Data Storage And Retrieval Using AVL Tree

Venkatakrishnan R

CH.EN.U4AIE20078

Computer Science and Engineering (AI)
Amrita Vishwa Vidyapeetham, Chennai

Siva Jyothi Nath Reddy B

CH.EN.U4AIE20063

Computer Science and Engineering (AI)
Amrita Vishwa Vidyapeetham, Chennai

Sarthak Yadav

CH.EN.U4AIE20058

Computer Science and Engineering (AI)
Amrita Vishwa Vidyapeetham, Chennai

Shaik Huziafa Fazil
CH.EN.U4AIE20060

Computer Science and Engineering (AI)
Amrita Vishwa Vidyapeetham, Chennai

Pravine Mukesh
CH.EN.U4AIE20050
Computer Science and Engineering (AI)
Amrita Vishwa Vidyapeetham, Chennai

1.Introduction

One of the most essential data structures is the tree, which is used to conduct operations like insertion, deletion, and searching of items efficiently. Construction of a well-balanced tree for sorting all data is not practicable when working with a huge number of data, though. As a result, only valuable data is saved as a tree, and the actual volume of data used changes over time as new data is inserted and old data is deleted. It is possible to conduct traversals, insertions, and deletions without utilizing either stack or recursion in some circumstances where the NULL link to a binary tree to special links is referred to as threads. In this project, we use the Height balance tree which is also known as the AVL tree.

2. AVL Tree

A binary search tree in which the height difference between the left and right subtrees of each node is less than or equal to one is known as an AVL tree. Adelson, Velskii, and Landi discovered the approach for balancing the height of binary trees, which is known as the AVL tree or Balanced Binary Tree.

AVL tree can be defined as follows :

Let T be a non-empty binary tree with the left and right subtrees TL and TR. If the following conditions are met, the tree is height balanced:

- TL and TR are height balanced
- hL - hR <= 1, where hL - hR are the heights of TL and TR

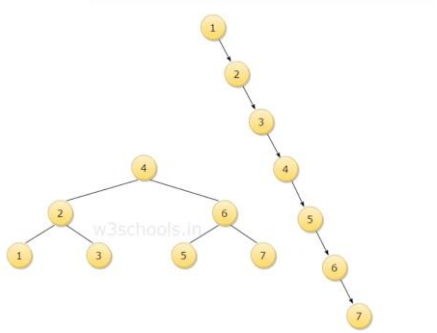
Depending on whether the height of the left subtree is larger, less than, or equal to the height of the right

subtree, the Balance factor of a node in a binary tree can be 1, -1, or 0.

3. Advantages of AVL Tree

Because AVL trees are height balance trees, operations such as insertion and deletion are quick. Consider the following scenario:

The binary tree will seem like the second figure if you have the following tree with keys 1, 2, 3, 4, 5, 6, 7:



The procedure for inserting a node with the key Q in a binary tree requires seven comparisons, but the algorithm for inserting the same key in an AVL tree requires three comparisons, as shown in the first picture.

4. Operations on AVL Tree

1) For Insertion
    - First, use BST's (Binary Search Tree) insertion logic to add a new element to the tree.
    - You must check the Balance Factor of each node once you have inserted the items.
    - The algorithm will proceed to the next operation once the Balance Factor of each node has been determined to be 0 or 1 or -1.
    - When any node's balance factor falls outside of the aforementioned three ranges, the tree is considered to be unbalanced. The algorithm will then proceed to the next operation after performing the appropriate Rotation to make it balanced.

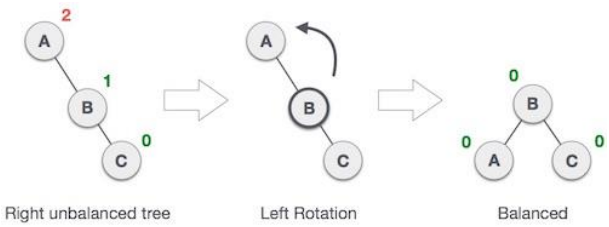
2) For Deletion
    - To begin, locate the node where k is stored.
    - Secondly, remove the node's contents (Suppose the node is x)
    - Claim: In an AVL tree, eliminating a leaf can lower the size of a node. There are three scenarios that could occur:
        - ➢ When x has no children then, delete x.
        - ➢ When x has one child, let x' becomes the child of x.
        - ➢ Notice: x' cannot have a child, since subtrees of T can differ in height by at most one :
            - ❖ then replace the contents of x with the contents of x'
            - ❖ then delete x' (a leaf)
    - When x has two children,
        - ➢ then find x's successor z (which has no left child)
        - ➢ then replace x's contents with z's contents, and
        - ➢ delete z

In all of the three cases, you will end up removing a leaf.

5. AVL Rotation

1) Left Rotation

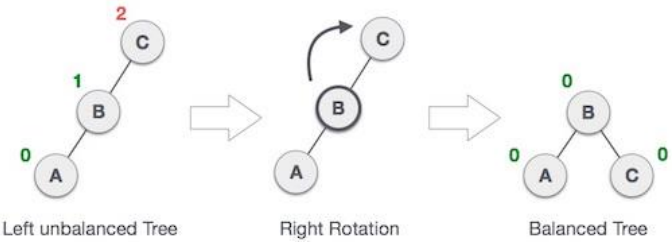
When a node is added into the right subtree of the right subtree, the tree becomes unbalanced, and we do a single left rotation.



