```
isBalanced() {
  return (this.findMinHeight() >= this.findMaxHeight() - 1)
}
```

IsBalanced calls the **findMinHeight** and sees if it's bigger than or equal to **findMaxHeight** minus 1.

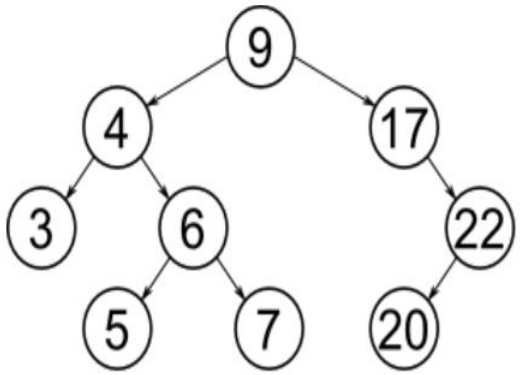
...so back to our first example:

-Minimum height: distance from the root node (**9**) to the first node with less than 2 children (**17**).

-Maximum height: distance from the root node (**9**) to a leaf node (**5,7** or **20**).

- The distance from **9** (h0) to **17** (h1) is 1.

- The distance from **9** (h0) to **20** (h3) is 3.




```
isBalanced() {
  return (this.findMinHeight() >= this.findMaxHeight() - 1)
}
```

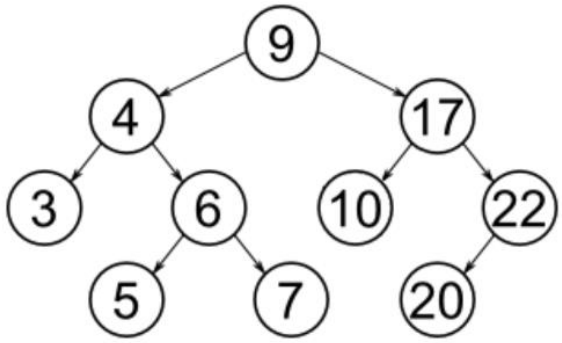
Is 1 greater than or equal to 3 minus 1? FALSE

-Minimum Height: distance from the root node (**9**) to the first node with less than 2 children (**10** or **22**)

-Maximum Height: distance from the root node (**9**) to a leaf node (**20**).

- The distance from **9** (h0) to **10** (h2) is 2.

- The distance from **9** (h0) to **20** (h3) is 3.




```
isBalanced() {
  return (this.findMinHeight() >= this.findMaxHeight() - 1)
}
```

Is 2 greater than or equal to 3 minus 1? TRUE

The **findMinHeight** is a recursive function:

```
}
findMinHeight(node = this.root) {
    if (node == null) {
        return -1;
    };
    let left = this.findMinHeight(node.left);
    let right = this.findMinHeight(node.right);
    if (left < right) {
        return left + 1;
    } else {
        return right + 1;
    };
```

You can pass in a node, but if not, it will be set as the root node.

It will check if it's a null, in which case it will return a –1.
If you haven't added anything to the search tree, it will return –1 for the height.

Set the left and right to calling the findMinHeight on both sides. This is where it gets recursive. Eventually, it will either be the left or the right node that returns –1. Why?... because either the left or right will be null.

If left is less than right, we will add +1 to the left.

Else, we add +1 to the right.

The **findMaxHeight** is similar, but opposite:

```
findMaxHeight(node = this.root) {
    if (node == null) {
        return -1;
    };
    let left = this.findMaxHeight(node.left);
    let right = this.findMaxHeight(node.right);
    if (left > right) {
        return left + 1;
    } else {
        return right + 1;
    };
}
```

...we return **left + 1** if left is greater than right.

