Lesson:

Interview Problems on BST-1

Pre Requisites:

- Trees
- BST

Concepts Involved

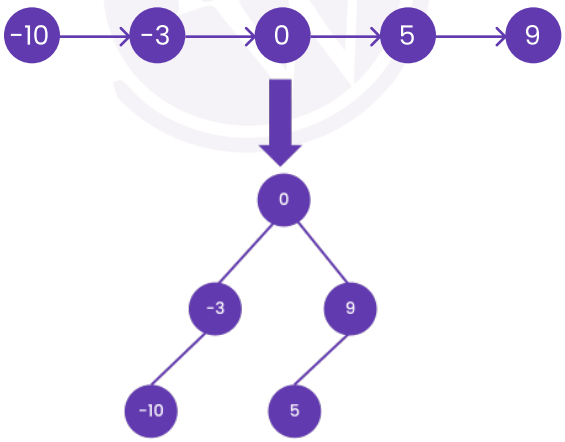
- Convert Sorted List to Binary Search Tree
- Construct BST from Preorder Traversal
- Convert BST to Greater Sum Tree
- Minimum Distance between BST Nodes
- Inorder of Tree using Morris Traversal

Q1. Convert Sorted List to Binary Search Tree [Leetcode-109]

Given the head of a singly linked list where elements are sorted in ascending order, convert it to a height-balanced binary search tree.

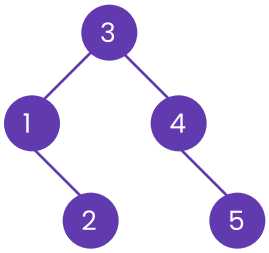
Take input of linked list from the user and output the level order and inorder traversal of BST.

Input:
head = [-10,-3,0,5,9]
Output:
Level order traversal = [0, -3, 9, -10, 5]
Inorder traversal = [-10, -3, 0, 9, 5]



Input:
head = [1, 2, 3, 4, 5]
Output:
Level order traversal = [3, 1, 4, 2, 5]
Inorder traversal = [1, 2, 3, 4, 5]

```
1 -> 2 -> 3 -> 4 -> 5
```



Solution:
Code: https://pastebin.com/j7MmXSVy

Explanation:

1. Find the middle element of the linked list using the two-pointer technique (slow and fast pointers). This can be done by advancing the slow pointer one step at a time and the fast pointer two steps at a time. The slow pointer will be pointing to the middle element when the fast pointer reaches the end of the list or null.
2. Create a new tree node with the value of the middle element. This node will be the root of the current subtree.
3. Recursively repeat steps 1 and 2 for the left half of the linked list, i.e., the nodes before the middle element. Set the left child of the current node as the result of the recursive call.
4. Recursively repeat steps 1 and 2 for the right half of the linked list, i.e., the nodes after the middle element. Set the right child of the current node as the result of the recursive call.
5. Return the root of the subtree.

The above steps ensure that the binary search tree is height-balanced because the middle element of the linked list is chosen as the root of each subtree, which helps maintain balance. The recursive approach allows us to divide the list into smaller halves until the base case is reached (when there are no more nodes in the linked list).

Output:

```
Enter the elements of the sorted Linked List: 1 2 3 4 5

Level Order Traversal of the BST: 3 2 5 1 4
Inorder Traversal of the BST: 1 2 3 4 5

...Program finished with exit code 0
Press ENTER to exit console.
```

```
Enter the elements of the sorted Linked List: -10 -3 0 5 9

Level Order Traversal of the BST: 0 -3 9 -10 5
Inorder Traversal of the BST: -10 -3 0 5 9

...Program finished with exit code 0
Press ENTER to exit console.
```

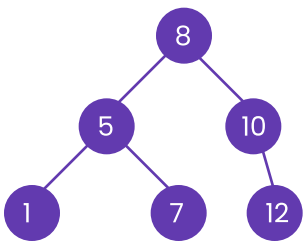
Time complexity: O(n), where n is the number of elements in the linked list. This is because we need to visit each element of the linked list once to construct the binary search tree.

