Lesson:

BST-2

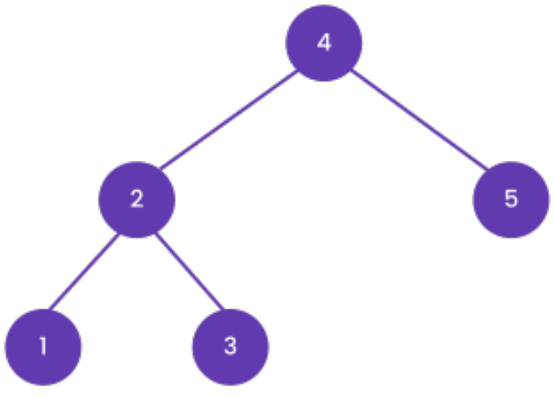
PW SKILLS

Q1. Binary Tree is BST or not

Input: Given a binary search tree (BST) i.e. a node-based binary tree data structure that has the following properties:

- 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.
- Each node (item in the tree) has a distinct key.

Given Tree:



Output:

```
The binary tree is a BST.
```

Solution Code: https://pastebin.com/E1SqaSMA

Explanation:

- We are following the given approach:
- If the current node is null then return true
- If the value of the left child of the node is greater than or equal to the current node then return false
- If the value of the right child of the node is less than or equal to the current node then return false
- If the left subtree or the right subtree is not a BST then return false
- Else return true

Time complexity: O(n), where n is the number of nodes in the binary tree. This is because the code traverses each node exactly once in a depth-first manner. In the worst case, when the binary tree is a valid BST, every node needs to be visited to validate the BST property.

Space complexity: O(h), where h is the height of the binary tree. This is because the code utilizes the call stack during the recursive function calls, which depends on the height of the tree. In the worst case, when the tree is skewed, the height is equal to the number of nodes, resulting in O(n) space complexity. However, in a balanced binary tree, the height is logarithmic to the number of nodes, resulting in O(log n) space complexity.

Q2. BSTs are identical

Input:

Given two vectors that represent a sequence of keys. Imagine we make a Binary Search Tree (BST) from each

array. We need to tell whether two BSTs will be identical or not without actually constructing the tree.
arr1 = {4, 2, 5, 1, 3}
arr2 = {4, 5, 2, 3, 1}

Output:

```
BSTs are identical.
```

Solution Code: https://pastebin.com/43uWGsqC

Explanation:

- According to the BST property, elements of the left subtree must be smaller and elements of right subtree must be greater than root.
- Two arrays represent the same BST if, for every element x, the elements in left and right subtrees of x appear after it in both arrays. And same is true for roots of left and right subtrees.
- The idea is to check of if next smaller and greater elements are same in both arrays. Same properties are recursively checked for left and right subtrees. The idea looks simple, but implementation requires checking all conditions for all elements.

Time complexity: O(n^2), where n is the number of elements in the arrays representing the BSTs. This is because for each element in one array, we search for a corresponding element in the other array, resulting in a nested loop structure. In the worst case, we may need to iterate over all elements in both arrays.

Space complexity: O(1) because we are not using any additional data structures that grow with the input size. We only use a few integer variables to keep track of indices and values during the recursive calls. Thus, the space required remains constant regardless of the input size.

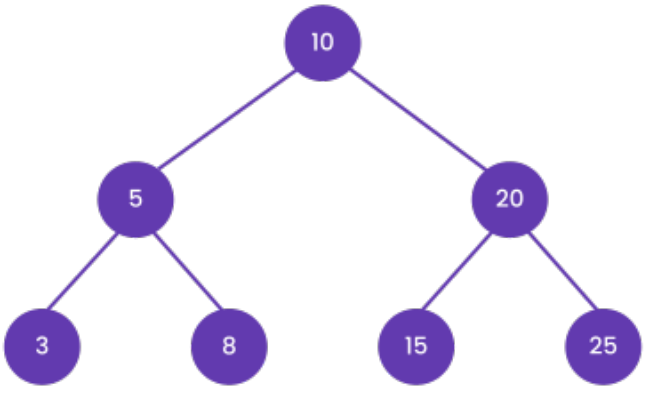
Q3. Construct BST from preorder

Input: Given the preorder traversal of a binary search tree, construct the BST.

