Test 2: CompSci 100

Name (print): 

Community Standard acknowledgment (signature): 

<table>
<thead>
<tr>
<th>Problem</th>
<th>value</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem 1</td>
<td>14 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 2</td>
<td>9 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 3</td>
<td>7 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 4</td>
<td>12 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 5</td>
<td>8 pts.</td>
<td></td>
</tr>
<tr>
<td>TOTAL:</td>
<td>50 pts.</td>
<td></td>
</tr>
</tbody>
</table>

This test has 8 pages, be sure your test has them all. Do NOT spend too much time on one question — remember that this class lasts only 50 minutes and there are 50 points on the exam. That means you should spend no more than 1 minute per point.

You may consult your five (5) sheets and no other resources. You may not use any computers, calculators, or cell phones. You may refer to any program text supplied in lectures or assignments.

Don’t panic. Just read all the questions carefully to begin with, and first try to answer those parts about which you feel most confident. Do not be alarmed if some of the answers are obvious.

If you think there is a syntax error or an ambiguity in a problem’s specification, then please ask.

In writing code you do not need to worry about specifying the proper `import` statements. Assume that all libraries and packages we’ve discussed are imported in any code you write.
PROBLEM 1:  (Short ones (14 points))

1. You are creating a boggle word search program and you want to find all valid words where the letters are adjacent. What data structure would be best suited to hold the dictionary (i.e. lexicon)?
   A. Queue  
   B. Heap  
   C. Trie  
   D. AVL Tree  
   E. Hash Table

2. In order to use the class Point containing fields x and y in a HashSet, you are considering multiple hash functions. Of these hash functions, which one would give the best performance in a HashSet? Assume that your points are likely to be between (0, 0) and (1280, 1024) (the size of the average computer monitor).
   A. public int hashCode () { return super.hashCode(); }  
   B. public int hashCode () { return 42; }  
   C. public int hashCode () { return x; }  
   D. public int hashCode () { return x + y; }  
   E. public int hashCode () { return x * 3 + y; }  
   F. public int hashCode () { return x * 1000 + y; }

3. 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 an AVL 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.
**PROBLEM 2 : (Reverse (9 points))**

Each of the Java functions on the left take a string `s` as input, and returns its reverse. For each of the following, state the recurrence (if applicable) and give the big-Oh complexity bound.

Recall that concatenating two strings in Java takes time proportional to the sum of their lengths, and extracting a substring takes constant time.

A. public static String reverse1(String s) {
    int N = s.length();
    String reverse = "",
    for (int i = 0; i < N; i++)
        reverse = s.charAt(i) + reverse;
    return reverse;
}

B. public static String reverse2(String s) {
    int N = s.length();
    if (N <= 1) return s;
    String left = s.substring(0, N/2);
    String right = s.substring(N/2, N);
    return reverse2(right) + reverse2(left);
}

C. public static String reverse3(String s) {
    int N = s.length();
    char[] a = new char[N];
    for (int i = 0; i < N; i++)
        a[i] = s.charAt(N-i-1);
    return new String(a);
PROBLEM 3:  (Sorting (7 points))

The column on the left is the original input of strings to be sorted. The columns to the right are the contents at some intermediate step during one of the 8 sorting algorithms listed below. Match up each algorithm by writing its number under the corresponding column. Use each number exactly once.

