Test 2: Compsci 100

Owen Astrachan

November 14, 2006

Name: _____

Login: _____

Honor code acknowledgment (signature) _____

	value	grade
Problem 1	30 pts.	
Problem 2	22 pts.	
Problem 3	18 pts.	
Problem 4	22 pts.	
TOTAL:	92 pts.	

This test has 17 pages, be sure your test has them all. Do NOT spend too much time on one question — remember that this class lasts 75 minutes.

Some common recurrences and their solutions.

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

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.

TreeNodes on this test are implemented using the following declaration which is nested inside a class `Test` in which all methods are written.

```
public static class TreeNode {
    String info;
    TreeNode  left;
    TreeNode  right;
    TreeNode(String val, TreeNode  lptr, TreeNode  rptr) {
        info = val;
        left = lptr;
        right = rptr;
    }
}
```

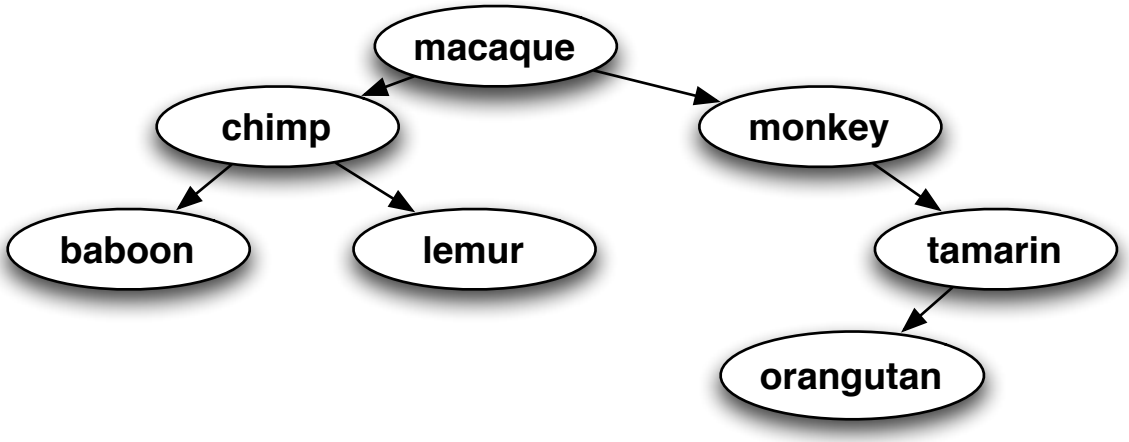
blank page

PROBLEM 1 : (*Oversees, Refugees, Guarantees, Search-trees (30 points)*)

The tree below is a search tree.

Part A (4 points)

What is the pre-order traversal of the tree? (The first value printed is *macaque*)



Part B (2 points)

Provide a word (it doesn't have to be a real word, it must contain at least four letters) that could be inserted as a left-child of monkey so that the tree is still a search tree.

Part C (2 points)

Show by drawing where a node with *gibbon* would be inserted into the tree.

Part D (4 points)

The tree above has a height of four and has three leaves. Draw a *search* tree with the same values, a height of four, and which has four leaves. Draw the tree on the previous page.

continued →

Part E (4 points)

The method `printQ` below prints one line for every node in a tree. The first value printed when called with the tree above is the string *macaque*. What is the complete output?

```
public static void printQ(TreeNode root){
      Queue<TreeNode> q = new LinkedList<TreeNode>();
      if (root != null){
          q.add(root);
      }
      while (q.size() != 0){
          root = q.remove();
          System.out.println(root.info);
          if (root.left != null) q.add(root.left);
          if (root.right != null) q.add(root.right);
      }
  }
```

Part F (4 points)

