# บทที่ 7

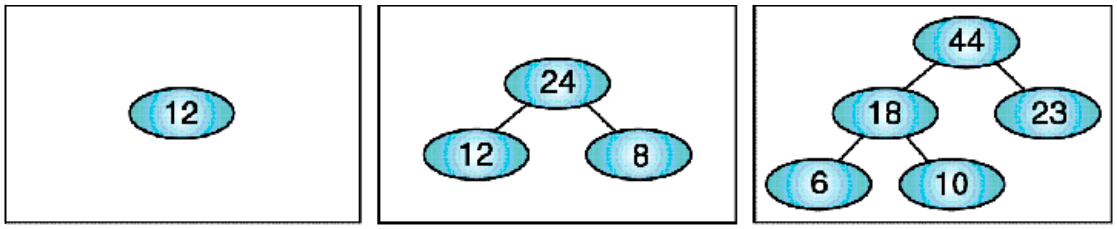
# ฮีป (Heaps) และ คิวที่มีลำดับความสำคัญ (Priority Queues)

คิวที่มีลำดับความสำคัญ (Priority Queue) เป็นคิวประเภทหนึ่งที่การให้บริการผู้ที่อยู่ในคิวไม่ได้ใช้หลักการมาก่อนรับบริการก่อน (First Come, First Serve หรือ First In, First Out — FIFO) แต่จะให้บริการผู้ที่มีความสำคัญสูงสุด (highest-priority) ซึ่งการพิจารณาความสำคัญของสมาชิกที่อยู่ในคิว ขึ้นอยู่กับระบบงานที่นำเอาคิวประเภทนี้ไปใช้ เช่น คิวของผู้รอรับการรักษาในห้องอุบัติเหตุ (emergency room) ของโรงพยาบาล หมอจะต้องให้การรักษาผู้มีอาการหนักที่สุดก่อน ซึ่งถือว่าเป็นผู้ที่มีความสำคัญมากที่สุดในคิว โดยใช้ขั้นตอนวิธีแบบฮีป (heap)

## 7.1 ฮีป (Heaps)

**นิยาม7.1** ฮีป เป็นต้นไม้ทวิภาคเกือบสมบูรณ์ (nearly complete binary tree) ที่มีคุณสมบัติว่า ค่าของข้อมูลที่เก็บที่โหนด i ใด ๆ ต้องมีค่ามากกว่าหรือเท่ากับค่าข้อมูลที่เก็บอยู่ที่โหนดลูกทางซ้าย และทางขวาของโหนด i เสมอ

**ตัวอย่าง 7.1** ตัวอย่างแสดงต้นไม้ทวิภาคเกือบสมบูรณ์ ที่เป็นฮีป




(a) (b) (c)

รูปที่ 7.1 ฮีปที่มีความสูงเป็น 0, 1, และ 2 ตามลำดับ

## 7.2 การจัดเก็บฮีปในโครงสร้างอาเรย์

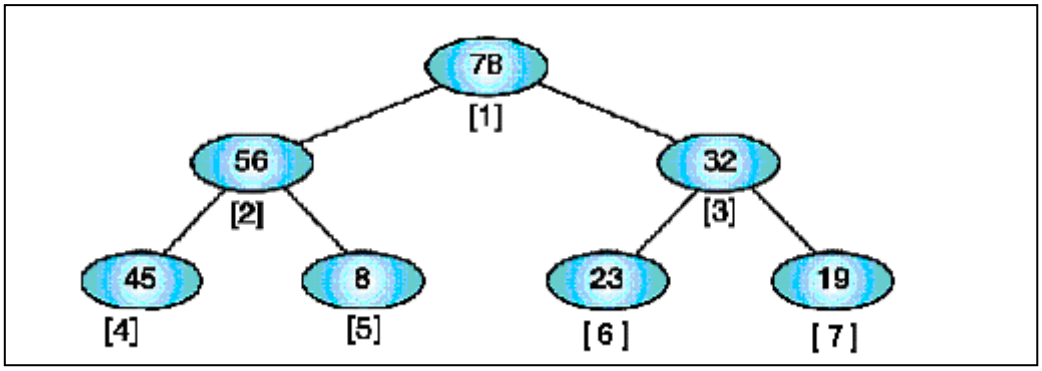
เนื่องจากฮีปเป็นต้นไม้ทวิภาค แต่มีคุณลักษณะพิเศษที่ทำให้สามารถนำไปจัดเก็บในโครงสร้างอาเรย์ ณ ตำแหน่งเฉพาะของแต่ละสมาชิกได้ ดังนี้

เมื่อ i เป็น ดัชนีที่บอกตำแหน่งของข้อมูลในอาเรย์ จะได้ว่า

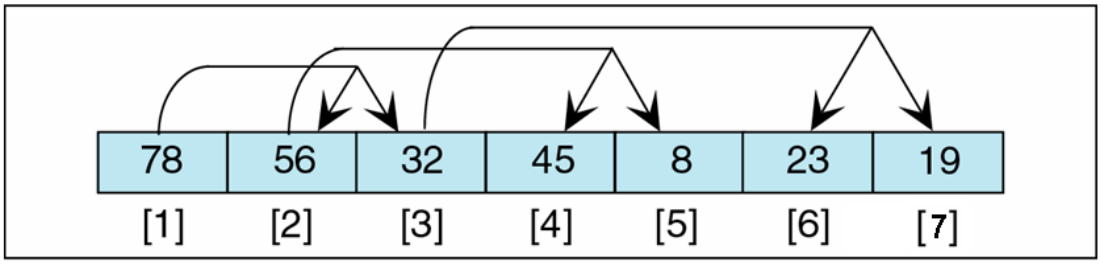
1. *Parent (i)* = $\frac{i}{2}$ *(พ่อของโหนด i)*
2. *Left (i)* = $2i$ *(ลูกทางซ้ายของโหนด i)*
3. *Right (i)* = $2i + 1$ *(ลูกทางขวาของโหนด i)*

**หมายเหตุ** โปรแกรมภาษาจาวาที่เราเขียน เริ่มเก็บข้อมูลในอาเรย์ตำแหน่งที่ 1

**ตัวอย่าง 7.2** แสดงการนำสมาชิกของฮีปในรูปที่ 7.2 ไปเก็บในอาเรย์ จะได้ดังรูปที่ 7.3




รูปที่ 7.2 ฮีปที่มีความสูงเป็น 2




รูปที่ 7.3 อาเรย์ที่จัดเก็บข้อมูลในฮีปของรูปที่ 7.2

ถึงแม้จะใช้อาเรย์ในการจัดการกับฮีป แต่เนื่องจากฮีปเป็นต้นไม้ทวิภาค จึงเรียกแต่ละสมาชิกของฮีปว่า โหนด