Given Preorder: 10, 5, 3, 8, 20, 15, 25

Output:

```
Inorder traversal of the constructed BST: 3 5 8 10 15 20 25
```



Solution Code: https://pastebin.com/qpmF1TDG

Explanation:

NOTE : For building a tree any two traversals must be known. In case of BST, preorder is given and we know inorder traversal of bst is just the sorted preorder array.

- The first element of preorder traversal is always the root.
- We first construct the root.
- Then we find the index of the first element which is greater than the root.
- Let the index be 'i'.
- The values between root and 'i' will be part of the left subtree, and the values between 'i'(inclusive) and 'n-1' will be part of the right subtree.
- Divide the given pre[] at index "i" and recur for left and right sub-trees.

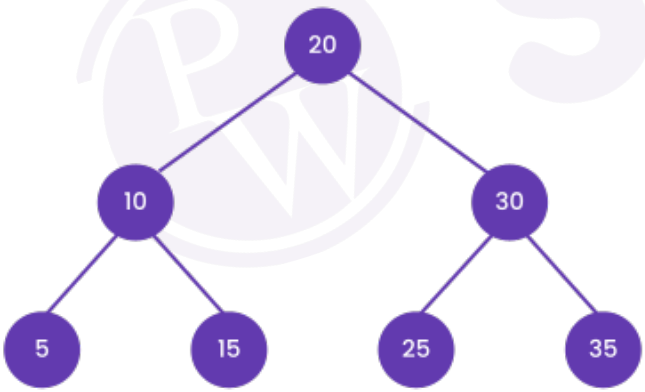
Time complexity: O(n), where n is the number of elements in the traversal. This is because we visit each element once and perform constant-time operations for each node.

Space complexity: O(n) as well, considering the space required for the recursive function calls on the call stack. In the worst-case scenario, the function calls will be nested to the depth of the tree, which is proportional to the number of nodes, resulting in O(n) space complexity. Additionally, the space complexity also includes the space required to store the BST itself, which is also O(n) in the worst case.

Q4. Shortest distance between nodes in BST

Input: Given a Binary Search Tree with unique value of nodes and two keys in it. Find the distance between two nodes with given two keys. It may be assumed that both keys exist in BST.

Given BST:



Output:

```
The shortest distance between 5 and 30: 3
```

Solution Code: https://pastebin.com/3GUxb61F

Explanation:

- If both keys are greater than the current node, we move to the right child of the current node.
- If both keys are smaller than current node, we move to left child of current node.
- If one keys is smaller and other key is greater, current node is Lowest Common Ancestor (LCA) of two nodes. We find distances of current node from two keys and return sum of the distances.

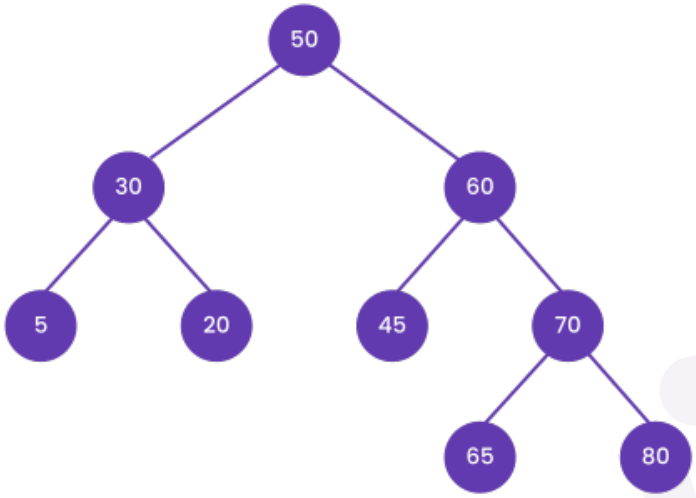
Time complexity: O(h) where h is the height of the Binary Search Tree.

Space complexity: O(h) as well, since we need a recursive stack of size h.

