**Aleksandr Itskovich (ai1221)**

**Collaborator:** Elsie Kenyon

**Assignment 2**

**Question 1: Binary Search Trees**

a)

* Insert: 30, 15, 40, 10, 17, 50, 8



* Explain why this tree is an AVL tree.

An AVL tree is characterized by the following invariant: that the difference in heights between the subtrees rooted at any node is at most 1.

We see that this invariant is met in the tree drawn above. Take, for example, the root node at 30. The height of its left subtree, LH, (rooted at 15) is 2, since the longest path from 15 to a leaf has length 2. The height of the right subtree, meanwhile is 1. Repeating the same analysis for each node yields:

|  |  |  |  |
| --- | --- | --- | --- |
| **Node** | **Left Subtree Height** | **Right Subtree Height** | **Difference** |
| 30 | 2 | 1 | 1 |
| 15 | 1 | 0 | 1 |
| 40 | -1 | 0 | 1 |
| 10 | 0 | -1 | 1 |
| 17 | -1 | -1 | 0 |
| 50 | -1 | -1 | 0 |
| 8 | -1 | -1 | 0 |

We can see that the AVL height invariant is maintained at every node in the binary tree, therefore the tree is an AVL tree.

* Next, insert the key 9. Explain why the result of the insert no longer satisfies the AVL property.

Diagram

Description automatically generated

We appended an additional node to the left subtree of 30. Now we see that for node 30, the height of its left subtree rooted at 15 is 3, while the height of the right subtree remains unchanged at 1. Because the difference in height between the two subtrees is greater than 1. Therefore the tree is no longer an AVL tree.

* Show how you could perform at most 2 rotations in order to restore the AVL tree property.

We could rotate twice to restore the tree heights back to having a difference of at most 1.

Diagram, schematic

Description automatically generatedCalendar

Description automatically generated

In general, when an AVL tree is out of balance there are 4 types of rotations that are similar to the ones discussed in the RB Tree case: two bent rotations (handling left-right and right-left cases) and two straight rotations (handling left-left and right-right cases). After performing either straight rotation, the tree is balanced. Moreover, we can always convert a bent case into a straight case by performing one rotation. That means we are never more than 2 rotations away from having a balanced AVL tree.

We either have

1. Bent Case rotation, followed by a straight case rotation (2 rotations)
2. Or just a straight case (1 rotation)

These cases are summarized below:

A picture containing text, flying, map, group

Description automatically generated

b) Suppose T1 and T2 and T3 each reference the root nodes of a binary search tree on n

nodes. We wish to combine these three trees into one binary search tree. Explain how this can be done in time, in such a way that the resulting BST has height . You do not need to write pseudo-code for your procedure, but you must properly describe your steps.

We are given that we have 3 nodes that point at the roots of different binary search trees. Let’s assume that the sizes of the trees are n1, n2 and n3, respectively.

We can carry out the following procedure to build a minimal binary search tree in linear time.

1. Perform an in-order traversal of each tree, storing the values in an array as they are encountered. This produces 3 separate, *sorted* arrays, with sizes n1, n2 and n3, respectively.
   1. We know from class that in-order traversal is linear for each tree.
   2. That means the total work done for all 3 trees is
2. Merge the 3 sorted arrays produced in step one into a single array. The logic behind this step is identical to the way arrays are merged in merge sort, only now we have a 3-way merge instead of a 2 way merge. This produces a single, *sorted* array.
   1. Overall, we have to combine elements, so the total time is again .
3. Now that we have a single, sorted array, we know from class that we can produce a minimal BST from a sorted array by recursively selecting the median element from the left and right subarrays, and inserting that median element into the tree.
   1. Getting the median element recursively is given by , which we know to have a solution of by the Master Method.
   2. Inserting into the tree takes constant time per element because we don’t have to traverse the tree searching for the position of the newly inserted element; we know exactly where to insert elements since we’re selecting medians from subsequent sub-arrays, so in this case insertion is constant is constant. ]
4. Overall, we have 3 steps, each of which is shown to take time. That means the overall procedure also runs in time.

c)

Question 2: Red-Black Trees

a)

* For height b, what is the maximum number of nodes in the tree (excluding NIL nodes)? Describe the shape of the tree and its coloring.

To achieve the maximum number of nodes in a RB tree, we will want to have a full tree (that is, all possible nodes occupied) with the maximum height. We know from class that to achieve maximum height in a RB tree, we will want nodes levels to be alternating between red and black.

For a given black height, b, we know that for every black level, we are going to have an additional red level (because we want the tree to be as full as possible for a given black height, with alternation). That means that the number of levels is going to be given by 2b.

The number of nodes in each level is given by the level number raised to the 2. That means, the number of nodes in the first level (level is) is 20 = 1, the number of nodes in the second level is given by 21 = 2, etc.

Adding up all the nodes in all the levels gives us: