ADVANCE DATA STRUCTURE NOTES
UNIT-I

What is a priority queue?

A priority queue is an abstract data type that behaves similarly to the normal queue except that each element has some priority, i.e., the element with the highest priority would come first in a priority queue. The priority of the elements in a priority queue will determine the order in which elements are removed from the priority queue.

The priority queue supports only comparable elements, which means that the elements are either arranged in an ascending or descending order.

For example, suppose we have some values like 1, 3, 4, 8, 14, 22 inserted in a priority queue with an ordering imposed on the values is from least to the greatest. Therefore, the 1 number would be having the highest priority while 22 will be having the lowest priority.

Characteristics of a Priority queue

A priority queue is an extension of a queue that contains the following characteristics:

- Every element in a priority queue has some priority associated with it.
- An element with the higher priority will be deleted before the deletion of the lesser priority.
- If two elements in a priority queue have the same priority, they will be arranged using the FIFO principle.

Let's understand the priority queue through an example.

We have a priority queue that contains the following values:

1, 3, 4, 8, 14, 22

All the values are arranged in ascending order. Now, we will observe how the priority queue will look after performing the following operations:

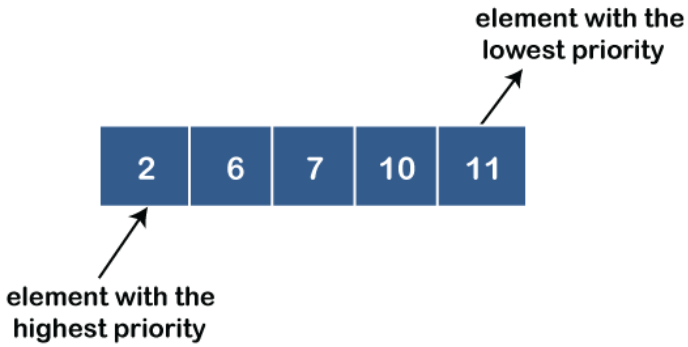
- **poll():** This function will remove the highest priority element from the priority queue. In the above priority queue, the '1' element has the highest priority, so it will be removed from the priority queue.
- **add(2):** This function will insert '2' element in a priority queue. As 2 is the smallest element among all the numbers so it will obtain the highest priority.

- **poll():** It will remove '2' element from the priority queue as it has the highest priority queue.
- **add(5):** It will insert 5 element after 4 as 5 is larger than 4 and lesser than 8, so it will obtain the third highest priority in a priority queue.

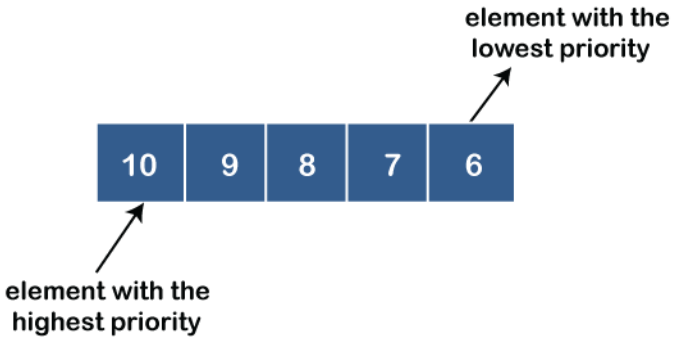
Types of Priority Queue

There are two types of priority queue:

- **Ascending order priority queue:** In ascending order priority queue, a lower priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in an ascending order like 1,2,3,4,5; therefore, the smallest number, i.e., 1 is given as the highest priority in a priority queue.



- **Descending order priority queue:** In descending order priority queue, a higher priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in descending order like 5, 4, 3, 2, 1; therefore, the largest number, i.e., 5 is given as the highest priority in a priority queue.



Representation of priority queue

Now, we will see how to represent the priority queue through a one-way list.

We will create the priority queue by using the list given below in which **INFO** list contains the data elements, **PRN** list contains the priority numbers of each data element available in the **INFO** list, and LINK basically contains the address of the next node.

	INFO	PNR	LINK
0	200	2	4
1	400	4	2
2	500	4	6
3	300	1	0
4	100	2	5
5	600	3	1
6	700	4	

Let's create the priority queue step by step.

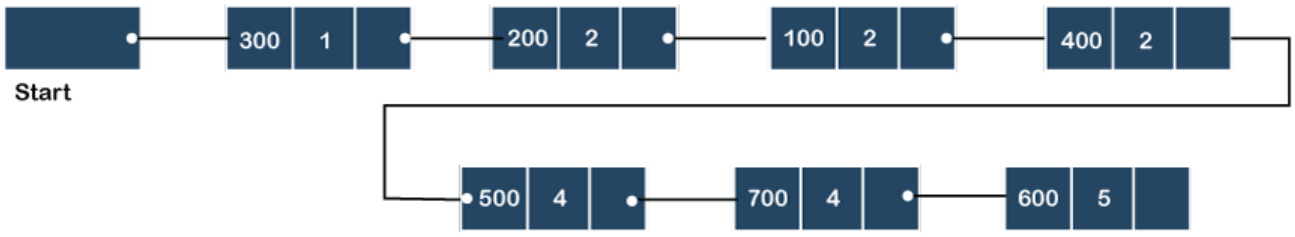
In the case of priority queue, lower priority number is considered the higher priority, i.e., lower priority number = higher priority.

Step 1: In the list, lower priority number is 1, whose data value is 333, so it will be inserted in the list as shown in the below diagram:

Step 2: After inserting 333, priority number 2 is having a higher priority, and data values associated with this priority are 222 and 111. So, this data will be inserted based on the FIFO principle; therefore 222 will be added first and then 111.

Step 3: After inserting the elements of priority 2, the next higher priority number is 4 and data elements associated with 4 priority numbers are 444, 555, 777. In this case, elements would be inserted based on the FIFO principle; therefore, 444 will be added first, then 555, and then 777.

Step 4: After inserting the elements of priority 4, the next higher priority number is 5, and the value associated with priority 5 is 666, so it will be inserted at the end of the queue.



Implementation of Priority Queue

The priority queue can be implemented in four ways that include arrays, linked list, heap data structure and binary search tree. The heap data structure is the most efficient

way of implementing the priority queue, so we will implement the priority queue using a heap data structure in this topic. Now, first we understand the reason why heap is the most efficient way among all the other data structures.

Analysis of complexities using different implementations

Implementation	add	Remove	peek
Linked list	O(1)	O(n)	O(n)
Binary heap	O(logn)	O(logn)	O(1)
Binary search tree	O(logn)	O(logn)	O(1)

Heap Data Structure

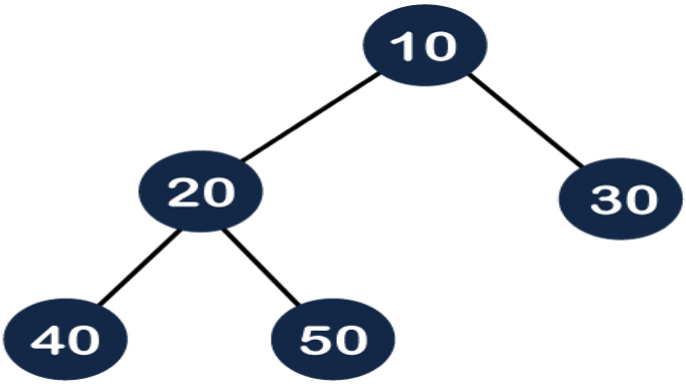
What is Heap?

A heap is a complete binary tree, and the binary tree is a tree in which the node can have utmost two children. Before knowing more about the heap data structure, we should know about the complete binary tree.

