CS314 Fall 2025 Exam 3 Solution and Commentary

A tough test. But not awful.

Text in parenthesis not required. It is simply grading guidance and / or a brief explanation for answer.

A. O(N) (The classic dependent loop that many mistake as O(NlogN). Yes, get is average case O(N)< but this isn't average case. How many nodes do we visit? 1 + 2 + 4 + 8 + … + N/4 + N/2 = N - 1 total work.)

B. 10

C. 5

D. 90 seconds

E. 44 seconds

F. 2

G. 175 (The file has 15 distinct values. The PEOF does not occur in the file. That is added later. The Huffman tree was 16 leaves not 15. Many folks missed this and that is why we curve and bump scores if necessary.)

H. 001 (The TAs had to point out I picked 67. Completely unintentional. The nice thin about his is it didn't matter if you added the PEOF or not. Does not affect the answer if followed the algorithm from assignment 10.)

I. A, D (can do addAll in terms of add. Can do intersection by getting union and removing all of the elements in the differences. Can do size by counting up the number of elements with the iterator. We can't implement add and iterator but we can call those methods in other methods.)

J. O(V) (I am surprise this had a lower average than question K. Recall we just need to loop through the vertices, O(N) as they are stored in a HashSet, and find which adjacent list is the largest.)

K. O(VE) (This is not as fast. Need to loop through all of the vertices and for each loop through its edges and increment the scratch of the destination of the edge. Can track the max as we go.)

L. 14

M. CBIH

N. 24 (Recall Prim's algorithm. Pick an arbitrary Vertex to be the root. Next node is the one that is not part of the tree with the lowest cost edge from the tree. A -> G 3, G -> E 1, E -> H 2, H -> I 2, I -> C 4, C -> B 2, I -> F 6, F -> D 4
3 + 1 + 2 + 2 + 4 + 2 + 6 + 4 = 24)

O. O(N)

P. A

Q. O(1)

R. A
S. C
T. 6
U. D
V. 5
W. 9
X. D
Y. 12
Z. 62

2. Comments: A tough graph question for sure. Lots to do. But a lot of guidance was given. Not enough to just to a neighbor of neighbors check. Have to do a whole breadth first search as the question said. Note, if we start at Vertex v1 if we ever get to Vertex v2 we know the edge in question is not a bridge and can stop. Avoid unnecessary computation.

```
    // Returning false as soon as we visit v2
    public boolean isBridge(String v1, String v2) {
        clearAll();
        Vertex start = vertices.get(v1);
        Vertex other = vertices.get(v2);
        // Remove the edge(s). Actually, only need to remove one
        // as setting start scratch to 1 will avoid going back.
        Edge removedEdge = start.removeEdge(vertices.get(v2));
        start.scratch = 1;
        Queue314<Vertex> queue = new Queue314<>();
        queue.add(start);
        while (!queue.isEmpty()) {
            Vertex current = queue.dequeue();
            for (Edge e : current.adjacent) {
                Vertex dest = e.dest;
                // Make sure we have not already visited this node.
                if (dest.scratch == 0) {
                    /* If this is the other Vertex of the Edge in
                       question, it is still connected to start
                       and we know our answer! */
                    if (dest == other) {
                        start.adjacent.add(removedEdge); // Add edge back
                        return false;
                    }
                    // Otherwise a Vertex other than the first 2.
                    dest.scratch = 1;
                    queue.enqueue(dest);
                }
            }
        }
        // Add the edge back.
        start.adjacent.add(removedEdge);
        return true;
    }


/* Counting the number of vertices we visit. A common solution, but
          this is -3 for not returning when answer know. */
    public boolean isBridge(String v1, String v2) {
        clearAll();
        Vertex start = vertices.get(v1);
        // Remove the edge(s). Actually, only need to remove one
        // as setting start scratch to 1 will avoid going back.
        Edge removedEdge = start.removeEdge(vertices.get(v2));
        start.scratch = 1;
```

```
        int vertexCount = 1;
        Queue314<Vertex> queue = new Queue314<>();
        queue.add(start);
        while (!queue.isEmpty()) {
            Vertex current = queue.dequeue();
            for (Edge e : current.adjacent) {
                Vertex dest = e.dest;
                // Make sure we have not already visited this node.
                if (dest.scratch == 0) {
                    vertexCount++;
                    dest.scratch = 1;
                    queue.enqueue(dest);
                }
            }
        }
        // Add the edge back.
        start.adjacent.add(removedEdge);
        return vertexCount < vertices.size();
    }
```

Rubric:

-1 Does not call clearAll() at start as necessary. Instance method, so NOT Graph.clearAll()
-1 Does not handle start vertex correctly. (pull from map, add to queue. Lose if try to reuse v1 and / or v2.)
-1 does not remove and store edge correctly. Can be both edges or just one if start vertex marked as visited. (scratch set) (If lose this, likely lose restore as well.)
-1 Does not create queue correctly.** `<Vertex> not <E>`** Must be a Queue314 not a LinkedList.
-1 Loop to do search missing or incorrect. While loop for BFS.
-1 Does not dequeue next Vertex from queue correctly.
-3 Does not return false when other vertex of target edge (v2 typically) visited. Must correctly compare String to String OR Vertex to Vertex. Unnecessary computation.
-3 Failure to avoid going back to vertices we have already visited. (Typically, by checking scratch. Unnecessary computation. Possible infinite loop.)
-2 Does not restore edge(s). Must also do if has correct return when answer know in loop. Lose if use Edge constructor. Edge constructor not given for question.
-1 No return. Lose if incorrect value.
-3 **OTHER** Uses other data structures, creates more than 1 queue. OR iterator
-3 **OTHER** Does not perform breadth first search AND / OR adds all vertices at start.
-3 **OTHER** Loops through map keySet. keySet method disallowed. Not necessary.
-2 **OTHER** repeat the BFS after restoring edges. Unnecessary computation.
NOTE No BFS typically -10 when rubric applied. Typically a nested for loop that just checks neighbors and neighbors of neighbors of v1 and v2.
-8 Recursive implementation

