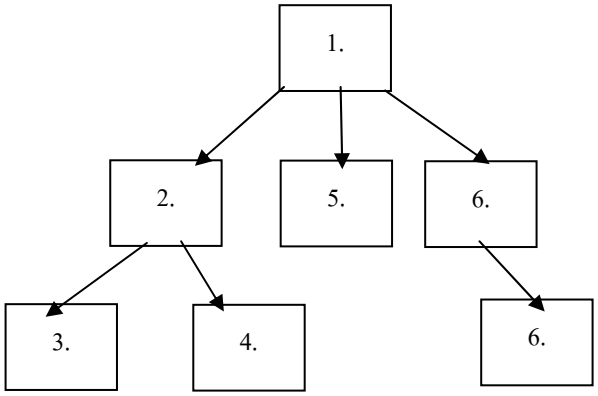
บทที่ 6
โครงสร้างต้นไม้

ในบทนี้เราจะมาดูโครงสร้างต้นไม้ (ต่อไปนี้จะเรียกว่า tree ด้วย)

ลักษณะทั่วไปของโครงสร้างต้นไม้

ตัว tree นั้นจะมีลักษณะดังรูป 6.1



รูป 6.1 ลักษณะทั่วไปของ tree

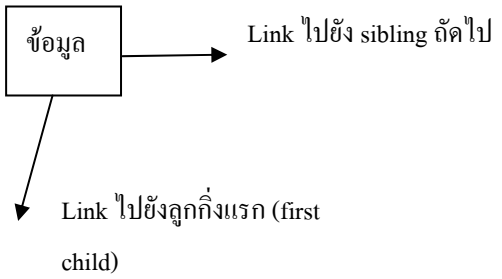
ซึ่งดูเป็นต้นไม้ที่กลับหัว ซึ่งมี โนด และกิ่งต่อกันไปเรื่อยๆ โดย โนด หนึ่งๆมีกี่กิ่งก็ได้ ส่วนประกอบโดยทั่วไปของ tree จะเป็นดังรูป 6.1 คือ

- ตัว โนด บนสุดจะเรียกว่าเป็น **ราก (root)** ของ tree
- tree ส่วนย่อย (มี โนด 2 เป็น root) เรียกได้ว่าเป็น **subtree** ของ treeใหญ่
- โนด 1 จะเป็น **บัพแม่ (parent)** ของ โนด 2,5,6 ส่วน โนด 2 เป็น parent ของ โนด 3,4 หรือเรียกอีกอย่างว่า โนด 3,4 เป็น **บัพลูก (child)** ของ โนด 2

- โนด 3 เป็น**บัพพี่น้อง (sibling)** กับ โนด 4
- ถ้ามีความสัมพันธ์แบบ parent ตั้งแต่หนึ่งทอดขึ้นไป เราเรียกว่า **บัพบน (ancestor)** เช่น โนด 1 เป็น ancestor ของ โนด 7
- โนด ที่ไม่มีกิ่งออกไป เราเรียกว่า **บัพใบ หรือ ใบ (leaf)** เช่นในรูป 6.1 leaf คือ โนด 3, 4, 5 และ 7
- เราจะสังเกตได้ว่า มีแค่ 1 **เส้นทาง (path)** จาก root ไปยัง โนด แต่ละ โนด เท่านั้น
- **ความลึก (depth)** ของ โนด n_i คือความยาวของ path จาก root ถึง n_i ตัวอย่างเช่น depth ของ โนด เบอร์ 7 คือ 2
- **ความสูง (height)** ของ โนด n_i คือความยาวของ path ที่ยาวที่สุดจาก n_i ถึง leaf ตัวอย่างเช่น height ของ โนด เบอร์ 7 คือ 0 และ height ของ โนด เบอร์ 1 คือ 2
- height ของ tree ก็คือ height ของ root นั่นเอง ถ้า tree มีแค่ root เราให้ height เป็น 0 และถ้า tree เป็น empty tree เราให้ height เป็น -1
- height ของต้นไม้ใดๆ จะเท่ากับ height ของ subtree ที่สูงที่สุด บวกด้วย 1
- **ระดับ (level)** ของ โนด คือ ลำดับชั้นของ โนด นั้น โดย child ของ โนด หนึ่งๆจะมีระดับ1 ระดับมากกว่า parent ของมัน (root ถือว่ามีระดับเป็น 0)

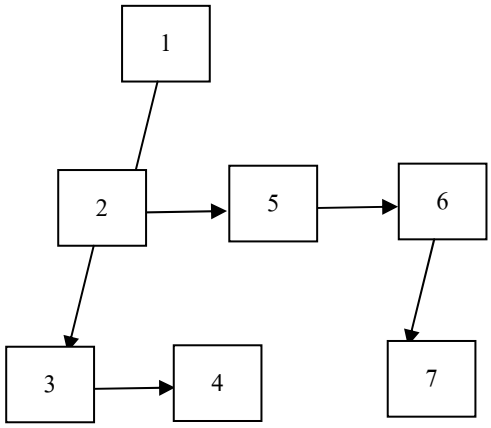
แนวคิดในการสร้างต้นไม้โดยทั่วไป

โดยทั่วไป ถ้าให้แต่ละ โนด มีกี่กิ่งก็ได้ เราไม่มีทางกำหนดจำนวนกิ่งที่แน่นอนได้ เพราะไม่รู้ว่าจะครบหรือเปล่า และการเตรียมกิ่งไว้ล่วงหน้ามากๆก็เป็นการเปลืองหน่วยความจำโดยใช่เหตุ วิธีหนึ่งที่อาจแก้ปัญหานี้ได้คือ ให้แต่ละ โนด เก็บข้อมูลดังรูป 6.2



รูป 6.2 โนด ของต้นไม้แบบทั่วไป

ถ้าใช้โครงสร้างข้อมูลแบบในรูป 6.2 เราจะสามารถเติมกิ่งใหม่เมื่อไรก็ได้ ตัวต้นไม้จากรูป 6.1 จะกลายเป็นดังรูป 6.3



รูป 6.3 ลักษณะของต้นไม้จากรูป 6.1 ที่ ทำจากแนวคิดในรูป 6.2

นี่ก็เป็นแนวคิดในการเขียนโครงสร้างต้นไม้อย่างคร่าวๆ แต่ว่าในหนังสือเล่มนี้เราจะไม่สนโครงสร้างต้นไม้แบบนี้ เราจะสนแต่โครงสร้างต้นไม้ที่มีได้อย่างมากที่สุด 2 กิ่งเท่านั้น (binary tree)

