Final Exam Review Questions

The final exam is cumulative including material from Test 1 and Test 2 as well as Huffman Coding and graphs.

PROBLEM 1:  (Short ones)

1. For each of the following methods and data structures in the matrix below, give the Big-OH. If worst-case and average-case are different, note that.

<table>
<thead>
<tr>
<th></th>
<th>search</th>
<th>insert</th>
<th>delete</th>
<th>findMin</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsorted ArrayList</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sorted ArrayList</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LinkedList</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TreeSet</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HashSet</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PriorityQueue</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trie</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2. True or False State whether the following statement is true or false. If false, you should give a specific counterexample.

I. A certain hash table contains $N$ integer keys, all distinct, and each of its buckets contains at most $K$ elements. Assuming that the hashing function and the equality test require constant time, the time required to find all keys in the hash table that are between $L$ and $U$ is $O(K \times (U - L))$ in the worst case.

II. Instead of using a heap, we use a balanced tree to represent a priority queue. The worst-case big-Oh of add (insert) and poll (deleteMin) do not change.

III. Instead of using a heap, we use a sorted ArrayList to represent a priority queue. The worst-case big-Oh of add and poll do not change.

IV. Given the preorder and postorder traversals of a binary tree (i.e. printing out all of the elements but not the null nodes), it is possible to reconstruct the original tree.

V. Given the preorder and inorder traversals of a binary tree, it is possible to reconstruct the original tree.
3. Suppose that a flood fill starts at the point marked with an o in this diagram:

```
XXXXXXXXXX
XX XXXX This is row number 1
XX XX XXXX This is row number 2
XX o XXX This is row number 3
XXXXXXXXXX
```

Suppose that the first recursive call is always left; the second recursive call is always right; the third recursive call is always up; the fourth recursive call is always down. The pseudocode for the method is below.

```java
flood_fill(row, col)
    if (! out-of-bounds && ! visited )
        mark row,col as visited
        fill row,col
        flood-fill(row, col-1)
        flood_fill(row, col+1)
        flood_fill(row-1, col)
        flood_fill(row+1, col)
```

How will the rows be completely filled?

A. Row 1 is filled first, then row 2, then row 3
B. Row 1 is filled first, then row 3, then row 2
C. Row 2 is filled first, then row 3, then row 1
D. Row 3 is filled first, then row 1, then row 2
E. Row 3 is filled first, then row 2, then row 1
PROBLEM 2:  (Bits)

You would like to implement the set operations for the integers 0-31 using a BitSet class. The ith bit is a 1 if and only if i is in the set. For example, 00000000000000000000000000000000 indicates that there are no elements in the set. 10000000000000000000000000000000 indicates that 1, 3, and 31 are in the set.

The second set can be created with the following client code:

    BitSet s = new BitSet();
    s.set(1, true);
    s.set(3, true);
    s.set(31, true);

Below, we have given the constructor and the set method for BitSet.

```java
public class BitSet {
    private int myBits;
    private final static int BITS_PER_INT = 32;

    public class BitSet(int bits) {
        // set all bits to 0
        myBits = 0;
    }

    /**
     * Sets the bit at the specified index to the specified value.
     * @param bitIndex a bit index that should be between 0 and 31
     * @param value a boolean value to set.
     */
    public void set(int bitIndex, boolean value) {
        if (bitIndex < 0 || bitIndex >= BITS_PER_INT) // out of bounds
            return;
        if (value) //
            myBits = myBits | (1 << bitIndex); // set bit to on
        else
            myBits = myBits & ~(1 << bitIndex); // set bit to off
    }
}
```
A. Complete the `get` method. In the above example `s.get(3)` should return `true` while `s.get(7)` should return `false`.

```java
/**
 * Returns the value of the bit with the specified index. The value
 * is true if the bit with the index bitIndex is currently set in
 * this BitSet; otherwise, the result is false.
 * @param bitIndex the bit index.
 * @return the value of the bit with the specified index.
 */
public boolean get(int bitIndex) {
    // TODO: Complete get
```

B. You would like to implement the union operation for the `BitSet` class. The union operation (∪) says that if $A = (1, 2, 3, 4, 5)$ and $B = (1, 3, 5, 7, 9)$, then $A \cup B = (1, 2, 3, 4, 5, 7, 9)$. Write the method `union` below.

```java
/**
 * Performs a union of this bit set with the bit set argument. This bit set is modified so that
 * a bit in it has the value true if and only if it either already had the
 * value true or the corresponding bit in the bit set argument has the value true.
 * @param set a bit set.
 */
public void union(BitSet set) {
```
The Huffman compression algorithm uses a tree to encode the codewords, where each node has either two or zero children. Someone has given you a tree that contains some nodes with only one child.

