PW SKILLS

Lesson Plan

BST-1

Java

Today's Checklist

- Why Binary Search Tree?
- What is Binary Search Tree?
- Advantages
- Disadvantages
- Applications
- Insertion
- Traversal - Inorder, Preorder, Postorder
- Searching
- Practice problems on BST

Why Binary Search Tree?

Say, for a college trip, we have created a booking system for booking seats in a bus of capacity 30. Now, our booking system does two functions:

1. Book the seat number entered by the user.
2. Search if the given seat is already booked or not.

We have seat numbers from 1 to 20. And the stream of integers gets added up and the search operation for seat number can be performed anytime.

Currently following seats are booked:

Seats Booked: 3, 15, 8, 1, 29, 14
A user wishes to check if seat 10 is booked or not.
So, what is the most efficient way to search? Binary Search?
But But, in that case we need to sort the integers.

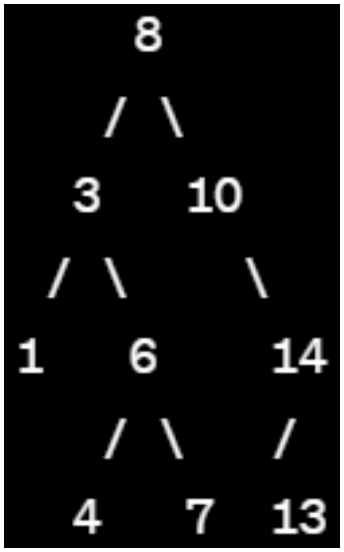
Seats Booked (Sorted): 1, 3, 8, 14, 15, 29
Let's say we sorted them, then if 10 is added we need to sort it again before another search.

Adding new booked seat: 1, 3, 8, 14, 15, 29, 10
If we do this way, then let's checkout the time complexity of our system's algorithm.
Binary Seach takes O(log2N) for search operation. (N: No. of seats booked)
Everytime a new seat is booked we need to sort the seats for binary search.
So, O(N log 2N) is the cost of booking new seat. And if there are say M queries, then the time complexity becomes O(M*N*log2N) which is clearly not efficient.

Thus, whenever our data set is constantly updating i.e. dynamic and we need to perform different operations on it like seach, add, update, all at once, then we use Binary Search Tree.

What is Binary Search Tree?

A binary search tree (BST) is a type of tree data structure where each node has at most two child nodes, referred to as the left child and the right child. The values stored in the left subtree of a node are less than or equal to the value of the node, while the values stored in the right subtree are greater than the value of the node. This property makes searching and insertion of data efficient, with an average time complexity of O(log n), where n is the number of nodes in the tree.



In this example, the root node has a value of 8. The left subtree contains nodes with values 3, 1, 6, 4, and 7, while the right subtree contains nodes with values 10, 14, and 13. The values in the left subtree are all less than the value of the root node, while the values in the right subtree are all greater than the value of the root node. This satisfies the property of a binary search tree.

Advantages:

Some advantages of using a binary search tree (BST) are:

1. **Efficient searching:** The structure of a BST allows for efficient searching of elements. The time complexity of searching for an element in a BST is O(log n) on average, where n is the number of elements in the tree.
2. **Sorted ordering:** BSTs maintain a sorted ordering of elements, with smaller elements to the left of a node and larger elements to the right. This makes it easy to perform operations such as finding the minimum or maximum element in the tree.
3. **Easy insertion and deletion:** Inserting and deleting elements in a BST is relatively easy and efficient, with a time complexity of O(log n) on average. This makes BSTs a good choice for applications where elements may need to be added or removed frequently.
4. **Space efficiency:** BSTs can be implemented with a relatively small amount of memory, as each node only needs to store three-pointers (to its left and right children, and to its parent node).
5. **Versatility:** BSTs can be used to implement a variety of other data structures, such as sets, maps, and priority queues. They can also be used for a variety of applications, such as searching, sorting, and indexing.

Disadvantages

Some disadvantages of using a binary search tree (BST):

1. **Unbalanced trees:** If the BST is not balanced, it can lead to worst-case time complexity of O(n) for certain operations, where n is the number of elements in the tree. This can happen if the tree is heavily skewed to one side due to the order in which elements were inserted or deleted.

2. **Memory requirements:** Although BSTs are relatively space-efficient, they can require a lot of memory for large trees. This is because each node in the tree requires three pointers (to its left and right children, and to its parent node).

3. **Lack of support for range queries:** Although BSTs allow efficient searching for a specific element, they do not provide built-in support for range queries (such as finding all elements between two values). This can make certain types of queries more difficult or inefficient to perform.

4. **Slow performance for certain operations:** Although searching, insertion, and deletion are efficient on average, certain operations such as finding the kth smallest element in the tree or balancing the tree can have worst-case time complexity of O(n).

