56、二叉树中序遍历，考点 or 实现——数据结构/栈

题目描述

根据给定的二叉树结构描述字符串，输出该二叉树按照 中序遍历 结果字符串。中序遍历顺序为：左子树，根结点，右子树。

输入描述

由大小写字母、左右大括号、逗号组成的字符串:字母代表一个节点值，左右括号内包含该节点的子节点。

左右子节点使用逗号分隔，逗号前为空则表示左子节点为空，没有逗号则表示右子节点为空。

二叉树节点数最大不超过100。

注:输入字符串格式是正确的，无需考虑格式错误的情况。

输出描述

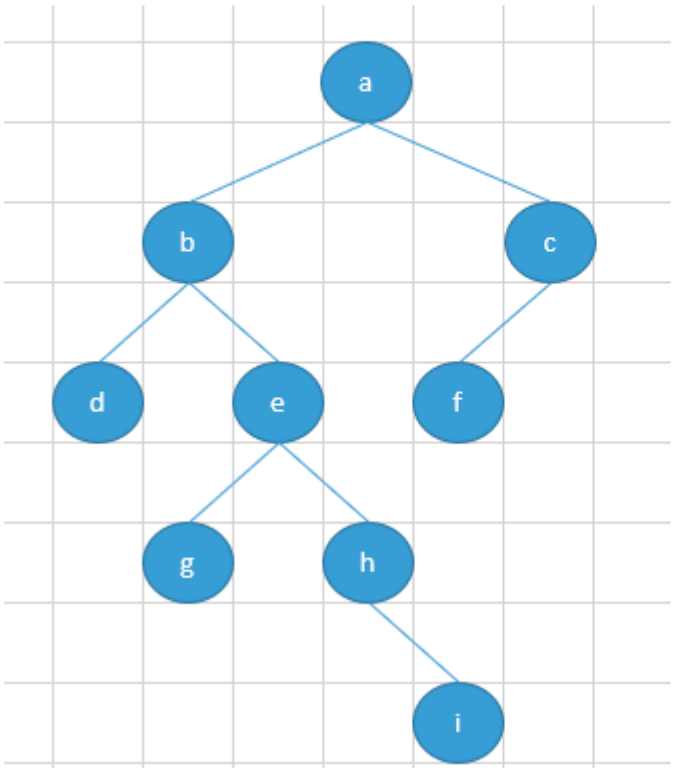
输出一个字符串为 二叉树中序遍历 各节点值的拼接结果。

用例

输入	a{b{d,e{g,h{,i}}},c{f}}
输出	dbgehiafc
说明	无

题目解析

按照题目意思，用例对应的二叉树如下：



按照中序遍历规则，左根右，遍历结果为：dbgehiafc 。

本题首先需要从给定字符串中解析出根、左子树、右子树信息，但是如果是从外到内解析，比如先提取最大的根：a，以及它的左右子树，那么将非常困难。大家可以尝试下。

我的解题思路是，利用栈结构，找到匹配的{,}，然后提取出根、以及其左右子树，实现如下：

首先定义两个栈结构stack，idxs，

然后，遍历输入的字符串的字符：默认将非 } 的字符都压入stack栈中，如果遇到 { 字符，则记录它被压入stack栈中的索引位置到idxs中。

如果遇到 } 字符

则弹出idxs栈顶的 { 索引值idx，此时我们可以根据idx索引得到：

1. 一颗子树的根，即stack[idx-1]，比如h：a{b{d,e{g,**h**{,i
2. 根下的左右叶子：即stack的idx+1~stack.size-1，比如,i ： a{b{d,e{g,h{,**i**

