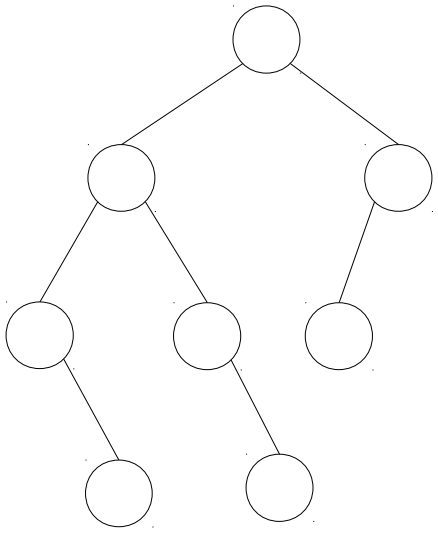
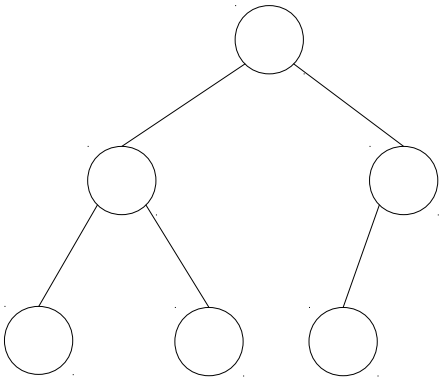
Binary Trees

I. Introduction

A **binary tree** is a structure where nodes have a parent-child relationship. Each node can have up to two children and only one parent. The root is the only node that doesn't have a parent. Nodes that have no children are called leaves. An example can be shown in the figure below



A binary tree is called **complete** if all of its levels are complete apart from the last one where some of the leaves on the right might be missing. A complete binary tree is shown below.



The distance of the root to the leaves (or in other words the total number of level that the tree has) is called the **depth** of the tree.

A complete binary tree of depth n contains exponentially many children, roughly 2^n. Thus, in order to travel in a path from the root to the leaves in a tree of n nodes we require time O(logn).

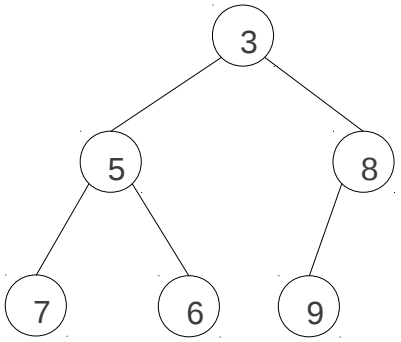
II. Implementation

Binary trees can be implemented with arrays and with the class BSTNode. For the array implementation, the tree needs to be complete (otherwise it is inefficient and problematic).

For the array implementation, the root is placed in position 0 of the array and then we continue by putting elements in the array starting from top to bottom and, within the same level, from left to right. The tree in the figure below can be represented with an array as follows:

Array representation:

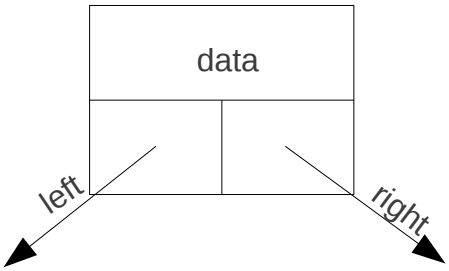
[3 5 8 7 6 9]



For the array representation, to go from a node indexed i to its parent, one needs to visit node indexed (i-1)/2. Its left and right children are nodes indexed 2i+1 and 2i+2 respectively.

The class BSTNode is as follows:

```
class BSTNode{
        public:
                int data;
                BSTNode* left;
                BSTNode* right;
                BSTNode(){left = right = NULL};
};
```



Each object of the class represents a node in the tree. The two pointers left and right point to the left and right child respectively (if the child does not exist the pointer points to NULL).

III. Priority Queues – Heaps

A priority queue is an abstract data type that provides methods void push(int) and int pop(), where pop extracts always the minimum (maximum) element of the priority queue.

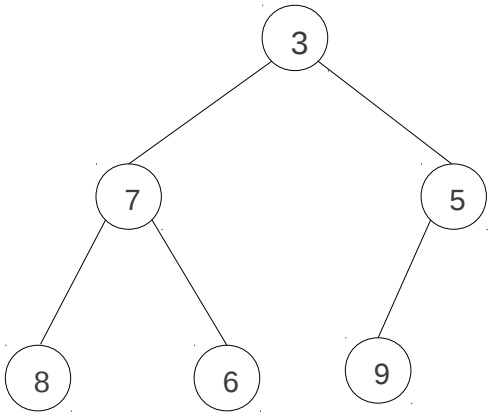
Priority queues are usually implemented with heaps. A heap is a complete binary tree where each node is smaller than its children. Heaps are implemented with arrays in the way that we saw above.

Let's describe how push and pop work:

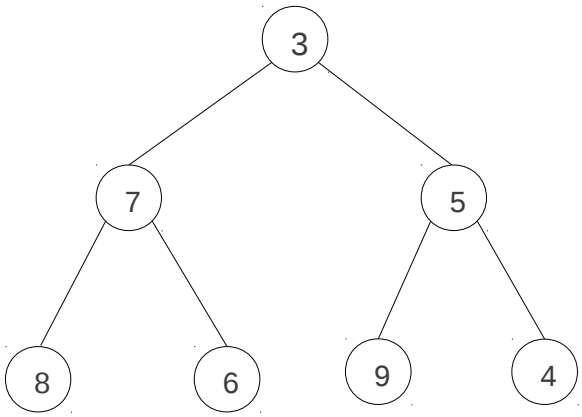
When we want to push an element in the heap, we put it last and then compare it with its ancestors all the way up in the path to the root until we locate its correct position. As long as the element is smaller

than its parent it continues its path upward.
In order to pop from the list, we mark the root as the element to be poped and then move the last element to the root. This is potentially a large element so it needs to move all the way down towards a path to the leaves until we locate its correct position. To move downwards we first compare the value of its two children to find the minimum one and then compare it with the minimum. If it is larger then it is swapped with the minimum child.
Since push and pop
Push and pop are illustrated below:

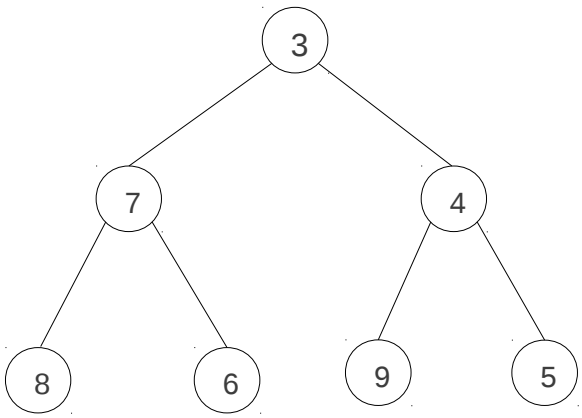
For a heap that looks like the figure below



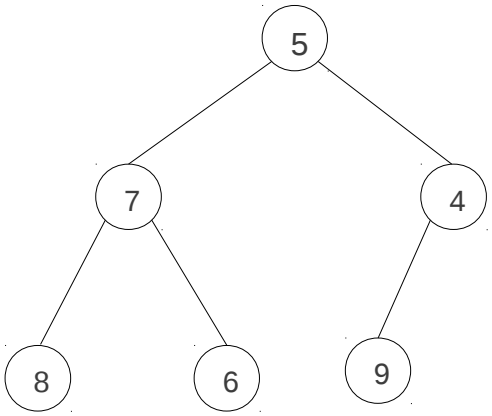
push(4) will first push the element 4 in the heap



