Binary Search Trees (BSTs)

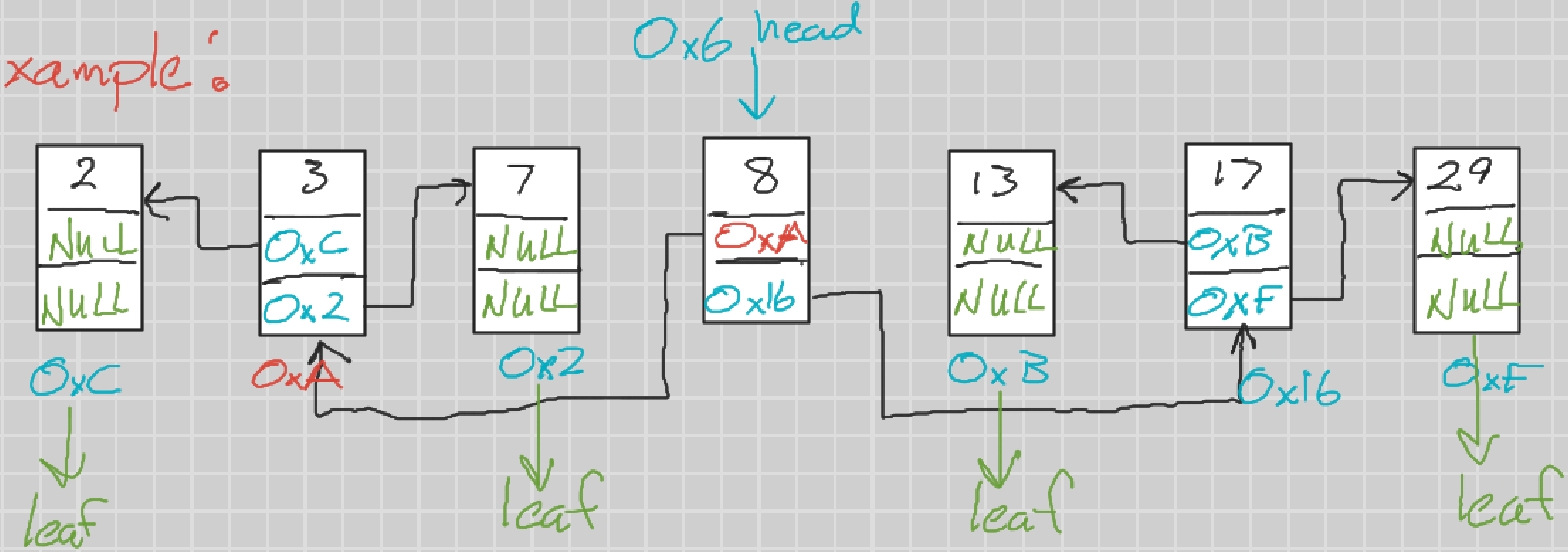
- Binary Search Trees are a tree data structure that enables us to perform Binary Searches
- We can't use a linear structure like a linked list to perform a binary search because they don't have indexes that we can use to find midpoints.

- BST have node just like a singly linked list, however they have pointers that point to a left child and a right child. The left subtree will contain values less than the root, the right subtree will contain values greater than the root, this structure is repeated recursively in each subtree.

- The HEAD will point to the ROOT of the tree, this way we can tell if we have found the value we are searching for in the tree is the first one, without having to traverse the entire tree.

- If we were applying a binary search, the next step would be to check if the value at the root of the left subtree or right subtree is the value we are looking for. You would choose right or left depending on whether or not the value is greater or less than the root of the tree.

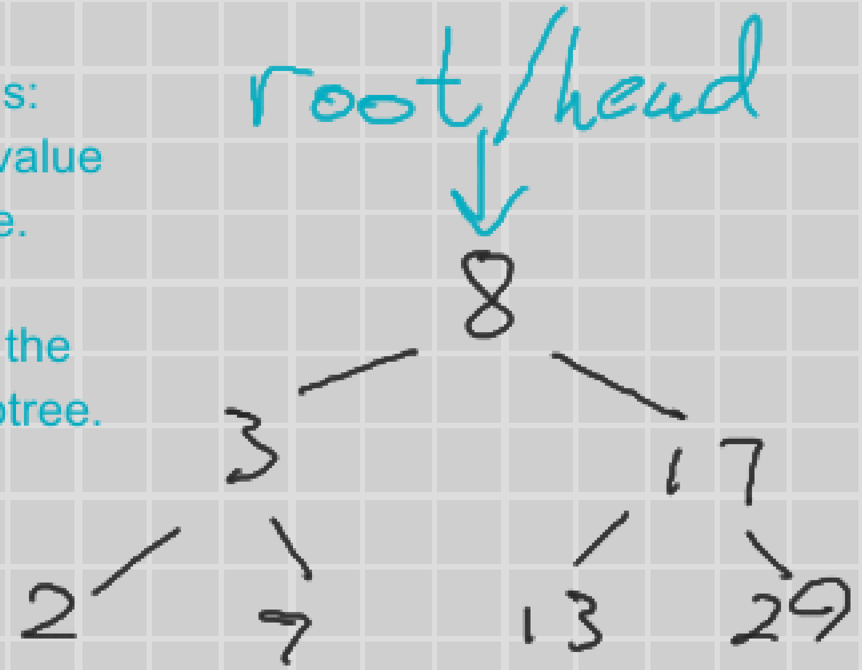
Example:



For a tree to be a BST, it has to meet the following conditions:

1. All nodes stored in the left subtree must be less tahn the value of the root, this needs to follow recursively in each subtree.

2. All nodes stored in the right subtree must be greater than the value of the root, this needs to follow recursively in each subtree.



* This is a balance BST, they won't always work out this way when inserting nodes. It's common to come across the worste case scenario. For a self balancing tree we would use another similar structure like a self balancing AVL tree or a Red-Black tree.

BST: Insert

- In a BST, data is organized in a hierarchal fashion following the rules stated in the previous slide.
- There are three main operations we will review:
  1) Insert
  2) Search
  3) Delete

INSERT has three possible case when attempting to add a value to the tree:

1) ROOT is NULL: in this case we must create the node, assign the value, and return it.

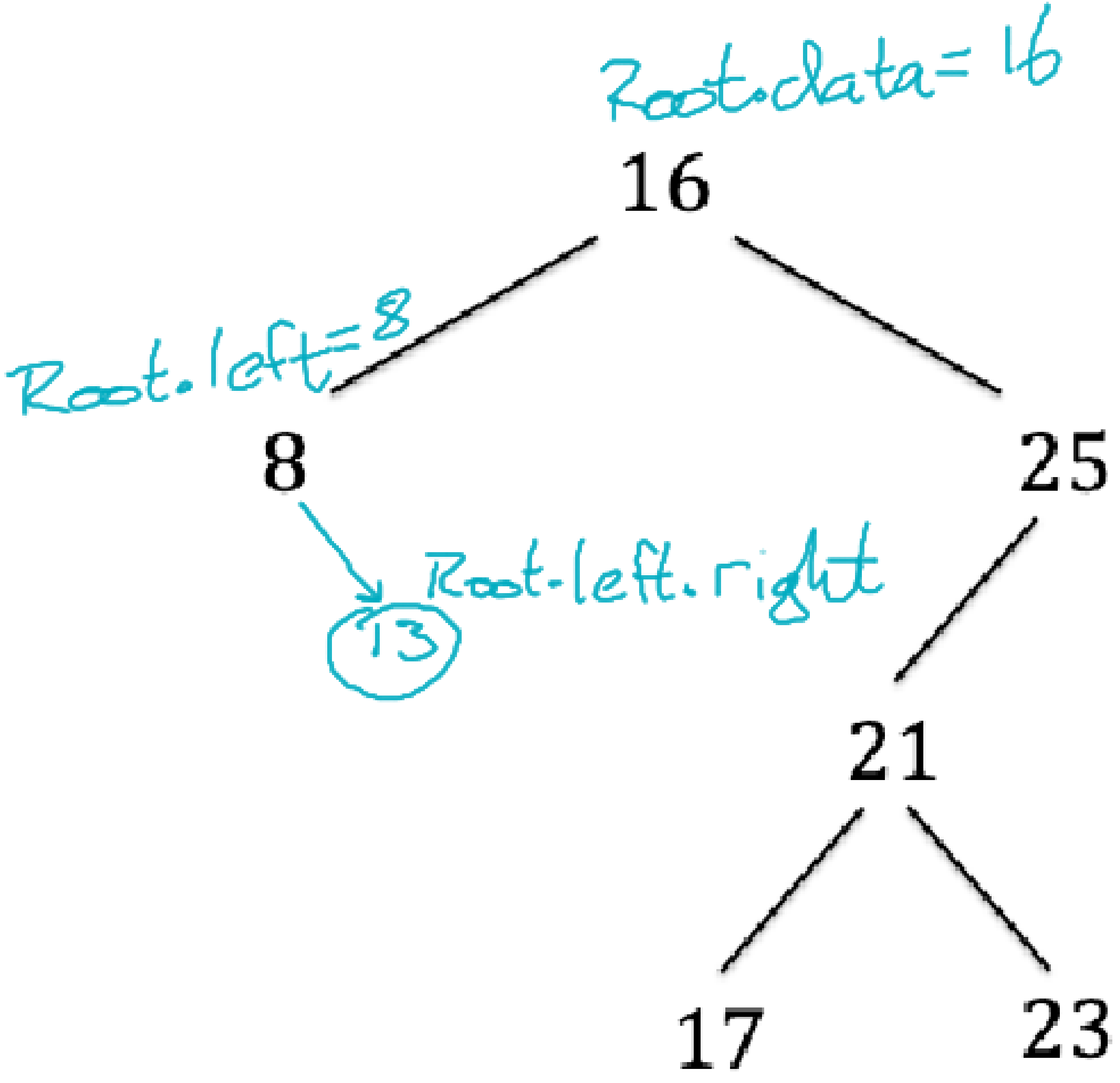
2) VALUE being inserted is LESS than the ROOT: in this case we insert the node as the LEFT child.

