```
// Practical - 1 SY BTECH, SCET, MIT AOE, Alandi(D), Sem - II, 2020 - 21
// BST Implementation Insert, Delete, Recursive Traversals, Non Recursive Traversals,
// Binary Search Traversals / Level order Printing.ST etc.

#include <bits/stdc++.h>
#include <iostream>

using namespace std;

class Node{
public:
   int data;
   Node* left;
   Node* right;
   Node(int d){
      data = d;
      left = NULL;
      right = NULL;
   }

   Node* search(Node* root, int key) {
      if(root == NULL || root->data == key)
         return root;

      // Key is greater than root's data
      if(root->data < key)
         return search(root->right,key);

      // Key is smaller than root's data
      return search(root->left,key);
   }
```

```
Node* insert(Node* root, int data) {
    if(root == NULL){
        return new Node(data);
    }
 else{
        Node* cur;
        if(data <= root->data) {
            cur = insert(root->left, data);
            root->left = cur;
        }
        else {
            cur = insert(root->right, data);
            root->right = cur;
        }
    return root;
  }
}

Node* deletenode(Node* root, int k)
 {
    // Base case
    if (root == NULL)
        return root;

    //If root->data is greater than k then we delete the root's subtree
    if(root->data > k){
        root->left = deletenode(root->left, k);
        return root;
    }
    else if(root->data < k){
```

```
    root->right = deletenode(root->right, k);
    return root;
}

// If one of the children is empty
if (root->left == NULL) {
    Node* temp = root->right;
    delete root;
    return temp;
}
else if (root->right == NULL) {
    Node* temp = root->left;
    delete root;
    return temp;
}
else {
  Node* Parent = root;
  // Find successor of the node
  Node *succ = root->right;
  while (succ->left != NULL) {
      Parent = succ;
      succ = succ->left;
    }

    if (Parent != root)
        Parent->left = succ->right;
    else
        Parent->right = succ->right;

    // Copy Successor Data
    root->data = succ->data;
```

```
        // Delete Successor and return root
        delete succ;
        return root;
    }
}

void inorder(Node* root){
    if(root == NULL)
        return;

    //First recur on left subtree
    inorder(root->left);
    //Then read the data of child
    cout << root->data << " ";
    // Recur on the right subtree
    inorder(root->right);
}
void preorder(Node* root){
    if(root == NULL)
        return;

    //First read the data of child
    cout << root->data << " ";
    //Then recur on left subtree
    preorder(root->left);
    //Then Recur on the right subtree
    preorder(root->right);
}
void postorder(Node* root){
    if(root == NULL)
```

```
        return;

    //Then recur on left subtree
    postorder(root->left);
    //Then Recur on the right subtree
    postorder(root->right);
    //First read the data of child
    cout << root->data << " ";
}

// Non recursive Inorder Traversal using Stack.
void inOrder(Node *root)
{
    stack<Node *> s;
    Node *curr = root;

    while (curr != NULL || s.empty() == false)
    {
        /* Reach the left most Node of the
        curr Node */
        while (curr !=  NULL)
        {
            /* place pointer to a tree node on
            the stack before traversing
            the node's left subtree */
            s.push(curr);
            curr = curr->left;
        }

        /* Current must be NULL at this point */
        curr = s.top();
```

```
        s.pop();

        cout << curr->data << " ";

        /* we have visited the node and its
        left subtree.  Now, it's right
        subtree's turn */
        curr = curr->right;
    } /* end of while */
}

// Non recursive Preorder Traversal using Stack.
void preOrder(Node* root)
{
    // Base Case
    if (root == NULL)
        return;

    // Create an empty stack and push root to it
    stack<Node *> s;
    s.push(root);

    /* Pop all items one by one. Do following for every popped item a) print it
       b) push its right child c) push its left child Note that right child is pushed first so that left is
processed first */

    while (s.empty() == false) {
        // Pop the top item from stack and print it
        Node* node = s.top();
        cout<<node->data<<" ";
        s.pop();
```

```
        // Push right and left children of the popped node to stack
        if (node->right)
            s.push(node->right);
        if (node->left)
            s.push(node->left);
    }
}

/*
// Non recursive Postorder Traversal using Stack.
void postOrder(Node* root)
{
    // Check for empty tree
    if (root == NULL)
        return;

    //struct Stack* stack = createStack(MAX_SIZE);
    stack<Node *> s;

    do
    {
        // Move to leftmost node
        while (root)
        {
            // Push root's right child and then root to stack.
            if (root->right)
                //push(stack, root->right);
                s.push(root->right);

            //push(stack, root);
```

```
            s.push(root);

            // Set root as root's left child
            root = root->left;
        }

        // Pop an item from stack and set it as root
        //root = pop(stack);
        root=s.top();
        s.pop();

        // If the popped item has a right child and the right child is not
        // processed yet, then make sure right child is processed before root

        // plese think on this as this is peep

        //Node *curr=s.top();

        if (root->right && s.top() == root->right)
        {
            s.pop(); // remove right child from stack
            s.push(root); // push root back to stack
            root = root->right; // change root so that the right child is processed next
        }
        else // Else print root's data and set root as NULL
        {
            cout<<root->data<<" ";
            root = NULL;
        }
    } while (!s.empty());
}
```

```
*/

// Non recursive Postorder Traversal using two Stack.
