

第7章 内部排序

柳银萍

ypliu@cs.ecnu.edu.cn

1. 什么是排序？

将一组杂乱无章的**数据**按一定**规律**顺次排列起来。



存放在数据表中

The diagram illustrates the sorting process. A purple callout box on the left points to the word '数据' (data) in the definition above. A purple callout box on the right points to the word '规律' (rule) in the same definition. Below these boxes are two rounded rectangular boxes: '存放在数据表中' (Store in data table) under the left box and '按关键字排序' (Sort by key) under the right box.

按关键字排序

2. 排序的目的是什么？

——便于查找！

3. 什么叫内部排序？ 什么叫外部排序？



若待排序记录都在内存中，称为内部排序；



若待排序记录一部分在内存，一部分在外存，则称为外部排序。

注：外部排序时，要将数据分批调入内存来排序，中间结果还要及时存入外存，显然外部排序要复杂得多。

4. 排序算法的好坏如何衡量？



时间效率

比较次数与
移动次数

空间效率

占内存辅助
空间的大小

稳定性

A和B的关键字相等，
排序后A、B的先后次
序保持不变，则称这
种排序算法是稳定的。

▶▶▶ 排序算法分类

规则不同



插入排序
交换排序
选择排序
归并排序
基数排序

时间复杂度不同



简单排序 $O(n^2)$
先进排序 $O(n \log_2 n)$

一、插入排序

基本思想：

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。

即边插入边排序，保证子序列中随时都是排好序的。

▶▶▶ 插入排序

不同的具体实现方法导致不同的算法描述

最简单的排序法！

- 直接插入排序（基于顺序查找）
- 折半插入排序（基于折半查找）
- 希尔排序（基于逐趟缩小增量）

▶▶▶ 直接插入排序

例 (13, 6, 3, 31, 9, 27, 5, 11)

排序过程：整个排序过程为 $n-1$ 趟插入，即先将序列中第1个记录看成是一个有序子序列，然后从第2个记录开始，逐个进行插入，直至整个序列有序。

【13】 , 6, 3, 31, 9, 27, 5, 11

【6, 13】 , 3, 31, 9, 27, 5, 11

【3, 6, 13】 , 31, 9, 27, 5, 11

【3, 6, 13, 31】 , 9, 27, 5, 11

【3, 6, 9, 13, 31】 , 27, 5, 11

【3, 6, 9, 13, 27, 31】 , 5, 11

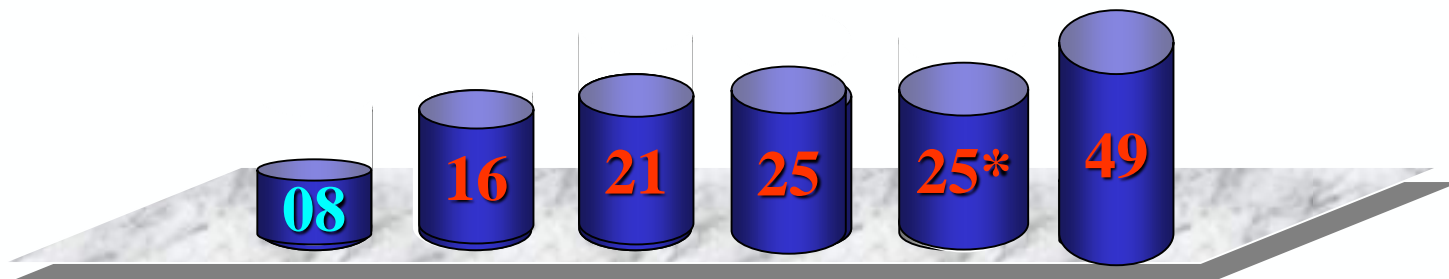
【3, 5, 6, 9, 13, 27, 31】 , 11

【3, 5, 6, 9, 11, 13, 27, 31】

直接插入排序

(21, 25, 49, 25*, 16, 08)

*表示后一个25



▶▶▶ 直接插入排序

```
def InsertSort(R):  
    for i in range(1,len(R)):  
        if R[i]<R[i-1]:  
            tmp=R[i]  
            j=i-1  
            while True:  
                R[j+1]=R[j]  
                j-=1  
                if j<0 or R[j]<=tmp:  
                    break  
            R[j+1]=tmp
```

#对R[0..n-1]按递增有序进行直接插入排序
#从元素R[1]开始
#反序时
#取出无序区的第一个元素
#有序区R[0..i-1]中向前找R[i]的插入位置
#将大于tmp的元素后移
#继续向前比较
#若j<0或者R[j]<=tmp,退出循环
#在j+1处插入R[i]