因此，遇到 } 字符，意味着，我们就能够解析出一颗子树。

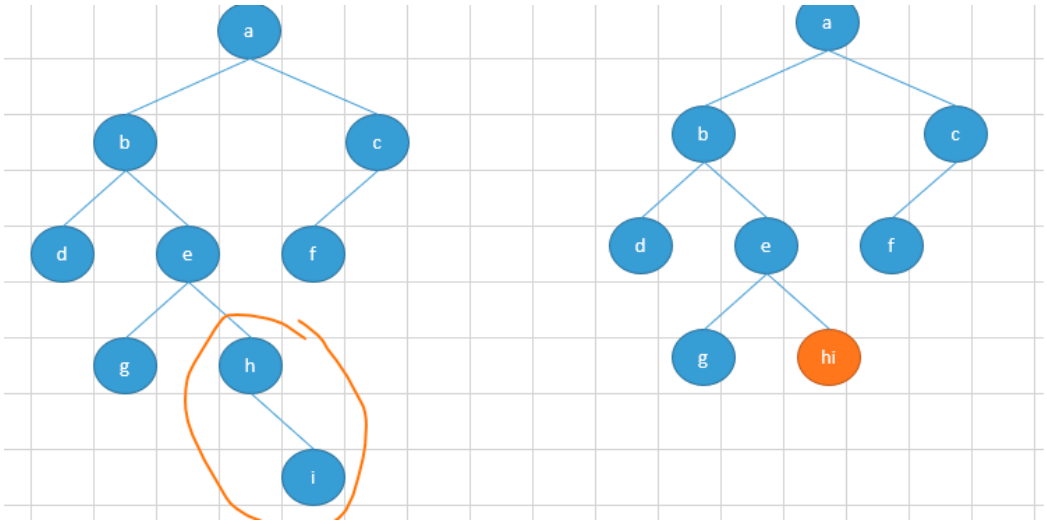
当解析出子树的根、以及左右叶子后，我们需要将这个子树按照中序遍历的特点重组为一个叶子节点，比如上面例子中，解析出子树的根为h，其左叶子为空，右叶子为i，则按照中序遍历，可以将该子树重组 "" + h + i ，得到一个值为 hi 得叶子。