ประโยชน์ของโครงสร้างต้นไม้

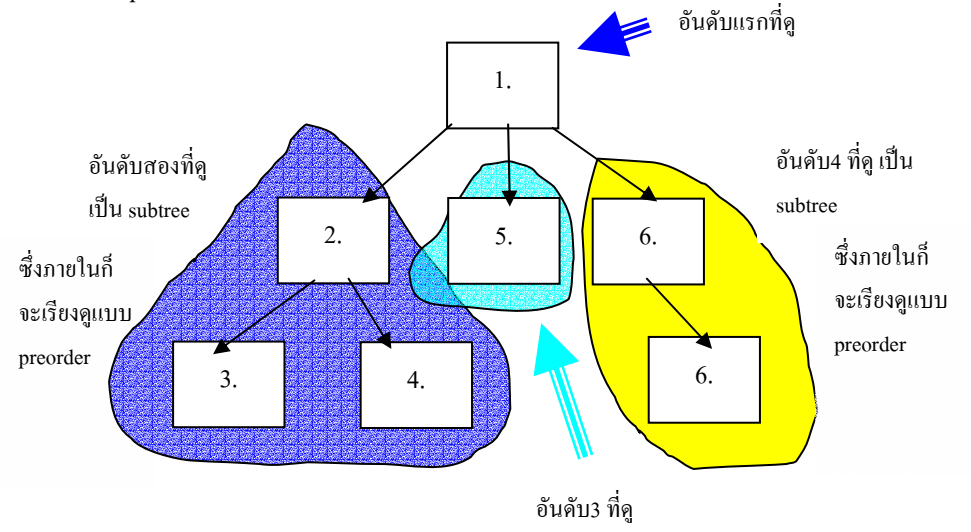
อย่างแรกที่เห็นชัดสุดๆก็คงจะเป็นโครงสร้าง directory หรือ folder ในระบบปฏิบัติการของเรานั่นแหละ ทั้งวินโดวส์ ยูนิกซ์ การใช้โครงสร้างต้นไม้มีความง่าย และยังทำให้ของที่อยู่ต่าง folder ใช้ชื่อเหมือนกันได้

รูปแบบการเรียงดูสมาชิกภายในต้นไม้

ยามที่เราเขียนโปรแกรมเพื่อตรวจสอบสมาชิกในต้นไม้จำต้องมีวิธีการที่แน่นอนเพื่อไม่ให้มีโนดไหนหลุดรอดจากการสำรวจไป ในปัจจุบันมี 4 วิธีหลักๆที่นิยมใช้ในการเรียงดูสมาชิกภายในต้นไม้ ดังนี้

ก่อนลำดับ (preorder)

คือการดูของจาก root โนด ที่เราเริ่มพิจารณา แล้วจึงไปดูของใน subtree โดยใน subtree ก็เริ่มดูจาก root โนด เช่นเดียวกัน ทำอย่างนี้ซ้ำไปเรื่อยๆ รูป 6.4 แสดงการเรียงสมาชิกในต้นไม้ของรูป 6.1 แบบ preorder



รูป 6.4 การเรียงดูสมาชิกของต้นไม้แบบ preorder

ผลการเรียงสมาชิกใน โนด จากรูป 6.4 คือ 1,2,3,4,5,6,7

หลังลำดับ (postorder)

คือการเรียงดู subtree ทั้งหมด แล้วจึงมาดูของใน root ซึ่งภายใน subtree นั้นก็ต้องมีการเรียงดูแบบเดียวกัน เพราะฉะนั้นถ้าเราเรียงสมาชิกใน tree ในรูป 6.4 แบบ postorder จะได้ผลลัพธ์เป็น 3,4,2,5,7,6,1

ตามลำดับ (inorder)

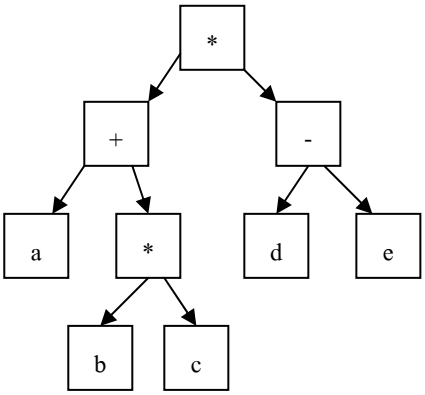
คือการเรียงสมาชิกโดยสมาชิกของ subtree ด้านซ้ายของ root จะถือว่าต้องมาก่อน root ส่วนสมาชิกของ subtree ที่อยู่ด้านขวาของ root จะถือว่ามาทีหลัง root นอกจากนี้ส่วนข้างในตัว subtree ก็ใช้วิธีการเรียงอย่างเดียวกัน ดังนั้นถ้าดูจากรูป 6.4 (ไม่นับ subtree ที่มี 5 เพราะไม่รู้จะจัดอยู่ข้างไหนของ root) เราจะเรียงสมาชิกได้เป็น 3,2,4,1,6,7

การค้นหาในแนวกว้าง (breadth-first search)

วิธีการสามแบบแรกที่กล่าวมาข้างต้นนั้น เป็นวิธีการค้นหาในแนวลึก (depth-first search) ซึ่งถ้ามีการดู subtree ใดแล้วจะต้องตรวจสอบทั้ง subtree นั้นให้เสร็จ จึงจะสำรวจ root หรือ subtree อื่นๆที่เป็น sibling ต่อไปได้ วิธีการตรวจสอบแบบ breadth-first จะตรวจสอบทีละระดับของ tree โดยเริ่มจากระดับของ root ดังนั้น tree ในรูป 6.4 จะมีการเรียงสมาชิกได้เป็น 1,2,5,6,3,4,7

แนวคิดในการเขียนโปรแกรมเพื่อทำการค้นหาแบบนี้ก็จะต่างจากสามแบบแรกอย่างสิ้นเชิง สำหรับสามแบบแรกนั้น ถ้าเรารู้ว่าจะทำต้นไม้ของเราแบบใดเราก็สามารถประยุกต์ใช้ recursion ได้อย่างไม่ยากนัก แต่การค้นต้นไม้เป็นระดับๆนั้นยากที่จะใช้ recursion ดังนั้นจึงมีการนำแถวคอย (คิว) เข้ามาช่วย

โดยตอนที่ดูโนดโนดหนึ่ง เราจะบันทึก child ของโนดนั้นลงในคิวที่สร้างไว้สำหรับเก็บโนดในระดับถัดลงมา (next level queue) เช่น ถ้าเรามีต้นไม้ expression tree ของ (a+(b*c))*(d-e) ดังรูป 6.5



รูป 6.5 expression tree

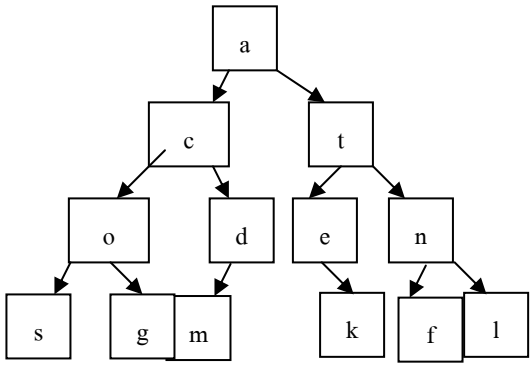
ขั้นตอนการทำการค้นหาในแนวกว้างจะเป็นดังนี้

1. เมื่อเราดู root เราจะบันทึก child โนด ของ root ไว้บน next level queue ดังนั้น ในขั้นนี้ next level queue มี + กับ –
2. update ให้ this level queue = next level queue แล้ว เคลียร์ next level queue
3. สำรวจ this level queue จะพบ + เป็นตัวแรก ตอนนี้ให้ดูตัวต้นไม้แล้วใส่ลูกของ + ลงใน next level queue นั่นคือ ในตอนนี้ next level queue = a,*
4. สำรวจ this level queue ต่อ จะพบ – เป็นสมาชิกตัวต่อมา ใส่ลูกของ - ลงใน next level queue นั่นคือ ในตอนนี้ next level queue = a,*,d,e
5. ไม่มีสมาชิกใน this level queue แล้ว ดังนั้น update ให้ this level queue = next level queue นั่นคือ this level queue = a,*,d,e
6. เริ่มเยี่ยม this level queue ตั้งแต่ต้นอีกครั้ง และทำซ้ำไปเรื่อยๆ

