

e3 - A

列表

循位置访问

邓俊辉

deng@tsinghua.edu.cn

Don't lose the link. - Robin Milner

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：put、insert、remove

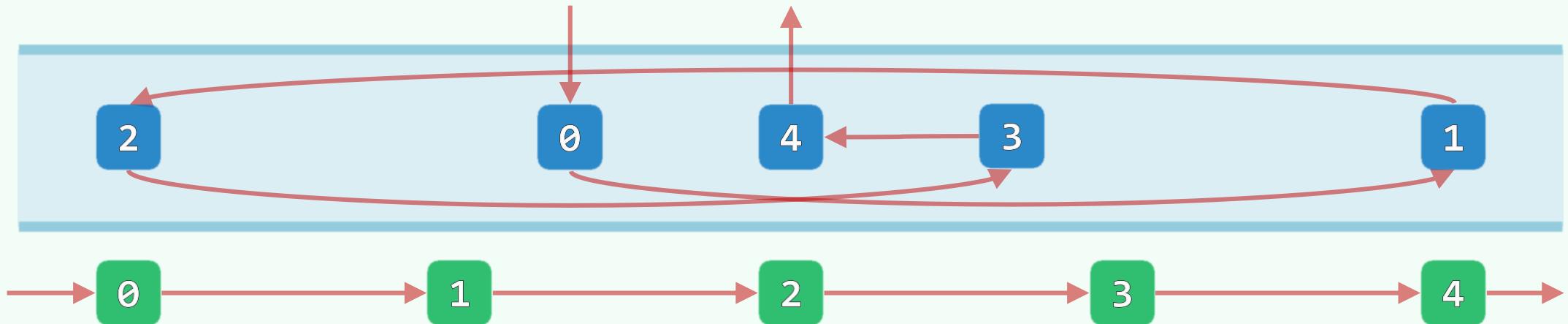
❖ 与操作方式相对应地，数据元素的存储与组织方式也分为两种

- 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**: $\mathcal{L} = \{ a_0, a_1, a_2, \dots, a_{n-1} \}$



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

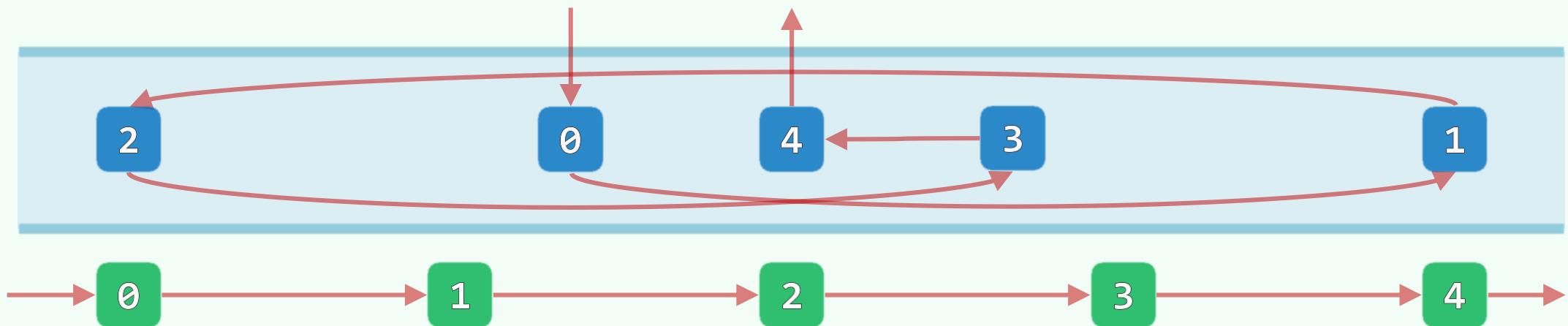
没有前驱/后继的节点称作首 (first/front) /末 (last/rear) 节点

Call-By-Position

❖ 如此，列表中各元素的物理地址将不再决定于逻辑次序

动态操作可以在局部完成，复杂度有望控制在 $\mathcal{O}(1)$

❖ 循位置访问：利用节点之间的相互引用，找到特定的节点



❖ 顺藤摸瓜：找到我的...朋友A的...亲戚B的...同事C的...战友D的...同学Z

如果是按逻辑次序的连续访问，单次也是 $\mathcal{O}(1)$

e3-B

列表

接口与实现

邓俊辉

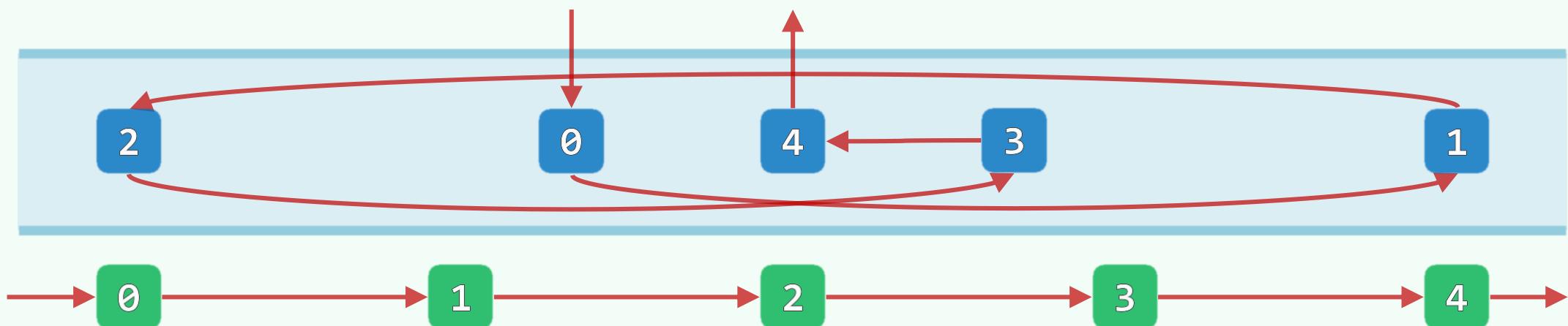
deng@tsinghua.edu.cn

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

ListNode ADT

- ❖ 作为列表的基本元素
列表节点首先需要
独立地“封装”实现
- ❖ 列车 ~ 车厢 ~ 货物
`list ~ node ~ data`

操作接口	功能
<code>pred() / succ()</code>	当前节点前驱/后继节点的位置
<code>data()</code>	当前节点所存数据对象
<code>insertPred() / insertSucc()</code>	插入前驱/后继节点，返回新节点位置



ListNode

```
template <typename T> using ListNodePosi = ListNode<T>*; //列表节点位置 (C++11)
```

```
template <typename T> struct ListNode { //简洁起见，完全开放而不再严格封装
```

```
    T data; //数值
```

```
    ListNodePosi<T> pred; //前驱
```

```
    ListNodePosi<T> succ; //后继
```



```
ListNode() {} //针对head和tail的构造
```

```
ListNode(T const & e, ListNodePosi<T> p = NULL, ListNodePosi<T> s = NULL)
```

```
    : data(e), pred(p), succ(s) {} //默认构造器 (类T须已定义复制方法)
```

```
ListNodePosi<T> insertPred( T const & e ); //前插入
```

```
ListNodePosi<T> insertSucc( T const & e ); //后插入
```

```
}
```

List ADT

操作接口	功能	适用对象
<code>size() / empty()</code>	报告节点总数 / 判定是否为空	列表
<code>first() / last()</code>	返回首 / 末节点的位置	列表
<code>insertFirst(e) / insertLast(e)</code>	将e当作首 / 末节点插入	列表
<code>insert(p, e), insert(e, p)</code>	将e当作节点p的直接后继、前驱插入	列表
<code>remove(p)</code>	删除节点p	列表
<code>sort(p, n) / sort()</code>	区间 / 整体排序	列表
<code>find(e, n, p) / search(e, n, p)</code>	在指定区间内查找目标e	列表 / 有序列表
<code>dedup() / uniquify()</code>	剔除相等的节点	列表 / 有序列表
<code>traverse(visit())</code>	遍历列表，统一按visit()处理所有节点	列表

List

```
#include "listNode.h" //引入列表节点类

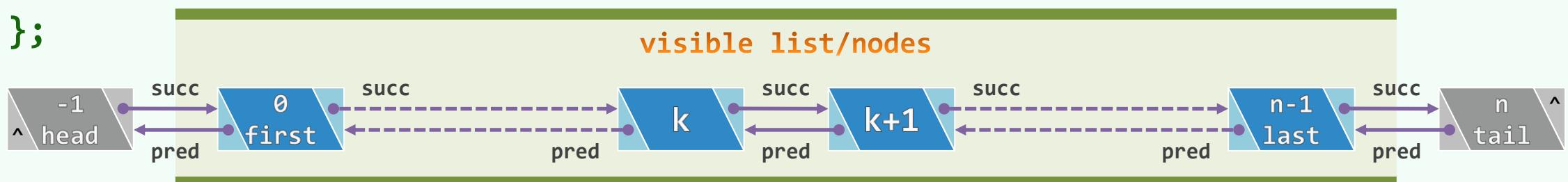
template <typename T> class List { //列表模板类

private:    Rank _size; ListNodePosi<T> head, tail; //哨兵
            //头、首、末、尾节点的秩，可分别理解为-1、0、n-1、n

protected: /* ... 内部函数 */

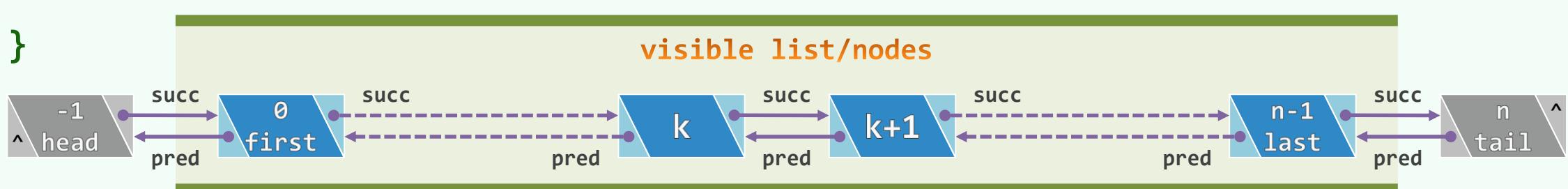
public:   /* ... 构造函数、析构函数、只读接口、可写接口、遍历接口 */

};
```



初始化

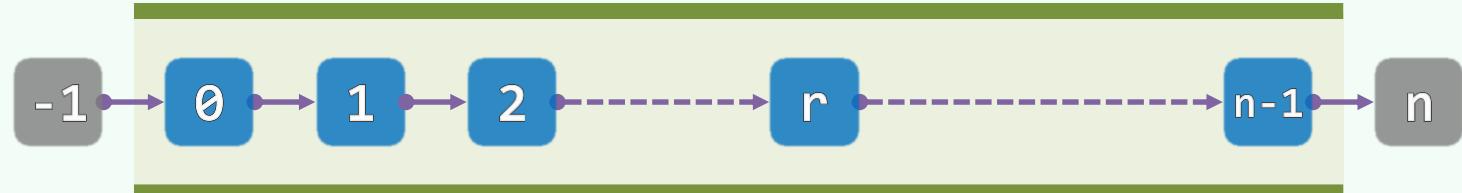
```
template <typename T> void List<T>::init() { // 初始化，创建列表对象时统一调用  
    head = new ListNode<T>;  
    tail = new ListNode<T>;  
    head->succ = tail; head->pred = NULL;  
    tail->pred = head; tail->succ = NULL;  
  
    _size = 0;  
}
```