<table>
<thead>
<tr>
<th>Jane</th>
<th>Adam</th>
<th>Anna</th>
<th>Will</th>
<th>Jada</th>
<th>Abby</th>
<th>Adam</th>
<th>Abby</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adam</td>
<td>Alex</td>
<td>Adam</td>
<td>Seth</td>
<td>Emma</td>
<td>Adam</td>
<td>Dave</td>
<td>Adam</td>
</tr>
<tr>
<td>Mary</td>
<td>Cole</td>
<td>Abby</td>
<td>Ryan</td>
<td>Ella</td>
<td>Alex</td>
<td>Erik</td>
<td>Alex</td>
</tr>
<tr>
<td>Jeff</td>
<td>Dave</td>
<td>Ella</td>
<td>Sean</td>
<td>Maya</td>
<td>Anna</td>
<td>Erin</td>
<td>Anna</td>
</tr>
<tr>
<td>Erik</td>
<td>Erik</td>
<td>Emma</td>
<td>Mark</td>
<td>Anna</td>
<td>Cole</td>
<td>Evan</td>
<td>Cole</td>
</tr>
<tr>
<td>Dave</td>
<td>Erin</td>
<td>Dave</td>
<td>Noah</td>
<td>Sara</td>
<td>Dave</td>
<td>Jack</td>
<td>Dave</td>
</tr>
<tr>
<td>Evan</td>
<td>Evan</td>
<td>Alex</td>
<td>Owen</td>
<td>Eric</td>
<td>Ella</td>
<td>Jada</td>
<td>Ella</td>
</tr>
<tr>
<td>Sean</td>
<td>Jack</td>
<td>Cole</td>
<td>Sara</td>
<td>Jane</td>
<td>Emma</td>
<td>Jane</td>
<td>Emma</td>
</tr>
<tr>
<td>Erin</td>
<td>Jada</td>
<td>Eric</td>
<td>Hart</td>
<td>Dave</td>
<td>Eric</td>
<td>Jeff</td>
<td>Eric</td>
</tr>
<tr>
<td>Jada</td>
<td>Jane</td>
<td>Jada</td>
<td>Joey</td>
<td>Luke</td>
<td>Erik</td>
<td>Mary</td>
<td>Erik</td>
</tr>
<tr>
<td>Jack</td>
<td>Jeff</td>
<td>Jack</td>
<td>Jack</td>
<td>Kyle</td>
<td>Erin</td>
<td>Noah</td>
<td>Erin</td>
</tr>
<tr>
<td>Noah</td>
<td>Kyle</td>
<td>Noah</td>
<td>Maya</td>
<td>Cole</td>
<td>Evan</td>
<td>Sean</td>
<td>Evan</td>
</tr>
<tr>
<td>Kyle</td>
<td>Mary</td>
<td>Kyle</td>
<td>Kyle</td>
<td>Jeff</td>
<td>Kyle</td>
<td>Kyle</td>
<td>Jack</td>
</tr>
<tr>
<td>Owen</td>
<td>Noah</td>
<td>Owen</td>
<td>Mary</td>
<td>Noah</td>
<td>Owen</td>
<td>Owen</td>
<td>Jada</td>
</tr>
<tr>
<td>Seth</td>
<td>Owen</td>
<td>Seth</td>
<td>Jeff</td>
<td>Seth</td>
<td>Seth</td>
<td>Seth</td>
<td>Seth</td>
</tr>
<tr>
<td>Cole</td>
<td>Sean</td>
<td>Sean</td>
<td>Eric</td>
<td>Leah</td>
<td>Jada</td>
<td>Cole</td>
<td>Jane</td>
</tr>
<tr>
<td>Alex</td>
<td>Seth</td>
<td>Evan</td>
<td>Alex</td>
<td>Josh</td>
<td>Mary</td>
<td>Alex</td>
