CS 14: Introduction to
Data Structures & Algorithms

May 22, 2003
Quiz 2, Form: A

First Name: ____________________

Last Name: ____________________

ID Number: ____________________

Signature: ____________________

This test contains two sections, a MULTIPLE-CHOICE section and a SHORT-ANSWER section. Please make sure to pace yourself properly so that you have time to work on both sections.

You may use ONLY the test's last sheet (front and back pages) for scratch work; I will not look at your scratch work and nothing there will be graded. Do NOT remove the scratch pages.

Good luck. :)

Part 1. MULTIPLE-CHOICE section. Please choose only one of the alternatives provided for each question and remember to mark your answers on the scantron form because ONLY THOSE will be considered. If scantron forms are not being used in this test, then make sure to CIRCLE your chosen answer.

Each question is worth 1 point and NO partial credit is given. No negative points are given for wrong answers. Please note that you will NOT get any credit for leaving questions unanswered.

1. In an arbitrary **Binary Search Tree**, the operation of **retrieving the minimum key** has a runtime efficiency which is

   (a) linear on the number of nodes of the tree.

   (b) linear on the height of the tree.

   (c) logarithmic on the number of nodes of the tree.

   (d) logarithmic on the height of the tree.

   (e) None of the above.

2. The **smallest** key stored in a **Binary Search Tree** is found on the **left-most** node of the tree.

   (a) Always.

   (b) Never.

   (c) Only if the tree is a **MinBST**.

   (d) Only if the tree is a **MaxBST**.

3. The **height** of a **Binary Search Tree** grows with the number of nodes in the tree. The growth is

   (a) Always linear.

   (b) Always logarithmic.

   (c) Linear only if the tree is balanced.

   (d) Logarithmic only if the tree is balanced.

   (e) None of the above.

4. Every **Binary Search Tree** is an **AVL** tree.

   (a) True.

   (b) False.

5. A **Binary Tree** is a tree where every node has two children.

   (a) True.

   (b) False.

6. In a **Minimum Binary Heap** with more than two nodes, the left-most node stores

   (a) the maximum key value.

   (b) the minimum key value.

   (c) Neither the maximum nor the minimum key value.

   (d) None of the above.

7. Building a **Minimum Binary Heap** from a sequence of n values

   (a) always takes $\mathcal{O}[n \log n]$ time.

   (b) always takes $\mathcal{O}[n]$ time.

   (c) can be done in $\mathcal{O}[n]$ time.

   (d) cannot be done in $\mathcal{O}[n]$ time.

   (e) cannot be done in $\mathcal{O}[n \log n]$ time.

8. In the array representation of a **Binary Heap** where the root element is stored at index 0, the **parent** node of the node stored at index i is found at index

   (a) $(i-1)/2$.

   (b) $(i+1)/2$.

   (c) $i/2$.

   (d) $2i+1$.

   (e) $2(i+1)$.

9. After an insertion into an **AVL** tree has rendered the tree unbalanced, the most important node to look for in order to re-balance the tree (the node I referred to as **X** in the lectures) is

   (a) the first node from the root to a leaf, along the path of insertion, which violates the balancing condition.

   (b) the first node from a leaf to the root, along the path of insertion, which violates the balancing condition.

   (c) the leaf on the path of insertion.

   (d) the first node from the root to a leaf, along the path of insertion, which does not violate the balancing condition.

   (e) the first node from a leaf to the root, along the path of insertion, which does not violate the balancing condition.

10. The **median** key stored in a **Binary Search Tree** is found on the **root** of the tree. [The median of a sequence of numbers is the number from the sequence which is larger than half the numbers in the sequence and smaller than the other half.]

    (a) Always.

    (b) Never.

    (c) Possible, but we cannot give a definite rule for when that happens.

    (d) Only for BSTs which are perfectly complete.

11. Every **Binary Search Tree** is a **Binary Heap**.

    (a) True.

    (b) False.

Part 2. SHORT-ANSWER section. You may use ONLY the test's last sheet (front and back pages) for scratch work, and your final answers to this section must be SHORT,

CLEAN, COHERENT, LEGIBLE, and written ONLY in the spaces provided, or your test will NOT be graded. NO exceptions will be made.