จะเห็นได้ว่าเราได้สำรวจสมาชิกของต้นไม้นี้เรียงตามระดับจริงๆ

แบบฝึกหัด

1. จงเรียงสมาชิกในต้นไม้ต่อไปนี้แบบ preorder inorder และ postorder



2. จงเรียงสมาชิกของต้นไม้จากรูป 6.5 แบบ preorder inorder และ postorder

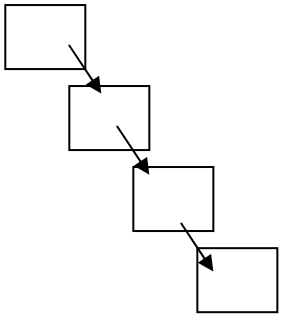
3. ถ้ามี postfix notation abc*+de-*

จงเสนอวิธีการทำ postfix notation นี้ ให้เป็น expression tree

4. จงสร้าง Java class ที่ เป็นต้นไม้ในแบบของรูป 6.3 ขึ้นมา

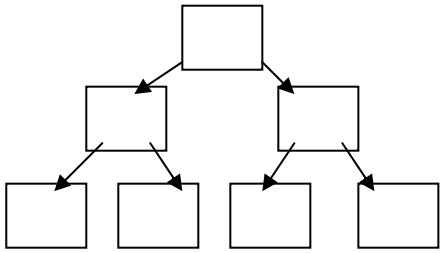
ต้นไม้แบบทวิภาค (Binary Tree)

เป็นต้นไม้ที่แต่ละ โนด มีกิ่งออกไปได้มากที่สุดสองกิ่ง ดังนั้นอาจมีกิ่งเดียวในทุกระดับก็ได้ ซึ่งก็จะกลายเป็น linked list ไป หรืออาจเรียกอีกอย่างได้ว่า skewed tree หรือ chain ดังรูป 6.6



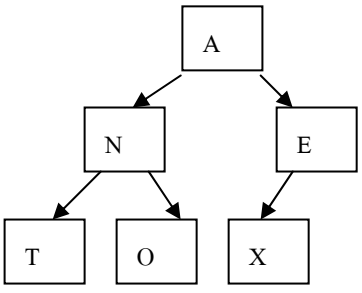
รูป 6.6 Skewed tree

ส่วน binary tree ที่เป็นรูปสามเหลี่ยมอย่างสมบูรณ์ ไม่ขาด leaf ใดๆในระดับของ leaf ที่สูงที่สุด เราเรียกว่าเป็นต้นไม้แบบเต็มต้น (full binary tree หรือ perfectly balanced tree) ดังรูป 6.7



รูป 6.7 ต้นไม้แบบเต็มต้น

ส่วน binary tree ที่เต็มถึงระดับก่อน leaf ที่สูงที่สุดแต่ระดับของ leaf ที่สูงที่สุดอาจไม่เต็ม (แต่ยังมี leaf เติมใส่ต้นจากซ้ายไปขวา) เรียกว่า ต้นไม้บริบูรณ์ (complete tree) ดังรูป 6.8



รูป 6.8 ต้นไม้บริบูรณ์

ต้นไม้แบบเต็มต้นถือว่าเป็นต้นไม้บริบูรณ์แน่นอน แต่ต้นไม้บริบูรณ์ไม่จำเป็นต้องเต็มต้น มีข้อสังเกตอีกอย่างคือ ถ้าเราเอาสมาชิกของต้นไม้บริบูรณ์ใส่ในอาร์เรย์ โดยเรียงใส่แบบตามแนวกว้าง (breadth-first) สมาชิกที่ตำแหน่ง i จะมีลูกซ้ายอยู่ที่ตำแหน่ง 2i+1 และลูกขวาอยู่ที่ตำแหน่ง 2i+2 ส่วนตัว parent ของ i จะอยู่ที่ตำแหน่ง (i-1)/2 (ปัดทศนิยมทิ้งไป) ตัวอย่างเช่นถ้านำต้นไม้ในรูป 6.8 มาใส่ในอาร์เรย์ เราจะได้

A	N	E	T	O	X

จากอาร์เรย์นี้ index ของ E คือ 2 จากกฎข้างต้น ลูกซ้ายของ E จะต้องมี index =2*2+1 =5 ซึ่งก็คือ X นั่นเอง เป็นไปตามที่มีในต้นไม้จริงๆ ส่วน parent ของ E ก็จะมี index = (2-1)/2 = 0 เมื่อปัดลงแล้ว ซึ่งก็เป็นจริงตามที่มีในต้นไม้

นอกจากนี้ยังมี binary tree อีกแบบที่จะต้องเป็น empty tree หรือไม่ก็มี non-leaf node ที่ต้องมีสองกิ่งเท่านั้น binary tree แบบนี้เราเรียกว่า two-tree จากรูป 6.8 ถ้าเราตัด โนด ที่มี X ทิ้งไป ต้นไม้ในรูปก็จะเป็น two-tree และจากนิยามของ two-tree ต้นไม้แบบเต็มต้นจะต้องเป็น two-tree แน่ๆ (แต่ two-tree ไม่จำเป็นต้องเต็มต้น)

ทฤษฎีต่างๆของต้นไม้ทวิภาค

ถ้าเราให้

- leaves(t) แทนฟังก์ชั่นที่บอกจำนวนใบของต้นไม้ t
- n(t) เป็นจำนวน โนด ของ ต้นไม้ t
- height(t) เป็นความสูงของต้นไม้ t
- leftsubtree(t) เป็นต้นไม้ย่อยด้านซ้ายของ t
- rightsubtree(t) เป็นต้นไม้ย่อยด้านขวาของ t
- max(a,b) เป็นค่าที่มากที่สุดเมื่อเทียบ a กับ b

สำหรับต้นไม้ทวิภาคที่ไม่ใช่ต้นไม้ว่าง จะมีความสัมพันธ์ระหว่างฟังก์ชั่นต่างๆดังนี้

นิยามที่ 6-1

$$leaves(t) \le \frac{n(t)+1}{2.0}$$

นิยามที่ 6-2

$$\frac{n(t)+1}{2.0} \le 2^{height(t)}$$

นิยามที่ 6-3

ถ้า t เป็น two-tree แล้วละก็ $leaves\ (t) = \frac{n(t)+1}{2.0}$

นิยามที่ 6-4

ถ้า $leaves\ (t) = \frac{n(t)+1}{2.0}$ แล้วละก็ t เป็น two-tree แน่นอน

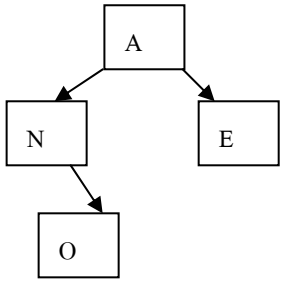
นิยามที่ 6-5

ถ้า t เป็นต้นไม้แบบเต็มต้น แล้วละก็ $\frac{n(t)+1}{2.0} = 2^{height\ (t)}$

นิยามที่ 6-6

ถ้า $\frac{n(t)+1}{2.0} = 2^{height\ (t)}$ แล้วละก็ t เป็นต้นไม้แบบเต็มต้น

อย่าลืมว่าเราใช้ 2.0 เพราะฉะนั้นค่าจากสูตรเหล่านี้ผมจะถือว่าเป็นทศนิยมนะ เพราะจำนวนเต็มจะใช้กับกฎข้อ 4 ไม่ได้ ลองสังเกตดูจากรูป 6.9 ถ้าเราใช้จำนวนเต็ม leaves(t) ก็จะมีค่า (n(t)+1)/2 แต่เราเห็นได้ชัดว่าต้นไม้ในรูปนี้ไม่ใช่ two-tree



รูป 6.9 ต้นไม้ที่ผิดกฎข้อ 4 เพราะเราใช้จำนวนเต็ม

กฎทุกข้อสามารถพิสูจน์ได้ด้วยวิธีอุปนัยทางคณิตศาสตร์ (math induction) โดยใช้ความสูงของต้นไม้เป็นหลัก

พิสูจน์นิยาม 6-1

$$leaves(t) \le \frac{n(t)+1}{2.0}$$

base case คือเมื่อ t เป็น ต้นไม้ที่มีแต่ราก

$$leaves\ (t) = 1$$

$$\frac{n(t)+1}{2.0} = 1$$ ซึ่งเห็นชัดว่าทำให้สมการเป็นจริง

inductive case คือ เมื่อ t เป็นต้นไม้ที่สูง h

ให้ $leaves(t) \le \frac{n(t)+1}{2.0}$

สิ่งที่ต้องพิสูจน์ คือ เมื่อ t เป็นต้นไม้ที่สูง h+1 แล้ว $leaves(t) \le \frac{n(t)+1}{2.0}$

ในกรณีนี้ เราได้

$$leaves(t) = leaves(leftsubtree(t)) + leaves(rightsubtree(t))$$

$$\le \frac{n(leftsubtree(t))+1}{2.0} + \frac{n(rightsubtree(t))+1}{2.0}$$

โดยแทนค่าจาก inductive case ลงไปได้ เพราะ leftsubtree กับ rightsubtree ของ t เป็นต้นไม้ที่สูง h

เรารู้ว่า $n(t) = n(\text{leftsubtree}(t)) + n(\text{rightsubtree}(t)) + 1$

เพราะฉะนั้นเราสามารถแทนค่า n(t) ลงไปในอสมการได้ ผลคือได้

$$leaves(t) \le \frac{n(t)+1}{2.0}$$

ซึ่งเป็นสิ่งที่เราต้องการพิสูจน์จริงๆ

พิสูจน์นิยาม 6-2

$$\frac{n(t)+1}{2.0} \le 2^{\text{height(t)}}$$

base case คือเมื่อ t เป็น ต้นไม้ที่มีแต่ราก

$$\frac{n(t)+1}{2.0} = 1$$

$2^{\text{height(t)}} = 2^0 = 1$ ซึ่งเห็นชัดว่าทำให้สมการเป็นจริง

inductive case คือ เมื่อ t เป็นต้นไม้ที่สูง h

ให้ $\frac{n(t)+1}{2.0} \le 2^{\text{height(t)}}$

สิ่งที่ต้องพิสูจน์ คือ เมื่อ t เป็นต้นไม้ที่สูง h+1 แล้ว $\frac{n(t)+1}{2.0} \le 2^{\text{height(t)}}$

ในกรณีนี้เราได้ค่า

$$\frac{n(t)+1}{2.0} = \frac{n(\text{leftsubtree}(t)) + n(\text{rightsubtree}(t)) + 1 + 1}{2.0}$$

$$= \frac{n(\text{leftsubtree}(t))+1}{2.0} + \frac{n(\text{rightsubtree}(t))+1}{2.0}$$

$$\le 2^{height(leftsubtree(t))} + 2^{height(rightsubtree(t))}$$

$$\le 2 * 2^{\max(height(leftsubtree(t)),height(rightsubtree(t)))}$$

$$\le 2^{\max(height(leftsubtree(t)),height(rightsubtree(t)))+1}$$

$$\le 2^{height(t)}$$

ซึ่งได้ตามที่เราต้องการพิสูจน์

พิสูจน์นิยาม 6-3

ถ้า t เป็น two-tree แล้วละก็ $leaves(t) = \frac{n(t)+1}{2.0}$

base case คือเมื่อ t เป็น ต้นไม้ที่มีแต่ราก

$leaves\ (t\ ) = 1$

$\frac{n(t)\ +1}{2.0} = 1$ ซึ่งเห็นชัดว่าทำให้สมการเป็นจริง

inductive case คือ เมื่อ t เป็น two-tree ที่สูง h

ให้ $leaves(t) = \frac{n(t)+1}{2.0}$

สิ่งที่ต้องพิสูจน์ คือ เมื่อ t เป็น two-tree ที่สูง h+1

ในกรณีนี้เมื่อเราดูที่ t

$$leaves(t) = leaves(leftsubtree(t)) + leaves(rightsubtree(t))$$

เพราะว่า ทั้ง leftsubtree(t) และ rightsubtree(t) ต่างก็เป็น two-tree ดังนั้น จาก inductive case เราจะได้ว่า

$$leaves(t) = \frac{n(leftsubtree(t))+1}{2.0} + \frac{n(rightsubtree(t))+1}{2.0}$$

เรารู้ว่า $n(t) = n(leftsubtree(t)) + n(rightsubtree(t)) + 1$

ดังนั้นแทนค่านี้ลงไปในสมการของ leaves(t) จะได้

$leaves(t) = \frac{n(t)+1}{2.0}$ ตามนิยาม

แบบฝึกหัด

1. จงพิสูจน์นิยามที่ 6-4 ถึง 6-6

จากนิยามที่ 6-5 และ 6-6 เราจะเห็นว่าการเก็บข้อมูลในต้นไม้ในรูปแบบเต็มๆนั้น

มี $\frac{n(t)+1}{2.0} = 2^{height\ (t)}$

ซึ่งเขียนใหม่ได้เป็น

$$\log_2(\frac{n(t)+1}{2.0}) = \log_2 2^{height(t)}$$
$$\log_2(n(t)+1) - 1 = height(t)$$

เพราะฉะนั้น ความสูงของต้นไม้ทวิภาคซึ่งเราพยายามเติมให้เต็มที่สุดนั้นจะเป็น log ของจำนวนโนด ซึ่งนี่ก็คือตัวกำหนดเวลาในการเอาสมาชิกเข้า/ออกจากต้นไม้

ยังมีอีกนิยามอีกชุดหนึ่งที่น่าสนใจคือ

นิยามที่ 6-7

External Path Length

ให้ t เป็นต้นไม้ทวิภาคที่มีราก จะได้ว่า External Path length, E(t) คือผลบวกของความลึกของทุกใบ

จากรูป 6.5 ค่า E(t) จะเป็น 2+3+3+2+2=12

นิยามที่ 6-8

ถ้า t เป็นต้นไม้ทวิภาคที่มีจำนวนใบเป็น k (ซึ่งมากกว่า 0) จะได้ว่า

$$E(t) \geq (k/2)\lfloor \log_2 k \rfloor$$

พิสูจน์โดย

ก่อนอื่น พิสูจน์โดยอุปนัยทางคณิตศาสตร์ว่า ที่ระดับ L มีจำนวนใบที่เป็นไปได้มากที่สุดคือ 2^L (ขอไม่แสดงส่วนนี้)

ที่ระดับต่ำกว่า L จะมีจำนวนใบ ไม่ถึง 2^L แน่นอน

ถ้าให้ L เป็น $\lfloor \log_2 k \rfloor - 1$ จะได้ว่าที่ระดับต่ำกว่า L จะมีจำนวนใบ
$\leq 2^{\lfloor \log_2 k \rfloor - 1}$
$\leq 2^{\log_2 k - 1}$
$\leq k/2$

