This test has 14 pages, be sure your test has them all. Do NOT spend too much time on one question — remember that this class lasts 75 minutes.

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.
Some common recurrences and their solutions.

\[ A \quad T(n) = T(n/2) + O(1) \quad O(\log n) \]
\[ B \quad T(n) = T(n/2) + O(n) \quad O(n) \]
\[ C \quad T(n) = 2T(n/2) + O(1) \quad O(n) \]
\[ D \quad T(n) = 2T(n/2) + O(n) \quad O(n \log n) \]
\[ E \quad T(n) = T(n-1) + O(1) \quad O(n) \]
\[ F \quad T(n) = T(n-1) + O(n) \quad O(n^2) \]
\[ G \quad T(n) = 2T(n-1) + O(1) \quad O(2^n) \]

The class `TreeNode` used on this test.

```java
public static class TreeNode {
    String info;
    TreeNode left;
    TreeNode right;
    TreeNode(String val, TreeNode lptr, TreeNode rptr)
    {
        info = val;
        left = lptr;
        right = rptr;
    }
}
```
PROBLEM 1:  (Stuff (20 points))

Part A (5 points)
A priority queue of $n$ elements implemented using a HashMap yields $O(n)$ complexity for removing the minimal element. Using a TreeMap yields $O(\log n)$ complexity for removing the minimal element. Explain why both statements are true.

Part B (5 points)
Draw each of the five heaps that result from inserting the numbers 20, 15, 7, 25, and 13 into an initially empty heap. The first heap you draw should have just 20 in it. The second should have 20 and 15 in it, the third should have 20, 15, and 7, the fourth should have 20, 15, 7, and 25 in it and the last should have all five values in it.
Part C (10 points)
Method deepLeaf below returns the deepest leaf in a binary tree, i.e., the leaf furthest from the root. What is the complexity of deepLeaf in the average case when trees are roughly balanced? What is the complexity of deepLeaf in the worst case when trees are not balanced? Justify both answers.

```java
public int height(TreeNode t){
    if (t == null) return 0;
    return Math.max(height(t.left),height(t.right)) + 1;
}

public TreeNode deepLeaf(TreeNode t){
    if (t == null) return null;
    if (t.left == null && t.right == null){
        return t;
    }
    if (height(t.left) > height(t.right)){
        return deepLeaf(t.left);
    }
    else { 
        return deepLeaf(t.right);
    }
}
```
Here’s a search tree.

The in-order traversal of the tree is in alphabetical order because it’s a search tree:

apple cherry guava lemon orange papaya tangerine watermelon

Part A (4 points)
Write the pre-order traversal of the tree.

Part B (4 points)
Write the post-order traversal of the tree.

Part C (4 points)
Add a node with banana and a node with persimmon to the tree above so that the tree remains a search tree.
**Part D (8 points)** If each root-to-leaf path is printed, with the paths ordered alphabetically by the leaf in the path, this is what’s printed:

lemon cherry apple  
lemon cherry guava  
lemon papaya orange  
lemon papaya tangerine watermelon

The method `leafPathPrint` below prints these paths by calling an auxiliary, recursive method to do the work. The auxiliary method prints the path when a leaf is reached. The parameter `list` holds the current root-to-node path each time `leafPathPrintAux` is called. Complete the method so that it correctly prints all root-to-leaf paths, one per line as shown above. (Hint: this is similar to exploring paths in the Boggle program: add a value to `list` and remove it as appropriate.)

```java
public void leafPathPrint(TreeNode root){
    ArrayList<String> list = new ArrayList<String>();
    leafPathPrintAux(root,list);
}
/**
 * Print all root-to-leaf paths.
 * @param t is the subtree being visited
 * @param list contains values from root to parent of Node t
 */
private void leafPathPrintAux(TreeNode t, ArrayList<String> list){
    if (t == null) {
        return;    // nothing to print
    }
    else if (t.left == null && t.right == null){
        for(String s : list){
            System.out.print(s+ " ");    // print root-to-parent of leaf
        }
        System.out.println(t.info);    // now print leaf
        return;
    }
    // code here manipulates list and recurses as appropriate
}
```
PROBLEM 3: (Smash, Crash, Hash, Trash (20 points))

In the java.util package there are two implementations of maps based on hash-tables: HashMap and LinkedHashMap. Both use an array of buckets, where each bucket is a linked-list of the key/value pairs in the map that hash (keys) to the bucket. However, the LinkedHashMap maintains a separate linked-list of all the entries in the map, with the entries linked in the order in which they are inserted into the map. For example, the diagram below shows a LinkedHashMap with two buckets. Keys have been inserted into the map in this order:

"apple", "horse", "cow", "bean", "lemon"

The values associated with they keys aren’t shown in the diagram, just the keys and the array of buckets/linked-lists.

Part A (4 points)
Assume the array of buckets in the diagram contains 1003 buckets. Given the illustration, make useful and relevant statements about the values returned by the method hashCode() for the keys "cow", "lemon", and "horse".
Part B (4 points)
In some applications the bucket of linked-lists implementation of a hash-table should be replaced by a bucket of balanced-search trees implementation. In other words, rather than using a linked-list of the entries that hash to the same bucket, use a balanced-search tree of entries that hash to the same bucket. Provide reasons/justification for using a balanced-search tree in some situations, but explain why a linked-list is usually used. Use big-O in your justification.

