WEEK 14

Lesson 27-28

Objectives

- Heap
  - What is Heap?
  - Description
  - Why we use Heap?
  - Insertion in Heap
  - Deletion in Heap
  - Java Code

What is Heap?

A heap is a binary tree with these characteristics:

- It's complete. This means it's completely filled in, reading from left to right across each row, although the last row need not be full. Figure shows complete and incomplete trees.
- It's (usually) implemented as an array. Describes how binary trees can be stored in arrays, rather than using references to connect the nodes.
- Each node in a heap satisfies the heap condition, which states that every node's key is larger than (or equal to) the keys of its children.
- The fact that a heap is a complete binary tree implies that there are no "holes" in the array used to represent it. Every cell is filled, from 0 to N-1.

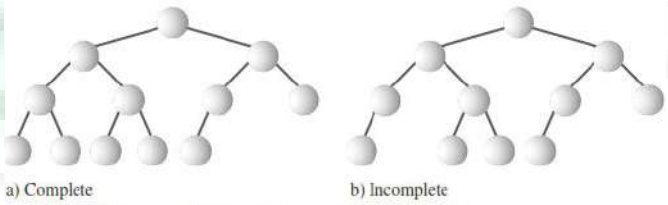
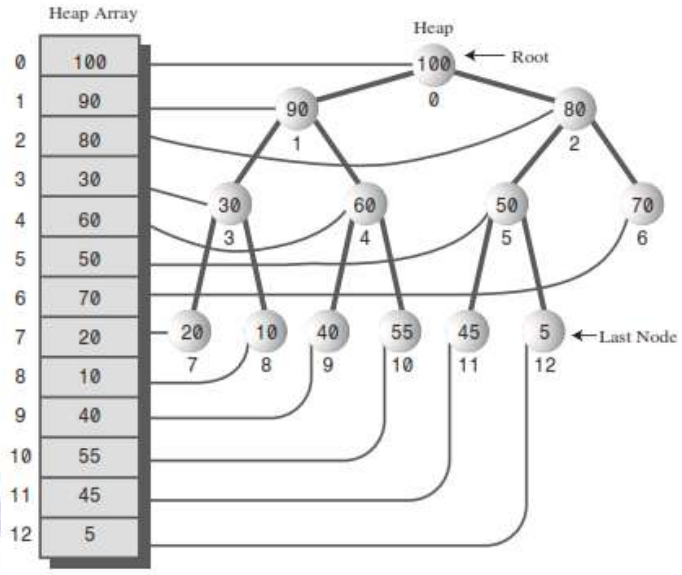


FIGURE Complete and incomplete binary trees.



? A heap and its underlying array.

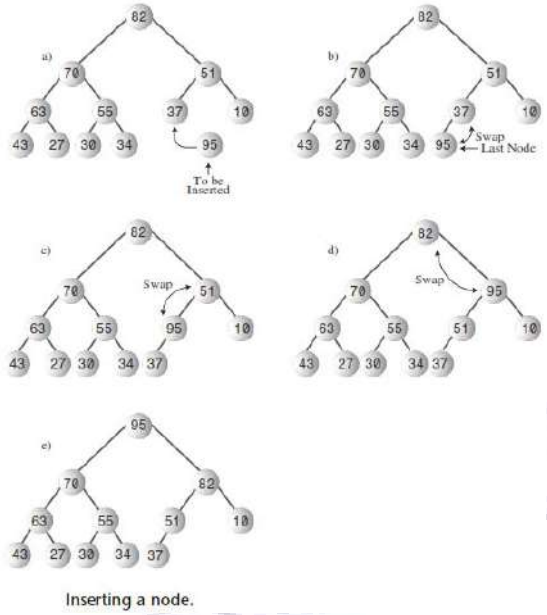
A heap is weakly ordered compared with a binary search tree, in which all a node's left descendants have keys less than all its right descendants. This implies, as we saw, that in a binary search tree you can traverse the nodes in order by following a simple algorithm.
In a heap, traversing the nodes in order is difficult because the organizing principle (the heap condition) is not as strong as the organizing principle in a tree. All you can say about a heap is that, along every path from the root to a leaf, the nodes are arranged in descending order. the nodes to the left or right of a given node, or on higher or lower levels—provided they're not on the same path—can have keys larger or smaller than the node's key. Except where they share the same nodes, paths are independent of each other. Because heaps are weakly ordered, some operations are difficult or impossible. Besides its failure to support traversal, a heap also does not allow convenient searching for a specified key. This is because there's not enough information to decide which of a node's two children to pick in trying to descend to a lower level during the search. It follows that a node with a specified key can't be deleted, at least in O(logN) time, because there's no way to find it. (These operations can be carried out, by looking at every cell of the array in sequence, but this is only possible in slow O(N) time.) Thus, the organization of a heap may seem dangerously close to randomness.

