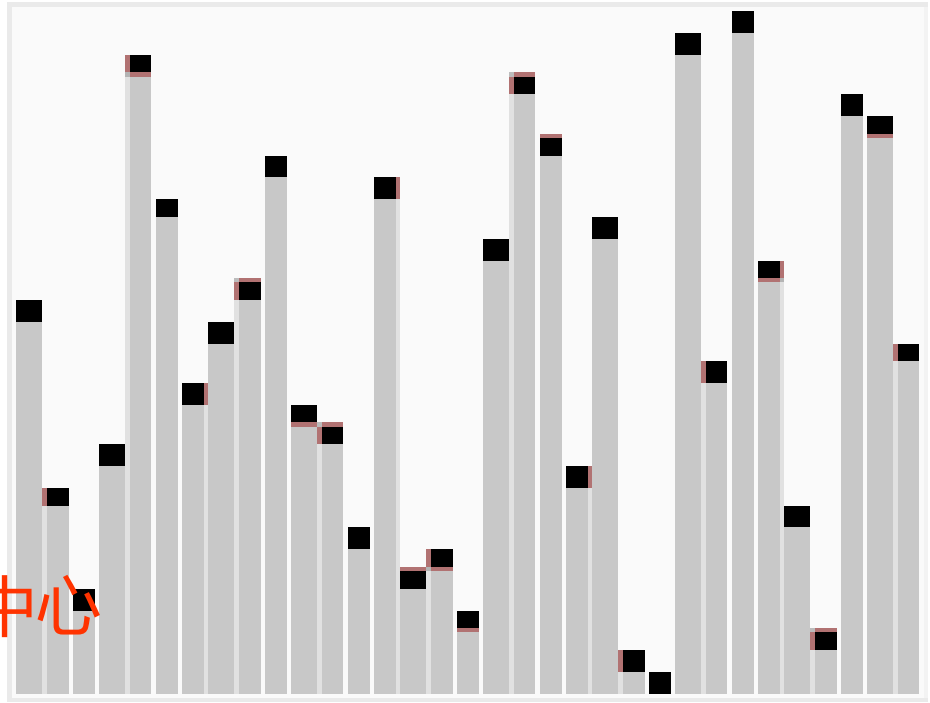
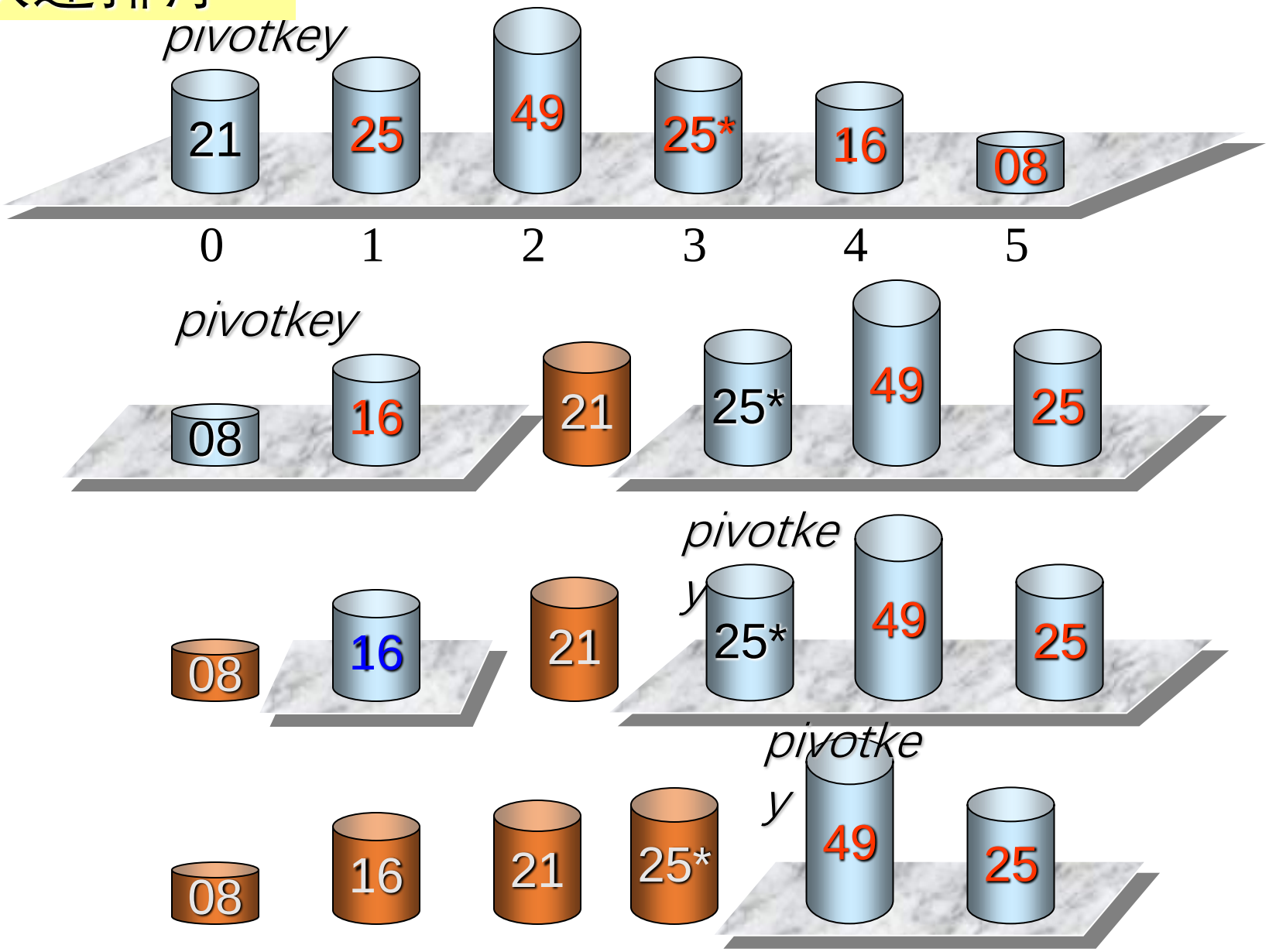
快速排序

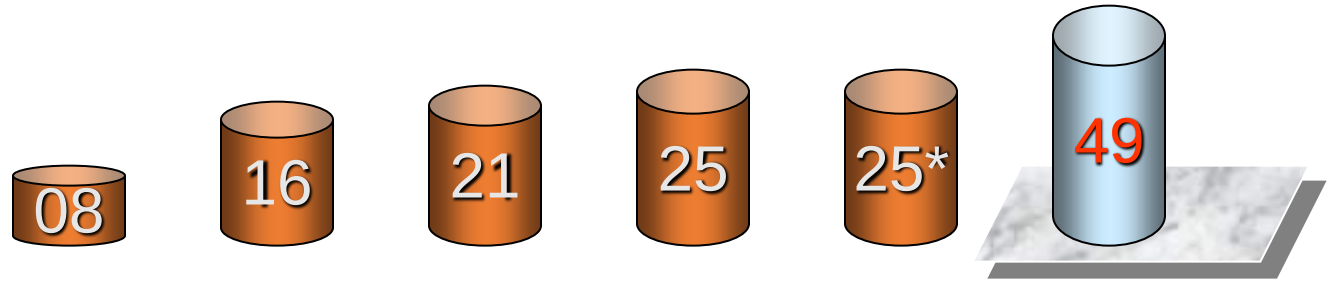


基本思想：

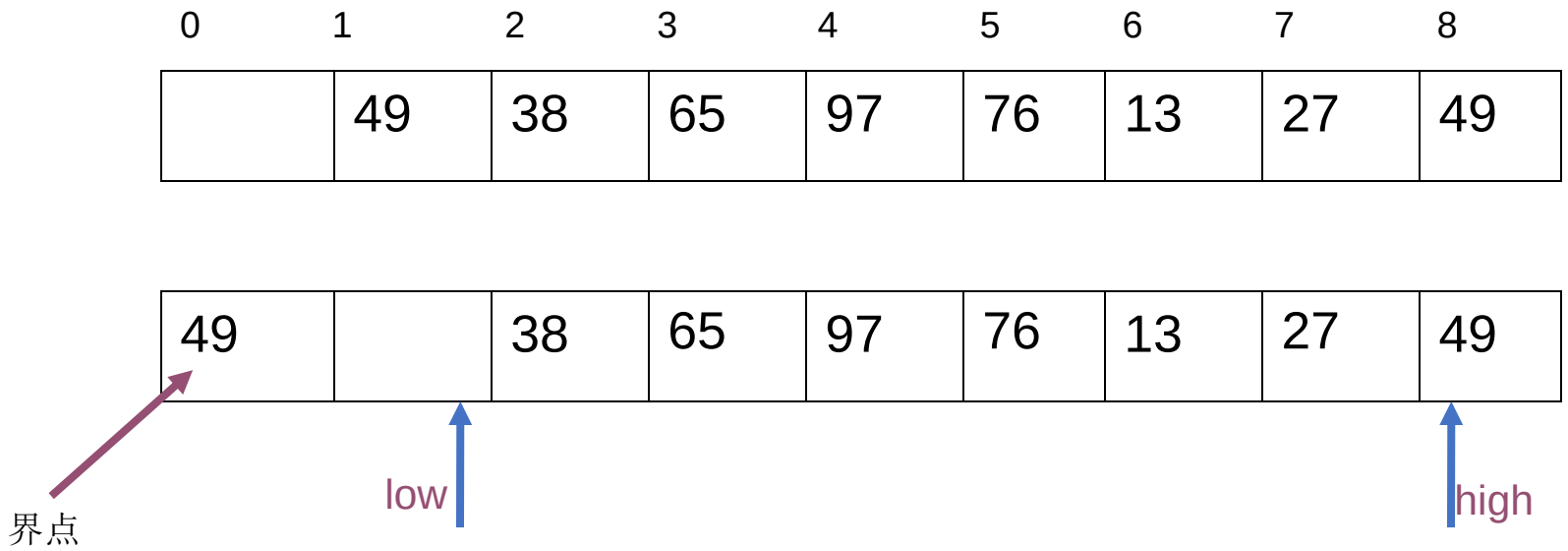
- 任取一个元素 (如第一个) 为中心
- 所有比它小的元素一律前放，比它大的元素一律后放，形成左右两个子表；
- 对各子表重新选择中心元素并依此规则调整，直到每个子表的元素只剩一个

快速排序

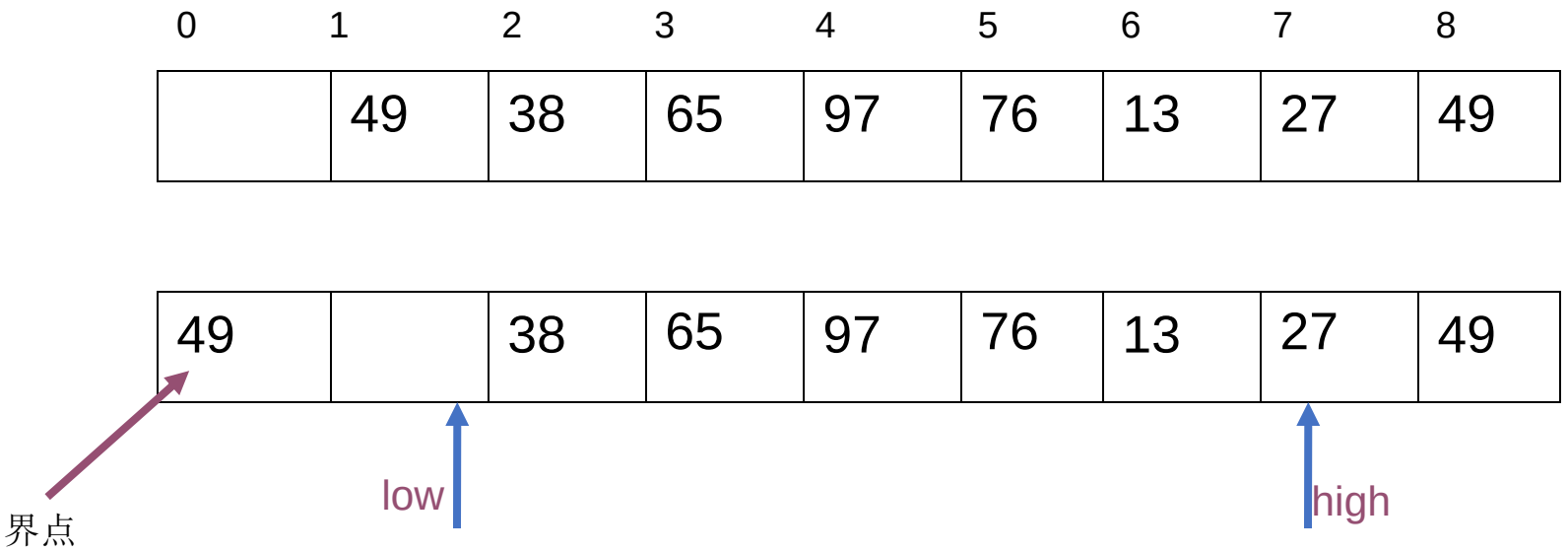

pivotkey
21
25
49
25*
16
08
0
1
2
3
4
5
pivotkey
08
16
21
25*
49
25
pivotke
y
08
16
21
25*
49
25
pivotke
y
08
16
21
25*
49
25


