Develop the Campus Navigation and Utility Planner
Theme: Trees and Graph Algorithms

GitHub:

https://github.com/mayank24000/University_Assignments/tree/main/Ds_%20Assignments/Lab_Assignments/Lab_Assignment_4

Code:

```
#include <iostream>
#include <string>
#include <vector>
#include <queue>
#include <stack>
#include <algorithm>
#include <climits>
#include <iomanip>

using namespace std;

class Building {
public:
    int buildingID;
    string buildingName;
    string locationDetails;
    Building() {
        buildingID = 0;
        buildingName = "";
        locationDetails = "";
    }

    Building(int id, string name, string location) {
        buildingID = id;
        buildingName = name;
        locationDetails = location;
    }

    void display() {
        cout << "ID: " << buildingID
             << ", Name: " << buildingName
             << ", Location: " << locationDetails << endl;
    }
};

// ==================== BINARY SEARCH TREE ====================

class BSTNode {
public:
    Building data;
    BSTNode* left;
    BSTNode* right;

    BSTNode(Building building) {
        data = building;
        left = nullptr;
        right = nullptr;
    }
};

class BinarySearchTree {
private:
    BSTNode* root;

    // Helper function for insertion
    BSTNode* insertHelper(BSTNode* node, Building building) {
        if (node == nullptr) {
            return new BSTNode(building);
        }

        if (building.buildingID < node->data.buildingID) {
            node->left = insertHelper(node->left, building);
        } else if (building.buildingID > node->data.buildingID) {
            node->right = insertHelper(node->right, building);
        }

        return node;
    }

    // Helper functions for traversals
    void inorderHelper(BSTNode* node) {
        if (node != nullptr) {
            inorderHelper(node->left);
            node->data.display();
            inorderHelper(node->right);
        }
    }

    void preorderHelper(BSTNode* node) {
        if (node != nullptr) {
            node->data.display();
            preorderHelper(node->left);
            preorderHelper(node->right);
        }
    }

    void postorderHelper(BSTNode* node) {
        if (node != nullptr) {
            postorderHelper(node->left);
            postorderHelper(node->right);
            node->data.display();
        }
    }

    // Helper function for search
    BSTNode* searchHelper(BSTNode* node, int id) {
        if (node == nullptr || node->data.buildingID == id) {
            return node;
        }

        if (id < node->data.buildingID) {
            return searchHelper(node->left, id);
        }

        return searchHelper(node->right, id);
    }

public:
    BinarySearchTree() {
        root = nullptr;
    }

    void insertBuilding(Building building) {
        root = insertHelper(root, building);
        cout << "Building inserted successfully in BST!" << endl;
    }

    void inorderTraversal() {
        cout << "\n--- Inorder Traversal (Sorted by ID) ---" <<
endl;
        inorderHelper(root);
    }

    void preorderTraversal() {
        cout << "\n--- Preorder Traversal ---" << endl;
        preorderHelper(root);
    }

    void postorderTraversal() {
        cout << "\n--- Postorder Traversal ---" << endl;
        postorderHelper(root);
    }

    Building* searchBuilding(int id) {
        BSTNode* result = searchHelper(root, id);
        if (result != nullptr) {
            return &(result->data);
        }
        return nullptr;
    }
};

// ==================== AVL TREE ====================

class AVLNode {
public:
    Building data;
    AVLNode* left;
    AVLNode* right;
    int height;

    AVLNode(Building building) {
        data = building;
        left = nullptr;
        right = nullptr;
        height = 1;
