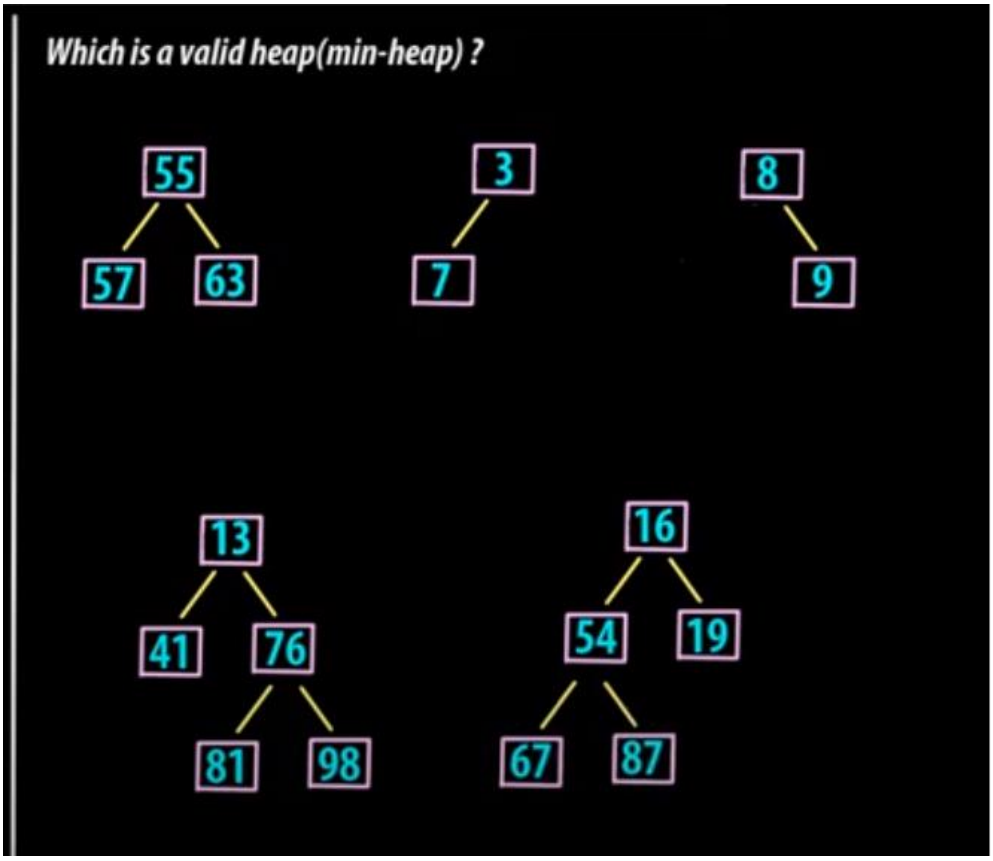
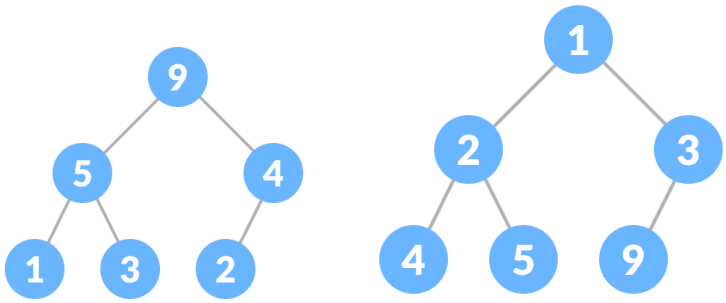
A complete binary tree is a special binary tree in which

- every level, except possibly the last, is filled
- all the nodes are as far left as possible



Heap Property is the property of a node in which

- (**for max heap)** key of each node is always greater than its child node/s and the key of the root node is the largest among all other nodes;



- **(for min heap)** key of each node is always smaller than the child node/s and the key of the root node is the smallest among all other nodes.

