Step 3: Optimized Binary Tree Implementation with LLM Assistance

---

LLM Analysis and Recommendations

The LLM analyzed the provided **unoptimized binary tree** and recommended improvements in the following areas:

1. **Tree Balancing:**
   - The existing binary tree lacked balancing, making worst-case operations **O(n)** instead of **O(log n)**.
   - Solution: Implement **AVL Tree** (self-balancing BST) to maintain efficiency.
2. **Search Functionality:**
   - No search function was implemented.
   - Solution: Introduce an **efficient search method** to retrieve tasks.
3. **Recursive Optimization:**
   - Some operations (e.g., height calculation) were repeatedly computed in recursive calls.
   - Solution: Store **height as a class attribute** and update it dynamically.
4. **Memory and Performance Enhancements:**
   - Avoid excessive recursion depth by optimizing insert and search operations.
   - Solution: Reduce redundant recursive calls and implement **iterative search**.

Optimized AVL Tree Implementation (C#)

Here's the improved **AVL Tree** implementation incorporating the LLM's suggestions.

```
using System;


public class Node

{

   public int Value;

   public Node Left, Right;

   public int Height;
```

```
    public Node(int value)
    {
        Value = value;
        Left = Right = null;
        Height = 1; // Initialize height to 1 instead of recalculating later
    }
}

public class AVLTree
{
    private Node root;

    // Get height of a node (prevents redundant recursion)
    private int GetHeight(Node node) => node?.Height ?? 0;

    // Calculate balance factor of a node
    private int GetBalanceFactor(Node node) => node == null ? 0 : GetHeight(node.Left) -
GetHeight(node.Right);

    // **Rotation functions for AVL balancing**
    private Node RotateRight(Node y)
    {
        Node x = y.Left;
        Node T2 = x.Right;

        // Perform rotation
        x.Right = y;
        y.Left = T2;

        // Update heights
```

```
    y.Height = Math.Max(GetHeight(y.Left), GetHeight(y.Right)) + 1;
    x.Height = Math.Max(GetHeight(x.Left), GetHeight(x.Right)) + 1;

    return x;
}

private Node RotateLeft(Node x)
{
    Node y = x.Right;
    Node T2 = y.Left;

    // Perform rotation
    y.Left = x;
    x.Right = T2;

    // Update heights
    x.Height = Math.Max(GetHeight(x.Left), GetHeight(x.Right)) + 1;
    y.Height = Math.Max(GetHeight(y.Left), GetHeight(y.Right)) + 1;

    return y;
}

// **Optimized Insert Function (Balanced Insertion)**
private Node Insert(Node node, int value)
{
    // Base case: Insert new node
    if (node == null)
        return new Node(value);
```

```
// Standard BST insertion
if (value < node.Value)
    node.Left = Insert(node.Left, value);
else if (value > node.Value)
    node.Right = Insert(node.Right, value);
else
    return node; // Avoid duplicates

// Update height dynamically instead of recalculating
node.Height = Math.Max(GetHeight(node.Left), GetHeight(node.Right)) + 1;

// Get balance factor to check if balancing is needed
int balance = GetBalanceFactor(node);

// **Apply necessary rotations to balance the tree**
if (balance > 1 && value < node.Left.Value)
    return RotateRight(node);

if (balance < -1 && value > node.Right.Value)
    return RotateLeft(node);

if (balance > 1 && value > node.Left.Value)
{
    node.Left = RotateLeft(node.Left);
    return RotateRight(node);
}

if (balance < -1 && value < node.Right.Value)
{
```

```
        node.Right = RotateRight(node.Right);
        return RotateLeft(node);
    }

    return node;
}

public void Insert(int value) => root = Insert(root, value);

// **Optimized Search Function (Iterative)**
public bool Search(int value)
{
    Node current = root;

    while (current != null)
    {
        if (value == current.Value)
            return true;
        else if (value < current.Value)
            current = current.Left;
        else
            current = current.Right;
    }

    return false; // Not found
}

// **Inorder Traversal (Memory Optimized)**
public void PrintInOrder() => PrintInOrder(root);
```

```
    private void PrintInOrder(Node node)
    {
        if (node == null) return;
        PrintInOrder(node.Left);
        Console.Write(node.Value + " ");
        PrintInOrder(node.Right);
    }
}

class Program
{
    static void Main()
    {
        AVLTree tree = new AVLTree();

        tree.Insert(30);
        tree.Insert(20);
        tree.Insert(40);
        tree.Insert(10);
        tree.Insert(25);

        Console.WriteLine("Inorder traversal:");
        tree.PrintInOrder(); // Output: 10 20 25 30 40

        Console.WriteLine("\nSearch 25: " + tree.Search(25)); // Output: True
        Console.WriteLine("Search 15: " + tree.Search(15)); // Output: False
    }
}
```

Annotated Code Comments (LLM-Generated Improvements)

1. **Tree Balancing (AVL Tree)**
   - Implemented **AVL rotations** (RotateLeft() and RotateRight()) to **maintain balance** and prevent skewed trees.
   - Ensures insert/search operations remain **O(log n)**.
2. **Search Functionality**
   - Introduced **iterative search** (Search()) instead of a **recursive approach** to **reduce stack memory usage**.
3. **Recursive Optimization**
   - Removed redundant height recalculations by **storing the height** within each node.
   - GetHeight() now retrieves the stored height instead of **recalculating via recursion**.
4. **Memory & Performance Enhancements**
   - **Avoids unnecessary recursion depth** by using **iterative search**.
   - **Prevents duplicate insertions** by checking if the value already exists.

Step 4: Reflection

How did the LLM assist in refining the code?

The LLM provided the following key insights:

- Suggested using **AVL Tree** instead of an unbalanced BST for better performance.
- Recommended **iterative search** to optimize memory.
- Identified redundant height recalculations and **optimized height storage**.
- Pointed out the **need for a self-balancing mechanism** (rotations).
- Highlighted **performance bottlenecks** in recursion depth.

Were any LLM-generated suggestions inaccurate or unnecessary?

- Initially, the LLM suggested using a **Red-Black Tree**, which is better for frequent insertions and deletions. However, since **task retrieval (search) is the priority**, an **AVL Tree** was more appropriate.

- It also suggested implementing a **priority queue-based solution (Heap)**, but for this problem, an **AVL Tree** was the best balance between insertion speed and search efficiency.

What were the most impactful improvements you implemented?

1. **Implemented AVL Tree for balancing**, ensuring **O(log n) performance** for search and insert.
2. **Optimized search with an iterative approach**, reducing **memory overhead** from recursion.
3. **Reduced redundant calculations**, improving overall efficiency.
4. **Maintained a dynamically updated height attribute**, eliminating unnecessary recursive calls.

Final Thoughts

The LLM was highly beneficial in identifying inefficiencies and optimizing the binary tree implementation. The final AVL Tree solution ensures **fast, balanced insertions and searches**, making SwiftCollab's task assignment system **scalable and efficient**. 🚀