重载下标操作符，可模仿向量的循秩访问方式

template <typename T> // $\mathcal{O}(r)$ 效率，虽方便，勿多用

```
ListNodePosi<T> List<T>::operator[]( Rank r ) const { // $0 \leq r < \text{size}$ 
    ListNodePosi<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; //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

03 - C1

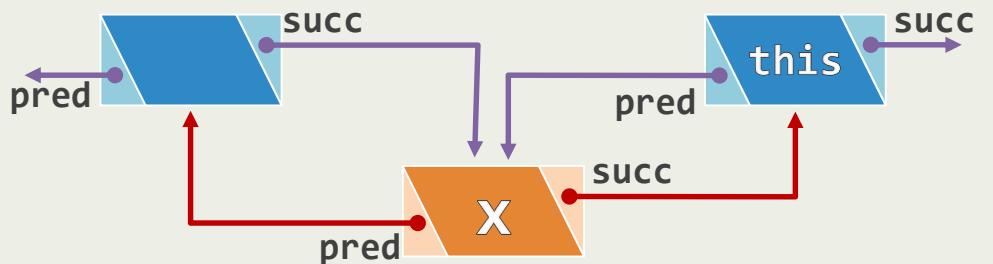
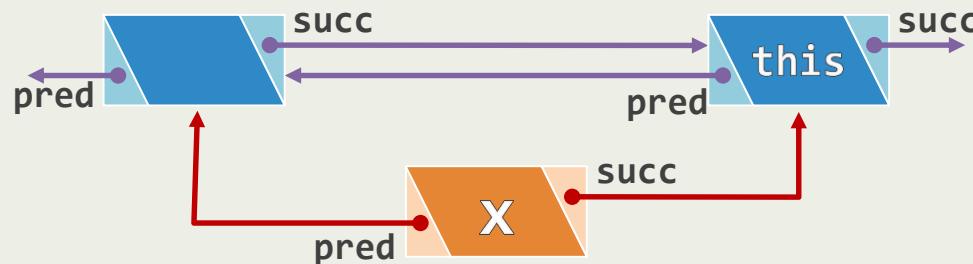
邓俊辉

deng@tsinghua.edu.cn

List::insert(e, p)

template <typename T> ListNodePosi<T> List<T>:: //e当作p的前驱插入

insert(T const & e, ListNodePosi<T> p) { _size++; return p->insertPred(e); }



ListNode::insertPred()

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

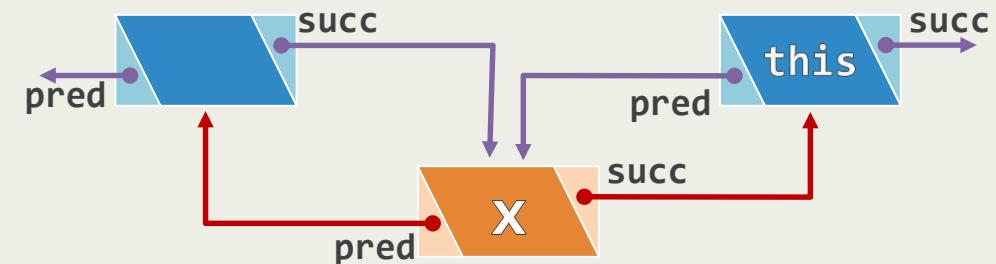
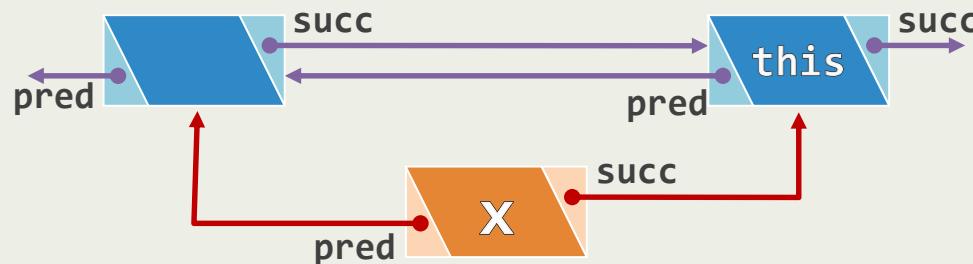
ListNodePosi<T> ListNode<T>::insertPred(T const & e) { //O(1)

ListNodePosi<T> x = new ListNode(e, pred, this); //创建

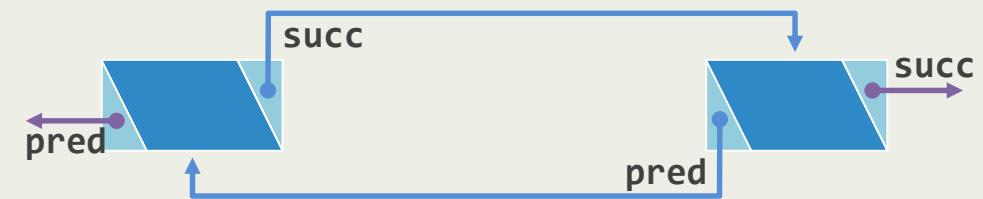
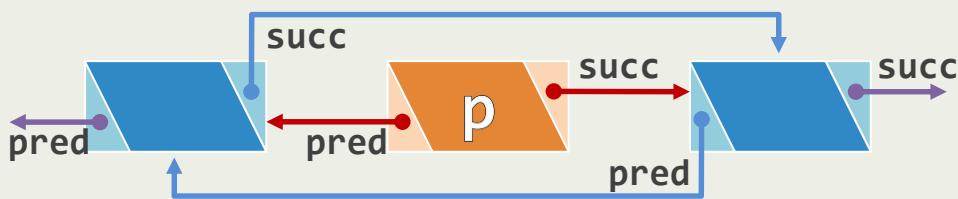
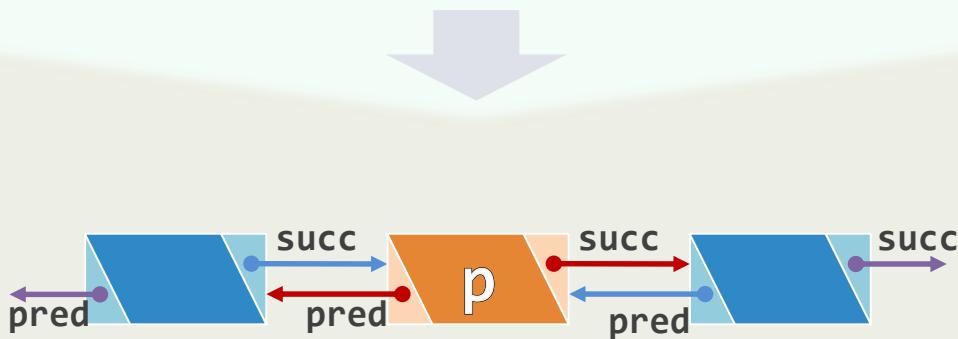
pred->succ = x; pred = x; //次序不可颠倒

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insertFirst(e)

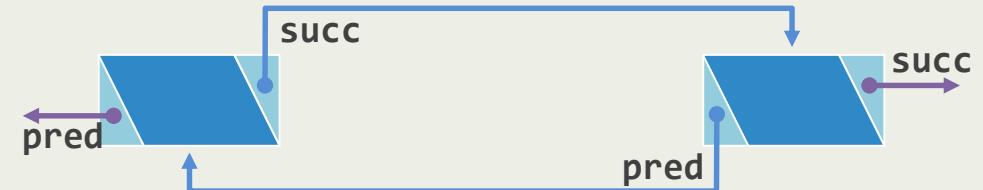
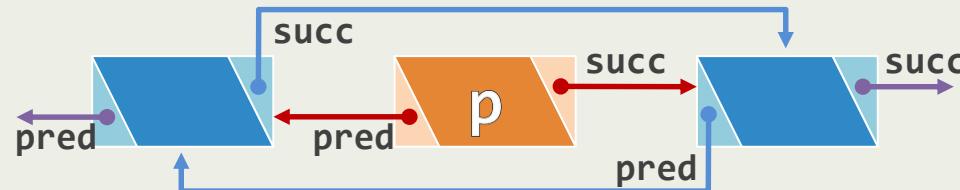


List::remove(p): 思路 + 过程



List::remove(p): 实现

```
template <typename T> T List<T>::remove( ListNodePosi<T> p ) { //删除合法节点p  
    T e = p->data; //备份待删除节点存放的数值 (设类型T可直接赋值)  
    p->pred->succ = p->succ;    p->succ->pred = p->pred; //短路联接  
    delete p; _size--; return e; //返回备份的数值  
} //O(1)
```



列表

无序列表：构造与析构



“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。

精神与我们的官能同生长，同样萎黄：

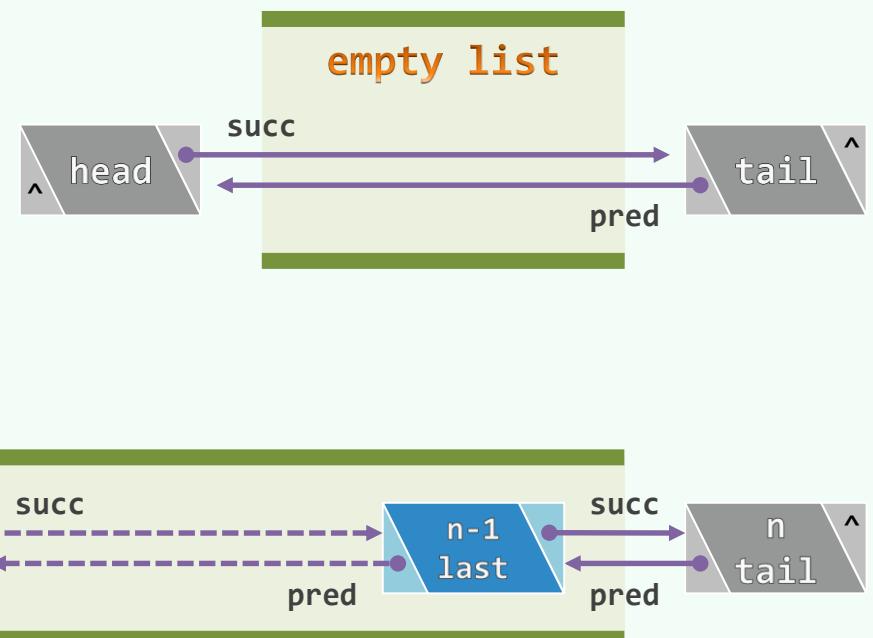
哎呀！它一样要死亡

邓俊辉

deng@tsinghua.edu.cn

copyNodes() + 构造

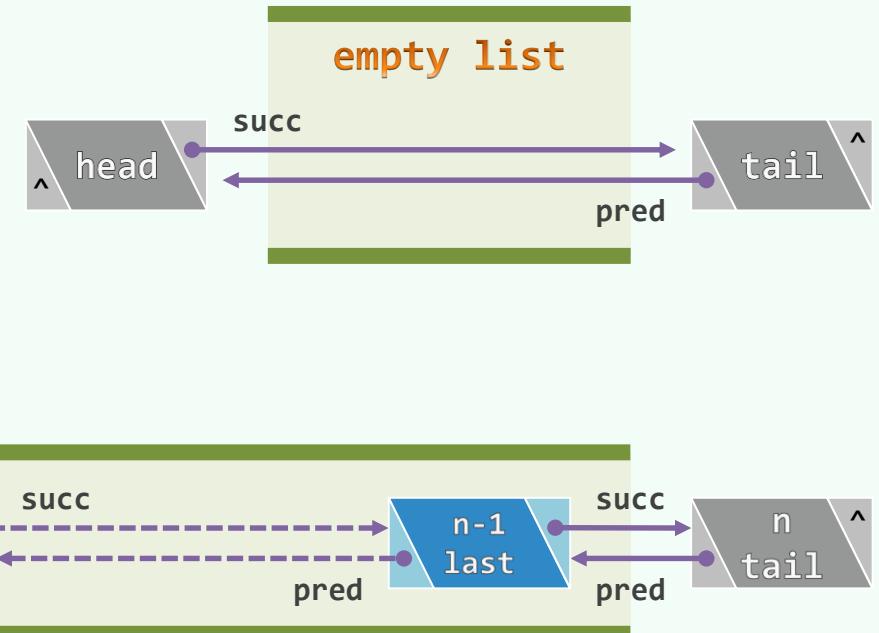
```
template <typename T> void List<T>::copyNodes( ListNodePosi<T> p, Rank n ) { //O(n)  
    init(); //创建头、尾哨兵节点并做初始化  
    while ( n-- ) { //将起自p的n项依次作为末节点  
        insertLast( p->data ); //插入  
        p = p->succ;  
    }  
}  
List<T>::List( List<T> const & L ) { copyNodes( L.first(), L._size ); }
```



clear() + 析构

```
template <typename T> List<T>::~List() //列表析构
{   clear(); delete head; delete tail; } //清空列表，释放头、尾哨兵节点

template <typename T> Rank List<T>::clear() { //清空列表
    Rank oldSize = _size;
    while ( 0 < _size ) //反复
        remove( head->succ ); //删除首节点, O(n)
    return oldSize;
}
```



❖ 若remove(head->succ)改作remove(tail->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
ListNodePosi<T> List<T>::find( T const & e, Rank n, ListNodePosi<T> p ) const {
    while ( 0 < n-- ) //自后向前
        if ( e == ( p = p->pred ) ->data ) //逐个比对 (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
} //O(n)
```



```
template <typename T>
ListNodePosi<T> find( T const & e ) const { return find( e, _size, tail ); }
```

去重

```
template <typename T> Rank List<T>::dedup() {
```

```
Rank oldSize = _size;
```

```
ListNodePosi<T> p = first();
```

q?

p

[r, n)

(a)

```
for ( Rank r = 0; p != tail; p = p->succ ) //O(n)
```

```
if ( ListNodePosi<T> q = find( p->data, r, p ) ) //O(n)
```

```
remove ( q );
```

p

(c)

```
else
```

```
r++; //无重前缀的长度
```

p

(b)

```
return oldSize - _size; //删除元素总数
```

```
} //正确性及效率分析的方法与结论，与Vector::dedup()相同
```

列表

无序列表：遍历

邓俊辉

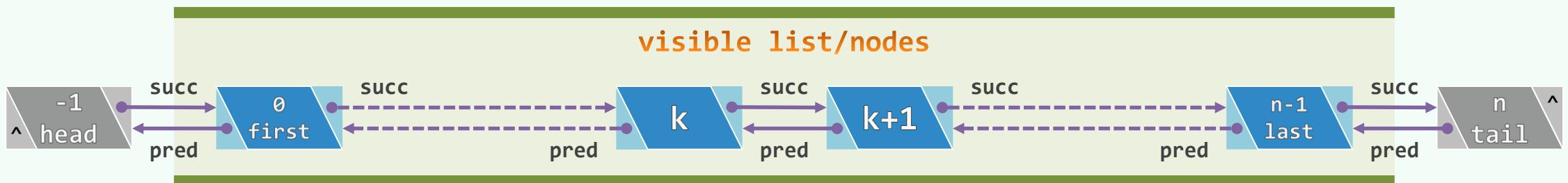
deng@tsinghua.edu.cn

03 - C4

八戒道：不要扯，等我一家六吃将来

traverse(): 函数指针/函数对象

```
template <typename T> void List<T>::traverse( void ( * visit )( T & ) )  
{ for( NodePosi<T> p = head->succ; p != tail; p = p->succ ) visit( p->data ); }
```



```
template <typename T> template <typename VST> void List<T>::traverse( VST & visit )  
{ for( NodePosi<T> p = head->succ; p != tail; p = p->succ ) 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

古者诗三千余篇，及至孔子，去其重，取可施于礼义

03 - D1

邓俊辉
deng@tsinghua.edu.cn

uniquify()



```
template <typename T> Rank List<T>::uniquify() {  
    if ( _size < 2 ) return 0; //平凡列表，自然不含相等元素  
  
    Rank 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 ( tail != ( 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（相等）直接删除后者，不必如向量那样间接地完成删除  
  
    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， $\mathcal{O}(n)$ 
```

列表

有序列表：查找



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

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

邓俊辉

deng@tsinghua.edu.cn

search()

```
template <typename T> //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
ListNodePosi<T> List<T>::search( T const & e, Rank n, ListNodePosi<T> p ) const {
```

do { //初始有: $0 \leq n \leq \text{rank}(p) < \text{_size}$; 此后, n总是等于p在查找区间内的秩
 p = p->pred; n--; //从右向左

} while ((-1 != n) && (e < p->data)); //逐个比较, 直至越界或命中

