数据结构与算法
排序

臧天仪 教授

tianyi.zang@hit.edu.cn

哈尔滨工业大学计算学部

学习目标

- 掌握：排序的基本概念和常用术语
- 熟练掌握：重要排序算法的基本思想、算法原理、排序过程和算法实现，包括：
  - 冒泡排序、快速排序；
  - 选择排序、锦标赛排序、堆排序；
  - 插入排序、希尔排序；
  - 归并排序；
  - 基数排序。
- 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。

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

4.8 拓扑排序算法(cont.)

- 偏序关系：若集合X上的关系R是自反的、反对称的和传递的，则称R是集合X上的偏序关系。
  - 自反性：任意$x \in X$，$(x, x) \in R$
  - 反对称性：任意$x, y \in X$，若$(x, y) \in R$且$(y, x) \in R$，则$x = y$
  - 传递性：任意$x, y, z \in X$，$(x, y) \in R$且$(y, z) \in R$，则$(x, z) \in R$
  - 偏序只对部分元素有关系（部分可比）
- 全序关系：若集合X上的关系R是反对称的、传递的、完全的，则称R是集合X上的全序关系。
  - 全序性：设R是集合X上的偏序关系，如果对每个$x, y \in X$，必有$(x, y) \in R$或$(y, x) \in R$。
  - 完全性本身也包括了自反性
  - 对集合中任意两个元素都有关系
  - 全序关系必是偏序关系

非传递关系

6.1 基本概念

- 排序(Sorting)：
  - 输入：n个记录序列：$\{R_1, R_2, R_3, \ldots, R_n\}$，及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

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

    $K_{p_1} \ge K_{p_2} \ge K_{p_3} \ge \ldots \ge K_{p_n}$(非递增)
  - 实质是确定1，2，…,n的一种排列(permutation) 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 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

  若稳定算法，排序后得：　3　6　8　8　9　15

  若不稳定算法，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

6.1 基本概念(cont.)

排序分类：

- 按照排序对象存放设备，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：排序过程中，数据对象并非完全在内存中的排序。
- 按照排序基本操作是否基于关键字比较，分为：
  - 基于比较：基本操作为关键字比较和记录移动，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：根据组成关键字的分量及其分布特征排序，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包括：
    - 比较：关键字之间比较；
    - 移动：记录从一个位置移动到另一个位置。
  - 辅助存储空间
    - 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外，排序算法执行所需要的额外存储空间。
  - 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
  - 假定采用顺序存储结构：

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

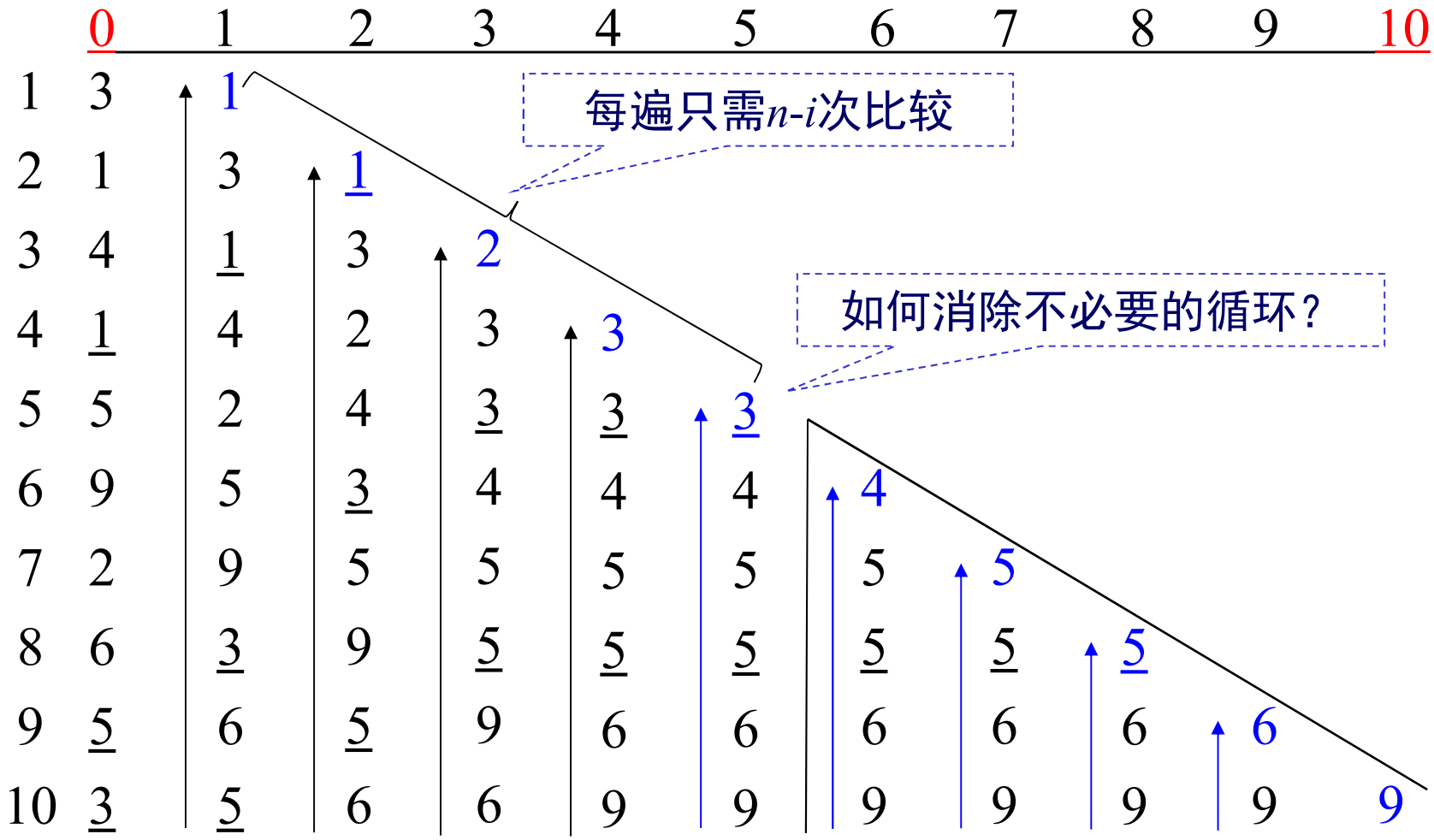
  - Sort(int n , LIST &A)：对n个记录的数组按照关键字不减的顺序进行排序。

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，要上浮。
  - 对这个“气泡”序列进行n-1遍（趟）处理。
    - 所谓1遍（趟）处理，就是自底向上检查1遍该序列，比较相邻关键字顺序是否正确。
    - 如果顺序不对，即 轻记录在下面，则交换它们的位置。
  - 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。在第2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。
  - 第i遍处理时，不必检查第i高位置以上记录的关键字，因为经过前面i-1遍处理，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 */
```

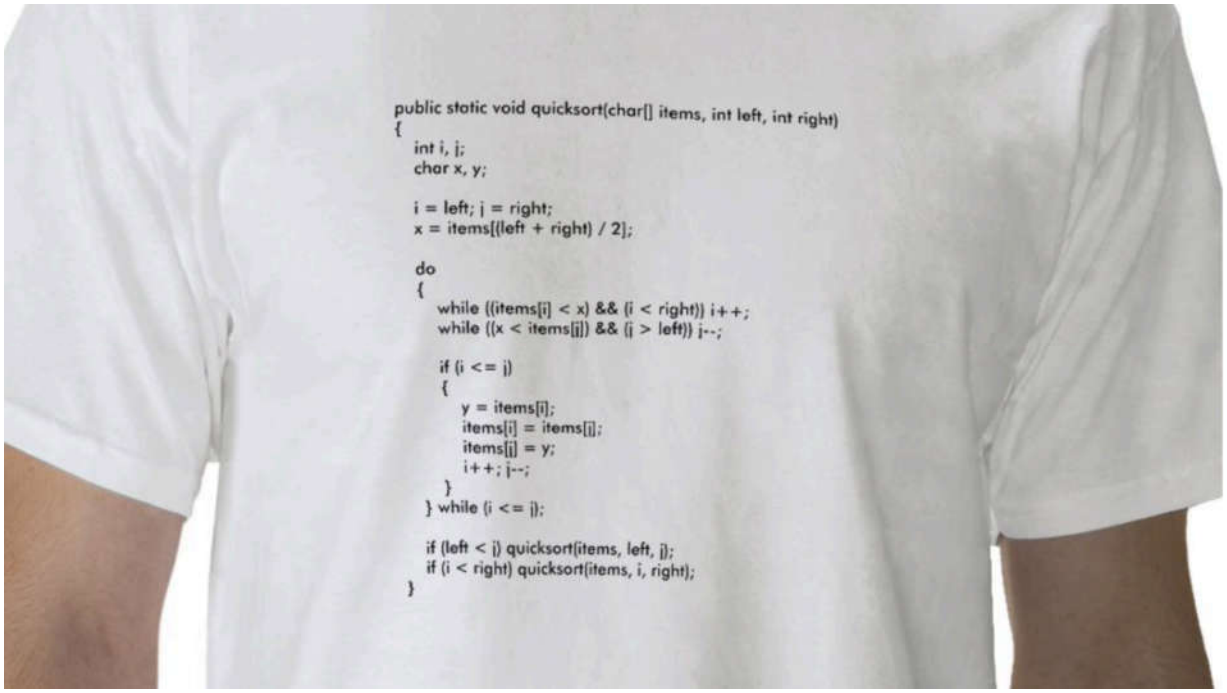
```
void  swap(records &x,  records &y)
{   records   t ;
    t = x ;     x = y ;     y = t ;
}
```

6.2冒泡排序(cont.)

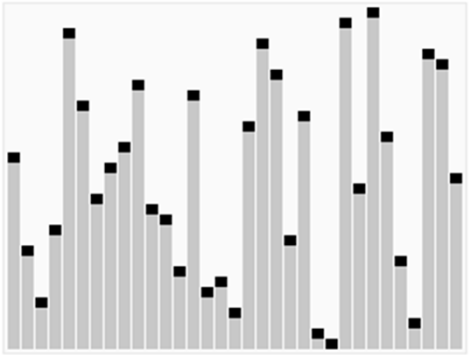
- 算法（时间）性能分析：
  - 最好情况（正序）：
    - 比较次数：n-1
    - 移动次数：0
    - 时间复杂度： $O(n)$;
  - 最坏情况（反序）：
    - 比较次数： $\sum_{i=1}^{n-1}(n-i) = \frac{n(n-1)}{2}$
    - 移动次数： $\sum_{i=1}^{n-1}3(n-i) = \frac{3n(n-1)}{2}$
    - 时间复杂度： $O(n^2)$;
  - 平均情况：时间复杂度为：$O(n^2)$

    空间复杂度： $O(1)$ //辅助存储空间
  - 稳定性：稳定排序
    - 相等时不交换

6.3 快速排序(Quicksort)

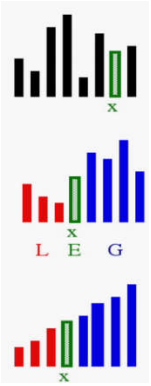


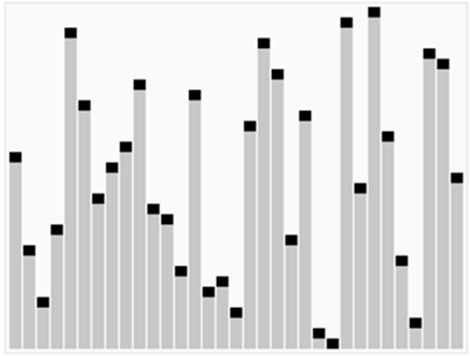
```
public static void quicksort(char[] items, int left, int right)
{
   int i, j;
   char x, y;

   i = left; j = right;
   x = items[(left + right) / 2];

   do
   {
      while ((items[i] < x) && (i < right)) i++;
      while ((x < items[j]) && (j > left)) j--;

      if (i <= j)
      {
         y = items[i];
         items[i] = items[j];
         items[j] = y;
         i++; j--;
      }
   } while (i <= j);

   if (left < j) quicksort(items, left, j);
   if (i < right) quicksort(items, i, right);
}
```

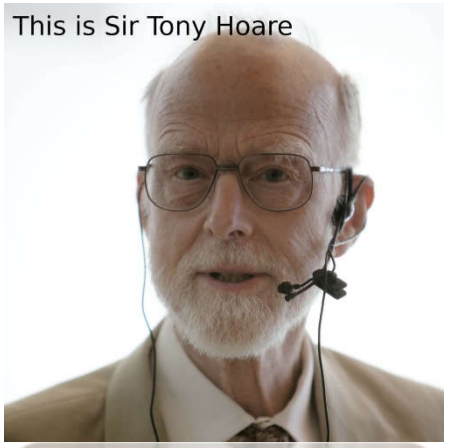


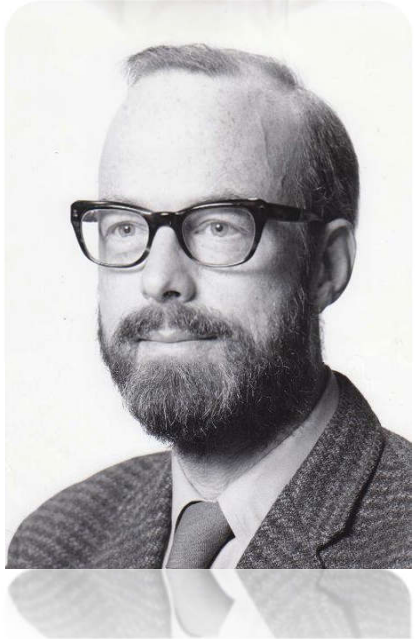
6.3 快速排序(Quicksort)

- Tony Hoare 或 C. A. R. Hoare
  全名：Sir Charles Antony Richard Hoare
- 1934年1月生人，牛津大学荣休教授
  (Emeritus Professor)
- 1980年图灵奖得者
  - CSP(Communicating Sequential Processes), Z
  - Hoare logic
  - Quicksort， 于1960年提出，26岁。
- 一种分治算法(Divide and Conquer)









6.3 快速排序

- 快速排序算法是对冒泡排序的改进：
  - 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。
- 改进之处：
  - 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
  - 每次跳跃式交换（交换距离增大）：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
- 快速排序算法基本策略：
  - 采用分治策略(Divide and Conquer)
  - 是一种划分交换排序，以减少排序过程的比较次数。

6.3 快速排序(cont.)

快速排序算法的基本步骤：

(1)基准元素选取：选择一个数组元素v作为基准元素，以此进行划分；(怎么选择?)

(2)划分：利用基准元素v把无序区A[i],…,A[j]划分成左、右两部分，使得v左边的元素都小于等于v；v右边的y元素都大于等于v；(如何划分?)

$\leqslant v$	v	$\geqslant v$

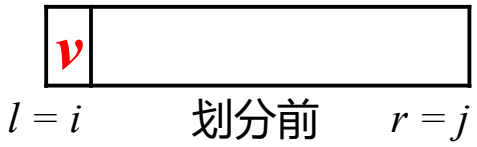
(3)递归求解：重复(1)~(2)，分别对左边和右边部分递归地进行快速排序；

(4)组合：左、右两部分均有序，整个序列有序。

6.3 快速排序(Cont.)

无序区的划分排序：

设待排序的无序区为A[i…j]，v = A[i]



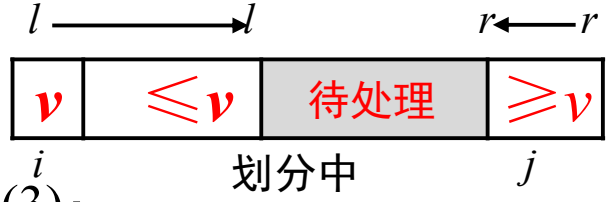
(1) 扫描与移动：

令下标 r 从右（初始时 $r = j$）向左扫描，越过大于等于v 的元素，直到遇到A[r] < v；

若 $l < r$，则A[l] = A[r]。

令下标 l 从左（初始时 $l = i$） 向右扫描，越过小于等于v 的元素，直到遇到A[l] > v；

若 $l < r$，则A[r] = A[l]。

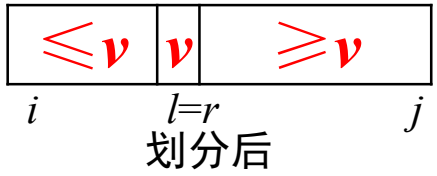


(2) 测试 l 和 r：若$l < r$，则转(1)；否则($l = r$) 转(3)；

(3) 基准定位：A[l] = v（基准元素v定位于其排序后的最终位置）

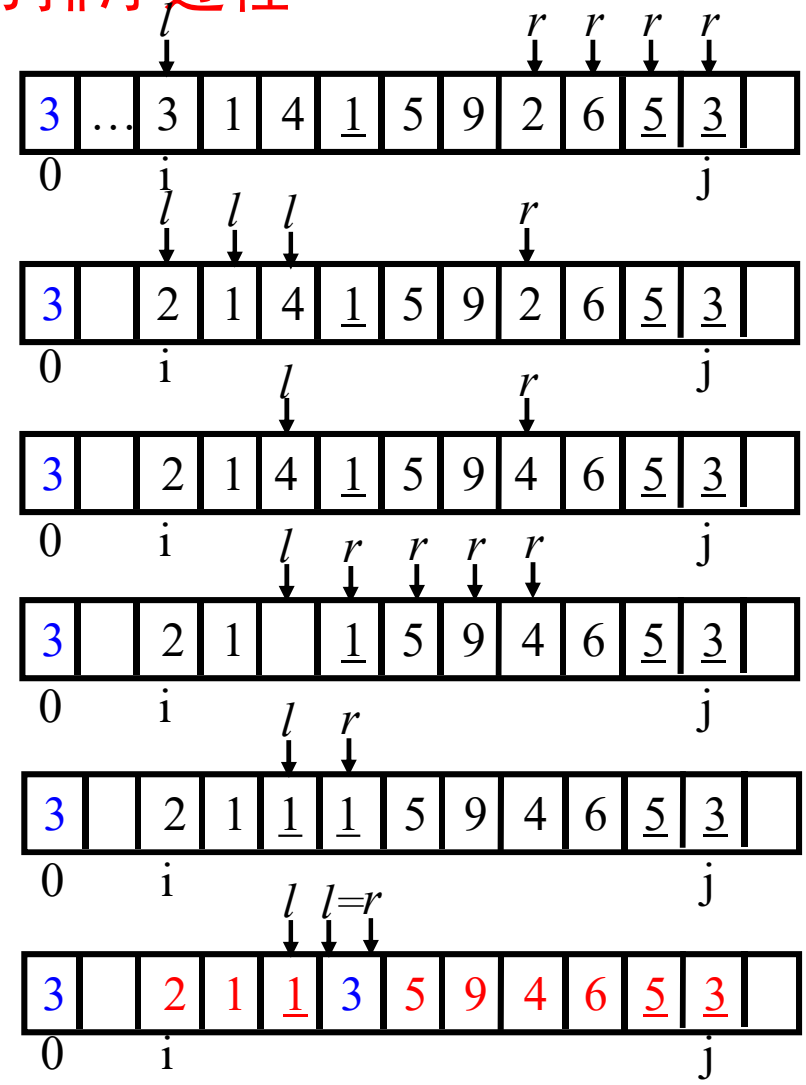
此时，A[i…j]被划分成为独立的两部分：

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



6.3 快速排序(cont.)

示例：一次划分排序过程



6.3 快速排序(cont.)

对A[i…j]进行一次划分：

```
int Partition (LIST A , int i , int j , keytype  pivot )
/*对A[i…j]进行一次划分，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

- 基准元素的选取是任意的，但不同选取方法对算法性能影响很大。
- 快排的性能不仅取决于待排序列的初始状态，还与基准元素的选取有关。
- 一般原则：子问题平衡策略
  - 最好情况下，每次都能将待排序列划分为规模相等的两部分。
  - 此时，划分遍数为$\log_2 n$，全部比较次数$O(n\log_2 n)$，交换次数$O(n\log_2 n)$。

6.3 快速排序(Cont.)

