DATA STRUCTURES

ASSIGNMENT-3

SUBMITTED By – Kunal

ROLL NO - 2401420055

PROGRAM - B.Tech CSE (Ds)

SEMESTER - III

SUBMITTED TO - Ms. Swati

Code

```
from dataclasses import dataclass
from typing import Optional, List, Tuple
import heapq

#  BUILDING
@dataclass
class Building:
    id: int
    name: str
    details: str
    def __str__(self):
        return f"[{self.id}] {self.name} - {self.details}"

#  BST (Binary Search Tree)
class BSTNode:
    def __init__(self, b: Building):
        self.b = b
        self.left = None
        self.right = None

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

    def insert(self, b):
        if self.root is None:
            self.root = BSTNode(b)
            return

        cur = self.root
        while True:
            if b.id < cur.b.id:
                if cur.left is None:
                    cur.left = BSTNode(b)
                    return
                cur = cur.left
            elif b.id > cur.b.id:
                if cur.right is None:
                    cur.right = BSTNode(b)
                    return
                cur = cur.right
            else:
                return  # ignore duplicates

    def inorder(self):
        res = []
        def dfs(n):
            if not n: return
            dfs(n.left)
            res.append(n.b)
            dfs(n.right)
        dfs(self.root)
        return res

    def search(self, id_):
        cur = self.root
        while cur:
            if id_ == cur.b.id:
                return cur.b
            cur = cur.left if id_ < cur.b.id else cur.right
        return None

    def height(self):
        def h(n):
            if not n: return 0
            return 1 + max(h(n.left), h(n.right))
        return h(self.root)

#  AVL TREE

class AVLNode:
    def __init__(self, b):
        self.b = b
        self.left = None
        self.right = None
        self.h = 1

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

    def height(self, n):
        return n.h if n else 0

    def update(self, n):
        n.h = 1 + max(self.height(n.left), self.height(n.right))

    def rotate_left(self, x):
        y = x.right
        T = y.left
        y.left = x
        x.right = T
        self.update(x)
        self.update(y)
        return y

    def rotate_right(self, y):
        x = y.left
        T = x.right
        x.right = y
        y.left = T
        self.update(y)
        self.update(x)
        return x

    def balance_factor(self, n):
        return self.height(n.left) - self.height(n.right)

    def insert_rec(self, node, b):
        if not node:
            return AVLNode(b)

        if b.id < node.b.id:
            node.left = self.insert_rec(node.left, b)
        elif b.id > node.b.id:
            node.right = self.insert_rec(node.right, b)
        else:
            return node

        self.update(node)
        bf = self.balance_factor(node)

        # LL
        if bf > 1 and b.id < node.left.b.id:
            return self.rotate_right(node)

        # RR
        if bf < -1 and b.id > node.right.b.id:
            return self.rotate_left(node)

        # LR
        if bf > 1 and b.id > node.left.b.id:
            node.left = self.rotate_left(node.left)
            return self.rotate_right(node)

        # RL
        if bf < -1 and b.id < node.right.b.id:
            node.right = self.rotate_right(node.right)
            return self.rotate_left(node)

        return node

    def insert(self, b):
        self.root = self.insert_rec(self.root, b)

    def inorder(self):
        res = []
        def dfs(n):
            if not n: return
            dfs(n.left)
            res.append(n.b)
            dfs(n.right)
        dfs(self.root)
        return res


#  GRAPH (Adjacency List)

class Graph:
    def __init__(self, n):
        self.n = n
        self.adj = [[] for _ in range(n)]

    def add_edge(self, u, v, w, undirected=True):
        self.adj[u].append((v, w))
        if undirected:
            self.adj[v].append((u, w))

    def bfs(self, s):
        vis = [False] * self.n
        q = [s]
        vis[s] = True
        order = []
        while q:
            u = q.pop(0)
            order.append(u)
            for v, _ in self.adj[u]:
                if not vis[v]:
                    vis[v] = True
                    q.append(v)

        return order

    def dfs(self, s):
        vis = [False] * self.n
        order = []

        def rec(u):
            vis[u] = True
            order.append(u)
            for v, _ in self.adj[u]:
                if not vis[v]:
                    rec(v)

        rec(s)
        return order

    def dijkstra(self, s):
        dist = [float('inf')] * self.n
        dist[s] = 0
        parent = [-1] * self.n
        pq = [(0, s)]

        while pq:
            d, u = heapq.heappop(pq)
            if d > dist[u]: continue

            for v, w in self.adj[u]:
                if dist[v] > dist[u] + w:
                    dist[v] = dist[u] + w
                    parent[v] = u
                    heapq.heappush(pq, (dist[v], v))

        return dist, parent

    def kruskal(self):
        edges = []
        for u in range(self.n):
            for v, w in self.adj[u]:
                if u < v:
                    edges.append((w, u, v))

        edges.sort()
        parent = list(range(self.n))
        rank = [0] * self.n

        def find(x):
            if parent[x] != x:
                parent[x] = find(parent[x])
            return parent[x]
        def union(a, b):
            a = find(a)
            b = find(b)
            if a == b: return False
            if rank[a] < rank[b]: parent[a] = b
            else:
                parent[b] = a
