Lesson:

Binary Tree Interview Problems

PW SKILLS

Pre-Requisites

- Recursion In java
- Trees in java

Topics to be covered

- Diameter of Tree
- Symmetric tree
- Max path sum in a tree
- Least Common ancestor in a tree

Interview problem: Diameter of Tree

- The diameter of a binary tree can be defined as the number of edges between the longest paths connecting any two nodes in a binary tree. The diameter of the binary tree is also known as the width of the binary tree. The path represents the diameter of the binary tree may or may not pass through the root of the binary tree. The path includes two of the leaf nodes, among which the diameter is getting calculated.
- There can be two possibilities for the longest path between two nodes representing the diameter of the binary tree:

1. **Via Root Node:** It will pass through the root node of the binary tree also counts the root node.
2. **Not via a Root Node:** In this case, the chosen path will not pass through the root node of the binary tree and will not count the root node in the path.

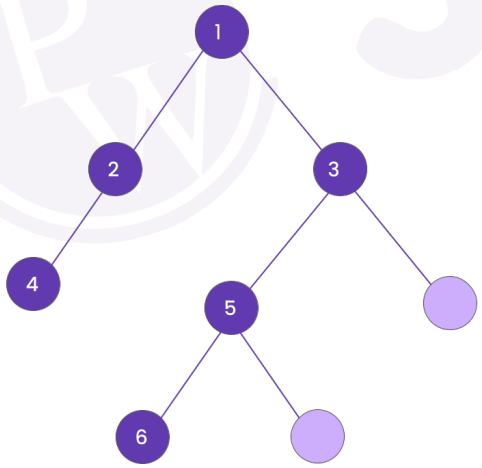


Figure 1

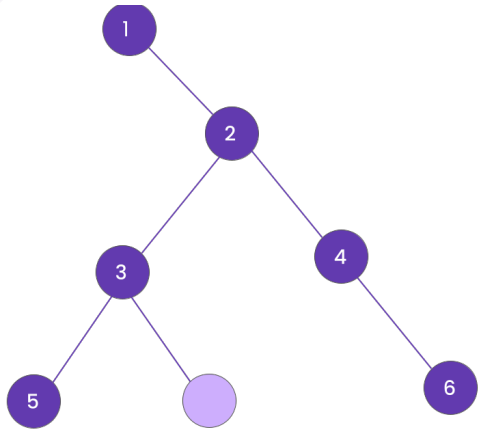


Figure 2

In the above figure 1, the longest path is between leaf node 4 and leaf node 6 that will pass through the root node 1. The diameter of the binary tree shown in figure 1 is 6, starting from leaf node 4 to leaf node 6 i.e. node 4 - node 2 - node 1 - node 3 - node 5 - node 6 also covering the root node.

Whereas in the binary tree shown in figure 2, the longest path for getting the diameter of the binary tree is starting from the leaf node 5 to the leaf node 6, but not including the root node 1. The diameter of this binary tree is 5 following the path node 5 - node 3 - node 2 - node 4 - node 6 excluding the root node 1.

Approach to solve this problem

A simple solution would be to calculate the left and right subtree's height for each node in the tree. The maximum node path that passes through a node will have a value one more than the sum of the height of its left and right subtree. Finally, the diameter is maximum among all maximum node paths for every node in the tree.
The time complexity of this solution is O(n2) as there are n nodes in the tree, and for every node, we are calculating the height of its left and right subtree that takes O(n) time.

We can solve this problem in linear time by doing a **postorder traversal** on the tree. Instead of calculating the height of the left and the right subtree for every node in the tree, get the height in constant time.
The idea is to start from the bottom of the tree and return the height of the subtree rooted at a given node to its parent. The height of a subtree rooted at any node is one more than the maximum height of the left or right subtree.

We will be using Atomic Integer in the code so here is the reference for the same.

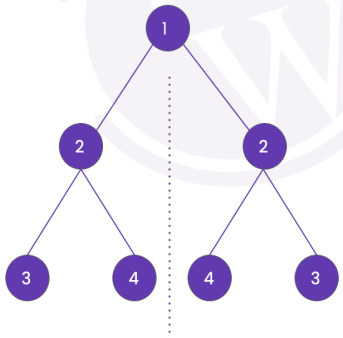
LP_Code1.java

Output:

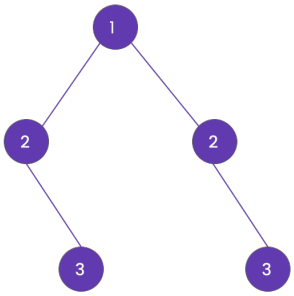
```
The diameter of the tree is 6
```

Interview Problem: Symmetric Tree

A binary tree is said to be symmetric if its mirror image of the left subtree around center is the same as the right subtree or vice versa. for eg.



It's a symmetric tree



It's not a symmetric tree

Approach:
There are two parts of the tree - the left one and the right one.

Some Observations :

- The leftnode of left subtree is equal to the rightnode of right subtree.
- Similarly the rightnode of left subtree is equal to the leftnode of right subtree.

So we can apply these recursively to check if the nodes are equal or not.
If the tree is symmetric then the algorithm reaches the leaf nodes where each node is null, then we return true.