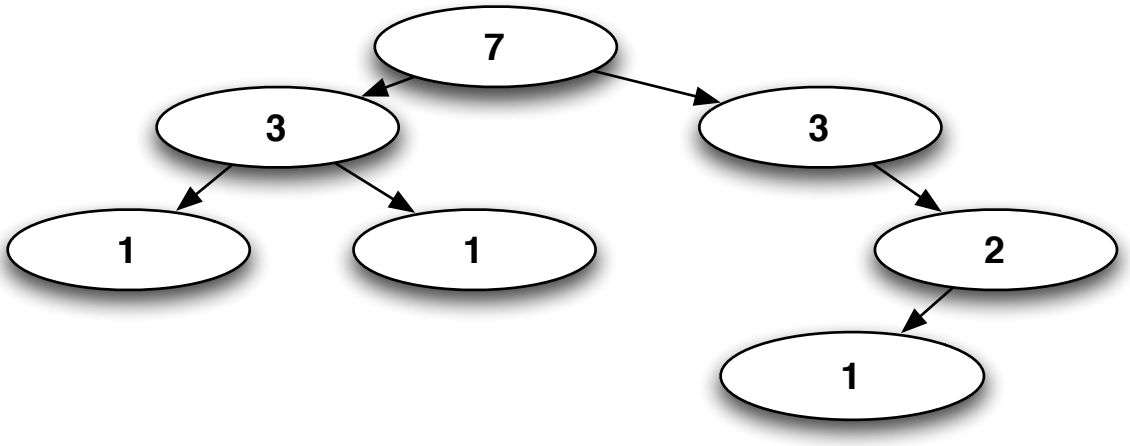
The method `printS` below prints one line for every node in a tree. The first value printed when called with the tree above is the string *macaque*. What is the complete output?

```
  public static void printS(TreeNode root){
      Stack<TreeNode> st = new Stack<TreeNode>();
      if (root != null){
          st.push(root);
      }
      while (st.size() != 0){
          root = st.pop();
          System.out.println(root.info);
          if (root.right != null) st.push(root.right);
          if (root.left != null) st.push(root.left);
      }
  }
```

continued →

Part G (4 points)

The tree below is formed by copying the shape of the monkey/primate tree and storing in each node the count of how many nodes are in the subtree rooted at the node. This new tree could be created by the code shown below the diagram.



The complexity of method `copyCount` below is **not** $O(n)$ for an n-node tree. What is the big-Oh complexity of the method `copyCount` below? Justify your answer (assume trees are rougly balanced.)

```
public static int count(TreeNode root){
    if (root == null) return 0;
    return 1 + count(root.left) + count(root.right);
}

public static TreeNode copyCount(TreeNode root){
    if (root == null) return null;
    return new TreeNode(""+count(root),copyCount(root.left), copyCount(root.right));
}
```

Part H (6 points)

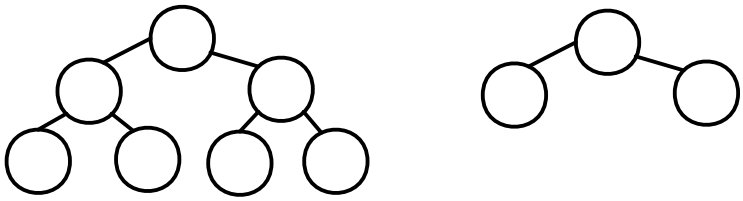
You've found part of an $O(n)$ solution to generating the tree of counts as shown on the previous page, i.e., in which each node stores the count of the nodes in the subtree rooted at the node. The code below is part of a correct solution, you can complete it by adding lines of code. You should not delete any code already written.

The completed method `fastCopy` should run in $O(n)$ time for an n-node tree and should create a counted-copy as described previously. Justify that your solution is $O(n)$ by writing a recurrence for it.

```
public static TreeNode fastCopy(TreeNode root){
    if (root == null) return null;
    root = new TreeNode("",fastCopy(root.left), fastCopy(root.right));

    // do O(1) work below
    int lcount = 0;
    int rcount = 0;
    if (root.left != null) lcount = Integer.parseInt(root.left.info);

    // fill in code below









    return root;
}
```

PROBLEM 2 : (*Trees (22 points)*)

For the purposes of this problem, a *full, complete* binary tree with n levels has 2^{n-1} leaf nodes and, more generally, 2^{k-1} nodes at level k where the root is at level 1, the root's two children are at level 2, and so on. The diagram below shows two such trees, the tree on the left is a level-3 full, complete tree and the tree on the right is a level-2 full, complete tree.