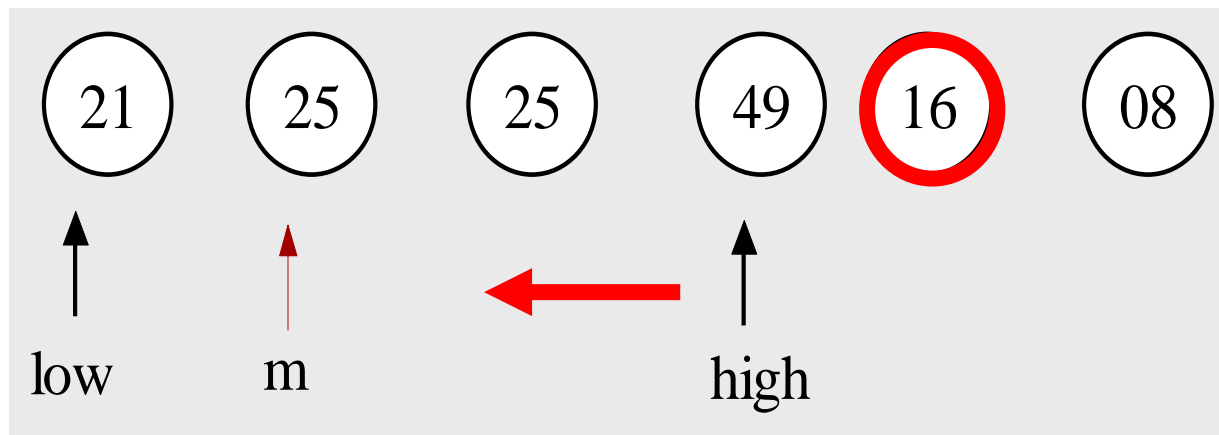
向前找到第一个 \leq tmp的R[j]，在其后插入tmp

- 时间复杂度为 $O(n^2)$
- 空间复杂度为 $O(1)$
- 是一种稳定的排序方法



在插入 $r[i]$ 时，利用折半查找法寻找 $r[i]$ 的插入位置

折半插入排序



▶▶▶ 折半插入排序

```
def BinInsertSort(R):  
    for i in range(1,len(R)):  
        if R[i]<R[i-1]:  
            tmp=R[i]  
            low,high=0,i-1  
            while low<=high:  
                mid=(low+high)//2  
                if tmp<R[mid]:  
                    high=mid-1  
                else:  
                    low=mid+1  
            for j in range(i-1,high,-1):  
                R[j+1]=R[j]  
            R[high+1]=tmp
```

#对R[0..n-1]按递增有序进行折半插入排序
#反序时
#将R[i]保存到tmp中
#在R[low..high]中折半查找插入位置high+1
#取中间位置
#插入点在左区间
#插入点在右区间
#元素集中后移
#插入原来的R[i]

- 减少了比较次数，但没有减少移动次数
- 平均性能优于直接插入排序

▶▶▶ 时间复杂度下界

对于下标 $i < j$, 如果 $A[i] > A[j]$, 则称 (i, j) 是一个逆序对。

例如：序列{34, 8, 64, 51, 32, 21}中有多少个逆序对？

交换两个相邻元素正好消去一个逆序对！

直接插入排序在基本有序时，效率较高

在待排序的记录个数较少时，效率较高

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：任何以交换相邻元素来排序的算法，其平均时间复杂度为 $\Omega(N^2)$ 。

基本思想:

- 先将整个待排记录序列分割成若干子序列, 分别进行直接插入排序, 待整个序列中的记录“基本有序”时, 再对全体记录进行一次直接插入排序。
- 希尔排序可以一次消去若干个逆序对!

技巧

子序列的构成不是简单地“逐段分割”，将相隔某个增量 d_k 的记录组成一个子序列，让增量 d_k 逐趟缩短（如依次取5, 3, 1）直到 $d_k=1$ 为止。

优点

小元素跳跃式前移，最后一趟增量为1时，序列已基本有序。平均性能优于直接插入排序。

希尔排序

例：关键字序列 $T=(49, 38, 65, 97, 76, 13, 27, 49^*, 55, 04)$

$r[i]$	0	1	2	3	4	5	6	7	8	9	10
初态：		49	38	65	97	76	13	27	49*	55	04
第1趟 ($d_k=5$)		13	27	49*	55	04	49	38	65	97	76
第2趟 ($d_k=3$)		13	04	49*	38	27	49	55	65	97	76
第3趟 ($d_k=1$)		04	13	27	38	49*	49	55	65	76	97

