第十章 内部排序

10.4 选择排序

一. 简单选择排序

- 选择排序
  - 选择排序是每次从当前待排序的记录中选取关键字最小（或最大）的记录，然后与待排序的记录序列中的第一个记录进行交换，直到整个记录序列有序为止。
  - 简单选择排序(Simple Selection Sort ，又称为直接选择排序)的基本操作是：每一趟(例如第i趟,i=1,2,…,n-1)通过n-i次比较，在n-i+1个待排序记录中选出关键字最小的记录,与第i个记录交换。

10.4 选择排序

一. 简单选择排序

- 举例

	1	2	3	4	5	6	
初始	21	25	49	25*	16	08	
	21	25	49	25*	16	08	最小者 08 交换21, 08
i=1	08	25	49	25*	16	21	最小者 16 交换25, 16
i=2	08	16	49	25*	25	21	最小者 21 交换49, 21

10.4 选择排序

一. 简单选择排序

- 举例

i =3	08	16	21	25*	25	49	最小者 25* 不需交换
i =4	08	16	21	25*	25	49	最小者 25 不需交换
i =5	08	16	21	25*	25	49	结束

10.4 选择排序

一. 简单选择排序

- 算法实现：

```
void SelectSort( SqList &L )  // 算法10.9
{  // 对顺序表L作简单选择排序
  for ( i=1; i<L.length; ++i)  // 选择第i小的记录，并交换到位
  {
     // 在L.r[i..L.length]中选择最小的记录并将其位置赋给MiniPos
     MinPos = SelectMinKey( L, i );

     // 将未排序部分的最小记录换到有序部分的最后位置i
     if ( i != MinPos  )
               L.r[i]←→L.r[MinPos ];     }
}
```

10.4 选择排序

一. 简单选择排序

- 算法分析：
  - 整个算法是二重循环：
  - ✓ 外循环控制排序的趟数，对n个记录进行排序的趟数为n-1趟；
  - ✓ 内循环控制每一趟的排序，进行第i趟排序时，关键字比较次数KCN 与记录的初始排列无关，所需的比较次数总是 n-i次。总的关键字比较次数为

$$KCN = \sum_{i=1}^{n-1} (n-i) = n(n-1)/2$$

记录移动次数RMN与记录的初始排列有关，最好情况是记录已经有序，RMN=0，达到最少；最坏情况是每一趟都要进行交换，总的记录移动次数为 RMN = 3(n-1)。

  - 时间复杂度是**O(n²)**，空间复杂度是**O(1**）

10.4 选择排序

一. 简单选择排序

- 直接选择排序是一种不稳定的排序方法，例如：

初始序列 5 ，3， 6， 9， 8， 2， 5*， 1

排序后 1， 2， 3， 5 *， 5， 6， 8， 9

10.4 选择排序

二. 堆排序

- 背景

直接选择排序算法中， 进行第i趟排序时，关键字所需的比较次数总是 n-i次，原因是没有保存之前的比较结果，会重复执行很多比较操作。如果可以做到每次在选择到最小记录的同时，并根据比较结果对其他记录做出相应的调整，那样就会提高排序的总体效率。

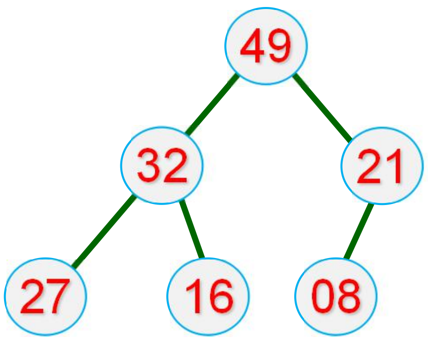
试想，如果能把待排序的数据元素集合构成一个完全二叉树结构，则每次选择出一个最大（或最小）的数据元素值需比较完全二叉树的高度次，即logn次，则排序算法的时间复杂度就是O(nlogn)。

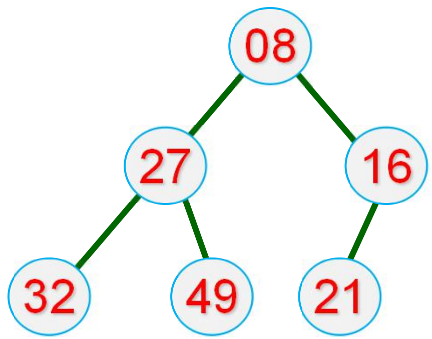
10.4 选择排序

二. 堆排序

设有一个关键字集合，按完全二叉树的顺序存储方式存放在一个一维数组中。对它们从根开始，自顶向下、自左向右从 1 开始连续编号，

- 若满足 $K_i \geq K_{2i}$ && $K_i \geq K_{2i+1}$，则称该关键字集合构成一个最大堆（大顶堆）。
- 若满足 $K_i \leq K_{2i}$ && $K_i \leq K_{2i+1}$，则称该关键字集合构成一个最小堆（小顶堆）。





- 由此可见，堆的根结点必定是最大值或最小值

10.4 选择排序

二. 堆排序

- 堆排序：利用堆顶记录的关键字值最小(或最大)的性质，从当前待排序的记录中依次选取关键字最小(或最大)的记录，就可以实现对数据记录的排序，这种排序方法称为堆排序。
- 若采用最小堆，排序后得到的是非递增序列；
- 若采用最大堆，排序后得到的是非递减序列。

10.4 选择排序

二. 堆排序

- 算法思想：
  - 对一组待排序的记录，按堆的定义建立初始堆；
  - 将堆顶记录和最后一个记录交换位置，则前n-1个记录是无序的，而最后一个记录是有序的；
  - 堆顶记录被交换后，前n-1个记录不再是堆，需将前n-1个待排序记录重新组织成为一个堆，然后将堆顶记录和倒数第二个记录交换位置，即将整个序列中次小（或次大）关键字值的记录调出无序区，而进入有序区；
  - 重复上述步骤，直到全部记录排好序为止。

10.4 选择排序

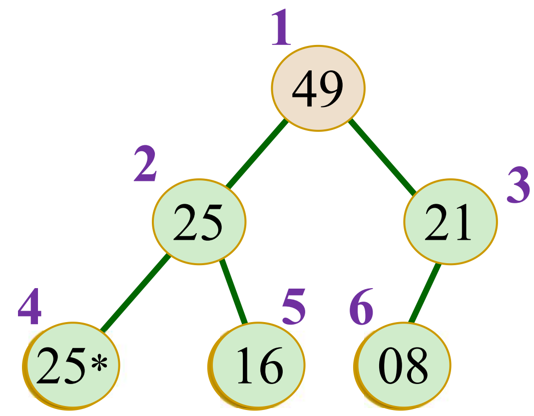
二. 堆排序

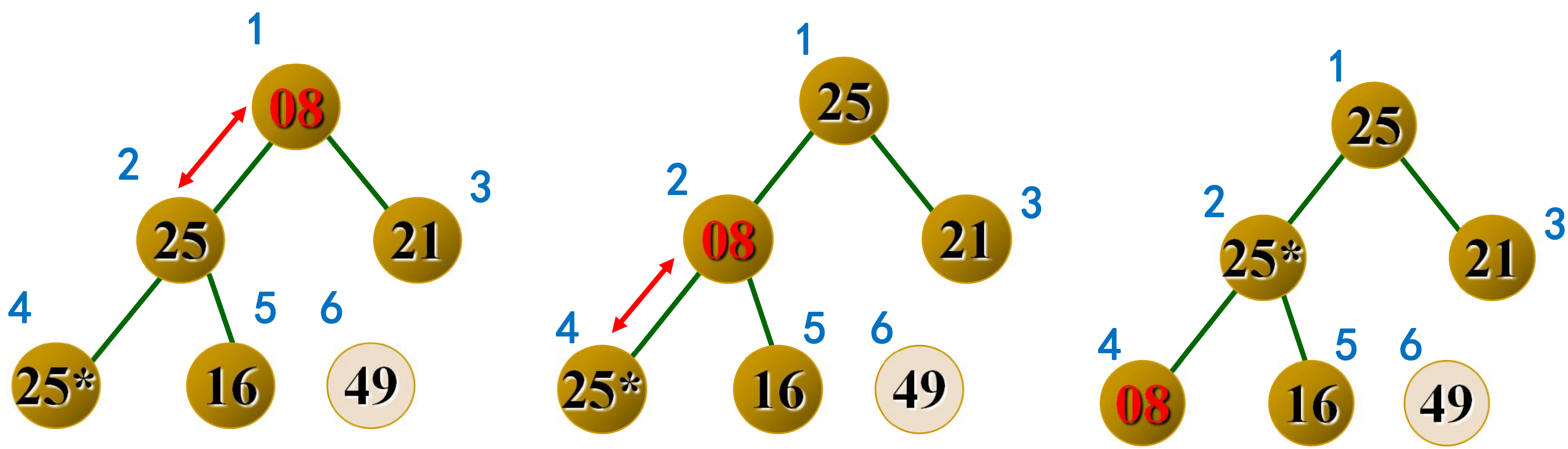
- 算法实现的主要问题：
  - 如何根据给定的序列建初始堆？
  - 如何在交换掉根结点后，将剩下的结点调整为新的堆(筛选)？

