Bahria University, Islamabad

Department of Software Engineering

Data Structre And Algorithms

(Fall-2024)

Teacher: Engr. Aleem Ahmad

Student : Lotfullah Muslimwal

Enrollment : 01-131232-039

Lab Journal: X

Date:

Task No:	Task Wise Marks		Documentation Marks		Total Marks (20)
	Assigned	Obtained	Assigned	Obtained	
1	3		5		
2	3				
3	3				
4	3				
5	3				

Comments:

Signature

LAB

Lab No: Open Ended

Task 1: Open Ended

Code:

```
#include <iostream>

using namespace std;
struct TreeNode {
   int data;
   TreeNode* left;
   TreeNode* right;
};

class CustomBinaryTree {
private:
   TreeNode* root;

   TreeNode* createNode(int value) {
      TreeNode* newNode = new TreeNode();
      newNode->data = value;
      newNode->left = newNode->right = nullptr;
      return newNode;
   }

   TreeNode* insertNode(TreeNode* node, int value) {
      if (!node) return createNode(value);
      if (value < node->data) node->left = insertNode(node->left, value);
      else if (value > node->data) node->right = insertNode(node->right, value);
      return node;
   }

   TreeNode* deleteNode(TreeNode* root, int value) {
      if (!root) return root;
      if (value < root->data) root->left = deleteNode(root->left, value);
      else if (value > root->data) root->right = deleteNode(root->right, value);
      else {
         if (!root->left) {
            TreeNode* temp = root->right;
            delete root;
            return temp;
         }
```

```
            else if (!root->right) {
                TreeNode* temp = root->left;
                delete root;
                return temp;
            }
            TreeNode* temp = findMinNode(root->right);
            root->data = temp->data;
            root->right = deleteNode(root->right, temp->data);
        }
        return root;
    }

    TreeNode* findMinNode(TreeNode* node) {
        while (node && node->left) node = node->left;
        return node;
    }

    void inorderTraversal(TreeNode* node) {
        if (!node) return;
        inorderTraversal(node->left);
        cout << node->data << " ";
        inorderTraversal(node->right);
    }

    void preorderTraversal(TreeNode* node) {
        if (!node) return;
        cout << node->data << " ";
        preorderTraversal(node->left);
        preorderTraversal(node->right);
    }

    void postorderTraversal(TreeNode* node) {
        if (!node) return;
        postorderTraversal(node->left);
        postorderTraversal(node->right);
        cout << node->data << " ";
    }

public:
    CustomBinaryTree() : root(nullptr) {}

    void insert(int value) {
        root = insertNode(root, value);
        cout << "Value " << value << " inserted.\n";
```

```
    }

    void remove(int value) {
        root = deleteNode(root, value);
        cout << "Value " << value << " deleted.\n";
    }

    void displayInOrder() {
        inorderTraversal(root);
        cout << endl;
    }

    void displayPreOrder() {
        preorderTraversal(root);
        cout << endl;
    }

    void displayPostOrder() {
        postorderTraversal(root);
        cout << endl;
    }
};

class WeightedGraph {
private:
    int vertices;
    int** adjMatrix;

public:
    WeightedGraph(int v) : vertices(v) {
        adjMatrix = new int* [vertices];
        for (int i = 0; i < vertices; i++) {
            adjMatrix[i] = new int[vertices];
            for (int j = 0; j < vertices; j++) {
                adjMatrix[i][j] = (i == j) ? 0 : INT_MAX;
            }
        }
    }

    ~WeightedGraph() {
        for (int i = 0; i < vertices; i++) {
            delete[] adjMatrix[i];
        }
        delete[] adjMatrix;
```

```
    }

    void addConnection(int src, int dest, int weight) {
        adjMatrix[src][dest] = weight;
        cout << "Connection from " << src << " to " << dest << " with weight " << weight << "
added.\n";
    }

    void findShortestPath(int src, int dest) {
        bool* visited = new bool[vertices];
        int* dist = new int[vertices];
        for (int i = 0; i < vertices; i++) {
            dist[i] = INT_MAX;
            visited[i] = false;
        }
        dist[src] = 0;

        for (int i = 0; i < vertices - 1; i++) {
            int u = getMinDistance(dist, visited);
            visited[u] = true;
            for (int v = 0; v < vertices; v++) {
                if (!visited[v] && adjMatrix[u][v] != INT_MAX &&
                    dist[u] != INT_MAX && dist[u] + adjMatrix[u][v] < dist[v]) {
                    dist[v] = dist[u] + adjMatrix[u][v];
                }
            }
        }

        if (dist[dest] == INT_MAX) {
            cout << "No path from " << src << " to " << dest << endl;
        }
        else {
            cout << "Shortest path from " << src << " to " << dest << " is " << dist[dest] << endl;
        }
        delete[] visited;
        delete[] dist;
    }

private:
    int getMinDistance(int* dist, bool* visited) {
        int min = INT_MAX, minIndex = -1;
        for (int v = 0; v < vertices; v++) {
            if (!visited[v] && dist[v] <= min) {
                min = dist[v];
```

```
            minIndex = v;
          }
        }
        return minIndex;
    }
};

void mergeArrays(int arr[], int left, int mid, int right, int& comparisons, int& swaps) {
    int n1 = mid - left + 1;
    int n2 = right - mid;
    int* L = new int[n1];
