

数据结构：表达式树

姓名：鲁国锐

学号：17020021031

专业：电子信息科学与技术

2019 年 4 月 11 日

Contents

| | | |
|-----|--------------------------|----|
| 1 | 问题分析 | 2 |
| 1.1 | 题目描述 | 2 |
| 1.2 | 问题分析 | 2 |
| 2 | 解决方案 | 2 |
| 2.1 | 设计输入输出流程 | 2 |
| 2.2 | 将中缀表达式转换为后缀表达式 | 3 |
| 2.3 | 构建表达式树 | 3 |
| 3 | 算法设计 | 4 |
| 4 | 编程实现 ¹ | 6 |
| 5 | 结果分析 | 12 |
| 5.1 | 结果展示 | 12 |
| 5.2 | 分析 | 14 |
| 6 | 总结体会 | 14 |
| 6.1 | 被忽略的空字符 | 14 |
| 6.2 | 循环析构 | 14 |

1 问题分析

1.1 题目描述

表达式树的树叶是操作数 (*operand*)，而其他的节点为操作符 (*operator*)。比如下面的示例：

代表：中缀表达式： $(a + b) * (c * (d + e))$ 以及相应的后缀表达式： $ab + cde + **$ 。

作业要求输入一个中缀表达式，构建其相应的表达式树，并输出验证。

- 样例输入： $(a+b)*(c*(d+e))$
- 样例输出：（格式可修改为更加美观的图形化输出）

×

+ ×

a b c +

d e

请测试至少三组不同输入。

提示：先将中缀表达式转换为后缀表达式，然后结合栈 (*stack*) 构造表达式树。

1.2 问题分析

根据题目，我们需要解决的问题有：

1. 设计输入输出流程并对输入表达式的合法性进行检查（见2.1节）；
2. 实现函数 *to_RPN* 将中缀表达式转换为后缀表达式（见2.2节）；
3. 用后缀表达式构建表达式树（见2.3节）。

2 解决方案

2.1 设计输入输出流程

因为考虑到操作数可能是多位数，所以方便起见我们用 *string* 来表示操作数和运算符，并将它们存在一个 *vector* 中。而为了能够将每一个 *string* 分开，我们规定所有的操作数和运算符之间必须用空格分隔。

对于输出，根据表达式树的特性：对其进行后序遍历并输出得到的是一个后缀表达式，我们可以先将得到的后缀表达式输出一遍，再对构造好的表达式树后序遍历输出一次，稍微调整一下缩进，使输出的两个表达式对齐，可以很方便地检查二者是否一致。至于以树状形式输出，由于每个操作数不等长，所以直接以层序输出再调整缩进的话会非常复杂。后来受到网上一篇博文¹的启发，我们只需要将树旋转 90°，就可以完美地解决这个问题。具体算法很简单，在此不做过多叙述，但要稍微强调一下，由于我输入的操作数长度可能不同，故对该博文的算法做了一些改动：把原先的输出 *depth* 个空格换成了输出 $depth \times MAX_LEN$ 个空格，这里的 *MAX_LEN* 表示树中最长元素的长度。其余的细节见图2。

为了使程序能够正常运行并退出，我们还要对输入进行检查以确保它的合法性，防止在程序执行过程中出现数组越界等情况。不过时间有限，这里我主要考虑了以下几种情况：

1. 输入中有非法字符；
2. 两字符之间没有加空格；

¹博文网址（注意该博文中贴出的代码是有问题的）：<https://blog.csdn.net/qiaoge134/article/details/12857851>

3. 括号未匹配。

由于我们再判断两个运算符的优先级时用到了数组，所以以上三种情况可能会导致数组的下标越界。至于解决方案因为涉及到将中缀表达式转换为后缀表达式的方法，我们放在下一节中介绍。

2.2 将中缀表达式转换为后缀表达式

关于怎么将中缀表达式转换为后缀表达式的算法，课本 [1] 上有比较详细的介绍，但在这里我们仍需要对其进行修改和补充以适应我们自己的要求。

在开始实现 `to_RPN` 函数前我们首先要实现一个比较运算符之间优先级的函数 `order`。这里我们借鉴课本 [1] 上的做法，用一个枚举类型的变量将各个运算符的英文表示换成数字。然后用一个二维数组来表示两个运算符之间的优先级关系。为了方便把原始的运算符跟其英文表示对应起来，这里我们还用一个 `map` 把它们一一对应起来。这样我们在比较两个运算符时，只需要把它们作为 `key` 值传入 `map` 得到其英文表示对应的数字，再将数字作为下标传给数组来搜寻二者的优先级关系即可。

到这里我们就可以阐述一下2.1节中提到的第一种和第二种情况的解决方法了。如果输入中有非法字符，那么我们在 `map` 中是找不到对应的 `key` 值的，所以我们在对原始运算符进行转换之前要先在 `map` 中查找一遍，若没有这个 `key` 值，需要抛出一个异常，终止函数的进行并让用户重新输入。而对于第二种情况，操作数和运算符因为中间缺少空格而连在了一起，我们在对这个了“连体儿”进行判断时，认为它是操作符，显然作为运算符它是不合法的，这样就把第二种情况化归成了第一种情况。

接下来就是中缀表达式转后缀表达式的具体算法了。这里我们需要借用一个运算符栈 `optr`。首先我们从前往后遍历存放中缀表达式的 `vector`，若：

1. 当前扫到的是操作数，直接将它接到要返回的结果 `RPN` 后面；
2. 当前扫到的是运算符，则开始判断 `optr` 栈顶运算符和当前运算符之间的优先级关系，若：
 - (a) 栈顶运算符优先级低于当前运算符，将当前运算符压入栈中，令下标 `i` 增一，开始访问下一元素；
 - (b) 栈顶运算符优先级与当前运算符一样，弹出栈顶元素，令下标 `i` 增一，开始访问下一元素；
 - (c) 栈顶运算符优先级高于当前运算符，弹出栈顶元素并把它接入 `RPN`，此时不能令下标自增，而是要继续比较当前运算符与栈中下一运算符的优先级关系。
 - (d) 不是以上任何一种情况，说明两个本不可能相邻的运算符碰到了一起，这就是2.1节中所说的第三种情况：括号未匹配。注意这里右括号以及空字符的优先级是最低的，而左括号的优先级是最高的，所以其它任何栈顶运算符遇到了右括号或空字符都会被弹出，任何当前运算符遇到左括号都会被压入。如果括号正常匹配的话，左括号或右括号是不会遇到空字符的，而一旦遇到，则说明括号一定没有匹配上。因此我们在数组中给这些情况赋一个其它的值，以便检查括号的匹配情况。

注意这里我们以空字符表示表达式的开始与结束，所以我们在一开始先将一个空字符入栈，同时给输入的最后接上一个空字符。这样我们重复以上操作直至最后两个空字符相遇，栈内的所有元素被弹出，同时输入也已遍历到了最后，整个转换过程就此结束，返回的 `RPN` 即为后缀表达式。

2.3 构建表达式树

仔细分析一下我们可以发现，构建表达式树的过程其实和后缀表达式求值的过程是一样的。表达式树中的每一个含运算符的节点都可视为是它的两个孩子用它进行运算后的结果。所以我们直接按照后缀表达式求值的过程来构建表达式树即可。

我们从前往后遍历得到的后缀表达式，如果当前元素是操作数，用其构建一个节点压入辅助栈中；如果当前元素是运算符，则弹出一个或两个元素（视该运算符是单目还是双目而定），分别作为右孩子和左孩子接在

以当前元素构建的节点上，再将新构建的节点压回栈中。注意先弹出的接为右孩子，后弹出的接为左孩子。如此反复直至遍历完整个后缀表达式，此时栈中应当还剩下一个元素，即为表达式树的根节点。我们对其进行后序遍历并输出，得到的结果应与后缀表达式一致。

