* **Code(AVL):**

#include <iostream>

#include <algorithm>

using namespace std;

class AVLNode {

public:

    int key;

    int height;

    AVLNode\* left;

    AVLNode\* right;

    AVLNode(int k) : key(k), height(1), left(NULL), right(NULL) {}

};

class AVLTree {

private:

    AVLNode\* root;

    int getHeight(AVLNode\* node) {

        return (node == NULL) ? 0 : node->height;

    }

    int getBalanceFactor(AVLNode\* node) {

        return getHeight(node->left) - getHeight(node->right);

    }

    void updateHeight(AVLNode\* node) {

        node->height = 1 + max(getHeight(node->left), getHeight(node->right));

    }

    AVLNode\* rotateRight(AVLNode\* y) {

        AVLNode\* x = y->left;

        AVLNode\* T2 = x->right;

        x->right = y;

        y->left = T2;

        updateHeight(y);

        updateHeight(x);

        return x;

    }

    AVLNode\* rotateLeft(AVLNode\* x) {

        AVLNode\* y = x->right;

        AVLNode\* T2 = y->left;

        y->left = x;

        x->right = T2;

        updateHeight(x);

        updateHeight(y);

        return y;

    }

    AVLNode\* insert(AVLNode\* node, int key) {

        if (node == NULL) {

            return new AVLNode(key);

        }

        if (key < node->key) {

            node->left = insert(node->left, key);

        } else if (key > node->key) {

            node->right = insert(node->right, key);

        } else {

            return node;

        }

        updateHeight(node);

        int balance = getBalanceFactor(node);

        if (balance > 1) {

            if (key < node->left->key) {

                return rotateRight(node);

            } else {

                node->left = rotateLeft(node->left);

                return rotateRight(node);

            }

        }

        if (balance < -1) {

            if (key > node->right->key) {

                return rotateLeft(node);

            } else {

                node->right = rotateRight(node->right);

                return rotateLeft(node);

            }

        }

        return node;

    }

    AVLNode\* minValueNode(AVLNode\* node) {

        AVLNode\* current = node;

        while (current->left != NULL) {

            current = current->left;

        }

        return current;

    }

    AVLNode\* deleteNode(AVLNode\* root, int key) {

        if (root == NULL) {

            return root;

        }

        if (key < root->key) {

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

        } else if (key > root->key) {

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

        } else {

            // Node with only one child or no child

            if (root->left == NULL) {

                AVLNode\* temp = root->right;

                delete root;

                return temp;

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

                AVLNode\* temp = root->left;

                delete root;

                return temp;

            }

            // Node with two children

            AVLNode\* temp = minValueNode(root->right);

            root->key = temp->key;

            root->right = deleteNode(root->right, temp->key);

        }

        updateHeight(root);

        int balance = getBalanceFactor(root);

        if (balance > 1) {

            if (getBalanceFactor(root->left) >= 0) {

                return rotateRight(root);

            } else {

                root->left = rotateLeft(root->left);

                return rotateRight(root);

            }

        }

        if (balance < -1) {

            if (getBalanceFactor(root->right) <= 0) {

                return rotateLeft(root);

            } else {

                root->right = rotateRight(root->right);

                return rotateLeft(root);

            }

        }

        return root;

    }

    AVLNode\* search(AVLNode\* root, int key) {

        if (root == NULL || root->key == key) {

            return root;

        }

        if (key < root->key) {

            return search(root->left, key);

        }

        return search(root->right, key);

    }

    void inorderTraversal(AVLNode\* root) {

        if (root != NULL) {

            inorderTraversal(root->left);

            cout << root->key << " ";

            inorderTraversal(root->right);

        }

    }

public:

    AVLTree() : root(NULL) {}

    void insert(int key) {

        root = insert(root, key);

    }

    void remove(int key) {

        root = deleteNode(root, key);

    }

    bool search(int key) {

        return search(root, key) != NULL;

    }

    void printInorder() {

        cout << "Inorder Traversal: ";

        inorderTraversal(root);

        cout << std::endl;

    }

};

int main() {

    AVLTree avl;

    avl.insert(10);

    avl.insert(20);

    avl.insert(30);

    avl.insert(40);

    avl.insert(50);

    avl.insert(60);

    avl.printInorder();

    avl.remove(20);

    avl.printInorder();

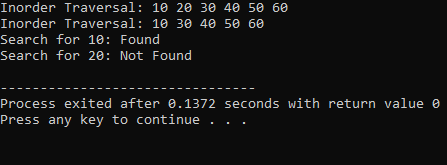
    std::cout << "Search for 10: " << (avl.search(10) ? "Found" : "Not Found") << std::endl;

    std::cout << "Search for 20: " << (avl.search(20) ? "Found" : "Not Found") << std::endl;

    return 0;

