✓ **Congratulations! You passed!**

TO PASS 80% or higher

Keep Learning

GRADE
100%

Week 3 Quiz

LATEST SUBMISSION GRADE

100%

1. **Create a binary search tree by inserting the following five values one at a time:**

1 / 1 point

4 6 5 7 8

What is the height of this tree?

(Recall how we calculate the height of a tree or subtree: The height of a leaf node by itself is 0. The height of a non-existent tree is -1. Otherwise, the height of a tree is the longest path length from the root of that tree to any one of its leaves.)

3

✓ Correct

The tree **4 6 5 7 8** has a height of three (e.g. the lineage **4 6 7 8** has three edges between its top node **4** and its bottom node **8**).

2. **Create a binary search tree by inserting the following eight values one at a time:**

1 / 1 point

3 1 2 4 6 5 7 8

What is the balance factor of the root node of this tree? (For this question, do not perform any rotations on this tree as you insert the items. It's just a binary search tree, not necessarily a balanced BST such as an AVL tree.)

2

✓ Correct

The root node is **3**. Its left subtree consists of **1 2** and so has a height of one. The root's right subtree consists of **4 6 5 7 8**, where the longest path is **4 6 7 8**, which has a height of three. The balance factor of the root is the height of its right subtree (3) minus the height of the left subtree (1), in this case resulting in a balance factor of 2.

3. **For the binary search tree created by inserting these items in this order: 4 3 5 1 2, which node among 1 through 5 is the deepest node with a balance factor of magnitude two or greater? (For this question, do not perform any balancing rotations as you insert these items.)**

1 / 1 point

3

✓ Correct

Looking at the subtree rooted at node **3**, there is no right child, and so the height of its non-existent right subtree is -1. (Leaf nodes are subtrees of height 0, so non-existent subtrees have height -1 for consistency). The left subtree consists of node **1** and its right child **2**. That is a subtree of height 1.

Therefore, the balance factor of node 3 is (height of right subtree) - (height of left subtree) = (-1) - (1) = -2.

Nodes 2 and 5 are leaf nodes, so their balance factors are (-1) - (-1) = 0. Node 1 has a balance factor of (0) - (-1) = 1.

The root node 4 has a balance factor of (0) - (2) = -2, which is the same balance factor as node 3, but node 3 is deeper in the tree, so node 3 is the correct answer: It is the deepest node whose height balance factor has magnitude of two or greater.

4. **Consider the binary search tree that you constructed in the previous question. If we interpret it now as an AVL tree, it has an imbalance that can be fixed with a rotation. (Remember that we focus on the deepest point of imbalance, where the magnitude of the balance factor is 2 or greater, to perform the rotation.)**

1 / 1 point

After performing the correct balancing rotation about the node that we identified in the previous question, the resulting tree is identical to which one of the following binary search trees? (We'll describe these other trees by listing the order in which you would insert items to create the trees directly.)

○ Inserting **2 1 4 3 5** one node at a time.

- [✓ Inserting **4 2 5 1 3** one node at a time.]
- Inserting **3 5 2 4 1** one node at a time. FOO
- Inserting **3 2 4 1 5** one node at a time.

✓ **Correct**
This tree is indeed better balanced than before, and it's the correct result of performing a left-right rotation about the node 3 in the previous tree. In this new tree, the balance factor of greatest magnitude anywhere is now found at the root node, which has balance factor (0) - (1) = -1.

5. **The code that ensures the balance of an AVL tree after node insertion or removal only checks if the height balance factor is +2 or -2. What happens if the height balance factor of a node in an AVL tree after node insertion or removal is greater that +2 or less than -2?** 1 / 1 point

- [✓ An AVL tree never has a node with a height balance factor greater than +2 or less than -2, even after a node insertion or removal.]
- We ignore nodes in an AVL tree with height balance factor greater than +2 or less than -2 because they are statistically rare and are unstable, such that they are removed as soon as any tree balancing rotation occurs.
- There is additional code not shown that handles the cases when the height balance factor is greater than +2 or less than -2.
- When insertion and removal create a node whose height balance factor is greater than +2 or less than -2, that node always has a descendant with a height balance factor equal to +2 or -2 and when all of its descendant nodes are resolved, then its height balance factor will be no greater than +2 or no less than -2.