3) VALUE being inserted is GREATER than the ROOT: in this case we insert the node as the RIGHT child.

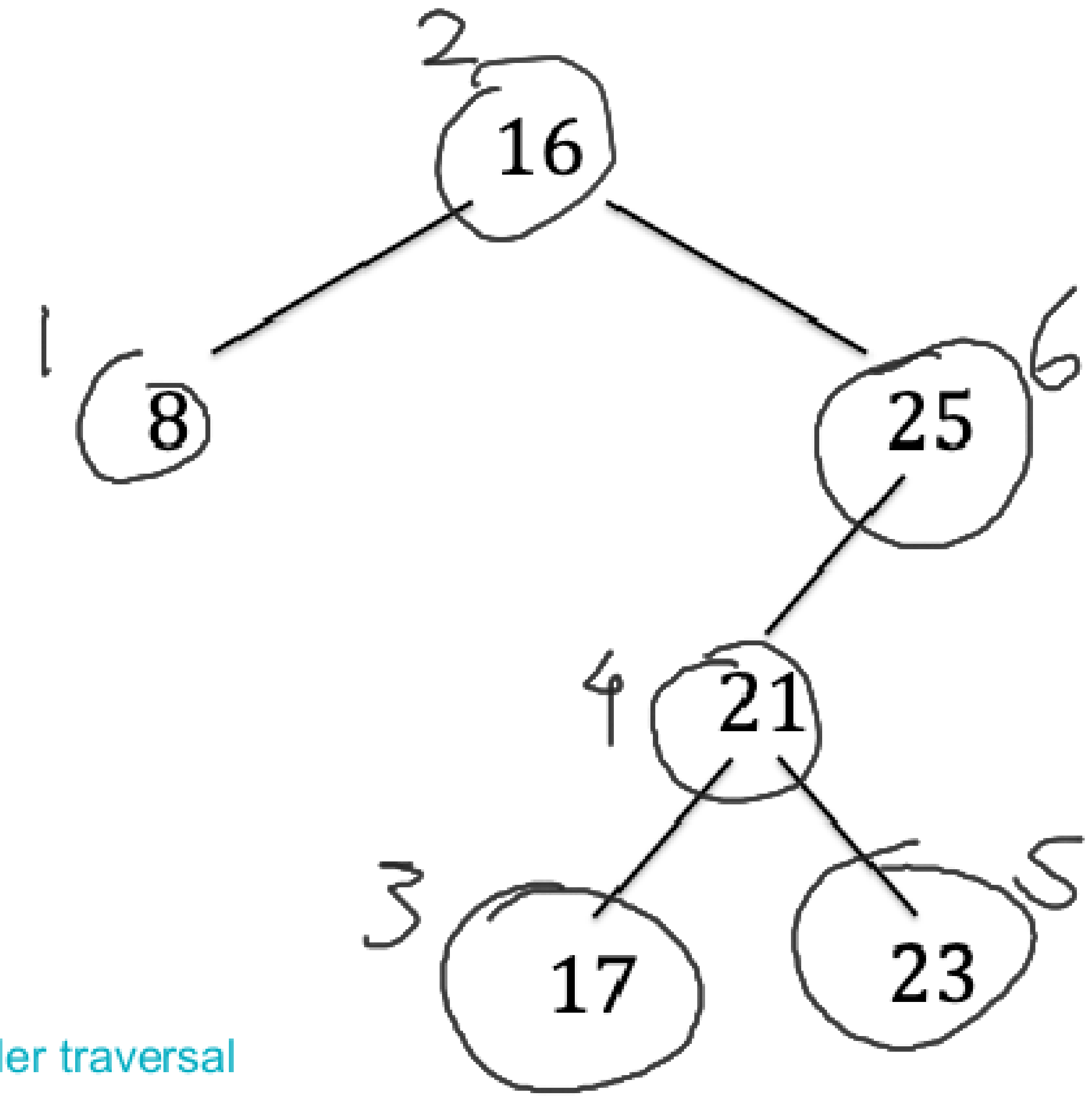
BST: INSERT PSEUDOCODE

```
function INSERT(ROOT, VALUE)
    if ROOT = null
        newNode <-- new NODE(VALUE)
        ROOT <-- newNODE
    else
        if VALUE < ROOT.data
            INSERT(ROOT.left, VALUE)     // Recursive call
        else
            INSERT(ROOT.right, VALUE)     // Recursive call
```

