AVL Tree Documentation

Dávid Blaško

December 2024

1 Introduction

This paper contributes to the semester project in the Introduction to Mathematical Programming course at the Brno University of Technology, Faculty of Mechanical Engineering, Department of Automation and Computer Science. I have chosen this algorithm as an extension to the semester-covered topics, specifically binary search trees (BSTs).

2 Adelson-Velsky Landis Binary Search Tree

2.1 Brief history

The AVL tree was the first known self-balancing binary search tree named after two soviet inventors, **Georgy Adelson-Velsky** and **Evgenii Landis**. It was published in **1962** as *An algorithm for the organization of information.*

Georgy Maximovich Adelson-Velsky

Beginning in 1963, Adelson-Velsky headed the development of a computer chess program at the Institute for Theoretical and Experimental Physics in Moscow. In August 1992, Adelson-Velsky moved to Israel. He worked as a professor in the department of Mathematics and Computer Science, Bar Ilan University. Adelson-Velsky died on 26 April 2014, aged 92.

Figure 1: Adelson-Velsky in 1980s.

Evgenii Mikhailovich Landis

Was jewish soviet mathematician who worked mainly on partial differential equations. He studied and worked at Moscow State University. Later, he worked on uniqueness theorems for elliptic and parabolic differential equations, Harnack inequalities, and Phragmén–Lindelöf type theorems. Died on 1997, aged 76.

Figure 2: Landis in 1987 at Prague Potential Theory Conference.

2.2 Description

Its primary purpose is to maintain balance in the height of the tree to ensure efficient **search, insertion,** and **deletion** operations. The AVL tree algorithm enforces a balance condition where the height difference (balance factor) between the left and right subtrees of any node is at most 1. To maintain this condition, it uses rotations (single or double) to rebalance the tree after insertions or deletions.

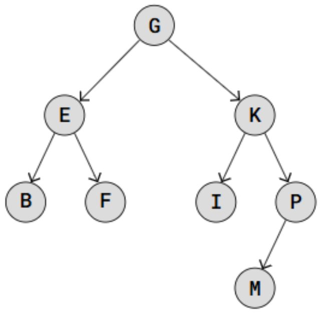

Figure 3: AVL Binary Search Tree visualization.

The balance factor for this figure is calculated as:

$$\text{Balance Factor} = \text{Height of Left Subtree} - \text{Height of Right Subtree}$$

$$-1 = 3 - 4$$

2.3 Time complexity

This innovation improved the worst-case time complexity of search operations to O(log n), ensuring consistent performance compared to unbalanced binary search trees. It remains a fundamental concept in data structures and algorithms.

AVL trees are often compared with red–black trees because both support the same set of operations and take O(log n) time for the basic operations. For lookup-intensive applications, AVL trees are faster than red–black trees because they are more strictly balanced. Height of AVL tree is 1.44*O(log n) compared to the Red-Black tree where it is 2*O(log n), where n is number of nodes (vertices).

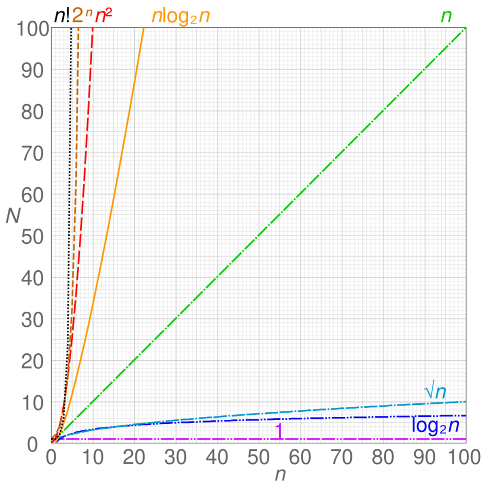

Figure 4: Time complexity graph.