5. **Complexity of implementation:** Implementing a binary search tree correctly can be complex, particularly for operations such as balancing the tree or handling cases where nodes have multiple children.

Q. Can a BST contain duplicate elements ?

Ans. BSTs can accommodate duplicate elements depending on the implementation. Some versions allow duplicates, positioning them to the right or left of a node, while others may handle them differently or disallow duplicates altogether.

Applications

Binary search trees (BSTs) have a wide range of applications. Here are some common applications of BSTs:

1. **Searching and indexing:** BSTs are commonly used for searching and indexing applications. Because BSTs maintain a sorted ordering of elements, they allow for efficient searching and retrieval of elements, with a time complexity of O(log n) on average.

2. **Sets and maps:** BSTs can be used to implement sets and maps, where the elements in the set or map are stored in sorted order. This makes it easy to perform operations such as finding the minimum or maximum element in the set or map.

3. **Priority queues:** BSTs can be used to implement priority queues, where the highest-priority element is always at the root of the tree. This makes it easy to perform operations such as adding and removing elements in priority order.

4. **File systems:** BSTs are commonly used in file systems to store the hierarchy of files and directories. In a file system, each directory is represented by a node in the tree, with its children representing the files and subdirectories contained within it.

5. **Network routing:** BSTs can be used in network routing algorithms to find the shortest path between two nodes in a network. In this application, each node in the tree represents a network node, and the edges between nodes represent the paths between them.

Real-life applications of BST

1. **Phone book:** In a phone book, the names are usually arranged in alphabetical order. BSTs can be used to store the names and phone numbers, with each node representing a person and its left and right subtrees representing people whose names come before and after in alphabetical order. This allows for efficient searching and retrieval of phone numbers.

2. **Dictionary:** A dictionary is a collection of words and their definitions arranged in alphabetical order. BSTs can be used to store the words and definitions, with each node representing a word and its left and right subtrees representing words that come before and after in alphabetical order. This allows for efficient searching and retrieval of words and definitions.

3. **Code autocomplete:** Code editors often have an autocomplete feature that suggests code snippets based on the user's input. BSTs can be used to store the code snippets, with each node representing a snippet and its left and right subtrees representing snippets that come before and after in alphabetical order. This allows for efficient searching and retrieval of code snippets.

4. **Stock market analysis:** In stock market analysis, it is often useful to track the performance of individual stocks over time. BSTs can be used to store the stock prices, with each node representing a price and its left and right subtrees representing prices that come before and after in time order. This allows for efficient searching and retrieval of historical stock prices.

5. **Genome sequencing:** In genome sequencing, it is often useful to search for specific patterns of DNA. BSTs can be used to store the DNA sequences, with each node representing a sequence and its left and right subtrees representing sequences that come before and after in alphabetical order. This allows for efficient searching and retrieval of DNA sequences.

Q. Search in a BST [LeetCode 700]

You are given the root of a binary search tree (BST) and an integer val.

Find the node in the BST that the node's value equals val and return the subtree rooted with that node. If such a node does not exist, return null.

Input: root = [4,2,7,1,3], val = 2
Output: [2,1,3]

Code:

```
class Solution {
    public TreeNode searchBST(TreeNode root, int val) {
        if (root == null || root.val == val) return root;
        root = (root.val > val) ? root.left : root.right;
        return searchBST(root, val);
    }
}
```

Explanation:
If root is null or value of root is equal to given value, we return root (which is null if root is null). (case 1)
Otherwise, (case 2)
If value in root is greater than given value, we make root equal to its left child.
If value in root is less than given value, we make root equal to its right child.
We recursively call the function until case 1 is met

Time Complexity: O(log N) in a balanced BST, O(N) in an unbalanced tree.
Space Complexity: O(log N) in a balanced BST, O(N) in an unbalanced tree due to recursive stack usage.

- **Insertion (Leetcode 701)**

Code:

```
import java.util.Scanner;

class Node {
    int data;
    Node left;
    Node right;

    Node(int value) {
        data = value;
        left = null;
        right = null;
    }
}

class BST {
    Node root;

    BST() {
        root = null;
    }

    void insert(int value) {
        Node newNode = new Node(value);
        if (root == null) {
            root = newNode;
            return;
        }
        Node current = root;
        while (true) {
            if (value < current.data) {
                if (current.left == null) {
                    current.left = newNode;
                    return;
                }
                current = current.left;
```

```
            } else {
                if (current.right == null) {
                    current.right = newNode;
                    return;
                }
                current = current.right;
            }
        }
    }

    void preorderTraversal(Node node) {
        if (node == null) {
            return;
        }
        System.out.print(node.data + " ");
        preorderTraversal(node.left);
        preorderTraversal(node.right);
    }
}

public class Main {
    public static void main(String[] args) {
        BST bst = new BST();
        int value;
        char choice;
        Scanner scanner = new Scanner(System.in);
        do {
            System.out.print("Enter a value to insert: ");
            value = scanner.nextInt();
            bst.insert(value);
            System.out.print("Preorder of BST: ");
            bst.preorderTraversal(bst.root);
            System.out.println();
            System.out.print("\nDo you want to insert another
value? (y/n): ");
            choice = scanner.next().charAt(0);
        } while (choice == 'y' || choice == 'Y');
        scanner.close();
    }
}
```