และที่ระดับสูงกว่า L (นั่นคืออยู่ระดับ $\lfloor \log_2 k \rfloor$ ขึ้นไป) จะมีจำนวนใบอย่างน้อย k/2

เพราะว่า E(t) คือผลบวกของความลึกของทุกใบ ฉะนั้น
$E(t) \geq \lfloor \log_2 k \rfloor * \frac{k}{2}$ เป็นไปตามนิยาม

แนวคิดในการเขียนโปรแกรมสร้างต้นไม้ทวิภาค

เรามาดูกันว่าต้นไม้ชนิดนี้ควรมีอะไรเป็นส่วนประกอบ ก่อนอื่น ตัวต้นไม้ควรประกอบด้วยสามส่วนคือ

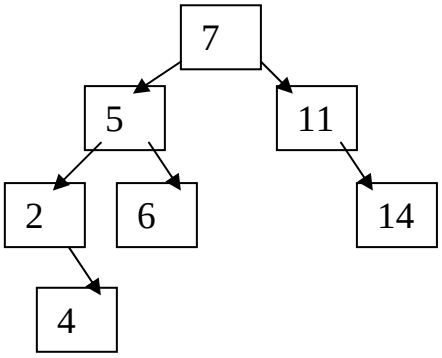
1. root
2. left subtree
3. right subtree

ต้นไม้ค้นหาแบบทวิภาค (Binary Search Tree)

Binary Search Tree t คือ binary tree ซึ่งเป็น empty tree หรือเป็นต้นไม้ซึ่ง

1. ทุกสมาชิกใน leftsubtree(t) มีค่าน้อยกว่าสมาชิกที่รากของ t
2. ทุกสมาชิกใน rightsubtree(t) มีค่ามากกว่าสมาชิกที่รากของ t
3. ทั้ง leftsubtree(t) และ rightsubtree(t) ต่างก็เป็น binary search tree

จะสังเกตได้ว่านิยามของเราไม่มีการนับค่าที่ซ้ำเข้าไปใน tree อีก รูป 6.10 แสดง binary search tree ของจำนวนเต็ม



รูป 6.10 binary search tree

การเรียงสมาชิกแบบนี้จะทำให้สืบค้นได้ง่าย เช่น จากในรูป 6.10 ถ้าเราต้องการหาว่า 4 อยู่ในต้นไม้หรือไม่ เมื่อเราดู โนด แต่ละ โนด เราสามารถรู้ได้ทันทีว่าจะไปสำรวจต้นไม้ข้างใดต่อไป ทำให้เวลาในการค้นหาขึ้นอยู่กับความสูงของต้นไม้เท่านั้น ซึ่งก็แปรตาม log ของจำนวนโนดอีกทีหนึ่ง

แนวคิดในการเขียนโปรแกรมสร้างต้นไม้ค้นหาแบบทวิภาค

ก่อนอื่นเรามานิยามโนดหนึ่งอันเสียก่อน รูปแบบของ โนดจะใช้ตามโค้ดรูป 6.11 โดยส่วนที่เป็นโนดนั้น ประกอบด้วย

- element ซึ่งเป็นข้อมูลภายในโนดนั้น
- left ซึ่งเป็นโนดที่เป็นโนดรากของต้นไม้ย่อยข้างซ้าย
- right ซึ่งเป็นโนดที่เป็นโนดรากของต้นไม้ย่อยข้างขวา

```
1:    class BinaryNode{
2:        // Constructors
3:        BinaryNode(Comparable theElement){
4:            this( theElement, null, null );
5:        }
6:
7:        BinaryNode(Comparable theElement, BinaryNode lt,
BinaryNode rt ){
8:                element  = theElement;
9:                left     = lt;
10:               right    = rt;
11:           }
12:
13:       // ข้อมูลของโนดนี้
14:       Comparable element; // ข้อมูลภายในโนด
15:       BinaryNode left;    // ลูกข้างซ้าย
16:       BinaryNode right;   // ลูกข้างขวา
17:   }
```

รูป 6.11 โค้ดของส่วนที่เป็นโนด

เพื่อให้สมาชิกของต้นไม้เปรียบเทียบกันได้ง่ายไม่ว่ามาจากคลาสใดก็ตาม ผมจะกำหนดให้สมาชิกแต่ละตัวเป็น object ของคลาส ที่ implements Comparable interface (รูป 6.12)

```
1:    public interface Comparable{
2:        /**
3:         * @return จำนวนเต็มที่น้อยกว่าศูนย์ ถ้า this มีค่าน้อยกว่า x
4:         * return จำนวนเต็มที่มากกว่าศูนย์ ถ้า this มีค่ามากกว่า x
5:         * ไม่งั้นก็ return ศูนย์
6:        */
7:        public int compareTo(Object x);
8:    }
```

รูป 6.12 Comparable interface

สำหรับในจาวาเองก็มีคลาสที่ implements Comparable อยู่แล้วนั่นคือ String ดังนั้นถ้าเรามี

```
1:    String s = “men”;
2:    int x = s.compareTo(“man”);
```

รูป 6.13 ตัวอย่างการใช้งาน Comparable interface

ค่าที่โค้ดนี้ return จะมากกว่า 0 เพราะ men มาทีหลัง man เรียงตามในพจนานุกรม ค่าจริงๆนั้นคือ 4 เพราะ e กับ a ห่างกัน 4 ตำแหน่ง

เมธอดต่างๆของคลาส BinarySearchTree ของเรา จะเป็นดังรูป 6.14

```
1:    // Comparable find( x ) --> หา x แล้วรีเทิร์นออกมา
2:    // Comparable findMin( )  --> หาของที่ค่าน้อยสุด
3:    // Comparable findMax( )  --> หาของที่ค่ามากสุด
4:    // void insert( x ) --> ใส่ x
5:    // void remove( x ) --> เอา x ออก
6:    // boolean isEmpty( ) --> true ถ้าเป็นต้นไม้เปล่าๆ ไม่งั้นก็false
7:    // void makeEmpty( )  --> เอาของออกจากต้นไม้หมด
8:    //  void printTree( )  --> พิมพ์สมาชิก เรียงจากน้อยไปมาก
```

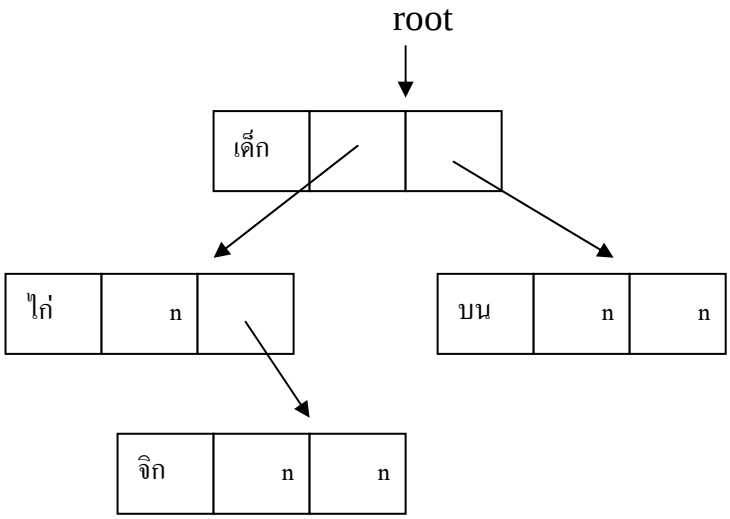
รูป 6.14 เมธอดของ binary search tree

ตัวของต้นไม้เองเราสร้างจากแค่ โนด ที่เป็นรากเท่านั้น (อาจให้มีตัวแปร size เก็บจำนวน โนดทั้งหมดอีกตัวหนึ่ง) ลิงค์ที่ออกจากรากไปจะต่อไปยังส่วนต่างๆของต้นไม้เอง ฉะนั้นโครงสร้างของคลาสของเราจะเป็นดังรูป 6.15

```
1:    public class BinarySearchTree{
2:        private BinaryNode root;
3:
4:        //เมธอด  ต่างๆ...
```

