาราง เช่น ในรูปที่ 11-8 (1) แสดงตารางที่ได้จากการเพิ่ม 10, 6, 12, 18, และ 24 ด้วย $h(x) = x$ % 6 ถ้าต้องลบ 18 ก็หา 18 จนพบในช่องที่ 2, ทำให้ช่องนี้ว่าง แล้วนำ 24 และ 10 มาแฮชใหม่ (เราหยุดที่คีย์ 10 เพราะช่องถัดไปเป็นช่องว่าง) คีย์ 24 ถูกแฮชใหม่ลงที่ช่อง 0 แล้วตรวจไปจนพบช่องว่างที่ช่อง 2 ส่วนคีย์ 10 ถูกแฮชใหม่ลงที่ช่อง 4 ซึ่งก็คือช่องเดิม ได้ดังรูป (3) รหัสที่ 11-14 แสดงรายละเอียดการทำงานของ `remove`



(1) (2) (3)

รูปที่ 11-8 การลบ 18 ต้องแฮช 24 และ 10 ใหม่ (ฟังก์ชันแฮช $h(x) = x$ % 6)

```
  public void remove(Object key) {
    int i = indexOf(key);
    if (table[i] != null) {
      table[i] = null;
      --size;
      for(++i; table[i]!=null; i=(i+1)%table.length) {
        Entry e = table[i];
        table[i] = null;
        table[indexOf(e.key)] = e;
      }
    }
  }
```

ลบคือการใส่ null ในช่องนั้น ตามด้วยการแฮชข้อมูลที่เกาะกลุ่มกันใหม่

ลบออก

ถ้าตารางกระจายดี เกิดการแฮชใหม่ตรงนี้ไม่มาก

แล้วเพิ่มกลับเข้าไปใหม่

รหัสที่ 11-14 เมท็อด `remove` ของคลาส `LinearProbingHashMap`

หลายคนอาจกังวลว่า การลบโดยต้องนำชุดข้อมูลที่ติดกันหลังตัวที่ถูกลบมาแฮชใหม่นั้นคงช้า อีกทั้งการค้นหาที่ไล่เปรียบเทียบจากตำแหน่งแรกที่แฮชจนกว่าจะพบคีย์ที่ต้องการ หรือพบ `null` นั้นก็น่าจะช้าเหมือนกัน ตารางแฮชแบบนี้ก็ไม่น่าจะเร็วอย่างที่โฆษณา ถ้าข้อมูลในตารางมีสภาพเกาะเป็น*กลุ่ม* (cluster) คือมีชุดข้อมูลติดกันยาว ๆ ในตารางโดยไม่มี `null` คั่น สมมติว่า ในตารางแฮชมีการเกาะกลุ่มของข้อมูลจำนวน t ตัว ตั้งแต่ช่องที่ i ถึงช่องที่ $i+t-1$ (แสดงว่า ช่องที่ $i-1$ และช่องที่ $i+t$ เป็น `null`) กำหนดให้คีย์ x ไม่อยู่ในกลุ่มข้อมูลนี้ ถ้า x ถูกแฮชแล้วตกในช่องที่ j โดยที่ $i \le j \le i+t-1$ การตรวจเชิงเส้นย่อมจบที่ช่องที่ $i+t$ (ที่เป็น `null`) โดยต้องตรวจเป็นจำนวน $i + t - j + 1$ ช่อง หากเราต้องการคำนวณผลรวมจำนวนช่องในตารางที่ต้องตรวจเมื่อค้นหาข้อมูลแล้วไม่พบ จะได้ว่า การเกาะกลุ่มขนาด t ช่อง จะเพิ่มผลรวมที่ต้องการเป็นจำนวน $\sum_{j=i}^{i+t-1}(i+t-j+1) = t + t(t+1)/2$ ถ้าเรารู้ว่าในตารางขนาด m เก็บข้อมูลจำนวน n ตัว มีการเกาะกลุ่มขนาด $t_1, t_2, ..., t_k$ จะได้ว่า $t_1 + t_2 + ... + t_k = n$

(เพราะผลรวมขนาดของทุกกลุ่มต้องเท่ากับจำนวนข้อมูล) และมีอยู่ $m–n$ ช่องที่เป็น `null` (การค้นหาคีย์ที่แฮชมาที่ช่อง `null` เหล่านี้ ก็ต้องตรวจ 1 ช่อง) ดังนั้นจำนวนช่องเฉลี่ยในตารางที่ต้องตรวจเมื่อค้นหาคีย์ไม่พบ เท่ากับ

$$
\begin{aligned}
u_{n,m} &= \frac{1}{m}\left((m-n)+\sum_{i=1}^{k}\left(t_i+\frac{t_i(t_i+1)}{2}\right)\right) = \frac{m-n}{m}+\frac{1}{m}\sum_{i=1}^{k}t_i+\frac{1}{2m}\sum_{i=1}^{k}t_i(t_i+1)\\
&= \frac{m-n}{m}+\frac{n}{m}+\frac{1}{2m}\sum_{i=1}^{k}t_i(t_i+1)\\
&= 1+\frac{1}{2m}\sum_{i=1}^{k}t_i(t_i+1)
\end{aligned}
$$

ผู้เขียนลองเขียนโปรแกรมสร้างตารางแฮช แล้วนำข้อมูลสุ่มมาเพิ่มในตาราง วัดขนาดของการเกาะกลุ่ม แล้วนำมาเข้าสูตรข้างบนนี้ ได้ผลดังตารางที่ 11-4 ให้สังเกตว่า ผลที่ได้ขึ้นกับสัดส่วน $\lambda = n/m$ ซึ่งสะท้อนความ "แน่น" ของตาราง เราเรียกสัดส่วนนี้ว่า *สัดส่วนบรรจุ* (load factor) ได้มีผู้วิเคราะห์ว่าจำนวนช่องเฉลี่ยที่ต้องตรวจเมื่อค้นหาคีย์กรณีพบ และกรณีไม่พบ เมื่อใช้การตรวจเชิงเส้นคือ

$$\frac{1}{2}\left(1+\frac{1}{1-\lambda}\right) \quad \text{และ} \quad \frac{1}{2}\left(1+\frac{1}{(1-\lambda)^2}\right) \text{ ตามลำดับ}$$

ตารางที่ 11-4 $u_{n,m}$ = จำนวนช่องเฉลี่ยที่ต้องตรวจเมื่อค้นหาคีย์ไม่พบ ในตารางขนาด m มีข้อมูล n

	m = 10,000		m = 100,000		m = 1,000,000	
	n	$u_{n,m}$	n	$u_{n,m}$	n	$u_{n,m}$
λ = 0.3	3,000	1.51	30,000	1.52	300,000	1.52
λ = 0.4	4,000	1.83	40,000	1.90	400,000	1.89
λ = 0.5	5,000	2.50	50,000	2.52	500,000	2.50
λ = 0.6	6,000	3.42	60,000	3.65	600,000	3.63
λ = 0.7	7,000	6.08	70,000	6.22	700,000	6.02

ดังนั้นการใช้ตารางแฮชจึงควรควบคุมสัดส่วนบรรจุ เพื่อควบคุมเวลาการทำงาน เช่น หากเดิมเราตั้ง $\lambda = 0.75$ หมายความว่า ถ้าต้องการเก็บข้อมูล 7,500 ตัว ก็สร้างตารางแฮชขนาด 10,000 ช่อง แต่ถ้ามาพบภายหลังว่า การทำงานของระบบเกิดคอขวดในการทำงานของตารางแฮช หากปรับสัดส่วนบรรจุให้น้อยลงเป็น $\lambda = 0.5$ ก็ต้องสร้างตารางใหญ่ขึ้นเป็น 15,000 ช่อง เห็นได้ชัดว่า เปลืองเนื้อที่มากขึ้น แต่จะทำให้การทำงานเร็วขึ้น เป็นปรากฏการณ์ใหม่ในการนำเนื้อที่หน่วยความจำแลกกับเวลาการทำงาน ซึ่งไม่มีในโครงสร้างข้อมูลประเภทอื่น

และในกรณีที่เราไม่ทราบขนาดของข้อมูลที่แน่ชัด ก็ควรกำหนดสัดส่วนบรรจุที่ต้องการ แล้วคอยควบคุมไม่ให้เกิน โดยต้องปรับเมท็อด `put` ให้คอยตรวจสอบ ถ้าเกินระดับที่ตั้งไว้ก็ให้สร้างตารางใหม่ให้ใหญ่กว่าเดิม แล้วนำข้อมูลทุกตัวในตารางเก่าแฮชใส่ตารางใหม่ ด้วยเมท็อด `rehash` ดังแสดงในรหัสที่ 11-15

```
  public Object put(Object key, Object value) {
    Object oldValue = null;
    int i = indexOf(key);
    if (table[i] == null) {
      table[i] = new Entry(key, value);
      ++size;
      if (size > table.length/2) rehash();
    } else {
      oldValue = table[i].value;
      table[i].value = value;
    }
    return oldValue;
  }
  private void rehash() {
    Entry[] oldT = table;
    table = new Entry[2 * table.length];
    for (int i = 0; i < oldT.length; i++) {
      if (oldT[i] != null) table[indexOf(oldT[i].key)] = oldT[i];
    }
  }
```

เพิ่มแล้วมีข้อมูลเกินครึ่งตาราง ก็ rehash

ขยายตาราง หลัง rehash สัดส่วนบรรจุเหลือ 0.25

แฮช Entry จากตารางเก่าไปใส่ในตารางใหม่

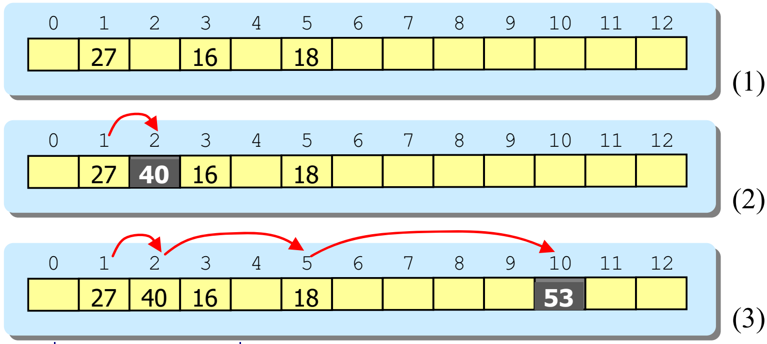
รหัสที่ 11-15 เมท็อด `put` ที่ขยายขนาดตารางถ้าสัดส่วนบรรจุมีขนาดมากเกิน 0.5

การตรวจกำลังสอง

ด้วยลักษณะการไล่ตรวจไปทีละช่องของการตรวจเชิงเส้น ทำให้ข้อมูลเกาะกลุ่มกันง่าย กลุ่มใดที่เกาะกันยาว ก็มีโอกาสจะยาวขึ้นได้มากกว่ากลุ่มที่สั้นกว่า เพราะคีย์ใหม่ย่อมมีโอกาสมากกว่าที่จะถูกแฮชลงในบริเวณของกลุ่มยาว ซึ่งจะต้องตรวจเชิงเส้นไปตกใส่ช่อง `null` ที่ปิดท้ายกลุ่ม ทำให้กลุ่มนี้ยาวขึ้นไปอีก เรียกว่า *การเกาะกลุ่มปฐมภูมิ* (primary clustering) *การตรวจกำลังสอง* (quadratic probing) ถูกออกแบบให้ตรวจช่อง $h(x)$, $h(x)+1^2$, $h(x)+2^2$, $h(x)+3^2$, ...ในระยะที่ห่างเพิ่มขึ้น ๆ เพื่อลดการเกาะกลุ่ม โดยมีการคำนวณช่องที่ตรวจดังนี้

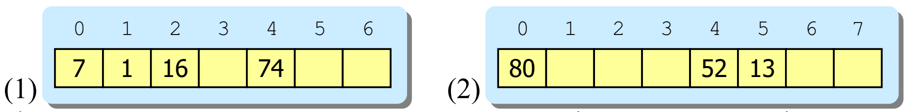
$$h_j(x) = (h(x) + j^2)\ \%\ m$$

(ทวนความจำเล็กน้อยว่า $h_j(x)$ คือเลขช่องที่ตรวจหลังการชนครั้งที่ j) แทน j ด้วย $j-1$ จะได้ $h_{j-1}(x) = (h(x) + (j-1)^2)\ \%\ m$ นำไปลบกับสมการข้างบนนี้จะได้ $h_j(x) = (h_{j-1}(x) + 2j - 1)\ \%\ m$ มาลองดูกันสักตัวอย่าง เราใช้ฟังก์ชันแฮชง่าย ๆ $h(x) = x\ \%\ 13$ กับตารางแฮชขนาด 13 ช่อง แล้วเพิ่ม คีย์ 27, 16 และ 18 ข้อมูลทั้ง 3 ตัวนี้แฮชแล้วไม่ชนกันเลยได้ผลดังรูปที่ 11-9 (1) จากนั้นเพิ่ม 40 ซึ่งแฮชลงช่อง 1 ชนกับ 27 ก็ตรวจช่อง $1 + 1^2=2$ พบว่าว่าง จึงเพิ่มคีย์ 40 ลงช่อง 2 ดังรูป (2) ตามด้วยการเพิ่มคีย์ 53 แฮชลงช่อง 1 อีกแล้ว พบว่า ต้องตรวจช่อง 1, 2, 5, และ 10 จึงพบช่องว่าง เพิ่มคีย์ 53 ได้ ดังรูป (3) เห็นได้ว่า การตรวจกำลังสองนี้จะตรวจช่องในระยะที่ห่างเพิ่มขึ้น ๆ ทั้งนี้เพื่อต้องการลดการเกาะกลุ่มของข้อมูล



รูปที่ 11-9 ตารางแฮชที่ใช้การตรวจกำลังสอง ใช้ฟังก์ชันแฮช $h(x) = x \% 13$

การเพิ่มในตารางแฮชที่ใช้การตรวจกำลังสองมีเรื่องแปลก ๆ ที่ต้องคำนึงถึง สมมติว่า ตารางมีขนาด 7 ช่อง ถ้าแฮช x ลงช่องที่ 0, จะได้ลำดับการตรวจกำลังสองคือ $(0+j^2)\%7, j = 0,1,2,3,...$ ซึ่งคือช่องที่ 0, 1, 4, 2, 2, 4, 1, 0, 1, 4, 2, ... ซ้ำกันไปมาอยู่แค่ 4 ช่อง ถ้าตารางขนาด 7 ช่องเก็บข้อมูลในช่องที่ 0, 1, 2, และ 4 เหลือช่องว่าง 3 ช่อง ดังตัวอย่างในรูปที่ 11-10 (1) แล้วต้องการเพิ่ม 70 ซึ่งถูกแฮช ลงช่อง 0 จะค้นไม่พบช่องว่างทั้ง ๆ ที่ยังมีเหลืออยู่ และก็ไม่จำเป็นว่า ถ้าตารางขนาดใหญ่ขึ้น จะตรวจช่องจำนวนมากขึ้น เช่น ให้ตารางมีขนาด 8 ช่อง ถ้าแฮชลงช่อง 4 จะได้ลำดับการตรวจคือ $(4 + j^2) \% 8, j = 0,1,2,3,...$ ซึ่งคือช่องที่ 4, 5, 0, 5, 4, 5, 0, 5, 4, ...ซ้ำกันไปมาเพียง 3 ช่อง ดังตัวอย่างในรูปที่ 11-10 (2) แล้วต้องการเพิ่ม 12 ที่แฮชลงช่องที่ 4 ย่อมหาช่องว่างไม่พบ

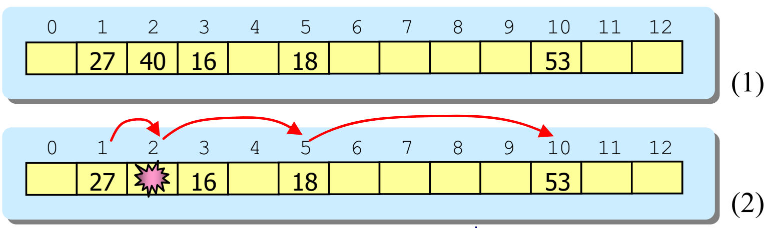


รูปที่ 11-10 การตรวจกำลังสองไม่ได้ตรวจครบทุกช่อง ลองเพิ่ม 0 ในรูป (1) และเพิ่ม 4 ในรูป (2)

อย่างไรก็ตาม ถ้าตารางมีขนาด m โดยที่ m เป็นจำนวนเฉพาะ เราสามารถพิสูจน์ได้ว่า การตรวจครั้งที่ 1 จนถึงครั้งที่ $1+\lfloor m/2 \rfloor$ จะไม่ซ้ำช่องกันเลย นั่นคือช่องที่ $(h(x)+j^2) \% m$ เมื่อ $j = 0, 1, 2, ..., \lfloor m/2 \rfloor$ เป็นช่องที่ต่างกันหมด ดังนั้นถ้าขนาดของตารางเป็นจำนวนเฉพาะ และมีสัดส่วนบรรจุไม่เกินครึ่ง (นั่นคือมีช่องที่ว่างเกินครึ่ง) การตรวจกำลังสองจะหาช่องว่างพบเสมอ ซึ่งสามารถพิสูจน์ด้วยข้อขัดแย้งดังนี้ สมมติว่า มี i และ j โดยที่ $0 \le i < j \le \lfloor m/2 \rfloor$ ที่ทำให้ $(h(x) + i^2) \% m = (h(x) + j^2) \% m$ นั่นคือ $(h(x) + i^2) \equiv (h(x) + j^2) \pmod{m}$, ตัด $h(x)$ ออกทั้งสองข้างได้ $i^2 \equiv j^2 \pmod{m}$, ย้ายข้างจัดรูปแบบได้เป็น $(i-j)(i+j) \equiv 0 \pmod{m}$ ซึ่งจะเป็นจริงได้เมื่อ $(i-j)$ เป็น 0, หรือ $(i+j)$ เป็น m, หรือว่า ผลคูณของ $(i-j)(i+j)$ เป็น m แต่เนื่องจาก $0 \le i < j \le \lfloor m/2 \rfloor$, $(i-j)$ ไม่เป็น 0, และ $(i+j)$ บวกกันก็ไม่ถึง m, และเนื่องจาก m เป็นจำนวนเฉพาะ จึงแยกเป็นตัวประกอบ $(i-j)(i+j)$ ไม่ได้ สรุปได้ว่า $(h(x)+j^2) \% m$ เมื่อ $j = 0, 1, 2, ..., \lfloor m/2 \rfloor$ มีค่าไม่ซ้ำช่องกันเลย เมื่อ m เป็นจำนวนเฉพาะ

ขอสรุปตอนนี้ก่อนจะลืมว่า การตรวจกำลังสองช่วยลดการเกาะกลุ่มกันของข้อมูล แต่มีเงื่อนไขว่า ต้องสร้างตารางให้มีขนาดเป็นจำนวนเฉพาะ และรักษาสัดส่วนบรรจุไว้อย่าให้เกิน 0.5 จะได้มั่นใจว่า สามารถหาช่องว่างเพื่อเพิ่มข้อมูลได้สำเร็จแน่ ๆ อีกสิ่งหนึ่งที่ต้องระวังคือการลบข้อมูล เราจะลบโดยใช้วิธีที่ได้นำเสนอสำหรับการตรวจเชิงเส้นไม่ได้ ถ้ายังจำได้ ในการตรวจเชิงเส้น การลบข้อมูลที่ช่อง i ทำได้ด้วยการใส่ `null` ที่ช่องนั้น แล้วนำข้อมูลตั้งแต่ช่องที่ i+1 จนถึงช่องก่อนช่อง `null` ถัดไปมาแฮชใหม่ วิธีนี้ใช้ไม่ได้กับการตรวจกำลังสอง ก็เพราะลำดับการตรวจไม่ได้ทำช่องติด ๆ กันเหมือนในการตรวจเชิงเส้น จึงต้องหากระบวนการลบวิธีอื่น

ขอนำเสนอกลวิธีการลบที่ใช้ได้ทั้งกับการตรวจเชิงเส้น การตรวจกำลังสอง และการแฮชสองชั้น (ที่จะนำเสนอในหัวข้อต่อไป) แทนที่จะนำค่า `null` ไปใส่ในช่องที่จะลบ เพื่อให้มีสภาพเป็นช่องว่าง (ซึ่งสร้างปัญหากับการค้นในอนาคต เพราะขั้นตอนการตรวจค้นจะหยุดเมื่อพบช่องว่าง) เราจะนำอ็อบเจกต์พิเศษ (ให้ชื่อว่า `DELETED`) ไปใส่แทน โดยมีคีย์ที่ไม่เหมือนใคร (เมื่อเปรียบเทียบกับคีย์อื่น จะไม่เท่าแน่ ๆ) ทำให้การตรวจใน `indexOf` ไม่หยุดที่ช่องเหล่านี้แน่ ๆ เพราะไม่ใช่ `null` และก็มีคีย์ที่ไม่เท่ากับที่ต้องการหา รูปที่ 11-11 แสดงการลบ 40 ออกจากตารางในรูปบน โดยการใส่ `DELETED` ไว้ในช่อง 2 ที่เก็บ 40 ทำให้ไม่ขัดจังหวะการตรวจค้นข้อมูลในอนาคต เช่น การค้นหา 53 จะยังคงทำได้ ไม่ไปหยุดที่ช่อง 2 ดังแสดงในรูปล่าง



รูปที่ 11-11 การลบข้อมูลโดยใส่อ็อบเจกต์พิเศษ ในช่องที่ถูกลบไม่ขัดจังหวะการตรวจในอนาคต

การลบทำนองนี้เข้าข่ายกลวิธีการลบที่เรียกว่า *การลบแบบเกียจคร้าน* (lazy deletion) คือตอนลบก็ไม่ย่อมลบจริง แต่ทิ้งสภาวะบางอย่างค้าง ๆ ไว้เสมือนเป็นขยะภายใน ทั้งเปลืองที่ ทั้งเสียเวลาการตรวจด้วย ซึ่งเราต้องนับเป็นส่วนหนึ่งในการคำนวณสัดส่วนบรรจุ เมื่อถึงคราวที่สัดส่วนบรรจุเกินครึ่งต้องแฮชใหม่ทั้งหมด ก็จะเป็นโอกาสดีในการล้าง `DELETED` ให้ออกไปจากตาราง

ดังนั้นจะขอเขียนคลาส `QuadraticProbingHashMap` ใหม่เลย สิ่งที่เหมือนกับคลาส `LinearProbingHashMap` คือมีคลาสภายในชื่อ `Entry`, แถวลำดับ `table`, ตัวแปร `size`, เมท็อด `size`, `isEmpty`, `containsKey`, `get`, และ `h` ที่เหมือนกัน ที่เพิ่มเติมและเปลี่ยนแปลงคือมีอ็อบเจกต์ของ `Entry` ชื่อ `DELETED` มีคีย์เป็นอ็อบเจกต์ใหม่หนึ่งตัว (คีย์นี้สร้างจาก `Object`

เลย จะได้เปรียบเทียบแล้วไม่เหมือนใคร) ให้ DELETED เป็นแบบ static ด้วย จะได้ใช้ร่วมกันได้ มีตัวแปรชื่อ numNonNulls ไว้จำจำนวนช่องในตารางที่ไม่ใช่ null ซึ่งคือจำนวนข้อมูลจริง ๆ กับจำนวนช่องที่เก็บ DELETED โดยเราจะใช้จำนวนนี้คำนวณสัดส่วนบรรจุของตารางว่า สมควรแฮชตารางใหม่หรือไม่ นอกจากนี้มีตัวสร้างรับ m ซึ่งคือขนาดของตาราง แต่จะไม่สร้างแถวลำดับตามขนาดที่ได้รับ เพราะเราต้องการขนาดของตารางที่เป็นจำนวนเฉพาะ ดังนั้นจึงต้องหาจำนวนเฉพาะที่มีค่าถัดจาก m (ตรงนี้ใช้บริการ nextProbablePrime ของคลาส BigInteger ในจาวา เช่น ให้ m = 10000 จะได้ 10007 เป็นจำนวนเฉพาะ) สรุปช่วงแรกนี้ก่อนได้ดังรหัสที่ 11-16

```
public class QuadraticProbingHashMap implements Map {
  private static class Entry { ... }
  private static final Entry DELETED=new Entry(new Object(),null);

  private Entry[] table;
  private int size;

  private int numNonNulls;

  public QuadraticProbingHashMap(int m) {
    table = new Entry[nextPrime(m)];
  }
  private int nextPrime(int m) {
    BigInteger b = new BigInteger(Integer.toString(m));
    return b.nextProbablePrime().intValue();
  }
  public int size() { ... }
  public boolean isEmpty() { ... }
  public boolean containsKey(Object key) { ... }
  public Object get(Object key) { ... }
  private int h(Object key) { ... }
```

มีไว้ใส่ในช่องที่ถูกลบ

คีย์นี้ไม่เท่ากับใครแน่ ๆ

เก็บจำนวนช่องในตารางที่ไม่เท่ากับ null

คืนจำนวนเฉพาะตัวถัดจาก m

เมท็อดเหล่านี้เหมือนกับของ LinearProbingHashMap

รหัสที่ 11-16 ส่วนต้นของคลาส QuadraticProbingHashMap

รหัสที่ 11-17 แสดงเมท็อด indexOf ซึ่งเป็นตัวจักรหลักในการตรวจค้นหาช่องที่เป็น null หรือช่องที่เก็บคีย์ที่ต้องการ โดยมีการปรับระยะก้าวกระโดดตามสูตร $h_j(x) = (h_{j-1}(x) + 2j - 1)\ \%\ m$ ของการตรวจกำลังสอง สำหรับเมท็อด remove ทำได้ง่าย ๆ ด้วยการเติมเพียงแค่ DELETED ในช่องที่ค้นพบนั้น เมท็อด put ต่างจากที่นำเสนอมาเพียงแค่มีการเพิ่มค่าของ numNonNulls ทุกครั้งที่มีการเพิ่มข้อมูลใหม่ ให้สังเกตว่า numNonNulls ไม่ลดตอนที่เราลบข้อมูล หลังการเพิ่มข้อมูล ถ้า numNonNulls มีค่าเกินครึ่งหนึ่งของตาราง ก็ต้องแฮชใหม่ทั้งตารางด้วยการเรียก rehash ซึ่งทำเหมือนเดิม ต่างกันแค่ขนาดของตารางที่จอง ให้จองเป็น 4 เท่าของจำนวนข้อมูล เพื่อให้หลัง rehash สัดส่วนบรรจุจริงของตารางมีค่าเป็น 0.25 และที่จะลืมไม่ได้ก็คือต้องจองขนาดให้เป็นจำนวนเฉพาะด้วย

```
private int indexOf(Object key) {
  int h = h(key);
  for (int j = 1; j < table.length; j++) {
    if (table[h] == null) break;
    if (table[h].key.equals(key)) break;
    h = (h + 2 * j - 1) % table.length;
  }                           นี่คือการตรวจกำลังสอง
  return h;
}
public void remove(Object key) {
  int i = indexOf(key);
  if (table[i] != null) {
    table[i] = DELETED;     นำ DELETED ใส่ช่องที่ถูกลบ
    --size;
  }
}
public Object put(Object key, Object value) {
  Object oldValue = null;
  int i = indexOf(key);
  if (table[i] == null) {
    table[i] = new Entry(key, value);
    ++size; ++numNonNulls;
    if (numNonNulls > table.length/2)  rehash();
  } else {
    oldValue = table[i].value;    rehash ถ้าช่องว่างมีน้อยกว่าครึ่งของตาราง
    table[i].value = value;
  }
  return oldValue;
}
private void rehash() {                    หลัง rehash สัดส่วนบรรจุเป็น 0.25
  Entry[] oldT = table;
  table = new Entry[nextPrime(4 * size)];
  for (int i = 0; i < oldT.length; i++) {
    if (oldT[i] != null && oldT[i] != DELETED) {
      int j = indexOf(oldT[i].key);
      table[indexOf(oldT[i].key)] = oldT[i];
    }
  }
  numNonNulls = size;
}
```

รหัสที่ 11-17 เมท็อดของ QuadraticProbingHashMap ที่ต่างจากของ LinearProbingHashMap

ให้สังเกตว่า เมท็อด `put` ในรหัสที่ 11-17 ไม่ได้นำช่องที่เป็น `DELETED` มาใช้ใหม่เลย ซึ่งออกจะสิ้นเปลืองมาก ขอละไว้เป็นแบบฝึกหัดให้ผู้อ่านปรับปรุง ซึ่งจะทำให้เกิดการ `rehash` น้อยลง และมีประสิทธิภาพโดยรวมดีขึ้น

การแฮชสองชั้น

การตรวจกำลังสองช่วยแก้ปัญหาการเกาะกลุ่มปฐมภูมิที่เกิดในการตรวจเชิงเส้น แต่กลุ่มข้อมูลที่มี $h_0(x)$ เหมือนกันจะยังคงมีลำดับการตรวจ $h_1(x)$, $h_2(x)$, $h_3(x)$, ... เหมือนกัน สภาพเช่นนี้เรียกว่า *การเกาะกลุ่มทุติยภูมิ* (secondary clustering) นั่นเป็นเพราะระยะก้าวกระโดดของการตรวจมีค่าขึ้นกับว่าเป็นการชนครั้งที่เท่าใดในลำดับการตรวจ *การแฮชสองชั้น* (double hashing) เป็นกลวิธีที่นำตัวคีย์มากำหนดระยะก้าวกระโดด ดังนั้นกลุ่มข้อมูลที่มี $h_0(x)$ เหมือนกัน ก็อาจมีระยะก้าวกระโดดแตกต่างกันได้ เป็นการขจัดสภาพการเกาะกลุ่มทุติยภูมิ การแฮชสองชั้นคำนวณช่องที่ตรวจดังนี้

$$h_j(x) = (h(x) + j{\cdot}g(x))\ \%\ m$$

หรือเขียนได้ว่า $h_j(x) = (h_{j-1}(x) + g(x))\ \%\ m$ โดย $g(x)$ ก็คือฟังก์ชันแฮชอีกตัวหนึ่งที่มีไว้เพื่อกำหนดระยะก้าวกระโดด ซึ่งหมายความว่า ระยะก้าวกระโดดของแต่ละคีย์จะเป็นเช่นใดนั้น ยากต่อการคาดเดา (เนื่องจากใช้แนวคิดเดียวกับฟังก์ชันแฮช) รหัสที่ 11-18 แสดงเมท็อด `indexOf` ซึ่งเปลี่ยนให้มีการตรวจแบบการแฮชสองชั้น และแสดงตัวอย่างเมท็อด $g(x)$

```
private int indexOf(Object key) {
  int h = h(key);
  int g = g(key);
  for (int j = 1; j < table.length; j++) {
    if (table[h] == null) break;
    if (table[h].key.equals(key)) break;
    h = (h + g) % table.length;
  }
  return h;
}
private int g(Object key) {
  return 1 + (key.hashCode() & 0x7FFFFFFF) % (table.length / 2);
}
```

ตัวอย่างฟังก์ชันแฮชอีกตัวที่คำนวณระยะกระโดด

รหัสที่ 11-18 ลำดับการตรวจของการแฮชสองชั้น

อย่าลืมว่า $g(x)$ ต้องให้ค่าที่ $g(x)\ \%\ m \neq 0$ เพราะไม่เช่นนั้นการตรวจจะย่ำอยู่กับที่ นอกจากนี้ $g(x)$ ต้องไม่มีตัวหารร่วมกับ m (ยกเว้น 1) เพื่อให้สามารถตรวจได้ครบทุกช่อง เช่น ถ้า $m = 12$, $g(x)$ ได้ผลเป็น 9, $h(x)$ ได้ผลเป็น 3, ลำดับการตรวจคือ 3, (3+9)%12, (3+18)%12, (3+27) %12 ... จะตรวจซ้ำช่องที่ 0, 9, 6, และ 3 แต่ถ้าเปลี่ยน $g(x)$ เป็น 5 หรือ 7 หรือ 11 หรือ 13 จะตรวจครบทุกช่อง เนื่องจากเราต้องการให้ $g(x)$ มีค่าหลากหลาย ดังนั้นวิธีประกันว่า m และ $g(x)$ ไม่มีตัวหารร่วมกันที่ง่ายสุดคือการให้ m เป็นจำนวนเฉพาะ ก็จะสบายใจได้ว่า การแฮชสองชั้นจะตรวจครบทุกช่อง

ขอไม่แสดงรายละเอียดของคลาส `DoubleHashingHashMap` เพราะเหมือนกับของคลาส `QuadraticProbingHashMap` ทุกประการ ต่างกันแค่ `indexOf`, `g`, และตัวสร้าง

ตารางที่ 11-5 แสดงการเปรียบเทียบจำนวนช่องเฉลี่ยที่ตรวจระหว่างการใช้การตรวจเชิงเส้น การตรวจกำลังสอง และการแฮชสองชั้น ทั้งในกรณีค้นหาพบ และกรณีค้นหาไม่พบ โดยปรับสัดส่วนบรรจุของตารางตั้งแต่ 0.3 จนถึง 0.9 (ผู้อ่านอาจงง ก็เพิ่งบอกว่า สัดส่วนบรรจุของการตรวจกำลังสองต้องไม่เกิน 0.5 เพราะถ้าเกิน 0.5 แล้วมีช่องว่าง อาจหาไม่พบ ขอเน้นว่า <u>อาจหาไม่พบ</u> แต่ในการทดลองที่ผู้เขียนได้ทำมานี้ ไม่เกิดเหตุการณ์ดังกล่าว) ตารางที่ใช้ในการทดลองมีขนาด 1,000,003 ช่อง ไม่มีการตรวจสอบสัดส่วนบรรจุเพื่อขยายตารางเพราะจองตารางให้ใหญ่พอ คีย์เป็นจำนวนสุ่มแบบ `double` ทดลอง 10 ครั้งเพื่อหาผลเฉลี่ย (การทดลองนี้มีแต่การเพิ่มข้อมูล ไม่มีการลบ) จะเห็นได้ว่าการตรวจเชิงเส้นตรวจช่องเป็นจำนวนมากสุด ในขณะที่ของการตรวจกำลังสองและการแฮชสองชั้นนั้นตรวจเป็นจำนวนพอ ๆ กัน ผลการทดลองตอกย้ำว่า เราควรควบคุมสัดส่วนบรรจุดอย่าให้มีค่ามาก และถ้าฟังก์ชันแฮชกระจายดี ตารางแฮชจะให้ประสิทธิภาพการทำงานที่ดีมาก ๆ

