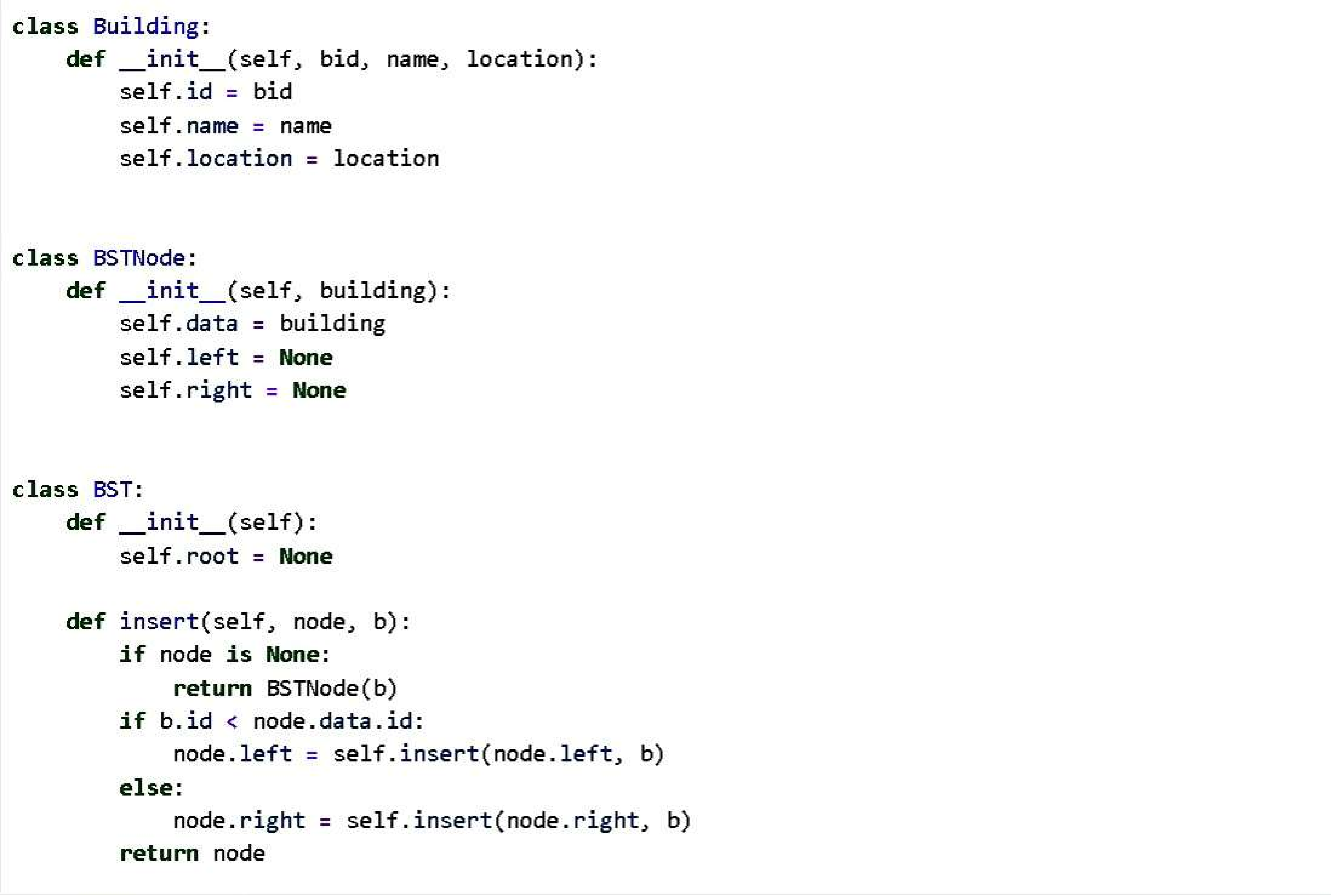
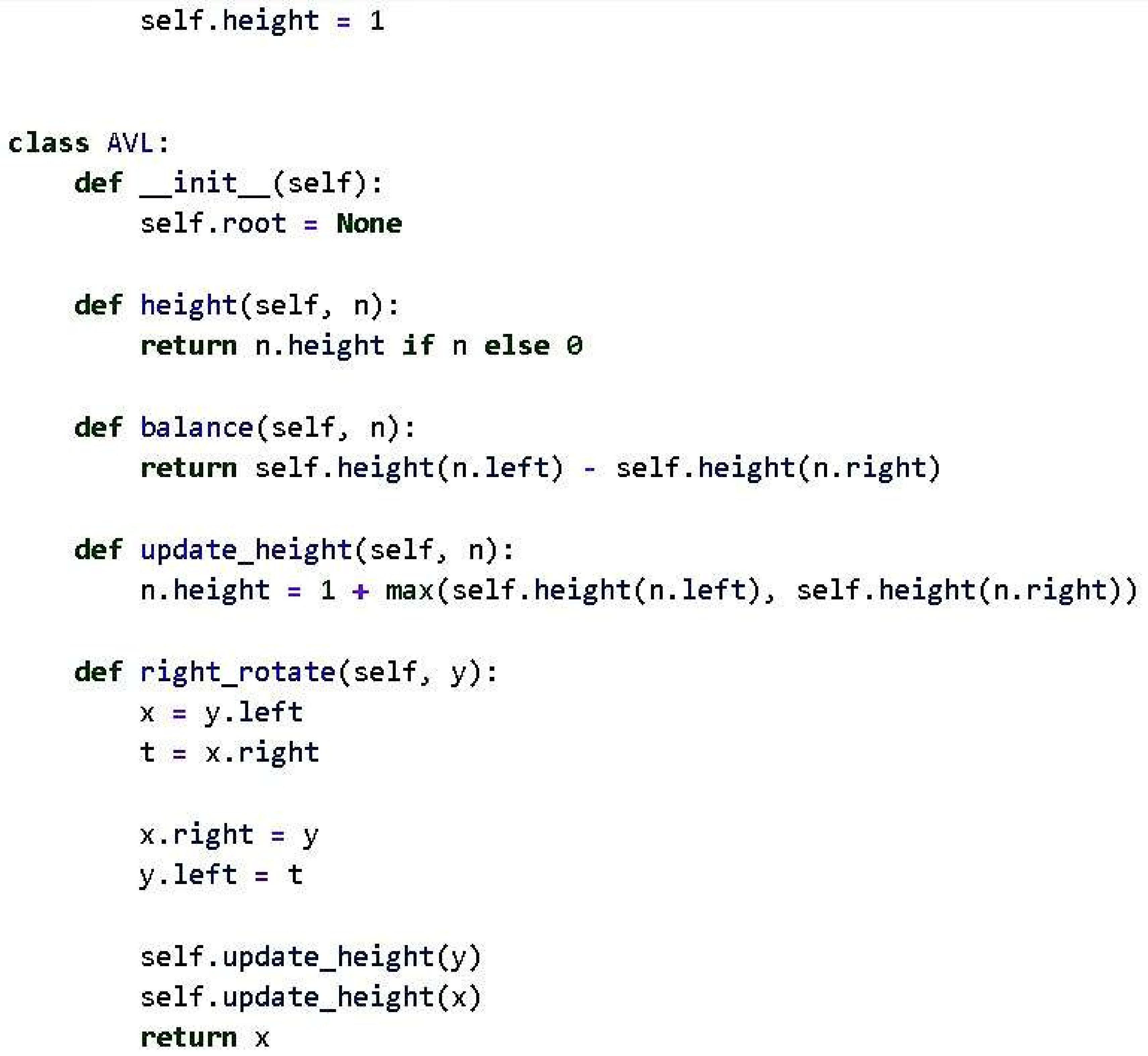
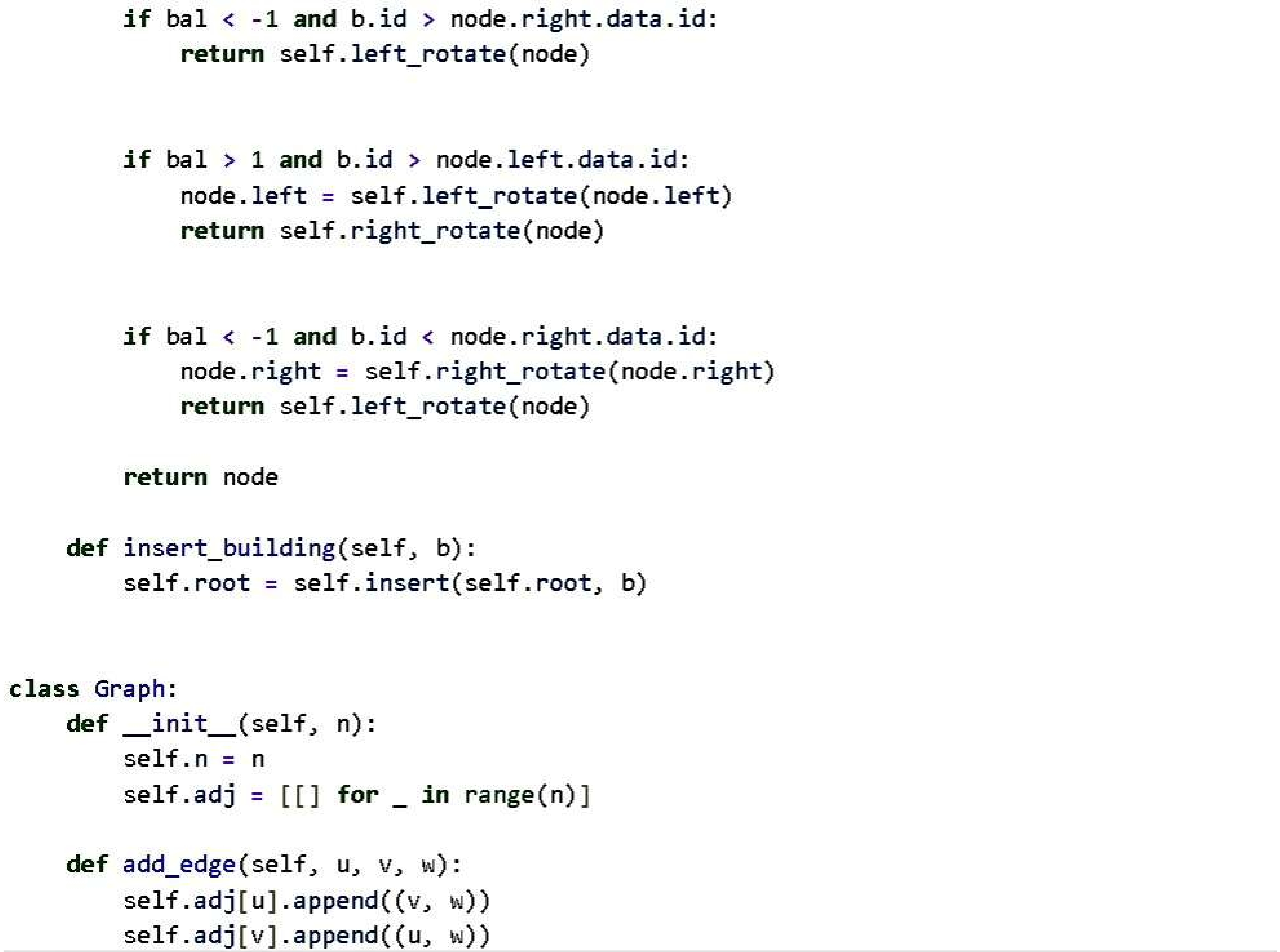


```python
class Building:
    def __init__(self, bid, name, location):
        self.id = bid
        self.name = name
        self.location = location


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


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

    def insert(self, node, b):
        if node is None:
            return BSTNode(b)
        if b.id < node.data.id:
            node.left = self.insert(node.left, b)
        else:
            node.right = self.insert(node.right, b)
        return node
```

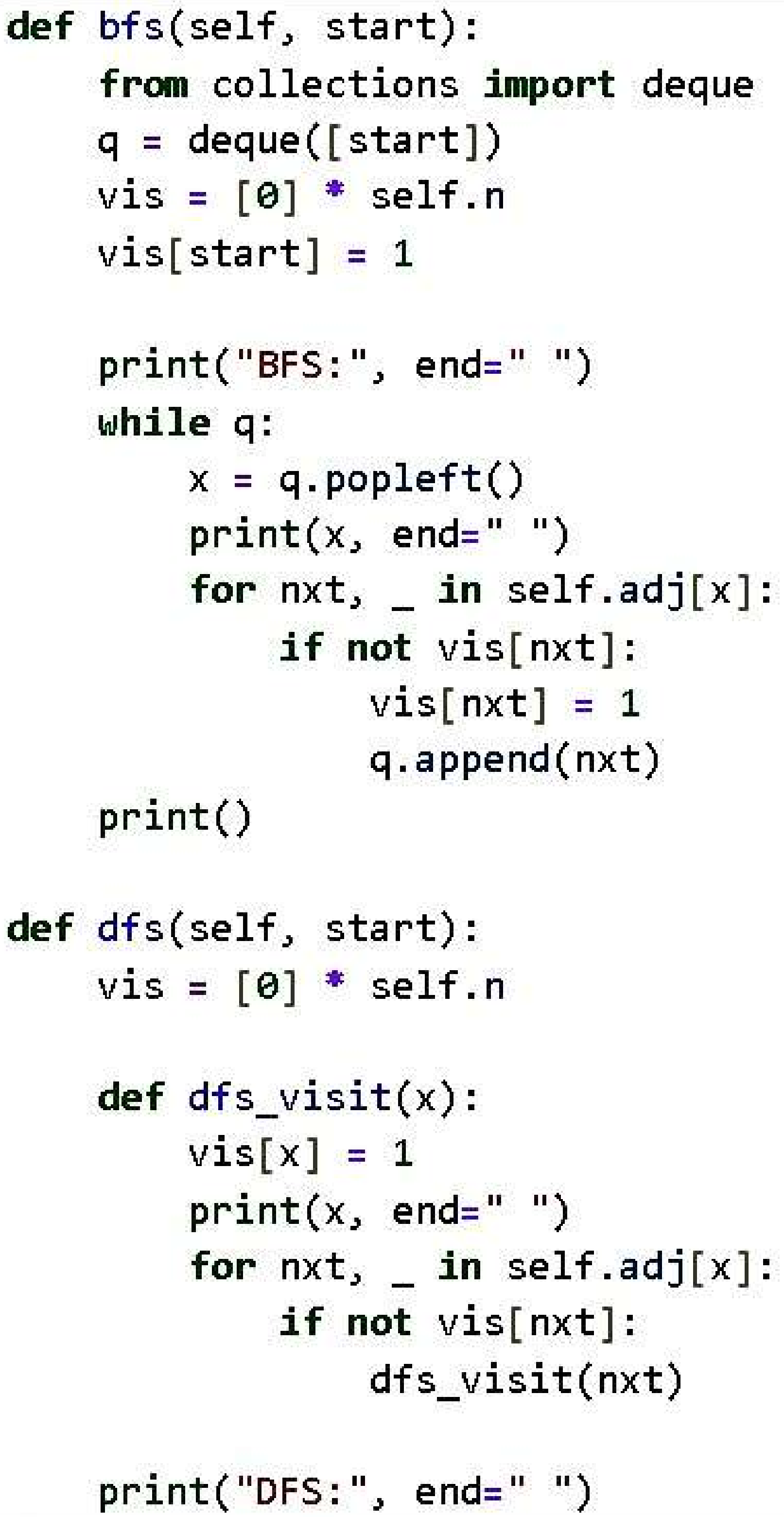


```python