10.4 选择排序

二. 堆排序

- 问题二：筛选方法





10.4 选择排序

二. 堆排序

- 问题二：筛选方法
  - 输出根结点
  - 用序列中最后一个结点代替根结点值
  - 比较当前根结点与两个孩子结点的值，如果小于其孩子结点，则选择值大的孩子结点与根结点交换
  - 继续将交换的结点与其孩子结点比较
  - 重复上述操作，直到叶子结点或者根结点值大于两个孩子结点，将得到新的堆。称这个从堆顶至叶子的调整过程为“筛选”。

10.4 选择排序

二. 堆排序

- 问题一：创建初始堆

利用筛选算法，可以将任意无序的记录序列建成一个堆：

- 根据给定的序列，从1至n按顺序存储创建一个完全二叉树
- 将二叉树的每棵子树都筛选成为堆。只需从最后一棵孩子不为空的子树开始进行筛选，即从第 $\left\lfloor \frac{n}{2} \right\rfloor$ 个记录到第1个记录依次进行筛选就可以建立堆。
  - 只有根结点的树是堆；
  - 由二叉树的性质5，最后一个非终端结点的序号为 $\left\lfloor \frac{n}{2} \right\rfloor$ ，也就是说第 $\left\lfloor \frac{n}{2} \right\rfloor$ 个结点之后的所有结点都是叶子结点，只有根而没有子树。

