

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

从静态到动态

◇ 根据是否修改数据结构，所有操作大致分为两类方式

- 1) 静态：仅读取，数据结构的内容及组成一般不变：get, search $\log n$
- 2) 动态：需写入，数据结构的局部或整体将改变：insert, remove $O(n)$

◇ 与操作方式相对应地，数据元素的存储与组织方式也分为两种

- 1) 静态：数据空间整体创建或销毁
数据元素的物理存储次序与其逻辑次序严格一致
可支持高效的静态操作
比如向量，元素的物理地址与其逻辑次序线性对应
- 2) 动态：为各数据元素动态地分配和回收的物理空间
逻辑上相邻的元素记录彼此的物理地址，在逻辑上形成一个整体
可支持高效的动态操作

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

列表节点：ADT接口

◇ 作为列表的基本元素，列表节点首先需要独立地“封装”实现
为此，可设置并约定若干基本的操作接口

操作	功能
pred()	当前节点前驱节点的位置
succ()	当前节点后继节点的位置
data()	当前节点所存数据对象
insertAsPred(e)	插入前驱节点，存入被引用对象e，返回新节点位置
insertAsSucc(e)	插入后继节点，存入被引用对象e，返回新节点位置

列表模板类：

列表：List模板类

```
#include "listNode.h" //引入列表节点类

template <typename T> class List { //列表模板类
private:
    Posi(T) header; Posi(T) trailer; //头、尾哨兵
protected: /* ... 内部函数 */
public: /* ... 构造函数、析构函数、只读接口、可写接口、遍历接口 */
};
```

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。

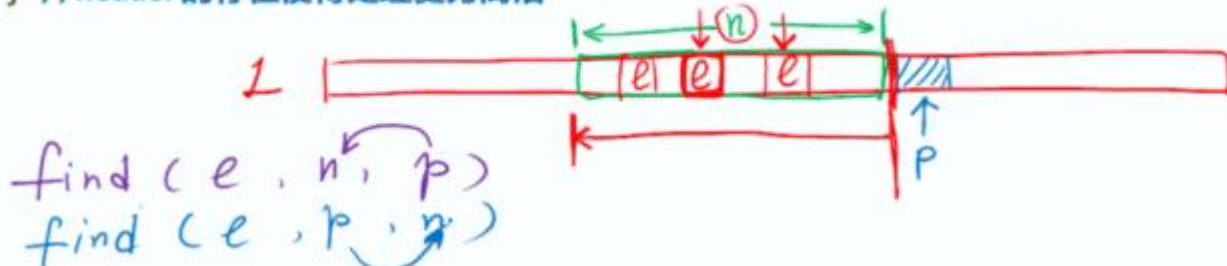
```
template <typename T> //assert: 0 <= r < size
T List<T>::operator[](Rank r) const { //O(r), 效率低下, 可偶尔为之, 却不宜常用
    Posi(T) p = first(); //从首节点出发
    while (0 < r--) p = p->succ; //顺数第r个节点即是
    return p->data; //目标节点
} //任一节点的秩, 亦即其前驱的总数
```

效率低下，复杂度为 $O(n)$

5.2.2 查找

❖ 在节点p (可能是trailer) 的n个 (真) 前驱中, 找到等于e的最后者

```
template <typename T> //从外部调用时, 0 <= n <= rank(p) < _size
Posi(T) List<T>::find(T const & e, int n, Posi(T) p) const { //顺序查找
    while (0 < n--) //从右向左, 逐个将p的前驱与e比对
        if (e == (p = p->pred)->data) return p; //直至命中或范围越界
    return NULL; //若越出左边界, 意味着查找失败
} //header的存在使得处理更为简洁
```



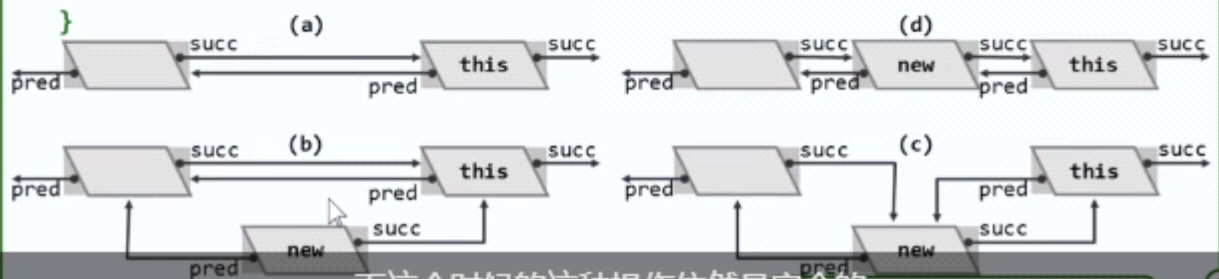
5.2.3 插入与复制

插入:

插入

```
❖ template <typename T> Posi(T) List<T>::insertBefore(Posi(T) p, T const& e)
{ _size++; return p->insertAsPred(e); } //e当作p的前驱插入
```

```
❖ template <typename T> //前插入算法（后插入算法完全对称）
Posi(T) List<T>::insertAsPred(T const& e) { //O(1)
    Posi(T) x = new List<T>::ListNode(e, pred, this); //创建（耗时100倍）
    pred->succ = x; pred = x; return x; //建立链接，返回新节点的位置
}
```



基于复制的构造

```
❖ template <typename T> //基本接口
void List<T>::copyNodes(Posi(T) p, int n) { //O(n)
    init(); //创建头、尾哨兵节点并做初始化
    while (n--) //将起自p的n项依次作为末节点插入
    { insertAsLast(p->data); p = p->succ; }
}
```

