算法与数据结构

3. 列表

赵登阳

中国计量大学信息工程学院

2024 年 9 月 23 日

中國計量大學
CHINA JILIANG UNIVERSITY

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）

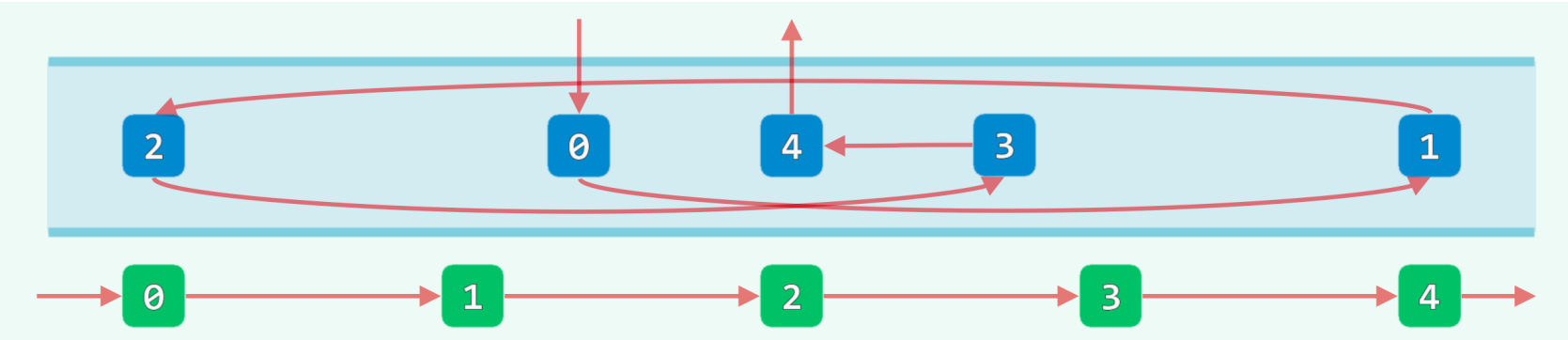
从静态到动态

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

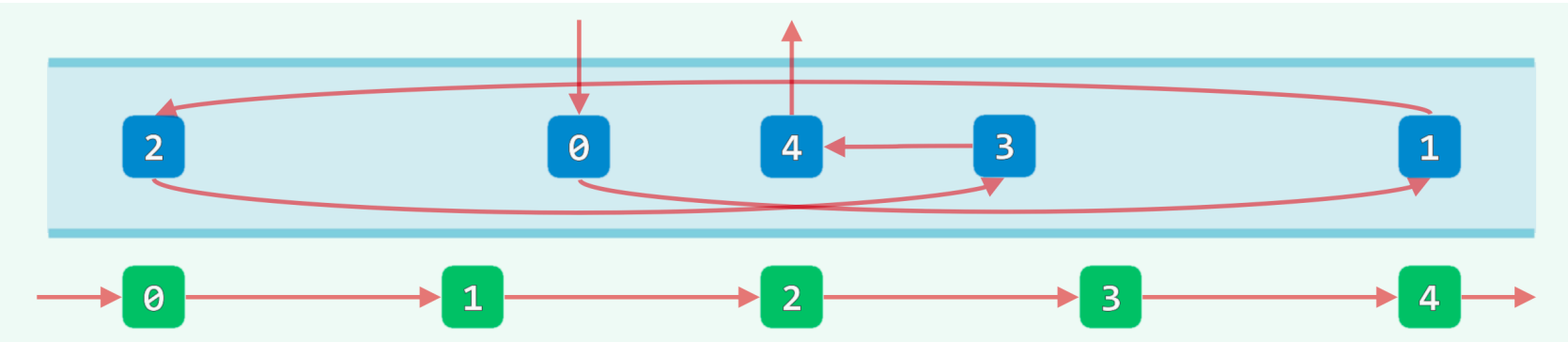
列表

- 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：$\mathrm{L} = \{ a_0, a_1, \ldots, a_{n-1} \}$
- 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



由秩到位置

- 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 $\mathcal{O}(1)$



列表节点：ADT 接口

- 列车 - 车厢 - 货物
- list - node - data

操作接口	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

列表节点：模版类

```
1  typedef int Rank;
2  #define ListNodePosi(T) ListNode<T>*;
3  template <typename T> struct ListNode {
4      T data;
5      ListNodePosi(T) pred;
6      ListNodePosi(T) succ;
7      // 构造函数
8      ListNode() {}
9      ListNode(T e, ListNodePosi(T) p=NULL,
10                 ListNodePosi(T) s=NULL)
11         : data(e), pred(p), succ(s) {}
12     // 操作接口
13     ListNodePosi(T) insertAsPred(T const& e);
14     ListNodePosi(T) insertAsSucc(T const& e);
15 }
```

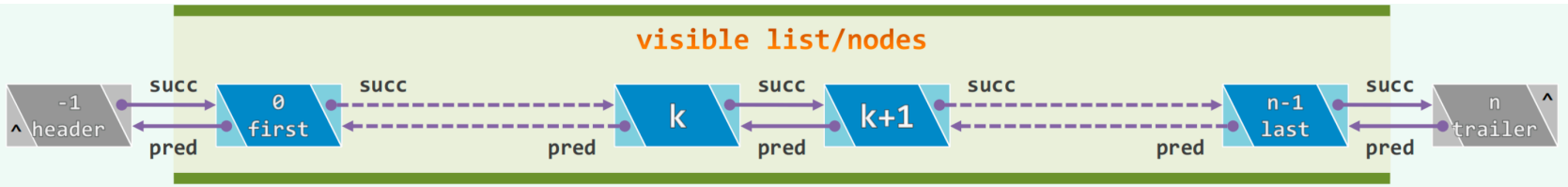
列表：ADT 接口

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

列表模版类

```
1  #include "listNode.h" //引入列表节点类
2  template <typename T> class List { //列表模板类
3  private:
4      Rank _size;
5      ListNodePosi<T> header, trailer; //哨兵
6      // 头、首、末、尾节点的秩
7      // 可分别理解为 -1、0、n-1、n
8  protected:
9      /* ... 内部函数 */
10 public:
11     /* ... 构造函数、析构函数 */
12     /* ... 只读接口、可写接口、遍历接口 */
13 };
```