```
return p; //最终停止的位置; 失败时为区间左边界前驱 (可能就是head)
```

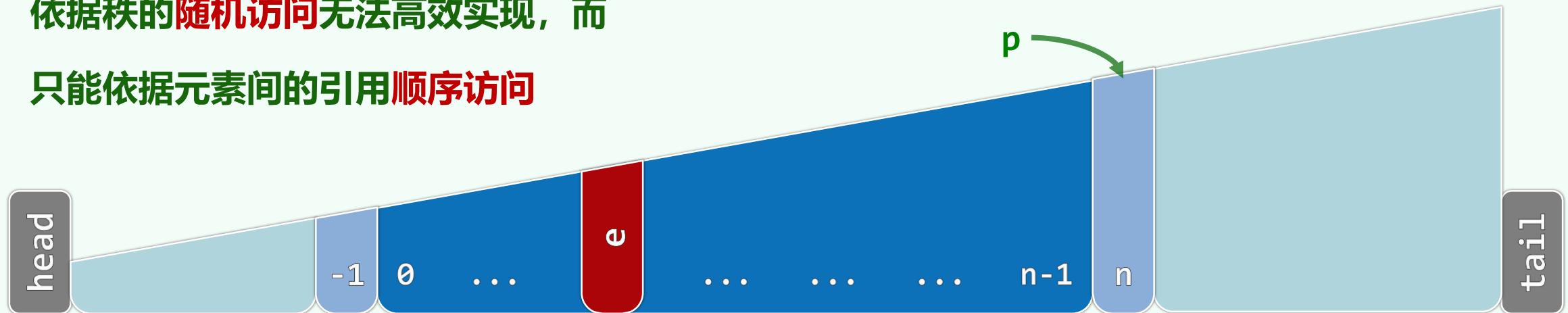
} //调用者可据此判断查找是否成功



性能 + 拓展

- ❖ 最好 $\mathcal{O}(1)$, 最坏 $\mathcal{O}(n)$; 等概率时平均 $\mathcal{O}(n)$, 正比于区间宽度
- ❖ 语义与向量相似, 便于插入排序等后续操作: `insert(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

卡修斯永远讲道德，永远正经
他认为容忍恶棍的人自己就近于恶棍
只有在吃饭的时候——无疑他要选择
一个有鹿肉的坏蛋，而不要没肉的圣者

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

e3 - E

邓俊辉

deng@tsinghua.edu.cn

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

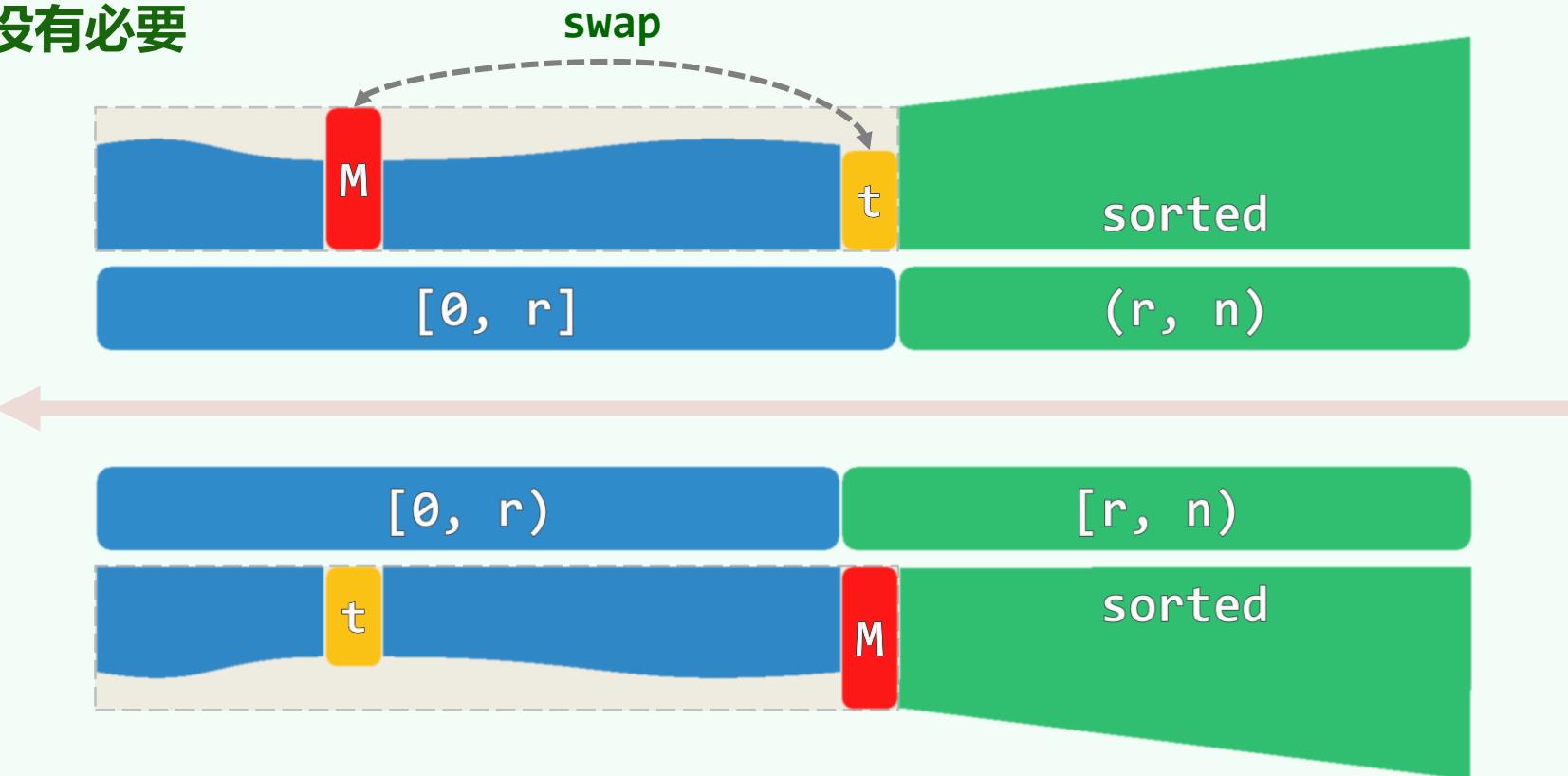
- 通过比较找到当前的

最大元素M，并

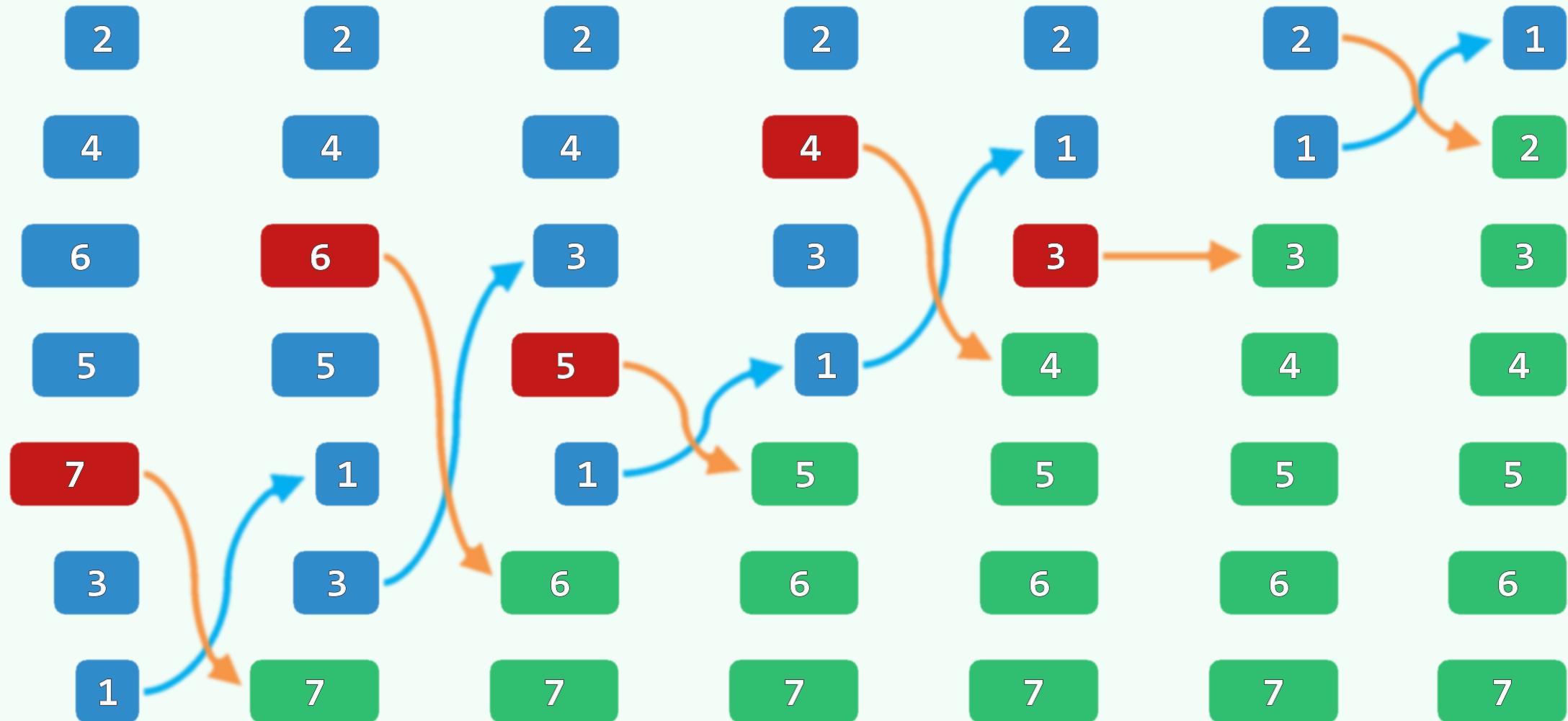
- 通过交换使之就位

❖ 如此看来

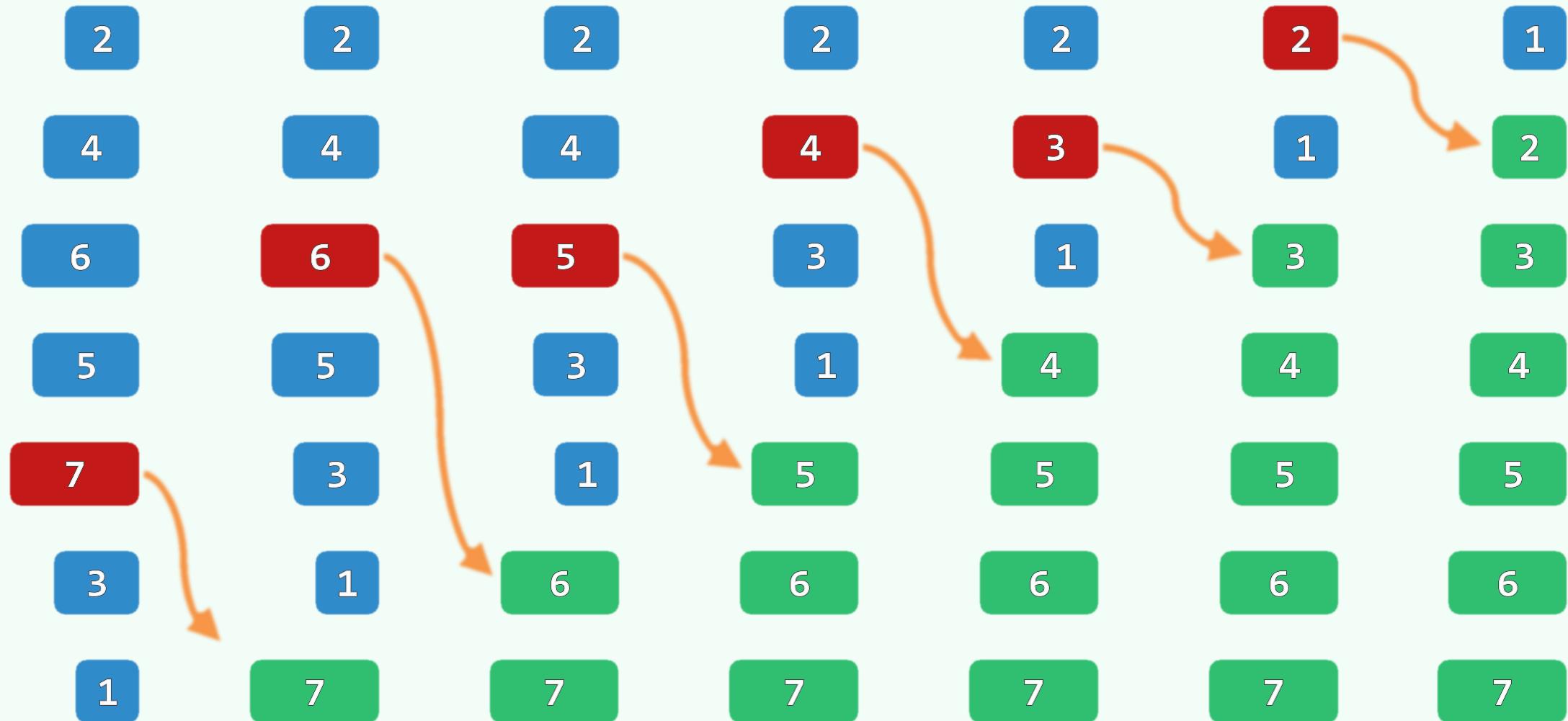
在经 $\Theta(n)$ 次比较确定M之后，仅需一次交换即足矣



交换法

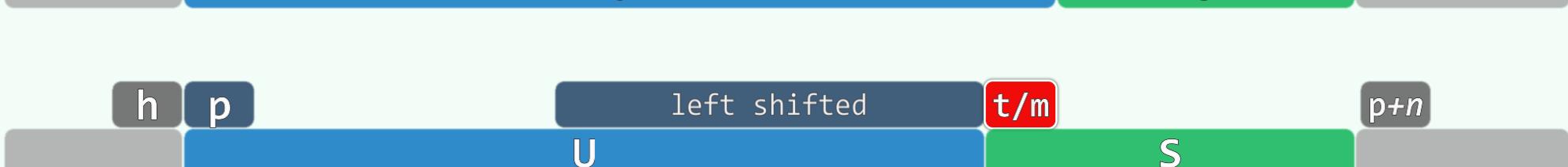
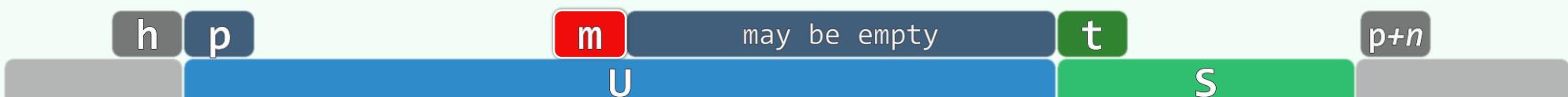


平移法



selectionSort()

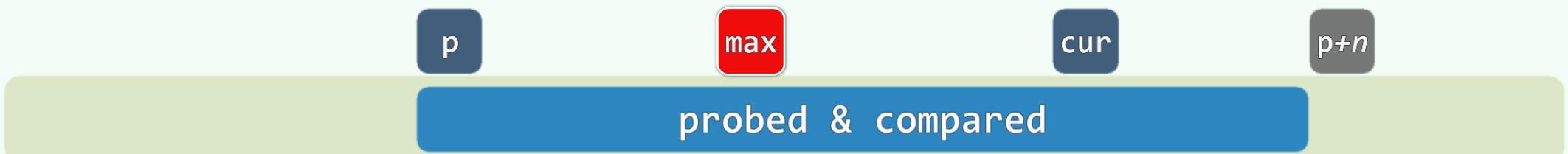
```
template <typename T> void List<T>::selectionSort( ListNodePosi<T> p, Rank n ) {  
    ListNodePosi<T> h = p->pred; //待排序区间为(h, t)  
  