รูป 6.15 โครงสร้างของคลาส binary search tree

รูป 6.16 แสดงต้นไม้ของข้อมูลในพจนานุกรมภาษาไทยที่สร้างขึ้นมาโดยใช้คลาสนี้



รูป 6.16 ต้นไม้ตัวอย่าง

รายละเอียดของเมธอดต่างๆของต้นไม้ทวิภาค

เราเริ่มจากอันที่ง่ายก่อนเลยนะ คือตัว constructor ดูรูป 6.17 (เรายังอยู่ในคลาส BinarySearchTree อยู่นะ)

```
1:    public BinarySearchTree( ){
2:        root = null;
3:    }
```

รูป 6.17 constructor ของ binary search tree

ต่อไปจะเป็นเมธอดที่ตรวจดูของเข้าไปในต้นไม้ แล้วรีเทิร์นของนั้นออกมาถ้าหาเจอ หรือรีเทิร์น null ถ้าหาของชิ้นนั้นไม่เจอ นี่คือเมธอด find สำหรับเมธอดนี้นั้นมีแนวคิดในการเขียนดังนี้

- เมื่อเราเริ่มดู โนดโนด หนึ่ง (ราก จะเป็นโนดแรกที่ดู)
  - ถ้า โนด ที่เราดู เป็น null ให้ return null เลย
  - ถ้า element ใน โนด นั้น น้อยกว่า x ให้เปลี่ยนไปดูที่ โนด ลูกขวาของมัน
  - ถ้า element ใน โนด นั้น มากกว่า x ให้เปลี่ยนไปดูที่ โนด ลูกซ้ายของมัน

- ถ้า element ใน โนด นั้น เท่ากับ x ให้ รีเทิร์นโนดที่มี x นั้นเลย

ตัวโค้ดอยู่ในรูป 6.18 ซึ่งเป็นการหา x โดยเริ่มหาจากโนด t

```
1:    private BinaryNode find( Comparable x, BinaryNode t ){
2:        if( t == null )
3:          return null;
4:        if( x.compareTo( t.element ) < 0 )
5:          return find( x, t.left );
6:        else if( x.compareTo( t.element ) > 0 )
7:          return find( x, t.right );
8:        else
9:          return t;     //เจอแล้ว
10:   }
```

รูป 6.18 เมธอด find

ซึ่งตอนใช้งานจริง เรารีเทิร์น Comparable และก็เริ่มหาจากรากด้วย ดังนั้นต้องมีการปรับโดยเขียนโปรแกรมเพิ่มเติมเล็กน้อย ดังที่เห็นในรูป 6.19 โดยเมธอดที่โอเวอร์โหลด find นั้นบังคับให้เริ่มหาจากราก แล้วจากนั้นให้หาของข้างในโนดออกมาอีกที ด้วยเมธอด elementAt

```
1:    public Comparable find( Comparable x ){
2:        return elementAt( find( x, root ) );
3:    }
4:
5:    private Comparable elementAt( BinaryNode t ){
6:        return t == null ? null : t.element;
7:    }
```

รูป 6.19 โค้ดเพิ่มเติมเพื่อการใช้งานที่ถูกต้องของ find

จริงๆโค้ดในรูป 6.18 จะเขียนแบบลูปก็ได้ ลูปนั้นจะเร็วและกินที่ในหน่วยความจำน้อยกว่า recursion

---

แบบฝึกหัด

1. จงเขียนเมธอด find ด้วยการใช้ลูป

ในเรื่องของเวลานั้น เวลาที่แย่ที่สุดจะเกิดเมื่อเราสำรวจ skewed tree ซึ่งจะแปรตามจำนวน โนด ส่วนเวลาโดยเฉลี่ย เราคิดโดย ก่อนอื่นดูที่ต้นไม้แบบเต็มต้นก่อน จากนิยามที่ 6-8

$$E(t) \geq (k/2)\lfloor \log_2 k \rfloor$$

k คือจำนวนใบ และ E(t) คือผลบวกของความลึกของทุกใบ ดังนั้นเวลาโดยเฉลี่ยในการสำรวจต้นไม้เต็มต้นก็ต้องเป็น E(t) หารด้วย k

$$\frac{E(t)}{k} \geq \frac{(k/2)\lfloor \log_2 k \rfloor}{k}$$

จากนิยาม 6-3 เรารู้ว่าต้นไม้แบบเต็มต้นจะมี $leaves(t) = \frac{n(t)+1}{2.0}$ นั่นคือ เราแทนค่านี้แทน k ได้

$$\frac{E(t)}{k} \geq 0.5\left\lfloor \log_2(\frac{n(t)+1}{2.0}) \right\rfloor$$

นั่นคือ เวลานี้จะมากกว่าหรือเท่ากับฟังก์ชั่นของ $\log_2(n(t))$ อย่าลืมว่านี่คือต้นไม้แบบเต็มต้น ถ้าเป็นต้นไม้ทวิภาคแบบธรรมดา เวลาก็ต้องนานกว่านี้ ซึ่งอสมการจะยังคงเดิม เราสรุปได้ว่าเวลาเฉลี่ย = $\Omega(\log_2 n)$ เมื่อ n เป็นจำนวน โนด

มาลองดูอีกสักเมธอดที่ใช้ในการหาของในต้นไม้ สมมติว่าเราต้องการจะหาค่าสมาชิกที่น้อยที่สุดในต้นไม้ แน่นอนว่าสมาชิกตัวนี้จะต้องอยู่ในโนดซ้ายสุด เราสามารถเขียนโค้ดได้ดังรูป 6.20 โดยตอนที่เรียกใช้ก็ให้เริ่มค้นหาจาก โนด ที่เป็น root ซึ่งจะเหมือนกับในกรณีของ find คือเราเขียนเมธอดที่เริ่มหาจากโนดใดก็ได้โนดหนึ่งก่อน จากนั้นจึงเขียนอีกเมธอดหนึ่งซึ่งกำหนดให้โนดที่จะเริ่มหาเป็น root แล้วหาค่าออกมา

```
1:    private BinaryNode findMin( BinaryNode t ){
2:        if( t == null )
3:            return null;
4:        else if( t.left == null )
5:            return t;
6:        return findMin( t.left );
7:    }
8:
9:    public Comparable findMin( ){
10:       return elementAt( findMin( root ) );
11:   }
```

รูป 6.20 เมธอดหาค่าน้อยที่สุดในต้นไม้

การหาค่าที่มากที่สุดก็เช่นเดียวกัน ดังแสดงในรูป 6.21 แต่ตัวอย่างนี้จะเป็นการทำโดยใช้ลูป แทนเพื่อให้เห็นตัวอย่างที่ชัดเจน

```
1:    private BinaryNode findMax( BinaryNode t ){
2:        if( t != null )
3:            while( t.right != null )
4:                t = t.right;
5:        return t;
6:    }
7:
8:    public Comparable findMax( ){
9:        return elementAt( findMax( root ) );
10:   }
```

