บทที่ 8
ไพรออริตี้ คิว

บทนี้เราจะมาเรียนโครงสร้างข้อมูลไพรออริตี้คิว (priority queue) (ต่อไปในบทนี้ผมจะใช้ชื่อเป็นภาษาอังกฤษ เพื่อความเข้าใจที่ตรงกัน) ซึ่งหมายถึง คิวที่เราสามารถเข้าถึงสมาชิกได้ตามความสำคัญของตัวสมาชิกนั้น ตัวอย่างของ priority queue ที่เห็นชัดก็อย่างเช่นคิวของคนไข้ในโรงพยาบาล ถ้ามีคนที่ขาหักรอคิวอยู่ แต่มีคนที่เลือดพุ่งไม่หยุดมาทีหลัง คนที่เลือดพุ่งไม่หยุดก็ควรได้รับการรักษาก่อน ไม่เช่นนั้นก็ตายแน่นอน ตัวอย่างอีกอย่างหนึ่งก็คือ คิวของการใช้เครื่องพิมพ์ ซึ่งคนที่พิมพ์จำนวนหน้าน้อยควรได้รับการพิจารณาก่อน (ตัวอย่างนี้ซับซ้อนกว่าที่คิดเพราะถ้าให้แต่คนหน้าน้อยก่อนอย่างเดียว คนที่พิมพ์หลายหน้าก็จะไม่ได้งานซักที ดังนั้นจึงต้องมีการถ่วงสมดุลเอาไว้ด้วย)

โดยทั่วไป ถ้าเป็นคิวที่สมาชิกเทียบค่ากันได้นั้น จะให้ค่าน้อยถือว่าสำคัญกว่าค่ามาก และถ้าสมาชิกสองตัวมีค่าเท่ากัน ตัวที่อยู่ในคิวมานานกว่าก็ควรจะถือว่าสำคัญกว่า การเปรียบเทียบค่านั้น เราใช้ Comparator หรือ Comparable Interface แบบที่ใช้เปรียบเทียบสิ่งของในโครงสร้างต้นไม้

ก่อนอื่นขอทบทวน Comparable Interface ก่อน อินเตอร์เฟสนี้มีเมธอดอยู่เมธอดเดียว คือ int compare(Object o1, Object o2) ซึ่ง o1 และ o2 นั้นต้องเป็นออบเจ็กต์ของคลาสเดียวกันหรือของสับคลาสกัน (ไม่เช่นนั้นจะเกิด CastCastException) เอ้าท์พุตจะเป็นค่าลบถ้า o1 ถือว่ามีค่าน้อยกว่า o2 เป็นค่าบวกถ้า o1 ถือว่ามีค่ามากกว่าo2 หรือมีค่า 0 ถ้า o1 ถือว่ามีค่าเท่ากับ o2

เราลองมาดูเมธอดพื้นฐานทั้งหมดที่ priority queue ควรมีกัน รูป 8.1 แสดงโค้ดของอินเตอร์เฟสที่เราจะใช้ทำ priority queue

```
1:    public interface PriorityQueue{
2:       //รีเทิร์นจำนวนสมาชิกในคิวนี้
3:       int size();
4:
5:       //รีเทิรืน true ถ้าคิวนี้ไม่มีสมาชิก ไม่เช่นนั้นก็รีเทิร์น false
6:       boolean isEmpty();
7:
8:       //สมาชิกใหม่จะถูกเติมลงในคิว
9:       void add(Object element);
10:
11:      // Precondition: ต้องไม่เป็นคิวที่ว่าง ไม่งั้นจะ throw
12:      // NoSuchElementException
13:      // Postcondition: รีเทิร์นออบเจ็คต์ตัวที่สำคัญที่สุดใน PriorityQueue นี้
14:      Object getMin( );
15:
16:      // Precondition: ต้องไม่เป็นคิวที่ว่าง ไม่งั้นจะ throw
17:      // NoSuchElementException
18:      // Postcondition: เอาออบเจ็คต์ตัวที่สำคัญที่สุดใน PriorityQueue นี้
19:      // ออกจากคิวและรีเทิร์นออบเจ็คต์นี้เป็นคำตอบของเมธอดด้วย
20:      Object removeMin();
21:   }
```

รูป 8.1 PriorityQueue Interface

การสร้าง priority queue ในภาษาจาวา

วิธีการทำนั้นมีหลายวิธี เราจะแยกพูดถึงเป็นหัวข้อไปดังนี้

- ใช้คิวธรรมดา ทำไม่ได้เพราะไม่มีการจัดลำดับความสำคัญของสมาชิก
- ใช้อาร์เรย์ของคิว คือมีแต่ละช่องเป็นลิสต์ที่เก็บของในแต่ละ priority ไว้ วิธีนี้มีข้อเสีย คือ ถ้ามี priority จำนวนมากจะทำให้เปลืองที่มาก นอกจากนี้เราก็ยังไม่รู้อีกด้วยว่าจะมีกี่ priority ดังนั้นจึงไม่รู้ว่าจะต้องสร้างอาร์เรย์ขนาดใหญ่เท่าใด
- ใช้อาร์เรย์ลิสต์ การใช้วิธีนี้ทำให้เรียงสมาชิกได้ง่าย getMin สามารถทำได้ด้วยเวลาคงที่ แต่ตอนเติมจะเสียเวลาหาตำแหน่งที่เติม และเสียเวลาเลื่อนอาร์เรย์ตอนเอาของ

ออกด้วย (อาจมีการเลื่อนช่องเยอะ) เวลาที่เสียนั้นเป็นเวลาเชิงเส้น แต่ก็ยังถือว่าช้าไปอยู่ดี

- ใช้ Linked List วิธีนี้ตอนที่เอาของออกจะไม่ต้องทำอะไรเพิ่ม แค่ update pointer เพราะฉะนั้นเวลาในการ removeMin จะคงที่ แต่ตอนเติมของลงในคิวก็ต้องใช้เวลาหาช่องเติม

การใช้ลิงค์ลิสต์สร้าง priority queue

เรามาลองดู Priority queue ที่สร้างจาก linked list กัน ดังรูป 8.2 จะเห็นว่าเราสามารถนำเมธอดของลิสต์หลายๆเมธอดมาใช้งาน รูปนี้แสดงเมธอดพื้นฐานเท่านั้น

```
1:    public class LinkedPQ implements PriorityQueue{
2:       LinkedList list;
3:       Comparator comparator;
4:
5:     public LinkedPQ( ){
6:       list = new LinkedList( );
7:       comparator = null; //โดย default นั้นใช้ Comparable แทน
8:     }
9:
10:   public LinkedPQ(Comparator comp){
11:      this();
12:      comparator = comp;
13:   }
14:
15:   public int size( ){
16:      return list.size( );
17:   }
18:
19:   public Object getMin( ){
20:      return list.getFirst( );
21:    }
22:
23:   // ยังมีเมธอดอื่นๆอีก ดังจะได้แสดงในรูปต่อๆไป
24:   }
```

