FEU INSTITUTE OF TECHNOLOGY

COLLEGE OF COMPUTER STUDIES AND MULTIMEDIA ARTS

CCS0015L (DATA STRUCTURES AND ALGORITHMS)

EXERCISE 7

BINARY SEARCH TREE

Student Name / Group Name:	Fernando Dane I. Borja
Members (if Group):	**Name** / **Role**
Section:	TC05 BSIT – CST
Professor:	Sir. Joseph Calleja

Name	Role

I. PROGRAM OUTCOME/S (PO) ADDRESSED BY THE LABORATORY EXERCISE

- Identify, analyze and solve computing problems using fundamental principles of mathematics and computing sciences. [PO: B]

II. COURSE LEARNING OUTCOME/S (CLO) ADDRESSED BY THE LABORATORY EXERCISE

- Apply the fundamental principles of data structures and algorithms: concepts of abstract data; types of common data structures used; description, properties, and storage allocation of data structures. [CLO: 2]

III. INTENDED LEARNING OUTCOME/S (ILO) OF THE LABORATORY EXERCISE

At the end of this exercise, students must be able to:

- Learn how to create a template version of the Binary Search Tree ADT.
- Apply different operations on Binary Search Tree.
- Solve programming problems that implement Binary Search Tree.

IV. BACKGROUND INFORMATION

A binary tree is a special kind of tree in which each node can have at most two children: they are distinguished as a left child and a right child. The subtree rooted at the left child of a node is called its left subtree and the subtree rooted at the right child of a node is called its right subtree.

Searching a Binary Search Tree

Suppose we wish to search for an element with key x. We are at the root. If the root is 0, then the search tree contains no elements, and the search terminates unsuccessfully. Otherwise, we compare x with key in root. If x equals key in root, then search terminates successfully. If x is less than key in root, then no element in right sub tree can have key value x, and only left subtree is to be searched. If x is larger than key in root, the no element in left subtree can have the key x, and only right subtree is to be searched. The subtrees can be searched recursively.

Insertion into a Binary Search Tree

To insert an element x, we must first verify that its key is different from those of existing elements. To do this, a search is carried out. If the search is unsuccessful, then the element is inserted at the point where the search terminated.

Deleting from a Binary Search Tree

Deletion from a leaf element is achieved by simply removing the leaf node and making its parent's child field to be null. Other cases are deleting a node with one subtree and two subtrees.

V. LABORATORY ACTIVITY

Directions: Complete the program below that satisfy the given problem specification and output. Look for the functions with //TODO. Paste your completed source code in the Code portion then highlight the statement(s) that you filled in the incomplete program with color yellow.

ACTIVITY 7.1: Binary Search Tree

Binary search tree is a data structure that quickly allows us to maintain a sorted list of numbers. It is called a binary tree because each tree node has a maximum of two children.

The binary search tree operations are:

- insert(int key): Inserts a new node with the given key value into the binary search tree. If the key value already exists in the tree, the function does nothing.
- search(int key): Searches the binary search tree for a node with the given key value. If a node with the given key value is found, the function returns a pointer to the node. Otherwise, the function returns a NULL pointer.
- remove(int key): Removes the node with the given key value from the binary search tree. If the node with the given key value is not found in the tree, the function does nothing.
- inorder(): Performs an in-order traversal of the binary search tree, printing the key values of each node in ascending order.
- preorder(): Performs a pre-order traversal of the binary search tree, printing the key values of each node in the order that they are visited.
- postorder(): Performs a post-order traversal of the binary search tree, printing the key values of each node after visiting all of its children.
- display(): Displays the binary search tree as a tree structure, with each node represented by its key value and indented according to its level in the tree.