รูป 6.21 เมธอดหาค่ามากที่สุดในต้นไม้

แบบฝึกหัด

1. จงเขียนเมธอด findMin เสียใหม่ด้วยการใช้ลูป
2. จงเขียนเมธอด findMax เพื่อหาค่าสมาชิกที่มากที่สุดในต้นไม้ โดยไม่ใช้ลูป

คราวนี้มาดู insert บ้าง หลักการจะเหมือนกับ find เลย หลักการคือ

ถ้าหา x ไปทั้งต้นแล้วยังไม่เจอ (เจอ null เป็นตัวสุดท้ายนั่นเอง)ให้เอา x ใส่ไว้ในที่ๆ มันควรอยู่ (สร้างโนดใหม่โดยมี x อยู่ข้างใน แล้วเอาโนดนั้นใส่แทน null ตัวที่เจอ) ส่วนในการ return นั้น จะ return โนดที่เราใช้เริ่มค้น(แต่สภาพต้นไม้จะเป็นหลังจากได้ insert x แล้วนั่นเอง) นอกจากนี้ยังมีเมธอด insert อีกตัวหนึ่ง ซึ่งเป็นตัวบอกว่า การค้นนั้นจะเริ่มจาก root โปรดสังเกตว่าถ้ามี x อยู่แล้ว เมธอด insert จะไม่ทำอะไร โค้ดของเมธอด insert ทั้งสองเมธอดนั้นอยู่ในรูป 6.22

```
1:    private BinaryNode insert(Comparable x,BinaryNode t )
2:    {
3:        if( t == null )
4:           t = new BinaryNode( x, null, null );
5:        else if( x.compareTo( t.element ) < 0 )
6:           t.left = insert( x, t.left );
7:        else if( x.compareTo( t.element ) > 0 )
8:           t.right = insert( x, t.right );
9:        else
10:          ;  // Duplicate; do nothing
11:       return t;
12:   }
13:
14:   public void insert( Comparable x )
15:   {
16:       root = insert( x, root );
17:   }
```

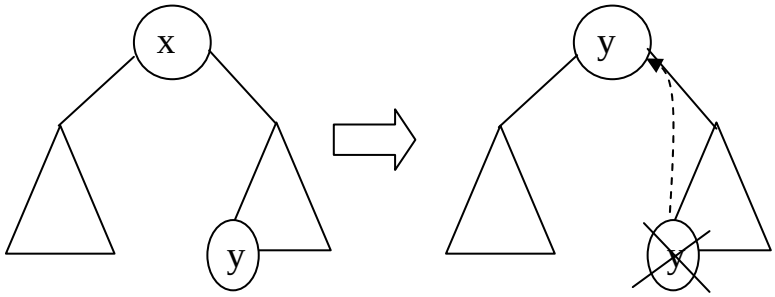
รูป 6.22 โค้ดของ insert

ส่วนการเอาของออกจากต้นไม้นั้น ทำด้วยเมธอด remove ซึ่งหลักการจะคล้ายกับการ insert คือ

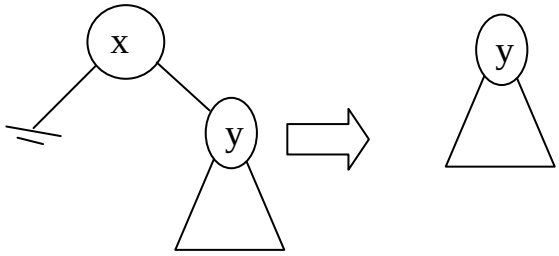
- ถ้าโนดที่เราเริ่มหา เป็น null อยู่แล้ว ก็ไม่ต้องทำอะไร
- ถ้า x ที่เราต้องการหา มีค่าน้อยกว่าสมาชิกที่อยู่ในโนดที่เรากำลังสนใจอยู่ในปัจจุบัน ให้ไปลบ x ออกจากกิ่งซ้าย
- ถ้า x ที่เราต้องการหา มีค่ามากกว่าสมาชิกที่อยู่ในโนดที่เรากำลังสนใจอยู่ในปัจจุบัน ให้ไปลบ x ออกจากกิ่งขวา
- ถ้า x ที่เราต้องการหา มีค่าน้อยเท่ากับสมาชิกที่อยู่ในโนดที่เรากำลังสนใจอยู่ในปัจจุบัน ดูว่า
    - ถ้ากิ่งย่อยทั้งซ้ายและขวาไม่เป็นกิ่งว่าง ให้เอาสมาชิกตัวที่น้อยที่สุดของกิ่งขวาขึ้นมาแทนที่ x จากนั้นเอาสมาชิกตัวนั้นออกจากกิ่งขวา (ดูรูป 6.23) (จริงๆแล้วอาจจะเอาสมาชิกที่ค่ามากที่สุดของกิ่งซ้ายมาแทน x ก็ได้)

- ถ้ากิ่งย่อยซ้ายหรือขวากิ่งใดกิ่งหนึ่งว่าง ให้เอากิ่งข้างที่มีอยู่ขึ้นมาแทนโนดที่มี x (รูป 6.24)

- หลังจากการทำทุกอย่างเสร็จสิ้น รีเทิร์นโนดที่เราสนใจตอนแรกออกมา



รูป 6.23 เอาสมาชิกค่าน้อยที่สุดของกิ่งขวาขึ้นมาแทนสมาชิกตัวที่ถูกลบ



รูป 6.24 การเอา subtree ขึ้นมาแทนที่โนดที่ถูกลบ

โค้ดของการ remove นี้อยู่ในรูป 6.25 โดยมีเมธอดที่สอง ที่กำหนดว่าต้องเริ่มหาจาก root ให้มาด้วย

แบบฝึกหัด

1. จงเขียนเมธอด insert เสียใหม่ด้วยการใช้ลูป
2. จงเขียนเมธอด remove ใหม่ด้วยการใช้ลูป
3. จงเขียนเมธอด removeMin ซึ่งลบค่าที่น้อยที่สุดออกจากต้นไม้

```
1:    /* Internal method to remove from a subtree.
2:     * @param x the item to remove.
3:     * @param t the node that roots the tree.
4:     * @return the new root.
5:     */
6:    private BinaryNode remove(Comparable x,BinaryNode t )
7:    {
8:       if( t == null )
9:          return t;   // Item not found; do nothing
10:      if( x.compareTo( t.element ) < 0 )
11:         t.left = remove( x, t.left );
12:      else if( x.compareTo( t.element ) > 0 )
13:         t.right = remove( x, t.right );
14:      else if( t.left != null && t.right != null )
15:      {
16:         t.element = findMin( t.right ).element;
17:         t.right = remove( t.element, t.right );
18:      }
19:      else
20:         t = ( t.left != null ) ? t.left : t.right;
21:
22:      return t;
23:   }
24:
25:   public void remove( Comparable x )
26:   {
27:      root = remove( x, root );
28:   }
```

