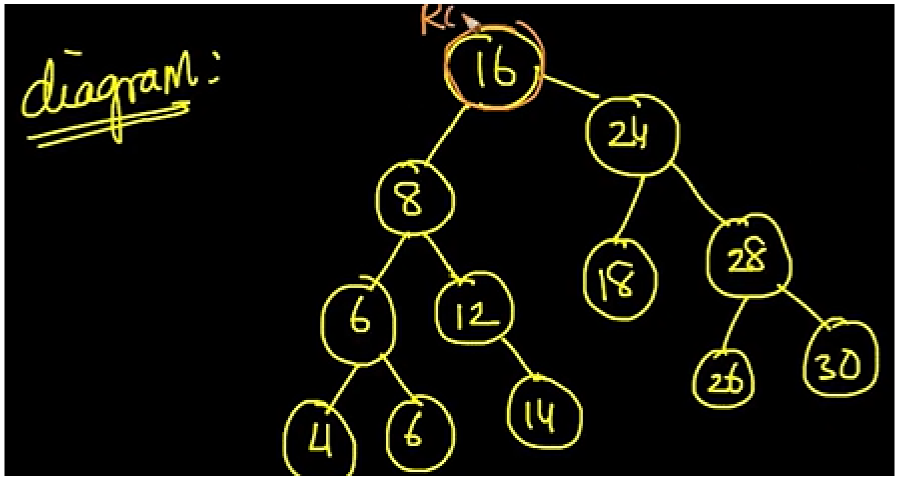
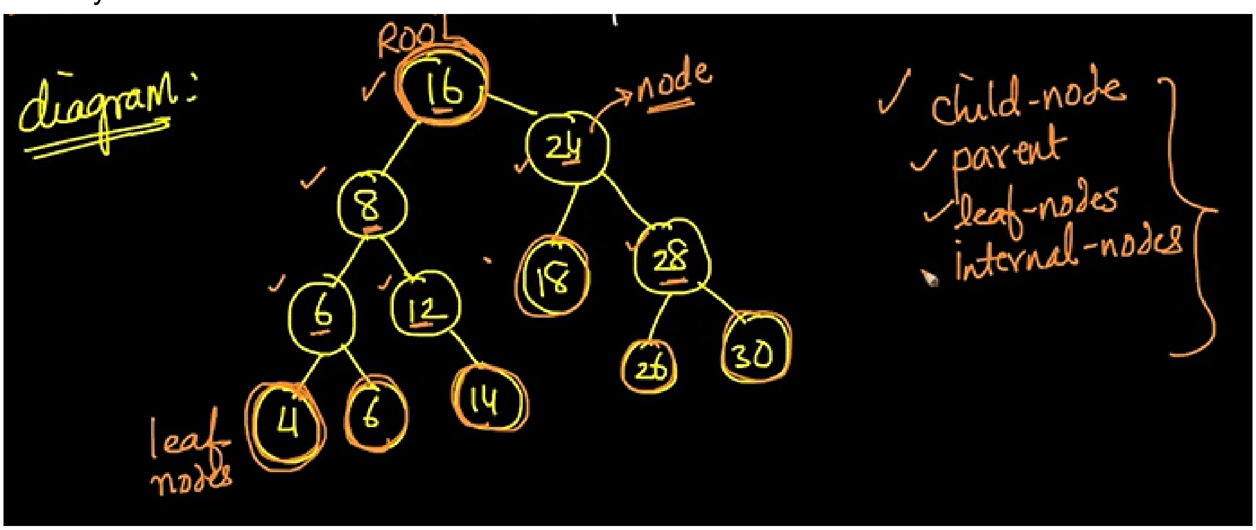
Learning BST

1. BST is an old algo which is dersince 1960's
2. It a collection of items …which stores the data in tree structure
3. We'll learn some operations on BST and also searching an ele in BST takes O(log n) in avg case



