

列表

循位置访问

Don't lose the link.

- Robin Milner

邓俊辉

deng@tsinghua.edu.cn

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

- 静态：仅读取，数据结构的内容及组成一般不变：get、search
- 动态：需写入，数据结构的局部或整体将改变：insert、remove

❖ 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

- 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

- 动态：为各数据元素动态地分配和回收的物理空间

相邻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

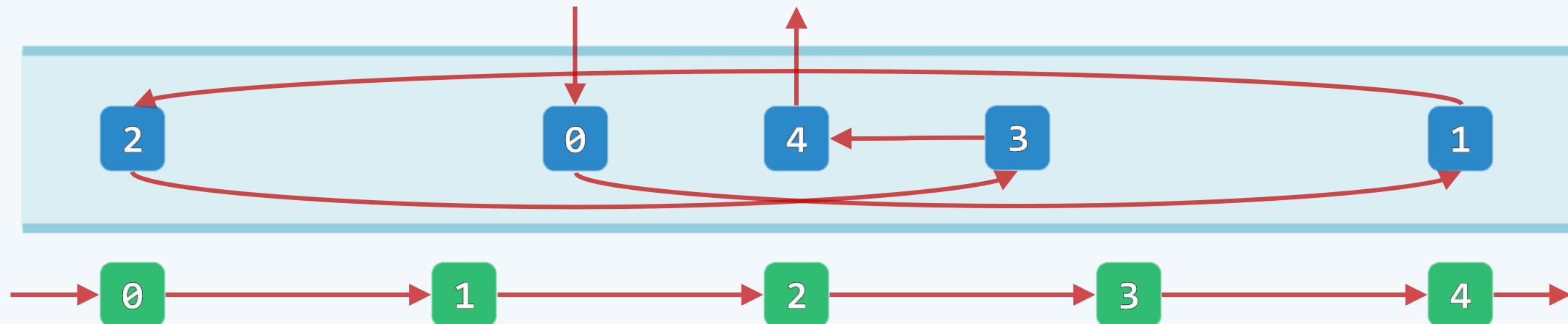
从向量到列表

❖ 列表 (list) 是采用动态储存策略的典型结构

- 其中的元素称作节点 (node) , 通过指针或引用彼此联接
- 在逻辑上构成一个线性序列 : $L = \{ a_0, a_1, \dots, a_{n-1} \}$

❖ 相邻节点彼此互称前驱 (predecessor) 或后继 (successor)

- 没有前驱/后继的节点称作首 (first/front) /末 (last/rear) 节点



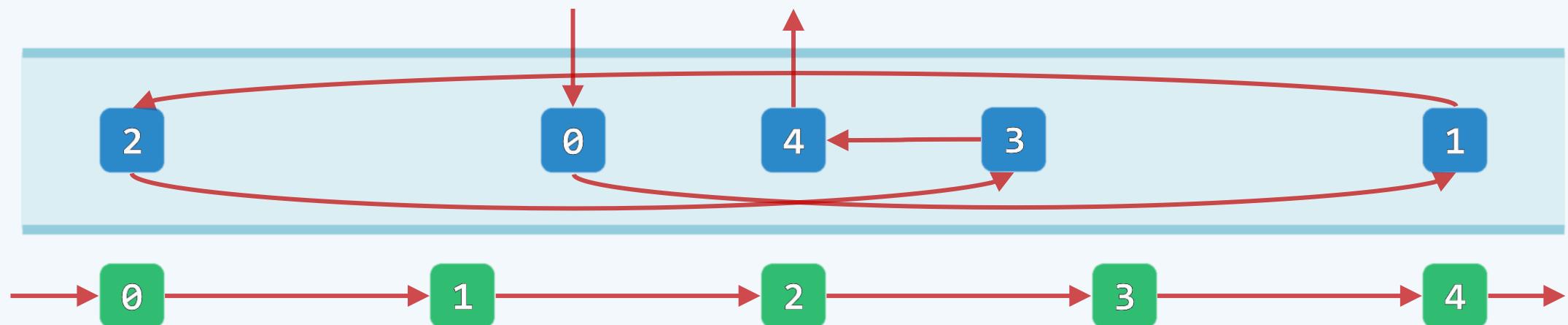
从秩到位置

❖ 向量支持循秩访问（call-by-rank）：根据元素的秩，可在 $\Theta(1)$ 时间内直接确定其物理地址



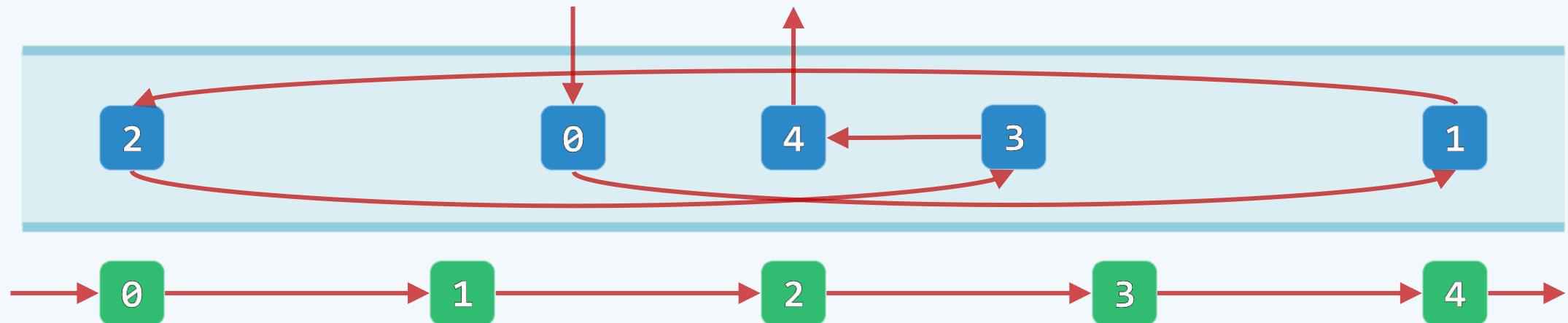
❖ 这种高效的方式，可否被列表沿用？

❖ 比如，从头/尾端出发，沿后继/前驱引用... //List::operator[](Rank r)，下节详解



从秩到位置

- ❖ 然而，此时的循秩访问成本过高，已不合时宜
- ❖ 因此，应改用循位置访问（call-by-position）的方式
亦即，转而利用节点之间的相互引用，找到特定的节点
- ❖ 比喻：找到我的朋友A的亲戚B的同事C的战友D的...的同学Z



列表

接口与实现

百只骆驼绕山走，九十八只在山后
尾驼露尾不见头，头驼露头出山沟

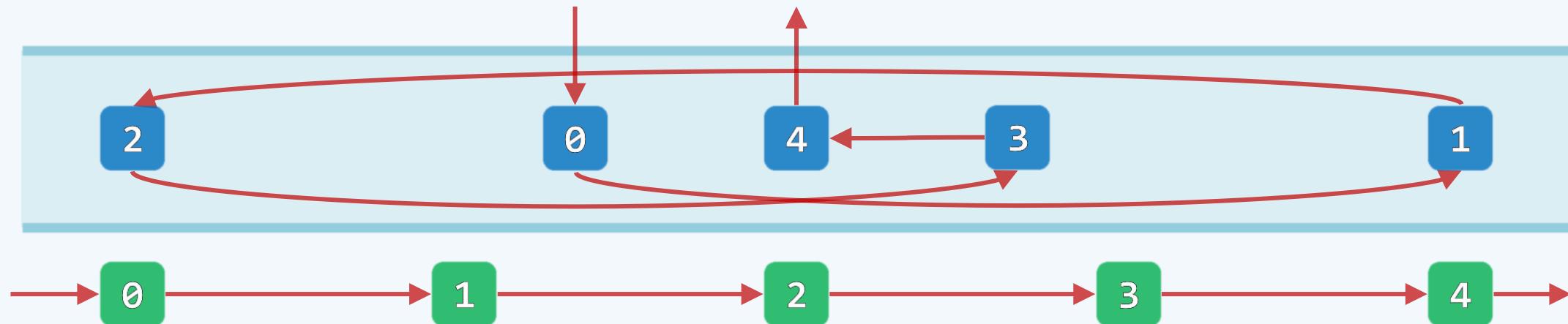
邓俊辉

deng@tsinghua.edu.cn

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

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



列表节点：模板类

- ❖ `#define Posi(T) ListNode<T>* //列表节点位置 (ISO C++.0x, template alias)`
- ❖ `template <typename T> //简洁起见，完全开放而不再严格封装`
- `struct ListNode { //列表节点模板类 (以双向链表形式实现)`
- `T data; //数值`
- `Posi(T) pred; //前驱`
- `Posi(T) succ; //后继`
- `ListNode() {} //针对header和trailer的构造`
- `ListNode(T e, Posi(T) p = NULL, Posi(T) s = NULL)`
- `: data(e), pred(p), succ(s) {} //默认构造器`
- `Posi(T) insertAsPred(T const& e); //前插入`
- `Posi(T) insertAsSucc(T const& e); //后插入`

`};`

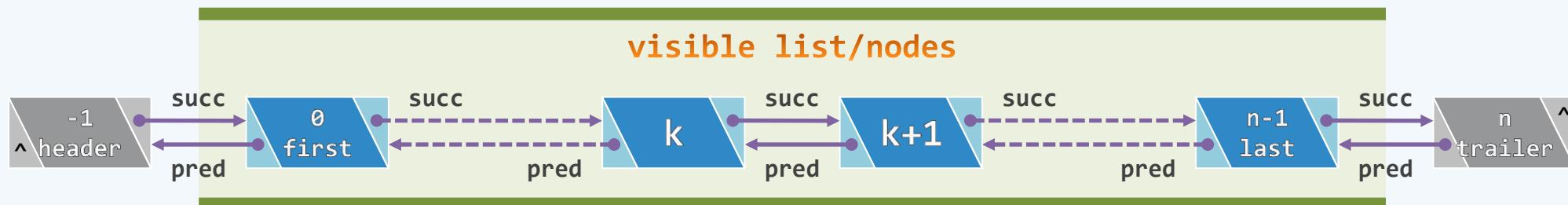


列表 : ADT接口

操作接口	功能	适用对象
<code>size()</code>	报告列表当前的规模 (节点总数)	列表
<code>first(), last()</code>	返回首、末节点的位置	列表
<code>insertAsFirst(e), insertAsLast(e)</code>	将e当作首、末节点插入	列表
<code>insertA(p, e), insertB(p, e)</code>	将e当作节点p的直接后继、前驱插入	列表
<code>remove(p)</code>	删除位置p处的节点，返回其引用	列表
<code>disordered()</code>	判断所有节点是否已按非降序排列	列表
<code>sort()</code>	调整各节点的位置，使之按非降序排列	列表
<code>find(e)</code>	查找目标元素e，失败时返回NULL	列表
<code>search(e)</code>	查找e，返回不大于e且秩最大的节点	有序列表
<code>deduplicate(), uniquify()</code>	剔除重复节点	列表/有序列表
<code>traverse()</code>	遍历列表	列表

列表：模板类

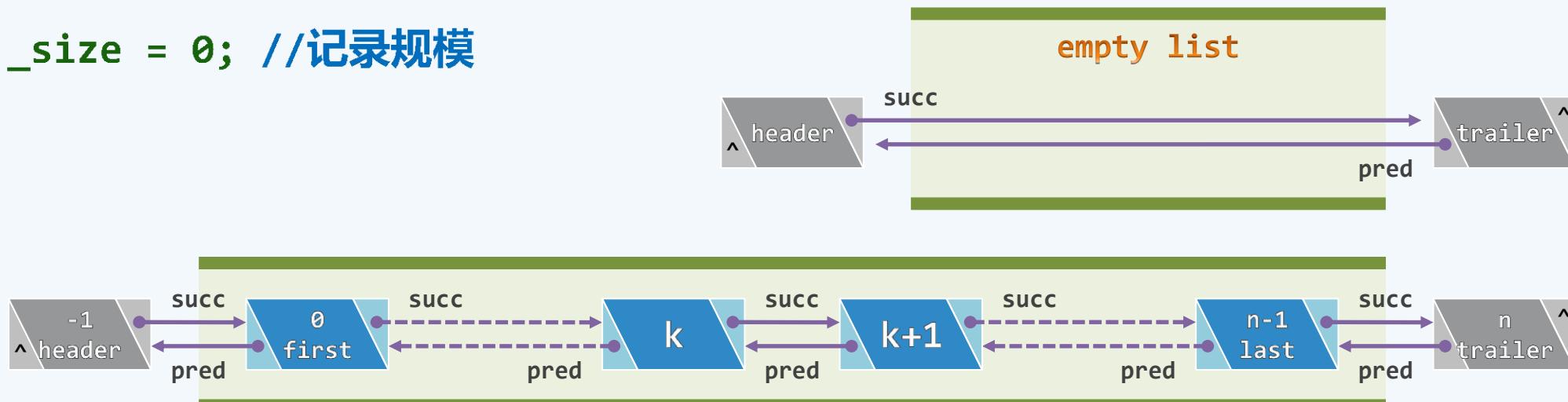
```
❖ #include "listNode.h" //引入列表节点类  
❖ template <typename T> class List { //列表模板类  
  
private:    int _size; //规模  
  
           Posi(T) header; Posi(T) trailer; //哨兵  
  
protected: /* ... 内部函数 */  
  
public:   /* ... 构造函数、析构函数、只读接口、可写接口、遍历接口 */  
};
```