A. Why can such a tree not be created using the Huffman encoding algorithm discussed in class?

B. Write a function called `tighten` that given such an encoding tree will remove those nodes with one child. The diagram below shows a “loose” tree on the left and its tightened equivalent on the right. The three shaded nodes are the ones that were removed.

Use the definition of a Huffman `TreeNode` below.

```java
/**
 * remove nodes with one child from Huffman tree
 * @param root is the root of a Huffman tree (may be null)
 * @return tree where nodes with one child are removed
 */
public static TreeNode tighten(TreeNode root) {
```
C. State the recurrence and the big-Oh for your solution.

D. You would like to check to make sure the character counts in the tree sum up correctly. That is the weight field of a node should equal the sum of the weights of its children. Write `validWeights` that checks to see if each node is the sum of its children.

```java
/**
 * @return true iff each internal node's weights is the sum of its children's weights
 */
public static boolean validWeights(TreeNode root)
{
```
PROBLEM 4:  *(Huff)*

A. Fill in the following table with dictionary entries that could be produced for a Huffman encoding of the following string.

```
accctggccccccg
```

<table>
<thead>
<tr>
<th>Char</th>
<th>Codeword</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
</tr>
<tr>
<td>g</td>
<td></td>
</tr>
<tr>
<td>t</td>
<td></td>
</tr>
</tbody>
</table>

B. Given the following 21 character string,

```
aaccccacttgggttttccgg
```

are the following 43 bits a possible Huffman encoding of the message above?

```
0000001111000101010010010010101010111001001
```

Justify your answer as concisely and rigorously as possible.
PROBLEM 5:  *(store e-sort)*

A. The code below correctly sorts an array of String values. What is the big-Oh complexity of this code? Justify your answer. The `java.util.PriorityQueue` class is implemented using a heap.

```java
public static void psort(String[] list){
    PriorityQueue<String> pq = new PriorityQueue<String>();
    pq.addAll(Arrays.asList(list));
    int index = 0;
    while (pq.size() != 0){
        list[index++] = pq.poll();
    }
}
```
B. The code below sorts an array of Strings representing DNA by creating arrays for every possible 4-character prefix, e.g., "aaaa" "aaag" "aat" ... "agtc" ... "gatt" ... "tttt"; sorting each of these arrays, and then combining these sorted arrays together.

```java
public static void dnasort(String[] dna){
    Map<String,ArrayList<String>> prefixMap = new TreeMap<String,ArrayList<String>>(());
    for(String s : dna){
        String prefix = s.substring(0, 4);
        ArrayList<String> list = prefixMap.get(prefix);
        if (list == null){
            list = new ArrayList<String>();
            prefixMap.put(prefix, list);
        }
        list.add(s);
    }
    for(ArrayList<String> list : prefixMap.values()){  
        Collections.sort(list);
    }
    ArrayList<String> combined = new ArrayList<String>();
    for(ArrayList<String> list : prefixMap.values()){  
        combined.addAll(list);
    }
    System.arraycopy(combined.toArray(new String[0]), 0, dna, 0, dna.length);
}
```

The code above is faster than calling `Arrays.sort(dna)` for an array of one-million strings representing dna strands (e.g., all the strings contain just the characters 'a', 'g', 't', 'c'). The code is slower when sorting an array of one-thousand strands. Explain why.
C. The code below reads in a file of integer values, stores these in arrays and an ArrayList and then sorts these using the java.util sorts. In class we discussed that Arrays.sort for an int[] uses a variant of quicksort whereas both Arrays.sort for Object[] and Collections.sort use a modified merge sort.

In the code below the values in the lists list, alist, ilist are the same before the sorts are called.

```java
public void sortParadise() throws FileNotFoundException{
    ArrayList<Integer> alist = new ArrayList<Integer>();
    Scanner s = new Scanner(new File("demonicintegers.txt"));
    while (s.hasNextInt()){  
        alist.add(s.nextInt());
    }
    int[] list = new int[alist.size()];
    for(int k=0; k < alist.size(); k++){
        list[k] = alist.get(k);
    }
    Integer[] ilist = alist.toArray(new Integer[0]);

    Collections.sort(alist);
    Arrays.sort(ilist);
    Arrays.sort(list);
}
```

When this code is executed the last call to Arrays.sort, when the int[] list array is sorted, results in a stack-overflow error. This is because the file demonicintegers.txt contains a worst-case ordering of 250,000 integers. However, if this line is added after the while loop:

```java
Collections.shuffle(alist);
```

then there is no stack-overflow. Explain briefly why Arrays.sort(ilist) doesn’t generate a stack overflow but Arrays.sort(list) does and why shuffling the elements fixes the “problem”.