    ListNodePosi<T> t = p; for ( Rank i = 0; i < n; i++ ) t = t->succ;  
  
    while ( 1 < n ) { //反复从(非平凡)待排序区间内找出最大者，并移至有序区间前端  
        insert( remove( selectMax( h->succ, n ) ), t ); //可能就在原地  
  
        t = t->pred; n--; //待排序区间、有序区间的范围，均同步更新  
    }  
}
```



selectMax()

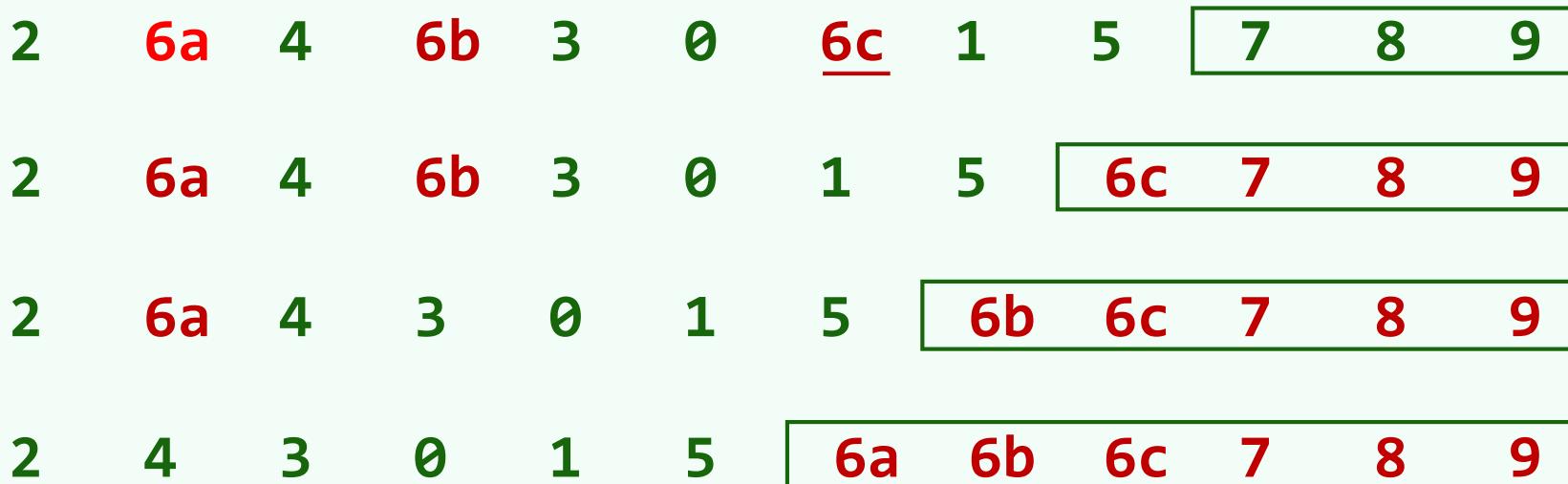
```
template <typename T> //从起始于位置p的n个元素中选出最大者, 1 < n  
ListNodePosi<T> List<T>::selectMax( ListNodePosi<T> p, Rank n ) { //Θ(n)  
    ListNodePosi<T> max = p; //最大者暂定为p  
  
    for ( ListNodePosi<T> cur = p; 1 < n; n-- ) //后续节点逐一与max比较  
        if ( !(cur = cur->succ)->data < max->data ) //data ≥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

}

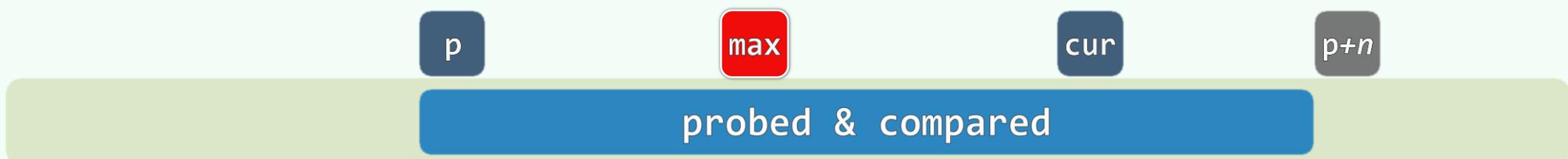


稳定性: 有多个元素同时命中时, 约定返回其中最靠后者

❖ 回看List<T>::selectMax(): 这里是如何保证后者优先的?



❖ 若采用平移法, 如此即可保证稳定性: 每一组相等的元素, 都保持输入时的相对次序



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() + insert()

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 $\mathcal{O}(\log n)$

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

//稍后分解

//保持兴趣

列表

循环节

e3 - F

“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

Cycle

- ❖ 任何一个序列 $\mathcal{A}[0, n)$, 都可以分解为若干个循环节 //设元素之间确实可以定义次序
- ❖ 任何一个序列 $\mathcal{A}[0, n)$, 都对应于一个有序序列 $\mathcal{S}[0, n)$ //经排序之后
- ❖ 元素 $\mathcal{A}[k]$ 在 \mathcal{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: $k, r(k), r(r(k)), r(r(r(k))), \dots, \overbrace{r(\dots(r(r(k)))\dots)}^d = k$
 $r^0(k), r^1(k), r^2(k), r^3(k), \dots, r^{(d)}(k) = k$
- ❖ 任一循环节的长度 $d \leq n$
- ❖ 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

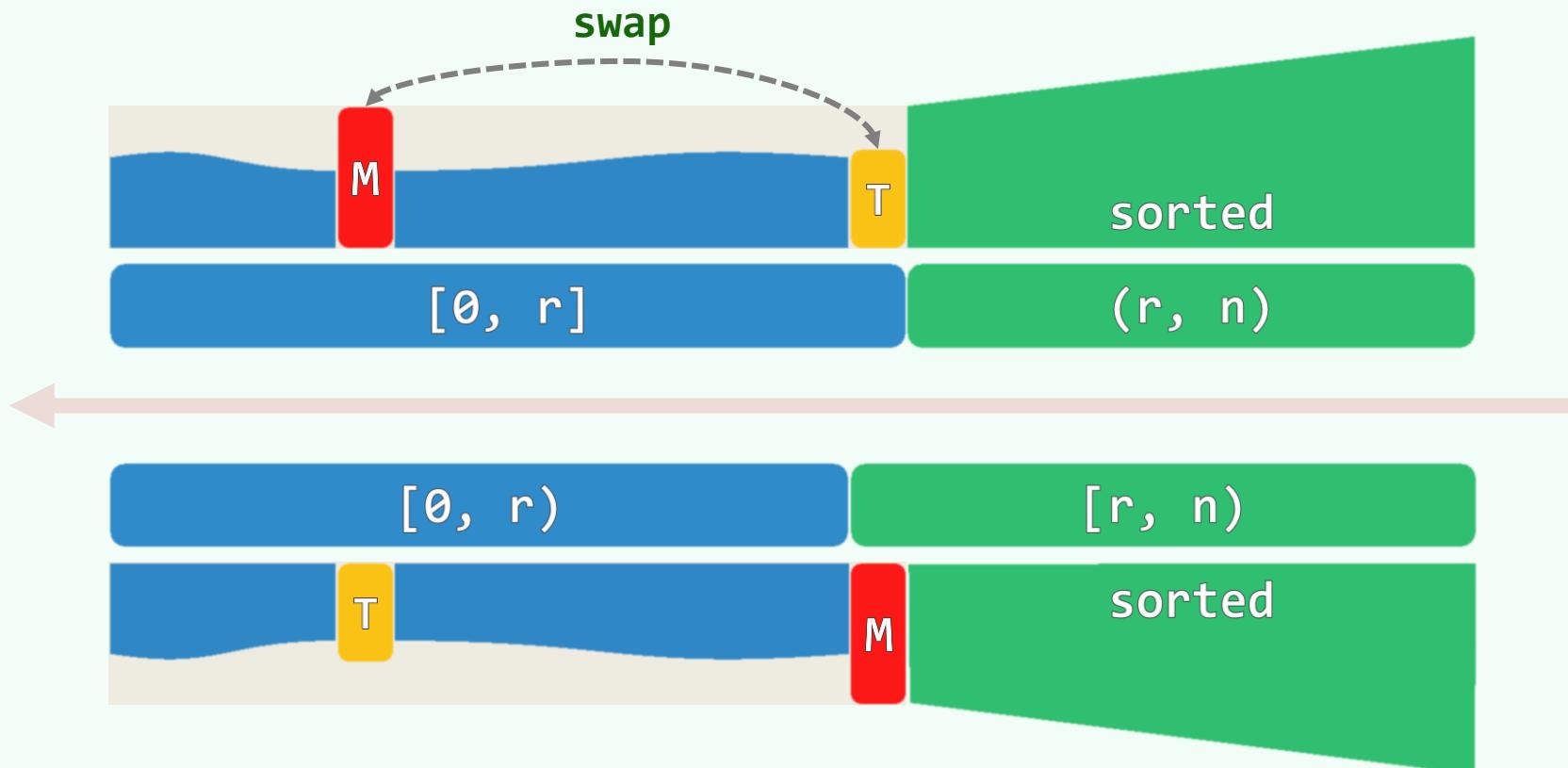
$S[]$: A B C D E F G H I J K L M N O P

$r[]$: 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

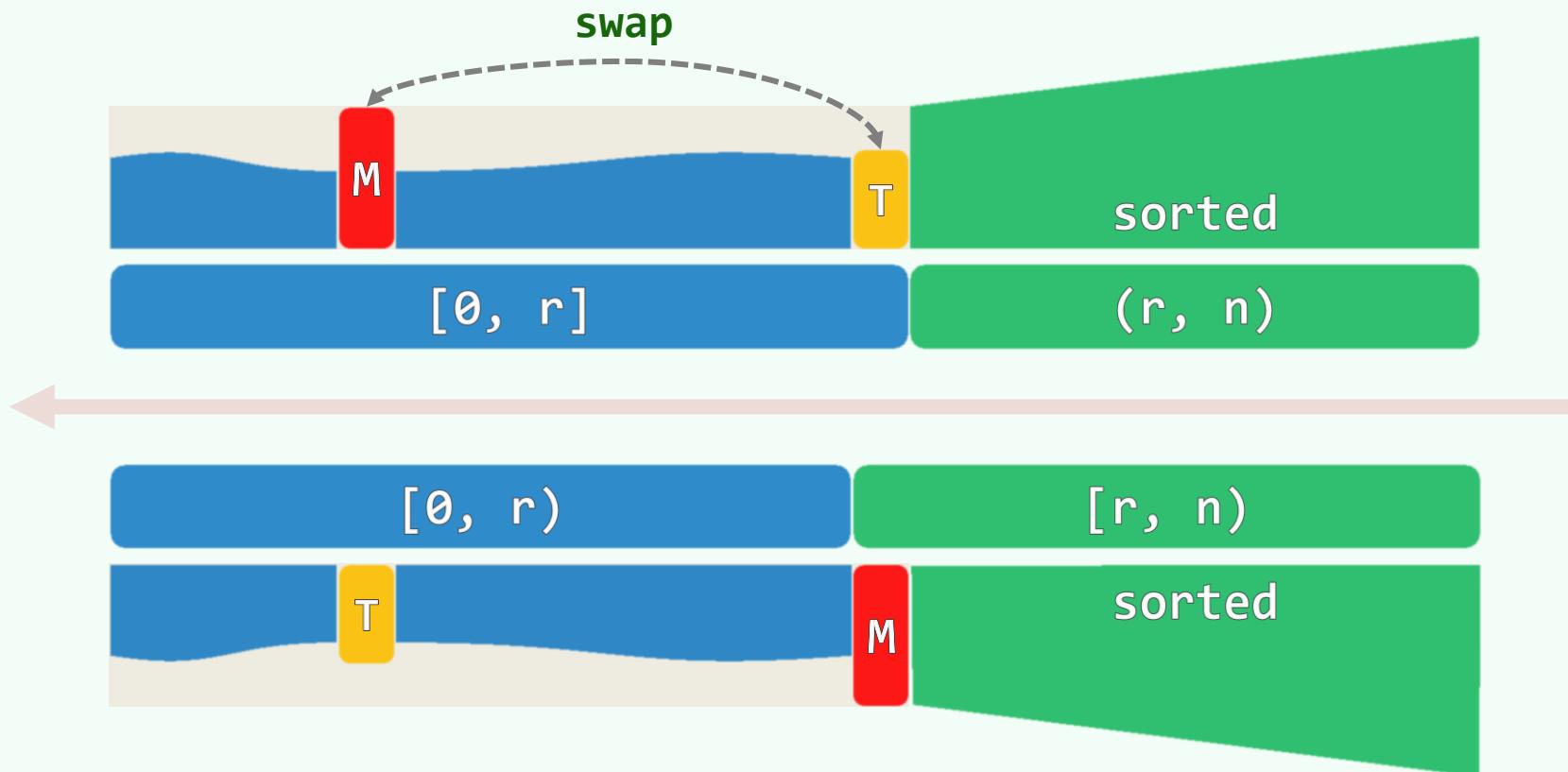
- 采用交换法，每迭代一步，M都会脱离原属的循环节，自成一个循环节



- 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

多余的交换

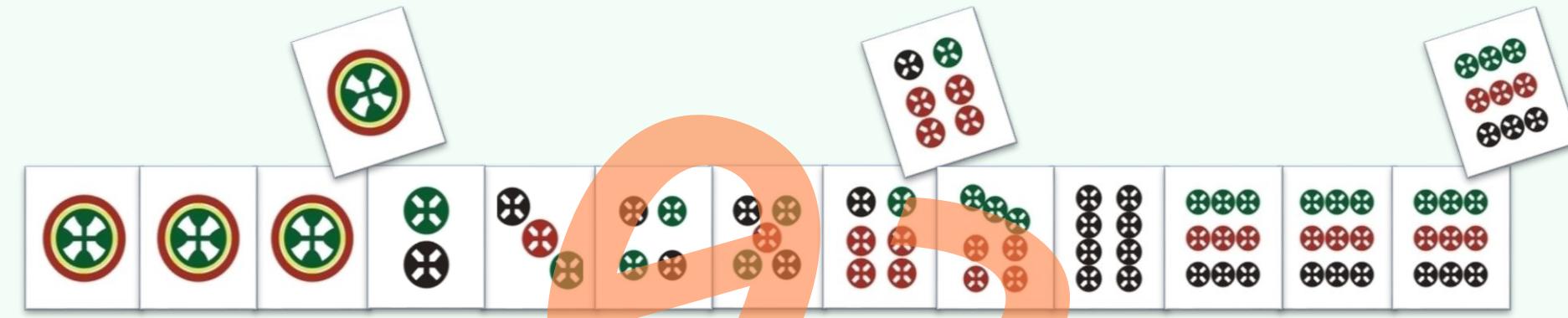
- ❖ 若M已经就位，则实际上**无需交换** —— 这种情况会出现几次？



- ❖ 最初有 c 个循环节，**总体**就出现 c 次 —— 最大值为 n ，期望 $\Theta(\log n)$...

