EECS2101 Assignment4 Solution

Huanrui Cao 219256809 saikoro@my.yorku.ca

Problem 1 :

```
Algorithm removeSubMap(T, k1, k2):
    Input: Binary Search Tree T, key range k1 to k2
    Output: None

    // Helper method to remove the node with the minimum key in the given
subtree
    Method removeMin(node):
        if node.left is null:
            return node.right
        node.left ← removeMin(node.left)
        return node

    // Recursive method to remove entries within the specified key range
    Method removeSubMapRec(node, k1, k2):
        if node is null:
            return null
        // Recursively remove from the left subtree
        node.left = removeSubMapRec(node.left, k1, k2)

        // Check if the current node's key is within the specified range
        if k1 <= node.key <= k2:
            // Remove the current node
            if node.left is null:
                return node.right
            else if node.right is null:
                return node.left
            // Node has two children, replace with the minimum node from the
right subtree
            temp ← node
            node ← removeMin(temp.right)
            node.right ← removeMin(temp.right)
            node.left ← temp.left

        // Recursively remove from the right subtree
        node.right ← removeSubMapRec(node.right, k1, k2)
        return node

    // Call the recursive method starting from the root
    T.root ← removeSubMapRec(T.root, k1, k2)
```

Explanation:

- The `Algorithm` keyword is used to declare the algorithm `removeSubMap`.
- The `Method` keyword is used to declare the helper methods `removeMin` and `removeSubMapRec`.
- The algorithm employs a top-down recursive approach to remove entries within the specified key range.
- The `removeSubMapRec` method recursively traverses the tree, removing nodes whose keys fall within the specified range.
- The `removeMin` method is a helper method to remove the node with the minimum key in a given subtree.
- The removal operation is based on the comparison of keys with the specified range (k_1 to k_2).
- The time complexity is analyzed to be $O(s + h)$, where s is the number of entries removed, and h is the height of T.

Problem 2 :

```
Algorithm matchNutsAndBolts(nuts, bolts, low, high):
    Input: Arrays nuts and bolts, low and high are indices
    Output: None (arrays nuts and bolts are matched in place)

    if low < high:
        // Partition nuts array using the bolt at a random index
        pivotIndex ← partition(nuts, bolts[randomIndex(low, high)], low, high)

        // Partition bolts array using the nut at the pivot index
        partition(bolts, nuts[pivotIndex], low, high)

        // Recursively match the remaining nuts and bolts
        matchNutsAndBolts(nuts, bolts, low, pivotIndex - 1)
        matchNutsAndBolts(nuts, bolts, pivotIndex + 1, high)

Algorithm partition(arr, pivot, low, high):
    Input: Array arr, pivot element, low and high are indices
    Output: Index of pivot element after partitioning

    // Locate the index of the pivot element in the array
    pivotIndex ← findIndex(arr, pivot)

    // Swap the pivot element with the element at the high index
    swap(arr[pivotIndex], arr[high])
    // Perform partitioning similar to QuickSort
    i ← low
    for j = low to high - 1:
        if compare(arr[j], pivot) < 0:
            swap(arr[i], arr[j])
            i ← i+1

    // Swap the pivot element back to its correct position
    swap(arr[i], arr[high])

    return i

Algorithm findIndex(arr, element):
    Input: Array arr, element to find
    Output: Index of the element in the array

    for i = 0 to arr.length - 1:
        if compare(arr[i], element) == 0:
            return i

    // Element not found (error handling)

Algorithm compare(a, b):
    Input: Two elements a and b
    Output: -1 if a < b, 0 if a == b, 1 if a > b

    // Comparison logic based on thread sizes
    // (Implementation depends on the specific type of nuts and bolts)
```

Explanation:

- The `matchNutsAndBolts` algorithm is a divide-and-conquer approach.
- It partitions the nuts array based on the bolt at a random index.
- Then, it partitions the bolts array using the nut at the pivot index obtained in the previous step.
- The process is repeated recursively for the subarrays until all nuts and bolts are matched.
- The `partition` algorithm is similar to the partitioning step in QuickSort, which efficiently places the pivot in its correct position.
- The `findIndex` algorithm locates the index of an element in the array.
- The `compare` algorithm is a placeholder for the actual comparison logic based on thread sizes. The implementation depends on the specific type of nuts and bolts.

Time Complexity:

- The time complexity of this algorithm is expected to be efficient, similar to QuickSort, and is expected to be close to $O(\text{n log n})$ in average and best cases.
- The random choice of pivot indices helps avoid worst-case scenarios frequently encountered in traditional QuickSort.

Note: The actual comparison logic (`compare` function) should be implemented based on the specific characteristics of nuts and bolts.

Problem 3:

Part (a) - Proving Inequalities:

Claim: For every graph G, radius≤diameter≤2×radiusradius≤diameter≤2×radius.