}

* **Output:**



* **Code(BST):**

#include <iostream>

#include <string>

using namespace std;

class TreeNode {

public:

    int id;

    string name;

    double salary;

    TreeNode\* left;

    TreeNode\* right;

    TreeNode(int i, string n, double s) {

        id = i;

        name = n;

        salary = s;

        left = right = NULL;

    }

};

class BST {

private:

    TreeNode\* root;

public:

    BST() {

        root = NULL;

    }

    void insert(int id, string name, double salary) {

        root = insertRecursive(root, id, name, salary);

    }

    void displayInOrder() {

        inOrderRecursive(root);

    }

    void displayPreOrder() {

        preOrderRecursive(root);

    }

    void displayPostOrder() {

        postOrderRecursive(root);

    }

private:

    TreeNode\* insertRecursive(TreeNode\* root, int id, string name, double salary) {

        if (root == NULL) {

            return new TreeNode(id, name, salary);

        }

        if (id < root->id) {

            root->left = insertRecursive(root->left, id, name, salary);

        } else if (id > root->id) {

            root->right = insertRecursive(root->right, id, name, salary);

        }

        return root;

    }

    void inOrderRecursive(TreeNode\* root) {

        if (root != NULL) {

            inOrderRecursive(root->left);

            cout << "Id: " << root->id << ", Name: " << root->name <<  ", Salary: " << root->salary<<endl;

            inOrderRecursive(root->right);

        }

    }

    void preOrderRecursive(TreeNode\* root) {

        if (root != NULL) {

            cout << "Id: " << root->id << ", Name: " << root->name <<  ", Salary: " << root->salary<<endl;

            preOrderRecursive(root->left);

            preOrderRecursive(root->right);

        }

    }

    void postOrderRecursive(TreeNode\* root) {

        if (root != NULL) {

            postOrderRecursive(root->left);

            postOrderRecursive(root->right);

            cout << "Id: " << root->id << ", Name: " << root->name <<  ", Salary: " << root->salary<<endl;

        }

    }

};

int main() {

    BST bst;

    bst.insert(1, "Sushant", 100000);

    bst.insert(2, "Ajay", 75000);

    bst.insert(3, "Zoro", 50000);

    cout << "In Order Traversal:" << endl;

    bst.displayInOrder();

    cout << "Pre Order Traversal:" << endl;

    bst.displayPreOrder();

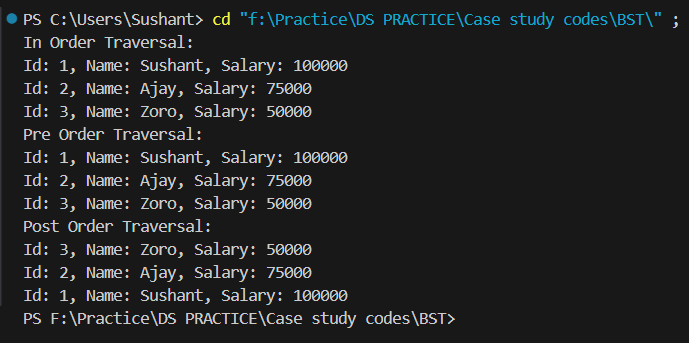
    cout << "Post Order Traversal:" << endl;

    bst.displayPostOrder();

    return 0;

}

* **Output:**

****

* **Code(Bubble Sort):**

def bubble\_sort\_books(book\_list):

    n = len(book\_list)

    for i in range(n):

        for j in range(0, n - i - 1):

            if book\_list[j]["book\_id"] > book\_list[j + 1]["book\_id"]:

                temp = book\_list[j]

                book\_list[j] = book\_list[j + 1]

                book\_list[j + 1] = temp

books = [

    {"book\_id": 103, "title": "Book C", "author": "Author C", "quantity": 5},

    {"book\_id": 101, "title": "Book A", "author": "Author A", "quantity": 3},

    {"book\_id": 102, "title": "Book B", "author": "Author B", "quantity": 8},

]

bubble\_sort\_books(books)

print("Books after Bubble Sort:")

for book in books:

    print(book)

**OR**

#include <iostream>

#include <vector>

using namespace std;

struct Book {

    int book\_id;

    string title;

    string author;

    int quantity;

