Data Structures Exam 2 Review

Question 1

You are given the following classes:

```
Class LLNode {

    String key, String value;
    Int hashCode; LLNode next;

    LLNode(String K, String V, int h, LLNode nxt) {

        Key = key; value = value;
        hashCode = h; next = nxt;

    }
}

Class Hashtable{

    LLNode [] table;
    Int numValues;
    float loadFactor;
    HashTable(float LoadFactor) {....}
}
```

(a)
Implement a function to insert a key-value pair into the hash table, using the function h **mod** N to map a hash code h to a table location. N is table size. Calls rehash if load factor is exceeded. Note: String implements a HashCode() method which returns the hash code of a string

```
Public void insert (String key, String value) {

        Int hashCode = key.HashCode();
        Int index = hashCode % table.length;
        LLNode elem = new LLNode();
        elem.key = key;
        elem.value = value;
        elem.hashCode = hashCode;

        elem.next  = table[index];
        Table[index] = elem;
```

```
        numValues++;

        If ( numValues/table.length  > loadFactor) {

                rehash();
        }
}
```

(b)
Also implement a rehash method, which doubles the table size when expanding it. Your implementation **MUST NOT** end up creating any new nodes- it should only recycle the nodes already in the table.

```
Void rehash () {

        LLNode oldtable = table;
        Int capacity = oldtable.length*2;
        LLNode [] newTable = new LLNode[capacity];
        table = newTable;

        for ( int i = 0; i < oldtable.length; i++ ) {

                LLNode e = oldtable[i];

                While (e!=null) {

                        Int hashCode = e.key.HashCode();
                        Int index = hashCode % capacity;
                        LLNode temp = e.next;
                        e.next = newTable[index];
                        newTable[index] = e;
                        e = temp;

                }

        }


}
```

(c)
Suppose you insert 125 integer key into a hash table with an initial capacity of 25 and a load factor threshold of 2. The hash code is the key itself, and the function key mod table capacity is used to map a key to a table position. Derive the total units of time that will be used to insert all keys. Only counting one unit of time each to do the mapping, insert an entry into a linked list, and check load factor against threshold. Assume a rehash doubles the table capacity, the load factor is checked after an entry is mapped and inserted into a chain. Show work

51 mappings, 51 inserts, 51 loadFactorCheck

Rehash: 51 mappings, 51 inserts size = 50 elementsLeft = 74

50 mappings, 50 inserts, 50 loadFactorCheck

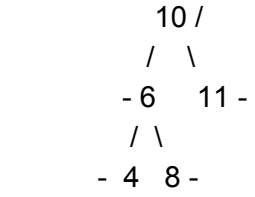
Rehash: 101 mapping, 101 inserts size = 100 elementsLeft = 24

24 mappings, 24 inserts, 24 loadFactorCheck

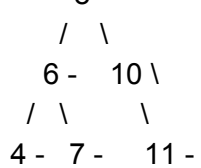
Total operations = 679

Question 2

Draw a 5 node AVL-tree with integers at the nodes, to which adding an integer results in the root being unbalanced, needing two rotations to rebalance. Show the original tree with integers at the nodes, mark the balance factors at the nodes, show where the insertion is done, fix balance factors as needed. Then apply rotation at the root, one rotation at a time, and show the final tree with the integers and balance factors.



Inserting 7 into this tree will result in the tree being unbalance. I will draw the final tree but I want you guys to do the rotations yourselves.



Question 3

Assuming you have two binary search trees A and B (The are not AVL trees). BST A has x nodes and BST B has Y nodes. Assume there are no duplicates in the entire set of A and B. Describe the fastest algorithm to output the combined elements of A and B in sorted order. You may use an array but no other data structure .

Algorithm: Inorder traversal of BST A and append the elements into an array. Time O(x)
Inorder traversal of BST B and append the elements into another array. Time O(y)
Using the two pointer algorithm, output elements in both array in ascending order.Time O(x+y)

Total runtime is the biggest, so O(x+y)

```
      |
4 6 10 15
3 5  11 20
       |
3 4 5 6 10 11 15 20
```

Question 4

Assuming we have the same Binary Search Trees from above and we are trying to find common items in both trees, derive the running time for:

(a) We will perform an inorder traversal of A, and for each item encountered, we will perform a search in B. What is the Big-O running time of this approach.

Worst case of a single search in B is O(y)
We have x elements in A and for each of those elements, we are doing a search in B.

O(XY)

(b) We will perform an inorder traversal of A, appending the elements to an output array as they are encountered. We will do the same for B, appending to a second output array. We will then find the common elements in both arrays. What is the worst case Big-O running time ?

Similar to analysis of question 3

Question 5

Given the following set of character-probability pairs.