基准元素的选取方法

设 FindPivot(A, i , j) 求A[*i*].key,…,A[*j*].key的基准，返回其下标*k*。

- *v* =序列的最左关键字（固定选取基准）
- 随机选取
- *v* =(A[*i*].key, A[(*i*+*j*)/2].key, A[*j*].key的中值)
- *v* =从A[*i*].key到A[*j*].key 最先遇到的两个不同关键字中的较大者
  - 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。

3.2.2 快速排序(Cont.)

基准元素的选取：左边两个不同关键字的较大者

```
int  FindPivot( LIST A, int  i, int  j )
/* 若A[i],…,A[j]的关键字全部相同，则返回0；否则，
   返回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 ;                          //用k从左(i+1)向右(j)遍历
    for ( k=i+1 ;  k<=j; k++ )        // 从左到右查找不同的关键字
        if ( A[k].key > firstkey )       // 选择较大的关键字
            return  k ;                  // 返回大者下标k
        else if ( A[k].key < firstkey )
            return  i ;                  // 返回大者下标i
    return  0 ;                       //所有关键字都相同
}/* FindPivot */
```

6.3 快速排序(cont.)

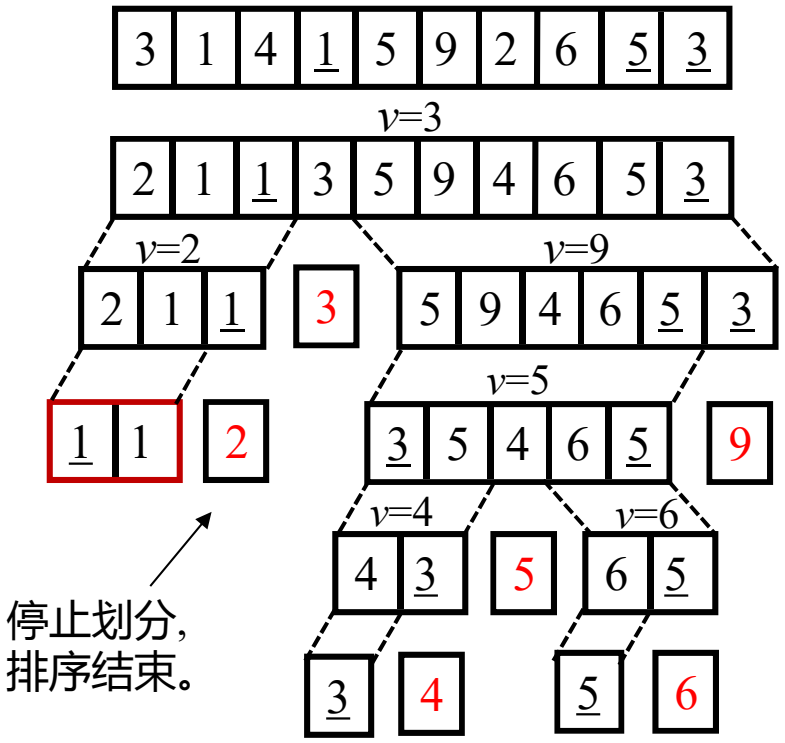
快速排序算法的实现

```
void QuickSort ( LIST A, int i , int  j ) {
      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 ; //pivot在数组A中的下标
      pivotindex = FindPivot ( A, i , j );    //基准元素的选取
      if ( pivotindex != 0 && i < j ) {       //递归排序直到终止条件
           if ( pivotindex != i ) //若所选pivot不是第1个记录，则与第1个记录交换
                swap( A[i] , A[pivotindex] );
           pivot = A[i].key ; //始终用序列的第1个关键字作为基准元素
           k= Partition ( A, i , j , pivot );    //一次划分
           QuickSort ( A, i , k-1);
           QuickSort ( A, k+1 , j);
      }
}/*QuickSort ( A, 1, n ) */
```

6.3 快速排序(Cont.)

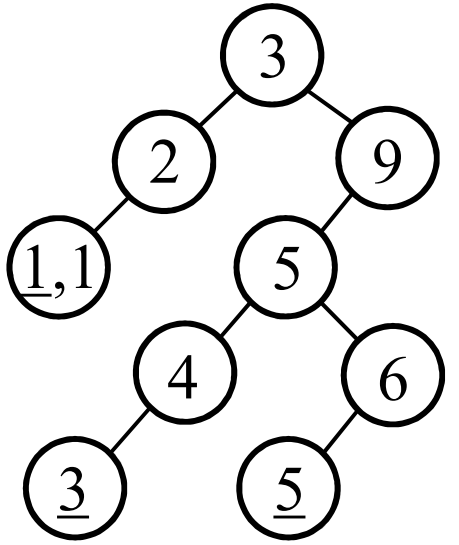
算法的性能分析

- 选取左边最先遇到的两个不同关键字的较大者作为基准，排序过程：



排序过程对应的判定树：

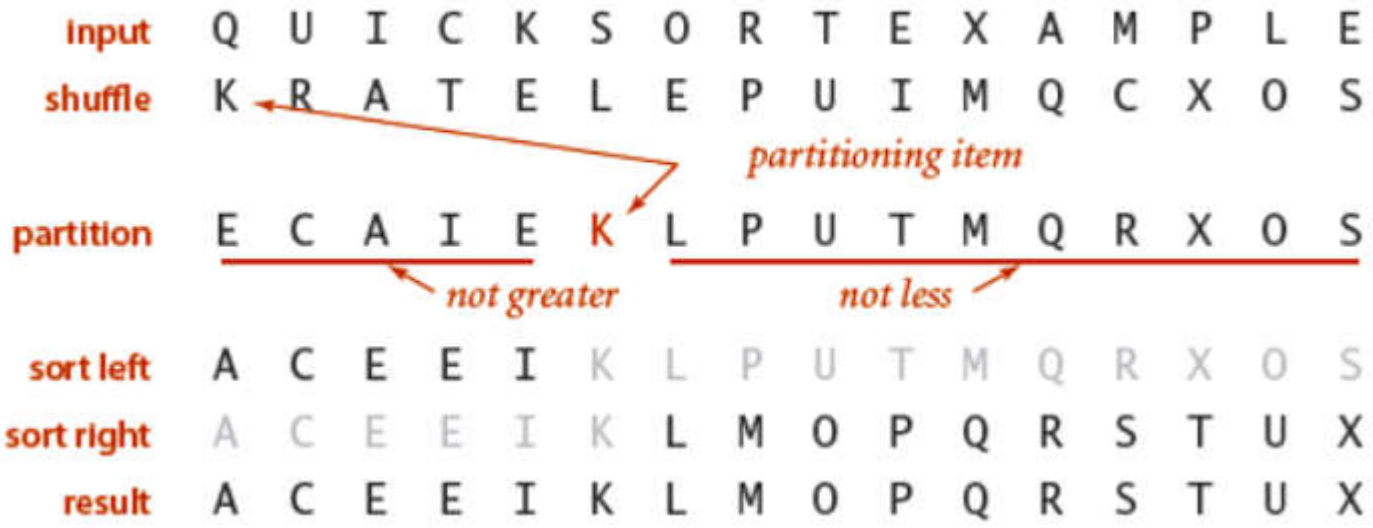
- 判定树每层对应一遍划分排序，因此，判定树的高度就是划分的遍数。
- 每一遍划分，最多扫描一遍整个数组，即比较次数不超过n次。



6.3 快速排序(Cont.)

快速排序：数组快速排序实现：随机选取基准元素

1. 对数组进行洗牌（shuffle the array），然后取第一元素作为基准。
2. 划分：确定一个元素的最后位置
3. 递归划分：直到所有元素都确定了最后位置



6.3 快速排序(Cont.)

划分基本步骤：//通过交换实现划分

1. 从左到右扫描i，找到属于右侧的记录；
2. 从右到左扫描j，找到属于左侧的记录；
3. 交换A[i]与A[j];
4. 重复直到i,j交叉。

6.3 快速排序(cont.)

快速排序：数组快速排序实现：取中间元素值作为基准元素

```
public static void quicksort(char[] items, int left, int right)
{
   int i, j;
   char x, y;

   i = left; j = right;
   x = items[(left + right) / 2];

   do
   {
      while ((items[i] < x) && (i < right)) i++;
      while ((x < items[j]) && (j > left)) j--;

      if (i <= j)
      {
         y = items[i];
         items[i] = items[j];
         items[j] = y;
         i++; j--;
      }
   } while (i <= j);

   if (left < j) quicksort(items, left, j);
   if (i < right) quicksort(items, i, right);
}
```

6.3 快速排序(cont.)

- 快速排序性能分析：最好情况
  - 每一次划分后，划分点的左侧子表与右侧子表长度相同，则有：

$$\begin{cases} T(n) \le 2T(n/2)+n \\ T(1) = C,\ C\text{为正的常数} \end{cases}$$

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

  - 时间复杂度为$O(n\log_2 n)$
  - 空间复杂度为$O(\log_2 n)$ //辅助存储空间
    - 递归树的深度为$\log_2 n$

6.3 快速排序(cont.)

- 快速排序性能分析：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$O(\log_2 n)$ //辅助存储空间
    - 每次只能排除一个元素，要递归剩下n-1个元素；
    - 需要进行n-1次递归调用
    - 其空间复杂度为$O(n)$
  - 稳定性：不稳定排序

3.2.2 快速排序(Cont.)

- 快速排序性能分析：平均情况
  - 假设$T(n)$是对n个记录A[1,…,n]进行快排所需时间，则QuickSort算法可知：

$$T(n) = cn + T(k\text{-}1) + T(n\text{-}k)$$

  - 假设待排序列的记录是随机排列的，则在一遍排序之后，k取 1 至 n 的之间的任意值的概率相同，因此，快排所需的平均时间为：

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

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

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

  - 时间复杂度为$O(n\log_2 n)$
  - 空间复杂度为$O(\log_2 n)$ //辅助存储空间

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：
    - 随机选择基准
    - 将待排序列重新随机排列
    - 区间内数据较少时直接用其他方法排序，以减小递归深度。

```
/*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;
        swap(A[index], A[last]);
       last--;
    }
```

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：
    - 如何处理包括重复元素的数据对象集合？
      - 如，[3,3,3,3,3,3,3,…]
    - 采用3-way 快排(quicksort):
      - 小于pivot的为一组
      - 等于pivot的为一组
      - 大于pivot的为一组

