

DUKE UNIVERSITY  
Department of Computer Science

CPS 100  
Fall 2001

J. Forbes  
Test #2

**PROBLEM 1 : (*Drawing (10 points)*)**

a. Given the following map from characters to code words, draw the Huffman tree that generated them

A	11
B	01
C	00
D	101
E	100

b. Insert the following elements into an AVL tree. Make sure you show the tree before and after each rotation.

2 9 5 1 10 15 6 7

**PROBLEM 2 : (Clean up your heap (10 points))**

The following routine is the *heapify* routine discussed in class. Given that the two subtrees of a particular element are heaps, it will maintain the heap property for the heap for the tree rooted at that element.

- a. Show how we can build a heap using a recursive divide and conquer function `BuildHeap(v, i)` where `v` is the vector holding the elements of the heap and `i` is the index of the root of tree to be built. To build the entire heap, we would call `BuildHeap(myList, 1)`; Your solution should make the recursive calls to *BuildHeap* on the left and right subtrees and call *heapify* where necessary. Make sure that when you call *heapify* its precondition is satisfied.

```
BuildHeap(tvector<int> &myList, int vroot)
// pre: there exist items in vector from index 1 to myList.size()-1
// post: subtree starting at index i forms a heap
{
    if (vroot <= myList.size()/2)
    {
        BuildHeap(myList, vroot*2);
        BuildHeap(myList, vroot*2 + 1);
        Heapify(myList, vroot);
    }
}
```

- b. Give the recurrence and big-Oh for the *BuildHeap* function you wrote.

$$\begin{aligned} T(n) &= 2T(n/2) + O(\log n) \\ T(1) &= 1 \\ &\in O(n) \end{aligned}$$

**PROBLEM 3 : (Deepest, Greenest (17 points))**

The code for function *DeepLeaf* given below returns a pointer to a leaf in the tree that is farthest from the root, i.e., the deepest leaf in the tree.

```
Tree * DeepLeaf(Tree * t)
{
    if (t == NULL)
        return NULL;
    else if (IsLeaf(t))
        return t;
    else if (height(t->left) >= height(t->right))
        return DeepLeaf(t->left);
    else
        return DeepLeaf(t->right);
}
```

- a. We can instead write *DeepLeaf* without making any calls to *height*. Fill in the recursive calls of *DoDeep* below so that *DeepLeaf* works correctly.

```
Tree * DeepLeaf(Tree * t)
```

```

    {
        Tree * deep = NULL; int max = 0;
        DoDeep(t,0,max,deep);
        return deep;
    }
void DoDeep(Tree * t,int depth, int & maxDepth, Tree * & deepTree)
// precondition:
// postcondition:
{
    if (t == NULL)
        return;
    if (IsLeaf(t))
    {
        if (depth > maxDepth){
            maxDepth = depth;
            deepTree = t;
        }
    } else {
        DoDeep( t->left, depth+1 ,maxDepth,deepTree);

        DoDeep( t->right, depth+1 ,maxDepth,deepTree);
    }
}

```

- b. Write the recurrence for *DoDeep*.

$$T(n) = T(n/2) + 1$$

$$T(1) = 1$$

- c. The following definition is used for implementing tries: Write a function *LongestWordLength* that returns the length of the longest word in the trie. For the tree above, the function would return 4. Remember that the end of the longest word will be located at the deepest leaf node. You will quite likely find it easier if you use an auxiliary function like in *DeepestLeaf* above.

```

int LongestWordLength(Trie * t)
// pre: Assume all leaf nodes have isWord as true
// post: Trie is unchanged, returns length of longest word in trie
{
    int depth = 0;
    GoLong(t, 0, depth);
    return depth;
}
void GoLong(Trie * t, int depth, int maxdepth)
{
    if (t ==0) return;

    if (isLeaf(t))
    {
        if (t->isWord && depth > maxdepth)
            maxdepth = depth;
    }
    else

```

```

for (int i=0; i < t->index.size(); i++)
    GoLong(t, depth+1, maxdepth);
}
    
```

**PROBLEM 4 : (The Billboard Top  $k$  (9 points))**

Given a set of  $n$  ints, we wish to find the  $k$  smallest in sorted order where  $n$  is much larger than  $k$ . Find the algorithm that implements . Analyze the running time of the following methods in terms of  $n$  and  $k$ . No justification is necessary

a. Sort the numbers and list the  $k$  smallest

sort:  $O(n \log n)$   
 list :  $k * O(1)$   
 $O(n \log n + k) \in O(n \log n)$

b. Build a heap from from the numbers and call *deletemin*  $k$  times. Build heap:  $O(n)$

Calls to delete min:  $k * O(\log n)$   
 $O(n + k \log n)$

c. You are given a function *Select* that can find the  $k$ th smallest number in  $O(n)$  time. Use *Select* to find the  $k$ th smallest number, partition (from QuickSort), and then sort the  $k$  smallest numbers using MergeSort.

Select  $k$ th element:  $O(n)$   
 Sort  $k$  smallest:  $O(k \log k)$   
 $O(n + k \log k)$

d. Which algorithm gives the best asymptotic worst-case running time? Why?

Part a is clearly out since  $O(n \log n)$  is worse than  $O(n + k \log k)$ , since  $n \log n > \max(n, k \log k)$ . Part c takes the least time since  $O(k \log k) < O(k \log n)$  because for all  $k < n$ ,  $\log k < \log n$ .

**PROBLEM 5 : (Know your history (2 points EXTRA CREDIT))**

What network application/protocol was invented by 2 Duke grad students with a UNC grad student that has made all of our lives better this semester? For an extra point, name one of the students.

NNTP/News was invented by Tom Truscott and Jim Ellis from Duke along with Steve Bellovin of UNC.

**PROBLEM 6 : (Sort away (4 points))**

In the following table, match the sorting algorithm to the number of comparisons.

	Sort 1	Sort 2	Sort 3	Sort 4	Sort 5
Set 1	20	6	6	11	28
Set 2	17	14	42	11	28
Set 3	20	12	6	9	28
Set 4	18	15	24	14	28
Set 5	411	330	3969	306	2080
<b>YOUR ANSWER GOES HERE</b>	<b>H</b>	<b>Q</b>	<b>B</b>	<b>M</b>	<b>S</b>