I will not look at your scratch work and nothing there will be graded. Do NOT remove the scratch pages.

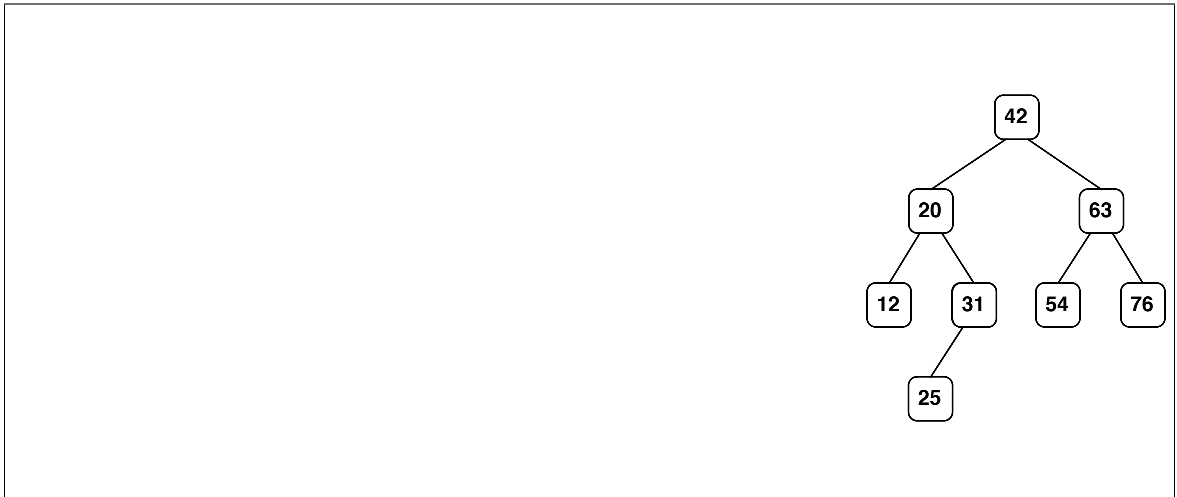
Unless indicated otherwise, each question is worth 1 point and NO partial credit is given. No negative points are given for wrong answers. Please note that you will NOT get any credit for leaving questions unanswered.

1. Define the **ordering** property of a **Binary Search Tree**. **Use ONLY the boxed space provided below (it may appear on the next page) and please write LEGIBLY**.

2. Define the **balancing condition** of an **AVL Tree**. **Use ONLY the boxed space provided below (it may appear on the next page) and please write LEGIBLY**.

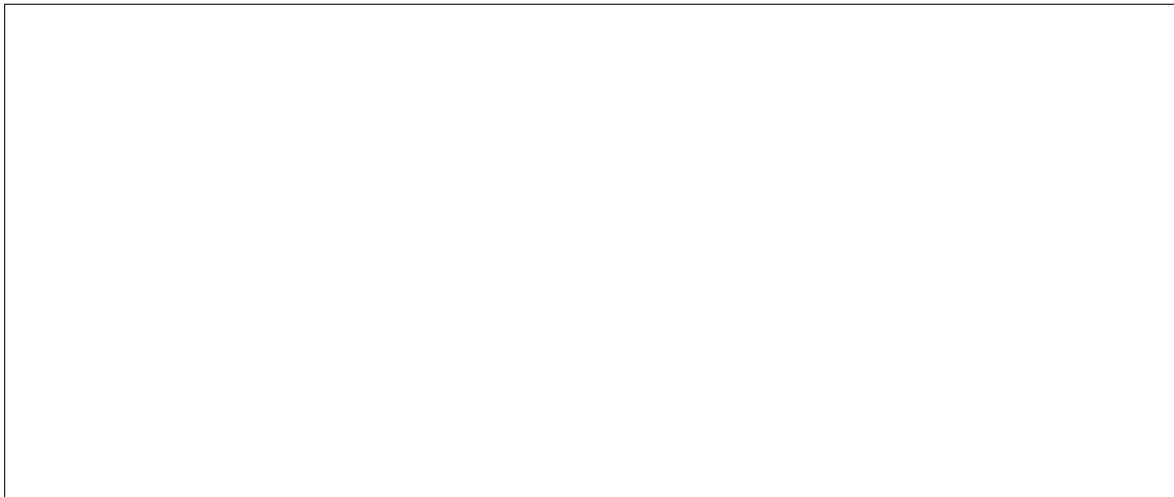
3. [2 points] Define both the **structural** and the **ordering** properties of a **Maximum Binary Heap**. If you need to make any assumptions of your own, make sure to specify them explicitly. Also, I do not want just drawings; I want an explanation in plain English. **Use ONLY the boxed space provided below (it may appear on the next page) and please write LEGIBLY**.

