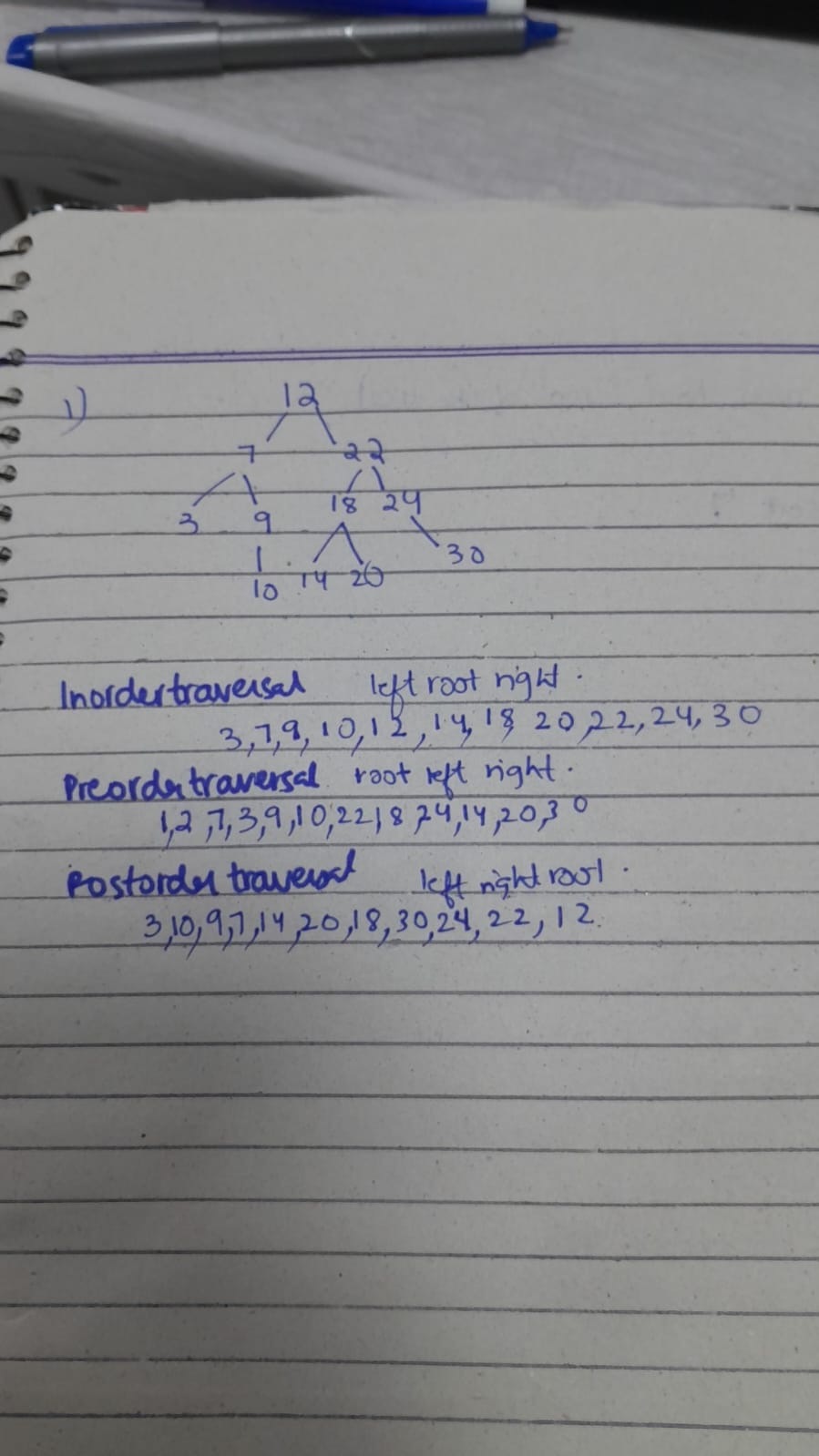
**Fatima nazir-45317**

**LAB TASK 1**

****

**Task 2**

**1**

#include <iostream>

#include <string>

using namespace std;

class Employee {

public:

int employeeNumber;

string name;

double salary;

Employee(int empNo, string empName, double empSalary) {

employeeNumber = empNo;

name = empName;

salary = empSalary;

}

};

class Node {

public:

Employee\* employee;

Node\* left;