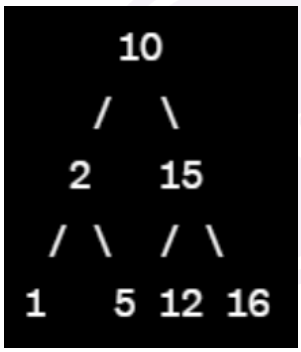
Explanation:

1. The program starts by defining a Node class and a BST class.
2. The Node class represents a node in the BST, and the BST class represents the BST itself.
3. The insert method in the BST class takes a value as input and inserts a new node with that value in the BST.
4. If the BST is empty, the new node becomes the root.
5. Otherwise, the method iterates through the BST starting from the root, following the left or right branch depending on whether the value is smaller or larger than the value of the current node until it finds an empty spot to insert the new node.

Output:

```
Enter a value to insert: 10
Preorder of BST: 10

Do you want to insert another value? (y/n): y
Enter a value to insert: 2
Preorder of BST: 10 2

Do you want to insert another value? (y/n): y
Enter a value to insert: 15
Preorder of BST: 10 2 15

Do you want to insert another value? (y/n): y
Enter a value to insert: 5
Preorder of BST: 10 2 5 15

Do you want to insert another value? (y/n): y
Enter a value to insert: 1
Preorder of BST: 10 2 1 5 15

Do you want to insert another value? (y/n): y
Enter a value to insert: 16
Preorder of BST: 10 2 1 5 15 16

Do you want to insert another value? (y/n): y
Enter a value to insert: 12
Preorder of BST: 10 2 1 5 15 12 16

Do you want to insert another value? (y/n): n
```

BST created:



Time Complexity:

Inserting a node in a BST takes O(log n) time on average, where n is the number of nodes in the tree. This is because in each iteration of the while loop in the insert method, the size of the subtree being searched is halved, leading to a logarithmic time complexity.
If the BST is unbalanced and degenerates into a linked list, the time complexity of inserting a node becomes O(n), where n is the number of nodes in the tree.

Space Complexity:

The space complexity of the program is O(n), where n is the number of nodes in the BST. This is because the program creates a new node object for each value inserted into the BST, and these node objects are stored in memory until the program terminates.

Traversal

Code:

```
class Node {
    int data;
    Node left;
    Node right;

    Node(int value) {
        data = value;
        left = null;
        right = null;
    }
}

class BST {
    Node root;

    BST() {
        root = null;
    }

    Node createNode(int val) {
        Node newNode = new Node(val);
        newNode.left = null;
        newNode.right = null;
        return newNode;
    }

    void insert(int val) {
        Node newNode = createNode(val);
        if (root == null) {
            root = newNode;
        } else {
            Node curr = root;
            while (true) {
                if (val < curr.data) {
                    if (curr.left == null) {
                        curr.left = newNode;
                        break;
                    } else {
                        curr = curr.left;
                    }
                } else {
                    if (curr.right == null) {
                        curr.right = newNode;
                        break;
                    } else {
                        curr = curr.right;
                    }
```

```
                }
            }
        }
    }

    void inOrderTraversal(Node node) {
        if (node == null) {
            return;
        }
        inOrderTraversal(node.left);
        System.out.print(node.data + " ");
        inOrderTraversal(node.right);
    }

    void preOrderTraversal(Node node) {
        if (node == null) {
            return;
        }
        System.out.print(node.data + " ");
        preOrderTraversal(node.left);
        preOrderTraversal(node.right);
    }

    void postOrderTraversal(Node node) {
        if (node == null) {
            return;
        }
        postOrderTraversal(node.left);
        postOrderTraversal(node.right);
        System.out.print(node.data + " ");
    }
}

public class Main {
    public static void main(String[] args) {
        BST bst = new BST();
        bst.insert(5);
        bst.insert(3);
        bst.insert(7);
        bst.insert(2);
        bst.insert(4);
        bst.insert(6);
        bst.insert(8);

        System.out.print("Inorder traversal of the binary search
tree: ");
        bst.inOrderTraversal(bst.root);
        System.out.println();
```

```
        System.out.print("Preorder traversal of the binary search
tree: ");
        bst.preOrderTraversal(bst.root);
        System.out.println();

        System.out.print("Postorder traversal of the binary
search tree: ");
        bst.postOrderTraversal(bst.root);
        System.out.println();
    }
}
```

Explanation: This program creates a BST with some nodes and performs inorder, preorder, and postorder traversals on it. The createNode function creates a new node with the given value. The insert function inserts a new node into the BST in the correct position based on the node's value. The inOrderTraversal, preOrderTraversal, and postOrderTraversal functions perform the corresponding tree traversals.

