Test 2 Practice: Compsci 100

Owen Astrachan

March 31

Name: ______________________________________ (2 points)
Login: __________

Honor code acknowledgment (signature) ___________________________________

<table>
<thead>
<tr>
<th></th>
<th>value</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem 1</td>
<td>15 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 2</td>
<td>12 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 3</td>
<td>25 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 4</td>
<td>21 pts.</td>
<td></td>
</tr>
</tbody>
</table>

TOTAL: 75 pts.

This test has 11 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.
Part A (6 points)

Suppose TreeNode objects are used to implement a structure such as that shown below in which nodes may be referenced by more than one pointer and there may be cycles in the structure. Such a structure is called a nest. In this problem nests only contain positive integers.

```
1
 / \
2   3
 /   / \
4   5   6
 |   |   |
7   8   
```

Complete the function nestSum whose header is given below. nestSum returns the sum of all reachable nodes in its parameter nest — nodes should only be “summed” once. If passed a pointer to the node labeled 1 in the nest shown above, nestSum should return 36, if passed a pointer to node labeled 6 nestSum should return 29. Your code is permitted to overwrite the info fields of the nodes in nest as a way of “marking” nodes. Recall that in this problem nests store positive integers. In “marking” a node by overwriting the info field, you may use any integer value (positive, zero, or negative).

```java
/**
 * Returns sum of nest, counting each positive value once.
 * @param nest is a nest whose sum is determined/returned
 * @return sum of values in nest, every node altered as a result of
 * accumulating/returning sum
 */
public static int nestSum(TreeNode nest) {
    if (nest == null) return 0; // cover empty case
    if (nest.info < 0) return 0; // cover case seen before
```

```
Part B (4 points)
The function `nestCount` correctly counts the number of nodes in a nest without altering the values stored in the nest. A Map of nodes seen before is used to avoid counting the same node twice.

Write a recurrence relation for the function `nestCountAux` called by `nestCount` (for the average case). Explain how each line of `nestCountAux` is accounted for in the recurrence. Do not solve the recurrence.

```java
/**
 * Returns sum of all nodes in nest, but doesn’t accumulate
 * sum for any nodes stored in the visited Map.
 * @param nest is a nest whose sum is returned
 * @param visited stores each node of the nest once, mapping
 * the node to the empty string "" simply as a way of marking the
 * node as visited.
 */
public static int nestCountAux(TreeNode nest, Map<TreeNode,TreeNode> visited) {
    if (nest == null) return 0; // don’t count nothing
    if (visited.get(nest) != null) return 0; // or nodes seen before
    visited.put(nest,nest); // mark as visited
    return 1 + nestCountAux(nest.left,visited)
        + nestCountAux(nest.right,visited);
}

public static int nestCount(TreeNode nest) {
    Map<TreeNode,TreeNode> visited= new IdentityHashMap<TreeNode,TreeNode>();
    return nestCountAux(nest,visited);
}
```
Part C (3 points)
In the method `nestCount` an `IdentityHashMap` is created. If a `TreeMap` is used, the code would not work as intended: the code would generate a class-cast exception in the map-implementation with an error message indicating the keys stored in the map do not implement `Comparable`. In the case of an `IdentityHashMap`, the code runs correctly because the HashMap uses references for hashing and determining equality, i.e., if two references refer to the same object in memory, the references will be treated as equal. However, if the two references refer to different objects that have the same “content”, the references will not be treated as equal.

Explain this in a few sentences.
Part D (8 points)
The function copy below is designed to create and return a copy of a nest without altering the nest. A map of pointers in the nest being copied to corresponding newly created nodes in the copied nest is maintained to avoid copying the same node twice. Complete copyAux so that it works as specified.

```java
/**
 * Return a copy of nest, using map to avoid regenerating nodes
 * already copied.
 * @param nest is nest to be copied
 * @param seen stores key->value pairs that associate a TreeNode
 * copy (value) for any node (key) already visited/copied.
 * @return copy of nest
 */
public static TreeNode copyAux(TreeNode nest, Map<TreeNode,TreeNode> seen) {
    if (nest == null) return null;
    if (seen.get(nest) != null){
        return seen.get(nest);
    }

    // create node for returning/storing, first without
    // explicit children so the copy can be stored in the map,
    // then fill in children
    TreeNode result = new TreeNode(nest.info,null,null);

    // insert code here,
}

public static TreeNode copy(TreeNode nest) {
    Map<TreeNode,TreeNode> seen = new IdentityHashMap<TreeNode,TreeNode>();
    return copyAux(nest,seen);
}
```
PROBLEM 1:  (Mind your P’s and Q’s)

Part A
The code below is not complete, but it is intended to merge two sorted queues into one sorted queue. Complete the function so that works as specified by writing code after the loop.

