CL-218 Data Structures LAB

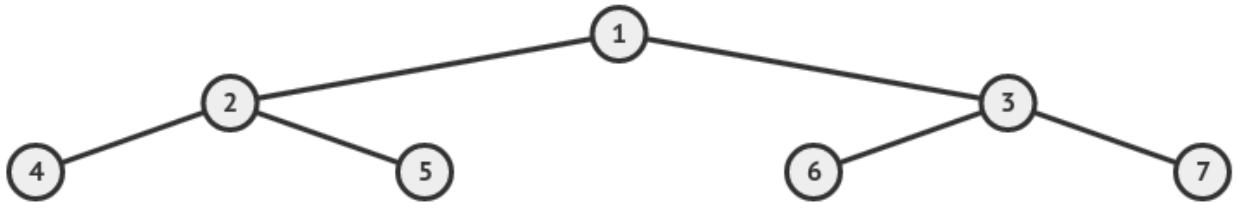
Binary Search Trees

INSTRUCTOR: MUHAMMAD HAMZA

SEMESTER SPRING 2020

A **binary tree** is a hierarchical data structure whose behavior is similar to a tree, as it contains root and leaves (a node that has no child). The *root* of a binary tree is the topmost node. Each node can have at most two children, which are referred to as the *left child* and the *right child*. A node that has at least one child becomes a *parent* of its child. A node that has no child is a *leaf*. In this tutorial, you will be learning about the Binary tree data structures, its principles, and strategies in applying this data structures to various applications.

Take a look at the following binary tree:



From the preceding binary tree diagram, we can conclude the following:

- The root of the tree is the node of element **1** since it's the topmost node
- The children of element **1** are element **2** and element **3**
- The parent of elements **2** and **3** is **1**
- There are four leaves in the tree, and they are element **4**, element **5**, element **6**, and element **7** since they have no child

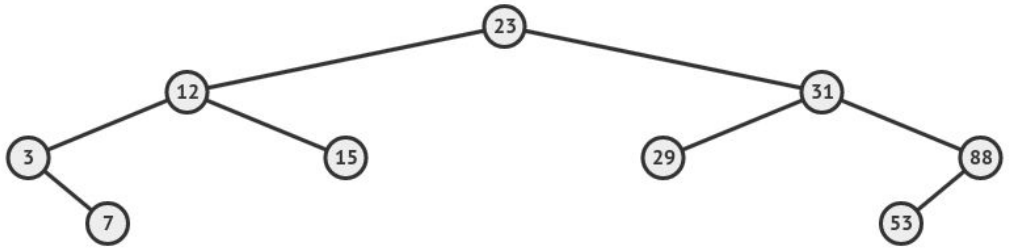
This hierarchical data structure is usually used to store information that forms a hierarchy, such as a file system of a computer.

Building a binary search tree ADT

A **binary search tree** (**BST**) is a sorted binary tree, where we can easily search for any key using the binary search algorithm. To sort the BST, it has to have the following properties:

- The node's left subtree contains only a key that's smaller than the node's key
- The node's right subtree contains only a key that's greater than the node's key
- You cannot duplicate the node's key value

By having the preceding properties, we can easily search for a key value as well as find the maximum or minimum key value. Suppose we have the following BST:



As we can see in the preceding tree diagram, it has been sorted since all of the keys in the root's left subtree are smaller than the root's key, and all of the keys in the root's right subtree are greater than the root's key. The preceding BST is a balanced BST since it has a balanced left and right subtree. We also can define the preceding BST as a balanced BST since both the left and right subtrees have an equal *height* (we are going to discuss this further in the upcoming section).

There are several basic operations which BST usually has, and they are as follows:

- Insert() is used to add a new node to the current BST. If it's the first time we have added a node, the node we inserted will be a root node.

- Tree Traversal() is used to print all of the keys in the BST.
- Search() is used to find a given key in the BST. If the key exists it returns TRUE, otherwise it returns FALSE.
- FindMin() and FindMax() are used to find the minimum key and the maximum key that exist in the BST.
- Successor() and Predecessor() are used to find the successor and predecessor of a given key. We are going to discuss these later in the upcoming section.
- Remove() is used to remove a given key from BST.