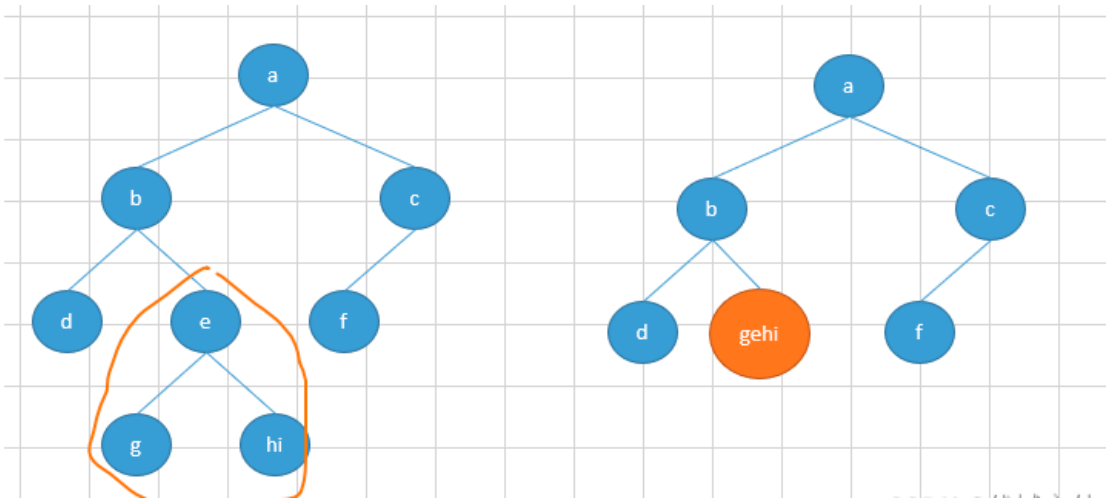
即此时stack栈内容为：a{b{d,e{g,**hi**

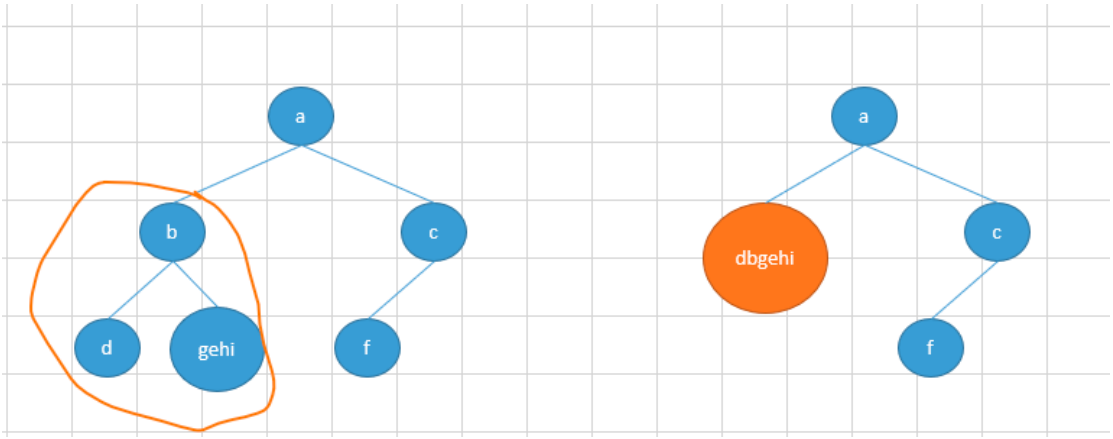
之后再遇到 }，则重复该逻辑，比如

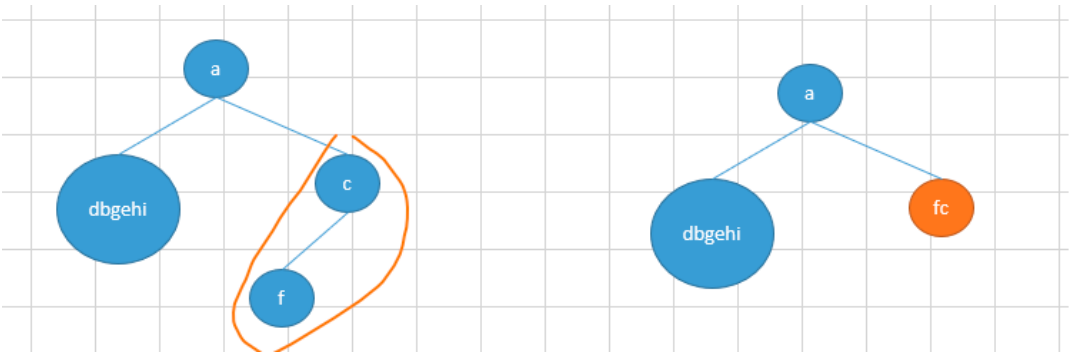
- a{b{d,**gehi**
- a{**dbgehi**
- a{dbgehi,
- a{dbgehi,c
- a{dbgehi,c{
- a{dbgehi,c{f
- a{dbgehi,**fc**
- **dbgehiafc**