};

void bubble\_sort\_books(vector<Book> &book\_list) {

    int n = book\_list.size();

    for (int i = 0; i < n; ++i) {

        for (int j = 0; j < n - i - 1; ++j) {

            if (book\_list[j].book\_id > book\_list[j + 1].book\_id) {

                // Swap the books

                Book temp = book\_list[j];

                book\_list[j] = book\_list[j + 1];

                book\_list[j + 1] = temp;

            }

        }

    }

}

int main() {

    vector<Book> books = {

        {103, "Book C", "Author C", 5},

        {101, "Book A", "Author A", 3},

        {102, "Book B", "Author B", 8}

    };

    bubble\_sort\_books(books);

    cout << "Books after Bubble Sort:" << endl;

    for (const auto &book : books) {

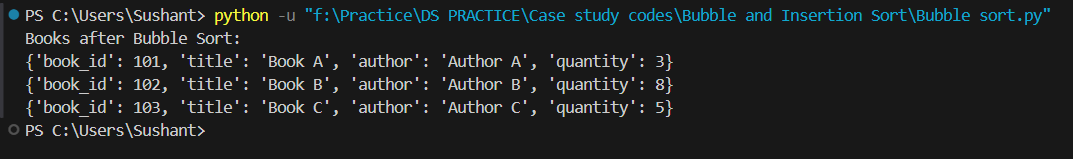
        cout << "Book ID: " << book.book\_id << ", Title: " << book.title << ", Author: " << book.author << ", Quantity: " << book.quantity << endl;

    }

    return 0;

}

* **Output:**



* **Code(Insertion Sort):**

def insertion\_sort\_books(book\_list):

    for i in range(1, len(book\_list)):

        current\_book = book\_list[i]

        j = i - 1

        while j >= 0 and current\_book["book\_id"] < book\_list[j]["book\_id"]:

            book\_list[j + 1] = book\_list[j]

            j -= 1

        book\_list[j + 1] = current\_book

books = [

    {"book\_id": 103, "title": "Book C", "author": "Author C", "quantity": 5},

    {"book\_id": 101, "title": "Book A", "author": "Author A", "quantity": 3},

    {"book\_id": 102, "title": "Book B", "author": "Author B", "quantity": 8},

]

insertion\_sort\_books(books)

print("Books after Insertion Sort:")

for book in books:

    print(book)

**OR**

#include <iostream>

#include <vector>

using namespace std;

struct Book {

    int book\_id;

    string title;

    string author;

    int quantity;

};

void insertion\_sort\_books(vector<Book> &book\_list) {

    for (int i = 1; i < book\_list.size(); ++i) {

        Book current\_book = book\_list[i];

        int j = i - 1;

        while (j >= 0 && current\_book.book\_id < book\_list[j].book\_id) {

            book\_list[j + 1] = book\_list[j];

            --j;

        }

        book\_list[j + 1] = current\_book;

    }

}

int main() {

    vector<Book> books = {

        {103, "Book C", "Author C", 5},

        {101, "Book A", "Author A", 3},

        {102, "Book B", "Author B", 8}

    };

    insertion\_sort\_books(books);

    cout << "Books after Insertion Sort:" << endl;

    for (const auto &book : books) {

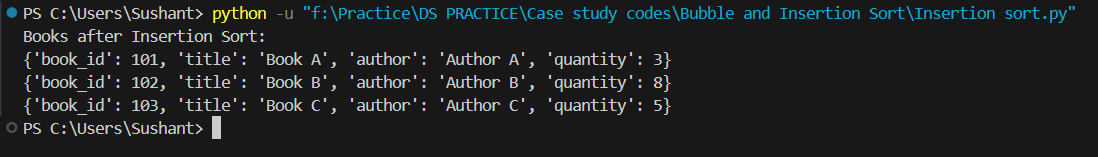
        cout << "Book ID: " << book.book\_id << ", Title: " << book.title << ", Author: " << book.author << ", Quantity: " << book.quantity << endl;

    }

    return 0;

}

* **Output:**



* **Code(dictionary):**

#include <iostream>

#include <map>

#include <string>

using namespace std;

map<string, string> capitals;

void insert\_elements()

{

    string key;

    string value;

    cout << "\nEnter the name of the country : ";

    cin >> key;

    cout << "Enter the capital of " << key << " : ";

    cin >> value;

    capitals[key] = value;

}

void print\_elements()

{

    for (auto item : capitals)

    {

        cout << "Name of the country " << item.first << ": Name of the capital ";

        cout << item.second << endl;

    }

}

void delete\_elements()

{

    string key\_to\_be\_deleted;

    cout << "\nEnter the name of the country that you want to delete : ";

    cin >> key\_to\_be\_deleted;

    capitals.erase(key\_to\_be\_deleted);