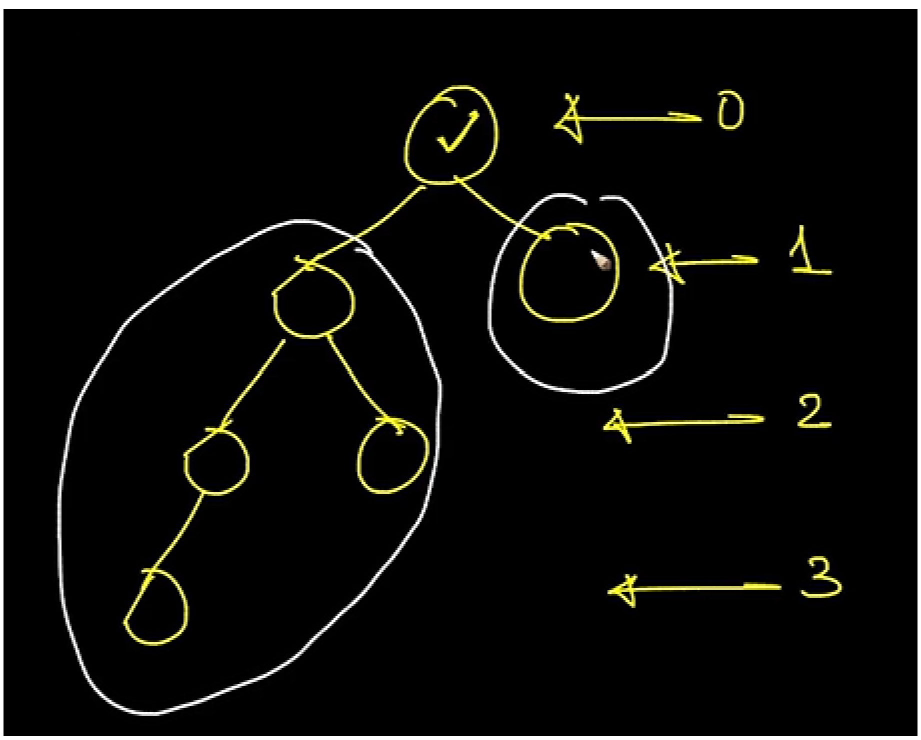
4. Structure of BST
5. Here each circle is called a node..and 16 is the root node
6. And here 16 is the parent of 8 and 24..similarly 8 is the parent of 6 and 12
7. If node does not have any child nodes ..then they are leaf nodes
8. And any node other than leaf nodes are called as internal nodes



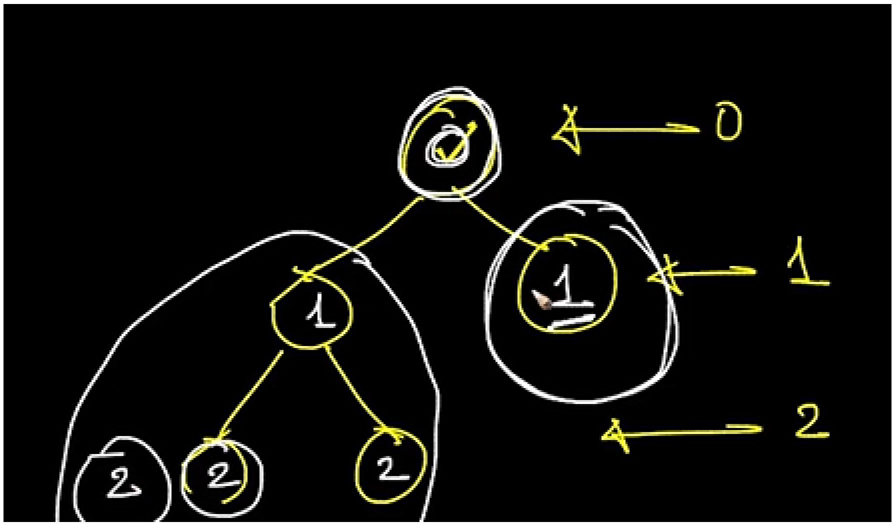
9. Next we have a concept of subtrees..for example here left subtree of 16 is with root 8…and right subtree of 16 is root 24

10. 