```java
/**
 * Merge two sorted queues into a new sorted queue, return the result.
 * The sorted queues are emptied in the process
 * @param a is a sorted queue
 * @param b is a sorted queue
 * @return sorted queue with all elements from a and b (which are then empty )
 */
public static Queue<String> merge(Queue<String> a, Queue<String> b){
    Queue<String> result = new LinkedList<String>();
    while (a.size() != 0 && b.size() != 0){
        String as = a.peek();
        String bs = b.peek();
        if (as.compareTo(bs) <= 0){
            result.add(a.remove());
        } else {
            result.add(b.remove());
        }
    }
    // fill in code here
    return result;
}
```
Part B

Code to sort the elements of a queue is shown below. What is the asymptotic/big-Oh complexity of this code. Justify your answer.

```java
public void qsort(Queue<String> q)
{
    if (q.size() <= 1) return;

    Queue<String> a = new LinkedList<String>();
    Queue<String> b = new LinkedList<String>();
    String item;
    int size = q.size();
    for (int k=0; k < size; k++){
        String item = q.remove();
        if (k % 2 == 0){
            a.add(item);
        }
        else {
            b.add(item);
        }
    }
    qsort(a);
    qsort(b);
    q.addAll(merge(a,b));
}
```
Part C (4 points)
The code below prints every element in a queue. The storage/memory used (in addition to the queue) is $O(1)$. Explain how this code works but why it’s not possible to print the contents of a stack using this approach, and why printing a stack without destroying its contents requires $O(n)$ storage. Be brief.

```java
public void print(Queue<String> q){
    int size = q.size();

    for(int k=0; k < size; k++){
        String item = q.remove();
        System.out.print(item+" ");
        q.add(item);
    }
    System.out.println();
}
```
Part D

Write function `filterQ` that removes some elements from a queue of strings. The elements that remain in the queue should be in the same relative order they were before the removal process executes.

To check whether an element is removed, your code will call the Predicate object `pred`'s method `satisfies`. If the `satisfies` method returns true, the element is removed.

For example, if the queue `qu` is represented by

"juice", "mom", "racecar", "peace", "radar", "police"

then the call `filterQ(qu, palindrome)` would change the queue `qu` to be the following:

"juice", "peace", "police"

assuming `palindrome` returns true precisely when its string parameter is a palindrome (a word reading the same backward as forward).

The `IPredicate` interface is defined as:

```java
public interface IPredicate {
    public boolean satisfies(String s);
}
```

Complete the function `filterQ` below. You can allocate O(1) additional storage in writing `filterQ`, you cannot create another queue or vector or stack or any structure in which more than O(1) elements from the queue being filtered are stored.

```java
public void filterQ(Queue<String> q, IPredicate pred) {
    // Implementation goes here
}
```
**Part E (4 points)**

Write the definition for a class named `VowelStarter` that implements `IPredicate` and that can be used to filter all strings from a queue that start with a vowel. A vowel is 'a', 'e', 'i', 'o', 'u'. Assume strings are all lowercase letters. For example, the code below should remove all strings that start with a vowel from the queue `sq`.

```java
IPredicate vs = new VowelStarter();
Queue<String> sq = new LinkedList<String>();
// fill sq with values
filterQ(sq,vs);
// now sq contains no strings that begin with vowels
```

In writing your class you may find this code snippet useful.

```java
void snippet(const string& s)
{
    String vowels = "aeiouAEIOU";
    if (vowels.indexOf(s[0]) != -1) {
        System.out.println("begins with vowel");
    }
}
```

Write the implementation of `VowelStarter` below.
PROBLEM 2: (Trees and Lists oh My (20 points))

In this problem you’ll write methods to convert a sorted ArrayList to a balanced binary search tree containing the same values stored in the ArrayList, and vice versa: a method to convert a tree to a sorted list. The picture below shows one example of a list and the tree that corresponds to the list. (the tree shows numbers, but Strings are stored in the tree, see the declarations at the beginning of this test).

Part A (5 points)

The method tree2list returns a sorted ArrayList containing the same values as the tree passed to the method. The general algorithm for the helper method tree2listAux is to perform an inorder traversal of the tree, adding a tree value to the list when the tree node is visited during the traversal. Since the inorder traversal of a search tree visits nodes in sorted order, the resulting list will be sorted. Complete the method tree2listAux, you should not write more than four statements and you may not modify the code in tree2list.