}

void search\_elements()

{

    string key\_to\_be\_searched;

    cout << "\nEnter the name of the country that you want to search : ";

    cin >> key\_to\_be\_searched;

    cout << "Capital of " << key\_to\_be\_searched << " is " << capitals[key\_to\_be\_searched] << "\n";

}

void update\_elements()

{

    string key\_to\_be\_updated;

    string new\_key;

    cout << "\nEnter the name of the country whose capital you want to update : ";

    cin >> key\_to\_be\_updated;

    cout << "Enter the name of new capital : ";

    cin >> new\_key;

    capitals[key\_to\_be\_updated] = new\_key;

}

int main()

{

    int choice;

    cout << "\n1. To insert data into the Dictionary." << endl;

    cout << "2. To print data from the Dictionary." << endl;

    cout << "3. To delete data from the Dictionary." << endl;

    cout << "4. To search data from the Dictionary." << endl;

    cout << "5. To update data from the Dictionary." << endl;

    cout << "0. To exit the code." << endl;

    while (1)

    {

        cout << "\nEnter your choice: ";

        cin >> choice;

        switch (choice)

        {

        case 0:

            exit(0);

        case 1:

            insert\_elements();

            break;

        case 2:

            cout << endl;

            cout << "Contents of the Dictionary are :- \n";

            print\_elements();

            break;

        case 3:

            cout << endl;

            delete\_elements();

            cout << "Element deleted sucessfully.\n";

            break;

        case 4:

            cout << endl;

            cout << "Result of the search in the dictionary is :- ";

            search\_elements();

            break;

        case 5:

            cout << endl;

            update\_elements();

            cout << "Contents of the Dictionary updated sucessfully.\n";

            break;

        default:

            cout << "Please Enter valid input.";

