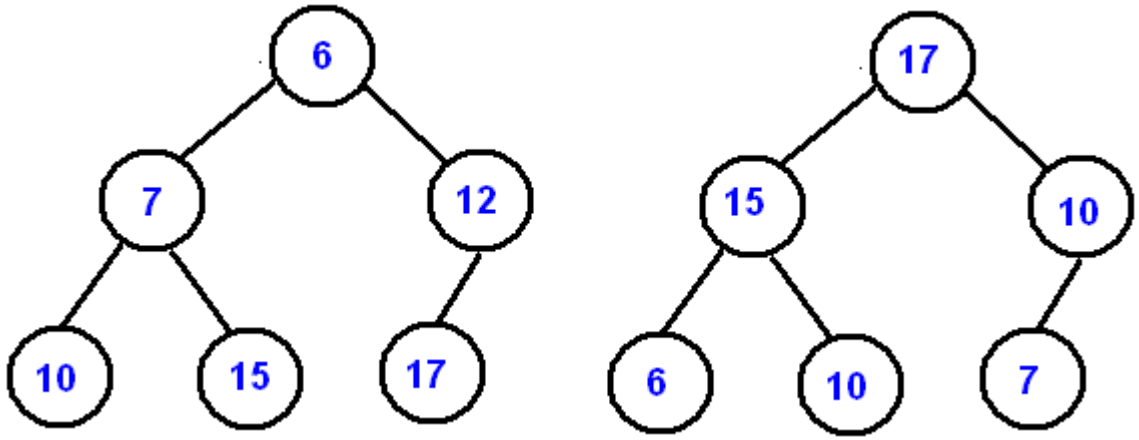
Binary Heaps

Introduction

A binary heap is a complete binary tree which satisfies the heap ordering property. The ordering can be one of two types:

- the *min-heap property*: the value of each node is greater than or equal to the value of its parent, with the minimum-value element at the root.
- the *max-heap property*: the value of each node is less than or equal to the value of its parent, with the maximum-value element at the root.

Throughout this chapter the word "heap" will always refer to a min-heap.



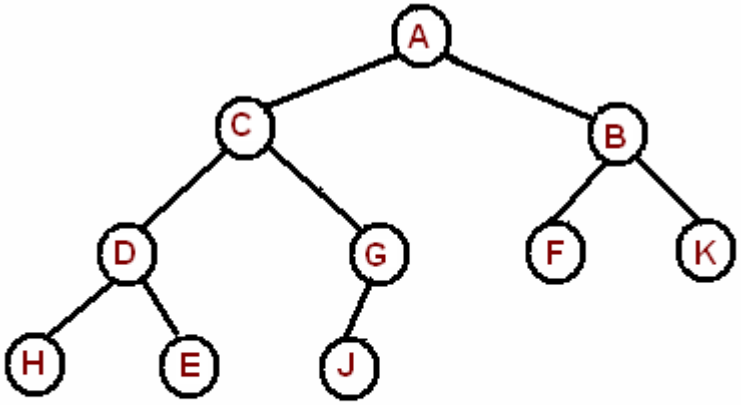
In a heap the highest (or lowest) priority element is always stored at the root, hence the name "heap". A heap is not a sorted structure and can be regarded as partially ordered. As you see from the picture, there is no particular relationship among nodes on any given level, even among the siblings.

Since a heap is a complete binary tree, it has a smallest possible height - a heap with N nodes always has O(log N) height.

A heap is useful data structure when you need to remove the object with the highest (or lowest) priority. A common use of a heap is to implement a priority queue.

Array Implementation

A complete binary tree can be uniquely represented by storing its level order traversal in an array.



