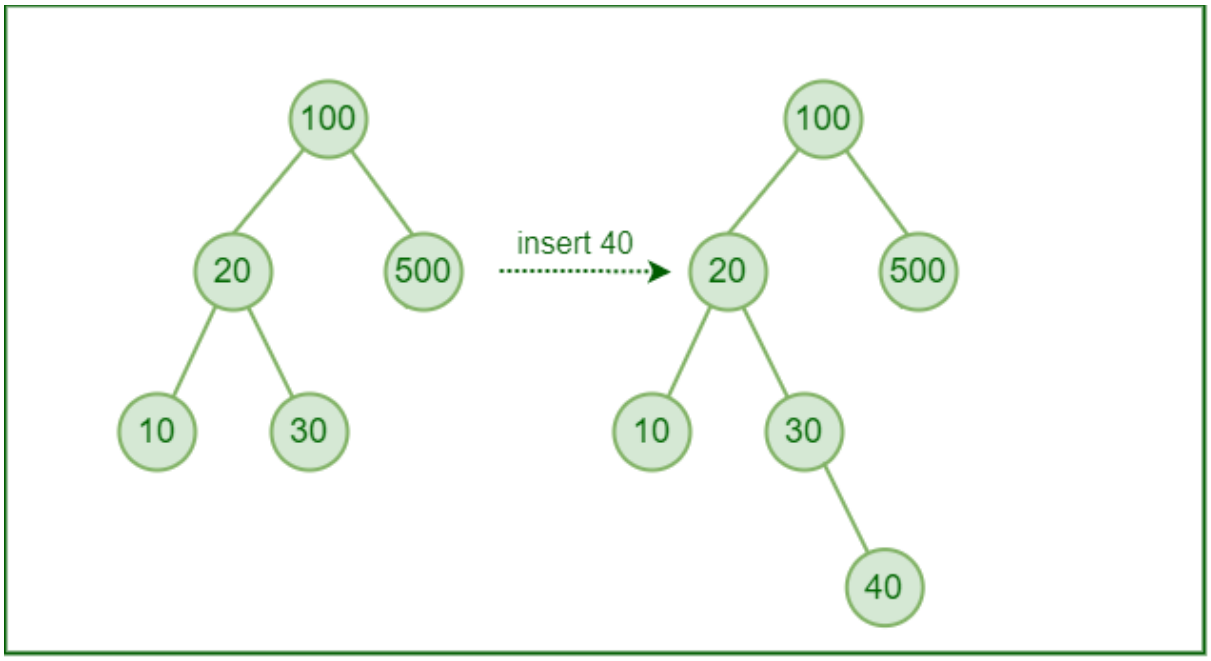
Insertion in Binary Search Tree (BST)

Given a **BST**, the task is to insert a new node in this **BST**.
Example:



Insertion in Binary Search Tree

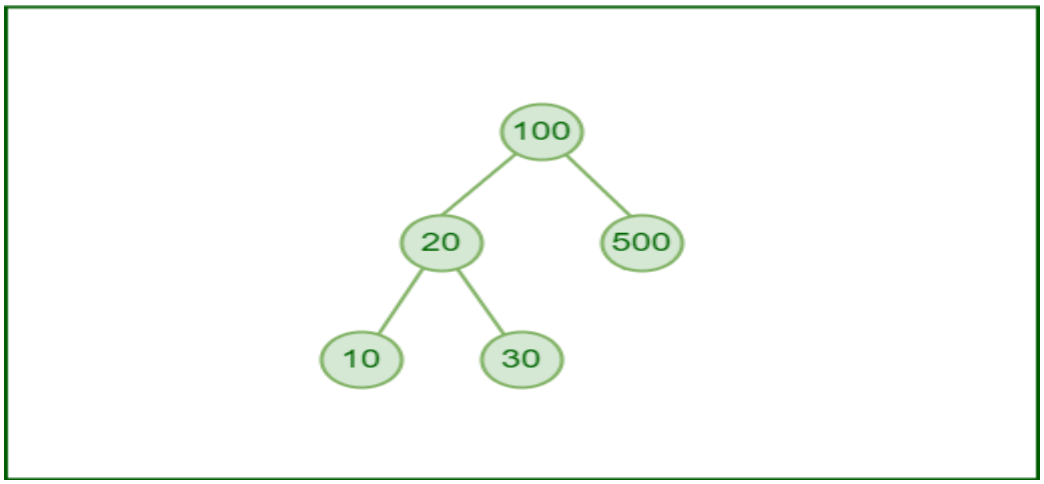
How to Insert a value in a Binary Search Tree:

A new key is always inserted at the leaf by maintaining the property of the binary search tree. We start searching for a key from the root until we hit a leaf node. Once a leaf node is found, the new node is added as a child of the leaf node. The below steps are followed while we try to insert a node into a binary search tree:

- Check the value to be inserted (say **X**) with the value of the current node (say **val**) we are in:
    - If **X** is less than **val** move to the left subtree.
    - Otherwise, move to the right subtree.
- Once the leaf node is reached, insert **X** to its right or left based on the relation between **X** and the leaf node's value.

