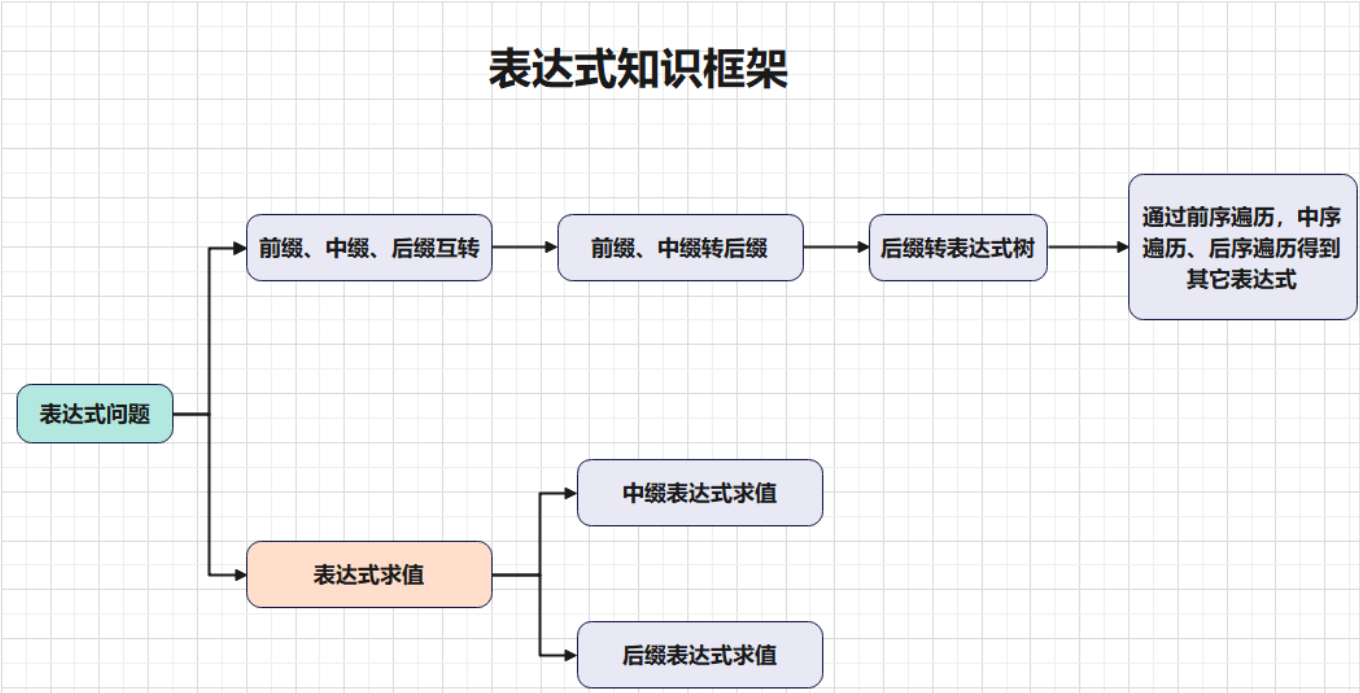


表达式求值专题

一、理论知识

22张图带你深入剖析前缀、中缀、后缀表达式以及表达式求值

####前缀/中缀/后缀----表达式之间的相互转换



二、练习题

1、中缀转后缀

```
#include <bits/stdc++.h>
using namespace std;

// 中缀表达式转后缀表达式
/*
测试用例1:
a+b*c+(d*e+f)*g

答案:
abc*+de*f+g*+

测试用例2:
(6+3*(7-4))-8/2

答案:
6 3 7 4 - * + 8 2 / -

测试用例3:
(24*(9+6/3-5)+4)
```

答案:

24 9 6 3 / + 5 - * 4 +

测试用例4:

