ADSA/STADSA
LAB- 01
Balancing Trees
Date- 25-08-25

- **What are trees?**
  A way to organize and store data in hierarchical manner.
  Key Characteristics of a Tree Data Structure:
  - **Root:** The starting point of the tree.
  - **Nodes:** Individual elements within the tree.
  - **Edges:** Connections between nodes.
  - **Child nodes:** Nodes connected to a parent node.
  - **Parent node:** A node with one or more child nodes.
  - **Leaf nodes:** Nodes without children.

- **What are binary trees?**
  A binary tree is a tree (a connected graph with no cycles) of binary nodes: a linked node container, similar to a linked list node, having a constant number of fields:
  - a pointer to an item stored at the node,
  - a pointer to a parent node (possibly None),
  - a pointer to a left child node (possibly None), and
  - a pointer to a right child node (possibly None).

- **Why is a binary node called "binary"?**
  In actuality, a binary node can be connected to three other nodes (its parent, left child, and right child), not just two. However, we will differentiate a node's parent from it's children, and so we call the node "binary" based on the number of children the node has.

Binary Search Trees (BSTs)

- **What are Binary Search Trees?**
  A Binary Search Tree (BST) is a specific type of binary tree where the values of the left child of a node are always less than the parent node's value, and the values of the right child are always greater. This ordering property makes BSTs efficient for search, insertion, and deletion operations.

- **Are they different from binary trees?**

Yes, they are different. While all BSTs are binary trees, not all binary trees are BSTs. The key distinction lies in the ordering of nodes. A binary tree doesn't impose any order on its nodes, whereas a BST strictly adheres to the left-less-than-parent-less-than-right rule.

- **How do you think they are different from binary trees?**
  As mentioned, the primary difference is the ordering of nodes. BSTs leverage this order to provide efficient search, insertion, and deletion operations, which is not guaranteed in general binary trees.

- **Why are trees actually needed as a data structure?**
  Trees are valuable data structures due to their hierarchical structure, which mirrors many real-world relationships. They offer:
  - **Efficient search, insertion, and deletion** in the case of BSTs.
  - **Hierarchical representation** of data, making them suitable for representing tree-like structures (e.g., file systems, organizational charts).
  - **Dynamic structure**, allowing for efficient modifications.
- **Why would we want to store items in a binary tree?**
  The difficulty with a linked list is that many linked-list nodes can be O(n) pointer hops away from the head of the list, so it may take O(n) time to reach them. By contrast, as we've seen in earlier recitations, it is possible to construct a binary tree on n nodes such that no node is more than O(log n) pointer hops away from the root, i.e., there exist binary trees with logarithmic height. The power of a binary tree structure is if we can keep the height h of the tree low, i.e., O(log n), and only perform operations on the tree that run in time on the order of the height of the tree, then these operations will run in O(h) = O(log n) time (which is much closer to O(1) than to O(n)).

- **A little about the complexity**
  - **Best-case:** For balanced BSTs, the time complexity for search, insertion, and deletion is O(log n), where n is the number of nodes.
  - **Worst-case:** For unbalanced BSTs, these operations can degrade to O(n) in the worst case.

- **Balancing Binary Tree Structures**
  A balanced binary tree is a tree where the height of the left and right subtrees of any node differ by at most one. Balancing is crucial for maintaining efficient operations in BSTs (search, insert, delete) in O(log n). Unbalanced trees can degenerate into linked lists in the worst case, leading to poor performance.

- **How do we balance tree structures?**
  Several techniques are used to balance BSTs:
  - **AVL trees:** Maintain balance by using height information at each node and performing rotations to restore balance after insertions or deletions.
  - **Red-Black trees:** Use color information (red or black) and rotations to ensure balance.
  - **B-trees:** Designed for external storage, they maintain balance by allowing multiple keys and children per node.

  These techniques aim to keep the tree height as small as possible, ensuring efficient operations.

- **Tree rotations:**
  Balance Factor: height of left subtree-height of the right subtree

1. Left Rotation

A left rotation is performed when the right subtree of a node becomes too heavy.

Logic:

a) **Identify the imbalance:** Determine if the right subtree's height is greater than the left subtree's by 2.
b) **Set new root:** The node's right child becomes the new root. (We find out the balance factor of each node if it is not in the set {-1,0,1} then set this node's right child as the root)
c) **Adjust child pointers:**
   - The new root's left child becomes the old root.
   - The old root's right child becomes the new root's right child.
d) **Update heights:** Adjust the heights of the affected nodes.

2. Right Rotation

A right rotation is performed when the left subtree of a node becomes too heavy.

Logic:

a) **Identify the imbalance:** Determine if the left subtree's height is greater than the right subtree's by 2.
b) **Set new root:** The node's left child becomes the new root. (We find out the balance factor of each node if it is not in the set {-1,0,1} then set this node's left child as the root)
c) **Adjust child pointers:**

- The new root's right child becomes the old root.
- The old root's left child becomes the new root's left child.

d) **Update heights:** Adjust the heights of the affected nodes.

In terms of codes:

Pointers and Node Structure

A typical tree node structure includes:

- data: The value stored in the node.
- left: A pointer to the left child node.
- right: A pointer to the right child node.
- height: (Optional) The height of the subtree rooted at this node.

Code snippet:

```
struct Node {
  int data; // Data stored in the node
  struct Node *left; // Pointer to the left child
  struct Node *right; // Pointer to the right child
  int height; // Height of the subtree rooted at this node (for AVL trees)
};
```

Left Rotation using Pointers

1. **Identify the node:** Determine the node that needs a left rotation (usually based on balance factors).
2. **Set new root:** The node's right child becomes the new root.
3. **Adjust left child:** The new root's left child becomes the old root.
4. **Adjust right child:** The old root's right child becomes the new root's right child.

```
struct Node* leftRotate(struct Node* x) {
  // Function to perform a left rotation on the given node x
  struct Node* y = x->right; // Set y as the right child of x
  x->right = y->left; // Make the left child of y as the right child of x
```

```
 // Establish the link between x and the subtree rooted at y
 y->left = x;
 // Update heights of the affected nodes (for AVL trees)
 x->height = max(height(x->left), height(x->right)) + 1;
 y->height = max(height(y->left), height(y->right)) + 1;
 // Return the new root of the subtree
 return y;
}
```

Right Rotation using Pointers

1. **Identify the node:** Determine the node that needs a right rotation.
2. **Set new root:** The node's left child becomes the new root.
3. **Adjust right child:** The new root's right child becomes the old root.
4. **Adjust left child:** The old root's left child becomes the new root's left child.

```
struct Node* rightRotate(struct Node* y) {
 // Function to perform a right rotation on the given node y
 struct Node* x = y->left; // Set x as the left child of y
 y->left = x->right; // Make the right child of x as the left child of y
 // Establish the link between y and the subtree rooted at x
 x->right = y;
 // Update heights of the affected nodes (for AVL trees)
 x->height = max(height(x->left), height(x->right)) + 1;
 y->height = max(height(y->left), height(y->right)) + 1;
 // Return the new root of the subtree
 return x;}
```