❖ 等效地，头、首、末、尾节点的秩可分别理解为-1、0、n-1、n

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  
}
```



循秩访问

❖ 通过重载下标操作符，可模仿向量的循秩访问方式

❖ `template <typename T> //assert: 0 <= r < size`

`T List<T>::operator[](Rank r) const { // $\mathcal{O}(r)$ 效率低下，可偶尔为之，却不宜常用`

`Posi(T) p = first(); //从首节点出发`

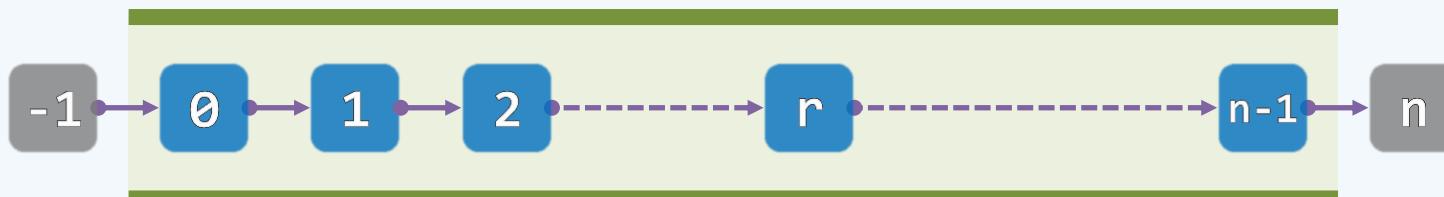
`while (0 < r--) p = p->succ; //顺数第r个节点即是`

`return p->data; //目标节点`

`} //秩 == 前驱的总数`

❖ 时间复杂度为 $\mathcal{O}(r)$

均匀分布时，期望复杂度为 $(1 + 2 + 3 + \dots + n)/n = \mathcal{O}(n)$



列表

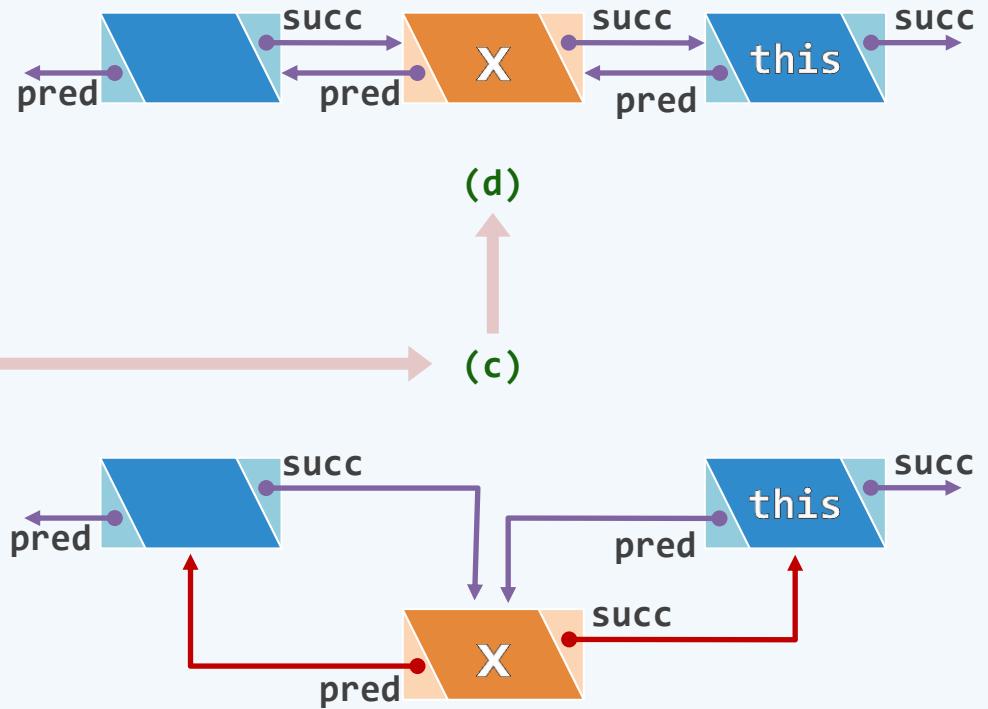
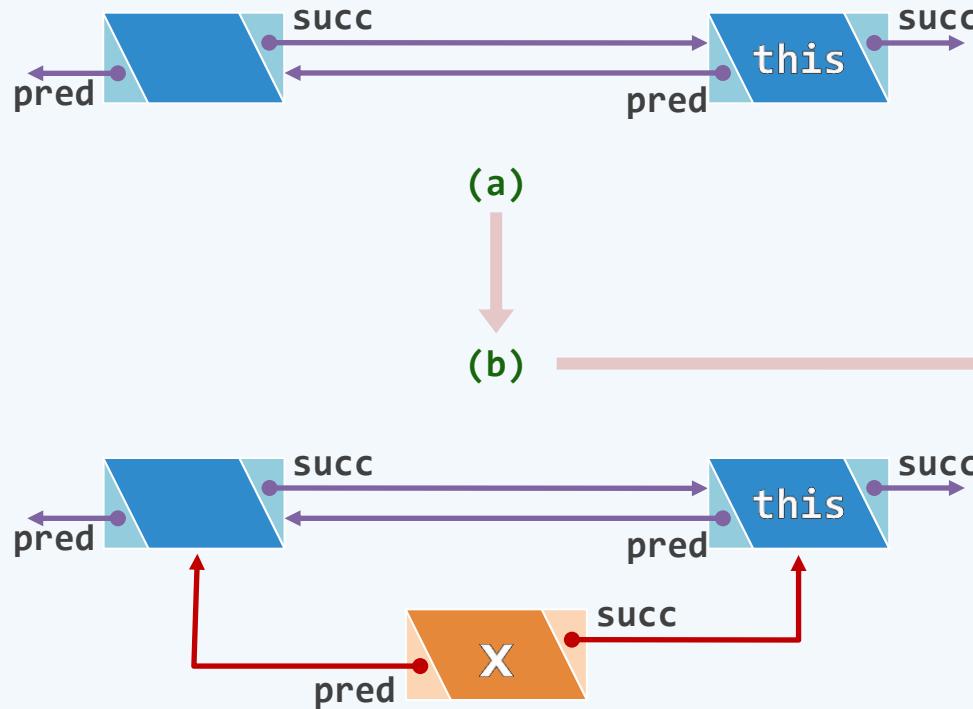
无序列表：插入与删除

邓俊辉

deng@tsinghua.edu.cn

插入 : 思路 + 过程

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



插入 : 实现

❖ template <typename T> //前插入算法 (后插入算法完全对称)

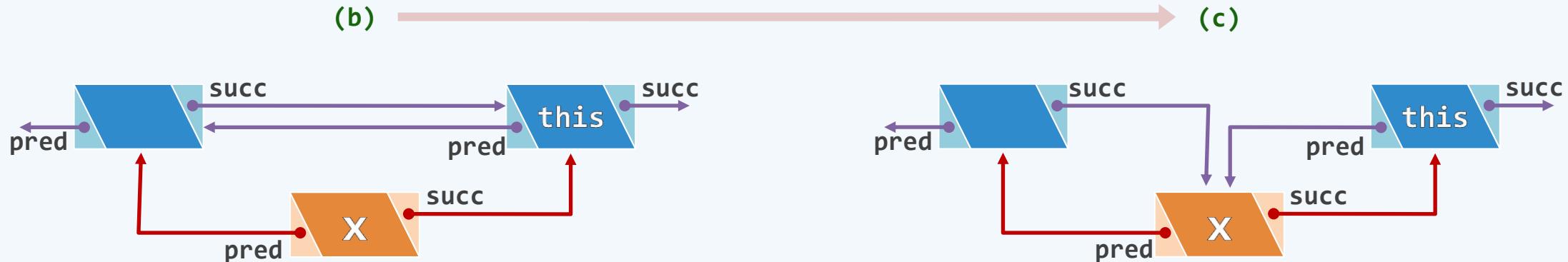
```
Posi(T) ListNode<T>::insertAsPred( T const & e ) { //O(1)
```

```
    Posi(T) x = new ListNode( e, pred, this ); //创建 (耗时100倍)
```

```
    pred->succ = x; pred = x; //次序不可颠倒
```

```
    return x; //建立链接，返回新节点的位置
```

} //得益于哨兵，即便this为首节点亦不必特殊处理——此时等效于insertAsFirst(e)



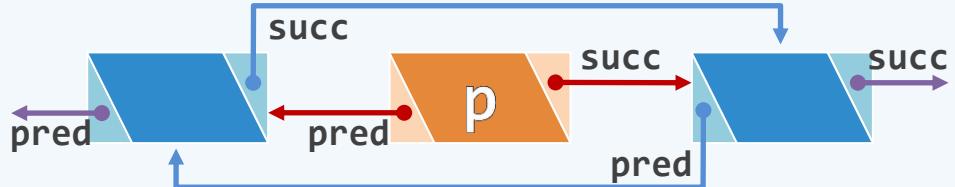
删除 : 思路 + 过程



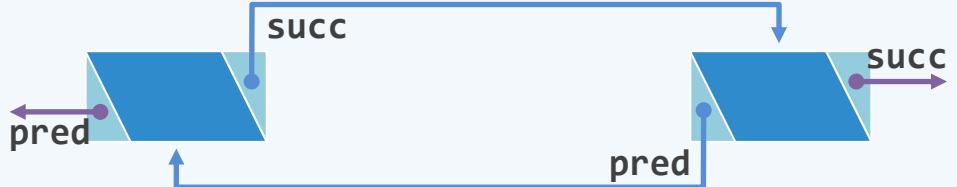
(a)



(d)



(b)

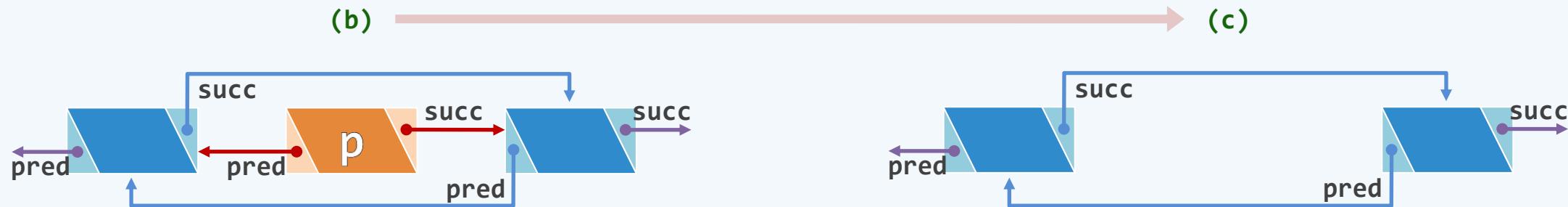


(c)

删除：实现

❖ 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  
}
```



列表

无序列表：构造与析构

A scientist discovers that which exists, an engineer
creates that which never was.

“宇宙里有生有死……爱情里也有死有生。”

“这是什么意思？”剑云低声说，没有人回答他。

邓俊辉

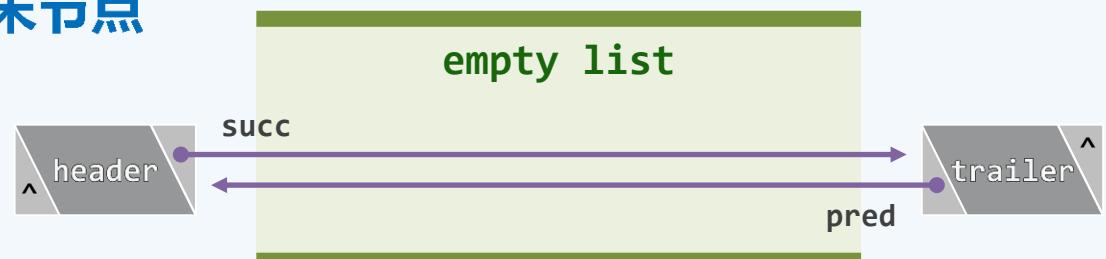
deng@tsinghua.edu.cn

❖ `template <typename T> 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

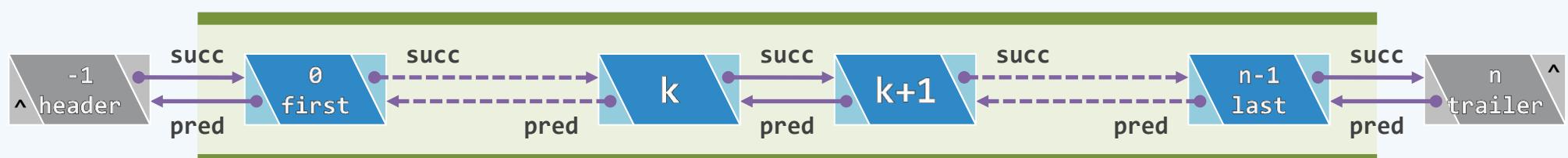
`while (n--) { //将起自p的n项依次作为末节点`

insertAsLast(p->data); //插入

`p = p->succ;`



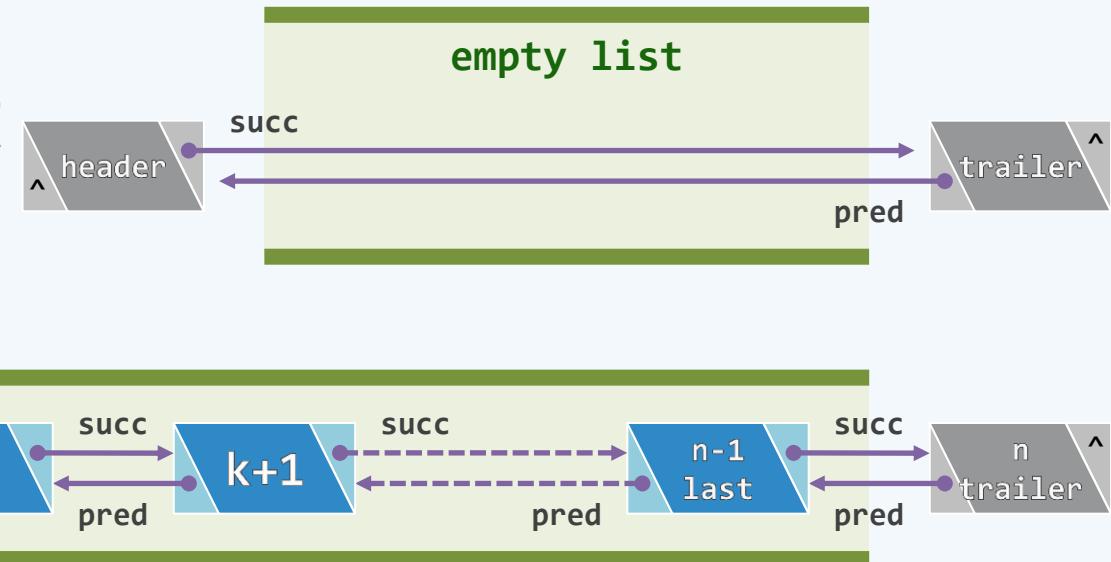
`}`



`}`

❖ `List<T>::List(List<T> const & L) { copyNodes(L.first(), L._size); }`

- ❖ `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，O(n)`
`int oldSize = _size;`
`while (0 < _size) //反复删除首节点`
`remove(header->succ);`
`return oldSize;`



- ❖ 若remove(header->succ)改作remove(trailer->pred)呢？

列表

无序列表：查找与去重

顧長康噉甘蔗，先食尾。問所以，云：“漸至佳境。”

