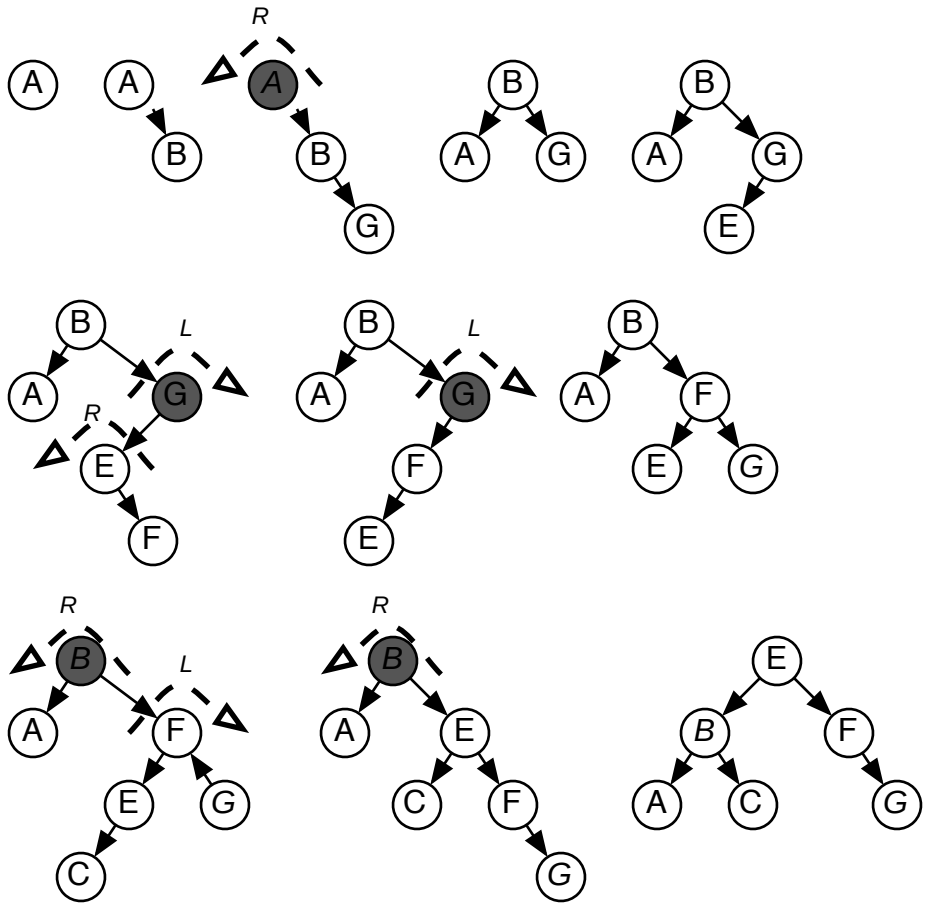
DUKE UNIVERSITY
Department of Computer Science

CPS 100 **J. Forbes**
Fall 2003

Test #2 Solutions

PROBLEM 1 : (*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.

	12	21	31	36	32	38		

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

	12	21	23	36	32	38	31	

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

***insert*: $O(\text{height}) = O(\log n)$ for a AVL tree**

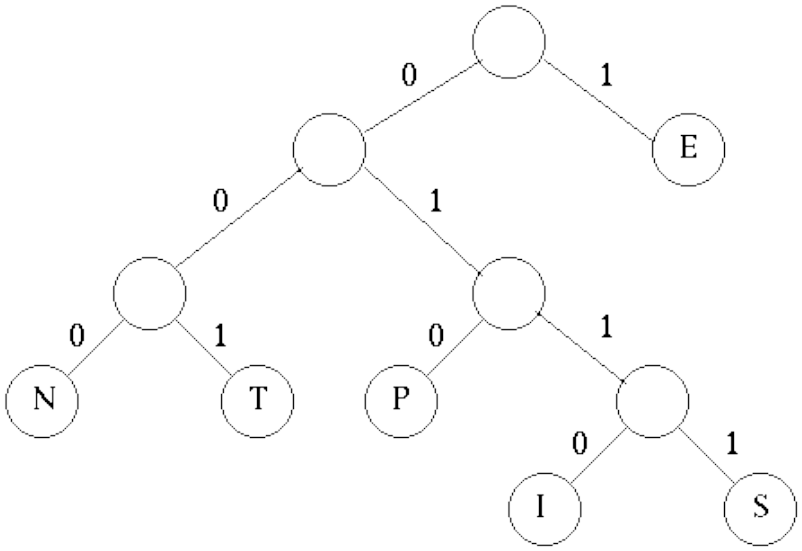
***deleteMin*: $O(\log n)$ for find min + $O(\log n)$ for delete = $O(\log n)$**