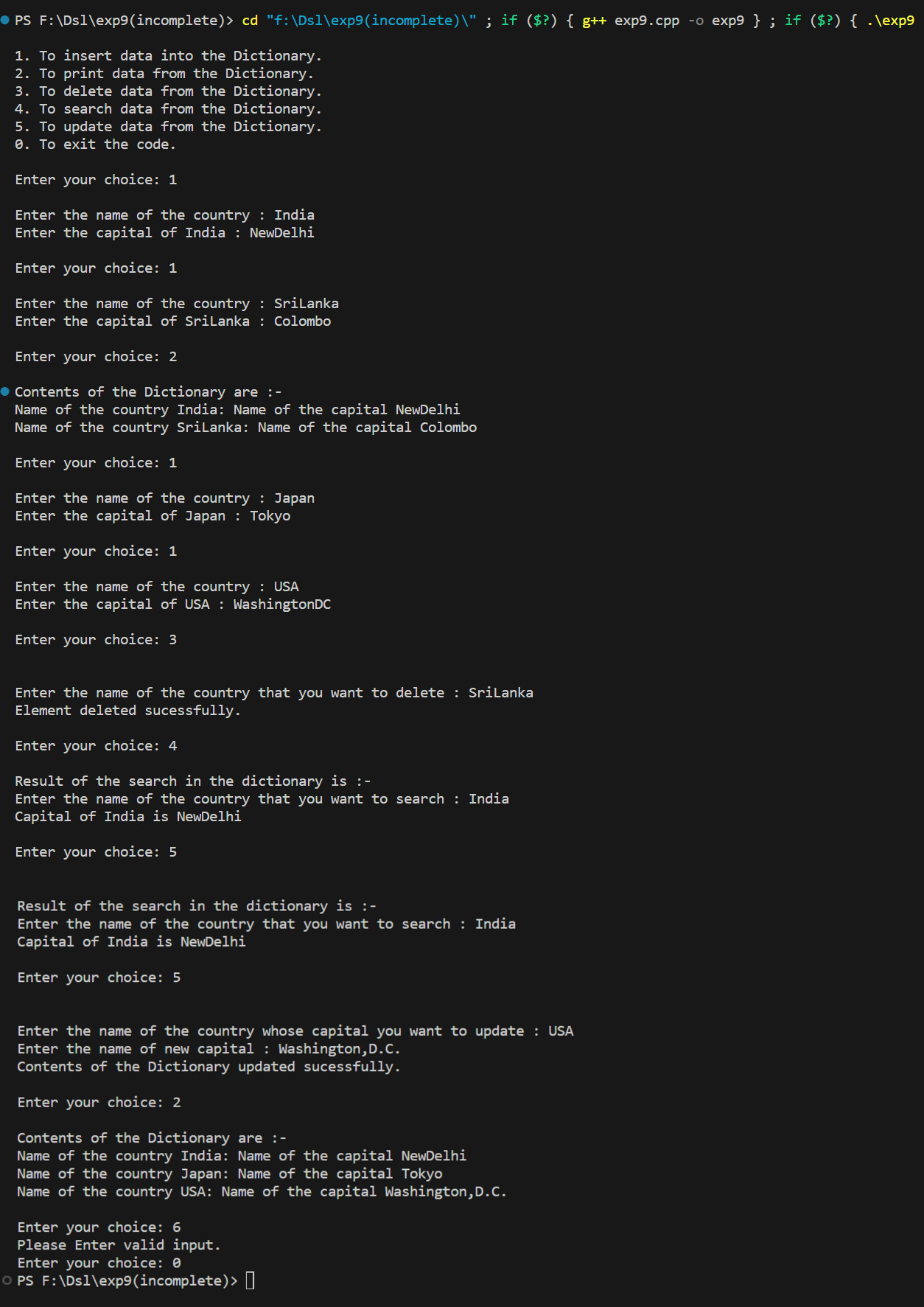
        }

    }

    return 0;

}

* **Output:**



* **Code(dijikstra):**

#include <iostream>

#include <vector>

#include <queue>

#include <climits>

using namespace std;

#define INF INT\_MAX

class Graph {

public:

    int V; // Number of vertices (cities)

    vector<vector<pair<int, int>>> adj; // Adjacency list

    Graph(int vertices) : V(vertices), adj(vertices) {}

    void addEdge(int u, int v, int weight) {

        adj[u].push\_back({v, weight});

        adj[v].push\_back({u, weight}); // For undirected graph

    }

    void dijkstra(int start) {

        vector<int> dist(V, INF);

        dist[start] = 0;

        priority\_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;

        pq.push({0, start});

        while (!pq.empty()) {

            int u = pq.top().second;

            pq.pop();

            for (const auto& neighbor : adj[u]) {

                int v = neighbor.first;

                int weight = neighbor.second;

                if (dist[u] != INF && dist[u] + weight < dist[v]) {

                    dist[v] = dist[u] + weight;

                    pq.push({dist[v], v});

                }

            }

        }

        // Print the shortest distances from the start city

        cout << "Shortest distances from city " << start << ":\n";

        for (int i = 0; i < V; ++i) {

            cout << "To city " << i << ": ";

            if (dist[i] == INF)

                cout << "INF";

            else

                cout << dist[i];

            cout << "\n";

        }

    }

};

int main() {

    int numCities, numEdges;

    cout << "Enter the number of cities: ";

    cin >> numCities;

    Graph g(numCities);

    cout << "Enter the number of edges: ";

    cin >> numEdges;

    cout << "Enter the edges and weights (format: u v weight):\n";

    for (int i = 0; i < numEdges; ++i) {

        int u, v, weight;

        cout << "Edge with weight - " << i + 1 << ": ";

        cin >> u >> v >> weight;

        g.addEdge(u, v, weight);

    }

    int startCity;

    cout << "Enter the start city: ";

    cin >> startCity;

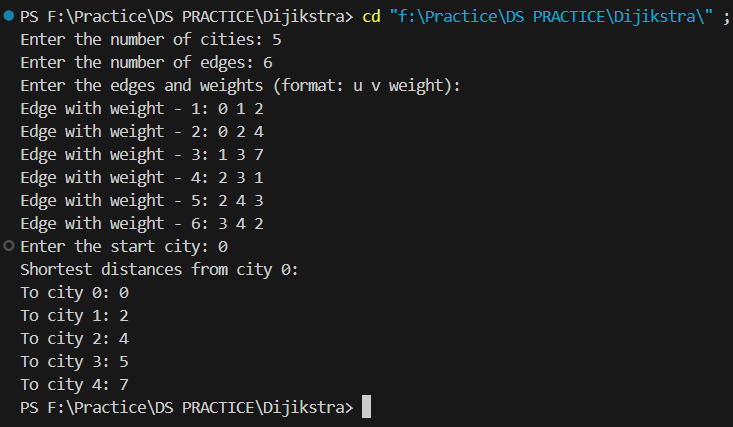
    // Run Dijkstra's algorithm

    g.dijkstra(startCity);

    return 0;

}

* **Output:**



* **Code(File Handling):**

#include <iostream>

#include <fstream>

#include <string.h>

using namespace std;

class student

{

    typedef struct stud

    {

        int roll;

        char name[10];

        char div;

        char add[10];