```java
/**
 * Return sorted list containing same values stored in search tree t.
 * @param t is a search tree
 * @return sorted list containing same values in t
 */
public static ArrayList<String> tree2list(TreeNode t){
    ArrayList<String> list = new ArrayList<String>();
    tree2listAux(t,list);
    return list;
}
/**
 * Add values of tree to list in sorted order.
 * @param list stores values (in order) already visited
 * @param root is the root of the (sub) tree to be added to list
 */
private static void tree2listAux(TreeNode root, ArrayList<String> list){
    if (root != null){
    }
}
```
Part B (5 points)
The general algorithm for converting a list to a tree is:

- Find the root of the tree, which is the middle element of the list.
- Make a node for the root, and recursively make left and right subtrees.

The method `list2tree` calls a helper method that does the work, the helper method has parameters indicating the range in the list that is converted to a tree. Complete `list2treeAux` so that it works as specified.

```java
/**
 * Returns balanced tree from sorted list.
 * @param list is sorted
 * @return balanced search tree containing same values in list
 */
public static TreeNode list2tree(ArrayList<String> list){
    return list2treeAux(list,0,list.size()-1);
}

/**
 * Return balanced tree consisting of values in list in range
 * list[first]..list[last], including first and last, where list
 * is sorted. Returns null if first > last.
 * @param list is sorted
 * @param first is first index of list to be put into tree
 * @param last is last index of list to be put into tree
 * @return balanced tree containing values list[first]..list[last]
 */
private static TreeNode list2treeAux(ArrayList<String> list, int first, int last){
    if (first > last) return null;
    int mid = (first + last)/2;
    String rootValue = list.get(mid);

    // add code here
}
```
Part C (3 points)
Write a recurrence relation for the code you wrote for Part A. Justify the recurrence and indicate what its big-Oh solution is.

Part D (3 points)
Write a recurrence relation for the code you wrote for Part B. Justify the recurrence. (If you wrote the simplest code for Part B the solution should be $O(n)$ for an n-element array.)

Part E (4 points)
Suppose you want to convert a sorted linked-list into a search tree. Using the algorithm for Part B above yields an $O(n \log n)$ solution because finding the middle element takes $O(n)$ time. Explain why this results in an $O(n \log n)$ algorithm and explain how to do the conversion in $O(n)$ time by first creating an array from the linked list.
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 Counter that indicates how many time the word occurs in the file. Part of the code is reproduced below.

```java
Scanner s = new Scanner(f);
Map<String,Counter> m = new TreeMap<String,Counter>();
while (s.hasNext()){
    String str = s.next();
    Counter c = m.get(str);
    if (c == null){
        m.put(str, new Counter());
    } else {
        c.increment();
    }
}
```

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.

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

Iterator<String> it = m.keySet().iterator();
while (it.hasNext()){
    String str = it.next();
    ...
}
System.out.println("most frequent " + smax + " "+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.

**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` 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. You should mention how the `Comparator` works in the code.

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

    public int compare(Map.Entry<String,Counter> o1,
                        Map.Entry<String,Counter> o2) {
        Counter c1 = o1.getValue();
        Counter c2 = o2.getValue();
        return c2.getCount() - c1.getCount();
    }
}

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

    ArrayList<Map.Entry<String,Counter>> list =
        new ArrayList<Map.Entry<String,Counter>>(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.

**Explain in words why** the `EntryComparator` is needed when creating the priority queue **and 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.

```java
public static String[] topThirty2(Map<String, Counter> m) {
    PriorityQueue<Map.Entry<String, Counter>> pq =
        new PriorityQueue<Map.Entry<String, Counter>>(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 class `RevComp` is used in the code; (B) why the index 30-k-1 is used in the statement below

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

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

```java
public static String[] topThirty3(Map<String, Counter> m) {
    PriorityQueue<Map.Entry<String, Counter>> pq =
        new PriorityQueue<Map.Entry<String, Counter>>(10,
            Collections.reverseOrder(new EntryComparator()));

    Iterator<Map.Entry<String, Counter>> it = m.entrySet().iterator();
    while (it.hasNext()){
        pq.add(it.next());
        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;
}
```
nothing on this page