Inorder Traversal:

1. Traverse the left subtree by calling inOrderTraversal on the left child node.
2. Visit the root node and output its value.
3. Traverse the right subtree by calling inOrderTraversal on the right child node.

Preorder Traversal:

1. Visit the root node and output its value.
2. Traverse the left subtree by calling preOrderTraversal on the left child node.
3. Traverse the right subtree by calling preOrderTraversal on the right child node.

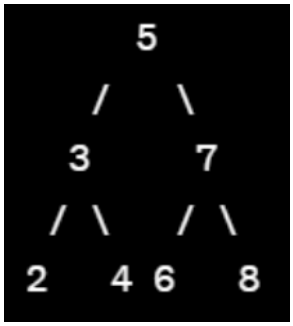
Postorder Traversal:

1. Traverse the left subtree by calling postOrderTraversal on the left child node.
2. Traverse the right subtree by calling postOrderTraversal on the right child node.
3. Visit the root node and output its value.

Output:

```
Inorder traversal of the binary search tree: 2 3 4 5 6 7 8
Preorder traversal of the binary search tree: 5 3 2 4 7 6 8
Postorder traversal of the binary search tree: 2 4 3 6 8 7 5
```

This output shows the inorder, preorder, and postorder traversal of a binary search tree with the following structure:



Time Complexity: The time complexity of the traversal functions inOrderTraversal, preOrderTraversal, and postOrderTraversal is O(n), as each node is visited once.

Space Complexity: The space complexity of the traversal functions is O(h), as it requires space for the recursive function calls on the call stack. In the worst case, when the tree is skewed, the space complexity of the traversal functions is O(n).

Searching

Explanation:

To search in a BST, we follow the following steps:

1. Start at the root node of the BST.
2. Compare the value you are searching for with the value of the current node.
3. If the value is equal to the current node's value, return the node.
4. If the value is less than the current node's value, move to the left child node of the current node (if it exists).
5. If the value is greater than the current node's value, move to the right child node of the current node (if it exists).
6. Repeat steps 2-5 until either the value is found or a null node is reached (indicating that the value is not in the BST).
7. If the value is not found in the BST, return null or a message indicating that the value was not found.

That's it! Searching in a BST is a simple process that takes advantage of the tree's structure and the ordering of its nodes to efficiently locate values.

Recursive Code:

```
class Node {
    int data;
    Node left, right;

    Node(int value) {
        data = value;
        left = right = null;
    }
}

public class Main {

    public static boolean search(Node root, int value) {
        if (root == null) {
            return false;
        } else if (root.data == value) {
            return true;
        } else if (root.data > value) {
            return search(root.left, value);
        } else {
            return search(root.right, value);
```

```
        }
    }

    public static Node createNode(int value) {
        return new Node(value);
    }

    public static Node insert(Node root, int value) {
        if (root == null) {
            return createNode(value);
        } else if (value < root.data) {
            root.left = insert(root.left, value);
        } else {
            root.right = insert(root.right, value);
        }
        return root;
    }

    public static void main(String[] args) {
        Node root = null;
        int value;

        // get values from user and insert in BST
        System.out.print("Enter values to insert in BST (-1 to
stop): ");
        Scanner scanner = new Scanner(System.in);
        value = scanner.nextInt();
        while (value != -1) {
            root = insert(root, value);
            value = scanner.nextInt();
        }

        // search for a value in BST
        System.out.print("Enter a value to search in BST: ");
        value = scanner.nextInt();
        if (search(root, value)) {
            System.out.println("Value " + value + " is present in
BST.");
        } else {
            System.out.println("Value " + value + " is not
present in BST.");
        }

        scanner.close();
    }
}
```

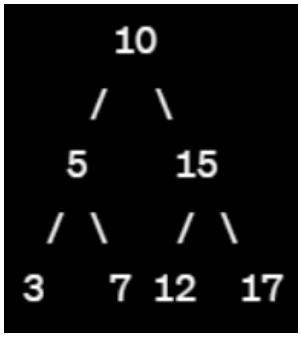
Recursive Code Explanation: In this program, we first create a BST by getting values from the user and inserting them one by one. Then we ask the user to enter a value and check if it's present in the BST or not using the search function recursively. If the value is present, we print a message saying so, otherwise we print a message saying the value is not present.

