**UNIT-2**

**Priority Queues**

* **priority queue**: is a queue in which each element is associated with a priority level. Elements are inserted in the order of arrival or priority order but the element with highest priority is processed first (deleted from the queue)

1. **Implementations of priority queues** 
   1. Unsorted list
      1. Insert at the beginning - O(1)
      2. Find the minimum - O(N)
   2. Sorted list (BST)
      1. Insert O(N)
      2. Find minimum - O(1)
   3. Binary heap
      1. Insert O(logN)
      2. Find minimum O(logN)

Deleting the minimal element takes a constant time, however after that the heap structure has to be adjusted, and this requires O(logN) time.

* **Binary Heap :** Heap tree is a binary tree with two properties

**Heap order property:** The element at the root should be less than the elements at child nodes.

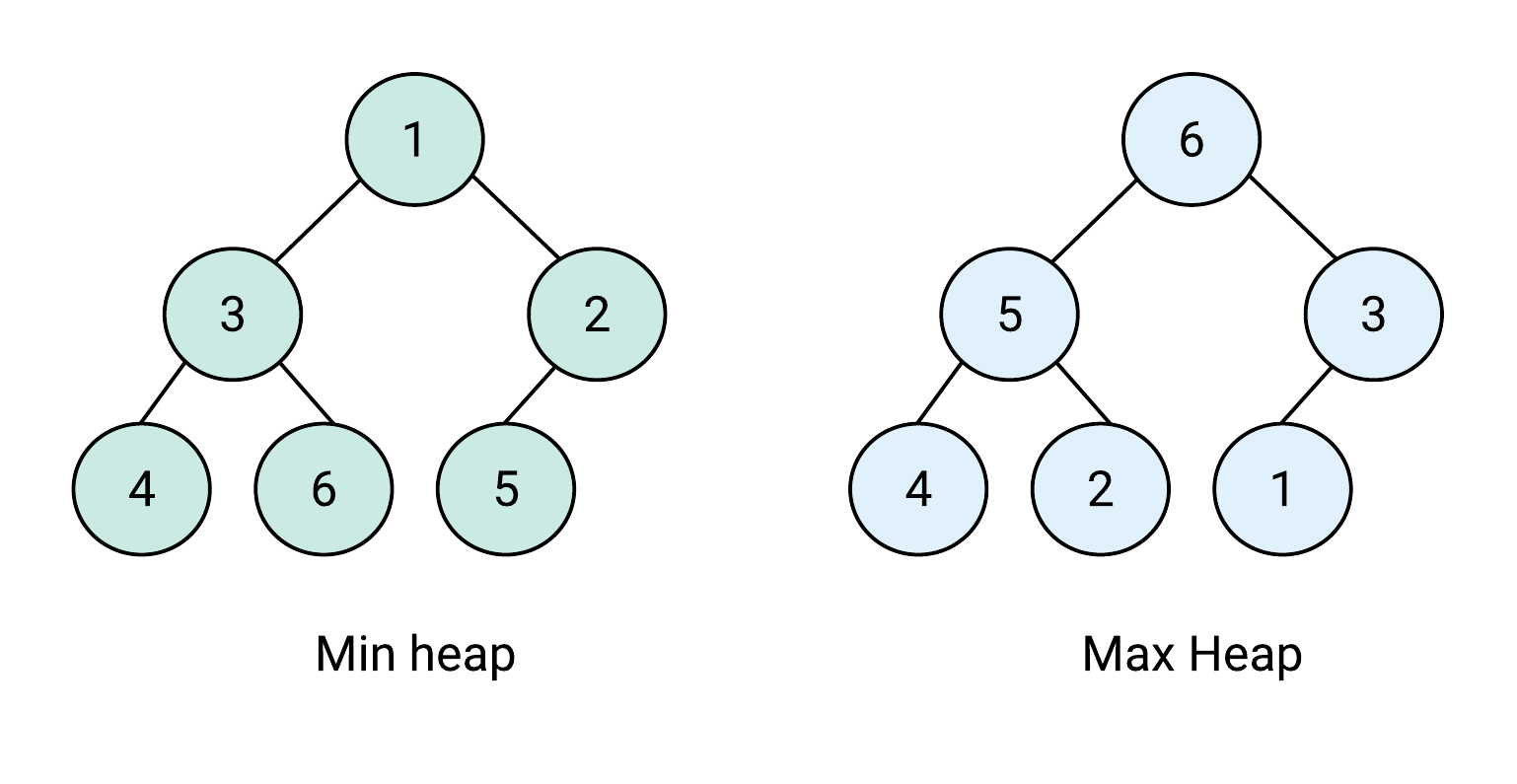
**Heap structure property**: All the levels should be completely filled except the last two. And the elements should be filled from left to right.