08
16
21
25
25*
49

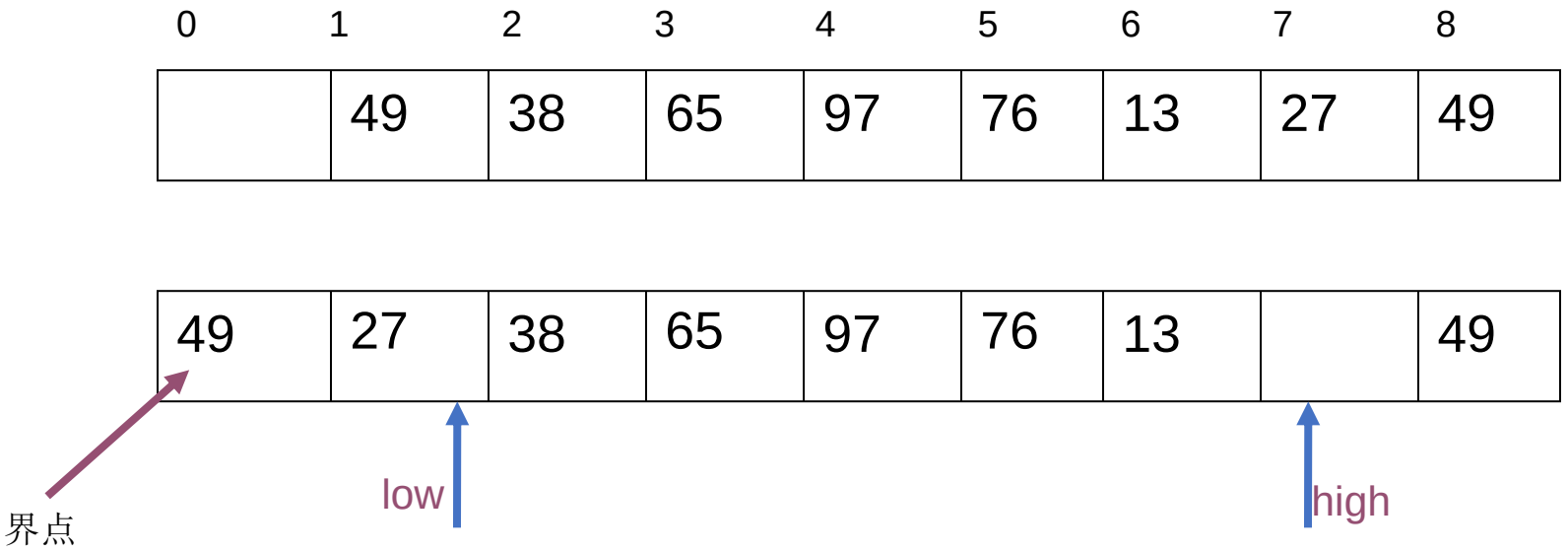
快速排序


0
1
2
3
4
5
6
7
8
49
38
65
97
76
13
27
49
49
38
65
97
76
13
27
49
界点
low
high

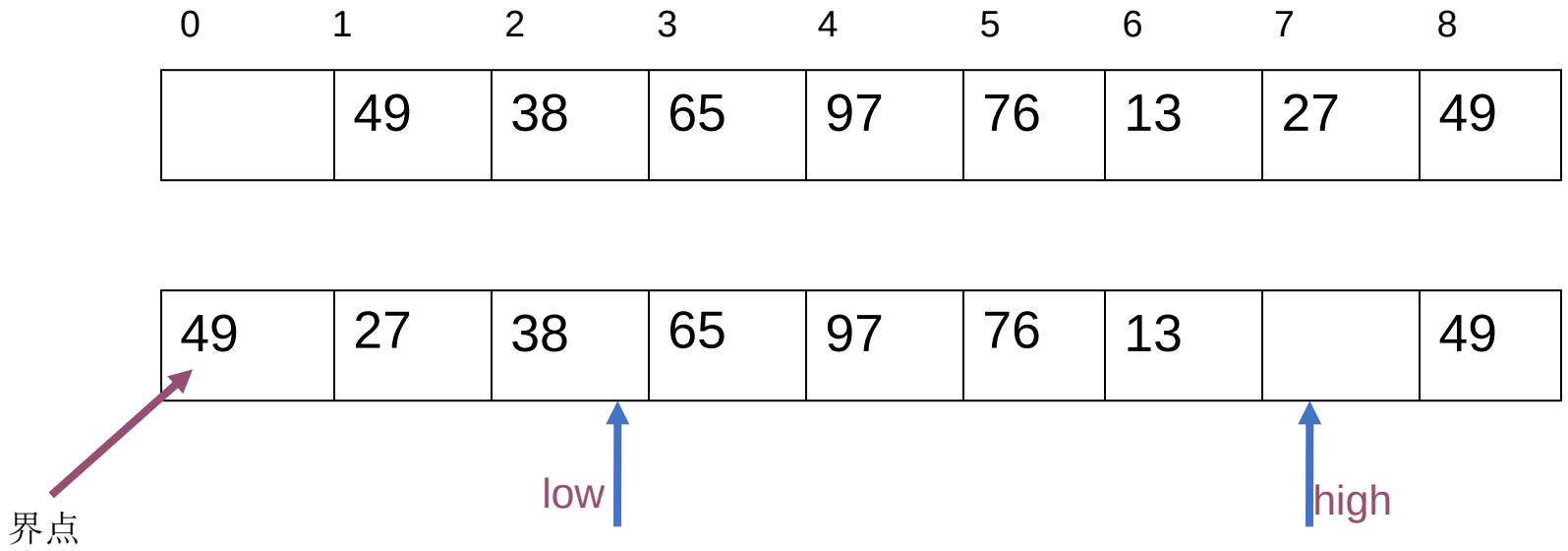
快速排序


0
1
2
3
4
5
6
7
8
49
38
65
97
76
13
27
49
49
38
65
97
76
13
27
49
界点
low
high

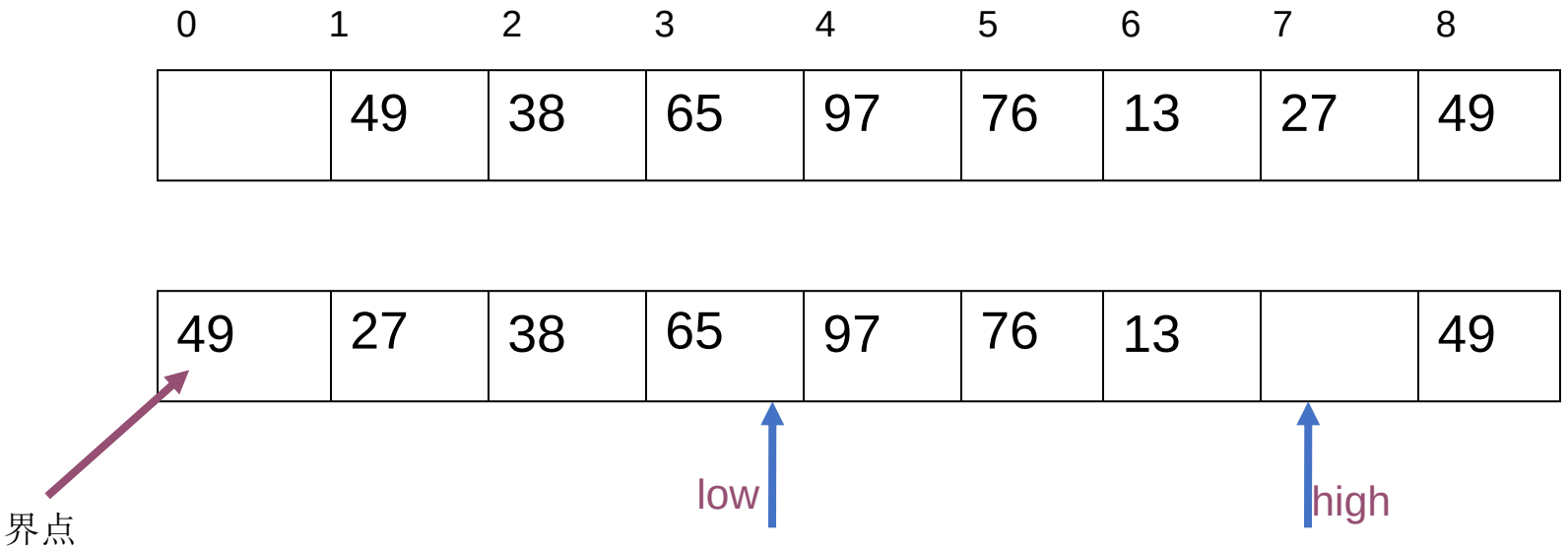
快速排序


0
1
2
3
4
5
6
7
8
49
38
65
97
76
13
27
49
49
27
38
65
97
76
13
49
界点
low
high

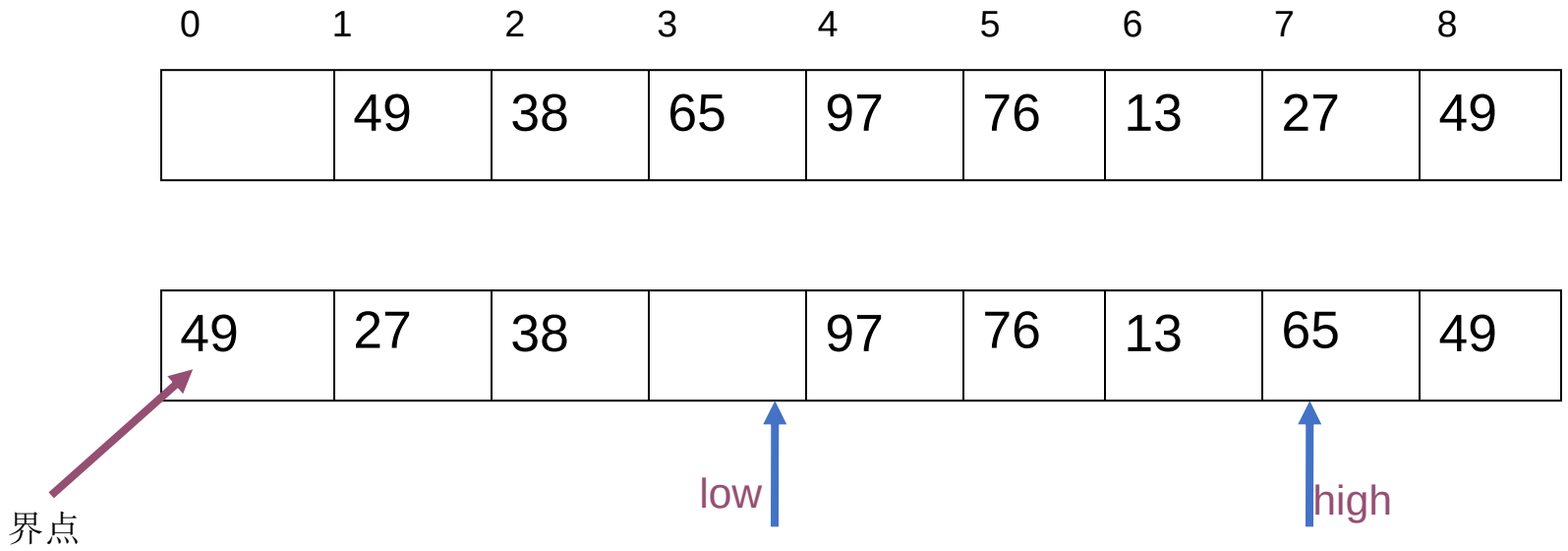
快速排序


0
1
2
3
4
5
6
7
8
49
38
65
97
76
13
27
49
49
27
38
65
97
76
13
49
界点
low
high

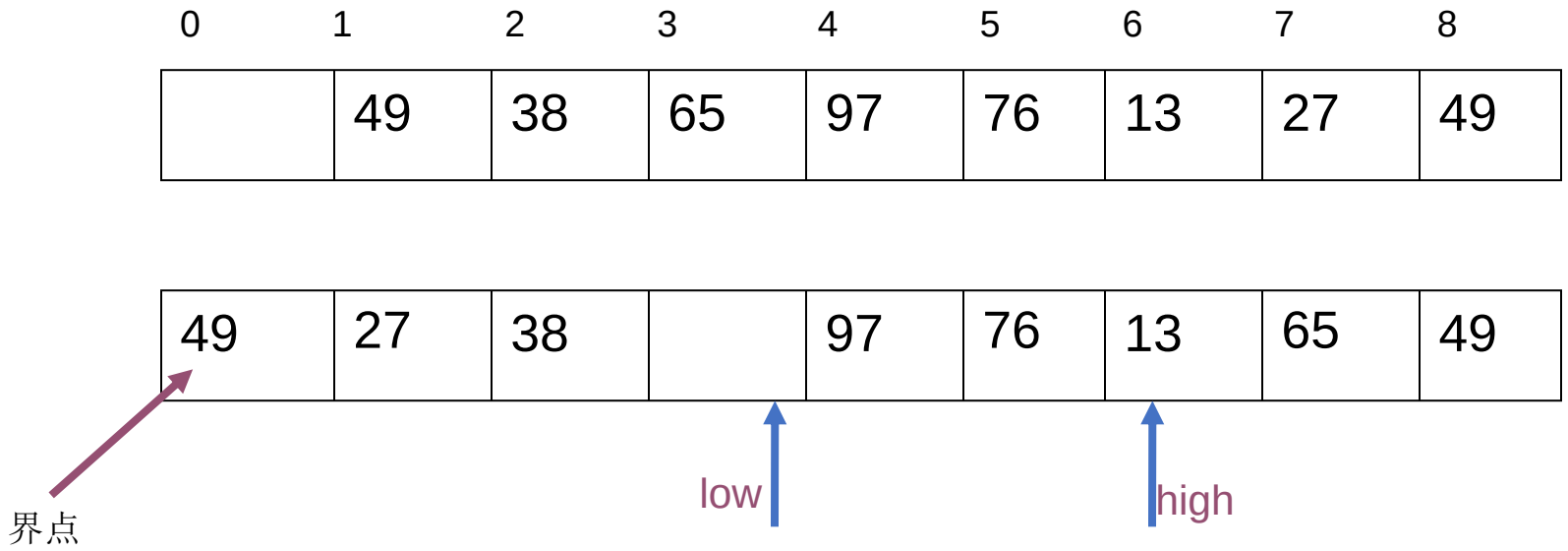
快速排序


0
1
2
3
4
5
6
7
8
49
38
65
97
76
13
27
49
49
27
38
65
97
76
13
49
界点
low
high

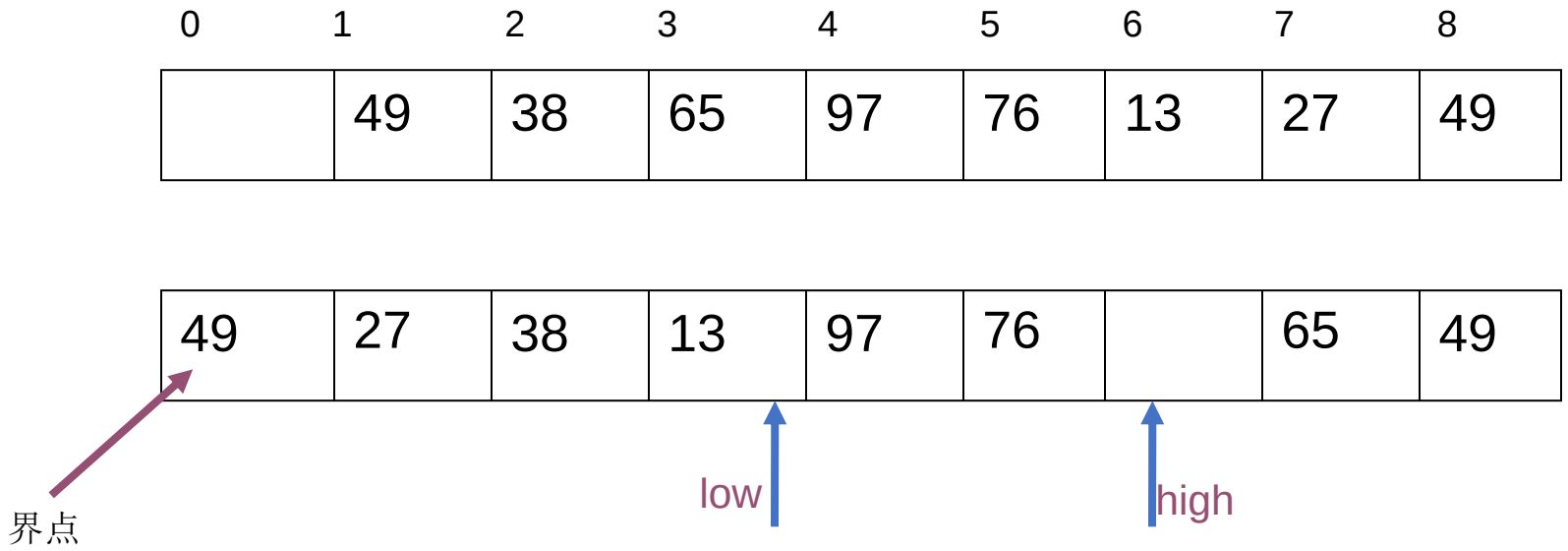
快速排序


0
1
2
3
4
5
6
7
8
49
38
65
97
76
13
27
49
49
27
38
97
76
13
65
49
界点
low
high

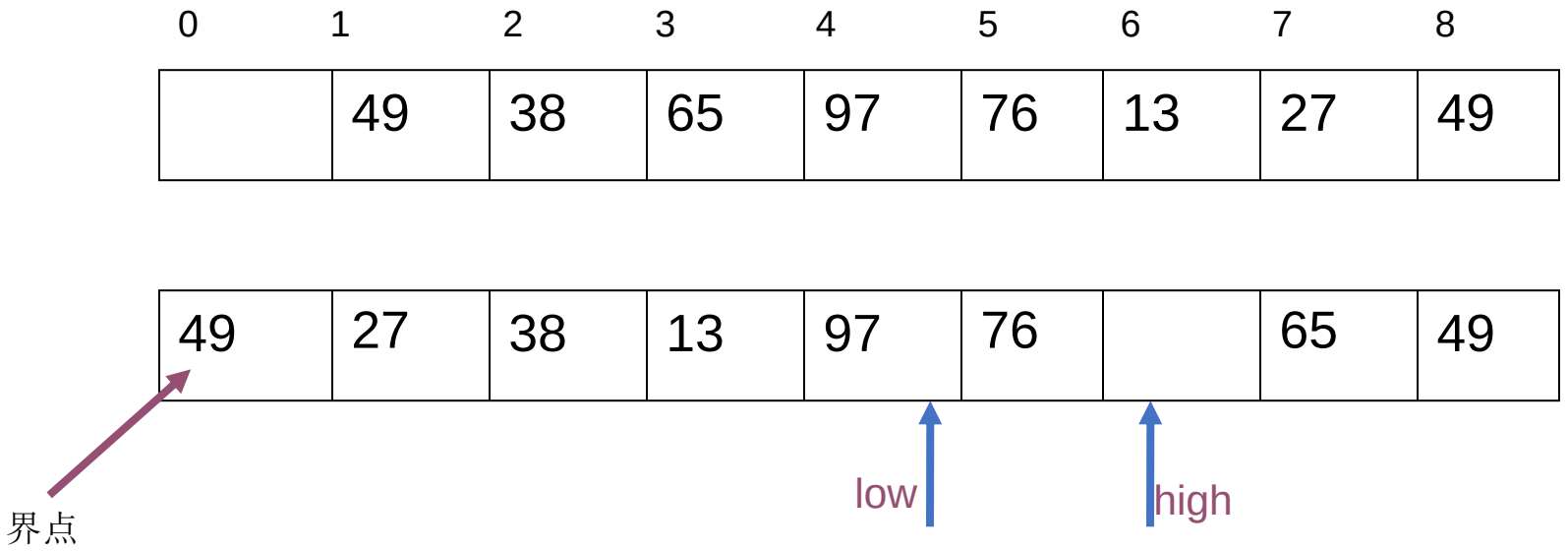
快速排序


0
1
2
3
4
5
6
7
8
49
38
65
97
76
13
27
49
49
27
38
97
76
13
65
49
界点
low
high

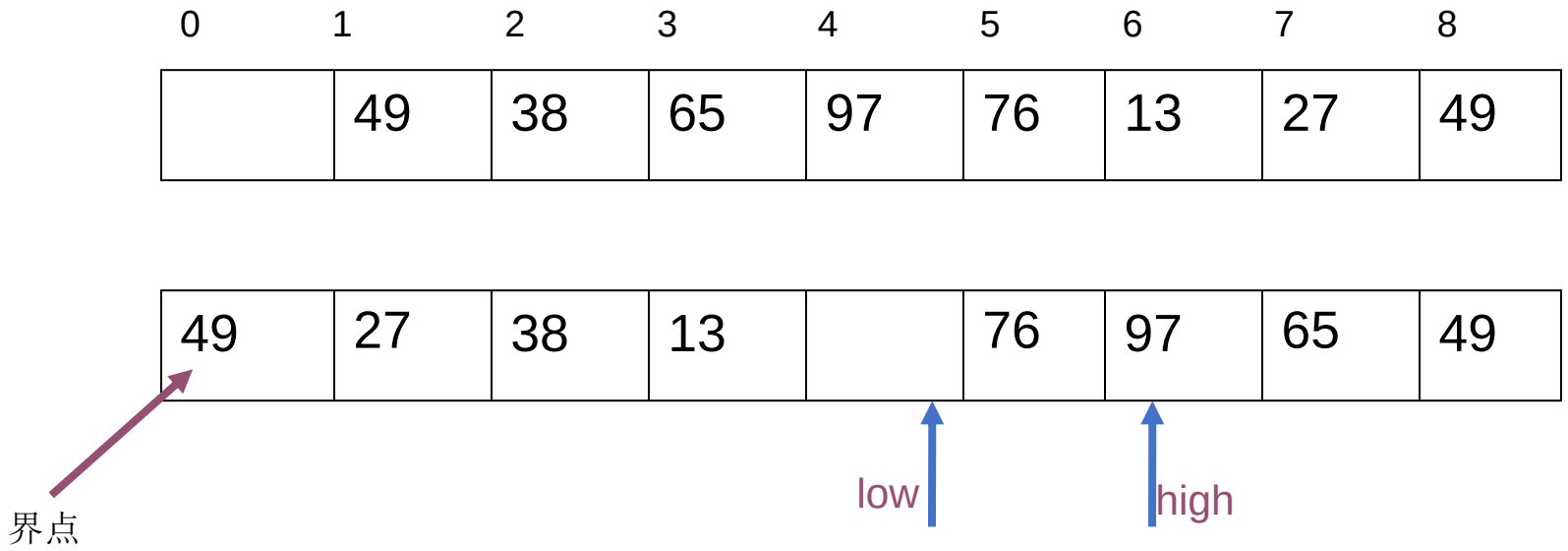
快速排序


0
1
2
3
4
5
6
7
8
49
38
65
97
76
13
27
49
49
27
38
13
97
76
65
49
界点
low
high

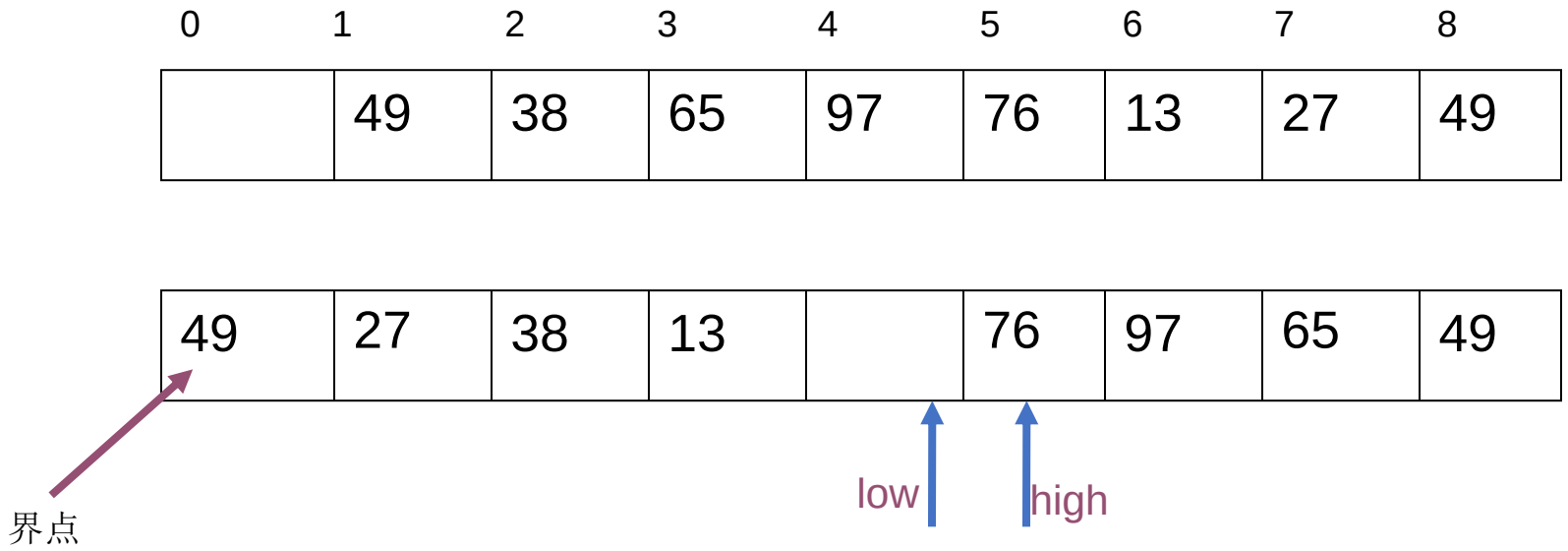
快速排序


0
1
2
3
4
5
6
7
8
49
38
65
97
76
13
27
49
49
27
38
13
97
76
65
49
界点
low
high

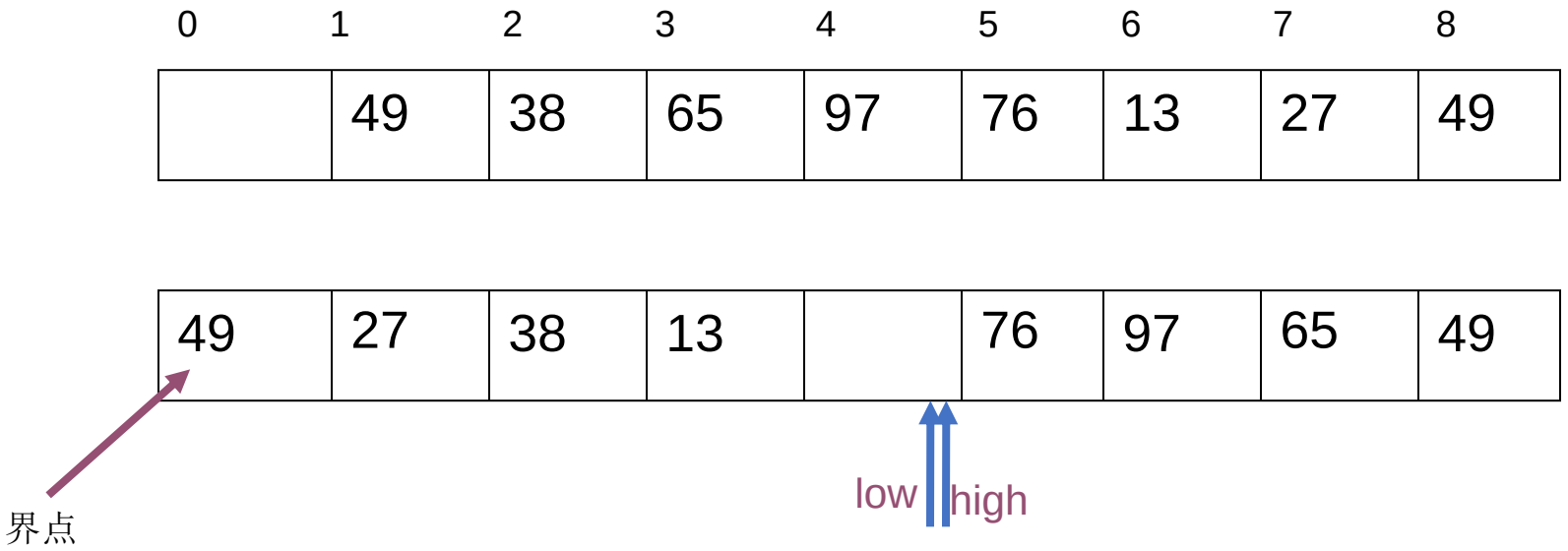
快速排序


0
1
2
3
4
5
6
7
8
49
38
65
97
76
13
27
49
49
27
38
13
76
97
65
49
界点
low
high

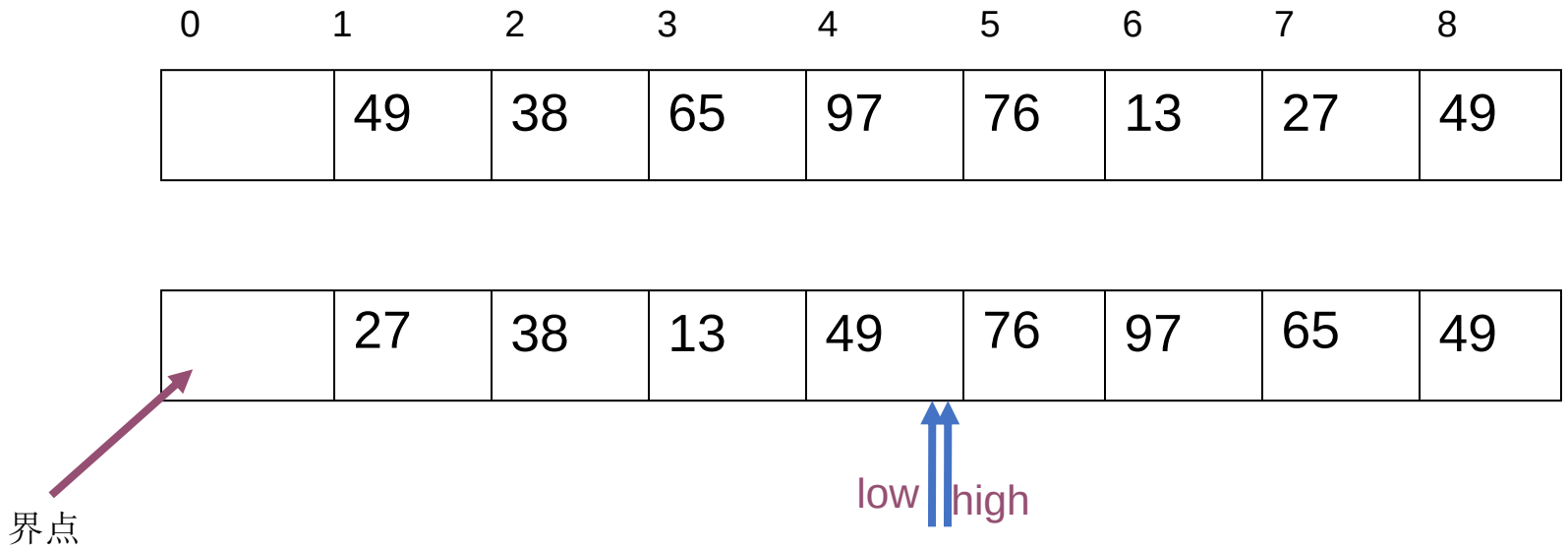
快速排序


0
1
2
3
4
5
6
7
8
49
38
65
97
76
13
27
49
49
27
38
13
76
97
65
49
界点
low
high

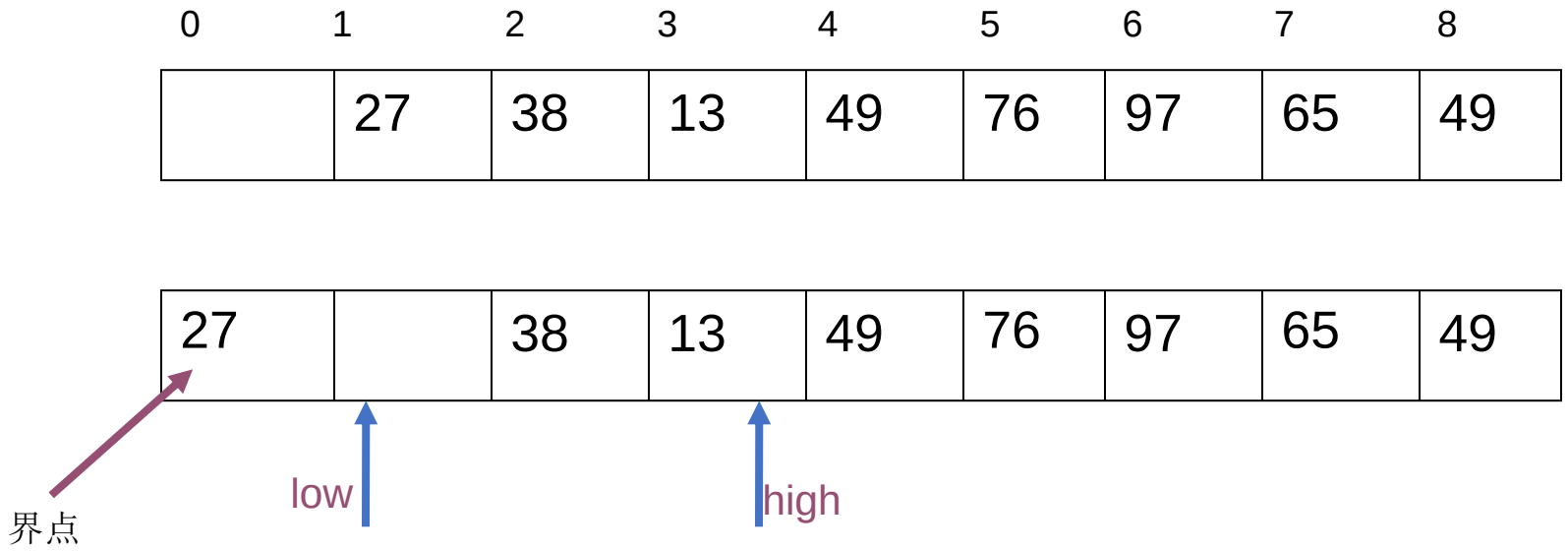
快速排序


0
1
2
3
4
5
6
7
8
49
38
65
97
76
13
27
49
49
27
38
13
76
97
65
49
界点
low
high

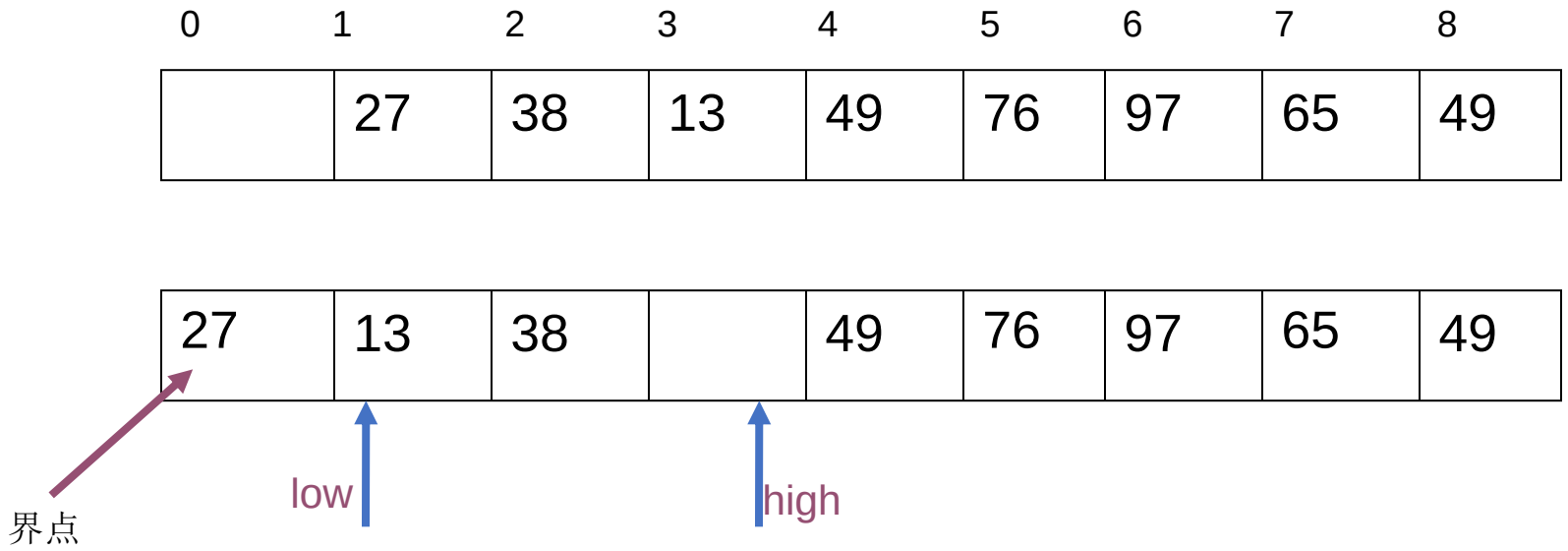
快速排序


0
1
2
3
4
5
6
7
8
49
38
65
97
76
13
27
49
27
38
13
49
76
97
65
