数据结构与算法

Data Structure and Algorithms

第十九课

西安交通大学自动化系

蔡忠闽 周亚东

第十章 排序

- 10.1 概述
- 10.2 插入排序
- 10.3 交换排序
- 10.4 选择排序
- 10.5 归并排序
- 10.6 基数排序
- 10.7 小结

10.3 交换排序

10.3.1 起泡排序 (Bubble Sort)

- 基本方法：设待排序的文件为r[1..n]。

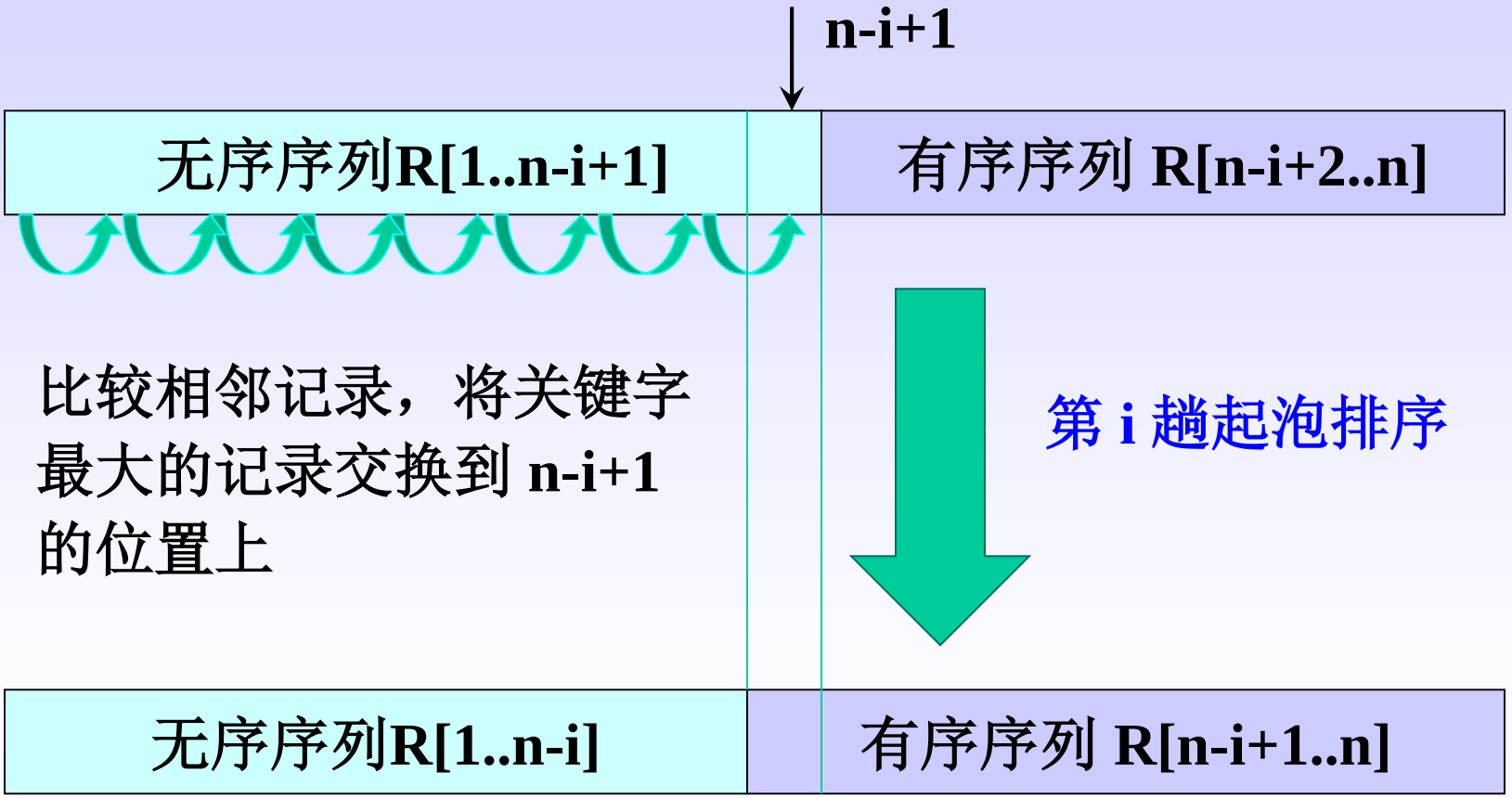
  第1趟：从r[1]开始,依次比较相邻的r[j].key和r[j+1].key, 若r[j].key>r[j+1].key，则交换r[j]和r[j+1] 的位置；否则,不交换。(j=1,2,...n-1)。第1趟之后,n个元素中关键字最大的移到了r[n]的位置上。

  第2趟： 令j=1,2,...n-2，重复上述比较和交换操作。第2趟之后,前n-1个元素中关键字最大的移到了r[n-1]的位置上。 .....

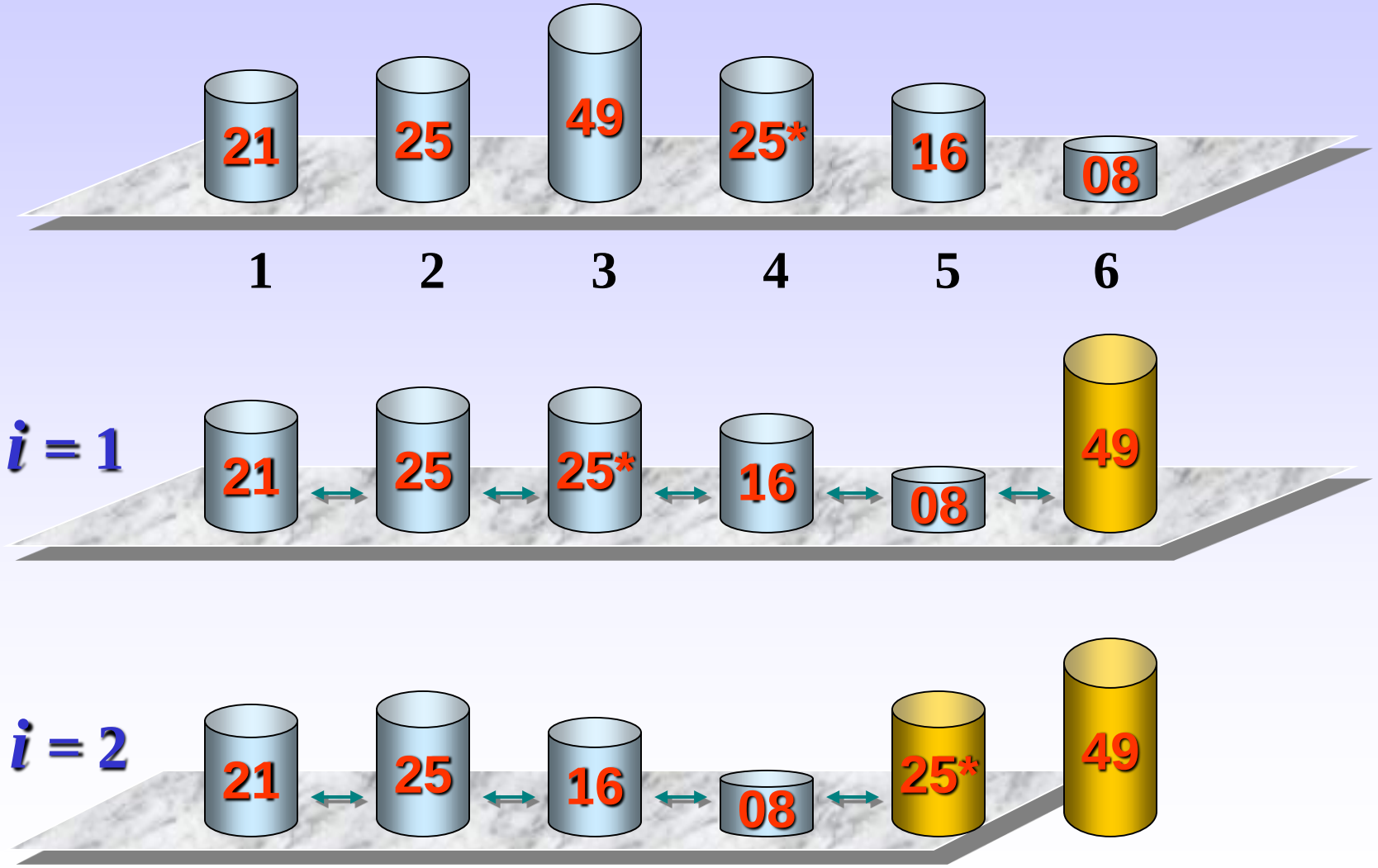
  作完n-1趟，或者不需再交换记录时为止。

10.3 交换排序

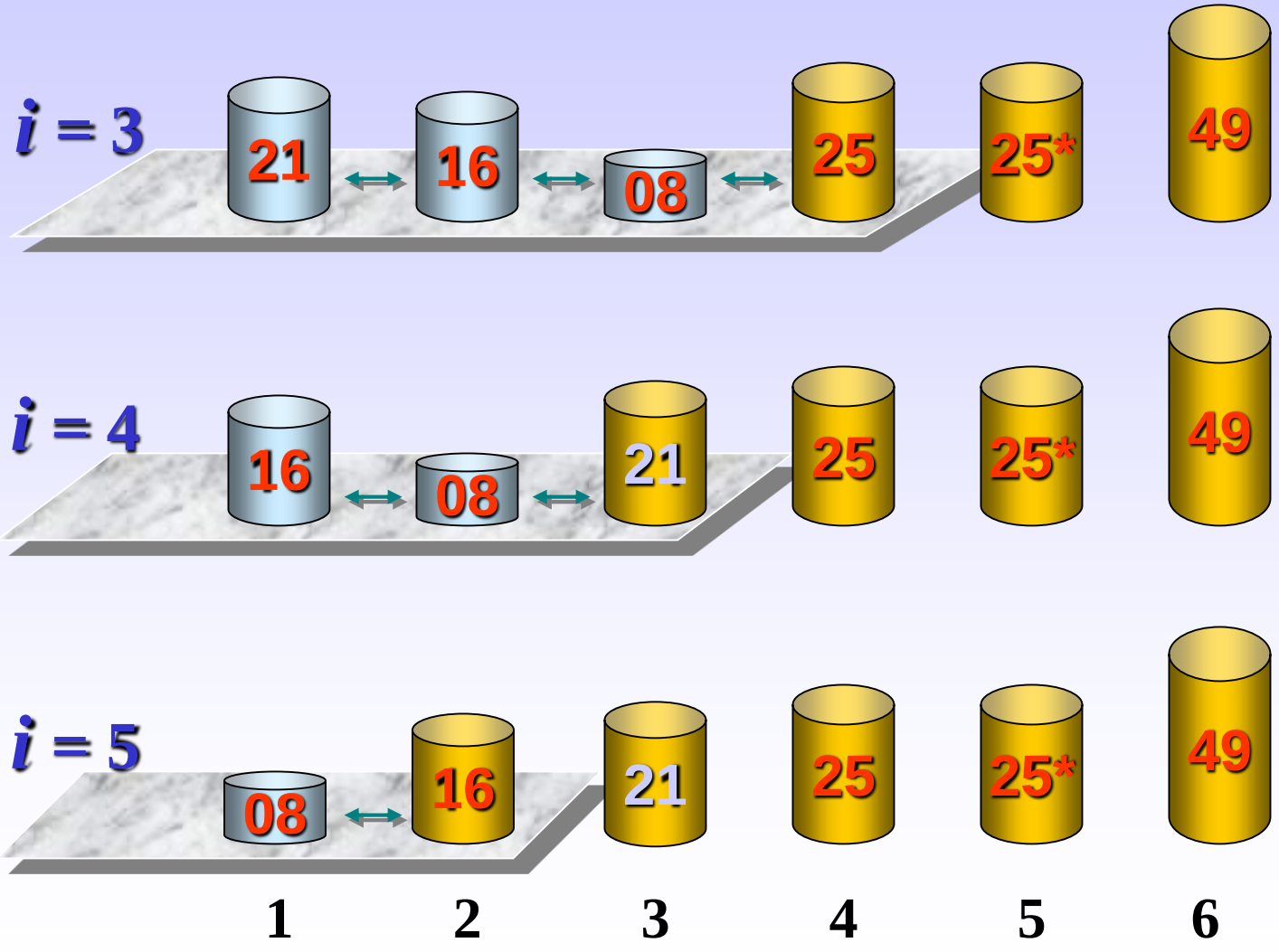
假设在排序过程中，记录序列R[1..n]的状态为：



10.3 交换排序



10.3 交换排序



10.3 交换排序

- 起泡排序算法

```
void  bubble(SqList &L)
{   swapflag=TRUE;
    for(i=1; (i<L.length) && swapflag;  i++)    //作n-1趟排序，退出条件
    {   swapflag=FALSE;                 //或者不需再交换记录为止
        for(j=1; j<=L.length-i; j++)
            if (L.r[j].key>L.r[j+1].key)
            {   L.r[j]<->L.r[j+1];      //交换记录
                swapflag=TRUE;
            }
    }
}
```

10.3 交换排序

算法分析

- 在元素的初始排列已经按关键字从小到大排好序时，此算法只执行一趟起泡，做 n-1 次关键字比较，不移动元素。这是最好的情形。
- 最坏的情形是算法执行了n-1趟起泡，第 i 趟 $(1 \le i < n)$ 做了 n-i 次关键字比较，执行了n-i 次元素交换。总的关键字比较次数和元素移动次数为：

$$\text{比较次数} = \sum_{i=1}^{n-1}(n-i) = \frac{n(n-1)}{2} \qquad \text{移动次数} = 3\sum_{i=1}^{n-1}(n-i) = \frac{3n(n-1)}{2}$$

- 起泡排序是一个稳定的排序方法。

10.3 交换排序

10.3.2 快速排序 (Quick Sort)

- 基本思想：首先在r[1..n]中，确定一个r[i]，经过比较和移动，将r[i]放到“中间”某个位置上，使得r[i]左边所有记录的关键字小于等于r[i].key，r[i]右边所有记录的关键字大于等于r[i].key。以r[i]为界,将文件划分为左、右两个子文件。用同样的方法分别对这两个子文件进行划分, 得到4个更小的子文件。继续进行下去,使得每个子文件只有一个记录为止, 便得到原文件的有序文件。


1 2 3 4 5 6
pivot
21 25 49 25* 16 08
21
i
j
08 25 49 25* 16 08
21
i
j
将1个序列
划分为2个
子序列
08 25 49 25* 16 25
21
i
j
08 16 49 25* 49 25
21
i
j
08 16 49 25* 49 25
21
ij
08 16 21 25* 49 25

```
int Partition(Sqlist L, int low, int high)
 {   L.r[0] = L.r[low];
     pivotkey = L.r[low].key;
     while (low < high) {
          while(low < high && L.r[high].key >=  pivotkey )   --high;
          L.r[low] = L.r[high];
          while(low < high && L.r[low].key <=  pivotkey )    ++low;
          L.r[high] = L.r[low];
     }
     L.r[low] = L.r[0];
     return low;
}
```

10.3 交换排序

快速排序过程

10.3 交换排序

```
void quksort (Sqlist L, int low, int high)
{   if (low<high) {                          // 有两个以上记录
    pivotloc = Partition(L, low, high);      // 划分结束
    quksort(L, low, pivotloc -1);            // 递归处理左子文件
    quksort(L, pivotloc+1, high);            // 递归处理右子文件
   }
}
```

10.3 交换排序

算法分析

- 快速排序是递归的，需要一个栈存放每层递归调用时的指针和参数。
- 快速排序的平均时间开销为$knln(n)$，k为某个常数。
- 实验结果表明：就平均情况而言，快速排序是所讨论的所有内部排序方法中最好的一个。
- 最大递归调用层次数在理想情况为$\lceil \log_2(n+1) \rceil$.

10.3 交换排序

- 设C(n)表示对长度为n的序列进行快速排序所需的比较次数，显然，它应该等于对长度为n的无序区进行划分所需的比较次数n-1，加上递归地对划分所得的左右两个无序子区进行快速排序所需的比较次数。

10.3 交换排序

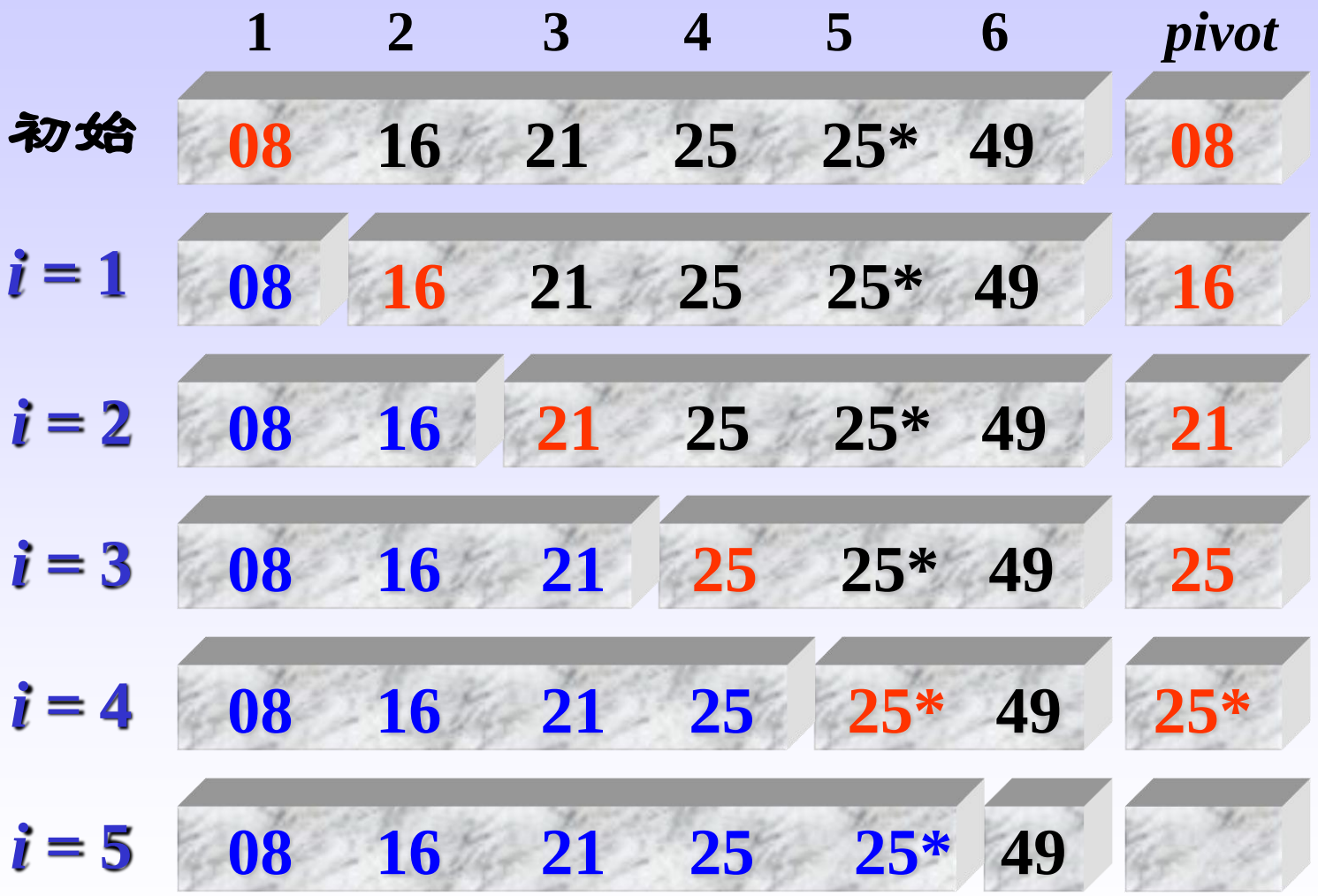
- 最好的情况，假设文件长度$n=2^k$，$k=\log_2 n$:

$$
\begin{aligned}
&C(n)\\
&\le n+2C(n/2)\\
&\le n+2[n/2+2C(n/2^2)]=2n+4C(n/2^2)\\
&\le 2n+4[n/4+2C(n/2^3)]=3n+8C(n/2^3)\\
&\le \cdots\cdots\\
&\le kn+2^kC(n/2^k)=n\log_2 n+nC(1)\\
&=O(n\log_2 n)
\end{aligned}
$$

10.3 交换排序

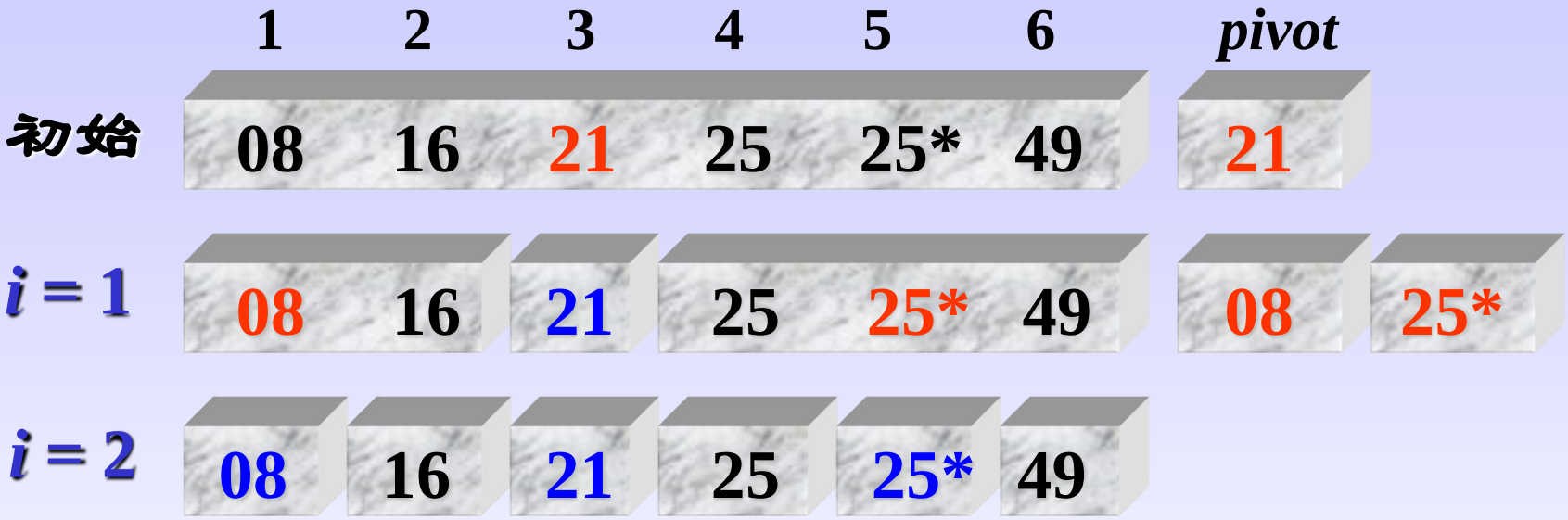
- 在最坏的情况，即待排序元素序列已经按其关键字从小到大排好序的情况下，每次划分只得到一个比上一次少一个元素的子序列。这样，必须经过*n*-1趟才能把所有元素定位，而且第*i*趟需要经过*n*-*i* 次关键字比较才能找到第*i*个元素的安放位置，总的关键字比较次数将达到：$\sum_{i=1}^{n-1}(n-i)=\frac{1}{2}n(n-1)\approx\frac{n^2}{2}$
- 其排序速度退化到简单排序的水平，比直接插入排序还慢。占用附加存储(即栈)将达到o(*n*)。

