DSAL Practical's

Consider the telephone book database of N clients. Make use of a hash table implementation to quickly look up a client's telephone number. Make use of two collision handling techniques and compare them using number of comparisons required to find a set of telephone numbers

```
def run():
    htable_linear = [None] * 10
    htable_quadratic = [None] * 10
    n = int(input("Enter number of Phone Numbers: "))
    for _ in range(n):
        key = int(input("Enter Mobile Number: "))
        position = key % 10
        if htable_linear[position] is None:
            htable_linear[position] = [key]
        else:
            htable_linear = linear_probe(key, htable_linear, position)
            print("Linear Probing for empty slot..")
        if htable_quadratic[position] is None:
            htable_quadratic[position] = [key]
        else:
            htable_quadratic = quadric_probe(key, htable_quadratic, position)
            print("Quadratic Probing for empty slot..")

        display(htable_linear, "Linear Probing")
        display(htable_quadratic, "Quadratic Probing")
    return htable_linear, htable_quadratic

def linear_probe(key, htable, pos):
    for i in range(1, 10):
        proble_position = (pos + i) % 10
        if htable[proble_position] is None:
            htable[proble_position] = [key]
            return htable
    print("Hash table is full")
    return htable

def quadric_probe(key, htable, pos):
    for i in range(1, 10):
        probe_position = (pos + i * i) % 10
        if htable[probe_position] is None:
            htable[probe_position] = [key]
            return htable
    print("Hash table is full")
    return htable
```

```
def display(htable, method):
    print(f"\n{method} Hash Table:")
    for pos in range(10):
        val = htable[pos]
        print(f"Position {pos}: {val}")

def search(key, ht, method):
    comparisons = 0
    pos = key % 10
    if ht[pos] is not None and ht[pos] == key:
        comparisons += 1
        return pos, comparisons
    for i in range(1, 10):
        comparisons += 1
        if method == "linear":
            probe_position = (pos + i) % 10
        else:
            if method == "quadratic":
                probe_position = (pos + i * i) % 10
        if ht[probe_position] is not None and ht[probe_position] == key:
            return probe_position, comparisons
    return -1, comparisons

def main():
    ht_linear, ht_quadratic = None, None
    while True:
        print("\nMenu:")
        print("1. Insert Phone Number")
        print("2. Search Phone Number")
        print("3. Exit")
        choice = int(input("Enter your choice: "))

        if choice == 1:
            if ht_linear is None and ht_quadratic is None:
                ht_linear, ht_quadratic = run()
            else:
                print("Phone numbers already inserted.")
        elif choice == 2:
            if ht_linear is None and ht_quadratic is None:
                print("No phone numbers to search.")
            else:
                key = int(input("Enter Mobile Number to search: "))
                pos, comparisons = search(key, ht_linear, "linear")
                if pos != -1:
```

```
                    print(f"Phone number {key} found at position {pos} in Linear
Probing with {comparisons} comparisons.")
                else:
                    print(f"Phone number {key} not found in Linear Probing with
{comparisons} comparisons.")

                pos, comparisons = search(key, ht_quadratic, "quadratic")
                if pos != -1:
                    print(f"Phone number {key} found at position {pos} in
Quadratic Probing with {comparisons} comparisons.")
                else:
                    print(f"Phone number {key} not found in Quadratic Probing
with {comparisons} comparisons.")
        elif choice == 3:
            print("Exiting...")
            break
        else:
            print("Invalid choice. Please try again.")

if __name__ == "__main__":
    main()
```

To create ADT that implements the “Set” concept. Add, Remove, Union, Intersection, Difference, check membership

```
class setBuilder:
    def __init__(self):
         self.set = set()

    def addElements(self):
        n = int(input("No. of elements to insert: "))
        for i in range(n):
            element = input(f"Enter element {i}: ")
            if element not in self.set:
                self.set.add(element)

    def removeElement(self,element):
         self.set.remove(element)
```

```
    def setUnion(self, s2):
        unionSet = set()
        unionSet = self.set.copy()
        for element in s2.set:
            if element not in self.set:
                unionSet.add(element)
        return unionSet

    def setIntersection(self, set2):
        intersectionSet = set()
        for element in self.set:
            if element in set2.set:
                intersectionSet.add(element)
        return intersectionSet

    def setDifference(self, set2):
        differenceSet = set()
        for element in self.set:
            if element not in set2.set:
                differenceSet.add(element)
        return differenceSet

    def checkMembership(self, ch):
        element = input("Enter element to check: ")
        if element in self.set:
            print(f"{element} is present in Set{ch}")
        else:
            print(f"{element} is not present in Set{ch}")

def main():
     set1 = setBuilder()
     set2 = setBuilder()
     while True:
        print("**SET OPERATIONS**")
        print("1. Insert elements")
        print("2. Remove element")
        print("3. Display both sets")
        print("4. Find union of set 1 and set 2")
        print("5. Set Intersection")
        print("6. Set Difference")
        print("7. Check membership")
        print("8. Exit")
        choice = input("Enter your choice: ")
        if choice == '1':
```

```
            ch = input("Insert elements in Set '1' or Set '2'?")
            if ch == '1':
                print("Enter data for Set1")
                set1.addElements()
            elif ch == '2':
                print("Enter data for Set2")
                set2.addElements()
        elif choice == '2':
            ch = input("Remove element from  Set '1' or Set '2'?")
            if ch == '1':
                ele = input("Enter element to remove: ")
                set1.removeElement(ele)
            elif ch == '2':
                ele = input("Enter element to remove: ")
                set2.removeElement(ele)
        elif choice == '3':
            print("Set 1: ", set1.set)
            print("Set 2: ", set2.set)
        elif choice == '4':
            print("Set 1: ", set1.set)
            print("Set 2: ", set2.set)
            unionSet = set1.setUnion(set2)
            print("Union of Set1 and Set2: " , unionSet)
        elif choice == '5':
            print("Set 1: ", set1.set)
            print("Set 2: ", set2.set)
            iSet = set1.setIntersection(set2)
            print("Intersection of Set1 and Set2: ", iSet)
        elif choice == '6':
            print("Set 1: ", set1.set)
            print("Set 2: ", set2.set)
            dSet = set1.setDifference(set2)
            print("Difference of Set1 and Set2: ", dSet)
        elif choice == '7':
            ch = input("Check membership in Set '1' or Set '2'?")
            if ch == '1':
                set1.checkMembership(ch)
            elif ch == '2':
                set2.checkMembership(ch)
        elif choice == '8':
            break
        else:
            print("Invalid choice. Please try again...")
```

```
if __name__ == "__main__":
    main()
```

Beginning with an empty binary search tree, Construct a binary search tree by inserting the values in the order given. After constructing a binary tree –

i. Insert new node,
ii. Find number of nodes in longest path from root,
iii. Minimum data value found in the tree,
iv. Change a tree so that the roles of the left and right pointers are swapped at every node,
v. Search a value

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

struct Node {
    int data;
    Node *left, *right;
    Node(int val)
    {
        data = val;
        left = right = nullptr;
    }
};

