CSE 222 HOMEWORK #10

Construction of AVL Trees with various properties

Orhan Aksoy

09104302

1. Problem

An AVL Tree is a height-balanced tree where every node in the tree is height-balanced – that is, the heights of every node's left and right sub trees differ by at most one. However, a number of items can be stored in an AVL Tree in more than one way. A methodology is to be developed in order to create an AVL Tree filled with N items (that are known beforehand),with the following properties:

a. An AVL Tree with the maximum height of log(N)
b. An AVL Tree with the maximum height
c. An AVL Tree that is a complete tree.

The methodology will be applied into the textbook AVL Tree implementation with three constructors, each of which will implement one of the trees described above.

2. Analysis

The height of an AVL Tree satisfies the following inequality.[1]

$$\log(N+1) \le h \le 1.44\log(N+2) - 1.32$$

This means that the smallest height of an AVL tree with N nodes is log(N+1), which can only happen if the AVL Tree is full.

The biggest height of an AVL Tree with N nodes is calculated by first deriving the minimum number of nodes required to create an AVL Tree with height h. This tree can be considered the ‘worst case AVL Tree’. The two sub trees of a worst-case AVL Tree of height h are also worst-case AVL Trees. The first child is a worst case AVL Tree of height (h-1), and the other (h-2).[2]. For a given height of h, the minimum number of nodes required is:

n = Fib(h+2)-1

where Fib(i) is the i’th Fibonacci number. Using the analytical evaluation techniques of Fibonacci numbers, the maximum height of an AVL Tree with N nodes is calculated as 1.44log(N+2) – 1.32.

Using this information, the methodology required to build the trees described in section 1 are:

a. An AVL Tree with the maximum height of log(N):

The minimum height of an AVL Tree is log(N+1) which is very close to log(N). For any tree to have this height, it needs to be full; that is, the balance of all nodes should be exactly ‘0’. This can only happen for specific values of N (1, 3, 7, .., 2^h-1), so, in order to satisfy the requirement for any number of inputs, the best possible tree should be generated, where the level difference of the deepest node and any other node that doesn’t have two children is at most 1. In other words, either the tree is full, or the tree can be made full by removing all of the nodes from the deepest level.

Such a tree can be created by creating a single-item AVL Tree with the smallest input, and then adding the inputs in increasing order.

Proof:

Every added item will be bigger than all of the items in the tree, so will be placed on the right-most side of the tree. So, the balance condition that will trigger the rotations will always be ‘right-right’ case.

[1] http://www.eli.sdsu.edu/courses/fall96/cs660/notes/avl/avl.html
[2] http://pages.cs.wisc.edu/~siff/CS367/Notes/AVLs/

This means that all of the inputs on the left side of the root will always get there after a left rotation of the root.

For the root to have a left rotation after an addition,

a. *The balance of the root should be +2*
b. *The balance of all the right child, right child of the right child, etc. of the root should be exactly +1 other than the added item. (If they were +2, they would have been rotated instead of the root). So*
   I. *If the right child has no left child, then it can have at most one right child. Then, the tree rotation will be as shown below:*

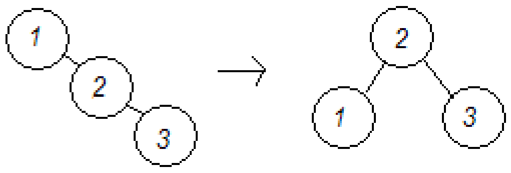

Figure 1 AVL Rotation after 4th insertion

   II. *If the right child has a left sub-tree, then the height of its right sub-tree is one bigger than its left sub-tree. Then, the tree rotation will be as shown below:*

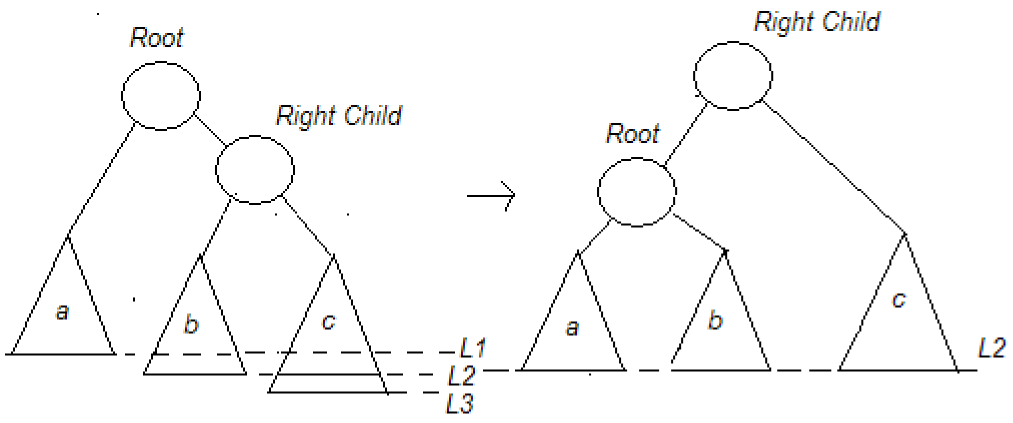

Figure 2 Rotation case of the root

Using (I) and (II), it is shown below inductively that all of the left sub-trees are full. This proves that, as long as the items are added in increasing order, all of the sub-trees will be full, so the level above the added item will always be full. So, the claim in (a) is valid.