有些事，你一辈子总也忘不掉。凡是让你揪心的事，在你身上，都会发生两次。或两次以上。

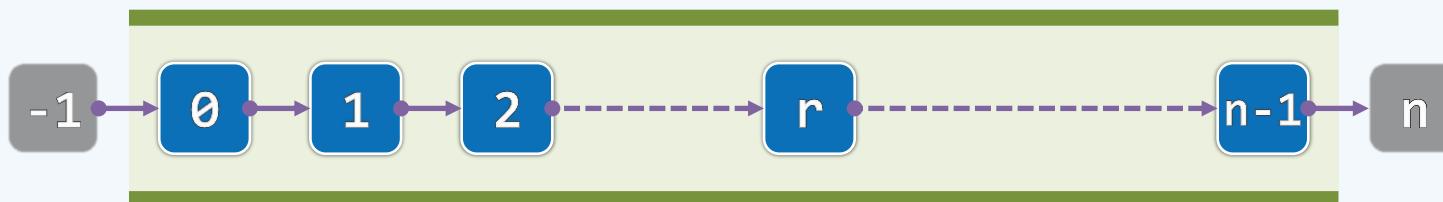
邓俊辉

deng@tsinghua.edu.cn

查找

```
❖ template <typename T> //0 ≤ n ≤ rank(p) < _size  
Posi(T) List<T>::find( T const & e, int n, Posi(T) p ) const { //O(n)  
    while ( 0 < n-- ) //自后向前逐个比对  
        if ( e == ( p = p->pred ) ->data ) //假定类型T已重载 “==”  
            return p; //在p的n个前驱中，等于e的最靠后者  
    return NULL; //失败
```

}



```
❖ Posi(T) find( T const & e ) const { return find( e, _size, trailer ); }
```

去重

```
❖ template <typename T> int List<T>::deduplicate() {  
    int oldSize = _size;  
  
    ListNodePosi(T) p = first(); ListNodePosi(T) q = NULL;  
  
    for ( Rank r = 0; p != trailer; p = p->succ, q = find( p->data, r, p ) )  
        q ? remove( q ): r++; //r为无重前缀的长度  
  
    return oldSize - _size; //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
```

列表

无序列表：遍历

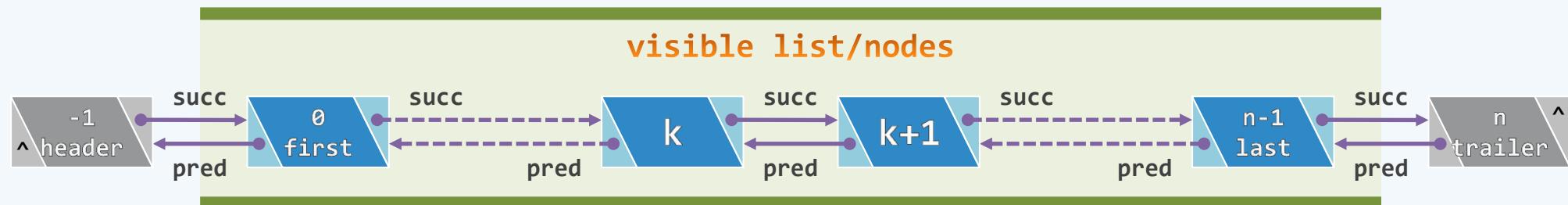
邓俊辉

deng@tsinghua.edu.cn

八戒道：“不要扯，等我一家家吃将来。”

❖ template <typename T>

```
void List<T>::traverse( void ( * visit )( T & ) ) //函数指针
{ Posi(T) p = header; while ( ( p = p->succ ) != trailer ) visit( p->data ); }
```



❖ template <typename T> template <typename VST>

```
void List<T>::traverse( VST & visit ) //函数对象
{ Posi(T) p = header; while ( ( p = p->succ ) != trailer ) visit( p->data ); }
```

列表

有序列表：唯一化

昨夜
我梦见
你和我
说同一个字

Most people are other people. Their thoughts are someone else's opinions, their lives a mimicry, their passions a quotation.

邓俊辉

deng@tsinghua.edu.cn

```
❖ template <typename T> int List<T>::uniquify() { //剔除重复元素  
    if ( _size < 2 ) return 0; //平凡列表自然无重复  
  
    int oldSize = _size; //记录原规模  
  
    ListNodePosi(T) p = first(); ListNodePosi(T) q; //各区段起点及其直接后继  
  
    while ( trailer != ( q = p->succ ) ) //反复考查紧邻的节点对(p,q)  
        if ( p->data != q->data ) p = q; //若互异，则转向下一对  
        else remove(q); //否则（雷同），删除后者  
  
    return oldSize - _size; //规模变化量，即被删除元素总数  
} //只需遍历整个列表一趟， $O(n)$ 
```

列表

有序列表：查找

种种念起，无不出于找。离了现前，向外去找，根本让我们直觉麻木的是这个找。

于是，他们急忙把自己的布袋卸在地上，各人打开自己的布袋。管家就搜查，从最大的开始，查到最小的。那杯竟在便雅悯的布袋里搜出来了

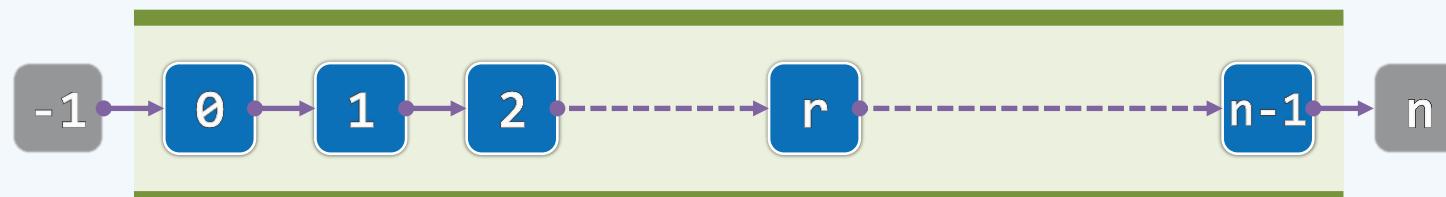
邓俊辉

deng@tsinghua.edu.cn

- ❖ 在有序列表内节点p的n个(真)前驱中，找到不大于e的**最靠后者**

- ❖ `template <typename T> // assert: 0 <= n <= rank(p) < _size`

```
Posi(T) List<T>::search( T const & e, int n, Posi(T) p ) const {  
    do { p = p->pred; n--; } //从右向左  
    while ( ( -1 < n ) && ( e < p->data ) ); //逐个比较，直至命中或越界  
    return p;  
}
```



- ❖ 失败时，返回区间左边界的前驱（可能是header）

- ❖ 最好 $O(1)$ ，最坏 $O(n)$ ；等概率时平均 $O(n)$ ，正比于区间宽度

❖ template <typename T>

```
Posi(T) List<T>::search(T const & e, int n, Posi(T) p) const ;
```