Node* insert(Node* root, int val)
{
    if(root == nullptr){
        cout <<endl<< "Root node created" <<endl;
        return new Node(val);
    }
    Node *current = root;
    while (true){
        if(val < current->data){
            if(current->left == nullptr){
                current->left = new Node(val);
                cout <<endl<< "Left Node Inserted" <<endl;
                break;
            } else {
                current = current -> left;
            }
        } else if (val > current->data){
```

```
            if(current->right == nullptr){
                current->right = new Node(val);
                cout <<endl<< "Right Node Inserted" <<endl;
                break;
            } else {
                current = current -> right;
            }
        } else {
            cout << endl << "Entry repeated" <<endl;
            break;
        }
    }
    return root;
}

int height(Node* node) {
    if (node == nullptr)
        return 0;
    int leftHeight = height(node->left);
    int rightHeight = height(node->right);
    return max(leftHeight, rightHeight) + 1;
}

int findMin(Node* node) {
    if (node == nullptr){
        cout <<endl<< "Tree is empty" <<endl;
        return -1;
    }
    while (node->left != nullptr) {
        node = node->left;
    }
    return node->data;
}

void mirror(Node* node) {
    if (node == nullptr)
        return;
    swap(node->left, node->right);
    mirror(node->left);
    mirror(node->right);
}

bool search(Node* root, int key) {
    Node* current = root;
    while  (current != nullptr) {
```

```
        if( key < current -> data) {
            current = current -> left;
        } else if (key > current -> data) {
            current = current -> right;
        } else {
            return true;
        }
    }
    return false;
}

void display(Node* temp) {
    if (temp != nullptr) {
        display(temp->left);
        cout << "\t" << temp->data << " ";
        display(temp->right);
    }
}

// // Level-order (layer-wise) display function
// void displayLevelOrder(Node* root) {
//     if (root == nullptr) {
//         cout << "\nTree is empty";
//         return;
//     }

//     // Using a queue to process nodes level by level
//     queue<Node*> q;
//     q.push(root);

//     while (!q.empty()) {
//         int levelSize = q.size(); // Number of nodes at current level
//         cout << "\nLevel: ";

//         // Process all nodes at current level
//         for (int i = 0; i < levelSize; i++) {
//             Node* current = q.front();
//             q.pop();

//             cout << current->data << " ";

//             // Enqueue children for next level
//             if (current->left != nullptr)
//                 q.push(current->left);
//             if (current->right != nullptr)
```

```
//                 q.push(current->right);
//         }
//     }
// }

int main() {
    Node* root = nullptr;
    int ch, val;
    while (true){
        cout << "\n====BST MENU====";
        cout << "\n1.Insert";
        cout << "\n2.Display";
        cout << "\n3.Search the node";
        cout << "\n4.Height of the tree";
        cout << "\n5.Minimum value in the tree";
        cout << "\n6.Mirror the tree";
        cout << "\n7.Exit";
        cout << "\nEnter your choice: ";
        cin >> ch;

        switch (ch) {
            case 1:
                cout << "Enter value to insert: ";
                cin >> val;
                root = insert(root, val);
                break;
            case 2:
                cout << "\nTREE:\n";
                if (root == nullptr)
                    cout << "\nTree is empty";
                else
                    display(root);
                break;
            case 3:
                cout << "Enter the value to search: ";
                cin >> val;
                if (!search(root, val)) {
                    cout << "\nValue is absent";
                } else {
                    cout << "\nValue is present";
                }
                break;
            case 4:
                cout << "\nHeight of the tree: " << height(root);
                break;
```

```
            case 5:
                val = findMin(root);
                if (val != -1)
                    cout << "\nMinimum value in the tree: " << val;
                break;
            case 6:
                mirror(root);
                cout << "\nTree mirrored successfully";
                break;
            case 7:
                cout << "\nExiting the program...";
                return 0;
            default:
                cout << "\nINVALID INPUT";
                break;
        }
    }
}
```

Construct an expression tree from the given prefix expression eg. +--a*bc/def and traverse it using postorder traversal (non recursive) and then delete the entire tree.

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

struct Node {
    char data;
    Node *left, *right;
    Node(char ch) {
        data = ch;
        left = right = nullptr;
    }
};

class ExpressionTree {
    private:
        Node* root;
    public:
        ExpressionTree(){
            root = nullptr;
        }
```

```
void constructTree(string prefix){
    stack<Node*> s;
    for (int i = prefix.length() - 1; i >= 0; i--) {
        if(isalnum(prefix[i]))
            s.push(new Node(prefix[i]));
        else {
            Node *newNode = new Node(prefix[i]);
            newNode->left = s.top(); s.pop();
            newNode->right = s.top(); s.pop();
            s.push(newNode);
        }
    }
    root = s.top();
}

void postOrder(){
    if (root == nullptr){
        cout << "Tree is empty." << endl;
        return;
    }
    stack<Node*> s;
    string postorder = "";
    s.push(root);
    while (!s.empty()){
        Node* temp = s.top();
        s.pop();
        postorder += temp->data;
        if (temp->left)
            s.push(temp->left);
        if (temp->right)
            s.push(temp->right);
    }
    reverse(postorder.begin(), postorder.end());
    cout << "Postorder Traversal: " << postorder << endl;
}

void deleteTree(){
    if (root == nullptr){
        cout << "Tree is already empty." << endl;
        return;
    }
    stack<Node*> s;
    s.push(root);
    while (!s.empty()){
```

```
                Node *temp = s.top();
                s.pop();
                if(temp->left)
                    s.push(temp->left);
                if(temp->right)
                    s.push(temp->right);
                delete temp;
            }
            cout << "Tree deleted successfully." << endl;
        }
};

//menu driven
int main() {
    ExpressionTree tree;
    string prefix;
    int choice;
    do {
        cout << endl <<" ===Expression Tree Menu===";
        cout << "\n1. Construct Expression Tree from Prefix Expression";
        cout << "\n2. Postorder Traversal of the Tree";
        cout << "\n3. Delete the Tree";
        cout << "\n4. Exit";
        cout << "\nEnter your choice: ";
        cin >> choice;
        switch (choice) {
            case 1:
                cout << "Enter prefix expression: ";
                cin >> prefix;
                tree.constructTree(prefix);
                break;
            case 2:
                tree.postOrder();
                break;
            case 3:
                tree.deleteTree();
                break;
            case 4:
                cout << "Exiting..." << endl;
                break;
            default:
                cout << "Invalid choice! Please try again." << endl;
        }
    } while (choice != 4);
    return 0;
```

```
}
```

A Dictionary stores keywords and its meanings. Provide facility for adding new keyword, deleting keywords, updating values of entry. Provide facility to display whole data sorted in ascending /Descending order. Also find how many maximum comparisons may require for finding any keyword. Use Binary Search Tree for implementation.