## 7.3 การดำเนินการกับฮีป

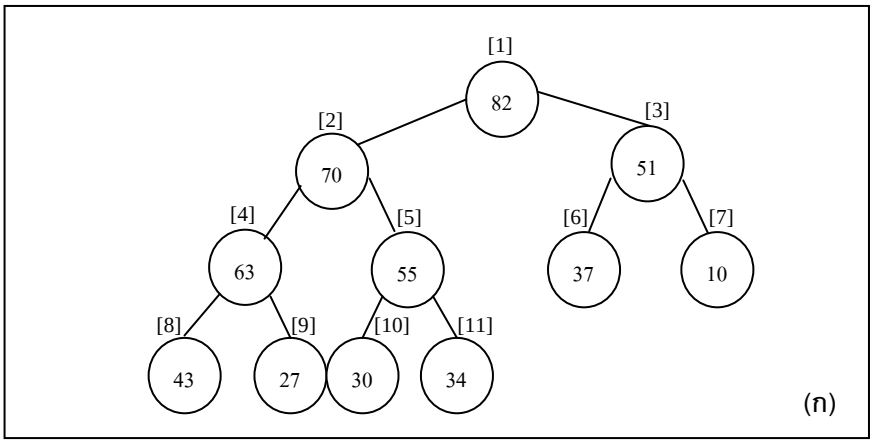
### 7.3.1 การเพิ่มโหนดใหม่

การเพิ่มโหนดสามารถทำได้ดังนี้

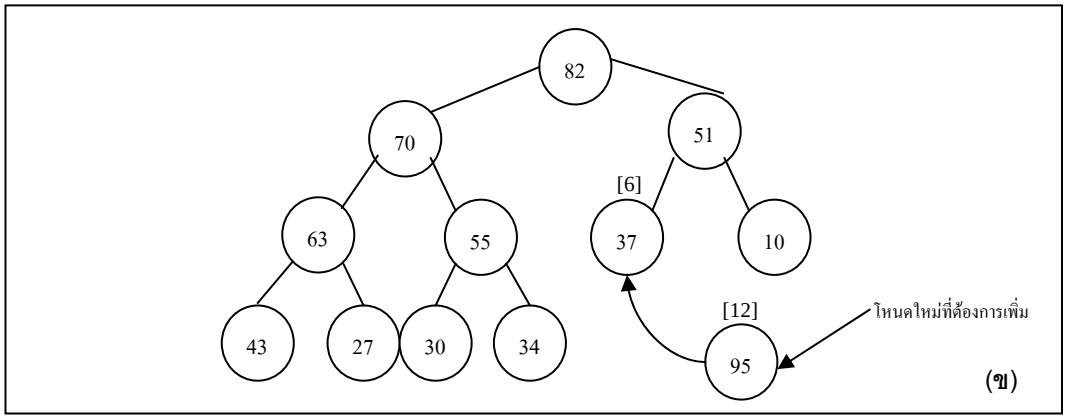
1. เพิ่มสมาชิกใหม่เข้าไปที่ตำแหน่งต่อจากข้อมูลตัวสุดท้ายของอาเรย์
2. ทำการปรับต้นไม้ให้มีคุณสมบัติเป็นฮีป โดยทำการปรับตั้งแต่โหนดที่เพิ่มใหม่กับโหนดพ่อแม่ ถ้าข้อมูลที่โหนดพ่อแม่มีค่าน้อยกว่าโหนดใหม่ให้สลับค่าข้อมูล แล้วทำการเปรียบเทียบกับโหนดพ่อแม่ในลำดับถัดขึ้นไปเรื่อยๆ และสลับที่ข้อมูลจนกระทั่งถึงรากของต้นไม้

**ตัวอย่าง 7.3** แสดงการเพิ่มโหนดใหม่ที่มีค่าข้อมูลเป็น 95 เข้าไปในฮีปในรูปที่ 7.4 (ก) และทำการปรับฮีป (Heapify) ด้วย method trickleUp ( ) ตามลำดับ

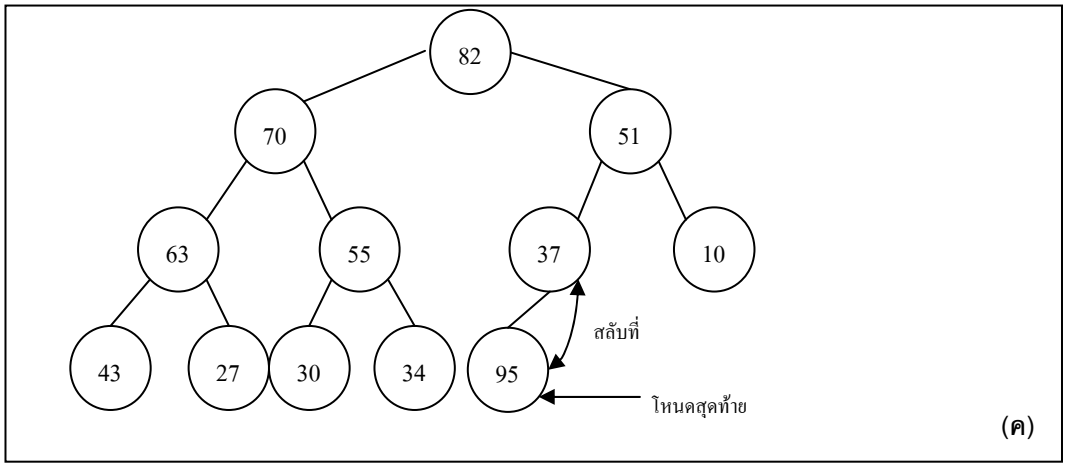
วิธีทำ H = {82,70,51,63,55,37,10,43,27,30,34}



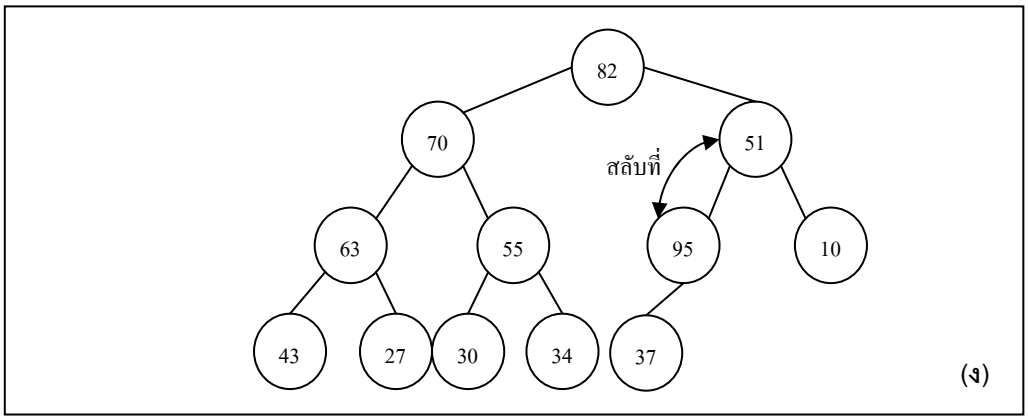

1. เพิ่มโหนดที่ 12 ให้ H[12] = 95



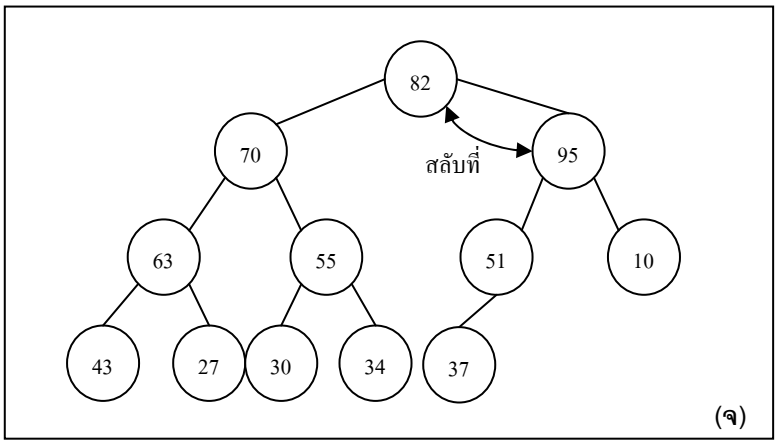

2. เปรียบเทียบ H[12] กับโหนดพ่อ H[6] ถ้า H[12] > H[6] ให้สลับค่ากัน



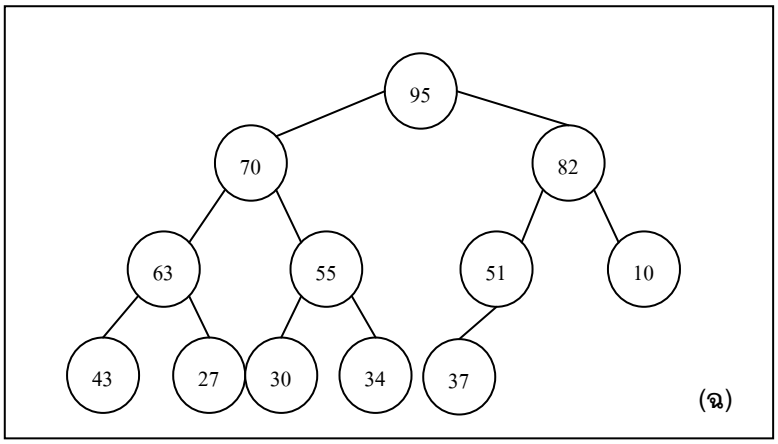
3. เปรียบเทียบ H[6] กับโหนดพ่อ H[3] ถ้า H[6] > H[3] ให้สลับค่ากัน




