# Maximum Width of Binary Tree

Given a binary tree, write a function to get the maximum width of the given tree. The width of a tree is the maximum width among all levels. The binary tree has the same structure as a **full binary tree**, but some nodes are null.

The width of one level is defined as the length between the end-nodes (the leftmost and right most non-null nodes in the level, where the `null` nodes between the end-nodes are also counted into the length calculation.

## Example 1:

```
Input: 

           1
         /   \
        3     2
       / \     \  
      5   3     9 

Output: 4
Explanation: The maximum width existing in the third level with the length 4 (5,3,nul
l,9).
```

## Example 2:

```
Input: 

          1
         /  
        3    
       / \       
      5   3     

Output: 2
Explanation: The maximum width existing in the third level with the length 2 (5,3).
```

## Example 3:

```
Input: 

          1
         / \
        3   2 
       /        
      5      

Output: 2
Explanation: The maximum width existing in the second level with the length 2 (3,2).
```

## Example 4:

```
Input: 

          1
         / \
        3   2
       /     \  
      5       9 
     /         \ 
    6           7
Output: 8
Explanation:The maximum width existing in the fourth level with the length 8 (6,null,
null,null,null,null,null,7).
```

**Note:** Answer will in the range of 32-bit signed integer.

## Solution 1

We know that a binary tree can be represented by an array (assume the root begins from the position with index `1` in the array). If the index of a node is `i`, the indices of its two children are `2*i` and `2*i + 1`. The idea is to use two arrays (`start[]` and `end[]`) to record the the indices of the leftmost node and rightmost node in each level, respectively. For each level of the tree, the width is `end[level] - start[level] + 1`. Then, we just need to find the maximum width.

Java version:

```
    public int widthOfBinaryTree(TreeNode root) {
        return dfs(root, 0, 1, new ArrayList<Integer>(), new ArrayList<Integer>());
    }

    public int dfs(TreeNode root, int level, int order, List<Integer> start, List<In
teger> end){
        if(root == null)return 0;
        if(start.size() == level){
            start.add(order); end.add(order);
        }
        else end.set(level, order);
        int cur = end.get(level) - start.get(level) + 1;
        int left = dfs(root.left, level + 1, 2*order, start, end);
        int right = dfs(root.right, level + 1, 2*order + 1, start, end);
        return Math.max(cur, Math.max(left, right));
    }
```

C++ version (use `vector<pair<int,int>>` to replace the arrays `start` and `end` in Java ):

```
   int widthOfBinaryTree(TreeNode* root) {
        return dfs(root, 0, 1, vector<pair<int, int>>() = {});
    }

    int dfs(TreeNode* root, int level, int order, vector<pair<int, int>>& vec){
        if(root == NULL)return 0;
        if(vec.size() == level)vec.push_back({order, order});
        else vec[level].second = order;
        return max({vec[level].second - vec[level].first + 1, dfs(root->left, level
+ 1, 2*order, vec), dfs(root->right, level + 1, 2*order + 1, vec)});
    }
```

written by Hao Cai original link here

## Solution 2

The idea is to use heap indexing:

```
        1
   2         3
 4   5     6   7
8 9 x 11  x 13 x 15
```

Regardless whether these nodes exist:

- Always make the id of left child as `parent_id * 2`;
- Always make the id of right child as `parent_id * 2 + 1`;

So we can just:

1. Record the `id` of `left most node` at each level of the tree(you can tell be check the size of the container to hold the first nodes);
2. At each node, compare the `distance` from it the left most node with the current `max width`;

### DFS C++

```
class Solution {
public:
    int widthOfBinaryTree(TreeNode* root) {
        vector<int> lefts; // left most nodes at each level;
        int maxwidth = 0;
        dfs(root, 1, 0, lefts, maxwidth);
        return maxwidth;
    }
private:
    void dfs(TreeNode* node, int id, int depth, vector<int>& lefts, int& maxwidth) {
        if (!node) return;
        if (depth >= lefts.size()) lefts.push_back(id);  // add left most node
        maxwidth = max(maxwidth, id + 1 - lefts[depth]);
        dfs(node->left, id * 2, depth + 1, lefts, maxwidth);
        dfs(node->right, id * 2 + 1, depth + 1, lefts, maxwidth);
    }
};
```

