University of Illinois at Urbana-Champaign
Department of Computer Science

Second Examination

CS 225 Data Structures and Software Principles
Sample Exam 2
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 5 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	12		
2	30		
3	18		
4	15		
5	15		
Total	90		

1. **[Short Answer – 12 points (4 points each)].**

   (a) What was the "problem" with using path compression and union-by-height together? That is, what difficulty does using the two techniques together present? Please be specific. (The word "problem" is in quotes because we said it turned out that this "problem" didn't actually affect things too badly, even if it seems like it would.)

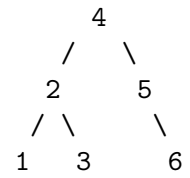
   Path compression (potentially) changes the height of the tree, due to (potentially) shortening a path that was (potentially) the deepest path in the tree. In that case, the height stored at the root would become incorrect, as neither path compression nor union-by-height has any provision for recalculating the correct height in such a circumstance (and adding one would make either algorithm far too expensive, time-wise).

   (b) If you have a complete tree of 17 nodes, how many nodes are on the deepest level?

   2

   (c) Insert the integers `1` through `6`, in that order, into an AVL tree. Draw the resulting tree. How many rotation operations, total, did you perform? Count a "double rotation" operation as one rotation operation.

   3 rotations total



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

(a) Explain why an in-order traversal on a binary search tree should produce the values of the tree in lexigraphical order (i.e. numerical, alphabetical, or whatever the order is that is appropriate for those values).

An in-order traversal will process all the values in the left subtree of the root before processing the root. Now, since the left subtree consists of all the values in the tree that are less than the root, all the values less than the root will be processed before the root itself. Likewise for the right subtree – an in-order traversal will process all the values in the right subtree of the root, *after* the root is processed, and since all the values greater than the root are in the right subtree, that means the root is processed prior to the processing of any values greater than the root. So the root is guaranteed to be placed in the correct order relative to its descendants. And since this process is recursive, *every* node is guaranteed to be placed in the proper order relative to its descendants, and thus all the values are in the proper order.

(b) In an AVL tree, why does storing the height of a subtree, in the root node of that subtree, improve the efficiency of the AVL rebalancing work (versus not storing the height at all)?

We want to be able to recalculate the height of a node quickly, and yet, to calculate the height of a node, we would need to find the heights of both subtrees, so that we could compare the heights and do the appropriate arithmetic. If the height of every node is stored in that node, then given a pointer to a node, learning the heights of that node's children is simply a matter of reading them from the child nodes – which is constant time. But if we need to calculate the height of your node from scratch, then we will need to make a recursive call on every descendant of that node, which could take as long as linear time, rather than constant time. So, storing heights in nodes means that, at each node, knowing the heights of subtrees (which we need for rebalancing) can be accomplished in constant time, not linear time.

(c) After performing a combine operation during B-Tree removal, why is it that we need to check the parent for underflow? i.e. justify that such a combine operation could have caused the parent to underflow.

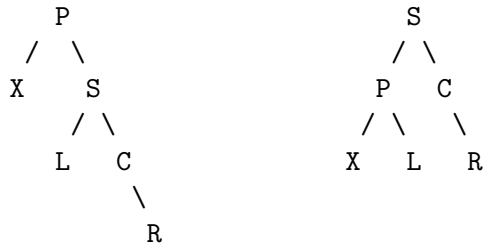
Combining a node merges two nodes into one...so where the parent used to have two nodes, it now has just one. So if the parent had the minumum number of children before, it now has one less than the minimum number of children, because two of those children have been combined into one.

(d) Explain why we can implement a complete tree using an array – that is, explain why we don't lose information when we get rid of the pointers, i.e. explain why it is that, given an array, we can always produce the corresponding complete tree.

Because of the way a complete tree is defined, the number of nodes defines the structure – that is, every complete tree of N nodes, has those nodes in the same place as every other complete tree of N nodes. And the array is just the level order traversal of the complete tree. So, given an array with N elements, we know exactly what structure the tree is supposed to have, and we know how that tree was traversed to produce the array, meaning we know what array cells correspond to what nodes in the tree.

(e) Explain the "repair case" of the Red-Black Tree removal algorithm (the "repair case" was case 2b, where the node we labelled "x" had a black sibling and that black sibling had a red child in the child position further from "x'). That is, explain what we do in this case and justify that it fixes the problems we have without causing new ones.

If the parent of X is black, and we perform a single rotation on the parent toward X, and color the new subtree root (S below) and its two children (P and C below) black:



Then we used to have one black node on the way to X, namely, P, but now we encounter both S and P on the way to X. On the other hand, there are still two black nodes on the way to L, two black nodes on the way to C's left child, and two black nodes on the way to R, since previously, P and S were black and C was red, and now, S and C are both black. In other words, we add one black node to all paths containing X, but the paths that do not contain X have the same number of black nodes as before.

Similarly if P is red - same rotation, but S becomes red and P and C (as before) become black. The same analysis as above is true, but since S is red instead of black, all paths have one fewer black node than they did in the above paragraph. Nevertheless, we have added one black node to the paths through X, while not changing the black heights of the paths that do not travel through X.

