***Heap***

The (binary) heap data structure is an array object that we can view as a

nearly complete binary tree.

Each node of the tree corresponds to an element of the array. The tree is com-

pletely filled on all levels except possibly the lowest, which is filled from the left up to a point.

An array A that represents a heap is an object with two attributes:

* **A.length**, which (as usual) gives the number of elements in the array, and
* **A.heap-size**, which represents how many elements in the heap are stored within array A.

That is, although **A[1.... A.length]** may contain numbers, only the ele-

ments in **A[1....A.heap-size]**, where **0 <= A.heap-size <= A.length**, are valid ele-

ments of the heap.

The root of the tree is **A[1]**, and given the index i of a node, we

can easily compute the indices of its parent, left child, and right child.

**There are two kinds of binary heaps**:

1. **max-heaps** and
2. **min-heaps**.

In both kinds, the values in the nodes satisfy a heap property, the specifics of which depend on the kind of heap.

* In a max-heap, the max-heap property is that for every node i

other than the root,

**A[PARENT(i)]>=A[i]** ,

**that is, the value of a node is at most the value of its parent**. Thus, the largest element in a max-heap is stored at the root, and the subtree rooted at a node contains values no larger than that contained at the node itself.

* A min-heap is organized in

the opposite way; the min-heap property is that for every node i other than the

root,

**A[Parent(i)]<=A[i]**

**The smallest element in a min-heap is at the root.**

-> Viewing a heap as a tree, we define the height of a node in a heap to be the

number of edges on the longest simple downward path from the node to a leaf, and

we define the height of the heap to be the height of its root. Since a heap of n elements is based on a complete binary tree, its height is theta(lg n).

**Applications of Heaps:**

1) [Heap Sort](http://geeksquiz.com/heap-sort/): Heap Sort uses Binary Heap to sort an array in O(nLogn) time.

2) Priority Queue: Priority queues can be efficiently implemented using Binary Heap because it supports insert(), delete() and extractmax(), decreaseKey() operations in O(logn) time. Binomoial Heap and Fibonacci Heap are variations of Binary Heap. These variations perform union also efficiently.

3) Graph Algorithms: The priority queues are especially used in Graph Algorithms like [Dijkstra’s Shortest Path](http://www.geeksforgeeks.org/greedy-algorithms-set-7-dijkstras-algorithm-for-adjacency-list-representation/) and[Prim’s Minimum Spanning Tree](http://www.geeksforgeeks.org/graph-and-its-representations/).

4) Many problems can be efficiently solved using Heaps. See following for example.  
a) [K’th Largest Element in an array](http://www.geeksforgeeks.org/k-largestor-smallest-elements-in-an-array/).  
b) [Sort an almost sorted array/](http://www.geeksforgeeks.org/nearly-sorted-algorithm/)  
c) [Merge K Sorted Arrays](http://www.geeksforgeeks.org/merge-k-sorted-arrays/).