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.

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

Solution Walkthrough

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.

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 '(', just push it 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.

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

Python

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

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

            node.right = helper(stack)
            node.left = helper(stack)
            return node

        stack = []
        prec = {'+': 0, '-': 0, '*': 1, '/': 1}
        num = ""
        for c in reversed("("+s+")"):
            if c.isdigit():
                num += c
                stack.append(Node(int(num)))
                num = ""
            elif c in prec:
                while len(stack) > 1 and prec[stack[-2].val] >= prec[c]:
                    stack.append(Node(stack.pop(), stack.pop(), stack.pop()))
                stack.append(Node(c))
            elif c == ")":
                stack.append(Node(c))
            elif c == "(" and len(stack) > 1:
                while stack[-2].val != ")":
                    stack.append(Node(stack.pop(), stack.pop(), stack.pop()))
                stack.pop()
        return helper(stack)
```

Java

```
lava
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;
        else
            return 0;
    }

    public static Node build(char c, Node node1, Node node2) {
        return new Node(String.valueOf(c), node2, node1);
    }
}
```

C++

```
cpp
class Solution {
    private:
        Node* buildNode(char op, Node* right, Node* left) {
            return new Node(op, left, right);
        }
        // Returns true if op1 is a operator and priority(op1) >= priority(op2)
        bool compare(char op1, char op2) {
            if (op1 == '(' || op1 == ')')
                return false;
            return op1 == '*' || op1 == '/' || op2 == '+' || op2 == '-';
        }
        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());
      ops.push(c);
    }
  }
  while (ops.length) nodes.push(build());
  return nodes[0];

  function opsStillGoes() {
    var c = ops[ops.length - 1], c2 = ops[ops.length - 2];
    if (c == ')' || c2 == '(' || (c == '+' || c == '-') && (c2 == '+' || c2 == '/')) return false;
    return true;
  }
  function build() {
    var c = ops.pop(), r = nodes.pop(), l = nodes.pop();
    if (c == '(') return r;
    return new Node(c, l, r);
  }
}
```

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

C#

```
csharp
public class Solution {
    public Node ExpTree(string s) {
        Stack < char > ops = new Stack < char > ();
        Stack < Node > nodes = new Stack < Node > ();

        foreach(char c in s.ToCharArray()) {
            if (c >= '0' && c <= '9') nodes.Push(new Node(c.ToString()));
            else if (c == '(') ops.Push(c);
            else if (c == ')') {
                while (ops.Count > 0 && ops.Peek() != '(' && Precedence(ops.Peek()) >= Precedence(c)) nodes.Push(Bui
                ops.Push(c);
            } else {
                while (ops.Peek() != '(') nodes.Push(BuildNode(ops.Pop(), nodes.Pop(), nodes.Pop()));
                ops.Pop();
            }
        }

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

        return nodes.Peek();
    }

    public Node BuildNode(char op, Node right, Node left) {
        return new Node(op.ToString(), left, right);
    }

    public int Precedence(char op) {
        if (op == '+' || op == '-') return 1;
        else if (op == '*' || op == '/') return 2;
        else return 0;
    }
}
```

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

Note

Please ensure a thorough understanding of infix expressions, reversed polish notation, and tree data structures is key to solving 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 temporarily hold operators and operands in a sequence, and retrieve them as soon as their respective counterparts are available.

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.
- Then, `4` is a digit, create a new node and push it onto `nodes` stack.
- `)` indicates matching open parenthesis '(', so pop two nodes and an operator from the stacks, build a new node, and push it onto `nodes` stack.
- `*` is an operator, push it onto `ops` stack.
- Finally, `2` is a digit, create a new node and push it onto `nodes` 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.