        self.height = 1


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

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

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

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

    def right_rotate(self, y):
        x = y.left
        t = x.right

        x.right = y
        y.left = t

        self.update_height(y)
        self.update_height(x)
        return x
```

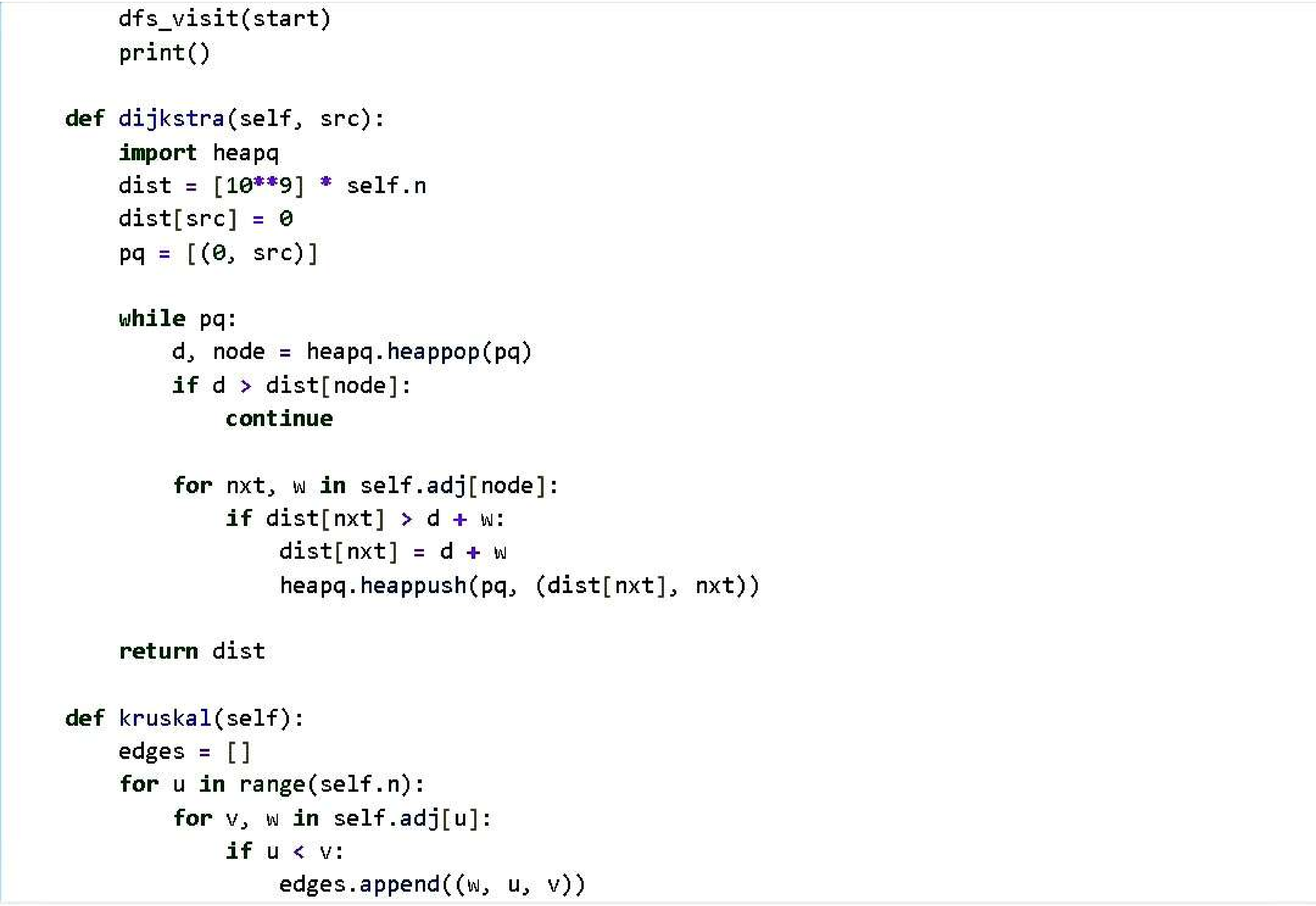
```
def left_rotate(self, x):
    y = x.right
    t = y.left

    y.left = x
    x.right = t

    self.update_height(x)
    self.update_height(y)
    return y

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

    if b.id < node.data.id:
        node.left = self.insert(node.left, b)
    else:
        node.right = self.insert(node.right, b)

    self.update_height(node)
    bal = self.balance(node)

    # LL
    if bal > 1 and b.id < node.left.data.id:
        return self.right_rotate(node)
```



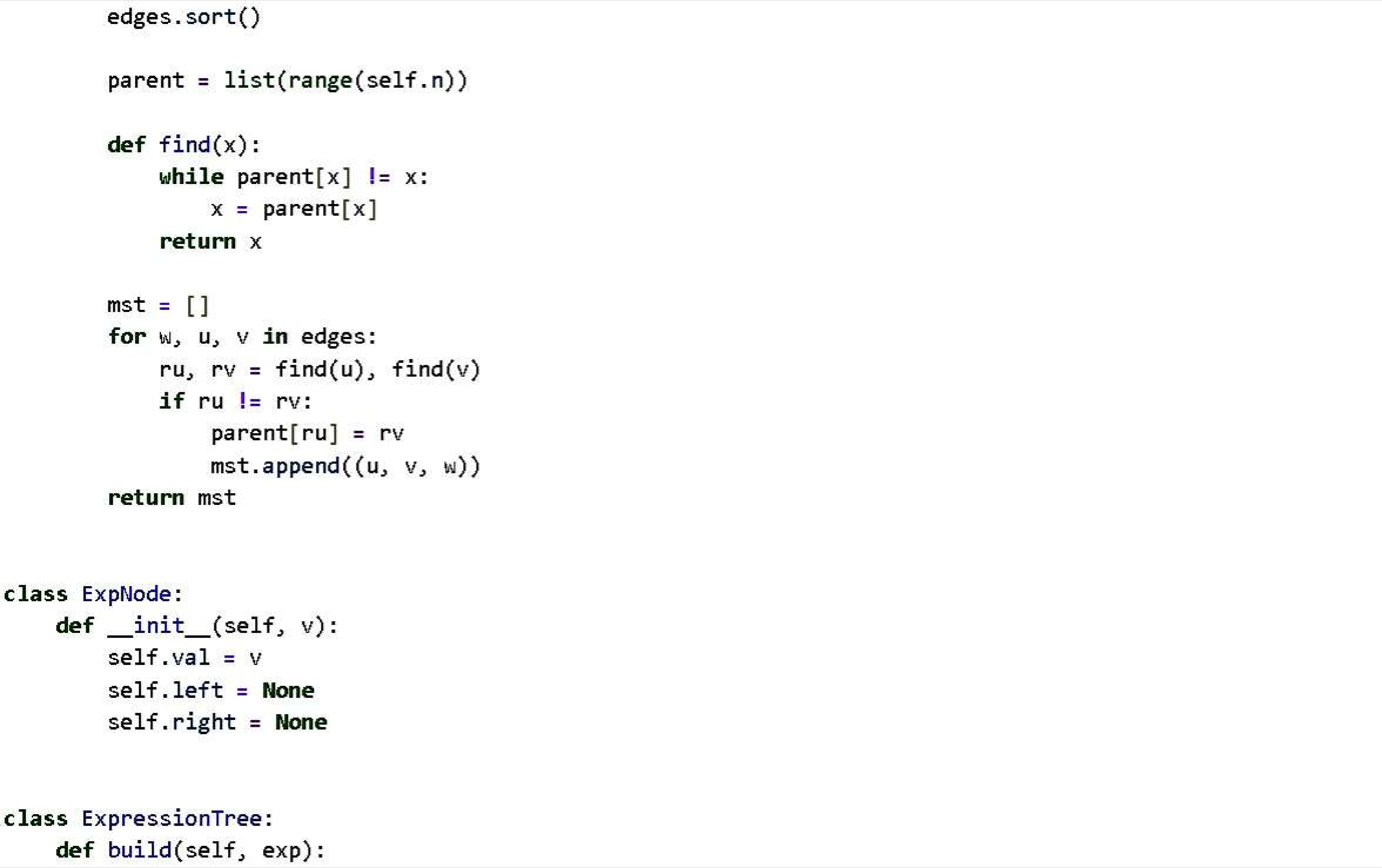
```
        if bal < -1 and b.id > node.right.data.id:
            return self.left_rotate(node)


        if bal > 1 and b.id > node.left.data.id:
            node.left = self.left_rotate(node.left)
            return self.right_rotate(node)


        if bal < -1 and b.id < node.right.data.id:
            node.right = self.right_rotate(node.right)
            return self.left_rotate(node)

