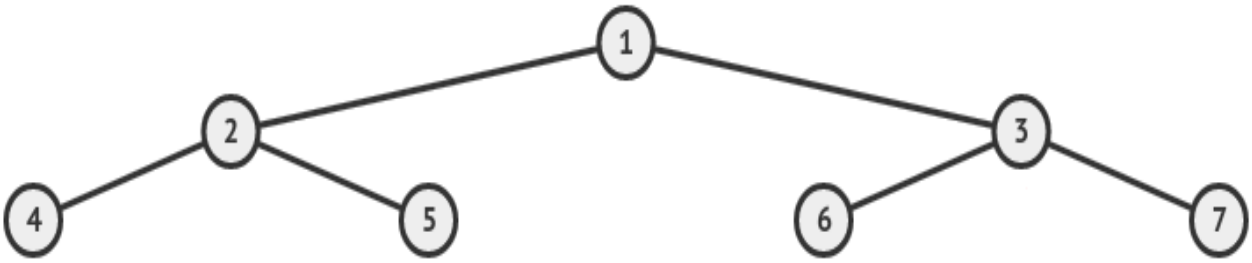
Lab: Build a binary tree

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.

This C++ tutorial has been taken from *C++ Data Structures and Algorithms*. Read more here.

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

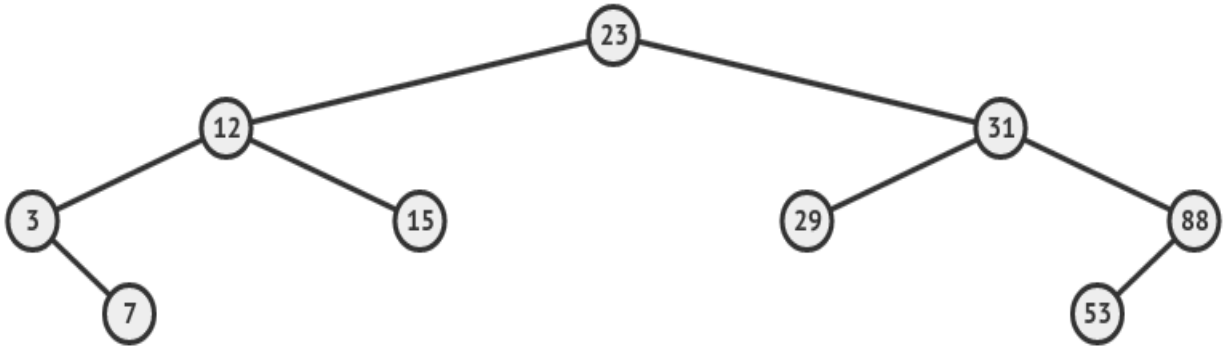
This hierarchical data structure is usually used to store information that forms a hierarchy, such as a file system of a computer.

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:

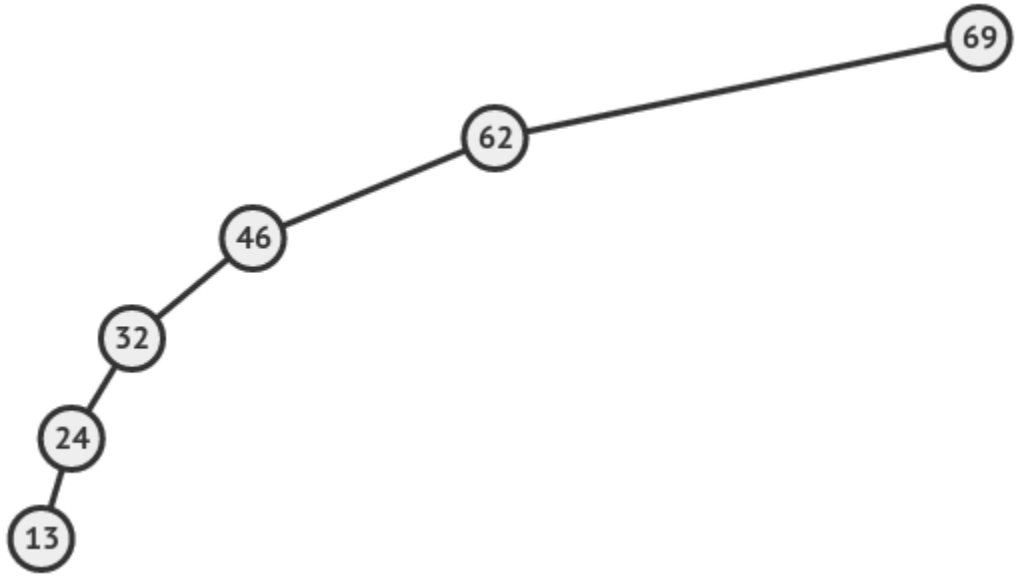
- The node's left subtree contains only a key that's smaller than the node's key
- The node's right subtree contains only a key that's greater than the node's key
- You cannot duplicate the node's key value