รูป 8.2 โค้ดส่วนที่หนึ่งของ priority queue ที่สร้างจากลิสต์

รูป 8.3 แสดงโค้ดของเมธอด removeMin ที่เอาของที่มีความสำคัญที่สุดออกจากคิว และ add ซึ่งเติมของลงในคิวไปตำแหน่งต่างๆตามลำดับความสำคัญ (รูปอาจจะเล็กลงไปสักนิดแต่จะได้อ่านโค้ดได้เข้าใจมากขึ้น ไม่เช่นนั้นจะหลายบรรทัดเกินไป)

```
1:      public Object removeMin( ){
2:         return list.removeFirst( );
3:      }
4:
5:      public void add (Object element){
6:         if(list.isEmpty( ) || compare(element, list.get(list.size() –
7:                                                          1)) >= 0)
8:            list.add (element);
9:         else {
10:           ListIterator itr = list.listIterator( );
11:           while (compare (element, itr.next( )) >= 0)
12:              ;
13:           itr.previous( );  // back up one position
14:           itr.add (element);
15:        }
16:     }
```

ทำไมต้องย้อน?

รูป 8.3 โค้ดของเมธอด removeMin และ add

นิสิตอาจสังเกตว่า ทำไมจึงต้องย้อน iterator ไปหนึ่งทีด้วย ขอให้อย่าลืมว่าตอนที่ while loop exit ออกนั้น แม้จะได้ compare(element,itr.next()) <0 ก็ตาม ตัว itr.next(); ก็จะทำให้iterator เลื่อนไปอีกหนึ่งตำแหน่งอยู่ดี ทำให้ถ้าแทรกสมาชิกใหม่โดยไม่ถอยหลัง จะเป็นการเติมสมาชิกใหม่เลยออกไปจากที่ที่ควรจะอยู่หนึ่งตำแหน่ง ส่วนเวลานั้นเป็น O(n) ก็เพราะจำนวนลูปขึ้นอยู่กับ n (n คือจำนวนสมาชิกในคิว)

หลายคนอาจจะยังงงอยู่ เรื่องที่ต้องใช้ previous ดังนั้นผมจะขอยกตัวอย่างสั้นๆตัวอย่างหนึ่งก็แล้วกัน

- ถ้าเรามี 1,2,4,5 และต้องการ add 3
- โปรแกรมการ add นั้นจะหยุดเมื่อ itr.next() เป็น 4
- แต่ตัว iterator ได้เลื่อนไปที่ 5 ซะแล้ว
- ดังนั้น ต้อง previous() กลับมา ก่อนที่จะใส่ 3

ต่อไปเรามาดูเมธอด compare กัน โดยหลักการทำงานก็คือ ถ้าไม่ได้กำหนด comparator ให้ใช้เมธอด compare ของ Comparable interface โดยตัวออบเจ็กต์ที่เราเอามาเปรียบเทียบกันจะต้อง

implement Comparable interface ส่วนถ้ามี comparator ให้ใช้เมธอดของ comparator ตัวนั้น โค้ดของเมธอดนี้อยู่ในรูป 8.4

```
1:      protected int compare (Object elem1, Object elem2) {
2:        return (comparator==null ? ((Comparable)elem1).compareTo(elem2)
3:                  : comparator.compare (elem1, elem2));
4:      }
```

รูป 8.4 โค้ดแสดงการใช้เมธอด compare

การใช้เซตสร้าง priority queue

โครงสร้างจริงๆที่จะใช้เป็นตัวอย่างในที่นี้คือ treeset ซึ่งเป็นหนึ่งในโครงสร้างข้อมูลของจาวา คอลเลคชั่น ให้มีการกำหนดตัว comparator ด้วย ตัวคอนสตรัคเตอร์ของเราก็รับ comparator ตัวนี้ เป็นอินพุต รูป 8.5 แสดงคอนสตรัคเตอร์และตัวแปรของเซตที่ใช้ทำ priority queue นี้ ส่วนรูป 8.6 แสดงเมธอด add และเมธอดอื่นๆที่ใช้กับเซตนี้

```
1:    public class SetPQ implements PriorityQueue{
2:       TreeSet set;
3:       Comparator comparator;
4:
5:       public SetPQ (Comparator c){
6:          comparator = c;
7:          set = new TreeSet(c);
8:       }
9:
10:      //มีเมธอดอื่นต่ออีก
11:   }
```

รูป 8.5 โครงสร้างเซตที่ใช้สร้าง priority queue

เมธอดในรูป 8.6 ทั้งสามเมธอดนั้นมี worstTime(*n*) เป็น LOGARITHMIC ของ *n* เนื่องจากเป็นต้นไม้ ดังนั้นเวลาจึงอยู่ในรูปแบบของ log ส่วนเมธอดต่างๆภายในนั้น เรียกใช้ของเซตได้เลย

เป็นอันว่า ตัวอย่างการใช้วิธีต่างๆ อย่างง่ายๆ ในการสร้าง priority queue ก็จบลงเพียงเท่านี้ ในส่วนต่อไปเราจะมาดูการสร้าง priority queue ด้วยต้นไม้บริบูรณ์ ซึ่งเป็นวิธีที่ทำให้สามารถนำข้อมูลออกมาจากคิวนี้ได้เร็วที่สุด วิธีนี้เรียกว่าการสร้างฮีป(heap) ซึ่งจะเป็นเรื่องหลักของบทนี้

```
1:    public void add(Object element){
2:       set.add(element);
3:     }
4:
5:     public Object getMin( ){
6:       return set.first( );
7:     }
8:
9:     public Object removeMin( ){
10:      Object temp = set.first( );
11:      set.remove(set.first( ));
12:      return temp;
13:    }
```

รูป 8.6 เมธอดของ priority queue ที่สร้างจากเซต

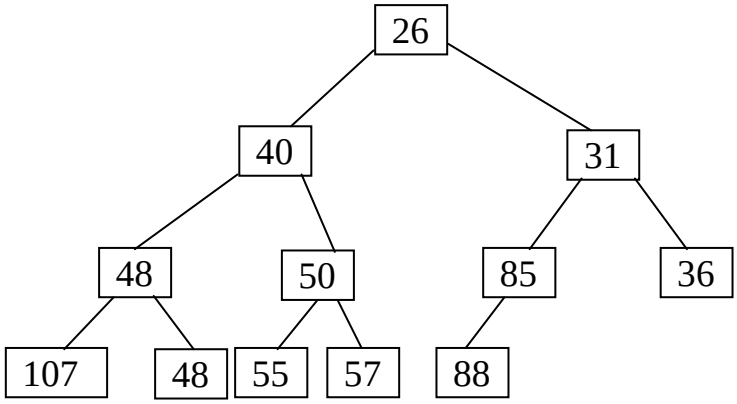
ฮีป(Heap)

