数据结构与算法

Data Structrues and Algorithms

张岩

海量数据计算研究中心


哈工大计算机科学与技术学院

第6章 内部排序

学习目标

- 掌握排序的基本概念和常用术语
- 熟练掌握插入排序、希尔排序；冒泡排序、快速排序；选择排序、锦标赛排序、堆排序；归并排序；基数排序的基本思想、算法原理、排序过程和算法实现。
- 掌握各种排序算法的性能及其分析方法，以及各种排序方法的比较和选择等。

本章主要内容

- 6.1 基本概念
- 6.2 交换排序
  - 冒泡排序
  - 快速排序
- 6.3 选择排序
  - 直接选择排序
  - 锦标赛排序
  - 堆排序
- 6.4插入排序
  - 直接插入排序
  - 折半插入排序
  - 希尔排序
- 6.5 (二路)归并排序
- 6.6 基数排序
- 本章小结

6.1 基本概念

- 排序（Sorting）也称分类:
  - 输入：n个记录的序列：$\{R_1, R_2, R_3, \ldots, R_n\}$，

    其对应的关键字序列：　$\{K_1, K_2, K_3, \ldots, K_n\}$
  - 输出：　n个记录序列　$R_{p_1}, R_{p_2}, R_{p_3}, \ldots, R_{p_n}$，使得

    $K_{p_1} \le K_{p_2} \le K_{p_3} \le \ldots \le K_{p_n}$(升序)，或者

    $K_{p_1} \ge K_{p_2} \ge K_{p_3} \ge \ldots \ge K_{p_n}$(降序)
    - 实质是确定1，2，…,n的一种置换p_1，p_2，…，p_n，使其对应的关键字按非递减（或非递增）的顺序排列
    - 当待排序记录的关键字值均不相同时，则排序的结果是唯一的，否则排序结果可能不唯一
- 排序的目的：方便查询和处理。例如：折半查找，字典等

6.1 基本概念

- 排序算法的稳定性：
  - 假设 $K_i = K_j$，且排序前序列中 R_i 领先于 R_j；
  - 若在排序后的序列中 R_i 仍领先于 R_j，则称排序方法是稳定的。
  - 若在排序后的序列中 R_j 仍领先于 R_i，则称排序方法是不稳定的。
  - 例如，序列　3　15　8　8　6　9
  - 若排序后得　3　6　8　8　9　15
  - 若排序后得　3　6　8　8　9　15
  - 不稳定的排序说明什么？不能说明什么？
    - 不能说明排序算法不正确，只能说明可能做了不必要的交换
    - 排序方法的稳定性是针对所有输入实例而言的

稳定的
不稳定的

6.1 基本概念(cont.)

- 排序分类:
  - 按照排序时排序对象存放的设备，分为，
    - 内部排序:排序过程中数据对象全部在内存中的排序。
    - 外部排序:排序过程数据对象并非完全在内存中的排序
  - 按照排序是的基本操作是否基于关键字的比较？分为，
    - 基于比较：基本操作——关键字的比较和记录的移动，其最差时间下限已经被证明为$\Omega(n\log_2 n)$。
    - 交换排序（冒泡、快速排序）；选择排序（直接选择、堆排序）；插入排序（直接插入、折半插入、希尔排序）、归并排序（二路归并排序）。
    - 不基于比较：根据组成关键字的的分量及其分布特征，如基数排序。

6.1 基本概念(cont.)

- 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：关键字之间的比较；
    - 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。

6.1 基本概念(cont.)

- 排序算法及其存储结构:
  - 从操作角度看，排序是线性结构的一种操作，待排序记录可以用顺序存储结构或链接存储结构存储。我们假定采用顺序存储结构：

```
struct   records {
        keytype   key ;
        fields   other ;
};
typedef   records  LIST[maxsize] ;
```

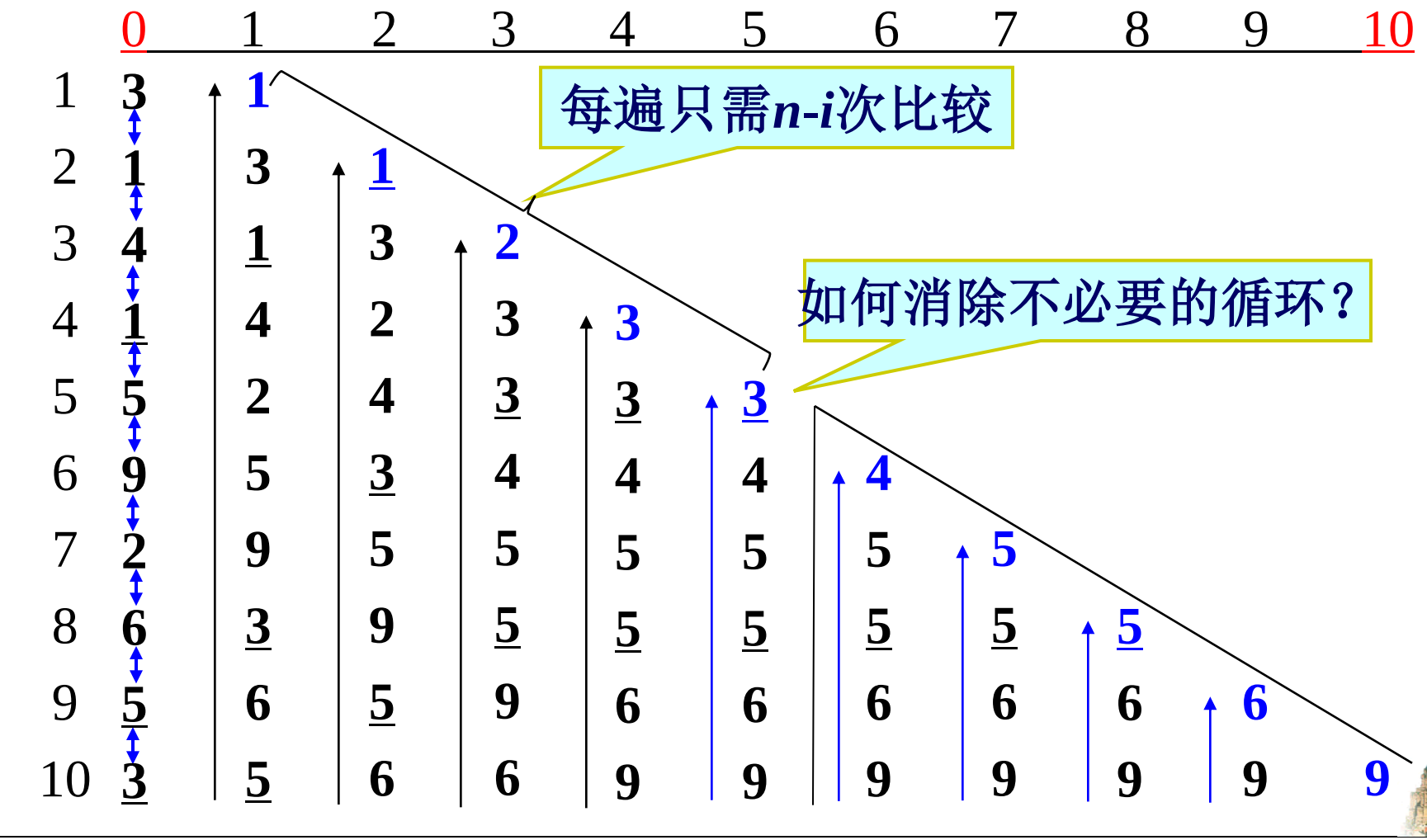
  - Sort（ int n , LIST &A）:对n个记录的数组按照关键字不减的顺序进行排序。

6.2 冒泡排序

- 算法的基本思想:
  - 将待排序的记录看作是竖着排列的“气泡”，关键字较小的记录比较轻，从而要往上浮。
  - 对这个“气泡”序列进行*n*-1遍（趟）处理。所谓一遍（趟）处理，就是自底向上检查一遍这个序列，并注意两个相比较的关键字的顺序是否正确。如果发现顺序不对，即“轻”的记录在下面，就交换它们的位置。
  - 显然，处理1遍之后，“最轻”的记录就浮到了最高位置；处理2遍之后，“次轻”的记录就浮到了次高位置。在作第二遍处理时，由于最高位置上的记录已是“最轻”的，所以不必检查。一般地，第i遍处理时，不必检查第i高位置以上的记录的关键字，因为经过前面*i*-1遍的处理，它们已正确地排好序。

6.2 冒泡排序

- 示例：3，1，4，1，5，9，2，6，5，3



6.2冒泡排序(cont.)

- 算法的实现:

```
void  BubbleSort ( int  n , LIST  &A )//内外层循环优化
{   for ( int i =1; i <= n−1; i++ ){ //一共要排序n-1趟
      int sp = 0; //每趟排序标志位都要先置为0，判断内层循环是否发生了交换
      for ( int j =n; j >= i+1; j--) {//选出该趟排序的最小值前移；内层循环已优化
         if ( A[j].key < A[j-1].key ){
            swap (A[j], A[j-1]);
            sp = 1; //只要有发生交换，sp就置为1
         }
      }
      if (sp ==0){ //若标志位为0，说明所有元素已经有序，就直接返回
          return;
      }
   }
}/* BubbleSort */
```

```
void  swap(records &x,  records &y)
{   records   t ;
    t = x ;     x = y ;     y = t ;
}
```

6.2冒泡排序(cont.)

- 算法（时间）性能分析:
  - 最好情况（正序）:
    - 比较次数：n-1
    - 移动次数：0
    - 时间复杂度： $O(n)$;
  - 最坏情况（反序）:
    - 比较次数： $\sum_{i=1}^{n-1}(n-i) = \frac{n(n-1)}{2}$
    - 移动次数： $\sum_{i=1}^{n-1}3(n-i) = \frac{3n(n-1)}{2}$
    - 时间复杂度： $O(n^2)$;
  - 平均情况：时间复杂度为$O(n^2)$。

空间复杂度： $O(1)$。

6.3 快速排序

- 快速算法是对冒泡排序的改进，改进的着眼点：
  - 在气泡排序中，记录的比较和移动是在相邻单元中进行的，记录每次交换只能上移或下移一个单元，因而总的比较次数和移动次数较多。

减少总的比较次数和移动次数

↓

增大记录的比较和移动距离

↓

较大记录从前面直接移动到后面
较小记录从后面直接移动到前面

每次交换是跳跃式

6.3 快速排序(cont.)

- 基本思想：
  - 是1980年图灵奖得主C.R.A.Hoare 在1960年提出的一种划分交换排序。采用分治策略，以减少排序过程的比较次数
- 分治法的基本思想
  - 分解(划分)：将原问题分解为若干个与原问题相似的子问题
  - 求解：递归地求解子问题。若子问题规模足够小，则直接求解
  - 组合：将每个子问题的解组合成原问题的解。
- 快排算法的思想
  - 任选一个记录的关键字作为基准元素(控制关键字、枢轴)
  - 将待排序的记录划分成独立的两部分，其中一部分的所有记录关键字都比另外一部分的所有记录关键字都小（划分交换，一次划分排序、一遍快速排序）
  - 递归地对这两部分进行划分排序，以达到整个记录序列变成有序序列。

6.3 快速排序(cont.)

- 算法的实现步骤：

设被排序的无序区为A[*i*],……,A[*j*]

1. 基准元素选取：选择其中的一个记录的关键字*v*作为基准元素(控制关键字)；(怎么选择?)

2. 划分：通过基准元素*v*把无序区A[*i*],……,A[*j*]划分成左、右两部分，使得*v*左边的各记录的关键字都小于等于*v*；*v*右边的各记录的关键字都大于等于*v*；(如何划分?)

3. 递归求解：重复(1)~(2)，分别对左边和右边部分递归地进行快速排序；

4. 组合：左、右两部分均有序，整个序列有序。

3	1	4	1	5	9	2	6	5	3

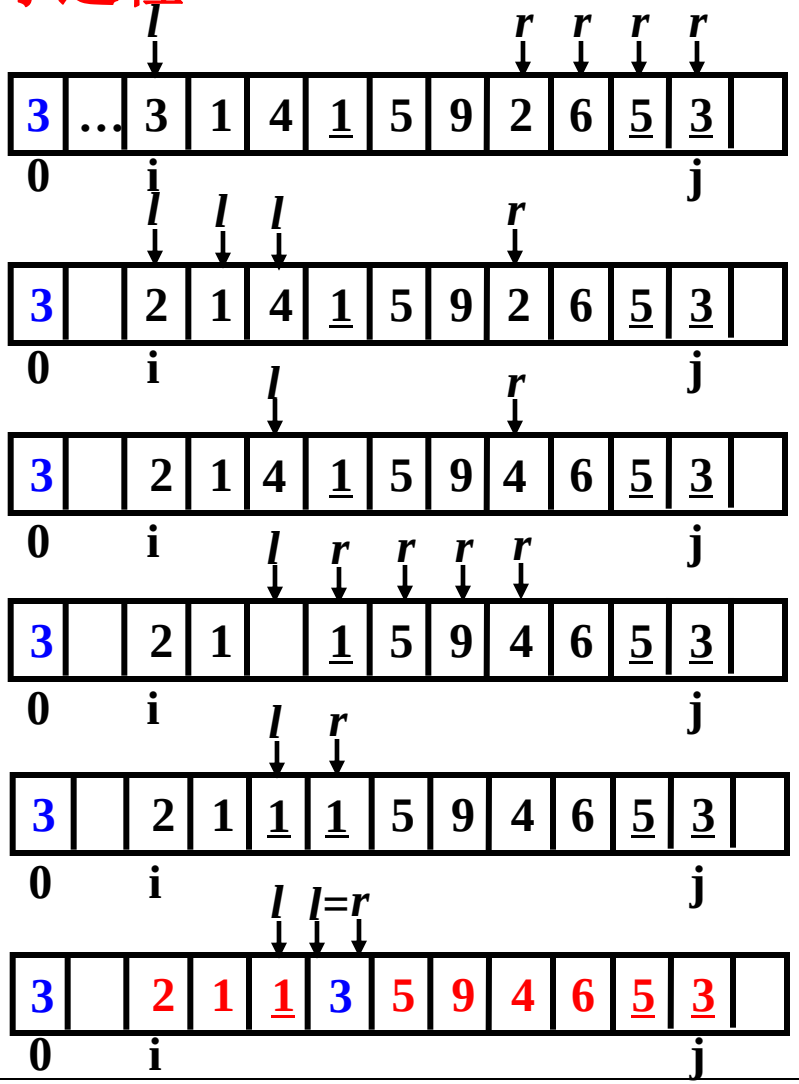
6.3 快速排序(cont.)