49
界点
low
high

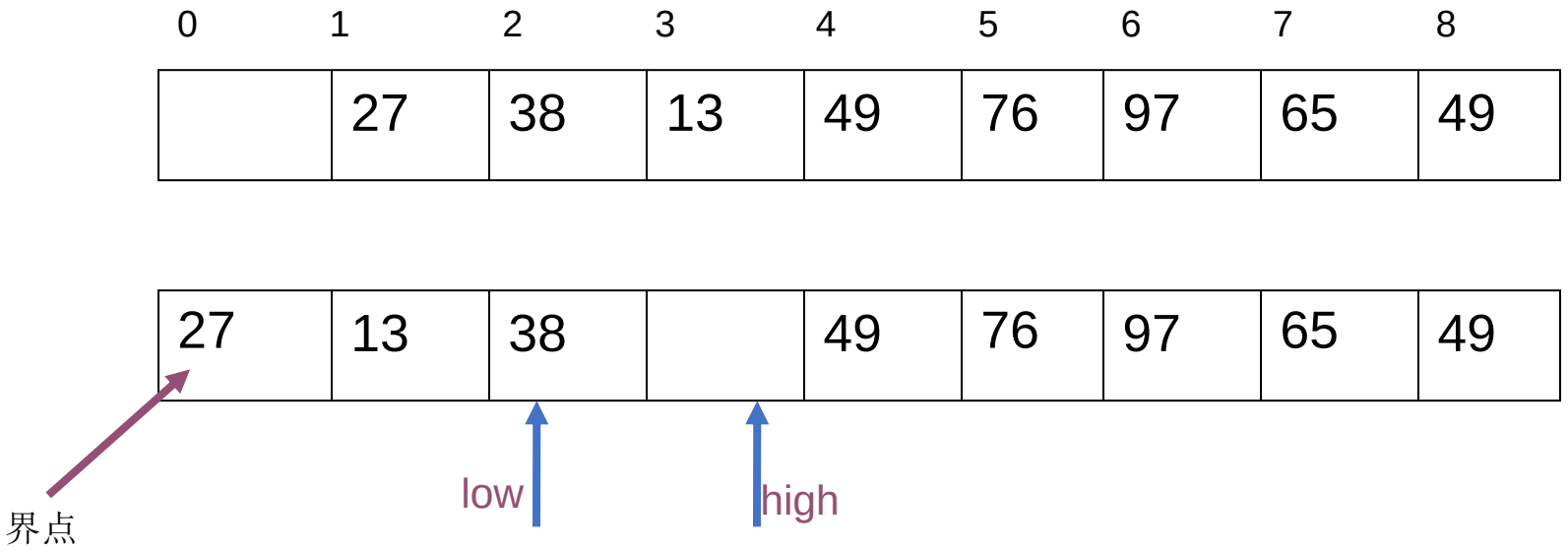
快速排序


0
1
2
3
4
5
6
7
8
27
38
13
49
76
97
65
49
27
38
13
49
76
97
65
49
界点
low
high

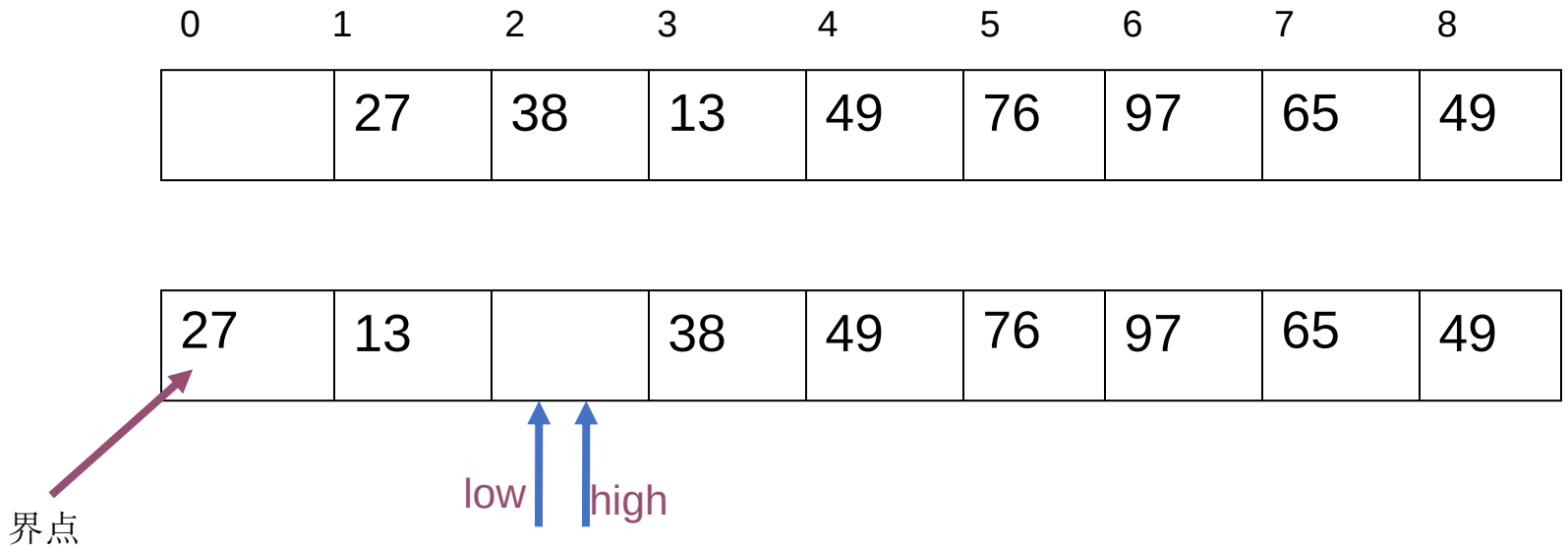
快速排序


0
1
2
3
4
5
6
7
8
27
38
13
49
76
97
65
49
27
13
38
49
76
97
65
49
界点
low
high

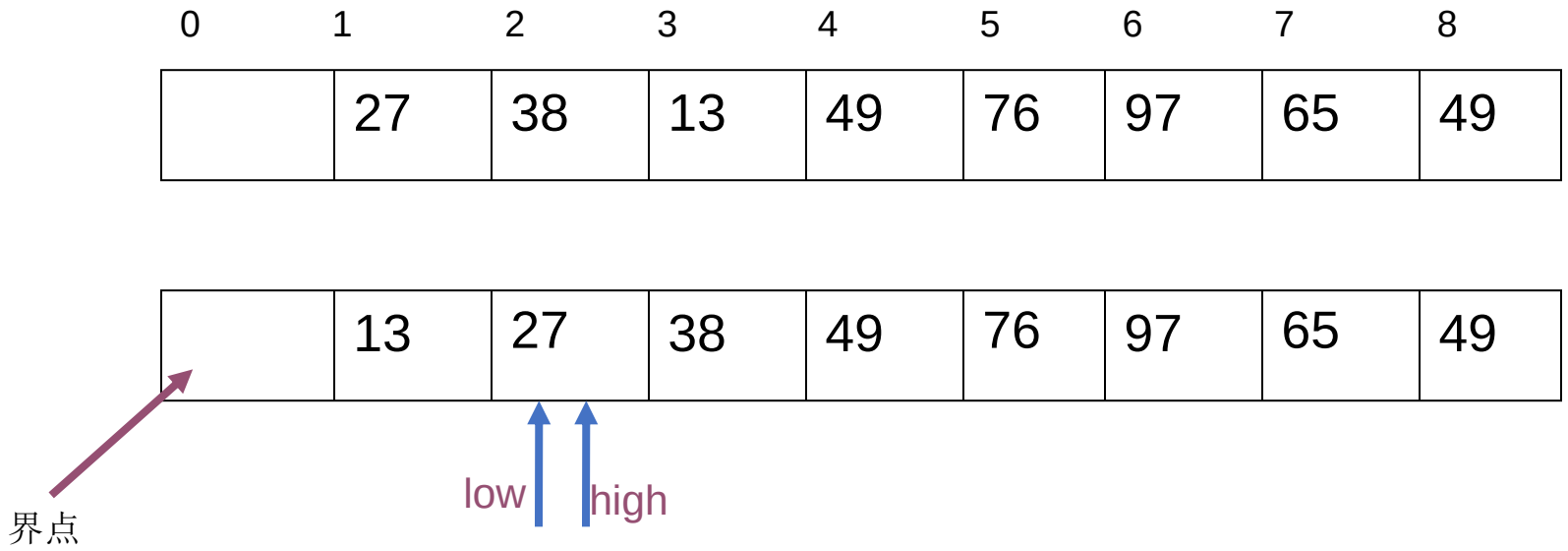
快速排序


0
1
2
3
4
5
6
7
8
27
38
13
49
76
97
65
49
27
13
38
49
76
97
65
49
界点
low
high

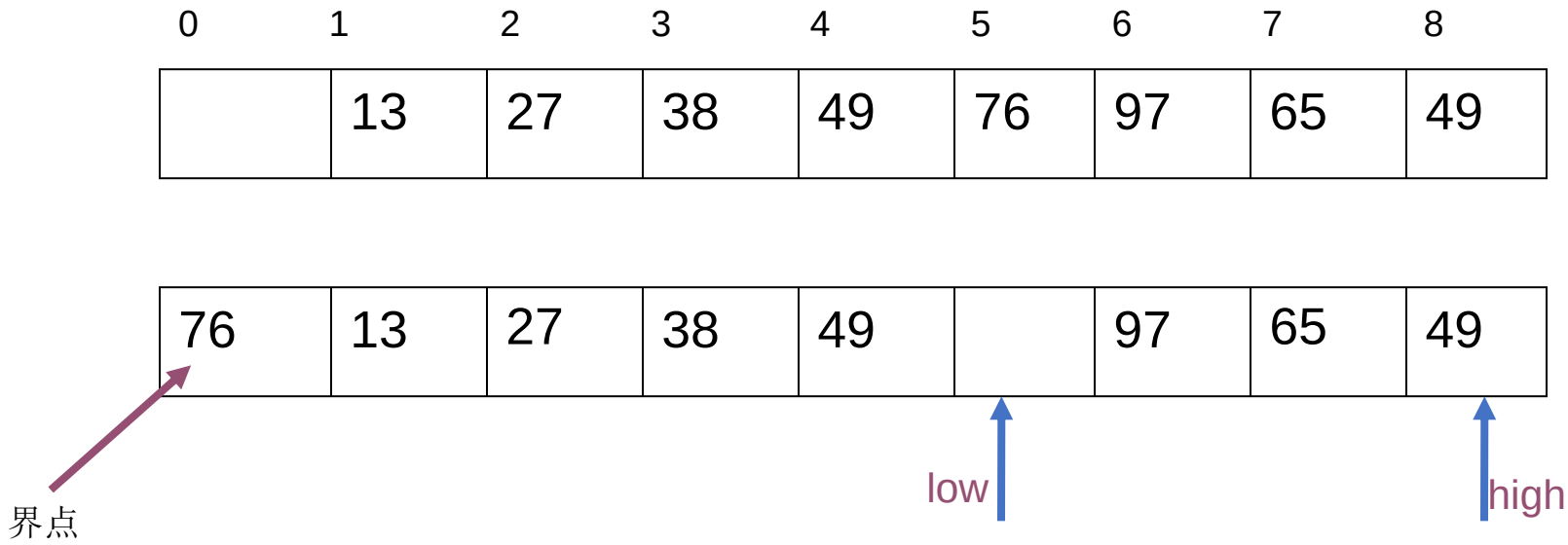
快速排序


0
1
2
3
4
5
6
7
8
27
38
13
49
76
97
65
49
27
13
38
49
76
97
65
49
界点
low
high

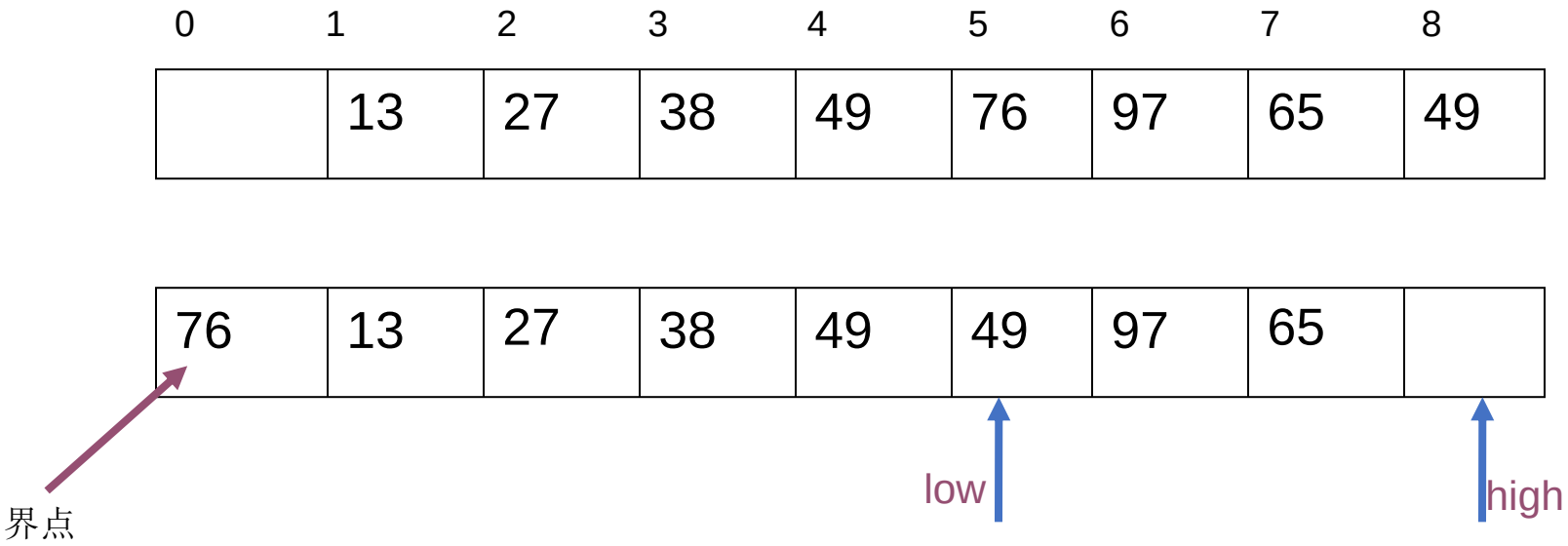
快速排序


0
1
2
3
4
5
6
7
8
27
38
13
49
76
97
65
49
13
27
38
49
76
97
65
49
界点
low
high

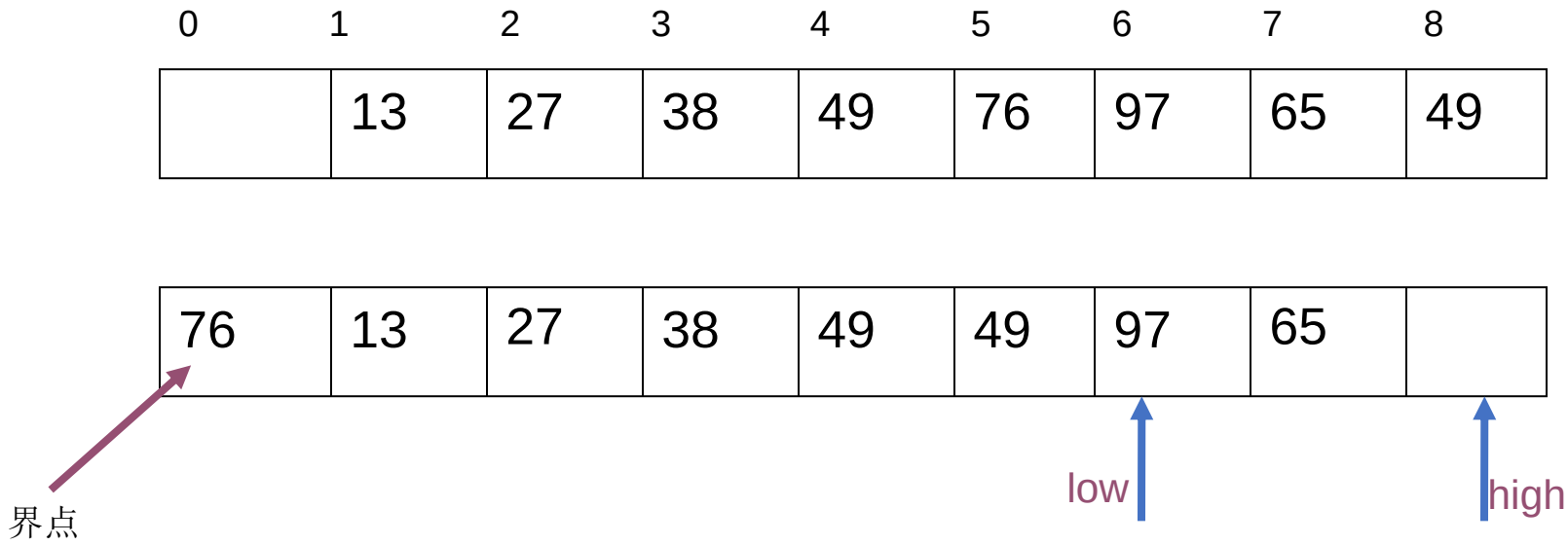
快速排序


0
1
2
3
4
5
6
7
8
13
27
38
49
76
97
65
49
76
13
27
38
49
97
65
49
界点
low
high

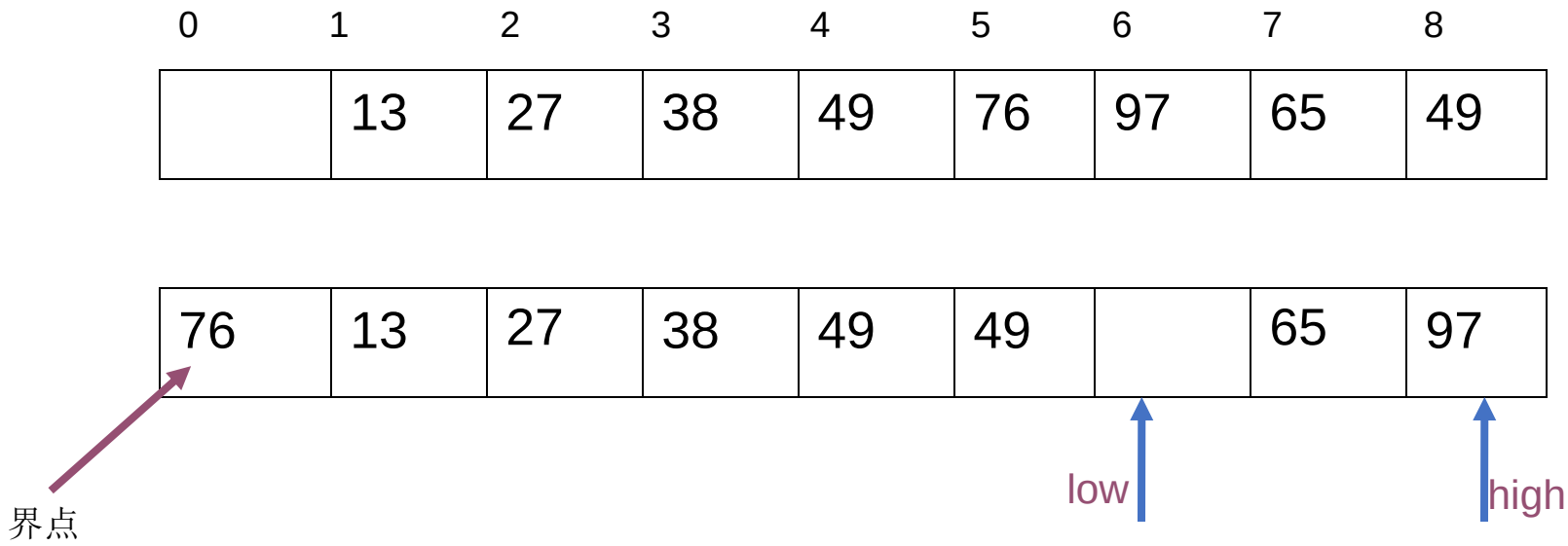
快速排序


0
1
2
3
4
5
6
7
8
13
27
38
49
76
97
65
49
76
13
27
38
49
49
97
65
界点
low
high

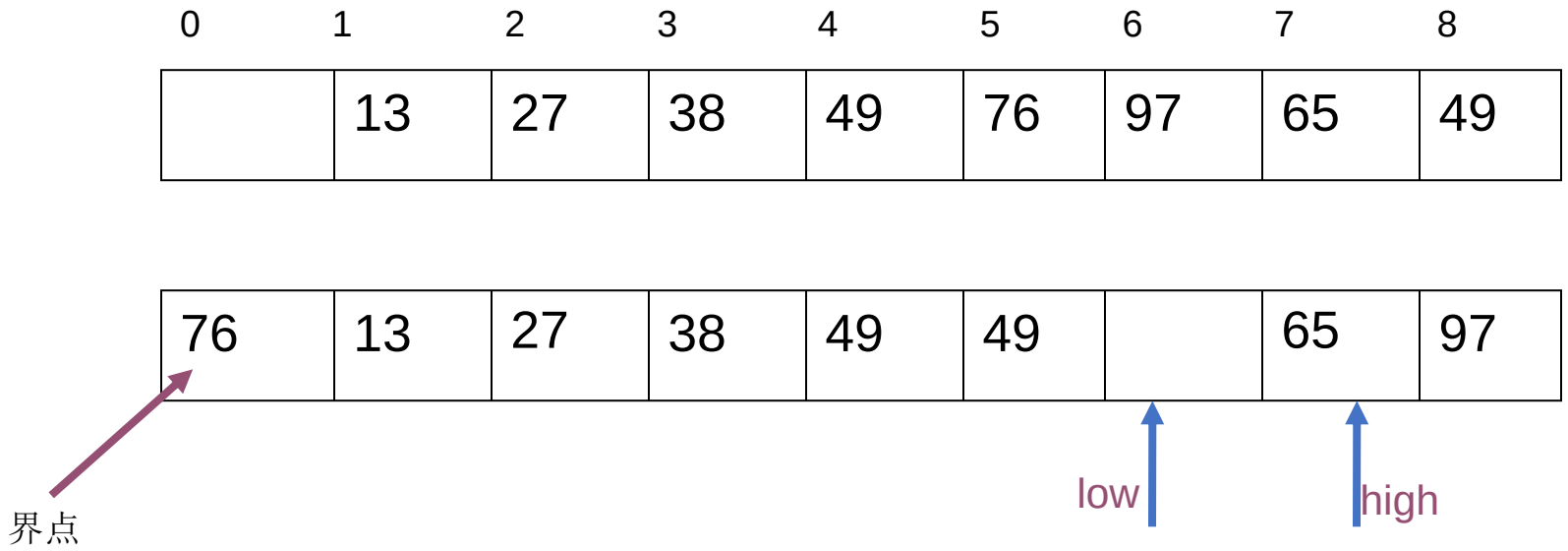
快速排序


0
1
2
3
4
5
6
7
8
13
27
38
49
76
97
65
49
76
13
27
38
49
49
97
65
界点
low
high

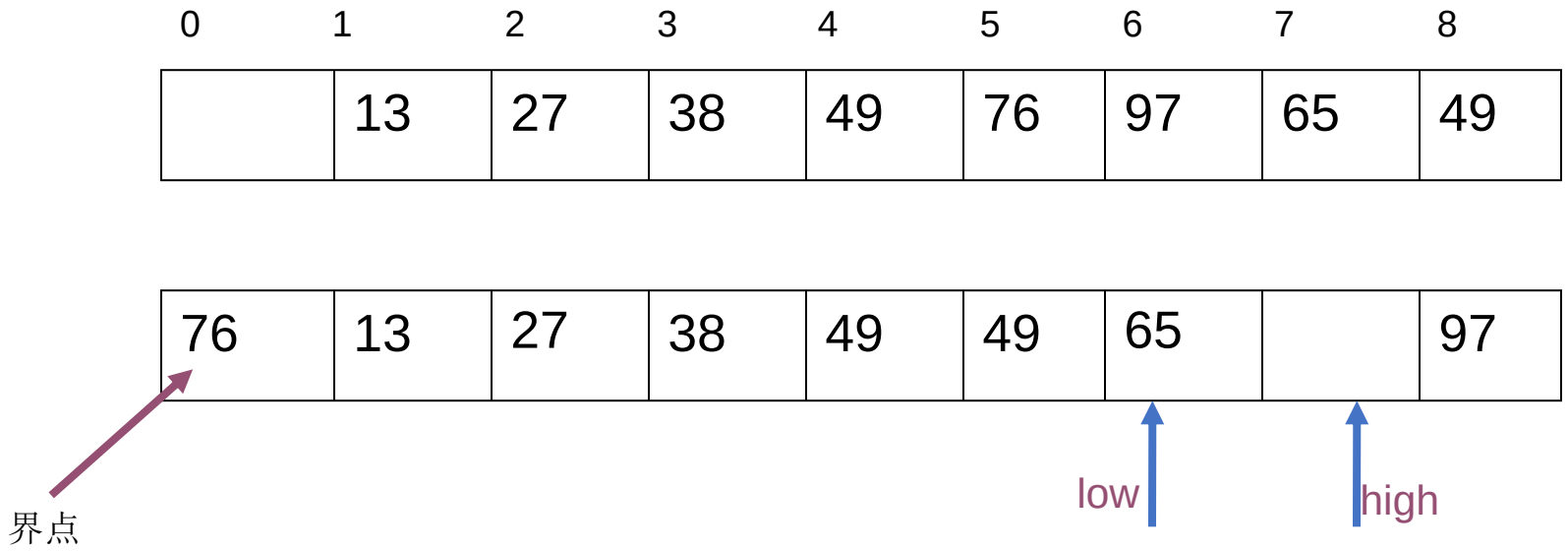
快速排序

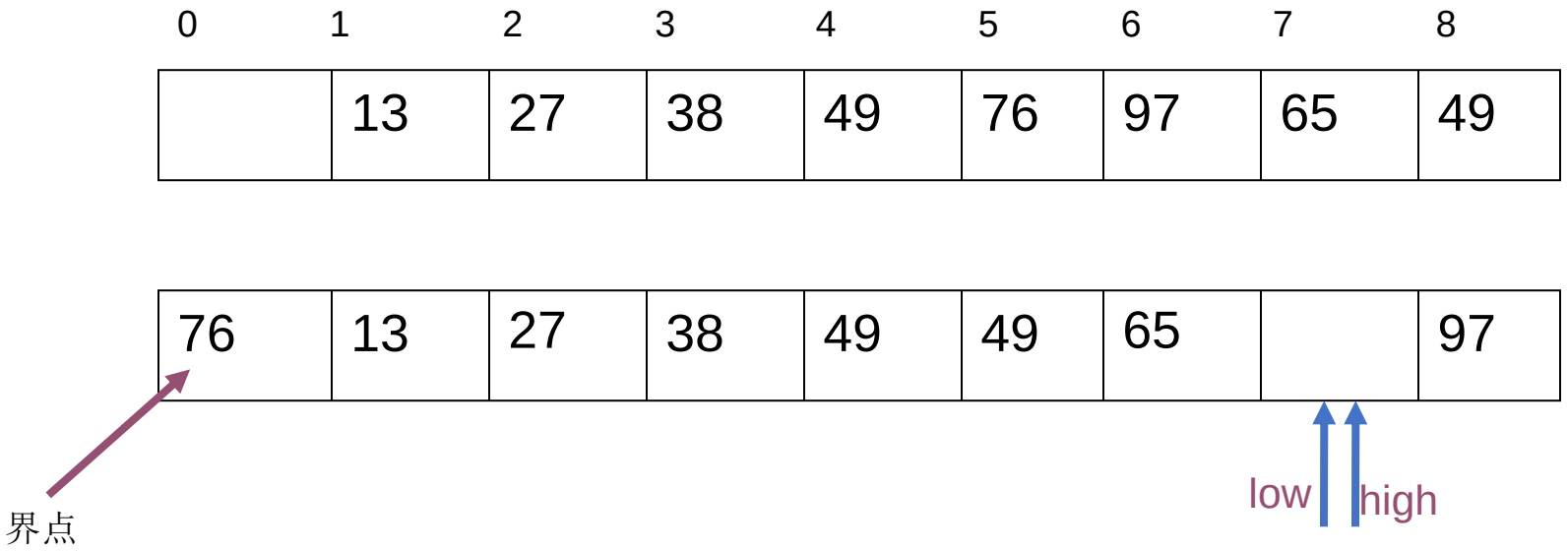

0
1
2
3
4
5
6
7
8
13
27
38
49
76
97
65
49
76
13
27
38
49
49
65
97
界点
low
high

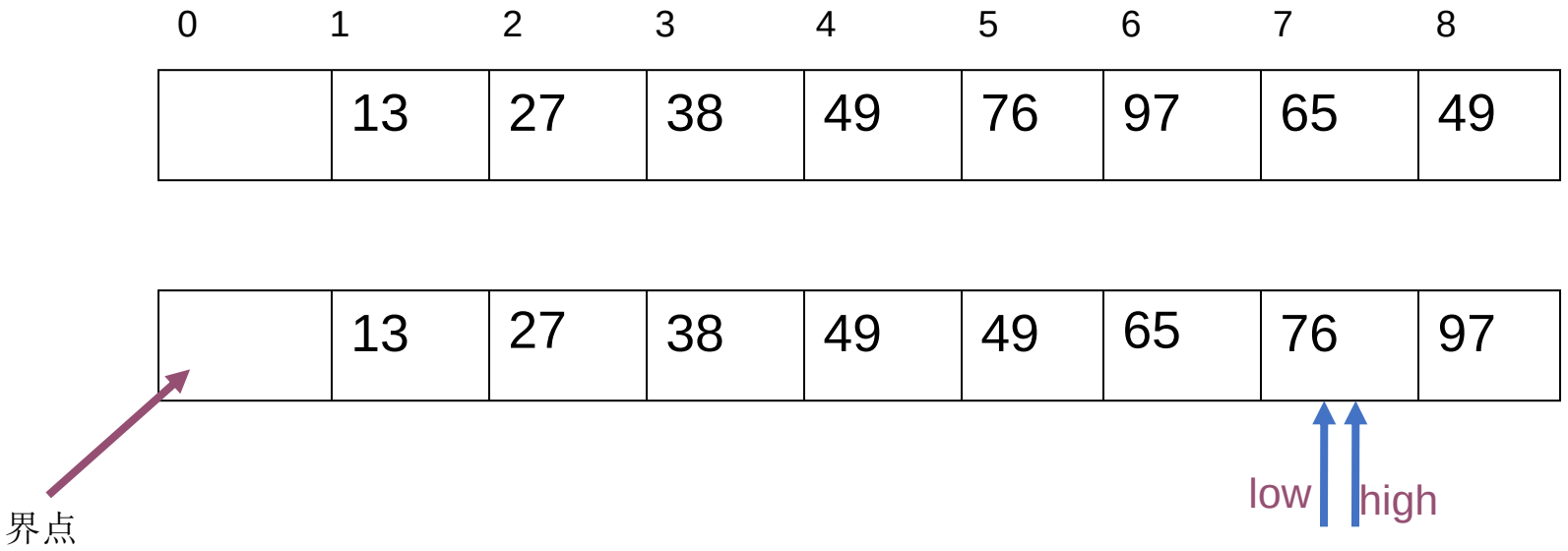
快速排序


0
1
2
3
4
5
6
7
8
13
27
38
49
76
97
65
49
76
13
27
38
49
49
65
97
界点
low
high

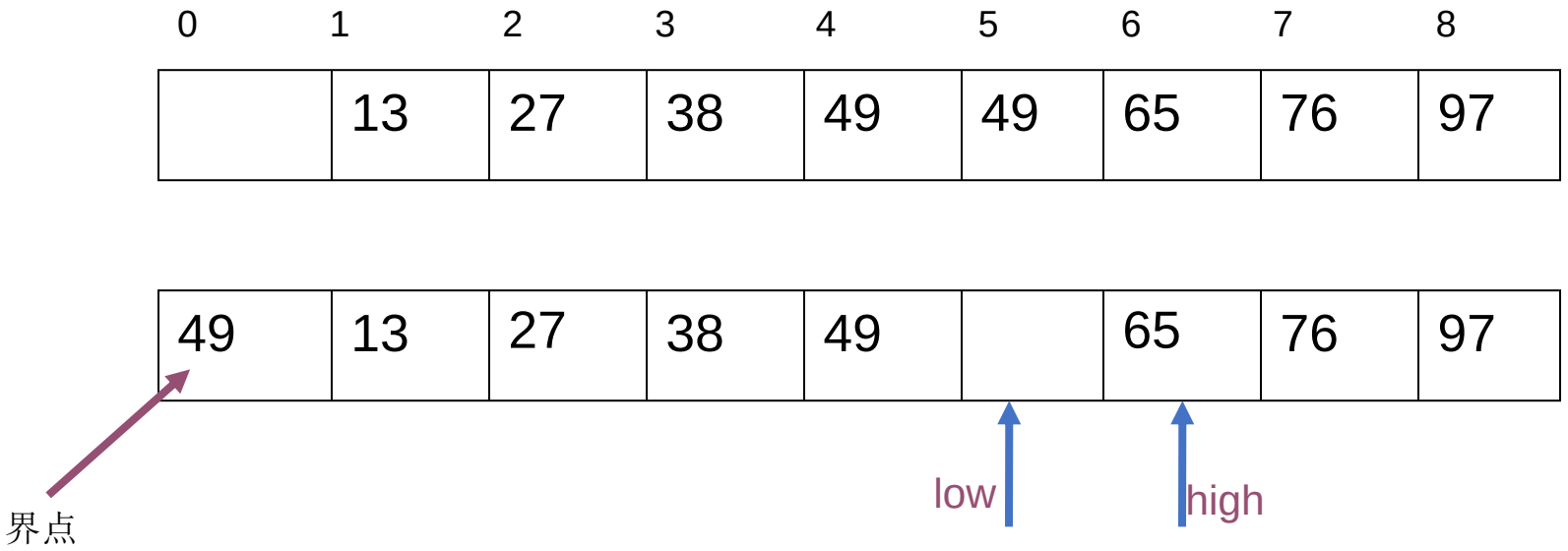
快速排序


0
1
2
3
4
5
6
7
8
13
27
38
49
76
97
65
49
76
13
27
38
49
49
65
97
界点
low
high

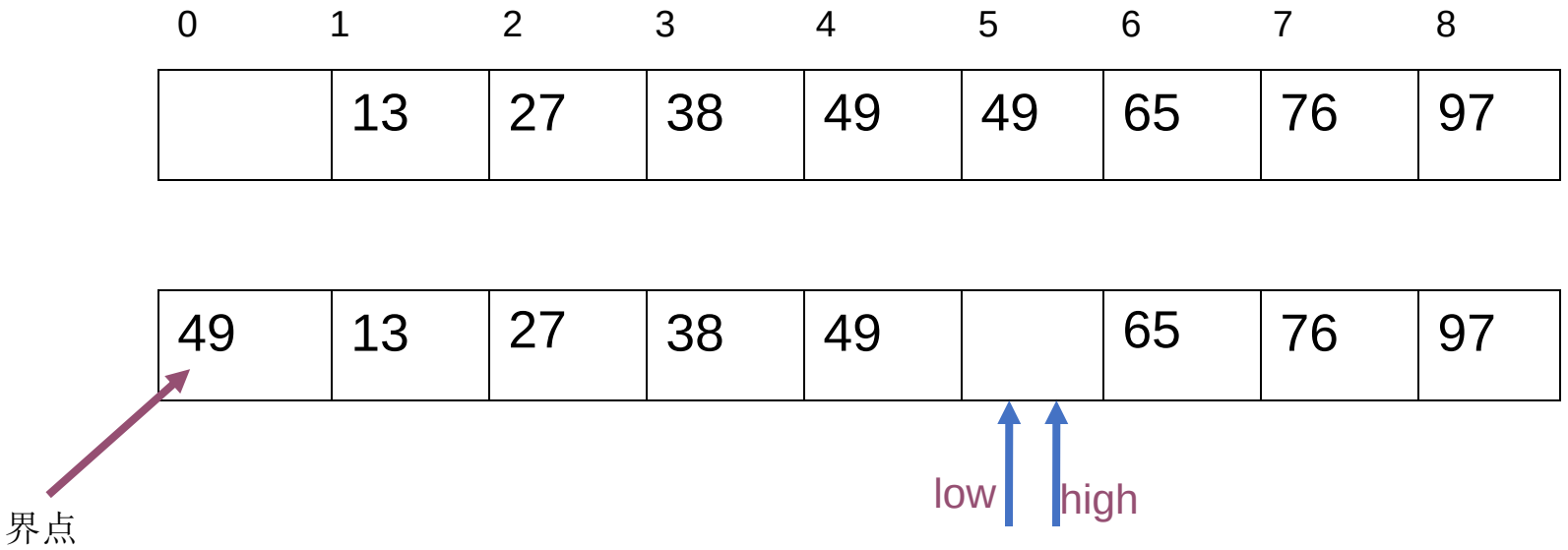

0
1
2
3
4
5
6
7
8
13
27
38
49
76
97
65
49
76
13
27
38
49
49
65
97
界点
low
high


0
1
2
3
4
5
6
7
8
13
27
38
49
76
97
65
49
13
27
38
49
49
65
76
97
界点
low
high

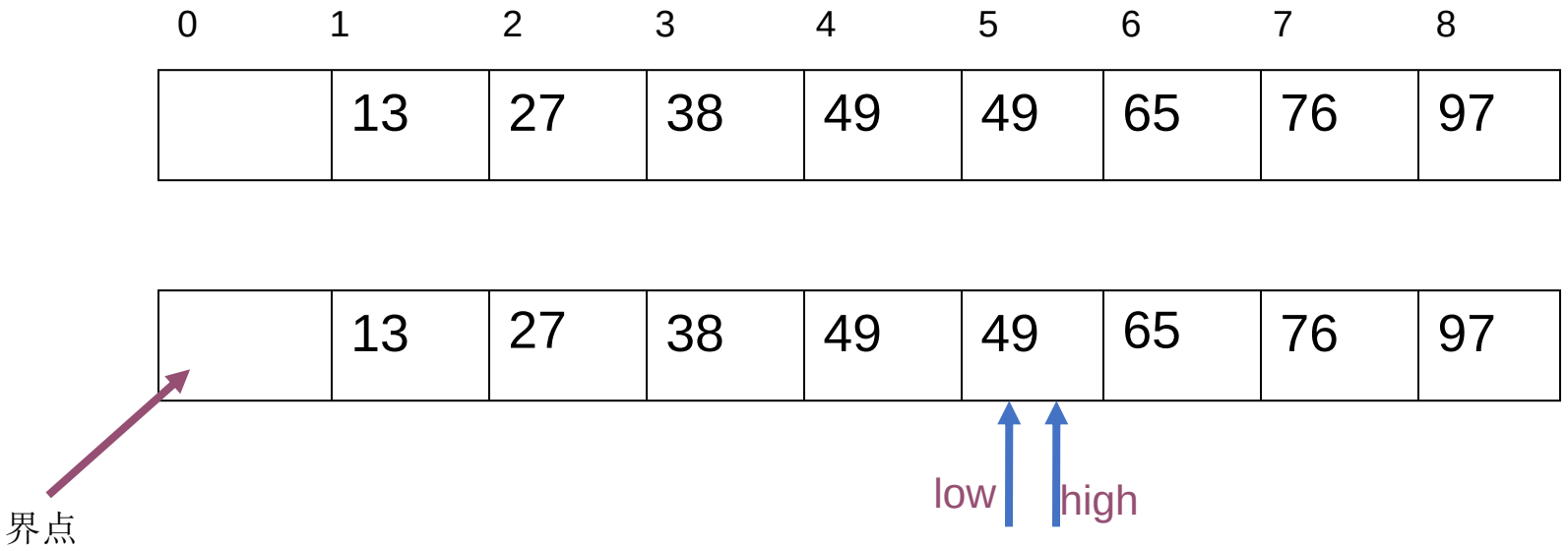
快速排序


0
1
2
3
4
5
6
7
8
13
27
38
49
49
65
76
97
49
13
27
38
49
65
76
97
界点
low
high

快速排序


0
1
2
3
4
5
6
7
8
13
27
38
49
49
65
76
97
49
13
27
38
49
65
76
97
界点
low
high

快速排序


0
1
2
3
4
5
6
7
8
13
27
38
49
49
65
76
97
13
27
38
49
49
65
76
97
界点
low
high

快速排序

①每一趟的子表的形成是采用从两头向中间交替式逼近法；

②由于每趟中对各子表的操作都相似，可采用递归算法。

2020年6月22日

```
void main ( )
 {   QSort ( L, 1, L.length ); }
```

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

```
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;
}
```

算法分析

- 可以证明，平均计算时间是$O(n\log_2 n)$。
- 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。
- 快速排序是递归的，需要有一个栈存放每层递归调用时参数（新的low和high）。
- 最大递归调用层次数与递归树的深度一致，因此，要求存储开销为 $O(\log_2 n)$。

算法分析

- 最好：划分后，左侧右侧子序列的长度相同
- 最坏：从小到大排好序，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，必须经过 n-1 趟才能把所有对象定位，而且第 i 趟需要经过 n-i 次关键码比较才能找到第 i 个对象的安放位置

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

算法分析

时间效率：$O(n\log_2 n)$ —每趟确定的元素呈指数增加

空间效率：$O(\log_2 n)$ —递归要用到栈空间

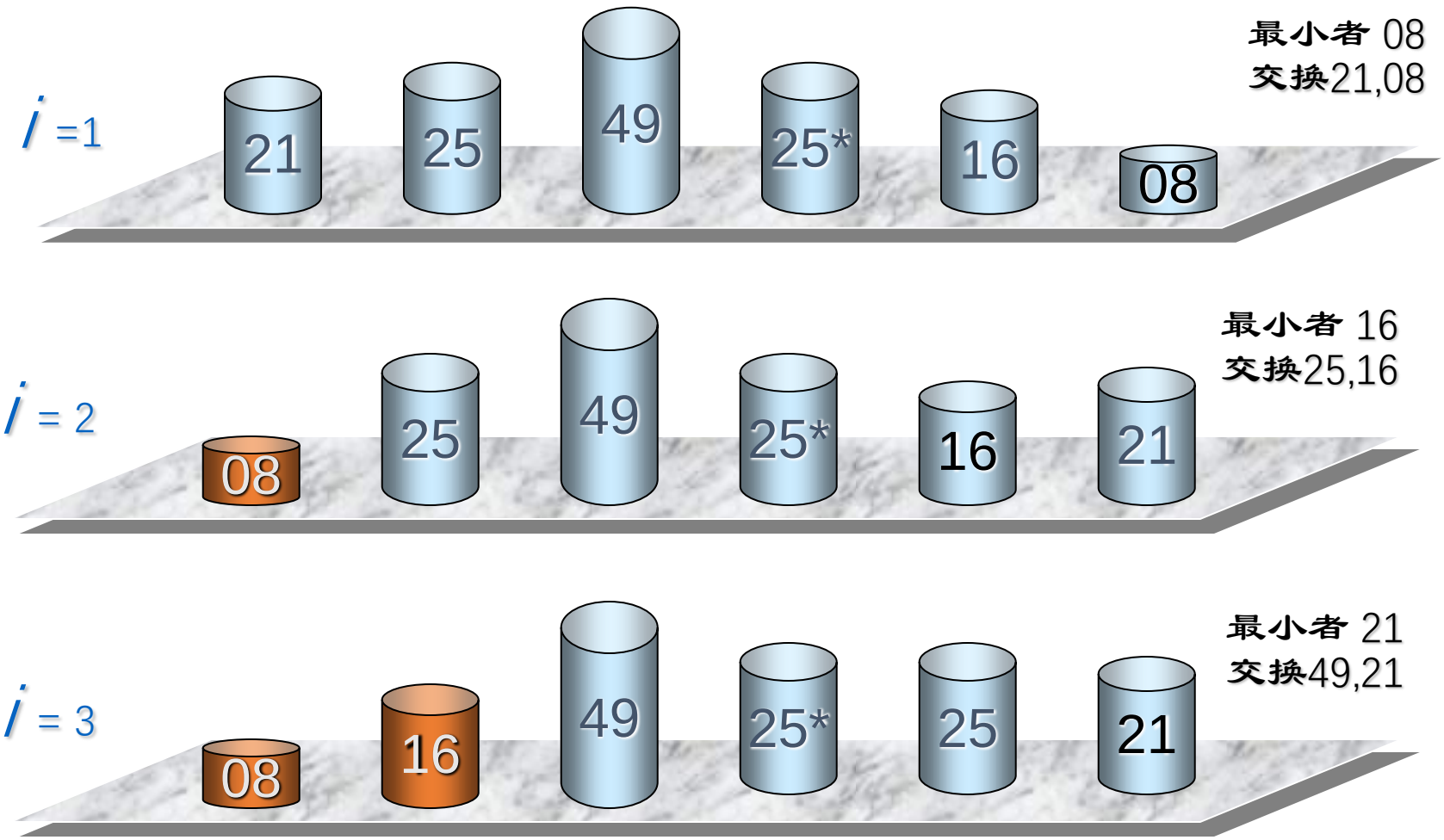
稳 定 性： 不稳定 —可选任一元素为支点。

8.4 选择排序

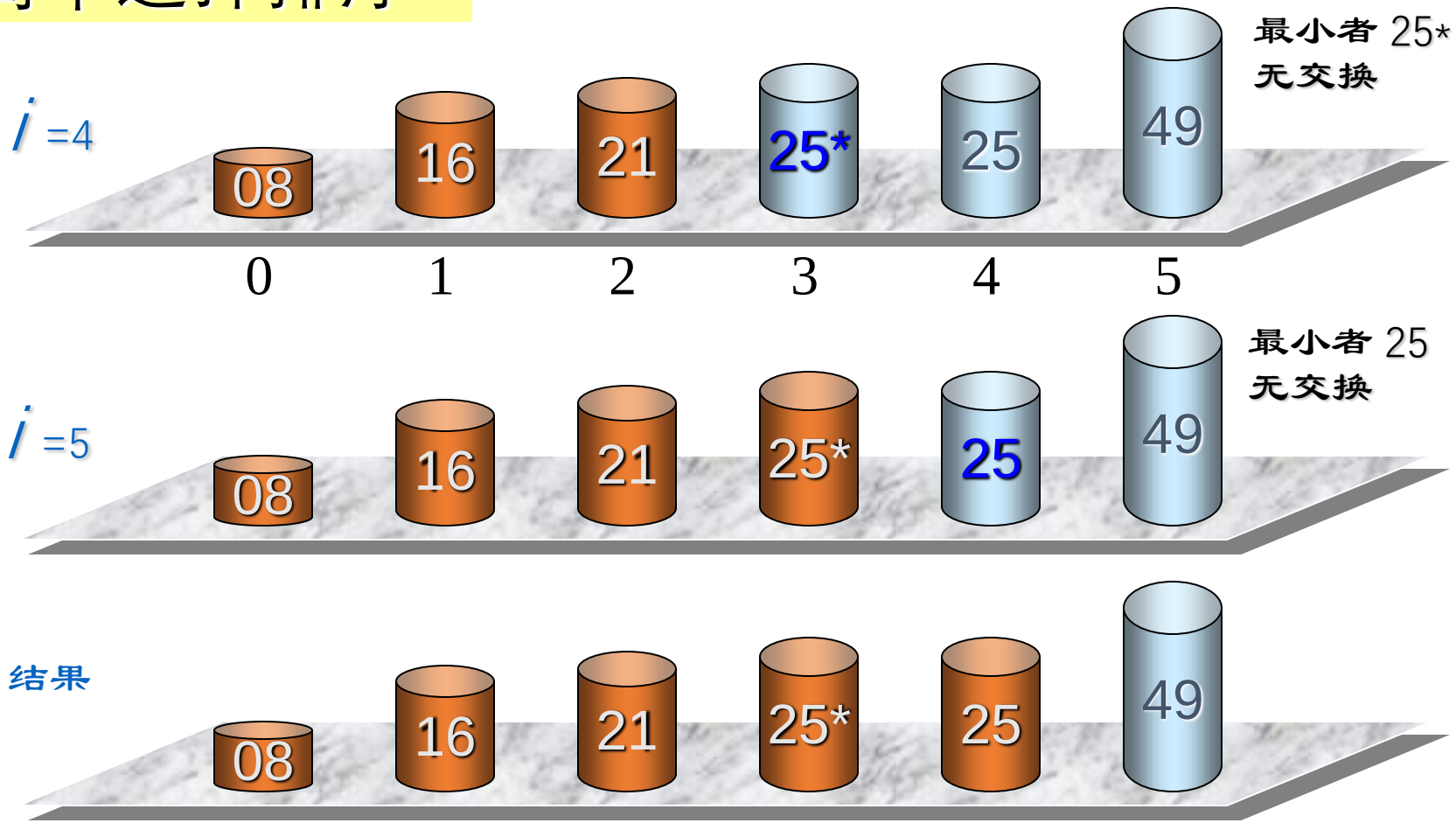
基本思想:

每一趟在后面 n-i +1个中选出关键码最小的对象, 作为有序序列的第 i 个记录

简单选择排序


i =1
21
25
49
25*
16
08
最小者 08
交换21,08
i = 2
08
25
49
25*
16
21
最小者 16
交换25,16
i = 3
08
16
49
25*
25
21
最小者 21
交换49,21

简单选择排序



各趟排序后的结果

简单选择排序

```
void SelectSort(SqList &K)
 {
    for (i=1; i<L.length; ++i)
    { //在L.r[i..L.length] 中选择key最小的记录
        k=i;
        for( j=i+1;j<=L.length ; j++)
            if ( L.r[j].key <L.r[k].key) k=j;
        if(k!=i)L.r[i]←→L.r[k];
    }
}
```

算法分析

移动次数

最好情况：0

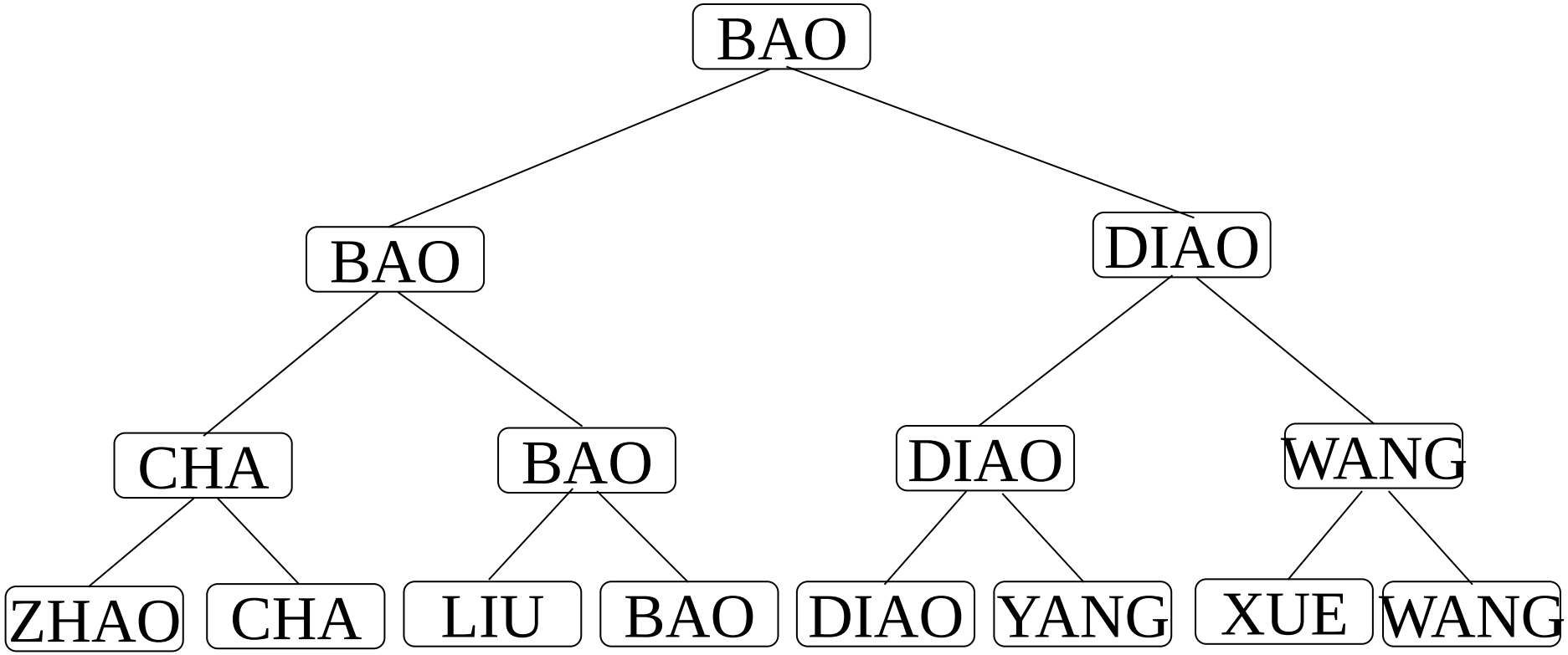
最坏情况：3(n-1)

比较次数：

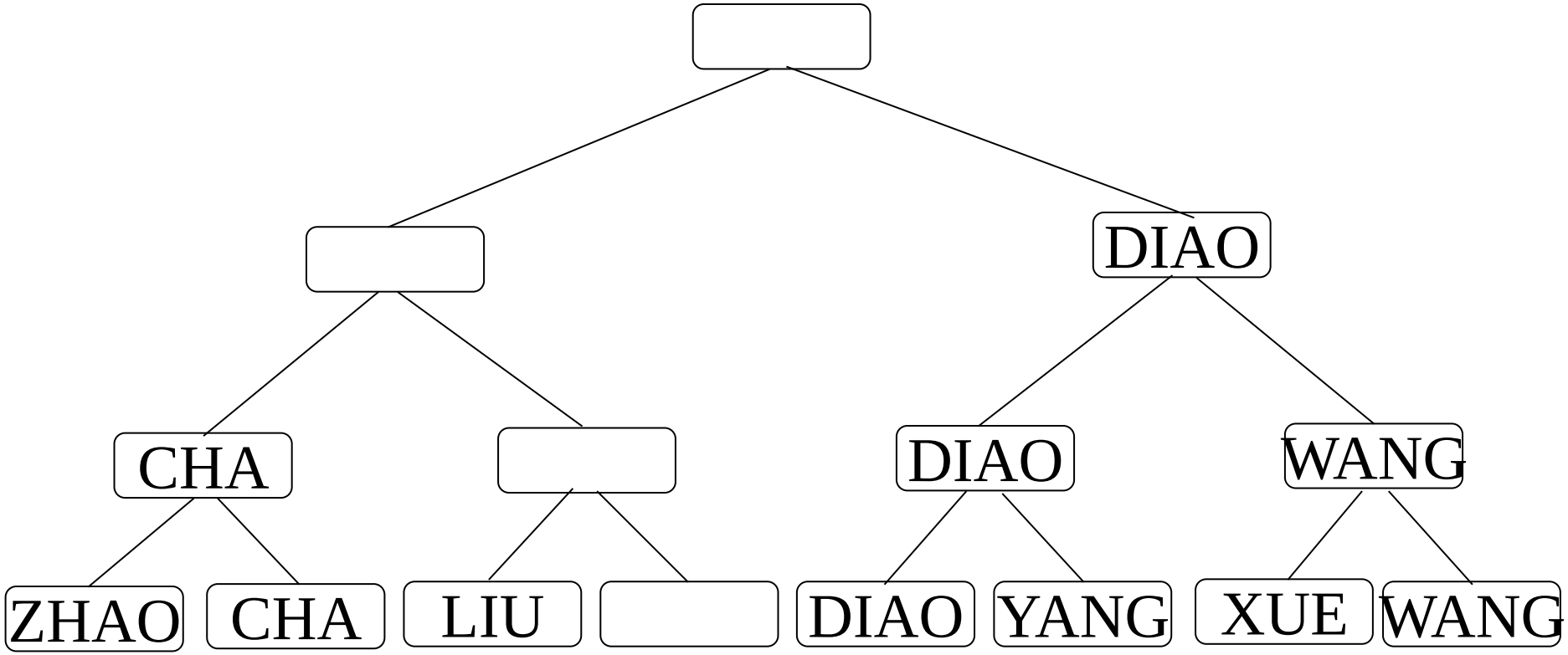
$$\sum_{i=1}^{n-1}(n-i)=\frac{1}{2}(n^2-n)$$

时间复杂度：$O(n^2)$
空间复杂度：$O(1)$
稳定

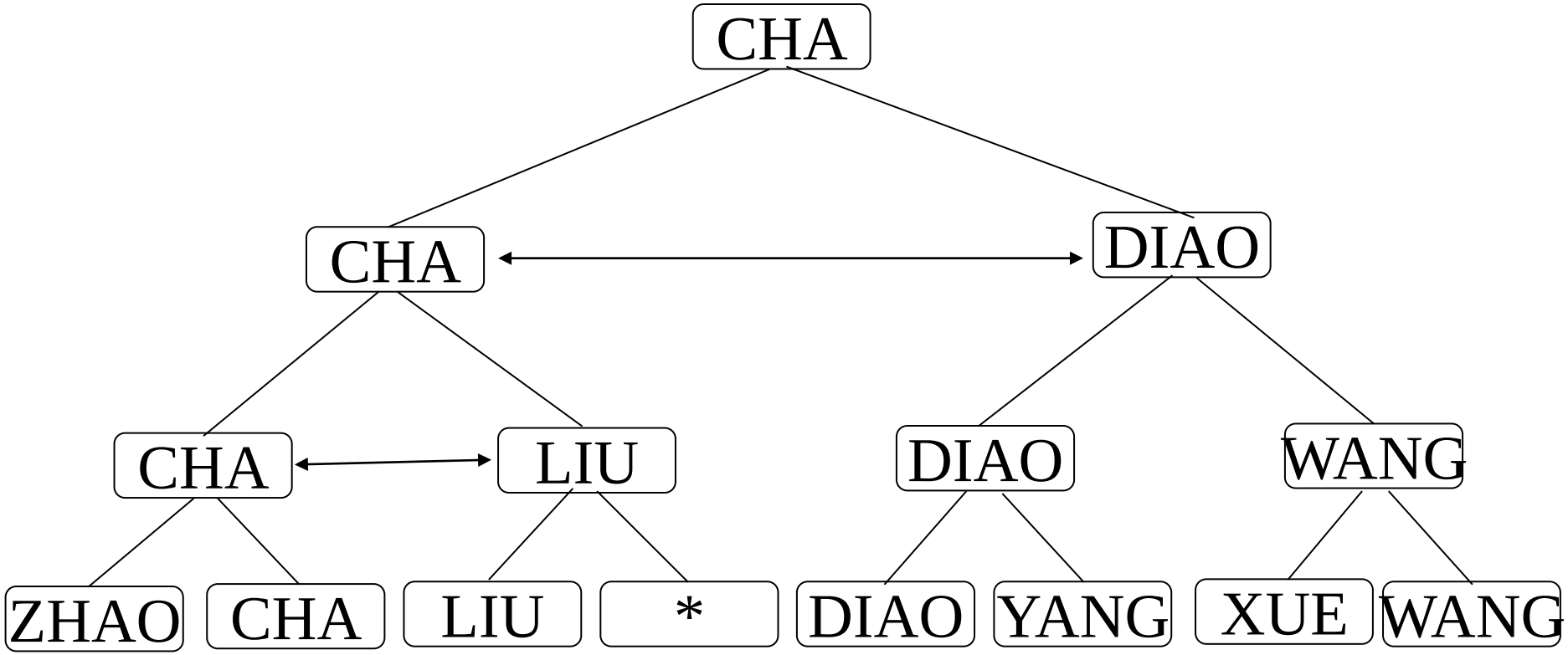
树形选择排序


BAO
BAO
DIAO
CHA
BAO
DIAO
WANG
ZHAO
CHA
LIU
BAO
DIAO
YANG
XUE
WANG

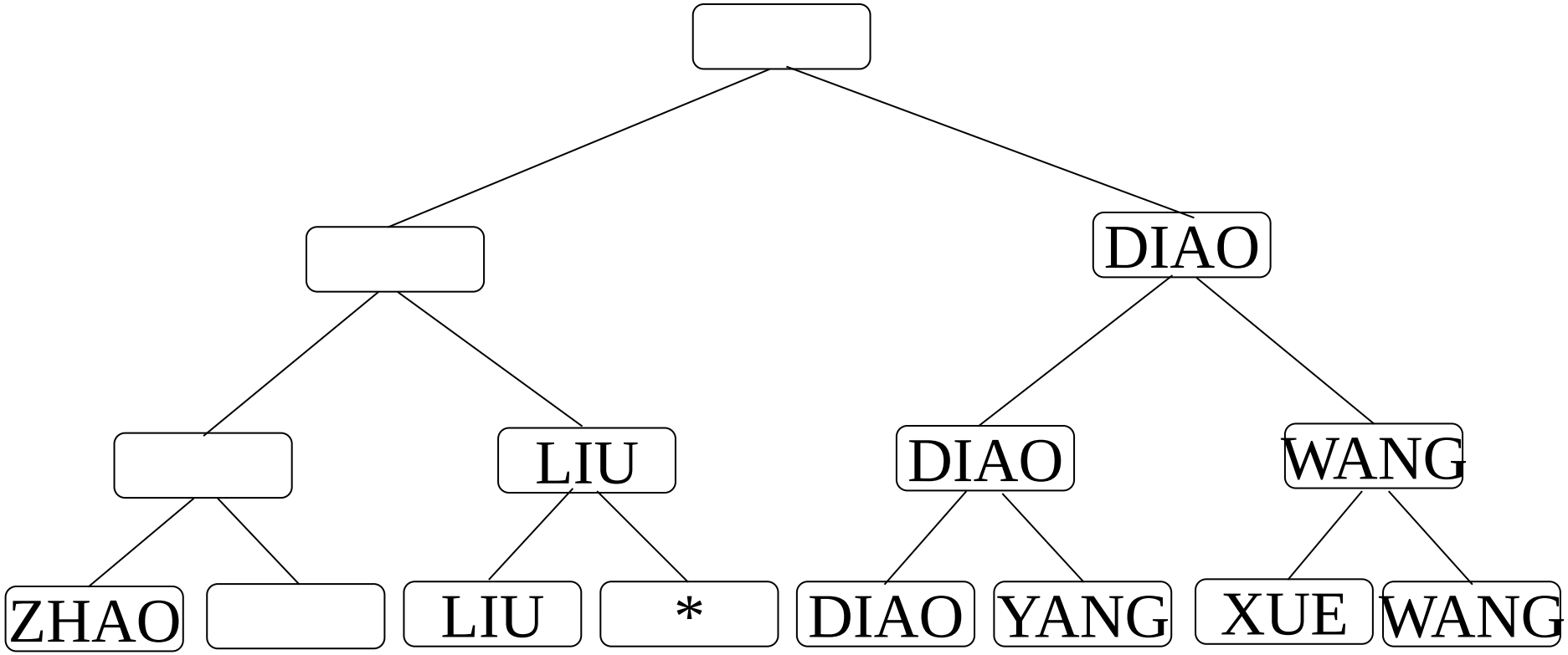
树形选择排序



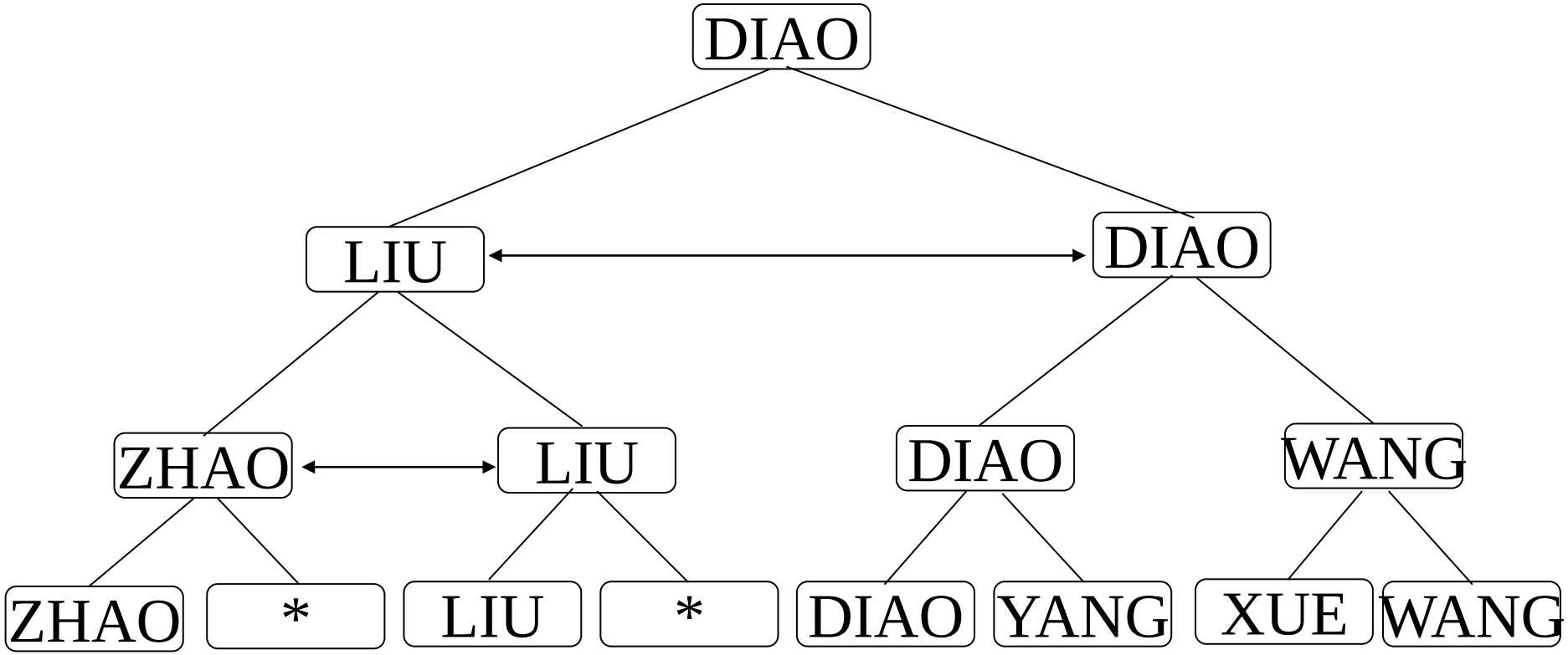
树形选择排序



树形选择排序

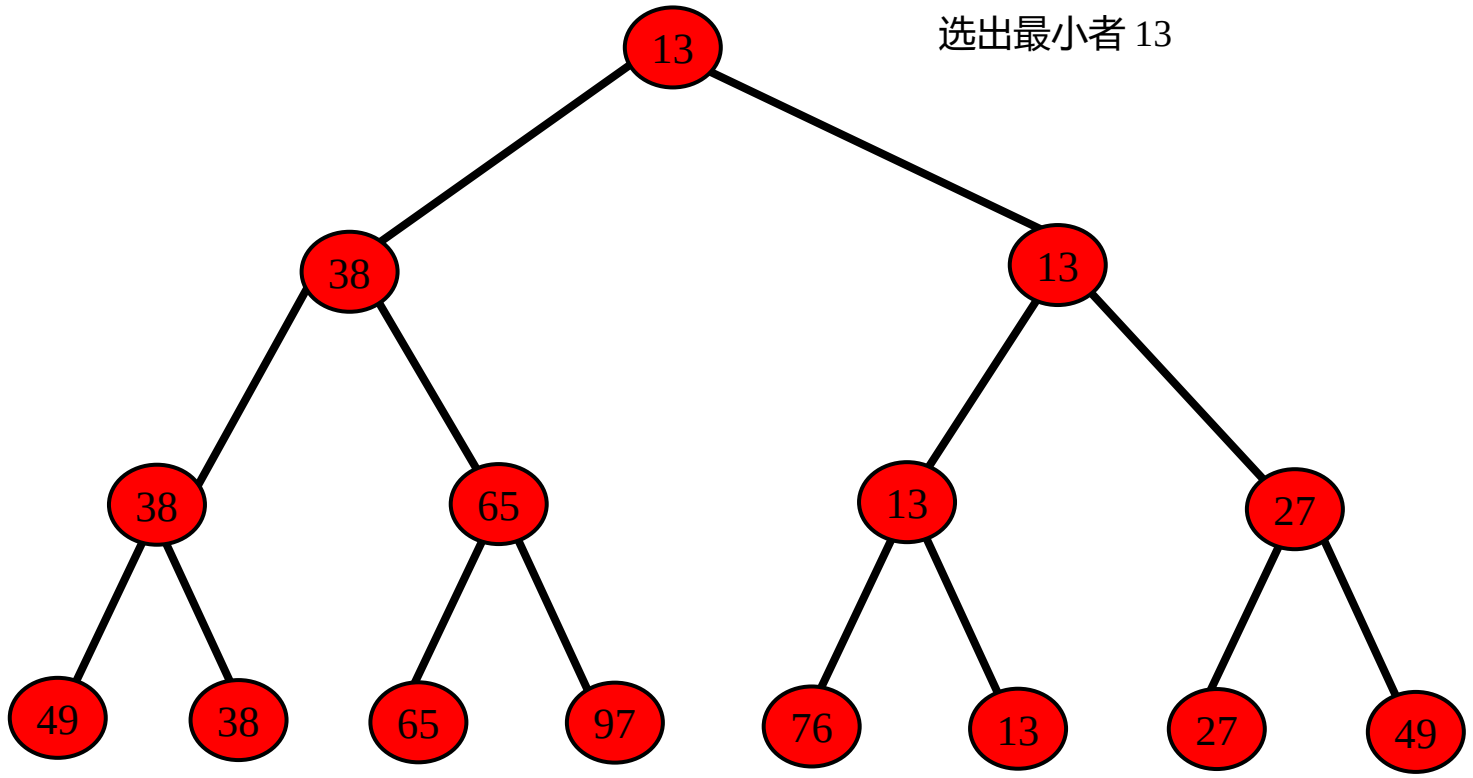


树形选择排序



树形选择排序

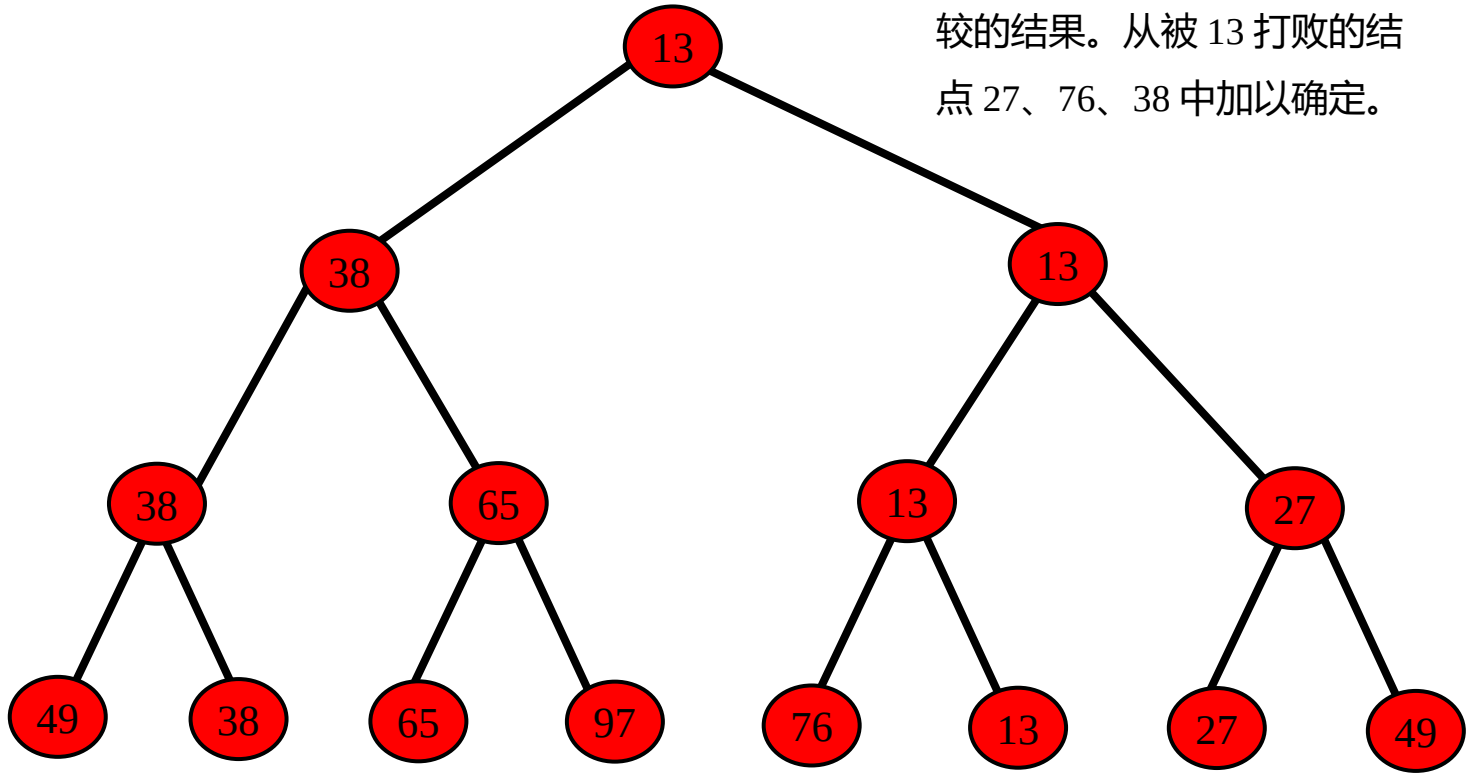
- 改进：简单选择排序没有利用上次选择的结果，是造成速度慢的重要原因。如果，能够加以改进，将会提高排序的速度。



树形选择排序

- 改进：简单选择排序没有利用上次选择的结果，是造成速度满的重要原因。如果，能够加以改进，将会提高排序的速度。

选出次最小者，应利用上次比较的结果。从被 13 打败的结点 27、76、38 中加以确定。



堆排序

什么是堆?

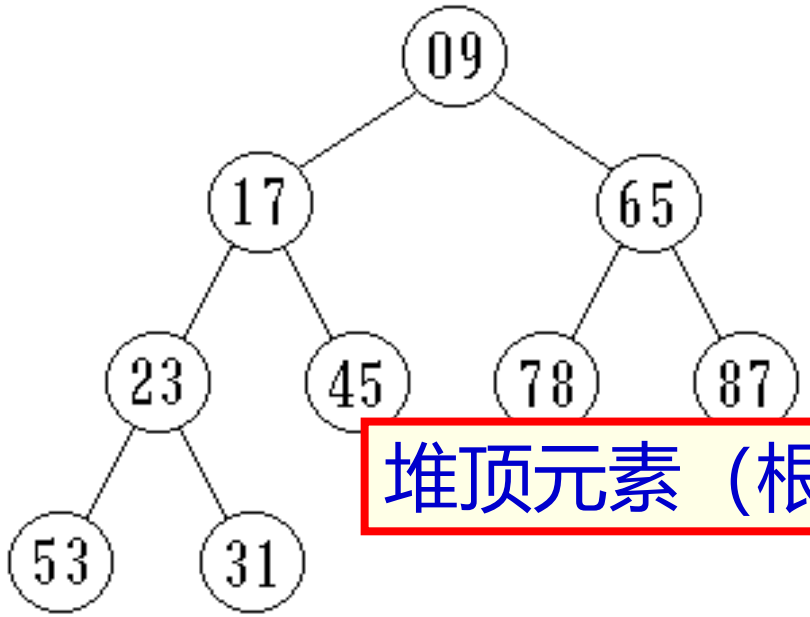
n个元素的序列{k1,k2,...,kn}，当且仅当满足下列关系时，成为堆：

$$\begin{cases} k_i \le k_{2i} \\ k_i \le k_{2i+1} \end{cases} \text{或} \begin{cases} k_i \ge k_{2i} \\ k_i \ge k_{2i+1} \end{cases}$$

- 如果将序列看成一个完全二叉树，非终端结点的值均小于或大于左右子结点的值。

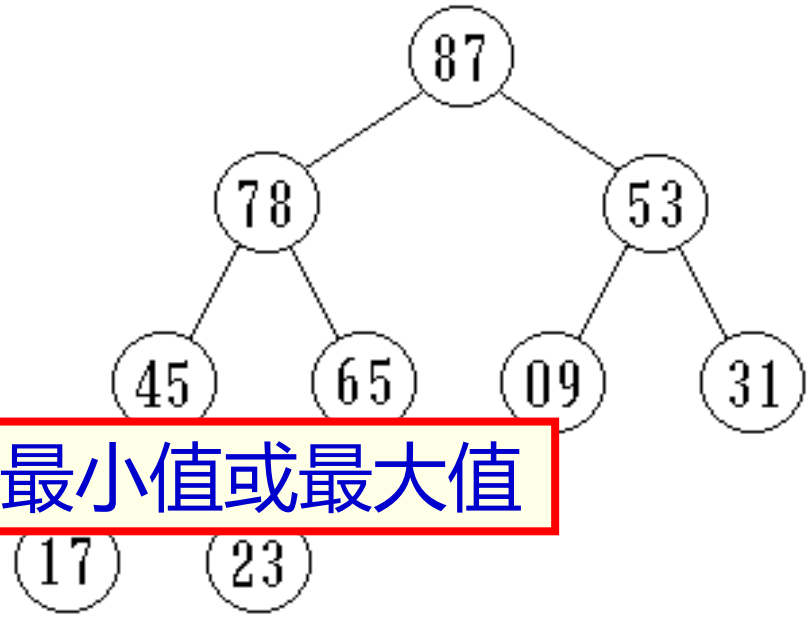
– 利用树的结构特征来描述堆，所以树只是作为堆的描述工具，堆实际是存放在线形空间中的。

(09,17,65,23,45,78,87,53,31)



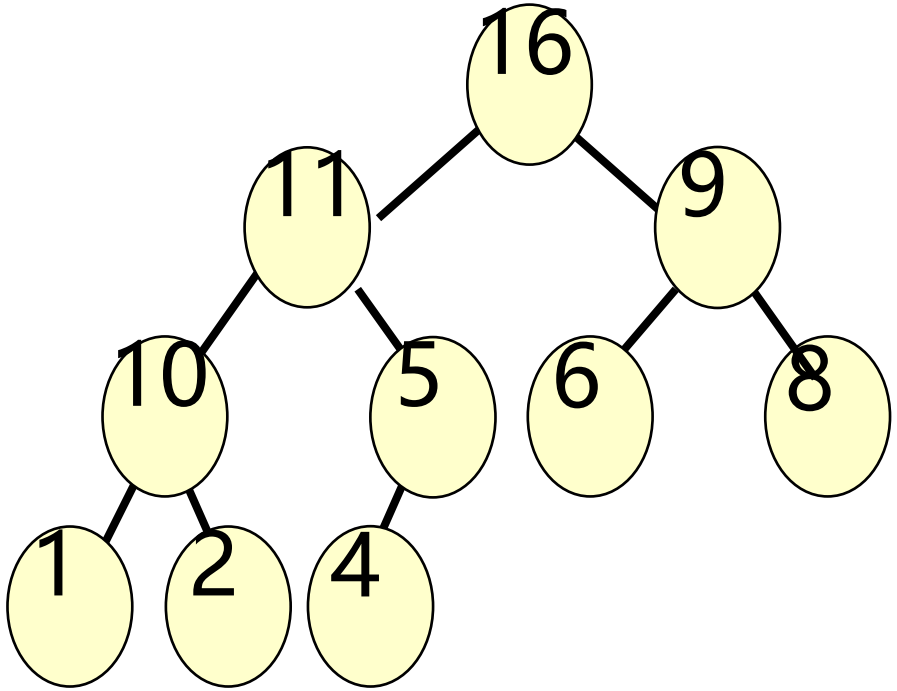
(a) 最小堆

(87,78,53,45,65,09,31,17,23)

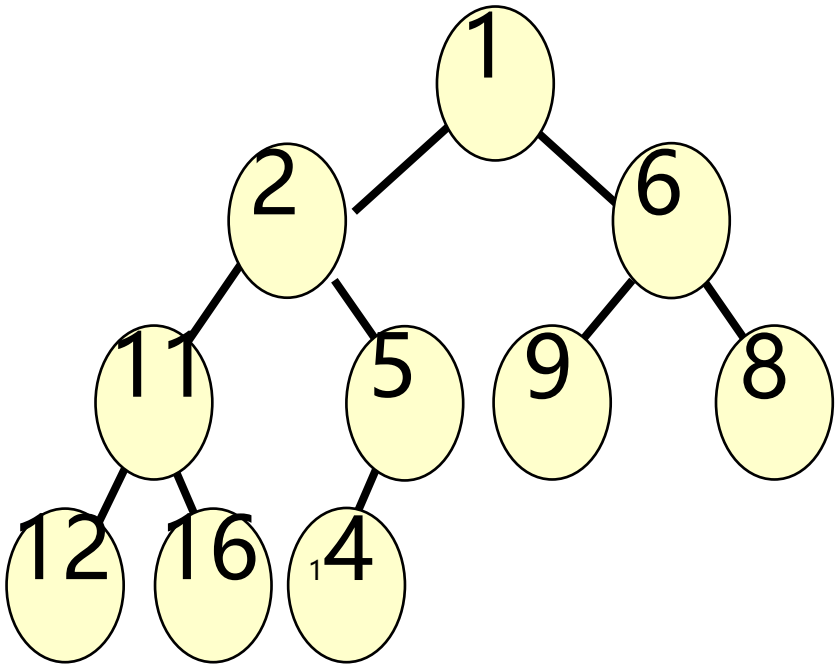


(b) 最大堆

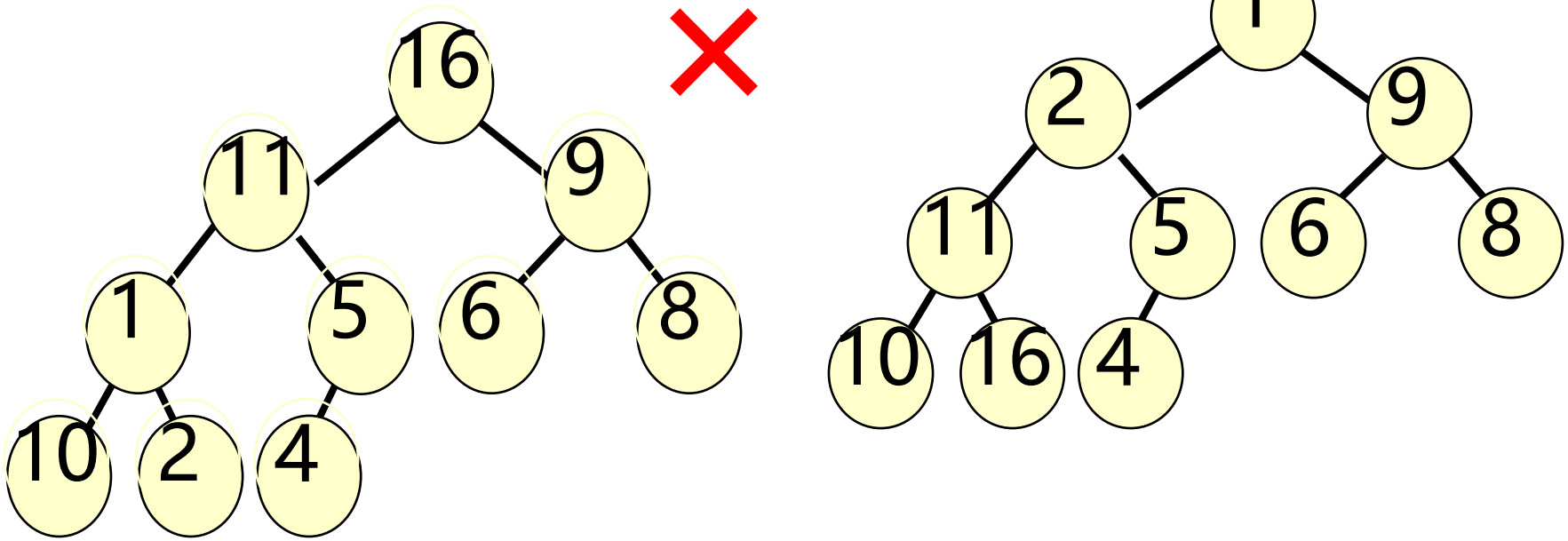
堆顶元素（根）为最小值或最大值



大根堆

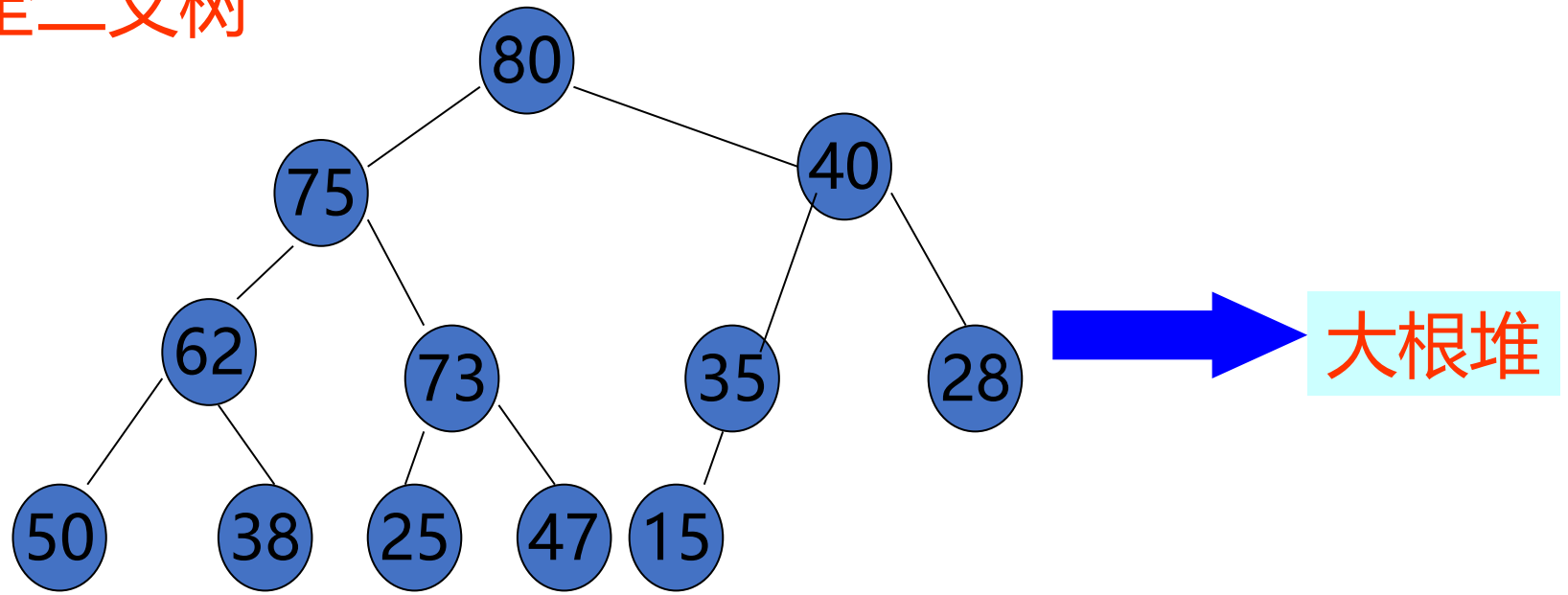


小根堆


16
11
9
1
5
6
8
10
2
4
1
2
9
11
5
6
8
10
16
4

判定(80,75,40,62,73,35,28,50,38,25,47,15)是否为堆

完全二叉树



堆排序

如何建？？

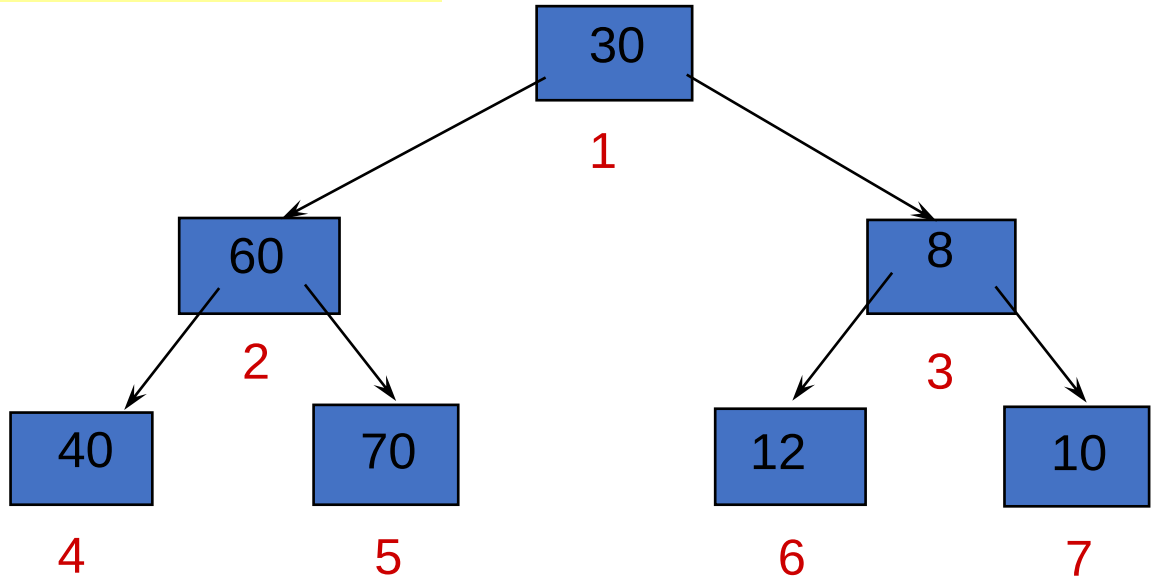
基本思想：

- ✓将无序序列建成一个堆
- ✓输出堆顶的最小（大）值
- ✓使剩余的n-1个元素又调整成一个堆，则可得到n个元素的次小值
- ✓重复执行，得到一个有序序列

如何调整？？

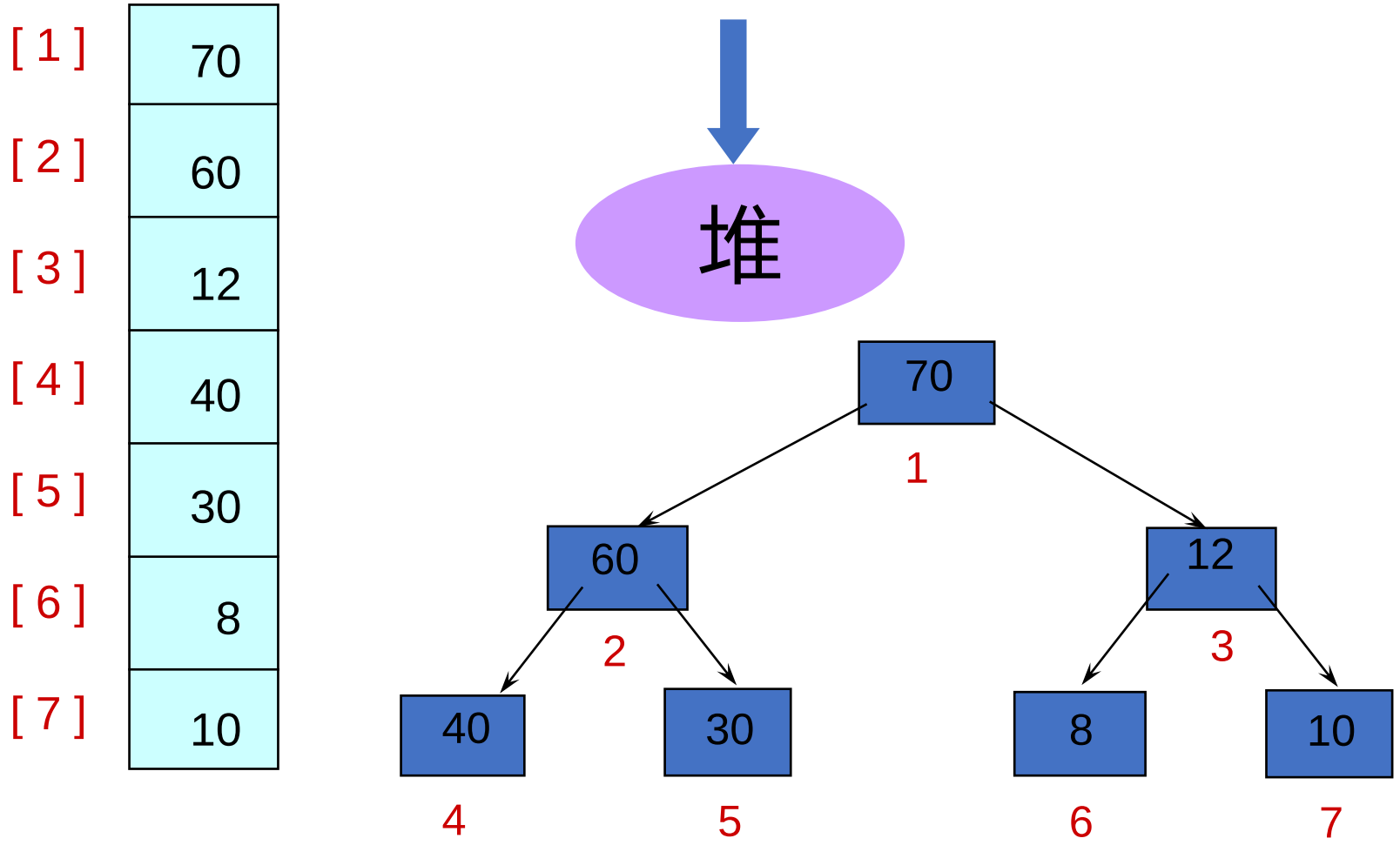
如何将无序序列建成堆

[1]	30
[2]	60
[3]	8
[4]	40
[5]	70
[6]	12
[7]	10

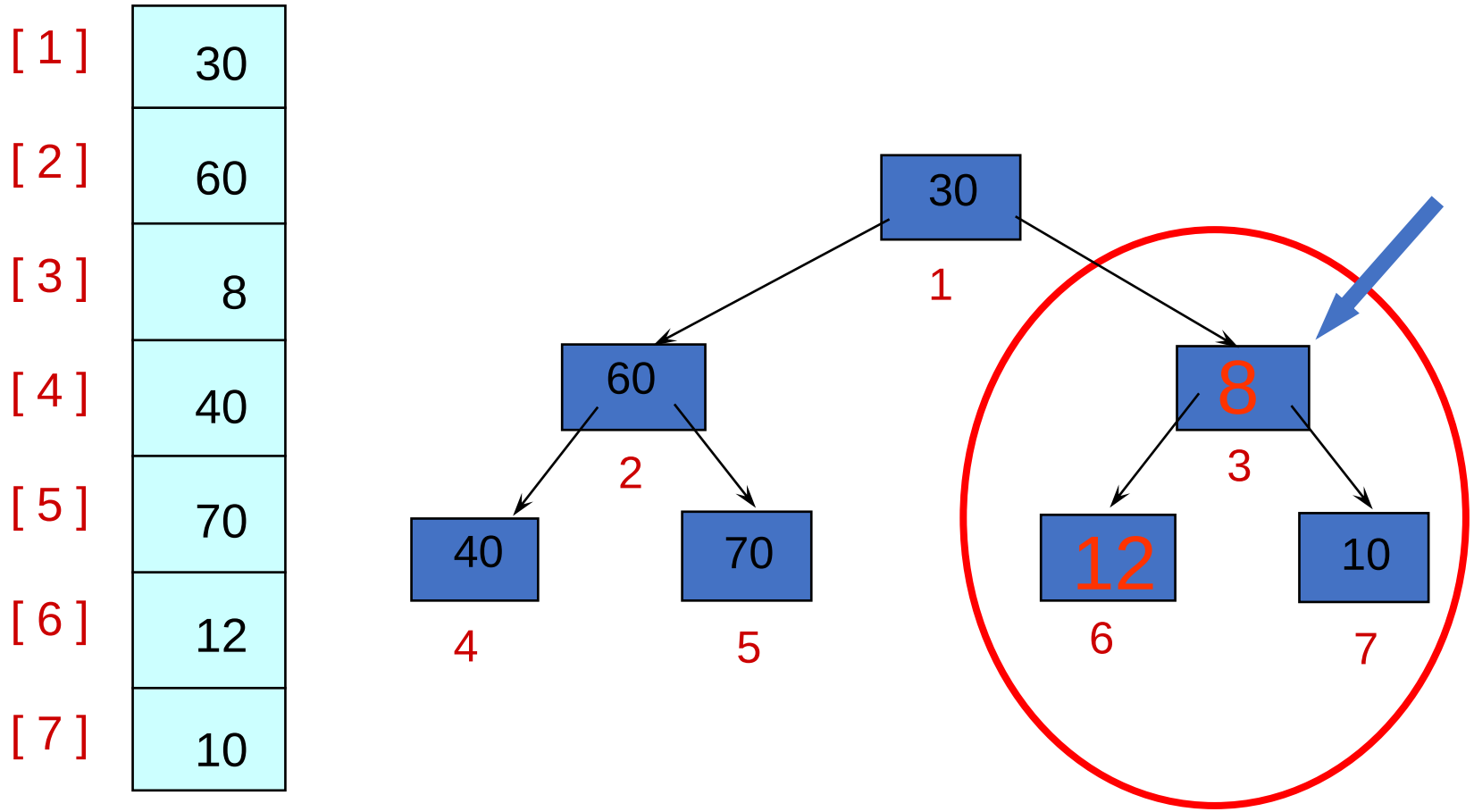


思考：有n 个结点的完全二叉树，最后一个分支结点的标号是多少？$\lfloor n/2 \rfloor$

从第$\lfloor n/2 \rfloor$个元素起，至第一个元素止，进行反复筛选

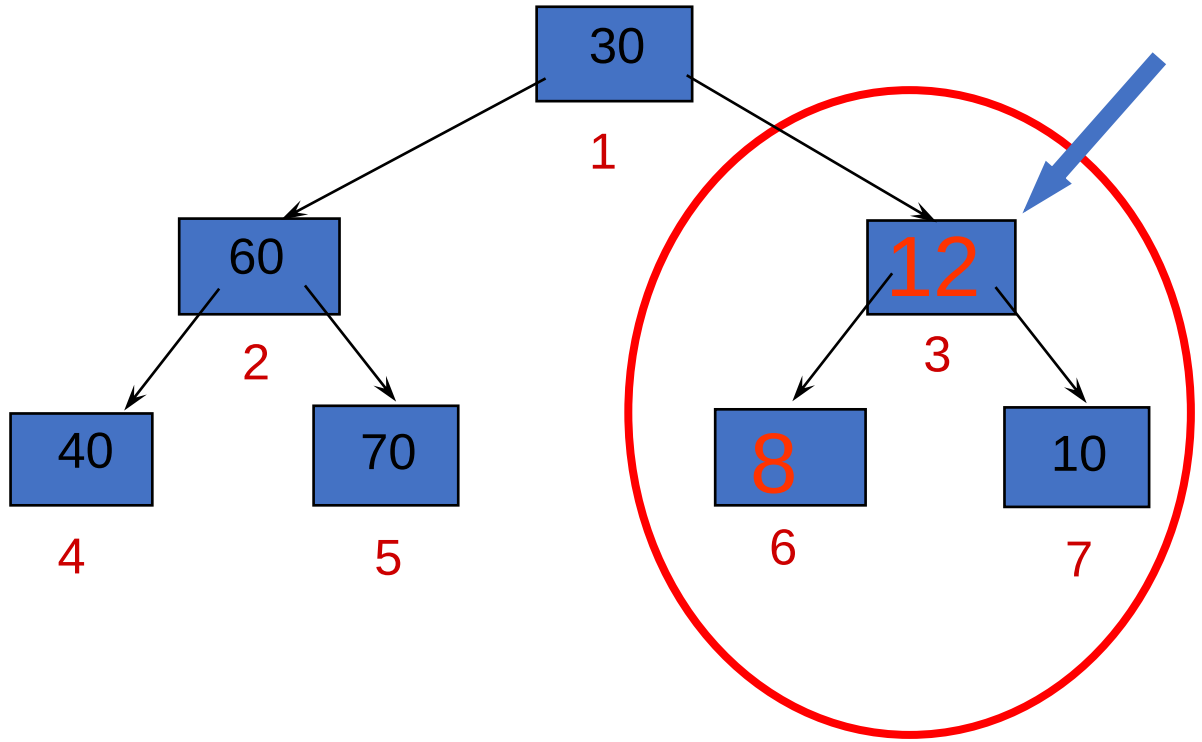


无序序列建成堆－1

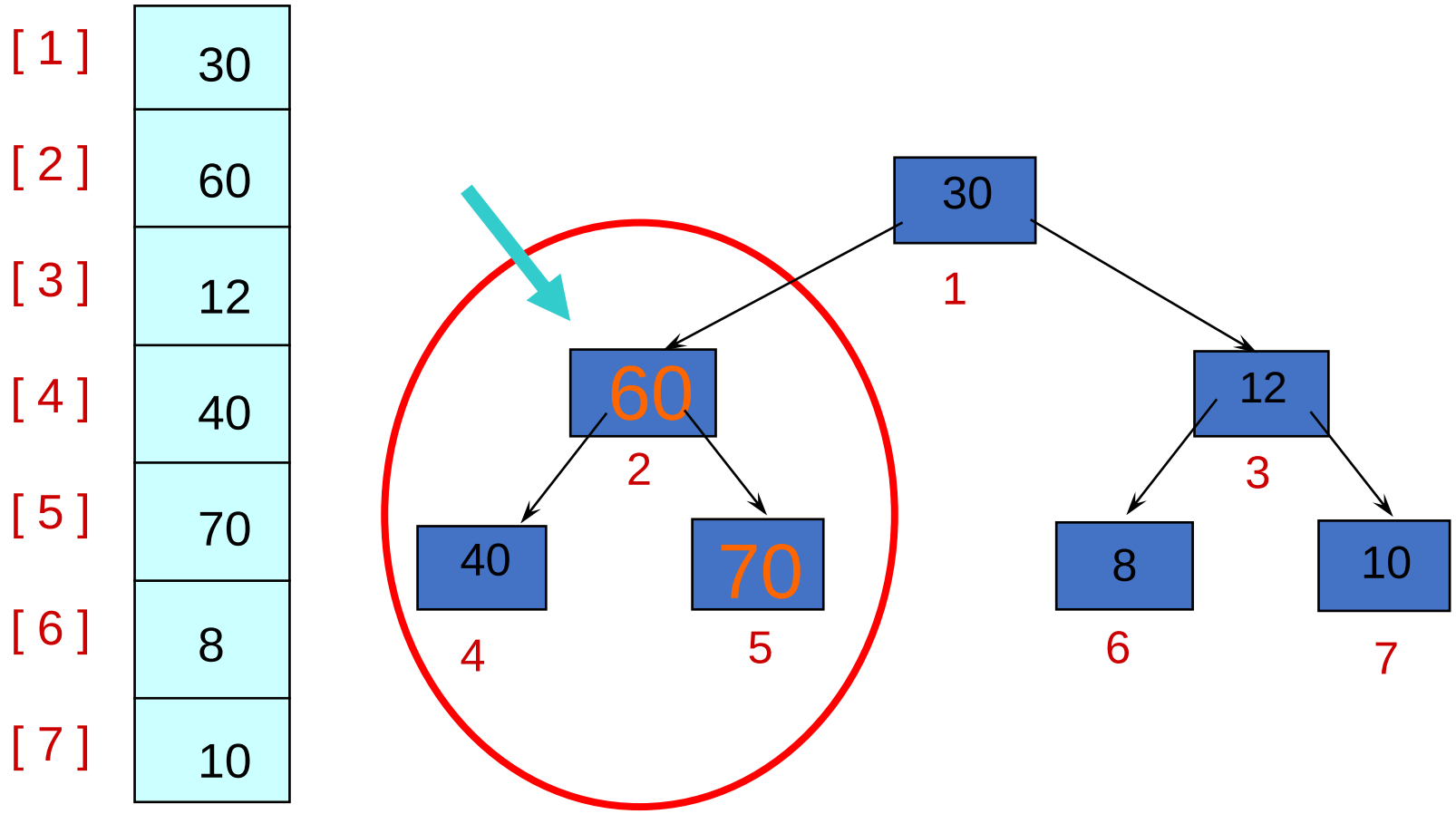


无序序列建成堆－1

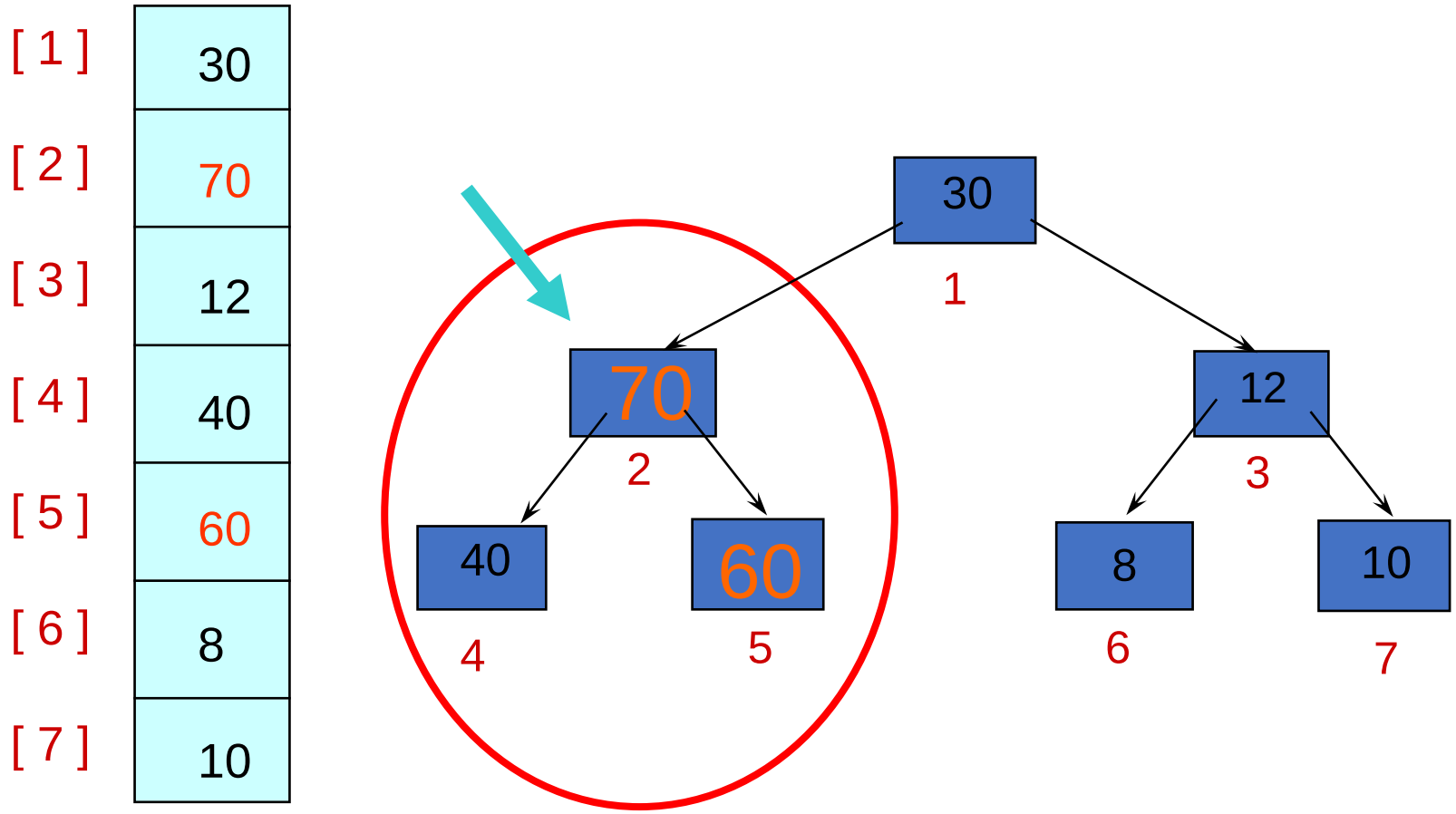
[1]	30
[2]	60
[3]	12
[4]	40
[5]	70
[6]	8
[7]	10



无序序列建成堆－2

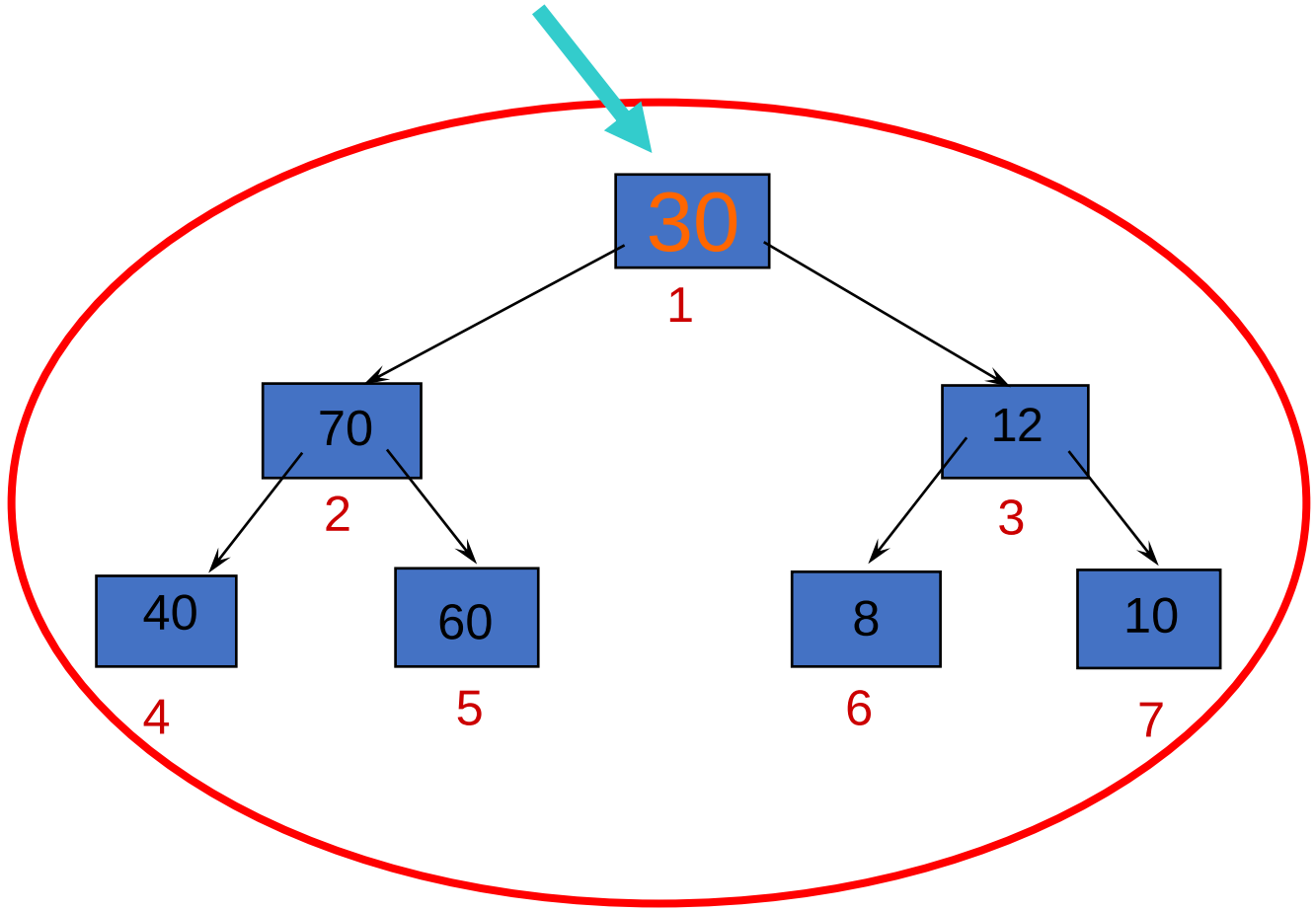


无序序列建成堆－2



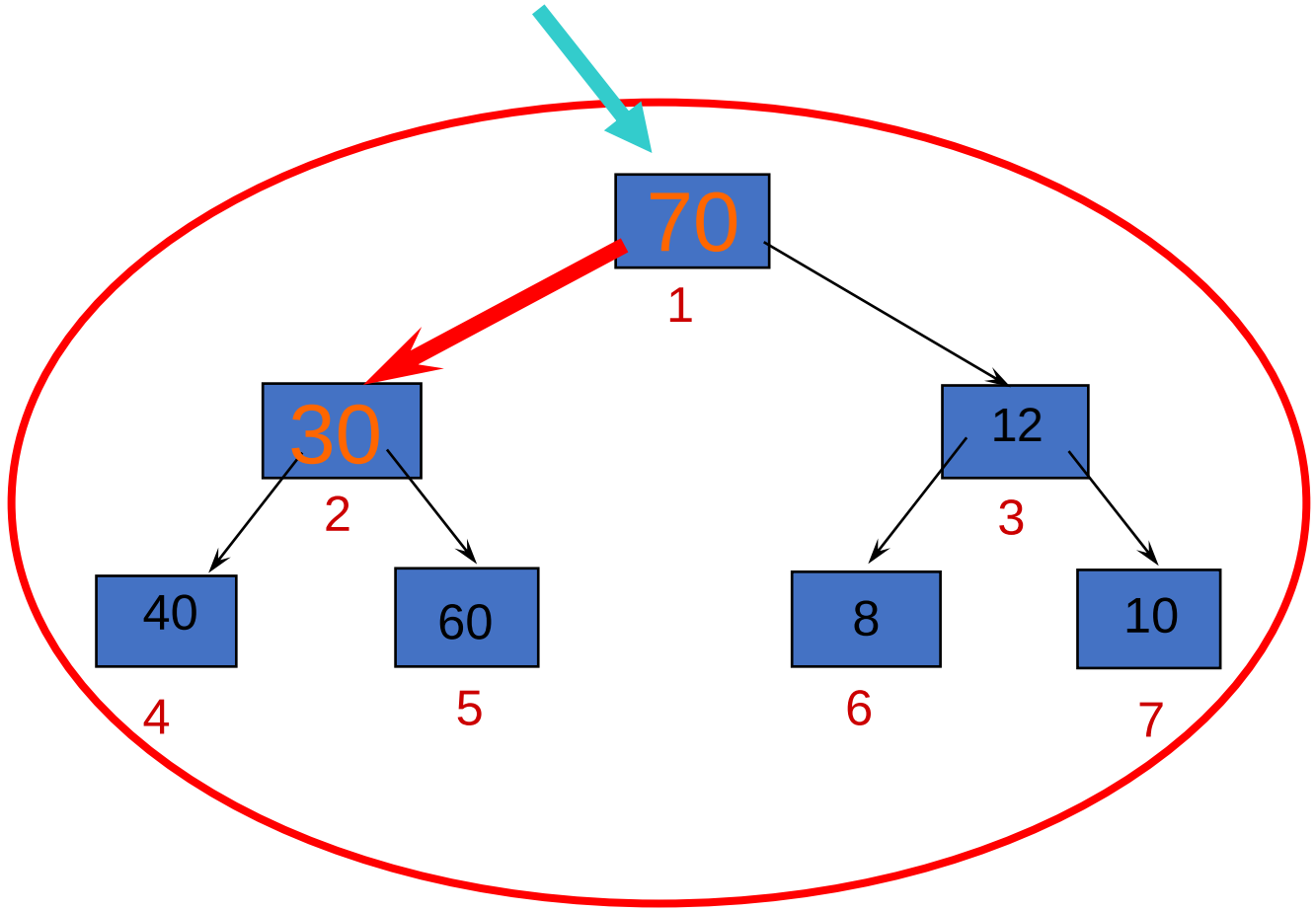
无序序列建成堆－3

[1]	30
[2]	70
[3]	12
[4]	40
[5]	60
[6]	8
[7]	10



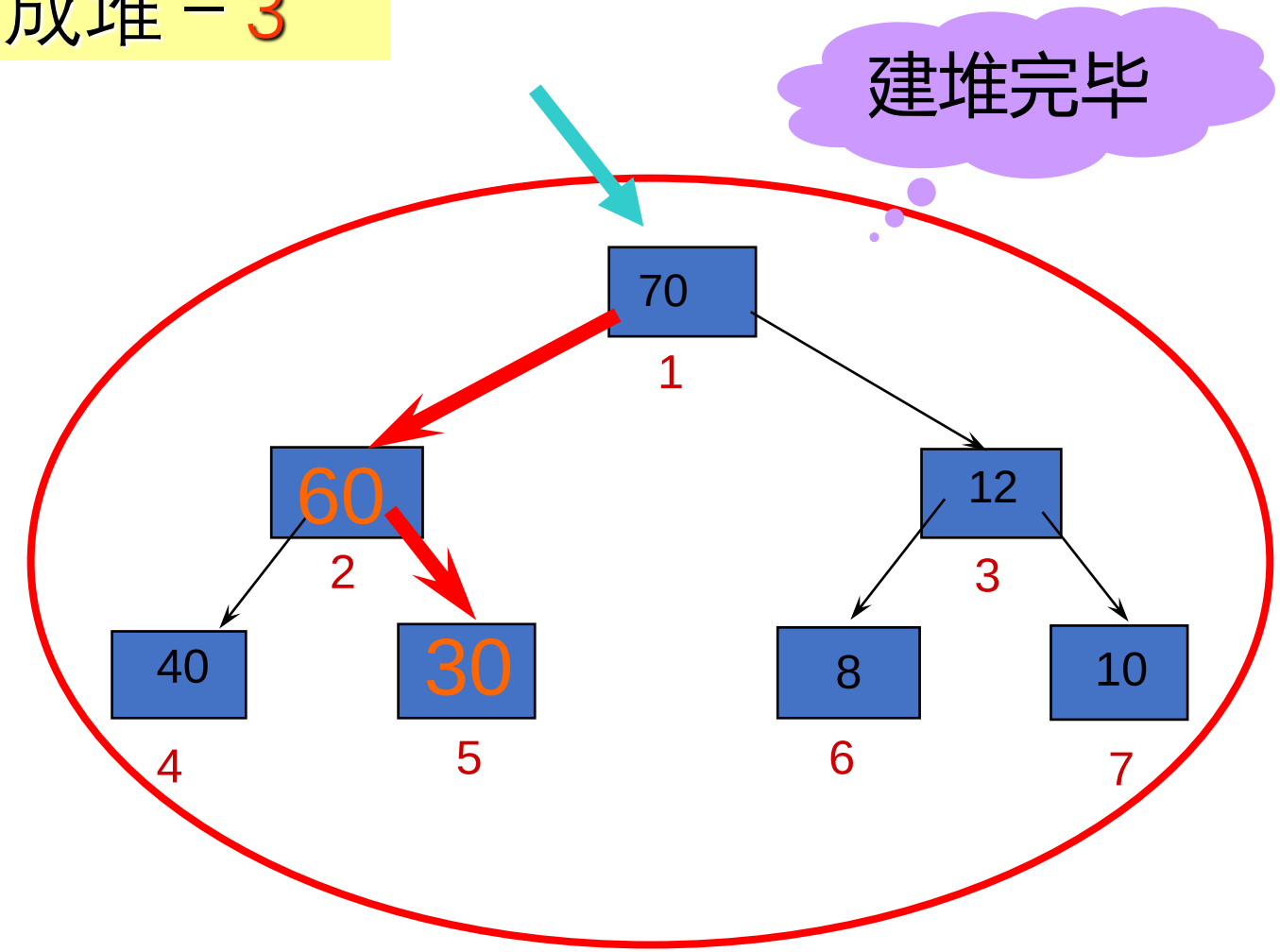
无序序列建成堆－3

[1]	70
[2]	30
[3]	12
[4]	40
[5]	60
[6]	8
[7]	10



无序序列建成堆－3

[1]	70
[2]	60
[3]	12
[4]	40
[5]	30
[6]	8
[7]	10



Back

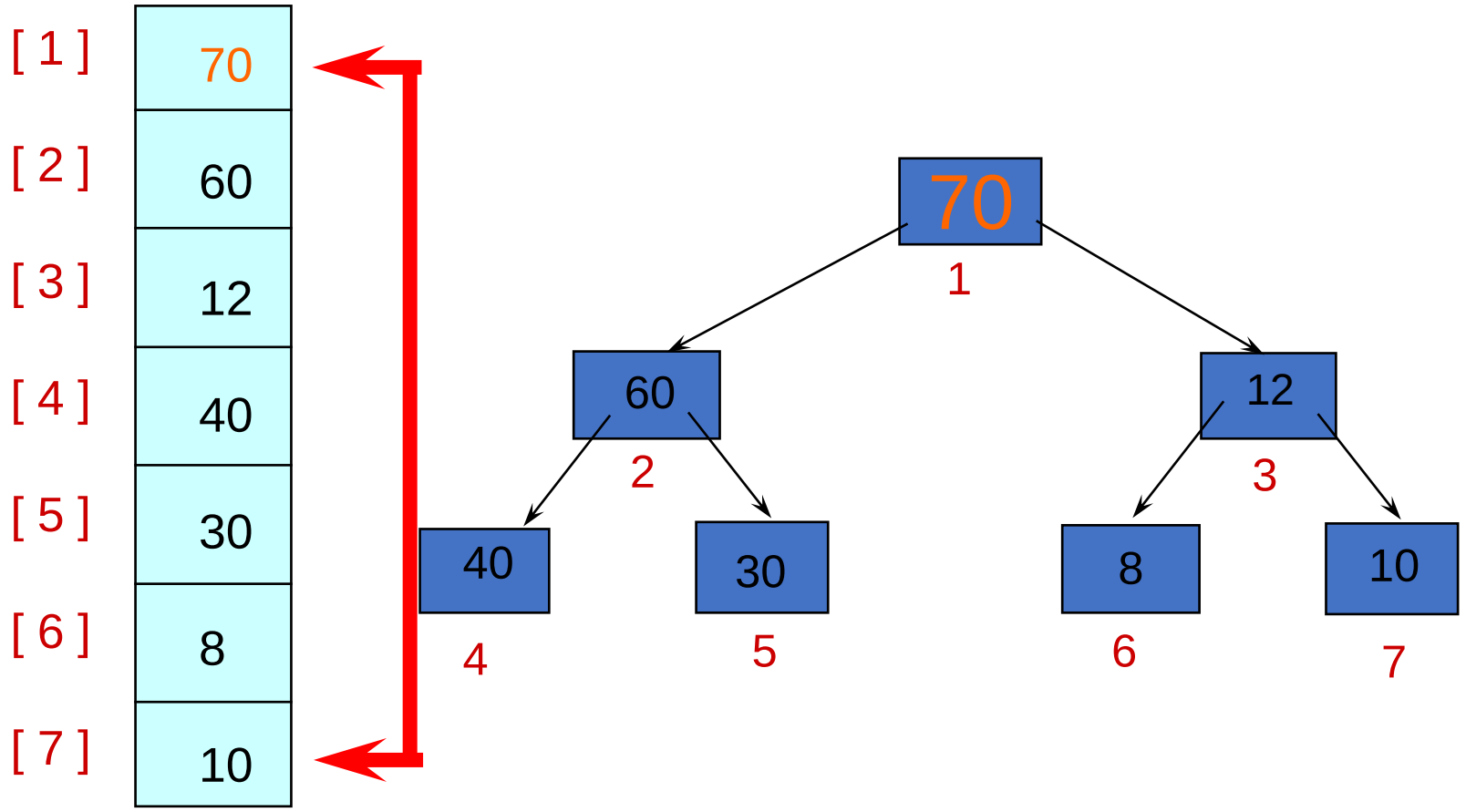
堆的重新调整

如何在输出堆顶元素后调整，使之成为新堆？

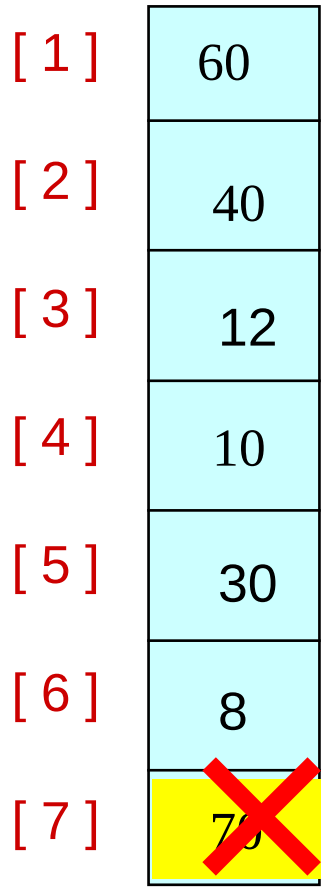
筛选

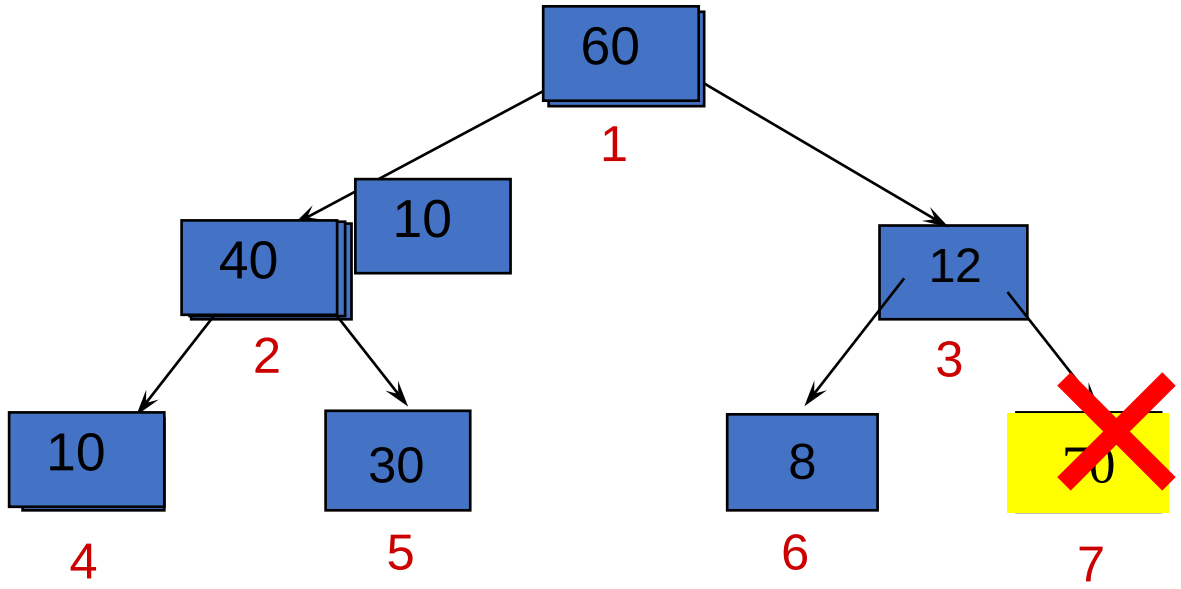
- ✓输出堆顶元素后，以堆中最后一个元素替代之
- ✓将根结点与左、右子树根结点比较，并与小者交换
- ✓重复直至叶子结点，得到新的堆

堆的重新调整 – 1

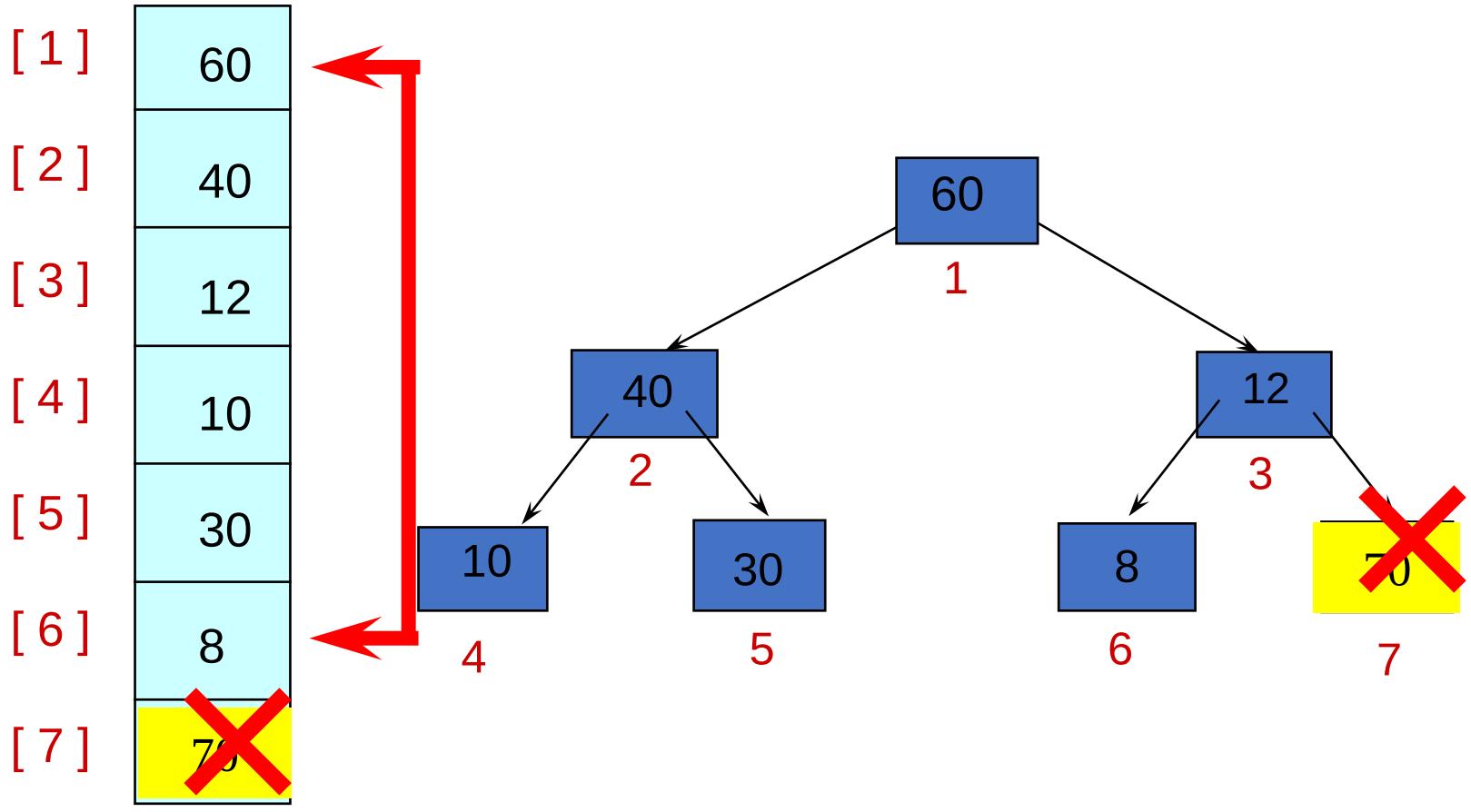


堆的重新调整 – 1

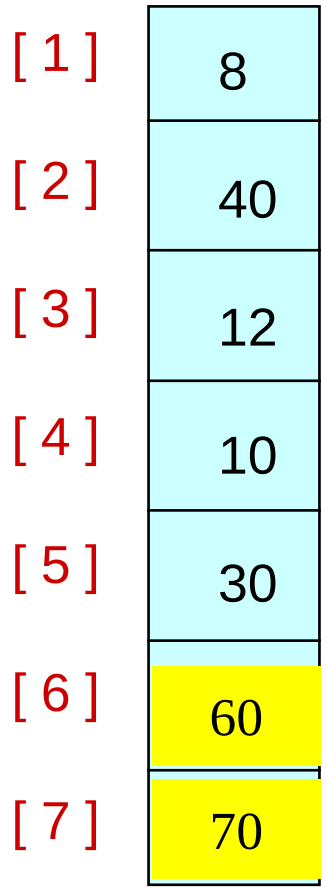


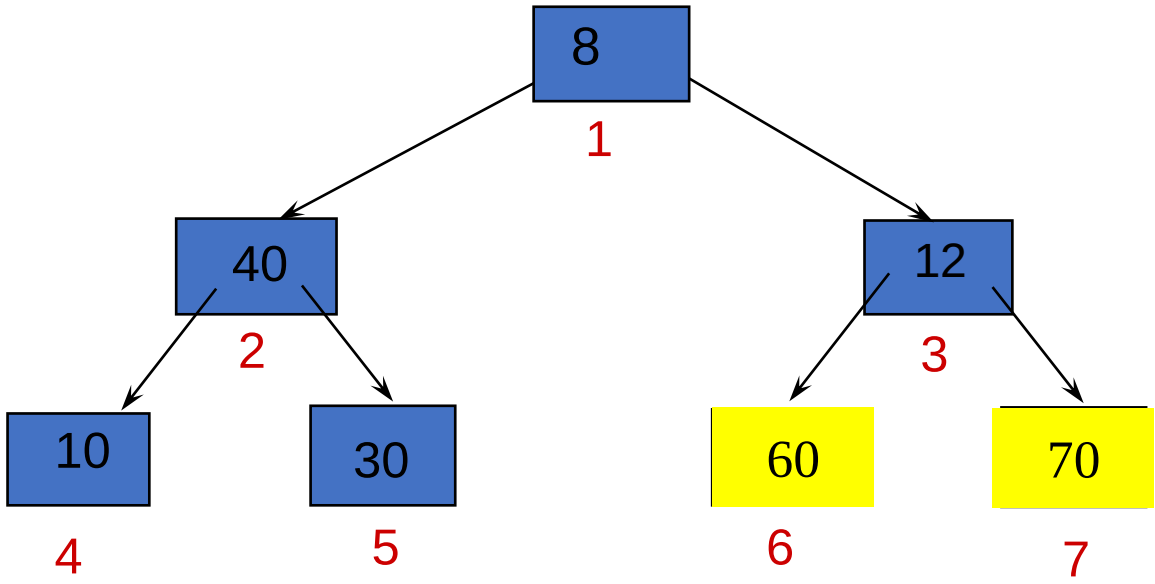


堆的重新调整 – 2

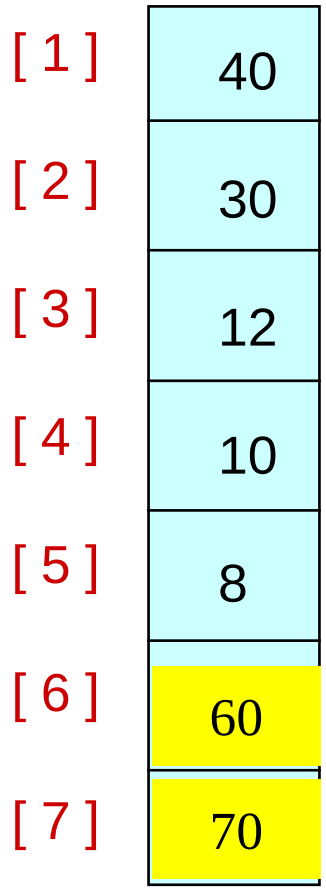


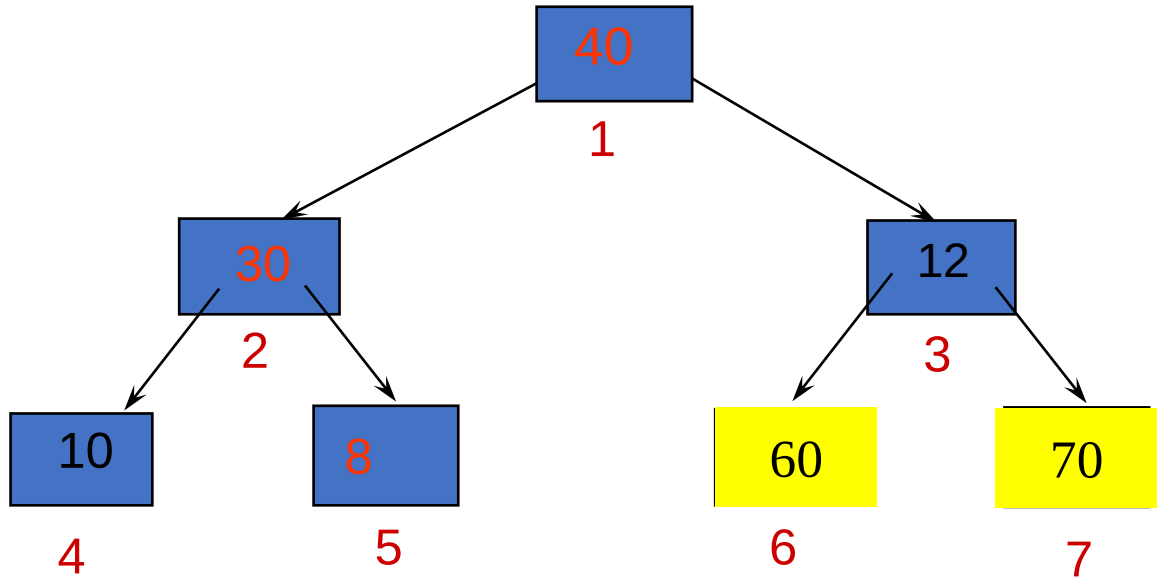
堆的重新调整 – 2



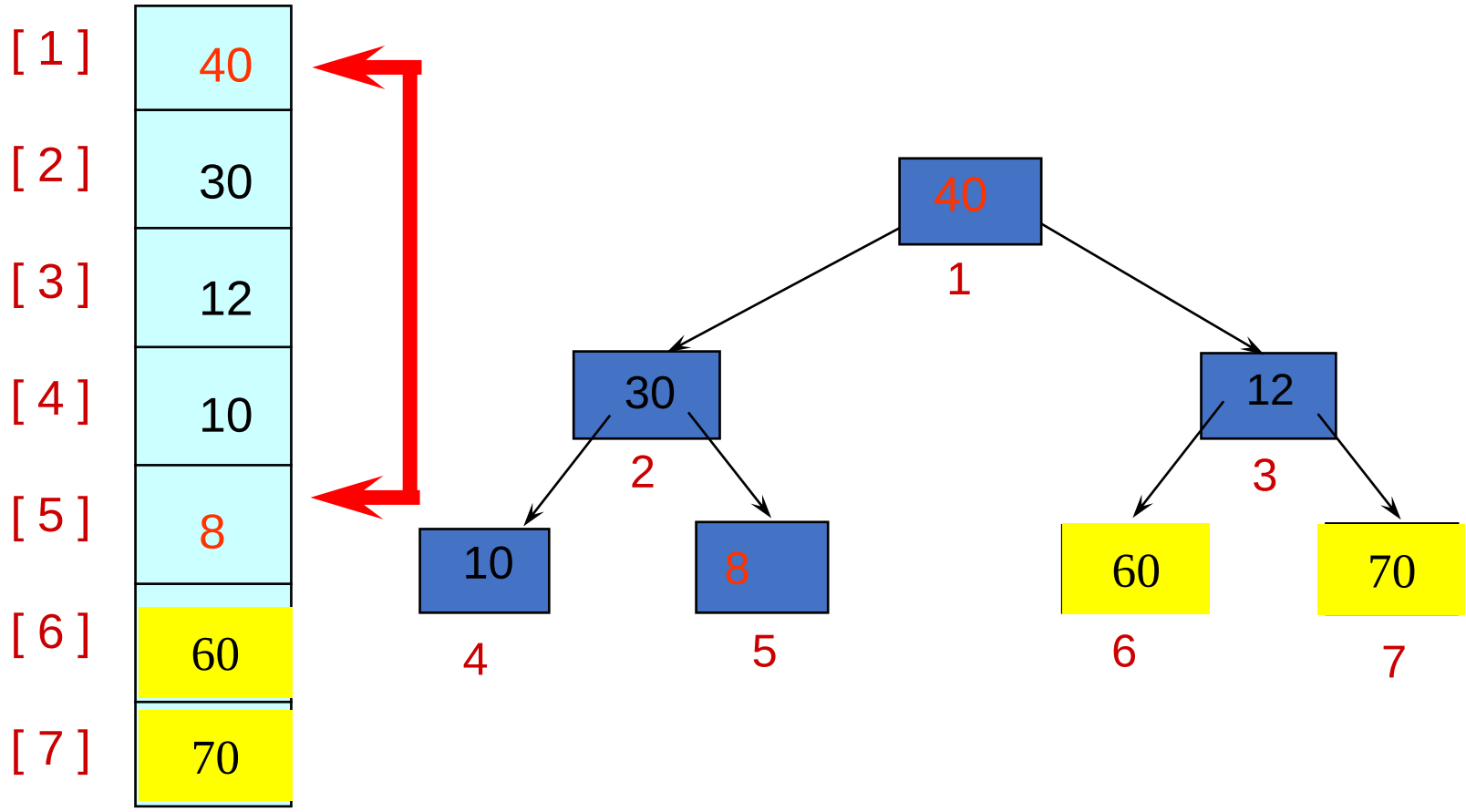


堆的重新调整 – 2

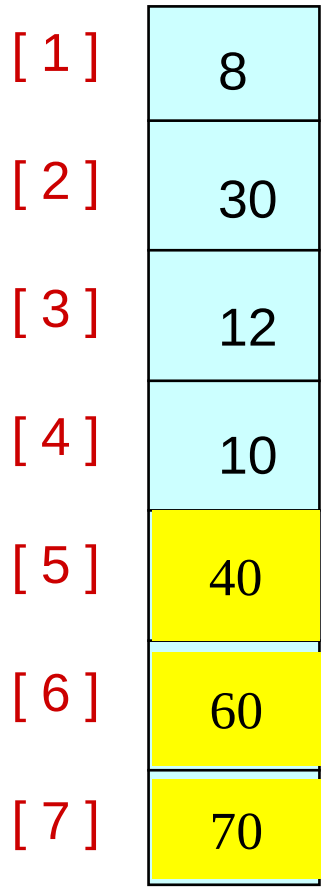


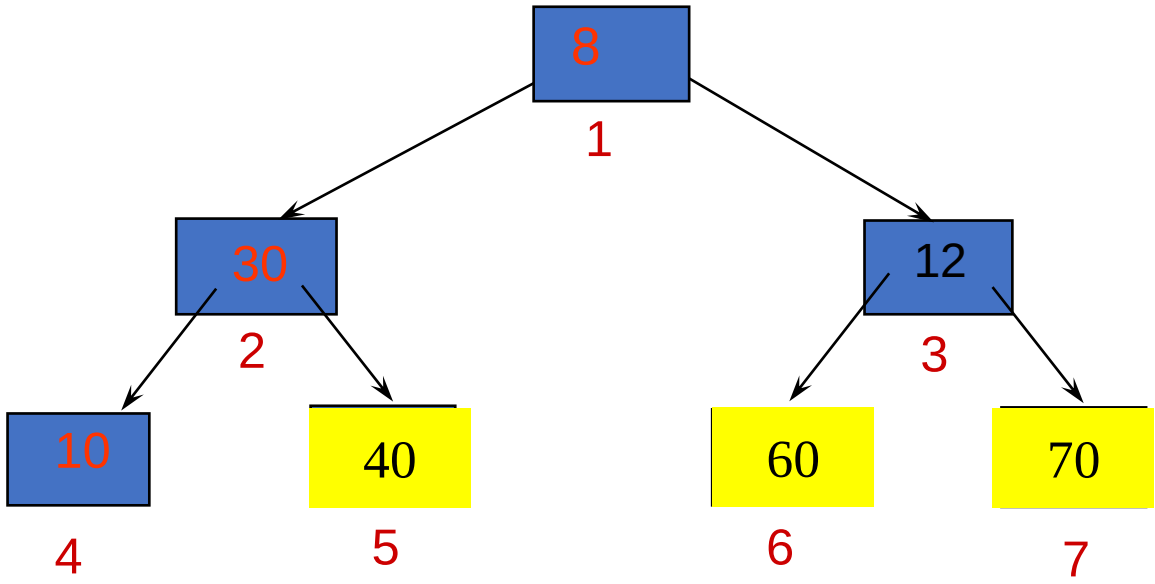


堆的重新调整 – 3

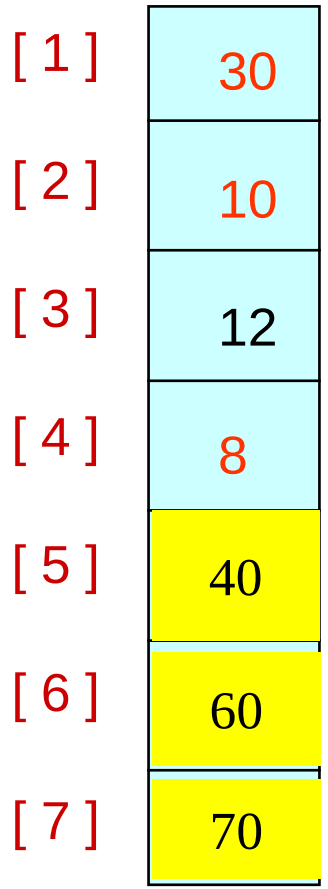


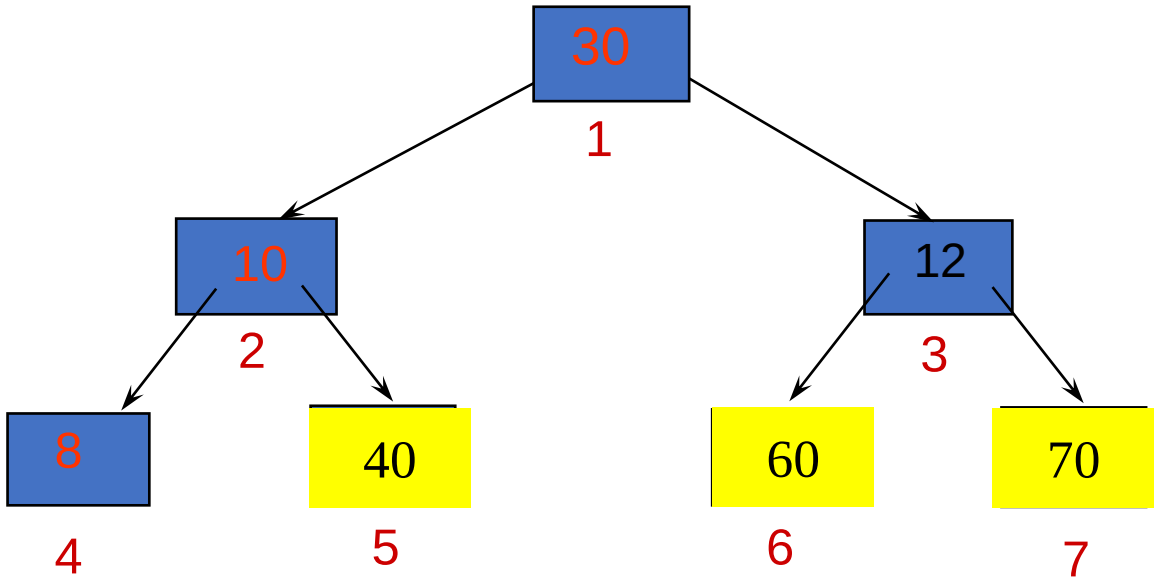
堆的重新调整 – 3



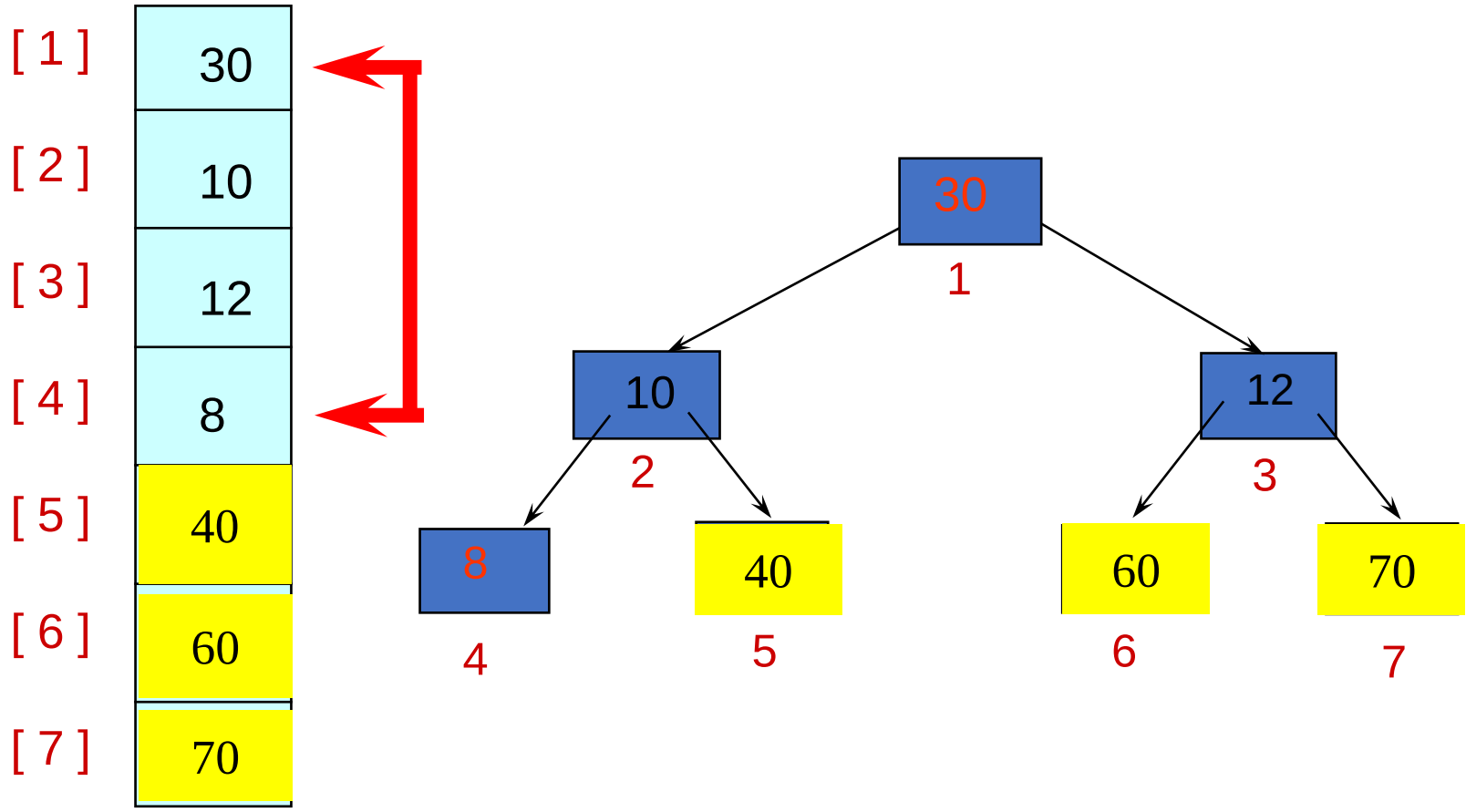


堆的重新调整 – 3

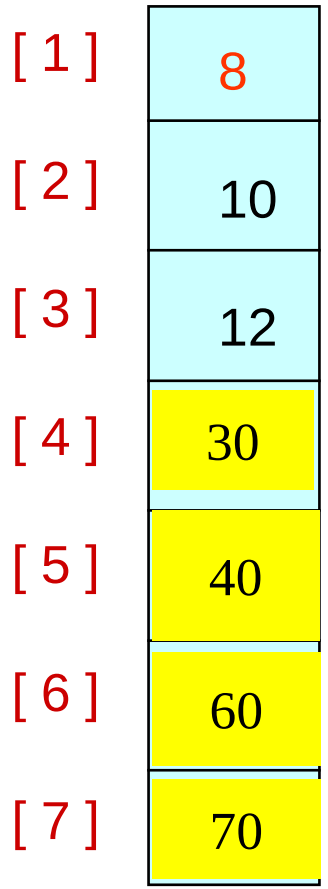


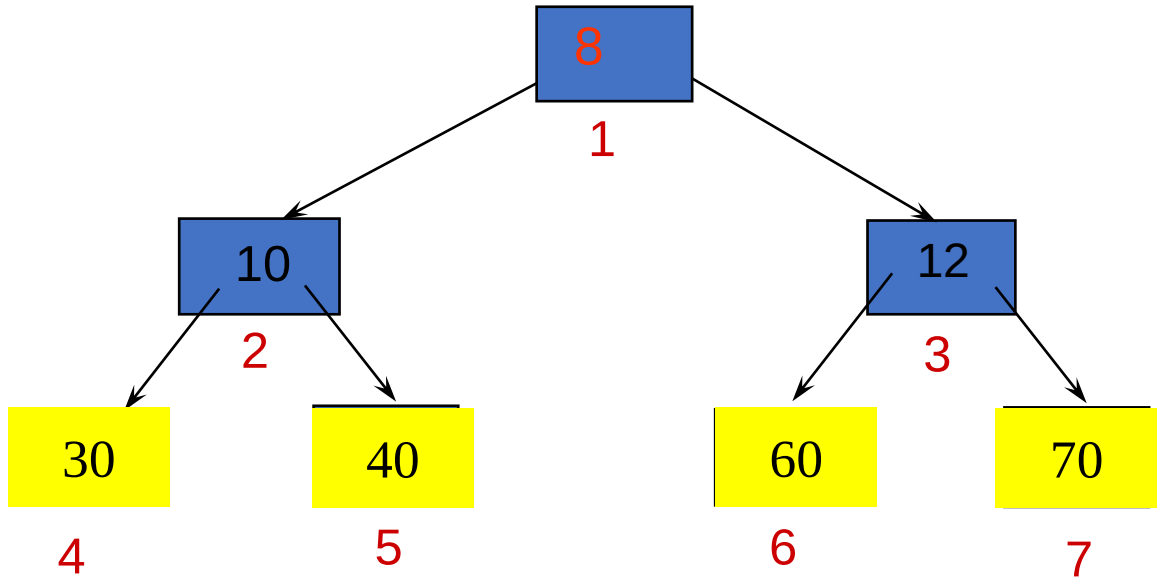


堆的重新调整 – 4

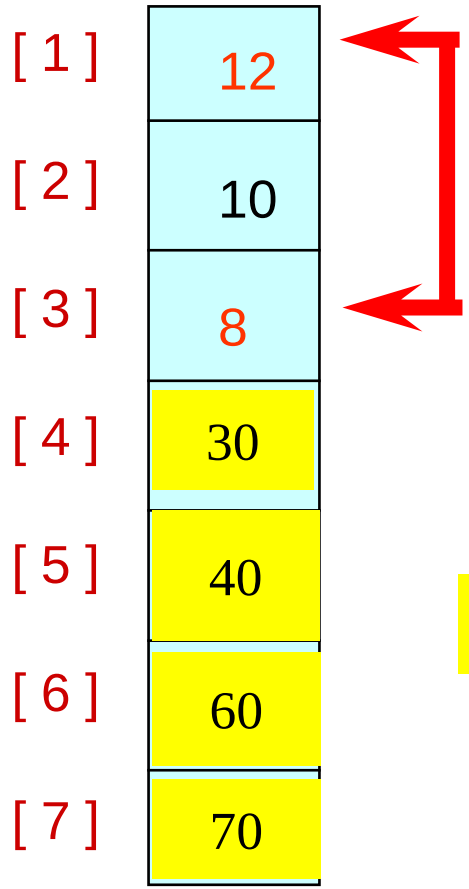


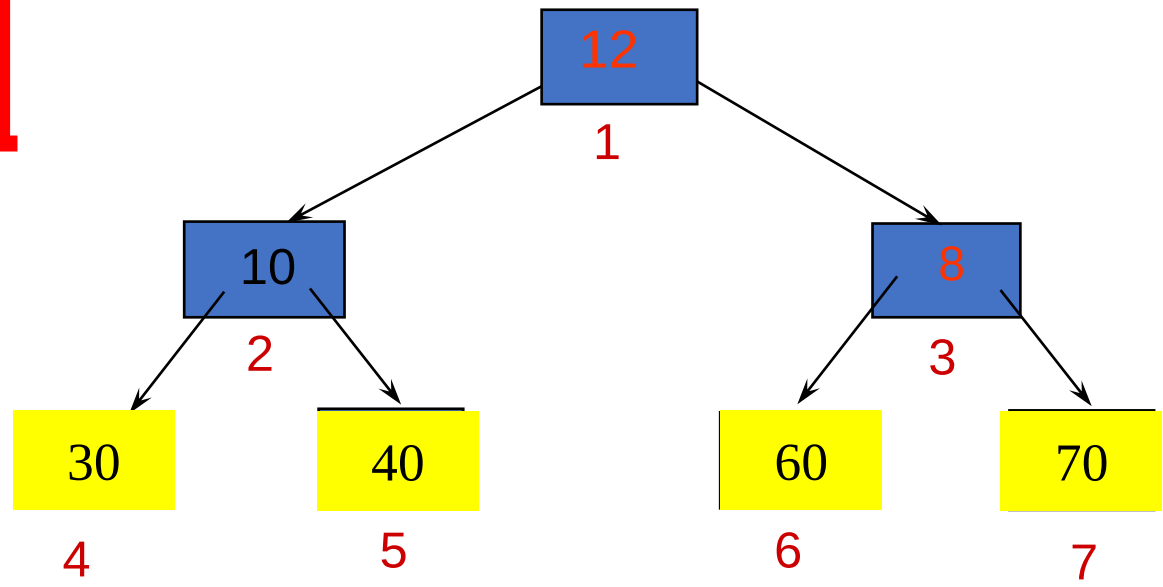
堆的重新调整 – 4



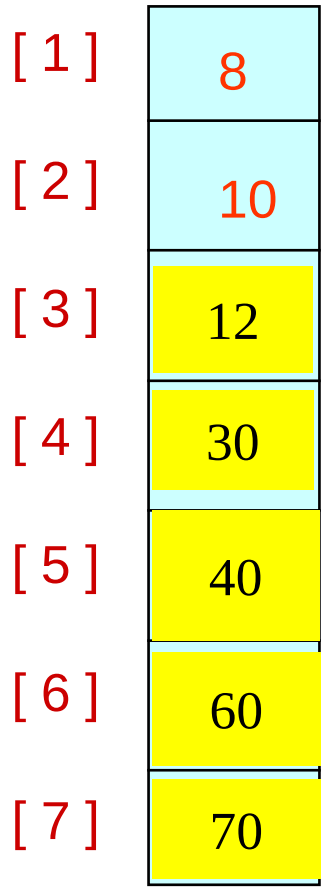


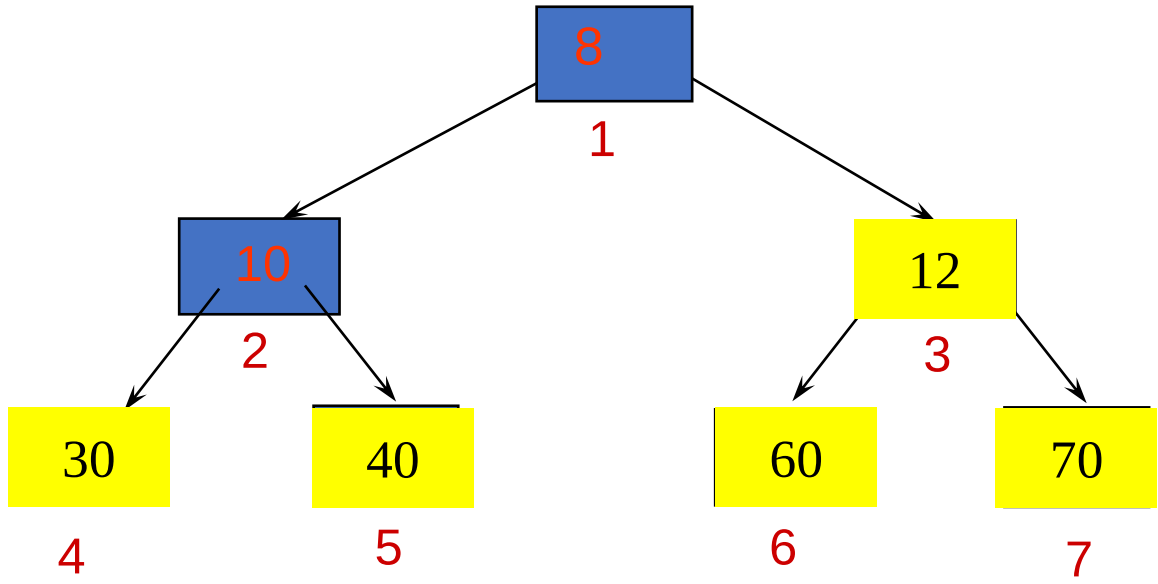
堆的重新调整 – 5



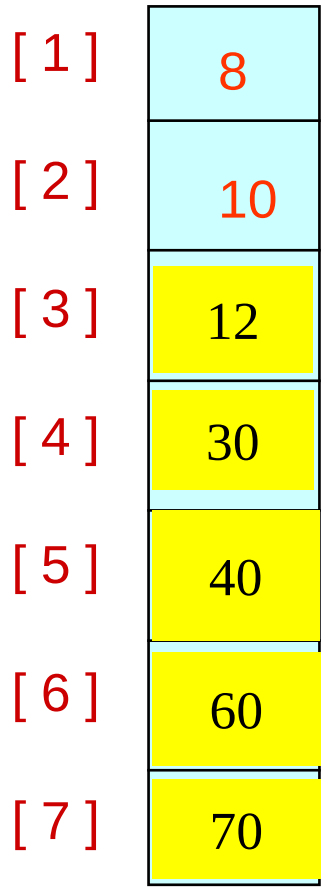


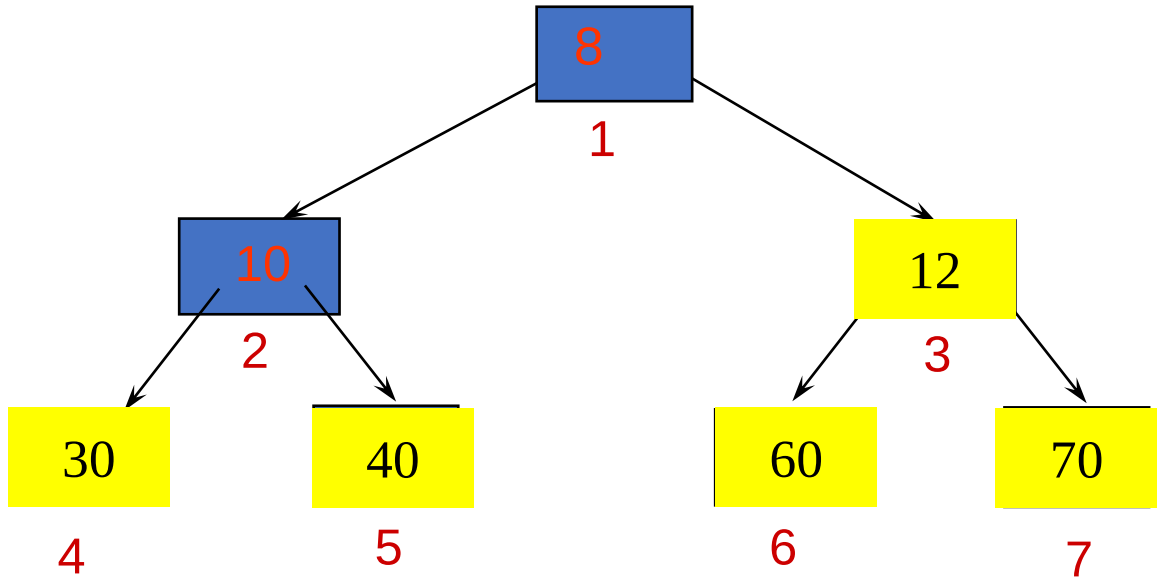
堆的重新调整 – 5



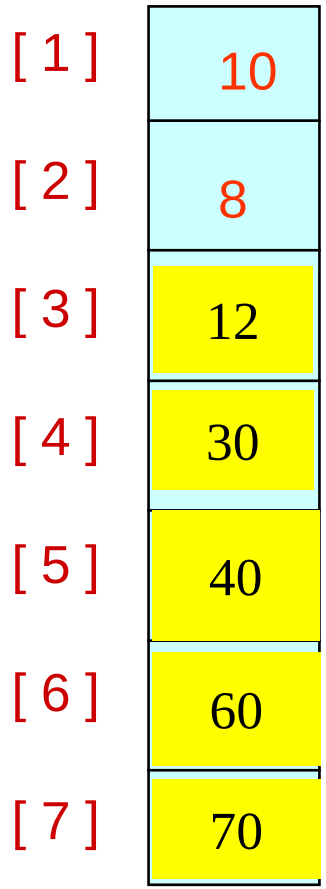


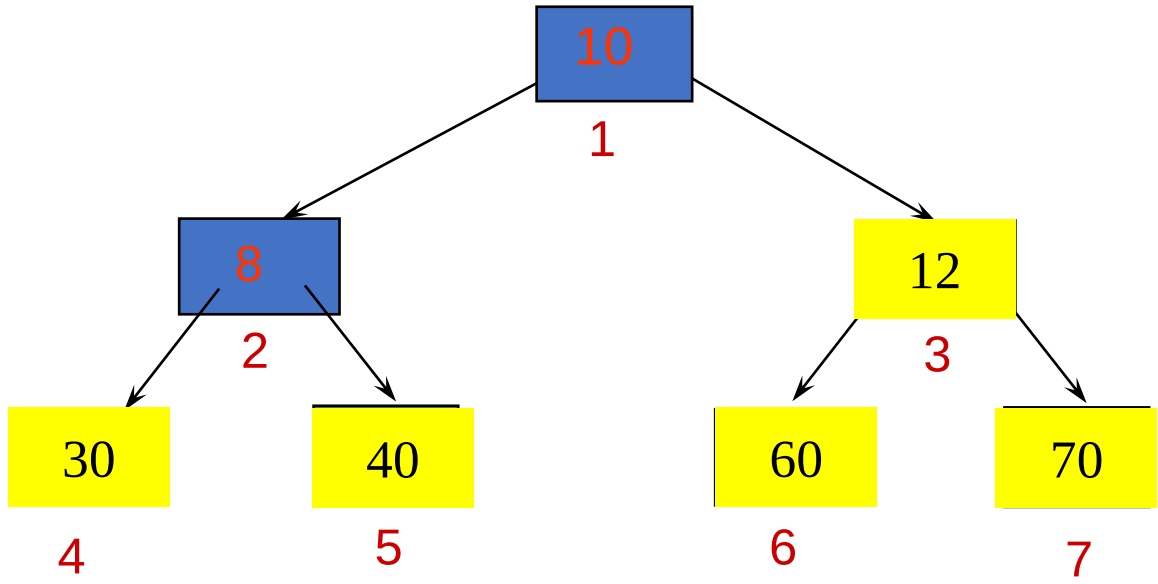
堆的重新调整 – 5



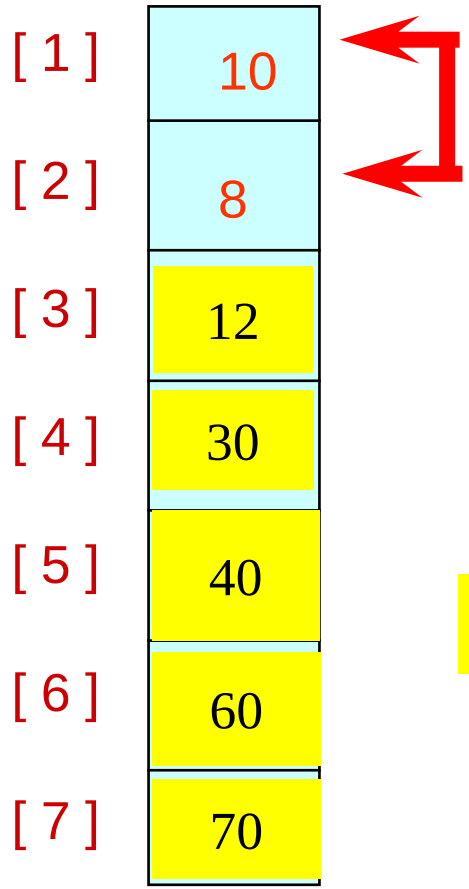


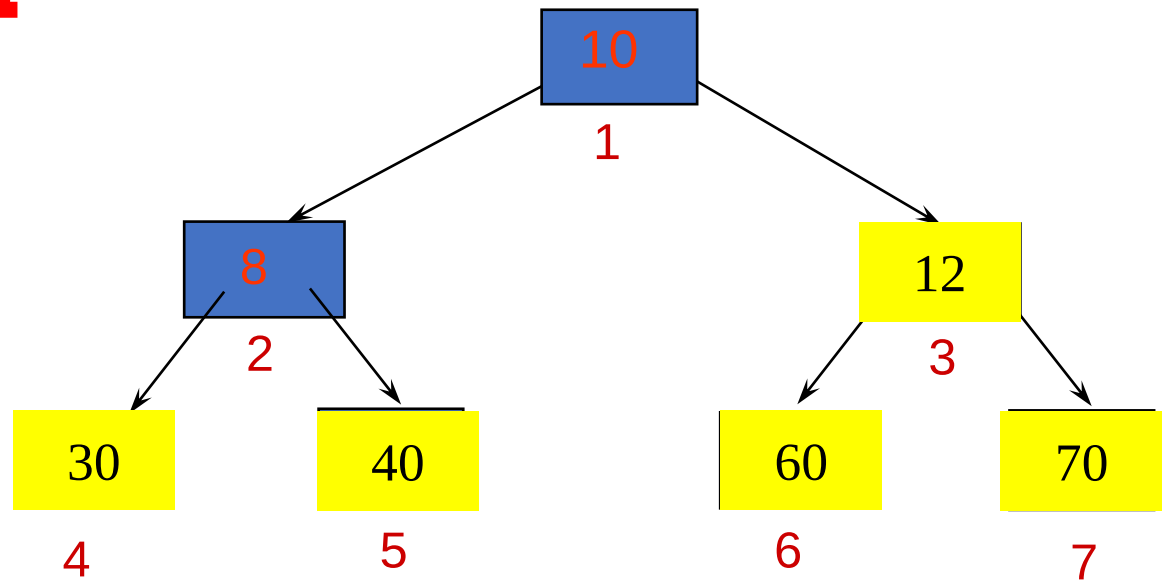
堆的重新调整 – 5



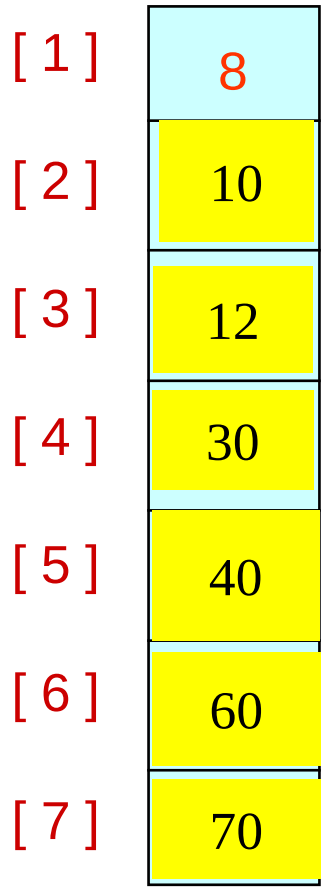


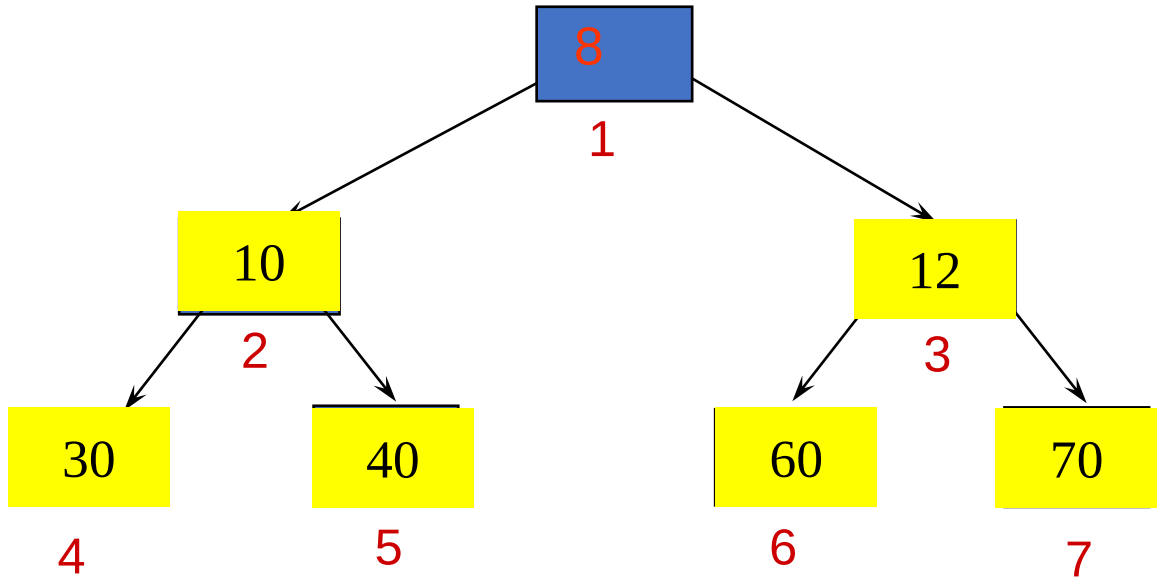
堆的重新调整 – 6





堆的重新调整 – 6





算法分析

时间效率：$O(n\log_2 n)$

空间效率：O（1）

稳 定 性：不稳定

适用于n 较大的情况

8.5 归并排序

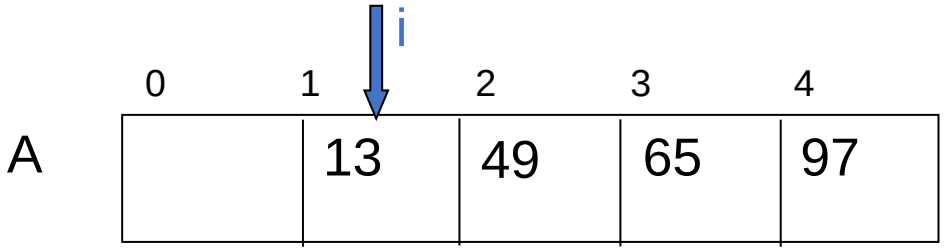
归并：将两个或两个以上的有序表组合成一个新有序表

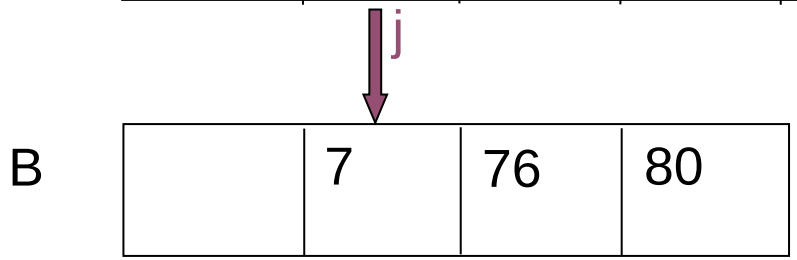
2-路归并排序

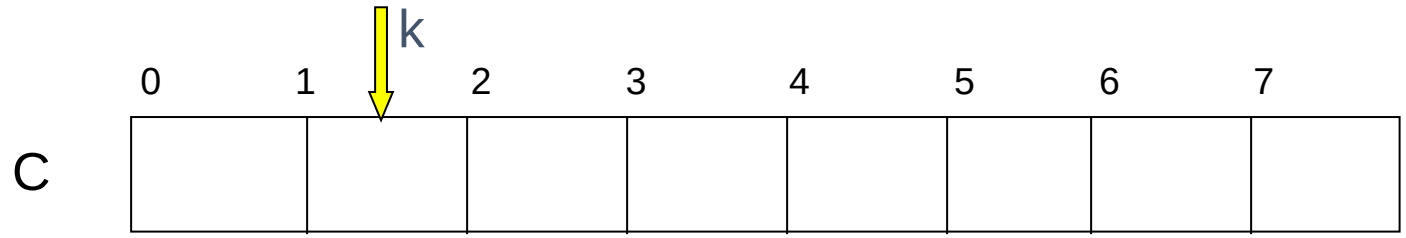
排序过程

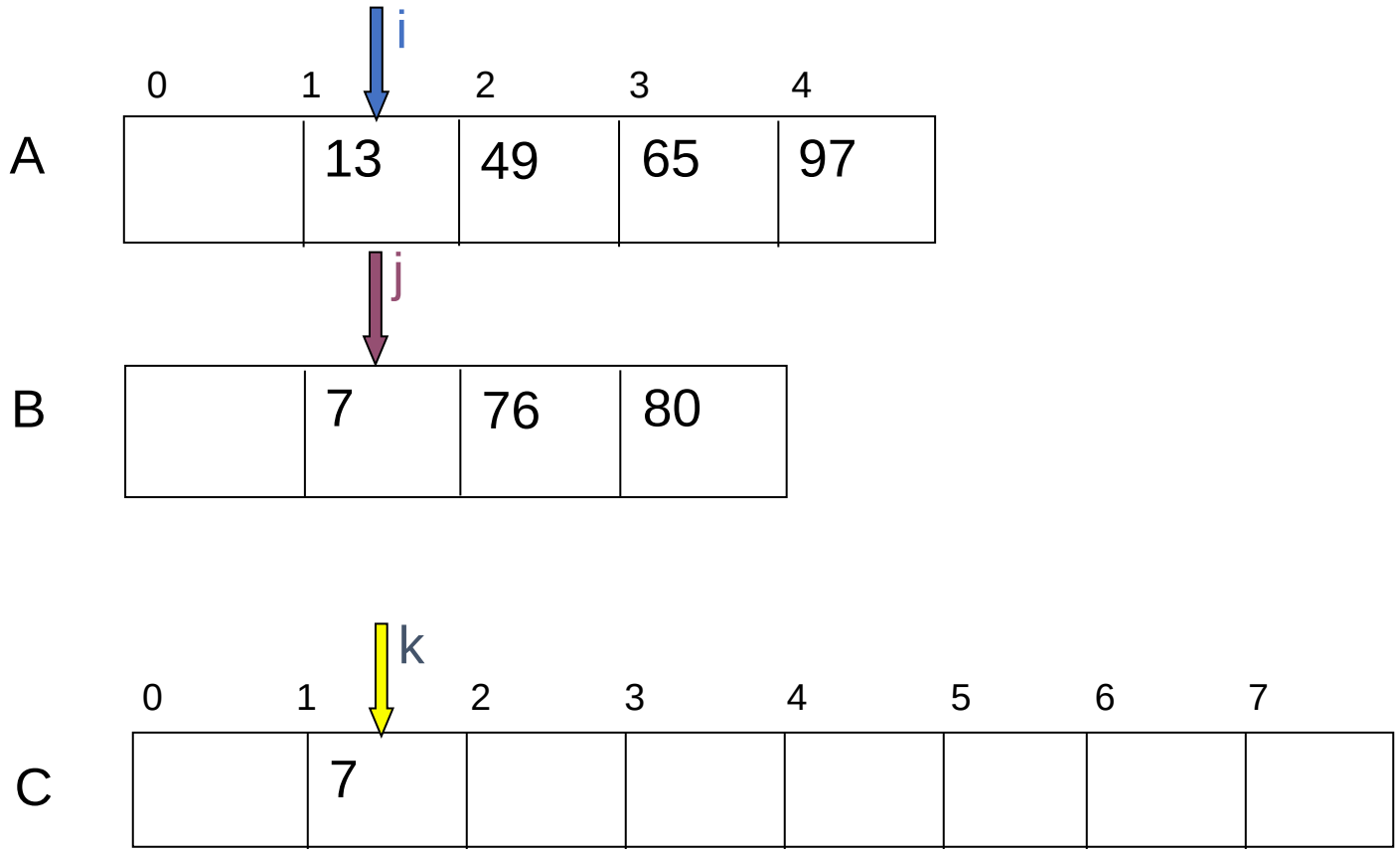
- ✓ 初始序列看成n个有序子序列，每个子序列长度为1
- ✓ 两两合并，得到$\lfloor n/2 \rfloor$个长度为2或1的有序子序列
- ✓ 再两两合并，重复直至得到一个长度为n的有序序列为止

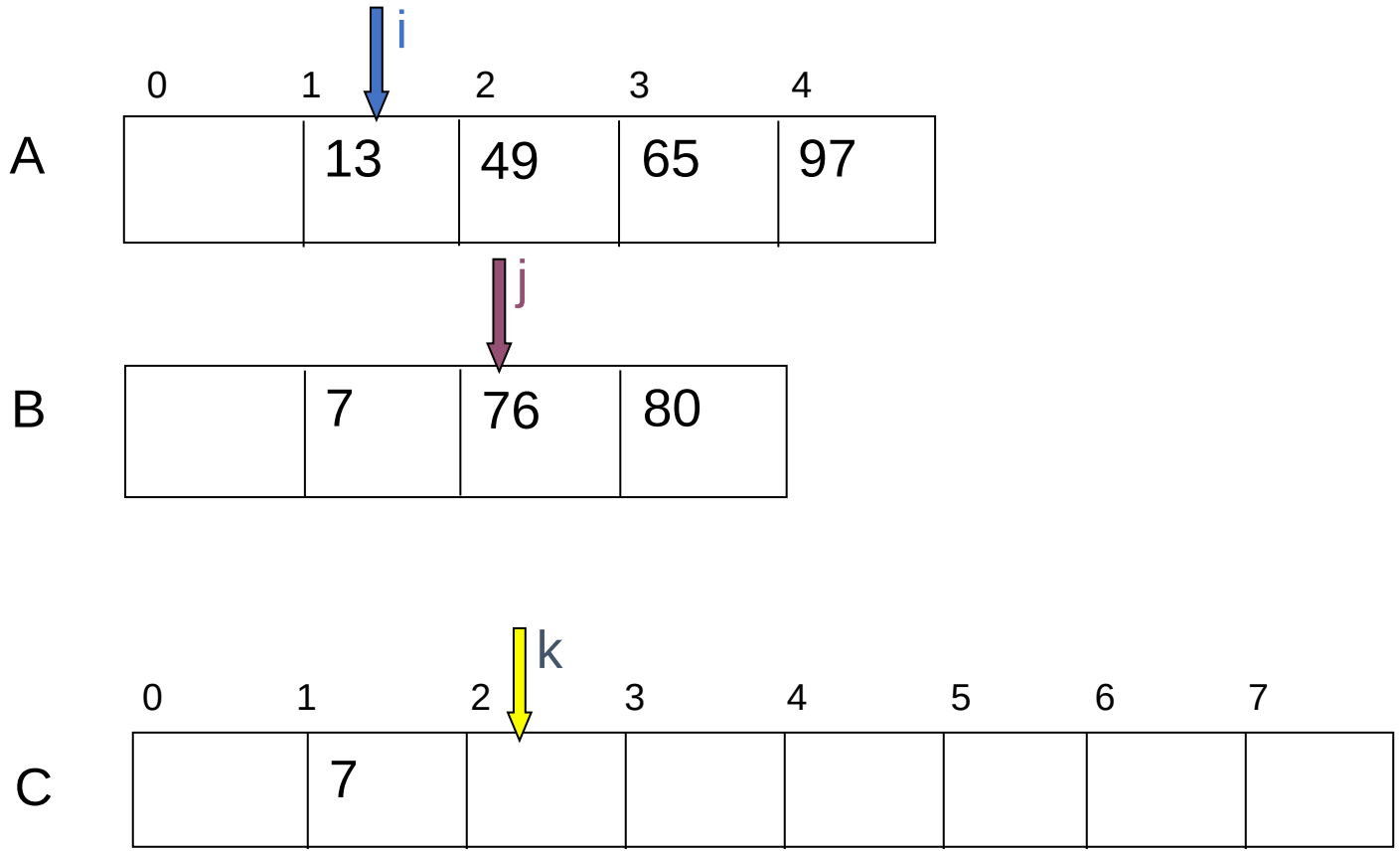
将两个顺序表合并成一个有序表

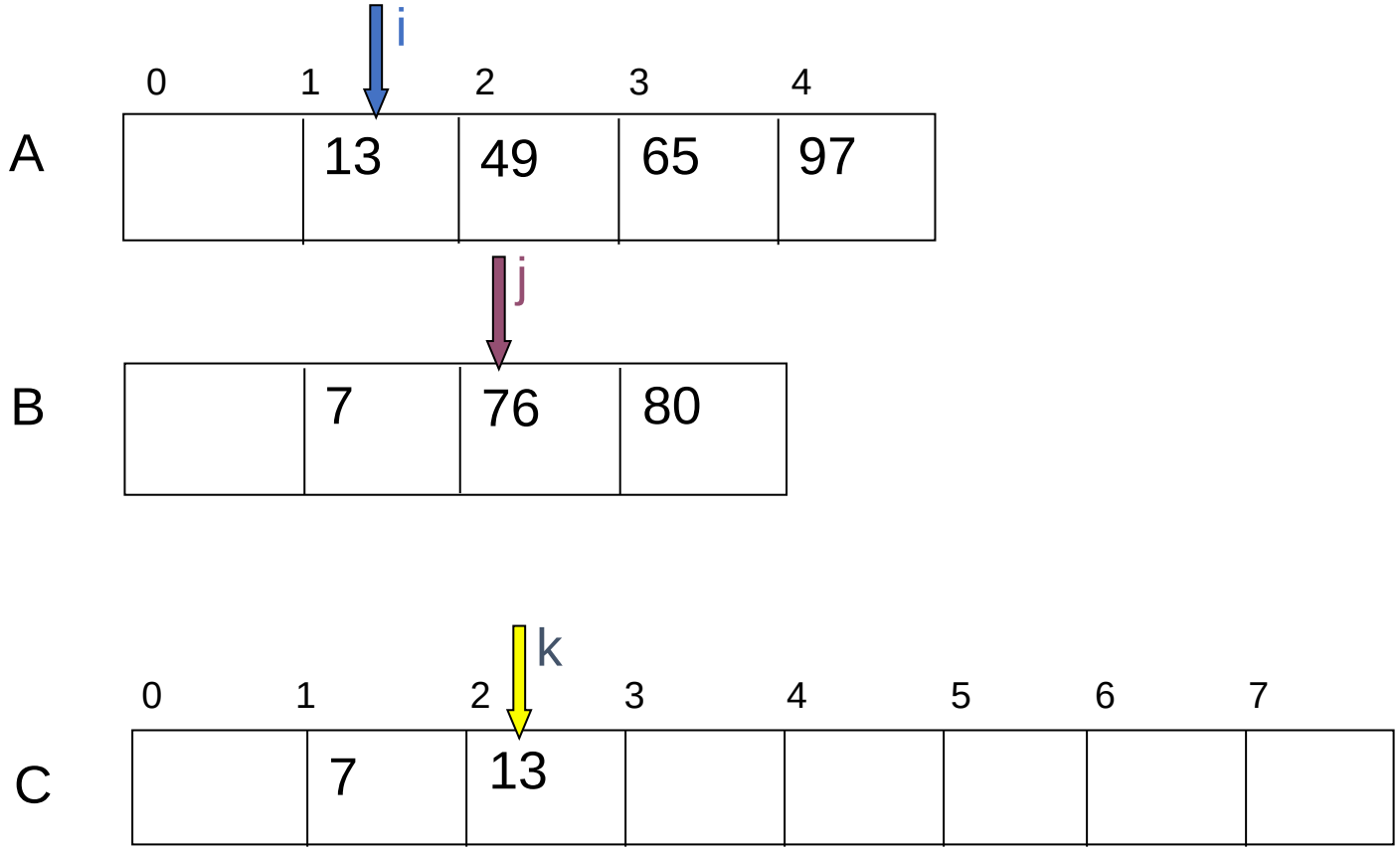


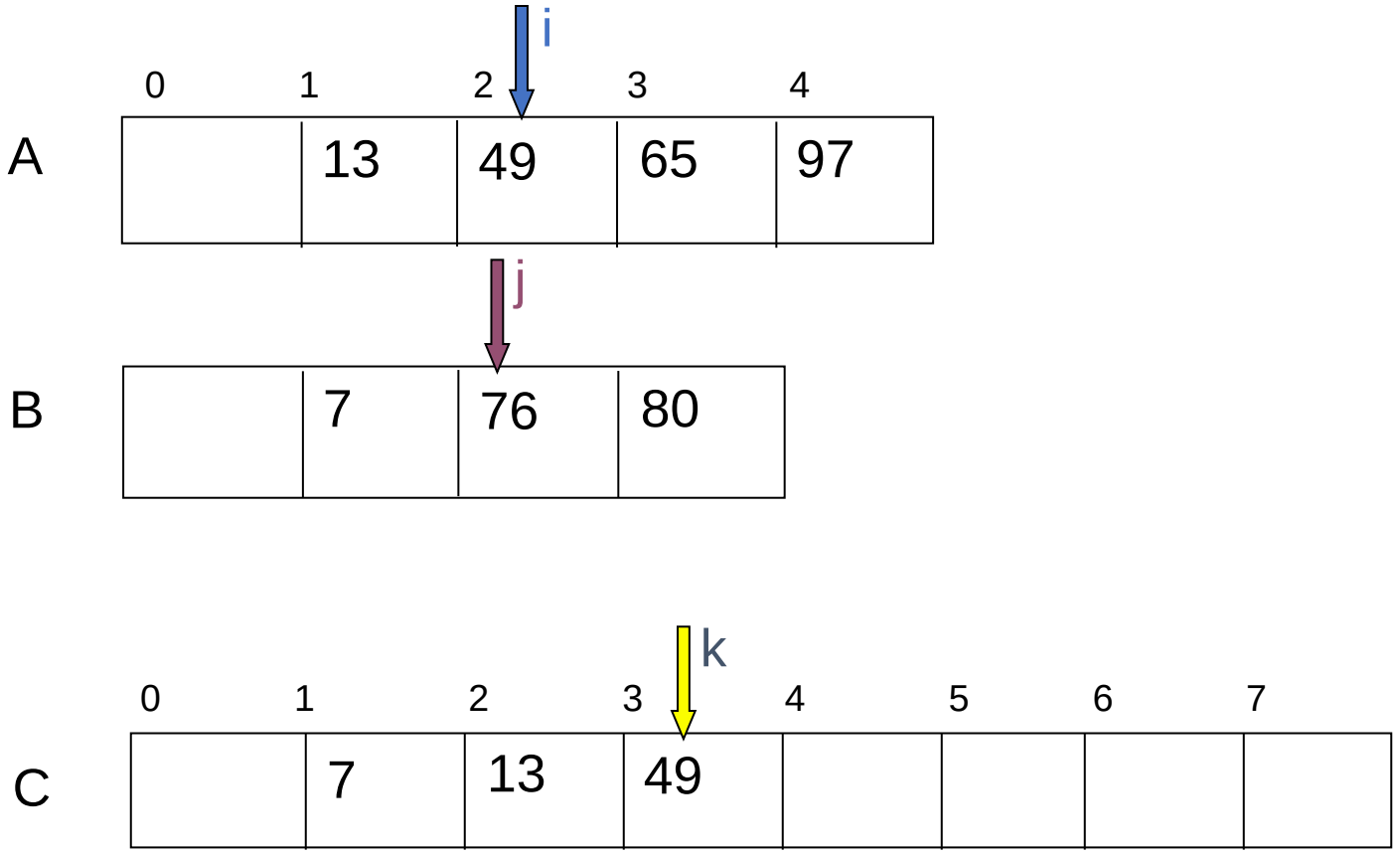


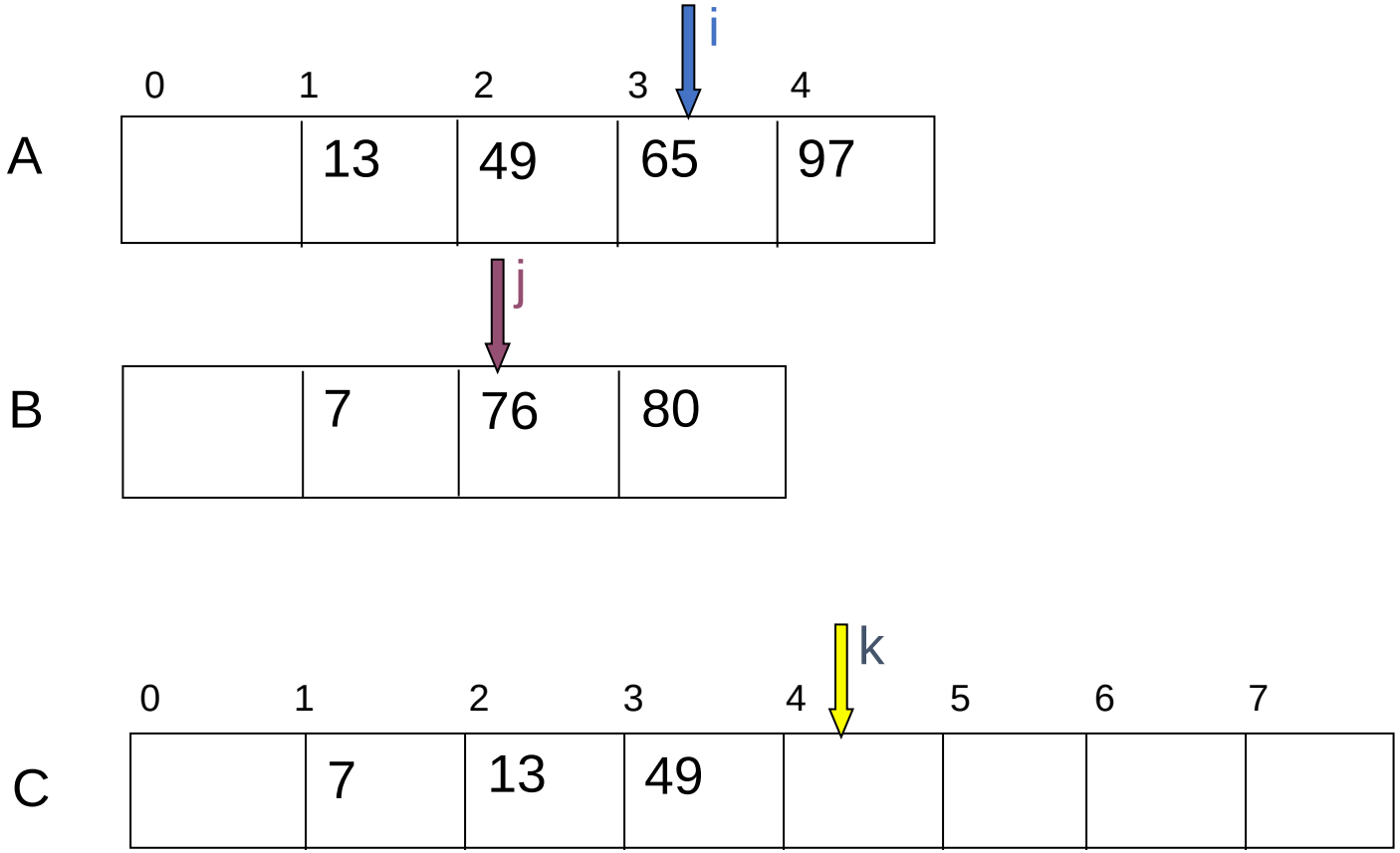


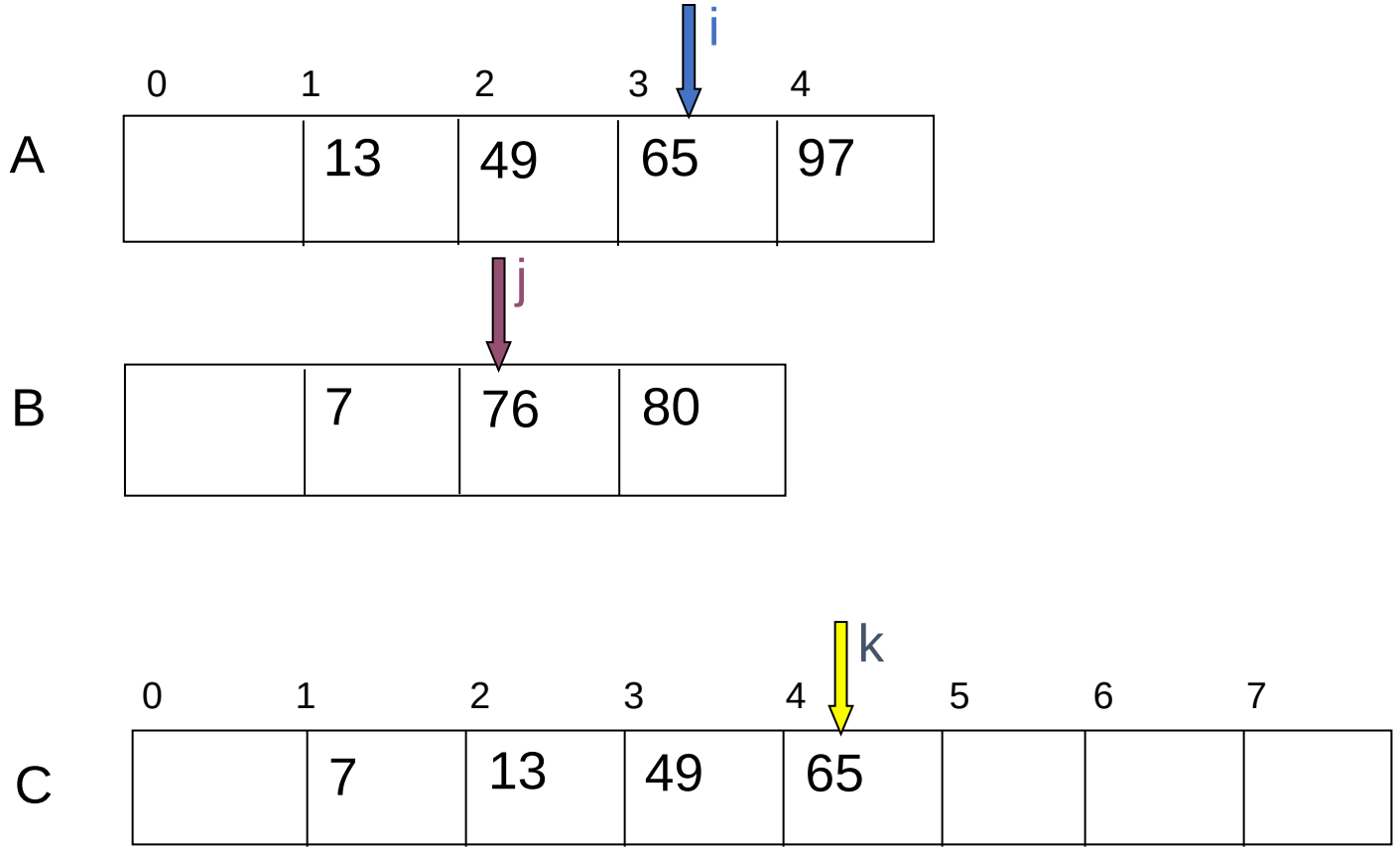

i
0
1
2
3
4
A
13
49
65
97
j
B
7
76
80
k
0
1
2
3
4
5
6
7
C
7

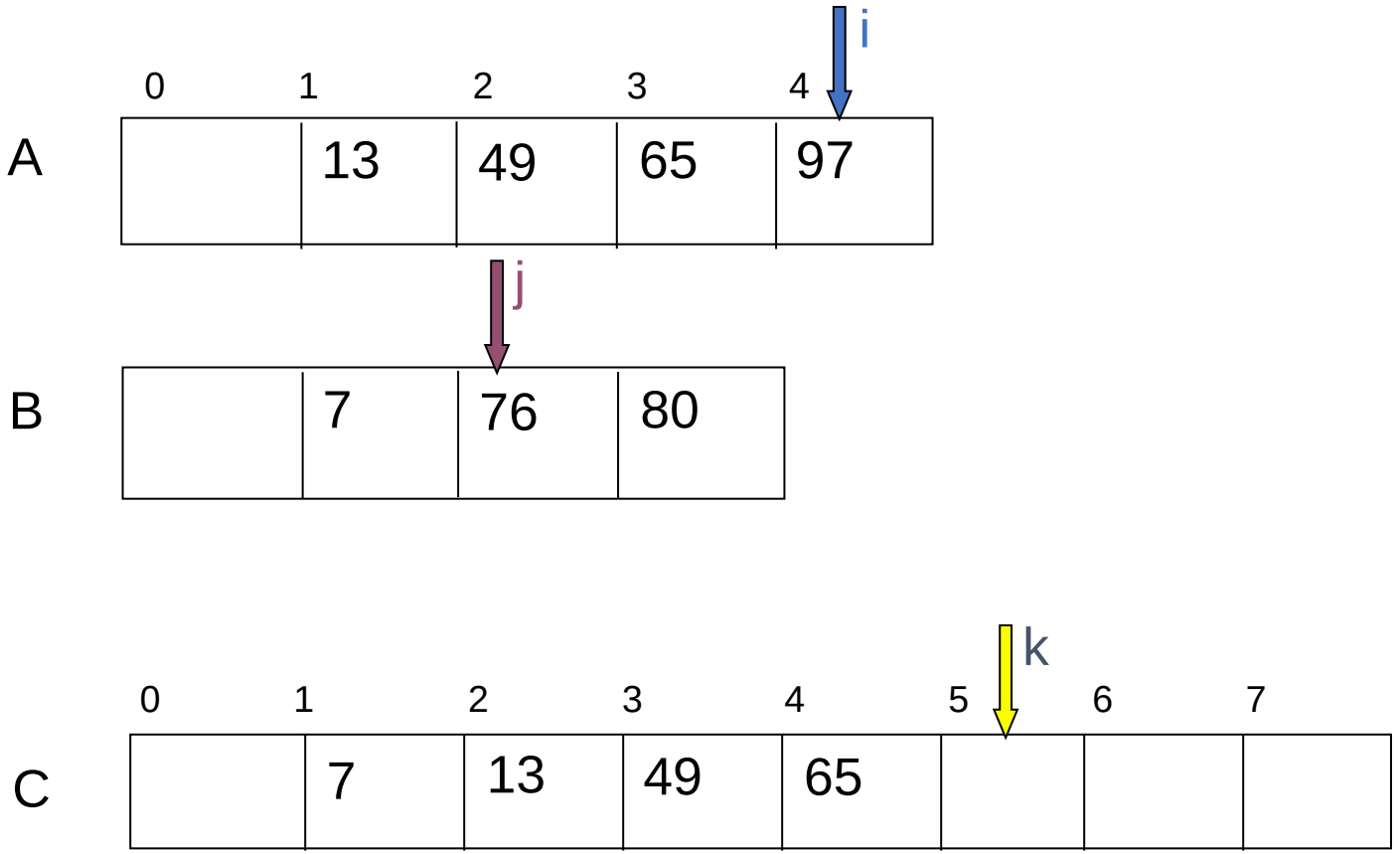

i
0
1
2
3
4
A
13
49
65
97
j
B
7
76
80
k
0
1
2
3
4
5
6
7
C
7

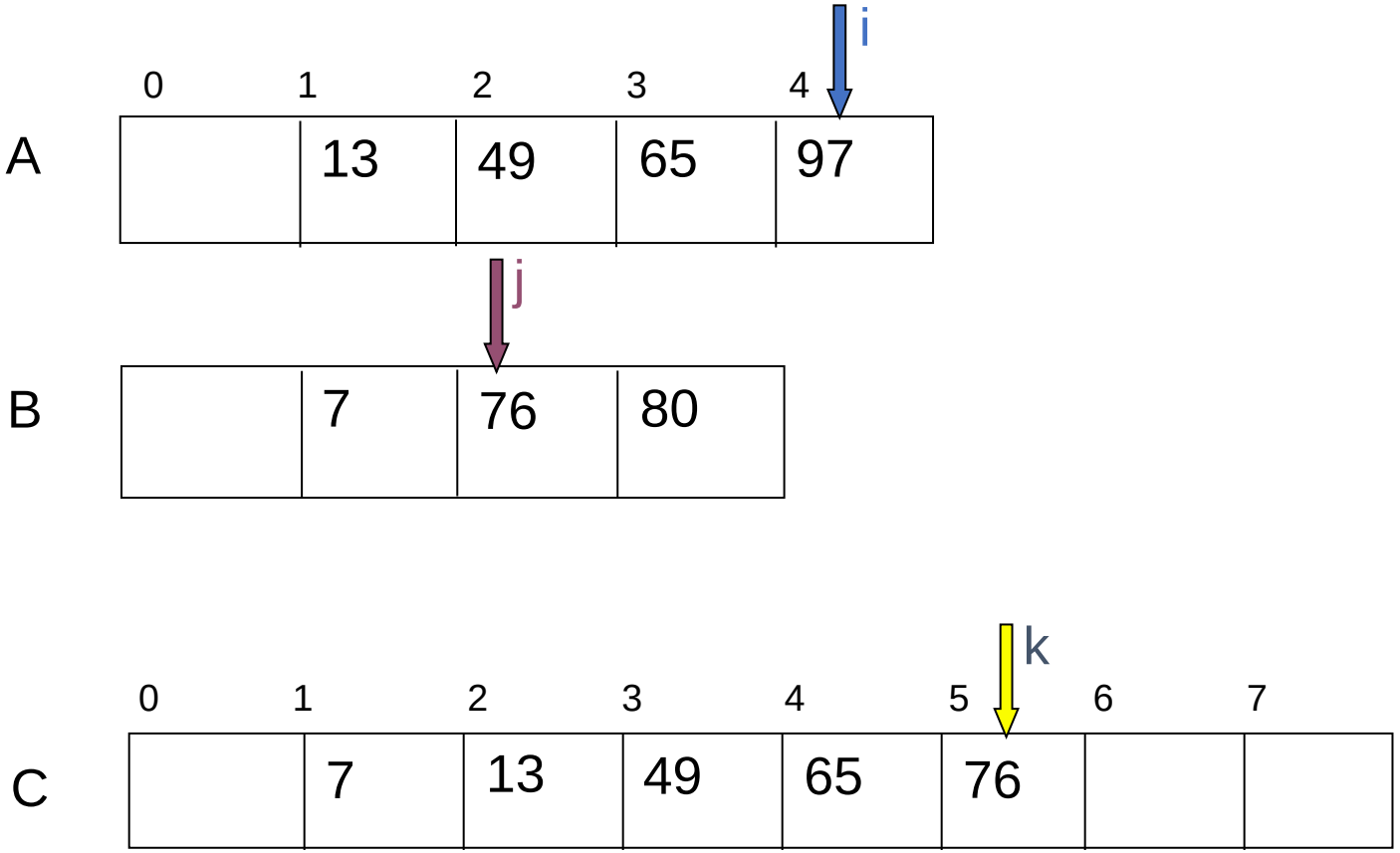

i
0
1
2
3
4
A
13
49
65
97
j
B
7
76
80
k
0
1
2
3
4
5
6
7
C
7
13

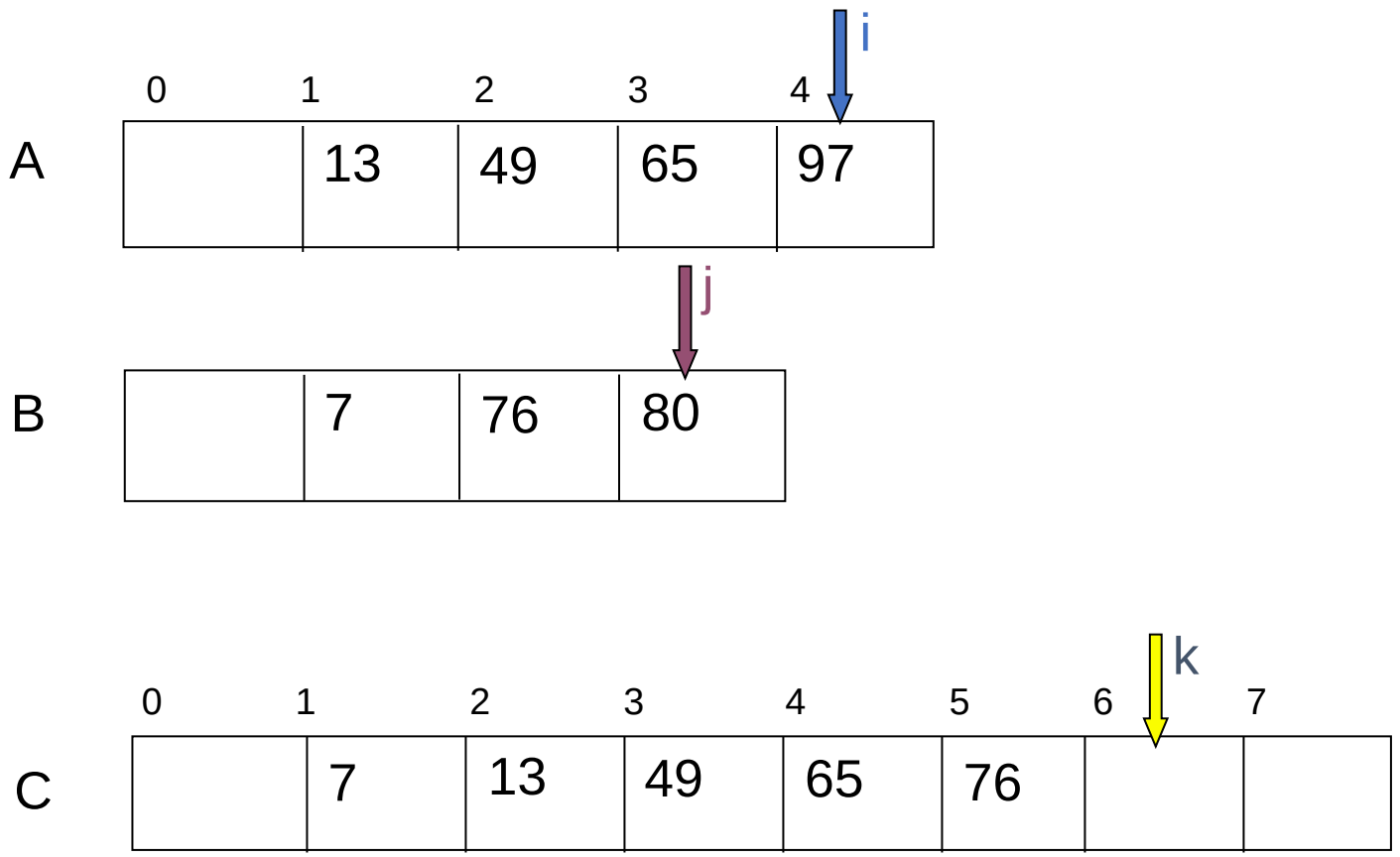

i
0
1
2
3
4
A
13
49
65
97
j
B
7
76
80
k
0
1
2
3
4
5
6
7
C
7
13
49

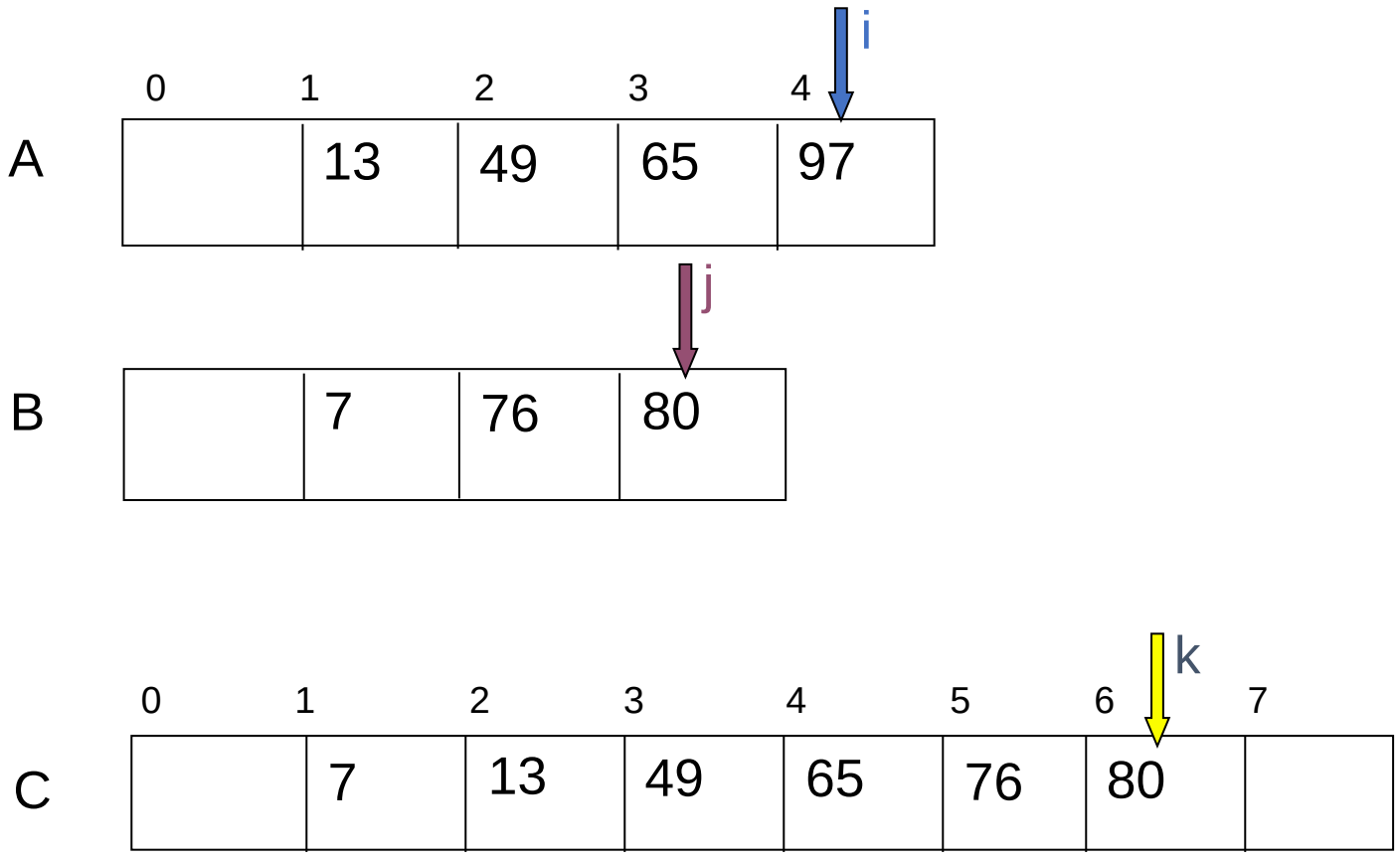

i
0
1
2
3
4
A
13
49
65
97
j
B
7
76
80
k
0
1
2
3
4
5
6
7
C
7
13
49

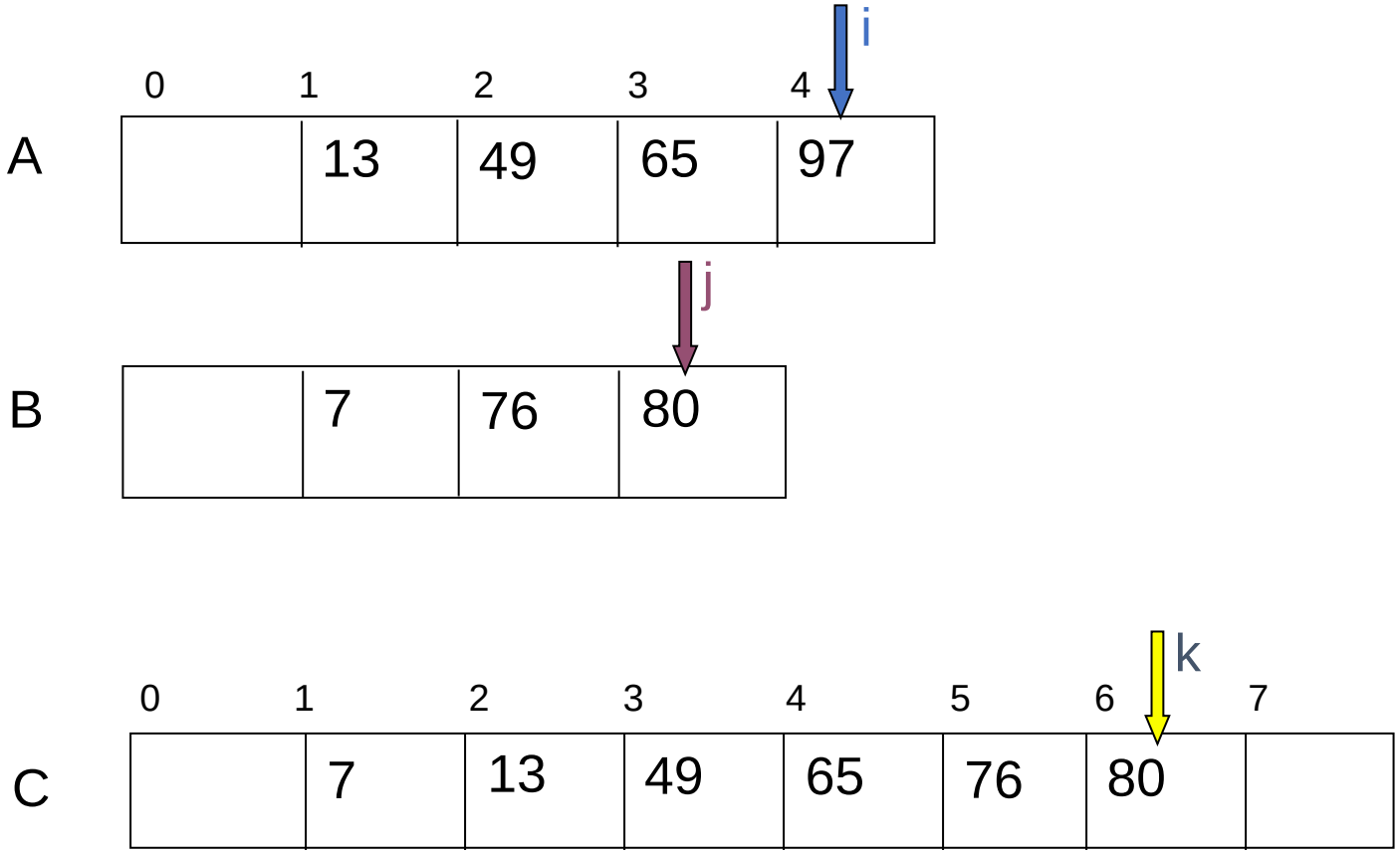

i
0
1
2
3
4
A
13
49
65
97
j
B
7
76
80
k
0
1
2
3
4
5
6
7
C
7
13
49
65

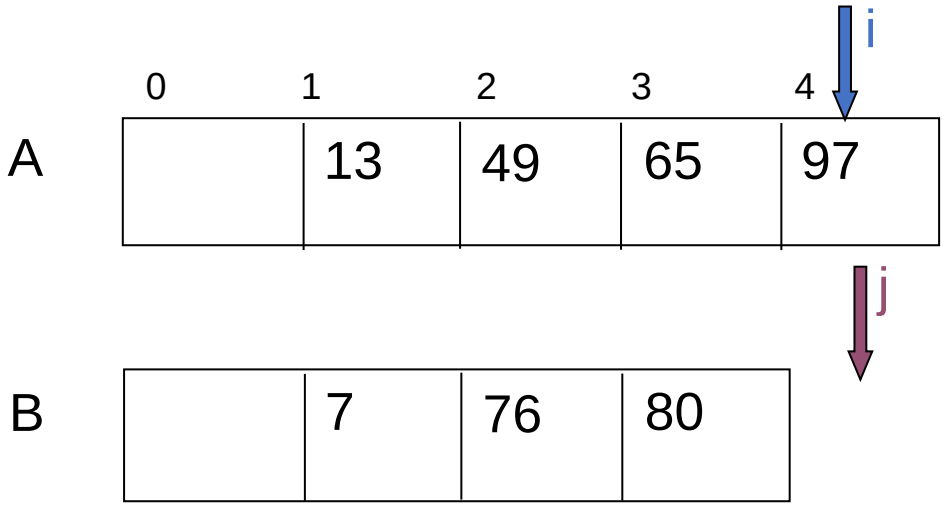

i
0
1
2
3
4
A
13
49
65
97
j
B
7
76
80
k
0
1
2
3
4
5
6
7
C
7
13
49
65


i
0
1
2
3
4
A
13
49
65
97
j
B
7
76
80
k
0
1
2
3
4
5
6
7
C
7
13
49
65
76

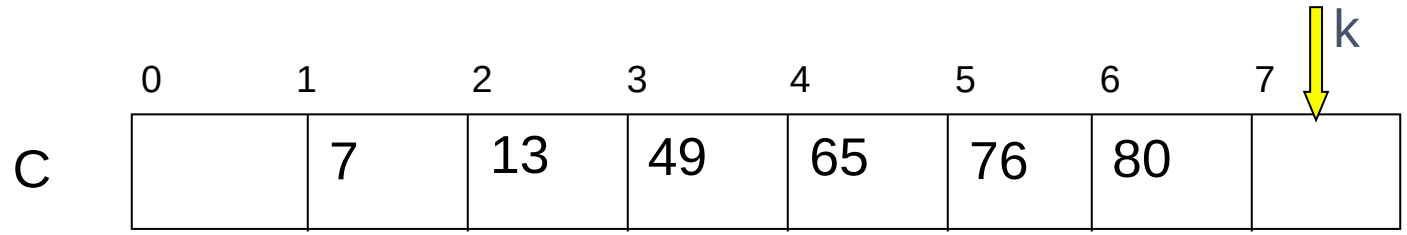

i
0
1
2
3
4
A
13
49
65
97
j
B
7
76
80
k
0
1
2
3
4
5
6
7
C
7
13
49
65
76

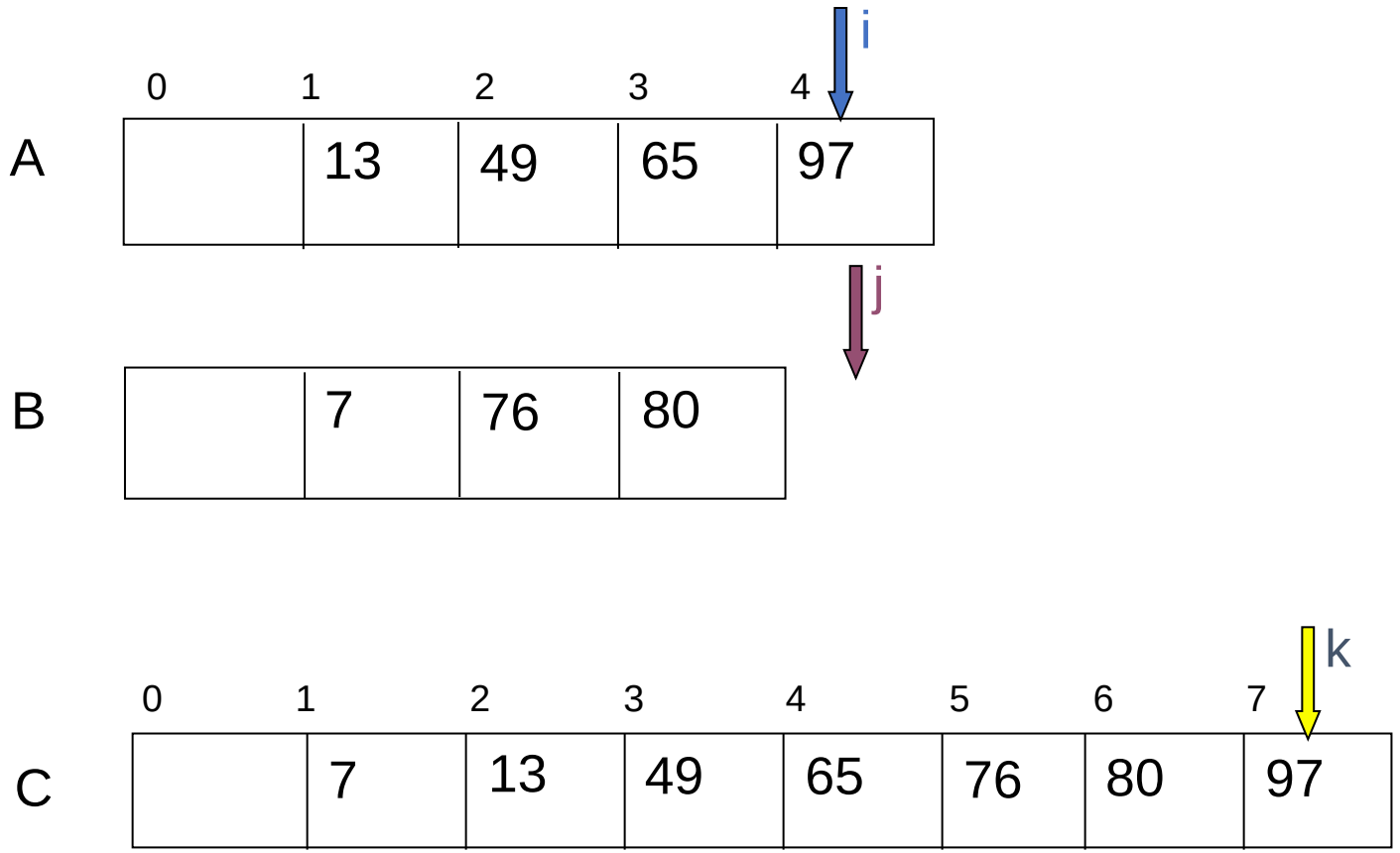

i
0
1
2
3
4
A
13
49
65
97
j
B
7
76
80
k
0
1
2
3
4
5
6
7
C
7
13
49
65
76
80


i
0
1
2
3
4
A
13
49
65
97
j
B
7
76
80
k
0
1
2
3
4
5
6
7
C
7
13
49
65
76
80



B 表的元素都已移入 C 表，只需将 A 表的剩余部分移入 C 表即可




i
0
1
2
3
4
A
13
49
65
97
j
B
7
76
80
k
0
1
2
3
4
5
6
7
C
7
13
49
65
76
80
97

例

初始关键字：　[49]　[38]　[65]　[97]　[76]　[13]　[27]

一趟归并后：　[38　49]　[65　97]　[13　76]　[27]

二趟归并后：　[38　49　65　97]　[13　27　76]

三趟归并后：　[13　27　38　49　65　76　97]

算法分析

时间效率：$O(n\log_2 n)$

空间效率：O（n）

稳 定 性：稳定

- 以扑克牌排序为例。每张扑克牌有两个“排序码”：花色和面值。其有序关系为：
  - 花色：♣ < ♦ < ♥ < ♠
  - 面值：2 < 3 < 4 < 5 < 6 < 7 < 8 < 9 < 10 < J < Q < K < A

  可以把所有扑克牌排成以下次序：

  ♣ 2, …, ♣ A, ♦ 2, …, ♦ A, ♥ 2, …, ♥ A, ♠ 2, …, ♠ A 花色相同的情况下，比较面值。