```python
class TreeNode:
    """Node class for Binary Search Tree"""
    def __init__(self, val=0):
        self.val = val
        self.left = None
        self.right = None

class BinarySearchTree:
    """Binary Search Tree implementation with common operations"""

    def __init__(self):
        self.root = None

    def insert(self, val):
        """Insert a value into the BST"""
        self.root = self._insert_recursive(self.root, val)

    def _insert_recursive(self, node, val):
        # Base case: create new node
        if not node:
            return TreeNode(val)

        # Insert in left or right subtree
        if val < node.val:
            node.left = self._insert_recursive(node.left, val)
        elif val > node.val:
            node.right = self._insert_recursive(node.right, val)
        # If val == node.val, we don't insert duplicates

        return node

    def search(self, val):
        """Search for a value in the BST"""
        return self._search_recursive(self.root, val)

    def _search_recursive(self, node, val):
        # Base cases
        if not node or node.val == val:
            return node

        # Search in left or right subtree
        if val < node.val:
            return self._search_recursive(node.left, val)
        else:
            return self._search_recursive(node.right, val)

    def delete(self, val):
        """Delete a value from the BST"""
        self.root = self._delete_recursive(self.root, val)

    def _delete_recursive(self, node, val):
        # Base case
        if not node:
            return node
```

```
    # Find the node to delete
    if val < node.val:
        node.left = self._delete_recursive(node.left, val)
    elif val > node.val:
        node.right = self._delete_recursive(node.right, val)
    else:
        # Node to be deleted found
        # Case 1: Node has no children
        if not node.left and not node.right:
            return None

        # Case 2: Node has one child
        if not node.left:
            return node.right
        if not node.right:
            return node.left

        # Case 3: Node has two children
        # Find inorder successor (smallest in right subtree)
        successor = self._find_min(node.right)
        node.val = successor.val
        node.right = self._delete_recursive(node.right, successor.val)

    return node

def _find_min(self, node):
    """Find the minimum value node in a subtree"""
    while node.left:
        node = node.left
    return node

def find_min(self):
    """Find minimum value in the BST"""
    if not self.root:
        return None
    return self._find_min(self.root).val

def find_max(self):
    """Find maximum value in the BST"""
    if not self.root:
        return None
    node = self.root
    while node.right:
        node = node.right
    return node.val

def inorder_traversal(self):
    """Inorder traversal (left, root, right) - gives sorted order"""
    result = []
    self._inorder_recursive(self.root, result)
    return result

def _inorder_recursive(self, node, result):
    if node:
        self._inorder_recursive(node.left, result)
        result.append(node.val)
        self._inorder_recursive(node.right, result)
```

```python
    def preorder_traversal(self):
        """Preorder traversal (root, left, right)"""
        result = []
        self._preorder_recursive(self.root, result)
        return result

    def _preorder_recursive(self, node, result):
        if node:
            result.append(node.val)
            self._preorder_recursive(node.left, result)
            self._preorder_recursive(node.right, result)

    def postorder_traversal(self):
        """Postorder traversal (left, right, root)"""
        result = []
        self._postorder_recursive(self.root, result)
        return result

    def _postorder_recursive(self, node, result):
        if node:
            self._postorder_recursive(node.left, result)
            self._postorder_recursive(node.right, result)
            result.append(node.val)

    def height(self):
        """Calculate height of the BST"""
        return self._height_recursive(self.root)

    def _height_recursive(self, node):
        if not node:
            return -1  # Height of empty tree is -1
        return 1 + max(self._height_recursive(node.left),
                       self._height_recursive(node.right))

    def size(self):
        """Count total number of nodes"""
        return self._size_recursive(self.root)

    def _size_recursive(self, node):
        if not node:
            return 0
        return 1 + self._size_recursive(node.left) + self._size_recursive(node.right)

# BST Validation Functions
def is_valid_bst_v1(root):
    """
    Validate BST using inorder traversal
    A valid BST's inorder traversal should be in ascending order
    """
    def inorder(node, values):
        if node:
            inorder(node.left, values)
            values.append(node.val)
            inorder(node.right, values)

    values = []
    inorder(root, values)
```

```
    # Check if values are in ascending order
    for i in range(1, len(values)):
        if values[i] <= values[i-1]:
            return False
    return True

def is_valid_bst_v2(root):
    """
    Validate BST using bounds checking (more efficient)
    Each node must be within valid min/max bounds
    """
    def validate(node, min_val, max_val):
        if not node:
            return True

        if node.val <= min_val or node.val >= max_val:
            return False

        return (validate(node.left, min_val, node.val) and
                validate(node.right, node.val, max_val))

    return validate(root, float('-inf'), float('inf'))

# Example usage and testing
if __name__ == "__main__":
    # Create BST and test operations
    bst = BinarySearchTree()

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

    print("BST created with values:", values)
    print("Inorder traversal (sorted):", bst.inorder_traversal())
    print("Preorder traversal:", bst.preorder_traversal())
    print("Postorder traversal:", bst.postorder_traversal())
    print("Height:", bst.height())
    print("Size:", bst.size())
    print("Min value:", bst.find_min())
    print("Max value:", bst.find_max())

    # Search operations
    print("\nSearch for 40:", "Found" if bst.search(40) else "Not found")
    print("Search for 100:", "Found" if bst.search(100) else "Not found")

    # Validate BST
    print("\nIs valid BST (method 1):", is_valid_bst_v1(bst.root))
    print("Is valid BST (method 2):", is_valid_bst_v2(bst.root))

    # Delete operation
    print("\nDeleting 30...")