0	1	2	3	4	5	6	7	8	9	10
	A	C	B	D	G	F	K	H	E	J

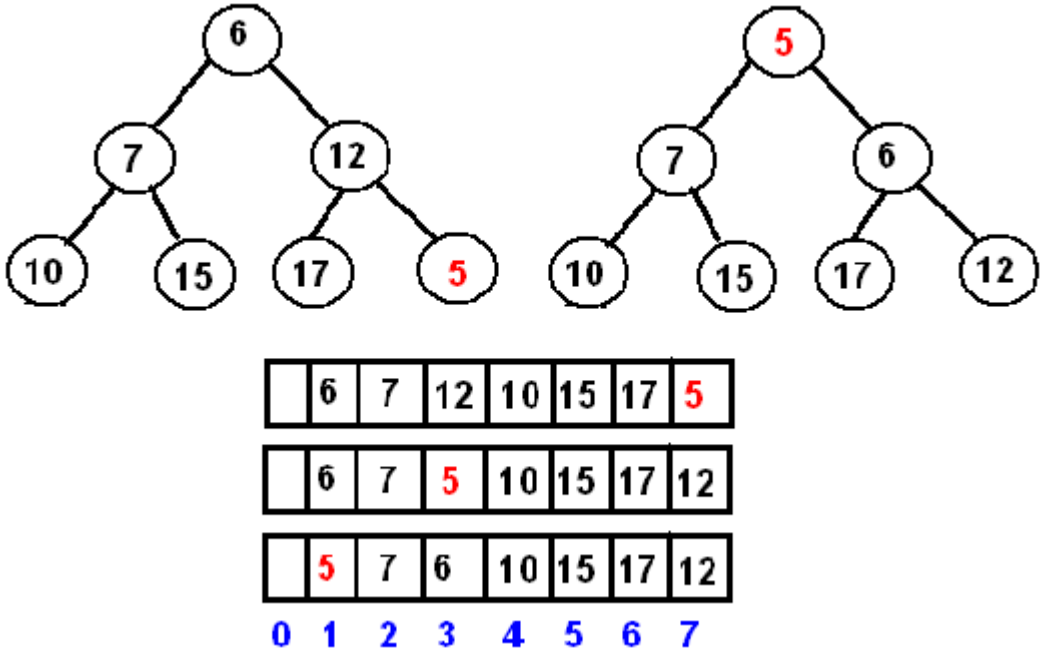
The root is the second item in the array. We skip the index zero cell of the array for the convenience of implementation. Consider k-th element of the array, the

its left child is located at 2*k index
its right child is located at 2*k+1. index
its parent is located at k/2 index

See Heap.java. for implementation details.

Insert

The new element is initially appended to the end of the heap (as the last element of the array). The heap property is repaired by comparing the added element with its parent and moving the added element up a level (swapping positions with the parent). This process is called "percolation up". The comparison is repeated until the parent is larger than or equal to the percolating element.



The following code example demonstrates the algorithm

```
public void insert(Comparable x)
{
        if(size == heap.length - 1) doubleSize();

        //Insert a new item to the end of the array
        int pos = ++size;

        //Percolate up
        for(; pos > 1 && x.compareTo(heap[pos/2]) < 0; pos = pos/2 )
                heap[pos] = heap[pos/2];

        heap[pos] = x;
}
```

The worst-case runtime of the algorithm is O(log n), since we need at most one swap on each level of a heap on the path from the inserted node to the root.

DeleteMin

The minimum element can be found at the root, which is the first element of the array. We remove the root and replace it with the last element of the heap and then restore the heap property by *percolating down*. Similar to insertion, the worst-case runtime is O{log n).