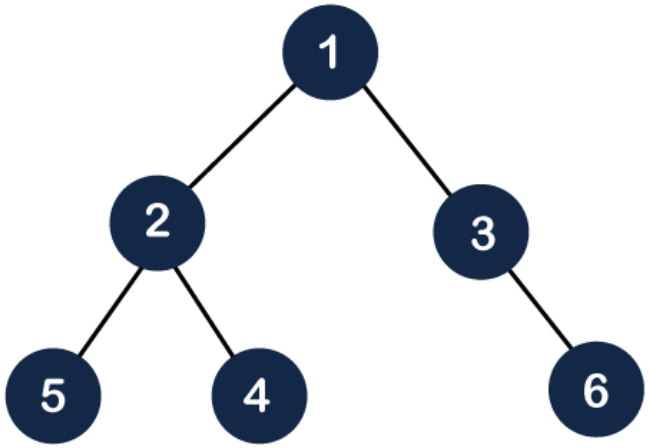
What is a complete binary tree?

A complete binary tree is a binary tree in which all the levels except the last level, i.e., leaf node should be completely filled, and all the nodes should be left-justified.

Let's understand through an example.



In the above figure, we can observe that all the internal nodes are completely filled except the leaf node; therefore, we can say that the above tree is a complete binary tree.



The above figure shows that all the internal nodes are completely filled except the leaf node, but the leaf nodes are added at the right part; therefore, the above tree is not a complete binary tree.

Note: The heap tree is a special balanced binary tree data structure where the root node is compared with its children and arrange accordingly.

How can we arrange the nodes in the Tree?

There are two types of the heap:

- Min Heap
- Max heap

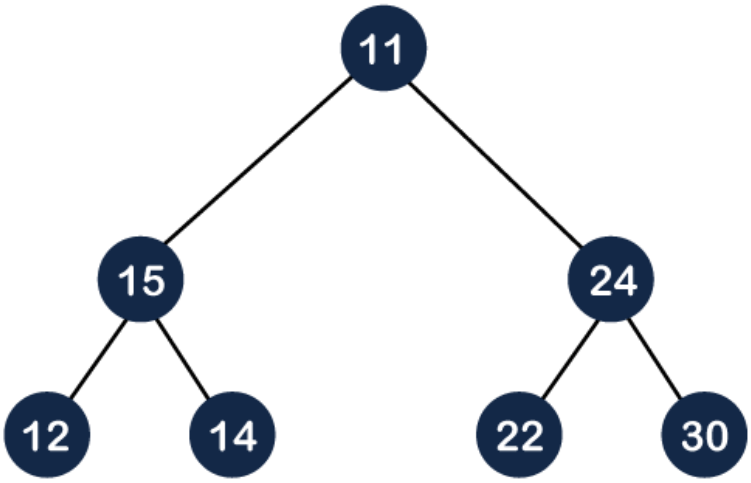
Min Heap: The value of the parent node should be less than or equal to either of its children.

Or

In other words, the min-heap can be defined as, for every node i, the value of node i is greater than or equal to its parent value except the root node. Mathematically, it can be defined as:

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

Let's understand the min-heap through an example.



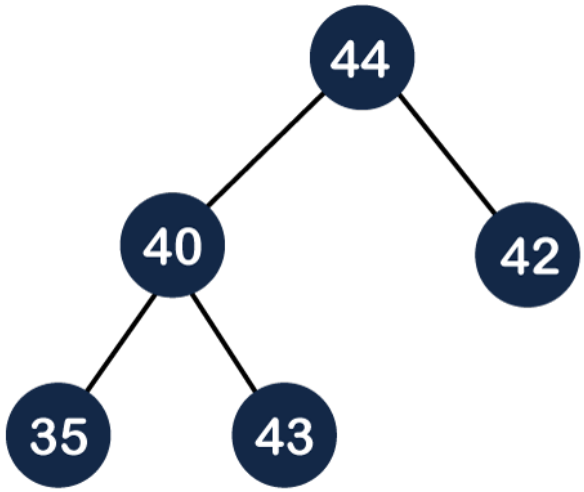
In the above figure, 11 is the root node, and the value of the root node is less than the value of all the other nodes (left child or a right child).

Max Heap: The value of the parent node is greater than or equal to its children.

Or

In other words, the max heap can be defined as for every node i; the value of node i is less than or equal to its parent value except the root node. Mathematically, it can be defined as:

A[Parent(i)] >= A[i]



The above tree is a max heap tree as it satisfies the property of the max heap. Now, let's see the array representation of the max heap.

Time complexity in Max Heap

The total number of comparisons required in the max heap is according to the height of the tree. The height of the complete binary tree is always logn; therefore, the time complexity would also be O(logn).

Algorithm of insert operation in the max heap.

```
1. // algorithm to insert an element in the max heap.
2. insertHeap(A, n, value)
3. {
4. n=n+1; // n is incremented to insert the new element
5. A[n]=value; // assign new value at the nth position
6. i = n; // assign the value of n to i
7. // loop will be executed until i becomes 1.
8. while(i>1)
9. {
10.             parent= floor value of i/2; // Calculating the floor value of i/2
11.// Condition to check whether the value of parent is less than the given node or not
12.             if(A[parent]<A[i])
13.             {
14.             swap(A[parent], A[i]);
15.             i = parent;
16.             }
17.             else
18.             {
19.             return;
20.             }
21.             }
22.             }
```

Let's understand the max heap through an example.

In the above figure, 55 is the parent node and it is greater than both of its child, and 11 is the parent of 9 and 8, so 11 is also greater than from both of its child. Therefore, we can say that the above tree is a max heap tree.

Insertion in the Heap tree

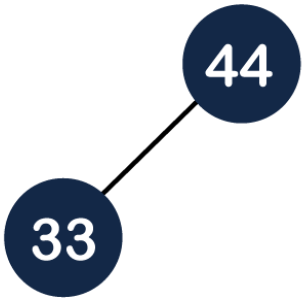
44, 33, 77, 11, 55, 88, 66

Suppose we want to create the max heap tree. To create the max heap tree, we need to consider the following two cases:

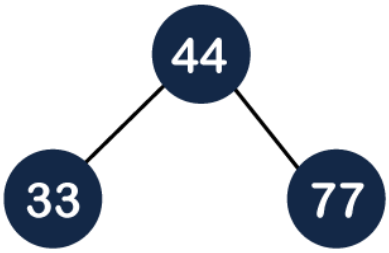
- First, we have to insert the element in such a way that the property of the complete binary tree must be maintained.
- Secondly, the value of the parent node should be greater than the either of its child.

Step 1: First we add the 44 element in the tree as shown below:

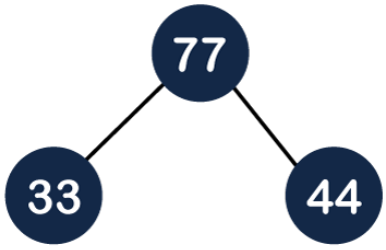
Step 2: The next element is 33. As we know that insertion in the binary tree always starts from the left side so 44 will be added at the left of 33 as shown below:



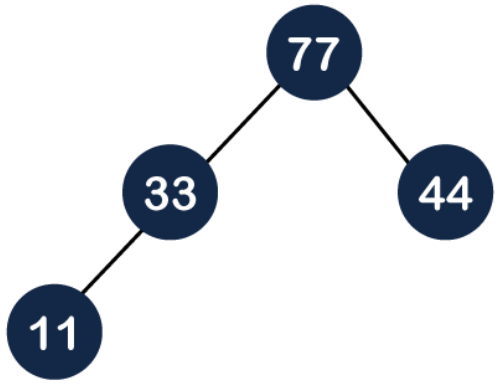
Step 3: The next element is 77 and it will be added to the right of the 44 as shown below:



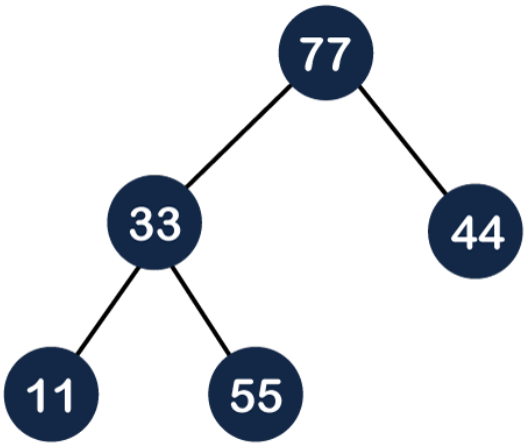
As we can observe in the above tree that it does not satisfy the max heap property, i.e., parent node 44 is less than the child 77. So, we will swap these two values as shown below:



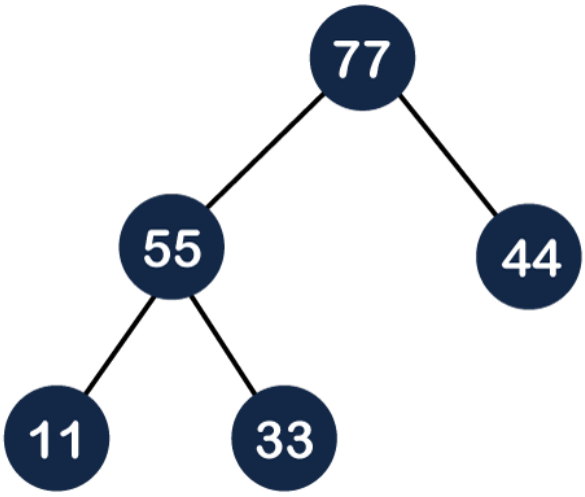
Step 4: The next element is 11. The node 11 is added to the left of 33 as shown below:



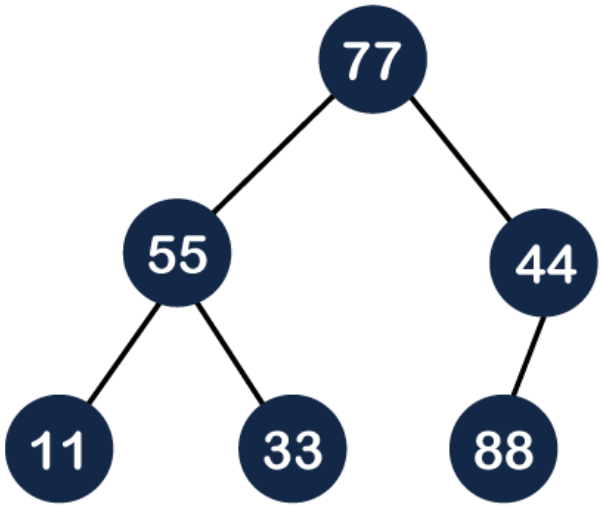
Step 5: The next element is 55. To make it a complete binary tree, we will add the node 55 to the right of 33 as shown below:



As we can observe in the above figure that it does not satisfy the property of the max heap because 33<55, so we will swap these two values as shown below:



Step 6: The next element is 88. The left subtree is completed so we will add 88 to the left of 44 as shown below:



As we can observe in the above figure that it does not satisfy the property of the max heap because 44<88, so we will swap these two values as shown below:

Again, it is violating the max heap property because 88>77 so we will swap these two values as shown below:

Step 7: The next element is 66. To make a complete binary tree, we will add the 66 element to the right side of 77 as shown below:

In the above figure, we can observe that the tree satisfies the property of max heap; therefore, it is a heap tree.

Deletion in Heap Tree

In Deletion in the heap tree, the root node is always deleted and it is replaced with the last element.

Let's understand the deletion through an example.

Step 1: In the above tree, the first 30 node is deleted from the tree and it is replaced with the 15 element as shown below:

Now we will heapify the tree. We will check whether the 15 is greater than either of its child or not. 15 is less than 20 so we will swap these two values as shown below:

Again, we will compare 15 with its child. Since 15 is greater than 10 so no swapping will occur.

Algorithm to heapify the tree

```
1. MaxHeapify(A, n, i)
2. {
3. int largest =i;
4. int l= 2i;
5. int r= 2i+1;
6. while(l<=n && A[l]>A[largest])
7. {
8. largest=l;
9. }
10.             while(r<=n && A[r]>A[largest])
11.{
12.             largest=r;
13.             }
14.             if(largest!=i)
15.             {
16.             swap(A[largest], A[i]);
17.             heapify(A, n, largest);
18.             }}
```

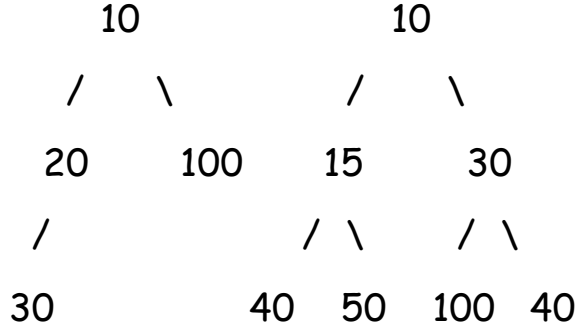
Binary Heap

A Binary Heap is a Binary Tree with following properties.

1) It's a complete tree (All levels are completely filled except possibly the last level and the last level has all keys as left as possible). This property of Binary Heap makes them suitable to be stored in an array.

2) A Binary Heap is either Min Heap or Max Heap. In a Min Binary Heap, the key at root must be minimum among all keys present in Binary Heap. The same property must be recursively true for all nodes in Binary Tree. Max Binary Heap is similar to MinHeap.

Examples of Min Heap:



How is Binary Heap represented?

A Binary Heap is a Complete Binary Tree. A binary heap is typically represented as an array.

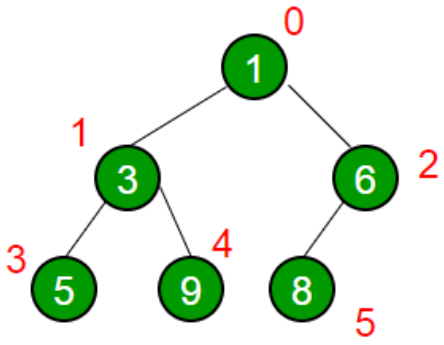
- The root element will be at Arr[0].
- Below table shows indexes of other nodes for the i^{th} node, i.e., Arr[i]:

·1)/2] ıs the parent node

*i)+1] ıs the left child node

*i)+2] ıs the right child node

The traversal method use to achieve Array representation is **Level Order**



1	3	6	5	9	8
0	1	2	3	4	5

Please refer Array Representation Of Binary Heap for details.

Applications of Heaps:

1) 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. Binomial 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 and Prim's Minimum Spanning Tree.

4) Many problems can be efficiently solved using Heaps. See following for example.

a) K'th Largest Element in an array.

b) Sort an almost sorted array/

c) Merge K Sorted Arrays.

Operations on Min Heap:

1) getMini(): It returns the root element of Min Heap. Time Complexity of this operation is O(1).

2) extractMin(): Removes the minimum element from MinHeap. Time Complexity of this Operation is O(Logn) as this operation needs to maintain the heap property (by calling heapify()) after removing root.

