AVL :

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

struct Node {
    int key;
    Node* left;
    Node* right;
    int height;
};

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

// Create a new node
Node* newNode(int key) {
    Node* node = new Node();
    node->key = key;
    node->left = nullptr;
    node->right = nullptr;
    node->height = 1; // New node is initially added at leaf
    return node;
}

// Get balance factor of node
int getBalance(Node* node) {
    if (node == nullptr)
        return 0;
    return height(node->left) - height(node->right);
}

// Right rotate
Node* rightRotate(Node* y) {
    Node* x = y->left;
    Node* T2 = x->right;

    // Rotation
    x->right = y;
    y->left = T2;
```

```
    // Update heights
    y->height = 1 + (height(y->left) > height(y->right) ? height(y->left) : height(y->right));
    x->height = 1 + (height(x->left) > height(x->right) ? height(x->left) : height(x->right));

    return x;
}

// Left rotate
Node* leftRotate(Node* x) {
    Node* y = x->right;
    Node* T2 = y->left;

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

    // Update heights
    x->height = 1 + (height(x->left) > height(x->right) ? height(x->left) : height(x->right));
    y->height = 1 + (height(y->left) > height(y->right) ? height(y->left) : height(y->right));

    return y;
}

// Insert a node
Node* insert(Node* node, int key) {
    // Normal BST insertion
    if (node == nullptr)
        return newNode(key);

    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);
    else // Duplicate keys not allowed
        return node;

    // Update height
    node->height = 1 + (height(node->left) > height(node->right) ? height(node->left) :
height(node->right));

    // Get balance factor
    int balance = getBalance(node);

    // Balance the node if needed
```

```
    // Left Left Case
    if (balance > 1 && key < node->left->key)
        return rightRotate(node);

    // Right Right Case
    if (balance < -1 && key > node->right->key)
        return leftRotate(node);

    // Left Right Case
    if (balance > 1 && key > node->left->key) {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    // Right Left Case
    if (balance < -1 && key < node->right->key) {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    return node;
}

// Inorder traversal
void inorder(Node* root) {
    if (root != nullptr) {
        inorder(root->left);
        cout << root->key << " ";
        inorder(root->right);
    }
}

int main() {
    Node* root = nullptr;
    int n, key;

    cout << "Enter number of nodes to insert: ";
    cin >> n;

    cout << "Enter the nodes: " << endl;
    for (int i = 0; i < n; i++) {
        cin >> key;
        root = insert(root, key);
```

```
    }

    cout << "Inorder traversal of the constructed AVL tree is:" << endl;
    inorder(root);
    cout << endl;

    return 0;
}
```

OUTPUT :

```
Enter number of nodes to insert: 5
Enter the nodes:
30
20
40
10
25

Inorder traversal of the constructed AVL tree is:
10 20 25 30 40


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

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

#define MAX_VERTICES 100
#define INF 99999 // Define a large number as infinity

void dijkstra(int graph[MAX_VERTICES][MAX_VERTICES], int V, int source) {
    int distance[MAX_VERTICES];  // To store shortest distance from source
    bool visited[MAX_VERTICES];  // To mark visited vertices

    // Initialize distances and visited array
    for (int i = 0; i < V; i++) {
        distance[i] = INF;
        visited[i] = false;
    }

    distance[source] = 0;

    for (int count = 0; count < V - 1; count++) {
        // Find the vertex with the minimum distance value
        int minDistance = INF, u = -1;

        for (int v = 0; v < V; v++) {
            if (!visited[v] && distance[v] <= minDistance) {
                minDistance = distance[v];
                u = v;
            }
        }

        // Mark the chosen vertex as visited
        visited[u] = true;

        // Update distance value of the adjacent vertices of the picked vertex
        for (int v = 0; v < V; v++) {
            if (!visited[v] && graph[u][v] && distance[u] != INF
                && distance[u] + graph[u][v] < distance[v]) {
                distance[v] = distance[u] + graph[u][v];
            }
        }
    }

    // Print the distance array
    cout << "Vertex \t Distance from Source" << endl;
    for (int i = 0; i < V; i++) {
```

```
        cout << i << " \t\t " << distance[i] << endl;
    }
}

int main() {
    int V, E;
    int graph[MAX_VERTICES][MAX_VERTICES] = {0};

    cout << "Enter number of vertices: ";
    cin >> V;
    cout << "Enter number of edges: ";
    cin >> E;

    cout << "Enter edges in format (source destination weight):" << endl;
    for (int i = 0; i < E; i++) {
        int u, v, w;
        cin >> u >> v >> w;
        graph[u][v] = w;
        graph[v][u] = w; // For undirected graph
    }

    int source;
    cout << "Enter the source vertex: ";
    cin >> source;
    dijkstra(graph, V, source);

return 0;
```

OUTPUT :