LP_Code2.java

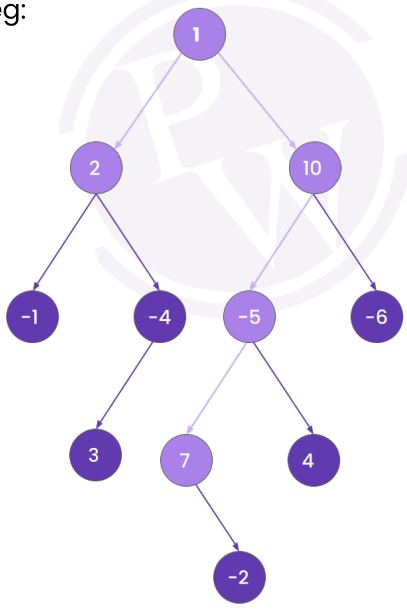
Output:

```
The given tree is Symmetric: true
```

- **Time complexity:** O(n)
- **Space complexity:** O(1)

Interview Problem: Max Path sum in a tree

The problem is to find the maximum path sum between any nodes of a tree.
The path can start from any node and end at any node and need not go through the root of the tree.
eg:



The path highlighted via green nodes is the maximum path sum of this tree and the value is 15.

Approach:
A simple solution would be to traverse the tree and, for every node, calculate the maximum sum path starting from it and passing through its left and right child. Keep track of the maximum among all the maximum sum paths found and return it after all nodes are processed. The time complexity of this solution is O(n2), where n is the total number of nodes in the binary tree.

We can reduce the time complexity to linear by traversing the tree in a bottom-up manner. Each node returns the maximum path sum "starting" at that node to its parent. To ensure that the maximum path sum "starts" at that node, at most, one child of the node should be involved.

The maximum path sum passing "through" a node is the maximum of the following:

1. Node's value.
2. Node's value + maximum path sum "starting" from its left child.
3. Node's value + maximum path sum "starting" from its right child.
4. Node's value + maximum path sum "starting" from its left child + maximum path sum "starting" from its right child.

So the idea is we will start traversing the tree in a bottom up manner and at each node will check the maximum path sum passing through that particular node and update the global maximum path. And we will return the maximum path passing through this node i.e. either the left subtree or right subtree or the node itself only.

LP_Code3.java

Output:

```
The maximum path sum is 15
```

The time complexity of the above solution is O(n), where n is the total number of nodes in the binary tree. The auxiliary space required by the program is O(h) for the call stack, where h is the height of the tree.

Interview problem: Least Common Ancestor in a tree

The lowest common ancestor (LCA) of two nodes x and y in a binary tree is the lowest (i.e., deepest) node that has both x and y as descendants, where each node can be a descendant of itself (so if x is reachable from w, w is the LCA). In other words, the LCA of x and y is the shared ancestor of x and y that is located farthest from the root.

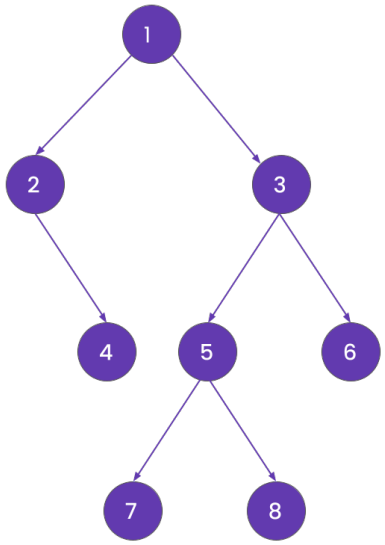
For example, consider the following binary tree. Let x = 6 and y = 7. The common ancestors of nodes x and y are 1 and 3. Out of nodes 1 and 3, the LCA is 3 as it is farthest from the root.

LCA(6,7)=3

LCA(5,8)=5

LCA(3,6)=3

LCA(2,5)=1



Approach:

- A simple solution would be to store the path from root to x and the path from the root to y in two auxiliary arrays.
- Then traverse both arrays simultaneously till the values in the arrays match.
- The last matched value will be the LCA. If the end of one array is reached, then the last seen value is LCA.

The time complexity of this solution is O(n), where n is the total number of nodes in the binary tree. But the auxiliary space used by it is O(n) required for storing two arrays.

Optimised Approach

We can recursively find the lowest common ancestor of nodes x and y present in the binary tree. The trick is to find the node in a binary tree with one key present in its left subtree and the other key present in the right subtree. If any such node is present in the tree, then it is LCA; if y lies in the subtree rooted at node x, then x is the LCA; otherwise, if x lies in the subtree rooted at node y, then y is the LCA.

So the idea is, we start from the root node to find the LCA node

We will check for 2 base cases.

If (root== null) return false
If(root == x || root==y) lca == root; return true;
Now we know that if x and y are present they must be in different parts of a root node. So we will check those two parts and if both of them return true

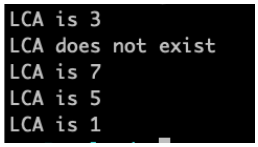
```
// recursively check if `x` or `y` exists in the left subtree
    boolean left = findLCA(root.left, lca, x, y);

    // recursively check if `x` or `y` exists in the right subtree
    boolean right = findLCA(root.right, lca, x, y);

    // if `x` is found in one subtree and `y` is found in the other subtree,
    // update lca to the current node
    if (left && right) {
      lca.node = root;
    }

// return true if `x` or `y` is found in either left or right subtree
    return left || right;
```

Output:



Next Class Teaser

- Binary Search Tree