第10章 内部排序

插入排序（直接插入、折半插入、希尔排序）

交换排序（起泡排序、快速排序）

选择排序（简单选择排序、树形选择排序、堆排序）

归并排序

基数排序

概述

- 排序：将一组杂乱无章的数据排列成一个按关键字有序的序列。
- 数据表(datalist)：待排序数据对象的有限集合。
- 关键字(key)：通常数据对象有多个属性域，即多个数据成员组成，其中有一个属性域可用来区分对象，作为排序依据，该域即为关键字。

每个数据表用哪个属性域作为关键字，要视具体的应用需要而定。即使同一个表，在解决不同问题的场合也可能取不同的域做关键字。

- 主关键字：如果在数据表中各个对象的关键字互不相同，这种关键字即主关键字。按照主关键字进行排序，排序的结果是唯一的。
- 次关键字：数据表中有些对象的关键字可能相同，这种关键字称为次关键字。按照次关键字进行排序，排序的结果可能不唯一。
- 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]，它们的关键字 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。

例如，排序前关键字序列是：34 12 34* 08 96

若排序后：08 12 34 34* 96　　稳定的

若排序后：08 12 34* 34 96　　不稳定的

- 内排序与外排序：内排序是指在排序期间数据对象全部存放在内存的排序；外排序是指在排序期间全部对象个数太多，不能同时存放在内存，必须根据排序过程的要求，不断在内、外存之间移动的排序。
- 排序的时间开销：
  - 排序的时间开销是衡量算法好坏的最重要的标志。
  - 排序的时间代价可用算法执行中的数据比较次数与数据移动次数来衡量。
  - 通常给出的算法运行时间代价的大略估算一般都按平均情况进行估算。对于那些受对象关键字序列初始排列及对象个数影响较大的，需要按最好情况和最坏情况进行估算。
- 算法执行时所需的附加存储：衡量算法好坏的另一标准。

- 静态排序：排序的过程是对数据对象本身进行物理地重排，经过比较和判断，将对象移到合适的位置。这时，数据对象一般都存放在一个顺序的表中。
- 动态排序：给每个对象增加一个链接指针，在排序的过程中不移动对象或传送数据，仅通过修改链接指针来改变对象之间的逻辑顺序，从而达到排序的目的。
- 部分排序：有些排序算法在每趟排序过程中，都会按次序找出最大（或最小）的元素被放置在其最终的位置上（就位）。

排序算法的衡量标准

1.排序的时间代价
- 排序时所需要的平均比较次数
- 排序时所需要的平均移动次数

2.排序的空间代价
- 排序时所需要的平均辅助存储

3.算法本身的繁杂程度

数据表的类型定义

```
#define MAXSIZE 20//示例一个小的顺序表的最大长度
typedef int KeyType;  //关键字类型
typedef struct{
    KeyType key;   //关键字项
    InfoType otherinfo; //其他记录项
}RedType; //记录类型
typedef struct{
    RedType r[MAXSIZE+1]; //r[0]闲置，用作哨兵单元
    int length;  //顺序表长
}SqList;   //顺序表类型
```

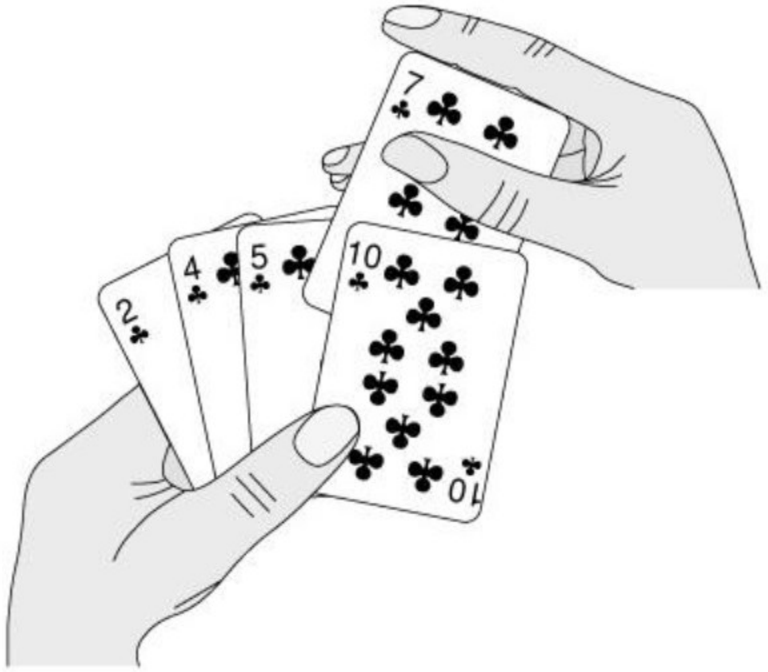
插入排序 (Insert Sorting)

- 基本方法：每步将一个待排序的对象，按其关键字大小，插入到前面已经排好序的一组对象的适当位置上，直到对象全部插入为止。

直接插入排序

折半插入排序

Shell(希尔)排序

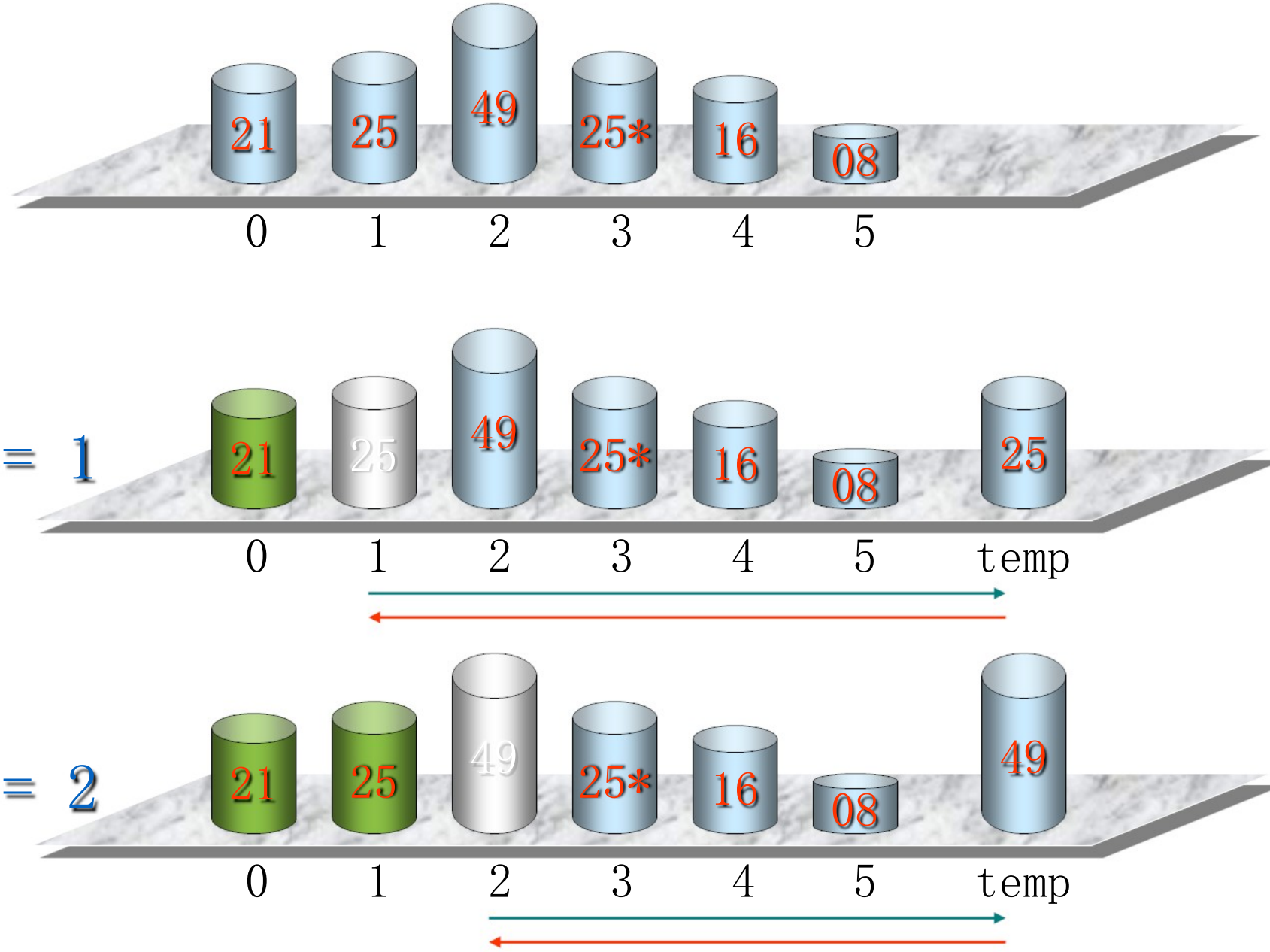


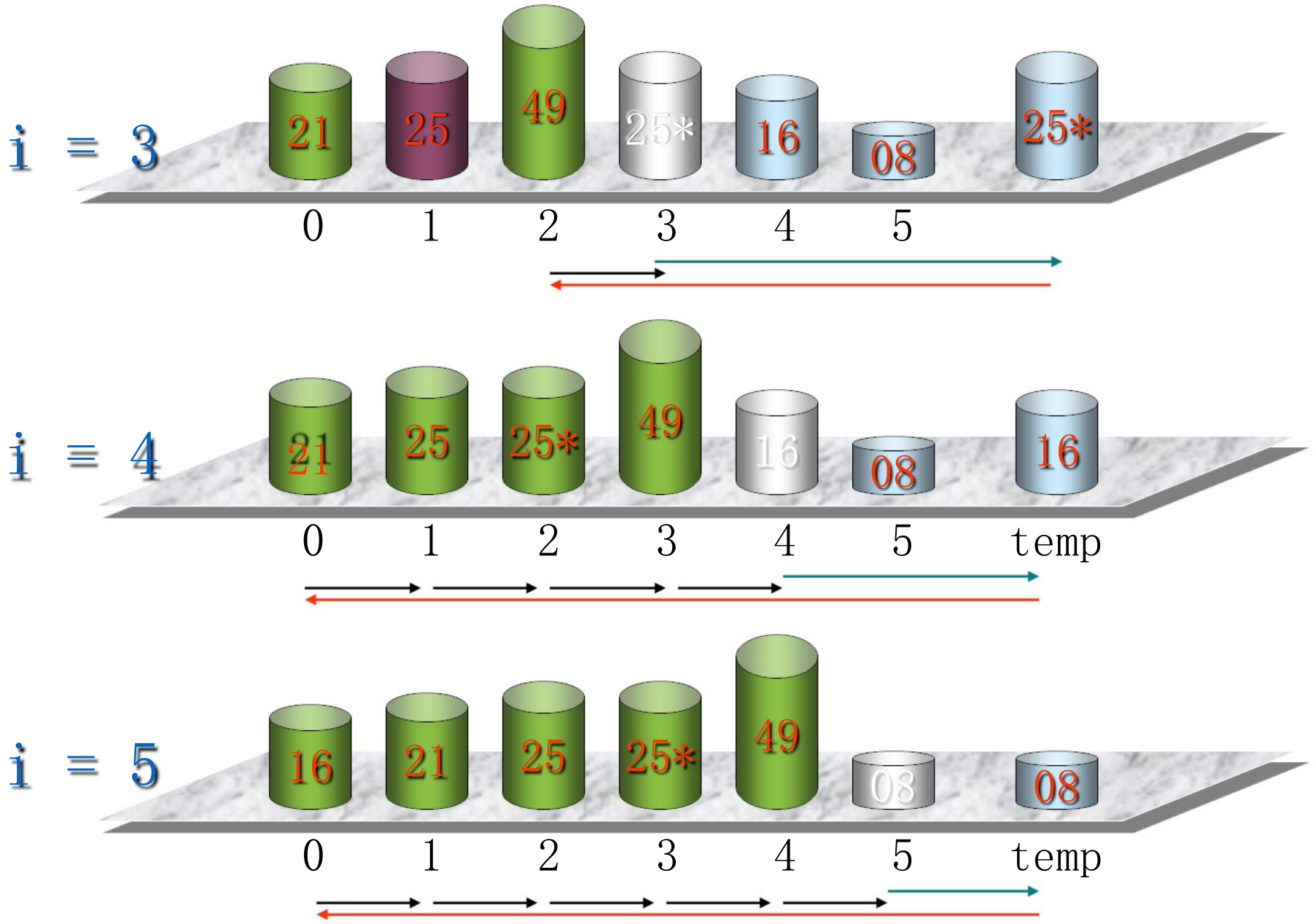
直接插入排序 (Insert Sorting)

- 基本思想:

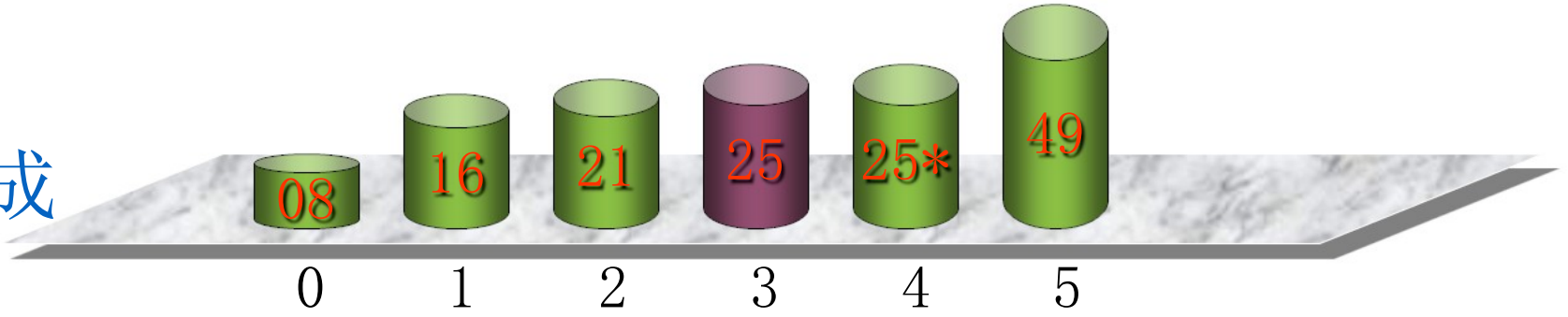
  当插入第i (i >= 1) 个对象时，前面的V[0],V[1],…, v[i-1]已经排好序。这时，用v[i]的关键字与v[i-1],v[i-2],…的关键字顺序进行比较，找到插入位置即将v[i]插入，原来位置上之后的所有对象依次向后顺移。

各趟排序结果


21
25
49
25*
16
08
0
1
2
3
4
5
i = 1
21
25
49
25*
16
08
25
0
1
2
3
4
5
temp
i = 2
21
25
49
25*
16
08
49
0
1
2
3
4
5
temp

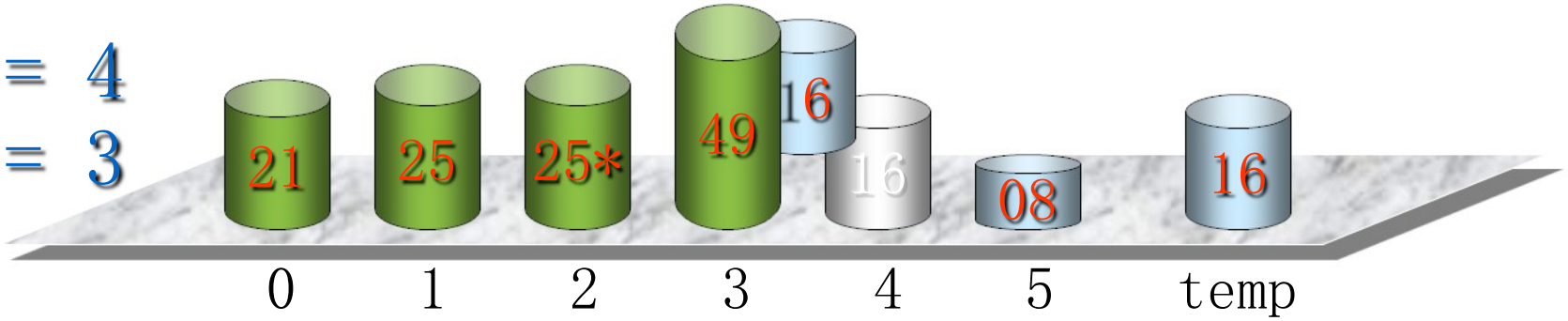

i = 3
21
25
49
25*
16
08
25*
0
1
2
3
4
5
i = 4
21
25
25*
49
16
08
16
0
1
2
3
4
5
temp
i = 5
16
21
25
25*
49
08
08
0
1
2
3
4
5
temp

完成

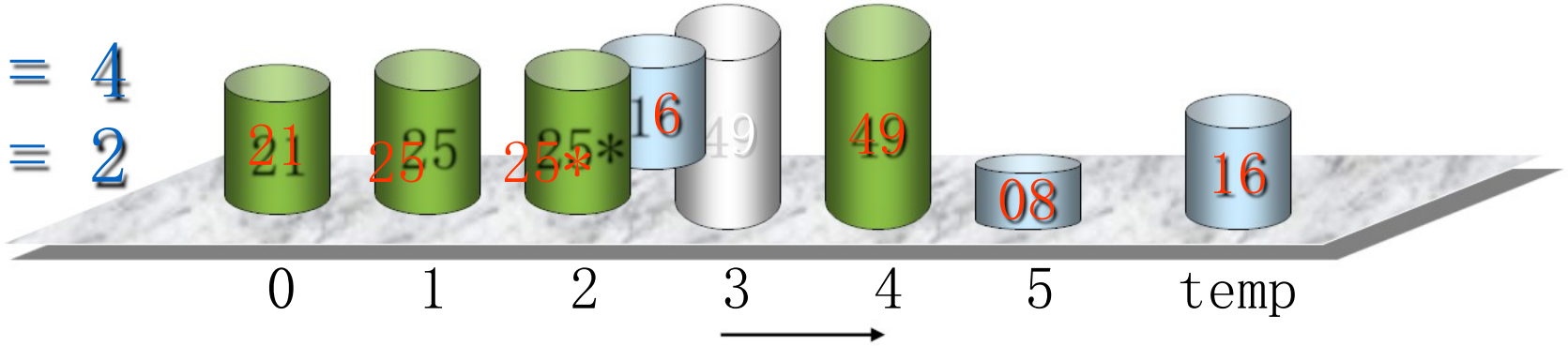


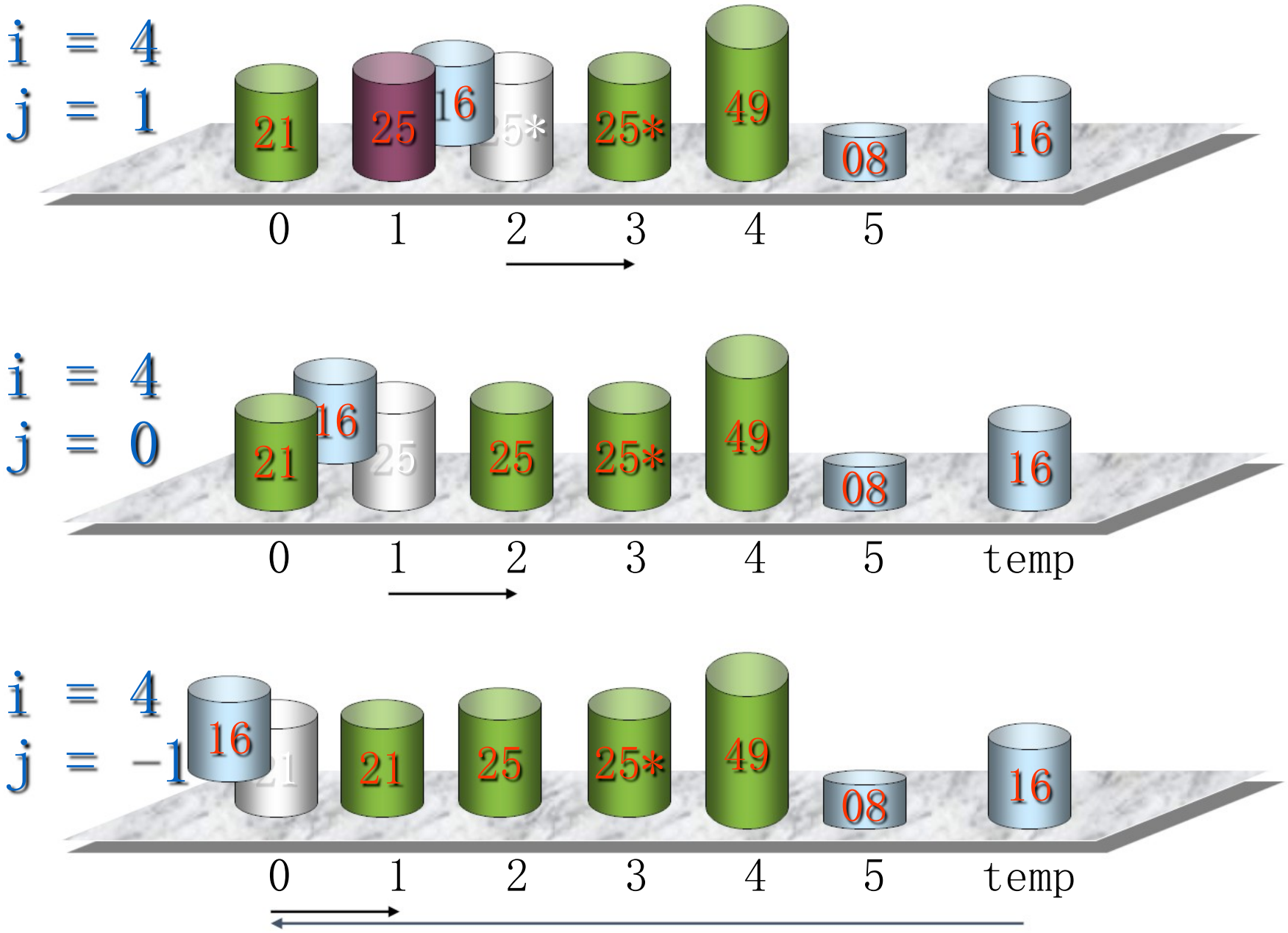
i = 4 时的排序过程

i = 4
j = 3



i = 4
j = 2




i = 4
j = 1
21
25
16
25*
25*
49
08
16
0
1
2
3
4
5
i = 4
j = 0
21
16
25
25
25*
49
08
16
0
1
2
3
4
5
temp
i = 4
j = -1
16
21
21
25
25*
49
08
16
0
1
2
3
4
5
temp

直接插入排序的代码实现

■ 算法10.1(p265)

```
void InsertSort(SqList &L){
    int i,j;
    for (i=2;i<=L.length;++i)
      if (LT(L.r[i].key,L.r[i-1].key)){
            L.r[0]=L.r[i];// L.r[0]为监视哨
            L.r[i]=L.r[i-1];
            for (j=i-2;LT(L.r[0].key, L.r[j].key);--j)
                  L.r[j+1]=L.r[j];
            L.r[j+1]=L.r[0];
      }
}
```

- 例如，原始序列49 38 65 97 76 13 27 49*，排序过程如下：

i=1: [49] 38 65 97 76 13 27 49*

i=2:(38) [38 49] 65 97 76 13 27 49*

i=3:(38) [38 49 65] 97 76 13 27 49*

i=4:(38) [38 49 65 97] 76 13 27 49*

i=5:(76) [38 49 65 76 97] 13 27 49*

i=6:(13) [13 38 49 65 76 97] 27 49*

i=7:(27) [13 27 38 49 65 76 97] 49*

i=8:(49) [13 27 38 49 49* 65 76 97]

监视哨**L.r[0]**

算法分析

- 设对象个数L.length=n, 则主程序执行n-1趟。
- 时间代价:
  - 关键字的比较次数和移动次数与对象关键字的初始排列有关。
  - 最好情况下：排序前对象已经按关键字有序，每趟只需与前面的有序对象序列的最后一个对象的关键字比较 1 次，总的关键字比较次数达到最小为 n-1，不需要移动元素，移动次数为0。
  - 最坏情况下：排序前对象完全逆序

    每趟比较i次,总比较次数: $\sum_{i=2}^{n} i = (n+2)(n-1)/2$

    移动i+1次,移动次数 $\sum_{i=2}^{n} (i+1) = (n+4)(n-1)/2$

•平均情况：若待排序对象序列中出现各种可能排列的概率相同，则可取上述最好情况和最坏情况的平均情况。

关键字比较次数: $C_{avg} = (n-1)(n+4)/4$

对象移动次数: $M_{avg} = (n+4)(n-1)/4$

比较次数和移动次数均约为： $n^2/4$

∴时间复杂度: $O(n^2)$。

■空间代价：O(1)

一个记录的辅助存储空间　监视哨

■直接插入排序是一种稳定的排序方法。

■不具有部分排序功能。

折半插入排序 (Binary Insertsort)