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

struct Node {
    string keyword, meaning;
    Node *left, *right;
    Node(string key, string m) {
        keyword = key;
        meaning = m;
        left = right = nullptr;
    }
};

Node* insert(Node* root, string key, string meaning) {
    if(root == nullptr)
        return new Node(key, meaning);
    if(key < root->keyword)
        root->left = insert(root->left, key, meaning);
    else if(key > root->keyword)
        root->right = insert(root->right, key, meaning);
    return root;
}

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

Node* deleteKeyword(Node* root, string key)
{
    if(root == nullptr)
        return root;
    if(key < root->keyword)
        root->left = deleteKeyword(root->left, key);
    else if(key > root->keyword)
        root->right = deleteKeyword(root->right, key);
    else {
```

```
        if(root->left == nullptr && root->right == nullptr)
            return nullptr;
        else if(root->left == nullptr)
            return root->right;
        else if(root->right == nullptr)
            return root->left;
        else {
            Node* minNode = findMinNode(root->right);
            root->keyword = minNode->keyword;
            root->meaning = minNode->meaning;
            root->right = deleteKeyword(root->right, minNode->keyword);
        }
    }
    return root;
}



void updateMeaning(Node* root, string key, string newMeaning){
    Node* current = root;
    while(current != nullptr){
        if(current->keyword == key){
            current ->meaning = newMeaning;
            cout << "Meaning updated." << endl;
            return;
        } else if (key < current->keyword){
            current = current->left;
        } else {
            current = current->right;
        }
    }
    cout << "Keyword not found." << endl;
}

//inorder traversal
void ascending(Node* &root){
    if(root != nullptr){
        ascending(root->left);
        cout << root->keyword << " " << root->meaning <<endl;
        ascending(root->right);
    }
}

//reverse inorder traversal
void decending(Node* &root){
```

```
    if(root != nullptr){
        decending(root->right);
        cout << root->keyword << " " << root->meaning <<endl;
        decending(root->left);
    }
}

int search(Node* root, string key)
{
    Node* current = root;
    int comparisons = 0;
    while(current != nullptr){
        comparisons++;
        if(key < current->keyword)
            current = current->left;
        else if(key > current->keyword)
            current = current->right;
        else {
            cout << "Keyword found: " << current->keyword << " - " << current-
>meaning << endl;
            return comparisons;
        }
    }
    return -1;
}

int main() {
    Node* root = nullptr;
    string keyword, meaning, newMeaning;
    char addMore;
    int comparisons;
    while(true){
        cout << "====DICTIONARY====" << endl;
        cout << "1. Add keyword" << endl;
        cout << "2. Display keywords in ascending order" << endl;
        cout << "3. Display keywords in descending order" << endl;
        cout << "4. Delete keyword" << endl;
        cout << "5. Search keyword" << endl;
        cout << "6. Update meaning" << endl;
        cout << "7. Exit" << endl;
        cout << "Enter your choice: ";
        int choice;
        cin >> choice;
        switch(choice){
            case 1:
```

```
            addMore = 'y';
            while (addMore == 'y' || addMore == 'Y') {
                cout << "Enter keyword: ";
                cin >> keyword;
                cout << "Enter meaning: ";
                cin.ignore();
                getline(cin, meaning);
                root = insert(root, keyword, meaning);
                cout << "Add more keywords? (y/n): ";
                cin >> addMore;
            }
        break;
        case 2:
            ascending(root);
            break;
        case 3:
            decending(root);
            break;
        case 4:
            cout << "Enter keyword to delete: ";
            cin >> keyword;
            root = deleteKeyword(root, keyword);
            break;
        case 5:
            cout << "Enter keyword to search: ";
            cin >> keyword;
            comparisons = search(root, keyword);
            if(comparisons == -1)
                cout << "Keyword not found." << endl;
            else
                cout << "Comparisons made: " << comparisons << endl;
            break;
        case 6:
            cout << "Enter keyword to update: ";
            cin >> keyword;
            cout << "Enter new meaning: ";
            cin.ignore();
            getline(cin, newMeaning);
            updateMeaning(root, keyword, newMeaning);
            break;
        case 7:
            cout << "Exiting..." << endl;
            return 0;
        default:
            cout << "Invalid choice. Try again." << endl;
```

```
        }
    }
}
```

Experiment No. 6

Represent a given graph using adjacency matrix/list to perform DFS and using adjacency list to perform BFS. Use the map of the area around the college as the graph. Identify the prominent landmarks as nodes and perform DFS and BFS on that.

```
#include<iostream>
#include<stack>
#include<queue>
#include<vector>
#include<map>
#include<iomanip>

using namespace std;

class Graph {
    private:
        int vertices;
        map<string, int> landmarkIndex;
        map<int, string> indexLandmark;
        vector<vector<int>> adjMatrix;
        map<int, vector<int>> adjList;

    public:
        Graph(int v){
            vertices = v;
            adjMatrix.resize(vertices, vector<int>(vertices, 0));
        }

        void addLandmark(string name, int index){
            landmarkIndex[name] = index;
            indexLandmark[index] = name;
        }

        void addEdge(string src, string dest){
            if(landmarkIndex.find(src) == landmarkIndex.end() ||
landmarkIndex.find(dest ) == landmarkIndex.end()){
                cout << "Invalid landmarks entered!\n";
                return;
```

```
        }
        int u = landmarkIndex[src];
        int v = landmarkIndex[dest];
        adjMatrix[u][v] = 1;
        adjMatrix[v][u] = 1; // For undirected graph
        adjList[u].push_back(v);
        adjList[v].push_back(u); // For undirected graph
    }

    void dfs()
    {
        cout << "Enter starting landmark for DFS: ";
        string startLandmark;
        getline(cin, startLandmark);
        if (landmarkIndex.find(startLandmark) == landmarkIndex.end()) {
            cout << "Invalid starting landmark!" << endl;
            return;
        }
        int start = landmarkIndex[startLandmark];

        vector<bool> visited(vertices, false);
        stack<int> s;
        s.push(start);
        visited[start] = true;

        cout << "DFS Traversal: ";
        while(!s.empty())
        {
            int node = s.top();
            s.pop();
            cout << indexLandmark[node] << " ";

            for( int i = 0; i < vertices; i++)
            {
                if(adjMatrix[node][i] != 0 && !visited[i])
                {
                    s.push(i);
                    visited[i] = true;
                }
            }
        }
    }

    void bfs()
    {
```

```
        cout << "Enter starting landmark for BFS: ";
        string startLandmark;
        getline(cin, startLandmark);
        if (landmarkIndex.find(startLandmark) == landmarkIndex.end()) {
            cout << "Invalid starting landmark!" << endl;
            return;
        }
        int start = landmarkIndex[startLandmark];

        vector<bool> visited(vertices, false);
        queue<int> q;
        q.push(start);
        visited[start] = true;

        cout << "BFS Traversal: ";
        while(!q.empty())
        {
            int node = q.front();
            q.pop();
            cout << indexLandmark[node] << " ";
            for( int neighbor : adjList[node])
            {
                if(!visited[neighbor]){
                    visited[neighbor] = true;
                    q.push(neighbor);
                }
            }
        }
     }

    void displayGraph()
    {
      int maxwidth = 0;
      for( int i = 0; i < vertices; i++){
         if(indexLandmark[i].length() > maxwidth){
             maxwidth = indexLandmark[i].length();
         }
      }

      maxwidth += 2; // For padding
      cout << "Graph Adjacency Matrix:" << endl;
      cout << setw(maxwidth) << " ";
      for( int i = 0; i < vertices; i++){
         cout << setw(maxwidth) << indexLandmark[i];
      }
```

```
        cout << endl;
        for ( int i = 0; i < vertices; i++){
           cout << setw(maxwidth) << indexLandmark[i];
           for( int j = 0; j < vertices; j++){
               cout << setw(maxwidth) << adjMatrix[i][j];
           }
           cout << endl;
        }

        cout << "Graph Adjacency List:" << endl;
        for( auto &pair : adjList){
           cout << setw(maxwidth) << indexLandmark[pair.first] << "-> ";
           for( int neighbor : pair.second){
               cout << indexLandmark[neighbor] << " ";
           }
           cout << endl;
        }

       }


};


int main()
{
    int num_landmarks;
    int choice;
    cout << "Enter the number of landmarks: ";
    cin >> num_landmarks;
    cin.ignore();

    Graph g(num_landmarks);
    cout << "Enter the landmarks (name): " << endl;
    for (int i = 0; i< num_landmarks; i++){
        string name;
        getline(cin, name);
        g.addLandmark(name, i);
    }

    //menu driven
    do{
        cout<< endl << "Menu: " << endl;
        cout<< "1. Add Edge" << endl;
        cout<< "2. Display Graph" << endl;
```

```
        cout<< "3. DFS Traversal" << endl;
        cout<< "4. BFS Traversal" << endl;
        cout<< "5. Exit" << endl;
        cout<< "Enter your choice: ";
        cin >> choice;
        cin.ignore(); // To ignore the newline character after the integer input
        switch (choice) {
            case 1: {
                string src, dest;
                cout << "Enter source landmark: ";
                getline(cin, src);
                cout << "Enter destination landmark: ";
                getline(cin, dest);
                g.addEdge(src, dest);
                break;
            }
            case 2:
                g.displayGraph();
                break;
            case 3:
                g.bfs();
                break;
            case 4:
                g.dfs();
                break;
            case 5:
                cout << "Exiting program.\n";
                return 0;
            default:
                cout << "Invalid choice! Try again.\n";
        }

    }while(choice != 5);

    return 0;


}
```

Experiment No. 7

You have a business with several offices; you want to lease phone lines to connect them up with each other; and the phone company charges different amounts of money to connect different pairs of cities. You want a set of lines that connects all your offices with a minimum total cost. Solve the problem by suggesting appropriate data structures.

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

class Prims {
    private:
        int vertices, edges;
        vector<vector<int>> adjMatrix;
    public:
        Prims(int v, int e) {
            vertices = v;
            edges = e;
            adjMatrix.resize(vertices, vector<int>(vertices, 0));
        }

        void addEdge(){
            cout<<"Enter edges (source destination weight):"<<endl;
            for(int i =0; i<edges; i++){
                int source, dest, weight;
                cin >>source>>dest>>weight;
                if(source >= 0 && source < vertices && dest >= 0 && dest <
vertices){
                    adjMatrix[source][dest] = weight;
                    adjMatrix[dest][source] = weight;
                } else{
                    cout<<"Invalid edge! Please enter valid vertices."<<endl;
                    i--;
                }
            }
        }

        void display(){
            cout <<endl<< "Adjacency Matrix:" <<endl<< "   ";
            for(int i=0; i < vertices; i++)
                cout << i << " ";
            cout <<endl;

            for(int i = 0 ; i < vertices; i++){
                cout << i << " ";
```

```
                for(int j = 0; j < vertices; j++){
                    cout << adjMatrix[i][j] << " ";
                }
                cout <<endl;
            }
        }

        void prims()
        {
            int parent[vertices];
            int minWeight[vertices];
            bool visited[vertices];
            for(int i = 0; i < vertices; i++){
                minWeight[i] = INT_MAX;
                visited[i] = false;
            }
            minWeight[0] = 0;
            parent[0]= -1;
            for(int edgeCount = 0; edgeCount < vertices - 1; edgeCount++){
                int vertex = minWeightVertex(visited, minWeight);
                visited[vertex] = true;

                for(int j = 0; j < vertices; j++){
                    if(adjMatrix[vertex][j] && !visited[j] &&
adjMatrix[vertex][j] < minWeight[j])
                    {
                        parent[j] = vertex;
                        minWeight[j] = adjMatrix[vertex][j];
                    }
                }
            }
            printMST(parent);
        }

        int minWeightVertex(bool visited[], int minWeight[]){
            int min = INT_MAX;
            int minIndex;
            for(int i=0; i < vertices -1 ; i++){
                if(!visited[i] && minWeight[i] < min){
                    min = minWeight[i];
                    minIndex = i;
                }
            }
            return minIndex;
        }
```

```
        void printMST(int parent[]){
            cout << endl << "Minimum Spanning Tree (Phone Line Connections):" <<
endl;
            cout << "Edge \tWeight" << endl;
            int totalCost = 0;
            for(int i = 1; i < vertices; i++){
                cout << parent[i] << " - " << i << "\t" <<
adjMatrix[parent[i]][i] << endl;
                totalCost += adjMatrix[parent[i]][i];
            }
            cout << "\nTotal cost of the minimum spanning tree: " << totalCost <<
endl;
        }
};

int main(){
    int vertices, edges;
    cout << "Enter number of vertices(offices): ";
    cin >> vertices;
    cout << "Enter number of edges(phone lines): ";
    cin >> edges;
    Prims p(vertices, edges);
    p.addEdge();

    int choice;
    while(true){
        cout << endl << "====MENU====" << endl;
        cout << "1. Display Adjacency Matrix" << endl;
        cout << "2. Find Minimum Spanning Tree" << endl;
        cout << "3. Exit" << endl;
        cout << "Enter your choice: ";
        cin >> choice;
        switch(choice){
            case 1:
                p.display();
                break;
            case 2:
                p.prims();
                break;
            case 3:
                cout << "Exiting..." << endl;
                return 0;
            default:
                cout << "Invalid choice! Please try again." << endl;
```

```
        }
    }
}
```

Experiment No. 8

Given sequence k = k1<K2…< Kn of n sorted keys, with a search probability pi for each key ki. Build the Binary search tree that has the least search cost given the access probability for each key?