Follow the below illustration for a better understanding:

Illustration:
Consider the below tree:



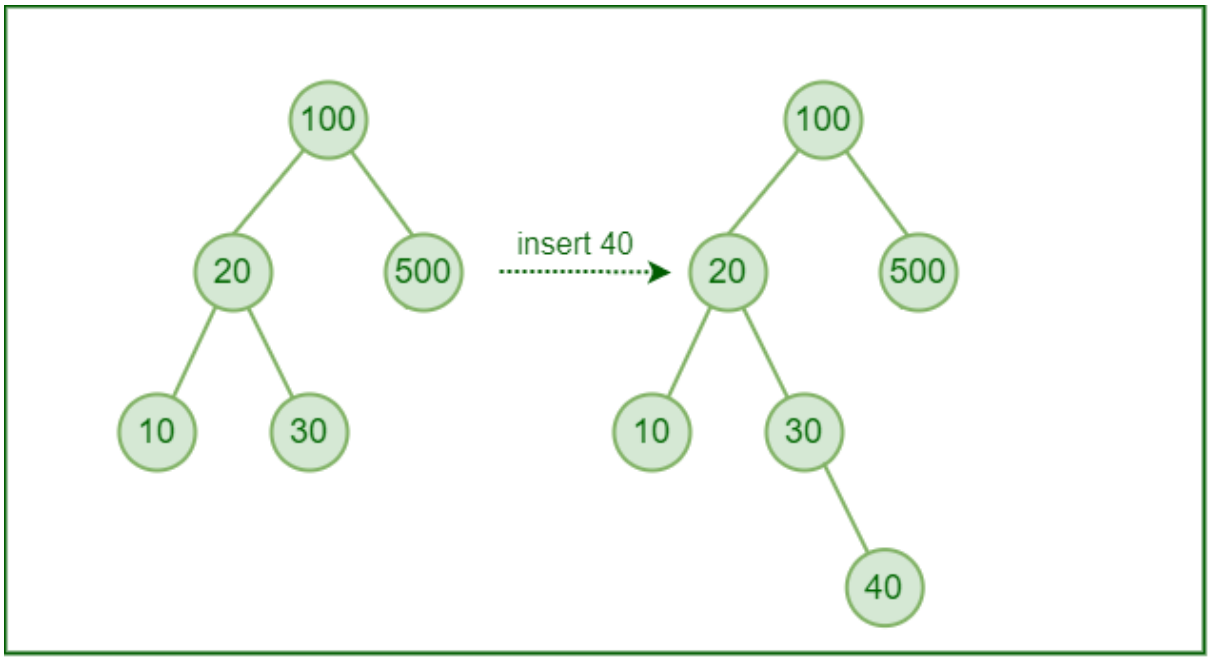
Binary Search Tree

*Let us try to insert a node with value **40** in this tree:*
***1st step:** 40 will be compared with root, i.e., 100.*
- *40 is less than 100.*
- *So move to the left subtree of 100. The root of the left subtree is 20.*

***2nd step:** 40 is now compared with 20.*
- *It is greater than 20.*
- *So move to the right subtree of 20 whose root is 30.*

***3rd step:** 30 is a leaf node.*
- *So we have to insert 40 to the left or right of 30.*
- *As 40 is greater than 30, insert 40 to the right of 30.*

The new tree will look like the following:

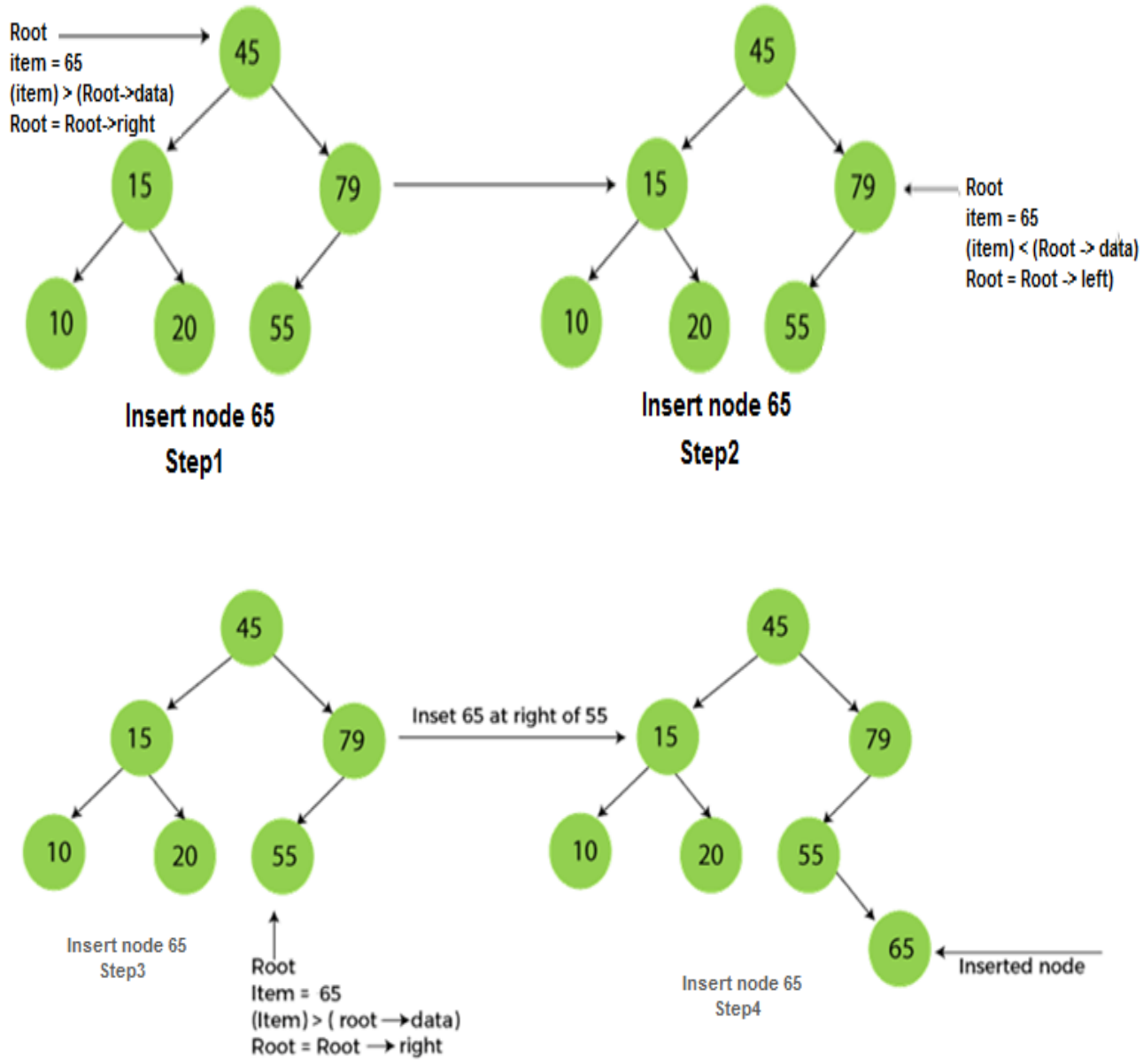


Insertion in Binary Search Tree

Insertion in Binary Search tree

A new key in BST is always inserted at the leaf. To insert an element in BST, we have to start searching from the root node; if the node to be inserted is less than the root node, then search for an empty location in the left subtree. Else, search for the empty location in the right subtree and insert the data. Insert in BST is similar to searching, as we always have to maintain the rule that the left subtree is smaller than the root, and right subtree is larger than the root.

Now, let's see the process of inserting a node into BST using an example.