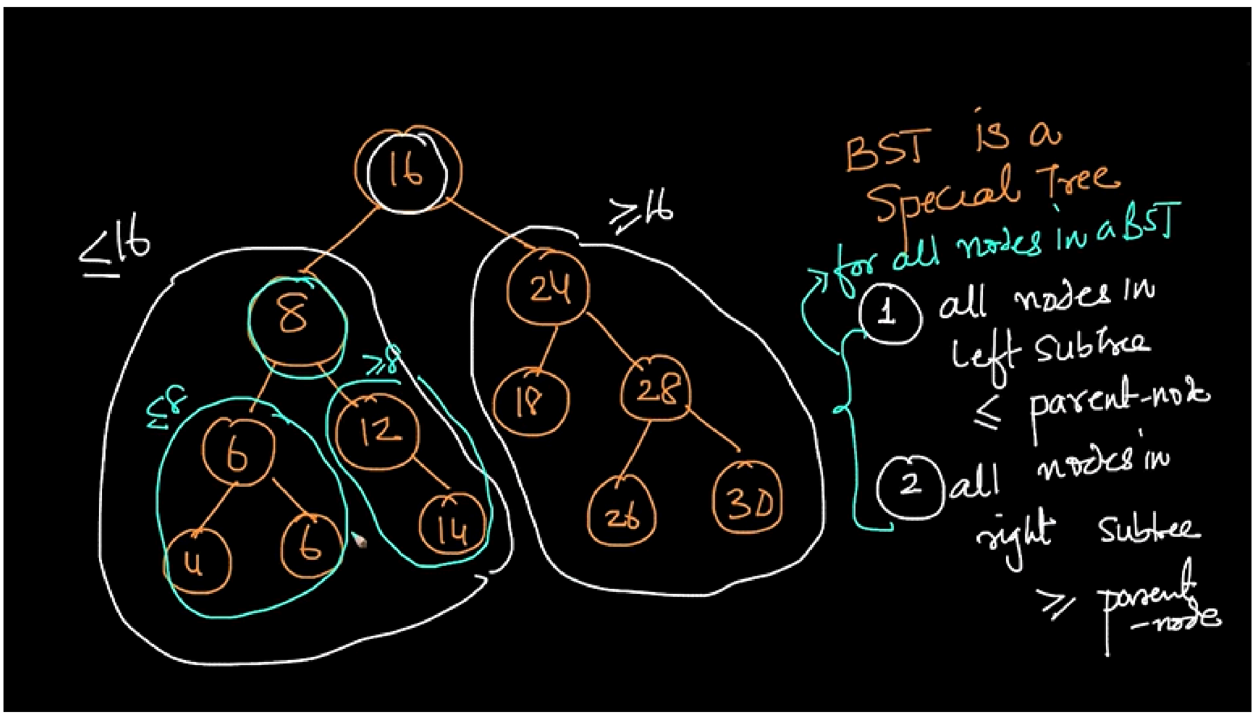
11. Here 16 is at depth 0 and 8,24 are at depth1..check pic
12. Depth of tree = max depth of any node
13. And each circle is called a node or vertex and line between the nodes are called as edges
14. Balanced Tree
15. If depth of RST and LST does not differ greater than 1..then it is a balanced tree
16. Examples of balanced tree



17. Here the depth of LST is 2 and depth of RST is 0…then LST-RST = 2 > 1…it is an unbalanced tree

18. 