Output:

```
Enter values to insert in BST (-1 to stop): 10 15 5 12 7 17 3 -1
Enter a value to search in BST: 12
Value 12 is present in BST.
```

```
Enter values to insert in BST (-1 to stop): 10 15 5 12 7 17 3 -1
Enter a value to search in BST: 99
Value 99 is not present in BST.
```

BST Diagram:



Time Complexity: The time complexity of this program is O(log n) in the average case and O(n) in the worst case i.e. when the tree is skewed, where n is the number of nodes in the BST.

Space Complexity: The space complexity is O(h), where h is the height of the BST and this is because of the call stack created due to recursive function calls.

Iterative Code:

```
class Node {
    int data;
    Node left, right;

    Node(int value) {
        data = value;
        left = right = null;
    }
}

public class Main {

    public static boolean search(Node root, int value) {
        Node current = root;
        while (current ≠ null) {
            if (current.data == value) {
                return true;
            } else if (current.data > value) {
                current = current.left;
            } else {
```

```
                current = current.right;
            }
        }
        return false;
    }

    public static Node createNode(int value) {
        return new Node(value);
    }

    public static Node insert(Node root, int value) {
        if (root == null) {
            return createNode(value);
        } else if (value < root.data) {
            root.left = insert(root.left, value);
        } else {
            root.right = insert(root.right, value);
        }
        return root;
    }

    public static void main(String[] args) {
        Node root = null;
        int value;

        // get values from user and insert in BST
        System.out.print("Enter values to insert in BST (-1 to
stop): ");
        Scanner scanner = new Scanner(System.in);
        value = scanner.nextInt();
        while (value != -1) {
            root = insert(root, value);
            value = scanner.nextInt();
        }

        // search for a value in BST iteratively
        System.out.print("Enter a value to search in BST: ");
        value = scanner.nextInt();
        if (search(root, value)) {
            System.out.println("Value " + value + " is present in
BST.");
        } else {
            System.out.println("Value " + value + " is not
present in BST.");
        }

        scanner.close();
    }
}
```

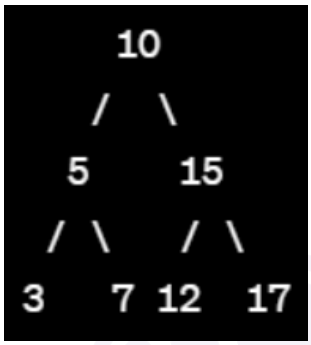
Iterative Code Explanation: In this program, we first create a BST by getting values from the user and inserting them one by one. Then we ask the user to enter a value and check if it's present in the BST or not using the search function which is implemented iteratively. If the value is present, we print a message saying so, otherwise we print a message saying the value is not present.

Output:

```
Enter values to insert in BST (-1 to stop): 10 15 5 12 7 17 3 -1
Enter a value to search in BST: 7
Value 7 is present in BST.
```

```
Enter values to insert in BST (-1 to stop): 10 5 15 7 12 17 3 -1
Enter a value to search in BST: 1000
Value 1000 is not present in BST.
```

BST Diagram:



Time Complexity: The time complexity of this program is O(log n) in the average case and O(n) in the worst case i.e. when the tree is skewed, where n is the number of nodes in the BST.

Space Complexity: The space required by the iterative implementation does not depend on the height of the tree, but only on the constant number of variables used in the implementation. Therefore, the space complexity of the iterative implementation of searching in a BST is O(1).

MCQ Questions

Q. Consider a binary search tree with n nodes. What is the maximum possible height of the tree?

A) O(n)
B) O(log n)
C) O(n log n)
D) O(sqrt(n))
Answer: A) O(n)

Explanation: The maximum possible height of a binary search tree occurs when the tree is a linked list, with all nodes connected in a straight line. In this case, the height of the tree would be n - 1. Therefore, the maximum possible height of the tree is O(n).

Q. Consider a binary search tree with n nodes. What is the minimum number of comparisons required to search for a value in the worst-case scenario?

A) O(1)
B) O(log n)
C) O(n log n)
D) O(n)
Answer: B) O(log n)

Explanation: In the worst-case scenario, the value being searched for is either the smallest or the largest value in the tree, and it is located at the bottom-most level of the tree. In a balanced BST, the height of the tree is O(log n), so the worst-case scenario requires O(log n) comparisons to reach the bottom-most level of the tree. Therefore, the minimum number of comparisons required to search for a value in the worst-case scenario is O(log n).