2. Where must number 13 be inserted in this BST?



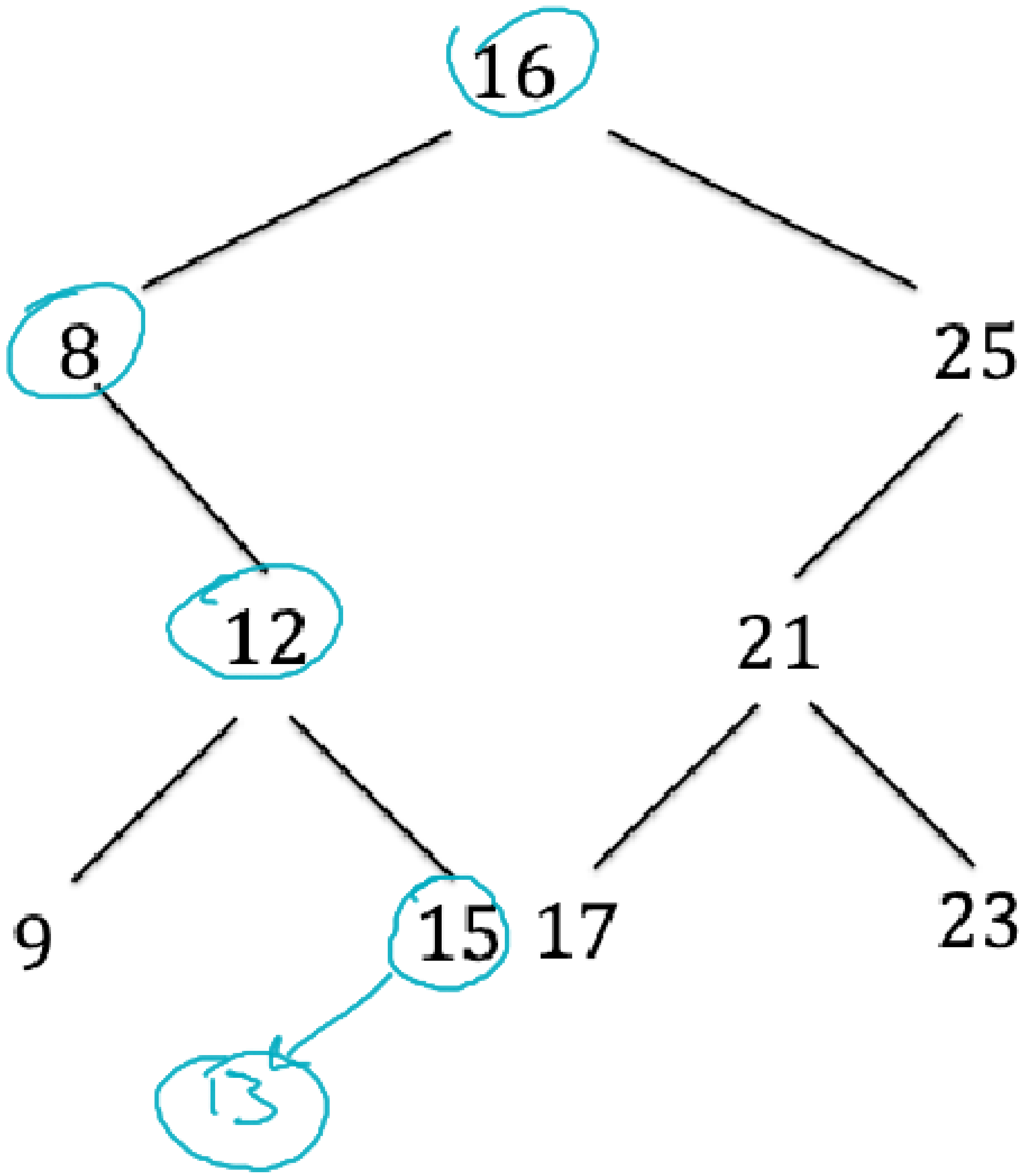
1. For this BST:



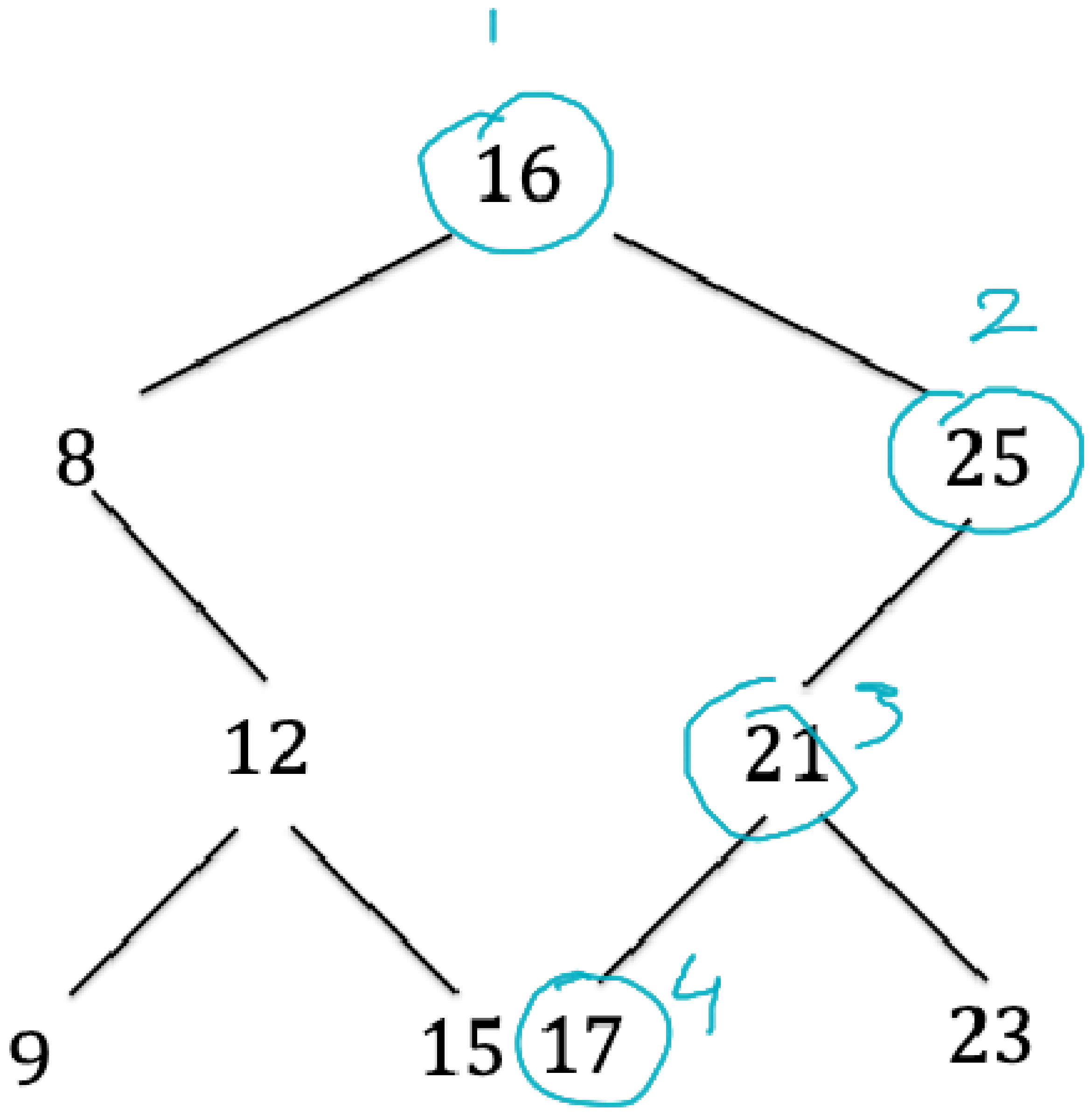
Answer: In order traversal

What traversal visits the nodes in ascending order? That is: 8, 16, 17, 21, 23, 25.

3. Where must number 13 be inserted in this BST?



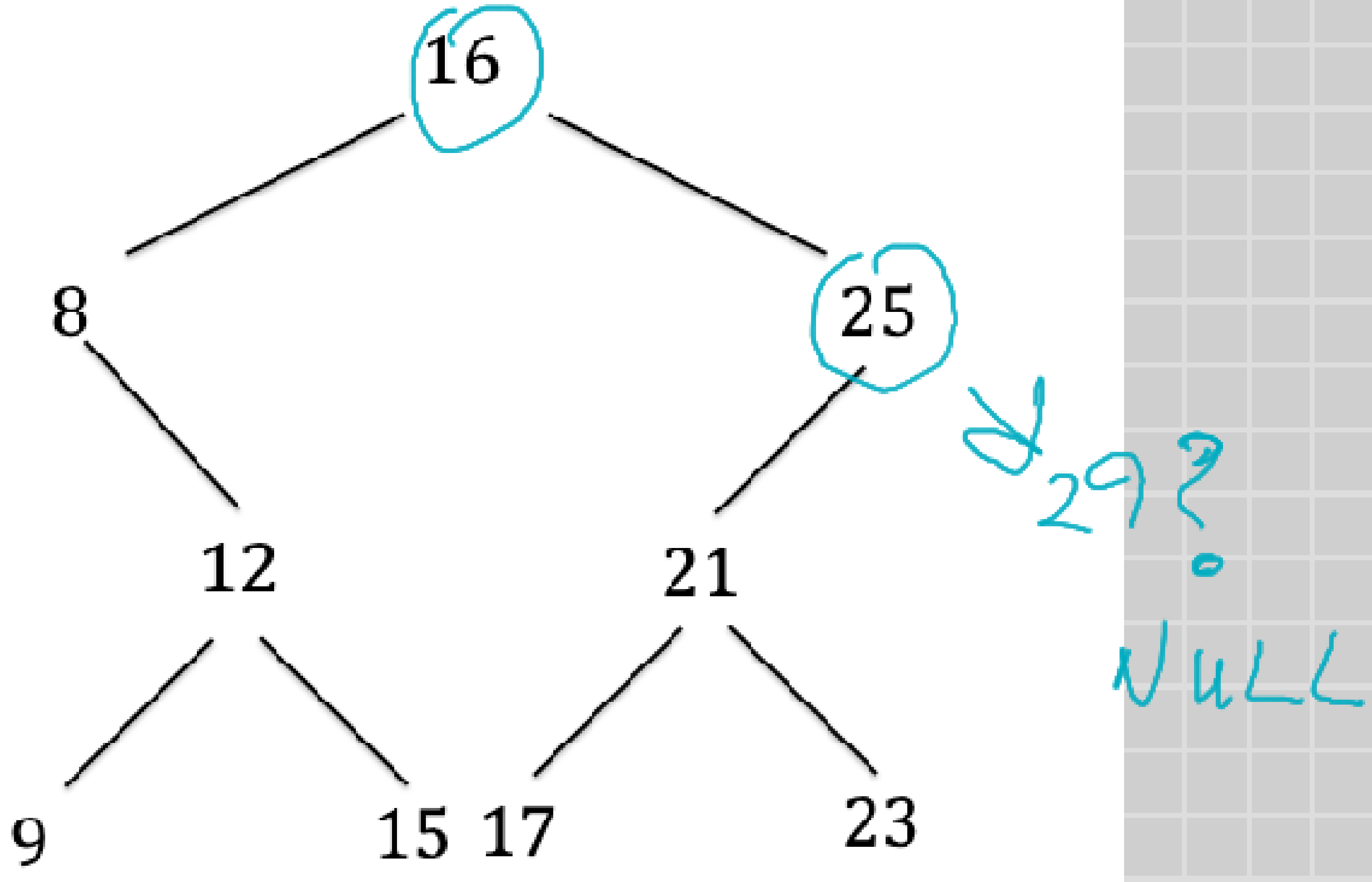
4. For the following BST:



We visit 4 nodes:
16, 25, 21, 17

Which nodes are visited before determining that number 17 is in the tree?

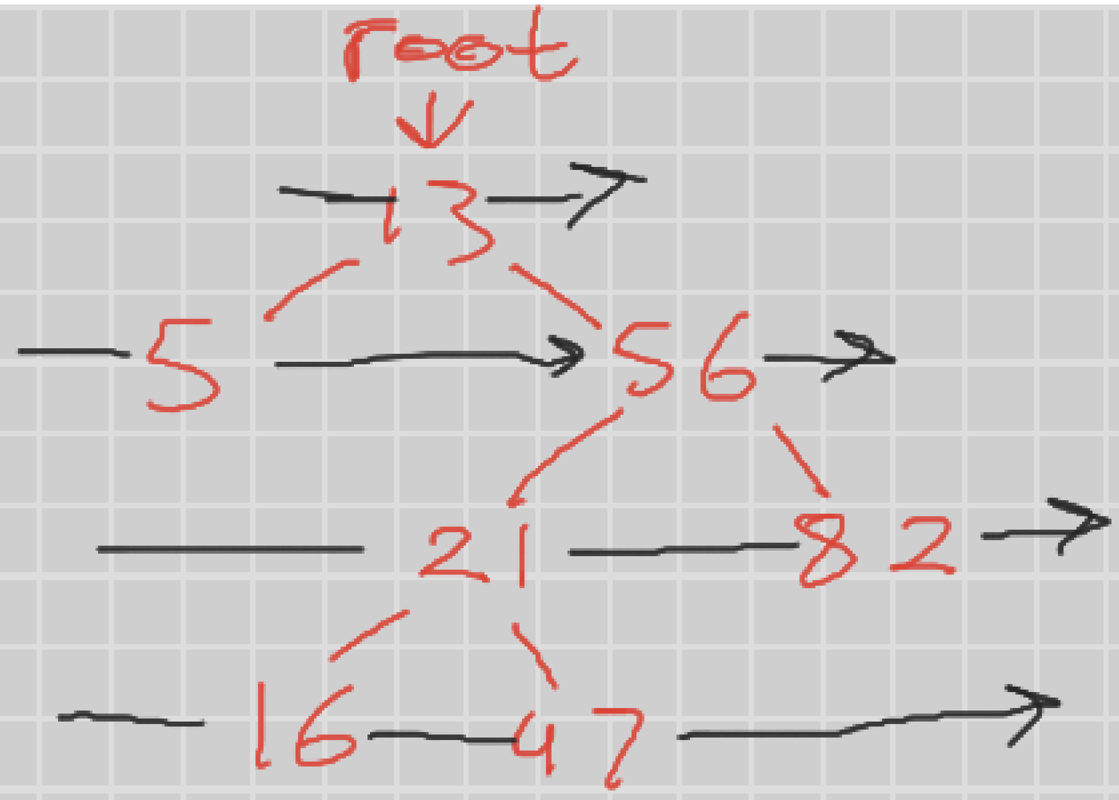
5. For the following BST:



16, 25 : 2 nodes visited

Which nodes are visited before determining that number 29 is not in the tree?

6. The following numbers are inserted in a BST (in this order) : 13, 56, 21, 5, 82, 16, 47. In what order are nodes visited when using a breadth-first traversal?



Breadt-first traversal

13, 5, 56, 21, 82, 16, 47

BST: SEARCH

- The search function of a BST is similar to that of the INSERT function , instead of inserting a value we just return true or false if the value if present.

```
function SEARCH(ROOT, VALUE)
    if ROOT = null
        return false                    // If root is null, it's an empty tree, return false
    else
        if VALUE = ROOT.data
            return true                 // if the value matches root's data then we've found the value
        else                            // otherwise search left or right subtree
            if VALUE < ROOT.data
                SEARCH(ROOT.left, VALUE)   // Recursive call
            else
                SEARCH(ROOT.right, VALUE)   // Recursive call
```

1. The following description of the function SEARCH has a bug:

```
function SEARCH_BST(root,x)
   if (root==NULL)
      return FALSE
   else
      if(x == root->data)
         return TRUE
      else
         if (x> root->data)
         return SEARCH_BST(root->left,x)
         else
           return SEARCH_BST(root->right,x)
end function
```

The bug is:

2. The following description of the function SEARCH has a bug:

```
function SEARCH_BST(root,x)
      if(x == root->data)
         return TRUE
      else
         if (x> root->data)
          return SEARCH_BST(root->right,x)
         else
          return SEARCH_BST(root->left,x)
end function
```

The bug is:

3. Search on a binary search tree with N nodes takes the following time in the worst case:

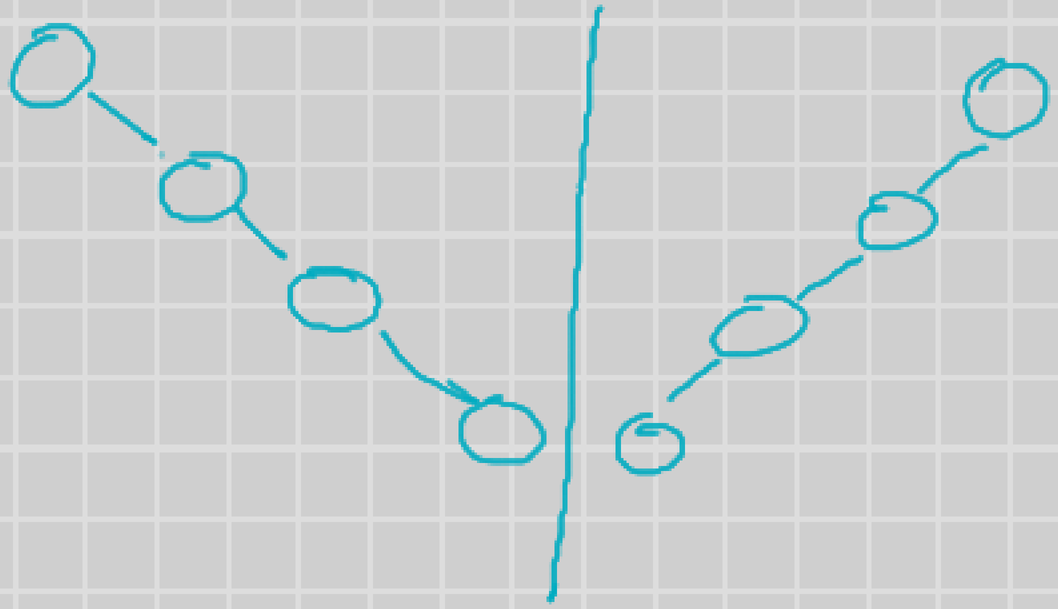
$\Theta\ N \log N$

$\Theta\ N$

$\Theta\ \log N$

$\Theta\ 1$

$\Theta\ N^2$



4. Search on a full binary search tree with N nodes takes the following time in the worst case:

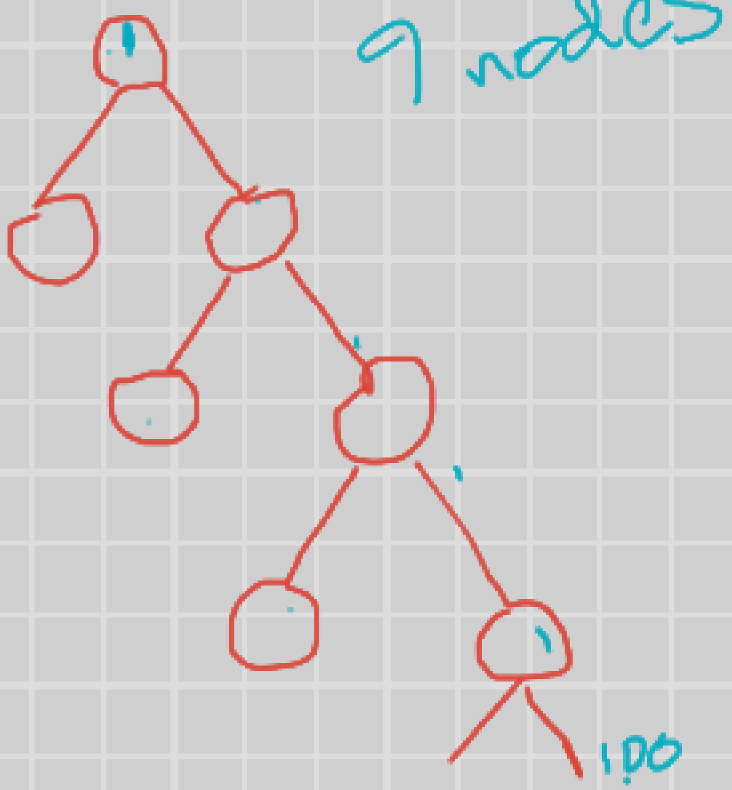
~~$O(\log N)$~~

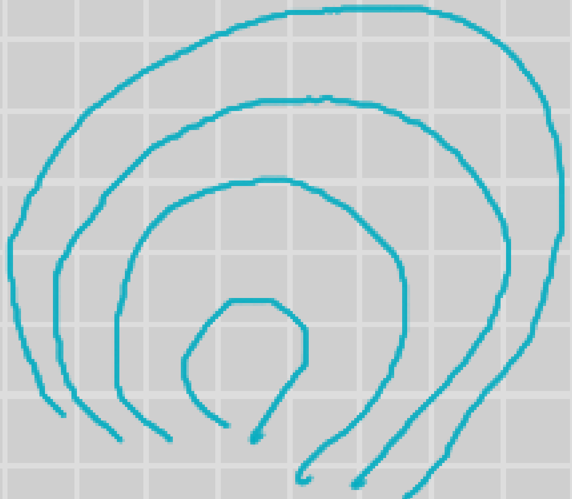
$O(N)$

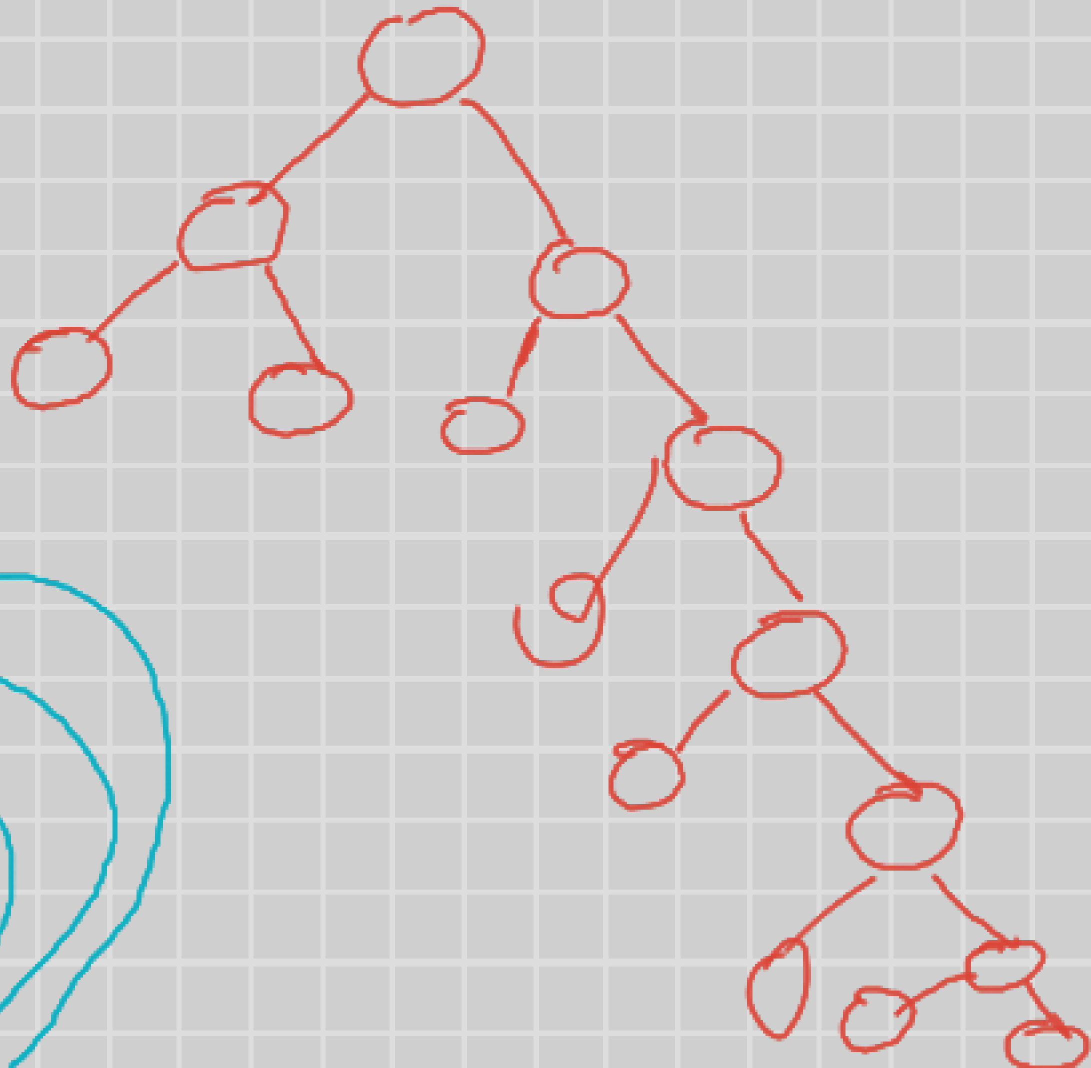
$O(N \log N)$

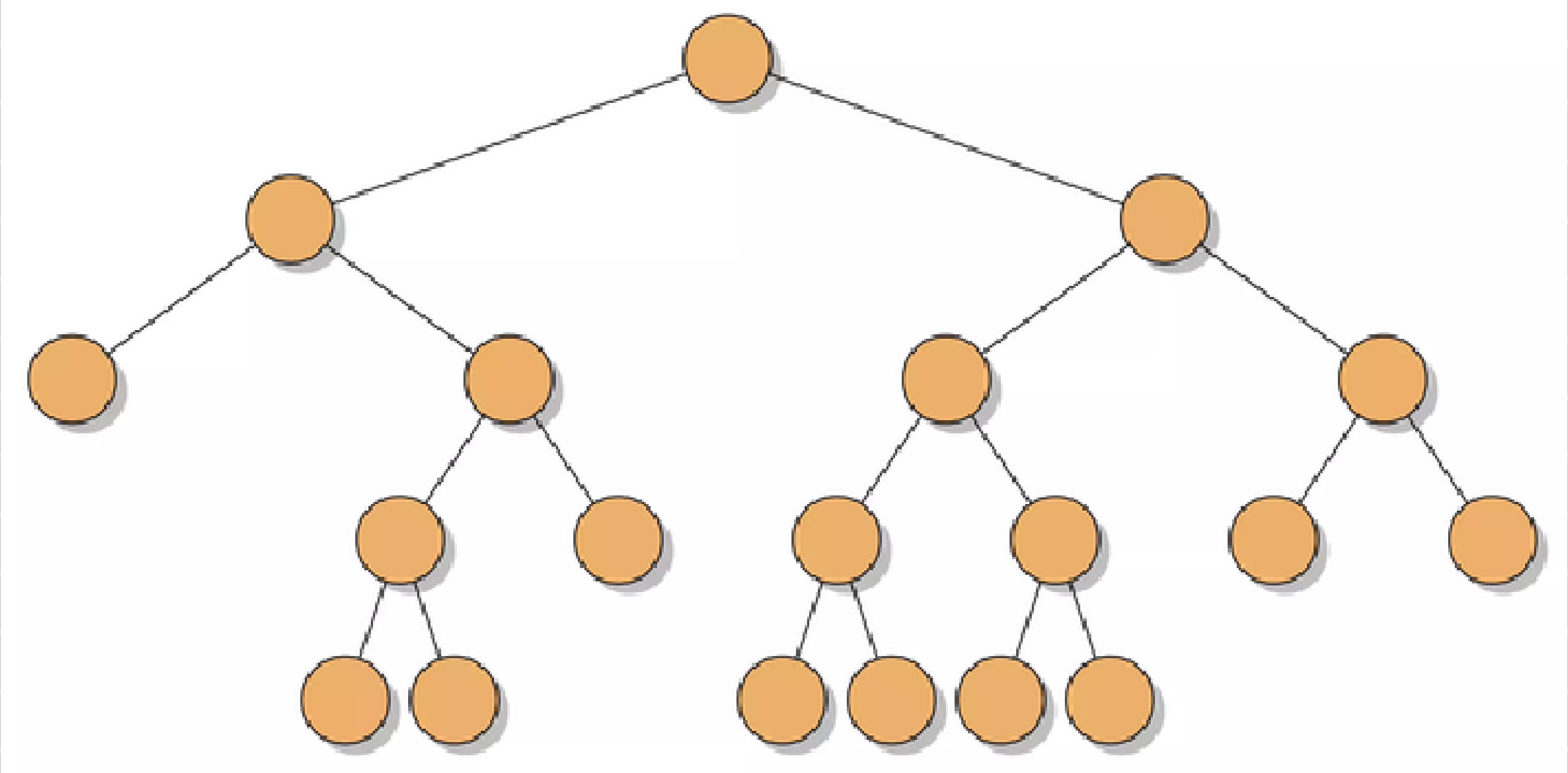
$O(1)$

$O(N^2)$

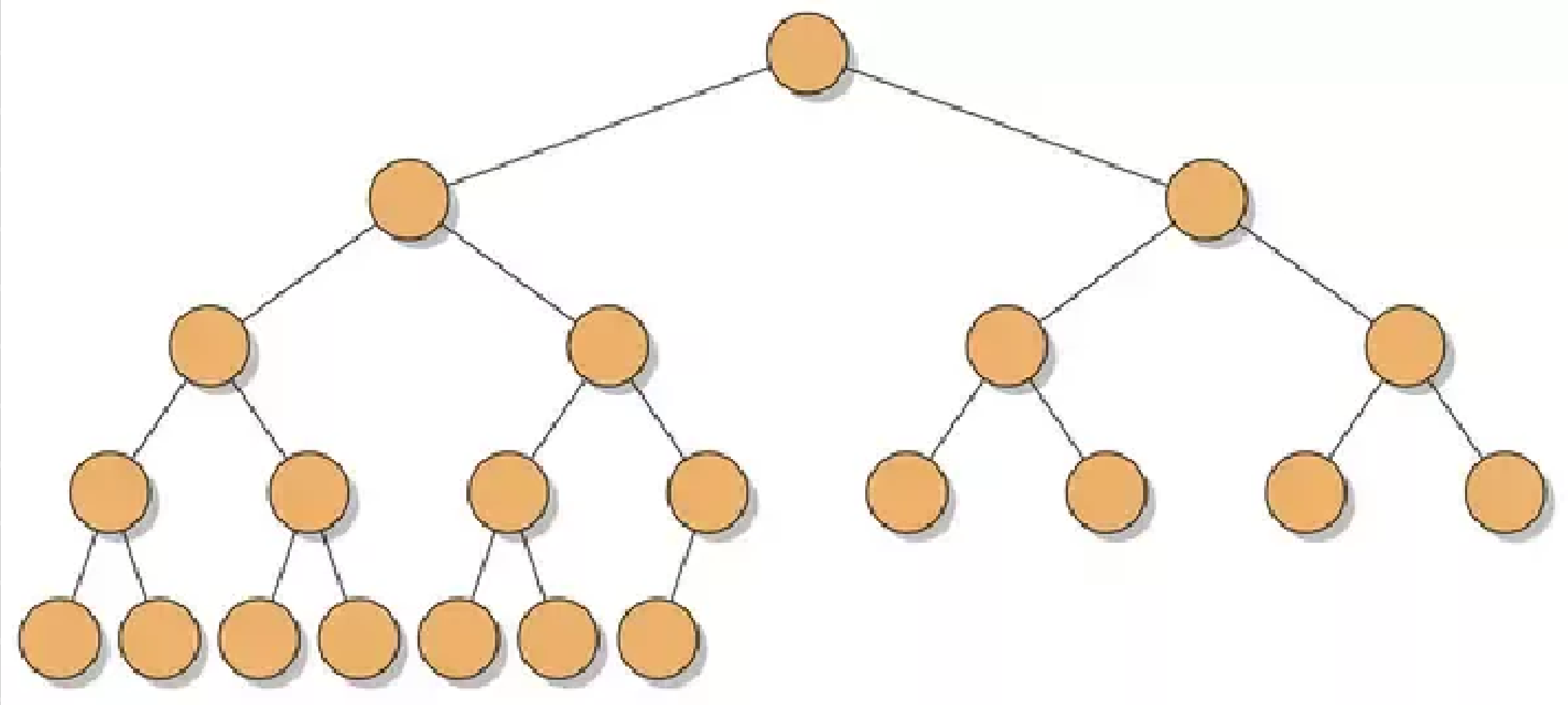








Full Binary Tree: All nodes have 0 or 2 children.



Complete Binary Tree: All levels exceot the last are completely filled. All nodes at the final level are to the left as much as possible

https://www.quora.com/What-is-the-difference-between-complete-and-full-binary-trees

4. Search on a full binary search tree with N nodes takes the following time in the worst case:

1 / 1 point

- [ ] $O(1)$
- [x] $O(N)$

> ✓ **Correct**
> Correct! All functions g(N) that grow faster than logN meet the Big-O notation condition

- [x] $O(N \log N)$

> ✓ **Correct**
> All functions g(N) that grow faster than logN meet the Big-O notation condition