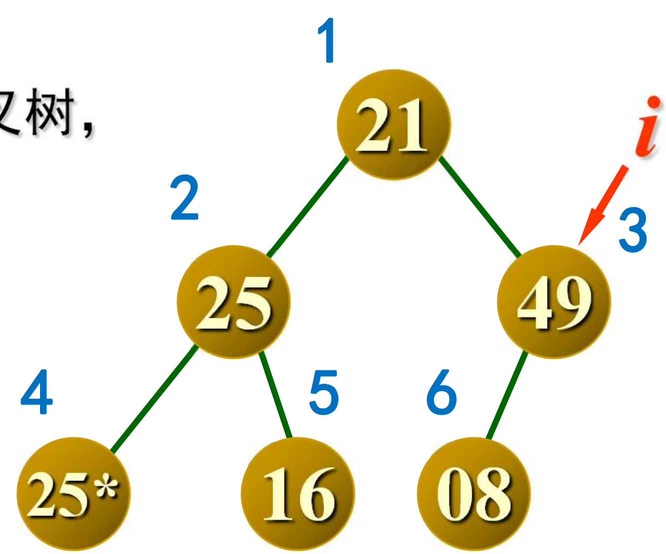
10.4 选择排序

二. 堆排序

- 创建初始堆举例
  - 已知待排序的一组记录的初始排列为：21, 25, 49, 25*, 16, 08

21	25	49	25*	16	08

按编号1到n创建完全二叉树，若不是堆，调整成堆。

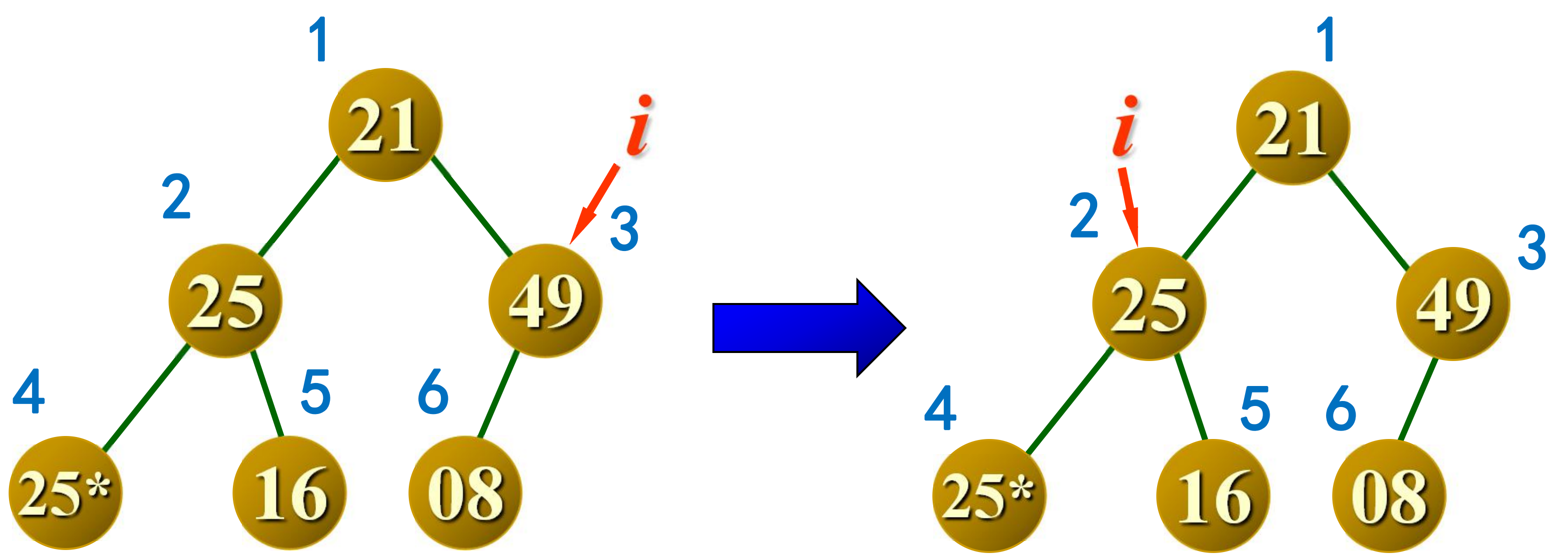


10.4 选择排序

二. 堆排序

- 创建初始堆举例
  - i=3时，结点49比它的叶子结点大，无须调整

调整成初始堆：



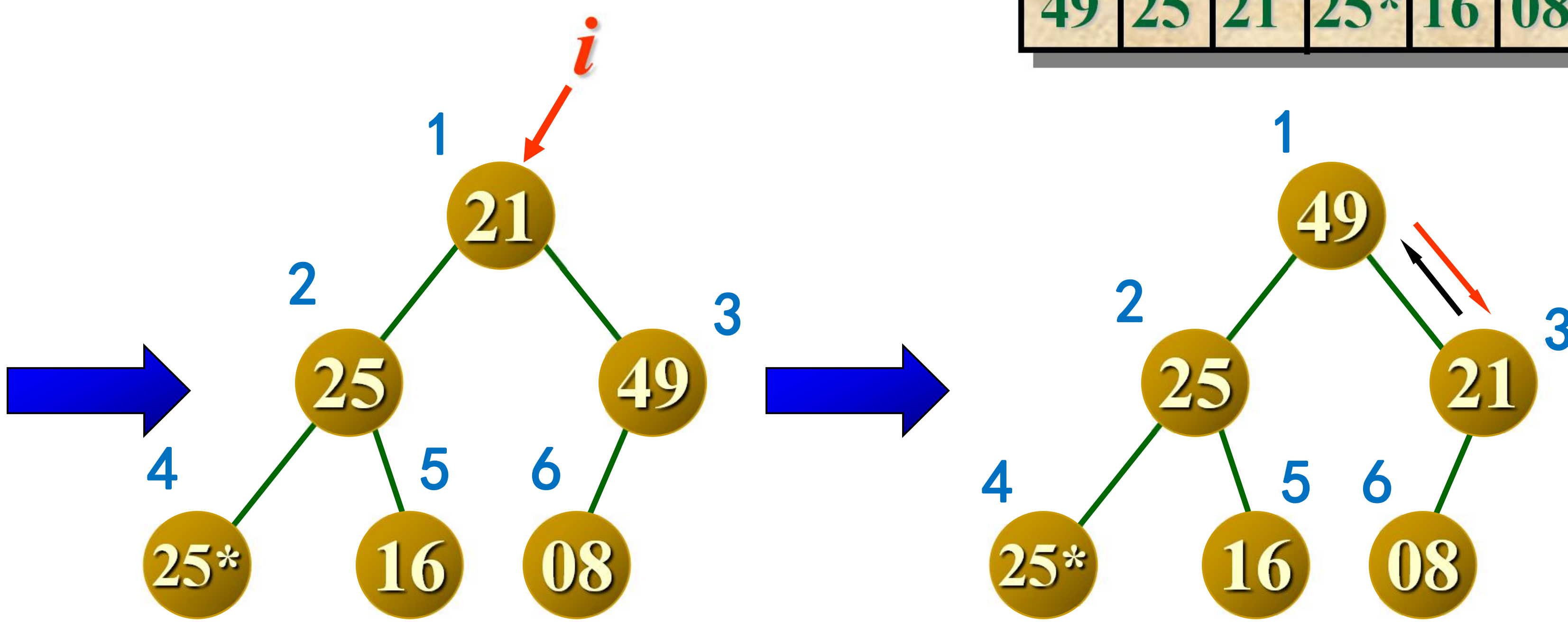
10.4 选择排序

二. 堆排序

- 创建初始堆举例
  - i=2时，结点比它的叶子结点大，无须调整
  - i=1时，发生局部调整

初始堆创建完成：

49	25	21	25*	16	08



10.4 选择排序

二. 堆排序

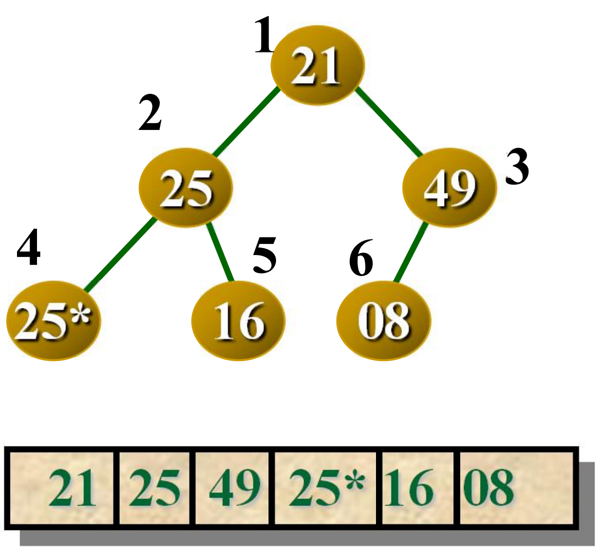
- 堆排序算法流程

1. 将初始序列从1至n按顺序创建一个完全二叉树
2. 将完全二叉树调整为堆
3. 用最后结点代替根结点值，输出最后结点
4. 重复步骤2，直到输出最后一个结点

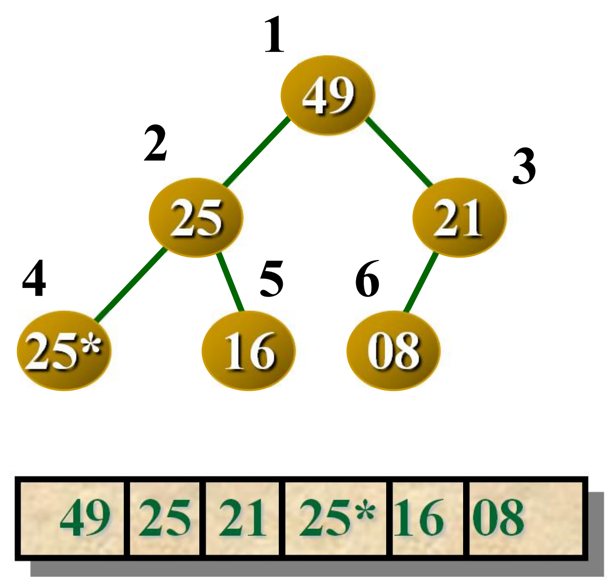
10.4 选择排序

二. 堆排序

- 堆排序举例
  - 已知待序的一组记录的初始排列为：21, 25, 49, 25*, 16, 08
  - 创建初始最大堆



初始二叉树

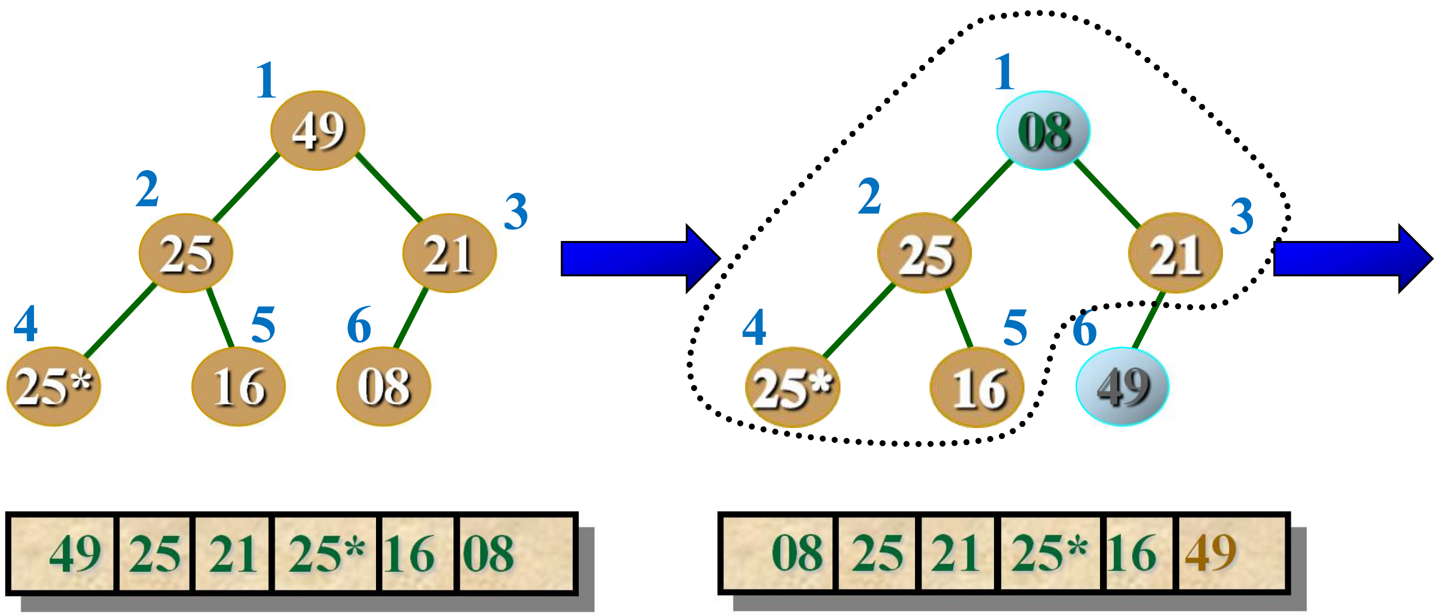


初始最大堆

10.4 选择排序

二. 堆排序

- 堆排序举例
  - 经过初始最大堆后，根结点为最大值49，交换到末尾，然后排除末尾结点重建新的最大堆



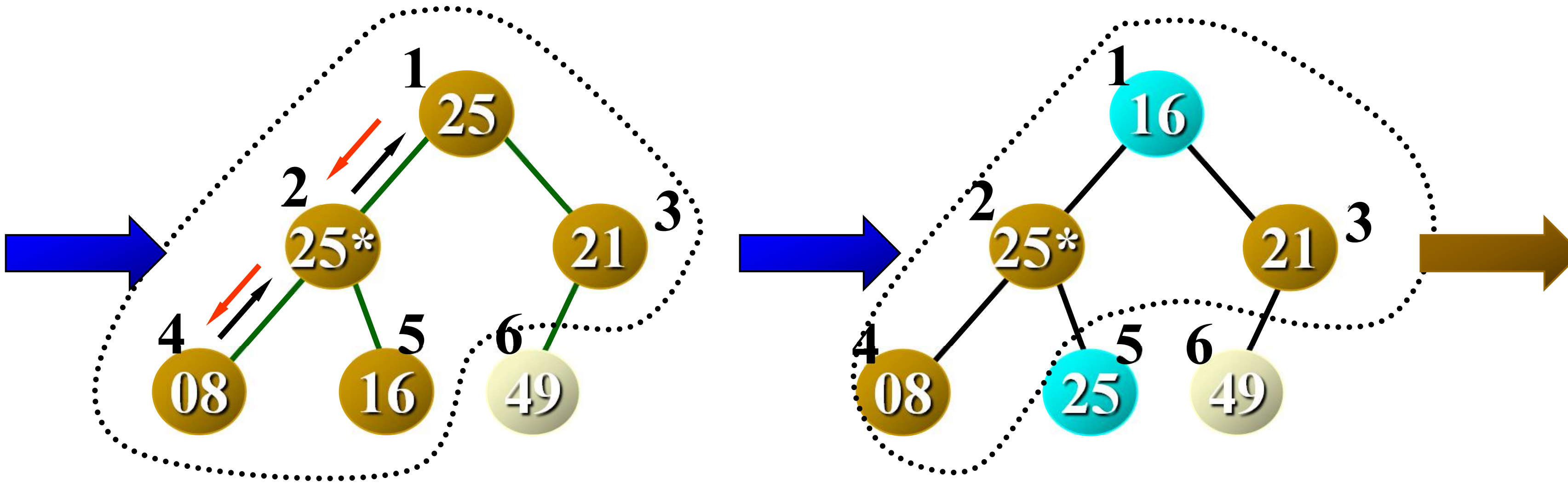
初始最大堆

交换 1 号与 6 号记录，6号记录就位

10.4 选择排序

二. 堆排序

- 堆排序举例



25	25*	21	08	16	49

从 1 号到 5 号 重新
调整为最大堆

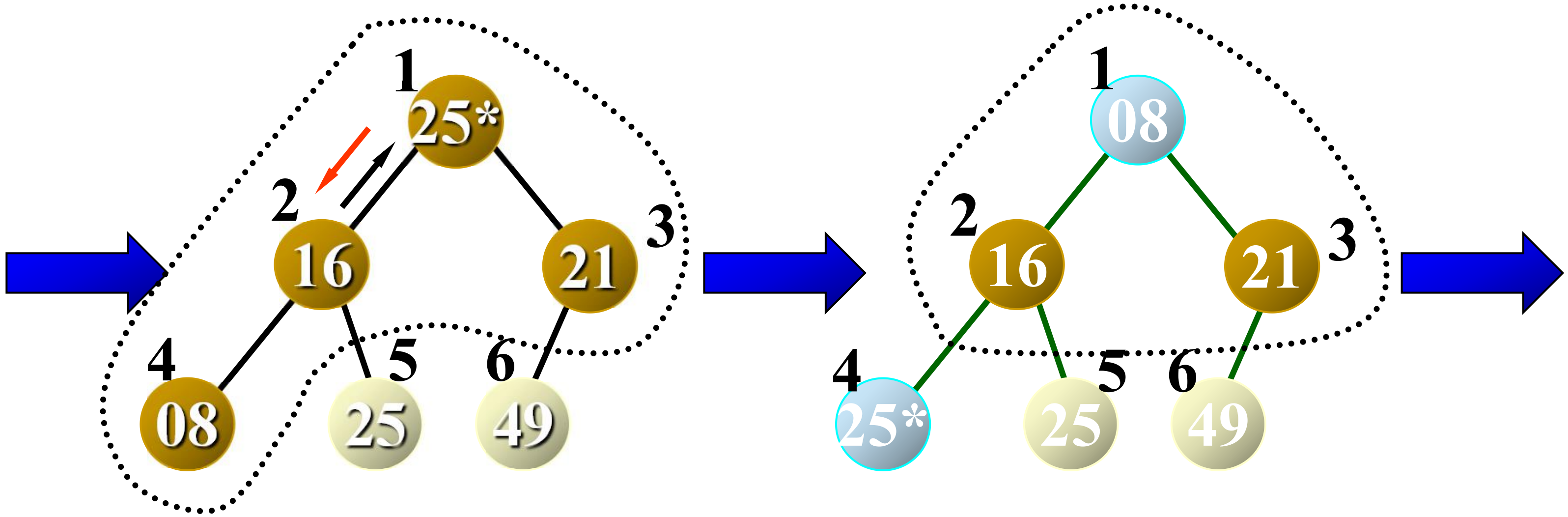
16	25*	21	08	25	49

交换 1 号与 5号记录，
5 号记录就位

10.4 选择排序

二. 堆排序

■ 堆排序举例



25*	16	21	08	25	49

从 1 号到 4 号 重新调整为最大堆

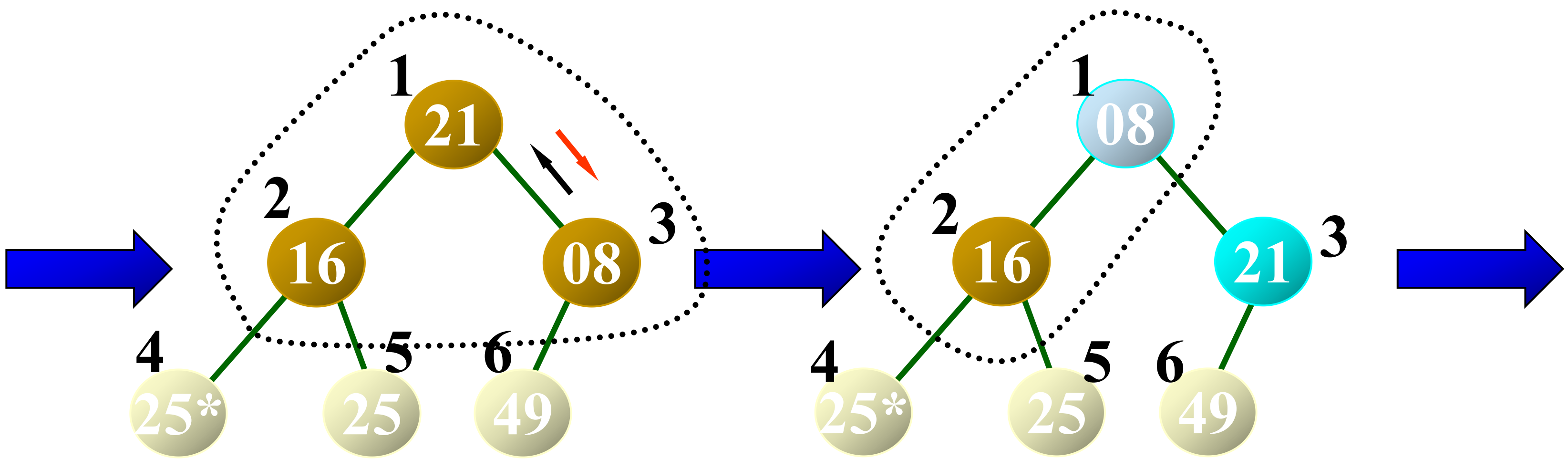
08	16	21	25*	25	49

交换 1 号与 4 号记录，4 号记录就位

10.4 选择排序

二. 堆排序

- 堆排序举例



21	16	08	25*	25	49

从 1 号到 3 号 重新调整为最大堆

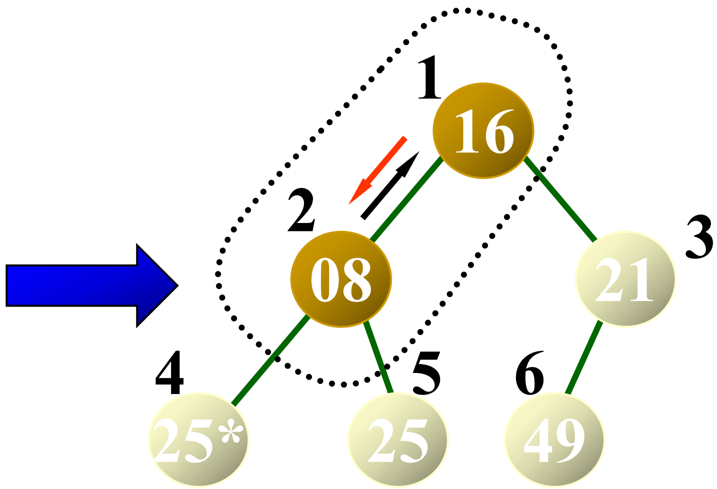
08	16	21	25*	25	49

交换 1 号与 3 号记录，3 号记录就位