ในส่วนนี้เราจะนำหลักการของต้นไม้บริบูรณ์มาสร้าง priority queue ก่อนอื่นต้องมาเตือนความจำกันก่อน ว่าต้นไม้บริบูรณ์คืออะไร ต้นไม้บริบูรณ์ (complete binary tree) คือต้นไม้ที่เต็มทุกระดับ ยกเว้นระดับของใบ ซึ่งระดับของใบนี้ จะต้องถูกเติมจากซ้ายไปขวาเท่านั้น จะไม่มีรูว่างเป็นหย่อมๆ โปรดดูรูป 6.8 ในบทที่หกประกอบ

ฮีปนั้นคือต้นไม้บริบูรณ์ที่

- ไม่มีอะไรอยู่ข้างใน หรือ
- สมาชิกที่อยู่ที่ราก มีค่าน้อยที่สุด(เราถือว่าค่าน้อยสำคัญกว่าค่ามาก ซึ่งฮีปที่เรียงของแบบนี้ เราเรียกว่า minheap)
- ต้นไม้ย่อย ทั้งต้นซ้ายและขวา ก็เป็นฮีปเช่นเดียวกัน

โปรดสังเกต ว่าฮีปนั้นไม่ใช่ต้นไม้ค้นหาแบบทวิภาค(binary search tree) เพราะอย่างไรก็ตามค่าที่น้อยที่สุดก็อยู่ที่รากของมัน รูป 8.7 แสดงตัวอย่างของฮีป จะเห็นว่าค่าน้อยที่สุดจะอยู่ที่รากเสมอไม่ว่าจะเป็นต้นไม้ย่อยใดๆ



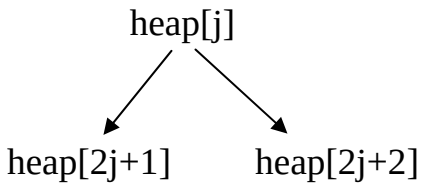
รูป 8.7 ตัวอย่างของฮีป

ต้นไม้บริบูรณ์นั้น เราเก็บในอาร์เรย์ได้ง่าย หา parent กับ child ของโนดต่างๆได้ด้วยการดูค่าดัชนี(index) ดังนั้นฮีปก็สามารถเก็บในอาร์เรย์ได้ง่ายเช่นเดียวกัน รูป 8.8 คือฮีป จากรูป 8.7 นำมาทำเป็นอาร์เรย์

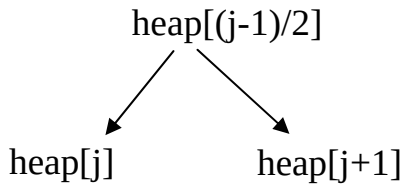
26	40	31	48	50	85	36	107	48	55	57	88	

รูป 8.8 ใช้อาร์เรย์สร้างฮีป โดยสมาชิกภายในเอาจากรูป 8.7

โดยเราสามารถหาดัชนี(index) ของสมาชิกตัวที่อยู่ในโนด child และ parent ได้ด้วยสูตรตามรูป 8.9 และ 8.10 ตามลำดับ โดยถ้าโนดใดมีดัชนีเป็น j ลูกข้างซ้ายของมันจะเป็นโนดที่ 2j+1 และลูกข้างขวาจะเป็นโนดที่ 2j+2 ส่วนถ้าเรานับจากทางลูกย้อนกลับขึ้นไป ถ้าลูกมีค่าดัชนี j หรือ j+1 (สำหรับลูกทางขวา) แล้ว parent จะมีค่าดัชนีเป็น (j-1)/2 (สังเกตรูป 8.7 และ 8.8 เป็นตัวอย่างประกอบได้)



รูป 8.9 ค่าดัชนีของโนดลูก ทั้งซ้ายและขวา นับลงมาจากโนด parent



รูป 8.10 ค่าดัชนีของโนด parent เมื่อนับขึ้นไปจากโนดลูก

การใช้งานฮีป

เราสามารถใช้งานฮีปได้ ดังเช่นตัวอย่างต่อไปนี้ โค้ดในรูป 8.11 ถึง 8.14 แสดงข้อมูลนักเรียน โดยให้คนเกรดน้อยมีความสำคัญ ในโค้ดนี้ให้ดูส่วนที่ขีดเส้นใต้เท่านั้น ส่วนอื่นๆเป็นแค่ส่วนประกอบที่ให้โปรแกรมทำงานได้เท่านั้น

```
1:     import java.util.*;
2:
3:     public class StudentHeap implements Process{
4:         protected final static String PROMPT =
5:        "In the Input line, please enter " +
6:               "a student's name and GPA. To quit, enter ";
7:
8:         protected final String SENTINEL = "***";
9:
10:        GUI gui;
11:
12:        Heap heap;  // same as PriorityQueue heap;
13:
14:        //Postcondition: this StudentHeap has been initialized.
15:        public StudentHeap( ){
16:            gui = new GUI (this);
17:            heap = new Heap ( );
18:            gui.println (PROMPT + SENTINEL);
19:
20:        }
21:        //มีโค้ดส่วนอื่นๆอีก โปรดดูรูปอื่นประกอบ
22:    }
```

รูป 8.11 ข้อมูลนักเรียนที่เราจะใช้ฮีปเก็บ

ฮีปที่ผมนิยามในบทนี้ เป็นคลาสใหม่ที่ถูกสร้างขึ้นมา ในรูป 8.11 เราจะเห็นได้ว่ามีการนิยามตัวแปรและ new ฮีป ขึ้นมาใช้เพื่อเก็บข้อมูลนักเรียน รูป 8.12 แสดงเมธอด processInput ซึ่งเป็นการ

เติมนักเรียนหนึ่งคนที่มีชื่อตามตัวแปร s ลงในฮีป ถ้า s ไม่ใช่ sentinel (ค่าที่บอกว่าเลิกรับอินพุตได้แล้ว) ก็ให้เติมนักเรียนลงในฮีปโดยใช้เมธอด add แต่ถ้าเป็น sentinel ให้เคลียร์ฮีปแล้วเอาท์พุตสิ่งที่อยู่ในฮีปทั้งหมดลงบนจอภาพ โดยวนลูปเรียกเมธอด removeMin เท่านี้เราก็จะได้รายชื่อของนักเรียนโดยคนไม่เก่งจะมีชื่อก่อน

```
1:     public void processInput(String s){
2:        final String RESULTS =
3:           "\nHere are the student names and GPAs:";
4:        final String CLOSE_WINDOW_PROMPT =
5:           "\nThe execution of the project is completed.  " +
6:           " Please close this window when you are ready.";
7:
8:        if(!s.equals(SENTINEL)){
9:           gui.println (s + "\n");
10:          heap.add(new Student(s));
11:          gui.println (PROMPT + SENTINEL);
12:       } else {
13:          gui.println (RESULTS);
14:          while(!heap.isEmpty())
15:             gui.println(heap.removeMin( ));
16:             gui.println(CLOSE_WINDOW_PROMPT);
17:              gui.freeze( );
18:       }
19:    }
```

