Data structures and Algorithms

Note07-AVL Trees

Edited By

Arshia Gharooni

2023

0.1 Introduction

In the previous session, it was mentioned that the order of various operations in a tree is equal to the height of the tree O(height(T), where the height can range from 1 to n. However, the goal in binary search trees is to reduce the height and reach a balanced tree, in which case the height becomes O(logn).

In general, any tree with a balanced logarithmic height is considered a balanced tree, and vice versa

0.2 AVL trees

Balanced trees are a data structure that was first introduced in 1962. The main feature of a balanced tree is the difference in height between the children of each node, such that the difference in height between the left and right children of each node is at most one.

If a node has only one child, the missing child is referred to as "none" and is considered to have a height of -1.

0.2.1 Height of AVL trees

we will prove that the height of a balanced AVL tree is at most $2log_n(n)$.

To prove this claim, we define N_h to be the minimum number of nodes at height h. Let h1 and h2 denote the heights of the left and right children of a given node, respectively. For an ordered pair (h1, h2), we have three cases:

- (h1, h2) = (h - 1, h - 1)
- (h1, h2) = (h - 1, h - 2)
- (h1, h2) = (h - 2, h - 1)

It is clear that the first case does not allow us to minimize the number of nodes. The second and third cases are equivalent, so we consider the heights (h - 1, h - 2):

$N_h = N_{h-1} + N_{h-2} + 1$

Since $N_{h-1} \geq N_{h-2}$, it follows that:

$N_h \geq 2N_{h-2} \geq 4N_{h-4} \geq 8N_{h-8} \geq ... \implies N_h \geq 2^{h/2}$

We know that:

$$n \geq N_h \implies n \geq 2^{h/2} \implies log_2 n \geq h/2 \implies h \leq 2log_2 n$$

Therefore, the height of a balanced AVL tree is at most $2log_2 n$.

0.2.2 Rotations

Rotations are a fundamental operation used to modify the structure of a binary tree while preserving its traversal order. The goal is to change the structure of a tree to reduce its height without altering the order in which nodes are visited during a traversal. A rotation is an operation that modifies the tree by changing the links between nodes in a way that maintains the order of the nodes during a traversal.

A rotation is a simple operation that changes the structure of a binary tree while preserving its traversal order. It relinks O(1) pointers to modify tree structure and maintain traversal order. The goal of a rotation is to reduce the height of a tree while preserving its traversal order.

A rotation can be performed on a binary tree with a specific node as the pivot point. The pivot point is the node around which the rotation is performed. There are two types of rotations: left rotations and right rotations.

Left Rotation

A left rotation is a type of rotation that is performed on a binary tree rooted at node A, with a right child B. During a left rotation, node B becomes the new root of the subtree, and node A becomes the left child of node B. The left subtree of node B becomes the right subtree of node A.

A left rotation can be performed using the following procedure:

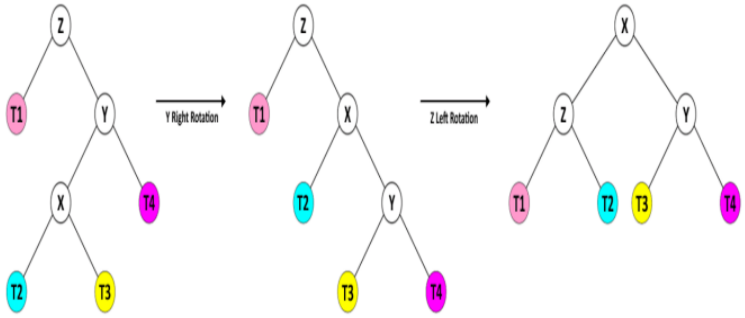
```
rotate_left(A):
    B = A.right
    A.right = B.left
    B.left = A
    return B
```

Left rotation operation

A right rotation is a type of rotation that is performed on a binary tree rooted at node A, with a left child B. During a right rotation, node B becomes the new root of the subtree, and node A becomes the right child of node B. The right subtree of node B becomes the left subtree of node A.

A right rotation can be performed using the following procedure:

```
rotate_right(A):
    B = A.left
    A.left = B.right
    B.right = A
    return B
```



0.2.3 Defined Operations for AVL Tree

The code related to the $find-next()$ and $find-prev()$ functions are implemented similar to a binary search tree.
However, for the insert() and delete() operations, we need to balance the tree by performing rotations on the affected nodes after insertion or deletion. For this purpose, starting from the bottom (leaves), we consider the first node that is unbalanced. Let's call it X. After performing an insertion or deletion operation, if its height is considered k+2, we assume the right child's height as k+1 and the left child's height as k-1.

Now, if we consider the right child Y, we can have three cases for the heights of Y, its right child h1, and its left child h2:

- (h1, h2) = (k, k): This case is impossible after inserting a node and can only happen after deleting a node.
- (h1, h2) = (k, k-1): In this case, we can solve the problem with a single left rotation on the node Y, and after rotation, the heights of the left and right children become equal. In this case, there is no need to examine the higher nodes.
- (h1, h2) = (k-1, k): In this case, we first perform a right rotation on Y and then a left rotation on X. After these steps, we move up again and if there is another unbalanced node, we perform the above steps according to the heights of its left and right children.

Sorting with AVL Tree

We know that an order-in AVL tree is sorted. Based on this fact, we can provide the following code for sorting:

```
def AVL_sort(A):
    T = AVL()
    for i in A:
        T.insert(i)
    return in_order_traversal(T)
```