❖ 语义与向量相似，便于插入排序等后续操作：

```
insertA( search( e, r, p ), e )
```

❖ 为何未能借助有序性提高查找效率？实现不当，还是根本不可能？

❖ 按照循位置访问的方式，物理存储地址与其逻辑次序无关

依据秩的随机访问无法高效实现，而只能依据元素间的引用顺序访问

列表

选择排序

天下只有两种人。譬如一串葡萄到手，一种人挑最好的先吃，
另一种人把最好的留在最后吃。

邓俊辉

deng@tsinghua.edu.cn

当下又选了几样果菜与凤姐送去，凤姐儿也送了几样来。

起泡排序：温故知新

每趟扫描交换都需 $\Theta(n)$ 次比较、 $\Theta(n)$ 次交换；然而其中， $\Theta(n)$ 次交换完全没有必要

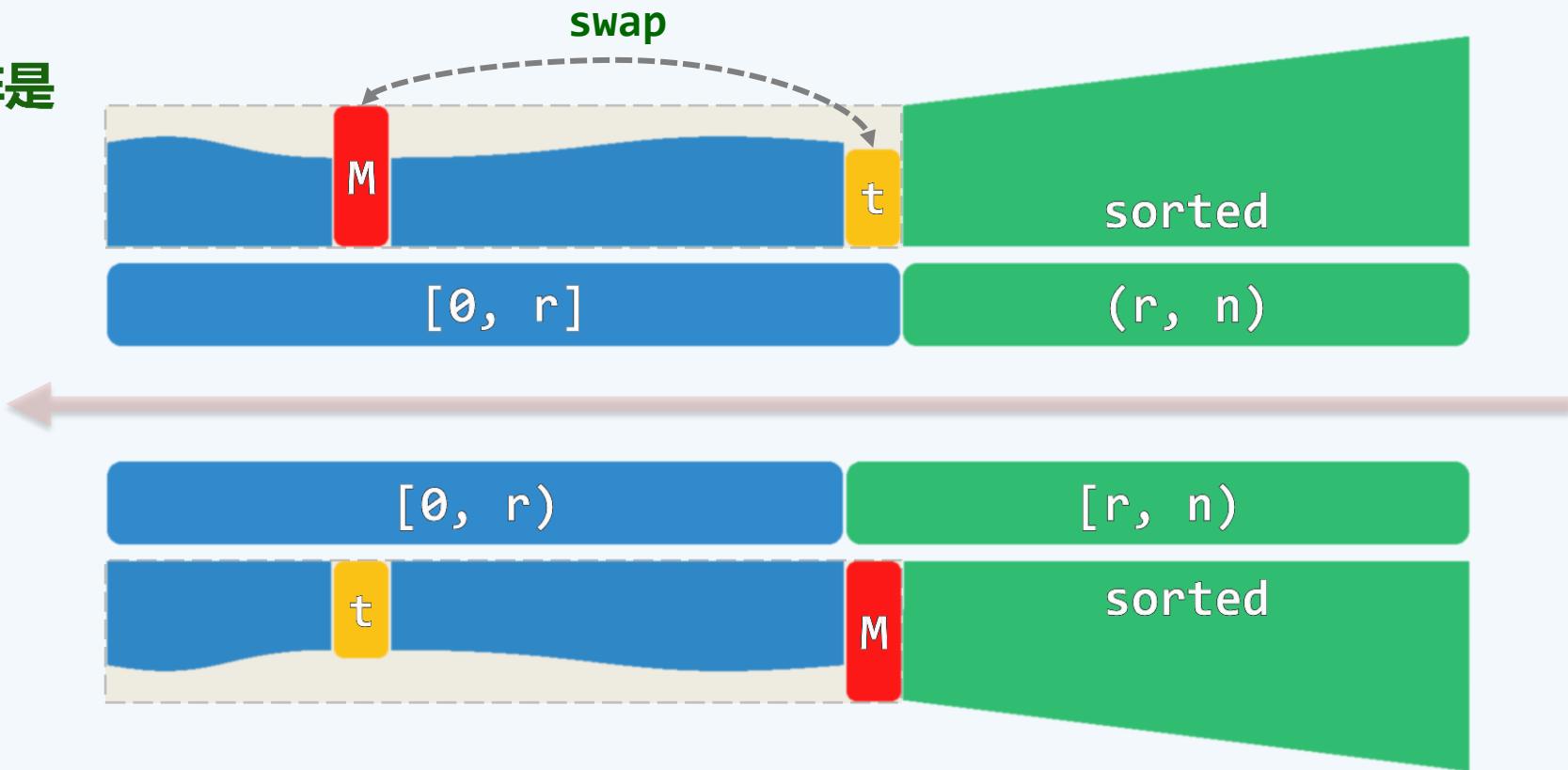
扫描交换的实质效果无非是

- 通过比较找到当前的

最大元素M，并

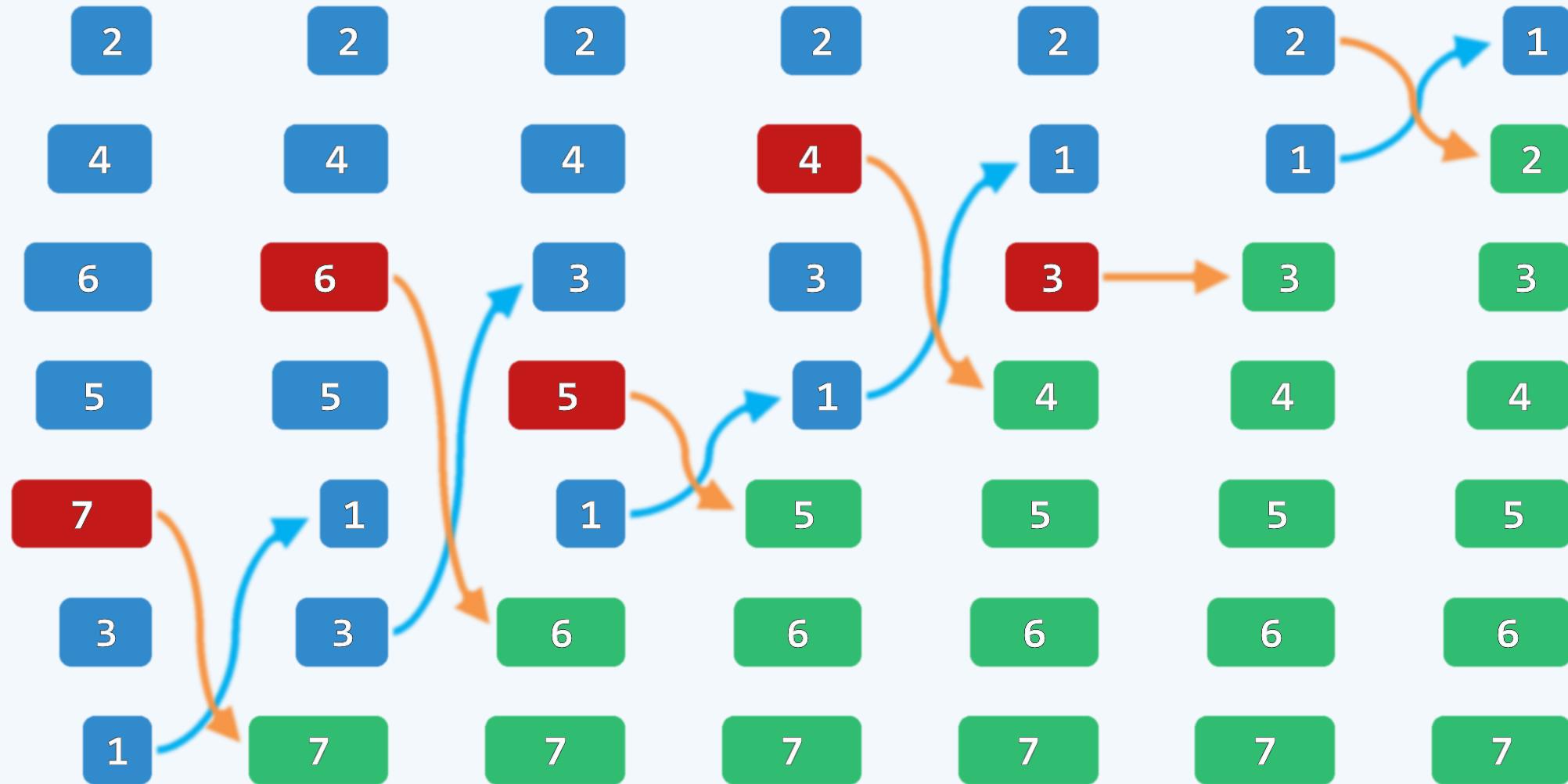
- 通过交换使之就位

如此看来

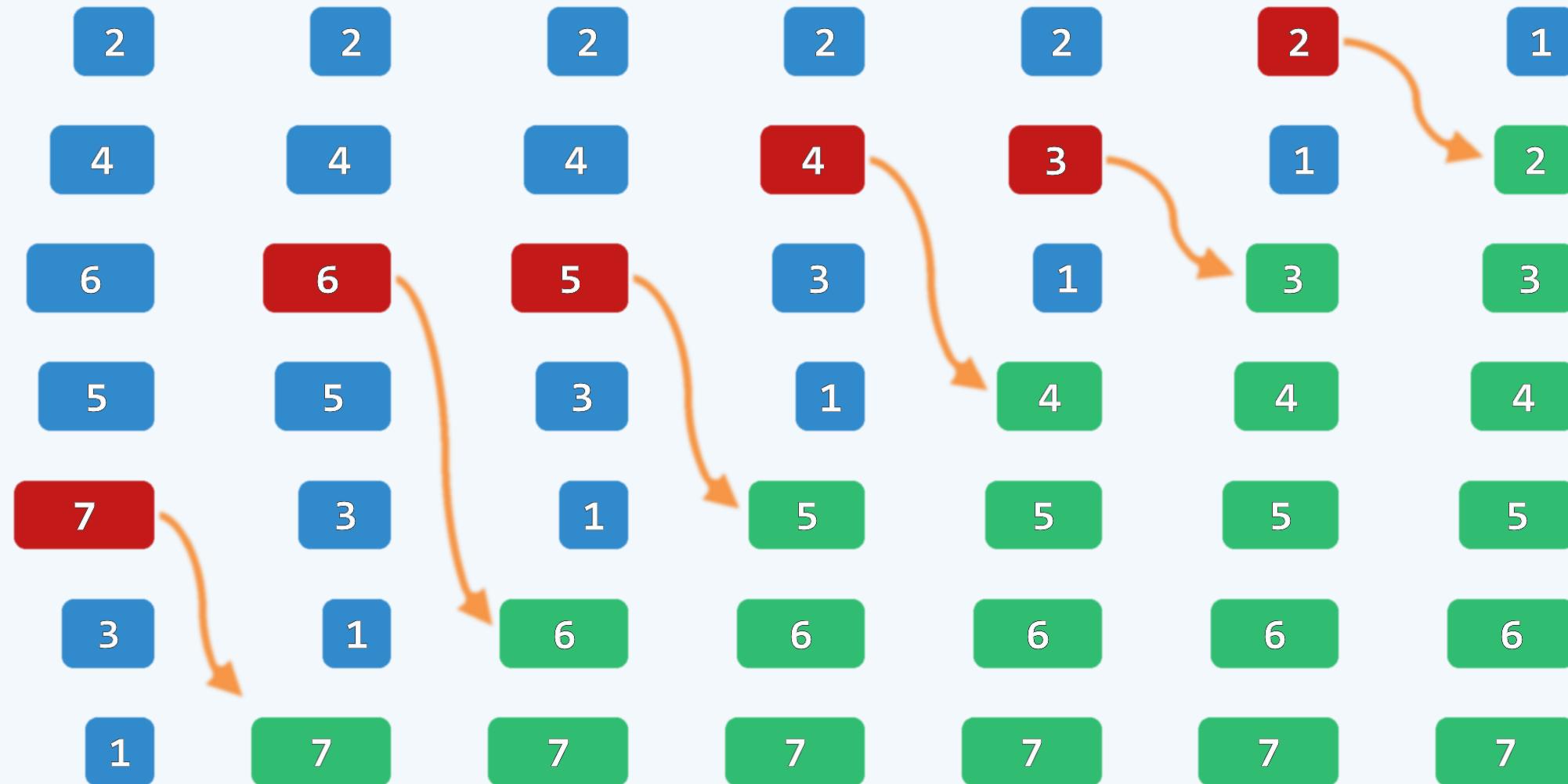


在经 $\Theta(n)$ 次比较确定M之后，仅需一次交换即足矣

交换法

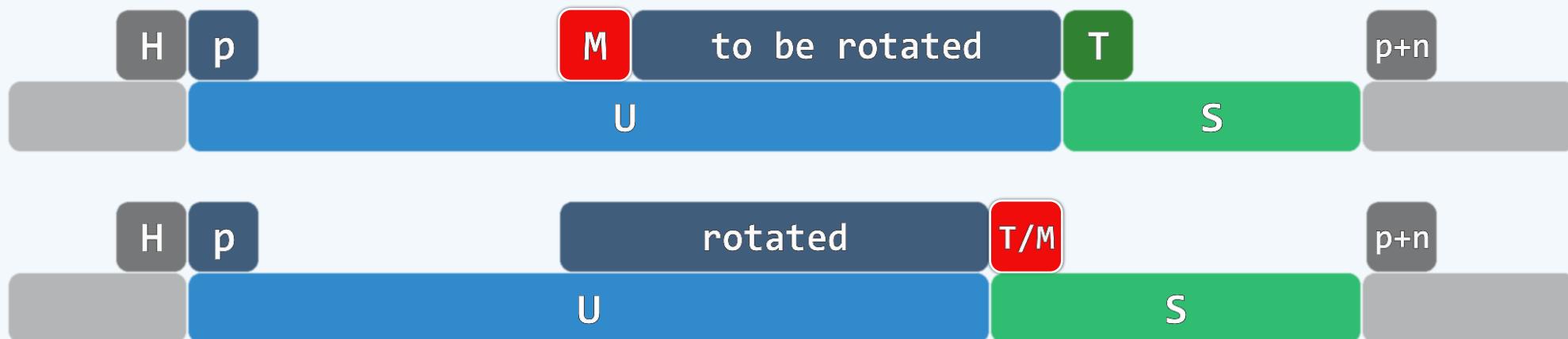


平移法



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）待排序区间内找出最大者，并移至有序区间前端
        insertB( tail, remove( selectMax( head->succ, n ) ) ); //改进...
        tail = tail->pred; n--; //待排序区间、有序区间的范围，均同步更新
    }
}
```



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 ) ) //data ≥ max  
            max = cur; //则更新最大元素位置记录  
  
    return max; //返回最大节点位置
```



稳定性

- ◆ 有多个重复元素同时命中时，往往需要按照某种**附加的约定**，返回其中**特定的某一个**
- ◆ 比如，通常都约定“**靠后者优先返回**”
- ◆ 为此，必须采用**比较器!lt()**或**ge()**，即等效于**后者优先**

2	6a	4	6b	3	0	<u>6c</u>	1	5	7	8	9
2	6a	4	6b	3	0	1	5	6c	7	8	9
2	6a	4	3	0	1	5	6b	6c	7	8	9
2	4	3	0	1	5	6a	6b	6c	7	8	9

- ◆ 如此即可保证，重复元素在列表中的相对次序，与其插入次序一致



性能分析

◆ 共迭代 n 次，在第 k 次迭代中

- selectMax() 为 $\Theta(n - k)$ //算术级数
- swap() 为 $\mathcal{O}(1)$ //或 remove() + insertB()

故总体复杂度应为 $\Theta(n^2)$

◆ 尽管如此，元素的移动操作远远少于起泡排序 //实际更为费时

也就是说， $\Theta(n^2)$ 主要来自于元素的比较操作 //成本相对更低

◆ 可否...每轮只做 $\mathcal{O}(n)$ 次比较，即找出当前的最大元素？

◆ 可以！...利用高级数据结构，selectMax()可改进至 $\mathcal{O}(n \log n)$ //稍后分解

当然，如此立即可以得到 $\mathcal{O}(n \log n)$ 的排序算法 //保持兴趣

列表

循环节

“.....莫非你吃了我吩咐你不可吃的那树上的果子吗？”

“你所赐给我、与我同居的女人，她把那树上的果子给我，我就吃了。”

“你作的是什么事呢？”

“那蛇引诱我，我就吃了。”

邓俊辉

deng@tsinghua.edu.cn

- ❖ 任何一个序列 $\mathcal{A}[0, n)$ ，都可以分解为若干个循环节 //设元素之间可定义次序
- ❖ 任何一个序列 $\mathcal{A}[0, n)$ ，都对应于一个有序序列 $S[0, n)$ //经排序之后
- ❖ 元素 $\mathcal{A}[k]$ 在 S 中对应的秩，记作 $r(\mathcal{A}[k]) = r(k) \in [0, n)$
- ❖ 元素 $\mathcal{A}[k]$ 所属的循环节是：
$$\mathcal{A}[k], \mathcal{A}[r(k)], \mathcal{A}[r(r(k))], \mathcal{A}[r(r(r(k)))], \dots, \mathcal{A}[r(\dots(r(r(k))\dots)] = \mathcal{A}[k]$$
- ❖ 每个循环节，长度均不超过 n
- ❖ 循环节之间，互不相交

实例

rank: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

$\mathcal{A}[]$: J N P M A I G O D C H B K L F E

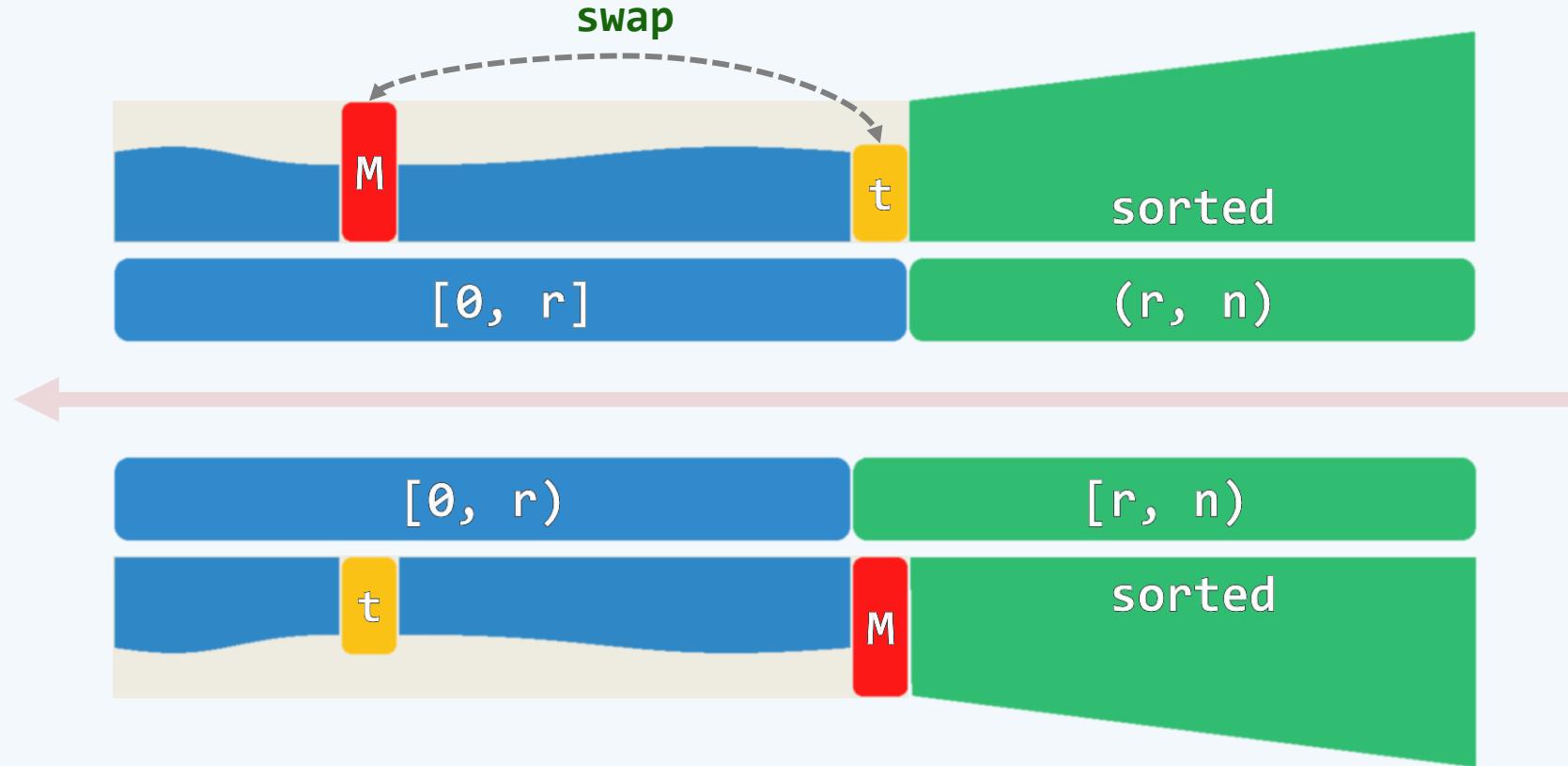
$S[]$: A B C D E F G H I J K L M N O P

$r[]$: 9 13 15 12 0 8 6 14 3 2 7 1 10 11 5 4

J	.	P	.	A	C	E
.	N	B	.	L
.	.	.	M	.	I	.	O	D	.	H	.	K	.	F	.
.	G

单调性

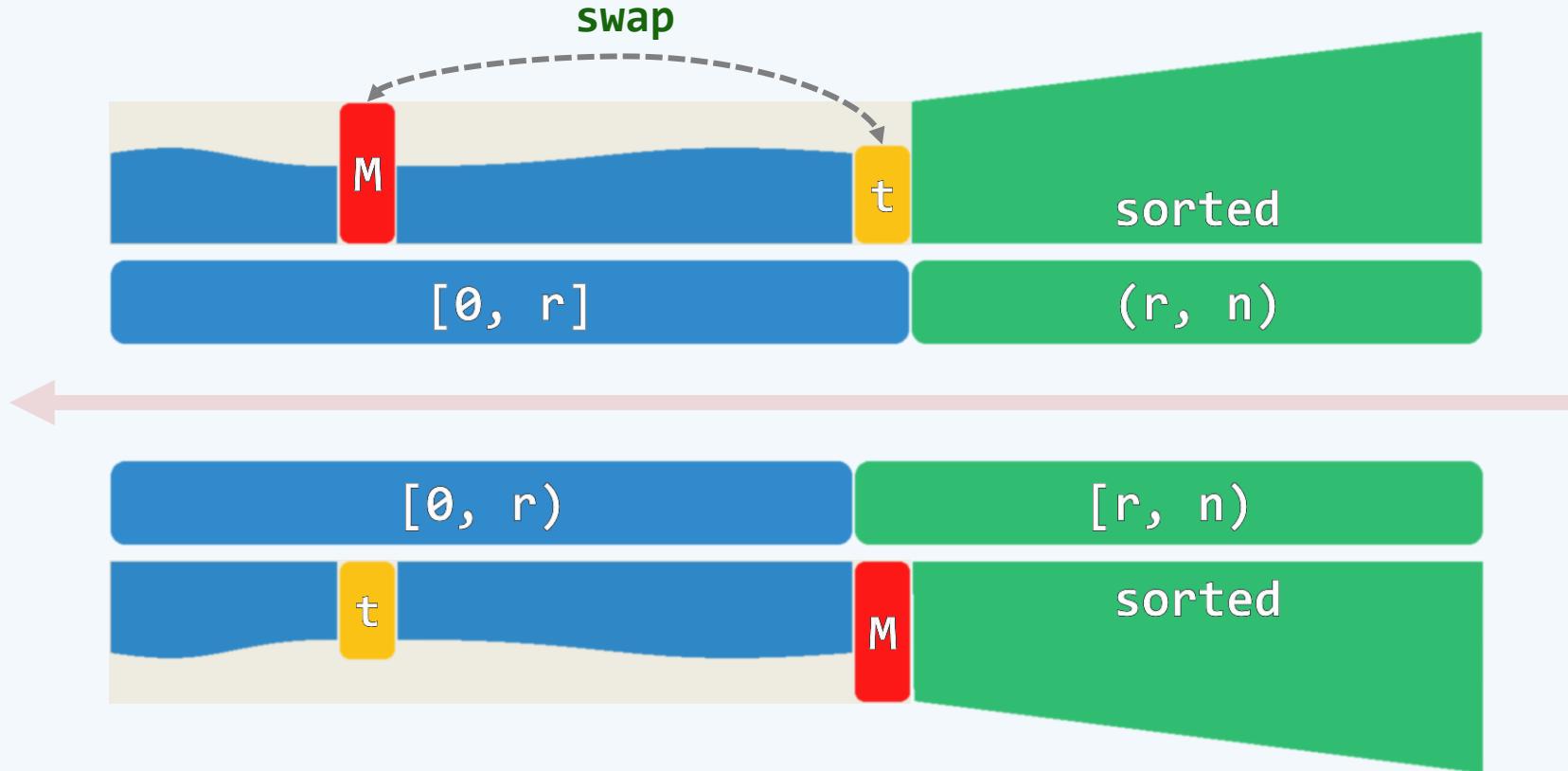
❖ 采用交换法，每迭代一步，M都会脱离原属的循环节，自成一个循环节



❖ M原所属循环节，长度恰好减少一个单位；其余循环节，保持不变

无效的交换

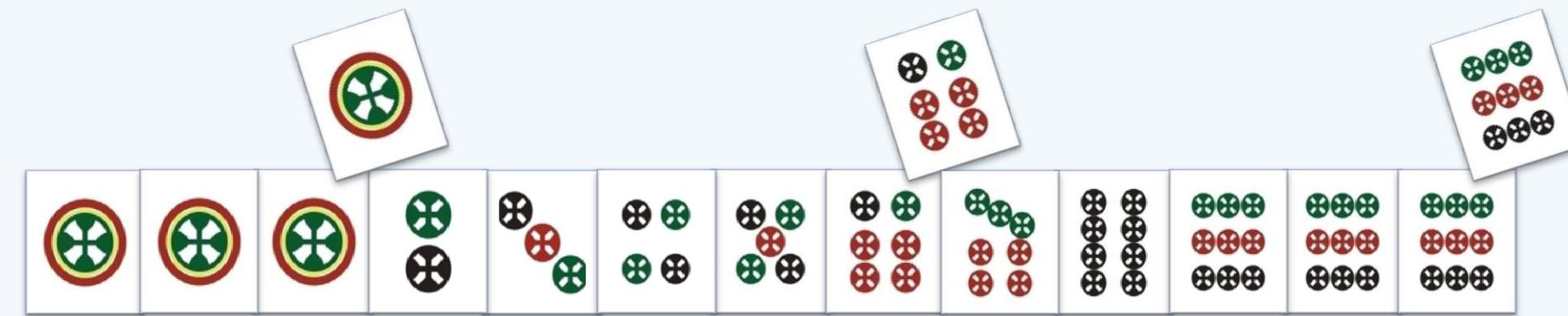
❖ M已经就位，无需交换 —— 这种情况会出现几次？



❖ 有 c 个循环节，就出现 $c-1$ 次 —— 最大值为 n ，期望 $\Theta(\log n)$

3. 列表

插入排序

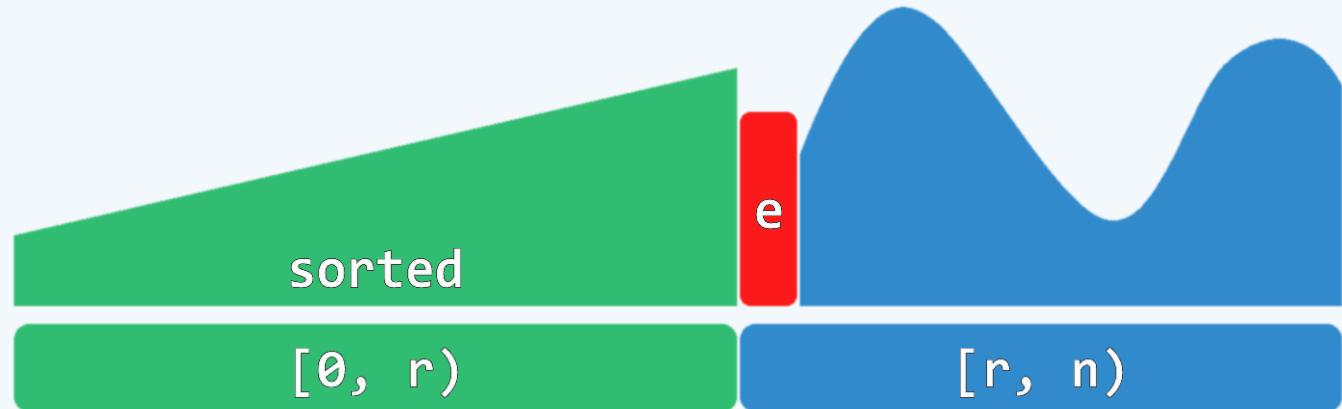


一语未了，只见宝玉笑嘻嘻的捐了一枝红梅进来，众丫鬟忙已接过，插入瓶内。

邓俊辉

deng@tsinghua.edu.cn

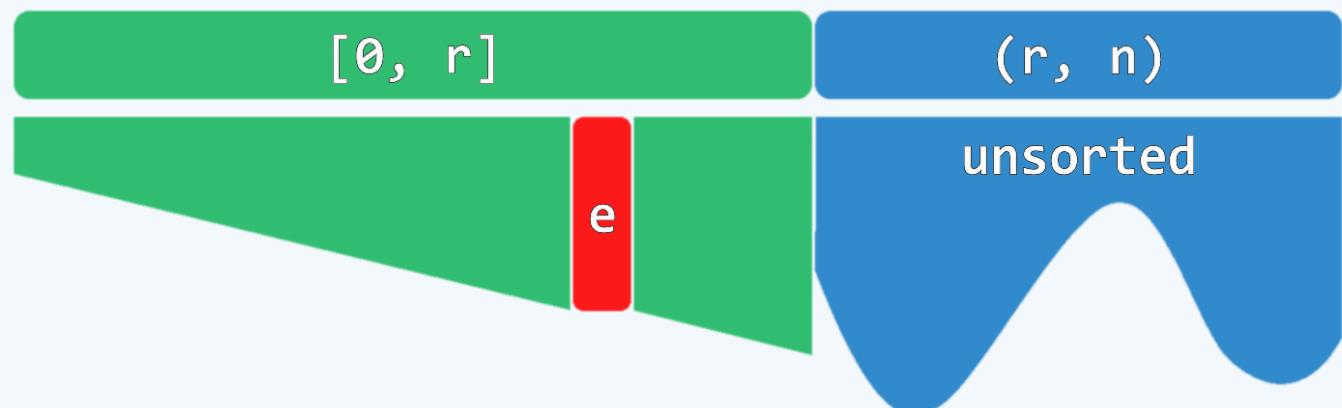
减而治之



❖ 不变性

序列总能视作两部分：

$$S[0, r) + U[r, n)$$



❖ 初始化

$$|S| = r = 0$$

❖ 反复地，针对 $e = A[r]$

在 S 中 **查找** 适当位置，以 **插入** e

❖ 二分查找？顺序查找？

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

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  
        insertA( 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
```

❖ 紧邻于search()接口返回的位置之后插入当前节点，总是保持有序

❖ 验证各种情况下的正确性，体会哨兵节点的作用：

s_r中含有/不含与p相等的元素；s_r中的元素均严格小于/大于p

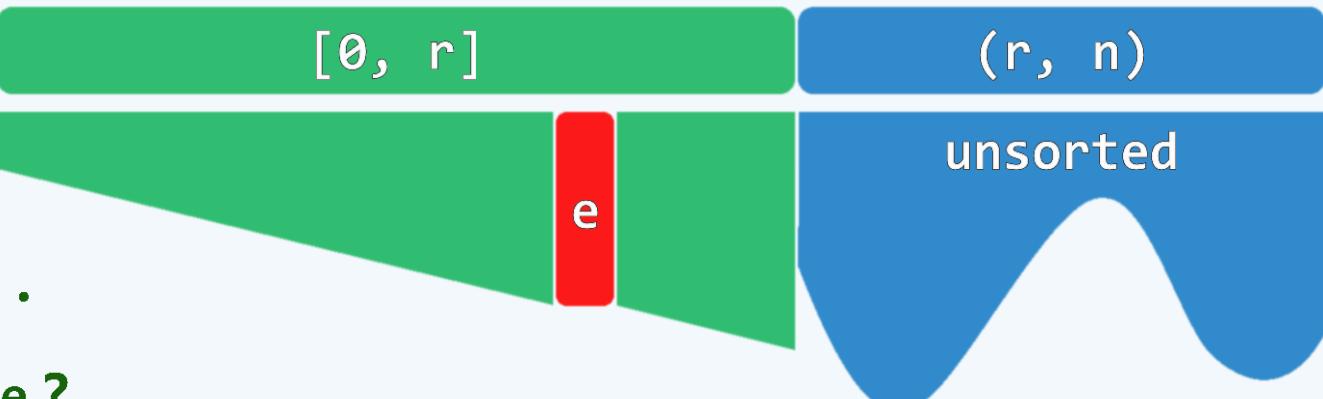
性能分析

- ❖ 属于就地算法 (in-place)
- ❖ 属于在线算法 (online)
- ❖ 具有输入敏感性 (input sensitivity)
 - 后面将会看到 : Shellsort 之类算法的高效性 , 完全依赖于 insertionsort 的这一特性
- ❖ 最好情况 : 完全 (或几乎) 有序
 - 每次迭代 , 只需 1 次比较 , 0 次交换 : 累计 $\Theta(n)$ 时间 !
- ❖ 最坏情况 : 完全 (或几乎) 逆序
 - 第 k 次迭代 , 需 $\Theta(k)$ 次比较 , 1 次交换 : 累计 $\Theta(n^2)$ 时间 !
- ❖ “优化” 的可能 : 在有序前缀中的查找定位 , 为何采用了顺序查找 , 而不是二分查找 ?

平均性能：后向分析

❖ 假定：各元素的取值系均匀独立分布

于是：平均要做多少次元素比较？



❖ 考查： $e=[r]$ 刚插入完成的那一时刻...

试问：此时的有序前缀 $[0, r]$ 中，谁是 e ？

❖ 观察：其中的 $r+1$ 个元素均有可能，且概率均为 $1/(r+1)$

❖ 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为 $\sum_{r=0}^{n-1} r/2 + 1 = \mathcal{O}(n^2)$

❖ 再问：在 n 次迭代中，平均有多少次无需交换呢？

//习题[3-10]

列表

归并排序

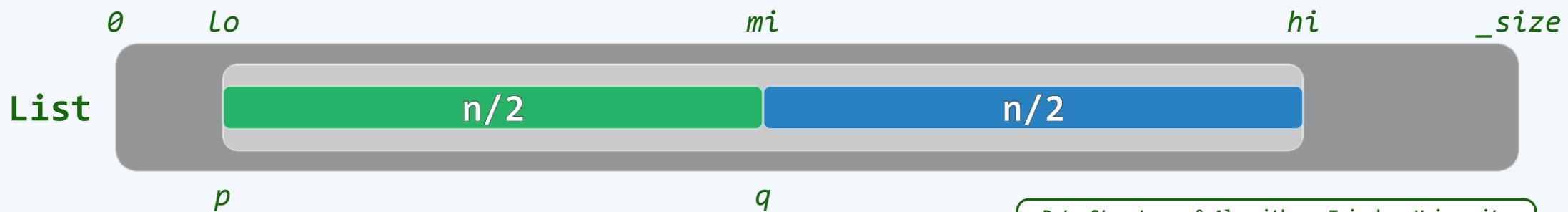
邓俊辉

deng@tsinghua.edu.cn

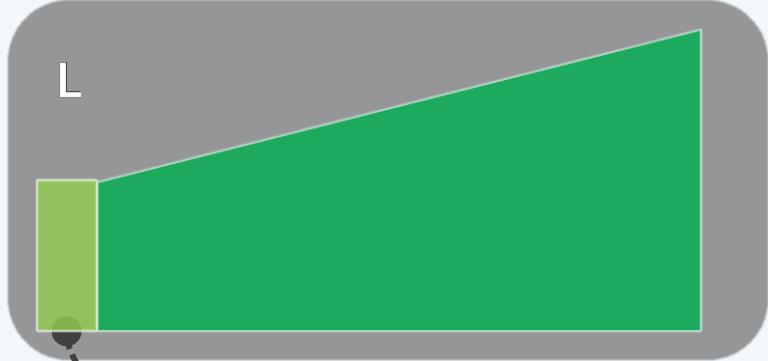
曰两美其必合兮，孰信修而慕之？思九州岛之博大兮，岂惟是其有女？

主算法

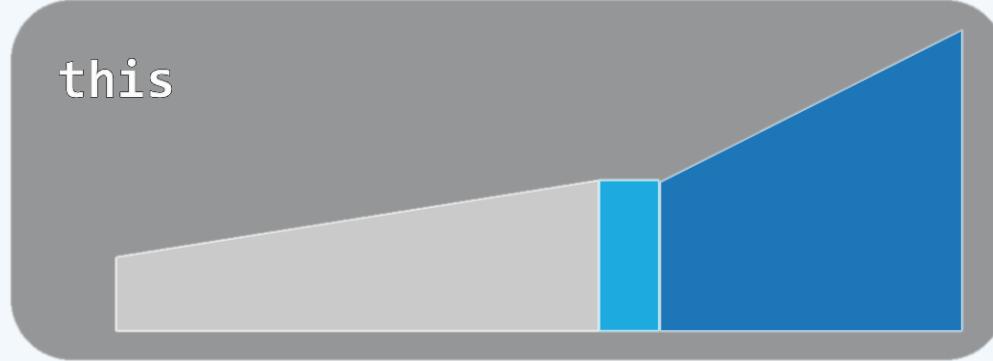
```
❖ template <typename T> //valid(p) && rank(p) + n <= size  
void List<T>::mergeSort( Posi(T) & p, int n ) { //对起始于位置p的n个元素排序  
    if ( n < 2 ) return; //待排序范围足够小时直接返回，否则...  
    Posi(T) q = p; int m = n >> 1; //以中点为界  
    for ( int i = 0; i < m; i++ ) q = q->succ; //均分列表： $\mathcal{O}(m) = \mathcal{O}(n)$   
    mergeSort( p, m ); mergeSort( q, n - m ); //子序列分别排序  
    merge( p, m, *this, q, n - m ); //归并  
} //若归并可在线性时间内完成，则总体运行时间为 $\mathcal{O}(n \log n)$ 
```



二路归并：算法

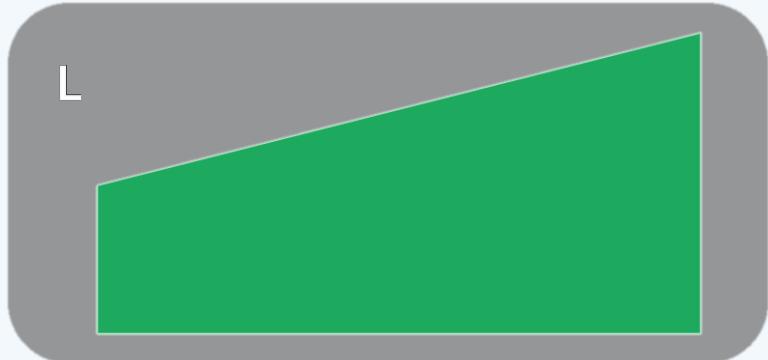


q

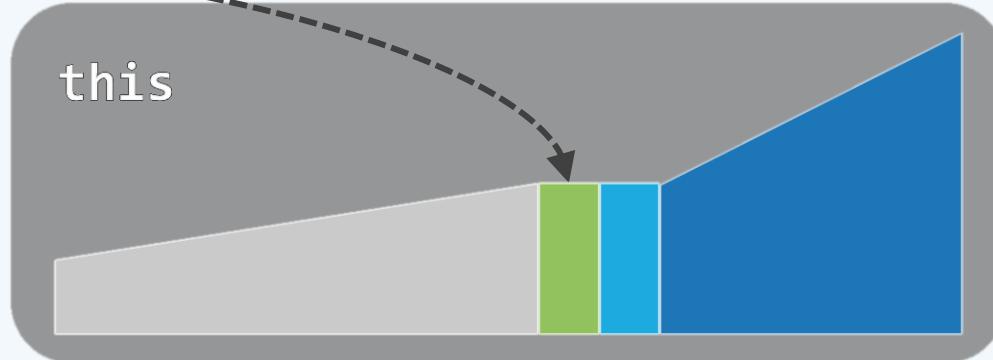


p

transfer



q



p

二路归并：实现

```
template <typename T> //自p起的n个元素，与L中自q起的m个元素归并（归并排序时L==this）  
void List<T>::merge( Posi(T) & p, int n, List<T> & L, Posi(T) q, int m ) {  
    ListNodePosi(T) pp = p->pred; //借助前驱（可能是header），以便返回前 ...  
    while ( 0 < m ) //在q尚未移出区间之前  
        if ( ( 0 < n ) && ( p->data <= q->data ) ) //若p仍在区间内且v(p) <= v(q)  
            { if ( q == (p = p->succ) ) break; n--; } //则将p直接后移  
        else //若p已超出右界或v(q) < v(p)，则将q插至p之前  
            { insertB( p, L.remove( ( q = q->succ )->pred ) ); m--; }  
        p = pp->succ; //确定归并后区间的（新）起点  
    } //运行时间O(n + m)，线性正比于节点总数
```

列表

逆序对

邓俊辉

deng@tsinghua.edu.cn

有象斯有對，對必反其為；有反斯有讎，讎必和而解

Inversion

♦ 考查序列 $A[0, n)$ ，设元素之间可比较大小

$\langle i, j \rangle$ is called an inversion if $0 \leq i < j < n$ and $A[i] > A[j]$

♦ 为便于统计，可将逆序对统一记到**后者的**账上

$\mathcal{I}(j) = \| \{ 0 \leq i < j \mid A[i] > A[j] \text{ and hence } \langle i, j \rangle \text{ is an inversion} \} \|$

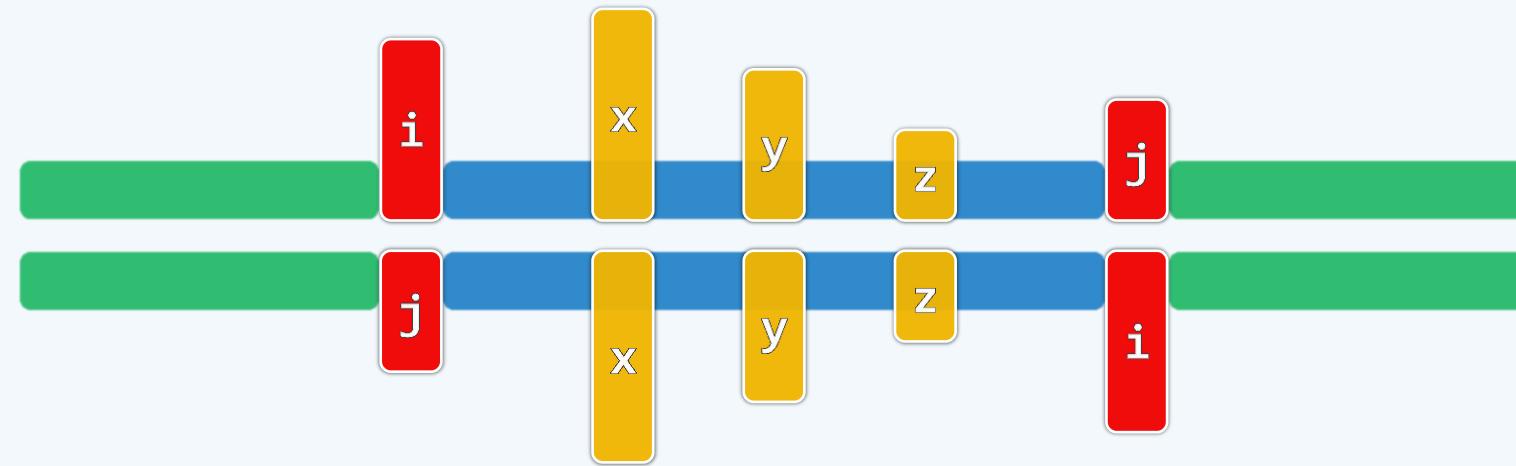
♦ 例： $A[] = \{ 5, 3, 1, 4, 2 \}$ 中，共有 $0 + 1 + 2 + 1 + 3 = 7$ 个逆序对

$A[] = \{ 1, 2, 3, 4, 5 \}$ 中，共有 $0 + 0 + 0 + 0 + 0 = 0$ 个逆序对

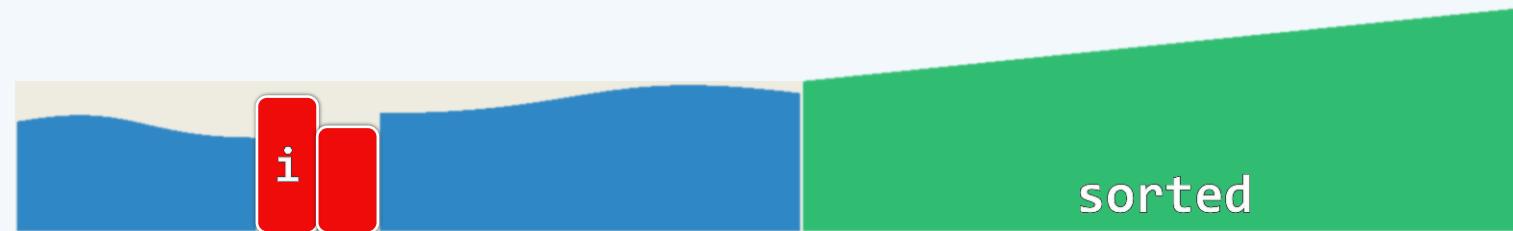
$A[] = \{ 5, 4, 3, 2, 1 \}$ 中，共有 $0 + 1 + 2 + 3 + 4 = 10$ 个逆序对

♦ 显然，逆序对总数 $\mathcal{I} = \sum_j \mathcal{I}(j) \leq \binom{n}{2} = \mathcal{O}(n^2)$

Bubblesort



❖ 在序列中交换一对逆序元素，逆序对总数必然减少



❖ 在序列中交换一对紧邻的逆序元素，逆序对总数恰好减一

❖ 因此对于Bubblesort算法而言，交换操作的次数恰等于输入序列所含逆序对的总数

Insertionsort

❖ 针对任一元素 $e = A[r]$ 的那一步迭代

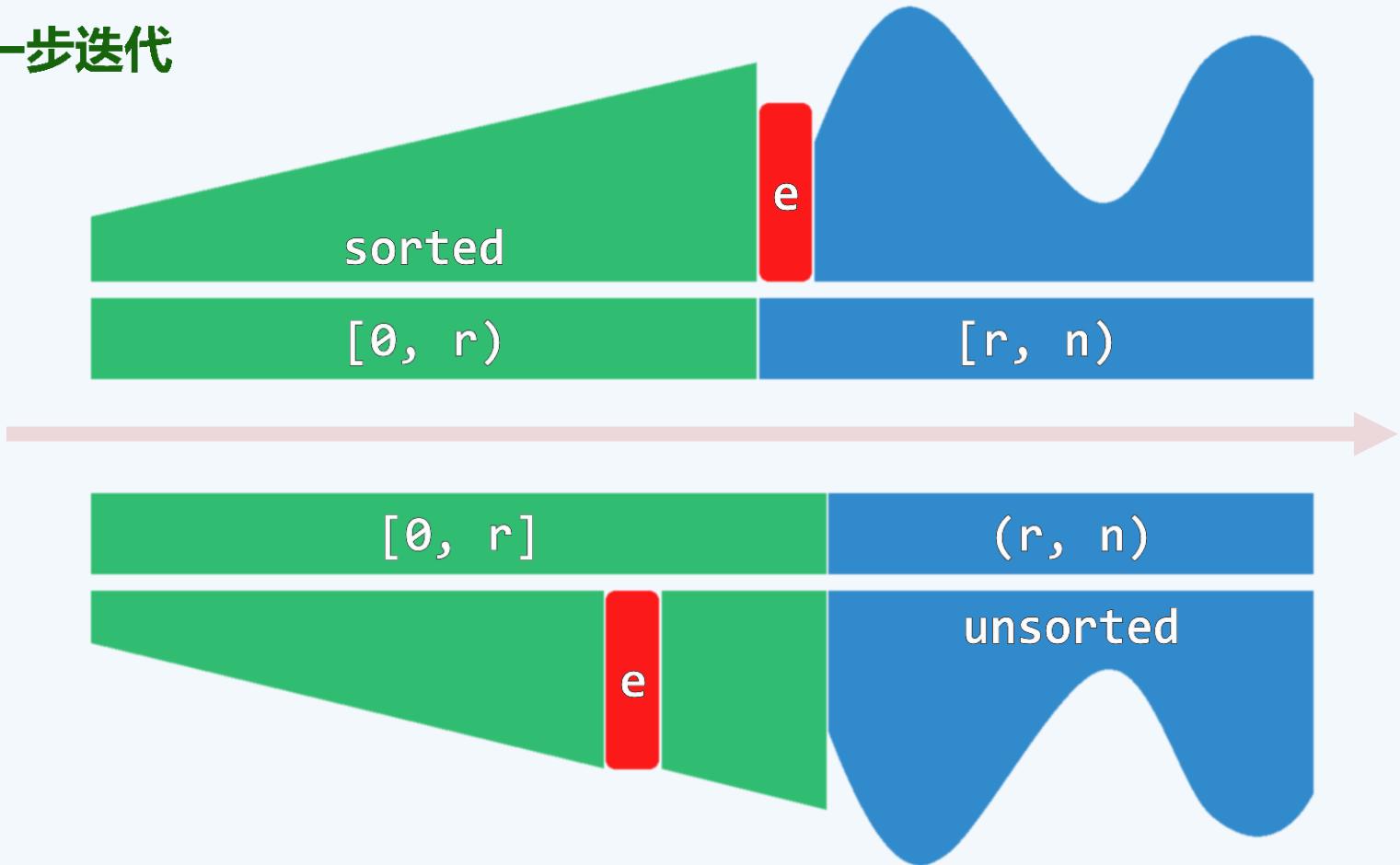
恰好需要做 $\mathcal{I}(r)$ 次比较

❖ 若共含 \mathcal{I} 个逆序对，则

- 关键码比较次数为 $\mathcal{O}(\mathcal{I})$
- 运行时间为 $\mathcal{O}(n + \mathcal{I})$

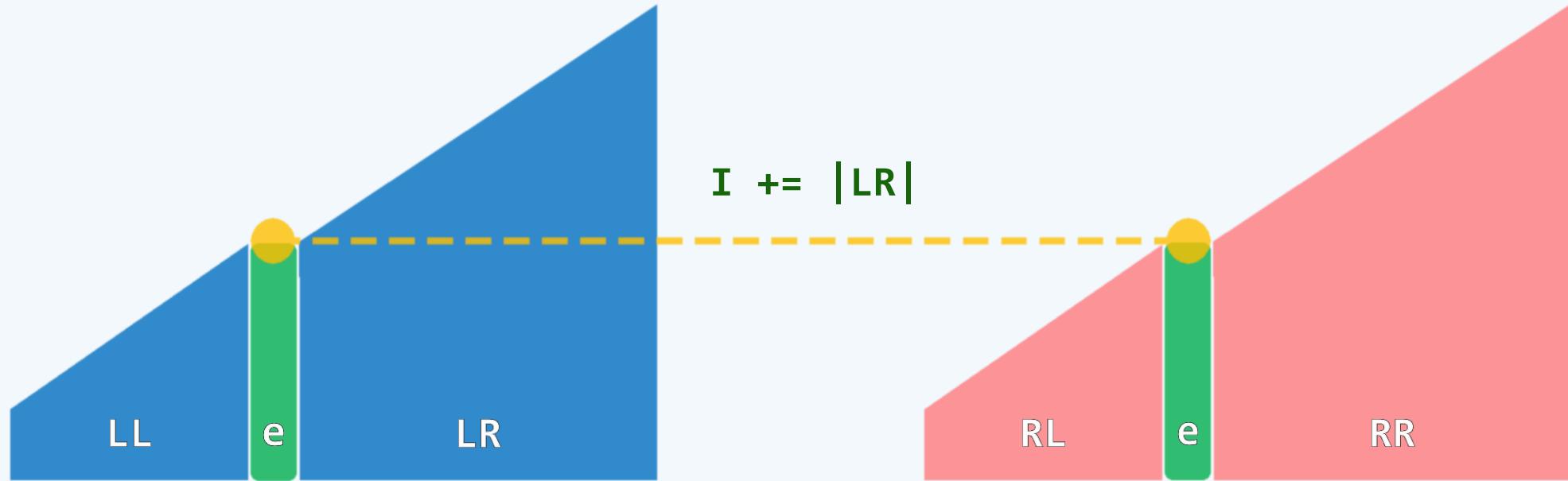
// 习题[3-11]

// 输入敏感性



计数

- ❖ 任意给定一个序列，如何统计其中逆序对的总数？
- ❖ 蛮力算法需要 $\Omega(n^2)$ 时间；借助归并排序，仅需 $\mathcal{O}(n \log n)$ 时间...



列表

游标实现

邓俊辉

deng@tsinghua.edu.cn

动机与构思

❖ 某些特定语言或环境中

- 或者不（直接）支持指针
- 或者不支持动态空间分配

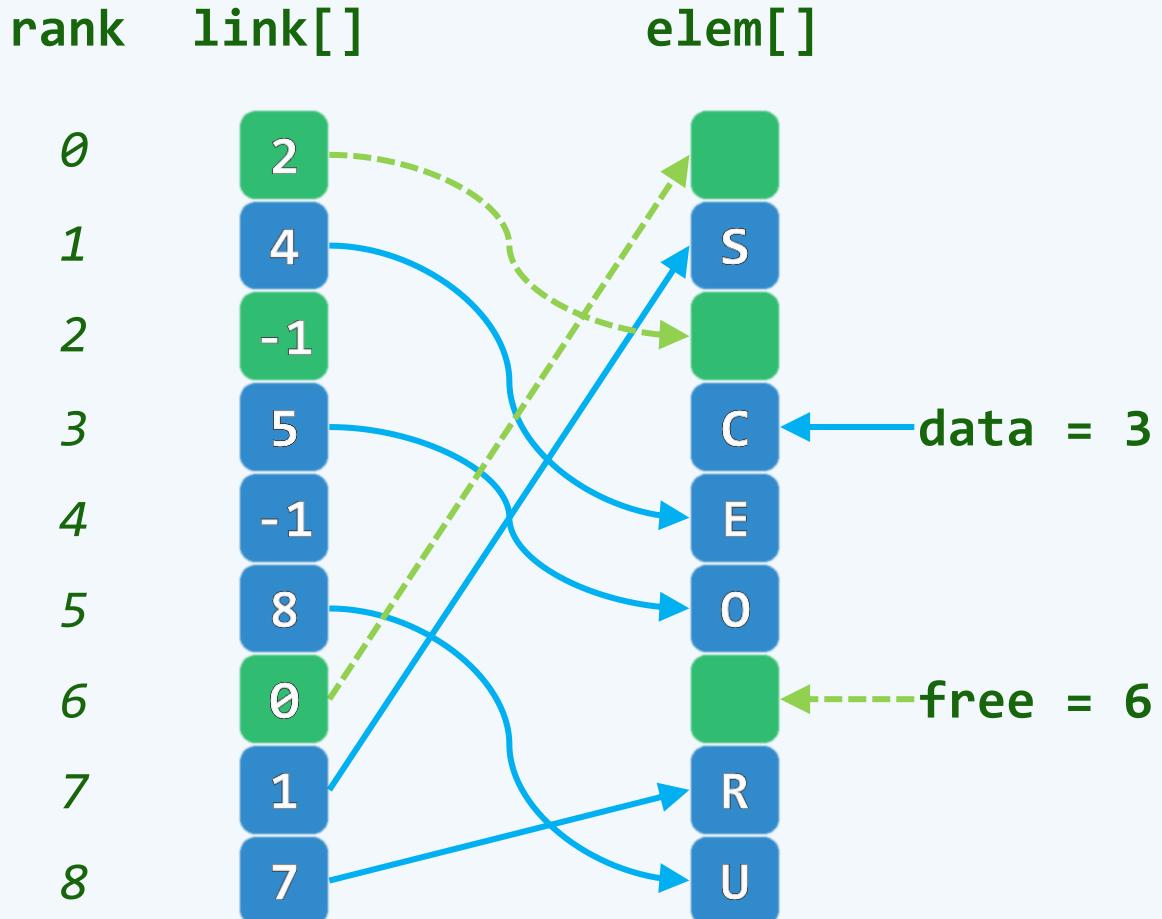
此时，如何实现列表结构呢？

❖ 利用线性数组，以游标方式模拟列表

- elem[] : 对外可见的数据项
- link[] : 数据项之间的引用

❖ 维护逻辑上互补的列表data和free

❖ 在插入或删除元素时，应如何调整？

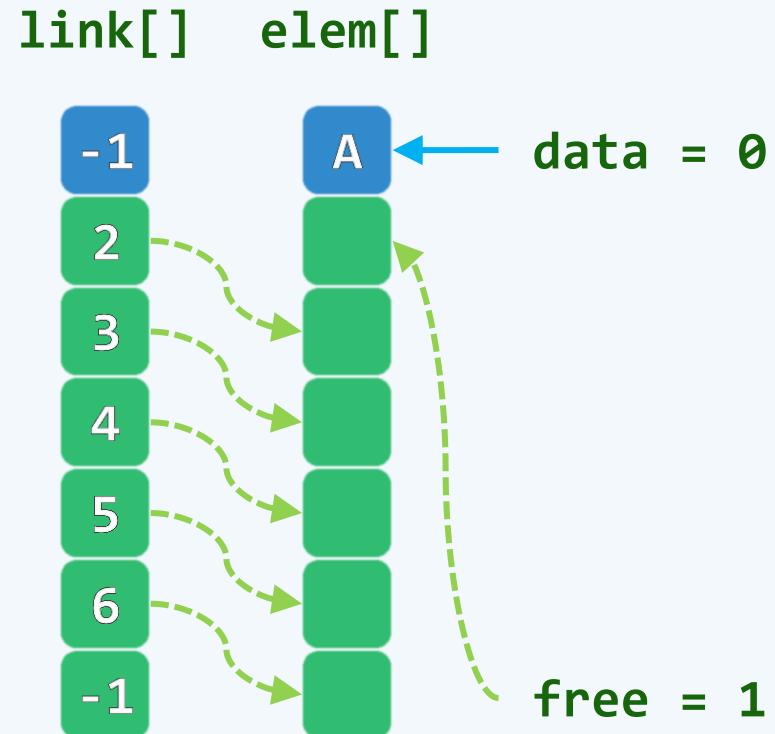


实例 (1/4)

init(7)

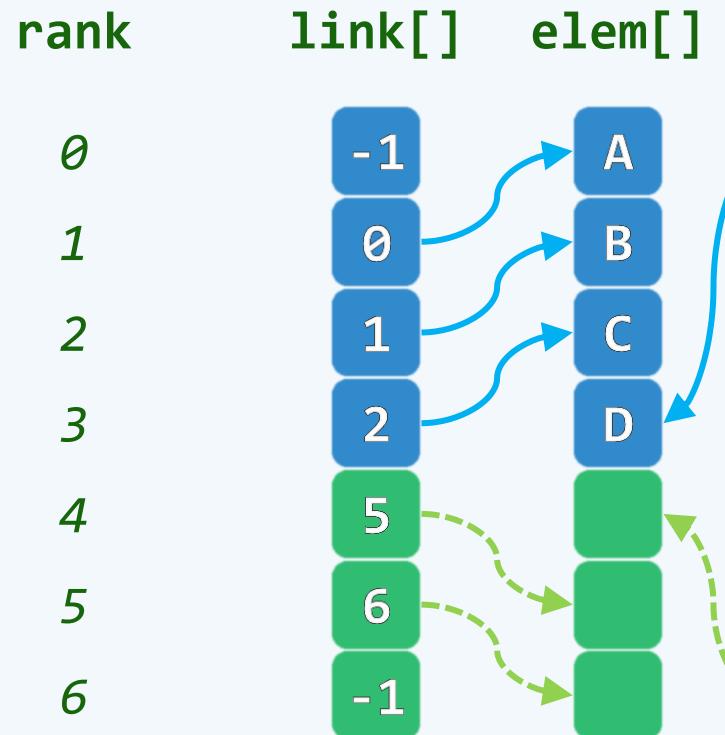


insert('A')

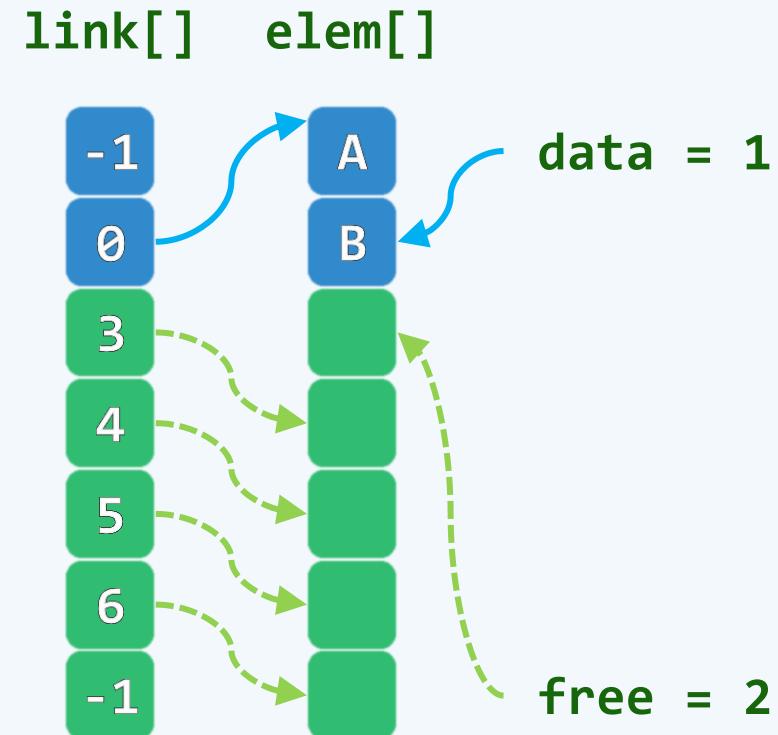


实例 (2/4)

`insert('C')`
`insert('D')`

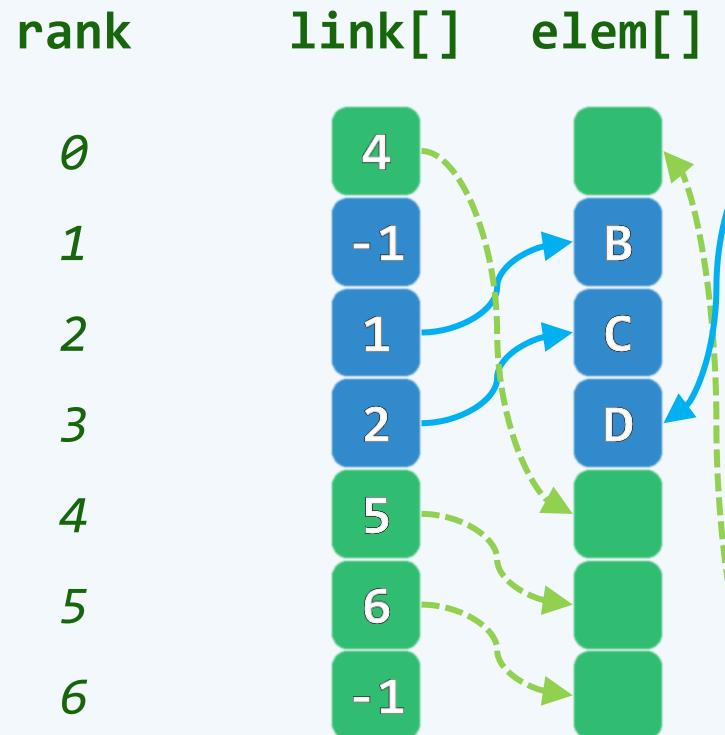


`insert('B')`

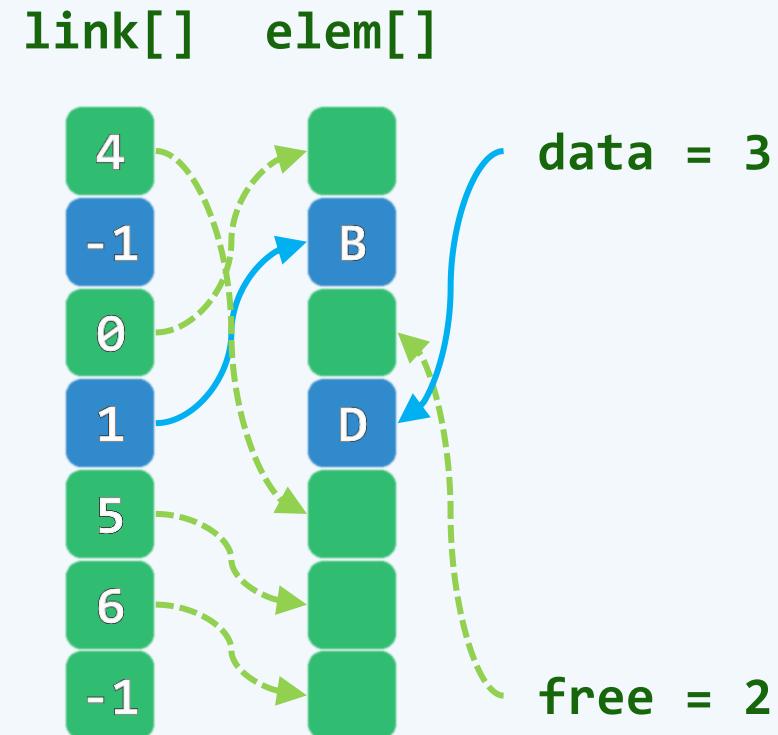


实例 (3/4)

remove('A')



remove('C')



实例 (4/4)

insert('c')

rank	link[]	elem[]
0		
1		
2	2	C
3	-1	B
4	3	A
5	1	D
6	5	
7	6	
8	-1	

Diagram illustrating a linked list structure with two arrays: link[] and elem[]. The link[] array contains indices pointing to the next element in the list. The elem[] array contains the data elements. A pointer variable 'data' is set to 0, and a variable 'free' is set to 4.

insert('A')

link[] elem[]

data = 2

free = 0

列表

Java序列

邓俊辉

deng@tsinghua.edu.cn

Interface : 定义

❖ Java支持ADT的一种机制：在同一接口规范下，允许不同的实现

❖ `interface Geometry { //几何物体`

`final double PI = 3.1415926; //常量定义，类定义可直接使用`

`double area(); //无参数的接口方法`

`boolean inside(Point p); //带参数的接口方法`

`}`

❖ `interface`不能直接实例化为对象

符合`interface`定义的任何类，都需要具体地**实现**其中的接口方法

Interface : 实现