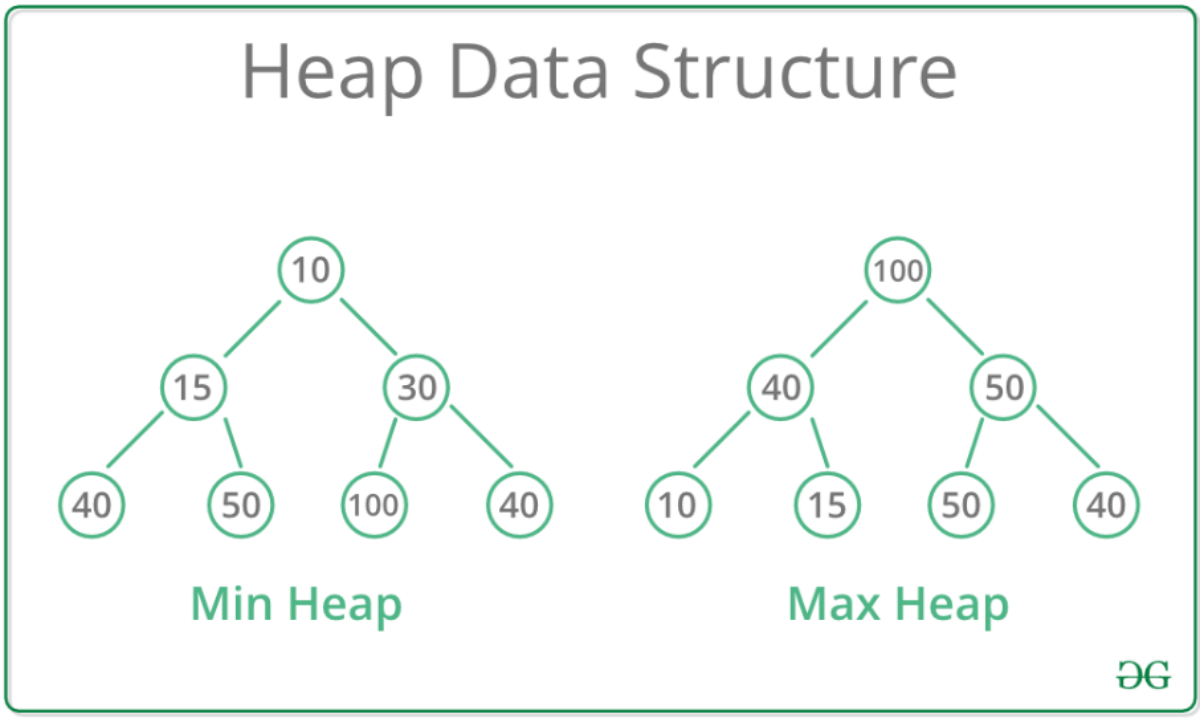
3) decreaseKey(): Decreases value of key. The time complexity of this operation is O(Logn). If the decreases key value of a node is greater than the parent of the node, then we don't need to do anything. Otherwise, we need to traverse up to fix the violated heap property.

4) insert(): Inserting a new key takes O(Logn) time. We add a new key at the end of the tree. IF new key is greater than its parent, then we don't need to do anything. Otherwise, we need to traverse up to fix the violated heap property.

5) delete(): Deleting a key also takes O(Logn) time. We replace the key to be deleted with minum infinite by calling decreaseKey(). After decreaseKey(), the minus infinite value must reach root, so we call extractMin() to remove the key.

Unit 1

A Heap is a special Tree-based data structure in which the tree is a complete binary tree.



Operations of Heap Data Structure:

- **Heapify:** a process of creating a heap from an array.
- **Insertion:** process to insert an element in existing heap time complexity O(log N).
- **Deletion:** deleting the top element of the heap or the highest priority element, and then organizing the heap and returning the element with time complexity O(log N).
- **Peek:** to check or find the most prior element in the heap, (max or min element for max and min heap).

Types of Heap Data Structure

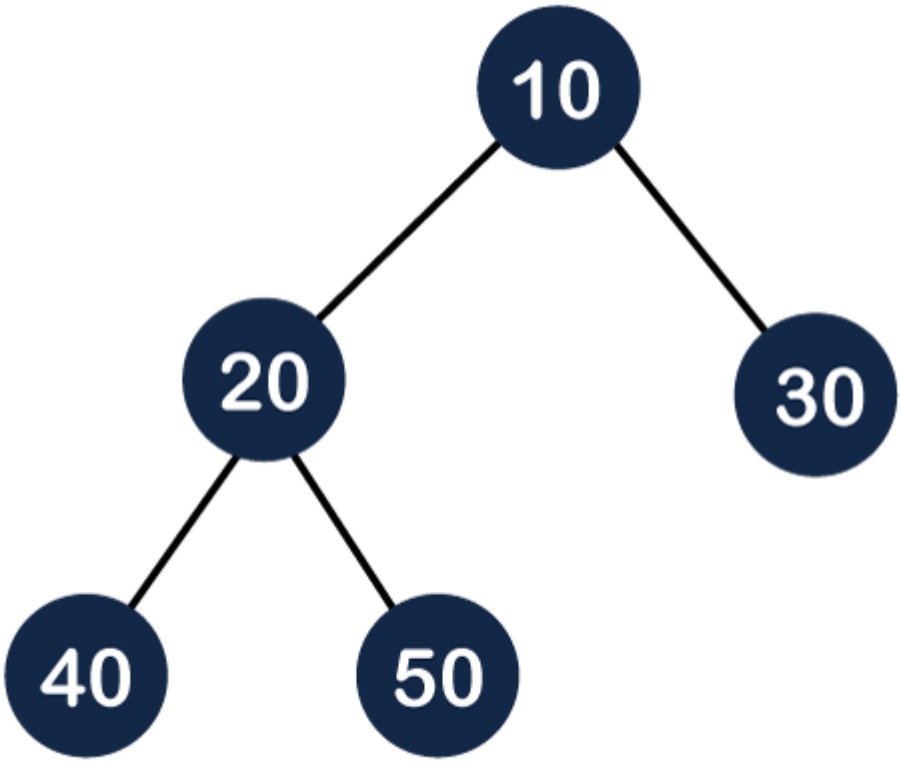
Generally, Heaps can be of two types:

1. **Max-Heap**: In a Max-Heap the key present at the root node must be greatest among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.
2. **Min-Heap**: In a Min-Heap the key present at the root node must be minimum among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.

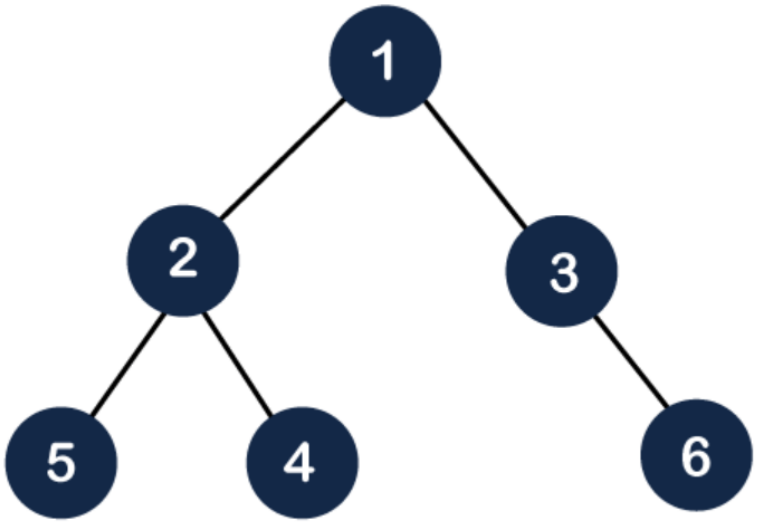
What is a complete binary tree?

A complete binary tree is a binary tree in which all the levels except the last level, i.e., leaf node should be completely filled, and all the nodes should be left-justified.

Let's understand through an example.

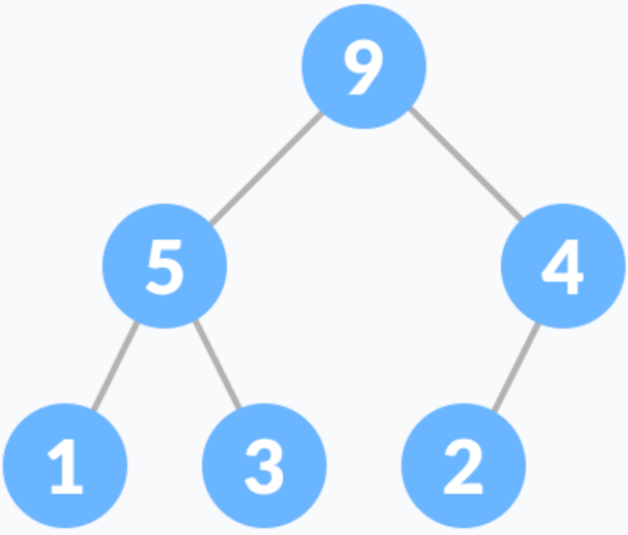


In the above figure, we can observe that all the internal nodes are completely filled except the leaf node; therefore, we can say that the above tree is a complete binary tree.

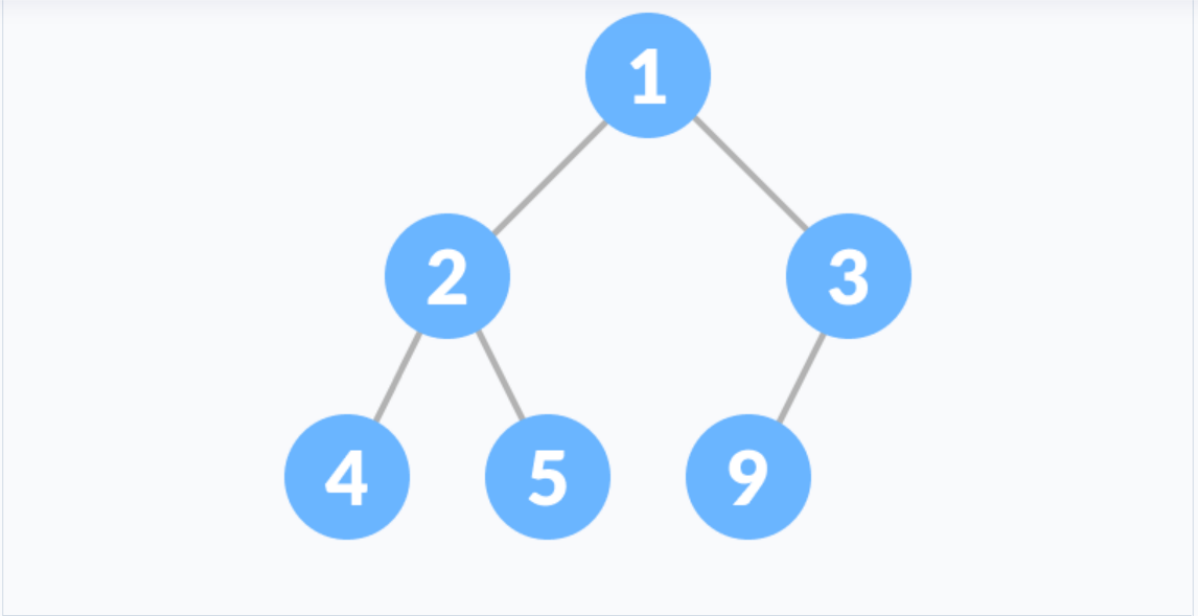


The above figure shows that all the internal nodes are completely filled except the leaf node, but the leaf nodes are added at the right part; therefore, the above tree is not a complete binary tree.

> 📝 Note: The heap tree is a special balanced binary tree data structure where the root node is compared with its children and arrange accordingly.



Max-heap



Min-heap

This type of data structure is also called a **binary heap.**

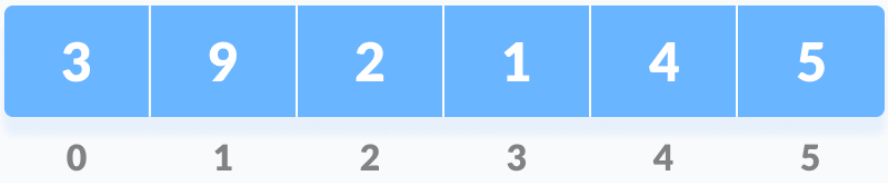
Heap Operations

Some of the important operations performed on a heap are described below along with their algorithms.

Heapify

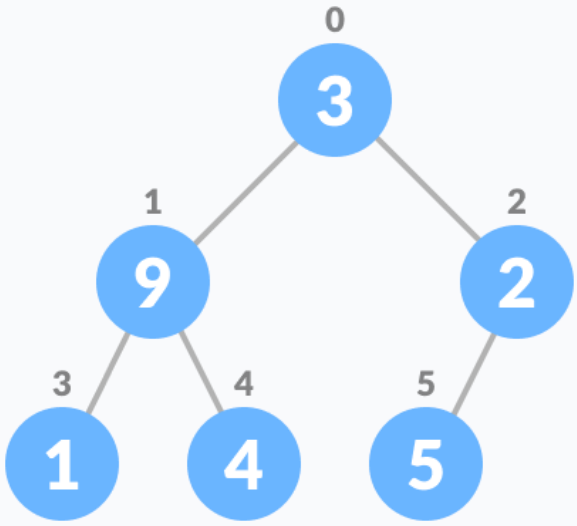
Heapify is the process of creating a heap data structure from a binary tree. It is used to create a Min-Heap or a Max-Heap.

1. Let the input array be



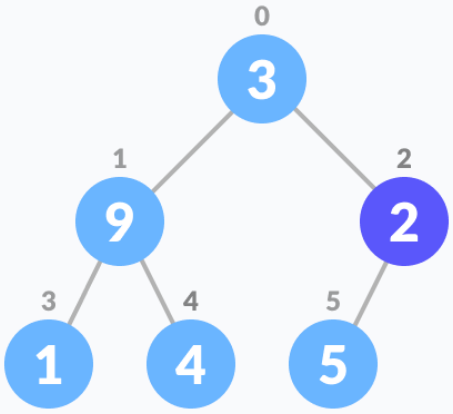
Initial Array

2. Create a complete binary tree from the array



Complete binary tree

3. Start from the first index of non-leaf node whose index is given by `n/2 - 1`.



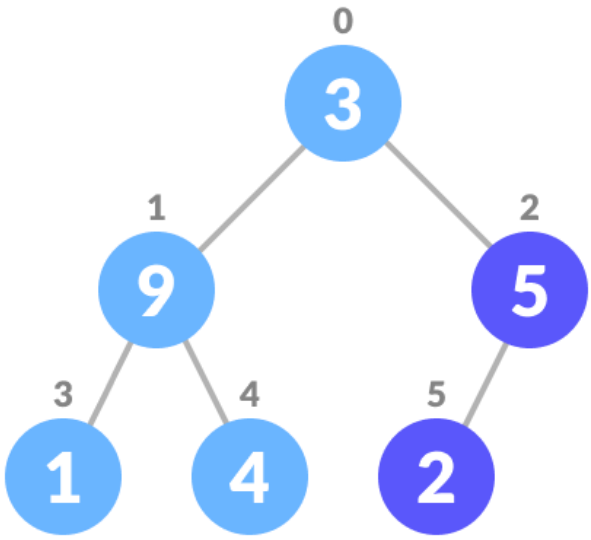
4. Start from the first on leaf node
5. Set current element `i` as `largest`.
6. The index of left child is given by `2i + 1` and the right child is given by `2i + 2`.

   If `leftChild` is greater than `currentElement` (i.e. element at `ith` index), set `leftChildIndex` as largest.

If `rightChild` is greater than element in `largest`,

set `rightChildIndex` as `largest`.

7. Swap `largest` with `currentElement`

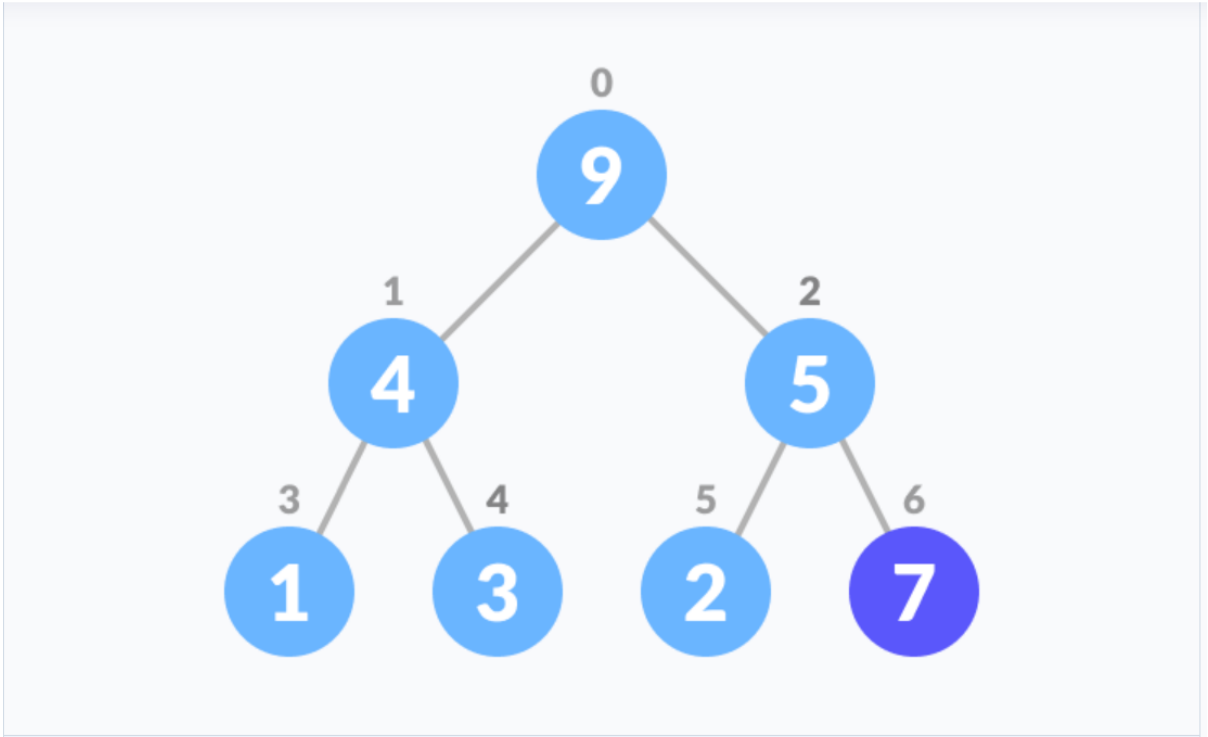


8. Swap if necessary
9. Repeat steps 3-7 until the subtrees are also heapified.

Insert Element into Heap

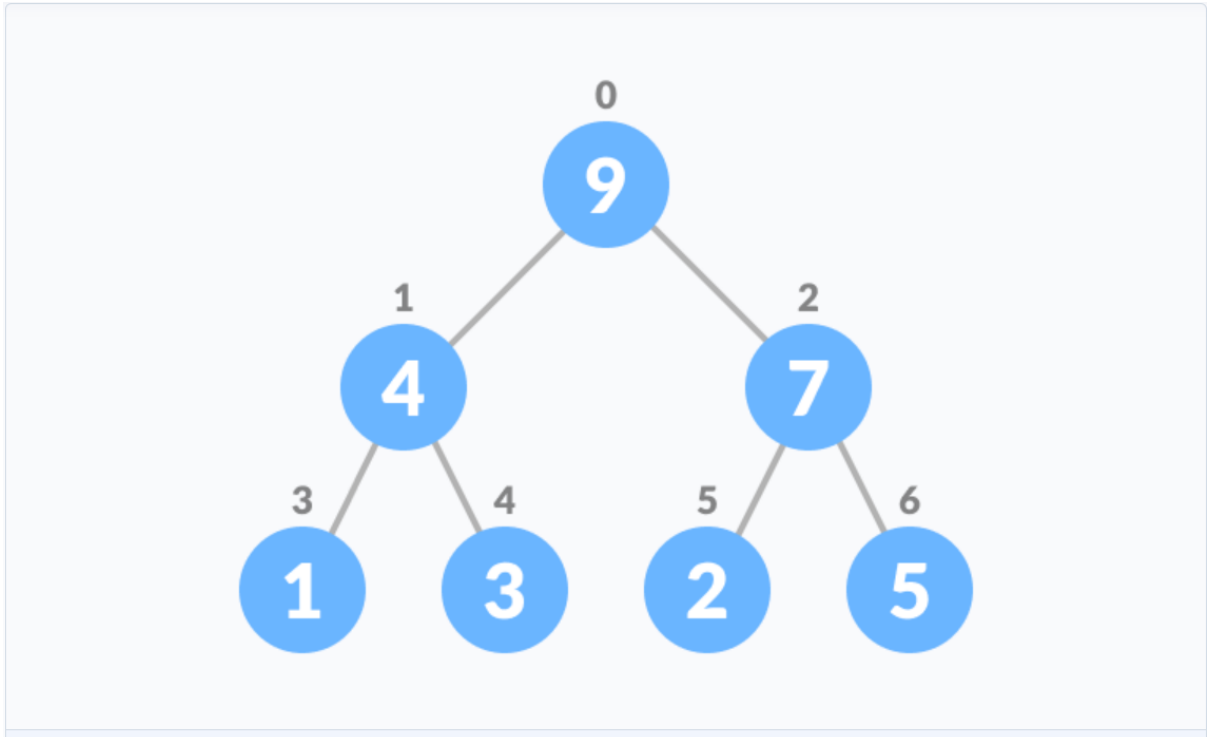
Algorithm for insertion in Max Heap

1. Insert the new element at the end of the tree.



Insert at the end

2. Heapify the tree.



Heapify the array

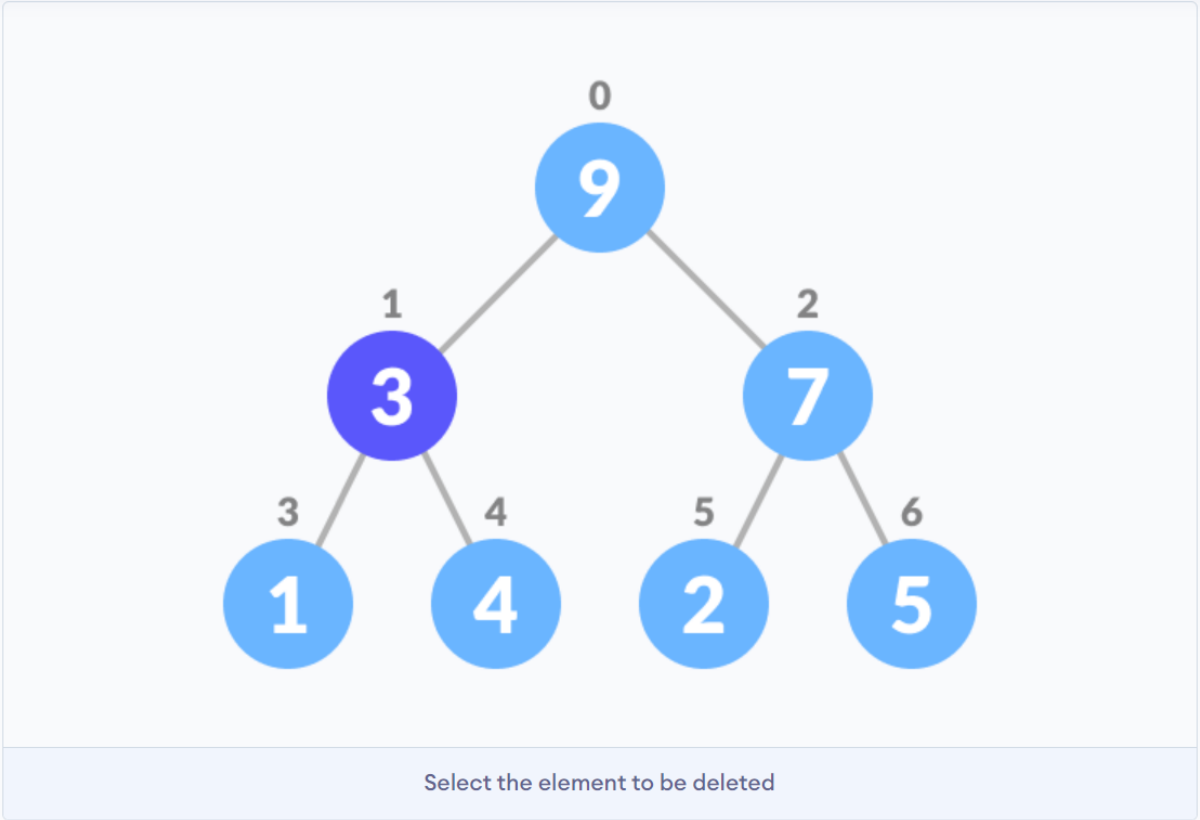
For Min Heap, the above algorithm is modified so that parentNode is always smaller than newNode.