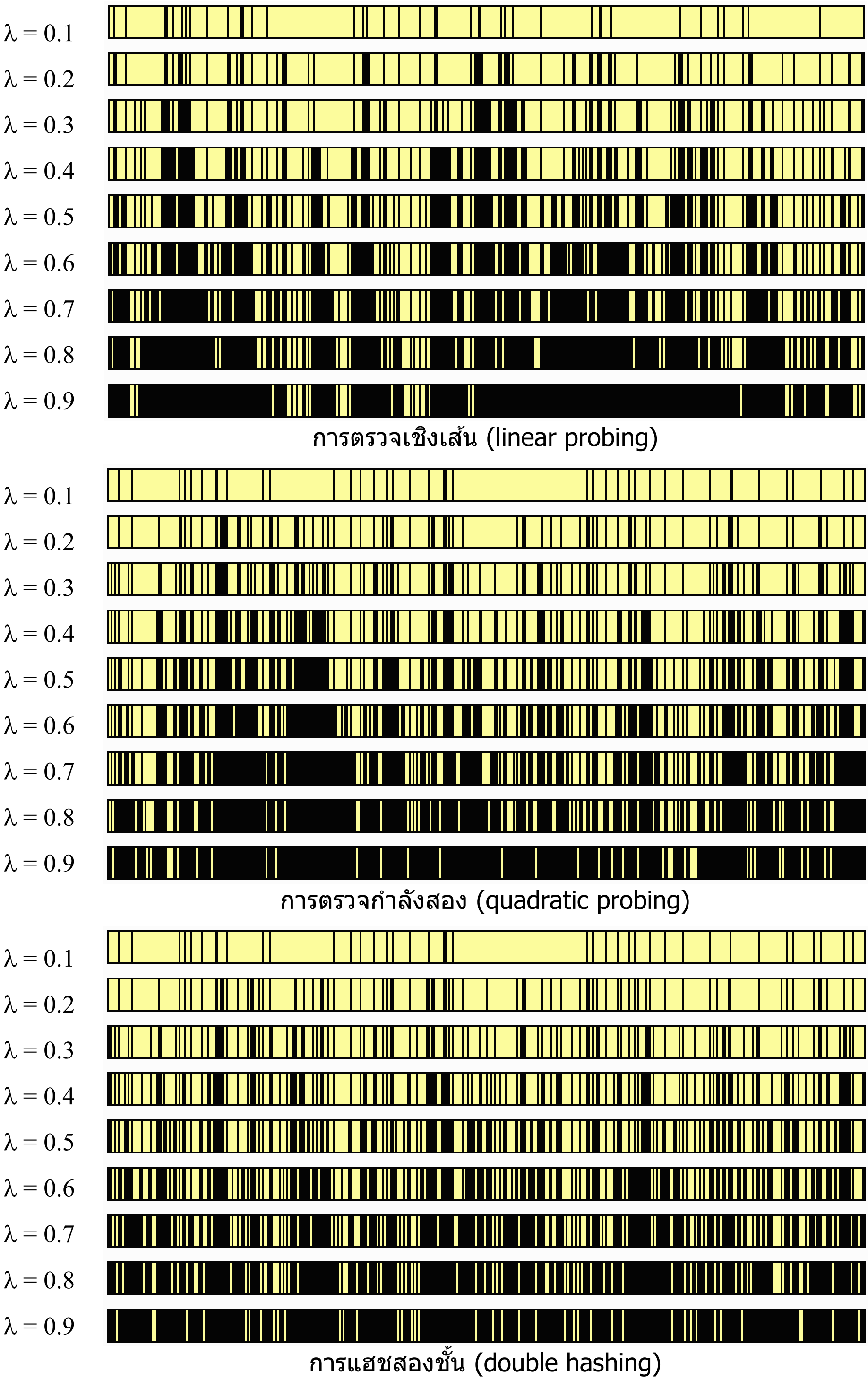
ตารางที่ 11-5 จำนวนช่องเฉลี่ยที่ตรวจ ของการแก้ปัญหาการชนแบบต่าง ๆ

	Linear Probing		Quadratic Probing		Double Hashing	
	พบ	ไม่พบ	พบ	ไม่พบ	พบ	ไม่พบ
λ = 0.3	1.21	1.52	1.21	1.47	1.19	1.43
λ = 0.4	1.33	1.89	1.31	1.75	1.28	1.67
λ = 0.5	1.50	2.50	1.43	2.14	1.39	2.02
λ = 0.6	1.75	3.63	1.59	2.72	1.53	2.54
λ = 0.7	2.16	6.02	1.82	3.70	1.74	3.44
λ = 0.8	3.00	12.84	2.16	5.64	2.05	5.32
λ = 0.9	5.44	49.70	2.79	11.37	2.67	11.63

รูปที่ 11-12 แสดงการกระจายของข้อมูลในตารางระหว่างการเพิ่มข้อมูลชุดเดียวกันขนาด 360 ตัว เก็บในตารางขนาด 400 ช่อง โดยแสดงขณะที่ตารางมีสัดส่วนบรรจุเป็น 0.1 ไปจนถึง 0.9 (ไม่มีการตรวจสอบสัดส่วนบรรจุเพื่อขยายตาราง) เพื่อเปรียบเทียบผลของการตรวจทั้งสามแบบ จะเห็นการเกาะกลุ่มอย่างชัดเจนของการตรวจเชิงเส้น ในขณะที่การตรวจอีกสองแบบมีการเกาะกลุ่มที่น้อยกว่า

ประสิทธิภาพของการแฮชเอกรูป

จนถึงปัจจุบัน ยังไม่มีผู้สามารถวิเคราะห์ประสิทธิภาพการทำงานของการแฮชกำลังสอง แต่สำหรับการแฮชสองชั้นนั้นได้ผู้วิเคราะห์ไว้ว่า มีพฤติกรรมคล้ายกับ*การแฮชเอกรูป* (uniform hashing) ซึ่งคือการแฮชที่ลำดับการตรวจทุกกรณีที่เป็นไปได้มีโอกาสเกิดขึ้นเท่า ๆ กัน (ตารางขนาด m ช่อง มีลำดับการตรวจที่เป็นไปได้ $m!$ กรณี) หรือพูดง่าย ๆ ว่า ช่องต่อไปที่จะตรวจมีสิทธิ์ที่จะเป็นช่องใด ๆ ก็ได้ที่ยังไม่ถูกตรวจ เราเริ่มด้วยการพิจารณากรณีที่ค้นไม่พบข้อมูล ซึ่งมีการตรวจไปเรื่อย ๆ พบแต่ช่องเก็บคีย์ที่ไม่ใช่ตัวที่ต้องการ จนกระทั่งพบช่องว่าง กำหนดให้ p_j คือความน่าจะเป็นที่การค้นไม่พบต้องตรวจเป็นจำนวน j ช่อง ดังนั้นค่าคาดหมายของจำนวนการตรวจในกรณีค้นไม่พบข้อมูลเท่ากับ $\sum_{j=1}^{\infty} j \cdot p_j$



รูปที่ 11-12 การกระจายของข้อมูลในตารางระหว่างการเพิ่มข้อมูลจนมีสัดส่วนบรรจุเป็น 0.9

การคำนวณค่า p_j เพื่อให้ได้ $\sum_{j=1}^{\infty} j \cdot p_j$ ทำได้ยาก แต่เราสามารถคิดอีกแบบโดยกำหนดให้ q_j คือความน่าจะเป็นที่การค้นไม่พบต้องตรวจอย่างน้อย j ช่อง จะได้ว่า

$$q_1 = p_1 + p_2 + p_3 + \ldots$$
$$q_2 = p_2 + p_3 + p_4 + \ldots$$

นำ $q_1, q_2, q_3, \ldots$ จากสูตรข้างบนนี้มาบวกกัน จะได้ $\sum_{j=1}^{\infty} q_j = \sum_{j=0}^{\infty}\left(j \cdot p_j\right)$ เนื่องจากการค้นใด ๆ ก็ต้องตรวจอย่างน้อย 1 ช่อง ดังนั้น $q_1 = 1$ ถ้ามีข้อมูล n ตัวเก็บในตาราง m ช่อง การตรวจครั้งแรกนี้ถ้าพบช่องว่างก็จบ แต่ถ้าพบช่องไม่ว่าง (ซึ่งมีโอกาส n/m) ก็ต้องตรวจต่อ นั่นคือต้องตรวจอย่างน้อย 2 ครั้งด้วยความน่าจะเป็น $q_2 = n/m$ การตรวจครั้งที่สองนี้ถ้าพบช่องว่างก็จบ แต่ถ้าพบช่องไม่ว่าง ซึ่งมีโอกาส $(n-1)/(m-1)$ ก็ต้องตรวจต่อ นั่นคือ q_3 จะเท่ากับ $(n/m)((n-1)/(m-1))$ ด้วยแนวคิดนี้สรุปได้ว่า

$$q_j = \left(\frac{n}{m}\right)\left(\frac{n-1}{m-1}\right)\cdots\left(\frac{n-j-2}{m-j-2}\right) \le \left(\frac{n}{m}\right)^{j-1} = \lambda^{j-1}$$

โดยที่ $\lambda = n/m$ คือสัดส่วนบรรจุของตาราง ดังนั้นค่าคาดหมายของจำนวนการตรวจเมื่อค้นไม่พบคือ

$$\sum_{j=0}^{\infty} j \cdot p_j = \sum_{j=1}^{\infty} q_j \le 1 + \lambda + \lambda^2 + \ldots = \frac{1}{1-\lambda}$$

เนื่องจากการเพิ่มก็คือการตรวจจนพบช่องว่างแล้วจึงใส่ข้อมูล ซึ่งก็มีจำนวนการตรวจเท่ากับกรณีค้นไม่พบ ดังนั้นการเพิ่มข้อมูลจึงมีค่าคาดหมายของจำนวนการตรวจไม่เกิน $1/(1-\lambda)$ เช่นกัน

สำหรับการค้นแล้วพบข้อมูลนั้นจะยุ่งเล็กน้อย ถ้าค้น x แล้วพบ x ก็แสดงว่า การค้นนี้ตรวจตามลำดับเดียวกับตอนที่เพิ่ม x ลงในตาราง ถ้า x เป็นข้อมูลตัวที่ i+1 ที่ถูกเพิ่ม แสดงว่า ตอนที่จะเพิ่ม x นั้น สัดส่วนบรรจุเป็น i/m ดังนั้นค่าคาดหมายของจำนวนการตรวจตอนเพิ่ม x คือ $1/(1-i/m)$ ซึ่งก็คือค่าคาดหมายของจำนวนการตรวจเมื่อค้นพบ x ดังนั้นการคำนวณค่าคาดหมายของการค้นแล้วพบก็เพียงแต่หาค่าเฉลี่ยของข้อมูลทั้ง n ตัว (ถือว่าโอกาสที่จะหาข้อมูลตัวใด ๆ มีพอ ๆ กัน) ซึ่งเท่ากับ

$$\begin{aligned}
\frac{1}{n}\sum_{i=0}^{n-1}\frac{1}{1-i/m} &= \frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i} \\
&= \frac{1}{\lambda}\left(\frac{1}{m} + \frac{1}{m-1} + \frac{1}{m-2} + \cdots + \frac{1}{m-n+1}\right) \\
&= \frac{1}{\lambda}\left(H_m - H_{m-n}\right) \\
&\approx \frac{1}{\lambda}\left(\ln m - \ln(m-n)\right) = \frac{1}{\lambda}\ln\frac{m}{m-n} \\
&= \frac{1}{\lambda}\ln\frac{1}{1-\lambda}
\end{aligned}$$

โดยที่ H_k คือจำนวนฮาร์มอนิกตัวที่ k, $H_k = \sum_{i=1}^{k}\left(1/i\right) \approx \ln k$

ตารางที่ 11-6 สรุปจำนวนช่องเฉลี่ยที่ตรวจของการตรวจเชิงเส้นและการแฮชสองชั้นทั้งกรณีหาพบและหาไม่พบ ซึ่งเป็นฟังก์ชันของสัดส่วนบรรจุ λ ของตาราง ในทางกลับกัน หากเราต้องการให้จำนวนช่องที่ต้องตรวจ เพื่อค้นหาในตารางแฮชมีค่าเฉลี่ยเป็น t ครั้ง ก็สามารถหาได้โดยการแทนค่ากลับ สำหรับการตรวจเชิงเส้น ก็ให้ $t = (1+1/(1-\lambda)^2)/2$ จะได้ $\lambda = 1 - 1/\sqrt{2t-1}$ สำหรับการแฮชสองชั้น ก็ให้ $t = 1/(1 - \lambda)$ จะได้ $\lambda = 1 - 1/t$ เช่น ถ้าต้องการให้ตรวจสัก 5 ช่องโดยเฉลี่ย ก็ต้องควบคุมสัดส่วนบรรจุของการตรวจเชิงเส้นไม่ให้เกิน $1 - 1/\sqrt{2\times5-1} = 0.67$ และของการแฮชสองชั้นต้องไม่ให้เกิน $1 - 1/5 = 0.8$

ตารางที่ 11-6 จำนวนช่องเฉลี่ยที่ตรวจของการตรวจเชิงเส้นและการแฮชสองชั้น

	จำนวนช่องเฉลี่ยที่ตรวจ	
	หาพบ	หาไม่พบ
การตรวจเชิงเส้น	$\frac{1}{2}\left(1+\frac{1}{1-\lambda}\right)$	$\frac{1}{2}\left(1+\frac{1}{(1-\lambda)^2}\right)$
การแฮชสองชั้น	$\frac{1}{\lambda}\ln\frac{1}{1-\lambda}$	$\frac{1}{1-\lambda}$

การเลือกใช้ตารางแฮช

เราได้นำเสนอตารางแฮชแบบแยกกันโยงในตอนต้นบท สรุปตอนนี้อีกครั้งว่า ตารางแฮชแบบนี้ นำข้อมูลที่มี $h(x)$ เหมือนกันมาโยงเก็บอยู่ในรายการเดียวกันที่ช่องที่ $h(x)$ ของตารางแฮช ถ้าให้ตารางมีขนาด m ช่อง เก็บข้อมูล n ตัว ให้สังเกตว่า $n > m$ ได้ สำหรับตารางแฮชแบบแยกกันโยง ซึ่งต่างจากกรณีของตารางแฮชแบบกำหนดเลขที่อยู่เปิดซึ่ง $n \le m$ ถ้าฟังก์ชันแฮชทำหน้าที่กระจายคีย์ต่าง ๆ ไปบนตารางแฮชได้ดี แต่ละช่องในตารางแฮชแบบแยกกันโยงย่อมเก็บรายการโยงที่มีความยาวพอ ๆ กัน คือยาวประมาณ n/m ซึ่งคือสัดส่วนบรรจุ λ ดังนั้นในกรณีค้นไม่พบ ต้องวิ่งครบทุกปมในรายการโยงกับ `null` ตัวสุดท้าย รวมเป็น $1+\lambda$ สำหรับกรณีค้นพบ รายการโยงที่เก็บคีย์ที่ต้องการจะยาว 1 (ซึ่งคือตัวคีย์ที่ต้องการ) บวกกับ $(n-1)/m$ ซึ่งคือคีย์อื่น ๆ เฉลี่ยกันใน m ช่อง จากความรู้ที่ว่า การค้นข้อมูลในรายการที่ยาว k ต้องเปรียบเทียบข้อมูลเป็นจำนวนเฉลี่ย $(k+1)/2$ ครั้ง จึงต้องตรวจเป็นจำนวนเฉลี่ย $(1+(n-1)/m + 1)/2 \approx 1 + n/(2m) = 1 + \lambda/2$ ช่อง ประสิทธิภาพการทำงานจึงขึ้นกับสัดส่วนบรรจุของตาราง ซึ่งก็เหมือนกับของตารางแฮชแบบกำหนดเลขที่อยู่เปิด ดังแสดงในตารางที่ 11-6 ไม่ว่าตารางแฮชจะเป็นแบบใดก็อนุญาตให้เราปรับสัดส่วนบรรจุเพื่อปรับประสิทธิภาพการทำงานได้ เช่น ถ้าปรับ λ ให้น้อยลง ก็คือการยอมสิ้นเปลืองเนื้อที่เพื่อแลกการเวลาการทำงานที่เร็วขึ้น

แล้วน่าใช้แบบใดมากกว่ากัน ? เริ่มด้วยประเด็นของเนื้อที่หน่วยความจำที่ต้องใช้ หลายคนอาจสงสัยว่า แบบแยกกันโยงนั้นเราใช้รายการโยงที่จองปมข้อมูลตามจำนวนข้อมูลที่เก็บอยู่จริง ในขณะที่อีกแบบนั้น โดยทั่วไปแนะนำว่า ให้จองตารางแฮชให้ได้สัดส่วนบรรจุไม่เกิน 0.5 แสดงว่า จองร้อยแต่ใช้ได้อย่างมากเพียงห้าสิบ ซึ่งอ่านแล้วไม่ค่อยคุ้มเท่าไร แต่ต้องอย่าลืมว่า ปมข้อมูลแต่ละปมในรายการโยงต้องเก็บตัวโยงไปยังปมถัดไปซึ่งก็ใช้เนื้อที่เสริมเช่นกัน ลองมาเปรียบเทียบให้เห็นจริงกัน สมมติว่าเราใช้การตรวจเชิงเส้น ถ้าตั้ง λ=0.5 ต้องการเก็บข้อมูล 1,200 ตัว ต้องสร้างตาราง 2,400 ช่อง ให้หนึ่งช่องในตารางแฮชใช้เนื้อที่ 4 ไบต์ (ซึ่งก็คือเนื้อที่สำหรับตัวอ้างอิงของจาวา) แสดงว่า ต้องเสียเนื้อที่ของตารางเท่ากับ 9,600 ไบต์ นำ 0.5 ไปแทน λ ในสูตรของตารางที่ 11-6 จะได้ว่า ถ้าค้นไม่พบ ต้องตรวจประมาณ 2.5 ช่อง คราวนี้มาดูแบบแยกกันโยงบ้าง เมื่อค้นไม่พบต้องตรวจเป็นจำนวน 1+λ โดยที่ λ คือความยาวของรายการโยง เพื่อให้มีประสิทธิภาพเหมือนกัน 1+λ = 2.5 แต่ละรายการโยงจึงต้องยาว 1.5 ดังนั้นเมื่อต้องการเก็บข้อมูล 1,200 ตัว ก็ต้องจองตาราง 800 ช่อง ตารางแฮชแต่ละช่องใช้เนื้อที่ 4 ไบต์ ปมข้อมูลแต่ละปมมีข้อมูลสองส่วน ส่วนหนึ่งเก็บตัวอ้างอิงข้อมูล 4 ไบต์ และอีกส่วนเก็บตัวโยงไปยังปมถัดไปอีก 4 ไบต์ ต้องมีทั้งหมด 1,200 ปม ดังนั้นรวมแล้วใช้เนื้อที่ทั้งหมด 800×4 + 1,200×8 = 12,800 ไบต์ เห็นได้ว่า ด้วยประสิทธิภาพพอกัน แบบแยกกันโยงใช้เนื้อที่มากกว่า

แต่ถ้าเรากำหนดให้ใช้เนื้อที่เท่ากัน จากตัวอย่างก่อน ให้แบบแยกกันโยงเก็บเหมือนเดิม แต่ให้แบบตรวจเชิงเส้นใช้เนื้อที่ 12,800 ไบต์ นั่นหมายความว่า จองตารางได้ถึง 3,200 ช่อง ทำให้ λ ลดลงเหลือ 800/3200 = 0.25 ใช้สูตรในตารางที่ 11-6 พบว่าต้องตรวจช่องเป็นจำนวน 1.39 ช่อง ซึ่งเร็วกว่า

แล้วแบบแยกกันโยงมีอะไรดี ? ต้องอย่าลืมว่า แบบแยกกันโยงนั้นเขียนง่าย การลบก็ตรงไปตรงมา สามารถนำรายการโยงที่เคยเขียน มาใช้ใหม่ได้ แต่จุดเด่นอยู่ตรงที่ ประสิทธิภาพการทำงานไม่ไวต่อคุณภาพของฟังก์ชันแฮชมากนัก ข้อมูลกลุ่มใดที่ชนกันเองน้อย รายการโยงของข้อมูลกลุ่มนั้นก็จะสั้น ไม่ได้รับผลกระทบจากกลุ่มข้อมูลอื่นที่ชนกันเองมาก เพราะต่างกลุ่มต่างแยกกันอยู่ กลุ่มละรายการ ซึ่งไม่เหมือนกับกรณีการกำหนดเลขที่อยู่เปิด กลุ่มที่ชนกันมาก จะระรานไปแย่งช่องของข้อมูลกลุ่มอื่น ข้อมูลที่มาหลังก็ต้องไปหาช่องอื่น ซึ่งก็ไปแย่งใช้ช่องของข้อมูลที่ตามหลังมาต่อเนื่องไปเรื่อย ๆ นอกจากนี้หากสัดส่วนบรรจุเพิ่มขึ้น ๆ ประสิทธิภาพการทำงานแบบแยกกันโยงจะค่อย ๆ ช้าลง แบบไม่ฉับพลัน สำหรับกรณีที่แบบแยกกันโยงใช้เนื้อที่เปลืองกว่านั้น ก็ไม่เป็นเช่นนั้นเสมอไป ถ้าเราสร้างตารางแฮชให้แต่ละช่องเก็บข้อมูลขนาดใหญ่ เช่น แบบ `long` (หนึ่งตัว 8 ไบต์) ไม่ได้เก็บตัวอ้างอิงไปยังอ็อบเจกต์ในแบบที่เขียนมา ถ้าใช้กับตัวอย่างก่อนนี้ด้วยประสิทธิภาพทัดเทียมกัน ตารางแฮชแบบตรวจเชิงเส้นใช้เนื้อที่ 2,400×8 = 19,200 ไบต์ ในขณะที่แบบแยกกันโยงใช้ 800×4 + 1,200×12 = 17,600 ไบต์ ซึ่งน้อยกว่า

ข้อควรระวัง

เราได้แสดงให้เห็นว่า การเก็บข้อมูลในตารางแฮช ได้ผลที่ดีมาก หากควบคุมสัดส่วนบรรจุให้อยู่ในเกณฑ์ สามารถกล่าวได้ว่า การเพิ่ม ลบ และค้นหาข้อมูลใช้เวลาคงตัว แล้วจะมีเหตุผลใดเล่าที่ต้องไปสนใจต้นไม้ค้นหาแบบทวิภาคสารพัดแบบ หรือโครงสร้างข้อมูลอื่น ๆ ที่ได้กล่าวมาตั้งแต่บทแรก ต้องขอย้ำว่า โครงสร้างข้อมูลแต่ละแบบก็มีจุดเด่นของตัวเอง สิ่งที่ตารางแฮชไม่ชอบคือบริการที่เกี่ยวกับอันดับของข้อมูล เช่น การหาคีย์ตัวน้อยสุด ตัวมากสุด หรือตัวถัดจากตัวที่กำหนดให้ บริการเหล่านี้ล้วนต้องวิ่งไล่เปรียบเทียบทุกช่องทุกข้อมูลในตารางจึงจะได้คำตอบ ทั้งนี้ก็เพราะการใช้ฟังก์ชันแฮชที่ทำให้คีย์กระจาย ทำให้คีย์ที่มีค่าใกล้กันเมื่อผ่านแฮชก็อาจแยกกันอยู่ที่ใดก็มิอาจคาดได้ ดังตัวอย่างการหาคีย์น้อยสุดของตารางแฮชที่ใช้การตรวจกำลังสองแสดงในรหัสที่ 11-19 ต้องนำคีย์ของช่องที่ไม่ใช่ `null` และไม่ใช่ `DELETED` ทั้งหมดมาเปรียบเทียบ จึงใช้เวลาเป็น $\Theta(m)$ สำหรับกรณีของตารางแฮชแบบแยกกันโยง ต้องวิ่งทุกช่องในตารางแฮช และเปรียบเทียบทุกปมของทุกรายการ จึงใช้เวลาเป็น $\Theta(m+n)$ โดยที่ m คือขนาดของตาราง และ n คือจำนวนข้อมูล

```
public class QuadraticProbingHashMap implements Map {
  ...
  public Object getMin() {          ต้องลุยทุกช่องในตาราง
    Object min = null;
    for (int i=0; i<table.length; i++) {
      if (table[i] != null && table[i] != DELETED) {
        if (min == null ||
             ((Comparable) min).compareTo(table[i].key) > 0)
          min = table[i].key;
      }
    }
    return min;
  }
```

รหัสที่ 11-19 การหาค่าน้อยสุดต้องเปรียบเทียบหาทั้งตาราง

ในจาวา ฟังก์ชันแฮชที่เราได้เขียนมา ตัวเมท็อด `h` อาศัยการเรียก `hashCode` ของคีย์ เพื่อได้จำนวนเต็มคืนกลับ จากนั้นลบบิตซ้ายสุดให้เป็น 0 (ด้วยการแอนด์กับ 0x7FFFFFFF) เพื่อทำให้เป็นจำนวนไม่ติดลบ แล้วมอดุโลด้วยขนาดตาราง ฟังก์ชันแฮชจะมีคุณภาพหรือไม่ก็ขึ้นกับ `hashCode` ของคีย์ที่ใช้ แต่ในมุมมองของตารางแฮช ตราบเท่าที่ `hashCode` คืนจำนวนเต็มอะไรมา ไม่ชนก็แล้วไป ชนก็หาที่เก็บให้ จึงสามารถเก็บข้อมูลได้ตลอด เช่น `hashCode` ที่คืน 0 เสมอ ก็ยังใช้งานได้ แต่ข้อมูลทุกตัวจะชนกันแหลก การเก็บข้อมูลในตารางแฮชที่คิดว่าดี กลับทำงานเสมือนกับเก็บข้อมูลในรายการ ถ้าข้อมูลน้อย ก็อาจไม่รู้สึกว่าช้า แต่พอข้อมูลมากขึ้น ๆ ระบบทำงานช้าลง ๆ ผู้ออกแบบระบบอาจไม่ทราบได้ว่า สาเหตุทั้งหมดมาจากตัวฟังก์ชันแฮช เพราะคิดว่า ตารางแฮชไม่น่ามีปัญหาอะไร

เนื่องจากใช้คลาสมาตรฐานที่ระบบมีให้ ด้วยเหตุนี้ผู้ออกแบบคลาสของคีย์ที่จะถูกเก็บในตารางแฮชจึงต้องให้ความสำคัญกับเมท็อด `hashCode` ซึ่งขอย้ำอีกครั้งว่า `hashCode` ที่ถูกต้อง คือ `hashCode` ที่คืนจำนวนเต็มเดียวกันเมื่อเรียกกับคีย์ที่เปรียบเทียบด้วย `equals` แล้วเท่ากัน นอกจากนี้ `hashCode` ที่ดี ต้องทำหน้าที่ "บดสับ" คีย์ให้ "เละ" เพื่อให้ได้ผลลัพธ์ที่กระจายในช่วงจำนวนเต็มที่กว้างกว่าขนาดของตารางด้วย

จาวามีคลาสซึ่งใช้ตารางแฮชเก็บข้อมูล เช่น `HashMap`, `HashSet`, `LinkedHashSet` `LinkedHashMap`, `IdentityHashMap`, `ConcurrentHashMap`, `WeakHashMap` เป็นต้น แต่ละคลาสถูกออกแบบมาเพื่อการใช้งานแตกต่างกันไป (ขอไม่อธิบายในที่นี้) โดยเป็นตารางแฮชแบบแยกกันโยง ผู้ใช้สามารถกำหนดขนาดของตารางและสัดส่วนบรรจุที่ต้องการควบคุมได้ ภายในบางคลาสมีการนำ `hashCode` ของคีย์มาเรียงสับเปลี่ยนบิตเพิ่มอีก (เพราะไม่ไว้ใจว่า ผู้ใช้อาจเขียน `hashCode` ได้ไม่ดีนัก) เช่น ใน คลาส `HashMap` สร้างตารางขนาด 2^k มีเมท็อด `hash` ดังแสดงข้างล่างนี้ ที่คืนผลแล้วจะถูกแอนด์ด้วย 2^k-1 ได้เป็นเลขที่ช่องของตาราง

```
static int hash(Object x) {
  int h = x.hashCode();
  h += ~(h << 9);
  h ^=  (h >>> 14);
  h +=  (h << 4);
  h ^=  (h >>> 10);
  return h;
}
```

แบบฝึกหัด

1. จงเขียนคลาส `SeparateChainingHashMap` และ `QuadraticProbingHashMap` ด้วยตนเอง โดยไม่ดูรายละเอียดในหนังสือ

2. จงเขียนการเปลี่ยนแปลงของข้อมูลในตารางแฮช ในแต่ละข้อย่อยข้างล่างนี้ เมื่อเราเพิ่มข้อมูลที่มีคีย์เป็นจำนวนเต็มด้วยลำดับ 6, 27, 23, 14, 49, 40, 36, 39, 66 โดยใช้ $h(x) = x \% 13$

   2.1. ตารางแฮชแบบแยกกันโยง

   2.2. ตารางแฮชแบบการตรวจเชิงเส้น

   2.3. ตารางแฮชแบบการตรวจกำลังสอง

   2.4. ตารางแฮชแบบการแฮชสองชั้น โดยที่ $g(x) = 7 - (x \% 7)$

3. จงหาตัวอย่างการเขียนเมท็อด `hashCode` ของคลาสในคลังคลาสมาตรฐานของจาวาจากรหัสต้นฉบับ (หมายเหตุ : รหัสต้นฉบับของคลาสต่าง ๆ ในคลังคลาสมาตรฐานของจาวาอยู่ในแฟ้มชื่อ `src.zip` ภายใต้สารบบที่เราติดตั้งชุดพัฒนาโปรแกรมภาษาจาวา เช่น ของเครื่องที่ผู้เขียนใช้จะอยู่ที่ `"C:\Program Files\Java\jdk1.5.0_06\src.zip"`)

4. จงเขียนคลาส `SeparateChainingHashMap` โดยที่ภายในใช้รายการโยงที่สร้างจากคลาสต่อไปนี้ที่ได้เคยเขียนกันมาให้เป็นประโยชน์

    4.1. `ArrayList`

    4.2. `SinglyLinkedList`

    4.3. `LinkedList`

5. จงเขียนคลาสใหม่ที่เป็นแบบ `abstract` ชื่อ `OpenAddressingHashMap` เพื่อเป็นคลาสแม่ให้กับคลาส `LinearProbingHashMap`, `QuadraticProbingHashMap` และ `DoubleHashingHashMap` โดยนำเมท็อดที่ใช้ร่วมกันในคลาสลูกทั้งสามไปไว้ที่คลาสแม่

6. จงปรับปรุงให้คลาส `SeparateChainingHashMap` สามารถควบคุมสัดส่วนบรรจุของตารางให้เป็นไปตามความต้องการของผู้ใช้

7. จงเขียนคลาส `SeparateChainingHashSet` และ `LinearProbingHashSet` ที่ใช้ตารางแฮชสร้างเซต พร้อมทั้งเขียนเมท็อด `equals` เพื่อเปรียบเทียบว่า เซตสองเซตมีข้อมูลเหมือนกันหรือไม่

8. จงเขียนโปรแกรมเพื่อแสดงให้เห็นจริงว่า เมท็อด `hashCode` ของคลาส `String` ซึ่งมองตัวอักษรเป็นเลขฐาน 31 นั้นให้ผลเป็นจำนวนเต็มที่ไม่ซ้ำกันเลย เมื่อใช้กับคีย์ที่เป็นสตริงตัวอักษรอังกฤษตัวใหญ่ ยาว 6 ตัวอักษรทุกรูปแบบ

9. ประสิทธิภาพของการแฮชสองชั้นจะเป็นเช่นไร ถ้า $g(x) = 1$ และเป็นเช่นไรถ้า $g(x) = x \% (m/2)$ โดยที่ m คือขนาดของตาราง

10. เด็กชายคิดเสนอว่า การเก็บข้อมูลในตารางแฮชแบบแยกกันโยงนั้น สามารถทำให้ดีขึ้นได้โดยนำข้อมูลที่มี $h(x)$ เหมือนกันมาเก็บในต้นไม้เอวีแอลต้นเดียวกัน แล้วเก็บรากของต้นไม้ไว้ในช่องที่ $h(x)$ อยากทราบว่า วิธีที่เด็กชายคิดนำเสนอนั้นมีข้อดีข้อเสียอย่างไร

11. จงเปรียบเทียบผลการทดลองที่แสดงในตารางที่ 11-5 กับการคำนวณโดยใช้สูตรในตารางที่ 11-6

12. ในรหัสที่ 11-17 เราลบโดยการใส่ `DELETED` ไว้ในช่องที่จะลบ พอตอนจะเพิ่มก็ใช้ `indexOf` หาจนพบคีย์ หรือไม่ก็พบ `null` จึงทำให้ช่องที่เป็น `DELETED` ไม่มีทางถูกนำมาใช้ใหม่ จนกว่าจะเกิดการ `rehash` คุณนัทแนะนำว่า ระหว่างการตรวจตอนจะเพิ่มข้อมูล ก็ให้หยุดเมื่อพบ `DELETED` แล้วนำข้อมูลใหม่ใส่ลงแทนเพียงเท่านี้ก็เป็นการนำช่องที่เคยถูกลบมาใช้ใหม่ได้ ผู้เขียนบอกนัทว่า วิธีนี้ผิด อยากทราบว่า ผิดอย่างไร ยกตัวอย่างประกอบ และจงเสนอวิธีที่สามารถนำช่องที่มี `DELETED` กลับมาใช้ใหม่ได้อย่างถูกต้อง

13. คุณนันทกรเสนอวิธีแก้ไขปัญหาการชนแบบใหม่ดังนี้ กำหนดให้ตารางแฮชมีขนาด m ช่อง ให้ $h_j(\mathrm{x}) = (\ h(x) + F(j)\ )\ \%\ m$ ซึ่งคือช่องที่ต้องตรวจหลังการชนครั้งที่ j โดยที่ให้ $F(0) = 0$ และ $F(1),\ F(2)\ ,\ \ldots,\ F(m-1)$ คือการเรียงสับเปลี่ยนอย่างสุ่ม (random permutation) ของเลข 1 ถึง $m-1$ (ซึ่งเราเตรียมอย่างสุ่มไว้ในตัวสร้างตารางแฮช หมายความว่า ตารางแฮชแต่ละตารางอาจมีค่า $F(j)$ ต่างกันได้) อยากทราบว่า (อธิบายเหตุผลประกอบด้วย)

    13.1. วิธีนี้จะตรวจพบช่องว่างในตารางแฮชหรือไม่ ถ้าข้อมูลยังไม่เต็มตาราง

    13.2. วิธีนี้จะแก้ปัญหาการเกาะกลุ่มปฐมภูมิหรือไม่

    13.3. วิธีนี้จะแก้ปัญหาการเกาะทุติยภูมิ หรือไม่

14. คุณคณิตต้องการนำอ็อบเจกต์แบบ `ArrayList` ไปเก็บในแต่ละช่องของตารางแฮชได้ เขาจึงต้องเขียน `hashCode` ให้กับคลาส `ArrayList` ดังแสดงข้างล่างนี้ แต่ผู้เขียนแนะนำว่า ไม่ค่อยดีเท่าไร จงอธิบายว่า ทำไมไม่ดี และควรเป็นเช่นใด

```
public class ArrayList implements List {
  ...
  public int hashCode() {
    int h = 0;
    for(int i=0; i<elementData.length; i++)
      h += elementData[i].hashCode();
    return h;
  }
```

15. ตารางที่ 11-1 นำเสนอความถี่ของคำที่ใช้กันมากในเพลงไทย โดยใช้เนื้อเพลงจำนวนเพียงแค่พันกว่าเพลงเท่านั้น จงนำเนื้อเพลงที่หาได้จากอินเตอร์เน็ตหรือจากแผ่นซีดีรอมคาราโอเกะที่มีขายตามท้องตลาด (ภายในมีทั้งเนื้อร้องและทำนอง) มาหาความถี่ของคำที่ใช้มากสุด 20 อันดับแรก แยกวิเคราะห์สำหรับเนื้อเพลงไทยและอังกฤษทุกเพลงที่หาได้ (ข้อแนะนำ : โปรแกรมนี้เขียนได้ง่าย ๆ โดยใช้แมป สำหรับการแยกคำไทยออกจากข้อความให้ใช้คลาสมาตรฐานของจาวาชื่อ `BreakIterator` ในชุด `java.text`)