```
class Disk implements Geometry { //符合Geometry接口的Disk类  
    Point c;  double r;  
  
    public Disk( Point center, double radius ) //构造方法  
    {c = center; r = radius;}  
  
    public double perimeter() { return 2 * PI * r; } //类方法  
    public double area() { return PI * r * r; } //接口方法的实现  
    public boolean inside( Point p ) { //接口方法的实现  
        double dx = p.x - c.x, dy = p.y - c.y;  
        return dx*dx + dy*dy < r*r;  
    }  
}
```

向量接口 : Vector.java

```
public interface Vector {  
  
    public int getSize();  
  
    public boolean isEmpty();  
  
    public Object getAtRank( int r ) throws ExceptionBoundaryViolation;  
  
    public Object replaceAtRank( int r, Object obj )  
        throws ExceptionBoundaryViolation;  
  
    public Object insertAtRank( int r, Object obj )  
        throws ExceptionBoundaryViolation;  
  
    public Object removeAtRank( int r ) throws ExceptionBoundaryViolation;  
}
```

向量实现1 : Vector_Array.java

```
public class Vector_Array implements Vector {  
    private final int N = 1024; //数组容量固定  
    private Object[] A; private int n = 0;  
    public Vector_Array() { A = new Object[N]; n = 0; }  
    public int getSize() { return n; }  
    public boolean isEmpty() { return 0 == n; }  
    public Object insertAtRank( int r, Object obj ) throws ExceptionBoundaryViolation {  
        if ( 0 > r || r > n ) throw new ExceptionBoundaryViolation( "out of range" );  
        if ( n >= N ) throw new ExceptionBoundaryViolation( "overflow" );  
        for ( int i = n; i > r; i-- ) A[i] = A[i - 1];  
        A[r] = obj; n++; return obj;  
    }  
    /* ..... */  
}
```

向量实现2 : Vector_ExtArray.java

```
public class Vector_ExtArray implements Vector {  
    private int N = 8; //数组的初始容量，可不断增加  
    /* ..... */  
    public Object insertAtRank( int r, Object obj ) throws ExceptionBoundaryViolation {  
        if ( 0 > r || r > n ) throw new ExceptionBoundaryViolation( "out of range" );  
        if ( N <= n ) { //空间溢出的处理  
            N *= 2; Object B[] = new Object[ N ]; //容量加倍  
            for ( int i = 0; i < n; i++ ) B[i] = A[i]; A = B; //用B[]替换A[]  
        }  
        for ( int i = n; i > r; i-- ) A[i] = A[i - 1]; //后续元素顺次后移  
        A[r] = obj; n++; return obj;  
    }  
    /* ..... */  
}
```

序列接口及其实现

❖ interface List

```
{ /* ... */ }
```

```
class List_DLNode
```

```
implements List
```

```
{ /* ... */ }
```

❖ interface Sequence

```
extends Vector, List
```

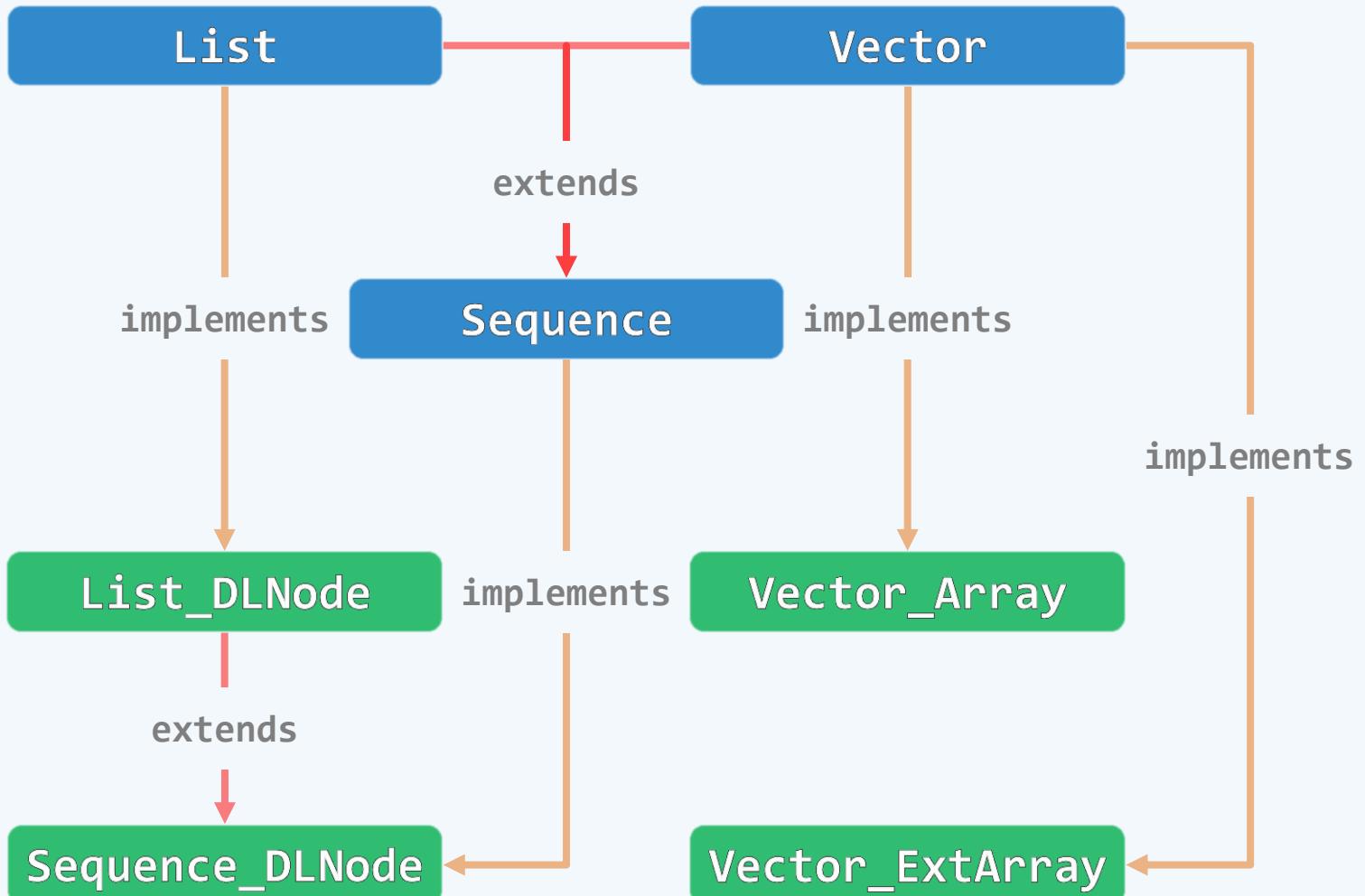
```
{ /* ... */ }
```

```
class Sequence_DLNode
```

```
extends List_DLNode
```

```
implements Sequence
```

```
{ /* ... */ }
```



列表

Python列表

邓俊辉

deng@tsinghua.edu.cn

Python List

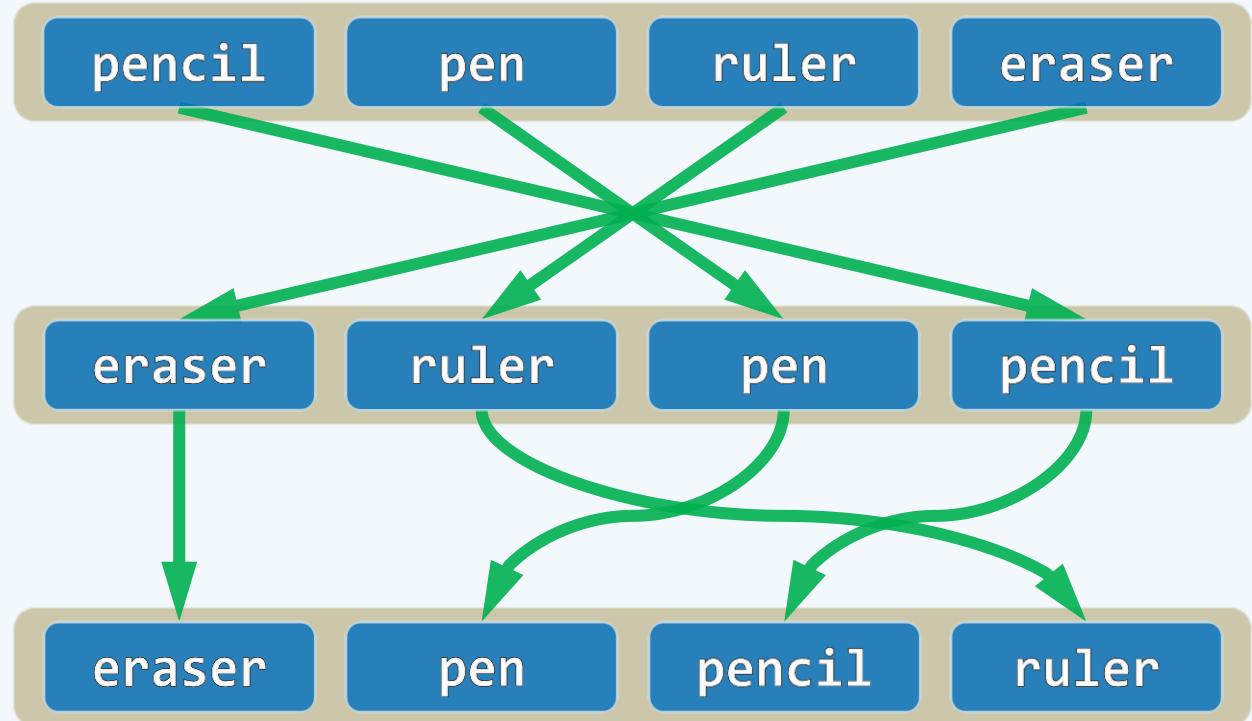
❖ 在Python中，List属于内置的标准数据类型

❖ `box = ['pencil', 'pen', 'ruler', 'eraser']; print box
['pencil', 'pen', 'ruler', 'eraser']`

❖ `for item in box: print item,
pencil pen ruler eraser`

❖ `box.reverse()
for item in box: print item,
eraser ruler pen pencil`

❖ `box.sort()
for item in box: print item,
erase pen pencil ruler`



Python List

```
❖ for i in range(0, len(box)): # [0, n)  
    print box[i],  
    # eraser pen pencil ruler  
  
❖ for i in range(len(box)-1, -1, -1): # [n-1, -1)  
    print box[i],  
    # ruler pencil pen eraser  
  
❖ for i in range(-1, -len(box)-1, -1): # [-1, -n-1)  
    print box[i],  
    # ruler pencil pen eraser
```

eraser

pen

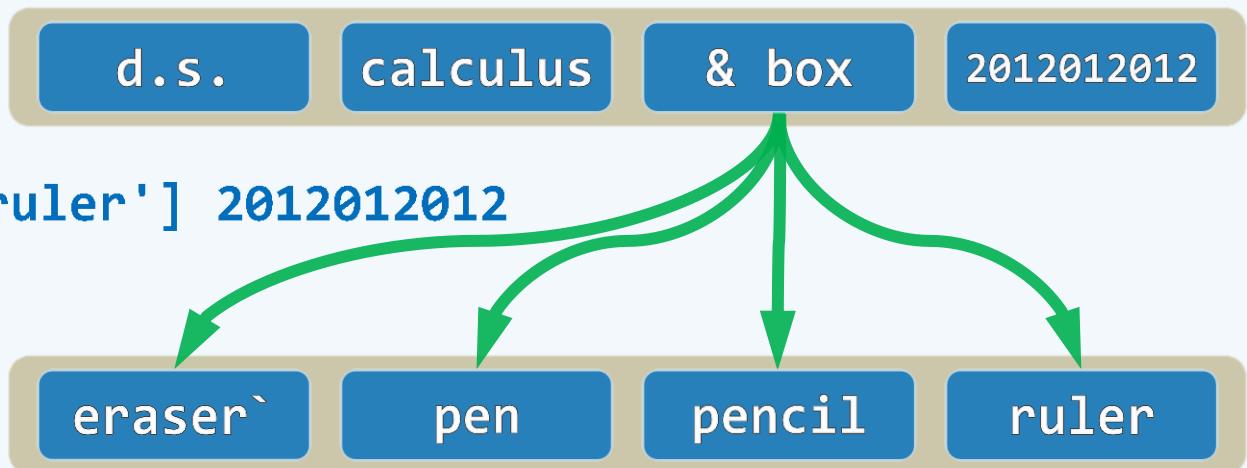
pencil

ruler

Python List

- ❖ bag = ['data structures', 'calculus', box, 2012012012]
print bag

['data structures', 'calculus',
['eraser', 'pen', 'pencil', 'ruler'], 2012012012]
- ❖ for item in bag: print item,
data structures calculus
['eraser', 'pen', 'pencil', 'ruler'] 2012012012
- ❖ for item in bag[2]: print item,
eraser pen pencil ruler
- ❖ for item in bag[2][1:3]: print item,
pen pencil



reverse()

❖ def reverse_1(L): # 循秩访问？

```
lo, hi = 0, len(L) - 1 # 从首、末元素开始  
while lo < hi: # 依次令对称元素  
    L[lo], L[hi] = L[hi], L[lo] # 互换，然后  
    lo, hi = lo + 1, hi - 1 # 考查下一对元素  
return L # 最终即得倒置后的列表
```

❖ def reverse_2(L): # 循位置访问？

```
for i in range(len(L)): # 对[0,n)内的每个i，依次  
    L.insert(i, L.pop()) # 将末元素转移至位置i  
return L # 最终即得倒置后的列表
```