Node\* right;

Node(Employee\* emp) {

employee = emp;

left = nullptr;

right = nullptr;

}

};

class EmployeeBST {

public:

Node\* root;

EmployeeBST() {

root = nullptr;

}

void insert(int empNo, string empName, double empSalary) {

Employee\* newEmployee = new Employee(empNo, empName, empSalary);

Node\* newNode = new Node(newEmployee);

if (root == nullptr) {

root = newNode;

return;

}

Node\* current = root;

Node\* parent = nullptr;

while (current != nullptr) {

parent = current;

if (empNo < current->employee->employeeNumber) {

current = current->left;

} else if (empNo > current->employee->employeeNumber) {

current = current->right;

} else {

return;

}

}

if (empNo < parent->employee->employeeNumber) {

parent->left = newNode;

} else {

parent->right = newNode;

}

}

Employee\* search(int employeeNumber) {

Node\* current = root;

while (current != nullptr) {

if (current->employee->employeeNumber == employeeNumber) {

return current->employee;

} else if (employeeNumber < current->employee->employeeNumber) {

current = current->left;

} else {

current = current->right;

}

}

return nullptr;

}

void display() {

inOrderTraversal(root);

}

private:

void inOrderTraversal(Node\* node) {

if (node == nullptr) {

return;

}

inOrderTraversal(node->left);

cout << "Employee Number: " << node->employee->employeeNumber

<< ", Name: " << node->employee->name

<< ", Salary: $" << node->employee->salary << endl;

inOrderTraversal(node->right);

}

};

int main() {

EmployeeBST tree;

tree.insert(102, "John Doe", 50000.0);

tree.insert(101, "Alice Smith", 55000.0);

tree.insert(104, "Bob Brown", 48000.0);

tree.insert(103, "Charlie White", 60000.0);

cout << "Employee Records (In-order Traversal):" << endl;

tree.display();

int searchEmpNo = 103;

Employee\* emp = tree.search(searchEmpNo);

if (emp != nullptr) {

cout << "\nEmployee Found: " << endl;

cout << "Employee Number: " << emp->employeeNumber

<< ", Name: " << emp->name

<< ", Salary: $" << emp->salary << endl;

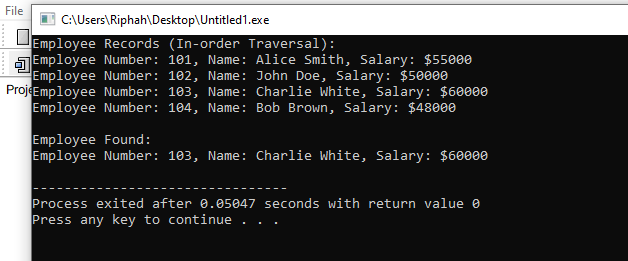
} else {

cout << "Employee with Employee Number " << searchEmpNo << " not found." << endl;

}

return 0;

}



**2**

#include <iostream>

#include <string>

using namespace std;

// Employee class to store employee details

class Employee {

public:

int employeeNumber;

string name;

double salary;

// Constructor to initialize employee details

Employee(int empNo, string empName, double empSalary) {

employeeNumber = empNo;

name = empName;

salary = empSalary;

}

};

// Node class to represent each node in the Binary Search Tree (BST)

class Node {

public:

Employee\* employee;

Node\* left;

Node\* right;

// Constructor to create a new node

Node(Employee\* emp) {

employee = emp;

left = nullptr;

right = nullptr;

}

};

// Binary Search Tree (BST) class

class EmployeeBST {

public:

Node\* root;

// Constructor to initialize the root of the tree

EmployeeBST() {

root = nullptr;

}

// Method to insert a new employee into the BST

void insert(int empNo, string empName, double empSalary) {

Employee\* newEmployee = new Employee(empNo, empName, empSalary);

Node\* newNode = new Node(newEmployee);

if (root == nullptr) {

// If the tree is empty, the new node becomes the root

root = newNode;

return;

}

Node\* current = root;

Node\* parent = nullptr;

// Traverse the tree to find the correct position for the new node

while (current != nullptr) {

parent = current;

if (empNo < current->employee->employeeNumber) {

current = current->left;

} else if (empNo > current->employee->employeeNumber) {

current = current->right;

} else {

// If employee number already exists, do nothing

return;