16. ทำไมการทำงานของส่วนโปรแกรมข้างล่างนี้ จึงได้ผลเป็น `true`, `false`, `true` (คลาส `Point` ข้างล่างนี้เป็นคลาสในคลังคลาสมาตรฐานจาวา อยู่ในชุด `java.awt`)

```
Map m = new LinearProbingHashMap(10);
Point p = new Point(10,20);
m.put(p, "ok");
System.out.println(m.containsKey(p));  // true
p.x = 99;
System.out.println(m.containsKey(p));  // false
p.x = 90;
System.out.println(m.containsKey(p));  // true
```

12 ตัวแจงย้ำ

การเข้าถึงข้อมูลที่จัดเก็บในคอลเล็กชัน รายการ เซต หรือแมปที่ได้นำเสนอมามีหลากหลายรูปแบบ เช่น ใช้ `toArray` เพื่อคืนแถวลำดับที่เก็บข้อมูลทุกตัว, ใช้ `get(i)` เพื่อคืนข้อมูลที่เก็บ ณ ตำแหน่ง `i` ของรายการ, หรือใช้ตัวเยี่ยมชมผนวกกับการแวะผ่านต้นไม้เป็นต้น `toArray` นั้นเปลืองเนื้อที่และยังเสียเวลาเติมข้อมูลทั้งหมดลงแถวลำดับ `get(i)` ทำงานได้ดีเฉพาะเมื่อโครงสร้างภายในเก็บแบบแถวลำดับ โดยใช้เลขที่ช่องเป็นตัวอ้างอิง จึงใช้ได้ดีกับ `ArrayList` แต่ไม่ค่อยมีประสิทธิภาพถ้าใช้กับ `LinkedList` ส่วนการใช้ตัวเยี่ยมชมกับการแวะผ่านต้นไม้นั้นอาจดูไม่ค่อยเป็นที่คุ้นเคยที่ต้องสร้างอ็อบเจกต์ซึ่งเก็บส่วนประมวลผลเพื่อส่งให้เจ้าของข้อมูลเป็นผู้เรียกใช้ ปัญหาการเข้าถึงข้อมูลที่มีหลายลักษณะและประสิทธิภาพการทำงานต่างกันนี้เอง จึงเป็นที่มาของการใช้แนวคิดของ *ตัวแจงย้ำ* (iterator) ซึ่งเป็นอ็อบเจกต์ที่เปรียบเสมือนตัวอ้างอิงตำแหน่งข้อมูลในที่เก็บข้อมูล มีไว้ให้ผู้ใช้แจกแจงข้อมูลในที่เก็บออกมาใช้ทีละตัว ๆ ทำให้การเข้าถึงข้อมูลในที่เก็บหลากหลายลักษณะอยู่ในรูปแบบเดียวกัน และมีประสิทธิภาพด้วย

การใช้งาน

สมมติว่า เราต้องการเขียนเมท็อด `countLessThan(d,x)` เพื่อนับว่า มีข้อมูลกี่ตัวใน `d` ที่มีค่าน้อยกว่า `x` จะเขียนเมท็อดนี้อย่างไรก็คงขึ้นกับว่า ที่เก็บข้อมูล `d` มีโครงสร้างอย่างไร และเรามีสิทธิ์เข้าถึงโครงสร้างภายในได้มากน้อยเพียงใด โดยปกติผู้ออกแบบโครงสร้างข้อมูลมักไม่เปิดเผย และไม่อนุญาตให้ผู้ใช้เข้าถึงโครงสร้างภายใน (เห็นได้จากตัวอย่างที่เราได้เขียนคลาสกันมาว่า เรามักให้ตัวแปรภายในเป็นแบบ `private` หรือไม่ก็เป็น `protected` เพื่อให้คลาสลูกใช้เท่านั้น แต่จะไม่ให้เป็นแบบ `public`) จึงต้องพยายามใช้บริการที่ `d` มีให้ เพื่อทำในสิ่งที่ต้องการ รหัสที่ 12-1 แสดงการ

เขียนเมท็อด `countLessThan` โดยเขียนไว้สามแบบ overload กัน ขึ้นกับประเภทของที่เก็บข้อมูล `d` ที่ได้รับ ถ้าเป็น `LinkedCollection` ก็ใช้ `toArray` ดึงข้อมูลเก็บใส่แถวลำดับ แล้วค่อยนำมานับ ถ้าเป็น `ArrayList` ก็ใช้ `get(i)` ค่อย ๆ หยิบมานับ แต่ถ้าเป็น `BSTree` ก็อาศัยตัวเยี่ยมชมกับการแวะผ่านต้นไม้ จะเห็นได้ว่า การเขียนในลักษณะนี้ไม่มีรูปแบบการเข้าถึงอย่างเป็นมาตรฐาน เป็นการยืมกลไกอื่นมาช่วย

```
  public static int countLessThan(LinkedCollection d, Object x) {
    Object[] a = d.toArray();
    int c = 0;
    for (int i=0; i<a.length; i++) {
      if (((Comparable)a[i]).compareTo(x) < 0) c++;
    }
    return c;
  }
  public static int countLessThan(ArrayList d, Object x) {
    int c = 0, n = d.size();
    for (int i=0; i<n; i++) {
      if (((Comparable)d.get(i)).compareTo(x) < 0) c++;
    }
    return c;
  }
  public static int countLessThan(BSTree d, final Object x) {
    final int[] c = new int[1];
    d.preOrder(new Visitor() {
                    public void visit(Object e) {
                      if(((Comparable)e).compareTo(x) < 0) c[0]++;
                    }
                  });
    return c[0];
  }
```

ใช้ toArray

ใช้ get(i)

ใช้ตัวเยี่ยมชมกับการแวะผ่านต้นไม้

รหัสที่ 12-1 การเข้าถึงข้อมูลภายในโครงสร้างข้อมูลด้วยวิธีแตกต่างกัน

ขอนำเสนอการเข้าถึงข้อมูลในที่เก็บโดยอาศัยสิ่งที่เรียกว่า *ตัวแจงย้ำ* (iterator) เราขออ็อบเจกต์ตัวแจงย้ำ *i* จากที่เก็บข้อมูล *d* เพื่อแจกแจงข้อมูลจาก *d* โดยมักเรียกใช้ *i* ย้ำ ๆ เพื่อแจกแจงข้อมูลจาก *d* ออกมาประมวลผลหลาย ๆ ตัวจนกว่าจะพอใจ หรือจนกว่าจะหมด รหัสที่ 12-2 แสดงอินเทอร์เฟซ `Iterator` ของจาวา (ในชุด `java.util`) ซึ่งเราจะใช้ในการสร้างตัวแจงย้ำ ประกอบด้วยเมท็อด `next` ที่คืนอ็อบเจกต์ตัวถัดไปในที่เก็บ เมท็อด `hasNext` ซึ่งคืนค่าจริง ถ้ายังมีข้อมูลเหลือให้แจง

```
public interface Iterator {
  public boolean hasNext();
  public Object next();
  public void remove();   // optional operation
}
```

จะให้บริการลบหรือไม่ก็ได้

รหัสที่ 12-2 อินเทอร์เฟซ `Iterator` ของตัวแจงย้ำ

ด้วย next (ส่วน remove ขอยังไม่ลงรายละเอียดขณะนี้) ถ้าคลาส ArrayList, BSTree, และ LinkedCollection มีเมท็อด iterator ที่คืนตัวแจงย้ำของที่เก็บ เราก็สามารถเขียนรหัสที่ 12-1 ใหม่ให้ทำงานโดยใช้ตัวแจงย้ำได้ดังรหัสที่ 12-3

```
public int countLessThan(LinkedCollection d, Object x) {
  int c = 0;
  for (Iterator itr = d.iterator(); itr.hasNext();) {
    if (((Comparable) itr.next()).compareTo(x) < 0) c++;
  }
  return c;
}
public int countLessThan(ArrayList d, Object x) {
  int c = 0;
  for (Iterator itr = d.iterator(); itr.hasNext();) {
    if (((Comparable) itr.next()).compareTo(x) < 0) c++;
  }
  return c;
}
public int countLessThan(BSTree d, Object x) {
  int c = 0;
  for (Iterator itr = d.iterator(); itr.hasNext();) {
    if (((Comparable) itr.next()).compareTo(x) < 0) c++;
  }
  return c;
}
```

รหัสที่ 12-3 การใช้ตัวแจงย้ำดึงข้อมูลภายในโครงสร้างข้อมูลออกมาประมวลผล

เห็นได้ชัดจากรหัสที่ 12-3 ว่า เราเขียนการประมวลผลด้วยตัวแจงย้ำในลักษณะเดียวกันหมดกับโครงสร้างข้อมูลที่ต่างกัน ต้องบอกตรงนี้เลยว่า ตัวแจงย้ำของที่เก็บข้อมูลที่ต่างกัน ย่อมมีกลไลการทำงานภายในต่างกันเพื่อแจงแจงข้อมูลภายในออกมาด้วยประสิทธิภาพที่ดีสุด เพราะผู้ออกแบบที่เก็บข้อมูลเป็นผู้ออกแบบตัวแจงย้ำของที่เก็บข้อมูลนั้น ซึ่งในมุมมองของผู้ใช้ตัวแจงย้ำ ก็ไม่น่าต้องสนใจกลไกการทำงานภายใน ตราบเท่าที่ตัวแจงย้ำยังคงมีเมท็อดให้บริการซึ่งเป็นไปตามอินเทอร์เฟซ Iterator ก็เป็นใช้ได้

```
public interface Iterable {
  public Iterator iterator();
}
```

Iterable อยู่ในชุด java.lang ของจาวา

รหัสที่ 12-4 อินเทอร์เฟซ Iterable บังคับเมท็อด iterator ที่คืนตัวแจงย้ำ

จาวามีอินเทอร์เฟซอีกตัวหนึ่งชื่อว่า Iterable (รหัสที่ 12-4) ซึ่งบังคับเมท็อด iterator ที่คืนตัวแจงย้ำ ดังนั้นถ้าเราปรับปรุงที่เก็บข้อมูลต่าง ๆ ที่ได้เขียนกันมาให้มีเมท็อด iterator และปรับให้คลาสเหล่านั้น implements Iterable ก็สามารถเขียน countLessThan เพียงแค่

เมท็อดเดียวที่รับพารามิเตอร์แบบ Iterable ดังรหัสที่ 12-5 แล้วส่งที่เก็บข้อมูลแบบ BSTree, ArrayList, และ LinkedCollection ให้กับ countLessThan แบบใหม่นี้ได้

```
public int countLessThan(Iterable d, Object x) {
  int c = 0;
  for (Iterator itr = d.iterator(); itr.hasNext();) {
    if (((Comparable)itr.next()).compareTo(x) < 0) c++;
  }
  return c;
}
```

รหัสที่ 12-5 เมท็อด countLessThan ที่รับ Iterable ซึ่งมีเมท็อด iterator ให้เรียกใช้

อินเทอร์เฟซ Iterable เป็นอินเทอร์เฟซใหม่ที่เพิ่งถูกเพิ่มเข้าในคลังคลาสมาตรฐานของระบบจาวาตั้งแต่รุ่นที่ 5 นอกจากนี้ภาษาจาวาตั้งแต่รุ่นที่ 5 เป็นต้นไป มีคำสั่งวงวน for แบบใหม่ที่เรียกว่า for each ซึ่งอำนวยความสะดวกในการเขียนวงวนที่แจงแจงข้อมูลด้วยตัวแจงย้ำ คือแทนที่เราจะเขียนวงวนข้างล่างนี้ (เหมือนกับในรหัสที่ 12-5)

```
for (Iterator itr = d.iterator(); itr.hasNext(); ) {
  Object x = itr.next();
  ...
}
```

ก็สามารถเขียนให้สั้นลง โดยไม่ต้องแสดงตัวแจงย้ำให้เห็นได้ดังนี้

```
for (Object x : d) {
  ...
}
```

อ่านว่า "สำหรับแต่ละอ็อบเจกต์ x ใน d" ทำให้เขียนโปรแกรมได้สั้น อ่านได้ใจความ แต่ก็ต้องเข้าใจด้วยว่า จะเขียนเช่นนี้ได้ d ต้องเป็นอ็อบเจกต์แบบ Iterable (แถวลำดับของจาวาก็เป็นอ็อบเจกต์แบบ Iterable จึงสามารถใช้ได้กับคำสั่ง for each เช่นกัน)

ตัวแจงย้ำสำหรับการเก็บข้อมูลในแถวลำดับ

โครงสร้างง่ายสุดในการเก็บข้อมูลเห็นจะหนีไม่พ้นแถวลำดับ ที่ผ่านมาเราได้นำแถวลำดับมาเก็บข้อมูล (ตั้งชื่อว่า elementData) มีตัวแปร size เก็บจำนวนข้อมูล โดยให้เก็บตั้งแต่ช่องที่ 0 ติด ๆ กันไป จนถึงช่องที่ size-1 คลาสต่าง ๆ ที่เขียนลักษณะเช่นนี้ ได้แก่ ArrayCollection, ArrayList, ArrayStack, และ BinaryHeap จากนี้ไปจะขอใช้ ArrayCollection เป็นคลาสตัวอย่างในการนำเสนอการเขียนตัวแจงย้ำให้กับคลาสที่ใช้แถวลำดับเก็บข้อมูล [1]

---

[1] ขอเตือนว่าอย่าสับสนกับคำสามคำนี้ Iterable, Iterator และ iterator, สองตัวแรกเป็นอินเทอร์เฟซ ตัวหลังเป็นชื่อเมท็อด อินเทอร์เฟซ Iterable บังคับให้มีเมท็อด iterator ที่ต้องคืนอ็อบเจกต์แบบ Iterator สำหรับอินเทอร์เฟซ Iterator บังคับเมท็อด next, hasNext, และ remove ผู้อ่านต้องอ่านอย่างระมัดระวัง

รหัสที่ 12-6 แสดงรายละเอียดสิ่งที่ต้องเพิ่มเติมให้กับคลาส `ArrayCollection` เพื่อให้บริการตัวแจงย้ำ ภาระแรกคือการประกาศให้ `ArrayCollection` เป็นแบบ `Iterable` โดยเขียนไว้ที่หัวคลาสให้ `implements Iterable` จากนั้นเขียนเมท็อด `iterator` ให้สร้างและคืนตัวแจงย้ำ ในที่นี้เราสร้างเป็นอ็อบเจกต์ของคลาส `Itr` กลับคืนไป โดยคลาสนี้เป็นคลาสภายในซึ่ง `implements Iterator` (จึงต้องมีเมท็อด `hasNext`, `next` และ `remove`) ภายในอ็อบเจกต์ตัวแจงย้ำมีตัวแปร `cursor` เก็บเลขช่องของ `elementData` ที่จะคืนให้กลับผู้เรียก `next()` ครั้งถัดไป โดย `cursor` มีค่าเป็น `0` ตอนเริ่มต้น (บรรทัดที่ 48) ดังนั้นในเมท็อด `next()` จึงคืนช่องที่ `cursor` ของ `elementData` แล้วเพิ่มค่า `cursor` (บรรทัดที่ 54) แต่ก่อนจะคืนต้องตรวจสอบให้มั่นใจว่า มีตัวถัดไปให้คืน ด้วย `hasNext` ถ้าได้ค่าเท็จ แสดงว่า มีอะไรบางอย่างผิดพลาด ก็ให้โยนสิ่งผิดปกติให้ระบบรับทราบ (บรรทัดที่ 53) เนื่องจากข้อมูลถูกเก็บใน `elementData` ตั้งแต่ช่องที่ `0` ถึงช่องที่ `size-1` ดังนั้นการตรวจสอบว่า ยังมีข้อมูลตัวถัดไปใน `hasNext` ก็เพียงแค่ตรวจสอบว่า `cursor < size` หรือไม่ ถ้าใช่ แสดงว่ายังมีข้อมูลเหลืออยู่ให้ `next` (บรรทัดที่ 50)

```
01 public class ArrayCollection implements Collection, Iterable {
02   private Object[] elementData;
03   private int size;
..   ...
44   public Iterator iterator() {
45     return new Itr();
46   }
47   private class Itr implements Iterator {
48     int cursor = 0;
49     public boolean hasNext() {
50       return cursor < size;
51     }
52     public Object next() {
53       if (!hasNext()) throw new IllegalStateException();
54       return elementData[cursor++];
55     }
56     public void remove() {
57       throw new UnsupportedOperationException();
58     }
59   }
```

ต้องประกาศว่าเป็น Iterable

เมท็อด iterator คืนตัวแจงย้ำให้ไปใช้งาน

cursor เก็บเลขช่องที่จะคืนข้อมูลในการ next ครั้งต่อไป

ไม่ให้บริการลบ ถ้าเรียกจะเกิดสิ่งผิดปกติ

รหัสที่ 12-6 ตัวแจงย้ำของคลาส `ArrayCollection`

สำหรับเมท็อด `remove` นั้น โดยความหมายแล้วหมายความว่า ให้ลบข้อมูลตัวที่เพิ่งถูก `next` ตัวล่าสุด ข้อกำหนดของ `Iterator` ในจาวาบอกว่า เมท็อดนี้เป็นแบบ optional หมายความว่าผู้ ออกแบบตัวแจงย้ำสามารถเลือกให้บริการหรือไม่ก็ได้ ส่วนผู้ใช้ตัวแจงย้ำก็ต้องศึกษาให้ดีว่า ตัวแจงย้ำที่ใช้นั้นเขาให้บริการการลบหรือเปล่า เพราะไม่ได้บังคับไว้ ในกรณีที่ผู้ออกแบบไม่ต้องการ

ให้บริการ remove ก็ให้เขียนเอาไว้ในคู่มือด้วยว่า จะไม่ให้บริการ (แต่จะไม่เขียน remove นั้นไม่ได้ เพราะมันถูกระบุไว้ในอินเทอร์เฟซ Iterator) แล้วจะเขียนอะไรไว้ภายในดี โดยทั่วไปให้ throw UnsupportedOperationException (บรรทัดที่ 57) ซึ่งอาจดูแปลก ใครมาเรียก remove ต้องเกิดปัญหา ซึ่งต้องเข้าใจด้วยว่า ก็บอกในคู่มือแล้วว่าไม่ให้บริการ แล้วยังลองดีมาเรียกก็จะเกิดปัญหา ถึงแม้แนวคิดนี้อาจรู้สึกไม่ดี แต่เราไม่สามารถทำอะไรได้ดีกว่านี้ เราอาจเลือกเขียนให้ remove ไม่ทำอะไร เงียบ ๆ แล้วก็คืนการทำงาน ซึ่งอาจแลดูสุภาพเรียบร้อย แต่นั่นกลับเป็นผลเสีย เพราะผู้ใช้ที่ไม่ได้อ่านคู่มือ อาจไม่รู้ คิดว่าสั่ง remove แล้วจะลบ แต่กลับไม่ลบ ก็อาจสร้างปัญหาในภายหลังว่า อ้าว สั่งให้ลบแต่ไม่ลบ ดังนั้นการโยนสิ่งผิดปกติให้เกิดกับระบบจึงเป็นการเตือนให้ทราบทันทีที่พบปัญหา ทำให้ผู้ทดสอบระบบพบความผิดพลาดของตัวโปรแกรมได้แต่เนิ่น ๆ

แต่ถ้าต้องการให้บริการลบจริง ๆ ก็สามารถเขียนได้ดังรหัสที่ 12-7 โดยเรารู้ว่า cursor คือช่องที่จะได้ next ตัวถัดไป ช่องที่ cursor-1 จึงเป็นตัวที่ต้องถูกลบ ดังนั้นจึงลบได้ด้วยการนำช่องหลังสุดคือ elementData[size-1] มาแทนใน elementData[cursor-1] ลด size ลงหนึ่งเพราะข้อมูลลด และลดค่า cursor ลงหนึ่งด้วย (บรรทัดที่ 58) แต่ก็อย่าลืมกรณีแปลกเช่นเรียกให้ลบหลังจากสร้างตัวแจงย้ำใหม่ ๆ โดยยังไม่ได้เรียก next เลย ดังนั้นจึงต้องตรวจสอบก่อนว่าถ้า cursor เป็น 0 แสดงว่า ยังไม่เคย next เลย ก็ให้โยนสิ่งผิดปกติให้รับทราบด้วย

```
47   private class Itr implements Iterator {
48     int cursor = 0;
..     ...
56     public void remove() {
57       if (cursor == 0) throw new IllegalStateException();
58       elementData[--cursor] = elementData[--size];
59     }
60   }
```

ยังไม่เคย next ห้ามลบ

ลบช่องที่ cursor-1 ด้วยการนำข้อมูลตัวท้ายมาแทน

รหัสที่ 12-7 การลบด้วยตัวแจงย้ำของคลาส ArrayCollection (ยังผิดอยู่)

จากรหัสที่ 12-7 ถ้าผู้ใช้เรียก next สองครั้ง ตามด้วย remove สองครั้ง จะเกิดอะไรขึ้น จะลบสองตัวล่าสุดหรือไม่ (ผู้อ่านลองคิดดู) เพื่อให้ง่ายต่อการกำหนดพฤติกรรมการทำงานได้แน่ชัด ข้อกำหนดของ Iterator ในจาวาเขียนไว้ว่า การเรียก remove นั้นจะลบเฉพาะตัวที่ถูก next ตัวล่าสุดหนึ่งตัวเท่านั้น ถ้าต้องการลบสองตัว ต้องเรียก next แล้ว remove, next แล้ว remove จึงจะลบได้ ดังนั้นสิ่งที่ต้องปรับปรุงคือ ป้องกันไม่ให้เรียก remove โดยไม่ได้ next ซึ่งแก้ไขง่าย ๆ ด้วยการเพิ่มตัวแปร removable เป็นแบบ boolean (ดูรหัสที่ 12-8) เก็บสถานะของตัวแจงย้ำว่ายอมให้ลบหรือไม่ โดยตอนเริ่มต้นกับตอนหลังลบต้องให้เป็น false แต่ถ้ามีการ next ก็ให้เป็น true ถ้าเรียก remove ตอนที่ removable เป็นเท็จ ก็ให้โยนสิ่งผิดปกติทันที

```
    private class Itr implements Iterator {
      int cursor = 0;
      boolean removable = false;
      ...
      public Object next() {
        if (!hasNext()) throw new IllegalStateException();
        removable = true;
        return elementData[cursor++];
      }
      public void remove() {
        if (!removable) throw new IllegalStateException();
        elementData[--cursor] = elementData[--size];
        removable = false;
      }
```

ตอนเริ่มต้น ไม่ให้ลบ

next แล้ว เรียก remove ได้

ลบแล้ว เรียก remove อีกไม่ได้

รหัสที่ 12-8 ตัวแจงย้ำลบได้เฉพาะตัวที่ next ไปตัวล่าสุดหนึ่งตัวเท่านั้น

ผู้อ่านอาจสงสัยว่า remove ของตัวแจงย้ำจะมีประโยชน์อะไร ในเมื่อเราก็มี remove ของ ArrayCollection ให้เรียกได้อยู่แล้ว สมมติว่า ต้องการเขียนเมท็อด removeLessThan รับ d ซึ่งเป็นที่เก็บข้อมูลแบบ ArrayCollection และ x เป็นอ็อบเจกต์ โดยมีวัตถุประสงค์ให้ลบข้อมูลทุกตัวใน d ที่มีค่าน้อยกว่า x เราสามารถเขียนได้ง่าย ๆ โดยไม่ใช้ remove ของตัวแจงย้ำ แต่ไปเรียกของ ArrayCollection แทน ได้ดังรหัสที่ 12-9 การทำเช่นนี้เสียเวลาโดยใช่เหตุ เพราะ remove ของ ArrayCollection ต้องวิ่งไล่ค้นหาในแถวลำดับให้พบแล้วค่อยลบ ทั้ง ๆ ที่ภายในตัวแจงย้ำนั้นรู้ตำแหน่งของตัวที่อยากลบอยู่แล้ว ดังนั้นจึงควรเขียนดังรหัสที่ 12-10 จะได้ประสิทธิภาพที่ดีกว่า (และจริง ๆ แล้ว เราจะไม่อนุญาตให้ทำแบบรหัสที่ 12-9 เนื่องจาก d เปลี่ยนแปลง โดยที่ตัวแจงย้ำไม่รู้เรื่องเลย อาจทำให้การแจกแจงข้อมูลออกมาผิดพลาดได้ ซึ่งจะได้นำเสนอกันต่อไป)

```
  void removeLessThan(ArrayCollection d, Object x) {
    for (Iterator itr = d.iterator(); itr.hasNext(); ) {
      Object e = itr.next();
      if (((Comparable) e).compareTo(x) < 0) d.remove(e);
    }
  }
```

ลบ e ออกจาก d

รหัสที่ 12-9 การลบข้อมูลทุกตัวใน d ที่น้อยกว่า x โดยใช้ remove ของ ArrayCollection

```
  void removeLessThan(ArrayCollection d, Object x) {
    for (Iterator itr = d.iterator(); itr.hasNext(); ) {
      Object e = itr.next();
      if (((Comparable) e).compareTo(x) < 0) itr.remove();
    }
  }
```

ลบตัวที่ itr เพิ่ง next

รหัสที่ 12-10 การลบข้อมูลทุกตัวใน d ที่น้อยกว่า x โดยใช้ remove ของ Iterator

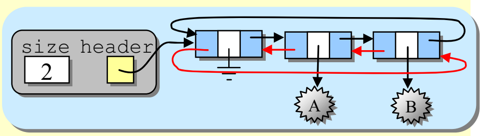
ตัวแจงย้ำสำหรับรายการโยง

การเก็บข้อมูลด้วยรายการโยงดังเช่นที่ได้ทำในคลาส LinkedCollection, LinkedList, SinglyLinkedList สามารถทำให้เป็น Iterable ได้ไม่ยาก รหัสที่ 12-11 แสดงการปรับ LinkedList (รายการโยงคู่แบบวนที่มีปมหัว) เริ่มด้วยการเขียนให้คลาสเป็น Iterable ตามด้วยการเขียนเมท็อด iterator เพื่อสร้างและคืนอ็อบเจกต์ของคลาส Itr โดยที่ Itr เป็นคลาสภายในที่เป็น Iterator มีตัวแปร nextReturn ชี้ปมที่เตรียมจะคืนข้อมูลในการ next ครั้งถัดไป มี removable เก็บสถานะว่า จะลบได้หรือไม่ hasNext ก็เพียงตรวจว่า nextReturn วนกลับมาที่ปมหัวหรือไม่ ถ้าใช่แสดงว่าไม่มีข้อมูลเหลือ ส่วนเมท็อด next เพียงแค่คืนข้อมูลที่ปม nextReturn กลับไป, เลื่อน nextReturn ถัดไปหนึ่งปม, และตั้งให้ removable เป็นจริง สำหรับ remove ก็ตรวจสอบ removable ก่อนลบ ถ้าลบได้ก็เรียก removeNode ซึ่งเป็นเมท็อดของ LinkedList ที่ได้เคยเขียนเพื่อรับปมไปลบทิ้ง โดยเราส่งปมก่อนหน้า nextReturn ไปลบ

```
public class LinkedList implements List, Iterable {
  ...
  private int size;
  private LinkedNode header;
  ...
  public Iterator iterator() {
    return new Itr();
  }
  private class Itr implements Iterator {
    LinkedNode nextReturn = header.next;
    boolean removable = false;
    public boolean hasNext() {
      return nextReturn != header;
    }
    public Object next() {
      if (!hasNext()) throw new IllegalStateException();
      Object x = nextReturn.element;
      nextReturn = nextReturn.next;
      removable = true;
      return x;
    }
    public void remove() {
      if (!removable) throw new IllegalStateException();
      removeNode(nextReturn.prev);
      removable = false;
    }
  }
```

```
private static class LinkedNode {
  private Object element;
  private LinkedNode prev;
  private LinkedNode next;
  ...
}
```

nextReturn ชี้ปมที่จะ next ครั้งถัดไป



ใช้ removeNode ในคลาส LinkedList เพื่อลบปมก่อนหน้า nextReturn

รหัสที่ 12-11 ตัวแจงย้ำของคลาส LinkedList

ตัวแจงย้ำสำหรับตารางแฮช

ตัวแจงย้ำสำหรับการเก็บข้อมูลในตารางแฮช อาศัยการคืนเฉพาะช่องที่มีข้อมูล รหัสที่ 12-12 แสดงการให้บริการตัวแจงย้ำกับคลาส `QuadraticProbingHashMap` ให้คลาสนี้เป็น `Iterable` มี `iterator` คืนอ็อบเจกต์ของคลาส `Itr` ภายในมีตัวแปร `cursor` เก็บเลขช่องที่ได้ `next` ไปตัวล่าสุด (ตรงนี้ต่างกับของ `ArrayCollection` ที่ให้ `cursor` เก็บเลขช่องที่จะคืนตัวถัดไป) นอกจากนี้มีตัวแปร `numEntriesLeft` เก็บจำนวนข้อมูลที่ยังเหลือไม่ได้ถูก `next` กลับไป โดยมีค่าเริ่มต้นเท่ากับ `size` ซึ่งถูกลดลงหนึ่งทุกครั้งที่ `next` และใช้ใน `hashNext` ซึ่งคืนจริงถ้า `numEntriesLeft > 0` เมท็อด `next` ต้องค้นเริ่มที่ช่อง `cursor +1` หาจนพบช่องที่ไม่ใช่ `null` และไม่ใช่ `DELETED` เพื่อคืนคีย์กลับไป ส่วน `remove` ก็เพียงแค่เติม `DELETED` ใส่ในช่อง `cursor` ซึ่งคือเลขช่องที่ `next` ตัวล่าสุด โดยจะไม่อนุญาตให้ลบถ้าเป็นตอนเริ่มต้น (`cursor` มีค่าเป็น `-1`) หรือได้ลบตัวล่าสุดไปแล้ว (`table[cursor]` มีค่าเป็น `DELETED`)

```
public class QuadraticProbingHashMap implements Map, Iterable {
  private Entry[] table;
  private int size;
  ...
  public Iterator iterator() {
    return new Itr();
  }
  private class Itr implements Iterator {
    int numEntriesLeft = size;
    int cursor = -1;
    public boolean hasNext() {
      return numEntriesLeft > 0;
    }
    public Object next() {
      if (!hasNext()) throw new IllegalStateException();
      for(++cursor; cursor<table.length; ++cursor)
        if (table[cursor]!=null && table[cursor]!=DELETED) break;
      numEntriesLeft--;
      return table[cursor].key;
    }
    public void remove() {
      if (cursor == -1 || table[cursor] == DELETED)
        throw new IllegalStateException();
      table[cursor] = DELETED;
      --size;
    }
  }
```

cursor เก็บเลขช่องที่ได้ next ไปแล้วครั้งล่าสุด

numEntriesLeft เก็บจำนวน ข้อมูลที่ยังไม่ next

ค้นในตาราง หาช่องที่เก็บข้อมูลช่องถัดไป

รหัสที่ 12-12 ตัวแจงย้ำของคลาส `QuadraticProbingHashMap`

ตัวแจงย้ำสำหรับต้นไม้แบบทวิภาค

การทำงานของตัวแจงย้ำสำหรับต้นไม้แบบทวิภาคจะซับซ้อนกว่าที่ผ่านมา โดยจะเขียนตัวแจงย้ำให้กับคลาส BinaryTree เพื่อให้คลาสลูก ๆ หลาน ๆ เช่น BSTree, Expression, AVLTree ได้ใช้ด้วย แต่จะทำได้เพียงการแจงแจงข้อมูลในต้นไม้เท่านั้นไม่สามารถให้บริการลบได้ เนื่องจากการลบปมในต้นไม้แบบทวิภาคออก แล้วจะเปลี่ยนโครงสร้างไปเป็นอย่างไรนั้น เราไม่อาจทราบได้ในคลาส BinaryTree จึงเป็นหน้าที่ของคลาสลูกที่ต้องเพิ่ม remove ให้ตัวแจงย้ำเอง (ถ้าต้องการให้บริการ) สิ่งที่ต้องคำนึงอีกประการหนึ่งคือ จะแจกแจงข้อมูลออกมาในลำดับอย่างไร ข้อกำหนดของตัวแจงย้ำโดยกว้าง ๆ แล้วไม่ได้ระบุว่า จะต้องแจงตัวใดก่อน ตัวใดหลัง (ยกเว้นว่า ผู้ออกแบบต้องการกำหนดให้ชัดเจน เช่น ตัวแจงย้ำของรายการระบุว่า จะแจงข้อมูลตามลำดับจากต้นรายการไปยังท้ายรายการ) จากนี้ไปจะขอนำเสนอการเขียนตัวแจงย้ำให้กับคลาส BinaryTree ซึ่งแจงข้อมูลให้ได้ลำดับเช่นเดียวกับการแวะผ่านแบบหลังลำดับ และโดยเขียนให้รองรับตัวแจงย้ำของ BSTree ที่ขยายความสามารถของ BinaryTree ให้สามารถลบได้ในหัวข้อถัดไป

รหัสที่ 12-13 แสดงตัวแจงย้ำของคลาส BinaryTree เริ่มด้วยการเขียนให้คลาสนี้เป็น Iterable จากนั้นเพิ่มเมท็อด iterator (บรรทัดที่ 100) ที่คืนอ็อบเจกต์ของคลาส Itr มีตัวสร้างรับปม r ซึ่งแทนรากของต้นไม้ที่ต้องการแจง การทำงานของตัวแจงย้ำของปม r อาศัยตัวแจงย้ำของลูกต้นซ้ายและลูกต้นขวา ซึ่งสร้างเตรียมเก็บไว้ใน leftItr และ rightItr (บรรทัดที่ 110 และ 111) มีตัวแปร curItr ไว้เก็บตัวแจงย้ำที่กำลังใช้ เนื่องจากเราจะแจงข้อมูลแบบหลังลำดับ ดังนั้นจึงเริ่มให้ curItr=leftItr ก่อน (บรรทัดที่ 112) ถ้าแจงลูกต้นซ้ายหมดแล้วในเมท็อด next ก็เปลี่ยน curItr=rightItr (บรรทัดที่ 121) และถ้าแจงลูกต้นขวาหมดแล้วด้วย ก็ให้คืนข้อมูลที่ปม r (ตามข้อกำหนดแบบหลังลำดับ) แล้วถือโอกาสตั้งค่าให้ r=null (บรรทัดที่ 124 ถึง 126) เพื่อบอกว่าแจงข้อมูลหมดแล้ว ไว้ใช้ตรวจสอบได้ง่าย ๆ ใน hasNext (บรรทัดที่ 116) (การตรวจสอบเช่นนี้ยังครอบคลุมกรณีแจงข้อมูลในต้นไม้ว่างด้วย)