        return node

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


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

    def add_edge(self, u, v, w):
        self.adj[u].append((v, w))
        self.adj[v].append((u, w))
```



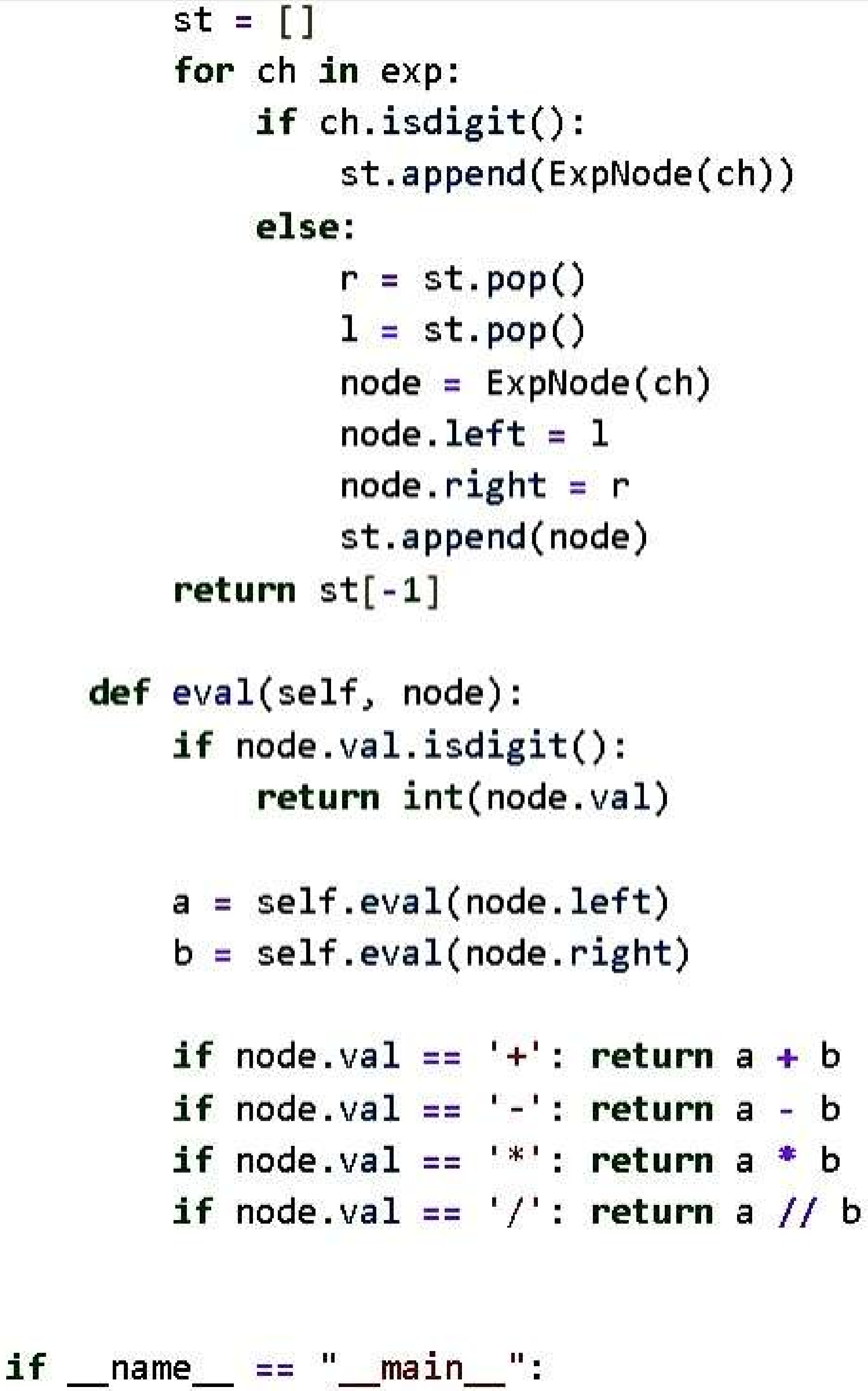
```python
    def bfs(self, start):
        from collections import deque
        q = deque([start])
        vis = [0] * self.n
        vis[start] = 1

        print("BFS:", end=" ")
        while q:
            x = q.popleft()
            print(x, end=" ")
            for nxt, _ in self.adj[x]:
                if not vis[nxt]:
                    vis[nxt] = 1
                    q.append(nxt)
        print()

    def dfs(self, start):
