In [1]:
```python
from collections import deque
import heapq

class Building:
    def __init__(self, building_id, building_name, location_details):
        self.building_id = building_id
        self.building_name = building_name
        self.location_details = location_details

    def __str__(self):
        return f"[ID={self.building_id}, Name={self.building_name}, Location={self.

    __repr__ = __str__


class BSTNode:
    def __init__(self, building):
        self.building = building
        self.left = None
        self.right = None


class BuildingBST:
    def __init__(self):
        self.root = None

    def insert_building(self, building):
        new_node = BSTNode(building)
        if self.root is None:
            self.root = new_node
        else:
            self._insert_recursive(self.root, new_node)

    def _insert_recursive(self, current_node, new_node):
        if new_node.building.building_id < current_node.building.building_id:
            if current_node.left is None:
                current_node.left = new_node
            else:
                self._insert_recursive(current_node.left, new_node)
        else:
            if current_node.right is None:
                current_node.right = new_node
            else:
                self._insert_recursive(current_node.right, new_node)

    def search_building(self, building_id):
        return self._search_recursive(self.root, building_id)

    def _search_recursive(self, current_node, building_id):
        if current_node is None:
            return None
        if building_id == current_node.building.building_id:
            return current_node.building
        elif building_id < current_node.building.building_id:
```

```
            return self._search_recursive(current_node.left, building_id)
        else:
            return self._search_recursive(current_node.right, building_id)

    def inorder_traversal(self):
        result = []
        self._inorder(self.root, result)
        return result

    def _inorder(self, node, result):
        if node is not None:
            self._inorder(node.left, result)
            result.append(node.building)
            self._inorder(node.right, result)

    def preorder_traversal(self):
        result = []
        self._preorder(self.root, result)
        return result

    def _preorder(self, node, result):
        if node is not None:
            result.append(node.building)
            self._preorder(node.left, result)
            self._preorder(node.right, result)

    def postorder_traversal(self):
        result = []
        self._postorder(self.root, result)
        return result

    def _postorder(self, node, result):
        if node is not None:
            self._postorder(node.left, result)
            self._postorder(node.right, result)
            result.append(node.building)

    def get_height(self):
        return self._get_height_recursive(self.root)

    def _get_height_recursive(self, node):
        if node is None:
            return 0
        left_height = self._get_height_recursive(node.left)
        right_height = self._get_height_recursive(node.right)
        return 1 + max(left_height, right_height)


class AVLNode:
    def __init__(self, building):
        self.building = building
        self.left = None
        self.right = None
        self.height = 1
```

```
class BuildingAVL:
    def __init__(self):
        self.root = None

    def get_height(self, node):
        if node is None:
            return 0
        return node.height

    def get_balance_factor(self, node):
        if node is None:
            return 0
        return self.get_height(node.left) - self.get_height(node.right)

    def right_rotation(self, y):
        x = y.left
        t2 = x.right
        x.right = y
        y.left = t2
        y.height = 1 + max(self.get_height(y.left), self.get_height(y.right))
        x.height = 1 + max(self.get_height(x.left), self.get_height(x.right))
        return x

    def left_rotation(self, x):
        y = x.right
        t2 = y.left
        y.left = x
        x.right = t2
        x.height = 1 + max(self.get_height(x.left), self.get_height(x.right))
        y.height = 1 + max(self.get_height(y.left), self.get_height(y.right))
        return y

    def insert_building(self, building):
        self.root = self._insert_recursive(self.root, building)

    def _insert_recursive(self, node, building):
        if node is None:
            return AVLNode(building)
        if building.building_id < node.building.building_id:
            node.left = self._insert_recursive(node.left, building)
        else:
            node.right = self._insert_recursive(node.right, building)

        node.height = 1 + max(self.get_height(node.left), self.get_height(node.righ
        balance = self.get_balance_factor(node)

        if balance > 1 and building.building_id < node.left.building.building_id:
            return self.right_rotation(node)
        if balance < -1 and building.building_id > node.right.building.building_id:
            return self.left_rotation(node)
        if balance > 1 and building.building_id > node.left.building.building_id:
            node.left = self.left_rotation(node.left)
            return self.right_rotation(node)
        if balance < -1 and building.building_id < node.right.building.building_id:
            node.right = self.right_rotation(node.right)
            return self.left_rotation(node)
```

```
        return node

    def search_building(self, building_id):
        return self._search_recursive(self.root, building_id)

    def _search_recursive(self, node, building_id):
        if node is None:
            return None
        if building_id == node.building.building_id:
            return node.building
        elif building_id < node.building.building_id:
            return self._search_recursive(node.left, building_id)
        else:
            return self._search_recursive(node.right, building_id)

    def inorder_traversal(self):
        result = []
        self._inorder(self.root, result)
        return result

    def _inorder(self, node, result):
        if node is not None:
            self._inorder(node.left, result)
            result.append(node.building)
            self._inorder(node.right, result)

    def preorder_traversal(self):
        result = []
        self._preorder(self.root, result)
        return result

    def _preorder(self, node, result):
        if node is not None:
            result.append(node.building)
            self._preorder(node.left, result)
            self._preorder(node.right, result)

    def postorder_traversal(self):
        result = []
        self._postorder(self.root, result)
        return result

    def _postorder(self, node, result):
        if node is not None:
            self._postorder(node.left, result)
            self._postorder(node.right, result)
            result.append(node.building)

    def get_tree_height(self):
        return self.get_height(self.root)


class CampusGraph:
    def __init__(self):
        self.building_id_to_index = {}
```

```
        self.index_to_building_id = []
        self.adjacency_matrix = []
        self.adjacency_list = {}

    def _ensure_building_exists(self, building_id):
        if building_id in self.building_id_to_index:
            return
        new_index = len(self.index_to_building_id)
        self.building_id_to_index[building_id] = new_index
        self.index_to_building_id.append(building_id)
        # pad existing rows and add a new row
        for row in self.adjacency_matrix:
            row.append(0)
        self.adjacency_matrix.append([0] * (new_index + 1))
        self.adjacency_list[building_id] = []

    def add_path(self, from_building_id, to_building_id, distance):
        self._ensure_building_exists(from_building_id)
        self._ensure_building_exists(to_building_id)
        u = self.building_id_to_index[from_building_id]
        v = self.building_id_to_index[to_building_id]
        # ensure matrix is square & sized correctly
        n = len(self.index_to_building_id)
        for row in self.adjacency_matrix:
            if len(row) < n:
                row.extend([0] * (n - len(row)))
        if len(self.adjacency_matrix) < n:
            self.adjacency_matrix.extend([[0] * n for _ in range(n - len(self.adjac

        self.adjacency_matrix[u][v] = distance
        self.adjacency_matrix[v][u] = distance
        self.adjacency_list[from_building_id].append((to_building_id, distance))
        self.adjacency_list[to_building_id].append((from_building_id, distance))

    def bfs(self, start_building_id):
        visited = set()
        order = []
        queue = deque()
        queue.append(start_building_id)
        visited.add(start_building_id)
        while queue:
            current = queue.popleft()
            order.append(current)
            for neighbor, _ in self.adjacency_list.get(current, []):
                if neighbor not in visited:
                    visited.add(neighbor)
                    queue.append(neighbor)
        return order

    def dfs(self, start_building_id):
        visited = set()
        order = []
        if start_building_id in self.adjacency_list:
            self._dfs_recursive(start_building_id, visited, order)
        return order
```

```
    def _dfs_recursive(self, current_id, visited, order):
        visited.add(current_id)
        order.append(current_id)
        for neighbor, _ in self.adjacency_list.get(current_id, []):
            if neighbor not in visited:
                self._dfs_recursive(neighbor, visited, order)

    def dijkstra(self, start_building_id):
        distances = {bid: float('inf') for bid in self.adjacency_list}
        if start_building_id not in distances:
            return distances
        distances[start_building_id] = 0
        pq = [(0, start_building_id)]
        while pq:
            current_distance, current_id = heapq.heappop(pq)
            if current_distance > distances[current_id]:
                continue
            for neighbor, w in self.adjacency_list.get(current_id, []):
                nd = current_distance + w
                if nd < distances[neighbor]:
                    distances[neighbor] = nd
                    heapq.heappush(pq, (nd, neighbor))
        return distances

    def dijkstra_path(self, start, end):
        if start not in self.adjacency_list or end not in self.adjacency_list:
            return float('inf'), []
        distances = {bid: float('inf') for bid in self.adjacency_list}
        prev = {bid: None for bid in self.adjacency_list}
        distances[start] = 0
        pq = [(0, start)]
        while pq:
            current_distance, current = heapq.heappop(pq)
            if current == end:
                break
            if current_distance > distances[current]:
                continue
            for neighbor, w in self.adjacency_list.get(current, []):
                nd = current_distance + w
                if nd < distances[neighbor]:
                    distances[neighbor] = nd
                    prev[neighbor] = current
                    heapq.heappush(pq, (nd, neighbor))
        if distances[end] == float('inf'):
            return float('inf'), []
        path = []
        cur = end
        while cur is not None:
            path.append(cur)
            cur = prev[cur]
        path.reverse()
        return distances[end], path

    def kruskal_mst(self):