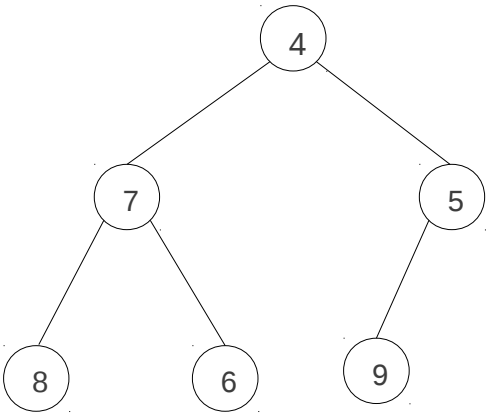
and then swap 4 with its parent 5, because 4 is less than 5. Since 3 is less than 4, this will be the final configuration of the heap.



pop() will return 3. The last element, 5, will first move to the root



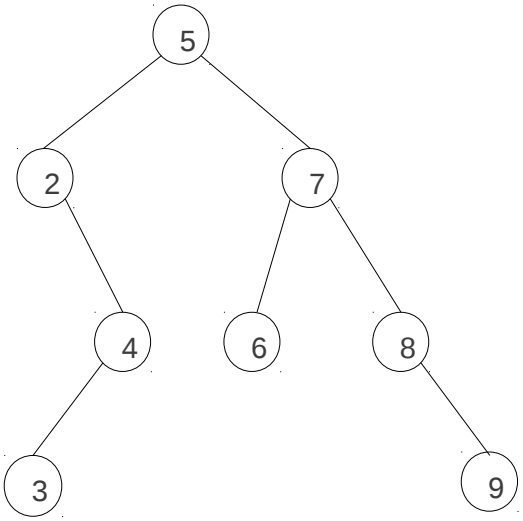
and then it will follow its way downwards by first finding its minimum child, 4, and then swapping it with 4 (since 4 is smaller). It will stop there because 5's new unique child, 9, is smaller.



Heapsort is a sorting method that works in time O(nlogn) and uses the heap structure. For any unsorted sequence, push all the elements one after the other in a heap and then pop

IV. Binary Search Trees

A Binary search tree is a tree where each node value is larger than its left child and smaller than its right. A binary search tree can be shown in the figure below.



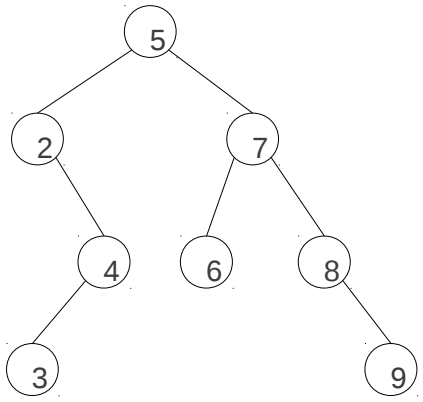
In order to print the elements in a binary search tree we need to perform one of the traversal methods. These include **preorder**, **inorder** and **postorder**. Traversing is done recursively: we visit the node and do recursion in the left and right child.

The proposition (pre-, in-, post-) denotes the place where the node is visited. If it is visited first then it is preorder. If it is visited after the left recursion then it is inorder. If it is visited last it is postorder. Below we see a function to traverse a tree preorder.

```
void preorder(BSTNode* root)
{
        if (root == NULL)
                return;
        cout << root-> data << “, “;
        preorder(root->left);
        preorder(root->right);
}
```

An example of the output of preorder, inorder and postorder traversals can be seen below.

Preorder traveral: 5, 2, 4, 3, 7, 6, 8, 9
Inorder Traversal: 2, 3, 4, 5, 6, 7, 8, 9
Postorder Traversal: 3, 4, 2, 6, 9, 8, 7, 5



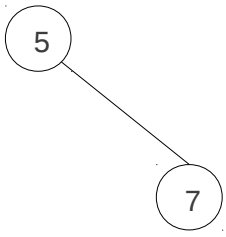
Observe that the inorder traversal prints the elements of the tree **sorted**. That is because of the property of binary search trees that for any node, descendants on the left side are smaller of the node itself and descendants on the right are larger.