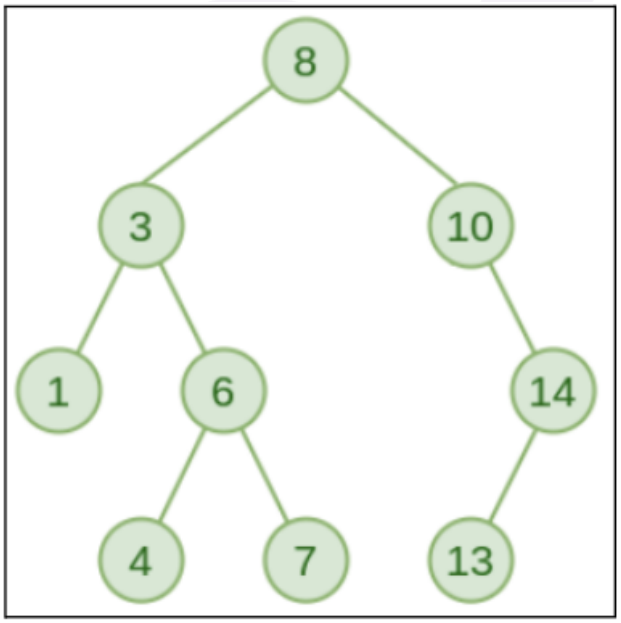
Coding Questions

Q. Lowest Common Ancestor of BST (Leetcode 235)

Given a Binary Search Tree (BST) and two values. You need to find the LCA i.e. Lowest common ancestor of the two nodes provided both the nodes exist in the BST. [easy]

Example:

Consider the following BST:



Input-1:
n = 9
values = [8, 3, 1, 6, 4, 7, 10, 14,13]
node-1 = 3
node-2 = 13
Output-1:
Lowest Common Ancestor = 8

Input-2:
n = 9
values = [8, 3, 1, 6, 4, 7, 10, 14,13]
node-1 = 14
node-2 = 13
Output-2:
Lowest Common Ancestor = 14

Recursive Approach:

- We are going to create a recursive function that takes a node and the two values n1 and n2.
- If the value of the current node is less than both n1 and n2, then LCA lies in the right subtree. Call the recursive function for the right subtree.
- If the value of the current node is greater than both n1 and n2, then LCA lies in the left subtree. Call the recursive function for the left subtree.
- If both the above cases are false then return the current node as LCA.

Solution Code:

```
class Node {
    int data;
    Node left;
    Node right;

    Node(int value) {
        data = value;
        left = null;
        right = null;
    }
}

public class Main {
    public Node lca(Node root, int n1, int n2) {
        if (root == null)
            return null;

        // If both n1 and n2 are smaller than root
        // LCA lies in left
        if (root.data > n1 && root.data > n2)
            return lca(root.left, n1, n2);

        // If both n1 and n2 are greater than root
        // LCA lies in right
        if (root.data < n1 && root.data < n2)
            return lca(root.right, n1, n2);

        return root;
    }
```

```
    public static void main(String[] args) {
        // Your main code logic can go here if needed
    }
}
```

Iterative Approach:

1. Start from the root node of the BST.
2. While the root node is not null:

- If both nodes have values greater than the value of the root node, then move to the right child of the root node.
- If both nodes have values less than the value of the root node, then move to the left child of the root node.
- If one node has a value greater than the value of the root node and the other node has a value less than the value of the root node, then the root node is the LCA of the two nodes.

3. If either of the two nodes being searched for is equal to the value of the current root node, then the current root node is the LCA of the two nodes.
4. If neither of the above conditions is met, then update the current root node to be the child node in the appropriate direction and repeat the above steps until the LCA is found.
5. Return the LCA as the output of the function.

Solution Code:

```
class Node {
    int data;
    Node left;
    Node right;

    Node(int value) {
        data = value;
        left = null;
        right = null;
    }
}

public class Main {
    public Node lca(Node root, int n1, int n2) {
        while (root ≠ null) {
            // If both n1 and n2 are smaller than root
            // LCA lies in left
            if (root.data > n1 && root.data > n2)
                root = root.left;
            // If both n1 and n2 are greater than root
            // LCA lies in right
            else if (root.data < n1 && root.data < n2)
                root = root.right;
            else
                break;
```

```
        }
        return root;
    }

    public static void main(String[] args) {
        // Your main code logic can go here if needed
    }
}
```

Time complexity: O(H) where H is the height of the tree. As we are starting to check for LCA from the root till we find the given nodes.

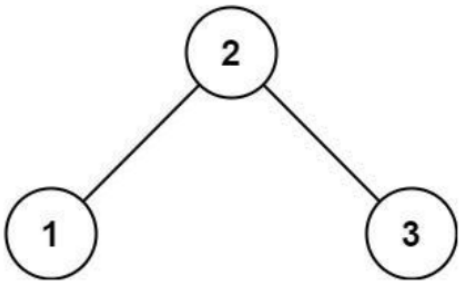
Space complexity: O(H) since we need a recursive stack of size H in case of recursive solution.
However, in the case of an iterative solution, the space complexity is O(1), constant, because no call stack is created.

Q. Validate BST (Leetcode 98)

Given the root of a binary tree, determine if it is a valid binary search tree (BST).