As a node is inserted in the right subtree of A's right subtree, node A becomes unbalanced in this example. By making A the left-subtree of B, we may do the left rotation.

2) Right Rotation :

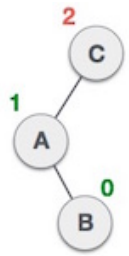
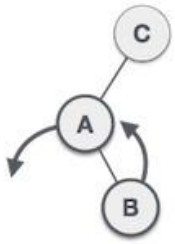
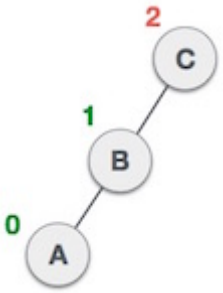
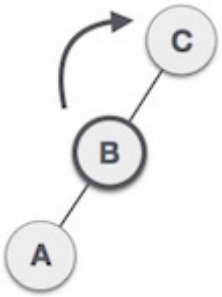
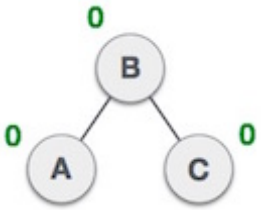
If a node is inserted in the left subtree of the left subtree, the AVL tree may become unstable. The tree must then be rotated correctly.



By completing a right rotation, the unbalanced node becomes the right child of its left child, as shown.

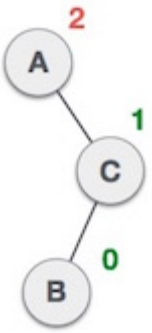
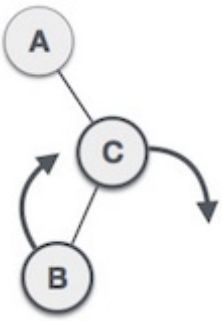
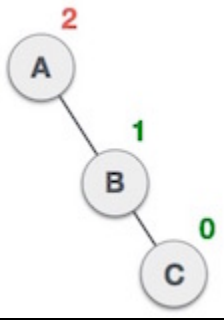
3) Left-Right Rotation

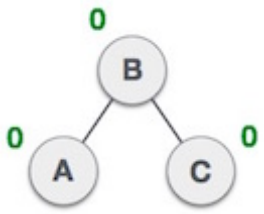
Double rotations are a more complicated variation of previously explained rotations. Take note of each action performed during rotation to better comprehend them. First, let's look at how to perform a Left-Right rotation. A combination of left and right rotations is known as a left-right rotation.

Action	State
A node has been added to the left subtree's right subtree. As a result, C is an unbalanced node. The AVL tree rotates left to right in these cases.	
On the left subtree of C, we first do the left rotation. As a result, A is the left subtree of B.	
Although Node C is still unbalanced, it is now due to the left-subtree of the left-subtree.	
We'll now right-rotate the tree, making B the new subtree's root node. C is now the left subtree of its own left subtree's right subtree.	
The tree is now balanced.	

4) Right-Left Rotation

Right-Left Rotation is the second type of double rotation. It consists of a combination of right and left rotations.

Action	State
A node has been added to the right subtree's left subtree. As a result, A becomes an unbalanced node with a balance factor of 2.	
To begin, we rotate C node to the right, making C the right subtree of its own left subtree B. B is now the right-hand subtree of A.	
Because of the right subtree of its right subtree, Node A is still imbalanced and requires a left rotation.	

The tree is now balanced.	

6. Results

We are creating an AVL Tree to store the data of COVID – 19.

The below snapshot shows time taken to create the AVL Tree for the given dataset :

```
CovidData [Java Application] C:\Users\ss261\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full\
AVLTrees of countries are created
Total creation time: 10.237
If details are required for all countries/date, enter 'All'
If not, enter the name of a specific country:
```

Now we will retrieve the details of COVID – 19 by entering the country name and specific date. This gives us the data of confirmed cases, deaths in country, and recovered cases of the above mentioned country. Also it shows the time taken to retrieve the data:

```
<terminated> CovidData [Java Application] C:\Users\ss261\.p2\pool\plugins\org.eclipse.justj.openjdk.hots
AVLTrees of countries are created
Total creation time: 10.237
If details are required for all countries/date, enter 'All'
If not, enter the name of a specific country:
US
Enter the date for which details are required in dd-mm-yy:
10-06-2020
The number of confirmed cases in US on 10-06-2020 are 2000464
The number of deaths in US on 10-06-2020 are 112924
The number of recovered cases in US on 10-06-2020 are 533504
Total execution time: 0.001
```

Note : We can also retrieve COVID – 19 data of all countries at once using this algorithm.

References

https://www.tutorialspoint.com/data_structures_algorithms/avl_tree_algorithm.htm

https://www.javatpoint.com/binary-search-tree-vs-avl-tree

https://www.w3schools.in/data-structures-tutorial/avl-trees/

https://sci-hub.hkvisa.net/10.1145/800197.806043