        vis = [0] * self.n

        def dfs_visit(x):
            vis[x] = 1
            print(x, end=" ")
            for nxt, _ in self.adj[x]:
                if not vis[nxt]:
                    dfs_visit(nxt)

        print("DFS:", end=" ")
```



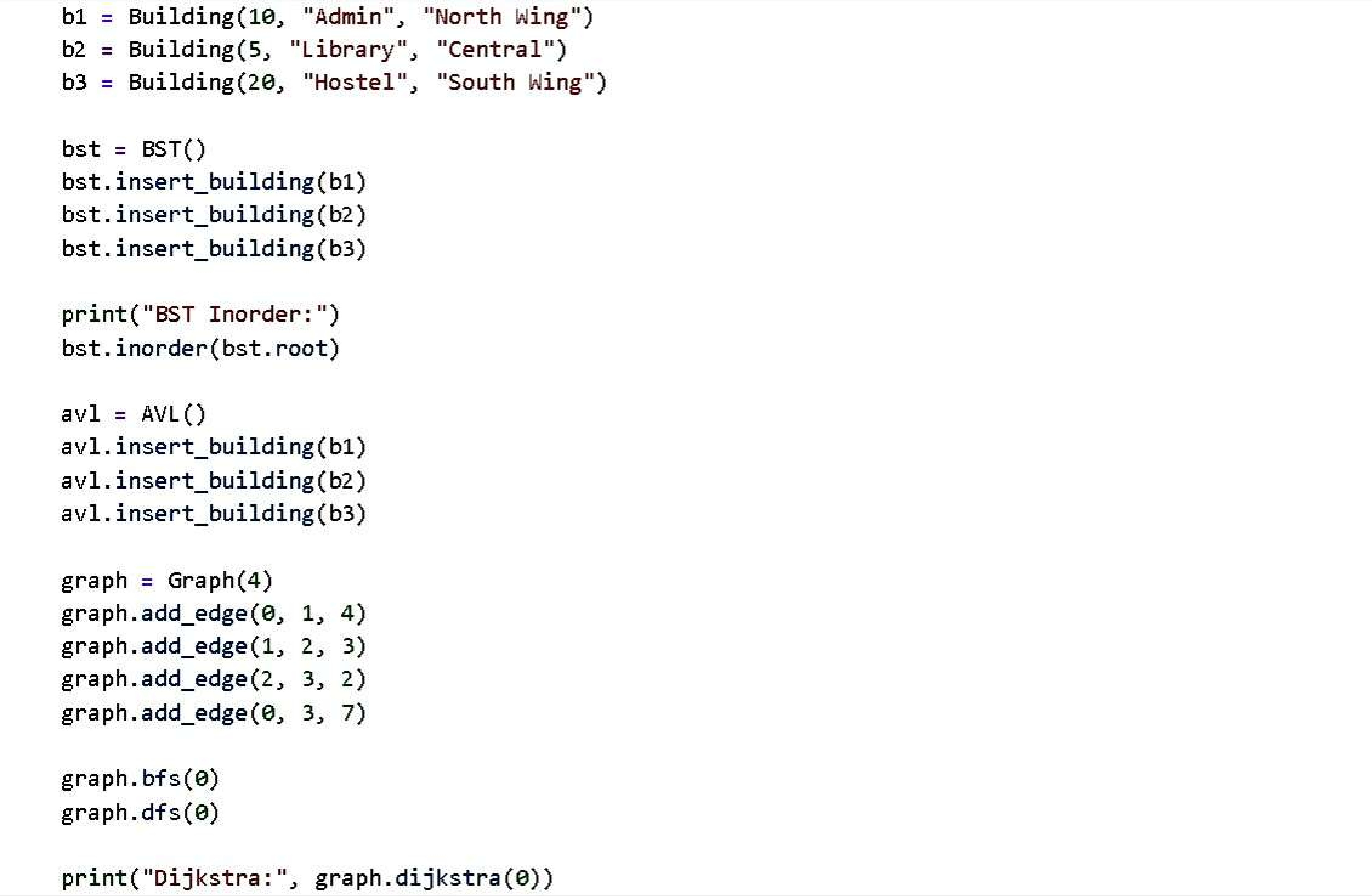
```
        dfs_visit(start)
        print()

    def dijkstra(self, src):
        import heapq
        dist = [10**9] * self.n
        dist[src] = 0
        pq = [(0, src)]

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

            for nxt, w in self.adj[node]:
                if dist[nxt] > d + w:
                    dist[nxt] = d + w
                    heapq.heappush(pq, (dist[nxt], nxt))