3 算法设计

见图1、图2²。

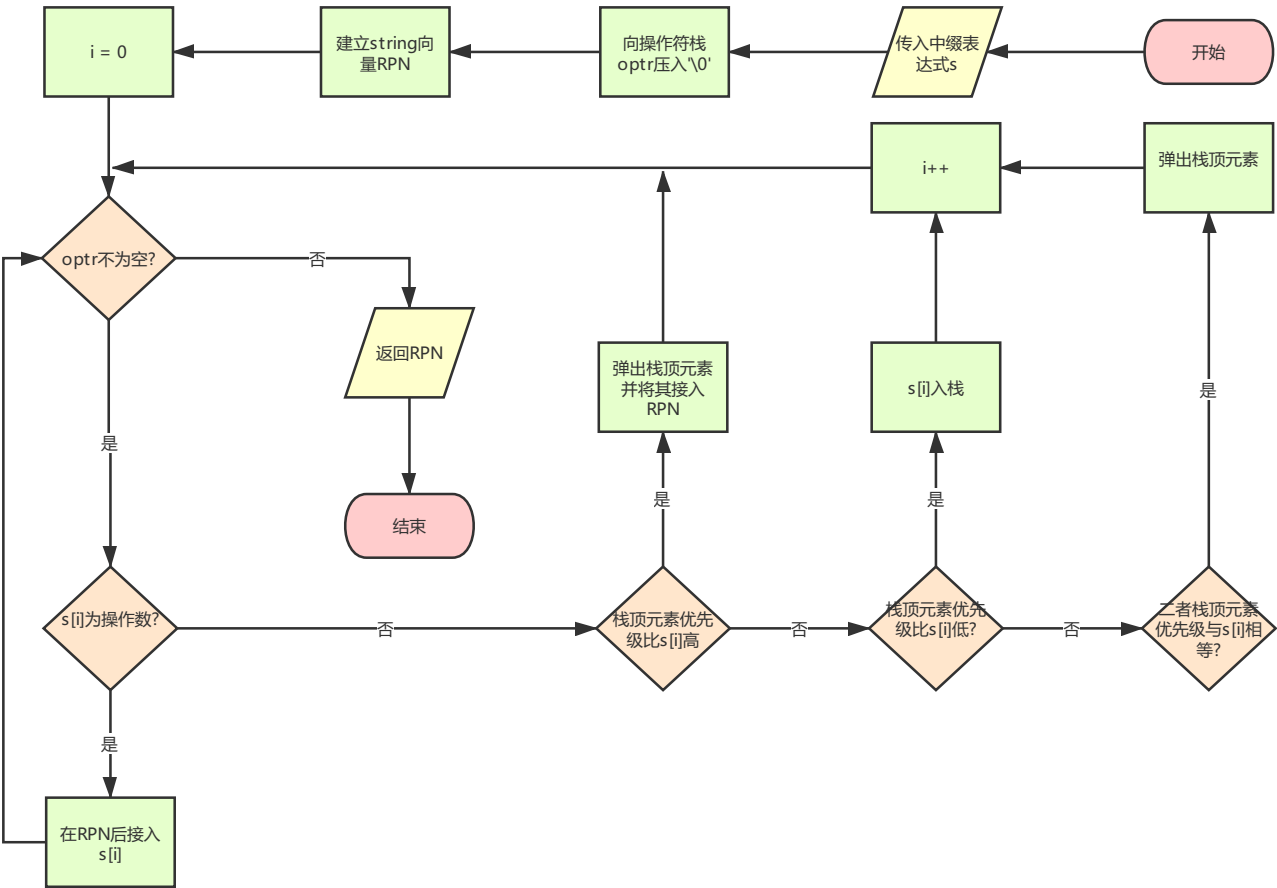


图 1: `to_RPN` 函数流程图

²为使流程图相对简洁，未在其中画出捕获异常的过程。

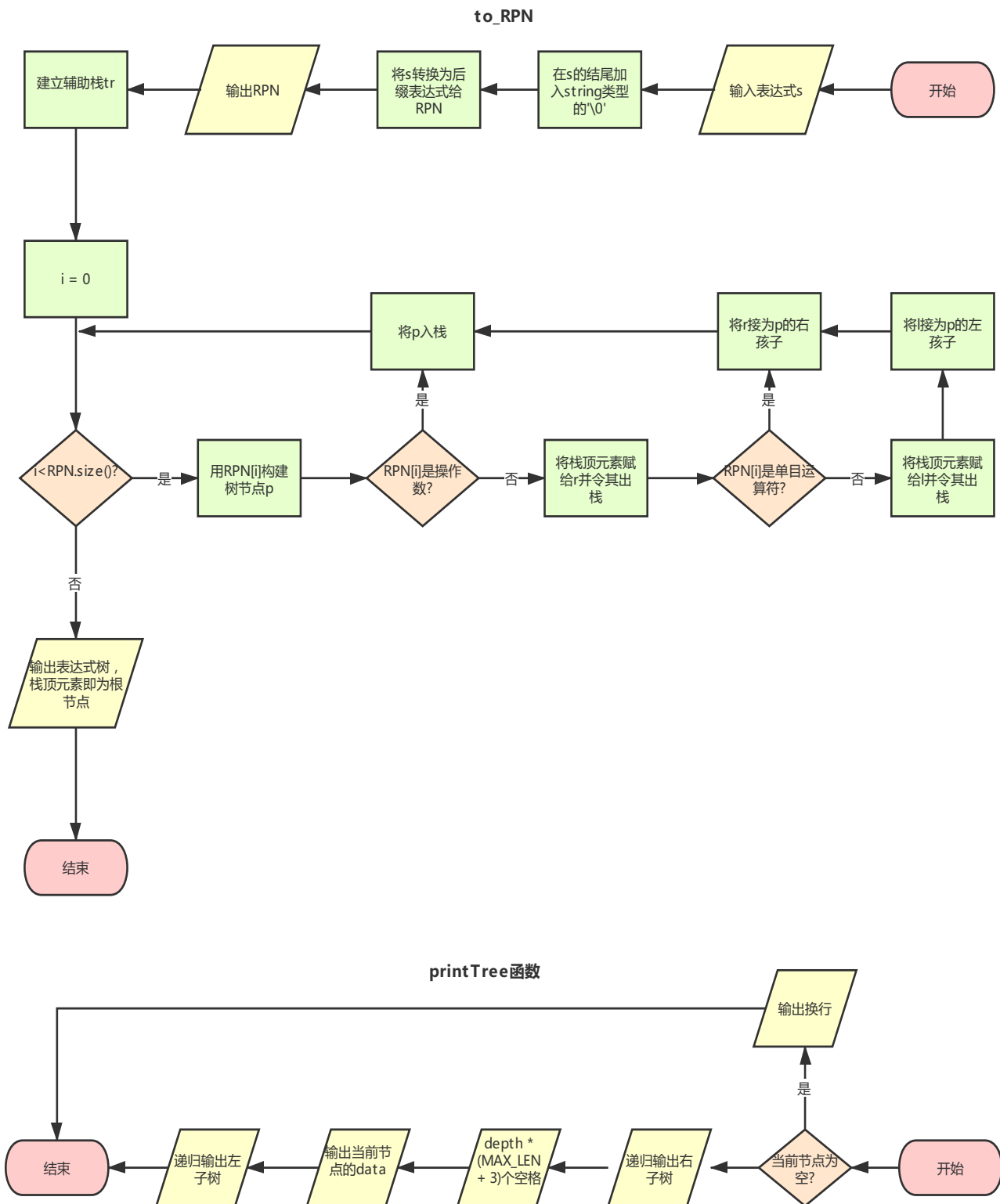


图 2: *main* 函数流程图

4 编程实现¹

Listing 1: expression_tree 代码

```
1 #include <iostream>
2 #include <string>
3 #include <stack>
4 #include <vector>
5 #include <typeinfo>
6 #include <map>
7 #include <stdio.h>
8 #include <algorithm>
9
10
11 using namespace std;
12 #define N_OPTR 9
13 typedef enum { ADD, SUB, MUL, DIV, POW, FAC, L_P, R_P, EOE } Operator;
14 //      +   -   *   /   ^   !   (   )   \0
15
16 const char pri[N_OPTR][N_OPTR] =
17 {
18     '>', '>', '<', '<', '<', '<', '<', '>', '>',
19     '>', '>', '<', '<', '<', '<', '<', '>', '>',
20     '>', '>', '>', '>', '<', '<', '<', '>', '>',
21     '>', '>', '>', '>', '<', '<', '<', '>', '>',
22     '>', '>', '>', '>', '>', '<', '<', '>', '>',
23     '>', '>', '>', '>', '>', '>', ' ', '>', '>',
24     '<', '<', '<', '<', '<', '<', '<', '=', ' ',
25     ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',
26     '<', '<', '<', '<', '<', '<', '<', ' ', '='
27 };
28
29
30 #define IsRoot ( !((x).parent) )
31 #define IsLc(x) ( ! IsRoot(x) && ( &(x) == (x).parent->lc ) )
32 #define IsRc(x) ( ! IsRoot(x) && ( &(x) == (x).parent->rc ) )
33 // #define FromParentTo(x) ( IsRoot(x) )
34
35
36 int MAX_LEN = 0;
37
38 char order(string a, string b)
39 {
40     static map<string, int> symbols;
41     symbols["+"] = ADD;
42     symbols["-"] = SUB;
43     symbols["*"] = MUL;
44     symbols["/"] = DIV;
45     symbols["^"] = POW;
```

