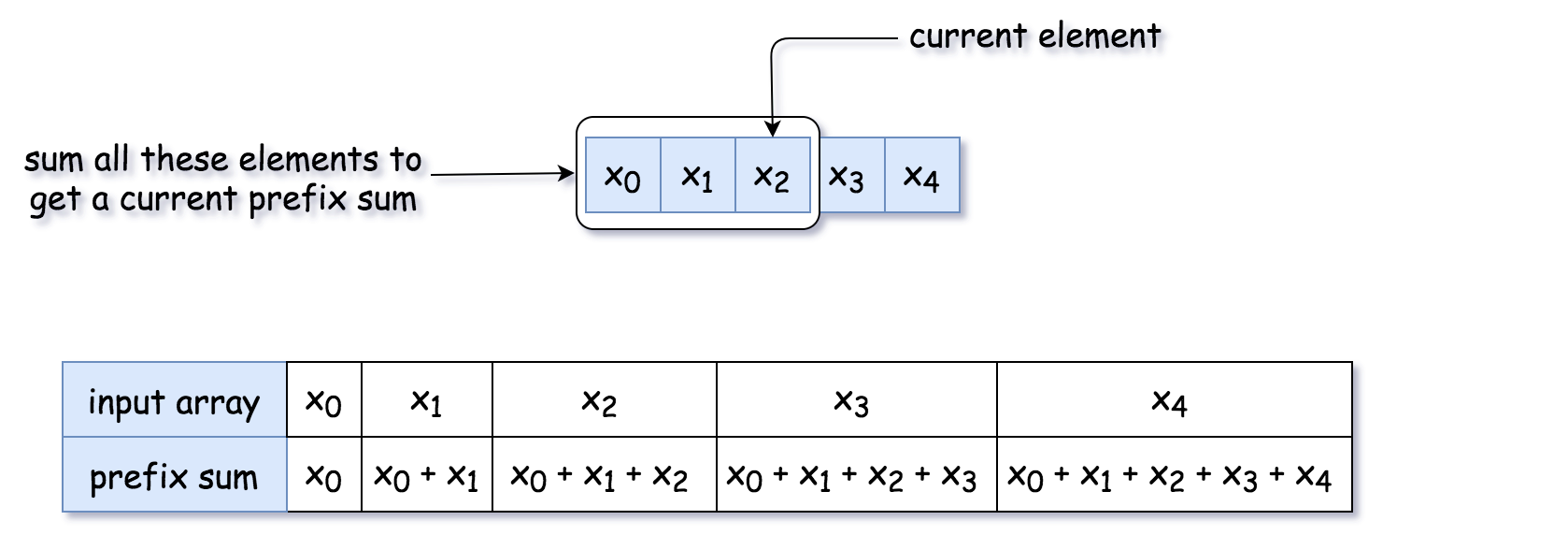
Solution

Prefix Sum: What is it

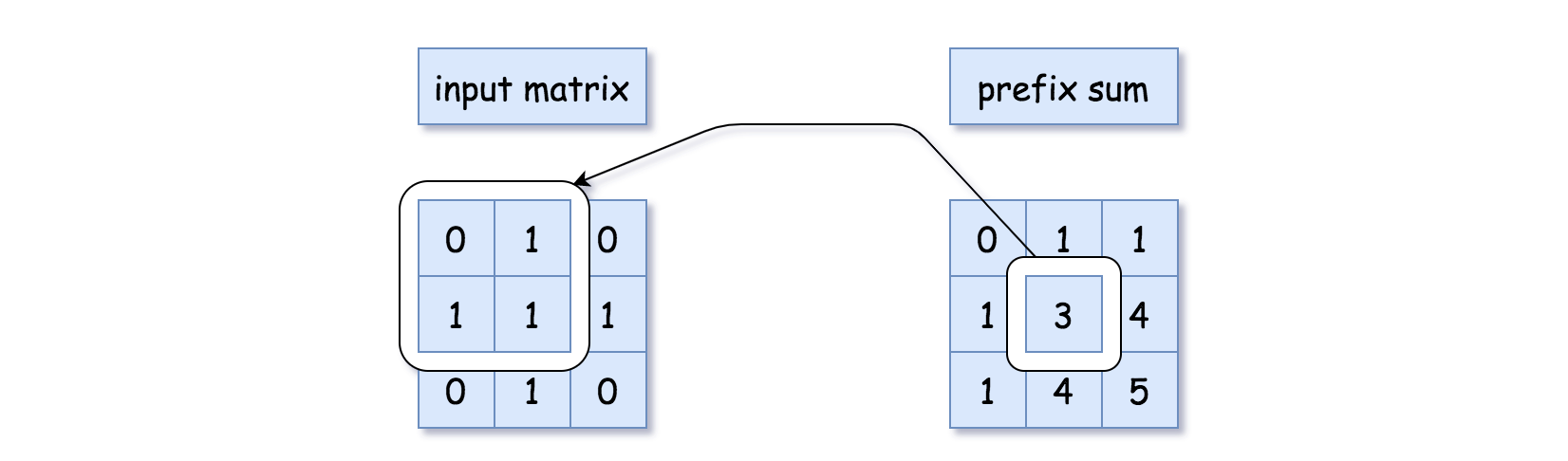
In this article, we're going to discuss simple but powerful [prefix sum technique](https://en.wikipedia.org/wiki/Prefix_sum): one pass + linear time complexity.

*Prefix sum* is a sum of the current value with all previous elements starting from the beginning of the structure.

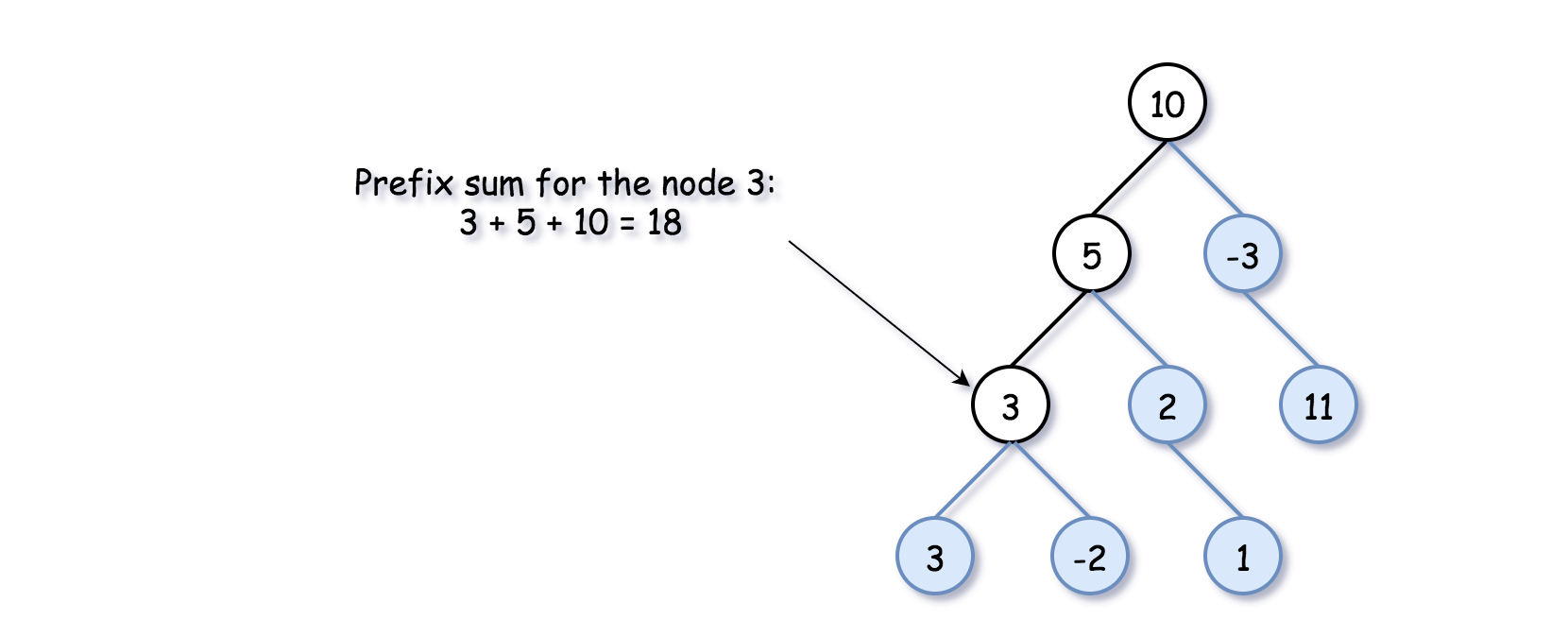
It could be defined for 1D arrays (sum the current value with all the previous integers),

