Test 2 Addendum: CPS 100

Owen Astrachan       Jeffrey Forbes

April 2, 2004

Name: ______________________________________________________
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>22 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 2</td>
<td>20 pts.</td>
<td></td>
</tr>
<tr>
<td>TOTAL:</td>
<td>42 pts.</td>
<td></td>
</tr>
</tbody>
</table>

This test has 9 pages, be sure your test has them all.
You may use your book and notes, but you may not use the Internet except to post questions on the class bulletin board (no searching for solutions). You may not talk to any person about these questions. If you submit your solution, you have agreed that you have adhered to these requirements and the Duke Community Standard.
Some common recurrences and their solutions.

<table>
<thead>
<tr>
<th></th>
<th>Recurrence</th>
<th>Solution 1</th>
<th>Solution 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>$T(n) = T(n/2) + O(1)$</td>
<td>$O(\log n)$</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>$T(n) = T(n/2) + O(n)$</td>
<td>$O(n)$</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>$T(n) = 2T(n/2) + O(1)$</td>
<td>$O(n)$</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>$T(n) = 2T(n/2) + O(n)$</td>
<td>$O(n \log n)$</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>$T(n) = T(n-1) + O(1)$</td>
<td>$O(n)$</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>$T(n) = T(n-1) + O(n)$</td>
<td>$O(n^2)$</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>$T(n) = 2T(n-1) + O(1)$</td>
<td>$O(2^n)$</td>
<td></td>
</tr>
</tbody>
</table>
PROBLEM 1: (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.

```c
struct TreeNode {
    string info;
    TreeNode * left, * right, * parent;
    TreeNode(const string& s, TreeNode * lptr, TreeNode * rptr, TreeNode * pptr)
        : info(s), left(lptr), right(rptr), parent(pptr)
    {}
};
```

Part A (6 points)

Write the function `makeComplete` that returns a full-complete binary tree with the specified number of levels. The call `makeComplete(3,0)` should return a tree such as the one above on the left; `makeComplete(1,0)` should return a single-node tree. The root of the tree has a NULL/0 parent; all other tree nodes should have correct parent pointers. Use the empty string "" for the `info` value when creating nodes.

```c
TreeNode * makeComplete(int level, TreeNode * parent)
// pre: 1 <= level, parent points to parent of node created at this level
// post: returns a full, complete tree with # levels specified by level
{
```
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, 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.

![Tournament Tree Diagram]

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:

```cpp
#include <iostream>
#include <stack>

using namespace std;

int main()
{
    stack<string> names;
    names.push("Dartmouth");
    names.push("Stanford");
    names.push("MIT");
    names.push("Duke");
    // Call assign2leaves with the root of the tree and the stack
    assign2leaves(root, names);
    return 0;
}
```

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.

```cpp
void assign2leaves(TreeNode* root, tstack<string>& names)
// pre: names has at least as many values as their are leaves
// in the full, complete tree pointed to by root.
// post: values in names have been assigned to leaf-nodes.
// names will decrease in size each time a value is added to a leaf
{
```
```
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.

```c++
string DukeMITwinner(tmap<pair<string,string>, string> * winnerMap)
{
    pair<string,string> pp = make_pair("Duke", "MIT");
    return winnerMap->get(pp);
}
```

Write the function `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 function 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.

```c++
void assignwinners(TreeNode * root,
    tmap<pair<string,string>, string> * 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
{
```
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 \( \text{xOr} \) and a helper function \( \text{xorHelper} \) (the other functions are from class, we've studied them before, but they are included here for completeness). The call \( \text{xOr}(a,b) \) returns a search tree that is the symmetric difference, or xor, of the tree parameters.

```cpp
struct Tree
{
    string info;
    Tree * left; Tree * right;
    Tree(const string& s, Tree* llink, Tree * rlink)
    : info(s), left(llink), right(rlink)
    {}
};

bool contains(Tree * t, const string& key)
{
    if (t == 0) return false;
    if (t->info == key) return true;
    if (t->info < key) return contains(t->right,key);
    else return contains(t->left,key);
}

Tree * insert(Tree * t, const string& key)
{
    if (t == 0) return new Tree(key,0,0);
    if (t->info < key) t->right = insert(t->right,key);
    else t->left = insert(t->left,key);
    return t;
}

Tree * xorHelper(Tree * a, Tree * b ,Tree * result)
{
    if (a != 0){
        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;
}

Tree * xOr(Tree * a, Tree * b)
{
    Tree * result = 0;
    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 xOr 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 vector 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 a vector.

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

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

Write code to implement the method described above. You should call the functions you write from xOr rather than calling the helper function provided.