                if rank[a] == rank[b]: rank[a] += 1
            return True

        mst = []
        for w, u, v in edges:
            if union(u, v):
                mst.append((u, v, w))

        return mst


#  EXPRESSION TREE

class ExprNode:
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None

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

    def build(self, tokens):
        st = []
        for t in tokens:
            if t in "+-*/^":
                r = st.pop()
                l = st.pop()
                n = ExprNode(t)
                n.left = l
                n.right = r
                st.append(n)
            else:
                st.append(ExprNode(t))
        self.root = st[-1]

    def eval(self):
        def rec(n):
            if n.val not in "+-*/^":
                return float(n.val)
            L = rec(n.left)
            R = rec(n.right)
            if n.val == "+": return L + R
            if n.val == "-": return L - R
            if n.val == "*": return L * R
            if n.val == "/": return L / R
            return L ** R
        return rec(self.root)


#  DEMO FUNCTION
```

```python
def demo():
    buildings = [
        Building(101, "Admin", "Administrative block"),
        Building(102, "Library", "Library building"),
        Building(103, "CSE", "Computer Science Dept"),
        Building(104, "ECE", "Electronics Dept"),
        Building(105, "Cafeteria", "Student canteen"),
        Building(106, "Gym", "Sports complex"),
    ]

    id_to_i = {b.id: i for i, b in enumerate(buildings)}

    print("=== BST & AVL ===")
    bst = BST()
    avl = AVL()

    for b in buildings:
        bst.insert(b)
        avl.insert(b)

    print("BST inorder:")
    for b in bst.inorder(): print(" ", b)

    print("AVL inorder:")
    for b in avl.inorder(): print(" ", b)

    print("BST height:", bst.height())
    print("AVL height:", avl.height())

    print("\n=== GRAPH ===")
    g = Graph(len(buildings))
    g.add_edge(id_to_i[101], id_to_i[102], 5)
    g.add_edge(id_to_i[101], id_to_i[103], 10)
    g.add_edge(id_to_i[103], id_to_i[104], 6)
    g.add_edge(id_to_i[103], id_to_i[105], 4)
    g.add_edge(id_to_i[105], id_to_i[106], 3)

    print("BFS:", g.bfs(id_to_i[101]))
    print("DFS:", g.dfs(id_to_i[101]))

    dist, parent = g.dijkstra(id_to_i[101])
    print("Dijkstra:", dist)

    print("MST:", g.kruskal())

    print("\n=== Expression Tree ===")
    et = ExpressionTree()
    et.build(["100", "0.18", "*", "50", "+", "1.12", "*"])
    print("Energy Bill:", et.eval())


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

Introduction

This report presents the complete implementation of a Campus Planner System using Python. The project integrates key Data Structure concepts, including Binary Search Trees (BST), AVL Trees, Graph Data Structures, and Expression Trees. Each structure is implemented from scratch without external libraries (except built-in ones), following academic standards for a Data Structures course.

The project simulates a college campus consisting of multiple buildings connected through paths. Various algorithms like BFS, DFS, Dijkstra's shortest path, and Kruskal's Minimum Spanning Tree (MST) are used to analyze and compute meaningful results such as traversal order, shortest path, and minimum cabling cost.

1. Binary Search Tree (BST)

The Binary Search Tree stores campus buildings sorted by ID. Each node contains: Building ID Building Name Building Details **Operations Implemented:** Insertion Search Inorder traversal (sorted output) Height calculation **Time Complexity:** Insertion: O(h) where h is tree height Search: O(h) Traversal: O(n) BST may become skewed (like a linked list) in the worst case, resulting in height O(n).

2. AVL Tree

The AVL tree improves upon BST by ensuring self-balancing using rotations: **LL Rotation RR Rotation LR Rotation RL Rotation** AVL maintains the balance factor (difference between left and right subtree heights).

Advantages: Height guaranteed to be O(log n) Search and insert remain efficient

3. Graph Structure

The campus is represented as an undirected weighted graph. Each node represents a building, and edges represent paths with distances.

Graph Representations: Adjacency List (Primary) Adjacency Matrix (for understanding; optional) **Algorithms Implemented: Breadth-First Search (BFS):** explores level by level **Depth-First Search (DFS):** explores deep paths first **Dijkstra's Algorithm:** finds shortest path between buildings **Kruskal's MST Algorithm:** computes minimum cost wiring/cabling

4. Expression Tree

The Expression Tree evaluates arithmetic expressions given in postfix notation. Used here for an example calculation of an energy bill.

Supported Operations: Addition (+) Subtraction (-) Multiplication (*) Division (/) Exponent (^) The tree is evaluated recursively. Every operator node has two children, while numeric values are leaf nodes.

Sample Output Results

The program prints: BST inorder traversal (sorted building IDs) AVL inorder traversal BFS traversal from the Admin building DFS traversal Dijkstra's shortest path distances MST edges with total minimum cost Expression Tree evaluation result These outputs confirm that all data structures and algorithms function correctly.

Conclusion

This project successfully demonstrates fundamental and advanced data structures used in computer science. It integrates trees and graph algorithms into a practical scenario of a campus navigation and planning system. The structured approach to building and evaluating these data structures strengthens problem-solving and algorithmic understanding essential for computer science engineering.