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

一日与樊、饶、章、蒋太太等看竹甘余周，余又负十余元。后又看众人打poker，觉无意味，二点始睡

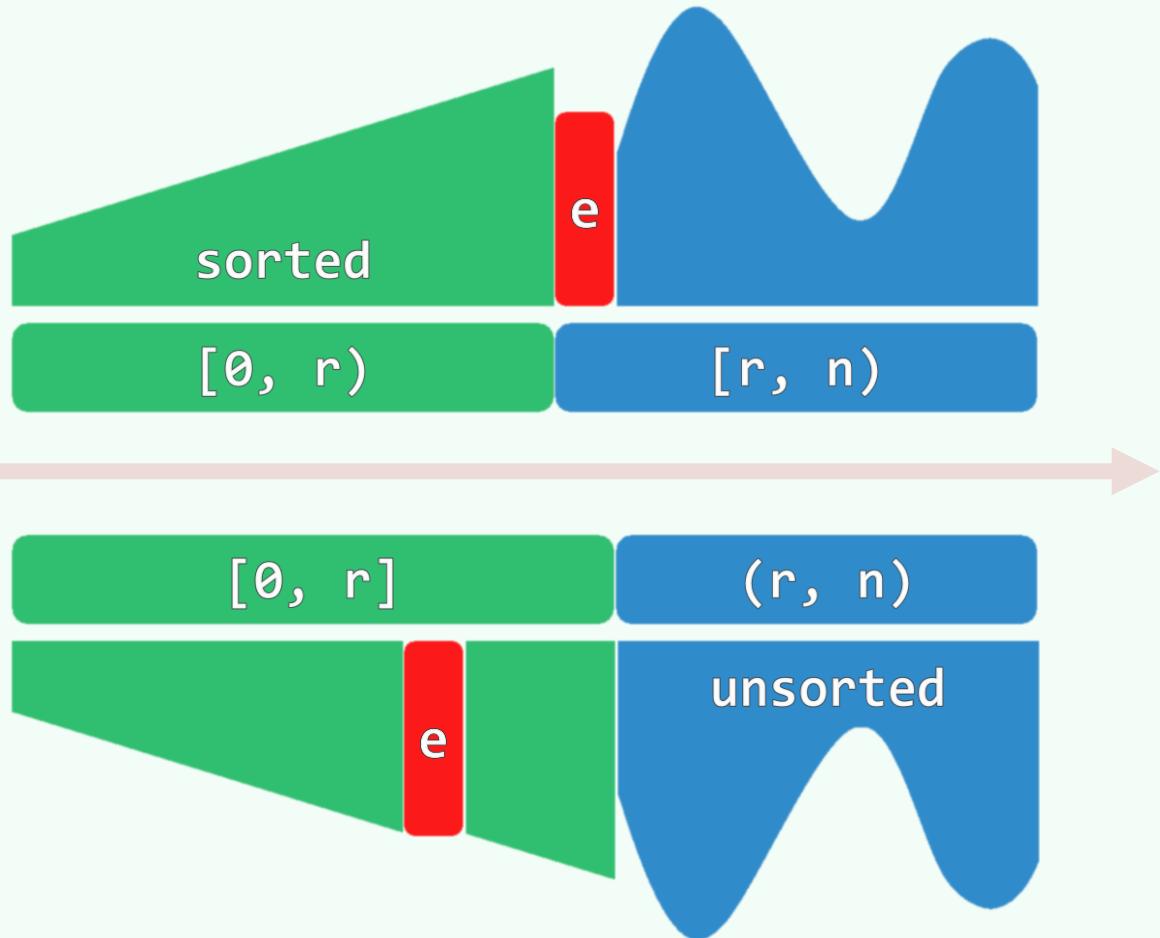
人生如此美好，慢慢走，欣赏啊！

G

邓俊辉

deng@tsinghua.edu.cn

减而治之



❖ 始终将序列视作两部分：

- 前缀 $s[0, r)$: 有序
- 后缀 $u[r, n)$: 待排序

❖ 初始化: $|s| = r = 0$

❖ 反复地, 针对 $e = A[r]$

- 在 s 中 **查找适当位置**
- **插入** e
- $r++$

实例

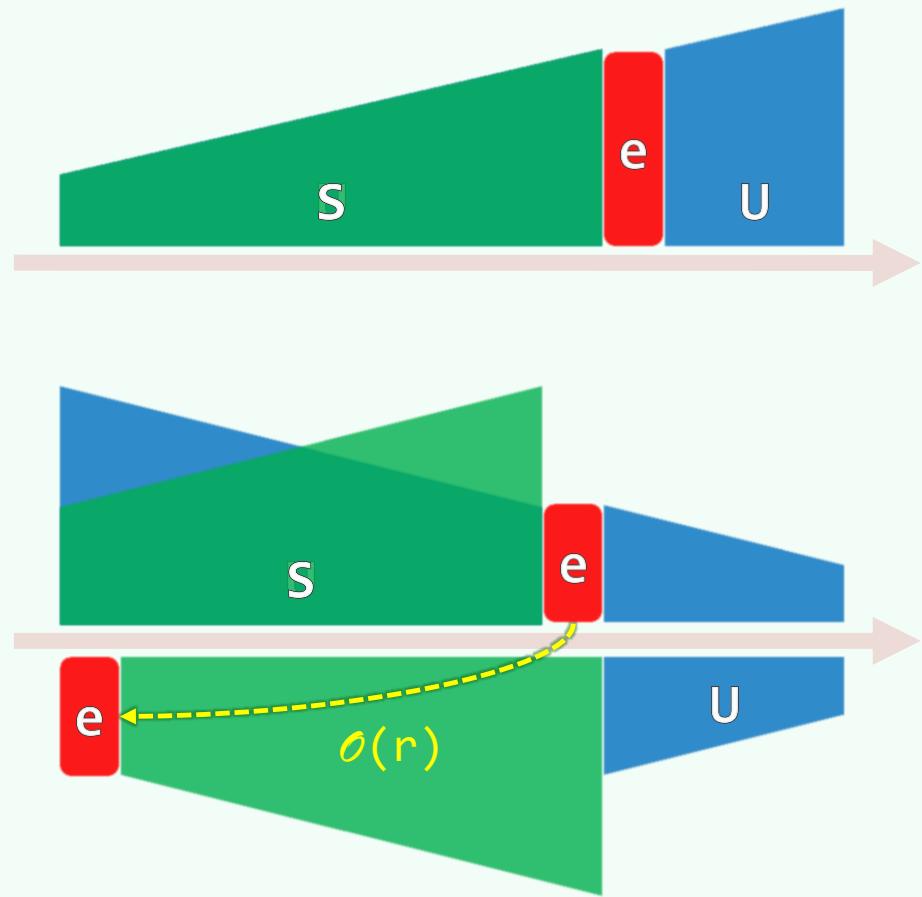
								5	2	7	4	6	3	1	
iteration	sorted prefix							e	random suffix						
0	^							5	2	7	4	6	3	1	
1	5							2	7	4	6	3	1		
2	2 5							7	4	6	3	1			
3	2 5 7							4	6	3	1				
4	2 4 5 7							6	3	1					
5	2 4 5 6 7							3	1						
6	2 3 4 5 6 7							1							
7	1 2 3 4 5 6 7														

实现

```
template <typename T> void List<T>::insertionSort( ListNodePosi<T> p, Rank n ) {  
    for ( Rank r = 0; r < n; r++ ) { //逐一引入各节点, 由sr得到sr+1  
        insert( 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
```

- ❖ 得益于此前约定的search()接口语义, 前缀的确总是**保持有序**, 而且**稳定**
- ❖ 验证以下情况, 体会**哨兵**的作用: 前缀中有元素与p相等; p在前缀中**最小/最大**; 前缀**为空**; ...

最好 & 最坏



❖ **最好:** $\mathcal{O}(n)$ //比如，已经有序，相当于验证

❖ **最坏:** $\mathcal{O}(n^2)$ //比如，完全逆序

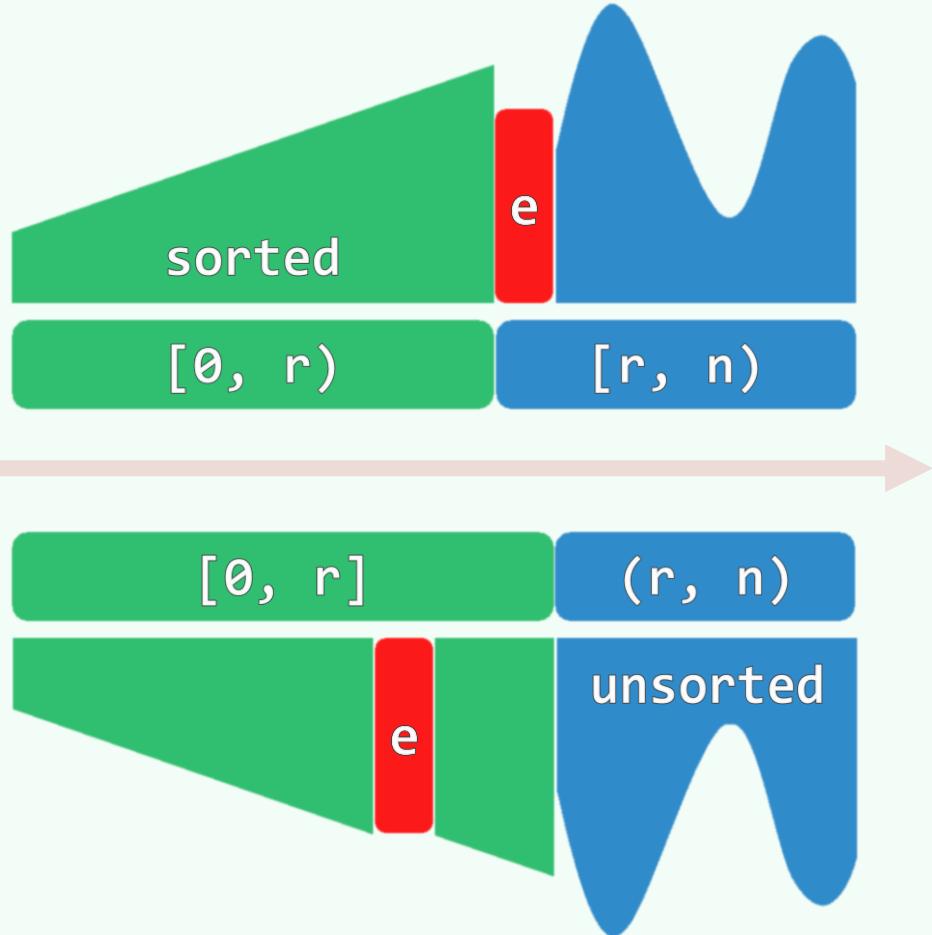
❖ **有实质的差异!**

Selectionsort呢?

❖ 主要消耗来自查找

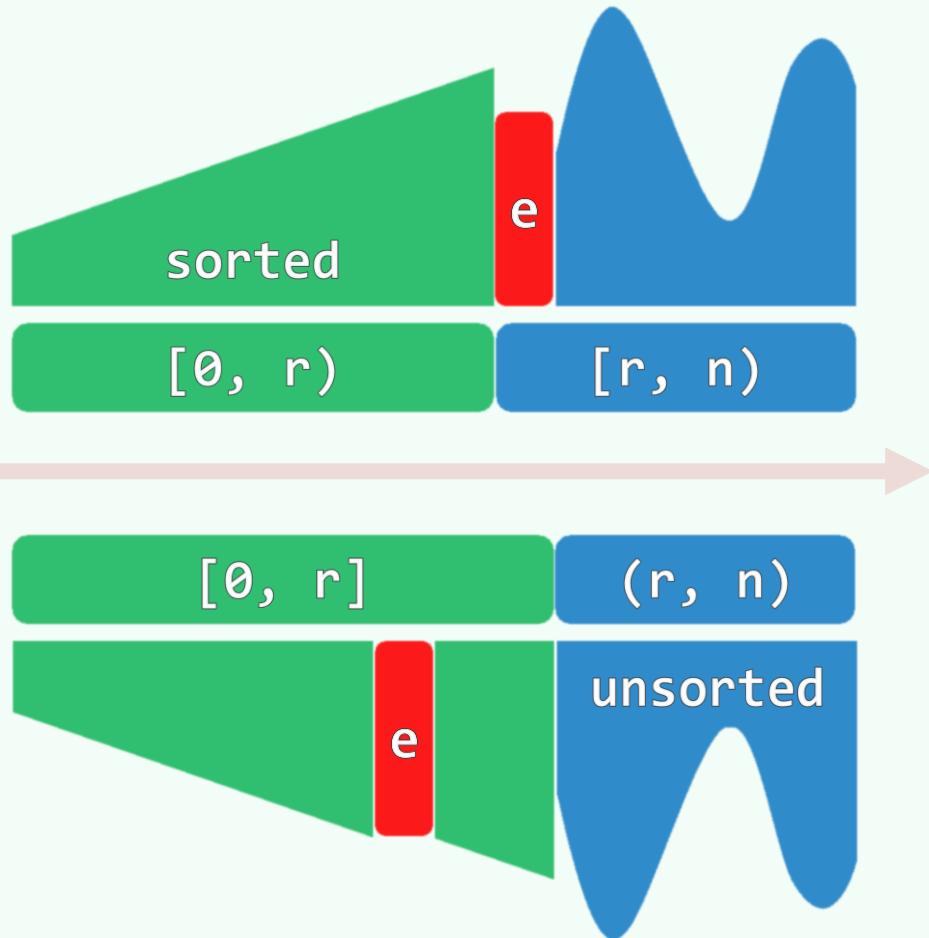
为何不...改用**binSearch()**?

