DSA LAB 09

Name: Syed Wahaaj Ali
Roll No: CT-24035
Course Code: CT-159
CSIT
Section: A

Q1:

1. Define following methods in Example 01 by using stack.
    - Preorder traversal
    - Postorder travsersal

Code:

```
#include <iostream>
#include <stack>
using namespace std;

/*
 Syed Wahaaj Ali
 CT-24035
*/

class TreeNode {
	public:
		int val;
		TreeNode* left;
		TreeNode* right;

		TreeNode(int value) {
			val = value;
			left = nullptr;
			right = nullptr;
		}
};

class BinarySearchTree {
	public:
		TreeNode* root;

		BinarySearchTree() {
			root = nullptr;
		}
```

```
void insert(int key) {
	root = insertRec(root, key);
}

void deleteNode(int key) {
	root = deleteRec(root, key);
}

void inorder() {
	inorderRec(root);
	cout << endl;
}

void preorder() {
	if (root == nullptr) return;
	stack<TreeNode*> st;
	st.push(root);
	while (!st.empty()) {
		TreeNode* node = st.top();
		st.pop();
		cout << node->val << " ";
		if (node->right) st.push(node->right);
		if (node->left) st.push(node->left);
	}
	cout << endl;
}

void postorder() {
	if (root == nullptr) return;
	stack<TreeNode*> st1, st2;
	st1.push(root);
	while (!st1.empty()) {
		TreeNode* node = st1.top();
		st1.pop();
		st2.push(node);
		if (node->left) st1.push(node->left);
		if (node->right) st1.push(node->right);
	}
	while (!st2.empty()) {
		cout << st2.top()->val << " ";
		st2.pop();
	}
	cout << endl;
}

TreeNode* insertRec(TreeNode* node, int key) {
```

```
        if (node == nullptr)
                return new TreeNode(key);
        if (key < node->val)
                node->left = insertRec(node->left, key);
        else if (key > node->val)
                node->right = insertRec(node->right, key);
        return node;
}

TreeNode* deleteRec(TreeNode* node, int key) {
        if (node == nullptr)
                return node;
        if (key < node->val)
                node->left = deleteRec(node->left, key);
        else if (key > node->val)
                node->right = deleteRec(node->right, key);
        else {
                if (node->left == nullptr) {
                        TreeNode* inorderSuccessor = node->right;
                        delete node;
                        return inorderSuccessor;
                } else if (node->right == nullptr) {
                        TreeNode* inorderSuccessor = node->left;
                        delete node;
                        return inorderSuccessor;
                }
                TreeNode* inorderSuccessor = minValueNode(node->right);
                node->val = inorderSuccessor->val;
                node->right = deleteRec(node->right, inorderSuccessor->val);
        }
        return node;
}

TreeNode* minValueNode(TreeNode* node) {
        TreeNode* current = node;
        while (current && current->left != nullptr)
                current = current->left;
        return current;
}

void inorderRec(TreeNode* root) {
        if (root != nullptr) {
                inorderRec(root->left);
                cout << root->val << " ";
                inorderRec(root->right);
        }
```

```
                }
};

int main() {
        BinarySearchTree b;
        b.insert(50);
        b.insert(30);
        b.insert(20);
        b.insert(40);
        b.insert(70);
        b.insert(60);
        b.insert(80);

        cout << "Inorder: ";
        b.inorder();
        cout << "Preorder: ";
        b.preorder();
        cout << "Postorder: ";
        b.postorder();
}
```

Output:

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


=== Code Execution Successful ===
```

Q2:

2. You are tasked with designing an employee management system for a small company. Each employee has a unique ID, name, and department. You are required to store and manage employee records in a way that allows quick insertion, deletion, and search operations based on the employee ID. You decide to use a Binary Search Tree (BST) to store the employee records. Each node in the BST will store: Employee ID (used as the key for the BST), Employee Name, Employee Department.

- Create a C++ class EmployeeNode representing each employee in the BST. Each node should store: int id (Employee ID), string name (Employee Name), string department (Employee Department), A pointer to the left child (EmployeeNode* left), a pointer to the right child (EmployeeNode* right).
- Create a class EmployeeBST with the following member functions:
    - insert(int id, string name, string department): Inserts a new employee into the BST.
    - search(int id): Searches for an employee by ID. If found, return their name and department; otherwise, print an appropriate message.
    - deleteNode(int id): Deletes an employee from the BST based on their ID.
    - inOrderTraversal(): Prints all employees in ascending order of their ID, showing their ID, name, and department.
    - findMin(): Returns the name and department of the employee with the smallest ID.
    - findMax(): Returns the name and department of the employee with the largest ID.
- Also, Handle edge cases such as: Inserting an employee with a duplicate ID, Deleting an employee that does not exist, Deleting nodes with no children, one child, and two children, Handling an empty tree for search or delete operations.

Code:

```
#include <iostream>
#include <string>
using namespace std;

/*
 Syed Wahaaj Ali
 CT-24035
*/

class EmployeeNode {
        public:
                int id;
                string name;
                string department;
                EmployeeNode* left;
                EmployeeNode* right;

                EmployeeNode(int id, string name, string department) {
                        this->id = id;
                        this->name = name;
                        this->department = department;
                        left = nullptr;
                        right = nullptr;
```

```
        }
};

class EmployeeBST {
    public:
        EmployeeNode* root;

        EmployeeBST() {
            root = nullptr;
        }

        void insert(int id, string name, string department) {
            EmployeeNode* newNode = new EmployeeNode(id, name, department);
            if (root == nullptr) {
                root = newNode;
                return;
            }

            EmployeeNode* current = root;
            EmployeeNode* parent = nullptr;

            while (current != nullptr) {
                parent = current;
                if (id < current->id)
                    current = current->left;
                else if (id > current->id)
                    current = current->right;
                else {
                    cout << "Duplicate ID not allowed: " << id << endl;
                    return;
                }
            }

            if (id < parent->id)
                parent->left = newNode;
            else
                parent->right = newNode;
        }

        void inOrderTraversal(EmployeeNode* node) {
            if (node == nullptr)
                return;
            inOrderTraversal(node->left);
            cout << "ID: " << node->id << " | Name: " << node->name << " | Department: " << node-
>department << endl;
            inOrderTraversal(node->right);
```

```
	}

	void inOrder() {
		if (root == nullptr) {
			cout << "Tree is empty.\n";
			return;
		}
		inOrderTraversal(root);
	}

	EmployeeNode* searchNode(EmployeeNode* node, int id) {
		if (node == nullptr)
			return nullptr;
		if (id == node->id)
			return node;
		else if (id < node->id)
			return searchNode(node->left, id);
		else
			return searchNode(node->right, id);
	}

	void search(int id) {
		EmployeeNode* result = searchNode(root, id);

		if (result == nullptr)
			cout << "Employee with ID " << id << " not found.\n";
		else
			cout << "Found ID: " << result->id << " | Name: " << result->name << " |
Department: " << result->department << endl;
	}

	EmployeeNode* findMinNode(EmployeeNode* node) {
		while (node->left != nullptr)
			node = node->left;
		return node;
	}

	void findMin() {
		if (root == nullptr) {
			cout << "Tree is empty.\n";
			return;
		}
		EmployeeNode* smallest = findMinNode(root);
		cout << "Min ID: " << smallest->id << " | Name: " << smallest->name << " | Department:
" << smallest->department << endl;
	}
```

```
void findMax() {
	if (root == nullptr) {
		cout << "Tree is empty.\n";
		return;
	}
	EmployeeNode* node = root;
	while (node->right != nullptr)
		node = node->right;
	cout << "Max ID: " << node->id << " | Name: " << node->name << " | Department: " <<
node->department << endl;
}