Insertion in Binary Search Tree using Recursion:

Below is the implementation of the insertion operation using recursion.

```
// C program to demonstrate insert
// operation in binary
// search tree.

#include <stdio.h>
#include <stdlib.h>

struct node {
    int key;
    struct node *left, *right;
};

// A utility function to create a new BST node
struct node* newNode(int item)
{
    struct node* temp
        = (struct node*)malloc(sizeof(struct node));
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}

// A utility function to do inorder traversal of BST
void inorder(struct node* root)
{
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->key);
        inorder(root->right);
    }
}

// A utility function to insert
// a new node with given key in BST
struct node* insert(struct node* node, int key)
{
    // If the tree is empty, return a new node
    if (node == NULL)
        return newNode(key);

    // Otherwise, recur down the tree
    if (key < node->key)
```

```
        node->left = insert(node->left, key);
    elseif(key > node->key)
        node->right = insert(node->right, key);

    // Return the (unchanged) node pointer
    returnnode;
}

// Driver Code
intmain()
{
    /* Let us create following BST
              50
           /     \
          30      70
         / \    / \
       20   40  60   80 */
    structnode* root = NULL;
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);

    // Print inorder traversal of the BST
    inorder(root);

    return0;
}
```

Output
20 30 40 50 60 70 80

Time Complexity:

- The worst-case time complexity of insert operations is **O(h)** where **h** is the height of the Binary Search Tree.
- In the worst case, we may have to travel from the root to the deepest leaf node. The height of a skewed tree may become **n** and the time complexity of insertion operation may become **O(n).**

Auxiliary Space: The auxiliary space complexity of insertion into a binary search tree is **O(1)**

Insertion in Binary Search Tree using Iterative approach:

Instead of using recursion, we can also implement the insertion operation iteratively using a **while loop**. Below is the implementation using a while loop.

```
// C++ Code to insert node and to print inorder traversal
// using iteration

#include <bits/stdc++.h>
usingnamespacestd;

// BST Node
classNode {
public:
    intval;
    Node* left;
    Node* right;
    Node(intval)
        : val(val)
        , left(NULL)
        , right(NULL)
    {
    }
};

// Utility function to insert node in BST
voidinsert(Node*& root, intkey)
{
    Node* node = newNode(key);
    if(!root) {
        root = node;
        return;
    }
    Node* prev = NULL;
    Node* temp = root;
    while(temp) {
        if(temp->val> key) {
            prev = temp;
            temp = temp->left;
        }
        elseif(temp->val< key) {
            prev = temp;
            temp = temp->right;
        }
    }
    if(prev->val> key)
```

```
        prev->left = node;
    else
        prev->right = node;
}

// Utility function to print inorder traversal
voidinorder(Node* root)
{
    Node* temp = root;
    stack<Node*>st;
    while(temp != NULL || !st.empty()) {
        if(temp != NULL) {
            st.push(temp);
            temp = temp->left;
        }
        else{
            temp = st.top();
            st.pop();
            cout<< temp->val<<" ";
            temp = temp->right;
        }
    }
}

// Driver code
intmain()
{
    Node* root = NULL;
    insert(root, 30);
    insert(root, 50);
    insert(root, 15);
    insert(root, 20);
    insert(root, 10);
    insert(root, 40);
    insert(root, 60);

    // Function call to print the inorder traversal
    inorder(root);

    return0;
}
```

Output

10 15 20 30 40 50 60

The **time complexity** of **inorder traversal** is **O(n)**, as each node is visited once. The **Auxiliary space** is **O(n)**, as we use a stack to store nodes for recursion.

Deletion in Binary Search tree

In a binary search tree, we must delete a node from the tree by keeping in mind that the property of BST is not violated. To delete a node from BST, there are three possible situations occur -

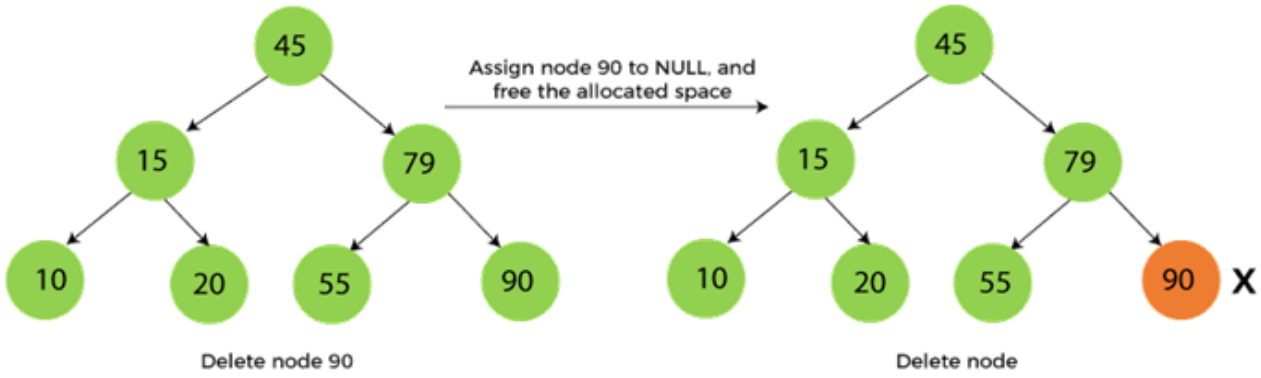
- The node to be deleted is the leaf node, or,
- The node to be deleted has only one child, and,
- The node to be deleted has two children

We will understand the situations listed above in detail.