3. **[Analysis – 18 points (9 points each)].**

   (a) If you want to remove some value from a min-heap – not necessarily the minimum value, just some random value from the heap – one way you could go about this would be to decrease the priority of the value so that it rises to the top of the heap – i.e. decrease the priority of the value so that it is the minimum value in the heap – and then perform a `DeleteMin` operation. Assuming you already know where the value you want to remove is located in the min-heap, what would be the order of growth of the running time of the above removal procedure? Express your answer in big-$\mathcal{O}$ notation and justify your answer.

   To decrease the priority of a value and percolate it to the top would potentially be the same as the cost of insertion – i.e. the cost of swapping a value upward from the deepest level of the heap to the root. That cost is $\mathcal{O}(\lg V)$. And a delete min is also $\mathcal{O}(\lg V)$. So total, the cost of this operation will be $\mathcal{O}(\lg V)$.

(b) Explain why it is that the rebalancing work performed by the AVL tree insert or remove is at most $\mathcal{O}(\lg n)$ on a tree of height $\mathcal{O}(\lg n)$. Your answer should be detailed enough to convince us you know what you are talking about. You don't need to *justify* the steps of the algorithm here – simply indicate what those steps are and their running times – and indicate that those running times add up to what we claim they add up to.

As you return from the BST recursive calls (after doing BST insert or BST remove), at each node there are three things that need doing:

i. Recalculate the height of the node, by reading the heights of the two children, choosing the maximum of those two heights, and adding `1`. Given a pointer to a node, you can access the node's children in constant time (`ptr->left` and `ptr->right`), and since the height is stored in the node itself, once you have a pointer to a node you can retrieve its height in constant time (`ptr->left->height`, for example). So reading the heights of the two children is constant time, and the rest is just arithmetic, which is also constant time.

ii. Recalculate the balance of the node – that, again, is just arithmetic on the heights of the child nodes, and we've already established that arithmetic on the heights of the child nodes can be done in constant time.

iii. If the balance is illegal, perform the appropriate rotation. Comparing the balance from (2) to +2 or -2 is constant, decding what rotation to perform will be constant (because you are just reading the heights of the children and grandchildren, all of which are reachable in constant time), and each rotation is a constant time operation, so no matter which one you do, rotation takes constant time.

All three of those steps are constant time, so we spend a total of constant time at this node. Since we have $\mathcal{O}(\lg n)$ levels to move upward through as we return from recursive calls, the total work will be the number of levels multiplied by the time spend on each level, which will be $\mathcal{O}(\lg n)$ times $\mathcal{O}(1)$, or $\mathcal{O}(\lg n)$.

4. **[List to tree – 15 points].**

   You have the following two standard node classes (which are publicly accessible and not encapsulated in another class):

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

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

   Write a function `LevelOrderToTree`. The function should take as parameter a pointer to a `ListNode`, which is the first element of a list that represents the level-order traversal of a *perfect* binary tree. This function should reproduce the binary tree from the level-order listing received as a parameter. That is, `LevelOrderToTree` should return a `TreeNode` pointer which will be the root of a perfect binary tree such that, if a level-order traversal is run on it, it will yield the same listing as the one received as parameter. If the parameter `ListNode` pointer is NULL, the the returned `TreeNode` pointer should also be NULL.

   You have one `Queue` available to you to use as a local variable, if you wish.

```
TreeNode* LevelOrderToTree(ListNode* head) {
   // your code goes here
```

   SOLUTION ON NEXT PAGE – essentially a modification of level-order traversal.

(List to tree, continued)

```
TreeNode* LevelOrderToTree(ListNode* head) {
   // your code goes here

   if (head == NULL)
      return NULL;
   else {
      // make root node and put it on queue
      Queue<TreeNode*> Q;
      TreeNode* root = new TreeNode();
      root->element = head->element;
      Q.Enqueue(root);
      TreeNode* tempInTree; // this will point to whatever node
                            // we are working with in tree

      ListNode* currentInList = head->next;  // since we've used
                                             // head value already

      while (!Q.IsEmpty()) {
         tempInTree = Q.Dequeue();
         if (currentInList == NULL) {  // no more children, this is a leaf
            tempInTree->left = NULL;
            tempInTree->right = NULL;
         }
         else {  // we have children to add to our dequeued node
            // add left child
            tempInTree->left = new TreeNode();
            tempInTree->left->element = currentInList->element;
            Q.Enqueue(tempInTree->left);
            currentInList = currentInList->next;

            // add right child
            tempInTree->right = new TreeNode();
            tempInTree->right->element = currentInList->element;
            Q.Enqueue(tempInTree->right);
            currentInList = currentInList->next;
         }
      }
   }
}
```

5. **[Counting Leaves – 15 points].**

   You have the following node class available to you, which is publicly accessible and not encapsulated in another class:

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

   Write a function `CountLeaves` that takes as a parameter, a pointer to a `TreeNode`, and returns the number of leaves in the tree whose root is that `TreeNode`. (Hint: Use recursion)

```
int CountLeaves(TreeNode* ptr) {
   // your code goes here

   if (ptr == NULL)
      return 0;
   else if ((ptr->left == NULL) && (ptr->right == NULL))
      return 1;
   else
      return CountLeaves(ptr->left) + CountLeaves(ptr->right);
}
```

(Counting Leaves, continued)

.

(scratch paper)