10.3 交换排序



快速排序退化的例子

10.3 交换排序



用关键字居中的元素作为基准

- 改进办法：取待排序元素序列的第一个、最后一个和位置正中的3个元素，取其关键字居中者作为基准元素。
- 快速排序是一种不稳定的排序方法。

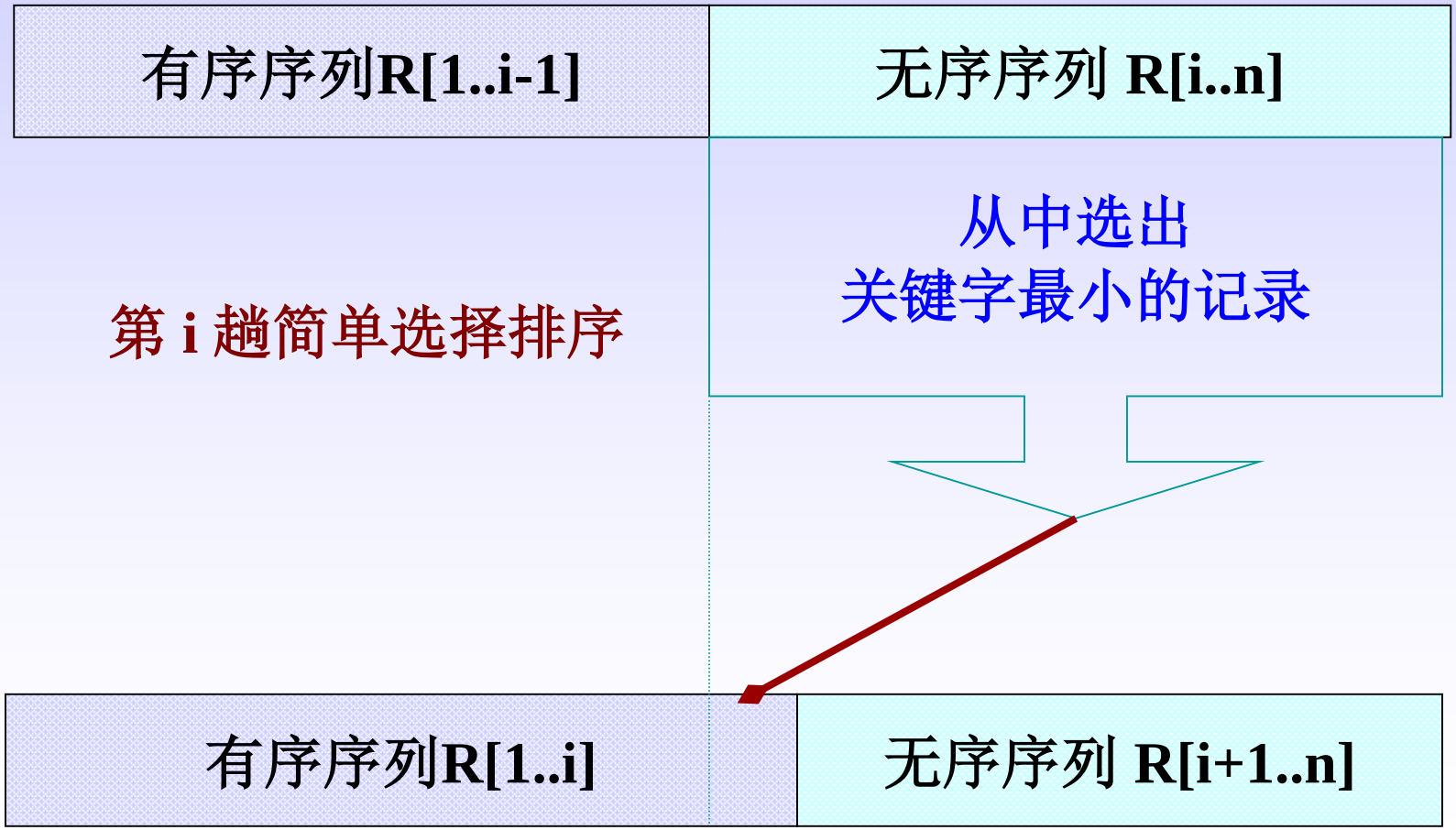
10.4 选择排序

10.4.1 直接选择排序 (Select Sort)

- 直接选择排序是一种简单的排序方法，其基本步骤是： 设待排序的文件为(r[1],r[2],...,r[n]),
  第1趟(遍)：在(r[1],r[2],...,r[n])中,选出关键字最小的记录r[min].key，若min<>1,则交换r[1]和r[min]；
  第2趟(遍)：在n-1个记录(r[2],...,r[n])中,选出关键字最小的记录r[min].key，若min<>2,则交换r[2]和r[min]； .....
  第n-1趟(遍)：在最后的2个记录记录(r[n-1],r[n])中,选出关键字最小的记录r[min].key，若min<>n-1,则交换 r[n-1]和r[min]。

10.4 选择排序

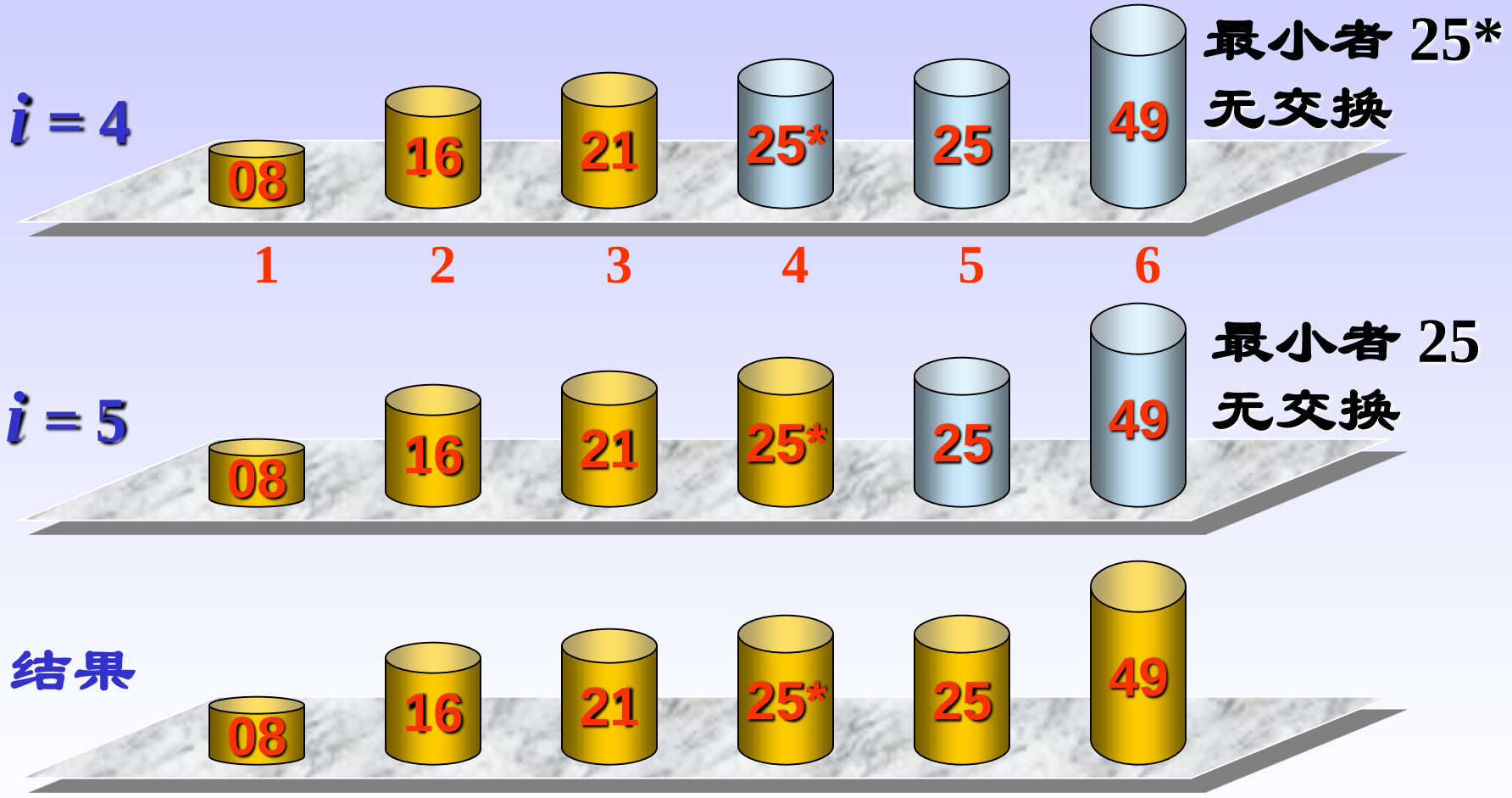
假设排序过程中，待排记录序列的状态为：



10.4 选择排序


初始
21
25
49
25*
16
08
1
2
3
4
5
6
i = 1
最小者 08
交换21,08
21
25
49
25*
16
08
i = 2
最小者 16
交换25,16
08
25
49
25*
16
21
i = 3
最小者 21
交换49,21
08
16
49
25*
25
21

10.4 选择排序



各趟排序后的结果

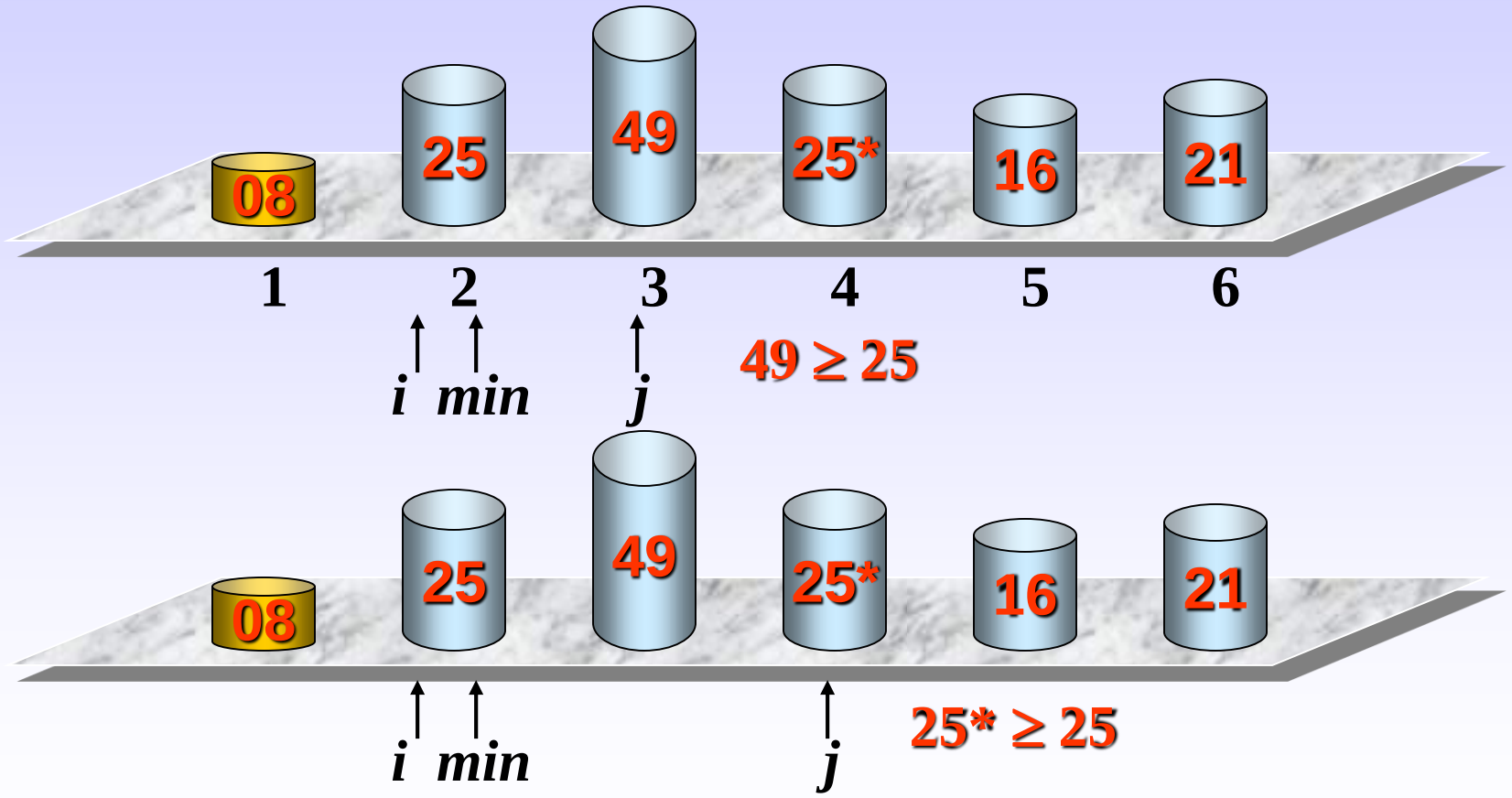
10.4 选择排序

简单选择排序算法：

```
void SelectSort(SqList &L)          // 对顺序表L作简单选择排序
{   for (i=1; i<L.length; i++)      //共n-1趟
    {   min=i;                      //r[i]为最小记录r[min]
        for (j=i+1; j<=L.length; j++)
            if (L.r[j].key<L.r[min].key)
                min=j;              //修改min
        if (min!=i)
            L.r[i]<->L.r[min];      //交换r[min]和r[i]
    }
}
```

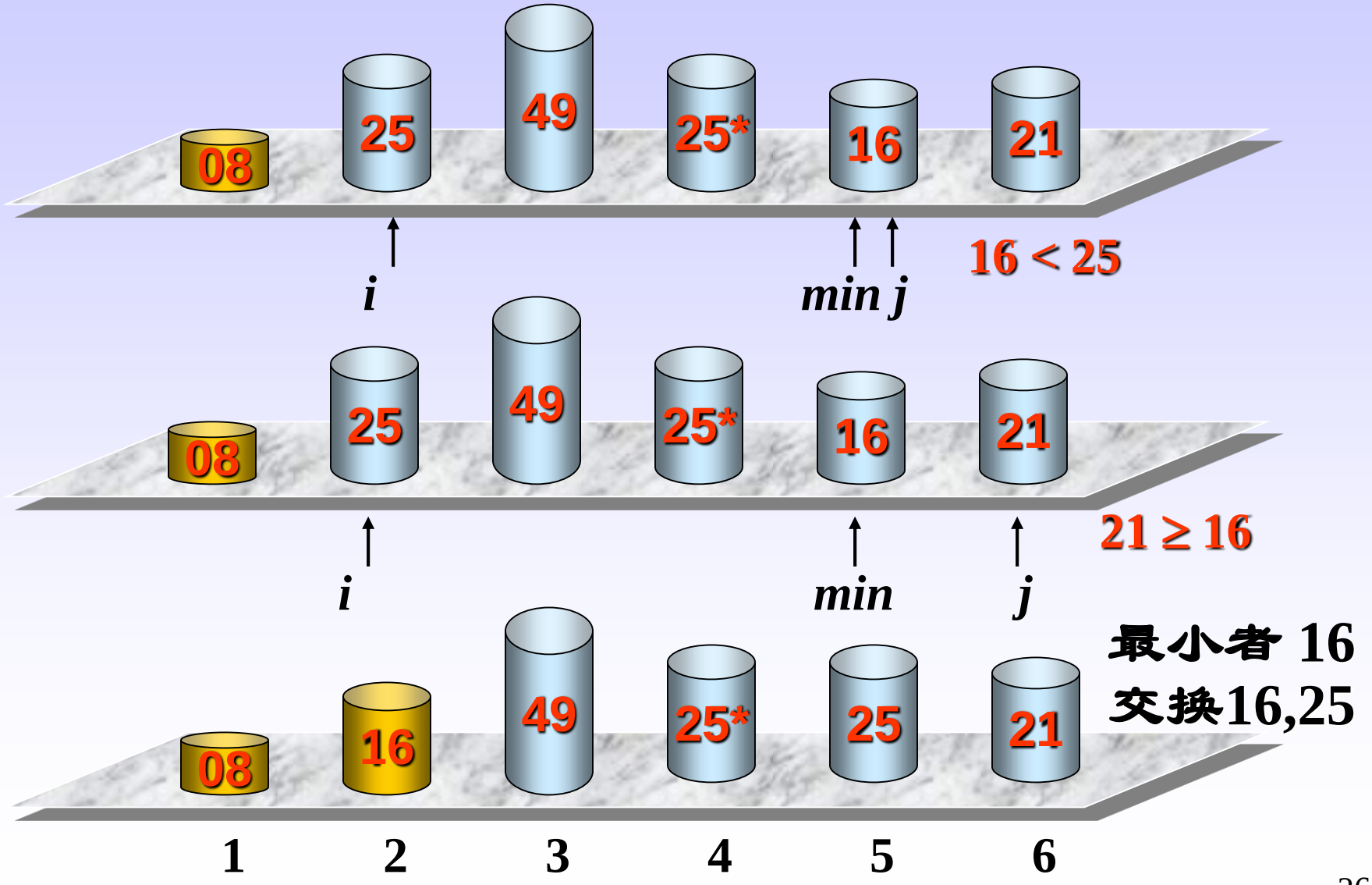
10.4 选择排序

i=2时选择排序的过程



*min*指示当前序列中最小者

10.4 选择排序


08
25
49
25*
16
21
i
min j
16 < 25
08
25
49
25*
16
21
i
min
j
21 ≥ 16
08
16
49
25*
25
21
最小者 16
交换16,25
1
2
3
4
5
6

10.4 选择排序

算法分析

- 第*i*趟选择具有最小关键字元素所需的比较次数总是*n-i* 次。因此，总的关键字比较次数为：

$$\text{比较次数} = \sum_{i=1}^{n-1}(n-i) = \frac{n(n-1)}{2}$$

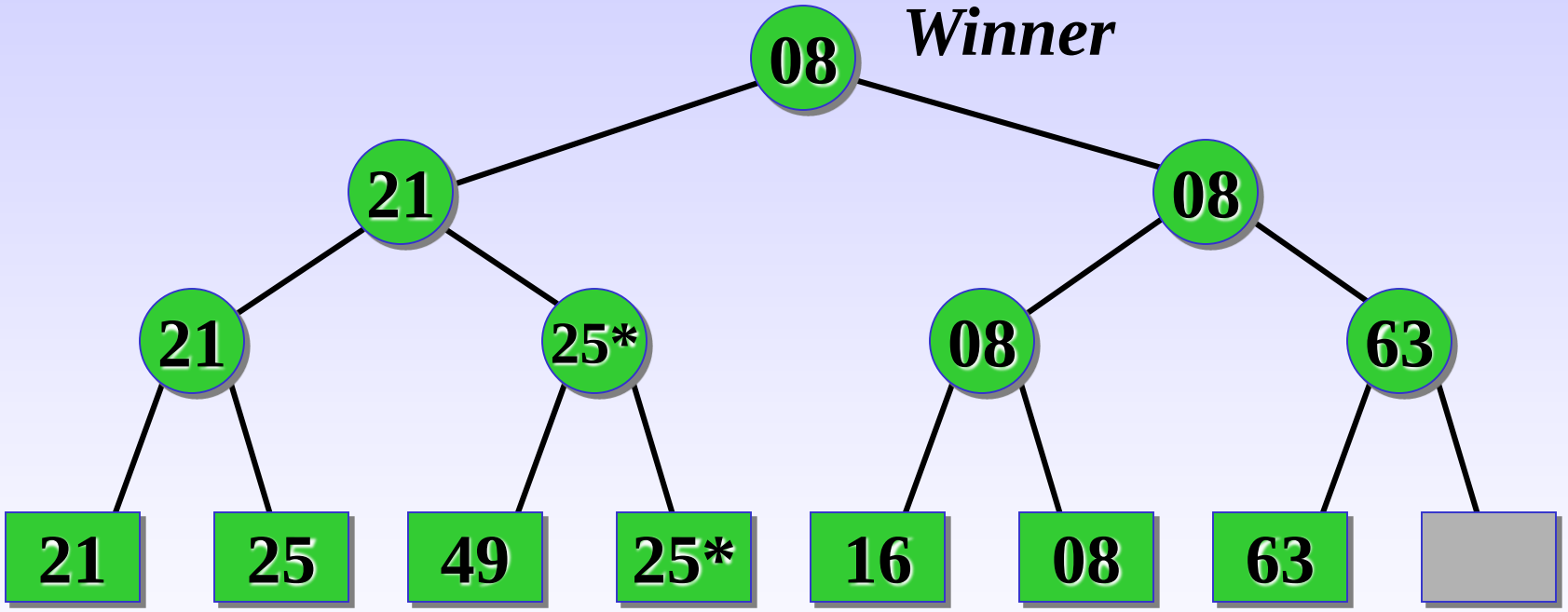
- 当元素的初始状态是按其关键字从小到大有序的时候，元素的移动次数为0。最坏情况是每一趟都要进行交换，总的元素移动次数为3(*n*-1)。
- 直接选择排序是一种不稳定的排序方法。

10.4 选择排序

10.4.2 树形选择排序 (Tree Selection Sort)

- 树形选择排序的思想与体育比赛时的淘汰赛类似。首先取n个元素的关键字，进行两两比较，得到$\lceil n/2 \rceil$个比较的优胜者(关键字小者)，然后对这$\lceil n/2 \rceil$个元素再进行关键字的两两比较，…，如此重复，直到选出一个关键字最小的元素为止。在图例中，最下面是元素排列的初始状态，相当于一棵满二叉树的叶结点，它存放的是所有参加排序的元素的关键字。

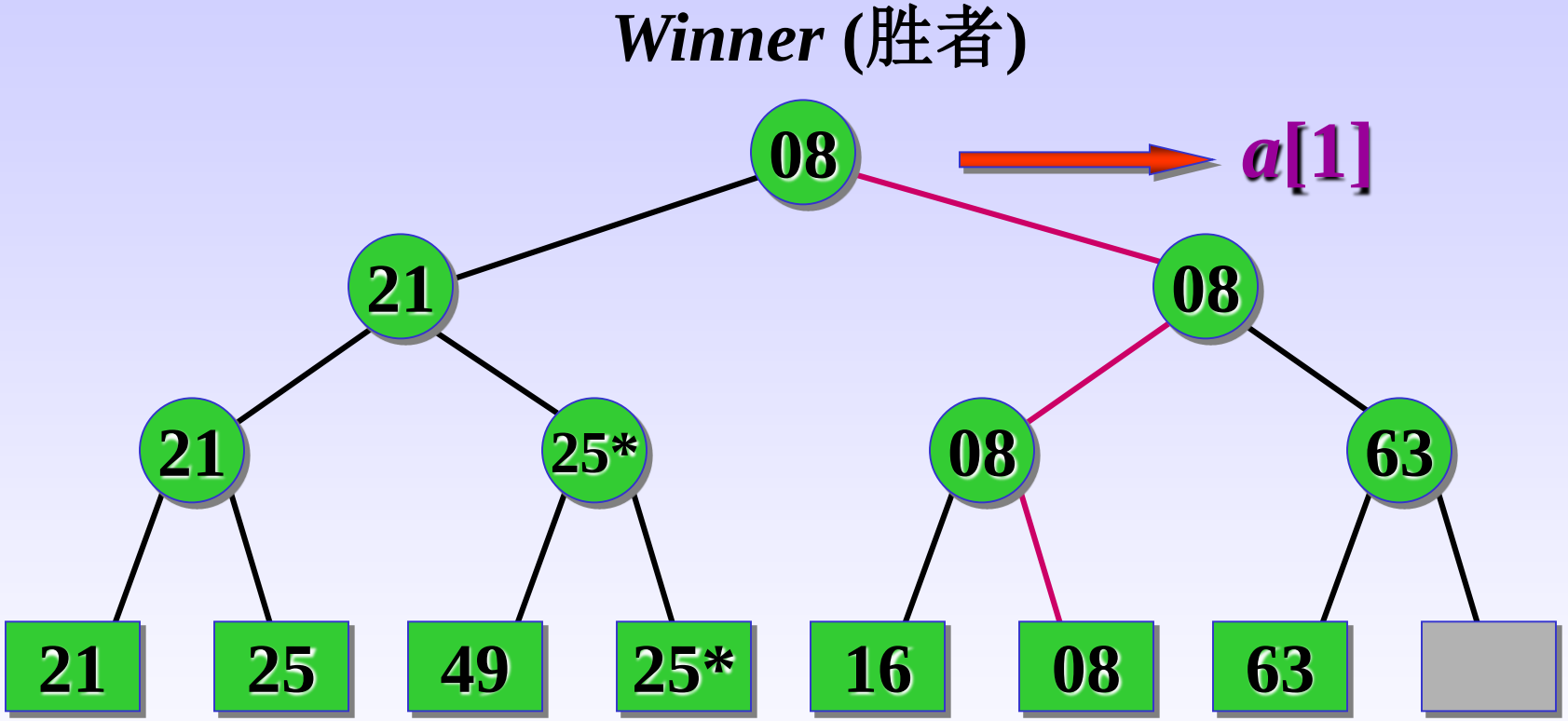
10.4 选择排序



10.4 选择排序

- 如果 n 不是2的 k 次幂，则让叶结点数补足到满足 $2^{k-1} < n \le 2^k$ 的2^k个。叶结点上面一层的非叶结点是叶结点关键字两两比较的结果。
- 每次两两比较的结果是把关键字小者作为优胜者上升到双亲结点，称这种比赛树为胜者树。
- 每个结点除了存放元素的关键字 *data* 外，还存放了此元素是否要参选的标志 *Active* 和此元素在满二叉树中的序号*index*。
- 胜者树最顶层是树的根，表示最后选择出来的具有最小关键字的元素。

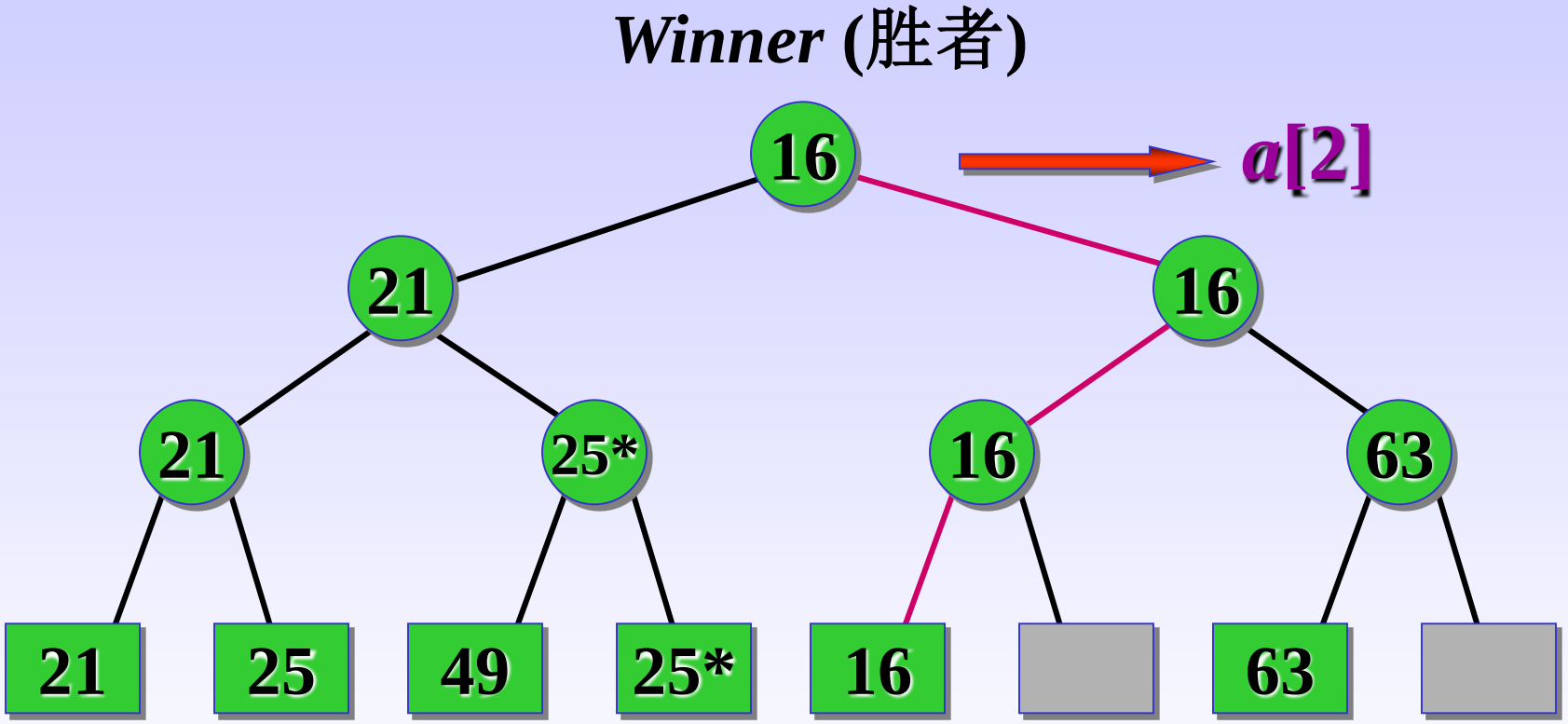
10.4 选择排序



形成初始胜者树（最小关键字上升到根）

关键字比较次数：6

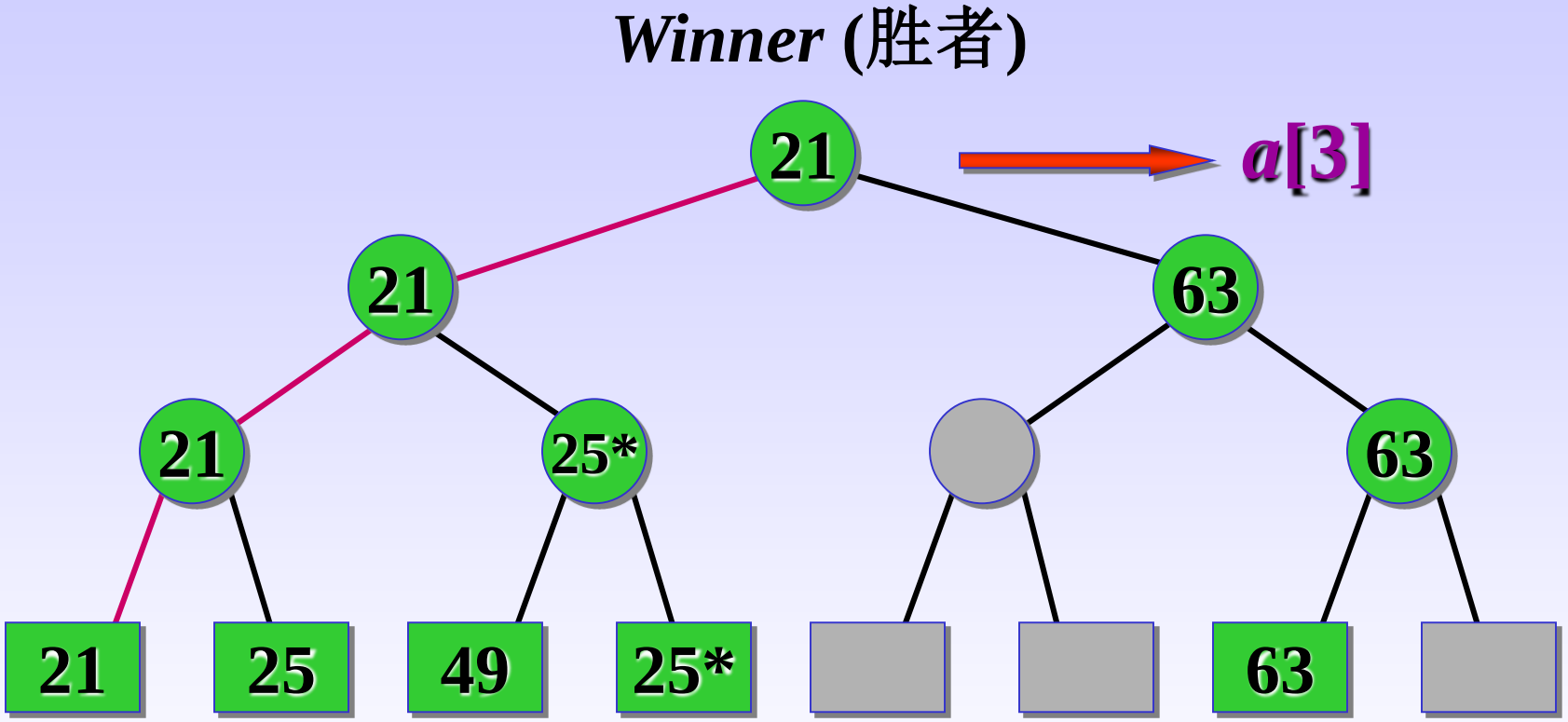
10.4 选择排序



输出冠军并调整胜者树后树的状态

关键字比较次数：2

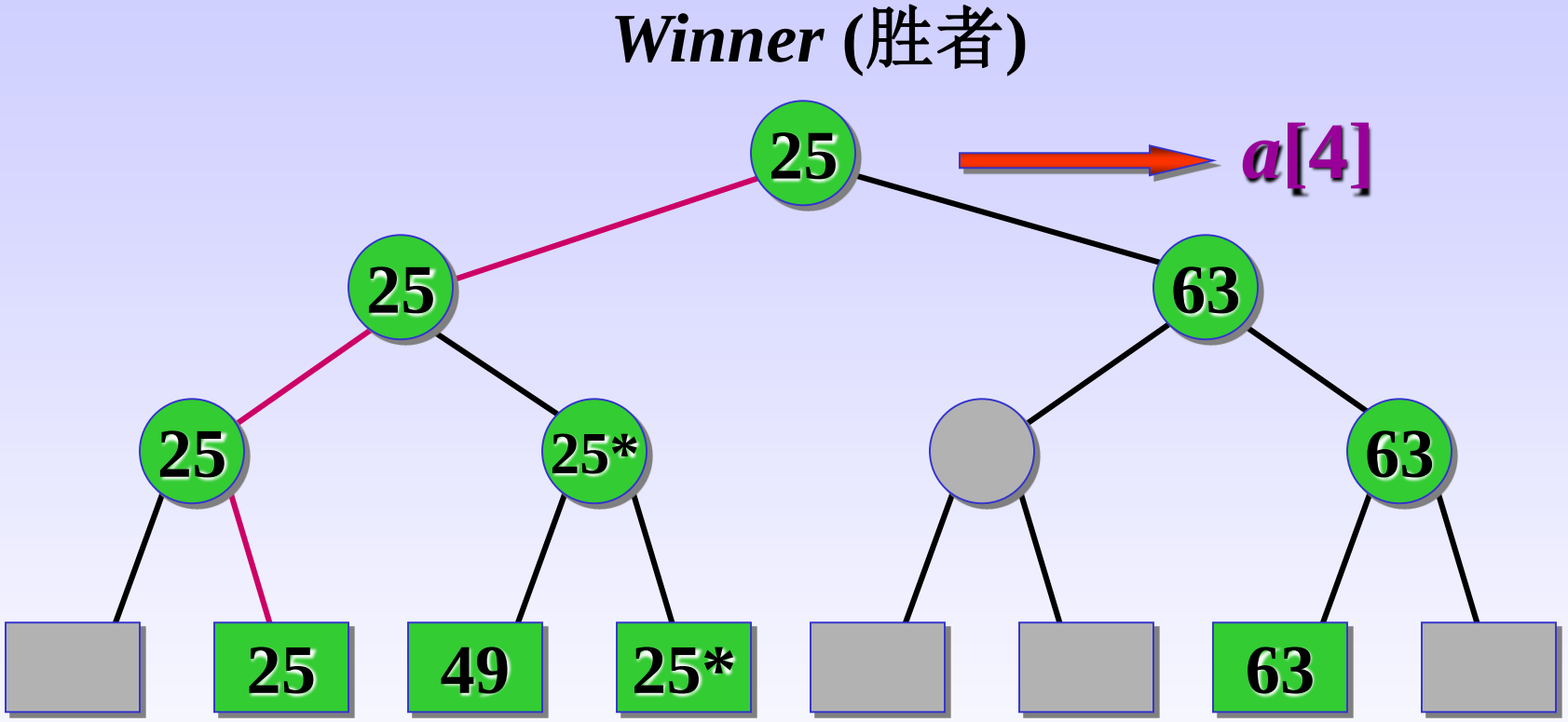
10.4 选择排序



输出亚军并调整胜者树后树的状态

关键字比较次数：2

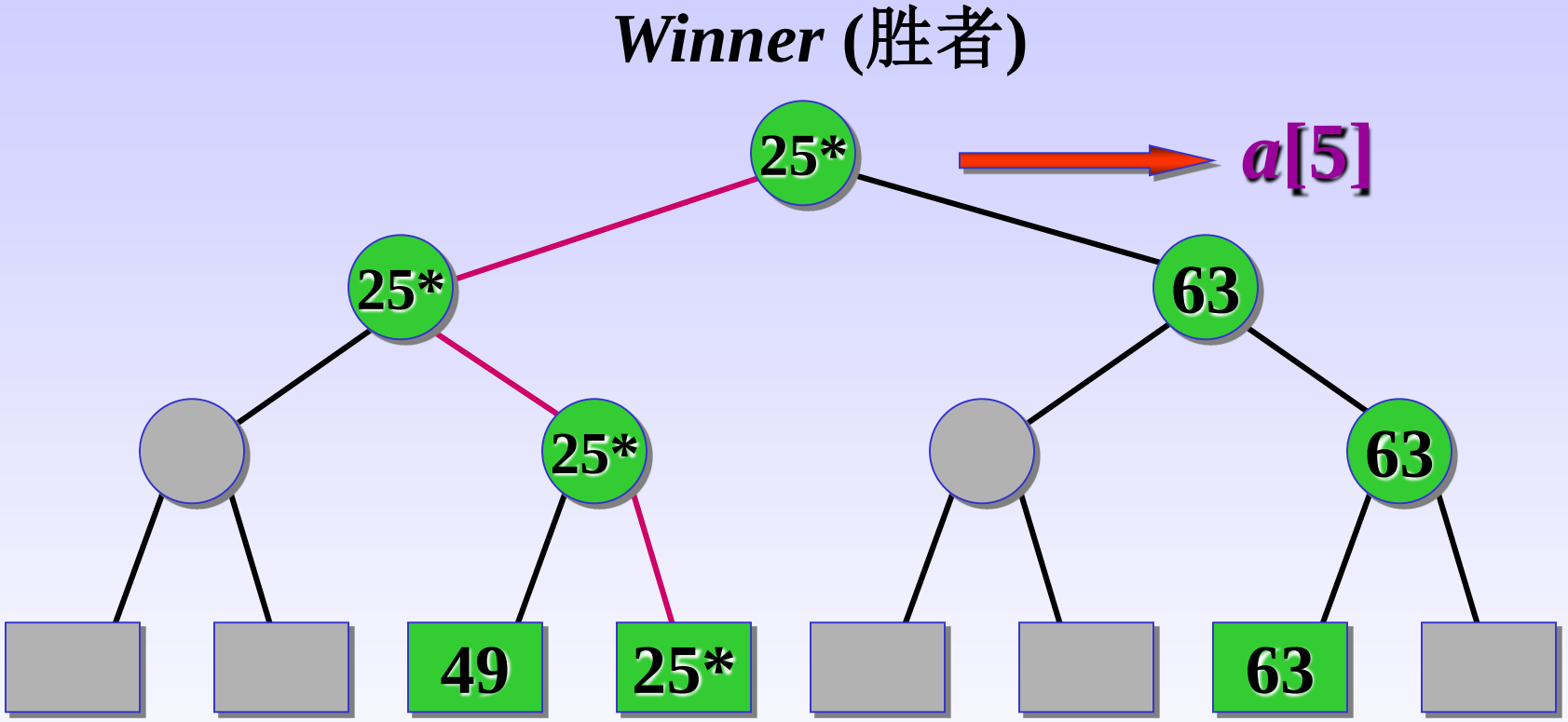
10.4 选择排序



输出第三名并调整胜者树后树的状态

关键字比较次数：2

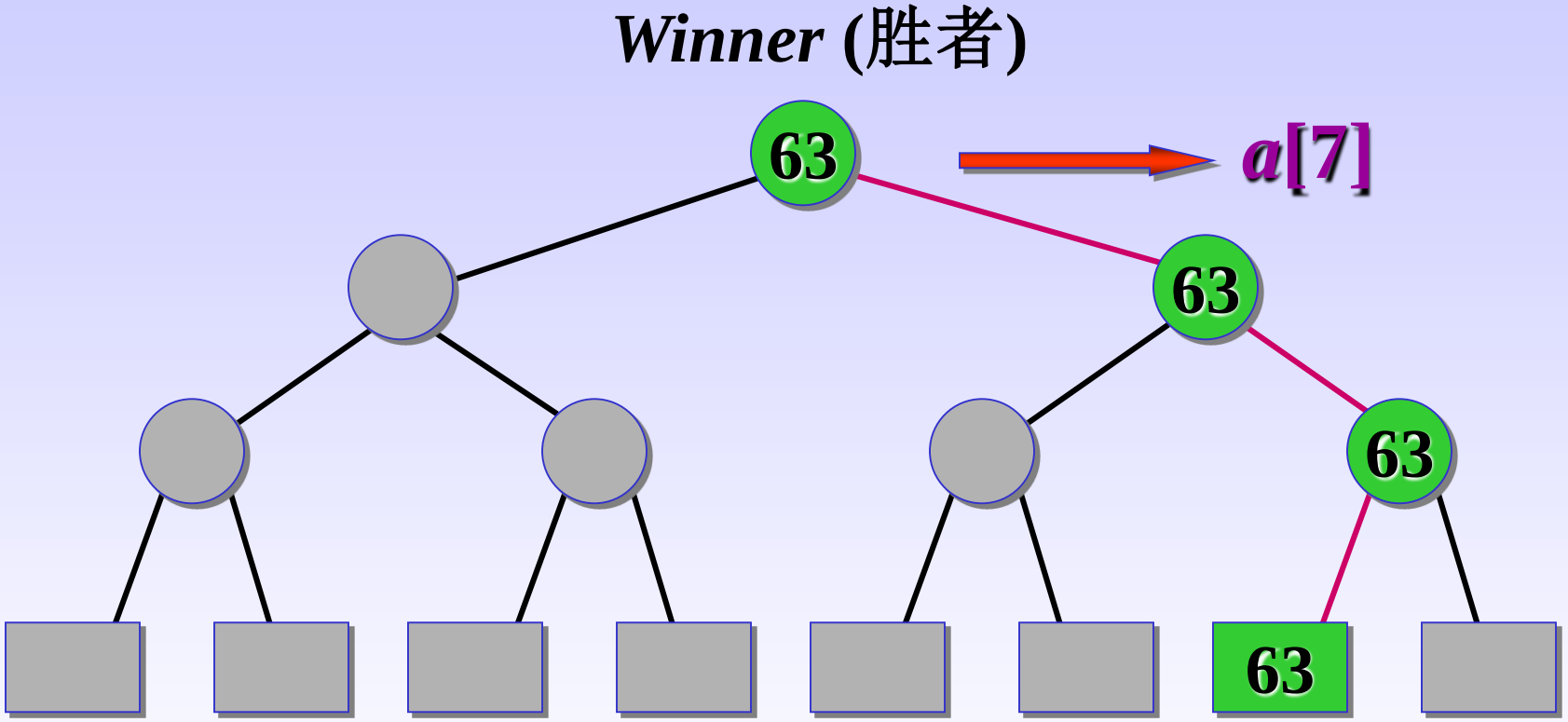
10.4 选择排序



输出第四名并调整胜者树后树的状态

关键字比较次数：2

10.4 选择排序



全部比赛结果输出时树的状态

关键字比较次数：2

10.4 选择排序

- 胜者树是满的完全二叉树，其深度为$\lceil \log_2(n+1) \rceil$。
- 除第一次选择具有最小关键字的元素需要进行 n-1 次关键字比较外，重构胜者树选择具有次小、再次小关键字元素所需的关键字比较次数均为 $O(\log_2 n)$。总关键字比较次数为$O(n\log_2 n)$。
- 元素的移动次数不超过关键字的比较次数，所以锦标赛排序总的时间复杂度为$O(n\log_2 n)$。
- 这种排序方法虽然减少了排序时间，但是使用了较多的附加存储。
- 锦标赛排序是一个稳定的排序方法。