(B, 0.07), (R, 0.13), (M, 0.20), (S, 0.27), (E, 0.33)

Show the final huffman tree

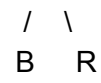
Queue : (B, 0.07), (R, 0.13) , (M, 0.22), (S, 0.25), (E, 0.33) (after all elements are enqueued)

Tree Queue: null

Note: In order to maintain a consistent tree shape, make sure that the element with the higher frequency goes on the right and the element with the lower frequency goes on the left (this also applies to merging trees)

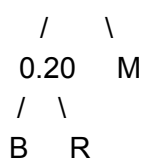
Step 1. Dequeue B and R since they have the lowest frequencies, merge and enqueue into tree queue

Queue: (M, 0.22) , (S, 0.23) , (E, 0.33)

Tree Queue:


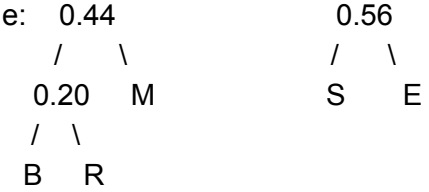
Step 2. Dequeue M and the tree, M goes to the right since it has a larger frequency

Queue: (S, 0.23), (E, 0.33)

Tree Queue:


Step 3: Dequeue S and E and enqueue the resulting tree in the tree queue

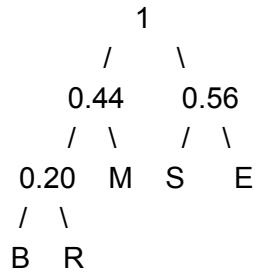
Queue: null

Tree Queue:


Step 4: Dequeue both trees, merge then and enqueue the final tree to the tree Queue

Queue: nul

Tree Queue:



And we are done. Note: if you are asked to label the edges, it is 1 for branching right and 0 for branching left

Part b.
How many total units of time did it take to build the queue. Count enqueue, dequeue, **creating a leaf node**, **creating a new tree out of two subtrees,** and **picking the minimum of two probabilities**. Each operation takes a unit of time. All queues are initially empty.

This question is pretty simple since we already mapped out the steps in the previous question.

Since Both queues are initially empty, we have to first **create leaf nodes** and **enqueue** all nodes to the queue: 5 (create leaf node) + 5 (Enqueue nodes) = 10

Total Dequeues : 8 + 1 (the final dequeue is to return the final Huffman tree from the queue)

Total Enqueues: 4

Total Merges : 4

Pick minimum: 3.

Total = 10 + 9 + 4 + 4+ 3 = 30

Explanation for Pick Minimum:
We do a comparison between the head of the tree queue with the head of the regular queue. If the head of tree queue is greater, we dequeue from regular queue. If dequeued from the regular queue, then we have to do another comparison with the new head of the regular queue before we decide which to dequeue. If we dequeued from the tree queue and it becomes empty then we don't need another comparison, just dequeue the next node in the regular queue. These

comparisons only happens to get the image in step 2 and 3. (use image from previous steps to see what comparisons actually happen)

Example: to get the image in step 2 (using image in step 1 as a reference point)

First compare head of Queue(M) to head of Tree Queue: 1
Since we dequeued from tree queue and it is now empty, we just need to dequeue next node in Queue.

Can you figure out how we got 2 comparisons to make the tree in step 3 ?

Question 6 (K-th Smallest)
(a)
Implement a recursive method that returns the kth smallest node in a bst. The method should throw a NoSuchElementException if K is out of range.

```
Public class BSTNode<T> {

        T data; BSTNode<T> left, right;
        Int leftSize;

}
```

```
Public static T kthSmallest (BSTNode<T> root, int k) throws NoSuchElementException {

                If ( k < 0 || root == null) {

                        throw new NoSuchElementException();
                }


                If ( root.leftSize == k -1) {  //base case

                        return root;
                }
```

```
        If ( root.leftSize >= K ) {

                return kthSmallest(root.left, k);
        }


        else {

                return kthSmallest(root.right, k - root.leftsize -1);
        }

}
```

Try to do this one yourselves

Extra Question: A hash table of initial size (capacity) 5 is set up to store integers. The hash code is the integer itself which is directly mapped to the table using the **integer mod table capacity**. Come up with the smallest dataset of distinct integers that can be inserted with the following conditions (1) After inserting all integers, the worst case number of comparisons for a successful search is strictly less than two and the worst case number of comparison is 4. (2) the LoadFactor threshold 1.5. (3) there must be exactly one rehash when the threshold is exceeded.

Show the hash table just before rehash and after rehash with all entries in it. The worst case and average case requirements only apply to the final Hash Table. What is the average number of comparisons for a successful search on the final Hash Table.