Space complexity: O(log n) on average, considering the recursive calls made during the construction of the binary search tree. This is because the height of a balanced binary search tree is logarithmic in the number of nodes. However, in the worst-case scenario where the linked list is already sorted in ascending or descending order, the space complexity can be O(n) due to the recursive call stack reaching its maximum depth.

Q2. Construct BST from Preorder Traversal [Leetcode-1008]

Take preorder traversal from the user and create a BST from it. Output inorder and level order traversal of the BST.

Input:
preorder = [8,5,1,7,10,12]
Output:
Inorder Traversal of the BST: 1 5 7 8 10 12
Level Order Traversal of the BST: 8 5 10 1 7 12



Code: https://pastebin.com/2aYAkp5H

Explanation:

1. Take the first element of the preorder traversal and create a new node with that value. This node will be the root of the BST.
2. Iterate through the remaining elements of the preorder traversal.
3. For each element, compare it with the current node's value:
4. If the element is smaller, it belongs to the left subtree. Create a new node with the element and make it the left child of the current node.
5. If the element is larger, it belongs to the right subtree. Create a new node with the element and make it the right child of the current node.
6. Keep track of the index where the first element larger than the current node's value is found. This index separates the elements for the left subtree from those for the right subtree.
7. Recursively repeat steps 3 and 4 for the left and right subtrees:
8. For the left subtree, use the elements from the start of the preorder traversal up to the index found in step 3.
9. For the right subtree, use the elements from the index found in step 3 to the end of the preorder traversal.
10. Continue this process until all elements in the preorder traversal have been processed.
11. The resulting structure is a valid BST.

```
Enter the number of nodes: 6
Enter the preorder traversal: 8 5 1 7 10 12

Inorder Traversal of the BST: 1 5 7 8 10 12
Level Order Traversal of the BST: 8 5 10 1 7 12

...Program finished with exit code 0
Press ENTER to exit console.
```

Time complexity: O(n), where n is the number of nodes in the BST. This is because each node in the preorder

traversal is processed exactly once, resulting in a linear time complexity.

Space complexity: O(n), where n is the number of nodes in the BST. This is because the algorithm requires additional space to store the recursive call stack during the construction process. In the worst case scenario, when the BST is skewed, the space complexity approaches O(n) due to the recursive calls.

Q3. Convert BST to Greater Sum Tree [Leetcode-538]

Input: A pointer to the root of the Binary Search Tree
Output: A pointer to the root of the Greater Sum Tree
A greater sum tree is a binary tree where each node's value is replaced by the sum of all node values greater than itself.
Explanation:

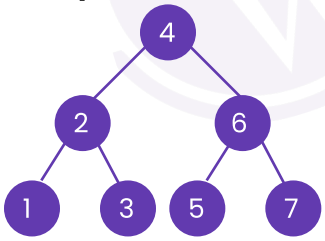
1. The idea behind this program is to traverse the BST in reverse order (right, root, left) while keeping track of the sum of all nodes visited so far.
2. We start at the rightmost node (the largest value in the BST) and add its value to the running sum.
3. We then update the node's value to be the running sum.
4. We then traverse to the left child and repeat the process.
5. Since we are traversing the BST in reverse order, we are essentially converting it to a Greater Sum Tree.