4. [2 points] Given the **Binary Search Tree** below, draw **both** possible results of deleting the node storing the value **20**. **Please use ONLY the boxed space provided below (it may appear on the next page), write LEGIBLY, and draw ONLY the FINAL results**. Do any scratch work on the scratch pages, **NOT** here!



5. [2 points] Given the **AVL Tree** below, draw the resulting **balanced** tree after the insertion of a node with the value **20**. **Please use ONLY the boxed space provided below (it may appear on the next page), write LEGIBLY, and draw ONLY the FINAL result**. Do any scratch work on the scratch pages, **NOT** here!

6. [2 points] Given the integers {9, 11, 12, 5, 6, 15, 16, 14, 10, 7}, draw the **Minimum Binary Heap** resulting from inserting those integers in the order they are given. **Please use ONLY the boxed space provided below (it may appear on the next page), write LEGIBLY, and draw ONLY the FINAL result**. Do any scratch work on the scratch pages, **NOT** here!



7. [3 points] Suppose that you're given two classes, BST and BSTNode, defining a **binary search tree** and its nodes, respectively, with UML diagrams appearing below. Write the C++ implementation of a **recursive** version of the function **void BST::prePrint(BSTNode * np)**, which prints the keys stored in the nodes of the sub-tree rooted at a given node, in **pre-order** fashion. **Use ONLY the boxed space provided below (it may appear on the next page) and please write LEGIBLY**. Do any scratch work on the scratch pages,

NOT here! *Pay attention to the UML diagrams!*

Answer Key for Exam A

Part 1. MULTIPLE-CHOICE section. Please choose only one of the alternatives provided for each question and remember to mark your answers on the scantron form because ONLY THOSE will be considered. If scantron forms are not being used in this test, then make sure to CIRCLE your chosen answer.

Each question is worth 1 point and NO partial credit is given. No negative points are given for wrong answers. Please note that you will NOT get any credit for leaving questions unanswered.

1. In an arbitrary **Binary Search Tree**, the operation of **retrieving the minimum key** has a runtime efficiency which is
   - (a) linear on the number of nodes of the tree.
   - **(b) linear on the height of the tree.**
   - (c) logarithmic on the number of nodes of the tree.
   - (d) logarithmic on the height of the tree.
   - (e) None of the above.
2. The **smallest** key stored in a **Binary Search Tree** is found on the **left-most** node of the tree.
   - **(a) Always.**
   - (b) Never.
   - (c) Only if the tree is a **MinBST**.
   - (d) Only if the tree is a **MaxBST**.
3. The **height** of a **Binary Search Tree** grows with the number of nodes in the tree. The growth is
   - (a) Always linear.
   - (b) Always logarithmic.
   - (c) Linear only if the tree is balanced.
   - **(d) Logarithmic only if the tree is balanced.**
   - (e) None of the above.