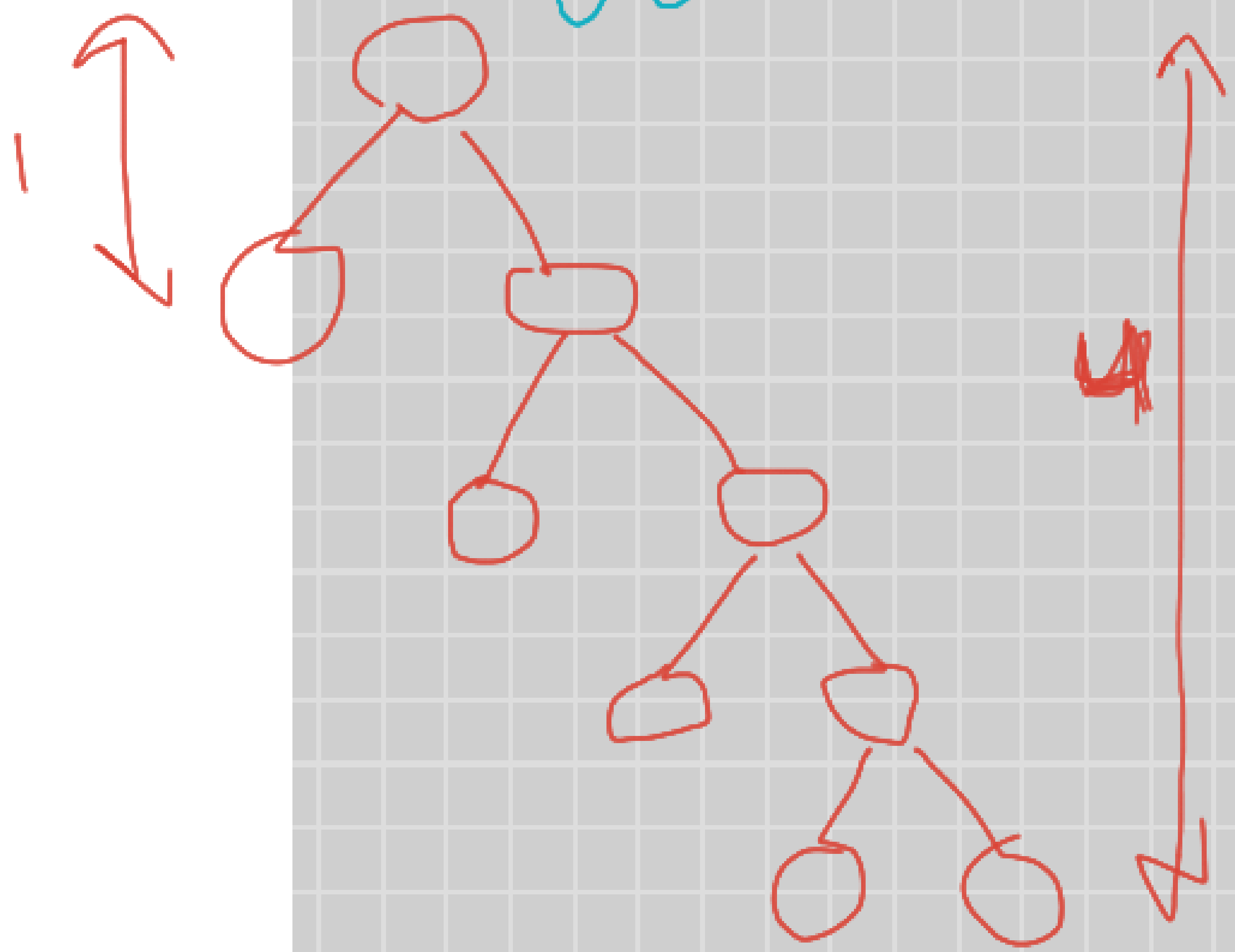
- [ ] None of the others
- [x] $O(\log N)$

> ✓ **Correct**
> Correct! A full binary search tree is a completely balanced tree

- [x] $O(N^2)$

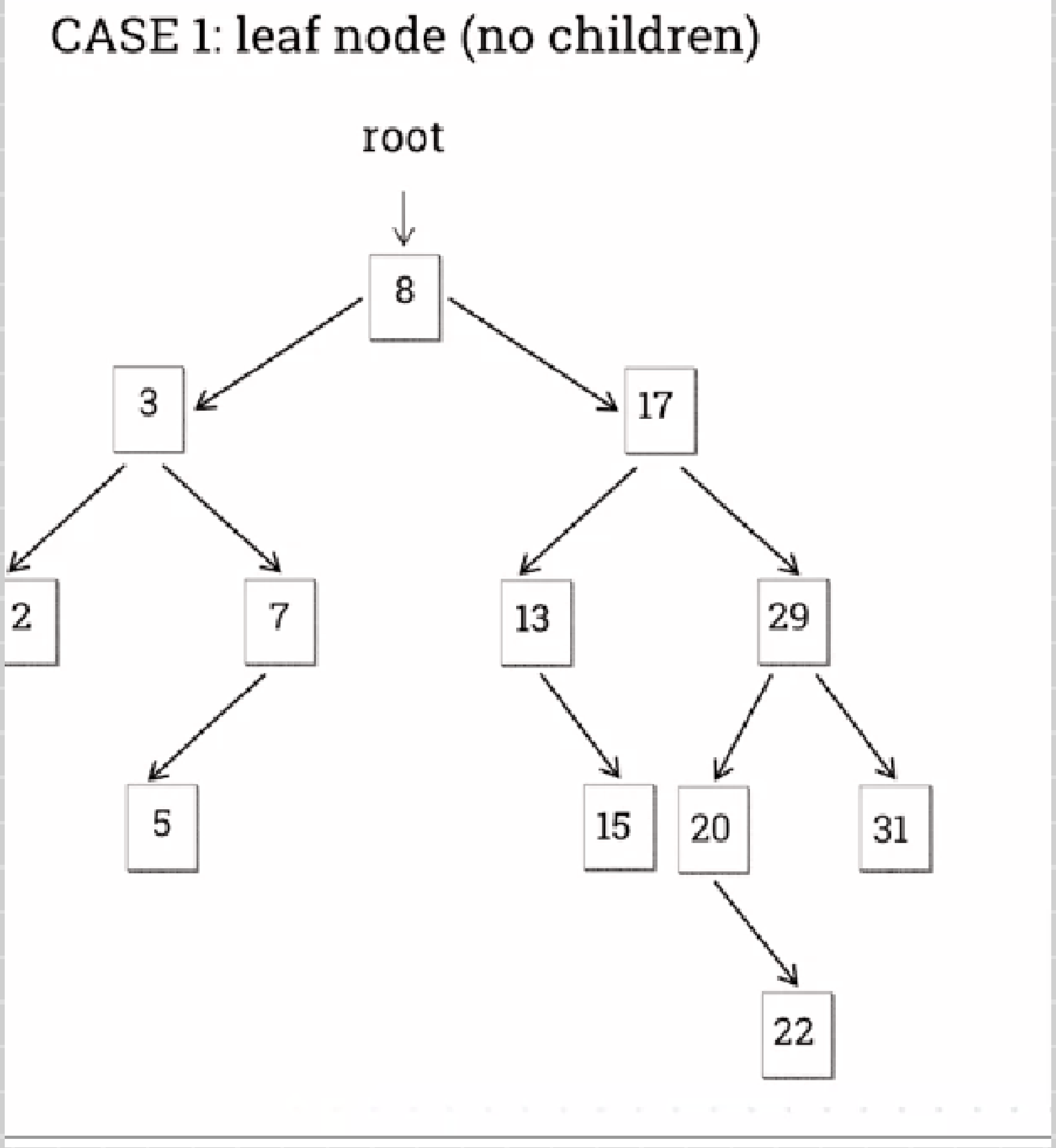
> ✓ **Correct**
> All functions g(N) that grow faster than logN meet the Big-O notation condition

Full binary tree.
0 or 2 nodes.



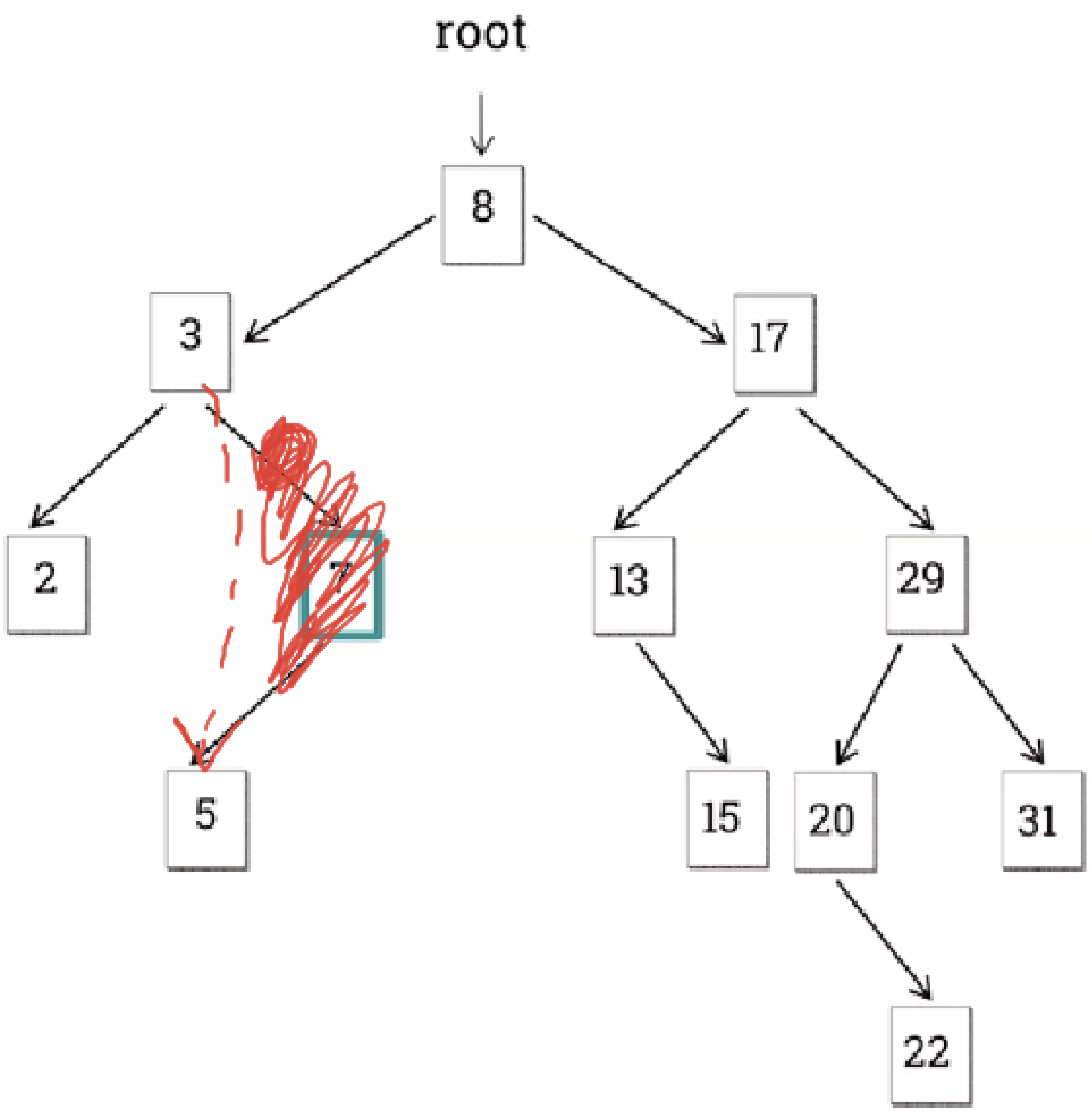
Not balanced because height difference between left and right is greater than 1.

BST Delete:

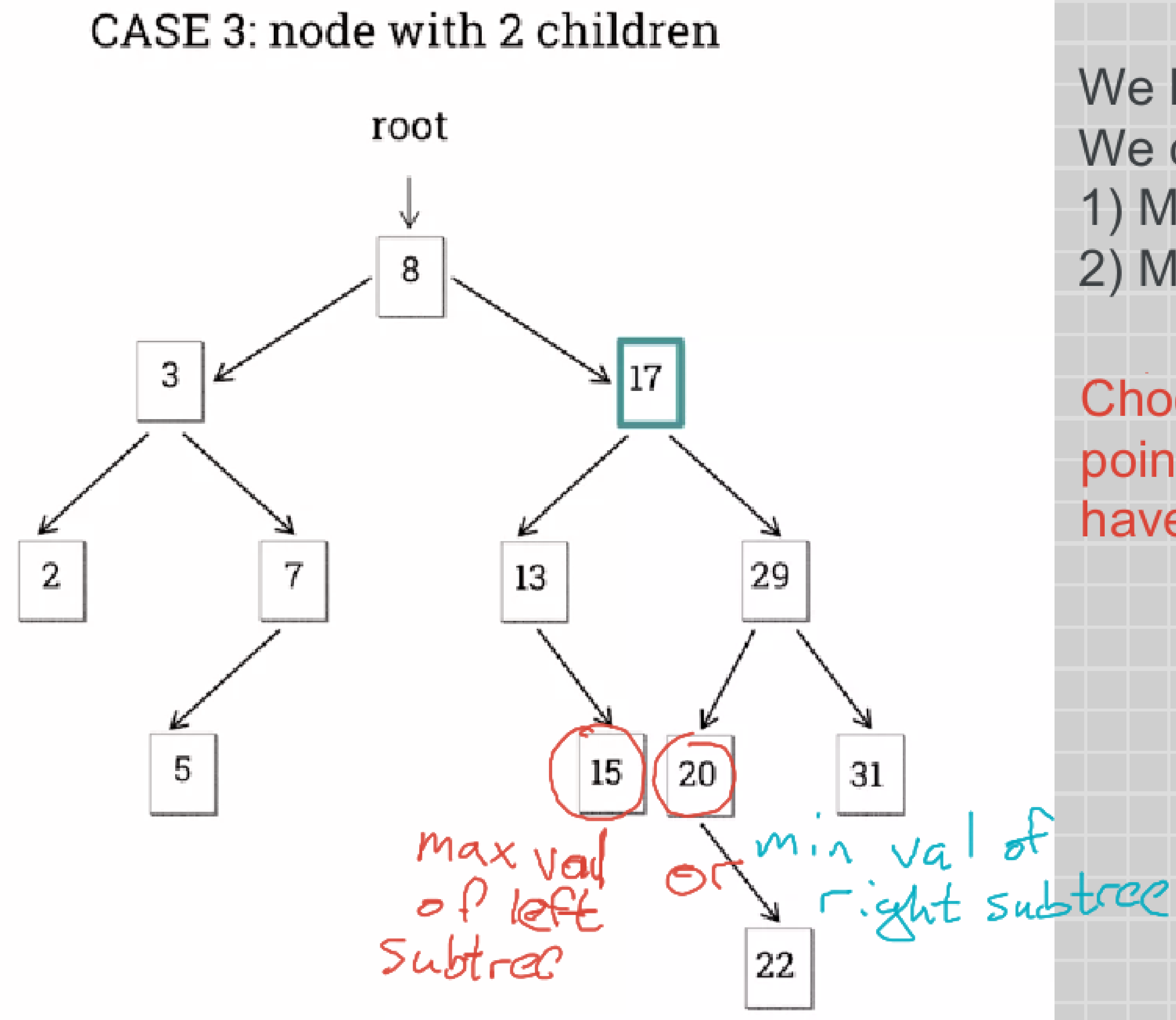


To delete 5 from the tree, we simply point node with value 7, left child to null.

CASE 2: node with only 1 child



To delete the node with value 7, we must point the parent node of 7 (which is 3) to the child of 7.



We have two choices.
We can replace the node being removed with the:
1) Minimum value from the right subtree
2) Max value of the left subtree

Choosing to replace with 15, means that we point the right pointer of number 8 to 15, and have the right pointer of 13 point to null.

Choosing to replace with 20 means that we point the right pointer of 8 to 20, the left pointer of 29 to 22, the left pointer of 20 to 13.

BST Delete psuedocode

```
function DELETE(ROOT, VALUE)
    if ROOT = null
        return null
    else
        if VALUE < ROOT.data
            ROOT.left <-- DELETE(ROOT.left, VAL
        else
            if VALUE > ROOT.data
                ROOT.right <-- DELETE(ROOT.right
            else  // node found
                if VALUE = ROOT.data
                    assign the parent node to null (eith
                else if
                    if
```

```
function DELTE_BST(root, x)
    if root = NULL then
        return NULL
    else
        if x < root.data then
            root.left ← DELETE_BST(root.left, x)
        else if x > root.data then
            root.right ← DELETE_BST(root.right, x)
        else
            if root.left = NULL and root.right = NULL then
                root ← NULL
                return root
            else if root.left = NULL then
                root ← root.right
                return root
            else if root.right = NULL then
                root ← root.left
                return root
            else
                tmp ← getRMin(root.right)
                root.data ← tmp.data
                root.right ←DELETE_BST(root.right, root.data)
            end if
        end if
    end if
end function
```