In this problem tree nodes have parent pointers. The declaration for such tree nodes follows.

```
public static class TreeNode {
    String info;
    TreeNode left, right, parent;
    TreeNode(String s, TreeNode lptr, TreeNode rptr, TreeNode pptr) {
       info = s;
       left = lptr; right = rptr;
       parent = pptr;
    }
}
```

Part A (6 points)

Write the method *makeComplete* that returns a full-complete binary tree with the specified number of levels. The call `makeComplete(3,null)` should return a tree such as the one above on the left; `makeComplete(1,null)` should return a single-node tree. The root of the tree has a null parent; all other tree nodes should have correct parent pointers. Use the empty string `""` for the `info` value when creating nodes.

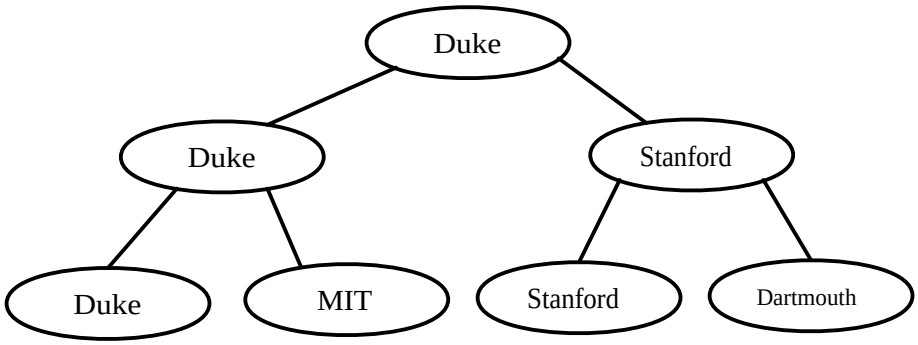
```
/**
 * Return root of a full complete binary tree with # levels specified,
 * returning null when level == 0.
 * @param level is the level of the full/complete tree
 * @param parent is the parent of the root being created and returned
 */
TreeNode makeComplete(int level, TreeNode parent) {









}
```

Part B (4 points) What is the recurrence relation, and big-Oh solution for the code you wrote for part A for an n-level tree? Justify your answer

Part C (6 points)

For this problem you'll treat the full complete tree like a *tournament tree*. In a tournament tree, leaf-value store names. Each internal node stores the winner of the values stored in its two children (since the tree is complete, all non-leaf/internal nodes have two children).

For example, the tree below shows a hypothetical tournament tree with the leaf value storing the names of schools competing in a computer programming contest tournament.



Assume you have a full, complete binary tree, e.g., as would be returned by the function `makeComplete` from Part A. Write the function `assign2leaves` that assigns values in a stack passed to the function to the leaves. For example, suppose the stack is created by the code below:

```
Stack<String> names = new Stack<String>();
names.push("Dartmouth");
names.push("Stanford");
names.push("MIT");
names.push("Duke");
```

then the call `assign2leaves(root, names)` where root is the root of a level-3 full, complete tree should assign values as shown above to the leaves. Note that `"Duke"` is the value at the top of the stack and is stored as the left-most leaf. Your code should do this – this means also that the right-most leaf gets the first value pushed onto the stack.

(continued)

Complete the function below.

```
/**
 * Assign values from stack to leaves of a tournament tree, the
 * stack is emptied in the process.
 * @param stack has at least as many values as there are leaves
 * @param root is root of full, complete tree
 */
public void assign2leaves(TreeNode root, Stack<String> names) {









}
```

Part D (6 points)

In this problem you'll assign winners to the internal nodes of a tree. Assume all leaf nodes have been assigned values, e.g., as in the tournament tree diagrammed previously. You can also assume that a method `determineWinner` exists that will determine which of two teams has won (or will win) a match. For example, here's code to determine the winner of a match between `"Duke"` and `"MIT"`.

```
String winner = determineWinner("Duke","MIT");
```

Here's the javadoc/method header.

```
/**
 * Determine (or predict) winner of a contest/match between two teams
 * and return the winner. Return null if no winner can be determined.
 * @param teamA is one of the teams in the match
 * @param teamB is the other team in the match
 * @return one of teamA or teamB depending on who wins, returns null
 * if winner cannot be determined.
 */
public String determineWinner(String teamA String teamB)
```

Write the method `assignwinners` whose header is given below. The function is passed the root of a tournament tree like the one diagrammed above. Assume all the leaf values have been filled in. The method should assign values to internal nodes so that each internal node stores the winner of the match played between the internal node's children. The winner is determined by calling method `determineWinner`, assume the method always returns a non-null String.

```
public void assignwinners(TreeNode root)
{









}
```

PROBLEM 3 : (*E-voting (18 points)*)

You've been contracted to write a small piece of software to determine the winner of an election using online voting software. Specifically, each person voting has three votes – they can vote for three different people or the same person three times. Candidates are ranked and stored in an array: the first candidate (index zero) receives 3 points, the second candidate (index one) receives 2 points, and the third candidate receives 1 point.

Votes are stored in a map as shown below, this code shows how the map could be constructed, it's not the code that actually records the votes – but it illustrates in code how the map is structured.

```
Map<String,String[]> votes = new HashMap<String,String[]>();

votes.put("Jim", new String[]{"Bob", "John", "Mary"});
votes.put("Sam", new String[]{"John", "Mary", "Mary"});
votes.put("Chris", new String[]{"Mary","Bob","Alice"});
```

These votes show that Jim gave three points to Bob, two to John, and one point to Mary. The final election results in Bob with 5 points, John with 5 points, Mary with 7 points, and Alice with 1 point; Mary would be the winner.

Part A (10 points)

Write the method `voteCalc` that returns a map providing vote totals for every candidate voted for as described above. The method returns a map in which keys are candidate names and in which corresponding values are the vote total for the candidate. This means that for the map returned for the data shown above the value of `map.get("Mary")` should be 7 and the value of `map.get("Alice")` should be 1.

```
/**
 * Return a map of candidates->vote totals based on votes recorded.
 */
public Map<String,Integer> voteCalc(Map<String,String[]> votes){









}
```

Part B (8 points)

One of your colleagues has written code to determine the names of the candidates who would be in a runoff. Any candidate receiving at least 25% of the votes cast participates in a runoff. The code below was written by your colleague before being hired away by an Australian voting firm. The code below adds candidate names to a priority queue so that the first element removed from the queue is the candidate with the most votes, i.e., the priority is determined by votes with high vote-getters having a higher priority.

You should add code to return an ArrayList of those candidates who participate in a runoff. The candidate with the most votes should be stored at index zero of the ArrayList and in general the ArrayList should be ordered from highest to lowest vote-getter of the runoff candidates (ties don't matter). The idea is to repeatedly remove candidate names from the priority queue and add them to an ArrayList if the vote total for the candidate has at least the 25% threshold of total votes. As soon as a candidate doesn't have 25% of the total votes you can stop since the priority queue will remove candidate names from highest vote-getter to lowest. Assume method `voteCalc` works as specified. For example, for the sample data on the previous page the list returned should contain "Mary" first and then "Bob" and "John" in some order since the total number of votes is 18, 25% of that is 4.5, and Bob and John are tied with five votes while Mary has 7.

```
public ArrayList<String> runOff(Map<String,String[] votes) {

      final Map<String,Integer> tally  = voteCalc(votes);

      PriorityQueue<String> pq = new PriorityQueue<String>(10,
              new Comparator<String>(){
                  public int compare(String o1, String o2) {
                     return tally.get(o2) - tally.get(o1);
                  }
      });
      pq.addAll(tally.keySet());         // all candidates in pq

      double total = 0.0;
      for(int count : tally.values()){   // find total # votes cast
          total += count;
      }
      // add code below









}
```

PROBLEM 4 : (*Family Trees (22 points)*)

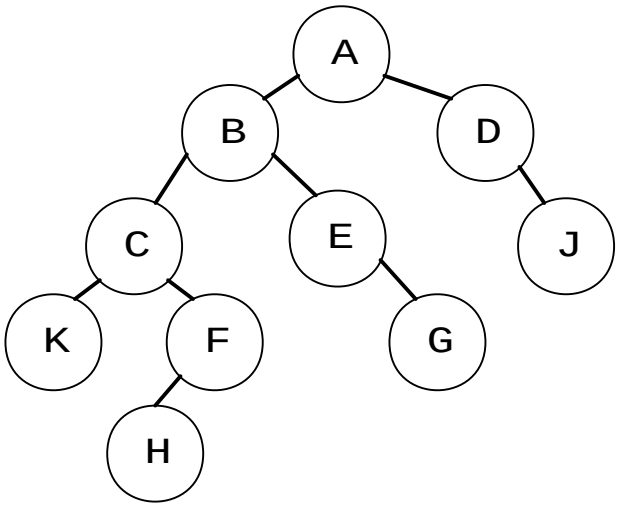
In this problem assume all values in trees are unique, no value appears more than once in a tree. **In this problem the tree is not necessarily a search tree**.

The code in the function `leastAncestor` shown below returns a pointer to the least ancestor of two strings in a tree.

The *least ancestor* of two string values `p` and `q` is the node furthest from the root (deepest) which is an ancestor of both `p` and `q` (there is a path from the least ancestor to both `p` and `q`).

For example, the tree diagrammed below on the right yields the values shown in the table on the left.

`p`	`q`	**least ancestor**
F	E	B
C	G	B
K	H	C
H	J	A
A	G	A
G	A	A



Part A (8 points)

The complexity of *leastAncestor* shown on the next page is not $O(n)$ for an n-node tree. What is the complexity and why? Justify using recurrence relations and an explanation of each part of the recurrence. Provide big-Oh complexities in both the **average case** (assume trees are roughly balanced) and the **worst case**.

Here's code to find the least ancestor, the helper method `inTree` is called from method `leastAncestor`. Note (again) that in this problem trees are not search trees.

```
public static boolean inTree(TreeNode root, String s){
    if (root == null) return false;
    if (root.info.equals(s)) return true;
    return inTree(root.left,s) || inTree(root.right,s);
}

public static TreeNode leastAncestor(TreeNode t, String p, String q){

    if (t == null) return null;

    // first check subtrees (lower than me) for ancestor

    TreeNode result = leastAncestor(t.left, p,q);
    if (result != null) return result;

    result = leastAncestor(t.right,p,q);
    if (result != null) return result;

    // didn't find in subtrees, am I the least ancestor? check
    // me and left/right subtrees for p/q (vice versa)

    if ( (t.info.equals(p) || inTree(t.left,p))  &&
         (t.info.equals(q) || inTree(t.right,q))) {
        return t;
    }
    if ( (t.info.equals(q) || inTree(t.left,q))  &&
         (t.info.equals(p) || inTree(t.right,p))) {
        return t;
    }

    return null;
}
```

Part B (6 points)

Write the method *findPath* whose header is given below. The method stores values in the `ArrayList` parameter representing the path from the root of `t` to the node containing `target` if there is a path (in which case it returns true). If there is no path from the root to `target` then `list` is empty and the method should return false.

For example, given the tree on the previous page we have:

call	**ArrayList list**	**return**
`findPath(t,"F", list)`	`(A,B,C,F)`	true
`findPath(t,"A", list)`	`(A)`	true
`findPath(t,"J", list)`	`(A,D,J)`	true
`findPath(t,"G", list)`	`(A,B,E,G)`	true
`findPath(t,"B", list)`	`(A,B)`	true
`findPath(t,"X", list)`	`()`	false

Hint: the value of the current node is tentatively on the path before recursive call(s) are made, and is removed from the path if the recursive call(s) fail.

```
/**
  * Add values to list so that they represent strings
  * on path from t to node containing target. If target
  * not in the tree then no values added to list.
  * @return true if path found, return false otherwise
  */

public static boolean findPath(TreeNode t, String target, ArrayList<String> list) {
    if (t == null) return false;
    if (t.info.equals(target)){
        list.add(t.info);
        return true;
    }
    // add code here









 }
```

Part C (8 points)

Write a version of `leastAncestor` that runs in $O(n)$ time for an n-node tree. There are at least two approaches to consider (you can code any algorithm that runs in O(n) time, minimal credit will be given for algorithms that do not run in O(n) time.)

One approach is to call `findPath` twice (from the previous part, assume it works as specified) and find the last value that's the same in the lists returned, this is the least common ancestor.

Another approach is write an auxiliary function that returns three values (e.g., in an array of Objects or a class with three fields): an ancestor-pointer and a boolean that tells if p is in the tree and a boolean that tells if q is in the tree.

Write the code and justify that it runs in $O(n)$ time.

```
/**
 * Should run in O(n) time for an n-node tree.
 */
public static TreeNode leastAncestor(TreeNode t, String p, String q){









}
```