**POST – LAB**

#include <iostream>

using namespace std;

class BNode

{

public:

BNode()

{data=0;}

void setLeftChild(BNode\* n)

{left=n;}

BNode\* getLeftChild()

{return left;}

void setRightChild(BNode\* n)

{ right=n;}

BNode\* getRightChild()

{return right;}

int getData()

{return data;}

public:

int data;

BNode\* left;

BNode\* right;

};

// Data structure to store a Binary Search Tree node

class BinarySearchTree

{

public:

int data;

BinarySearchTree \*left, \*right;

};

// Function to create a new binary tree node having given key

BinarySearchTree\* newNode(int key)

{

BinarySearchTree\* node = new BinarySearchTree;

node->data = key;

node->left = node->right = nullptr;

return node;

}

// Recursive function to insert a key into BST

BinarySearchTree\* insert(BinarySearchTree\* root, int key)

{

// if the root is null, create a new node and return it

if (root == nullptr)

return newNode(key);

// if given key is less than the root node, recur for left subtree

if (key < root->data)

root->left = insert(root->left, key);

// if given key is more than the root node, recur for right subtree

else

root->right = insert(root->right, key);

return root;

}

BinarySearchTree\* search(BinarySearchTree \* root,int key)

{ if (root == NULL || root->data == key)

return root;

if (root->data< key)

return search(root->right, key);

return search(root->left, key);

}

// Function to perform inorder traversal of the BST

void printSorted(BinarySearchTree \*root)

{

if (root == nullptr)

return;

printSorted(root->left);

cout << root->data << " ";

printSorted(root->right);

}

// Note that curr & parent are passed by reference

void searchKey(BinarySearchTree\* &curr, int key, BinarySearchTree\* &parent)

{

// traverse the tree and search for the key

while (curr != nullptr && curr->data != key)

{

// update parent node as current node

parent = curr;

// if given key is less than the current node, go to left subtree

// else go to right subtree

if (key < curr->data)

curr = curr->left;

else

curr = curr->right;

}

}

// Helper function to find minimum value node in subtree rooted at curr

BinarySearchTree\* minimumKey(BinarySearchTree\* curr)

{

while (curr->left != nullptr) {

curr = curr->left;

}

return curr;

}

// Function to delete node from a BST

void deleteNode(BinarySearchTree\*& root, int key)

{

// pointer to store parent node of current node

BinarySearchTree\* parent = nullptr;

// start with root node

BinarySearchTree\* curr = root;

// search key in BST and set its parent pointer

searchKey(curr, key, parent);

// return if key is not found in the tree

if (curr == nullptr)

return;

// Case 1: node to be deleted has no children i.e. it is a leaf node

if (curr->left == nullptr && curr->right == nullptr)

{

// if node to be deleted is not a root node, then set its

// parent left/right child to null

if (curr != root)

{

if (parent->left == curr)

parent->left = nullptr;

else

parent->right = nullptr;

}

// if tree has only root node, delete it and set root to null

else

root = nullptr;

// deallocate the memory

free(curr); // or delete curr;

}

// Case 2: node to be deleted has two children

else if (curr->left && curr->right)

{

// find its in-order successor node

BinarySearchTree\* successor = minimumKey(curr->right);

// store successor value

int val = successor->data;

// recursively delete the successor. Note that the successor

// will have at-most one child (right child)

deleteNode(root, successor->data);

// Copy the value of successor to current node

curr->data = val;

}

// Case 3: node to be deleted has only one child

else

{

// find child node

BinarySearchTree\* child = (curr->left)? curr->left: curr->right;

// if node to be deleted is not a root node, then set its parent

// to its child

if (curr != root)

{

if (curr == parent->left)

parent->left = child;

else

parent->right = child;

}

// if node to be deleted is root node, then set the root to child

else

root = child;

// deallocate the memory

free(curr);

}

}

int main()

{BinarySearchTree\* root = nullptr;

//creating an object of binary search tree

//BinarySearchTree \*BST=new BinarySearchTree();

//following insertions should happen successfully as we are inserting

//unique values

root=insert(root,12);

root=insert(root,4);

root=insert(root,9);

root=insert(root,2);

root=insert(root,14);

root=insert(root,16);

root=insert(root,13);

root=insert(root,1);

//prints data carried by the BST in sorted manner

printSorted(root);

cout<<endl;

cout<<"node carrying 16 deleted successfully"<<endl;

//deleting leaf node

deleteNode(root,16);

printSorted(root);

cout<<endl;

cout<<"node carrying 2 deleted successfully"<<endl;

//deleting degree 1 node

deleteNode(root,2);

printSorted(root);

cout<<endl;

cout<<"node carrying 12 deleted successfully"<<endl;

//deleting degree 2 node

deleteNode(root,12);

cout<<endl;

//prints data carried by the BST in sorted manner

printSorted(root);

cout<<endl;

cout<<"DESTRUCTOR IS CALLING !! deleting all nodes "<<endl;

//destructor called

delete root;

return 0;

}

**OUTPUT**