4. เปรียบเทียบ H[3] กับโหนดพ่อ H[1] ถ้า H[3] > H[1] ให้สลับค่ากัน




5. เปรียบเทียบไปเรื่อยจนถึงโหนดราก (H[1]) จะได้ว่า H[1] มีค่ามากสุด




รูปที่ 7.4 แสดงการเพิ่มโหนดใหม่เข้าไปในฮีป

### 7.3.2 การลบโหนดออกจากฮีป

การลบโหนดออกจากฮีป เป็นการนำสมาชิกที่มีค่าข้อมูลมากที่สุดออกจากฮีป ซึ่งโหนดที่มีค่าข้อมูลมากที่สุดจะอยู่ที่รากของฮีป (ต้นไม้) เสมอ

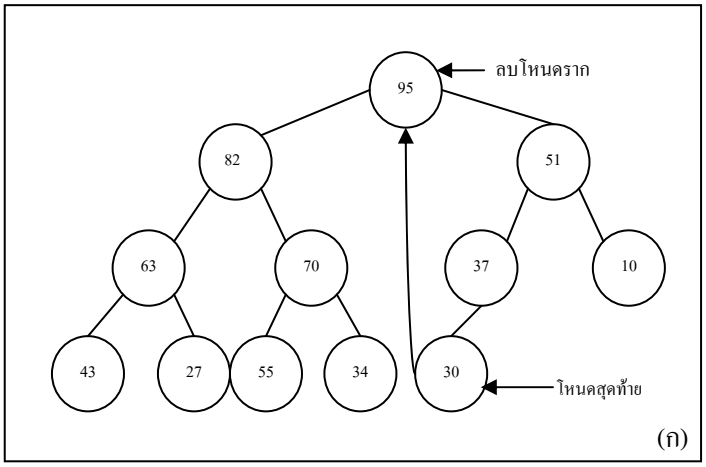
ให้ n เป็นจำนวนสมาชิกทั้งหมดในอาเรย์ heapArray ที่ใช้เก็บข้อมูลในฮีป

ขั้นตอนในการลบโหนดที่มีค่าข้อมูลมากที่สุด เป็นดังนี้

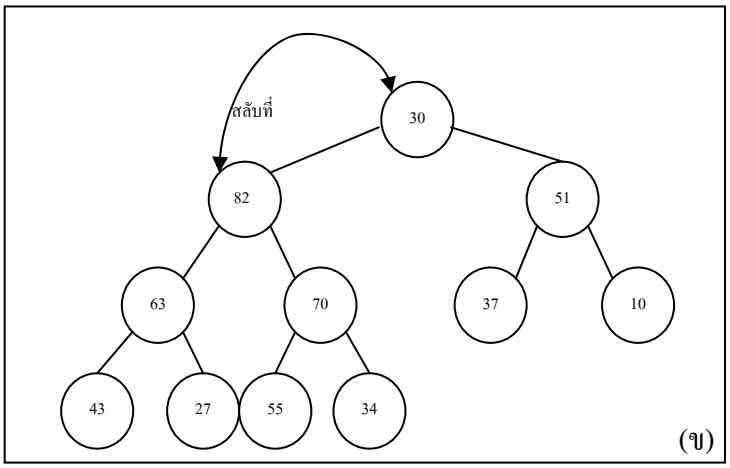
1. ลบข้อมูลออกจากรากของต้นไม้ ( maxNode = heapArray [1])
2. นำข้อมูลจากโหนดสุดท้ายไปเก็บไว้ที่ราก ( แทนที่ข้อมูลที่ลบออกไป) heapArray[1] = heapArray [n]
3. ทำการปรับต้นไม้ โดยสลับข้อมูลที่โหนดรากกับโหนดลูกไปเรื่อยๆ ด้วย method trickledown( ) จนกระทั่งต้นไม้ผลลัพธ์เป็นฮีป นั่นคือ (ค่าข้อมูลที่โหนด i ใดๆ จะมีค่ามากกว่าเท่ากับข้อมูลที่โหนดลูกทางซ้ายและโหนดลูกทางขวา)

**ตัวอย่าง 7.4** แสดงการลบโหนดที่มีค่ามากที่สุดออกจากฮีป และทำการปรับฮีป (heapify) โดย method trickledown( ) ตามรูป 7.5

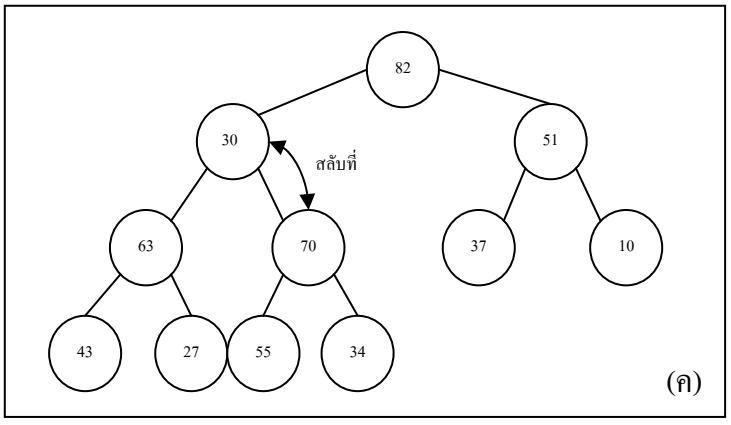
<u>วิธีทำ</u> 1. ลบข้อมูลออกจากรากของต้นไม้ แล้วนำข้อมูลจากโหนดสุดท้ายไปเก็บไว้ที่ราก คือ ให้ H[1] = H[12]



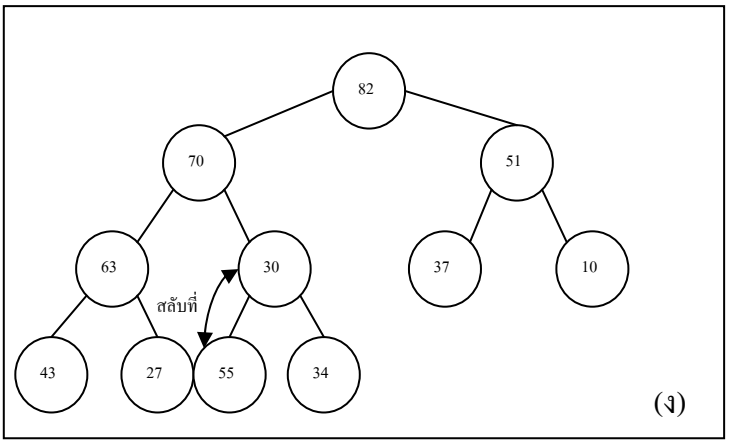
2. เปรียบเทียบลูกซ้ายหรือลูกขวาของ H[1] ว่าโหนดใดมีค่ามากกว่ากัน H[2] > H[3] ให้สลับค่า H[1] กับ H[2]