รูป 8.12 เมธอด processInput ของคลาสที่ใช้งานฮีปของเรา

รูป 8.12 แสดงการใช้งานฮีปอยู่สามจุดคือ เมธอด add เมธอด isEmpty และเมธอด removeMin

ส่วนที่จำเป็นในการนิยามคลาสที่เก็บข้อมูลนักเรียน อีกส่วนหนึ่ง ก็คือ เมธอด compareTo ของนักเรียน โดยเราดู GPA เป็นหลัก โปรดดูโค้ดในรูป 8.13 ซึ่งเมธอดนี้รีเทิร์นจำนวนเต็มที่ < 0, = 0, หรือ > 0 ขึ้นอยู่กับว่า GPA ของ this Student ออบเจ็กต์นั้นน้อยกว่า เท่ากัน หรือมากกว่า GPA ของ o1

ในส่วนต่อไป เราจะมาดูโค้ดของฮีปเลย

```
1:    public class Student implements Comparable {
2:
3:        protected String name;
4:        protected double GPA;
5:
6:        public int compareTo (Object o1) {
7:            if (GPA < ((Student)o1).GPA)
8:                return -1;
9:            if (GPA == ((Student)o1).GPA)
10:               return 0;
11:           return 1;
12:       }
13:   }
```

รูป 8.13 เมธอดแสดงการเปรียบเทียบนักเรียนโดยใช้คะแนนเป็นตัวเทียบ

อิมพลีเม้นเตชั่นของฮีป

ฮีปที่เราทำนี้เป็นอาร์เรย์ที่เก็บต้นไม้บริบูรณ์นั่นเอง โดยเราจะใช้ประโยชน์จากสูตรการหาดัชนีของโนดลูกและโนดพ่อแม่ให้เต็มที่ ฮีปที่เราสร้างจะใช้ตัวแปรดังต่อไปนี้

protected Object[ ] heap;

protected int size;

protected Comparator comparator;

สำหรับคอนสตรัคเตอร์นั้น อยู่ในรูป 8.14 ซึ่งการทำงานของมันก็เพียงแค่สร้างตัวอาร์เรย์ขึ้นมา

```
1:    public Heap( ) {
2:     final int DEFAULT_INITIAL_CAPACITY = 11;
3:       heap = new Object [DEFAULT_INITIAL_CAPACITY];
4:    }
5:
6:    public Heap(Comparator comp){
7:     this();
8:     comparator = comp;
9:    }
```

รูป 8.14 คอนสตรัคเตอร์ของฮีป

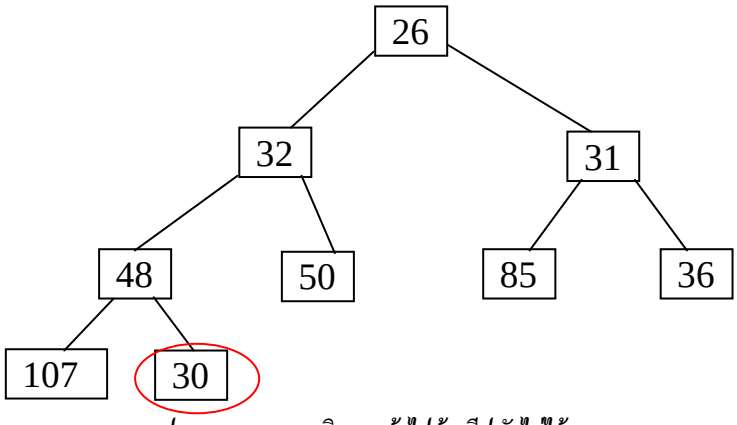
ต่อไปจะเป็นเมธอด add ซึ่งใช้เติมของลงในฮีป อย่าลืมว่าพอเติมเสร็จแล้วฮีปต้องยังเป็นฮีปอยู่ เมธอดนั้นอยู่ในรูป 8.15

```
1:     // Postcondition: element ถูกเติมเป็นสมาชิกของ heap
2:     // เวลา worst case = O(n)   → ถ้าเต็มอาร์เรย์ ก็ใช้เวลาขยาย
3:     // ส่วนเวลาเฉลี่ยเป็นค่าคงที่ → ขึ้นกับเวลา percolateUp
4:     public void add(Object element){
5:
6:          if (++size == heap.length){ //ถ้าอาร์เรย์เต็ม ต้องขยายก่อน
7:
8:               Object[ ] newHeap = new Object [2 * heap.length];
9:               System.arraycopy  (heap, 0, newHeap, 0, size);
10:              heap = newHeap;
11:         }
12:
13:         heap [size - 1] = element; //เติมของลงไปท้ายอาร์เรย์ (ใบของต้นไม้)
14:         percolateUp( ); //แล้วเลื่อนของขึ้นต้นไม้มาจนถึงที่เหมาะสม
15:    }
```

รูป 8.15 เมธอด add สำหรับฮีป

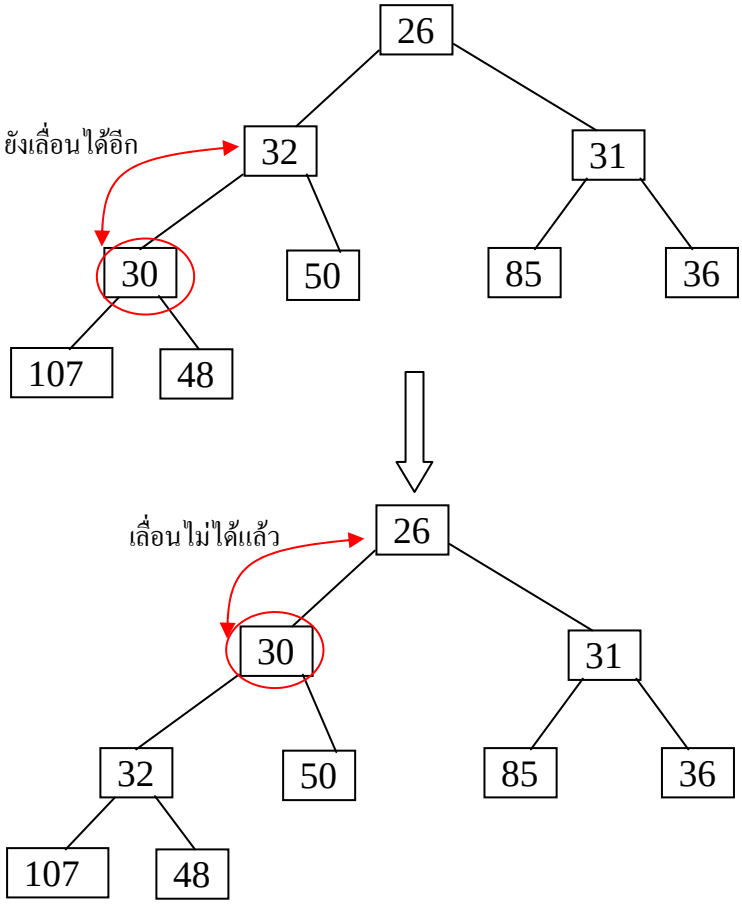
O(n) เกิดจากการขยายอาร์เรย์ (ในการก็อปปี้อาร์เรย์นั้นเราดึงเอาเมธอดของ System มาใช้) เวลาเฉลี่ยนั้นขึ้นกับ percolateUp (เป็นเมธอดในการปรับเลื่อนของที่ใส่ไป ขึ้นต้นไม้จนกว่าจะได้ที่ที่ทำให้ต้นไม้มีคุณสมบัติเป็นฮีปดังเดิม) ดังนั้นเดี๋ยวเราค่อยดูที่ percolateUp