平均 ~ 期望



- ❖ 若知道插入每个元素所需的**期望时间**其**总和**即**时总体的期望成本**
 - ❖ 后向分析：当前前缀 $[0, r]$ 中，谁是**最后插入的**？
 - ❖ 无非 $r+1$ 种情况，且概率均等
- 数学期望： $1 + \sum_{k=0}^r k/(r+1) = \underline{1 + r/2}$
- ❖ 故总和为： $\sum_{r=0}^{n-1} \underline{(1 + r/2)} = \mathcal{O}(n^2)$
 - ❖ 有的元素可能**无需移动**，特判并**节省**？

输入敏感



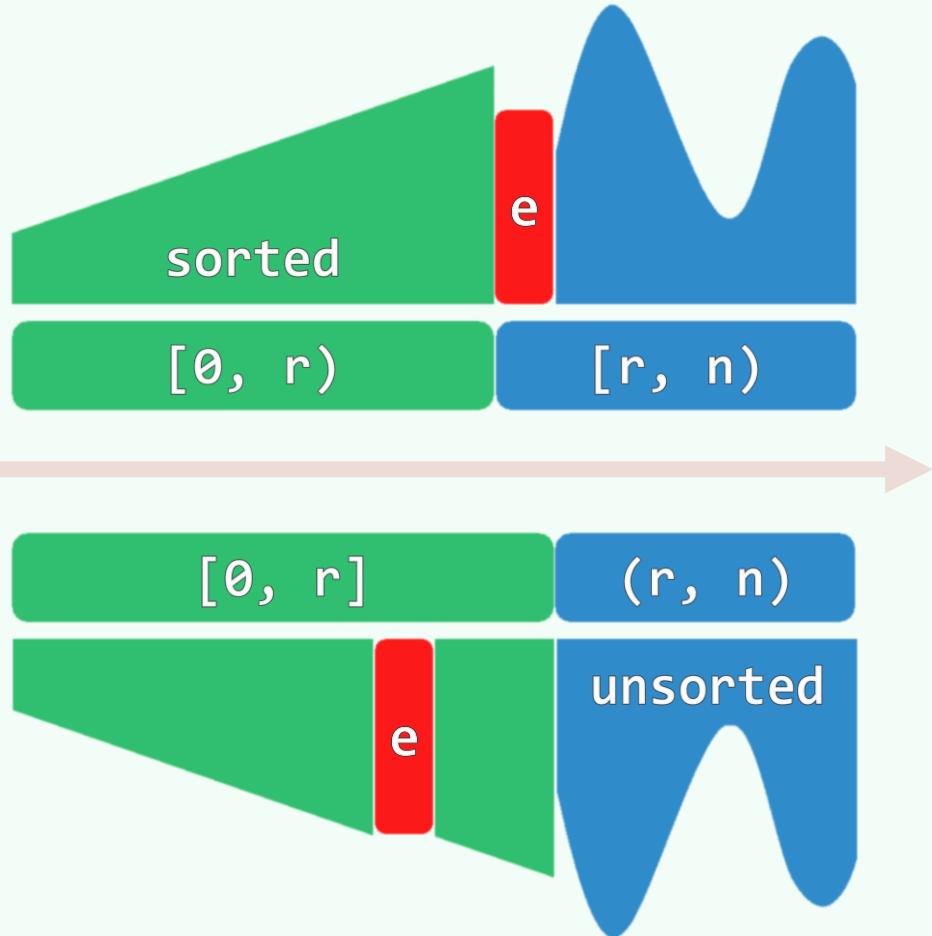
❖ 从 $\mathcal{O}(n)$ 到 $\mathcal{O}(n^2)$ ，中间情况如何？

❖ 有序/乱序的程度

可简明度量，而且
与时间成本之成正比...

❖ 输入敏感性/input-sensitivity

// 量出制入，丰俭由人
(稍后详解)



- ❖ 日常牌戏中，为何你我他都用它？
- ❖ 改用其他算法...
发牌完毕，才能各自开始理牌
- ❖ Insertionsort
可以边发边理；发完即理好
- ❖ online：在数据完全就绪之前，即可开始计算！
// 何必一味追求结果，不妨充分享受过程

03-H

日两美其必合兮，孰信修而慕之
思九州之博大兮，岂惟是其有女

列表

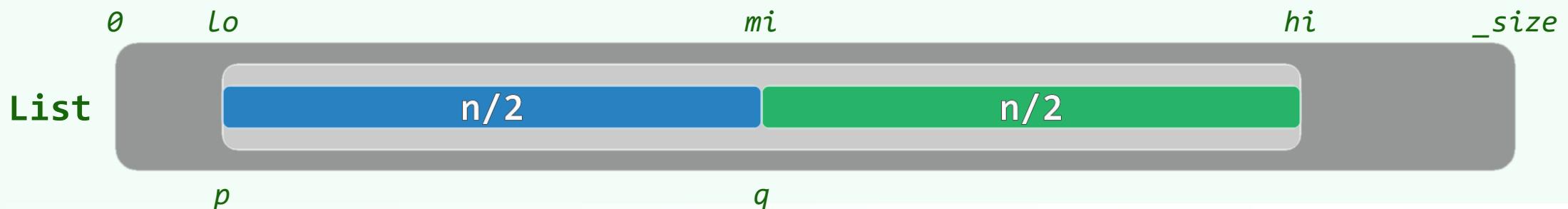
归并排序

邓俊辉

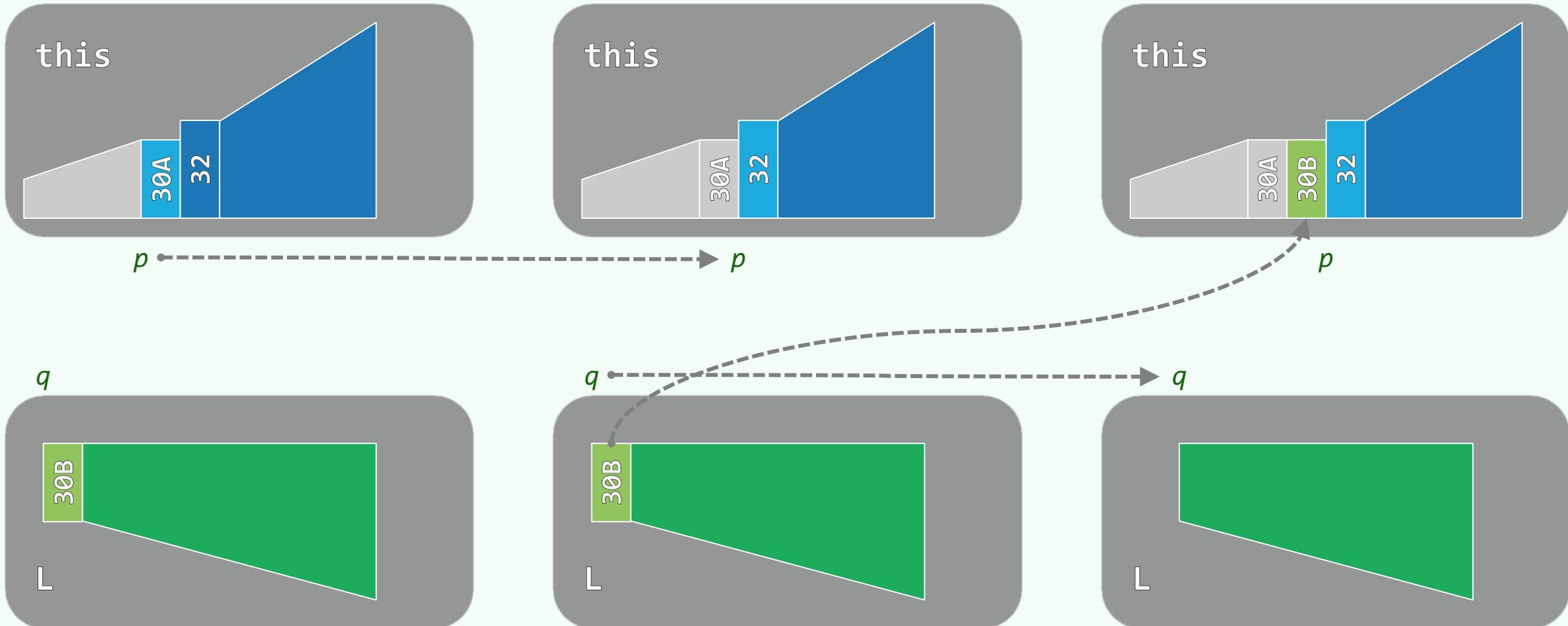
deng@tsinghua.edu.cn

主算法

```
template <typename T> void List<T>::mergeSort( ListNodePosi<T> & p, Rank n ) {  
    if ( n < 2 ) return; //待排序范围足够小时直接返回，否则...  
  
    ListNodePosi<T> q = p; Rank m = n >> 1; //以中点为界  
  
    for ( Rank 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  
  
    p = merge( p, m, *this, q, n - m ); //归并 $\quad$   
}  
//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



二路归并：实现

```
template <typename T> ListNodePosi<T> //this.[p +n) & L.[q +m): 归并排序时, L == this
List<T>::merge( ListNodePosi<T> p, Rank n, List<T>& L, ListNodePosi<T> q, Rank m ) {
    ListNodePosi<T> pp = p->pred; //归并之后p或不再指向首节点, 故需先记忆, 以便返回前更新
    while ( ( 0 < m ) && ( q != p ) ) //小者优先归入
        if ( (0 < n) && (p->data <= q->data) ) { p = p->succ; n--; } //p直接后移
        else { insert( L.remove( (q = q->succ)->pred ), p ); m--; } //q转至p之前
    return pp->succ; //更新的首节点
} //运行时间 $\mathcal{O}(n + m)$ , 线性正比于节点总数
```

列表

逆序对



有象斯有對，對必反其為
有反斯有讎，讎必和而解

邓俊辉

deng@tsinghua.edu.cn

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**

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

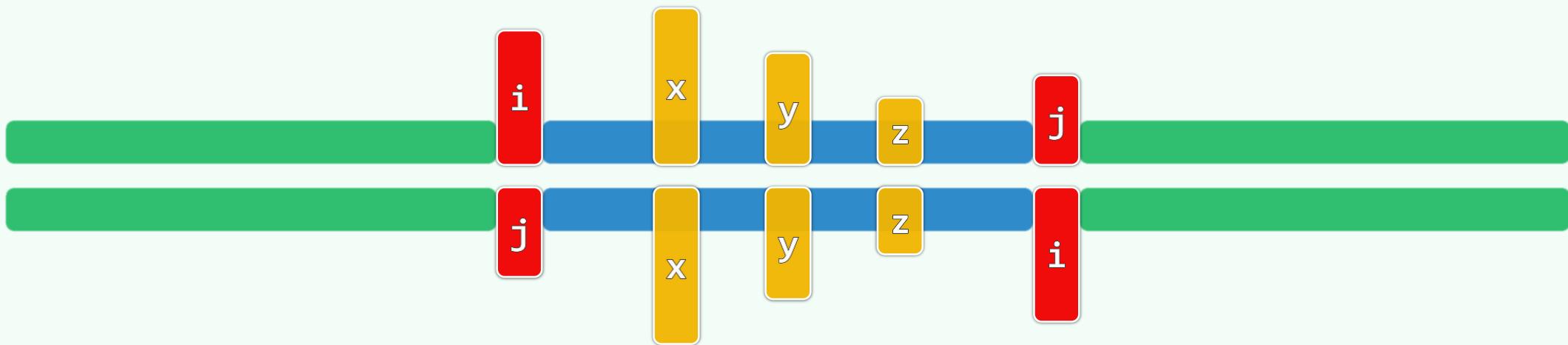
♦ 例: $A[] = \{ 5, 3, 1, 4, 2 \}$ 中, 共有 $0 + 1 + 2 + 1 + 3 = 7$ 个逆序对

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

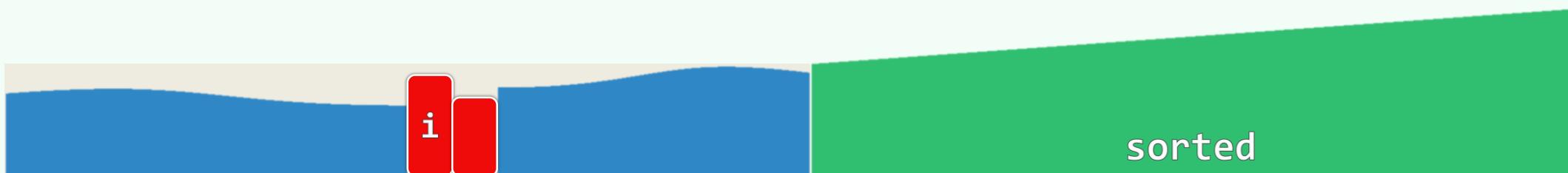
$A[] = \{ 5, 4, 3, 2, 1 \}$ 中, 共有 $0 + 1 + 2 + 3 + 4 = 10$ 个逆序对

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

Bubblesort



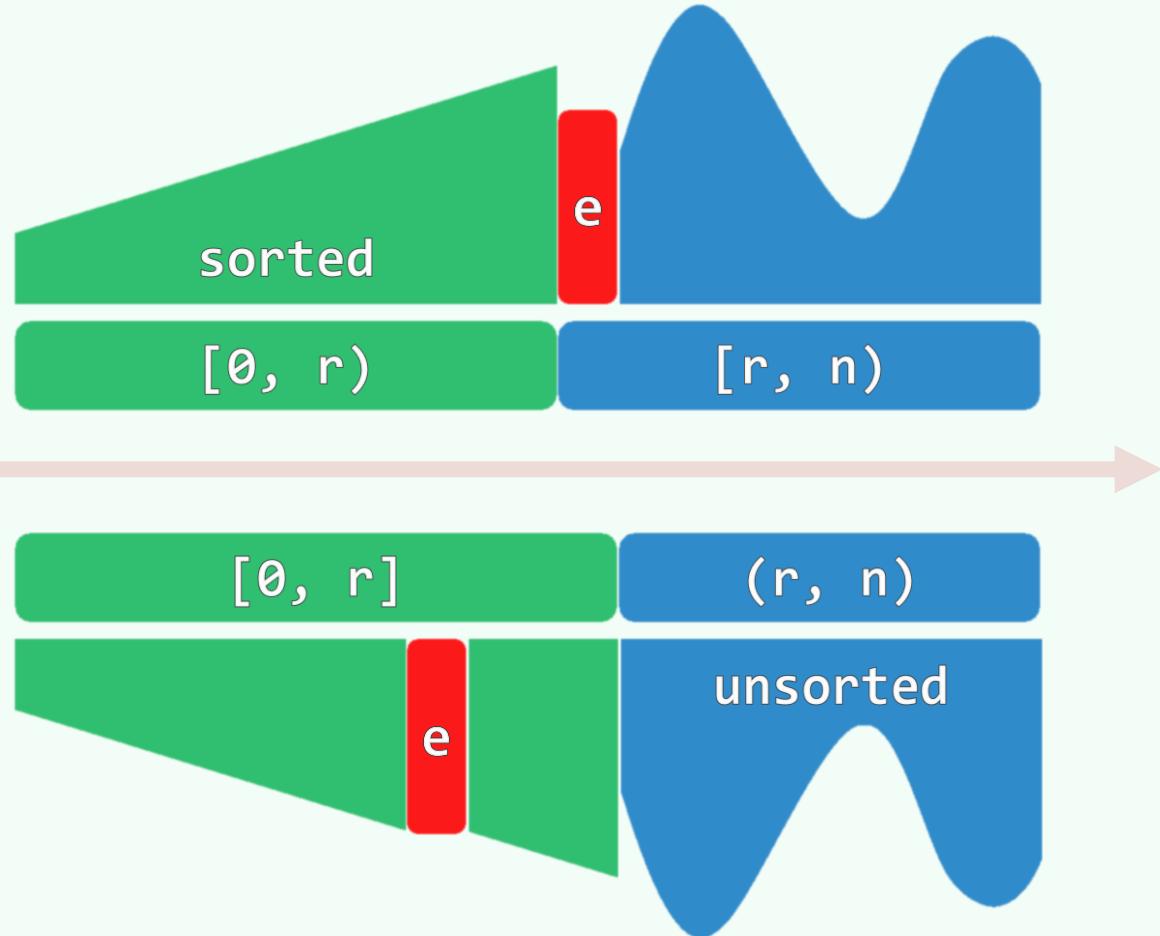
❖ 在序列中交换一对逆序元素，逆序对总数必然减少



❖ 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 $e = A[r]$ 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

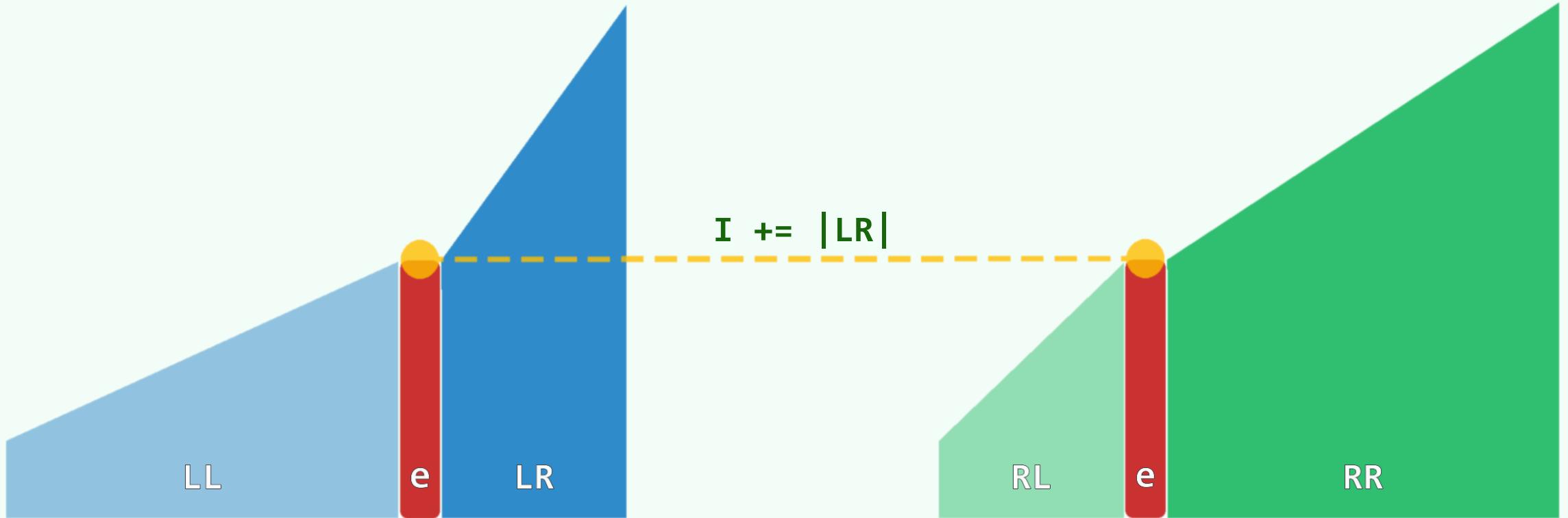
- 关键码比较次数为 $\mathcal{O}(\mathcal{I})$
- 运行时间为 $\mathcal{O}(n + \mathcal{I})$

//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；而借助归并排序，仅需 $\mathcal{O}(n \log n)$ 时间 . . .



e3 - J

列表

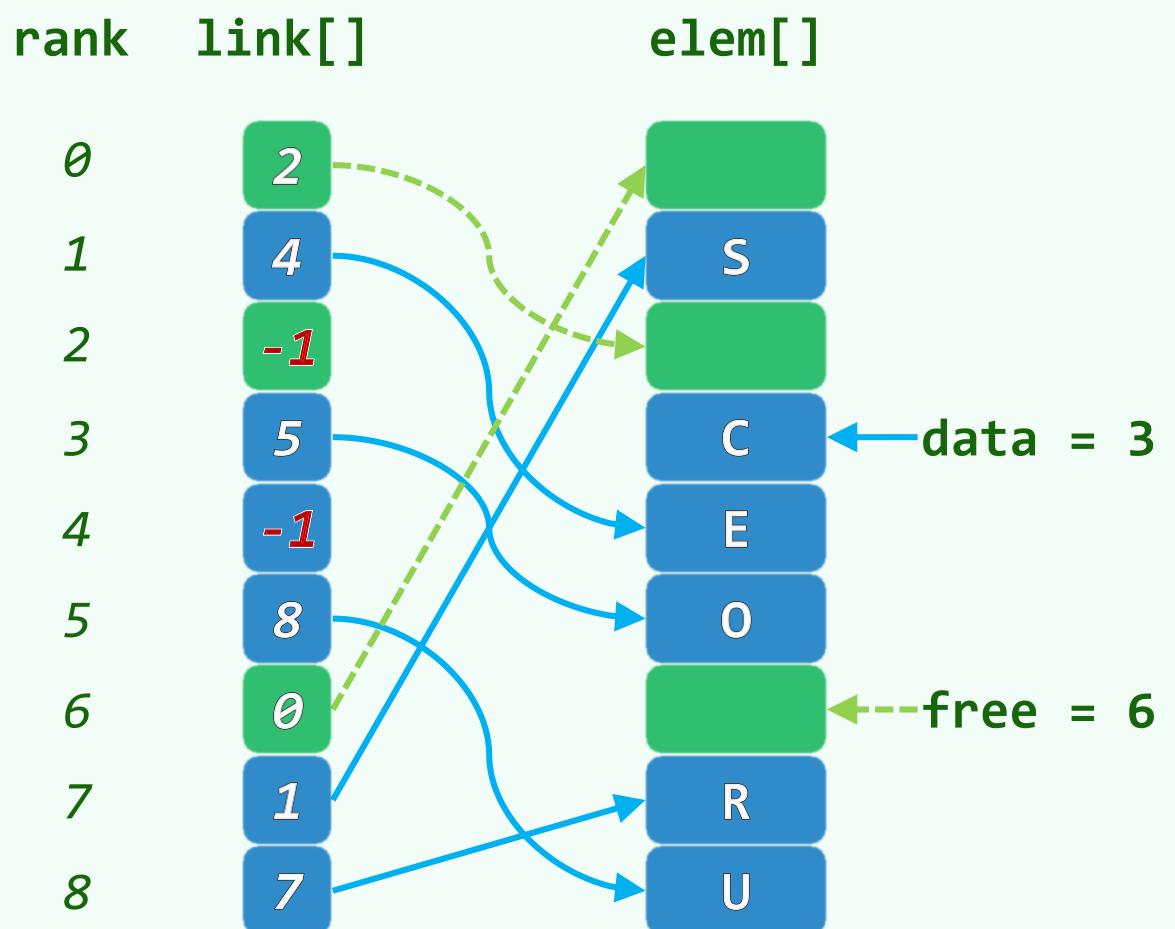
游标实现

邓俊辉

deng@tsinghua.edu.cn

动机与构思

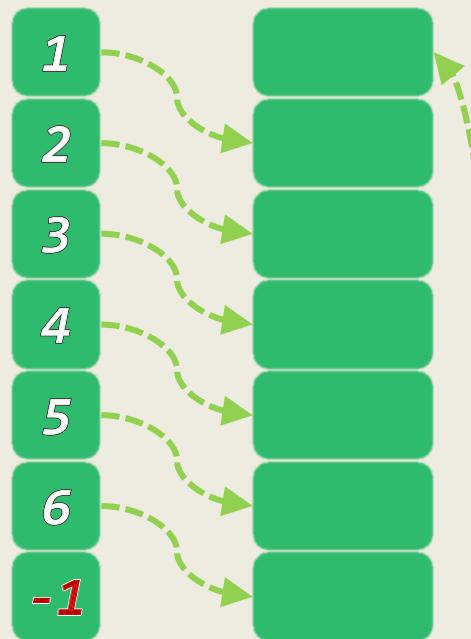
- ❖ 某些语言不支持指针类型，即便支持频繁的动态空间分配也影响总体效率
- ❖ 此时，又当如何实现列表结构呢？
- ❖ 利用线性数组，以游标方式模拟列表
 - elem[]：对外可见的数据项
 - link[]：数据项之间的引用
- ❖ 维护逻辑上互补的列表data和free
- ❖ 在插入或删除元素时，应如何调整？



实例 (1/4)

init(7)

link[] elem[]



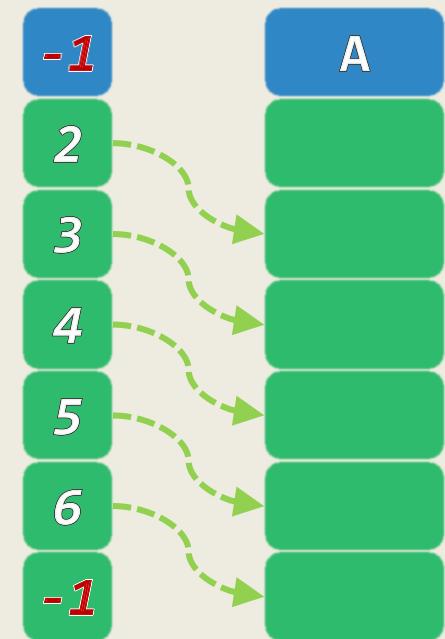
data = -1
free = 0

rank

0
1
2
3
4
5
6

insert('A')

link[] elem[]

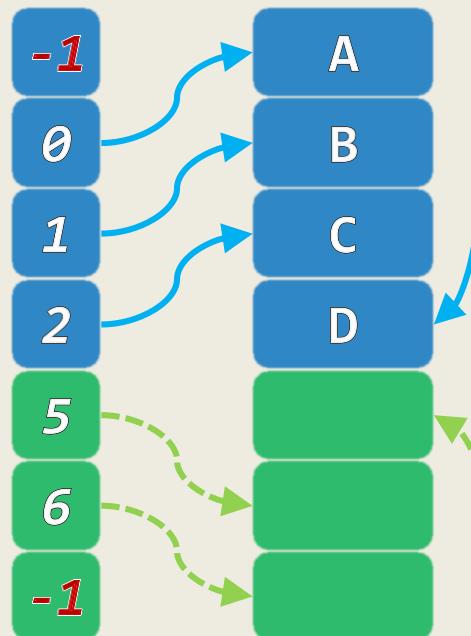


data = 0
free = 1

实例 (2/4)

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

`link[]` `elem[]`

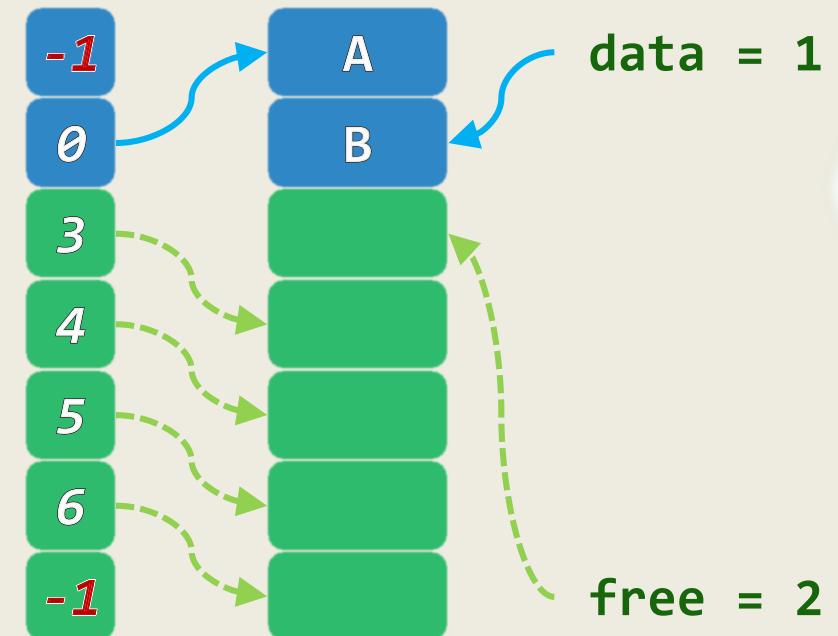


`insert('B')`

`rank`

0
1
2
3
4
5
6

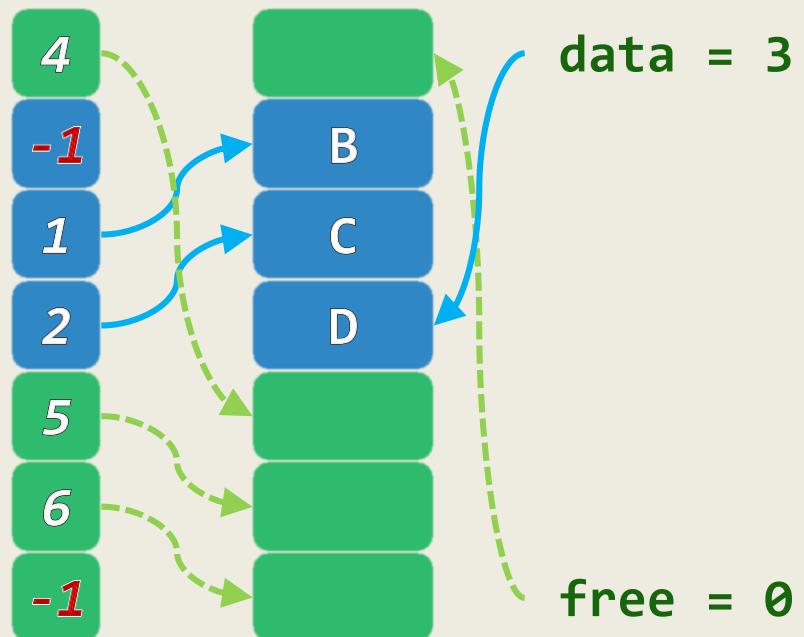
`link[]` `elem[]`



实例 (3/4)

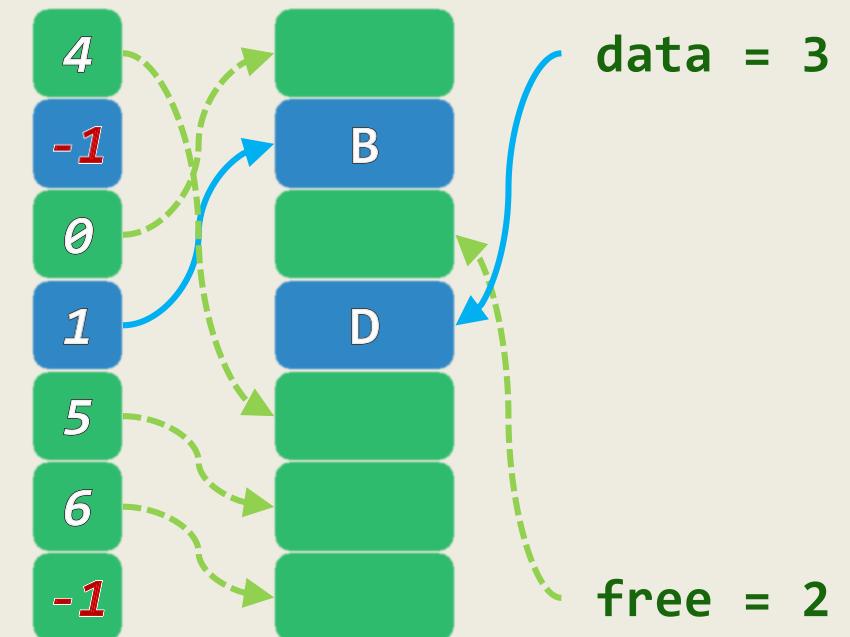
remove('A')

link[] elem[]



remove('C')

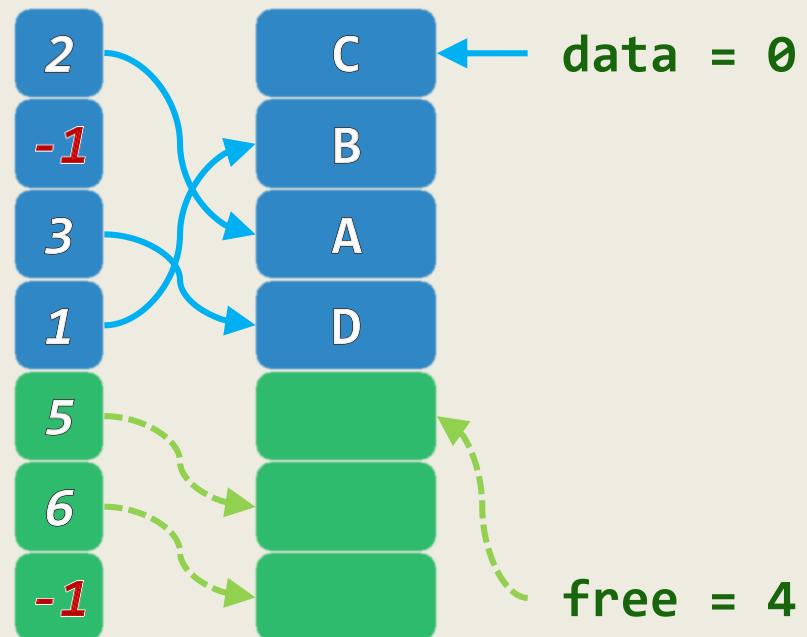
link[] elem[]



实例 (4/4)

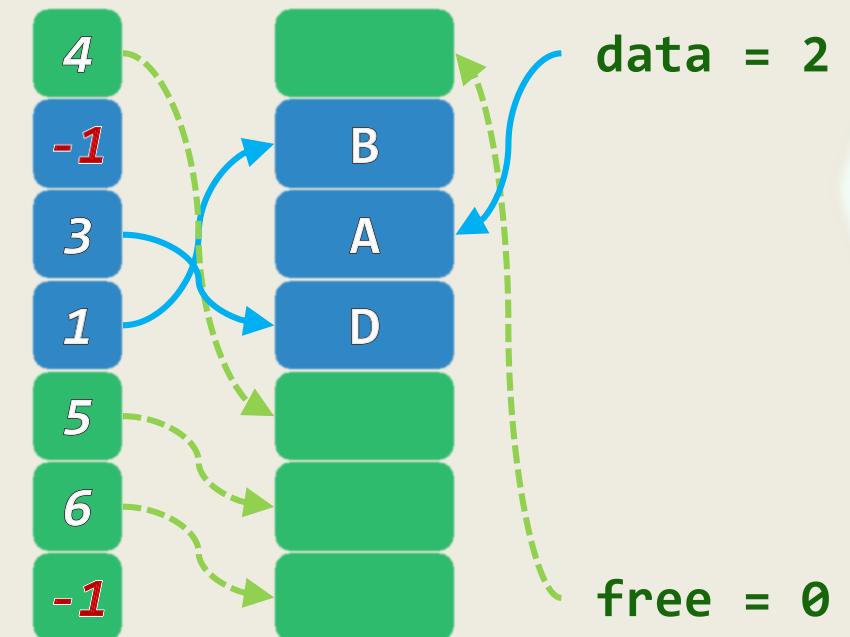
`insert('C')`

`link[]` `elem[]`



`insert('A')`

`link[]` `elem[]`



03 - XA

One of the things that Java is good at is giving you this
homogeneous view of a reality that's usually very heterogeneous.

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; } //接口实现  
    public boolean inside( Point p ) { //接口实现  
        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

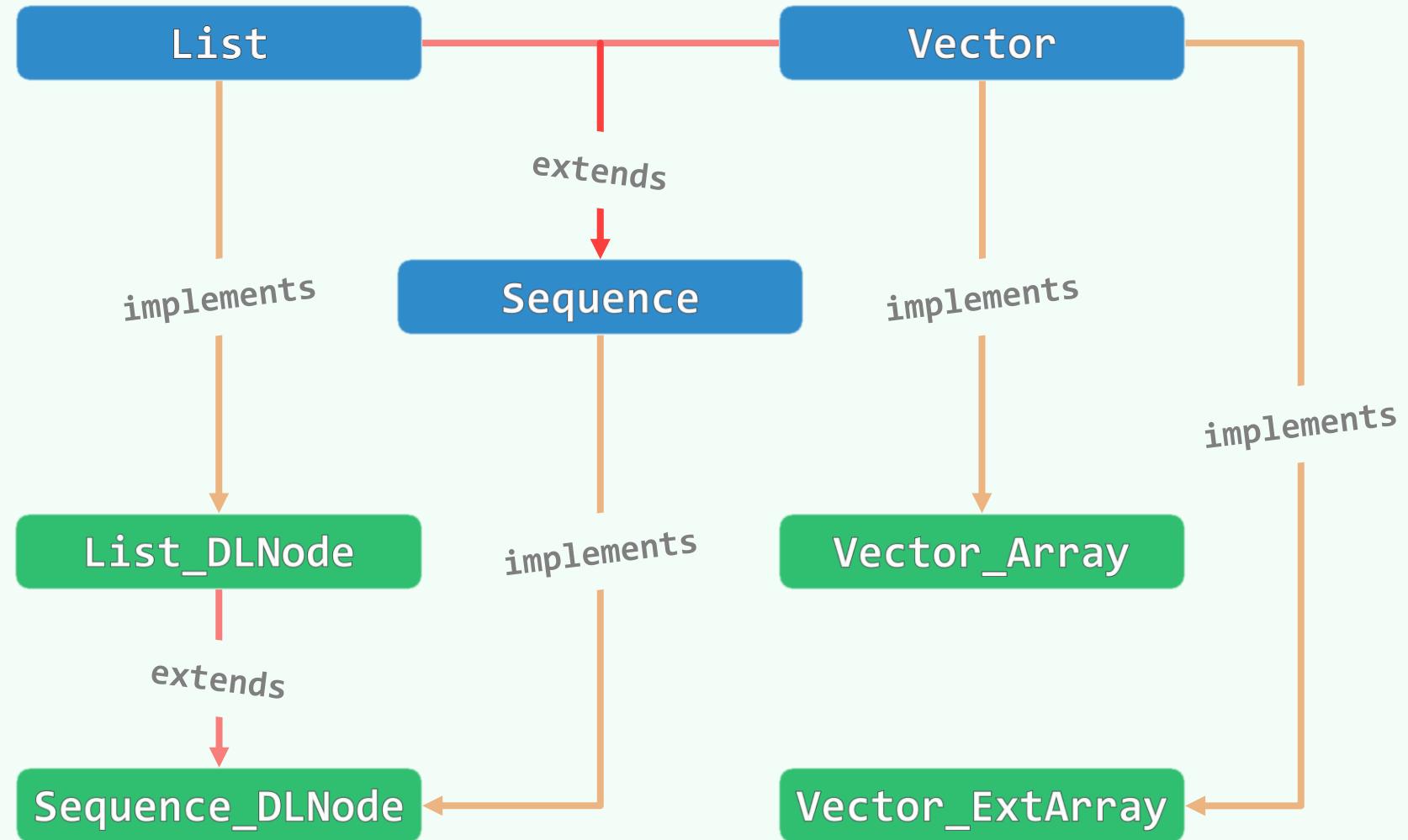
```
{ /* ... */ }
```

class Sequence_DLNode

extends List_DLNode

implements Sequence

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



03 - KB

列表

Python列表

邓俊辉

deng@tsinghua.edu.cn

PYTHON = Programmers Yearning To Homestead Our Noosphere.

声明 + 倒置 + 排序

❖ 在Python中，List属于内置的标准数据类型

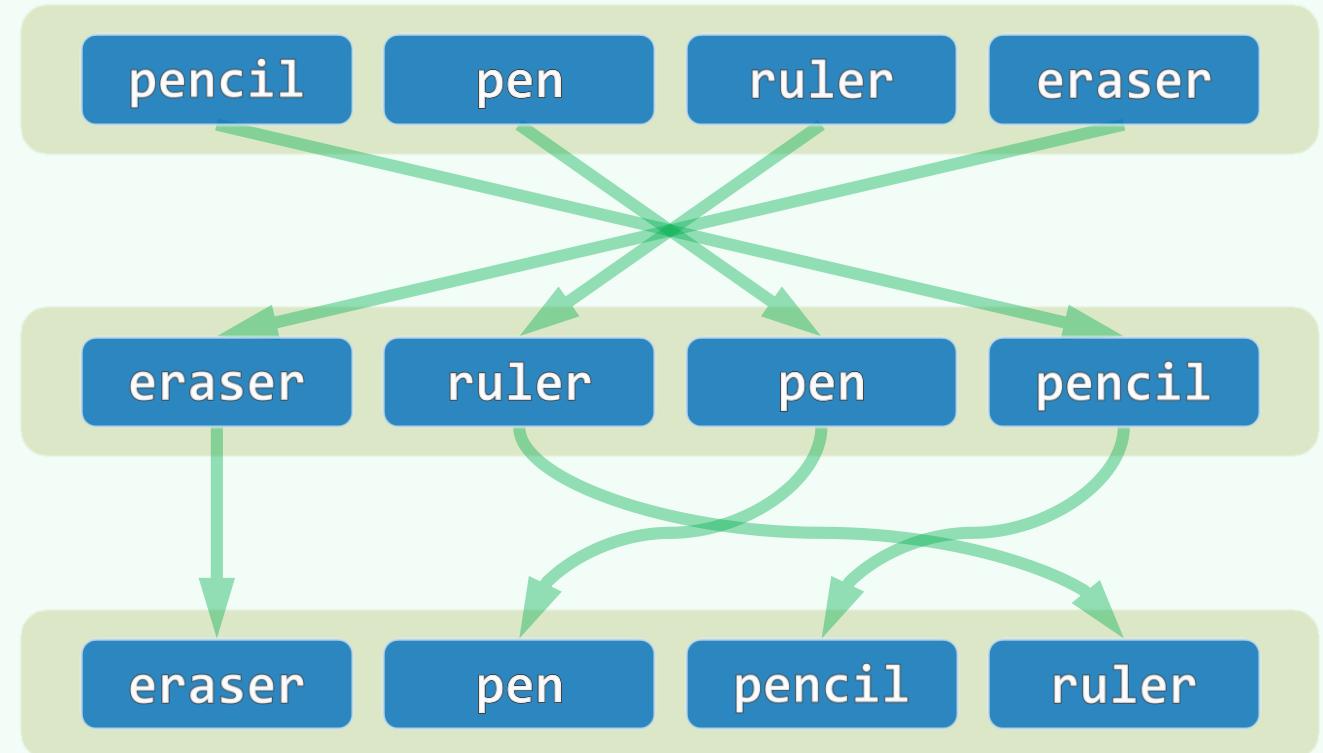
❖ `box = ['pencil', 'pen', 'ruler', 'eraser']; print(box)`

```
# ['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),`
eraser pen pencil ruler



区间遍历

```
❖ 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

集合遍历

❖ `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
```

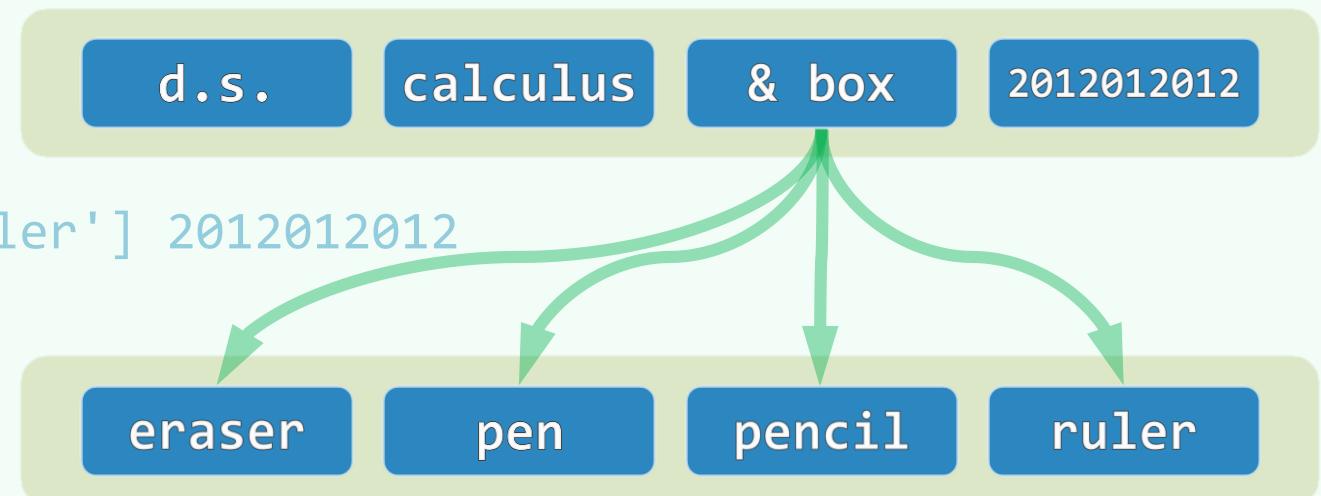
```
['eraser', 'pen', 'pencil', 'ruler'] 2012012012
```

❖ `for item in bag[2]: print(item),`

```
# eraser pen pencil ruler
```

❖ `for item in bag[2][1:3]: print(item),`

```
# pen pencil
```



reverse(): 秩 + 位置

```
def reverse_1(L): # 循秩访问?
```

```
    lo, hi = 0, len(L) - 1
```

```
    while lo < hi:
```

```
        L[lo], L[hi] = L[hi], L[lo]
```

```
        lo, hi = lo + 1, hi - 1
```

```
    return L
```

```
def reverse_2(L): # 循位置访问?
```

```
    for i in range(len(L)):
```

```
        L.insert(i, L.pop())
```

```
    return L
```