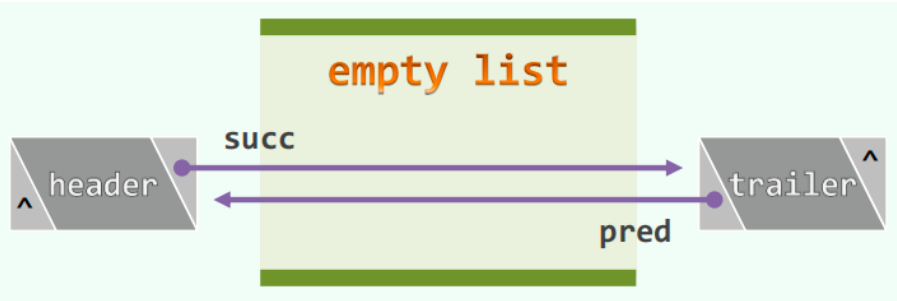
头、尾节点



- 哨兵节点：header、trailer
- 从外部被等效地视作 NULL
- 设置哨兵节点后，对于从外部可见的任一节点而言，其前驱和后继在内部都必然存在，可以简化算法的描述与实现
- 哨兵的引入，使得相关算法不必再对各种边界退化情况做专门的处理，从而避免出错的可能

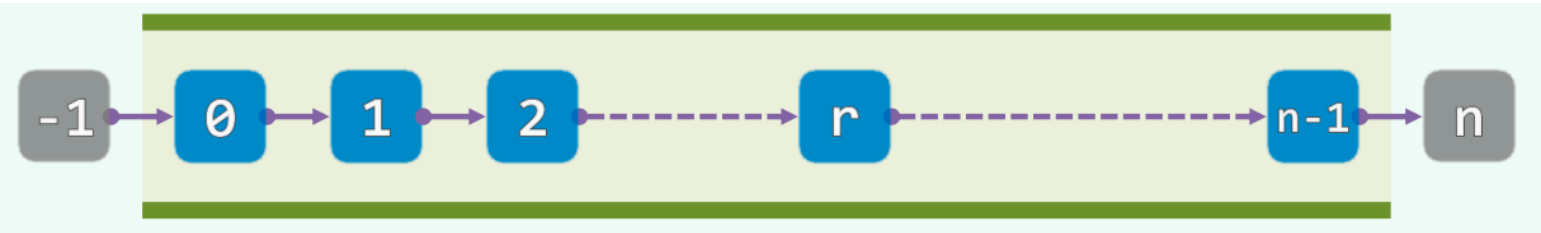
默认构造方法

```
1  template <typename T>
2  void List<T>::init() {
3      header = new ListNode<T>;
4      trailer = new ListNode<T>;
5      header->succ = trailer;
6      header->pred = NULL;
7      trailer->pred = header;
8      trailer->succ = NULL;
9      _size = 0;
10 }
```



由秩到位置的转换

```
1 // /O(r)效率，虽方便，勿多用
2 template <typename T> ListNodePosi<T> List<T>::
3 operator[]( Rank r ) const { //0 <= r < size
4     ListNodePosi<T> p = first(); //从首节点出发
5     while ( 0 < r-- )
6         p = p->succ; //顺数第r个节点即是
7     return p; //目标节点
8 } //秩 == 前驱的总数
```