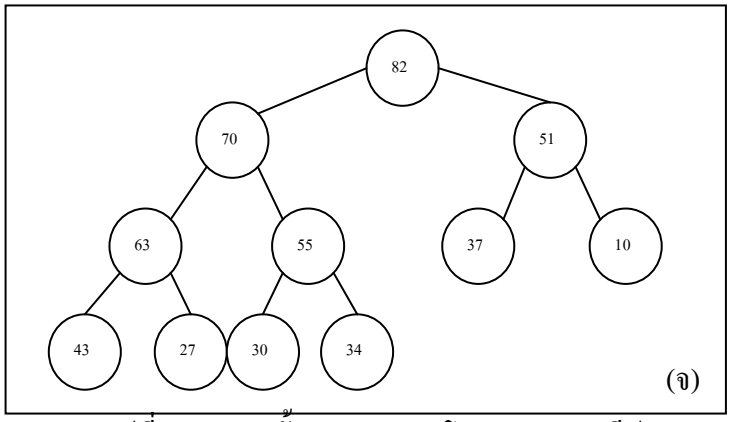

3. เปรียบเทียบลูกซ้ายหรือลูกขวาของ H[2] ว่าโหนดใดมีค่ามากกว่ากัน H[5] > H[4] ให้สลับค่า H[2] กับ H[5]




4. เปรียบเทียบลูกซ้ายหรือลูกขวาของ H[5] ว่าโหนดใดมีค่ามากกว่ากัน H[10] > H[11] ให้สลับค่า H[5] กับ H[11]



5. ทำไปเรื่อยๆ จนถึงโหนดที่ไม่มีลูกแล้ว จนได้ heap ที่สมบูรณ์

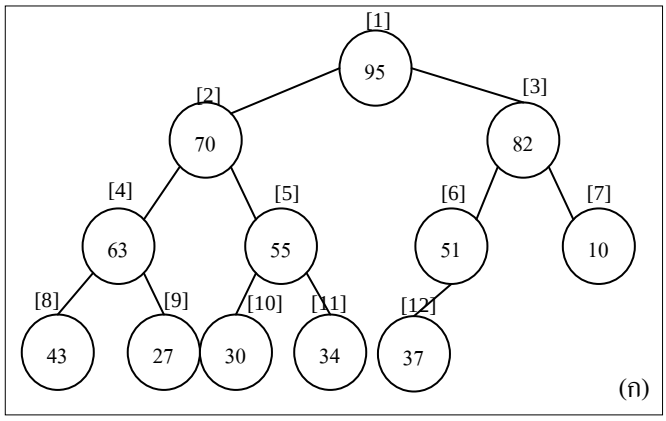



รูปที่ 7.5 แสดงขั้นตอนการลบโหนดออกจากฮีป

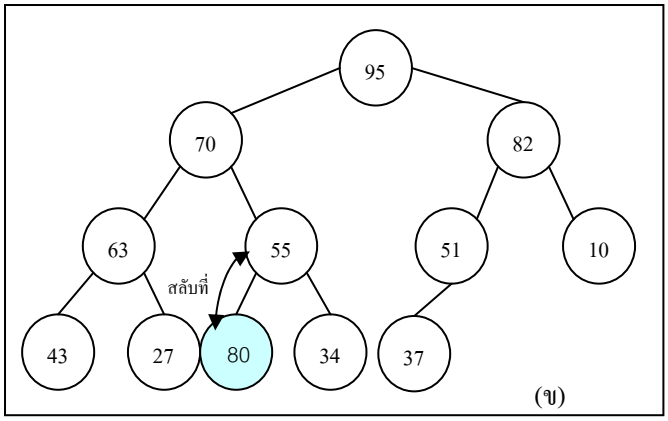
### 7.3.3 การเปลี่ยนแปลงค่า key

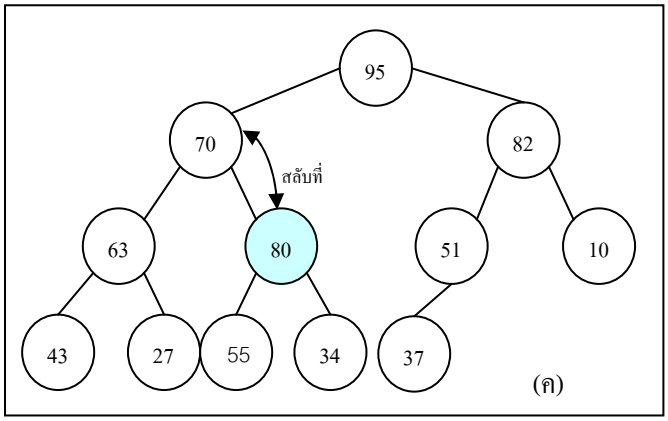
ถ้าค่าที่เปลี่ยนใหม่มีค่ามากกว่าค่าเดิมก็ให้ทำการ trickleUp( ) แต่ถ้าค่าที่เปลี่ยนใหม่มีค่าน้อยกว่าค่าเดิมก็ให้ทำการ trickleDown( )

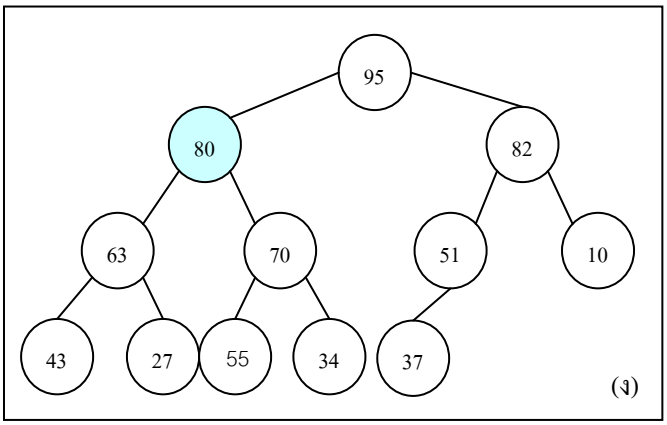
**ตัวอย่าง 7.5** ค่าใน heap เดิม คือ H={95,70,62,63,55,51,10,43,27,30,34,37} ต้องการเปลี่ยนค่า H[10] จาก 30 เป็น 80




วิธีทำ ให้ H[10] = 80 และ เนื่องจากค่าที่เปลี่ยนใหม่มีค่ามากกว่าค่าเดิม เรียก method trickle Up( ) จนได้คุณสมบัติของ heap ตามรูป 7.6




