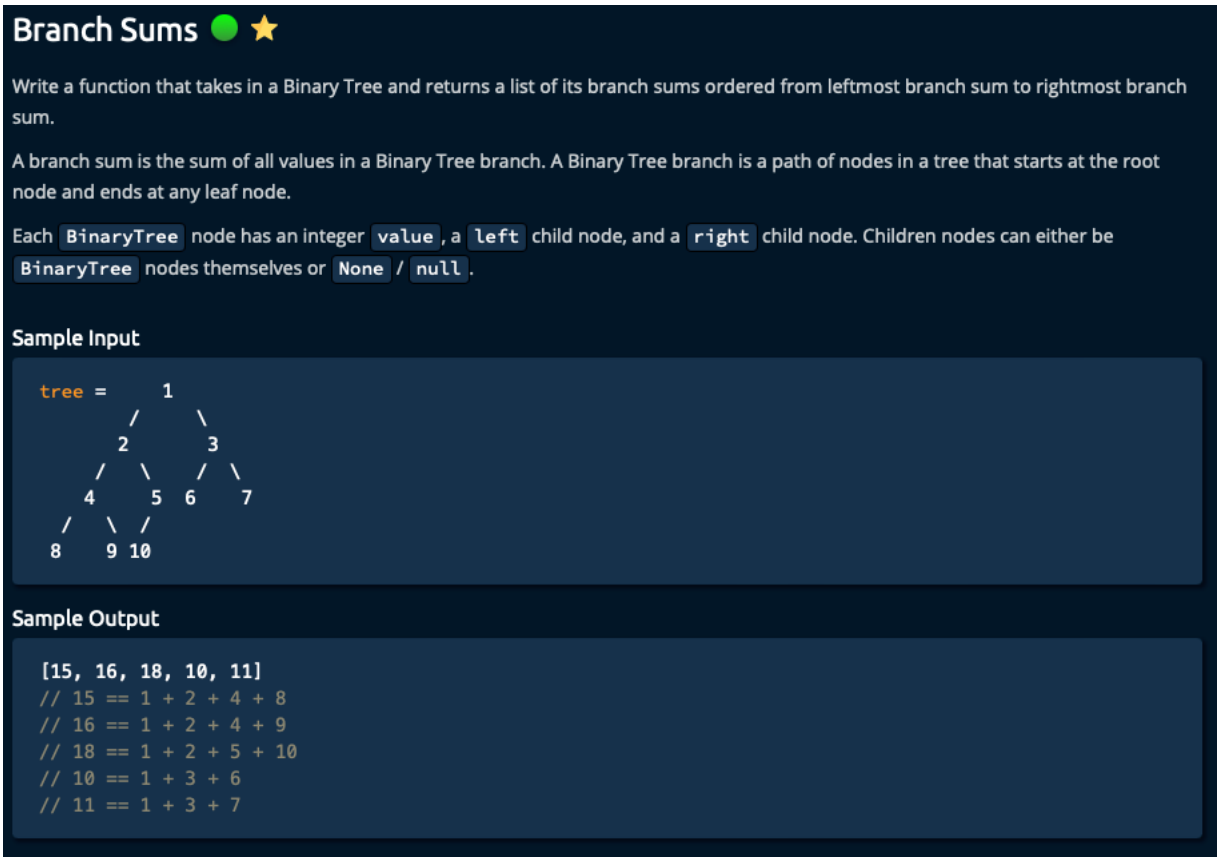
Code Challenge #5 Branch Sums (Easy)



Branch Sums

Write a function that takes in a Binary Tree and returns a list of its branch sums ordered from leftmost branch sum to rightmost branch sum.

A branch sum is the sum of all values in a Binary Tree branch. A Binary Tree branch is a path of nodes in a tree that starts at the root node and ends at any leaf node.

Each `BinaryTree` node has an integer `value`, a `left` child node, and a `right` child node. Children nodes can either be `BinaryTree` nodes themselves or `None` / `null`.

Sample Input

```
tree =     1
         /   \
        2     3
      /  \   /  \
     4    5 6    7
   /  \  /
  8   9 10
```

Sample Output

```
[15, 16, 18, 10, 11]
// 15 == 1 + 2 + 4 + 8
// 16 == 1 + 2 + 4 + 9
// 18 == 1 + 2 + 5 + 10
// 10 == 1 + 3 + 6
// 11 == 1 + 3 + 7
```

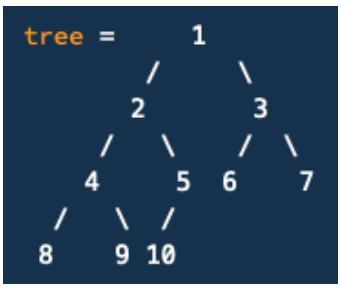
Solution

```
1. class BinaryTree {
2.   constructor(value) {
3.     this.value = value;
4.     this.left = null;
5.     this.right = null;
6.   }
7. }
8.
9. function branchSums(root) {
10.
11.    const sums = []
12.         calculateBranchSums( root, 0, sums);
13.         return sums;
14. }
15.
16. function calculateBranchSums(node, runningSum, sums) {
```

```
17.
18.   if (!node) return;
19.
20.         const newRunningSum = runningSum + node.value;
21.
22.         if (!node.left && !node.right) {
23.                 sums.push(newRunningSum);
24.                 return;
25.         }
26.
27.         calculateBranchSums(node.left, newRunningSum, sums);
28.
29.         calculateBranchSums(node.right, newRunningSum, sums);
30.
31. }
```

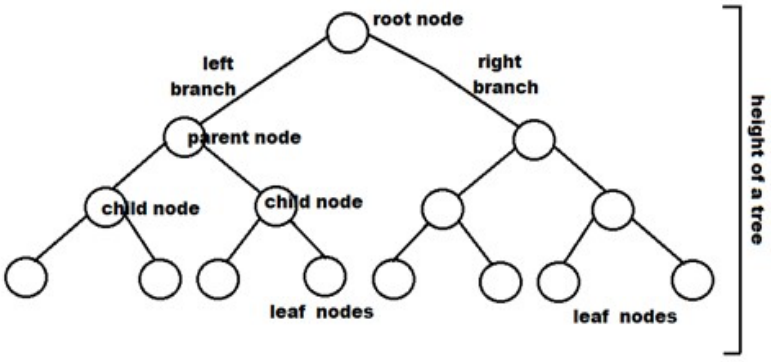
Explanation

The Branch Sums problem is based on a binary tree structure where we want to find the sum of all the different branches of the tree.




```
[15, 16, 18, 10, 11]
// 15 == 1 + 2 + 4 + 8
// 16 == 1 + 2 + 4 + 9
// 18 == 1 + 2 + 5 + 10
// 10 == 1 + 3 + 6
// 11 == 1 + 3 + 7
```

Binary Tree Structure Labeled



Iterating through the binary tree structure is usually done with a recursive approach since we do not know the depth of the binary tree structure. Using recursion helps us through iterate through the binary tree structure and terminate only when we get to the base case of the recursive function. In order to solve this problem, we will use a helper method which acts as the recursive function. This recursive function will be part of the main function called branchSums.

In branchSums function we take in an argument called root. The root argument represents the root of the binary tree. In the branchSums function we create a variable called const sum which equals an empty array. The next line of the function we call the recursive function calculateBranchSums with three arguments. The tree arguments are root, 0 and sums, zero presents an initial starting value for the branch sums. The final part of the function is the return of sums.

The recursive function calculateBranchSums is defined with three arguments which are node, runningSum and sums. The node represented the binary tree, runningSum represents the total for each particular branch and the sums represents an array which contains a list of the total values for each branch of the binary tree. In the recursive function we first put an edge case where if we a experience a child node of null or falsey value we break the function using a return statement. The next line of the function is a constant variable called newRunningSum which is equal to the runningSum (started off as 0 previously) plus the node.value which is the value found in the Node branches. The next part of the recursive function is where we check to see if there are no child branches left on the left side and the right side. If there aren't any then we have reached the leaf of the binary tree (end) and we can no longer iterate through the branches.

In this particular situation we push the newRunningSum in the sums array and break the function by adding a return. If we don't get to this situation, we continue to recursively call the function twice on both the left and right branches. This calling of the recursive function on both branches one and a type helps us to traverse the binary tree in a cascading effect because of how a recursive function stores memory in call stacks and finishes all possible eventualities. Since we call the function recursively on the left side first and then the right side the left side will finish first before the right side. If we reversed the recursive function call where we called the right side first, then the left side the right side will finish first. The cascading effect of the recursive function is the beauty of using recursion to solve this type of tree traversal problems.