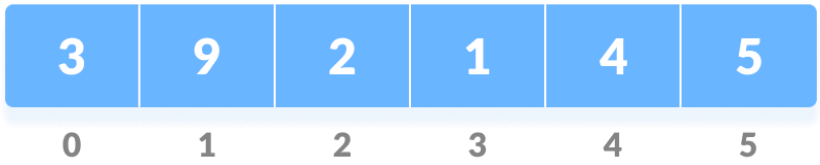
Heap Operations

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

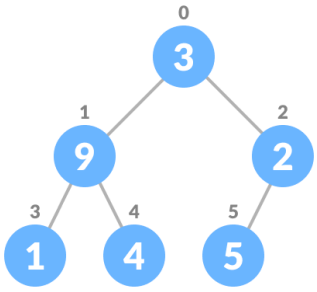
Heapify

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.

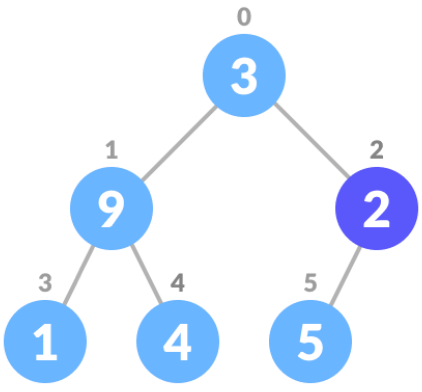
1. Let the input array be



2. Create a complete binary tree from the array



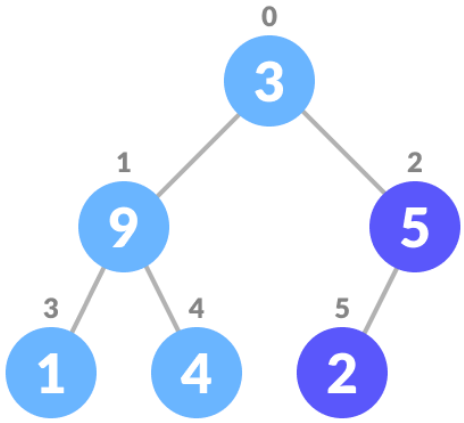
3. Start from the first index of non-leaf node whose index is given **by n/2 - 1.**



1. Set current element i as **largest**.

2. 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**.

3. Swap largest with **currentElement**



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]:

Arr[(i-1)/2]	Returns the parent node
Arr[(2*i)+1]	Returns the left child node
Arr[(2*i)+2]	Returns the right child node

Repeat steps 3-7 until the subtrees are also heapified.

Algorithm

```
Heapify(array, size, i)
  set i as largest
  leftChild = 2i + 1
  rightChild = 2i + 2

  if leftChild > array[largest]
    set leftChildIndex as largest
  if rightChild >array[largest]
    set rightChildIndex as largest
swap array[i] and array[largest]


MaxHeap(array, size)
  loop from the first index of non-leaf node down to zero
    call heapify
```

For Min-Heap, both leftChild and rightChild must be smaller than the parent for all nodes.rray[i] and array[largest]

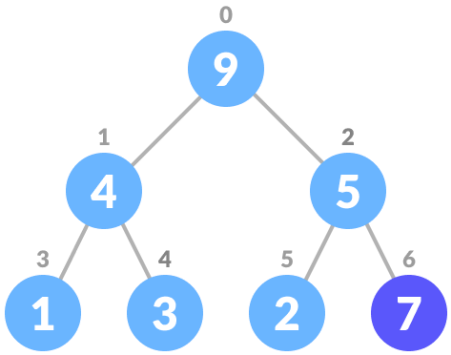
Insert Element into Heap

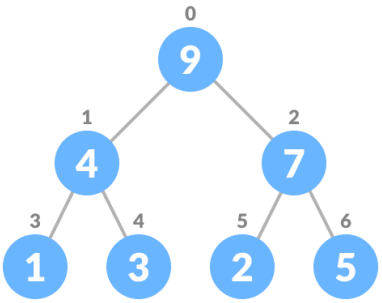
Algorithm for insertion in Max Heap

```
If there is no node,
  create a newNode.
else (a node is already present)
  insert the newNode at the end (last node from left to right.)

heapify the array
```

