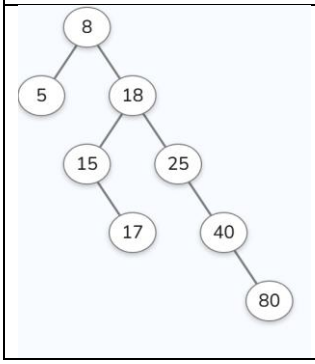
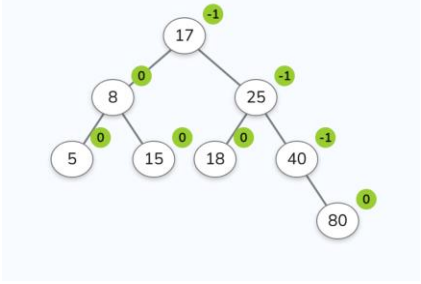
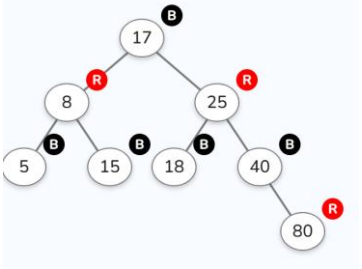
Exam Correction

Exercise 1

a)insert [8, 18, 5, 15, 17, 25, 40, 80]

BST (0.5pts)	**AVL** (1pts)	**RB TREE** (1pts)
		

b) Indicate the correct answer(s) for each of the following questions:

Q1)Why might open addressing be preferred over chaining in certain systems?

(0.5pts)

B. It improves cache locality due to contiguous memory storage

C. It avoids pointer overhead associated with linked structures

Q2)Which factors explain why tabulation is often preferred over memoization in practice?

(0.5pts)

A. It avoids recursion overhead and stack overflow risks

C. It enables predictable memory access patterns favorable to CPU caches

Q3)Why is studying NP-complete problems fundamental in computer science? (0.5pts)

A. It helps identify problems unlikely to have efficient exact solutions

C. It guides the use of approximation algorithms and heuristics

Q4)A large NP-complete problem is solved using dynamic programming with frequent state lookups. Which design choices are most justified?(0.5pts)

A. Tabulation to eliminate recursion overhead

C. Open addressing to improve cache efficiency

5)Which statements about NP-complete problems are correct?

(0.5pts)

A. A polynomial-time solution to one NP-complete problem implies P = NP

C. NP-complete problems are in NP

Exercise 2 (5pts)

```
def minimumTotal(triangle):
    memo = {}
    def dfs(row, col):
        # Base case: reached the bottom row
        if row == len(triangle) - 1:return triangle[row][col]

        if (row, col) in memo:   return memo[(row, col)]
        min_path = triangle[row][col] + min(
            dfs(row + 1, col),      # Move to same index
            dfs(row + 1, col + 1)   # Move to next index
        )

        memo[(row, col)] = min_path
        return min_path

    return dfs(0, 0)
```

Exercise 3(5pts)

```
def boundaryOfBinaryTree(root):
    if not root: return []
    result = [root.val]
    getLeftBoundary(root.left, result)
     getLeaves(root, result)
    getRightBoundary(root.right, result)
    return result
def getLeftBoundary(node, result):
    if not node:  return
    if node.left or node.right:
        result.append(node.val)

    if node.left:
        getLeftBoundary(node.left, result)
    else:
        getLeftBoundary(node.right, result)
def getRightBoundary(node, result):
    if not node: return
    if node.right:
        getRightBoundary(node.right, result)
    else:
        getRightBoundary(node.left, result)
    if node.left or node.right:
        result.append(node.val)
def getLeaves(node, result):
    if not node:    return
    if not node.left and not node.right:
        result.append(node.val)
        return
    getLeaves(node.left, result)
    getLeaves(node.right, result)
```

Exercise 4 (5pts)

def countCompleteConnectedComponents(n, edges):

```
    # Build adjacency list
    adj = [ [] for _ in range(n)]
    for u, v in edges:
        adj[u].append(v)
        adj[v].append(u)

    visited = set()
    complete_count = 0

    def dfs(node, component):
        visited.add(node)
        component.append(node)

        # Recursively visit all unvisited neighbors
        for neighbor in adj[node]:
            if neighbor not in visited:
                dfs(neighbor, component)
    # Find all connected components
    for i in range(n):
        if i not in visited:
            component = []
            dfs(i, component)
             # Check if this component is complete
            # A complete graph with k vertices has k*(k-1)/2 edges
            k = len(component)
            required_edges = k * (k - 1) // 2
            # Count actual edges in this component
            actual_edges = 0
            for node in component:
                actual_edges += len(adj[node])
            # Each edge is counted twice (once from each endpoint)
            actual_edges //= 2

            if actual_edges == required_edges:
                complete_count += 1

    return complete_count
```