<u>รูปที่ 7.6</u> แสดงการเปลี่ยนค่า key โหนดที่ 10 จาก 30 เป็น 80

**โปรแกรม 7.1** Heap.java แสดงการสร้างคลาส Heap และ การดำเนินการกับ Heap

```
// heap.java
// demonstrates heaps
// to run this program: C>java HeapApp
import java.io.*;               // for I/O
import java.lang.Integer;       // for parseInt()
////////////////////////////////////////////////////////////////
class Node
  {
  public int iData;             // data item (key)
  public Node(int key)          // constructor
    { iData = key; }
  }  // end class Node
////////////////////////////////////////////////////////////////
class Heap
  {
  private Node[] heapArray;
  private int maxSize;          // size of array
  private int currentSize;      // number of nodes in array
// -------------------------------------------------------------
  public Heap(int mx)           // constructor
    {
    maxSize = mx;
    currentSize = 1;
    heapArray = new Node[maxSize];  // create array
    }
// -------------------------------------------------------------
  public boolean isEmpty()
    { return currentSize==1; }
// -------------------------------------------------------------
  public boolean insert(int key)
    {
    if(currentSize==maxSize)
      return false;
    Node newNode = new Node(key);
    heapArray[currentSize] = newNode;
    trickleUp(currentSize++);
    return true;
    }  // end insert()
// -------------------------------------------------------------
  public void trickleUp(int index)
    {
    int parent = (index) / 2;
    Node bottom = heapArray[index];
```

```
    while( index > 1 &&  heapArray[parent].iData < bottom.iData )
      {
            heapArray[index] = heapArray[parent];  // move it down
             index = parent;
             parent = (parent) / 2;
      }  // end while
    heapArray[index] = bottom;
    }  // end trickleUp()
// -------------------------------------------------------------
  public Node remove()          // delete item with max key
    {                           // (assumes non-empty list)
    Node root = heapArray[1];
    heapArray[1] = heapArray[--currentSize];
    trickleDown(1);
    return root;
    }  // end remove()
// -------------------------------------------------------------
  public void trickleDown(int index)
    {
    int largerChild;
    Node top = heapArray[index];       // save root
    while(index <= currentSize/2)      // while node has at
      {                                //    least one child,
      int leftChild = 2*index;
      int rightChild = leftChild+1;
                                       // find larger child
      if(rightChild < currentSize &&  // (rightChild exists?)
          heapArray[leftChild].iData <   heapArray[rightChild].iData)
        { largerChild = rightChild;  }
      else
        { largerChild = leftChild; }
      If  ( top.iData >= heapArray[largerChild].iData)          // top >= largerChild?
           break;                 // shift child up
      heapArray[index] = heapArray[largerChild];
      index = largerChild;           // go down
      }  // end while
    heapArray[index] = top;          // root to index
    }  // end trickleDown()
// -------------------------------------------------------------
  public boolean change(int index, int newValue)
    {
      if(index<1 || index>=currentSize)
```

```
            return false;
         int oldValue = heapArray[index].iData;  // remember old
         heapArray[index].iData = newValue;  // change to new
        if(oldValue < newValue)             // if raised,
            trickleUp(index);               // trickle it up
        else                                // if lowered,
            trickleDown(index);             // trickle it down
      return true;
      }  // end change()
// -------------------------------------------------------------
   public void displayHeap()
      {
      System.out.print("heapArray: ");    // array format
      for(int m=1; m<currentSize; m++)
        if(heapArray[m] != null)
          System.out.print( heapArray[m].iData + " ");
        else
          System.out.print( "-- ");
      System.out.println();
      // heap format
      int nBlanks = 32;
      int itemsPerRow = 1;
      int column = 0;
      int j = 1;                        // current item
      String dots = "................................";
      System.out.println(dots+dots);      // dotted top line

      while(currentSize > 1)              // for each heap item
        {
          if(column == 0)                 // first item in row?
             for(int k=0; k<nBlanks; k++)  // preceding blanks
                System.out.print(' ');
                                  // display item
           System.out.print(heapArray[j].iData);
          if(++j == currentSize)          // done?
               break;
          if(++column==itemsPerRow)       // end of row?
           {
             nBlanks /= 2;                // half the blanks
             itemsPerRow *= 2;            // twice the items
             column = 0;                  // start over on
             System.out.println();        //    new row
           }
```

```
            else                        // next item on row
              for(int k=0; k<nBlanks*2-2; k++)
                System.out.print(' ');     // interim blanks
          }  // end while
        System.out.println("\n"+dots+dots); // dotted bottom line
      }  // end displayHeap()
  // --------------------------------------------------------------
  }  // end class Heap
////////////////////////////////////////////////////////////////
class HeapApp
  {
        public static int h[] = {82,70,51,63,55,37,10,43,27,30,34};
        public static void main(String[] args) throws IOException
       {
            int value, value2, i;
            Heap theHeap = new Heap(31);  // make a Heap; max size 31
            boolean success;
          for(i=0;i<11;++i){
             theHeap.insert(h[i]);   }        // insert 10 items
          while(true)                  // until [Ctrl]-[C]
          {
             putText("Enter first letter of ");
             putText("show, insert, remove, change: ");
            int choice = getChar();
            switch(choice)
              {
                case 's':                     // show
                   theHeap.displayHeap();
                   break;
                case 'i':                     // insert
                   putText("Enter value to insert: ");
                  value = getInt();
                  success = theHeap.insert(value);
                  if( !success )
                      putText("Can't insert; heap is full" + '\n');
                  break;
               case 'r':                     // remove
                  if( !theHeap.isEmpty() )
                        theHeap.remove();
                   else
                        putText("Can't remove; heap is empty" + '\n');
                   break;
              case 'c':                     // change
```

```
                    putText("Enter index of item: ");
                    value = getInt();
                    putText("Enter new priority: ");
                    value2 = getInt();
                    success = theHeap.change(value, value2);
                    if( !success )
                        putText("Can't change; invalid index" + '\n');
                    break;
                default:
                    putText("Invalid entry\n");
            }  // end switch
        }  // end while
    }  // end main()
// --------------------------------------------------------------
  public static void putText(String s)
    {
        System.out.print(s);
        System.out.flush();
    }
//--------------------------------------------------------------
  public static String getString() throws IOException
    {
    InputStreamReader isr = new InputStreamReader(System.in);
    BufferedReader br = new BufferedReader(isr);
    String s = br.readLine();
    return s;
    }
//--------------------------------------------------------------
  public static char getChar() throws IOException
    {
    String s = getString();
    return s.charAt(0);
    }
//--------------------------------------------------------------
  public static int getInt() throws IOException
    {
    String s = getString();
    return Integer.parseInt(s);
    }
//--------------------------------------------------------------
 }  // end class HeapApp
```

**Outputs**

```
Enter first letter of show, insert, remove, change: s
heapArray: 82 70 51 63 55 37 10 43 27 30 34
.....................................................................
                        82
            70                      51
      63          55          37          10
  43      27      30      34
.....................................................................
Enter first letter of show, insert, remove, change: i
Enter value to insert: 95
Enter first letter of show, insert, remove, change: s
heapArray: 95 70 82 63 55 51 10 43 27 30 34 37
.....................................................................
                        95
            70                      82
      63          55          51          10
  43      27      30      34      37
.....................................................................
Enter first letter of show, insert, remove, change: r
Enter first letter of show, insert, remove, change: s
heapArray: 82 70 51 63 55 37 10 43 27 30 34
.....................................................................
                        82
            70                      51
      63          55          37          10
  43      27      30      34
.....................................................................
Enter first letter of show, insert, remove, change: i
Enter value to insert: 95
Enter first letter of show, insert, remove, change: s
heapArray: 95 70 82 63 55 51 10 43 27 30 34 37
.....................................................................
                        95
            70                      82
      63          55          51          10
  43      27      30      34      37
.....................................................................
Enter first letter of show, insert, remove, change: c
Enter index of item: 10
Enter new priority: 80
Enter first letter of show, insert, remove, change: s
heapArray: 95 80 82 63 70 51 10 43 27 55 34 37
.....................................................................
                        95
            80                      82
      63          70          51          10
  43      27      55      34      37
.....................................................................
```

## 7.4 การเรียงลำดับแบบฮีป (Heap Sort)

โครงสร้างข้อมูลฮีปเป็นโครงสร้างข้อมูลที่สามารถนำมาใช้ในการเรียงลำดับข้อมูลได้อย่างมีประสิทธิภาพ เราเรียกการเรียงลำดับโดยใช้โครงสร้างข้อมูลฮีปว่า การเรียงลำดับแบบฮีป