We now get to the traversal part of this section. A binary tree can be traversed in three ways

(1) In-order

(2) Pre-order

(3) Post-order

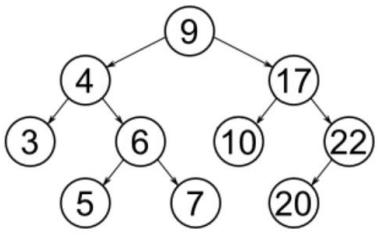
These three are actually quite similar to each other. Each of them will receive a starting node and a recurring function that the algorithm will use to traverse through the binary tree.

```
inOrder() {
  if (this.root == null) {
    return null;
  } else {
    var result = new Array();
    function traverseInOrder(node) {
      node.left && traverseInOrder(node.left);
      result.push(node.data);
      node.right && traverseInOrder(node.right);
    }
    traverseInOrder(this.root);
    return result;
  };
}
```

1. **inorder(node)** – It performs inorder traversal of a tree starting from a given *node*

   Algorithm for inorder:

   ```
   1. Traverse the left subtree i.e perform inorder on left subtree
   2. Visit the root
   3. Traverse the right subtree i.e perform inorder on right subtree
   ```



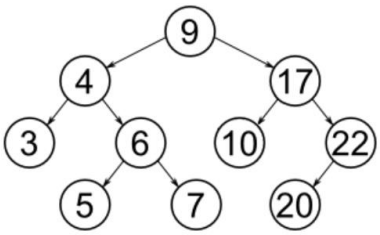

```
"inOrder: 3,4,5,6,7,9,10,17,20,22"
```

```
preOrder() {
  if (this.root == null) {
    return null;
  } else {
    var result = new Array();
    function traversePreOrder(node) {
      result.push(node.data);
      node.left && traversePreOrder(node.left);
      node.right && traversePreOrder(node.right);
    };
    traversePreOrder(this.root);
    return result;
  };
}
```

2. **preorder(node)** – It performs preorder traversal of a tree starting from a given *node*.

Algorithm for preoder:

```
1. Visit the root
2. Traverse the left subtree i.e perform inorder on left subtree
3. Traverse the right subtree i.e perform inorder on right subtree
```



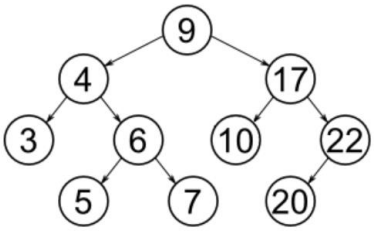

```
"preOrder: 9,4,3,6,5,7,17,10,22,20"
```

```
postOrder() {
  if (this.root == null) {
    return null;
  } else {
    var result = new Array();
    function traversePostOrder(node) {
      node.left && traversePostOrder(node.left);
      node.right && traversePostOrder(node.right);
      result.push(node.data);
    };
    traversePostOrder(this.root);
    return result;
  }
}
```

3. **postorder(node)** – It performs postorder traversal of a tree starting from a given *node*.

   Algorithm for postorder:

```
1. Traverse the left subtree i.e perform inorder on left subtree
2. Traverse the right subtree i.e perform inorder on right subtree
3. Visit the root
```

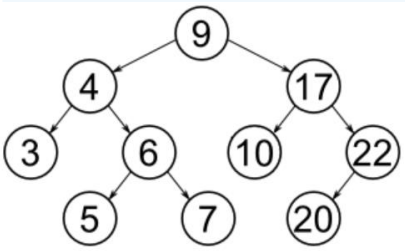



```
"postOrder: 3,5,7,6,4,10,20,22,17,9"
```

```
  levelOrder() {
      let result = [];
      let Q = [];
      if (this.root != null) {
          Q.push(this.root);
          while(Q.length > 0) {
              let node = Q.shift();
              result.push(node.data);
              if (node.left != null) {
                  Q.push(node.left);
              };
              if (node.right != null) {
                  Q.push(node.right);
              };
          };
          return result;
      } else {
          return null;
      };
  };
}
```

A level-order traversal on a tree performs the following steps starting from the root:

1) Add the root to a queue.
2) Pop the last node from the queue, and return its value.
3) Add all children of popped node to queue, and continue from step 2 until queue is empty.




```
"levelOrder: 9,4,17,3,6,10,22,5,7,20"
```