The Efficiency of Heapsort

Heapsort runs in O(N*logN) time. Although it may be slightly slower than quicksort, an advantage over quicksort is that it is less sensitive to the initial distribution of data. Certain arrangements of key values can reduce quicksort to slow O(N) time, whereas heapsort runs in O(N*logN) time no matter how the data is distributed.

Insertion

Inserting a node is also easy. Insertion uses trickle up, rather than trickle down. Initially, the node to be inserted is placed in the first open position at the end of the array, increasing the array size by one: heapArray[N] = newNode; N++;



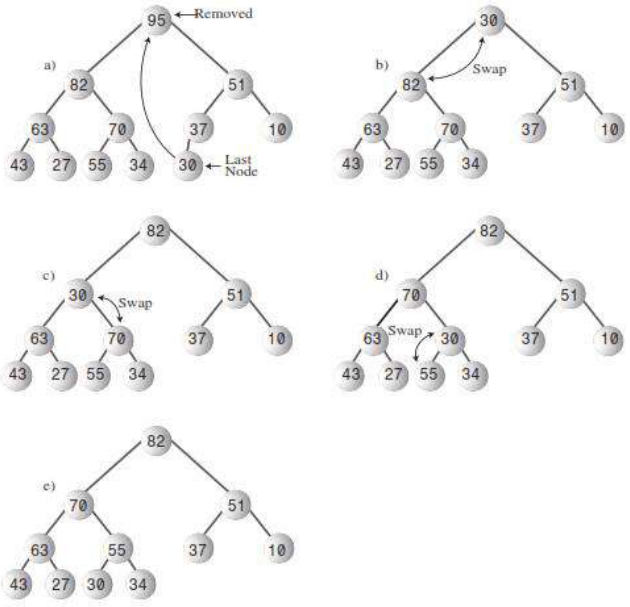
Inserting a node.

Removal

Removal means removing the node with the maximum key. This node is always the root, so removing it is easy. The root is always at index 0 of the heap array: maxNode = heapArray[0];

The problem is that once the root is gone, the tree is no longer complete; there's an empty cell. This "hole" must be filled in. We could shift all the elements in the array down one cell, but there's a much faster approach. Here are the steps for removing the maximum node:

1. Remove the root.
2. Move the last node into the root.
3. Trickle the last node down until it's below a larger node and above a smaller one.



Removing the maximum node.

The heapSort.java Program

```
import java.io.*;
////////////////////////////////////////////////////////////////
class Node
{
private int iData;            // data item (key)
// -------------------------------------------------------------public
Node(int key)          // constructor
{ iData = key; }
// -------------------------------------------------------------
public int getKey()
{ return iData; }
// -------------------------------------------------------------}
 // 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 = 0;
heapArray = new Node[maxSize];
}
// -------------------------------------------------------------public
Node remove()          // delete item with max key
{                      // (assumes non-empty list)
Node root = heapArray[0];
heapArray[0] = heapArray[--currentSize];
trickleDown(0);
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+1;
int rightChild = leftChild+1;
// find larger child
if(rightChild < currentSize &&   // right ch exists?
heapArray[leftChild].getKey() <
heapArray[rightChild].getKey())
largerChild = rightChild;
else
largerChild = leftChild;
// top >= largerChild?
if(top.getKey() >= heapArray[largerChild].getKey())
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 = 0;                        // current item
String dots = "................................";
System.out.println(dots+dots);      // dotted top line
while(currentSize > 0)             // 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].getKey());
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=0; j<maxSize; j++)
System.out.print(heapArray[j].getKey() + " ");
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;
```

```
System.out.print("Enter number of items: ");
size = getInt();
Heap theHeap = new Heap(size);
for(j=0; j<size; j++)       // fill array with
{                           //    random nodes
int random = (int)(java.lang.Math.random()*100);
Node newNode = new Node(random);
theHeap.insertAt(j, newNode);
theHeap.incrementSize();
}
System.out.print("Random: ");
theHeap.displayArray();  // display random array
for(j=size/2-1; j>=0; j--)  // make random array into heap
theHeap.trickleDown(j);
System.out.print("Heap:   ");
theHeap.displayArray();     // display heap array
theHeap.displayHeap();      // display heap
for(j=size-1; j>=0; j--)    // remove from heap and
{                           //    store at array end
Node biggestNode = theHeap.remove();
theHeap.insertAt(j, biggestNode);
}
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);
}
// --------------------------------------------------------------}
```