第9章 排序

本章重点与难点

- **重点：**

  插入排序、交换排序、选择排序、归并排序、基数排序的思想和算法。

- **难点：**

  堆排序的思想和算法，在实际应用中如何根据实际情况，选择最优的排序算法。

第九章 排序

第九章 排序

9.1 概述

- 排序是计算机程序设计中的一种重要操作，它的功能是将一个数据元素(或记录)的任意序列，重新排列成一个按关键字有序的序列。
- 排序的目的之一是方便数据查找。

9.1 概述

内部排序和外部排序

在排序过程中，只使用计算机的内存存放待排序记录，称这种排序为内部排序。

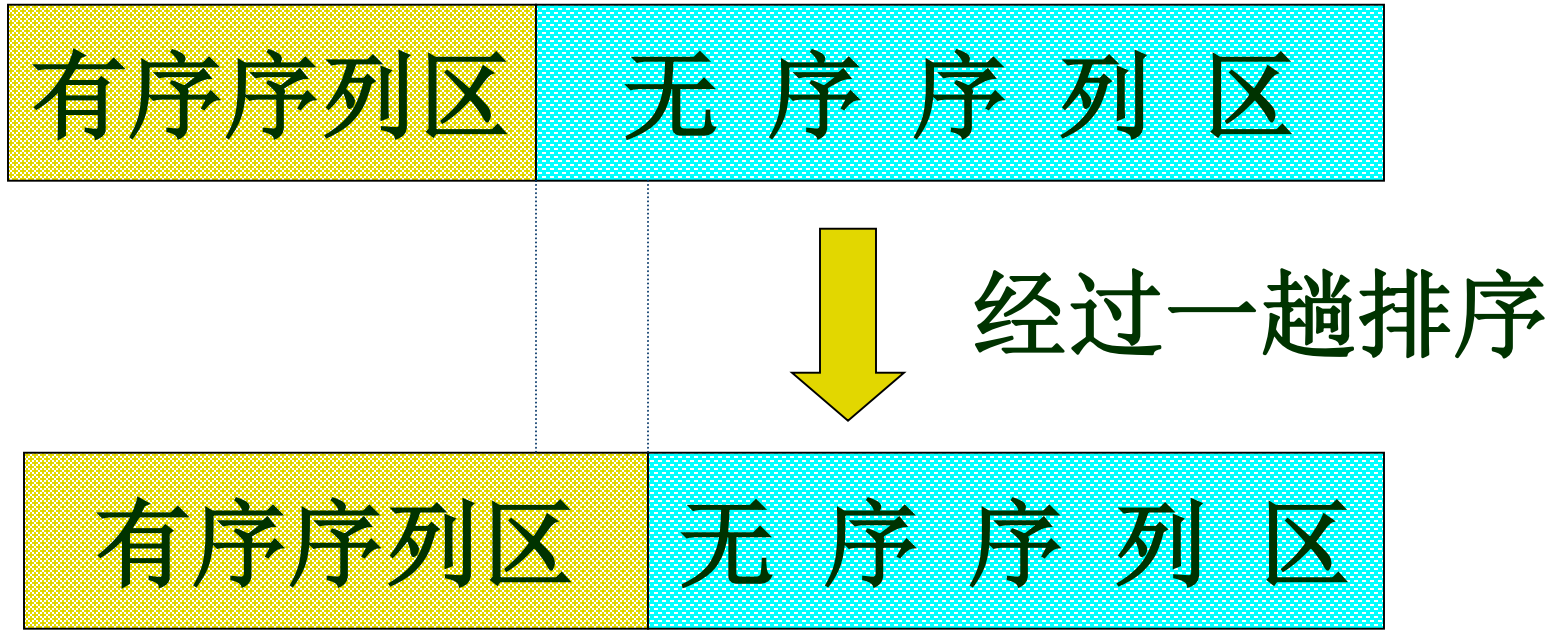
排序期间文件的全部记录不能同时存放在计算机的内存中，要借助计算机的外存才能完成排序，称之为“外部排序”。

内外存之间的数据交换次数就成为影响外部排序速度的主要因素。

9.1 概述

- 内部排序

内部排序的过程是一个逐步扩大记录的有序序列长度的过程。



9.1 概述

内部排序

基于不同的“扩大”有序序列长度的方法，内部排序方法大致可分下列几种类型：

插入类　　交换类　　选择类

归并类　　其它方法

9.1 概述

内部排序

排序算法的性能：

- **基本操作**。内排序在排序过程中的基本操作：
  - 比较：关键码之间的比较；
  - 移动：记录从一个位置移动到另一个位置。
- **辅助存储空间**。
  - 辅助存储空间是指在数据规模一定的条件下，除了存放待排序记录占用的存储空间之外，执行算法所需要 的其他额外存储空间。
- **算法本身的复杂度**。

9.1 概述

◘ 排序方法的稳定和不稳定

在排序过程中，有若干记录的关键字相等，即$K_i=K_j$($1\leqslant i\leqslant n$，1 $1\leqslant j\leqslant n$，$i\neq j$），在排序前后，含相等关键字的记录的相对位置保持不变，即排序前R_i在R_j之前，排序后R_i仍在R_j之前，称这种排序方法是稳定的；

反之，若可能使排序后的序列中R_i在R_j之后，称所用排序方法是不稳定的。

9.1 概述

存储结构-顺序表

```
#define MAXSIZE  1000 // 待排顺序表最大长度
typedef  int  KeyType;  // 关键字类型为整数类型
    typedef  struct {
       KeyType   key;                    // 关键字项
        InfoType  otherinfo;             // 其它数据项
    } RcdType;                           // 记录类型
typedef  struct {
   RcdType    r[MAXSIZE+1];             // r[0]闲置
   int            length;               // 顺序表长度
} SqList;                                // 顺序表类型
```

第九章 排序

9.2 插入排序

内部排序

1. 插入类

将无序子序列中的一个或几个记录“插入”到有序序列中，从而增加记录的有序子序列的长度。

9.2 插入排序

9.2.1 直接插入排序

9.2.2 折半插入排序

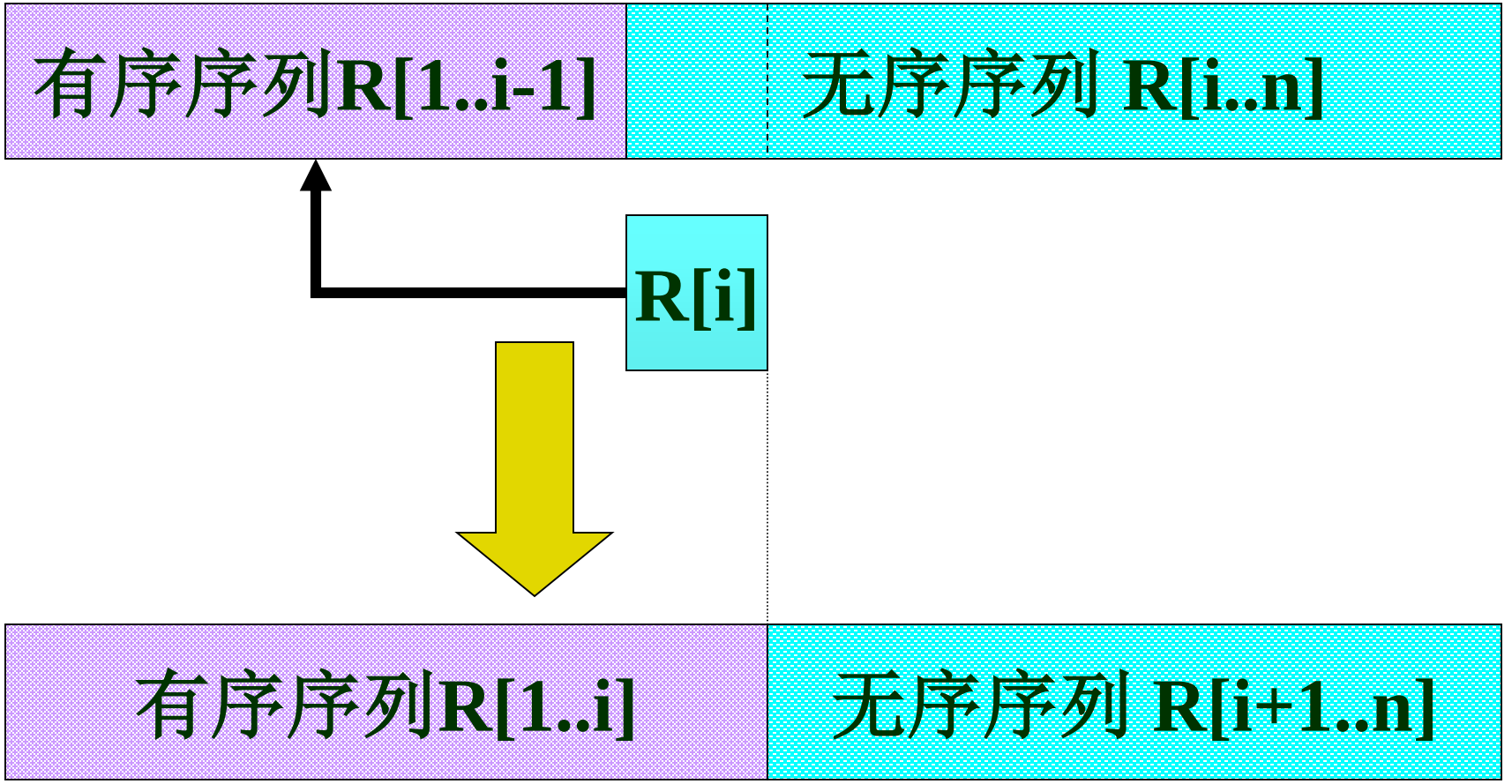
9.2.3 *表插入排序

9.2.4 希尔排序

9.2.1 直接插入排序

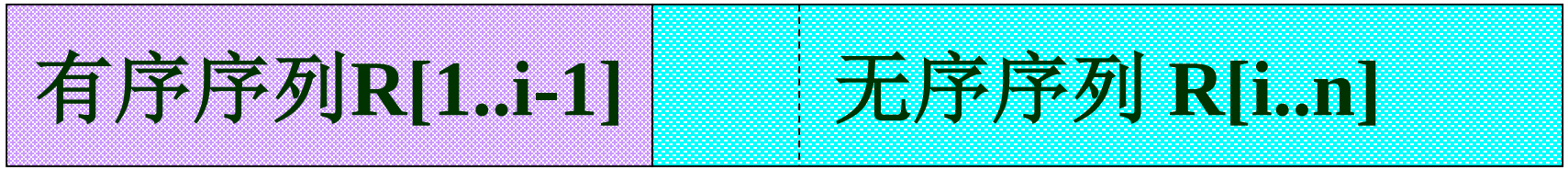
◘ 插入排序的思想

一趟直接插入排序的基本思想：



9.2.1 直接插入排序

□ 插入排序的思想



实现“一趟插入排序”可分三步进行：

1．在R[1..i-1]中查找R[i]的插入位置，

$R[1..j].key \le R[i].key < R[j+1..i-1].key$；

2．将R[j+1..i-1]中的所有记录均后移一个位置；

3．将R[i] 插入(复制)到R[j+1]的位置上。

9.2.1 直接插入排序

□ 直接插入排序算法概述

(1)将序列中的第1个记录看成是一个有序的子序列;

(2)从第2个记录起逐个进行插入，直至整个序列变成按关键字有序序列为止;

每一趟排序需要进行比较、后移记录、插入适当位置。从第二个记录到第n个记录共需n-1趟。

9.2.1 直接插入排序

利用“顺序查找”实现

“在R[1..i-1]中查找R[i]的插入位置”

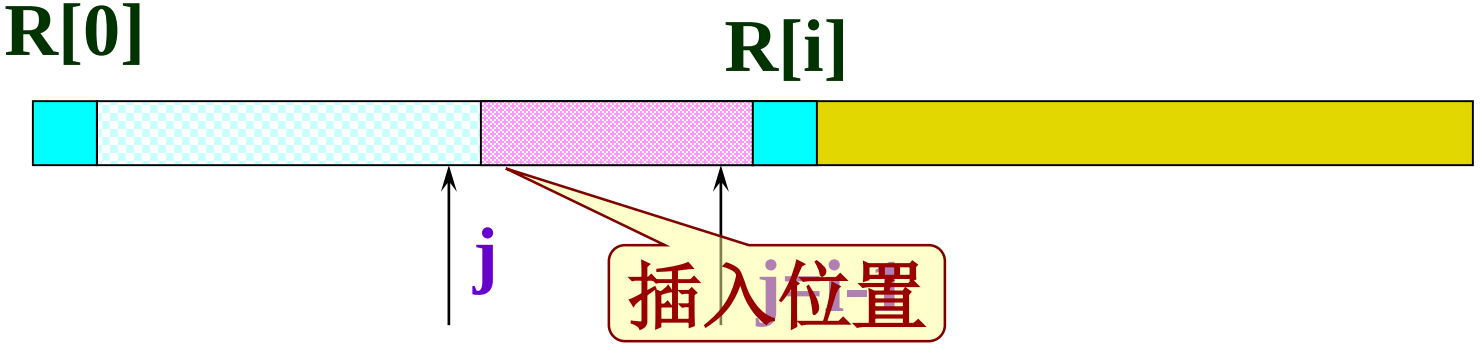
算法的实现要点：

9.2.1 直接插入排序

□ 直接插入排序算法要点

从R[i-1]起向前进行顺序查找，

监视哨设置在R[0]；




```
R[0] = R[i];          // 设置“哨兵”
for (j=i-1; R[0].key<R[j].key; --j);
                      // 从后往前找
```

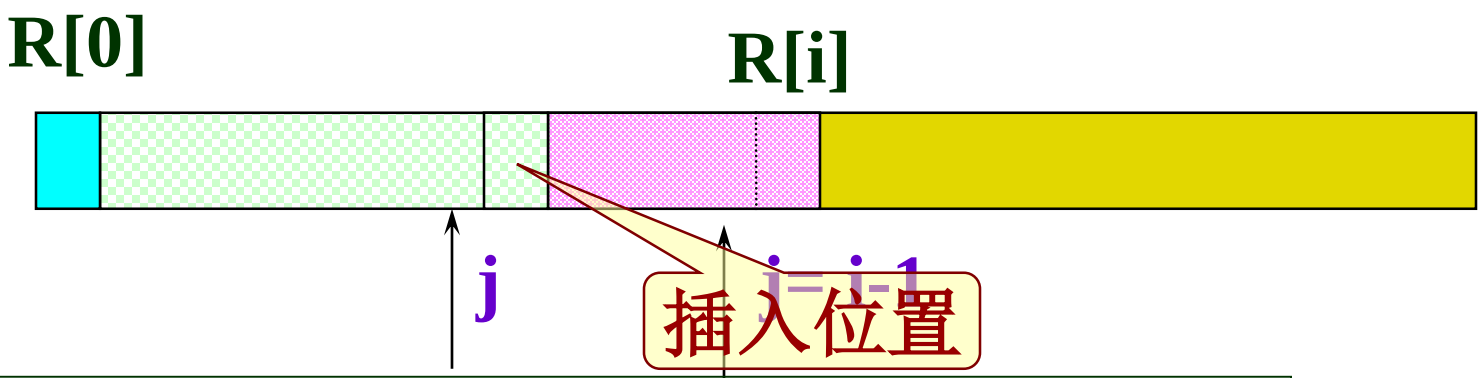
循环结束表明R[i]的插入位置为 *j +1*

9.2.1 直接插入排序

□ 直接插入排序算法要点

对于在查找过程中找到的那些关键字不小于R[i].key的记录，并在查找的同时实现记录向后移动；

```
for (j=i-1; R[0].key<R[j].key; --j);
    R[j+1] = R[j]
```



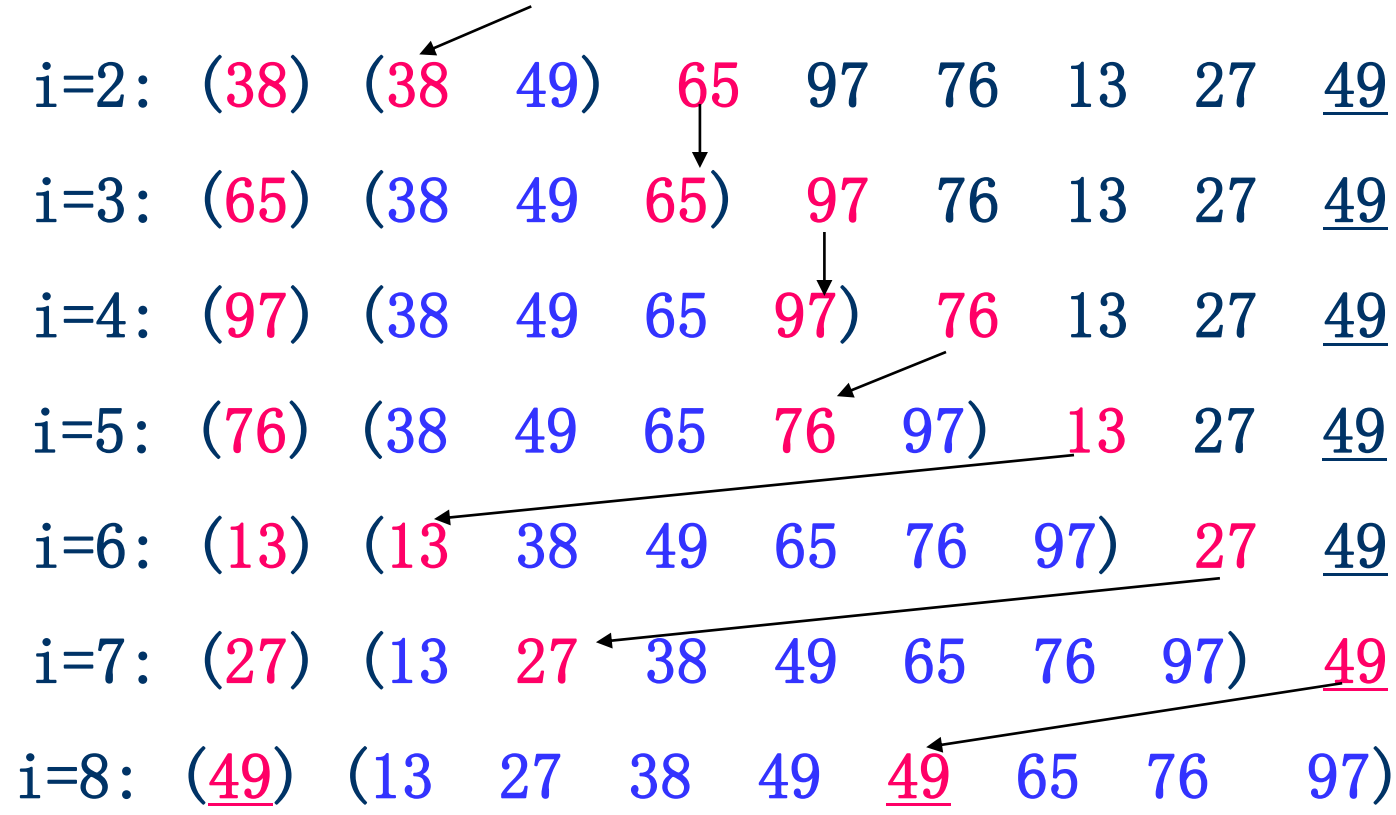
上述循环结束后可以直接进行“插入”

9.2.1 直接插入排序

□ 直接插入排序示例演示

例1 直接插入排序的过程。

初始关键字： (49) 38 65 97 76 13 27 49



9.2.1 直接插入排序

□直接插入排序算法

```
void InsertionSort ( SqList &L ) { //升序
  // 对顺序表 L 作直接插入排序。
   for ( i=2; i<=L.length; ++i )
      if (L.r[i].key < L.r[i-1].key) {
          L.r[0] = L.r[i];                  // 复制为监视哨
          for ( j=i-1; L.r[0].key < L.r[j].key;  -- j )
              L.r[j+1] = L.r[j];            // 记录后移
          L.r[j+1] = L.r[0];                // 插入到正确位置
      }
} // InsertSort
```

9.2.1 直接插入排序

□直接插入排序算法分析

实现直接插入排序的基本操作有两个：

（1）“比较”序列中两个关键字的大小；

（2）“移动”记录。

9.2.1 直接插入排序

□直接插入排序算法分析

最好的情况（关键字在记录序列中顺序有序）：

“比较”的次数：

$$\sum_{i=2}^{n} 1 = n - 1$$

“移动”的次数：

0

时间复杂度

比较O(n),移动O(1);

9.2.1 直接插入排序

□直接插入排序算法分析

最坏的情况（关键字在记录序列中逆序有序）：

“比较”的次数：

$$\sum_{i=2}^{n} i = \frac{(n+2)(n-1)}{2}$$

“移动”的次数：

$$\sum_{i=2}^{n} (i\text{-}1) = \frac{n(n-1)}{2}$$

时间复杂度

比较$O(n^2)$,移动$O(n^2)$;

9.2.1 直接插入排序

□直接插入排序算法分析

(1)稳定性

直接插入排序是稳定的排序方法。

R[0].key<R[j].key

(2)算法效率

a.时间复杂度

最好情况O(n):比较O(n),移动O(1);

最坏情况$O(n^2)$:比较$O(n^2)$,移动$O(n^2)$;

平均$O(n^2)$

b.空间复杂度

O(1)。

9.2.1 直接插入排序

□直接插入排序算法分析

算法的性能分析

- 直接插入排序算法简单、容易实现，适用于待排序记录基本有序或待排序记录较小时。
- 当待排序的记录个数较多时，大量的比较和移动操作使直接插入排序算法的效率降低。

改进的直接插入排序-----折半插入排序

直接插入排序，在插入第 i（$i>1$）个记录时，前面的 i-1 个记录已经排好序，则在寻找插入位置时，可以用折半查找来代替顺序查找，从而较少比较次数。

9.2.2 折半插入排序

◘ 折半插入排序思想

(1)在直接插入排序中，r[1..i-1] 是一个按关键字有序的有序序列;

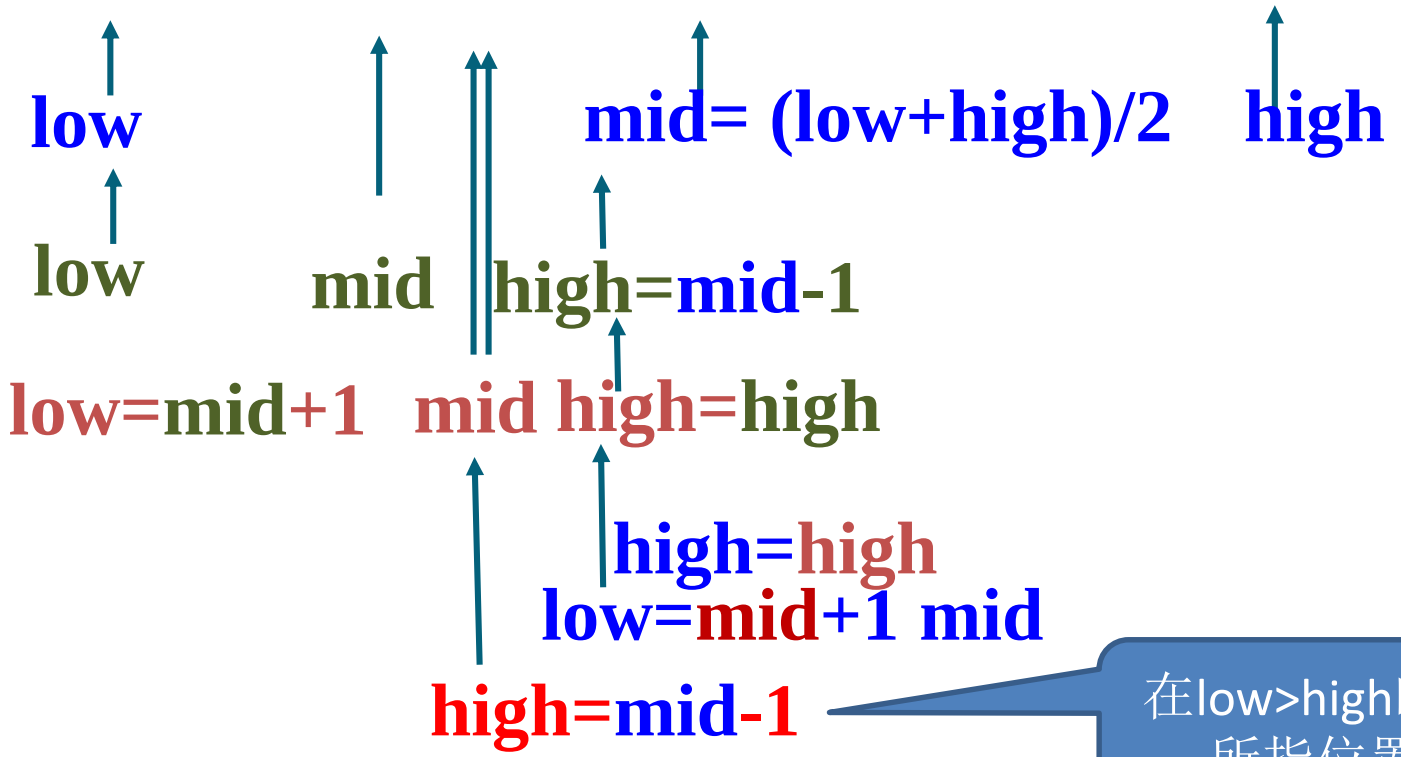
(2)可以利用折半查找实现“在r[1..i-1]中查找r[i]的插入位置”；

(3)称这种排序为折半插入排序。

9.2.2 折半插入排序

□ 折半插入排序的演示过程

查22　05　13　19　21　37　56　64　75　80　88　90



示例

在low>high时停止，在high+1所指位置进行插入操作

9.2.2 折半插入排序

□ 折半插入排序算法

```
void BiInsertionSort ( SqList &L )
{   for ( i=2; i<=L.length; ++i ) {
        L.r[0] = L.r[i];                    // 将 L.r[i] 暂存到 L.r[0]
        在 L.r[1..i-1]中折半查找插入位置；//见下一页
        for ( j=i-1;  j>=high+1;  --j )//插入位置为high+1
        L.r[j+1] = L.r[j];                  // 记录后移
        L.r[high+1] = L.r[0];               // 插入
    }                                       // for
}                                           // BInsertSort
```

9.2.2 折半插入排序

折半插入排序算法

```
low = 1;   high = i-1;              //查找区间
while (low<=high) {
    m = (low+high)/2;                       // 折半
    if (L.r[0].key < L.r[m].key)
        high = m-1;              // 插入点在低半区
      else  low = m+1;           // 插入点在高半区
}
```

9.2.2 折半插入排序

不同的具体实现方法导致不同的算法描述

- 直接插入排序—（基于顺序查找）
- 折半插入排序（基于折半查找）

} 移动相同数量的关键字

- 表插入排序（基于链表存储）→ 改进排序的存储结构
- 希尔排序（基于逐趟缩小增量）

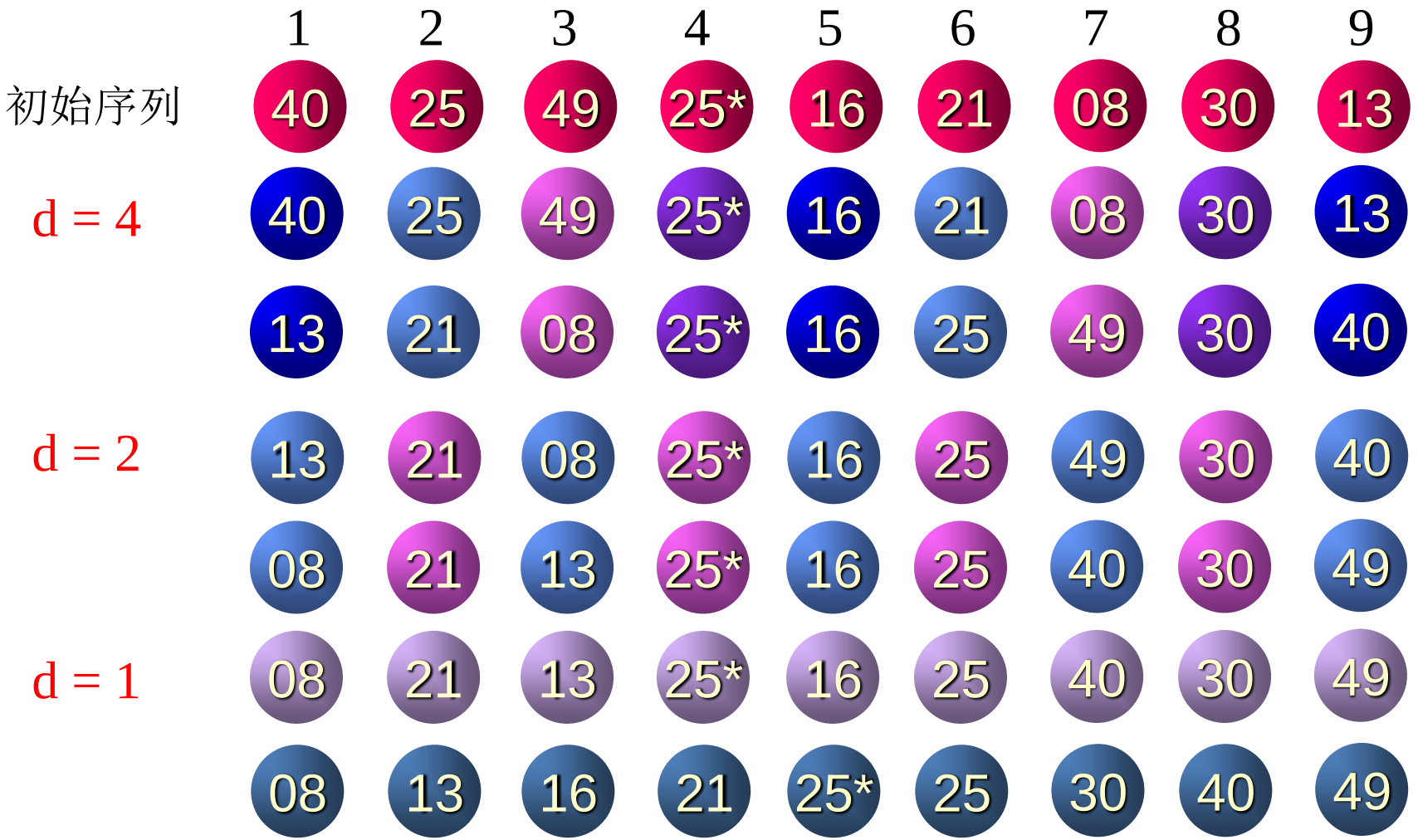
9.2.4 希尔排序

□ 希尔排序的思想

- 希尔排序是对直接插入排序的改进，改进的着眼点：
  - 若待排序记录按关键字值基本有序时，直接插入排序的效率可以大大提高；
  - 由于直接插入排序算法简单，则在待排序记录数量*n*较小时效率也很高。
- 希尔排序的基本思想：
  - 将整个待排序记录分割成若干个子序列，在子序列内分别进行直接插入排序，待整个序列中的记录基本有序时，对全体记录进行直接插入排序。
- 需解决的关键问题？
  - 分组：应如何分割待排序记录，才能保证整个序列逐步向基本有序发展？
  - 组内直接插入排序：子序列内如何进行直接插入排序？

9.2.4 希尔排序

- 示例：缩小增量排序



不稳定

9.2.4 希尔排序

希尔排序算法


```
void ShellSort(int n, LIST A)
{   int i, j, d;
    for (d=n/2; d>=1; d=d/2) {
        for (i=d+1; i<=n; i++) { // 将A[i]插入到所属的子序列中
            A[0].key= A[i].key;   //暂存待插入记录
            j=i-d;          //j指向所属子序列的最后一个记录
            while (j>0 && A[0].key< A[j].key) {
                A[j+d]= A[j];  //记录后移d个位置
                j=j-d;   //比较同一子序列的前一个记录
            }
            A[j+d]= A[0];
        }
    }
}
```

9.2.4 希尔排序

□ 希尔排序算法分析

■空间复杂度 O(1)

■时间复杂度

■希尔排序开始时增量（步长）较大，每个子序列中的记录个数较少，从而排序速度较快；当增量（步长）较小时，虽然每个子序列中记录个数较多，但整个序列已基本有序，排序速度也较快。

①步长的选择是希尔排序的重要部分。只要最终步长为1，任何步长序列都可以工作（当步长为1时，算法变为直接插入排序，这就保证了数据一定会被排序）。

②希尔排序算法的时间性能是所取增量（步长）的函数，而到目前为止尚未有人求得一种最好的增量序列。

③希尔排序的时间性能在$O(n^2)$和$O(n\log_2 n)$之间。当n在某个特定范围内，希尔排序所需的比较次数和记录的移动次数约为$O(n^{1.3})$ 。

第九章 排序

9.3 交换排序

- 内部排序

2. 交换类

通过“交换”无序序列中的记录从而得到其中关键字最小或最大的记录，并将它加入到有序子序列中，以此方法增加记录的有序子序列的长度。

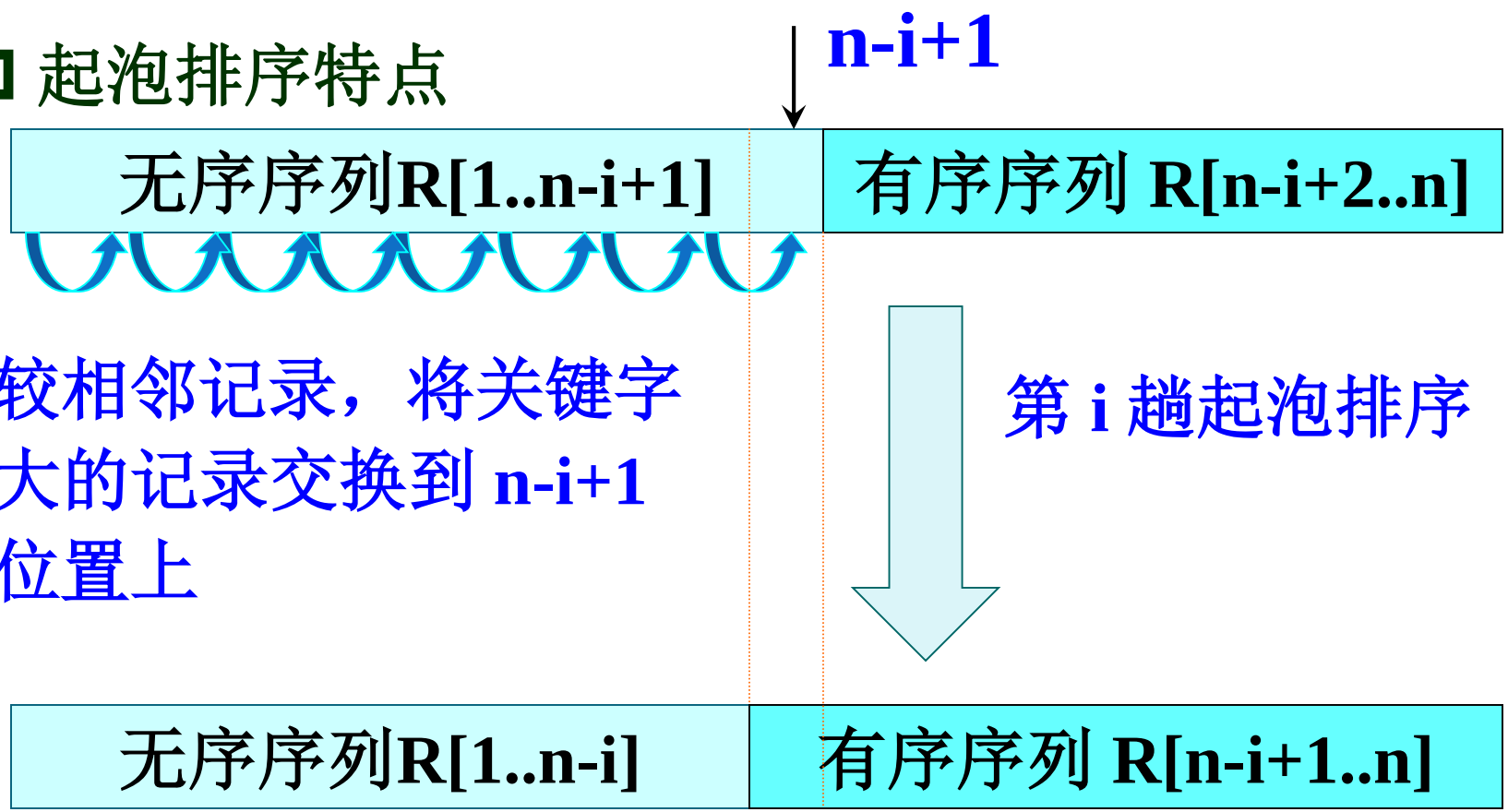
9.3 交换排序

9.3.1 起泡排序

9.3.2 快速排序

9.3.1 起泡排序

起泡排序特点



起泡排序的特点：

有序序列中的关键字值都比无序序列中的大

9.3.1 起泡排序

例：关键字序列 T=(21，25，49，25*，16，08），请写出冒泡排序的具体实现过程。

初态： 21，25，49， 25*，16， 08

第1趟 21，25，25*，16， 08 ， 49

第2趟 21，25， 16， 08 ，25*，49

第3趟 21，16， 08 ，25， 25*，49

第4趟 16，08 ，21， 25， 25*，49

第5趟 08，16， 21， 25， 25*，49

稳定

9.3.1 起泡排序

基本思想

(1) 从第一个记录开始，两两记录比较，若F[1].key>F[2].key，则将两个记录交换；

(2)第1趟比较结果将序列中关键字最大的记录放置到最后一个位置，称为“沉底”；

(3)n个记录最多比较n-1遍(趟)。

(4) 若一趟排序中没有交换，即可结束程序

9.3.1 起泡排序

算法

```
void BubbleSort(SqList &L )
{   int i,j,noswap; SqList  temp;
    for(i=1;i<=n-1;i++)
  {    noswap=TRUE;
       for(j=1;j<=n-i;j++)
          if (L.r[j].key>L.r[j+1].key)
             {temp=L.r[j]; L.r[j]=L.r[j+1]; L.r[j+1]=temp;
              noswap=FALSE; }
       if (noswap)  break;
  }
}
```

9.3.1 起泡排序

算法分析

最好的情况（关键字在记录序列中顺序有序）：

只需进行一趟起泡

“比较”的次数：

n-1

“移动”的次数：

0

最坏的情况（关键字在记录序列中逆序有序）：

需进行**n-1**趟起泡，每一次比较交换都移动**3**次

“比较”的次数：

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

“移动”的次数：

$$3\sum_{i=n}^{2}(i-1)=\frac{3n(n-1)}{2}$$

9.3.1 起泡排序

□ 算法分析

(1)稳定性

起泡排序是稳定的排序方法。

(2)时间复杂性

最好情况：比较O(n),移动O(1)

最坏情况：比较$O(n^2)$,移动$O(n^2)$

平均情况：$O(n^2)$

(3)空间复杂性

O(1)

9.3.2 快速排序

- （希尔）Shell 排序改进了插入排序
- 快速排序可以改进起泡排序

9.3.2 快速排序

□ 基本思想

任取待排序对象序列中的某个对象v(枢轴，基准，支点)，按照该对象的关键字大小，将整个序列划分为左右两个子序列；

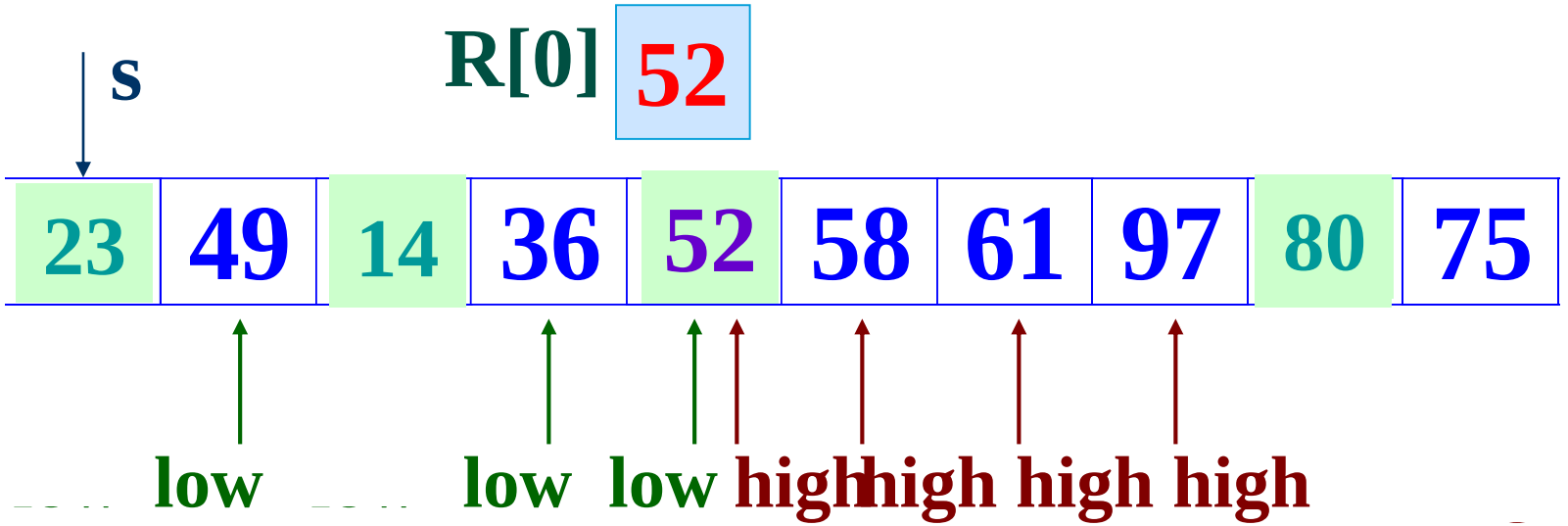
(1)左侧子序列中所有对象的关键字都小于或等于对象v的关键字；

(2)右侧子序列中所有对象的关键字都大于或等于对象v的关键字；

(3)对象v则排在这两个子序列中间(也是它最终的位置)。

9.3.2 快速排序

示例演示



设 R[s]=52 为枢轴

要求R[high].key ≥ 枢轴的关键字

要求R[low].key ≤ 枢轴的关键字

两个指针相对走，碰上的位置就是枢轴的位置

9.3.2 快速排序

快速排序的一趟排序过程

初始关键字　49　49*　65　97　17　27　50
low　high　high

一次交换　27　49*　65　97　17　49　50
low　low　high

二次交换　27　49*　49　97　17　65　50
low　high

三次交换　27　49*　17　97　49　65　50
low　high

四次交换　27　49*　17　49　97　65　50
high

完成一趟排序　low

不稳定

9.3.2 快速排序

算法关键语句

初始关键字 **49 49* 65 97 17 27 50**
low high high

```
pivotkey=L.r[low].key;
while ((low<high)&& (L.r[high].key>=pivotkey))
--high;
L.r[low] ←→ L.r[high];
```

一次交换 **27 49* 65 97 17 49 50**
low low high

```
while ((low<high)&& (L.r[low].key<=pivotkey))
++low;
L.r[low] ←→ L.r[high];
```

9.3.2 快速排序

算 法

```
int Partition(SqList &L, int low, int high)
{ KeyType pivotkey;pivotkey = L.r[low].key;
while (low<high) {
while ((low<high)&& (L.r[high].key>=pivotkey))
    --high;
    L.r[low] ←→ L.r[high];
while ((low<high)&& (L.r[low].key<=pivotkey))
     ++low;
     L.r[low] ←→ L.r[high];
   }
   return low;                              // 返回枢轴位置
} // Partition
```

示例

9.3.2 快速排序

□ 算 法

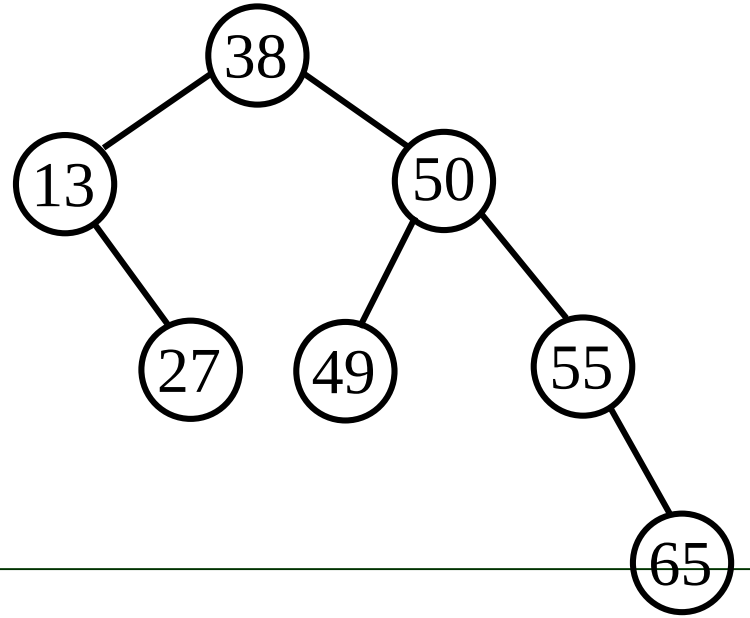
```
void QSort (ElemType R[],  int low,  int  high ) {
  // 对记录序列R[low..high]进行快速排序
  if (low <high-1) {          // 长度大于1
    pivotloc = Partition(L, low, high);
                    // 将R[low..high] 进行一次划分
   Qsort(R, low, pivotloc-1);
          // 对低子序列递归排序，pivotloc是枢轴位置
   Qsort(R, pivotloc+1,high); // 对高子序列递归排序
  }
} // QSort
```

9.3.2 快速排序

算法分析

快速排序的递归执行过程可以用递归树描述。

例如，序列 {38, 27, 55, 50, 13, 49, 65}的快速排序递归树如下:



时间复杂度为$O(n\log_2 n)$

9.3.2 快速排序

算法分析

若待排记录的初始状态为按关键字有序时，快速排序将蜕化为起泡排序，其时间复杂度为$O(n^2)$。

为避免出现这种情况，需在进行一次划分之前，进行“予处理”，即：

先对 R(low).key，R(high).key 和 R[⌊(low+high)/2⌋].key，进行相互比较，然后取关键字为“三者之中”的记录为枢轴记录。

第九章 排序

9.1 概述

- 内部排序

3. 选择类

从记录的无序子序列中“选择”关键字最小或最大的记录，并将它加入到有序子序列中，以此方法增加记录的有序子序列的长度。

9.4 选择排序

9.4.1 简单选择排序

9.4.2 树形选择排序

9.4.3 堆排序

9.4.1 简单选择排序

□ 基本思想

(1)第一次从n个关键字中选择一个最小值,确定第一个;

(2)第二次再从剩余元素中选择一个最小值,确定第二个;

(3)共需n-1次选择。

9.4.1 简单选择排序

操作过程

设需要排序的表是A[n+1]:

(1)第一趟排序是在无序区A[1]到A[n]中选出最小的记录，将它与A[1]交换，确定最小值;

(2)第二趟排序是在A[2]到A[n]中选关键字最小的记录，将它与A[2]交换，确定次小值;

(3)第i趟排序是在A[i]到A[n]中选关键字最小的记录，将它与A[i]交换;

(4)共n-1趟排序。

9.4.1 简单选择排序

□ 操作过程

简单选择排序与起泡排序的区别在：

起泡排序每次比较后 ，如果发现顺序不对立即进行交换，而选择排序不立即进行交换而是找出最小关键字记录后再进行交换。

和直接插入排序的区别：

直接插入排序是从无序区随意选择一个，插入有序区相应位置

简单选择排序是从无序区中选择最小（大），直接放入有序区最小或最大位置

9.4.1 简单选择排序

算 法

```
void  SelectSort(SqList &L)
{int i,j,low;
  for(i=1;i<L.length;i++)
   {low=i;
    for(j=i+1;j<=L.length;j++)
        if(L.r[j].key<L.r[low].key)
            low=j;
     if(i!=low)
       {L.r[0]=L.r[i];L.r[i]=L.r[low];L.r[low]=L.r[0];
       }
     }
}
```

时间复杂度为$O(n^2)$。

9.4.1 简单选择排序

算法分析

(1)稳定性

简单选择排序方法是不稳定的。例（2,2',1）

(2)时间复杂度

比较$O(n^2)$,

移动最好O(1),最差O(n)

(3)空间复杂度

为O(1)。

9.4.2 树形选择排序

引入

树形选择排序，又称锦标赛排序:按锦标赛的思想进行排序，目的是减少选择排序中的重复比较次数。

例如： 4,3,1,2 在选择排序中3和4的比较次数共发生了三次。

9.4.2 树形选择排序

引入

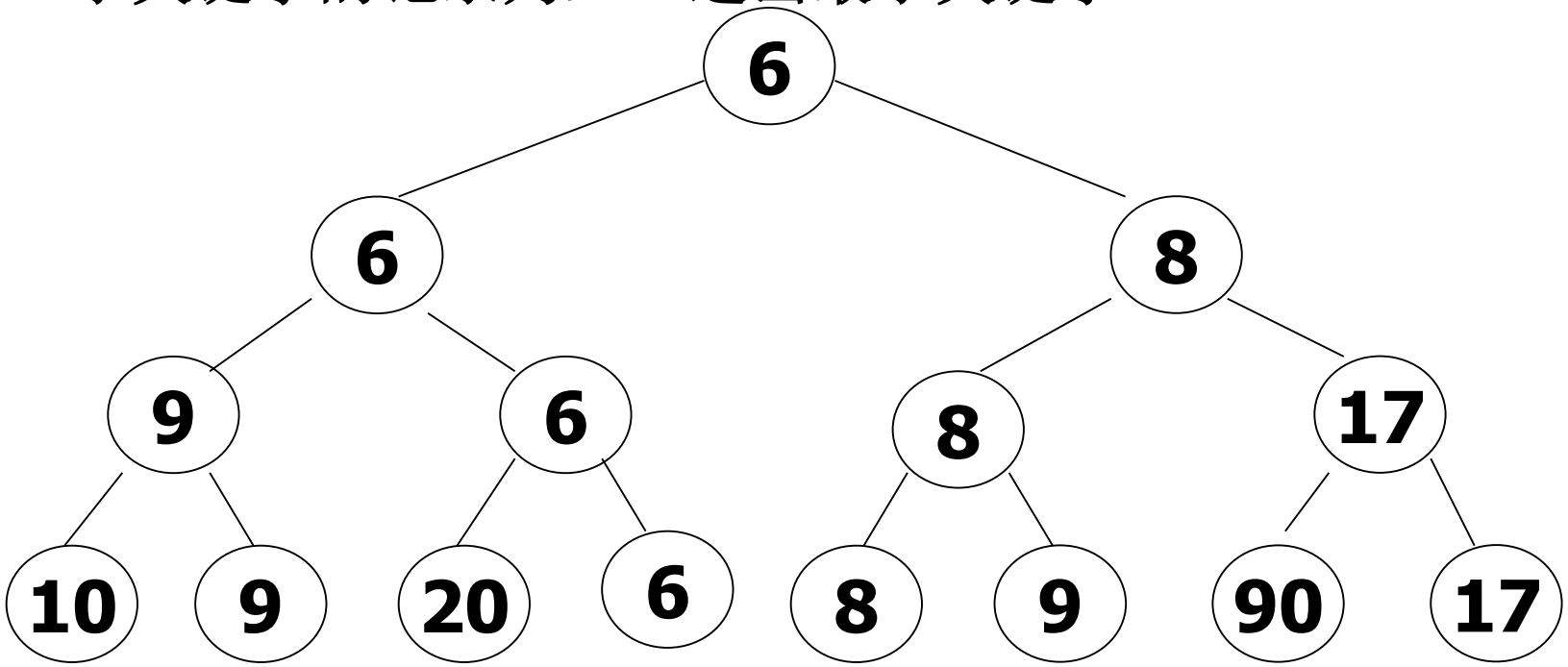
- ✓ 显然，在n个关键字中选出最小值，至少进行n-1次比较，然而，继续在剩余n-1个关键字中选择次小值就并非一定要进行n-2次比较，若能利用前n-1次比较所得信息，则可减少以后各趟选择排序中所用的比较次数。
- ✓ 实际上，体育比赛中的锦标赛便是一种选择排序。

例如，在8个运动员决出前3名至多需要11场比赛，而不是7（全比一次）+6（去掉第一之后再全比一次）+5（去掉两个后再比一次）=18场比赛（它的前提是若乙胜丙，甲胜乙，则认为甲必能胜丙）

9.4.2 树形选择排序

示例演示

按照一种锦标赛的思想进行选择排序的方法。首先对n个记录的关键字进行两两比较，然后在其中[n/2](向上取整)个较小者之间再进行两两比较，如此重复，直到选出最小关键字的记录为止。选出最小关键字.

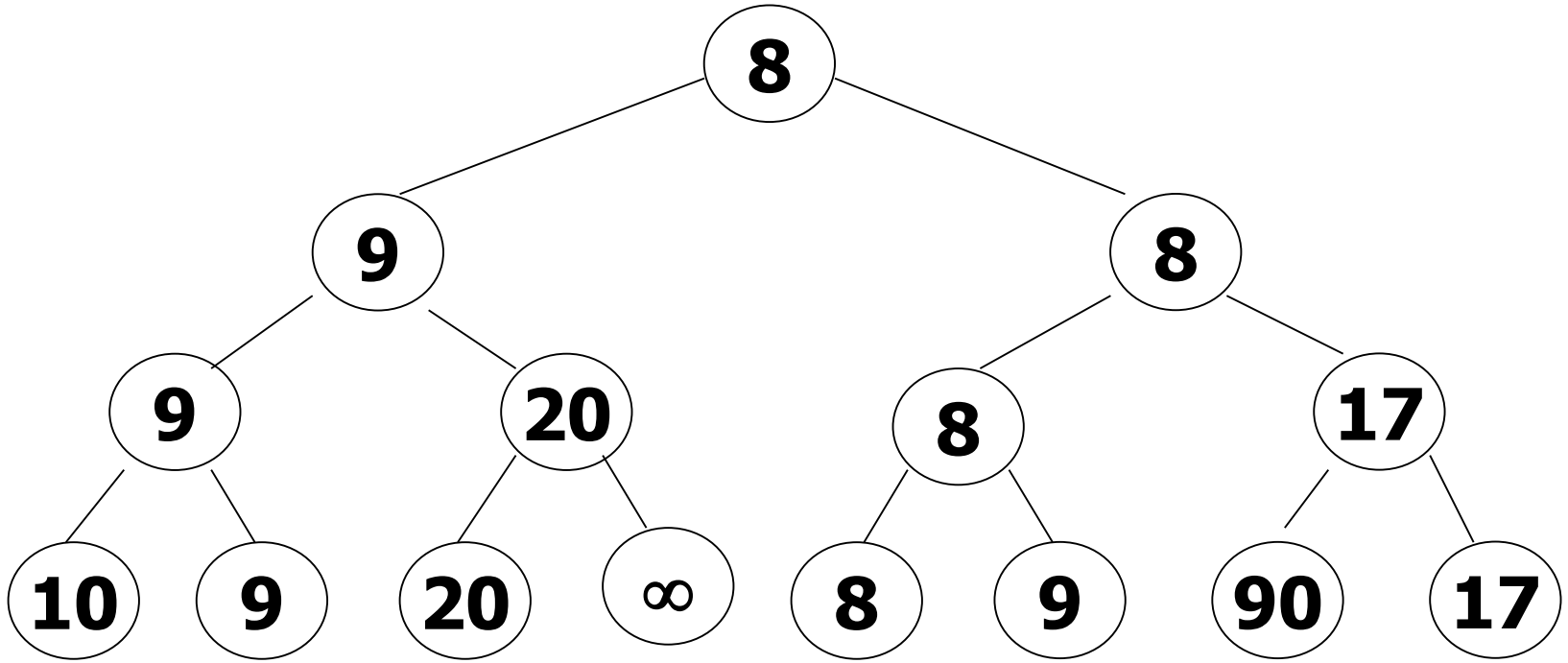


输出6

9.4.2 树形选择排序

示例演示

然后把最小关键字放最大值，然后选出次小关键字.



输出8

浪费存储空间比较大，和最大值进行多余的比较等缺点。

1964年J. Willioms提出了堆排序。

9.4.3 堆排序

引　入

堆排序属于选择排序:出发点是利用前一次比较的结果，减少“比较”的次数。

若能利用每趟比较后的结果，也就是在找出关键字值最小记录的同时，也找出关键字值较小的记录，则可减少后面的选择中所用的比较次 数，从而提高整个排序过程的效率。

减少关键字之间的比较次数

查找最小值的同时，找出较小值

9.4.3 堆排序

□ 堆的定义

把具有如下性质的数组A表示的完全二叉树称为小根堆：

(1) 若2*i≤ n，则A[i].key ≤ A[2*i].key ；

(2) 若2*i+1 ≤ n，则A[i].key ≤ A[2*i+1].key

把具有如下性质的数组A表示的完全二叉树称为大根堆：

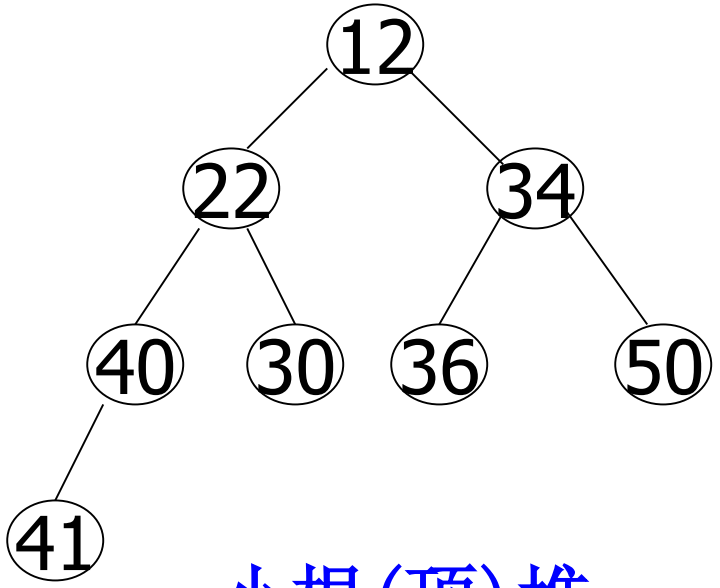
(1) 若2*i≤ n，则A[i].key ≥ A[2*i].key ；

(2) 若2*i+1 ≤ n，则A[i].key ≥ A[2*i+1].key

9.4.3 堆排序

小根堆例子

0	1	2	3	4	5	6	7	8
	12	22	34	40	30	36	50	41

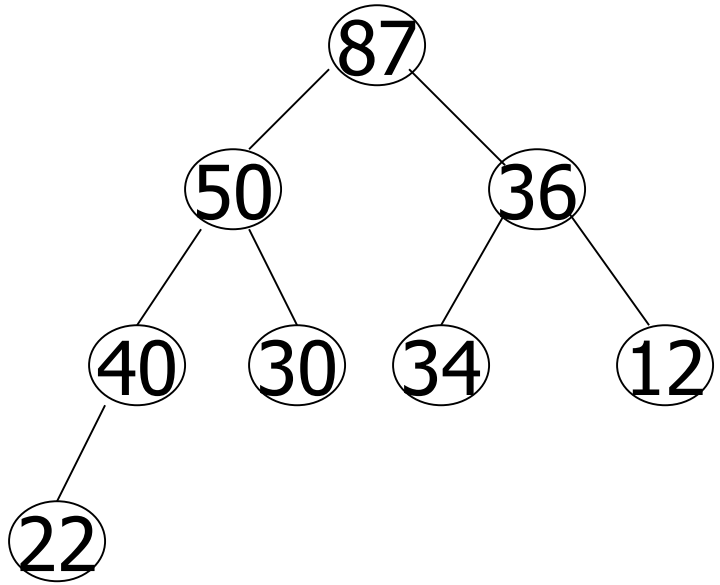


小根(顶)堆

9.4.3 堆排序

大根堆例子

0	1	2	3	4	5	6	7	8
	87	50	36	40	30	34	12	22



大根(顶)堆

9.4.3 堆排序

堆的性质（小根堆）

- 对于任意一个非叶结点的关键字，都不大于其左、右儿子结点的关键字。即**A[i/2].key≤A[i].key　1 ≤ i/2 < i ≤n**
- 在堆中，以任意结点为根的子树仍然是堆。特别地，每个叶结点也可视为堆。每个结点都代表(是)一个堆。
  - ✓ 以堆（的数量）不断扩大的方式进行初始建堆。
- 在堆中（包括各子树对应的堆），其根结点的关键字是最小的。删除该关键字后，调整堆。
  - ✓ 以堆的规模逐渐缩小的方式进行堆排序。

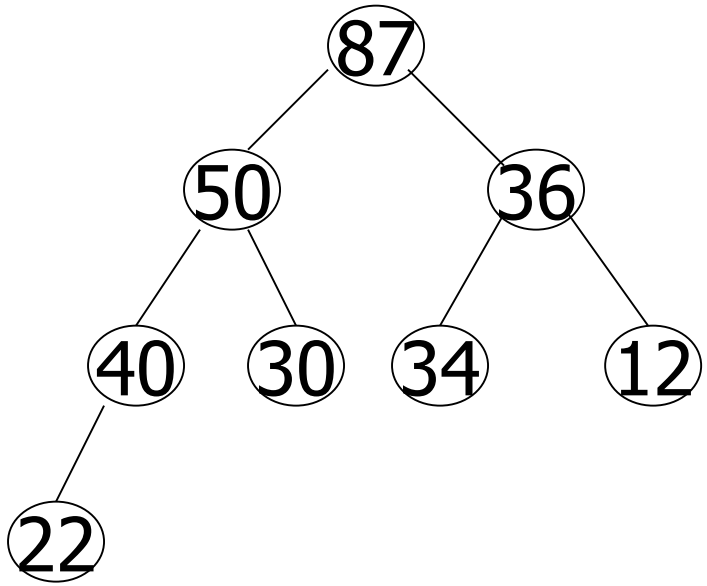
9.4.3 堆排序

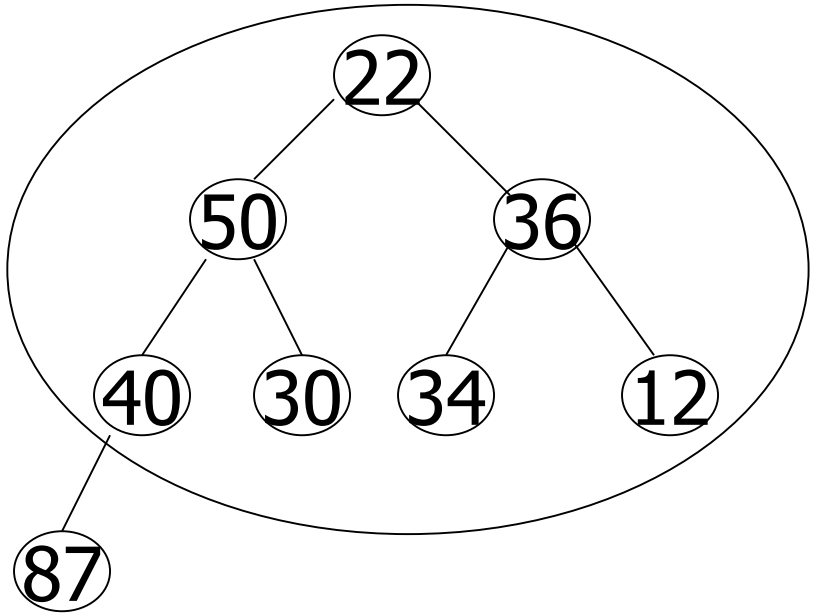
□堆排序

若输出堆顶的最小值后，使得剩余n-1个元素的序列重又建成一个堆，则得到n个元素的次小值。如此反复执行，便得到一个有序序列，这个过程称为堆排序。

	87	50	36	40	30	34	12	22

	22	50	36	40	30	34	12	87





9.4.3 堆排序

◘ 堆排序的算法(采用大根堆)

(1) 按关键字建立A[1],A[2],…A[n]的大根堆;

(2) 输出堆顶元素，采用堆顶元素A[1]与最后一个元素A[n]交换，最大元素得到正确的排序位置;

(3) 此时前n-1个元素不再满足堆的特性，需重建堆;

(4) 循环执行b,c两步，到排序完成。

9.4.3 堆排序

□ 堆排序的思想

堆排序需解决两个问题：

(1) 由一个无序序列建成一个堆。

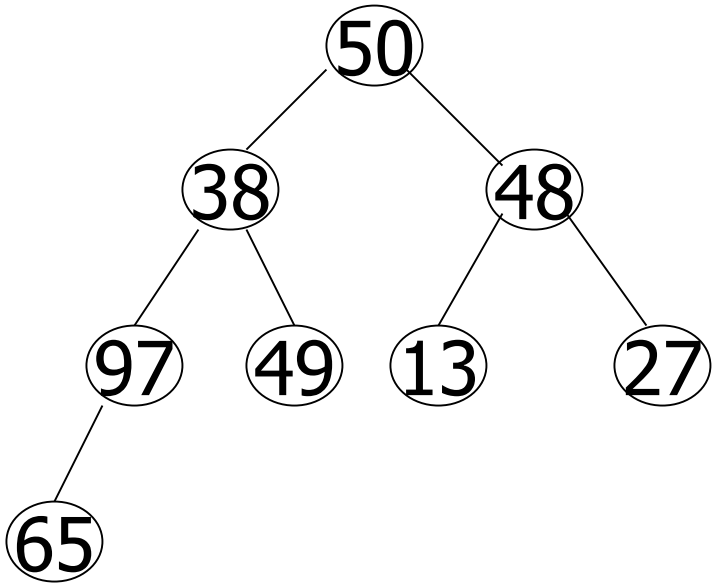
(2) 在输出堆顶元素之后，调整剩余元素成为一个新的堆。

9.4.3 堆排序

□ 算法示例演示

例 对无序序列{50，38，48，97，49，13，27，65}进行堆排序。

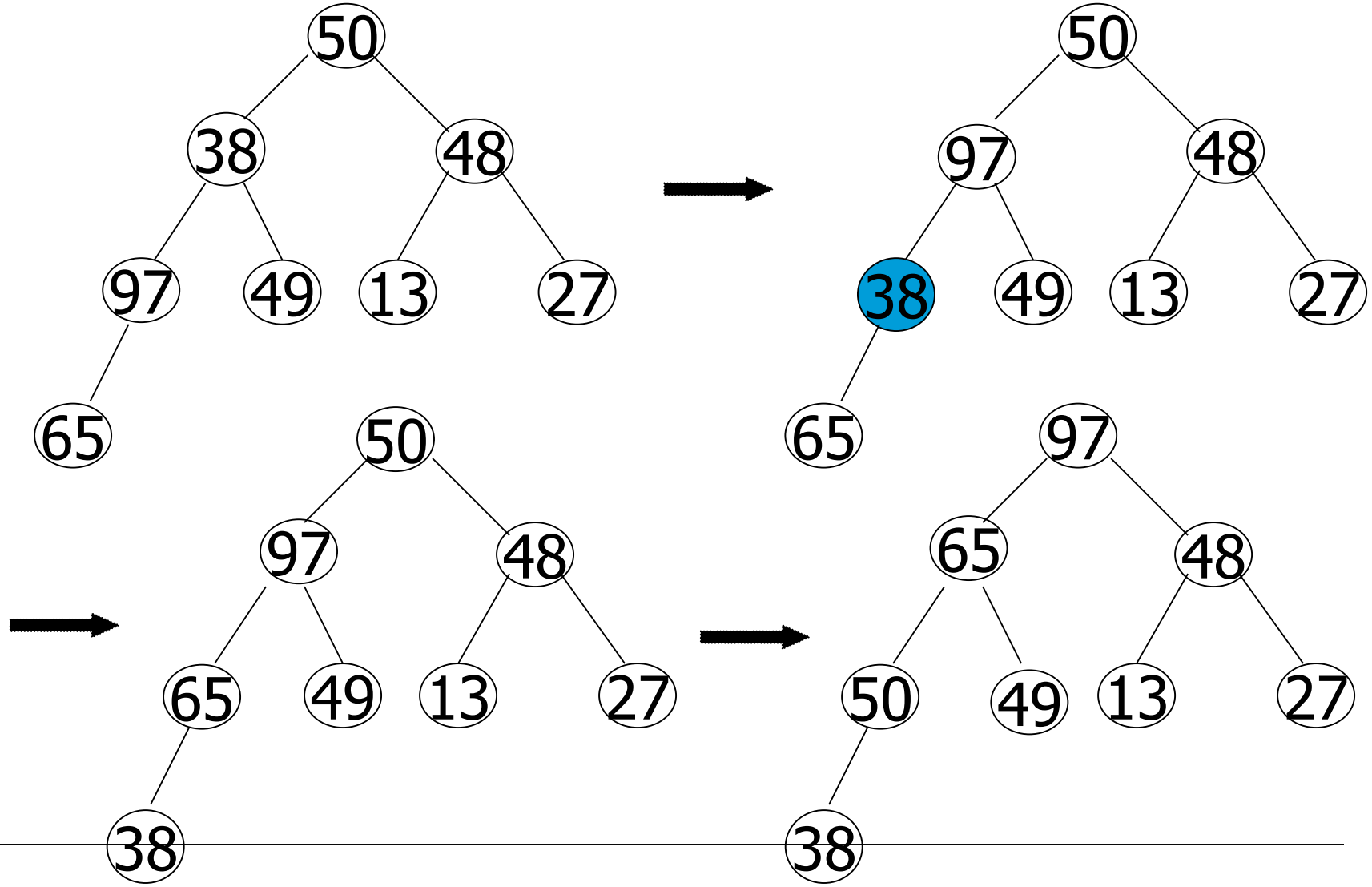
(1)先建一个完全二叉树：



2) 从最后一个非终端结点开始建堆（大根堆）；
n个结点, 最后一个非终端结点的下标是⌊n/2⌋

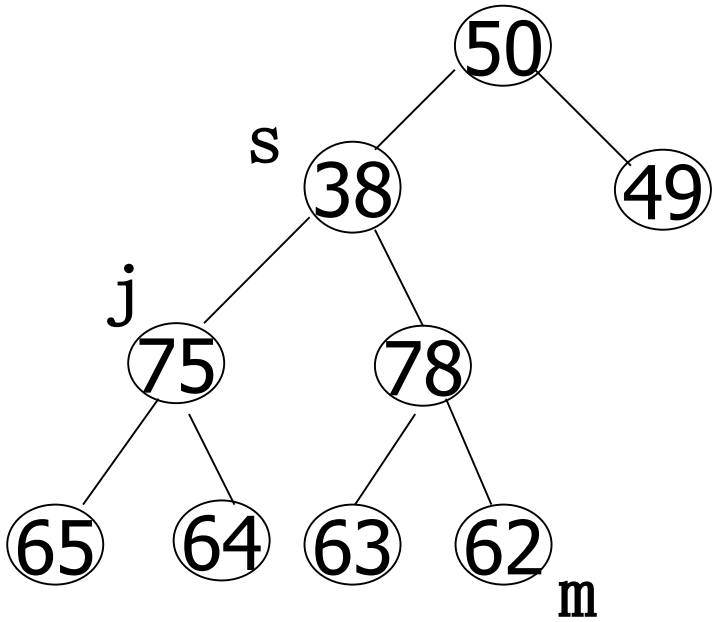
9.4.3 堆排序

算法示例演示



9.4.3 堆排序

算法示例演示

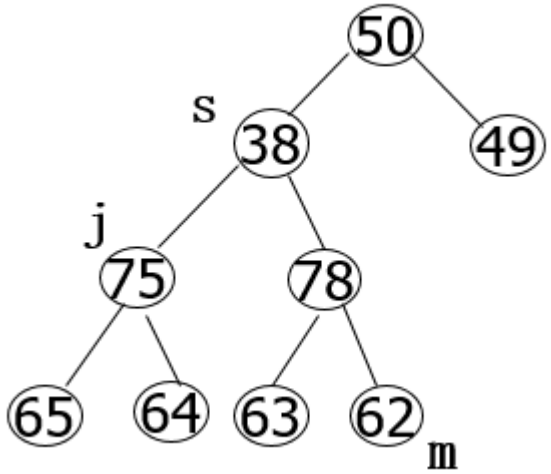


9.4.3 堆排序

筛选算法(下坠）

```
void HeapAdjust(HeapType &H,  int s, int m)
 {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
    {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;
    H.r[s] = H.r[j];  s = j; //rc应插入在位置s上
   }
  H.r[s] = rc;  // 插入
} // HeapAdjust  时间复杂度O(h)
```

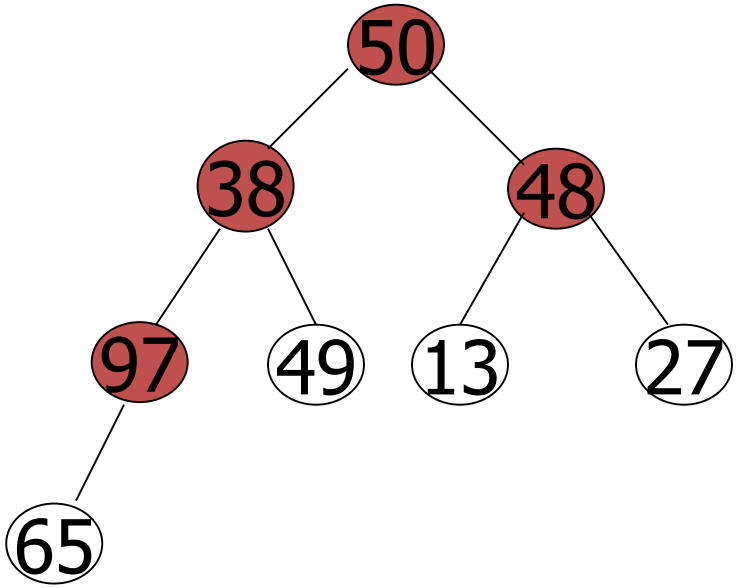
//已知H.r[s...m]中的记录关键字除H.r[s].key之外均满足堆的定义，本函数调整H.r[s]的关键字，使得H.r[s...m]成为大根堆；



9.4.3 堆排序

建堆算法代码

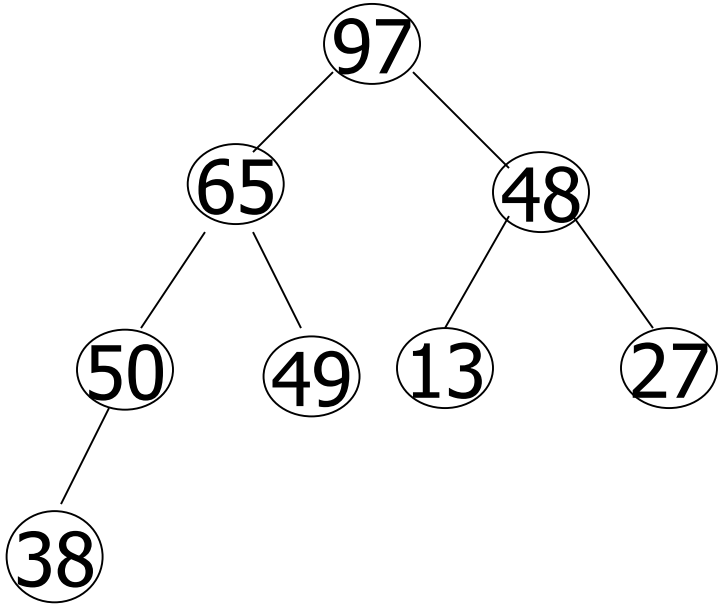
```
for (i=H.length/2; i>0; --i)//对每个非终端结点进行调整
HeapAdjust ( H, i, H.length );
```



9.4.3 堆排序

堆排序算法代码

```
void HeapSort(HeapType &H) {
  int i;
  RcdType temp;
  for (i=H.length/2; i>0; --i)
    HeapAdjust ( H, i, H.length ); //建堆
  for (i=H.length; i>1; --i) {
      temp=H.r[i];H.r[i]=H.r[1];
      H.r[1]=temp; //将堆顶记录和子序列最后一个记录交换
      HeapAdjust(H, 1, i-1); //重新调整为大根堆
    }
} // HeapSort  时间复杂度O(n*h)
```



9.4.3 堆排序

算法分析

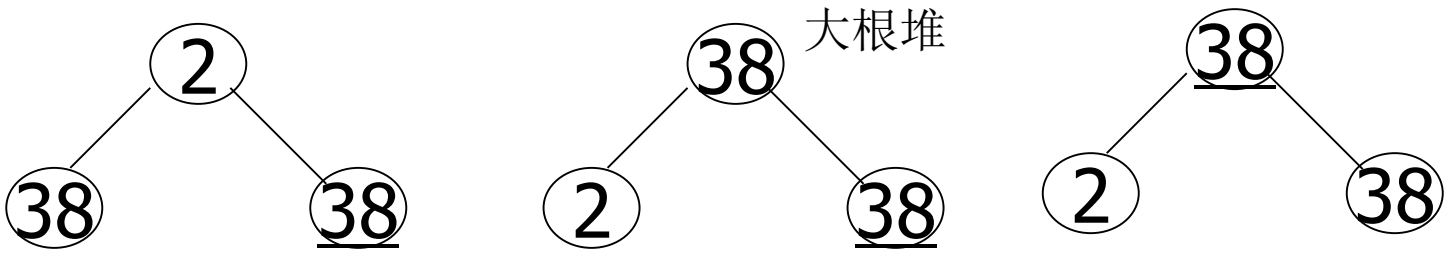
(1)堆排序是不稳定的排序。

(2)时间复杂度为$O(n\log_2 n)$。

最坏情况下时间复杂度为$O(n\log_2 n)$的算法。

(3)空间复杂度为$O(1)$。

{2，38，38}



第九章 排序

9.5 归并

- 内部排序

4.归并类

通过“归并”两个或两个以上的记录有序子序列，逐步增加记录有序序列的长度。

9.5 归并排序

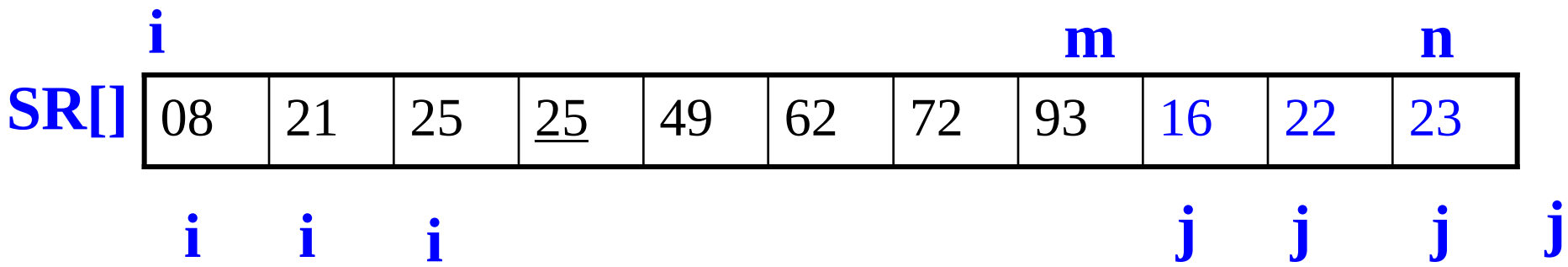
归并的定义

归并又叫合并，两个或两个以上的有序序列合并成一个有序序列。

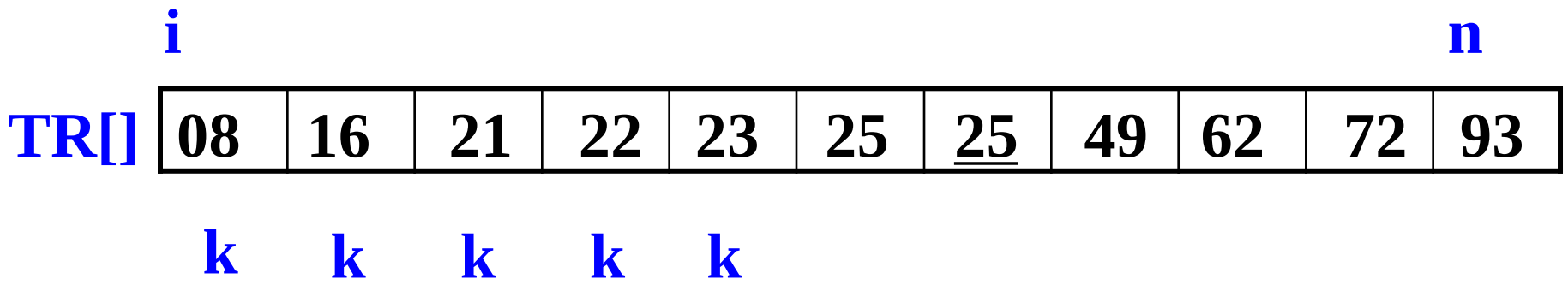
归并排序的主要操作是归并，其主要思想是：将若干有序序列逐步归并，最终得到一个有序序列。

9.5 归并排序

归并示例演示

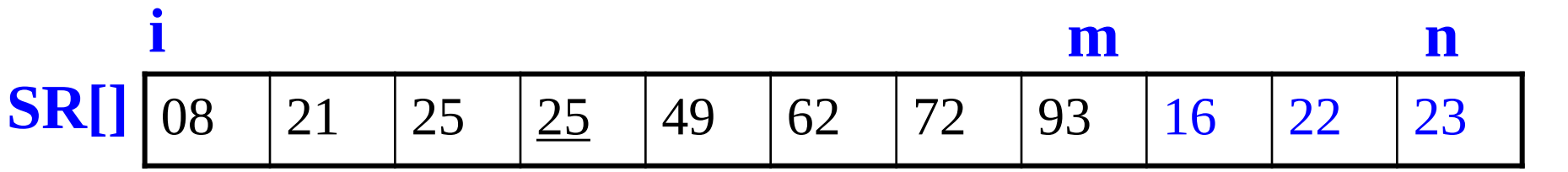



```
for (j=m+1, k=i;  i<=m && j<=n;  ++k) {
    if (SR[i].key <= SR[j].key)
    TR[k] = SR[i++]; else TR[k] = SR[j++]; }
```



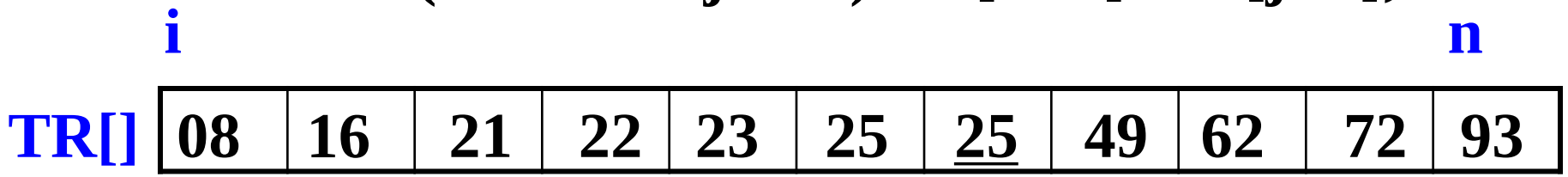
9.5 归并排序

归并示例演示



i i i j j j j

```
if (i<=m)
    while (k<=n && i<=m) TR[k++]=SR[i++];
if (j<=n)
    while (k<=n &&j <=n) TR[k++]=SR[j++];
```



k k k k k

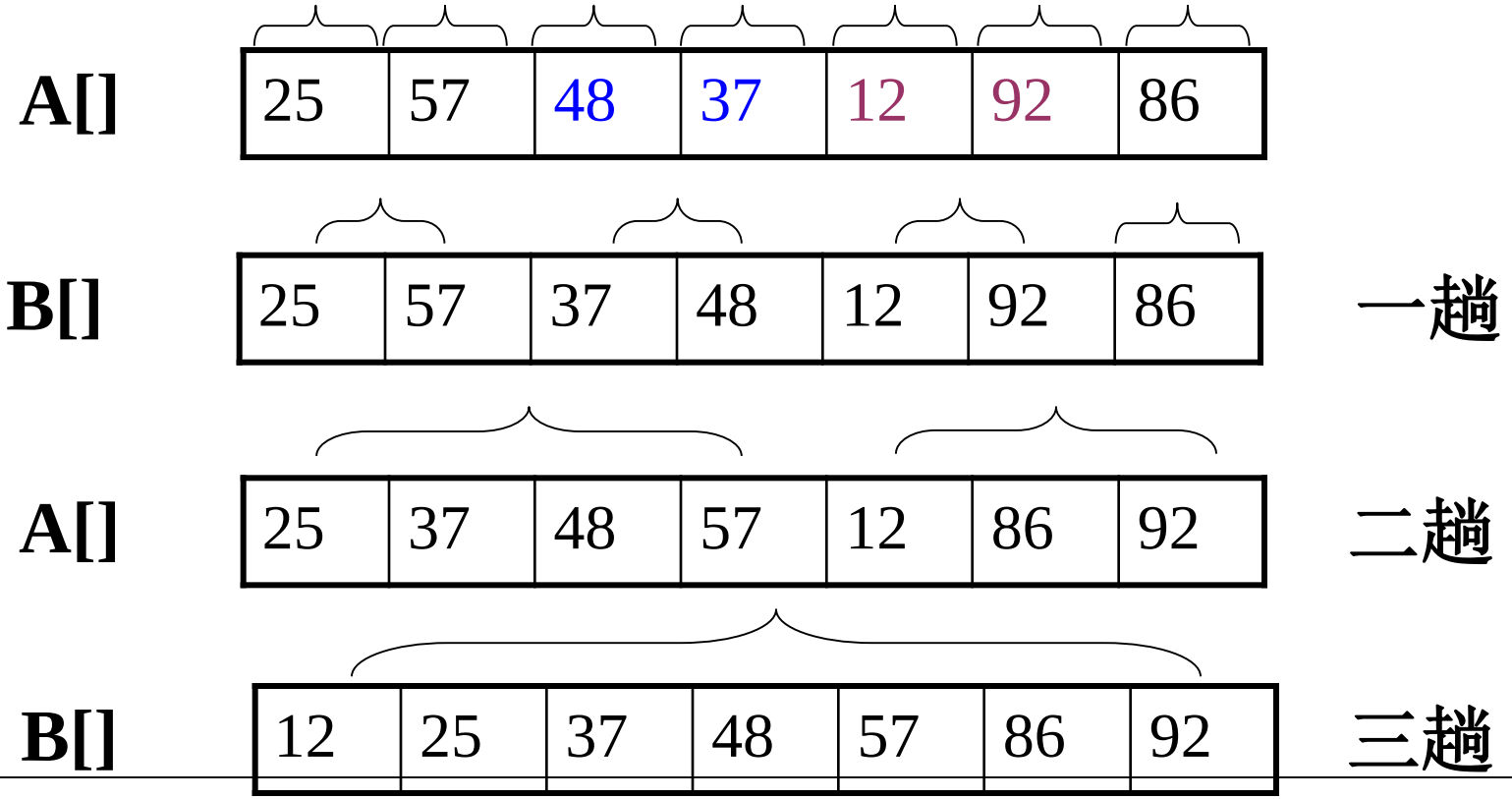
9.5 归并排序

```
void Merge (RcdType SR[], RcdType TR[], int i, int m, int n)
{ int j,k;
  for (j=m+1, k=i;  i<=m && j<=n;  ++k) {
    if (SR[i].key<=SR[j].key) TR[k] = SR[i++];
    else TR[k] = SR[j++];
  }
  if (i<=m)
    while (k<=n && i<=m) TR[k++]=SR[i++];
  if (j<=n)
    while (k<=n &&j <=n) TR[k++]=SR[j++];
 } // Merge
```

9.5 归并排序

2-路归并排序方法示例

例 给定排序码25，57，48，37，12，92，86，写出二路归并排序过程。



9.5 归并排序

2-路归并排序方法思想

(1) 将n个记录看成是n个长度为1的有序子表；

(2) 将两两相邻的有序子表进行归并，若子表数为奇数，则留下的一个子表直接进入下一次归并；

(3) 重复步骤(2)，直到归并成一个长度为n的有序表。

9.5 归并排序

□ 2-路归并排序-非递归

25	57	48	37

怎样完成一趟归并？

```
/*把A中长度为 h 的相邻序列归并成长度为 2h 的序列*/
void  MergePass (int n , int h , LIST A , LIST B)
{    int i , t ;
     for ( i=1 ; i+2*h-1<= n ; i+=2*h )
          Merge(i, i+h-1, i+2*h-1, A, B) ;//归并长度为h的两个有序子序列
     if ( i+h-1< n )   /* 尚有两个子序列，其中最后一个长度小于 h */
          Merge( i, i+h-1, n , A, B) ;    /* 归并最后两个子序列 */
     else /* 若i<= n且i+h-1>= n 时， 则剩余一个子序列轮空，直接复制 */
          for ( t= i ; t<= n ; t++ )
               B[t] = A[t] ;
} /* Mpass */
```

9.5 归并排序

2-路归并排序-非递归

25	57	48	37

（二路）归并排序算法：如何控制二路归并的结束？

```
void MergeSort ( int n , LIST A )
{   /* 二路归并排序 */
    int h = 1 ;/* 当前归并子序列的长度，初始为1 */
    LIST B ;
    while (h< n){
        MergePass (n , h , A , B) ;
        h = 2*h ;
        MergePass (n , h , B , A) ; /* A、B互换位置 */
        h = 2*h ;
    }
}/* MergeSort */
```

9.5 归并排序

□ 2-路归并排序-递归（分治算法）

25	57	48	37

算法的基本思想

分解：将当前待排序的序列A[low], …, A[high]一分为二，即求分裂点 mid = (low + high) / 2 ；

求解：递归地对序列 A[low], …, A[mid]和A[mid+1], …, A[high]进行归并排序；

组合：将两个已排序子序列归并为一个有序序列。

递归的终止条件

是子序列长度为 1，

因为一个记录自然有序。

算法实现

9.5 归并排序

2-路归并排序-递归

25	57	48	37

```
void MergeSort ( LIST A , LIST B , int low , int high )
/* 用分治法对A[low], …, A[high]进行二路归并 */
{    int mid = (low+high)/2 ;
     if (low<high){ /* 区间长度大于 1 ， high-low>0 */
          MergeSort (A , B , low , mid) ;
          MergeSort (A , B , mid+1 , high) ;
          Merge (low , mid , hight , A , B) ;
     }
}/* MergeSort */
```

9.5 归并排序

□ 2-路归并排序算法分析

(1) 稳定性

归并排序是稳定的排序方法。

(2) 时间复杂度

每趟归并所花时间比较移动都是O(n)；

归并趟数为$\log_2 n$;

时间复杂度为$O(n\log_2 n)$。

(3) 空间复杂度是O(n)。

第九章 排序

9.6.1 顺序基数排序

□ 基数排序的起源

(1)理论上可以证明，对于基于关键字之间比较的排序，无论用什么方法都至少需要进行$\log_2 n!$次比较。

(2)由Stirling公式可知，$\log_2 n! \approx n\log_2 n - 1.44n + O(\log_2 n)$。所以基于关键字比较的排序时间的下界是$O(n\log_2 n)$。因此不存在时间复杂性低于此下界的基于关键字比较的排序!

(3)只有不通过关键字比较的排序方法,才有可能突破此下界.

9.5.1 顺序基数排序

基数排序的起源

(4) 基数排序（时间复杂性可达到线性级O(n))

- 不比较关键字的大小，而根据构成关键字的每个分量的值，排列记录顺序的方法，称为分配法排序（基数排序）。
- 而把关键字各个分量所有可能的取值范围的最大值称为基数或桶或箱

(5) 基数排序的适用范围

- 显然，要求关键字分量的取值范围必须是有限的，否则可能要无限的箱。

9.5.1 顺序基数排序

算法的基本思想

- 设待排序的序列的关键字都是位相同的整数（不相同时,取位数的最大值），其位数为d，每个关键字可以各自含有d个分量，每个分量的值取值范围为0,1,…,9即基数r为10。依次从低位考查，每个分量。
- 首先把全部数据装入一个队列A，然后按下列步骤进行:

1.初态:设置10个队列，分别为Q[0],Q[1],…,Q[9]，并且均为空
2.分配:依次从队列中取出每个数据data；第pass遍处时，考查data.key右起第pass位数字，设其为r，把data插入队列Q[r]，取尽A，则全部数据被分配到Q[0],Q[1],…,Q[9].
3.收集:从Q[0]开始，依次取出Q[0],Q[1],…,Q[9]中的全部数据，并按照取出顺序，把每个数据插入排队A。
4.重复1,2,3步，对于关键字中有figure位数字的数据进行figure遍处理，即可得到按关键字有序的序列。

9.6.1 顺序基数排序

示例演示

321 986 123 432 543 018 765 678 987 789 098 890 109 901 210 012

分量个数：3
分量范围：0-9
基数：10

Q[0]:890 210
Q[1]:321 901
Q[2]:432 012
Q[3]:123 543
Q[4]:
Q[5]:765
Q[6]:986
Q[7]:987
Q[8]:018 678 098
Q[9]:789 109

890 210 321 901 432 012 123 543 765 986 987 018 678 098 789 019

9.6.1 顺序基数排序

示例演示

890 210 321 901 432 012 123 543 765 986 987 018 678 098 789 019

Q[0]:901 109
Q[1]:210 012 018
Q[2]:321 123
Q[3]:432
Q[4]:543
Q[5]:
Q[6]:765
Q[7]:678
Q[8]:986 987 789
Q[9]:890 098

901 109 210 012 018 321 123 432 543 765 678 986 987 789 890 098

9.6.1 顺序基数排序

示例演示

901 109 210 012 018 321 123 432 543 765 678 986 987 789 890 098

Q[0]:012 018 098
Q[1]:109 123
Q[2]:210
Q[3]:321
Q[4]:432
Q[5]:543
Q[6]:678
Q[7]:765 789
Q[8]:890
Q[9]: 901 986 987

012 018 098 109 123 310 321 432 543 678 765 789 890 901 986 987

9.6.1 顺序基数排序

排序过程

设待排记录A的关键字是d位的正整数。

(1) 从最低位(个位)开始，扫描关键字的第p位,把等于0的插入Q[0],…,等于9的插入Q[9]。

(2) 将Q[0],…,Q[9]中的数据依次收集到A[]中。

(3) p+1，重复执行1，2两步直到p=d

9.6.1 顺序基数排序

- 求关键字k的第p位（从后往前）算法

```
int RADIX(int k,int p)
  {
    int  power= 1 ;
    for ( int i=1; i<=p-1 ; i++ )
          power = power * 10 ;
    return      (( k%(power*10))/power) ;
  }
```

9.6.1 顺序基数排序

基数排序算法

```
void radixsort(int d,QUEUE &A){
    QUEUE Q[10]; records data;
    int p,r,i;      //p用于位数循环,r取位数对应值
    for(p=1;p<=d;p++){
      for(i=0;i<=9;i++)
          MAKENULL(Q[i])//置空队列
      while(!EMPTY(A)) {
          data=FRONT(A);//取队头元素
          DEQUEUE(A);//删除队头元素
          r=RADIX(data.key,p);//取位数值
          ENQUEUE(data,Q[r]); //入队
       }
```

9.6.1 顺序基数排序

基数排序算法

```
For(i=0;i<=9;i++)
While(!EMPTY(Q[i])) //对队列Qi中的每个元素收集到A中
{data=FRONT(Q[i]);
  DEQUEUE(Q[i]); //
  ENQUEUE(data,A);
 }
 }
}
```

9.6.1 顺序基数排序

问题分析

如果采用顺序队列，队列的长度怎么确定？

110 920 230 030 090 320 100 400

如果采用数组表示队列,队列的长度很难确定,太大造成浪费,小了会产生溢出。

一般采用链队列.

9.6.2 链式基数排序

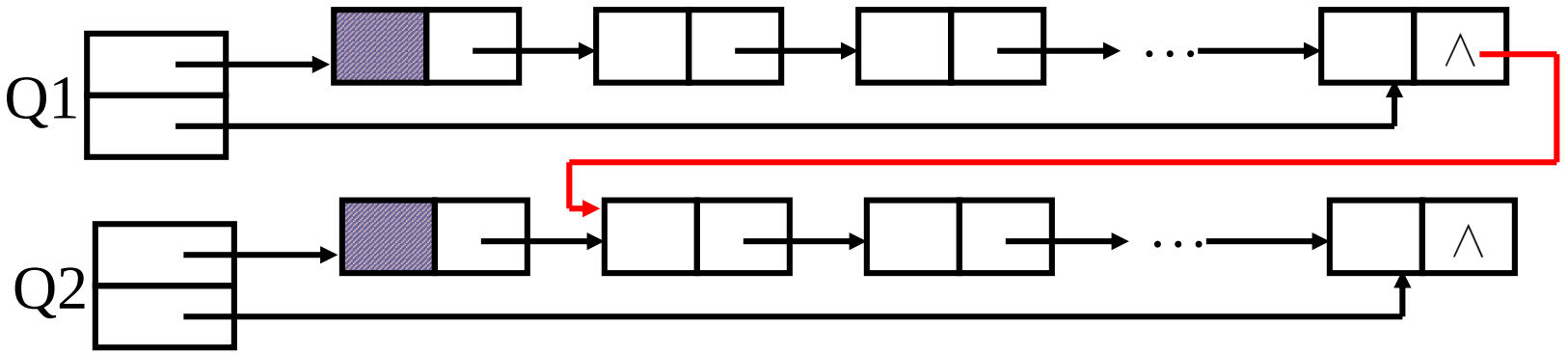
存储结构

```
struct celltype{
    records element;
    celltype *next
}//结点类型
struct QUEUE{
    celltype *front;
    celltype *rear;
}//队列定义
```

9.6.2 链式基数排序

示例演示

已经分配好；一个队列有两个指针：头指针和尾指针；
让头指针指向链表的头节点，尾指针指向最后一个节点。



收集过程：
每个队列的最后一个结点指向下一个队列的第一个结点；
所有都收集到Q0中。

9.6.2 链式基数排序

收集算法关键语句

```
void CONCATENATE(QUEUE &Q[0],
                 QUEUE &Q[1]){
    if(!EMPTY(Q[1])){
      Q[0].rear->next=Q[1].front->next;
      Q[0].rear=Q[1].rear;
    }//把Q0的最后一个节点指向Q1的第一个结点；修改Q0尾指针
}
```

9.6.2 链式基数排序

- 两种收集算法的比较

顺序收集算法

```
For(i=0;i<=9;i++)
While(!EMPTY(Q[i]))
{data=FRONT(Q[i]);
 DEQUEUE(Q[i]);
 ENQUEUE(data,A);
 } //顺序需要入队和出队操作；
```

链式收集算法

```
for(i=1;i<=9;i++)
 CONCATENATE(Q[0],Q[i]);
   A=Q[0];
```

//但是链式只需要修改指针就可以了；
（把所有的关键字都收集到Q[0]中）

9.6.2 链式基数排序

算法性能分析

n----记录数，d----关键字（分量）个数，r----基数

时间复杂度：分配操作：O（n）,收集操作O(r)，需进行d趟分配和收集。时间复杂度：O(d(n+r))

空间复杂度：所需辅助空间为队首和队尾指针2r个，此外还有为每个记录增加的链域空间n个。空间复杂度O((n+r))

算法的推广

若被排序的数据关键字由若干域组成，可以把每个域看成一个分量按照每个域进行基数排序

9.6.2 链式基数排序

例 某学校有 10000 学生，将学生信息按年龄递减排序

生日可拆分为三组关键字：年(1991~2005)、月(1~12)、日(1~31)

基数排序，时间复杂度 = O(d(n+r)) ≈ O(30000)

若采用$O(n^2)$的排序，≈$O(10^8)$

若采用$O(n\log_2 n)$的排序，≈O(140000)

基数排序擅长解决的问题：

①数据元素的关键字可以方便地拆分为 d 组，且 d 较小

②每组关键字的取值范围不大，即 r 较小 。反例：给中文人名排序

③数据元素个数 n 较大

第九章 排序

第九章 排序

- 对排序算法应该从以下几个方面综合考虑：
  - (1)时间复杂度；
  - (2)空间复杂度；
  - (3)稳定性；
  - (4)算法简单性；
  - (5)待排序记录个数n的大小；
  - (6)记录本身信息量的大小；
  - (7)关键字值的分布情况。

- (1)时间复杂度比较：

排序方法	平均情况	最好情况	最坏情况
直接插入排序	$O(n^2)$	$O(n)$	$O(n^2)$
希尔排序	$O(n^{1.3})$		$O(n^2)$
起泡排序	$O(n^2)$	$O(n)$	$O(n^2)$
快速排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n^2)$
直接选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$
堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$
(二路)归并排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$
基数排序	$O(d(n+r))$	$O(d(n+r))$	$O(d(n+r))$

- (2)空间复杂度比较和(3)稳定性比较：

排序方法	辅助空间	稳定性/不稳定举例
直接插入排序	$O(1)$	是
希尔排序	$O(1)$	否/3,2,2’(d=2,d=1)
起泡排序	$O(1)$	是
快速排序	$O(\log_2 n)$ ~$O(n)$	否/2,2’,1
直接选择排序	$O(1)$	否/2,2’,1
堆排序	$O(1)$	否/1,2,2’(最小堆)
(二路)归并排序	$O(n)$	是
基数排序	$O(n+r)$	是

第九章 排序

- (4)算法简单性比较：从算法简单性看，
  - 一类是简单算法，包括直接插入排序、直接选择排序和起泡排序；
  - 另一类是改进后的算法，包括希尔排序、堆排序、快速排序和归并排序，这些算法都很复杂。

- (5)待排序的记录个数比较：从待排序的记录个数n的大小看，
  - n越小，采用简单排序方法越合适；
  - n越大，采用改进的排序方法越合适。
  - 因为n越小，$O(n^2)$同$O(n\log_2 n)$的差距越小，并且输入和调试简单算法比输入和调试改进算法要少用许多时间。

第九章 排序

- (6)记录本身信息量比较：
  - 记录本身信息量越大，移动记录所花费的时间就越多，所以对记录的移动次数较多的算法不利。
- (7)关键字值的分布情况比较：

  当待排序记录按关键的值有序时，
  - 插入排序和起泡排序能达到$O(n)$的时间复杂度；
  - 对于快速排序而言，这是最坏的情况，此时的时间性能蜕化为$O(n^2)$；
  - 选择排序、堆排序和归并排序的时间性能不随记录序列中关键字的分布而改变。

第九章 排序

- (1)若 *n* 较小(如 *n* ≤ 50)，可采用直接插入或直接选择排序。

  当记录规模较小时，直接插入排序较好；否则因为直接选择移动的记录数少于直接插人，应选直接选择排序为宜。

- (2)若文件初始状态基本有序(指正序)，则应选用直接插人、冒泡或随机的快速排序为宜；且以直接插入排序最佳。

- (3)若 *n* 较大，则应采用时间复杂度为O(*n*lg*n*)的排序方法：

- 快速排序、堆排序或归并排序。

第九章 排序

- 快速排序是目前基于比较的内部排序中被认为是最好的方法，当待排序的关键字是随机分布时，快速排序的平均时间最短；
- 堆排序所需的辅助空间少于快速排序，并且不会出现快速排序可能出现的最坏况。这两种排序都是不稳定的。
- 基数排序适用于 *n* 值很大而关键字较小的序列。
- 若要求排序稳定，则可选用归并排序，基数排序稳定性最佳。

第九章 排序

对100万个数据排序统计结果(单位：毫秒)

序号	排序方法	平均情况	最坏情况（逆序）	最好情况（正序）
1	冒泡排序	549432.000	1534035.000	366936.000
	选择排序	478694.000	587240.000	367658.000
	插入排序	253115.000	515621.000	0.897
2	希尔排序/增量 3	61.000	203.000	35.000
3	堆排序	79.000	126.000	74.800
4	归并排序	70.000	140.000	61.000
5	快速排序	39.000	93.000	30.000
6	基数排序/进制 100	117.000	118.000	116.000
	基数排序/进制 1000	89.000	90.000	88.000

第九章 排序

部分算法的时间效率比较 （单位： 毫秒）

序号	10	100	1K	10K	100K	1M
冒泡排序	0.000276	0.005643	0.545	61.000	8174.000	549432
选择排序	0.000237	0.006438	0.488	47.000	4717.000	478694
插入排序	0.000258	0.008619	0.764	56.000	5145.000	515621
希尔排序/增量 3	0.000522	0.003372	0.036	0.518	4.152	61
堆排序	0.000450	0.002991	0.041	0,531	6.506	79
归并排序	0.000723	0.006225	0.066	0.561	5.480	70
快速排序	0.000291	0.003051	0.030	0.311	3.634	39
基数排序/进制100	0.005181	0.021000	0.165	1.650	11.428	117
基数排序/进制1000	0.016134	0.026000	0.139	1.264	8.394	89

*来自于学生测试数据

本章小结

- ✓ 熟练掌握：
  - 直接插入排序、希尔排序、冒泡排序、快速排序、简单选择排序、堆排序、归并排序、基数排序的思想和算法。充分了解各种排序算法的应用背景和优缺点。
- ✓ 重点学习：
  - 加强各种排序算法在实际应用中的训练，提高实际应用水平。

结课啦！

Good luck！