DUKE UNIVERSITY
Department of Computer Science

CPS 100
Fall 2001

Name: ____________________________________________
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>10 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 2</td>
<td>10 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 3</td>
<td>17 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 4</td>
<td>9 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Problem 6</td>
<td>4 pts.</td>
<td></td>
</tr>
<tr>
<td>Free!</td>
<td>4 pts.</td>
<td></td>
</tr>
<tr>
<td>TOTAL:</td>
<td>50 pts.</td>
<td></td>
</tr>
</tbody>
</table>

This test has 6 questions on 8 pages. Be sure your test has them all.

This is an open-book test. You have at least 50 minutes to complete it. That means you should spend no more than 1 minute per point. If the number You may consult any books, notes, or other inanimate objects (other than computers) available to you. You may use any program text supplied in lectures, assignments, or solutions. In writing code you do not need to worry about specifying the proper #include header files. Assume that all the header files we’ve discussed are included in any code you write.

Please write your answers in the spaces provided in the test. Make sure to put your name, and login in the space provided above. Put your login and initials clearly on each page of this test and on any additional sheets of paper you use for your answers.

Don't panic. Just read all the questions carefully to begin with, and first try to answer those parts about which you feel most confident. Do not be alarmed if some of the answers are obvious.

Some common recurrences and their solutions:

- \( T(n) = T(n/2) + O(1) \quad O(\log n) \)
- \( T(n) = T(n/2) + O(n) \quad O(n) \)
- \( T(n) = 2T(n/2) + O(1) \quad O(n) \)
- \( T(n) = 2T(n/2) + O(n) \quad O(n \log n) \)
- \( T(n) = T(n-1) + O(1) \quad O(n) \)
- \( T(n) = T(n-1) + O(n) \quad O(n^2) \)
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.

```c
heapify(tvector<int> &myList, int vroot)
// precondition: subheaps of vroot satisfy heap property (and shape)
// postcondition: heap rooted at vroot satisfies heap property
{
    int last = myList[vroot];
    int child, k = vroot;
    int myNumElts = myList.size()-1; // index 0 is empty
    while (2*k <= myNumElts) // index is empty
    {
        // find minimal child (assume left, then check right)
        child = 2*k;
        if (child < myNumElts &&
            myComp.compare(myList[child], myList[child+1]) > 0)
        {
            child++; // it goes here
            continue;
        }
        if (myComp.compare(last, myList[child]) <= 0) // it goes here
            break;
        else
        {
            myList[k] = myList[child];
            k = child;
        }
    } // found “resting place”, insert ‘last element’
    myList[k] = last;
}
```

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.

```c
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
{
```
b. Give the recurrence and big-Oh for the \texttt{BuildHeap} function you wrote.

\textbf{PROBLEM 3: \textit{(Deepest, Greenest (17 points))}}

The code for function \texttt{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.

\begin{verbatim}
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); 
}
\end{verbatim}

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

\begin{verbatim}
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( 
        ,maxDepth,deepTree); 
        DoDeep( 
        ,maxDepth,deepTree); 
    } 
}
\end{verbatim}
b. Write the recurrence for DoDeep.

c. The following definition is used for implementing tries:

```cpp
const int ALPH_SIZE = 129;
struct Trie{
    bool isWord; // true if word, false if not
    Trie * index[ALPH_SIZE]; // following chars
    Trie()
        : isWord(false), links(ALPH_SIZE, 0) // all initially NULL/O
    {
    }
};
```

and this is an example of the use of a trie in `insert`:

```cpp
void insert(Trie * trie, const string& s)
    // pre: trie != NULL
    // post: s stored in trie
{
    Node * t = trie; // start at root
    for(int k=0; k < s.length(); k++){
        if (t->links[s[k]] == 0) {
            t->links[s[k]] = new Node();
        }
        t = t->links[s[k]];
    }
    t->isWord = true;
}
```

You may find the function `isLeaf` defined here useful for the following question.

```cpp
bool isLeaf(Trie *t)
{
    int k;
    for(k=0; k < ALPH_SIZE; k++)
        if (t->index[k] != NULL)
            return false;
    return true;
}
```
A diagram of a small trie is shown below, the circles indicate that isWord is true (a word ends at the node).