ต้องขอบอกว่า ตัวแจงย้ำในรหัสที่ 12-13 นี้ใช้ทั้งเนื้อที่และเวลาเป็น $\Theta(n)$ ตั้งแต่ตอนสร้างตัวแจงย้ำ ยังไม่ได้เริ่มแจงสักกะตัว ก็เสียเนื้อที่และเวลาเต็มที่แล้ว เพราะเราสร้างตัวแจงย้ำของลูกทั้งสองเตรียมไว้ในตัวสร้าง จึงส่งผลให้ต้องสร้างตัวแจงย้ำของหลาน ของเหลน ... เสมือนกับมีการสร้างต้นไม้ของตัวแจงย้ำที่มีรูปร่างเหมือนกับต้นไม้ที่ต้องการแจงข้อมูลนั่นเอง !!! เมื่อเป็นเช่นนี้ ทำไมไม่เขียนตัวแจงย้ำโดยเรียก toArray เก็บข้อมูลทุกตัวไว้ภายใน แล้วค่อย ๆ ปล่อยข้อมูลในแถวลำดับนี้ออกมาทีละตัว ทุกครั้งที่ next ถูกเรียก ซึ่งง่ายกว่า และเปลืองที่น้อยกว่า

```
public abstract class BinaryTree implements Iterable {
  protected Node root;

  protected static class Node {
    public Object element;
    public Node left;
    public Node right;
    ...
  }
  ...
  public Iterator iterator() {
    return new Itr(root);
  }
  protected class Itr implements Iterator {
    protected Node r;
    protected Itr leftItr, rightItr, curItr;

    protected Itr(Node r) {
      this.r = r;
      if (r != null) {
        leftItr = new Itr(r.left);
        rightItr = new Itr(r.right);
        curItr = leftItr;
      }
    }
    public boolean hasNext() {
      return (r != null);
    }
    public Object next() {
      if (!hasNext()) throw new IllegalStateException();
      if (!curItr.hasNext() && curItr == leftItr)
        curItr = rightItr;
      if (curItr.hasNext())
        return curItr.next();
      Object e = r.element;
      r = null;
      return e;
    }
    public void remove() {
      throw new UnsupportedOperationException();
    }
  }
```

สร้างตัวแจงย้ำสำหรับต้นไม้ที่มี root เป็นราก

สร้างตัวแจงย้ำของต้นซ้ายและต้นขวา และให้เริ่มแจงต้นซ้าย

r เป็น null แสดงว่าไม่มีข้อมูลเหลือ

ถ้าแจงต้นซ้ายหมดแล้ว ก็เปลี่ยนไปแจงต้นขวา

ยังแจงลูกไม่หมด ก็แจงต่อ

ถ้าแจงลูก ๆ หมดแล้ว ก็คืนข้อมูลที่ปม r และให้ r เป็น null

ไม่ให้บริการลบที่ BinaryTree

รหัสที่ 12-13 ตัวแจงย้ำของคลาส `BinaryTree` (แบบง่าย แต่เปลืองเนื้อที่)

ขอปรับปรุงรหัสที่ 12-13 ใหม่ให้ใช้เนื้อที่น้อยลงเท่าที่จำเป็น ดังแสดงในรหัสที่ 12-14 เราใช้เฉพาะ `curItr` ตัวเดียวไว้เก็บตัวแจงย้ำของลูกที่กำลังใช้งาน มีตัวแปร `state` เก็บสภาวะการทำงานที่มีอยู่ 5 สภาวะ ได้แก่ `'S'` แทนสภาวะเริ่มต้น, `'L'` และ `'R'` แทนเมื่อกำลังแจงข้อมูลในลูกซ้ายและลูกขวา และ `'E'` แทนเมื่อแจงข้อมูลหมดแล้ว การเปลี่ยนค่าของ `state` จะเป็นตามลำดับ

'S'→'L'→'R'→'E' ด้วยการปรับปรุงเช่นนี้ตัวแจงย้ำของลูกๆ จะถูกสร้างก็เมื่อต้องการเท่านั้น ตัวใดถูกใช้แจงจนหมดแล้ว ก็กลายเป็นขยะไป เพราะเราใช้ curItr เพียงตัวเดียวอ้างอิงตัวปัจจุบันที่กำลังใช้งาน ณ ขณะใดขณะหนึ่งจะมีตัวแจงย้ำในระบบอย่างมากเท่ากับความสูงของต้นไม้ ให้สังเกตว่าเรามีเมท็อด newIterator (บรรทัดที่ 130) ทำเพียงแค่สร้างและคืนตัวแจงย้ำ เพื่อให้เรียกใช้ในเมท็อด next (บรรทัดที่ 119 และ 120) ผู้อ่านอาจสงสัยว่า ทำไมไม่เขียน new Itr(r.left) และ new Itr(r.right) ที่สองบรรทัดนี้เลยก็สิ้นเรื่อง เหตุผลที่เขียนเช่นนี้เพราะถ้ามีการเขียนคลาสลูกของ Itr ก็สามารถเขียน newIterator ใหม่ เพื่อสร้างตัวแจงย้ำของตัวเอง ทำให้เมท็อด next สร้างตัวแจงย้ำได้ถูกประเภท (จะได้เห็นการเขียนคลาสลูกของ Itr ในหัวข้อถัดไป)

```
100   public Iterator iterator() {
101     return new Itr(root);
102   }
103   protected class Itr implements Iterator {
104     protected Node r;
105     protected char state = 'S';
106     protected Itr curItr = null;
107
108     protected Itr(Node r) {
109       this.r = r;
110       if (r == null) state = 'E';
111     }
112     public boolean hasNext() {
113       return state != 'E';
114     }
115     public Object next() {
116       if (!hasNext()) throw new IllegalStateException();
117       while (curItr == null || !curItr.hasNext()) {
118         switch (state) {
119           case 'S': state='L'; curItr=newIterator(r.left); break;
120           case 'L': state='R'; curItr=newIterator(r.right); break;
121           case 'R': state='E'; return r.element;
122           default : throw new AssertionError();
123         }
124       }
125       return curItr.next();
126     }
127     public void remove() {
128       throw new UnsupportedOperationException();
129     }
130     protected Itr newIterator(Node r) {
131       return new Itr(r);
132     }
133   }
```

'S' : เริ่มต้น,
'L' : กำลังแจงข้อมูลในลูกซ้าย,
'R' : กำลังแจงข้อมูลในลูกขวา,
'E' : แจงข้อมูลหมดแล้ว

ต้นไม้ว่าง แสดงว่าไม่มีข้อมูล

ยังเหลือ ถ้า state != 'E'

'S' → 'L' เริ่มใช้ตัวแจงต้นซ้าย
'L' → 'R' แจงต้นซ้ายหมด เริ่มใช้ตัวแจงต้นขวา
'R' → 'E' แจงต้นขวาหมด คืนข้อมูลที่ปม r

รหัสที่ 12-14 ตัวแจงย้ำของคลาส BinaryTree (รุ่นปรับปรุง)

ตัวแจงย้ำสำหรับต้นไม้ค้นหาแบบทวิภาค

เนื่องจากเราสร้างต้นไม้ค้นหาแบบทวิภาค (BSTree) จากต้นไม้แบบทวิภาค (BinaryTree) จึงสามารถใช้ตัวแจงย้ำของ BinaryTree ได้กับ BSTree แต่เนื่องจากเราต้องการให้ตัวแจงย้ำที่ BSTree มีเมท็อด remove ด้วย จึงต้องปรับปรุงเล็กน้อยดังแสดงในรหัสที่ 12-15 ด้วยการสร้างตัวแจงย้ำใหม่ที่ BSTree ให้เป็นคลาสลูกของ Itr ใน BinaryTree ชื่อว่า BItr (บรรทัดที่ 108) ซึ่งเขียนเพิ่มเฉพาะตัวสร้าง และเมท็อด remove ส่วนเมท็อด iterator ก็ต้องให้คืนอ็อบเจกต์ของ BItr แทน

เคล็ดลับของการเขียน remove ได้สั้นคือ เมื่อผู้ใช้แจงข้อมูลมาถึงข้อมูลที่ปม r แล้วต้องการลบ นั่นหมายความว่า ข้อมูลทั้งหลายในต้นไม้ย่อยที่มีปม r เป็นรากนั้นถูกแจงไปหมดแล้ว เพราะเราเขียนตัวแจงย้ำใน BinaryTree ให้แจงแบบหลังลำดับ (คือแจงลูก ๆ ของ r จนหมดก่อนแล้วค่อยคืนข้อมูลที่ r) ดังนั้นการลบข้อมูลที่ปม r โดยเกิดการเปลี่ยนโครงสร้างของต้นไม้เฉพาะที่ต้นไม้ย่อยที่มี r เป็นราก ย่อมไม่กระทบการแจงต้นไม้ส่วนอื่นในอนาคต (เพราะโครงสร้างที่อื่นไม่เปลี่ยน) ซึ่งก็ตรงกับพฤติกรรมของการลบข้อมูลของ BSTree ที่ได้นำเสนอมา (ถ้ายังจำได้ การลบข้อมูลที่ปม r ทำได้โดยนำข้อมูลตัวน้อยสุดในต้นไม้ย่อยที่เป็นลูกขวามาแทนในปม r แล้วไปลบตัวน้อยสุดนั้นทิ้ง แต่ถ้าปม r ไม่มีลูกขวา ก็ให้ลบปม r ทิ้ง ได้รากของลูกต้นซ้ายเป็นรากใหม่หลังการลบ)

ดังนั้นเราเขียน remove ให้เรียกอีกเมท็อดหนึ่งชื่อ removeR ที่คืนรากของต้นไม้หลังการลบ (บรรทัดที่ 113) หากเราใช้ตัวแจงย้ำของลูกในการ next ครั้งล่าสุด (state มีค่าเป็น 'L' หรือ 'R') ก็ผลักภาระการลบให้กับตัวแจงย้ำ curItr ได้ผลกลับมาเป็นรากใหม่ของลูกหลังการลบ ก็อย่าลืมว่าต้องตั้งรากใหม่นี้ให้กับ r.left หรือ r.right ขึ้นกับว่าเป็นตัวแจงย้ำลูกซ้าย หรือลูกขวาตามลำดับ (บรรทัดที่ 115 หรือ 116) แต่ถ้าข้อมูลที่คืนด้วย next ครั้งล่าสุดคือข้อมูลที่ปม r (state เป็น 'E') ก็ให้เรียก BSTree.this.remove [2] เพื่อลบข้อมูลที่ปม r ทิ้ง (บรรทัดที่ 117) ได้ผลลัพธ์เป็นรากใหม่หลังการลบ

ให้สังเกตด้วยว่า เราเขียนเมท็อด newIterator ใหม่ใน BItr เพื่อใช้แทนของคลาสพ่อ Itr ใน BinaryTree ด้วย โดยให้สร้างตัวแจงย้ำแบบ BItr แทน

---

[2] คลาสภายในสามารถเรียกเมท็อดของคลาสนอกที่ครอบอยู่ได้ เช่นใน BItr เราเรียกเมท็อดของ BSTree ได้ แต่ถ้าไปเรียกเมท็อดของคลาสนอกที่มีชื่อซ้ำกับคลาสภายใน ต้องเขียนให้เต็มยศ เช่น BSTree.this.remove หมายความว่าให้เรียก remove ของอ็อบเจกต์ this ของคลาส BSTree

```
public class BSTree extends BinaryTree {
  ...
  public Iterator iterator() {
    return new BItr(root);
  }
  private class BItr extends Itr {
    protected BItr(Node r) { super(r); }
    public void remove() {
      root = removeR();
    }
    private Node removeR() {
      switch (state) {
        case 'L': r.left = ((BItr)curItr).removeR(); break;
        case 'R': r.right = ((BItr)curItr).removeR(); break;
        case 'E': r = BSTree.this.remove(r, r.element); break;
        default : throw new AssertionError();
      }
      return r;
    }
    protected Itr newIterator(Node r) {
      return new BItr(r);
    }
  }
```

ตัวแจงย้ำนี้ยังมีปัญหาถ้าเรียก remove โดยที่ยังไม่ได้ next ขอละไว้เป็นแบบฝึกหัด

ถ้ากำลังใช้ตัวแจงย้ำของลูก ก็สั่งต่อให้ตัวแจงย้ำของลูกลบให้

override newIterator ของคลาส BinaryTree เพื่อให้สร้าง BItr ใช้ตอนแจงข้อมูลใน next

รหัสที่ 12-15 ตัวแจงย้ำของคลาส BSTree ซึ่งขยายความสามารถของ BinaryTree ให้ลบได้

ตัวแจงย้ำแบบขัดข้องอย่างเร็ว

ที่ผ่านมาเราเขียนตัวแจงย้ำ เพื่อแจกแจงข้อมูลให้ผู้ใช้ ภายใต้สมมติฐานว่า ที่เก็บข้อมูลนั้นนิ่ง ข้อมูลภายในไม่เปลี่ยนแปลง ไม่เพิ่ม ไม่ลด ถ้าจะเปลี่ยนได้ก็ต้องผ่านเมท็อดของด้วยตัวแจงย้ำเอง (remove ของตัวแจงย้ำจัดการลบเอง ก็ต้องระวังไม่ให้เกิดปัญหา) แต่ถ้าที่เก็บข้อมูลถูกเปลี่ยนแปลงด้วยวิธีอื่น ระหว่างการใช้ตัวแจงย้ำ อาจทำให้การแจงย้ำทำงานผิดพลาด เช่น ส่วนของโปรแกรมในรหัสที่ 12-16 ทางซ้าย จะแจง "1" ออกมาซ้ำสองครั้ง เพราะคำสั่ง c.add(0,"0") ไปแทรก "0" ไว้ต้นรายการ โดยที่ตัวแจงย้ำไม่รับทราบ ในขณะที่ส่วนของโปรแกรมทางขวา จะพลาดไม่แจง "2" ออกมา เพราะแทนที่จะลบด้วย i.remove() กลับไปเรียก c.remove(e)

```
ArrayList c = new ArrayList();
c.add("1"); c.add("2");
Iterator i = c.iterator();
i.next();
c.add(0,"0");
System.out.println(i.next());
```

```
ArrayList c = new ArrayList();
c.add("1"); c.add("2"); c.add("3");
Iterator i = c.iterator();
Object e = i.next();
c.remove(e);
System.out.println(i.next());
```

รหัสที่ 12-16 การเปลี่ยนแปลงระหว่างการแจงย้ำทำให้แจงข้อมูลซ้ำ (ซ้าย) แจงข้อมูลไม่ครบ (ขวา)

แล้วจะป้องกันหรือแก้ไขเหตุการณ์เช่นนี้ได้อย่างไร ? ในฐานะที่เป็นผู้ออกแบบตัวแจงย้ำ คงลำบากที่จะห้ามผู้เขียนโปรแกรมไม่ให้เขียนผิด และถ้าจะแก้ไขให้ตัวแจงย้ำรับรู้การเปลี่ยนแปลงของที่เก็บในทุกรูปแบบก็คงเป็นภาระที่หนัก สิ่งที่ทำได้ก็คือดักตรวจว่า มีเหตุการณ์เช่นนี้เกิดหรือไม่ขณะให้บริการการแจงย้ำ ถ้าพบ ให้รายงานทันทีว่า ตัวแจงย้ำเกิดเหตุขัดข้องไม่สามารถแจงข้อมูลได้ เราเรียกตัวแจงย้ำที่ทำงานในลักษณะเช่นนี้ว่า *แบบขัดข้องอย่างเร็ว* (fail-fast) ซึ่งทำได้โดยอาศัยการเพิ่มตัวแปร `modCount` ไว้ในตัวที่เก็บข้อมูล ตัวแปรนี้จะเพิ่มค่าทุกครั้งเมื่อที่เก็บข้อมูลเกิดการเปลี่ยนแปลง เมื่อใดมีการสร้างตัวแจงย้ำ ก็จะทำสำเนา `modCount` ของที่เก็บข้อมูลมาเก็บไว้ในตัวแปร `expectedModCount` ของตัวแจงย้ำด้วย เพียงแค่นี้ ทุกครั้งที่จะให้บริการการแจงย้ำ จะตรวจสอบว่า `expectedModCount` ของตัวแจงย้ำเหมือนกับ `modCount` ของที่เก็บข้อมูลหรือไม่ ถ้ายังเหมือนกัน ก็ทำงานได้ ถ้าไม่เหมือนกันแสดงว่า ที่เก็บข้อมูลมีการเปลี่ยนแปลงระหว่างการแจงย้ำ โดยที่เจ้าตัวไม่รับทราบ แสดงว่าเกิดเหตุขัดข้อง ให้โยนสิ่งผิดปกติทันที ดังแสดงในรหัสที่ 12-17

```
public class ArrayCollection implements Collection, Iterable {
  ...
  private int modCount;
  ...
  public void add(Object e) {
    ...
    ++modCount;
  }
  public void remove(Object e) {
      ...
      ++modCount;
  }
  public Iterator iterator() { return new Itr(); }
  private class Itr implements Iterator {
    int cursor = 0;
    int expectedModCount = modCount;
    public boolean hasNext() {
      checkForComodification();
      return cursor < size;
    }
    public Object next() {
      checkForComodification();
      if (!hasNext()) throw new IllegalStateException();
      return elementData[cursor++];
    }
    private void checkForComodification() {
      if (expectedModCount != modCount)
        throw new ConcurrentModificationException();
    }
    ...
```

modCount เพิ่มทุกครั้งที่มีการเปลี่ยนแปลง

เก็บ modCount ตอนสร้างไว้ตรวจสอบ

ตรวจสอบก่อนทำงาน

ตรวจสอบก่อนทำงาน

ไม่เท่า แสดงว่าที่เก็บข้อมูลเปลี่ยนแปลง

รหัสที่ 12-17 การใช้ `modCount` เพื่อดักตรวจการเปลี่ยนแปลงที่เก็บข้อมูลระหว่างการแจงย้ำ

สำหรับเมท็อด remove ของตัวแจงย้ำต้องทำเพิ่มอีกเล็กน้อย เนื่องจากเกิดการเปลี่ยนแปลงในที่เก็บข้อมูล จึงต้องเพิ่มค่าของ modCount และตั้งค่าของ expectedModCount ให้เหมือนค่าใหม่ของ modCount ด้วย ดังแสดงในรหัสที่ 12-18

```
public class ArrayCollection implements Collection, Iterable {
  ...
  private int modCount;
  ...
  private class Itr implements Iterator {
    ...
    int expectedModCount = modCount;
    ...
    public void remove() {
      checkForComodification();
      if (!removable) throw new IllegalStateException();
      elementData[--cursor] = elementData[--size];
      removable = false;
      expectedModCount = ++modCount;
    }
  }
}
```

ลบเอง รับรู้สภาพ แต่ต้องปรับ expectedModCount

รหัสที่ 12-18 การปรับ modCount และ expectedModCount ในเมท็อด remove

ในคลังคลาสจาวามีคลาสชื่อ CopyOnWriteArrayList (ใน java.util.concurrent) ทำหน้าที่คล้าย ArrayList แต่เราสามารถเปลี่ยนแปลงตัวรายการได้ ระหว่างการแจงย้ำ เขาใช้วิธีง่าย ๆ คือเมท็อดของคลาสนี้ที่เปลี่ยนแปลงข้อมูลในรายการ จะทำสำเนาของแถวลำดับที่เก็บข้อมูลชุดใหม่ให้เหมือนของเก่า แล้วค่อยเปลี่ยนในชุดใหม่นี้ เมื่อใดมีการสร้างตัวแจงย้ำจะใช้แถวลำดับปัจจุบันเป็นฐานในการแจงข้อมูล ดังนั้นการเปลี่ยนแปลงใด ๆ ในที่เก็บข้อมูลที่เกิดขึ้นระหว่างการแจงจะเกิดในแถวลำดับใหม่ จึงไม่ส่งผลมายังการทำงานของตัวแจงย้ำ ที่สร้างไว้ก่อนหน้านี้ เช่น รหัสที่ 12-16 จะไม่มีปัญหาถ้าหันมาใช้ CopyOnWriteArrayList ดังนี้

```
List c = new CopyOnWriteArrayList();
c.add("1"); c.add("2");
Iterator i = c.iterator();
System.out.println(i.next());
c.add(0,"0");
System.out.println(i.next());
```

และ

```
List c = new CopyOnWriteArrayList();
c.add("1"); c.add("2"); c.add("3");
Iterator i = c.iterator();
Object e = i.next(); System.out.println(e);
c.remove(e);
System.out.println(i.next());
```

จะแจงได้ "1" และ "2" ทั้งคู่ แต่ต้องยอมรับว่า การเปลี่ยนแปลงรายการจะทำงานช้ามาก ดังนั้นจึงเหมาะกับงานที่มีการแจงย้ำข้อมูลบ่อย ๆ แต่มีการเปลี่ยนแปลงบ้างเล็กน้อย ข้อมูลที่แจงอาจล้าสมัยบ้าง แต่ไม่กระเทือนระบบ เมื่อเป็นเช่นนี้ตัวแจงย้ำของ CopyOnWriteArrayList จึงไม่ให้บริการลบข้อมูล

แบบฝึกหัด

1. จงเขียนตัวแจงย้ำให้กับคลาสต่อไปนี้ (ให้บริการ `remove` ด้วย)
   1.1. `ArrayQueue`
   1.2. `SeparateChainingHashMap`
   1.3. `BinaryHeap`
2. ถ้าให้ `Collection` เป็นอินเทอร์เฟซลูกของ `Iterable` จงเขียนเมท็อดต่อไปนี้ ให้กับคลาส `ArrayCollection` โดยใช้ตัวแจงย้ำเป็นกลไกหลักในการทำงาน
   2.1. `void clear()` เพื่อล้างคอลเล็กชันให้ไม่มีข้อมูลเหลืออยู่เลย
   2.2. `void removeAll(Object e)` เพื่อลบ `e` ทุกตัวในคอลเล็กชันออกให้หมด
   2.3. `boolean containsAll(Collection c)` เพื่อตรวจสอบว่า คอลเล็กชันนี้มีข้อมูลทุกตัวที่ `c` มีหรือไม่ เช่น ถ้า `c1` เก็บ [A, B, C, A, D] และ `c2` เก็บ [A, B, B, A] จะได้ `c1.containsAll(c2)` คืนค่า `true` แต่ `c2.containsAll(c1)` คืนค่า `false`
3. คลาส `BItr` ในรหัสที่ 12-15 ยังมีปัญหาเรื่องลบ เช่น สั่งให้ลบทั้ง ๆ ที่ยังไม่ `next` หรือการสั่งลบสองครั้งติดกัน จงแก้ไขให้ทำงานถูกต้อง
4. ในคลังคลาสมาตรฐานจาวามีอินเทอร์เฟซ `ListIterator` ให้กับรายการ โดยตัวแจงย้ำแบบนี้มีบริการเพิ่มและแจงถอยหลัง จงศึกษาอินเทอร์เฟซนี้และปรับปรุงให้คลาส `LinkedList` และ `SinglyLinkedList` ให้บริการตัวแจงย้ำแบบนี้
5. เราใช้ตัวแจงย้ำหลายตัวแจงข้อมูลจากที่เก็บข้อมูลเดียวกันได้หรือไม่ จะได้ในสถานการณ์อย่างไร จะไม่ได้ในสถานการณ์อย่างไร
6. จงให้ความเห็นว่า โครงสร้างข้อมูลที่เราได้นำเสนอมาตั้งแต่บทที่ 1 ตัวใดบ้างเขียนตัวแจงย้ำได้ง่าย ตัวใดเขียนยาก ตัวใดไม่ควรต้องมีตัวแจงย้ำ (ตัวแจงย้ำที่กล่าวถึงนี้เป็นแบบมีบริการ `remove` สำหรับลบข้อมูลด้วย)
7. จงเขียนเมท็อด `inorderIterator`, `preorderIterator` ให้กับคลาส `BinaryTree` เพื่อคืนตัวแจงย้ำที่แจกแจงข้อมูลในต้นไม้แบบทวิภาคที่แจกแจงแบบตามลำดับและก่อนลำดับ

8. แนวคิดของตัวแจงย้ำไม่ได้จำกัดอยู่แค่การแจกแจงข้อมูลจากโครงสร้างข้อมูลเท่านั้น เราสามารถให้บริการตัวแจงย้ำกับแหล่งผลิตข้อมูลแบบอื่น เพื่อให้ผู้ใช้เรียกใช้ได้ง่าย จงเขียนคลาสชื่อ `IteratbleTextFile` ซึ่ง implements `Iterable` ที่มีตัวสร้างรับชื่อของแฟ้มข้อความ (text file) โดยตัวแจงย้ำที่ได้จะอ่านแฟ้มและคืนข้อความจากแฟ้มกลับมาทีละบรรทัด ตัวอย่างเช่น โปรแกรมข้างล่างนี้ตรวจสอบว่า แฟ้มชื่อ `"c:\data.txt"` มีกี่บรรทัดที่มีคำว่า `"java"`

```
public class Test {
  public static void main(String[] args) {
    int c = 0;
    IterableTextFile f = new IterableTextFile("c:\\data.txt");
    for (Iterator itr = f.iterator(); itr.hasNext();) {
      String text = (String) itr.next();
      if (text.indexOf("java") >= 0) c++;
    }
    f.close();
    System.out.println("มี " + c + " บรรทัด ที่มีคำว่า java");
  }
}
```

13 การเรียงลำดับข้อมูล

การเรียงลำดับข้อมูลเป็นกระบวนการที่สำคัญและต้องทำเป็นประจำในการประมวลผลข้อมูล เนื่องจากข้อมูลที่เรียงลำดับอย่างมีระเบียบ มักทำให้การตีความ การหาความสัมพันธ์ของข้อมูลต่าง ๆ กระทำได้ง่ายขึ้น โดยทั่วไปคลังคำสั่งในระบบพัฒนาโปรแกรมจะมีบริการการเรียงลำดับข้อมูลให้ผู้เขียนโปรแกรมเรียกใช้ได้ แต่ก็ไม่จำเป็นเสมอไปว่า บริการที่คลังคำสั่งมีจะเหมาะกับกลุ่มข้อมูลที่เราต้องการเรียงลำดับ การศึกษาขั้นตอนวิธีการเรียงลำดับข้อมูลซึ่งมีอยู่มากมายหลากหลายวิธีจึงเป็นเรื่องสำคัญ เพื่อให้เข้าใจแนวคิดและประสิทธิภาพการทำงาน รวมถึงจุดเด่นจุดด้อยของขั้นตอนวิธีดังกล่าว บทนี้จะนำเสนอขั้นตอนวิธีพื้นฐานสำหรับการเรียงลำดับข้อมูลที่ควรรู้ ได้แก่ การเรียงลำดับแบบเลือก แบบฟอง แบบแทรก แบบเชลล์ แบบฮีป แบบผสาน และแบบเร็ว

ข้อกำหนด

ก่อนจะลงรายละเอียดของแต่ละขั้นตอนวิธี ต้องขอระบุข้อกำหนดของการเรียงลำดับข้อมูลที่จะนำเสนอกันก่อน โดยทั่วไปรายการของข้อมูลที่นำมาเรียงลำดับมีทั้งแบบเก็บในหน่วยความจำหลัก (เรียกว่าแบบภายใน) และเก็บในหน่วยความจำสำรอง เช่น ฮาร์ดดิกส์ (เรียกว่าแบบภายนอก) สำหรับแบบภายในนั้นเราเก็บรายการข้อมูลได้ทั้งในแถวลำดับและในรายการโยง ในบทนี้จะสนใจเฉพาะการเรียงลำดับข้อมูลที่ได้รับในแถวลำดับ โดยให้ผลที่ได้เรียงจากซ้ายไปขวาในแถวมีค่าจากน้อยไปมาก (ในลำดับที่ค่าของข้อมูลไม่ลดลง : nondecreasing order) นอกจากนี้เราจะเรียงลำดับด้วยการย้ายหรือสลับข้อมูล โดยอาศัยผลการเปรียบเทียบข้อมูลระหว่างกันทีละคู่ ๆ ว่าใครน้อยกว่า เท่ากัน หรือมากกว่า ถ้าข้อมูลเป็นแบบพื้นฐาน (เช่น `short`, `int`, `long`, `double` เป็นต้น) ก็เปรียบเทียบข้อมูลได้แทบทุกประเภท ยกเว้นเฉพาะ `boolean` ในกรณีที่เป็นอ็อบเจกต์ ก็ต้องเป็นแบบ `Comparable` เราได้

เห็นการเรียงลำดับแบบฐานกันในบทที่ 7 ซึ่งอาศัยการแยกข้อมูลออกเป็นส่วน ๆ มาใช้ในการเรียงลำดับ โดยไม่มีการเปรียบเทียบข้อมูลระหว่างกันเอง ซึ่งไม่ใช่แบบที่เราจะสนใจในบทนี้

เพื่อให้สามารถนำเสนอขั้นตอนวิธีได้อย่างกระชับ ขอเขียนเมท็อด `lessThan` และ `swap` เพื่อจะได้เรียกใช้อย่างสะดวกดังแสดงในรหัสที่ 13-1 `lessThan(a,b)` มีไว้เปรียบเทียบว่า `a` น้อยกว่า `b` หรือไม่ โดยมอง `a` เป็น `Comparable` แล้วค่อยเรียก `compareTo(b)` ส่วน `swap` มีไว้สลับข้อมูลช่องที่ `i` และ `j` ของแถวลำดับ `d` ที่ได้รับ

```java
public class ArrayUtil {
  private static boolean lessThan(Object a, Object b) {
    return ((Comparable) a).compareTo(b) < 0;
  }
  private static void swap(Object[] d, int i, int j) {
    Object t = d[i];
    d[i] = d[j];
    d[j] = t;
  }
```

คืนจริง ถ้า a มีค่าน้อยกว่า b

สลับข้อมูลในช่องที่ i กับ j ของแถวลำดับ d

รหัสที่ 13-1 เมท็อด `lessThan` เพื่อเปรียบเทียบข้อมูล และ `swap` เพื่อสลับข้อมูลในแถวลำดับ

ถ้า n แทนจำนวนข้อมูล โดยทั่วไปเราประเมินการเรียงลำดับข้อมูลด้วยเวลาการทำงาน ซึ่งในการวิเคราะห์เชิงทฤษฎีแล้วมักวัดกันด้วยจำนวนครั้งของการเปรียบเทียบ (ด้วย `lessThan`) ว่าต้องทำกี่ครั้งจึงได้ข้อมูลที่เรียงลำดับ ซึ่งพอจัดได้เป็นสามกลุ่มคือ แบบที่ใช้เวลาเป็น $O(n^2)$, $O(n \log n)$ และแบบที่ดีกว่าแบบแรกแต่แย่กว่าแบบหลังคือเป็น $O(n^{1.xx})$, เราจะแสดงให้เห็นว่า ขั้นตอนวิธีการเรียงลำดับใด ๆ ที่อาศัยการเปรียบเทียบข้อมูลต้องเปรียบเทียบเป็นจำนวน $\Omega(n \log_2 n)$ ครั้ง นั่นหมายความว่า ในทางทฤษฎี แบบที่ใช้เวลาเป็น $O(n \log n)$ ถือได้ว่าดีที่สุดแล้ว นอกจากนี้เรายังประเมินการเรียงลำดับกันด้วยปริมาณเนื้อที่หน่วยความจำเสริมที่ใช้ระหว่างการเรียงลำดับซึ่งพอจัดได้เป็นสามกลุ่มคือ ที่ใช้เนื้อที่เสริมอีก $O(1)$, $O(\log n)$, และ $O(n)$ เมื่อต้องเรียงลำดับข้อมูลจำนวนมาก ๆ ปัจจัยนี้จะมีผลมากต่อการทำงาน

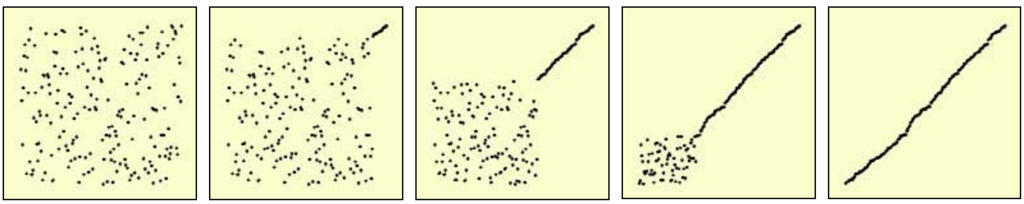
โดยทั่วไปข้อมูลที่นำมาเรียงลำดับประกอบด้วยข้อมูลย่อยหลายส่วนโดยอาศัยข้อมูลบางส่วน (เรียกว่าคีย์) เป็นตัวกำหนดลำดับของข้อมูล *การเรียงลำดับแบบเสถียร* (stable sort) คือการเรียงลำดับซึ่งข้อมูลที่มีคีย์เหมือนกันจะมีลำดับสัมพัทธ์เหมือนกันทั้งก่อนและหลังการเรียงลำดับ เช่น ข้อมูลขาเข้าคือ (3, ☺), (2, ☹), (3, 😐) โดยใช้ตัวเลขเป็นคีย์ เมื่อเรียงแล้วได้ (2, ☹), (3, ☺), (3, 😐) ถือว่าเสถียร เพราะ (3, ☺) เคยอยู่ทางซ้ายของ (3, 😐) ตอนเริ่มต้น ก็ยังเหมือนเดิมหลังเรียงแล้ว แต่ถ้าได้ผลเป็น (2, ☹), (3, 😐), (3, ☺) จะไม่เสถียร ในงานบางประเภทต้องการความเสถียรของการเรียงลำดับ ซึ่งขั้นตอนวิธีการเรียงลำดับบางวิธีสนองความต้องการนี้ได้ง่าย ๆ ในขณะที่บางวิธีทำได้ลำบาก

การเรียงลำดับแบบเลือก

การเรียงลำดับแบบเลือก (selection sort) น่าจะเป็นการเรียงลำดับที่มีแนวคิดง่ายสุด อาศัยการ เลือกข้อมูลตัวที่มีค่ามากสุดในกลุ่ม นำไปสลับกับข้อมูลตัวหลังสุดในกลุ่ม กระทำเช่นนี้ไปเรื่อย ๆ โดยลดขนาดของกลุ่มลงทีละหนึ่ง จนเหลือเพียงตัวเดียวก็เป็นอันเรียงลำดับเสร็จ ดังตัวอย่างในรูปที่ 13-1 แถวแรกแสดงข้อมูลของแถวลำดับตอนเริ่มต้น มีข้อมูลอยู่ 5 ตัว หาตัวมากสุดได้ค่า 55 อยู่ช่องที่ 3 ก็นำไปสลับกับช่องที่ 4, เหลือข้อมูล 4 ตัว หาตัวมากสุดได้ค่า 32 อยู่ช่องที่ 0 นำไปสลับกับช่องที่ 3, กระทำการหาตัวมากสุดเพื่อสลับกับตัวท้ายของกลุ่มที่เหลือ จะได้ผลลัพธ์ดังแถวล่างสุด ขอแสดงการทำงานในอีกลักษณะหนึ่ง รูปที่ 13-2 แสดงภาพการเปลี่ยนแปลงของข้อมูลระหว่างการเรียงลำดับแบบเลือก แต่ละจุดบนระนาบในรูปแทนข้อมูลหนึ่งตัว พิกัด x ของจุดแทนตำแหน่งของข้อมูลนั้นในแถวลำดับ และพิกัด y แทนค่าของข้อมูลนั้น เมื่อเรียงลำดับเสร็จแล้วจะได้จุดต่าง ๆ เรียงเป็นเส้นทแยงมุมทอดยาวจากมุมขวาบนลงมายังมุมซ้ายล่างดังแสดงในรูปขวาสุด จะเห็นได้ว่า ข้อมูลตัวมากสุดถูกสลับไปอยู่ทางขวาสุดของกลุ่ม จึงเห็นเป็นเส้นจากมุมขวาบนยาวลงมาเรื่อย ๆ จนในที่สุดเป็นเส้นแทนข้อมูลที่เรียงลำดับเรียบร้อย

0	1	2	3	4
32	23	11	55	18
32	23	11	18	**55**
18	23	11	**32**	**55**
18	11	**23**	**32**	**55**
11	**18**	**23**	**32**	**55**

