**Experiment 8**

(PART B : TO BE COMPLETED BY STUDENTS)

***(Students must submit the soft copy as per following segments within two hours of the practical. The soft copy must be uploaded on the Blackboard or emailed to the concerned lab in charge faculties at the end of the practical in case the there is no Black board access available)***

|  |  |
| --- | --- |
| Roll No. C013 | Name: Ashmit Jain |
| Class : B | Batch : B1 |
| Date of Experiment: | Date of Submission: 28/09/2024 |
| Grade : | Time of Submission: |
| Date of Grading: |  |

**B.1 Software Code written by student:**

***(Paste your code completed during the 2 hours of practical in the lab here)***

**Task1:**

Write a C/C++ program to implement insertion and deletion on BST

#*include*<iostream>

using namespace std;

class Node {

public:

    int data;

    Node\* left;

    Node\* right;

*Node*(int value = 0) {

        data = value;

        left = nullptr;

        right = nullptr;

    }

};

class BST {

public:

// *Main root node of class Node*

    Node\* root;

// *Constructor to initialize the root node*

*BST*() {

        root = nullptr;

    }

    void *insert*(int value) {

        root = *insertData*(root, value);

    }

    bool *search*(int value) {

*return* *searchRec*(root, value);

    }

    void *inOrderTraversal*() {

*inOrderTraversal*(root);

        cout << endl;

    }

    void *deleteNode*(int value) {

        root = *deleteNodeRec*(root, value);

    }

private:

    Node*\** *insertData*(Node*\** node, int value) {

// *If the tree is empty, create a new node and return it*

*if* (node == nullptr) {

*return* new *Node*(value);

        }

// *Otherwise, recur down the tree*

*if* (value < node->data) {

            node->left = *insertData*(node->left, value);

        } *else* *if* (value > node->data) {

            node->right = *insertData*(node->right, value);

        }

// *Return the (unchanged) node pointer*

*return* node;

    }

    bool *searchRec*(Node*\** node, int value) {

// *Base case: root is null or value is present at root*

*if* (node == nullptr) {

*return* false;

        }

*if* (node->data == value) {

*return* true;

        }

// *Value is greater than root's data*

*if* (value < node->data) {

*return* *searchRec*(node->left, value);

        }

// *Value is smaller than root's data*

*return* *searchRec*(node->right, value);

    }

    void *inOrderTraversal*(Node*\** node) {

*if* (node != nullptr) {

*inOrderTraversal*(node->left);

            cout << node->data << " -> ";

*inOrderTraversal*(node->right);

        }

    }

// *Helper function to find the minimum value node in a subtree*

    Node*\** *findMin*(Node*\** node) {

*while* (node && node->left != nullptr) {

            node = node->left;

        }

*return* node;

    }

// *Function to delete a node from the BST*

    Node*\** *deleteNodeRec*(Node*\** node, int value) {

*if* (node == nullptr) {

*return* node;

        }

// *Traverse the tree to find the node to be deleted*

*if* (value < node->data) {

            node->left = *deleteNodeRec*(node->left, value);

        } *else* *if* (value > node->data) {

            node->right = *deleteNodeRec*(node->right, value);

        } *else* {

// *Node to be deleted found*

*if* (node->left == nullptr) {

                Node\* temp = node->right;

                delete node;

*return* temp;

            } *else* *if* (node->right == nullptr) {

                Node\* temp = node->left;

                delete node;

*return* temp;

            }

// *Node with two children: get the inorder successor (smallest in the right subtree)*

            Node\* temp = *findMin*(node->right);

// *Copy the inorder successor's data to this node*

            node->data = temp->data;

// *Delete the inorder successor*

            node->right = *deleteNodeRec*(node->right, temp->data);

        }

*return* node;

    }

};

int *main*() {

    BST tree;

    int inp;

    char choice;

// *Inserting nodes into the tree*

    tree.*insert*(5);

    tree.*insert*(3);

    tree.*insert*(7);

    tree.*insert*(2);

    tree.*insert*(4);

    tree.*insert*(6);

    tree.*insert*(8);

    cout << "Binary Search Tree created with nodes: 5, 3, 7, 2, 4, 6, 8" << endl;

    tree.*inOrderTraversal*();

    cout << endl;

*while* (true) {

        cout << "\nOptions:\n";

        cout << "S: Search\n";

        cout << "I: Insert\n";

        cout << "D: Delete\n";

        cout << "P: Print in-order traversal\n";

        cout << "Q: Quit\n";

        cout << "Enter your choice: ";

        cin >> choice;

*switch* (choice) {

*case* 'S':

*case* 's':

                cout << "Input value to search: ";

                cin >> inp;

*if* (tree.*search*(inp)) {

                    cout << inp << " found in the tree." << endl;

                } *else* {

                    cout << inp << " not found in the tree." << endl;

                }

*break*;

*case* 'I':