2*3*2+5/3

*/

```
unordered_map<char, int> g{{'+', 1}, {'-', 1}, {'*', 2}, {'/', 2}}; // 操作符优化级
string t; // 结果串
stack<char> stk; // 栈
// 中缀转后缀
string infixToPostfix(string s) {
    for (int i = 0; i < s.size(); i++) {
        // ①数字
        if (isdigit(s[i])) {
            int x = 0;
            while (i < s.size() && isdigit(s[i])) {
                x = x * 10 + s[i] - '0';
                i++;
            }
            i--;
            t += to_string(x), t += ' ';
        } else if (isalpha(s[i])) // ②字符,比如a,b,c
            t += s[i], t += ' ';
        else if (s[i] == '(') // ③左括号
            stk.push(s[i]); // 左括号入栈
        else if (s[i] == ')') { // ④右括号
            while (stk.top() != '(') { // 让栈中元素(也就是+-*/和左括号)一直出栈,直到匹配的左括号出栈
                t += stk.top(), t += ' ';
                stk.pop();
            }
            stk.pop(); // 左括号也需要出栈
        } else {
            // ⑤操作符 +-*/
            while (stk.size() && g[s[i]] <= g[stk.top()]) {
                t += stk.top(), t += ' ';
                stk.pop();
            }
            stk.push(s[i]); // 将自己入栈
        }
    }
    // 当栈不为空时,全部输出
    while (stk.size()) {
        t += stk.top(), t += ' ';
        stk.pop();
    }
    return t;
}

int main() {
    string infix = "(24*(9+6/3-5)+4)";
    string postfix = infixToPostfix(infix);
}
```

```
    cout << "中缀表达式: " << infix << endl;
    cout << "后缀表达式: " << postfix << endl;
    return 0;
}
```

2、中缀转后缀(逻辑)

```
#include <bits/stdc++.h>
using namespace std;
/*
中缀逻辑表达式转后缀逻辑表达式

测试用例：
0&(0|1|0)

答案：
001|0|&
*/
unordered_map<char, int> g{{'|', 1}, {'&', 2}};
string s, t;
stack<char> stk;

int main() {
    cin >> s;
    for (int i = 0; i < s.size(); i++) {
        if (isdigit(s[i]) || isalpha(s[i]))
            t += s[i];
        else if (s[i] == '(')
            stk.push(s[i]);
        else if (s[i] == ')') {
            while (stk.top() != '(') {
                t += stk.top();
                stk.pop();
            }
            stk.pop();
        } else {
            while (stk.size() && g[s[i]] <= g[stk.top()]) {
                t += stk.top();
                stk.pop();
            }
            stk.push(s[i]);
        }
    }
    while (stk.size()) {
        t += stk.top();
        stk.pop();
    }
    cout << t << endl;
    return 0;
}
```

3、后缀表达式求值

```
#include <bits/stdc++.h>
using namespace std;

// 计算后缀表达式的值

// 判断字符是否是操作符
bool isOp(char c) {
    return c == '+' || c == '-' || c == '*' || c == '/';
}

// 后缀表达式求值
int eval(string s) {
    stack<int> stk; // 后缀表达式求值要使用到一个栈
    // 构造一个字符串流，方便处理带空格的后缀表达式
    stringstream s1(s);
    string c;
    while (s1 >> c) {
        if (isOp(c[0])) { // 如果是操作数，将其转换为整数并入栈
            int a = stk.top();
            stk.pop();
            int b = stk.top();
            stk.pop();
            if (c == "+") stk.push(b + a);
            if (c == "-") stk.push(b - a);
            if (c == "*") stk.push(b * a);
            if (c == "/") stk.push(b / a);
        } else
            stk.push(stoi(c));
    }
    return stk.top();
}

int main() {
    /*
    中缀表达式: (24*(9+6/3-5)+4)
    后缀表达式: 24 9 6 3 / + 5 - * 4 +
    值=148
    */
    string s = "24 9 6 3 / + 5 - * 4 +";
    cout << "计算结果为: " << eval(s) << endl;
    return 0;
}
```

4、AcWing 3302. 中缀表达式求值

加减乘除四则运算，利用栈，中缀表达式求值，运算符优先顺序表

直接计算版本

```
#include <bits/stdc++.h>
using namespace std;

/*
测试用例I:
(2+2)*(1+1)

答案: 8

测试用例II:
2+(3*4)-((5*9-5)/8-4)

答案: 13

*/

stack<int> num; // 数字栈
stack<char> op; // 操作符栈

// 优先级表
unordered_map<char, int> g{{'+', 1}, {'-', 1}, {'*', 2}, {'/', 2}};

/**
 * 功能: 计算两个数的和差积商
 */
void eval() {
    int a = num.top(); // 第二个操作数
    num.pop();

    int b = num.top(); // 第一个操作数
    num.pop();

    char p = op.top(); // 运算符
    op.pop();

    int r; // 结果
    // 计算结果
    if (p == '+') r = b + a;
    if (p == '-') r = b - a;
    if (p == '*') r = b * a;
    if (p == '/') r = b / a;
    // 结果入栈
    num.push(r);
}

int main() {
    // 读入表达式
    string s;
    cin >> s;
    // 遍历字符串的每一位
    for (int i = 0; i < s.size(); i++) {
        // ① 如果是数字, 则入栈
        if (s[i] >= '0' && s[i] <= '9') {
```

```

        // 读出完整的数字
        int x = 0;
        while (i < s.size() && s[i] >= '0' && s[i] <= '9') {
            x = x * 10 + s[i] - '0';
            i++;
        }
        i--; // 加多了一位, 需要减去

        num.push(x); // 数字入栈
    }
    // ② 左括号无优先级, 入栈
    else if (s[i] == '(')
        op.push(s[i]);
    // ③ 右括号时, 需计算最近一对括号里面的值
    else if (s[i] == ')') {
        // 从栈中向前找, 一直找到左括号
        while (op.top() != '(') eval(); // 将左右括号之间的计算完, 维护回栈里
        // 左括号出栈
        op.pop();
    } else { // ④ 运算符
        // 如果待入栈运算符优先级低, 则先计算
        while (op.size() && g[op.top()] >= g[s[i]]) eval();
        op.push(s[i]); // 操作符入栈
    }
}
while (op.size()) eval(); // ⑤ 剩余的进行计算

printf("%d\n", num.top()); // 输出结果
return 0;
}

```

中转后, 后求值版本

```

#include <bits/stdc++.h>
using namespace std;
// 测试用例:
// (23+124)*(98-2)

// 中缀转后缀
unordered_map<char, int> g{{'+', 1}, {'-', 1}, {'*', 2}, {'/', 2}}; // 操作符优先级
string infixToPostfix(string s) {
    stack<char> stk; // 栈

    string t; // 结果串

    for (int i = 0; i < s.size(); i++) {
        // ①数字
        if (isdigit(s[i])) {
            int x = 0;
            while (i < s.size() && isdigit(s[i])) {
                x = x * 10 + s[i] - '0';
            }
        }
    }
}