}

}

// Insert the new node at the appropriate position

if (empNo < parent->employee->employeeNumber) {

parent->left = newNode;

} else {

parent->right = newNode;

}

}

// Method to search for an employee by employee number

Employee\* search(int employeeNumber) {

Node\* current = root;

// Traverse the tree to search for the employee

while (current != nullptr) {

if (current->employee->employeeNumber == employeeNumber) {

return current->employee; // Employee found

} else if (employeeNumber < current->employee->employeeNumber) {

current = current->left; // Go left if the number is smaller

} else {

current = current->right; // Go right if the number is larger

}

}

return nullptr; // Employee not found

}

// Method to display all employees in the BST (In-order traversal)

void display() {

inOrderTraversal(root);

}

private:

// Helper function to perform an in-order traversal

void inOrderTraversal(Node\* node) {

if (node == nullptr) {

return;

}

// Traverse the left subtree

inOrderTraversal(node->left);

// Display the employee details

cout << "Employee Number: " << node->employee->employeeNumber

<< ", Name: " << node->employee->name

<< ", Salary: $" << node->employee->salary << endl;

// Traverse the right subtree

inOrderTraversal(node->right);

}

};

// Main function to test the Binary Search Tree

int main() {

EmployeeBST tree;

// Inserting employees into the tree

tree.insert(102, "John Doe", 50000.0);

tree.insert(101, "Alice Smith", 55000.0);

tree.insert(104, "Bob Brown", 48000.0);

tree.insert(103, "Charlie White", 60000.0);

// Display all employees in sorted order

cout << "Employee Records (In-order Traversal):" << endl;

tree.display();

// Searching for an employee by employee number

int searchEmpNo = 103;

Employee\* emp = tree.search(searchEmpNo);

if (emp != nullptr) {

cout << "\nEmployee Found: " << endl;

cout << "Employee Number: " << emp->employeeNumber

<< ", Name: " << emp->name

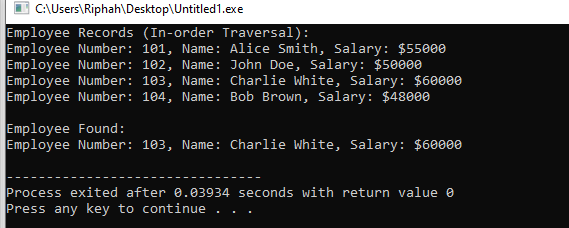
<< ", Salary: $" << emp->salary << endl;

} else {

cout << "Employee with Employee Number " << searchEmpNo << " not found." << endl;

}

return 0;

} 

**3**

#include <iostream>

#include <string>

using namespace std;

// Employee class to store employee details

class Employee {

public:

int employeeNumber;

string name;

double salary;

// Constructor to initialize employee details

Employee(int empNo, string empName, double empSalary) {

employeeNumber = empNo;

name = empName;

salary = empSalary;

}

};

// Node class to represent each node in the Binary Search Tree (BST)

class Node {

public:

Employee\* employee;

Node\* left;

Node\* right;

// Constructor to create a new node

Node(Employee\* emp) {

employee = emp;

left = nullptr;

right = nullptr;

}

};

// Binary Search Tree (BST) class

class EmployeeBST {

public:

Node\* root;

// Constructor to initialize the root of the tree

EmployeeBST() {

root = nullptr;

}

// Method to insert a new employee into the BST

void insert(int empNo, string empName, double empSalary) {

Employee\* newEmployee = new Employee(empNo, empName, empSalary);

Node\* newNode = new Node(newEmployee);

if (root == nullptr) {

root = newNode;

return;

}

Node\* current = root;

Node\* parent = nullptr;

while (current != nullptr) {

parent = current;

if (empNo < current->employee->employeeNumber) {

current = current->left;

} else if (empNo > current->employee->employeeNumber) {

current = current->right;

} else {

return; // Employee number already exists

}

}

if (empNo < parent->employee->employeeNumber) {

parent->left = newNode;

} else {

parent->right = newNode;

}

}

// Method to search for an employee by employee number

Employee\* search(int employeeNumber) {

Node\* current = root;

while (current != nullptr) {

if (current->employee->employeeNumber == employeeNumber) {

return current->employee;

} else if (employeeNumber < current->employee->employeeNumber) {

current = current->left;

} else {

current = current->right;

