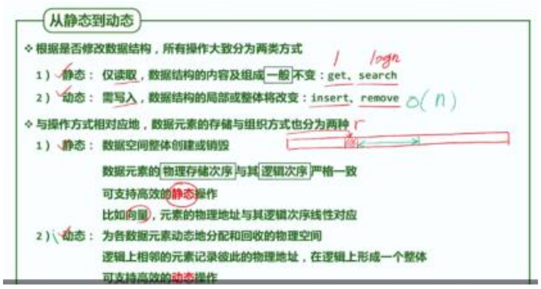
第一节 接口与实现

2016年8月26日　　16:04

5.1.1 从静态到动态



5.1.2 从向量到列表

前驱、后继，首节点、末节点。
各节点通过指针或引用彼此相连，在逻辑上构成一个线性序列

5.1.3 从秩到位置

从循秩访问->循位置访问

5.1.4 实现

列表模板类：

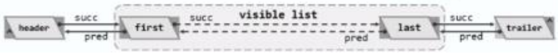
Header头元素——Trailer尾元素　不可见 与生俱来 不相同
first首元素——last末元素　　可见 可相同 可不存在

头、首、末、尾的秩可理解为：-1、0、n-1、n

生成一个空列表：

构造

```
❖ template <typename T> void List<T>::init() { //初始化，创建列表对象时统一调用
      header  = new ListNode<T>; //创建头哨兵节点
      trailer = new ListNode<T>; //创建尾哨兵节点
      header->succ = trailer; header->pred = NULL; //互联
      trailer->pred = header; trailer->succ = NULL; //互联
      _size = 0; //记录规模
  }
```

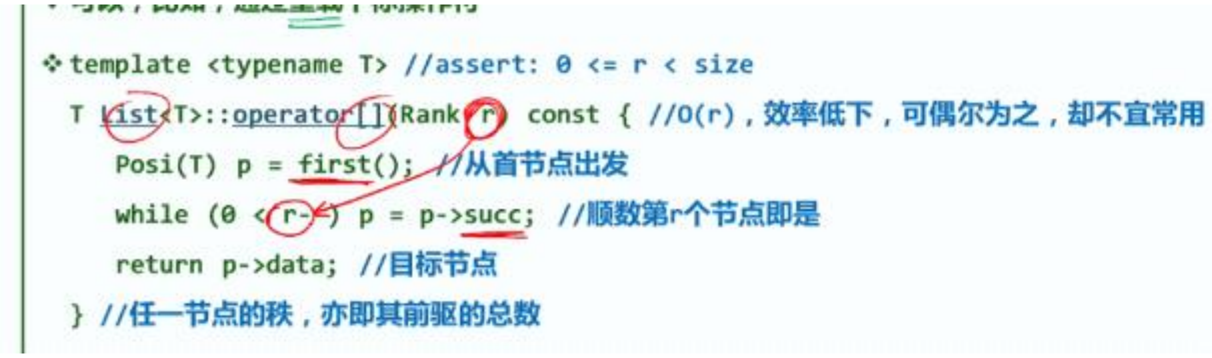


第二节 无序列表

2016年8月26日　　16:51

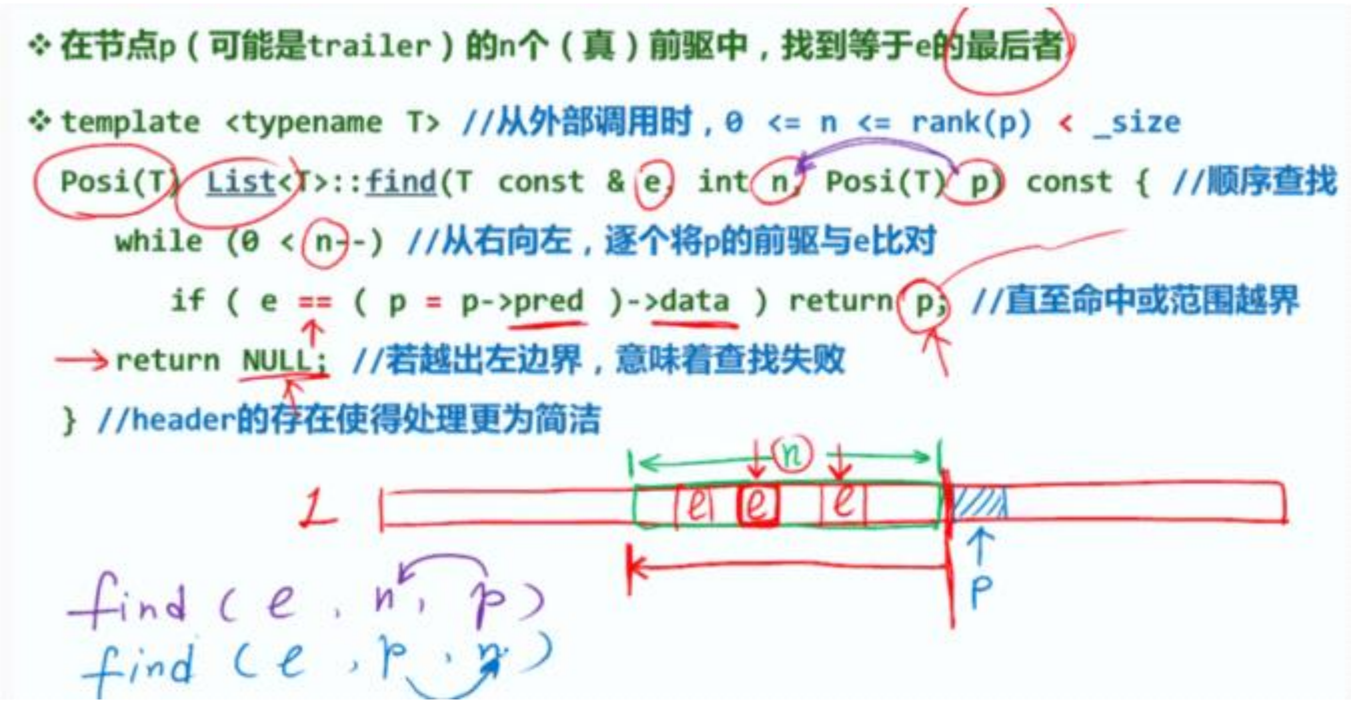
5.2.1 循秩访问

我们为了方便采用向量的方式进行循秩访问，可以采用对[]运算符的重载，使其可以采用下标对列表进行访问。



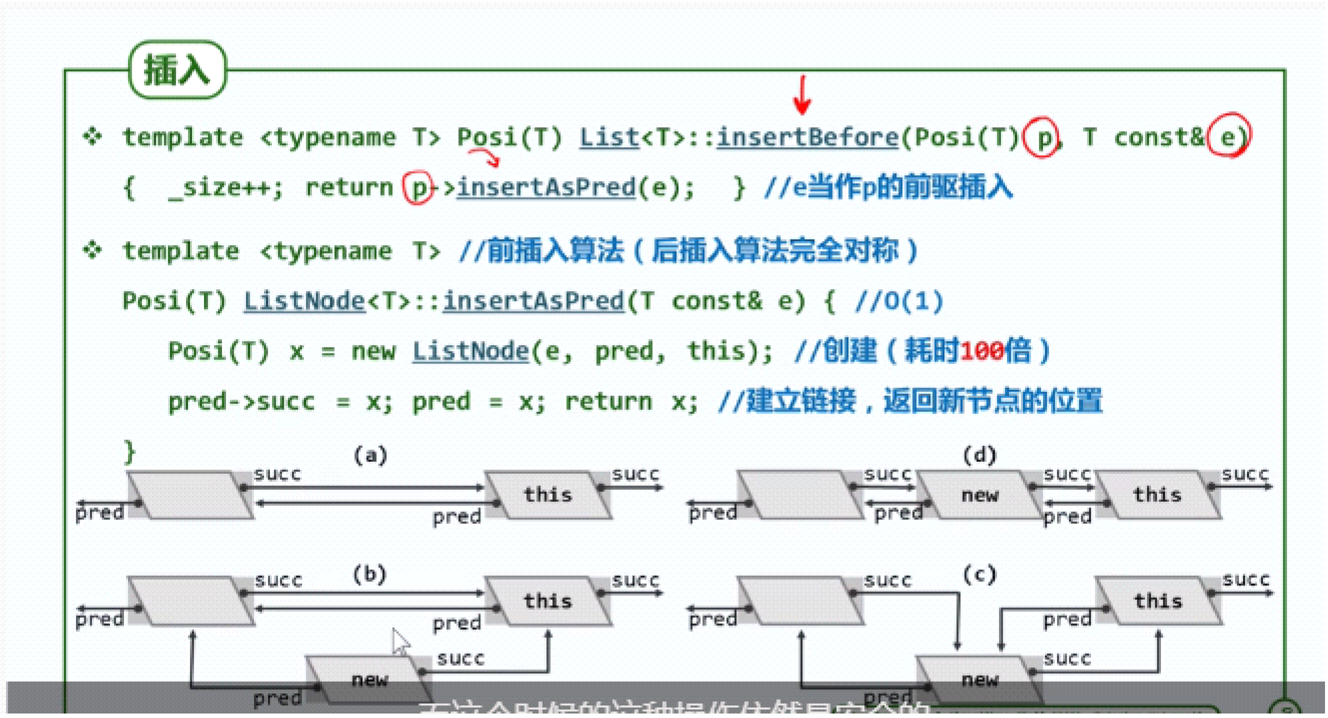
效率低下，复杂度为O(n)

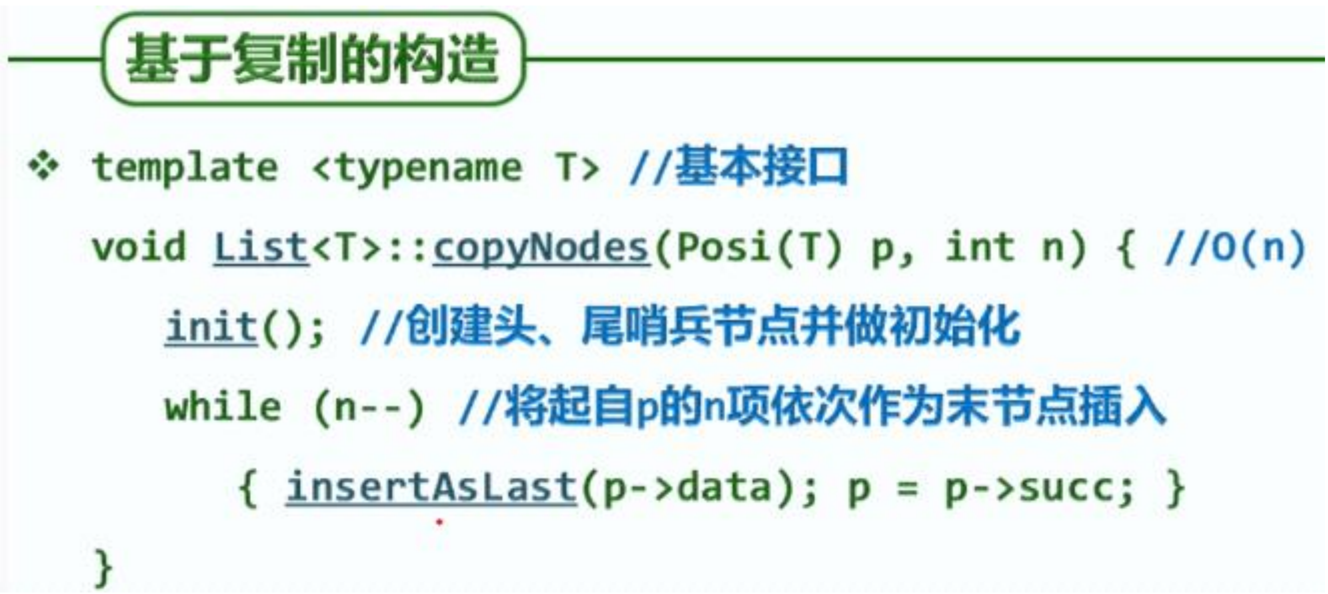
5.2.2 查找



5.2.3 插入与复制

插入：





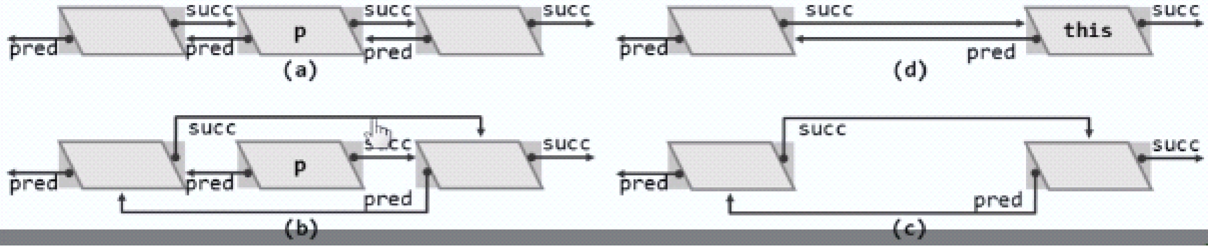
图中的insertaslast()实质上就是insertassbefore(trailer)

5.2.4 删除与析构

删除：

删除

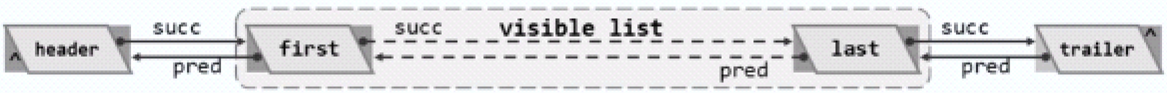
```
❖ template <typename T> //删除合法位置p处节点，返回其数值
  T List<T>::remove(Posi(T) p) { //O(1)
     T e = p->data; //备份待删除节点数值（设类型T可直接赋值）
     p->pred->succ = p->succ;
     p->succ->pred = p->pred;
     delete p; _size--; return e; //返回备份数值
  }
```



析构：首先删除所有可见部分，然后删除两个哨兵header和trailer

析构

```
❖ template <typename T> List<T>::~List() //列表析构
  { clear(); delete header; delete trailer; } //清空列表，释放头、尾哨兵节点
❖ template <typename T> int List<T>::clear() { //清空列表
     int oldSize = _size;
     while (0 < _size) //反复删除首节点，直至列表变空
        remove(header->succ);
     return oldSize;
  } //O(n)，线性正比于列表规模
```



5.2.5 唯一化

选择删除前域中的q而不是当前元p:

删除p后在下一步迭代中，p = p->succ会出现错误（此时p已经不存在）

第三节 有序列表

2016年8月26日 17:33

5.3.1 唯一化•构思

有序列表的唯一化与有序向量唯一化的构思基本相同。

5.3.2 唯一化•实现

由于该算法为线性操作，所以其总体复杂度为O(n)

5.3.3 查找

由于列表访问方式的限制，查找方式的复杂度并不尽如人意。

Vector -> rank -> RAM模型
List -> posi -> 图灵机模型

第四节 选择排序

2016年8月26日 17:57

5.4.1 构思（selection sort）

选择排序：每次选出最大元，直至结束。
e.g. 起泡排序
起泡排序低效率原因：找到最大元后，采用小步慢跑的方式，每次与相邻元交换，没有一步到位。

5.4.2 实例

实例

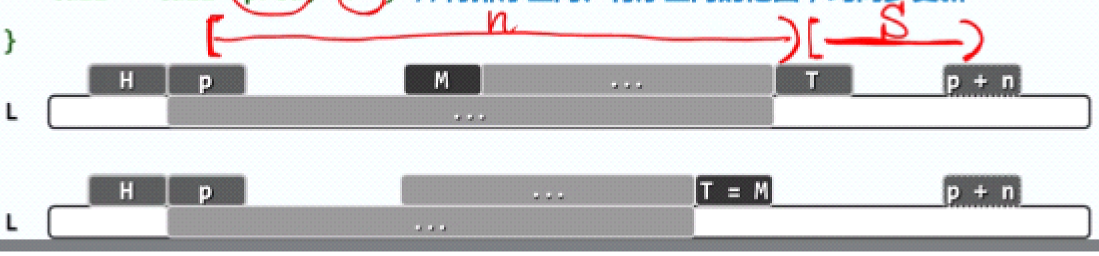
U ≤ S

迭代轮次	前缀无序子序列	后缀有序子序列
0	5 2 $\boxed{7}$ 4 6 3 1	^
1	5 2 4 $\boxed{6}$ 3 1	7
2	$\boxed{5}$ 2 4 3 1	6 7
3	2 $\boxed{4}$ 3 1	5 6 7
4	2 $\boxed{3}$ 1	4 5 6 7
5	$\boxed{2}$ 1	3 4 5 6 7
6	$\boxed{1}$	2 3 4 5 6 7
7	^	1 2 3 4 5 6 7

5.4.3 实现

实现：selectionSort()

```
//对列表中起始于位置p的连续n个元素做选择排序，valid(p) && rank(p) + n <= size
template <typename T> void List<T>::selectionSort(Posi(T) p, int n) {
   Posi(T) head = p->pred; Posi(T) tail = p; //待排序区间(head, tail)
   for (int i = 0; i < n; i++) tail = tail->succ; //head/tail可能是头/尾哨兵
   while (1 < n) { //反复从（非平凡的）待排序区间内找出最大者，并移至有序区间前端
      insertBefore( tail, remove( selectMax( head->succ, n ) ) );
      tail = tail->pred; n--; //待排序区间、有序区间的范围，均同步更新
   }
}
```



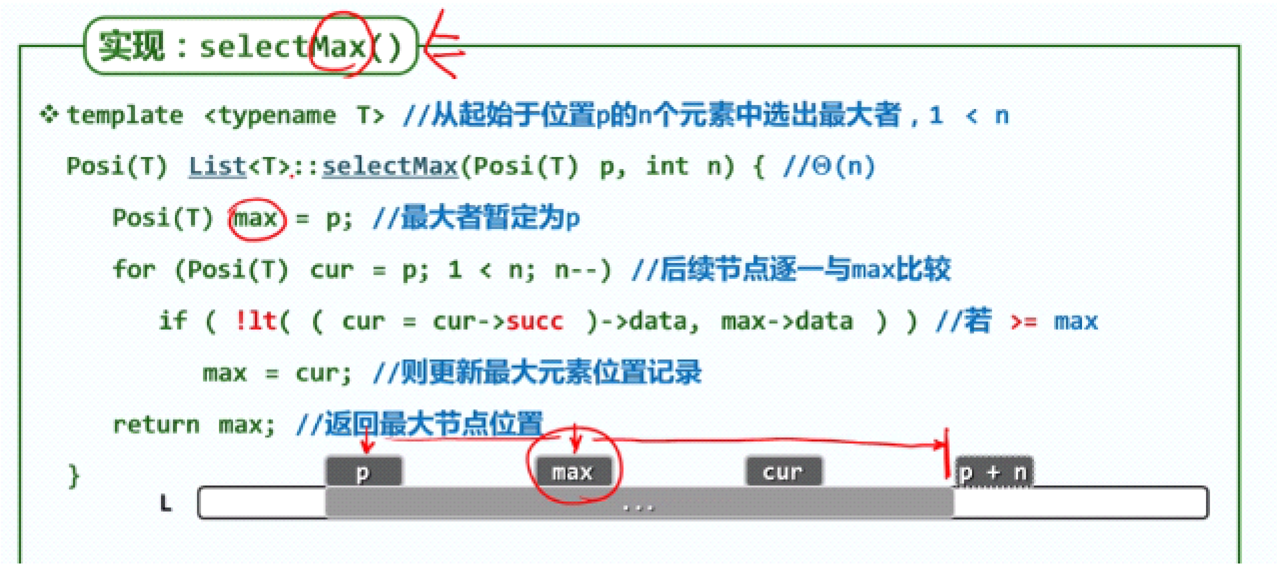
5.4.4 推敲

上面的实现并不完美：

1. 我们调用的insertbefore()函数和remove()函数中都有new与delete这两种操作，而这两种操作是其他基本操作复杂度的100倍，所以应该避开

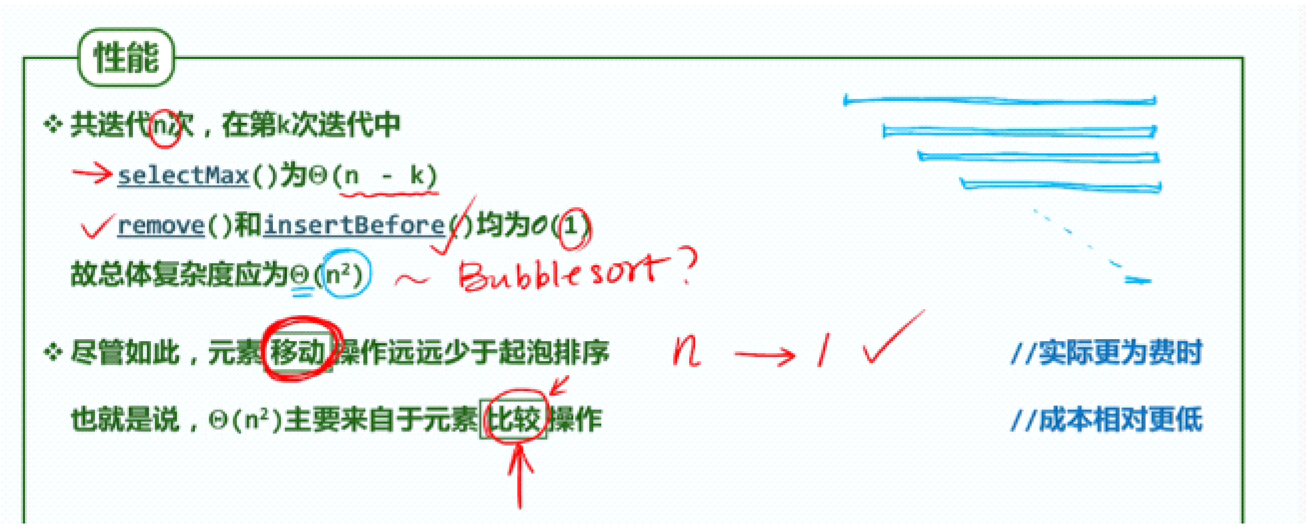
2. 存在当最大元已经是有序序列最后元素的的前驱，不必进行操作，但是这种情况发生概率极低，考虑添加if语句时，会添加比较操作，得不偿失。

5.4.5 selectMax()