<td>Jeff</td>
</tr>
<tr>
<td>Hart</td>
<td>Abby</td>
<td>Hart</td>
<td>Erin</td>
<td>Erik</td>
<td>Hart</td>
<td>Hart</td>
<td>Joey</td>
</tr>
<tr>
<td>Mark</td>
<td>Anna</td>
<td>Mark</td>
<td>Jada</td>
<td>Jack</td>
<td>Mark</td>
<td>Mark</td>
<td>John</td>
</tr>
<tr>
<td>Joey</td>
<td>Ella</td>
<td>Joey</td>
<td>Erik</td>
<td>Mark</td>
<td>Joey</td>
<td>Joey</td>
<td>Josh</td>
</tr>
<tr>
<td>Emma</td>
<td>Emma</td>
<td>Erik</td>
<td>Emma</td>
<td>Will</td>
<td>Sean</td>
<td>Emma</td>
<td>Kyle</td>
</tr>
<tr>
<td>Ella</td>
<td>Eric</td>
<td>Jeff</td>
<td>Ella</td>
<td>Adam</td>
<td>Noah</td>
<td>Ella</td>
<td>Leah</td>
</tr>
<tr>
<td>Lily</td>
<td>Hart</td>
<td>Lily</td>
<td>Lily</td>
<td>Evan</td>
<td>Lily</td>
<td>Lily</td>
<td>Lily</td>
</tr>
<tr>
<td>Maya</td>
<td>Jake</td>
<td>Maya</td>
<td>Dave</td>
<td>Sean</td>
<td>Maya</td>
<td>Maya</td>
<td>Luke</td>
</tr>
<tr>
<td>Leah</td>
<td>Joey</td>
<td>Leah</td>
<td>Leah</td>
<td>Erin</td>
<td>Leah</td>
<td>Leah</td>
<td>Mark</td>
</tr>
<tr>
<td>Abby</td>
<td>John</td>
<td>Mary</td>
<td>Abby</td>
<td>Owen</td>
<td>Jane</td>
<td>Abby</td>
<td>Mary</td>
</tr>
<tr>
<td>Anna</td>
<td>Josh</td>
<td>Jane</td>
<td>Anna</td>
<td>John</td>
<td>Jeff</td>
<td>Anna</td>
<td>Maya</td>
</tr>
<tr>
<td>John</td>
<td>Leah</td>
<td>John</td>
<td>John</td>
<td>Ryan</td>
<td>John</td>
<td>John</td>
<td>Noah</td>
</tr>
<tr>
<td>Ryan</td>
<td>Lily</td>
<td>Ryan</td>
<td>Evan</td>
<td>Hart</td>
<td>Ryan</td>
<td>Ryan</td>
<td>Owen</td>
</tr>
<tr>
<td>Josh</td>
<td>Mark</td>
<td>Josh</td>
<td>Alex</td>
<td>Josh</td>
<td>Josh</td>
<td>Josh</td>
<td>Ryan</td>
</tr>
<tr>
<td>Jake</td>
<td>Maya</td>
<td>Jake</td>
<td>Jake</td>
<td>Mary</td>
<td>Jake</td>
<td>Jake</td>
<td>Sara</td>
</tr>
<tr>
<td>Sara</td>
<td>Ryan</td>
<td>Sara</td>
<td>Jane</td>
<td>Joey</td>
<td>Sara</td>
<td>Sara</td>
<td>Sean</td>
</tr>
<tr>
<td>Will</td>
<td>Sara</td>
<td>Will</td>
<td>Cole</td>
<td>Lily</td>
<td>Will</td>
<td>Will</td>
<td>Seth</td>
</tr>
<tr>
<td>Eric</td>
<td>Will</td>
<td>Erin</td>
<td>Adam</td>
<td>Abby</td>
<td>Jack</td>
<td>Eric</td>
<td>Will</td>
</tr>
</tbody>
</table>

