University of Illinois at Urbana-Champaign
Department of Computer Science

Third Examination

CS 225 Data Structures and Software Principles
Summer 2005
3:00pm – 4:15pm Wednesday, July 27

Name:	SOLUTIONS
NetID:	
Lab Section (Day/Time):	

- This is a **closed book** and **closed notes** exam. No electronic aids are allowed, either.
- You should have 6 sheets total (the cover sheet, plus numbered pages 1-11). The last sheet is scratch paper; you may detach it while taking the exam, but must turn it in with the exam when you leave. can use this sheet as reference while taking the exam.
- Unless otherwise stated in a problem, assume the best possible design of a particular implementation is being used.
- Unless the problem specifically says otherwise, (1) assume the code compiles, and thus any compiler error is an exam typo (though hopefully there are not any typos), and (2) assume you are NOT allowed to write any helper methods to help solve the problem, nor are you allowed to use additional arrays, lists, or other collection data structures unless we have said you can.

Problem	Points	Score	Grader
1	15		
2	15		
3	15		
4	15		
5	20		
6	10		
Total	90		

1. **[AVL Trees – 15 points].**

   You are given the `AVLTreeNode` class shown on page 8 of the exam. You want to write a function `doubleRight` that has one parameter and returns nothing. The parameter will be of type reference-to-pointer-to-`AVLTreeNode`. You can assume this parameter points to the root of a subtree for which it is possible to perform a double right rotation. The function should perform that rotation – including fixing the heights of whatever nodes will need their heights fixed. You can assume all heights in all nodes, are correct prior to the double rotation operation.

```
void doubleRight(AVLTreeNode * & ptr) {
   // your code goes here

   TreeNode* firstRot = ptr->left;

   TreeNode* temp = firstRot->right;
   firstRot->right = temp->left;
   temp->left = firstRot;
   firstRot = temp;
   // fix height

   ptr->left = temp->right;
   temp->right = ptr;
   ptr = temp;
   // fix height
```

2. **[Tries – 15 points].**

   You are given the `TrieNode` class shown on page 8 of the exam (as well as the companion `TrieListNode` class on that same page). This is the node class for a de la Briandais tree.

   You want to write a function `doubleStart` that has one parameter, of type pointer-to-`TrieNode`, and which returns an int. Your function should return the number of `TrieNodes` which are the starting nodes of a double letter pattern – that is, nodes for which a there is a subtree at a given letter, which also contains that letter. For example, if the `'c'` subtree existed for some node *mynode*, and the root node of that subtree, *also* had a `'c'` subtree that existed, then *mynode* qualifies as one of the nodes you should count. If a node starts multiple double-letter patterns – for example, if the node has a `'c'` subtree which itself has a `'c'` subtree, *and* it has an `'s'` subtree which itself has an `'s'` subtree – you would still only count the node once.

```
int doubleStart(TrieNode * ptr) {
   // your code goes here
```

3. **[Node Creation – 15 points].**

   You have the DSNode class shown on page 8 of the exam, as well as the Array class shown on page 9 of the exam. You want to write a function makeTrees, which has one parameter, an Array<int> named sets. This array will be indexed from 1 through sets.size(), representing a collection of disjoint sets implemented with union-by-size. You want to return a value of type Array<DSNode*> (you are allowed to create this one new array in your function, but no other additional new arrays besides that one), which will implement the exact same collection of disjoint sets as the parameter array – only using actual uptree nodes rather than simply array cells. Your returned array should be indexed from 1 through sets.size(), and each cell should point to a DSNode containing that cell's index as the node key. Those DSNode objects should have their variables initialized in such a way that they exactly implement the same uptrees represented by the parameter array (i.e. do not compress any paths).

```
Array<DSNode*> makeTrees(Array<int> sets) {
   // your code goes here

   Array<DSNode*> dynamicSets(1, sets.size());
   for (int i = 1; i <= dynamicSets.size(); i++)
      dynamicSets[i] = new DSNode(i);

   for (int i = 1; i <= sets.size(); i++)
      dynamicSets[i].parent = dynamicSets[sets[i]];

   return dynamicSets;
}
```

4. **[Complete Trees – 15 points].**

   You want to write a method `countLeftFirst` that has one parameter, an `Array<int>` value (see page 9 for the `Array` class) named `tree`, indexed from `1` to `tree.size()`. This array is the array implementation of a complete tree of size `tree.size()`. You want to return the number of values in the array for which the value's "left child" is both less than the value's "right child" and also less than the value itself. Any value that does not have both a left and right child, would automatically not be counted in this total.

```
int countLeftFirst(Array<int> tree) {
   // your code goes here

   int total = 0;
   for (int i = 1; i <= tree.size()/2; i++)
      if ((tree[2 * i] < tree[i]) && (tree[2 * i] < tree[2 * i + 1]))
         total++;

   return total;
}
```

5. **[Red-Black Trees – 20 points].**

   (a) You have the values `1` through `15`, inserted into a red-black tree of all black nodes, that also happens to be a perfect tree. If you remove `1` from the tree, which nodes (if any) will now be red? Briefly explain why. (You do not need to justify that parts of the red-black tree removal algorithm are correct; it's enough to say, "in this case, the algorithm says to do this".)