- 快速排序的实现： /*对外部数组A 的元素A[i],…,A[j]进行快速排序*/

```
void QuickSort ( LIST A, int i , int  j ) //挖坑/填坑+分治
{    keytype pivot;  //基准元素
     int k; //关键字≥pivot的记录在序列中的起始下标
     int pivotindex ; //关键字为pivot的记录在数组A中的下标
     pivotindex = FindPivot ( A, i , j );
     if( pivotindex != 0 ) {  //递归终止条件，一般地 i<j
          pivot=A[pivotindex].key;
          k=Partition ( A, i , j , pivot );    //挖坑/填坑
          QuickSort ( A, i , k-1);
          QuickSort ( A, k+1 , j);
     }
} //对数组A[1],…,A[n]进行快速排序可调用QuickSort(A, 1, n)实现
```

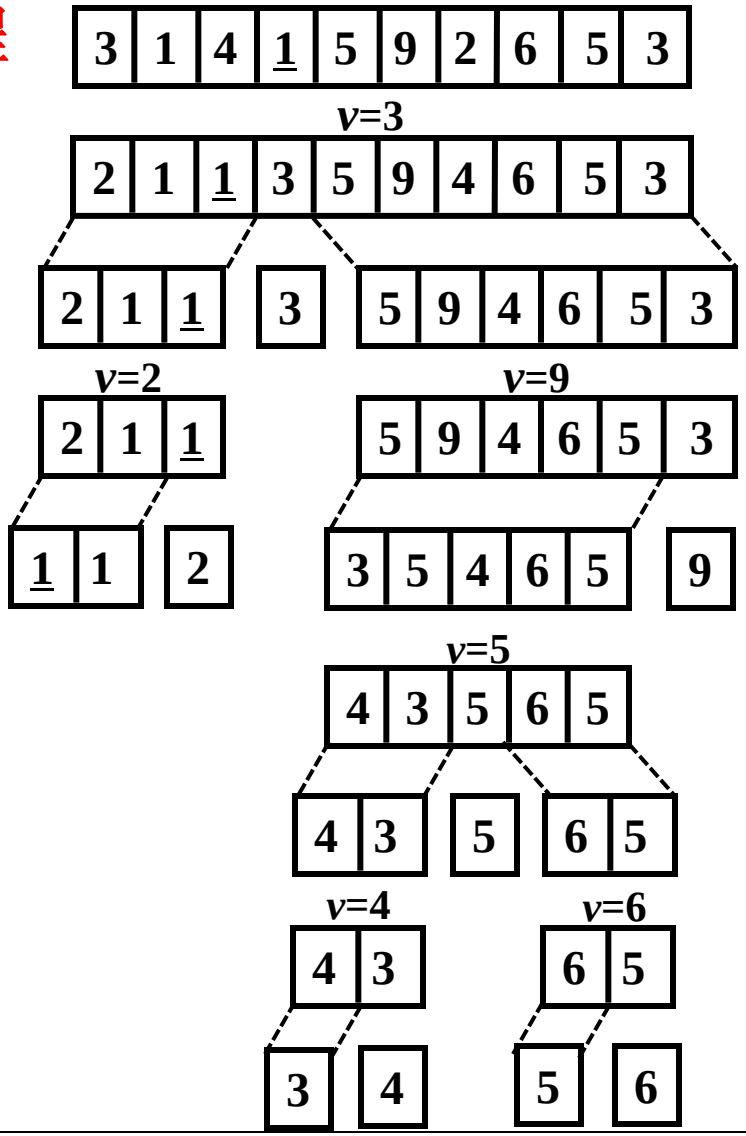
6.3 快速排序(cont.)

示例：一次划分排序过程



6.3 快速排序(cont.)

示例：快速排序过程



6.3 快速排序(cont.)

- 无序区划分（分割）：

3	1	4	1	5	9	2	6	5	3

设被排序的无序区为A[*i*…*j*]

(1) 扫描与移动---挖坑+填坑

- 令游标 *r* 从右(初始时 *r* = *j*)向左扫描，越过关键字大于等于*v* 的所有记录，直到遇到A[*r*].key<*v*； A[*l*]=A[r]。
- 令游标 *l* 从左(初始时*l* = *i*)向右扫描，越过关键字小于等于*v* 的所有记录，直到遇到A[*l*].key≥*v*； A[r]=A[*l*]。

(2) 测试*l* 和 *r*: 若*l* < *r*，则转(1)； 否则(*l* = *r*,)转(3)；

(3) 基准定位： A[*l*]=*v*(基准记录*v*定位于其排序后的最终位置).
此时A[*i*…*j*]被划分成为独立的两部分:

A[i…*l*-1].key ≤ A[*l*].key=*v* ≤ A[*l*+1…j].key

6.3 快速排序(cont.)

```
int Partition (LIST A , int i , int j , keytype  pivot )
/*对A[i…j]进行一次划分排序,使A[i…l-1].key≤ A[l].key=pivot ≤A[l+1…j].key,
返回pivot所在的下标 */
{    int l = i, r = j;
     do { /*从右向左找第1个小于pivot/* /
          while( l < r && A[r].key >= pivot )   r=r−1;
          A[l] = A[r];  /*将A[r]移至A[l] */
          /*从左向右找第1个大于pivot*/
          while( l < r && A[l].key <= pivot )   l=l+1;
          A[r] = A[l];  /*将A[l]移至A[r] */
    } while ( l < r );
    A[l] = pivot ; /*将基准记录定位到排序的最终位置 */
    return  l ;
} /* Partition */
```

6.3 快速排序(cont.)

- 基准元素的选取:

3	1	4	1	5	9	2	6	5	3

  - 基准元素的选取是任意的，但不同的选取方法对算法性能影响很大;
  - 一般原则：是每次都能将表划分为规模相等的两部分（最佳情况）。此时，划分次数为$\log_2 n$，全部比较次数$n\log_2 n$，交换次数$(n/6)\log_2 n$。
  - 设 FindPivot(A, *i* , *j*)是求A[*i*].key,…,A[*j*].key的基准元素*v*=A[*k*]，返回其下标*k* 。
    - *v* =(A[*i*].key，A[(*i*+*j*)/2].key，A[*j*].key的中值)
    - *v* =从A[*i*].key到A[*j*].key 最先找到的两个不同关键字中的最大者。（若A[*i*].key,…A[*j*].key之中至少有两个关键字不相同 ）
      - 优点：若无两个关键字不同，则*A*[*i*]到A[*j*]已有序，排序结束。

6.3 快速排序(cont.)

```
int   FindPivot( LIST A, int  i, int  j )      /* 设A是外部数组 */
/*若A[i],…A[j]的关键字全部相同，则返回0;
   否则，左边两个不同关键字中的较大者的下标。*/
{   keytype   firstkey = A[i].key ; /* 第1个关键字的值A[i].key */
    int  k ;                                  /* 从左到右查找不同的关键字  */
    for ( k=i+1 ;  k<=j; k++ )          /* 扫描不同的关键字 */
        if ( A[k].key > firstkey )   /* 选择较大的关键字 */
                return  k ;
        else if ( A[k].key < firstkey )
                return  i ;
    return  0 ;
}/* FindPivot */
```

3	1	4	1	5	9	2	6	5	3

6.3 快速排序(cont.)

- 快速排序（时间）性能分析
  - 最好情况：
    - 每一次划分后，划分点的左侧子表与右侧子表的长度相同，则有，为$O(n\log_2 n)$。
    - $$\begin{cases} T(n) \le 2T(n/2)+n \\ T(1)=C,\ C\text{为正的常数} \end{cases}$$

$$\begin{aligned} T(n) &\le 2T(n/2)+1n \\ &\le 2(2T(n/4)+n/2)+n = 4T(n/4)+2n \\ &\le 4(2T(n/8)+n/4)+2n = 8T(n/8)+3n \\ &\cdots\cdots\cdots \\ &\le nT(1)+(\log_2 n)\,n = O(n\log_2 n) \end{aligned}$$

    - 时间复杂度为$O(n\log_2 n)$
    - 空间复杂度为$O(\log_2 n)$

6.3 快速排序(cont.)

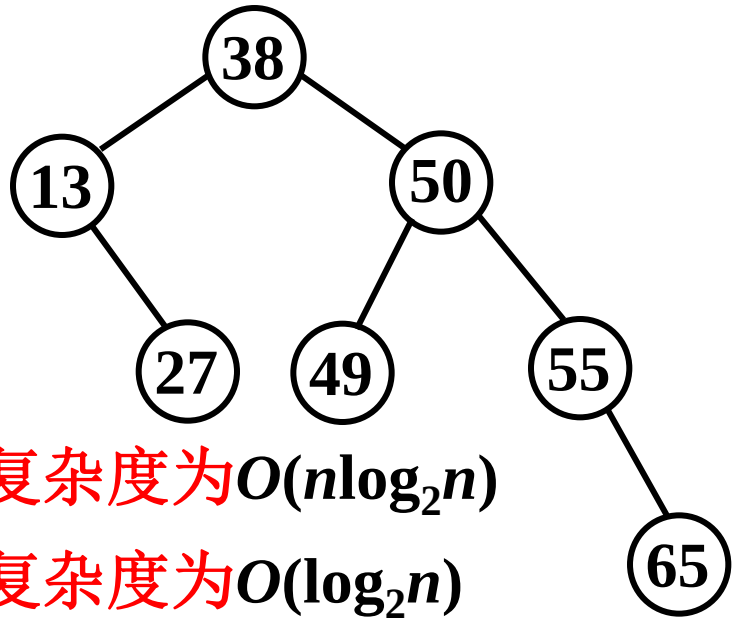
- 快速排序（时间）性能分析
  - 最坏情况：
    - 每次划分只得到一个比上一次划分少一个记录的子序列（另一个子序列长度为0），则有

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

    - 时间复杂度为$O(n^2)$
    - 空间复杂度为$O(n)$

6.3 快速排序(cont.)

- 快速排序（时间）性能分析
  - 平均情况：
    - 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: (pivot=序列的第1个元素)



    - 时间复杂度为$O(n\log_2 n)$
    - 空间复杂度为$O(\log_2 n)$

6.3 快速排序(cont.)

快速排序（时间）性能分析

平均情况：

$$\begin{cases} T(n)=cn+\frac{1}{n}\sum_{k=0}^{n-1}[T(k)+T(n-1-k)] \\ T(0)=T(1)=c \end{cases}$$

$$T(n)=cn+\frac{1}{n}\sum_{k=0}^{n-1}[T(k)+T(n-1-k)]$$
$$=cn+\frac{2}{n}\sum_{k=0}^{n-1}T(k)$$

$$nT(n)=cn^2+2\sum_{k=0}^{n-1}T(k)$$
$$(n+1)T(n+1)=c(n+1)^2+2\sum_{k=1}^{n}T(k)$$

$$(n+1)T(n+1)-nT(n)=c(n+1)^2-cn^2+2T(n)$$
$$=c(2n+1)+2T(n)$$

$$(n+1)T(n+1)=c(2n+1)+(n+2)T(n)$$

$$T(n+1)=\frac{c(2n+1)}{n+1}+\frac{n+2}{n+1}T(n)$$

$$T(n+1)\le 2c+\frac{n+2}{n+1}T(n)$$
$$=2c+\frac{n+2}{n+1}\left(2c+\frac{n+1}{n}T(n-1)\right)$$
$$=2c+\frac{n+2}{n+1}\left(2c+\frac{n+1}{n}\left(2c+\frac{n}{n-1}\right)T(n-2)\right)$$
$$=2c+\frac{n+2}{n+1}\left(2c+\ldots+\frac{4}{3}\left(2c+\frac{3}{2}T(1)\right)\right)$$
$$=2c\left(1+\frac{n+2}{n+1}+\frac{n+2}{n+1}\frac{n+1}{n}+\ldots+\frac{n+2}{n+1}\frac{n+1}{n}\ldots\frac{3}{2}\right)$$
$$=2c\left(1+(n+2)\left(\frac{1}{n+1}+\frac{1}{n}+\ldots+\frac{1}{2}\right)\right)$$
$$=2c+2c(n+2)(H_{n+1}-1)$$

$$H_n=1+\frac{1}{2}+\ldots+\frac{1}{n}=\theta(\ln n) \quad (\ln n<H_n<1+\ln n)$$

$$T(n)=\theta(n\log n)$$

6.3 快速排序(cont.)

快速排序优化

- 快速排序还有很多改进版本，例如随机选择基准，区间内数据较少时直接用其他方法排序以减小递归深度。
- 随机选择基准与将待排序列重新随机排列
- /*产生k个[m,n)范围内的不重复随机整数（m<n,0<k<=n-m)*/

```
int last = n-m-1;
srand((unsigned)time(NULL));
for(int i = 0; i < last; i++){
    int index = rand()% last;
    int temp = sequence[index];
    sequence[index] = sequence[last];
    sequence[last]=temp;
    last--;
}
```

6.4 直接选择排序

算法的基本思想

- 选择排序的主要操作是选择，其主要思想是：每趟排序在当前待排序序列中选出关键字值最小（最大）的记录，添加到有序序列中。
- 直接选择排序，对待排序的记录序列进行n-1遍的处理，第1遍处理是将A[1…n]中最小者与A[1]交换位置，第2遍处理是将A[2…n]中最小者与A[2]交换位置，……，第i遍处理是将A[i…n]中最小者与A[i]交换位置。这样，经过i遍处理之后，前i个记录的位置就已经按从小到大的顺序排列好了。
- 直接选择排序与冒泡排序的区别在：冒泡排序每次比较后，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。

6.4 直接选择排序(cont.)

- 算法的实现

```
void  SelectSort (int n, LIST A )
{   keytype lowkey;      //当前最小关键字
    int  i, j，  lowindex; //当前最小关键字的下标
    for(i=1; i<n; i++) {  //在A[i…n]中选择最小的关键字，与A[i]交换
        lowindex = i ;
        lowkey = A[i].key ;
        for ( j = i+1; j<=n; j++) //SelectMinKey( int i , List A)
            if (A[j].key<lowkey) { //用当前最小关键字与每个关键字比较
                lowkey=A[j] ;
                lowindex = j ;
            }
        if ( i != lowindex ) swap(A[i], A[lowindex]) ;//减少不必要交换
    }
}/* SelectSort*/
```

6.4 直接选择排序(cont.)

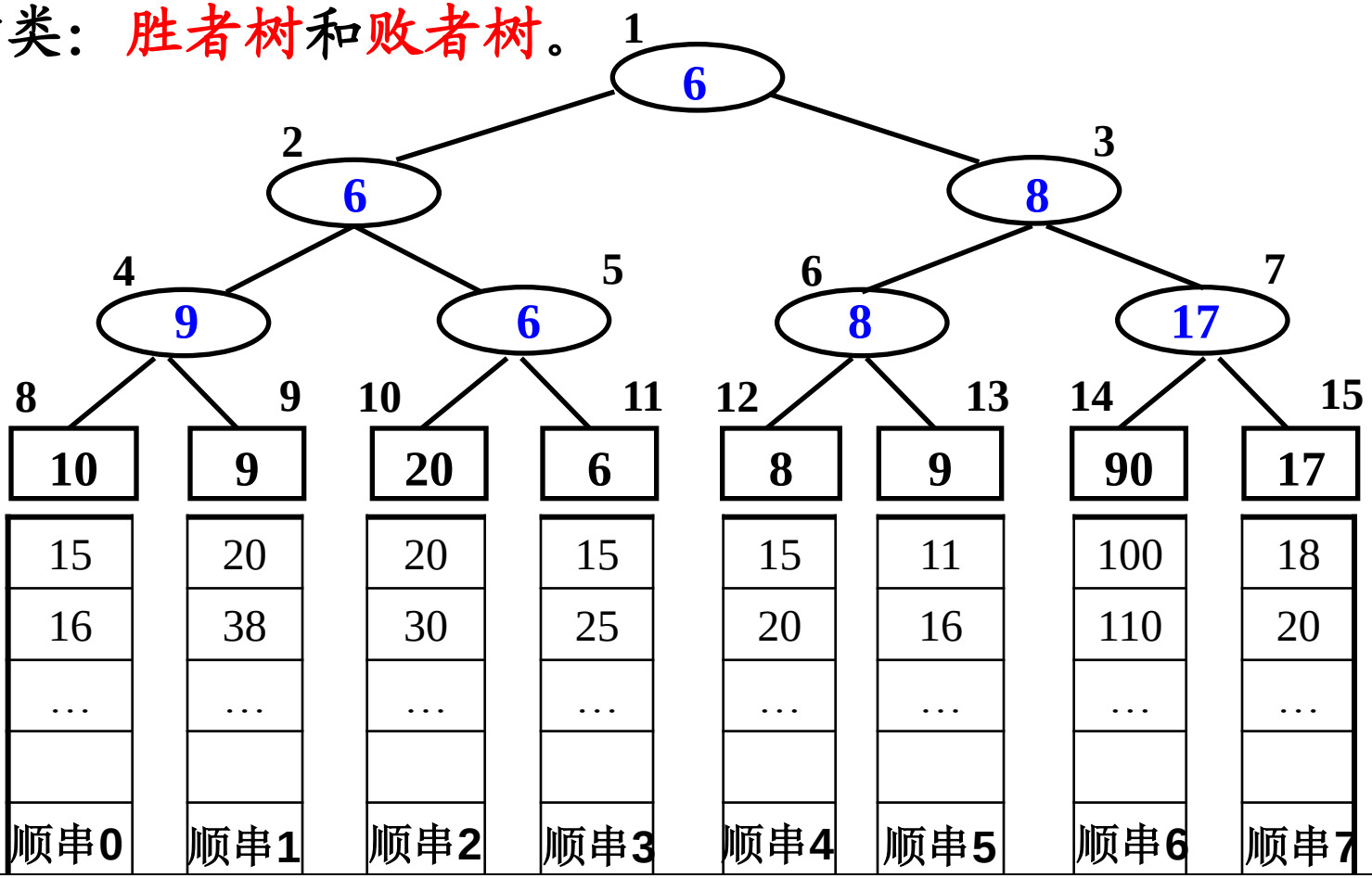
- 性能分析
  - 移动次数:
    - 最好情况（正序）： 0次
    - 最坏情况: 3(*n*–1)次
  - 比较次数:

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

  - 时间复杂度为$O(n^2)$。
  - 空间复杂度为$O(1)$。

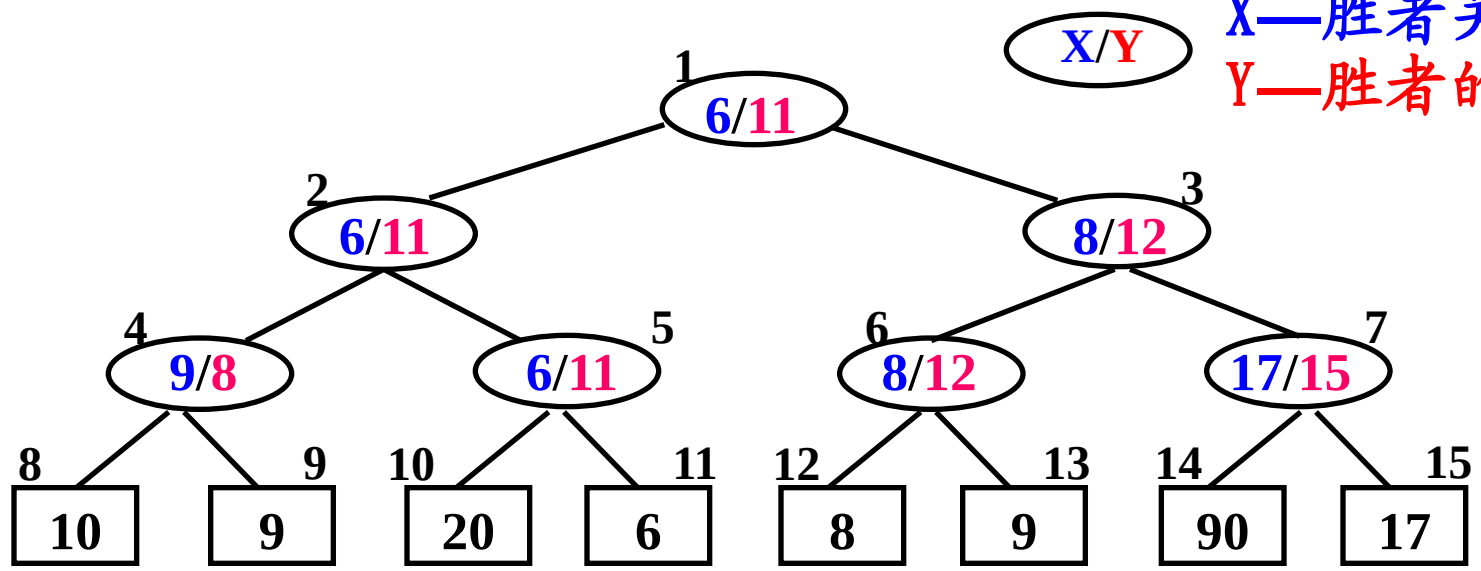
6.5 锦标赛排序(cont.)

- 选择树：也称Tournament Tree是能够记载上一次比较所获知识的完全二叉树。
- 种类：胜者树和败者树。



6.5 锦标赛排序(cont.)

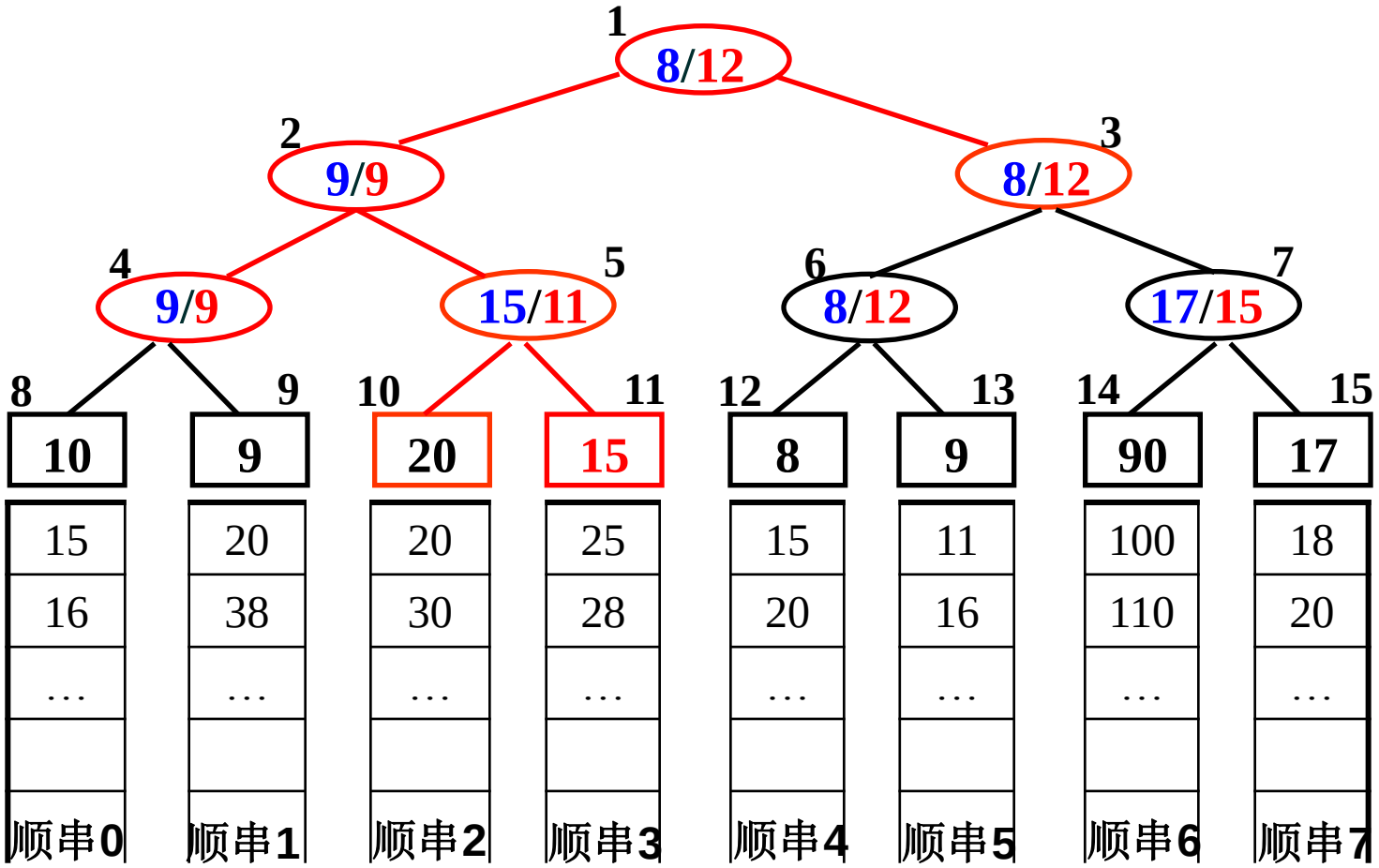
- 胜者树(Winner Tree)的构建



- 具有n个外结点和n-1个内结点
- 外结点为比赛选手,内结点为一次比赛,每一层为一轮比赛
- 比赛在兄弟结点间进行，胜者保存到父结点中
- 根结点保存最终的胜者

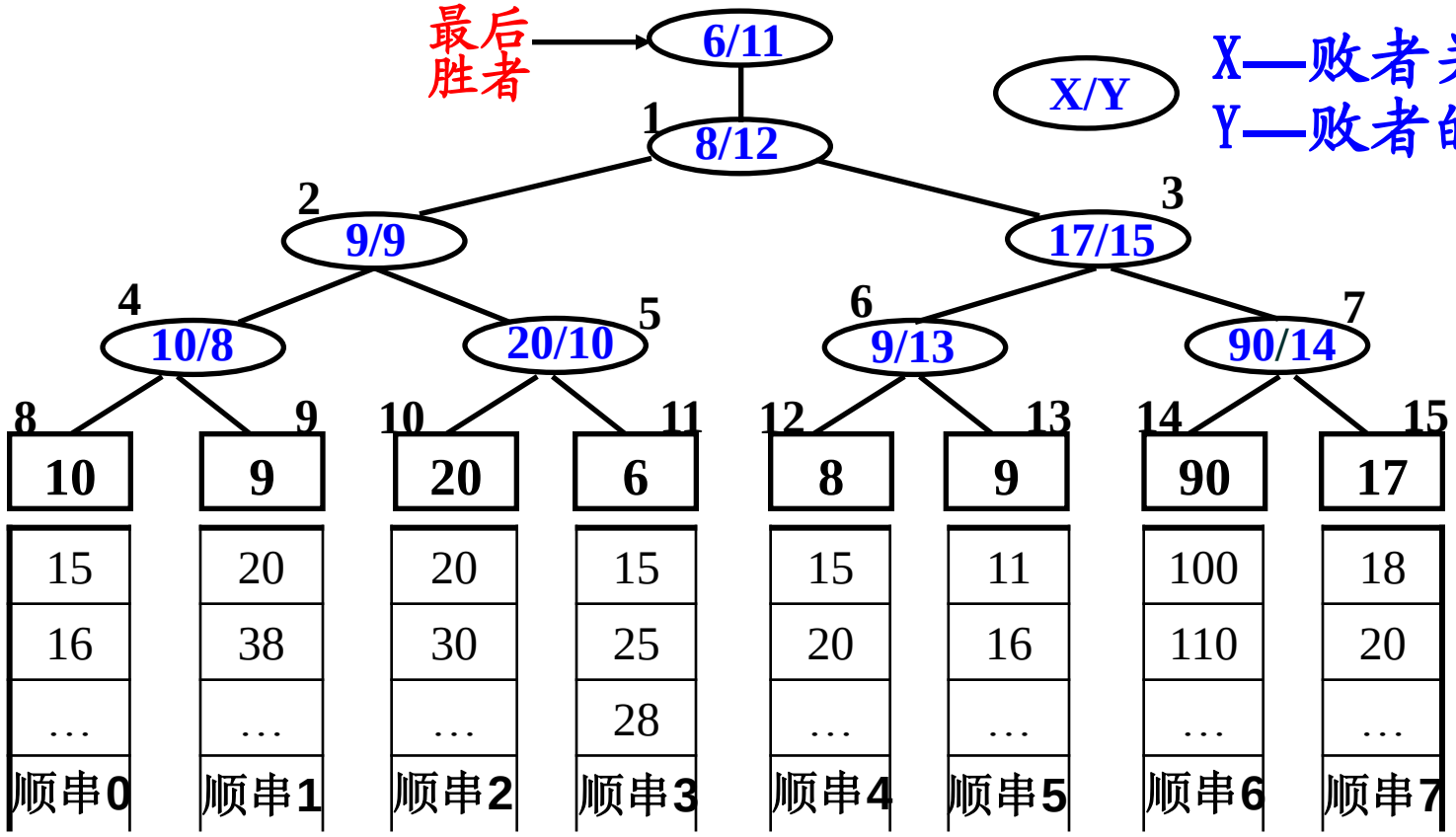
6.5 锦标赛排序(cont.)

- **胜者树的重构:**
  - 新进入的结点与兄弟结点比较，直到树根



6.5 锦标赛排序(cont.)

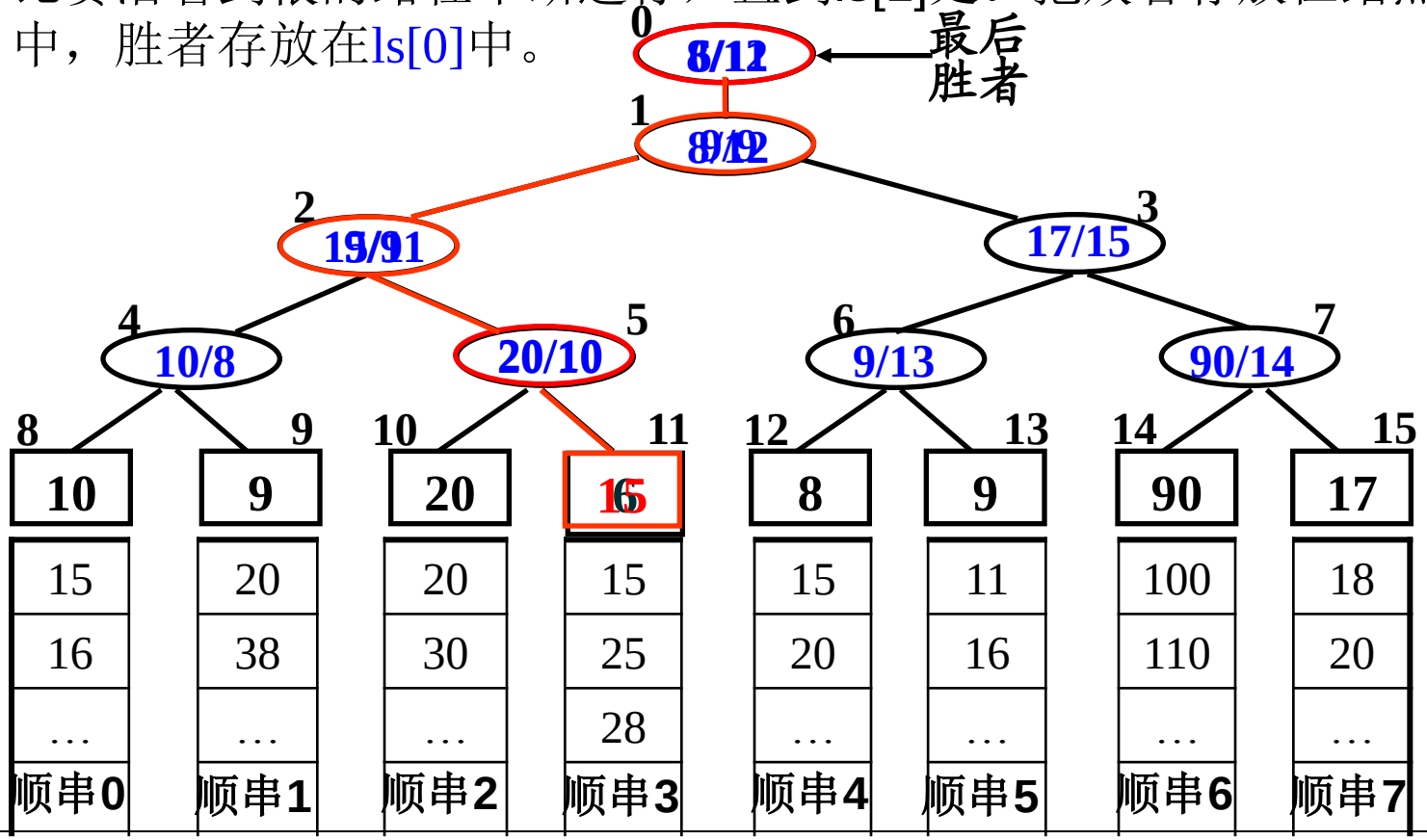
- 败者树(Loser Tree)的构建
  - 内部结点保存败者，胜者参加下一轮比赛
  - 根结点记录比赛的败者，最终的胜者需一个结点进行记录



6.5 锦标赛排序(cont.)

败者树的重构

- 将新进入的结点与其父结点进行比赛：将败者存入父结点中，胜者再与上一级的父结点比较。
- 比赛沿着到根的路径不断进行，直到ls[1]处。把败者存放在结点ls[1]中，胜者存放在ls[0]中。



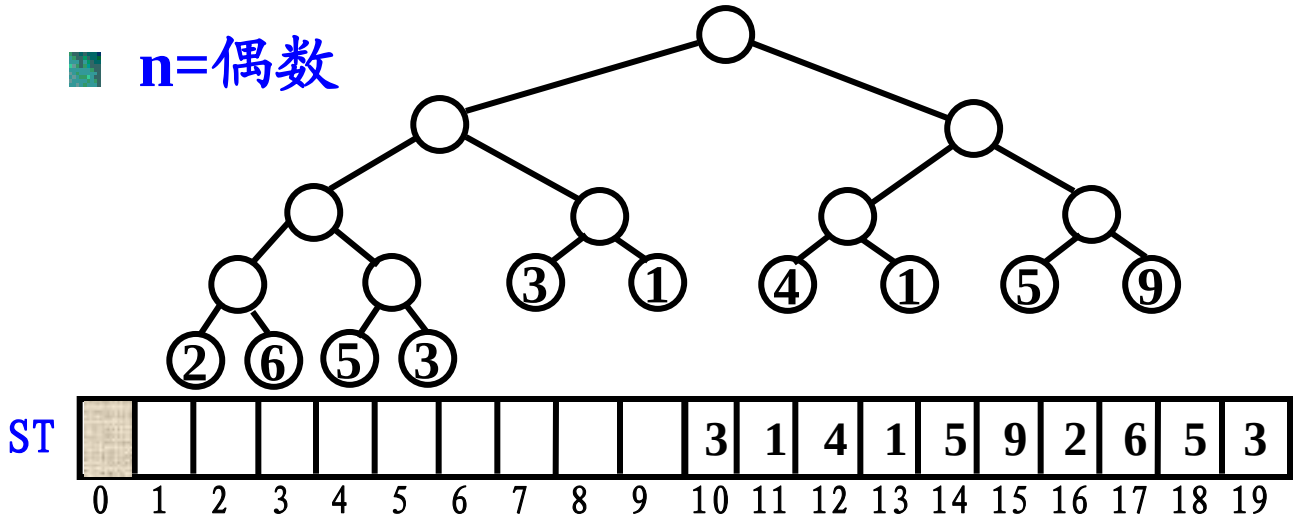
6.5 锦标赛排序

- 锦标赛排序的基本思想
  - 首先通过类似于淘汰赛的方式，选出最小（大）关键字记录（冠军）----构建选择树
  - 然后根据关键字大小，依次选出其他记录，从而实现对整个记录序列的排序----重构选择树
- 胜者树类型说明：

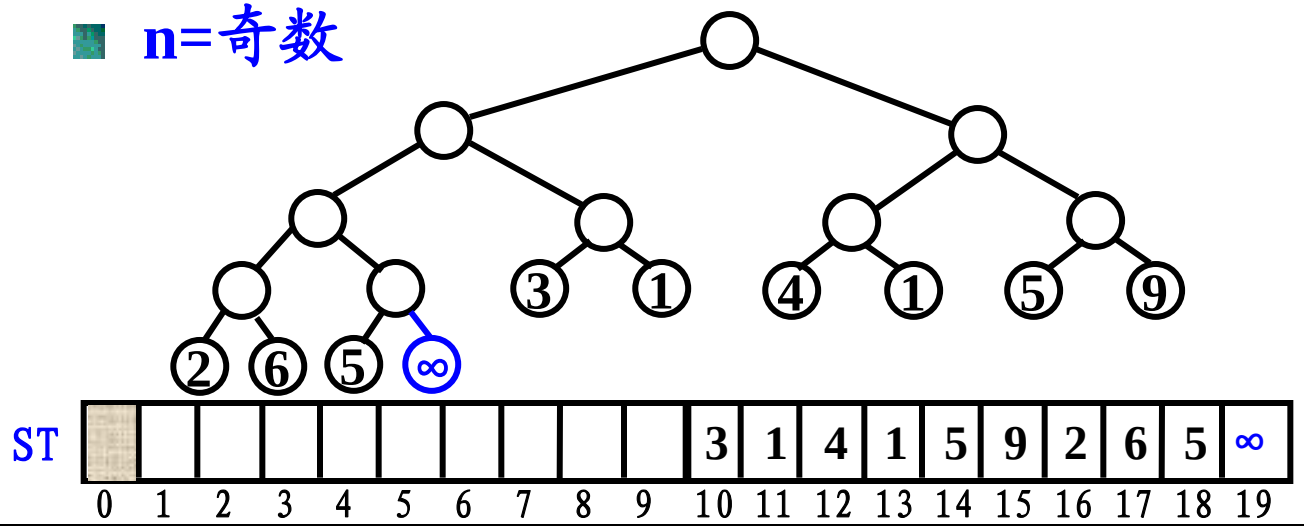
```
struct node{
    keytype  key;  /*排序关键字*/
    int  id;   /*关键字在胜者树中的下标*/
    fields  other ;
};
typedef  node  TREE[maxsize];
```

6.5 锦标赛排序(cont.)

- (1) 初始化选择树--无需补全至2^k个记录，最多补一个即可
  - n=偶数



  - n=奇数



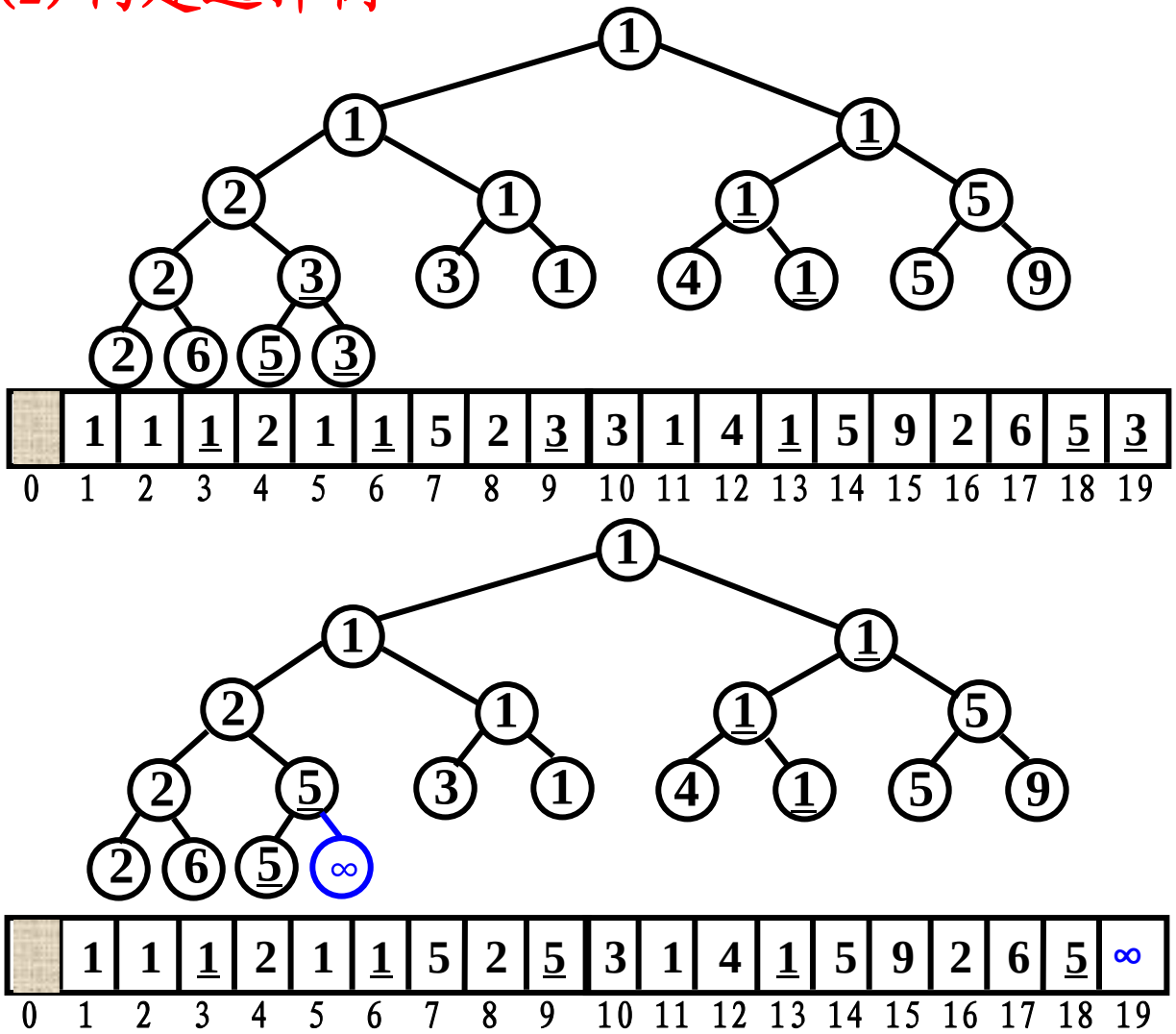
6.5 锦标赛排序(cont.)

- 初始化选择树的实现

```
int Initial( LIST A, int n, TREE ST[] )
 /* 用待排序列初始化选择树，确定并返回其长度 */
{
    if ( n%2==0){/*将待排序列依次存放在ST[n]至ST[2n-1]单元*/
        for( int i= n; i<=2*n-1; i++ ){
            ST[i].key = A[i-n+1].key;
            ST[i].id = i;
        }
    }
    else{ /*将待排序列依次存放在ST[n+1]至ST[2n]单元*/
        for( int i=n+1; i<=2*n; i++ ){
            ST[i].key = A[i-n].key;
            ST[i].id = i;
        }
        ST[2*n+1].key =∞; /*在选择树中增加一个虚拟结点*/
    }
}/* Initial */
```

6.5 锦标赛排序(cont.)

- (2) 构建选择树



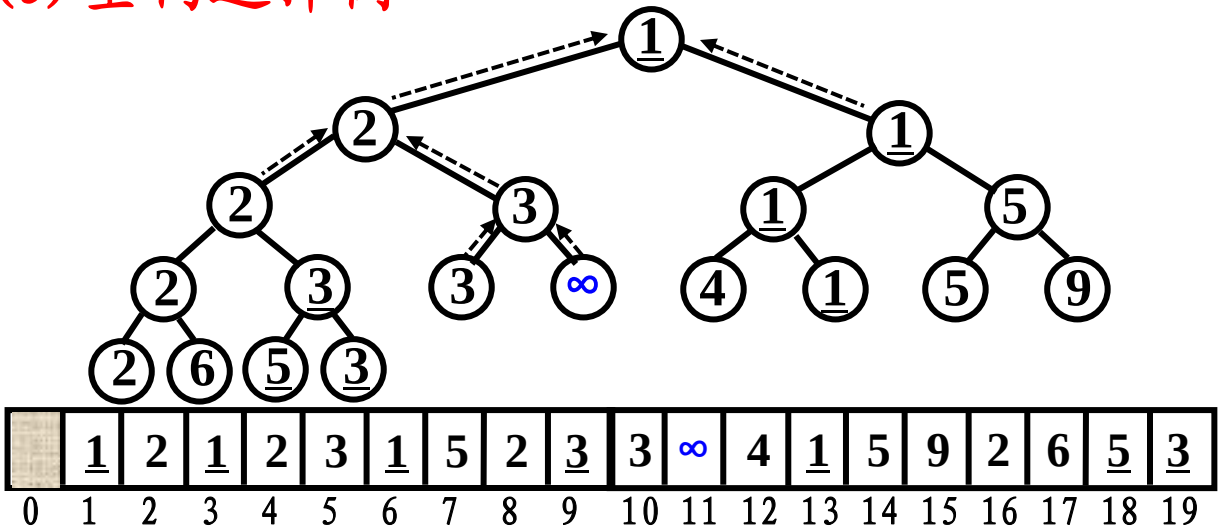
6.5 锦标赛排序(cont.)

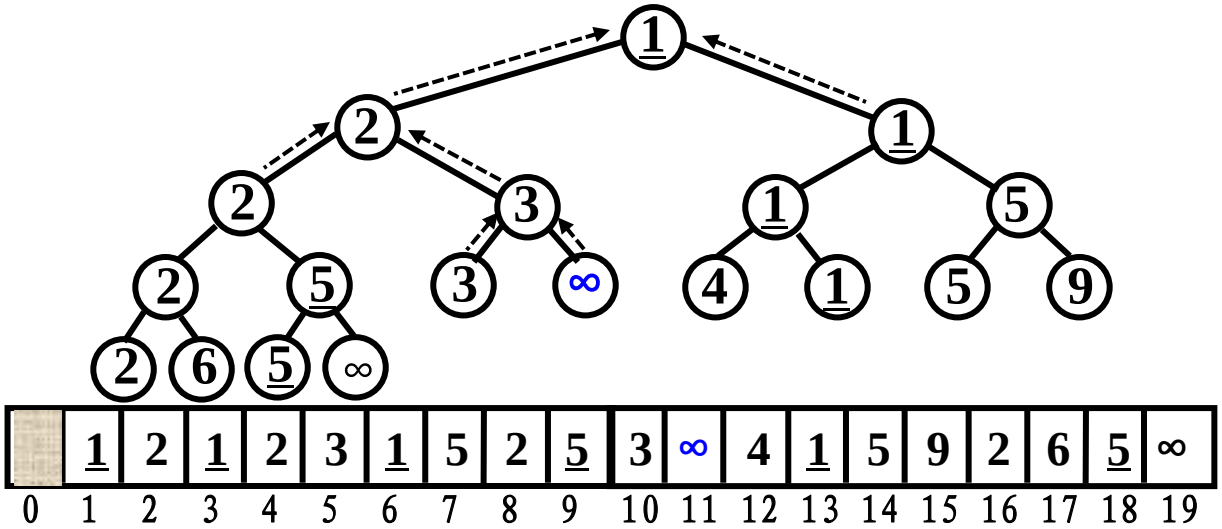
- 构建选择树的实现

```
int Build(TREE ST[], int treeLen) /*构建选择树，返回最小者下标*/
{
   for ( int i=treeLen; i!=1; i-=2 ){ /*从选择树的右到左，依次俩俩比较*/
      if (ST[i].key < ST[i-].key){
         ST[i/2].key = ST[i].key;     /*记录胜者到父结点*/
         ST[i/2].id = ST[i].id;       /*在父结点中记录胜者下标*/
      }
      else{                           /*若相等时，选左兄弟*/
         ST[i/2].key = ST[i-1].key;
         ST[i/2].id = ST[i-1].id;
      }
   }
   return ST[1].id;
}/* Build */
```

6.5 锦标赛排序(cont.)

- (3) 重构选择树





6.5 锦标赛排序(cont.)

- 重构选择树的实现

```
int Rebuild(TREE ST[], int minId) /* 重构选择树，返回新的最小者下标 */
{
   for(int i=minId; i!=1 ; i/=2){
      if( i%2==1 ){ /*若i是奇数,说明兄弟是i-1,父结点是i/2*/
         if(ST[i].key < ST[i-1].key){ /*i-1为兄弟结点*/
            ST[i/2].key = ST[i].key;
            ST[i/2].id = ST[i].id;
         }
         else{
            ST[i/2].key = ST[i-1].key;
            ST[i/2].id = ST[i-1].id;
         }
      }
      else{ /*若i是偶数,说明兄弟是i+1,父结点是i/2*/
         if(ST[i].key <= ST[i+1].key){ /*i+1为兄弟结点*/
            ST[i/2].key = ST[i].key;
            ST[i/2].id = ST[i].id;
         }
         else
         {
            ST[i/2].key = ST[i+1].key;
            ST[i/2].id = ST[i+1].id;
         }
      }
   }
   return ST[1].id;
}/* Rebuild */
```

6.5 锦标赛排序(cont.)

锦标赛排序算法实现

```
void TourSort( int n, LIST A )          /*锦标赛排序*/
{    int minId;                         /*最小者在选择树中的下标*/
     TREE ST[ maxsize ];                /*选择树作为排序的辅助存储空间*/
     int treeLen = Initial( A, n, ST );  /*初始化选择树*/
     minId = Build( ST, treeLen );      /*构建选择树，返回最小者下标*/
     A[1].key= ST[minId].key;           /*选出最小者*/
     ST[minId].key = ∞;                 /*将最小者单元置为∞*/
     for( int i =2; i <=n; i++ ){       /*n-1次重构选择树，依次选出较小者*/
          minId = Rebuild(ST, minId);   /*重构选择树，返回当前最小者下标*/
          A[i].key = ST[minId].key;     /*选出当前最小者*/
          ST[minId].key = ∞;            /*将当前最小者单元置为∞*/
     }
}/* TourSort */
```

6.5 锦标赛排序

- 性能分析
  - 时间性能： $O(n\log_2 n)$
    - 初始化：2n-1次赋值
    - 构建：n-1次比较
    - 重构：n-1遍重构，每遍重构$\lfloor\log_2 n\rfloor+1$比较
  - 空间需求：$O(n)$
    - n-1个辅助存储单元存放中间比较结果
  - 稳定性：稳定的排序
    - 在兄弟结点关键字比较时，若两个关键字相同，选取左兄弟结点的关键字

6.5 堆排序

- 堆排序是对直接选择排序的改进，改进的着眼点：
  - 如何减少关键字之间的比较次数？
    - 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

↓

选择树（锦标赛）排序 → 堆排序

  - 如何降低锦标赛排序的空间复杂度？

6.5 堆排序(cont.)

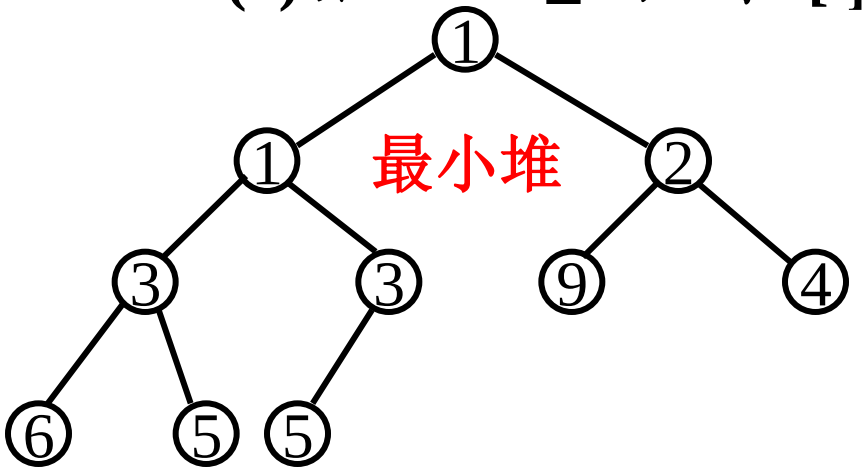
堆的定义

把具有如下性质的数组A表示的完全二叉树称为（最小）堆:

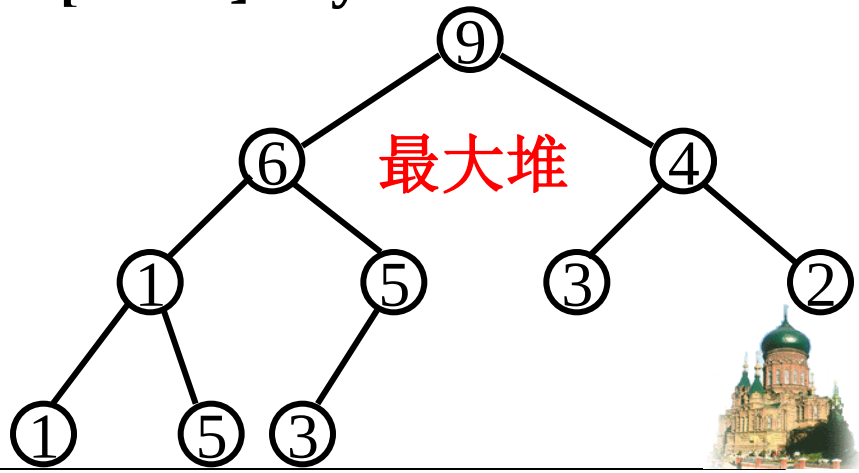
- (1) 若2*i*≤*n*，则A[*i*].key≤A[2*i*].key；
- (2) 若2*i*+1≤*n*，则A[*i*].key≤A[2*i*+1].key。

把具有如下性质的数组A表示的完全二叉树称为（最大）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。



最小堆



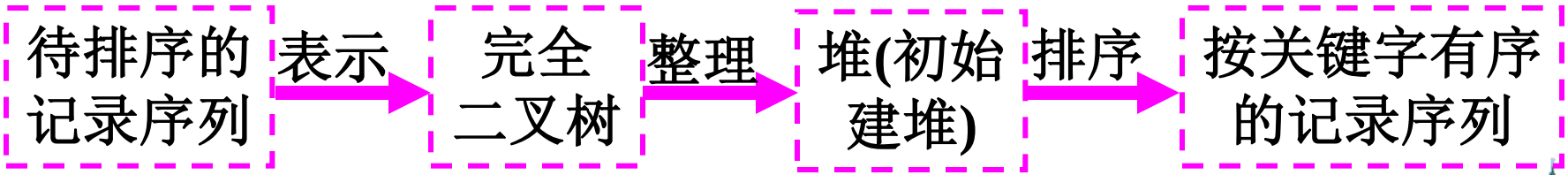
最大堆

6.5 堆排序(cont.)

- 堆的性质
  - 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$A[i/2].key \le A[i].key \quad 1 \le i/2 < i \le n$。
  - 在堆中，以任意结点为根的子树仍然是堆。特别地，每个叶也点也可视为堆。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。去掉堆中编号最大的叶结点后，仍然是堆。
    - 以堆的规模逐渐缩小的方式进行堆排序。
- 堆的其他应用
  - 优先级队列
  - TOP K问题
  - STL partial_sort(fisrt,mid,last)

6.5 堆排序(cont.)

- 堆排序的基本思想:
  - 首先将待排序的记录序列用完全二叉树表示;
  - 然后完全二叉树构造成一个堆，此时，选出了堆中所有记录关键字的最小者;
  - 最后将关键字最小者从堆中移走，并将剩余的记录再调整成堆，这样又找出了次小的关键字记录，以此类推，直到堆中只有一个记录。



6.5 堆排序(cont.)

- 堆排序的实现步骤:
  - 1.把待排序的记录序列用完全二叉树数组存储结构A表示
  - 2.初始建堆：把数组所对应的完全二叉树以堆不断扩大的方式整理成堆。令 $i= n/2,\ldots,2,1$并分别把以$n/2 ,\ldots,2,1$ 为根的完全二叉树整理成堆，即执行算法PushDown (A, i , n)
  - 3.堆排序(堆的重构)：令i = n, n-1 ,…, 2
    - (1)交换：把堆顶元素(当前最小的)与位置 i(当前最大的叶结点下标)的元素交换，即执行swap(A[1],A[i]);
    - (2)整理:把剩余的 i-1个元素整理成堆，即执行PushDown(A, 1 , i-1);
    - (3)重复执行完这一过程之后，则A[1],A[2],…,A[n]是按关键字不增顺序的有序序列。

6.5 堆排序(cont.)

- 堆排序的实现:

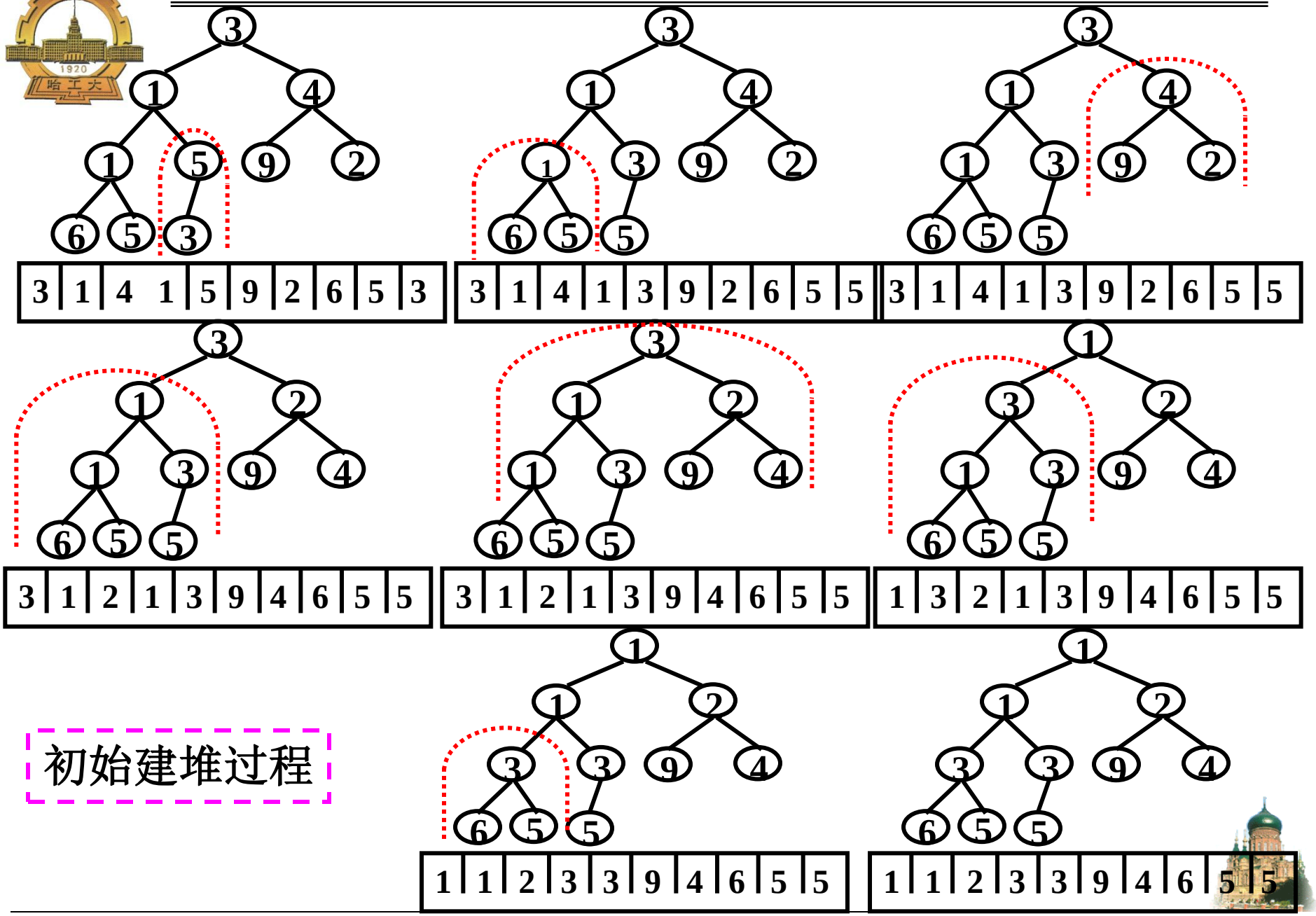
```
void HeapSort ( int n, LIST A )
{   int i;
    for( i=n/2; i>=1; i--)  /*初始建堆，从最右非叶结点开始*/
        PushDown(A, i, n);/*整理堆，把以i为根，最大下标的叶为n*/
    for( i=n; i>=2; i--) {
        swap(A[1],A[i]); //堆顶与当前堆中的下标最大的叶结点交换
        PushDown(A, 1, i-1 );
      /*整理堆把以1为根，最大叶下标为i-1的完全二元树整理成堆*/
    }
}/* HeapSort */
```

6.5 堆排序(cont.)

- 整理堆算法：PushDown(A, first, last)
  - 把以A[first]为根，以A[last]为最右边叶结点的完全二叉树整理成堆。根据堆的定义，它要完成的功能是，把完全二元树中的关键字最小的元素放到堆顶，而把原堆顶元素下推到适当的位置，使(A[first],…,A[last])成为堆。
  - 那么，怎样把关键字最小的元素放到堆顶，把堆顶元素下推到适当位置呢?
  - 具体操作(要点)如下:
    - 把完全二元树的根或者子树的根与其左、右儿子比较，如果它比其左／右儿子大，则与其中较小者交换（若左、右儿子相等，则与其左儿子交换）。重复上述过程，直到以A[first]为根的完全二元树是堆为止。
  - PushDown(A, first, last)算法实现如下:

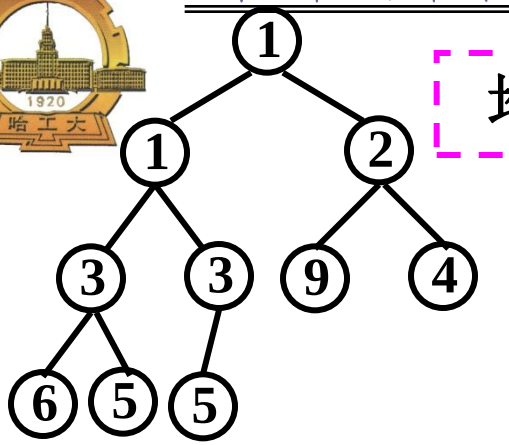
6.5 堆排序(cont.)

```
void PushDown(LIST A, int first, int last)
{     /*整理堆:把A[first]下推到完全二元树的适当位置*/
      int r=first; /* r是被下推到的适当位置，初始值为根first*/
      while(r<=last/2) /* A[r]不是叶，否则是堆 */
          if((r==last/2) && (last%2==0)) {/* r有一个儿子在2*r上且为左儿子*/
              if(A[r].key>A[2*r].key)
                  swap(A[r],A[2*r]);/*下推*/
              r=last;  /*A[r].key小于等于A[2*r].key或者"大于"，交换后到叶，循环结束*/
          } else if((A[r].key>A[2*r].key)&&(A[2*r].key<=A[2*r+1].key)) {
              /*根大于左儿子，且左儿子小于或等于右儿子*/
              swap(A[r],A[2*r]);       /*与左儿子交换*/
              r=2*r;                   /*下推到的位置也是下次考虑的根*/
          } else if((A[r].key>A[2*r+1].key)&&(A[2*r+1].key<A[2*r].key)) {
              /*根大于右儿子，且右儿子小于左儿子*/
              swap(A[r],A[2*r+1]);     /*与右儿子交换*/
              r=2*r+1;                 /*下推到的位置也是下次考虑的根*/
          }else /*A[r]符合堆的定义，不必整理，循环结束*/
              r=last;
}/*PushDown*/
```

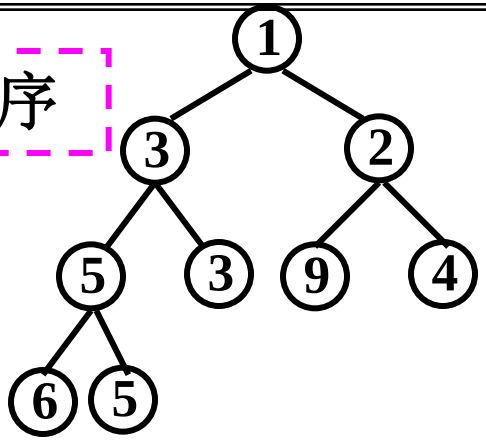

3 1 4 1 5 9 2 6 5 3
3 1 4 1 3 9 2 6 5 5
3 1 4 1 3 9 2 6 5 5
3 1 2 1 3 9 4 6 5 5
3 1 2 1 3 9 4 6 5 5
1 3 2 1 3 9 4 6 5 5
1 1 2 3 3 9 4 6 5 5
1 1 2 3 3 9 4 6 5 5

初始建堆过程

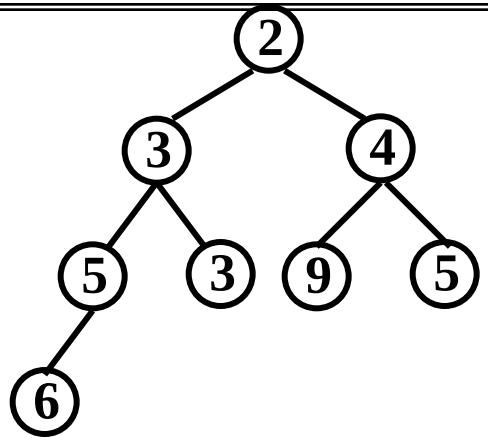
堆排序



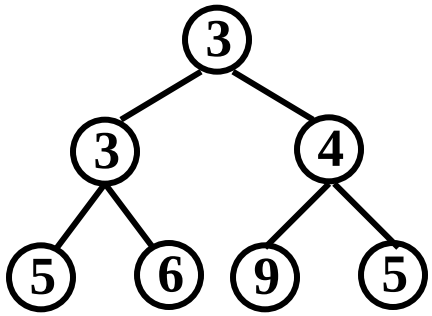
1	1	2	3	3	9	4	6	5	5



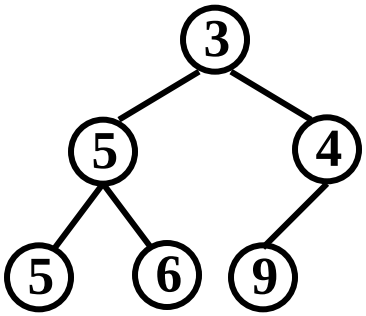
1	3	2	5	3	9	4	6	5	1



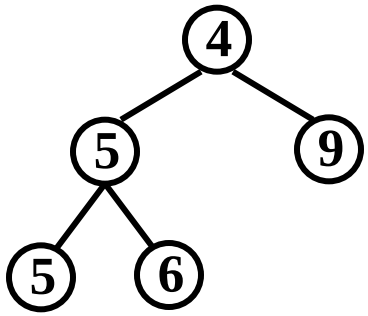
2	3	4	5	3	9	5	6	1	1



3	3	4	5	6	9	5	2	1	1

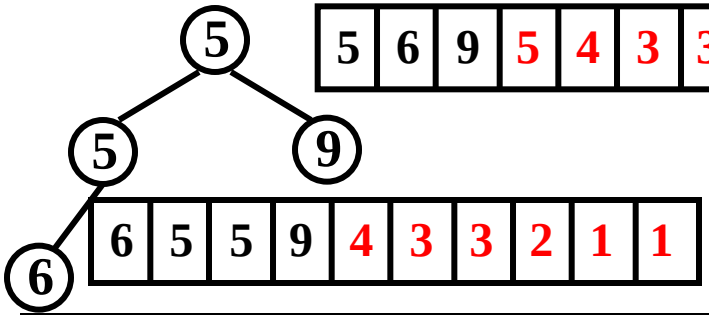


3	5	4	5	6	9	3	2	1	1

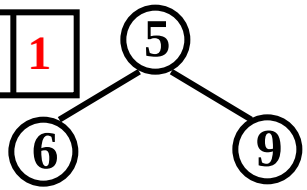


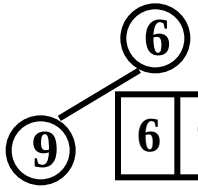
4	5	9	5	6	3	3	2	1	1

5	6	9	5	4	3	3	2	1	1



6	5	5	9	4	3	3	2	1	1





6	9	5	5	4	3	3	2	1	1

9	6	5	5	4	3	3	2	1	1

6.5 堆排序(cont.)

性能分析（粗）

- PushDown函数中，执行一次while循环的时间是常数。因为r 每次至少为原来的两倍，假设while循环执行次数为 i ，则当r 从first 变为first*2^i 时循环结束。此时r=first*2^i >last/2， 即 i > $\log_2$(last/first)-1。所以while循环体最多执行$\log_2$(last/first)次，即PushDown时间复杂度 O(log(last/first))=O($\log_2 n$)。
- 所以，HeapSort时间复杂度： $O(n \log_2 n)$ 。
- 这是堆排序的最好、最坏和平均的时间代价。
- HeapSort空间复杂度： O(1)。

6.5 堆排序(cont.)

- 性能分析----时间性能
  - PushDown函数的执行时间：
    - 设n个结点的完全二叉树的高度为h，则有$2^{h-1} \le n < 2^h$，即$h=\lfloor \log_2 n \rfloor + 1$。
    - PushDown函数在每次while循环时，左、右儿子先比较一次，然后左、右儿子的较小者与其父结点比较一次，
    - 所以总的比较次数不会超过$2(h-1)$次。

6.5 堆排序(cont.)

- 性能分析----时间性能：$O(n\log_2 n)$
  - 初始建堆的执行时间：$O(n)$----线性时间！
    - 在初始建堆过程中，在第i层（$1 \le i \le h-1$）至多有2^{i-1}个记录，每个记录下推时的比较次数至多为$2(h-i)$次
    - 因此，总的比较次数不会超过：

$$T_1 = \sum_{i=1}^{h-1} 2^{i-1} \times 2 \times (h-i) = \sum_{i=1}^{h-1} 2^i \times (h-i)$$

- 令$j=h-i$，则有 $T_1 = \sum_{j=1}^{h-1} 2^{h-j} \times j = 2 \times 2^{h-1} \sum_{j=1}^{h-1} \frac{j}{2^j}$
- 在令 $S = \sum_{j=1}^{h-1} \frac{j}{2^j} = \frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \ldots + \frac{h-1}{2^{h-1}}$
- 则有 $\frac{1}{2} \times S = \frac{1}{2} \sum_{j=1}^{h-1} \frac{j}{2^j} = \frac{1}{4} + \frac{2}{8} + \frac{3}{16} + \ldots + \frac{h-2}{2^{h-1}} + \frac{h-1}{2^{h+1}}$
- 于是 $\frac{1}{2} \times S = S - \frac{1}{2} \times S = \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots + \frac{1}{2^{h-1}} - \frac{h-1}{2^{h+1}}$

$$= \sum_{j=1}^{h-1} \frac{1}{2^j} - \frac{h-1}{2^{h+1}} \le \sum_{j=1}^{h-1} \frac{1}{2^j} < 1$$

- 即 $T_1 = 2 \times 2^{h-1} \times S < 2 \times n \times 2 = 4n$

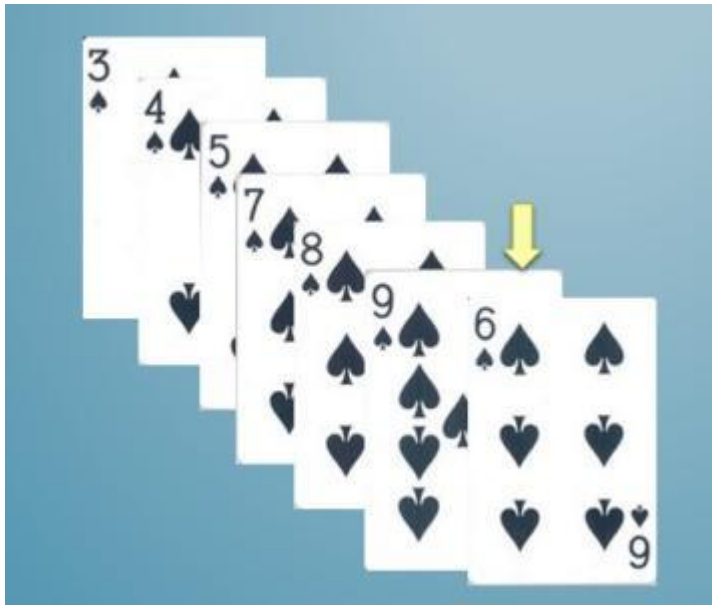
6.5 堆排序(cont.)

- 性能分析----时间性能
  - 堆的重构执行时间： $O(n\log_2 n)$
    - 需要调用PushDown算法n-1次
    - 这n-1次调用PushDown算法的比较次数分别不超过：$2\lfloor\log_2(n-1)\rfloor$，$2\lfloor\log_2(n-2)\rfloor$，…，$2\lfloor\log_2 2\rfloor$次，
    - 即总的比较次数不超过：

      $2(\lfloor\log_2(n-1)\rfloor+\lfloor\log_2(n-2)\rfloor+\ldots+\lfloor\log_2 2\rfloor) < 2n\lfloor\log_2 n\rfloor$
  - $O(n\log_2 n)$是堆排序的最好、最坏和平均的时间代价。
    - 性能分析----空间性能： $O(1)$
    - 稳定性： 不稳定的排序，如2，2，1

6.6 直接插入排序

- 基本思想：
  - 类似于整理手中的纸牌
  - 其基本思想是：每次将一个待排序的记录按其关键字的大小插入到一个已经排好序的有序序列中，直到全部记录排好序为止。
  - 插入排序的主要操作是插入



6.6 （直接）插入排序

- 实现步骤：
  - 初始，令第 1 个记录作为初始有序表；
  - 依次插入第 2 , 3 , …, i 个记录构造新的有序表；
  - 直至最后一个记录；
- 示例：序列　3　1　4　1　5　9　2　6　5　3

  ↑　↑　↑　↑　↑　↑　↑　↑　↑

  - 初始，S ={ 3 }

    { 1　3 }

    { 1　3　4 }

    { 1　1　3　4 }

    { 1　1　3　4　5 }

    { 1　1　3　4　5　9 }

    { 1　1　2　3　4　5　9 }

    { 1　1　2　3　4　5　6　9 }

    { 1　1　2　3　4　5　5　6　9 }

    { 1　1　2　3　3　4　5　5　6　9 }

6.6 （直接）插入排序(cont.)

- 算法的实现

```
void  InsertSort (int n, LIST A )
{   int  i, j ;
    A[0].key = -∞ ;//哨兵,不必检查当前位置j是否为1
    for(i=1; i<=n; i++) {
        j=i;
        while(A[j-1].key>A[j].key){
            swap(A[j-1],A[j]) ;
            j=j-1;
        }
    }
}/* InsertSort */
```

```
j=i;   tmp=A[j];
while (tmp.key<A[j-1].key){
   A[j]=A[j-1];
     j=j-1;
}
A[j]=tmp;
```

6.6 （直接）插入排序(cont.)

- 算法的性能分析
  - 最好情况下（正序）：
    - 比较次数： n-1
    - 移动次数： 0 or 2(n-1)?
    - 时间复杂度为$O(n)$。
  - 最坏情况下（反序）：
    - 比较次数： $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$
    - 移动次数： $\sum_{i=1}^{n} (i\text{-}1) = \frac{(n-1)n}{2}$
    - 时间复杂度为$O(n^2)$。
  - 平均情况下（随机排列）
    - 比较次数： $\sum_{i=2}^{n} i/2 = \frac{n(n+1)}{4}$
    - 移动次数： $\sum_{i=2}^{n} (i\text{-}1)/2 = \frac{(n-1)n}{4}$
    - 时间复杂度为$O(n^2)$。

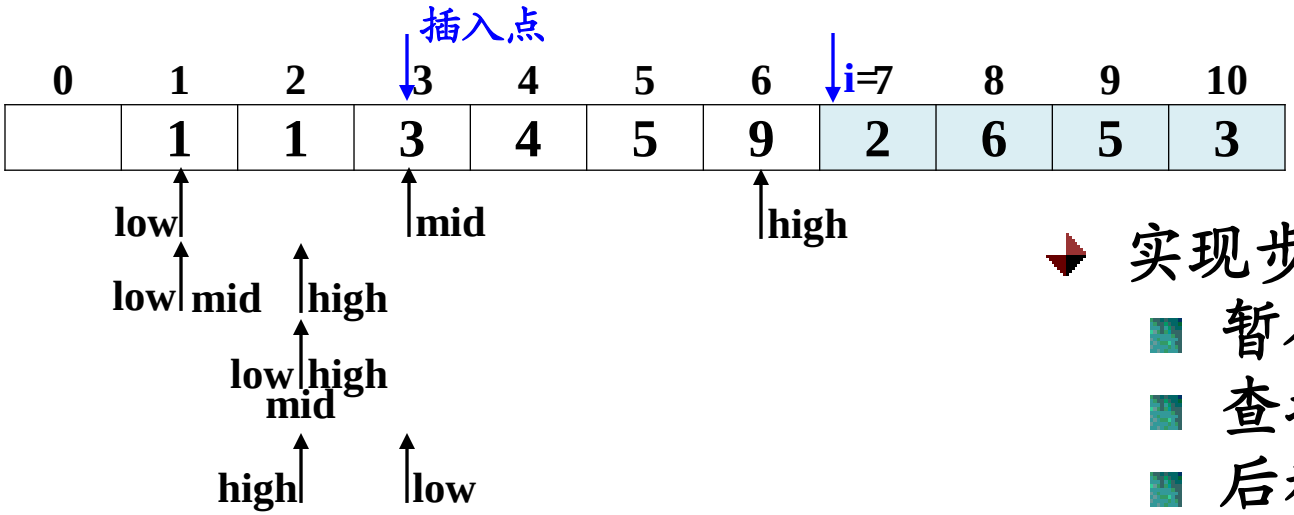
6.6 （直接）插入排序(cont.)

- 算法的性能分析
  - 空间复杂度：O(1)
  - 稳定性：稳定的排序
  - 首先，由于若待排序的序列已经按关键字非递减有序，直接插入排序的时间复杂度可优化至O（n）。
  - 因此，若待排记录序列按关键字基本有序，即待排序列中具有A[i].key<Max{A[j].key | 1≤j<i}性质的记录比较少时，直接插入排序的效率也会大大提高。
  - 其次，由于直接插入排序算法简单，且在n比较小时效率也比较高。
  - 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。

6.6 （直接）插入排序(cont.)

折半插入排序(Binary Insertion Sorting)

- 是对直接插入排序的一种改进
- 直接插入排序，在插入第 i（$i>1$）个记录时，前面的 i-1 个记录已排好序，则在寻找插入点时，可以用折半查找代替顺序查找，加快寻找插入点的速度，从而较少比较次数
- 例如，



- 实现步骤：
  - 暂存待插记录
  - 查找插入位置
  - 后移记录
  - 插入待插记录

6.6 （直接）插入排序(cont.)

折半插入排序算法的实现

```
void BinaryInsertSort(int n LIST A)
{     int i, j, mid, low, high,;
      for ( i = 1; i < =n; i++) {
            A[0] = A[i];                          //将A[i]暂存到A[0]
            low = 1; high = i-1;
            while ( low <= high ) {               //在A[low…high]中查找有序插入点
                  mid = (low +high) / 2;    //折半
                  if(A[mid].key > A[0].key) // 相同时low=mid+1,上区插入,保证稳定性
                        high = mid-1;       //插入点在左半区
                  else
                        low = mid+1;        //插入点在右半区
            }
            for ( j = i-1; j>=high+1; j--)
                  A[j+1] = A[j];                  //记录后移
           A[high+1] = A[0];                      //插入
    }
}/*BinaryInsertSort*/
```

6.6 （直接）插入排序(cont.)

稳定性与复杂度

- 稳定的排序方法：只是把大于的向后移动，等于的未移
- 与直接插入排序相比，明显地较少了关键字的比较次数，因此排序速度比直接插入排序算法快
- 但记录的移动次数并未减少，因此，时间复杂度与直接插入排序算法相同，仍然为$O(n^2)$
- 辅助存储空间开销O(1)

6.7 希尔排序----分组插入排序

- 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,直接插入排序效率高
  - 由于直接插入排序简单,则在记录数量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，使整个序列逐步向基本有序发展；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？
  - 如何分组？按一定间隔分割成不同的字序列
  - 组内如何排序？直接插入排序
    - 通过比较相距一定间隔的元素，每趟比较所用的步长随着算法的进行而减小，直到只比较相邻元素为止。由于增量一直在减小，因此该排序也称作缩减增量排序。

6.7 希尔排序----分组插入排序(cont.)

- 示例：缩减增量（步长）排序

6.7 希尔排序----分组插入排序(cont.)

- 算法的实现

```
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];
       }
    }
} /*ShellSort*/
```

6.7 希尔排序----分组插入排序(cont.)

算法的性能分析

- 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。
- 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）。
- 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。已知的最好步长序列是由Sedgewick提出的(1, 5, 19, 41, 109,...)
- 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$O(n^{1.3})$。

6.8 （二路）归并排序

- 归并排序
  - 归并：将两个或两个以上的有序序列合并成一个有序序列的过程。
  - 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。
- 如何将两个有序序列合成一个有序序列？（二路归并基础）
  - 设相邻的按关键字有序序列为A[s] ~ A[m]和A[m+1] ~ A[t]，归并成一个按关键字有序序列B[s] ~ B[t]

6.8 （二路）归并排序(cont.)

```
void Merge (int s , int m , int t , LIST  A , LIST  B)
/*将有序序列A[s],…,A[m]和A[m+1],…,A[t]合并为一个有序序列 B[s],…,B[t]*/
{   int i = s ;  j = m+1 , k = s ;//置初值
      /* 两个序列非空时，取小者输出到B[k]上 */
    while ( i <= m && j <= t )
        B[k++] = ( A[ i ].key <= A[ j ].key) ? A[i++] : A[j++] ;
      /* 若第一个子序列非空(未处理完)，则复制剩余部分到B */
    while ( i <= m )    B[k++] = A[i++] ;
      /* 若第二个子序列非空(未处理完)，则复制剩余部分到B */
    while ( j <= t )    B[k++] = A[j++] ;
}/*时间复杂度：O( t − s + 1 );     空间复杂度：O( t − s + 1 ) */
```

6.8 （二路）归并排序(cont.)

- 二路归并排序的基本思想（自底向上的非递归算法）
  - 将具有n个待排序记录的序列视为n个长度为1的有序序列；
  - 然后进行两两归并，得到$\lceil n/2 \rceil$个长度为2的有序序列；
  - 再进行两两归并，得到$\lceil n/4 \rceil$个长度为4的有序序列；
  - ……；
  - 直至得到1个长度为n的有序序列为止。
  - 共需归并$\log_2 n$趟

{26} {5} {77} {1} {61} {11} {59} {15} {48} {19}

{ 5 26 } { 1 77 } { 11 61 } { 15 59 } { 19 48 } #

{ 1 5 26 77 } { 11 15 59 61 } { 19 48 } #

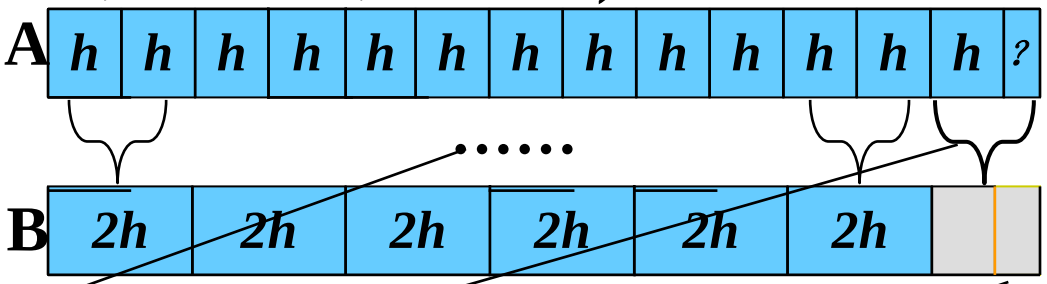
{ 1 5 11 15 26 59 61 77 } { 19 48 } *

{ 1 5 11 15 19 26 48 59 61 77 }

6.8 （二路）归并排序(cont.)

- 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] ;
} /* MergePass */
```



6.8 （二路）归并排序(cont.)

- （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 */
```

开始时,有序序列的长度h=1;结束时,有序序列的长度h=n,用有序序列的长度来控制排序的结束.

6.8 （二路）归并排序(cont.)

- （二路）归并排序算法性能分析
  - 时间性能:
    - 一趟归并操作是将A[1]~A[n]中相邻的长度为*h*的有序序列进行两两归并，并把结果存放到B[1]~B[n]中，这需要$O(n)$时间。整个归并排序需要进行$\lceil \log_2 n \rceil$趟，因此，总的时间代价是$O(n\log_2 n)$。
    - 这是归并排序算法的最好、最坏、平均的时间性能。
  - 空间性能:
    - 算法在执行时，需要占用与原始记录序列同样数量的存储空间，因此，空间复杂度为$O(n)$。

6.8 （二路）归并排序(cont.)

- （二路）归并排序分治算法
  - 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
    - 子序列长度为 1，
      因为一个记录自然有序。
  - 算法实现

6.8 （二路）归并排序(cont.)

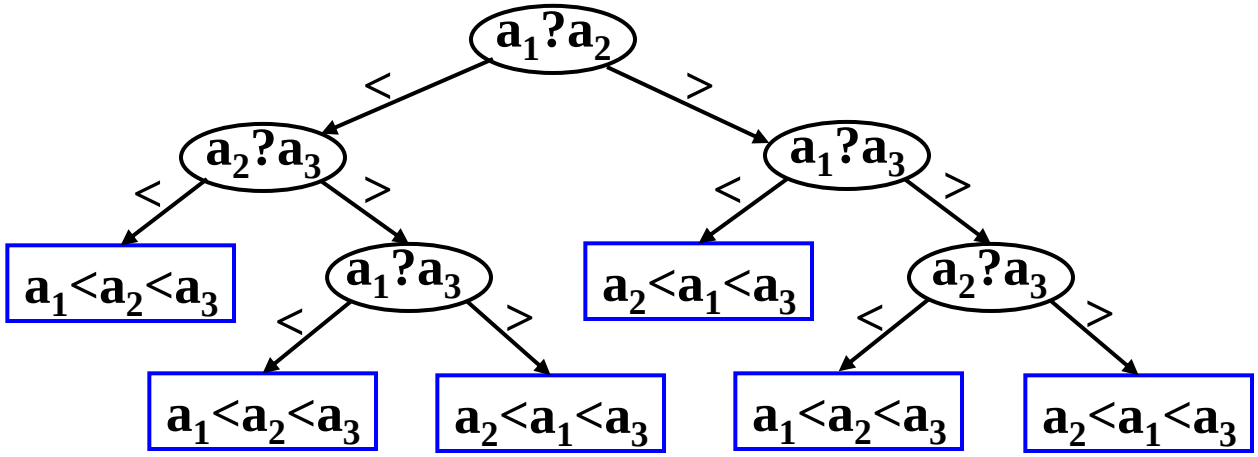
- （二路）归并排序分治算法
  - 算法实现

```
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 , high , A , B) ;
    }
}/* MergeSort */
```

6.9 基数排序----多关键字排序

基于比较排序的判定树

- 对n=3个互不相同的数进行比较排序可用如下判定树表示



- 在判定树的某个结点上，a_i与a_j进行比较时，
  - 若$a_i < a_j$，则转向其左子树；
  - 若$a_i > a_j$，则转向其右子树；
- 叶子结点表示（所有可能的）排序结果

6.9 基数排序----多关键字排序

- 高为h的二叉树至多有2^{h-1}个叶子结点
  - 即叶子结点数$N \le 2^{h-1}$，亦即$h \ge \log_2 N + 1$
- 对n个互不相同的数进行比较排序的判定树高度至少为$\log_2(n!)$
  - 由于n个不同的数有$n!$个排列，所以对n个不同的数的比较排序，就有$n!$个可能的输入
  - 而对这n个不同的数的排序结果可以是$n!$个排列中的任意一个，因此对n个不同数进行排序的判定树中必有$n!$个叶子
  - 所以判定树的高度至少为$\log_2(n!)$
- 对n个数进行比较排序，排序算法的时间下界为$\Omega(n\log_2 n)$
  - 对于$n>1$，$n! \ge n(n-1)\ldots n/2 \ge (n/2)^{n/2}$
  - 所以，取$n_0=4$，C=1/4，当$n \ge n_0$时，有
    - $\log_2(n!) \ge (n/2)\log_2(n/2) \ge (n/4)\log_2 n$
  - 即，$\log_2(n!) = \Omega(n\log_2 n)$

6.9 基数排序----多关键字排序

- 基于关键字比较的排序方法时间下界是$\Omega(n\log_2 n)$
  - 因此，不存在时间复杂度低于此下界的基于比较的排序！
  - 要突破此下界，不能再基于比较
- 基数排序（时间复杂度可达到线性级$O(n)$）
  - 不比较关键字的大小，而根据构成关键字的每个分量的取值，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，因此基数排序又称为桶排序。
- 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。

6.9 基数排序----多关键字排序(cont.)

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figure，每个关键字可以各自含有figure个分量，每个分量的值取值范围为0,1,…,9即基数为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。

6.9 基数排序----多关键字排序(cont.)

- 算法示例:

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

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

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

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

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

6.9 基数排序----多关键字排序(cont.)

- 算法实现:

```
void   RadixSort( int figure, QUEUE &A)
{   QUEUE Q[10]; records  data ;
    int  pass, r, i ;
    for ( pass=1; pass<=figure ; pass++ ){
      for ( i=0 ; i<=9 ; i++ )  /*置空队列*/
            MAKENULL( Q[i] ) ;
      while ( !EMPTY( A ) ){/* 分配 */
          data = FRONT ( A ) ;
          DEQUEUE ( A );
          r = Radix(data.key, pass) ;
          ENQUEUE( data , Q[r] ) ; }
      for ( i=0 ; i <=9 ; i++ ) /* 收集 */
          while ( !EMPTY( Q[i] ) ) {
              data = FRONT ( Q[i] ) ;
              DEQUEUE( Q[i] ) ;
              ENQUEUE( data, A );}
    }
}
```

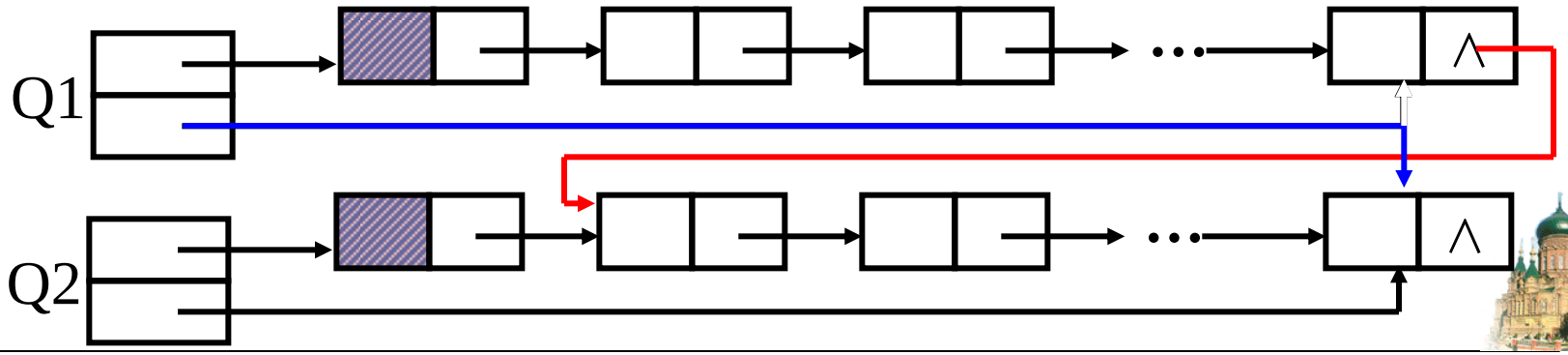
```
/ *求整数 k 的第 p 位 */
int  Radix ( int  k, int   p)
{  int  power= 1 ;
   for ( int i=1; i<=p-1 ; i++ )
        power = power * 10 ;
   return
(( k%(power*10))/power) ;
}
```

```
for (i=1;i<=9;i++) { //收集
    Concatenate(Q[0], Q[i]);
     A=Q[0];
}//缩短收集操作的时间:O(r)
```

6.9 基数排序----多关键字排序(cont.)

- 算法的改进:
  - 由于每个桶（箱）存放多少个关键字分量相同的记录个数无法预料，即队列Q[0]，Q[1],…,Q[9]长度很难确定，故桶一般设计成链式排队，两个排队链在一起的方法如下:

```
void Concatenate(QUEUE Q1, QUEUE Q2)
{   if ( !EMPTY( Q2 ) ) {
        Q1.rear->next=Q2.front->next;
        Q1.rear=Q2.rear;
    }
}
```



6.9 基数排序----多关键字排序(cont.)

- 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。
  - 若关键字各分量不是整数，则把各分量所有可以取值与一组自然数对应。
- 举例:
  - 如何在O(n)时间内，对0到n^2-1之间的n个整数进行排序

本章小结--各种排序方法的比较

- 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。

本章小结--各种排序方法的比较(cont.)

- 复杂度与稳定性比较：

类别	排序方法	时间复杂度			空间复杂度	稳定性
		平均情况	最好情况	最坏情况	辅助空间	
交换排序	冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	快速排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n^2)$	$O(\log_2 n)$	不稳定
选择排序	直接选择	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
	锦标赛	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n)$	稳定
	堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	不稳定
插入排序	直接插入	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	折半插入	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	希尔排序			$O(n^2)$	$O(1)$	不稳定
归并排序		$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n)$	稳定
基数排序		$O(d(n+r))$	$O(d(n+r))$	$O(d(n+r))$	$O(n+r)$	稳定

注：n----关键字个数，d----关键字分量个数，r----关键字的基数

本章小结--各种排序方法的比较(cont.)

- **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、直接选择排序和冒泡排序；
  - 另一类是改进后的算法，包括希尔排序、选择树/堆排序、快速排序和归并排序，这些算法都很复杂。
  -
- **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。

本章小结--各种排序方法的比较(cont.)

- 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。

排序方法	最好情况	最坏情况	平均情况
直接插入排序	$O(n)$	$O(n^2)$	$O(n^2)$
冒泡排序	0	$O(n^2)$	$O(n^2)$
直接选择排序	0	$O(n)$	$O(n)$

本章小结--各种排序方法的比较(cont.)

- 关键字值的分布情况比较:

  当待排序记录按关键字的值有序时，

  - 插入排序和冒泡排序能达到$O(n)$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。