รูปที่ 13-1 ตัวอย่างการเรียงลำดับแบบเลือก



รูปที่ 13-2 ภาพแสดงการเปลี่ยนแปลงของข้อมูลระหว่างการเรียงลำดับแบบเลือก

รหัสที่ 13-2 แสดงเมท็อดเรียงลำดับข้อมูลแบบเลือก วงวน `for` ในบรรทัดที่ 11 มีตัวแปร `k` เก็บจำนวนข้อมูลในกลุ่ม (แปรเริ่มจากจำนวนข้อมูลทั้งหมด ลดลงจนเหลือ 2) มีวงวน `for` ภายในอีกวงเพื่อหาเลขช่องที่มีข้อมูลมากสุดในกลุ่มตั้งแต่ `d[0]` ถึง `d[k-1]` เมื่อได้ค่ามากสุดก็สลับกับตัวขวาสุดของกลุ่ม (บรรทัดที่ 16) จบการเรียงลำดับเมื่อเหลือข้อมูลเพียงตัวเดียว (`k=1`)

```
10   public static void selectionSort(Object[] d) {
11     for (int k = d.length; k > 1; k--) {
12       int m = 0;
13       for (int j = 1; j < k; j++) {
14         if (lessThan(d[m], d[j])) m = j;
15       }
16       swap(d, m, k-1);
17     }
18   }
```

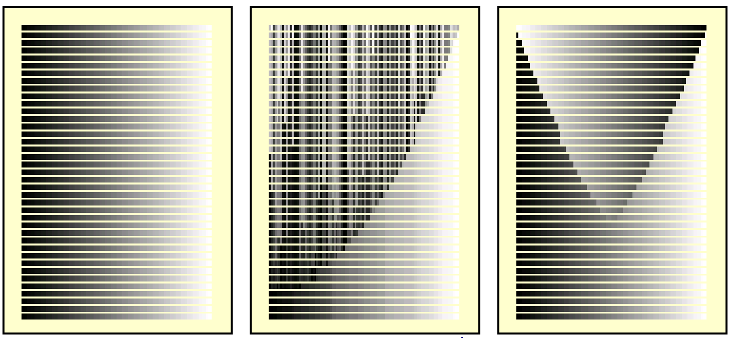
m คือช่องที่มีค่ามากสุดตั้งแต่ d[0] ถึง d[k-1]

k-1 คือช่องขวาสุดของกลุ่ม

รหัสที่ 13-2 การเรียงลำดับแบบเลือก (selection sort)

เวลาการทำงานสามารถวิเคราะห์ได้จากการนับจำนวนการเปรียบเทียบข้อมูล (บรรทัดที่ 14) วงวน `for` วงนอกหมุนเป็นจำนวน $n-1$ รอบ วงวนภายในจะหมุนเพื่อเปรียบเทียบเป็นจำนวน $k-1$ ครั้ง สรุปได้ว่า มีการเปรียบเทียบเป็นจำนวนทั้งสิ้น $\sum_{k=2}^{n}(k-1) = n(n-1)/2 = \Theta(n^2)$ ให้สังเกตว่า ไม่ว่าข้อมูลในแถวลำดับจะมีลักษณะเช่นใด การเรียงลำดับแบบเลือกจะวนทำงานเป็นจำนวนรอบเท่ากันเสมอ โดยเปรียบเทียบทั้งหมด $n(n-1)/2$ ครั้ง และเกิดการสลับข้อมูลทั้งสิ้น $n-1$ ครั้งเสมอ

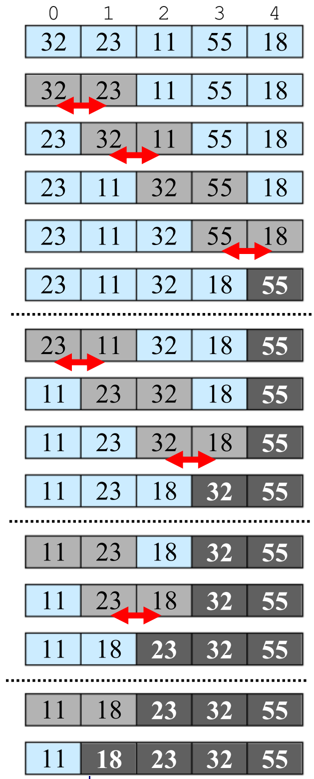
รูปที่ 13-3 แสดงภาพการเปลี่ยนแปลงข้อมูลในอีกลักษณะ ข้อมูลในแถวลำดับถูกแสดงเป็นแถบสีเทาแนวนอนหนึ่งแถบ ข้อมูลที่มีค่ามากแทนด้วยสีเทาอ่อน (สีขาวแทนค่ามากสุด) ส่วนค่าน้อยแสดงด้วยสีเทาเข้ม (สีดำแทนค่าน้อยสุด) แถบบนแทนข้อมูลขาเข้า แถบล่างแทนข้อมูลที่เรียงลำดับแล้ว แถบระหว่างกลางแทนการเปลี่ยนแปลงของข้อมูลในแถวลำดับตามเวลาระหว่างการทำงาน โดยแต่ละแถบถูกแสดงออกมาในช่วงเวลาที่เท่า ๆ กัน (ดังนั้นจำนวนแถบมากแสดงว่า ใช้เวลาเรียงลำดับนานกว่าจำนวนแถบน้อย) รูปนี้แสดงการเรียงลำดับที่มีลักษณะของข้อมูลขาเข้าต่างกันสามแบบ รูปทางซ้ายแทนข้อมูลขาเข้าที่เรียงลำดับอยู่แล้ว รูปกลางแทนข้อมูลขาเข้าที่มีลักษณะสุ่ม และรูปขวาแทนข้อมูลขาเข้าที่เรียงกลับลำดับคือจากมากไปน้อย รูปทั้งสามมีความสูงเท่ากันยืนยันผลการวิเคราะห์ว่าใช้เวลาการทำงานเท่ากันหมด อยากให้ผู้อ่านลองตีความการเปลี่ยนของแถบสีเทียบกับขั้นตอนการทำงานของการเรียงลำดับแบบเลือกที่ได้นำเสนอมา



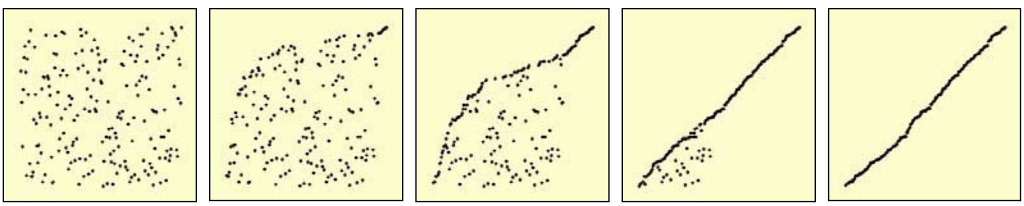
รูปที่ 13-3 แถบสีแสดงการเรียงลำดับแบบเลือกกับข้อมูลที่เรียงแล้ว แบบสุ่ม และแบบกลับลำดับ

การเรียงลำดับแบบฟอง

การเรียงลำดับแบบฟอง (bubble sort) อาศัยการเปรียบเทียบข้อมูลตัวติดกันว่า กลับลำดับกันหรือไม่ ถ้าใช่ก็สลับที่เก็บกัน กระทำการเปรียบเทียบและสลับข้อมูลไล่ตั้งแต่ซ้ายไปขวา หนึ่งรอบได้ตัวมากสุดมาอยู่หลังสุด ทำอีกรอบ (โดยลดจำนวนข้อมูลลงหนึ่ง) ก็จะได้ตัวมากสุดของที่เหลือย้ายมาอยู่ช่องขวาสุดของกลุ่มที่เหลือ ทำเช่นนี้ $n-1$ รอบจนเหลือข้อมูลตัวเดียว จะได้ข้อมูลในแถวลำดับเรียงลำดับตามต้องการ (ที่เรียกว่าแบบฟอง เพราะมองข้อมูลตัวมากค่อย ๆ เคลื่อนไปทางขวา เสมือนฟองอากาศในน้ำค่อย ๆ ปุดลอยไปที่ตำแหน่งที่ควรอยู่) รูปที่ 13-4 แสดงตัวอย่างการทำงาน แถวแรกแสดงข้อมูลขาเข้า มีข้อมูลอยู่ 5 ตัว, รอบที่ 1 เปรียบเทียบข้อมูลคู่ที่ติดกัน 4 ครั้ง เกิดการสลับ 3 ครั้งได้ 55 อยู่ช่องขวาสุด, รอบที่ 2 เปรียบเทียบคู่ที่ติดกัน 3 ครั้ง เกิดการสลับ 2 ครั้ง ได้ 32 อยู่ช่อง 3, รอบที่ 3 เปรียบเทียบคู่ที่ติดกัน 2 ครั้ง เกิดการสลับ 1 ครั้ง ได้ 23 อยู่ช่อง 2, รอบสุดท้ายเหลือคู่เดียว ไม่มีการสลับ ได้ข้อมูลเรียงลำดับเรียบร้อย รูปที่ 13-5 แสดงภาพการเปลี่ยนแปลงของข้อมูลระหว่างการเรียงลำดับแบบฟอง ให้สังเกตว่า มีลักษณะคล้ายการเรียงลำดับแบบเลือกในหัวข้อที่แล้ว นั่นคือข้อมูลตัวมากสุดทอดยาวจากมุมขวาบนลงมายังมุมซ้ายล่าง แต่ในขณะเดียวกันตัวน้อยกว่าก็ขยับมาทางซ้ายเข้าใกล้ตำแหน่งที่ควรอยู่ด้วย (สังเกตจากการที่ข้อมูลเริ่มย้ายมาอยู่ในแนวทแยงมากขึ้น ๆ)



รูปที่ 13-4 bubble sort



รูปที่ 13-5 ภาพแสดงการเปลี่ยนแปลงของข้อมูลระหว่างการเรียงลำดับแบบฟอง

รหัสที่ 13-3 แสดงเมท็อดเรียงลำดับข้อมูลแบบฟอง วงวน `for` ในบรรทัดที่ 22 มีตัวแปร `k` เก็บจำนวนข้อมูลในกลุ่ม (แปรเริ่มจากจำนวนข้อมูลทั้งหมด ลดลงจนเหลือ 2) มีวงวน `for` ภายในอีกวง ดูข้อมูลติดกันทีละคู่ตั้งแต่คู่ซ้ายสุด `d[0]` กับ `d[1]`, ตามด้วย `d[1]` กับ `d[2]` ไล่ไปจนถึงคู่

`d[k-2]` กับ `d[k-1]` คู่ใดกลับลำดับกัน ก็ให้สลับที่กัน (บรรทัดที่ 22) จบการเรียงลำดับเมื่อเหลือข้อมูลเพียงตัวเดียว (`k=1`)

```
19   public static void bubbleSort(Object[] d) {
20     for (int k = d.length; k > 1; k--) {
21       for (int j = 1; j < k; j++) {
22         if (lessThan(d[j], d[j-1])) swap(d, j-1, j);
23       }
24     }
25   }
```

ถ้าตัวติดกันกลับลำดับ ก็สลับตำแหน่งกัน

รหัสที่ 13-3 การเรียงลำดับแบบฟอง (bubble sort)

รหัสที่ 13-3 มีวงวน `for` เหมือนกับของรหัสที่ 13-2 ที่เราได้วิเคราะห์มาแล้ว ดังนั้นการเรียงลำดับแบบฟองจึงวนทำงานเป็นจำนวน $n(n-1)/2$ รอบ ใช้เวลาเป็น $\Theta(n^2)$ เช่นเดียวกัน ในกรณีที่รับข้อมูลที่เรียงลำดับอยู่แล้ว จะไม่เกิดการสลับข้อมูลเลย แต่ถ้าเริ่มด้วยข้อมูลที่เรียงกลับลำดับ ข้อมูลทุกคู่ที่เปรียบเทียบกันจะกลับลำดับกันหมด ต้องสลับทุกครั้ง เป็นจำนวนทั้งสิ้น $n(n-1)/2$ ครั้ง

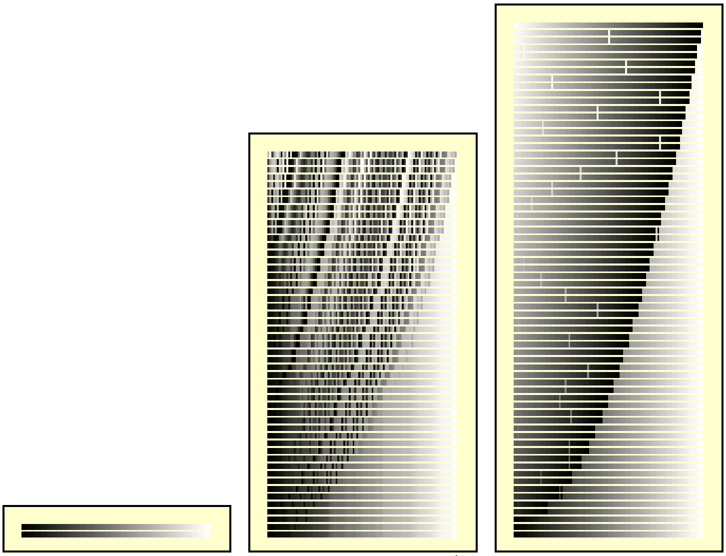
กำหนดให้*จำนวนกลับลำดับ* (number of inversions) คือจำนวนคู่ในรายการที่กลับลำดับกัน นั่นคือ `d[i]>d[j]` เมื่อ `i<j` เช่น รายการ $\langle 4, 2, 1, 3, 5\rangle$ มีจำนวนกลับลำดับเป็น 4 เพราะคู่ที่กลับลำดับมี 4 คู่คือ (4,2), (4,1), (4,3), และ (2,1) รายการที่เรียงลำดับแล้วจะมีจำนวนกลับลำดับเป็น 0 ในขณะที่รายการที่เรียงกลับลำดับจะมีจำนวนกลับลำดับเป็น $n(n-1)/2$ เพราะว่า ชุดข้อมูลมีจำนวน $C(n,2)$ คู่และทุกคู่กลับลำดับหมด เนื่องจากการเรียงลำดับแบบฟองสลับเฉพาะข้อมูลที่ติดกัน ดังนั้นการสลับข้อมูลหนึ่งครั้ง ทำให้จำนวนกลับลำดับลดลงหนึ่ง จึงต้องสลับข้อมูลเป็นจำนวนครั้งเท่ากับจำนวนกลับลำดับของรายการข้อมูลตอนเริ่มต้น จากข้อสังเกตนี้เอง หากการทำงานในวงวน `for` ภายในไม่เกิดการสลับข้อมูลเลย แสดงว่าข้อมูลเรียงลำดับแล้ว จึงสามารถหยุดการทำงานได้ รหัสที่ 13-4 แสดงการปรับปรุงรหัสที่ 13-3 ให้หยุดการทำงานเมื่อพบว่า ข้อมูลในแถวลำดับเรียงลำดับแล้ว

```
public static void bubbleSort(Object[] d) {
  for (int k = d.length; k > 1; k--) {
    boolean sorted = true;
    for (int j = 1; j < k; j++) {
      if (lessThan(d[j], d[j-1])) {
        swap(d, j-1, j);
        sorted = false;
      }
    }
    if (sorted) break;
  }
}
```

มีการสลับข้อมูลแสดงว่ายังไม่เรียง

ถ้าเรียงลำดับแล้วก็เลิกได้

รหัสที่ 13-4 การเรียงลำดับแบบฟอง (รุ่นปรับปรุง)



รูปที่ 13-6 แถบสีแสดงการเรียงลำดับแบบฟองกับข้อมูลที่เรียงแล้ว แบบสุ่ม และแบบกลับลำดับ

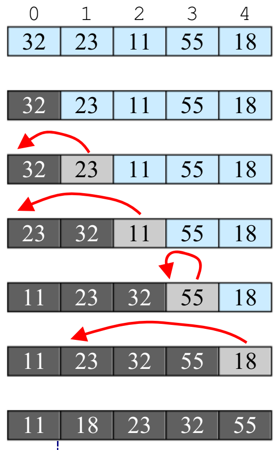
รูปที่ 13-6 แสดงการเปลี่ยนแปลงของข้อมูลในแถวลำดับด้วยแถบสี ระหว่างการเรียงลำดับแบบฟองซึ่ง (ด้วยรหัสที่ 13-4) รับข้อมูลเริ่มต้นในลักษณะต่างกัน รูปซ้ายทำงานได้เร็วสุด เพราะข้อมูลเรียงอยู่แล้ว ในขณะที่รูปขวาทำงานช้าสุดซึ่งคือกรณีที่ได้รับข้อมูลที่กลับลำดับ ส่วนรูปกลางนั้นข้อมูลเริ่มต้นมีลักษณะสุ่ม แต่ก็ใช้เวลาค่อนข้างใกล้กับกรณีช้าสุด

ถึงแม้ว่า การเรียงลำดับแบบฟองจะเป็นวิธีที่ได้รับความนิยมในการนำเสนอเป็นตัวอย่างกันมากมาย โดยเฉพาะอย่างยิ่งในวิชาการเขียนโปรแกรมเบื้องต้น แต่วิธีนี้กลับเป็นวิธีที่ทำงานช้ามาก เรียงลำดับข้อมูลสุ่มใช้เวลาพอ ๆ กับกรณีข้อมูลกลับลำดับ ในทางปฏิบัติพบว่า ช้ากว่าการเรียงลำดับแบบเลือกด้วยซ้ำ (จะได้เห็นผลการทดลองกันในท้ายบท) ได้มีผู้พยายามปรับปรุงให้เร็วขึ้น โดยเปลี่ยนจากการผลักตัวมากสุดไปทางขวาเพียงทิศเดียวในแต่ละรอบ ให้กลายเป็นผลักตัวมากสุดไปทางขวา สลับกับการผลักตัวน้อยสุดมาทางซ้าย ซึ่งมีชื่อเรียกหลากหลาย เช่น *แบบฟองสองทิศทาง* (bidirectional bubble), แบบ*กระสวย* (shuttle) หรือแบบเขย่า (shaker) เป็นต้น ซึ่งได้ผลดีขึ้น

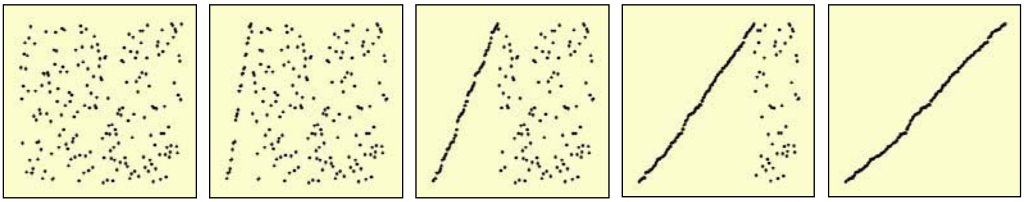
การเรียงลำดับแบบแทรก

การเรียงลำดับแบบแทรก (insertion sort) อาศัยแนวคิดการนำข้อมูลตัวใหม่ที่สนใจไปแทรกในรายการของข้อมูลทางซ้ายที่เรียงลำดับอยู่แล้ว ให้ได้รายการยาวขึ้นที่ยังคงเรียงลำดับ ทำการแทรกข้อมูลตัวใหม่ในลักษณะเช่นนี้จนข้อมูลหมด ก็ย่อมได้ข้อมูลทั้งหมดที่เรียงลำดับ รูปที่ 13-7 แสดงตัวอย่างการทำงาน แถวแรกแสดงข้อมูลขาเข้า มีข้อมูลอยู่ 5 ตัว, รอบที่ 1 เริ่มสนใจนำข้อมูลที่ช่อง 1

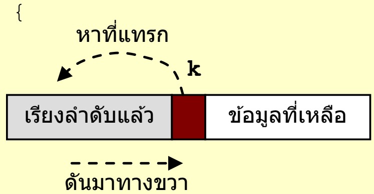
ซึ่งคือ 23 ไปหาที่แทรกในกลุ่มข้อมูลทางซ้ายของมันให้เรียงลำดับ พบว่า ต้องเลื่อน 32 มาทางขวาหนึ่งช่องแล้วจึงนำ 23 ไปใส่ในช่อง 0 ซึ่งคือการแทรกตัวใหม่ไว้ด้านหน้าได้ 23, 32, รอบที่ 2 สนใจช่อง 2 คือ 11 ต้องแทรกที่ช่อง 0 ได้เป็น 11, 23, 32, รอบที่ 3 สนใจช่อง 3 คือ 55 ไม่ย้ายไปไหน อยู่ช่องเดิม (เพราะมีค่ามากกว่าทุกตัวทางซ้าย), รอบที่ 4 สนใจช่อง 4 คือ 18 ต้องแทรกที่ช่อง 1 เกิดการย้าย 23, 32, และ 55 มาทางขวา แล้วนำ 18 ไปใส่ในช่อง 1 ได้ข้อมูลทั้งหมดเรียงลำดับ รูปที่ 13-8 แสดงภาพการเปลี่ยนแปลงของข้อมูลระหว่างการเรียงลำดับแบบแทรก ให้สังเกตว่า ข้อมูลทางซ้ายที่เรียงลำดับแล้ว จะค่อย ๆ มีขนาดใหญ่ขึ้น ๆ เนื่องจากเราค่อย ๆ นำข้อมูลตัวถัดไปมาแทรกในกลุ่มข้อมูลทางซ้ายให้เรียงลำดับ



รูปที่ 13-7 insertion sort


รูปที่ 13-8 ภาพแสดงการเปลี่ยนแปลงของข้อมูลระหว่างการเรียงลำดับแบบแทรก

```
31  public static void insertionSort(Object[] d) {
32    for (int k = 1; k < d.length; k++) {
33      Object t = d[k];
34      int j = k - 1;
35      while (j >= 0 && lessThan(t, d[j])) {
36        d[j + 1] = d[j];
37        j--;
38      }
39      d[j + 1] = t;
40    }
41  }
```



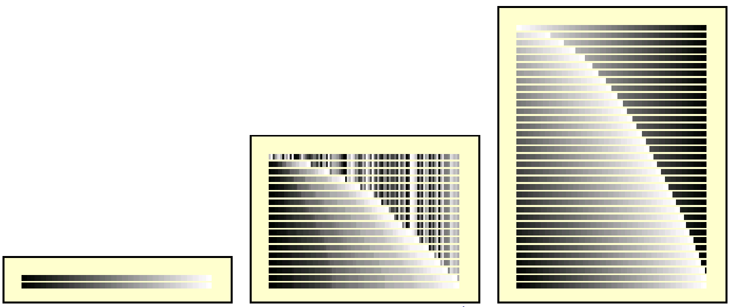
รหัสที่ 13-5 การเรียงลำดับแบบแทรก (insertion sort)

รหัสที่ 13-5 แสดงเมท็อดเรียงลำดับข้อมูลแบบแทรก วงวน `for` ในบรรทัดที่ 32 มีตัวแปร `k` เก็บเลขช่องของแถวลำดับ (เริ่มตั้งแต่ช่อง 1 เพิ่มไปทีละตัวถึงช่องขวาสุด) โดยมีข้อกำหนดว่าเมื่อสนใจข้อมูล `d[k]` ข้อมูล `d[0]` ถึง `d[k-1]` ต้องเรียงลำดับแล้ว ดังนั้นวงวน `for` นี้จึงมีหน้าที่หาที่แทรกให้กับ `d[k]` ระหว่างช่อง `0` ถึงช่อง `k` เริ่มด้วยการนำ `d[k]` ไปเก็บที่ตัวแปรอื่นชื่อ `t` ก่อน (บรรทัดที่ 33) จากนั้นเข้าวงวน `while` มีตัวแปร `j` เริ่มที่ `k-1` ลดลงทีละหนึ่ง เพื่อดันข้อมูลที่มีค่ามากกว่า `t` ไปทางขวาหนึ่งตำแหน่ง เมื่อหลุดจากวงวน `while` จะได้ `j+1` ชี้เลขช่องที่เมื่อนำ `t` ใส่ลง

ไปแล้วจะทำให้ `d[0]` ถึง `d[k]` เรียงจากน้อยไปมาก เช่น ข้อมูลเป็น 12,13,25,49,14, ... โดยที่ `k=4` นั่นคือ 14 เป็นข้อมูลที่สนใจ ข้อมูลทางซ้ายส่วนที่ขีดเส้นใต้ `d[0]` ถึง `d[3]` เรียงลำดับแล้ว เมื่อทำวงวน `while` เสร็จ จะได้ 12,13,25,25,49,... โดยที่ `j=1` นำ 14 (ซึ่งในตัวแปร `t`) ไปใส่ใน `d[j+1]` จะได้ 12,13,14,25,49,... เป็นผลลัพธ์

วงวน `for` ในบรรทัดที่ 32 ของรหัสที่ 13-5 หมุนเป็นจำนวน $n - 1$ รอบ แต่ละรอบจะหมุนในวงวน `while` อีกกี่รอบนั้นขึ้นกับลักษณะของข้อมูลขาเข้า ถ้าข้อมูลเริ่มต้นเรียงลำดับแล้ว `d[k]` ต้องไม่น้อยกว่า `d[k-1]` ทำให้เงื่อนไขของ `while` (บรรทัดที่ 35) เป็นเท็จทันที ไม่ต้องเข้าในวงวนเลย เกิดการเปรียบเทียบใน `for` รอบละครั้งเท่านั้น จึงเปรียบเทียบข้อมูลรวม $n - 1$ ครั้ง แต่ถ้าข้อมูลเริ่มต้นเรียงกลับลำดับจากมากไปน้อย `d[k]` ย่อมมีค่าน้อยกว่าทุกตัวที่อยู่ทางซ้าย จะได้ว่า ในวงวน `for` รอบที่ `k` ต้องนำ `d[k]` ไปแทรกไว้ทางซ้ายสุด เกิดการเปรียบเทียบที่วงวน `while` เป็นจำนวน `k` ครั้ง ดังนั้นรวมทั้งสิ้นต้องเปรียบเทียบ $1+2+...+(n-1) = n(n-1)/2$ ครั้ง

ถึงแม้ว่า กรณีเร็วสุดการเรียงลำดับแบบแทรกใช้เวลา $\Theta(n)$ และกรณีช้าสุดใช้เวลา $\Theta(n^2)$ แต่ถ้าคิดในกรณีเฉลี่ยจะยังคงใช้เวลา $\Theta(n^2)$ ซึ่งสามารถแสดงให้เห็นจริงดังนี้ พิจารณารอบที่ `k` ของวงวน `for` เมื่อหลุดจากวงวน `while` (บรรทัดที่ 39) ตำแหน่งที่เป็นไปได้สำหรับ `d[k]` คือ 0, 1, 2, 3, ..., หรือ k ซึ่งเกิดการเปรียบเทียบ $k, k, k-1, k-2, ..., 2$, หรือ 1 ครั้งตามลำดับ สมมติให้โอกาสที่ `d[k]` จะไปอยู่ที่ตำแหน่งต่าง ๆ เท่ากันคือ $1/(k+1)$ จะได้ว่า จำนวนการเปรียบเทียบโดยเฉลี่ยในรอบที่ k คือ $(k + k + (k-1) + (k-2) + ... + 2 + 1) / (k + 1) = (k + k(k+1)/2) / (k+1) \approx k/2 + O(1)$ ดังนั้นจำนวนการเปรียบเทียบรวมในกรณีเฉลี่ยเป็น $\sum_{k=1}^{n-1}(k/2 + O(1)) = n(n-1)/4 + O(n) = \Theta(n^2)$



รูปที่ 13-9 แถบสีแสดงการเรียงลำดับแบบแทรกกับข้อมูลที่เรียงแล้ว แบบสุ่ม และแบบกลับลำดับ

รูปที่ 13-9 แสดงการเปลี่ยนแปลงของข้อมูลในแถวลำดับด้วยแถบสี ระหว่างการเรียงลำดับแบบแทรกซึ่งรับข้อมูลเริ่มต้นในลักษณะต่างกัน รูปซ้ายทำงานได้เร็วสุด ไม่มีการดันข้อมูลเลย (เพราะข้อมูลเรียงอยู่แล้ว) ในขณะที่รูปขวาทำงานช้าสุดซึ่งคือกรณีที่ได้รับข้อมูลที่กลับลำดับ ต้องดันข้อมูลทุกตัวในทุกรอบ ส่วนรูปกลางนั้นข้อมูลเริ่มต้นมีลักษณะสุ่ม จุดเด่นของการเรียงลำดับแบบแทรกคือ

เวลาการทำงานแปรตามจำนวนกลับลำดับของข้อมูล นั่นคือ “ยิ่งเรียงยิ่งเร็ว” ประกอบกับการใช้วิธีดันข้อมูลแทนการสลับ จึงทำให้การเรียงลำดับแบบแทรกมีประสิทธิภาพที่ดีในทางปฏิบัติเมื่อเทียบกับแบบเลือกและแบบฟองที่ได้นำเสนอมา

การเรียงลำดับแบบเชลล์

การเรียงลำดับแบบเชลล์ (Shell sort : เชลล์คือชื่อของผู้ออกแบบวิธีนี้) นำการเรียงลำดับแบบแทรกมาปรับปรุงให้ทำงานได้เร็วขึ้น จุดด้อยของการเรียงลำดับแบบแทรกอยู่ตรงที่การดันข้อมูลนั้นเลื่อนข้อมูลไปทีละช่อง การเลื่อนหนึ่งครั้งเทียบได้กับการลดจำนวนกลับลำดับลงหนึ่ง ซึ่งช้า ทำไมไม่ขยับข้อมูลไปทีละไกล ๆ ย่อมทำให้จำนวนกลับลำดับลดลงฮวบฮาบอย่างรวดเร็วด้วยภาระที่น้อยกว่า

ดูตัวอย่างในรูปที่ 13-10 แถวบนสุด (#1) มีจำนวนกลับลำดับเป็น 69 หากเราใช้การเรียงลำดับแบบแทรกในหัวข้อที่แล้วจะต้องเปรียบเทียบข้อมูล 81 ครั้ง คราวนี้ขอทำแบบใหม่ เราแบ่งแถวบนออกเป็น 3 ชุด โดยแบ่งแบบห่างกันทีละ 3 ตัว ได้สามแถว (#2, #3, #4) ที่แสดงถัดลงมา (ต้องขอบอกตรงนี้ก่อนว่า เราไม่ได้สร้างแถวลำดับอีกสามชุดหรอก ยังคงใช้แถวเดิมนั่นแหละ แต่พิจารณาข้อมูลเฉพาะช่องที่เราสนใจเท่านั้น) แล้วนำข้อมูลสามชุดใหม่นี้ไปเรียงลำดับแบบแทรกทีละชุด โดยขณะเรียงลำดับให้คิดเสียว่า ข้อมูลที่ห่างกัน 3 ตัวคือตัวที่ติดกัน ทำให้การดันข้อมูลหนึ่งครั้งย้ายข้อมูลจากเดิมไป 3 ตำแหน่ง จำนวนการกลับลำดับจึงลดลงได้มากกว่าแบบย้ายทีละช่อง ได้ผลลัพธ์ดังแสดงในสามแถวต่อมา (#5, #6, #7) ซึ่งต้องเปรียบเทียบข้อมูลอีก 12, 9, และ 7 ครั้งตามลำดับ เมื่อนำผลลัพธ์ที่ได้มารวมกันจะได้ดังแถว #8 ซึ่งก็ยังเรียงไม่เสร็จ แต่ที่สำคัญคือจำนวนกลับลำดับของข้อมูลจากเดิมในแถว #1 มี 69 เหลือ 12 ในแถว #8 ลดลงไป 69 – 12 = 57 โดยใช้การเรียงลำดับย่อยสามครั้งที่เปรียบเทียบข้อมูลเพียง 12+9+7 = 28 ครั้ง (ด้วยวิธีการเรียงลำดับก่อนหน้านี้ จำนวนกลับลำดับลดลง 1 ต้องใช้การเปรียบเทียบอย่างน้อย 1 ครั้ง) เนื่องจากแถว #8 ยังเรียงไม่เสร็จ แต่เรียงมากกว่าในแถว #1 ก็ส่งไปเรียงลำดับแบบแทรกอีกครั้ง ได้ผลลัพธ์ที่เรียงเรียบร้อยในแถว #9 ต้องเปรียบเทียบข้อมูลเพิ่มอีก 27 ครั้ง ผลจำนวนการเปรียบเทียบทั้งสิ้นเป็น 12+9+7+27 = 55 ครั้ง ซึ่งน้อยกว่าแบบส่งแถว #1 ไปเรียงลำดับแบบแทรกครั้งเดียวซึ่งต้องเปรียบเทียบถึง 81 ครั้ง

รหัสที่ 13-6 แสดงส่วนของโปรแกรมที่เรียงลำดับข้อมูล `d[m]`, `d[m+h]`, `d[m+2h]`, ... ของ `d` โดยใช้การเรียงลำดับแบบแทรก ส่วนของโปรแกรมนี้ปรับจากการเรียงลำดับแบบแทรกในรหัสที่ 13-5 โดยเปลี่ยนจากที่เคยพิจารณาตัวที่ติดกันว่า ห่างกัน 1 ช่อง ก็ให้ถือเสียว่า ตัวที่ติดกันนั้น ห่างกัน `h` ช่อง (อะไรที่เคยบวก `1` ก็เปลี่ยนเป็นบวก `h` และที่เคยลบ `1` ก็เปลี่ยนเป็นลบ `h`)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	
#1	32	23	11	55	18	3	14	28	34	17	92	43	61	0	9	1	52	inv=69
#2	32			55			14			17			61			1		
#3		23			18			28			92			0			52	
#4			11			3			34			43			9			
#5	1			14			17			32			55			61		cmp+=12
#6		0			18			23			28			52			92	cmp+=9
#7			3			9			11			34			43			cmp+=7
#8	1	0	3	14	18	9	17	23	11	32	28	34	55	52	43	61	92	inv=12
#9	0	1	3	9	11	14	17	18	23	28	32	34	43	52	55	61	92	cmp+=27

รูปที่ 13-10 ตัวอย่างการเรียงลำดับแบบเชลล์

```
        for (int i = m + h; i < d.length; i += h) {
          Object t = d[i];
          int j = i - h;
          while (j >= 0 && lessThan(t, d[j])) {
            d[j + h] = d[j];
            j -= h;
          }
          d[j + h] = t;
        }
```

ถ้า h = 1 ก็เหมือน insertion sort

รหัสที่ 13-6 การเรียงลำดับที่ห่างกัน h ตัว

ขอเรียกการเรียงลำดับในรหัสที่ 13-6 ว่า $s(m,h)$ แทนการเรียงลำดับข้อมูลเริ่มช่องที่ m ข้อมูลแต่ละตัวห่างกัน h ช่อง ซึ่งคือช่องที่ m, $m+h$, $m+2h$, ... และเรียกการทำ $s(0,h)$, $s(1,h)$, $s(2,h)$, ..., $s(h-1, h)$ ว่า $s(h)$ ตัวอย่างเช่น รูปที่ 13-10 เราทำ $s(3)$ ตามด้วย $s(1)$ นั่นคือเริ่มด้วยการแบ่งข้อมูลเป็น 3 ชุด ($h = 3$) ทำ $s(0,3)$, $s(1,3)$, และ $s(2,3)$ ได้ผลดังแถว #5, #6, และ #7 ในรูป จากนั้นจึงทำ $s(1)$ ซึ่งคือการเรียงลำดับข้อมูลทุกตัวในแถว #9 การเรียงลำดับแบบเชลล์ คือ $s(h_k)$, $s(h_{k-1})$, ..., $s(h_2)$, $s(h_1)$ โดยที่ $h_k > h_{k-1}, > ... > h_2 > h_1$ และ $h_1 = 1$ การทำปิดท้ายด้วย $s(1)$ ทำให้มั่นใจว่า สุดท้ายแล้วเราจะได้ข้อมูลที่เรียงลำดับแน่ ๆ ไม่ว่าลำดับของ h จะเป็นเช่นใด รูปที่ 13-11 แสดงตัวอย่างการเรียงลำดับแบบเชลล์ที่มีลำดับ h เป็น 5, 3, และ 1