Proof:

i. *For the left sub-tree of height 1, the root rotation will happen in the case below:*

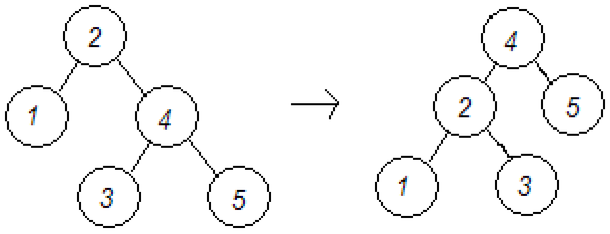

Figure 3 Rotation with 1 item on the left sub-tree. Base Case for Inductive Proof

After the rotation, the left sub-tree of the new root is full.

ii. *It is assumed that the left sub-tree of “20” after the rotation is a full tree (Refer to Figure 4). The root of the left sub tree is “10” and its left and right sub-trees are ‘a’ and ‘b’. As a result, the sub-trees ‘a’ and ‘b’ should also be full. The same condition holds for the case where the root around which the rotation happens was 20 and 40, too. So, if the left sub-tree of 20 is full, then all of the sub-trees, a, b, c and d are full.*

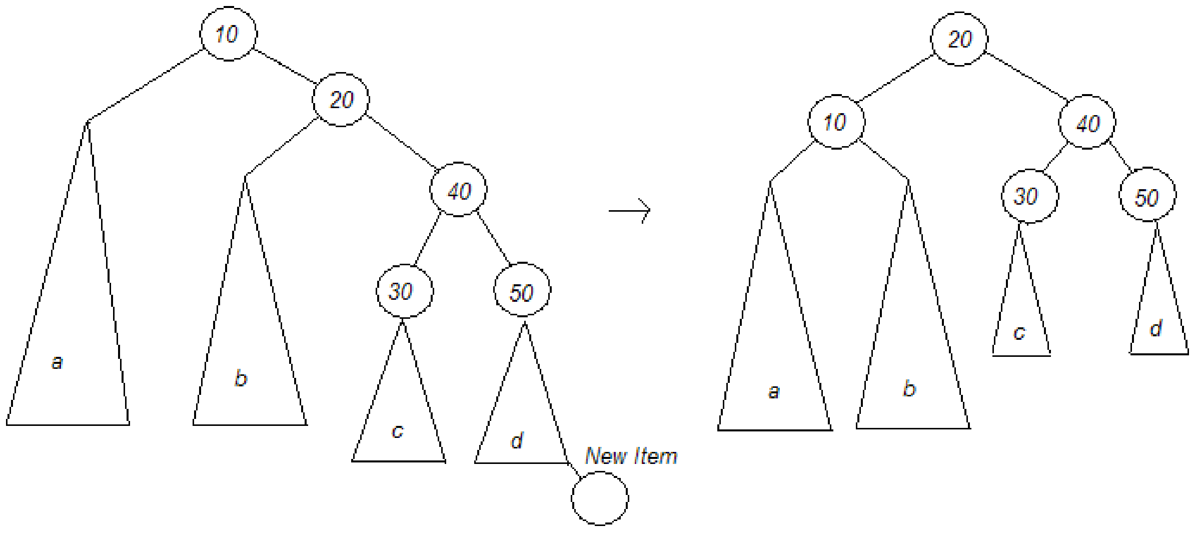

Figure 4 Rotation with left sub tree of height h - Inductive Step 2

iii. *The state where another left rotation is required on the root after a new addition is shown on* Figure 5. *In this tree, the left sub-tree of "40" (c') can only be formed after the sequence of events described above. This means that c' is also full. As a result, the left sub-tree of the root "40" has the root "20" whose both sub-trees are full, and thus, the left sub-tree of "40" is full.*