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.

```c
int LongestWordLength(Trie * t)
// pre: Assume all leaf nodes have isWord as true
// post: Trie is unchanged, returns length of longest word in trie
{ 
```
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

b. Build a heap from from the numbers and call \texttt{deletemin} \(k\) times.

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

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

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.
PROBLEM 6: \((\text{Sort away (4 points)})\)

You have been hired by the NotTooSmart Computer Company. Their main product is a very nice library of routines that sort strings alphabetically from smallest to largest. Unfortunately, a disgruntled employee removed all the source code from their system and they don’t remember which sort routine is which. They have been left with an executable program that can run any of five different sort routines (heapsort, mergesort, selection sort, quicksort, and bubblesort), but they do not know which is which. They present you with the five data sets described in Table 1 and the data shown in Table 2 (on the next page). Your job is to figure out which sort routine is which. You should not assume that the sorts are implemented identically to those presented in class. Use what you know about the algorithms and assume that the algorithms were implemented intelligently (e.g. if two values are the same, the sort tries not to rearrange them).

<table>
<thead>
<tr>
<th>Set 1</th>
<th>Set 2</th>
<th>Set 3</th>
<th>Set 4</th>
<th>Set 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>dog</td>
<td>squirrel</td>
<td>cat</td>
<td>deer</td>
<td>This set contains 64 randomly generated words in random order</td>
</tr>
<tr>
<td>dog</td>
<td>mouse</td>
<td>deer</td>
<td>mouse</td>
<td>Note: 64*64 = 4096;</td>
</tr>
<tr>
<td>dog</td>
<td>hamster</td>
<td>dog</td>
<td>cat</td>
<td>log(64) = 6;</td>
</tr>
<tr>
<td>dog</td>
<td>fox</td>
<td>fox</td>
<td>squirrel</td>
<td>6*64 = 384</td>
</tr>
<tr>
<td>dog</td>
<td>dog</td>
<td>hamster</td>
<td>fox</td>
<td></td>
</tr>
<tr>
<td>dog</td>
<td>deer</td>
<td>mouse</td>
<td>dog</td>
<td></td>
</tr>
<tr>
<td>dog</td>
<td>cat</td>
<td>squirrel</td>
<td>hamster</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Data Sets

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

<table>
<thead>
<tr>
<th>Set 1</th>
<th>Sort 1</th>
<th>Sort 2</th>
<th>Sort 3</th>
<th>Sort 4</th>
<th>Sort 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set 1</td>
<td>20</td>
<td>6</td>
<td>6</td>
<td>11</td>
<td>28</td>
</tr>
<tr>
<td>Set 2</td>
<td>17</td>
<td>14</td>
<td>42</td>
<td>11</td>
<td>28</td>
</tr>
<tr>
<td>Set 3</td>
<td>20</td>
<td>12</td>
<td>6</td>
<td>9</td>
<td>28</td>
</tr>
<tr>
<td>Set 4</td>
<td>18</td>
<td>15</td>
<td>24</td>
<td>14</td>
<td>28</td>
</tr>
<tr>
<td>Set 5</td>
<td>411</td>
<td>330</td>
<td>3969</td>
<td>306</td>
<td>2080</td>
</tr>
</tbody>
</table>

YOUR ANSWER GOES HERE Q

Table 2: Number of comparisons on each data set

Use the following abbreviations: Q for Quicksort, H for heapsort, M for mergesort, S for selection sort, and B for bubble sort. **NOTE: BE CAREFUL NOT TO SPEND TOO MUCH TIME ON THIS PROBLEM!!** Just put your choices down for the 4 sorts other than the already selected Quicksort.

**Observations to help you find the answer**

Selection sort takes the same number of operations regardless. Bubble sort will require only a single pass for identical items or in-order items. Next, think about how the sorts will behave on ordered data

Now, how do you differentiate heap and merge sort? Heapsort proceeds by building a heap in \(O(n)\) time and then calling \(\text{extract-min}\) \(n\) times at cost \(O(\log n)\) each. We can then make the observation that both are \(O(n \log n)\), but heap must actually do an \(O(n)\) pass to heap the tree in the first place. So, for small values of \(n\), you’d expect heap to be slower.