Problem 1597: Build Binary Expression Tree From Infix Expression

Problem Information

Difficulty: Hard
Acceptance Rate: 0.00%
Paid Only: No

Problem Description

A

binary expression tree

is a kind of binary tree used to represent arithmetic expressions. Each node of a binary expression tree has either zero or two children. Leaf nodes (nodes with 0 children) correspond to operands (numbers), and internal nodes (nodes with 2 children) correspond to the operators

'+'

(addition),

'-'

(subtraction),

'*'

(multiplication), and

'/'

(division).

'

For each internal node with operator

o

, the

infix expression

it represents is

(A o B)

, where

A

is the expression the left subtree represents and

B

is the expression the right subtree represents.

You are given a string

s

, an

infix expression

containing operands, the operators described above, and parentheses

'('

and

')'

.

Return

any valid

binary expression tree

, whose

in-order traversal

reproduces

s

after omitting the parenthesis from it.

Please note that order of operations applies in

s

.

That is, expressions in parentheses are evaluated first, and multiplication and division happen before addition and subtraction.

Operands must also appear in the

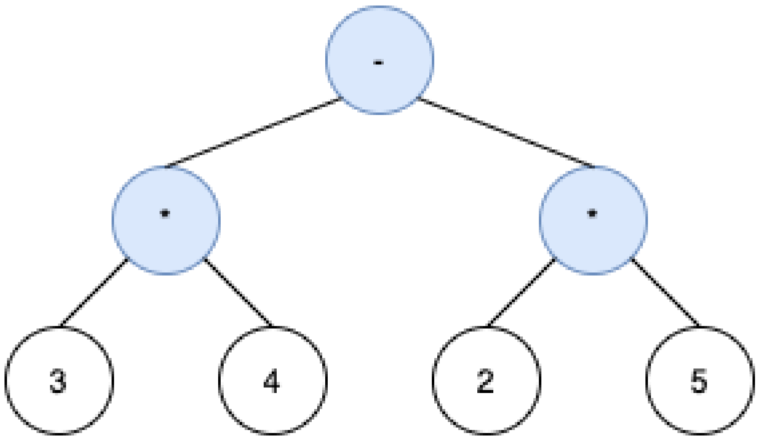
same order

in both

s

and the in-order traversal of the tree.

Example 1:



Input:

s = "3*4-2*5"

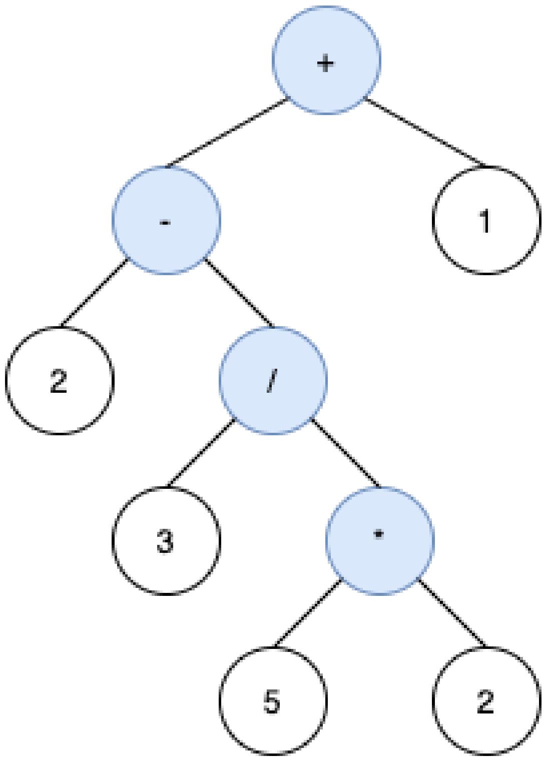
Output:

[-,*,*,3,4,2,5]

Explanation:

The tree above is the only valid tree whose inorder traversal produces s.

Example 2:



Input:

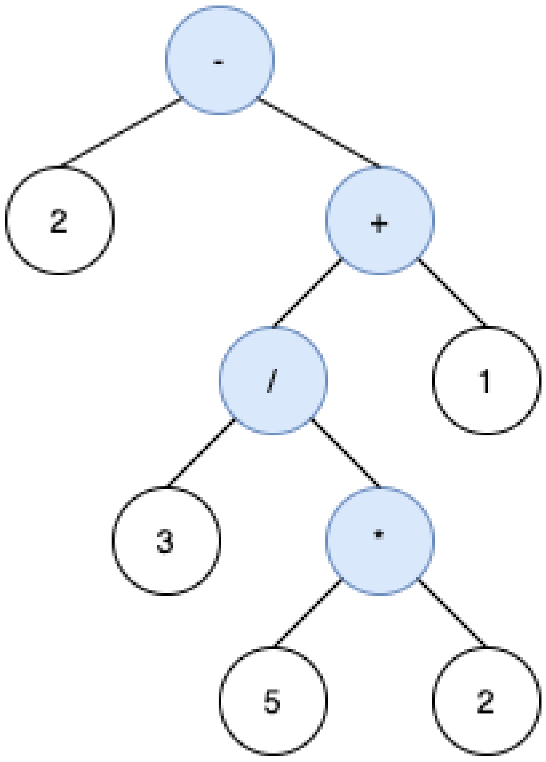
s = "2-3/(5*2)+1"

Output:

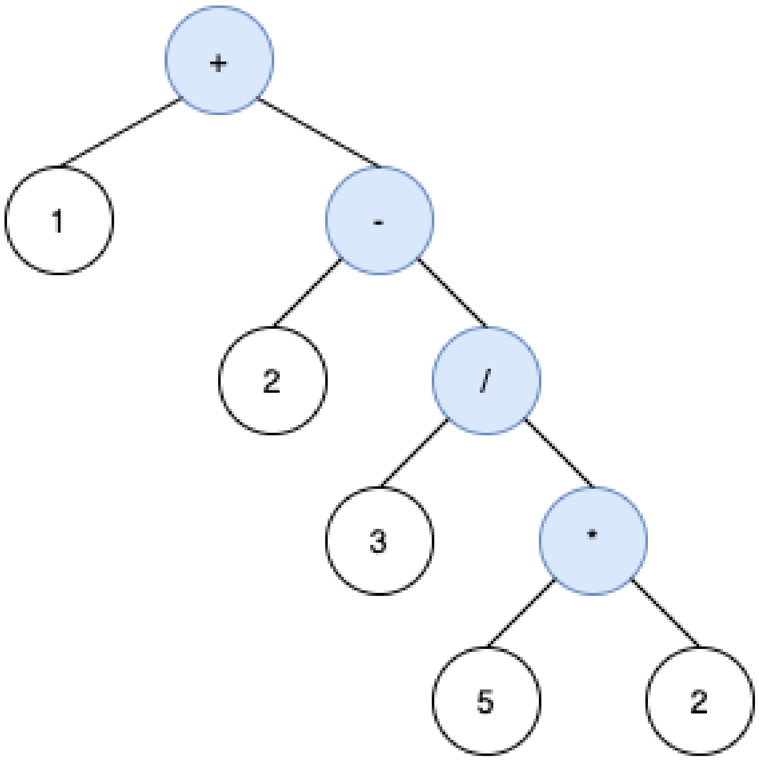
[+,-,1,2,/,null,null,null,null,3,*,null,null,5,2]

Explanation:

The inorder traversal of the tree above is 2-3/5*2+1 which is the same as s without the parenthesis. The tree also produces the correct result and its operands are in the same order as they appear in s. The tree below is also a valid binary expression tree with the same inorder traversal as s, but it not a valid answer because it does not evaluate to the same value.



The third tree below is also not valid. Although it produces the same result and is equivalent to the above trees, its inorder traversal does not produce s and its operands are not in the same order as s.



Example 3:

Input:

s = "1+2+3+4+5"

Output:

[+,+,5,+,4,null,null,+,3,null,null,1,2]

Explanation:

The tree [+,+,5,+,+,null,null,1,2,3,4] is also one of many other valid trees.

Constraints:

1 <= s.length <= 100

s

consists of digits and the characters

'('

,

')'

,

'+'

,

'-'

,

'*'

, and

'/'

.

Operands in

s

are

exactly

1 digit.

It is guaranteed that

s

is a valid expression.

Code Snippets

C++:

```
/**
* Definition for a binary tree node.
* struct Node {
* char val;
* Node *left;
* Node *right;
* Node() : val(' '), left(nullptr), right(nullptr) {}
* Node(char x) : val(x), left(nullptr), right(nullptr) {}
* Node(char x, Node *left, Node *right) : val(x), left(left), right(right) {}
* };
*/
class Solution {
public:
```

```
Node* expTree(string s) {

}
};
```

Java:

```
/**
* Definition for a binary tree node.
* class Node {
* char val;
* Node left;
* Node right;
* Node() {this.val = ' ';}
* Node(char val) { this.val = val; }
* Node(char val, Node left, Node right) {
* this.val = val;
* this.left = left;
* this.right = right;
* }
* }
*/
class Solution {
public Node expTree(String s) {

}
}
```

Python3:

```
# Definition for a binary tree node.
# class Node(object):
# def __init__(self, val=" ", left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
def expTree(self, s: str) -> 'Node':
```

Python:

```
# Definition for a binary tree node.
# class Node(object):
# def __init__(self, val=" ", left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution(object):
def expTree(self, s):
"""
:type s: str
:rtype: Node
"""
```

JavaScript:

```
/**
* Definition for a binary tree node.
* function Node(val, left, right) {
* this.val = (val===undefined ? " " : val)
* this.left = (left===undefined ? null : left)
* this.right = (right===undefined ? null : right)
* }
*/
/**
* @param {string} s
* @return {Node}
*/
var expTree = function(s) {

};
```

C#:

```
/**
* Definition for a binary tree node.
* public class Node {
* public char val;
* public Node left;
* public Node right;
* public Node(char val=' ', TreeNode left=null, TreeNode right=null) {
* this.val = val;
* this.left = left;
* this.right = right;
* }
* }
*/
public class Solution {
public Node ExpTree(string s) {

}
}
```

Solutions

C++ Solution:

```
/*
* Problem: Build Binary Expression Tree From Infix Expression
* Difficulty: Hard
* Tags: string, tree, stack
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

/**
* Definition for a binary tree node.
* struct Node {
* char val;
* Node *left;
* Node *right;
* Node() : val(' '), left(nullptr), right(nullptr) {}
* Node(char x) : val(x), left(nullptr), right(nullptr) {}
* Node(char x, Node *left, Node *right) : val(x), left(left), right(right) {}
* };
*/
class Solution {
public:
Node* expTree(string s) {

}
};
```

Java Solution:

```
/**
* Problem: Build Binary Expression Tree From Infix Expression
* Difficulty: Hard
* Tags: string, tree, stack
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

/**
* Definition for a binary tree node.
* class Node {
* char val;
* Node left;
* Node right;
* Node() {this.val = ' ';}
* Node(char val) { this.val = val; }
* Node(char val, Node left, Node right) {
* this.val = val;
* this.left = left;
* this.right = right;
* }
* }
*/
class Solution {
public Node expTree(String s) {

}
}
```

Python3 Solution:

```
"""
Problem: Build Binary Expression Tree From Infix Expression
Difficulty: Hard
Tags: string, tree, stack

Approach: String manipulation with hash map or two pointers
Time Complexity: O(n) or O(n log n)
```

```
Space Complexity: O(h) for recursion stack where h is height
"""

# Definition for a binary tree node.
# class Node(object):
# def __init__(self, val=" ", left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
def expTree(self, s: str) -> 'Node':
# TODO: Implement optimized solution
pass
```

Python Solution:

```
# Definition for a binary tree node.
# class Node(object):
# def __init__(self, val=" ", left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution(object):
def expTree(self, s):
"""
:type s: str
:rtype: Node
"""
```

JavaScript Solution:

```
/**
* Problem: Build Binary Expression Tree From Infix Expression
* Difficulty: Hard
* Tags: string, tree, stack
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/
```

```
/**
* Definition for a binary tree node.
* function Node(val, left, right) {
* this.val = (val===undefined ? " " : val)
* this.left = (left===undefined ? null : left)
* this.right = (right===undefined ? null : right)
* }
*/
/**
* @param {string} s
* @return {Node}
*/
var expTree = function(s) {

};
```

C# Solution:

```
/*
* Problem: Build Binary Expression Tree From Infix Expression
* Difficulty: Hard
* Tags: string, tree, stack
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

/**
* Definition for a binary tree node.
* public class Node {
* public char val;
* public Node left;
* public Node right;
* public Node(char val=' ', TreeNode left=null, TreeNode right=null) {
* this.val = val;
* this.left = left;
* this.right = right;
* }
* }
*/
```

```
public class Solution {
public Node ExpTree(string s) {

}
}
```