    }
};

class AVLTree {
```

```
private:
    AVLNode* root;

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

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

    AVLNode* rotateRight(AVLNode* y) {
        AVLNode* x = y->left;
        AVLNode* T2 = x->right;

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

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

        return x;
    }

    AVLNode* rotateLeft(AVLNode* x) {
        AVLNode* y = x->right;
        AVLNode* T2 = y->left;

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

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

        return y;
    }

    AVLNode* insertHelper(AVLNode* node, Building building) {
        // Standard BST insertion
        if (node == nullptr) {
            return new AVLNode(building);
        }

        if (building.buildingID < node->data.buildingID) {
            node->left = insertHelper(node->left, building);
        } else if (building.buildingID > node->data.buildingID) {
            node->right = insertHelper(node->right, building);
        } else {
            return node; // Duplicate keys not allowed
        }

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

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

        // Left Left Case
        if (balance > 1 && building.buildingID < node->left-
>data.buildingID) {
            return rotateRight(node);
        }

        // Right Right Case
        if (balance < -1 && building.buildingID > node->right-
>data.buildingID) {
            return rotateLeft(node);
        }

        // Left Right Case
        if (balance > 1 && building.buildingID > node->left-
>data.buildingID) {
            node->left = rotateLeft(node->left);
            return rotateRight(node);
        }

        // Right Left Case
        if (balance < -1 && building.buildingID < node->right-
>data.buildingID) {
            node->right = rotateRight(node->right);
            return rotateLeft(node);
        }

        return node;
    }

    void inorderHelper(AVLNode* node) {
        if (node != nullptr) {
            inorderHelper(node->left);
            node->data.display();
            inorderHelper(node->right);
        }
    }

public:
    AVLTree() {
        root = nullptr;
    }

    void insertBuilding(Building building) {
        root = insertHelper(root, building);
        cout << "Building inserted successfully in AVL Tree!" <<
endl;
    }

    void displayInorder() {
        cout << "\n--- AVL Tree Inorder Traversal ---" << endl;
        inorderHelper(root);
    }
};
```

```
// ==================== GRAPH REPRESENTATION ====================

class Graph {
private:
    int numBuildings;
    vector<vector<int>> adjacencyMatrix;
    vector<vector<pair<int, int>>> adjacencyList; //
pair<destination, weight>
    vector<Building> buildings;

public:
    Graph(int n) {
        numBuildings = n;
        adjacencyMatrix.resize(n, vector<int>(n, 0));
        adjacencyList.resize(n);
    }

    void addBuilding(Building building) {
        if (buildings.size() < numBuildings) {
            buildings.push_back(building);
        }
    }

    void addEdge(int src, int dest, int weight) {
        // For adjacency matrix
        adjacencyMatrix[src][dest] = weight;
        adjacencyMatrix[dest][src] = weight; // Undirected graph
        // For adjacency list
        adjacencyList[src].push_back({dest, weight});
        adjacencyList[dest].push_back({src, weight}); // Undirected
graph
    }

    void displayAdjacencyMatrix() {
        cout << "\n--- Adjacency Matrix ---" << endl;
        cout << "    ";
        for (int i = 0; i < numBuildings; i++) {
            cout << setw(4) << i;
        }
        cout << endl;

        for (int i = 0; i < numBuildings; i++) {