- 基本思想：

  设在顺序表中有一个对象序列V[0],V[1],…,v[n-1]。其中，v[0],V[1],…,v[i-1] 是已经排好序的对象。在插入 v[i] 时，利用折半搜索法寻找 v[i] 的插入位置。

■折半插入排序的算法

```
void BInsertSort(SqList &L){
   int low,high,mid;
     for (int i=2;i<=L.length;++i){
       L.r[0]=L.r[i];low = 1;high=i-1;
       while (low <= high){
          mid = (low+high)/2;
          if (LT(L.r[0].key,L.r[mid].key))    high=mid-1;
          else low=mid+1;}//end while
        for (int j=i-1;j>=high+1;--j)  L.r[j+1]=L.r[j];
        L.r[high+1]=L.r[0];
     }//end for
}
```

说明：low 或者 high+1为插入点

■稳定排序

算法分析

- 折半查找比顺序查找快，所以折半插入排序就平均性能来说比直接插入排序要快。
- 它所需要的关键字比较次数与待排序对象序列的初始排列无关，仅依赖于对象个数。在插入第 i 个对象时，需要经过 $\lfloor \log_2 i \rfloor + 1$ 次关键字比较，才能确定它应插入的位置。

因此，将n 个对象（为推导方便，设为$n=2^k$)用对分插入排序所进行的关键字比较次数为：$n\log_2 n$

$$\sum_{i=1}^{n-1}\left(\lfloor \log_2 i \rfloor + 1\right) = \underbrace{1}_{2^0} + \underbrace{2+2}_{2^1} + \underbrace{3+\cdots+3}_{2^2} +$$

$$+ \underbrace{4+\cdots+4}_{2^3} + \cdots + \underbrace{k+\cdots+k}_{2^{k-1}} =$$

$$= (1+2+2^2+\cdots+2^{k-1}) + (2+2^2+\cdots+2^{k-1}) +$$

$$+ (2^2+\cdots+2^{k-1}) + \cdots + 2^{k-1} =$$

$$= \sum_{i=1}^{k}\sum_{j=i}^{k} 2^{j-1} = \sum_{i=1}^{k} 2^{i-1}(1+2+\cdots+2^{k-i}) =$$

$$= \sum_{i=1}^{k} 2^{i-1}(2^{k-i+1}-1) = \sum_{i=1}^{k} 2^k - \sum_{i=1}^{k} 2^{i-1} =$$

$$= k\cdot 2^k - (2^k - 1) = n\log_2 n - n + 1 \approx n\log_2 n$$

- 当 n 较大时，总关键字比较次数比直接插入排序的最坏情况要好得多，但比其最好情况要差。

- 在对象的初始排列已经按关键字排好序或接近有序时，直接插入排序比折半插入排序执行的关键字比较次数要少。折半插入排序的对象移动次数与直接插入排序相同，依赖于对象的初始排列。

- 折半插入排序是一个稳定的排序方法。

时间复杂度下界

- 对于下标i<j，如果A[i] >A[j]，则称(i,j)是一对逆序对（inversion）。
- 问题：序列{34，8，64，51，32，21}中有多少逆序对?
- (34,8) (34,32) (34,21) (64,51) (64,32) (64,21)(51,32)(51,21) (32,21) 一共有9对
- 交换2个相邻元素正好消去一对逆序对!
- 插入排序：

  ---如果序列基本有序,则插入排序简单且高效

时间复杂度下界

- 定理：任意n个不同元素组成的序列，平均具有 $N(N-1)/4$ 个逆序对。
- 定理：任何仅以交换相邻两元素来排序的算法，其平均时间复杂度为$\Omega(n^2)$。
- 这意味着：要提高算法效率，我们必须：

  每次消去不止I个逆序对！

  每次交换相隔较远的2个元素！

希尔排序 (Shell Sort)

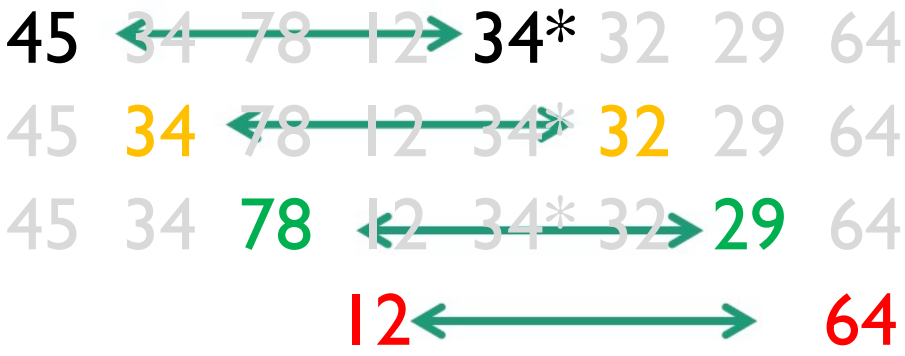
- 直接插入排序的两个性质：

  最好情况下(序列已有序)，时间代价为O(n)
  对于短序列，直接插入排序比较有效。

- 希尔排序方法又称为“缩小增量”排序。

- 希尔排序的基本思想是：
  - 先将整个待排对象序列按照一定间隔分割成为若干子序列，分别进行直接插入排序
  - 然后缩小间隔，对整个对象序列重复以上的划分子序列和分别排序工作，直到最后间隔为1。
  - 此时整个对象序列已“基本有序”，最后，进行一次直接插入排序。

- 45 34 78 12 34* 32 29 64
- gap=4



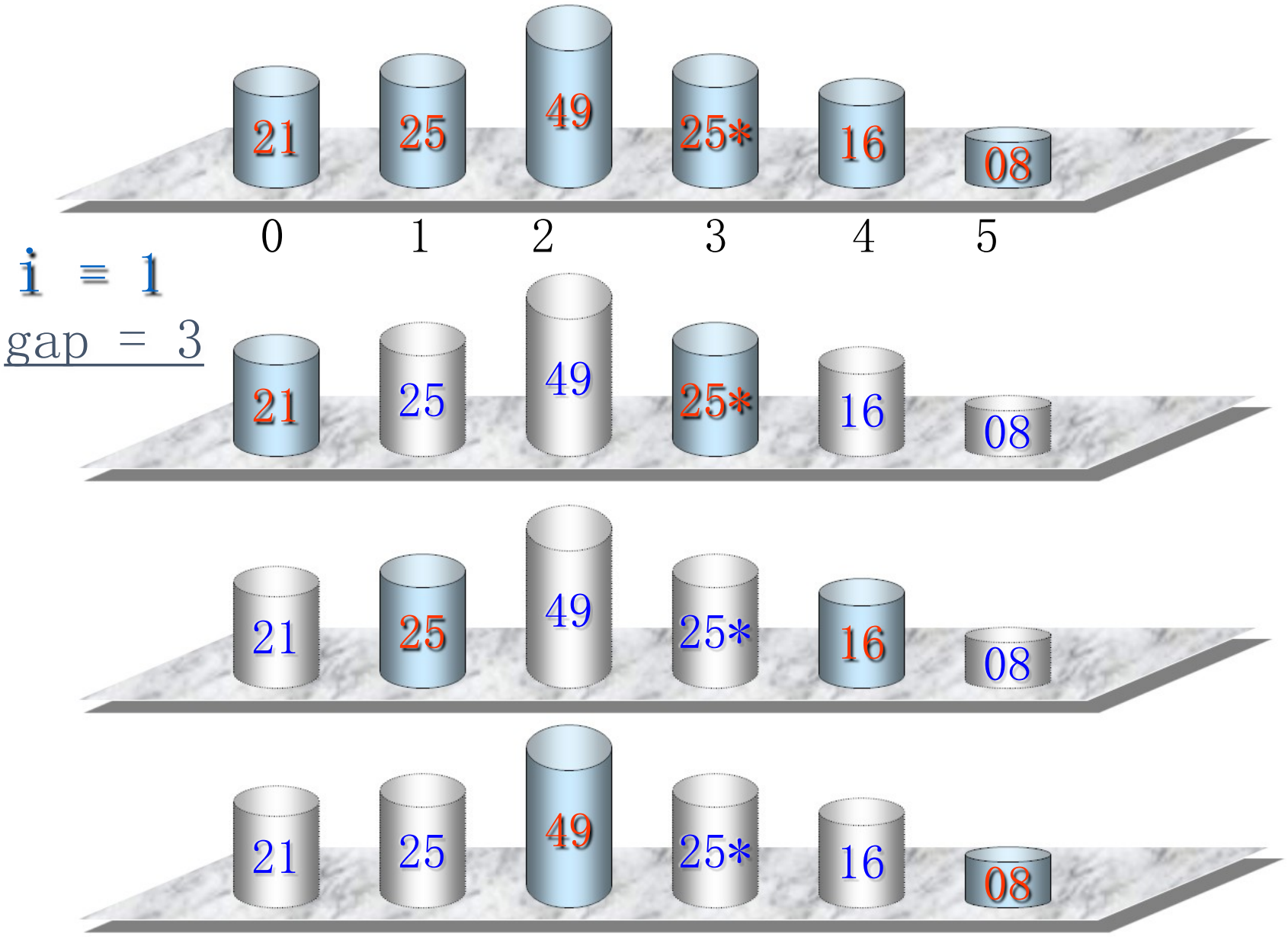
[34* 32 29 12 45 34 78 64] 一趟排序的结果

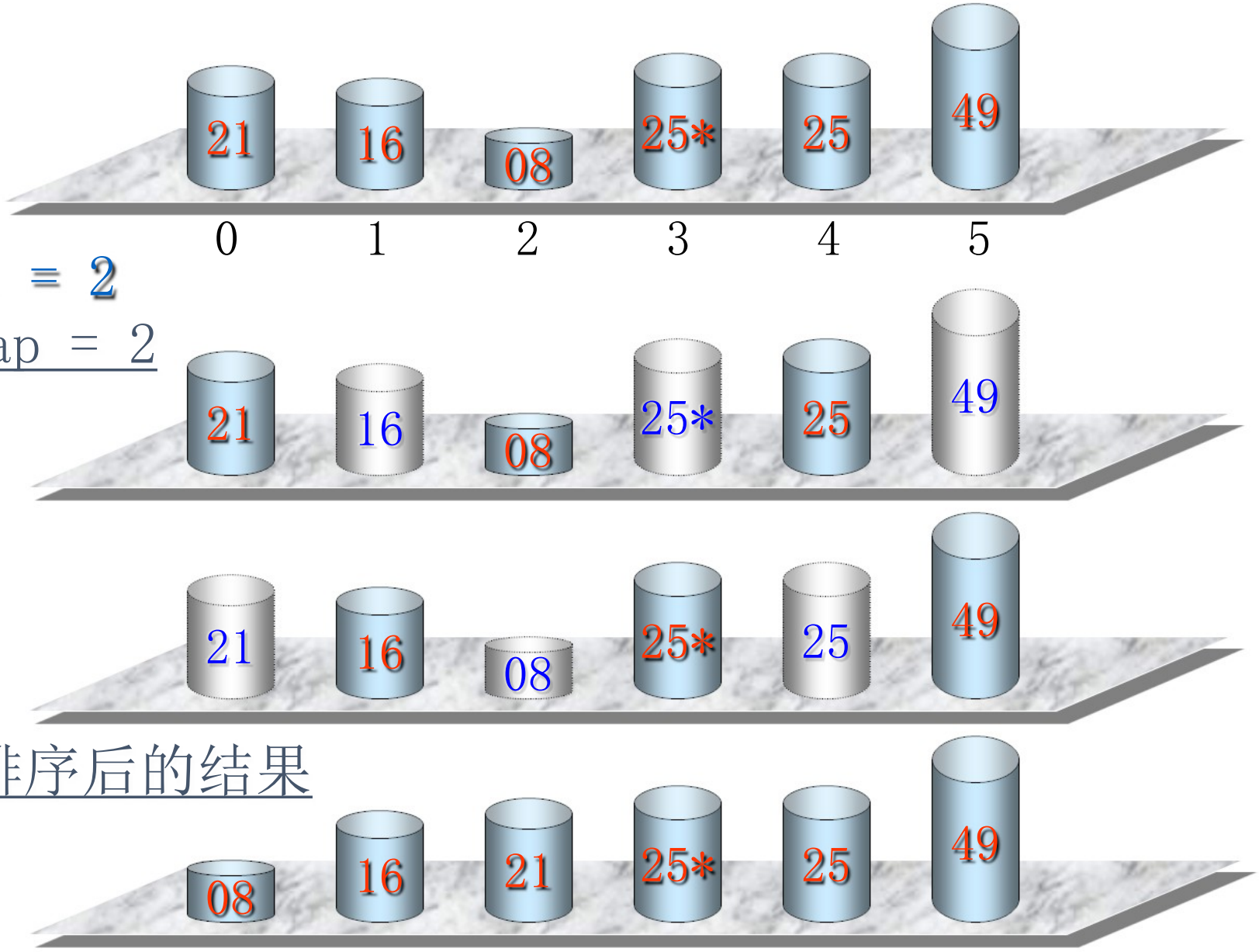
- gap=2

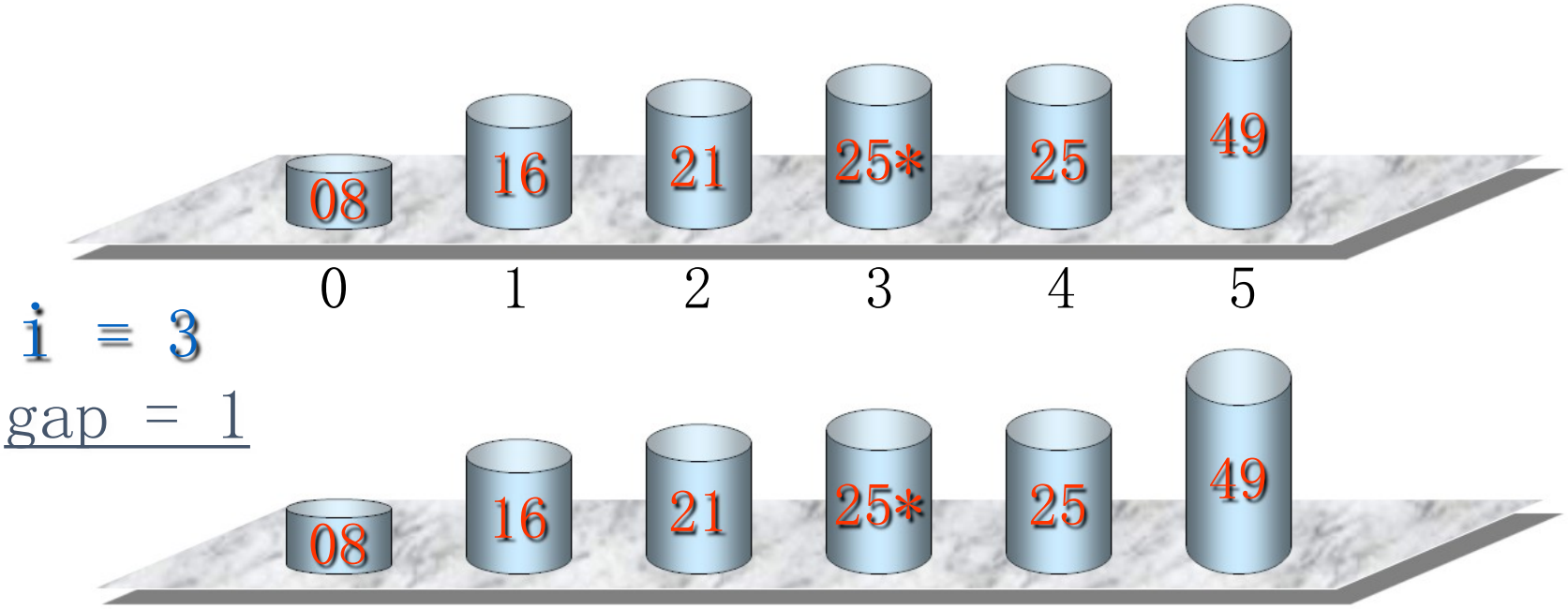
[29 12 34* 32 45 34 78 64] 二趟排序的结果

- gap=1

12 29 32 34* 34 45 64 78 三趟排序的结果


21
25
49
25*
16
08
0
1
2
3
4
5
i = 1
gap = 3
21
25
49
25*
16
08
21
25
49
25*
16
08
21
25
49
25*
16
08


21
16
08
25*
25
49
0
1
2
3
4
5
i = 2
gap = 2
21
16
08
25*
25
49
21
16
08
25*
25
49
排序后的结果
08
16
21
25*
25
49



- 开始时 gap（间隔值） 的值较大，子序列中的对象较少，排序速度较快；随着排序进展，gap 值逐渐变小，子序列中对象个数逐渐变多，由于前面工作的基础，大多数对象已基本有序，所以排序速度仍然很快。

■希尔排序的算法

```
void ShellSort(SqList &L,int dlta[],int t){
  for (int k=0;k<t;++k)
     { ShellInsert(L,dlta[k]);}
 }//说明：dlta[3]={5，3，1}
```

■//一趟希尔排序,按间隔dk划分子序列

```
void ShellInsert(SqList &L, int gap){
  for (int i=gap+1;i<=L.length;++i){
   if(LT(L.r[i].key,L.r[i-gap].key)){
     L.r[0]=L.r[i];int j=i-gap;
      for (;j>0 && LT(L.r[0].key,L.r[j].key);j-=gap)
         L.r[j+gap]=L.r[j]; //记录后移
      L.r[j+gap]=L.r[0];
     }  }  }
```

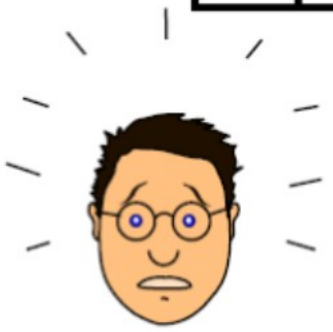
算法分析

- 对特定的待排序对象序列，可以准确地估算关键字的比较次数和对象移动次数。
- 但想要弄清关键字比较次数和对象移动次数与增量选择之间的依赖关系，并给出完整的数学分析，还没有人能够做到。
- gap的取法有多种。最初 shell 提出取 gap = ⌊n/2⌋，gap = ⌊ gap/2 ⌋，直到gap = 1。后来Knuth 提出取gap = ⌊ gap/3 ⌋+1。还有人提出都取奇数为好，也有人提出各gap互质为好。

其他取法：教材p272 倒数第二段

- Knuth利用大量的实验统计资料得出，当 n 很大时，关键字平均比较次数和对象平均移动次数大约在 n^1.25 到 1.6n^1.25 的范围内。这是在利用直接插入排序作为子序列排序方法的情况下得到的。
- 稳定性?

	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
8-间隔	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
4-间隔	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
2-间隔	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
1-间隔	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16



增量元素不互质，则小增量可能根本不起作用。

交换排序（Exchange Sort）

- 基本思想：

两两比较待排序对象的关键字，如果发生逆序(即排列顺序与排序后的次序正好相反)，则交换之，直到所有对象都排好序为止。

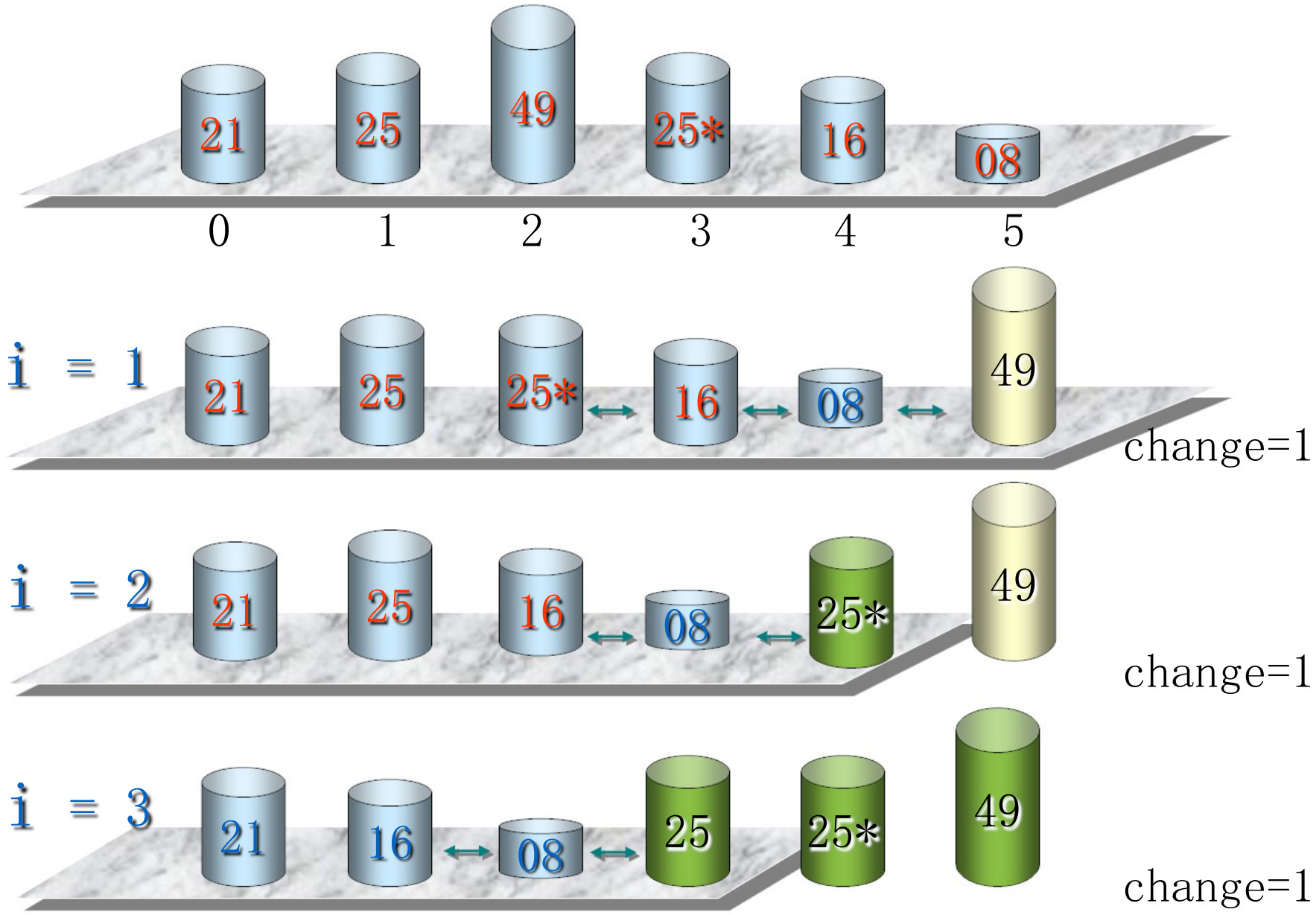
起泡排序（**Bubble Sort**）

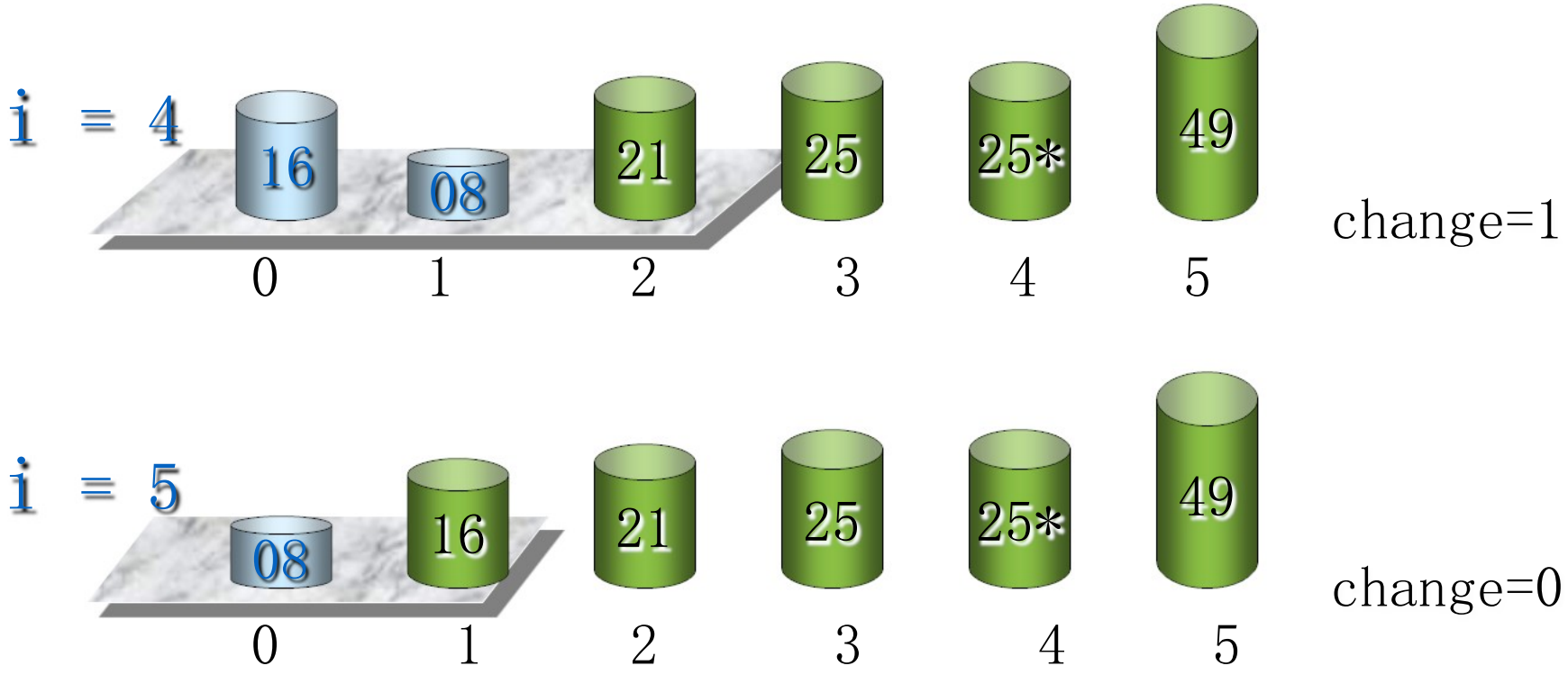
快速排序 **(Quick Sort)**

起泡排序 (Bubble Sort)