```

```

        i++;
    }
    i--;
    t += to_string(x) + " ";
} else if (isalpha(s[i])) // ②字符,比如a,b,c
    t.push_back(s[i]), t.push_back(' ');
else if (s[i] == '(') // ③左括号
    stk.push(s[i]); // 左括号入栈
else if (s[i] == ')') { // ④右括号
    while (stk.top() != '(') { // 让栈中元素(也就是+-*/和左括号)一直出栈,直到匹配的左括号出栈
        t.push_back(stk.top()), t.push_back(' ');
        stk.pop();
    }
    stk.pop(); // 左括号也需要出栈
} else {
    // ⑤操作符 +-*/
    while (stk.size() && g[s[i]] <= g[stk.top()]) {
        t.push_back(stk.top()), t.push_back(' ');
        stk.pop();
    }
    stk.push(s[i]); // 将自己入栈
}
}
// 当栈不为空时,全部输出
while (stk.size()) {
    t.push_back(stk.top()), t.push_back(' ');
    stk.pop();
}
return t;
}

// 判断字符是否是操作符
bool isOp(char c) {
    return c == '+' || c == '-' || c == '*' || c == '/';
}

// 后缀表达式求值
int eval(string s) {
    stack<int> stk; // 后缀表达式求值要使用到一个栈
    // 构造一个字符串流,方便处理带空格的后缀表达式
    stringstream s1(s);
    string c;
    while (s1 >> c) {
        if (isOp(c[0])) { // 如果是操作数,将其转换为整数并入栈
            int a = stk.top();
            stk.pop();
            int b = stk.top();
            stk.pop();
            if (c == "+") stk.push(b + a);
            if (c == "-") stk.push(b - a);
            if (c == "*") stk.push(b * a);
            if (c == "/") stk.push(b / a);
        } else

```

```

        stk.push(stoi(c));
    }
    return stk.top();
}

int main() {
    string s;
    cin >> s;
    // 中缀转后缀
    s = infixToPostfix(s);

    // 后缀表达式求值
    cout << eval(s) << endl;
    return 0;
}

```

5、表达式求值(逻辑)

```

#include <bits/stdc++.h>
using namespace std;

/*
0&(1|0)|(1|1|1&0)
答案: 1

(0|1&0|1|1|(1|1))&(0&1&(1|0)|0|1|0)&0
答案: 0
*/
unordered_map<char, int> g{{'|', 1}, {'&', 2}};
stack<int> num;
stack<char> op;

void eval() {
    int a = num.top();
    num.pop();

    int b = num.top();
    num.pop();

    char p = op.top();
    op.pop();

    int r;
    if (p == '|') r = b | a;
    if (p == '&') r = b & a;
    num.push(r);
}

int main() {
    string s;
    cin >> s;
}

```

```

    for (int i = 0; i < s.size(); i++) {
        if (isdigit(s[i]))
            num.push(s[i] - '0');
        else if (s[i] == '(')
            op.push(s[i]);
        else if (s[i] == ')') {
            while (op.top() != '(') eval();
            op.pop();
        } else {
            while (op.size() && g[op.top()] >= g[s[i]]) eval();
            op.push(s[i]);
        }
    }
    while (op.size()) eval();

    printf("%d\n", num.top());
    return 0;
}

```

6、AcWing 1623. 二叉表达式树转中缀表达式

- 给定一个**二叉表达式树**，请你输出相应的**中缀表达式**，并利用括号反映运算符的优先级。**dfs中序遍历**可以求中缀表达式的值，仔细观察用例，可以知道根不是任何人的儿子，需要记录入度找到入度为0的点，就是根

```
#include <bits/stdc++.h>
```

```
using namespace std;
```

```
const int N = 30;
```

```
/*
思路
```

- 1、数据处理,通过记录入度，找到入度为0的点，找到根节点
- 2、二元运算必须有符号右边那个式子，即根据数据构建完树后的右孩子
故在递归遍历树的时候，只要判断该节点是否有孩子，
然后返回的字符串数：递归左子树的string值 + 该节点的string值 + 递归右子树的string值
- 3、如果该节点不是根节点,则在该返回值的左右两边加上括号
- 4、最后输出返回值

黄海总结：

- 1、乍一看此题，感觉无从下手，因为不知道每个点都是什么节点号！后来慢慢从测试用例1中发现，data 都是什么 *,a,b,-,c之类，不是节点号，应该是具体的操作符，也就是值。
那节点号在哪里呢？再看一下后面的left_child,right_child,发现是

```

8 7
-1 -1
4 1