10.4 选择排序

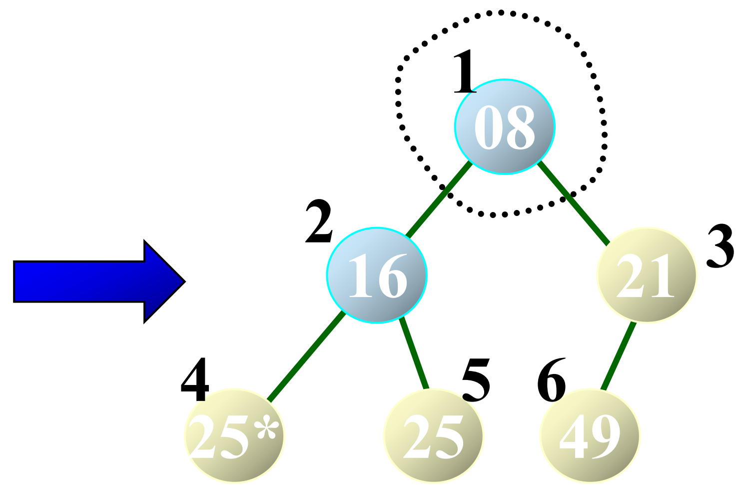
二. 堆排序

- 堆排序举例



16	08	21	25*	25	49

从 1 号到 2 号 重新调整为最大堆



08	16	21	25*	25	49

交换 1 号与 2 号记录，所有记录就位

二. 堆排序

- 算法实现：

```
void HeapAdjust(HeapType &H, int s, int m) {
  // 已知H.r[s..m]中记录的关键字除H.r[s].key之外均满足堆的定义，
  // 本函数调整H.r[s]的关键字，使H.r[s..m]成为一个大顶堆
  // （对其中记录的关键字而言）
  int j;
  RedType rc;
  rc = H.r[s];
  for (j=2*s; j<=m; j*=2) {   // 沿key较大的孩子结点向下筛选
    if (j<m && H.r[j].key<H.r[j+1].key) ++j; // j为key较大的记录的下标
    if (rc.key >= H.r[j].key) break;          // rc应插入在位置s上
    H.r[s] = H.r[j];  s = j;
  }
  H.r[s] = rc;  // 插入
} // HeapAdjust
```

10.4 选择排序

二. 堆排序

- 算法实现：

```
void HeapSort(HeapType &H) {
    // 对顺序表H进行堆排序
    int i;
    RedType temp;
    for (i=H.length/2; i>0; --i)  // 把H.r[1..H.length]建初始堆
      HeapAdjust( H, i, H.length );
    for (i=H.length; i>1; --i) {
      temp=H.r[i];
      H.r[i]=H.r[1];
      H.r[1]=temp;  // 将堆顶记录和当前未经排序子序列Hr[1..i]中
                    // 最后一个记录相互交换
      HeapAdjust(H, 1, i-1);  // 将H.r[1..i-1] 重新调整为堆
     }
} // HeapSort
```

10.4 选择排序

二． 堆排序

- 算法性能：
  - 堆排序时间主要耗费在初始建堆和筛选调整建新堆上
  - 对于长度为n的序列，其对应的完全二叉树的深度为k $(2^{k-1} \le n < 2^k)$，$k = \lfloor \log_2 n \rfloor + 1$
  - 初始建堆：是从完全二叉树最下层最右边的非终端结点开始构建，将它与其孩子进行比较和若有必要的互换，对于每个非终端结点来说，最多进行两次比较和互换操作，因此整个构建堆的时间复杂度为O(n)。
  - 筛选调整：在正式排序时，第i次取堆顶记录重建堆需要用O(logi)的时间（某个结点到根结点的距离为（$\lfloor \log_2 i \rfloor + 1$）），并且需要取n－1次堆顶记录，因此，重建堆的时间复杂度为O(nlogn)。