    int* R = new int[n2];

    for (int i = 0; i < n1; i++) L[i] = arr[left + i];
    for (int j = 0; j < n2; j++) R[j] = arr[mid + 1 + j];

    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        comparisons++;
        if (L[i] <= R[j]) {
            arr[k++] = L[i++];
        }
        else {
            arr[k++] = R[j++];
            swaps++;
        }
    }

    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];

    delete[] L;
    delete[] R;
}

void mergeSortArrays(int arr[], int left, int right, int& comparisons, int& swaps) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSortArrays(arr, left, mid, comparisons, swaps);
        mergeSortArrays(arr, mid + 1, right, comparisons, swaps);
        mergeArrays(arr, left, mid, right, comparisons, swaps);
    }
}
```

```
int partitionArray(int arr[], int low, int high, int& comparisons, int& swaps) {
    int pivot = arr[high];
    int i = low - 1;
    for (int j = low; j <= high - 1; j++) {
        comparisons++;
        if (arr[j] < pivot) {
            swaps++;
            swap(arr[++i], arr[j]);
        }
    }
    swaps++;
    swap(arr[i + 1], arr[high]);
    return i + 1;
}

void quickSortArrays(int arr[], int low, int high, int& comparisons, int& swaps) {
    if (low < high) {
        int pi = partitionArray(arr, low, high, comparisons, swaps);
        quickSortArrays(arr, low, pi - 1, comparisons, swaps);
        quickSortArrays(arr, pi + 1, high, comparisons, swaps);
    }
}

int getValidInteger() {
    int value;
    while (!(cin >> value)) {
        cin.clear();
        cin.ignore(numeric_limits<streamsize>::max(), '\n');
        cout << "Invalid input. Please enter an integer: ";
    }
    return value;
}

int main() {
    CustomBinaryTree tree;
    WeightedGraph* graph = nullptr;
    int userChoice;

    while (true) {
        cout << "\nChoose an option:\n";
        cout << "1. Binary Tree Operations\n";
        cout << "2. Graph Shortest Path Calculation\n";
        cout << "3. Array Sorting Algorithms\n";
        cout << "4. Exit Program\n";
```

```
        cout << "Enter your choice: ";
        userChoice = getValidInteger();

        switch (userChoice) {
        case 1: {
            int treeChoice, value;
            cout << "1. Insert Value\n2. Delete Value\n3. Inorder Traversal\n4. Preorder
Traversal\n5. Postorder Traversal\nChoose an operation: ";
            treeChoice = getValidInteger();
            switch (treeChoice) {
            case 1:
                cout << "Enter value to insert: ";
                value = getValidInteger();
                tree.insert(value);
                break;
            case 2:
                cout << "Enter value to delete: ";
                value = getValidInteger();
                tree.remove(value);
                break;
            case 3:
                tree.displayInOrder();
                break;
            case 4:
                tree.displayPreOrder();
                break;
            case 5:
                tree.displayPostOrder();
                break;
            default:
                cout << "Invalid choice.\n";
            }
            break;
        }
        case 2: {
            int vertices, src, dest, weight;
            if (!graph) {
                cout << "Enter number of vertices: ";
                vertices = getValidInteger();
                graph = new WeightedGraph(vertices);
            }
            cout << "1. Add Connection\n2. Find Shortest Path\nEnter choice: ";
            int graphChoice = getValidInteger();
            switch (graphChoice) {
```

```
            case 1:
                cout << "Enter source, destination, and weight: ";
                src = getValidInteger();
                dest = getValidInteger();
                weight = getValidInteger();
                graph->addConnection(src, dest, weight);
                break;
            case 2:
                cout << "Enter source and destination: ";
                src = getValidInteger();
                dest = getValidInteger();
                graph->findShortestPath(src, dest);
                break;
            default:
                cout << "Invalid choice.\n";
            }
            break;
        }
        case 3: {
            int n;
            cout << "Enter number of elements in the array: ";
            n = getValidInteger();
            int* arr1 = new int[n];
            int* arr2 = new int[n];
            cout << "Enter array elements: ";
            for (int i = 0; i < n; i++) {
                cin >> arr1[i];
                arr2[i] = arr1[i];
            }
            int comparisons = 0, swaps = 0;
            mergeSortArrays(arr1, 0, n - 1, comparisons, swaps);
            cout << "Merge Sort: Comparisons = " << comparisons << ", Swaps = " << swaps <<
"\n";
            comparisons = swaps = 0;
            quickSortArrays(arr2, 0, n - 1, comparisons, swaps);
            cout << "Quick Sort: Comparisons = " << comparisons << ", Swaps = " << swaps <<
"\n";
            delete[] arr1;
            delete[] arr2;
            break;
        }
        case 4:
            cout << "Exiting program.\n";
            delete graph;
```

```
            return 0;
        default:
            cout << "Invalid choice.\n";
        }
    }
}
```

GitHub-Link: https://github.com/lotfullahmsl/DSA-Lab-FA2024

Screenshot: