Final Review: CPS 100, Fall 2004

Owen Astrachan

December 8, 2004

Name: ________________________________ (1 point)

Login: __________

Honor code acknowledgment (signature) ________________________________

<table>
<thead>
<tr>
<th>Problem</th>
<th>value</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem 1</td>
<td>24 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 2</td>
<td>20 pts.</td>
<td></td>
</tr>
<tr>
<td>TOTAL:</td>
<td>44 pts.</td>
<td></td>
</tr>
</tbody>
</table>
PROBLEM 1:  (Falling Trees (20 points))

The *symmetric difference* of two sets $A$ and $B$ is the set of elements that are in either $A$ or in $B$, but not in both sets (i.e., not in their intersection). We'll define the *xor* of two search trees to be the same as the symmetric difference of the sets the trees represent: we'll assume the search trees contain distinct values.

The code below shows the function `xOr` and a helper function `xorHelper` (the other functions are from class, we've studied them before, but they are included here for completeness). The call `xOr(a,b)` returns a search tree that is the symmetric difference, or xor, of the tree parameters.

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

public boolean contains(TreeNode t, String key) {
    if (t == null) return false;
    if (t.info.equals(key)) return true;
    if (t.info.compareTo(key) < 0) return contains(t.right,key);
    else return contains(t.left,key);
}

public TreeNode insert(TreeNode t, String key) {
    if (t == null) return new TreeNode(key,null,null);
    if (t.info.compareTo(key) < 0) t.right = insert(t.right,key);
    else t.left = insert(t.left,key);
    return t;
}

public TreeNode xorHelper(TreeNode a, TreeNode b , TreeNode result) {
    if (a != null) {
        if (! contains(b,a.info)) { // this is insertion code
            result = insert(result,a.info); // this is insertion code
        } // this is insertion code
        result = xorHelper(a.left,b,result);
        result = xorHelper(a.right,b,result);
        return result;
    }
    return result;
}

public TreeNode xOr(TreeNode a, TreeNode b) {
    TreeNode result = null;
    result = xorHelper(a,b,result);
    result = xorHelper(b,a,result);
    return result;
}
```
Unless otherwise specified, assume the trees passed to `xOr` are roughly balanced so that insertion and contains are both $O(\log n)$ operations for an $n$-element tree/set.

**Part A (4 points)**

If the section of code labelled *this is insertion code* is moved so that it appears between the two recursive calls (rather than before them) the complexity of `xorHelper` will change. What is the complexity of the code as written – justify your answer briefly. What is the complexity of the code if the insertion code is moved between the recursive calls – justify your answer briefly.

**Part B (4 points)**

In the worst case (trees not nice) what is the complexity of the code for `xOr`? Justify your answer.
Part C (12 points)
The code for $x\text{or}$ is not $O(n)$ for two n-element trees/sets. An $O(n)$ algorithm is described as follows.

1. Convert each tree to a sorted ArrayList containing the same elements as in the tree. This can be done in $O(n)$ time using a standard traversal inserting each visited element into an ArrayList.

2. Traverse the sorted arrays in $O(n)$ time storing elements from the xor/symmetric difference of the ArrayList in a third, result ArrayList. The idea is to leverage the sortedness of the arrays in the traversal. For example, if the element in ArrayList A is smaller than the element in list B, then the ArrayList A element belongs in the resulting ArrayList and then the A-index is incremented.

3. Convert the sorted, result ArrayList into a search tree in $O(n)$ time.

Write code to implement the method described above. You should call the methods you write from $x\text{or}$ rather than calling the helper method provided.
PROBLEM 2: (Trees)

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.

```java
public 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.

```java
/**
 * Return a full complete binary tree with # levels specified
 */
TreeNode makeComplete(int level, TreeNode parent) {
```

}
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, or more generically items. 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.

[Diagram of a tournament tree]

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:

```java
Stack names = new Stack();
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. Note that "Duke" is the value at the top of the stack and is stored as the left-most leaf of the leaves. 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.

```java
/**
 * 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 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 map makes it possible to look up the winners of any pair of teams. For example, here’s code to determine the winner of a match between "Duke" and "MIT". This code shows syntactically and semantically how to use the map that stores the winner of a contest between any two teams. Note that any two strings can replace Duke and MIT in the code below to find the winner of a match between the teams corresponding to the strings.

```java
public String DukeMITwinner(Map winnerMap)
{
    Pair pp = new Pair("Duke", "MIT");
    return (String) winnerMap.get(pp);
}
```

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 using the map parameter.

```java
void assignwinners(TreeNode root, Map winnerMap)
// pre: leaf values of tournament tree with root have values assigned
// post: internal nodes of tournament tree have values assigned
//        consistent with winner information represented in winnerMap
{
```