AVL Trees and Red-Black Trees

University of Malta

Faculty of Information & Communication Technology

Bachelor of Science in Information Technology (Honours) (Artificial Intelligence) 2nd Year

ICS2210 – Data Structures and Algorithms 2

Dr Kristian Guillaumier & Dr John M. Abela

Francesca Maria Mizzi

118201L

Table of Contents

AVL Trees

Introduction

AVL Trees were first created by Adelson-Velsky and Landis in 1962. AVL trees are a special type of self-balancing binary tree where the height of the subtrees of a nodes can only differ by ± 1. [5]

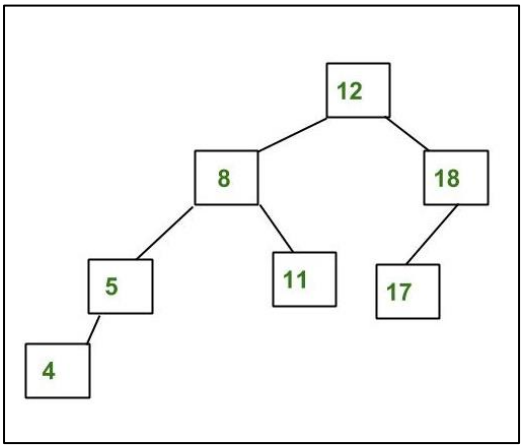

Figure 1 – AVL Tree

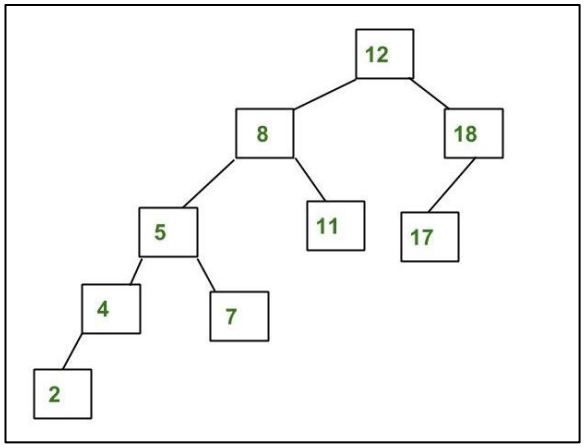

Figure 2 - Not an AVL Tree

In the above two diagrams we can see a clear explanation of what is meant by an AVL tree. In figure 1, we can see that the node with the number 4 in the left subtree is only one node below the lowest node in the right subtree, the node with the number 17. Figure 2, however, is not an AVL tree since the lowest leaf node, the node with the number 2, is two nodes below the lowest leaf node in the right subtree, the node with the number 17.

When inserting or deleting a node, it is important to make sure that the tree is still balanced in order to maintain the rule where the lowest leaf node in a subtree can only be ±1 node away from the complimentary subtree. This is done through a series or left and right rotations in order to bring back the balance.

Balanced vs. Unbalanced AVL Tree

While all AVL trees follow the node rule defined earlier, there are two types of AVL trees: balanced and unbalanced.

A balanced AVL Tree is where a node either has no sons (leaf node) or 2 sons. This means that no need has only 1 child, keeping the tree perfectly balanced as seen in figure 3.

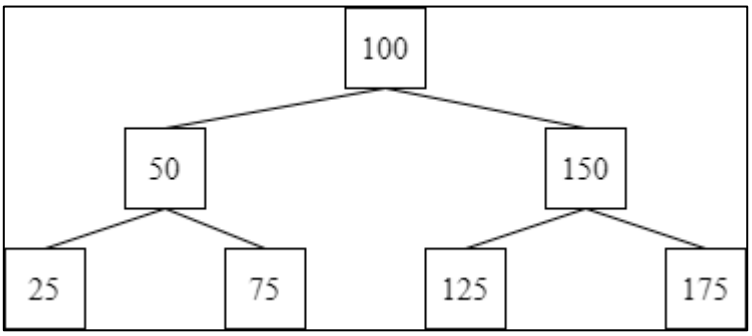

Figure 3 - Balanced AVL Tree

While an unbalanced AVL tree sounds contradictory, it merely refers to a type of AVL Tree where a node is allowed to have only child node as seen in figure 4.