- ✓ d_k 值较大，子序列中对象较少，速度较快；
- ✓ d_k 值逐渐变小，子序列中对象变多，但大多数对象已基本有序，所以排序速度仍然很快。

希尔排序算法

取 $d_1=n/2$, $d_{i+1}=\lfloor d_i/2 \rfloor$ 时的希尔排序的算法

```
def ShellSort(R):  
    d=len(R)//2  
    while d>0:  
        for i in range(d,len(R)):  
            if R[i]<R[i-d]:  
                tmp=R[i]  
                j=i-d  
                while True:  
                    R[j+d]=R[j]                #将大于tmp的元素后移  
                    j=j+d                        #继续向前找  
                    if j<0 or R[j]<=tmp:  
                        break                    #若j<0或者R[j]<=tmp,退出循环  
                R[j+d]=tmp                      #在j+d处插入tmp  
        d=d//2                                #递减增量  
#对R[0..n-1]按递增有序进行希尔排序  
#增量置初值  
#对所有相隔d位置的元素组采用直接插入排序  
#反序时
```

- 时间复杂度是 n 和 d 的函数：

$$O(n^{1.25}) \sim O(1.6n^{1.25}) \text{ —经验公式}$$

- 空间复杂度为 $o(1)$
- 是一种不稳定的排序方法

- ✓ 如何选择最佳 d 序列，目前尚未解决
- ✓ 最后一个增量值必须为1，无除1以外的公因子
- ✓ 不宜在链式存储结构上实现

二、交换排序

交换排序

基本思想：

两两比较，如果发生逆序则交换，直到所有记录都排好序为止。



冒泡排序 $O(n^2)$

快速排序 $O(n \log_2 n)$

冒泡排序

基本思想：

每趟不断将记录两两比较，并按“前小后大”规则交换

21, 25, 49, 25*, 16, 08

21, 25, 25*, 16, 08, 49

21, 25, 16, 08, 25*, 49

21, 16, 08, 25, 25*, 49

16, 08, 21, 25, 25*, 49

08, 16, 21, 25, 25*, 49



Two vertical bars, one red and one blue, are positioned on the left side of the slide.

优点：

每趟结束时，不仅能挤出一个最大值到最后面位置，还能同时部分理顺其他元素；

一旦下趟没有交换，还可提前结束排序

►►► 冒泡排序

在冒泡排序算法，若某一趟没有出现任何元素交换，说明所有元素已排好序了，就可以结束本算法。

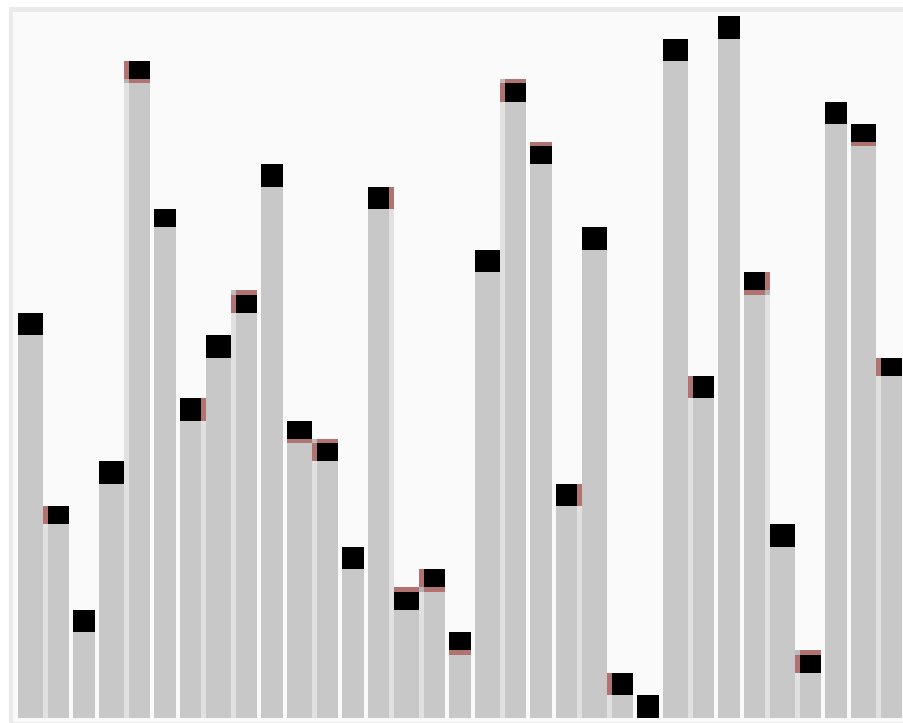
```
def BubbleSort(R):  
    for i in range(len(R)-1):  
        exchange=False  
        for j in range(len(R)-1,i,-1):  
            if R[j]<R[j-1]:  
                R[j],R[j-1]=R[j-1],R[j]  
                exchange=True  
        if exchange==False: return
```