    } stud;

    stud rec;

public:

    void create();

    void display();

    int search();

    void Delete();

};

void student::create()

{

    char ans;

    ofstream fout;

    fout.open("stud.dat", ios::out | ios::binary);

    do

    {

        cout << "\n\tEnter Roll No of Student    : ";

        cin >> rec.roll;

        cout << "\n\tEnter a Name of Student     : ";

        cin >> rec.name;

        cout << "\n\tEnter a Division of Student : ";

        cin >> rec.div;

        cout << "\n\tEnter a Address of Student  : ";

        cin >> rec.add;

        fout.write((char \*)&rec, sizeof(stud)) << flush;

        cout << "\n\tDo You Want to Add More Records: ";

        cin >> ans;

    } while (ans == 'y' || ans == 'Y');

    fout.close();

}

void student::display()

{

    ifstream fin;

    fin.open("stud.dat", ios::in | ios::binary);

    fin.seekg(0, ios::beg);

    cout << "\n\tThe Content of File are:\n";

    cout << "\n\tRoll\tName\tDiv\tAddress";

    while (fin.read((char \*)&rec, sizeof(stud)))

    {

        if (rec.roll != -1)

            cout << "\n\t" << rec.roll << "\t" << rec.name << "\t" << rec.div << "\t" << rec.add;

    }

    fin.close();

}

int student::search()

{

    int r, i = 0;

    ifstream fin;

    fin.open("stud.dat", ios::in | ios::binary);

    fin.seekg(0, ios::beg);

    cout << "\n\tEnter a Roll No: ";

    cin >> r;

    while (fin.read((char \*)&rec, sizeof(stud)))

    {

        if (rec.roll == r)

        {

            cout << "\n\tRecord Found...\n";

            cout << "\n\tRoll\tName\tDiv\tAddress";

            cout << "\n\t" << rec.roll << "\t" << rec.name << "\t" << rec.div << "\t" << rec.add;

            return i;

        }

        i++;

    }

    fin.close();

    return 0;

}

void student::Delete()

{

    int pos;

    pos = search();

    fstream f;

    f.open("stud.dat", ios::in | ios::out | ios::binary);

    f.seekg(0, ios::beg);

    if (pos == 0)

    {

        cout << "\n\tRecord Not Found";

        return;

    }

    int offset = pos \* sizeof(stud);

    f.seekp(offset);

    rec.roll = -1;

    strcpy(rec.name, "NULL");

    rec.div = 'N';

    strcpy(rec.add, "NULL");

    f.write((char \*)&rec, sizeof(stud));

    f.seekg(0);

    f.close();

    cout << "\n\tRecord Deleted";

}

int main()

{

    student obj;

    int ch, key;

    char ans;

    do

    {

        cout << "\n\t\*\*\*\*\* Student Information \*\*\*\*\*";

        cout << "\n\t1. Create\n\t2. Display\n\t3. Delete\n\t4. Search\n\t5. Exit";

        cout << "\n\t..... Enter Your Choice: ";

        cin >> ch;

        switch (ch)

        {

        case 1:

            obj.create();

            break;

        case 2:

            obj.display();

            break;

        case 3:

            obj.Delete();

            break;

        case 4:

            key = obj.search();

            if (key == 0)

                cout << "\n\tRecord Not Found...\n";

            break;

        case 5:

            break;

        }

        cout << "\n\t..... Do You Want to Continue in Main Menu: ";

