Data Structures Fall 2023

Lab#12

Week#14

AVL Trees with Unit Testing Using Templates

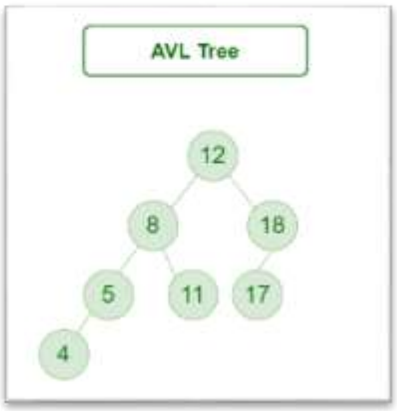
AVL Trees:

An AVL tree, named after its inventors Adelson-Velsky and Landis, is a self-balancing binary search tree. The key feature of AVL trees is that they maintain balance during insertions and deletions, ensuring that the tree remains relatively balanced, which helps in maintaining efficient search, insert, and delete operations.

In an AVL tree, for every node, the heights of the left and right subtrees differ by at most one. This balance condition is enforced after every insertion and deletion operation through a process called tree rotation. Tree rotations are local operations that maintain the balance of the tree while preserving the order of the elements.

The AVL tree's balancing property ensures that the height of the tree is logarithmic with respect to the number of nodes, resulting in efficient search times. The worst-case time complexity for basic operations (search, insert, delete) in an AVL tree is O(log n), where n is the number of nodes.

In summary, AVL trees provide a balanced and efficient data structure for dynamic sets or ordered maps, offering guaranteed logarithmic height and, consequently, efficient search and update operations.



Task 1:

Define a C++ class for an AVL Tree;

1. **AVLNode:** Implement a class for a node in an AVL tree. Include fields for data, a pointer to the left child, a pointer to the right child, and the height of the node.

Implement the following operations for the AVL Tree in;

AVLTree Class: Contains the root pointer of AVLNode type.

2. **AVL Tree Insertion:** Write a function to insert a new node into an AVL tree. Ensure that the tree remains balanced after the insertion by performing rotations if necessary

3. **AVL Tree Deletion**: Implement a function to delete a node with a given key from an AVL tree. Ensure that the tree remains balanced after the deletion by performing rotations if necessary

4. **AVL Tree Height:** Write a function to calculate the height of a node in an AVL tree.

5. **AVL Tree Balance Factor**: Implement a function to calculate the balance factor of a node in an AVL tree. The balance factor is the height of the left subtree minus the height of the right subtree.

6**. Check if AVL Tree:** Write a function to check if a given binary search tree is also an AVL tree. Ensure that the heights of the left and right subtrees differ by at most 1 for every node.

7. **AVL Tree Traversal:** Implement recursive functions for in-order traversal of an AVL tree.

8. **Find the Smallest and Largest Elements in the AVL Tree**: Write functions to find the smallest and largest elements in an AVL tree.

9. **AVL Tree Rotation:** Implement functions for left and right rotations in an AVL tree.

10. **AVL Tree Rebalancing:** Write a function that performs the necessary rotations to rebalance a node in an AVL tree after an insertion or deletion.

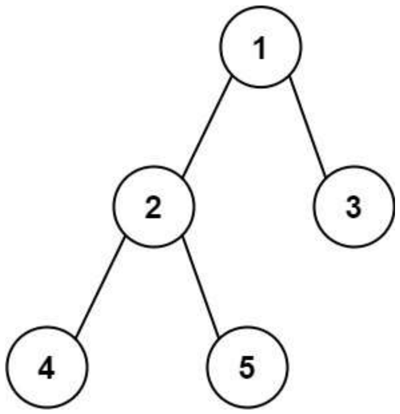
Task 2:

Given the root of a binary tree, return *the length of the* ***diameter*** *of the tree*.

The **diameter** of a binary tree is the **length** of the longest path between any two nodes in a tree. This path may or may not pass through the root.

The **length** of a path between two nodes is represented by the number of edges between them.

Example 1:



Input: root = [1,2,3,4,5]
Output: 3
Explanation: 3 is the length of the path [4,2,1,3] or [5,2,1,3].

Example 2:

Input: root = [1,2]
Output: 1