When the node to be deleted is the leaf node

It is the simplest case to delete a node in BST. Here, we have to replace the leaf node with NULL and simply free the allocated space.

We can see the process to delete a leaf node from BST in the below image. In below image, suppose we have to delete node 90, as the node to be deleted is a leaf node, so it will be replaced with NULL, and the allocated space will free.

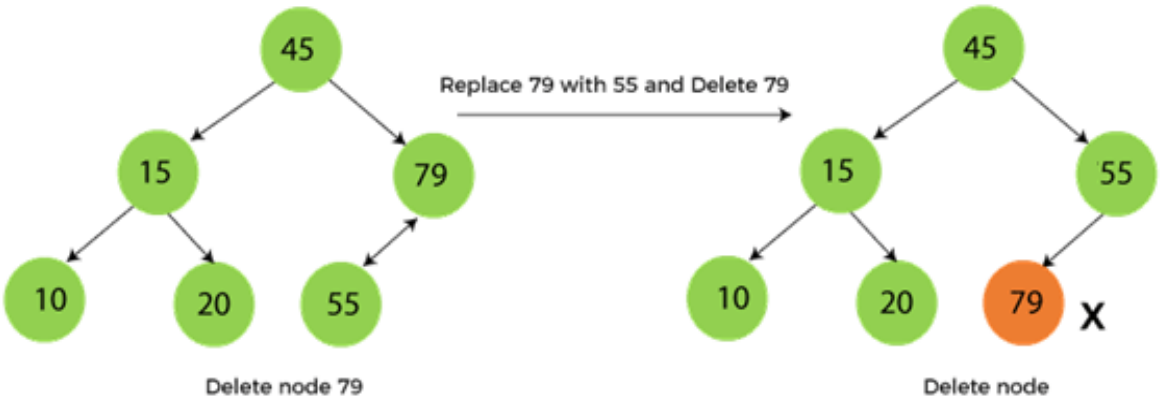


When the node to be deleted has only one child

In this case, we have to replace the target node with its child, and then delete the child node. It means that after replacing the target node with its child node, the child node will now contain the value to be deleted. So, we simply have to replace the child node with NULL and free up the allocated space.

We can see the process of deleting a node with one child from BST in the below image. In the below image, suppose we have to delete the node 79, as the node to be deleted has only one child, so it will be replaced with its child 55.

So, the replaced node 79 will now be a leaf node that can be easily deleted.



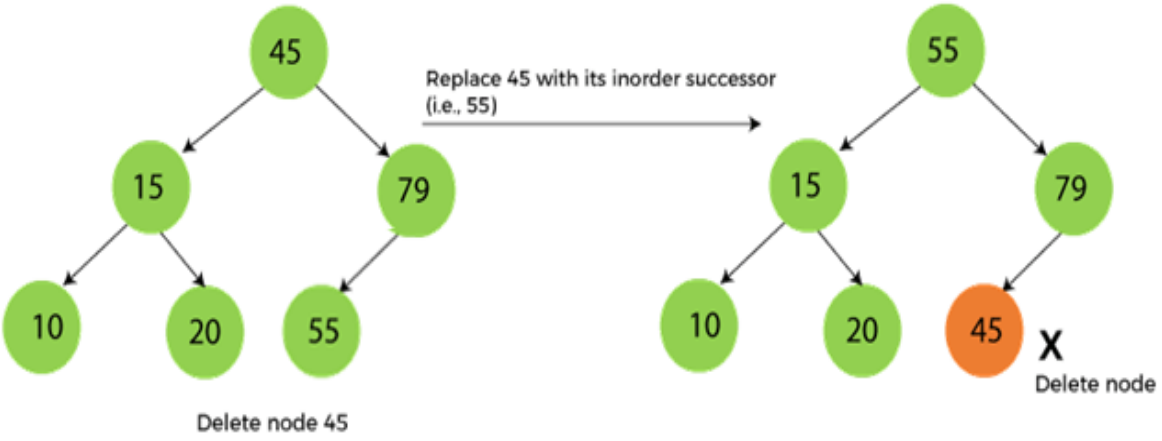
When the node to be deleted has two children

This case of deleting a node in BST is a bit complex among other two cases. In such a case, the steps to be followed are listed as follows -

- First, find the inorder successor of the node to be deleted.
- After that, replace that node with the inorder successor until the target node is placed at the leaf of tree.
- And at last, replace the node with NULL and free up the allocated space.