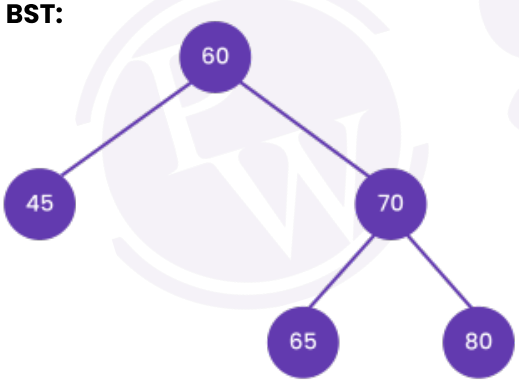
Q5. Largest BST in Binary Tree

Input: Given a Binary Tree, write a function that returns the size of the largest subtree which is also a Binary Search Tree (BST). If the complete Binary Tree is BST, then return the size of the whole tree.

Given Binary Tree:



Output:
BST:




```
Largest BST Subtree size: 5
```

Solution Code: https://pastebin.com/81daU43i

Explanation:
Here, we don't need to check explicitly if the binary tree is BST. A Tree is BST if the following is true for every node x:

1. The largest value in the left subtree (of x) is smaller than the value of x.
2. The smallest value in the right subtree (of x) is greater than the value of x.

So, we will just check if the largest value of the left subtree is less than the value of the root node and the

smallest value of right subtree is greater than the value of root node.

Time complexity: O(N), where N is the number of nodes in the binary tree. This is because we visit each node once during the recursive traversal to determine the largest BST subtree.

Space complexity: O(N) as well. This is due to the recursive nature of the algorithm, which involves maintaining recursive function calls on the call stack. In the worst case scenario, where the binary tree is skewed, the maximum space required on the call stack is proportional to the number of nodes in the tree.

Q6. Construct all possible BST for 1 to N

Input: Write a C++ program to construct all unique binary search trees (BSTs) for keys ranging from 1 to N. The program should prompt the user to enter the value of N, and then construct and display all possible BSTs for the given range of keys.

Given N = 4
Output:

```
Preorder traversals of all constructed BSTs are:
1 2 3 4
1 2 4 3
1 3 2 4
1 4 2 3
1 4 3 2
2 1 3 4
2 1 4 3
3 1 2 4
3 2 1 4
4 1 2 3
4 1 3 2
4 2 1 3
4 3 1 2
4 3 2 1
```

Solution Code: https://pastebin.com/VxE606GV

Explanation:

- Initialize list of BSTs as empty.
- For every number i where i varies from 1 to N, do following
  - Create a new node with key as 'i', let this node be 'node'
  - Recursively construct a list of all left subtrees.
  - Recursively construct a list of all right subtrees..
- Iterate for all left subtrees
  - For current left subtree, iterate for all right subtrees
  - Add current left and right subtrees to 'node' and add.
  - node' to list

Time complexity: O(2^n) This is because we are constructing all possible BSTs. Since at each level, we are generating two subproblems and at each level, there are O(2^n) subproblems, the time complexity is O(2^n).

Space complexity: O(2^n) The space complexity of this solution is also exponential. This is because we are constructing all possible BSTs. Since at each level, we are generating two subproblems and at each level, there are O(2^n) subproblems, the space complexity is O(2^n).

Q7. Replace with greater element on right

Input: Given an array of integers, replace every element with the least greater element on its right side in the array. If there are no greater elements on the right side, replace it with -1.

Output:

```
Original array:
20 16 32 21 93 54 22

Array after replacement:
21 21 54 22 -1 -1 -1
```

Solution Code: https://pastebin.com/SeB6nqA1

Explanation:
The approach used in the solution can be summarized in the following steps:

- 1Start from the rightmost element of the array and initialize an empty Binary Search Tree (BST).
- Iterate through the array from right to left and insert each element into the BST.
- During the insertion process, keep track of the successor for each element (the least greater element on its right side).
- If a successor is found, replace the current element with the successor value.
- If no successor is found (reached the end of the BST), replace the current element with -1.
- Repeat steps 2-5 for all elements in the array.
- Print the updated array after all replacements have been made.

This approach ensures that for each element, we find the least greater element on its right side by utilizing the properties of the Binary Search Tree.
The **time complexity** is O(nlogn) because we perform insertions and lookups in the Binary Search Tree (BST) for each element in the array.
The **space complexity** is O(n) because we store the elements in the BST, which can have up to n nodes in the worst case.