*case* 'i':

                cout << "Input value to insert: ";

                cin >> inp;

                tree.*insert*(inp);

                cout << inp << " inserted into the tree." << endl;

*break*;

*case* 'D':

*case* 'd':

                cout << "Input value to delete: ";

                cin >> inp;

                tree.*deleteNode*(inp);

                cout << inp << " deleted from the tree." << endl;

*break*;

*case* 'P':

*case* 'p':

                cout << "In-order Traversal: ";

                tree.*inOrderTraversal*();

*break*;

*case* 'Q':

*case* 'q':

*return* 0;

*default*:

                cout << "Invalid choice. Try again." << endl;

*break*;

        }

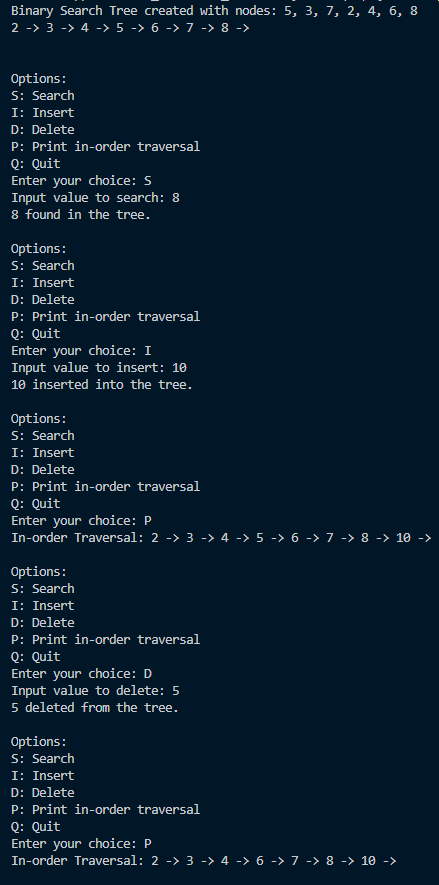
    }

*return* 0;

}

**B.2 Input and Output:**

**Task1:**

****

**B.3 Observations and learning [w.r.t. all tasks]:**

In this experiment, we successfully implemented the insertion and deletion operations on a Binary Search Tree (BST). The program adheres to the basic properties of a BST, where the left subtree contains nodes with keys lesser than the root, and the right subtree contains nodes with keys greater than the root. The insert() and deleteNode() functions were implemented to handle the following cases efficiently:

1. **Insertion**: The insertion function recursively finds the correct position in the tree and inserts a new node as a leaf in its appropriate place.
2. **Deletion**: The deletion function addresses three distinct cases:
   * **Case 1**: Deletion of a node with no children (a leaf node).
   * **Case 2**: Deletion of a node with one child (either left or right).
   * **Case 3**: Deletion of a node with two children, which involves replacing the node with its in-order successor (the smallest node in its right subtree) and deleting the successor.

The program also provides functionality to perform in-order traversal, which verifies the correctness of the tree's structure after insertion and deletion operations.

**B.4 Conclusion:**

By completing this experiment, we gained a clear understanding of how to implement and manage the insertion and deletion operations in a Binary Search Tree. The handling of special cases during deletion (such as nodes with one or two children) enhances the robustness of the tree structure. The experiment provided practical insights into tree-based algorithms, strengthening our knowledge of data structures and their operations. After completion, students are equipped to implement efficient binary search trees with insertion and deletion functionalities.

**B.5 Question of Curiosity**

Write algorithm to perform insertion and deletion (case 3: Node with 2 child nodes)

### Algorithm for Insertion in a Binary Search Tree (BST):

**Steps for Insertion:**

1. **Start at the root node** of the BST.
2. **Compare the value** to be inserted with the current node's data.
   * If the value is **less than** the current node's data:
     + Move to the **left child**.
   * If the value is **greater than** the current node's data:
     + Move to the **right child**.
3. **Repeat the comparison** until you reach a nullptr (an empty position in the tree).
4. Insert the new node in that position as a **leaf node**.

### Algorithm for Deletion in a Binary Search Tree (Case 3: Node with Two Children):

**Steps for Deleting a Node with Two Children:**

1. **Start at the root node** of the BST.
2. **Search for the node** to be deleted by comparing its value with the current node's data.
   * If the value is **less than** the current node’s data, move to the **left child**.
   * If the value is **greater than** the current node’s data, move to the **right child**.
   * If the value matches the current node, this is the node to be deleted.
3. **Check if the node has two children** (i.e., both left and right subtrees are non-null).
4. **Find the in-order successor**:
   * The in-order successor is the **smallest node in the right subtree** of the node to be deleted.
   * Start from the node's right child and keep moving to the left child until you find a node with no left child.
5. **Replace the value** of the node to be deleted with the value of the in-order successor.
6. **Delete the in-order successor** node:
   * Since the in-order successor will either have no children or one child, delete it following the appropriate deletion case (no child or one child).
7. **Return the updated tree.**