3. Comments: A very simple binary search tree problem. Folks did very well. However, since it is a binary search tree we may not have to look at every node. Just like getKth, getAllLesssThan, and getAllGreater than from assignment 9.

```
    public int numInRange(int low, int high) {
        return help(root, low, high);
    }

    private int help(IntNode n, int low, int high) {
        if (n == null) {
            // fell off tree
            return 0;
        } else if (n.val < low) {
            // Value in node less than low. Everything
            // in my left subtree less than low too, no need
            // to check there. Just check right subtree.
            return help(n.right, low, high);
        } else if (n.val > high ) {
         // Value in node greater than high. Everything
            // in my right subtree greater than high too, no need
            // to check there. Just check left subtree.
            return help(n.left, low, high);
        } else {
            // I must be in range. That is one value.
            // Need to check left and right subtrees.
            return 1 + help(n.left, low, high) + help(n.right, low, high);
        }
    }
// Alt, more explicit version of help
private int help(IntNode n, int low, int high) {
    if (n == null) {
        return 0;
    }
    int result = 0;
    // Am I in bounds?
    if (n.val >= low && n.val <= high) {
        result++;
    }
    // Do we need to go to the left? Only if n.val >= low
    if (n.val > low) {
        result += help(n.left, low, high);
    }
    // Do we need to go to the right? Only if n.val <= high
    if (n.val < high) {
        result += help(n.right, low, high);
    }
    return result;
}
```

Rubric:
-2 No helper method OR incorrect call to helper.
-3 cumulative sum variable as parameter OR other unnecessary parameters
-3 base case incorrect (If NPE due to empty, lose this)
-2 Incorrect test for current node value in bounds OR does not correctly add 1 to result if in bounds
-4 Visits too many nodes. No or incorrect checks to determine if necessary to go left or right. Unnecessary computation.
-2 In correct recursive call to left and / or right child. (Can still earn credit even if unnecessary work.)
-1 incorrect or missing return
-4 **OTHER** Doesn't visit enough nodes. Only recurses if current in range.
-4 **OTHER** Use disallowed methods or data structures

4. Comments: I love this problem. So, so many things involved that we talked about during the course and I hope you learned.

- hash table with internal array of lists that serve as the buckets.
- creating a new class with the necessary instance variables
- uses arrays, lists, and iterators
- implementing an iterator yet again! But the implementation details are so different than GLIterator or LLIterator.

By **FAR** the simplest thing to do was track 3 things:

- How many elements have we returned (the iterator shall return size elements at most(OR how many elements do we need to return and decrement that.
- The current list iterator
- the index in the internal array of the hash table, con

Common problems:

- Do an O(N) search for the first non empty bucket even if size == 0
- Using disallowed methods from List such as get and remove. I was evaluating your ability to use iterators.
- Conflating the index in the array with the number of elements we are returning or have returned
- not making instance variables private
- not decrementing the out size in the remove method. You should have been familiar with this requirement from the linked list assignment
- not finding the second bucket correctly
- awfully complicated approaches to the problem

```
private class HIterator implements Iterator<E> {

    private int numToReturn;
    private int bucketIndex;
    private Iterator<E> currentIterator;

    private HIterator() {
        numToReturn = size;
        if (numToReturn > 0) { // Find the first bucket.
            findNextBucket();
        }
    }

    // Either we have not found the first bucket
    // and / or we have used up the current bucket
    // and need to find the next one. Assumes hasNext is true.
    private void findNextBucket() {
        while (con[bucketIndex] == null) {
            bucketIndex++;
        }
        currentIterator = con[bucketIndex].iterator();
```

```
        }

        public boolean hasNext() {
            return numToReturn > 0;
        }

        public E next() {
            numToReturn--;
            if (!currentIterator.hasNext()) {
                // go on to the next element in the array
                bucketIndex++;
                findNextBucket();
            }
            return currentIterator.next();
        }

        public void remove() {
            currentIterator.remove();
            size--;
            if (con[bucketIndex].size() == 0) {
                con[bucketIndex] = null;
            }
        }
    }
```

Rubric:

-1 No attempt for instance variables
-2 Problem with instance variables and / or constructor. (Example assuming arrays have iterators. Instance variables not private.)
-3 hasNext incorrect and / or not O(1)
-3 next - Issue with finding and / or using inner List iterators. Example: Does not correctly find next non-null bucket when required, Creating multiple unnecessary instances of iterators.
-2 next - does not return correct element
-2 remove - does not correctly remove element from underlying bucket via current bucket iterator. Lose if remove O(N).
-2 remove - does not correctly null out bucket if now empty and / or causes Null Pointer Exception and / or AIBOE
-2 **OTHER** potential unsafe calls on List iterators.
-3 **OTHER** Treating an array like a list. (size method, calls to get, get an iterator for array)
-2 **OTHER** Disallowed methods from List class, get, remove
-3 **OTHER** O(N) space, making a copy of all elements in has table in another data structure
-2 **OTHER** adding return type to Iterator remove method. It is a void method.
-3 **OTHER** next unnecessarily inefficient. Not tracking current iterator or at least index in con.
-3 **OTHER** Altering outer size in next method.