Proof:

1. **Radius is less than or equal to Diameter (radius≤diameterradius≤diameter):**
   - The radius of a graph is the minimum eccentricity, which is the minimum distance from a vertex to the farthest vertex.
   - The diameter of a graph is the maximum eccentricity, which is the maximum distance from any vertex to the farthest vertex.
   - It's evident that the minimum distance from any vertex can be achieved when considering a vertex that achieves the maximum distance (farthest vertex). Therefore, the radius is less than or equal to the diameter.
2. **Diameter is less than or equal to Twice the Radius (diameter≤2×radiusdiameter≤2×radius):**
   - Let u and v be two vertices such that $\text{seccentricity}(u) = \text{radiu}$ and $\text{eccentricity}(v) = \text{diameter}$.
   - By the triangle inequality, `distance(u,v)≤eccentricity(v)`.
   - Since `eccentricity(v)=diameter` and `eccentricity(u)=radius`, we have `distance(u,v)≤diameter`.
   - Combining the above results, $\text{diameter} \leq 2 \times \text{radius}$.

Conclusion: The inequalities `radius≤diameter≤2×radius` are proven.

Part (b) - Pseudo-code for Graph Instance Methods:

```
Algorithm eccentricity(Graph G, Vertex v):
    // Use BFS to find the eccentricity of vertex v in graph G
    Initialize an empty queue Q
    Enqueue (v, 0) into Q  // Vertex v at distance 0
    Create a set visited and add v to it
    Initialize eccentricity to 0

    while Q is not empty:
        (current, distance) ← Dequeue from Q
        eccentricity ← distance  // Update eccentricity at each step

        for neighbor in neighbors of current:
            if neighbor not in visited:
                Enqueue (neighbor, distance + 1) into Q
                Add neighbor to visited

    return eccentricity
```

```
Algorithm diameter(Graph G):
    // Find the diameter of graph G
    Initialize maxEccentricity to 0

    for each vertex v in G:
        eccentricity_v ← eccentricity(G, v)
        maxEccentricity ← max(maxEccentricity, eccentricity_v)

    return maxEccentricity

Algorithm radius(Graph G):
    // Find the radius of graph G
    Initialize minEccentricity to infinity

    for each vertex v in G:
        eccentricity_v ← eccentricity(G, v)
        minEccentricity ← min(minEccentricity, eccentricity_v)

    return minEccentricity

Algorithm center(Graph G):
    // Find a center vertex of graph G
    Initialize minEccentricity to infinity
    Initialize centerVertex to null

    for each vertex v in G:
        eccentricity_v ← eccentricity(G, v)

        if eccentricity_v < minEccentricity:
            minEccentricity ← eccentricity_v
            centerVertex ← v

    return centerVertex
```

Time Complexity Analysis:

- The time complexity of each method is $O(V \times (V + E))$ where V is the number of vertices and E is the number of edges.
- This is because each BFS traversal takes $O(V + E)$ time, and we perform it for each vertex in the graph.
- The overall complexity of all four methods is $O(V^2 + V \times E)$, where V^2 accounts for the nested traversal over all vertices.

Problem 4:

To solve the problem of finding the maximum bandwidth path between two switching centers a and b in a telephone network, we can adapt Dijkstra's algorithm by considering the minimum bandwidth encountered so far in the path. Here's the pseudo-code for the algorithm:

```
Algorithm maxBandwidthPath(Graph G, Vertex a, Vertex b):
    // Adaptation of Dijkstra's algorithm to find the maximum bandwidth path
    Initialize a priority queue Q with elements (vertex, minBandwidth)
    Initialize a map distanceMap to store the minimum bandwidth for each vertex
    Enqueue (a, infinity) into Q  // Start from vertex a with infinite bandwidth
    Set distanceMap[a] to infinity

    while Q is not empty:
        (current, minBandwidth) ← Dequeue from Q

        if current is b:
            // Found the target vertex b, return the minimum bandwidth encountered
            return minBandwidth

        for neighbor, edgeBandwidth in neighbors of current:
            // Update minimum bandwidth encountered for the neighbor
            newMinBandwidth ← min(minBandwidth, edgeBandwidth)

            if newMinBandwidth > distanceMap[neighbor]:
                // If a higher bandwidth is found, update the minimum bandwidth for the neighbor
                distanceMap[neighbor] ← newMinBandwidth
                Enqueue (neighbor, newMinBandwidth) into Q

    // If the target vertex b is not reached, there is no path from a to b
    return "No path from a to b"
```

Time Complexity Analysis:

- The time complexity of the algorithm is determined by the number of edges and vertices in the graph.
- In the worst case, each edge is considered once, and each vertex is enqueued once, resulting in a time complexity of $O(V + ElogV)$ using a suitable priority queue implementation.
- The logarithmic factor comes from the priority queue operations.

Note:

- The algorithm uses a priority queue to efficiently select the vertex with the highest minimum bandwidth for exploration.
- The use of minimum bandwidth ensures that we explore paths with the maximum bandwidth encountered so far.

Please note that this algorithm assumes positive bandwidth values for the edges and may not be suitable if negative bandwidths are allowed. If negative bandwidths are possible, additional considerations may be needed.