# 数据结构

李春葆

清华大学

## 排序

# 目录 | CONTENTS

01

*Part One*

# 排序的基本定义

# 性能比较标准3个指标

- 空间复杂度
- 时间复杂度
- 稳定性

**稳定性**

**排序过后能使值相同的数据保持原顺序中的相对位置**

| | | | | |
|---|---|---|---|---|
| 49 | 32 | 56 | 49 | 27 |
| 27 | 32 | 49 | 49 | 56 |

**不稳定性**

**排序过后不能使值相同的数据保持原顺序中的相对位置**

# 基本定义

- 【内部排序】：待排序文件的全部记录存放在内存进行的排序，称为内部排序。
- 【外部排序】：借助外部的辅助存储器（比如：硬盘），由于数据是存在外存中，故数据不可随机被存取 。

# 排序基本操作

（1）比较两个排序码的大小；

（2）改变指向记录的指针或移动记录本身。

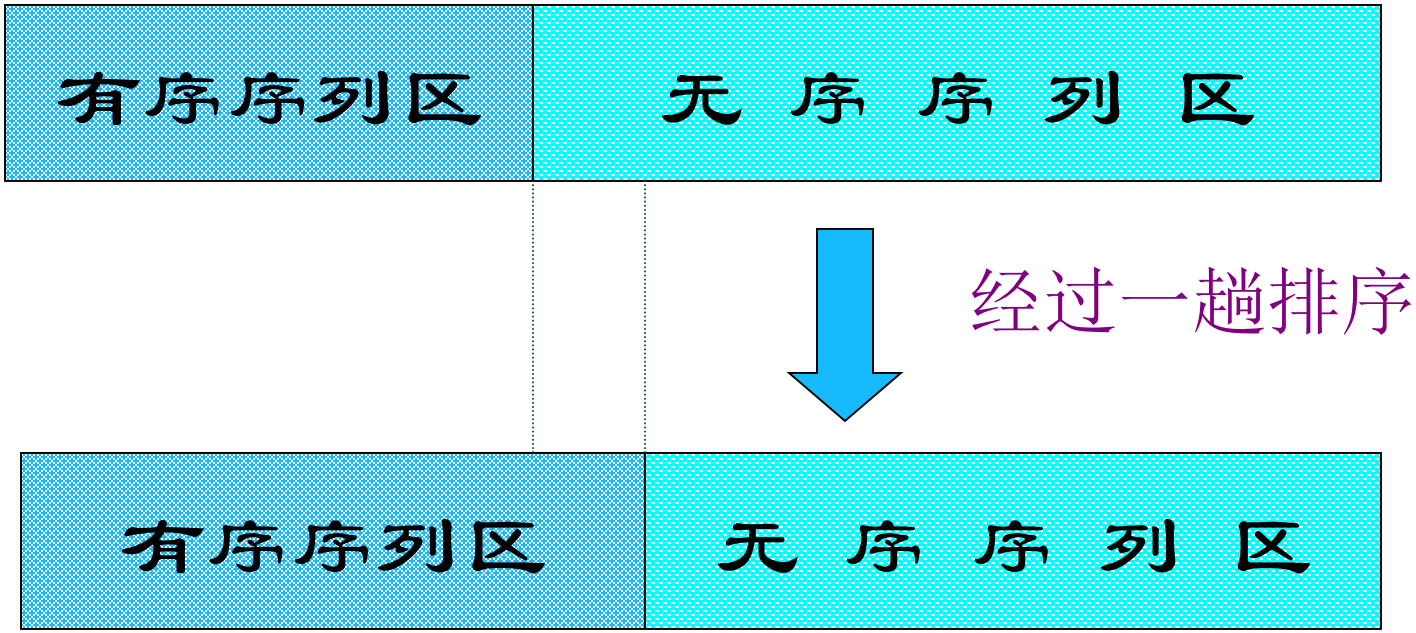
- **注意：**

  **第(2)种基本操作的实现依赖于待排序记录的存储方式。**

- 所以排序的时间开销可用算法执行中的**数据比较次数**与**数据移动次数**来衡量。

# 内排序的基本方法

- 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。



# 基本定义

- 【静态排序】：
  - 排序的过程是对数据对象本身进行物理地重排，经过比较和判断，将对象移到合适的位置。这时，数据对象一般都存放在一个顺序的表中。
- 【动态排序】：
  - 给每个对象增加一个链接指针，在排序的过程中不移动对象或传送数据，仅通过修改链接指针来改变对象之间的逻辑顺序，从而达到排序的目的。

# 基本定义--内排序分类

- 按排序过程依据的原则分为：

  插入排序

  交换排序

  选择排序

  归并排序

  基数排序

- 按排序过程所需的工作量分：

  简单排序　$O(n^2)$

  先进排序　$O(n\log n)$

  基数排序　$O(d.\ n)$

| 排序算法 | 时间复杂度 | 是否基于比较 |
| --- | --- | --- |
| 冒泡、插入、选择 | $O(n^2)$ | ✓ |
| 快排、归并 | $O(n\log n)$ | ✓ |
| 桶、计数、基数 | $O(n)$ | ✗ |

# 基本定义

## 排序算法执行效率从这几个方面来衡量

- 1. 排序的时间开销
  - 平均情况　最好情况 最坏情况
- 2. 比较次数和交换（或移动）次数
- 3. 时间复杂度的系数、常数 、低阶
  - 但是实际的软件开发中，排序的可能是 10 个、 100 个、 1000 个这样规模很小的数据，所以，在对同一阶时间复杂度的排序算法性能对比的时候，就要把系数、常数、低阶也考虑进来。

# 基本定义

## 排序算法的内存消耗

- 算法的内存消耗可以通过空间复杂度来衡量，排序算法也不例外
- 原地排序（Sorted in place）：就是特指空间复杂度是O(1)的排序算法。。

# 基本定义

## 排序算法的稳定性

- 为什么要考察排序算法的稳定性呢？

# 基本定义

## 例：订单排序

- 订单有两个属性：下单时间和订单金额。若有 10 万条订单数据，希望按照金额从小到大对订单数据排序。对于金额相同的订单，希望按照下单时间从早到晚有序。对于这样一个排序需求，我们怎么来做呢？

# 基本定义

- 【方案1】：先按照金额对订单数据进行排序，然后，再遍历排序之后的订单数据，对于每个金额相同的小区间再按照下单时间排序。
- 【缺点】：这种排序思路理解起来不难，但是实现起来会很复杂。

# 基本定义

- 【方案2】：先按照下单时间给订单排序，注意是按照下单时间，不是金额。排序完成之后，用稳定排序算法，按照订单金额重新排序。两遍排序之后，得到的订单数据就是按照金额从小到大排序，金额相同的订单按照下单时间从早到晚排序的。为什么呢？

## 按下单时间有序

| ID | 下单时间 | 金额 |
| --- | --- | --- |
| 1 | 2018-9-3 15:06:07 | 50 |
| 2 | 2018-9-3 16:08:10 | 30 |
| 3 | 2018-9-3 18:01:33 | 40 |
| 4 | 2018-9-3 20:23:31 | 30 |
| 5 | 2018-9-3 22:15:13 | 30 |
| 6 | 2018-9-4 05:07:33 | 60 |

## 按金额重新排序

| ID | 下单时间 | 金额 |
| --- | --- | --- |
| 1 | 2018-9-3 16:08:10 | 30 |
| 2 | 2018-9-3 20:23:31 | 30 |
| 3 | 2018-9-3 22:15:13 | 30 |
| 4 | 2018-9-3 18:01:33 | 40 |
| 5 | 2018-9-3 15:06:07 | 50 |
| 6 | 2018-9-4 05:07:33 | 60 |

# 02 *Part Two*

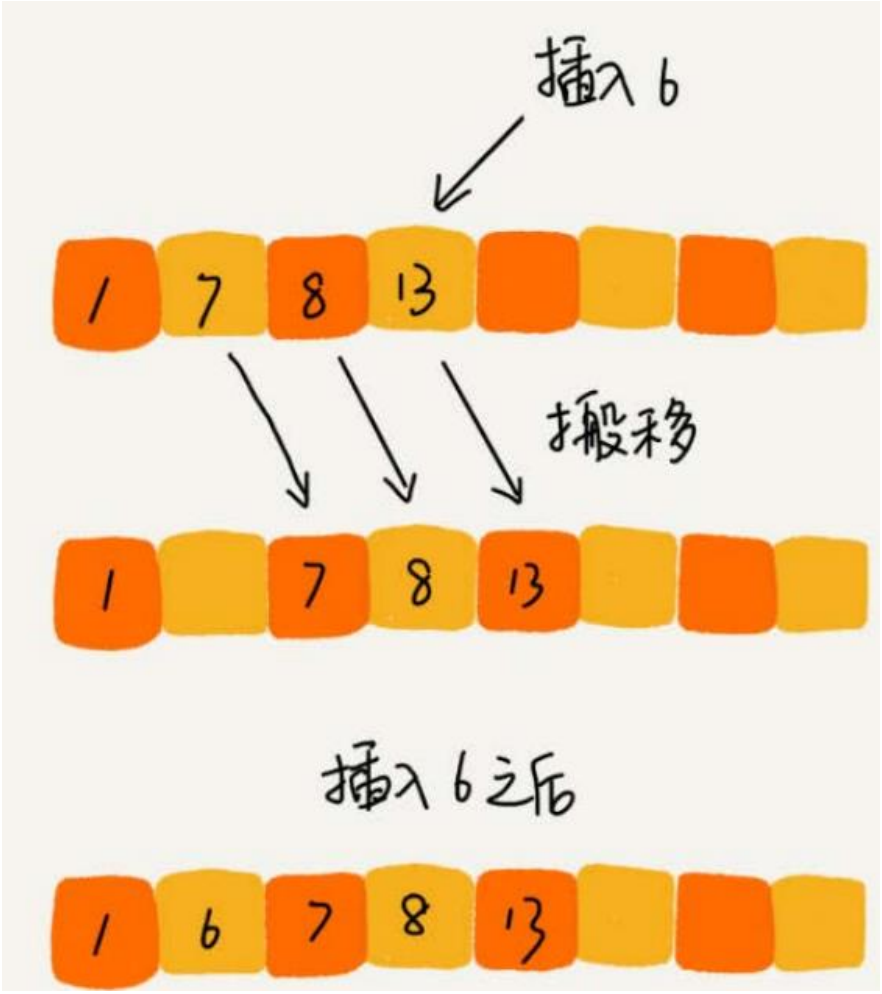
# 插入排序


# 直接插入排序

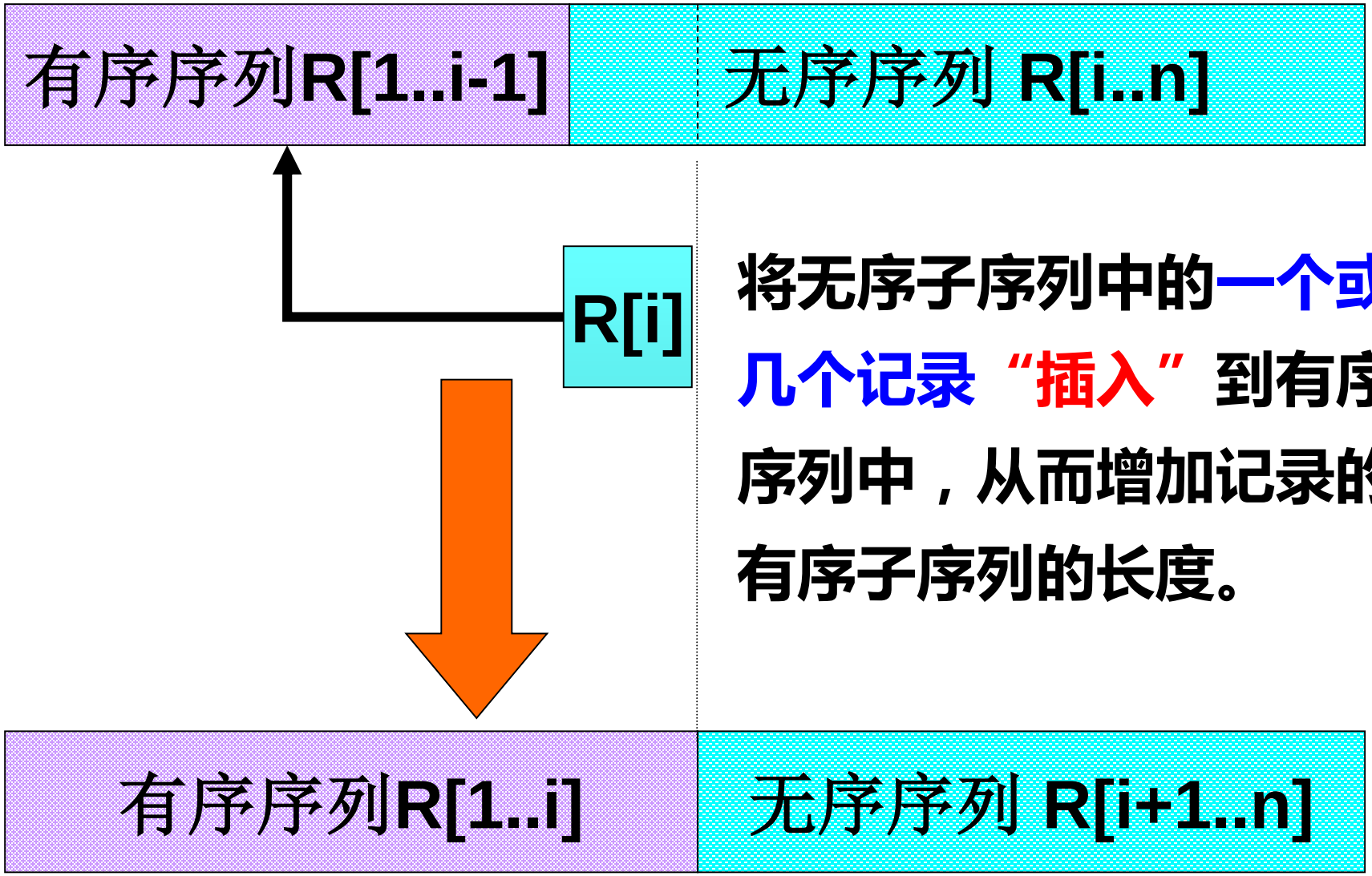
# 插入排序—直接插入排序

- 【直接插入排序】（最简单的排序方法）
- 【排序思想】：依次将每个待排序的记录插入到一个有序子文件的合适位置(有序子文件记录数增1)

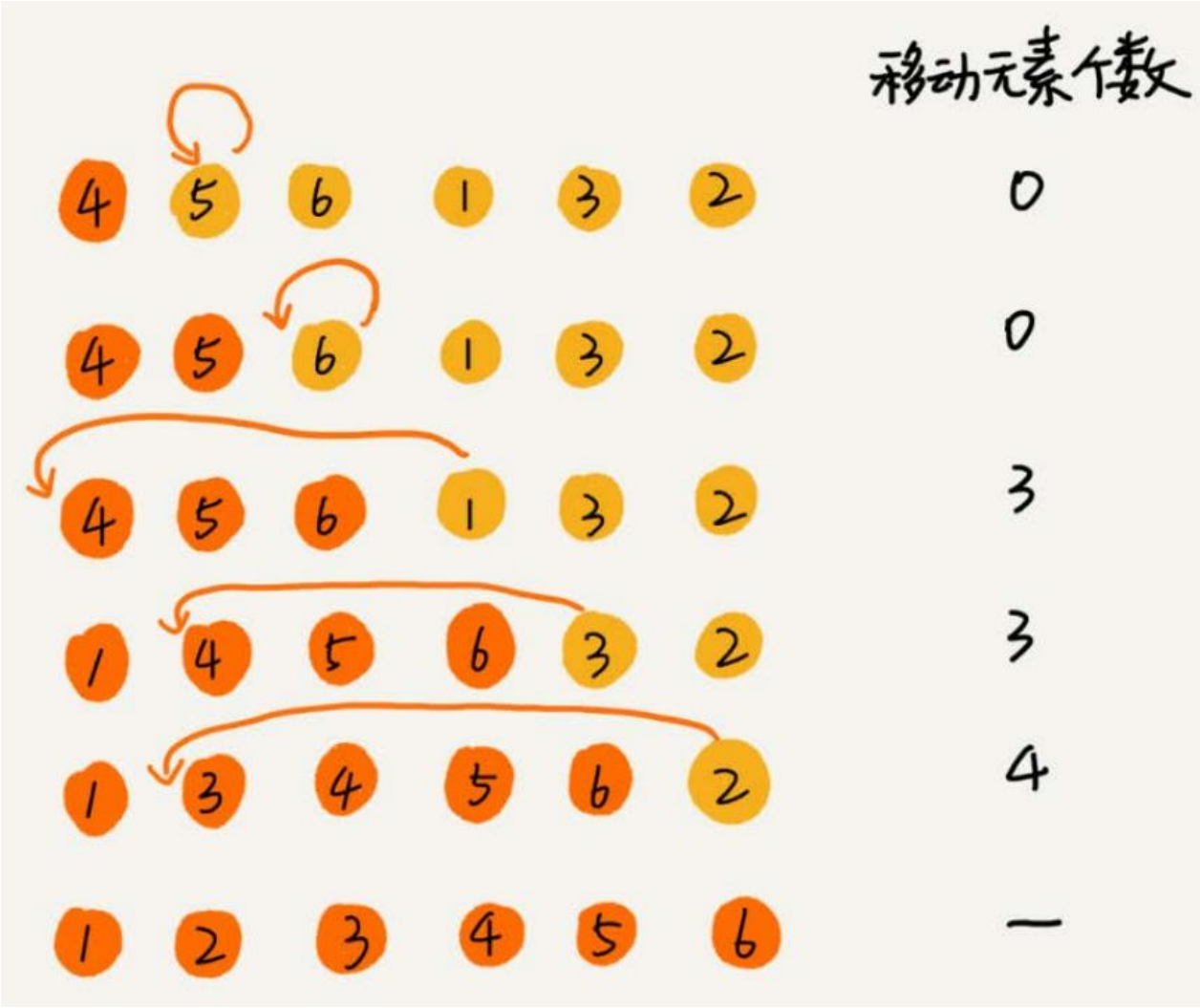
- 动态排序的过程，即动态地往有序集合中添加数据



# 一趟直接插入排序的基本思想

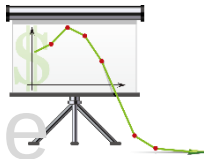



将无序子序列中的一个或几个记录“插入”到有序序列中，从而增加记录的有序子序列的长度。


移动元素个数
4 5 6 1 3 2
0
4 5 6 1 3 2
0
4 5 6 1 3 2
3
1 4 5 6 3 2
3
1 3 4 5 6 2
4
1 2 3 4 5 6
—

- 【例】：已有待排序文件为：21，25，49，25*，16
  - 首先将文件的第一个记录，视为有序文件，然后从第二个记录开始，直到最后一个记录，依次将他们插入到有序文件的合适位置。

# 插入排序—直接插入排序



```
void InsertSort(RecType R[],int n) /*对R[0..n-1]按递增有序进行直接插入排序*/
{      int i,j;     RecType temp;
       for (i=1;i<n;i++)
       {   temp=R[i];
         j=i-1;  /*从右向左在有序区R[0..i-1]找R[i]的插入位置*/
          while (j>=0 && temp.key<R[j].key)
          {     R[j+1]=R[j];   /*将关键字大于R[i].key的记录后移*/
                 j--;
          }
          R[j+1]=temp;       /*在j+1处插入R[i]*/
       }
}
```

# 插入排序—直接插入排序

- 【例】设待排序的表有10个记录，其关键字分别为{9, 8, 7, 6, 5, 4, 3, 2, 1, 0}。说明采用直接插入排序方法进行排序的过程。

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 初始关键字 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| i=1 | [8 | 9] | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| i=2 | [7 | 8 | 9] | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| i=3 | [6 | 7 | 8 | 9] | 5 | 4 | 3 | 2 | 1 | 0 |
| i=4 | [5 | 6 | 7 | 8 | 9] | 4 | 3 | 2 | 1 | 0 |
| i=5 | [4 | 5 | 6 | 7 | 8 | 9] | 3 | 2 | 1 | 0 |
| i=6 | [3 | 4 | 5 | 6 | 7 | 8 | 9] | 2 | 1 | 0 |
| i=7 | [2 | 3 | 4 | 5 | 6 | 7 | 8 | 9] | 1 | 0 |
| i=8 | [1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9] | 0 |
| i=9 | [0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9] |

# 插入排序—直接插入排序

- 【算法分析】
- (1) 空间上，只需i，j两个整型变量和一个记录的辅助空间r[0]，r[0]作为“监视哨”，控制待插入元素相对于有序子文件为最小时WHILE循环的结束，同时也用于暂存待插入元素。

# 插入排序—直接插入排序

- (2) 时间上，只包含比较关键字和移动记录两种操作。
  - ①比较次数 ：

1. 当待初始文件按关键字递增有序(正序)时：

a. 对每个记录只进行一次比较，整个排序过程共

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

进行了$n-1$次比较(最少)；

b. 每个记录都要进行r[i]移到r[0]和r[0]移到r[j+1]两次移动，因此共移动2(n-1)次(最少)。

# 插入排序—直接插入排序

- 2. 当待排序文件按关键字非递增有序(逆序)时

① 记录r[i](i=2,3,...n)均要和前i-1个记录及r[0]进行比较，整个排序过程共进行了

$\sum_{i=2}^{n}$ i=(n+2)(n-1)/2次比较(最多)；

# 插入排序—直接插入排序

② 移动记录次数：

每个记录都要进行r[i]移动到r[0]和r[0]移动到r[j+1]两次移动，另外当r[i].key＜r[j].key时，还将r[j]移动到r[j+1]，所以，当初始文件为正序时，移动记录次数最少为2（n-1）次，当初始文件为逆序时移动记录次数最多为

$$\sum_{i=2}^{n}(i-1)+2(n-1)=(n+4)(n-1)/2$$次(最多)。

# 问题?

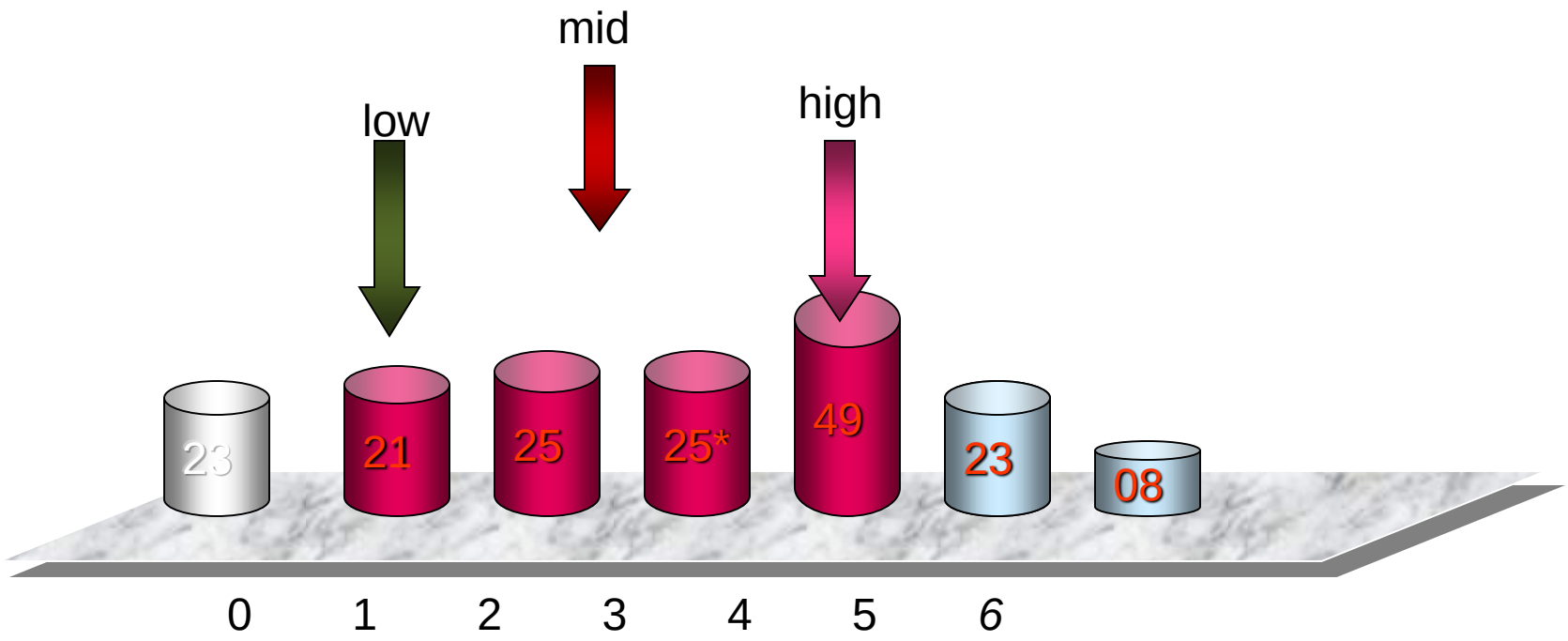
- 第一，插入排序是原地排序算法吗？
  - 插入排序算法的运行并不需要额外的存储空间，所以空间复杂度是 O(1)，也就是说，这是一个原地排序算法。
- 第二，插入排序是稳定的排序算法吗？
  - 插入排序是稳定的排序算法，“<”保证的
- 第三，插入排序的时间复杂度是多少？
  - 最好情况：数据已经是有序的 O(n)
  - 最坏情况：数组是倒序的 $O(n^2)$
  - 平均时间复杂度为 $O(n^2)$


# 折半插入排序

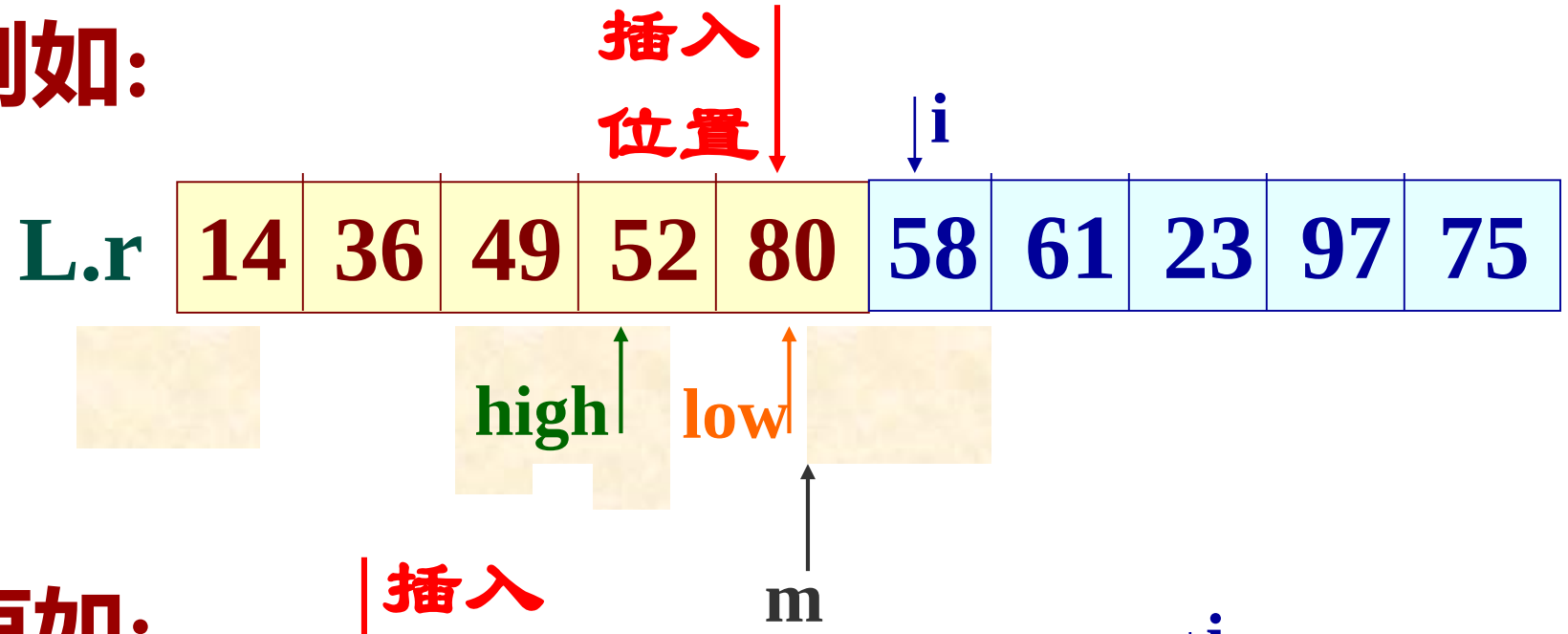
# 插入排序—折半插入排序

- 【折半插入排序】
- 由于是在有序子文件中确定插入的位置，因此可用折半查找来代替直接插入排序法中的顺序查找，从而可减少比较次数。

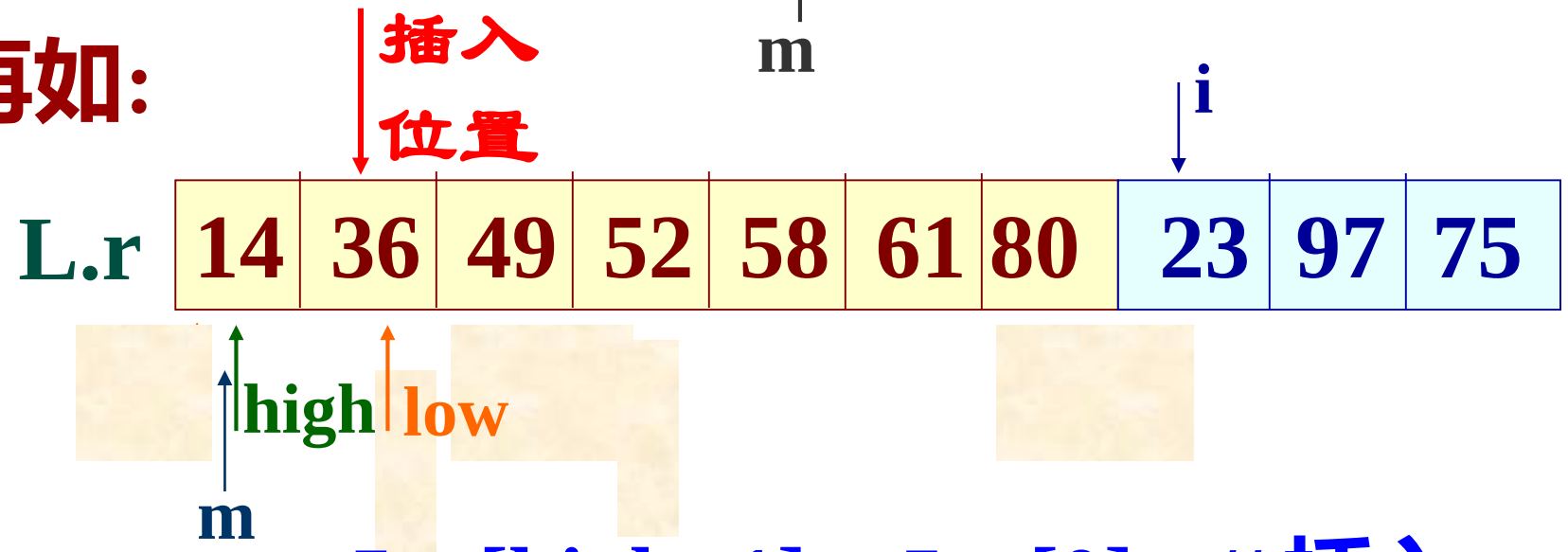
# 插入排序—折半插入排序



例如:

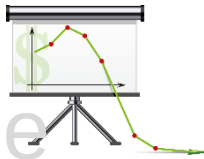



再如:




```
L.r[high+1] = L.r[0];  // 插入
```

# 插入排序—折半插入排序



```
Void  binsort(sqlist  &L ,int  n  )
    { FOR (i=2 ; I<=n ; ++I)
        {  l[0]=l[I];       low=1; high = I-1 ;
           WHILE (low<=high )
             {m=(low+high ) / 2 ;
               IF (L[0]＜L[m])     //＜确保稳定，若改为≤，则不稳定
                 }
                   high ＝m-1  ELSE low =m+1; }
            FOR (j=i-1 ; j>=high+1; --j)  L[j+1]=L[j];
                L[high+1]=L[0] }      //后移
      }
```

确定插入位置所进行的折半查找，关键码的比较次数至多为 $\lceil \log_2(n+1) \rceil$ 次，移动记录的次数和直接插入排序相同，故时间复杂度仍为$O(n^2)$。是一个稳定的排序方法。移动次数未变，故仍为$O(n^2)$

# 算法分析

- (1)稳定性：稳定
- (2)空间复杂度：O(1)
- (3)时间复杂度
- 比较次数与待排记录的初始顺序无关，只依赖记录个数；
- 插入每个记录需要$O(\log_2 i)$次比较
- 最多移动i+1次，最少2次（临时记录）
- 最佳情况下总时间代价为$O(n\log_2 n)$，最差和平均情况下仍为$O(n^2)$。

# 插入排序—希尔排序

- 【希尔排序（Shell`s Methool）】（又称为缩小增量排序）

  直接插入排序算法简单，在n值较小时，效率比较高，在n值很大时，若序列按关键码基本有序，效率依然较高，其时间效率可提高到O(n)。希尔排序即是从这两点出发，给出插入排序的改进方法

- 【1. 基本思想】：
- 分割成若干个较小的子文件，对各个子文件分别进行直接插入排序，当文件达到基本有序时，再对整个文件进行一次直接插入排序。


# 希尔排序

# 插入排序—希尔排序

- 【2. 依据】：
- (1) 若待排序文件"基本有序"，即文件中具有特性：

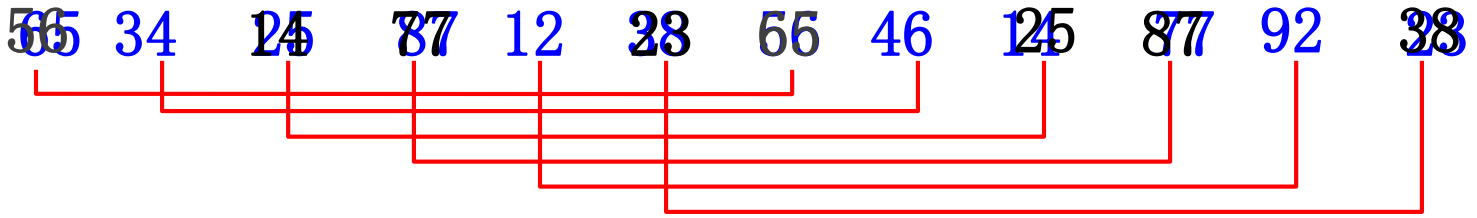
L[i] ＜Max{L[j ]} 1≤j＜i的记录数较少，

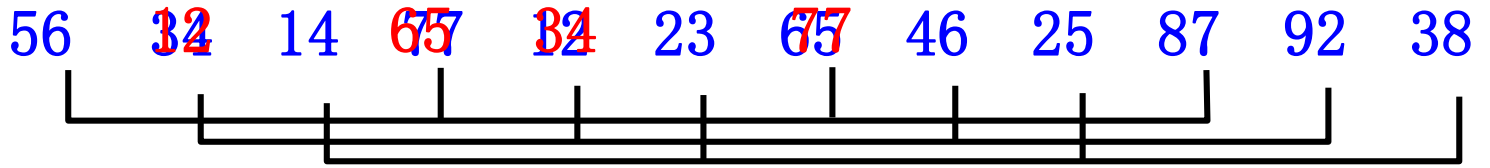
则文件中大多数记录都不需要进行插入。

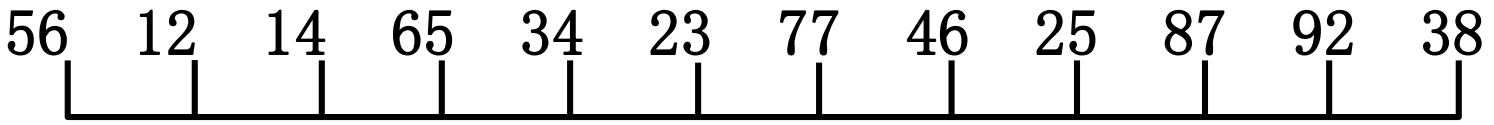
- (2) 基本有序时，直接插入排序效率可以提高，接近于O(n)。

# 插入排序—希尔排序

- 【例】 设待排序的表有12个记录说明采用希尔排序方法进行排序的过程。

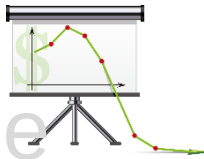



结果序列 12 14 23 25 34 38 46 56 65 77 87 92

# 插入排序—希尔排序



```
void ShellSort(RecType R[],int n)  /*希尔排序算法*/
{    int i,j,d;RecType temp;
     d=n/2;                                /*d取初值n/2*/
     while (d>0)
     {   for (i=d;i<n;i++) /*将R[d..n-1]分别插入各组有序区*/
         {  j=i-d;
            while (j>=0 && R[j].key>R[j+d].key)
            {   temp=R[j];                 /*R[j]与R[j+d]交换*/
                R[j]=R[j+d];R[j+d]=temp;
                j=j-d;
            }
         }
         d=d/2;                            /*递减增量d*/
     }
}
```

希尔排序的时间复杂度随d值取法的不同而不同，但d值的取法并无定式。需保证最后一个增量必须为1。

# 插入排序—希尔排序

- 【希尔排序的特点】

(1) 子文件（子序列）的构成不是简单地“逐段分割”，而是将相隔某个“增量”的记录组成一个子文件。

(2) 增量序列应是递减，且最后一个必须为1。

如上例的增量序列为：dlta[0..2],其值分别为：6，3，1

(3) **时间复杂度**

$$O(n^{3/2})$$

(4) 稳定性

**不稳定**

# 03 *Part Three*

# 交换排序

# 交换排序（ Exchange Sort ）

- 交换排序的基本思想是两两比较待排序对象的关键码，如果发生逆序(即排列顺序与排序后的次序正好相反)，则交换之，直到所有对象都排好序为止。

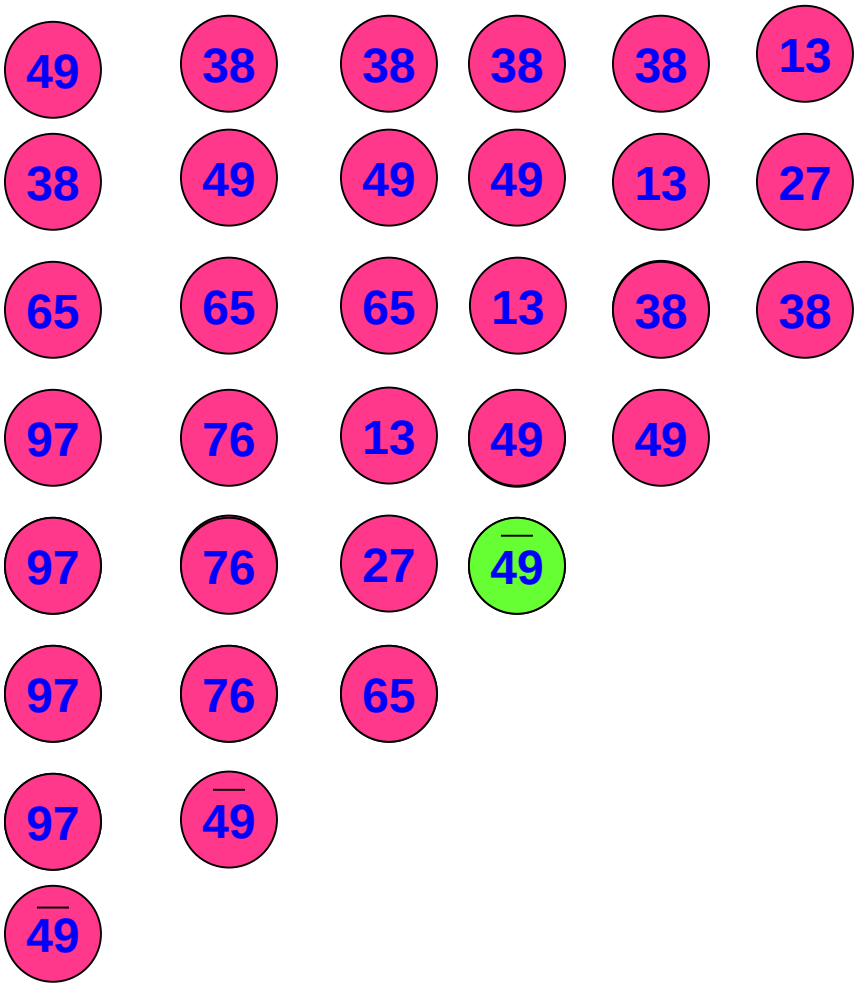

# 冒泡排序

# 交换排序—冒泡排序

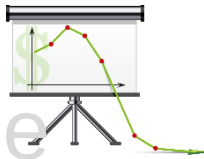
- 【基本思想】：
- 是通过对待排序序列从后向前（从下标较大的元素开始），依次比较相邻元素的排序码，若发现逆序则交换，使排序码较小的元素逐渐从后部移向前部（从下标较大的单元移向下标较小的单元），就象水底下的气泡一样逐渐向上冒。
- 因为排序的过程中，各元素不断接近自己的位置，如果一趟比较下来没有进行过交换，就说明序列有序，因此要在排序过程中设置一个标志flag判断元素是否进行过交换。从而减少不必要的比较。

**【特点】：**

- n个数排序共需进行n-1趟比较
- 第i趟共需要进行n-i次比较



# 交换排序—冒泡排序(原始算法)



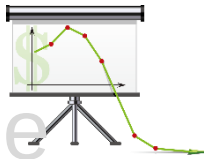
```
void bubblf_sort(SqList &L) {//自小至大有序
for(i=1,i<=L.length-1;++i)
    for(j=1;j<L.length-1;++j)
        if(LT(L.r[j+1].key,L.r[j].key))
            L.r[j] ←→L.r[j+1]
}//bubblf_sort
```

1 15 24 6 17 5 ⟶ 1 15 6 17 5 24

1 15 6 17 5 24 ⟶ 1 6 15 5 17 24

**假设L.length=n，则总共比较$n^2$次**

# 交换排序—冒泡排序（改进算法）



```
void bubblf_sort(SqList &L) {//自小至大有序
for(i=1;i<=L.length-1;++i)
    for(j=1;j<L.length-i;++j)  //n-i已经有序
      if(L.r[j+1].key<L.r[j].key))
        L.r[j] ←→L.r[j+1]
}//bubblf_sort
```

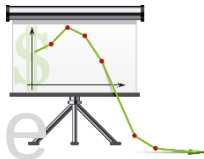
1 15 24 6 17 5 ⟶ 1 15 6 17 5 24

1 15 6 17 5 24 ⟶ 1 6 15 5 17 24

注意第i趟比较了n-i次，则可以有效的减少比较次数

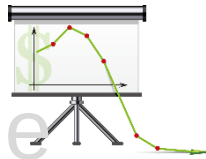
**比较的总的次数是1+2+…+n-1=n(n-1)/2**

# 交换排序—冒泡排序




```
void BubbleSort(RecType R[],int n)
{      int i,j;     RecType temp;
       for (i=0;i<n-1;i++)
       {    for (j=n-1;j>i;j--)       /*比较找本趟最小关键字的记录*/
                if (R[j].key<R[j-1].key)
              {    temp=R[j];                /*R[j]与R[j-1]进行交换*/
                   R[j]=R[j-1];
                   R[j-1]=temp;
              }
       }
}
```

# 交换排序—冒泡排序



**问题：刚才的改进算法是否已经最优？**

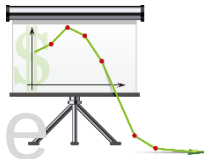
1 2 3 4 5 6 7 8 9 → 1 2 3 4 5 6 7 8 9

1 2 3 4 5 6 7 8 9 → 1 2 3 4 5 6 7 8 9

**分析：因为给定的待排序序列已经是一个正序的序列，经过第一趟的比较以后，已经没有交换发生，但是算法仍然会继续执行。**

**解决：添加一个算法的结束条件，即当某一趟排序过程中只有比较而没有交换的时候，就认定序列已经有序，可以提前结束循环。**

# 交换排序—冒泡排序




```
void BubbleSort(RecType R[],int n)
   {    int i,j,exchange;RecType temp;
        for (i=0;i<n-1;i++)
        {   exchange=0;
            for (j=n-1;j>i;j--)      /*比较,找出最小关键字的记录*/
                if (R[j].key<R[j-1].key)
                {  temp=R[j]; R[j]=R[j-1];R[j-1]=temp;
                   exchange=1;
                }
          if (exchange==0)   return;          /*中途结束算法*/
        }
   }
```

# 交换排序—冒泡排序

- 【例】 设待排序的表有10个记录，其关键字分别为{9, 8, 7, 6, 5, 4, 3, 2, 1, 0}。说明采用冒泡排序方法进行排序的过程。

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 初始关键字 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| i=0 | 0 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
| i=1 | 0 | 1 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 |
| i=2 | 0 | 1 | 2 | 9 | 8 | 7 | 6 | 5 | 4 | 3 |
| i=3 | 0 | 1 | 2 | 3 | 9 | 8 | 7 | 6 | 5 | 4 |
| i=4 | 0 | 1 | 2 | 3 | 4 | 9 | 8 | 7 | 6 | 5 |
| i=5 | 0 | 1 | 2 | 3 | 4 | 5 | 9 | 8 | 7 | 6 |
| i=6 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 9 | 8 | 7 |
| i=7 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 9 | 8 |
| i=8 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

# 交换排序—冒泡排序

最好的情况（关键字在记录序列中顺序有序）：

只需进行一趟冒泡

“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趟冒泡

“比较”的次数：

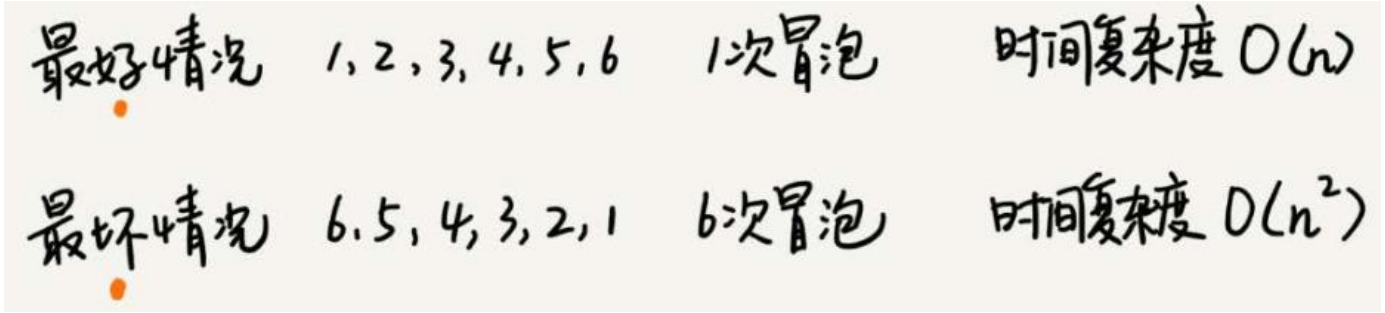
$$\sum_{i=0}^{n-1}(n-i-1)=\frac{n(n-1)}{2}$$

“移动”的次数：

$$\sum_{i=0}^{n-2}3(n-i-1)=\frac{3n(n-1)}{2}$$

# 交换排序—冒泡排序

- 【冒泡排序的效率分析】
- 从冒泡排序的算法可以看出



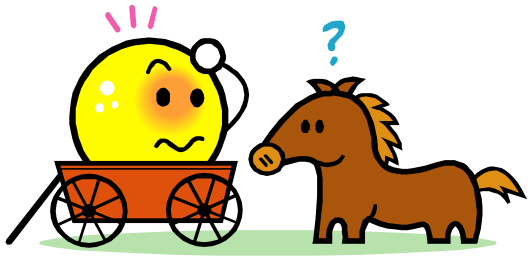

  - 若待排序为正序，需一趟排序，比较次数为（n-1）次，移动元素次数为0；
  - 若待排序为逆序，n-1趟排序，比较次数为$(n^2-n)/2$，移动次数为$3(n^2-n)/2$
  - 由于元素移动较多，属于内排序中速度较慢的一种。

# 交换排序—冒泡排序

- 平均时间复杂度

  $O(n^2)$

- 因为冒泡排序算法只进行元素间的顺序移动，所以是一个稳定的算法。
- 空间复杂度：$O(1)$



# 问题?

- 冒泡排序和插入排序的时间复杂度都是$O(n^2)$，都是原地排序算法，为什么插入排序要比冒泡排序更受欢迎呢？

插入排序中数据的移动操作：

```
if (a[j] > value) {
  a[j+1] = a[j];  // 数据移动
} else {
  break;
}
```

冒泡排序中数据的交换操作：

```
if (a[j] > a[j+1]) { // 交换
   int tmp = a[j];
   a[j] = a[j+1];
   a[j+1] = tmp;
   flag = true;
}
```


# 快速排序

# 交换排序—快速排序

- 快速排序是目前内部排序中最快的方法。
- 【基本思想】：
  - （1）找一个记录，以它的关键字作为“枢轴”，凡其关键字小于枢轴的记录均移动至该记录之前，反之，凡关键字大于枢轴的记录均移动至该记录之后。

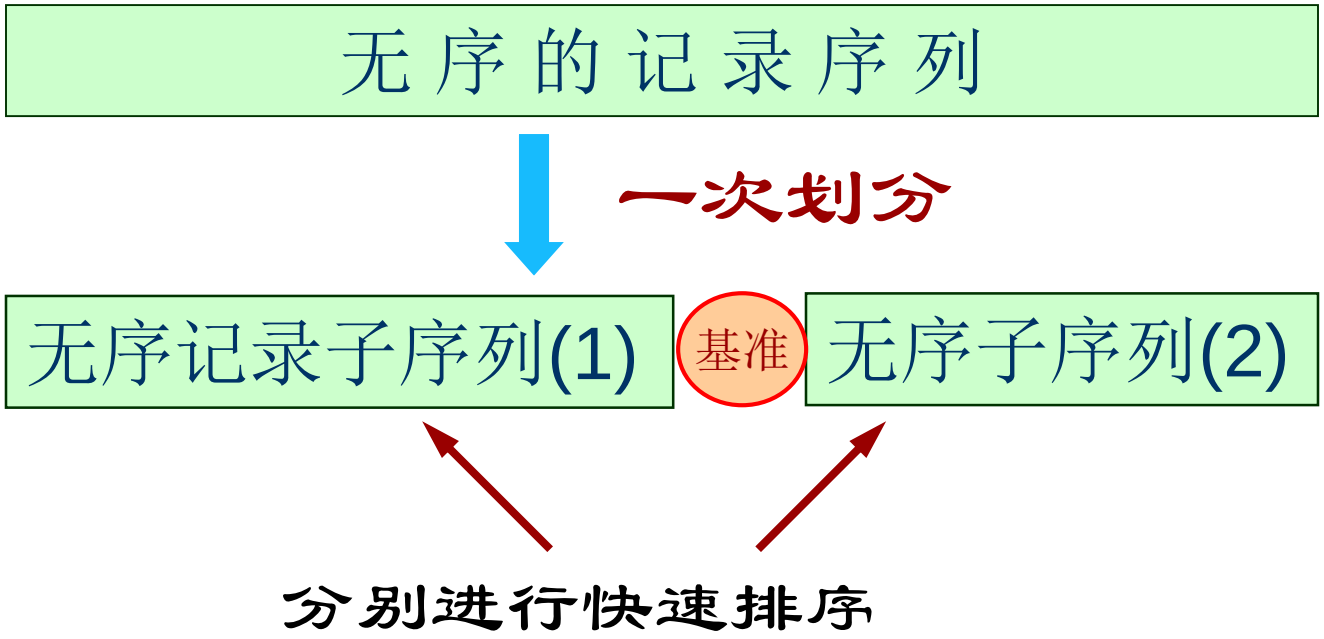
**L[s..i-1]　枢轴　L[i+1..t]**

  - （2）对分割 后的两个子表重复上述过程，直到子表的表长不超过1为止。

# 交换排序—快速排序

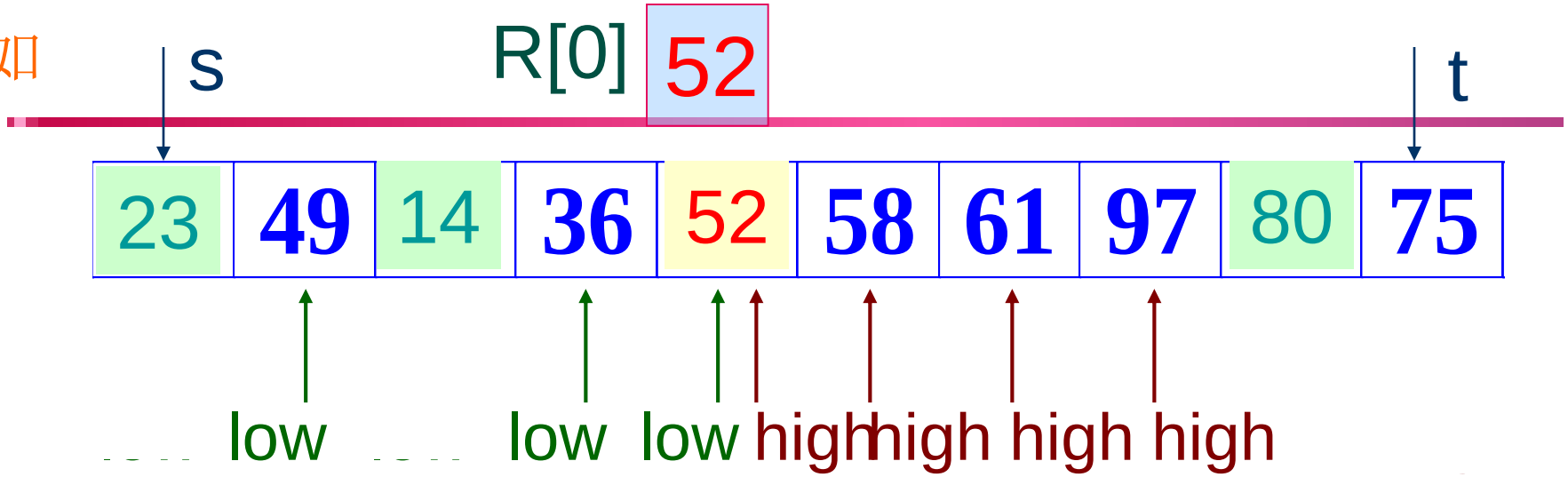
- 快速排序是对冒泡排序的一种改进方法
  - **由于在冒泡排序的扫描过程中只对相邻的元素进行比较，因此，在交换两个元素时，只能消除一个逆序。**
  - **如果能通过两个不相邻元素的交换，消除多个逆序，则会大大加快排序的速度。快速排序可以实现之。**

首先对无序的记录序列进行“一次划分”，之后分别对分割所得两个子序列“递归”进行快速排序。




需要确定排序的序列，采用什么存储结构？？？

例如




设 R[s]=52 为枢轴

将 R[high].key 和基准的关键字进行比较，要求R[high].key ≥基准的关键字

将 R[low].key 和基准的关键字进行比较，要求R[low].key ≤基准的关键字

可见，经过“一次划分”，将关键字序列

52, 49, 80, 36, 14, 58, 61, 97, 23, 75

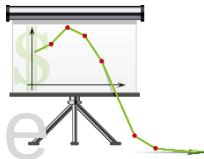
调整为： 23, 49, 14, 36, (52) 58, 61, 97, 80, 75

在调整过程中，设立了两个指针： low 和high，它们的初值分别为： s 和 t。

之后逐渐减小 high，增加 low，并保证

R[high].key＞52，和 R[low].key＜52,否则进行记录的“交换”。

# 交换排序—快速排序

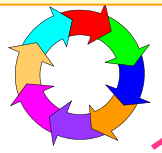



```
void QuickSort(RecType R[],int s,int t)
{   int i=s,j=t;   RecType temp;
    if (s<t)                                  /*区间内至少存在一个元素的情况*/
    {     temp=R[s];                          /*用区间的第1个记录作为基准*/
          while (i!=j)                        /*从两端交替向中间扫描,直至i=j为止*/
          {   while (j>i && R[j].key>temp.key)     j--;
              if (i<j)                        /*表示找到这样的R[j],R[i]、R[j]交换*/
              {   R[i]=R[j]; i++;  }
              while (i<j && R[i].key<temp.key)     i++;
              if (i<j)                        /*表示找到这样的R[i],R[i]、R[j]交换*/
              {   R[j]=R[i]; j--;   }
          }
          R[i]=temp;
          QuickSort(R,s,i-1);                 /*对左区间递归排序*/
          QuickSort(R,i+1,t);                 /*对右区间递归排序*/
    }}
```

划分

```
int Partition (SqList &L, int low, int high) {
    L.r [0] = L.r [low];  pivotkey = L.r [low].key;
    while (low<high) {
        while(low<high&& L.r [high].key>=pivotkey)
                -- high;      // 从右向左搜索
        L.r [low] = L.r [high];
        while (low<high && L.r [low].key<=pivotkey)
                ++ low;      // 从左向右搜索
        L.r [high] = L.r [low];
    }
    L.r [low] = L.r[0];     return low;
}// Partition
```

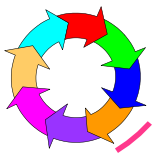
## 2.算法实现



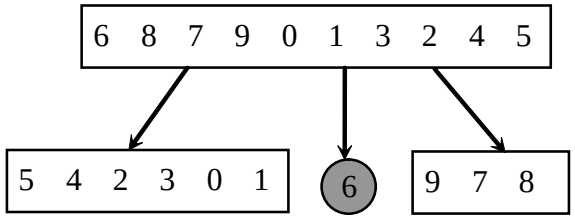
```
void QSort (SqList &L,int low,  int  high ) {
  // 对记录序列L[low..high]进行快速排序
  if (low < high) {// 长度大于1
    pivotloc = Partition(L, low, high);
                    // 对 L[low..high] 进行一次划分
    QSort(L, low, pivotloc-1);
       // 对低子表递归排序，pivotloc是枢轴位置
    QSort(L, pivotloc+1, high); // 对高子表递归排序
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

**第一次调用函数 Qsort 时，待排序记录序列的上、下界分别为 1 和 L.length。**

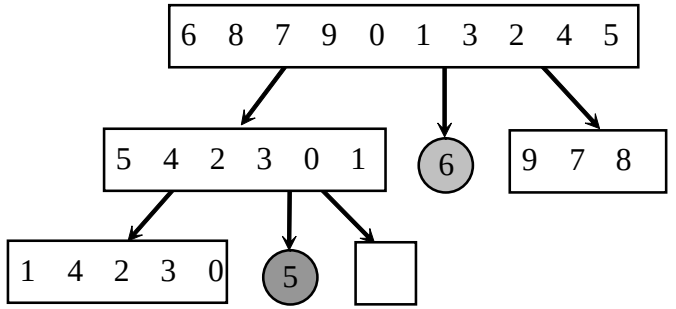
```
void QuickSort( SqList & L) {
   // 对顺序表进行快速排序
     QSort(L.r, 1, L.length);
} // QuickSort
```



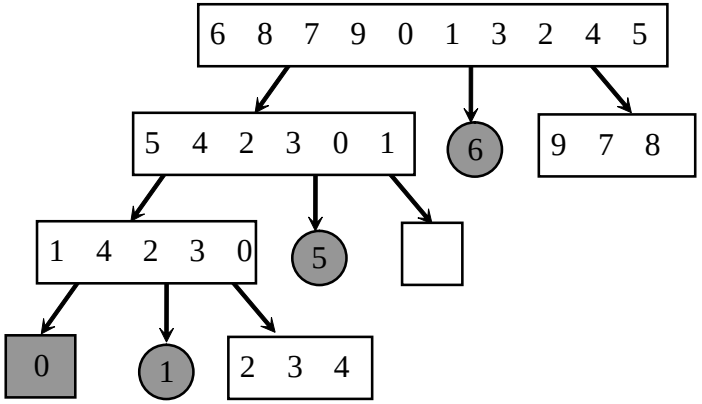
例10.4　设待排序的表有10个记录，其关键字分别为{6, 8, 7, 9, 0, 1, 3, 2, 4, 5}。说明采用快速排序方法进行排序的过程。



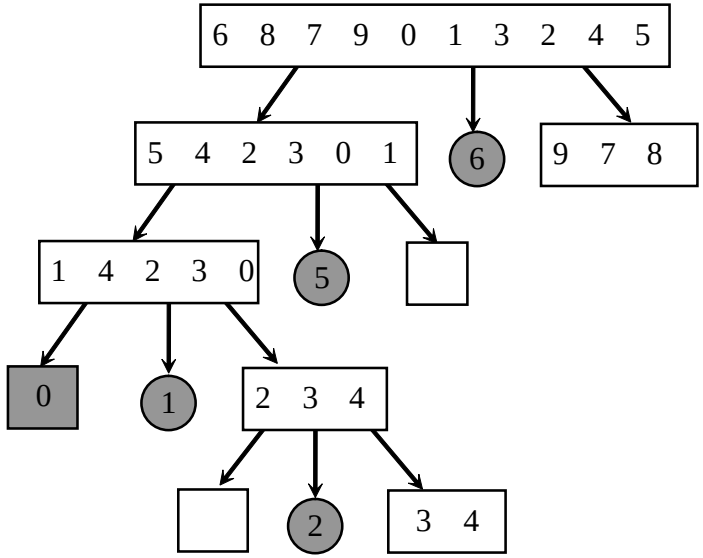

(a) 第 1 次划分



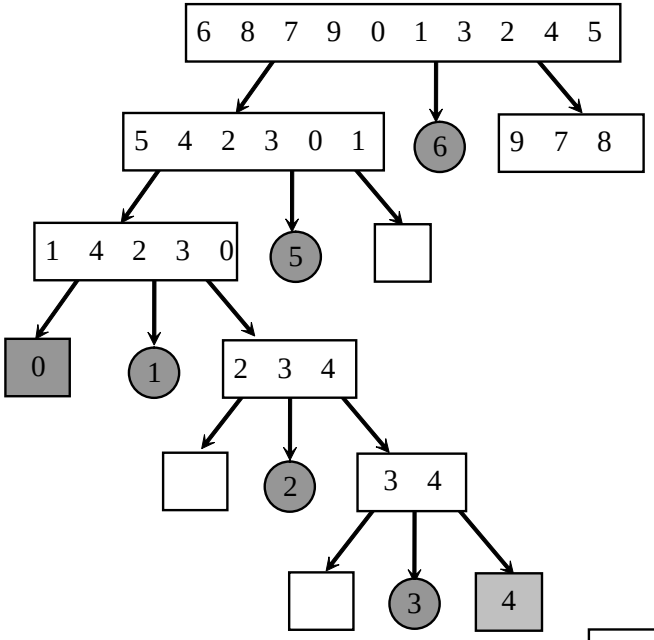

(b) 第 2 次划分



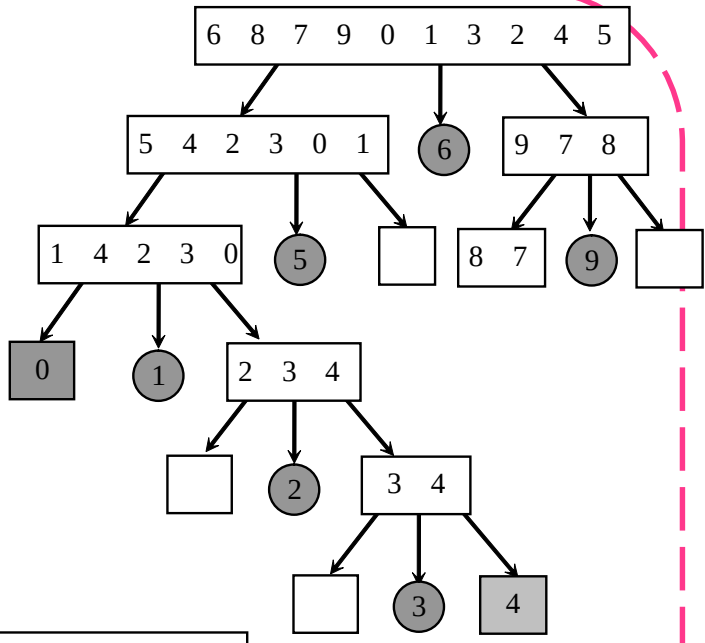

(c) 第 3 次划分



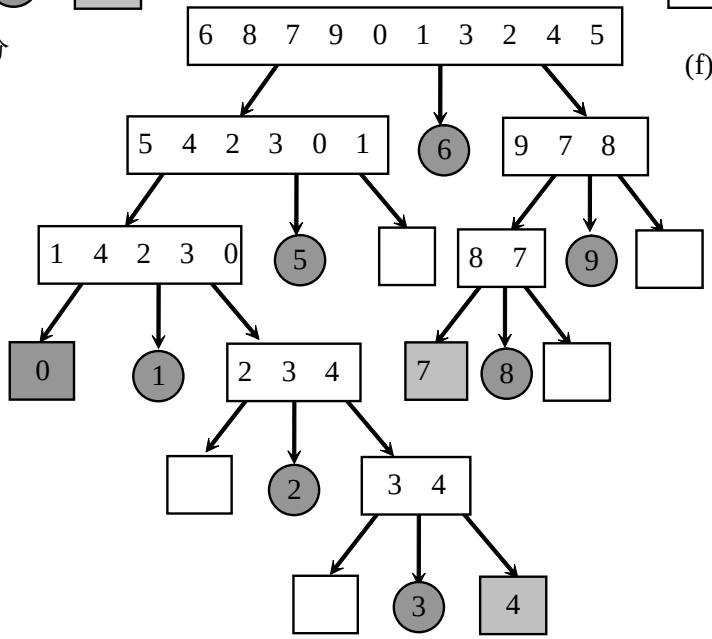

(d) 第 4 次划分




(e) 第 5 次划分




(f) 第 6 次划分

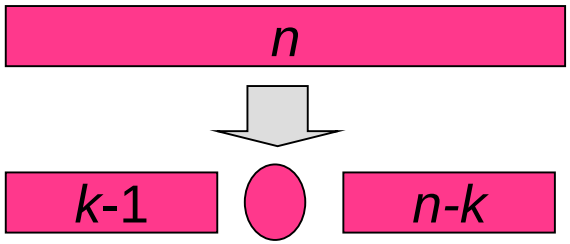



(g) 第 7 次划分

由此可得快速排序所需时间的平均值为：

$$T_{avg}(n) = Cn + \frac{1}{n}\sum_{k=1}^{n}\left[T_{avg}(k-1) + T_{avg}(n-k)\right]$$

1次划分的时间




则可得结果：

$T_{avg}(n)=Cn\log_2 n$。

结论: 快速排序的时间复杂度为$O(n\log_2 n)$

平均所需栈空间为
$O(\log_2 n)$。

# 3. 算法分析

## （1）时间复杂度

假设一次划分所得枢轴位置 $i=k$，则对 $n$ 个记录进行快排所需时间：

$$T(n) = T_{pass}(n) + T(k-1) + T(n-k)$$

其中 $T_{pass}(n)$ 为对 $n$ 个记录进行一次划分所需时间。

①最好情况 $O(n \log_2 n)$

②最坏情况 $O(n^2)$

③平均 $O(n\log n)$

**（2）空间复杂度**

若每次划分较为均匀，则其递归树的高度为O(lgn)，故递归后需栈空间为O(lgn)。最坏情况下，递归树的高度为O(n)，所需的栈空间为O(n)。

**（3）稳定性**

不稳定 。

# 交换排序—快速排序

- 快速排序是目前认为最好的一种内排序方法。
- 快速排序需要用栈来实现递归，所以附加空间较高。

# 交换排序—快速排序

- 【快速排序算法特点】
- (1) 快速排序算法是不稳定的

  对待排序文件 49 49’ 38 65，

  快速排序结果为：38 49’ 49 65

- (2) 快速排序的效率跟初始文件中关键字的排列和选取划分的记录有关。
  - 当初始文件按关键字有序(正序或逆序)时，性能最差，时间复杂度为O($n^2$)

# 交换排序—快速排序

- (3) 常用“三者取中”法来选取划分记录，即取首记录r[s].key.尾记录r[t].key和中间记录r[(s+t)/2].key三者的中间值为划分记录。
- (4)快速排序算法的
  - 平均时间复杂度：为O(nlogn)。
  - 空间复杂度：O(log n).（此为栈的最大深度。）

# 04 *Part Four*

# 选择排序

# 选择排序

- 选择排序的基本思想是：每一趟从待排序的记录中选出关键字最小的记录，顺序放在已排好序的子表的最后，直到全部记录排序完毕。

  两种选择排序方法：

  （1）简单选择排序（或称直接选择排序）

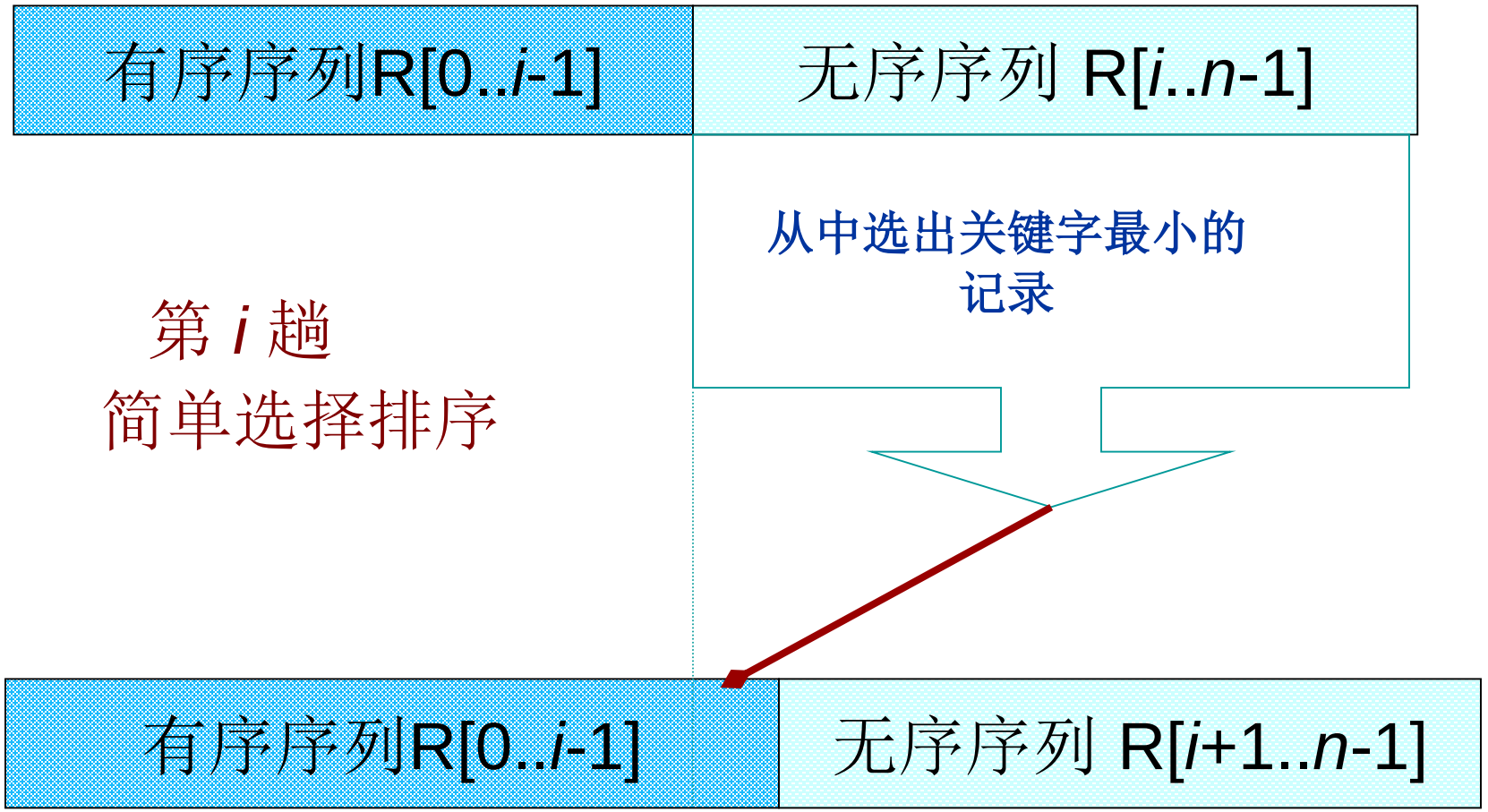
  （2）堆排序


# 简单选择排序

# 选择排序—简单选择排序

- 【简单选择排序】
- 【 基本思想】：每一趟从待排序的记录中选出关键字最小的记录，顺序放在已排好序的子表的最后，直到全部记录排序完毕。

假设排序过程中，待排记录序列的状态为：

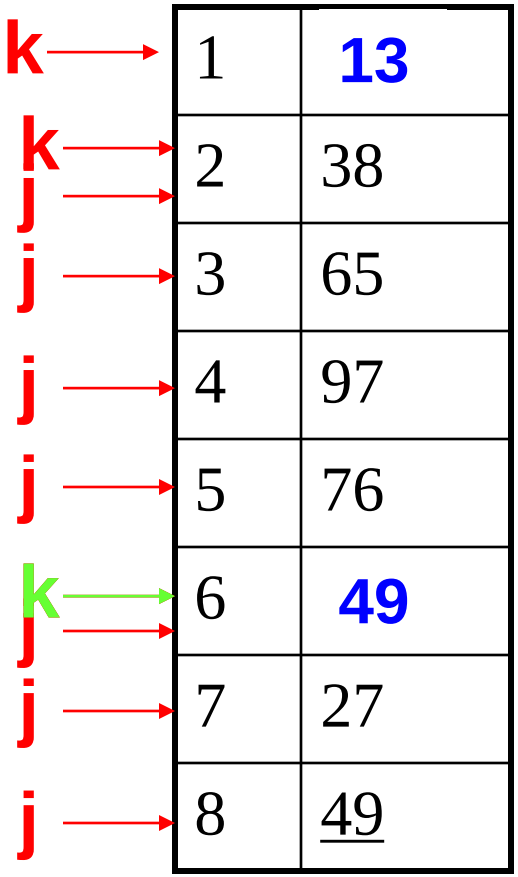



min
4 5 6 3 2 1
交换
min
1 5 6 3 2 4
min
1 2 6 3 5 4
min
1 2 3 6 5 4
1 2 3 4 5 6
1 2 3 4 5 6
1 2 3 4 5 6

# 问题?

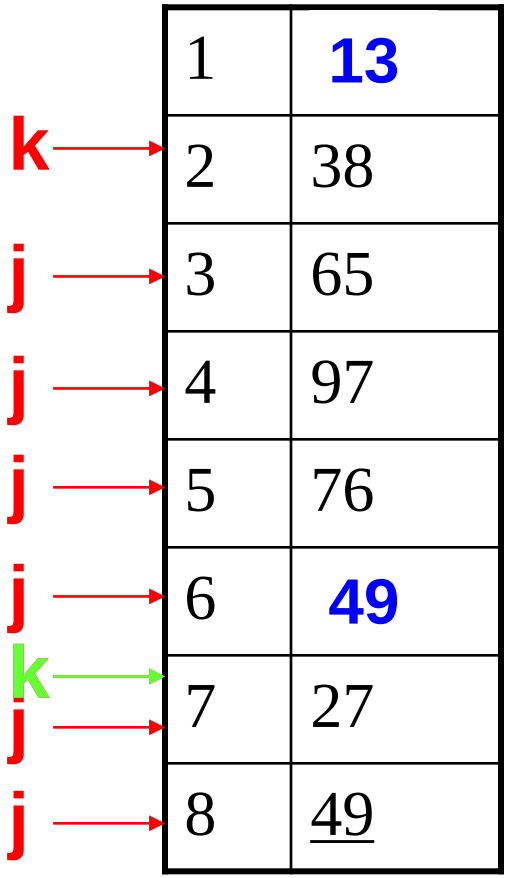
- 第一：选择排序是原地排序算法吗？
  - 选择排序空间复杂度为O(1)，是一种原地排序算法。
- 第二：插入排序是稳定的排序算法吗？
  - 不稳定：每次都要找最小值，并和前面的元素交换位置，这样破坏了稳定性。正是因此，相对于冒泡排序和插入排序，选择排序就稍微逊色了。
- 第三：插入排序的时间复杂度是多少？
  - 最好情况： $O(n^2)$
  - 最坏情况：$O(n^2)$
  - 平均时间复杂度： $O(n^2)$

# 例如：




i=1

# 例如：




i=2

# 选择排序—简单选择排序

- 【例】 设待排序的表有10个记录，其关键字分别为{6, 8, 7, 9, 0, 1, 3, 2, 4, 5}。说明采用直接选择排序方法进行排序的过程。

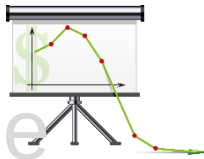
| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 初始关键字 | 6 | 8 | 7 | 9 | 0 | 1 | 3 | 2 | 4 | 5 |
| i=0 | 0 | 8 | 7 | 9 | 6 | 1 | 3 | 2 | 4 | 5 |
| i=1 | 0 | 1 | 7 | 9 | 6 | 8 | 3 | 2 | 4 | 5 |
| i=2 | 0 | 1 | 2 | 9 | 6 | 8 | 3 | 7 | 4 | 5 |
| i=3 | 0 | 1 | 2 | 3 | 6 | 8 | 9 | 7 | 4 | 5 |
| i=4 | 0 | 1 | 2 | 3 | 4 | 8 | 9 | 7 | 6 | 5 |
| i=5 | 0 | 1 | 2 | 3 | 4 | 5 | 9 | 7 | 6 | 8 |
| i=6 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 9 | 8 |
| i=7 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 9 | 8 |
| i=8 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

## 2.算法实现

```
void SelectSort (SqList &L) {
  // 对记录序列R[1.. L.length]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


      for (j=i+1; j<=L.length ; ++j)
          if(L.r[k].key >=L.r[j].key)     k=j;
      // 在 R[i.. L.length] 中选择关键字最小的记录
  }
      if (i!=k) Swap(L.r[i] ,L.r[k] );    //交换
} // SelectSort
```

# 简单选择排序算法




```
void SelectSort(RecType R[],int n)
{    int i,j,k;  RecType temp;
     for (i=0;i<n-1;i++)                          /*做第i趟排序*/
     {     k=i;
         for (j=i+1;j<n;j++)    /*在[i..n-1]中选key最小的R[k] */
              if (R[j].key<R[k].key)
                        k=j;          /*k记下的最小关键字所在的位置*/
          if (k!=i)                                  /*交换R[i]和R[k] */
          {    temp=R[i];  R[i]=R[k];  R[k]=temp;  }
       }
   }
```

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 15 | 8 | 6 | 12 | 16 | 5 | 4 |

# 选择排序—简单选择排序

- 【算法分析】

  (1) 交换次数：正序时交换次数最少，为0次，
  逆序时最多，为n-1次。

  (2) 比较次数：与初始文件关键字排列无关，
  为n(n-1)/2次。

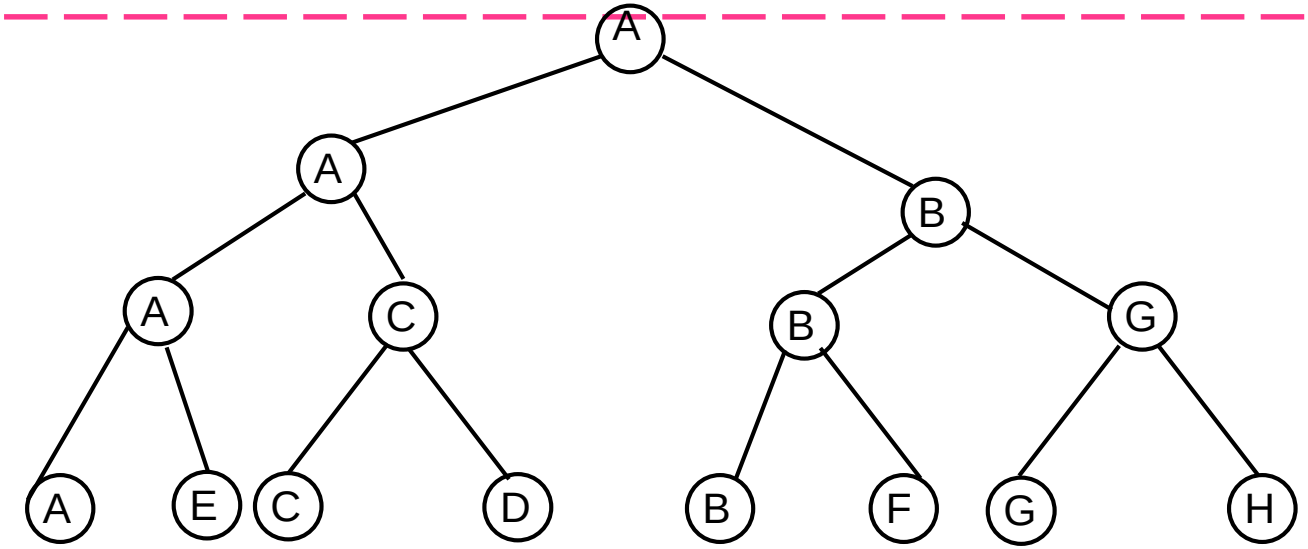
  (3) 简单选择排序时间复杂度为$O(n^2)$，
  并且是不稳定的排序。

# 选择排序—树形选择排序

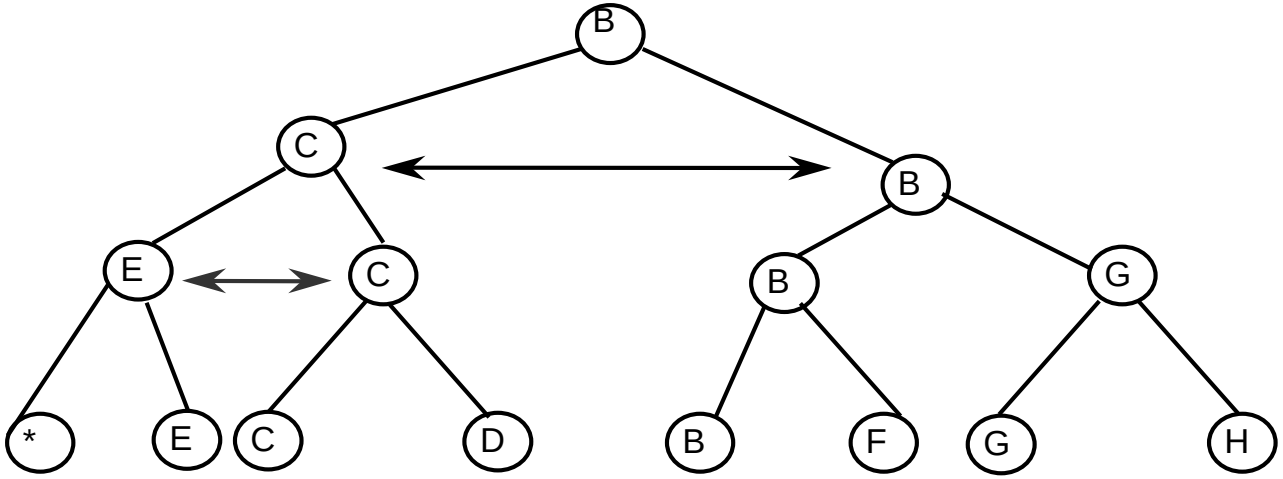
- 从直接选择排序可知，在n个排序码中，找出最小值需n-1次比较，找出第二小值需n-2次比较，找出第三小值需n-3次比较，其余依此类推。所以，总的比较次数为：(n-1)+(n-2)+…+3+2+1= $(n^2-n)/2$，
- 那么，能否对直接选择排序算法加以改进，使总的比较次数比 $(n^2-n)/2$小呢？
  - 显然，在n个排序码中选出最小值，至少进行n-1次比较，但是，继续在剩下的n-1个关键字中选第二小的值，就并非一定要进行n-2次比较，若能利用前面n-1次比较所得信息，则可以减少以后各趟选择排序中所

# 选择排序—树形选择排序

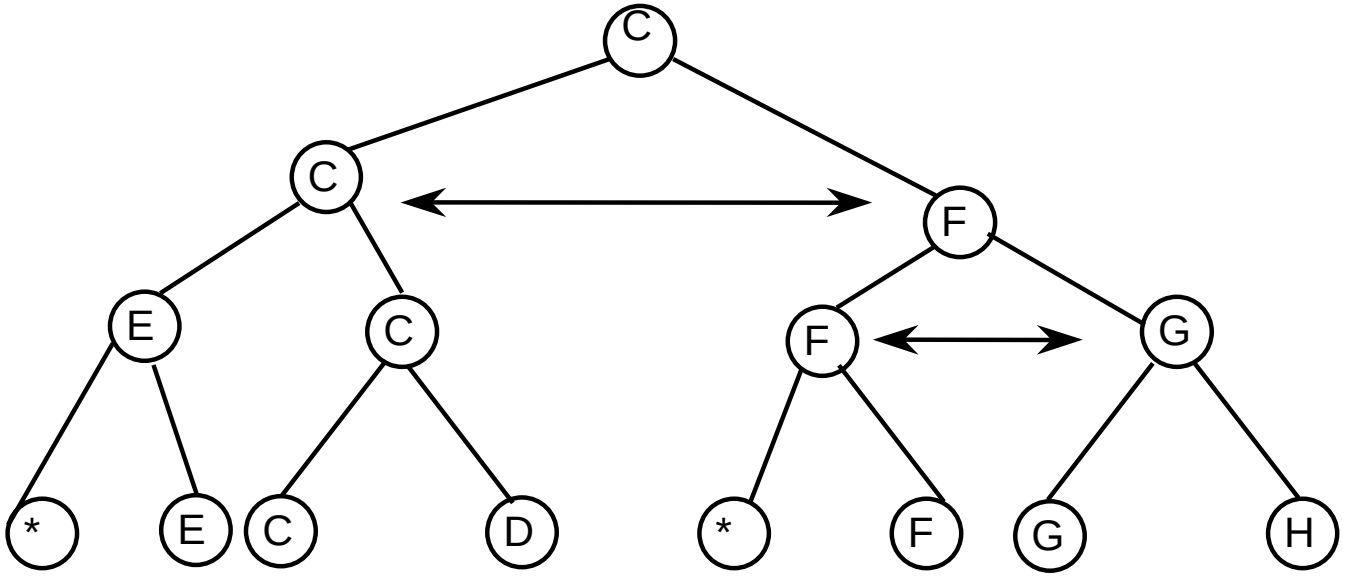
- 用的比较次数，比如8个运动员中决出前三名，不需要7+6+5=18场比赛（前提是，若甲胜乙，而乙胜丙，则认为甲胜丙），最多需要11场比赛即可（通过7场比赛产生冠军后，第二名只能在输给冠军的三个对手中产生，需 2场比赛，而第三名只能在输给亚军的三个对中产生，也需2场比赛，总共11场比赛）。具体见图所示。




(a) 8个队决出冠军的情形（共7 场比赛）




(b) 决出亚军的情形（共　　2 场比赛，少于　 6　场）




(c) 决出第三名的情形（共2 场比赛，少于6 场）

图　　决出比赛前三名的过程示意图

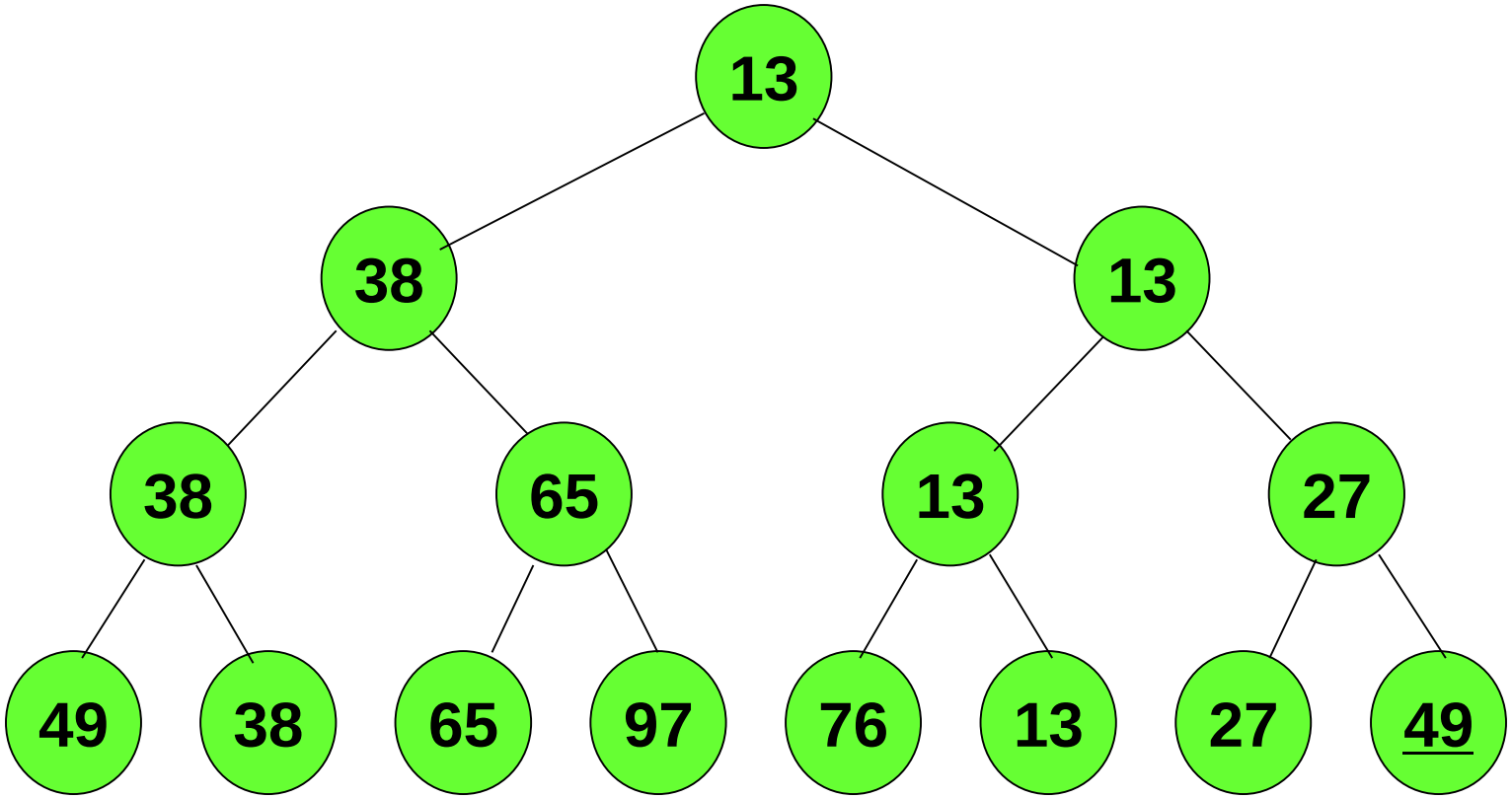
# 选择排序—树形选择排序

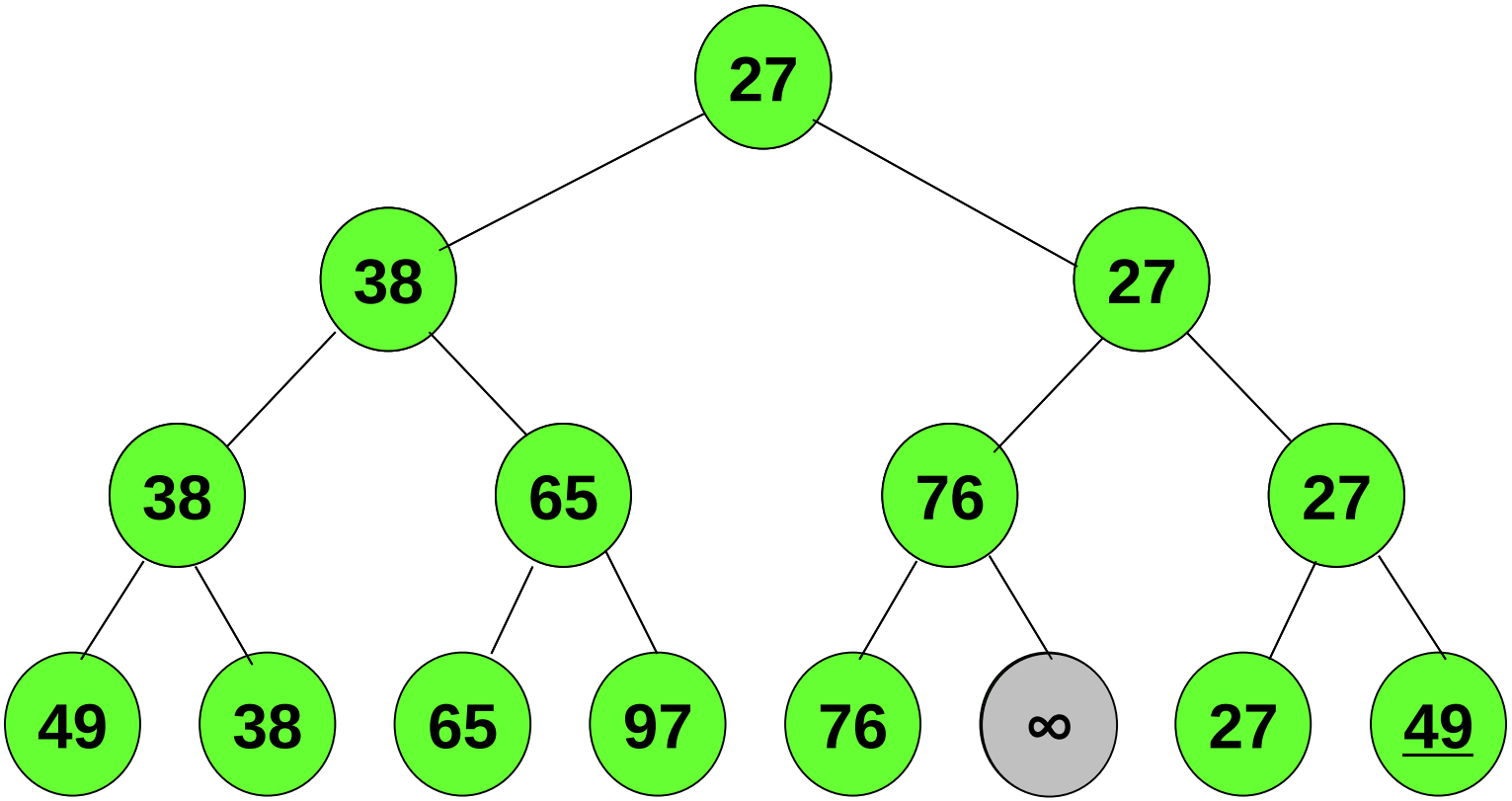
- 从图（a）可知，8个队经过4场比赛，获胜的4个队进入半决赛，再经过2场半决赛和1场决赛即可知道冠军属谁（共7场比赛）按照锦标赛的传递关糸，亚军只能产生于分别在决赛，半决赛和第一轮比赛中输给冠军的选取手中，于是亚军只能在b、c、e这3个队中产生（见图（b））， 即进行2场比赛（e 与c一场，e与c的胜队与b一场）后，即可知道亚军属谁。同理，第三名只需在c、f、g这3个队产生（见图（c））即进2场比赛（f与g一场，f与g的胜队与c一场）即可知道第三名属谁。

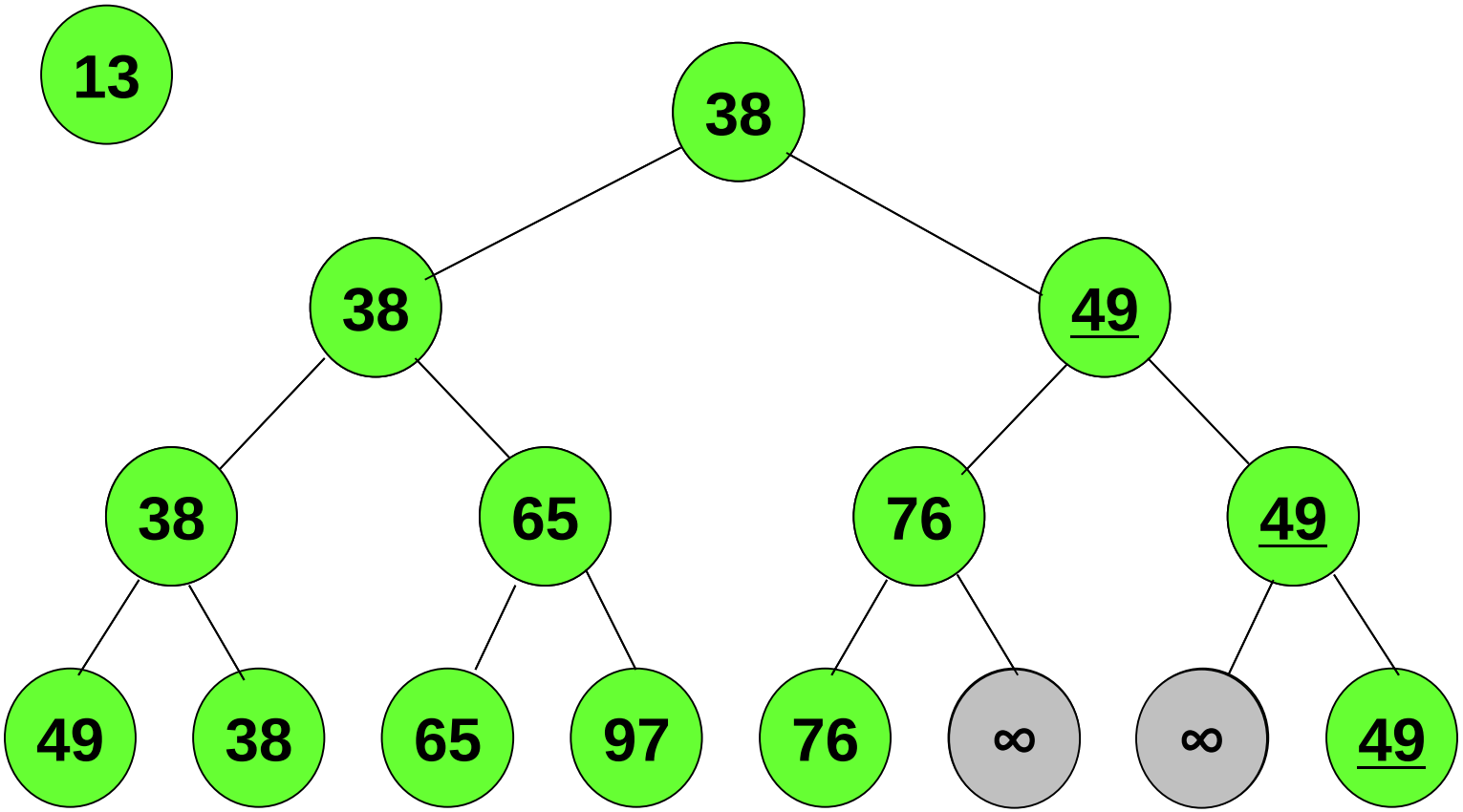
# 选择排序—树形选择排序

- 【树形选择排序】（tree selection sorting），又称锦标赛排序（tournament sorting），是一种按照锦标赛的思想进行选择排序的方法。
- 【算法思想】首先对n个记录的排序码进行两两比较，然后在其中n/2个较小者之间再进行两两比较，如此重复，直到选出最小排序码为止。

【例】：给定排序码头 50，37，66，98，75，12，26，49，树形选择排序过程见图。


13
38
13
38
65
13
27
49
38
65
97
76
13
27
49


27
38
27
38
65
76
27
49
38
65
97
76
∞
27
49


13
38
38
49
38
65
76
49
49
38
65
97
76
∞
∞
49

# 选择排序—树形选择排序

- 【树形选择排序的算法分析】

（1）含有n个叶子节点的完全二叉树的深度为$\log_2 n+1$，则选择排序的每一趟都需作$\log_2 n$次比较，排序的时间复杂度$O(n\log_2 n)$。

（2）需要辅助存储空间较多（n-1），和最大值作多余的比较等等。


# 堆排序

# 问题

- 一个包含10亿个搜索关键词的日志文件，如何能快速获取到热门榜Top 10的搜索关键词呢？

- 有一个访问量非常大的新闻网站，希望将点击量排名 Top 10 的新闻摘要，滚动显示在网站首页 banner 上，并且每隔 1 小时更新一次。你会如何来实现呢？

# 选择排序—堆排序

- 【堆排序】
- 1. 堆的定义：

若有n个元素的排序码$k_1$，$k_2$，$k_3$，…，$k_n$，当满足如下条件：

（1）$\begin{cases} k_i \le k_{2i} \\ k_i \le k_{2i+1} \end{cases}$ 或 (2) $\begin{cases} k_i \ge k_{2i} \\ k_i \ge k_{2i+1} \end{cases}$ 其中$i=1,2,\ldots,\lfloor n/2 \rfloor$

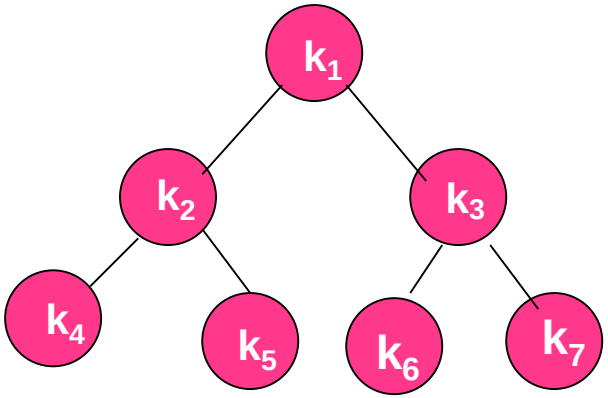
个元素的排序码$k_1$，$k_2$，$k_3$，…，$k_n$为一个堆。

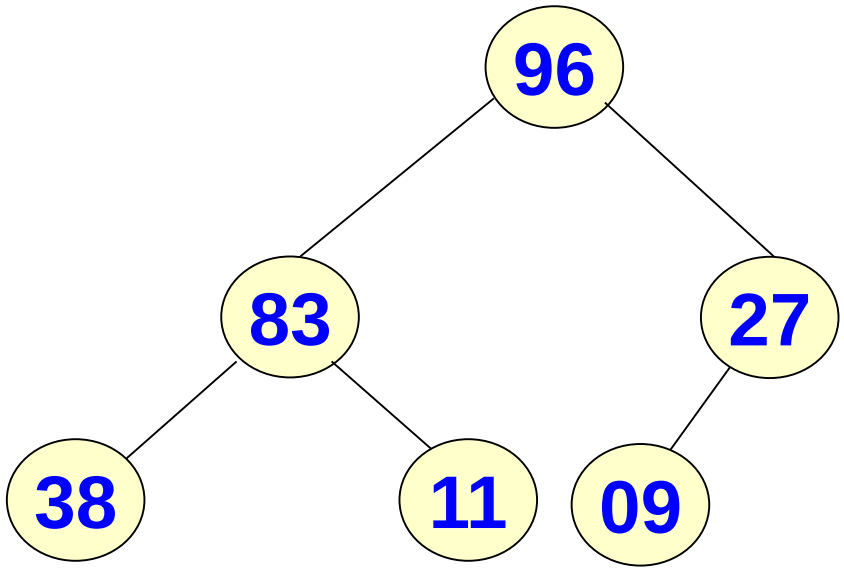
满足条件（1）是按由小到大排序

（2）是按由大到小排序

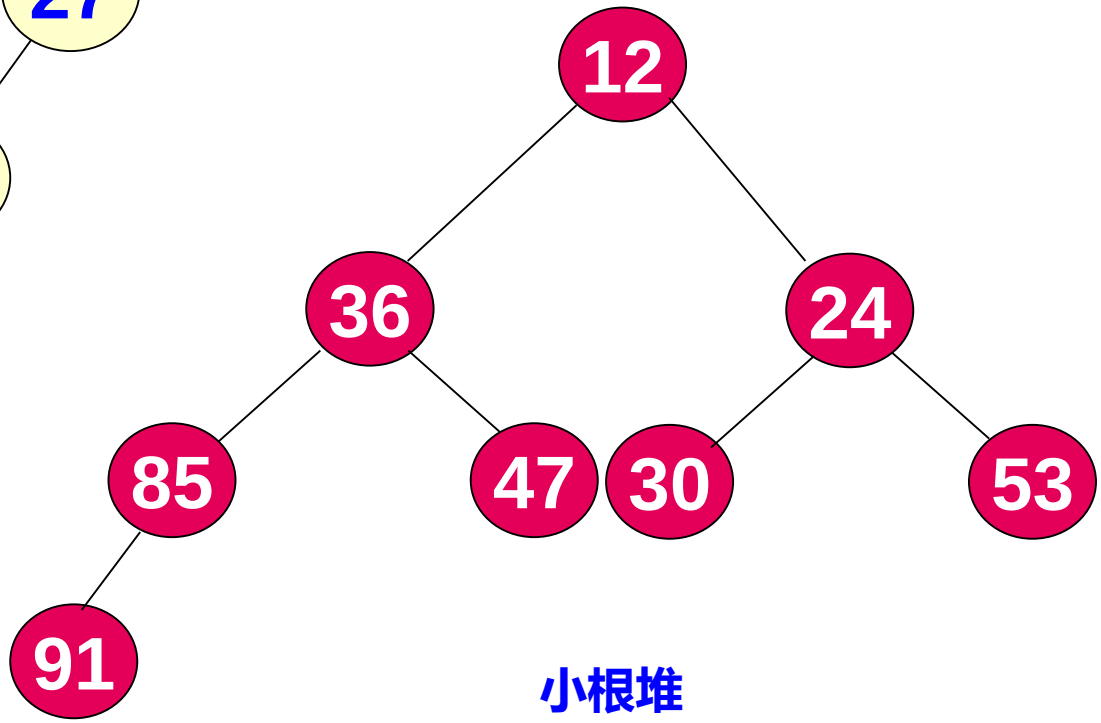
# 选择排序—堆排序

- 若将堆视为一个完全二叉树（图一），则堆的含义为：完全二叉树中所有非终端结点的值均不大于(或不小于)其左、右孩子的值（图二）。
- 堆顶元素(即完全二叉树的根)是序列中最小(或最大)的元素。






大根堆




小根堆

- 例如:

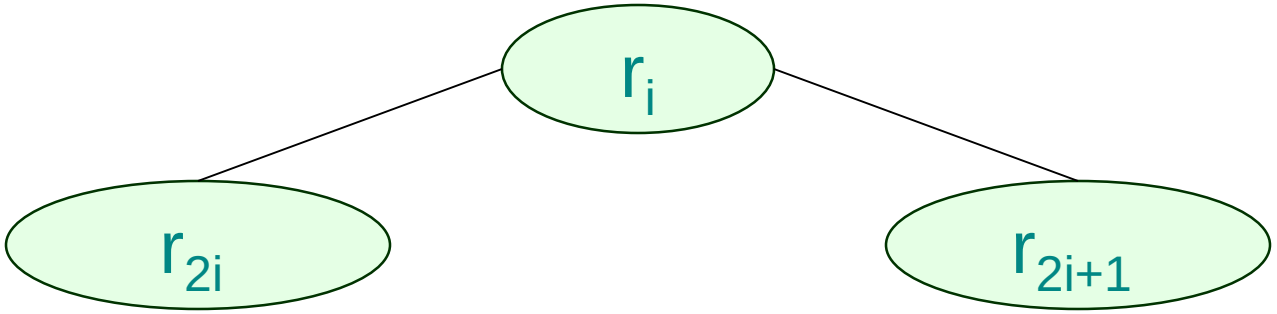
{12, 36, 27, 65, 40, 34, 98, 81, 73, 55, 49}

是小根堆

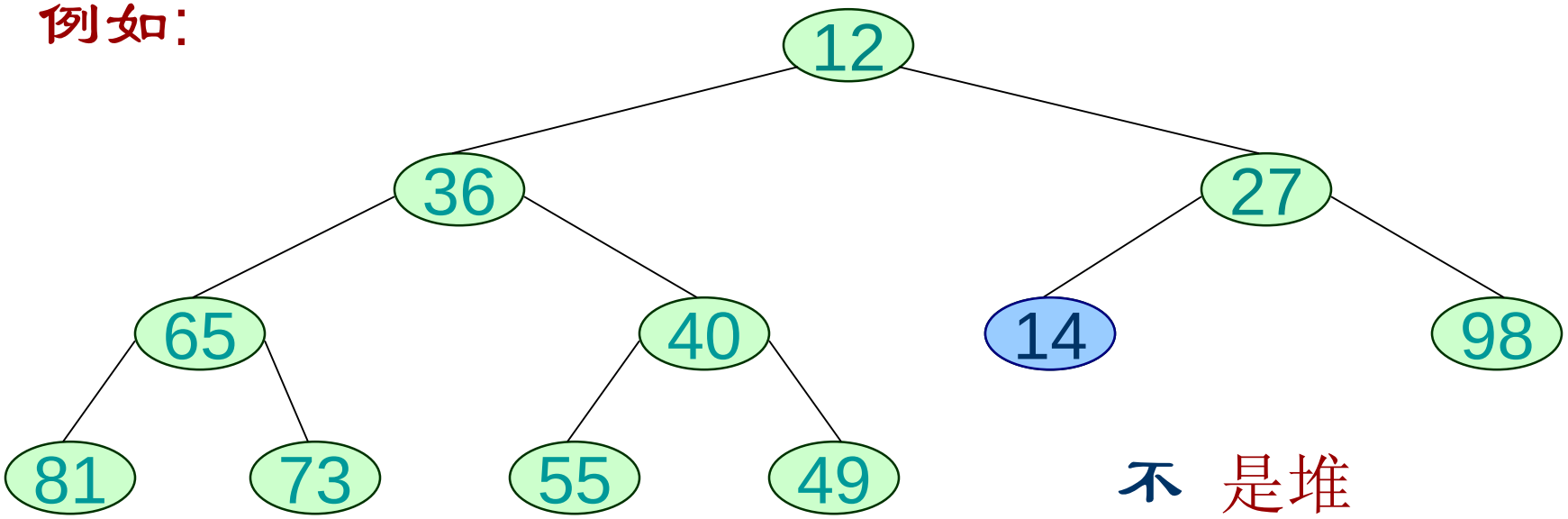
{12, 36, 27, 65, 40, 14, 98, 81, 73, 55, 49}

不是堆

若将该数列视作完全二叉树，则 $r_{2i}$ 是 $r_i$ 的左孩子； $r_{2i+1}$ 是 $r_i$ 的右孩子。



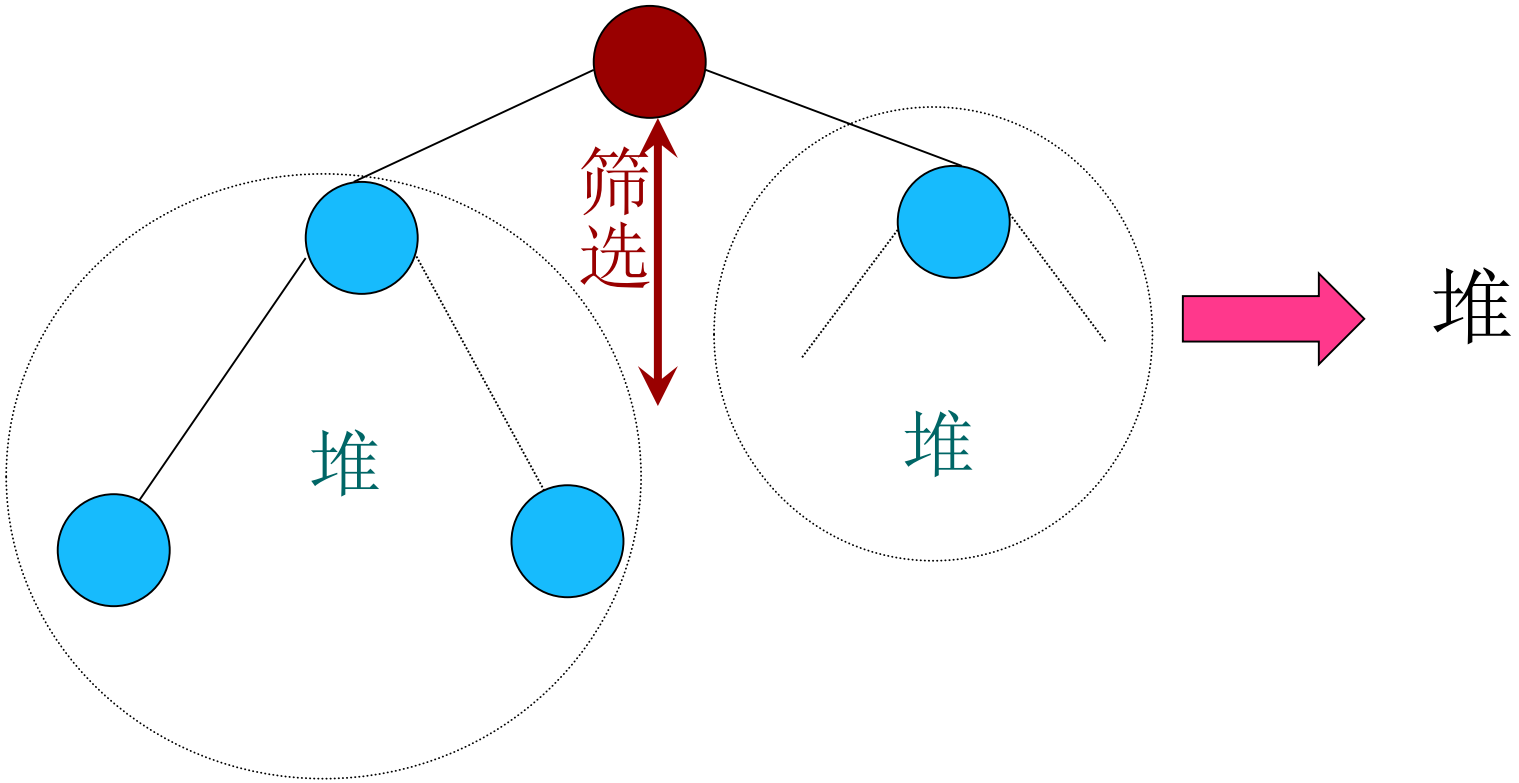

例如：




不 是堆

堆排序的关键是构造堆,这里采用筛选算法建堆。

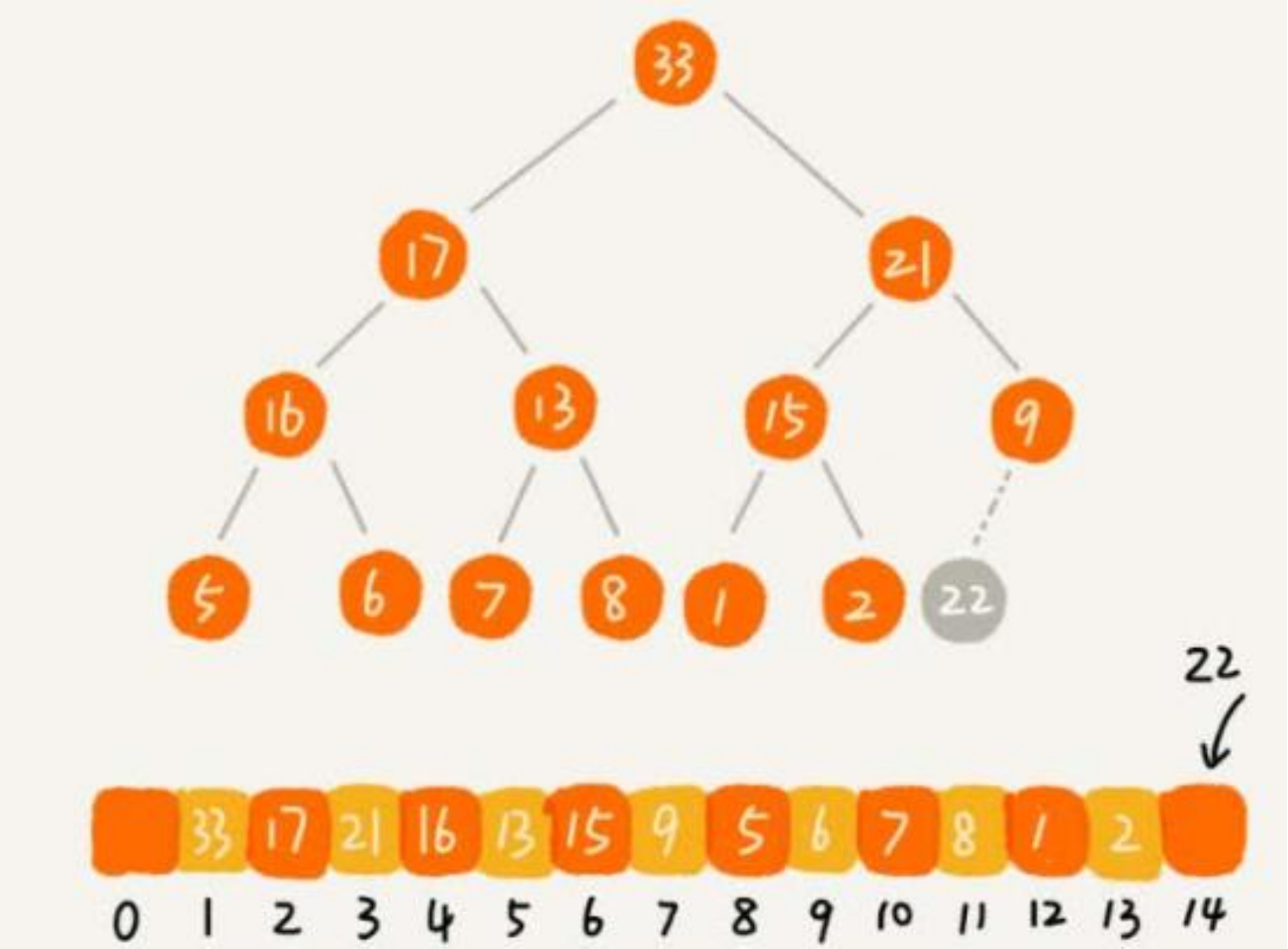
所谓“筛选”指的是，对一棵左/右子树均为堆的完全二叉树，“调整”根结点使整个二叉树也成为一个堆。

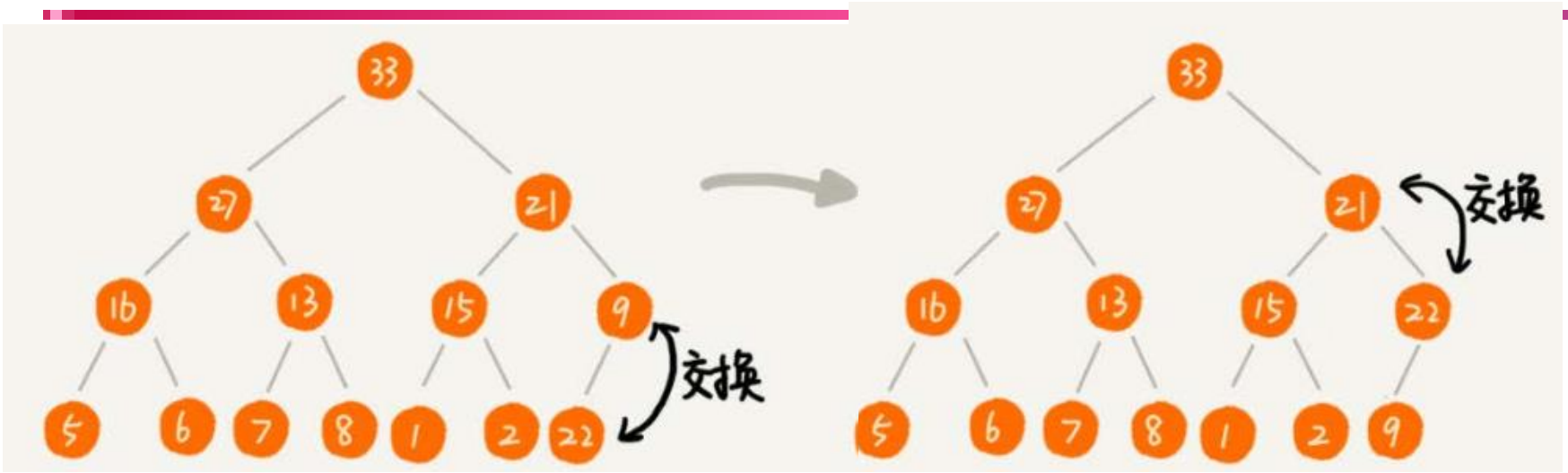


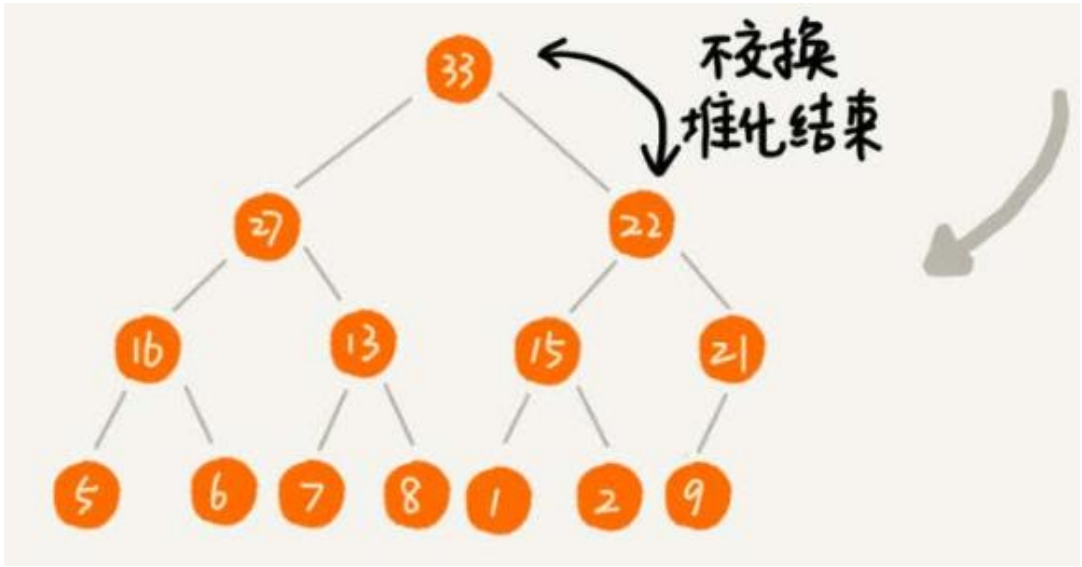
# 在堆中插入元素

- 堆中插入一个元素后，需要继续满足堆的两个特性
  - 堆必须是一个完全二叉树；
  - 堆中的每个节点的值必须大于等于（或者小于等于）其子树中每个节点的值。
- 插入操作的实现：

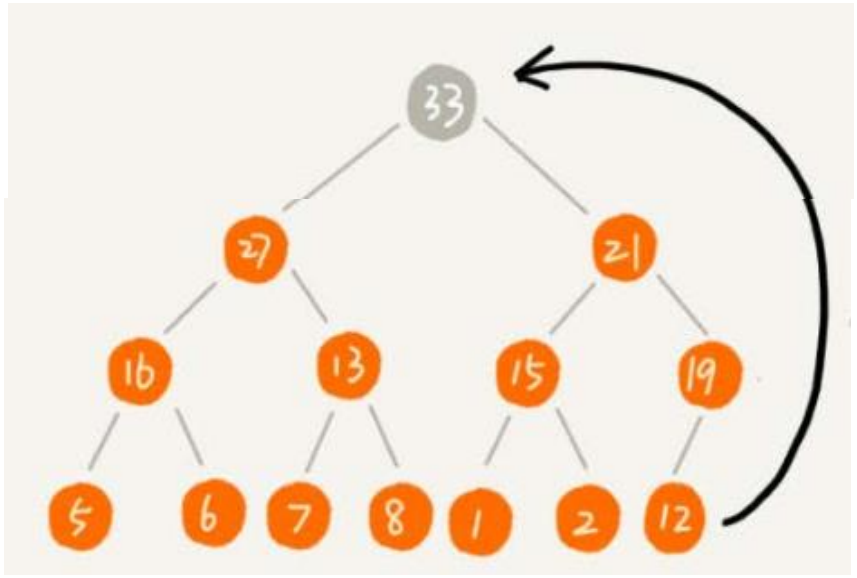
# 在堆中插入元素

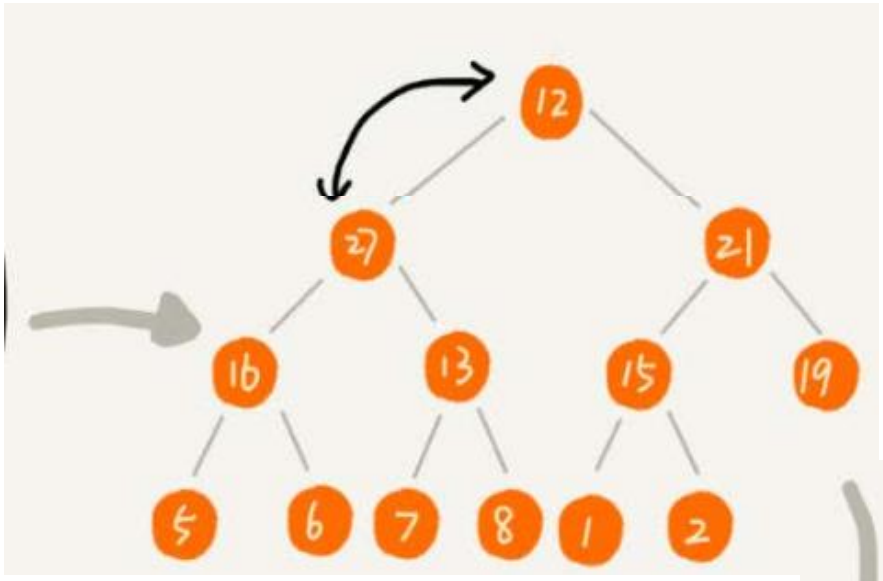

33
17
21
16
13
15
9
5
6
7
8
1
2
22
22
33 17 21 16 13 15 9 5 6 7 8 1 2
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

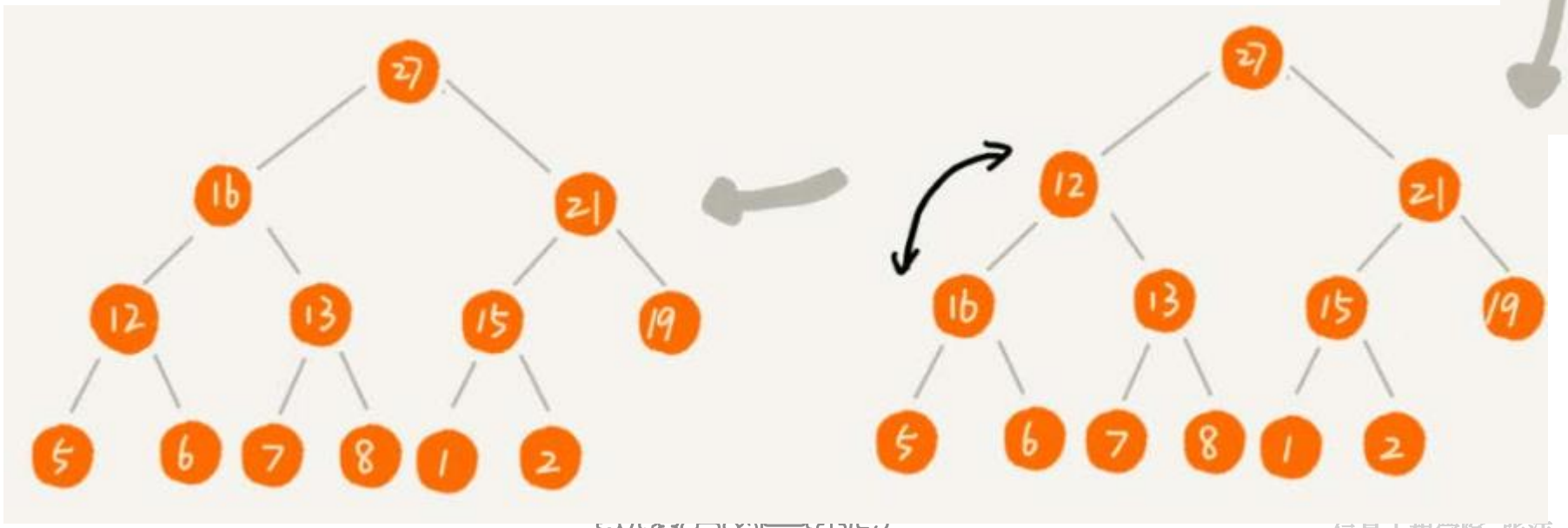

33
27
21
16
13
15
9
5
6
7
8
1
2
22
交换
33
27
21
交换
16
13
15
22
5
6
7
8
1
2
9


33
不交换
堆化结束
27
22
16
13
15
21
5
6
7
8
1
2
9

# 在堆中删除元素


33
27
21
16
13
15
19
5
6
7
8
1
2
12


12
27
21
16
13
15
19
5
6
7
8
1
2


27
16
21
12
13
15
19
5
6
7
8
1
2
27
12
21
16
13
15
19
5
6
7
8
1
2

# 选择排序—堆排序

- 【堆排序基本思想】

  (1) 以初始关键字序列，建立堆；

  (2) 输出堆顶最小元素；

  (3) 调整余下的元素，使其成为一个新堆；

  (4) 重复(2), (3) n 次，得到 一个有序序列。

- 关键要解决(1)和(3)，即如何由一个无序序列建成一个堆?
- 如何调整余下的元素成为一个新堆?

# 选择排序—堆排序

- 【调整方法】

  (1) 将堆顶元素和堆的最后一个元素位置交换；

  (2) 然后以当前堆顶元素和其左、右子树的根结点进行比较(此时，左、右子树均为堆)，并与值较小的结点进行交换；

  (3) 重复(2)，继续调整被交换过的子树，直到叶结点或没进行交换为止。

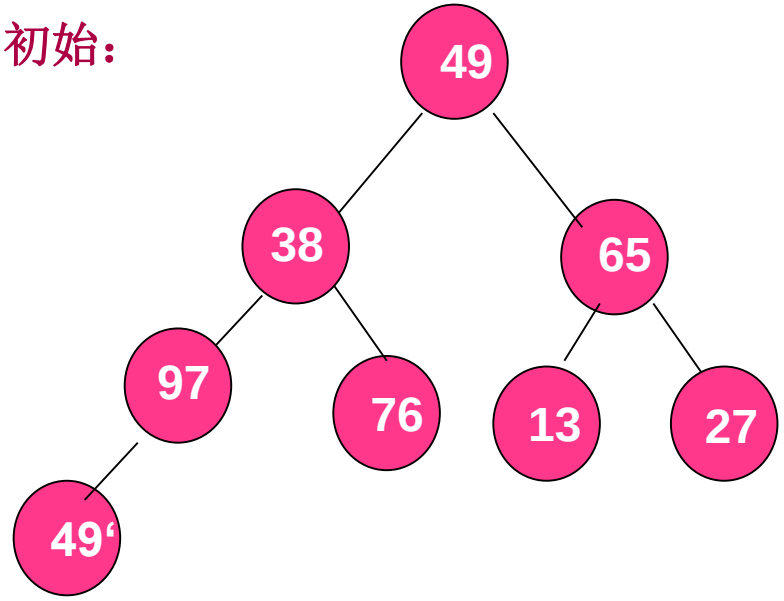
  称这个调整过程为"筛选"。

# 选择排序—堆排序

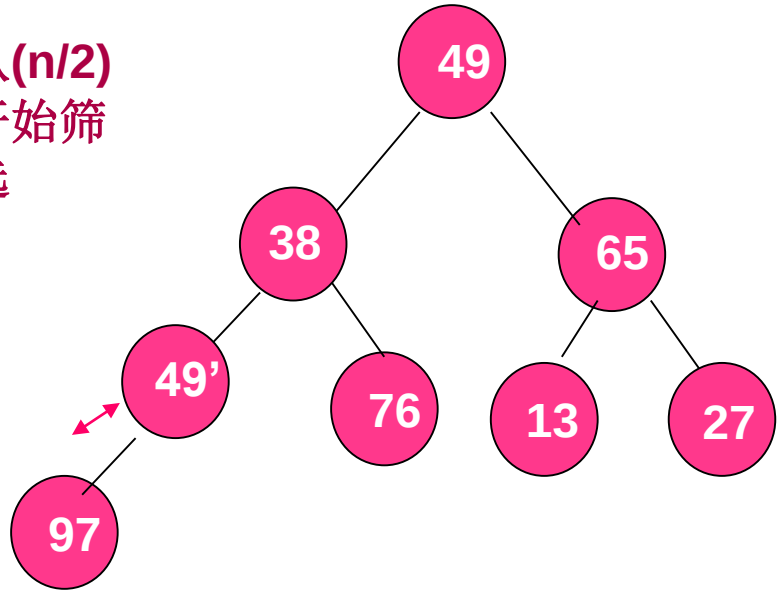
- 【无序序列建堆过程】
- 【方法】：从完全二叉树的最后一个非终端结点 $\lfloor n/2 \rfloor$ 开始，反复调用筛选过程，直到第一个结点，则得到一个堆。

待排序序列： (49,38,65,97,76,13,27,49‘)

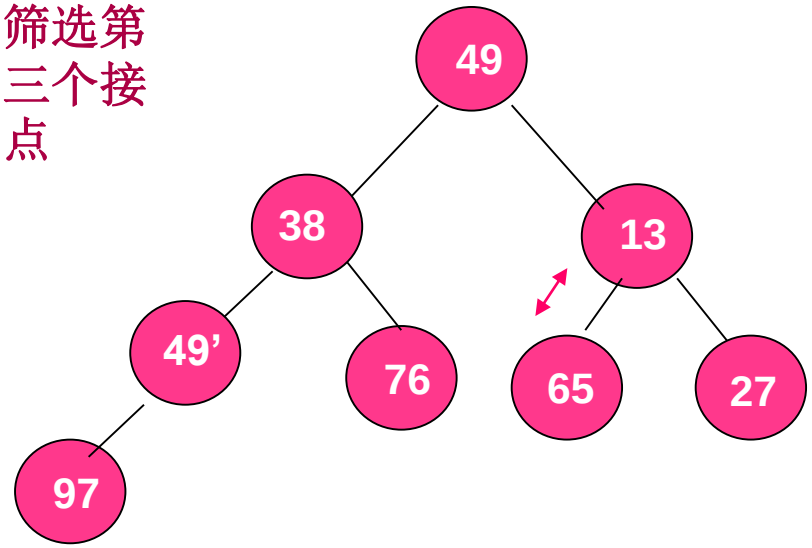
初始：



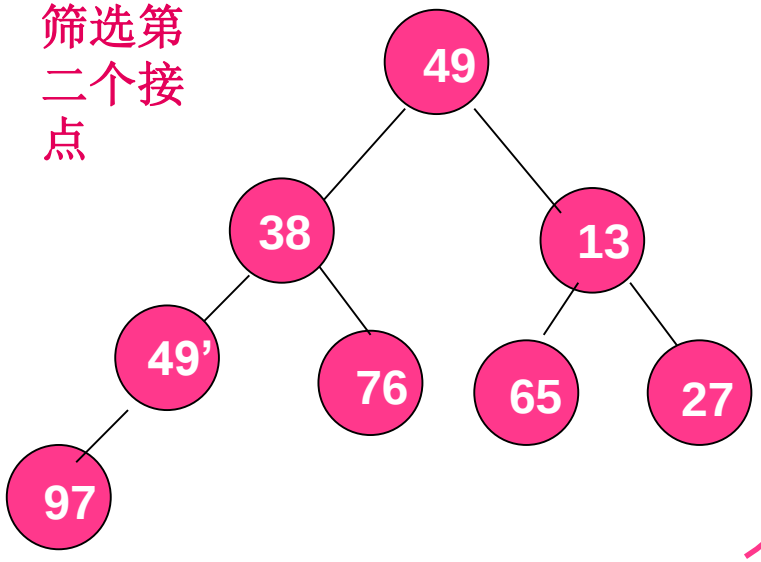

从(n/2)开始筛选




筛选第三个接点

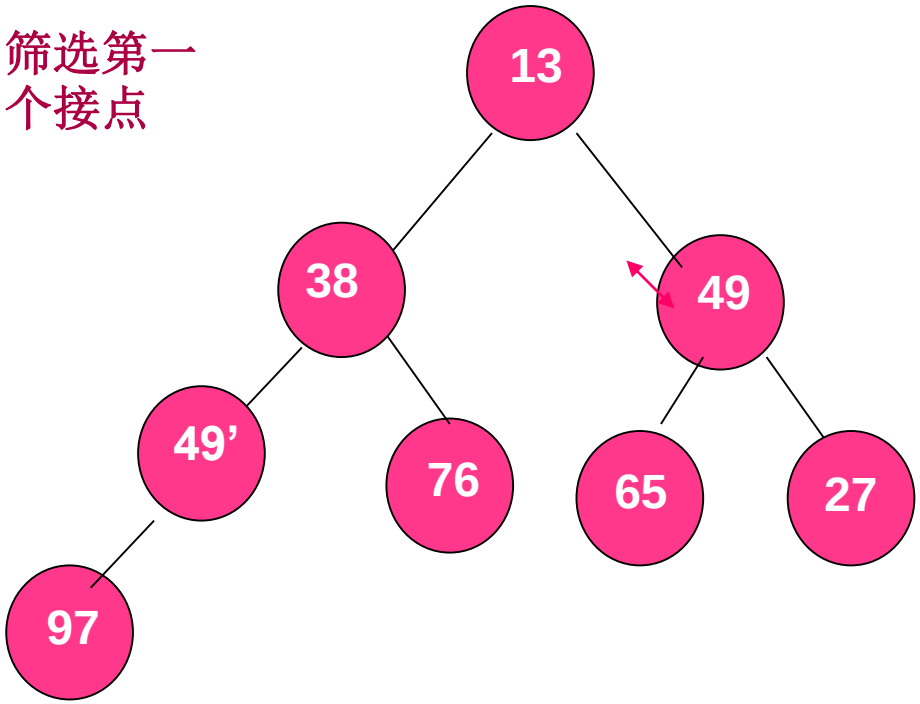



筛选第二个接点



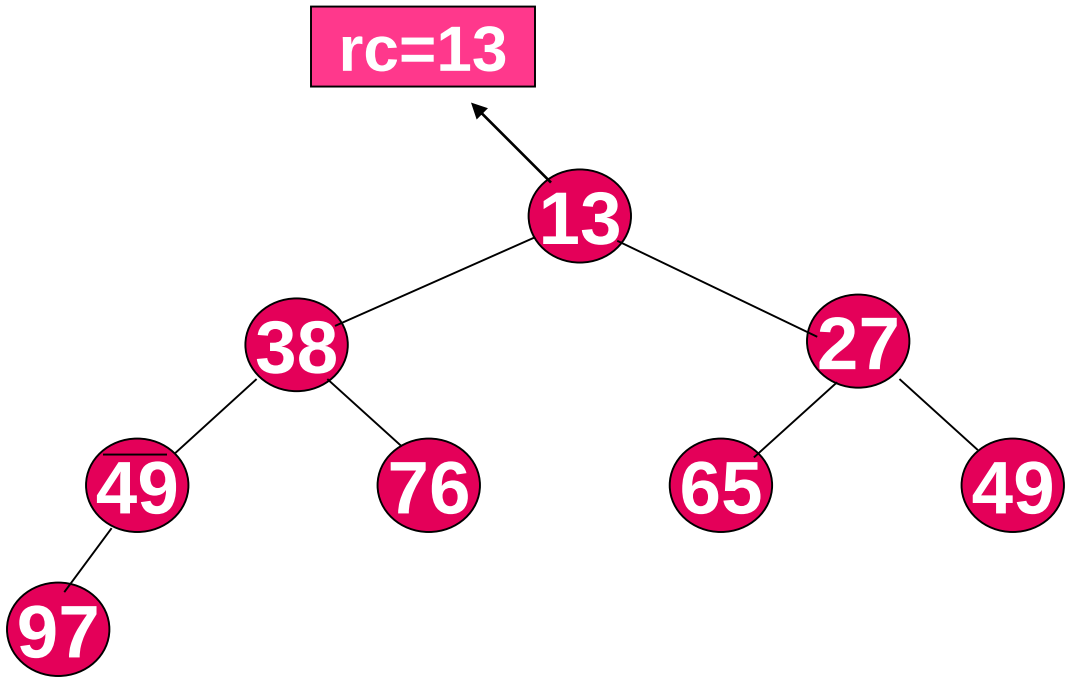
# 选择排序—堆排序

筛选第一个接点



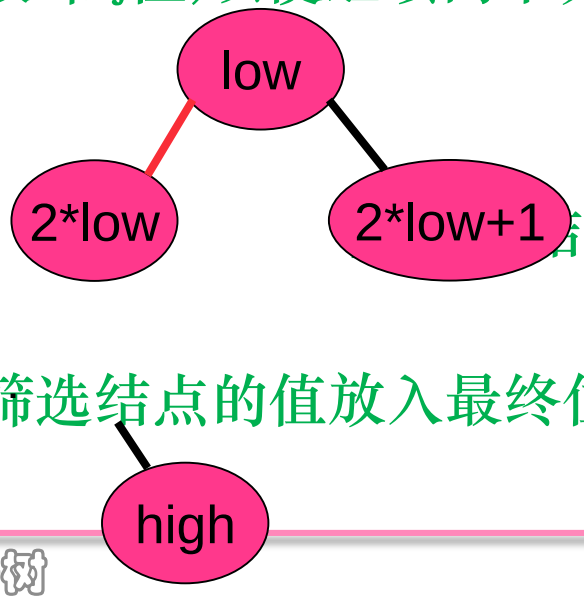

第一趟筛选结束，构成堆，堆顶元素是所有数据中最小的

# 例如：堆调整过程

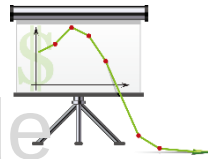



# 选择排序—堆排序

```
void sift(RecType R[],int low,int high)      /*调整堆的算法*/
{   int i=low,j=2*i;                         /*R[j]是R[i]的左孩子*/
    RecType temp=R[i];
    while (j<=high)
    {   if  (j<high && R[j].key<R[j+1].key)  j++;
        if   (temp.key<R[j].key)
        {   R[i]=R[j];                       /*将R[j]调整到双亲结点位置上*/
            i=j;                             /*修改i和j值,以便继续向下筛选*/
            j=2*i;
        }
        else break;                          /*筛选已结束*/
    }
    R[i]=temp;                               /*被筛选结点的值放入最终位置*/
}
```



# 选择排序—堆排序




有了调整堆的函数，利用该函数，将已有堆中的根与最后一个叶子交换，进一步恢复堆，直到一棵树只剩一个根为止。实现堆排序的算法如下：

```
void HeapSort(RecType R[],int n)
{    int i;        RecType temp;
     for (i=n/2;i>=1;i--)        /*循环建立初始堆*/
          sift(R,i,n);
     for (i=n;i>=2;i--)   /*进行n-1次循环,完成推排序*/
     {    temp=R[1];  /*将第一个元素同当前区间内R[1]对换*/
          R[1]=R[i];R[i]=temp;
          sift(R,1,i-1);      /*筛选R[1]结点,得到i-1个结点的堆*/
     }
}
```

# 选择排序—堆排序

- 【总结】：

从堆排序的算法中看出，除第一次建堆所化的时间较长，从第二次开始，每次只有堆顶元素不满足堆要求，因此，后面的建堆仅需调整很少元素即可，所以，当排序个数较大时，才能体现出堆排序的优越性。

- 【结论】

(1) 堆排序适应于n值较大的排序序列。

(2) 相对与快速排序，堆排序仅需一个附加空间 用与排序时的交换空间。

# 选择排序—堆排序

- 【堆排序的时间复杂度分析】：
  - 1. **对深度为 k 的堆，“筛选”所需进行的关键字比较的次数至多为**$2(k-1)$；
  - **2. 对 n 个关键字，建成深度为h(=**$\lfloor \log_2 n \rfloor +1$**)的堆，所需进行的关键字比较的次数至多 4n；**
  - 3. **调整“堆顶” n-1 次，总共进行的关键字比较的次数不超过** $2(\lfloor \log_2(n-1) \rfloor + \lfloor \log_2(n-2) \rfloor + \cdots + \log_2 2) < 2n(\lfloor \log_2 n \rfloor)$
- 因此，堆排序的时间复杂度为$O(n\log n)$。

# 总结

- 堆的三个操作
- 插入一个数据：新插入的数据放到数组的最后，然后从下往上堆化；
- 删除堆顶元素：删除堆顶数据的时候，把数组中的最后一个元素放到堆顶，然后从上往下堆化。
- 堆排序：堆排序包含两个过程，建堆和排序

# 讨论?

- 在实际开发中，为什么快速排序要比堆排序性能好？
- 第一点，堆排序数据访问的方式没有快速排序友好。
- 第二点，对于同样的数据，在排序过程中，堆排序算法的数据交换次数要多于快速排序

# 堆的应用一：优先级队列

- 优先级队列：队列，数据的出队顺序不是先进先出，而是按照优先级来，优先级最高的，最先出队。

- 从优先级队列中取出优先级最高的元素，就相当于取出堆顶元素

# 1. 合并有序小文件

- 假设有 100 个小文件，每个文件的大小是 100MB ，每个文件中存储的都是有序的数。希望将这些 100 个小文件合并成一个有序的大文件。

# 1. 合并有序小文件

- 【思路1】：
  - 从这 100 个文件中，各取第一个数，放入数组中，然后比较大小，把最小的那个数放入合并后的大文件中，并从数组中删除。
  - 假设，这个最小的字符串来自于 13.txt 这个小文件，就再从这个小文件取下一个字符串，并且放到数组中，重新比较大小，并且选择最小的放入合并后的大文件，并且将它从数组中删除。依次类推，直到所有的文件中的数据都放入到大文件为止。

# 1. 合并有序小文件

- 【缺点】：
  - 这里用数组这种数据结构，来存储从小文件中取出来的字符串。每次从数组中取最小字符串，都需要循环遍历整个数组，显然，这不是很高效。有没有更加高效方法呢？

# 1. 合并有序小文件

- 【思路2】：
  - 用到优先级队列，也可以说是堆。将从每个小文件中取出来的数放入到小顶堆中，那堆顶的元素，也就是优先级队列队首的元素，就是最小的。将这个数放入到大文件中，并将其从堆中删除。然后再从小文件中取出下一个数，放入到堆中。循环这个过程，就可以将 100 个小文件中的数据依次放入到大文件中。
- 【优】：
  - 删除堆顶数据和往堆中插入数据的时间复杂度都是O(logn)，n表示堆中的数据个数。比原来数组存储的方式高效了很多

# 2. 高性能定时器

- 设有一个定时器，定时器中维护了很多定时任务，每个任务都设定了一个要触发执行的时间点。定时器每过一个很小的单位时间（比如 1 秒），就扫描一遍
- 任务，看是否有任务到达设定的执行时间。如果到达了，就拿出来执行。

| | |
|---|---|
| 2018.11.28. 17:30 | Task A |
| 2018.11.28. 19:20 | Task B |
| 2018.11.28. 15:31 | Task C |
| 2018.11.28. 13:55 | Task D |

# 2. 高性能定时器

- 【缺】：
  - 但是，这样每过 1 秒就扫描一遍任务列表的做法比较低效，主要原因有两点：第一，任务的约定执行时间离当前时间可能还有很久，这样前面很多次扫描其实都是徒劳的；第二，每次都要扫描整个任务列表，如果任务列表很大的话，势必会比较耗时。

# 2. 高性能定时器

- 【方案】：可以用优先级队列来解决。我们按照任务设定的执行时间，将这些任务存储在优先级队列中，队列首部（也就是小顶堆的堆顶）存储的是最先执行的任务。

# 2. 高性能定时器

- 当 T 秒时间过去之后，定时器取优先级队列中队首的任务执行。然后再计算新的队首任务的执行时间点与当前时间点的差值，把这个值作为定时器执行下一个任务需要等待的时间。
- 这样，定时器既不用间隔 1 秒就轮询一次，也不用遍历整个任务列表，性能也就提高了。

# 思考题

- 1. 在堆排序建堆的时候，对于完全二叉树来说，下标从frac{n/2}+1到n的都是叶子节点，这个结论是怎么推导出来的呢？
- 2. 今天讲了堆的一种经典应用，堆排序。关于堆，你还能想到它的其他应用吗？


# 归并排序

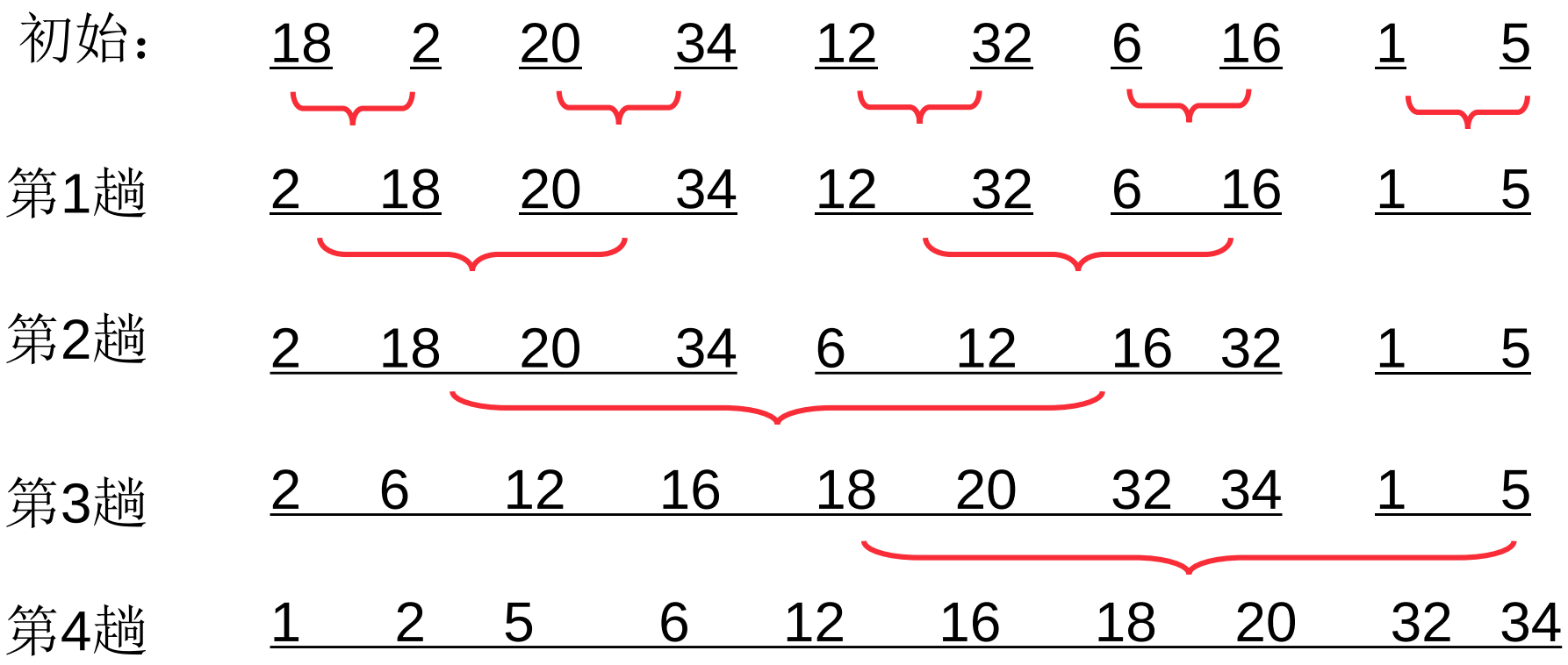
# 思考?

- 比如：如何在O(n)的时间复杂度内查找一个无序数组中的第K大元素？

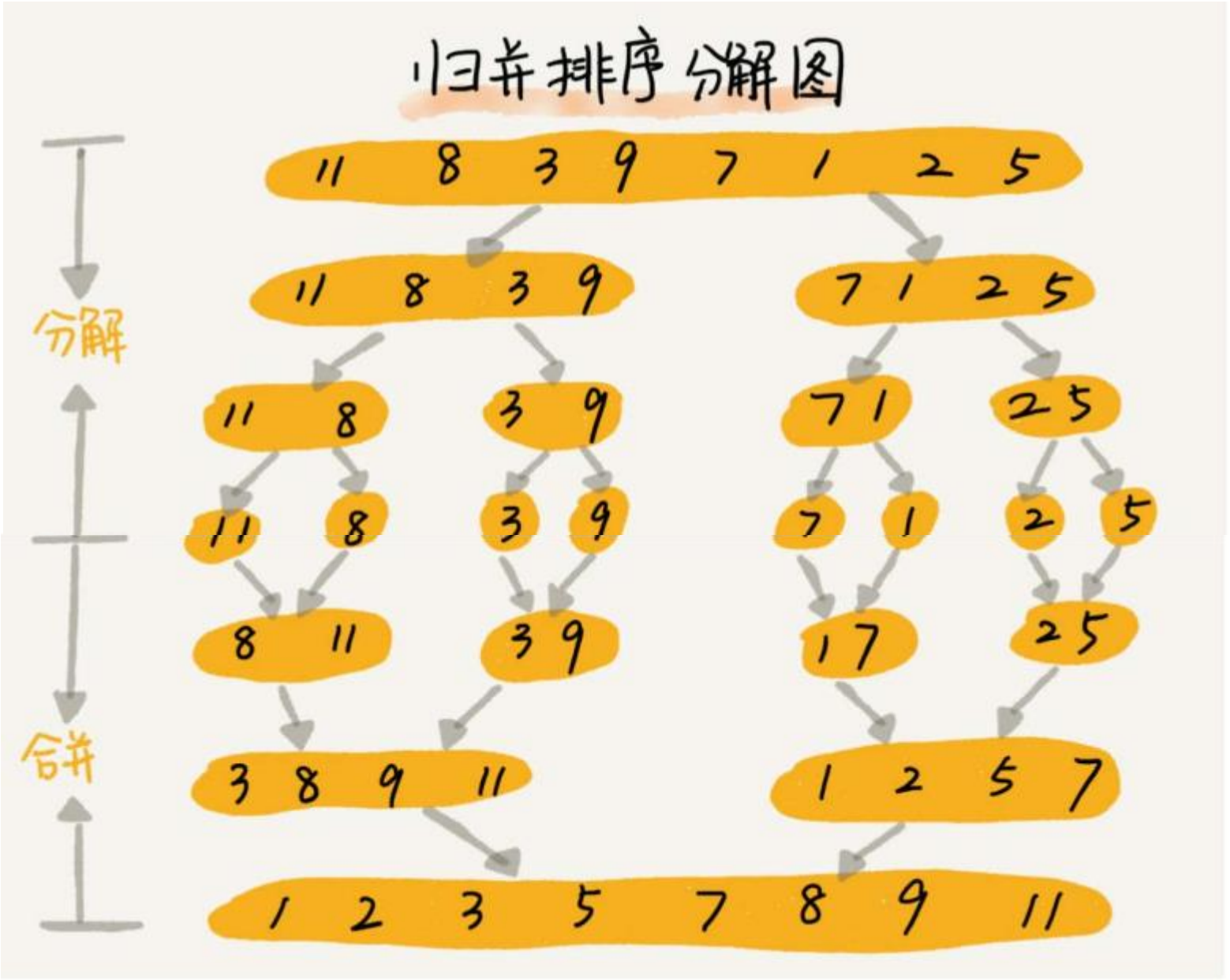
# 归并排序

- 【归并】：把 2 个有序子文件合并成一个有序文件的过程，称为2-路归并。
- 【基本思想】：⑴将文件的每个记录视为一个有序子文件；⑵然后两两子文件进行2-路归并；⑶重复⑵，直到只剩一个长度为n的有序文件

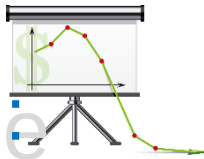
[ 例] 设待排序的表有8个记录,其关键字分别为{18,2,20,34,12,32,6,16,1,5}。说明采用归并排序方法进行排序的过程。




$\log_2 10$取上界为4

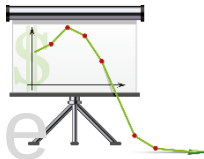

归并排序分解图
分解
合并
11 8 3 9 7 1 2 5
11 8 3 9
7 1 2 5
11 8
3 9
7 1
2 5
11
8
3
9
7
1
2
5
8 11
3 9
1 7
2 5
3 8 9 11
1 2 5 7
1 2 3 5 7 8 9 11

# 一趟归并排序算法 Merge()实现了一次归并：

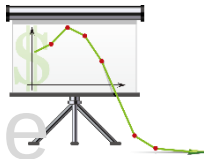