void postOrder(Node *root)
{
   // Base Case
   if (root == NULL) return;

   // Create two empty stack for post order traversal

   stack<Node *> s1;

   stack<Node *> s2;

   // Enqueue Root and initialize height
   s1.push(root);

   while (s1.empty() == false)
   {
      Node *node = s1.top();

      s1.pop();

      s2.push(node);

      if (node->left != NULL)
         s1.push(node->left);

      if (node->right != NULL)
         s1.push(node->right);
   }
```

```
    while(!s2.empty())
    {
        Node *n= s2.top();
        cout<<n->data<<" ";
        s2.pop();
    }
}

// Level Order Printing
void printLevelOrder(Node *root)
{
    // Base Case
    if (root == NULL) return;

    // Create an empty queue for level order tarversal
    queue<Node *> q;

    // Enqueue Root and initialize height
    q.push(root);

    while (q.empty() == false)
    {
        // nodeCount (queue size) indicates number
        // of nodes at current lelvel.
        int nodeCount = q.size();

        // Dequeue all nodes of current level and
        // Enqueue all nodes of next level
        cout<<" ===> ";
        while (nodeCount > 0)
        {
```

```
            Node *node = q.front();
            cout<<node->data<<" ";
            q.pop();
            if (node->left != NULL)
               q.push(node->left);
            if (node->right != NULL)
               q.push(node->right);
            nodeCount--;
         }
      }
   }
};

int main()
{
   Node Tree(0);

   Node* root = NULL;
   int choice;
   char ch='Y';

   while(ch=='Y'||ch=='y')
   {
      cout<<endl<<"\t You Can Perform Following Operations on Binary Search Tree :-"<<endl;
      cout<<endl<<"\t\t 1. Create"<<endl<<"\t\t 2. Insert"<<endl<<"\t\t 3. Delete"<<endl<<"\t\t 4.
Search"<<endl<<"\t\t 5. Display"<<endl<<"\t\t 6. Breadth First Search / LOP"<<endl<<"\t\t 7.
Quit"<<endl;
      cout<<endl<<"\t Enter your Choice :- ";
      cin>>choice;
      switch(choice)
      {
         case 1:
```

```
    //Number of nodes to be inserted
    int t;
    cout<<endl<<"\t Enter number of nodes you want to insert :- ";
    cin>>t;
    while(t--){
       int data;
       cout<<endl<<"\t\t Enter "<<t<<" Element.....:- ";
       cin>>data;
       root = Tree.insert(root,data);
    }
    cout<<endl<<"\t .....BST Constructed....."<<endl;
    break;
 case 2:
    t=0;
    cout<<endl<<"\t Enter data to insert in BST :- ";
    cin>>t;
    root = Tree.insert(root,t);
    cout<<endl<<"\t\t .....Elements Inserted.....";
    break;
 case 3:
    int delete_data;
    cout<<endl<<"\t Enter the node tobe Deleted :-";
    cin>>delete_data;
    //Node* d(0);
    Tree.deletenode(root,delete_data);
    cout<<endl<<"\t\t Inorder Traversal is of Tree is as :- ";
    Tree.inorder(root);
    break;
 case 4:
    // Searching
    if(root!=NULL){
```

```
        cout<<endl<<"\t\t Enter the data to Search in BST :";
        int data;
        cin>>data;
        //Node *s=Tree.search(root,data);
        if(Tree.search(root,data)!=NULL)
            cout<<endl<<"\t\t ......."<<data<<" is found in BST.....";
        else
            cout<<endl<<"\t\t ......."<<data<<" is not found in BST.....";
    }
    else
        cout<<endl<<"\t .....BST is Empty.....";
    break;
case 5:
    if(root!=NULL)
    {
        cout<<endl<<"\t Tree Traversals are as follws :- ";

        cout<<endl<<endl<<"\t\t Inorder Traversal of Tree is :- ";
        Tree.inorder(root);
        cout<<endl<<endl<<"\t\t Non-recursive Inorder Traversal of Tree is :- ";
        Tree.inOrder(root);

        cout<<endl<<endl<<"\t\t Preorder Traversal of Tree is :- ";
        Tree.preorder(root);
        cout<<endl<<endl<<"\t\t Non-recursive Preorder Traversal of Tree is :- ";
        Tree.preOrder(root);

        cout<<endl<<endl<<"\t\t Postorder Traversal of Tree is :- ";
        Tree.postorder(root);
        cout<<endl<<endl<<"\t\t Non-recursive Postorder Traversal of Tree is :- ";
        Tree.postOrder(root);
```

```
            }
            else
               cout<<endl<<"\t .............BST is Empty.............";
            break;
         case 6:
            if(root!=NULL)
            {
               cout<<endl<<"\t Breadth First Search Traversal (LOP) :- ";
               Tree.printLevelOrder(root);
            }
            else
               cout<<endl<<"\t .............BST is Empty.............";
            break;
         case 7: exit(0);
         default :
            cout<<endl<<"\t\t .............Invalid Choice.....Re-enter your choice";
            break;
      }

      cout<<endl<<endl<<"\t Do you want to continue (Y/N).........";
      cin>>ch;
      if(ch=='Y' || ch=='y')
         continue;
      else
         exit(0);
   }
   return 0;
}
```