### DFS Java

```
class Solution {
    public int widthOfBinaryTree(TreeNode root) {
        List<Integer> lefts = new ArrayList<Integer>(); // left most nodes at each l
evel;
        int[] res = new int[1]; // max width
        dfs(root, 1, 0, lefts, res);
        return res[0];
    }
    private void dfs(TreeNode node, int id, int depth, List<Integer> lefts, int[] re
s) {
        if (node == null) return;
        if (depth >= lefts.size()) lefts.add(id);   // add left most node
        res[0] = Integer.max(res[0], id + 1 - lefts.get(depth));
        dfs(node.left,  id * 2,     depth + 1, lefts, res);
        dfs(node.right, id * 2 + 1, depth + 1, lefts, res);
    }
}
```

## BFS C++

```
class Solution {
public:
    int widthOfBinaryTree(TreeNode* root) {
        if (!root) return 0;
        int max = 0;
        queue<pair<TreeNode*, int>> q;
        q.push(pair<TreeNode*, int>(root, 1));
        while (!q.empty()) {
            int l = q.front().second, r = l; // right started same as left
            for (int i = 0, n = q.size(); i < n; i++) {
                TreeNode* node = q.front().first;
                r = q.front().second;
                q.pop();
                if (node->left) q.push(pair<TreeNode*, int>(node->left, r * 2));
                if (node->right) q.push(pair<TreeNode*, int>(node->right, r * 2 + 1
));
            }
            max = std::max(max, r + 1 - l);
        }
        return max;
    }
};
```

## BFS Java

```java
import java.util.AbstractMap;
class Solution {
    public int widthOfBinaryTree(TreeNode root) {
        if (root == null) return 0;
        int max = 0;
        Queue<Map.Entry<TreeNode, Integer>> q = new LinkedList<Map.Entry<TreeNode, I
nteger>>();
        q.offer(new AbstractMap.SimpleEntry(root, 1));

        while (!q.isEmpty()) {
            int l = q.peek().getValue(), r = l; // right started same as left
            for (int i = 0, n = q.size(); i < n; i++) {
                TreeNode node = q.peek().getKey();
                r = q.poll().getValue();
                if (node.left != null) q.offer(new AbstractMap.SimpleEntry(node.lef
t, r * 2));
                if (node.right != null) q.offer(new AbstractMap.SimpleEntry(node.ri
ght, r * 2 + 1));
            }
            max = Math.max(max, r + 1 - l);
        }

        return maxwidth;
    }
}
```

written by alexander original link here

## Solution 3

It's easy to use a travel traversal template, use another queue to keep the index of each level nodes. left node index = this node index * 2, right = this node index*2 + 1. the width should be the last node index - first node index + 1

```
    public int widthOfBinaryTree(TreeNode root) {
        if(root == null) return 0;
        ArrayDeque<TreeNode> queue = new ArrayDeque<>();
        ArrayDeque<Integer>  count = new ArrayDeque<>();
        queue.offer(root);
        count.offer(0);
        int max = 1;

        while(!queue.isEmpty()) {
            int size = queue.size();
            int left = 0;
            int right = 0;
            for(int i = 0; i < size; i++) {
                TreeNode node = queue.poll();
                int index = count.poll();
                if(i == 0)  left = index;
                if(i == size-1)  right = index;
                if(node.left != null) {
                    queue.offer(node.left);
                    count.offer(index*2);
                }
                if(node.right != null) {
                    queue.offer(node.right);
                    count.offer(index*2 + 1);
                }
            }
            max = Math.max(max,right - left + 1);
        }
        return max;
    }
```

written by 寂寞土豆 original link here