Max Width Binary Tree

Give DSA notes along with this code: make clear and crispy

```
class Solution {
    public int widthOfBinaryTree(TreeNode root) {
        class Pair{
            TreeNode node;
            int val;
            private Pair(TreeNode node, int val){
                this.node = node;
                this.val = val;
            }
        }
        int maxWidth = 0;
        if(root!=null){
            Queue<Pair> queue = new LinkedList();
            queue.add(new Pair(root,1));
            while(!queue.isEmpty()){
                int size=queue.size();
                int arr[] = new int[2];
                arr[0] = -1;
                while(size>0){
                    Pair pair = queue.remove();
                    TreeNode node = pair.node;
                    int val = pair.val;
                    if(arr[0]==-1){
                        arr[0] = val;
                    }
                    if(node.left!=null){
                        queue.add(new Pair(node.left,2*(val-1)+1));
                    }
                    if(node.right!=null){
                        queue.add(new Pair(node.right,2*(val-1)+2));
                    }
                    arr[1] = val;
                   --size;
                }
                int currentWidth = (arr[1]-arr[0])+1;
                maxWidth = Math.max(maxWidth,currentWidth);
            }
        }
```

```
        return maxWidth;
    }
}
```

Here's a clear and concise explanation of the **Maximum Width of Binary Tree** problem along with the provided Java solution.

Problem Statement

Given the root of a binary tree, return the **maximum width** of the tree.
The **width** of a level is defined as the number of nodes between the leftmost and rightmost non-null nodes in that level (including null placeholders). The **maximum width** is the largest width among all levels.

Approach

- Perform a **level-order traversal** (BFS) using a queue.
- Use a **Pair class** to track each node and its position (`val`) in a **1-based index system**.
- The left and right child indices are determined using:
  - **Left child**: `2 * (val - 1) + 1`
  - **Right child**: `2 * (val - 1) + 2`
- At each level:
  1. Store the first and last node's position (`arr[0]` and `arr[1]`).
  2. Compute the width as `arr[1] - arr[0] + 1`.
  3. Update `maxWidth` accordingly.

Code Explanation

java

```java
class Solution {
    public int widthOfBinaryTree(TreeNode root) {
        class Pair {
            TreeNode node;
            int val;
            private Pair(TreeNode node, int val) {
                this.node = node;
                this.val = val;
            }
        }

        int maxWidth = 0;
        if (root != null) {
            Queue<Pair> queue = new LinkedList<>();
            queue.add(new Pair(root, 1));

            while (!queue.isEmpty()) {
                int size = queue.size();
                int arr[] = new int[2]; // Stores first and last node position
                arr[0] = -1; // Initial marker

                while (size > 0) {
                    Pair pair = queue.remove();
                    TreeNode node = pair.node;
                    int val = pair.val;

                    if (arr[0] == -1) {
                        arr[0] = val; // First node's position in the level
                    }

                    if (node.left != null) {
                        queue.add(new Pair(node.left, 2 * (val - 1) + 1));
                    }
                    if (node.right != null) {
                        queue.add(new Pair(node.right, 2 * (val - 1) + 2));
                    }

                    arr[1] = val; // Last node's position in the level
                    --size;
                }
```

```
                int currentWidth = (arr[1] - arr[0]) + 1;
                maxWidth = Math.max(maxWidth, currentWidth);
            }
        }

        return maxWidth;
    }
}
```

Time and Space Complexity

- **Time Complexity**: **O(N)**, where **N** is the number of nodes (since each node is processed once).
- **Space Complexity**: **O(N)** in the worst case (when the tree is a complete binary tree, requiring queue space for the last level).

Key Takeaways

1. **Breadth-First Search (BFS) is ideal** for level-based operations.
2. **Tracking node positions using indices** allows for correct width calculation.
3. **Avoiding overflow**: Using `val - 1` ensures smaller values when computing child indices.
4. **Handles null placements implicitly**, ensuring correct width calculation.

Let me know if you need further clarifications! 🚀

What is the pattern this

The pattern in this problem is based on **indexing nodes in a binary tree using a 1-based or 0-based position system**, similar to how **Complete Binary Trees** are represented in arrays.

Pattern Observed

1. **Assign an index to each node**:
   - The **root** is assigned an index (1 in a 1-based system or 0 in a 0-based system).
   - **Left child** of a node at index `i` is at index → `2 * i + 1` (0-based) or `2 * (i - 1) + 1` (1-based).
   - **Right child** of a node at index `i` is at index → `2 * i + 2` (0-based) or `2 * (i - 1) + 2` (1-based).
2. **Each level in BFS follows a sequence of indices**:
   - Example for a **1-based index system**:

     markdown

     ```
           1
         /   \
        2     3
       / \   / \
      4   5 6   7
     ```

     - Level 0: `[1]`
     - Level 1: `[2, 3]`
     - Level 2: `[4, 5, 6, 7]`
   - Indices:
     - `Root = 1`
     - `Left child = 2 * (1 - 1) + 1 = 1`
     - `Right child = 2 * (1 - 1) + 2 = 2`
     - `Left child of 2 = 2 * (2 - 1) + 1 = 3`
     - `Right child of 2 = 2 * (2 - 1) + 2 = 4`
     - **So on...**
3. **Using First and Last Index for Width Calculation**:
   - The width of a level is computed as:

$$\text{width} = \text{last index} - \text{first index} + 1$$

This pattern is useful in **tree traversal and width calculations** without needing explicit null placeholders.