**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)

LazyDelete(T, z):

while T != NIL

if T.deleted = false and T.key = z.key:

T.deleted = true

else

if T.key < z

T = T.right

else

T = T.left

* TreeSearch

We would have to update the TreeSearch method only slightly: if we find a node that has a key equal to the target search key, we also need to check if the node was deleted before returning the match; in other words, we just want to make sure we are only matching on non-deleted nodes. Otherwise the algorithm is the same. This change is shown below:

TreeSearch(T, z):

while T != NIL

**if T.deleted = false** and z = T.key

return T

else

if T.key < z

T = T.right

else

T = T.left

* TreeInsert

The TreeInsert algorithm doesn’t change. When inserting a child, we don’t care whether or not the parent is deleted or not, we just want the tree to maintain the necessary invariant for non-deleted nodes, which the algorithm in its current form already does. This also takes for granted that nodes being inserted are marked as non-deleted by default.

* Disadvantages:

Implementing lazy deletion makes the implementation of other methods more complicated. More importantly, it increases the overall space-complexity of the tree. Since deleted nodes are never removed from the tree, they continue to take up space. For example, if we delete half the nodes in a tree containing n nodes, we are effectively wasting half of the nodes. It also means that space only increases, never shrinks, since nodes are never truly removed. All of this also means that the time required to carry out an operation such as search or insert also increases by some constant factor as the number of deleted nodes increases.

**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. That way we have half black nodes and half red nodes:

Shape

Description automatically generated with medium confidence

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.

We need to total sum of all internal nodes. Adding up all the nodes in all the levels gives us:

However, because we don’t want to count the last level of nil nodes (and only the internal nodes) we want to be just .

Therefore the total number of internal nodes in the maximum case is given by.

* Repeat for minimum number of nodes:

To get the minimum number of nodes, we want to get the smallest possible BT tree. This corresponds to the case when all leaves are black.

The total number of levels of this tree will be just b. In this base, we have

As above, we don’t want to count the nil nodes so becomes just .

Therefore the total number of internal nodes in the minimum case is given by.

b)

We can only increase the black height when we insert a node that causes a recoloring. Rotations never increase the black height.

c) It is impossible to transform this tree into a proper red-black tree.

Consider the right subtree first. The right height is 2. We can either have all nodes be black, or alternate red and black nodes. Therefore the right subtree has . More specifically, it can either be 2 or 3:

Chart, line chart

Description automatically generated

Now, we can look at the right subtree.

In the left subtree we have multiple paths down to NIL nodes. These paths have sizes 3, 4, 5 and 6 (including the NIL nodes). The shortest path has height 3, and the longest has height 6.

Let’s examine the left subtree with respect to the right subtree.

* Case 1: the right subtree has BH 2:

If the right subtree has a BH of 2, then it is impossible for the left subtree to maintain a BH of 2 along its longest path of length 6. Because we cannot have adjacent red nodes, they must be alternated with black nodes. The maximum number of red nodes we could have along that height would be 3, which means we would need to have *at least* 3 black nodes (including NIL). That means it is impossible for the right subtree to have a BH of 2 and still create a valid RB tree with the left subtree.

* Case 2: the right subtree has BH 3:

If the right subtree has a BH of 3, the only way to maintain a BH of 3 along its shortest path of length 3 is to have all nodes along that path to be black. This is depicted in the diagram below:

Diagram

Description automatically generated

There is no other way to achieve a path of 3 along nodes **a** and **b**, and the NIL node on **b**.

However, nodes **a** and **b** also form the backbone of paths **root-a-b-c** (length 4) and **root-a-b-c-d** (length 5).

Because node **c** has a NIL node, it cannot be colored black. If we made it black, we would have a BH of 4 along nodes **a-b-c-NIL**. This violates the RB tree invariant since we know the right subtree has a BH of 3.

Therefore, we have to color node **c** red. However, when node **c** is red, node **d** must then be black because we cannot have two adjacent red nodes. Now, however, the path along **a-b-c-d-NIL** has a BH of 4, violating the RB tree invariant.

There is therefore no way to make this tree a valid red-black tree, according to the above analysis.

Question 3: Augmented BSTs

a)

**b)** Suppose we augment a binary search tree so that each node has an additional attribute

called *x.leaves* which represents the number of leaves in the subtree rooted at x. Let T be the root of such a BST. Your job is to update the BST-Delete(T, z) algorithm from class so that it correctly deletes node z from the tree T and updates the values of *x.leaves* for all necessary nodes in the tree. You must provide the pseudo-code for the updated version and explain why the runtime is still *O(h)*.

**BST-Delete(T, z)**

if z = T and z.left = NIL

return z.right

else if z = T and z.right = NIL

return z.left

else

if z.left = NIL and z.right = NIL

if z = z.parent.left

z.parent.left = NIL

else

z.parent.right = NIL

z.parent.leaves = z.parent.leaves - 1;

if z.parent.right != NIL and z.parent.left != NIL

y = z.parent;

while y != NIL

y = y.parent

y.leaves = y.leaves – 1

else if z.left = NIL

Replace(z, z.right)

else if z.right = NIL

Replace(z, z.left)

else

y = Find-Min(z.right)

z.key = y.key

if y.right = NIL # y is a leaf

y.parent.leaves = y.parent.leaves - 1

Replace(y, y.right)

return T

**c)** (8 points) In our practice problems we defined AVL trees. Each node of the AVL tree is augmented with an attribute called the *x.balance-factor* defined as follows: if both subtrees of x have the same height, the balance-factor is 0. Otherwise the balance-factor is 1 (left side is higher) or 􀀀1 (right side is higher). Given a tree T that is augmented with this information, update the Tree-Insert(T; z) algorithm from class so that the balance-factors are correctly updated after the insertion. If the resulting tree is not a proper AVL tree after the insertion, you do not need to repair it.

*\* The height of an empty tree is defined as -1*