6.4 直接选择排序

- 算法基本思想
  - 选择排序的主要操作是选择。
  - 基本思想：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。
  - 直接选择排序过程：对待排序记录序列进行n-1遍处理：
    - 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。
    - 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：
    - 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]) ;      //导致不稳定？
     }
 }/* SelectSort*/
```

6.4 直接选择排序(cont.)

- 性能分析
  - 比较次数：

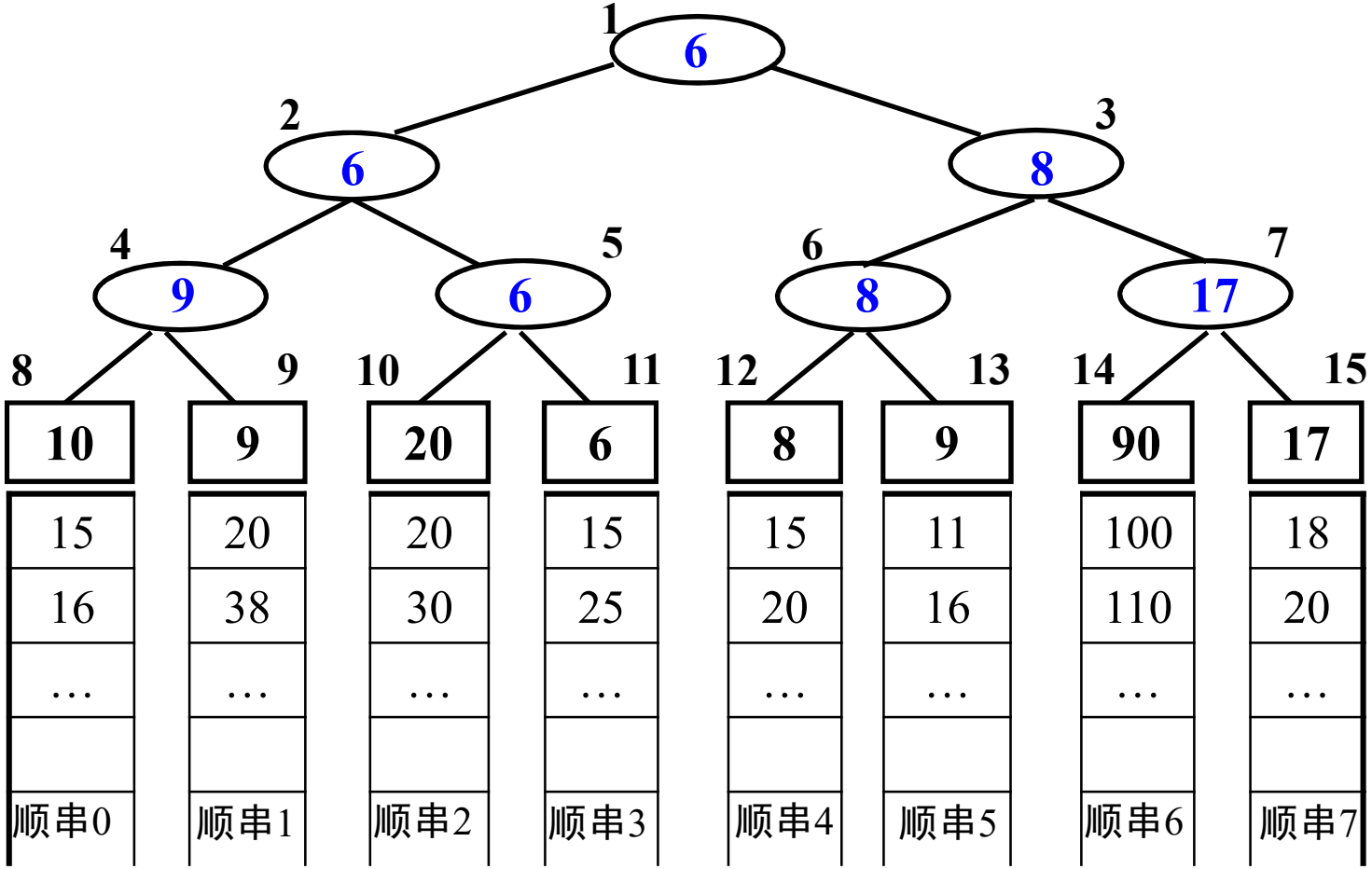
$$\sum_{i=1}^{n-1}(n-i)=\frac{1}{2}n(n-1)=O(n^2)$$

  - 每遍交换次数：
    - 最好情况（正序）：0次
    - 最坏情况：每次比较都交换
  - 时间复杂度为$O(n^2)$
  - 空间复杂度为$O(1)$ //辅助存储空间
  - 稳定性：不稳定排序

6.5 锦标赛排序

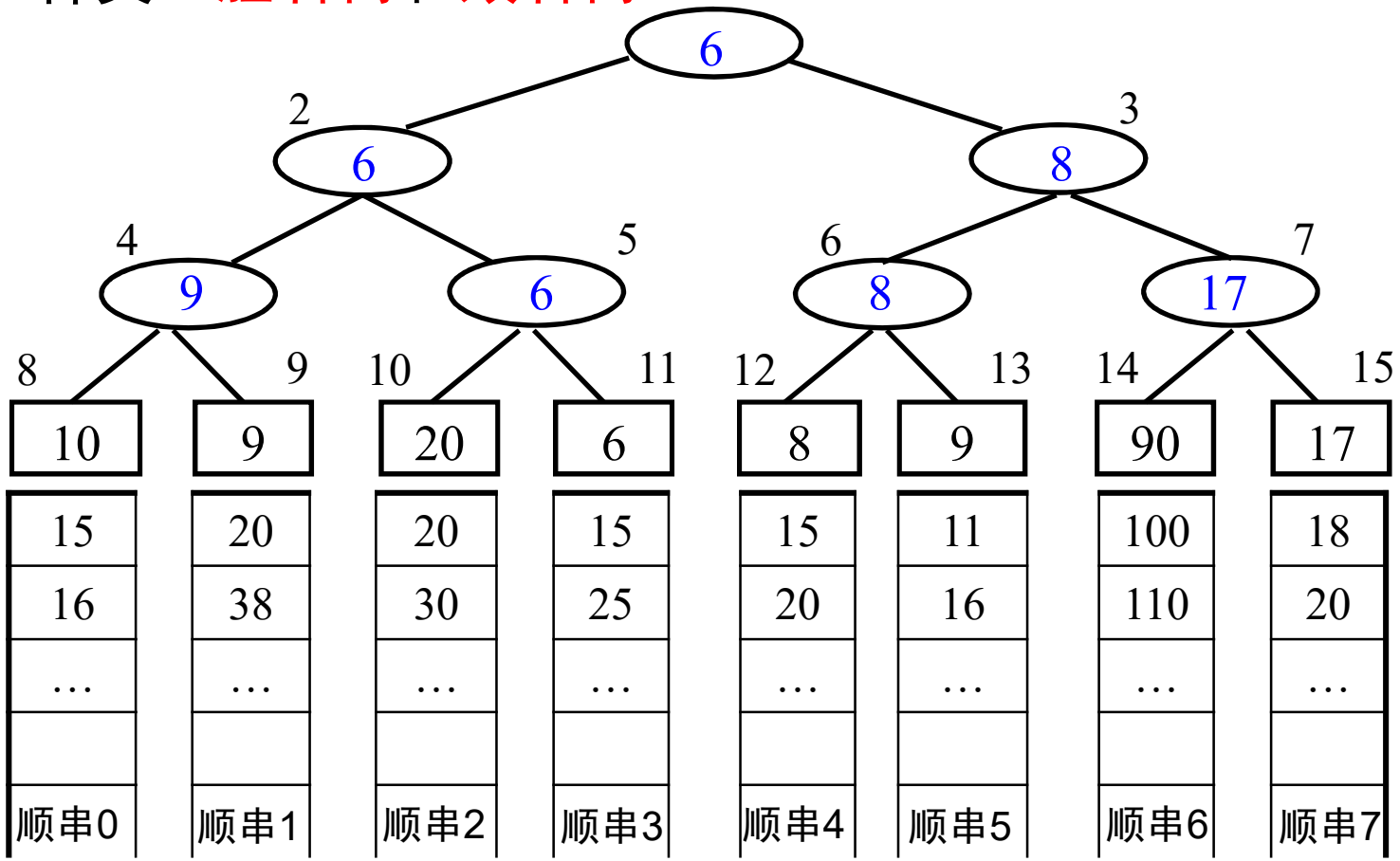
背景

- 如何从n个元素中选择最小的，进而对n个元素排序？
- 如何把K个非递减的序列归并成一个非递减的序列？



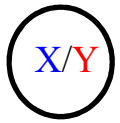
6.5 锦标赛排序(cont.)

- 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。

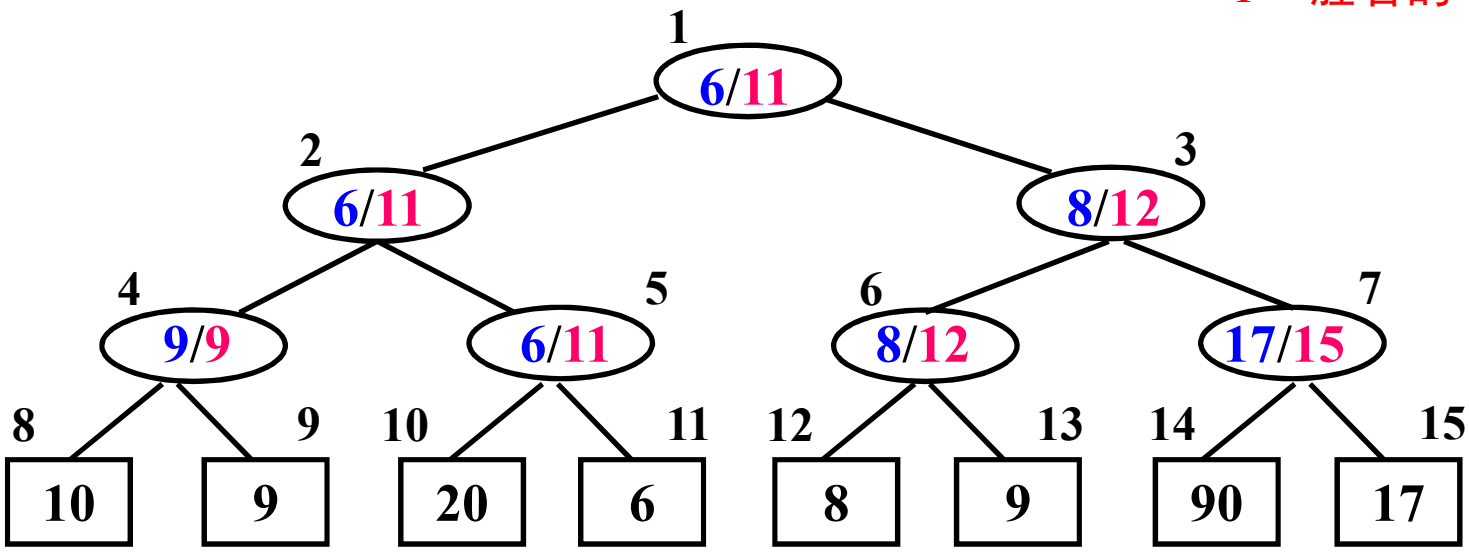


6.5 锦标赛排序(cont.)

胜者树构建



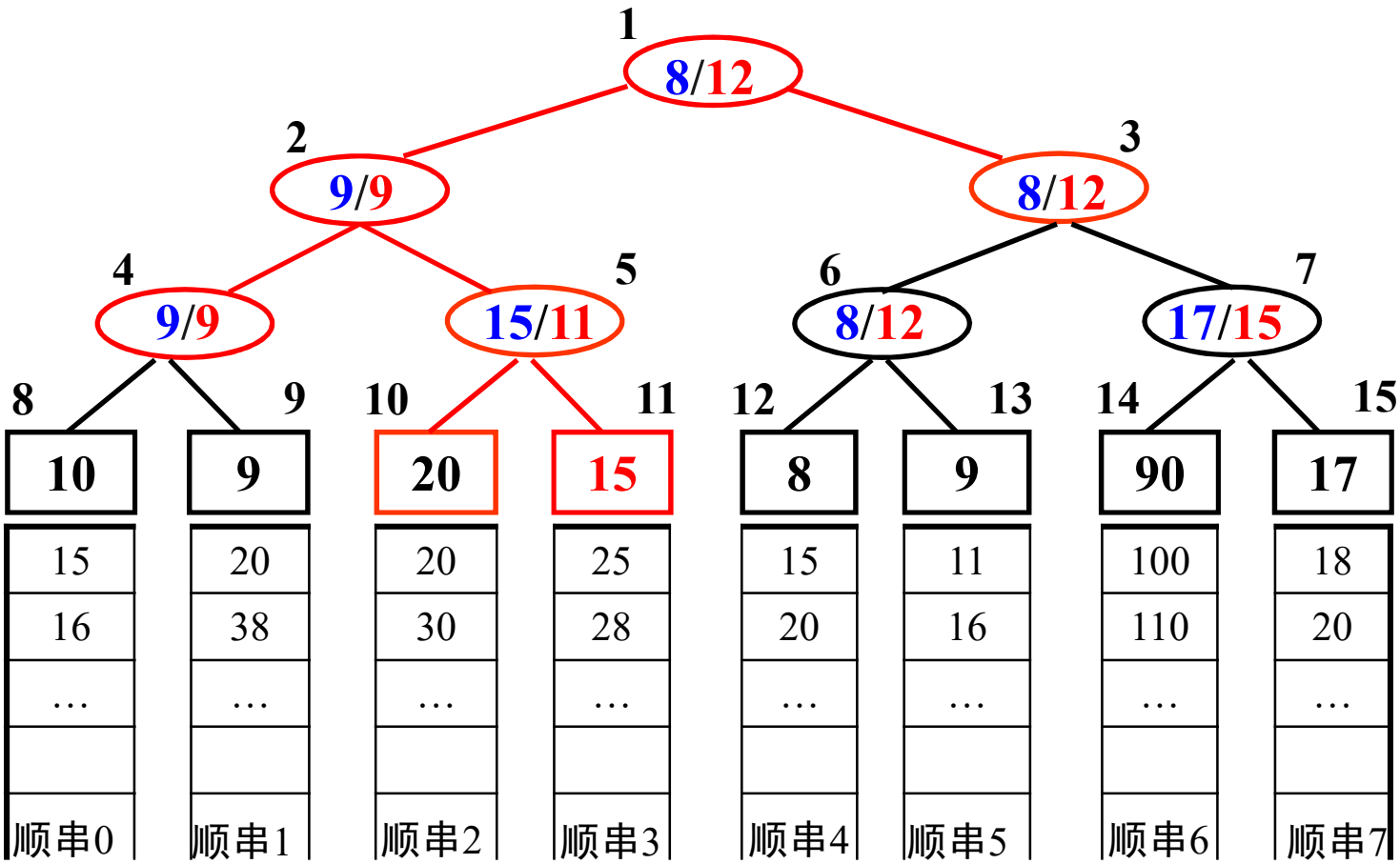
X—胜者关键字
Y—胜者的下标



- 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

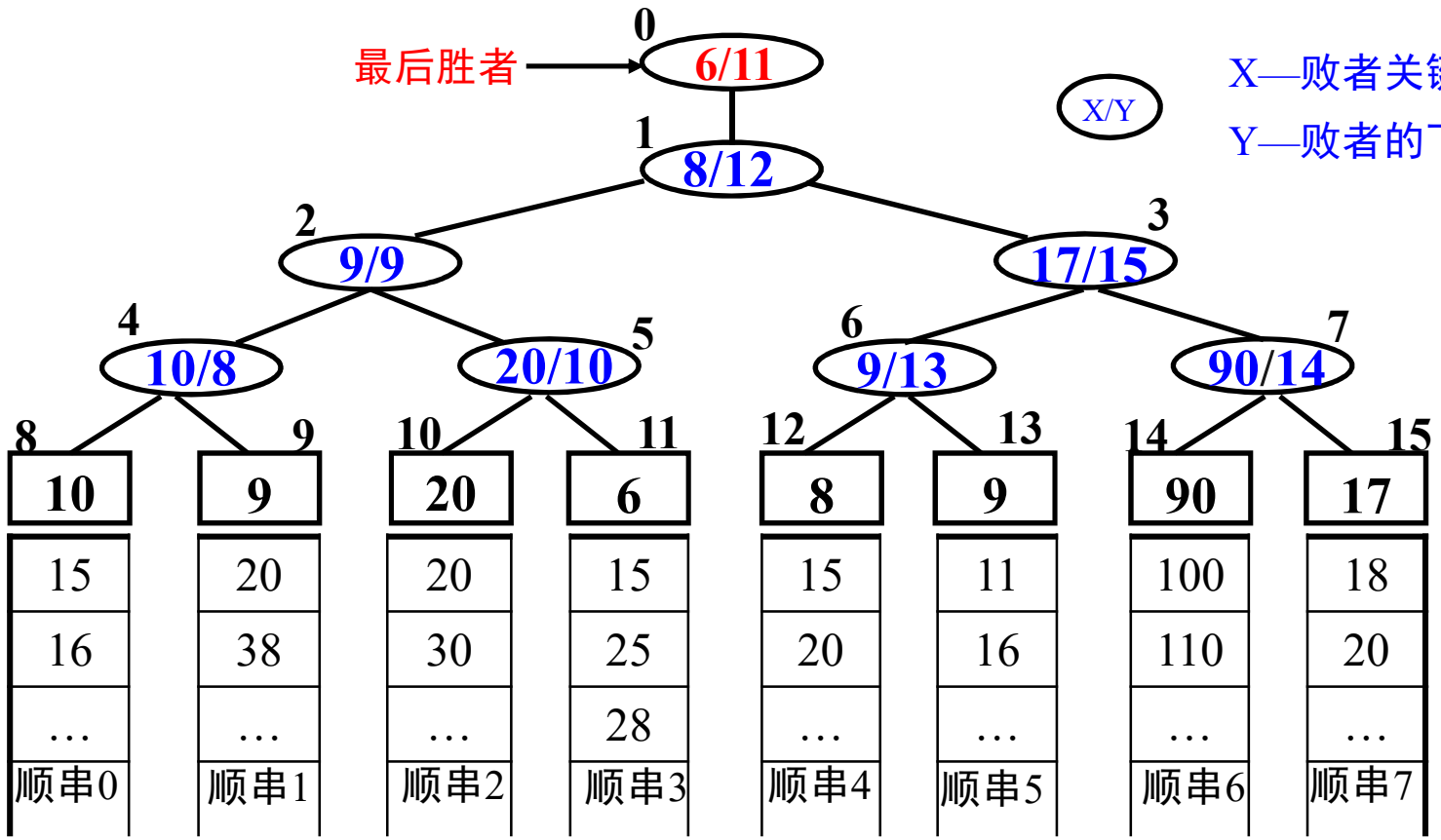
6.5 锦标赛排序(cont.)

- 胜者树的重构：
  - 新进入的结点与兄弟结点比较，直到树根



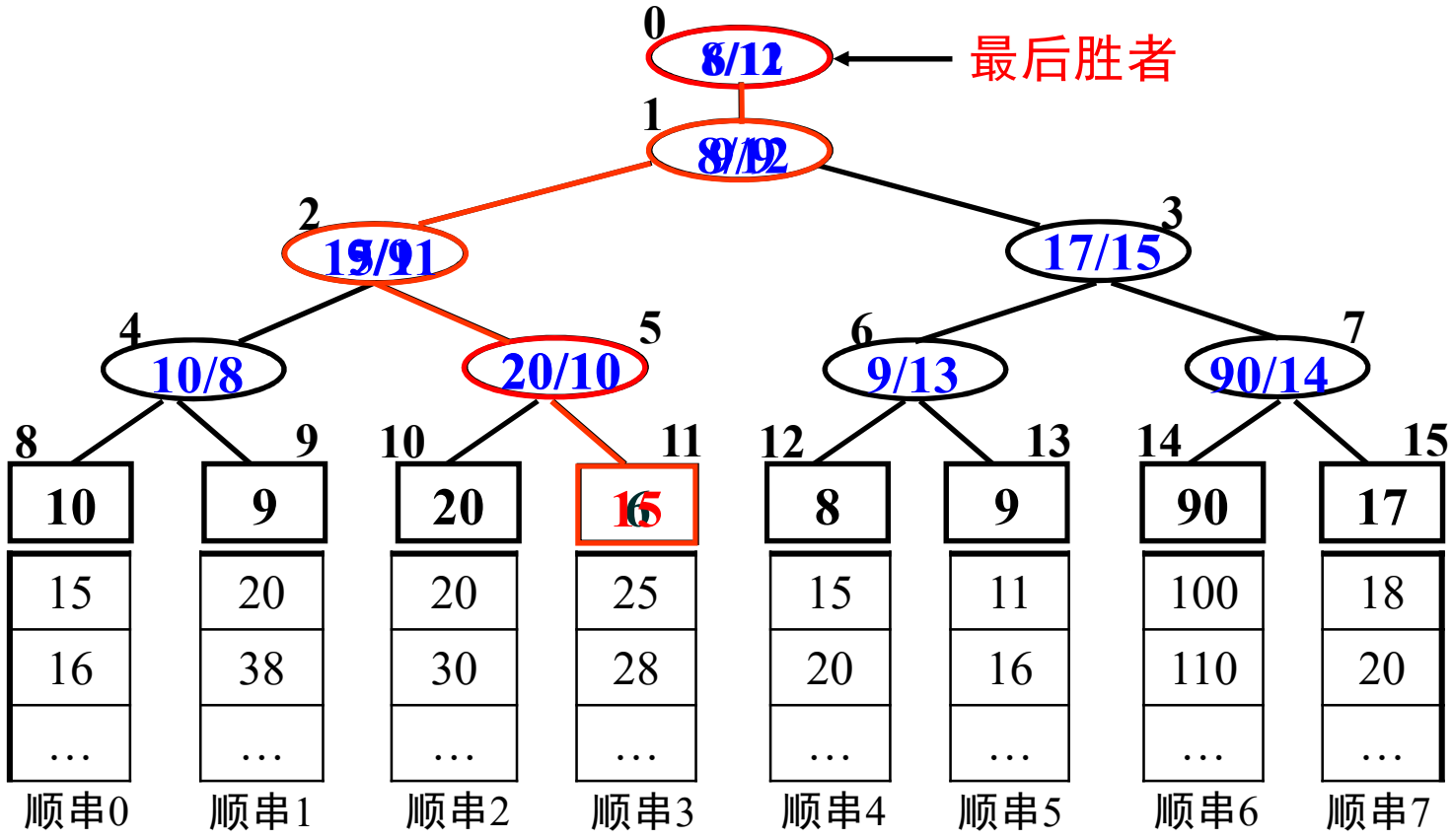
6.5 锦标赛排序(cont.)

- 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
  - 新进入的结点与其父结点进行比赛：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(cont.)

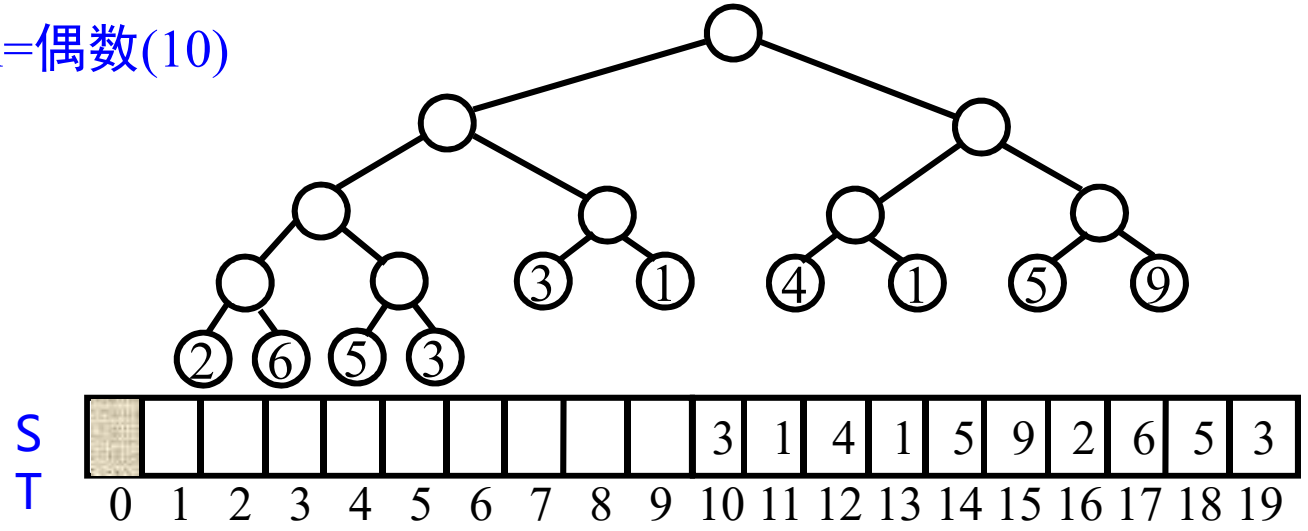
- 锦标赛排序的基本思想
  - 首先通过类似于淘汰赛的方式，选出最小（大）关键字记录（冠军）：构建选择树。
  - 然后根据关键字大小，依次选出其他记录，从而实现对整个记录序列的排序：重构选择树。
  - 胜者树类型说明：