c. How do those complexities compare to the respective values for a heap? (slower, same, or faster)

Both *insert* and *deleteMin* are the same big-Oh. You could say that they are slower by a constant factor because of the additional overhead in balancing the tree.

PROBLEM 3 : (*David's work (10 points)*)

1. Given the following Huffman tree:



Write the Huffman encoding for the string "SENT".

01111000001

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?

A good tree is a long tree. An example of character frequencies which yield good compression is the Fibonacci sequence where each number is the sum of the previous two numbers.

A bad tree is a full tree.

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?

The new file will be larger since all 8-bit sequences will be equally likely to appear and the resulting tree will be a full tree. The problem is that the new tree will take up some space in the new file and the resulting file willl be larger.

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

- **Already or nearly (i.e. few inversions) sorted data**
- n **is small**

b. Merge Sort

- **Need guarantees for performance**
- **Using linked lists**

c. Radix Sort

- **Limited number of digits,** d**, is less than** $\log n$
- **Short strings**

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)

```
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;                 // T(0) = O(1)

  // do the subtrees
  doubleTree(node->left);              // T(n/2)
  doubleTree(node->right);             // T(n/2)

  // duplicate this node to its left
  oldLeft = node->left;                // O(1)
  node->left = new TreeNode(node->info, 0, 0); // O(1)
  node->left->left = oldLeft;          // O(1)
}
```

$T(n) = 2T(n/2) + O(1) \in O(n)$

b. *isBST3* where n is the number of nodes in the tree (assume worst case)

```
bool ValsLess(TreeNode *t, string key)
// post: returns true iff all nodes in t are less than key
{
  if (t == 0) return true;           // T(0) = O(1)
  return t->info < key &&            // O(1)
         ValsLess(t->left, key) &&   // T(n/2)
         ValsLess(t->right, key);    // T(n/2)
}
```

$T(n) = 2T(n/2) + O(1) \in O(n)$

or in equivalent worst case

$T(n) = T(n-1) + +T(1) + O(1) = T(n-1) + O(1) \in O(n)$

```
bool ValsGreater(TreeNode *t, string key)
// post: returns true iff all nodes in t are less than key
{
  if (t == 0) return true;            // T(0) = O(1)
  return t->info > key &&             // O(1)
         ValsGreater(t->left, key) && // T(n/2)
         ValsGreater(t->right, key);  // T(n/2)
}
```

$T(n) = 2T(n/2) + O(1) \in O(n)$

or in equivalent worst case

$T(n) = T(n-1) + +T(1) + O(1) = T(n-1) + O(1) \in O(n)$

```
bool IsBST3(Tree * t)
// postcondition: returns true if t represents a binary search
//                tree containing no duplicate values;
//                otherwise, returns false.
{
    // T(0) = O(1)
    if (t == NULL) return true;          // empty tree is a search tree

    return ValsLess(t->left,t->info)     &&  // O(n/2)
           ValsGreater(t->right,t->info) &&  // O(n/2)
           IsBST3(t->left)                &&  // T(n/2)
           IsBST3(t->right);                  // T(n/2)
}
```

$T(n) = 2T(n/2) + O(n) \in O(n \log n)$

or in worst case

$T(n) = T(n-1) + O(n) \in O(n^2)$

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.
***An* order: 0 2 1 4 3 5**

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.

