MIT World Peace University
Advanced Data Structures

Assignment 9

Naman Soni Roll No. 10

Contents

1 Problem Statement

A Dictionary stores keywords amp; its meaning. Provide facility for adding new keywords, deleting keywords, updating values of any entry. Provide facility to display whole data sorted in ascending/ Descending order. Also find how many maximum comparisons may require for finding any keyword. Use Height balance tree and find the complexity for finding a keyword.

2 Objective

- To study the concept of AVL trees
- To study different rotations applied on AVL tree

3 Theory

3.1 What is AVL tree?

AVL tree is a self-balancing binary search tree that was named after its inventors, Adelson-Velskii and Landis. It is a height-balanced binary search tree, which means that the height of the left and right subtrees of any node differs by at most one. The balance factor of a node is defined as the difference between the height of its left and right subtrees. If the balance factor of a node is greater than 1 or less than -1, the tree is considered unbalanced and needs to be rebalanced.

AVL tree supports efficient insertion, deletion, and lookup operations, with a worst-case time complexity of O(log n), where n is the number of nodes in the tree. This makes AVL tree a suitable data structure for applications that require frequent data manipulation and searching.

The self-balancing feature of AVL tree is achieved through rotations, which are operations that modify the tree structure while maintaining the search tree property and balance factor. There are four types of rotations that can be performed on an AVL tree: left-rotation, right-rotation, left-right-rotation, and right-left-rotation. These rotations are used to balance the tree and maintain the height balance property.

AVL tree has a few advantages over other self-balancing binary search trees, such as red-black tree. AVL tree provides faster search and insertion operations, but has a slightly slower deletion operation. Red-black tree, on the other hand, provides faster deletion operation, but has a slightly slower search and insertion operations.

Overall, AVL tree is a powerful data structure that provides efficient searching and manipulation operations, and is suitable for applications that require frequent data modification and searching.

3.2 Explain Different cases of AVL trees.

In an AVL tree, the balance factor of a node is the difference between the height of its left and right subtrees. If the balance factor of a node is greater than 1 or less than -1, the tree is considered unbalanced and needs to be rebalanced. There are four cases of AVL trees that arise when a node is inserted or deleted, and the balance factor of one or more nodes becomes greater than 1 or less than -1. These cases are as follows:

- Left-Left Case: In this case, the balance factor of a node A is greater than 1 and the balance factor of its left child B is also greater than 1. This means that the left subtree of A is taller than the right subtree, and the left subtree of B is taller than its right subtree. To rebalance the tree, we perform a right rotation on A to make B the new root of the subtree, and then perform a left rotation on B to restore the balance of the tree.
- Right-Right Case: In this case, the balance factor of a node A is less than -1 and the balance factor of its right child B is also less than -1. This means that the right subtree of A is taller than the left

subtree, and the right subtree of B is taller than its left subtree. To rebalance the tree, we perform a left rotation on A to make B the new root of the subtree, and then perform a right rotation on B to restore the balance of the tree.

- Left-Right Case: In this case, the balance factor of a node A is greater than 1 and the balance factor of its left child B is less than -1. This means that the left subtree of A is taller than the right subtree, and the right subtree of B is taller than its left subtree. To rebalance the tree, we perform a left rotation on B to make its right child C the new root of the subtree, and then perform a right rotation on A to make C the new root of the subtree and restore the balance of the tree.
- Right-Left Case: In this case, the balance factor of a node A is less than -1 and the balance factor of its right child B is greater than 1. This means that the right subtree of A is taller than the left subtree, and the left subtree of B is taller than its right subtree. To rebalance the tree, we perform a right rotation on B to make its left child C the new root of the subtree, and then perform a left rotation on A to make C the new root of the subtree and restore the balance of the tree.

By performing the appropriate rotations in each of these cases, we can maintain the balance of the AVL tree and ensure that the height of the tree is always logarithmic in the number of nodes.

3.3 Construction of AVL trees and Data Structure used for creation.

AVL trees are constructed using a recursive algorithm that inserts new nodes into the tree and ensures that the balance factor of each node is between -1 and 1. The algorithm works as follows:

- Start with an empty AVL tree.
- For each new node to be inserted, perform a standard binary search tree insertion.
- After the new node is inserted, check the balance factor of its parent node. If the balance factor is greater than 1 or less than -1, the tree is unbalanced and a rotation needs to be performed.
- Perform the appropriate rotation to rebalance the tree. There are four types of rotations: left rotation, right rotation, left-right rotation, and right-left rotation. The choice of rotation depends on the balance factor of the node and its children.
- Recursively check the balance factor of the parent node of the current node and perform rotations if necessary.
- Repeat steps 2 to 5 for all nodes to be inserted.

The data structure used for creating AVL trees is a linked list of nodes. Each node in the tree contains a key and a pointer to its left and right children. In addition, each node also contains the height of its left and right subtrees and the balance factor, which is the difference between the heights of the left and right subtrees.

To implement the rotations, additional information such as the parent of each node is also stored in the node structure. This information is used to update the pointers of the rotated nodes and their ancestors.

AVL trees are usually implemented using a self-balancing binary search tree library or module provided by the programming language. Many programming languages provide such libraries or modules, including C++, Java, Python, and many others. These libraries or modules provide efficient and optimized implementations of the AVL tree algorithms, making it easy to use AVL trees in various applications.

4 Implementation

4.1 *Platform*

- 64-bit Mac OS

- Open Source C++ Programming tool like Visual Studio Code

subsection*Test Conditions*

1. Input min 10 elements.
2. Display Max and Min Heap
3. Find Maximum and Minimum marks obtained in a particular subject.

5 Conclusion

Thus, we have implemented various rotation on AVL trees

6 FAQ's

6.1 *Discuss AVL trees with suitable example?*

Sure, let's consider an example to understand how AVL trees work. Suppose we want to insert the following values into an empty AVL tree: 10, 20, 30, 40, 50, 60, and 70.

Insert 10 into the empty tree. The tree now looks like this:

```
10
```

Insert 20 into the tree. The balance factor of the root node is now -1, which is within the range of -1 to 1. So, the tree remains balanced.

```
  10
   \
   20
```

Insert 30 into the tree. The balance factor of the root node is now -2, which is less than -1. So, the tree is unbalanced and we need to perform a rotation. In this case, we need to perform a left rotation on the root node to make 20 the new root of the subtree.

```
  10
   \
   20
     \
     30
```

Insert 40 into the tree. The balance factor of the root node is now -2, which is less than -1. So, the tree is unbalanced and we need to perform a left rotation on the root node to make 30 the new root of the subtree.

```
  10
   \
   30
  /  \
 20   40
```

Insert 50 into the tree. The balance factor of the root node is now -2, which is less than -1. So, the tree is unbalanced and we need to perform a left-right rotation on the root node. First, we perform a right rotation on the left child of the root (30) to make 40 the new root of the left subtree. Then, we perform a left rotation on the root (30) to make 40 the new root of the entire subtree.

```
  10
   \
   40
  /  \
 30   50
/
20
```

Insert 60 into the tree. The balance factor of the root node is now 2, which is greater than 1. So, the tree is unbalanced and we need to perform a right rotation on the root node to make 40 the new root of the subtree.

```
    30
   /  \
  20   40
         \
          50
           \
            60
```

Insert 70 into the tree. The balance factor of the root node is now 2, which is greater than 1. So, the tree is unbalanced and we need to perform a right-left rotation on the root node. First, we perform a left rotation on the right child of the root (40) to make 50 the new root of the right subtree. Then, we perform a right rotation on the root (30) to make 50 the new root of the entire subtree.

```
    40
   /  \
  30   50
 /    /  \
20   45   60
           \
            70
```

Now, we have successfully inserted all the values into the AVL tree and ensured that the balance factor of each node is between -1 and 1. The height of the AVL tree is now logarithmic in the number of nodes, which ensures efficient searching, insertion, and deletion operations.

6.2 *Compute the time complexity of AVL tree creation?*

The time complexity of creating an AVL tree depends on the number of nodes being inserted.

When inserting a new node, we need to perform a search to find the correct position for insertion, which takes O(log n) time in a balanced tree like an AVL tree. Once we find the correct position, we need to insert the node and adjust the balance factors of the nodes along the path from the root to the newly inserted node. This adjustment involves rotations, which also take O(log n) time.

Therefore, the time complexity of creating an AVL tree with n nodes is O(n log n), since we need to perform O(log n) operations for each of the n nodes being inserted. This time complexity ensures that the AVL tree remains balanced and efficient for searching, insertion, and deletion operations.