4. Every **Binary Search Tree** is an **AVL** tree.
   (a) True.
   (b) False.
5. A **Binary Tree** is a tree where every node has two children.
   (a) True.
   (b) False.
6. In a **Minimum Binary Heap** with more than two nodes, the left-most node stores
   (a) the maximum key value.
   (b) the minimum key value.
   (c) Neither the maximum nor the minimum key value.
   (d) None of the above.
7. Building a **Minimum Binary Heap** from a sequence of n values
   (a) always takes $\mathcal{O}[n \log n]$ time.
   (b) always takes $\mathcal{O}[n]$ time.
   (c) can be done in $\mathcal{O}[n]$ time.
   (d) cannot be done in $\mathcal{O}[n]$ time.
   (e) cannot be done in $\mathcal{O}[n \log n]$ time.
8. In the array representation of a **Binary Heap** where the root element is stored at index 0, the **parent** node of the node stored at index i is found at index
   (a) $(i-1)/2$.
   (b) $(i+1)/2$.
   (c) $i/2$.
   (d) $2i+1$.
   (e) $2(i+1)$.

9. After an insertion into an **AVL** tree has rendered the tree unbalanced, the most important node to look for in order to re-balance the tree (the node I referred to as **X** in the lectures) is

   (a) the first node from the root to a leaf, along the path of insertion, which violates the balancing condition.

   (b) the first node from a leaf to the root, along the path of insertion, which violates the balancing condition.

   (c) the leaf on the path of insertion.

   (d) the first node from the root to a leaf, along the path of insertion, which does not violate the balancing condition.

   (e) the first node from a leaf to the root, along the path of insertion, which does not violate the balancing condition.

10. The **median** key stored in a **Binary Search Tree** is found on the **root** of the tree. [The median of a sequence of numbers is the number from the sequence which is larger than half the numbers in the sequence and smaller than the other half.]

    (a) Always.

    (b) Never.

    (c) Possible, but we cannot give a definite rule for when that happens.

    (d) Only for BSTs which are perfectly complete.

11. Every **Binary Search Tree** is a **Binary Heap**.

    (a) True.

    (b) False.

Part 2. SHORT-ANSWER section. You may use ONLY the test's last sheet (front and back pages) for scratch work, and your final answers to this section must be SHORT, CLEAN, COHERENT, LEGIBLE, and written ONLY in the spaces provided, or your test will NOT be graded. NO exceptions will be made.

I will not look at your scratch work and nothing there will be graded. Do NOT remove the scratch pages.

Unless indicated otherwise, each question is worth 1 point and NO partial credit is given. No negative points are given for wrong answers. Please note that you will NOT get any credit for leaving questions unanswered.

1. Define the **ordering** property of a **Binary Search Tree**. **Use ONLY the boxed space provided below (it may appear on the next page) and please write LEGIBLY**.

   Answer: For *every* node N of the tree, the maximum key stored on the left subtree rooted at N is *less* than the key stored at N, and the key stored at N is *less* than the minimum key stored on the right subtree rooted at N. Another way to say the same thing is to say that *every* node to the left of N has a key smaller than the key stored at node N and *every* node to the right of node N has a key larger than the key stored at node N. In more formal terms, for *all* nodes N, it must be the case that

   $$\max\Big[\,\mathsf{Left}(N)\,\Big] < N < \min\Big[\,\mathsf{Right}(N)\,\Big]\,.$$

2. Define the **balancing condition** of an **AVL Tree**. **Use ONLY the boxed space provided below (it may appear on the next page) and please write LEGIBLY**.

   Answer: For *every* node N of the tree, the *difference* in the *heights* of the left and right subtrees of node N is *at most* 1, in magnitude. Another way to say the same thing: for *every* node N of the tree, the *difference* in the *heights* of the left and right subtrees N is either -1, 0, or +1. And, in more formal terms, for *every* node N, it must be the case that

   $$\Big|\,\mathsf{Height}[\,\mathsf{Left}\ (N)\,] - \mathsf{Height}[\,\mathsf{Right}(N)\,]\,\Big| \le 1\,.$$