The binary search tree is a data structure useful for performing searching efficiently. Methods find, insert and delete can all be implemented in time O(logn) given that the tree is balanced (that means that for every node, the level of the left and right subtrees do not differ by more than one).

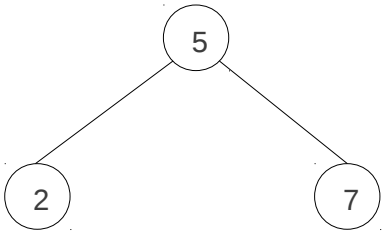
Insert is a recursive method: insert(new_data) checks whether the field data is smaller or larger than new_data and calls insert recursively in the left or the right child respectively. Below we see an illustration of method insert, for inserting a node in a binary tree rooted by the current node.

```
BSTNode* root = new BSTNode;
root->insert(5);
```

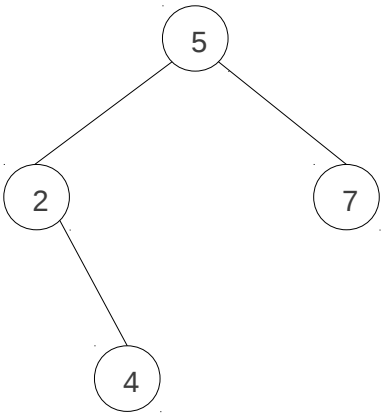

```
root->insert(7);
```



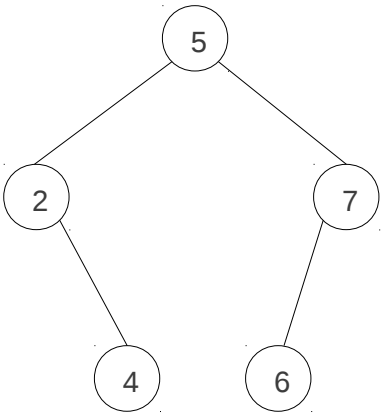
root->insert(2);



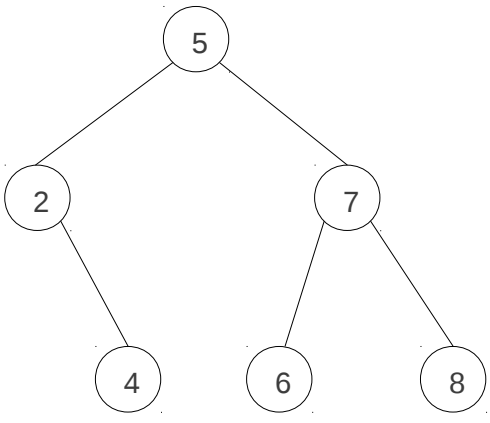
root->insert(4);



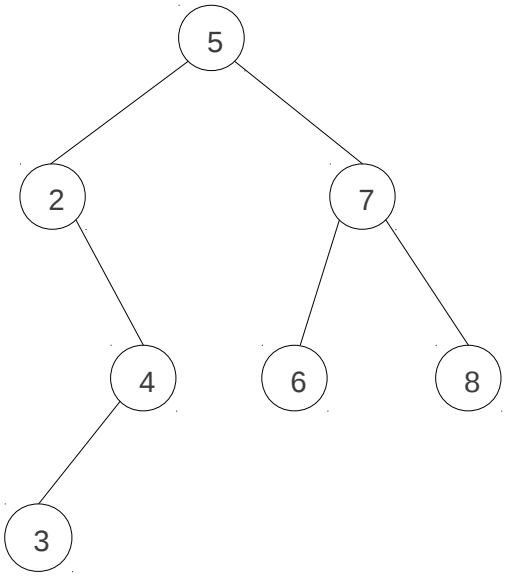
root->insert(6);



root->insert(8);



root->insert(3);



root->insert(9);