}

}

return nullptr; // Employee not found

}

// Method to display all employees in the BST (In-order traversal)

void display() {

inOrderTraversal(root);

}

// Method to delete an employee from the BST

void deleteEmployee(int employeeNumber) {

root = deleteNode(root, employeeNumber);

}

private:

// Helper function to perform an in-order traversal

void inOrderTraversal(Node\* node) {

if (node == nullptr) return;

inOrderTraversal(node->left);

cout << "Employee Number: " << node->employee->employeeNumber

<< ", Name: " << node->employee->name

<< ", Salary: $" << node->employee->salary << endl;

inOrderTraversal(node->right);

}

// Helper function to delete a node and return the new root of the subtree

Node\* deleteNode(Node\* root, int employeeNumber) {

if (root == nullptr) {

return root; // If the tree is empty or we've reached a leaf node

}

// Traverse the tree

if (employeeNumber < root->employee->employeeNumber) {

root->left = deleteNode(root->left, employeeNumber); // Go left

} else if (employeeNumber > root->employee->employeeNumber) {

root->right = deleteNode(root->right, employeeNumber); // Go right

} else {

// Node to be deleted is found

// Case 1: Node has no children (leaf node)

if (root->left == nullptr && root->right == nullptr) {

delete root;

return nullptr;

}

// Case 2: Node has one child

else if (root->left == nullptr) {

Node\* temp = root->right;

delete root;

return temp;

} else if (root->right == nullptr) {

Node\* temp = root->left;

delete root;

return temp;

}

// Case 3: Node has two children

else {

// Find the in-order successor (smallest node in the right subtree)

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

// Replace root's employee with the in-order successor

root->employee = temp->employee;

// Delete the in-order successor

root->right = deleteNode(root->right, temp->employee->employeeNumber);

}

}

return root;

}

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

Node\* findMin(Node\* node) {

Node\* current = node;

while (current && current->left != nullptr) {

current = current->left;

}

return current;

}

};

// Main function to test the Binary Search Tree

int main() {

EmployeeBST tree;

// Inserting employees into the tree

tree.insert(102, "John Doe", 50000.0);

tree.insert(101, "Alice Smith", 55000.0);

tree.insert(104, "Bob Brown", 48000.0);

tree.insert(103, "Charlie White", 60000.0);

// Display all employees in sorted order

cout << "Employee Records (In-order Traversal):" << endl;

tree.display();

// Delete an employee by employee number

int empToDelete = 103;

cout << "\nDeleting employee with Employee Number: " << empToDelete << endl;

tree.deleteEmployee(empToDelete);

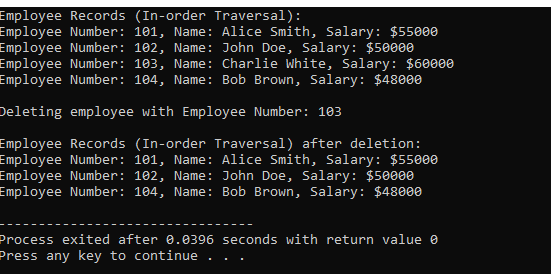
// Display the updated tree after deletion

cout << "\nEmployee Records (In-order Traversal) after deletion:" << endl;

tree.display();

return 0;

}



**4**

#include <iostream>

#include <string>

using namespace std;

// Employee class to store employee details

class Employee {

public:

int employeeNumber;

string name;

double salary;

// Constructor to initialize employee details

Employee(int empNo, string empName, double empSalary) {

employeeNumber = empNo;

name = empName;

salary = empSalary;

}

};

// Node class to represent each node in the Binary Search Tree (BST)

class Node {

public:

Employee\* employee;

Node\* left;

Node\* right;

// Constructor to create a new node

Node(Employee\* emp) {

employee = emp;

left = nullptr;

right = nullptr;

}

};

// Binary Search Tree (BST) class

class EmployeeBST {

public:

Node\* root;

// Constructor to initialize the root of the tree

EmployeeBST() {

root = nullptr;

}

// Method to insert a new employee into the BST

void insert(int empNo, string empName, double empSalary) {

Employee\* newEmployee = new Employee(empNo, empName, empSalary);

Node\* newNode = new Node(newEmployee);

if (root == nullptr) {

root = newNode;

return;

}

Node\* current = root;

Node\* parent = nullptr;

while (current != nullptr) {

parent = current;

if (empNo < current->employee->employeeNumber) {

current = current->left;

