```
print("ROHAN WAYAL")

# Binary Search Tree implementation with requested operations

class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

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

    # a) Insert (Handle duplicate by ignoring insertion)
    def insert(self, key):
        self.root = self._insert(self.root, key)

    def _insert(self, root, key):
        if root is None:
            return Node(key)
        if key < root.key:
            root.left = self._insert(root.left, key)
        elif key > root.key:
            root.right = self._insert(root.right, key)
        else:
            # Duplicate key, ignoring insertion
            pass
        return root

    # b) Delete
```

```
def delete(self, key):
    self.root = self._delete(self.root, key)

def _delete(self, root, key):
    if root is None:
        return root
    if key < root.key:
        root.left = self._delete(root.left, key)
    elif key > root.key:
        root.right = self._delete(root.right, key)
    else:
        # Node with only one child or no child
        if root.left is None:
            return root.right
        elif root.right is None:
            return root.left
        # Node with two children: Get inorder successor
        temp = self._minValueNode(root.right)
        root.key = temp.key
        root.right = self._delete(root.right, temp.key)
    return root

def _minValueNode(self, node):
    current = node
    while current.left is not None:
        current = current.left
    return current

# c) Search
def search(self, key):
    return self._search(self.root, key)
```

```
def _search(self, root, key):
    if root is None or root.key == key:
        return root
    if key < root.key:
        return self._search(root.left, key)
    else:
        return self._search(root.right, key)

# d) Display tree (Traversal) - Inorder, Preorder, Postorder
def inorder(self):
    res = []
    self._inorder(self.root, res)
    return res

def _inorder(self, root, res):
    if root:
        self._inorder(root.left, res)
        res.append(root.key)
        self._inorder(root.right, res)

def preorder(self):
    res = []
    self._preorder(self.root, res)
    return res

def _preorder(self, root, res):
    if root:
        res.append(root.key)
        self._preorder(root.left, res)
        self._preorder(root.right, res)
```

```
def postorder(self):
    res = []
    self._postorder(self.root, res)
    return res

def _postorder(self, root, res):
    if root:
        self._postorder(root.left, res)
        self._postorder(root.right, res)
        res.append(root.key)

# e) Display - Depth of tree
def depth(self):
    return self._depth(self.root)

def _depth(self, root):
    if root is None:
        return 0
    left_depth = self._depth(root.left)
    right_depth = self._depth(root.right)
    return max(left_depth, right_depth) + 1

# f) Display - Mirror image
def mirror(self):
    self._mirror(self.root)

def _mirror(self, root):
    if root:
        root.left, root.right = root.right, root.left
        self._mirror(root.left)
```

```
        self._mirror(root.right)

# g) Create a copy of the tree
def copy(self):
    new_tree = BST()
    new_tree.root = self._copy(self.root)
    return new_tree

def _copy(self, root):
    if root is None:
        return None
    new_node = Node(root.key)
    new_node.left = self._copy(root.left)
    new_node.right = self._copy(root.right)
    return new_node

# h) Display all parent nodes with their child nodes
def display_parents_with_children(self):
    result = []
    self._display_parents_with_children(self.root, result)
    return result

def _display_parents_with_children(self, root, result):
    if root:
        children = []
        if root.left:
            children.append(root.left.key)
        if root.right:
            children.append(root.right.key)
        if children:
            result.append((root.key, children))
```

```
        self._display_parents_with_children(root.left, result)
        self._display_parents_with_children(root.right, result)

# i) Display leaf nodes
def leaf_nodes(self):
    leaves = []
    self._leaf_nodes(self.root, leaves)
    return leaves

def _leaf_nodes(self, root, leaves):
    if root:
        if root.left is None and root.right is None:
            leaves.append(root.key)
        self._leaf_nodes(root.left, leaves)
        self._leaf_nodes(root.right, leaves)

# j) Display tree level wise (Level Order Traversal)
def level_order(self):
    res = []
    if self.root is None:
        return res
    queue = [self.root]
    while queue:
        level_size = len(queue)
        level_nodes = []
        for _ in range(level_size):
            node = queue.pop(0)
            level_nodes.append(node.key)
            if node.left:
                queue.append(node.left)
            if node.right:
```

```
                queue.append(node.right)
            res.append(level_nodes)
        return res


# Sample usage:

bst = BST()

# Insert nodes
for val in [50, 30, 20, 40, 70, 60, 80, 70]:  # 70 duplicate ignored
    bst.insert(val)

print("Inorder Traversal:", bst.inorder())
print("Preorder Traversal:", bst.preorder())
print("Postorder Traversal:", bst.postorder())

# Search for a node
key = 40
found = bst.search(key)
print(f"Search {key}:", "Found" if found else "Not Found")

# Depth of tree
print("Depth of tree:", bst.depth())

# Display parents with children
print("Parents with children:", bst.display_parents_with_children())

# Leaf nodes
print("Leaf nodes:", bst.leaf_nodes())
```

```
# Level order traversal
print("Level order traversal:", bst.level_order())

# Create a copy of the tree
bst_copy = bst.copy()
print("Copy - Inorder Traversal:", bst_copy.inorder())

# Mirror the tree
bst.mirror()
print("Mirror - Inorder Traversal:", bst.inorder())

# Delete a node
bst.delete(30)
print("After deleting 30, Inorder Traversal:", bst.inorder())
```