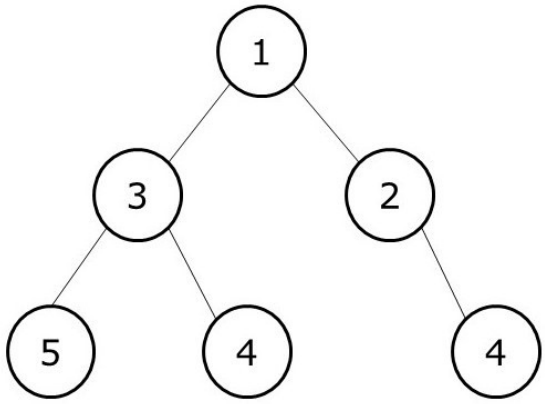
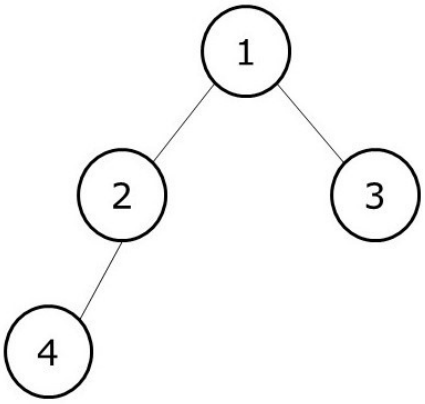
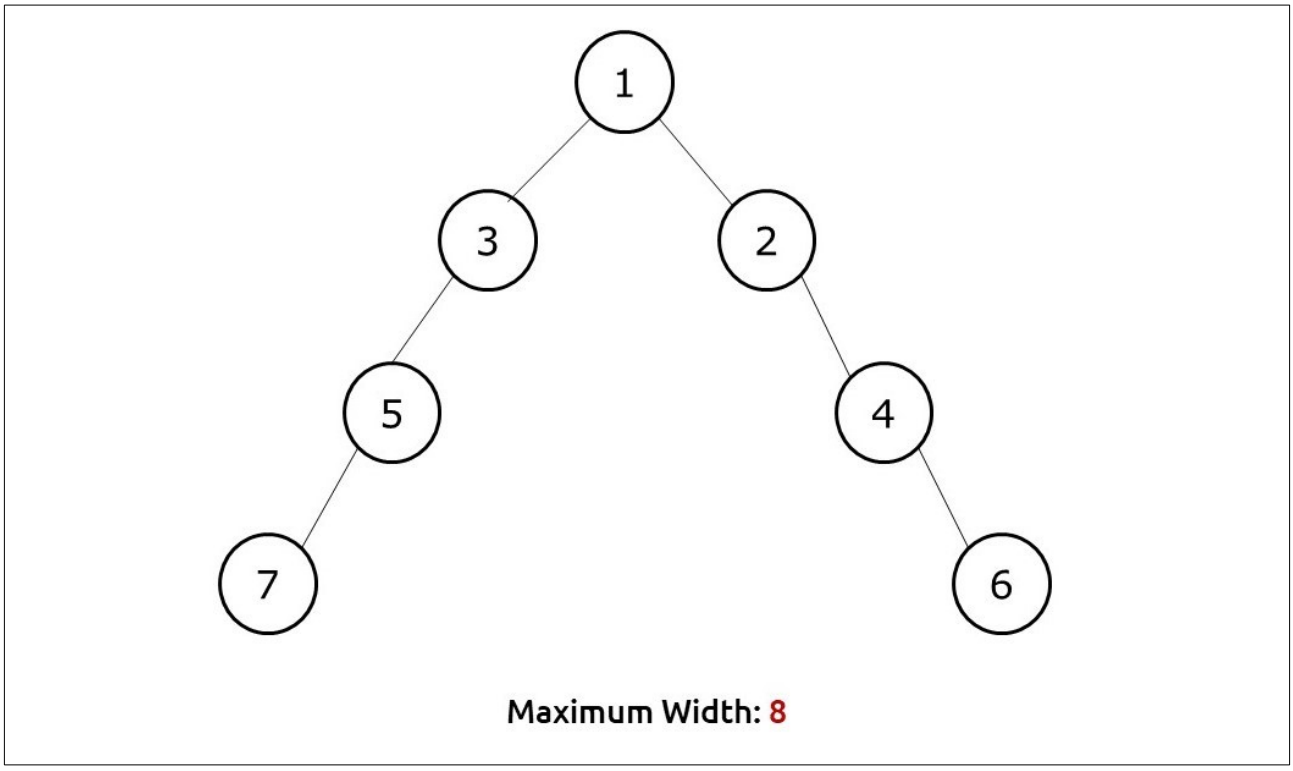
Maximum Width of a Binary Tree
Problem Statement: Write a program to find the **Maximum Width of A Binary Tree**.
Examples:



Maximum Width: 4



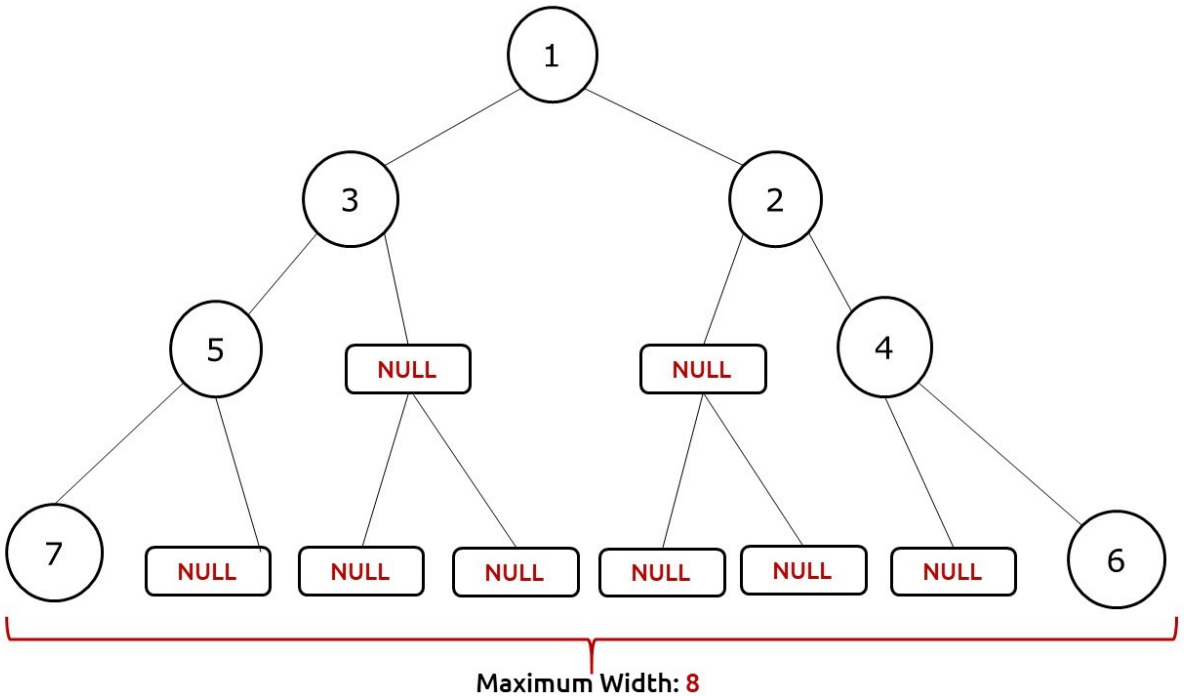
Maximum Width: 2



Problem Description:
The maximum width of a binary tree is the maximum of all the level widths. Width for a level is defined as the maximum number of nodes between the leftmost and rightmost node of the level(including the end nodes and the null nodes between the end nodes).
Pre-req: Level Order Traversal
Disclaimer. *Don't jump directly to the solution, try it out yourself first.*

Solution :

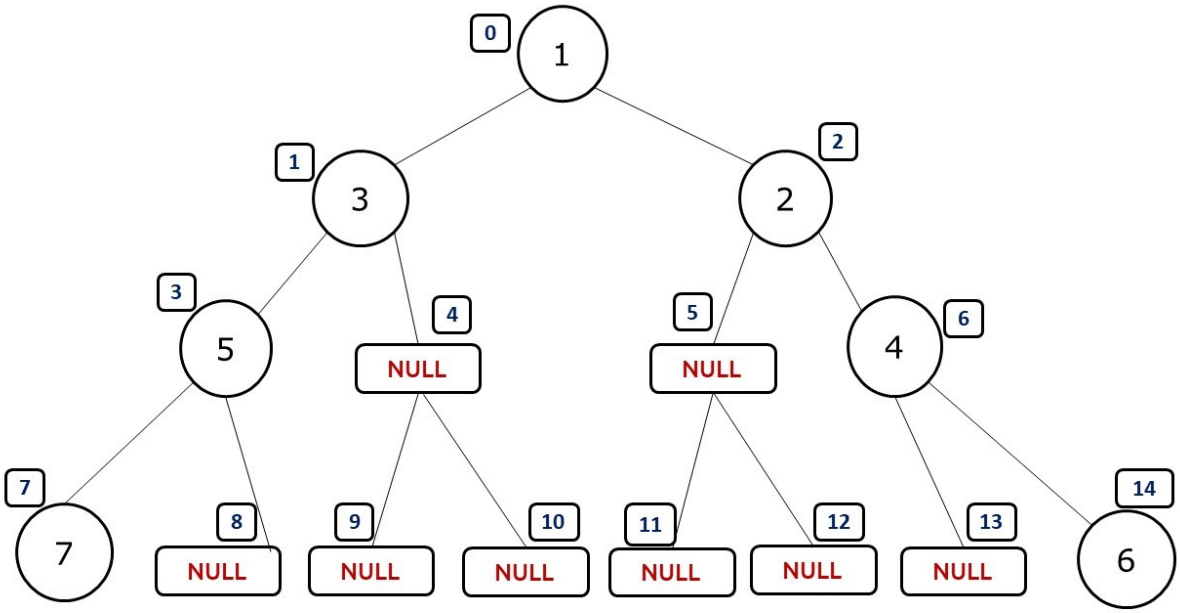
Intuition:
First of all, we need to understand the meaning of width at a level clearly. In the following image, we can see how the null nodes play an important role in the width calculation



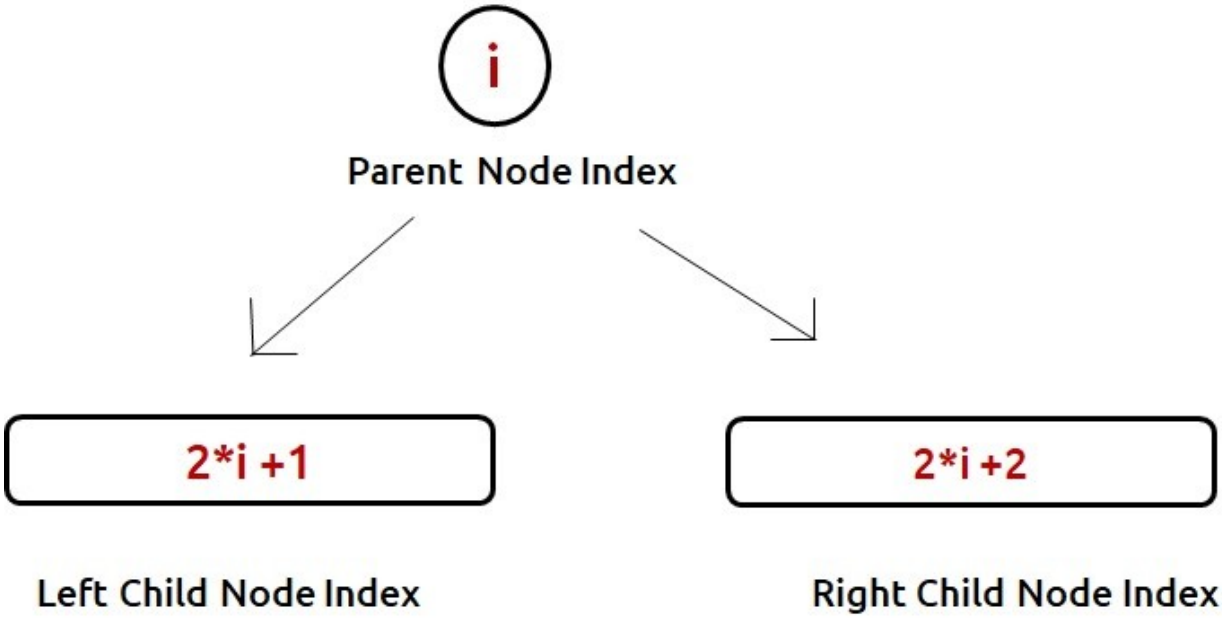
Now we see that the width is defined by the nodes of one particular level. Therefore we can use a level order traversal to traverse the tree and in every level, we try to find the leftmost and rightmost node of that level. To achieve this we would need a proper indexing strategy to uniquely index nodes of a level. Once we know the leftMost and rightMost nodes, width can be defined as (rightMost- leftMost +1).

Approach:

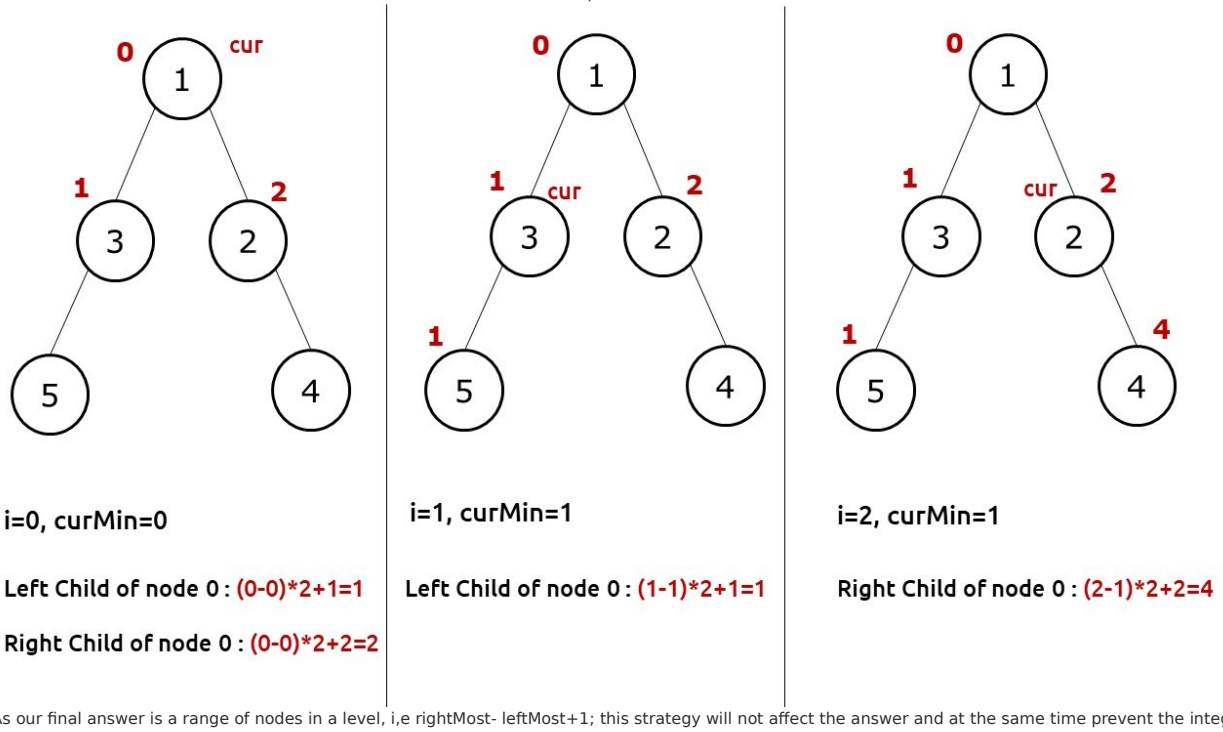
We will perform a special level order traversal with two loops where inner loops traverse the nodes of a single level. This is to ensure that we can do our calculations once a single level is traversed. In the traversal, we will assign an index to a node. The indexing strategy is described as below:



If we index the tree as shown above we can easily calculate the width of the tree as rightMostNode – leftMostNode +1. Then we can return the maximum width as our answer. To store the index, we can use a pair of values in our queue(that we use for level order traversal). If we are at index i, then its left and right child are(in 0-based indexing): 2*i+1 and 2*i+2 respectively. Please note that NULL nodes are not hampering the indexing in any way.



Prevention of Integer Overflow
This approach has a problem, as we are multiplying 2 to the current index, it can happen in a tree that we overshoot the bound of an integer. Therefore, we need to find a strategy to prevent it.
Before starting a level, we can store the left-most index in a variable(say curMin). Now whenever we assign the index for its children, we take the parent node index as (i-curMin) rather than i. The below illustration will clear the concept.



As our final answer is a range of nodes in a level, i,e rightMost- leftMost+1; this strategy will not affect the answer and at the same time prevent the integer overflow case.
The algorithm approach can be stated as:

- We take a queue and push the root node along with index 0.
- We traverse the tree using a level order traversal.
- In the level order traversal we set another loop to run for the size of the queue, so that we visit the same level nodes inside it.
- Before a level starts, we use a variable(say curMin) to store the index of the first node.
- We assign an index to every node, and to its children as described above.
- When the inner loop is at the first node of a level, we store its index in another variable(sayleftMost)
- When the inner loop is at the last node of a level, we store its index in another variable(say rightMost)
- After a level in the outer loop, we calculate the width of the level as (rightMost – leftMost +1).
- We return the maximum width as the answer.

Dry Run: In case you want to watch the dry run for this approach, please watch the video attached below.
Code:

- C++ Code

- Java Code

```cpp
#include <bits/stdc++.h>

using namespace std;

struct node {
   int data;
   struct node * left, * right;
};

int widthOfBinaryTree(node * root) {
   if (!root)
      return 0;
   int ans = 0;
   queue < pair < node * , int >> q;
   q.push({
      root,
      0
   });
   while (!q.empty()) {
      int size = q.size();
      int curMin = q.front().second;
      int leftMost, rightMost;
      for (int i = 0; i < size; i++) {
         int cur_id = q.front().second - curMin; // subtracted to prevent integer overflow
         node * temp = q.front().first;
         q.pop();
         if (i == 0) leftMost = cur_id;
         if (i == size - 1) rightMost = cur_id;
         if (temp -> left)
            q.push({
               temp -> left,
               cur_id * 2 + 1
            });
         if (temp -> right)
            q.push({
               temp -> right,
               cur_id * 2 + 2
            });
      }
      ans = max(ans, rightMost - leftMost + 1);
   }
   return ans;
}

struct node * newNode(int data) {
   struct node * node = (struct node * ) malloc(sizeof(struct node));
   node -> data = data;
   node -> left = NULL;
   node -> right = NULL;

   return (node);
}

int main() {

   struct node * root = newNode(1);
   root -> left = newNode(3);
   root -> left -> left = newNode(5);
   root -> left -> left -> left = newNode(7);
   root -> right = newNode(2);
   root -> right -> right = newNode(4);
   root -> right -> right -> right = newNode(6);

   int maxWidth = widthOfBinaryTree(root);
   cout << "The maximum width of the Binary Tree is " << maxWidth;

   return 0;
```

```
}
```

Output:

The maximum width of the Binary Tree is 8

Time Complexity: O(N)

Reason: We are doing a simple level order traversal. The inner loop simply traverses the nodes level-wise and doesn't add to the complexity.

Space Complexity: O(N)