#对R[0..n-1]按递增有序进行冒泡排序
#本趟前将exchange置为False
#一趟中找出最小关键字的元素
#反序时交换
#R[j]和R[j-1]交换,将最小元素前移
#本趟发生交换置exchange为True
#本趟没有发生交换，中途结束算法

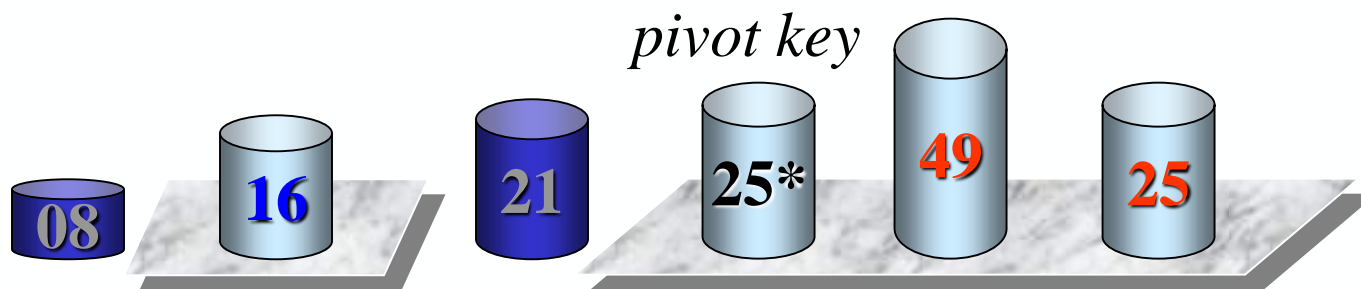
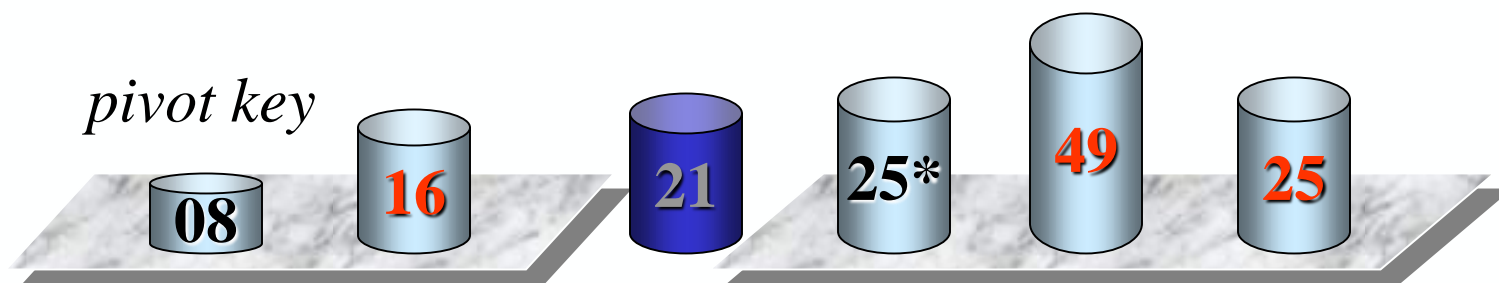
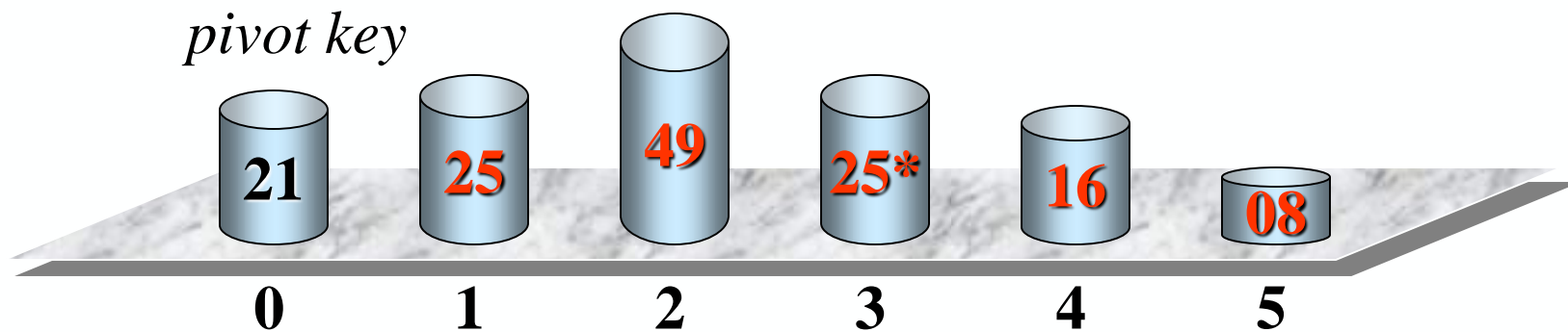
- ◆ 时间复杂度为 $O(n^2)$
- ◆ 空间复杂度为 $O(1)$
- ◆ 是一种稳定的排序方法
- ◆ 冒泡排序是唯一一种可在顺序存储或链式存储结构上进行排序的算法。

