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: SOLUTIONS
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);
```

Since we are inserting the values in numerical order, at any point while the above code snippet is running, our binary tree will have no left subtrees – i.e. if there are k nodes, they are in a single line connected by right pointers, with the left pointer of every node pointing to `NULL`.

In addition, again since we are inserting the values in numerical order, each insertion will move down the "list" of all the values that have been inserted earlier – that is, we'll keep travelling to the right, through the nodes 1, 2, 3, 4, ... k-1, before we reach a NULL pointer and can insert value k. This is because, if you are inserting k, whatever node you are at contains some value between 1 and k-1, inclusive, so it is guaranteed to be less than k and you are guaranteed to move to the right subtree with your recursive call.

And each recursive call takes constant time. So, the first insertion needs only one recursive call, inserting the value 2 needs two calls total, inserting the value 3 needs 3 calls total, etc. To insert the values 1 through n, you will need 1 + 2 + 3 + ... + n calls, and that sum is quadratic.

(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.

If the priority value within a min-heap nodes increases, it will still be greater than its parent – since it was already greater than its parent (due to the whole structure being a legal min-heap prior to the `IncreasePriority` operation) and it's just gotten bigger. However, before the operaton, the value was smaller than its children, and now there is a possibility is is larger than its children. In fact, if we consider the subtree that this node is a root of, this node used to be the smallest node in that subtree, by definition, and yet now it is potentially not the smallest value anymore.

So we have a situation where there is a complete tree (the subtree rooted at this node) where everything is partially ordered except for possibly the root value. This is no different than the stage of DeleteMin right after writing your last value into the first cell. i.e., to summarize all this up, the IncreasePriory operation requires nothing more than taking the increased priorty value and percolating it downward in the same way you would do for DeleteMin or BuildHeap. And since the heap has height $\mathcal{O}(\log n)$, this operation is at most $\mathcal{O}(\log n)$.

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
   SOLUTION ON NEXT PAGE
```

(Verifier, continued)

```
IntTriple Verify(TreeNode* ptr) {
   // we will use the other two variables of an IntTriple to hold
   //  the min and max of the tree
   IntTriple returnVal;

   if (ptr == NULL) {
      returnVal.first = 1;   // empty binary tree is a BST
      returnVal.second = 0;  // min of empty tree
      returnVal.third = -1;  // max of empty tree, it being < min
                             // signals it was empty tree
   }
   else {
      IntTriple left, right;
      left = Verify(ptr->left);
      right = Verify(ptr->right);
      returnVal.first = 1;     // assume it's a BST unless we find otherwise
      returnVal.second = ptr->elem; // assume root value is min and max
      returnVal.third = ptr->elem;  //    unless we find otherwise

      if ((left.first == 0) || (right.first == 0))
         returnVal.first = 0;  // not BST if subtrees not BSTs

      // if root < max of left subtree, or root > min of right subtree, not BST
      if (((left.second <= left.third) && (left.third > ptr->element)) ||
          ((right.second <= right.third) && (ptr->element > right.second)))
         returnVal.first = 0;

      if (left.second <= left.third) { // if left subtree exists
         if (left.second < returnVal.second) // if smaller min, save it
            returnVal.second = left.second;
         if (left.third > returnVal.third)  // if bigger max, save it
            returnVal.third = left.third;
      }

      if (right.second <= right.third) {  // if right subtree exists
         if (right.second < returnVal.second)  // if smaller min, save it
            returnVal.second = right.second;
         if (right.third > returnVal.third)    // if bigger max, save it
            returnVal.third = right.third;
      }

      return returnVal;
   }
}
```

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
   SOLUTION ON NEXT PAGE
```

(List of Tree Values, continued)

```
ListNode* TreeToList(TreeNode* ptr) {
   // your code goes here
   if (ptr == NULL)
      return NULL;
   ListNode* beforeme;
   ListNode* afterme;
   ListNode* me;

   me = new ListNode();         // node of the list with
   me->element=ptr->element;    // 'root' element
   beforeme = TreeToList(ptr->left);  //first listnode of 'left subtree'
   afterme = TreeToList(ptr->right);  //first listnode of 'right subtree'

   me->next = afterme;               // point next of 'root' node to
                                     // first node of 'right subtree'

   if (afterme != NULL)              // if there are nodes in the right subtree
      afterme->prev = me;            // point prev of first node of
                                     // 'right subtree' to 'root' node
   if (beforeme != NULL) {
      ListNode* temp = beforeme;
      while(temp->next != NULL)       // traverse to the end node of
         temp=temp->next;             // 'left subtree' list
      temp->next = me;           // point next of end node of'left subtree'
                                 //    to 'root' node
      me->prev=temp;             // point prev of 'root' node to
                                 // end node of 'left subtree'
      return beforeme;
   }
   else
      return me;
}
```

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

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

There are 2n+1 total vertex pointers in the tree, counting the root pointer – n of those to real nodes, and n+1 of them to null pointers. Over the lifetime of this algorithm, every one of these pointers will be enqueued – the root pointer is enqueued before the loop begins, and every other pointer is a child of some node, and when you dequeue that node, you'll enqueue its children. And, every one of those pointers will be dequeued, since the loop runs until the queue is empty. So there are 2n+1 enqueues and 2n+1 dequeues. Other than that, there's just a bit of additional overhead per pointer – there's one loop pass (and thus one loop comparison) for each dequeue, plus the last loop comparison which determines the queue is empty. And, every time we dequeue a pointer, we check it to see if it's null or not. So, overall, for each of 2n+1 pointers, you will (1) enqueue the pointer, (2) dequeue the pointer, (3) check to see if it's null, and (4) when that pointer is in front of the queue, the loop condition will be checked. All that adds up to constant time, and constant times 2n+1 is a linear-time operation.

(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?

If we call the tree that gets pointed to, "A", and refer to the tree whose root node is changed to point to "A", via the label "B", then any union will necessarily increase the depth of every node in "B" by 1, since now those nodes go through an extra edge – the new pointer from B's former root, to A – en route to their new root, A. By pointing the smaller sized tree to the larger sized tree, you increase the depth of fewer nodes than if you had done things the other way around. That is, union-by-size keeps the expected depth of a node as small as possible, by always choosing to increase the depth of fewer nodes rather than more nodes.

(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.

Such a requirement simply isn't possible to meet for many trees. For example, if you have four values, one must be at the root, two values on one side of the root, and one value on the other side of the root. The side with one node is a subtree of height 0; there is no way to arrange the two nodes on the other side to form a subtree of height 0.

(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.

If the node we are inspecting has a red parent but no grandparent, it means the parent of the node we are inspecting is the root. And yet, since we work from the bottom upward when rebalancing and recoloring the tree, if we are currently inspecting the level below the root, it means we have not reached the root yet. Therefore, we could not have colored the root red yet ourselves. And the root is black at the start of every insertion operation. So the case described above simply cannot happen – if we hadn't reached the root yet, it would still be black for certain, so there can be no case where we are inspecting a node and it has a red root for a parent.

(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.

The maximum number of children a B-tree node is allowed to have is `b` children, and you always have one fewer indices than children. So, if the node is overflowing, it means you have `b+1` children and thus `b` indices. When you split that, half the children go to each half, so each half gets `(b+1)/2` children, and since you always have one fewer indices than children, each half should also have `((b+1)/2)-1` indices. But, if each half has `((b+1)/2)-1` indices, the two halves together have `(((b+1)/2)-1) + (((b+1)/2)-1)` indices, which equals `b-1` indices. Yet, our node before the split had `b` indices. So, we know there is an index left over that was not placed in either of the two halves of the split.

(scratch paper)