        return dist

    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))
```



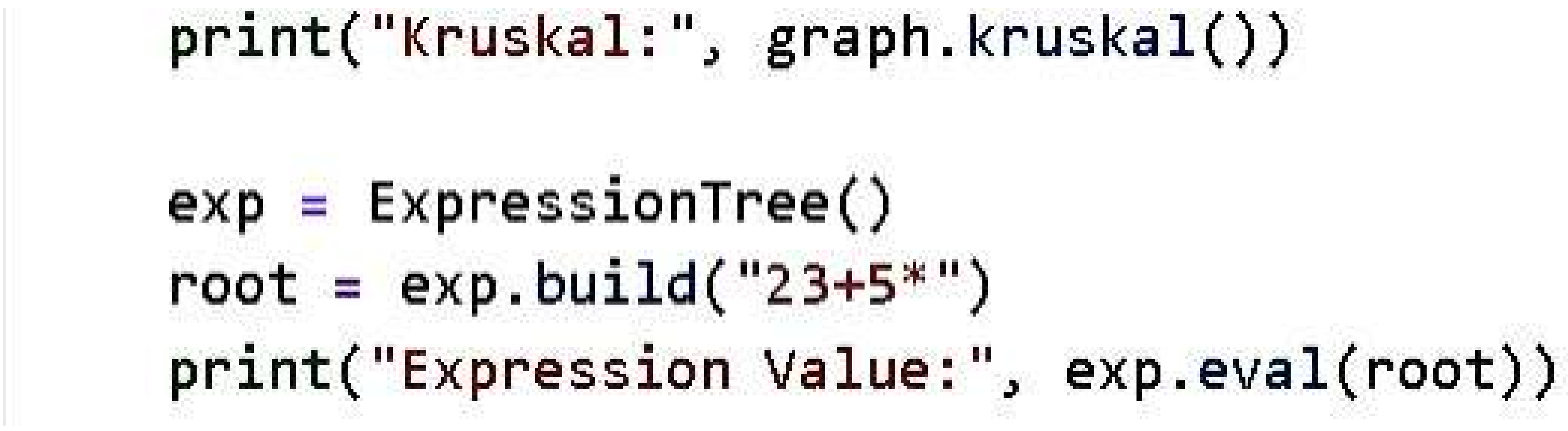
```python
        edges.sort()

        parent = list(range(self.n))

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

        mst = []
        for w, u, v in edges:
            ru, rv = find(u), find(v)
            if ru != rv:
                parent[ru] = rv
                mst.append((u, v, w))
        return mst


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


class ExpressionTree:
    def build(self, exp):
```



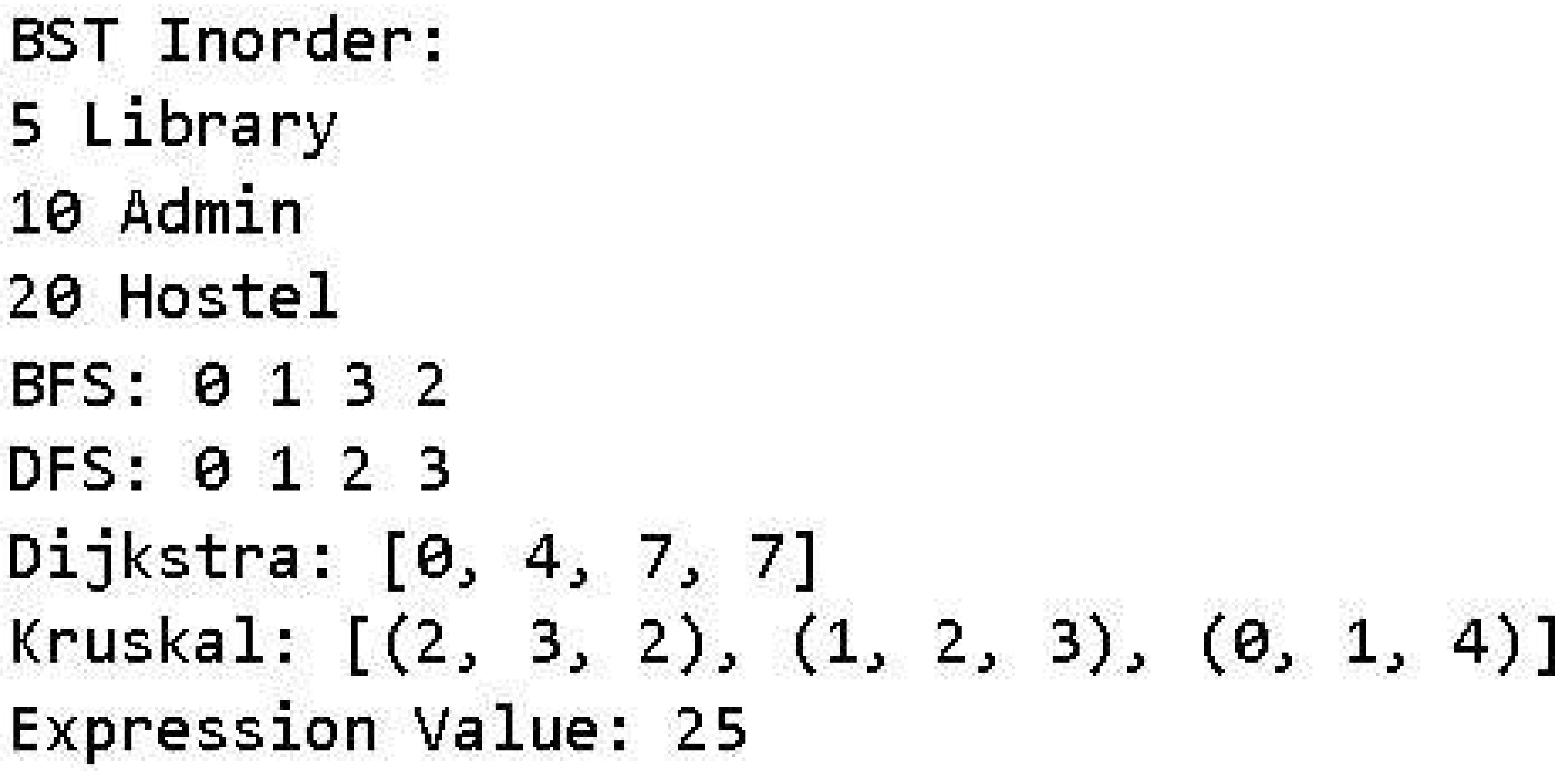
```
        st = []
        for ch in exp:
            if ch.isdigit():
                st.append(ExpNode(ch))