2.4 Search

1. The search operation starts at the root of the AVL tree.
2. At each node, compare the key (vertice data) you're searching for with the key stored in the current node. If the key matches the node's key, you have found the item, and the search ends.
3. If the key is less than the current node's key, move to the left child of the current node. If the key is greater than the current node's key, move to the right child of the current node.
4. Repeat this process, moving left or right based on the comparison, until you either find the key, or you reach a null (leaf) node, indicating that the key is not in the tree.

2.5 Operations

Only two operations (methods) are needed. It's RotateLeft and RotateRight. RotateLeft is described in Figure 5, this is trivial example with only 3 nodes to simplify the problem. RotateRight will work analogically but mirrored.

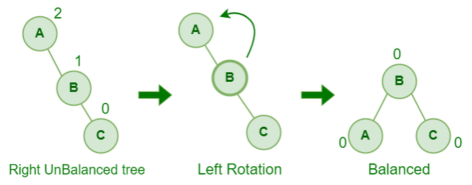

Figure 5: Left rotate method applied to right heavy tree.

2.6 Four Cases

With more than 3 nodes in tree we can have four different cases using those Left Rotate and Right Rotate methods. Those are:

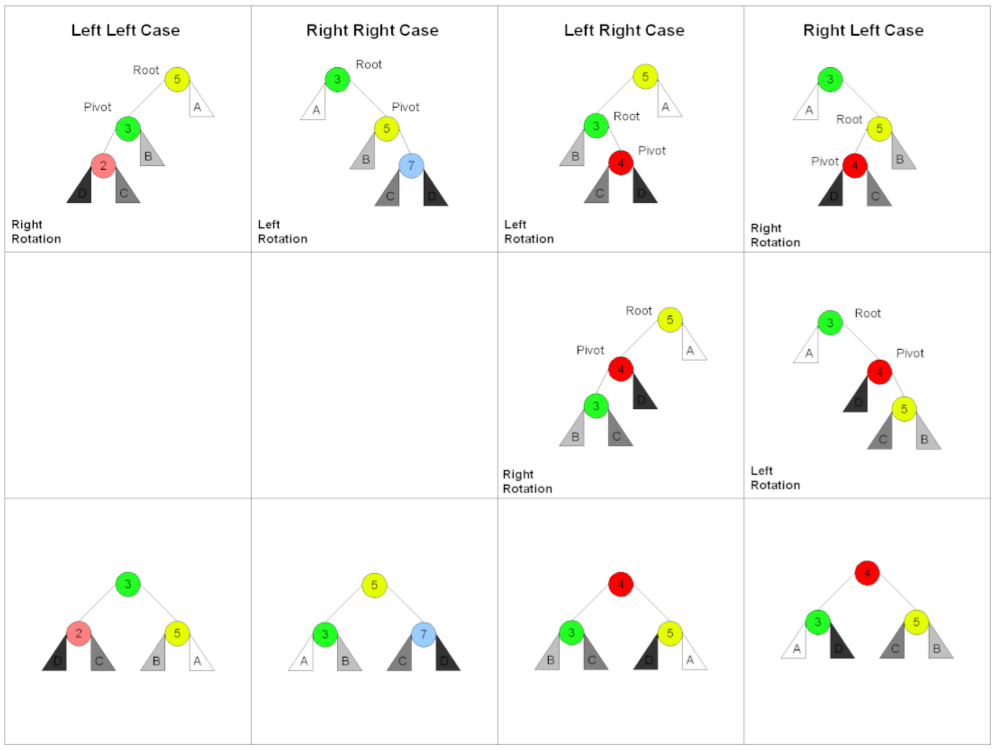

Figure 6: Four cases of AVL balancing.