A valid BST is defined as follows:

The left subtree of a node contains only nodes with keys less than the node's key.
The right subtree of a node contains only nodes with keys greater than the node's key.
Both the left and right subtrees must also be binary search trees.



Input: root = [2,1,3]
Output: true

Code:

```
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;

    TreeNode(int value) {
        val = value;
        left = null;
        right = null;
    }
}
```

```
public class Solution {
    private Integer curr = null;

    public boolean isValidBST(TreeNode root) {
        return dfs(root);
    }

    private boolean dfs(TreeNode head) {
        if (head.left ≠ null && !dfs(head.left)) return false;
        if (curr ≠ null && curr ⩾ head.val) return false;
        curr = head.val;
        if (head.right ≠ null && !dfs(head.right)) return false;
        return true;
    }
}
```

Explanation: The DFS algorithm need to make sure the value of * curr is in strictly increasing order.
If it doesn't satisfy the strictly increasing order then return false

Time Complexity: O(n) - where n is the number of nodes in the tree, as it traverses each node once.

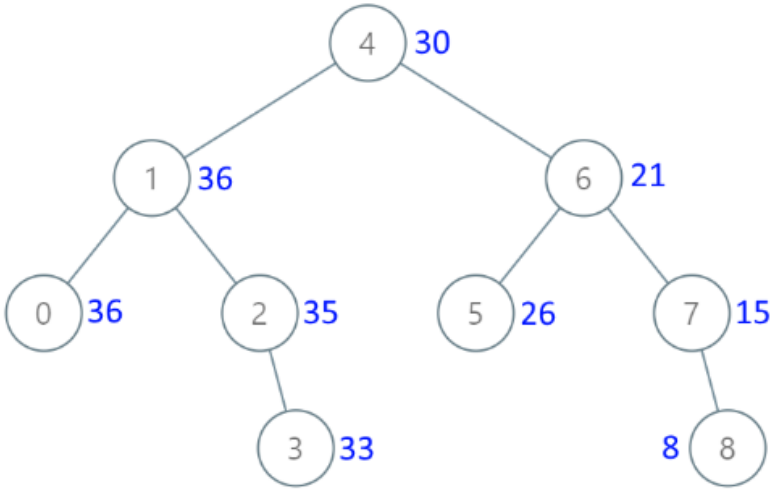
Space Complexity: O(h) - where h is the height of the tree, due to recursive stack space. In the worst case (unbalanced tree), it can be O(n), and for balanced trees, O(log n).

Q. Given a BST, transform it into a greater sum tree where each node contains the sum of all nodes greater than that node. [Leetcode 1038]

Given the root of a Binary Search Tree (BST), convert it to a Greater Tree such that every key of the original BST is changed to the original key plus the sum of all keys greater than the original key in BST.

As a reminder, a binary search tree is a tree that satisfies these constraints:

The left subtree of a node contains only nodes with keys less than the node's key.
The right subtree of a node contains only nodes with keys greater than the node's key.
Both the left and right subtrees must also be binary search trees.



Input: root = [4,1,6,0,2,5,7,null,null,null,3,null,null,null,8]
Output: [30,36,21,36,35,26,15,null,null,null,33,null,null,null,8]

Code:

```
public class Solution {
    private int currSum = 0;

    public TreeNode bstToGst(TreeNode root) {
        travelSum(root);
        return root;
    }

    private void travelSum(TreeNode root) {
        if (root == null)
            return;
        if (root.right != null)
            travelSum(root.right);
        currSum += root.val;
        root.val = currSum;
        if (root.left != null)
            travelSum(root.left);
    }
}
```

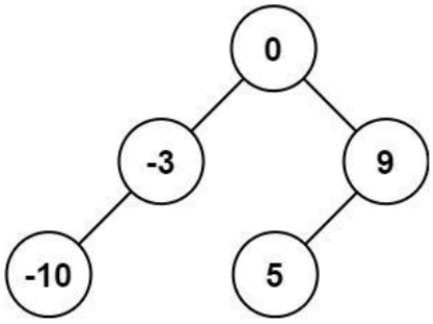
Explanation:
1. Go deep to the right most lead
2. Return back maintain a count variable
3. Assign the count to current root->val
4. Recur for left if not NULL

Time Complexity: O(n) - where n is the number of nodes in the tree. The function visits each node once in a single traversal.

Space Complexity: O(h) - where h is the height of the tree. The space used in the call stack due to recursive calls is proportional to the height of the tree. In the worst-case scenario of an unbalanced tree, space complexity can reach O(n), but for a balanced tree, it typically remains O(log n).