Code: https://pastebin.com/9AwzSeqV

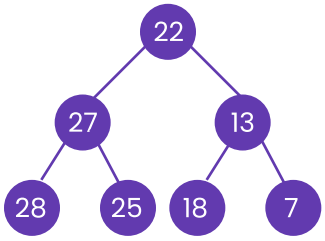
Output:

```
Binary Search Tree: 4 2 1 3 6 5 7
Resulting Greater Sum Tree: 22 27 28 25 13 18 7
```

Binary Search Tree:



Greater Sum Tree:

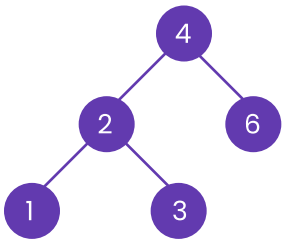


Time Complexity: The time complexity of this program is O(n), where n is the number of nodes in the BST. This is because we visit each node in the BST once.
Space Complexity: The space complexity of this program is O(h), where h is the height of the BST. This is because the maximum number of function calls on the call stack at any given time is equal to the height of the BST. In the worst case, where the BST is a degenerate tree (essentially a linked list), the space complexity would be O(n).

Q4. Minimum Distance between BST Nodes [Leetcode-783]

Given the root of a Binary Search Tree (BST), return the minimum difference between the values of any two different nodes in the tree.

Input: root = [4,2,6,1,3]
Output: 1



Code: https://pastebin.com/ARZnCAWq

Explanation:

1. Start with an initial minimum difference value set to a large number.
2. Initialize a reference to keep track of the previously visited node as null.
3. Perform an in-order traversal of the Binary Search Tree.
4. During the traversal:
   a. Recursively visit the left subtree.
   b. Compare the current node's value with the previously visited node's value.
   c. Update the minimum difference if the current difference is smaller.
   d. Update the reference to the current node as the previously visited node.
   e. Recursively visit the right subtree.
5. After the traversal, the minimum difference will be stored in the minimum difference variable.
6. Return the minimum difference as the result.

```
Minimum Difference: 1


...Program finished with exit code 0
Press ENTER to exit console.
```

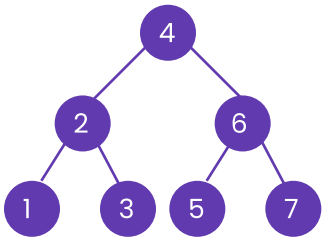
Time complexity: O(N), where N is the number of nodes in the tree. This is because we need to visit each node once during the in-order traversal.

Space complexity: O(H), where H is the height of the BST. In the worst case, where the tree is skewed, the height of the tree can be equal to the number of nodes N, resulting in O(N) space complexity. However, in a balanced BST, the height is logarithmic in N, resulting in a space complexity of O(log N). The space is used for the recursive call stack during the in-order traversal.

Q5. Inorder of Tree using Morris Traversal [Leetcode-94]

Given the root of a binary tree, return the inorder traversal of its nodes' values using Morris Traversal.

Input BST:



Output:
Inorder traversal: [1, 2, 3, 4, 5, 6, 7]

Code: https://pastebin.com/dR9RpM3e

Explanation:

1. Start with the current node as the root of the BST.
2. While the current node is not null:
   a. If the current node has no left child, visit the current node and move to its right child.
   b. If the current node has a left child, find its in-order predecessor (the rightmost node in its left subtree or the node with the largest value smaller than the current node).
      i. If the predecessor's right child is null, establish a temporary link from the predecessor to the current node and move to the left child of the current node.
      ii. If the predecessor's right child is the current node, break the temporary link, visit the current node, and move to its right child.
3. Repeat steps 2 until all nodes have been visited.

The Morris Traversal algorithm utilizes the fact that in the inorder traversal of a BST, the predecessor of a node is always visited immediately before the node itself.

```
Inorder traversal: [1, 2, 3, 4, 5, 6, 7]


...Program finished with exit code 0
Press ENTER to exit console.
```

Time complexity: O(n), where n is the number of nodes in the BST. This is because each node is visited exactly twice (once when establishing the temporary link and once when breaking it), resulting in a linear time complexity.

Space complexity: O(1) as the algorithm utilizes a constant amount of extra space for the temporary links, without requiring additional data structures like stacks or queues.

Upcoming Lecture

- Some more problems on BST