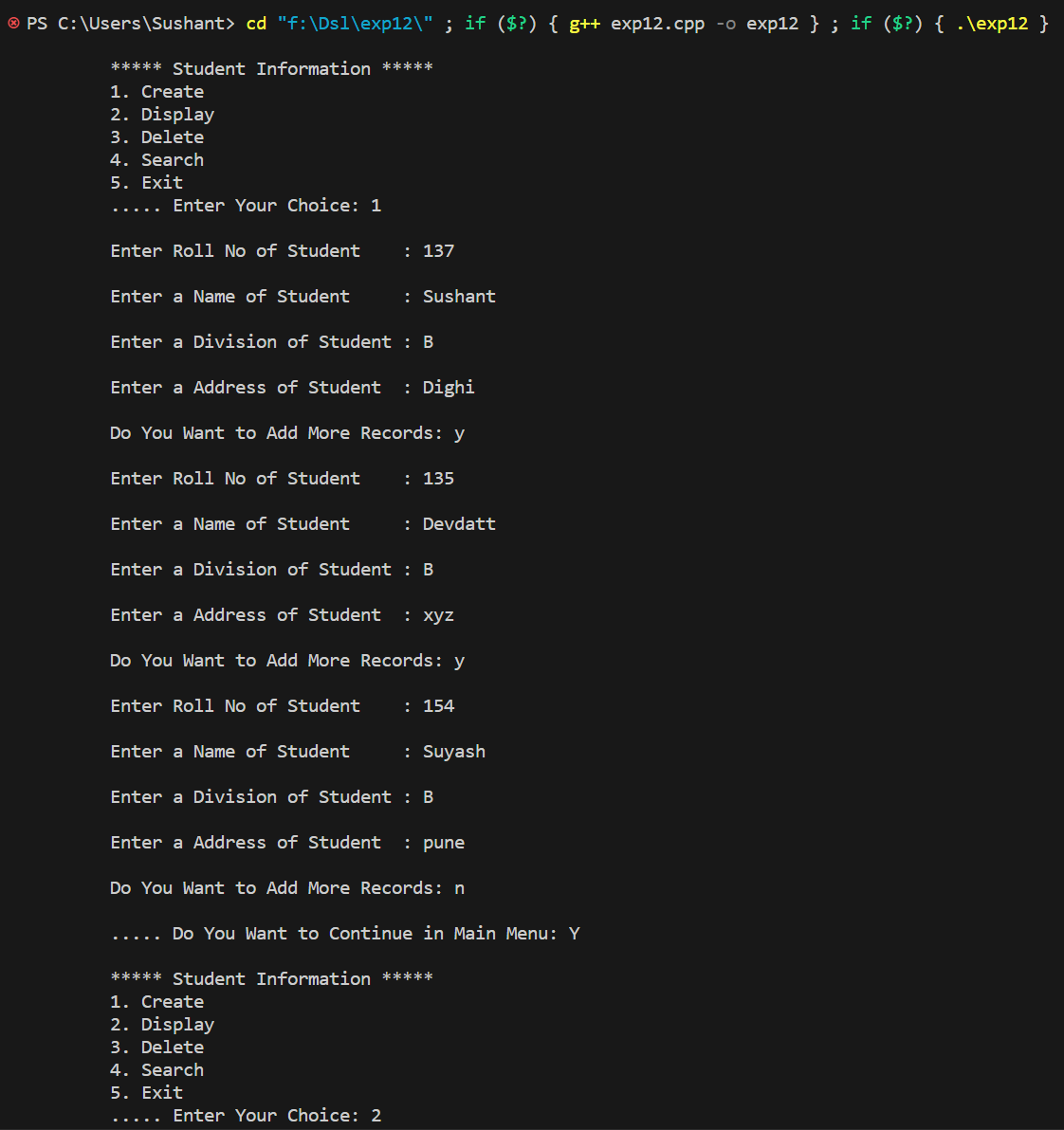
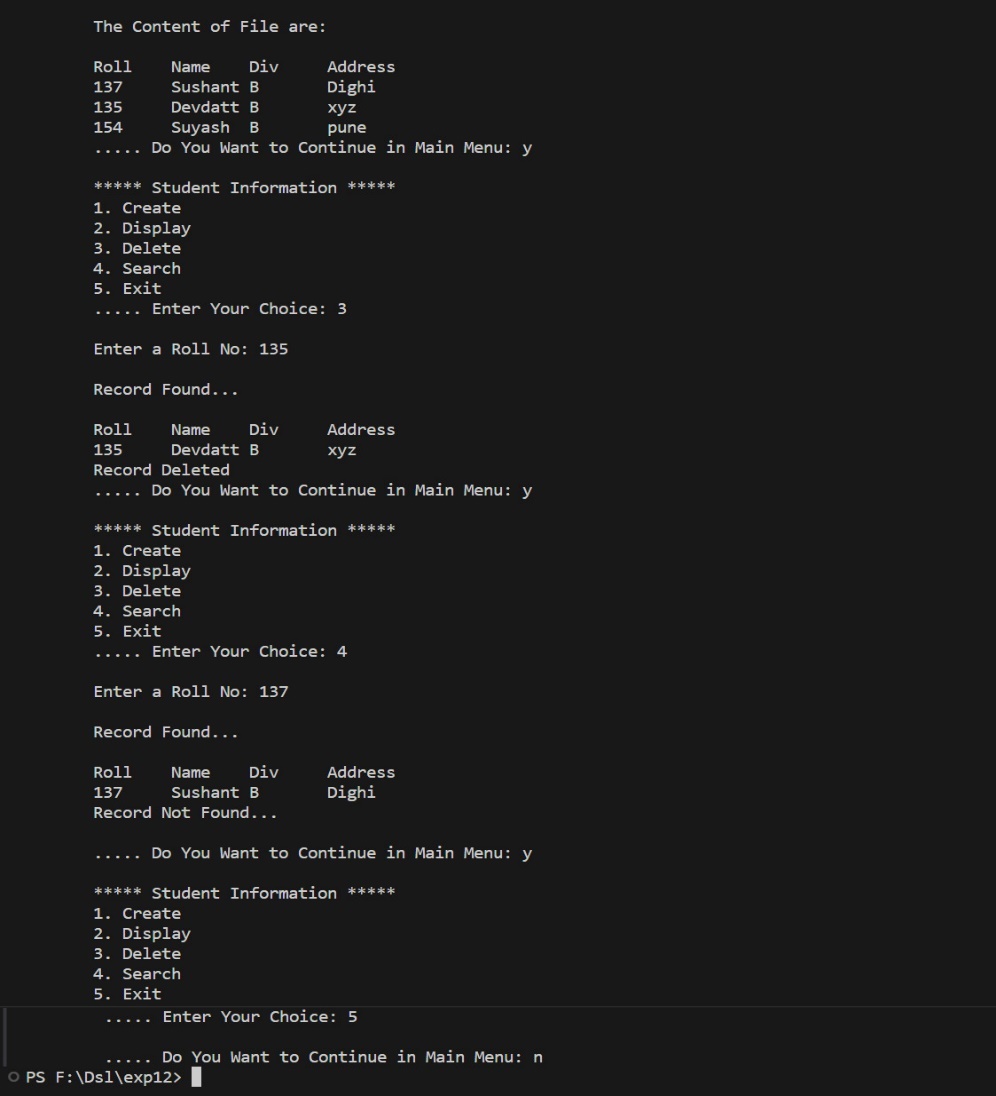
        cin >> ans;

