Test 2: Compsci 100

Owen Astrachan

April 2, 2008

Name: _____ (2 points)

Login: _____

Honor code acknowledgment (signature) _____

	value	grade
Problem 1	32 pts.	
Problem 2	6 pts.	
Problem 3	22 pts.	
Problem 4	20 pts.	
TOTAL:	80 pts.	

This test has 12 pages, be sure your test has them all. Do NOT spend too much time on one question — remember that this class lasts 75 minutes.

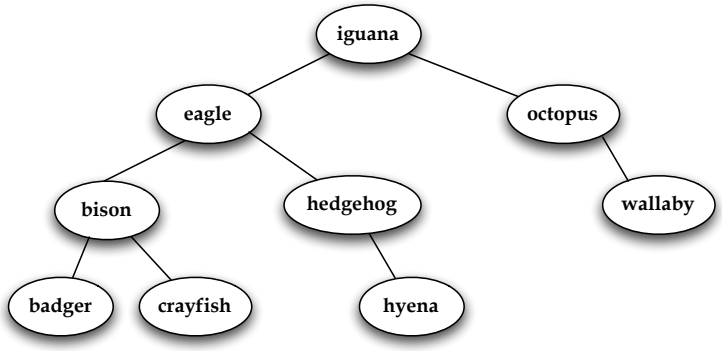
In writing code you do not need to worry about specifying the proper `import statements`. Assume that all libraries and packages we've discussed are imported in any code you write.

Unless indicated otherwise, here's the `TreeNode` class for this test.

```
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 : (*Trees (32 points)*)

The questions in this problem will use the tree below.



Part A (4 points)

The *inorder* traversal of the tree below is:

badger, bison, crayfish, eagle, hedgehog, hyena, iguana, octopus, wallaby.

What is the post-order traversal?

Part B (6 points)

Show where the values *fox*, *koala*, and *zebra* would appear if inserted into the search tree above by adding the nodes in the diagram.

Part C (4 points)

Complete the method below that returns a copy of the tree shown above (in general a copy of any tree).

```
public TreeNode copy(TreeNode root){
    if (root == null)                          // add code



    return new TreeNode(root.info,
                        copy(root.left),



                       );                      // add code


}
```

Part D (5 points)

Write method `oneChildCount` that returns the number of nodes in a tree that have one child (e.g., not zero children and not two children). In the tree at the beginning of this problem the value returned would be two since the nodes labeled *octopus* and *hedgehog* in the original tree each have one child.

```
public int oneChildCount(TreeNode root) {









}
```

Part E (5 points)

Implementing a priority queue using a balanced search tree results in *add*, *peek*, and *remove* operations that are all $O(\log N)$ for a N-element priority queue. However, heaps are typically used to implement priority queues because they offer better performance. Explain why heaps are better than balanced binary trees for implementing priority queues (be brief).

Part F (8 points)

The two methods `print` and `printItems` below generate the following output:

```
iguana eagle bison badger
iguana eagle bison crayfish
iguana eagle hedgehog hyena
iguana octopus wallaby
```

when invoked with the call `print(tree,new LinkedList<String>());` where `tree` is the root of the tree on the previous page.

```
public static void printItems(LinkedList<String> list){
    Queue<String> copy = (Queue<String>) list.clone();
    while (copy.size() != 0){
        System.out.print(copy.remove()+" ");
    }
    System.out.println();
}

public static void print(TreeNode root, LinkedList<String> q){
    if (root == null) return;
    if (root.left == null && root.right == null){
        q.add(root.info);
        printItems(q);
        q.removeLast();
        return;
    }
    q.add(root.info);
    print(root.left,q);
    print(root.right,q);
    q.removeLast();
}
```

If `LinkedList` is replaced everywhere with `Stack` – and `add` replaced by `push`; `removeLast` by `pop` and so forth in both methods above then what will be printed by the call `print(tree, new Stack<String>())` – you should show four lines of output in your answer.

You should also briefly explain why the queue is cloned in the method `printItems`.

PROBLEM 2 : (*It's all in the DNA (6 points)*)

In the DNA-linked-list assignment for this course a simulation of the construction of a *recombinant* strand of DNA consisted of breaking the original strand into B pieces that were then re-assembled by splicing a strand between the breaks to generate a new strand. You were given an implementation whose complexity was $O(N)$ for creating a final, recombinant strand of length N. Using a linked list the complexity was reduced to $O(B)$ regardless of the size of the strand spliced in and thus regardless of the size of the resulting strand.

One correct implementation of the `append` method which appends one strand to another is shown below. This method appends the strand represented by parameter `dna`, the code is part of the class `LinkStrand`.

```
public class Node{
    String info;
    Node next;
    public Node(String s, Node ptr){
        info = s;
        next = ptr;
    }
}
private Node myFirst;
private Node myLast;
private long mySize;

public IDnaStrand append(IDnaStrand dna) {
    LinkStrand appendee = null;
    if (dna instanceof LinkStrand){
        appendee = (LinkStrand) dna;
    }
    else {
        appendee = new LinkStrand();
        appendee.initializeFrom(dna.toString());
    }
    myLast.next = appendee.myFirst.next;
    myLast = appendee.myLast;
    mySize += appendee.size();
    return this;
}
```

For the creation of a recombinant strand to be $O(B)$, the code above must execute in $O(1)$ time. Explain why this code is $O(1)$ for appending a `LinkStrand` object, but $O(M)$ where M is the number of characters in the DNA strand, for appending any other object that implements the the interface *IDnaStrand*.

PROBLEM 3 : (*Mercator Projection (22 points)*)

The code handout at the end of this test shows a `main` that reads a file and creates a map in which keys are the words in the file and the value corresponding to a key is a count of how many times the word occurs. You can see all the code there, but the relevant parts are reproduced below.