**PROBLEM 6 : (Tradeoffs)**

You are given an array of $n$ ints (where $n$ is very large) and are asked to find the largest $m$ of them (where $m$ is much less than $n$).

**A.** Design an efficient algorithm to find the largest $m$ elements.

You can assume the existence of all data structures we discussed in class. You do not have to explain how any of the standard methods (e.g. constructing a heap) work. Be specific, however, about which data structures you are using and how these data structures are interconnected.

Your algorithm should work well for all values of $m$ and $n$, from very small to very large.

**B.** What is the running time of your algorithm? What is it for small $m$? What is it as $m \to n$ (i.e. as $m$ approaches $n$)?
PROBLEM 7: (User Logins)

In a hypothetical world, the OIT system was cracked allowing sensitive data to be leaked. The system’s administrators are able to pinpoint approximately when the security breach occurred, within a certain range of times. They ask you to determine who was logged into the system to help find the culprit.

As part of the logging mechanism, the system automatically records each login and logout of a user. One BST keeps information about user logins each time a user logs into the system, a new item is inserted into the BST with the time as the key. Similarly, when a user logs out of the system, a new item is inserted into a separate BST that records logouts. The BSTs are implemented using TreeNode that store these values as instances of Events. A user may not be logged in more than once at the same time.

Part A:

Given the following definitions of TreeNode and Event, fill in the following method userLogins that returns a List of the usernames that were logged in between the given start time and end time. Times are stored as int values (the number of seconds since the system was turned on). Also, a user is considered to be “logged in” during a range if she did not log out before the start time or log in after the end time. You may assume that no events occur simultaneously.

```java
/* Represents a user login or logout at a certain time */
public class Event {
    String username;
    int time;
    public Event(String s, int t) { username = s; time = t; }
}

public class TreeNode {
    Event info;
    TreeNode left;
    TreeNode right;
    TreeNode (Event e, TreeNode lt, TreeNode rt) {
        info = e;
        left = lt;
        right = rt;
    }
}

/* Computes and returns a List of the usernames that were logged
* into the system between start and end. logins is a TreeNode that
* stores Event instances indicating user logins. logouts is a TreeNode
* that stores Event instances indicating user logouts. */
static List userLogins(int start, int end, TreeNode logins, TreeNode logouts) {
    // FILL IN
```
Part B
After using your code for some time, the system’s administrators return to you with a complaint, ”It works great, but it’s far too slow.” However, the real performance issue lies in the way the system is storing information about logins and logouts. What’s bad about the way the system is storing this information?
Carefully describe its problems and give a tight asymptotic bound on the running time of your implementation of userLogins.

Part C
Propose an optimal means of storage of the information about logins and logouts so that storage is efficient (the system can store quickly and also not waste disk space/memory space) and information retrieval is fast (for operations such as userLogins). Include in your description the data structure you would use and how you would store information about logins and logouts in such a data structure. Remember that login/logout information is not all inserted at once, but rather, is stored as users log in and log out over time.

Part D
Describe how your userLogins method could be rewritten to take advantage of your new efficient data structure.
PROBLEM 8:  *(Think)*

Answer the questions below with a short paragraph for each.

A. Some mobile phones keep a list of \( N \) frequently dialed numbers. When a call is made from the phone’s phone book, the phone increments the number’s call count in the frequently dialed numbers list, potentially moving the number up in the list (indicating that the number is “more frequently dialed”). Suppose the user of such a cell phone looks at this (long) list from the most frequently dialed to least frequently dialed, and often doesn’t look through the entire list. Give a data structure and means of search and update of phone numbers in the list that optimize for speed of operation, based on the user behavior described. Give approximate running times (in terms of \( N \)) using asymptotic notation for search and update of phone numbers using your data structure.

B. Suppose a stack of \( N \) books of different sizes and a sheet of paper are on a table. You are asked to sort the books from smallest (physically) to largest so that the stack doesn’t fall over easily. Since the books are heavy and \( N \) is large, the only thing you can do is pull out one book from anywhere in the stack and place it at the top, but since you have quick eyes, you can immediately determine which book is the smallest or largest in some part of the stack (separated by the paper). You may insert the sheet of paper above or below any book in the stack. Give a set of steps to quickly sort the stack of books (and say where you put the paper, if anywhere). Which sorting algorithm is this similar to? Why? Why is this “quick” for this problem?