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

struct TreeNode {
    string key;
    TreeNode* left;
    TreeNode* right;

    TreeNode(string k){
        key = k;
        left = right = nullptr;
    }
};

double calculateOBSTWeight(const double* P, const double* Q, int n) {
    double W[101][101] = {0};

    for (int i = 0; i <= n; i++) {
        W[i][i] = Q[i];
    }
    for (int i = 0; i <= n; i++) {
        for (int j = i + 1; j <= n; j++) {
            W[i][j] = W[i][j - 1] + P[j - 1] + Q[j];
        }
    }

    cout << "Weights in OBST format by level:" << endl;
    for (int level = 0; level <= n; level++) {
        for (int i = 0; i + level <= n; i++) {
            int j = i + level;
            cout << "W[" << i << "][" << j << "] = " << W[i][j] << " ";
```

```
        }
        cout << endl;
    }

    return W[0][n];
}

double calculateOBSTCost(const double* P, const double* Q, int n) {
    double W[101][101] = {0};
    double C[101][101] = {0};

    for (int i = 0; i <= n; i++) {
        W[i][i] = Q[i];
        C[i][i] = 0;
    }
    for (int i = 0; i <= n; i++) {
        for (int j = i + 1; j <= n; j++) {
            W[i][j] = W[i][j - 1] + P[j - 1] + Q[j];
        }
    }
    for (int level = 1; level <= n; level++) {
        for (int i = 0; i + level <= n; i++) {
            int j = i + level;
            C[i][j] = INT_MAX;
            for (int k = i + 1; k <= j; k++) {
                double cost = C[i][k - 1] + C[k][j];
                if (cost < C[i][j]) {
                    C[i][j] = cost;
                }
            }
            C[i][j] += W[i][j];
        }
    }

    cout << "Costs in OBST format by level:" << endl;
    for (int level = 0; level <= n; level++) {
        for (int i = 0; i + level <= n; i++) {
            int j = i + level;
            cout << "C[" << i << "][" << j << "] = " << C[i][j] << " ";
        }
        cout << endl;
    }

    return C[0][n];
}
```

```
int calculateOBSTRoots(const double* P, const double* Q, int n, int R[101][101])
{
    double W[101][101] = {0};
    double C[101][101] = {0};

    for (int i = 0; i <= n; i++) {
        W[i][i] = Q[i];
        C[i][i] = 0;
    }

    for (int i = 0; i <= n; i++) {
        for (int j = i + 1; j <= n; j++) {
            W[i][j] = W[i][j - 1] + P[j - 1] + Q[j];
        }
    }

    for (int level = 1; level <= n; level++) {
        for (int i = 0; i + level <= n; i++) {
            int j = i + level;
            C[i][j] = INT_MAX;
            for (int k = i + 1; k <= j; k++) {
                double cost = C[i][k - 1] + C[k][j];
                if (cost < C[i][j]) {
                    C[i][j] = cost;
                    R[i][j] = k;
                }
            }
            C[i][j] += W[i][j];
        }
    }

    cout << "Roots in OBST format by level:" << endl;
    for (int level = 1; level <= n; level++) {
        for (int i = 0; i + level <= n; i++) {
            int j = i + level;
            cout << "R[" << i << "][" << j << "] = " << R[i][j] << " ";
        }
        cout << endl;
    }

    return R[0][n];
}

// Function to build the OBST using the R table
```

```
TreeNode* buildOptimalBST(int i, int j, int R[101][101], const string* keys) {
    if (i > j) return nullptr;

    int k = R[i][j];
    if (k == 0) return nullptr;

    TreeNode* root = new TreeNode(keys[k-1]);
    root->left = buildOptimalBST(i, k-1, R, keys);
    root->right = buildOptimalBST(k, j, R, keys);

    return root;
}

void inorderTraversal(TreeNode* root) {
    if (root == nullptr) return;
    inorderTraversal(root->left);
    cout << root->key << " ";
    inorderTraversal(root->right);
}

void preorderTraversal(TreeNode* root) {
    if (root == nullptr) return;
    cout << root->key << " ";
    preorderTraversal(root->left);
    preorderTraversal(root->right);
}

void postorderTraversal(TreeNode* root) {
    if (root == nullptr) return;
    postorderTraversal(root->left);
    postorderTraversal(root->right);
    cout << root->key << " ";
}

int main() {
    int n = 0;
    string keys[100];
    double P[100], Q[101];
    double weight = 0, cost = 0;
    int R[101][101] = {0};
    int root = 0;
    TreeNode* rootNode = nullptr;
    bool dataEntered = false;
    int choice;
```

```
    cout << "Enter the number of keys: ";
    cin >> n;

    cout << "Enter the keywords: ";
    for (int i = 0; i < n; i++)
        cin >> keys[i];

    cout << "Enter the probabilities of keys (P): ";
    for (int i = 0; i < n; i++)
        cin >> P[i];

    cout << "Enter the probabilities of dummy keys (Q): ";
    for (int i = 0; i <= n; i++)
        cin >> Q[i];

    dataEntered = true;
    cout << "Data entered successfully!\n";

    do {
        cout << "\n\n==== OPTIMAL BINARY SEARCH TREE MENU ====\n";
        cout << "1. Calculate and display OBST weight\n";
        cout << "2. Calculate and display OBST cost\n";
        cout << "3. Calculate and display OBST roots\n";
        cout << "4. Build OBST and display tree traversals\n";
        cout << "5. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;

        switch(choice) {
            case 1:
                weight = calculateOBSTWeight(P, Q, n);
                cout << "The weight of the OBST is: " << weight << endl;
                break;
            case 2:
                cost = calculateOBSTCost(P, Q, n);
                cout << "The cost of the OBST is: " << cost << endl;
                break;
            case 3:
                root = calculateOBSTRoots(P, Q, n, R);
                cout << "The root of the OBST is: " << root << endl;
                break;
            case 4:
                rootNode = buildOptimalBST(0, n, R, keys);

                cout << "\nTree Traversals:" << endl;
```

```
                cout << "Inorder: ";
                inorderTraversal(rootNode);
                cout << endl;

                cout << "Preorder: ";
                preorderTraversal(rootNode);
                cout << endl;

                cout << "Postorder: ";
                postorderTraversal(rootNode);
                cout << endl;
                break;
            case 5:
                cout << "Exiting program. Goodbye!\n";
                break;
            default:
                cout << "Invalid choice. Please try again.\n";
                break;
        }
    } while (choice != 5);
    return 0;
}
// Sample Input:
// Enter the number of keys: 4
// Enter the keywords: out float if while
// Enter the probabilities of keys (P): 1 4 2 1
// Enter the probabilities of dummy keys (Q): 4 2 4 1 1
// Data entered successfully!
// Enter the number of keys: 4
// Enter the keywords: do if int while
// Enter the probabilities of keys (P): 3 3 1 1
// Enter the probabilities of dummy keys (Q): 2 3 1 1 1
//W[i][j] = w[i][j-1] + p[j-1] + q[j]
//C[i][j] = min(C[i][k-1] + C[k][j]) + w[i][j] where i < k <= j
//R[i][j] = k

//intially
//W[i][i] = Q[i]
//C[i][i] = 0
//R[i][i] = 0

//tree root R[0][n] = k
//every root R[i][j] = k
//left subtree R[i][k-1]
//right subtree R[k][j]
```

Experiment No. 9

A Dictionary stores keywords and its meanings. Provide facility for adding new keywords, deleting keywords, updating values of any entry. Provide a facility to display whole data sorted in ascending/ Descending order. Also find how many maximum comparisons may require for finding any keyword. Use Height balance tree and find the complexity for finding a keyword

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

struct Node {
    string keyword, meaning;
    Node* left;
    Node* right;
    int height;

    Node(string key, string m)
    {
        keyword = key;
        meaning = m;
        left = right = nullptr;
        height = 1;
    }
};

class AVLTree {
public:
    Node* root = nullptr;
    AVLTree(){
        root = nullptr;
    }

    int getHeight(Node* node) {
        if (node == nullptr) {
            return 0;
        }
        return node->height;
    }
```

```
    void updateHeight(Node* node) {
        if (node != nullptr) {
            node->height = 1 + max(getHeight(node->left), getHeight(node-
>right));
        }
    }

    int calculateBalanceFactor(Node* node) {
        if (node == nullptr) {
            return 0;
        }
        return getHeight(node->left) - getHeight(node->right);
    }

    Node* rightRotate(Node* y) {
        cout << "Performing Right Rotation at node: " << y->keyword << endl;
        Node* x = y->left;
        Node* T2 = x->right;

        x->right = y;
        y->left = T2;

        updateHeight(y); // Update y first (now child)
        updateHeight(x); // Update x last (new root)

        return x;
    }

    Node* leftRotate(Node* x) {
        cout << "Performing Left Rotation at node: " << x->keyword << endl;
        Node* y = x->right;
        Node* T2 = y->left;

        y->left = x;
        x->right = T2;

        updateHeight(x); // Update x first (now child)
        updateHeight(y); // Update y last (new root)

        return y;
    }

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

```
            node = node->left;
        }
        return node;
    }

    Node* balanceTree(Node* node) {
        if (node == nullptr) {
            return node;
        }

        updateHeight(node);
        int balance = calculateBalanceFactor(node);

        // Left Heavy Cases
        if (balance > 1) {
            // Left-Right Case
            if (calculateBalanceFactor(node->left) < 0) {
                cout << "LR rotation needed at " << node->keyword << endl;
                node->left = leftRotate(node->left);
            }
            // Left-Left Case (or after LR's first step)
             cout << "LL rotation needed at " << node->keyword << endl;
            return rightRotate(node);
        }
        // Right Heavy Cases
        else if (balance < -1) {
            // Right-Left Case
            if (calculateBalanceFactor(node->right) > 0) {
                 cout << "RL rotation needed at " << node->keyword << endl;
                node->right = rightRotate(node->right);
            }
            // Right-Right Case (or after RL's first step)
             cout << "RR rotation needed at " << node->keyword << endl;
            return leftRotate(node);
        }
        return node;
    }

    Node* insert(Node* node, const string& keyword, const string& meaning, bool&
inserted) {
        if (node == nullptr) {
            inserted = true;
            return new Node(keyword, meaning);
        }
```

```
        if (keyword < node->keyword) {
            node->left = insert(node->left, keyword, meaning, inserted);
        } else if (keyword > node->keyword) {
            node->right = insert(node->right, keyword, meaning, inserted);
        } else {
            cout << "Keyword '" << keyword << "' already exists. Use update
option." << endl;
            inserted = false;
            return node;
        }

        if (!inserted) {
            return node;
        }

        return balanceTree(node);
    }

    Node* remove(Node* node, const string& keyword, bool& deleted) {
        if (node == nullptr) {
            deleted = false;
            return node;
        }

        if (keyword < node->keyword) {
            node->left = remove(node->left, keyword, deleted);
        } else if (keyword > node->keyword) {
            node->right = remove(node->right, keyword, deleted);
        } else {
            deleted = true;

            // Case 1 & 2: Node with only one child or no child
            if (node->left == nullptr || node->right == nullptr) {
                Node* temp = node->left ? node->left : node->right;

                if (temp == nullptr) {
                   temp = node;
                   node = nullptr;
                } else {
                   *node = *temp;
                }
                delete temp;
                temp = nullptr;
            } else {
```

```
                // Case 3: Node with two children - replace with inorder
successor
                Node* temp = findMinNode(node->right);
                node->keyword = temp->keyword;
                node->meaning = temp->meaning;

                bool dummy_deleted = true;
                node->right = remove(node->right, temp->keyword, dummy_deleted);
            }
        }

        if (node == nullptr) {
            return node;
        }

        if (deleted) {
             return balanceTree(node);
        } else {
             return node;
        }
    }

    void printInOrder(Node* node) {
        if (node != nullptr) {
            printInOrder(node->left);
            cout << node->keyword << " (BF: " << calculateBalanceFactor(node) <<
") : " << node->meaning << endl;
            printInOrder(node->right);
        }
    }

    void printReverseInOrder(Node* node) {
         if (node != nullptr) {
            printReverseInOrder(node->right);
            cout << node->keyword << " (BF: " << calculateBalanceFactor(node) <<
") : " << node->meaning << endl;
            printReverseInOrder(node->left);
        }
    }

    Node* findNode(const string& keyword) {
         Node* current = root;
         while (current != nullptr) {
             if (keyword == current->keyword) {
                 return current;
```

```
            } else if (keyword < current->keyword) {
                current = current->left;
            } else {
                current = current->right;
            }
        }
        return nullptr;
    }

    string search(const string& keyword, int& comparisons) {
        Node* current = root;
        comparisons = 0;
        while (current != nullptr) {
            comparisons++;
            if (keyword == current->keyword) {
                return current->meaning;
            } else if (keyword < current->keyword) {
                current = current->left;
            } else {
                current = current->right;
            }
        }
        return "Keyword not found.";
    }

    bool updateMeaning(const string& keyword, const string& newMeaning) {
        Node* node = findNode(keyword);
        if (node != nullptr) {
            node->meaning = newMeaning;
            return true;
        }
        return false;
    }

    void printLevelOrder() {
        if (root == nullptr) {
            cout << "Tree is empty." << endl;
            return;
        }

        queue<Node*> q;
        q.push(root);
        int level = 0;

        cout << "--- Level Order Traversal ---" << endl;
```

```
        while (!q.empty()) {
            int levelSize = q.size();
            cout << "Level " << level << ": ";
            bool level_has_nodes = false;
            for (int i = 0; i < levelSize; ++i) {
                Node* current = q.front();
                q.pop();

                if (current != nullptr) {
                    cout << current->keyword << "(BF:" <<
calculateBalanceFactor(current) << ") ";
                    level_has_nodes = true;
                    q.push(current->left);
                    q.push(current->right);
                } else {
                     cout << "null ";
                }
            }
            cout << endl;

            // Check if remaining nodes are all null
            bool all_null = true;
            queue<Node*> temp_q = q;
            while (!temp_q.empty()) {
                if (temp_q.front() != nullptr) {
                    all_null = false;
                    break;
                }
                temp_q.pop();
            }
            if (all_null) break;

            level++;
        }
        cout << "-----------------------------" << endl;
    }

    int getTreeHeight() {
        return getHeight(root);
    }

    bool isEmpty() const {
        return root == nullptr;
    }
```

```
};

int main() {
    AVLTree dictionary;
    string keyword, meaning, newMeaning;
    int comparisons = 0;
    char addMore;
    int choice;
    bool inserted, deleted;

    while (true) {
        cout << "\n==== AVL Dictionary Menu ====\n";
        cout << "1. Add new keyword(s)\n";
        cout << "2. Print Ascending\n";
        cout << "3. Print Descending\n";
        cout << "4. Delete a keyword\n";
        cout << "5. Search a keyword\n";
        cout << "6. Update meaning\n";
        cout << "7. Get Tree Height\n";
        cout << "8. Display Tree Level Order\n";
        cout << "9. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;