```

2 5

这个来理解一下，应该是指8号节点，7号节点，-1代表的是NULL节点，也就是到头了，是叶子。
再来仔细看一下，发现出现的数字是：1 2 4 5 6 7 8, 单单缺少了一个3！
为啥缺少3呢？

因为3不是任何人的left_child, right_child, 它是根！！！！，这样的话，整个图就完整了！

```

*/
int n;
string w[N];
int l[N], r[N], in[N];
int isleaf[N];
/*
8
* 8 7
a -1 -1
* 4 1
+ 2 5
b -1 -1
d -1 -1
- -1 6
c -1 -1

(a+b)*(c*(-d))
*/
string dfs(int u) {
    // 表达式树特点：叶子节点都是真实数字，非叶子节点都是操作符
    string left, right;

    // if (isleaf[u]) return w[u]; // 递归终点，如果是叶子，本句话可以省略掉
    // 这是因为isleaf[u]时, l[u]==-1 && r[u]==-1 ,则下面的两个分支都不会走，直接到了
    // return那句，left, right都是空，还是返回w[u]，一样的

    if (l[u] != -1) {
        left = dfs(l[u]); // 递归左儿子，
        if (!isleaf[l[u]]) left = "(" + left + " "; // 如果左儿子不是叶子，需要把表
        // 达式加上括号
    }
    if (r[u] != -1) {
        right = dfs(r[u]); // 右儿子不为空
        if (!isleaf[r[u]]) right = "(" + right + " "; // 如果右儿子不是叶子，需要把
        // 表达式加上括号
    }
    // 返回中序遍历结果
    return left + w[u] + right;
}

int main() {
    cin >> n;

    for (int i = 1; i <= n; i++) {
        cin >> w[i] >> l[i] >> r[i]; // data left_child
        // right_child
        if (l[i] != -1) in[l[i]]++; // 记录入度
    }
}

```

```

        if (r[i] != -1) in[r[i]]++; // 记录入度
        if (l[i] == -1 && r[i] == -1) isleaf[i] = 1; // 叶子
    }
    // 找根
    int root;
    for (root = 1; root <= n; root++)
        if (!in[root]) break; // 入度为零的是根

    // 通过中序遍历，构建中缀表达式
    cout << dfs(root) << endl;
    return 0;
}

```

7、AcWing 4274. 二叉表达式树转后缀表达式

- 给定一个**二叉表达式树**，请你输出相应的**后缀表达式**，要求使用括号反映运算符的优先级。
dfs后序遍历可以求后缀表达式的值，仔细观察用例，可以知道根不是任何人的儿子，需要记录入度找到入度为0的点，就是根

```

#include <bits/stdc++.h>
using namespace std;
const int N = 30;

```

```

int n;
string w[N];
int l[N], r[N], in[N];
int isleaf[N];

```

/*

题目描述：

给出中缀表达式的语法树 请你将后缀表达式输出 括号体现优先级 操作符不可用括号包含。

思路：

后缀表达式就是把运算符后置的表达式，可以抽象成一棵二叉树，其中的节点有两种情况：

(1) 有左右儿子

(2) 只有右儿子

只有右儿子的情况就是负数啦~ 测试用例： - -1 6

然后就是找根，这个很简单，统计所有节点的入度，唯一的一个入度是 0 的节点就是根。

再写个简单的递归就可以了。

```

8
* 8 7
a -1 -1
* 4 1
+ 2 5
b -1 -1

```

```
d -1 -1
- -1 6
c -1 -1
```

输出样例1:

```
((a)(b)+)((c)(-(d))*)*
*/
```

```
string dfs(int u) {
    if (isleaf[u]) return "(" + w[u] + ")";
    if (l[u] == -1 && r[u] != -1) return "(" + w[u] + dfs(r[u]) + ")"; // 负号
    return "(" + dfs(l[u]) + dfs(r[u]) + w[u] + ")"; // 后缀表达式
}

int main() {
    cin >> n;
    for (int i = 1; i <= n; i++) {
        cin >> w[i] >> l[i] >> r[i];
        if (l[i] != -1) in[l[i]]++;
        if (r[i] != -1) in[r[i]]++;
        if (l[i] == -1 && r[i] == -1) isleaf[i] = 1; // 叶子
    }
    // 找根
    int root;
    for (root = 1; root <= n; root++)
        if (!in[root]) break; // 入度为零的是根
    cout << dfs(root) << endl;

    return 0;
}
```

