数据结构与算法

主讲教师： 刘峤

第8章 排序

第9章 内容提要

9.1 插入排序

9.2 交换排序

9.3 选择排序

9.4 归并排序

9.5 基数排序

9.6 排序方法比较

排序的基本概念

- 排序的定义
  - 将一个数据元素（记录）的任意序列
  - 重新排列成一个按关键字有序的序列，称为排序
- 设：给定一个包含n个记录的序列 { **R1, R2, …, Rn** }
  - 其相应的关键字序列为 { **K1, K2, …, Kn** }
  - 这些关键字之间存在偏序关系（可相互比较）

$$K_{p1} \le K_{p2} \le \dots \le K_{pn}$$

  - 按此偏序关系将上式记录序列重新排列为

$$\{ R_{p1}, R_{p2}, \dots, R_{pn} \}$$

  - 将上述操作称为排序

排序的基本概念

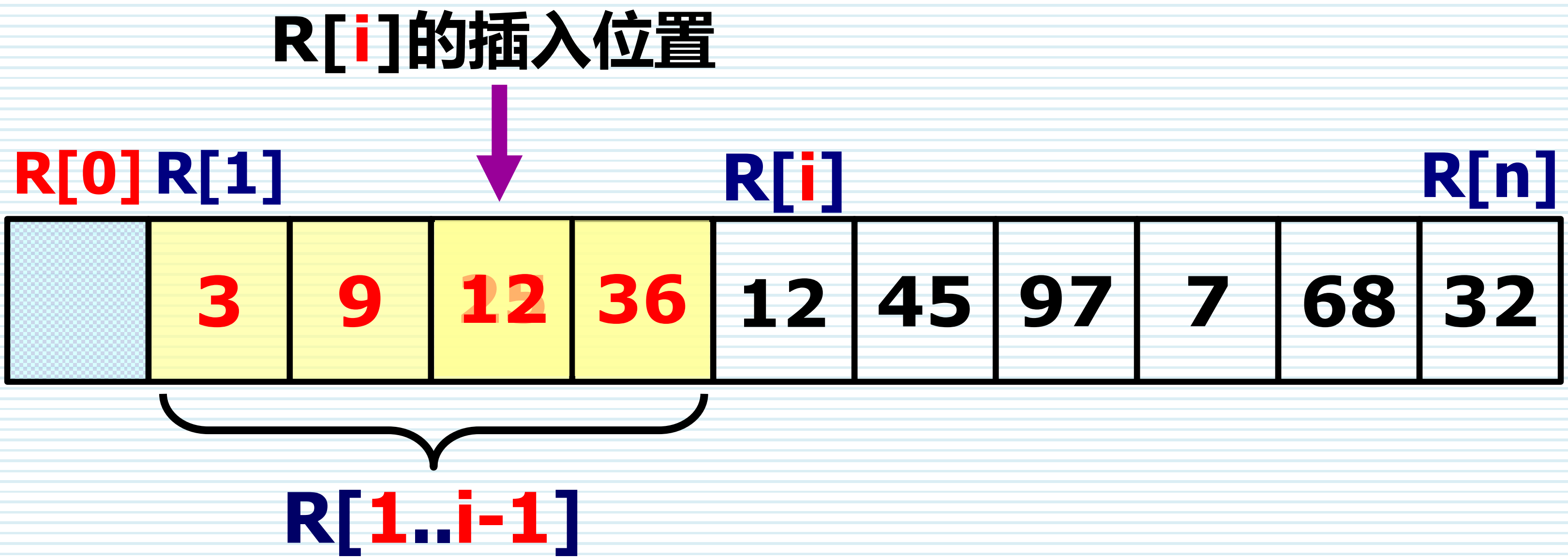
- 内部排序与外部排序
  - 是否存在内外存交换
- 稳定排序和不稳定排序
  - 对于任意的数据元素序列
  - 若在排序前后相同关键字数据的相对位置都保持不变
  - 这样的排序方法称为稳定的排序方法
  - 否则称为不稳定的排序方法
  - 例如：对于关键字序列 **3**，**2**，**3**，**4**
    - 若某种排序方法排序后变为 **2**，**3**，**3**，**4**
    - 则此排序方法就不稳定

插入排序

- 直接插入排序
- 折半插入排序
- 二路插入排序（自学）
- 希尔排序

直接插入排序（Insertion Sort）

- 利用**顺序查找**实现：在R[1..i-1]中查找R[i]的插入位置



直接插入排序

R[0]	R[1]					R[i]				R[n]
	3	9	12	25	36	45	97	7	68	32

- 排序算法：整个排序过程由n-1轮插入操作构成
  - 首先将序列中第1个记录看成是一个有序子序列
  - 然后从第2个记录开始，逐个将其插入前面的有序子序列
    - 查找过程中找到的那些关键字不小于**R[i]**的记录
    - 在查找的同时实现记录向后移动
  - 直至整个序列有序

直接插入排序

```
// R为顺序表，len为表长（R[0]闲置)
void insert_sort(int *R, int len){
    int i;
    for(i = 2; i <= len; i++){
        insert (R, i);  // 将R[i]插入到R[1..i]合适的位置上
    }
}
```

直接插入排序

```
// R[1..n-1]为有序表，n为表长
void insert (int *R, int n){
    int pos = n; R[0] = R[n];  // 设置监视哨
    // 从右至左查找第一个比R[n]小的数的位置
    while(R[0] < R[pos-1]) {
        R[pos] = R[pos-1];    // 元素移动
        pos--;
    }
    R[pos] = R[0]; // 将R[n]插入到合适的位置
}
```

直接插入排序算法的性能分析

- 空间性能分析　**S(n)=O(1)**
  - 需要一个辅助空间：R[0]
- 时间性能分析　**T(n)=O(n^2)**
  - 实现直接插入排序的基本操作有两个
    - **比较**：序列中两条记录的关键字大小
    - **移动**：序列中的记录以腾出插入位置

直接插入排序算法的性能分析

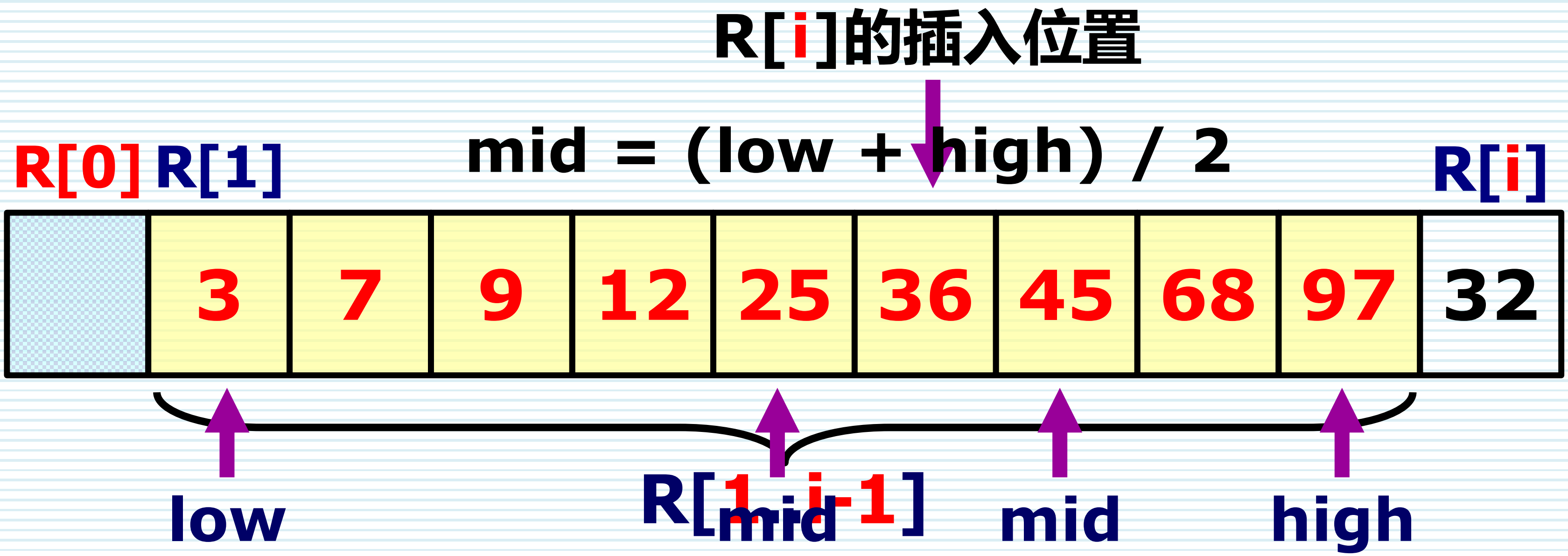
- 时间性能分析(续)　**T(n)=O(n²)**

- 最好的情况：记录序列中关键字按顺序有序
  - 元素比较的次数：$\sum_{i=2}^{n} 1 = n-1$
  - 元素移动的次数：0

- 最坏的情况：记录序列中关键字按逆序有序
  - 元素比较的次数：$\sum_{i=2}^{n} (i-1) = \frac{n(n-1)}{2}$
  - 元素移动的次数：$\sum_{i=2}^{n} (i+1) = \frac{(n+4)(n-1)}{2}$