```
Enter number of vertices: 3
Enter number of edges: 4
Enter edges in format (source destination weight):
0 1 2
0 2 4
1 2 1
1 3 7

Enter the source vertex: 0
Vertex   Distance from Source
0        0
1        2
2        3


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

```
#include <iostream>
#include <fstream>
using namespace std;
struct Account {
    int acc_no;
    char name[30];
    float balance;
    void writeToFile(fstream &file) const {
        file.write((char*)&acc_no, sizeof(acc_no));
        file.write(name, sizeof(name));
        file.write((char*)&balance, sizeof(balance));
    }
    void readFromFile(fstream &file) {
        file.read((char*)&acc_no, sizeof(acc_no));
        file.read(name, sizeof(name));
        file.read((char*)&balance, sizeof(balance));
    }
    void readFromFile(ifstream &file) {
        file.read((char*)&acc_no, sizeof(acc_no));
        file.read(name, sizeof(name));
        file.read((char*)&balance, sizeof(balance));
    }
    void updateAccount() {
        cout << "Enter New Name: ";
        cin.ignore();
        cin.getline(name, 30);
        cout << "Enter New Balance: ";
        cin >> balance;
    }
    void inputAccount() {
        cout << "Enter Account Number: ";
        cin >> acc_no;
        cout << "Enter Name: ";
        cin.ignore();
        cin.getline(name, 30);
        cout << "Enter Balance: ";
        cin >> balance;
    }
    void displayAccount() const {
        cout << "Account Number: " << acc_no << "\n";
        cout << "Name: " << name << "\n";
        cout << "Balance: " << balance << "\n";
    }
};

void insertAccount() {
    Account acc;
    acc.inputAccount();
    fstream file("bank.dat", ios::in | ios::out | ios::binary);
    if (!file) {
        file.open("bank.dat", ios::out | ios::binary);  // create file if not exists
        file.close();
        file.open("bank.dat", ios::in | ios::out | ios::binary);
    }
```

```
    file.seekp(acc.acc_no * sizeof(Account), ios::beg);
    acc.writeToFile(file);
    file.close();
    cout << "Account inserted successfully.\n";
}
void updateAccount() {
    Account acc;
    int acc_no;
    cout << "Enter Account Number to Update: ";
    cin >> acc_no;
    fstream file("bank.dat", ios::in | ios::out | ios::binary);
    if (!file) {
        cout << "File not found.\n";
        return;
    }
    file.seekg(acc_no * sizeof(Account), ios::beg);
    acc.readFromFile(file);
    if (acc.acc_no != acc_no) {
        cout << "Account not found.\n";
        file.close();
        return;
    }
    cout << "Current Details:\n";
    acc.displayAccount();
    acc.updateAccount();
    file.seekp(acc_no * sizeof(Account), ios::beg);
    acc.writeToFile(file);
    file.close();
    cout << "Account updated successfully.\n";
}
void searchAccount() {
    Account acc;
    int acc_no;
    cout << "Enter Account Number to Search: ";
    cin >> acc_no;
    ifstream file("bank.dat", ios::binary);
    if (!file) {
        cout << "File not found.\n";
        return;
    }
    file.seekg(acc_no * sizeof(Account), ios::beg);
    acc.readFromFile(file);
    if (acc.acc_no != acc_no) {
        cout << "Account not found.\n";
    } else {
        cout << "Account Found:\n";
        acc.displayAccount();
    }
    file.close();
}
int main() {
    int choice;
    do {
        cout << "\nBanking System Menu:\n";
```

```
        cout << "1. Insert Account\n";
        cout << "2. Update Account\n";
        cout << "3. Search Account\n";
        cout << "4. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;
        switch(choice) {
            case 1: insertAccount(); break;
            case 2: updateAccount(); break;
            case 3: searchAccount(); break;
            case 4: cout << "Exiting...\n"; break;
            default: cout << "Invalid choice.\n";
        }
    } while(choice != 4);
    return 0;
}
```

OUTPUT :

```
Banking System Menu:
1. Insert Account
2. Update Account
3. Search Account
4. Exit
Enter your choice: 1
Enter Account Number: 2
Enter Name: Trupti
Enter Balance: 5000
Account inserted successfully.

Banking System Menu:
1. Insert Account
2. Update Account
3. Search Account
4. Exit
Enter your choice: 1
Enter Account Number: 1
```

```
Enter Account Number: 1
Enter Name: Rohan
Enter Balance: 6000
Account inserted successfully.

Banking System Menu:
1. Insert Account
2. Update Account
3. Search Account
4. Exit
```