    } while (ans == 'y' || ans == 'Y');

    return 1;

}

* **Output:**

* **Code(Linear and Binary Search):**

#include <iostream>

using namespace std;

struct Student {

    int studentID;

    string name;

    double GPA;

};

int linearSearchNonRecursive(const Student students[], int size, int targetID) {

    for (int i = 0; i < size; ++i) {

        if (students[i].studentID == targetID) {

            return i;

        }

    }

    return -1;

}

int linearSearchRecursive(const Student students[], int size, int targetID, int index = 0) {

    if (index >= size) {

        return -1;

    }

    if (students[index].studentID == targetID) {

        return index;

    }

    return linearSearchRecursive(students, size, targetID, index + 1);

}

int binarySearchNonRecursive(const Student students[], int size, int targetID) {

    int left = 0;

    int right = size - 1;

    while (left <= right) {

        int mid = left + (right - left) / 2;

        if (students[mid].studentID == targetID) {

            return mid;

        } else if (students[mid].studentID < targetID) {

            left = mid + 1;

        } else {

            right = mid - 1;

        }

    }

    return -1;

}

int binarySearchRecursive(const Student students[], int targetID, int left, int right) {

    if (left > right) {

        return -1;

    }

    int mid = left + (right - left) / 2;

    if (students[mid].studentID == targetID) {

        return mid;

    } else if (students[mid].studentID < targetID) {

        return binarySearchRecursive(students, targetID, mid + 1, right);

    } else {

        return binarySearchRecursive(students, targetID, left, mid - 1);

    }

}

int main() {

    const int arraySize = 3;

    Student studentRecords[arraySize] = {

        {101, "Sushant", 3.5},

        {204, "Luffy", 3.2},

        {309, "Zoro", 3.9}

        // Add more student records as needed

    };

    int targetID = 101;

    int linearNonRecursiveResult = linearSearchNonRecursive(studentRecords, arraySize, targetID);

    cout << "Linear Search (Non-Recursive): Student found at index " << linearNonRecursiveResult << endl;

    int linearRecursiveResult = linearSearchRecursive(studentRecords, arraySize, targetID);

    cout << "Linear Search (Recursive): Student found at index " << linearRecursiveResult << endl;

    int binaryNonRecursiveResult = binarySearchNonRecursive(studentRecords, arraySize, targetID);

    cout << "Binary Search (Non-Recursive): Student found at index " << binaryNonRecursiveResult << endl;

    int binaryRecursiveResult = binarySearchRecursive(studentRecords, targetID, 0, arraySize - 1);

    cout << "Binary Search (Recursive): Student found at index " << binaryRecursiveResult << endl;

    return 0;

}

* **Output:**