Given Program:

```
#include <iostream>

using namespace std;

class Node {
public:
    int key;
    Node* left;
    Node* right;

    Node(int key) {
        this->key = key;
        this->left = NULL;
        this->right = NULL;
    }
};

class BST {
private:
    Node* root;

    void insertNode(Node*& node, int key) {
        if (node == NULL) {
            node = new Node(key);
            return;
        }

        if (key < node->key) {
            insertNode(node->left, key);
        } else if (key > node->key) {
            insertNode(node->right, key);
        }
    }

    Node* searchNode(Node* node, int key) {
   //TODO
    }

    Node* findMin(Node* node) {
        while (node->left != NULL) {
            node = node->left;
        }

        return node;
    }

    Node* deleteNode(Node*& node, int key) {
        if (node == NULL) {
            return node;
        }

        if (key < node->key) {
            node->left = deleteNode(node->left, key);
        } else if (key > node->key) {
            node->right = deleteNode(node->right, key);
        } else {
            if (node->left == NULL) {
                Node* temp = node->right;
                delete node;
                return temp;
            } else if (node->right == NULL) {
                Node* temp = node->left;
                delete node;
                return temp;
            }

            Node* temp = findMin(node->right);
            node->key = temp->key;
            node->right = deleteNode(node->right, temp->key);
        }

        return node;
    }

    void inorderTraversal(Node* node) {
   //TODO
    }

    void preorderTraversal(Node* node) {
   //TODO
    }

    void postorderTraversal(Node* node) {
   //TODO
    }

    void displayBST(Node* node, int space) {
        if (node == NULL) {
            return;
        }

        space += 5;
        displayBST(node->right, space);

        cout << endl;
        for (int i = 5; i < space; i++) {
            cout << " ";
        }
        cout << node->key << "\n";

        displayBST(node->left, space);
    }

public:
    BST() {
        this->root = NULL;
    }

    void insert(int key) {
        insertNode(this->root, key);
    }

    Node* search(int key) {
        return searchNode(this->root, key);
    }
    void remove(int key) {
        deleteNode(this->root, key);
    }

    void inorder() {
        inorderTraversal(this->root);
cout << endl;
}

void preorder() {
    preorderTraversal(this->root);
    cout << endl;
}

void postorder() {
    postorderTraversal(this->root);
    cout << endl;
}

void display() {
    displayBST(this->root, 0);
}
};

int main() {
BST bst;
bst.insert(50);
bst.insert(30);
bst.insert(70);
bst.insert(20);
bst.insert(40);
bst.insert(60);
bst.insert(80);

cout << "Inorder traversal: ";
bst.inorder();

cout << "Preorder traversal: ";
bst.preorder();

cout << "Postorder traversal: ";
bst.postorder();

cout << "Displaying BST:\n";
bst.display();

Node* searchedNode = bst.search(60);
if (searchedNode == NULL) {
    cout << "Node not found in the BST\n";
} else {
    cout << "Node found in the BST: " << searchedNode->key << endl;
}
```

```
bst.remove(30);

cout << "Inorder traversal after deleting a node: ";
bst.inorder();

return 0;
}
```

Sample Output:

```
Inorder traversal: 20 30 40 50 60 70 80
Preorder traversal: 50 30 20 40 70 60 80
Postorder traversal: 20 40 30 60 80 70 50
Displaying BST:

          80

     70

          60

50

          40

     30

          20
Node found in the BST: 60
Inorder traversal after deleting a node: 20 40 50 60 70 80
```