¹代码也可从该网址得到: https://github.com/chenfeng123456/CourseInOUC/blob/master/algorithm/expression_tree/expression_tree.cpp

```

46 symbols["!"] = FAC;
47 symbols["("] = L_P;
48 symbols[")"] = R_P;
49 symbols["\\0"] = EOE;
50
51 char res = '\\0';
52 if (symbols.find(a) != symbols.end() && symbols.find(b) != symbols.end())
53 res = pri[symbols[a]][symbols[b]];
54 else
55 throw "Illegal expresssion!";
56
57 return res;
58 }
59
60 template <typename T>
61 class Node
62 {
63 public:
64 T data;
65 Node<T> *parent;
66 Node<T> *lc;
67 Node<T> *rc;
68 int height;
69
70 Node() : parent(NULL), lc(NULL), rc(NULL), height(0) {}
71 Node(T e, Node<T> *p=NULL, Node<T> *l=NULL, Node<T> *r=NULL, int h=0) :
72 data(e), parent(p), lc(l), rc(r), height(h) {}
73 Node<T> *insertAsLc(Node<T> *e);
74 Node<T> *insertAsRc(Node<T> *e);
75 //~Node(){if(!parent) {remove(this);cout << "All nodes have been removed" << endl;}}
76 };
77 template <typename T>
78 Node<T>* Node<T>::insertAsLc(Node<T> *e)
79 {
80 lc = e;
81 e->parent = this;
82 return e;
83 }
84 template <typename T>
85 Node<T>* Node<T>::insertAsRc(Node<T> *e)
86 {
87 rc = e;
88 e->parent = this;
89 return e;
90 }
91
92
93
94 template <typename T>
95 void travPost(Node<T> *x)
96 {

```

```

97  if (!x) return;
98  travPost(x->lc);
99  travPost(x->rc);
100 cout << x->data << " ";
101 }
102
103 bool isdigit(string s)
104 {
105     int i = 0;
106     while ((s[i] >= '0' && s[i] <= '9') || (s[i] >= 'a' && s[i] <= 'z') || (s[i] >= 'A' && s[i]
        <= 'Z') || s[i] == '.')
107     {
108         i++;
109     }
110     return (i == s.length());
111 }
112
113 vector<string> toRPN(vector<string> s)
114 {
115     vector<string> RPN;
116     stack<string> optr;
117     optr.push("\\0");
118     int i = 0;
119     while (!optr.empty())
120     {
121         if (isdigit(s[i]))
122         {
123             RPN.push_back(s[i]);
124             i++;
125         }
126         else
127         {
128             char r;
129             try
130             {
131                 r = order(optr.top(), s[i]);
132             } catch(const char* msg)
133             {
134                 throw "Illegal expression!";
135                 return RPN;
136             }
137             switch (r)
138             {
139                 case '<':
140                 {
141                     optr.push(s[i]);
142                     i++;
143                     break;
144                 }
145                 case '=':
146                 {

```



```

147 optr.pop();
148 i++;
149 break;
150 }
151 case '>':
152 {
153     string op = optr.top();
154     optr.pop();
155     RPN.push_back(op);
156     break;
157 }
158 default :
159 {
160     throw "Illegal expression!";
161 }
162 }
163 }
164 }
165
166 return RPN;
167 }
168
169
170 template <typename T>
171 void rm(Node<T> *x)
172 {
173     if (!x) return;
174     rm(x->lc);
175     rm(x->rc);
176     //cout << x->data << " has been removed" << endl;
177     delete x;
178 }
179
180 template <typename T>
181 void remove(Node<T> *x)
182 {
183     if (x->parent)
184     {
185         if (x == x->parent->lc)
186             x->parent->lc = NULL;
187         if (x == x->parent->rc)
188             x->parent->rc = NULL;
189     }
190     rm(x);
191 }
192
193 template <typename T>
194 void printTree(Node<T> *root, int depth)
195 {
196     if (root == NULL)
197     {

```

```

198 cout << endl;
199 return;
200 }
201 printTree(root->rc, depth+1);
202 for (int i=0; i < depth; i++)
203 for (int j=0; j < MAX_LEN; j++)
204 cout << "    ";
205 cout << root->data;
206 printTree(root->lc, depth+1);
207 }
208
209
210 int main()
211 {
212 cout << "姓名: 鲁国锐" << endl;
213 cout << "学号: 17020021031" << endl;
214 cout << endl;
215
216 vector<string> s;
217 vector<string> RPN;
218 while(1)
219 {
220 string c;
221 while(cin >> c)
222 {
223 s.push_back(c);
224 }
225 s.push_back("\\0");
226
227 try
228 {
229 RPN = toRPN(s);
230 } catch(const char* msg)
231 {
232 cerr << msg << endl;
233 cin.clear();
234 cin.sync();
235 s.clear();
236 continue;
237 }
238 break;
239 }
240
241
242 cout << "后缀表达式: ";
243 for (int i=0; i < RPN.size(); i++)
244 cout << RPN[i] << " ";
245 cout << endl;
246
247 stack<Node<string*>> tr;
248 for (int i=0; i < RPN.size(); i++)

```

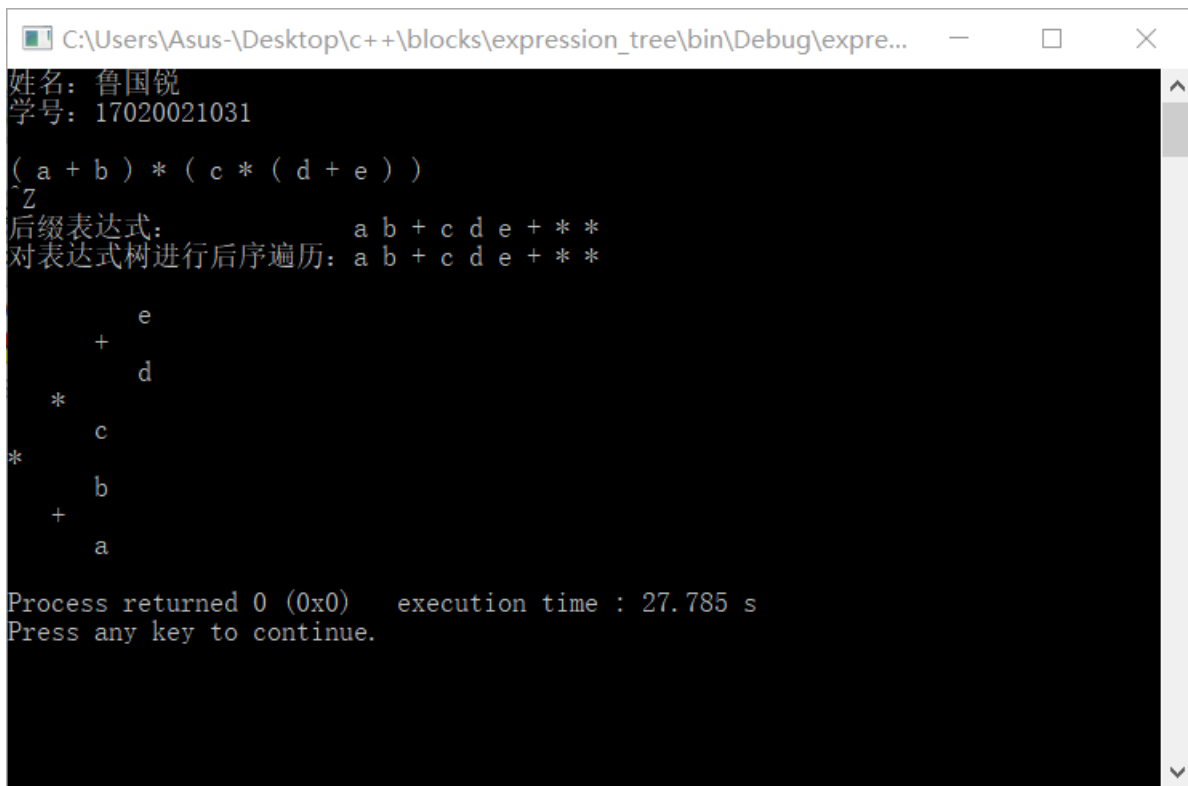
```

249 {
250     string temp_op = RPN[i];
251     MAX_LEN = (MAX_LEN > temp_op.length()) ? MAX_LEN : temp_op.length();
252     Node<string> *p = new Node<string>(temp_op);
253     if (isdigit(temp_op))
254         tr.push(p);
255     else
256     {
257         Node<string> *r = tr.top();
258         tr.pop();
259         // For unary operator such as "!".
260         // If they are placed at the second place
261         // then we shouldn't perform the operation
262         // of top() or pop() on the tr. Otherwise,
263         // it will be runtime error.
264         Node<string> *l;
265         if (!tr.empty())
266             l = tr.top();
267         if (temp_op != "!")
268         {
269             tr.pop();
270         }
271         p->insertAsRc(r);
272         if (temp_op != "!") p->insertAsLc(l);
273         tr.push(p);
274     }
275 }
276
277 cout << "对表达式树进行后序遍历： ";
278 travPost(tr.top());
279 cout << endl;
280 printTree(tr.top(), 0);
281 remove(tr.top());
282
283 return 0;
284 }

