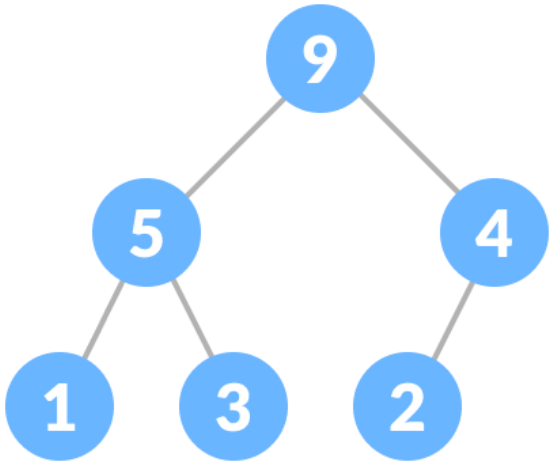
Heap Data Structure

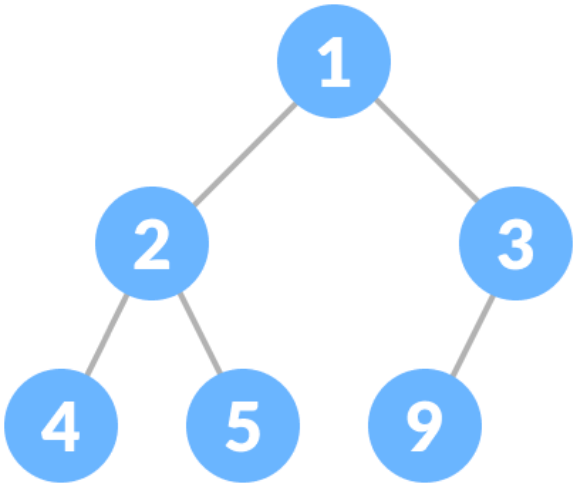
Heap data structure is a complete binary tree that satisfies the heap property, where any given node is

always greater than its child node/s and the key of the root node is the largest among all other nodes. This property is also called max heap property.

always smaller than the child node/s and the key of the root node is the smallest among all other nodes. This property is also called min heap property.



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.

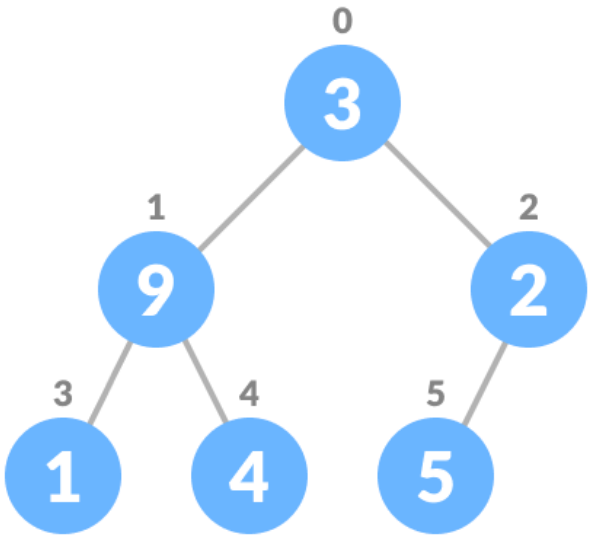
Heapify

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.

Let the input array be

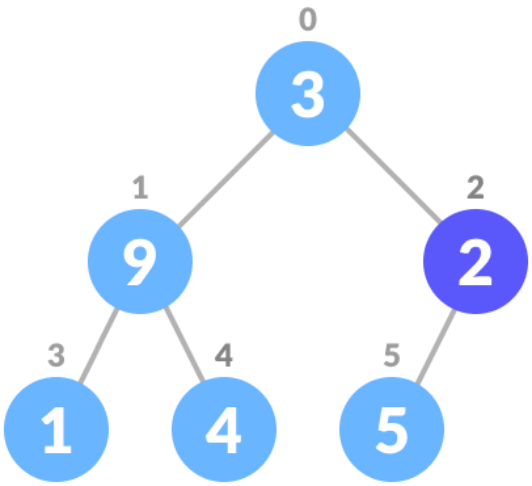
Initial Array

Create a complete binary tree from the array



Complete binary tree

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

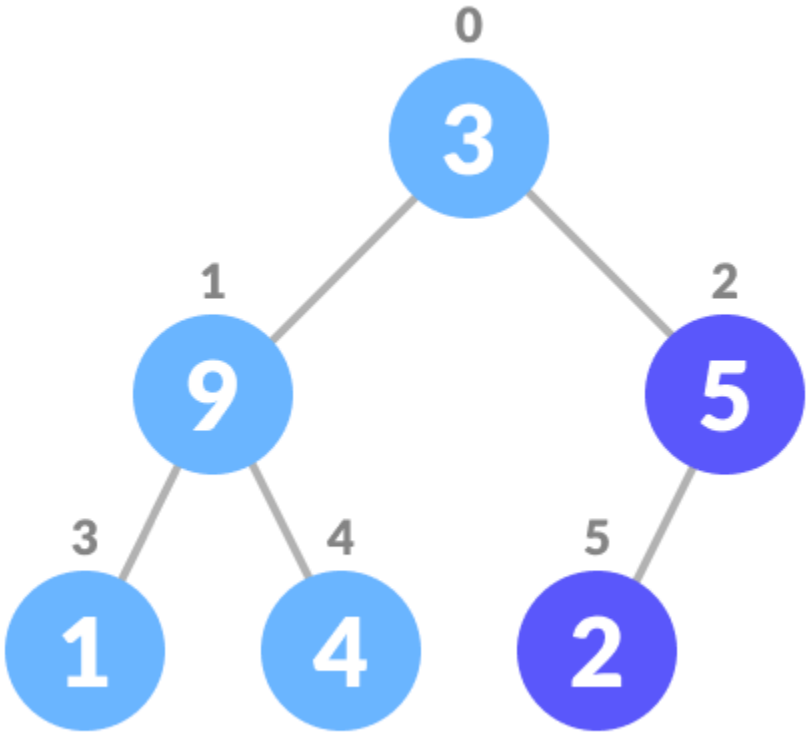


Start from the first on leaf node

Set current element i as largest.

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.

Swap largest with currentElement



Swap if necessary

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

Below is the full code including all the heap operations.

Heap is implemented using an array, but we have use vectors for ease. Not to get into the process of expanding and shrinking array manually.

getParent(int child)

This method calculates the parent index of a given child node in the heap. For any child node at index child, its parent is located at index (child - 1) / 2. The method checks if the child index is 0 (the root of the heap), in which case it returns -1, indicating no parent. Otherwise, it returns the calculated parent index. This calculation is based on the properties of a binary heap, where each node except the root has a parent node.

getLeft(int parent)

The method determines the left child of a given parent node in the heap. Given a parent node at index parent, its left child is located at 2 * parent + 1. The method checks if this calculated index is within the bounds of the heap (i.e., less than the size of htree). If it is within bounds, the index of the left child is returned; otherwise, -1 is returned, indicating the absence of a left child.

getRight(int parent)

Similar to getLeft, this method finds the right child of a given parent node. For a parent at index parent, the right child is at 2 * parent + 2. The method checks if this index is within the bounds of the heap. If so, it returns the index of the right child; if not, it returns -1, indicating no right child is present.

swap(int *a, int *b)

This method swaps the values of two integers. It takes pointers to these integers as arguments. A temporary variable is used to hold the value of *a while *a is set to *b, and then *b is set to the value stored in the temporary variable. This is a standard swapping technique in C++.