  - 堆排序时间复杂度为0(n *log*n)，空间复杂度为0(1)。
  - 堆排序是一个不稳定的排序方法。

练习

一. 已知数列为4、8、9、7、5*、5，要对数列进行从小到大的排序，采用堆排序，要求写出每次调整形成的堆。

10.5 归并排序

一. 归并排序

- 归并算法思路：
  - 归并是指将两个或两个以上的有序序列合并成一个有序序列。
  - 2-路归并排序
    - 初始时，将每个记录看成一个单独的有序序列，则n个待排序记录就是n个长度为1的有序子序列；
    - 将前后相邻的两个有序序列归并为一个有序序列（两两归并），得到$\lceil \frac{n}{2} \rceil$个长度为2或1的有序子序列——一趟归并；
    - 重复做两两归并操作，直到得到长度为n的有序序列为止。

其核心是如何将相邻的两个子序列归并成一个子序列。

10.5 归并排序

二. 2一路归并排序

- 举例：

初始关键字： [23] [38] [22] [45] [23*] [67] [31] [15] [41]

一趟归并后： [23 38] [22 45] [23* 67] [15 31] [41]

二趟归并后： [22 23 38 45] [15 23* 31 67] [41]

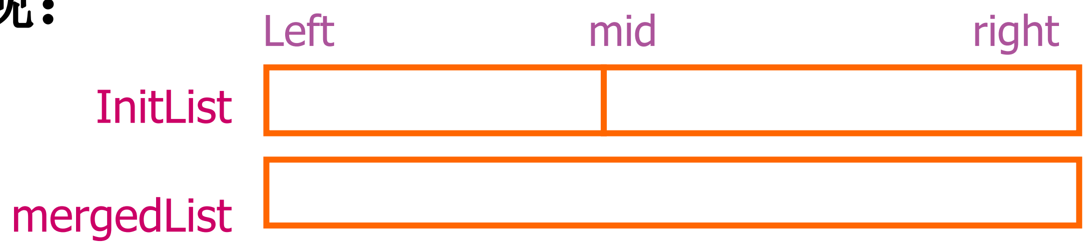
三趟归并后： [15 22 23 23* 31 38 45 67] [41]

四趟归并后： [15 22 23 23* 31 38 41 45 67

10.5 归并排序

一． 归并排序

- 2-路归并的算法实现：




```
typedef   int  SortData;
void Merge ( SortData InitList[ ],  SortData mergedList[ ],
                      int left, int mid,  int right ) {
   int i = left,  j = mid+1,  k = left;
   while ( i <= mid && j <= right )                                //两两比较将较小的并入
      if ( InitList[i] <= InitList[j] )  { mergedList [k] = InitList[i]; i++;  k++; }
      else                             { mergedList [k] = InitList[j]; j++;  k++; }
   while (i <= mid)
         { mergedList[k] = InitList[i];  i++;  k++; }//将mid前剩余的并入
   while (j <= right)
         { mergedList[k] = InitList[j];  j++;  k++; }//将mid后剩余的并入
}
```