折半排序

- 利用折半查找实现：在R[1..i-1]中查找R[i]的插入位置



low > high 时，查找结束

元素右移，完成插入

若：R[i] > R[mid] 则：low = mid + 1

若：R[i] ≤ R[mid] 则：high = mid - 1

折半排序

```
// R为顺序表，len为表长（R[0]闲置）
void binary_insert_sort(int *R, int len){
    int i;
    for(i = 2; i <= len; i++){
        binsert (R, i);  // 将R[i]插入到R[1..i]合适的位置上
    }
}
```

折半排序

```
// R[1..n-1]为有序表，n为表长
void binsert(int *R, int n){
    int i, mid, low = 1, high = n-1; R[0] = R[n];
    while(low <= high) {   // 查找待插入位置
        mid = (low + high)/2;
        if(R[0] < R[mid])  high = mid - 1;
        else  low = mid + 1;
    }
    for(i = n; i > low; --i){
        R[i] = R[i-1];  // 元素移动
    }
    R[low] = R[0];     // 将R[n]插入到合适的位置
}
```

折半排序

- 时间性能分析　　　$T(n)=O(n^2)$
  - 最好的情况：记录序列中关键字按顺序有序
    - 元素比较的次数： $n\log_2 n$
    - 元素移动的次数：0
  - 最坏的情况：记录序列中关键字按逆序有序
    - 元素比较的次数： $n\log_2 n$　**(比较次数得到了改善)**
    - 元素移动的次数： $\sum_{i=2}^{n}(i+1)=\frac{(n+4)(n-1)}{2}$

希尔排序算法

希尔排序（Shell Sort）

- 算法描述
  - 将待排序序列分割成若干个较小的子序列
    - 对各个子序列分别执行直接插入排序
  - 当序列达到基本有序时，对其执行一次直接插入排序
- 算法基本思想
  - 对待排记录序列先作宏观调整，再作微观调整
    - 宏观调整：分段执行插入排序
    - 微观调整：对全序列执行一次直接插入排序

希尔排序

{ R[1], R[1+d], R[1+2d], …, R[1+kd] }

{ R[2], R[2+d], R[2+2d], …, R[2+kd] }

… … … … … … … … … … … … …

{ R[d], R[d+d], R[d+2d], …, R[(k+1)d] }

- 例如：将 n 个记录分成 **d** 个子序列：
  - 其中正整数 d 称为增量
    - 它的值在排序过程中从大到小逐渐递减
    - 直至最后一趟排序减为 1

16	25	12	30	47	11	23	36	9	18	31

第一趟希尔排序，设置增量 **d=5**，分为5个子序列

11	23	12	9	18	16	25	36	30	47	31

第二趟希尔排序，设置增量 **d=3**，分为3个子序列

9	18	12	11	23	16	25	31	30	47	36

第三趟希尔排序，设置增量 **d=1**，对整个序列进行排序

9	11	12	16	18	23	25	30	31	36	47

希尔排序

```
// R[]为待排数组；n为待排数组长度（R[0]闲置)
// D[]为增量数组；n为增量数组长度
void shell_sort(int* R, int n, int* D, int m){
    int i, d;
    // 根据增量数组执行m轮分组排序
    for(i = 0; i < m; ++i){
        d = D[i];
        stepwise(R, d, n);
    }
}
```

希尔排序

```
// R[]为待排数组，n为待排数组长度，d为步长
void stepwise(int* R, int d, int n){
    int i, j, k;
    for(i = 1; i <= d; ++i){  // 数据被分成d组
        for(j = i + d; j < n; j += d){
            R[0] = R[j]; k = j;
            while( (k-d) > 0 && (R[0] < R[k-d])) {
                R[k] = R[k-d]; k = k - d;
            }
            R[k] = R[0];
        }
    }
}
```

希尔排序

```
// 修订书上的代码
void stepwise(int* R, int d, int n){
    int i, j, k;
    for(i = 1; i <= d; ++i){ // 数据被分成d组
        for(j = i + d; j < n; j += d){
            R[0] = R[j]; k = j;
            for(k = j; k-d > 0 ; k = k - d){
                if(R[0] < R[k-d]) R[k] = R[k-d];
                else break;
            }
            R[k] = R[0];
        }
    }
}
```

希尔排序

- 算法设计依据
  - 进行插入排序时：若待排序序列“基本有序”
    - 即：序列中具有如下特性的记录数较少

$$R[i] < \max\{ R[k] : 1 \leq k < i \}$$

    - 则序列中大多数记录都不需要进行插入和元素移动
  - 当序列基本有序时，直接插入排序的效率可大幅提高
    - 空间复杂度为：**O（1）**
    - 时间复杂度接近：**O（n）**

希尔排序

- 为什么希尔排序可提高排序速度?
  - 分组内采用直接插入排序，元素比较次数近似为：$O(n^2)$
    - 从元素比较次数来看：分组后n值减小，n^2更小
    - 从总体上看：元素比较次数大幅减少（$n^2<<N^2$）
  - 将相隔某个增量的记录组成一个子序列
    - 关键字较小的记录跳跃式前移
    - 最后一轮增量为1的插入排序时，数组已基本有序
    - 所以从总体上看元素移动次数减少
  - 希尔排序的时间复杂度近似等于：$\mathbf{O(\ n^{1.3}\ )}$

希尔排序性能分析

- 希尔排序时需要一个存储单元的辅助空间：S(n)=O(1)
- 希尔排序的时间性能与增量因子d_i 有直接关系
  - 取不同步长的时间复杂度不一样（目前尚无最佳取法）
  - Shell建议：$\mathbf{d_1=\lfloor n/2 \rfloor}$，$\mathbf{d_{i+1} = \lfloor d_i/2 \rfloor}$
  - 但必须满足：最后一个步长一定为1
- 希尔排序是一种不稳定的排序方法
  - 例如：对序列**{4，7，2，9，5，3，2}** 执行希尔排序
  - 结果：**{ 2，2，3，4，5，7}**

交换排序

- 冒泡排序
- 快速排序

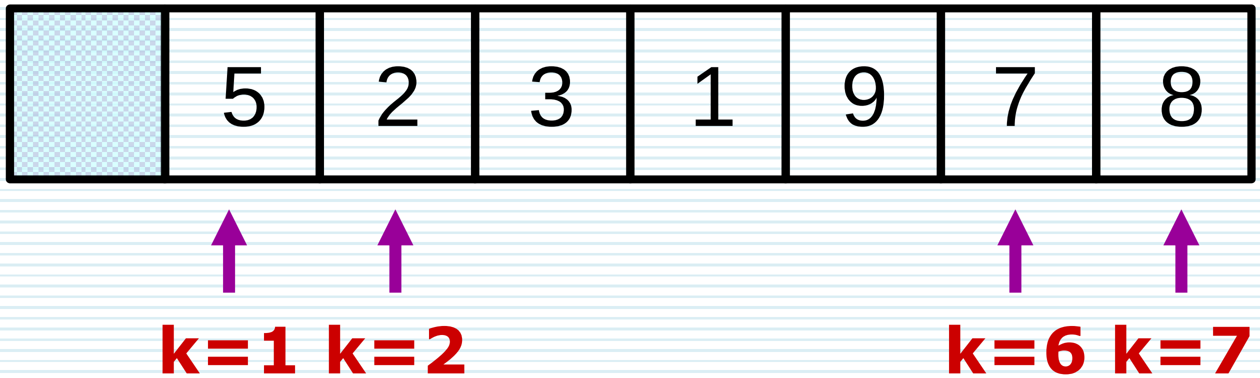
冒泡排序（Bubble Sort）

- 第一趟冒泡
  - 将第一个记录与第二个记录的关键字进行比较
    - 若为逆序：r[1].key>r[2].key，则交换记录值
  - 然后比较第二个记录与第三个记录，依次类推……
    - 直至第n-1个记录和第n个记录比较为止
  - 结果使关键字最大的记录被安置在最后一个记录上
- 对前n-1个记录进行第二趟冒泡排序
  - 结果使关键字次大的记录被安置在第n-1个记录位置
- 重复上述过程
  - 直到在一趟排序过程中没有进行过交换记录的操作为止

冒泡排序

```
// 待排序序列为R[1..n](R[0]闲置)
void bubble_sort (int* R, int n){
    int i, j, flag;
    for( i = 1; i < n; ++i ){
        flag = 0;  // 元素交换标志
        for( j = 1; j <= (n-i); ++j ){
            if( R[j] > R[j+1] ){
                R[0] = R[j+1];  R[j+1] = R[j];
                R[j] = R[0];  flag = 1; // 发生交换
            }
        }
        if( flag == 0) break;
    }
}
```