```
void Merge(RecType R[],int low,int mid,int high)
{    RecType *R1;
     int i=low,j=mid+1,k=0;
                          /*k是R1的下标,i、j分别为第1、2段的下标*/
     R1=(RecType *)malloc((high-low+1)*sizeof(RecType));
     while (i<=mid && j<=high)
          if (R[i].key<=R[j].key)  /*将第1段中的记录放入R1中*/
          {  R1[k]=R[i];  i++;k++;  }
           else                          /*将第2段中的记录放入R1中*/
           {  R1[k]=R[j];  j++;k++;  }
```

# 一趟归并排序算法



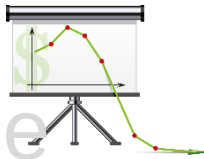

```
        while (i<=mid)                  /*将第1段余下部分复制到R1*/
        {  R1[k]=R[i];  i++;k++;   }
          while (j<=high)             /*将第2段余下部分复制到R1*/
        {   R1[k]=R[j];  j++;k++;  }
          for (k=0,i=low;i<=high;k++,i++) /*将R1复制回R中*/
                R[i]=R1[k];
}
```



code

```
void MergePass(RecType R[],int length,int n)
{      int i;
       for (i=0;i+2*length-1<n;i=i+2*length) /*归并length长的
两相邻子表*/
            Merge(R,i,i+length-1,i+2*length-1);
       if (i+length-1<n) /*余下两个子表,后者长度小于length*/
           Merge(R,i,i+length-1,n-1);          /*归并这两个子表*/
}
```

MergePass()实现了一趟归并




二路归并排序算法如下：

```
void MergeSort(RecType R[],int n)
    /*自底向上的二路归并算法*/
{
        int length;
        for (length=1;length<n;length=2*length)
                MergePass(R,length,n);
}
```

# 算法分析

（1）时间复杂度

每一趟归并的时间复杂度为 **O(n)**， 总共需进行 $\lceil \log_2 n \rceil$ 趟。所以对 n 个记录进行归并排序的时间复杂度为**O($n\log_2 n$)** 。

（2）空间复杂度

**O（n）**

（3）稳定性：**稳定**

# 归并排序

- 2. 归并排序过程（1）
- 此为一趟归并

- 归并排序是稳定的，时间复杂度为0(nlogn)，
- 需要和待排序记录相等数量的辅助空间r2。

# 归并排序

- 2. 归并排序过程（2）
- 经过多次调用一趟归并，直到所有数据归并入一个组。
- 【总结】
  - (1)时间复杂度为：o(n$\log_2$n)
  - (2)实现归并排序过程中需要一个与待排序数量相等的辅助空间空间复杂度为 o(n) .
  - (3)归并排序可以采用非递归排序

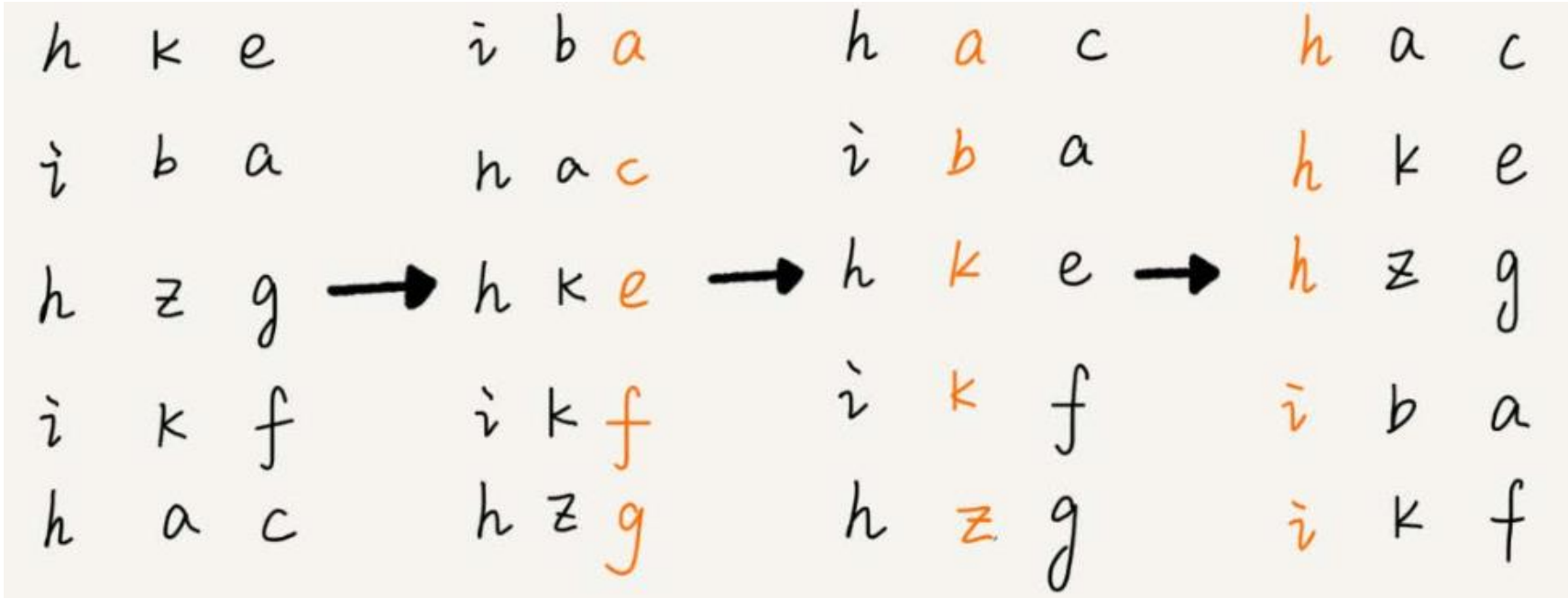
# 06 *Part Six*

# 基数排序

# 思考题?

- 假设有 10 万个手机号码，希望将这 10 万个手机号码从小到大排序，你有什么比较快速的排序方法呢？

# 字符串排序的例子


h k e
i b a
h z g
i k f
h a c
i b a
h a c
h k e
i k f
h z g
h a c
i b a
h k e
i k f
h z g
h a c
h k e
h z g
i b a
i k f

# 基数排序

- 【基数排序法】：是一种用多关键字排序思想对单逻辑关键字进行排序，而无需进行关键字比较的新排序方法，其基本操作是“分配”和“收集”。
- 【基数排序思想】：基数排序是按组成关键字的各位的值进行分配和收集，与前面介绍的排序方法不同，它无需进行关键字之间的比较。
- 设关键字有d位，每位的取值范围为r(称为基数)，则需要进行d 趟分配与收集，需要设立r个队列。例如，若每位是十进制数字，则需要设立10个队列，若每位由小写字母组成，则要设立26个队列 。

# 基数排序

- 【基数排序的步骤】

  (1) 从关键字的低位开始进行第i趟(i=1, 2, . . . d)分配即将单链表中的记录依次按关键字的第i位分配到相应编号的队列中；

  (2) 分配完毕后，将各队列的记录按队列编号顺序收集成一个单链表；

  (3) 重复(1)(2)，直到第d趟收集完毕，所得单链表已成为有序表。

# 基数排序

例：初始　278—109—063—930—589—184—505—269—008—083

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 第一趟分配 | | | | | | | | | | 269 |
| | | | | 083 | | | | | 008 | 589 |
| | 930 | | | 063 | 184 | 505 | | | 278 | 109 |

第一趟收集　930—063—083—184—505—278—008—109—589—269

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 第二趟分配 | 109 | | | | | | | | 589 | |
| | 008 | | | | | | 269 | | 184 | |
| | 505 | | | 930 | | | 063 | 278 | 083 | |

第二趟收集　505—008—109—930—063—269—278—083—184—589

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 第三趟分配 | 083 | | | | | | | | | |
| | 063 | 184 | 278 | | 589 | | | | | |
| | 008 | 109 | 269 | | 505 | | | | | 930 |

第三趟收集　008—063—083—109—184—269—278—505—589—930

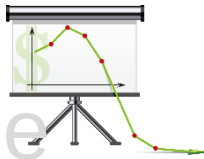
# 基数排序

- 【基数排序的特点】
  - (1) 基数排序的基本操作是"分配"和"收集"，而不是关键字之间的比较；
  - (2) 基数排序是稳定的，其时间复杂度为O(d(n+r))，其中n是记录数，d是关键字的位数，r是关键字各位的值域。
  - (3) 基数排序要进行d趟分配和收集，需r个队列

# 基数排序

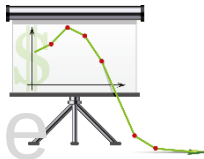
```
#define MAXE 20                 /*线性表中最多元素个数*/
#define MAXR 10                 /*基数的最大取值*/
#define MAXD 8                  /*关键字位数的最大取值*/
typedef struct node
{     char data[MAXD];          /*记录的关键字定义的字符串*/
       struct node *next;
} RecType;
```

# 基数排序




```
void RadixSort(RecType *&p,int r,int d)
{       RecType *head[MAXR],*tail[MAXR],*t;
        int i,j,k;
        for (i=d-1;i>=0;i--)    /*从低位到高位做d趟排序*/
        {       for (j=0;j<r;j++)   /*初始化各链队首、尾指针*/
                        head[j]=tail[j]=NULL;
                while (p!=NULL)  /*对于原链表中每个结点循环*/
                {       k=p->data[i]-'0';       /*找第k个链队*/
                        if (head[k]==NULL)
                                /*进行分配,即采用尾插法建立单链表*/
                        {       head[k]=p;  tail[k]=p;  }
                  else
                        {       tail[k]->next=p;  tail[k]=p;  }
                    p=p->next;     /*取下一个待排序的元素*/}
```

分配



code

收集

```
        p=NULL;
        for (j=0;j<r;j++)     /*对于每一个链队循环进行收集*/
            if (head[j]!=NULL)
            {       if (p==NULL)
                    {     p=head[j];
                          t=tail[j];
                    }
                    else
                    {      t->next=head[j];
                          t=tail[j];
                    }
            }
            t->next=NULL;
                    /*最后一个结点的next域置NULL*/
    }   }
```

- 基数排序对要排序的数据是有要求的，需要可以分割出独立的“位”来比较，而且位之间有递进的关系，如果a数据的高位比b数据大，那剩下的低位就不用比较了。除此之外，每一位的数据范围不能太大，要可以用线性排序算法来排序，否则，基数排序的时间复杂度就无法做到O(n)了。

# 开篇问题求解

- 如何根据年龄给 100 万用户排序？现在思考题是不是变得非常简单了呢？
- 【方案】：实际上，根据年龄给 100 万用户排序，就类似按照成绩给 50 万考生排序。我们假设年龄的范围最小 1 岁，最大不超过 120 岁。我们可以遍历这 100 万用户，根据年龄将其划分到这 120 个桶里，然后依次顺序遍历这 120 个桶中的元素。这样就得到了按照年龄排序的 100 万用户数据。

- 排序算法，有桶排序、计数排序、基数排序。它们对要排序的数据都有比较苛刻的要求，应用不是非常广泛。但是如果数据特征比较符合这些排序算法的要求，应用这些算法，会非常高效，线性时间复杂度可以达到 O(n) 。
- 桶排序和计数排序的排序思想是非常相似的，都是针对范围不大的数据，将数据划分成不同的桶来实现排序。基数排序要求数据可以划分成高低位，位之间有递进关系。比较两个数，只需要比较高位，高位相同的再比较低位。而且每一位的数据范围不能太大，因为基数排序算法需要借助桶排序或者计数排序来完成每一个位的排序工作。

# 08 *Part Eight*

# 各种排序的比较

# 各种内排序方法的比较和选择

- 本章介绍了多种排序方法，通常可按平均时间将排序方法分为三类：
  - （1）平方阶$O(n^2)$排序，一般称为简单排序，例如直接插入、直接选择和冒泡排序；
  - （2）线性对数阶$O(n\log_2 n)$排序，如快速、堆和归并排序；
  - （3）线性阶$O(n)$排序，如基数排序。

| 排序方法 | 时间复杂度 | | | 空间复杂度 | 稳定性 | 复杂性 |
|---|---|---|---|---|---|---|
| | 平均情况 | 最坏情况 | 最好情况 | | | |
| 直接插入排序 | $O(n^2)$ | $O(n^2)$ | $O(n)$ | $O(1)$ | 稳定 | 简单 |
| 希尔排序 | $O(n\log_2 n)$ | $O(n\log_2 n)$ | | $O(1)$ | 不稳定 | 较复杂 |
| 冒泡排序 | $O(n^2)$ | $O(n^2)$ | $O(n)$ | $O(1)$ | 稳定 | 简单 |
| 快速排序 | $O(n\log_2 n)$ | $O(n^2)$ | $O(n\log_2 n)$ | $O(n\log_2 n)$ | 不稳定 | 较复杂 |
| 直接选择排序 | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ | 不稳定 | 简单 |
| 堆排序 | $O(n\log_2 n)$ | $O(n\log_2 n)$ | $O(n\log_2 n)$ | $O(1)$ | 不稳定 | 较复杂 |
| 归并排序 | $O(n\log_2 n)$ | $O(n\log_2 n)$ | $O(n\log_2 n)$ | $O(n)$ | 稳定 | 较复杂 |
| 基数排序 | $O(d(n+r))$ | $O(d(n+r))$ | $O(d(n+r))$ | $O(n+r)$ | 稳定 | 较复杂 |