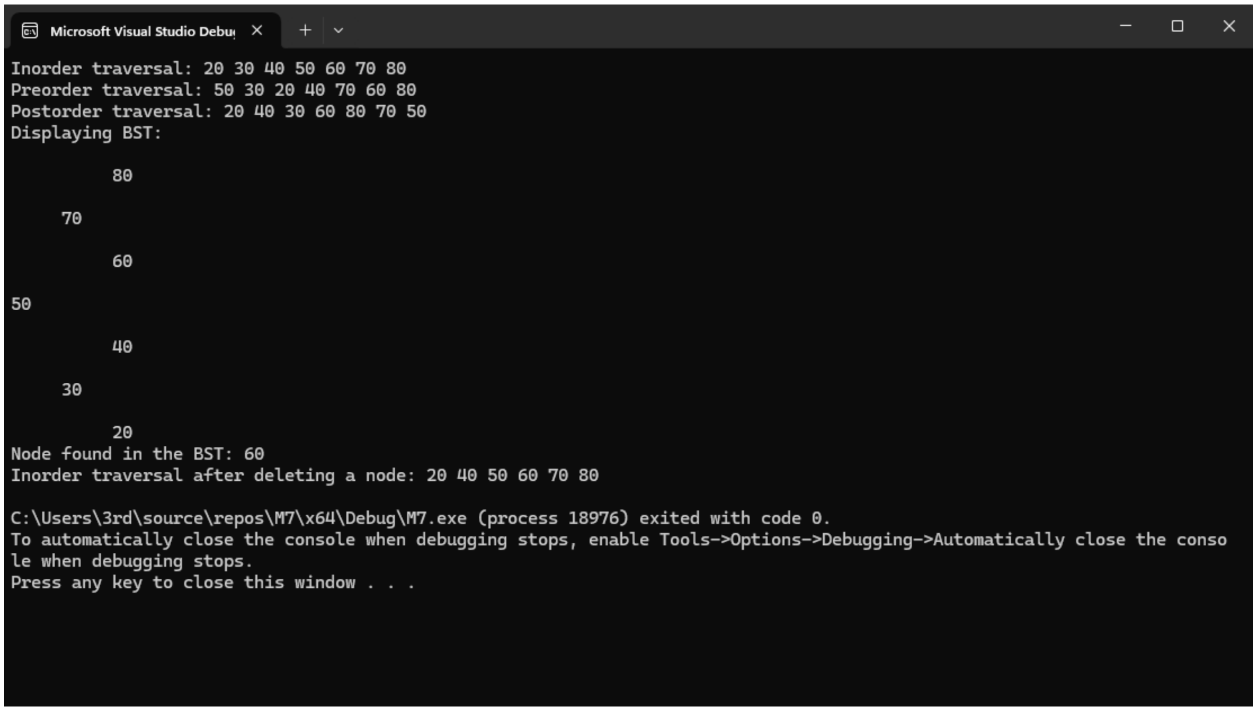
Code:

```
#include <iostream>

using namespace std;

class Node {
public:
    int key;
    Node* left;
    Node* right;

    Node(int key) {
        this->key = key;
        this->left = NULL;
        this->right = NULL;
    }
};
```

```cpp
class BST {
private:
    Node* root;

    void insertNode(Node*& node, int key) {
        if (node == NULL) {
            node = new Node(key);
            return;
        }

        if (key < node->key) {
            insertNode(node->left, key);
        }
        else if (key > node->key) {
            insertNode(node->right, key);
        }
    }

    Node* searchNode(Node* node, int key) {
        if (node == NULL || node->key == key) {
            return node;
        }

        if (key < node->key) {
            return searchNode(node->left, key);
        }
        else {
            return searchNode(node->right, key);
        }
    }

    Node* findMin(Node* node) {
        while (node->left != NULL) {
            node = node->left;
        }

        return node;
    }

    Node* deleteNode(Node*& node, int key) {
        if (node == NULL) {
            return node;
        }

        if (key < node->key) {
            node->left = deleteNode(node->left, key);
        }
        else if (key > node->key) {
            node->right = deleteNode(node->right, key);
        }
        else {
            if (node->left == NULL) {
                Node* temp = node->right;
                delete node;
                return temp;
            }
            else if (node->right == NULL) {
                Node* temp = node->left;
                delete node;
                return temp;
            }

            Node* temp = findMin(node->right);
            node->key = temp->key;
            node->right = deleteNode(node->right, temp->key);
        }

        return node;
    }

    void inorderTraversal(Node* node) {
        if (node == NULL) {
            return;
        }

        inorderTraversal(node->left);
        cout << node->key << " ";
        inorderTraversal(node->right);
    }

    void preorderTraversal(Node* node) {
        if (node == NULL) {
            return;
        }

        cout << node->key << " ";
        preorderTraversal(node->left);
        preorderTraversal(node->right);
    }

    void postorderTraversal(Node* node) {
        if (node == NULL) {
            return;
        }

        postorderTraversal(node->left);
        postorderTraversal(node->right);
        cout << node->key << " ";
    }

    void displayBST(Node* node, int space) {
        if (node == NULL) {
            return;
        }

        space += 5;
        displayBST(node->right, space);

        cout << endl;
        for (int i = 5; i < space; i++) {
```

```
                cout << " ";
            }
            cout << node->key << "\n";

            displayBST(node->left, space);
        }

public:
    BST() {
        this->root = NULL;
    }

    void insert(int key) {
        insertNode(this->root, key);
    }

    Node* search(int key) {
        return searchNode(this->root, key);
    }

    void remove(int key) {
        deleteNode(this->root, key);
    }

    void inorder() {
        inorderTraversal(this->root);
        cout << endl;
    }

    void preorder() {
        preorderTraversal(this->root);
        cout << endl;
    }

    void postorder() {
        postorderTraversal(this->root);
        cout << endl;
    }

    void display() {
        displayBST(this->root, 0);
    }
};

int main() {
    BST bst;
    bst.insert(50);
    bst.insert(30);
    bst.insert(70);
    bst.insert(20);
    bst.insert(40);
    bst.insert(60);
    bst.insert(80);

    cout << "Inorder traversal: ";
    bst.inorder();

    cout << "Preorder traversal: ";
    bst.preorder();

    cout << "Postorder traversal: ";
    bst.postorder();

    cout << "Displaying BST:\n";
    bst.display();

    Node* searchedNode = bst.search(60);
    if (searchedNode == NULL) {
        cout << "Node not found in the BST\n";
    }
    else {
        cout << "Node found in the BST: " << searchedNode->key << endl;
    }

    bst.remove(30);

    cout << "Inorder traversal after deleting a node: ";
    bst.inorder();

    return 0;
}
```