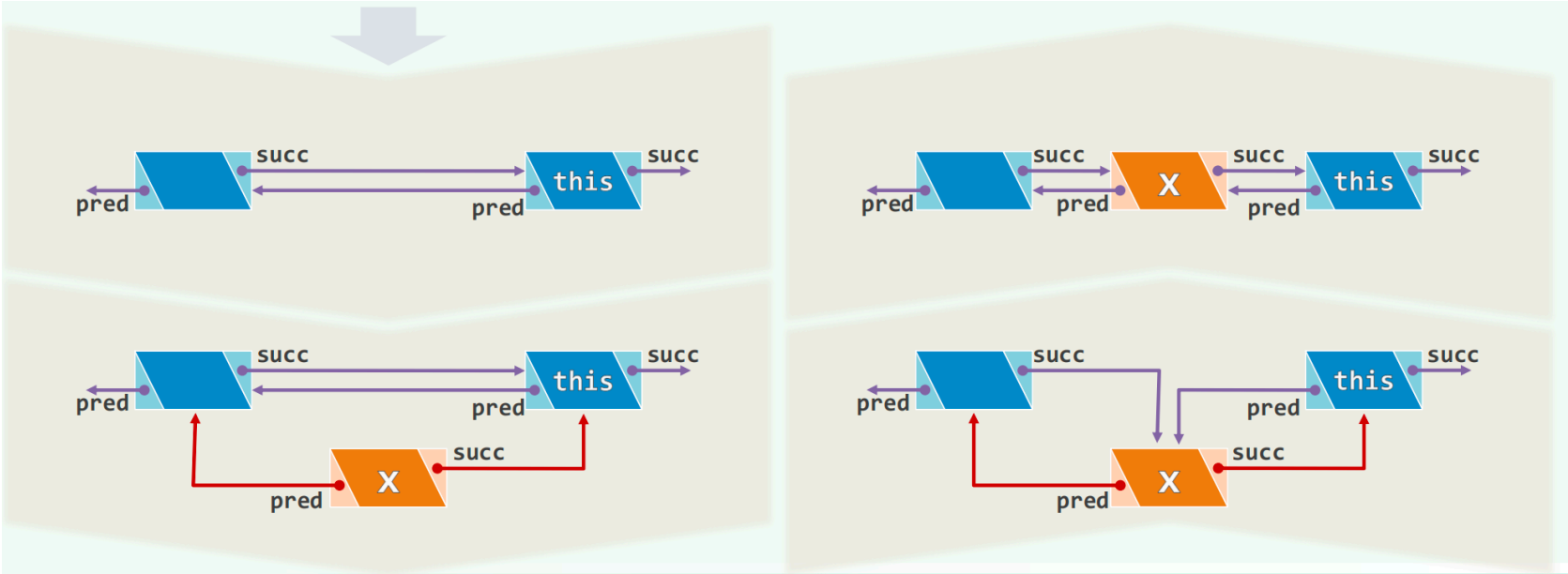
查找

- 算法思路和过程，与无序向量的顺序查找算法 Vector::find() 相仿
- 时间复杂度 $\mathcal{O}(n)$，线性正比于查找区间的宽度

```
1  template <typename T> ListNodePosi<T> List<T>::
2  find(T const& e, Rank n, ListNodePosi<T> p) const
3  {
4      while ( 0 < n-- ) //自后向前
5          //逐个比对（假定类型T已重载“==”）
6          if ( e == ( p = p->pred ) ->data )
7              //在p的n个前驱中，等于e的最靠后者
8              return p;
9      return NULL; //失败
10 } //O(n)
```

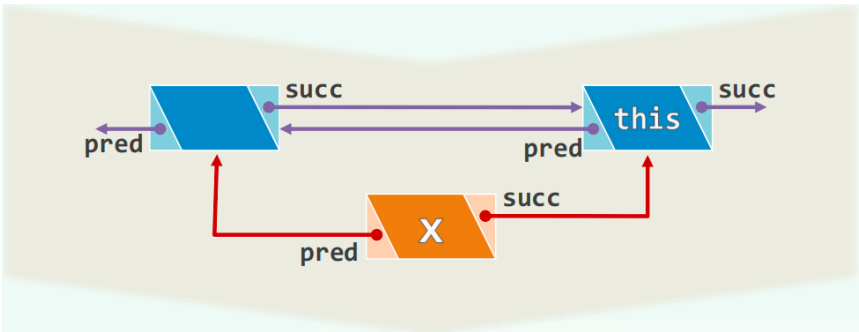
插入

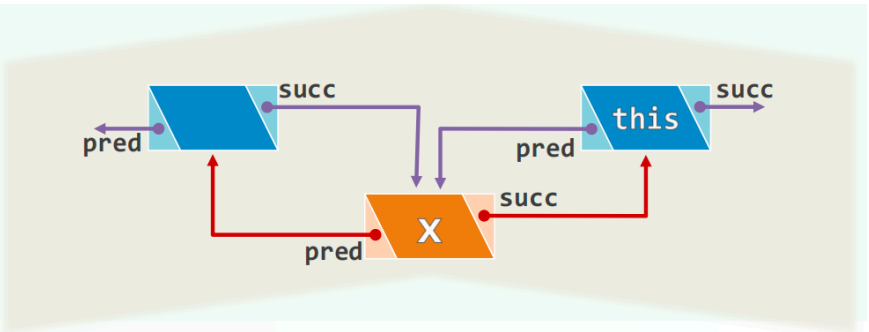
```
1 //e当作p的前驱插入
2 template <typename T> ListNodePosi<T> List<T>::
3 insert(T const & e, ListNodePosi<T> p)
4 { _size++; return p->insertAsPred( e ); }
```



插入

```
1  //前插入算法（后插入算法完全对称）
2  template <typename T> ListNodePosi<T>
3  ListNode<T>::insertAsPred( T const & e )
4  { //O(1)
5      ListNodePosi<T> x = new ListNode(e,pred,this);
6      pred->succ = x; pred = x; //次序不可颠倒
7      return x; //建立链接，返回新节点的位置
8  } // 得益于哨兵，即便this为首节点亦不必特殊处理
9    // 此时等效于insertAsFirst(e)
```





基于复制的构造

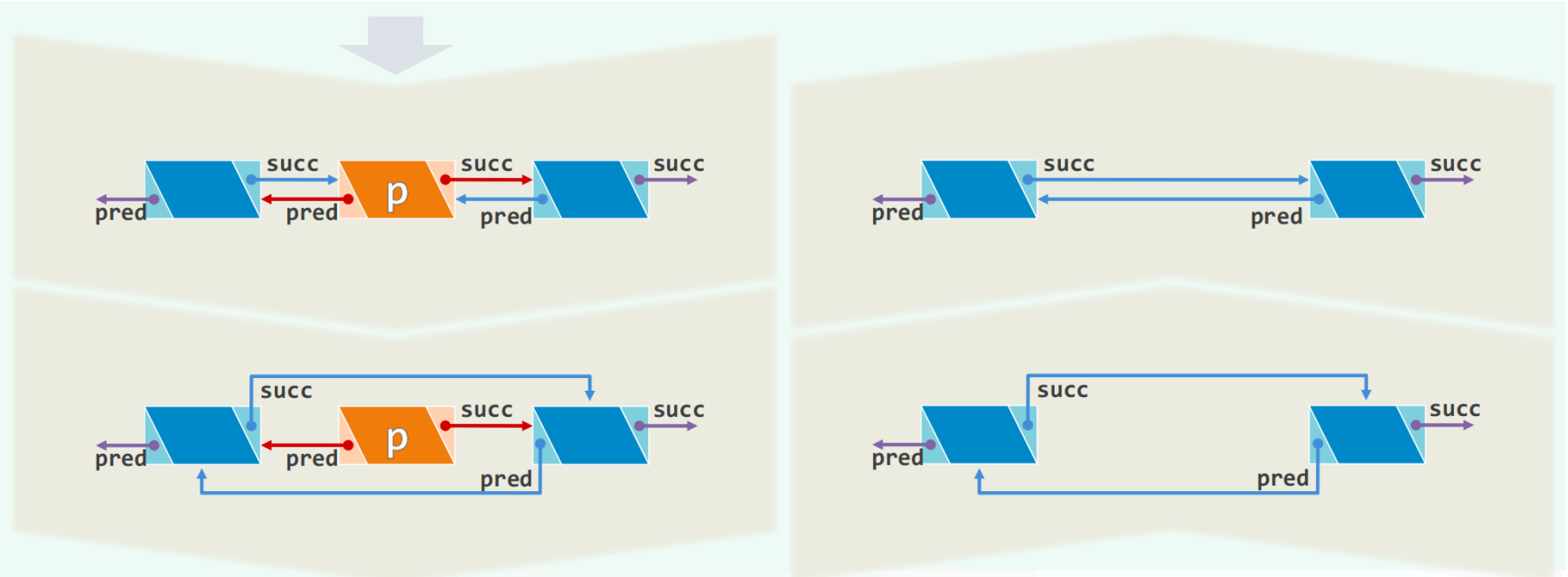
```
1  template <typename T> void List<T>::
2  copyNodes( ListNodePosi<T> p, Rank n ) { //O(n)
3      init(); //创建头、尾哨兵节点并做初始化
4      while ( n-- )
5      { //将起自p的n项依次作为末节点
6          insertAsLast( p->data ); //插入
7          p = p->succ;
8      }
9  }
```