基本思想：

- 任取一个元素为主元；
- 所有比它小的元素放主元前，比它大的元素放主元后，形成左右两个子表；
- 对各子表重新选择主元并依此规则调整，直到每个子表的元素只剩一个。



快速排序



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
----	--	----	----	----	----	----	----	----

pivot key 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
----	--	----	----	----	----	----	----	----

pivot key



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
----	----	----	----	----	----	----	--	----

pivot key

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
----	----	----	----	----	----	----	--	----

pivot key



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
----	----	----	----	----	----	----	--	----

pivot key



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
----	----	----	--	----	----	----	----	----

pivot key



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
----	----	----	--	----	----	----	----	----

pivot key

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
----	----	----	----	----	----	--	----	----

pivot key



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
----	----	----	----	----	----	--	----	----

pivot key



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
----	----	----	----	--	----	----	----	----

pivot key



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
----	----	----	----	--	----	----	----	----

pivot key



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
----	----	----	----	--	----	----	----	----

pivot key



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
--	----	----	----	----	----	----	----	----

pivot key



low high



第一趟结束后两个子序列，可递归

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
----	--	----	----	----	----	----	----	----

pivot key

low

high



每一趟的子表的形成
是采用从两头向中间
交替式逼近法；



由于每趟中对各子表
的操作都相似，可采
用递归算法。

快速排序

```
def Partition(R,s,t):  
    i,j=s,t  
    base=R[s]  
    while i!=j:  
        while j>i and R[j]>=base:  
            j-=1  
        if j>i:  
            R[i]=R[j]  
            i+=1  
        while i<j and R[i]<=base:  
            i+=1  
        if i<j:  
            R[j]=R[i]  
            j-=1  
    R[i]=base  
    return i
```

#划分算法2

#以表首元素为基准

#从表两端交替向中间遍历,直至*i=j*为止

#从后向前遍历,找一个小于基准的R[j]

#R[j]前移覆盖R[i]

#从前向后遍历,找一个大于基准的R[i]

#R[i]后移覆盖R[j]

#基准归位

#返回归位的位置

快速排序

```
def QuickSort(R):  
    QuickSort1(R,0,len(R)-1)  
  
def QuickSort1(R,s,t):  
    if s<t:  
        i=Partition (R,s,t)  
        QuickSort1(R,s,i-1)  
        QuickSort1(R,i+1,t)
```

#对R[0..n-1]的元素按递增进行快速排序

#对R[s..t]的元素进行快速排序
#表中至少存在两个元素的情况
#可以使用前面3种划分算法中的任意一种
#对左子表递归排序
#对右子表递归排序



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：

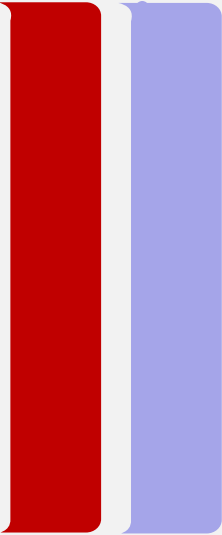
不稳定 — 可选任一元素为支点。

思考：

- 主元怎么选？随机取 `rand()`；取头、中、尾的中位数。
- 主元怎么选，快速排序效率很差？

三、选择排序

基本思想

Two vertical bars, one red and one blue, are positioned on the left side of the slide.