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:

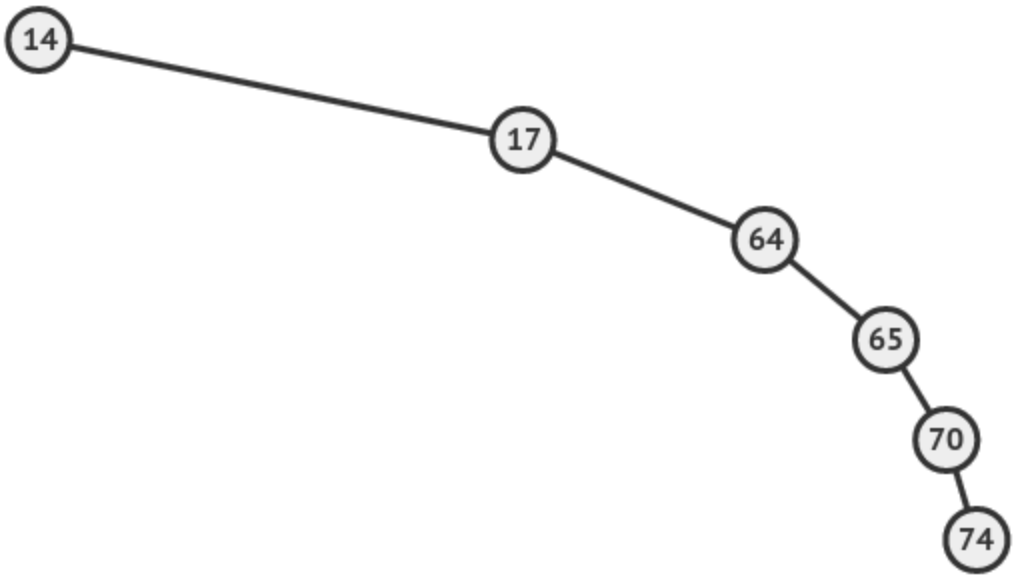


As we can see in the preceding tree diagram, it has been sorted since all of the keys in the root's left subtree are smaller than the root's key, and all of the keys in the root's right subtree are greater than the root's key. The preceding BST is a balanced BST since it has a balanced left and right 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).

However, since we have to put the greater new key in the right subtree and the smaller new key in the left subtree, we might find an unbalanced BST, called a **skewed left** or a **skewed right BST**. Please see the following diagram:



The preceding image is a sample of a *skewed left BST*, since there's no right subtree. Also, we can find a BST that has no left subtree, which is called a *skewed right BST*, as shown in the following diagram:



As we can see in the two skewed BST diagrams, the height of the BST becomes taller since the height equals to *N – 1* (where *N* is the total keys in the BST), which is five. Comparing this with the balanced BST, the root's height is only three.

To create a BST in C++, we need to modify our TreeNode class in the preceding binary tree discussion, *Building a binary tree ADT*. We need to add the Parent properties so that we can track the parent of each node. It will make things easier for us when we traverse the tree. The class should be as follows:

```
class BSTNode

{

public:

    int Key;

    BSTNode * Left;

    BSTNode * Right;

    BSTNode * Parent;

};
```

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.
- PrintTreeInOrder() is used to print all of the keys in the BST, sorted from the smallest key to the greatest key.
- 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.

Now, let's discuss these BST operations further.