| ---- | ---- | ---- | ---- | ---- | ---- | ---- |----|

(0) Original input  (1) Heap sort  (2) Insertion sort  (3) Mergesort  
(4) Quicksort  (5) Radix sort  (6) Selection sort  (7) All of them
PROBLEM 4: (Huffman Trees (12 points))

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.

The definition of a Huffman TreeNode is attached to the end of the test.

```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.

PROBLEM 5: (Tradeoffs (8 points))  
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 $m \rightarrow n$ (i.e. as $m$ approaches $n$)?
Throughout this test, assume that the following classes and methods are available. These classes are taken directly from the material used in class. There should be no methods you have never seen before here.

Definitions

Some common recurrences and their solutions.

\[
\begin{align*}
T(n) &= T(n/2) + O(1) \\ O(1) \\
T(n) &= T(n/2) + O(n) \\ O(n) \\
T(n) &= 2T(n/2) + O(1) \\ O(n) \\
T(n) &= 2T(n/2) + O(n) \\ O(n \log n) \\
T(n) &= T(n-1) + O(1) \\ O(n) \\
T(n) &= T(n-1) + O(n) \\ O(n^2)
\end{align*}
\]

List Node

```java
public class Node {
    String info;
    Node next;
    Node(String s, Node link) {
        info = s;
        next = link;
    }
}
```

TreeNode

```java
public class TreeNode {
    String info;
    TreeNode left;
    TreeNode right;
    TreeNode parent;
    TreeNode (String s, TreeNode lt, TreeNode rt, TreeNode p) {
        info = s;
        left = lt;
        right = rt;
        parent = p;
    }
}
```

String

```java
public class String {
    /* Compares this string to the specified object. */
    /* The result is true if and only if the argument */
    /* is not null and is a String object that */
    /* represents the same sequence of characters */
    /* public boolean equals(Object anObject) */

    /* Returns the index within this string of the */
    /* first occurrence of the specified substring. */
    /* "-1 if it does not exist */
    /* public int indexOf(String str) */

    /* Returns the length of this string. */
    /* public int length() */

    /* Returns a new string that is a substring of this */
    /* string. Begins at the specified beginIndex and */
    /* extends to the character at index endIndex - 1 */
    /* public String substring(int beginIndex, int endIndex) */
}
```

TreeSet/HashSet

```java
public class TreeSet {
    public TreeSet() { /* Constructs a new, empty set */
        public Iterator iterator() { /* Returns an iterator over the elements in */
            public int size() { /* ascending order. */
                public boolean contains(Object o) { /* Returns the number of elements in this set. */
                    public boolean add(Object o) { /* Returns true if this set contains o */
                        public Object remove() { /* Adds the specified element to this set */
                            public Object peek() { /* if it is not already present. */
                                public Object poll() { /* Returns the number of elements */
                                    public boolean empty() { /* */
                                        public boolean add(Object o) { /* */
                                            public boolean contains(Object o) { /* */
                                                public Object remove() { /* */
                                                    public Object peek() { /* */
                                                        public int size() { /* */
                                                            public boolean empty() { /* */
                                                                public boolean add(Object o) { /* */
                                                                    public boolean contains(Object o) { /* */
                                                                        public Object remove() { /* */
                                                                            public Object peek() { /* */
                                                                                public int size() { /* */
                                                                    }
                                                            }
                                                        }
                                                }
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
```
Utility binary-tree (Huffman tree) node for Huffman coding.
This is a simple, standard binary-tree node implementing
the comparable interface based on weight.

* @author Owen Astrachan
* @version 1.0 July 2000
*/

public class TreeNode implements Comparable<TreeNode> {

public int myValue;
public int myWeight;
public TreeNode myLeft;
public TreeNode myRight;

/**
 * construct leaf node (null children)
 * @param value is the value stored in the node (e.g., character)
 * @param weight is used for comparison (e.g., count of # occurrences)
 */

public TreeNode(int value, int weight) {
    myValue = value;
    myWeight = weight;
}

/**
 * construct internal node (with children)
 * @param value is stored as value of node
 * @param weight is weight of node
 * @param ltree is left subtree
 * @param rtree is right subtree
 */

public TreeNode(int value, int weight, TreeNode ltree, TreeNode rtree) {
    this(value, weight);
    myLeft = ltree;
    myRight = rtree;
}

/**
 * Return value based on comparing this TreeNode to another.
 * @return -1 if this < o, +1 if this > o, and 0 if this == 0
 */

public int compareTo(TreeNode rhs) {
    return myWeight - rhs.myWeight;
}