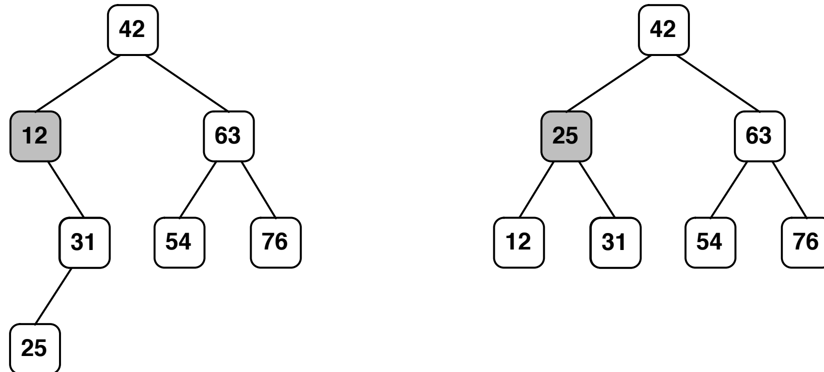
3. [2 points] Define both the **structural** and the **ordering** properties of a **Maximum Binary Heap**. If you need to make any assumptions of your own, make sure to specify them explicitly. Also, I do not want just drawings; I want an explanation in plain English. **Use ONLY the boxed space provided below (it may appear on the next page) and please write LEGIBLY**.

   Answer: The structural property of a Binary Heap (either Min or Max) is that the heap is a *complete* Binary Tree, that is, every level is filled with all possible nodes, except possibly for the last (deepest) level which, if not complete, must be filled from left to right without any gaps. The ordering property of a Maximum Binary Heap is that, for *every* node N of the Maximum heap, the key stored in the node itself is *larger* than or equal to both its children's keys. In a more formal way, for *every* node N, it must be the case that

   $$N \ge \mathsf{Left}\ (N) \quad \text{and} \quad N \ge \mathsf{Right}(N)\,.$$

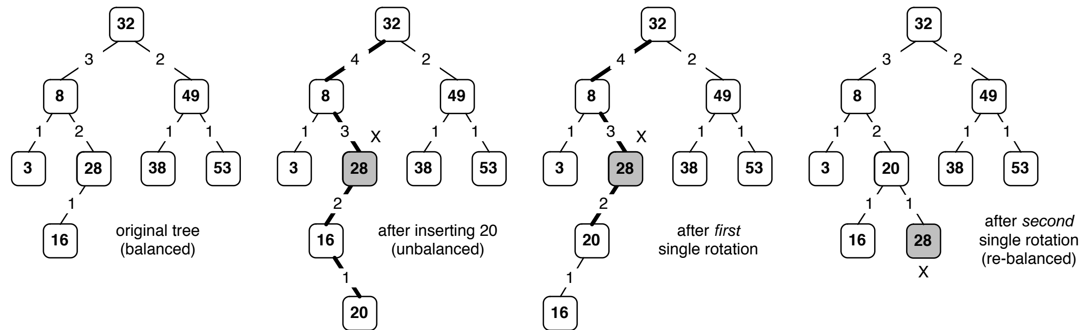
4. [2 points] Given the **Binary Search Tree** below, draw **both** possible results of deleting the node storing the value **20**. **Please use ONLY the boxed space provided below (it may appear on the next page), write LEGIBLY, and draw ONLY the FINAL results**. Do any scratch work on the scratch pages, **NOT** here!

Answer: Since the node containing the key 20 has both of its allowed children, deleting that node requires us to choose between (a) replacing it with the node containing the *largest* key, among all nodes from the *left* subtree of the node containing 20, and (b) replacing it with the node containing the *smallest* key, among all nodes from the *right* subtree of the node containing 20. Thus, the two possible results are those shown below:



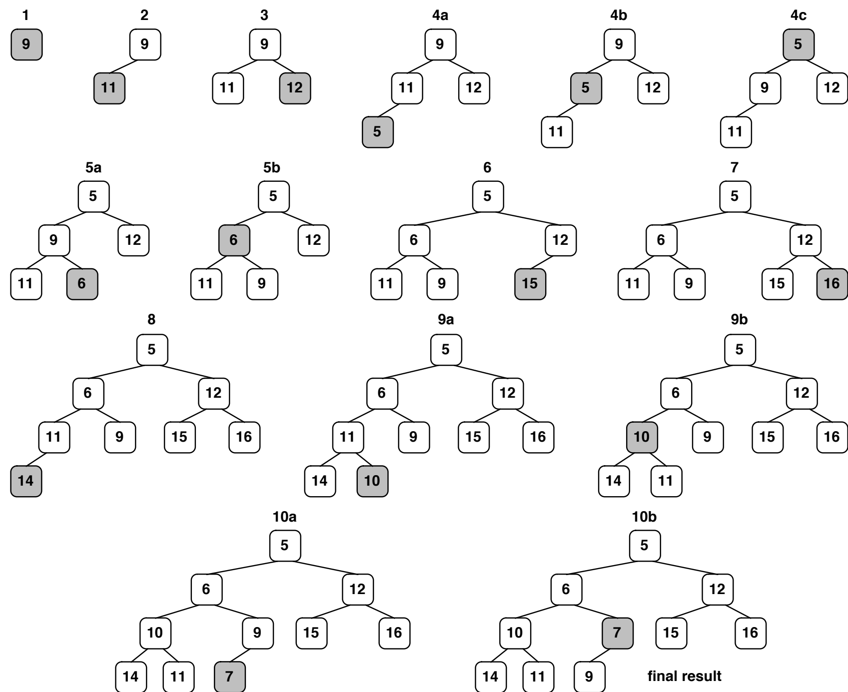
5. [2 points] Given the **AVL Tree** below, draw the resulting **balanced** tree after the insertion of a node with the value **20**. **Please use ONLY the boxed space provided below (it may appear on the next page), write LEGIBLY, and draw ONLY the FINAL result**. Do any scratch work on the scratch pages, **NOT** here!

Answer: The node containing the value 20 would have to be inserted as the right child of the node containing the value 16 but that would break the balance of the tree, according to the AVL condition (note that the tree was already AVL-balanced before the insertion). The path of insertion is {20, 16, 28, 8, 32}, from the inserted node up to the root. The *first* node in violation of the balancing condition, *up* along the path of insertion towards the root, is the node storing the key 28. That's what I called X. We know that balancing is restored by performing either a single rotation or a double rotation. Which one is it, though? To find out, we need to discover which of the four possible situations we're dealing with: LL, RR, LR, or RL. Since the insertion happened at the *right* subtree of the *left* child of X, this is an LR case. Now, recall that LL and RR cases require a single rotation, whereas LR and RL require a double rotation, so a double rotation is what we need. How do you do a double rotation? First, we do a single rotation between X's *child* along the path of insertion and X's *grandchild* along the path of insertion, and then we do another single rotation between X and *its new child*. Of course, I only wanted you to draw the **final** result but, since theses are supposed to be complete solutions, below you'll find the steps I outlined above.



6. [2 points] Given the integers {9, 11, 12, 5, 6, 15, 16, 14, 10, 7}, draw the **Minimum Binary Heap** resulting from inserting those integers in the order they are given. **Please use ONLY the boxed space provided below (it may appear on the next page), write LEGIBLY, and draw ONLY the FINAL result**. Do any scratch work on the scratch pages, **NOT** here!

   Answer: Once again, even though I only asked you for the final result, I'm going to show you all intermediate steps, including those needed to restore the heap order for those insertions that violate it.



7. [3 points] Suppose that you're given two classes, BST and BSTNode, defining a **binary search tree** and its nodes, respectively, with UML diagrams appearing below. Write the C++ implementation of a **recursive** version of the function **void BST::prePrint(BSTNode * np)**, which prints the keys stored in the nodes of the sub-tree rooted at a given node, in **pre-order** fashion. **Use ONLY the boxed space provided below (it may appear on the next page) and please write LEGIBLY**. Do any scratch work on the scratch pages, **NOT** here! *Pay attention to the UML diagrams!*

Answer: Recall that in a pre-order traversal of a binary tree, a node is visited before its left child and the left child is visited before the right child. An in-order traversal visits the left child of a node, then the node itself, then the right child. And, finally, in post-order traversal, the left child is visited first, then the right child is visited next, then finally their parent node is visited.

```
void BST::prePrint(BSTNode * np)
{
    if (np == NULL)
    { return; }

    // print the parent node first
    np -> printKey();

    // then the left child
    prePrint(np -> getLeft());

    // then the right child
    prePrint(np -> getRight());
}
```