Page No.
Date :

Assignment - 3

→
```
class Building:
    def __init__(self, buildingID, buildingName, locationDetails):
        self.buildingID = buildingID
        self.buildingName = buildingName
        self.locationDetails = locationDetails
        self.connections = []

    def __str__(self):
        return f"{self.buildingID} - {self.buildingName}
                ({self.locationDetails})"
```

(1) Binary Search tree (BST)

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

class BinarySearchTree:
```

Page No.
Date :

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

def insertBuilding(self, building):
    self.root = self._insert(self.root, building)

def _insert(self, node, building):
    if node is None:
        return BSTNode(building)
    if building.buildingID < node.key:
        node.left = self._insert(node.left, building)
    else:
        node.right = self._insert(node.right, building)
    return node

def search(self, key):
    return self._search(self.root, key)

def _search(self, node, key):
    if node is None:
        return None
    if node.key == key:
        return node.data
    if key < node.key:
        return self._search(node.left, key)
```

```
        else:
            return self._search(node.right, key)

    def traverseBuildings(self, orderType = "inorder"):
        result = []
        if orderType == "inorder":
            self._inorder(self.root, result)
        elif orderType == "preorder":
            self._preorder(self.root, result)
        elif orderType == "postorder":
            self._postorder(self.root, result)
        return result

    def _inorder(self, node, result):
        if node:
            self._inorder(node.left, result)
            result.append(str(node.data))
            self._inorder(node.right, result)

    def _preorder(self, node, result):
        if node:
            result.append(str(node.data))
            self._preorder(node.left, result)
            self._preorder(node.right, result)
```

Page No.
Date :

```
def _postorder(self, node, result):
    if node:
        self._postorder(node.left, result)
        self._postorder(node.right, result)
        result.append(str(node.data))

def height(self, node):
    if node is None:
        return 0
    return 1 + max(self.height(node.left), self.height
                   (node.right))
```

2. AVL TREE

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

class AVLTree:

    def height(self, node):
```

```
    if node is None:
        return 0
    return node.height

def balanceFactor(self, node):
    return self.height(node.left) - self.height(node.right

def rightRotate(self, y):
    print("RR Rotation (Right Rotate)")
    x = y.left
    t2 = x.right
    x.right = y
    y.left = t2

    y.height = 1 + max(self.height(y.left), self.height(y.right
    x.height = 1 + max(self.height(x.left), self.height(x.right
    return x

def leftRotate(self, x):
    print("LL Rotation (Left Rotate)")
    y = x.right
    t2 = y.left
    y.left = x
    x.right = t2
```

Page No.
Date :

```
x.height = 1 + max (self.height(x.left), self.height(x.right))
y.height = 1 + max(self.height(y.left), " " (y.right))

def insertBuilding (self, node, building):
    if node is None:
        return AVLNode (building)

    if building.buildingID < node.key:
        node.left = self.insertBuilding(node.left, building)
    else:
        node.right = self.insertBuilding(node.right, building)

    node.height = 1 + max (self.height(node.left), self.height
                                                    (node.right))

    bf = self.balanceFactor (node)

    return node
```

Page No.
Date :

P-2 Graph IMPLEMENTATION

(1) → from collections import deque

```
class CampusGraph:

    def __init__(self, totalBuildings):
        self.adjList = [[] for _ in range(totalBuildings)]

        self.adjMatrix = [[0 for _ in range(totalBuildings)]
                          for _ in range(totalBuildings)]

        self.total = totalBuildings

    def addPath(self, building1, building2, distance):

        self.adjList[building1].append((building2, distance))
        self.adjList[building2].append((building1, distance))

        self.adjMatrix[building1][building2] = distance
        self.adjMatrix[building2][building1] = distance
```

(2) BFS Traversal

Page No.
Date :

```
def BFS(self,start):
    visited = set()
    queue = deque([start])
    visited.add(start)
    bfs_order=[]

    while queue:
        node = queue.popleft()
        bfs_order.append(node)

        for neighbor, _ in self.adjList[node]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)

return bfs_order
```

(ii) DFS Traversal