```
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];
```

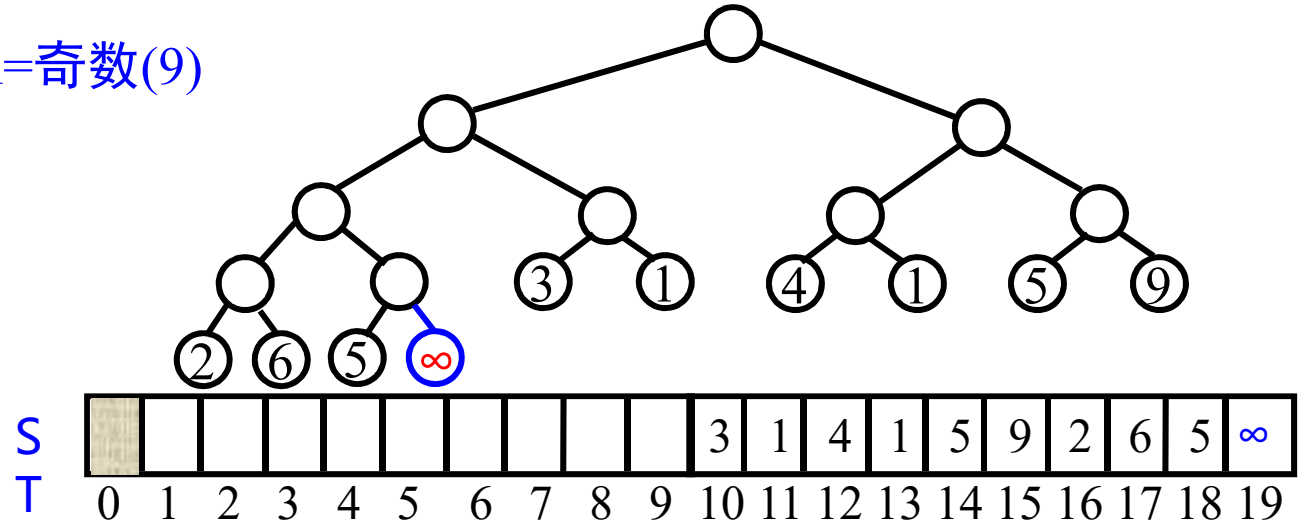
6.5 锦标赛排序(cont.)

（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(10)



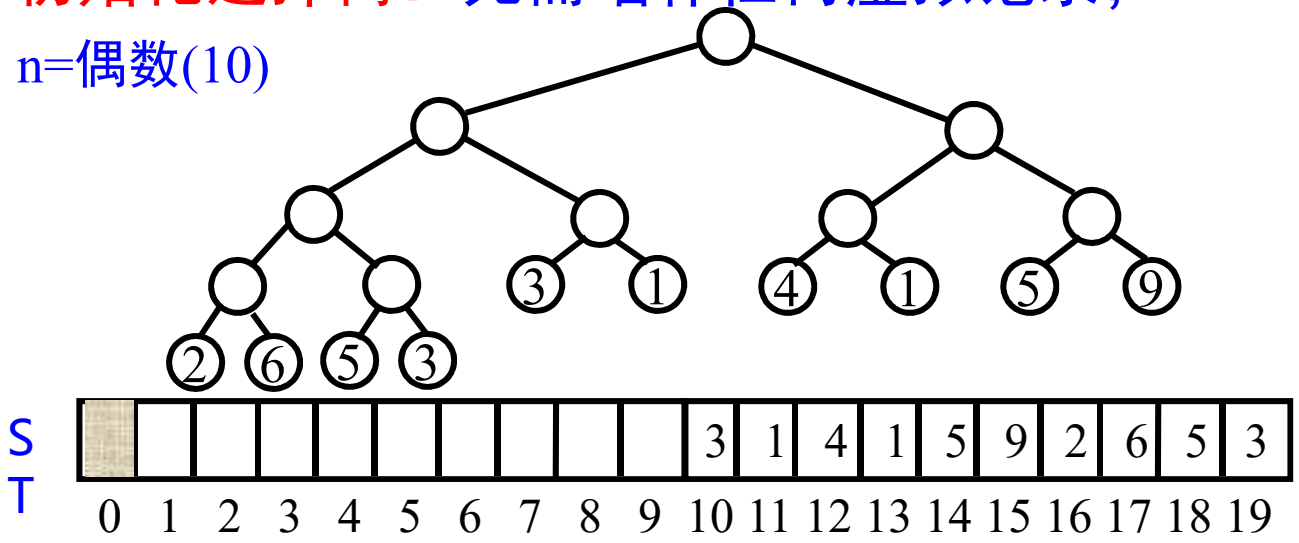
– n=奇数(9)



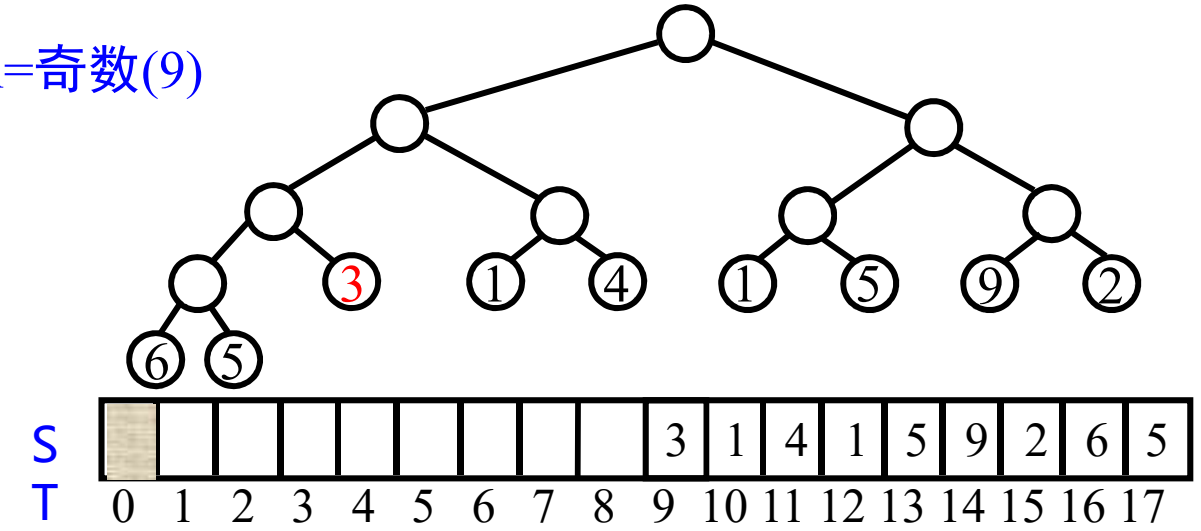
6.5 锦标赛排序(cont.)

（1）初始化选择树：无需增补任何虚拟记录，

– n=偶数(10)



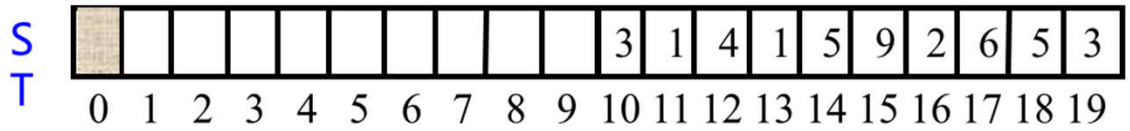
– n=奇数(9)

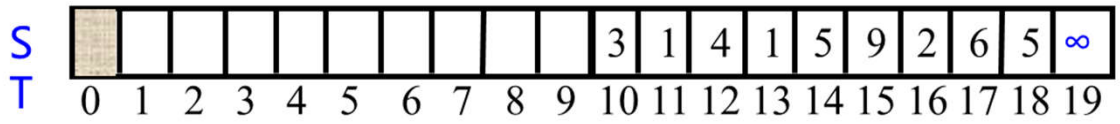


6.5 锦标赛排序(cont.)

初始化选择树的实现

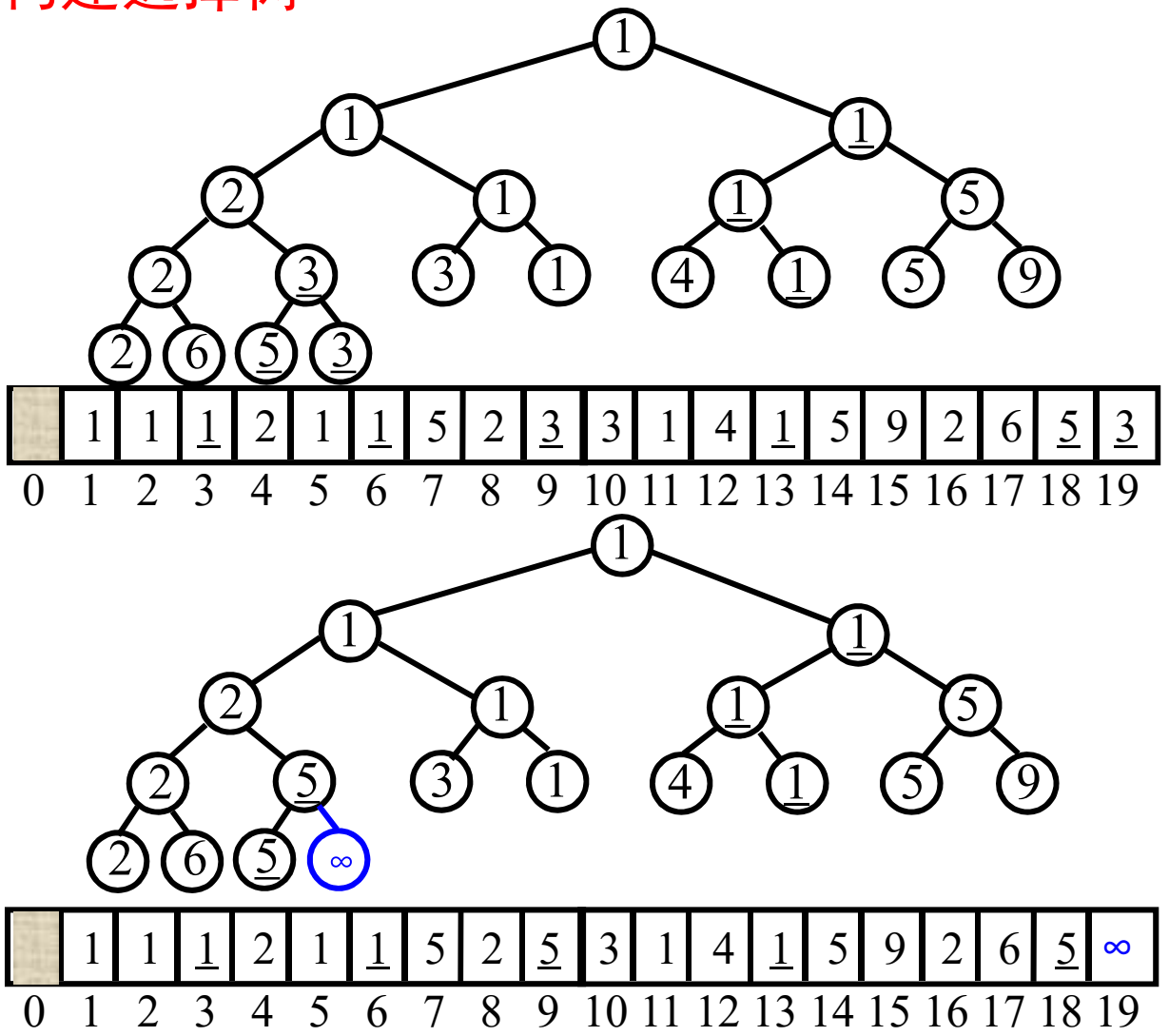
```
int Initial( LIST A, int n, TREE ST[] )
 /* 用待排序列初始化选择树，确定并返回其长度 */
{
    if ( n%2==0){/*偶数：将待排序列依次存放在ST[n]至ST[2n-1]单元*/
        for( int i= n; i<=2*n-1; i++ ){
            ST[i].key = A[i-n+1].key;
            ST[i].id = i;
        }
         return 2*n-1;
    }
    else{ /*奇数：将待排序列依次存放在ST[n+1]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*/
         return 2*n+1;
    }
}/* Initial */
```





6.5 锦标赛排序(cont.)

（2）构建选择树



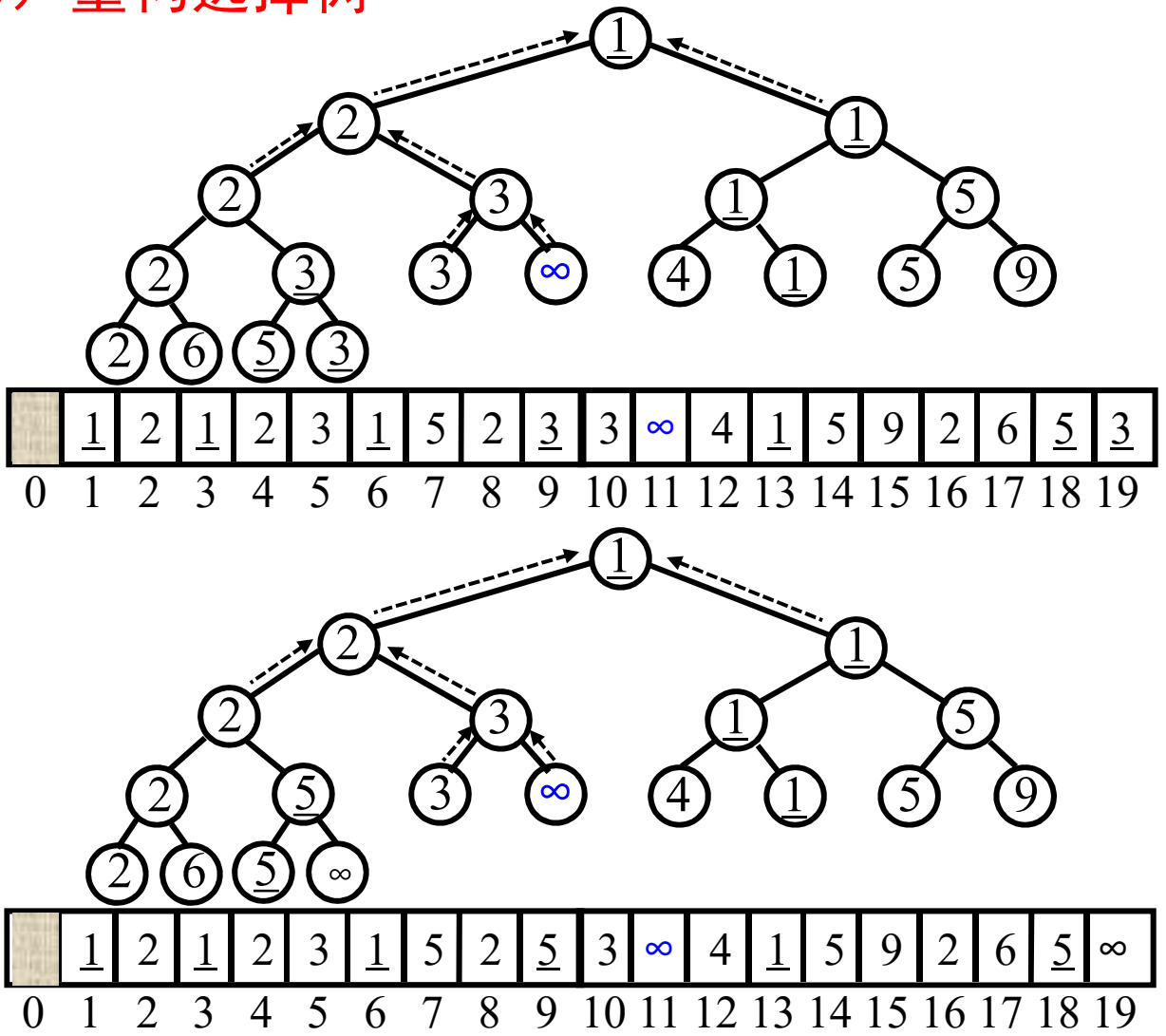
6.5 锦标赛排序(cont.)

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-1].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，相等时选左*/
            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，相等时选左*/
            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(cont.)

- 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，选取左兄弟结点关键字

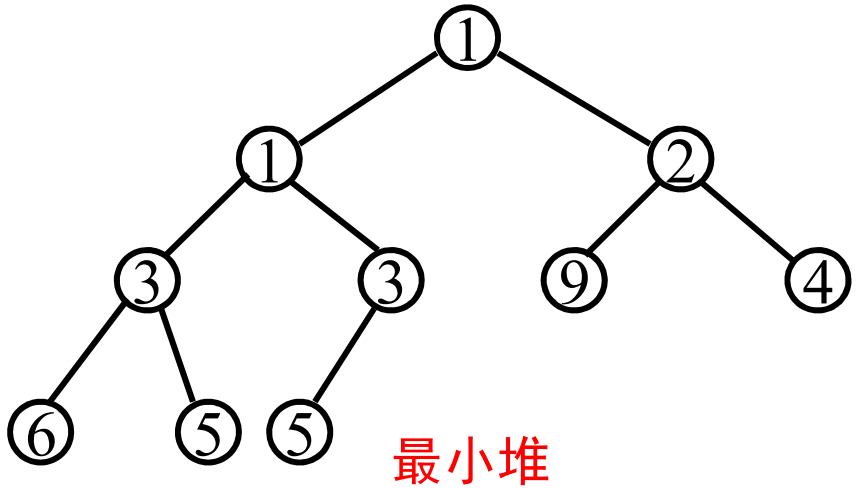
6.5 堆排序

- 堆排序是对直接选择排序的改进，改进着眼点：
  - 如何减少关键字之间的比较次数？
    - 利用好每趟比较后的结果：
      - 也就是在找出关键字值最小记录的过程中，同时也找出关键字值相对较小的记录，
    - 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。
  - 选择树（锦标赛）排序、堆排序

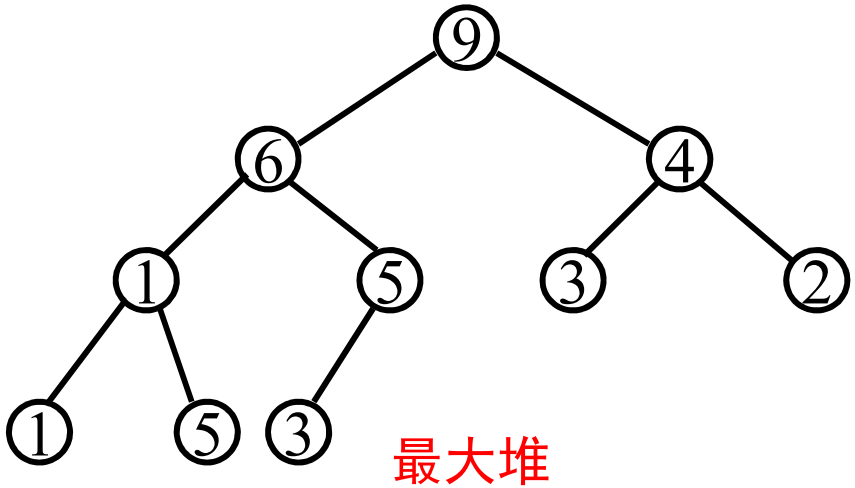
6.5 堆排序(cont.)

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*$i \le n$，则A[i].key $\le$ A[2*i].key ;
  - (2) 若2*i+1 $\le n$，则A[i].key $\le$ 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*$i \le n$，则A[i].key $\ge$ A[2*i].key ;
  - (2) 若2*i+1 $\le n$，则A[i].key $\ge$ A[2*i+1].key 。



最小堆



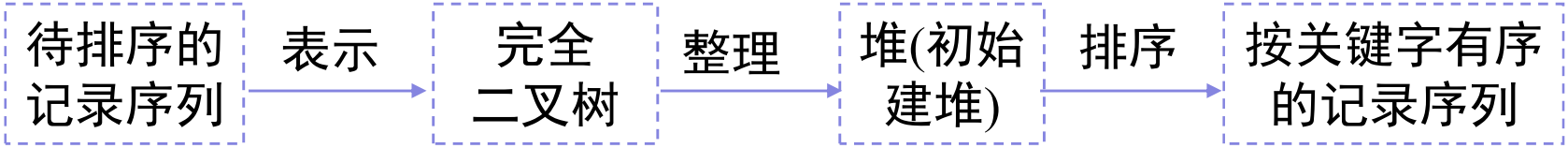
最大堆

6.5 堆排序(cont.)

- 堆的性质（最小堆）
  - 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　$1 \le i/2 < i \le n$。
  - 在堆中，以任意结点为根的子树仍然是堆。
  - 每个结点都代表(是)一个堆，每个叶也点可视为堆。
    - 以堆规模（结点数量）不断扩大的方式进行初始建堆。
  - 堆（包括各子树对应的堆）的根结点关键字是最小的。
  - 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.)

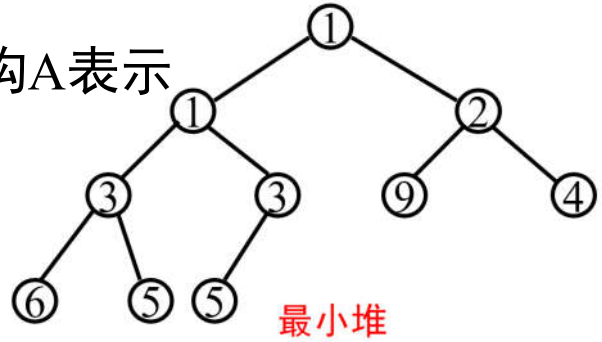
- 堆排序的基本思想：
  - 首先将待排序的记录序列用完全二叉树表示；
  - 然后将完全二叉树构造成一个堆，此时，选出了堆中所有记录关键字的最小者；
  - 最后将关键字最小者从堆中移走，并将剩余的记录再调整成堆，这样又找出了次小的关键字记录，以此类推，直到堆中只有一个记录。



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)

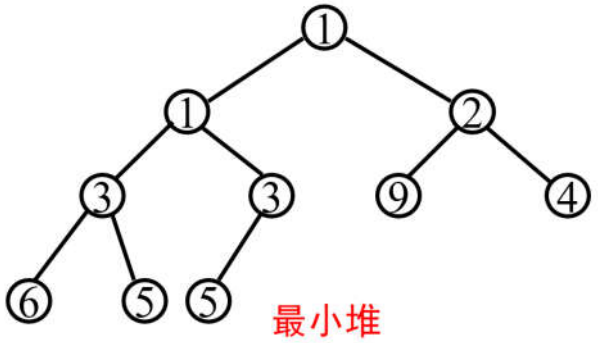
3. 堆的重构（堆排序）：令i = n, n-1 ,…, 2

(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.)



最小堆

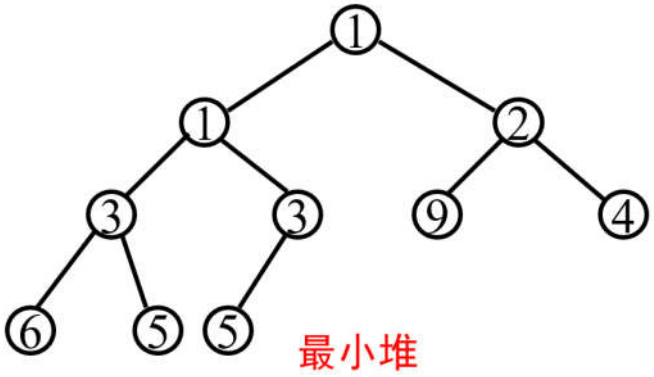
堆排序的实现：