    bst.delete(30)
    print("Inorder after deletion:", bst.inorder_traversal())
    print("Is still valid BST:", is_valid_bst_v2(bst.root))
```

```python
# Simple queue implementation using list
class SimpleQueue:
    def __init__(self):
        self.items = []

    def enqueue(self, item):
        self.items.append(item)

    def dequeue(self):
        if self.items:
            return self.items.pop(0)
        return None

    def is_empty(self):
        return len(self.items) == 0

# Tree Node for tree traversals
class TreeNode:
    def __init__(self, val=0):
        self.val = val
        self.left = None
        self.right = None

# ==================== TREE TRAVERSALS ====================

def bfs_tree(root):
    """BFS traversal of binary tree (level order)"""
    if not root:
        return []

    result = []
    queue = SimpleQueue()
    queue.enqueue(root)

    while not queue.is_empty():
        node = queue.dequeue()
        result.append(node.val)

        if node.left:
            queue.enqueue(node.left)
        if node.right:
            queue.enqueue(node.right)

    return result

def dfs_tree_recursive(root):
    """DFS traversal of binary tree (preorder) - Recursive"""
    if not root:
        return []

    result = [root.val]
    result.extend(dfs_tree_recursive(root.left))
    result.extend(dfs_tree_recursive(root.right))

    return result

def dfs_tree_iterative(root):
    """DFS traversal of binary tree (preorder) - Iterative"""
```

```
    if not root:
        return []

    result = []
    stack = [root]

    while stack:
        node = stack.pop()
        result.append(node.val)

        # Add right first, then left (so left is processed first)
        if node.right:
            stack.append(node.right)
        if node.left:
            stack.append(node.left)

    return result

# ==================== GRAPH TRAVERSALS ====================

def bfs_graph(graph, start):
    """BFS traversal of graph"""
    visited = set()
    result = []
    queue = SimpleQueue()

    queue.enqueue(start)
    visited.add(start)

    while not queue.is_empty():
        node = queue.dequeue()
        result.append(node)

        for neighbor in graph[node]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.enqueue(neighbor)

    return result

def dfs_graph_recursive(graph, start, visited=None):
    """DFS traversal of graph - Recursive"""
    if visited is None:
        visited = set()

    visited.add(start)
    result = [start]

    for neighbor in graph[start]:
        if neighbor not in visited:
            result.extend(dfs_graph_recursive(graph, neighbor, visited))

    return result

def dfs_graph_iterative(graph, start):
    """DFS traversal of graph - Iterative"""
    visited = set()
    result = []
```

```
    stack = [start]

    while stack:
        node = stack.pop()
        if node not in visited:
            visited.add(node)
            result.append(node)

            # Add neighbors in reverse order for consistent traversal
            for neighbor in reversed(graph[node]):
                if neighbor not in visited:
                    stack.append(neighbor)

    return result

# ==================== EXAMPLE USAGE ====================

if __name__ == "__main__":
    # Create a sample binary tree:
    #       1
    #      / \
    #     2   3
    #    / \
    #   4   5

    root = TreeNode(1)
    root.left = TreeNode(2)
    root.right = TreeNode(3)
    root.left.left = TreeNode(4)
    root.left.right = TreeNode(5)

    print("=== TREE TRAVERSALS ===")
    print("BFS (Level Order):", bfs_tree(root))
    print("DFS Recursive:", dfs_tree_recursive(root))
    print("DFS Iterative:", dfs_tree_iterative(root))

    # Create a sample graph:
    # A -- B -- D
    # |    |
    # C -- E

    graph = {
        'A': ['B', 'C'],
        'B': ['A', 'D', 'E'],
        'C': ['A', 'E'],
        'D': ['B'],
        'E': ['B', 'C']
    }

    print("\n=== GRAPH TRAVERSALS ===")
    print("Graph:", graph)
    print("BFS from A:", bfs_graph(graph, 'A'))
    print("DFS Recursive from A:", dfs_graph_recursive(graph, 'A'))
    print("DFS Iterative from A:", dfs_graph_iterative(graph, 'A'))

    # Compare different starting points
    print("\nBFS from C:", bfs_graph(graph, 'C'))
    print("DFS from C:", dfs_graph_recursive(graph, 'C'))
```