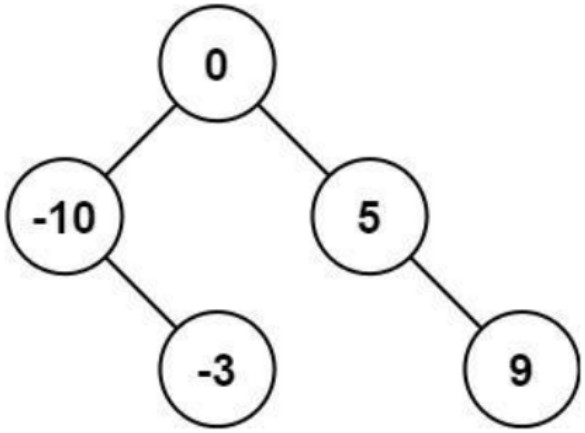
Q. Converted Sorted Array to Balanced BST [LeetCode 108]

Given an integer array nums where the elements are sorted in ascending order, convert it to a height-balanced binary search tree.



Input: nums = [-10,-3,0,5,9]
Output: [0,-3,9,-10,null,5]

Explanation: [0,-10,5,null,-3,null,9] is also accepted:



Code:

```
public class Solution {
    public TreeNode sortedArrayToBST(int[] nums) {
        if (nums.length == 0) return null;
        if (nums.length == 1) {
            return new TreeNode(nums[0]);
        }

        int middle = nums.length / 2;
        TreeNode root = new TreeNode(nums[middle]);

        int[] leftInts = Arrays.copyOfRange(nums, 0, middle);
        int[] rightInts = Arrays.copyOfRange(nums, middle + 1,
nums.length);

        root.left = sortedArrayToBST(leftInts);
        root.right = sortedArrayToBST(rightInts);

        return root;
    }
}
```

Explanation: Recursively call the sortedArrayToBST() method providing new vector for each call to construct left and right children:

Time Complexity: O(n) - Visits each element once to construct a balanced BST recursively.

Space Complexity: O(n) - Due to recursive call stack and extra space for left and right vectors.

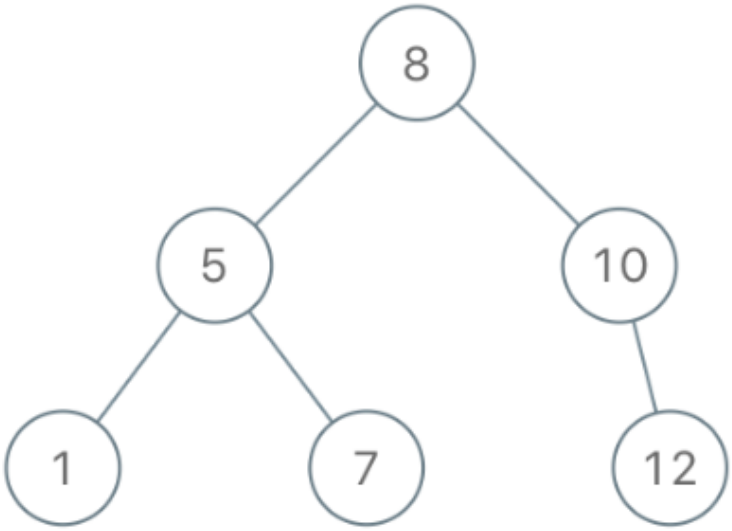
Q. Given the preorder traversal of a binary search tree, construct the BST. [Leetcode 1008]

Given an array of integers preorder, which represents the preorder traversal of a BST (i.e., binary search tree), construct the tree and return its root.

It is guaranteed that there is always possible to find a binary search tree with the given requirements for the given test cases.

A binary search tree is a binary tree where for every node, any descendant of Node.left has a value strictly less than Node.val, and any descendant of Node.right has a value strictly greater than Node.val.

A preorder traversal of a binary tree displays the value of the node first, then traverses Node.left, then traverses Node.right.



Input: preorder = [8,5,1,7,10,12]
Output: [8,5,10,1,7,null,12]

Code:

```
public class Solution {
    int idx = 0;

    public TreeNode bstFromPreorder(int[] preorder) {
        return bstFromPreorder(preorder, Integer.MAX_VALUE);
    }

    private TreeNode bstFromPreorder(int[] preorder, int pVal) {
        if (idx >= preorder.length || preorder[idx] > pVal)
            return null;
        TreeNode n = new TreeNode(preorder[idx++]);
        n.left = bstFromPreorder(preorder, n.val);
        n.right = bstFromPreorder(preorder, pVal);
        return n;
    }
}
```

Explanation: we are searching for a split point to divide the interval. Instead, we can pass the parent value to the recursive function to generate the left sub-tree. The generation will stop when the value in the preorder array exceeds the parent value. That will be our split point to start generating the right subtree.

Time Complexity: O(n) - Traverses each element in the `preorder` vector to construct the BST.

Space Complexity: O(h) - Due to recursive calls, occupying space in the call stack proportional to the tree's height.

PW SKILLS

THANK YOU !

PW SKILLS