```

5 结果分析

5.1 结果展示



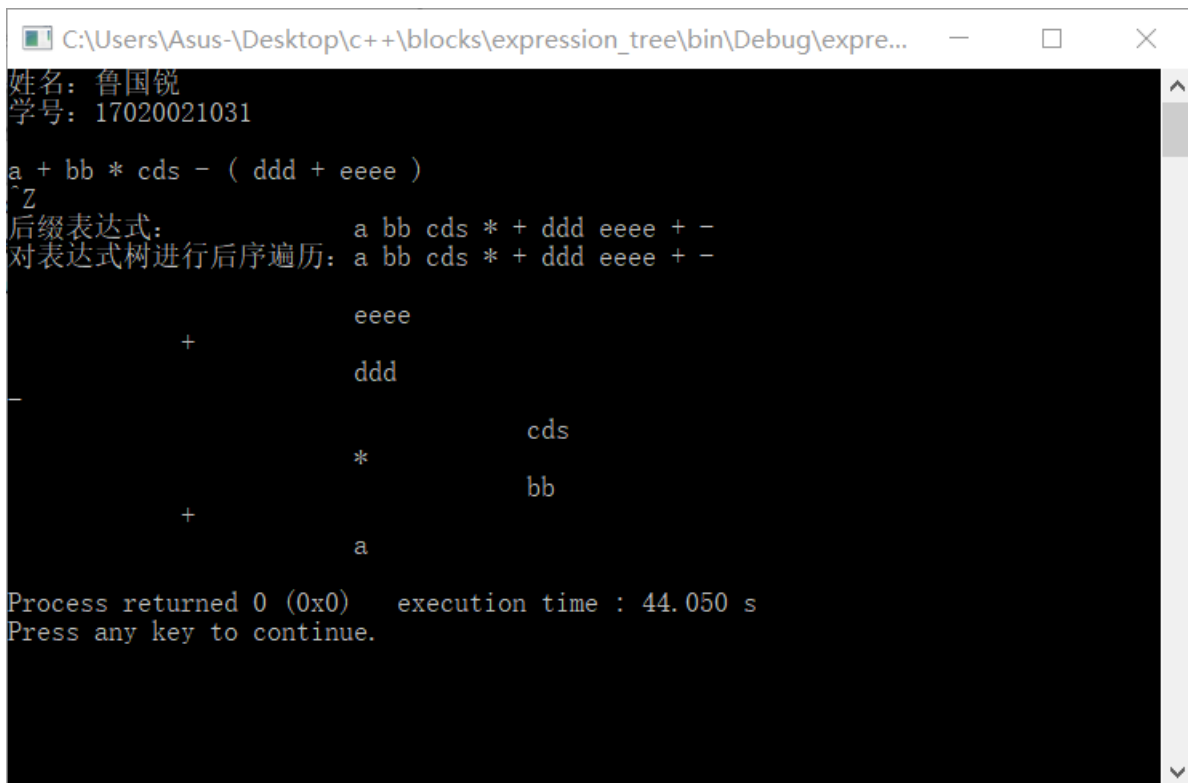
```
C:\Users\Asus\Desktop\c++\blocks\expression_tree\bin\Debug\expre...
姓名: 鲁国锐
学号: 17020021031

( a + b ) * ( c * ( d + e ) )
^Z
后缀表达式:          a b + c d e + * *
对表达式树进行后序遍历: a b + c d e + * *

      e
     +
    d
   *
  c
 *
b
+
a

Process returned 0 (0x0)   execution time : 27.785 s
Press any key to continue.
```

图 3: 结果 1



```
C:\Users\Asus\Desktop\c++\blocks\expression_tree\bin\Debug\expre...
姓名: 鲁国锐
学号: 17020021031

a + bb * cds - ( ddd + eeee )
^Z
后缀表达式:          a bb cds * + ddd eeee + -
对表达式树进行后序遍历: a bb cds * + ddd eeee + -

      eeee
     +
    ddd
   -
  cds
 *
bb
+
a

Process returned 0 (0x0)   execution time : 44.050 s
Press any key to continue.
```

图 4: 结果 2

```

C:\Users\Asus\Desktop\c++\blocks\expression_tree\bin\Debug\expre...
姓名: 鲁国锐
学号: 17020021031

1 + ( ( 232 + 23413 ) * 443242 ) - 500924.223
^Z
后缀表达式:      1 232 23413 + 443242 * + 500924.223 -
对表达式树进行后序遍历: 1 232 23413 + 443242 * + 500924.223 -

                        500924.223
                        -
                    443242
                        *
                    23413
                        +
                    232
                        +
                    1

Process returned 0 (0x0)   execution time : 24.054 s
Press any key to continue.

```

图 5: 结果 3

```

C:\Users\Asus\Desktop\c++\blocks\expression_tree\bin\Debug\expr...
姓名: 鲁国锐
学号: 17020021031

1 +2
^Z
Illegal expression!
1 + %
^Z
Illegal expression!
1- 2
^Z
Illegal expression!
( 1 + 2
^Z
Illegal expression!
) 1 + 2 (
^Z
Illegal expression!
! 2
^Z
后缀表达式:      2 !
对表达式树进行后序遍历: 2 !

    2
!

```

图 6: 结果 4

```
C:\Users\Asus\Desktop\c++\blocks\expression_tree\bin\Debug\expressio...
( 0 ! + 1 ) * 2 ^ ( 3 ! + 4 ) - ( 5 ! - 67 - ( 8 + 9 ) )
~Z
后缀表达式: 0 ! 1 + 2 3 ! 4 + ^ * 5 ! 67 - 8 9 + - -
对表达式树进行后序遍历: 0 ! 1 + 2 3 ! 4 + ^ * 5 ! 67 - 8 9 + - -

      9
     +
    8
   -
  67
 -
 5
!
4
+
3
!
2
*
1
+
0
!
```

图 7: 结果 5

5.2 分析

从图6中可以看出，仍有部分非法输入的情况代码无法分辨出来，但因为它们并不会导致诸如数组越界这样的错误，程序可以正常运行、退出，再加之时间有限，故没有再去单独考虑这些情况。

6 总结体会

在完成此次作业的过程中遇到了很多问题，其中有两个比较隐蔽的 *bug*，不太容易发现，在此稍作总结。

6.1 被忽略的空字符

在实现 *to_RPN* 的过程中，我发现数组总会越界，找了很久的原因，最后发现是栈内和输入末端的空字符没有起作用。分析其原因大概是由于 *string* 本质上是一个字符数组，而我将一个空字符作为一个字符串，系统在扫到空字符的时候就停止了，所以导致最后比较优先级时会出现意外的结果。

6.2 循环析构

对于树或者链表的析构，应当放在树类或链表类中进行。因为如果把析构函数放在节点类中的话，程序最后要释放每一个节点，调用其析构函数，而在析构函数中又会对节点自身进行 *delete* 操作，从而再次触发对析构函数的调用，形成死循环。所以 *delete* 操作必须放在节点外进行。

参考文献

[1] 邓俊辉. 数据结构 (c++ 语言版). 清华大学出版社. 3