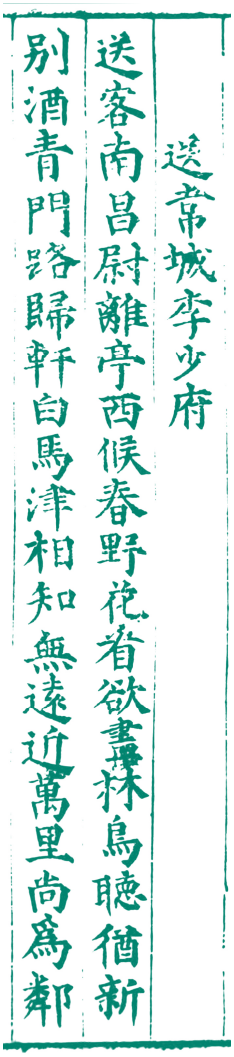
第3章

列表


送常城李少府
送客南昌尉離亭西候春野花着欲盡林鳥聽猶新
別酒青門路歸軒白馬津相知無遠近萬里尚爲鄰

前一章的主角向量，更擅长于静态操作，然而动态操作却显得十分吃力。究其根源在于，为落实循秩访问的规范，不得不将元素的物理次序与其逻辑次序直接地绑定；作为代价，机动灵活性丧失殆尽。由此视角反观，为从根本上改进动态操作，就必须着眼于将数据元素的物理位置与逻辑位置解耦。比如接下来将要登场的列表，就会转而采用循位置访问的规范。

本章的重点在于列表的相关各种算法，尤其是如何从数学概念出发来审视算法的原理、过程及结果，以及如何从在线处理能力、输入敏感性等维度，更为精细地考查、分析与界定不同的算法，从而更切实地理解它们各自的特性及适用场合。

当然，与向量一样地，列表的缺点也会在我们眼前逐渐清晰地暴露出来。这将会促使我们开始思考，如何才能同时回避二者的不足，并将其优势结合起来。

§3A 抽象数据类型

列表（list）与向量本是姊妹，都来自于序列（sequence）家族。其中的元素仍是依照线性次序排列，逻辑上也各自对应于一个秩，相邻元素亦互称前驱、后继，故依然可以记作：

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

但与向量根本不同的是，列表中各元素的物理次序不再与其逻辑次序直接相关；逻辑上紧邻的一对前驱与后继元素，物理上既不必保持前后次序，更未必相互紧邻。

3A1. 循位置访问

当然，为保证能够访问到每个元素，逻辑上的每一对紧邻元素，都需要各自记录指向对方的链接（link），即对方的物理位置。这里，将物理位置又进一步抽象为元素的位置（position）。于是如图3.1，我们便可以循着位置，在常数时间内从一个元素转入下一个。这种转移连续地执行多次，便可以抵达列表中的任何一个元素。这里的位置犹如名片，帮助我们通过某位朋友，找到他的某位亲戚，进而找到他的同事、同事的战友、战友的同学、同学的老师……。

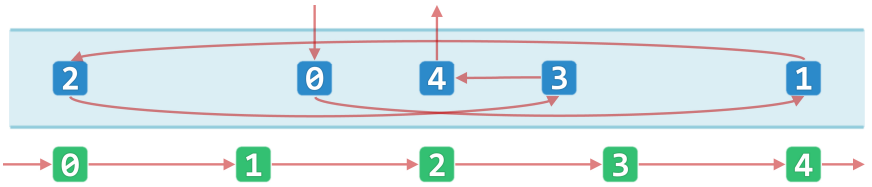


图3.1 循位置访问

这种顺藤摸瓜式的访问方式，称作循位置访问（call-by-position）。这种方式不仅令物理位置得以独立于逻辑次序，更使得动态操作仅需涉及局部的常数个元素，而不再是与全局相关，从而为实现高效率的动态操作提供了机制上的可能与保证。我们很快就会看到，列表的多数动态操作确实因此都可以在常数时间内完成。

那么接下来，这样的列表ADT又当如何具体实现呢？

3A2. 接口与实现

链表（linked list）是列表的一种典型实现方式。与列表元素相对应地，链表的基本单位称作节点（node），其中既存有元素所对应的数据项，也会以指针的形式记录前驱、后继节点的位置。于是，节点之间可通过指针相互索引、顺次访问；而为了引入新节点或删除原有节点，也只需在局部调整少量相关节点之间的指针，有望在常数时间内完成。

■ 节点

若将链表比作一列火车，则节点就是其中的一节车厢。如图3.2，节点内的数据项便是车厢内的货物，节点位置即车厢之间的车钩。



图3.2 列表节点

这样的链表，乃是一个分两级封装的结构。尽管如此，为简洁起见，我们并不会十分严格地对底层节点做封装。

从抽象数据类型的角度出发，列表节点需要支持表3.1中所列的操作接口。

表3.1 列表节点ADT的操作接口及其功能

操 作 接 口	功 能
data	获取当前节点所存数据对象
pred|succ	获取当前节点之前驱|后继的位置
insertPred(e)|insertSucc(e)	插入前驱|后继节点
movePred(s)|moveSucc(p)	将当前节点转移至紧靠于s之前|紧接于p之后

对照表3.1约定的接口，可实现ListNode模板类如下：

```
template <typename T> using LNP = ListNode<T>*; //列表节点位置|List Node Position
template <typename T> struct ListNode { //列表节点模板类（以双向链表形式实现）
   T data;  LNP<T> pred, succ; //数值、前驱、后继
   ListNode() : data(), pred(nullptr), succ(nullptr) {}
   ListNode(const T& e, LNP<T> p = nullptr, LNP<T> s = nullptr)
      : data(e), pred(p), succ(s) {} //类T已定义复制方法
   LNP<T> insertPred(const T& e); //紧靠当前节点之前插入新节点
   LNP<T> insertSucc(const T& e); //紧随当前节点之后插入新节点
   void movePred(LNP<T> s); //转移到紧靠于s之前
   void moveSucc(LNP<T> p); //转移到紧接于p之后
   LNP<T> operator+(Rank r); //循秩访问this[r]（效率低）
   Rank operator-(LNP<T> t); //统计区间[this，t)的长度（效率低）
}; // ListNode
```

代码3.1 列表节点模板类

■ 列表

作为一种抽象数据类型，列表对象应支持以下操作接口。

表3.2 列表ADT的操作接口及其功能

操 作 接 口	功 能
size()|empty()	报告节点总数|判定是否为空
first()|last()	返回首|末节点的位置
insertFirst(e)|insertLast(e)	将e当作首|末节点插入
insert(p,e)|insert(e,p)	将e当作节点p的后继|前驱插入
remove(p)	删除节点p
sort()	排序
find(e)	在无序列表中，查找与e相等者（如有多个，取最靠前者）
search(e)	在有序列表中，查找不大于e的最大者
dedup()	从无序列表中，剔除相等的元素（保留最靠后者）
uniquify()	从有序列表中，剔除相等的元素（保留最靠前者）
traverse(visit())	遍历列表，统一按visit()处理所有节点

对照表3.2约定的接口，可实现List模板类如下：

```
template <typename T> class List { //列表模板类
protected:
   Rank _size;  LNP<T> head, tail; //规模、头哨兵、尾哨兵
   void init(); //列表创建时的初始化
   void copy(LNP<T>, Rank); //复制列表中自位置p起的n项
   void mergeSort(LNP<T>, LNP<T>); //归并排序
   void selectionSort(LNP<T>, LNP<T>); //选择排序
   void insertionSort(LNP<T>, LNP<T>); //插入排序
   void radixSort(LNP<T>, LNP<T>); //基数排序
public:
   List() { init(); } //默认
   List(const List<T>& L, Rank r, Rank n) { copy(L[r], n); } //复制L中自第r项起的n项
   List(const List<T>& L) { copy(L.first(), L._size); } //整体复制列表L
   List(LNP<T> p, Rank n) { copy(p, n); } //复制列表中自位置p起的n项
   ~List(); //析构
//静态操作
   Rank size() const { return _size; } //规模
   bool empty() const { return _size <= 0; } //判空
   LNP<T> operator[](Rank r) const; //重载，支持循秩访问（效率低）
   LNP<T> first() const { return head->succ; } //首节点位置
   LNP<T> last() const { return tail->pred; } //末节点位置
   LNP<T> find(const T& e) const { return find(head, e, tail); } //无序查找
   LNP<T> find(LNP<T>, const T&, LNP<T>) const; //无序区间查找
   LNP<T> search(const T& e) const { return search(head, e, tail); } //有序查找
   LNP<T> search(LNP<T>, const T&, LNP<T>) const; //有序区间查找
   LNP<T> getMax(LNP<T>, LNP<T>); //找出区间中的最大者
   LNP<T> getMax() { return getMax(head, tail); } //整体最大者
//动态操作
   template <typename VST> void traverse(VST&); //遍历（函数对象）
   LNP<T> insertFirst(const T& e); //将e当作首节点插入
   LNP<T> insertLast(const T& e); //将e当作末节点插入
   LNP<T> insert(LNP<T> p, const T& e); //将e当作p的后继插入
   LNP<T> insert(const T& e, LNP<T> p); //将e当作p的前驱插入
   T remove(LNP<T>); //删除节点
   void sort(LNP<T>, LNP<T>); //区间排序
   void sort() { sort(head, tail); } //整体排序
   void merge(LNP<T>, LNP<T>, LNP<T>); //区间归并
   void merge(LNP<T> y) { merge(head, y, tail); } //整体归并
   void dedup(); //无序去重
   void uniquify(); //有序去重
}; //List
```

代码3.2 列表模板类

除了与列表各元素相对应的节点，每个链表还设有一对哨兵节点，分别称作头节点（head）和尾节点（tail）。虽然它们对外并不可见，却与链表终身相伴；虽然消耗了区区常数的空间，但我们很快就将体会到，它们将极大地简化算法的逻辑描述，提高实际的运行效率，以及更重要地，帮助我们在思考时跳过繁琐的细节，将注意力汇聚于宏观的操作。

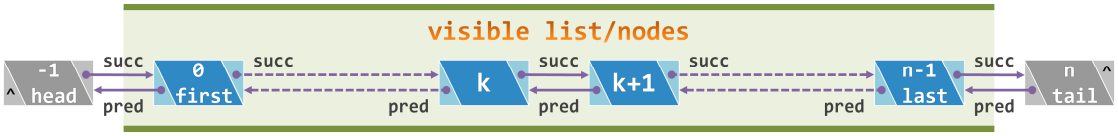


图3.3 链表的两级封装

为了避免叙述时的歧义，我们不妨约定，将第一个和最后一个对外可见的节点称作首节点（first node）和末节点（last node）。如图3.3即是一个典型的链表，及其哨兵节点与首、末节点。与哨兵节点不同，首、末节点未必总是存在；即便存在，也可能是同一个节点。

■ 初始化

首先，每个链表的最初状态都应当如图3.4，尚无任何对外可见的节点，其规模为零。



图3.4 初生的链表空空如也

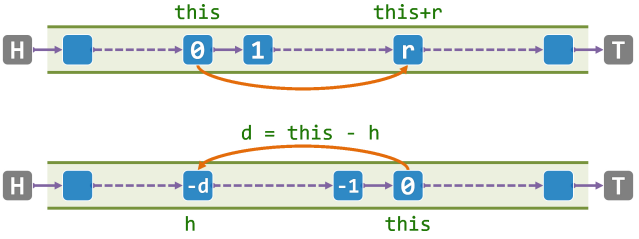


图3.5 模仿循秩访问：
由偏移距离确定节点（上），丈量节点之间的偏移距离（下）

即便如此，仍然应当遵照以上的约定，如算法3.1所示创建出哨兵节点head与tail，并令它们记住彼此的位置，为后续的元素以及进一步的操作插入打好基础。

```
1 template <typename T> void List<T>::init() { //列表初始化，在创建列表对象时统一调用
2    head = new ListNode<T>;  tail = new ListNode<T>; //创建头、尾哨兵节点
3    head->succ = tail;  tail->pred = head; //并令其相互联接
4    _size = 0; //记录规模
5 } //init
```

算法3.1 链表的初始化

■ 模拟循秩访问

既然链表也是线性结构，自然也可以如算法3.2，通过秩来确定任一节点：

```
1 template <typename T> //重载下标操作符，以通过秩直接访问列表节点
2 LNP<T> List<T>::operator[](Rank r) const //0 <= r < size
3    { return *first() + r; } //O(r)：虽方便，勿多用
```

算法3.2 模仿循秩访问

进一步地如算法3.3、算法3.4，还可通过重载操作符实现如图3.5所示的更多功能：

```
template <typename T> //重载加号操作符，以通过秩直接访问列表节点
LNP<T> ListNode<T>::operator+(Rank r) { //0 <= r < size-this
   for (LNP<T> p = this; ; p = p->succ, r--) //从当前节点出发
      if (0 == r) return p; //顺数第r个节点即是目标节点
} //O(r)：虽方便，勿多用
```

算法3.3 由偏移距离确定节点

```
template <typename T> //重载减号操作符，统计列表区间[h,this)的宽度
Rank ListNode<T>::operator-(LNP<T> h) { //h <= this
   for (Rank d = 0; ; h = h->succ, d++)
      if (h == this) return d; //区间内的节点，挨个报数
} //O(n)
```

算法3.4 计算节点之间的偏移距离

需要指出的是，尽管这些操作符可使代码更加精简，但毕竟其效率不高，只可偶尔一用。

§3B 无序列表

经过以上的铺垫与热身，我们已经可以着手实现更多接口了。

3B1．插入、删除与转移

■ 插入

为将节点插至列表，可视具体的需求，在代码3.3所列的多个接口中灵活选用。

```
template <typename T> LNP<T> List<T>::insertFirst(const T& e)
   { _size++;  return head->insertSucc(e); } //e当作首节点插入
template <typename T> LNP<T> List<T>::insertLast(const T& e)
   { _size++;  return tail->insertPred(e); } //e当作末节点插入
template <typename T> LNP<T> List<T>::insert(LNP<T> p, const T& e)
   { _size++;  return p->insertSucc(e); } //e当作p的后继插入
template <typename T> LNP<T> List<T>::insert(const T& e, LNP<T> p)
   { _size++;  return p->insertPred(e); } //e当作p的前驱插入
```

代码3.3 列表插入接口

可见，这些接口都转化为列表节点的前插入、后插入接口，具体实现如算法3.5所示。

```
template <typename T> LNP<T> ListNode<T>::insertPred(const T& e) {
   LNP<T> x = new ListNode(e, pred, this); //创建新节点（pred或为head）
   pred->succ = x;  pred = x;  return x; //列表接纳新节点（操作次序不可颠倒）
} //insertPred：紧靠当前节点之前，将e插入列表；O(1)

```

```
template <typename T> LNP<T> ListNode<T>::insertSucc(const T& e) {
   LNP<T> x = new ListNode(e, this, succ); //创建新节点（succ或为tail）
   succ->pred = x;  succ = x;  return x; //列表接纳新节点（操作次序不可颠倒）
} //insertSucc：紧随当前节点之后，将e插入列表：O(1)
```

算法3.5 列表节点的前插入、后插入

以前插入算法为例，图3.6给出了整个操作的具体过程。

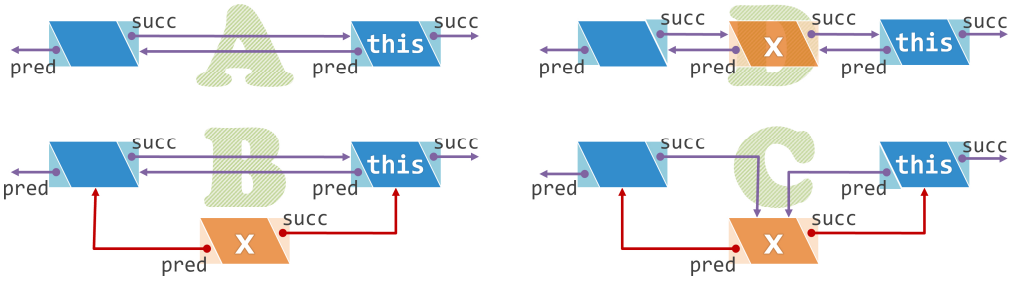


图3.6 前插入节点的过程

在插入之前，列表的局部如图A。接下来如图B，创建一个新节点x：构造函数将它的数据项置为e，并以当前节点this为其后继，以this的前驱为其前驱。随后如图C，令this的前驱以x为后继，令this以x为前驱。最终如图D，这个新节点被顺利地插至列表的这一局部。

后插入算法完全对称，不再赘述。两个算法只涉及局部的两个原有节点和一个新节点，而且不含任何迭代或递归，新节点x也可在常数时间内创建，故它们的时间复杂度均为$\mathcal{O}(1)$。

■ 删除

上述插入操作的过程颠倒过来，即可如图3.7，从列表中删除指定节点p：

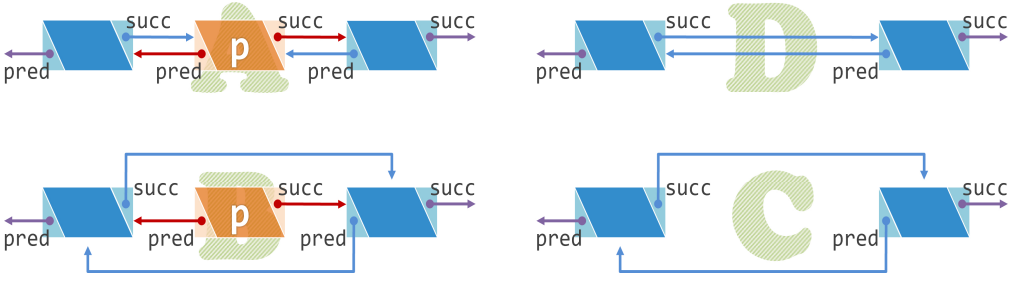


图3.7 删除节点的过程

如此删除节点的过程，可描述并实现为算法3.6：

```
template <typename T> T List<T>::remove(LNP<T> p) { //O(1)
   T e = p->data; //备份数据项（T类型可直接赋值）
   p->pred->succ = p->succ;  p->succ->pred = p->pred; //短路联接（pred|succ或为head|tail）
   delete p;  _size--;  return e; //释放节点，更新规模，返回数据项
} //remove
```

算法3.6 列表的节点删除

同样地，该算法只涉及局部的个别节点，亦无迭代或递归，故时间复杂度也是$\mathcal{O}(1)$。

■ 转移

节点的转移有对称的两个接口，其中之一如图3.8，将当前节点转移至节点s之前：

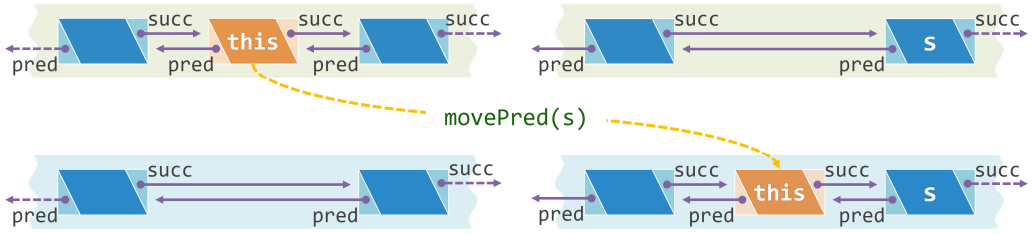


图3.8 转移节点的过程

该操作接口以及对称的另一个接口，可描述并实现为算法3.7：

```
template <typename T> void ListNode<T>::movePred(LNP<T> s) { //s != this
   pred->succ = succ;  succ->pred = pred; //取出当前节点
   pred = s->pred;  pred->succ = this;  succ = s;  s->pred = this; //转移到紧靠于s之前
} //O(1)

template <typename T> void ListNode<T>::moveSucc(LNP<T> p) { //p != this
   succ->pred = pred;  pred->succ = succ; //取出当前节点
   succ = p->succ;  succ->pred = this;  pred = p;  p->succ = this; //转移到紧接于p之后
} //O(1)
```

算法3.7 列表节点的转移

这里的基准节点s与p的位置，可以位于当前节点this之前，也可以位于其后，甚至紧邻于this之前或之后。两个算法都只涉及局部的个别节点，故时间复杂度均为$\mathcal{O}(1)$。

列表的算法远远不止以上介绍的这几个，但比照对应的向量算法，我们已经可以初步体会到，在将物理地址与逻辑次序分离、改为循位置访问之后，动态操作的效率确实可以大大提高。

3B2. 复制与析构

■ 深复制

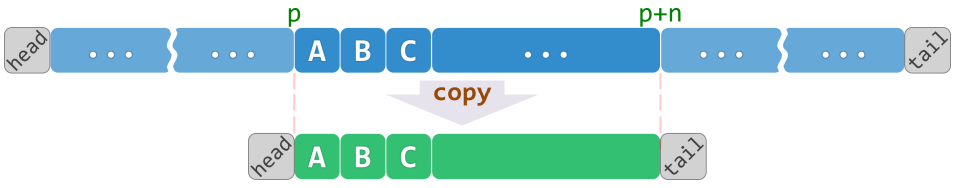


图3.9 列表的深复制

回观代码3.2中定义的几种构造方法，一个初生的链表可能是由init()创建的空表，从零开始；也可能是如图3.9，调用copy()接口，以现有的列表为蓝本复制出来。不过，对于以模板类的形式定义的数据结构，除了要创建出一个新的结构，还必须递归地复制蓝本中的每一层嵌套结构，以保证新结构与蓝本彼此独立，没有任何的数据关联——此即所谓的深复制（deep copy）。

得益于C++的规范，只要类型T本身支持深复制，便可按算法3.8的方式实现列表的深复制。

```
template <typename T> //依据蓝本列表的区间，复制出当前列表
void List<T>::copy(LNP<T> p, Rank n) {
   for (init(); n--; p = p->succ) //先创建空表；再将蓝本区间内各节点
      insertLast(p->data); //依次复制，并作为末节点插入
} //时间复杂度线性正比于n，以及每个T类型对象的深复制所需的时间
```

算法3.8 列表的深复制

■ 析构

自然地，对于列表这类多层次嵌套的结构，在析构时也应递归地释放每一层的数据。同样地，只要类型T本身可以递归地析构，便可按算法3.9的方式实现列表的递归析构。

```
template <typename T> List<T>::~List() { //从head到tail，逐个释放每个节点
   for (LNP<T> x = head; x != tail; delete (x = x->succ)->pred);
   delete tail;
} //时间复杂度线性正比于节点总数，以及每个T类型对象的递归析构所需的时间
```

算法3.9 列表的递归析构

3B3. 查找与去重

■ 查找

列表ADT针对整体和区间的查找，分别重载了一个接口。在代码3.2中，前者是作为特例直接调用后者。因此，只需如算法3.10所示实现区间查找接口。

```
template <typename T> //在无序列表区间(h,t)内，找到等于e的最靠前者：h < t
LNP<T> List<T>::find(LNP<T> h, const T& e, LNP<T> t) const {
   T bak = t->data;  t->data = e; //预置哨兵
   while (e != (h = h->succ)->data); //自前向后，逐个比对
   t->data = bak;  return h == t ? nullptr : h; //哨兵复原，返回终止位置
} //find: O(n)
```

算法3.10 无序列表的查找[①]



图3.10 无序列表的查找

该算法的原理及过程如图3.10所示，与向量的顺序查找算法2.5[p46]如出一辙，复杂度相同，各方面的性质也大体类似，故不再赘述。

① 我们约定，通过首、尾位置作为界桩来指示的列表区间，都是左右均开的——两个界桩都不属于区间的有效范围。

■ 去重

仿照算法2.6[p46]，也可实现无序列表的去重接口dedup()如下：

```
template <typename T> void List<T>::dedup() { //O(n^2)
   for (LNP<T> p = first(); p != tail; p = p->succ) //在无重的前缀(head,p)中
      if (LNP<T> q = find(head, p->data, p)) //查找可能与p相等的q
         remove(q); //若q确实存在，则（在常数时间内）删除之
} //dedup
```

算法3.11 无序列表的去重

当所有节点彼此互异时，该算法与算法2.6一样，需要$\mathcal{O}(n^2)$时间。若所有节点全等，则前缀始终仅含一个节点，每一次find()只需$\mathcal{O}(1)$时间；更重要的是，由于列表已经摆脱循秩访问的窠臼，每一次remove()也只需$\mathcal{O}(1)$时间——故最好情况下总体仅需$\mathcal{O}(n)$时间，属于输入敏感的算法。

3B4．遍历

列表的遍历接口，可实现如代码3.4。其设计思路与实现方式，与代码2.5[p47]向量的对应接口如出一辙，复杂度也相同，故不再赘述。

```
template <typename T> template <typename VST> void List<T>::traverse(VST& visit) //遍历
   { for (LNP<T> p = first(); p != tail; p = p->succ) visit(p->data); }
```

代码3.4 列表的遍历

你可以仿照第2B5节中的实例，借助遍历接口自行实现对列表的各种统一处理。比如，仿照算法2.9[p48]实现inv(List<T>)，通过遍历统计出列表所含相邻逆序对的总数。

§3C 有序列表

假定列表的基本类型T支持相互比较——无论是原生就支持，还是通过定义比较器。若通过上述inv()确认不含逆序对，则意味着所有节点的逻辑次序与其大小次序完全一致，这类列表称作有序列表（sorted list）。与有序向量一样，这里依然约定采用非降次序。

■ 去重

依然如图2.12所示，与有序向量一样，有序列表中的每一组相等节点，也必然（在逻辑上）彼此紧邻，构成一个区间。因此也可仿照算法2.10，实现有序列表的去重算法3.12。

```
template <typename T> void List<T>::uniquify() { //成批剔除重复元素，效率更高
   if (_size < 2) return; //过短的列表，自然不含相等元素
   for (LNP<T> p = first(), q = p->succ; q != tail; q = p->succ)
      if (p->data != q->data) p = q; //若p与其后继q不等，则前进一步
      else remove(q); //否则直接删除后者，而不必如向量那样间接删除
} //uniquify()
```

算法3.12 有序列表的去重

其中的p和q总是指向一对相邻节点，若二者雷同则删除q，否则转向下一对相邻节点。如此反复迭代，直至检查过了所有节点。耐人寻味的是，尽管这一思路与算法2.10如出一辙，却可以直接地实现$\mathcal{O}(n)$的效率，无需再如算法2.11那样做什么改进。表面上看，这是因为列表的remove()可在$\mathcal{O}(1)$时间内完成；究其根源，仍然是得益于列表摒弃了循秩访问的模式。

■ 查找

很遗憾，在动态操作性能大幅提高的同时，列表静态操作的性能却有所下降，比如查找操作。有序向量行之有效的二分查找策略在此无法照搬。其原因在于，从向量中取出居中的轴点只需要常数的时间，而列表则需要线性的时间。这就意味着，即便强行地采用折半策略，列表的查找也需要$\mathcal{O}(n)$时间。既然如此，倒不如就像无序列表那样，直接做顺序查找（算法3.10）。

```
template <typename T> //在有序列表区间(h,t)内，找到不大于e的最靠后者：h < t
LNP<T> List<T>::search(LNP<T> h, const T& e, LNP<T> t) const { //O(n)
   T bak = h->data;  h->data = e; //预置哨兵
   while (e < (t = t->pred)->data); //自后向前，逐个比较
   h->data = bak;  return t; //直至不大于e的最靠后者
} //特别地，在所有节点都大于e时，返回h（等效于h->data == inf)
```

算法3.13 有序列表的查找

对照有序向量的同名search()查找接口[p50]，这里也对返回值做了精准的约定：不大于目标的最大元素。这对于后续将要介绍的一些算法至关重要，比如插入排序算法3.16[p80]。

§3D 选择排序

在表3.2中，我们已经将sort()作为基本的操作接口纳入了列表ADT。而在代码3.2中，我们又为列表重载了两个sort()接口，分别针对整体或区间的排序，且前者是作为特例调用后者。

```
template <typename T> void List<T>::sort(LNP<T> h, LNP<T> t) {
   switch (rand() % 4) { //随机选取排序算法。可根据具体问题的特点灵活选取或扩充
      case 1  : insertionSort(h, t);  break; //插入排序
      case 2  : selectionSort(h, t);  break; //选择排序
      case 3  :     mergeSort(h, t);  break; //归并排序
      default :     radixSort(h, t);  break; //基数排序（第09章）
   } //switch
} //sort
```

代码3.5 有序列表的排序操作接口

如代码3.5实现的列表区间排序接口，背后有多种排序算法可选，我们先来考查选择排序。

■ 减而治之

回顾起泡排序算法1.3。我们曾将其不变性概括为：每迭代一步，当前的最大元素便会就位。于是采用减而治之的策略，只要能反复地摘出最大元素，便自然得到了所有元素的次序（逆序）。

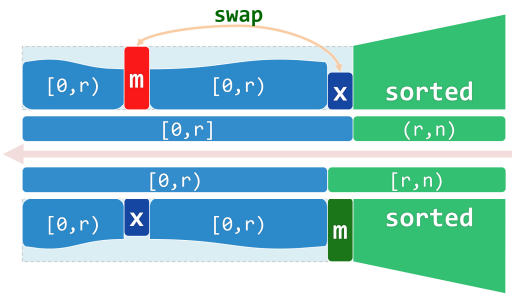


图3.11 选择排序的原理

还是来模仿起泡排序，如图3.11，始终将序列视作一个无序的前缀，以及一个有序的后缀，而且前缀不大于后缀。于是，实质任务无非就是：如何持续地缩短前缀，相应地扩张后缀？不难。

为此，只需反复地从前缀中选出最大元素m；在m与前缀的末元素x交换之后，有序的后缀便会增长一个单位。可见，这里的关键在于如何选出m，故这个算法称作选择排序（selectionsort）。

■ 实现

以下算法3.14虽然只是以列表为例的实现，但不难推广至包括向量在内的任何线性序列。

```
template <typename T> void List<T>::selectionSort(LNP<T> h, LNP<T> t) {
   for (LNP<T> x = t->pred; h->succ != x; t = x, x = t->pred) //反复地从无序前缀(h,t)中
      swap(getMax(h,t)->data, x->data); //找出最大者，并通过交换令其就位
      //getMax(h,t)->movePred(t); //亦可通过移动就位：同样是O(1)成本，且能保证排序的稳定性
} //selectionSort
```

算法3.14 列表的选择排序

从无序前缀中找出最大元素m的任务由getMax()算法3.15完成，为此需要做一趟遍历[①]，耗时线性正比于前缀的长度。这也是时间成本的主要部分，选择排序的整体复杂度为$\mathcal{O}(n^2)$。

```
template <typename T> LNP<T> List<T>::getMax(LNP<T> h, LNP<T> t) { //2 <= t-h
   LNP<T> m = t->pred; //当前的最大者
   for (t = m->pred; h != t; t = t->pred) //从后向前，逐个比较
      if (m->data < t->data) //见贤：若发现严格更大的元素
         m = t; //思齐：则更新记录
   return m; //最终的最大者：即便有多个，也总是取最靠后者
} //getMax：O(n)
```

算法3.15 列表最大节点的定位

细察之后不难发现，对于任何输入序列，getMax()的耗时总是$\Theta(n^2)$——固定不变，没有最好情况。尽管如此，较之渐近复杂度同为$\mathcal{O}(n^2)$的起泡排序[p56]，它还是有了长足的进步。实际上，虽然选择排序仍要做$\mathcal{O}(n^2)$次比较操作，但移动操作却从$\mathcal{O}(n^2)$次减至$\mathcal{O}(n)$次。就复杂度的常系数而言，后一类操作通常会高于前一类，对实际效率影响更大。

继续细审之后我们还会注意到，选出的最大元素有可能本已就位（m=x），此时的交换操作虽然还不致出错，但毕竟属于多余。那么，是否值得增加一步判断，以避免这类没有实效的原地交换呢？为了做出正确的选择，我们需要切换到一个新的角度，来重新审视选择排序。

① 这里的比较操作使用了严格不等号，以尽可能地减少更新（思齐）操作；
另外，扫描方向是自后向前，如此可以在有多个最大元素时，总是返回其中的最靠后者。

■ 轮换

考查由n个互异元素构成的集合$\mathcal{X}$，将对应的有序序列记作$S[0,n)$。于是，由$\mathcal{X}$中元素排成的任何一个序列$A[0,n)$，都对应于一个一一映射$\tau()$，称作$\mathcal{X}$的一个排列（permutation）：

$$\tau():[0,n)\mapsto[n,0)\text{：对任何}k\in[0,n)\text{，有}S[\tau(k)]=A[k]$$

如果$\mathcal{X}$是由前$n=16$个英文字母组成，那么对应的有序序列就是图3.13中的$S[0,16)$。该图还以一个具体的序列$A[0,16)$为例，逐项列出了映射$\tau()$。



图3.12 序列对应的排列

从任何一个$k\in[0,n)$出发，我们都可以循着排列$\tau()$持续地追索下去：

$$k=\tau^0(k)\ \mapsto\ \tau^1(k)\ \mapsto\ \tau^2(k)\ \mapsto\ \tau^3(k)\ \mapsto\ \dots$$

当然，经过$d\le n$后迟早会出现重复：$\tau^d(k)=k$。于是，便得到一个长度为d的轮换（cycle）：

$$\langle\,\tau^0(k),\,\tau^1(k),\,\tau^2(k),\,\tau^3(k),\,\dots,\,\tau^{d-1}(k)\,\rangle$$

不难理解，任何排列都可以分解为若干个彼此独立的轮换，且其长度总和恰好为n。比如上例中那个长度为16的排列，就可如图3.13，分解为4个轮换，其长度分别为5、3、7、1。



图3.13 排列的轮换分解

一个轮换的长度，介于1到n之间。长度为1时，对应的那个元素必然已经就位；反之亦然。比如在上例中，字母'G'便独自构成了一个长度为1的轮换$\langle 6\rangle$，它已经处于整体排序之后的位置。于是，一个序列所对应的排列可分解为n个长度为1的轮换，当且仅当它有序。

■ 数学期望

回到上一节留下的问题：通过增加判断来避免原地的交换，是否值得呢？结论取决于，这类情况出现的概率有多大。很遗憾，这个概率其实远低于我们的直觉。

现在，从轮换的视角来反观图3.11，每一次交换都意味着什么呢？不难看出，选择排序算法之所以要将m与x交换，乃是因为二者同属一个轮换C，且前后紧邻。于是在交换之后，m将从C中逸出并独自构成一个轮换，C的长度也相应地减一；于是同时，其余的轮换均保持不变。

于是，m与x是原地交换，当且仅当m=x，当且仅当C的长度原本就是1，当且仅当m是其最初所属轮换中残余的最后一个元素。因此，若输入序列原本包含k个轮换，则在选择排序过程中应恰好会出现k次原地的交换。当然，总有$1\le k\le n$，但是k的数学期望呢？

根据数学期望线性律（linearity of expectation），我们只需估计出在每一步迭代中出现这类原地交换的概率，然后累加。实际上，出现这类情况的另一等效的充要条件是，最大元素m恰为无序前缀的末元素；若当时的前缀长度为r，则对应的概率当为$1/r$。由此可知：

$$\mathbb{E}[k]\ =\ \sum_{r=1}^{n}1/r\ =\ \mathcal{O}(\log n)$$

在n步迭代中平均只会出现$\mathcal{O}(\log n)$次这类情况，得不偿失，故完全不必理会。

§3E 插入排序

■ 减而治之

我们都应该玩过各种牌戏。你或许与我一样，习惯于如图3.14那样，始终让手中的牌顺序排列。



图3.14 牌戏中的排序

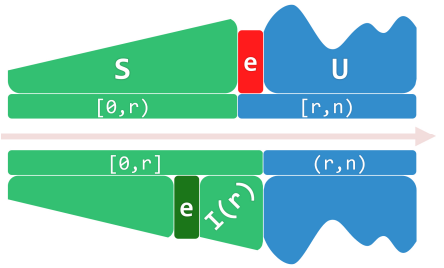


图3.15 插入排序的原理

这样的发牌过程中，每个时刻都如图3.15，满足不变性：

✧ 已拿到的 r 张牌构成一个有序前缀，而尚待拿到的 n - r 张牌则可视作一个无序后缀[①]。

于是依然采用减治策略，实质上需要解决的问题只有一个：

✧ 如何将下一张牌 [r] = e 插入前缀，并使之继续有序。

这样一个反复执行插入操作的算法，称作插入排序（ insertionsort ）。

■ 实现

实际上，有序列表的查找算法3.13[p77]已经为解决这一问题打好了基础。按照接口的约定，List::search(h,e,r)总是能够从当前的有序前缀(h,r)之中，查找到不大于[r] = e的那个最靠后的节点。于是接下来只要将元素e取出，并转移至该节点之后即可。

依照以上思路，便可针对列表实现插入排序算法3.16：

```
1 template <typename T> void List<T>::insertionSort(LNP<T> h, LNP<T> t) {
2    for (LNP<T> r = h->succ, s = r->succ; r != t; r = s, s = r->succ) //反复在有序前缀(h,r)
3       r->moveSucc(search(h, r->data, r)); //内找到不大于[r]的最靠后者，并将[r]移至紧邻其后
4 } //insertionSort
```

算法3.16 列表的插入排序

请注意，有序列表查找接口如上的约定，不仅保证了新节点插入之后前缀依然有序，而且也确保了相等节点之间的次序与它们到达的时序一致——这意味着，该算法是稳定的。

■ 性能分析

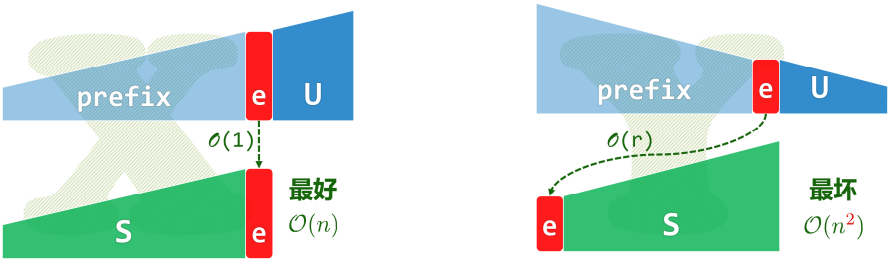


图3.16 插入排序的极端情况：输入序列低开高走（X）或高开低走（Y）

① 相对于选择排序，这里的前、后缀之间并无确定的大小，且互换了角色——后面[算法12.6, p273]将会理解为何如此安排。

最坏情况如图3.16Y，输入序列完全逆序，每个元素都需要遍历其对应的前缀才能完成插入，累计耗时$\sum_{r=0}^{n-1}\mathcal{O}(r)=\mathcal{O}(n^2)$。一般而言，为插入元素$[r]$，期望地需要花费$\mathcal{O}(r/2)=\mathcal{O}(r)$时间，故总体的期望复杂度还是$\mathcal{O}(n^2)$。最好情况则如图3.16X，输入序列已经整体有序，每个元素都是其对应前缀中的最大者，各自只需要常数时间，累计只需$\mathcal{O}(n)$时间。由此可见，插入排序也属于输入敏感的算法——这一点与选择排序不同，尽管二者从形式上看似乎彼此对称。

我们看到，列表插入排序的时间主要消耗于查找，而且是在有序的前缀中查找。然而很遗憾，正如3C节针对算法3.13已经指出的，我们在这里并不能仿效二分查找算法2.14[p53]以获得更高效率。当然，插入排序的思路完全可以推广至向量，但同样很遗憾，尽管向量支持高效率的二分查找，但其插入、删除和移动等动态操作的效率却很低。总之，以目前我们所掌握的数据结构，还不能兼顾插入排序中的静态、动态操作，最坏情况下必然顾此失彼，总是需要$\mathcal{O}(n^2)$时间。

那么在玩牌时，为什么你我都会不约而同地采用插入排序，而从来没有考虑过其他算法呢？这是因为该算法有个难得的优点：它能够随着待排序数据的逐项抵达，及时地予以处理——正如我们每拿到一张牌，总能及时更新手中的序列；待到最后一项出现，便可迅速地完成整个计算。而无论起泡排序、选择排序、归并排序还是后面我们将要讨论的其他排序算法，都需要等到输入数据完全就绪才能启动有效的计算——采用这些算法，我们只有等到所有的牌都已拿到之后才能开始理牌，牌戏也将因此变得索然无趣。如果将后一类算法称作离线算法或脱机算法（offline algorithm），那么插入排序则属于在线算法或联机算法（online algorithm）。

■ 逆序对

作为一个输入敏感的算法，除了极端的最好、最坏情况，插入排序算法在常规情况下的表现如何呢？为回答这个问题，我们需要切换到一个新的视角，来考查所谓的逆序对。

算法2.9[p48]已展示过如何通过遍历统计一个序列的逆序对，而以下将要讨论的同名指标则是对它的推广：在一个序列$S[0,n)$中，若有$0\le i<j<n$且$S[i]>S[j]$，则$\langle i,j\rangle$称作一个逆序对（inversion）。也就是说，任何一对左大右小的元素，无论是否相邻都属此列。

序列S中包含逆序对的数量记作$\mathcal{I}(S)$，这个指标可用来度量S到完全有序状态的距离。比如，S完全有序时，$\mathcal{I}(S)=0$；反过来若S完全逆序，则$\mathcal{I}(S)=\binom{n}{2}=\mathcal{O}(n^2)$。接下来我们将会看到，一个序列的这个指标，与它接受插入排序所需的时间紧密相关。

为便于统计，我们可按如下约定，将$\mathcal{I}(S)$分配到每个元素的账上。对任何$1\le j<n$[①]，令：

$$\mathcal{I}(j)=\Big\|\ \{\ 0\le i<j\ \Big|\ S[i]>S[j]\ \}\ \Big\|$$

也就是说，$\mathcal{I}(j)$记下的是元素$[j]$与其前缀中元素构成的逆序对总数。于是自然地，有：

$$\mathcal{I}(S)=\sum_{r=1}^{n-1}\mathcal{I}(r)$$

① 特别地，无论如何都有$\mathcal{I}(0)=0$。

现在，反观图3.15：对任何$1 \le r < n$，考查为插入元素$[r]$而做的那一次查找。实际上此时，在前缀$[0,r)$中与$[r]$构成逆序对的所有元素，都必然会集中在一起，构成$[0,r)$的一个长度为$\mathcal{I}(r)$的后缀；而这次查找在结束之前，恰好会经历$\mathcal{I}(r)$次失败的比较。因此查找需要$\mathcal{I}(r)+1$的时间，再加上转移操作，共计$\mathcal{I}(r)+2$的时间。于是，该算法累计消耗的时间量应为：

$$\sum_{r=1}^{n-1} (\mathcal{I}(r)+2) = \sum_{r=1}^{n-1} \mathcal{I}(r) + \sum_{r=1}^{n-1} 2 = \mathcal{O}(\mathcal{I}(S)+n)$$

可见，插入排序的性能，直接决定于逆序对的总数。从这个角度来看，排序算法其实都大同小异：无非是先检测出逆序对，再经交换予以修复。当然，检测逆序对的策略还是有高低之别的。起泡排序[p56]、选择排序[p78]、插入排序[p80]只会检测相邻的逆序对，也只能逐个地消除，所以时间复杂度不会低于$\mathcal{I}(S)$，最坏情况下必是$\Omega(n^2)$。相比之下，归并排序[p58]则更加高明：它会尽早地发现相距更远的逆序对，从而经过一次交换可能同时消除更多的逆序对，整体效率更高。

有意思的是，依托归并排序，我们还可以高效率地统计出任意序列中逆序对的总数。在介绍这个方法之前，我们还是先来将此前基于向量的归并排序算法，拓展到列表。

§3F 归并排序

■ 分而治之

尽管依托的数据结构由向量切换为列表，但归并排序算法的基本原理与策略依然不变，仍是分而治之：如图3.17，将列表均分为前缀与后缀，令其各自递归地完成排序，然后二路归并。

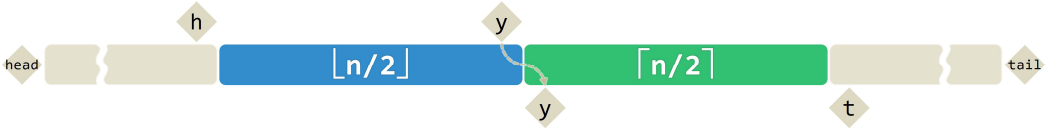


图3.17 列表归并排序的原理

如此，便可直接导出列表的归并排序算法3.17。

```
1 template <typename T> void List<T>::mergeSort(LNP<T> h, LNP<T> t) {
2    Rank n = *t - h - 1; //算出区间长度（线性时间）
3    if (n < 2) return; //不足2个元素时，直接返回；否则...
4    LNP<T> y = *h + n/2; //确定切分点：(h,t) = (h,y] + (y,t)（线性时间）
5    mergeSort(y, t); //后缀与
6    mergeSort(h, y = y->succ); //前缀各自排序
7    merge(h, y, t); //归并：(h,y) + [y,t)
8 } //mergeSort
```

算法3.17 列表的归并排序

当然，这里的实质任务也是一样。整个算法能够成功，取决于二路归并是否可行；算法整体效率的高低，取决于归并效率的高低。我们的目标是，与向量一样，在线性时间内完成归并。

■ 二路归并

如图3.18（上），这里待归并的子列表，都是在同一列表中，相邻的一对有序区间(h,y)和[y,t)。尽管与有序向量的二路归并[p59]本质原理相同，但有序列表的二路归并还是有些微妙的区别。比如，这里归并的效果可以理解为立足于前缀(h,y)，依次将后缀[y,t)中的节点转至前者中适当的位置。为此只要调整节点之间的链接，无需任何辅助空间。

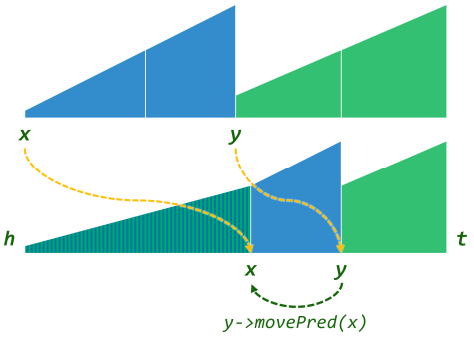


图3.18 有序列表的二路归并

具体地，我们分别用x和y来指示两个列表中的当前节点。若节点y更小，则如图（下），将y取出并转移至节点x之前①；反之，则只需令x指向下一个节点。这一思路可实现为算法3.18。

```
template <typename T> void List<T>::merge(LNP<T> h, LNP<T> y, LNP<T> t) { //(h,y) + [y,t)
   for (LNP<T> x = h->succ; (x != y) && (y != t); )
      if (x->data > y->data) //小者优先；相等时x优先，以保证稳定性
         (y = y->succ)->pred->movePred(x);
      else
         x = x->succ;
} //O(n+m)
```

算法3.18 有序列表的二路归并

可见，列表归并的另一便利之处在于，任何一个子列表耗尽，归并都可以随即结束，而不必像向量那样等到二者都已耗尽。列表在此处展现出来的优势，都得益于循位置访问的方式。

当然，循位置访问也不是尽善尽美的。比如，主算法3.17在划分子列表时，已不能如向量的算法2.18[p58]那样在常数时间内确定切分点。尽管算法3.3[p72]可以通过重载加号操作符简化描述，但并不能改变$\mathcal{O}(n)$的复杂度。幸好二路合并本身也要$\mathcal{O}(n)$时间，才不致拖累整体的效率。

综上，列表的归并排序与向量版一样，渐近时间复杂度依然是：

$$\mathrm{T}(n) = 2 \cdot \mathrm{T}(n/2) + \mathcal{O}(n) = \mathcal{O}(n \log n)$$

最后请确认，二路归并中的比较操作是合适的，可以保证列表归并排序的稳定性。

■ 统计逆序对

我们在3E节[p81]已经看到，逆序对的数量$\mathcal{I}(S)$刻画了序列S的混乱程度，也直接决定了插入排序等算法的复杂度。那么对于任意的序列，如何统计出这一关键的指标呢？若由其定义出发，逐个地比较每一对$\langle i, j \rangle$并做计数，则在最坏情况下需要$\Omega(n^2)$时间。能否更快呢？

可以的。为此，我们需要再次采取分治策略。

① 这与插入排序颇为相似：同样是一个有序的前缀，以及一个新插入的节点y。不同之处在于，为了确定后缀中每一个元素在前缀中的插入位置，这里不必整体查找，而只需从x——前一次插入的位置——继续查找。

如图3.19，如果将序列分为前缀L=[lo,mi)与后缀R=[mi,hi)，那么所有的逆序对无非两类。有些逆序对涉及的两个元素都来自于L，或者都来自于R。而另一些逆序对$\langle i, j \rangle$则会跨越前缀、后缀，即$lo \le i < mi \le j < hi$。前一类逆序对可通过递归来统计，而后一类的计数才是核心问题。

按照前面的约定，后一类逆序都应计入后缀R中各元素的账上。将每个元素$S[j]$此项账单的长度记作$\mathcal{I}(j) = \| \{lo \le i < mi \mid S[i] > S[j]\} \|$，$mi \le j < hi$，如何统计出这些$\mathcal{I}(j)$呢？

如果还是逐对比较，蛮力地统计，则总体的时间成本将是：

$$\mathrm{T}(n) = 2 \cdot \mathrm{T}(n/2) + n/2 \cdot n/2 = \mathcal{O}(n^2)$$

与直接地逐对比较，并无实质的不同。如何统计得更快呢？可以借鸡生蛋，也就是套用归并排序。

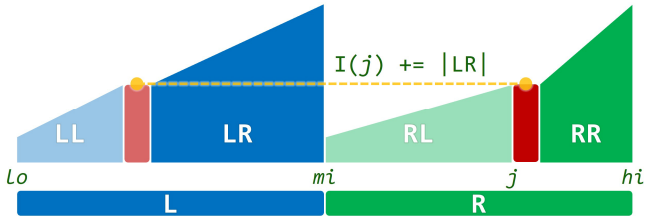


图3.19 通过分治统计逆序对的原理

假想着对序列S做归并排序。还是在如图3.19的二路归并阶段，让我们定格在$S[j]$就要出列、即将归入汇总序列的那一刹那。此时，前缀L中不大于它的元素（L的前缀LL，以及与$S[j]$相等者），都应该已经出列过了；反过来，仍然残留（在L的后缀LR中）的，恰好就是严格更大的哪些元素[①]。于是，只需要花常数时间丈量出此时此刻LR的宽度，即是$\mathcal{I}(j)$。

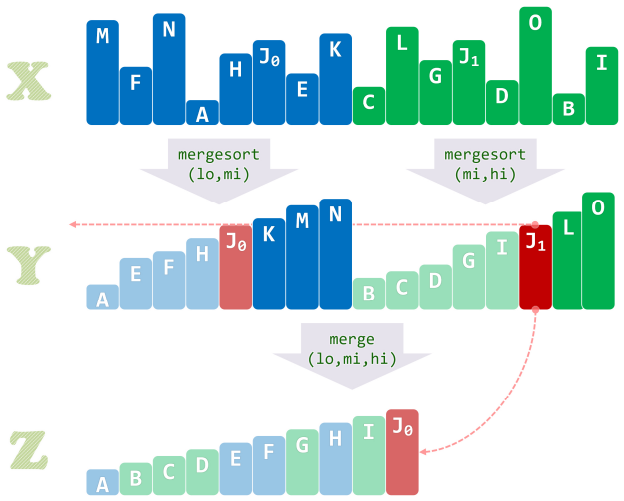


图3.20 通过分治统计逆序对的实例

以图3.20X为例，由16个英文字母构成的序列，其中包含两个重复的J。

接下来如图Y，前缀（蓝）、后缀（绿）已经分别递归地排序，并统计出了各自内部的逆序对总数$\mathcal{I}_{[0,8)}$、$\mathcal{I}_{[8,16)}$。以下，再通过二路归并完成整体的排序，并且统计出$\sum_{j=8}^{15} \mathcal{I}(j)$。三项累加起来，即是原序列中逆序对的总数$\mathcal{I}(S)$。

图Z就是后缀中元素J被选中，即将出列的定格时刻。能看到，前缀中所有不大于它的元素（含与之相等的那个J）均已出列；仍然残留在前缀中的，恰好就是原本就严格大于它的K、M和N。

综上，套用归并排序的框架，只需增加微不足道的时间成本，便可对任何序列完成逆序对的统计，渐近复杂度与归并排序相同，仍是$\mathcal{O}(n \log n)$。

① L内的元素尽管在此前可能移动过，但都限于L内部。故知此时LR内的元素，正是初始时大于$S[j]$者，既不多，也不少。