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

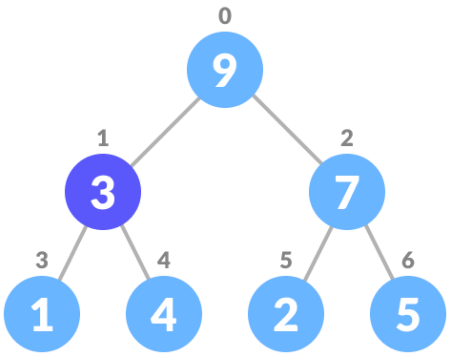




2. Heapify the tree.

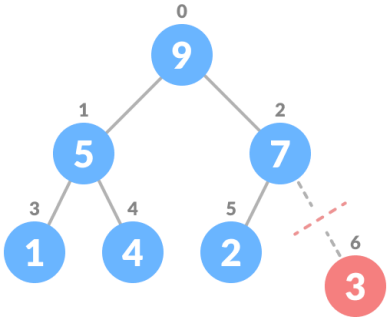
For Min Heap, the above algorithm is modified so that parentNode is always smaller than newNode.

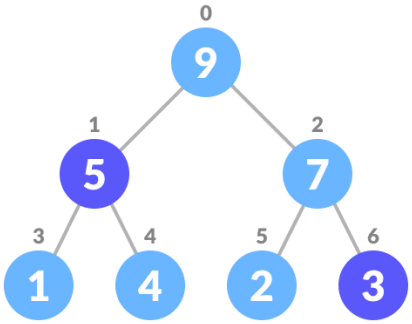
Delete Element from Heap
Algorithm for deletion in Max Heap
If nodeToBeDeleted is the leafNode
  remove the node
Else swap nodeToBeDeleted with the lastLeafNode
  remove noteToBeDeleted



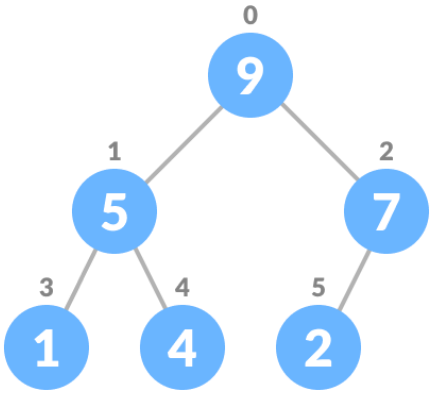
heapify the array

1. Select the element to be deleted.
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.

// Max-Heap data structure in C++

```
#include <iostream>
#include <vector>
using namespace std;

void swap(int *a, int *b)
{
  int temp = *b;
  *b = *a;
  *a = temp;
}
void heapify(vector<int> &hT, int i)
{
  int size = hT.size();
  int largest = i;
  int l = 2 * i + 1;
  int r = 2 * i + 2;
  if (l < size && hT[l] > hT[largest])
    largest = l;
  if (r < size && hT[r] > hT[largest])
    largest = r;

  if (largest != i)
  {
    swap(&hT[i], &hT[largest]);
    heapify(hT, largest);
  }
}
void insert(vector<int> &hT, int newNum)
{
  int size = hT.size();
  if (size == 0)
  {
    hT.push_back(newNum);
  }
  else
  {
    hT.push_back(newNum);
    for (int i = size / 2 - 1; i >= 0; i--)
```

```
    {
      heapify(hT, i);
    }
  }
}
void deleteNode(vector<int> &hT, int num)
{
  int size = hT.size();
  int i;
  for (i = 0; i < size; i++)
  {
    if (num == hT[i])
      break;
  }
  swap(&hT[i], &hT[size - 1]);

  hT.pop_back();
  for (int i = size / 2 - 1; i >= 0; i--)
  {
    heapify(hT, i);
  }
}
void printArray(vector<int> &hT)
{
  for (int i = 0; i < hT.size(); ++i)
    cout << hT[i] << " ";
  cout << "\n";
}

int main()
{
  vector<int> heapTree;

  insert(heapTree, 3);
  insert(heapTree, 4);
  insert(heapTree, 9);
  insert(heapTree, 5);
  insert(heapTree, 2);

  cout << "Max-Heap array: ";
  printArray(heapTree);

  deleteNode(heapTree, 4);

  cout << "After deleting an element: ";
```

```
printArray(heapTree);}
```