冒泡排序算法的改进



- 冒泡排序的结束条件为：最后一轮没有发生“记录交换”

  for (j = 1; j <= n-i; j++) if (R[j+1] < R[j]) {元素交换}

- 一般情况下：每经过一轮冒泡，k=(n-i) 的值减1

  - 但并不是在任何情况下都需要逐一递减 **k**！

改进的冒泡排序算法

```
// 待排序序列为R[1..n](R[0]闲置)
void bubble_sort (int* R, int n){
    int i = n,  j,  idx; // 本轮发生交换的最后一个记录的位置
    while( i > 1 ) {
        idx = 1;
        for( j = 1; j < i; ++j ){
            if( R[j] > R[j+1] ){
                R[0] = R[j+1];  R[j+1] = R[j];
                R[j] = R[0];  idx = j; // 发生交换
            }
        }
        i = idx;
    }
}
```

冒泡排序法算法的特点

- 算法的空间复杂度：**S(n)=O(1)**
- 算法的时间复杂度：**T(n)=O(n^2)**
  - 最好情况下（正序）
    - 比较次数： **n-1 次**
    - 移动次数： **0 次**
  - 最坏情况下（逆序）
    - 比较次数： $\sum_{i=1}^{n-1}(n-i)=\frac{1}{2}(n^2-n)$
    - 移动次数： $3\sum_{i=1}^{n}(n-i)=\frac{3}{2}(n^2-n)$

快速排序算法

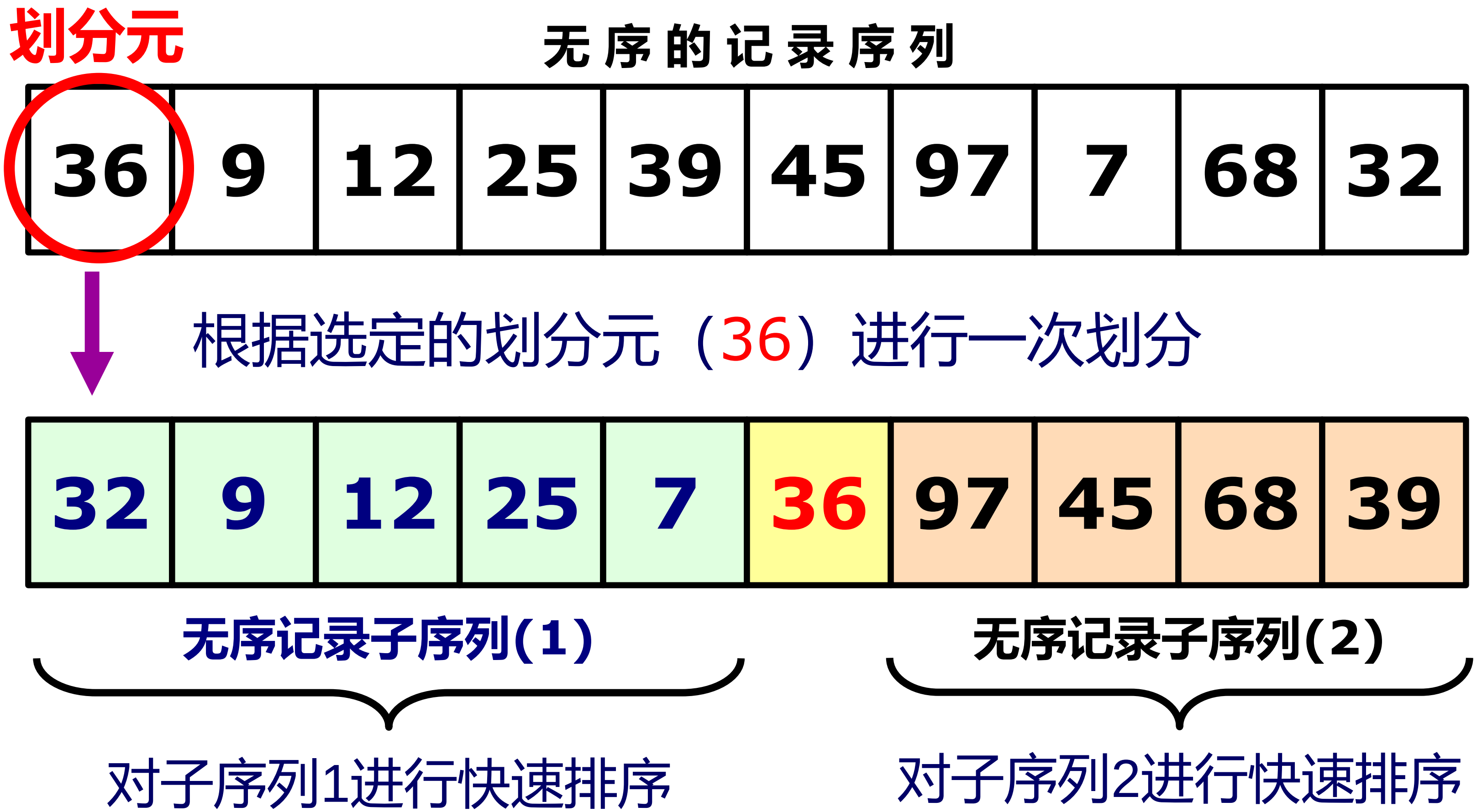
快速排序（Quick Sort）

算法基本思想

- 在数组中确定一个记录（的关键字）作为“**划分元**”
- 将数组中关键字小于划分元的记录均移动至该记录之前
- 将数组中关键字大于划分元的记录均移动至该记录之后
- 由此：一趟排序之后，序列R[s...t]将分割成两部分
  - R[s ... i-1] 和 R[i+1 ... t]
  - 且满足：R[s ... i-1]≤ R[i]≤ R[i+1...t]
  - 其中：R[i] 为选定的“**划分元**”
- 对各部分重复上述过程，直到每一部分仅剩一个记录为止

快速排序（Quick Sort）

- 首先对无序的记录序列进行一次划分
- 之后分别对分割所得两个子序列“递归”进行快速排序



快速排序算法流程

36	32	9	12	25	7	36	97	45	68	39

R[0]　low　　　　　　　　　　　　　　　　high

- 首先：设 R[s]=36 为划分元，将其暂存到R[0]
- 比较 R[high] 和划分元的大小，要求：R[high] ≥ 划分元
- 比较 R[low] 和划分元的大小，要求：R[low] ≤ 划分元
- 若条件不满足，则交换元素，并在low-high之间进行切换
- 一轮划分后得到：（32,9,12,25,7）36（97,45,68,39）

快速排序算法特点

- 时间复杂度：最好情况
  - T(n)=O(n logn) （每次总是选到中间值作划分元）
- 时间复杂度：最坏情况
  - $T(n)=O(n^2)$ （每次总是选到最小或最大元素作划分元）
  - 解决方案：三者取中，或者随机选取划分元
    - 三者取中：设首记录为R[h]、尾记录为R[t]
    - 取： R[h]、R[t]、R[(h+t)/2] 的中间值为划分元
- 快速排序算法的平均时间复杂度为：O(nlogn)
- 快速排序算法是不稳定的
  - 例如待排序序列： **49** 49 38 65
  - 快速排序结果为： 38 49 **49** 65

快速排序（Quick Sort）

```
void quicksort ( int R[], int low, int high) {
    int idx;
    if(low < high){
        // 调用划分过程将R一分为二，以idx保存“划分元”的位置
        idx = partition(R, low, high);
        quicksort (R, low, idx-1);    // 对低端序列递归
        quicksort (R, idx+1, high);  // 对高端序列递归
    }
}
```

快速排序（Quick Sort）

```
int partition(int R[], int low, int high){
    R[0] = R[low];    // 暂存划分元
    while(low < high){
        while( (low < high) && ( R[high] >= R[0] ) ) high--;
        if( low < high ){
            R[low] = R[high]; low++;
        }
        while( (low < high) && (R[low] <= R[0]))  low++;
        if( low < high ) {
            R[high] = R[low]; high--;
        }
    }
    R[low] = R[0];
    return low;
}
```