✓ **Correct**
Every node in an AVL tree has a height balance factor of -1, 0 or 1. Inserting a node can increase the height of a subtree by only 1, and so can change the height balance factor of any node to no more than 2 or no less than -2. Similarly, a node in a any binary search tree is removed by either by deleting a leaf node, shortening a chain of nodes by removing a node with a single child, or replacing a node with its immediate ordered predecessor, and all three of these operations change the height of a subtree by no greater than +1 or no less than -1.

6. **If, after inserting a new node into an AVL tree, you now have a node with a height balance factor of -2 with a child with a height balance factor of -1, which of the following operations should be performed?** 1 / 1 point

- [✓ Right Rotation]
- Right-Left Rotation
- Left-Right Rotation
- Left Rotation

✓ **Correct**
Since the parent and child height balance share the same sign, that means that there is a "stick" that needs to be rotated to form a "mountain." Since the shared sign of the height balance factors is negative, this means the left subtree is the cause of the imbalance and should be resolved by a *right* rotation.

7. **If, after inserting a new node into an AVL tree, you now have a node with a height balance factor of -2 with a child with a height balance factor of +1, which of the following operations should be performed?** 1 / 1 point

- [✓ Left-Right Rotation]
- Right Rotation
- Left Rotation
- Right-Left Rotation

✓ **Correct**
Since the height balance factors of the parent and child have different signs, this is an "elbow." Since the child's height balance factor is +1, we first perform a left rotation on it to turn it into a stick. Then since the parent's height balance factor is -2, we raise the middle of this "stick" to create a "mountain" by performing a right rotation on it.

8. **Which one of the following is NOT a valid reason to choose the B-Tree representation over a standard AVL binary search tree?** 1 / 1 point

- B-Trees require fewer block read accesses for tree operations.
- B-Trees work faster in networked cloud environments than do AVL trees.
- [✓ B-Trees have better algorithmic "Big-O" run-time complexity for the find operation.]
- B-Trees run faster on large data sets than do AVL trees.

✓ **Correct**

While the B-Tree find operation runs in O(log_m n) time, the m ends up being a constant factor and O(log_m n) = O(lg n) as a consequence of the big-O characterization of how run time increases as the number of data items (n) increases.

9. **Which of the following statements is NOT true for a B-Tree of order m?** 1 / 1 point

- ○ Each node can have at most one more child than key.
- ○ All leaf nodes are at the same level of the B-Tree.
- [✓ ● Each node can hold an ordered list of as many as m keys.]
- ○ Any node that is not the root or a leaf holds at least half of the total number of keys allowed in a node.

✓ **Correct**

This statement is indeed false. In an order-m B-tree, each tree node indeed holds multiple keys, but the number of keys is limited to m-1.

10. **If a B-Tree is completely filled, meaning every node holds its maximum number of keys and all non-leaf nodes has the maximum number of children, then what happens when an additional key is inserted into the B-Tree?** 1 / 1 point

- ○ A new node containing the new key is added above the previous root and becomes the new root. The new root will have one pointer leading to the old root node.
- ○ A new leaf node is simply added to the B-Tree.
- ○ Every leaf node in the entire B-Tree becomes parent to a new leaf node, but all but one of these leaf nodes are "blank" placeholder nodes that contain zero key values.
- [✓ ● After searching for the leaf node where the new key should go, the leaf is split in half as two separate leaf nodes, and then the middle value is thrown up to the layer above as an inserted key, and this insertion and rebalancing repeats until a new root key rises to the top, which adds a layer to the tree.]

✓ **Correct**

Indeed this must happen to maintain the properties of the B-tree.