Lt() not less than 画家算法（所得元为最后取得的元素）

5.4.5 性能



在这个优化中，元素的移动操作得到了很大优化：复杂度由n将至1；但是就比较操作而言没有得到优化。（参见第十章）

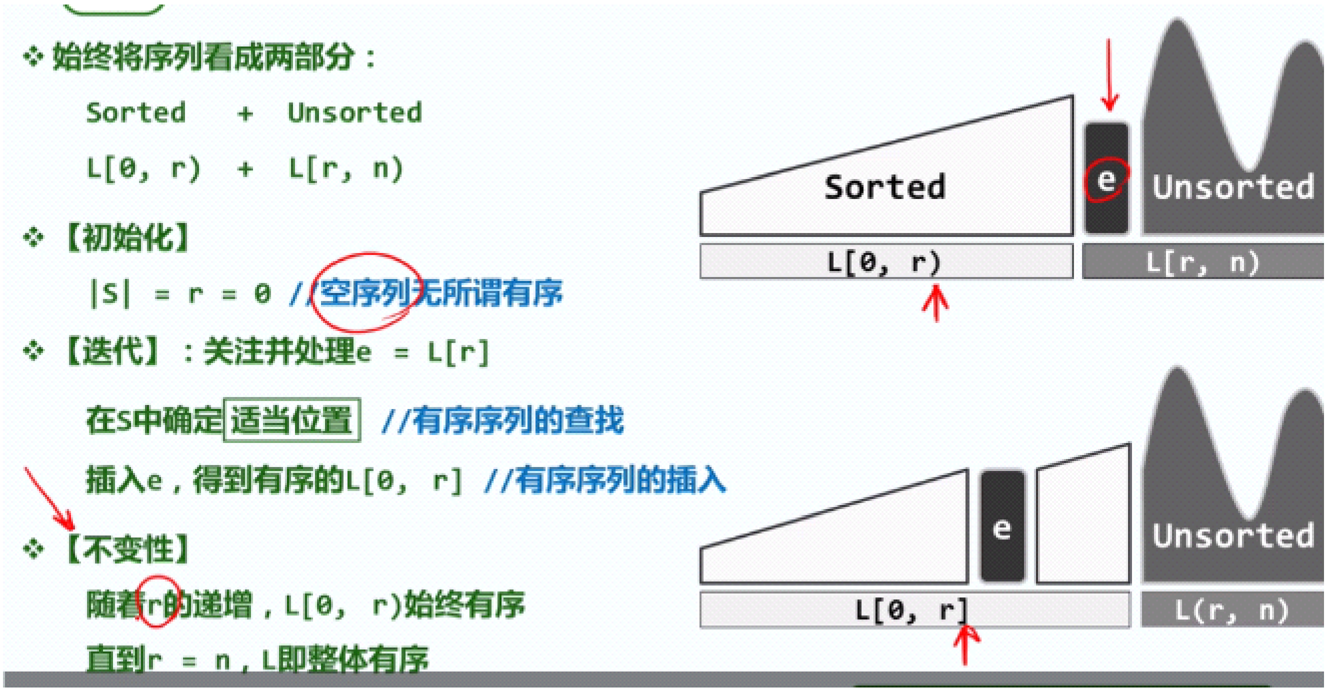
第五节 插入排序

2016年8月27日　　17:33

5.5.1 经验

Step1.定位
Step2.移动

5.5.2 构思



5.5.3 对比

插入排序与选择排序区别

5.5.4 实例

实例

迭代轮次	前缀有序子序列	当前元素	后缀无序子序列
-1	^	^	5 2 7 4 6 3 1
0	^	5	2 7 4 6 3 1
1	(5)	2	7 4 6 3 1
2	(2) 5	7	4 6 3 1
3	2 5 (7)	4	6 3 1
4	2 (4) 5 7	6	3 1
5	2 4 5 (6) 7	3	1
6	2 (3) 4 5 6 7	1	^

5	2 4 5 (6) 7	3	1
6	2 (3) 4 5 6 7	1	^
7	(1) 2 3 4 5 6 7	^	^

5.5.5 实现

实现

```
//对列表中起始于位置p的连续n个元素做插入排序，valid(p) && rank(p) + n <= size
  template <typename T> void List<T>::insertionSort(Posi(T) p, int n) {
     for (int r = 0; r < n; r++) { //逐一引入各节点，由Sr得到Sr+1
        insertAfter( search( p->data, r, p ), p->data ); //查找 + 插入
        p = p->succ; remove( p->pred ); //转向下一节点
     } //n次迭代，每次O(r + 1)
  } //仅使用O(1)辅助空间，属于就地算法
```

5.5.6 性能分析

最好情况是先前所有元素都已经有序，所以只需查找，复杂度为O(n)；
最坏情况是，所有元素都为逆序，所以复杂度为算术级数，为$O(n^2)$.

这里的查找是亦步亦趋进行的，必须这样。由于列表的结构导致查找时无法采用二分查找。

5.5.7 平均性能

假定：各元素取值遵守均匀、独立分布

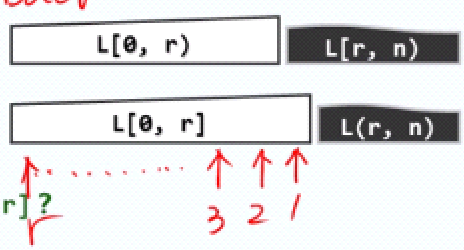
平均性能 backward Analysis

❖ 假定：各元素的取值遵守均匀、独立分布 n ? n^2

于是：平均要做多少次元素比较？

❖ 考查：L[r]刚插入完成的那一时刻（穿越？）

试问：此时的有序前缀L[0, r]中，哪个元素是此前的L[r]？



❖ 观察：其中的r + 1个元素$\boxed{\text{均有可能}}$，且概率$\boxed{\text{均等于}}$1/(r + 1)

❖ 因此，在刚完成的这次迭代中，为引入S[r]所花费时间的数学期望为

[r + (r - 1) + ... + 3 + 2 + 1 + 0] / (r + 1) + 1 = r/2 + 1

❖ 于是，总体时间的数学期望 = [0 + 1 + ... + (n - 1)] / 2 + 1 = $O(n^2)$

数学期望的线性率：一组随机变量总和的期望，等于他们各自期望的总和。（无论他们相关与否）

5.5.8 逆序对（序列中任意两元素）

序列中任意两元素，若构成逆序对，则将后一元素逆序对数目+1
所有元素逆序对数目总和记为I

我们发现，I即为插入排序中，所有元素所需要比较的次数。这样插入排序的复杂度可表示为：逆序对总数 + 插入次数（固定为n）
最好情况为：逆序对总数为0，插入次数为n
最坏情况为：逆序对总数为C_n^2，插入次数仍为n.