The inorder successor is required when the right child of the node is not empty. We can obtain the inorder successor by finding the minimum element in the right child of the node.

We can see the process of deleting a node with two children from BST in the below image. In the below image, suppose we have to delete node 45 that is the root node, as the node to be deleted has two children, so it will be replaced with its inorder successor. Now, node 45 will be at the leaf of the tree so that it can be deleted easily.



Implementation of Deletion operation in a BST:

```
// C program to implement optimized delete in BST.
#include <stdio.h>
#include <stdlib.h>

structNode {
    intkey;
    structNode *left, *right;
};

// A utility function to create a new BST node
structNode* newNode(intitem)
{
    structNode* temp = (structNode*)malloc(sizeof(structNode));
    temp->key = item;
    temp->left = temp->right = NULL;
    returntemp;
}

// A utility function to do inorder traversal of BST
voidinorder(structNode* root)
{
    if(root != NULL) {
        inorder(root->left);
        printf("%d ", root->key);
        inorder(root->right);
    }
}

/* A utility function to insert a new node with given key in
 * BST */
structNode* insert(structNode* node, intkey)
{
    /* If the tree is empty, return a new node */
    if(node == NULL)
        returnnewNode(key);

    /* Otherwise, recur down the tree */
    if(key < node->key)
        node->left = insert(node->left, key);
    else
        node->right = insert(node->right, key);

    /* return the (unchanged) node pointer */
    returnnode;
```

```
}

/* Given a binary search tree and a key, this function
   deletes the key and returns the new root */
structNode* deleteNode(structNode* root, intk)
{
    // Base case
    if(root == NULL)
        returnroot;

    // Recursive calls for ancestors of
    // node to be deleted
    if(root->key > k) {
        root->left = deleteNode(root->left, k);
        returnroot;
    }
    elseif(root->key < k) {
        root->right = deleteNode(root->right, k);
        returnroot;
    }

    // We reach here when root is the node
    // to be deleted.

    // If one of the children is empty
    if(root->left == NULL) {
        structNode* temp = root->right;
        free(root);
        returntemp;
    }
    elseif(root->right == NULL) {
        structNode* temp = root->left;
        free(root);
        returntemp;
    }

    // If both children exist
    else{

        structNode* succParent = root;

        // Find successor
        structNode* succ = root->right;
        while(succ->left != NULL) {
            succParent = succ;
            succ = succ->left;
```

```
        }

        // Delete successor.  Since successor
        // is always left child of its parent
        // we can safely make successor's right
        // right child as left of its parent.
        // If there is no succ, then assign
        // succ->right to succParent->right
        if(succParent != root)
            succParent->left = succ->right;
        else
            succParent->right = succ->right;

        // Copy Successor Data to root
        root->key = succ->key;

        // Delete Successor and return root
        free(succ);
        returnroot;
    }
}

// Driver Code
intmain()
{
    /* Let us create following BST
              50
           /     \
          30      70
         /  \    /  \
        20   40  60   80 */
    structNode* root = NULL;
    root = insert(root, 50);
    root = insert(root, 30);
    root = insert(root, 20);
    root = insert(root, 40);
    root = insert(root, 70);
    root = insert(root, 60);

    printf("Original BST: ");
    inorder(root);

    printf("\n\nDelete a Leaf Node: 20\n");
    root = deleteNode(root, 20);
    printf("Modified BST tree after deleting Leaf Node:\n");
    inorder(root);
```

```
    printf("\n\nDelete Node with single child: 70\n");
    root = deleteNode(root, 70);
    printf("Modified BST tree after deleting single child Node:\n");
    inorder(root);

    printf("\n\nDelete Node with both child: 50\n");
    root = deleteNode(root, 50);
    printf("Modified BST tree after deleting both child Node:\n");
    inorder(root);

    return0;
}
```

Output

Original BST: 20 30 40 50 60 70

Delete a Leaf Node: 20

Modified BST tree after deleting Leaf Node:

30 40 50 60 70

Delete Node with single child: 70

Modified BST tree after deleting single child No...

Time Complexity: O(h), where h is the height of the BST. **Auxiliary Space:** O(n).