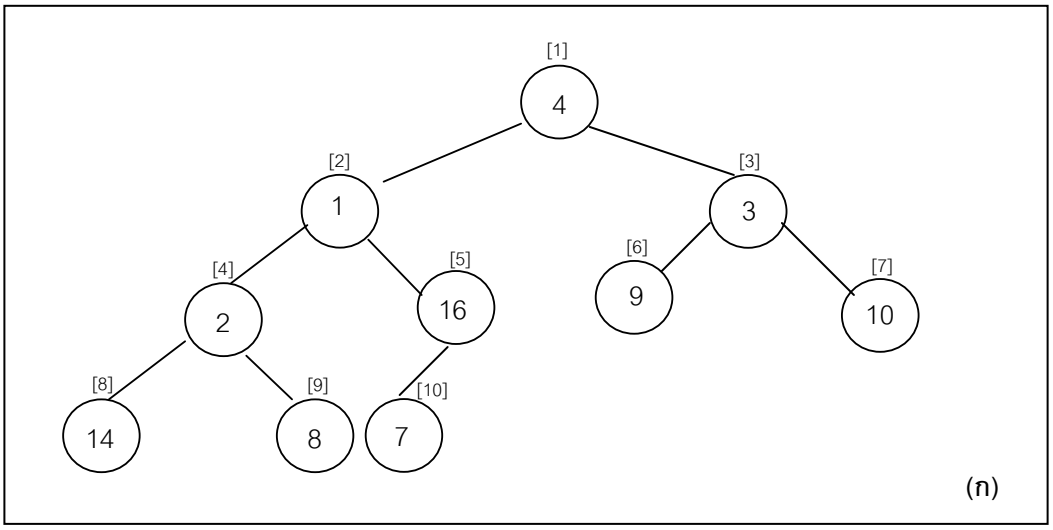
การเรียงลำดับแบบฮีปดำเนินการโดยเพิ่มข้อมูลที่ยังไม่ได้เรียงลำดับเข้าไปในอาเรย์ของฮีปทีละตัว ด้วยเมท็อด insertAt() หลังจากนั้นใช้กระบวนการ trickledown( ) เพื่อจัดให้เป็นไปตามคุณสมบัติของ heap จากนั้นนำข้อมูลที่มีค่ามากที่สุดออกจากฮีปทีละตัว ด้วยเมท็อด remove( )โดยนำสมาชิกที่ดึงออกตัวแรกไปเก็บไว้ที่ตำแหน่งในอาเรย์ที่มีดัชนีสูงสุด นำสมาชิกที่ดึงออกตัวที่สองไปไว้ที่ตำแหน่งในอาเรย์ที่มีดัชนีสูงสุดรองลงมา ดำเนินเช่นนี้ไปเรื่อย ๆ จนกระทั่งฮีปเป็นฮีปว่าง จะได้ข้อมูลในอาเรย์เรียงลำดับจากน้อยไปหามากตามต้องการ

**ตัวอย่าง 7.6** กำหนดอาเรย์ A ดังนี้

| index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| A | 4 | 1 | 3 | 2 | 16 | 9 | 10 | 14 | 8 | 7 |

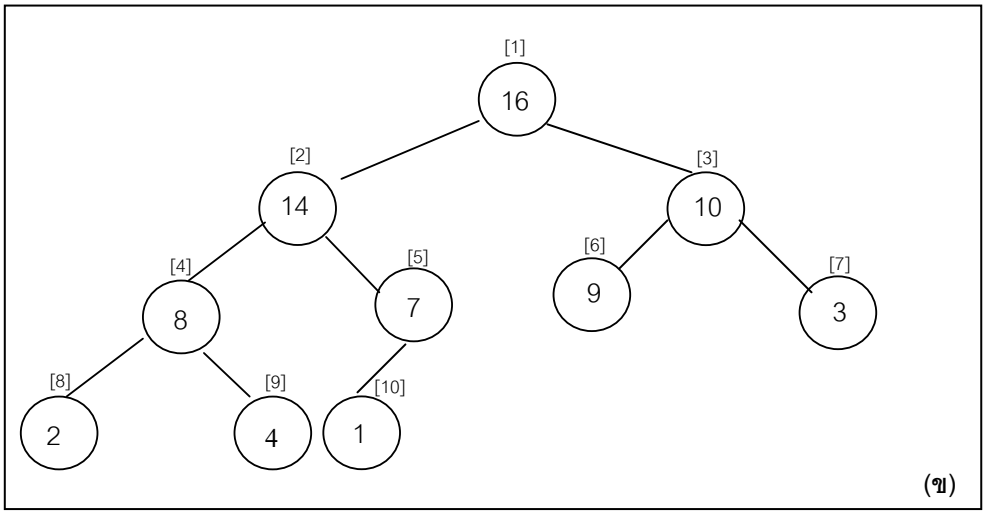
<u>วิธีทำ</u>

1. เรียกเมท็อด insertAt() ในการเพิ่มสมาชิกของอาเรย์เข้าไปในฮีป จะได้ผลลัพธ์ ดังรูปที่ 7.7 (ก)



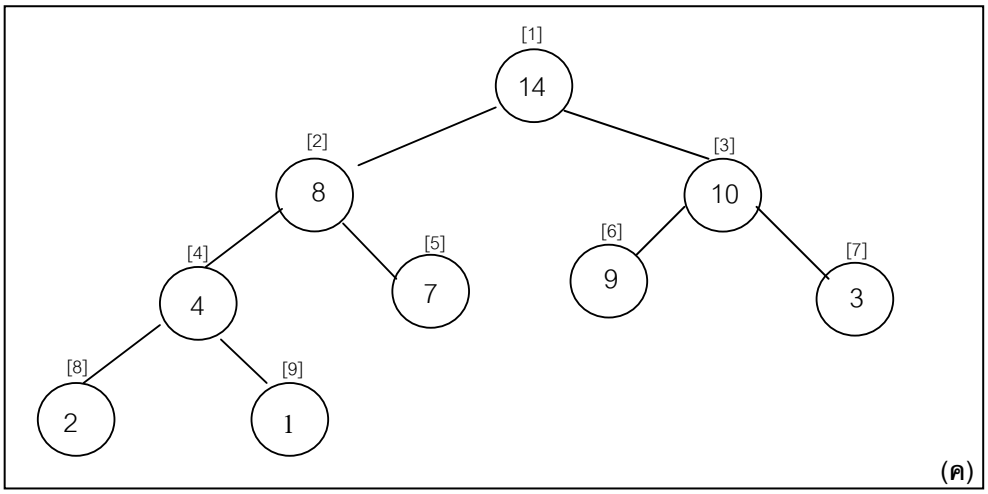
2. เรียก method trickledown ( ) เพื่อปรับให้ได้คุณสมบัติของ heap

| index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| A | 16 | 14 | 10 | 8 | 7 | 9 | 3 | 2 | 4 | 1 |



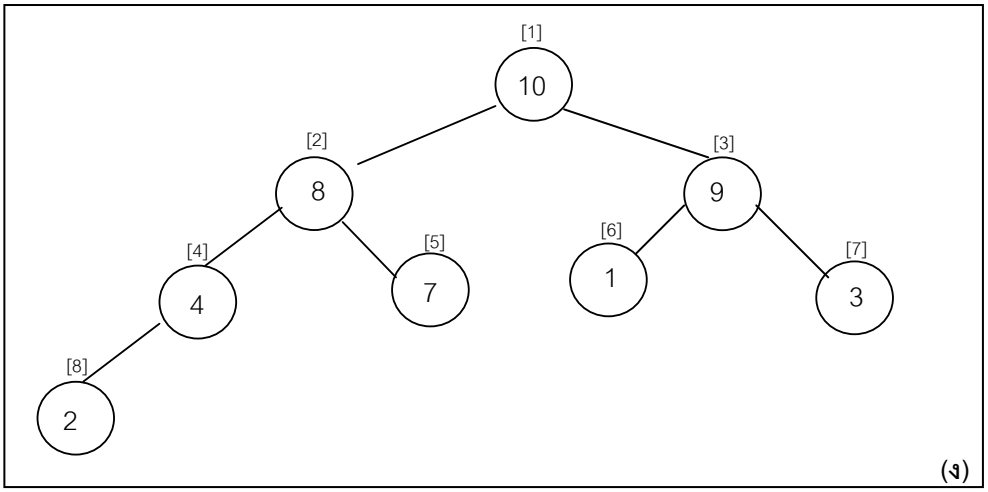

3. ดึงค่า H[1] ออกด้วยการ remove( ) แล้วเก็บค่าที่ไว้อาเรย์ตัวสุดท้าย H[10] ด้วย method insertAt( )

| index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| A | 14 | 8 | 10 | 4 | 7 | 9 | 3 | 2 | 1 | 16 |