รูป 6.25 โค้ดของ remove

มากันที่เมธอดชุดถัดไป เมธอด makeEmpty นั้น ทำให้ต้นไม้กลายเป็นต้นไม้ว่าง ด้วยการ ตั้งให้ root เป็น null ไปเสีย ส่วนเมธอด isEmpty นั้นก็ตรวจว่าต้นไม้ว่างหรือไม่ โดยดูจาก root ว่าเป็น null หรือไม่ นั่นเอง สองเมธอดนี้อยู่ในรูป 6.26

```
1:    public void makeEmpty( ){
2:        root = null;
3:    }
4:
5:    public boolean isEmpty( ){
6:       return root == null;
7:    }
```

รูป 6.26 โค้ดของ makeEmpty และ isEmpty

เมธอดชุดสุดท้ายที่เราจะกล่าวถึงกัน คือ เมธอดที่ใช้ในการพิมพ์สมาชิกที่อยู่ในต้นไม้ออกมาโดยพิมพ์เรียงลำดับจากน้อยไปมาก ซึ่งการทำงานของเมธอดนี้ไม่มีอะไรซับซ้อนเลย เพียงแค่เรียกเมธอดเดิมเพื่อทำการพิมพ์จาก left subtree ก่อน แล้วจึงพิมพ์ของที่อยู่ใน root จากนั้นจึงเรียกเมธอดเดิมอีกครั้งเพื่อทำการพิมพ์จาก right subtree โค้ดของเมธอดนี้อยู่ในรูป 6.27

```
1:    /**
2:      * Internal method to print a subtree in sorted order.
3:      * @param t the node that roots the tree.
4:      */
5:    private void printTree( BinaryNode t ){
6:       if( t != null ){
7:          printTree( t.left );
8:          System.out.println( t.element );
9:          printTree( t.right );
10:      }
11:   }
12:
13:   public void printTree( ){
14:      if( isEmpty( ) )
15:         System.out.println( "Empty tree" );
16:      else
17:         printTree( root );
18:   }
```

รูป 6.27 โค้ดของการพิมพ์สมาชิกในต้นไม้ออกมา

ต่อไปจะเป็นตัวอย่างโค้ดที่แสดงถึงการทำงานของต้นไม้ที่เรานิยาม คำอธิบายนั้นแทรกเป็นคอมเม้นต์ระหว่างบรรทัดในโค้ด ขอให้ดูรูป 6.28

```
1:    public static void main(String[ ] args){
2:       //สร้างต้นไม้
3:       BinarySearchTree t = new BinarySearchTree( );
4:       final int NUMS = 4000;
5:       final int GAP  =   37;
6:
7:       System.out.println( "Checking... (no more output
means success)" );
8:
9:       //ใส่ของเข้าไปในต้นไม้ โดยใส่ตัวเลขเพิ่มทีละ 37 แล้วmod 4000 ใส่จนกว่าจะ mod ได้
0
10:      for( int i = GAP; i != 0; i = ( i + GAP ) % NUMS )
11:         t.insert( new MyInteger( i ) );
12:
13:      //ลูปเอาของออก เอาเลขคี่ออกจากต้นไม้ให้หมด
14:      for( int i = 1; i < NUMS; i+= 2 )
15:         t.remove( new MyInteger( i ) );
16:
17:      if( NUMS < 40 )
18:         t.printTree( );
19:
20:      //ถ้าค่าที่น้อยที่สุดในต้นไม้ไม่เป็นสอง หรือ ค่าที่มากที่สุด ไม่เป็น NUMS-2 จะต้องแจ้งให้รู้
21:      if( ((MyInteger)(t.findMin( ))).intValue( ) != 2 ||
22:      ((MyInteger)(t.findMax())).intValue()!=NUMS - 2 )
23:      {
24:         System.out.print(“FindMin or FindMax error" );
25:         System.out.println();
26:      }
27:
28:      //ลองหาค่าเลขคู่ในออบเจ็กต์ แต่ถ้าค่าไม่ตรง แสดงว่า การหานั้นผิด หรือไม่ก็การใส่นั้นผิด
29:      for( int i = 2; i < NUMS; i+=2 )
30:         if(((MyInteger)(t.find(new MyInteger(i)
31:            ))).intValue()!=i )
32:            System.out.println( "Find error1!" );
33:      //ลองหาค่าเลขคี่ แต่ถ้าเจอ แสดงว่า ที่เอาออกตอนแรกนั้นทำไม่สำเร็จ ต้องแจ้ง error
34:      for( int i = 1; i < NUMS; i+=2 )
35:      {
36:         if( t.find( new MyInteger( i ) ) != null )
37:            System.out.println( "Find error2!" );
38:      }
39:   }
```

รูป 6.28 โปรแกรมตัวอย่างการใช้งาน binary search tree

สำหรับเรื่องต้นไม้นั้น เราจะเรียนกันแค่นี้ ในบทต่อไปเราจะมาเรียนเรื่องต้นไม้แบบพิเศษ ที่เรียกว่า ต้นไม้แบบ AVL

แบบฝึกหัด

1. จงอธิบายว่า binary search tree คืออะไร
2. จงเปรียบเทียบ binary search tree กับ linked list
3. มี binary search tree อีกรูปแบบหนึ่งที่เราสามารถบังคับความเร็วในการค้นได้ในระดับหนึ่ง จงอธิบายว่า binary search tree แบบนั้นเป็นอย่างไร
4. จงเขียนเมธอดของ binary search tree ต่อไปนี้

public LinkedList inOrder(BinaryNode t)

ซึ่งใส่ของในต้นไม้ (ต้นไม้มี root ที่ t) เข้าไปใน linked list โดยจัดเรียงแบล inorder

5. จงเขียนเมธอดของ binary search tree ต่อไปนี้

public LinkedList breadthFirst(BinaryNode t)

ซึ่งใส่ของจากต้นไม้ (ต้นไม้มี root ที่ t) เข้าไปใน linked list โดยจัดเรียงแบบ breadth first

6. จงเขียนเมธอด isBST เพื่อทดสอบว่า tree ที่เป็น input เป็น binary search tree หรือไม่

public Boolean isBST(BinaryNode t)

7. จงวาดรูปแสดงการทำงานของโปรแกรมในรูป 6.28 ลด NUMS ให้เหลือจำนวนน้อยๆก็พอ
8. จงเขียนเมธอด add2Tree ซึ่งรับ ต้นไม้ค้นหาแบบทวิภาค(binary search tree) สองต้นที่มีสมาชิกประเภทเดียวกัน แล้วรวมสองต้นนี้เข้าด้วยกัน รีเทิร์นต้นไม้ต้นใหม่ที่เป็นผลรวมของสองต้น โดยไม่เปลี่ยนต้นไม้ต้นฉบับ
9. จงเขียนเมธอดเพื่อพิมพ์ของในต้นไม้จากมากไปน้อย
10. จงเขียนเมธอด maxSort ซึ่งรับต้นไม้ค้นหาแบบทวิภาคเข้ามา แล้วเปลี่ยนการเรียงของในต้นไม้ใหม่ ให้เป็นจากค่ามากไปค่าน้อย

11. จงนิยามโนดใหม่ โดยให้มีตัวแปร parent เพื่อชี้กลับไปยังโนดพ่อได้ จากนั้นจงเขียนเมธอด findGrandParent ซึ่งรับโนดเป็นอินพุตแล้วรีเทิร์นโนดที่เป็นโนด parent ขึ้นไปสองขั้น
12. จงเขียนเมธอด nextValue ซึ่งรับComparable เป็นอินพุต แล้วรีเทิร์นค่าซึ่งเป็น Comparable ตัวถัดไป (ต้องอยู่ในต้นไม้) ออกมา