            cout << setw(4) << i;
            for (int j = 0; j < numBuildings; j++) {
                cout << setw(4) << adjacencyMatrix[i][j];
            }
            cout << endl;
        }
    }

    void displayAdjacencyList() {
        cout << "\n--- Adjacency List ---" << endl;
        for (int i = 0; i < numBuildings; i++) {
            cout << "Building " << i << ": ";
            for (auto& edge : adjacencyList[i]) {
                cout << "(" << edge.first << ", " << edge.second <<
") ";
            }
            cout << endl;
        }
    }

    // Dijkstra's Algorithm for shortest path
    void dijkstra(int start, int end) {
        vector<int> distance(numBuildings, INT_MAX);
        vector<int> parent(numBuildings, -1);
        vector<bool> visited(numBuildings, false);

        distance[start] = 0;

        for (int count = 0; count < numBuildings - 1; count++) {
            int minDist = INT_MAX, minIndex;

            for (int v = 0; v < numBuildings; v++) {
                if (!visited[v] && distance[v] <= minDist) {
                    minDist = distance[v];
                    minIndex = v;
                }
            }

            visited[minIndex] = true;

            for (int v = 0; v < numBuildings; v++) {
                if (!visited[v] && adjacencyMatrix[minIndex][v] &&
                    distance[minIndex] != INT_MAX &&
                    distance[minIndex] +
adjacencyMatrix[minIndex][v] < distance[v]) {
                    distance[v] = distance[minIndex] +
adjacencyMatrix[minIndex][v];
                    parent[v] = minIndex;
                }
            }
        }

        cout << "\n--- Shortest Path from Building " << start << "
to Building " << end << " ---" << endl;
        if (distance[end] == INT_MAX) {
            cout << "No path exists!" << endl;
            return;
        }

        cout << "Distance: " << distance[end] << " units" << endl;
        cout << "Path: ";

        // Reconstruct path
        vector<int> path;
        int curr = end;
        while (curr != -1) {
            path.push_back(curr);
            curr = parent[curr];
        }

        reverse(path.begin(), path.end());
        for (int i = 0; i < path.size(); i++) {
            cout << path[i];
            if (i < path.size() - 1) cout << " -> ";
        }
        cout << endl;
    }

    // Edge structure for Kruskal's algorithm
    struct Edge {
        int src, dest, weight;
        bool operator<(const Edge& other) const {
            return weight < other.weight;
        }
    };

    // Find operation for Union-Find
    int find(vector<int>& parent, int i) {
        if (parent[i] != i) {
            parent[i] = find(parent, parent[i]);
        }
        return parent[i];
    }

    // Union operation for Union-Find
    void unionSet(vector<int>& parent, vector<int>& rank, int x, int
y) {
        int xroot = find(parent, x);
        int yroot = find(parent, y);

        if (rank[xroot] < rank[yroot]) {
```

```
        parent[xroot] = yroot;
    } else if (rank[xroot] > rank[yroot]) {
        parent[yroot] = xroot;
    } else {
        parent[yroot] = xroot;
        rank[xroot]++;
    }
}

// Kruskal's Algorithm for Minimum Spanning Tree
void kruskal() {
    vector<Edge> edges;

    // Collect all edges
    for (int i = 0; i < numBuildings; i++) {
        for (int j = i + 1; j < numBuildings; j++) {
            if (adjacencyMatrix[i][j] != 0) {
                edges.push_back({i, j, adjacencyMatrix[i][j]});
            }
        }
    }

    // Sort edges by weight
    sort(edges.begin(), edges.end());

    vector<int> parent(numBuildings);
    vector<int> rank(numBuildings, 0);
    for (int i = 0; i < numBuildings; i++) {
        parent[i] = i;
        }


        cout << "\n--- Minimum Spanning Tree (Cable Layout) ---" <<
endl;
        int totalCost = 0;
        int edgesAdded = 0;


        for (const Edge& e : edges) {
            int x = find(parent, e.src);
            int y = find(parent, e.dest);


            if (x != y) {
                cout << "Connect Building " << e.src << " to
Building "
                     << e.dest << " (Cost: " << e.weight << ")" <<
endl;
                totalCost += e.weight;
                unionSet(parent, rank, x, y);
                edgesAdded++;


                if (edgesAdded == numBuildings - 1) break;
            }
        }


        cout << "Total Cable Cost: " << totalCost << " units" <<
endl;
    }
};


// ==================== EXPRESSION TREE ====================
```

```cpp
class ExprNode {
public:
    string value;
    ExprNode* left;
    ExprNode* right;

    ExprNode(string val) {
        value = val;
        left = nullptr;
        right = nullptr;
    }
};

class ExpressionTree {
private:
    ExprNode* root;

    bool isOperator(string s) {
        return (s == "+" || s == "-" || s == "*" || s == "/");
    }

    double evaluate(ExprNode* node) {
        if (node == nullptr) return 0;

        // Leaf node (operand)
        if (node->left == nullptr && node->right == nullptr) {
            return stod(node->value);
        }
        // Evaluate left and right subtrees
        double leftVal = evaluate(node->left);
        double rightVal = evaluate(node->right);

        // Apply operator
        if (node->value == "+") return leftVal + rightVal;
        if (node->value == "-") return leftVal - rightVal;
        if (node->value == "*") return leftVal * rightVal;
        if (node->value == "/") return leftVal / rightVal;

        return 0;
    }

    void inorderHelper(ExprNode* node) {
        if (node != nullptr) {
            if (isOperator(node->value)) cout << "(";
            inorderHelper(node->left);
            cout << node->value << " ";
            inorderHelper(node->right);
            if (isOperator(node->value)) cout << ")";
        }
    }

public:
    ExpressionTree() {
        root = nullptr;
    }
    // Build expression tree from postfix expression
    void buildFromPostfix(vector<string> postfix) {
        stack<ExprNode*> st;

        for (string token : postfix) {
            ExprNode* newNode = new ExprNode(token);

            if (isOperator(token)) {
                newNode->right = st.top(); st.pop();
                newNode->left = st.top(); st.pop();
            }

            st.push(newNode);
        }

        root = st.top();
    }

    double evaluateExpression() {
        return evaluate(root);
    }

    void displayInfix() {
        cout << "Infix Expression: ";
        inorderHelper(root);
        cout << endl;
    }
};
```

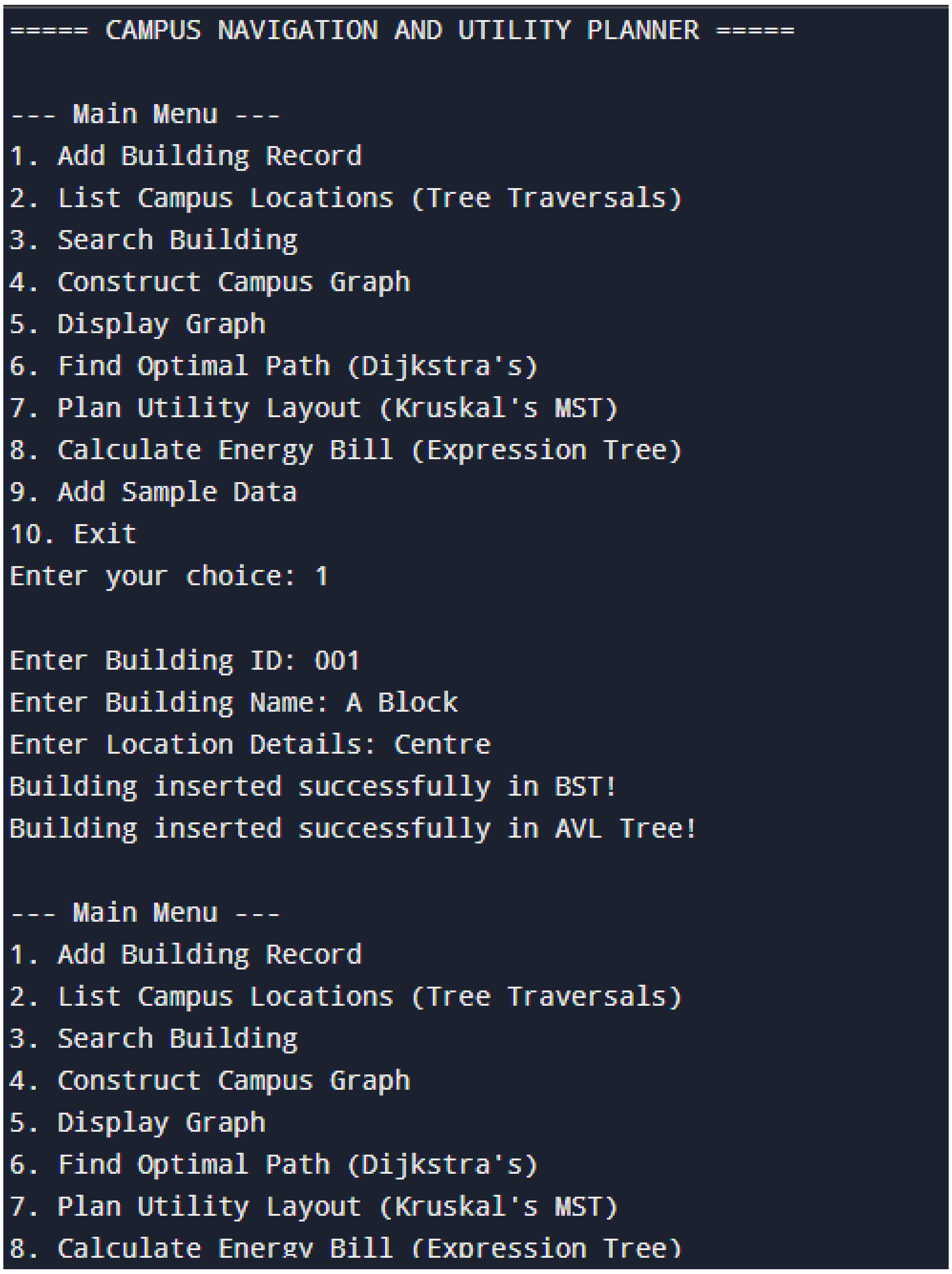
```
// ==================== CAMPUS NAVIGATION SYSTEM
====================

class CampusNavigationSystem {
private:
    BinarySearchTree bst;
    AVLTree avl;
    Graph* campusGraph;
    ExpressionTree exprTree;
    int buildingCount;

public:
    CampusNavigationSystem() {
        campusGraph = nullptr;
        buildingCount = 0;
    }

    void addBuildingRecord() {
        int id;
        string name, location;

        cout << "\nEnter Building ID: ";
        cin >> id;
        cin.ignore();

        cout << "Enter Building Name: ";
        getline(cin, name);

        cout << "Enter Location Details: ";
```

```cpp
        getline(cin, location);

        Building newBuilding(id, name, location);

        bst.insertBuilding(newBuilding);
        avl.insertBuilding(newBuilding);

        if (campusGraph != nullptr) {
            campusGraph->addBuilding(newBuilding);
        }

        buildingCount++;
    }