19. Here LST-RST = 1…it is a balanced tree
20. BST is a special tree



Solving a Problem on leetcode

96. Unique Binary Search Trees

1. We'll solve this problem using recursion first

2. Using this approach..we get TLE error..as the time complexity is
    - Time Complexity: The time complexity of this recursive approach is exponential, specifically O(4^n / (n^(3/2))). This is due to the large number of recursive calls and overlapping subproblems.
    - Space Complexity: The space complexity is O(n), where n is the depth of the recursion.

```python
class Solution:
    def numTrees(self, n: int) -> int:
        if n <= 1:
            return 1

        ans = 0
        for i in range(1, n + 1):
            ans += self.numTrees(i - 1) * self.numTrees(n - i)
        return ans
```

3. 💡 **Approach 2: Top-Down Approach (Memoization)**

4. Code : here in this code..we used memoization dp…and used memo as cache array

```python
class Solution:
    def numTrees(self, n: int) -> int:
        memo = [-1] * (n+1)
        return self.solve(n,memo)

    def solve(self,n,memo):
        if n<=1:
            return 1
        if memo[n] != -1:
            return memo[n]
        ans = 0
        for i in range(1,n+1): #if i as root
            ans += self.solve(i-1,memo) * self.solve(n-i,memo)
        memo[n] = ans
        return ans
```

5. ✨ **Explanation**

The top-down approach (also known as memoization) is a recursive approach with caching to avoid redundant calculations. We'll define a recursive function `solve(n)` that returns the number of unique BSTs that can be formed with `n` nodes.

The function `solve(n)` can be defined as follows:

1. If `n` is less than or equal to 1, return 1 because there is one unique BST for `n` equal to 0 or 1.
2. Initialize a variable `ans` to 0.
3. Iterate from `1` to `n`, and for each `i`, do the following:
   - Calculate `solve(i - 1)`, which represents the number of unique BSTs in the left subtree with `i-1` nodes.
   - Calculate `solve(n - i)`, which represents the number of unique BSTs in the right subtree with `n-i` nodes.
   - Multiply these two values to find the total number of unique BSTs with the current root `i`.
   - Add this result to `ans`.
4. Return `ans`, which will be the total number of unique BSTs with `n` nodes.

This approach ensures that we don't calculate the same subproblems multiple times by caching the results of subproblems in a memoization table.

6. Dry run :

Dry Run

Let's dry run this approach with an example where `n = 3`:

1. We call the function `solve(3)`.
2. Since `n` is not less than or equal to 1, we proceed with the calculations.
   - For `i = 1`:
     - `solve(0)` returns 1 (there is one unique BST with 0 nodes).
     - `solve(2)` returns 2 (we calculate it as follows):
       - For `i = 1`:
         - `solve(0)` returns 1 (there is one unique BST with 0 nodes).
         - `solve(1)` returns 1 (there is one unique BST with 1 node).
         - We multiply these two values to get 1.
       - For `i = 2`:
         - `solve(1)` returns 1 (there is one unique BST with 1 node).
         - `solve(0)` returns 1 (there is one unique BST with 0 nodes).
         - We multiply these two values to get 1.
       - We add these two results: 1 + 1 = 2.
     - We add the results for `i = 1` and `i = 2`: 1 + 2 = 3.
   - For `i = 2`:

- For `i = 2`:
    - `solve(1)` returns 1 (there is one unique BST with 1 node).
    - `solve(1)` returns 1 (there is one unique BST with 1 node).
    - We add these two results: 1 + 1 = 2.
- We add the results for `i = 1` and `i = 2`: 3 + 2 = 5.

3. The final answer is `5`.

The total number of unique BSTs for `n = 3` is 5.

🔍 Edge Cases

1. If `n` is 0, there is one unique BST (an empty tree).
2. If `n` is 1, there is one unique BST (a single node).

🕸️ Complexity Analysis

- Time Complexity: The time complexity of this top-down approach with memoization is O(n^2) due to overlapping subproblems.
- Space Complexity: The space complexity is O(n) to store the results of subproblems in a memoization table.