สิ่งที่น่าสนใจเกี่ยวกับการเรียงลำดับแบบเชลล์คือลำดับของ h ควรมีค่าเช่นใด ควรเป็น $\langle 3, 1\rangle$ หรือ $\langle 5, 3, 1\rangle$ หรือ $\langle 8, 4, 2, 1\rangle$ หรือ ลำดับอื่น ในปัจจุบัน ยังไม่มีใครรู้ว่า ลำดับ h ที่ดีสุดควรเป็นเช่นใด ตารางที่ 13-1 แสดงตัวอย่างของลำดับ h ซึ่งสามารถวิเคราะห์เวลาการเรียงลำดับได้ (วิธีวิเคราะห์นั้นซับซ้อน ขอไม่กล่าวถึง) ให้สังเกตว่า ลำดับ h ของคุณเชลล์ (ผู้คิดค้นการเรียงลำดับแบบนี้) กลับเป็นลำดับที่ไม่ค่อยดี ใช้เวลาเป็น $O(n^2)$ สมมติให้ $n = 16$ จะได้ลำดับ h ของเชลล์เป็น $\langle 8, 4, 2, 1\rangle$

นั่นแสดงว่า ข้อมูลในช่องเลขคู่ไม่เคยได้มาเปรียบเทียบกับข้อมูลในช่องเลขคี่เลยยกเว้นรอบสุดท้ายเมื่อ $h = 1$ ถ้าข้อมูลในตำแหน่งคู่เรียงลำดับ และช่องคี่ก็เรียงลำดับ การเรียงลำดับ $s(8)$, $s(4)$, $s(2)$ จะเสียเวลาไปโดยเปล่าประโยชน์เพราะข้อมูลไม่เปลี่ยนตำแหน่งเลย ขอให้ผู้อ่านทดลองใช้การเรียงลำดับแบบเชลล์ด้วยลำดับ h เป็น $\langle 8, 4, 2, 1\rangle$ กับข้อมูล 1, 9, 2, 10, 3, 11, 4, 12, 5, 13, 6, 15, 8, 16

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	32	23	11	55	18	3	14	28	34	17	92	43	61	0	9	1	52
	32					3					92					1	
		23					14					43					52
			11					28					61				
				55					34					0			
					18					17					9		
$s(0,5)$	1					3					32					92	
$s(1,5)$		14					23					43					52
$s(2,5)$			11					28					61				
$s(3,5)$				0					34					55			
$s(4,5)$					9					17					18		
$s(5)$	1	14	11	0	9	3	23	28	34	17	32	43	61	55	18	92	52
	1			0			23			17			61			92	
		14			9			28			32			55			52
			11			3			34			43			18		
$s(0,3)$	0			1			17			23			61			92	
$s(1,3)$		9			14			28			32			52			55
$s(2,3)$			3			11			18			34			43		
$s(3)$	0	9	3	1	14	11	17	28	18	23	32	34	61	52	43	92	55
$s(1)$	0	1	3	9	11	14	17	18	23	28	32	34	43	52	55	61	92

รูปที่ 13-11 ตัวอย่างการเรียงลำดับแบบเชลล์ด้วยลำดับของ h เป็น 5, 3, 1

ตารางที่ 13-1 ตัวอย่างลำดับ h ของการเรียงลำดับแบบเชลล์

	ลำดับ h	เวลาในการเรียงลำดับ
Hibbard	$\langle 2^m - 1, \ldots, 15, 7, 3, 1 \rangle$	$O(n^{3/2})$
Knuth	$\langle \lfloor (3^m - 1)/2 \rfloor, \ldots, 40, 13, 4, 1 \rangle$	$O(n^{3/2})$
Sedgewick	$\langle (4^{m+1} + 3\cdot 2^m + 1), \ldots, 77, 23, 8, 1 \rangle$	$O(n^{4/3})$
Shell	$\langle n/2, n/2^2, n/2^3, \ldots, 1 \rangle$	$O(n^2)$

รหัสที่ 13-7 แสดงการเรียงลำดับแบบเชลล์โดยใช้ลำดับ *h* ของ Hibbard ให้สังเกตว่าบรรทัดที่ 47 ถึง 55 นั้นเหมือนกับรหัสที่ 13-6 ซึ่งคือ $s(m,h)$ ซึ่งแทนการเรียงลำดับข้อมูลเริ่มช่องที่ *m* ห่างกัน *h* ช่อง นำมาครอบด้วยวงวน `for` (บรรทัดที่ 46) ที่แปรค่า $m = 0, 1, 2, ..., h - 1$ เพื่อทำ $s(h)$ จากนั้นครอบด้วยวงวน `for` (บรรทัดที่ 45) ที่แปรค่า *h* โดยลดค่าของ *h* ทีละครึ่งจากรอบที่แล้ว (วงวนในบรรทัดที่ 44 หาค่าเริ่มต้นของ *h* โดยเพิ่มค่า *h* ในลำดับ 1,3,7,15,... ไปจนเกิน *n*/4 เป็นครั้งแรก)

```
42   public static void shellSort(Object[] d) {
43     int h;
44     for (h = 1; h <= d.length/4; h = 2*h + 1);
45     for (; h > 0; h /= 2) {
46       for (int m = 0; m < h; m++) {
47         for (int i = m + h; i < d.length; i += h) {
48           Object t = d[i];
49           int j = i - h;
50           while (j >= 0 && lessThan(t, d[j])) {
51             d[j + h] = d[j];
52             j -= h;
53           }
54           d[j + h] = t;
55         }
56       }
57     }
58   }
```

หา h เริ่มต้นที่มากสุดที่น้อยกว่าจำนวนข้อมูล

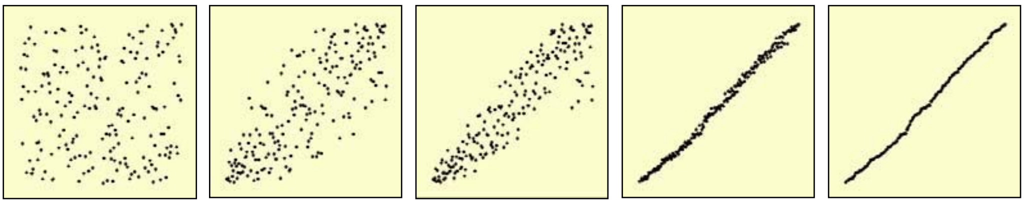
ทำ s(h)

ทำ s(m,h) เหมือนรหัสที่ 13-6

h ของ Hibbard ทำ...,s(31),s(15),s(7),s(3),s(1)

รหัสที่ 13-7 การเรียงลำดับแบบเชลล์ (Shell sort) ที่ใช้ลำดับ *h* ของ Hibbard ($2^m - 1$)

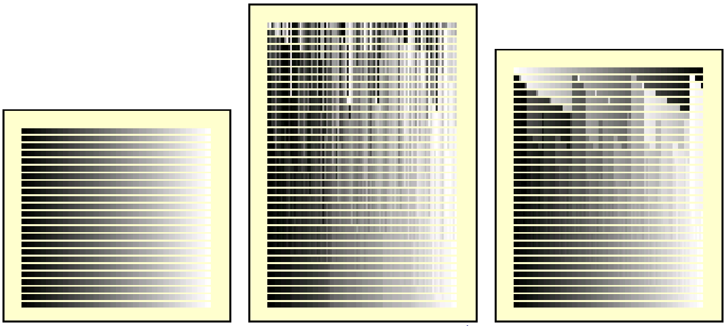
รูปที่ 13-12 แสดงภาพการเปลี่ยนแปลงของข้อมูลระหว่างการเรียงลำดับแบบเชลล์ ให้สังเกตว่าข้อมูลเริ่มย้ายตำแหน่งเข้าสู่แนวทแยงมุมใกล้ขึ้น ๆ อย่างรวดเร็ว ซึ่งสะท้อนให้เห็นว่า จำนวนกลับลำดับนั้นลดลงอย่างรวดเร็ว ด้วยจุดเด่นของการย้ายได้ไกลหรือใกล้ตามค่า *h* ที่ใช้ขณะนั้น



รูปที่ 13-12 ภาพแสดงการเปลี่ยนแปลงของข้อมูลระหว่างการเรียงลำดับแบบเชลล์

รูปที่ 13-13 แสดงแถบสีการเปลี่ยนแปลงของข้อมูลระหว่างการเรียงลำดับแบบเชลล์ รูปซ้ายรับข้อมูลที่เรียงลำดับ แต่ก็ต้องเสียเวลาเรียงลำดับหลายรอบสำหรับแต่ละค่าของ *h* และให้สังเกตว่า ข้อมูลขาเข้าที่เรียงกลับลำดับ (รูปขวา) ใช้เวลาน้อยกว่ากรณีข้อมูลขาเข้าแบบสุ่ม (รูปกลาง) ตารางที่ 13-2 แสดงเวลาการทำงานของการเรียงลำดับแบบเชลล์โดยใช้ลำดับ *h* ที่ต่างกัน จะเห็นได้ว่าลำดับ *h* มีผลต่อเวลาการทำงานอย่างเห็นได้ชัด การเรียงลำดับแบบเชลล์ใช้ได้ดีกับข้อมูลปริมาณเป็นหมื่นเป็นแสน

(ซึ่งถ้าใช้กับการเรียงลำดับแบบเลือก แบบฟอง หรือแบบแทรก จะใช้เวลานานกว่ามาก) เป็นการเรียงลำดับที่เขียนง่าย แต่การวิเคราะห์ประสิทธิภาพการทำงานไม่ง่ายเหมือนตัวโปรแกรม



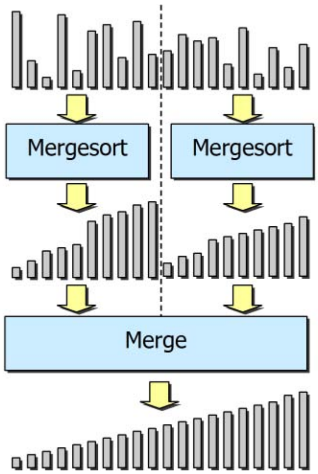
รูปที่ 13-13 แถบสีแสดงการเรียงลำดับแบบเชลล์กับข้อมูลที่เรียงแล้ว แบบสุ่ม และแบบกลับลำดับ

ตารางที่ 13-2 เวลาการเรียงลำดับแบบเชลล์กับข้อมูล 1 ล้านตัว โดยใช้ลำดับ h ต่างกัน

	ลำดับ h	เวลาการทำงาน (มิลลิวินาที)	
		ข้อมูลแบบสุ่ม	ข้อมูลเรียงกลับลำดับ
Hibbard	$\langle 2^m - 1, \ldots, 15, 7, 3, 1 \rangle$	1648	1019
Knuth	$\langle \lfloor (3^m - 1)/2 \rfloor, \ldots, 40, 13, 4, 1 \rangle$	1320	547
Sedgewick	$\langle (4^{m+1} + 3 \cdot 2^m + 1), \ldots, 77, 23, 8, 1 \rangle$	844	528
Shell	$\langle n/2, n/2^2, n/2^3, \ldots, 1 \rangle$	7617	2265

การเรียงลำดับแบบผสาน

การเรียงลำดับแบบผสาน (merge sort) อาศัยวิธีแก้ปัญหาที่เรียกว่า *การแบ่งแยกและเอาชนะ* (divide and conquer) โดยแบ่งแถวลำดับออกเป็นสองชุดย่อย ชุดทางซ้ายและชุดทางขวาขนาดพอ ๆ กัน นำข้อมูลทั้งสองชุดไปเรียงลำดับ แล้วนำผลที่ได้มาผสานกันได้ผลที่เรียงลำดับทั้งหมด ดังแสดงในรูปที่ 13-14 (ข้อมูลแต่ละตัวถูกแทนด้วยแท่งสี่เหลี่ยม แท่งสูงแทนข้อมูลที่มีค่ามาก) คำถามที่ตามมาก็คือ แบ่งอย่างไร เรียงลำดับชุดย่อยอย่างไร และผสานอย่างไร



รูปที่ 13-14 Merge sort

เราแบ่งแถวลำดับตรงกลาง จากนั้นเรียงลำดับชุดซ้ายและชุดขวาแบบเวียนเกิด (recursive) ด้วยการเรียงลำดับแบบผสาน บรรทัดที่ 63 ของรหัสที่ 13-8 แสดงเมท็อดเรียงลำดับข้อมูลใน

แถวลำดับ d เริ่มจากเลขช่อง left ถึง right (เมท็อดนี้ยังรับแถวลำดับเสริมอีกแถวชื่อ t ซึ่งใช้เพื่อการผสาน) เริ่มด้วยการทดสอบว่า left < right หรือไม่ ถ้าใช่ แสดงว่ามีข้อมูลในช่วงที่ให้มาเกิน 1 ตัว ก็คำนวณตำแหน่งตรงกลางช่วง (บรรทัดที่ 65) แล้วสั่งให้เรียงลำดับครึ่งซ้าย, เรียงลำดับครึ่งขวา (บรรทัดที่ 66 และ 67), ผสานข้อมูล (ด้วยเมท็อด merge) ในชุดซ้ายและขวาเข้าด้วยกันได้ผลเก็บใน t ปิดท้ายด้วยการย้ายข้อมูลจาก t กลับมา d ด้วยวงวนในบรรทัดที่ 69 เราเขียนเมท็อด mergeSort ในบรรทัดที่ 59 ให้เรียก mergeSortR เพื่อเรียงลำดับทั้งแถวลำดับ d พร้อมส่งแถวลำดับเสริมอีกแถวที่มีขนาดเท่ากับ d ให้เพื่อใช้เป็นที่เก็บเสริมในการผสาน

```
59   public static void mergeSort(Object[] d) {
60     mergeSortR(d, 0, d.length-1, new Object[d.length]);
61   }
62   private static
63   void mergeSortR(Object[] d, int left, int right, Object[] t) {
64     if (left < right) {
65       int m = (left + right) / 2;
66       mergeSortR(d, left, m, t);
67       mergeSortR(d, m + 1, right, t);
68       merge(d, left, m, right, t);
69       for (int i=left; i<=right; i++) d[i] = t[i];
70     }
71   }
```

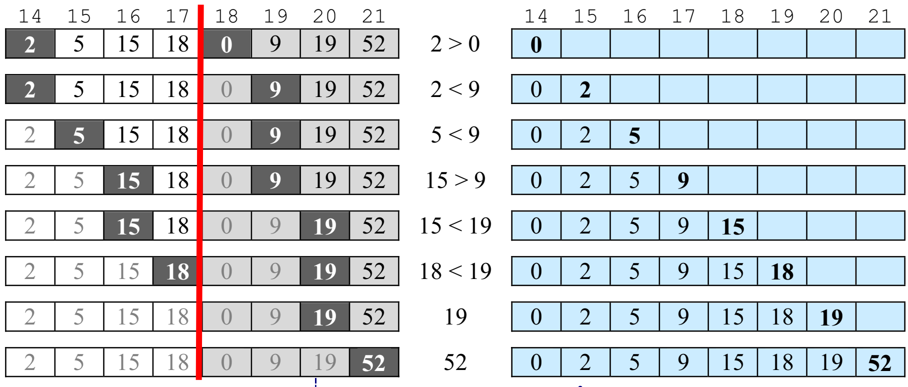
แถวลำดับเสริมที่ใช้ในการผสาน

เรียงลำดับครึ่งซ้ายและครึ่งขวา

ผสานแล้วได้ผลเก็บใน t

ย้ายผลจาก t กลับมา d

รหัสที่ 13-8 การเรียงลำดับแบบผสาน (merge sort)



รูปที่ 13-15 ตัวอย่างการผสานข้อมูล

สำหรับการผสานข้อมูลนั้น ต้องอย่าลืมว่า เราผสานข้อมูลสองชุด ที่แต่ละชุดเรียงลำดับเรียบร้อยแล้วให้เป็นชุดรวมที่เรียงลำดับ รูปที่ 13-15 แสดงตัวอย่างการผสานข้อมูลสองชุดที่เก็บในแถวลำดับทางซ้ายของรูป ข้อมูลชุดแรกอยู่ในช่องที่ 14 ถึง 17 อีกชุดอยู่ในช่องที่ 18 ถึง 21 ส่วนผลที่ได้เก็บในแถวลำดับทางขวา การผสานนำข้อมูลเริ่มจากตัวซ้ายของแต่ละชุดมาเปรียบเทียบกัน นำตัวน้อยกว่าไปใส่ในผลลัพธ์ แล้วเลื่อนไปสนใจข้อมูลตัวถัดไป (ตัวที่มากกว่ายังคงไว้) กระทำการ

เปรียบเทียบและนำตัวน้อยไปต่อท้ายผลลัพธ์เช่นนี้จนชุดใดชุดหนึ่งหมด แล้วนำข้อมูลที่เหลือในอีกชุดไปต่อท้ายผลลัพธ์ให้หมด ดังตัวอย่างในรูป หลังการย้าย 18 แล้ว ชุดแรกหมด ก็นำ 19 และ 52 ไปต่อท้ายผลลัพธ์

รหัสที่ 13-9 แสดงเมท็อด `merge` เพื่อผสานข้อมูลสองชุดใน `d` ชุดแรกคือ `d[left]` ถึง `d[mid]` ชุดหลังคือ `d[mid+1]` ถึง `d[right]` ผลลัพธ์เก็บในแถวลำดับ `t` จาก `t[left]` ถึง `t[right]` บรรทัดที่ 74 เตรียมตัวแปร `i` เริ่มที่ `left` ไว้เก็บเลขช่องของข้อมูลที่กำลังสนใจในชุดแรก และตัวแปร `j` เริ่มที่ `mid+1` เก็บเลขช่องของข้อมูลที่กำลังสนใจในชุดหลัง วงวนการผสาน (บรรทัดที่ 75) มีตัวแปร `k` เก็บเลขช่องของ `t` มีบรรทัดที่ 78 ทำหน้าที่หลักในการผสาน คือนำตัวน้อยระหว่าง `d[i]` กับ `d[j]` ไปเก็บใน `t[k]` ส่วนบรรทัดที่ 76 มีไว้ทำสำเนาข้อมูลที่เหลือของชุดหลังไปยัง `t` เมื่อข้อมูลชุดแรกหมดก่อน ในขณะที่บรรทัดที่ 77 ทำสำเนาข้อมูลที่เหลือของชุดแรกไปยัง `t` เมื่อข้อมูลชุดหลังหมดก่อน (เงื่อนไขของบรรทัดที่ 76 และ 77 จึงไม่มีทางเป็นจริงพร้อมกันแน่ ๆ)

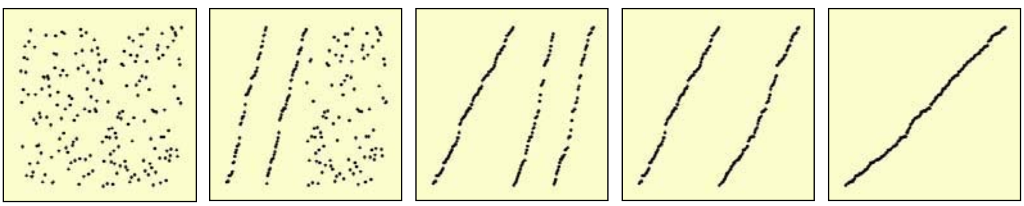
```
72   private static void merge(Object[] d, int left, int mid,
73                                int right, Object[] t) {
74     int i = left, j = mid+1;
75     for (int k = left; k <= right; k++) {
76       if (i > mid)    {t[k] = d[j++]; continue;}
77       if (j > right) {t[k] = d[i++]; continue;}
78       t[k] = lessThan(d[i], d[j]) ? d[i++] : d[j++];
79     }
80   }
```

ย้ายครึ่งขวา เมื่อครึ่งซ้ายหมดก่อน

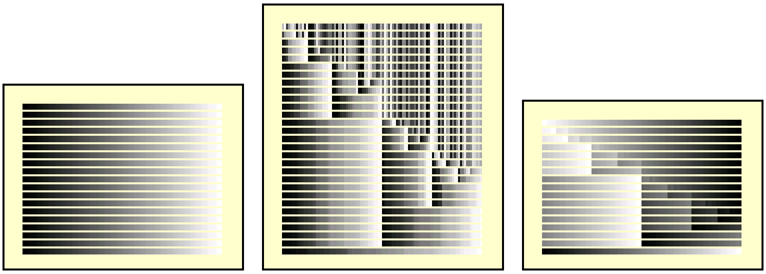
ย้ายครึ่งซ้าย เมื่อครึ่งขวาหมดก่อน

รหัสที่ 13-9 เมท็อด `merge` เพื่อการผสานข้อมูลจาก `left` ถึง `right` ของ `d` เก็บผลลัพธ์ไว้ที่ `t`

รูปที่ 13-16 และรูปที่ 13-17 แสดงภาพการเปลี่ยนแปลงของข้อมูลระหว่างการเรียงลำดับแบบผสาน ให้สังเกตการทำงานแบบเวียนเกิดที่เราเขียนให้เรียงลำดับครึ่งซ้ายก่อน จึงเห็นการเปลี่ยนแปลงที่เรียงลำดับครึ่งซ้ายทีละครึ่ง ๆ ผสานชุดเล็กกลายเป็นชุดใหญ่ขึ้น ๆ และทำกับครึ่งขวาของแถวลำดับในทำนองเดียวกัน และให้สังเกตว่า กรณีข้อมูลเรียงลำดับแล้ว กับกรณีเรียงกลับลำดับ ใช้เวลาการทำงานน้อยกว่ากรณีข้อมูลสุ่ม



รูปที่ 13-16 ภาพแสดงการเปลี่ยนแปลงของข้อมูลระหว่างการเรียงลำดับแบบผสาน



รูปที่ 13-17 แถบสีแสดงการเรียงลำดับแบบผสานกับข้อมูลที่เรียงแล้ว แบบสุ่ม และแบบกลับลำดับ

แล้วการเรียงลำดับแบบผสานดีกว่าแบบอื่น ๆ ที่ได้นำเสนอมาหรือไม่ ? กำหนดให้ $c(n)$ คือจำนวนการเปรียบเทียบข้อมูลในการเรียงลำดับข้อมูลจำนวน n ตัวแบบผสาน ซึ่งประกอบด้วยจำนวนการเปรียบเทียบสำหรับการเรียงลำดับครึ่งซ้ายและครึ่งขวาแบบผสานซึ่งเท่ากับ $c(\lceil n/2 \rceil)$ กับ $c(\lfloor n/2 \rfloor)$ ตามลำดับ ตามด้วยการผสานข้อมูลซึ่งเกิดการเปรียบเทียบ (บรรทัดที่ 78 ของรหัสที่ 13-9) อย่างน้อย $n/2$ ครั้ง อย่างมาก $n - 1$ ครั้ง (ลองคิดดูโดยใช้ตัวอย่างในรูปที่ 13-15 ประกอบ) เพื่อให้วิเคราะห์ง่ายขอกำหนดให้ n หารสองลงตัวตลอด (นั่นคือ $\lceil n/2 \rceil = \lfloor n/2 \rfloor = n/2$) จะได้ขอบเขตบนของ $c(n)$ ดังนี้

$$
\begin{aligned}
c(n) &\le 2c(n/2) + (n-1) \\
&\le 2(2c(n/4) + (n/2-1)) + (n-1) \\
&= 2^2 c(n/2^2) + 2n - (2+1) \\
&\le 2^3 c(n/2^3) + 3n - (4+2+1) \\
&\ \ \ldots \\
&\le 2^{\log_2 n} c(n/2^{\log_2 n}) + n\log_2 n - (2^{\log_2 n-1} + \ldots + 2^1 + 2^0) \\
&= n\log_2 n - n + 1
\end{aligned}
$$

สำหรับขอบเขตล่างสามารถวิเคราะห์ได้ว่า $c(n) \ge 2c(n/2) + n/2 = (n/2)\log_2 n$ สรุปได้ว่าการเรียงลำดับแบบผสานใช้เวลา $\Theta(n \log n)$ เมื่อเทียบกับการเรียงลำดับแบบเลือก แบบฟอง และแบบแทรก ซึ่งใช้เวลา $O(n^2)$ กับแบบเชลล์ที่ใช้เวลา $O(n^{1.xx})$ พบว่า การเรียงลำดับแบบผสานดีกว่ามาก (อย่าลืมว่า $n^{0.xx}$ เป็นฟังก์ชันที่โตเร็วกว่า $\log n$) นั่นหมายความว่า ถ้าต้องเรียงลำดับข้อมูลจำนวนมาก แบบผสานจะใช้เวลาน้อยกว่าแบบอื่น ๆ ที่ได้นำเสนอมา

โปรแกรมที่ได้เขียนมาจะเสียเวลาย้ายข้อมูลจากแถวลำดับ `d` ไป `t` ในขั้นตอนการผสาน และย้ายจาก `t` กลับมา `d` เมื่อผสานเสร็จ เราสามารถลดจำนวนการย้ายข้อมูลตรงนี้ได้ดังรหัสที่ 13-10 ด้วยการสร้างแถวลำดับเสริม `t` ให้มีค่าเหมือนกับ `d` ที่ได้รับตอนเริ่มต้น (ด้วยเมท็อด `clone` ของจาวา) จากนั้นแทนที่จะให้เรียงลำดับครึ่งซ้ายและขวาของ `d` ก็ให้ไปเรียงลำดับครึ่งซ้ายและขวาของ `t` แทน (เพราะมีข้อมูลภายในเหมือนกัน) ได้ผลกลับมา ก็ให้ผสานข้อมูลสองชุดใน `t` แล้วส่งผลลัพธ์ไปเก็บที่ `d` ก็ไม่ต้องเสียเวลาย้ายข้อมูลหลังการผสาน จากทดลองพบว่า ใช้เวลาน้อยลงประมาณ 20%

```
public static void mergeSort(Object[] d) {
  mergeSortR(d, 0, d.length-1, (Object[])d.clone());
}
private static
void mergeSortR(Object[] d, int left, int right, Object[] t) {
  if (left < right) {
    int m = (left + right) / 2;
    mergeSortR(t, left, m, d);
    mergeSortR(t, m + 1, right, d);
    merge(t, left, m, right, d);
  }
}
```

สำเนาแถวลำดับ ใช้ในการผสาน

สังเกตการใช้ t และ d สลับกับที่ปรากฏในรหัสที่ 13-8 เพื่อลดจำนวนการย้ายข้อมูล

รหัสที่ 13-10 การเรียงลำดับแบบผสาน (รุ่นปรับปรุง)

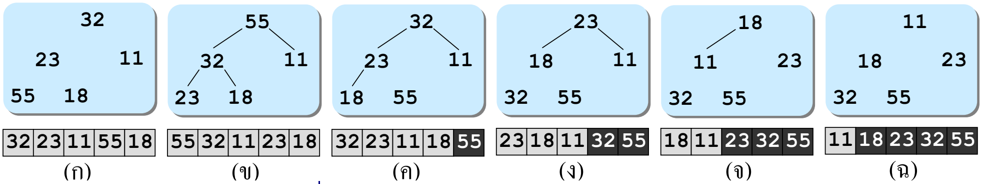
ข้อเสียของการเรียงลำดับแบบผสานคือต้องใช้เนื้อที่เสริมในการผสาน โดยใช้เท่ากับจำนวนข้อมูลเลยทีเดียว แสดงว่า ต้องการเรียงข้อมูลล้านตัว ก็ต้องจองอีกล้านช่อง (จริง ๆ แล้วมีวิธีผสานโดยไม่ต้องใช้เนื้อที่เสริมมากขนาดนั้น แต่เป็นขั้นตอนที่ซับซ้อนยุ่งยาก ไม่ขอนำเสนอ) นอกจากนี้ต้องอย่าลืมว่า ยังต้องใช้เนื้อที่เสริมอีก $\Theta(\log n)$ ในกองซ้อนของระบบสำหรับการเรียกแบบเวียนเกิดด้วย

การเรียงลำดับแบบฮีป

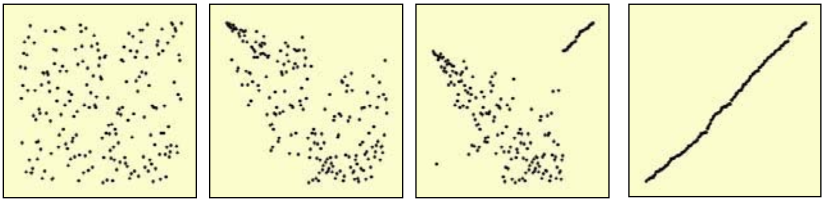
เราได้นำเสนอ*การเรียงลำดับแบบฮีป* (heap sort) กันมาแล้วในบทที่ 8 โดยได้เพิ่มเมท็อด `sort` ไว้ในคลาส `BinaryHeap` จึงขอสรุปสั้น ๆ ในที่นี้ การเรียงลำดับแบบฮีปอาศัยแนวคิดการทำงานเหมือนกับการเรียงลำดับแบบเลือก คือเลือกตัวที่มีค่ามากสุด เพื่อนำไปวางไว้ที่ตำแหน่งขวาสุดของกลุ่ม แต่เนื่องจากการหาตัวมากสุดของการเรียงลำดับแบบเลือกนั้นใช้วิธีค่อย ๆ เปรียบเทียบข้อมูลในรายการ ซึ่งต้องเปรียบเทียบ $n-1$ ครั้งสำหรับรายการที่มีข้อมูล n ตัว เนื่องจากต้องหาตัวมากสุดเช่นนี้ $n-1$ ครั้ง จึงใช้การเปรียบเทียบรวมเป็น $\Theta(n^2)$ ในขณะที่การเรียงลำดับแบบฮีป อาศัยการจัดกลุ่มข้อมูลให้เป็นฮีป (แบบมากสุด) การหาตัวมากสุดใช้เวลาคงตัว (เพราะตัวมากสุดจะอยู่ที่ช่องที่ 0 เสมอ) แต่การนำตัวมากสุดออก ต้องปรับฮีปด้วยการ `fixDown` ข้อมูลที่ราก เกิดการเปรียบเทียบและสลับข้อมูลจากรากลงไปด้านล่าง กรณีช้าสุดต้องทำถึงระดับล่างสุด มีการเปรียบเทียบเป็นจำนวนไม่เกิน $2\log_2 n$ ครั้ง ต้องลบตัวมากสุดเช่นนี้ n ครั้ง เปรียบเทียบรวมไม่เกิน $2n \log_2 n$ ครั้ง

รูปที่ 13-18 แสดงตัวอย่างการเรียงลำดับแบบฮีป เริ่มด้วยการนำแถวลำดับในรูป (ก) ไปสร้างเป็นฮีปในรูป (ข) แล้วเริ่มลบตัวมากสุด (ตัวที่อยู่ในช่องที่ 0) ไปวางในตำแหน่งขวาสุดของกลุ่ม กระทำเช่นนี้ไปจนเหลือข้อมูลในฮีปเพียง 1 ตัว ดังรูป (ค) ถึง (ฉ) ก็จบการทำงาน ส่วนรูปที่ 13-19 แสดงการเปลี่ยนแปลงข้อมูลระหว่างการทำงานกับปริมาณข้อมูลจำนวนมากในอีกมุมมองหนึ่ง รูปที่สองจากซ้ายเป็นสภาพของข้อมูลเมื่อข้อมูลถูกสร้างเป็นฮีปแล้ว หลังจากนั้นจะเห็นเส้นข้อมูลที่ทอดตามแนวทแยง

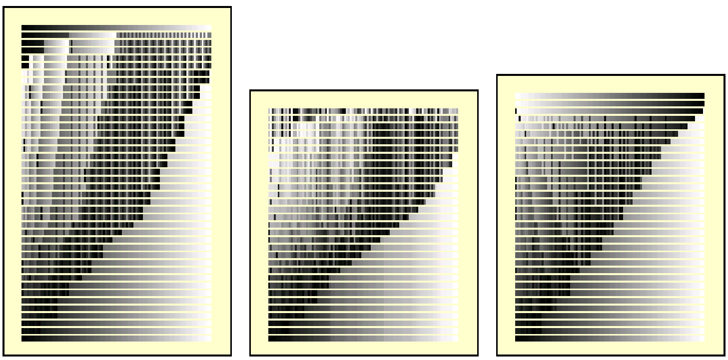
มุมจากมุมขวาบนลงมา ซึ่งแสดงให้เห็นถึงการนำตัวมากสุดในกลุ่มซ้ายไปไว้ที่ตำแหน่งขวาสุดของกลุ่ม ส่วนกลุ่มข้อมูลทางซ้ายก็เป็นฮีปที่มีขนาดเล็กลง ๆ จนได้ข้อมูลที่เรียงลำดับทั้งหมด



รูปที่ 13-18 ตัวอย่างการเรียงลำดับแบบฮีป


รูปที่ 13-19 ภาพแสดงการเปลี่ยนแปลงของข้อมูลระหว่างการเรียงลำดับแบบฮีป

รูปที่ 13-20 แสดงแถบสีแทนการเปลี่ยนแปลงข้อมูลในแถวลำดับ ที่มีค่าเริ่มต้นต่างกันระหว่างการเรียงลำดับแบบฮีป ให้สังเกตรูปทางขวาเป็นข้อมูลที่เริ่มต้นแบบกลับลำดับ เรียงจากมากไปน้อย ทำให้ขั้นตอนการสร้างฮีปไม่เกิดการเปลี่ยนแปลงอะไรเลยเพราะเป็นฮีปอยู่แล้ว ในขณะที่ถ้าข้อมูลเริ่มต้นเรียงลำดับจากน้อยไปมากในรูปซ้าย จะต้องเปลี่ยนเป็นฮีปเสียก่อน ทำให้ข้อมูลเปลี่ยนแปลงไปมากแล้วจึงค่อยปรับกลับสู่สภาพเรียงจากน้อยไปมากดังเดิมเมื่อเรียงเสร็จ (ซึ่งก็ดูแล้วแปลกดี)


รูปที่ 13-20 แถบสีแสดงการเรียงลำดับแบบฮีปกับข้อมูลที่เรียงแล้ว แบบสุ่ม และแบบกลับลำดับ

รหัสที่ 13-11 แสดงการเรียงลำดับแบบฮีป บรรทัดที่ 83 เปลี่ยนข้อมูลในแถวลำดับให้เป็นฮีปโดยการ `fixDown` จากปมภายในที่ระดับล่าง ๆ ไล่ขึ้นมาที่ราก จากนั้นเข้าสู่วงวนสลับตัวมากสุดในฮีปกับตำแหน่งทางขวาไล่จากขวาสุดถอยกลับมาทางซ้าย แล้วปรับฮีปด้วย `fixDown` (บรรทัดที่ 86) (`fixDown` นี้คล้ายกับที่ได้นำเสนอมาใน `BinaryHeap` ของบทที่ 8)

```
public static void heapSort(Object[] data) {
  int size = data.length;
  for(int k=size/2-1; k>=0; k--) fixDown(data, size, k);
  for(int k=size-1; k>0; k--) {
    swap(data, 0, k);
    fixDown(data, --size, 0);
  }
}
private static void fixDown(Object[] data, int size, int k) {
  int c;
  while ((c = 2 * k + 1) < size) {
    if (c < size-1 && lessThan(data[c], data[c+1])) c++;
    if (!lessThan(data[k], data[c])) break;
    swap(data, c, k);
    k = c;
  }
}
```

ปรับแถวลำดับให้เป็นฮีป ใช้เวลา O(n)

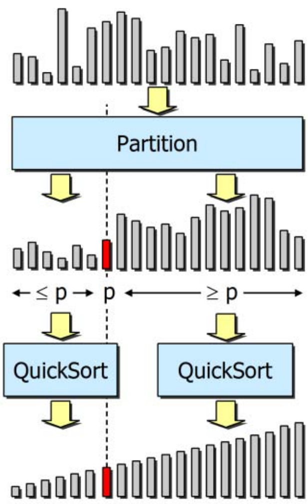
ลบตัวมากสุด นำไปไว้ตัวหลังสุด

รหัสที่ 13-11 การเรียงลำดับแบบฮีป (heap sort)

ในกรณีช้าสุดการเรียงลำดับแบบฮีปอาจต้องเปรียบเทียบข้อมูลเป็นสองเท่าของการเรียงลำดับแบบผสาน นอกจากนี้วงวนการปรับฮีปใน `fixDown` ยังซับซ้อนกว่าขั้นตอนการผสานข้อมูล ส่งผลให้การเรียงลำดับแบบฮีปจะทำงานช้ากว่าแบบผสาน แต่ต้องอย่าลืมว่า การเรียงลำดับแบบฮีปนั้นแทบไม่ต้องใช้เนื้อที่เสริมเลย ซึ่งต่างจากการเรียงลำดับแบบผสานที่ต้องจองที่เก็บเสริมขนาดเท่ากับจำนวนข้อมูลเพื่อการผสาน และต้องใช้เนื้อที่ในกองซ้อนระบบสำหรับการเรียกแบบเวียนเกิดอีกด้วย

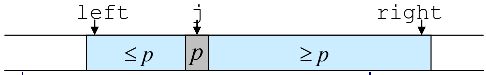
การเรียงลำดับแบบเร็ว

การเรียงลำดับแบบเร็ว (quick sort) อาศัยการ*แบ่งส่วน* (partition) ข้อมูลออกเป็นสองชุด (ไม่จำเป็นต้องมีขนาดเท่ากัน) ดังรูปที่ 13-21 ชุดซ้ายกับชุดขวา โดยเลือกข้อมูลตัวหนึ่งมาเป็น*ตัวหลัก* (pivot) ในการแบ่งส่วนข้อมูล เพื่อคัดสรรข้อมูลในกลุ่มให้ทุกตัวในชุดซ้ายมีค่าไม่มากกว่าตัวหลัก และข้อมูลในชุดขวามีค่าไม่น้อยกว่าตัวหลัก จากนั้นนำข้อมูลทั้งสองชุดไปเรียงลำดับ เมื่อทั้งชุดซ้ายและชุดขวาเรียงลำดับแล้ว จะได้ข้อมูลทั้งชุดเรียงลำดับทันที เพราะเราแบ่งส่วนให้ข้อมูลชุดซ้ายไม่มากกว่าชุดขวานั่นเอง คำถามที่ตามมาคือจะเลือกตัวหลักอย่างไร แบ่งส่วนอย่างไร และเขียนเป็นโปรแกรมได้อย่างไร ?