   The tree has to look like this to begin with:

```
             8
           /   \
          4      12
        /  \     / \
       2   6  10   14
      /\  /\  /\   / \
     1 3 5 7 9 11 13 15
```

   So when you remove 1, pointer to 1 becomes pointer to "x". Now, you have case 3: x's sibling, which is 3, is black with black children. So you handle this by coloring 3 red and relabelling 2 as "x". But 2's sibling – which is 6 – is also black with black children...so you have case 3 again. You color 6 red and relabel 4 as "x". Same idea there: 4's sibling – which is 12 – is black with black children. So color 12 red, and relabel 8 as "x". Now "x" is the root, so you exit the loop.

   So you have three nodes that are red: 3, 6, and 12.

(b) You have three values in a red-black tree – a root with two children, with the two children being red. What is the maxium number of rotations that the next insertion could result in? Justify your answer.

You will not have any rotations. Your next insertion has to be a leaf, so it has to be a child of one of the root's two children. A new node could not possibly go anywhere else, according to the BST insert algorithm. No matter which of the root's two children is the parent of our new node, our new node will be red, and it's parent and sibling will both be red. So this is the "red uncle" case of the insert algorithm; we solve this by coloring the new node's parent and parent's sibling black, the grandparent red, and relabelling the grandparent as "x". So now we have a red root with two black children, one of which has a red leaf as a child. The last thing we do is "leave the loop" and color the root black again...and then we are done. So no rotations are needed at all.

6. **[Analysis – 10 points].**

   You want to create a two-dimensional sparse array. That is, you want an interface that lets you pass in a "row" index and "column" index and, using those indices, gives you the unique value associated with that pair of indices. However, behind the scenes, we will not necessarily implement this with a two-dimensional array.

   Specifically, we are deciding between two possible implementations. One of our options is a linked list whose nodes each hold a row index and an array of integers. In this case, to look up a value, we first search the linked list for the desired row, and then, given the array at that linked list node, use our column index as the array index we want – the value in that cell is what we are looking for.

   The second option is a red-black tree whose nodes each hold a row index and an AVL tree. Searching the red-black tree for the row index will give us the relevant AVL tree. The AVL tree will hold two indices in each node – a column index and the value associated with that column for the given row we looked up in the red-black tree. So to obtain our value, we then search the AVL tree for the column index.

   If our goal is to have the fastest worst-case time we can for obtaining a value, given the value's row and colum, then which of these two implementations is preferable? Justify your answer convincingly. You can assume any running time we have stated in class about linked lists, arrays, red-black trees, or AVL trees.

   THE ANSWER: For our first option, array lookup is done by searching the linked list for a row value (this would be $\mathcal{O}(n)$) and then once we find it, access the array there to get the column we want (which would be $\mathcal{O}(1)$). So, we have $\mathcal{O}(n) + \mathcal{O}(1) = \mathcal{O}(n)$.

   Our second option is to search a red-black tree for a row value (this would be $\mathcal{O}(\log n)$) and then once we find it, access the AVL tree there and search it to get the column we want (this would also be $\mathcal{O}(\log n)$). So we have $\mathcal{O}(\log n) + \mathcal{O}(\log n) = \mathcal{O}(\log n)$.

   So the second way is the better of the two, since in order to find the value you need to acces *both* structures, and that total time will be better in the second implementation.

```
class AVLTreeNode {  // needed for problem 1
public:
   int element;
   int height;
   AVLTreeNode* left;
   AVLTreeNode* right;
};

class TrieNode {  // needed for problem 2
public:
   TrieListNode* head;   // see below
   int level;
};


class TrieListNode   // needed for class above which is needed for problem 2
public:
   char letter;
   TrieNode* subtree;
   TrieListNode* next;
};

class DSNode {  // needed for problem 3
public:
   int key;
   int size;   // for non-root nodes, this value can be anything you like
   DSNode* parent;

   DSNode(int value) { key = value; size = 1; parent = NULL; }
};
```

```
template <typename Etype>
class Array  {    // needed for problems 3 and 4

   // Here are the member function declarations for the Array class;
   // we've left the declarations for the variables, iterators and iterator
   // support functions (begin(), end(), etc.) out; you don't need them.
   Array();   // size 0 array, indiced 0 through -1
   Array(int low, int high);   // indices low through high
   Array(Array<Etype> const & origVal);  // copy constructor
   ~Array();                             // destructor
   Array<Etype> const & operator=(Array<Etype> const & origVal);//assignment op
   Etype const & operator[](int index) const;   // accesses cell at param index

   Etype & operator[](int index);   // accesses cell at param index
   void initialize(Etype const & initElement); // inits all cells to param
   void setBounds(int theLow, int theHigh);  // changes bounds of array,
   int size() const;   // returns number of cells in array
   int lower() const;  // returns lowest index
   int upper() const;  // returns upper index
};
```

(scratch paper, page 1)

(scratch paper, page 2)