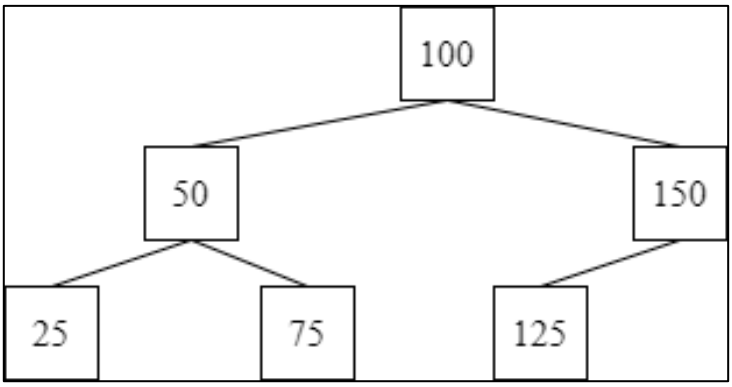

Figure 4 - Unbalanced AVL Tree

Balance Factor

The balance factor of a node in an AVL tree is the difference in height between it's left subtree and right subtree. Therefore

$$Balance\ Factor = Height\ of\ Left\ Subtree - Height\ of\ Right\ Subtree$$

The balance factor is what is used to determine whether or not the binary tree is an AVL tree, Therefore, for a tree to be AVL, the balance factor must be ±1.

Implementation

The implementation of an AVL requires 3 important aspects: insertion, deletion and rotation. All three of these aspects involve the manipulation of the binary tree in order to insert, remove or move a node. While insertion and deletion are relatively individual, rotation ties into both because when a node is inserted or deleted, the balance factor must be maintained and if it is not, the tree must be rotated.

Insertion

Once a node is inserted into the AVL tree following standard binary tree rules,

$$node(left) < node(root) < node(right)$$

, it must be checked if the balance factor is maintained. If it not found to be ±1, the tree must be rotated in order to rebalance the tree. There are two basic operations which can be carried out to rebalance the tree:

1. Left Rotation
2. Right Rotation

Pseudocode

1. Insert the new node using standard binary tree insertion rules
2. Update balance factor of the nodes based off of height of left and right subtrees
3. If the balance factor of a node is >1 or <-1, the node must be rotated in order to re-balance the tree
    a. If the balance factor > 1, the height of the left subtree is greater than that of the right subtree. So, do a left-right rotation
    b. If the balance factor < -1, the height of the right subtree is greater than that of the left subtree. So, do a right-left rotation [4]

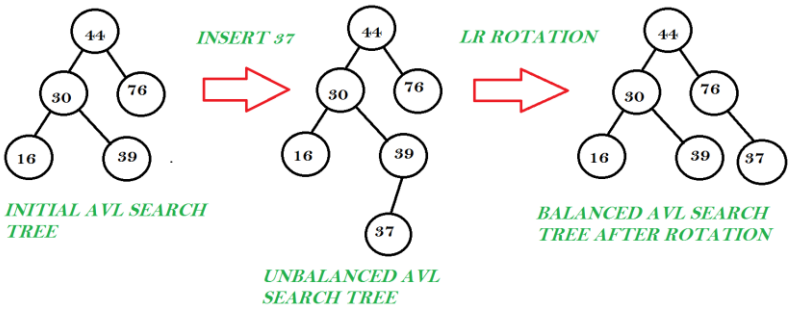

Figure 5 - Left-right rotation of an AVL Tree

Deletion

When deleting a node from an AVL tree, there are 3 cases to consider:

1. The node is a leaf node
2. The node to be deleted has one child
3. The node to be deleted has two children

The case of the node being a leaf node, the deletion is very simple as the node can just be removed.

In the case of the node having one child, the contents of the child node are swapped with the node which needs to be deleted. Therefore, the node to be deleted is a leaf node and can be removed.

When the node to be deleted has to children, the larger successor child is identified and the contents of the two nodes are swapped and the node to be deleted can be removed.

The balance factor of all the nodes must then be checked and if the tree is unbalanced, it must be rotated as follows:

1. If the balance factor of the node is greater than 1
    a. If the balance factor of the left child <= 0, rotate right
    b. Else do a left-right rotation
2. If the balance factor of the node is less than -1
    a. If the balance factor of the right child <= 0, do left rotation
    b. Else do a right-left rotation [4]

Red-Black Trees

Introduction

Red-Black trees were created in 1972 by Rudolph Bayer under the name *symmetric binary B-trees* [6]. Similar to AVL trees, Red-Black Trees are self-balancing binary trees where each node has an extra bit, the colour (red or black). The balance conditions for red-black trees are as follows:

- Red Condition - each red node must have a black parent
- Black Condition - each path from the root to an empty node contains exactly the same number of black nodes. This is also referred to as the black height.