Part C (4 points)
In the LinkedHashMap class iteration over all keys is an $O(n)$ operation when there are $n$ key/value entries in the map. In the HashMap class iteration over all keys depends on the number of key/value entries and on the size of the hash-table, i.e., on the number of buckets. This means if there are many more buckets than entries iteration for the LinkedHashMap is more efficient. Explain why.
Part D (8 points)
In a LinkedHashMap pointers to the first and last nodes in the linked-list of all insertion-order nodes are kept as shown in the diagram above. The operations put, remove, and get in a hash-table are supposed to execute in $O(1)$ time in the average case. One of these operations would require $O(n)$ time for an n-element map if singly-linked lists are used and $O(1)$ time if doubly-linked lists are used. Which operation necessitates the doubly-linked list for $O(1)$ time and why. Your justification should include why a singly-linked list is $O(1)$ for two operations and why it could be $O(n)$ for another and how a doubly-linked list helps.
Consider sorting a list of student names by the grades each student receives on a test. For example, if the $i^{th}$ name in array `names` has the $i^{th}$ grade given in `scores` (e.g., so that owen has an 75, susan has a 95, and robert has an 80) then the names in order by test score are

susan, jeff, robert, owen, dee

Here are the data

```java
String[] names = {
    "owen", "jeff", "dee", "susan", "robert"
};
int[] scores = {
    75, 90, 70, 95, 80
};

names = sortByScore(names, scores);
names = sortComp(names, scores);
```

Two methods for sorting the list of names are shown below. You’ll be asked to reason about the methods. Method `sortByScore` uses bubble sort and returns a list of sorted names. The method would be called as shown above.

```java
public static String[] sortByScore(String[] names, int[] score){
    for(int j=names.length-1; j > 0; j--){
        for(int k=0; k < j; k++){
            if (score[k+1] > score[k]){  
                int temp = score[k+1];
                score[k+1] = score[k];
                score[k] = temp;
                String stemp = names[k+1];
                names[k+1] = names[k];
                names[k] = stemp;
            }
        }
    }
    return names;
}
```
Here is method `sortComp` that returns an array of names sorted by score.

```java
public static String[] sortComp(String[] names, int[] score){
    final class Combo implements Comparable<Combo>{
        int count;
        String name;
        public Combo(String s, int c){
            name = s;
            count = c;
        }
        public int compareTo(Combo other){
            return other.count - count;
        }
    }
    ArrayList<Combo> list = new ArrayList<Combo>();
    for(int k=0; k < names.length; k++){
        list.add(new Combo(names[k], score[k]));
    }
    Collections.sort(list);
    ArrayList<String> slist = new ArrayList<String>();
    for(Combo c : list){
        slist.add(c.name);
    }
    return (String[]) slist.toArray(new String[0]);
}
```

Provide a justification for preferring the first method shown. Provide a justification for preferring the second method. Both should be valid justifications, i.e., there are scenarios in which either method might be preferred to the other.
PROBLEM 5: (Boggle My Mind (10 points))

Part A (5 points)

In playing regular Boggle letters cannot be re-used within a single word found on the board. The board below illustrates that re-using letters within a word can result in finding more words on a board. The word “TOMATO” cannot be formed unless letters are re-used within a word. Note that both “T” and “O” are used twice in the word shown, although it’s possible to make “TOMATO” re-using only the “T”.

Here’s a correct version of method cellsForWord in class WordFinder for a Boggle program in which letters cannot be re-used.

```
public List<BoardCell> cellsForWord(BoggleBoard board, String word) {
    ArrayList<BoardCell> list = new ArrayList<BoardCell>();
    for (int r = 0; r < board.size(); r++) {
        for (int c = 0; c < board.size(); c++) {
            if (find(word, 0, r, c, board,list)) {
                return list;
            }
        }
    }
    return null;
}
```

This method calls find to do the recursive work. The code below is correct for regular Boggle (except for words with the letter 'Q' which aren’t accounted for). Comment out one line of the method find on the next page so that letters can be re-used thus allowing “TOMATO” for the board shown above.
private boolean find(String word, int index, int row, int col, BoggleBoard board, List<BoardCell> list) {
    if (index >= word.length()) {
        return true;
    }
    if (row < 0 || row >= board.size() || col < 0 || col >= board.size()) {
        return false;
    }
    char currentChar = word.charAt(index);
    if (board.getFace(row, col).charAt(0) == currentChar) {
        BoardCell value = new BoardCell(row, col);
        if (list.contains(value)) return false;
        list.add(value);
        if (find(word, index + 1, row + 1, col, board, list)) return true;
        if (find(word, index + 1, row + 1, col + 1, board, list)) return true;
        if (find(word, index + 1, row + 1, col - 1, board, list)) return true;
        if (find(word, index + 1, row, col - 1, board, list)) return true;
        if (find(word, index + 1, row, col + 1, board, list)) return true;
        if (find(word, index + 1, row - 1, col, board, list)) return true;
        if (find(word, index + 1, row - 1, col + 1, board, list)) return true;
        if (find(word, index + 1, row - 1, col - 1, board, list)) return true;
        list.remove(list.size() - 1);
    }
    return false;
}
Part B (5 points)
In the code on the previous page, the same letter cannot be used twice in a row. For example, the word “MAMMAL” cannot be formed using the code from the previous page because the double “M” isn’t found using the code (the corrected version in which repeats are allowed). However, “MAMMAL” can be found as shown below if the same letter can be repeated without intervening letters.

Assuming the correct line is commented out so that repeated letters are allowed, i.e., you’ve done Part A correctly, the code above can be modified by adding one new recursive call to find so that the same letter can be repeated without intervening letters, thus allowing “MAMMAL” to be found. Indicate what new recursive call can be added, before the call to list.remove, so that “MAMMAL” and other such repeats are found (for example, “ROOT” and “TREE” will both be found with the proper line added). There will now be nine recursive calls.

Please justify briefly why adding the line results in words like “MAMMAL” being found.