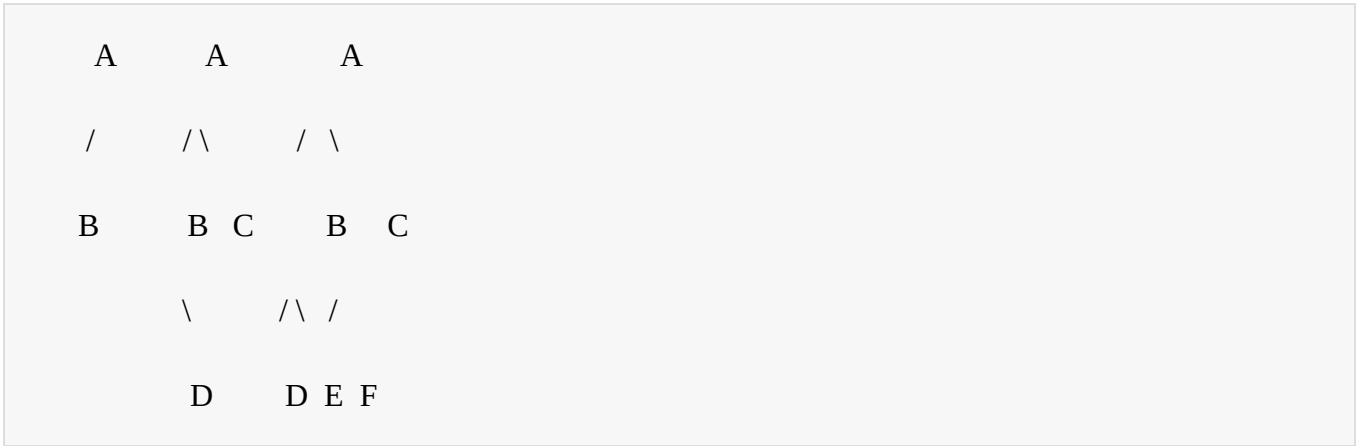
CHECK IF A BINARY TREE IS COMPLETE BINARY TREE

A Complete Binary Tree is a Binary Tree where each level is completely filled. For example, all the trees below are complete Binary trees.

And the trees below are not Complete (because there are holes in between):



Given a binary tree, write code to check if the tree is a Complete Binary Tree or not.

Wrong Solution-1: Strictly Binary tree is not complete Binary tree

If each node has either 2 or zero child then its a Complete binary Tree.

No

The below tree is not Complete (it is strict Binary tree, but not Complete).

```
    A
   / \
  B   C
     / \
    E   D
```

Wrong-Solution-2: If all leaf nodes are at same level

If all leaf nodes are at the same level then the tree is Complete Binary tree?

NO

The below tree has all the leaf nodes at the same level, but it is still not a complete binary tree

```
    A
     \
```

```
      C
     / \
    E   D
```

There are only two nodes E and F and both are at same level, but it is not a Complete Binary tree.

Correct Solution-1: Combine above two wrong solutions

If we combine the other two solutions. i.e All the leaf nodes should be at the same level AND all the nodes should have either zero or two childs.

Correct Solution-2: Use equation between height and number of nodes

Fact about Complete binary Tree.

If height of the tree is h, then the total number of nodes will be 2^h-1 and vice-versa.

Hence, If we just find height of the Binary tree, and then count the number of nodes in the tree then we can say for certainty, whether or not the tree is Complete.

Algorithm:

```
h = Height of Tree
n = Number of nodes in the Tree
if(n = 2^h -1)
   return true
```

```
else
    return false
```

Code:

If the Node of tree is defined as

```
1    struct Node
2    {
3        Node *lptr;
4        int data;
5        Node *rptr;
6    };
```

Then below are the functions to compute height and the number of Nodes in Binary tree:

```
1    /** Function to compute the height of the tree.
2     *  root is a pointer to the root of the tree
3     */
4    int getHeight(Node* root)
5    {
6        if (root == NULL)
7            return 0;
8
```

```
9         // Compute height of each tree
10        int heightLeft = getHeight(root->lptr);
11        int heightRight = getHeight(root->rptr);
12
13        /* use the larger one */
14        if (heightLeft > heightRight)
15            return(heightLeft + 1);
16        else
17            return(heightRight + 1);
18    }
19
20    /**
21     * Count the total number of nodes in a Binary tree.
22     */
23    int countNodes(Node* r)
24    {
25        if(r == NULL)
26            return 0;
27        return 1 + countNodes(r->rptr) + countNodes(r->lptr);
28    }
```

The two functions above can be merged into one, so that the tree is traversed only once. Pass a count parameter as a referenceparameter to the getHeight function and increment it once in each recursive function call.

The time complexity will still be O(n) but we will gain in the constant factor (In this case the tree is traversed twice – once to compute the height and then to count the number of nodes).

And finally below code check whether the tree is a complete binary tree of not.

```
1   // Helper function. to compute x^n. Assumes n to be positive.
2   int exponent(int x, int n)
3   {
4       int pow = 1;
5       int i=0;
6       for(i=0; i<n; i++)
7           pow *= x;
8       return pow;
9   }
10  /**
11   * returns true if a binary tree is complete binary tree, false otherwise
12   */
13  int checkCompleteTree(Node* r)
14  {
15      if(r == NULL)
16          return true;
17
18      int h = getHeight(r);
```

```
19          int n = countNodes(r);
20
21          if(n == exponent(2,h)-1)
22              return 1;
23          else
24              return 0;
25      }
```

The time complexity of above function is O(n).. Both getHeight and countNodes functions takes O(n) time each.

Source: http://www.ritambhara.in/check-if-a-binary-tree-is-complete-binary-tree/