```
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，以i为根，最大叶下标为n
     for( i=n; i>=2; i--) {
          swap(A[1],A[i]);     //堆顶与当前堆中最大叶下标的结点交换
          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)

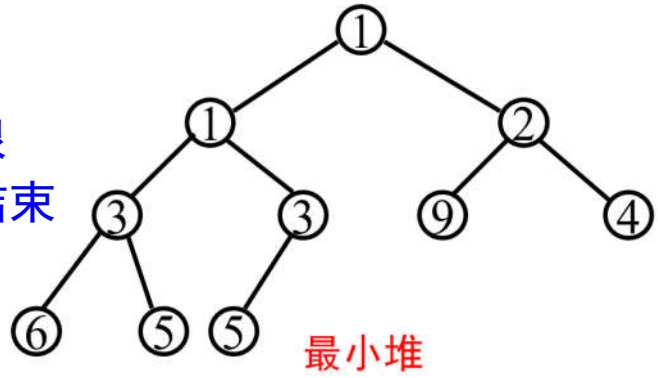
- 把以A[first]为根，以A[last]为最右边叶结点的完全二叉树整理成堆。
  - 根据堆定义，把完全二元树中关键字最小元素放到堆顶，而把原堆顶元素下推到适当位置；
  - 最后使(A[first],…,A[last])成为堆。
- 怎样把关键字最小的元素放到堆顶，把堆顶元素下推到适当位置呢？
- 具体操作(要点)如下：
  - 把完全二元树的根或者子树的根与其左、右儿子比较，如果大于其左 / 右儿子，则与其中较小者交换；
  - 若其左、右儿子相等，则与其左儿子交换；
  - 重复上述过程，直到以A[first]为根的完全二元树成为堆为止。



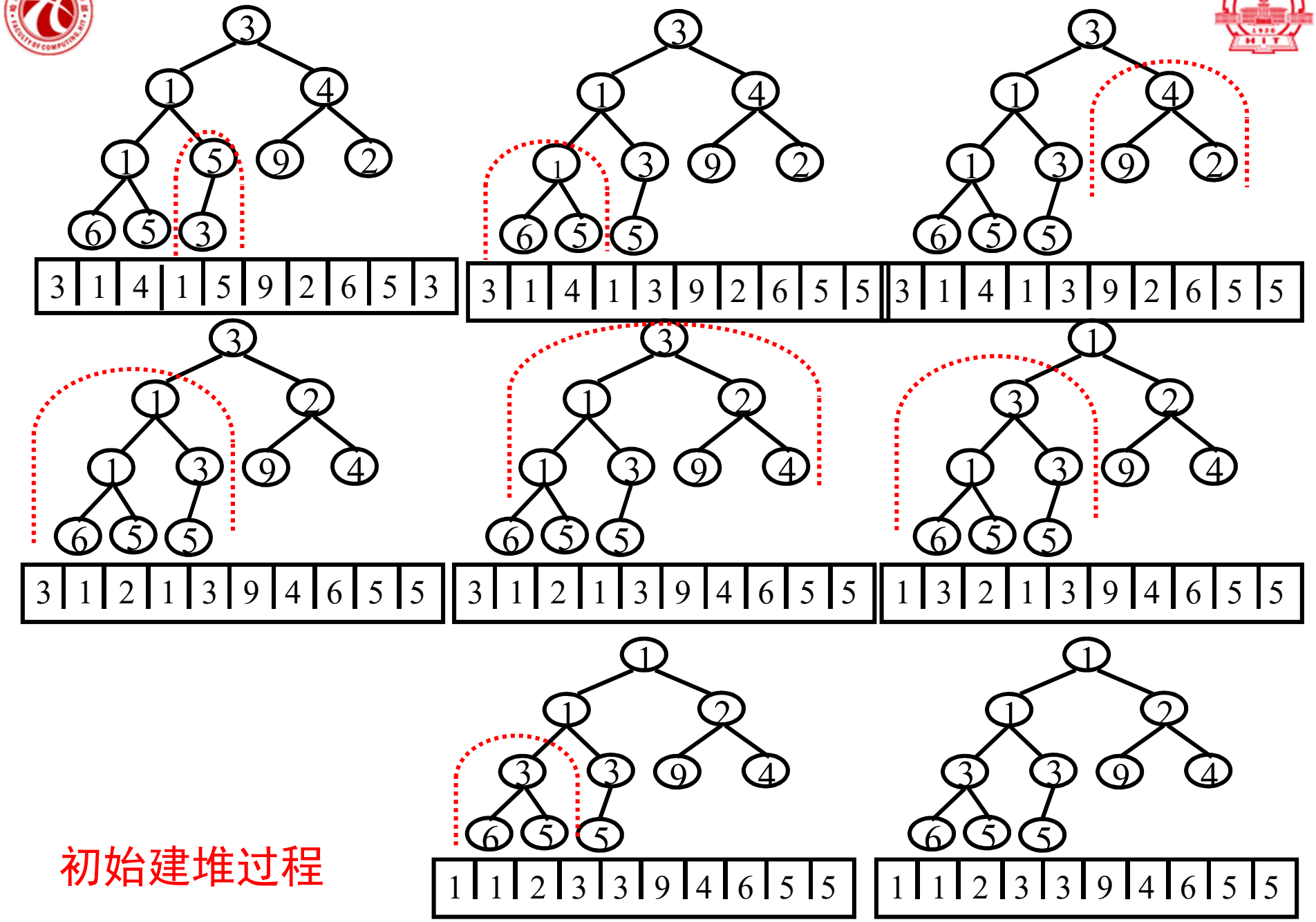
最小堆

6.5 堆排序(cont.)

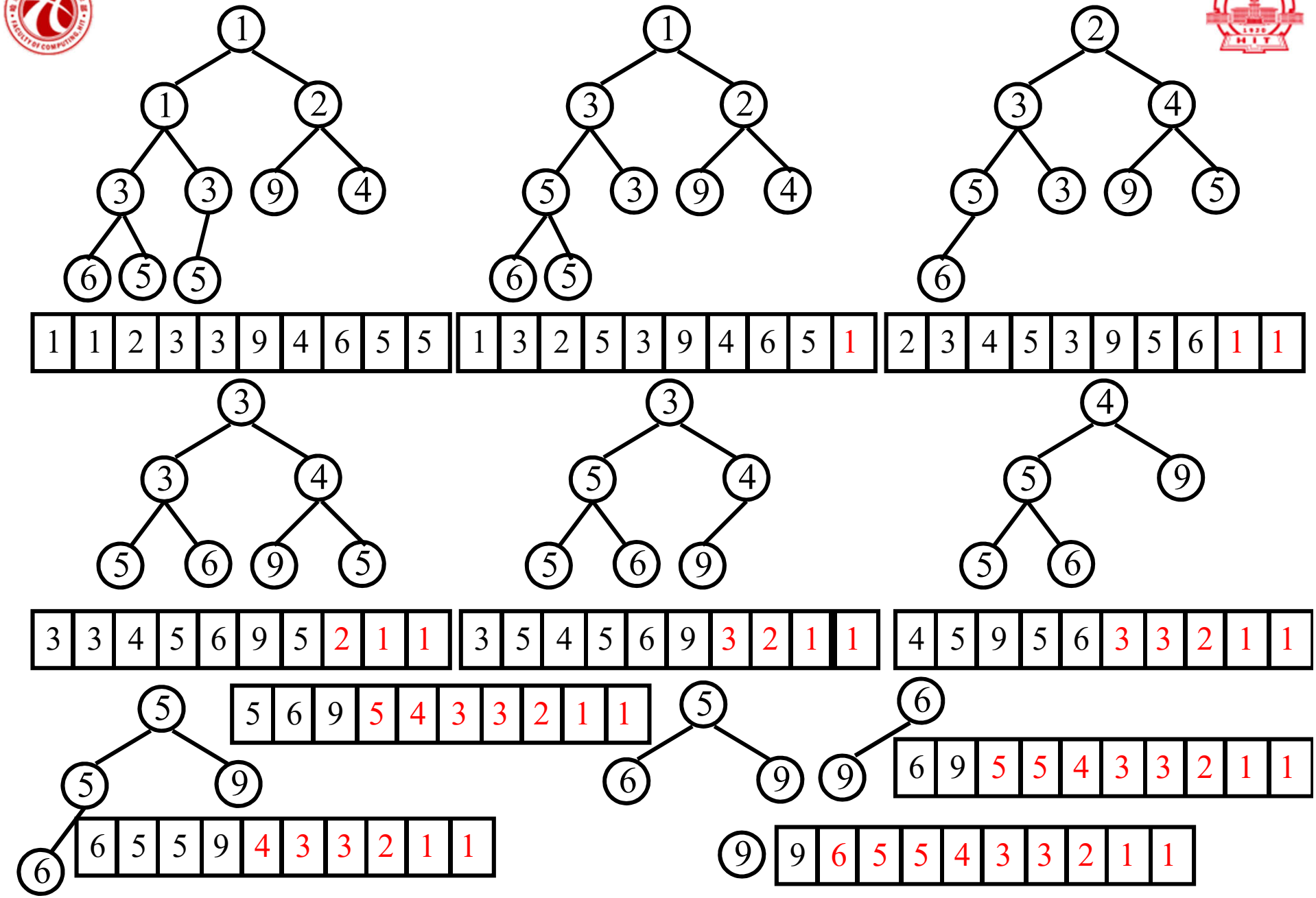
```
void PushDown(LIST A, int first, int last)   //PushDown(A, first, last)算法实现
{     //整理堆：把A[first]下推到完全二元树的适当位置
      int r=first;        // r是被下推到的适当位置，初始值为根first
      while(r<=last/2)                          //A[r]不是叶，否则是堆，时间复杂度
          if((r==last/2) && (last%2==0)) {// r有一个儿子在2*r上，且为左儿子
            if(A[r].key>A[2*r].key)
                swap(A[r],A[2*r]);          //与左儿子交换，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*/
```



最小堆


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


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

堆排序过程

6.5 堆排序(cont.)

- 性能分析：时间性能 $O(n\log_2 n)$
  - PushDown()函数的执行时间：
    - 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)
  - PushDown()函数的比较次数：
    - 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 $T_1 = \sum_{j=1}^{h-1} 2^{h-j} \times j = 2 \times 2^{h-1} \sum_{j=1}^{h-1} \frac{j}{2^j}$
  - 再令 $S = \sum_{j=1}^{h-1} \frac{j}{2^j} = \frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \ldots + \frac{h-1}{2^{h-1}}$
  - 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：

$$2(\lfloor\log_2(n-1)\rfloor+\lfloor\log_2(n-2)\rfloor+\ldots+\lfloor\log_2 2\rfloor) < 2n\lfloor\log_2 n\rfloor$$

- 堆排序的最好、最坏和平均时间复杂度： $O(n\log_2 n)$
- 堆排序空间复杂度： $O(1)$
- 堆排序稳定性：不稳定的排序

6.5 堆排序(cont.)

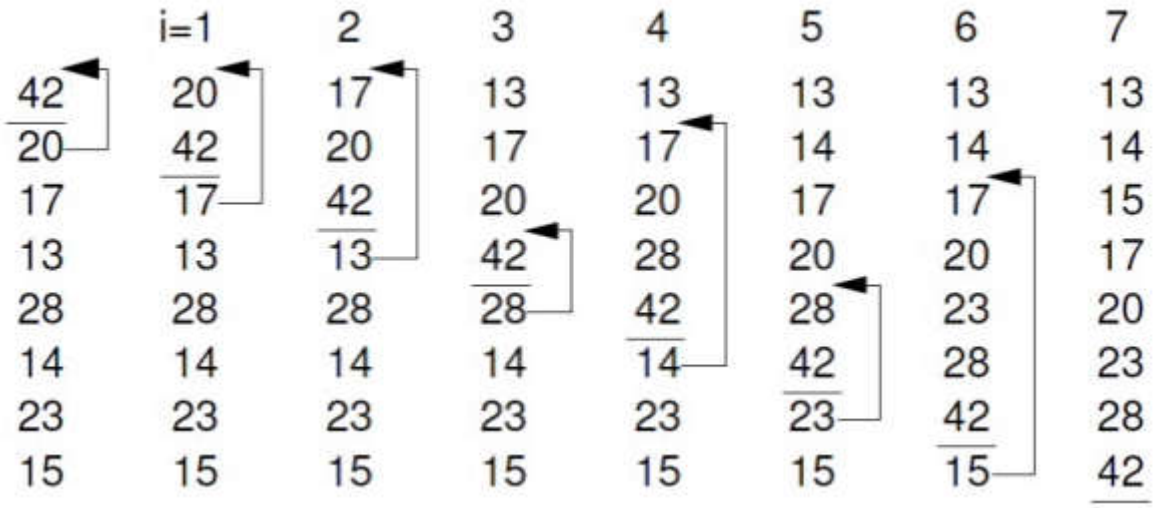
- 性能分析：堆与AVL树

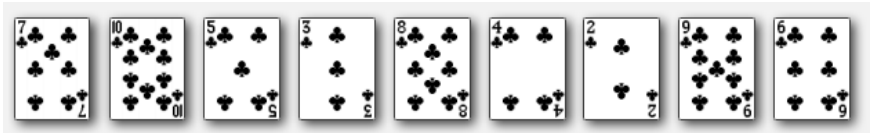
ADT	Heap	AVL tree
S=new-empty	O(1)	O(1)
S.insert(x)	$O(\log_2 n)$	$O(\log_2 n)$
S.delete(x)	$O(\log_2 n)$	$O(\log_2 n)$
y=S.predecessor(x)	O(n)	$O(\log_2 n)$
y=S.successor	O(n)	$O(\log_2 n)$

6.6 直接插入排序

基本思想：

- 类似整理手中纸牌
- 基本思想：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有序的序列中，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：从后往前，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.)

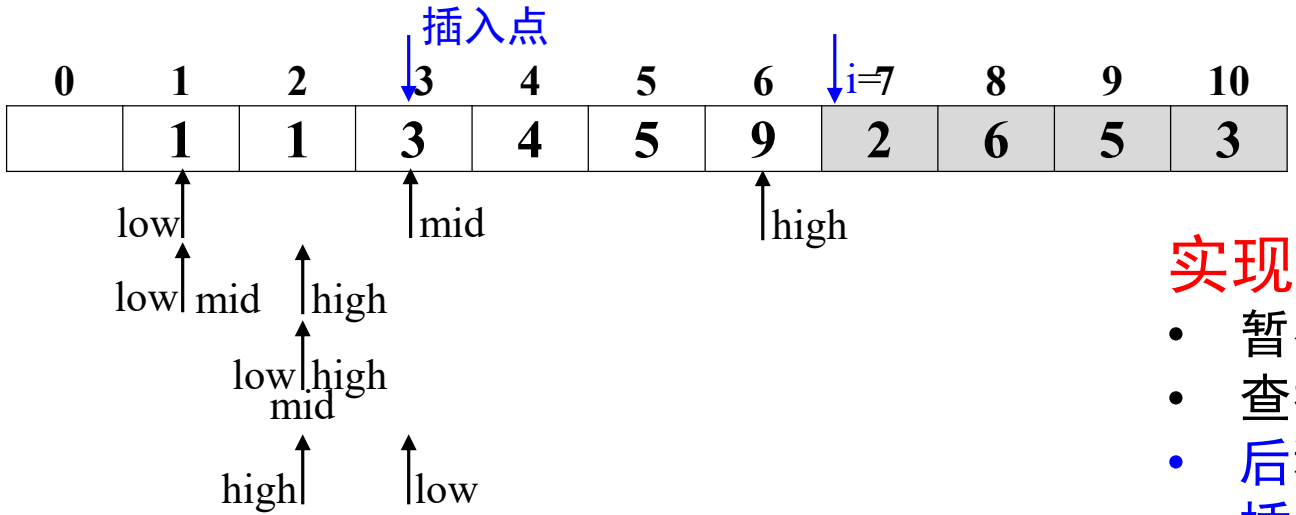
- 算法性能分析：时间复杂度
  - 最好情况下（正序）：
    - 比较次数：n-1
    - 移动次数：0
    - 时间复杂度为$O(n)$
  - 最坏情况下（反序）：
    - 时间复杂度为$O(n^2)$
  - 平均情况下（随机排列）
    - 时间复杂度为$O(n^2)$
- 算法的性能分析
  - 空间复杂度：O(1)
  - 稳定性：稳定的排序
  - 在线排序（online)

6.6 （直接）插入排序(cont.)

- 算法的性能分析
  - 首先，由于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，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)
  - 对直接插入排序进行改进
  - 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，从而减少比较次数，加快寻找插入点的速度， 。
  - 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，右区插入，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];                      //记录插入
    }
}/*BinaryInsertSort*/
```

6.6 （直接）插入排序(cont.)

稳定性与复杂度

- 稳定的排序方法：
  - 只是把大的向后移动，相等的没有移动
- 与直接插入排序相比，明显减少了关键字比较次数，因此排序速度比直接插入排序算法快
- 但记录移动次数并未减少，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

- 希尔排序对直接插入排序改进，改进着眼点：
  - 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子序列
  - 组内如何排序？直接插入排序
    - 比较相距一定间隔的元素，每趟比较所用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

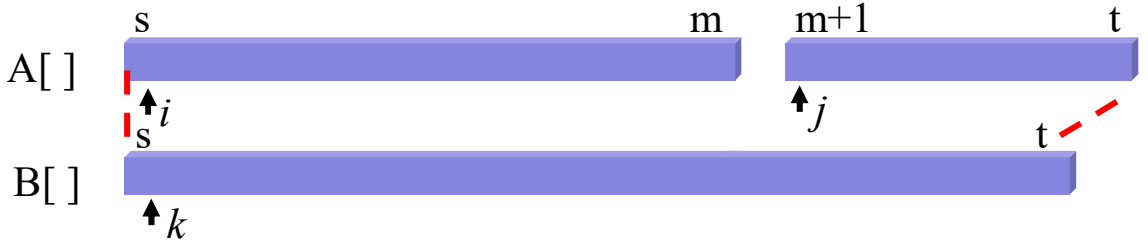
```
void ShellSort(int n, LIST A)
{    int i, j, d;
     for (d=n/2; d>=1; d=d/2) {     //处理不同d的子序列，直到间距为1
        for (i=d+1; i<=n; i++) {    //将A[i]插入到所属子序列中适当位置
           A[0].key= A[i].key;      //暂存待插入记录
           j=i-d;                   //j指向所属子序列的最后一个记录
           while (j>0 && A[0].key< A[j].key) {  //找插入位置
              A[j+d]= A[j];         //记录后移d个位置
              j=j-d;                //向左：比较同一子序列的前一个记录
                                    //比如, 13 与前面的比较
           }
           A[j+d]= A[0];            //找到位置，插入A[i]
        }
     }
} /*ShellSort*/
```

6.7 希尔排序：分组插入排序(cont.)

- 算法性能分析
  - 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的(1, 5, 19, 41, 109,...)
  - 希尔排序的时间性能在$O(n^2)$和$O(nlog_2n)$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
  - 主要思想：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
  - 设两个有序序列为A[s] ~ A[m]和A[m+1] ~ A[t]，归并成一个有序序列B[s] ~ B[t]



6.8 （二路）归并排序(cont.)

- 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]

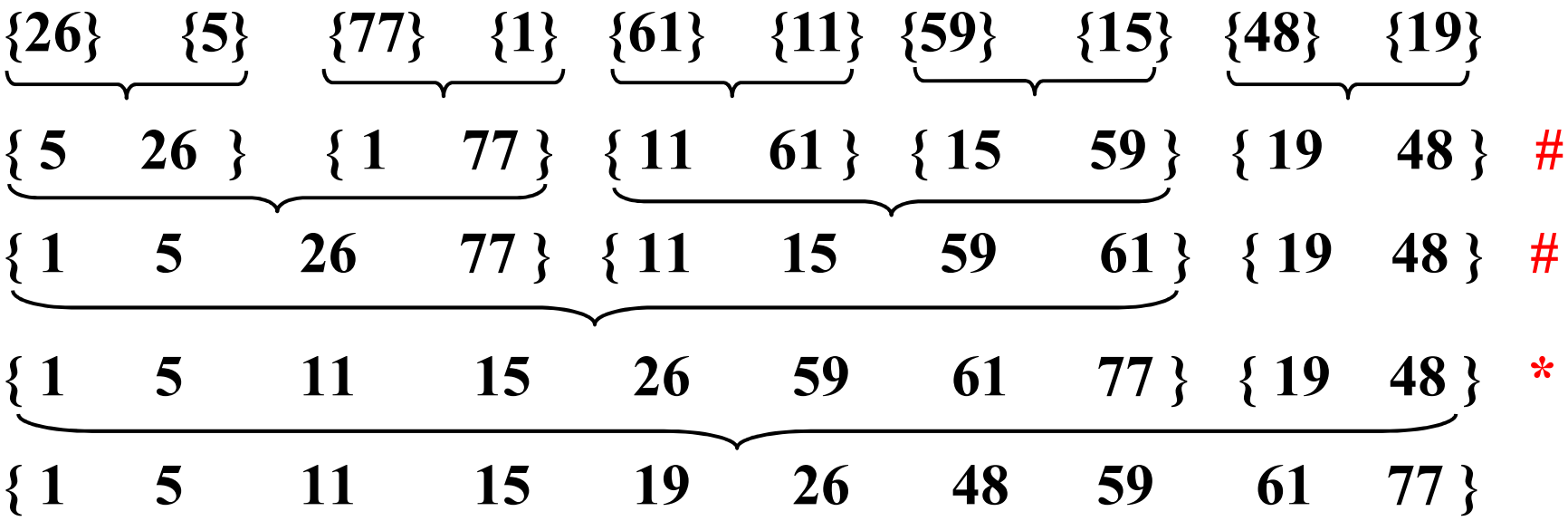
```
void Merge (int s , int m , int t , LIST  A , LIST  B)
{   int i = s ;  j = m+1 , k = s ;  //置初值
    //两个序列非空时，取小者输出到B[k]上
    while ( i <= m && j <= t )
        B[k++] = ( A[ i ].key <= A[ j ].key) ? A[i++] : A[j++] ;
    //第一个子序列非空(未处理完)，则复制剩余部分到B
    while ( i <= m )    B[k++] = A[i++] ;
    //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(*pass*)

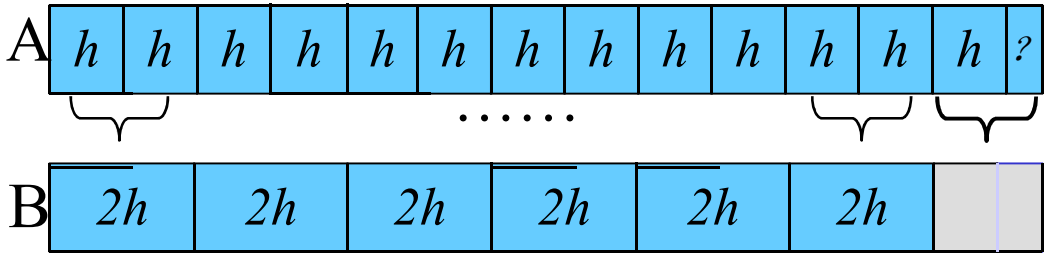


6.8 （二路）归并排序(cont.)

- 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 2*h* 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); //Case1 归并长度为h的两个有序子序列
     if ( i+h-1< n )    //Case2 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 // Case3 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，时间复杂度（总时间代价)：$O(n\log_2 n)$。
  - 这是归并排序算法的最好、最坏、平均的时间性能。
- 空间性能：
  - 算法执行时，需要占用与原始记录序列同样数量的存储空间。
  - 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(Divide and Conqur)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。
- 递归终止条件
  - 子序列长度为 1，一个记录自然有序。

6.8 （二路）归并排序(cont.)

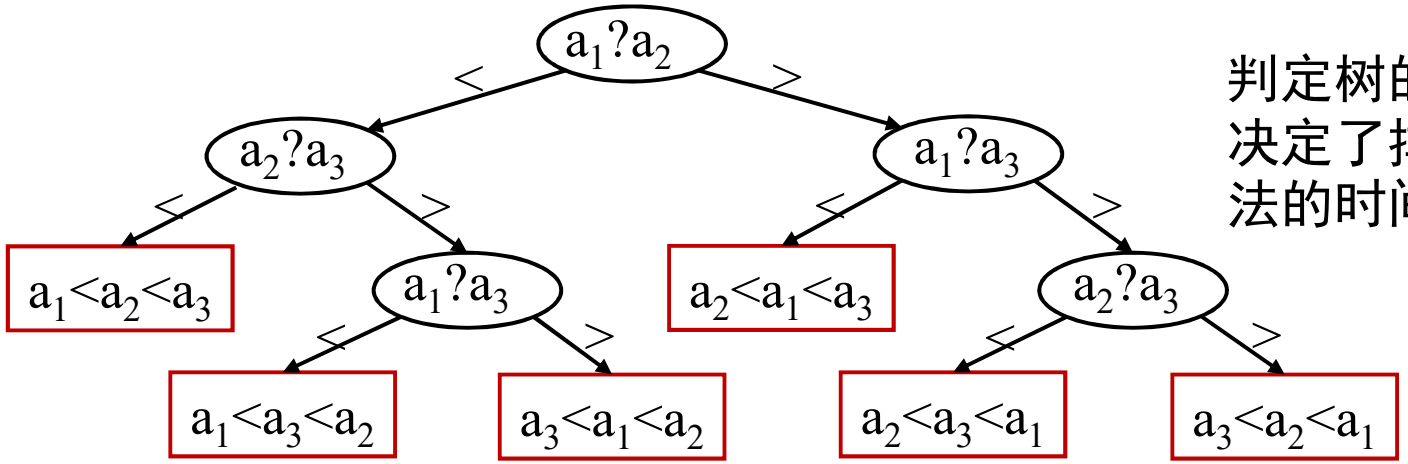
（二路）归并排序分治算法

- 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



判定树的高度决定了排序算法的时间性能

  - 在判定树的某个内部结点上，进行a_i与 a_j比较时，
  - 若$a_i < a_j$，则转向其左子树：表示当$a_i < a_j$时，要进行的其他比较；
  - 若$a_i > a_j$，则转向其右子树：表示当$a_i > a_j$时，要进行的其他比较；
  - 叶子结点表示（所有可能）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$N \leq 2^{h-1}$，亦即$h \geq \log_2 N + 1$
- 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个不同数有$n!$个排列，所以对n个不同数的比较排序，就有$n!$个可能输入。
  - 而对这n个不同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$n>1$，$n! \geq n(n-1)\ldots n/2 \geq (n/2)^{n/2}$
  - 所以，取$n_0=4$，C=1/4，当$n \geq n_0$时，有

$$\log_2(n!) \geq (n/2)\log_2(n/2) \geq (n/4)\log_2 n$$

  - 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$\Omega(n\log_2 n)$
  - 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)。
  - 把关键字各个分量所有可能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遍处理时，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[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