EmployeeNode* deleteRec(EmployeeNode* node, int id) {
	if (node == nullptr) {
		cout << "Employee not found: " << id << endl;
		return nullptr;
	}

	if (id < node->id)
		node->left = deleteRec(node->left, id);
	else if (id > node->id)
		node->right = deleteRec(node->right, id);
	else {
		if (node->left == nullptr && node->right == nullptr) {
			delete node;
			return nullptr;
		} else if (node->left == nullptr) {
			EmployeeNode* temp = node->right;
			delete node;
			return temp;
		} else if (node->right == nullptr) {
			EmployeeNode* temp = node->left;
			delete node;
			return temp;
		} else {
			EmployeeNode* successor = findMinNode(node->right);
			node->id = successor->id;
			node->name = successor->name;
			node->department = successor->department;
			node->right = deleteRec(node->right, successor->id);
		}
	}
	return node;
}
```

```
        void deleteNode(int id) {
            root = deleteRec(root, id);
        }
};

int main() {
    EmployeeBST t;
    t.insert(30, "Ali", "Sales");
    t.insert(15, "Ahmed", "IT");
    t.insert(40, "Ayaan", "Finance");
    t.insert(10, "Bilal", "HR");
    t.insert(50, "Sara", "Admin");

    cout << "\nInorder Traversal (Ascending by ID):\n";
    t.inOrder();
    cout << "\nSearching ID 15:\n";
    t.search(15);
    cout << "\nMinimum Employee ID:\n";
    t.findMin();
    cout << "\nMaximum Employee ID:\n";
    t.findMax();
    cout << "\nDeleting ID 30...\n";
    t.deleteNode(30);
    cout << "\nAfter Deletion:\n";
    t.inOrder();
    return 0;
}
```

Output:

```
Inorder Traversal (Ascending by ID):
ID: 10 | Name: Bilal | Department: HR
ID: 15 | Name: Ahmed | Department: IT
ID: 30 | Name: Ali | Department: Sales
ID: 40 | Name: Ayaan | Department: Finance
ID: 50 | Name: Sara | Department: Admin

Searching ID 15:
Found ID: 15 | Name: Ahmed | Department: IT

Minimum Employee ID:
Min ID: 10 | Name: Bilal | Department: HR

Maximum Employee ID:
Max ID: 50 | Name: Sara | Department: Admin

Deleting ID 30...

After Deletion:
ID: 10 | Name: Bilal | Department: HR
ID: 15 | Name: Ahmed | Department: IT
ID: 40 | Name: Ayaan | Department: Finance
ID: 50 | Name: Sara | Department: Admin


=== Code Execution Successful ===
```

Q3:

3. Given the two nodes of a binary search tree, return their least common ancestor.

Code:

```
#include <iostream>
using namespace std;

/*
 Syed Wahaaj Ali
 CT-24035
*/

class TreeNode {
	public:
		int val;
		TreeNode* left;
		TreeNode* right;
		TreeNode(int v) {
			val = v;
			left = nullptr;
			right = nullptr;
		}
};

class BST {
	public:
		TreeNode* root;

		BST() {
			root = nullptr;
		}

		void insert(int key) {
			root = insertRec(root, key);
		}

		TreeNode* leastCommonAncestor(int n1, int n2) {
			return leastCommonAncestorRec(root, n1, n2);
		}

	private:
		TreeNode* insertRec(TreeNode* node, int key) {
```

```
                if (node == nullptr)
                        return new TreeNode(key);
                if (key < node->val)
                        node->left = insertRec(node->left, key);
                else if (key > node->val)
                        node->right = insertRec(node->right, key);
                return node;
        }

        TreeNode* leastCommonAncestorRec(TreeNode* node, int n1, int n2) {
                if (node == nullptr) return nullptr;

                if (node->val > n1 && node->val > n2)
                        return leastCommonAncestorRec(node->left, n1, n2);

                if (node->val < n1 && node->val < n2)
                        return leastCommonAncestorRec(node->right, n1, n2);

                return node;
        }
};

int main() {
        BST b;
        b.insert(20);
        b.insert(10);
        b.insert(30);
        b.insert(5);
        b.insert(15);
        b.insert(25);
        b.insert(35);

        int n1 = 5, n2 = 15;
        TreeNode* lca = b.leastCommonAncestor(n1, n2);
        if (lca != nullptr) {
                cout << "LCA of " << n1 << " and " << n2 << " is: " << lca->val << endl;
        } else {
                cout << "LCA not found." << endl;
        }
        return 0;
}
```

Output:

```
LCA of 5 and 15 is: 10


=== Code Execution Successful ===
```

Q4:

4. Given the root of a binary search tree, recursively find the sum of all nodes of the tree.

Code:

```
#include <iostream>
using namespace std;

/*
 Syed Wahaaj Ali
 CT-24035
*/

class TreeNode {
	public:
		int val;
		TreeNode* left;
		TreeNode* right;

		TreeNode(int v) {
			val = v;
			left = nullptr;
			right = nullptr;
		}
};

class BinarySearchTree {
	public:
		TreeNode* root;

		BinarySearchTree() {
			root = nullptr;
		}

		void insert(int key) {
			root = insertRec(root, key);
		}
```

```
    int sumOfNodes() {
        return sumOfNodesRec(root);
    }

private:
    TreeNode* insertRec(TreeNode* node, int key) {
        if (node == nullptr)
            return new TreeNode(key);
        if (key < node->val)
            node->left = insertRec(node->left, key);
        else if (key > node->val)
            node->right = insertRec(node->right, key);
        return node;
    }

    int sumOfNodesRec(TreeNode* node) {
        if (node == nullptr) return 0;
        int leftSum = sumOfNodesRec(node->left);
        int rightSum = sumOfNodesRec(node->right);
        return leftSum + rightSum + node->val;
    }
};

int main() {
    BinarySearchTree bst;
    bst.insert(10);
    bst.insert(30);
    bst.insert(20);
    bst.insert(5);
    bst.insert(15);
    bst.insert(35);
    bst.insert(25);

    cout << "Sum of all Nodes: " << bst.sumOfNodes() << endl;
    return 0;
}
```

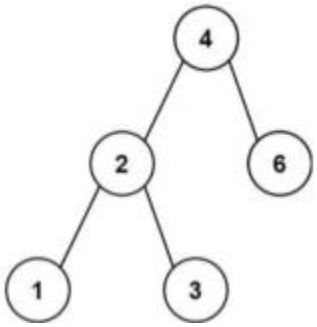
Output:

```
Sum of all Nodes: 140


=== Code Execution Successful ===
```

Q5:

5. Given the root of a Binary Search Tree (BST), return the minimum difference between the values of any two different nodes in the tree.



Code:

```
#include <iostream>
#include <climits>
using namespace std;

/*
 Syed Wahaaj Ali
 CT-24035
*/

class TreeNode {
	public:
		int val;
		TreeNode* left;
		TreeNode* right;
		TreeNode(int v) {
			val = v;
			left = nullptr;
			right = nullptr;
		}
};

class BST {
	public:
		TreeNode* root;

		BST() {
			root = nullptr;
		}

		void insert(int key) {
			root = insertRec(root, key);
		}
```

```
        int getMinDiff() {
            prev = -1;
            minDiff = INT_MAX;
            inOrderMinDiff(root);
            return minDiff;
        }

    private:
        TreeNode* insertRec(TreeNode* node, int key) {
            if (node == nullptr)
                return new TreeNode(key);
            if (key < node->val)
                node->left = insertRec(node->left, key);
            else if (key > node->val)
                node->right = insertRec(node->right, key);
            return node;
        }

        int prev;
        int minDiff;

        void inOrderMinDiff(TreeNode* node) {
            if (node == nullptr) return;

            inOrderMinDiff(node->left);

            if (prev != -1) {
                int diff = node->val - prev;
                if (diff < minDiff) minDiff = diff;
            }
            prev = node->val;

            inOrderMinDiff(node->right);
        }
};

int main() {
    BST b;
    b.insert(4);
    b.insert(2);
    b.insert(6);
    b.insert(1);
    b.insert(3);
    cout << "Minimum difference between any two Nodes: " << b.getMinDiff() << endl;
    return 0;
}
```

Output:

```
Minimum difference between any two Nodes: 1


=== Code Execution Successful ===
```