DUKE UNIVERSITY  
Department of Computer Science  

CPS 100  
Fall 2003  

Test #2

Name: ________________________________

Login: __________

Honor code acknowledgment (signature) ________________________________

<table>
<thead>
<tr>
<th></th>
<th>value</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Problem 1</td>
<td>6 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 2</td>
<td>8 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 3</td>
<td>10 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 4</td>
<td>9 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 5</td>
<td>10 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 6</td>
<td>8 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 7</td>
<td>18 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 8</td>
<td>6 pts.</td>
<td></td>
</tr>
<tr>
<td>Extra credit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TOTAL:</td>
<td>75 pts.</td>
<td></td>
</tr>
</tbody>
</table>

This test has 8 questions on 12 pages. Be sure your test has them all. This test is worth 75 points. You have 75 minutes to complete it. That means you should spend no more than 1 minute per point.

This is an open-book test. You may consult any books, notes, or other inanimate objects (other than computers) available to you. Your answers may refer to any program text supplied in lectures, assignments, or solutions.

Please write your answers in the spaces provided in the test. Make sure to put your name, and login in the space provided below. Put your login clearly on each page of this test (worth 1 point extra credit) 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.
PROBLEM 1:  \((\text{Balance (6 points)})\)
Insert the following elements into an AVL tree. Make sure you show the tree before and after each rotation.

A B G E F C
PROBLEM 2: (Heaps (10 points))

Below is a vector V in which we are storing a min-Heap, using the representation discussed in class (i.e. obeying the heap order property and the heap shape property). The current number of elements is 6.

\[
\begin{array}{ccccccc}
12 & 21 & 31 & 36 & 32 & 38 & \\
\end{array}
\]

a. Show the vector corresponding to the heap that results if we insert 23.

b. Instead of using a heap, we decide to use an AVL tree to represent our priority queue. What will be the worst-case big-Oh of insert and deleteMin for the AVL tree priority queue?

How do those complexities compare to the respective values for a heap? (slower, same, or faster)
PROBLEM 3:  (David’s work (10 points))

1. Given the following Huffman tree:

   ![Huffman Tree Diagram]

   Write the Huffman encoding for the string “SENT”.

2. The "shape" of a Huffman coding tree gives information about how well the algorithm will be able to compress the input. What tree shape leads to significant compression in the output? What shape that leads to little or no compression?

3. What happens if you use Huffman encoding to compress a file, image.hf, that has already been compressed with Huffman coding? Will the resulting file, image.hf.hf be smaller than the original (image.hf), the same size, or larger? You should briefly justify your answer by discussing the properties of the new encoding. What will the new Huffman Tree look like?
PROBLEM 4:  (Sort it out (9 points))

Upon graduation, you are rewarded with a job applying sorting algorithms to various data sets. You remember from CPS 100 that QuickSort works very well and you use that most of the time. However, you also have a few other sorting algorithms that you can use if necessary. In what cases if any would the sorting algorithms below be more appropriate than QuickSort. Briefly justify your answers.

a. Insertion Sort

b. Merge Sort

c. Radix Sort

PROBLEM 5:  (Relations (10 points))

For each of the following two code excerpts, express the recurrence relation and the resulting big-Oh.

a. doubleTree where n is the number of nodes in the tree rooted at node (assume average case)

```c
void doubleTree(TreeNode* node)
{
    // pre: node points to a BST
    // post: for each node in the BST, creates a new duplicate node
    //       and inserts the duplicate as the left child of the
    //       original node. The result tree is still a BST

    TreeNode* oldLeft;

    if (node==0) return;

    // do the subtrees
    doubleTree(node->left);
    doubleTree(node->right);

    // duplicate this node to its left
    oldLeft = node->left;
    node->left = new TreeNode(node->info, 0, 0);
    node->left->left = oldLeft;
}
```
b. isBST3 where n is the number of nodes in the tree (assume worst case)

```cpp
bool ValsLess(TreeNode *t, string key) {
    if (t == 0) return true;
    return t->info < key &&
           ValsLess(t->left, key) &&
           ValsLess(t->right, key);
}

bool ValsGreater(TreeNode *t, string key) {
    if (t == 0) return true;
    return t->info < key &&
           ValsGreater(t->left, key) &&
           ValsGreater(t->right, key);
}

bool IsBST3(TreeNode * t) {
    if (t == NULL) return true;  // empty tree is a search tree

    return ValsLess(t->left,t->info) &&
           ValsGreater(t->right,t->info) &&
           IsBST3(t->left) &&
           IsBST3(t->right);
}
```
PROBLEM 6: (Searching, searching (8 points))
Given the following definition of a graph:

Graph g;

g.addVertex("0"); g.addVertex("1");
g.addVertex("2"); g.addVertex("3");
g.addVertex("4"); g.addVertex("5");
g.addVertex("6");

g.addUndirectedEdge("0", "2");
g.addUndirectedEdge("0", "5");
g.addUndirectedEdge("1", "2");
g.addUndirectedEdge("3", "4");
g.addUndirectedEdge("3", "5");
g.addUndirectedEdge("4", "5");
g.addUndirectedEdge("2", "4");

a. Give the order that nodes are visited when performing a depth-first search on the above graph starting at vertex 0. The node order will not be unique. Drawing the graph may be useful but is not required.