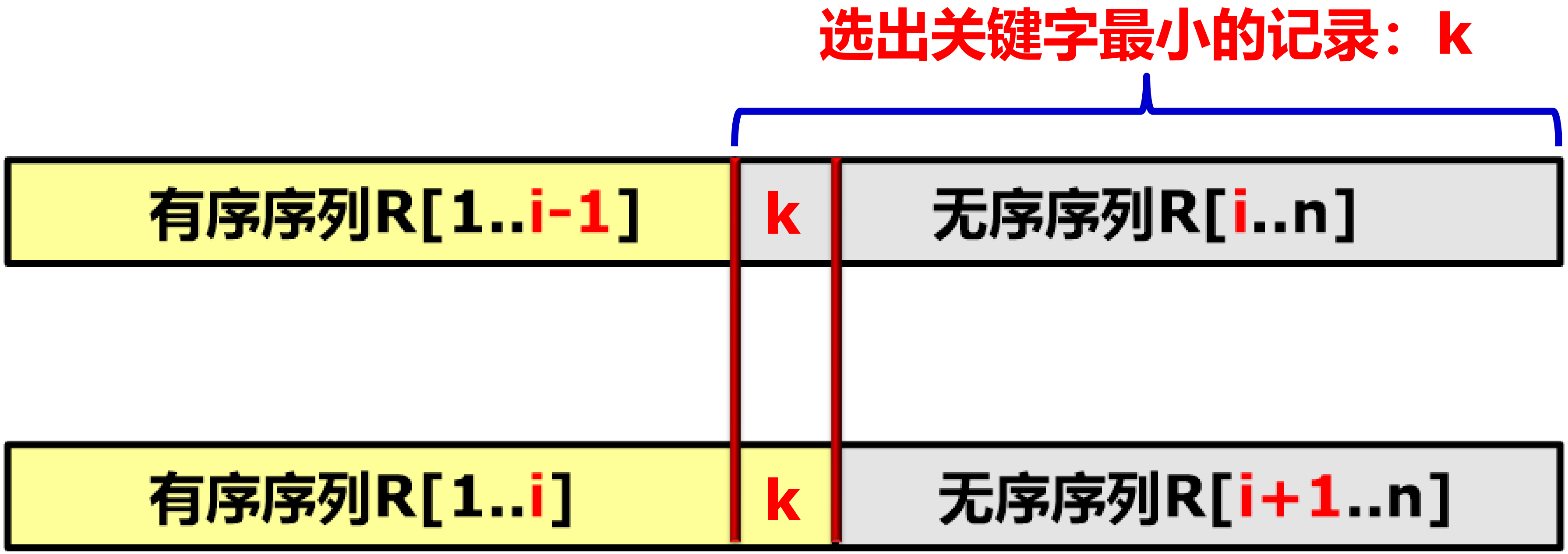
选择排序

- 简单选择排序
- 树形选择排序（自学）
- 堆排序（第六章）

简单选择排序（Selection Sort）

- 算法基本思想
  - 从无序子序列中选择关键字最小或最大的记录
  - 将其加入到有序子序列中（子序列初始长度为零）
  - 逐步增加有序子序列的长度直至长度等于原始序列
- 排序过程
  - 首先通过n-1次关键字比较，从n个记录中找出关键字最小的记录，将它与第一个记录交换
  - 再通过n-2次比较，从剩余的n-1个记录中找出关键字次小的记录，将它与第二个记录交换
  - 重复上述操作，共进行n-1趟排序后，排序结束

简单选择排序（Selection Sort）



- 选择排序思路：排序过程中
  - 设：第 i-1 趟直接选择排序之后待排记录序列的状态为
  - 则：第 i 趟直接选择排序之后待排记录序列的状态为

k 跟踪最小值

k k k

一趟： i=1 [13 38 65 97 76 49 27]

j j j j j j

k k

二趟： i=2 13 [27 65 97 76 49 38]

j j j j j

三趟： 13 27 [65 97 76 49 38]

四趟： 13 27 38 [97 76 49 65]

五趟： 13 27 38 49 [76 97 65]

六趟： 13 27 38 49 65 [97 76]

七趟： 13 27 38 49 65 76 [97]

结束： 13 27 38 49 65 76 97

CH07-6.cpp

对序列R[1..n]按升序进行简单选择排列

```
void select_sort( int* R, int n ){
    int i, j, k;        // k跟踪每一轮的最小值
    for(i = 1; i < n; ++i){   // n-1轮选择
        k = i;
        for(j = i+1; j <= n; ++j ){ // 找出最小值
            if( R[j] < R[k] )  k = j;
        }
        if( i != k ){   // 元素交换
            R[0] = R[k];  R[k] = R[i];  R[i] = R[0];
        }
    }
}
```

简单选择排序性能分析

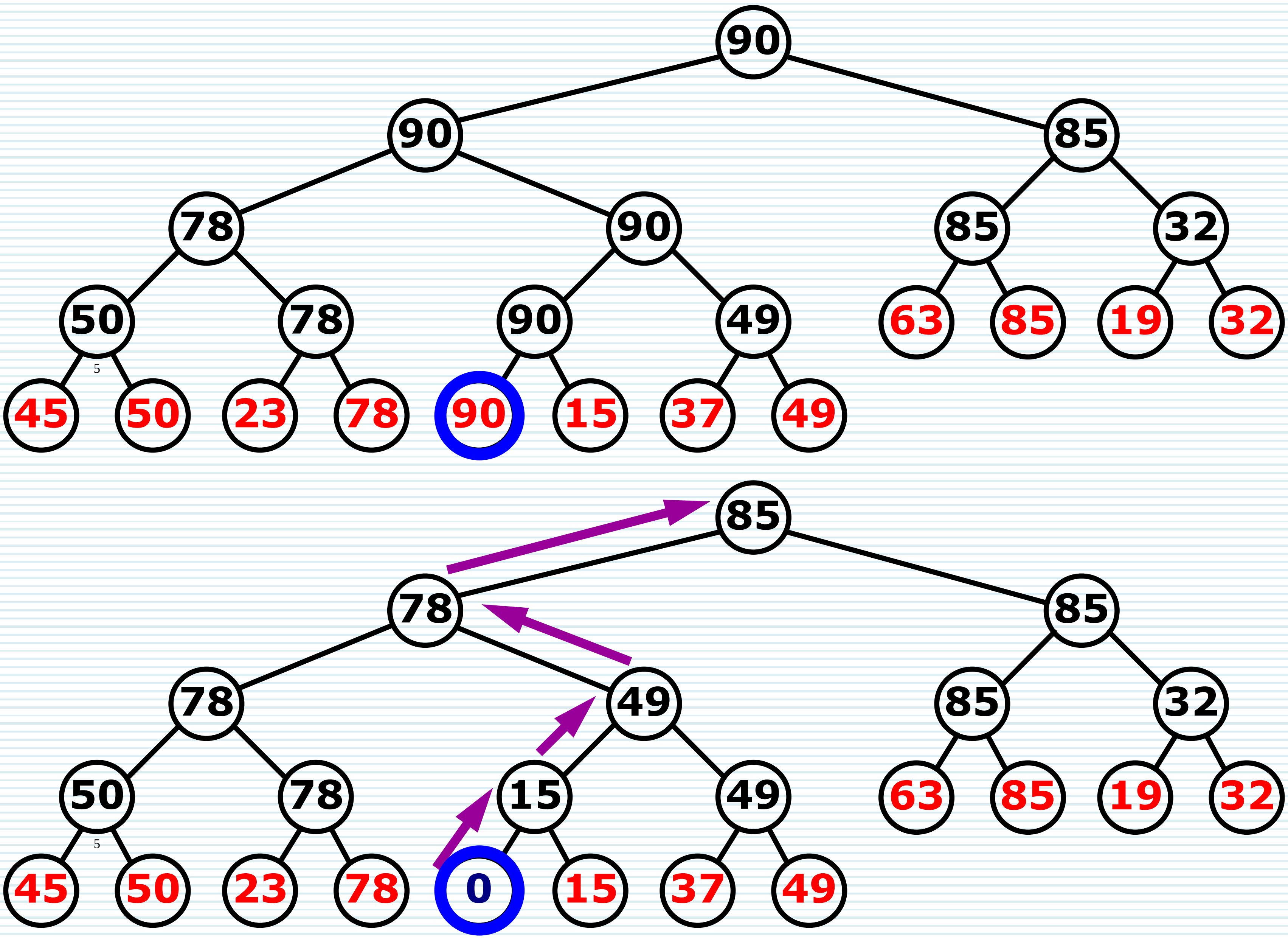
- 算法的时间复杂度　**T(n)=O(n^2)**
  - 记录移动次数
    - 最好情况下（正序）：　**0 次**
    - 最坏情况下（逆序）：　**3(n-1)次**
  - 记录比较次数: $\sum_{i=1}^{n-1}(n-i)=\frac{1}{2}(n^2-n)$
- 算法的空间复杂度
  - 只需要一个辅助存储单元:　S(n) = O(1)

树形选择排序（自学）

树形选择排序

- 算法流程简述
  - 将所有n个数据看成一棵完全二叉树的叶结点
  - 选出第一名：叶结点两两比较，胜出者进入上一层继续和兄弟进行比较；如果某个叶结点没有兄弟，该轮轮空，直接进入上一层；一直到二叉树的第二层的两个结点进行比较，胜出者形成根，产生出第一名
  - 产生其他名次：将刚选出的叶结点的成绩置为最差，再从该叶结点开始，沿向上路径依次和相应的兄弟结点进行比较，胜者进入上一层，最终形成根，得到当前名次
  - 重复n-2次，最终得到所有选手的排名

示例：45, 50, 23, 78, 90, 15, 37, 49, 63, 85, 19, 32



树形选择排序基本思想

1. 从初始关键字序列出发建立完全二叉树
   - 从叶结点开始，兄弟结点两两比较，胜出者进入上一层
   - 直到选出根结点为止：冠军结点
2. 输出根结点
3. 在余下元素中选出后续的根结点
   - 将刚得到名次的叶结点的成绩置为最差
   - 再从该叶结点出发，沿着到根的路径，依次进行兄弟结点间的比较，胜者进入上一层，直到选出根节点为止
4. 重复步骤2和3，直到n个元素输出，得到一个有序序列

树形选择排序

```
void tournament( int* R, int n ){
    int i, len = 2, idx, *T;
    while (len < n ) len <<= 1;  // 构造完全二叉树（T[0]闲置)
    T = (int *)malloc(sizeof(int) * len);
    for ( i = (len / 2); i < len; ++i ) {
        idx = i - len / 2;
        T[i] = ( idx < n ) ? R[idx] : INT_MAX;
    }
    for (i = (len / 2)-1 ; i > 0; --i)
        T[i] = ( T[2*i] < T[2*i+1] ) ? T[2*i] : T[2*i+1];
    for (i = 0; i < n; i++) {
        R[i] = T[1];  update( T, 1, len );
    }
    free(T);
}
```

更新二叉树的根节点（选出新的胜出记录）

```
void update(int *T, int root, int end){
    int lc = root * 2,  rc = lc + 1;
    // 到达叶节点：将当前的“冠军”替换为无穷大
    if ( lc >= end ) {
        T[root] = INT_MAX;   return;
    }
    if ( T[lc] == T[root] ) update(T, lc, end);
    else  update(T, rc, end);
    T[root] = (T[lc] < T[rc]) ? T[lc] : T[rc];
}
```

树形选择排序算法性能分析

- 选择第二名时，将刚选出的第一名置为最差（无穷大）
  - 与其兄弟进行比较，胜者上升到双亲结点
  - 继续与双亲的兄弟进行比较，直到形成根，得出第二名
- 这时需要比较的次数为树的深度，即：$\log_2 n$
- 产生后续名次需要比较的次数均为：$\log_2 n$
- 因此，树形排序的比较次数最多为：
  - $(n-1)\log_2 n + (n-1)$ --- （第2~第n名+第1名）
- 树形选择排序的时间复杂度为：$O(n\log_2 n)$

堆排序（参见第六章课件）

归并排序

二路归并排序

- **基本思想：**
  - 通过划分子序列，降低排序问题的复杂度
  - 通过合并有序的子序列，得到有序的序列
  - 核心思想：逐步增加记录有序序列的长度
- **2-路归并排序算法**
  - 每次归并操作仅处理两个位置相邻的有序子序列
  - 例如：给定如下关键字序列

6　15　45　23　9　78　35　38　18　27　20

算法复杂度分析示例2：归并排序

分解	6	15	45	23	9	78	35	38	18	27	20
归并	6	15 |	23	45 |	9	78 |	35	38 |	18	27 |	20
归并	6	15	23	45 |	9	35	38	78 |	18	20	27
归并	6	9	15	23	35	38	45	78 |	18	20	27
归并	6	9	15	18	20	23	27	35	38	45	78

二路归并排序

- 2-路归并排序算法流程
  - 设初始序列含有n个记录
  - 将原始序列划分为n个子序列（子序列长度为1）
  - 两两合并，得到 $\lfloor n/2 \rfloor$ 个长度为2或1的有序子序列
  - 合并规则：确保合并后的子序列中元素有序
    - 如果某一轮归并过程中，单出一个子序列
    - 则该子序列在该轮归并中轮空，等待下一趟归并
  - 如此重复，直至得到一个长度为n的有序序列为止

二路归并排序

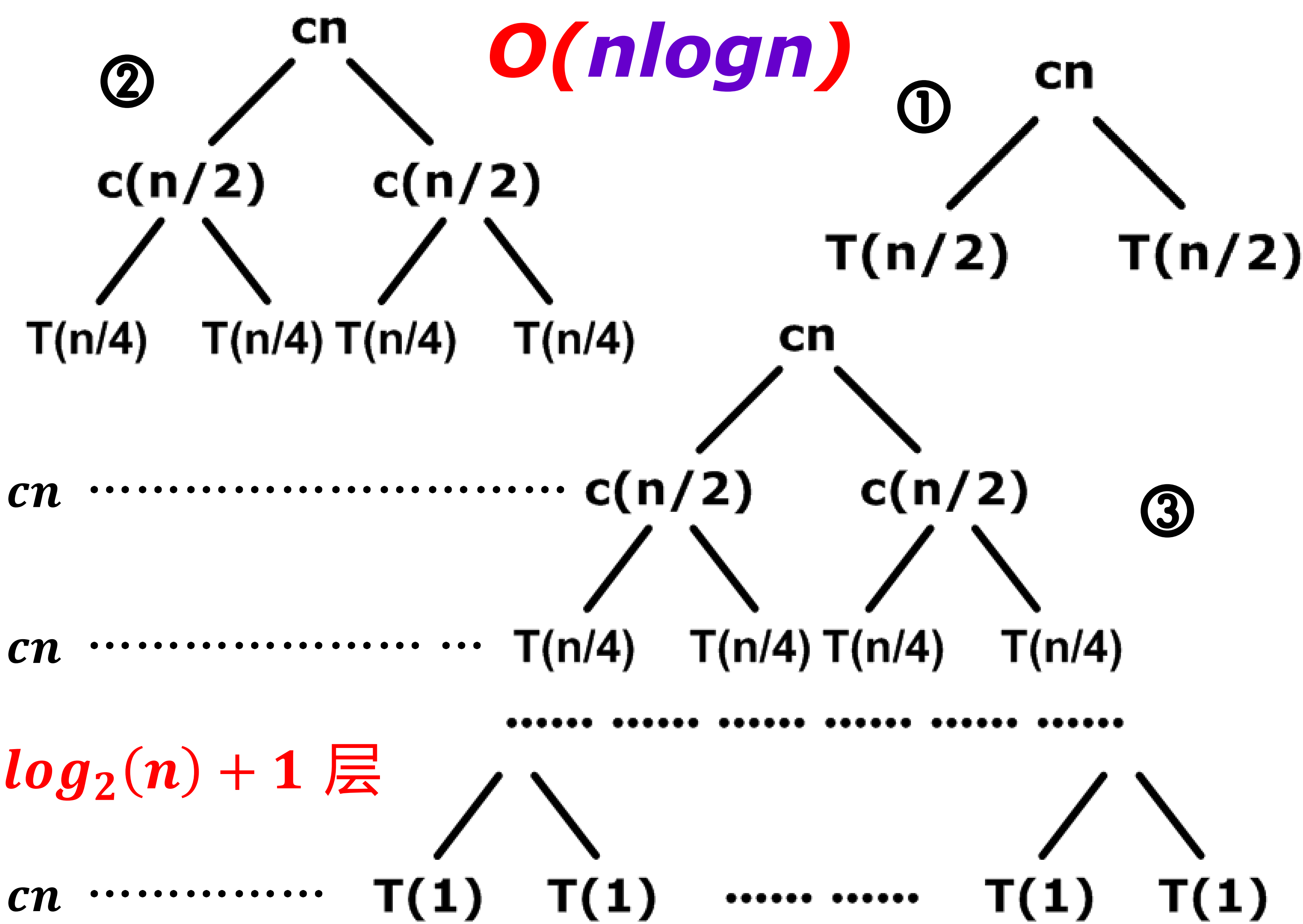
```
void merge_sort(int R[], int start, int end){
    int mid;
    if (start < end){
        mid = (start + end) / 2;
        merge_sort(R, start, mid);  ..................... T(n/2)
        merge_sort(R, mid+1, end);  ................. T(n/2)
        // 合并相邻的有序子序列
        merge(R, start, mid, end);  .....................Θ(n)
    }
}
```

$$T(n) = 2T(n/2) + \Theta(n)$$

二路归并排序

```
// Ra[h..t]的两部分Ra[h..s]和Ra[s+1..t]已按关键字有序
// Ra[h..s]和Ra[s+1..t]合并成有序表（Rb[s..t]为辅助表）
void merge(int Ra[], int Rb[], int h, int s, int t){
    int i = h, k = h; j = s + 1;
    while( i <= s && j <= t ){
        if(Ra[i] < Ra[j]) Rb[k++] = Ra[i++];
        else  Rb[k++] = Ra[j++];
    }
    while (i <= s) Rb[k++] = Ra[i++];
    while (j <= t) Rb[k++] = Ra[j++];
    for (i = h; i <= t; i++) Ra[i] = Rb[i];
}
```

使用递归树分析：$T(n) = 2T(n/2) + \Theta(n)$



2-路归并排序递归算法性能分析

- 空间复杂度
  - 2-路归并排序需要一个与原始序列等长的临时数组
  - 因此空间复杂度为O(n)
- 时间复杂度
  - 每一趟归并的时间复杂度为O(n)
  - 总共需要执行的归并次数为：$\log_2 n$
  - 因而，总的时间复杂度为$O(n\log_2 n)$

基数排序

基数排序（Radix Sort）

- 基数排序是
  - 一种借助“多关键字排序”的思想
  - 来实现“单关键字排序”的内部排序算法
  - 是采用“分配-收集”模式的排序方法
- 本节主要知识点
  - 多关键字的排序方法
  - 链式基数排序

多关键字排序

- 是一种无需进行关键字比较的排序方法
- 其基本操作是：“分配”和“收集”
- 例如：对52张扑克牌按以下次序排序

  ♦2<♦3<……<♦A < ♣2<♣3<……<♣A<

  ♥2<♥3<……<♥A < ♠2<♠3<……<♠A

- 序列中存在两类关键字：
  - 花色（♣<♦<♥<♠）和面值（**2<3<……<A**）
  - 并且“花色”地位高于“面值”

多关键字排序方法

- 最高位优先法（MSD）
  - 先对最高位关键字**k1**（如花色）排序
    - 将序列分成若干子序列
    - 每个子序列有相同的k1值
  - 然后让每个子序列对次关键字**k2**（如面值）排序
    - 进一步分成若干更小的子序列
  - 依次重复，直至每个子序列对最低位关键字kd排序
  - 最后将所有子序列依次连接在一起成为一个有序序列

最高位优先排序法：MSD

无序序列	**3,2,30**	**1,2,15**	**3,1,20**	**2,3,18**	**2,1,20**
对K¹排序	**1,2,15**	**2,3,18**	**2,1,20**	**3,2,30**	**3,1,20**
对K²排序	**1,2,15**	**2,1,20**	**2,3,18**	**3,1,20**	**3,2,30**
对K³排序	**1,2,15**	**2,1,20**	**2,3,18**	**3,1,20**	**3,2,30**

- 例如：学生记录含三个关键字:
  - 系别（K^1）、班号（K^2）和班内的序列号（K^3）
  - 其中以系别为最高关键字
- 高位优先排序的排序过程如图所示

多关键字排序方法

- 最低位优先法（LSD）
  - 从最低位关键字kd起进行排序
  - 然后再对高一位的关键字排序……
  - 依次重复，直至对最高位关键字k1完成排序
  - 便成为一个有序序列

最低位优先排序法：LSD

无序序列	3,2,30	1,2,15	3,1,20	2,3,18	2,1,20
对K^2排序	1,2,15	2,3,18	3,1,20	2,1,20	3,2,30
对K^1排序	3,1,20	2,1,20	1,2,15	3,2,30	2,3,18
对K^0排序	1,2,15	2,1,20	2,3,18	3,1,20	3,2,30

- 例如：学生记录含三个关键字：
  - 系别（K^1）、班号（K^2）和班内的序列号（K^3）
  - 其中以系别为最高关键字
- 低位优先排序的排序过程如图所示

多关键字排序方法

- MSD与LSD的不同之处
  - 按MSD排序：必须对原始序列逐层分割
    - 形成若干子序列，然后对各子序列分别排序
  - 按LSD排序：**不必分割成子序列**
    - 对每个关键字的排序都是整个序列参加排序
    - 并且可以**避免关键字比较**
      - 通过若干次分配与收集实现排序
- 问题：采用何种存储结构实现分配与收集？

链式基数排序

- 基数排序是一种基于“分配”和“收集”的思想将单关键字排序问题转换为多关键字排序问题的排序方法
- 实现基数排序时应采用链表作存储结构
  1. 待排序记录以指针相链，构成一个链表
  2. 分配时，按当前关键字的取值将记录分配到链队列的相应队列中，每个队列中记录的关键字取值相同
  3. 收集时，按当前关键字的取值从小到大将各队列首尾相连构成一个链表（即合并链队列为一个链表）
  4. 按照优先级由低到高顺序对每个关键字重复2和3两步

链式基数排序示例

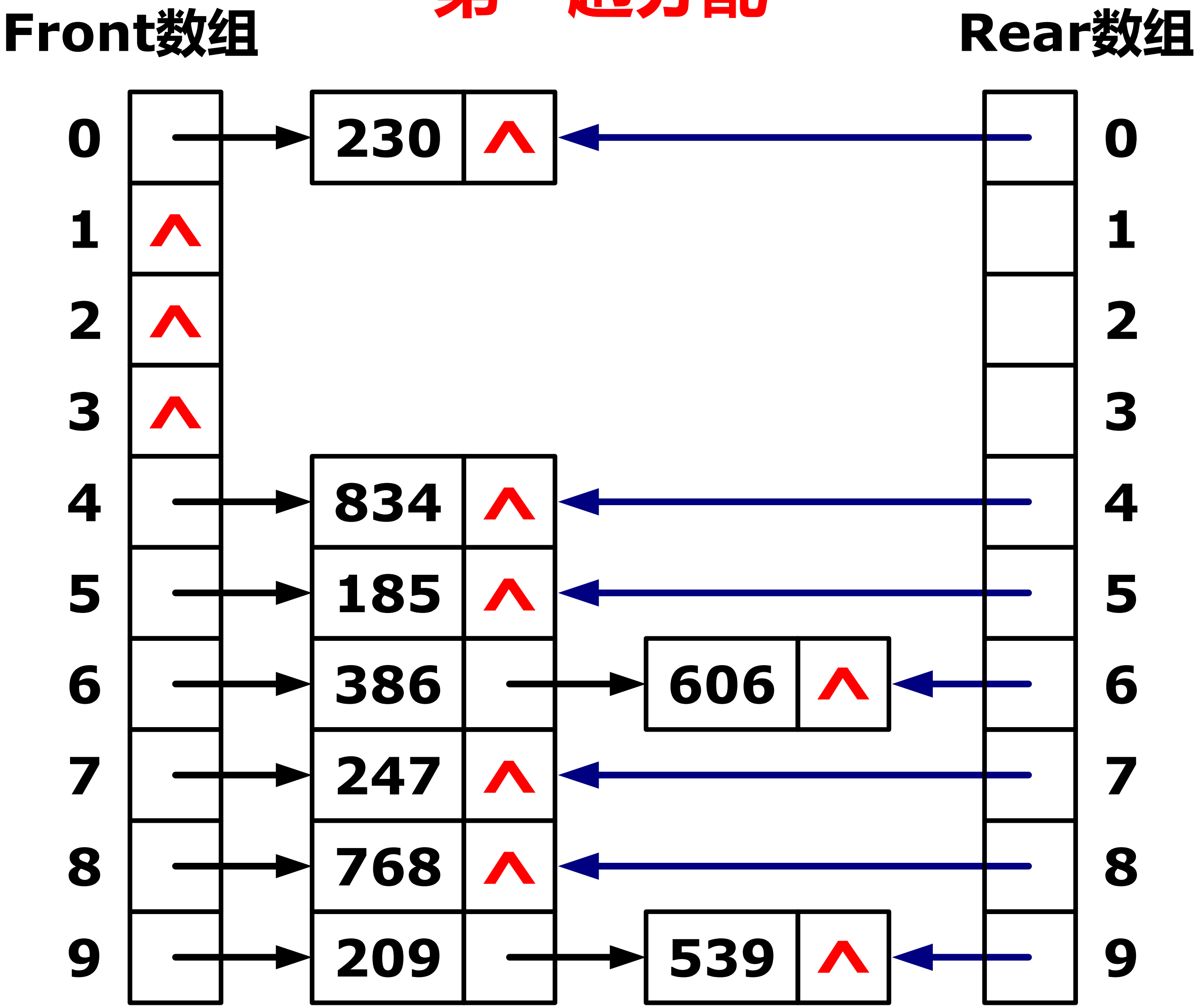
- 对如下序列执行基数排序

  { 209, 386, 768, 185, 247, 606, 230, 834, 539}

  - 首先按其个位数取值分别为 0, 1, …, 9 “分配” 成 10 组
    - 之后按从 0 至 9 的顺序将 它们 “收集” 在一起
  - 然后按其十位数取值分别为 0, 1, …, 9 “分配” 成 10 组
    - 之后按从 0 至 9 的顺序将它们 “收集” 在一起
  - 最后按其“百位数”重复一遍上述操作

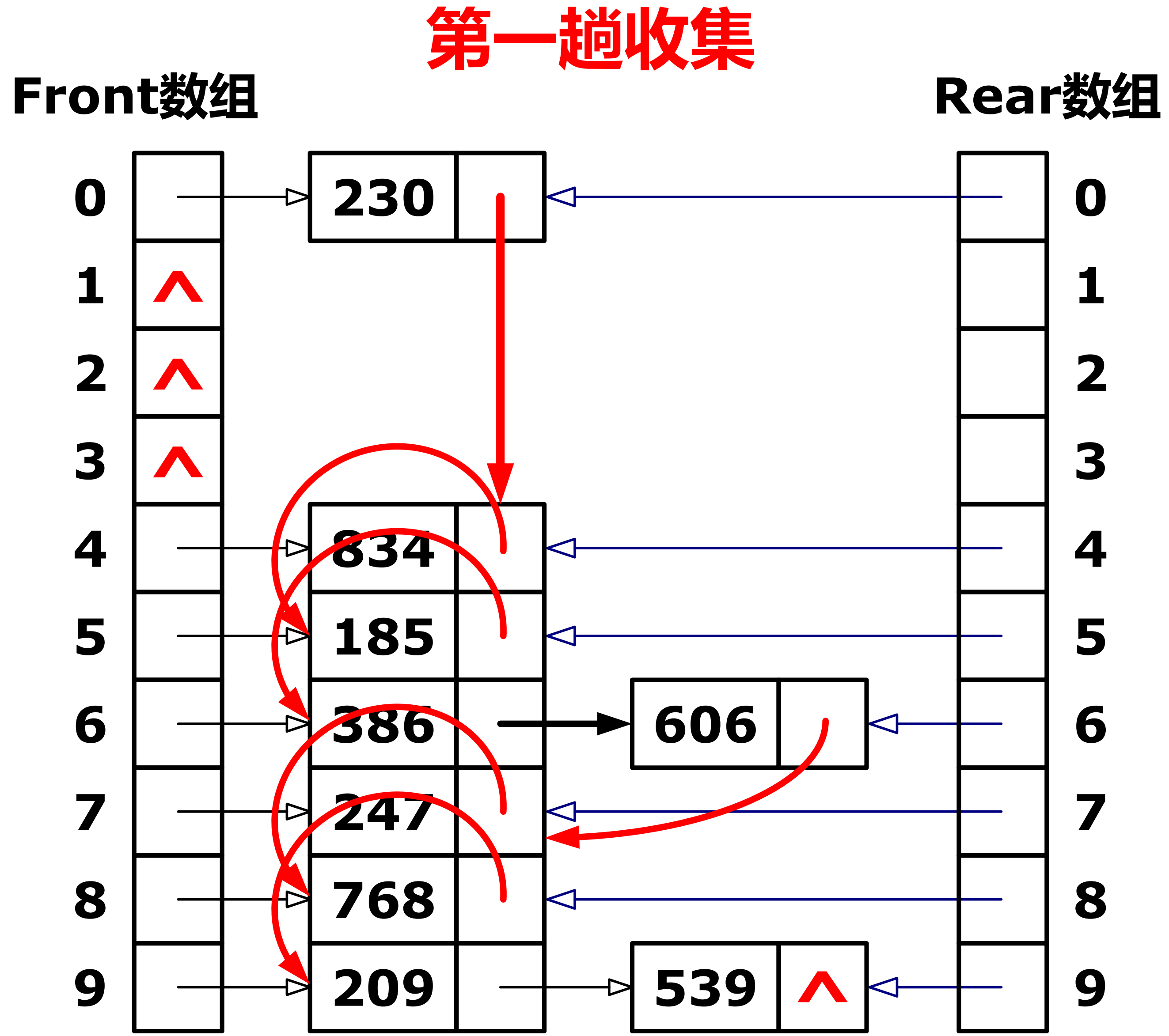
原始序列：{ 209, 386, 768, 185, 247, 606, 230, 834, 539 }

第一趟分配



原始序列：　209, 386, 768, 185, 247, 606, 230, 834, 539

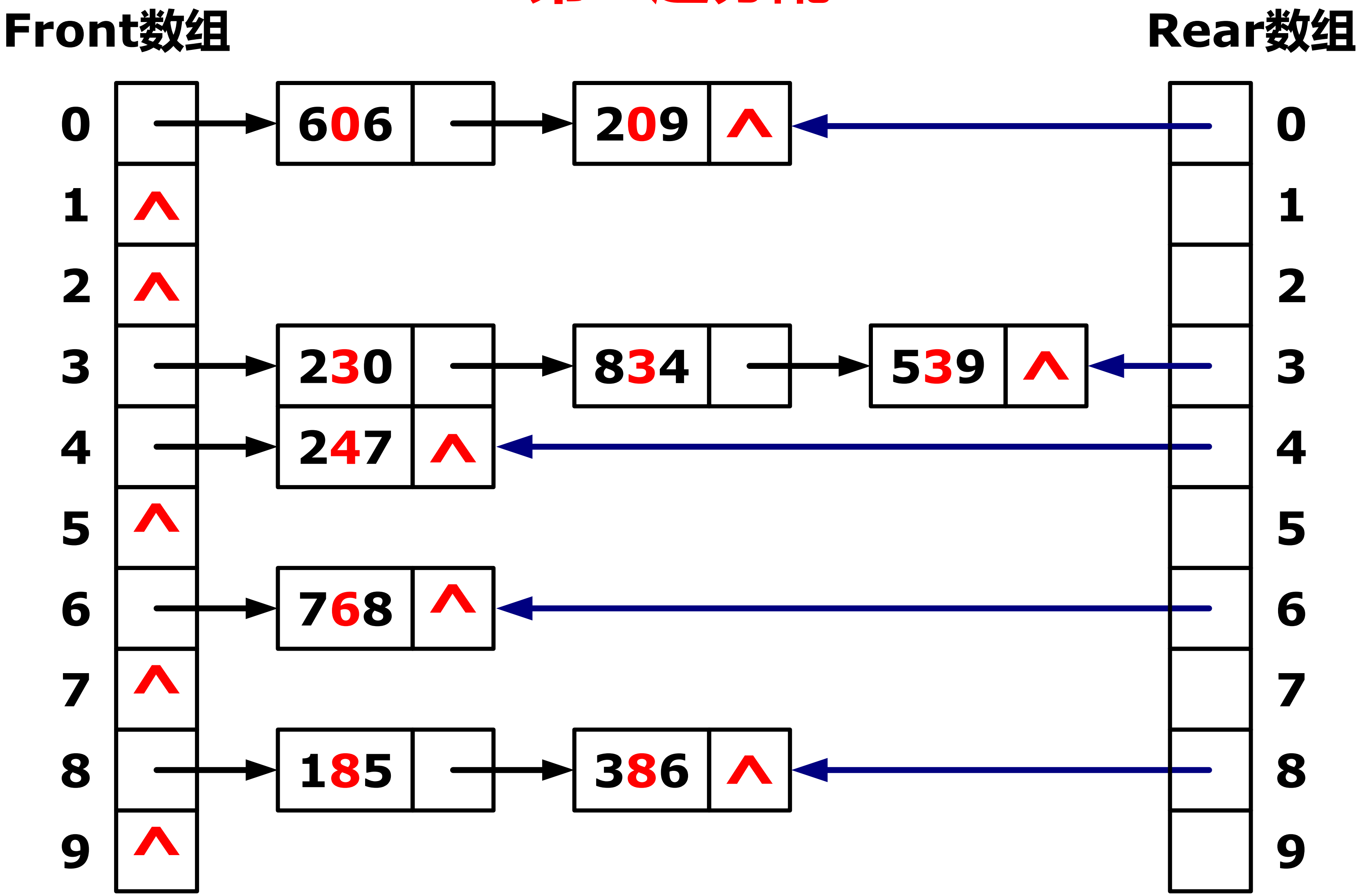
第一趟收集后：　230　834　185　386　606　247　768　209　539



第一趟收集后： 230 834 185 386 606 247 768 209 539

第二趟收集后： 606 209 230 834 539 247 768 185 386

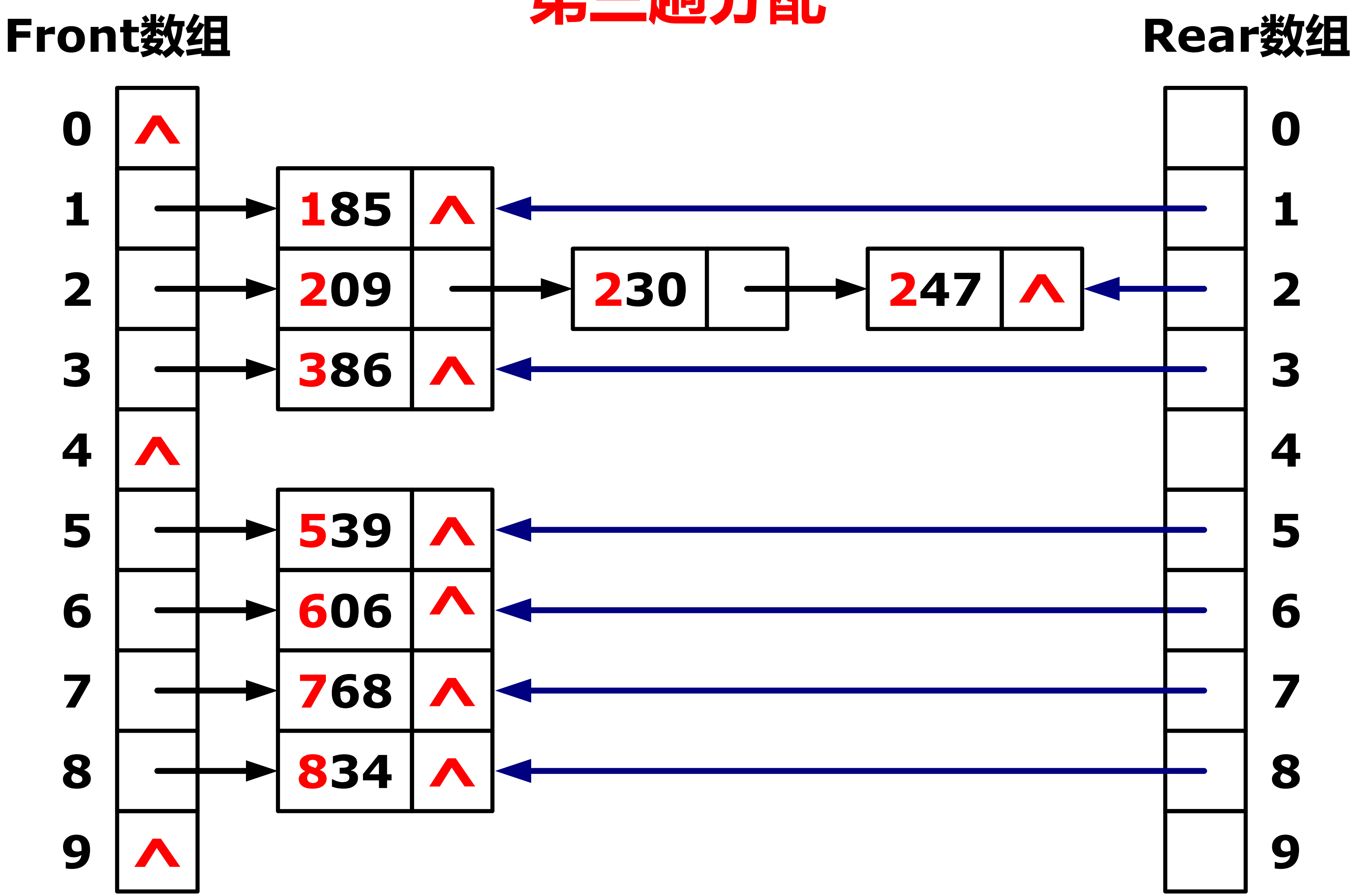
第二趟分配



第二趟收集后： 606 209 230 834 539 247 768 185 386

第三趟收集后： 185 209 230 247 386 539 606 768 834

第三趟分配



基数排序的基本数据结构

```
// 含next指针的待排元素
typedef struct node{
    int data;
    node *next;
}TNode;

// 首尾指针组合
typedef struct{
    node *front;
    node *rear;
}TPointer;
```

基数排序：构建辅助单链表

```
// 根据数组R构建带头结点的单链表
TNode* build_list(int R[], int n){
    int i; TNode* p, ph;
    ph = (TNode*)malloc(sizeof(TNode)); // 判空略
    ph->next = NULL;
    for(i = 0; i < n; ++i ){
        p = (TNode*)malloc(sizeof(TNode)); // 判空略
        p->data = R[i];
        p->next = ph->next;
        ph->next = p;
    }
    return ph;
}
```

对正整数构成的数组R执行基数排序

```
void radix_sort(int* R, int n){
    int i; TNode* p; TPointer Q[RADIX];
    int max_val = findmax(R, n);  // 求最大值
    TNode* ph = build_list(R, n);  // 构建链表
    for( i = 0;  max_val;  max_val/=10, ++i){
        dispatch(ph, Q, i); collect(ph, Q); // 分配收集
    } // 迭代次数由关键字位数决定
    p = ph->next; i = 0; // 将排序结果写入数组R
    while(p){ R[i] = p->data; p = p->next; ++i;  }
    destroy(ph); // 销毁辅助链表
}
```

基数排序的分配过程

```
void dispatch (TNode * ph, TPointer Q[], int d){
    int i, idx;  TNode * p = NULL;
    for( i = 0; i < RADIX; ++i ){
        Q[i].front = NULL;  Q[i].rear = NULL; }
    p = ph->next;        // 取原始链队列中第一个结点
    if(p){ ph->next = p->next; p->next = NULL; }
    while(p){
        idx = p->data;  //  取出*p中的第d位数字
        for(i = 0; i < d; ++i)  idx = idx / RADIX;
        idx = idx % 10;
        if( Q[idx].front == NULL){    // 将*p分配到相应队列中
            Q[idx].front = p; Q[idx].rear = p; }
        else{
            Q[idx].rear->next = p; Q[idx].rear = p; }
        p = ph->next;   // 取原始链队列中下一个结点
        if(p){  ph->next = p->next; p->next = NULL; }
    }
}
```

基数排序的收集过程

```
void collect(TNode* ph, TPointer * Q){
    int i; TNode*  p;
    // 找出Q数组中第一个指向非空队列的元素
    for(i = 0; !Q[i].front; ++i);
    // 将其链接到新的链表中
    ph->next = Q[i].front; p = Q[i].rear; i++;
    // 寻找其余非空队列，并将其顺序链接到主队列
    for(; i < RADIX; ++i){
        if(Q[i].front){
            p->next = Q[i].front; p = Q[i].rear;
        }
    }
    p->next = NULL;  // 修改链表尾结点
}
```

链式基数排序算法小结

- 若待排序列为整型值：基数排序过程中，首先将关键字分成几个关键字基数，再从个位开始执行“分配-收集”
  - 设置10个队列，F[i]和R[i]分别为第 i 个队列的头指针和尾指针
  - 第一趟分配：最低位关键字（个位）进行，修改记录的指针值，将记录分配至10个链队列中，每个队列记录的关键字的个位相同
  - 第一趟收集：改变所有非空队列的队尾记录的指针域，令其指向下一个非空队列的队头记录，重新将10个队列链成一个链表
  - 重复上述两步，进行第二趟、第三趟分配和收集，分别对十位、百位进行，最后得到一个有序序列
- 若为字符串，就从最右边开始分配-收集，若字符串长度不等则在短字符串右边补空格，规定空格比任何非空格字符都小

链式基数排序算法性能分析

- 设：n为待排序的数据个数，d为数据包含的关键字个数
- 设：Radix表示每个关键字取值个数（基数）
- 空间复杂度：O（n）
  - 需要两个长度为Radix的指针数组，指示链队列的头和尾
  - 需要n个数据存储单元（存储每个结点的数据和next指针）
- 时间复杂度：O（n）
  - 进行一轮分配所需的时间为：O(n)
  - 进行一轮收集所需时间为：O(Radix)
  - 一共需要执行d轮“分配-收集”
  - 因此总的时间复杂度为：$O(d \times (n+Radix))$
  - 当d和Radix可视为常数时，基数排序的时间复杂度为O(n)

排序方法比较

内部排序算法比较

排序算法	平均时间复杂度	最坏情况下时间复杂度	空间复杂度	稳定性
直接插入排序	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
折半插入排序	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
二路插入排序	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
希尔排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	不稳定
冒泡排序	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
快速排序	$O(n\log_2 n)$	$O(n^2)$	$O(\log_2 n)$	不稳定
直接选择排序	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
树形选择排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n)$	不稳定
堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	不稳定
归并排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n)$	稳定
基数排序	$O(d(n+r))$	$O(d(n+r))$	$O(r)$	稳定

各种排序方法的比较

- 对排序方法进行选择时主要从如下几方面考虑
  - 待排序记录个数n
    - 决定算法的时间复杂度和空间复杂度
  - 记录本身的大小
    - 影响算法的空间复杂度
  - 关键字的分布情况
    - 影响算法的实际性能表现（最好、最坏和平均性能）
  - 对排序结果的稳定性要求

时间特性

- 平均时间复杂度为**O(n²)**级别的算法
  - 插入排序、冒泡排序、选择排序
    - 插入排序（希尔排序）最常用，尤其当序列基本有序时
    - 选择排序移动记录的次数最少
- 时间复杂度为**O(nlogn)**级别的算法
  - 快速排序（交换）、堆排序（选择）、归并排序
    - 当待排序记录有序时，快速排序蜕化到$O(n^2)$
    - 在数据规模较大时，归并排序较堆排序更快
- 时间复杂度为**O(n)**级别的算法：基数排序
  - 当待排序记录有序时，插入和冒泡也可达到$O(n)$
- 选择排序、堆排序和归并排序的时间特性不受序列分布影响

空间特性

- 所有的简单排序方法的空间复杂度均为：O(1)
  - 插入排序（直接插入排序、折半插入排序、希尔排序）
  - 冒泡排序、简单选择排序、堆排序
- 快速排序的空间复杂度为：O(logn)
  - 为递归程序执行过程中栈所需的辅助空间
- 归并排序和基数排序的空间复杂度为：O(n)
  - 归并排序算法所需辅助空间最多：O(n)
  - 链式基数排序需附设队列首尾指针
    - 若不考虑新建链表，则空间复杂度为 O(RADIX)

电子科技大学

UESTC

1956