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 2 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??节）；
2. 设计输入输出流程（见??节）；
3. 实现 *PrintLots* 函数（见??节）和 *lazy deletion* 函数（见??节）及其相关的部分函数；
4. 分析 *PrintLots* 的时间复杂度（见??节）；
5. 分析懒惰删除的有点和缺点（见??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 [?]），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??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

见下页图??、图??。

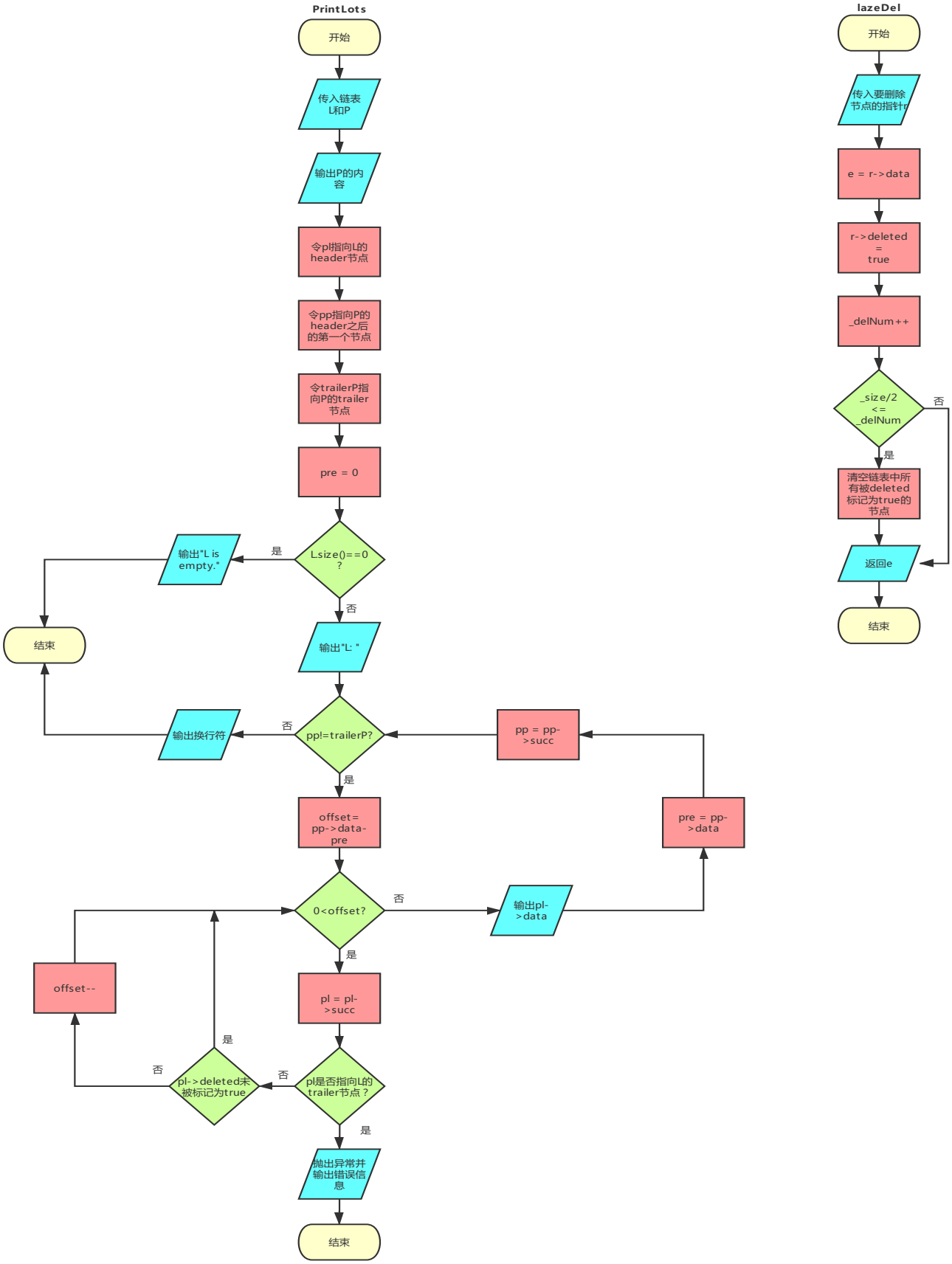

图 1: *PrintLots* 和 *lazeDel* 函数流程图

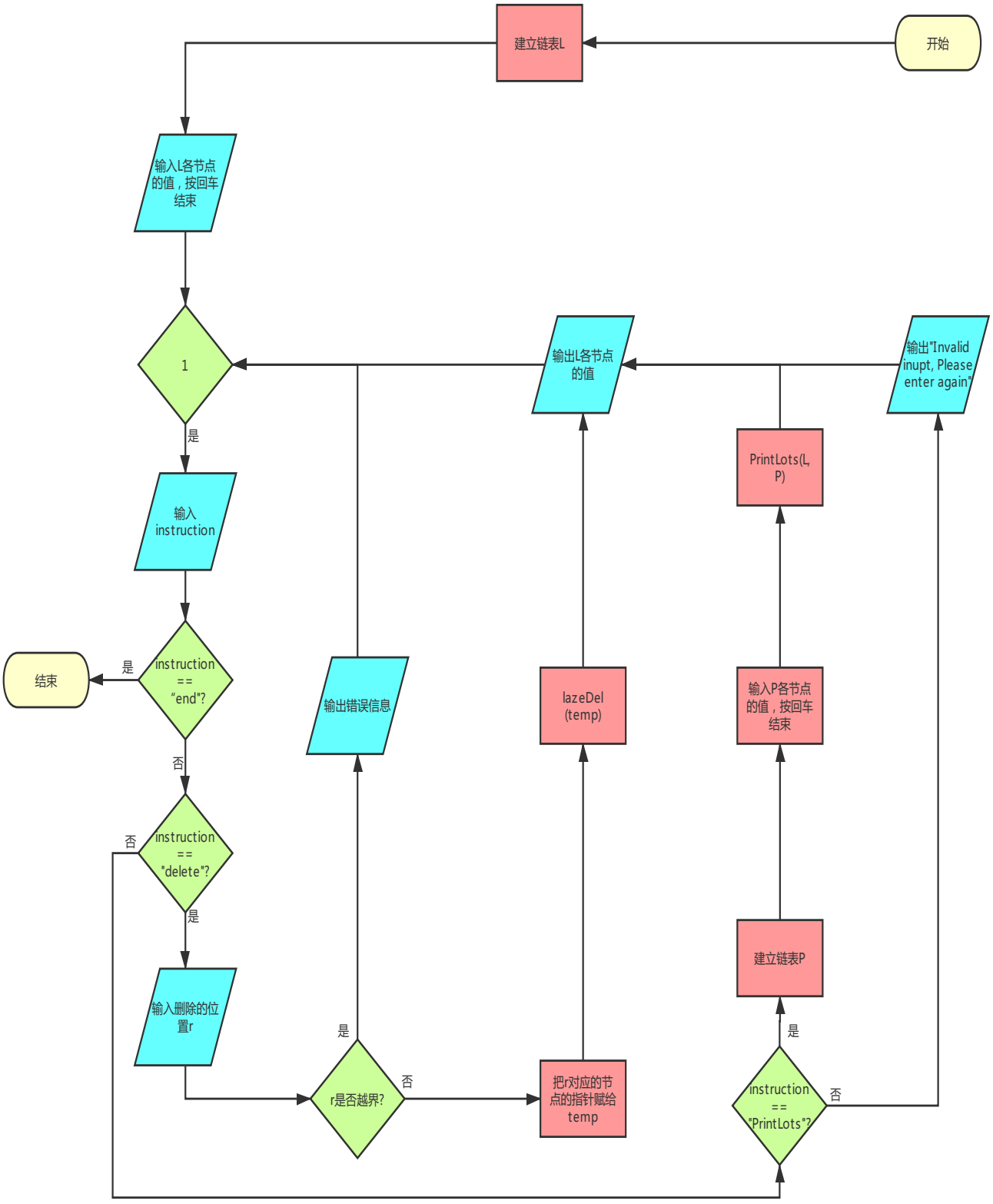

图 2: *main* 函数流程图

4 编程实现[1]

Listing 1: list_ADT 代码

```
#include<iostream>
#include<exception>
#include<stdexcept>
using namespace std;

template <typename T>
class Node
{
    public:
        // member variables
        T data;
        Node<T>* pred;
        Node<T>* succ;
        bool deleted;
        // constructors
        Node() : data(0), pred(NULL), succ(NULL){}
        Node(T e, Node<T>* p=NULL, Node<T>* s=NULL) : data(e), pred(p), succ(s), deleted(
            false) {}
        // operation interface
        //Node<T>* insertAsPred(T const& e);
        //Node<T>* insertAsSucc(T const& e);
};


template <typename T>
class List
{
    private:
        int _size;
        Node<T>* header;
        Node<T>* trailer;
        int _delNum;
    public:
        // constructors
        List();

        // destructor
        ~List();

        // operation interfaces
        Node<T>* insert(Node<T>* r, T const& e);// insert as pred
        void push_back(T e) {insert(trailer, e);}
        Node<T>* remove(Node<T>* r);
        int clear();
        void clear_mark();
        T lazeDel(Node<T>* r);
```

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

```
46          Node<T>* getP(int r);
47          void print();
48          Node<T>* getHeader() {return header;}
49          Node<T>* getTrailer() {return trailer;}
50          int size() {return _size;}
51  };
52  // implement the functions of the List
53  template <typename T>
54  List<T>::List()
55  {
56      header = new Node<T>;
57      trailer = new Node<T>;
58      header->succ = trailer;
59      header->pred = NULL;
60      trailer->pred = header;
61      trailer->succ = NULL;
62      _size = 0;
63      _delNum = 0;
64  }
65
66  template <typename T>
67  Node<T>* List<T>::insert(Node<T>* r, T const& e)
68  {
69      // insert as pred
70      Node<T>* x = new Node<T>(e, r->pred, r);
71      r->pred->succ = x;
72      r->pred = x;
73      _size++;
74      return x;
75  }
76
77  template <typename T>
78  Node<T>* List<T>::remove(Node<T>* r)
79  {
80      r->pred->succ = r->succ;
81      r->succ->pred = r->pred;
82      Node<T>* p = r->succ;
83      delete r;
84      _size--;
85      if (r->deleted) _delNum--;
86      return p;
87  }
88
89  template <typename T>
90  int List<T>::clear()
91  {
92      int oldSize = _size;
93      while(0 < _size)
94          remove(header->succ);
95      return oldSize;
96  }
```

```
97
98  template <typename T>
99  void List<T>::clear_mark()
100 {
101     Node<T>* p = header->succ;
102     while (p != trailer)
103     {
104         if (p->deleted)
105             p = remove(p);
106         else
107             p = p->succ;
108     }
109     _delNum = 0;
110     cout << "remove all nodes marked." << endl;
111 }
112
113 template <typename T>
114 List<T>::~List()
115 {
116     clear();
117     delete header;
118     delete trailer;
119 }
120
121 template <typename T>
122 T List<T>::lazeDel(Node<T>* r)
123 {
124     T e = r->data;
125     r->deleted = true;
126     _delNum++;
127     if (_size/2 <= _delNum)
128         clear_mark();
129     return e;
130 }
131
132 template <typename T>
133 Node<T>* List<T>::getP(int r)
134 {
135     Node<T>* p = header;
136     while (0 < r)
137     {
138         p = p->succ;
139         if (!p->deleted)
140             r--;
141         if (p == trailer)
142             throw out_of_range("Overflow!");
143
144     }
145
146     return p;
147 }
```

```
148
149 template <typename T>
150 void List<T>::print()
151 {
152     if (_size == 0)
153     {
154         cout << "The list is empty." << endl;
155         return;
156     }
157     Node<T>* p = header;
158     while ((p = p->succ) != trailer)
159         if (!p->deleted)
160             cout << p->data << " ";
161     cout << endl;
162 }
163
164
165
166
167 // implement PrintLots
168 template <typename T>
169 void PrintLots(List<T>& L, List<T>& P)
170 {
171     cout << "P: ";
172     P.print();
173     // pl points to the header of L
174     // and pp points to the first element of P
175     Node<T>* pl = L.getP(0);
176     Node<T>* pp = P.getP(1);
177     Node<T>* trailerP = P.getTrailer();
178     int pre = 0;
179     if (!L.size())
180     {
181         cout << "L is empty." << endl;
182         return;
183     }
184     cout << "L: ";
185     while (pp != trailerP)
186     {
187         int offset = pp->data - pre;
188         while (0 < offset)
189         {
190             pl = pl->succ;
191             try
192             {
193                 if (pl == L.getTrailer())
194                     throw out_of_range("Overflow!");
195             }catch(out_of_range &e)
196             {
197                 cerr << endl << "Index " << pp->data <<" is out of range. " << e.what() <<
                        endl;
```

```
198                 return;
199             }
200             if (!pl->deleted)
201                 offset--;
202         }
203         cout << pl->data << " ";
204         pre = pp->data;
205         pp = pp->succ;
206     }
207     cout << endl;
208 }
209
210
211
212
213 int main()
214 {
215     cout << "姓名: 鲁国锐" << endl;
216     cout << "学号: 17020021031" << endl;
217     cout << endl;
218     cout << "instructions:" << endl;
219     cout << "delete: 后跟一数字, 执行懒惰删除操作" << endl;
220     cout << "PrintLots: 后跟若干数字, 务以空格分隔, 以回车结束输入, 为链表P的节点值, 执行
        PrintLots操作" << endl;
221     cout << "end: 结束程序" << endl;
222
223     cout << endl << "请输入L中各节点的值, 务必以空格分隔, 以回车结束输入: " << endl;
224
225     List<int> L;
226
227     while (1)
228     {
229         int data;
230         char c;
231         cin >> data;
232         cin.get(c);
233         L.push_back(data);
234         if (c == '\n')
235             break;
236     }
237
238     while(1)
239     {
240         string instruction;
241         cin >> instruction;
242         if (instruction == "end")
243             break;
244         else if (instruction == "delete")
245         {
246             int r;
247             cin >> r;
```

```
            Node<int>* temp;
            try
            {
                temp = L.getP(r);
            }catch(out_of_range &e)
            {
                cerr << "Index " << r << " is out of range. " << e.what() << endl;
                continue;
            }
            L.lazeDel(temp);
        }
        else if (instruction == "PrintLots")
        {
            List<int> P;
            while (1)
            {
                int data;
                char c;
                cin >> data;
                cin.get(c);
                P.push_back(data);
                if (c == '\n')
                    break;
            }
            PrintLots(L, P);
        }
        else
        {
            cout << "Invalide input, please enter again." << endl;
            continue;
        }


        L.print();
    }
}
```

5 结果分析

5.1 结果展示

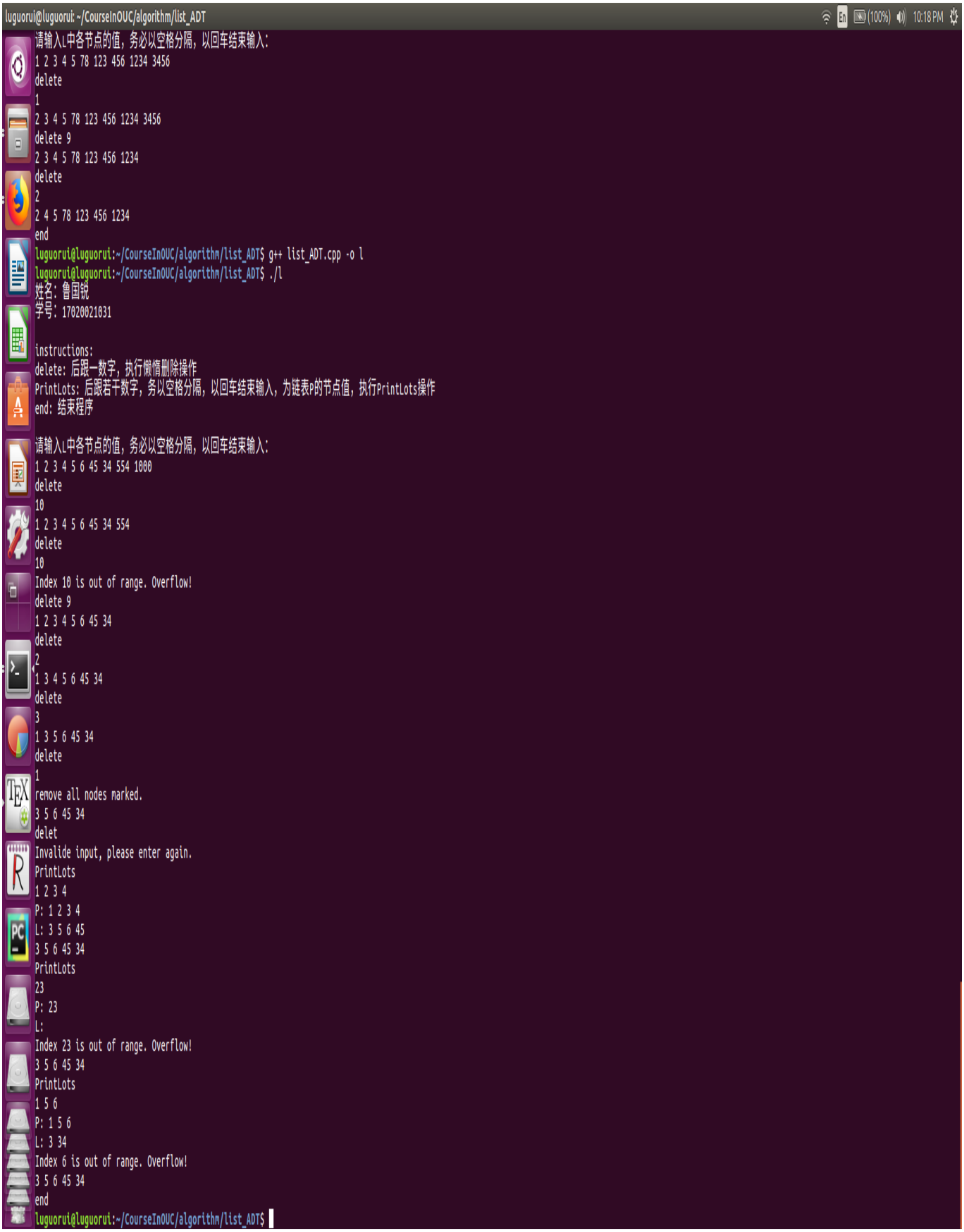
luguorui@luguorui: ~/CourseInOUC/algorithm/list_ADT
请输入L中各节点的值，务必以空格分隔，以回车结束输入:
1 2 3 4 5 78 123 456 1234 3456
delete
1
2 3 4 5 78 123 456 1234 3456
delete 9
2 3 4 5 78 123 456 1234
delete
2
2 4 5 78 123 456 1234
end
luguorui@luguorui:~/CourseInOUC/algorithm/list_ADT$ g++ list_ADT.cpp -o l
luguorui@luguorui:~/CourseInOUC/algorithm/list_ADT$ ./l
姓名: 鲁国锐
学号: 17020021031
instructions:
delete: 后跟一数字，执行懒惰删除操作
PrintLots: 后跟若干数字，务以空格分隔，以回车结束输入，为链表P的节点值，执行PrintLots操作
end: 结束程序
请输入L中各节点的值，务必以空格分隔，以回车结束输入:
1 2 3 4 5 6 45 34 554 1000
delete
10
1 2 3 4 5 6 45 34 554
delete
10
Index 10 is out of range. Overflow!
delete 9
1 2 3 4 5 6 45 34
delete
2
1 3 4 5 6 45 34
delete
3
1 3 5 6 45 34
delete
1
remove all nodes marked.
3 5 6 45 34
delet
Invalide input, please enter again.
PrintLots
1 2 3 4
P: 1 2 3 4
L: 3 5 6 45
3 5 6 45 34
PrintLots
23
P: 23
L:
Index 23 is out of range. Overflow!
3 5 6 45 34
PrintLots
1 5 6
P: 1 5 6
L: 3 34
Index 6 is out of range. Overflow!
3 5 6 45 34
end
luguorui@luguorui:~/CourseInOUC/algorithm/list_ADT$

图 3: 结果

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