Inserting a new key into a BST

Inserting a key into the BST is actually adding a new node based on the behavior of the BST. Each time we want to insert a key, we have to compare it with the root node (if there's no root beforehand, the inserted key becomes a root) and check whether it's smaller or greater than the root's key. If the given key is greater than the currently selected node's key, then go to the right subtree. Otherwise, go to the left subtree if the given key is smaller than the currently selected node's key. Keep checking this until there's a node with no child so that we can add a new node there.

Finding out whether a key exists in a BST

Suppose we have a BST and need to find out if a key exists in the BST. It's quite easy to check whether a given key exists in a BST, since we just need to compare the given key with the current node. If the key is smaller than the current node's key, we go to the left subtree, otherwise we go to the right subtree. We will do this until we find the key or when there are no more nodes to find

Retrieving the minimum and maximum key values

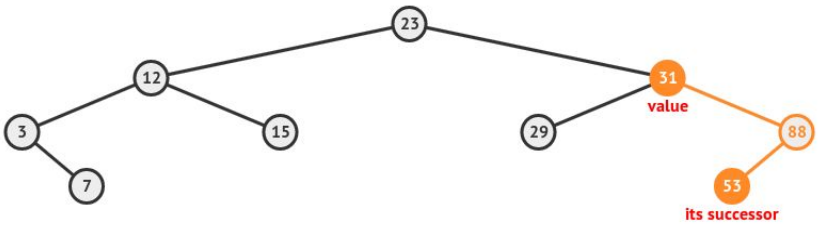
Finding out the minimum and maximum key values in a BST is also quite simple. To get a minimum key value, we just need to go to the leftmost node and get the key value. On the contrary, we just need to go to the rightmost node and we will find the maximum key value.

Finding out the successor of a key in a BST

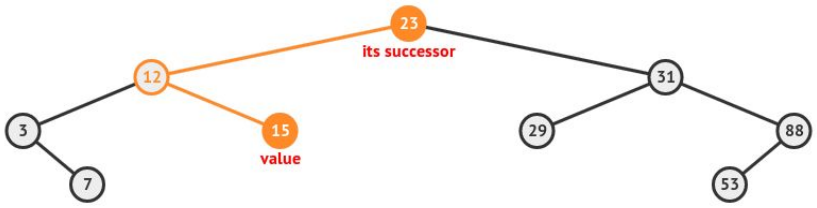
Other properties that we can find from a BST are the **successor** and the **predecessor**. We are going to create two functions named Successor() and Predecessor() in C++. But before we create the code, let's discuss how to find out the successor and the predecessor of a key of a BST. In this section, we are going to learn about the successor first, and then we will discuss the predecessor in the upcoming section.

There are three rules to find out the successor of a key of a BST. Suppose we have a key, k, that we have searched for using the previous Search() function. We will also use our preceding BST to find out the successor of a specific key. The successor of k can be found as follows:

1. If k has a right subtree, the successor of k will be the minimum integer in the right subtree of k. From our preceding BST, if k = 31, Successor(31) will give us 53 since it's the minimum integer in the right subtree of 31. Please take a look at the following diagram:



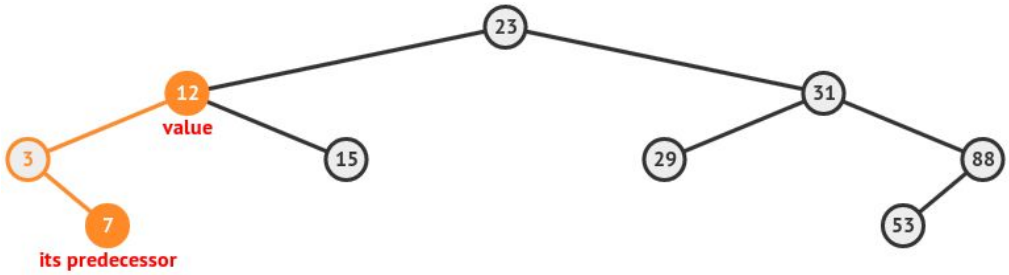
2. If k does not have a right subtree, we have to traverse the ancestors of k until we find the first node, n, which is greater than node k. After we find node n, we will see that node k is the maximum element in the left subtree of n. From our preceding BST, if k = 15, Successor(15) will give us 23 since it's the first greater ancestor compared with 15, which is 23. Please take a look at the following diagram:



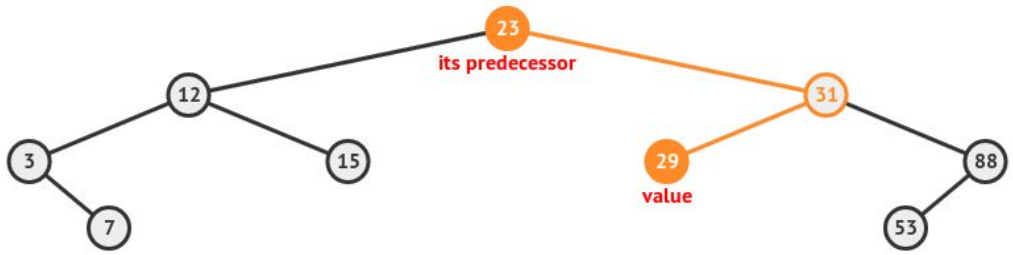
3. If k is the maximum integer in the BST, there's no successor of k. From the preceding BST, if we run Successor(88), we will get -1, which means no successor has been found, since 88 is the maximum key of the BST.

Finding out the predecessor of a key in a BST

1. If k has a left subtree, the predecessor of k will be the maximum integer in the left subtree of k. From our preceding BST, if k = 12, Predecessor(12) will be 7 since it's the maximum integer in the left subtree of 12. Please take a look at the following diagram:



2. If k does not have a left subtree, we have to traverse the ancestors of k until we find the first node, n, which is lower than node k. After we find node n, we will see that node n is the minimum element of the traversed elements. From our preceding BST, if k = 29, Predecessor(29) will give us 23 since it's the first lower ancestor compared with 29, which is 23. Please take a look at the following diagram:



3. If k is the minimum integer in the BST, there's no predecessor of k. From the preceding BST, if we run Predecessor(3), we will get -1, which means no predecessor is found since 3 is the minimum key of the BST.

Removing a node based on a given key

The last operation in the BST that we are going to discuss is removing a node based on a given key. We will create a Remove() operation in C++. There are three possible cases for removing a node from a BST, and they are as follows:

1. Removing a leaf (a node that doesn't have any child). In this case, we just need to remove the node. From our preceding BST, we can remove keys 7, 15, 29, and 53 since they are leaves with no nodes.
2. Removing a node that has only one child (either a left or right child). In this case, we have to connect the child to the parent of the node. After that, we can remove the target node safely. As an example, if we want to remove node 3, we have to point the Parent pointer of node 7 to node 12 and make the left node of 12 points to 7. Then, we can safely remove node 3.
3. Removing a node that has two children (left and right children). In this case, we have to find out the successor (or predecessor) of the node's key. After that, we can replace the target node with the successor (or predecessor) node. Suppose we want to remove node 31, and that we want 53 as its successor. Then, we can remove node 31 and replace it with node 53. Now, node 53 will have two children, node 29 in the left and node 88 in the right.

Traversing a BST:

Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree −

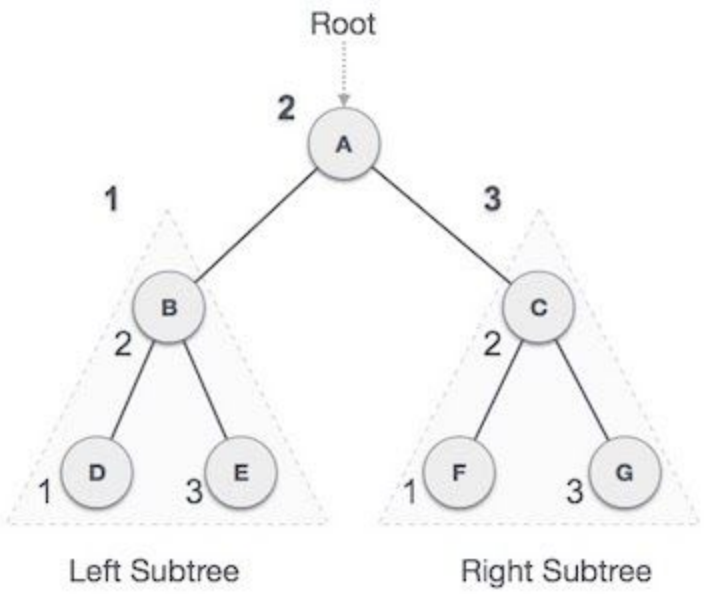
- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

Generally, we traverse a tree to search or locate a given item or key in the tree or to print all the values it contains.