ต่อไปเป็นตัวอย่างที่บอกถึงวิธีการ percolateUp สมมติว่าตอนนี้เราเติมสมาชิกใหม่ คือตัวเลข 30 ลงท้ายอาร์เรย์แล้ว ดังรูป 8.16



รูป 8.16 สภาพตอนเติม 30 เข้าไปท้ายฮีป ยังไม่ได้ percolateUp

การ percolateUp คือ สลับที่ 30 กับพ่อของเขาเรื่อยๆจนกว่าตัวพ่อจะน้อยกว่า พอทำเสร็จเราก็จะได้ฮีป รูป 8.17 แสดงการเลื่อน 30 ด้วยหลักนี้



รูป 8.17 การ percolateUp ของเลข 30

จากรูป 8.17 จะเห็นได้ว่า พอเลื่อนมาถึงใต้ 26 ก็เลื่อนต่อไม่ได้เพราะตัวพ่อมีค่าน้อยกว่า สังเกตดูจะพบว่าเราได้ฮีปเรียบร้อย

คราวนี้เรามาดูโค้ดของ percolateUp กัน แม้หลักจะดูจากต้นไม้แต่อย่าลืมว่าจริงๆเราใช้อาร์เรย์ ดังนั้นโค้ดจึงต้องเป็นตามอาร์เรย์ ดูรูป 8.18

```
1:     // Postcondition: จัด heap โดยเลื่อนตัวท้ายสุดขึ้นต้นไม้ เวลาที่แย่ที่สุดจะเป็นO(log n)
2:     // ส่วนเวลาเฉลี่ยจะเป็นค่าคงที่
3:     protected void percolateUp() {
4:          int parent;
5:          int child = size-1;
6:          Object temp;
7:          while (child>0) {
8:               parent = (child-1)/2;
9:               if(compare(heap[parent],heap[child]) <=0)
10:                   break;
11:              temp = heap[parent];
12:              heap[parent] = heap[child];
13:              heap[child] = temp;
14:              child = parent;
15:         }
16:    }
```

แปรตามความสูงของต้นไม้ เพราะเวลาแย่สุดเกิดตอนที่ต้องเลื่อนจนถึงยอด

รูป 8.18 โค้ดของ percolateUp

จากรูป 8.18 จะเห็นว่าเป็นการลูปสลับที่ลูกกับพ่อไปเรื่อยๆ จะออกจากลูปก็เมื่อเจอพ่อซึ่งน้อยกว่าหรือเท่ากับลูก ในที่นี้ใช้อาร์เรย์ แต่ง่ายเพราะว่าการหาค่าดัชนีของ parent นั้นตายตัว

สำหรับเวลาโดยเฉลี่ยของการ percolateUp นั้น คิดโดยสมมติว่าตัวที่เติมเข้าไปนี่มีค่าอยู่ตรงกลางของค่าทั้งหมด ดังนั้นต้องมีสมาชิกของอาร์เรย์ครึ่งหนึ่งที่มากกว่ามัน แต่เพราะเป็นต้นไม้บริบูรณ์ ดังนั้นครึ่งที่มากกว่านี้จะต้องอยู่ที่ใบแน่ๆ การ percolateUp จึงแค่ต้องเลื่อนให้พ้นระดับของใบเท่านั้น ดังนั้นจึงใช้แค่ 1หรือ 2 ครั้งเท่านั้น ดังนั้นจึงถือว่าเวลาคงที่

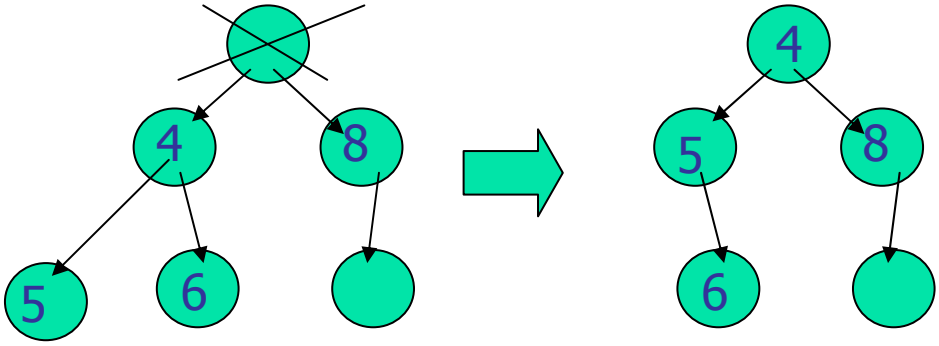
ก่อนที่จะไปดูเมธอด removeMin เรามาคั่นด้วยเมธอดง่ายๆกันก่อน นั่นคือ getMin ซึ่งรีเทิร์นค่าน้อยที่สุด(สำคัญที่สุด)ในฮีปออกมา โค้ดของเมธอดนี้อยู่ในรูป 8.19 ซึ่งไม่ได้มีอะไรยาก

```
1:     public Object getMin() {
2:        if (size == 0)
3:           throw new NoSuchElementException("Empty");
4:        return heap[0];
5:     }
```

รูป 8.19 โค้ดของ getMin

ต่อไปเป็นการเอาของออกจากหัวคิว(เมธอด removeMin นั่นเอง) ซึ่งจะซับซ้อนกว่าการเติมของ เพราะอาจทำให้เสียรูปของต้นไม้บริบูรณ์ได้ ถ้าเสียรูปเมื่อไหร่ยุ่งแน่นอน เพราะ percolateUp