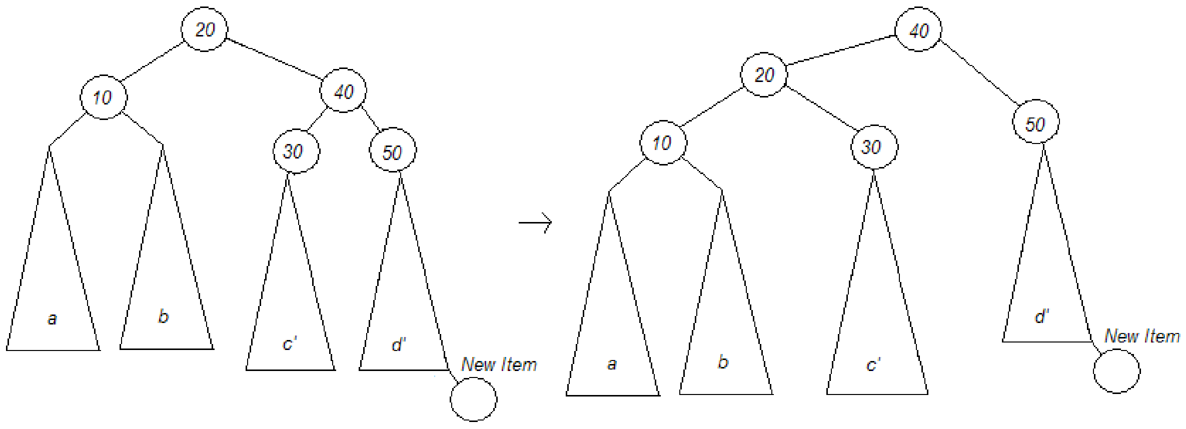

Figure 5 Rotation with left sub tree of height h+1 - Inductive Step 3

b. An AVL Tree with the maximum height

The maximum height of an AVL Tree with N nodes can be calculated by first deriving the minimum required number of nodes to create a tree with height h, which is Fib(h+2)-1.

In order to construct a tree with this properties with N values, first, the corresponding maximum h value is found where Fib(h+2)-1 is smaller than N. Then, this tree is extended by adding items to the nodes with the smallest height until the desired number of nodes is reached.

Then, the values are input from the sorted N values to the full tree with inorder traversal.

c. An AVL Tree that is a complete tree.

A complete tree with N nodes will be created, and then the values will be input from the sorted N values to the full tree with inorder traversal.

For this purpose, the number of nodes (m) is selected to create the maximum full tree with the available nodes ($m<=N$ and $m = 2^k-1$) , and then, the difference between this number and N is calculated to be the number of nodes in the deepest level.

In order to get these nodes to be located from left to right on the deepest level, they should be selected as 1st, 3rd, 5th, 7th, etc. items from the sorted list.

Proof:

At the deepest level, the left-most node is smaller than the rest of the tree, so it is the item #1 in the sorted list.

Any node which is a left child of its parent will be the previous item of its parent. If exists, the right child of its parent will be next item after its parent, so it will be next item after itself. So, the difference between two siblings is always 2.

For a right child at the deepest level,

I. *Its parent will be the previous one in the list.*
II. *Its parent will be the sibling of the parent of the next item at the same level, so the grandparent is common.*
III. *It's grandparent will come right after itself in the list, and similarly, right before the next item in the list. So the difference between them is always 2.*

As a result, starting from 1, the last level is always numbered 1, 1+2, 1+2+2, ….

Then, using m nodes, a full tree is created with value of '0' for all nodes, and the values are input from the sorted m values to the full tree with inorder traversal. Afterwards, the removed items are added to the AVL tree, and the binary search tree addition mechanism puts them to their proper locations in the last level.

3. Implementation

3.1. An AVL Tree with the maximum height of log(N)

First, the input array is sorted and then the items are inserted into the tree. The nodes with children count less than tree are either in the deepest level or one above as described in section 2. As a result, the searches in this tree is always log(N).

3.2. Worst case AVL Tree

First, the input array is sorted. Then, the worst case AVL tree is formed recursively, since the left sub-tree of a worst case tree with height h is another worst case tree with height (h-1), and the right sub tree with height (h-2). At every step during recursion, the balances of all of the nodes other than leaves are set to (-1) by first adding a '0' and a '-1'.

After the construction of the tree, and then the values of the items in the list are assigned to the nodes in the worst-case tree with in-order traversal.

3.3. An AVL Tree that is a complete tree

First, the input array is sorted and then the number of items in the deepest level are calculated. The 1st, 3rd, 5th, etc. items are then removed from the original list, leaving 2^m-1 items that form a complete tree.

Then, a balanced tree filled with ‘0’ is formed recursively and then the values of the items in the list are assigned to the nodes in the complete tree with in-order traversal. Then, the additional items are added to the tree with ‘add’ methods.

4. Testing

In order to test the application, the following sources are necessary:

The sources:

- Main.java, BinaryTree.java, SearchTree.java, BinarySearchTree.java, BinarySearchTreeWithRotate.java, AVLTree.java and ExtendedAVLTree.java.
-

Before testing, the sources should be compiled using

```
# javac Main.java
```

To run the application, use

```
# java Main <test type> <integer list>
```

Where
Test type is “-c” for complete tree, “-w” for worst case tree and “-l” for log(n) tre
Integer list is the list of integers to be stored in the tree.

Example:

```
# java Main –c 1 3 6 2
```

4.1. The Test Cases

4.1.1. Test Case 1 – The log(n) tree

The following runs are made and the result is observed. The only empty spots on the tree are on the deepest level.

```
#java Main –l 1 5 2
#java Main –l 10, 88, 33, 13, 46, 32
```

4.1.2. Test Case 2 – The worst-case tree

The following runs are made and the result is observed. The balance values of all nodes other than leaves should be -1.

```
#java Main –w 1 5 2
#java Main –w 10, 88, 33, 13, 46, 32
```

4.1.3. Test Case 3 – The complete tree

The following runs are made and the result is observed.

```
#java Main –c 7 3 5
#java Main –c 7 3 5 9
#java Main –c 7 3 5 9 11
#java Main –c 7 3 5 9 11 10
#java Main –c 7 3 5 9 11  10
```