数据结构：list ADT

姓名：鲁国锐　　学号：17020021031　　专业：电子信息科学与技术

2019 年 4 月 9 日

Contents

1 问题分析

1.1 题目描述

自己编写 *list ADT*, 调试并完成下面内容。注：本次作业不允许使用 $c++$ 标准模版库。

- 任务一：
  - 实现 $PrintLots(L,P)$, 并分析运行时间。
  - 有两个链表 L 和 P, 他们包含以升序排列的整数。操作 $PrintLots(L,P)$ 将打印 L 中那些由 P 所指定位置上的元素。如：$P=1,3,4,6$, 则，L 中第 1，第 3，第 4，第 6 个元素被打印出来。
- 任务二：懒惰删除
  - 列出懒惰删除的优点和缺点
  - 编写实现

    不同于我们已经给出的删除方法，另一种是使用懒惰删除（*lazy deletion*）。为了删除一个元素，我们只标记上该元素被删除。表中被删除和非被删除元素的个数作为数据结构的一部分被保留。如果被删除元素和非被删除元素一样多，我们遍历整个表，对所有被标记的节点执行标准的删除算法。

1.2 问题分析

根据题目，我们需要解决的问题有：

1. 编写 *Node* 和 *List* 类，包括实现 *List* 中一系列必须的成员函数，如：构造函数、析构函数、*insert* 函数、*remove* 函数等（见2.1节）；
2. 设计输入输出流程（见2.2节）；
3. 实现 *PrintLots* 函数（见2.3节）和 *lazy deletion* 函数（见2.4节）及其相关的部分函数；
4. 分析 *PrintLots* 的时间复杂度（见5.2节）；
5. 分析懒惰删除的有点和缺点（见5.3节）。

2 解决方案

2.1 编写 *Node* 类和 *List* 类

我们首先要写 *Node* 类，它的属性包括值 *val*、指向前驱的指针 *pred*、指向后继的指针 *succ* 以及在本题中需额外设置的一个属性 *deleted*，用以表示该节点是否被标记为删除。其构造函数相对简单，在此不做赘述。

接下来是 *List* 类，它的属性包括头指针 *header*、尾指针 *trailer*、长度 *_size*（不包括头指针和尾指针）和被标记为删除的元素个数 *_delNum*。至于其成员函数（参考教材 [1]），由于数量较多，我们只用一表格列出，并做简要说明。

表 1: *List* 成员函数

函数名	输入	输出	说明
构造函数	无	无	初始化整个链表，建立头结点和尾节点
析构函数	无	无	删除链表，释放内存
insert	指针 r，值 e	该节点的指针	在 r 所指向的节点之前插入一个节点
push_back	值 e	无	在尾节点之前插入一个结点
remove	指针 r	被删除节点下一个节点的指针	删除 r 指向的节点
clear	无	被删除元素的个数	清空链表（不包括头结点和尾节点）
clear_mark	无	无	调用 *remove* 函数清除所有被标记为已删除的节点
lazeDel	指针 r	被删除元素的值	用懒惰删除的方式删除 r 指向的节点
getP	序号（从 1 开始计数）	指向序号所对应的节点的指针	将输入的序号转换为指针，方便后续的操作
print	无	无	输出链表中所有节点的值（不包括头结点和尾节点）
getHeader	无	头指针	返回头指针
getTrailer	无	尾指针	返回尾指针
size	无	链表长度（不包括头结点和尾节点）	返回链表长度（不包括头结点和尾节点）

2.2 设计输入输出流程

我们首先输入链表 L 各节点的值，这些值以空格分隔，以回车结束。然后通过三条指令来控制这个程序的流程：

- *delete*：执行懒惰删除操作，后跟一数字表示要删除节点的序号；
- *PrintLots*：调用 *PrintLots* 函数，后跟一组以空格分隔的数字，表示链表 P 中各节点的元素；
- *end*：结束整个程序。

每次执行完 *delete* 或 *PrintsLots* 后，把 L 中各节点的值输出一遍。其中在执行 *PrintLots* 时还会把 P 以及 L 中对应于 P 所表示的序号的节点值输出一遍，分别以“P”和“L”开头。

2.3 实现 *PrintLots* 函数

有了表1中的各个函数，我们实现 *PrintLots* 就会轻松不少。

首先我们要检查 L 是否为空，若为空则直接结束函数的调用。若不为空，我们开始执行输出操作。这里定义了两个指针 pl 和 pp，分别初始化为指向 L 的头节点和 P 的第一个节点。同时为了避免在得到 P 中节点的值后要从头开始遍历 L，我们还需定义两个整型变量 pre 和 $offset$。其中 pre 中记录的是 P 中前一个节点的值，而 $offset$ 中记录的则是 pl 需要向后移动的位数。每次我们令 pp 向后移动一位，然后用 pp 指向的节点中的值减去 pre，所得结果赋给 $offset$，然后令 pl 向后移 $offset$ 位，输出其对应节点中的值。

另外在执行输出操作时我们还要不断检查 pl 是否越界。如果发现 pl 指向了尾节点，立马抛出异常并结束整个函数。

2.4 实现 *lazy deletion* 函数

这个函数实现起来相对简单。我们只需要把传入指针对应的节点中 *deleted* 属性置为 *true* 即可。同时还要将链表中的 $\_delNum$ 加一。然后判断 $\frac{size}{2} \leq \_delNum$ 是否成立。若成立，则执行 $clear_mark$ 函数，将链表中所有 *deleted* 为 *true* 的节点删除，并把 $\_delNum$ 值 0。

3 算法设计

见图1、图2。

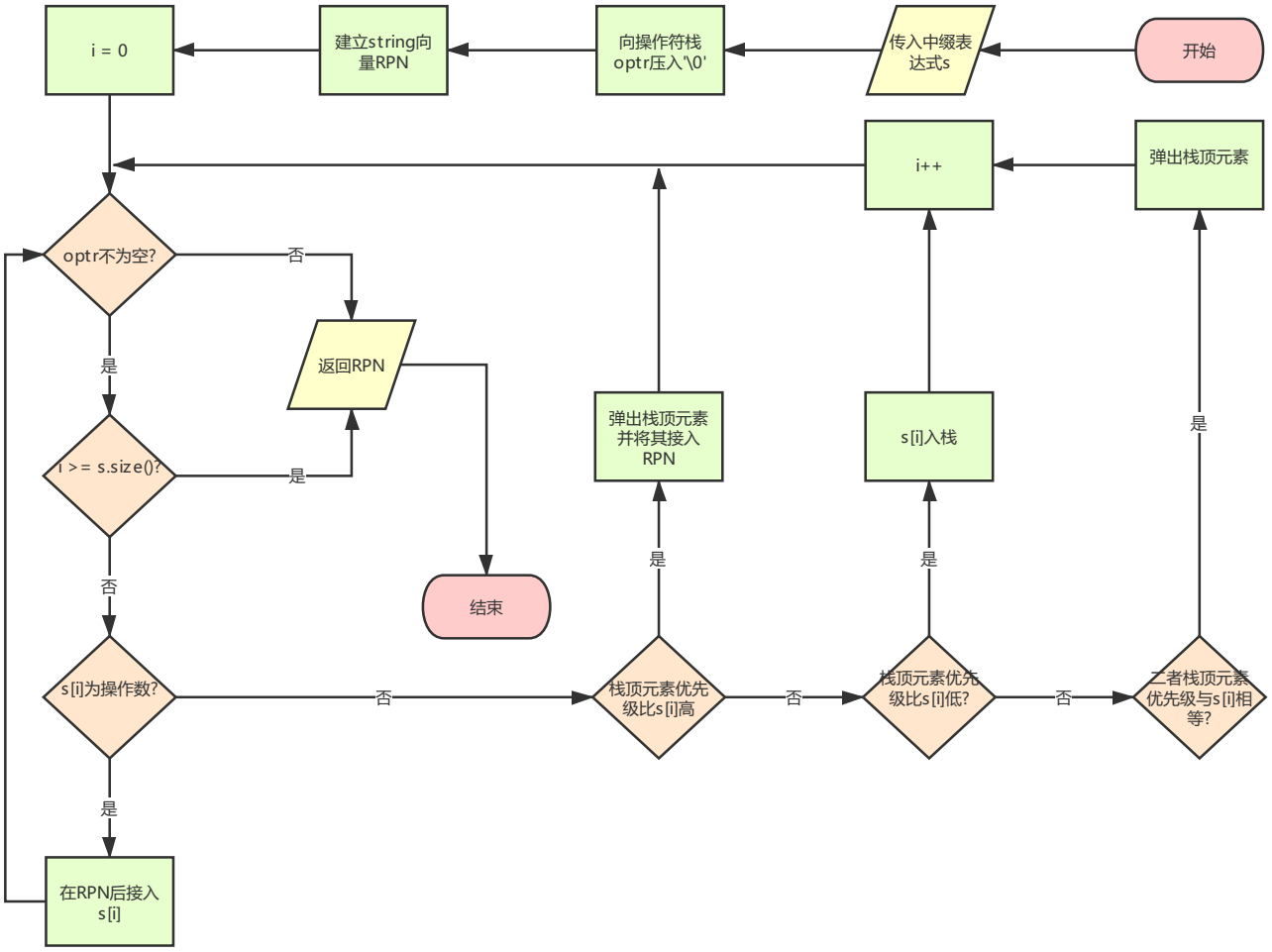

图 1: to_RPN 函数流程图

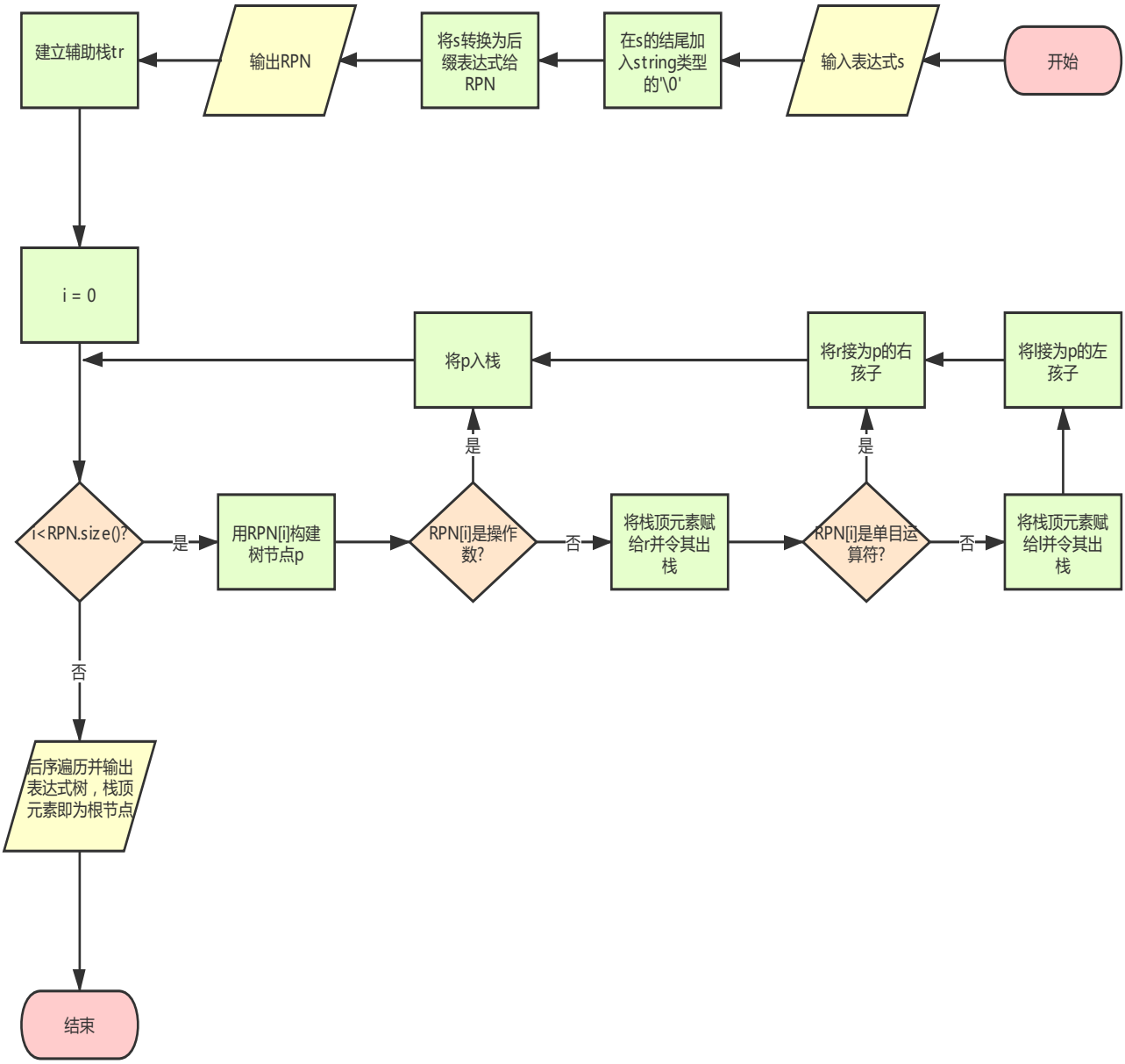

图 2: *main* 函数流程图

4 编程实现[1]

Listing 1: expression_tree 代码

```
#include <iostream>
#include <string>
#include <stack>
#include <vector>
#include <typeinfo>
#include <map>


using namespace std;
#define N_OPTR 9
typedef enum { ADD, SUB, MUL, DIV, POW, FAC, L_P, R_P, EOE } Operator;
//             +    -    *    /    ^    !    (    )    \0

const char pri[N_OPTR][N_OPTR] =
{
'>',   '>',   '<',   '<',   '<',   '<',   '<',   '>',   '>',
'>',   '>',   '<',   '<',   '<',   '<',   '<',   '>',   '>',
'>',   '>',   '>',   '>',   '<',   '<',   '<',   '>',   '>',
'>',   '>',   '>',   '>',   '<',   '<',   '<',   '>',   '>',
'>',   '>',   '>',   '>',   '>',   '<',   '<',   '>',   '>',
'>',   '>',   '>',   '>',   '>',   '>',   ' ',   '>',   '>',
'<',   '<',   '<',   '<',   '<',   '<',   '<',   '=',   ' ',
' ',   ' ',   ' ',   ' ',   ' ',   ' ',   ' ',   ' ',   ' ',
'<',   '<',   '<',   '<',   '<',   '<',   '<',   ' ',   '='
};


#define IsRoot ( !((x).parent))
#define IsLc(x) ( ! IsRoot(x) && ( &(x) == (x).parent->lc))
#define IsRc(x) ( ! IsRoot(x) && ( &(x) == (x).parent->rc))
//#define FromParentTo(x) (IsRoot(x)

char order(string a, string b)
{
static map<string, int> symbols;
symbols["+"] = ADD;
symbols["-"] = SUB;
symbols["*"] = MUL;
symbols["/"] = DIV;
symbols["^"] = POW;
symbols["!"] = FAC;
symbols["("] = L_P;
symbols[")"] = R_P;
symbols["\0"] = EOE;

```

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

```
return pri[symbols[a]][symbols[b]];
}

template <typename T>
class Node
{
public:
T data;
Node<T> *parent;
Node<T> *lc;
Node<T> *rc;
int height;

Node() : parent(NULL), lc(NULL), rc(NULL), height(0) {}
Node(T e, Node<T> *p=NULL, Node<T> *l=NULL, Node<T> *r=NULL, int h=0) :
data(e), parent(p), lc(l), rc(r), height(h) {}
Node<T> *insertAsLc(Node<T> *e);
Node<T> *insertAsRc(Node<T> *e);
};
template <typename T>
Node<T>* Node<T>::insertAsLc(Node<T> *e)
{
lc = e;
e->parent = this;
return e;
}
template <typename T>
Node<T>* Node<T>::insertAsRc(Node<T> *e)
{
rc = e;
e->parent = this;
return e;
}



template <typename T>
void travPost(Node<T> *x)
{
if (!x) return;
travPost(x->lc);
travPost(x->rc);
cout << x->data << " ";
}

bool isdigit(string s)
{
if ((s[0] >= '0' && s[0] <= '9') || (s[0] >= 'a' && s[0] <= 'z'))
return true;
return false;
}
```

```
97
98  vector<string>& toRPN(vector<string> s)
99  {
100 static vector<string> RPN;
101 //stack<string> opnd;
102 stack<string> optr;
103 optr.push("\0");
104 int i = 0;
105 while (!optr.empty())
106 {
107 //cout << endl;
108 //cout << "i = " << i << endl;
109 //cout << "RPN = " << RPN << endl;
110 //cout << "optr.size() = " << optr.size() << endl;
111 if (i >= s.size())
112 break;
113 if (isdigit(s[i]))
114 {
115 RPN.push_back(s[i]);
116 i++;
117 }
118 else
119 {
120 switch (order(optr.top(), s[i]))
121 {
122 case '<':
123 {
124 //cout << '<' << endl;
125 optr.push(s[i]);
126 i++;
127 break;
128 }
129 case '=':
130 {
131 //cout << '=' << endl;
132 optr.pop();
133 i++;
134 break;
135 }
136 case '>':
137 {
138 //cout << '>' << endl;
139 string op = optr.top();
140 optr.pop();
141 RPN.push_back(op);
142 //i++;
143 break;
144 }
145 }
146 }
147 }
```

```
148
149 return RPN;
150 }
151
152
153 template <typename T>
154 void rm(Node<T> *x)
155 {
156 if (!x) return;
157 rm(x->lc);
158 rm(x->rc);
159 delete x;
160 }
161
162 template <typename T>
163 void remove(Node<T> *x)
164 {
165 if (x->parent)
166 {
167 if (x == x->parent->lc)
168 x->parent->lc = NULL;
169 if (x == x->parent->rc)
170 x->parent->rc = NULL;
171 }
172 rm(x);
173 }
174
175
176 int main()
177 {
178
179 vector<string> s;
180
181 string c;
182 while(cin >> c)
183 {
184 s.push_back(c);
185 }
186 s.push_back("\0");
187 vector<string> RPN = toRPN(s);
188
189 for (int i=0; i < RPN.size(); i++)
190 cout << RPN[i] << " ";
191 cout << endl;
192
193 stack<Node<string>*> tr;
194 for (int i=0; i < RPN.size(); i++)
195 {
196 string temp_op = RPN[i];
197 Node<string> *p = new Node<string>(temp_op);
198 if (isdigit(temp_op))
```

```
199 tr.push(p);
200 else
201 {
202 //cout << RPN[i] << "   " << tr.size() << endl;
203 Node<string> *r = tr.top();
204 tr.pop();
205 // For unary operator such as "!".
206 // If they are placed at the second place
207 // then we shouldn't perform the operation
208 // of top() or pop() on the tr. Otherwise,
209 // it will be runtime error.
210 Node<string> *l;
211 if (!tr.empty())
212 l = tr.top();
213 if (temp_op != "!")
214 {
215 tr.pop();
216 }
217 p->insertAsRc(r);
218 if (temp_op != "!") p->insertAsLc(l);
219 tr.push(p);
220 }
221 }
222
223 //cout << "tr.size() = " << tr.size() << endl;
224 travPost(tr.top());
225 cout << endl;
226 return 0;
227 }
```

5 结果分析

5.1 结果展示

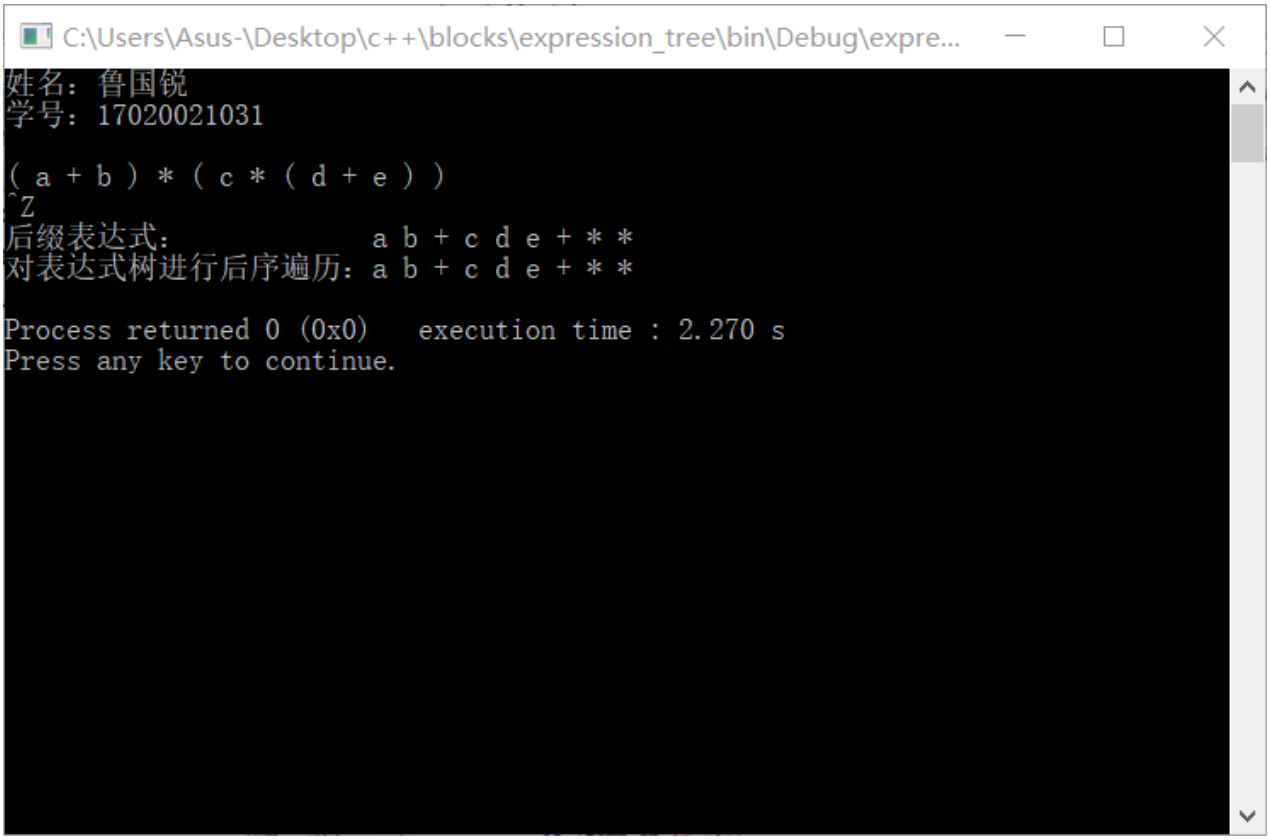
C:\Users\Asus-\Desktop\c++\blocks\expression_tree\bin\Debug\expre...
姓名：鲁国锐
学号：17020021031
(a + b) * (c * (d + e))
^Z
后缀表达式： a b + c d e + * *
对表达式树进行后序遍历：a b + c d e + * *
Process returned 0 (0x0)   execution time : 2.270 s
Press any key to continue.

图 3: 结果 1

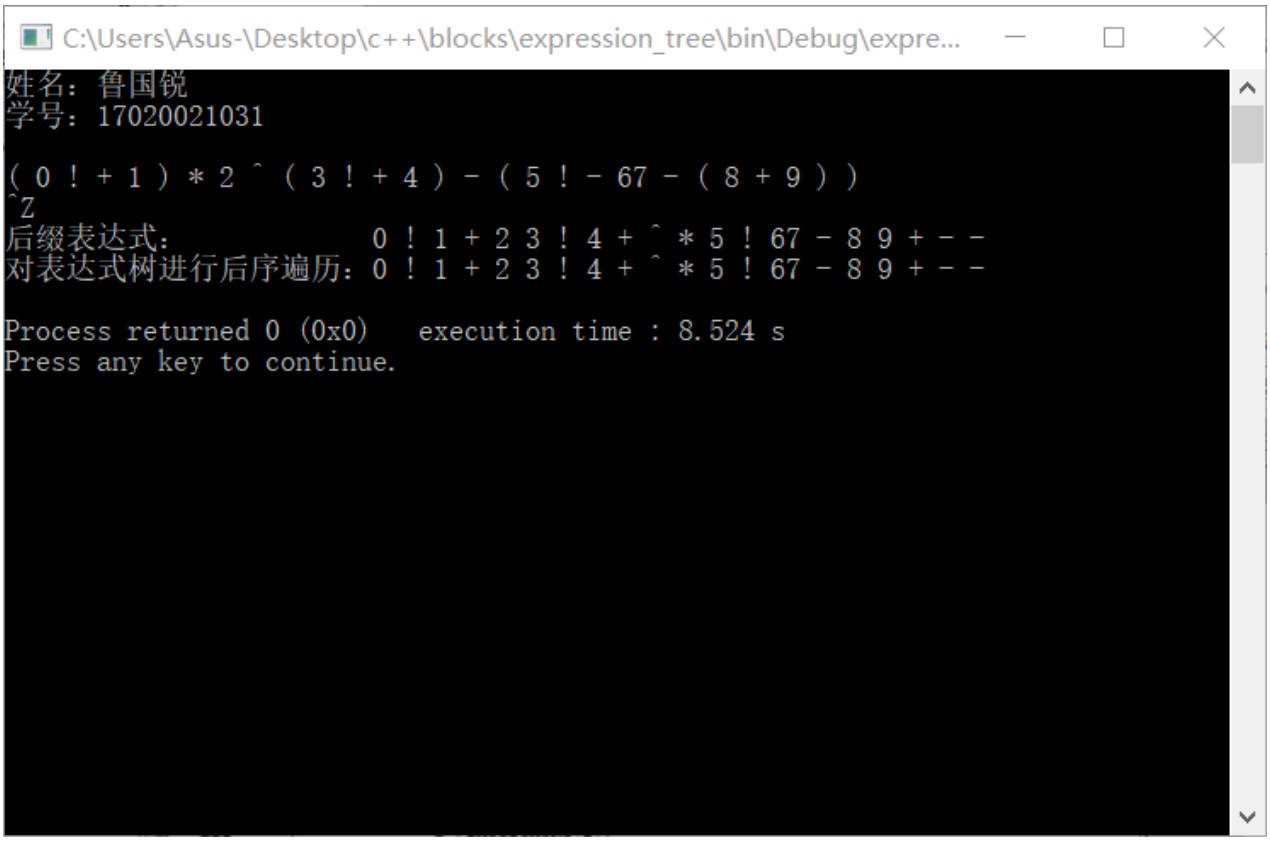
C:\Users\Asus-\Desktop\c++\blocks\expression_tree\bin\Debug\expre...
姓名：鲁国锐
学号：17020021031
(0 ! + 1) * 2 ^ (3 ! + 4) - (5 ! - 67 - (8 + 9))
^Z
后缀表达式： 0 ! 1 + 2 3 ! 4 + ^ * 5 ! 67 - 8 9 + - -
对表达式树进行后序遍历：0 ! 1 + 2 3 ! 4 + ^ * 5 ! 67 - 8 9 + - -
Process returned 0 (0x0)   execution time : 8.524 s
Press any key to continue.

图 4: 结果 2

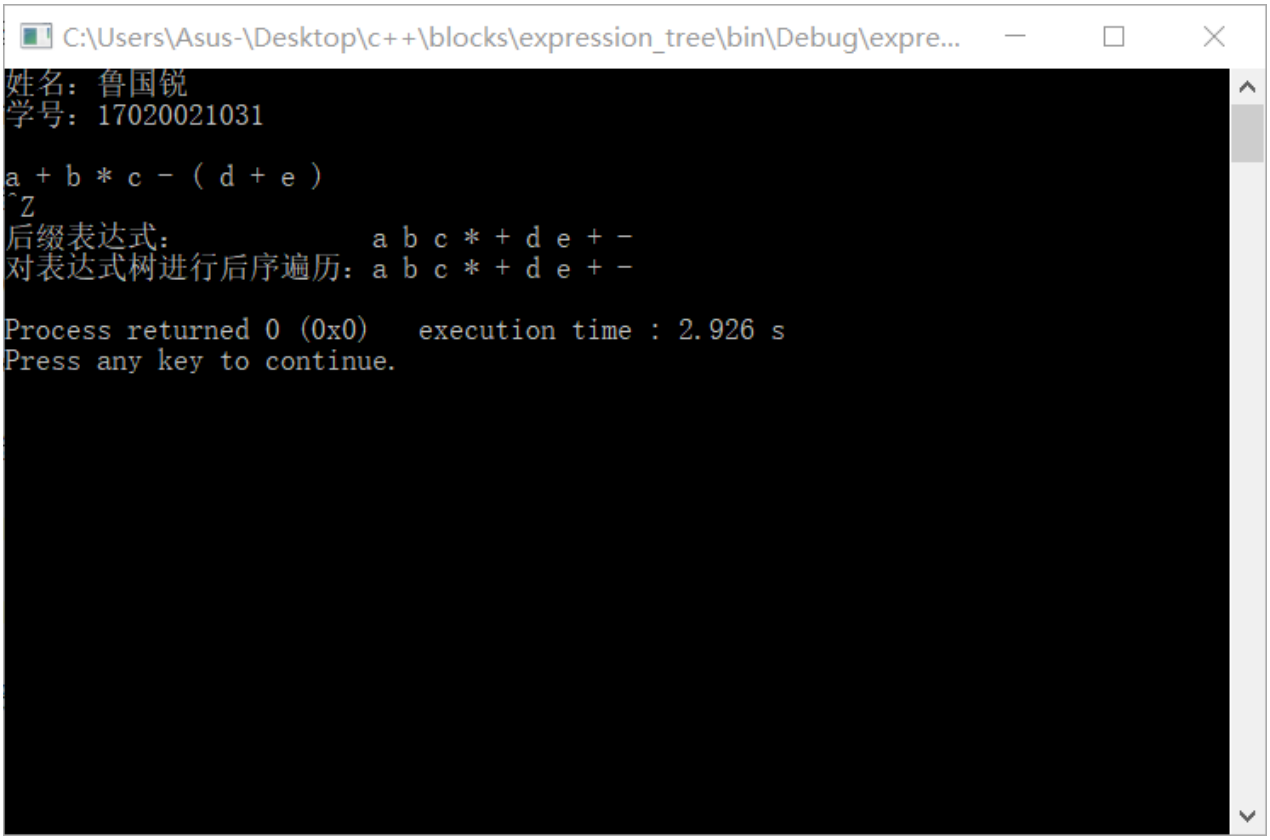
C:\Users\Asus-\Desktop\c++\blocks\expression_tree\bin\Debug\expre...
姓名：鲁国锐
学号：17020021031
a + b * c - (d + e)
^Z
后缀表达式： a b c * + d e + -
对表达式树进行后序遍历：a b c * + d e + -
Process returned 0 (0x0)   execution time : 2.926 s
Press any key to continue.

图 5: 结果 3

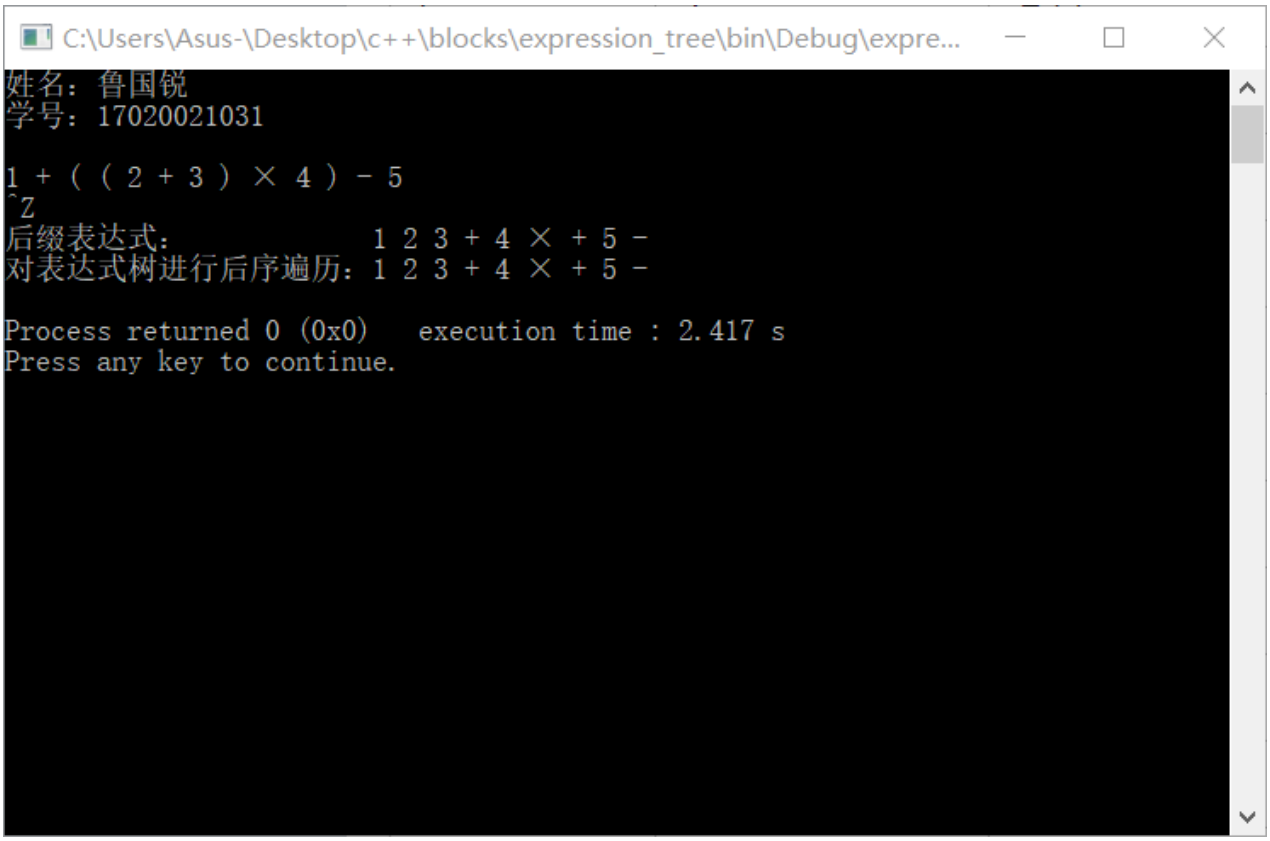
C:\Users\Asus-\Desktop\c++\blocks\expression_tree\bin\Debug\expre...
姓名：鲁国锐
学号：17020021031
1 + ((2 + 3) × 4) - 5
^Z
后缀表达式： 1 2 3 + 4 × + 5 -
对表达式树进行后序遍历：1 2 3 + 4 × + 5 -
Process returned 0 (0x0)   execution time : 2.417 s
Press any key to continue.

图 6: 结果 4

5.2 *PrintLots* 运行时间

由于我们在执行输出操作时引入 pre 和 $offset$ 两个变量从而避免了对 L 的重复遍历。因此，即使在最坏情况下，即需要输出 L 中的最后一个元素时，我们也只需令 pl 向后移动 $L.\_size$ 次，所需时间与 L 的长度成正比。故其时间复杂度为 $O(N)$。

5.3 懒惰删除的优点和缺点

懒惰删除的优点：

- 减少删除节点时所需的时间；
- 避免频繁地对内存进行操作；
- 简化代码复杂程度；
- 在对被删除位置进行插入时**可能**会减少时间。

懒惰删除的缺点：

- 会占用更多的空间；
- 由于在链表中由于并没有真正删除节点，所以会增减遍历链表所需的时间；

6 总结体会

在完成这次作业的过程中，由于一开始没有加入检查指针是否越界的代码，导致在调试过程中因指针越界使得电脑死机了，从而不得不强制关机重启。这次的教训让我更深刻地明白了检查指针越界的重要性，同时还促使我去学习了如何在 $C++$ 中捕获异常并输出错误信息。　　另外还没有完全解决的一点是，因为我不想在输入链表的值之前还要输入长度，所以采取的做法是当检测到输入数字后跟的是回车时就跳出循环。但如果在输入回车之前不小心多输入了空格，并在输入回车后输入命令，也会使电脑几乎陷入死机状态。我暂时还没想出是什么原因导致的，也没有很好的解决方法。

参考文献

[1] 邓俊辉. 数据结构（c++ 语言版）. 清华大学出版社. 2