- 基本方法：

  设待排序对象序列中的对象个数为 n。最多做 n-1 趟排序。在第 i 趟中顺次两两比较r[j-1].Key和r[j].Key，j = i,i+1,…, n-i-1。如果发生逆序，则交换r[j-1]和r[j]。


21
25
49
25*
16
08
0
1
2
3
4
5
i = 1
21
25
25*
16
08
49
change=1
i = 2
21
25
16
08
25*
49
change=1
i = 3
21
16
08
25
25*
49
change=1


i = 4
16
08
21
25
25*
49
0
1
2
3
4
5
change=1
i = 5
08
16
21
25
25*
49
0
1
2
3
4
5
change=0

起泡排序代码

```
void BubbleSort(SqList &L){
 for(int i=L.length,change=TRUE;i>1 && change;--i){
   change = FALSE;
     for (int j=1;j<i;++j)
        if (LT(L.r[j+1].key,L.r[j].key)){
            ElemType temp=L.r[j];
                L.r[j]=L.r[j+1];
                L.r[j+1]=temp;
                change =TRUE;
              }
    }
}
```

- 例如：45 34 78 12 34* 32 29 64

  第一趟结果：[34 45 12 34* 32 29 64] 78　change=1

  第二趟结果：[34 12 34* 32 29 45] 64 78　change=1

  第三趟结果：[12 34 32 29 34*] 45 64 78　change=1

  第四趟结果：[12 32 29 34] 34* 45 64 78　change=1

  第五趟结果：[12 29 32] 34 34* 45 64 78　change=1

  第六趟结果：[12 29] 32 34 34* 45 64 78　change=0

算法特点

- 每　趟　至少比较1次，至多比较n-1次
- 一趟排序后就有一个元素的位置确定
- 可实现部分排序
- 稳定排序

算法分析

- 最好的情形：初始排列已经按关键字从小到大排好序，此时算法只执行一趟起泡，做 n-1 次关键字比较，不移动对象。
- 最坏的情形：算法执行了n-1趟起泡，第 i 趟 (1<= i<n) 做了 n- i 次关键字比较，执行了n-i 次对象交换。总的关键字比较次数KCN和对象移动次数RMN为：

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

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

- 空间代价:需要一个附加对象以实现对象值的对换。

快速排序 (Quick Sort)

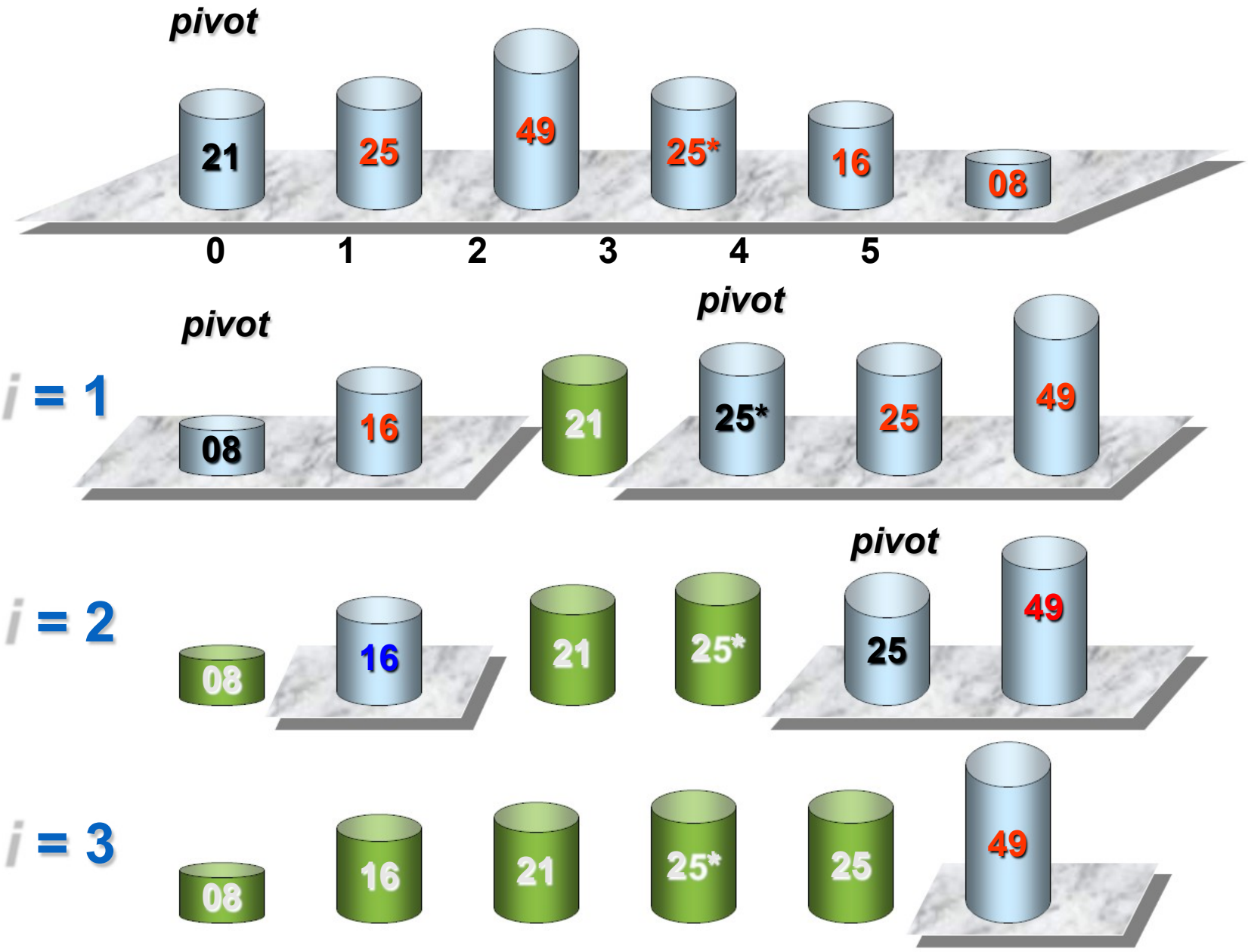
- 基本思想：任取待排序对象序列中的某个对象 (例如取第一个对象) 作为枢轴(pivot)或支点，按照该对象的关键字大小，将整个对象序列划分为左右两个子序列:
  - 左侧子序列中所有对象的关键字都小于或等于枢轴对象的关键字
  - 右侧子序列中所有对象的关键字都大于枢轴对象的关键字
  - 枢轴对象则排在这两个子序列中间(这也是该对象最终应安放的位置)。

  然后分别对这两个子序列重复施行上述方法，直到所有的对象都排在相应位置上为止。

- 快速排序是20世纪十大算法（top 10 algorithm of the century）
- 1962/7 London的Elliot Brothers Ltd的Tony Hoare提出
- 是一种基于分治思想的算法
- 分治思想：
  - 划分成独立的子问题
  - 求解子问题（子问题不重叠）
  - 合并子问题
- 例如：BST查找、插入、删除；二分检索；快速排序；归并排序

- 算法描述

```
QuickSort ( List ) {
  if ( List的长度大于1) {
    将序列List划分为两个子序列 LeftList 和 Right List;
    QuickSort ( LeftList );
    QuickSort ( RightList );
    将两个子序列 LeftList和RightList合并为一个序列List;
    }
}
```


pivot
21
25
49
25*
16
08
0
1
2
3
4
5
pivot
pivot
i = 1
08
16
21
25*
25
49
pivot
i = 2
08
16
21
25*
25
49
i = 3
08
16
21
25*
25
49

```
int Partition (Type a[], int p, int r){
    int i = p, j = r + 1;
    Type x=a[p];
    // 将< x的元素交换到左边区域
    // 将> x的元素交换到右边区域
    while (true) {
      while (a[++i] <x);
      while (a[- -j] >x);
      if (i >= j) break;
      Swap(a[i], a[j]);
     }
    a[p] = a[j];
    a[j] = x;
    return j;
    }
```

- 快速排序的递归代码

```
void QSort(SqList &L, int low, int high){
  if (low < high){
   int pivotloc = Partition(L,low,high);
     QSort(L,low, pivotloc-1);
    // PrintST(L);
     QSort(L,pivotloc+1, high);
    // PrintST(L);
    }
}
```

- 对所有元素进行快速排序

```
void QuickSort(SqList &L){
    QSort(L,1,L.length);}
```

- 例如：45 34 78 12 34* 32 29 64

  一趟划分后,划分成2个子序列：

  [29 34 32 12 34*] **45** [78 64]

  每个子序列继续划分为：

  [12] **29** [32 34 34*] 45 [78 64]

  继续划分，结果如下：

  12 29 [32 34 34*] 45 [78 64]

  12 29 **32** [34 34*] 45 [78 64]

  12 29 32 **34** [34*] 45 [78 64]

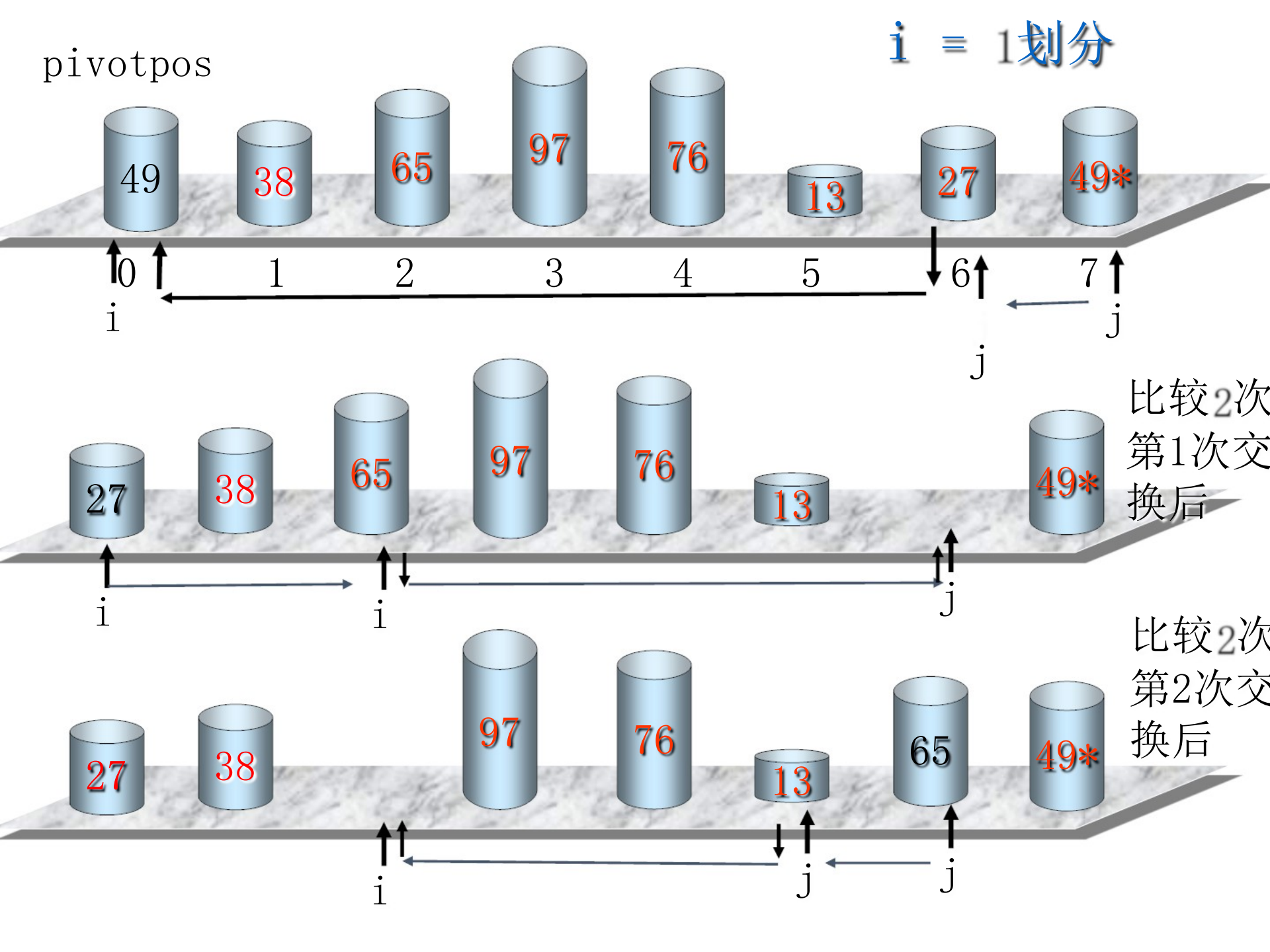
  12 29 32 34 **34*** 45 [78 64]

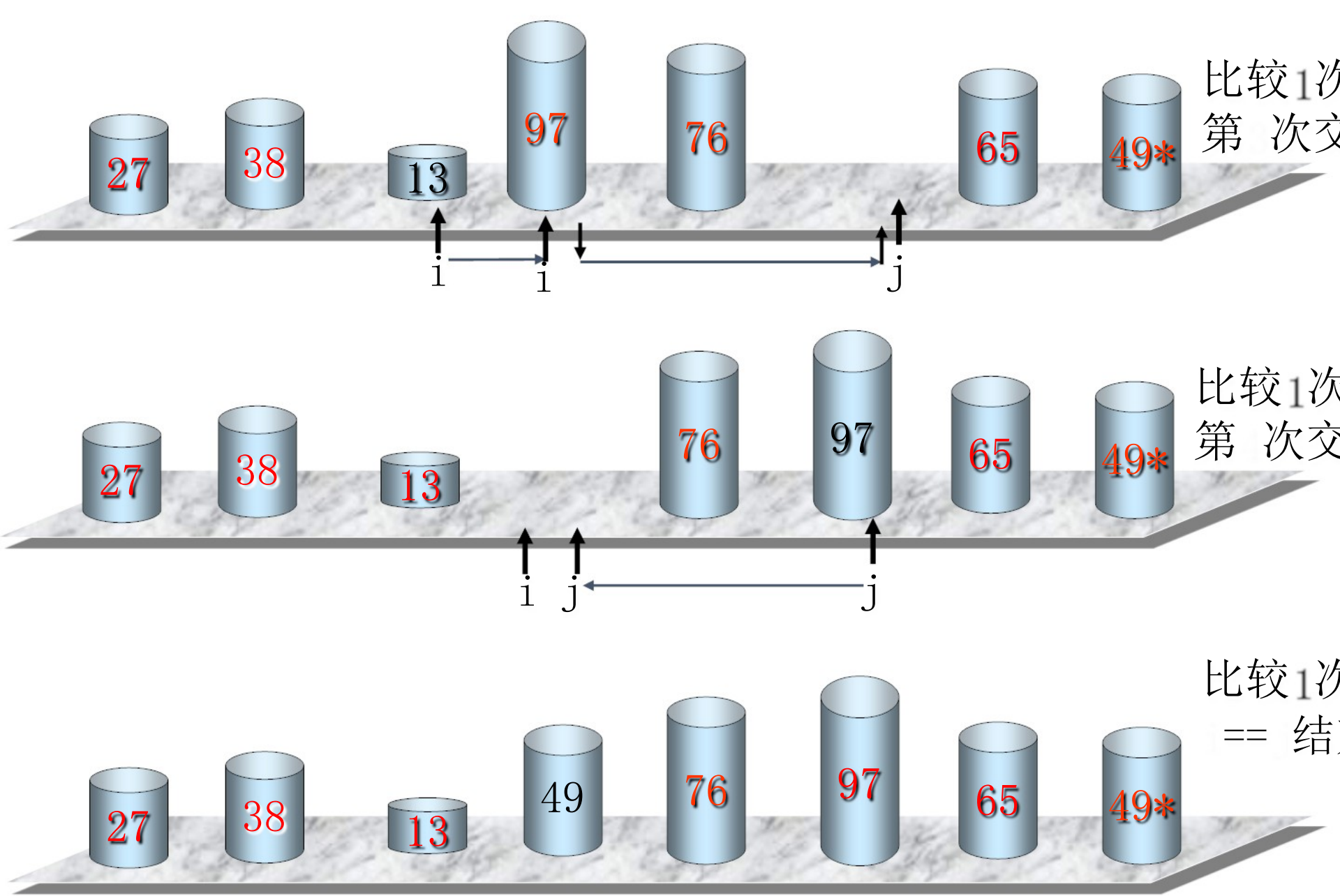
  12 29 32 34 34* 45 [64] **78**

  12 29 32 34 34* 45 **64** 78

- 算法partition的思想

  利用序列第一个对象作为枢轴，将整个序列划分为左右两个子序列。算法中执行了一个循环，只要是关键字小于枢轴对象关键字的对象都移到序列左侧，最后枢轴对象安放到位，函数返回其位置。


i = 1划分
pivotpos
49
38
65
97
76
13
27
49*
0
1
2
3
4
5
6
7
i
j
j
比较2次
第1次交换后
27
38
65
97
76
13
49*
i
i
j
比较2次
第2次交换后
27
38
97
76
13
65
49*
i
j
j


比较1次
第 次交
27
38
13
97
76
65
49*
i
i
j
比较1次
第 次交
27
38
13
76
97
65
49*
i
j
j
比较1次
== 结
27
38
13
49
76
97
65
49*

完成一趟排序

■一次快速划分的代码

```
int Partition(SqList &L,int low,int high){
    L.r[0] = L.r[low];//下标0存放枢轴元素
    int 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;
}
```

■举例：(一次划分的过程)

初始关键字序列为： 45 34 78 12 34* 32 29 64

pivotkey=45,

初始　45 34 78 12 34* 32 29 64

45 34 78 12 34* 32 29 64

第1次交换： 29 34 78 12 34* 32 64

第2次交换： 29 34 12 34* 32 78 64

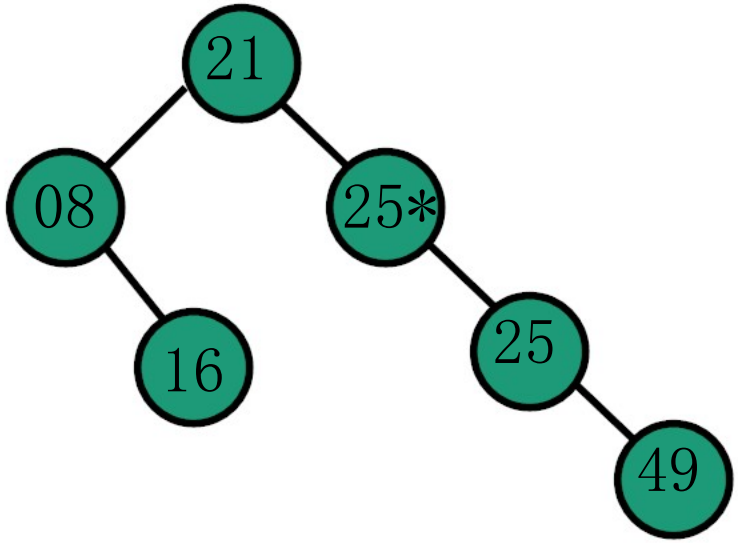
第3次交换：29 34 32 12 34* 78 64

35就位： 29 34 32 12 34* 45 78 64

（完成一趟划分）

算法分析

- 算法quicksort是一个递归的算法，其递归树如图所示。



- 从递归树可知，快速排序的趟数取决于递归树的深度。

算法分析

- 理想的情况：

  如果每次划分对一个对象定位后，该对象的左侧子序列与右侧子序列的长度相同，则下一步将是对两个长度减半的子序列进行排序。

  在n个元素的序列中，对一个对象定位所需时间为O(n)。若设 t(n) 是排序所需的时间，而且每次对一个对象正确定位后，正好把序列划分为长度相等的两个子序列，此时，总的计算时间为：

$$T(n) \le cn + 2\,t(n/2) \quad //c\text{是一个常数}$$
$$\le Cn + 2\,(\,cn/2 + 2t(n/4)\,) = 2cn + 4t(n/4)$$
$$\le 2cn + 4\,(\,cn/4 + 2t(n/8)\,) = 3cn + 8t(n/8)$$
$$\cdots\cdots\cdots$$
$$\le Cn\log_2 n + nt(1) = O(n\log_2 n)$$

快速排序是递归的，需要有一个栈存放每层递归调用时的指针和参数。

最大递归调用层次数与递归树的深度一致，理想情况为$\lceil log_2\ (n+1) \rceil$。因此，要求存储开销为$O(log_2 n)$。

- 可以证明，函数quicksort的平均计算时间也是$O(nlogn)$。
- 实验结果表明：就平均计算时间而言，快速排序是我们所讨论的所有内排序方法中最好的一个。

■最坏情况

待排序对象序列已经按其关键字从小到大排好序的情况下，其递归树成为单支树，每次划分只得到一个比上一次少一个对象的子序列。这样，必须经过 n-l 趟才能把所有对象定位，而且第 i 趟需要经过 n-i 次关键字比较才能找到第 i 个对象的安放位置，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)。

- 算法的改进：

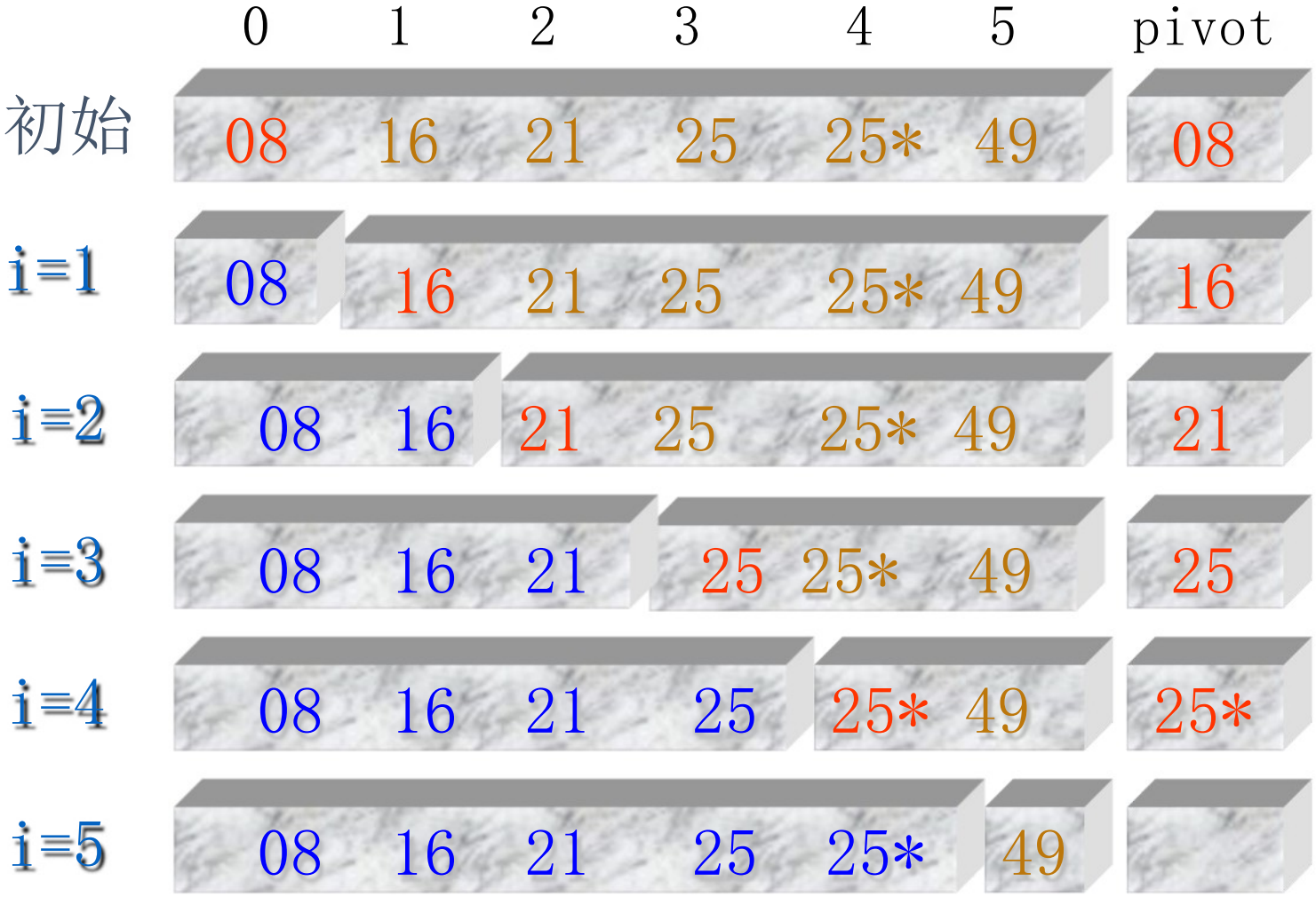
  若能更合理地选择基准对象，使得每次划分所得的两个子序列中的对象个数尽可能地接近，可以加速排序速度，但是由于对象的初始排列次序是随机的，这个要求很难办到。

  有一种改进办法：取每个待排序对象序列的第一个对象、最后一个对象和位置接近正中的3个对象，取其关键字大小居中者作为基准对象。(只要将该记录与L.r[low]互换,算法10.6(b)不变。)

- 稳定性

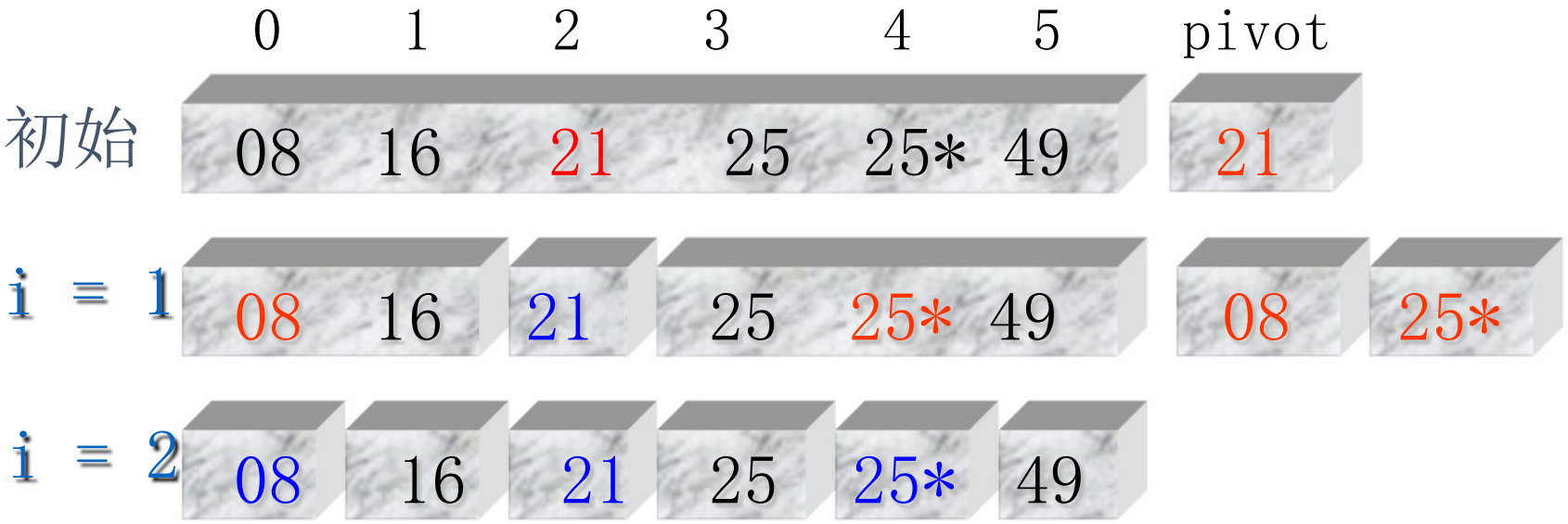
  快速排序是一种不稳定的排序方法。

- 对于 n 较大的平均情况而言，快速排序是“快速”的，但是当 n 很小时，这种排序方法往往比其它简单排序方法还要慢。



用第一个对象作为基准对象

快速排序退化的例子



用居中关键字对象作为基准对象

快速排序时间代价分析

- 长度为n的序列，时间为$T(n)$

$$T(0) = T(1) = 1$$

- 选择轴值时间为常数

  一次划分的时间为$\boldsymbol{cn}$

  分割后的长度分别为 i 和$n - i - 1$

  左右子序列排序时间为$T(i)$和$T(n - i - 1)$

- 求解递推方程：$T(n) = cn + T(i) + T(n - i - 1)$

最差情况

- 总的时间代价：

$$T(n) = T(n-1) + cn$$
$$T(n-1) = T(n-2) + c(n\text{-}1)$$
$$T(n-2) = T(n-3) + c(n\text{-}2)$$
$$\cdots$$
$$T(2) = T(1) + c(2)$$

所以，

$$T(n) = T(1) + c\sum_{k=0}^{n} i$$
$$= O(n^2)$$

最好情况

- 总的时间代价为

$$T(\boldsymbol{n}) = 2T(\boldsymbol{n}/2) + \boldsymbol{cn}$$

$$T(\boldsymbol{n})/n = T\left(\frac{\boldsymbol{n}}{2}\right)/\left(\frac{n}{2}\right) + \boldsymbol{c}$$

$$\frac{T\left(\frac{\boldsymbol{n}}{2}\right)}{n/2} = T\left(\frac{n}{4}\right)/\left(\frac{n}{4}\right) + \boldsymbol{c}$$

$$\cdots$$

$$T(\boldsymbol{2}/\boldsymbol{2}) = T(\boldsymbol{1}/\boldsymbol{1}) + \boldsymbol{c}$$

$$T(\boldsymbol{n})/\boldsymbol{n} = T(\boldsymbol{1})/\boldsymbol{1} + \boldsymbol{clogn}$$

所以，

$$T(\boldsymbol{n}) = n + \boldsymbol{cnlogn} = O(nlogn)$$

等概率情况

- 轴值将数组分成长度为 0 和 $n-1, 1$ 和 $n-2$，... 的子序列的概率是相等的，都为 $1/n$
- T(i)和T(n-i-l)的平均值均为

$$T(\mathrm{i}) = T(n - \mathrm{i} - 1)$$

$$= 1/\mathrm{n} \sum_{k=0}^{n-1} T(k)$$

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

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

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

- 总的时间代价为：$T(n) = O(nlogn)$

选择排序(Selection Sort)

- 基本思想：每一趟 (例如第 i 趟，i = 1,…,n-1) 在后面的 n-i+1 个待排序对象中选出关键字最小的对象,作为有序对象序列的第 i 个对象。待到第 n-1 趟作完，待排序对象只剩下1个，就不用再选了。

  简单选择排序

  锦标赛排序（树形选择排序）

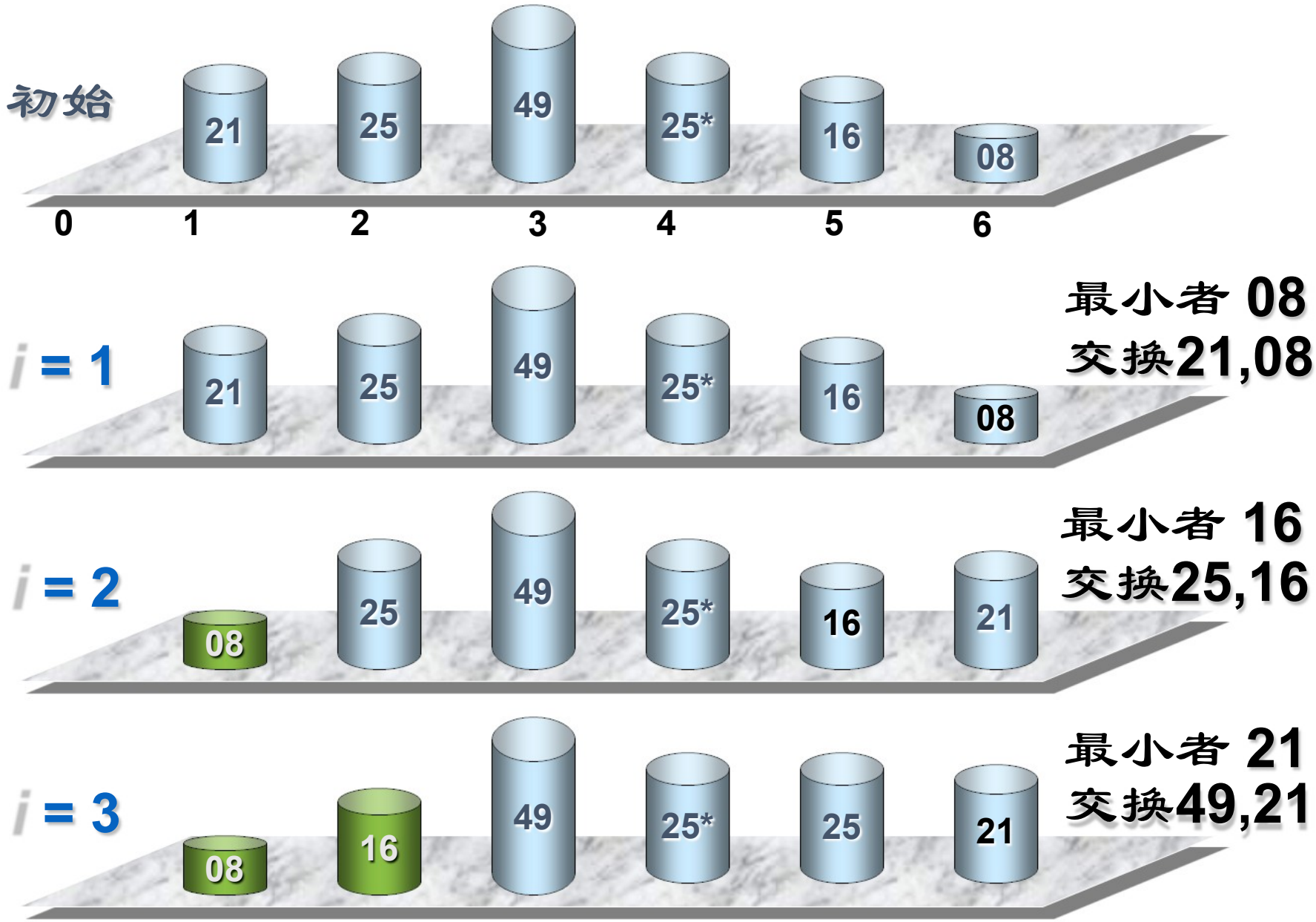
  堆排序(最大堆)

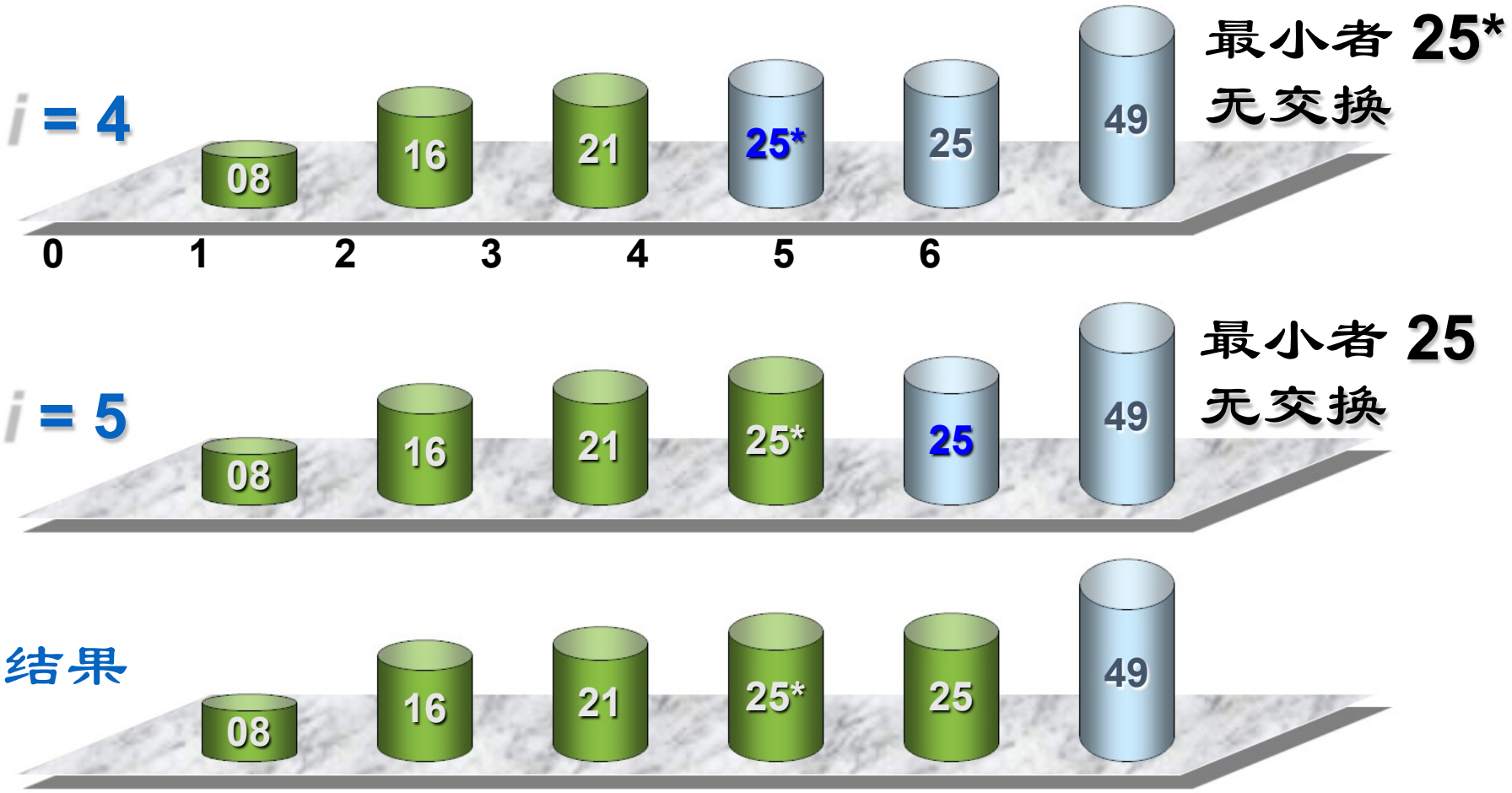
简单选择排序

- 基本步骤：

  i从1开始，直到n-1，进行n-1趟排序；

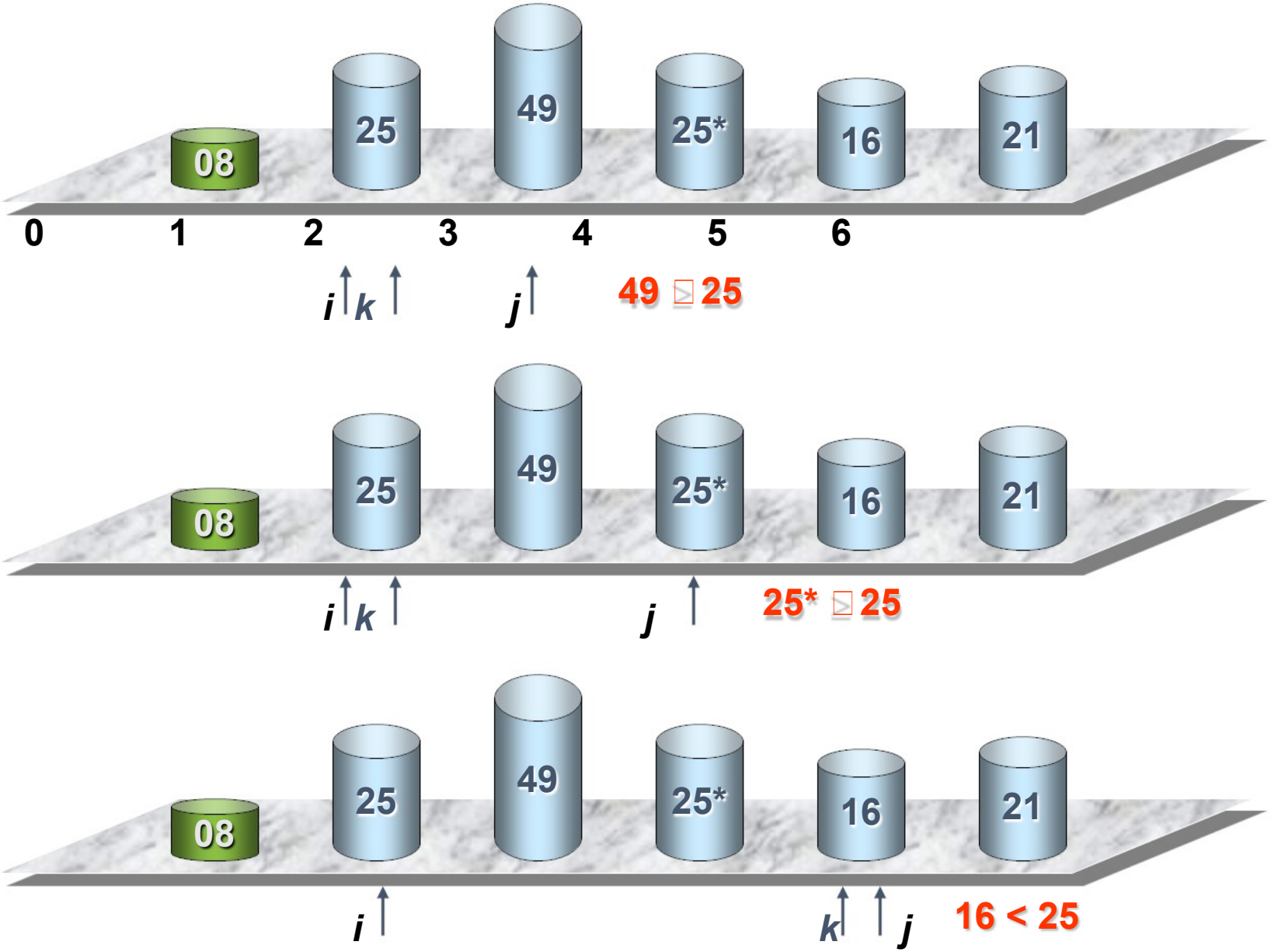
  第i 趟的排序过程为： 在一组对象r[i]～r[n] (i=1,2,…,n-1)中选择具有最小关键字的对象；并和第 i 个对象进行交换；

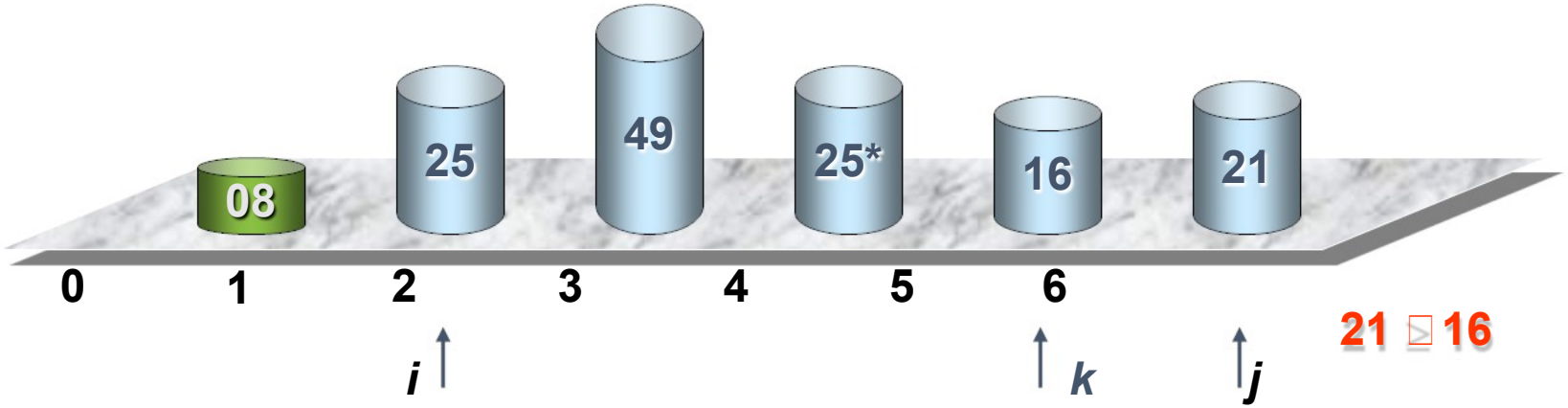

初始
21
25
49
25*
16
08
0
1
2
3
4
5
6
i = 1
最小者 08
交換21,08
i = 2
最小者 16
交換25,16
i = 3
最小者 21
交換49,21


i = 4
08
16
21
25*
25
49
最小者 25*
无交换
0
1
2
3
4
5
6
i = 5
08
16
21
25*
25
49
最小者 25
无交换
结果
08
16
21
25*
25
49

各趟排序后的结果

i =2时选择排序的过程


08
25
49
25*
16
21
0
1
2
3
4
5
6
i
k
j
49 ≥ 25
08
25
49
25*
16
21
i
k
j
25* ≥ 25
08
25
49
25*
16
21
i
k
j
16 < 25



k 指示当前序列中最小者

- 简单选择排序的算法

```
void SelectSort(SqList &L){
  for (int i=1;i<L.length;++i){
    int k=i;
      for (int j=i+1;j<=L.length;++j)
            if (L.r[k].key > L.r[j].key) k=j;
      if (i!=k){
            ElemType temp=L.r[i];
            L.r[i]=L.r[k];
            L.r[k]=temp;
      }
    }
}
```

- 例如：写出选择排序每一趟的排序结果，其关键字序列为：45 34 78 12 34* 32 29 64

第1趟： [12] 34 78 45 34* 32 29 64
第2趟： [12 29] 78 45 34* 32 34 64
第3趟： [12 29 32] 45 34* 78 34 64
第4趟： [12 29 32 34*] 45 78 34 64
第5趟： [12 29 32 34* 34] 78 45 64
第6趟： [12 29 32 34* 34 45] 78 64
第7趟： [12 29 32 34* 34 45 64] 78

算法分析

- 直接选择排序的关键字比较次数KCN与对象的初始排列无关。

  第 i 趟选择具有最小关键字对象所需的比较次数总是 n-i 次，此处假定整个待排序对象序列有 n 个对象。因此，总的关键字比较次数为

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

- 对象的移动次数与对象序列的初始排列有关。

  当这组对象的初始状态是按其关键字从小到大有序的时候，对象的移动次数RMN = 0，达到最少。

- 最坏情况是每一趟都要进行交换，总的对象移动次数为RMN = 3(n-1)。

- 直接选择排序是一种不稳定的排序方法。

锦标赛排序(Tournament Tree Sort)
树形选择排序(Tree Selection Sort)

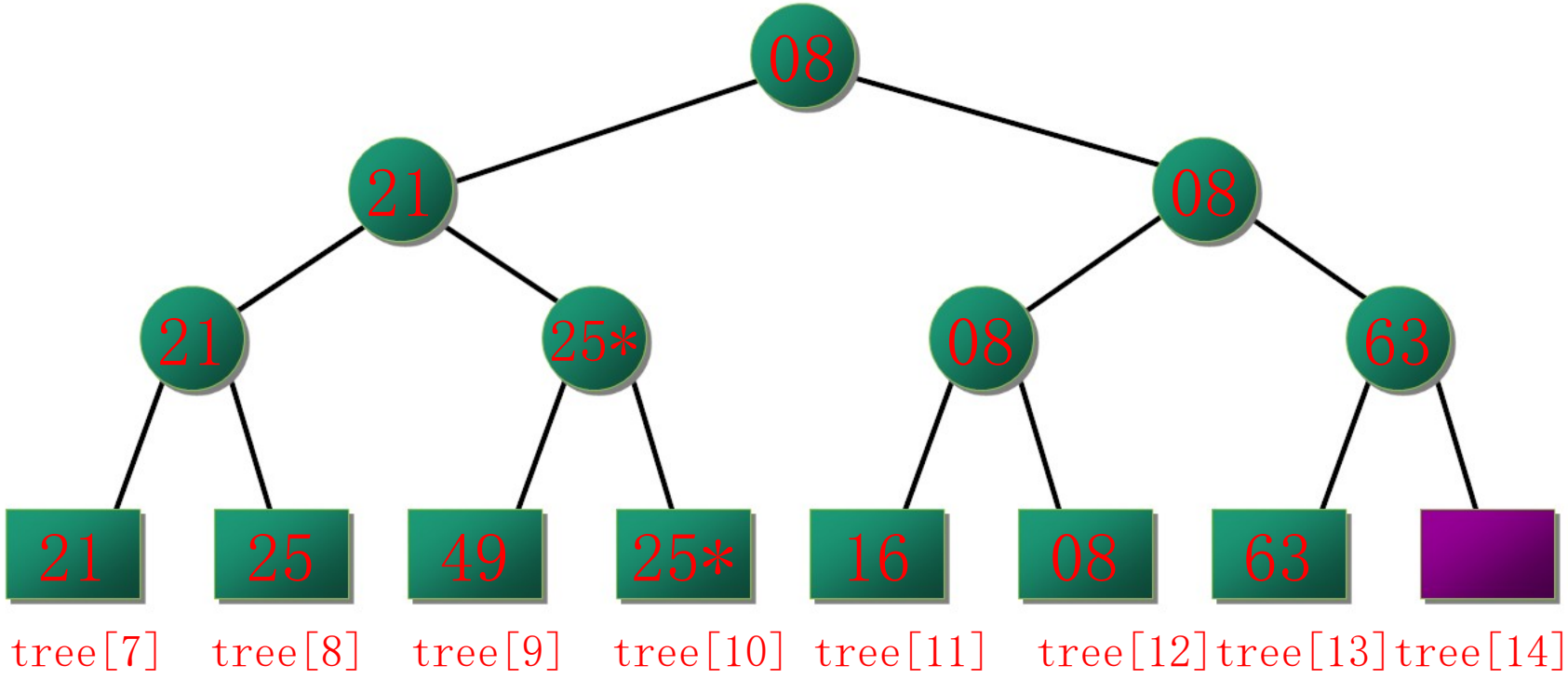
- 算法思想：与体育比赛时的淘汰赛类似。

  首先取得 n 个对象的关键字，进行两两比较，得到⌈n/2⌉个比较的优胜者(关键字小者)，作为第一步比较的结果保留下来。然后对这⌈n/2⌉个对象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对象为止。