In-order Traversal

In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.

If a binary tree is traversed in-order, the output will produce sorted key values in an ascending order.



We start from A, and following in-order traversal, we move to its left subtree B. B is also traversed in-order. The process goes on until all the nodes are visited. The output of inorder traversal of this tree will be −

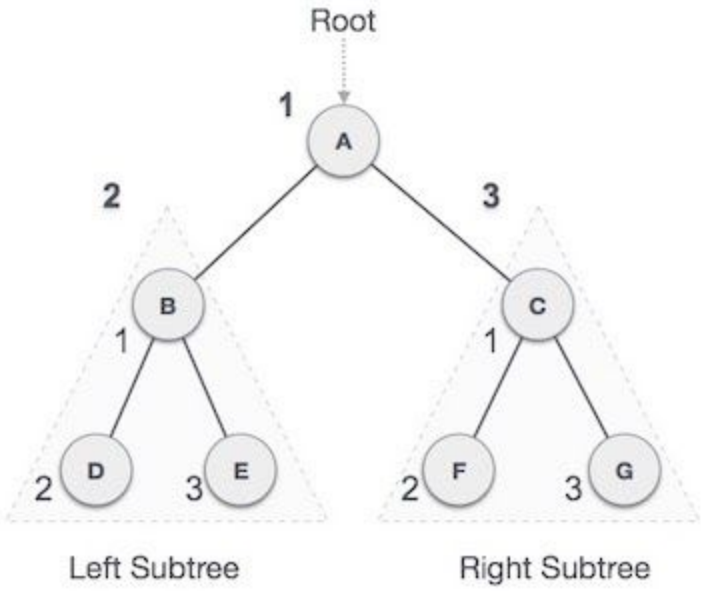
$$D \rightarrow B \rightarrow E \rightarrow A \rightarrow F \rightarrow C \rightarrow G$$

Algorithm

Until all nodes are traversed −
Step 1 − Recursively traverse left subtree.
Step 2 − Visit root node.
Step 3 − Recursively traverse right subtree.

Pre-order Traversal

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.



We start from A, and following pre-order traversal, we first visit A itself and then move to its left subtree B. B is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be −

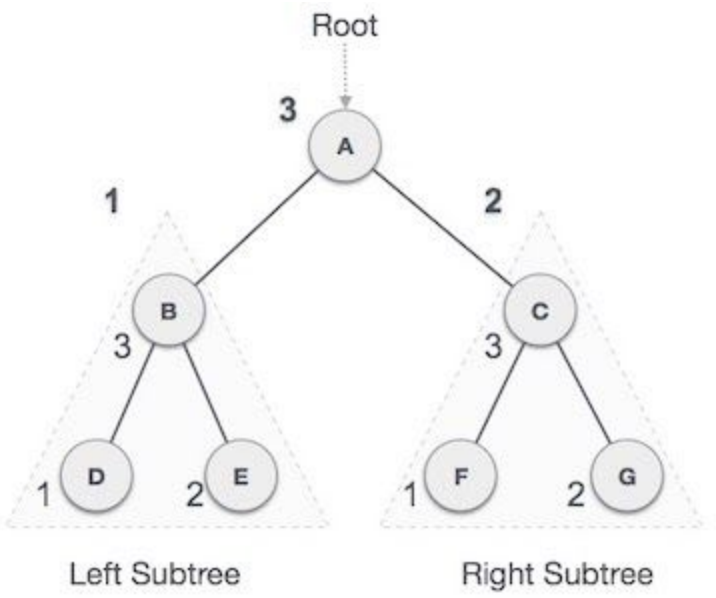
$$A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow F \rightarrow G$$

Algorithm

Until all nodes are traversed −
Step 1 − Visit root node.
Step 2 − Recursively traverse left subtree.
Step 3 − Recursively traverse right subtree.

Post-order Traversal

In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.



We start from A, and following Post-order traversal, we first visit the left subtree B. B is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be −

$$D \rightarrow E \rightarrow B \rightarrow F \rightarrow G \rightarrow C \rightarrow A$$

Algorithm

Until all nodes are traversed −
Step 1 − Recursively traverse left subtree.
Step 2 − Recursively traverse right subtree.
Step 3 − Visit root node.