The red condition implies that the root of the entire tree must be black.

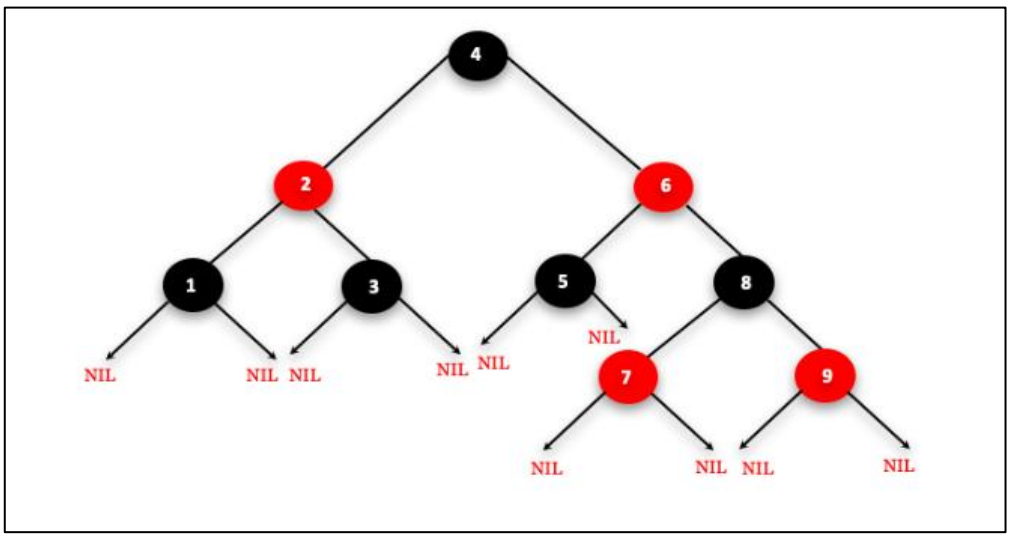

Figure 6 - Red-Black Tree

Implementation

Much like the AVL Trees, there are a variety of operations which can be performed on a red-black tree, namely insertion, deletion and rotation.

Insertion

When inserting a new node into a red-black tree, that node must always be red. This is because when inserting a red node, the depth property of the tree is not violated. If you attach a red node to a red node, then the rule is violated but it is easier to fix this problem than the problem introduced by violating the depth property.

Pseudocode

1. If the tree is empty, insert the new node as a root node and color it black.
2. If the tree is not empty, repeat the following steps until a leaf node is reached.
    a. Compare the new node with the root node.
    b. If the new node > root node, traverse through the right subtree.
    c. If the new node < root node, traverse through the left subtree.
3. Assign the parent of the leaf as a parent of the new node.
4. If the leaf node > new node, make the new node the right child.
5. If the leaf node < new node, make the new node the left child
6. Assign NULL to the left and right children of the new node.
7. Make the newly added node red.
8. Check to make sure the properties of the red-black tree are not being violated [7]

If the properties are being violated, for example, having two red nodes attached to each other, we carry out the following steps:

1. If p (new node) is the left child of grandparent node, do the following.
    a. Case 1:
        i. If the colour of the right child of the grandparent node is red, set the colour of both the children of the grandparent node as black and the colour of the grandparent node as red.
        ii. Assign the grandparent node to the new node.
    b. Case 2:
        i. If the new node is the right child of the red leaf node then, assign the parent node to the new node.
        ii. Left-rotate new node.
    c. Case 3:
        i. Set colour of the parent as black and the grandparent as red.
        ii. Right-rotate the grandparent.

2. Else, do the following.
    a. If the colour of the left child of the grandparent node is red, set the colour of both the children of the grandparent node as black and the colour of the grandparent node as red.
    b. Assign the grandparent node to the new node
    c. If the new node is the left child of the parent, assign the parent to the new node and right-rotate the new node.
    d. Set the colour of the parent as black and the colour of the grandparent as red.
    e. Left-rotate the grandparent node.
3. Set the root of the tree as black [7].

Deletion

Below is the pseudocode on how to delete a node from a red-black tree.