***An* order: 0 2 5 1 4 3**

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
    tvector<Trie *> links;   // links to child nodes
    Trie()
        : isWord(false),
          links(ALPH_SIZE,0)
        { }
};
```

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

```
// Recursive version
void AddWord(Trie * & t, string word)
// post: word added to Trie t

{
  if (t == 0)
  {
```

```
    t = new Trie;
  }
  if (word.length() == 0)
    t->isWord = true;
  else
  {
    int chIndex = word[0];
    AddWord(link[chIndex], word.substr(1));
  }
}

/* Iterative version */
void AddWord(Trie * & t, string word)
    int len = word.length();
    int k;
    if (t == 0)              // if initially empty, make new trie
        t = new Trie;

    Trie * hold = t;
    for(k=0; k < len; k++)
    {
        int chIndex = word[k];
        if (hold->index[chIndex] == 0)
        {
            hold->index[chIndex] = new Trie;
        }
        hold = hold->index[chIndex];
    }
    hold->isWord = true;  // ends on a word, make it so
}
```

b. In this problem, you will write a function *TrieIntersect* that returns a trie that is the intersection of two tries.

You may assume a function *CopyTrie* is already written. Its prototype is below:

```
Trie * CopyTrie(Trie * root);
// post: returns copy of trie pointed to by root
```

You might want to model your function after the following definition of *TrieUnion*:

```
Trie * TrieUnion(Trie * lhs, Trie * rhs)
// postconditon: 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.links[k] = TrieUnion(lhs->links[k],rhs->links[k]))
            {
                    somethingBelow = true;         // something found
            }
        }
        temp.isWord = lhs->isWord || sec->isWord;   // in union?
        if (somethingBelow || temp.isWord)            // something to return
        {
            ptr = new Trie;                           // make new node
            *ptr = temp;                              // copy temp to ptr
        }
        return ptr;
    }
}
```

Steps in writing *TrieIntersect*

(a) Make the recursive calls to see what is in the intersection below

(b) Examine `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)
// postconditon: returns a trie that is the intersection of a and b
//               i.e., every word in a AND in b is in returned trie
{
    Trie temp;
    Trie * ptr = 0;
    int k;
    bool somethingBelow = false;

    // nothing in intersection if empty
    if (a == 0) return 0;
    else if (b == 0) return 0;
    else
    {
        // make recursive calls

        for(k=0; k < ALPH_SIZE; k++)
        {
            if (temp.links[k] = TrieUnion(lhs->links[k],rhs->links[k]))
            {
                    somethingBelow = true;         // something found
            }
        }
        temp.isWord = a->isWord & b->isWord;   // in intersection?
        if (somethingBelow || temp.isWord)            // something to return
        {
            ptr = new Trie;                           // make new node
```

```
            *ptr = temp;                                    // copy temp to ptr
        }
        return ptr;
    }
}
```

PROBLEM 8 : (*Rankings (6 points)*)
You are given a vector of n `int`s (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 $m \to n$? **Possible solutions:**

   1. **Find the min (not a good solution)**
      i. **Find the minimum element m times ($O(mn)$)**

      This solution approaches $O(n)$ for small m and approaches $O(n^2)$ as $m \to n$.
   2. **The sort solution**
      i. **QuickSort (or MergeSort) the vector ($O(n \log n)$)**
      ii. **Print top m from vector ($O(m)$)**

      This solution is O($n \log n$) not (O($n \log n + m$) no matter what m is.
   3. **The Heap solution**
      i. **Heapify the vector $O(n)$**
      ii. **Call *deleteMin* m times ($O(m \log n)$)**

      This solution is $O(n + m \log n)$ which is $O(n)$ for small n and $O(n \log n$ as $m \to n$.
   4. **Radix?**
      i. **Radix sort the vector as 32-bit binary numbers ($O(32n + 32 \cdot 2)$) or as 10 digit decimal numbers ($O(10n + 10 \cdot 10)$)**
      ii. **Print top m from vector ($O(m)$)**
      This solution is more $O(n)$ but the constant factor is high. It's performance does not depend on m though.

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.

Andrew Dreher, Ethan Eade, and Garrett Castro.