ซึ่งใช้การคำนวณดัชนีของต้นไม้ก็จะเสียไปด้วย รูป 8.20 แสดงการเอารากออกแล้วทำให้ต้นไม้เสียความเป็นต้นไม้บริบูรณ์



รูป 8.20 เอารากออกแล้วทำให้เสียความเป็นต้นไม้บริบูรณ์

วิธีการแก้การเสียความเป็นต้นไม้บริบูรณ์นี้ ทำได้โดยสลับที่รากกับตัวท้ายสุดของอาร์เรย์แล้วลดขนาดอาร์เรย์ (จริงๆไม่ได้ลดแต่ลดค่า size) เพื่อไม่ให้คนเห็นรากที่สลับไปท้ายอาร์เรย์ ยังไงขนาดของอาร์เรย์ที่เรานับเป็นฮีปก็ต้องลดลงอยู่ดี

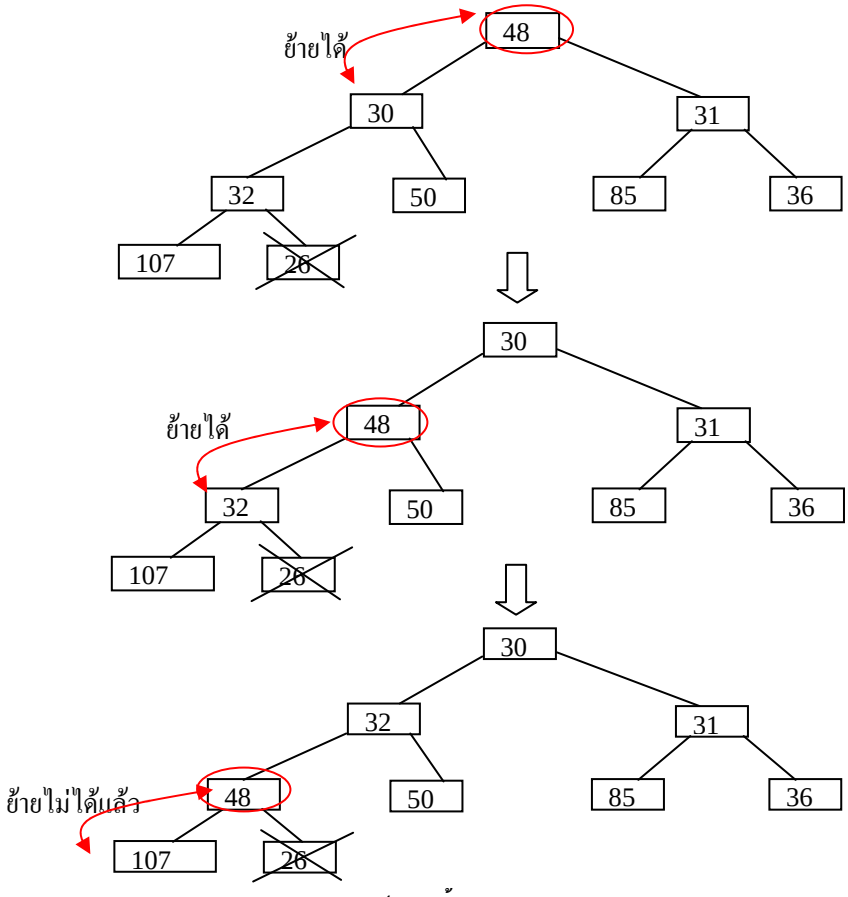
จากนั้นปรับอาร์เรย์ด้วยการเลื่อนตัวที่อยู่ที่รากลงมายังตำแหน่งที่ถูกต้อง ซึ่งเราเรียกว่าการ percolateDown (ในที่นี้ให้เริ่มที่ค่าดัชนี 0) ซึ่งในบทที่ 12 ของหนังสือ Java Collection Framework จะเป็นเรื่อง heap sort ให้ทุกคนไปดูเพิ่มเติมได้

โค้ดของ removeMin นั้นอยู่ในรูป 8.21 ถ้าเราเริ่มจากต้นไม้ผลลัพธ์จากรูป 8.17 แล้วจะเอา 26 ออก เราต้องสลับที่ 26 กับ 48 จากนั้นก็ลด size ลงหนึ่งหน่วย แล้วก็ทำ percolateDown ซึ่งมีหลักการคือ สลับสมาชิกที่ตำแหน่งรากกับลูกที่มีค่าน้อยที่สุด ทำจนกว่าจะสลับไม่ได้ พอทำเสร็จเราก็จะได้ฮีป

เราให้ 48 ย้ายไปอยู่ที่รากแล้วในตอนจะเริ่มทำการ percolateDown รูป 8.22 แสดงภาพขั้นตอนการ percolateDown ของต้นไม้จากรูป 8.17

```
1:     // Precondition: ต้องไม่เป็นคิวที่ว่าง ไม่งั้นจะthrow NoSuchElementException
2:     // Postcondition: เอาออบเจ็คต์ตัวที่สำคัญที่สุดใน PriorityQueue นี้
3:     // ออกจากคิวและรีเทิร์นออบเจ็คต์นี้เป็นคำตอบของเมธอดด้วย
4:     // เวลาที่แย่ที่สุดจะเป็น O(log n)          ขึ้นกับ percolateDown
5:     public Object removeMin( ) {
6:        if (size==0)
7:           throw new NoSuchElementException("Queue empty.");
8:        Object minElem = heap [0];
9:        heap [0] = heap [size – 1];              เอาท้ายอาร์เรย์ใส่ที่ root
10:       heap [--size] = minElem;
11:       percolateDown (0);                       แล้วค่อยปรับเลื่อนตำแหน่ง
12:       return minElem;
13:    }
```

รูป 8.21 โค้ดของ removeMin



รูป 8.22 ขั้นตอนการ percolateDown

โค้ดของการ percolateDown นั้นอยู่ในรูป 8.23

```
1:      // Postcondition: จัด heap โดยเลื่อนตัวบนสุดลงด้นไม้
2:      // เวลาที่แย่ที่สุด และเวลาเฉลี่ยจะเป็น O(log n)  ทั้งคู่
3:      protected void percolateDown(int start) {
4:        int parent = start;
5:        int child = 2*parent+1;
6:        Object temp;
7:        while (child<size) {
8:           if(child<size-1&&compare(heap[child],heap[child+1]) >0)
9:              child++;
10:          if(compare(heap[parent],heap[child]) <=0)
11:             break;
12:          temp = heap[child];
13:          heap[child] = heap[parent];
14:          heap[parent] = temp;
15:          parent = child; child= 2*parent+1;
16:       } // while
17:     }
```