        switch (choice) {
            case 1:
                addMore = 'y';
                while (addMore == 'y' || addMore == 'Y') {
                    cout << "Enter keyword: ";
                    cin >> keyword;
                    cout << "Enter meaning: ";
                    cin.ignore();
                    getline(cin, meaning);

                    inserted = false;
                    dictionary.root = dictionary.insert(dictionary.root, keyword,
meaning, inserted);
                    if (inserted) {
                        cout << "Keyword '" << keyword << "' inserted." << endl;
                    }

                    dictionary.printLevelOrder();

                    cout << "Add more keywords? (y/n): ";
                    cin >> addMore;
```

```
                }
                break;
            case 2:
                if (dictionary.isEmpty()) {
                    cout << "Dictionary is empty." << endl;
                } else {
                    cout << "--- Dictionary Entries (Ascending) ---" << endl;
                    dictionary.printInOrder(dictionary.root);
                    cout << "-------------------------------------" << endl;
                }
                break;
            case 3:
                if (dictionary.isEmpty()) {
                    cout << "Dictionary is empty." << endl;
                } else {
                    cout << "--- Dictionary Entries (Descending) ---" << endl;
                    dictionary.printReverseInOrder(dictionary.root);
                    cout << "-------------------------------------" << endl;
                }
                break;
            case 4:
                if (dictionary.isEmpty()) {
                    cout << "Dictionary is empty. Cannot delete." << endl;
                } else {
                    cout << "Enter keyword to delete: ";
                    cin >> keyword;
                    deleted = false;
                    dictionary.root = dictionary.remove(dictionary.root, keyword,
deleted);
                    if (deleted) {
                        cout << "Keyword '" << keyword << "' deleted." << endl;
                        dictionary.printLevelOrder();
                    } else {
                        cout << "Keyword '" << keyword << "' not found, cannot
delete." << endl;
                    }
                }
                break;
            case 5:
                 if (dictionary.isEmpty()) {
                     cout << "Dictionary is empty." << endl;
                } else {
                    cout << "Enter keyword to search: ";
                    cin >> keyword;
                    comparisons = 0;
```

```
                    string result = dictionary.search(keyword, comparisons);
                    if (result == "Keyword not found.") {
                         cout << "Keyword '" << keyword << "' not found.
(Comparisons: " << comparisons << ")" << endl;
                    } else {
                         cout << "Meaning: " << result << " (Comparisons: " <<
comparisons << ")" << endl;
                    }
                }
                break;
            case 6:
                 if (dictionary.isEmpty()) {
                     cout << "Dictionary is empty." << endl;
                } else {
                    cout << "Enter keyword to update: ";
                    cin >> keyword;
                    cout << "Enter new meaning: ";
                    cin.ignore();
                    getline(cin, newMeaning);
                    if (dictionary.updateMeaning(keyword, newMeaning)) {
                        cout << "Meaning updated successfully." << endl;
                    } else {
                        cout << "Keyword '" << keyword << "' not found, cannot
update." << endl;
                    }
                }
                break;
            case 7:
                cout << "Tree Height: " << dictionary.getTreeHeight() << endl;
                break;
            case 8:
                cout << "Current Tree State:" << endl;
                dictionary.printLevelOrder();
                break;
            case 9:
                 cout << "Exiting program." << endl;
                return 0;
            default:
                cout << "Invalid choice. Please choose a valid option." << endl;
        }
    }
}
```

Consider a scenario for hospital to cater services to different kinds of patients as Serious (top priority), b) non-serious (medium priority), c) General Check-up (Least priority). Implement the priority queue to cater services to the patients.

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

struct Patient
{
    string name;
    int priority;
    Patient *next;
    Patient(string n,int p){
        name = n;
        priority = p;
        next = nullptr;
    }
};

class PriorityQueue{
    private:
        Patient *head;
    public:
        PriorityQueue(){
            head = nullptr;
        }

        void enqueue(string name, int priority)
        {
            Patient *newPatient = new Patient(name,priority);
            if(head == nullptr || priority < head->priority){
                newPatient->next = head;
                head = newPatient;
            } else {
                Patient* current = head;
                while(current->next != nullptr && current->next->priority <=
priority)
                    current = current->next;
                newPatient->next = current->next;
                current->next = newPatient;
            }
        }

        void dequeue()
        {
```

```
            if(head == nullptr){
                cout << "The queue is empty. No patients to serve."<<endl;
                return;
            }
            Patient *current = head;
            head = head->next;
            cout << "Serving patient: " << current->name << " (Priority: " <<
current->priority << ")"<<endl;
            delete current;
        }

        void display()
        {
            if(head == nullptr){
                cout << "No patients in the queue."<<endl;
                return;
            }
            Patient *current = head;
            cout << "Patients in the queue" <<endl;
            while(current != nullptr){
                cout << current->name << " (Priority: " << current->priority <<
")"<<endl;
                current = current->next;
            }
        }
};

int main()
{
    PriorityQueue pq;
    int choice, priority;
    string name;
    while(true){
        cout << "\n====MENU====" << endl;
        cout << "1. Add patient" << endl;
        cout << "2. Serve patient" << endl;
        cout << "3. Display patients" << endl;
        cout << "4. Exit" << endl;
        cout << "Enter your choice: ";
        cin >> choice;
        switch(choice){
            case 1:
                cout << "Enter patient name: ";
                cin >> name;
                cout << "Enter priority (lower number = higher priority): ";
```

```
                cin >> priority;
                pq.enqueue(name, priority);
                break;
            case 2:
                pq.dequeue();
                break;
            case 3:
                pq.display();
                break;
            case 4:
                cout << "Exiting..."<<endl;
                return 0;
            default:
                cout << "Invalid choice. Try again."<<endl;
        }
    }
}
```