    void listCampusLocations() {
        cout << "\n--- Tree Traversal Options ---" << endl;
        cout << "1. Inorder Traversal (BST)" << endl;
        cout << "2. Preorder Traversal (BST)" << endl;
        cout << "3. Postorder Traversal (BST)" << endl;
        cout << "4. AVL Tree Inorder Traversal" << endl;
        cout << "Enter choice: ";

        int choice;
        cin >> choice;

        switch (choice) {
            case 1:
                bst.inorderTraversal();
                break;
            case 2:
                bst.preorderTraversal();
                break;
            case 3:
                bst.postorderTraversal();
                break;
            case 4:
                avl.displayInorder();
                break;
            default:
                cout << "Invalid choice!" << endl;
        }
    }

    void constructCampusGraph() {
        int n, edges;

        cout << "\nEnter number of buildings in graph: ";
        cin >> n;

        campusGraph = new Graph(n);

        cout << "Enter number of paths (edges): ";
        cin >> edges;

        cout << "Enter paths (source destination weight):" << endl;
        for (int i = 0; i < edges; i++) {
            int src, dest, weight;
            cin >> src >> dest >> weight;
```

```
            campusGraph->addEdge(src, dest, weight);
        }

        cout << "Campus graph constructed successfully!" << endl;
    }

    void displayGraph() {
        if (campusGraph == nullptr) {
            cout << "Please construct the campus graph first!" <<
endl;
            return;
        }

        cout << "\n--- Graph Display Options ---" << endl;
        cout << "1. Adjacency Matrix" << endl;
        cout << "2. Adjacency List" << endl;
        cout << "Enter choice: ";

        int choice;
        cin >> choice;

        switch (choice) {
            case 1:
                campusGraph->displayAdjacencyMatrix();
                break;
            case 2:
                campusGraph->displayAdjacencyList();
                break;
            default:
                cout << "Invalid choice!" << endl;
        }
    }

    void findOptimalPath() {
        if (campusGraph == nullptr) {
            cout << "Please construct the campus graph first!" <<
endl;
            return;
        }

        int start, end;
        cout << "\nEnter start building ID: ";
        cin >> start;
        cout << "Enter destination building ID: ";
        cin >> end;

        campusGraph->dijkstra(start, end);
    }

    void planUtilityLayout() {
        if (campusGraph == nullptr) {
            cout << "Please construct the campus graph first!" <<
endl;
            return;
        }

        campusGraph->kruskal();
    }
    void calculateEnergyBill() {
        cout << "\n--- Energy Bill Calculator ---" << endl;
        cout << "Enter expression in postfix notation" << endl;
        cout << "Example: For (10 + 5) * 2, enter: 10 5 + 2 *" <<
endl;
        cout << "Enter number of tokens: ";

        int n;
        cin >> n;

        vector<string> postfix(n);
        cout << "Enter tokens separated by space: ";
        for (int i = 0; i < n; i++) {
            cin >> postfix[i];
        }

        exprTree.buildFromPostfix(postfix);

        exprTree.displayInfix();
        cout << "Result: " << exprTree.evaluateExpression() << endl;
    }

    void searchBuilding() {
        int id;
        cout << "\nEnter Building ID to search: ";
        cin >> id;

        Building* result = bst.searchBuilding(id);
        if (result != nullptr) {
            cout << "Building found in BST:" << endl;
            result->display();
        } else {
            cout << "Building not found!" << endl;
        }
    }

    void addSampleData() {
        // Add sample buildings