基于复制的构造

```
1  template <typename T> //复制列表中自位置p起的n项
2  List<T>::List(ListNodePosi<T> p, Rank n)
3  { copyNodes(p, n); }
4
5  template <typename T> //整体复制列表L
6  List<T>::List(List<T> const& L)
7  { copyNodes(L.first(), L._size); }
8
9  template <typename T> //复制L中自第r项起的n项
10 List<T>::List(List<T> const& L, int n, int r)
11 { copyNodes(L[r], n); }
```

删除

思路 + 过程


succ
pred
p
succ
pred
succ
succ
pred
succ
pred
succ
succ
pred
p
succ
pred
succ
succ
pred
succ
pred

删除

```
1  //删除合法位置p处节点，返回其数值
2  template <typename T>
3  T List<T>::remove( ListNodePosi<T> p )
4  { //O(1)
5      //备份待删除节点数值（设类型T可直接赋值）
6      T e = p->data;
7      p->pred->succ = p->succ;
8      p->succ->pred = p->pred;
9      delete p; _size--;
10     return e; //返回备份数值
11 }
```

析构

```
1  template <typename T>
2  List<T>::~List() //清空列表，释放头、尾哨兵节点
3  { clear(); delete header; delete trailer; }
4
5  template <typename T> Rank List<T>::clear()
6  { //清空列表
7      Rank oldSize = _size;
8      while ( 0 < _size ) //反复
9          remove( header->succ ); //删除首节点O(n)
10     return oldSize;
11 }
```

唯一化

```
1  template <typename T>
2  int List<T>::deduplicate() { // O(n^2)
3      if (_size < 2) return 0;
4      int oldSize = _size;
5      ListNodePosi(T) p = header;
6      Rank r = 0; // p从首节点开始
7      while(trailer != (p = p->succ))
8      {    //依次直到末节点
9          ListNodePosi(T) q = find(p->data, r, p);
10         q ? remove(q) : r++; // why q?
11     }
12     return oldSize - _size;
13 }
```

- 正确性及效率分析的方法与结论，与 Vector::deduplicate() 相同

遍历

```
1  // 函数指针
2  template <typename T> void List<T>::
3  traverse( void ( * visit )( T & ) ) {
4      for( NodePosi<T> p = header->succ;
5           p != trailer; p = p->succ )
6          visit( p->data );
7  }
8  // 函数对象
9  template <typename T> template <typename VST>
10 void List<T>::traverse( VST & visit )
11 {
12     for( NodePosi<T> p = header->succ;
13          p != trailer; p = p->succ )
14         visit( p->data );
15 }
```

唯一化

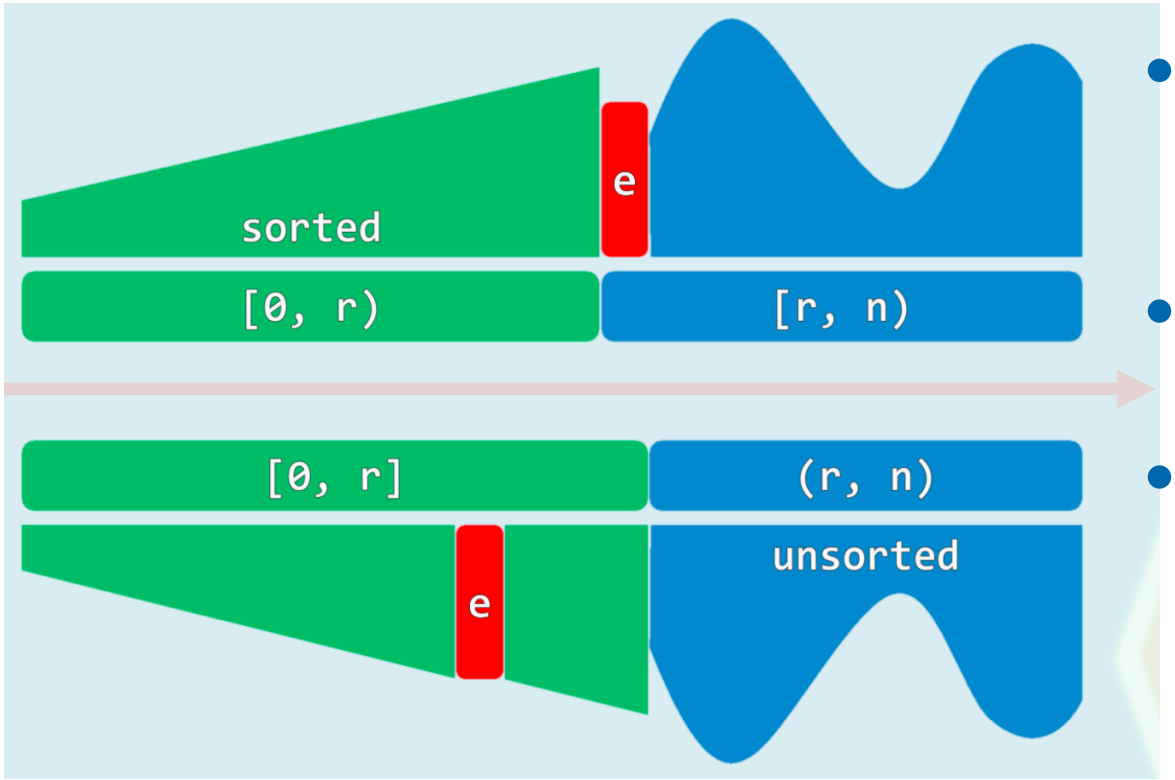
```
1  template <typename T> Rank List<T>::uniquify() {
2      if ( _size < 2 ) return 0; //平凡列表无重复
3      Rank oldSize = _size; //记录原规模
4      ListNodePosi<T> p = first();
5      ListNodePosi<T> q; //各区段起点及其直接后继
6      //反复考查紧邻的节点对(p,q)
7      while ( trailer != ( q = p->succ ) )
8          if ( p->data != q->data )
9              p = q; //若互异，则转向下一对
10         else // 否则（雷同）直接删除后者
11             remove(q); //不像向量使用间接删除
12     return oldSize - _size; //被删除元素总数
13 } //只需遍历整个列表一趟，O(n)
```

查找

```
1  template <typename T> //0 <= n <= rank(p) < _size
2  ListNodePosi(T) List<T>::search(T const& e,
3       Rank n, ListNodePosi(T) p) const
4  {
5       while( 0 <= n--)
6            if(((p = p->pred)->data) <= e)
7                 break;
8       return p; // 返回查找终止位置
9  }  // 失败返回左边界前驱，通过valid()判断成果与否
```

- 为什么与有序向量查找算法完全不同?

插入排序



- 不变性：序列总能视作 S[0, r) + U[r, n) 两部分
- 初始化 $|S| = r = 0$
- 针对 e = A[r]，反复在 S 中查找适当位置，以插入e

插入排序

迭代轮次	有序前缀	当前元素	无序后缀
-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	^	^

插入排序

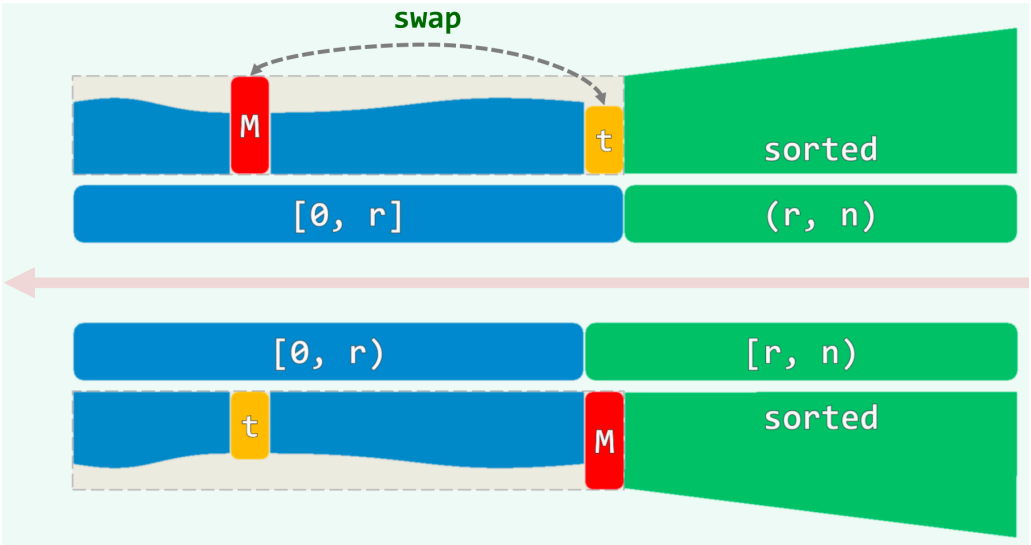
```
1  template <typename T> void List<T>::
2  insertionSort( ListNodePosi(T) p, int n )
3  {    //逐一引入各节点，由Sr得到Sr+1
4       for ( int r = 0; r < n; r++ ) {
5            insertA(search(p->data, r, p), p->data );
6            p = p->succ; remove( p->pred );
7       } //n次迭代，每次O(r + 1)
8  } //仅使用O(1)辅助空间，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

插入排序：性能分析

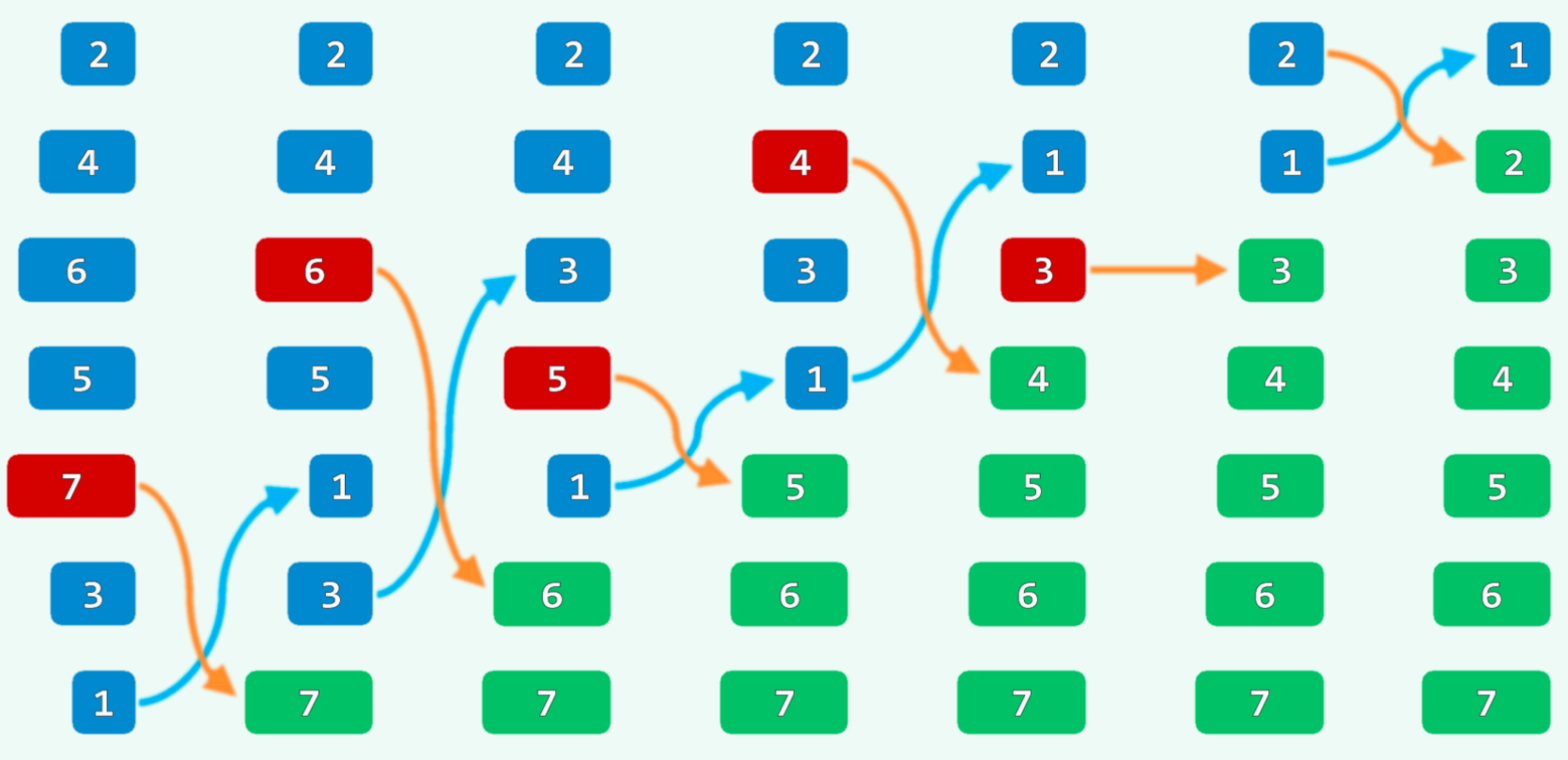
- 最好情况：完全（几乎）有序
  - 每次迭代，只需 1 次比较，0 次交换
  - 累计 $\mathcal{O}(n)$ 时间
- 最好坏情况：完全（几乎）逆序
  - 第 k 次迭代，需 $\mathcal{O}(k)$ 次比较，1 次交换
  - 累计 $\mathcal{O}(n^2)$ 时间

选择排序

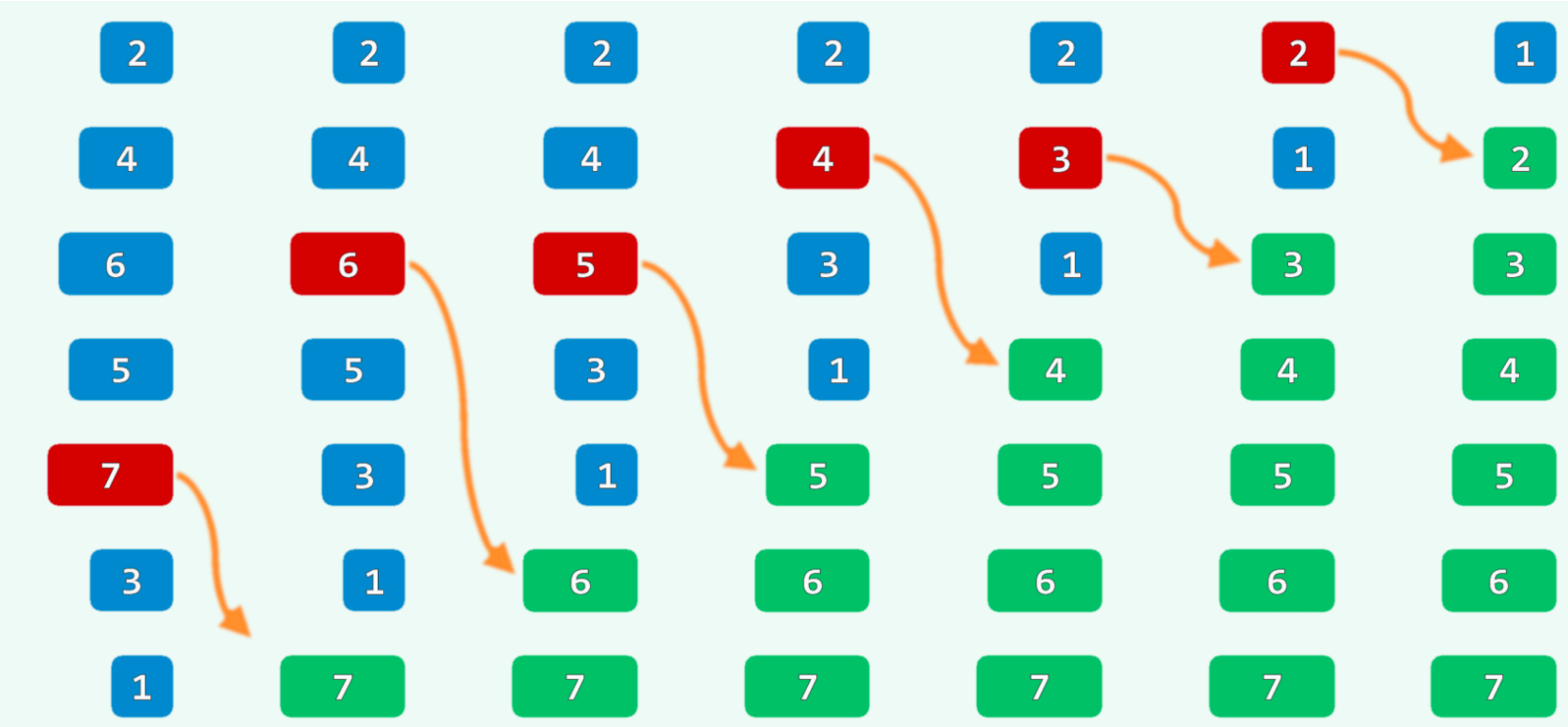


- 起泡排序每趟要 $\mathcal{O}(n)$ 次比较、$\mathcal{O}(n)$ 次交换
- 扫描交换的实质
  - 通过比较找到当前最大元素
  - 通过交换使之就位

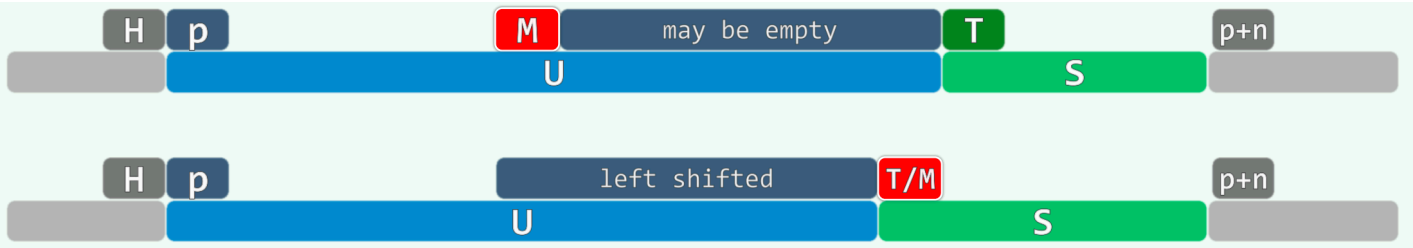
选择排序：交换法



选择排序：平移法

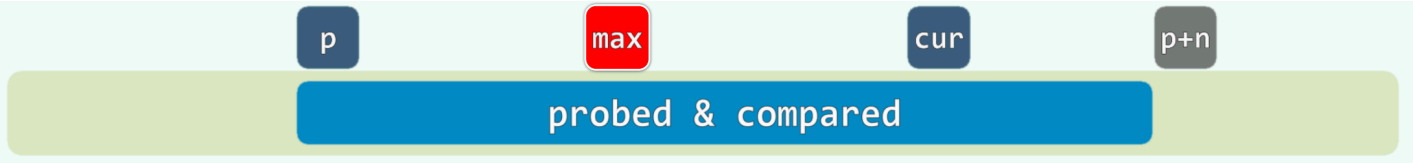


选择排序




```
1  template <typename T> void List<T>::
2  selectionSort( ListNodePosi(T) p, int n ) {
3      ListNodePosi(T) head = p->pred, tail = p;
4      for ( int i = 0; i < n; i++ )
5          tail = tail->succ; //待排区间(head, tail)
6      while ( 1 < n ) {
7          ListNodePosi(T) max =
8              selectMax( head->succ, n );
9          insertB( tail, remove(max) ) ;
10         tail = tail->pred; n--; }
11 }
```

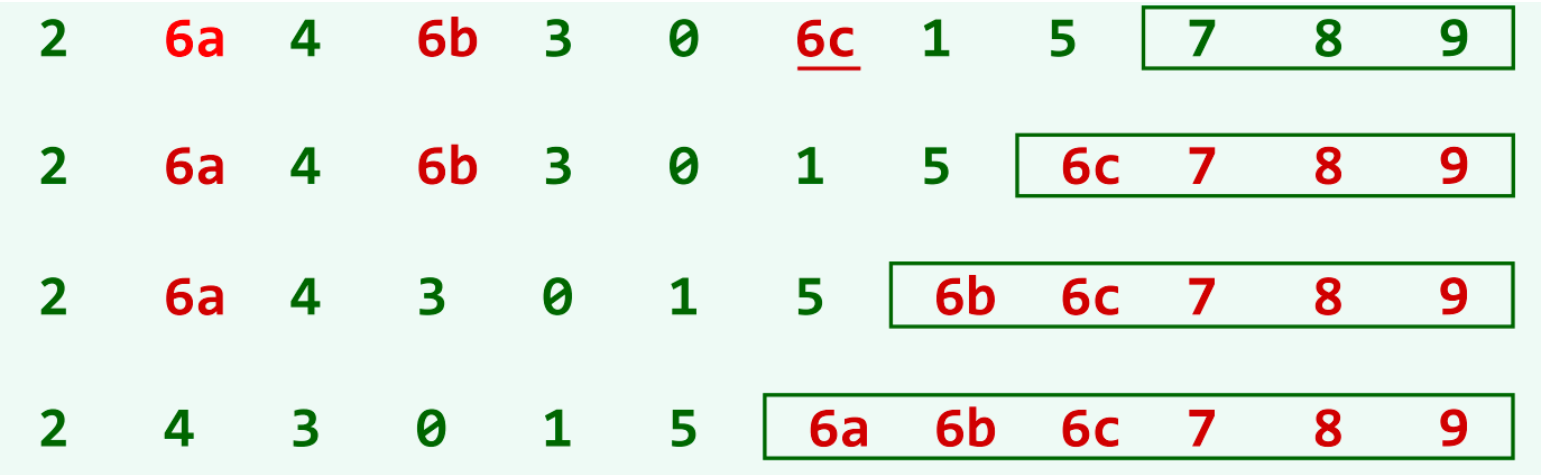
选择排序




```
1  template <typename T> ListNodePosi(T) List<T>::
2  selectMax( ListNodePosi(T) p, int n ) { //O(n)
3      ListNodePosi(T) max = p;
4      for ( ListNodePosi(T) cur = p; 1 < n; n-- )
5          // 为什么用 lt, 不直接用 data >= max?
6          if (!lt((cur=cur->succ)->data,max->data))
7              max = cur; //则更新最大元素位置记录
8      return max; //返回最大节点位置
9  }
```

选择排序：稳定性

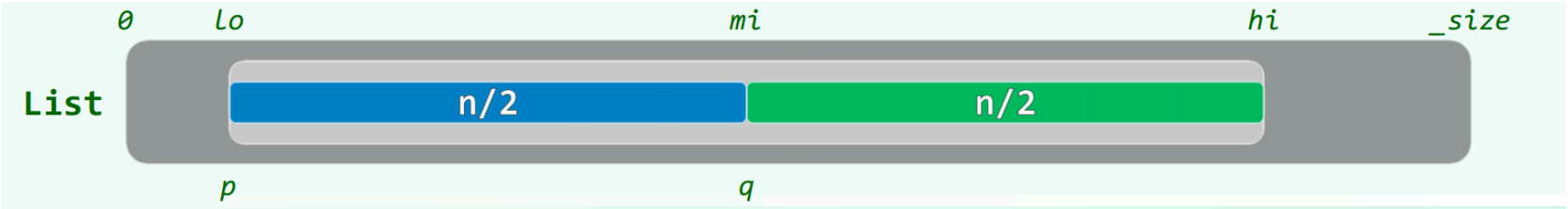
- 稳定性：有多个元素同时命中时，约定返回其中特定的某一个（比如最靠后者）
- 若采用平移法，如此即可保证，重复元素在列表中的相对次序，与其插入次序一致



选择排序：性能分析

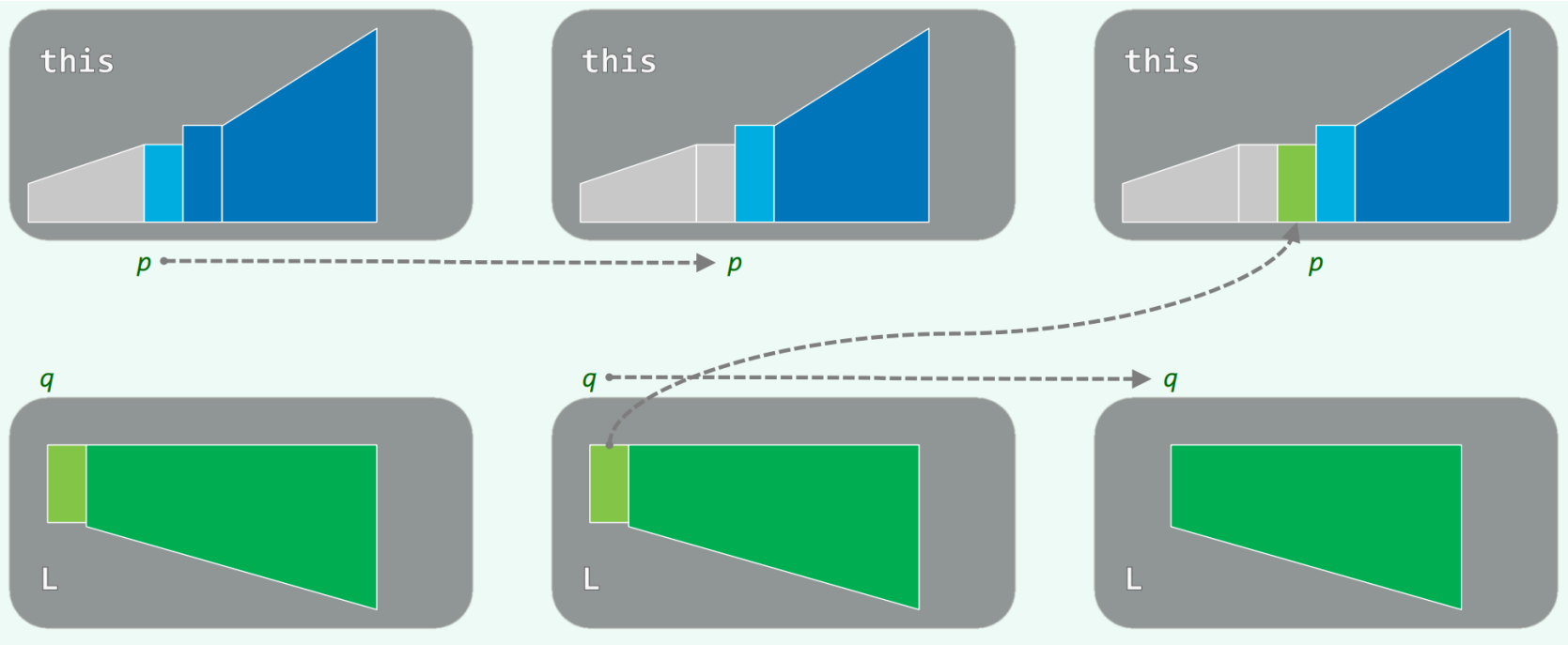
- 共迭代 n 次，在第 k 次迭代中
  - selectMax() 为 $\Theta(n-k)$
  - swap() 为 $\mathcal{O}(1)$
  - 故总体复杂度应为 $\Theta(n^2)$
- 尽管如此，元素的移动操作远远少于起泡排序，$\Theta(n^2)$ 主要来自于元素的比较操作
- 可否每轮只做$o(n)$ 次比较，即找出当前的最大元素？
- 利用高级数据结构，selectMax() 可改进至 $\mathcal{O}(logn)$！

归并排序




```
1  template <typename T> void List<T>::
2  mergeSort( ListNodePosi(T)& p, int n ) {
3      if ( n < 2 ) return;
4      Rank m = n >> 1;
5      ListNodePosi(T) q = p;
6      for ( int i = 0; i < m; i++ )
7          q = q->succ; //均分列表：O(m) = O(n)
8      mergeSort( p, m );
9      mergeSort( q, n - m );
10     p = merge( p, m, *this, q, n - m ); //归并
11 } //若归并可在线性时间内完成，时间复杂度O(nlogn)
```

归并排序



归并排序

```
1  template <typename T> void List<T>::
2  merge(ListNodePosi(T)& p, int n, List<T>& L,
3      ListNodePosi(T) q, int m )
4  {
5      ListNodePosi(T) pp = p->pred;
6      while (0 < m)
7          if ((0<n) && (p->data <= q->data)) {
8              if(q == (p = p->succ)) break;
9              n--;
10         }
11         else {
12             q = q->succ;
13             insertB(p, L.remove(q->pred));
14             m--;
15         }
16     p = pp->succ;
17 }
```