        edges = []
        seen = set()
```

```
        for u in self.adjacency_list:
            for v, w in self.adjacency_list[u]:
                if (v, u) not in seen:
                    edges.append((w, u, v))
                    seen.add((u, v))
        edges.sort()
        parent = {}
        rank = {}

        def find(x):
            if parent[x] != x:
                parent[x] = find(parent[x])
            return parent[x]

        def union(x, y):
            rx = find(x)
            ry = find(y)
            if rx == ry:
                return False
            if rank[rx] < rank[ry]:
                parent[rx] = ry
            elif rank[rx] > rank[ry]:
                parent[ry] = rx
            else:
                parent[ry] = rx
                rank[rx] += 1
            return True

        for bid in self.adjacency_list:
            parent[bid] = bid
            rank[bid] = 0
        mst = []
        for w, u, v in edges:
            if union(u, v):
                mst.append((u, v, w))
        return mst


class ExpressionNode:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None


class ExpressionTree:
    def __init__(self):
        self.root = None

    def build_from_postfix(self, tokens):
        stack = []
        ops = {'+', '-', '*', '/'}
        for t in tokens:
            if t not in ops:
                stack.append(ExpressionNode(t))
            else:
```

```
                r = stack.pop()
                l = stack.pop()
                node = ExpressionNode(t)
                node.left = l
                node.right = r
                stack.append(node)
        self.root = stack.pop() if stack else None

    def evaluate(self):
        return self._eval(self.root)

    def _eval(self, node):
        if node.left is None and node.right is None:
            return float(node.value)
        a = self._eval(node.left)
        b = self._eval(node.right)
        if node.value == '+':
            return a + b
        if node.value == '-':
            return a - b
        if node.value == '*':
            return a * b
        if node.value == '/':
            return a / b
        return 0.0


class CampusNavigationAndUtilityPlanner:
    def __init__(self, use_avl_tree=False):
        self.building_tree = BuildingAVL() if use_avl_tree else BuildingBST()
        self.campus_graph = CampusGraph()
        self.expression_tree = ExpressionTree()

    def addBuildingRecord(self, building_id, building_name, location_details):
        b = Building(building_id, building_name, location_details)
        self.building_tree.insert_building(b)
        self.campus_graph._ensure_building_exists(building_id)

    def listCampusLocations(self):
        print("\nInorder:")
        for b in self.building_tree.inorder_traversal():
            print(b)
        print("\nPreorder:")
        for b in self.building_tree.preorder_traversal():
            print(b)
        print("\nPostorder:")
        for b in self.building_tree.postorder_traversal():
            print(b)

    def constructCampusGraph(self, edges):
        for u, v, d in edges:
            self.campus_graph.add_path(u, v, d)

    def showGraphTraversals(self, start):
        print("\nBFS")
        print(" -> ".join(str(x) for x in self.campus_graph.bfs(start)))
```

```
        print("\nDFS")
        print(" -> ".join(str(x) for x in self.campus_graph.dfs(start)))

    def findOptimalPath(self, start, end):
        dist, path = self.campus_graph.dijkstra_path(start, end)
        if dist == float('inf'):
            print(f"\nNo path between {start} and {end}")
        else:
            print("\nPath", " -> ".join(str(x) for x in path))
            print("Distance:", dist)

    def planUtilityLayout(self):
        mst = self.campus_graph.kruskal_mst()
        print("\nEdges in MST:")
        total = 0
        for u, v, w in mst:
            print(f"{u} -- {v} ({w})")
            total += w
        print("Total cost:", total)

    def evaluateExpression(self, postfix_tokens):
        self.expression_tree.build_from_postfix(postfix_tokens)
        return self.expression_tree.evaluate()


if __name__ == "__main__":
    planner = CampusNavigationAndUtilityPlanner(use_avl_tree=False)
    planner.addBuildingRecord(10, "Library", "North wing")
    planner.addBuildingRecord(5, "Cafeteria", "Ground floor")
    planner.addBuildingRecord(15, "Lab", "East block")
    planner.addBuildingRecord(12, "Admin", "Main block")
    planner.addBuildingRecord(18, "Gym", "South")
    planner.listCampusLocations()
    edges = [(10, 5, 7), (10, 15, 10), (5, 12, 4), (12, 15, 3), (15, 18, 6)]
    planner.constructCampusGraph(edges)
    planner.showGraphTraversals(10)
    planner.findOptimalPath(5, 18)
    planner.planUtilityLayout()
    val = planner.evaluateExpression(["3", "4", "+", "2", "*", "7", "/"])
    print("\nExpression value:", val)
```

```
Inorder:
[ID=5, Name=Cafeteria, Location=Ground floor]
[ID=10, Name=Library, Location=North wing]
[ID=12, Name=Admin, Location=Main block]
[ID=15, Name=Lab, Location=East block]
[ID=18, Name=Gym, Location=South]

Preorder:
[ID=10, Name=Library, Location=North wing]
[ID=5, Name=Cafeteria, Location=Ground floor]
[ID=15, Name=Lab, Location=East block]
[ID=12, Name=Admin, Location=Main block]
[ID=18, Name=Gym, Location=South]

Postorder:
[ID=5, Name=Cafeteria, Location=Ground floor]
[ID=12, Name=Admin, Location=Main block]
[ID=18, Name=Gym, Location=South]
[ID=15, Name=Lab, Location=East block]
[ID=10, Name=Library, Location=North wing]

BFS
10 -> 5 -> 15 -> 12 -> 18

DFS
10 -> 5 -> 12 -> 15 -> 18

Path 5 -> 12 -> 15 -> 18
Distance: 13

Edges in MST:
15 -- 12 (3)
5 -- 12 (4)
15 -- 18 (6)
10 -- 5 (7)
Total cost: 20

Expression value: 2.0
```

In [ ]: