



# 数据结构：内排序

## Data Structure

主讲教师： 屈卫兰

Office number: 基地203

tel: 13873195964

# 内排序



- 排序术语及记号
- 三种代价为 $O(n^2)$ 的排序方法
- shell排序
- 快速排序
- 归并排序
- 堆排序
- 分配排序和基数排序
- 对各种排序算法的实验比较
- 排序问题的下限

# 排序术语及记号

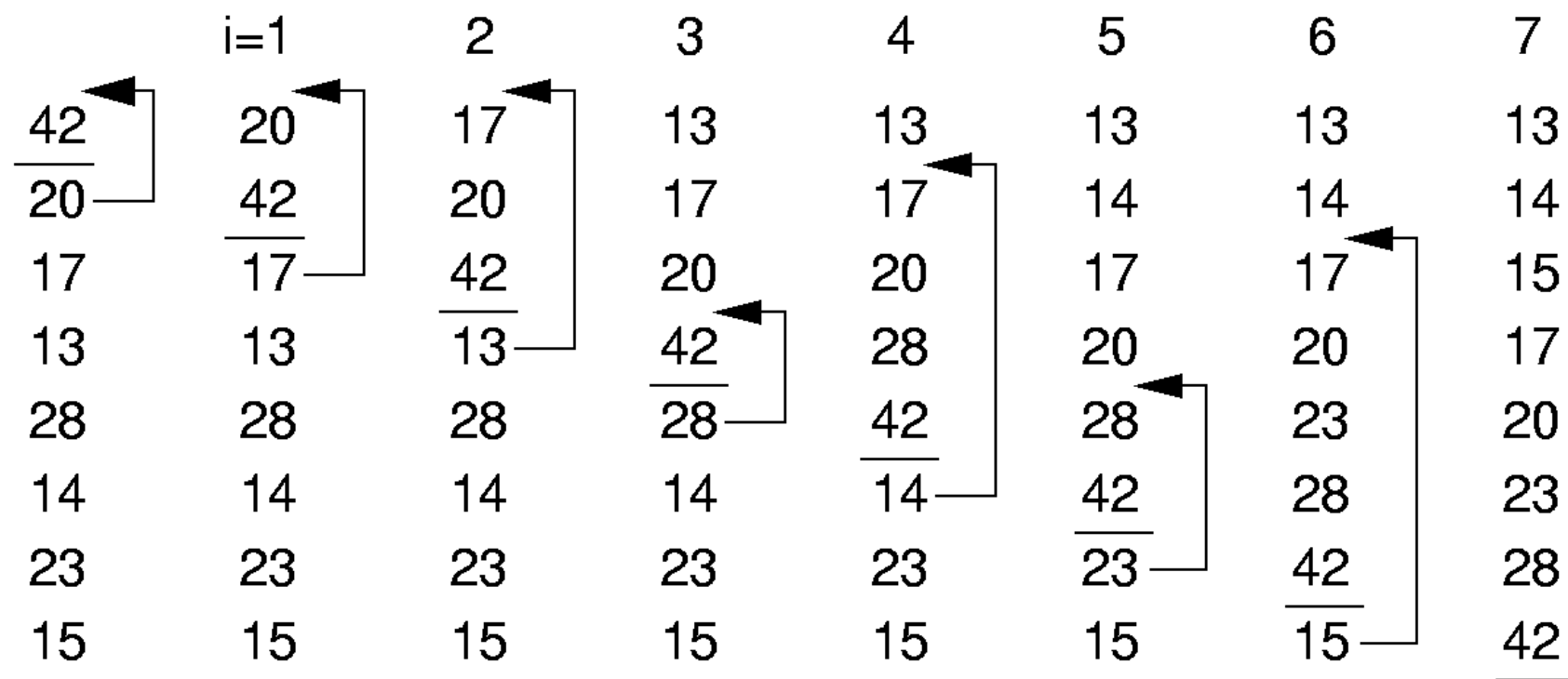
## 术语

- 记录——结点    文件——线性表
- 关键码：能够唯一确定结点的一个或若干域。
- 排序码：作为排序运算依据的一个或若干域。
- 组合排序码，（主关键码，次关键码）  
例，（总分数，数据结构分数，计算引论分数）
- 排序：排序就是将一组杂乱无章的数据按一定的规律排列起来

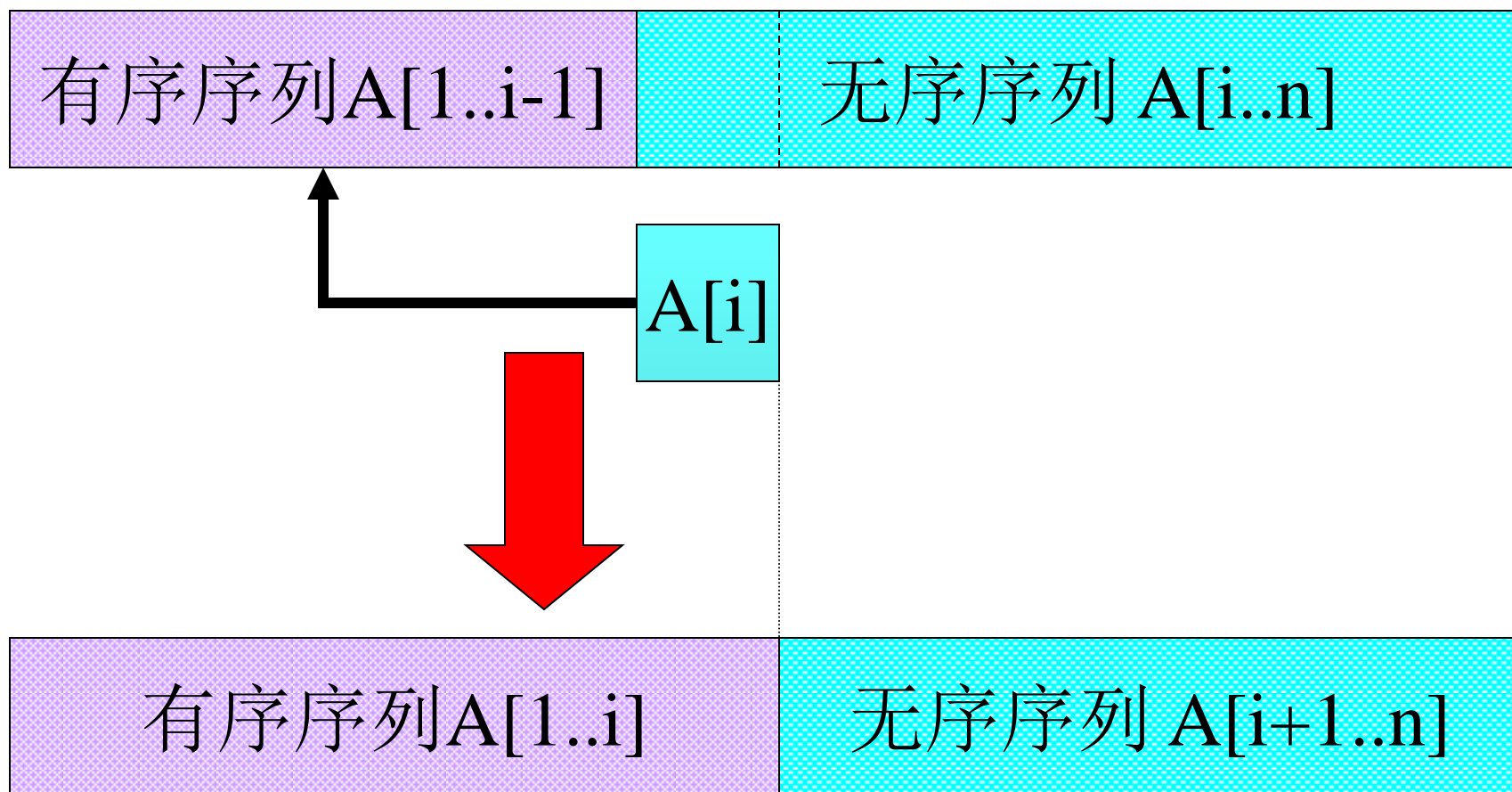
# 术语

- 排序问题：给定一组记录 $r_1, r_2, \dots, r_n$ ，其排序码分别为 $k_1, k_2, \dots, k_n$ ，将这些记录排成顺序为 $r_{s1}, r_{s2}, \dots, r_{sn}$ 的一个序列 $S$ ，满足条件 $k_{s1} \leq k_{s2} \leq k_{sn}$ 。
- 排序算法的稳定性：如果在对象序列中有两个对象 $r[i]$ 和 $r[j]$ ，它们的关键码 $k[i] == k[j]$ ，且在排序之前，对象 $r[i]$ 排在 $r[j]$ 前面。如果在排序之后，对象 $r[i]$ 仍在对象 $r[j]$ 的前面，则称这个排序方法是稳定的，否则称这个排序方法是不稳定的。
- 排序——在内存中进行的排序
- 外排序——排序过程还要访问外存(因为待排记录的数量太大，内存容纳不下)

# 插入排序



# 一趟直接插入排序的基本思想



# 实现“一趟插入排序”分三步进行

---

1. 在 $A[1..i-1]$ 中**查找** $A[i]$ 的插入位置,  
 $A[1..j].key \leq A[i].key < A[j+1..i-1].key$ ;
2. 将 $A[j+1..i-1]$ 中的所有**记录均后移**  
一个位置;
3. 将 $A[i]$  **插入**(复制)到 $A[j+1]$ 的位置上。

# 插入排序算法描述



```
template <typename E, typename Comp>  
void inssort(E A[], int n) {  
    for (int i=1; i<n; i++)  
        for (int j=i; (j>0)&&(Comp::prior(A[j], A[j-1])); j--)  
            swap(A,j,j-1);  
}
```



# 加入监视哨的插入排序算法

---

```
template < typename E, typename Comp >
void inssort(E A[], int n) {
    for (int i=2; i<=n; i++)
        { A[0]=A[i];j=i-1;
          while (Comp::prior(x, A[j]))
              { A[j+1]=A[j];j--;}
              A[j+1]=A[0];
          }
    }
```

# 直接插入排序时间分析

最好的情况（关键字在记录序列中顺序有序）：

“比较”的次数： “移动”的次数：

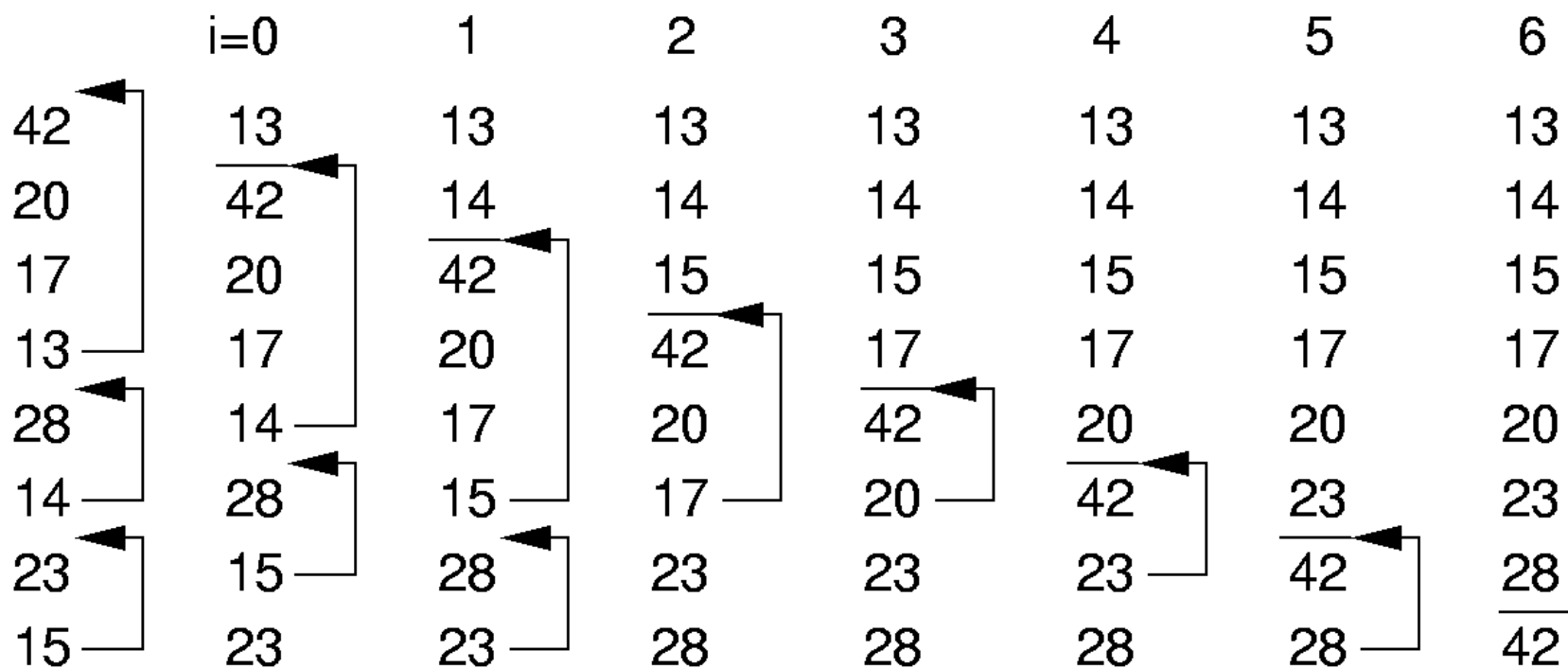
$$\sum_{i=2}^n 1 = n - 1 \qquad 2(n-1)$$

最坏的情况（关键字在记录序列中逆序有序）：

“比较”的次数： “移动”的次数：

$$\sum_{i=2}^n (i-1) = \frac{n(n-1)}{2} \qquad \sum_{i=2}^n (i+1) = \frac{(n+4)(n-1)}{2}$$

# 冒泡排序



# 冒泡排序算法



```
template < typename E, typename Comp >  
void bubsort(E A[], int n) {  
    for (int i=0; i<n-1; i++)  
        for (int j=n-1; j>i; j--)  
            if (Comp::prior(A[j], A[j-1]))  
                swap(A, j, j-1);  
}
```

# 改进的冒泡排序算法

---

```
template < typename E, typename Comp >
void bubsort(E A[], int n) {
    int flag;
    for (int i=0; i<n-1; i++)
        {flag=FALSE:
        for (int j=n-1; j>i; j--)
            if (Comp::prior(A[j], A[j-1]))
                { swap(A, j, j-1);flag=TRUE;}
        if(flag==FALSE) return;
    }}
```

# 冒泡排序时间分析

最好的情况（关键字在记录序列中顺序有序）：

只需进行一趟起泡

“比较”的次数：

**$n-1$**

“移动”的次数：

**0**

最坏的情况（关键字在记录序列中逆序有序）：

需进行 **$n-1$** 趟起泡

“比较”的次数：

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

“移动”的次数：

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

# 直接选择排序

	i=0	1	2	3	4	5	6
42	<u>13</u>	13	13	13	13	13	13
20	20	14	14	14	14	14	14
17	17	<u>17</u>	15	15	15	15	15
13	42	42	<u>42</u>	17	17	17	17
28	28	28	28	<u>28</u>	20	20	20
14	14	20	20	20	<u>28</u>	23	23
23	23	23	23	23	23	<u>28</u>	28
15	15	15	17	42	42	42	<u>42</u>

# 直接选择排序算法

```
template < typename E, typename Comp >
void selsort(E A[], int n) {
    for (int i=0; i<n-1; i++) {
        int lowindex = i; // Remember its index
        for (int j=n-1; j>i; j--) // Find least
            if (Comp::prior(A[j], A[lowindex]))
                lowindex = j; // Put it in place
        swap(A, i, lowindex);
    }
}
```



# 直接选择排序时间性能分析

对n个记录进行简单选择排序，所需进行的**关键字间的比较次数**总计为：

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

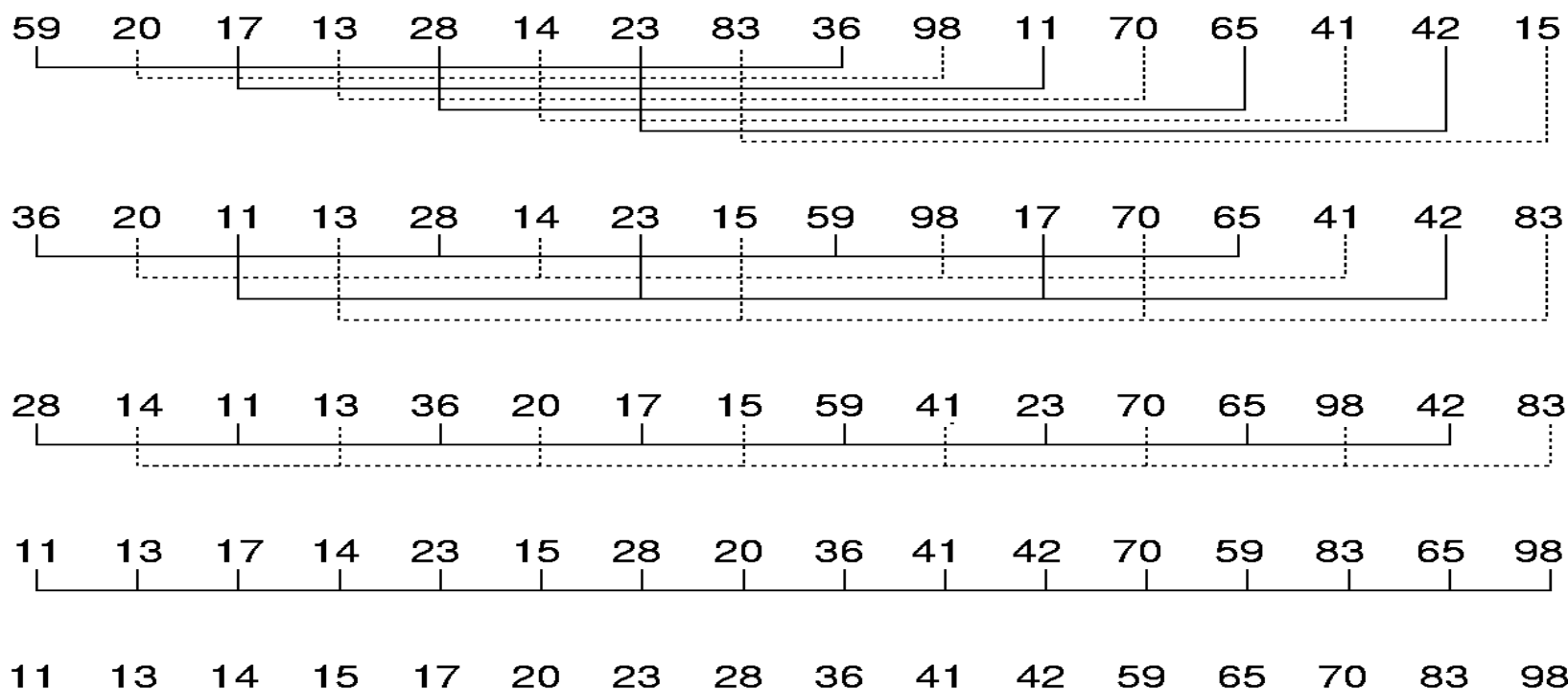
**移动记录的次数**，最小值为 0, 最大值为  $3(n-1)$  。

# 时间代价

	Insertion	Bubble	Selection
Comparisons:			
Best Case	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
Average Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Worst Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Swaps			
Best Case	0	0	$\Theta(n)$
Average Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$
Worst Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$

# Shell 排序

- 缩小增量排序法 (不稳定算法, 时间代价  $O(n^{1.5})$ )
- 原理:  $n$  很小时, 或基本有序时排序速度较快



# 希尔排序

将记录序列分成若干子序列，分别对每个子序列进行插入排序。

例如：将  $n$  个记录分成  $d$  个子序列：

$\{ A[1], A[1+d], A[1+2d], \dots, A[1+kd] \}$

$\{ A[2], A[2+d], A[2+2d], \dots, A[2+kd] \}$

...

$\{ A[d], A[2d], A[3d], \dots, A[kd], A[(k+1)d] \}$

其中， $d$  称为增量，它的值在排序过程中从大到小逐渐缩小，直至最后一趟排序减为 1。

例如:



16	25	12	30	47	11	23	36	9	18	31
----	----	----	----	----	----	----	----	---	----	----

第一趟希尔排序, 设增量  $d=5$

11	23	12	9	18	16	25	36	30	47	31
----	----	----	---	----	----	----	----	----	----	----

第二趟希尔排序, 设增量  $d=3$

9	18	12	11	23	16	25	31	30	47	36
---	----	----	----	----	----	----	----	----	----	----

第三趟希尔排序, 设增量  $d=1$

9	11	12	16	18	23	25	30	31	36	47
---	----	----	----	----	----	----	----	----	----	----

# shell排序算法

```
template < typename E, typename Comp >
void inssort2(E A[], int n, int incr) {
    for (int i=incr; i<n; i+=incr)
        for (int j=i;
              (j>=incr) &&
              (Comp::prior(A[j], A[j-incr])); j-=incr)
            swap(A, j, j-incr);
}
```



# shell排序算法

```
template < typename E, typename Comp >
void shellsort(E A[], int n) { // Shellsort
    for (int i=n/2; i>2; i/=2) // For each incr
        for (int j=0; j<i; j++) // Sort sublists
            inssort2<E,Comp>(&A[j], n-j, i);
    inssort2<E,Comp>(A, n, 1);
}
```

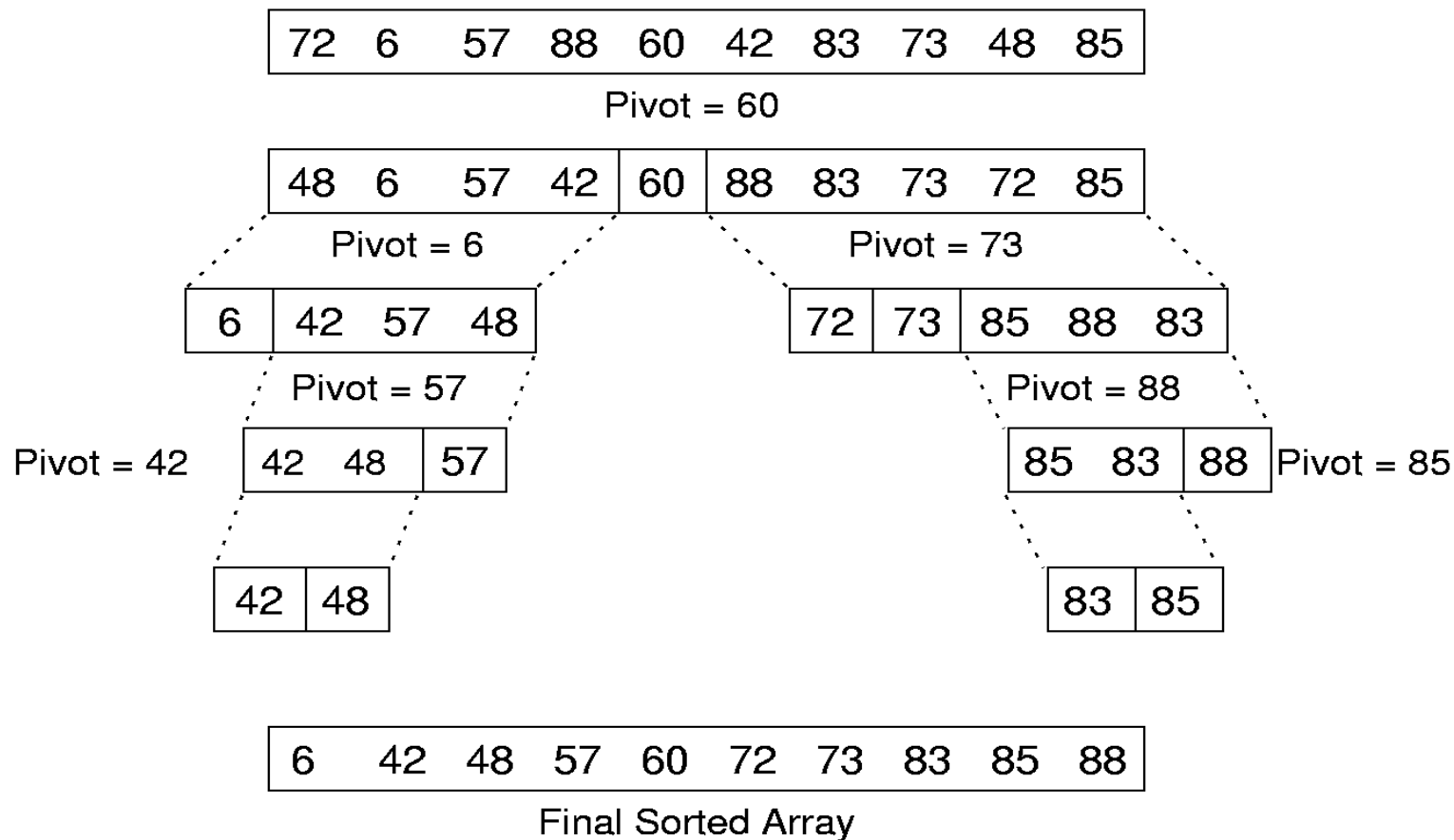


# 快速排序

Initial	72	6	57	88	85	42	83	73	48	60
										r
Pass 1	72	6	57	88	85	42	83	73	48	60
									r	
Swap 1	48	6	57	88	85	42	83	73	72	60
									r	
Pass 2	48	6	57	88	85	42	83	73	72	60
						r				
Swap 2	48	6	57	42	85	88	83	73	72	60
						r				
Pass 3	48	6	57	42	85	88	83	73	72	60
				r						
Swap 3	48	6	57	85	42	88	83	73	72	60
				r						
Reverse Swap	48	6	57	42		85	88	83	73	60
				r						



# 快速排序



# 一趟快速排序

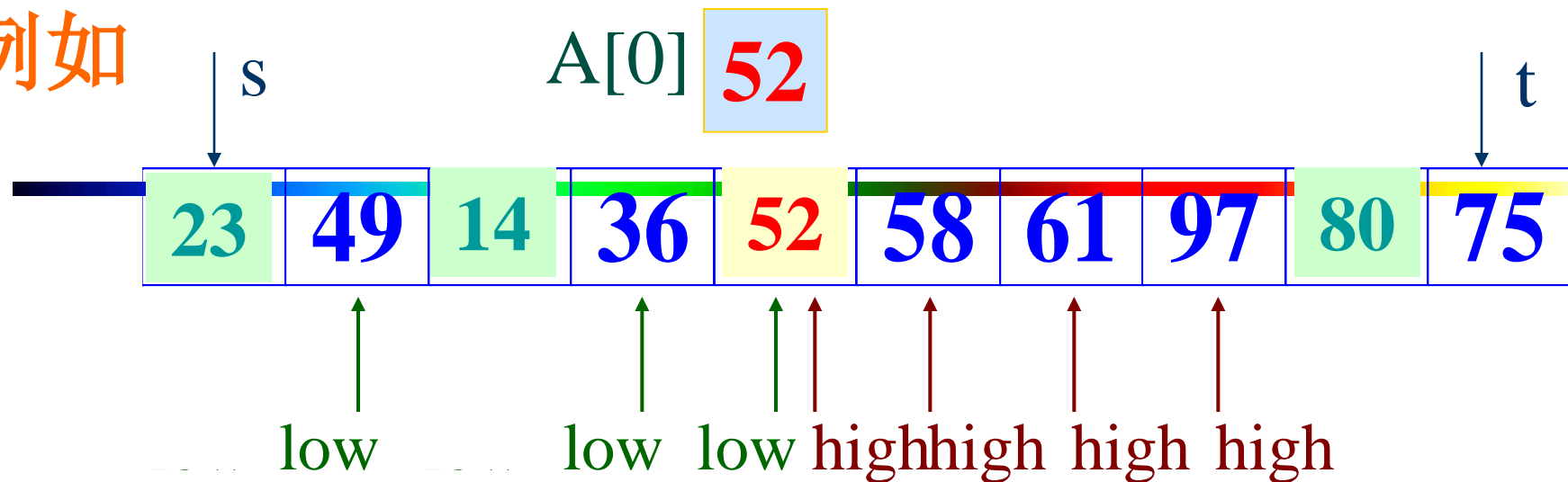
**目标：** 找一个记录，以它的关键字作为“**枢轴**”，凡其**关键字小于枢轴**的记录均移动至该记录之前，反之，凡**关键字大于枢轴**的记录均移动至该记录之后。

致使一趟排序之后，记录的无序序列 $A[s..t]$ 将分割成两部分： $A[s..i-1]$ 和 $A[i+1..t]$ ，且

$$A[j].key \leq A[i].key \leq A[j].key$$

$$(s \leq j \leq i-1) \quad \text{枢轴} \quad (i+1 \leq j \leq t)。$$

例如



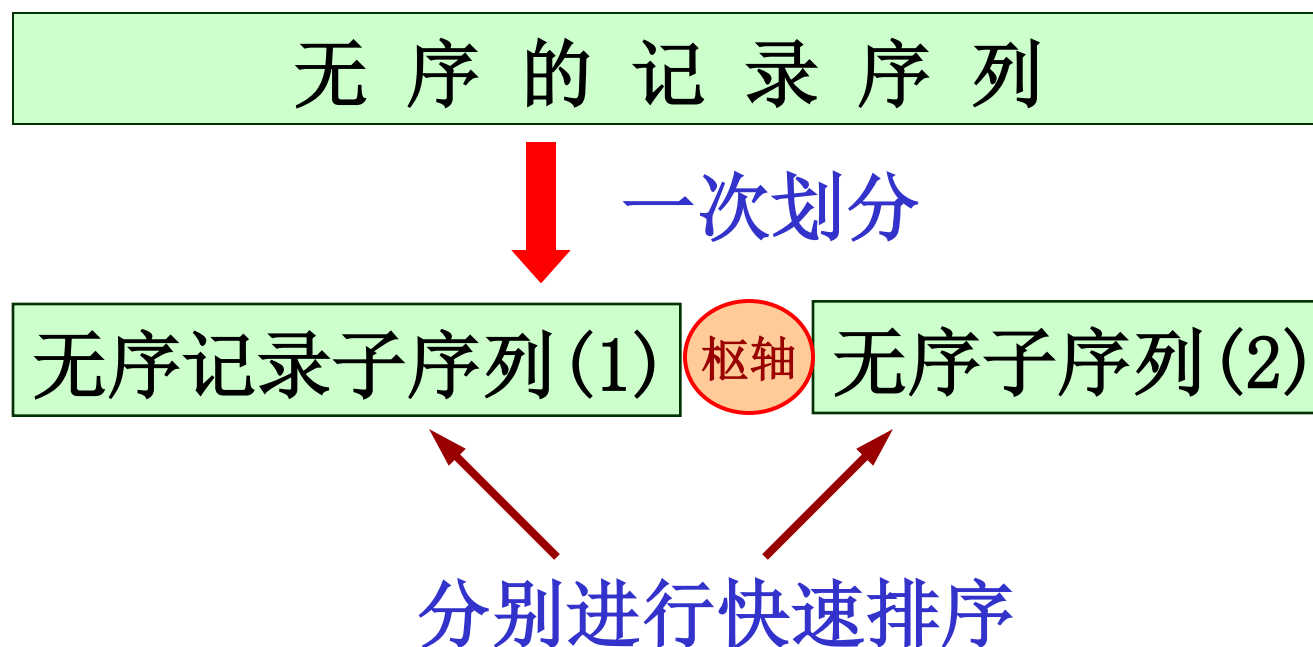
设  $A[s]=52$  为枢轴

将  $A[\text{high}].\text{key}$  和 枢轴的关键字进行比较，  
要求  $A[\text{high}].\text{key} \geq$  枢轴的关键字

将  $A[\text{low}].\text{key}$  和 枢轴的关键字进行比较，  
要求  $A[\text{low}].\text{key} \leq$  枢轴的关键字

# 快速排序

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

```
template < typename E, typename Comp >
void qsort(E A[], int i, int j) {
    if (j <= i) return; // List too small
    int pivotindex = findpivot(A, i, j);
    swap(A, pivotindex, j); // Put pivot at end
    // k will be first position on right side
    int k = partition<E,Comp>(A, i-1, j, A[j]);
    swap(A, k, j); // Put pivot in place
    qsort<E,Comp>(A, i, k-1);
    qsort<E,Comp>(A, k+1, j);
}
```

# 快速排序算法

```
template < typename E>
inline int findpivot(E A[], int i, int j)
{ return (i+j)/2; }
template < typename E, typename Comp >
inline int partition(E A[], int l, int r, E& pivot) {
    do { // Move the bounds in until they meet
        while (Comp::prior(A[++l], pivot));
        while ((l<r) && Comp::prior(pivot ,A[--r]));
        swap(A, l, r); // Swap out-of-place values
    } while (l < r); // Stop when they cross
    swap(A, l, r); // Reverse last swap
    return l;      // Return first pos on right
}
```

# 快速排序的时间分析

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

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

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

若待排序列中记录的关键字是随机分布的，则  $k$  取 1 至  $n$  中任意一值的可能性相同。

由此可得快速排序所需时间的平均值为：

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

设  $T_{avg}(1) \leq b$

则可得结果：

$$T_{avg}(n) < \left(\frac{b}{2} + 2c\right)(n+1) \ln(n+1)$$

**结论：快速排序的时间复杂度为  $O(n \log n)$**



# 快速排序的时间分析



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

# 归并排序

通常采用的是**2-路归并**排序。即：将两个位置相邻的记录有序子序列

有序子序列 $R[l..m]$	有序子序列 $R[m+1..n]$
-----------------	-------------------

归并为一个记录的有序序列。

有序序列 $R[l..n]$
----------------

# 归并排序

```
List mergesort(List inlist) {  
    if (inlist.length() <= 1) return inlist;  
    List l1 = half of the items from inlist;  
    List l2 = other half of items from inlist;  
    return merge(mergesort(l1), mergesort(l2));  
}
```

36   20   17   13   28   14   23   15

20   36	13   17	14   28	15   23
---------	---------	---------	---------

13   17   20   36	14   15   23   28
-------------------	-------------------

13   14   15   17   20   23   28   36
---------------------------------------

# 归并排序的算法

如果记录无序序列  $A[\textit{left}..\textit{right}]$  的两部分

$A[\textit{left}..\lfloor (\textit{left} + \textit{right})/2 \rfloor]$  和  $A[\lfloor (\textit{left} + \textit{right})/2 \rfloor + 1..\textit{right}]$

分别按关键字有序，

则利用上述归并算法很容易将它们归并成整个记录序列是一个有序序列。

由此，应该先分别对这两部分进行  
2-路归并排序。

例如:

52, 23, 80, 36, 68, 14 (left=1, right=6)

[ 52, 23, 80] [36, 68, 14]

[ 52, 23][80] [36, 68][14]

[ 52] [23]

[36][68]

[ 23, 52]

[36, 68]

[ 23, 52, 80]

[14, 36, 68]

[ 14, 23, 36, 52, 68, 80 ]

# 归并排序算法

```
template < typename E, typename Comp >
void mergesort(E A[], E temp[], int left, int right) {
    if (left == right) return;
    int mid = (left+right)/2;
    mergesort<E,Comp>(A, temp, left, mid);
    mergesort<E,Comp>(A, temp, mid+1, right);
    for (int i=left; i<=right; i++) // Copy
        temp[i] = A[i];
    int i1 = left; int i2 = mid + 1;
    for (int curr=left; curr<=right; curr++) {
        if (i1 == mid+1)    // Left exhausted
            A[curr] = temp[i2++];
        else if (i2 > right) // Right exhausted
            A[curr] = temp[i1++];
        else if (Comp::prior(temp[i1], temp[i2]))
            A[curr] = temp[i1++];
        else A[curr] = temp[i2++];
    }
}
```

# 优化归并排序算法

```
template < typename E, typename Comp >
void mergesort(E A[], E temp[],int left, int right) {
    if ((right-left) <= THRESHOLD) {
        inssort<E,Comp>(&A[left],right-left+1);
        return;
    }
    int i, j, k, mid = (left+right)/2;
    if (left == right) return;
    mergesort<E,Comp>(A, temp, left, mid);
    mergesort<E,Comp>(A, temp, mid+1, right);
    for (i=mid; i>=left; i--) temp[i] = A[i];
    for (j=1; j<=right-mid; j++)
        temp[right-j+1] = A[j+mid];
    for (i=left,j=right,k=left; k<=right; k++)
        if (temp[i] < temp[j]) A[k] = temp[i++];
        else A[k] = temp[j--];
}
```

# 归并排序算法效率

设需排序元素的数目为 $n$ ，递归的深度为 $\log n$ （简单起见，设 $n$ 是2的幂），第一层递归是对一个长度为 $n$ 的数组排序，下一层是对两个长度为 $n/2$ 的子数组排序，...，最后一层对 $n$ 个长度为1的子数组排序。

时间复杂度： $T(n)=O(n\log 2n)$

空间复杂度： $S(n)=O(n)$



# 堆排序

## 堆排序：

将无序序列建成一个堆，得到关键字最小（或最大）的记录；输出堆顶的最小（大）值后，使剩余的 $n-1$ 个元素重又建成一个堆，则可得到 $n$ 个元素的次小值；重复执行，得到一个有序序列，这个过程叫堆排序。

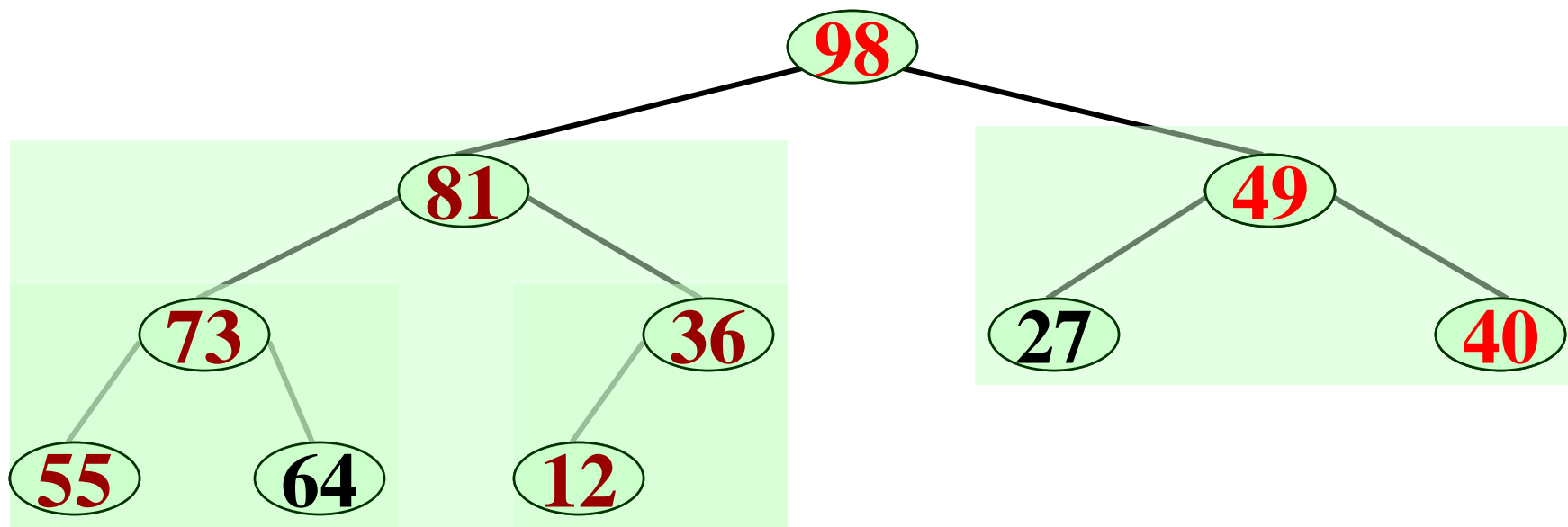
## 堆排序需解决的两个问题：

如何由一个无序序列建成一个堆？

如何在输出堆顶元素之后，调整剩余元素，使之成为一个新的堆？

# 建堆是一个从下往上进行“筛选”的过程

例如：排序之前的关键字序列为



现在，左/右子树都已经调整为堆，最后只要调整根结点，使整个二叉树是个“堆”即可。

# 堆排序算法

## 第二个问题解决方法——筛选

输出堆顶元素之后，以堆中最后一个元素替代之；然后将根结点值与左、右子树的根结点值进行比较，并与其中小者进行交换；重复上述操作，直至叶子结点，将得到新的堆，称这个从堆顶至叶子的调整过程为“筛选”。

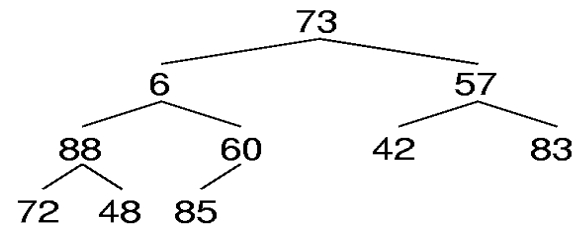
## 堆排序算法

```
template < typename E, typename Comp >
void heapsort(E A[], int n) { // Heapsort
    E maxval;
    maxheap<E,Comp> H(A, n, n);
    for (int i=0; i<n; i++)    // Now sort
        maxval=H.removefirst(); // Put max at end
}
```

# Heapsort Example (1)

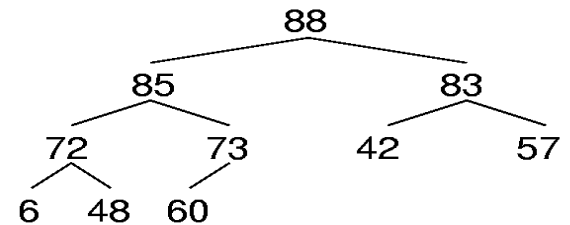
Original Numbers

73	6	57	88	60	42	83	72	48	85
----	---	----	----	----	----	----	----	----	----



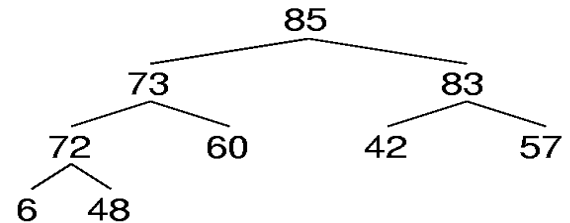
Build Heap

88	85	83	72	73	42	57	6	48	60
----	----	----	----	----	----	----	---	----	----



Remove 88

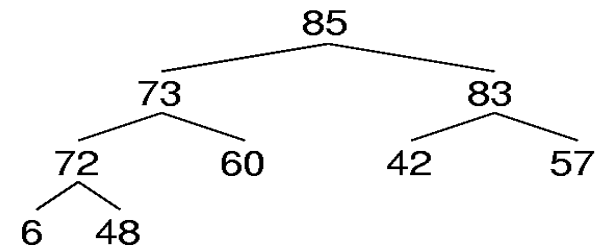
85	73	83	72	60	42	57	6	48	88
----	----	----	----	----	----	----	---	----	----



# Heapsort Example (2)

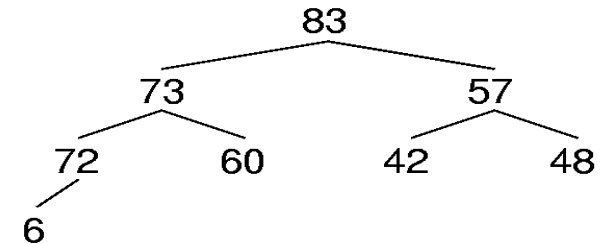
Remove 88

85	73	83	72	60	42	57	6	48	88
----	----	----	----	----	----	----	---	----	----



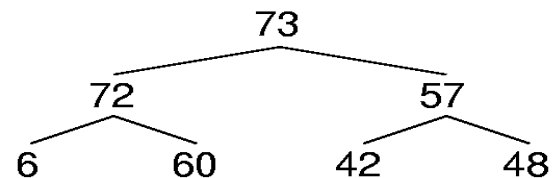
Remove 85

83	73	57	72	60	42	48	6	85	88
----	----	----	----	----	----	----	---	----	----



Remove 83

73	72	57	6	60	42	48	83	85	88
----	----	----	---	----	----	----	----	----	----



# 堆排序的时间复杂度分析

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

# 分配排序和基数排序

---

**A simple, efficient sort:**

```
for (i=0; i<n; i++)
```

```
    B[A[i]] = A[i];
```

**Ways to generalize:**

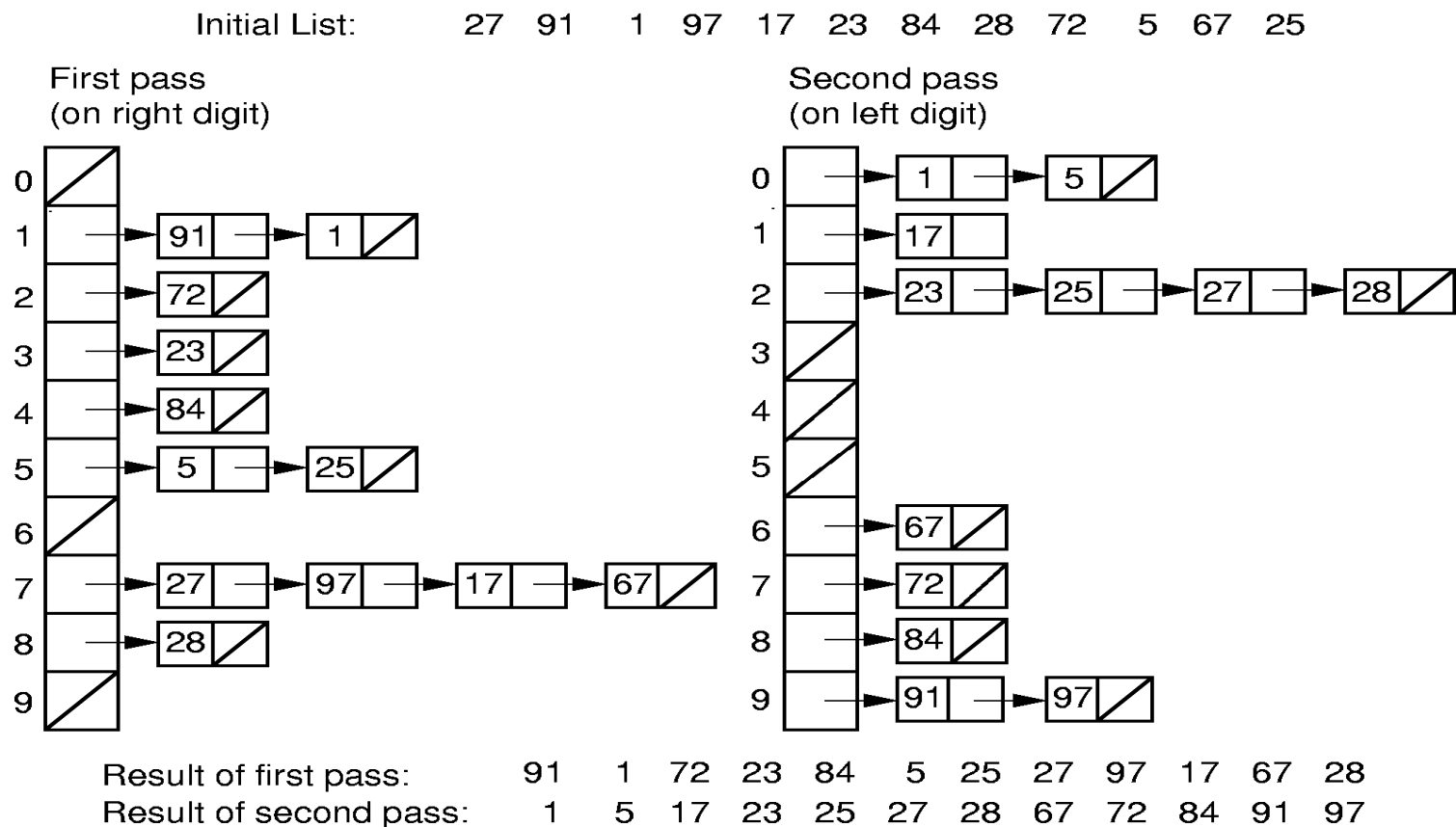
- **Make each bin the head of a list.**
- **Allow more keys than records.**

# 分配排序

```
template < typename E, class getKey >
void binsort(E A[], int n) {
    List<E> B[MaxKeyValue];
    E item;
    for (i=0; i<n; i++) B[A[i]].append(getKey::key(A[i]));
    for (i=0; i<MaxKeyValue; i++)
        for (B[i].setStart();
             B[i].getValue(item); B[i].next())
            output(item);
}
```



# 基数排序




Suppose that the record  $R_i$  has  $r$  keys.

  $K_i^j ::=$  the  $j$ -th key of record  $R_i$

  $K_i^0 ::=$  the **most** significant key of record  $R_i$

  $K_i^{r-1} ::=$  the **least** significant key of record  $R_i$

 A list of records  $R_0, \dots, R_{n-1}$  is **lexically sorted** with respect to the keys  $K^0, K^1, \dots, K^{r-1}$  iff

$$(K_i^0, K_i^1, \dots, K_i^{r-1}) \leq (K_{i+1}^0, K_{i+1}^1, \dots, K_{i+1}^{r-1}), \quad 0 \leq i < n-1.$$

That is,  $K_i^0 = K_{i+1}^0, \dots, K_i^l = K_{i+1}^l, K_i^{l+1} < K_{i+1}^{l+1}$  for some  $l < r-1$ .

【Example】 A deck of cards sorted on 2 keys

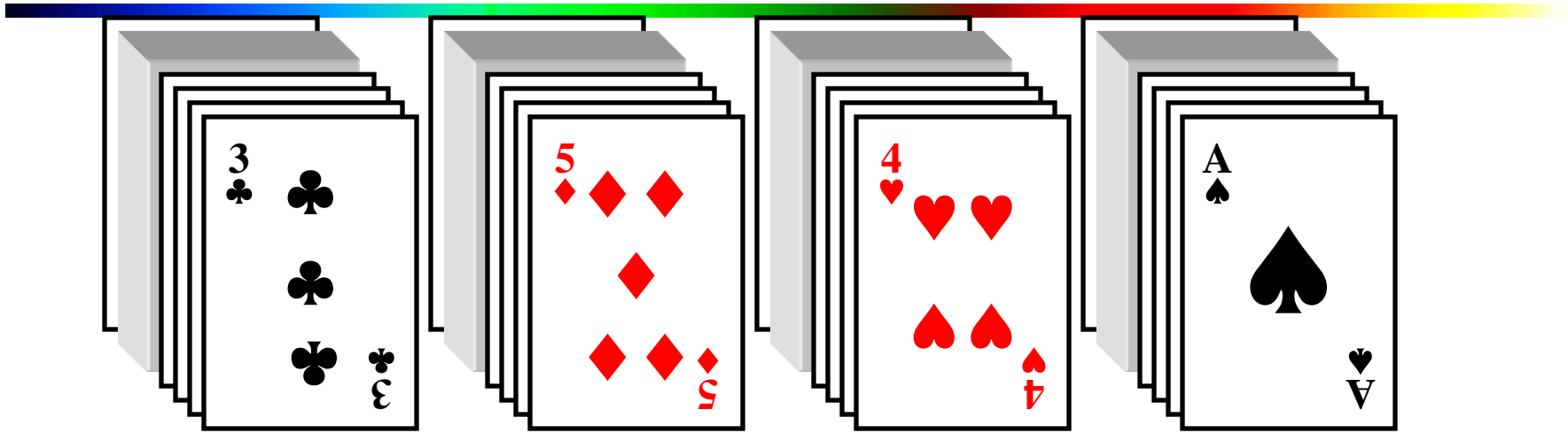
$K^0$  [Suit]                      ♣ < ♦ < ♥ < ♠

$K^1$  [Face value]    2 < 3 < 4 < 5 < 6 < 7 < 8 < 9 < 10 < J < Q < K < A

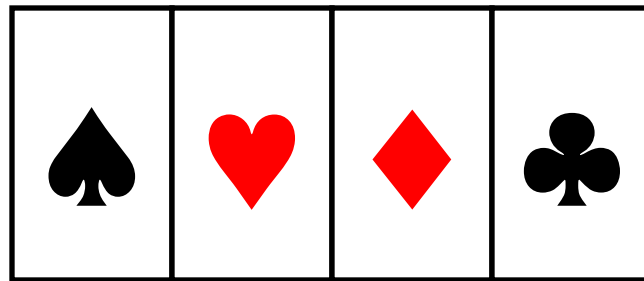
**Sorting result :**    2♣ ... A♣ 2♦ ... A♦ 2♥ ... A♥ 2♠ ... A♠

## ☞ MSD ( **M**ost **S**ignificant **D**igit ) Sort

① Sort on  $K^0$ : for example, create 4 buckets for the suits

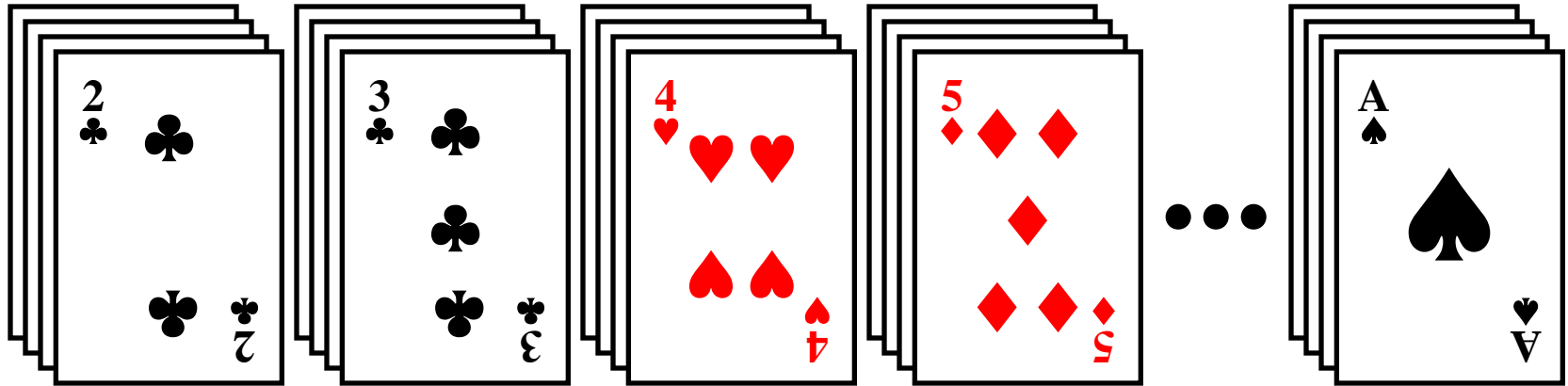


② Sort each bucket independently (using any sorting technique)



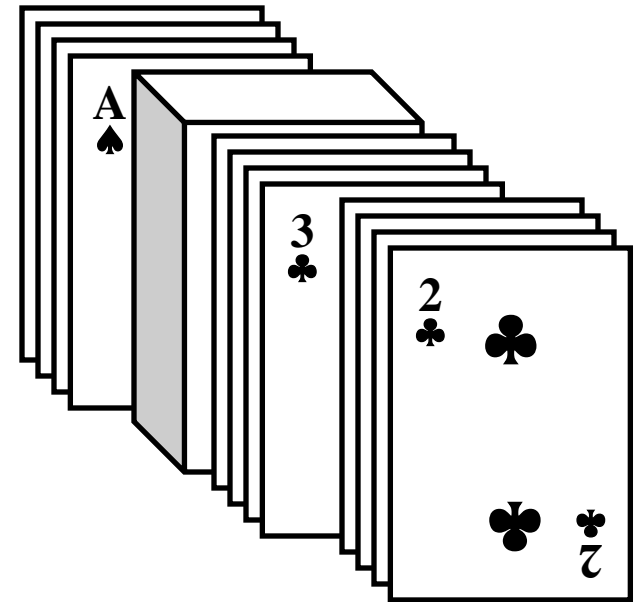
## 👉 LSD ( **L**east **S**ignificant **D**igit ) Sort

① Sort on  $K^1$ : for example, create 13 buckets for the face values



② Reform them into a single pile

③ Create 4 buckets and resort



# 基数排序算法

```
template < typename E, class getKey >
void radix(E A[], E B[], int n, int k, int r, int cnt[]) {
    // cnt[i] stores # of records in bin[i]
    int j;
    for (int i=0, rtoi=1; i<k; i++, rtoi*=r) {
        for (j=0; j<r; j++) cnt[j] = 0;
        // Count # of records for each bin
        for(j=0; j<n; j++) cnt[(getKey ::key(A[j])/rtoi)%r]++;
        // cnt[j] will be last slot of bin j.
        for (j=1; j<r; j++)
            cnt[j] = cnt[j-1] + cnt[j];
        for (j=n-1; j>=0; j--)\
            B[--cnt[(getKey ::key(A[j])/rtok)%r]] = A[j];
        for (j=0; j<n; j++) A[j] = B[j];
    }
}
```

# Radix Sort Example

Initial Input: Array A

27	91	1	97	17	23	84	28	72	5	67	25
----	----	---	----	----	----	----	----	----	---	----	----

First pass values for Count.  
rtok = 1.

0	1	2	3	4	5	6	7	8	9
0	2	1	1	1	2	0	4	1	0

Count array:  
Index positions for Array B.

0	1	2	3	4	5	6	7	8	9
0	2	3	4	5	7	7	11	12	12

End of Pass 1: Array A.

91	1	72	23	84	5	25	27	97	17	67	28
----	---	----	----	----	---	----	----	----	----	----	----

Second pass values for Count.  
rtok = 10.

0	1	2	3	4	5	6	7	8	9
2	1	4	0	0	0	1	1	1	2

Count array:  
Index positions for Array B.

0	1	2	3	4	5	6	7	8	9
2	3	7	7	7	7	8	9	10	12

End of Pass 2: Array A.

1	5	17	23	25	27	28	67	72	84	91	97
---	---	----	----	----	----	----	----	----	----	----	----

# 基数排序算法效率

## 代价分析：

对于 $n$ 个数据的序列，假设基数为 $r$ ，这个算法需要 $k$ 趟分配工作。每趟分配的时间为 $\Theta(n+r)$ ，因此总的时间开销为 $\Theta(nk+rk)$ 。因为 $r$ 是基数，它一般是比较小的。可以把它看成是一个常数。变量 $k$ 与关键码长度有关，它是以 $r$ 为基数时关键码可能具有的最大位数。在一些应用中我们可以认为 $k$ 是有限的，因此也可以把它看成是常数。在这种假设下，基数排序的最佳、平均、最差时间代价都是 $\Theta(n)$ ，这使得基数排序成为我们所讨论过的具有最好渐近复杂性的排序算法。

# 各种排序方法时间性能

## 1. 平均的时间性能

时间复杂度为  $O(n \log n)$  :

快速排序、堆排序和归并排序

时间复杂度为  $O(n^2)$  :

直接插入排序、起泡排序和  
简单选择排序

时间复杂度为  $O(n)$  :

基数排序



# 各种排序方法时间性能

## 2. 当待排记录序列按关键字顺序有序时

直接插入排序和起泡排序能达到 $O(n)$ 的时间复杂度，

快速排序的时间性能蜕化为 $O(n^2)$ 。

3. 简单选择排序、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。

# 各种排序方法空间性能

指的是排序过程中所需的辅助空间大小

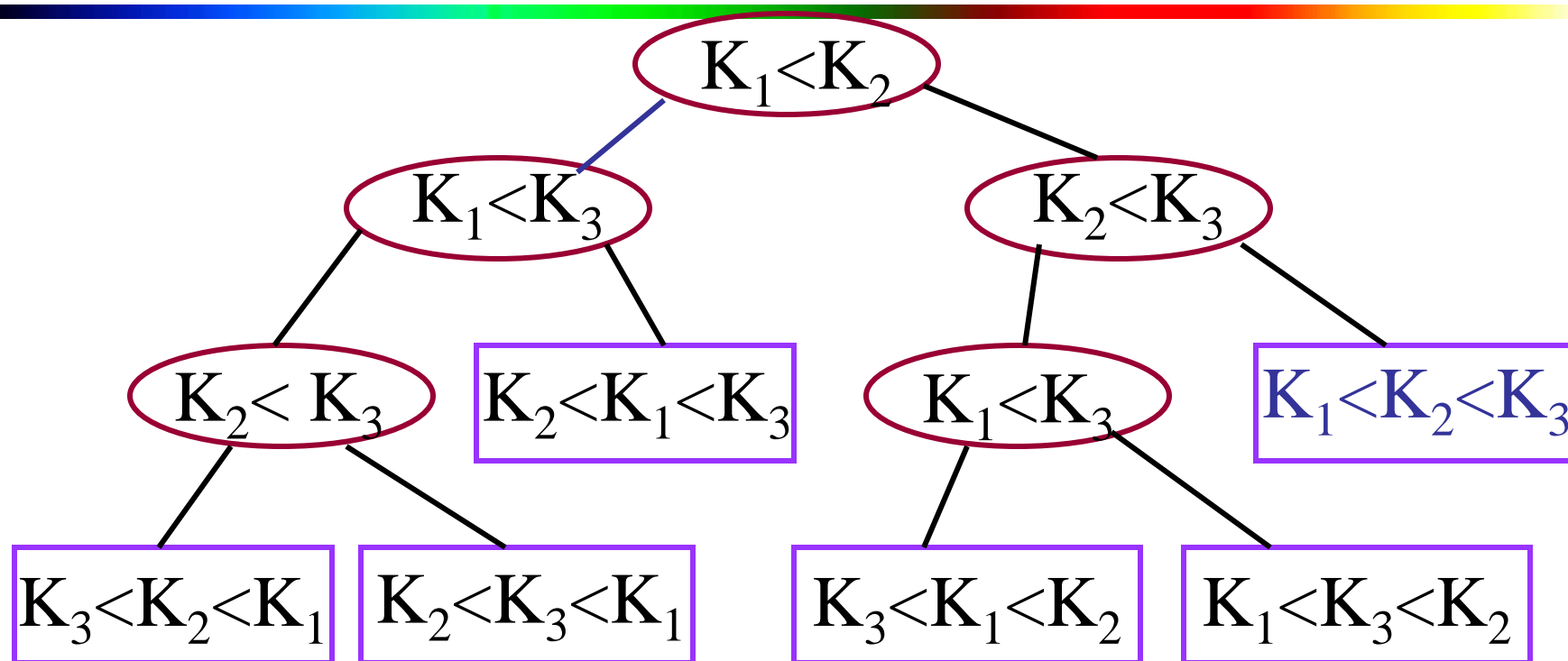
1. 所有的简单排序方法（包括：直接插入、起泡和简单选择）和堆排序的空间复杂度为 $O(1)$ ；
2. 快速排序为 $O(\log n)$ ，为递归程序执行过程中，栈所需的辅助空间；
3. 归并排序所需辅助空间最多，其空间复杂度为 $O(n)$ ；

# 排序方法的时间复杂度的下限

本章讨论的各种排序方法，除基数排序外，其它方法都是**基于“比较关键字”进行排序的排序方法。**

可以证明，这类排序法**可能达到的最快的时间复杂度为 $O(n\log n)$** 。（基数排序不是基于“比较关键字”的排序方法，所以它不受这个限制。）

例如:对三个关键字进行排序的判定树如下:



1. 树上的每一次“比较”都是必要的
2. 树上的叶子结点包含所有可能情况。

一般情况下，对 $n$ 个关键字进行排序，可能得到的结果有 $n!$ 种，由于含 $n!$ 个叶子结点的二叉树的深度不小于 $\lceil \log_2(n!) \rceil + 1$ ，则对 $n$ 个关键字进行排序的比较次数至少是 $\lceil \log_2(n!) \rceil \approx n \log_2 n$  (斯蒂林近似公式)。

所以，基于“比较关键字”进行排序的排序方法，可能达到的最快的时间复杂度为  $O(n \log n)$ 。