1597. Build Binary Expression Tree From Infix Expression

Explanation

The given problem asks to construct a binary tree from an infix expression. An infix expression is an algebraic expression where operators appear in-between two operands.

subtraction. This is generally enforced using parentheses, but if they aren't present it's important to follow the order of precedence. Consider an expression like: "2-3/(5*2)+1". Here, the operations must be organized in the following order: 1. Multiplication: 5*2

The task requires determining the hierarchy of operations. Multiplication and division have higher priority than addition and

2. Division: 3/(5*2) 3. Subtraction: 2-3/(5*2)

4. Addition: 2-3/(5*2)+1

In binary tree representation, operators are internal nodes with 2 children, while operands (numbers) are leaf nodes with no children.

This problem involves the use of two stack data structures, nodes and ops. The nodes stack stores the operands and the expressions created so far. The ops stack keeps track of the operators appeared in the infix expression.

The solution makes use of algorithm postfixed expression (also known as Reverse Polish Notation - RPN), which is a

mathematical notation where every operator follows all of its operands.

Solution Walkthrough

For each character in the string: If it's a number, create a new node and add it to the nodes stack. If it's an operator, then:

 While the operator at the top of the ops stack has a priority equal or greater than the current operator, pop the operator and two nodes from the stacks and build a new node. Add this new node to the nodes stack. Push the current operator onto the ops stack.

• If it's a ')', keep popping operators and building new nodes until we find a '(' in the ops stack. This will construct the binary tree for the expression inside the parentheses.

def expTree(self, s: str) -> 'Node':

return node

return node

stack = []

num = ""

node.right = helper(stack)

node.left = helper(stack)

for c in reversed("("+s+")"):

if c.isdigit():

num += c

num = ""

elif c in prec:

elif c == ")":

return helper(stack)

stack.pop()

prec = {'+': 0, '-': 0, '*': 1, '/': 1}

stack.append(Node(c))

stack.append(Node(c))

elif c == "(" and len(stack) > 1:

while stack[-2].val != ")":

stack.append(Node(int(num)))

while len(stack) > 1 and prec[stack[-2].val] >= prec[c]:

stack.append(Node(stack.pop(), stack.pop(), stack.pop()))

stack.append(Node(stack.pop(), stack.pop(), stack.pop()))

If it's a '(', just push it onto the ops stack.

empty. The nodes stack will now have one item, which is the root of the binary tree. Lastly, we just return the remaining node in the nodes stack which is the root of our binary expression tree.

Once all characters have been processed, keep popping operators from the ops stack, building new nodes, until the ops stack is

Python

def helper(stack): node = stack.pop() if node.val.isdigit():

class Solution:

```
Java
java
class Solution {
   public Node expTree(String s) {
        Stack<Node> nodes = new Stack<>();
        Stack<Character> ops = new Stack<>();
        for(char c : s.toCharArray())
            if(Character.isDigit(c)) nodes.push(new Node(String.valueOf(c)));
           else if(c == '(') ops.push(c);
            else if(c == ')') {
                while(ops.peek() != '(') {
                   nodes.push(build(ops.pop(), nodes.pop(), nodes.pop()));
                ops.pop();
            } else {
                while(!ops.isEmpty() && ops.peek() != '(' && precedence(ops.peek()) >= precedence(c)) {
                    nodes.push(build(ops.pop(), nodes.pop(), nodes.pop()));
               ops.push(c);
        while(!ops.isEmpty()) nodes.push(build(ops.pop(), nodes.pop(), nodes.pop()));
       return nodes.pop();
   public static int precedence(char c){
       if(c == '+' || c == '-')
                return 1;
            else if(c == '*' || c == '/')
                return 2;
```

cpp

class Solution {

private:

ops.push(c);

function opsStillGoes() {

if (c == '(') return r;

return new Node(c, l, r);

return nodes[0];

return true;

C#

csharp

function build() {

public class Solution {

public Node ExpTree(string s) {

else if (c != ')') {

ops.Push(c);

ops.Pop();

public int Precedence(char op) {

else return 0;

} else {

while (ops.length) nodes.push(build());

var c = ops.pop(), r = nodes.pop(), l = nodes.pop();

operator - right hierarchy and return this to be pushed back to nodes.

Stack < char > ops = new Stack < char > ();

foreach(char c in s.ToCharArray()) {

else if (c == '(') ops.Push(c);

Stack < Node > nodes = new Stack < Node > ();

public Node BuildNode(char op, Node right, Node left) {

return new Node(op.ToString(), left, right);

if (op == '+' || op == '-') return 1;

else if (op == '*' || op == '/') return 2;

Then, 4 is a digit, create a new node and push it onto nodes stack.

• Finally, 2 is a digit, create a new node and push it onto nodes stack.

* is an operator, push it onto ops stack.

else

return 0;

public static Node build(char c, Node node1, Node node2) {

Node* buildNode(char op, Node* right, Node* left) {

// Returns true if op1 is a operator and priority(op1) >= priority(op2)

return op1 == '*' || op1 == '/' || op2 == '+' || op2 == '-';

return new Node(op, left, right);

bool compare(char op1, char op2) {

return false;

if (op1 == '(' || op1 == ')')

return new Node(String.valueOf(c), node2, node1);

```
char pop(stack<char>& ops) {
            const char op = ops.top();
           ops.pop();
            return op;
       Node* pop(stack<Node*>& nodes) {
           Node* node = nodes.top();
           nodes.pop();
            return node;
   public:
       Node* expTree(string s) {
            stack<Node*> nodes; // Stores nodes.
            stack<char> ops; // Stores operators and parentheses.
            for (const char c : s)
                if (isdigit(c)) {
                    nodes.push(new Node(c));
                } else if (c == '(') {
                   ops.push(c);
                } else if (c == ')') {
                   while (ops.top() != '(')
                        nodes.push(buildNode(pop(ops), pop(nodes), pop(nodes)));
                    ops.pop(); // Remove '('.
                } else if (c == '+' || c == '-' || c == '*' || c == '/') {
                   while (!ops.empty() && compare(ops.top(), c))
                        nodes.push(buildNode(pop(ops), pop(nodes), pop(nodes)));
                    ops.push(c);
           while (!ops.empty())
                nodes.push(buildNode(pop(ops), pop(nodes), pop(nodes)));
            return nodes.top();
};
JavaScript ES6
js
function expTree(s) {
 var ops = [], nodes = [];
  for (const c of s) {
   if ('0' <= c && c <= '9') nodes.push(new Node(c));
   else if (c == '(') ops.push(c);
   else if (c == ')') while (ops[ops.length - 1] != '(') nodes.push(build());
   else {
     while (ops.length && opsStillGoes()) nodes.push(build());
```

var c = ops[ops.length - 1], c2 = ops[ops.length - 2];
if (c == '(' || c2 == '(' || (c == '+' || c == '-') && (c2 == '*' || c2 == '/')) return false;

Here, the build function checks if there are nodes available for creation of subtree by popping ops and checking the operator. If

'(', it means an expression within parenthesis has been calculated and placed in nodes, so avoid creating unnecessary Node and

instead return the precalculated Node. If any other operator, it pops two latest Nodes from nodes to create a new Node with left -

while (ops.Count > 0 && ops.Peek() != '(' && Precedence(ops.Peek()) >= Precedence(c)) nodes.Push(Buil

while (ops.Peek() != '(') nodes.Push(BuildNode(ops.Pop(), nodes.Pop(), nodes.Pop()));

In this C# solution, we iterate through s with foreach loop and break down discriminative components: numbers, operators and

and with their corresponding operator from the ops stack, new nodes are created and pushed back to the nodes stack. In the

end the final expression tree should be left in the nodes stack which is returned to complete the requirement of the problem.

Please ensure a thorough understanding of infix expressions, reversed polish notation, and tree data structures is key to solving

temporarily hold operators and operands in a sequence, and retrieve them as soon as their respective counterparts are available.

•) indicates matching open parenthesis '(', so pop two nodes and an operator from the stacks, build a new node, and push it onto nodes stack.

this problem. It should also be noted that the stack data structure plays a crucial role in constructing a binary tree from infix

expressions. Stacks are simple yet powerful data structures that follow the LIFO (last-in-first-out) principle. This allows us to

while (ops.Count > 0) nodes.Push(BuildNode(ops.Pop(), nodes.Pop(), nodes.Pop())); return nodes.Peek();

if (c >= '0' && c <= '9') nodes.Push(new Node(c.ToString()));

parentheses. Operators are kept in ops stack and node characters are kept in nodes stack. We use the function Precedence to determine 'weight' of the operators so as to resolve which operator to evaluate first. After the major calculations inside the respective stacks, now we are left with rest of the nodes in the nodes stack. The rest of the all nodes are removed from the stack

Note

Let's take a look at an example in order to better understand how stacks are used in the solution. Consider the infix expression: "(3+4)*2". Here's how the solution works: Following the algorithm, push (onto ops stack.) • 3 is a digit, create a new node and push it onto nodes stack. • + is an operator, push it onto ops stack.

At the end of the expressions, ops stack is not empty, so pop two nodes from nodes stack, and one operator from ops stack, build a new node and push it back onto nodes stack. Now, nodes stack contains the root of binary expression tree.

In conclusion, handling infix expressions to construct a binary tree involves carefully working with stack data structures, dealing with the operator precedence and handling parenthesis in the expression. Each operation in the expression becomes a node in the binary tree, with the operand or sub-expressions as their child nodes, thus forming an entire binary tree structure that represents the original expression. With practice and comprehension of these basic data structures and algorithms, such problems can be dealt with effectively.

Python Java

Solution Implementation

C++

TypeScript

Recommended Readings