University of Illinois at Urbana-Champaign
Department of Computer Science

Second Examination

CS 225 Data Structures and Software Principles
Sample Exam 1
75 minutes permitted

Print your name, netID, and lab section day/time neatly in the space provided below; print your name at the upper right corner of every page.

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

- This is a **closed book** and **closed notes** exam. In addition, you are not allowed to use any electronic aides of any kind.
- Do all 4 problems in this booklet. Read each question very carefully.
- You should have 7 sheets total (the cover sheet, plus numbered pages 1-12). 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.
- 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	20		
2	20		
3	20		
4	30		
Total	90		

1. **[Analysis – 20 points (10 points each)].**

   (a) You are given a `BinarySearchTree` class whose `insert` method inserts its parameter into the binary search tree, using the usual algorithm. Express, using big-$\mathcal{O}$ notation, the order of growth of the worst-case running time of the following code, as `n` increases. Justify your answer.

```
BinarySearchTree bst;  // creates an empty binary search tree
for (int i = 1; i <= n; i++)   // this is the n we are talking about above
   bst.insert(i);
```

(b) You have a min-heap of n values. You have a function `IncreasePriority(int i, int addition)` which will take the value at index `i`, and increase its priority value by adding the value `addition` to it. Now that the priority of this value has been increased, you might not have a legal min-heap any longer, so you need to then do the minimum work necessary to repair the min-heap. `IncreaseKey` does all that. What is the worst-case running time of `IncreaseKey`? Express your answer in Big-$\mathcal{O}$ notation and explain convincingly why your answer is correct.

2. **[Verifier – 20 points].**

Given the following two classes:

```
class IntTriple {
public:
   int first;
   int second;
   int third;
};
```

```
class TreeNode {
public:
   int element;
   TreeNode* left;
   TreeNode* right;
};
```

you want a function `Verify` that takes one parameter, a pointer to a `TreeNode`. This function should return an object of type `IntTriple`, whose member variable `first` will hold `1` if the parameter pointer points to a binary search tree, and which will instead hold `0` if the parameter pointer points to a binary tree that is NOT a binary search tree. You can use the other two variables of the `IntTriple` however you like. Please note that if the parameter pointer is NULL, then that *is* considered to be a binary search tree (although an empty one).

```
IntTriple Verify(TreeNode* ptr) {
   // your code goes here
```

(Verifier, continued)

3. **[Lists of Tree Values – 15 points].**

   You have the following two standard node classes:

```
class ListNode {
public:
   int element;
   ListNode* next;
   ListNode* prev;
};
```

```
class TreeNode {
public:
   int element;
   TreeNode* left;
   TreeNode* right;
};
```

   Write a function `TreeToList` which takes as a parameter, a pointer to a `TreeNode` which is the root of a *binary search tree*. This function should return a `ListNode` pointer to a list of all elements in the binary search tree, in numerical order from lowest to highest (i.e. your list should be sorted). If the parameter `TreeNode` pointer is `NULL`, then the returned `ListNode` pointer should also be `NULL`. (Hint: some recursion can help you here.)

```
ListNode* TreeToList(TreeNode* ptr) {
   // your code goes here
```

(List of Tree Values, continued)

4. **[Algorithms – 30 points (6 points each)].**

   (a) Explain convincingly that Level-Order traversal on a binary tree of n nodes is $\mathcal{O}(n)$.

(b) Assume you have a disjoint set structure. Furthermore, assume you are NOT using path compression. Justify the use of the union-by-size smart-union algorithm, rather than the union-by-height smart-union algorithm. That is, why wouldn't you *always* use union-by-height in these circumstances? What advantage does union-by-size give you that can make it worth using?

(c) The AVL Tree balance property was that for any node, the two subtrees of that node differed by at most `1` in height. Why not `0`? That is, explain convincingly that requiring each node to have two subtrees of equal height would be problematic.

(d) In the red-black tree insertion algorithm, all the cases for handling a red-parent-with-red-child conflict (we'll refer to this as a "red-red conflict") assume that there is another level above the red-red conflict – i.e. they assume that the upper red node in the red-red conflict has a parent. We did not have any case to handle the situation where you have a red node, and the parent of that node happens to be red, but that red parent node has no parent itself. Explain convincingly why we do not need such a case.

(e) Explain convincingly that for a B-Tree of order `b` (remember, we said the order, `b`, would always be an odd number), splitting a B-tree node during an insertion must always leave an extra index value that could be moved into the parent of the split node.

(scratch paper)