            else:
                r = st.pop()
                l = st.pop()
                node = ExpNode(ch)
                node.left = l
                node.right = r
                st.append(node)
        return st[-1]

    def eval(self, node):
        if node.val.isdigit():
            return int(node.val)

        a = self.eval(node.left)
        b = self.eval(node.right)

        if node.val == '+': return a + b
        if node.val == '-': return a - b
        if node.val == '*': return a * b
        if node.val == '/': return a // b


if __name__ == "__main__":
```



```
b1 = Building(10, "Admin", "North Wing")
b2 = Building(5, "Library", "Central")
b3 = Building(20, "Hostel", "South Wing")

bst = BST()
bst.insert_building(b1)
bst.insert_building(b2)
bst.insert_building(b3)

print("BST Inorder:")
bst.inorder(bst.root)

avl = AVL()
avl.insert_building(b1)
avl.insert_building(b2)
avl.insert_building(b3)

graph = Graph(4)
graph.add_edge(0, 1, 4)
graph.add_edge(1, 2, 3)
graph.add_edge(2, 3, 2)
graph.add_edge(0, 3, 7)

graph.bfs(0)
graph.dfs(0)

print("Dijkstra:", graph.dijkstra(0))
```



```
print("Kruskal:", graph.kruskal())

exp = ExpressionTree()
root = exp.build("23+5*")
print("Expression Value:", exp.eval(root))
```



```
BST Inorder:
5 Library
10 Admin
20 Hostel
BFS: 0 1 3 2
DFS: 0 1 2 3
Dijkstra: [0, 4, 7, 7]
Kruskal: [(2, 3, 2), (1, 2, 3), (0, 1, 4)]
Expression Value: 25
```