10.5 归并排序

二. 2—路归并排序

- 程序实现:

```
void MSort(RcdType SR[], RcdType TR1[], int s, int t) { // 算法10.13
  // 将SR[s..t]归并排序为TR1[s..t]
  int m;
  RcdType TR2[20];
  if (s==t) TR1[t] = SR[s];
  else {
    m=(s+t)/2;          // 将SR[s..t]平分为SR[s..m]和SR[m+1..t]
    MSort(SR,TR2,s,m);    // 递归地将SR[s..m]归并为有序的TR2[s..m]
    MSort(SR,TR2,m+1,t);  // 将SR[m+1..t]归并为有序的TR2[m+1..t]
    Merge(TR2,TR1,s,m,t); // 将TR2[s..m]和TR2[m+1..t]归并到TR1[s..t]
  }
} // MSort

void MergeSort(SqList &L) {  // 算法10.14
 // 对顺序表L作归并排序
 MSort(L.r, L.r, 1, L.length);
} // MergeSort
```

10.5 归并排序

一. 归并排序

- 有序表归并性能分析：
  - 假设待归并的两个有序表长度分别为m和n，则归并后，新的有序表长度为m+n, 若是顺序存储，则需要额外空间m+n
  - 归并操作至多只需要m+n次移位和m+n次比较，因此归并的时间复杂度为O(m+n)

10.5 归并排序

二．2－路归并排序

- 性能分析：
  - 如果待排序的记录为n个，则需要做$\lceil \log_2 n \rceil$趟2－路归并排序，每趟2－路归并排序的时间复杂度为O(n)，因此2－路归并排序的时间复杂度为O($n\log_2 n$)。
  - 需要额外空间，大小与待排序记录空间相同，则空间复杂度为O(n)
  - 归并排序是一种稳定的排序方法

练习

一. 设关键字序列为（ **4，8，5*，7，5，3，2，9，6** ）的一组记录，请给出**2**路归并排序的每一趟结果。

10.6 基数排序

一. 多关键字的排序

- 有时候排序不仅只有1个关键字，可能包含多个

例：对52张扑克牌按以下次序排序：

♣2<♣3<......<♣A<♦2<♦3<......<♦A<
♥2<♥3<......<♥A<♠2<♠3<......<♠A

两个关键字：花色（♣<♦<♥<♠）
面值（2<3<...<A）

并且“花色”地位高于“面值”

你会怎样排序?

10.6 基数排序

一. 多关键字的排序

- 基数排序(Radix Sorting)：按待排序记录的关键字的组成成分(或“位”)进行排序。

  基数排序和前面的各种内部排序方法完全不同，不需要进行关键字的比较和记录的移动。借助于多关键字排序思想实现单逻辑关键字的排序。

10.6 基数排序

一. 多关键字的排序

- 设有n个记录$\{R_1, R_2, \cdots, R_n\}$，每个记录$R_i$含有d个关键字，即关键字形如：$\{K_i^1, K_i^2, \cdots, K_i^d\}$ ($d>1$)，则称序列$\{R_1, R_2, \cdots, R_n\}$有序的，指的是$\forall i, j \in [1, n], i<j$，关键字满足

  $\{K_i^1, K_i^2, \cdots K_i^d\} < \{K_j^1, K_j^2, \cdots K_j^d\}$，即$K_i^p \leqslant K_j^p$ ($p=1, 2, \cdots d$)，

  其中，K^1是最主位关键字，K^d是最次位关键字。

10.6 基数排序

一. 多关键字的排序

- 多关键字排序思想
  - 最高位优先(MSD,Most Significant Digit first):
    - 从最高位关键字K^1起进行排序，将记录序列分成若干个子序列，每个子序列有相同的K^1值；
    - 分别对每个子序列再按K^2进行排序，每个子序列又被分成若干个更小的子序列；……
    - 如此重复，直到按最后一个关键字K^d进行排序。
    - 最后，将所有的子序列依次联接成一个有序的序列。

10.6 基数排序

一. 多关键字的排序

- 多关键字排序思想
  - 最低位优先(LSD, Least Significant Digit first)
    - 对所有记录，先按照最低位关键字K^d进行排序，形成一个新的序列；
    - 然后对所有记录，再按照高一位的关键字K^{d-1}排序，形成一个新的序列；
    - 依次重复，直至对最高位关键字K^1排序后，便成为一个有序序列。

10.6 基数排序

一. 多关键字的排序

- 基数排序：设有n个待排序记录{R_1, R_2, …, R_n}， (单)关键字是由确定的d位(部分)组成，每位有确定的radix种取值，则按照关键字的不同值将记录“分配”到radix个队列中后再“收集”，如此重复d次，可以得到记录的有序序列。
- 可以采用MSD或LSD方法，借助“分配”和“收集”对单逻辑关键字进行排序。

10.6 基数排序

一. 多关键字的排序

- 最低位优先法LSD举例

	0	1	2	3	4	5
	21	25	49	25*	16	08
最低位(个位)排序后	21	25	25*	16	08	49
最高位(十位)排序后	08	16	21	25	25*	49

10.6 基数排序

一. 多关键字的排序

■ 算法实现

(1) 链式基数排序方法，首先以链表存储n个待排序记录；

(2) 一趟排序的过程是：

① 分配： 按K^d值的升序顺序，改变记录指针，将链表中的记录结点按次序分配到r个链队列中，每个链队列中所有记录的关键字的最低位K^d的值都相等，用f[i]、e[i]作为第i个链队列的头指针和尾指针；

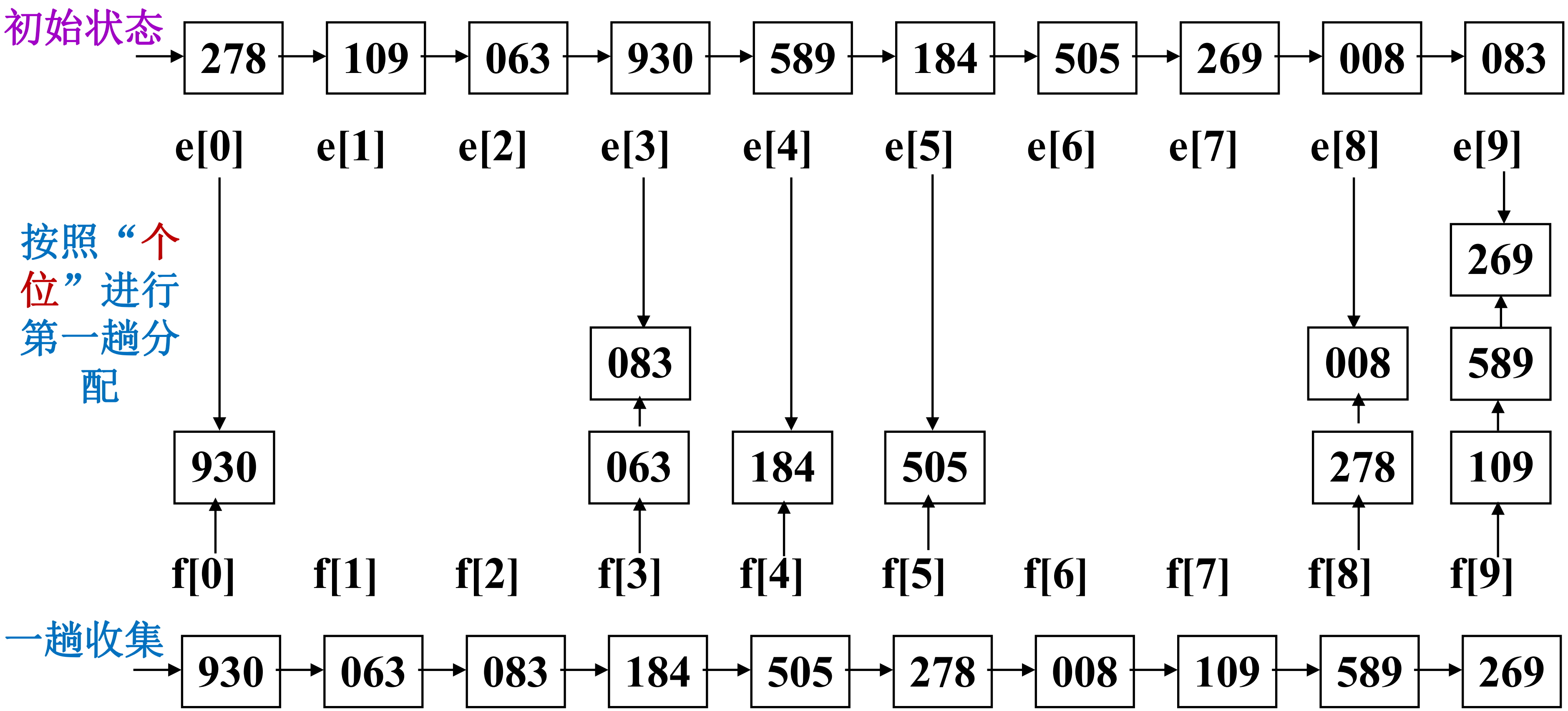
② 收集：改变所有非空链队列的队尾记录的指针域，使其指向下一个非空链队列的队头记录，从而将r个链队列中的记录重新链接成一个链表；

(3) 如此依次再按K^{d-1}, K^{d-2}, … K^1分别进行，共进行d趟后即完成排序。

10.6 基数排序

二. 链式基数排序

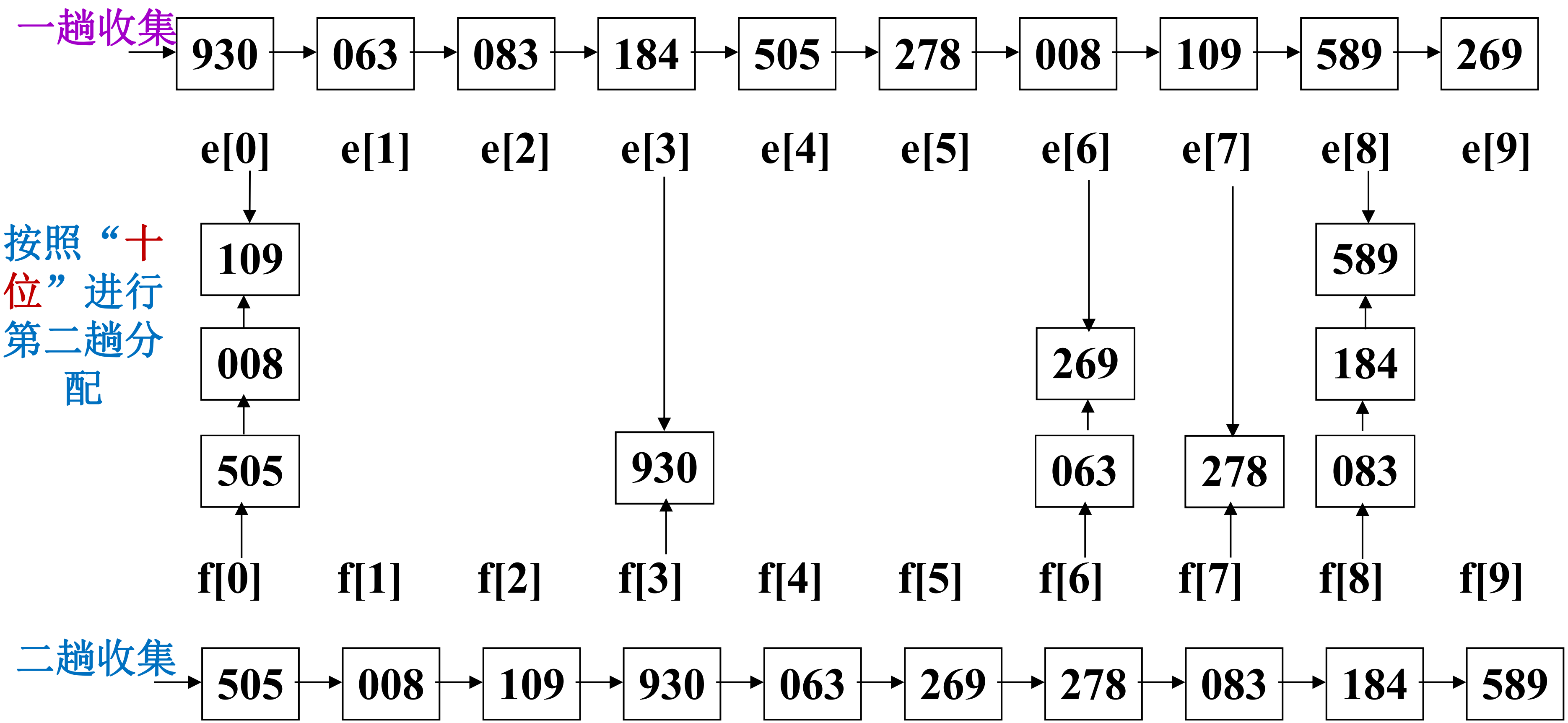
- 举例，假设关键字都是3位整数



10.6 基数排序

二. 链式基数排序

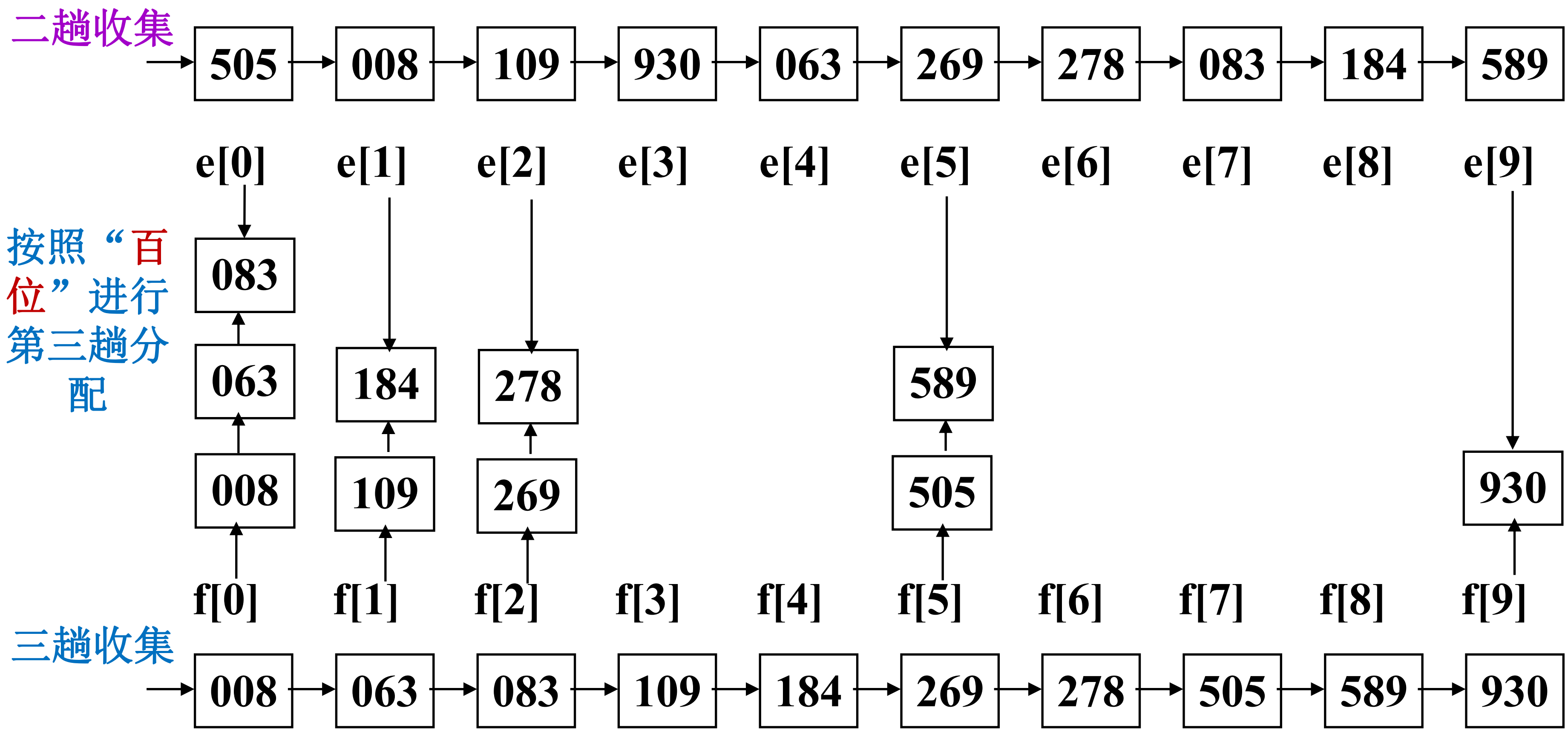
- 举例



10.6 基数排序

二. 链式基数排序

- 举例



10.6 基数排序

二. 链式基数排序

- 程序实现

```
void RadixSort( SLList &L ) { // 算法10.17
  // 对L作基数排序，使得L成为按关键字自小到大的有序静态链表，L.r[0]为头结点
  ArrType  f, e ;
  for (i=1; i<L.recnum; ++i)   L.r[i-1].next = i;
  L.r[L.recnum].next = 0;     // 将L改造为静态链表
  for (i=0; i<L.keynum; ++i)
 {
    // 按最低位优先依次对各关键字进行分配和收集
    Distribute(L, i, f, e);    // 第i趟分配
    Collect(L, i, f, e);       // 第i趟收集
    print_SLList2(L, i);
  }
}
```

10.6 基数排序

二. 链式基数排序

- 程序实现

```
void Distribute(SLList &L, int i, ArrType &f, ArrType &e) {
// 静态链表L的r域中记录已按(keys[0],...,keys[i-1])有序，
 // 本算法按第i个关键字keys[i]建立RADIX个子表，
 // 使同一子表中记录的keys[i]相同。f[0..RADIX-1]和e[0..RADIX-1]
 // 分别指向各子表中第一个和最后一个记录。
 int j, p;
 for (j=0; j<RADIX; ++j) f[j] = 0;    // 各子表初始化为空表
 for (p=L.r[0].next;  p;  p=L.r[p].next)
 {
   j = L.r[p].keys[i]-'0';  // 将记录中第i个关键字映射到[0..RADIX-1]，
   if (!f[j]) f[j] = p;
   else L.r[e[j]].next = p;
   e[j] = p;            // 将p所指的结点插入第j个子表中
 }
} // Distribute
```

10.6 基数排序

二. 链式基数排序

- 程序实现

```
void Collect(SLList &L, int i, ArrType f, ArrType e) {  // 算法10.16
  // 本算法按keys[i]自小至大地将f[0..RADIX-1]所指各子表依次链接成
  // 一个链表，e[0..RADIX-1]为各子表的尾指针
  for ( j=0; !f[j]; j++ ) ; // 找第一个非空子表，succ为求后继函数: ++
  L.r[0].next = f[j];  // L.r[0].next指向第一个非空子表中第一个结点
  t = e[j];
  while ( j<RADIX )
 { for (j=j+1; j<RADIX && !f[j]; j++);      // 找下一个非空子表
   if ( j<RADIX )    // 链接两个非空子表
     {   L.r[t].next = f[j];    t = e[j];   }
  }
  L.r[t].next = 0;   // t指向最后一个非空子表中的最后一个结点
} // Collect
```

10.6 基数排序

二. 链式基数排序

- 性能分析

若每个关键字有 d 位，每位的基数为radix，则

