Binary Tree Path Sum (easy)

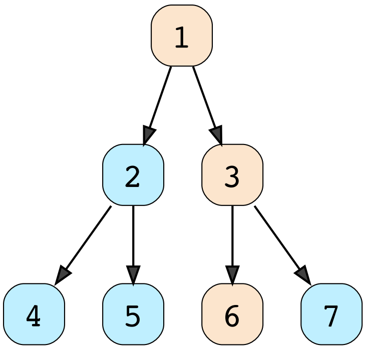
We'll cover the following

Problem Statement

Given a binary tree and a number ‘S’, find if the tree has a path from root-to-leaf such that the sum of all the node values of that path equals ‘S’.

Example 1:

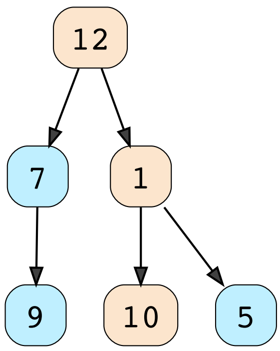
S: 10
Output: true
Explanation: The path with sum '10' is highlighted



Example 2:

S: 23
Output: true
Explanation: The path with sum '23' is highlighted

S: 16
Output: false
Explanation: There is no root-to-leaf path with sum '16'.



Try it yourself

Try solving this question here:

Java | Python3 | JS | C++

```
class TreeNode:
  def __init__(self, val, left=None, right=None):
    self.val = val
    self.left = left
    self.right = right


def has_path(root, sum):
  # TODO: Write your code here
  return False

def main():

  root = TreeNode(12)
  root.left = TreeNode(7)
  root.right = TreeNode(1)
  root.left.left = TreeNode(9)
  root.right.left = TreeNode(10)
  root.right.right = TreeNode(5)
  print("Tree has path: " + str(has_path(root, 23)))
  print("Tree has path: " + str(has_path(root, 16)))


main()

```

Solution

As we are trying to search for a root-to-leaf path, we can use the **Depth First Search (DFS)** technique to solve this problem.

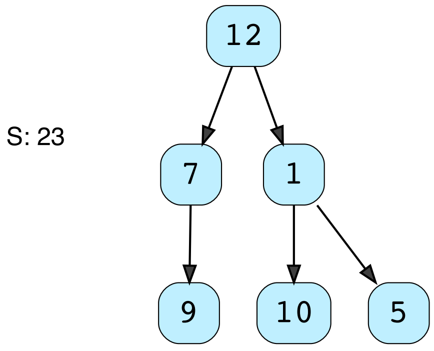
To recursively traverse a binary tree in a DFS fashion, we can start from the root and at every step, make two recursive calls one for the left and one for the right child.

Here are the steps for our Binary Tree Path Sum problem:

1. Start DFS with the root of the tree.
2. If the current node is not a leaf node, do two things:
   - Subtract the value of the current node from the given number to get a new sum => `S = S - node.value`
   - Make two recursive calls for both the children of the current node with the new number calculated in the previous step.
3. At every step, see if the current node being visited is a leaf node and if its value is equal to the given number 'S'. If both these conditions are true, we have found the required root-to-leaf path, therefore return `true`.

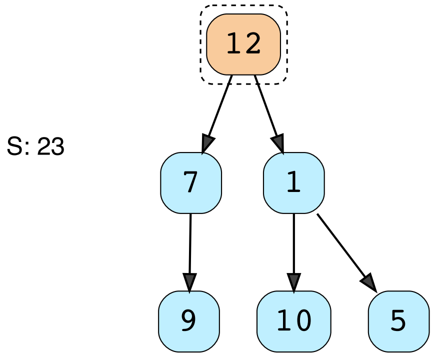
4. If the current node is a leaf but its value is not equal to the given number ‘S’, return false.

Let’s take the example-2 mentioned above to visually see our algorithm:



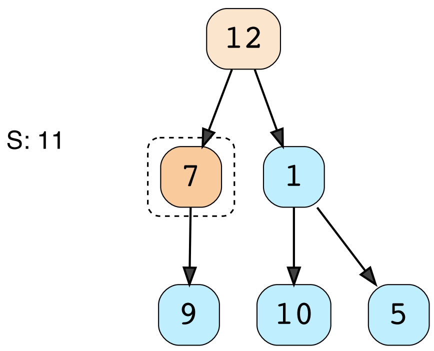
Let's start with the root node

1 of 10



Not a leaf node, make recursive calls for children.

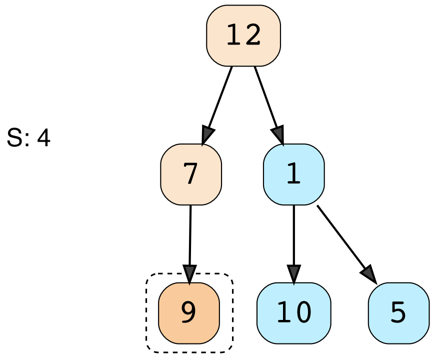
2 of 10



Not a leaf node, make recursive call for the left child.

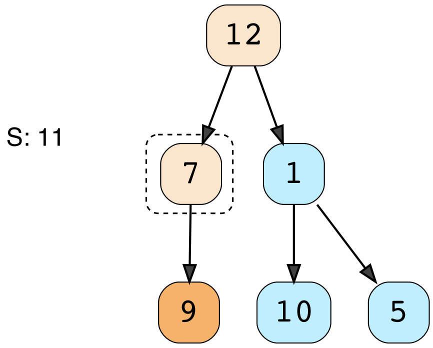
3 of 10

educative



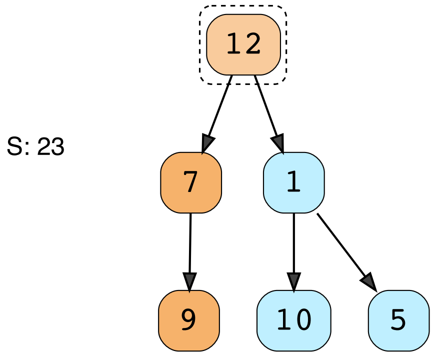
Leaf node, but S != 9, therefore return false.

4 of 10



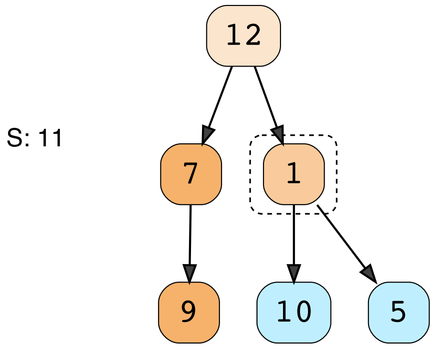
After traversing the left-child, make a recursive call for the right child. This recursive all will return false, as the right child is 'null'.

5 of 10



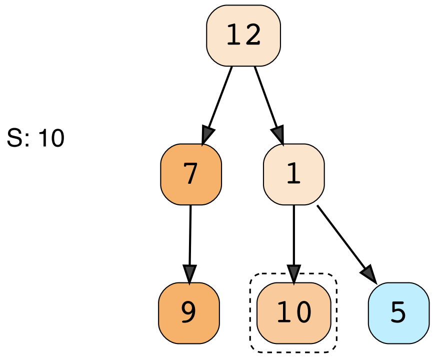
After traversing the left-child, make a recursive call for the right child, as the left-child failed in finding the path.

6 of 10



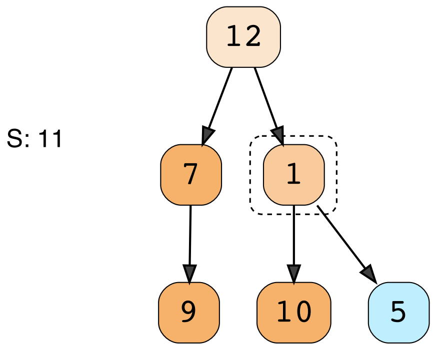
Not a leaf node, make recursive call for the left child.

7 of 10



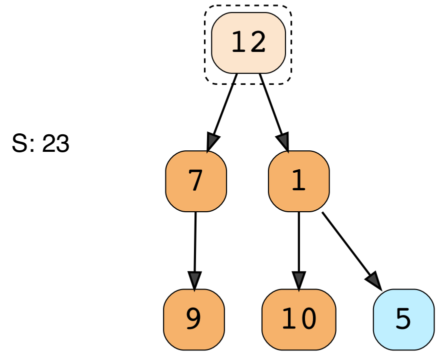
Leaf node, but S == 10, we have found a path; therefore return true.

8 of 10



As the left-child returned 'true', return 'true' without processing further.

9 of 10



As the right-child returned 'true', return 'true', we have found the path.

10 of 10

Code

Here is what our algorithm will look like:

Java | Python3 | C++ | JS

```
class TreeNode:
  def __init__(self, val, left=None, right=None):
    self.val = val
    self.left = left
    self.right = right


def has_path(root, sum):
  if root is None:
    return False

  # if the current node is a leaf and its value is equal to the sum, we've found a path
  if root.val == sum and root.left is None and root.right is None:
    return True

  # recursively call to traverse the left and right sub-tree
  # return true if any of the two recursive call return true
  return has_path(root.left, sum - root.val) or has_path(root.right, sum - root.val)


def main():

  root = TreeNode(12)
  root.left = TreeNode(7)
  root.right = TreeNode(1)
  root.left.left = TreeNode(9)
  root.right.left = TreeNode(10)
  root.right.right = TreeNode(5)
```

educative

Time complexity

The time complexity of the above algorithm is $O(N)$, where 'N' is the total number of nodes in the tree. This is due to the fact that we traverse each node once.

Space complexity

The space complexity of the above algorithm will be $O(N)$ in the worst case. This space will be used to store the recursion stack. The worst case will happen when the given tree is a linked list (i.e., every node has only one child).

Back

Introduction

Next

All Paths for a Sum (medium)

Mark as Completed

Report an Issue

Ask a Question
(https://discuss.educative.io/tag/binary-tree-path-sum-easy__pattern-tree-depth-first-search__grokking-the-coding-interview-patterns-for-coding-questions)