b. Give the order that nodes are visited when performing a breadth-first search on the above graph starting at vertex 0. Once again, the order will not necessarily be unique.
PROBLEM 7:  *(Trie again (18 points))*

Given the following definition of Trie:

```
const int ALPH_SIZE = 129;  // # "real" chars

/**
 * standard trie node, links
 * to all children, initialized to NULL/0
 */
struct Trie
{
    bool isWord;  // true iff node ends a word
    vector<Trie*> links;  // links to child nodes
    Trie()
    :
        isWord(false),
        links(ALPH_SIZE,0)
    {}
};
```

a. Write a function to add a word to a trie.

```
void AddWord(Trie * & t, string word)
// post: word added to Trie t
```
b. In this problem, you will write a function \textit{TrieIntersect} that returns a trie that is the intersection of two tries.

You may assume a function \textit{CopyTrie} is already written. Its prototype is below:

\begin{verbatim}
Trie * CopyTrie(Trie * root);
// post: returns copy of trie pointed to by root
\end{verbatim}

You might want to model your function after the following definition of \textit{TrieUnion}:

\begin{verbatim}
Trie * TrieUnion(Trie * lhs, Trie * rhs)
// postcondition: returns a trie that is the union of lhs and rhs
// i.e., every word in lhs or in rhs is in returned trie
{
    Trie temp;
    Trie * ptr = 0;
    int k;
    bool somethingBelow = false;

    if (lhs == 0) return CopyTrie(rhs);
    else if (rhs == 0) return CopyTrie(lhs);
    else
        { // make recursive calls
            for(k=0; k < ALPH_SIZE; k++)
                {
                    if (temp.index[k] = TrieUnion(lhs->index[k], rhs->index[k]))
                    {
                        somethingBelow = true; // something found
                    }
                }
            temp.isWord = lhs->isWord || rhs->isWord; // in union?
            if (somethingBelow || temp.isWord) // something to return
                {
                    ptr = new Trie;
                    // make new node
                    *ptr = temp; // copy temp to ptr
                }
            return ptr;
        }
}
\end{verbatim}

Steps in writing \textit{TrieIntersect}

(a) Make the recursive calls to see what is in the intersection below
(b) Examine \textit{isWord} fields to determine if the current node/path is in the intersection
(c) Return an empty trie unless there is something below or the current node/path is in the intersection
Trie* TrieIntersect(Trie * a, Trie * b)
// postcondition: returns a trie that is the intersection of a and b
// i.e., every word in a AND in b is in returned trie
{

PROBLEM 8:  (Rankings (6 points))
You are given a vector of \( n \) ints (where \( n \) is very large) and are asked to find the largest \( m \) of them (where \( m \) is much less than \( n \)).

a. Design an efficient algorithm to do this. You can just write the algorithm in English, you do not have to use C++ code. However, your language should be precise. For example, “Put them all in a map and take out the top \( m \)” is not a good answer. You would need to specify what kind of map and exactly how you would obtain the top \( m \).

Assume that you have all available data structures described in class. Your algorithm should work well for all values of \( m \) and \( n \), from very small to very large.

b. What is the running time of your algorithm? What is it for small \( m \)? What is it as \( m \to n \)?

Extra Credit

Name a member of the Duke programming team that finished second in the ACM Mid-Atlantic USA Programming Contest on November 8, 2003 and qualified for the world finals in Prague. Hint: one of them is a UTA for this course.
Definitions

Some common recurrences and their solutions.

\[
\begin{align*}
T(n) &= T(n/2) + O(1) \quad O(\log n) \\
T(n) &= T(n/2) + O(n) \quad O(n) \\
T(n) &= 2T(n/2) + O(1) \quad O(n) \\
T(n) &= 2T(n/2) + O(n) \quad O(n \log n) \\
T(n) &= T(n-1) + O(1) \quad O(n) \\
T(n) &= T(n-1) + O(n) \quad O(n^2)
\end{align*}
\]

// Tree definition
struct TreeNode
{
  string info;
  TreeNode * left;
  TreeNode * right;
  TreeNode (int val, TreeNode * lt, TreeNode * rt)
  {
  info(val), left(lt), right(rt)
  }
};

// Graph definition
class Graph
{
public:
  Graph();
  ~Graph();

  int addVertex(const string& s);
  void addEdge(const string& from, const string& to);
  void addEdge(const string& from, const string& to, double weight);
  void addUndirectedEdge(const string& from, const string& to);
  void addUndirectedEdge(const string& from, const string& to, double weight);

  void removeEdge(const string& from, const string& to);

  int getNum(const string& s) const;
  string getName(int index) const;
  int vertexSize() const;
  int edgeSize() const;
  double getWeight(const string& from, const string& to) const;
  double getWeight(int from, int to) const;
  bool hasEdge(const string& from, const string& to) const;

  void getVertices(vector<string>& list) const;
  void getAdjacent(const string& vertex, vector<string>& list) const;

  void clear();

private:
  // ...
};