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.

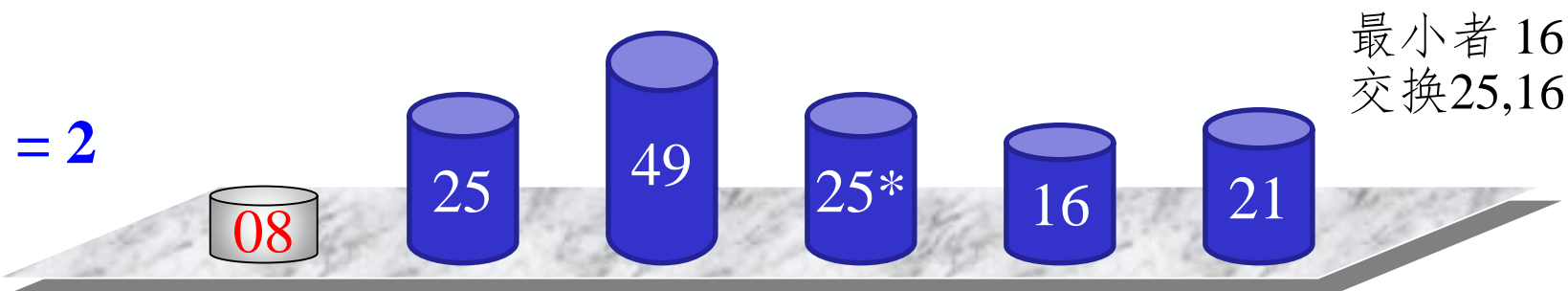
- 选择排序 $O(n^2)$
- 堆排序 $O(n\log n)$

简单选择排序

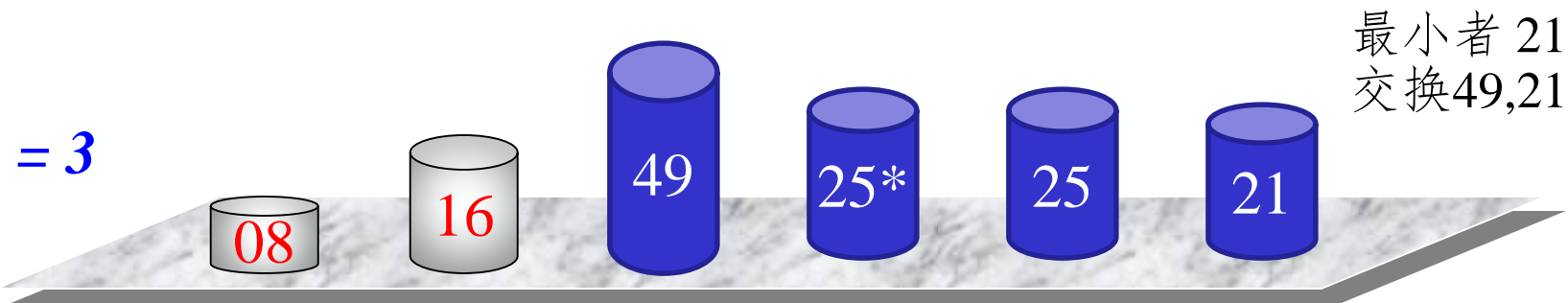
$i = 1$



$i = 2$



$i = 3$



▶▶▶ 简单选择排序

```
def SelectSort(R):  
    for i in range(len(R)-1):  
        minj=i  
        for j in range(i+1,len(R)):  
            if R[j]<R[minj]:  
                minj=j  
  
        if minj!=i:  
            R[i],R[minj]=R[minj],R[i]
```

```
#对R[0..n-1]元素进行简单选择排序  
#做第i趟排序  
#minj先置为区间中的首元素序号  
#从R[i..n-1]中选最小元素的R[minj]  
#与区间中其他元素比较  
  
#R[minj]不是无序区首元素  
#交换R[i]和R[minj]
```

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: **$O(1)$**

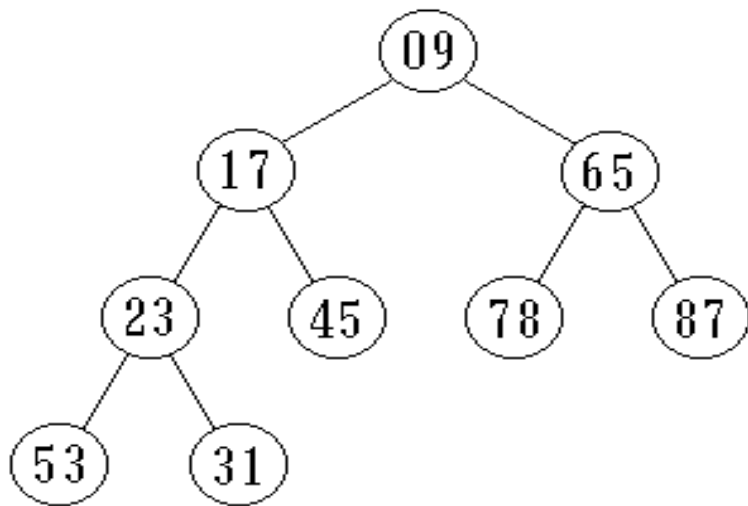
稳定

堆排序

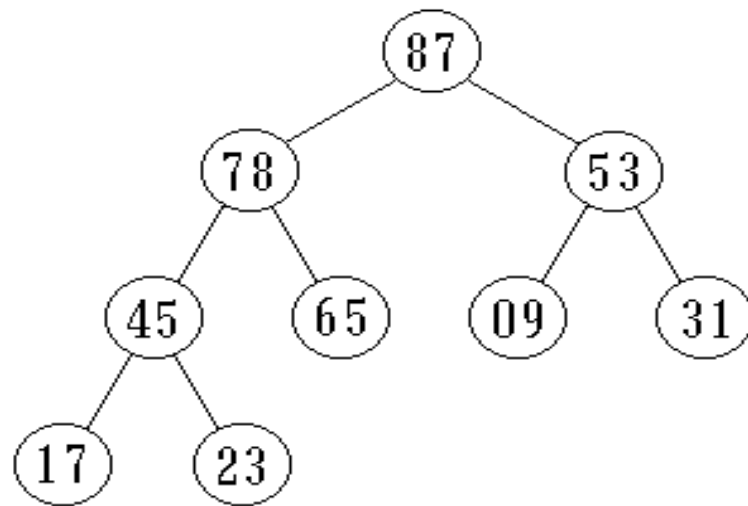
- 利用树的结构特征来描述堆，树只是作为堆的描述工具，堆实际是存放在线形结构中。