图中的`insertAsLast()`实质上就是`insertAsBefore(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)`

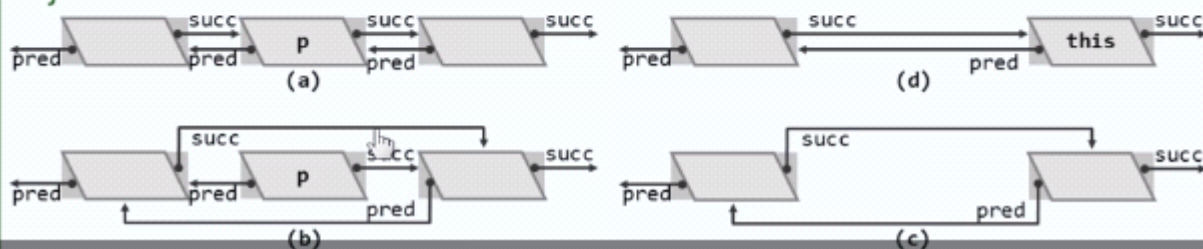
`T e = p->data; //备份待删除节点数值（设类型T可直接赋值）`

`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`

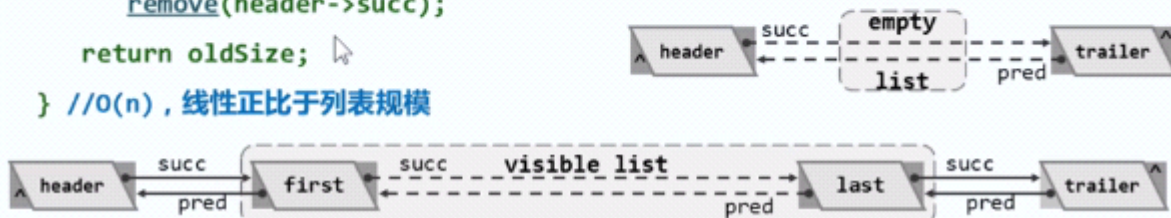
`int oldSize = _size;`

`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

唯一化

```

❖ template <typename T> int List<T>::deduplicate() { //剔除无序列表中的重复节点
    ✓ if ( _size < 2) return 0; //平凡列表自然无重复
    int oldSize = _size; //记录原规模
    Posi(T) p = first(); Rank r = 1; //p从首节点起
    while ( trailer != ( p = p->succ ) ) { //依次直到末节点
        Posi(T) q = find(p->data, r, p); //在p的r个(真)前驱中, 查找与之雷同者
        q ? remove(q) : r++; //若的确存在, 则删除之; 否则秩递增——可否remove(p)?
    } //assert: 循环过程中的任意时刻, p的所有前驱互不相同
    return oldSize - _size; //列表规模变化量, 即被删除元素总数
} //正确性及效率分析的方法与结论, 与Vector::deduplicate()相同

```



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, $p = p \rightarrow \text{succ}$ 会出现错误 (此时p已经不存在)

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 7 4 6 3 1	^			
1	5 2 4 6 3 1	7			
2	5 2 4 3 1	6 7			
3	2 4 3 1	5 6 7			
4	2 3 1	4 5 6 7			
5	2 1	3 4 5 6 7			
6	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
    }
}
```

Diagram illustrating the selection sort process on a linked list structure:

Initial state: List L contains nodes H, p, M, ..., T, p+n. The range [p, p+n) is marked as the initial sorting range (U).

After one iteration: The maximum element M is moved to the end of the list, becoming T. The new sorting range is [p, T), and the sorted range is [T, p+n). The range [p, T) is marked as the new sorting range (U), and the range [T, p+n) is marked as the sorted range (S).

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()

实现：selectMax()

```
template <typename T> //从起始于位置p的n个元素中选出最大者, 1 < n
Posi(T) List<T>::selectMax(Posi(T) p, int n) { //Θ(n)
    Posi(T) max = p; //最大者暂定为p
    for (Posi(T) cur = p; 1 < n; n--) //后续节点逐一与max比较
        if ( !lt( ( cur = cur->succ )->data, max->data ) ) //若 >= max
            max = cur; //则更新最大元素位置记录
    return max; //返回最大节点位置
}
```

Lt() not less than 画家算法（所得元为最后取得的元素）

5.4.5 性能

性能

- ❖ 共迭代n次，在第k次迭代中
→ selectMax()为 $\Theta(n - k)$
✓ remove()和insertBefore()均为 $O(1)$
故总体复杂度应为 $\Theta(n^2)$ ~ Bubble sort?
- ❖ 尽管如此，元素移动操作远远少于起泡排序
也就是说， $\Theta(n^2)$ 主要来自于元素比较操作

$n \rightarrow 1$ ✓ //实际更为费时
//成本相对更低

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

始终将序列看成两部分：

Sorted + Unsorted
 $L[0, r) + L[r, n)$

【初始化】
 $|S| = r = 0$ // 空序列无所谓有序

【迭代】：关注并处理 $e = L[r]$

在s中确定适当位置 // 有序序列的查找

插入e，得到有序的 $L[0, r]$ // 有序序列的插入

【不变性】
随着r的递增， $L[0, r)$ 始终有序
直到 $r = n$ ，L即整体有序

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

❖ 假定: 各元素的取值遵守均匀、独立分布。
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 $r + 1$ 个元素均有可能, 且概率均等于 $1/(r + 1)$

❖ 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

$$[r + (r - 1) + \dots + 3 + 2 + 1 + 0] / (r + 1) + 1 = r/2 + 1$$

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

backward Analysis

$L[0, r)$

$L[r, n)$

$L[0, r]$

$L(r, n)$

↑ ↑ ↑
3 2 1

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.