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. The following is the implementation of the Insert() operation in C++:

```
BSTNode * BST::Insert(BSTNode * node, int key)

{

    // If BST doesn't exist

    // create a new node as root

    // or it's reached when

    // there's no any child node

    // so we can insert a new node here

    if(node == NULL)
```

```
{
    node = new BSTNode;
    node->Key = key;
    node->Left = NULL;
    node->Right = NULL;
    node->Parent = NULL;
}
// If the given key is greater than
// node's key then go to right subtree
else if(node->Key < key)
{
    node->Right = Insert(node->Right, key);
    node->Right->Parent = node;
}
// If the given key is smaller than
// node's key then go to left subtree
else
{
    node->Left = Insert(node->Left, key);
    node->Left->Parent = node;
}
```

```
return node;

}
```

As we can see in the preceding code, we need to pass the selected node and a new key to the function. However, we will always pass the root node as the selected node when performing the Insert() operation, so we can invoke the preceding code with the following Insert() function:

```
void BST::Insert(int key)

{

    // Invoking Insert() function

    // and passing root node and given key

    root = Insert(root, key);

}
```

Based on the implementation of the Insert() operation, we can see that the time complexity to insert a new key into the BST is *O(h)*, where *h* is the height of the BST. However, if we insert a new key into a non-existing BST, the time complexity will be *O(1)*, which is the best case scenario. And, if we insert a new key into a skewed tree, the time complexity will be *O(N)*, where *N* is the total number of keys in the BST, which is the worst case scenario.

Traversing a BST in order

We have successfully created a new BST and can insert a new key into it. Now, we need to implement the PrintTreeInOrder() operation, which will traverse the BST in order from the smallest key to the greatest key. To achieve this, we will go to the leftmost node and then to the rightmost node. The code should be as follows:

```
void BST::PrintTreeInOrder(BSTNode * node)

{

    // Stop printing if no node found

    if(node == NULL)
```

```
        return;

// Get the smallest key first

// which is in the left subtree

PrintTreeInOrder(node->Left);



// Print the key

std::cout << node->Key << " ";



// Continue to the greatest key

// which is in the right subtree

PrintTreeInOrder(node->Right);

}
```

Since we will always traverse from the `root` node, we can invoke the preceding code as follows:

```
void BST::PrintTreeInOrder()

{

    // Traverse the BST

    // from root node

    // then print all keys

    PrintTreeInOrder(root);

    std::cout << std::endl;

}
```

The time complexity of the PrintTreeInOrder() function will be *O(N)*, where *N* is the total number of keys for both the best and the worst cases since it will always traverse to all keys.

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. The implementation of the Search() operation should be as follows:

```
BSTNode * BST::Search(BSTNode * node, int key)

{

    // The given key is

    // not found in BST

    if (node == NULL)

        return NULL;

    // The given key is found

    else if(node->Key == key)

        return node;

    // The given is greater than

    // current node's key

    else if(node->Key < key)

        return Search(node->Right, key);

    // The given is smaller than

    // current node's key

    else
```

```
        return Search(node->Left, key);

}
```

Since we will always search for a key from the root node, we can create another Search() function as follows:

```
bool BST::Search(int key)

{

    // Invoking Search() operation

    // and passing root node

    BSTNode * result = Search(root, key);

// If key is found, returns TRUE

// otherwise returns FALSE

return result == NULL ?

false :

true;

}
```

The time complexity to find out a key in the BST is *O(h)*, where *h* is the height of the BST. If we find a key which lies in the root node, the time complexity will be *O(1)*, which is the best case. If we search for a key in a skewed tree, the time complexity will be *O(N)*, where *N* is the total number of keys in the BST, which is the worst case.

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. The following is the

implementation of the FindMin() operation to retrieve the minimum key value, and the FindMax() operation to retrieve the maximum key value:

```
int BST::FindMin(BSTNode * node)

{

    if(node == NULL)

        return -1;

    else if(node->Left == NULL)

        return node->Key;

    else

        return FindMin(node->Left);

}

int BST::FindMax(BSTNode * node)

{

if(node == NULL)

return -1;

else if(node->Right == NULL)

return node->Key;

else

return FindMax(node->Right);

}
```

We return -1 if we cannot find the minimum or maximum value in the tree, since we assume that the tree can only have a positive integer. If we intend to store the negative integer as well, we need to

modify the function's implementation, for instance, by returning NULL if no minimum or maximum values are found.