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

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

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) ;
}
```

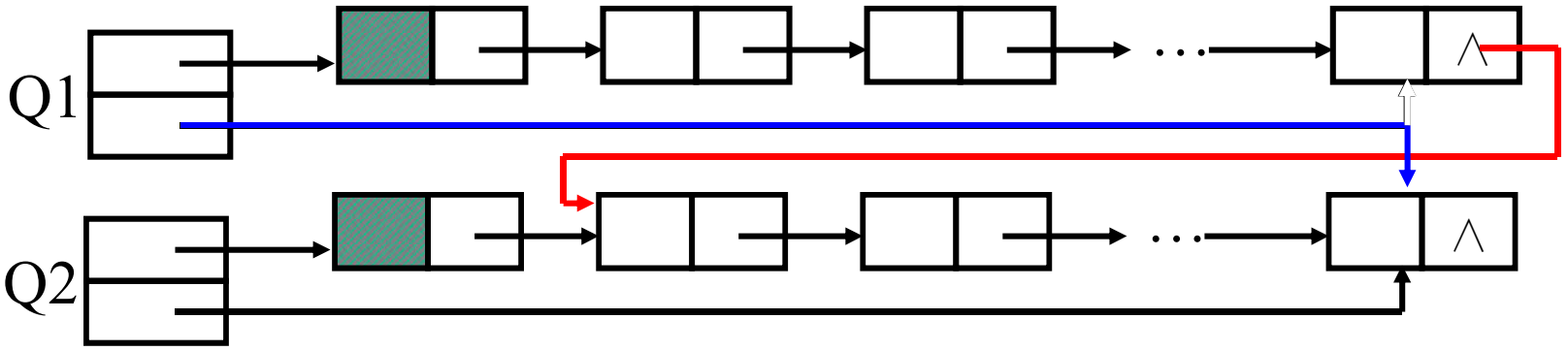
```
for (i=1;i<=9;i++) { //收集
     Concatenate(Q[0], Q[i]);
      A=Q[0];
}//缩短收集操作的时间:O(r)
```

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

7.1 磁盘文件归并排序

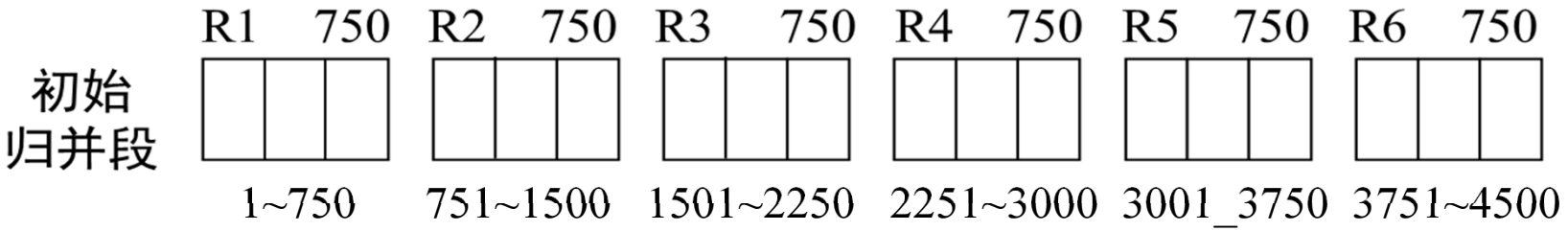
- 外部排序概念
  - 在排序过程中，数据的主要部分存放在外存储器上，借助内存储器，来调整外存储器上数据的位置。
- 外部排序归并方法：两阶段过程
  - 第一阶段：形成初始归并段
    - 首先，将文件中数据分段输入到内存，在内存中采用内部排序方法对其进行排序，然后将有序段写回外存。
    - 排序完的文件段，称为归并段(run)。
    - 整个文件经过在内存逐段排序又逐段写回外存，这样在外存中形成多个初始归并段。
  - 第二阶段：多路归并
    - 对初始归并段采用某种归并排序方法，进行多遍归并，最后整个文件成为单一归并段（整个文件有序）。

7.1 磁盘文件的归并排序

- 外部归并排序重点研究问题
  - 如何减少待排文件的归并遍数?
    - 多路(平衡)归并
  - 如何巧妙运用内存的缓冲区使I/O和CPU尽可能并行工作?
    - 并行操作缓冲区处理
  - 如何增加初始归并段长度?
    - 初始归并段生成方法
  - 如何将初始归并段更有效地归并?
    - 初始归并段的归并顺序
- 分类:
  - 磁盘和磁带归并排序
  - 磁盘是随机存储设备
  - 磁带是顺序存储设备

7.1 磁盘文件的归并排序

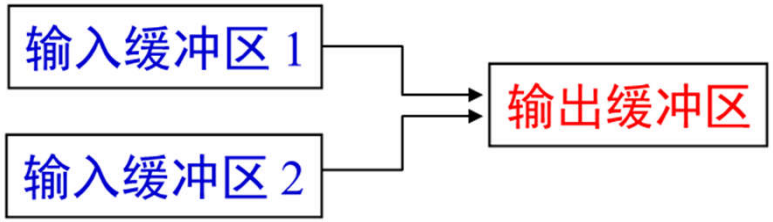
- 归并排序示例：
  - 设有一个包含4500个记录的输入文件。
  - 现用一台其内存至多可容纳750个记录的计算机对该文件进行排序。
  - 输入文件放在磁盘上，磁盘每个页块可容纳250个记录，这样全部记录可存储在 4500 / 250＝18 个页块中。
  - 输出文件也放在磁盘上，用以存放归并结果。
- 第一阶段：形成初始归并段
  - 内存可用存储容量750 个记录，故内存恰好能存3个页块的记录。
  - 外排序一开始，把18块记录，每3块一组，读入内存。
  - 利用某种内排序方法进行内排序，形成初始归并段，再写回外存。
  - 总共可得到6个初始归并段。
- 第二阶段：2 路归并
  - 一趟一趟进行归并排序。



7.1 磁盘文件的归并排序

第二阶段： 2 路归并

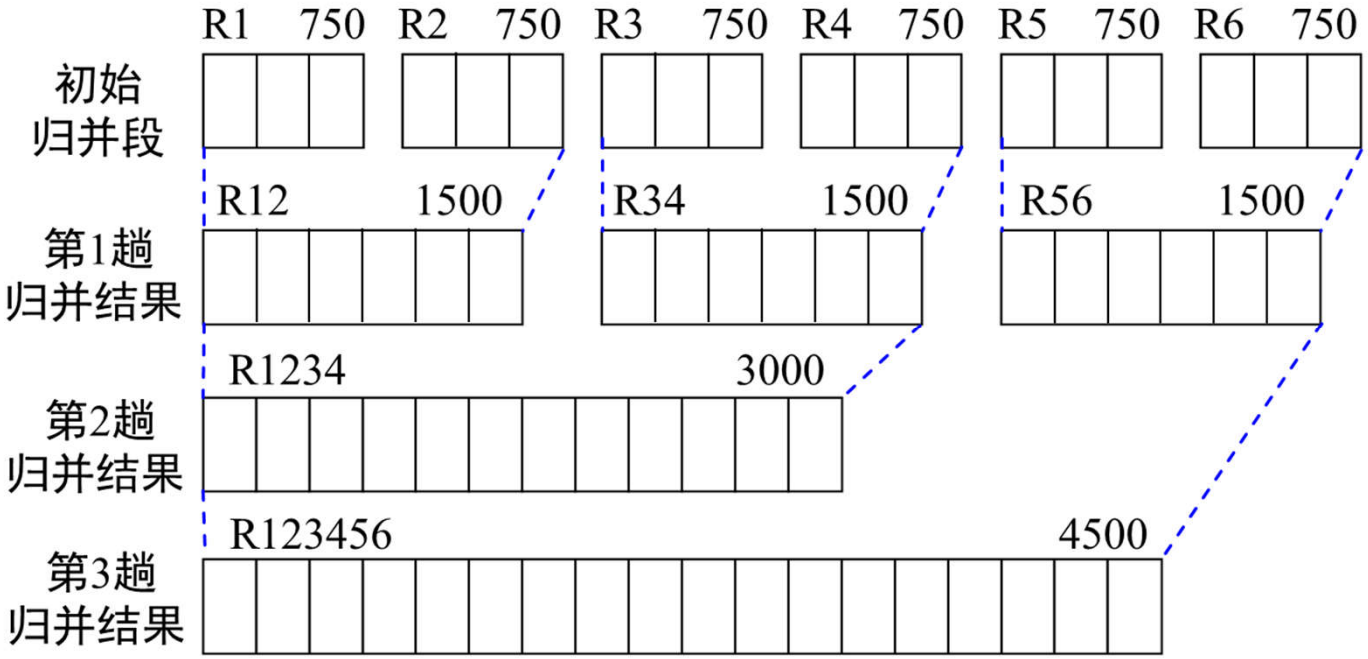
- 把内存区域等分为 3 个缓冲区：
  - 两个为输入缓冲区
  - 一个为输出缓冲区
- 利用简单 2 路归并函数 MergeSort( ) 实现 2 路归并
  - 首先, 从两个输入归并段 *R*1 和 *R*2 中，分别读入一块，放入输入缓冲区1 和输入缓冲区2 。
  - 然后在内存中进行 2 路归并，归并结果顺序存放到输出缓冲区。
  - 当输出缓冲区装满250个记录时，就输出到磁盘。



7.1 磁盘文件的归并排序

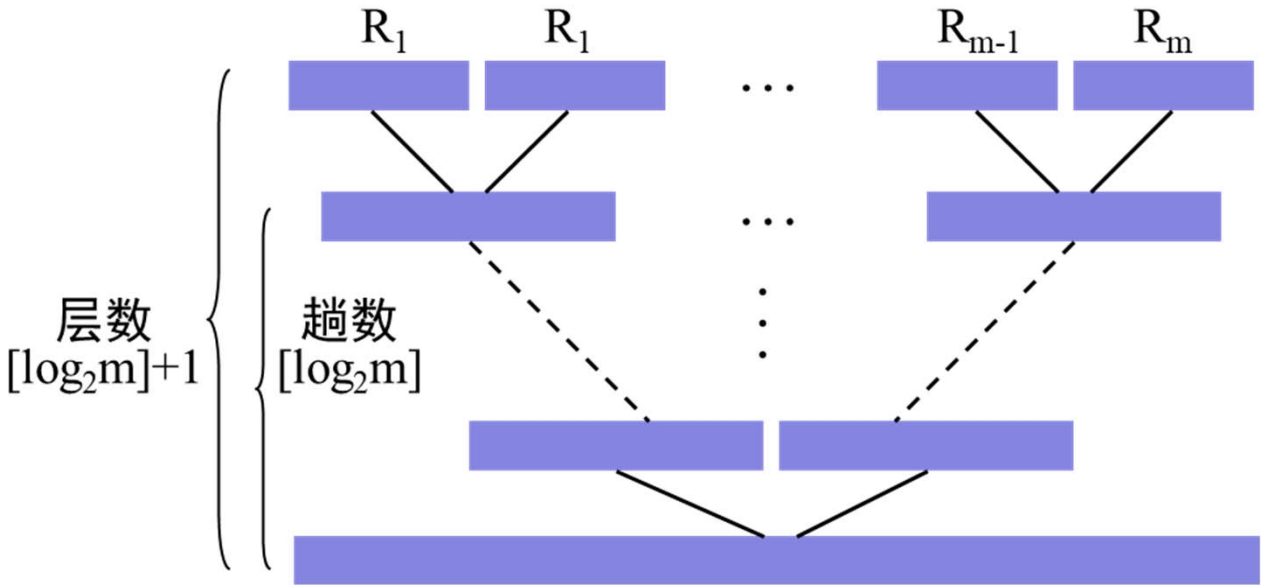
第二阶段： 2 路归并

- 如果归并期间某个输入缓冲区空了，就立即向该缓冲区继续装入所对应归并段的一块记录信息，使之与另一个输入缓冲区剩余记录归并，直到R1和R2归并为R12、R3和R4归并为R34、R5和R6归并为R56为止。
- R12和R34归并为R1234，最后R1234和R56归并为R123456。



7.1 磁盘文件的归并排序

- 归并的趟数
  - m个初始段，进行2路归并，需要$\lceil \log_2 m \rceil$趟归并。
  - m个初始段，采用K路归并，需要$\lceil \log_K m \rceil$趟归并。
  - K越大，归并趟(遍)数越少，可以提高归并的效率。



m 个归并段的归并过程

7.1 磁盘文件的归并排序

（1）多路归并：可减少归并趟数，但能增加比较次数

- K路归并时，需从 K 个关键字中选择最小记录，要比较K-1次。若记录总数为 n，每趟要比较 n*(K-1)次，$\lceil \log_K m \rceil$趟要比较的次数为（ m 为归并段数量）：

$$n*(K-1)\lceil \log_K m \rceil = n*(K-1)\lceil \log_2 m / \log_2 K \rceil$$

- 可见，随着归并路数K增大，$(K-1)/\log_2 K$ 也增大， CPU 处理时间(比较次数)也随之增多。
- 当K值增大到一定程度时，能使CPU处理时间大于因K值增大而减少归并趟数所节省的时间。
- 多路归并应考虑：
  ① 选择恰当的归并路数，以减少排序中比较次数
  ② 选择好的初始归并段形成方法，增大归并段长度 (即减少初始归并段数量)，从而提高归并排序效率。

7.1 磁盘文件的归并排序

（2）K 路平衡归并：胜者树或败者树

- 第一次建立选择树的比较时间开销为:

$$O(K-1) = O(K)$$

- 每次重构选择树所需时间开销为：$O(\log_2 K)$
- n 个记录一趟归并处理时间为：初始建立选择树的时间加上 n-1 次重建选择树的时间开销

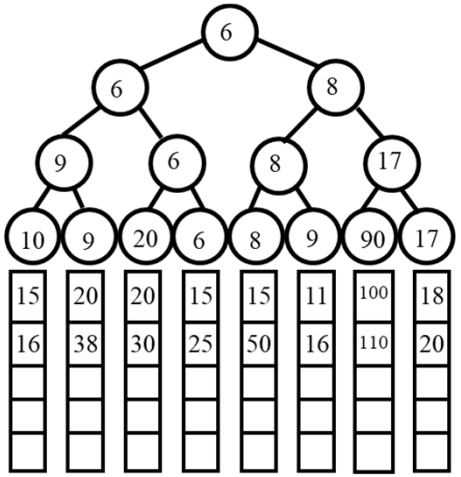
$$O((n-1)\cdot\log_2 K)+O(K) = O(n\cdot\log_2 K)$$

- 上式为K路归并一趟所需CPU处理时间，归并趟数为 $\log_K m$，总时间为：

$$O(n\cdot\log_2 K\cdot\log_K m) = O(n\cdot\log_2 m)$$

/* m 为归并段数量 */

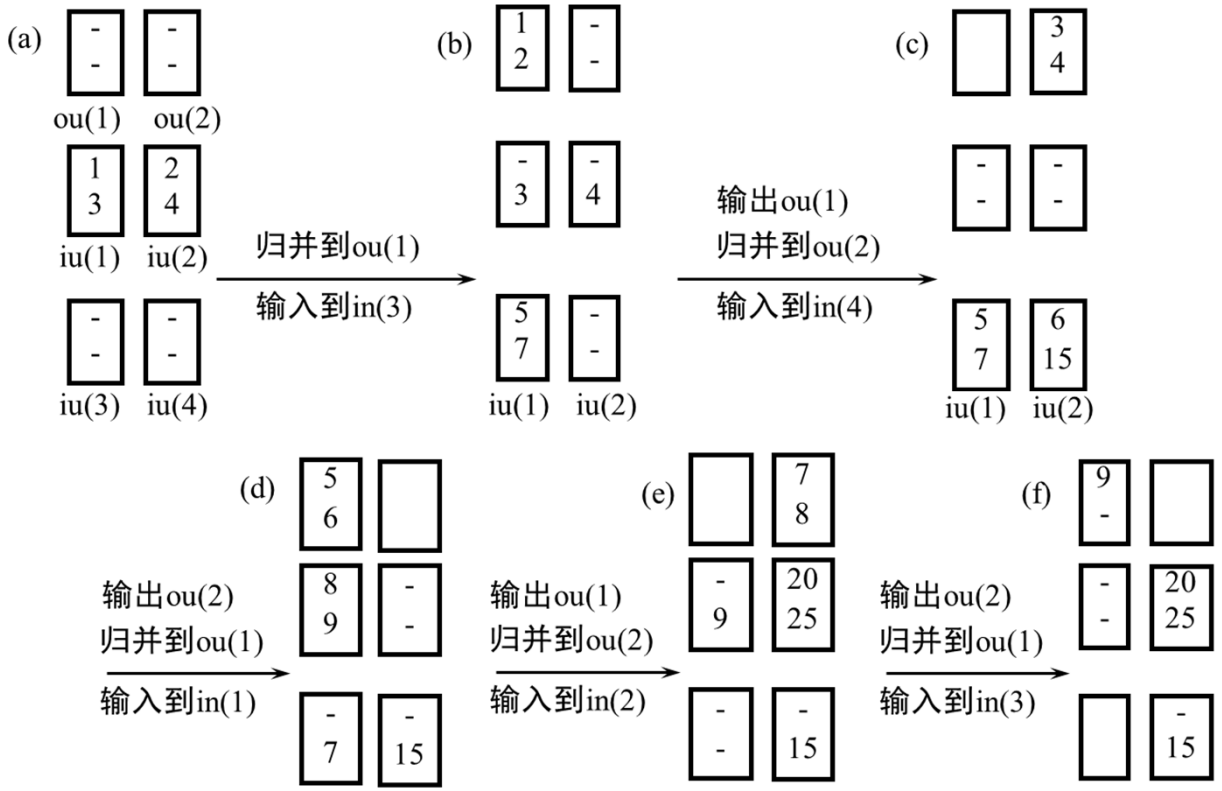
K 路归并时间开销与 K 无关!



7.1 磁盘文件的归并排序

（3）缓冲区处理并行操作：I/O和 CPU 处理尽可能重叠

- K个归并段进行 K 路归并至少需K个输入和1个输出缓冲区。
- 需要2(K+1)个缓冲区实现并行操作，输入、输出和归并可同时进行。



7.1 磁盘文件的归并排序

（4）初始归并段生成：尽量增加初始归并段长度，从而减少归并段数量

- 任何内部排序算法都可作为生成初始归并段
- 初始归并段长度 ≥ 缓冲区长度

- 置换-选择排序：选择树法
  - 设初始待排序文件为输入文件FI，初始归并段文件为输出文件FO，内存缓冲区为W，可容纳P个记录。FO与W初始为空，则置换-选择过程：

(1) 从FI输入P个记录到缓冲区W;

(2) 从W中选择出关键字最小的记录MIN；

(3) 将MIN记录输出到FO中去；

(4) 若FI不空，则从FI输入下一个记录到W;

(5) 从W中所有比MIN关键字大的记录中，选出最小关键字记录，作为新的MIN；/* W中比最后输出记录关键字小的暂不能选为当前归并段记录，等待生成下一个归并段。*/

(6) 重复(2)~(5)，直到W中选不出新MIN为止，则得到一个初始归并段，输出归并段结束标志到FO中。

(7) 重复(2)~(6)，直到W为空，由此得到全部初始归并段。

7.1 磁盘文件的归并排序

- 示例：缓冲区W的长度P=4，输入序列为：

  15 19 04 83 12 27 11 25 16 34 26 07 10 90 06 …

- 注意：如果新输入记录的关键字小于最后输出记录的关键字，则新输入记录不能成为当前归并段的一部分。它要等待作为生成下一个归并段候选。
- 采用选择树法生成初始归并段的平均长度是缓冲区长度的两倍。

步	1	2	3	4	5	6	7	8	9	10	11	12	13	…
缓冲区内容	15	15	15	(11)	(11)	(11)	(11)	(11)	(11)	11	11	(06)	…	…
	19	19	19	19	25	(16)	(16)	(16)	(16)	16	16	16	…	…
	04	12	27	27	27	27	34	(26)	(26)	26	26	26	…	…
	83	83	83	83	83	83	83	83	(07)	10	90	90	…	…
输出结果	04	12	15	19	25	27	34	83	07	10	11	16	…	
	R_1								R_2					

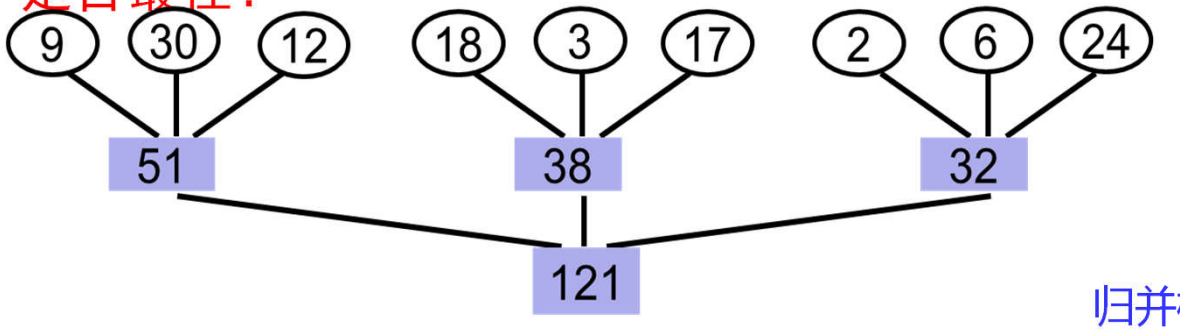
7.1 磁盘文件的归并排序

（5）最佳归并树：使外存读写次数最少

- 由置换-选择排序所得初始归并段的长度可能不等，这对于多路平衡归并将产生什么影响？