####8、 HDU1123 Complicated Expressions

题意： 给定一个 **中缀表达式**（只有小写字母，加减乘除，左右括号），**会带有很多冗余的括号**，要求去除这些括号时满足以下的要求：(1)、完全没有意义的括号去除；(2)、加法和乘法的结合律 $\rightarrow a+(b+c)$ 等价于 $a+b+c$ ， $a*(bc)$ 等价于 $a*b*c$ ；故这种括号要去除(3)、有明显的**不影响运算的括号去除**，如 $a+(b*c) \rightarrow a+bc$ **思路：** 先不考虑上述要求，(1)对于给定的中缀表达式，先转化成后缀表达式（中缀 $\rightarrow$ 后缀）；(2)然后利用后缀表达式建立一颗二叉树（**后缀二叉树建立表达式树**）；(3)利用表达式树得到中缀表达式。做完以上要求后，把要求的括号去除与否的规则考虑进去就可以了

```
#include <bits/stdc++.h>

using namespace std;
// 中缀转后缀
unordered_map<char, int> g{{'+', 1}, {'-', 1}, {'*', 2}, {'/', 2}}; // 操作符优先级
string infixToPostfix(string s) {
    string t; // 结果串
    stack<char> stk; // 栈
    for (int i = 0; i < s.size(); i++) {
        // ①数字
```

```

        if (isdigit(s[i])) {
            int x = 0;
            while (i < s.size() && isdigit(s[i])) {
                x = x * 10 + s[i] - '0';
                i++;
            }
            i--;
            t += to_string(x), t += ' ';
        } else if (isalpha(s[i])) // ②字符,比如a,b,c
            t += s[i], t += ' ';
        else if (s[i] == '(') // ③左括号
            stk.push(s[i]); // 左括号入栈
        else if (s[i] == ')') { // ④右括号
            while (stk.top() != '(') { // 让栈中元素(也就是+*/和左括号)一直出栈,直到匹配的左括号出栈
                t += stk.top(), t += ' ';
                stk.pop();
            }
            stk.pop(); // 左括号也需要出栈
        } else {
            // ⑤操作符 +-*/
            while (stk.size() && g[s[i]] <= g[stk.top()]) {
                t += stk.top(), t += ' ';
                stk.pop();
            }
            stk.push(s[i]); // 将自己入栈
        }
    }
    // 当栈不为空时,全部输出
    while (stk.size()) {
        t += stk.top(), t += ' ';
        stk.pop();
    }
    return t;
}

// 后缀转表达式树
const int N = 500, M = N << 1;
#define ls e[h[u]] // 左儿子
#define rs e[ne[h[u]]] // 右儿子,由于表达式树是二叉树,有右儿子时一定有左儿子
// 链式前向星
int e[M], h[N], idx, ne[M];
void add(int a, int b) {
    e[idx] = b, ne[idx] = h[a], h[a] = idx++;
}
char a[N];
int a1 = 25; // a[]:节点号对应的字符,0~25为数字a~z在表达式树中节点的编号,26及以上为每一个操作符的编号,注意:如果前后出现多次同一运算符,它们的节点号是不一样的

// 构造表达式树
int build(string postfix) {
    stack<int> stk; // 节点号栈
    // 链式前向星初始化
    memset(h, -1, sizeof h);