ตอนแรกให้ childเป็นลูกซ้ายไว้ก่อน

ห้ามเป็นตัวสุดท้าย เพราะจะไม่มีพี่น้องทางขวาให้เทียบ

ถ้าลูกขวาน้อยกว่า ก็ปรับ child ให้เป็นลูกขวา

รูป 8.23 โค้ดของ percolateDown

จากรูป 8.23 จะเห็นว่าหลักการก็เหมือนกับ percolateUp คือวนลูปสลับ child กับ parent เรื่อยๆ จนสลับไม่ได้ก็ออกจากลูป แต่จะต้องทำการเลือกลูกที่น้อยที่สุดมาเป็น child เสมอ ส่วนเรื่องเวลานั้น Worst case ก็คือเมื่อต้องเลื่อนถึงใบ ดังนั้นจึงเป็น log n ตามความสูงของต้นไม้ ส่วนกรณี Average case นั้น สมมติว่าตัวที่เติมเข้าไปนี่มีค่าอยู่ตรงกลางของค่าทั้งหมด ดังนั้นต้องมีสมาชิกของอาร์เรย์ครึ่งหนึ่งที่มากกว่ามัน แต่เพราะเป็นต้นไม้ ดังนั้นครึ่งที่มากกว่านี้จะต้องอยู่ที่ใบแน่ๆ การ percolatDown จึงต้องเลื่อนให้เกือบระดับของใบ ดังนั้นจึงเกือบเป็น worst case เลยทีเดียว ดังนั้นเวลาจึงเป็น log n ด้วย

แบบฝึกหัด

1. จงวาดรูปต้นไม้ ณ เวลาต่างๆที่โค้ดต่อไปนี้ถูกรัน

Heap myHeap = new Heap( );

myHeap.add (new Integer (60));

myHeap.add (new Integer (50));

myHeap.add (new Integer (40));

myHeap.add (new Integer (30));

myHeap.add (new Integer (20));

```
myHeap.add (new Integer (10));
myHeap.removeMin( );
myHeap.removeMin( );
myHeap.removeMin( );
```

ฮัฟแมนโค้ดดิ้ง(Huffman Coding)

ในส่วนนี้เราจะมาดูวิธีการประยุกต์เอาฮีปไปใช้ในการบีบอัดข้อมูลที่เรียกว่าฮัฟแมนโค้ดดิ้ง ก่อนอื่นเรามาดูว่าสิ่งที่เราต้องการคืออะไร สิ่งที่เราต้องการคือ จะบีบอัด(compress) ไฟล์ โดยไม่ให้เสียข้อมูลเลย ได้อย่างไร

ให้ M เป็นข้อมูลที่เราต้องการบีบอัด สมมติว่ามันมีขนาด 100,000 ตัวอักษร โดยประกอบไปด้วยตัวอักษร a ถึง e เท่านั้น เราให้ E เป็นข้อมูลที่บีบอัดเสร็จแล้ว

โดยปกติแล้ว พื้นที่ในการเก็บตัวอักษร 1 ตัว ใช้จำนวนบิต 16 บิต ดังนั้นถ้าเรามี 100,000 ตัว ก็ต้องใช้ 16,000,000 บิต เราลองมาหาวิธีลดจำนวนบิตกันดู ก่อนอื่น ในเมื่อเรารู้ว่ามีแค่ a ถึง e เท่านั้น เราก็สามารถกำหนดโค้ดสำหรับ a จนถึง e ได้ดังนี้

a = 000

b = 001

c = 010

d = 011

e = 100

ถ้าใช้เพียงแค่สามบิตแบบนี้ หนึ่งแสนตัวอักษรก็จะใช้สามแสนบิต เรายังสามารถลดจำนวนบิตได้อีก เช่นให้

a = 0

b =1

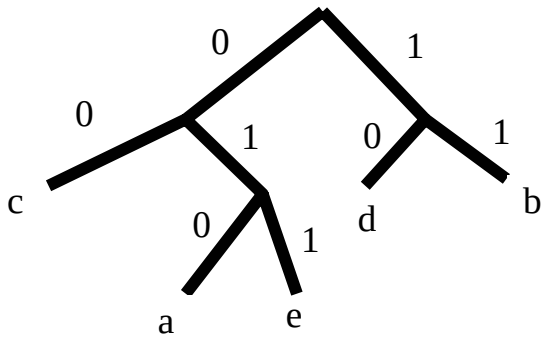
c = 00

d = 01

e = 10

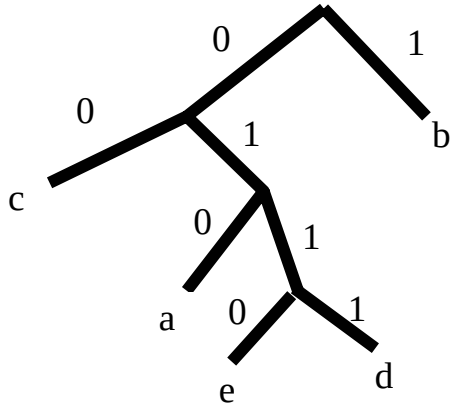
จำนวนบิตก็จะน้อยกว่าสามแสนเสียอีก แต่ว่า 001010 อาจเป็นได้ทั้ง aababa หรือ cbda ก็ได้ ดังนั้นถึงสามารถบีบอัดได้ แต่ก็ไม่มีทางตีความกลับคืนได้ เพราะความหมายไม่แน่นอน

วิธีแก้ปัญหาความหมายที่กำกวมนี้ ทำได้โดยใช้ prefix-free encoding โดยวิธีทำคือ สร้างต้นไม้ทวิภาค ให้กิ่งซ้ายถือเป็นเลข 0 กิ่งขวาเป็นเลข 1 แล้วเราก็ให้ตัวอักษรแต่ละตัวอยู่ที่ใบ ดังนั้นจะไม่มีตัวอักษรสองตัวที่อยู่บนเส้นทางเดียวกันจากรากแน่นอน ต้นไม้ตัวอย่างอยู่ในรูป 8.24



รูป 8.24 ต้นไม้ตัวอย่างการสร้าง prefix-free encoding

จากรูป 8.24 เราได้ a = 010, b = 11, c = 00, d = 10, e = 011 ซึ่งคราวนี้จะไม่มีความกำกวมแล้ว เพียงแต่ว่า ต้นไม้ที่แทน a ถึง e นี้ เป็นไปได้หลายรูปแบบ รูป 8.25 แสดงต้นไม้ที่ใช้กำหนดบิตของ a ถึง e อีกต้นหนึ่ง



รูป 8.25 ต้นไม้ที่สร้างบิตที่ไม่กำกวมของ a ถึง e อีกแบบหนึ่ง

แล้วแบบต้นไม้จากรูป 8.24 กับ 8.25 ต้นไหนจะดีกว่ากัน แนวคิดเพื่อการประหยัดที่ก็คือ ตัวอักษรที่แทนด้วยจำนวนบิตเยอะๆ ไม่ควรจะมีในข้อความของเรามาก ส่วนตัวอักษรที่เกิดขึ้นบ่อยในข้อความของเราก็พยายามให้จำนวนบิตน้อยๆ ดังนั้นเราต้องสร้างต้นไม้โดยดูจากความถี่ของตัวอักษรในข้อความที่จะบีบอัด ต้นไม้ที่สร้างจากแนวคิดนี้เรียกว่าต้นไม้ฮัฟแมน(Huffman Tree)