HeapSort

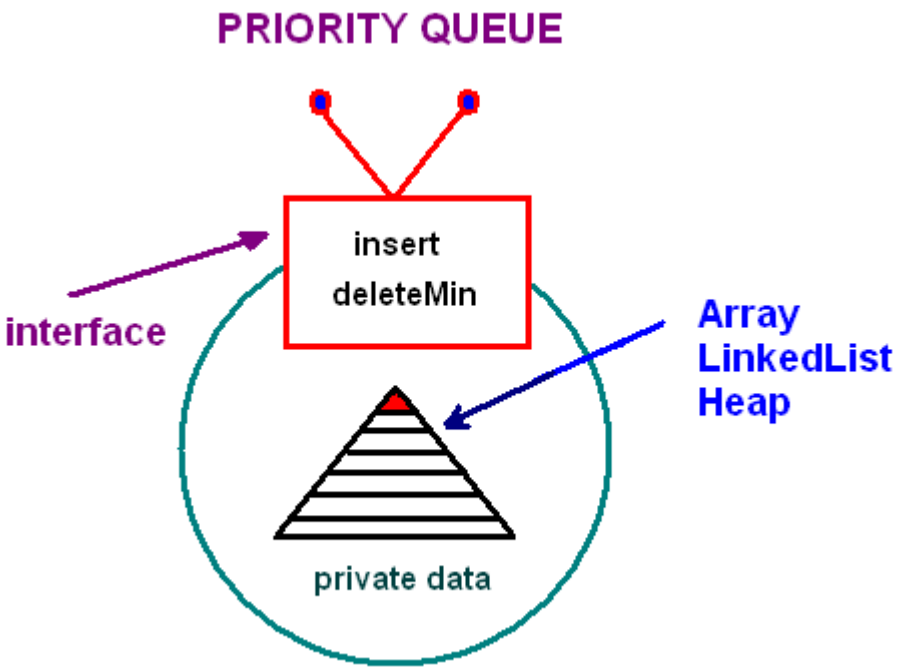
The algorithm runs in two steps. Given an array of data, first, we build a heap and then turn it into a sorted list by calling deleteMin. The running time of the algorithm is O(n log n).

Example. Given an array {3, 1, 6, 5, 2, 4}. We will sort it with the HeapSort.

```
1. build a heap     1, 2, 4, 5, 3, 6
2. turn this heap into a sorted list

     deleteMin
     1, 2, 4, 5, 3, 6           swap 1 and 6
     6, 2, 4, 5, 3,        1    restore heap
     2, 6, 4, 5, 3,        1
     2, 3, 4, 5, 6,        1

     deleteMin
     2, 3, 4, 5, 6,        1    swap 2 and 6
     6, 3, 4, 5,        2, 1    restore heap
     3, 6, 4, 5,        2, 1
     3, 5, 4, 6,        2, 1

     deleteMin
     3, 5, 4, 6,       2, 1     swap 3 and 6
     6, 5, 4,       3, 2, 1     restore heap
     4, 5, 6,       3, 2, 1     restore heap

     deleteMin
     4, 5, 6,       3, 2, 1     swap 4 and 6
     6, 5,       4, 3, 2, 1     restore heap
     5, 6,       4, 3, 2, 1

     deleteMin

     6,       5, 4, 3, 2, 1
```

Priority Queue

Priority queues are useful for any application that involves processing elements based on some priority. It supports two major operations `insert(object)` and `deleteMin()`. The elements of a priority queue must be comparable to each other, either through the Comparable or Comparator interfaces. We have introduced a priority queue when we discussed Java's collection classes. In this chapter we reinforce priority queue operations with a binary heap. Using a heap to implement a priority queue, we will always have the element of highest priority in the root node of the heap.



The reason we re-implement a priority queue is to improve its effiiciency. When we implemented a priority queue with an array or a linked list, the efficiency of some operations were O(n).

	insert	deleteMin	remove	findMin
ordered array	O(n)	O(1)	O(n)	O(1)
ordered list	O(n)	O(1)	O(1)	O(1)
unordered array	O(1)	O(n)	O(1)	O(n)
unordered list	O(1)	O(n)	O(1)	O(n)

Using a binary heap, the runtime of both the deleteMin and insert operations is O(log n).

	insert	deleteMin	remove	findMin

binary heap	O(log n)	O(log n)	O(log n)	O(1)

Victor S.Adamchik, CMU, 2009