- 例：假设经置换-选择排序先后得到的归并段长度分别为:

  9，30，12，18，14，25，31，7，27
- 每个记录占一个物理块，则进行3-路平衡归并排序时访问外存的次数是多少？
- 三叉树的带权路径长度：

  $WPL=(9+30+12+18+3+17+2+6+24)\times 2=242$
- 访问外存次数：$242\times 2=484$ (读/写各1次)
- 是否最佳？

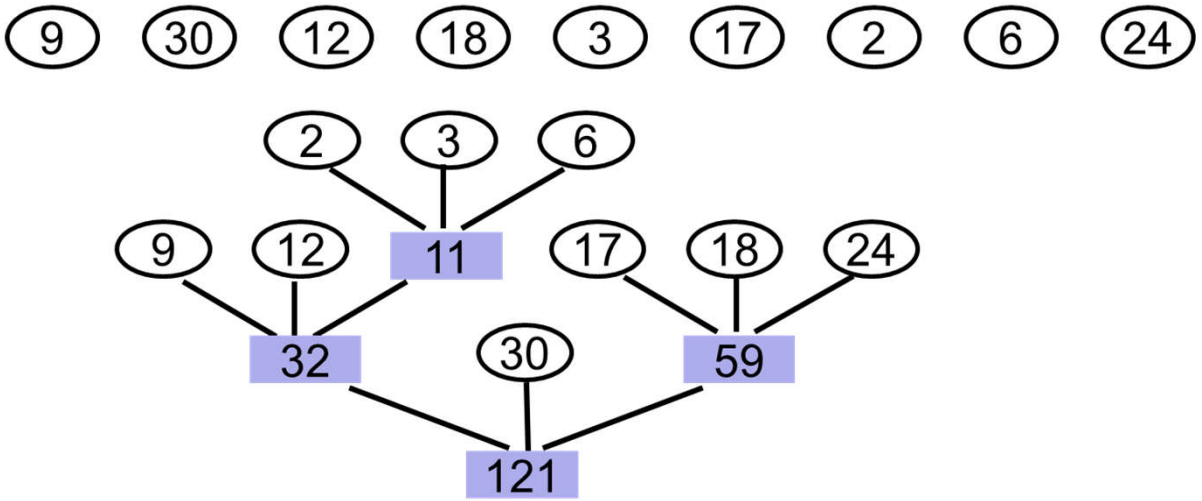


归并树

7.1 磁盘文件的归并排序

- **最佳归并树**
  - 最佳归并树的带权路径长度：

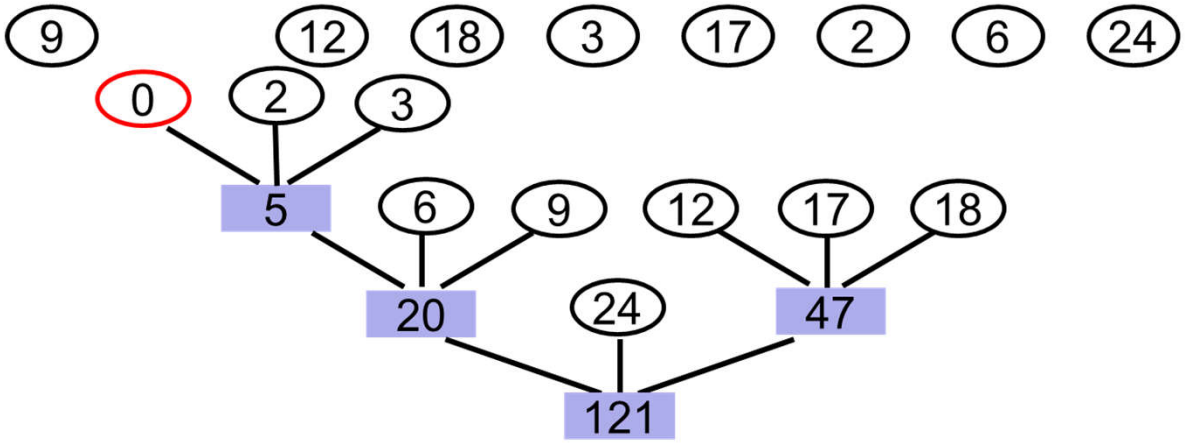
    WPL=(2+3+6)×3+(9+12+17+18+24)×2+30 × 1= 223
  - 访问外存次数：223×2＝446 (读/写各1次)



- 如果去掉一个长度为30的初始归并段，求最佳归并树？
  - WPL=(2+3+5)×3+(21+59)×2= 193
  - 访问外存次数： 193 × 2＝386

7.1 磁盘文件的归并排序

- 最佳归并树：
  - WPL = (2+3)×3+(15+47)×2+24 ×1 = 163
  - 访问外存次数：163 × 2＝326



- 问题：当初始归并段的数目不足时，怎样求最佳归并树？
  - 最佳归并树应该是一棵“正则树”
  - 对 K 路归并而言，设初始归并段为 m，若：

    (m-1) % (K-1) ＝0

    则不需要加虚段；否则，需要加虚段的个数为：

    K-(m-1) % (K-1)-1

7.2 磁带文件的归并排序

- K路平衡归并排序
  - 磁带机数量：2K
  - 与磁盘不同，磁带是顺序存储设备，读取信息块的时间与信息块位置有关。
  - 磁带排序需要了解信息块分布。

输入：T_1, T_2, ……, T_k　　输出

输出：T_{k+1}, T_{k+2}, ……, T_{2k}　　输入

磁带机	T_1	T_2	…	T_k
归并段	R_1	R_2	…	R_k
	R_{k+1}	R_{k+2}	…	R_{2k}
	…	…	…	…
	R_{mk+1}	…	…	…

T_1:R_1(1000),R_3(1000),R_5(1000)
T_2:R_2(1000),R_4(1000),R_6(1000)
T_3:Ø
T_4:Ø

T_1: Ø
T_2: Ø
T_3: R_1(2000),R_3(2000)
T_4: R_2(2000)

T_1: R_1(4000)
T_2: R_2(2000)
T_3: Ø
T_4: Ø

→

T_1: Ø
T_2: Ø
T_3: R_1(6000)
T_4: Ø

7.2 磁带文件的归并排序

- 多阶段归并排序
  - K+1台磁带机，实现 k 路归并
- K阶Fibonacci序列：

$$
\begin{aligned}
F_n^{(k)} &= 0 && 0<=n<=K-2\\
F_n^{(k)} &= 1 && n=K-1\\
F_n^{(k)} &= F_{n-1}^{(k)} + F_{n-2}^{(k)} + \cdots + F_{n-k}^{(k)} && n>=K
\end{aligned}
$$

- K+1台磁带机K路多阶归并段归并，在n-j步归并段数分布规律：

$$
\begin{aligned}
t_1^j &= F_{j+k-2}^{(k)}\\
t_2^j &= F_{j+k-3}^{(k)} + F_{j+k-2}^{(k)}\\
&\cdots\\
t_{k-1}^j &= F_j^{(k)} + F_{j+1}^{(k)} + \cdots + F_{j+k-2}^{(k)}\\
t_k^j &= F_{j-1}^{(k)} + F_j^{(k)} + \cdots + F_{j+k-2}^{(k)}
\end{aligned}
$$

- 第j步归并段总数：

$$
F_{G(j+k-2)}^{(k)} = t_1^j + t_2^j + \cdots + t_{j+k-2}^{(k)}
$$

i 遍后	t_1	t_2	t_3
开始	13(1L)	21(L)	空
1	空	8(1L)	13(2L)
2	8(3L)	空	5(2L)
3	3(3L)	5(5L)	空
4	空	2(5L)	3(8L)
5	2(13L)	空	1(8L)
6	1(13L)	1(21L)	空
7	空	空	1(34L)

步	t_1	T_2	t_3	总段数
n	0	0	1	1
n-1	1	1	0	2
n-2	2	0	1	3
n-3	0	2	3	5
n-4	3	5	0	8
n-5	8	0	5	13
n-6	0	8	13	21
n-7	13	21	0	34

- 若初始段数不是K阶Fibonacci，则必须附加一些虚的归并段，凑成K阶Fibonacci

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：

  ⑴时间复杂度；

  ⑵空间复杂度；

  ⑶稳定性；

  ⑷算法简单性；

  ⑸待排序记录个数n的大小；

  ⑹记录本身信息量的大小；

  ⑺关键字值的分布情况。

本章小结：各种排序方法的比较(cont.)

排序算法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，这些算法相对比较复杂。

- 待排序的记录个数n比较：
  - 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，
  - 对记录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	$O(n)$	$O(n)$

- 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，这是最坏情况，此时时间性能蜕化为$O(n^2)$；
  - 选择排序、堆排序和归并排序的时间性能不随记录序列关键字的分布而改变。

本章小结：外部排序

- 外部排序主要研究的技术问题:
  - 如何进行多路归并以减少文件的归并遍数；
    - K越大，归并趟数越少，可提高归并的效率，但可能增加整个归并过程的比较次数：
      $$n*(K-1)\lceil \log_K m \rceil = n*(K-1)\lceil \log_2 m/\log_2 K \rceil$$
  - 如何运用内存的缓冲区使I/O和CPU尽可能并行工作；
  - 根据外存的特点选择较好的产生初始归并段的方法；
  - 最佳归并树：最小化多路平衡归并的外存读写次数。
- 多路平衡归并：选择树法
  - 建立选择树的时间: $O(K-1) = O(K)$
  - 重构选择树的时间：$O(\log_2 K)$
  - 一趟归并的时间： $O((n-1)\cdot \log_2 K)+O(K) = O(n\cdot \log_2 K)$
  - 总的归并时间： $O(n\cdot \log_2 K\cdot \log_K m) = O(n\cdot \log_2 m)$

考试信息

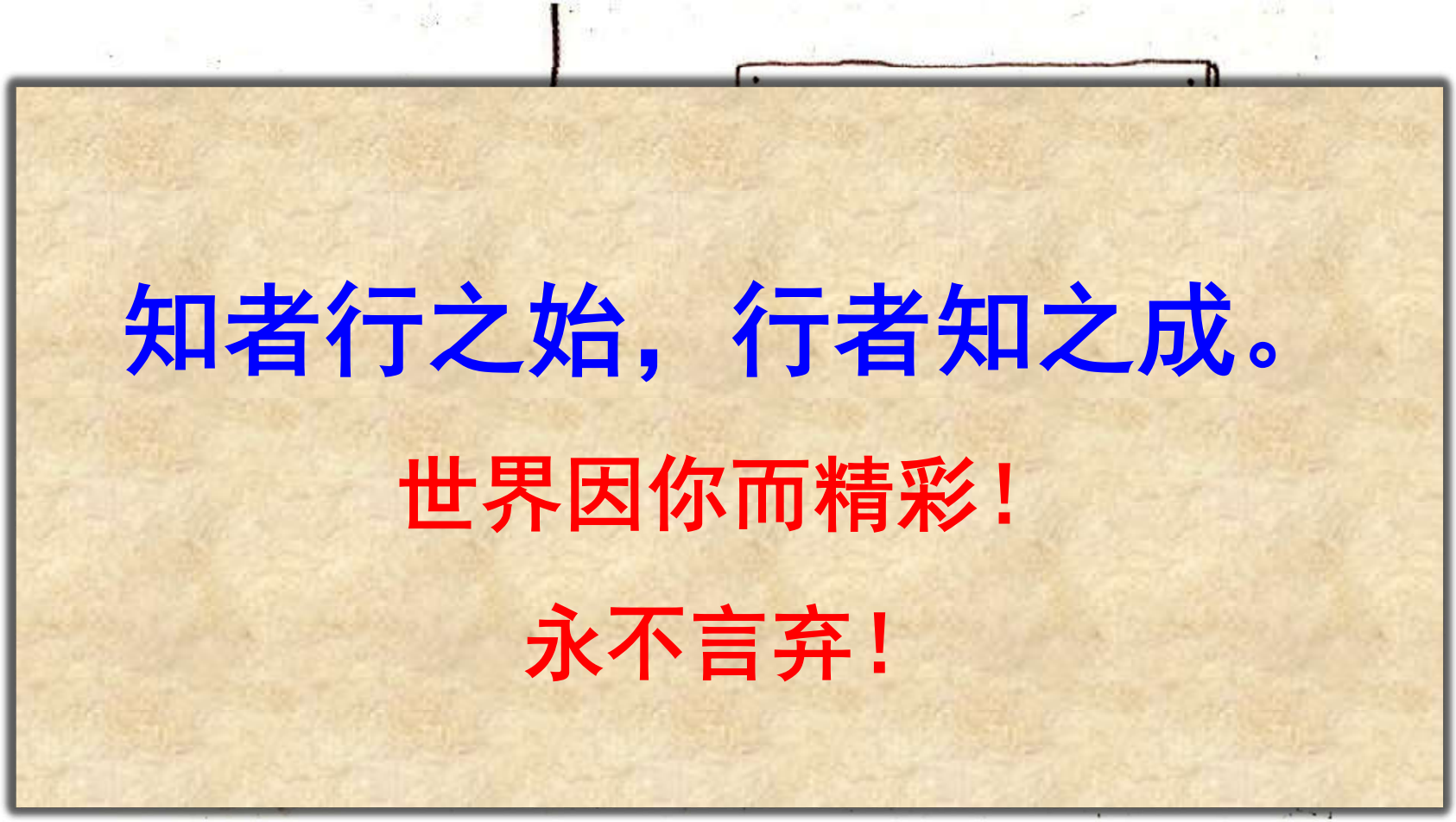
- 考试时间：2021.11.27，周六，10:00-12:00，第12周
- 考试地点：正心楼

CS32132	数据结构与算法	2021-11-27	第12周 星期六10:00-12:00	一校区	正心23
CS32132	数据结构与算法	2021-11-27	第12周 星期六10:00-12:00	一校区	正心24
CS32132	数据结构与算法	2021-11-27	第12周 星期六10:00-12:00	一校区	正心31
CS32132	数据结构与算法	2021-11-27	第12周 星期六10:00-12:00	一校区	正心32
CS32132	数据结构与算法	2021-11-27	第12周 星期六10:00-12:00	一校区	正心33
CS32132	数据结构与算法	2021-11-27	第12周 星期六10:00-12:00	一校区	正心34
CS32132	数据结构与算法	2021-11-27	第12周 星期六10:00-12:00	一校区	正心41
CS32132	数据结构与算法	2021-11-27	第12周 星期六10:00-12:00	一校区	正心42

" Computers do not solve problems, they execute solutions " — Laurent Gasser

"The real problem is not whether machines think but whether men do." — B. F. Skinner

"Scientists investigate that which already is; Engineers create that which has never been." — Albert Einstein.


知者行之始，行者知之成。
世界因你而精彩！
永不言弃！