**Max-Heap**

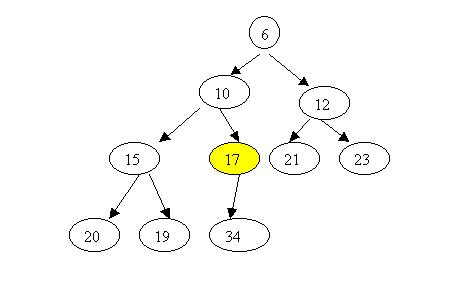
In max heap, the key value of a node is greater than or equal to the key value at its child nodes.

**Min-Heap**

In min heap, the key value of a node is less than or equal to the key value at its child nodes.



**Binary heap implementation with an array**



Since the tree is complete, we use the following array implementation:

Given array **A**:

**A(1)** - contains the root  
The node in **A(i)** has its left child in **A(2\*i)**, its right child in **A(2\*i+1)**,   
and its parent in **A([i/2])**.

The smallest element is always at the root, thus the access time to the element with highest priority is constant O(1).

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **6** | **10** | **12** | **15** | **17** | **21** | **23** | **20** | **19** | **34** |  |

This is the array that implements the binary heap in the example.   
The nodes of the tree are written in the array level by level from left to right.   
The first element in the array is 6 - this is the node at level 0.   
Then come 10 and 12 - the nodes at level 1.Further we have 15, 17 (children of 10),   
then 18 and 23 (children of 12) - all at level 2.   
Finally we have 20, 19 and 34 - the nodes at the last level.

Let's take a node in the array and find its parent and its children. Consider for example 17.

Its position in the array is 5.  
Its parent 12 is at position [5/2] = 2  
Its left child is at position 5\*2 = 10, this is the element 34,   
and its right child is at position 2\*5 + 1 = 11, (still empty).

* **Binary Heap Operations:**

we can perform following operations on Binary Heap.

* Build Heap
* Insert
* Delete
* Decrease Key
* Increase Key

**Build Heap Algorithm:**

BUILD-HEAP(A)

heapsize := size(A);

for i := floor(heapsize/2) downto 1

do HEAPIFY(A, i);

end for

END

**Complexity analysis**

algorithm suggests that the running time is O(nlogn), since each call to **Heapify** costs O(logn) and **Build-Heap** makes O(n) such calls.

**Insertion Algorithm**

**Step 1** − Create a new node at the end of heap.

**Step 2** − Assign new value to the node.

**Step 3** − Compare the value of this child node with its parent. Percolate

**Step 4** − If value of parent is less than child, then swap them. Down

**Step 5** − Repeat step 3 & 4 until Heap property holds.

*See the examples in note book.*

**Complexity analysis**

In the worst case, the newly inserted node has to be swapped at each level from the bottom up to the root node to maintain the heap property. Now we know that a heap tree is a balanced [complete tree](https://www.baeldung.com/cs/binary-tree-intro) data structure.

In the worst case, we need one swap at each level of the tree. So the total number of the swaps would be equal to the [height](https://www.baeldung.com/cs/height-balanced-tree) of the heap tree. The height of a balanced complete tree with N number of nodes is logN. Each swap takes O(1) time.

*See the examples in note book.*

**Deletion Algorithm**

**Step 1** − Remove root node.

**Step 2** − Move the last element of last level to root.

**Step 3** − Compare the value of this child node with its parent.

**Step 4** − If value of parent is less than child, then swap them. Percolate Up

**Step 5** − Repeat step 3 & 4 until Heap property holds.

*See the examples in note book.*

**Complexity analysis**

In order to delete the height priority element which is always at root will be replaced with last element. Then the tree should be heapified at the root. In the worst case, the root has to be swapped with its child on each level until it reaches the bottom level of the heap, meaning that the delete operation has a time complexity relative to the height of the tree, or O(log *n*).