The department maintains student information. The file contains roll number, name, division and address. Allow users to add, delete information about students. Display information of a particular employee. If the record of the student does not exist an appropriate message is displayed. If it is, then the system displays the student details. Use a sequential file to maintain the data.

```
#include<iostream>
#include<fstream>
#include<iomanip>
#include<sstream>
using namespace std;

struct Student {
    string rollNum, name, division, address;
};

void addStudent()
{
    ofstream outFile("students.txt", ios::app);
    if(!outFile){
        cerr << "Error opening file for writing." << endl;
        return;
    }
    Student s;
    cout << "Enter Roll Number: ";
    cin >> s.rollNum;
    cout << "Enter Name: ";
    cin >> s.name;
```

```
    cout << "Enter Division: ";
    cin >> s.division;
    cout << "Enter Address: ";
    cin >> s.address;
    outFile << s.rollNum << "," << s.name << "," << s.division << ","
        << s.address << endl;
    outFile.close();
    cout << "Student added successfully." << endl;
}

void deleteStudent(const string &roll)
{
    ifstream inFile("students.txt");
    ofstream outFile("temp.txt");
    if(!inFile || !outFile){
        cerr << "Error opening file." << endl;
        return;
    }
    string line;
    bool found = false;
    while(getline(inFile, line)){
        stringstream ss(line);
        string tokens[4];
        for(int i =0 ; i < 4; i++)
            getline(ss, tokens[i],',');
        if(tokens[0] == roll){
            found = true;
        } else {
            outFile << line << endl;
        }
    }
    inFile.close();
    outFile.close();
    if(found){
        remove("students.txt");
        rename("temp.txt", "students.txt");
        cout << "Student with roll number " << roll << " deleted successfully."
<< endl;
    } else {
        remove("temp.txt");
        cout << "Student with roll number " << roll << " not found." << endl;
    }
}

void searchStudent(const string &roll)
```

```
{
    ifstream inFile("students.txt");
    if(!inFile){
        cerr << "Error opening file." << endl;
        return;
    }
    string line;
    bool found = false;
    while(getline(inFile, line)){
        stringstream ss(line);
        string tokens[4];
        for(int i =0 ; i < 4; i++)
            getline(ss, tokens[i],',');
        if(tokens[0] == roll){
            cout << "Student found:" << endl;
            cout << setw(15) << left << "Roll Number" << setw(20) << left <<
"Name" << setw(15) << left << "Division" << setw(30) << left << "Address" <<
endl;
            cout << setw(15) << left << tokens[0] << setw(20) << left <<
tokens[1] << setw(15) << left << tokens[2] << setw(30) << left << tokens[3] <<
endl;
            found = true;
            break;
        }
    }
    inFile.close();
    if(!found){
        cout << "Student with roll number " << roll << " not found." << endl;
    }
}

void displayAllStudents()
{
    ifstream inFile("students.txt");
    if(!inFile){
        cerr << "Error opening file for reading." << endl;
        return;
    }
    string line;
    cout << "Reading file..." << endl;
    cout << setw(15) << left << "Roll Number" << setw(20) << left << "Name" <<
setw(15) << left << "Division" << setw(30) << left << "Address" << endl;
    while(getline(inFile, line)){
        stringstream ss(line);
        string tokens[4];
```

```
        for(int i =0 ; i < 4; i++)
            getline(ss, tokens[i],',');
        cout << setw(15) << left << tokens[0] << setw(20) << left << tokens[1] <<
setw(15) << left << tokens[2] << setw(30) << left << tokens[3] << endl;
    }
    inFile.close();
}

int main()
{
    int choice;
    string roll;
    while(true){
        cout << "\n====MENU====" << endl;
        cout << "1. Add Student" << endl;
        cout << "2. Delete Student" << endl;
        cout << "3. Search Student" << endl;
        cout << "4. Display All Students" << endl;
        cout << "5. Exit" << endl;
        cout << "Enter your choice: ";
        cin >> choice;
        switch(choice){
            case 1:
                addStudent();
                break;
            case 2:
                cout << "Enter Roll Number to delete: ";
                cin >> roll;
                deleteStudent(roll);
                break;
            case 3:
                cout << "Enter Roll Number to search: ";
                cin >> roll;
                searchStudent(roll);
                break;
            case 4:
                displayAllStudents();
                break;
            case 5:
                cout << "Exiting..." << endl;
                return 0;
            default:
                cout << "Invalid choice. Please try again." << endl;
        }
    }
```

```
}
```

Implementation of a direct access file -Insertion and deletion of a record from a direct access file

```
#include<iostream>
#include<fstream>
#include<iomanip>

using namespace std;
const char FILENAME[] = "persons.dat";
const int MAX_RECORDS = 10;
struct Person{
    int id;
    char name[50];
    int age;
    Person(){
        id = -1;
        age = 0;
        name[0] = '\0';
    }
};

void initializeFile(){
    fstream file(FILENAME, ios::in | ios::binary);
    if (!file){
        file.open(FILENAME, ios::out | ios::binary);
        Person p;
        for (int i = 0; i < MAX_RECORDS; ++i)
            file.write((char*)&p, sizeof(Person));
            //or file.write(reinterpret_case<char*>(&p), sizeof(Person));
        file.close();
    }
}

void insertRecord(const Person& p){
    if(p.id < 0 || p.id >= MAX_RECORDS){
        cout << "Invalid ID. Must be between 0 and " << MAX_RECORDS-1 << endl;
        return;
    }
    fstream file(FILENAME, ios::in | ios::out | ios::binary);
    file.seekg(p.id * sizeof(Person));
    Person temp;
    file.read((char*)&temp, sizeof(Person));
    if(temp.id != -1){
        cerr << "Slot already filled. Delete first to insert.\n";
```

```
        file.close();
        return;
    }
    file.seekp(p.id * sizeof(Person));
    file.write((char*)&p, sizeof(Person));
    file.close();
    cout << "Record inserted at slot " << p.id << endl;
}

void deleteRecord(int id){
    if(id < 0 || id >= MAX_RECORDS){
        cout << "Invalid ID. Must be between 0 and " << MAX_RECORDS-1 << endl;
        return;
    }
    fstream file(FILENAME, ios::in | ios::out | ios::binary);
    file.seekg(id * sizeof(Person));
    Person temp;
    file.read((char*)&temp, sizeof(Person));
    if(temp.id == -1){
        cout << "Slot already empty.\n";
        file.close();
        return;
    }
    Person empty;
    file.seekp(id * sizeof(Person));
    file.write((char*)&empty, sizeof(Person));
    file.close();
    cout << "Record deleted from slot " << id << endl;
}

void displayFile(){
    ifstream file(FILENAME, ios::in | ios::binary);
    if(!file){
        cerr << "Error opening file." << endl;
        return;
    }
    cout << left << setw(8) << "Sr. No" << setw(8) << "Status" << setw(8) << "ID"
<< setw(20) << "Name" << setw(8) << "Age" << endl;
    cout << "-------------------------------------------------------" << endl;
    Person p;
    for(int i = 0; i < MAX_RECORDS; ++i){
        file.read((char*)&p, sizeof(Person));
        if(p.id == -1){
            cout << left << setw(8) << i << setw(8) << "Empty" << setw(8) << "-"
<< setw(20) << '-' << setw(8) << "-" << endl;
```

```
        } else {
            cout << left << setw(8) << i << setw(8) << "Filled" << setw(8) <<
p.id << setw(20) << p.name << setw(8) << p.age << endl;
        }
    }
    file.close();
}

int main()
{
    initializeFile();
    Person p;
    int choice;
    while(true){
        cout << "\n====MENU====" << endl;
        cout << "1. Insert record" << endl;
        cout << "2. Delete record" << endl;
        cout << "3. Display records" << endl;
        cout << "4. Exit" << endl;
        cout << "Enter your choice: ";
        cin >> choice;
        switch(choice){
            case 1:
                cout << "Enter ID (0-" << MAX_RECORDS-1 << "): ";
                cin >> p.id;
                cout << "Enter name: ";
                cin.ignore();
                cin.getline(p.name, 50);
                cout << "Enter age: ";
                cin >> p.age;
                insertRecord(p);
                break;
            case 2:
                int id;
                cout << "Enter ID to delete (0-" << MAX_RECORDS-1 << "): ";
                cin >> id;
                deleteRecord(id);
                break;
            case 3:
                displayFile();
                break;
            case 4:
                return 0;
            default:
                cout << "Invalid choice. Try again."<<endl;
```

```
        }
    }
}
```