```

```

    idx = 0, al = 25;

    // 后缀表达式->表达式树, 可以转化: 前缀表达式, 中缀表达式, 后缀表达式
    for (int i = 0; i < postfix.size(); i++) { // 枚举后缀表达式每一个字符
        char c = postfix[i];
        if (c == ' ') continue;
        if (isalpha(c))
            stk.push(c - 'a'); // 不是操作符, 是数字, 按字符-'a'的数字号入栈
        else { // 如果是运算符
            int x = stk.top(); // 取出两个节点
            stk.pop();
            int y = stk.top();
            stk.pop();

            a[++al] = c; // 操作符, 申请一个节点号, 并且, 记录下来节点号与
原字符的关系
            add(al, x), add(al, y); // 由操作符向左右两个儿子连出边
            stk.push(al); // 操作符节点入栈
        }
    }
    return stk.top(); // 栈顶部的字符就是根
}

// 输出节点u
char out(int u) {
    if (u >= 26) return a[u]; // 节点号大于等于26的是操作符+-*/
    return u + 'a'; // a~z, 记录的是0~25, 输出需要加上偏移量'a'
}

// 判断操作符
bool isOp(char c) {
    return c == '+' || c == '-' || c == '*' || c == '/';
}

// 获取符合题意的中缀表达式
string inorder(int u) {
    string ans = "";
    if (h[u] == -1) {
        ans += out(u);
        return ans;
    }
    string lstr = inorder(ls);
    string rstr = inorder(rs);

    // 下面的代码是本题的题意要求
    if (isOp(a[ls]) && g[a[u]] > g[a[ls]]) { // 如果左儿子是操作符, 并且当前运算符优
优先级大于左儿子运算优先级, 需要左儿子加上括号
        ans += '(';
        ans += lstr;
        ans += ')';
    } else
        ans += lstr; // 否则, 左儿子不加括号

    ans += out(u); // 左儿子完事, 把自己加上

```

```

    if (isOp(a[rs]) && g[a[rs]] <= g[a[u]]) { // 如果右儿子是操作符，并且，右儿子运算
        符优先级小于等于当前运算符优先级
        if ((a[u] == '+' && a[rs] == '-') // 当前是+, 右儿子是-
            || (a[u] == '*' && a[rs] == '/') // 当前是*, 右儿子是/
            || (a[u] == '+' && a[rs] == '+') // 当前是+, 右儿子是+
            || (a[u] == '*' && a[rs] == '*')) // 当前是*, 右儿子是*
            ans += rstr; // 都不需要加括号
        else
            ans += '(' + rstr + ')'; // 否则需要加括号
    } else
        ans += rstr; // 否则右儿子不加括号

    return ans;
}

int main() {
#ifdef ONLINE_JUDGE
    freopen("HDU1123.in", "r", stdin);
    freopen("HDU1123.out", "w", stdout);
#endif
    int n;
    cin >> n;
    while (n--) {
        string infix;
        cin >> infix;
        // 中缀转后缀
        string postfix = infixToPostfix(infix);
        // 根据后缀表达式创建表达式树
        int root = build(postfix);
        cout << inorder(root) << endl;
    }
    return 0;
}

```

三、TODO 以后再做

【洛谷】P1449 后缀表达式

【洛谷】P7073 [CSP – J2020] 表达式

【洛谷】P8815 [CSP – J2022] 逻辑表达式

【洛谷】P1981 [NOIP2013 普及组] 表达式求值

【洛谷】P1310 [NOIP2011 普及组] 表达式的值

AcWing 2769. **表达式【中等】**

AcWing 151. **表达式计算4【中等】**

AcWing 448. **表达式的值**【中等】

AcWing 454. **表达式求值**【中等】

AcWing 486. **等价表达式**【中等】

AcWing 1247. **后缀表达式**【中等】

AcWing 1330. **表达式求值**【中等】

AcWing 4730. **逻辑表达式**【中等】