Delete Element from Heap

Algorithm for deletion in Max Heap

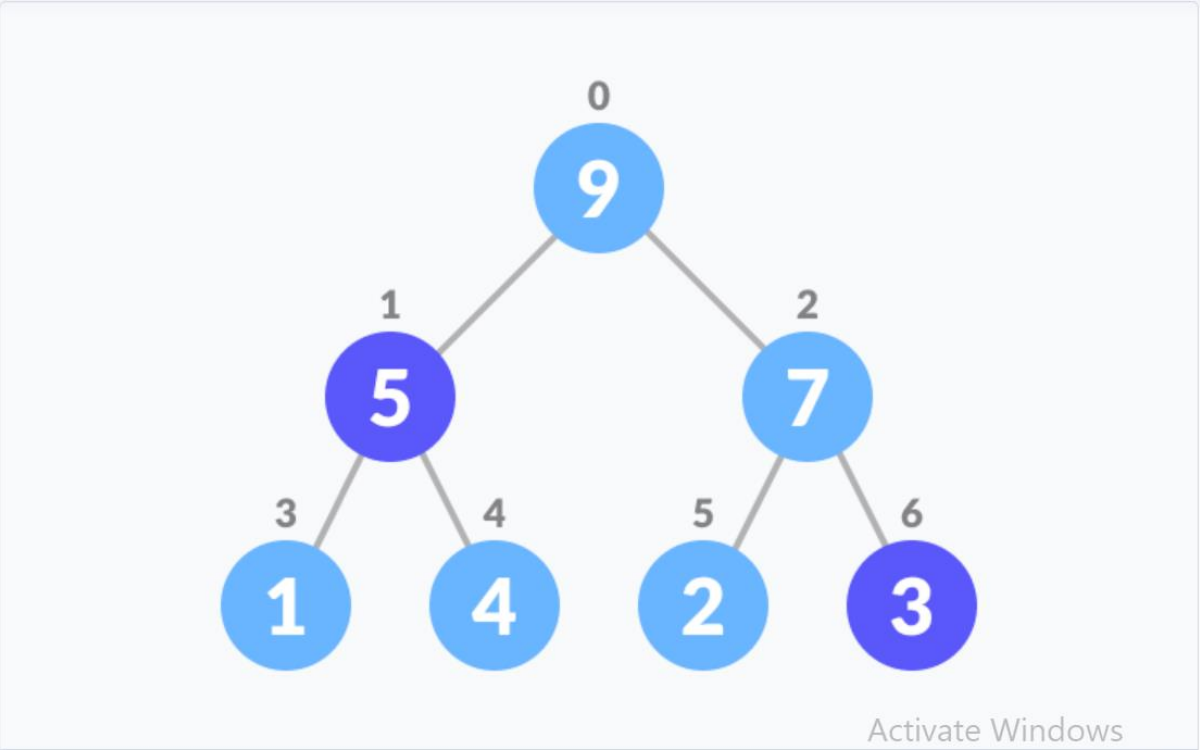
```
If nodeToBeDeleted is the leafNode
  remove the node
Else swap nodeToBeDeleted with the lastLeafNode
  remove noteToBeDeleted

heapify the array
```