รูปที่ 13-21 Quick sort

เราเขียนการเรียงลำดับแบบเร็วในลักษณะเดียวกับการเรียงลำดับแบบผสาน คือเขียนแบบเวียนเกิด รหัสที่ 13-12 แสดงเมท็อด `quickSortR` เพื่อเรียงลำดับข้อมูล `d` ตั้งแต่ช่องที่ `left` ถึง `right` ถ้า `left<right` (บรรทัดที่ 102) แสดงว่า มีข้อมูลมากกว่าหนึ่งตัว ก็เรียก `partition` เพื่อแบ่งส่วน ได้เลขช่องของตัวหลักคืนกลับมา นอกจากนี้ `partition` ยังสับเปลี่ยนข้อมูลตั้งแต่ `left` ถึง `right` เพื่อให้ข้อมูลทางซ้ายของตัวหลักไม่มากกว่าตัวหลัก และข้อมูลทางขวาของตัวหลักไม่น้อยกว่าตัวหลักดังรูปที่ 13-22 หลังแบ่งส่วนเสร็จ ก็ให้ไปเรียงลำดับชุดซ้ายและชุดขวาด้วย `quickSortR` แบบเวียนเกิดในบรรทัดที่ 104 และ 105 ตามลำดับ



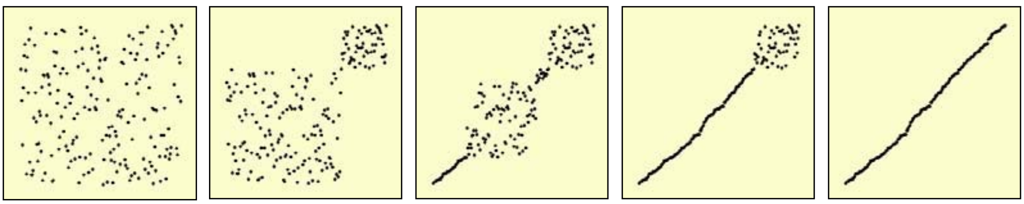
รูปที่ 13-22 ผลการแบ่งส่วนข้อมูลโดยที่ p คือตัวหลัก

```
98    public static void quickSort(Object[] d) {
99      quickSortR(d, 0, d.length-1);
100   }
101   private static void quickSortR(Object[] d, int left, int right){
102     if (left < right) {
103       int j = partition(d, left, right);
104       quickSortR(d, left, j - 1);
105       quickSortR(d, j + 1, right);
106     }
107   }
```

j คือเลขช่องของตัวหลักหลังการแบ่งส่วน

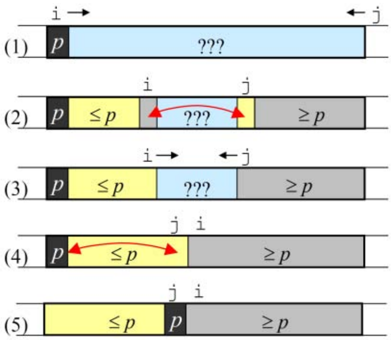
รหัสที่ 13-12 การเรียงลำดับแบบเร็ว (quick sort)

รูปที่ 13-23 แสดงภาพการเปลี่ยนแปลงข้อมูลระหว่างการเรียงลำดับแบบเร็ว รูปที่สองทางซ้ายเป็นผลจากการแบ่งส่วนข้อมูลครั้งแรก จะเห็นได้ว่า กลุ่มซ้ายอยู่ต่ำ ในขณะที่กลุ่มขวาอยู่สูง รูปต่อมาแสดงให้เห็นการแบ่งส่วนต่อแบบเวียนเกิดไปเรื่อย ๆ เนื่องจากสั่งเรียงลำดับทางซ้ายก่อนทางขวา ดังนั้นจึงเห็นข้อมูลทางซ้ายถูกแบ่งส่วนและเรียงลำดับก่อนชุดทางขวา



รูปที่ 13-23 ภาพแสดงการเปลี่ยนแปลงของข้อมูลระหว่างการเรียงลำดับแบบเร็ว

ก็มาถึงส่วนที่สำคัญที่สุดของการเรียงลำดับแบบเร็วคือขั้นตอนการแบ่งส่วนข้อมูล ผลของการแบ่งส่วนขึ้นกับตัวหลักที่เลือกใช้ ขอเลือกตัวหลักแบบง่าย ๆ (ซึ่งแน่นอนว่า จะได้ผลไม่ค่อยดี แต่ขอใช้แบบง่ายไปก่อน แล้วค่อยนำเสนอวิธีที่ดีกว่าต่อไป) คือเลือกตัวซ้ายสุดของกลุ่มเป็นตัวหลักให้ชื่อว่า



รูปที่ 13-24 การแบ่งส่วนข้อมูล

p (ดูรูปที่ 13-24) เราใช้ตัวแปร i และ j เก็บเลขที่ช่อง ตัว i เริ่มทางซ้ายมีค่าเพิ่มขึ้นทีละหนึ่งไปทางขวา ส่วนตัว j เริ่มทางขวามีค่าลดลงทีละหนึ่งมาทางซ้าย ลักษณะของข้อมูลที่อยู่ทางซ้ายของ i และทางขวา j จะเป็นดังแสดงในรูป (3) ตลอดเวลาที่ i และ j กำลังเปลี่ยนแปลง นั่นคือข้อมูลทุกตัวตั้งแต่ left ถึง i ต้องไม่มากกว่าตัวหลัก และทุกตัวจาก j ถึง right ต้องไม่น้อยกว่าตัวหลัก ตัว i จะหยุดเพิ่ม และ j จะหยุดลด เมื่อพบว่า d[i]>p และ d[j]<p ตามลำดับ ดังแสดงในรูป (2) เพราะมันเป็นสภาพที่ไม่ต้องการ ซึ่งแก้ไขได้โดยเพียงแค่สลับ d[i] กับ d[j] ทำให้เราเพิ่ม i และลด j ต่อไปได้ กระทำการเลื่อน i และ j, หยุดสลับ d[i] และ d[j] ตามเงื่อนไข จนกระทั่ง i และ j สวนกัน (คือเมื่อ i>j) ก็หยุด ปิดท้ายด้วยการสลับตำแหน่ง d[left] กับ d[j] จากรูป (4) ได้ผลดังรูป (5) เพื่อนำตัวหลักที่อยู่ซ้ายสุดมาวางที่ตำแหน่งที่ควรอยู่ เป็นอันเสร็จภาระการแบ่งส่วน

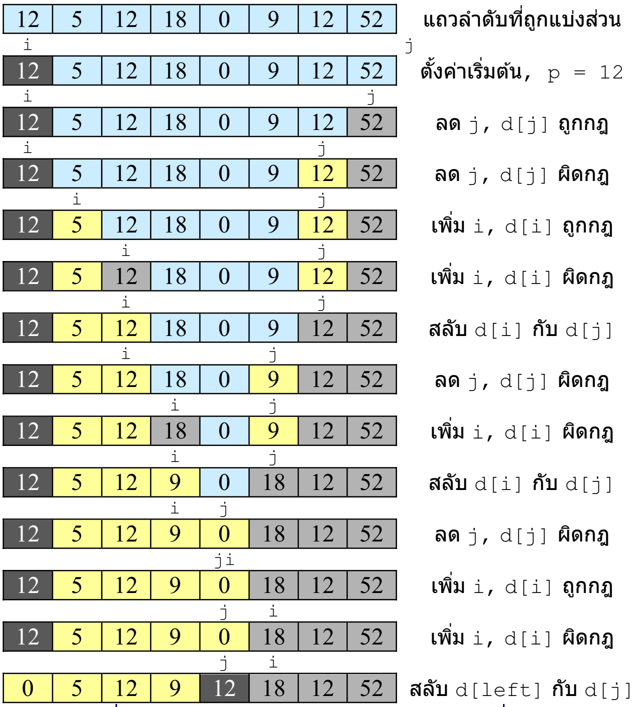
รหัสที่ 13-13 แสดงเมท็อด partition สำหรับแบ่งส่วนข้อมูลเป็นสองครึ่งใน d ตั้งแต่ช่อง left จนถึง right ตามข้อกำหนดข้างต้น เริ่มด้วยการเลือก d[left] เป็นตัวหลักเก็บในตัวแปร p (บรรทัดที่ 109) จากนั้นตั้งค่าเริ่มต้นให้กับตัวแปร i และ j แล้วเข้าวงวน (บรรทัดที่ 111) หมุนทำงานตราบเท่าที่ i ยังไม่สวนกับ j ภายในมีวงวนอีกสองวง วงแรกเลื่อน j มาทางซ้ายจนพบ p≥d[j] ที่ผิดเงื่อนไข อีกวงเลื่อน i มาทางขวาจนพบ d[i]≥p ที่ผิดเงื่อนไขเช่นกัน เมื่อถึงบรรทัดที่ 114 ถ้า i ไม่สวนกับ j ก็ให้สลับ d[i] กับ d[j] แล้วกลับไปเลื่อน j และ i ต่อเพื่อสลับข้อมูลให้ถูกต้อง เมื่อหลุดจากวงวนนอกแสดงว่า i สวนกับ j ก็ให้สลับตัวหลักที่อยู่ทางซ้ายสุดของช่วงกับตำแหน่ง j ซึ่งเป็นช่องที่ตัวหลักควรอยู่ แล้วคืนตำแหน่ง j คืนกลับไป รูปที่ 13-25 แสดงตัวอย่างการแบ่งส่วนช่วงของแถวลำดับที่ได้รับ

```
108    private static int partition(Object[] d, int left, int right) {
109      Object p = d[left];
110      int i = left, j = right + 1;
111      while (i < j) {
112        while (lessThan(p, d[--j]));
113        while (lessThan(d[++i], p)) if (i == right) break;
114        if (i < j) swap(d, i, j);
115      }
116      swap(d, left, j);
117      return j;
118    }
```

ไม่ต้องกลัวว่า j < left เพราะมี p อยู่ที่ left

ป้องกันไม่ให้เพิ่ม i จนเลย right

รหัสที่ 13-13 ขั้นตอนการแบ่งส่วน (partitioning)



รูปที่ 13-25 ตัวอย่างการแบ่งส่วนตามรหัสที่ 13-13

มีข้อข้องใจเกี่ยวกับรหัสที่ 13-13 อยู่สามประเด็น ประเด็นแรก ในวงวนของบรรทัดที่ 113 ตัว i เพิ่มค่าไปทางขวาขึ้นเรื่อย ๆ จึงมีโอกาสเกิน right จึงต้องมีการตรวจสอบภายในวงวน แต่ทำไมวงวนของบรรทัดที่ 112 ไม่ต้องตรวจสอบ j ว่า ตกขอบ left ที่เป็นเช่นนี้ก็เพราะเรามั่นใจว่า ตัวหลักนั้นอยู่ที่ d[left] จึงต้องหลุดจากวงวนแน่นอนก่อนที่จะเลย left ไป ประเด็นที่สอง ทำไมไม่เขียนเงื่อนไขของวงวน while ทั้งสองให้เหมือนกับที่อธิบายตอนต้นว่า จะยังคงเพิ่ม i ตราบเท่าที่ d[i]≤p และลด j ตราบเท่าที่ d[j]≥p ลองคิดดู สมมติว่า d[i] และ d[j] มีค่าเท่ากับ p ตามรหัสที่ 13-13 จะหลุดจากวงวน แล้วสลับ d[i] และ d[j] ที่มีค่าเหมือนกัน ซึ่งดูไร้เหตุผลสิ้นดี เสียเวลาโดยเปล่าประโยชน์ เหตุผลที่ทำเช่นนี้เพราะถ้าเป็นไปได้เราต้องการให้ตำแหน่งที่คืนจาก partition อยู่ตรงกลาง ๆ ช่วงที่ให้แบ่งส่วน เช่น สมมติว่า ข้อมูลในช่วงที่จะแบ่งส่วนมีค่าเท่ากันหมด การทำตามรหัสที่ 13-13 แน่นอนว่า จะเกิดการสลับที่ "ไร้ประโยชน์" เป็นจำนวนมาก และในที่สุด i และ j จะมาสวนกันตรงกลางช่วง ได้ตำแหน่ง j ที่คืนไปอยู่ตรงกลาง ๆ ทำให้แบ่งส่วนแล้วได้ครึ่งซ้ายและขวามีขนาดพอ ๆ กัน ในขณะที่ถ้าเราเปลี่ยนเงื่อนไข เพื่อป้องกันไม่ให้เกิดการสลับค่าที่เหมือนกัน จะทำให้ j วิ่งมาหยุดที่ขอบซ้าย (เราต้องป้องกัน j ตกขอบซ้ายด้วย) ทำให้แบ่งส่วนแล้วได้ครึ่งซ้ายไม่มีข้อมูล ซึ่งจะส่งผลไม่ดีต่อประสิทธิภาพการทำงานโดยรวมซึ่งจะได้วิเคราะห์ต่อไป

ข้อข้องใจประเด็นสุดท้ายคือมีวิธีทำให้วงวนของบรรทัดที่ 113 ไม่ต้องป้องกันการตกขอบขวาของช่วงหรือไม่ เพราะถ้าทำได้ วงวนทั้งสองเล็กลงไปอีก ส่งผลให้ทำงานน้อยลง อีกทั้งรหัสเครื่องที่แทนวงวนดังกล่าวสามารถถูกเก็บในหน่วยความจำความเร็วสูงของหน่วยประมวลผลตลอดการหมุน ทำให้ทำงานได้เร็วมากขึ้นอีก ซึ่งเราจะได้แสดงวิธีการปรับปรุงต่อไป โดยวงวนที่เล็กในขั้นตอนการแบ่งส่วนนี่เองที่ทำให้การเรียงลำดับแบบนี้ทำงานได้เร็วสมชื่อ

การเรียงลำดับแบบเร็ว เร็วจริงหรือไม่ ขอวิเคราะห์ทางคณิตศาสตร์กันก่อน แล้วค่อยทำการทดลองวัดเวลาจริงกันทีหลัง กำหนดให้ $c(n)$ แทนจำนวนการเปรียบเทียบข้อมูลในการเรียงลำดับแบบเร็วกับข้อมูล n ตัว (ซึ่งก็คือการวิเคราะห์เมท็อด `quickSortR` ในรหัสที่ 13-12 หน้าที่ 321 โดยจำนวนข้อมูลก็คือ `right-left+1` ตัว) ประกอบด้วยจำนวนการเปรียบเทียบข้อมูลระหว่างการแบ่งส่วน บวกกับจำนวนการเปรียบเทียบในการเรียงลำดับข้อมูลครึ่งซ้ายและครึ่งขวาที่ได้จากการแบ่งส่วน สมมติว่า หลังการแบ่งส่วน ครึ่งซ้ายมีข้อมูล k ตัว จะได้ครึ่งขวามีข้อมูล $n-k-1$ ตัว (อย่าลืมว่า เราไม่ต้องส่งตัวหลักไปเรียงลำดับต่อ) การแบ่งส่วนด้วย `partition` ในรหัสที่ 13-13 ต้องเปรียบเทียบจำนวน $n-1$ ครั้ง เนื่องจากมีข้อมูลหนึ่งตัวถูกเลือกเป็นตัวหลัก และนำข้อมูลที่เหลือ $n-1$ ตัวมาเปรียบเทียบกับตัวหลัก ดังนั้น $c(n) = c(k) + c(n-k-1) + (n-1)$ โดยที่ $k = 0, 1, \ldots,$ หรือ $n-1$

กำหนดให้ $c_{\min}(n)$ แทน $c(n)$ ที่มีค่าน้อยสุด ซึ่งจะเกิดขึ้นเมื่อ $k = n/2$ โดยทุก ๆ ครั้งที่เรียงลำดับชุดย่อยก็ต้องเป็นในลักษณะที่น้อยสุดด้วย จะได้ว่า

$$\begin{aligned} c_{\min}(n) &= c_{\min}\left(\lfloor n/2 \rfloor\right) + c_{\min}\left(n - \lfloor n/2 \rfloor - 1\right) + (n-1) \\ &\le 2c_{\min}\left(n/2\right) + (n-1) \\ &= n\log_2 n - n + 1 \end{aligned}$$

ได้ความสัมพันธ์เวียนเกิดที่เหมือนกับของการเรียงลำดับแบบผสานที่ได้วิเคราะห์แล้ว (ในหน้าที่ 317) สรุปว่า ใช้เวลาการทำงานกรณีเร็วสุดเป็น $\Theta(n \log n)$

ในกรณีช้าสุด กำหนดให้ $c_{\max}(n)$ แทน $c(n)$ ที่มีค่ามากสุด ซึ่งจะเกิดขึ้นเมื่อ $k = 0$ (หรือ $n-1$) โดยทุก ๆ ครั้งที่เรียงลำดับชุดย่อย ก็ต้องเป็นในลักษณะที่มากสุดด้วย จะได้ว่า

$$\begin{aligned} c_{\max}(n) &= c_{\max}\left(0\right) + c_{\max}\left(n-1\right) + (n-1) \\ &= c_{\max}\left(n-2\right) + (n-2) + (n-1) \\ &\cdots \\ &= c_{\max}\left(1\right) + 1 + 2 + \ldots + (n-3) + (n-2) + (n-1) \\ &= n(n-1)/2 \\ &= \Theta(n^2) \end{aligned}$$

ซึ่งไม่คู่ควรกับชื่อ "แบบเร็ว" ที่ตั้งให้เลย มีศักดิ์ศรีเทียบเท่าแบบฟอง แบบเลือก แบบแทรก แต่ต้องอย่าลืมว่า ผลข้างบนนี้สำหรับกรณีช้าสุด แบบเลือกนั้น $\Theta(n^2)$ ตลอด ไม่ว่าจะเร็วสุดหรือช้าสุด สำหรับแบบฟองและแบบแทรก กรณีเร็วสุดเป็น $\Theta(n)$ ช้าสุดเป็น $\Theta(n^2)$ แต่ที่แย่คือ กรณีเฉลี่ยก็เป็น $\Theta(n^2)$ แล้วแบบเร็วที่เรากำลังสนใจอยู่เป็นเช่นไร กรณีเร็วสุด $\Theta(n \log n)$ ช้าสุด $\Theta(n^2)$ แล้วกรณีเฉลี่ยคืออะไร ต้องมาวิเคราะห์กัน

กำหนดให้ $c_{\text{avg}}(n)$ แทน $c(n)$ ในกรณีเฉลี่ย สมมติให้ข้อมูลทั้ง n ตัวมีค่าแตกต่างกันหมด ให้โอกาสที่ตัวหลักจะอยู่ที่ตำแหน่งใด ๆ มีพอ ๆ กัน ดังนั้นโอกาสที่ k จะมีค่า 0, 1, 2, ..., หรือ n–1 ก็มีค่าเท่า ๆ กันด้วย ดังนั้น

$$
\begin{aligned}
c_{\text{avg}}(n) &= \frac{1}{n}\sum_{k=0}^{n-1}\left(c_{\text{avg}}(k) + c_{\text{avg}}(n-k-1)\right) + (n-1) \\
&= \frac{2}{n}\sum_{k=0}^{n-1} c_{\text{avg}}(k) + (n-1)
\end{aligned}
$$

พจน์ $c_{\text{avg}}(k)$ และ $c_{\text{avg}}(n-k-1)$ เขียนต่างกัน แต่ถ้าแปร k จาก 0 ถึง $n-1$ พจน์ทั้งสองจะถูกแจงออกมาเหมือนกัน สามารถหาผลเฉลยได้ดังนี้ (การวิเคราะห์ข้างล่างนี้เหมือนกับการหาความลึกเฉลี่ยของปมในต้นไม้ค้นหาแบบทวิภาคที่เคยนำเสนอมาในบทที่ 10)

$$nc_{\text{avg}}(n) = 2\sum_{k=0}^{n-1} c_{\text{avg}}(k) + n(n-1)$$

คูณ n ตลอด

$$(n-1)c_{\text{avg}}(n-1) = 2\sum_{k=0}^{n-2} c_{\text{avg}}(k) + (n-1)(n-2)$$

เปลี่ยน n เป็น n–1

$$nc_{\text{avg}}(n) = (n+1)c_{\text{avg}}(n-1) + 2(n-1)$$

นำสองความสัมพันธ์ข้างต้นมาลบกัน ย้าย $(n-1)c_{\text{avg}}(n-1)$ มาทางขวา

$$
\begin{aligned}
\frac{c_{\text{avg}}(n)}{n+1} &= \frac{c_{\text{avg}}(n-1)}{n} + \frac{2(n-1)}{n(n+1)} \\
&= \frac{c_{\text{avg}}(1)}{2} + 2\sum_{i=2}^{n}\frac{(i-1)}{i(i+1)} \\
&\approx 2\sum_{i=2}^{n}\frac{1}{(i+2)} \\
&= 2\left(\frac{1}{4} + \frac{1}{5} + \cdots + \frac{1}{n} + \frac{1}{n+1} + \frac{1}{n+2}\right) \\
&= 2\left(\left(H_n - 1 - \frac{1}{2} - \frac{1}{3}\right) + \frac{1}{n+1} + \frac{1}{n+2}\right)
\end{aligned}
$$

หาร $n(n+1)$ ตลอด แล้วคลี่ความสัมพันธ์เวียนเกิด โดยที่ $c_{\text{avg}}(1) = 0$ จากนั้นใช้การประมาณ $\frac{(i-1)}{i(i+1)} \approx \frac{1}{i+2}$ เพื่อเปลี่ยนพจน์ในผลรวมให้ง่ายขึ้น ซึ่งเมื่อเขียนแจกแจงออกมาพบว่า สามารถเขียนในรูปของจำนวนฮาร์มอนิก H_n ซึ่งเท่ากับ $1+\frac{1}{2}+\frac{1}{3}+\ldots+\frac{1}{n}$ มีค่าประมาณ $\ln n + \gamma$ โดยที่ γ คือค่าคงตัวของออยเลอร์ มีค่าประมาณ 0.577...

$$
\begin{aligned}
c_{\text{avg}}(n) &\approx 2(n+1)\left(H_n - \frac{11}{6} + \frac{1}{n+1} + \frac{1}{n+2}\right) \\
&\approx 2nH_n + 2H_n - \frac{11n}{3} + \frac{1}{3} \\
&\approx 2n(\ln n + \gamma) + 2(\ln n + \gamma) - \frac{11n}{3} + \frac{1}{3} \\
&\approx 1.39 n \log_2 n + O(n) \\
&= \mathrm{O}(n \log n)
\end{aligned}
$$

คูณด้วย $(n+1)$ ตลอด แล้วประมาณให้ $(n+1)/(n+2)$ มีค่าประมาณ 1 เมื่อ n มีค่ามาก

ประมาณค่าของ H_n ด้วย $\ln n + \gamma$

สรุปได้ว่า $c_{\text{avg}}(n) = \mathrm{O}(n \log n)$

สรุปได้ว่า การเรียงลำดับแบบเร็ว กรณีเร็วสุดใช้เวลา $\Theta(n \log n)$ กรณีช้าสุด $\Theta(n^2)$ กรณีเฉลี่ยใช้เวลา $\mathrm{O}(n \log n)$ โดยกรณีเฉลี่ยช้ากว่ากรณีเร็วสุดเพียงประมาณ 39% จึงนับว่าเป็นข่าวดี

อย่างไรก็ตามเราต้องระวังไม่ให้กรณีช้าสุดเกิดขึ้นได้ง่าย จะช้าหรือเร็ว ขึ้นกับขั้นตอนการแบ่งส่วน สำหรับวิธีที่เราได้นำเสนอมาจะเห็นว่า หากข้อมูลเริ่มต้นเรียงลำดับเรียบร้อยแล้ว ซึ่งน่าจะเป็นผลดี แต่ขอบอกว่า กลับเป็นกรณีที่แย่มาก ๆ เพราะการแบ่งส่วนข้อมูลด้วยรหัสที่ 13-13 จะเลือกตัวซ้ายสุด ซึ่งมีค่าน้อยสุดเป็นตัวหลัก ทำให้แบ่งส่วนแล้วไม่มีตัวใดน้อยกว่าตัวหลัก ครึ่งซ้ายของตัวหลักจึงไม่มี เมื่อแบ่งส่วนชุดเล็กลงก็ยังเป็นสภาพนี้เช่นกัน ตรงกับกรณีช้าสุดที่ได้วิเคราะห์มา

ถ้าผู้อ่านทดลองใช้รหัสที่ 13-12 และรหัสที่ 13-13 ประกอบกันเพื่อเรียงลำดับข้อมูลที่เรียงอยู่แล้วสักประมาณ 2,000 ตัว จะพบว่า ทำงานช้ากว่าเรียงลำดับข้อมูลสุ่ม อย่างเห็นได้ชัด และถ้าทดลองเรียงลำดับข้อมูลที่เรียงลำดับแล้วในปริมาณที่มากขึ้น เช่น สัก 10,000 ตัว จะพบข้อน่าตกใจอีกประการหนึ่งคือจะเกิด `StackOverflowError` นั่นเป็นเพราะการแบ่งส่วนข้อมูลที่เรียงแล้ว n ตัว จะไม่มีข้อมูลในครึ่งซ้าย ต้องไปเรียงครึ่งขวา $n-1$ ตัว แบ่งส่วนและทำต่อ ก็ไปเรียงครึ่งขวา $n-2$ ตัว เป็นเช่นนี้ไปเรื่อย ๆ จนเหลือ 1 ตัว การเรียงลำดับแต่ละครั้งคือการเรียกเมท็อด `quickSortR` หนึ่งครั้ง ซึ่งเรียกซ้อนจากครั้งก่อนจึงใช้เนื้อที่ในกองซ้อนของระบบเป็นจำนวน $n-1$ ครั้ง ซึ่งถ้ามากเกินกว่าที่ระบบยอมให้ใช้ จะเกิดข้อผิดพลาด ถ้าพิจารณาการเรียงลำดับแบบผสาน ก็พบว่า มีการเรียกแบบเวียนเกิดในทำนองเดียวกัน แต่จะใช้เนื้อที่กองซ้อนของระบบเป็นจำนวน $\lceil \log_2 n \rceil$ เพราะเราแบ่งข้อมูลออกเป็นสองครึ่ง ๆ ละเท่า ๆ กัน จึงเรียกแบบเวียนเกิดเพียง $\lceil \log_2 n \rceil$ ครั้ง ถ้า $n = 10{,}000{,}000$ จะได้ $\lceil \log_2 n \rceil$ มีค่าเพียง 24 เท่านั้น ในขณะที่ถ้าใช้การเรียงลำดับแบบเร็วที่ได้นำเสนอมาจะใช้เนื้อที่ในกองซ้อนเกือบสิบล้าน ดังนั้นกรณีแย่สุด ๆ ของการเรียงลำดับแบบเร็วนั้นมีปัญหาทั้งเรื่องเวลาและเนื้อที่

แล้วจะแก้ปัญหานี้ได้อย่างไร ? วิธีง่าย ๆ ก็คือต้องพยายามเลือกตัวหลักไม่ให้เกิดเหตุการณ์ดังกล่าว วิธีที่ปลอดภัยสุด ๆ คือเลือกมัธยฐาน (median) ของกลุ่มเป็นตัวหลัก ซึ่งประกันได้ว่า ครึ่งซ้ายและครึ่งขวาหลังการแบ่งส่วนต้องมีปริมาณพอกัน ซึ่งเป็นกรณีที่ดีสุด แต่ต้องเข้าใจด้วยว่า การหาตัวมัธยฐานนั้นใช้เวลาพอสมควร ทำให้ไม่คุ้ม อีกวิธีหนึ่งคือการเลือกตัวหลักอย่างสุ่มจากกลุ่มข้อมูล ก็

พอทำให้มั่นใจได้ว่า โอกาสที่เราจะโชคร้ายเลือกสุ่มทุกครั้งในทุกระดับของการเรียก แล้วได้การแบ่งส่วนที่แย่สุดทุกครั้ง คงมีน้อยมาก ๆ รหัสที่ 13-14 แสดงการเลือกตำแหน่งในช่วงที่ได้รับอย่างสุ่มเพื่อสลับข้อมูลตำแหน่งนั้นกับตัวซ้ายสุด แล้วเข้าสู่การแบ่งส่วนของเดิมตามปกติ

```
private static int partition(Object[] d, int left, int right) {
  int m = left + (int)(Math.random()*(right - left + 1));
  swap(d, left, m);
  Object p = d[left];
  ...
}
```

สุ่มตำแหน่งระหว่าง left ถึง right

สลับตัวที่สุ่มกับตัวซ้ายเพื่อใช้เป็นตัวหลัก

รหัสที่ 13-14 การแบ่งส่วนโดยเลือกตัวหลักแบบสุ่ม

มีการแบ่งส่วนอีกแบบที่ใช้ได้ดีในทางปฏิบัติ อีกทั้งทำให้วงวนภายในการแบ่งส่วนทำงานได้เร็วขึ้นด้วย คือการใช้ตัวหลักที่เป็นตัวมัธยฐานของข้อมูลสามตัวจากกลุ่มข้อมูล (median-of-three partitioning) ข้อมูลทั้งสามตัวนี้เลือกมาจากตำแหน่ง `left`, `right` และตำแหน่งตรงกลางซึ่งคือ `(left+right)/2` จากนั้นสลับข้อมูล 3 ตัวนี้ให้ตัวน้อยอยู่ที่ `(left+right)/2` ตัวมัธยฐานของสามตัวนี้อยู่ที่ `left` และตัวมากอยู่ที่ `right` ดูตัวอย่างในรูปที่ 13-26 ข้อมูล 3 ตัวที่สนใจคือ 11, 18, และ 12 ตัวมัธยฐานของ 3 ตัวนี้คือ 12 ดังนั้นจึงให้อยู่ที่ `left` ตัวมากคือ 18 ไปอยู่ที่ `right` และตัวน้อยคือ 11 ไปอยู่ตรงกลาง จากนั้นสั่งให้แบ่งส่วนด้วยวิธีเดิม โดยให้ `j` ในรหัสที่ 13-13 เริ่มที่ `right` ได้ (ของเดิมให้ `j=right+1`) และในวงวนการเพิ่มค่า `i` ก็ไม่ต้องกลัวตกขอบขวาอีกต่อไป คือไม่ต้องมีการตรวจสอบ `i==right` เพราะมีตัวไม่น้อยกว่าตัวหลักอยู่ที่ช่อง `right` แล้ว ทำให้วงวนการแบ่งส่วนเล็กลงและเร็วขึ้น รหัสที่ 13-15 แสดงรายละเอียดการแบ่งส่วนที่ได้นำเสนอมา (สี่บรรทัดแรกรับผิดชอบจัดข้อมูลในช่องซ้าย ขวา และตรงกลาง ขอให้ผู้อ่านลองทำความเข้าใจเอง)

11	5	15	18	0	9	10	12	ข้อมูล 3 ตัวที่สนใจ
12	5	15	11	0	9	10	18	จัดข้อมูล 3 ตัวนี้ตามกฎ
i							j	
12	5	15	11	0	9	10	18	ให้ `i=left,j=right`

รูปที่ 13-26 การใช้มัธยฐานจากข้อมูลสามตัวเป็นตัวหลัก

วิธีการเลือกตัวหลักข้างต้นให้ผลดี (จากการทดลองพบว่า ใช้เวลาน้อยกว่าการเลือกตัวซ้ายเป็นตัวหลัก และเลือกตัวหลักแบบสุ่ม) แต่ขอให้ระวังด้วยว่า ผู้ไม่ประสงค์ดีที่รู้วิธีการเลือกตัวหลักของเราสามารถออกแบบชุดข้อมูลที่ "แกล้ง" ให้เกิดการแบ่งส่วนที่แย่สุดได้ ส่งผลให้การเรียงลำดับใช้เวลานาน หรือไม่ก็เกิดกรณีเนื้อที่ของกองซ้อนไม่พอ รูปที่ 13-27 แสดงตัวอย่างชุดข้อมูลที่ไม่ดี ทำให้การแบ่งส่วนข้อมูล n ตัวเหลือครึ่งขวามีข้อมูล $n - 2$ ตัวตลอด ถ้าวิเคราะห์จะพบว่า ใช้เวลาการทำงานเป็น $\Theta(n^2)$ จึงเป็นสิ่งที่ควรระวังเมื่อใช้การเรียงลำดับแบบเร็ว หากยอมรับสภาพนี้ไม่ได้ ก็คงต้องหันไปเลือกตัวหลักแบบสุ่ม หรือใช้วิธีการเรียงลำดับแบบอื่น เช่น แบบผสาน หรือแบบฮีป เป็นต้น

```
  private static int partition(Object[] d, int left, int right) {
    int center = (left + right)/2;
    if (lessThan(d[left],d[center]))  swap(d, left, center);
    if (lessThan(d[right],d[center])) swap(d, center, right);
    if (lessThan(d[right],d[left]))   swap(d, left, right);
    Object p = d[left];
    int i = left, j = right;
    while (i < j) {
      while (lessThan(p, d[--j]));
      while (lessThan(d[++i], p));
      if (i < j) swap(d, i, j);
    }
    swap(d, left, j);
    return j;
  }
```