- 需要重复执行d 趟“分配”与“收集”，每趟对 n 个对象进行“分配”，时间复杂度为O(n)；对radix个队列进行“收集”，时间复杂度为O(radix)，所以总时间复杂度为O(d(n + radix))。
- 若基数radix相同，对于记录个数较多而关键字位数较少的情况，使用链式基数排序较好。
- 链式基数排序需要增加n个指针域和2radix个队列指针，所以

  空间复杂度为 O(n+ radix)。
- 基数排序是稳定的排序方法。

练习

一. 已知数列为 125、45、388、272、165、39、272*、428、64，要对数列进行从小到大的排序，若采用链式基数排序，要求写出每趟排序结果。

内部排序

各种内部排序按所采用的基本策略分别是：

1. 插入排序：依次将无序序列中的一个记录，按关键字值的大小插入到已排好序一个子序列的适当位置，直到所有的记录都插入为止。具体的方法有：直接插入、折半插入和希尔排序。

2. 交换排序：对于待排序记录序列中的记录，两两比较记录的关键字，并对反序的两个记录进行交换，直到整个序列中没有反序的记录偶对为止。具体的方法有：冒泡排序、快速排序。

内部排序

3. 选择排序：不断地从待排序的记录序列中选取关键字最小（大）的记录，直到所有记录都被选取为止。具体的方法有：简单选择排序、堆排序。

4. 归并排序：利用“归并”技术不断地对待排序记录序列中的有序子序列进行合并，直到合并为一个有序序列为止。

5. 基数排序：按待排序记录的关键字的组成成分(“位”)从低到高(或从高到低)进行。每次是按记录关键字某一“位”的值将所有记录分配到相应的队列中，再按队列的编号依次将记录进行收集，最后得到一个有序序列。

内部排序

各种排序方法比较

排序方法	平均时间	最坏情况	辅助存储	适合情况
直接插入排序	$0(n^2)$	$0(n^2)$	$0(1)$	记录数不很多
希尔排序	$0(n(\log_2 n)^2)$	$0(n^2)$	$0(1)$	不太多
快速排序	$0(n\log_2 n)$	$0(n^2)$	$0(\log_2 n)$	较多
堆排序	$0(n\log_2 n)$	$0(n\log_2 n)$	$0(1)$	较多
归并排序	$0(n\log_2 n)$	$0(n\log_2 n)$	$0(n)$	都可以
基数排序	$0(d(n+rd))$	$0(d(n+rd))$	$0(rd)$	关键字位数少

注：蓝色的是稳定的排序算法

内部排序

- 本节讨论的排序方法是在顺序存储结构上实现的，在排序过程中需要移动大量记录。当记录数很多、时间耗费很大时，可以采用静态链表作为存储结构。但有些排序方法，若采用静态链表作存储结构，则无法实现表排序。

- 选取排序方法的主要考虑因素：
  - ✓ 待排序的记录数目n；
  - ✓ 每个记录的大小；
  - ✓ 关键字的结构及其初始状态；
  - ✓ 时间复杂度、空间复杂度、是否要求排序的稳定性；
  - ✓ 存储结构的初始条件和要求；
  - ✓ 语言工具和开发工作的复杂程度的平衡点等特性

本章总结

- 排序的概念：内部排序、外部排序、稳定排序、KCN和RMN
- 直接插入排序、折半插入排序
  - 时间复杂度为$O(n^2)$，是一种稳定排序方法
- 希尔排序：划分子序列进行插入排序，通过增量gap进行子序列划分
  - 时间复杂度约为$O(n \times (\log_2 n)^2)$，是一种不稳定的排序方法
- 起泡排序：依次比较相邻两个记录，每趟排序找出最大对象放在末尾
  - 时间复杂度为$O(n^2)$，是一种稳定排序方法
- 快速排序：选定枢轴记录来回扫描，将小于枢轴的数据放在枢轴左边，大于枢轴的数据放在右边，然后对左右子序列递归排序
  - 时间复杂度为$O(n\log_2 n)$，是一种不稳定排序方法
- 简单选择排序：每一趟选出i…n的最小记录，交换到前面
  - 时间复杂度为$O(n^2)$，是一种不稳定排序方法
- 堆排序：将初始序列建成最大堆，筛选根结点后，重复调整最大堆
  - 时间复杂度为$O(n\log_2 n)$，是一种不稳定排序方法
- 归并排序：2路归并一路排序
- 基数排序：最低位优先法LSD、链式基数排序

本章总结

- 6 分钟演示 15 种排序算法