- 排序过程用一棵满二叉树表示。

  在下列图例中，最下面是对象排列的初始状态，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对象的关键字。

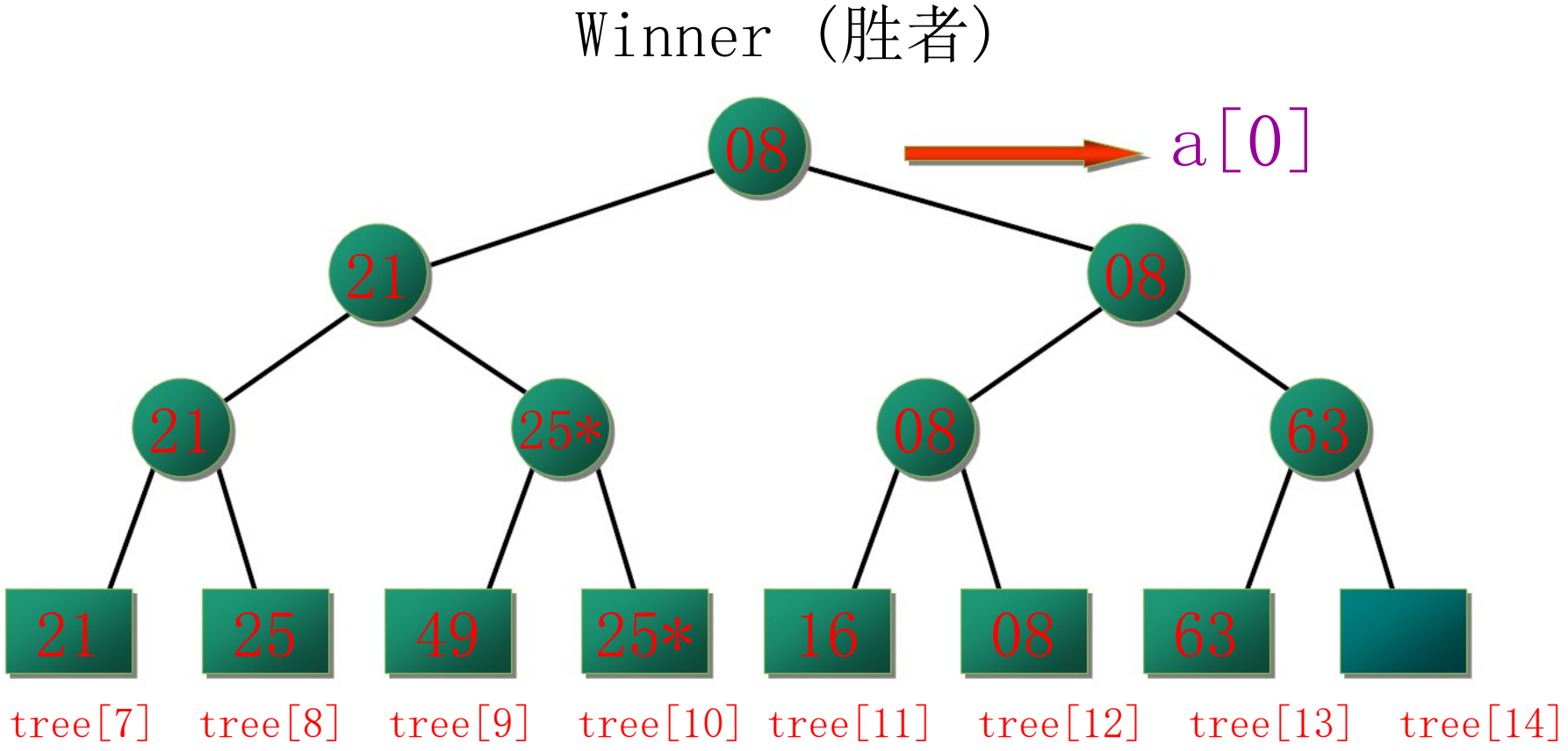
Winner



如果 n 不是2的 k 次幂，则让叶结点数补足到满足 2^(k-l) < n <= 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。最顶层是树的根。

胜者树的概念

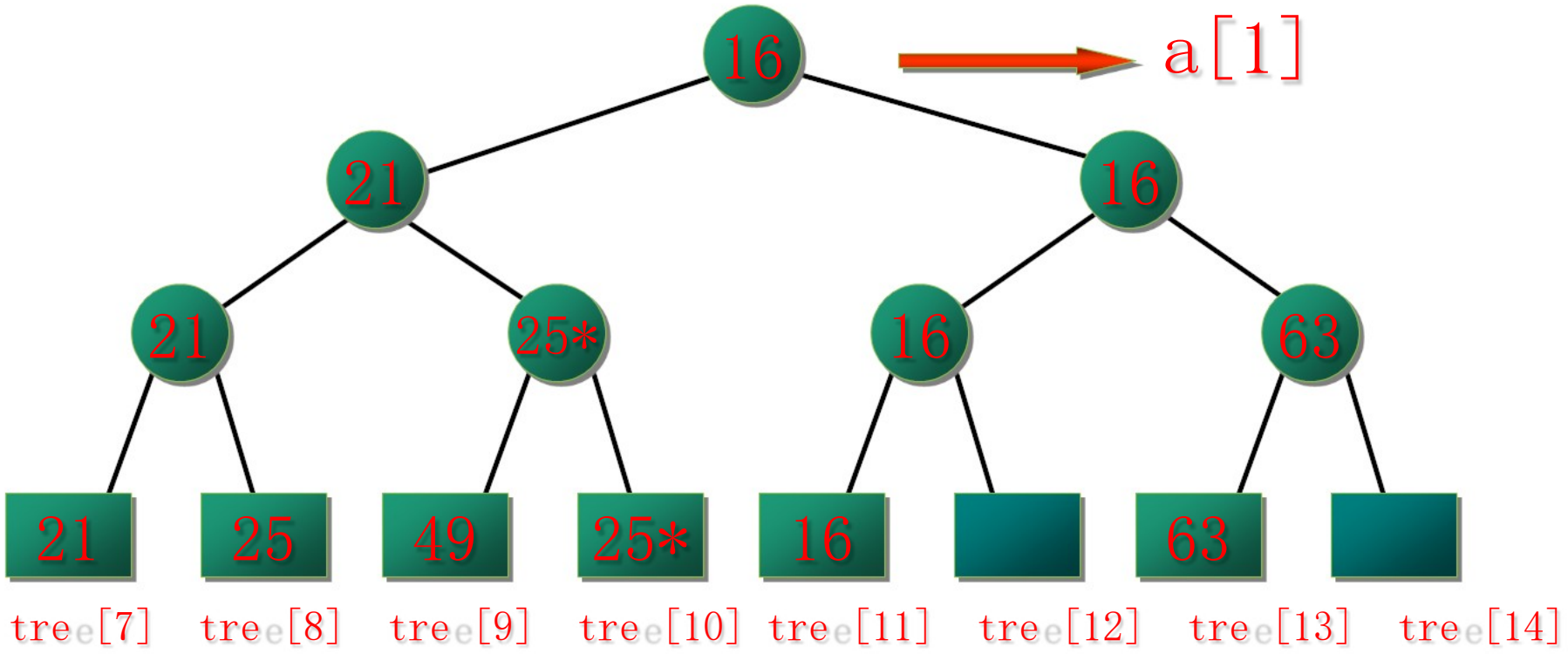
- 每次两两比较的结果是把关键字小者作为优胜者上升到双亲结点，称这种比赛树为胜者树。
- 位于最底层的叶结点叫做胜者树的外结点，非叶结点称为胜者树的内结点。每个结点除了存放对象的关键字 key 外，还存放了此对象是否要参选的标志 Active 和此对象在满二叉树中的序号index。
- 胜者树最顶层是树的根，表示最后选择出来的具有最小关键字的对象。



形成初始胜者树（最小关键字上升到根）

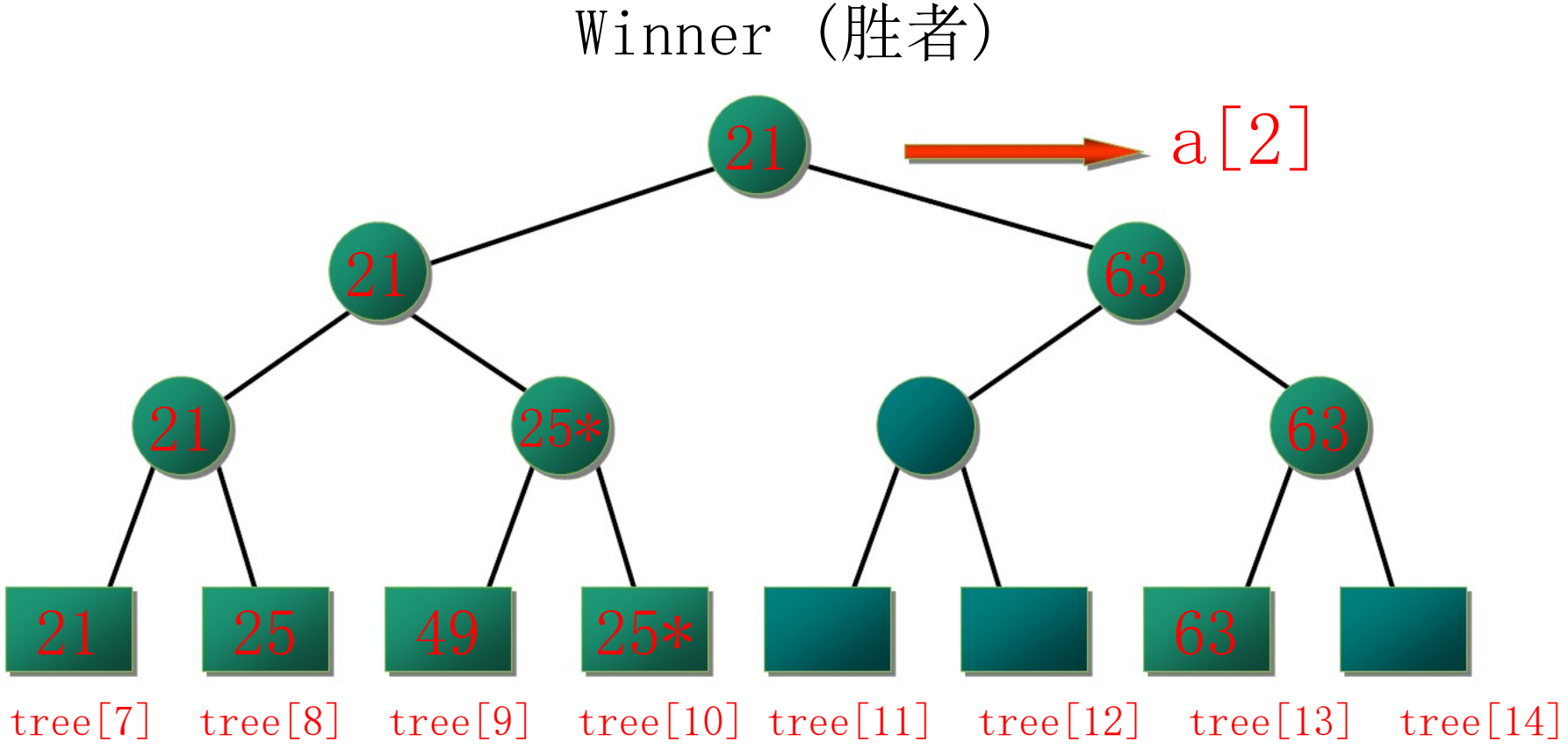
关键字比较次数 ： 6

Winner （胜者）



输出冠军并调整胜者树后树的状态

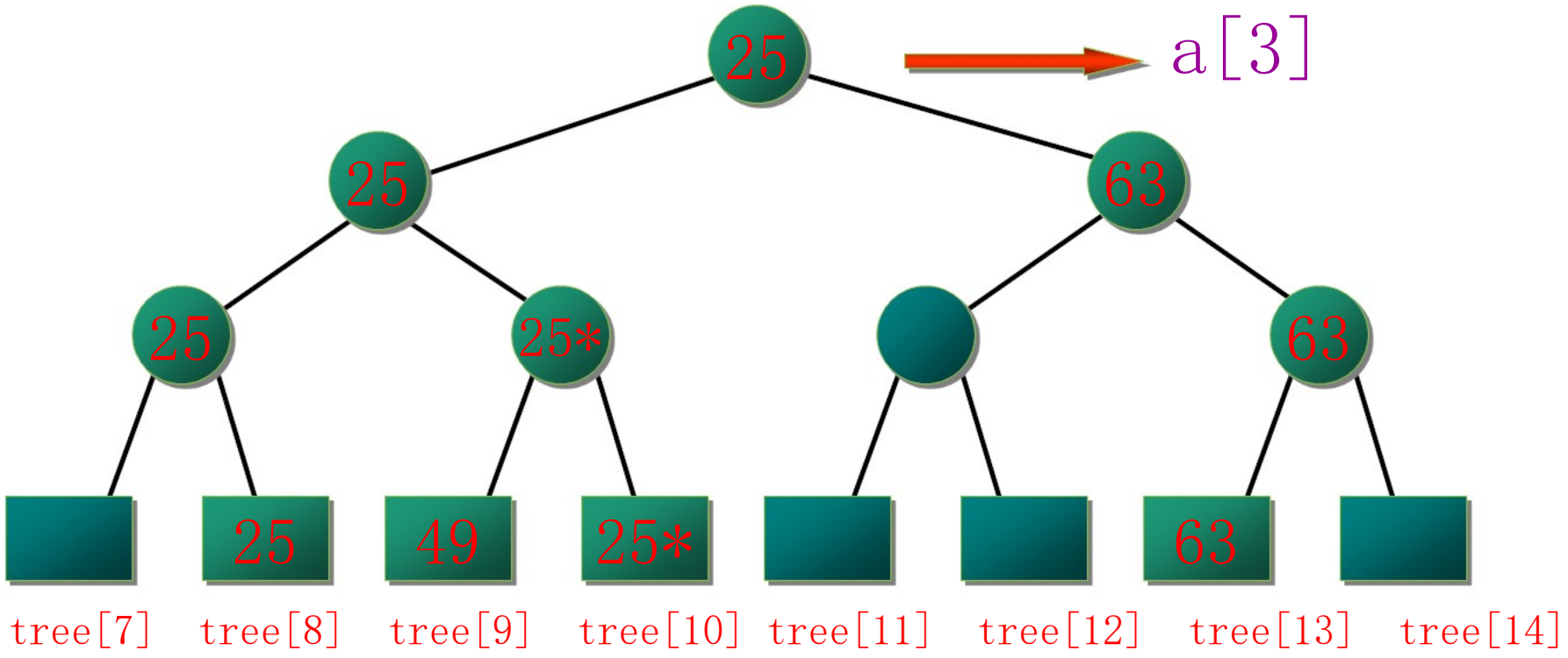
关键字比较次数 ： 2



输出亚军并调整胜者树后树的状态

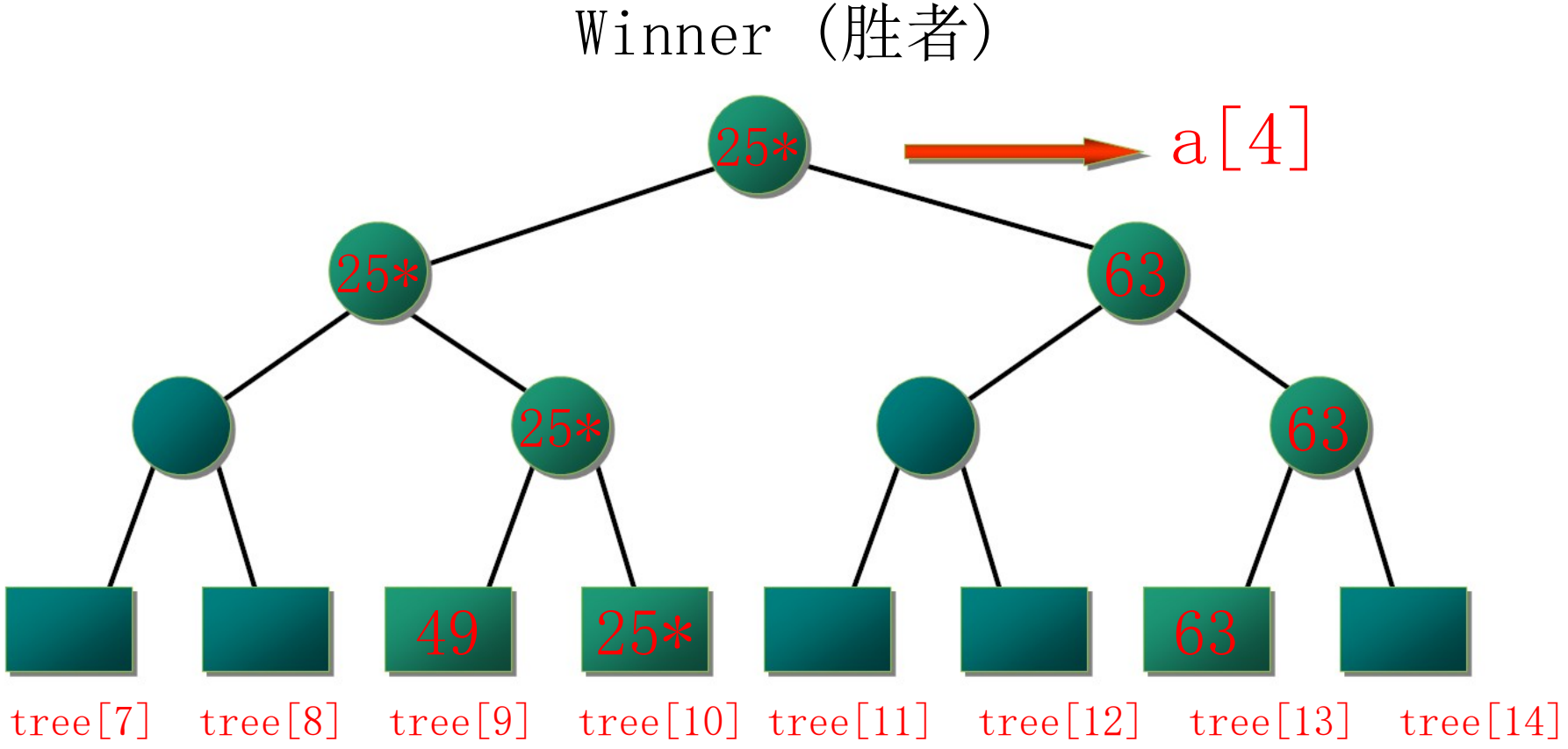
关键字比较次数 ： 2

Winner（胜者）



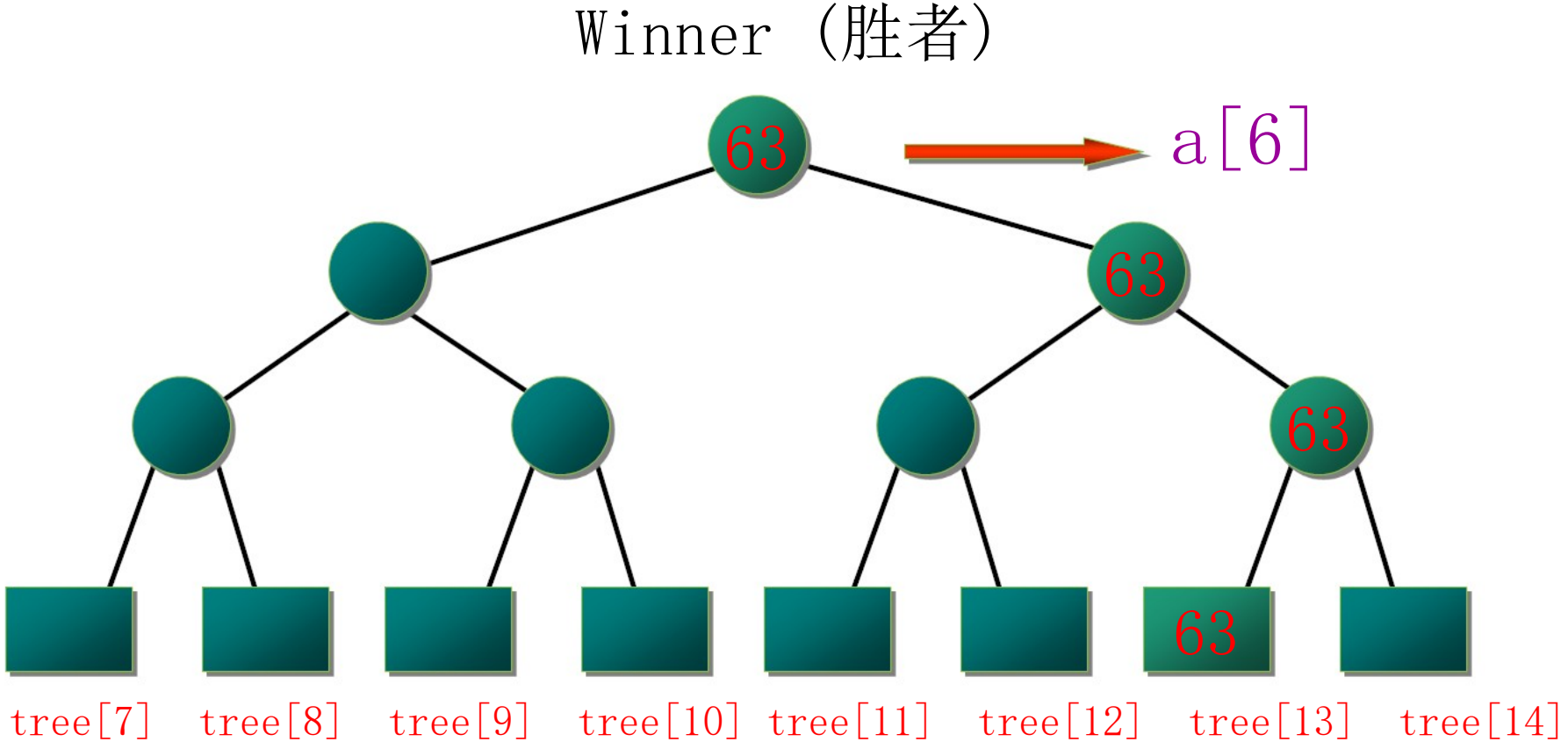
输出第三名并调整胜者树后树的状态

关键字比较次数 ： 2



输出第四名并调整胜者树后树的状态

关键字比较次数 ： 2



全部比赛结果输出时树的状态

关键字比较次数 ： 2

算法分析

- 锦标赛排序构成的树是满的完全二叉树，其深度为 $\lceil \log_2(n+1) \rceil$，其中 n 为待排序元素个数。
- 除第一次选择具有最小关键字的对象需要进行 n-1 次关键字比较外，重构胜者树选择具有次小、再次小关键字对象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 $O(n \log_2 n)$。
- 对象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许多排序时间，但是使用了较多的附加存储。

- 如果有 n 个对象，必须使用至少 2n-l 个结点来存放胜者树。最多需要找到满足 $2^{k-1} < n <= 2^k$ 的k，使用 $2^* 2^k$ -l 个结点。每个结点包括关键字、对象序号和比较标志三种信息。
- 锦标赛排序是一个稳定的排序方法。

堆排序(Heap Sort)

- **堆的定义**：

n个元素的序列$\{K_1, K_2 \cdots K_n\}$，当且仅当满足如下关系(1)或者(2)时，称作**堆**。

(1) $K_i \leq K_{2i}$ && $K_i \leq K_{2i+1}$ (i>0) 或

(2) $K_i \geq K_{2i}$ && $K_i \geq K_{2i+1}$ (i>0)

满足(1)的称作**小顶堆（最小堆）**。例如：

{12，36，24，85，47，30，53，91}

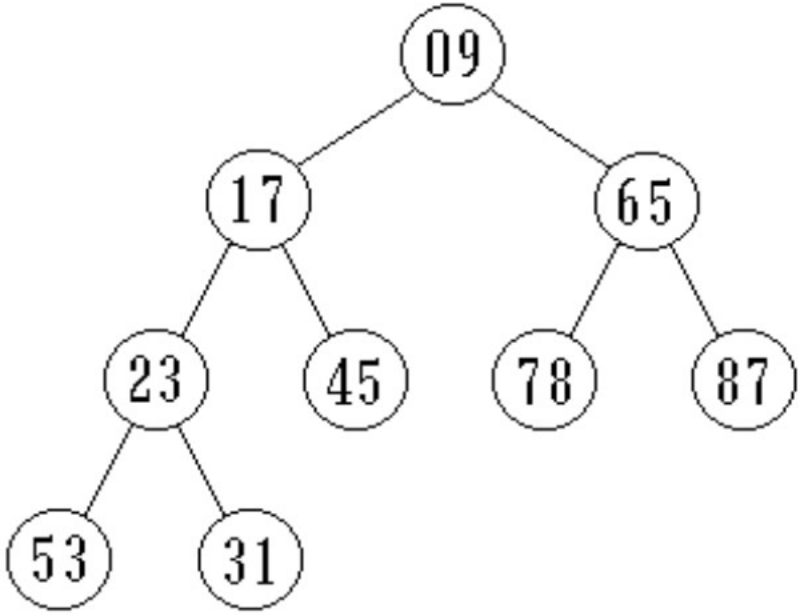
{9，17，65，23，45，78，87，54，31}

满足(2)的称作**大顶堆（最大堆）**。例如：

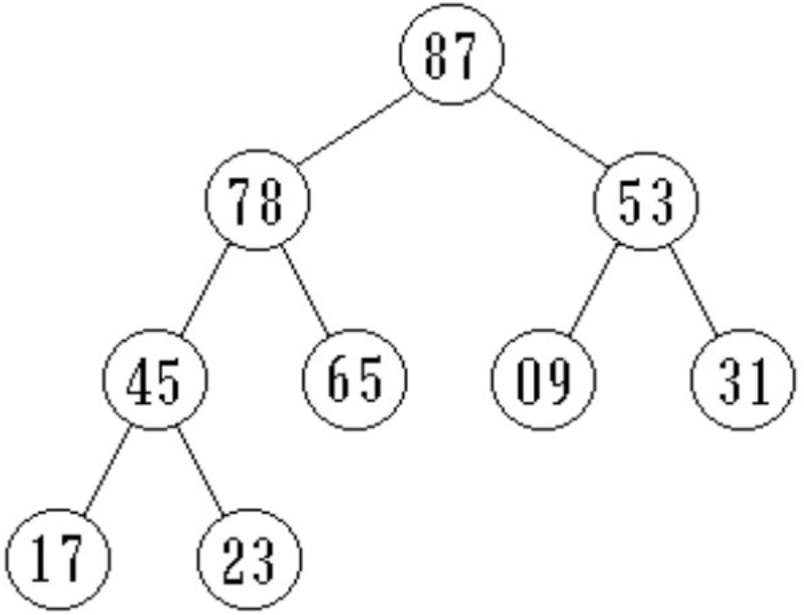
{96，83，27，3，11，09}

{87，78，53，45，65，09，31，17，23}

堆 (Heap)



(a) 最小堆



(b) 最大堆

最小堆的数组表示

$K_i \leq K_{2i}$ &&
$K_i \leq K_{2i+1}$ (i>0)

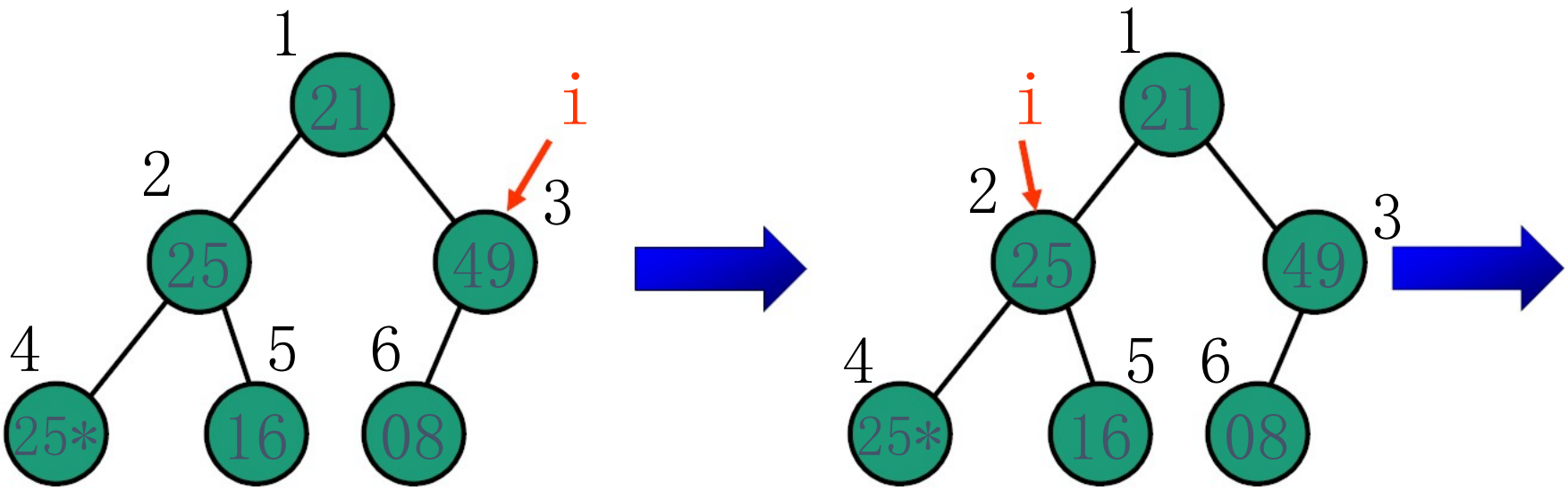
最大堆的数组表示

$K_i \geq K_{2i}$ &&
$K_i \geq K_{2i+1}$ (i>0)

堆排序

- 利用堆及其运算，可以很容易地实现选择排序的思路。
- 堆排序分为两个步骤：
  1. 建立初始堆。根据初始输入数据，利用堆的调整算法 HeapAdjust形成初始堆(大顶堆)；
  2. 通过一系列的对象交换和重新调整堆进行排序。
     - 最大堆的第一个对象r[1]具有最大的关键字，将r[1]与r[n]对调，把具有最大关键字的对象交换到最后，再对前面的n-1个对象，使用堆的调整算法HeapAdjust(H,1,n-1)，重新建立最大堆。结果具有次最大关键字的对象又上浮到堆顶，即r[1]位置。
     - 再对调r[1]和r[n-1]，调用HeapAdjust (H,1,n-2)，对前n-2个对象重新调整，…。
     - 如此反复执行，最后得到全部排序好的对象序列。这个算法即堆排序算法。

建立初始的最大堆

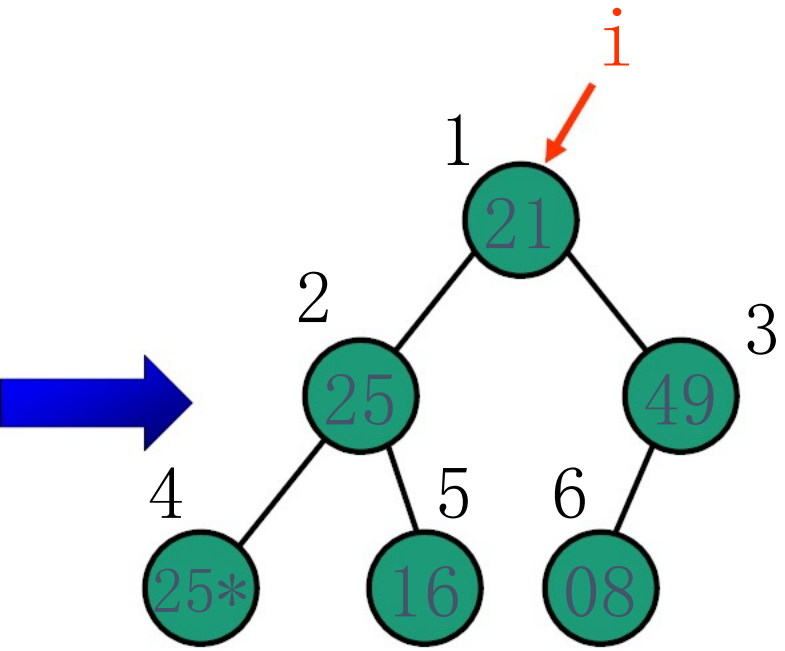


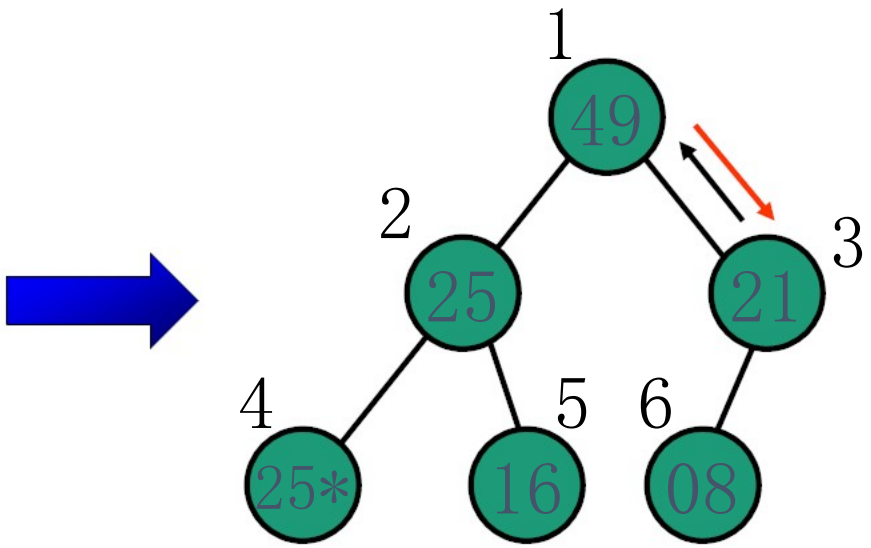
21	25	49	25*	16	08

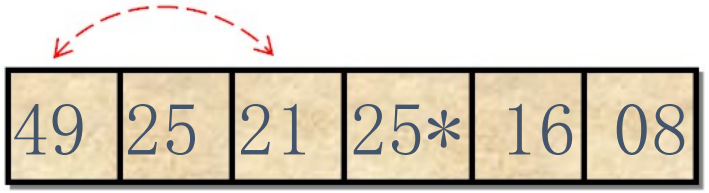
初始关键字集合

21	25	49	25*	16	08

i = 3 时的局部调整







i = 2 时的局部调整

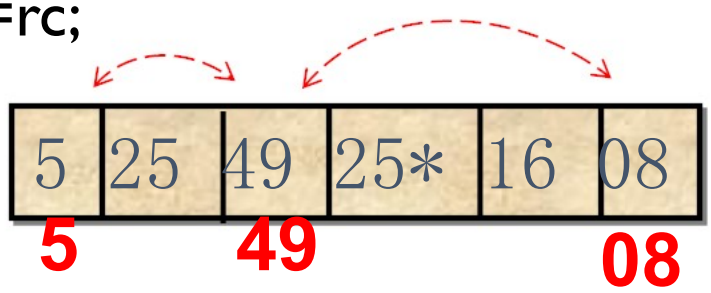
i = 1 时的局部调整
形成大顶堆

最大堆的向下调整算法

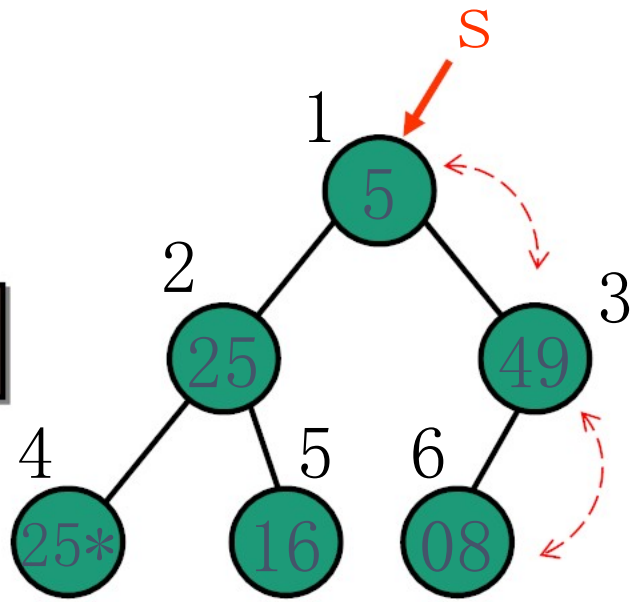
假设以s为根的二叉树，只有根节点s不满足堆的性质。m为堆的元素个数。

```
void HeapAdjust(HeapType &H,int s,int m){
  ElemType rc=H.r[s];
  for (int j=2*s;j<=m;j*=2){
    if ((j<m) && (LT(H.r[j].key,H.r[j+1].key))) ++j;
          if (!(LT(rc.key,H.r[j].key))) break;
          H.r[s].key=H.r[j].key; s=j;
        }//end for
  H.r[s]=rc;
}
```

例如H：



s=1,m=6



建堆

- 堆的向下调整算法HeapAdjust是一个自堆顶至叶子的调整过程，也称为“筛选”。
- 从无序序列建堆的过程是一个反复“筛选”的过程。
- 具体步骤：

  1.将无序序列存放在数组中，下标从1开始。可将此数组看作完全二叉树的存储结构。

  2.选取最后一个非终端结点i，即下标为⌊n/2⌋的结点。

  3. 调用向下调整HeapAdjust(H,i,n);

  4. i--,

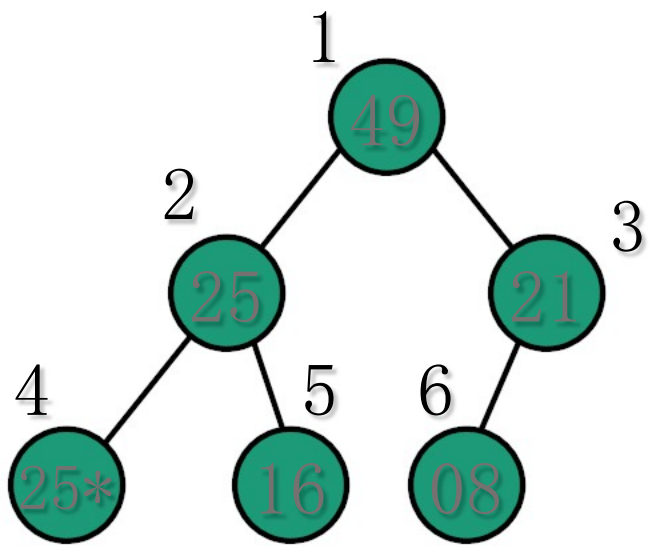
  5. 重复上面3，4步，直到i==1结束。

建堆代码

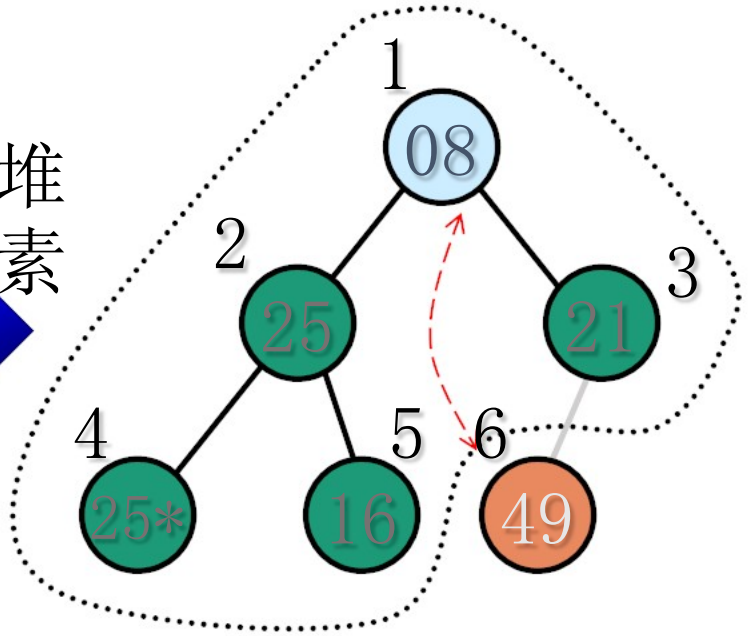
- 从最后1个非终端结点开始，自下而上，一直到根节点，依次调用向下调整算法。

(Robert Floyd，1964)

```
for (int i=H.length/2; i>0; --i)
        HeapAdjust(H,i,H.length);
```



删除堆顶元素



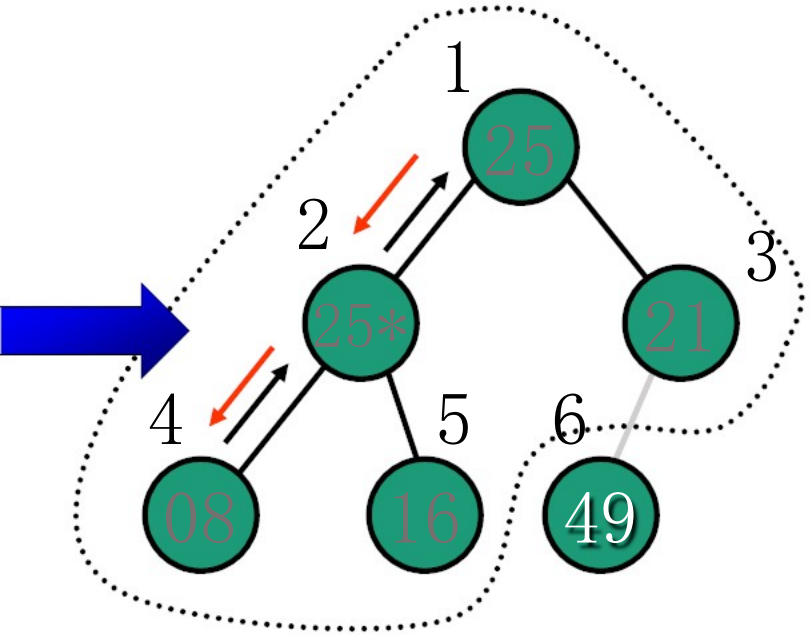
堆顶向下调整

49	25	21	25*	16	08

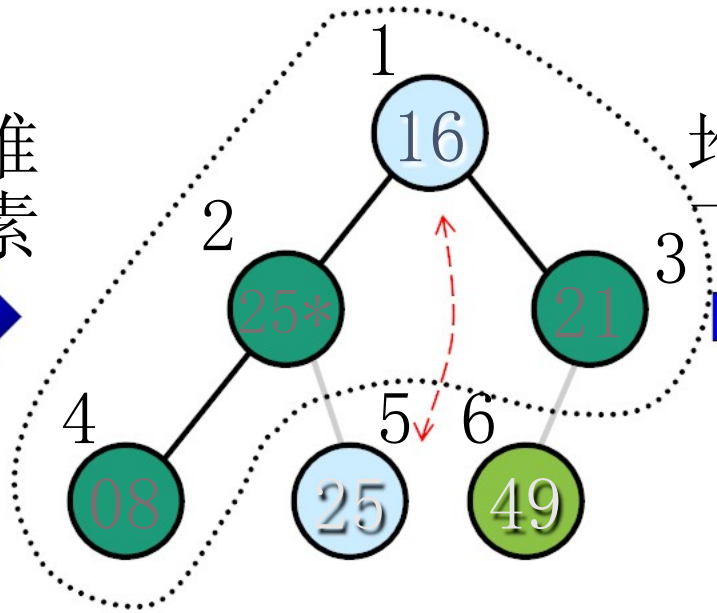
初始最大堆

08	25	21	25*	16	49

交换 1 号与 6 号对象，6 号对象就位



删除堆顶元素



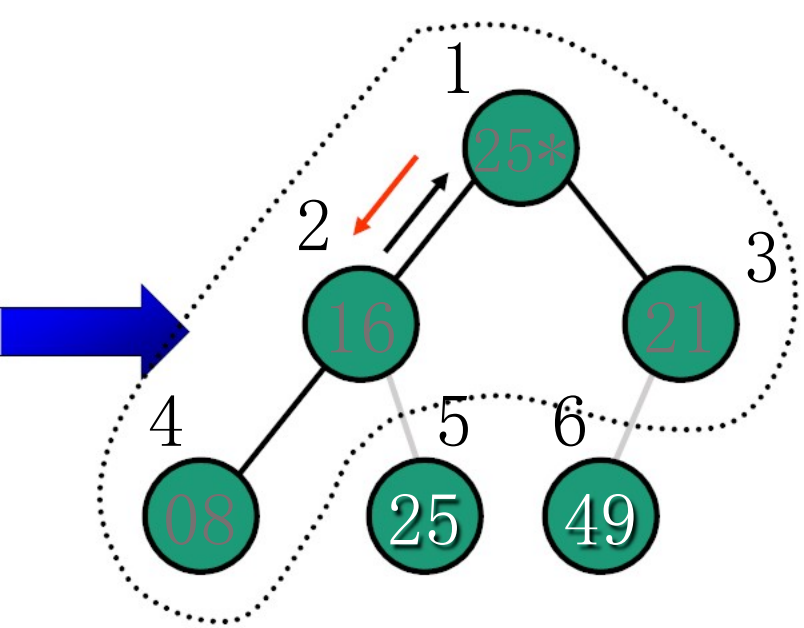
堆顶向下调整

25	25*	21	08	16	49

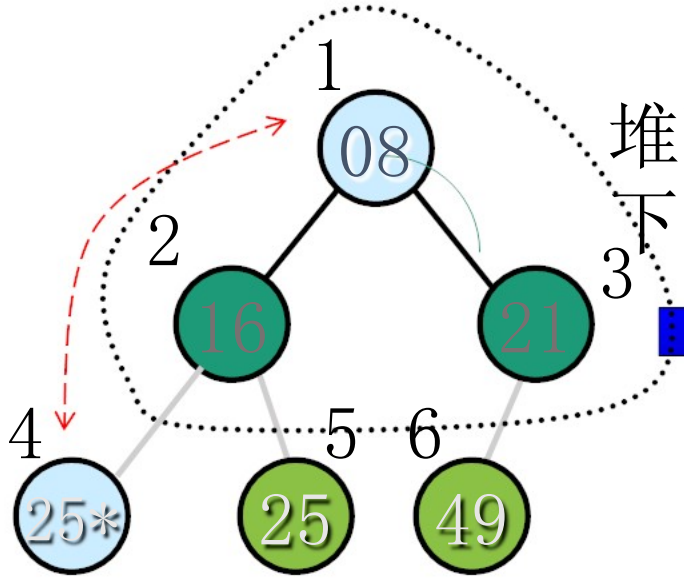
从 1 号到 号 重新调整为最大堆

16	25*	21	08	25	49

交换 1 号与 5 号对象，5 号对象就位



删除堆顶元素



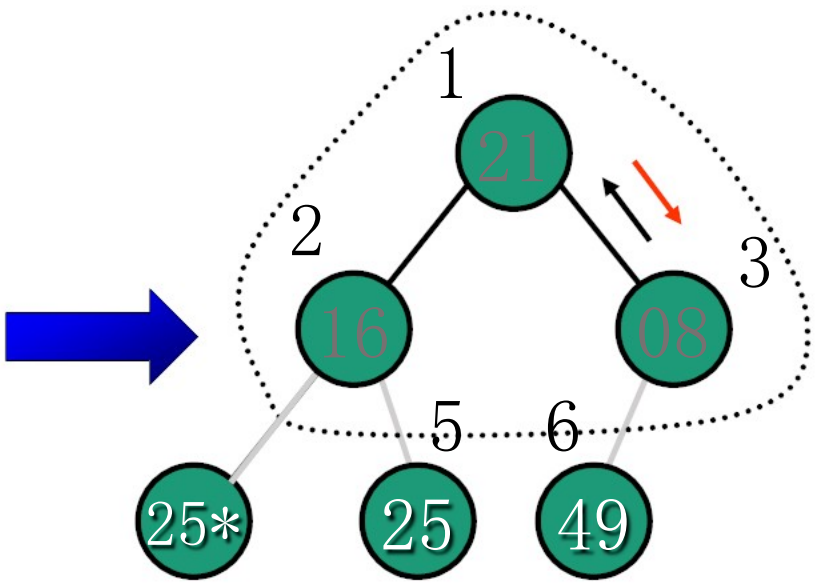
堆顶向下调整

25*	16	21	08	25	49

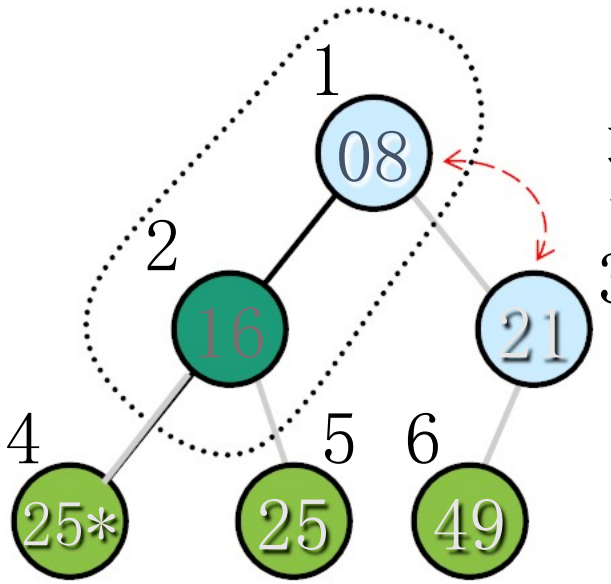
从 1 号到 4 号 重新调整为最大堆

08	16	21	25*	25	49

交换 1 号与 4 号对象，4 号对象就位



删除堆顶元素



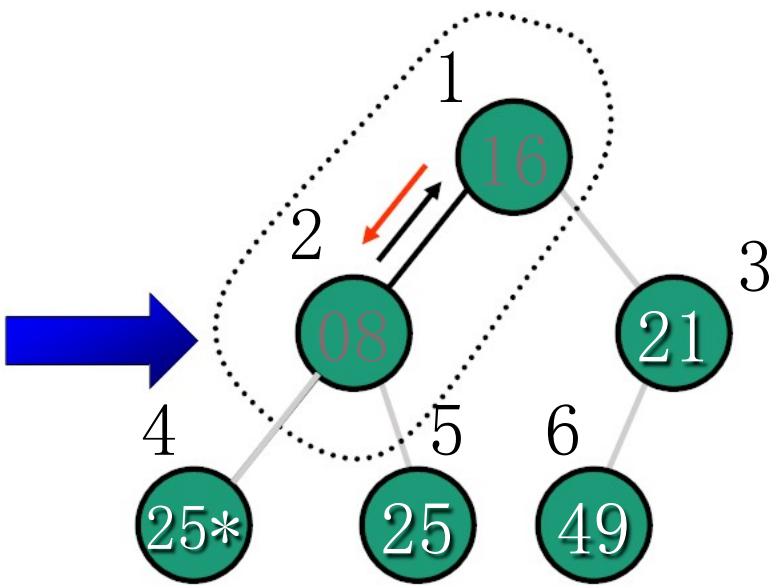
堆顶向下调整

21	16	08	25*	25	49

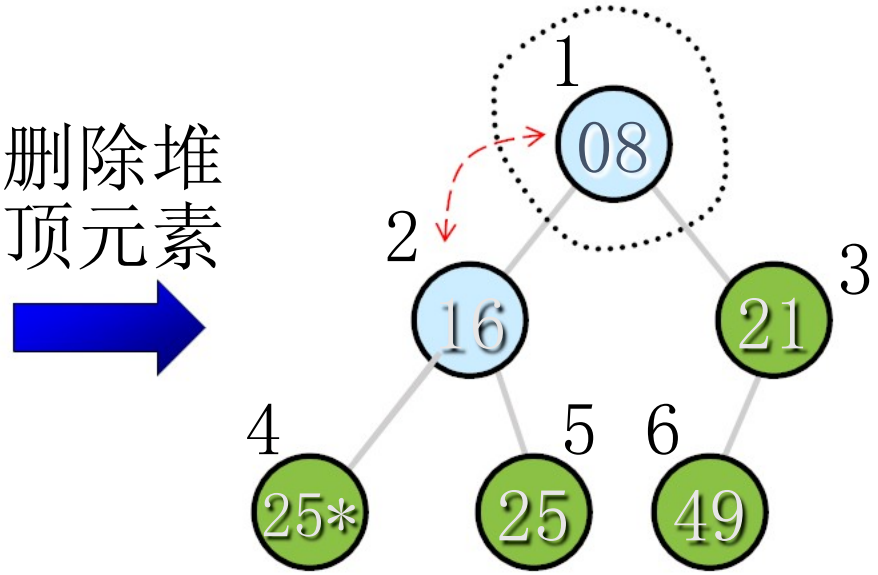
从 1 号到 3 号 重新调整为最大堆

08	16	21	25*	25	49

交换 1 号与 3 号对象，3 号对象就位



删除堆顶元素



16	08	21	25*	25	49

从 1 号到 2 号 重新调整为最大堆

08	16	21	25*	25	49

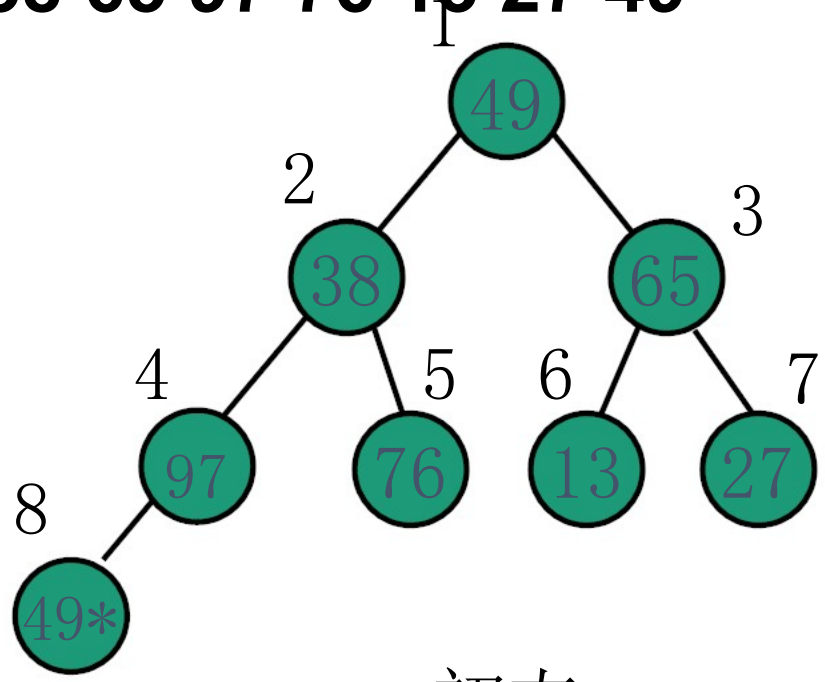
交换 1 号与 2 号对象，2 号对象就位

■堆排序的算法

```
void HeapSort(HeapType &H) {
    ElemType temp;
    for (int i=H.length/2; i>0; --i)   //建堆
            HeapAdjust(H,i,H.length);
    for (i=H.length; i>1; --i)    //排序
      { temp=H.r[1];
            H.r[1]=H.r[i];
            H.r[i]=temp;
            HeapAdjust(H,1,i-1);
      }
}
```

例如：对下列序列建立大顶堆

49 38 65 97 76 13 27 49*



初态

堆排序过程：

1. 初态：原始序列
2. 建立大顶堆
3. 堆排序：

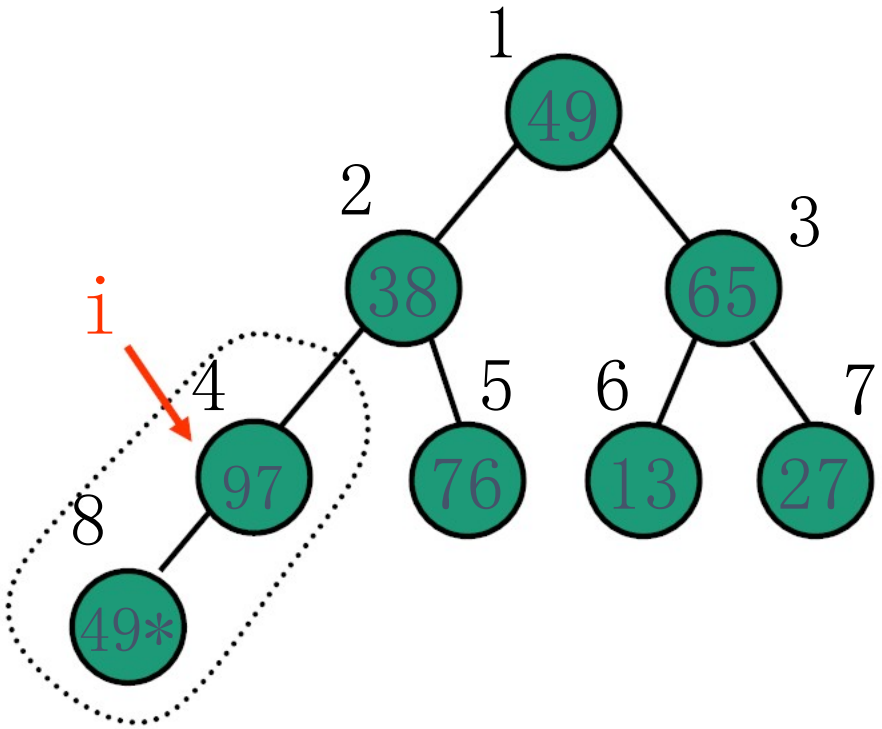
a)将堆顶元素与堆的最后一个元素交换，堆的元素个数减1。

b)堆顶元素做向下调整。

c)重复a、b, 直到堆的元素个数=1。排序完成。

49	38	65	97	76	13	27	49*

关键字集合



2. 建堆过程：

元素个数为m

令i=[m/2],

（1）对第i个元素做向下调整;

（2）i--;

（3）重复（1）（2），直到i<1

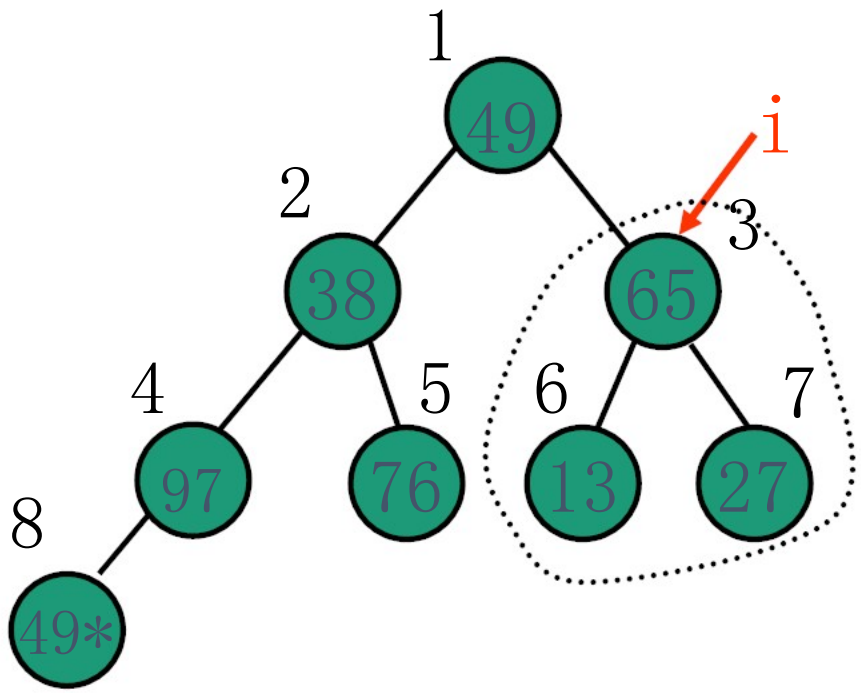
如左图：m=8, i=4;

此时i=4这棵子树已经是一个大顶堆，不需调整。

49	38	65	97	76	13	27	49*

初态

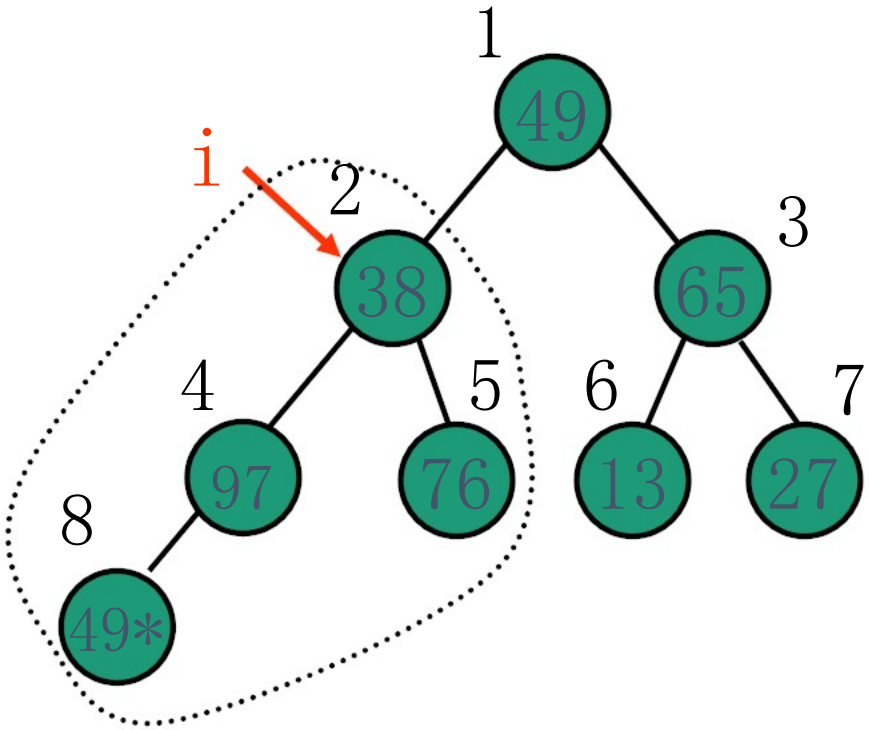
关键字集合



i=3,
对65做向下调整
65>max(13,27)
i=3这棵子树已经是一个大顶堆，不需调整

49	38	65	97	76	13	27	49*

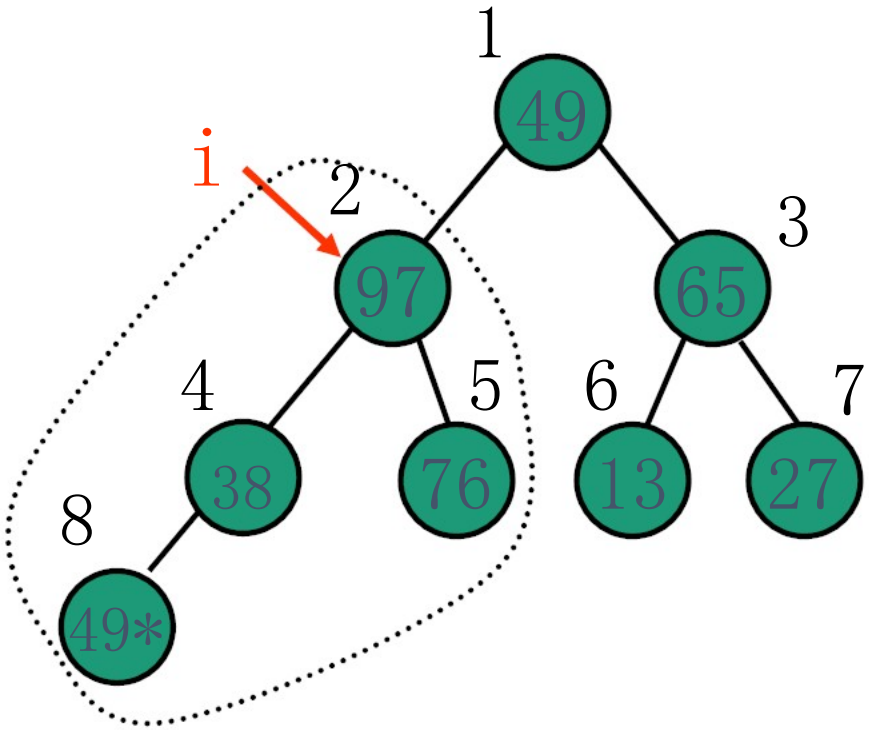
关键字集合



i=2,
对38做向下调整：
38与97交换；
38与49*交换

49	38	65	97	76	13	27	49*

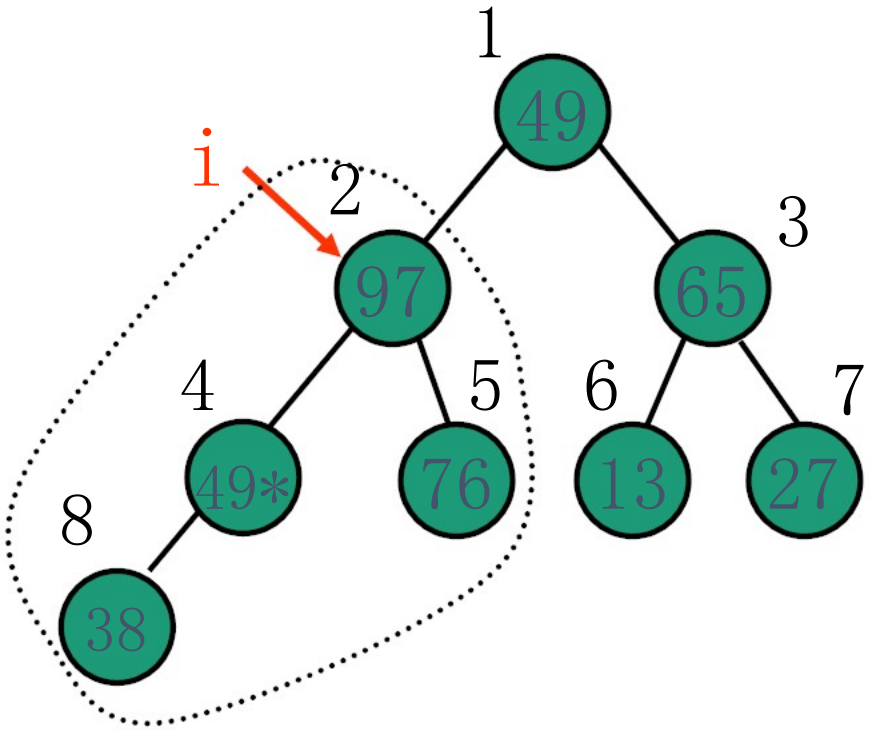
关键字集合



i=2,
对38做向下调整：
38与97交换；
38继续与49*交换

49	97	65	38	76	13	27	49*

关键字集合

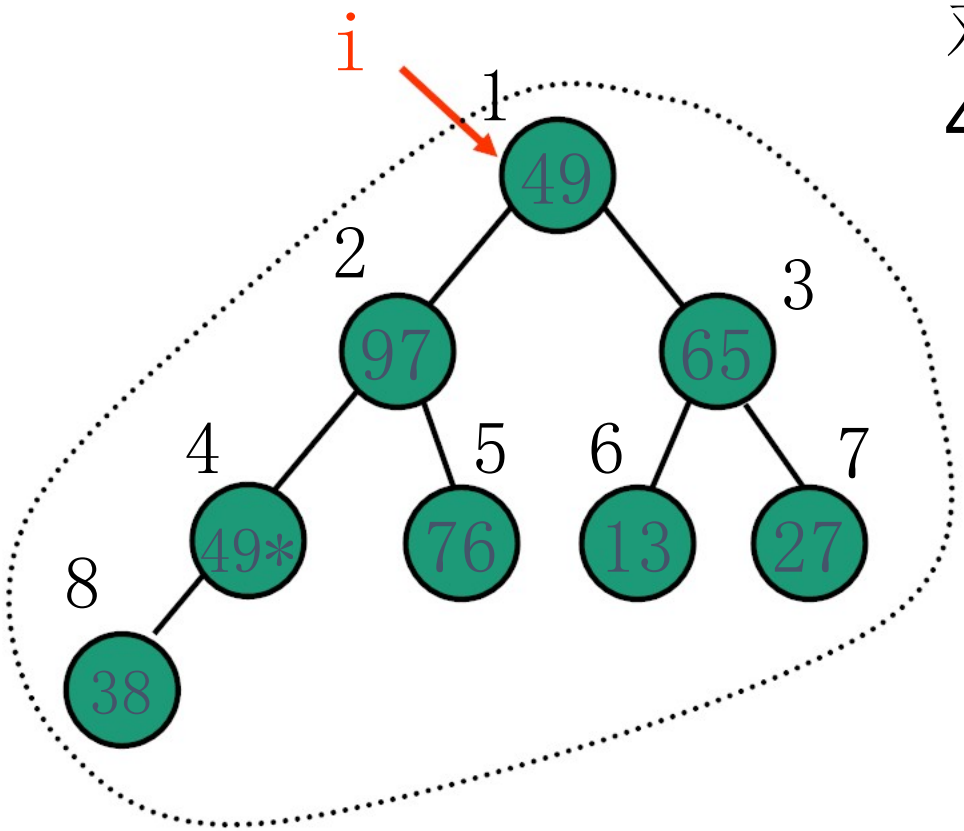


i=2,
对38做向下调整：
38与97交换；
38继续与49*交换
38成为叶子节点，38调整结束
此时i=2这棵子树成为大顶堆

49	97	65	49*	76	13	27	38

关键字集合

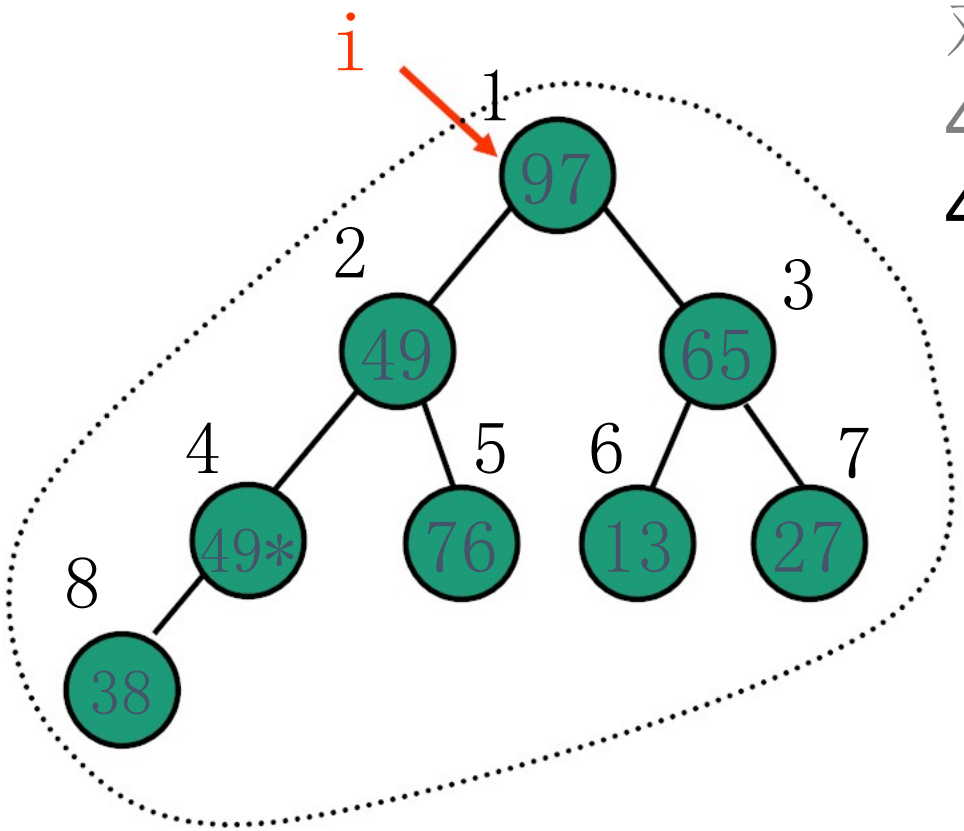
i=1,
对49做向下调整：
49与97交换；



49	97	65	49*	76	13	27	38

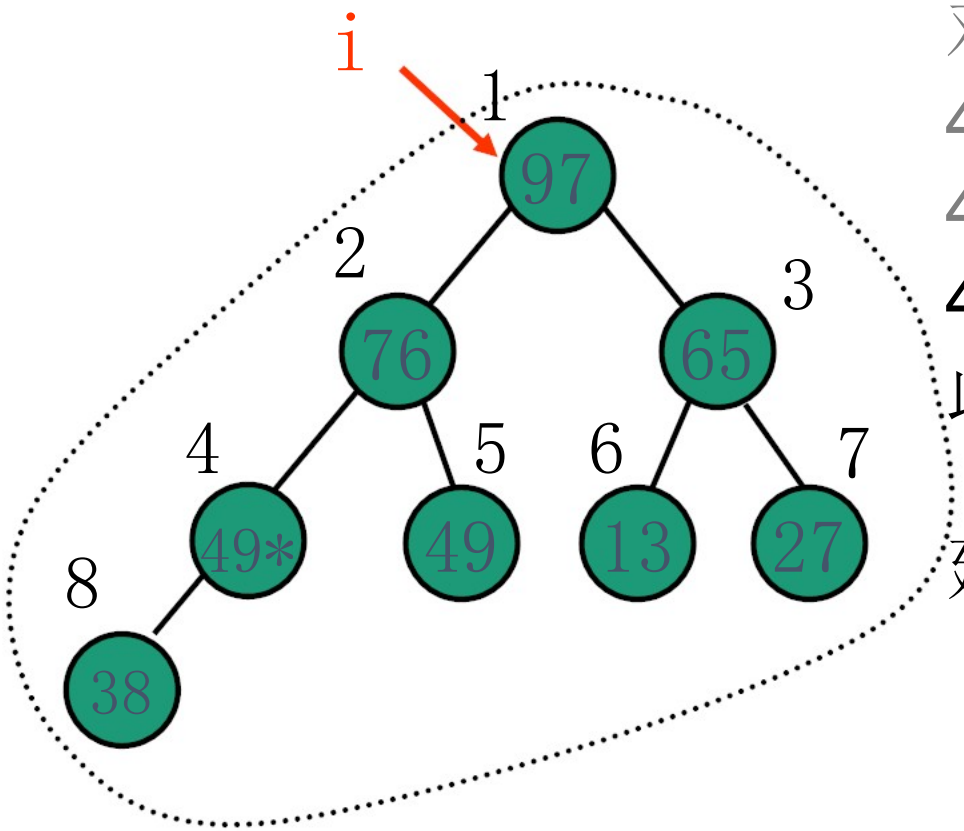
关键字集合

i=1,
对49做向下调整：
49与97交换；
49继续与76交换



97	49	65	49*	76	13	27	38

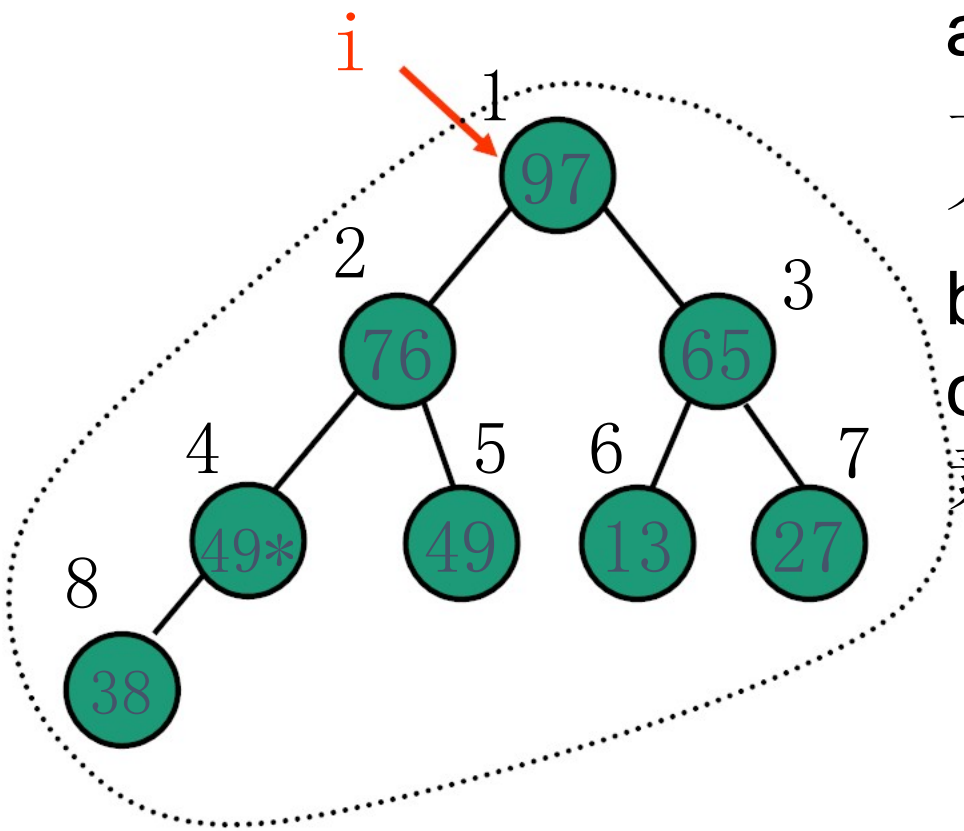
关键字集合



i=1,
对49做向下调整：
49与97交换；
49继续与76交换
49是一个叶子节点
此时49调整结束
i=1这棵子树成为大顶堆
建堆完成。

97	76	65	49*	49	13	27	38

关键字集合



3. 堆排序：

a)将堆顶元素与堆的最后一个元素交换，堆的元素个数减1。

b)堆顶元素做向下调整。

c)重复a、b, 直到堆的元素个数=1。排序完成。

97	76	65	49*	49	13	27	38

关键字集合

算法分析

- 建堆：

  若设堆中有 n 个结点，且 2^(k-1) <= n < 2^k，则对应的完全二叉树有 k 层。在第 i 层上的结点数 <= 2^(i-1) (i = 1,…,k)。在第一个形成初始堆的for循环中对每一个非叶结点调用了一次堆调整算法 HeapAdjust ( )，因此该循环所用的计算时间为：

$$2 \cdot \sum_{i=1}^{k-1} 2^{i-1} \cdot (k-i)$$

  其中，i 是层序号，2^{i-1} 是第 i 层的最大结点数，(k-i) 是第 i 层结点能够移动的最大距离。

$$2\cdot\sum_{i=1}^{k-1}2^{i-1}\cdot(k-i)=\sum_{j=1}^{k-1}2^{k-j}\cdot j=$$

$$=2\cdot 2^{k-1}\sum_{j=1}^{k-1}\frac{j}{2^j}\le 2\cdot n\sum_{j=1}^{k-1}\frac{j}{2^j}\le 4n$$

- 排序过程：交换和调整

  在第二个for循环中，调用了n-1次HeapAdjust算法，该循环的计算时间为$O(nlog_2n)$。因此，堆排序的时间复杂性为$O(nlog_2n)$。适合于文件比较大的情况。

- 附加存储：O(1)

  主要是在第二个for循环中用来执行对象交换时所用的一个临时对象。

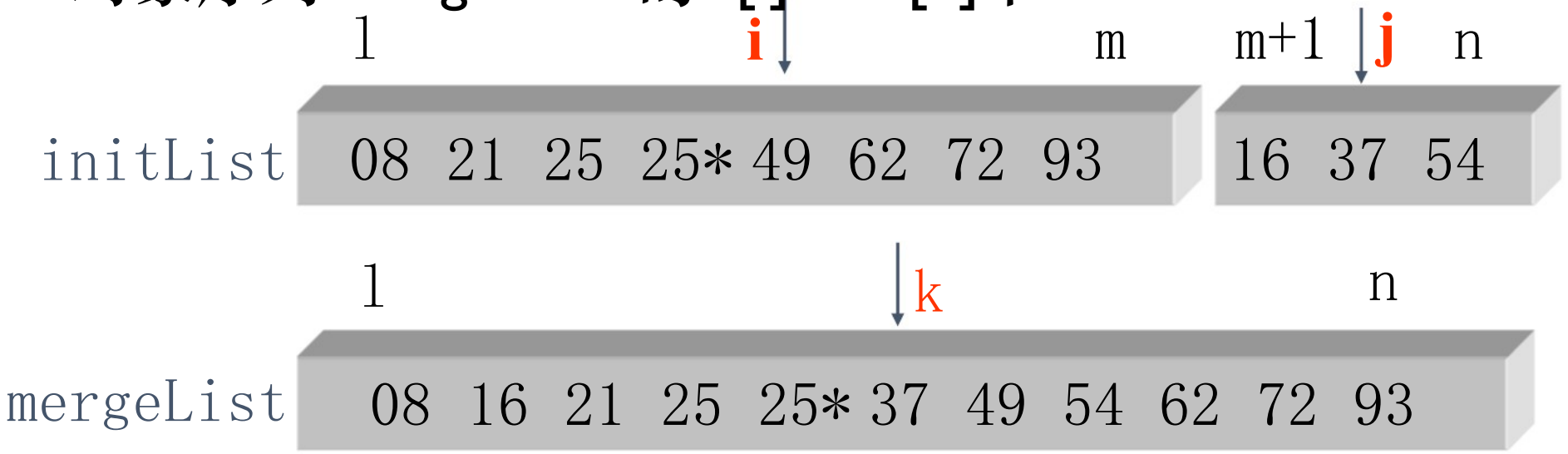
- 堆排序是一个不稳定的排序方法。

■堆排序的算法

```
void HeapSort(HeapType &H) {
    ElemType temp;
    for (int i=H.length/2; i>0; --i)   //建堆
            HeapAdjust(H,i,H.length);
    for (i=H.length; i>1; --i)    //排序
      { temp=H.r[1];
            H.r[1]=H.r[i];
            H.r[i]=temp;
            HeapAdjust(H,1,i-1);
      }
}
```

归并排序 (Merge Sort)

- 归并，是将两个或两个以上的有序表合并成一个新的有序表。
- 对象序列 initList 中有两个有序表V[l] …V[m]和V[m+l] …V[n]。它们可归并成一个有序表，存于另一对象序列 mergedList 的V[l] …V[n]中。



- 这种归并方法称为2-路归并 (2-way merging)。

归并排序 (Merge Sort)

- 2-路归并算法的基本思想是:

  设两个有序表A和B的对象个数(表长)分别为 al 和 bl，变量 i 和 j 分别是表A和表B的当前检测指针。设表C是归并后的新有序表，变量 k 是它的当前存放指针。

  当 i 和 j 都在两个表的表长内变化时，根据A[i]与B[j]的关键字的大小，依次把关键字小的对象排放到新表C[k]中；

  当 i 与 j 中有一个已经超出表长时，将另一 个表中的剩余部分照抄到新表C[k]中。

两路归并

```
void Merge(ElemType SR[],ElemType TR[],int i, int m,int n){ int k,j;
    for (j=m+1,k=i;i<=m && j<=n; ++k){
            if (LQ(SR[i].key ,SR[j].key))
                    TR[k] = SR[i++];
            else TR[k] = SR[j++];
    }
    if (i<=m)
      for (int n1=k,n2=i;n1<=n && n2<=m;n1++,n2++)
            TR[n1]=SR[n2];
    if (j<=n)
    for (int n1=k,n2=j;n1<=n && n2<=n;n1++,n2++)
            TR[n1]=SR[n2];
}
```

归并排序算法

- 迭代的归并排序算法就是利用两路归并过程进行排序的。
- 基本思想是：

  假设初始对象序列有 n 个对象，首先把它看成是 n 个长度为 1 的有序子序列 (归并项)，先做两两归并，得到 [n / 2]个长度为 2 的归并项 (如果 n 为奇数，则最后一个有序子序列的长度为1)；再做两两归并，…，如此重复，最后得到一个长度为 n 的有序序列。

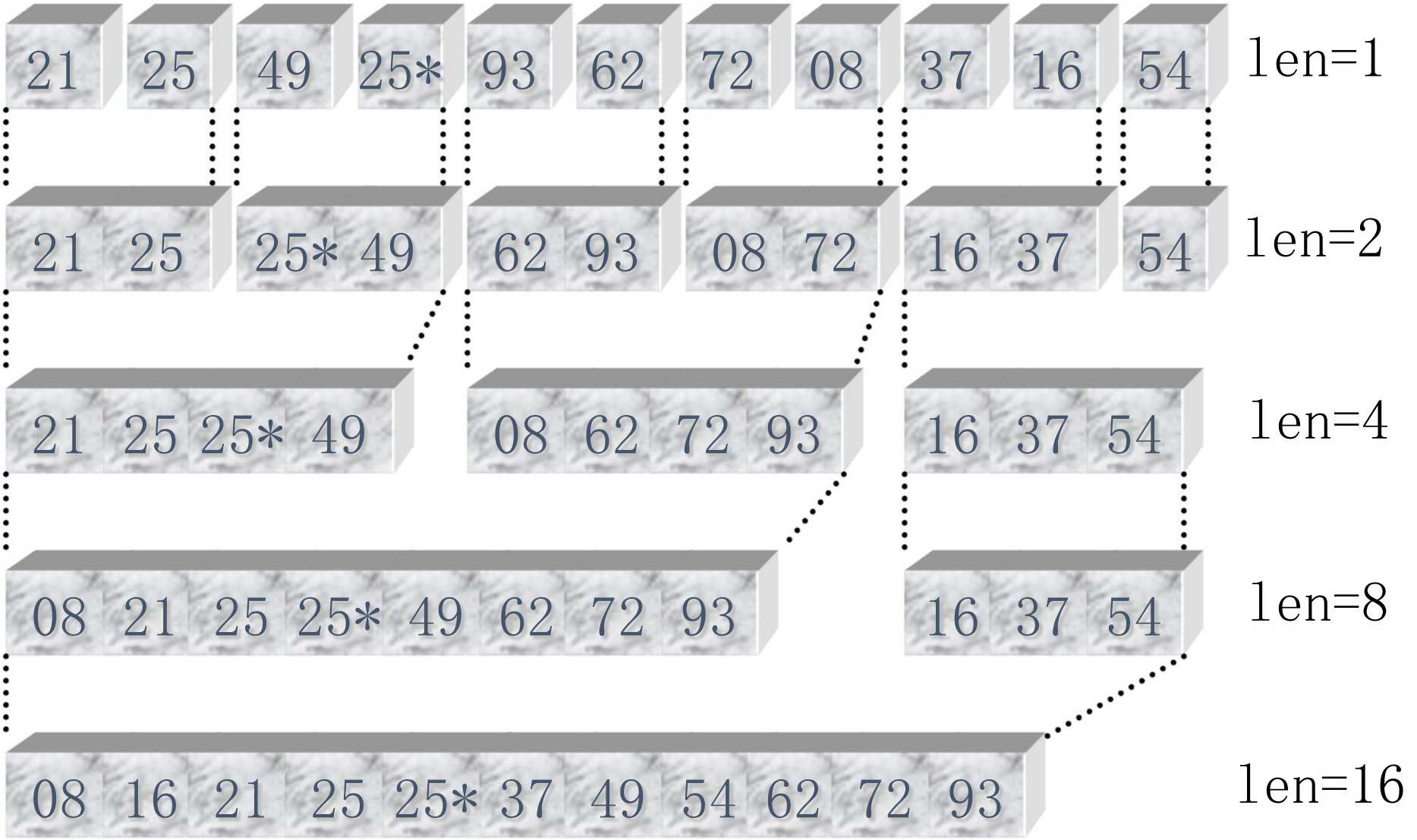
迭代的归并排序（子序列长度为k）

```
void MSort1(ElemType a[], ElemType b[], int n, int k) {
   int i=1;       ElemType b2[MAXSIZE];
   while (n-i+1 >= 2*k){//可以形成偶对
    {  Merge (a,b2,i,i+k-1,i+2*k-1);
       i += 2*k;    }
    if(n - i+1 > k) //大于一个文件长度
       Merge (a,b2,i,i+k-1,n);
    else // 小于一个文件的长度，无须归并
                  for(;i<=n;i++) b2[i]=a[i];
         for (i=1;i<=n;i++) b[i]=b2[i];
}
```

归并排序（迭代）

```
void MergeSort(SqList &L){
    int len=1;
    for (;len<L.length;len*=2)
            MSort1(L.r,L.r,L.length,len);
}
```

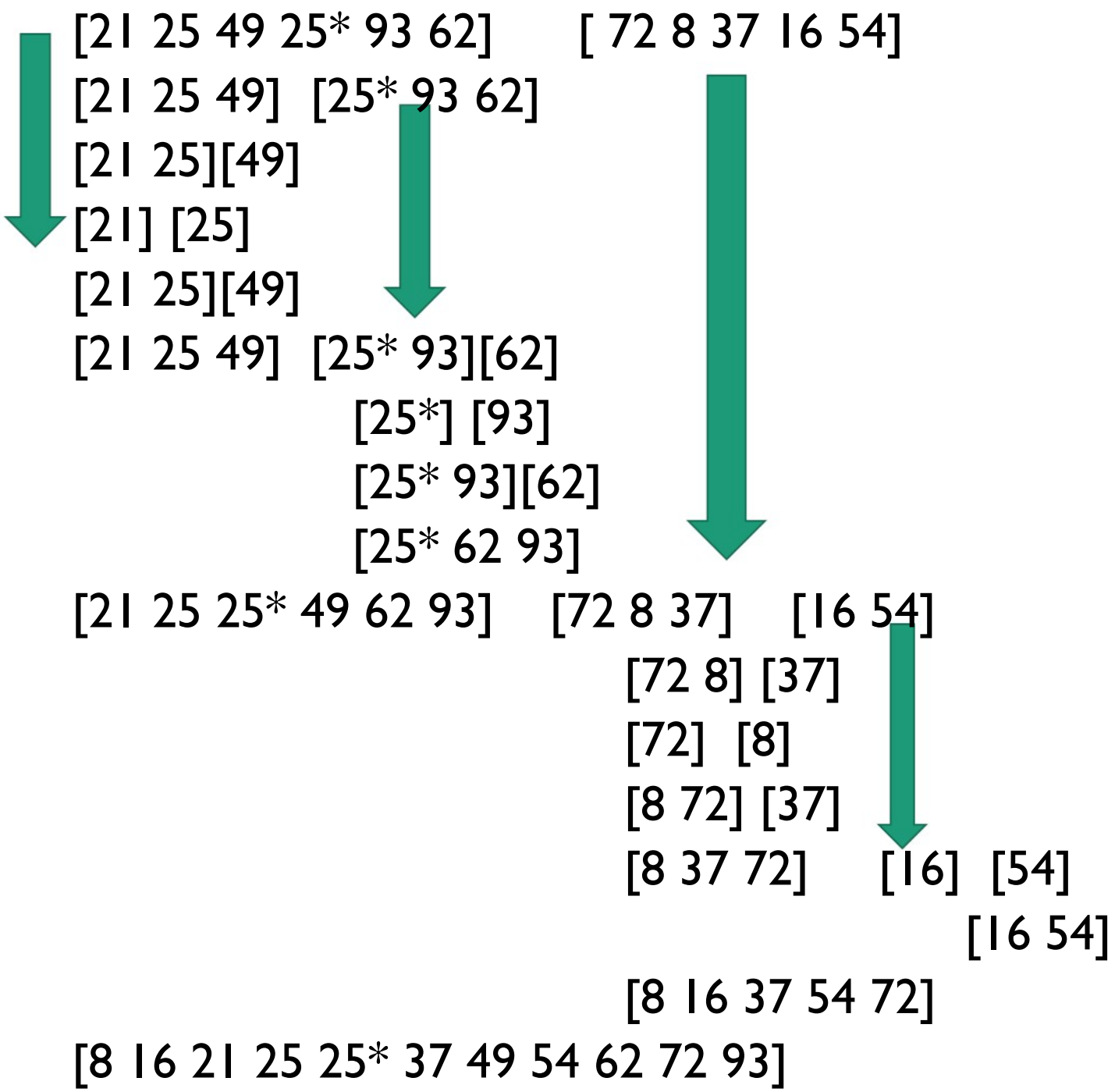
归并排序算法（迭代）

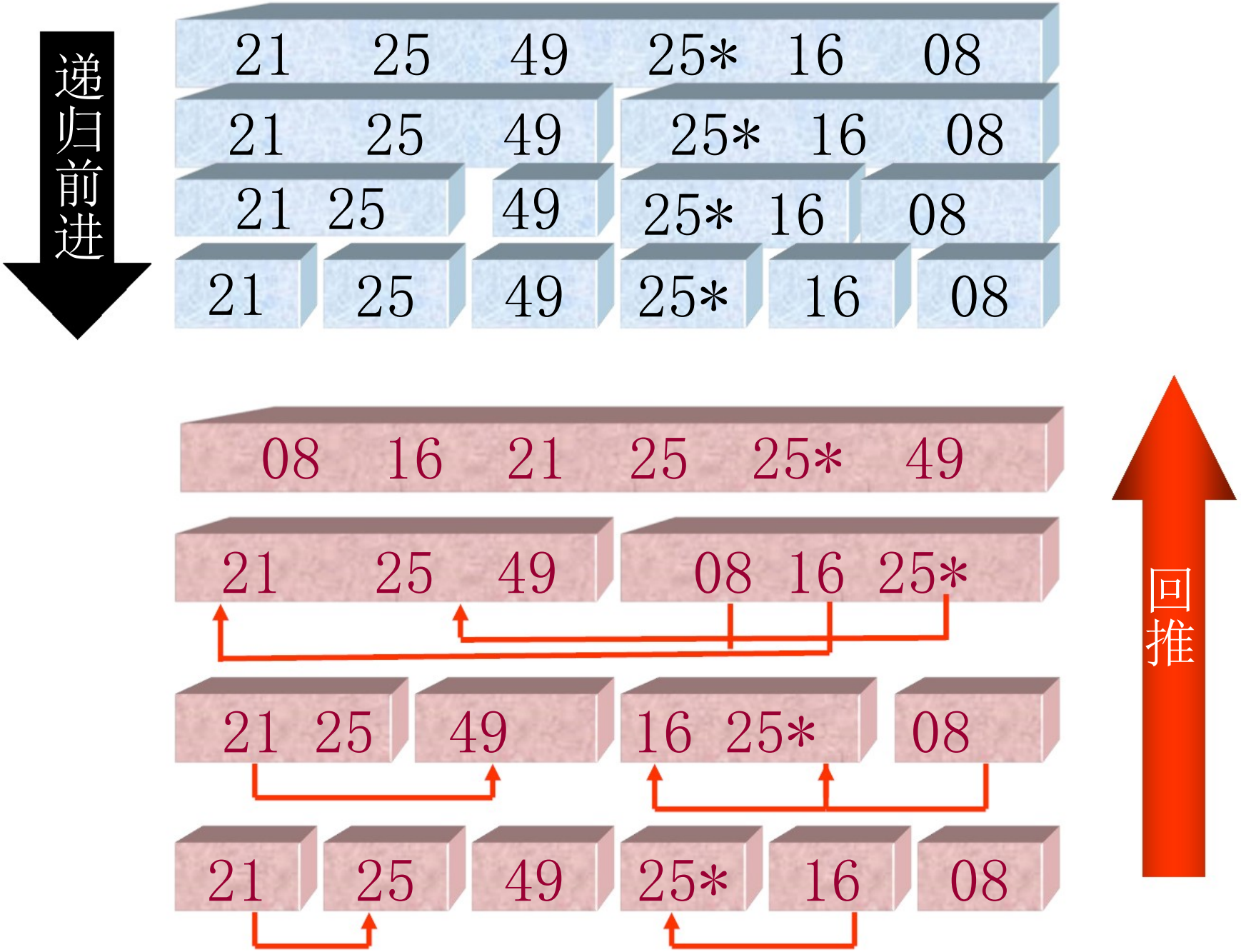

21 25 49 25* 93 62 72 08 37 16 54
len=1
21 25 25* 49 62 93 08 72 16 37 54
len=2
21 25 25* 49 08 62 72 93 16 37 54
len=4
08 21 25 25* 49 62 72 93 16 37 54
len=8
08 16 21 25 25* 37 49 54 62 72 93
len=16

递归的归并排序算法

```
void MSort(ElemType SR[],ElemTypeTR1[],int s,int t){//s是起始位置，t是结束位置
        ElemType *TR2=
(ElemType*)malloc(sizeof(ElemType)*MAXSIZE);
        if (s==t)TR1[s]=SR[s];
        else {int m=(s+t)/2;
                MSort(SR,TR2,s,m);
                MSort(SR,TR2,m+1,t);
                Merge(TR2,TR1,s,m,t);
        } free(TR2);//一定要释放
}
void MergeSort  (SqList &L){
        MSort(L.r,L.r,1,L.length);}
```

递归过程举例


[21 25 49 25* 93 62]
[72 8 37 16 54]
[21 25 49]
[25* 93 62]
[21 25][49]
[21] [25]
[21 25][49]
[21 25 49]
[25* 93][62]
[25*] [93]
[25* 93][62]
[25* 62 93]
[21 25 25* 49 62 93]
[72 8 37]
[16 54]
[72 8] [37]
[72] [8]
[8 72] [37]
[8 37 72]
[16] [54]
[16 54]
[8 16 37 54 72]
[8 16 21 25 25* 37 49 54 62 72 93]


递归前进
21 25 49 25* 16 08
21 25 49 25* 16 08
21 25 49 25* 16 08
21 25 49 25* 16 08
08 16 21 25 25* 49
21 25 49 08 16 25*
21 25 49 16 25* 08
21 25 49 25* 16 08
回推

算法分析

- 时间复杂度 $O(n\log_2 n)$。

  在归并排序算法中，递归深度为$O(\log_2 n)$，对象关键字的比较次数为$O(n\log_2 n)$。

- 空间复杂度 $O(n)$

  归并排序占用附加存储较多，需要另外一个与原待排序对象数组同样大小的辅助数组。这是这个算法的缺点。

- 归并排序是一个稳定的排序方法。