การสร้างต้นไม้ฮัฟแมน

สมมติในข้อความที่เราจะบีบอัดมี

a 5,000 ตัว

b 10,000 ตัว

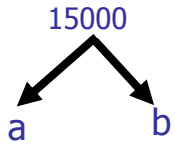
c 20,000 ตัว

d 31,000 ตัว

e 34,000 ตัว

เราสามารถสร้างต้นไม้ฮัฟแมนได้โดยเอาไพรออริตีคิวมาช่วย โดยเอาคู่ (ตัวอักษร,ความถี่) ใส่ไพรออริตี้คิว ให้ถือว่าความถี่ต่ำสำคัญที่สุด ตอนแรก คิวของเราจะเป็น (a:5000) (b: 10000) (c: 20000) (d: 31000) (e: 34000)

ให้เอาสองตัวที่น้อยสุดออกจากคิว ตัวแรกเอามาเป็นกิ่งซ้าย ตัวที่สองมาเป็นกิ่งขวา ส่วนผลบวกของความถี่เราเอามาเป็นราก (รูป 8.26)

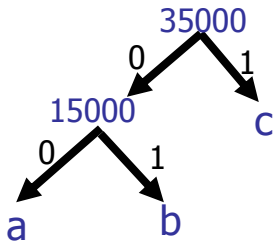


รูป 8.26 สร้างต้นไม้ฮัฟแมน (1)

จากนั้นก็เอาผลใส่กลับลงในคิว ซึ่งจะกลายเป็น (:15000) (c: 20000) (d: 31000) (e: 34000)

สำหรับ (: 15000) ที่ใส่กลับเข้าไปนั้น มี tree ของมันเป็นสิ่งที่อยู่หน้าตัวเครื่องหมาย :

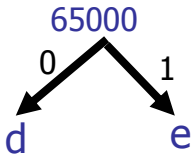
ต่อไปก็ เอาสองตัวที่น้อยสุดออกจากคิว ทำเหมือนเดิม (รูป 8.27)



รูป 8.27 สร้างต้นไม้ฮัฟแมน (2)

เมื่อเอาต้นไม้ในรูป 8.27 กลับเข้าไพรออริตี้คิว ตัวคิวจะกลายเป็น (d: 31000) (e: 34000) (: 35000)

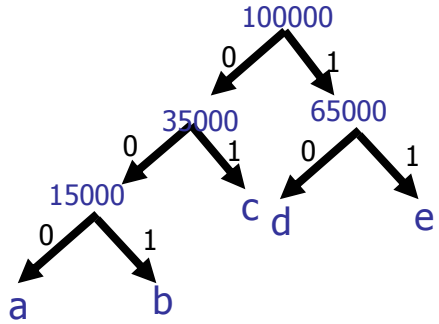
ต่อไปก็ยังเอาสองตัวที่น้อยที่สุดออกจากคิวไปเป็นกิ่งซ้ายและกิ่งขวาเหมือนเดิม (รูป 8.28)



รูป 8.28 สร้างต้นไม้ฮัฟแมน (3)

ในรูป 8.28 เราสร้างต้นไม้อีกต้นแล้ว ไม่ใช่ต้นเดิม เมื่อเอาต้นไม้นี้กลับเข้าไพรออริตี้คิว ตัวคิวจะกลายเป็น (: 35000) (: 65000)

เราก็ยังทำเหมือนเดิม คือ เอาสองตัวที่ความถี่น้อยที่สุดออกจากคิวมาสร้างเป็นกิ่งซ้ายและขวา นี่เป็นสองตัวสุดท้าย ดังนั้นเราได้ต้นไม้ฮัฟแมนตามที่ต้องการแล้ว (รูป 8.29)



รูป 8.29 สร้างต้นไม้ฮัฟแมน (ผลลัพธ์)

แบบฝึกหัด

1. จงสร้างต้นไม้ฮัฟแมนของตัวอักษรต่อไปนี้ (ที่ให้ไปเป็นตัวอักษร และความถี่ในข้อความ) และจงบอกโค้ดของตัวอักษรแต่ละตัว

‘a’ 700

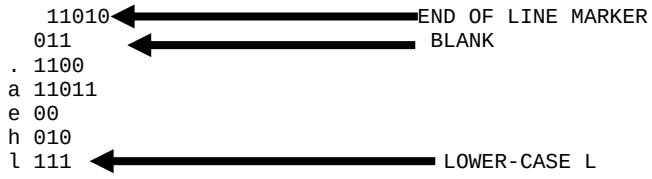
‘b’ 400

‘c’ 100

‘d’ 300

‘e’ 500

2. ถ้าเราสร้างฮีปของจำนวนเต็ม 16, 15, 8, 9, 4, 5, 3, 11, 10 ซึ่งใส่ฮีปเรียงตามลำดับ ถามว่าฮีปผลลัพธ์สุดท้ายจะมีรูปร่างอย่างไร
3. ถ้ามีฮีปอยู่สองฮีป จงเขียนโค้ดเพื่อรวมฮีปสองฮีปนี้เข้าด้วยกัน
4. จงเขียนโค้ดเพื่อเปลี่ยน min heap เป็น max heap โดยห้ามใช้การ percolate โดยเด็ดขาด
5. จงไปค้นคว้าและอธิบายวิธีการทำ heap sort
6. จงใช้ไพรออริตี้คิวทำสแตก อนุญาตให้ใช้ตัวแปรเพิ่มได้หนึ่งตัวเท่านั้น
7. จงใช้ไพรออริตี้คิวทำคิวธรรมดา อนุญาตให้ใช้ตัวแปรเพิ่มได้หนึ่งตัวเท่านั้น
8. จงเขียนเมธอด findGreaterThan(int k) ซึ่งพิมพ์ตัวเลขทุกตัวในฮีปที่มากกว่า k ออกมา เมธอดนี้มี big O เท่าไร
9. จงแปลงข้อความ “We have a new enemy, Luke Skywalker. The force is strong within him. He will join us, or die, my master.” ให้เป็นโค้ด โดยใช้การสร้างต้นไม้ฮัฟแมน
10. จงเขียนโค้ดเพื่อสร้าง ต้นไม้ฮัฟแมน จากหลักการที่เรียนไปในบทนี้
11. สมมติว่ามีรหัสแทนด้วยบิตต่างๆดังนี้



และมีข้อความดังนี้

10010000111000111111100111000110110111001000111111110110011010

จงหาว่านี่คือข้อความว่าอะไร