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

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

    # Insert a new key (handles duplicates by inserting to the right)
    def insert(self, key):
        self.root = self._insert(self.root, key)

    def _insert(self, node, key):
        if node is None:
            return TreeNode(key)
        if key < node.key:
            node.left = self._insert(node.left, key)
        else:  # Handle duplicates by inserting in the right subtree
            node.right = self._insert(node.right, key)
        return node

    # Delete a key from the BST
    def delete(self, key):
        self.root = self._delete(self.root, key)

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

    def _find_min(self, node):
        current = node
```

```
        while current.left is not None:
            current = current.left
        return current

    # Display the tree using inorder, preorder, and postorder traversals
    def inorder(self):
        print("Inorder Traversal:", end=" ")
        self._inorder(self.root)
        print()

    def _inorder(self, node):
        if node:
            self._inorder(node.left)
            print(node.key, end=" ")
            self._inorder(node.right)

    def preorder(self):
        print("Preorder Traversal:", end=" ")
        self._preorder(self.root)
        print()

    def _preorder(self, node):
        if node:
            print(node.key, end=" ")
            self._preorder(node.left)
            self._preorder(node.right)

    def postorder(self):
        print("Postorder Traversal:", end=" ")
        self._postorder(self.root)
        print()

    def _postorder(self, node):
        if node:
            self._postorder(node.left)
            self._postorder(node.right)
            print(node.key, end=" ")

# Example usage
if __name__ == "__main__":
    bst = BST()
    # Insert keys
    keys = [50, 30, 70, 20, 40, 60, 80, 30]  # Includes duplicate 30
    for key in keys:
        bst.insert(key)

    print("After Insertion:")
    bst.inorder()
```

```
    bst.preorder()
    bst.postorder()

    # Delete keys
    print("\nDeleting 70 and 30 (duplicate handled):")
    bst.delete(70)
    bst.delete(30)
    bst.inorder()
    bst.preorder()
    bst.postorder()
```

B

```
# Definition for a node in the binary search tree
class TreeNode:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

# Definition of the Binary Search Tree
class BST:
    def __init__(self):
        self.root = None

    # Insert a key into the BST (handles duplicates by placing them in the right subtree)
    def insert(self, key):
        self.root = self._insert_recursive(self.root, key)

    def _insert_recursive(self, node, key):
        if node is None:
            return TreeNode(key)
        if key < node.key:
            node.left = self._insert_recursive(node.left, key)
        else:
            node.right = self._insert_recursive(node.right, key)
        return node

    # Search for a key in the BST
    def search(self, key):
        return self._search_recursive(self.root, key)

    def _search_recursive(self, node, key):
        if node is None:
            return False  # Key not found
        if node.key == key:
            return True   # Key found
        elif key < node.key:
            return self._search_recursive(node.left, key)
```

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

# Display the tree using different traversals
def inorder(self):
    result = []
    self._inorder_recursive(self.root, result)
    return result

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

def preorder(self):
    result = []
    self._preorder_recursive(self.root, result)
    return result

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

def postorder(self):
    result = []
    self._postorder_recursive(self.root, result)
    return result

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

# Find the depth (height) of the tree
def tree_depth(self):
    return self._calculate_depth(self.root)

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

```
# Example usage
if __name__ == "__main__":
    bst = BST()
    keys = [50, 30, 70, 20, 40, 60, 80, 30]  # Duplicate 30 is handled
    for key in keys:
        bst.insert(key)

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

    search_key = 40
    print(f"Search for {search_key}:", "Found" if bst.search(search_key) else "Not Found")

    search_key = 90
    print(f"Search for {search_key}:", "Found" if bst.search(search_key) else "Not Found")