heapifyUp(int node)

heapifyUp is a key method for maintaining the heap property after inserting a new element. Starting from the node at index node, the method compares the node with its parent. If the node's value is less than its parent's, they are swapped. This process is repeated recursively up the heap until the node is larger than its parent or it becomes the root node. This ensures that the smallest element is always at the root of the heap.

heapifyDown(int node)

This method is used to maintain the heap property after a deletion. It starts at the given node and compares it with its left and right children. The smallest of the three is determined. If the node is larger than the smallest child, they are swapped. This process continues recursively down the heap. The method ensures that after a deletion, the heap property (smallest element at the root) is restored.

insert(int value)

To insert a new value into the heap, this method adds the value at the end of the htree vector. If the heap was previously empty, this is the only step required. However, if there are other elements, the method calls heapifyUp starting from the last index. This ensures that the newly added element is moved to its correct position in the heap, maintaining the heap property.

del(int value)

The delete operation first locates the value to be deleted in the heap. Once found, it swaps this value with the last element in the heap and then removes the last element (which is now the value to be deleted). After this, heapifyDown is called from the index where the deleted value was located, to ensure that the heap property is maintained after the deletion.

display()

This method iterates through the htree vector and prints each element. It provides a simple way to visually inspect the current state of the heap.

These methods collectively enable the heap class to perform basic operations like insertion, deletion, and display, while always maintaining the heap property - the fundamental characteristic of a binary heap data structure.

Code:

```
#include<iostream>
#include <vector>
using namespace std;
class heap {
        public:
                vector <int> htree;
                int getParent(int child)
                {
                        int p=(child-1)/2;
```

```
        if(child==0)
        {
                return -1;
        }
        else
        {
                return p;
        }
}

int getLeft(int parent)
{
        int child= 2*parent+1;
        if(child<htree.size())
        {
                return child;
        }
        else
        {
                return -1;
        }
}

int getRight(int parent)
{
        int child= 2*parent+2;
        if(child<htree.size())
        {
                return child;
        }
        else
        {
                return -1;
        }
}

void swap(int *a, int *b) {
```

```
	int temp=*a;
	*a=*b;
	*b=temp;
}

void heapifyUp(int node) {
	int parent=getParent(node);
	if(node >=0 && parent >=0 && htree[parent] > htree[node])
	{
		swap(&htree[node], &htree[parent]);
		heapifyUp(getParent(node));
	}
}

void heapifyDown(int node)
{
	int lchild= getLeft(node);
	int rchild= getRight(node);
	int smallest;
	if(lchild >=0 && rchild>=0)
	{
		if(htree[lchild] < htree[rchild])
		{
			smallest=lchild;
		}
		else if(htree[rchild] < htree[lchild])
		{
			smallest=rchild;
		}
		if(smallest>0)
		{
			swap(&htree[smallest], &htree[node]);
			heapifyDown(smallest);
		}
	}
}
```

```
        void insert(int value) {
            int size= htree.size();
            if(size==0) {
                htree.push_back(value);
            } else {
                htree.push_back(value);
                heapifyUp(htree.size()-1);
            }
        }

        void del(int value) {
            int index=-1;
            for(int i=0; i<htree.size(); i++)
            {
                if(htree[i]==value)
                {
                    index=i;
                    break;
                }
            }
            if(index!=-1)
            {
                swap(&htree[index], &htree[htree.size()-1]);
                htree.pop_back();
                heapifyDown(index);
            }
        }

        void display() {
            for(int i=0; i<htree.size(); i++) {
                cout<<htree[i]<<"  ";
            }
            cout<<endl;
        }

};
int main() {
```

```
        heap h;
        h.insert(5);
        h.insert(3);
        h.insert(8);
        h.insert(2);
        h.insert(7);
        h.display();
        h.del(5);
        h.del(3);
        h.display();
}
```

Output:

```
2  3  8  5  7
2  7  8
```

Note: Practice Max heap on your own, only conditions are changed. In max heap the max element should be on the top.