**decreaseKey():** Decreases value of key. The time complexity of this operation is O(Logn). If the decreases key value of a node is greater than the parent of the node, then we don’t need to do anything. Otherwise, we need to traverse up to fix the violated heap property.

**increaseKey():** increases value of key. The time complexity of this operation is O(Logn). If the increases key value of a node is less than the parent of the node, then we don’t need to do anything. Otherwise, we need to traverse up to fix the violated heap property.

* **Heap sort:**

**Technique:**

1. Consider the values of the elements as priorities and build the heap tree.
2. Start deleteMin operations, storing each deleted element at the end of the heap array.

**Algorithm:**

***Heapify (A, i)***

1. *l ← left [i]*
2. *r ← right [i]*
3. *if l ≤ heap-size [A] and A[l] > A[i]*
4. *then largest ← l*
5. *else largest ← i*
6. *if r ≤ heap-size [A] and A[i] > A[largest]*
7. *then largest ← r*
8. *if largest  ≠ i*
9. *then exchange A[i] ↔ A[largest]*
10. *Heapify (A, largest)*

***BUILD\_HEAP (A)***

1. *heap-size (A) ← length [A]*
2. *For i ←  floor(length[A]/2) down to 1 do*
3. *Heapify (A, i)*

***HEAPSORT (A)***

1. *BUILD\_HEAP (A)*
2. *for i ← length (A) down to 2 do  
   exchange A[1] ↔ A[i]  
   heap-size [A] ← heap-size [A] - 1  
   Heapify (A, 1)*

**Complexity analysis**

Time complexity of heapify is O(Logn). Time complexity of createAndBuildHeap() is O(n) and the overall time complexity of Heap Sort is O(nLogn).

*See the examples in note book.*

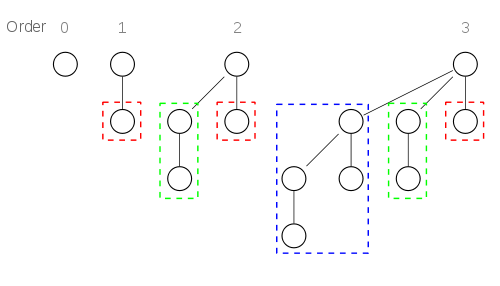
* **Binomial heap**

A **binomial heap** is a heap similar to a [binary heap](http://en.wikipedia.org/wiki/Binary_heap) (takes O(N) complexity for merge) but also supports quickly merging two heaps. This is achieved by using a special tree structure called binomial tree.

**Binomial tree**

A binomial heap is implemented as a collection of [binomial](http://en.wikipedia.org/wiki/Binomial) [trees](http://en.wikipedia.org/wiki/Tree_data_structure). A **binomial tree** is defined recursively:

* A binomial tree of order 0 is a single node
* A binomial tree of order *k* has a root node whose children are roots of binomial trees of orders *k*−1, *k*−2, ..., 2, 1, 0 (in this order).
* A binomial tree of order *k* has 2k nodes, height *k*.
* Because of its unique structure, a binomial tree of order *k* can be constructed from two trees of order *k*−1 trivially by attaching one of them as the leftmost child of root of the other one. This feature is central to the *merge* operation of a binomial heap, which is its major advantage over other conventional heaps.
* The name comes from the shape: a binomial tree of order has nodes at depth d.

[](http://en.wikipedia.org/wiki/File:Binomial_Trees.svg)

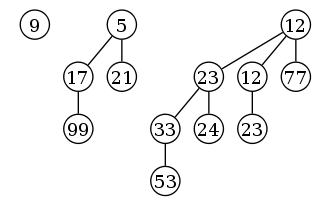
Binomial trees of order 0 to 3: Each tree has a root node with subtrees of all lower ordered binomial trees, which have been highlighted. For example, the order 3 binomial tree is connected to an order 2, 1, and 0 binomial tree.