As usual, we will always find the minimum and maximum key values from the root node, so we can invoke the preceding operations as follows:

```
int BST::FindMin()

{

    return FindMin(root);

}

int BST::FindMax()

{

return FindMax(root);

}
```

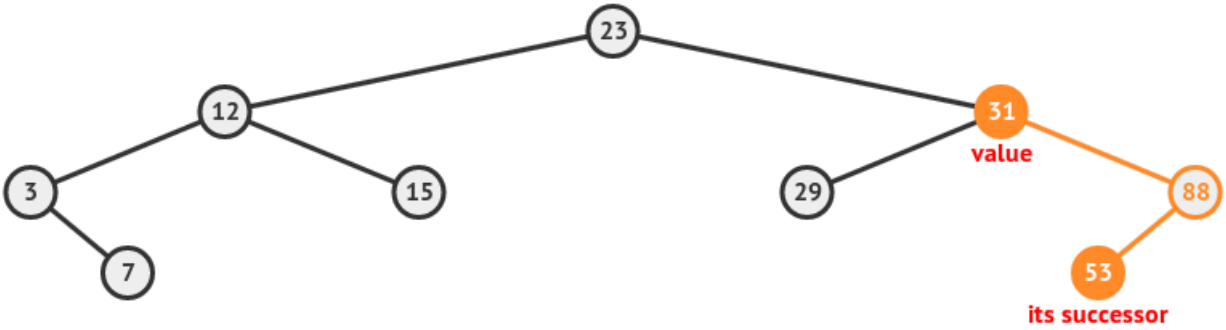
Similar to the Search() operation, the time complexity of the FindMin() and FindMax() operations is *O(h)*, where *h* is the height of the BST. However, if we find the maximum key value in a skewed left BST, the time complexity will be *O(1)*, which is the best case, since it doesn't have any right subtree. This also happens if we find the minimum key value in a skewed right BST. The worst case will appear if we try to find the minimum key value in a skewed left BST or try to find the maximum key value in a skewed right BST, since the time complexity will be *O(N)*.

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.



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.

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.

Based on our preceding discussion about how to find out the successor of a given key in a BST, we can create a Successor() function in C++ with the following implementation:

```
int BST::Successor(BSTNode * node)

{
```

```
    // The successor is the minimum key value
    // of right subtree
    if (node->Right != NULL)
    {
        return FindMin(node->Right);
    }
    // If no any right subtree
    else
    {
        BSTNode * parentNode = node->Parent;
        BSTNode * currentNode = node;
// If currentNode is not root and
// currentNode is its right children
// continue moving up
while ((parentNode != NULL) &&
(currentNode == parentNode->Right))
{
currentNode = parentNode;
parentNode = currentNode->Parent;
}
```

```
// If parentNode is not NULL

// then the key of parentNode is

// the successor of node

return parentNode == NULL ?

-1 :

parentNode->Key;

}

}
```

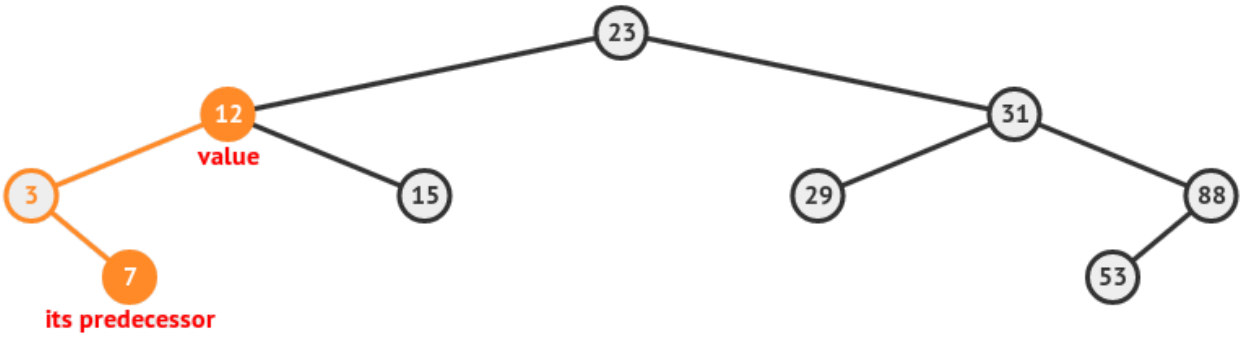
However, since we have to find a given key's node first, we have to run Search() prior to invoking the preceding Successor() function. The complete code for searching for the successor of a given key in a BST is as follows:

```
int BST::Successor(int key)

{

    // Search the key's node first

    BSTNode * keyNode = Search(root, key);

// Return the key.

// If the key is not found or

// successor is not found,

// return -1

return keyNode == NULL ?

-1 :

Successor(keyNode);
```

```
}
```