2.7 Insertion and deletion algorithms

Algorithm 1: Pseudocode of method insert vertex

```
if $root = NULL$ then
    return Node(vertice_data) // Create and return a new node
else if $vertice\_data < root.vertice\_data$ then
    $root.left \leftarrow$ Insert_Vertice($root.left, vertice\_data$)
else
    $root.right \leftarrow$ Insert_Vertice($root.right, vertice\_data$)
// Update the height of the current node
$root.height \leftarrow 1 +$ max(Get_Height($root.left$), Get_Height($root.right$))
// Calculate the balance factor
$balance\_factor \leftarrow$ Get_Balance_Factor($root$)
// Rebalance the tree if necessary
if $balance\_factor > 1$ and $vertice\_data < root.left.vertice\_data$ then
    return Right_Rotate(root)
else if $balance\_factor < -1$ and $vertice\_data > root.right.vertice\_data$
 then
    return Left_Rotate(root)
else if $balance\_factor > 1$ and $vertice\_data > root.left.vertice\_data$
 then
    $root.left \leftarrow$ Left_Rotate($root.left$)  return Right_Rotate(root)
else if $balance\_factor < -1$ and $vertice\_data < root.right.vertice\_data$
 then
    $root.right \leftarrow$ Right_Rotate($root.right$)  return Left_Rotate(root)
return $root$ // Return the updated root
```

Algorithm 2: Pseudocode of method delete vertice

```
if $root = NULL$ then
    return $root$ // Base case:  tree is empty
if $vertice\_data < root.vertice\_data$ then
    $root.left \leftarrow$ Delete_Vertice($root.left, vertice\_data$)
else if $vertice\_data > root.vertice\_data$ then
    $root.right \leftarrow$ Delete_Vertice($root.right, vertice\_data$)
else
    // Node to be deleted is found
    if $root.left = NULL$ then
        $temp \leftarrow root.right$  $root \leftarrow$ NULL // Delete the node
        return $temp$
    else if $root.right = NULL$ then
        $temp \leftarrow root.left$  $root \leftarrow$ NULL // Delete the node
        return $temp$
    // Find the in-order successor
    $temp \leftarrow$ Get_Min_Node($root.right$)
    $root.vertice\_data \leftarrow temp.vertice\_data$
    $root.right \leftarrow$ Delete_Vertice($root.right, temp.vertice\_data$)
// Update the height of the current node
$root.height \leftarrow 1 +$ max(Get_Height($root.left$), Get_Height($root.right$))
// Rebalance the tree if necessary
$balance\_factor \leftarrow$ Get_Balance_Factor($root$)  if
 $balance\_factor > 1$ and Get_Balance_Factor($root.left$) $\geq 0$ then
    return Right_Rotate(root)
else if $balance\_factor < -1$ and Get_Balance_Factor($root.right$) $\leq 0$
 then
    return Left_Rotate(root)
else if $balance\_factor > 1$ and Get_Balance_Factor($root.left$) $< 0$
 then
    $root.left \leftarrow$ Left_Rotate($root.left$)  return Right_Rotate(root)
else if $balance\_factor < -1$ and Get_Balance_Factor($root.right$) $> 0$
 then
    $root.right \leftarrow$ Right_Rotate($root.right$)  return Left_Rotate(root)
return $root$ // Return the updated root
```

3 Conclusion

In conclusion, the AVL tree remains a cornerstone in the study of data structures, offering an efficient solution for managing dynamic datasets with guaranteed logarithmic time complexity for search, insertion, and deletion operations. By maintaining a strict balance through rotations, AVL trees minimize the height, ensuring consistent performance even in worst-case scenarios.

This paper has explored the historical context, key concepts, and operations of AVL trees, emphasizing their advantages over unbalanced binary search trees and alternative self-balancing structures like red-black trees. While AVL trees are optimal for lookup-heavy applications due to their strict balancing, they may incur slightly higher overhead during insertions and deletions compared to less balanced structures.

The insights provided here underscore the importance of AVL trees in computer science, particularly in systems requiring real-time data access and efficient memory utilization. Understanding these principles is crucial for applying AVL trees in practical scenarios and appreciating their role in the evolution of algorithm design.