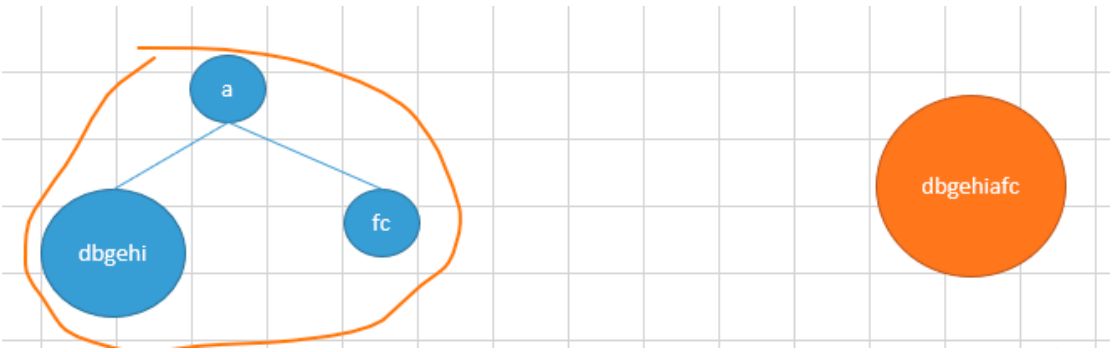
上面过程画图，如下所示




a
b
c
d
e
f
g
hi
a
b
c
d
gehi
f


a
b
c
d
gehi
f
a
dbgehi
c
f


a
dbgehi
c
f
a
dbgehi
fc


a
dbgehi
fc
dbgehiafc

JavaScript算法源码

```javascript
/* JavaScript Node ACM模式 控制台输入获取 */
const readline = require("readline");

const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout,
});

rl.on("line", (line) => {
  console.log(getResult(line));
});

function getResult(str) {
  const idxs = [];
  const stack = [];
  for (let i = 0; i < str.length; i++) {
    const c = str[i];

    if (c == "}") {
      const idx = idxs.pop(); // 左括号索引
      const root = stack[idx - 1]; // 根
      const [left, right] = stack.splice(idx).slice(1).join("").split(",");
      stack.pop();
      stack.push((left ?? "") + root + (right ?? ""));
      continue;
    }

    if (c == "{") {
      idxs.push(stack.length);
    }

    stack.push(c);
  }

  return stack[0];
}
```

Java算法源码

```java
import java.util.LinkedList;
import java.util.Scanner;

public class Main {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);

    System.out.println(getResult(sc.next()));
  }

  public static String getResult(String str) {
    LinkedList<Integer> idxs = new LinkedList<>();
    LinkedList<String> stack = new LinkedList<>();

    for (int i = 0; i < str.length(); i++) {
      char c = str.charAt(i);

      if (c == '}') {
        // 如果遇到}，则需要将它和最近的一个{匹配，而最近的{的索引就是idxs的栈顶值
        int idx = idxs.removeLast(); // 左括号在栈中的索引位置idx

        // 将{、}之间的子树内容提取出来，即从{索引+1开始提取，一直到stack栈顶
        String subTree = removeStackEles(stack, idx + 1);

        // 此时stack栈顶元素是{，我们需要去除它
        stack.removeLast();

        // 此时stack栈顶元素是子树根
        String root = stack.removeLast();

        // 将子树内容按照逗号切割，左边的是左子树，右边的是右子树
        String[] split = subTree.split(",");
        String left = split[0];
        // 如果没有逗号，则没有右子树
        String right = split.length > 1 ? split[1] : "";

        // 按照中序遍历顺序，合成:左根右
        stack.addLast(left + root + right);
        continue;
      }

      if (c == '{') {
        idxs.addLast(stack.size());
      }
```

```
45
46        stack.addLast(c + "");
47      }
48
49      return stack.get(0);
50    }
51
52    // 将栈stack，从start索引开始删除，一直删到栈顶，被删除元素组合为一个字符串返回
53    public static String removeStackEles(LinkedList<String> stack, int start) {
54      StringBuilder sb = new StringBuilder();
55      while (start < stack.size()) {
56        sb.append(stack.remove(start));
57      }
58      return sb.toString();
59    }
60  }
```

Python算法源码

```
1  # 输入获取
2  s = input()
3
4
5  # 算法入口
6  def getResult(s):
7      idxs = []
8      stack = []
9
10     for i in range(len(s)):
11         c = s[i]
12
13         if c == '}':
14             idx = idxs.pop()  # 左括号索引
15             root = stack[idx - 1]  # 根
16
17             left, right = "", ""
18             tmp = "".join(stack[(idx + 1):]).split(",")
19             if len(tmp) == 1:  # 对应c{f}这种没有右子节点的情况
20                 left = tmp[0]
21             else:
22                 left, right = tmp
23
24             stack = stack[:idx - 1]
25             stack.append(left + root + right)
26             continue
27
```

```
28          if c == '{':
29              idxs.append(len(stack))
30
31          stack.append(c)
32
33      return stack[0]
34
35
36  # 算法调用
37  print(getResult(s))
```