Pseudocode

1. Save the colour of the node to be deleted.
2. If the left child of node to be deleted is empty
    a. Assign the right child of node to be deleted to x.
    b. Transplant the node which needs to be deleted with x.
3. If the right child of node to be deleted is empty
    a. Assign the left child of node to be deleted into x.
    b. Transplant the node which needs to be deleted with x.
4. Else
    a. Assign the lowest value of the right subtree of node to be deleted into y.
    b. Save the colour of y.
    c. Assign the right child of y into x.
    d. If y is a child of the node to be deleted, set the parent of x as y.
    e. Else, transplant y with the right child of y.
    f. Transplant the node to be deleted with y.
    g. Set the colour of y with its original colour.
5. Check that the properties are not being violated [7]

If the properties are being violated, for example, the original colour is black, we carry out the following steps:

1. Do the following until the x is not the root of the tree and the colour of x is BLACK
2. If x is the left child of its parent then,
    a. Assign w to the sibling of x.
    b. If the right child of parent of x is red,
        i. Case 1:
            1. Set the colour of the right child of the parent of x as black.
            2. Set the colour of the parent of x as red.
            3. Left-rotate the parent of x.
            4. Assign the right child of the parent of x to w.
    c. If the colour of both the right and the left of w is black,
        i. Case 2:
            1. Set the colour of w as red
            2. Assign the parent of x to x.
    d. Else if the colour of the right child of w is black
        i. Case 3:
            1. Set the colour of the left child of w as black
            2. Set the colour of w as red
            3. Right-rotate w.
            4. Assign the right child of the parent of x to w.
    e. If any of the above cases do not occur, then do the following.
        i. Case-IV 4:
            1. Set the colour of w as the colour of the parent of x.
            2. Set the colour of the parent of x as black.
            3. Set the colour of the right child of w as black.
            4. Left-rotate the parent of x.
            5. Set x as the root of the tree.
3. Else the same as above with right changed to left and vice versa.
4. Set the colour of x as black [7].

AVL Trees vs Red-Black Trees

As was discussed in this report earlier, AVL trees and red-black trees are binary search trees which will self-sort and self-balance.

Since AVL trees are more strictly balanced than a red-black trees, AVL's have a generally quick look up time than red-black's.

However, the more relaxed balancing of red-black trees allow the insertion and removal of nodes to be much quicker than that of AVL trees.

While Red-Black trees stores just 1 bit for each node, AVL trees store heights for every node. This gives AVL trees a time complexity of O(n) and Red-Black trees a time complexity of O(1) [8].

In a real-world situation, red-black trees are more likely to be used in C++ language libraries such as map, multimap and multiset while AVL trees have better usage in a database setting were quick lookup times are vital.

References

[1] “AVL Tree | Set 1 (Insertion) - GeeksforGeeks,” *GeeksforGeeks*, Feb. 23, 2012. https://www.geeksforgeeks.org/avl-tree-set-1-insertion/

[2] “Red-Black Tree | Set 1 (Introduction) - GeeksforGeeks,” *GeeksforGeeks*, Feb. 04, 2014. https://www.geeksforgeeks.org/red-black-tree-set-1-introduction-2/?ref=lbp

[3] “Red-Black Tree,” *Programiz.com*, 2021. https://www.programiz.com/dsa/red-black-tree

[4] “AVL Tree,” *Programiz.com*, 2021. https://www.programiz.com/dsa/avl-tree#:~:text=Balance%20factor%20of%20a%20node,right%20subtree%20of%20that%20node.&text=The%20self%20balancing%20property%20of,1%2C%200%20or%20%2B1.

[5] A. K. Tsakalidis, “AVL-trees for localized search,” *Information and Control*, vol. 67, pp. 173–194, Oct. 1985.

[6] F. Paris and C. Okasaki, “Workshop on Algorithmic Aspects of Advanced Programming Languages WAAAPL’99,” 1999.

[7] “Red-Black Tree,” *Programiz.com*, 2021. https://www.programiz.com/dsa/red-black-tree

[8] C. Davis, J. Jackson, J. Oldfield, T. Johnson, and M. Hale, “A TIME COMPARISON BETWEEN AVL TREES AND RED BLACK TREES,.” Available: https://csce.ucmss.com/cr/books/2019/LFS/CSREA2019/FCS2407.pdf.

Table of Figures

Statement of Completion

Item	Completed (Yes/No/Partial)
AVL tree (insert, delete, search)	Partial (without search)
Red-Black tree (insert, delete, search)	Partial (without search)
Discussion comparing AVL to RBT	Yes