Binary Trees, Recursion and Tail Call Optimization in Javascript

Use recursion to determine the height of a binary tree. You must figure out how to implement the binary tree in JavaScript.

We are covering each aspect of the job interviewing process. You have already seen some theoretical questions that demonstrate how well you can use JavaScript. You have seen some coding challenges that not only let you showcase your problem-solving abilities, but also demonstrate your theoretical knowledge and your algorithmic skills.

You are yet to experience some longer homework assignment type of tasks that challenge your abilities to write maintainable software. Some of these challenges are timed, some require you to use some frameworks or libraries, while others need you to structure your code.

The challenge I have chosen for this session is an online coding challenge on a site called HackerRank.

I have already recommended that you go through the challenges of a similar site called Codility. HackerRank ups the ante a bit more by giving you problems of continuously increasing difficulty.

Once you sign up, HackerRank recommends a thirty-day challenge for you. You get one exercise a day, which often takes just a couple of minutes. The thirty-day challenge is very healthy, because it builds a habit of coding just a bit every single day.

Consider the option of using challenges like the ones HackerRank provides to improve your problem-solving skills.

As an illustration, I will now solve a coding challenge that you can find in the Data Structures section of HackerRank. The problem is called Height of a Binary Tree.

For advanced positions, you will be expected to know some data structures and algorithms, as well as some programming techniques like pure functional programming and recursion. We will build on some of this knowledge.

You can either read the task by signing up on HackerRank and visiting the link, or by reading my summary here:

Suppose a binary tree is given with root `R`*. Each node may be connected to zero, one, or two child nodes. The edges of the tree are directed from the parent nodes towards child nodes. Determine the height of the tree, defined as the maximal number of edges from* `R` *to any node in the tree.*

The JavaScript data structure of a node is as follows:

```
type Node = {
  data: number,
  left: Node | null,
  right: Node | null
}
```

Solution:

Let's sketch a plan:

- The height of a tree with one node with no children is `0`
- The height of a tree with a left and a right subtree is `1 plus the maximum of the height of the left subtree and the right subtree`

These are all the ideas you need to demonstrate to be able to solve this exercise. Let's create a solution function.

```
const treeHeight = tree =>
    Math.max(
        tree.left === null ? 0 : 1 + treeHeight( tree.left ),
        tree.right === null ? 0 : 1 + treeHeight( tree.right )
    );
```

That's it. We have solved this exercise with recursion.

Notice the solution is purely functional, as it is not relying on any side-effects. Also, notice the elegance of the solution in a sense that we just described the

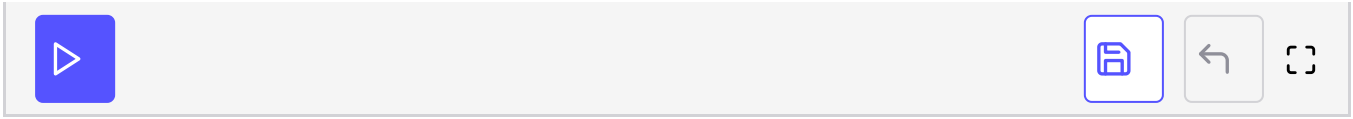
input (`tree`) and the return value.

Now, your interviewer may ask you to solve this exercise without recursion. Remember, for every recursive solution there exists an equivalent iterative solution. To find the iterative solution, we need to save the upcoming recursive calls in a queue-like data structure (a simple array will do), and introduce some *accumulator variables* that store the current state of the computation.

Let's start writing the frame of substituting recursion:

```
const treeHeight = root => {
    let nodes = [{ root, distance: 0 }];
    let maxHeight = 0;

    while ( nodes.length > 0 ) {
        let node = nodes.pop();
        // ...
    }
}
```



We put the tree in a data structure where we save the distance from the root. We also initialize the maximum height of the tree to zero.

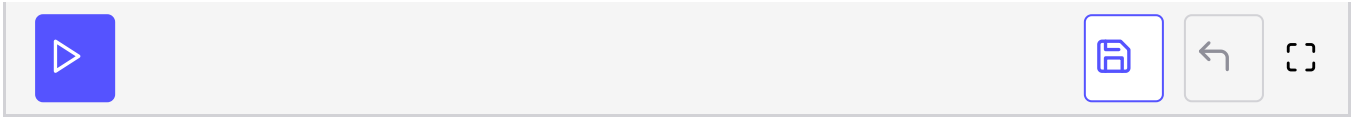
Instead of recursion, we have a while loop. As long as there are nodes in the `nodes` array, we pop one, and process its consequences. During the processing, we may push more nodes to the array:

```
const treeHeight = root => {
    let nodes = [{ node: root, distance: 0 }];
    let maxHeight = 0;

    while ( nodes.length > 0 ) {
        let currentTree = nodes.pop();
        maxHeight = Math.max( maxHeight, currentTree.distance );
        if ( currentTree.node.left !== null ) {
            nodes.push( {
                node: currentTree.node.left,
                distance: currentTree.distance + 1
            } );
        }
        if ( currentTree.node.right !== null ) {
            nodes.push( {
                node: currentTree.node.right,
                distance: currentTree.distance + 1
            } );
```

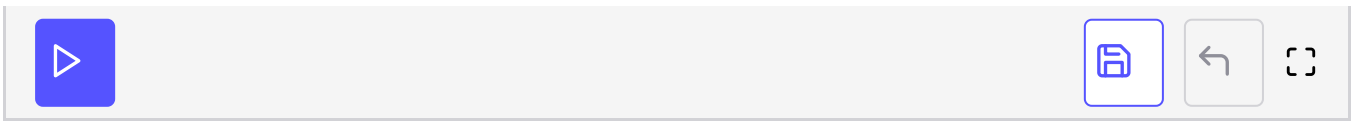
```
        }
    }

    return maxHeight;
}
```



Now that you have completed the iterative solution, a common question is whether you can write a recursive solution that is tail call optimized. Let's see the original recursive solution:

```
const treeHeight = tree =>
    Math.max(
        tree.left === null ? 0 : 1 + treeHeight( tree.left ),
        tree.right === null ? 0 : 1 + treeHeight( tree.right )
    );
```



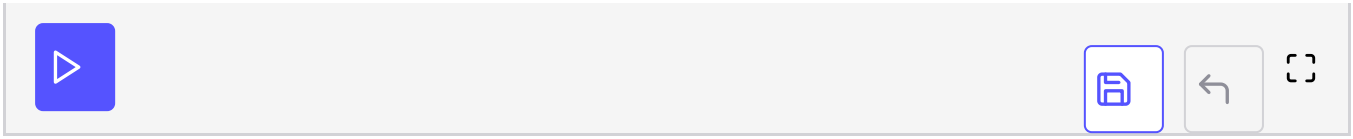
The recursive calls are inside `Math.max`, so they are not in tail position. We have to extract them out from the `Math.max`. The question is how.

The iterative solution always gives you an idea for tail recursion. Even if you are unsure about the exact definition of *tail position* for recursive function calls, you can take the state space of the iterative function and implement the while loop using recursion:

```
const treeHeight = root =>
    treeHeightRecursive( [{ node: root, distance: 0}], 0 );

const treeHeightRecursive = ( nodes, maxHeight ) => {
    let currentTree = nodes.pop();
    maxHeight = Math.max( maxHeight, currentTree.distance );
    if ( currentTree.node.left !== null ) {
        nodes.push( {
            node: currentTree.node.left,
            distance: currentTree.distance + 1
        } );
    }
    if ( currentTree.node.right !== null ) {
        nodes.push( {
            node: currentTree.node.right,
            distance: currentTree.distance + 1
        } );
    }
    if ( nodes.length === 0 ) return maxHeight;
    return treeHeightRecursive( nodes, maxHeight );
```

```
}
```



One minor difference concerning the iterative solution is that we have to manually create an exit condition from recursion with the condition `nodes.length === 0`.