สามบรรทัดนี้ทำให้
d[center] ≤ d[left] ≤ d[right]

รหัสที่ 13-15 ขั้นตอนการแบ่งส่วนโดยใช้มัธยฐานจากข้อมูลสามตัวเป็นตัวหลัก

1	5	3	7	2	4	6	8	9	**ข้อมูล เริ่มต้น** 9 **ตัว**
1	2	3	7	5	4	6	8	9	**แบ่งส่วนครั้งที่** 1 **เหลือครึ่งขวา** 7 **ตัว**
1	2	3	4	5	7	6	8	9	**แบ่งส่วนครั้งที่** 2 **เหลือครึ่งขวา** 5 **ตัว**
1	2	3	4	5	6	7	8	9	**แบ่งส่วนครั้งที่** 3 **เหลือครึ่งขวา** 3 **ตัว**
1	2	3	4	5	6	7	8	9	**แบ่งส่วนครั้งที่** 4 **เหลือครึ่งขวา** 1 **ตัว**

รูปที่ 13-27 ตัวอย่างชุดข้อมูลที่ไม่เหมาะกับการใช้มัธยฐานจากข้อมูลสามตัวเป็นตัวหลัก

คลาส Arrays (ในชุด java.util) มีเมท็อด sort ให้บริการเรียงลำดับข้อมูล ซึ่งสามารถรับแถวลำดับของจำนวนที่เป็นข้อมูลพื้นฐาน เช่น int[], double[] เป็นต้น และรับแถวลำดับของอ็อบเจกต์ (Object[]) ได้ด้วย หากเป็นเมท็อดที่รับแถวลำดับของข้อมูลพื้นฐาน จะใช้การเรียงลำดับแบบเร็ว ถ้าเป็นเมท็อดที่รับแถวลำดับของอ็อบเจกต์จะใช้การเรียงลำดับแบบผสาน ทั้งนี้ก็เพราะต้องการความเสถียรของข้อมูลในการเรียงลำดับอ็อบเจกต์ (นั่นคือตำแหน่งสัมพัทธ์ของอ็อบเจกต์ที่เท่ากันไม่เปลี่ยนแปลงหลังการเรียงลำดับ) ในที่ขณะที่ถ้าเป็นข้อมูลพื้นฐานแล้วเราไม่ต้องคำนึงถึงความเสถียรของข้อมูล (เพราะจำนวนที่เท่ากันแยกกันไม่ออก) จึงเลือกใช้แบบเร็วเพราะเร็วกว่าแบบผสาน

การเรียงลำดับแบบเร็วที่ใช้ในคลาส Arrays นั้นมีการปรับการทำงานให้เร็วขึ้นด้วยกลวิธีที่นำเสนอในบทความของ Jon L. Bentley และ M. Douglas McIlroy's เรื่อง "Engineering a Sort Function", วารสาร Software-Practice and Experience, Vol. 23(11) P. 1249-1265 (November 1993). ดังนี้

1. ถ้าจำนวนข้อมูลมีน้อยกว่า 7 ตัว จะใช้การเรียงลำดับแบบแทรก
2. ถ้าจำนวนข้อมูลอยู่ระหว่าง 7 ถึง 39 ตัว จะเรียงลำดับแบบเร็วโดยเลือกมัธยฐานจากข้อมูลตัวซ้าย กลาง ขวา
3. ถ้าจำนวนข้อมูลมีตั้งแต่ 40 ตัวขึ้นไป จะเรียงลำดับแบบเร็ว โดยใช้เลือกข้อมูลสามชุด ชุดละ 3 ตัว, หามัธยฐานของแต่ละชุด, แล้วหามัธยฐานจากมัธยฐาน 3 ตัวที่หาได้มา เพื่อใช้เป็นตัวหลัก

การเปรียบเทียบวิธีเรียงลำดับแบบต่าง ๆ

เพื่อให้เห็นความแตกต่างของเวลาการทำงานของการเรียงลำดับข้อมูลแบบต่าง ๆ ที่ได้นำเสนอมา ผู้เขียนได้ทดลองเรียงลำดับข้อมูลปริมาณต่าง ๆ กันตั้งแต่ 200 ตัว เพิ่มขนาดขึ้นทีละสองเท่า โดยมีลักษณะของข้อมูลเริ่มต้นต่างกันสามแบบ คือแบบเรียงลำดับแล้ว แบบเรียงกลับลำดับ และแบบสุ่ม ได้ผลดังแสดงในตารางที่ 13-3, ตารางที่ 13-4, และตารางที่ 13-5 ตามลำดับ โดยใช้รหัสต่าง ๆ ที่ได้นำเสนอมาตั้งแต่รหัสที่ 13-2 จนถึงรหัสที่ 13-15 สำหรับแบบเชลล์ใช้ลำดับ *h* ของ Sedgewick แบบผสานใช้แบบที่ลดจำนวนการย้ายข้อมูลในรหัสที่ 13-10 และแบบเร็วใช้มัธยฐานของข้อมูลสามตัวเป็นตัวหลักในการแบ่งส่วนในรหัสที่ 13-15 สังเกตได้อย่างชัดเจนว่า การเรียงลำดับแบบเลือกนั้นช้าคงเส้นคงวาตลอด ไม่ขึ้นกับลักษณะเริ่มต้นของข้อมูล สำหรับแบบฟองและแบบแทรกทำงานได้เร็วมากกับข้อมูลที่เรียงลำดับแล้ว แต่พอเป็นกรณีอื่นก็ช้า เวลาการทำงานของสามแบบแรกนี้เมื่อข้อมูลเพิ่มจำนวนขึ้นเป็น 2 เท่าจะใช้เวลาเพิ่มขึ้นประมาณ 4 เท่าซึ่งสอดคล้องกับผลการวิเคราะห์ที่ใช้เวลาเป็น $O(n^2)$ การเรียงลำดับแบบเชลล์ให้ผลที่ดีมาก เร็วกว่าแบบฮีปด้วยซ้ำไป แต่ก็ต้องอย่าลืมว่า มีอัตราการเพิ่มของเวลาการทำงานที่มากกว่า และแบบเร็วก็เร็วสมชื่อ เร็วกว่าทุกแบบที่นำเสนอมา

ตารางที่ 13-3 การทดลองเรียงลำดับข้อมูลที่เรียงลำดับอยู่แล้ว (เวลาที่แสดงเป็นมิลลิวินาที)

n	เลือก	ฟอง	แทรก	เชลล์	ผสาน	ฮีป	เร็ว
200	0.44	0.01	0.01	0.03	0.06	0.11	0.04
400	1.80	0.01	0.02	0.06	0.13	0.25	0.10
800	7.01	0.02	0.03	0.18	0.28	0.56	0.21
1,600	28.59	0.04	0.06	0.29	0.59	1.24	0.44
3,200	112.64	0.07	0.11	0.73	1.28	2.78	0.92
6,400	459.64	0.14	0.23	1.54	2.81	8.68	2.10
12,800	1814.66	0.28	0.45	3.48	6.32	13.29	4.38
25,600	7240.25	0.57	1.00	12.10	12.21	29.54	9.18
51,200	29179.25	1.25	1.91	17.31	29.54	63.55	19.31
102,400	123337.25	2.64	3.91	61.43	58.51	143.34	41.31

ตารางที่ 13-4 การทดลองเรียงลำดับข้อมูลที่เรียงลำดับกลับลำดับ (เวลาที่แสดงเป็นมิลลิวินาที)

n	เลือก	ฟอง	แทรก	เชลล์	ผสาน	ฮีป	เร็ว
200	0.44	0.81	0.55	0.06	0.06	0.10	0.05
400	1.80	3.23	2.29	0.15	0.13	0.24	0.10
800	7.36	12.86	9.73	0.29	0.28	0.66	0.20
1,600	28.43	51.01	35.18	0.64	0.60	1.21	0.45
3,200	112.62	210.30	141.22	1.36	1.29	2.70	0.93
6,400	458.97	844.04	573.99	4.57	2.75	5.92	2.10
12,800	1816.83	3349.86	2266.79	6.55	6.06	12.35	4.36
25,600	7250.50	13449.50	8988.00	15.49	14.56	26.95	9.34
51,200	29249.75	53754.75	36625.25	31.10	27.49	58.67	19.83
102,400	124168.50	221318.25	153943.75	102.35	58.67	133.79	42.68

ตารางที่ 13-5 การทดลองเรียงลำดับข้อมูลแบบสุ่ม (เวลาที่แสดงเป็นมิลลิวินาที)

n	เลือก	ฟอง	แทรก	เชลล์	ผสาน	ฮีป	เร็ว
200	0.45	0.19	0.30	0.08	0.07	0.11	0.05
400	1.82	3.13	1.10	0.19	0.15	0.25	0.11
800	7.02	15.38	4.38	0.45	0.32	0.57	0.24
1,600	31.01	49.71	16.93	1.10	0.68	1.27	0.52
3,200	112.86	207.47	74.17	2.49	1.48	2.82	1.11
6,400	452.57	823.90	284.56	5.83	3.86	6.44	2.63
12,800	1834.78	3318.25	1147.65	12.11	9.68	16.01	7.16
25,600	7273.00	13349.25	4639.25	27.83	19.62	31.17	15.65
51,200	30183.50	56241.00	20123.75	62.43	43.83	70.25	37.85
102,400	200588.50	332638.25	171231.25	184.83	102.82	201.17	86.38

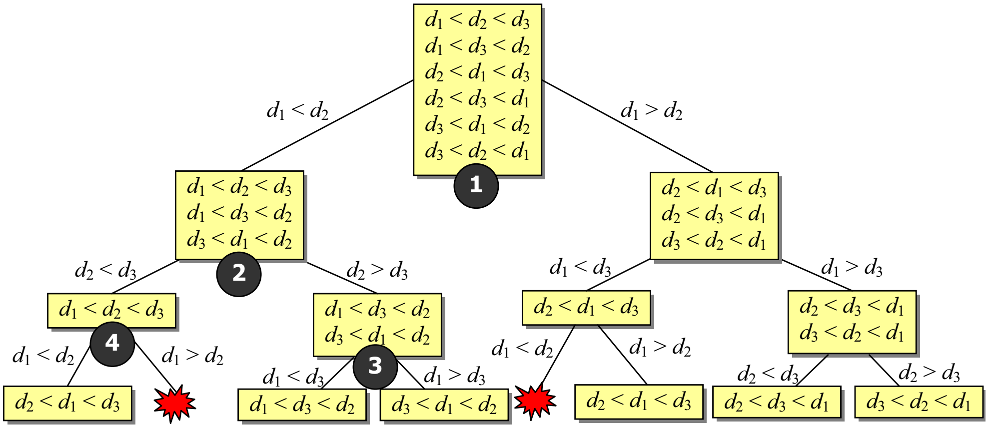
ขอบเขตล่างของเวลาการเรียงลำดับ

เราได้ศึกษากันมาแล้วว่า การเรียงลำดับแบบเลือกใช้เวลา $\Theta(n^2)$ แบบฟองและแบบแทรกใช้เวลา $O(n^2)$ แบบเชลล์ใช้เวลา $O(n^{1.xx})$ แบบเร็วใช้เวลา $O(n^2)$ แต่ในกรณีเฉลี่ยใช้เวลา $O(n \log n)$ ในขณะที่แบบผสานและแบบฮีปใช้เวลา $O(n \log n)$ คำถามที่น่าสนใจก็คือมีวิธีอื่นไหมที่ดีกว่านี้ เช่น ใช้เวลา $O(n)$ หรือ $O(n\sqrt{\log n})$ เป็นต้น คงเห็นได้ชัดเจนว่า การเรียงลำดับใด ๆ อย่างน้อยก็ต้องพิจารณาข้อมูลทุกตัว ดังนั้นต้องใช้เวลา $\Omega(n)$ (เตือนความจำเล็กน้อยว่า เราใช้ O เพื่อบอกขอบเขตบน และใช้ Ω เพื่อบอกขอบเขตล่าง) และถ้ายังไม่ลืม การเรียงลำดับแบบฐาน (radix sort) นั้นใช้เวลา $O(kn)$ โดยที่ k คือจำนวนหลักของข้อมูล ถ้าเราต้องการเรียงลำดับจำนวนเต็มทั่วไป เช่น `int` จะได้ว่า k เป็นค่าคงตัว ดังนั้นแบบฐานก็ใช้เวลา $O(n)$ ซึ่งดีกว่าทั้งเจ็ดแบบที่เราอธิบายกันมา แต่ก็ต้องอย่าลืมว่า ทั้งเจ็ดแบบนี้เป็นแบบที่อาศัยการนำข้อมูลทีละคู่มาเปรียบเทียบกัน เพื่อตัดสินใจว่า จะสลับตำแหน่งของข้อมูลหรือไม่อย่างไร ไม่เหมือนกับแบบฐานที่ใช้การแตกข้อมูลออกเป็นส่วน ๆ เพื่อใช้ประกอบการพิจารณาเรียงลำดับข้อมูลใหม่ ซึ่งไม่แน่เสมอไปว่า ข้อมูลที่สนใจเรียงลำดับจะแตกเป็นส่วน ๆ ได้ สิ่งที่เราจะสนใจในหัวข้อนี้ก็คือ "การเรียงลำดับข้อมูลแบบที่ใช้การเปรียบเทียบข้อมูลทีละคู่เป็นเครื่องมือหลักในการตัดสินใจสลับตำแหน่งข้อมูล (comparison-based sorting) ต้องเปรียบเทียบข้อมูลอย่างน้อยกี่ครั้งเพื่อเรียงลำดับข้อมูลจำนวน n ตัว"

เราจะใช้*ต้นไม้ตัดสินใจ* (decision tree) ช่วยหาคำตอบของคำถามข้างต้น สมมติว่า เราต้องการเรียงลำดับข้อมูล 3 ตัว d_1, d_2, และ d_3 ด้วยการเรียงลำดับแบบเลือก การทำงานเริ่มด้วยการหาตัวมากสุดในสามตัวได้ผลไปสลับกับตัวที่สาม ตามด้วยการหาตัวมากสุดของสองตัวที่เหลือ ได้ผลไปสลับกับตัวที่สอง ก็เป็นอันจบการทำงาน ซึ่งสามารถแทนการทำงานดังกล่าวได้ดังรูปที่ 13-28

กำหนดให้ข้อมูลทั้ง 3 ตัวมีค่าต่างกันหมด แต่ละปมในต้นไม้แทนลำดับของข้อมูล 3 ตัวที่เป็นไปได้ที่จะเรียงจากน้อยไปมาก หนึ่งปมแตกกิ่งออกสองกิ่ง โดยอาศัยผลของการเปรียบเทียบข้อมูล

หนึ่งคู่ น้อยกว่าไปด้านซ้าย มากกว่าไปด้านขวา เริ่มที่รากของต้นไม้ (ปมหมายเลข 1) แทนลำดับของข้อมูลที่เป็นไปได้ 6 แบบ เพราะมีข้อมูล 3 ตัว จึงมีลำดับที่เป็นไปได้ 3! = 6 แบบ เริ่มต้นเราต้องหาค่ามากสุดจากข้อมูล 3 ตัว จึงเริ่มเปรียบเทียบ d_1 กับ d_2 ถ้า $d_1 < d_2$ ก็ไปปมซ้าย (ปมหมายเลข 2) แล้วก็เปรียบเทียบ d_2 กับ d_3 ต่อ ถ้า $d_2 > d_3$ ก็ไปทางปมขวา (ปมหมายเลข 3) สรุปได้ว่า d_2 มีค่ามากสุด จากนั้นหาค่ามากสุดในสองตัวที่เหลือ คือเปรียบเทียบ d_1 กับ d_3 ถ้า $d_1 < d_3$ ได้ลำดับที่เรียงคือ d_1, d_3, d_2 แต่ถ้า $d_1 > d_3$ ได้ลำดับที่เรียงคือ d_3, d_1, d_2 (ซึ่งคือปมลูกทั้งสองของปมหมายเลข 3)

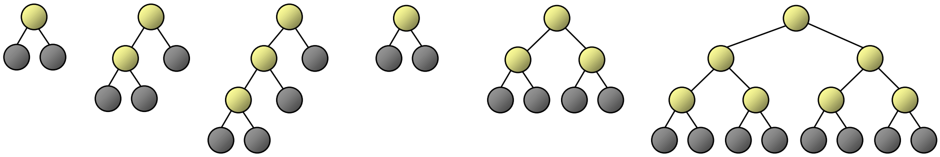


รูปที่ 13-28 ต้นไม้ตัดสินใจที่แทนการเรียงลำดับข้อมูล 3 ตัวแบบเลือก

ถ้าสังเกตที่ปมหมายเลข 4 จะพบว่า ตามวิธีของการเรียงลำดับแบบเลือก เกิดการเปรียบเทียบ d_1 กับ d_2 ซึ่งความจริงไม่ต้องเปรียบเทียบก็ได้ เพราะเคยทำไปแล้วตอนอยู่ปมหมายเลข 1 แต่ช่วยไม่ได้ เพราะเมื่ออยู่ที่ปมหมายเลข 2 แล้วรู้ว่า $d_2 < d_3$ สรุปได้แค่ว่า d_3 มีค่ามากสุดต้องอยู่ตำแหน่งท้าย เหลือภาระการหาตัวมากสุดของ d_1 กับ d_2 จึงเป็นที่มาของการเปรียบเทียบซ้ำซ้อน (ตามขั้นตอนการเรียงลำดับแบบเลือก) สำหรับปมอื่น ๆ ที่เหลือขอให้ผู้อ่านลองไล่ดู

ขอเน้นว่า เราไม่ได้สร้างต้นไม้ตัดสินใจขึ้นมาในหน่วยความจำ มันเป็นเพียงแค่แบบจำลองเพื่อให้เราเรียบเรียงความคิดและขั้นตอนการทำงานของการเปรียบเทียบข้อมูล เพื่อนำไปสู่ผลสรุปว่าข้อมูลทั้งหลายควรเรียงอย่างไร ไม่ว่าเราจะมีขั้นตอนการเรียงลำดับอย่างไร ไม่ว่าจะเป็นแบบฟอง แบบเลือก แบบเร็ว หรือแบบอื่น ๆ ที่เราได้นำเสนอมา หรือแบบอื่น ๆ ที่จะมีผู้ออกแบบนำเสนอในอนาคต ตราบเท่าที่ใช้การเปรียบเทียบข้อมูลทีละคู่เป็นเครื่องมือหลักในการช่วยตัดสินใจเพื่อเรียงลำดับข้อมูลย่อมสามารถวาดต้นไม้ตัดสินใจจำลองการทำงานได้ทั้งสิ้น ต้นไม้ตัดสินใจนี้เป็นต้นไม้แบบทวิภาค ทุกปมมีสองลูก ยกเว้นใบ ใบนั้นแทนสถานะที่สรุปได้แล้วว่า ลำดับของข้อมูลที่เรียงแล้วควรเป็นเช่นใด ถ้าเราวาดต้นไม้ตัดสินใจสำหรับการเรียงลำดับข้อมูล n ตัว แน่นอนว่า ต้นไม้นี้ต้องมีใบอย่างน้อย $n!$ ใบ เพราะลำดับของข้อมูลจำนวนต่างกัน n ตัวที่เป็นไปได้มี $n!$ แบบ ต้นไม้จะมีรูปร่างอย่างไร

ขึ้นกับขั้นตอนการทำงานของวิธีเรียงลำดับ แต่เรารู้อยู่อย่างหนึ่งว่า ความสูงของต้นไม้ตัดสินใจแทนจำนวนครั้งมากสุดของการเปรียบเทียบที่ต้องทำก่อนจะเรียงลำดับเสร็จ คำถามก็คือต้นไม้แบบทวิภาคที่มี m ใบสูงเท่าใด พิจารณารูปที่ 13-29 ก็คงสรุปได้ว่าสูง h โดยที่ $\lceil \log_2 m \rceil \le h \le m - 1$ เนื่องจากต้นไม้ตัดสินใจที่แทนการเรียงลำดับข้อมูล n ตัวมีใบอย่างน้อย $n!$ ใบ จึงสูงอย่างน้อย $\lceil \log_2 n! \rceil$ ตีความได้ว่า วิธีการเรียงลำดับข้อมูลจำนวน n ตัวแบบใด ๆ ก็ตามที่ใช้การเปรียบเทียบข้อมูลทีละคู่ต้องเปรียบเทียบอย่างน้อย $\lceil \log_2 n! \rceil$ ครั้ง



รูปที่ 13-29 ต้นไม้ทวิภาคที่มี m ใบมีความสูง h โดยที่ $\lceil \log_2 m \rceil \le h \le m-1$

พจน์ $\lceil \log_2 n! \rceil$ อาจดูไม่ค่อยคุ้นนัก ในบทที่ 3 เราได้แสดงให้เห็นจริงในตัวอย่างที่ 3-7 แล้วว่า $\log_2 n! = \Theta(n \log n)$ ดังนั้นสรุปได้ว่า "วิธีการเรียงลำดับข้อมูลจำนวน n ตัวแบบใด ๆ ก็ตามที่ใช้การเปรียบเทียบข้อมูลทีละคู่ ต้องเปรียบเทียบเป็นจำนวน $\Omega(n \log n)$" นั่นแสดงให้เรารู้ว่า การเรียงลำดับแบบผสานและแบบฮีปซึ่งมีประสิทธิภาพการทำงานเป็น $O(n \log n)$ จึงเป็นวิธีที่มีอัตราการเพิ่มของเวลาการทำงานเมื่อจำนวนข้อมูลเพิ่มขึ้นที่ดีสุด แต่ก็อย่าลืมว่า สัญกรณ์ O นั้นสะท้อนกรณีที่ข้อมูลมีจำนวนมาก และใช้แสดงเฉพาะอัตราการเติบโตของฟังก์ชัน จึงไม่ได้หมายความว่า แบบผสานและแบบฮีปจะเร็วสุดในทางปฏิบัติ

แบบฝึกหัด

1. จงเขียนเมท็อดเพื่อการเรียงลำดับแบบเลือก แบบฟอง แบบแทรก แบบเชลล์ แบบผสาน แบบฮีป และแบบเร็ว ด้วยตนเอง โดยไม่ดูรายละเอียดในหนังสือ

2. เมท็อดการเรียงลำดับแบบใด ที่ได้นำเสนอมา เป็นการเรียงลำดับแบบเสถียร จะปรับปรุงวิธีที่ไม่เสถียรให้เป็นแบบเสถียรได้หรือไม่ อย่างไร

3. ในกรณีที่ข้อมูลมีค่าเท่ากันหมด ประสิทธิภาพของวิธีการเรียงลำดับข้อมูลที่ได้นำเสนอมาจะมีประสิทธิภาพอย่างไร

4. กำหนดให้ `d[i] > d[i+k]` การสลับข้อมูลคู่นี้จะทำให้จำนวนกลับลำดับลดลงอย่างน้อยเท่าใด อย่างมากเท่าใด

5. จำนวนกลับลำดับของแถวลำดับที่เรียงลำดับแล้วมีค่าเป็น 0, ถ้าเป็นของแถวลำดับที่เรียงกลับลำดับจะมีค่าเป็น $n(n-1)/2$ จงพิสูจน์ว่า โดยเฉลี่ยแล้วจำนวนกลับลำดับของข้อมูลที่มีค่าต่างกัน n ตัวมีค่าเป็น $n(n-1)/4$

6. จงทดลองเขียนโปรแกรมเพื่อเปรียบเทียบเวลาการทำงาน และสรุปผลที่ได้ด้วย

   6.1. การเรียงลำดับข้อมูลสุ่มโดยใช้แบบฟอง (รหัสที่ 13-4) แบบแทรก (รหัสที่ 13-5) และแบบแทรกที่เปลี่ยนมาใช้การสลับข้อมูลแทน

   6.2. การเรียงลำดับข้อมูลกลับลำดับและข้อมูลสุ่มโดยใช้แบบฟอง (รหัสที่ 13-4) และแบบฟองที่ไม่ต้องตรวจสอบให้หยุดเมื่อไม่มีการสลับข้อมูล (รหัสที่ 13-3)

   6.3. การเรียงลำดับข้อมูลสุ่มแบบเร็วโดยเลือกตัวซ้ายเป็นตัวหลัก (รหัสที่ 13-13), สุ่มเลือกตัวหลัก (รหัสที่ 13-14) และใช้มัธยฐานของข้อมูลสามตัวเป็นตัวหลัก (รหัสที่ 13-15)

   6.4. การเรียงลำดับข้อมูลจำนวนเต็มแบบ `Integer` และแบบ `int` (สำหรับกรณี `int` ต้องเขียนเมท็อดการเรียงลำดับทั้งหมดให้รับข้อมูลแบบ `int[]` และการเปรียบเทียบใช้ < เลย)

   6.5. การเรียงลำดับข้อมูลแบบเชลล์ (รหัสที่ 13-7) กับการเรียงลำดับแบบฮีป (รหัสที่ 13-11) เพื่อหาว่า เมื่อข้อมูลมีปริมาณเท่าใดแบบฮีปจึงเร็วกว่าแบบเชลล์

7. รหัสข้างล่างนี้ปรับปรุงจากเมท็อด `shellSort` จากเดิมมีวงวนสี่วงซ้อนกันในรหัสที่ 13-7 เหลือเพียงวงวนสามวง อยากทราบว่า รหัสข้างล่างนี้ทำงานถูกต้องหรือไม่ อย่างไร

```
public static void shellSort(Object[] d) {
  int h;
  for (h = 1; h <= d.length/4; h = 2*h + 1);
  for (; h > 0; h /= 2) {
    for (int i = h; i < d.length; i++) {
      Object t = d[i];
      int j = i - h;
      while (j >= 0 && lessThan(t, d[j])) {
        d[j + h] = d[j];
        j -= h;
      }
      d[j + h] = t;
    }
  }
}
```

8. Gonnet เสนอลำดับ h ของการเรียงลำดับแบบเชลล์ อีกแบบที่คำนวณได้ง่ายเริ่มด้วย $n/2$ และลดค่าลงด้วยการหารด้วยค่า 2.2 ในแต่ละรอบ เขียนเป็นวงวนเปลี่ยนค่า h ได้ดังแสดงข้างล่างนี้ จงทำการทดลองเปรียบเทียบเวลาการทำงานระหว่างการใช้ลำดับของ Gonnet กับของ Sedgewick

```
public static void shellSort(Object[] d) {
  for (int h=d.length/2; h>0; h = h==2 ? 1 : (int)(h/2.2)) {
    ...
  }
}
```

9. จงพิสูจน์ว่า การผสานข้อมูลสองชุด (มีจำนวนรวม n ตัว) ด้วยเมท็อด `merge` (รหัสที่ 13-9) จะเปรียบเทียบข้อมูลไม่เกิน $n - 1$ ครั้ง

10. จงพิสูจน์ว่า การเรียงลำดับแบบผสาน (รหัสที่ 13-8) จะเกิดการเรียกเมท็อด `mergeSortR` แบบเวียนเกิดซ้อน ๆ กันไม่เกิน $n \log_2 n$ ครั้ง

11. จงเขียนเมท็อด `int numberOfInversions(Object[] d)` เพื่อคำนวนจำนวนกลับลำดับของข้อมูลในแถวลำดับ `d` โดยใช้เวลาทำงานเป็น $O(n \log n)$

12. การเลือกตัวตรงกลาง `d[(left+right)/2]` มาเป็นตัวหลักในการแบ่งส่วนมีผลดีผลเสียอย่างไร

13. คำถามนี้เกี่ยวกับการเขียนวิธีการเรียงลำดับแบบเร็วที่ไม่เกิด `StackOverflowError`

 13.1. เราสามารถเขียนเมท็อด `quickSortR` ในรหัสที่ 13-12 ที่มีการเรียกแบบเวียนเกิดสองครั้ง ให้เหลือเพียงครั้งเดียวได้ง่าย ๆ ดังแสดงข้างล่างนี้ จงอธิบายการทำงาน

```
static void quickSortR(Object[] d, int left, int right) {
  while (left < right) {
    int i = partition(d, left, right);
    quickSortR(d, left, i-1);
    left = i+1;
  }
}
```

 13.2. จากข้อที่แล้ว เราสามารถเขียนได้อีกแบบ ดังแสดงข้างล่างนี้ สองแบบนี้ต่างกันแค่จะนำการเรียกแบบเวียนเกิดใดมาเปลี่ยนเป็นวงวนแทน ซึ่งทำได้ทั้งสองแบบ ถ้าเราคิดสักเล็กน้อยจะพบว่า บางครั้งอาจไปเรียกเวียนเกิดเพื่อเรียงลำดับชุดซ้าย บางครั้งอาจไปเรียกเวียนเกิดเพื่อเรียงลำดับชุดขวา ถ้าเลือกให้ดีจะทำให้เกิดการเรียกแบบเวียนเกิดซ้อน ๆ กันน้อย จงปรับปรุงให้เกิดการเรียกแบบเวียนเกิดซ้อน ๆ กัน (นั่นคือใช้เนื้อที่ของกองซ้อนระบบในการเรียกเวียนเกิด) ไม่เกิน $\log_2 n$ ครั้ง (ทำให้เราไม่ต้องกังวล `StackOverflowError`)

```
static void quickSortR(Object[] d, int left, int right) {
  while (left < right) {
    int i = partition(d, left, right);
    quickSortR(d, i+1, right);
    right = i-1;
  }
}
```

14. ในกรณีที่เรารู้ว่า ข้อมูลจะนำมาเรียงลำดับนั้นมีข้อมูลซ้ำกันมาก เราควรแบ่งส่วนในแบบที่เรียกว่า การแบ่งส่วนสามทาง (three-way partitioning) คือแบ่งให้ชุดซ้ายน้อยกว่าตัวหลัก ชุดกลางเท่ากับตัวหลัก และชุดขวามากกว่าตัวหลัก ทำให้การเรียงชุดซ้ายแลชุดขวาจะมีขนาดเล็กลง จงเขียนเมท็อดที่แบ่งส่วนข้อมูลแบบสามทาง

15. การเรียงลำดับข้อมูลแบบต้นไม้ (tree sort) ที่ได้นำเสนอในบทที่ 10 ซึ่งอาศัยการนำข้อมูลในแถวลำดับไปสร้างต้นไม้ค้นหาแบบทวิภาค จากนั้นแวะผ่านต้นไม้แบบตามลำดับเพื่อนำข้อมูลใส่กลับคืนในแถวลำดับ ก็จะได้ข้อมูลที่เรียงลำดับ อยากทราบว่า การเรียงลำดับแบบนี้ เกิดการเปรียบเทียบข้อมูลที่คล้ายกับการเรียงลำดับแบบใดที่ได้นำเสนอในบทนี้

16. ถ้าเรารู้ก่อนล่วงหน้าว่า ข้อมูลที่จะนำมาเรียงลำดับเป็นจำนวนเต็มในช่วง [1, 5000] จงนำเสนอการเรียงลำดับข้อมูลชุดนี้ในเวลา O(n)

17. การเรียงลำดับข้อมูลแบบใดที่ได้นำเสนอมา สามารถปรับปรุงให้ใช้กับชุดข้อมูลที่เก็บด้วยรายการโยง ได้อย่างมีประสิทธิภาพ

บรรณานุกรม

หนังสือและแหล่งความรู้ทางโครงสร้างข้อมูลมีมากมาย ผู้เขียนขอแสดงรายชื่อหนังสือและเว็บไซต์ที่ผู้เขียนได้เคยอ่าน และนำความรู้จากแหล่งเหล่านี้กลั่นกรองจนได้หนังสือเล่มนี้

[1] Alfred V. Aho, J. D. Ullman, J. E. Hopcroft, “Data Structures and Algorithms”, Pearson Education (1983)

[2] Duane Bailey , “Java Structures: Data Structures in Java for the Principled Programmer”, McGraw-Hill (July 2002)

[3] Joshua Bloch , “Effective Java Programming Language Guide”, Addison-Wesley Professional (June 5, 2001)

[4] Timothy Budd , “Classic Data Structures in Java”, Pearson Education (October 2000)

[5] William Collins, “Data Structures and the Java Collections Framework”, McGraw-Hill Science/Engineering/Math (August 2001)

[6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, “Introduction to Algorithms”, The MIT Press; 2 edition (September 1, 2001)

[7] Elisabeth Freeman, Eric Freeman, Bert Bates, Kathy Sierra, “Head First Design Patterns” , O'Reilly Media, Inc. (October 25, 2004)

[8] Gaston H. Gonnet, R. Baeza-Yates (Editor), “Handbook of Algorithms and Data Structures in Pascal and C”, Addison-Wesley Pub (Sd) (May 1991)

[9] Michael T. Goodrich, Roberto Tamassia, “Data Structures and Algorithms in Java”, John Wiley & Sons; 4 edition (August 24, 2005)

[10] John R. Hubbard, J. R. Hubbard , “Schaum's Outline of Data Structures with Java”, McGraw-Hill (November 2000)

[11] Robert L. Kruse, “Data Structures and Program Design”, Prentice Hall; (January 3, 1994)

[12] Dinesh P. Mehta (Editor), Sartaj Sahni (Editor), “Handbook of Data Structures and Applications”, CRC Press (October 2004)

[13] Ron Penton, “Data Structures for Game Programmers”, Muska & Lipman/Premier-Trade (November 25, 2002)

[14] Sartaj Sahni, “Data Structures, Algorithms, and Applications in Java”, McGraw-Hill (April 2000)

[15] Robert Sedgewick, “Algorithms in Java, Parts 1-4: Fundamentals, Data Structure, Sorting, Searching”, Addison-Wesley Professional (July 13, 1998)

[16] Robert Sedgewick, Philippe Flajolet, “An Introduction to the Analysis of Algorithms”, Addison-Wesley Professional (November 30, 1995)

[17] Mark Allen Weiss, “Data Structures and Algorithm Analysis in Java”, Addison Wesley (October 1998)

[18] Mark Allen Weiss, “Data Structures and Problem Solving Using Java”, Addison Wesley; 3 edition (February 14, 2005)

[19] สมชาย ประสิทธิ์จูตระกูล, “การออกแบบและวิเคราะห์อัลกอริทึม”, สวทช (2544)

[20] http://en.wikipedia.org/wiki/Main_Page

[21] http://www.nist.gov/dads/ (Dictionary of Algorithms and Data Structures)

[22] http://www.cs.ubc.ca/spider/harrison/Java/sorting-demo.html. (sorting animation)

[23] http://java.net (Java)

[24] http://java.sun.com/j2se/1.5.0/docs/guide/collections/ (Collections Framework)

[25] http://java.sun.com/docs/books/tutorial/collections/ (Tutorial : Collections Framework)

ดัชนี

ก

ข

ค

จ

ฐ, ด

ต

ถ, ท, น

ป

ผ, พ, ฟ

ม

ร

ล, ว, ส

อ, ฮ

1

A

B

C

D

E

F, G

H

I

J, K

L

M

N, O

P

Q

R

S

T

U, V, W