        bst.insertBuilding(Building(1, "Main Library", "Central
Campus"));
        bst.insertBuilding(Building(2, "Engineering Block", "North
Campus"));
        bst.insertBuilding(Building(3, "Science Lab", "East
Campus"));
        bst.insertBuilding(Building(4, "Admin Building", "Central
Campus"));
        bst.insertBuilding(Building(5, "Cafeteria", "South
Campus"));

        avl.insertBuilding(Building(1, "Main Library", "Central
Campus"));
        avl.insertBuilding(Building(2, "Engineering Block", "North
Campus"));
        avl.insertBuilding(Building(3, "Science Lab", "East
Campus"));
        avl.insertBuilding(Building(4, "Admin Building", "Central
Campus"));
        avl.insertBuilding(Building(5, "Cafeteria", "South
Campus"));
```

```
        // Create sample graph
        campusGraph = new Graph(5);
        campusGraph->addEdge(0, 1, 10);
        campusGraph->addEdge(0, 3, 5);
        campusGraph->addEdge(1, 2, 8);
        campusGraph->addEdge(1, 3, 2);
        campusGraph->addEdge(2, 4, 6);
        campusGraph->addEdge(3, 4, 15);

        cout << "Sample data added successfully!" << endl;
    }
};

int main() {
    CampusNavigationSystem system;
    int choice;

    cout << "===== CAMPUS NAVIGATION AND UTILITY PLANNER =====" <<
endl;

    while (true) {
        cout << "\n--- Main Menu ---" << endl;
        cout << "1. Add Building Record" << endl;
        cout << "2. List Campus Locations (Tree Traversals)" <<
endl;
        cout << "3. Search Building" << endl;
        cout << "4. Construct Campus Graph" << endl;
        cout << "5. Display Graph" << endl;
        cout << "6. Find Optimal Path (Dijkstra's)" << endl;
        cout << "7. Plan Utility Layout (Kruskal's MST)" << endl;
        cout << "8. Calculate Energy Bill (Expression Tree)" <<
endl;
        cout << "9. Add Sample Data" << endl;
        cout << "10. Exit" << endl;
        cout << "Enter your choice: ";
        cin >> choice;

        switch (choice) {
            case 1:
                system.addBuildingRecord();
                break;
            case 2:
                system.listCampusLocations();
                break;
            case 3:
                system.searchBuilding();
                break;
            case 4:
                system.constructCampusGraph();
                break;
            case 5:
                system.displayGraph();
                break;
            case 6:
                system.findOptimalPath();
                break;
            case 7:
                system.planUtilityLayout();
                break;
            case 8:
                system.calculateEnergyBill();
                break;
            case 9:
                system.addSampleData();
                break;
            case 10:
                cout << "Thank you for using Campus Navigation
System!" << endl;
                return 0;
            default:
                cout << "Invalid choice! Please try again." << endl;
        }
    }

    return 0;
}
```

OUTPUT:



```
===== CAMPUS NAVIGATION AND UTILITY PLANNER =====

--- Main Menu ---
1. Add Building Record
2. List Campus Locations (Tree Traversals)
3. Search Building
4. Construct Campus Graph
5. Display Graph
6. Find Optimal Path (Dijkstra's)
7. Plan Utility Layout (Kruskal's MST)
8. Calculate Energy Bill (Expression Tree)
9. Add Sample Data
10. Exit
Enter your choice: 1

Enter Building ID: 001
Enter Building Name: A Block
Enter Location Details: Centre
Building inserted successfully in BST!
Building inserted successfully in AVL Tree!

--- Main Menu ---
1. Add Building Record
2. List Campus Locations (Tree Traversals)
3. Search Building
4. Construct Campus Graph
5. Display Graph
6. Find Optimal Path (Dijkstra's)
7. Plan Utility Layout (Kruskal's MST)
8. Calculate Energy Bill (Expression Tree)
```