Test 2: CPS 103

Owen Astrachan

April 8, 1994

Name: ________________________________

Honor code acknowledgement (signature) ________________________________

<table>
<thead>
<tr>
<th>Problem</th>
<th>Value</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem 1</td>
<td>12 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 2</td>
<td>4 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 3</td>
<td>10 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 4</td>
<td>16 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 5</td>
<td>12 pts.</td>
<td></td>
</tr>
<tr>
<td>TOTAL:</td>
<td>54 pts.</td>
<td></td>
</tr>
</tbody>
</table>
PROBLEM 1:  \((Short Stuff: 12\ points)\)

For each of the terms below, write at most a two sentence description of the term in the context of computer science, programming, and C/C++.

1. *radix sort*

2. *AVL tree*

3. *template*

PROBLEM 2:  \((Priority Trees (4\ points))\)

If items and priorities are stored in a binary search tree ordered by priority [the priorities form the keys as to how the tree is organized] then what are the complexities of the *Insert* and *DeleteMin* operations [use O-notation, assume the tree has \(n\) nodes] — give both average and worst case times with a brief justification.

---

For the remainder of this test assume that linked lists and trees are implementing using the definitions *ListNode* and *TreeNode*, respectively, given below:

```c
struct ListNode{
    int info;
    ListNode * next;
    ListNode(int, ListNode *);
};
ListNode::ListNode(int datum, ListNode * link = NULL) // postcondition: construct list node from given params
{
    info = datum;
    next = link;
}
```
struct TreeNode{
    int info;
    TreeNode * left, * right;
    TreeNode(int, TreeNode *, TreeNode *);
};
TreeNode::TreeNode(int datum, TreeNode * llink = NULL, TreeNode * rlink = NULL)
// postcondition: construct tree node from given params
{
    info = datum;
    left = llink;
    right = rlink;
}

PROBLEM 3:  (Trees I (10 points))

A tree is **height-balanced** if for all nodes in the tree, the heights of the left and right subtrees differ by no more than one and both subtrees are height-balanced. The function `IsBalanced` shown below correctly determines whether `tree` is balanced.

```c
int IsBalanced(TreeNode * tree)
// postcondition: returns 1 if tree is balanced, else returns 0
{
    int retval = 1;   // assume balanced
    if (tree != NULL){
        if (IsBalanced(tree->left) && IsBalanced(tree->right) &&
            abs(Height(tree->left) - Height(tree->right)) <= 1){
            retval = 1;
        }
        else {
            retval = 0;
        }
    }
    return retval;
}
```

**Part A: (3 points)** What is the complexity of the function `IsBalanced` shown above in terms of `n`, the number of nodes in `tree`. Briefly justify your answer.
Part B (7 points)  Complete the function AuxBalance below so that the function IsBalanced2 returns 1 if t is balanced and works in $O(n)$ time.

```c
int IsBalanced2(TreeNode *t)
{
    int dummyHeight; int retval = 1;
    if (t != NULL){
        retval = AuxBalance(t, dummyHeight);
    }
    return retval;
}

int AuxBalance(TreeNode * t, int & height)
// postcondition: returns 1 if t is balanced, else returns 0
//                  sets height to the height of tree t
{
    int retval = 1;
    if (t != NULL){
        int left, right, leftHeight, rightHeight;

        left = AuxBalance(t->left, leftHeight);
        right = AuxBalance(t->right, rightHeight);

    }
    return retval;
}
```
PROBLEM 4:  (Trees II: (16 points))

Part A: 4 points  Write the function OneChildCount whose header is given below. The function returns the number of nodes in tree that have exactly one child. For example, for the tree below the call OneChildCount(t) should evaluate to 1.

```
int OneChildCount(TreeNode * tree)
// postcondition: returns # of nodes with exactly one child
```
Part B: 12 points  Write the function TreeToList whose header is given below. The function constructs a singly-linked list whose nodes are in numerical order using the information stored in the parameter tree (which is assumed to be a binary search tree of integers). Pointers to the first and last nodes of the constructed list are returned. For example, the call TreeToList(t, first, last) would create the linked list diagrammed below (where t is diagrammed on the previous page):

```
 3 → 5 → 12 → 13 → 39 → 88
```

```c
void TreeToList(TreeNode * tree, ListNode * & first, ListNode * & last)
// precondition: tree is a binary search tree
// postcondition: first points to first node of sorted linked list
// last points to last node of the sorted list
// info fields of elements in list are same as in tree
// if tree is NULL then first and last are NULL
```
PROBLEM 5: (Tries (15 points))

The following definition is used for implementing tries (assume 'a' corresponds to index[0], and 'z' to index[25]):

```c
const int ALPH_SIZE = 26;           // all letters lower-case
struct Trie{
  int isWord;               // 1 if word, 0 if not
  Trie * index[ALPH_SIZE];  // following chars
  Trie(int);                // constructor
};
Trie::Trie(int word)
{
  isWord = word;
  int k;
  for(k=0; k < ALPH_SIZE; k++){
    index[k] = NULL;
  }
}
```

A diagram of a small trie is shown below, the circles indicate that isWord is 1 (a word ends at the node).

![Trie Diagram](image)

Part A: (6 points) Write the function WordCount whose header is given below. WordCount should return the number of words stored in t. For example, the call WordCount(tr) should evaluate to 8 where tr is diagrammed above.

```c
int WordCount(Trie * t)
// postcondition: returns # of words stored in t
```

(turn page ——)
Part B: (6 points) Write the function NoPrefixCount whose header is given below. This function returns the number of words in t such that no prefix of the word is also a word. In the trie above, there are five such words: do, tax, tea, ten, and to. The words dot, don, and tent all have prefixes that are also words (do, do, and ten respectively) and thus don’t contribute to the count returned by NoPrefixCount.

    int NoPrefixCount(Trie * t)
    // postcondition: returns # of words without prefixes

PROBLEM 6: (Extra Credit (5 points))

This question is extra, it is not required

Write a function LengthLongest that returns the length of the longest word in a trie. The function should have one parameter, a pointer to a trie. You may find it useful to write an auxiliary function with an int parameter representing the current “depth” (or length of word being looked at) in the trie.