 *Figure 1. Prefix sum for 1D array.*

for 2D arrays (sum of the current value with the integers above or on the left)

 *Figure 2. Prefix sum for 2D array.*

or for the binary trees (sum the values of the current node  
and all parent nodes),

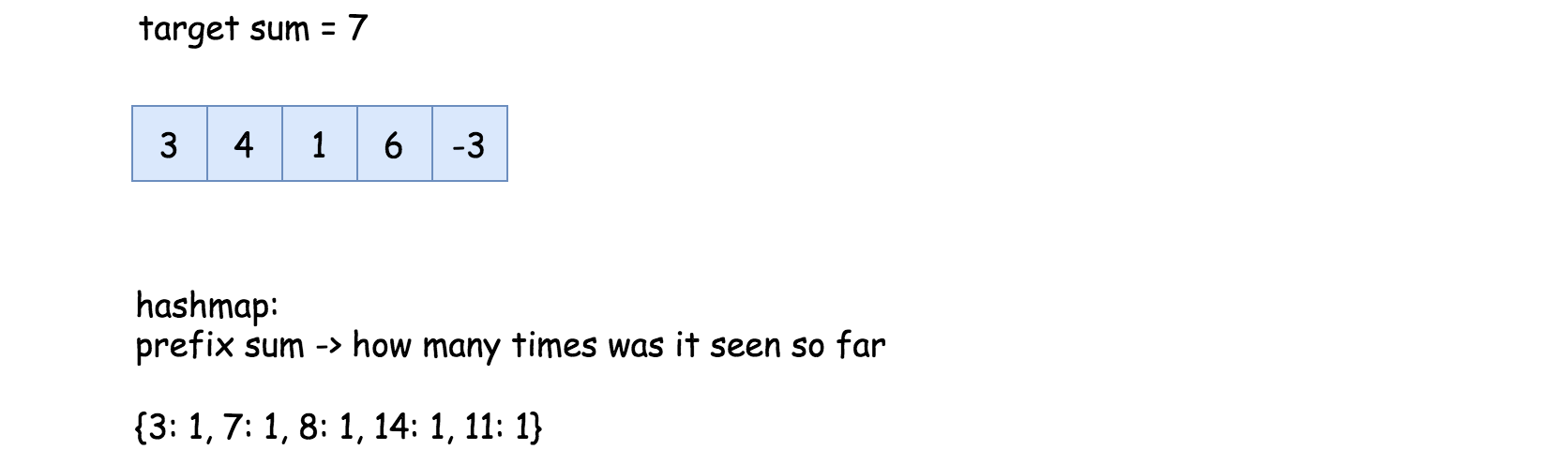
 *Figure 3. Prefix sum for the binary tree.*

Prefix Sum: How to Use: Number of Continuous Subarrays that Sum to Target

You might want to use the prefix sum technique for the problems like "Find a number of *continuous* subarrays/submatrices/tree paths that sum to target".

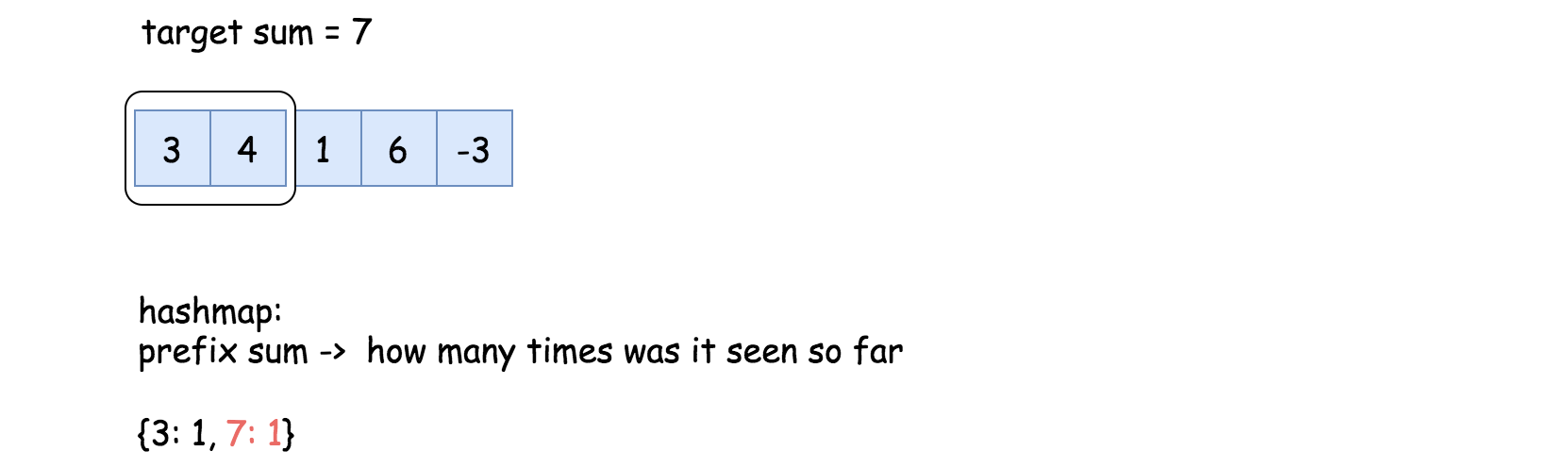
Before going to the current problem with the tree, let's check the idea on a simpler example [Find a number of continuous subarrays that sum to target](https://leetcode.com/problems/subarray-sum-equals-k/).

* Use a variable to track the current prefix sum and a hashmap "prefix sum -> how many times was it seen so far".

 *Figure 4. Find a number of continuous subarrays that sum to target.*

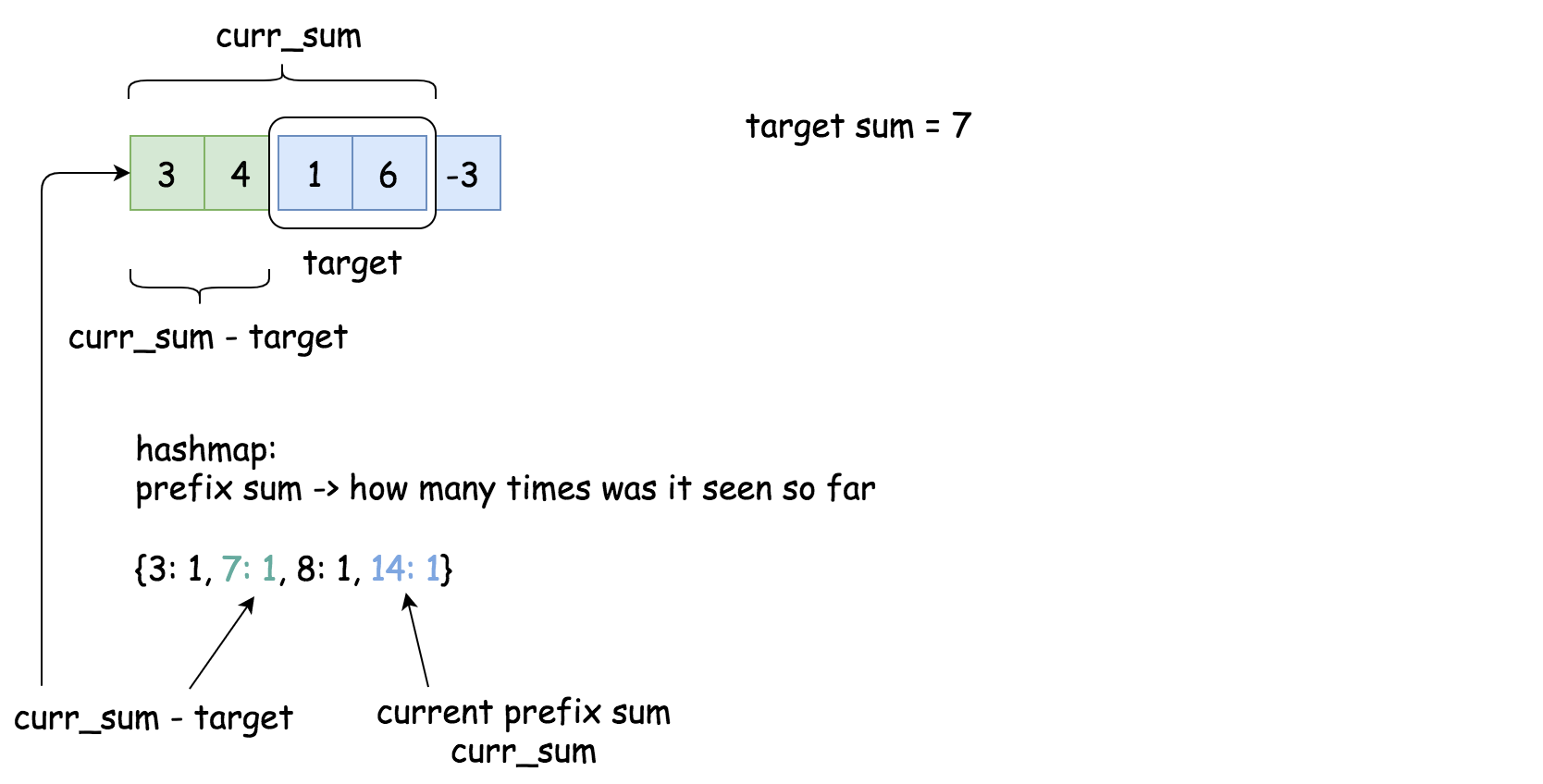
* Parse the input structure and count the requested subarrays/submatrices/tree paths along the way with the help of that hashmap. How to count?

There could be two situations. In situation 1, the subarray with the target sum starts from the beginning of the array. That means that the current prefix sum is equal to the target sum, and we increase the counter by 1.

 *Figure 5. Situation 1: The subarray starts from the beginning of the array.*

In situation 2, the subarray with the target sum starts somewhere in the middle. That means we should add to the counter the number of times we have seen the prefix sum curr\_sum - target so far: count += h[curr\_sum - target].

The logic is simple: the current prefix sum is curr\_sum, and some elements before the prefix sum was curr\_sum - target. All the elements in between sum up to curr\_sum - (curr\_sum - target) = target.

 *Figure 6. Situation 2: The subarray starts somewhere in the middle.*

**Solution for Number of Continuous Subarrays that Sum to Target**

Here is an implementation of the solution for [Find a number of continuous subarrays that sum to target](https://leetcode.com/problems/subarray-sum-equals-k/).

Approach 1: Prefix Sum

Now let's reuse the same algorithm and the same code for the case of the binary tree.

There is just one thing that is particular for the binary tree. There are two ways to go forward - to the left and to the right. To keep parent->child direction, we shouldn't blend prefix sums from the left and right subtrees in one hashmap.

**Algorithm**

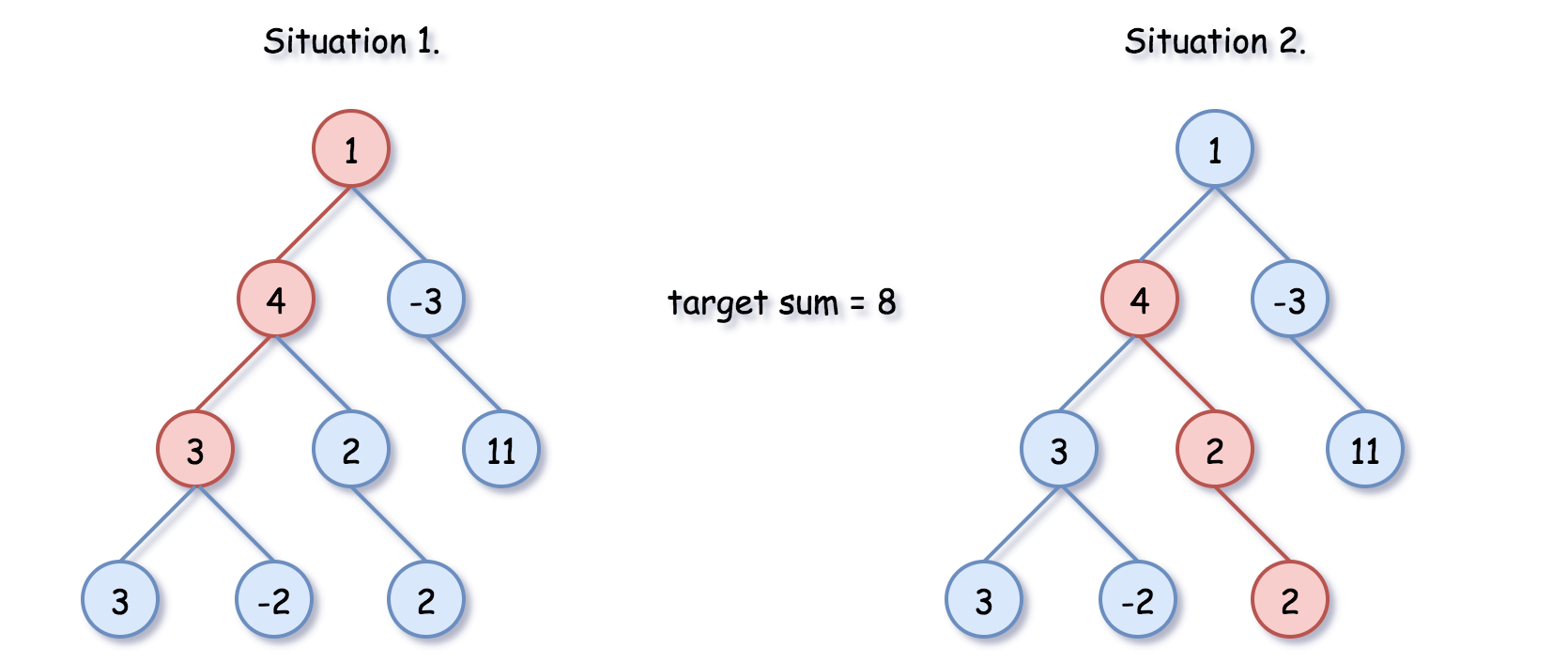
* Let's initialize tree paths counter count = 0, and the hashmap h "prefix sum -> how many times was it seen so far".
* One could parse the tree using [recursive preorder traversal](https://leetcode.com/articles/sum-root-to-leaf-numbers/): node -> left -> right: preorder(node: TreeNode, curr\_sum: int) -> None. This function takes two arguments: a tree node and a prefix sum before that node. To start the recursion chain, one should call preorder(root, 0).
  + The first thing is to update the current prefix sum by adding the value of the current node: curr\_sum += node.val.
  + Now one could update the counter. One should consider two situations.

In situation 1, the tree path with the target sum starts from the root. That means the current prefix sum is equal to the target sum curr\_sum == k, so one should increase the counter by 1: count += 1.

In situation 2, the tree path with the target sum starts somewhere downwards. That means we should add to the counter the number of times we have seen the prefix sum curr\_sum - target so far: count += h[curr\_sum - target].

The logic is simple: the current prefix sum is curr\_sum, and several elements before the prefix sum was curr\_sum - target. All the elements in between sum up to curr\_sum - (curr\_sum - target) = target.

* + Now it's time to update the hashmap: h[curr\_sum] += 1.
  + Let's parse left and right subtrees: preorder(node.left, curr\_sum), preorder(node.right, curr\_sum).
  + Now the current subtree is processed. It's time to remove the current prefix sum from the hashmap, in order not to blend the parallel subtrees: h[curr\_sum] -= 1.
* Now the preorder traversal is done, and the counter is updated. Return it.

 *Figure 7. Situation 1 vs situation 2.*

**Implementation**

On the following example the target sum is equal to 8.

Diagram

Description automatically generated

9 / 17

**Complexity Analysis**

* Time complexity: \mathcal{O}(N)O(*N*), where N*N* is a number of nodes. During preorder traversal, each node is visited once.
* Space complexity: up to \mathcal{O}(N)O(*N*) to keep the hashmap of prefix sums, where N*N* is a number of nodes.