1. Select the element to be deleted.

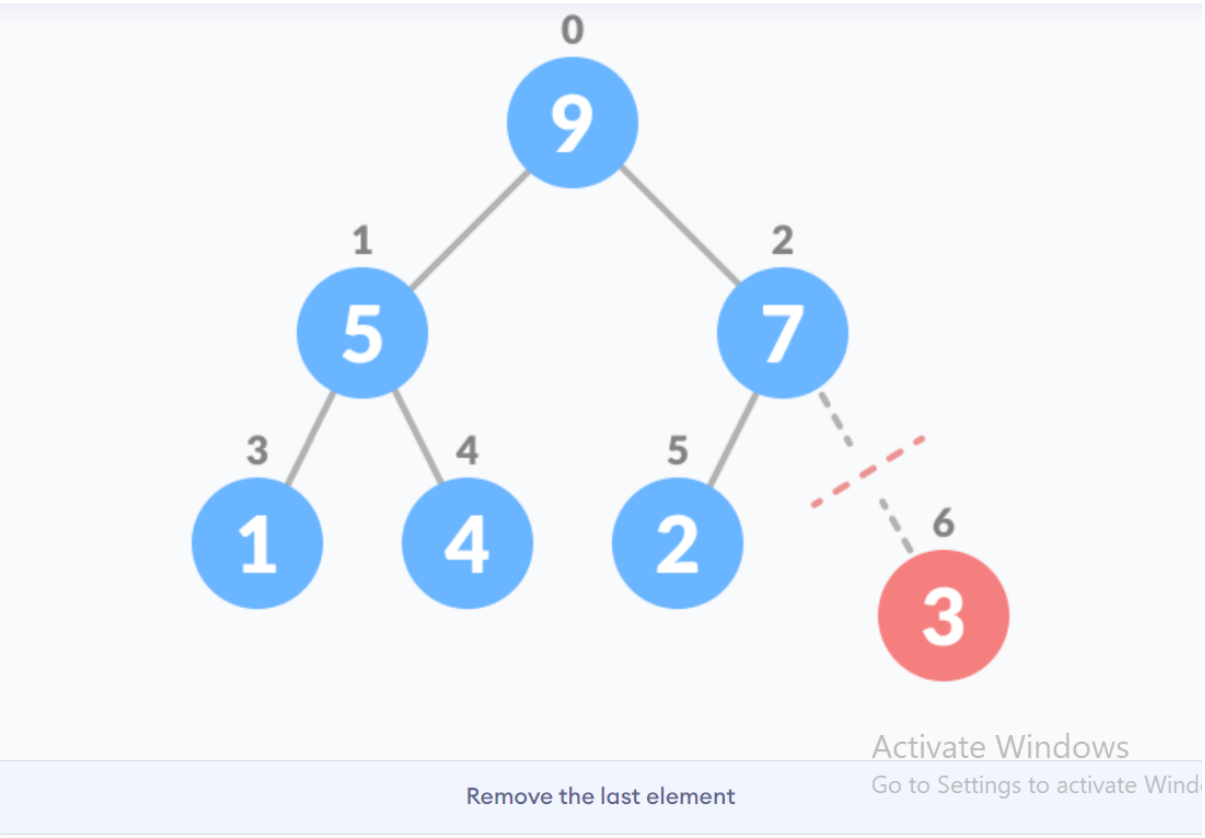


Select the element to be deleted

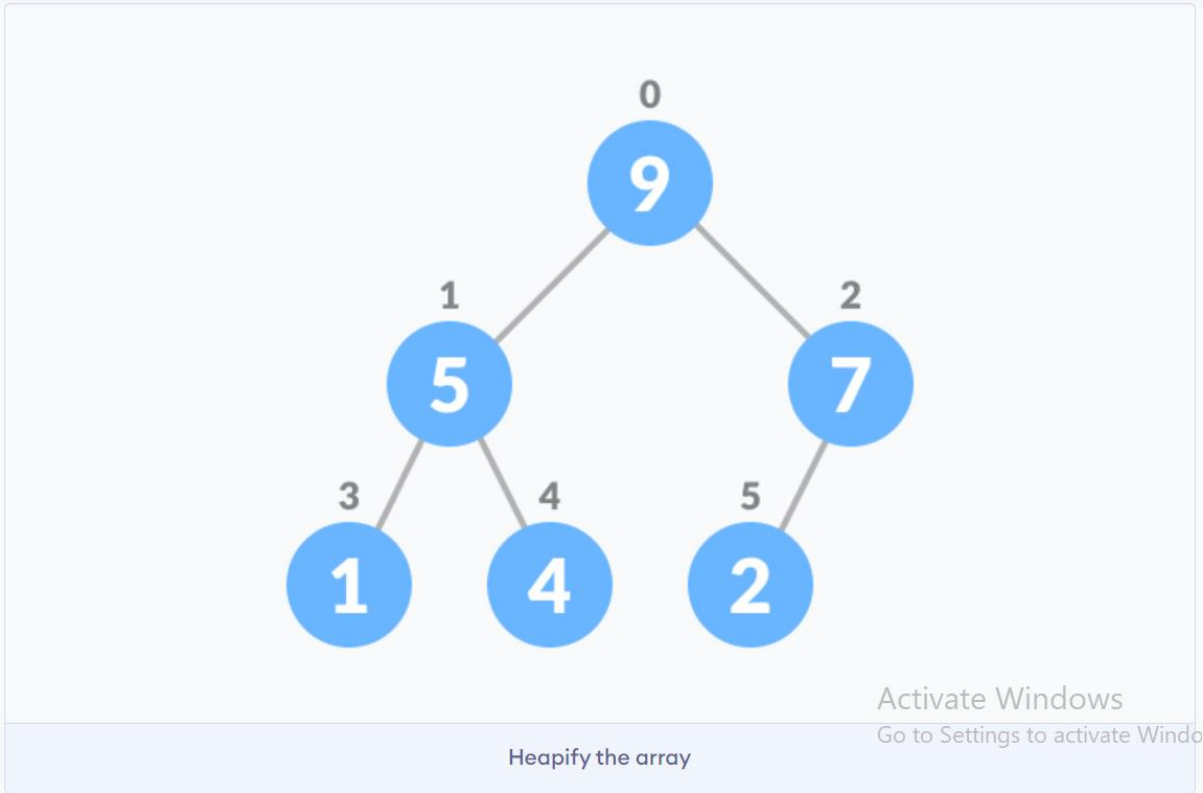
2. Swap it with the last element.



3. Remove the last element.



4. Heapify the tree.



For Min Heap, above algorithm is modified so that both childNodes are greater smaller than currentNode.

Peek (Find max/min)

Peek operation returns the maximum element from Max Heap or minimum element from Min Heap without deleting the node.

For both Max heap and Min Heap

```
return rootNode
```

Extract-Max/Min

Extract-Max returns the node with maximum value after removing it from a Max Heap whereas Extract-Min returns the node with minimum after removing it from Min Heap.

Priority Queue:

Priority Queue is similar to queue where we insert an element from the back and remove an element from front, but with a difference that the logical order of elements in the priority queue depends on the priority of the elements. The element with highest priority will be moved to the front of the queue and one with lowest priority will move to the back of the queue. Thus it is possible that when you enqueue an element at the back in the queue, it can move to front because of its highest priority.

Example:

Let's say we have an array of 5 elements : {4, 8, 1, 7, 3} and we have to insert all the elements in the max-priority queue.

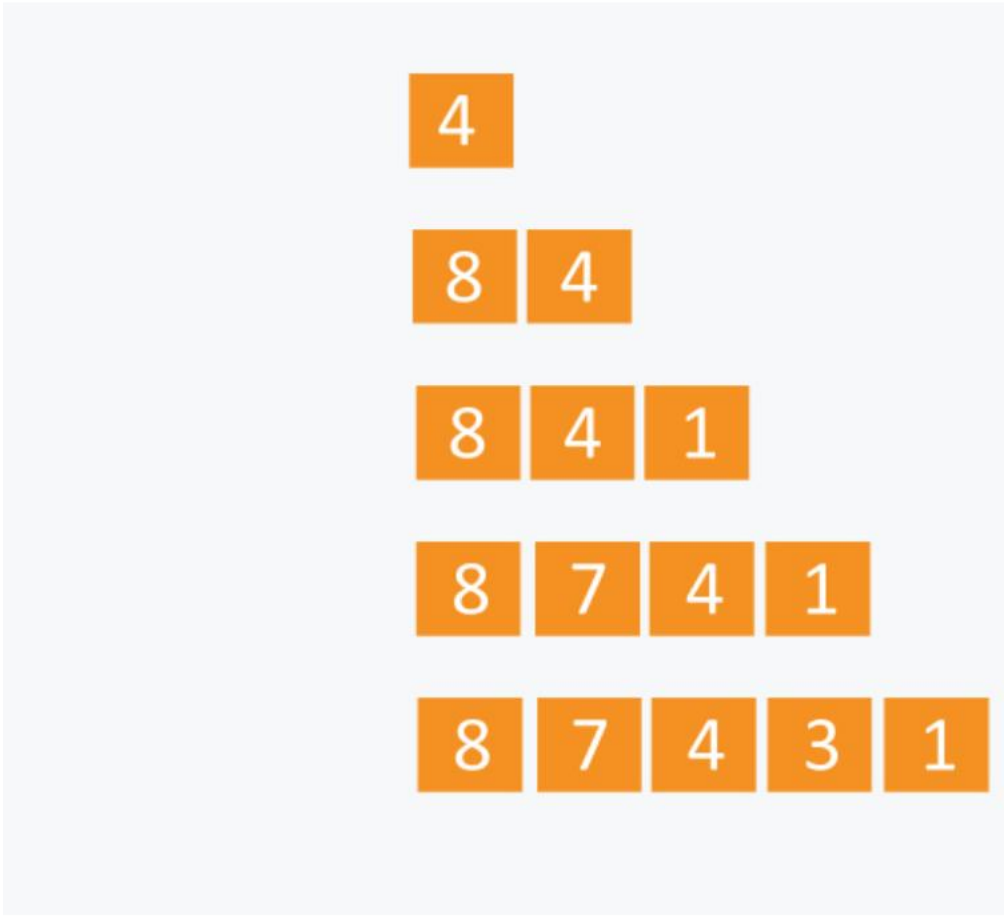
First as the priority queue is empty, so 4 will be inserted initially.

Now when 8 will be inserted it will moved to front as 8 is greater than 4.

While inserting 1, as it is the current minimum element in the priority queue, it will remain in the back of priority queue.

Now 7 will be inserted between 8 and 4 as 7 is smaller than 8.

Now 3 will be inserted before 1 as it is the 2^{nd} minimum element in the priority queue. All the steps are represented in the diagram below:



We can think of many ways to implement the priority queue.

Efficient Approach:
We can use heaps to implement the priority queue. It will take $O(logN)$ time to insert and delete each element in the priority queue.

Based on heap structure, priority queue also has two types max- priority queue and min - priority queue.

Let's focus on Max Priority Queue.

What is heap order property?
The heap ordering property states that **the parent always precedes the children**. There is no precedence required between the children. The precedence must be an order realtionship

Heap order property

- Every node is less than or equal to its children
- Or every node is greater than or equal to its children