Output:(screenshot of the output)



Microsoft Visual Studio Debu

```
Inorder traversal: 20 30 40 50 60 70 80
Preorder traversal: 50 30 20 40 70 60 80
Postorder traversal: 20 40 30 60 80 70 50
Displaying BST:

          80

     70

          60

50

          40

     30

          20
Node found in the BST: 60
Inorder traversal after deleting a node: 20 40 50 60 70 80

C:\Users\3rd\source\repos\M7\x64\Debug\M7.exe (process 18976) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .
```

VI. QUESTION AND ANSWER

Directions: Briefly answer the questions below.

• Differentiate binary tree and binary search tree. A binary search tree (BST) preserves the characteristic that the left child is bigger than the parent and the right child is smaller than the parent. A binary tree is a hierarchical data structure in which each node has a maximum of two children.
• Discuss why searching for an element in the Binary search tree is easy. Finding an element in the Binary search tree is simple as we can always tell which subtree contains the thing we're looking for. Insertion and deletion operations in BST are faster than in arrays and linked lists.

VII. REFERENCES

- Wittenberg, Lee.(2018). Data structures and algorithms in C++. s.l.: Mercury Learning
- Baka, Benjamin(2017). Python data structures and algorithms : improve the performance and speed of your applications. Birmingham, U.K : Packt Publishing
- Downey, Allen.(2017). Think data structures : algorithms and information retrieval in Java. Sebastopol, CA: O'Reilly
- Chang, Kung-Hua(2017). Data Structures Practice Problems for C++ Beginners. S.I : Simple and Example
- Hemant Jain(2017). Problem Solving in Data Structures & Algorithms Using C++: Programming Interview Guide. USA: CreateSpace Independent Publishing Platform

RUBRIC:

Criteria	**4**	**3**	**2**	**1**	**Score**
Solution(x5)	A completed solution runs without errors. It meets all the specifications and works for all test data.	A completed solution is tested and runs but does not meet all the specifications nd/or work for all test data.	A completed solution is implemented on the required platform, and uses the compiler specified. It runs, but has logical errors.	An incomplete solution is implemented on the required platform. It does not compile and/or run.	
Program Design(x3)	The program design uses appropriate structures. The overall program design is appropriate.	The program design generally uses appropriate structures. Program elements exhibit good design.	Not all of the selected structures are appropriate. Some of the program elements are appropriately designed.	Few of the selected structures are appropriate. Program elements are not well designed.	
Completeness of Document(x2)	All required parts of the document are complete and correct(code, output of screenshots)	All required parts in the document are present and correct but not complete.	There are few parts of the document are missing but the rest are complete and correct.	Most of the parts of the document are missing and incorrect.	

					Total