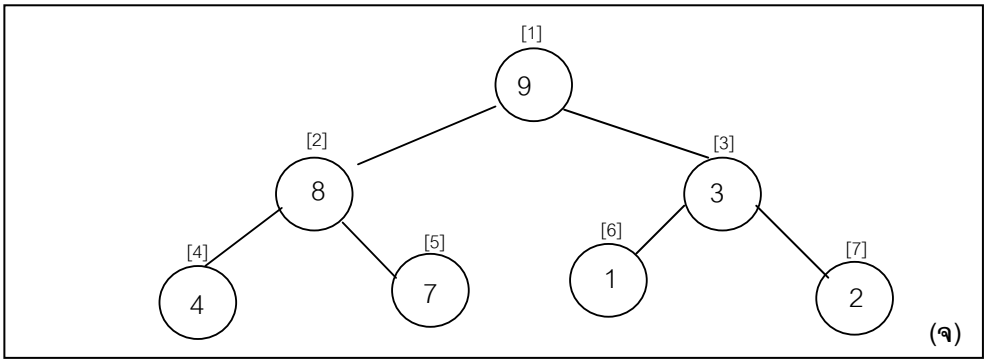

4. ดึงค่า H[1] ออกด้วยการ remove( ) แล้วเก็บค่าที่ไว้อาเรย์ตัวสุดท้าย H[9] ด้วย method insertAt(9)

| index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| A | 10 | 8 | 9 | 4 | 7 | 1 | 3 | 2 | 14 | 16 |




5. ทำซ้ำดึงค่า H[1] ออกด้วยการ remove( ) แล้วเก็บค่าที่ไว้อาเรย์ตัวสุดท้าย H[8] ด้วย method insertAt(8)

| index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| A | 9 | 8 | 3 | 4 | 7 | 1 | 2 | 10 | 14 | 16 |




6. ทำซ้ำเช่นนี้ไปเรื่อยๆ จนครบทุกจำนวน

| index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| A | 9 | 8 | 3 | 4 | 7 | 1 | 2 | 10 | 14 | 16 |
| | 8 | 7 | 3 | 4 | 2 | 1 | 9 | 10 | 14 | 16 |
| | 7 | 4 | 3 | 1 | 2 | 8 | 9 | 10 | 14 | 16 |
| | 4 | 3 | 2 | 1 | 7 | 8 | 9 | 10 | 14 | 16 |
| | 3 | 2 | 1 | 4 | 7 | 8 | 9 | 10 | 14 | 16 |
| | 2 | 1 | 3 | 4 | 7 | 8 | 9 | 10 | 14 | 16 |
| | 1 | 2 | 3 | 4 | 7 | 8 | 9 | 10 | 14 | 16 |
| | 1 | 2 | 3 | 4 | 7 | 8 | 9 | 10 | 14 | 16 |

รูปที่ 7.7 แสดงการเรียงลำดับ Heapsort

**โปรแกรม 7.2** HeapSort.java

```
// heapSort.java
// demonstrates heap sort
// to run this program: C>java HeapSortApp
import java.io.*;               // for I/O
import java.lang.Integer;       // for parseInt()
////////////////////////////////////////////////////////////////
class Node
  {
  public int iData;             // data item (key)
  public Node(int key)          // constructor
    { iData = key; }
  }  // end class Node
////////////////////////////////////////////////////////////////
class Heap
  {
  private Node[] heapArray;
  private int maxSize;          // size of array
  private int currentSize;      // number of items in array
// -------------------------------------------------------------
  public Heap(int mx)           // constructor
    {
    maxSize = mx;
    currentSize = 1;
    heapArray = new Node[maxSize];
    }
// -------------------------------------------------------------
  public Node remove()          // delete item with max key
    {                           // (assumes non-empty list)
    Node root = heapArray[1];
    heapArray[1] = heapArray[--currentSize];
    trickleDown(1);
    return root;
    }  // end remove()
// -------------------------------------------------------------
  public void trickleDown(int index)
    {
    int largerChild;
    Node top = heapArray[index];       // save root
    while(index <= currentSize/2)      // not on bottom row
      {
      int leftChild = 2*index;
      int rightChild = leftChild+1;
                                 // find larger child
```

```
      if(rightChild < currentSize &&   // right ch exists?
            heapArray[leftChild].iData <   heapArray[rightChild].iData)
        largerChild = rightChild;
      else
        largerChild = leftChild;
                                  // top >= largerChild?
      if(top.iData >= heapArray[largerChild].iData)
        break;
                                  // shift child up
      heapArray[index] = heapArray[largerChild];
      index = largerChild;            // go down
      }  // end while
    heapArray[index] = top;           // root to index
    }  // end trickleDown()
// -------------------------------------------------------------
  public void displayHeap()
    {
    int nBlanks = 32;
    int itemsPerRow = 1;
    int column = 0;
    int j = 1;                        // current item
    String dots = "................................";
    System.out.println(dots+dots);      // dotted top line

    while(currentSize > 1)            // for each heap item
      {
      if(column == 0)                 // first item in row?
        for(int k=0; k<nBlanks; k++)  // preceding blanks
          System.out.print(' ');
                                  // display item
      System.out.print(heapArray[j].iData);

      if(++j == currentSize)          // done?
        break;

      if(++column==itemsPerRow)       // end of row?
        {
        nBlanks /= 2;                 // half the blanks
        itemsPerRow *= 2;             // twice the items
        column = 0;                   // start over on
        System.out.println();         //    new row
        }
      else                        // next item on row
```

```
          for(int k=0; k<nBlanks*2-2; k++)
            System.out.print(' ');     // interim blanks
        }  // end for
      System.out.println("\n"+dots+dots); // dotted bottom line
      }  // end displayHeap()
// --------------------------------------------------------------
   public void displayArray()
      {
      for(int j=1; j<maxSize; j++)
        System.out.print(heapArray[j].iData + " ");
      System.out.println("");
      }
// --------------------------------------------------------------
   public void insertAt(int index, Node newNode)
      { heapArray[index] = newNode; }
// --------------------------------------------------------------
   public void incrementSize()
      { currentSize++; }
// --------------------------------------------------------------
   }  // end class Heap
////////////////////////////////////////////////////////////////
class HeapSortApp
   {
   public static void main(String[] args) throws IOException
      {
      int size, j;
      int h[]= {0, 4,            1, 3,  2, 16,  9, 10,  14, 8, 7};
      size = 11;
      Heap theHeap = new Heap(size);
      for(j=1; j<size; j++)       // fill array with
        {
            Node newNode = new Node(h[j]);
            theHeap.insertAt(j, newNode);
            theHeap.incrementSize();
        }

      System.out.print("Random: ");
        theHeap.displayArray();  // display random array
      theHeap.displayHeap();      // display heap

      for(j=size/2-1; j>=1; j--)  // make random array into heap
        theHeap.trickleDown(j);
```

```
    System.out.print("Heap:   ");
    theHeap.displayArray();     // dislay heap array
    theHeap.displayHeap();      // display heap
    for(j=size-1; j>=1; j--)    // remove from heap and
      {                         //    store at array end
      Node biggestNode = theHeap.remove();
       theHeap.insertAt(j, biggestNode);
       System.out.print("Remove & InsertAt :   ");
       theHeap.displayArray();     // dislay heap array
       theHeap.displayHeap();
     }
    System.out.print("Sorted: ");
    theHeap.displayArray();     // display sorted array
    }  // end main()
// --------------------------------------------------------------
  public static String getString() throws IOException
    {
    InputStreamReader isr = new InputStreamReader(System.in);
    BufferedReader br = new BufferedReader(isr);
    String s = br.readLine();
    return s;
    }
//--------------------------------------------------------------
  public static int getInt() throws IOException
    {
    String s = getString();
    return Integer.parseInt(s);
    }
// --------------------------------------------------------------
 }  // end class HeapSortApp
```