From our preceding Successor() operation, we can say that the average time complexity of running the operation is *O(h)*, where *h* is the height of the BST. However, if we try to find out the successor of a maximum key in a skewed right BST, the time complexity of the operation is *O(N)*, which is the worst case scenario.

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. 

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.

Now, we can implement the Predecessor() operation in C++ as follows:

```
int BST::Predecessor(BSTNode * node)

{

    // The predecessor is the maximum key value

    // of left subtree

    if (node->Left != NULL)

    {

        return FindMax(node->Left);

    }

    // If no any left subtree

    else

    {

        BSTNode * parentNode = node->Parent;

        BSTNode * currentNode = node;

// If currentNode is not root and

// currentNode is its left children

// continue moving up

while ((parentNode != NULL) &&

(currentNode == parentNode->Left))
```

```
{

currentNode = parentNode;

parentNode = currentNode->Parent;

}



// If parentNode is not NULL

// then the key of parentNode is

// the predecessor of node

return parentNode == NULL ?

-1 :

parentNode->Key;

}

}
```

And, similar to the Successor() operation, we have to search for the node of a given key prior to invoking the preceding Predecessor() function. The complete code for searching for the predecessor of a given key in a BST is as follows:

```
int BST::Predecessor(int key)

{

    // Search the key's node first

    BSTNode * keyNode = Search(root, key);

// Return the key.

// If the key is not found or
```

```
// predecessor is not found,

// return -1

return keyNode == NULL ?

-1 :

Predecessor(keyNode);

}
```

Similar to our preceding Successor() operation, the time complexity of running the Predecessor() operation is *O(h)*, where *h* is the height of the BST. However, if we try to find out the predecessor of a minimum key in a skewed left BST, the time complexity of the operation is *O(N)*, which is the worst case scenario.

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.

Also, similar to the Search() operation, if the target node doesn't exist, we just need to return NULL. The implementation of the Remove() operation in C++ is as follows:

```
BSTNode * BST::Remove(

    BSTNode * node,

    int key)

{

    // The given node is

    // not found in BST

    if (node == NULL)

        return NULL;

// Target node is found

if (node->Key == key)

{

// If the node is a leaf node

// The node can be safely removed

if (node->Left == NULL && node->Right == NULL)

node = NULL;

// The node have only one child at right

else if (node->Left == NULL && node->Right != NULL)

{

// The only child will be connected to

// the parent's of node directly

node->Right->Parent = node->Parent;
```

```
// Bypass node

node = node->Right;

}

// The node have only one child at left

else if (node->Left != NULL && node->Right == NULL)

{

// The only child will be connected to

// the parent's of node directly

node->Left->Parent = node->Parent;

// Bypass node

node = node->Left;

}

// The node have two children (left and right)

else

{

// Find successor or predecessor to avoid quarrel

int successorKey = Successor(key);



// Replace node's key with successor's key

node->Key = successorKey;

// Delete the old successor's key
```

```
node->Right = Remove(node->Right, successorKey);

}

}

// Target node's key is smaller than

// the given key then search to right

else if (node->Key < key)

node->Right = Remove(node->Right, key);

// Target node's key is greater than

// the given key then search to left

else

node->Left = Remove(node->Left, key);



// Return the updated BST

return node;

}
```

Since we will always remove a node starting from the root node, we can simplify the preceding Remove() operation by creating the following one:

```
void BST::Remove(int key)

{

    root = Remove(root, key);

}
```

As shown in the preceding Remove() code, the time complexity of the operation is *O(1)* for both case **1** (the node that has no child) and case **2** (the node that has only one child). For case **3** (the node that has two children), the time complexity will be *O(h)*, where *h* is the height of the BST, since we have to find the successor or predecessor of the node's key.