} else if (empNo > current->employee->employeeNumber) {

current = current->right;

} else {

return; // Employee number already exists

}

}

if (empNo < parent->employee->employeeNumber) {

parent->left = newNode;

} else {

parent->right = newNode;

}

}

// Method to search for an employee by employee number

Employee\* search(int employeeNumber) {

Node\* current = root;

while (current != nullptr) {

if (current->employee->employeeNumber == employeeNumber) {

return current->employee;

} else if (employeeNumber < current->employee->employeeNumber) {

current = current->left;

} else {

current = current->right;

}

}

return nullptr; // Employee not found

}

// Method to delete an employee from the BST

void deleteEmployee(int employeeNumber) {

root = deleteNode(root, employeeNumber);

}

// Method to display all employees in the BST (In-order traversal)

void displayInOrder() {

cout << "In-order Traversal:" << endl;

inOrderTraversal(root);

cout << endl;

}

// Method to display all employees in the BST (Pre-order traversal)

void displayPreOrder() {

cout << "Pre-order Traversal:" << endl;

preOrderTraversal(root);

cout << endl;

}

// Method to display all employees in the BST (Post-order traversal)

void displayPostOrder() {

cout << "Post-order Traversal:" << endl;

postOrderTraversal(root);

cout << endl;

}

private:

// Helper function to perform an in-order traversal

void inOrderTraversal(Node\* node) {

if (node == nullptr) return;

inOrderTraversal(node->left);

cout << "Employee Number: " << node->employee->employeeNumber

<< ", Name: " << node->employee->name

<< ", Salary: $" << node->employee->salary << endl;

inOrderTraversal(node->right);

}

// Helper function to perform a pre-order traversal

void preOrderTraversal(Node\* node) {

if (node == nullptr) return;

cout << "Employee Number: " << node->employee->employeeNumber

<< ", Name: " << node->employee->name

<< ", Salary: $" << node->employee->salary << endl;

preOrderTraversal(node->left);

preOrderTraversal(node->right);

}

// Helper function to perform a post-order traversal

void postOrderTraversal(Node\* node) {

if (node == nullptr) return;

postOrderTraversal(node->left);

postOrderTraversal(node->right);

cout << "Employee Number: " << node->employee->employeeNumber

<< ", Name: " << node->employee->name

<< ", Salary: $" << node->employee->salary << endl;

}

// Helper function to delete a node and return the new root of the subtree

Node\* deleteNode(Node\* root, int employeeNumber) {

if (root == nullptr) {

return root;

}

if (employeeNumber < root->employee->employeeNumber) {

root->left = deleteNode(root->left, employeeNumber);

} else if (employeeNumber > root->employee->employeeNumber) {

root->right = deleteNode(root->right, employeeNumber);

} else {

if (root->left == nullptr && root->right == nullptr) {

delete root;

return nullptr;

} else if (root->left == nullptr) {

Node\* temp = root->right;

delete root;

return temp;

} else if (root->right == nullptr) {

Node\* temp = root->left;

delete root;

return temp;

} else {

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

root->employee = temp->employee;

root->right = deleteNode(root->right, temp->employee->employeeNumber);

}

}

return root;

}

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

Node\* findMin(Node\* node) {

Node\* current = node;

while (current && current->left != nullptr) {

current = current->left;

}

return current;

}

};

// Main function to test the Binary Search Tree

int main() {

EmployeeBST tree;

// Inserting employees into the tree

tree.insert(102, "John Doe", 50000.0);

tree.insert(101, "Alice Smith", 55000.0);

tree.insert(104, "Bob Brown", 48000.0);

tree.insert(103, "Charlie White", 60000.0);

// Display tree traversals

tree.displayInOrder(); // In-order Traversal

tree.displayPreOrder(); // Pre-order Traversal

tree.displayPostOrder(); // Post-order Traversal

// Delete an employee and display the tree again

int empToDelete = 103;

cout << "\nDeleting employee with Employee Number: " << empToDelete << endl;

tree.deleteEmployee(empToDelete);

// Display the updated tree after deletion

tree.displayInOrder(); // In-order Traversal after deletion

tree.displayPreOrder(); // Pre-order Traversal after deletion

tree.displayPostOrder(); // Post-order Traversal after deletion

return 0;

}