**Outputs**

```
Begin array : 4 1 3 2 16 9 10 14 8 7
                      4
           1                     3
      2          16          9          10
   14     8     7
TrickleDown :   16 14 10 8 7 9 3 2 4 1
                      16
           14                    10
      8          7          9          3
   2     4     1
Remove & InsertAt : 14 8 10 4 7 9 3 2 1 16
```

```
                    14
          8                   10
     4         7         9         3
  2     1
Remove & InsertAt : 10 8 9 4 7 1 3 2 14 16
                    10
          8                   9
     4         7         1         3
  2
Remove & InsertAt : 9 8 3 4 7 1 2 10 14 16
                    9
          8                   3
     4         7         1         2
Remove & InsertAt : 8 7 3 4 2 1 9 10 14 16
                    8
          7                   3
     4         2         1
Remove & InsertAt : 7 4 3 1 2 8 9 10 14 16
                    7
          4                   3
     1         2
Remove & InsertAt : 4 2 3 1 7 8 9 10 14 16
                    4
          2                   3
     1
Remove & InsertAt : 3 2 1 4 7 8 9 10 14 16
                    3
          2                   1
Remove & InsertAt : 2 1 3 4 7 8 9 10 14 16
                    2
          1
Remove & InsertAt : 1 2 3 4 7 8 9 10 14 16
                    1
Remove & InsertAt : 1 2 3 4 7 8 9 10 14 16
.....................................................................
Sorted: 1 2 3 4 7 8 9 10 14 16
```

**ประสิทธิภาพ**

เพราะว่าเมท็อด remove () ใช้เวลา 0(log n) สำหรับสมาชิกแต่ละตัว ดังนั้นในการเรียงลำดับข้อมูลทั้งหมด n ตัว จึงใช้เวลาทั้งหมด 0(n log n) ซึ่งมีประสิทธิภาพเท่ากับ Quicksort. แต่ Quicksort จะสามารถดำเนินการได้เร็วกว่า ทั้งนี้เพราะ Heapsort มีกระบวนการจัดการขั้นตอนการดำเนินการภายในลูป trickledown() มากกว่าลูปภายในของ Quicksort

## 7.5 คิวที่มีลำดับความสำคัญ (Priority Queues)

เรามักนำประโยชน์จากคุณสมบัติของ heap มาใช้กับคิวที่มีลำดับความสำคัญ เพราะมีฟังก์ชั่นการทำงานสอดคล้องกัน โดยโหนดแรกที่มีค่า key สูงสุดจะถูก remove ออกจากคิวเพื่อให้บริการก่อน สำหรับการสร้าง คลาสของคิวที่มีลำดับความสำคัญ (priority queue) ด้วยการเรียกคลาสของ heap มาใช้

**ตัวอย่าง 7. 7** สร้างคิวที่มีลำดับความสำคัญ PQ = { 30, 50, 10, 40, 20 } ผลลัพธ์ จะเป็นคิวที่มีลำดับความสำคัญก่อนตามลำดับ คือ 50, 40, 30, 20, 10

**โปรแกรม 7.3** priorityQ.java แสดงการสร้างคลาส PriorityQ โดยรวม class Heap มาเรียกใช้

```
// priorityQueue.java
// demonstrates priority queue
// to run this program: C>java PriorityQApp
//--------------------------------------------------------------------------------------------------------------
-----------------
class Heap
{  private Node heapArray[ ];
  public void insert (Node nd)
  {                          }
  public Node remove ( )
  {                          }
}//end class Heap
class PriorityQueue
{   private Heap theHeap;
   public PriorityQ(int mx)          // constructor
     {   theHeap = new Heap(mx);   }
   public void insert(Node nd)
    {   theHeap.insert(nd);   }
   public Node remove()
   {   return theHeap.remove();  }
}//class PriorityQueue
class PriorityQApp
  { public static void main(String[] args) throws IOException
    {   Node item;
```

```
        PriorityQ   thePQ = new PriorityQ(6);
       thePQ.insert(30);
       thePQ.insert(50);
       thePQ.insert(10);
       thePQ.insert(40);
       thePQ.insert(20);
       for(int i=1;i<=5;++i)
       {   item = thePQ.remove( );
          System.out.print(item.iData + " ");
       } // next i
      System.out.println();
    } // end main()
}//end class PriorityQApp
```

**Outputs**

```
50  40   30  20  10
```

## แบบฝึกหัด

1. กำหนดอาเรย์ theHeap = {3, 5, 6, 7, 20, 8, 2, 9, 12, 15, 30, 17}
   (a) จงสร้าง Complete Binary Tree ของ the Heap
   (b) ทำการปรับต้นไม้ใน (a) ให้เป็นฮีป โดยใช้เมท็อด heapify(0
   (c) ลบข้อมูลออกจากฮีปในข้อ (b) ออก 3 ตัว โดยเมท็อด remove()
   (d) เพิ่มข้อมูล 15, 20 และ 45 เข้าไปในฮีปทีละตัวตามลำดับ
2. ใช้โจทย์เดียวกับข้อ 1 ดำเนินการกับอาเรย์ {10, 2, 7, 6, 5, 9, 12, 2, 35, 22, 15, 1, 3, 4}
3. ใช้โจทย์เดียวกับข้อ 1 ดำเนินการกับอาเรย์ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 15, 22, 35}
4. จงวาดรูปฮีปที่ประกอบด้วยสมาชิกที่มีค่าคีย์เป็นจำนวนคี่และมีค่าตั้งแต่ 1 – 59 โดยที่เมื่อเพิ่มค่าคีย์ใหม่ที่มีค่า 32 จะมีผลทำให้มีการปรับฮีปโดยที่ค่าคีย์ 32 ถูกสลับกับโหนดพ่อแม่ให้ลอยขึ้นไปข้างบนเรื่อย ๆ จนกระทั่งถึงโหนดที่เป็นลูกของรากของต้นไม้ แล้วให้ทำการสลับที่ระหว่างโหนดลูกของราก กับโหนดที่มีค่าคีย์เป็น 32
5. จงแสดงขั้นตอนของขั้นตอนวิธี heapSort เมื่อข้อมูลนำเข้าตามลำดับ คือ (2, 5, 14, 6, 10, 23, 39, 16, 25, 15)
6. จงเขียนขั้นตอนวิธีในการดำเนินการกับสแตกโดยใช้ไพรออริตีคิว และตัวแปรชนิดจำนวนเต็มเพียง 1 ตัว เท่านั้น
7. จงเขียนขั้นตอนวิธีในการดำเนินการกับคิวโดยใช้ไพรออริตีคิว และตัวแปรชนิดจำนวนเต็มเพียง 1 ตัว เท่านั้น
8. กำหนดฮีป T และ ค่าคีย์ k จงเขียนขั้นตอนวิธีในการคำนวณสมาชิกทุกตัวในฮีป T ที่มีค่าคีย์น้อยกว่าหรือเท่ากับ k