```
// scan every word in  file f and count how many times it occurs
Scanner s = new Scanner(f);
Map<String,Integer> m = new TreeMap<String,Integer>();

while (s.hasNext()){
    String str = s.next();
    if (m.containsKey(str)){
        m.put(str,m.get(str)+1);
    }
    else {
        m.put(str, 1);
    }
}
```

Part A (4 points)

Complete the code fragment below so that it correctly stores in `smax` the word that occurs most frequently, and stores the number of occurrences of this word in `max`. Assume this code executes after the code above so that the Map `m` is full of data. If more than one word occurs with the same maximal frequency it doesn't matter which of these your code returns.

```
int max = 0;         // most frequent occurrences
String smax = null;  // most frequently occurring word

for(String s : m.keySet()) {









}
System.out.println("most frequent "+smax + " # occurrences: "+max);
```

Three methods are shown in the code at the end of the test: `topThirty1`, `topThirty2`, and `topThirty3`. Each does the same thing: returns an array of the 30 most-frequently occurring words in order so that the first element in the returned array is the most frequently occurring word, the second element in the array is the second most frequently occurring word, and so on. Each of the three methods works correctly. You'll be asked questions about each method which are reproduced below. For words that occur with the same frequency ties are broken alphabetically.

Part B (6 points)

Recall that the class `Map.Entry` stores both the key and the corresponding value of an entry in a map; both the key and the value can be obtained as shown in the code below.

Explain in words why the method `topThirty1` works. That is why it returns an array with the most frequently occurring word in the map stored at index 0 of the array, the second most frequently occurring word stored at index 1, and so on – as part of your explanation include a discussion of the `Comparator` used. What is the complexity of this code for a map with N elements. Justify your answer.

```
private static class EntryComparator
        implements Comparator<Map.Entry<String,Integer>>{

    public int compare(Map.Entry<String,Integer> a,
                       Map.Entry<String,Integer> b) {
       int diff =  b.getValue() - a.getValue();
       if (diff != 0) return diff;
       return a.getKey().compareTo(b.getKey());
    }
}

public static String[] topThirty1(Map<String,Integer> m){

    ArrayList<Map.Entry<String,Integer>> list =
        new ArrayList<Map.Entry<String,Integer>>(m.entrySet());

    Collections.sort(list, new EntryComparator());

    String[] a = new String[30];
    for(int k=0; k < 30; k++){
        a[k] = list.get(k).getKey();
    }
    return a;
}
```

Part C (6 points)

The method `topThirty2` below uses a *PriorityQueue* to obtain the 30 most frequently occurring words in a map.

(Note: the 10 in the constructor for the priority-queue is the initial size of the priority queue). **Explain what the big-Oh complexity** of the method below is for finding the top 30 words in a map with N elements. Justify your answer.

```
public static String[] topThirty2(Map<String,Integer> m){

    PriorityQueue<Map.Entry<String,Integer>> pq =
       new PriorityQueue<Map.Entry<String,Integer>>(10,new EntryComparator());
    pq.addAll(m.entrySet());

    String[] list = new String[30];
    for(int k=0; k < 30; k++){
        list[k] = pq.remove().getKey();
    }
    return list;

}
```

Part D (6 points)

The code below also uses a *PriorityQueue*, but never stores more than 30 elements in the queue. Explain three things: (A) why the `Collections.reverseOrder` comparator is used in the code; (B) why the index `30-k-1` is used in the statement below

```
list[30-k-1] = pq.remove().getKey();
```

instead of using `list[k]`; and (C) why this method is more efficient than the other two methods. Justify your answers.

```
public static String[] topThirty3(Map<String,Integer> m){

   PriorityQueue<Map.Entry<String,Integer>> pq =
       new PriorityQueue<Map.Entry<String,Integer>>(10,
           Collections.reverseOrder(new EntryComparator()));

    for(Map.Entry<String, Integer> entry : m.entrySet()){
        pq.add(entry);
        if (pq.size() > 30) pq.remove();
    }

    String[] list = new String[30];
    for(int k=0; k < 30; k++){
       list[30-k-1] = pq.remove().getKey();
    }
    return list;
}
```

PROBLEM 4 : (*store e-sort (20 points)*)

Part A (5 points)

Put the letter from the list of speakers on the left next to the phrase that person did say or could have said based on what we've studied in class. You can use a letter more than once, you don't need to use all the letters.

A. Barack Obama

B. Edsger Dijkstra

C. Tony Hoare

D. Fred Brooks

E. Travis Bickle

F. Niklaus Wirth

G. David Huffman

1. _____ Bubblesort's not the way to go
2. _____ I invented quicksort
3. _____ One of these days I gotta get myself organizized
4. _____ My algorithm helps route packets in the Internet
5. _____ My compression algorithm is greedy and optimal

Part B (5 points)

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.

```
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.remove();
   }
}
```

Part C (5 points)

The code below sorts an array of Strings representing DNA by creating arrays for every possible 4-character prefix, e.g., `"aaaa"` `"aaag"` `"aaat"` ... `"agtc"` ... `"gatt"` ... `"tttt"`; sorting each of these arrays, and then combining these sorted arrays together.

```
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.

Part D (5 points)

The code below reads in a file of integer values, stores these 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.

```
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:

```
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".