    print("Depth of the tree:", bst.tree_depth())
```

C.

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

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

    # 1. Insert a node into the BST (duplicates go to the right)
    def insert(self, key):
        if not self.root:
            self.root = TreeNode(key)
            return
        self._insert_recursively(self.root, key)

    def _insert_recursively(self, node, key):
        if key < node.key:
            if node.left is None:
                node.left = TreeNode(key)
            else:
                self._insert_recursively(node.left, key)
        else:  # For duplicates, insert to the right
            if node.right is None:
                node.right = TreeNode(key)
            else:
                self._insert_recursively(node.right, key)
```

```
    # 2. Display the tree using in-order traversal
    def display_tree(self):
        print("In-order Traversal of BST:")
        self._inorder(self.root)
        print()

    def _inorder(self, node):
        if node:
            self._inorder(node.left)
            print(node.key, end=" ")
            self._inorder(node.right)

    # 3. Show parent-child relationships
    def show_parent_child(self):
        print("\nParent-Child Relationships:")
        self._show_parent_child(self.root)

    def _show_parent_child(self, node):
        if node:
            if node.left:
                print(f"Parent: {node.key} -> Left Child: {node.left.key}")
            if node.right:
                print(f"Parent: {node.key} -> Right Child: {node.right.key}")
            self._show_parent_child(node.left)
            self._show_parent_child(node.right)

    # 4. Display all leaf nodes
    def display_leaf_nodes(self):
        print("\nLeaf Nodes in BST:")
        self._print_leaves(self.root)
        print()

    def _print_leaves(self, node):
        if node:
            if not node.left and not node.right:
                print(node.key, end=" ")
            self._print_leaves(node.left)
            self._print_leaves(node.right)


# Driver Code
def main():
    bst = BinarySearchTree()
    while True:
        print("\n********** Binary Search Tree Operations **********")
        print("1. Insert a node")
        print("2. Display the tree (In-order Traversal)")
```

```
        print("3. Show Parent-Child Nodes")
        print("4. Display Leaf Nodes")
        print("5. Exit")
        choice = input("Enter your choice: ")

        if choice == "1":
            key = int(input("Enter the key to insert: "))
            bst.insert(key)
        elif choice == "2":
            bst.display_tree()
        elif choice == "3":
            bst.show_parent_child()
        elif choice == "4":
            bst.display_leaf_nodes()
        elif choice == "5":
            print("Exiting...")
            break
        else:
            print("Invalid choice! Please try again.")

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

D.

```
from collections import deque

# Definition for a node in the binary search tree
class TreeNode:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

# Definition of the Binary Search Tree
class BST:
    def __init__(self):
        self.root = None

    # Insert a key into the BST (handles duplicates by placing them in the right subtree)
    def insert(self, key):
        self.root = self._insert_recursive(self.root, key)

    def _insert_recursive(self, node, key):
        if node is None:
            return TreeNode(key)
        if key < node.key:
            node.left = self._insert_recursive(node.left, key)
        else:  # Handles duplicates by placing them in the right subtree
```

```
        node.right = self._insert_recursive(node.right, key)
    return node

# Display the tree level-wise (BFS traversal)
def display_levelwise(self):
    if not self.root:
        print("Tree is empty.")
        return

    print("\nTree displayed level-wise:")
    queue = deque([self.root])
    while queue:
        current = queue.popleft()
        print(current.key, end=" ")

        if current.left:
            queue.append(current.left)
        if current.right:
            queue.append(current.right)
    print()

# Display the mirror image of the tree
def display_mirror(self):
    print("\nMirror Image of the Tree (Level-wise):")
    mirror_root = self._create_mirror(self.root)
    self._levelwise_display_from_root(mirror_root)

def _create_mirror(self, node):
    if node is None:
        return None
    # Swap the left and right subtrees recursively
    mirrored = TreeNode(node.key)
    mirrored.left = self._create_mirror(node.right)
    mirrored.right = self._create_mirror(node.left)
    return mirrored

def _levelwise_display_from_root(self, root):
    if not root:
        return
    queue = deque([root])
    while queue:
        current = queue.popleft()
        print(current.key, end=" ")

        if current.left:
            queue.append(current.left)
        if current.right:
            queue.append(current.right)
```

```
        print()

    # Create and return a copy of the tree
    def create_copy(self):
        print("\nCopy of the Tree (Level-wise):")
        copied_root = self._copy_tree(self.root)
        self._levelwise_display_from_root(copied_root)

    def _copy_tree(self, node):
        if node is None:
            return None
        # Create a new node and recursively copy the subtrees
        new_node = TreeNode(node.key)
        new_node.left = self._copy_tree(node.left)
        new_node.right = self._copy_tree(node.right)
        return new_node

# Example usage
if __name__ == "__main__":
    bst = BST()
    keys = [50, 30, 70, 20, 40, 60, 80, 30]  # Duplicate 30 is handled
    for key in keys:
        bst.insert(key)

    # Display the tree level-wise
    bst.display_levelwise()

    # Display the mirror image of the tree
    bst.display_mirror()

    # Create and display a copy of the tree
    bst.create_copy()
```