A binomial heap is implemented as a set of binomial trees that satisfy the *binomial heap properties*:

* Each binomial tree in a heap obeys the [*minimum-heap property*](http://en.wikipedia.org/wiki/Minimum-heap_property): the key of a node is greater than or equal to the key of its parent.
* There can only be either *one* or *zero* binomial trees for each order, including zero order.

The first property ensures that the root of each binomial tree contains the smallest key in the tree, which applies to the entire heap.

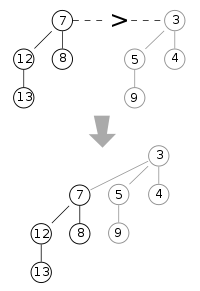
The second property implies that a binomial heap with *n* nodes consists of at most [log](http://en.wikipedia.org/wiki/Binary_logarithm)*n* + 1 binomial trees. In fact, the number and orders of these trees are uniquely determined by the number of nodes *n*: each binomial tree corresponds to one digit in the [binary](http://en.wikipedia.org/wiki/Binary_numeral_system) representation of number *n*. For example number 13 is 1101 in binary, , and thus a binomial heap with 13 nodes will consist of three binomial trees of orders 3, 2, and 0 (see figure below).

[](http://en.wikipedia.org/wiki/File:Binomial-heap-13.svg)

*Example of a binomial heap containing 13 nodes with distinct keys. The heap consists of three binomial trees with orders 0, 2, and 3.*

### Merge binomial trees:

As mentioned above, the simplest and most important operation is the merging of two binomial trees of the same order within two binomial heaps. Due to the structure of binomial trees, they can be merged trivially. As their root node is the smallest element within the tree, by comparing the two keys, the smaller of them is the minimum key, and becomes the new root node. Then the other tree become a subtree of the combined tree. This operation is basic to the complete merging of two binomial heaps.

[](http://en.wikipedia.org/wiki/File:Binomial_heap_merge1.svg)

The operation of **merging** two heaps is perhaps the most interesting and can be used as a subroutine in most other operations. The lists of roots of both heaps are traversed simultaneously, similarly as in the merge.

If only one of the heaps contains a tree of order *j*, this tree is moved to the merged heap. If both heaps contain a tree of order *j*, the two trees are merged to one tree of order *j*+1 so that the minimum-heap property is satisfied. Note that it may later be necessary to merge this tree with some other tree of order *j*+1 present in one of the heaps. In the course of the algorithm, we need to examine at most three trees of any order (two from the two heaps we merge and one composed of two smaller trees).Each tree has order at most log *n* and therefore the running time is *O*(log *n*).

### Insert

**Inserting** a new element to a heap can be done by simply creating a new heap containing only this element and then merging it with the original heap. Due to the merge, insert takes O(log *n*) time, however it has an *amortized* time of O(1) (i.e. constant).

### Find minimum or ExtractMin

To find the **minimum** element of the heap, find the minimum among the roots of the binomial trees. This can again be done easily in *O*(log *n*) time, as there are just *O*(log *n*) trees and hence roots to examine.

By using a pointer to the binomial tree that contains the minimum element, the time for this operation can be reduced to *O*(1). The pointer must be updated when performing any operation other than Find minimum. This can be done in *O*(log *n*) without raising the running time of any operation.

### Delete minimum

To **delete the minimum element** from the heap, first find this element, remove it from its binomial tree, and obtain a list of its subtrees. Then transform this list of subtrees into a separate binomial heap by reordering them from smallest to largest order. Then merge this heap with the original heap. Since each tree has at most log *n* children, creating this new heap is *O*(log *n*). Merging heaps is *O*(log *n*), so the entire delete minimum operation is *O*(log *n*).

**decreaseKey:** decreaseKey() is also similar to Binary Heap. We compare the decreases key with it parent and if parent’s key is more, we swap keys and recur for the parent. We stop when we either reach a node whose parent has a smaller key or we hit the root node. Time complexity of decreaseKey() is O(Logn).

*See the examples in note book.*

**Applications of Binomial Heap:**

* Discrete event simulation  
  A discrete-event simulation (DES) models the operation of a system as a (discrete) sequence of events in time.
* Network Bandwidth management  
  Priority queuing can be used to manage limited resources such as bandwidth on a transmission line from a network router.