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

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



(a) 最小堆

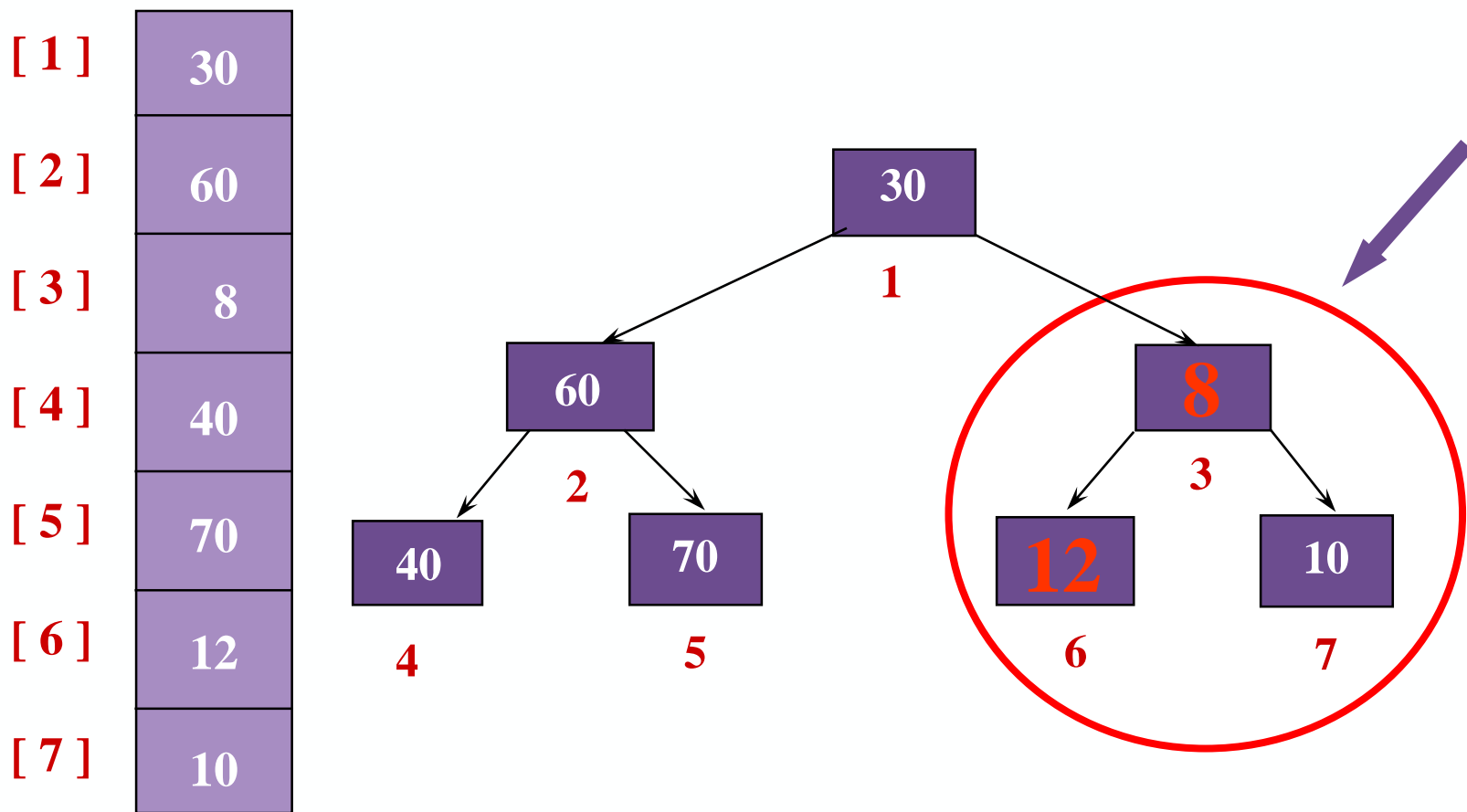


(b) 最大堆

堆顶元素（根）为最小值或最大值

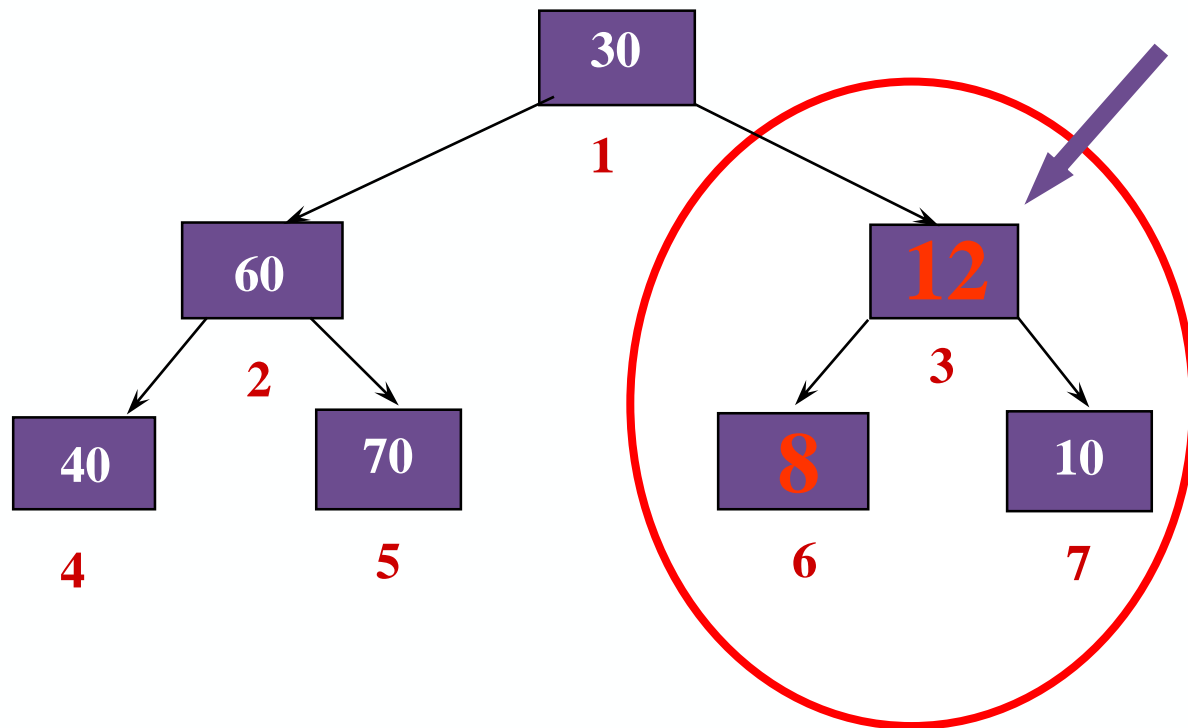
▶▶▶ 无序序列建堆

假设建立最大堆!

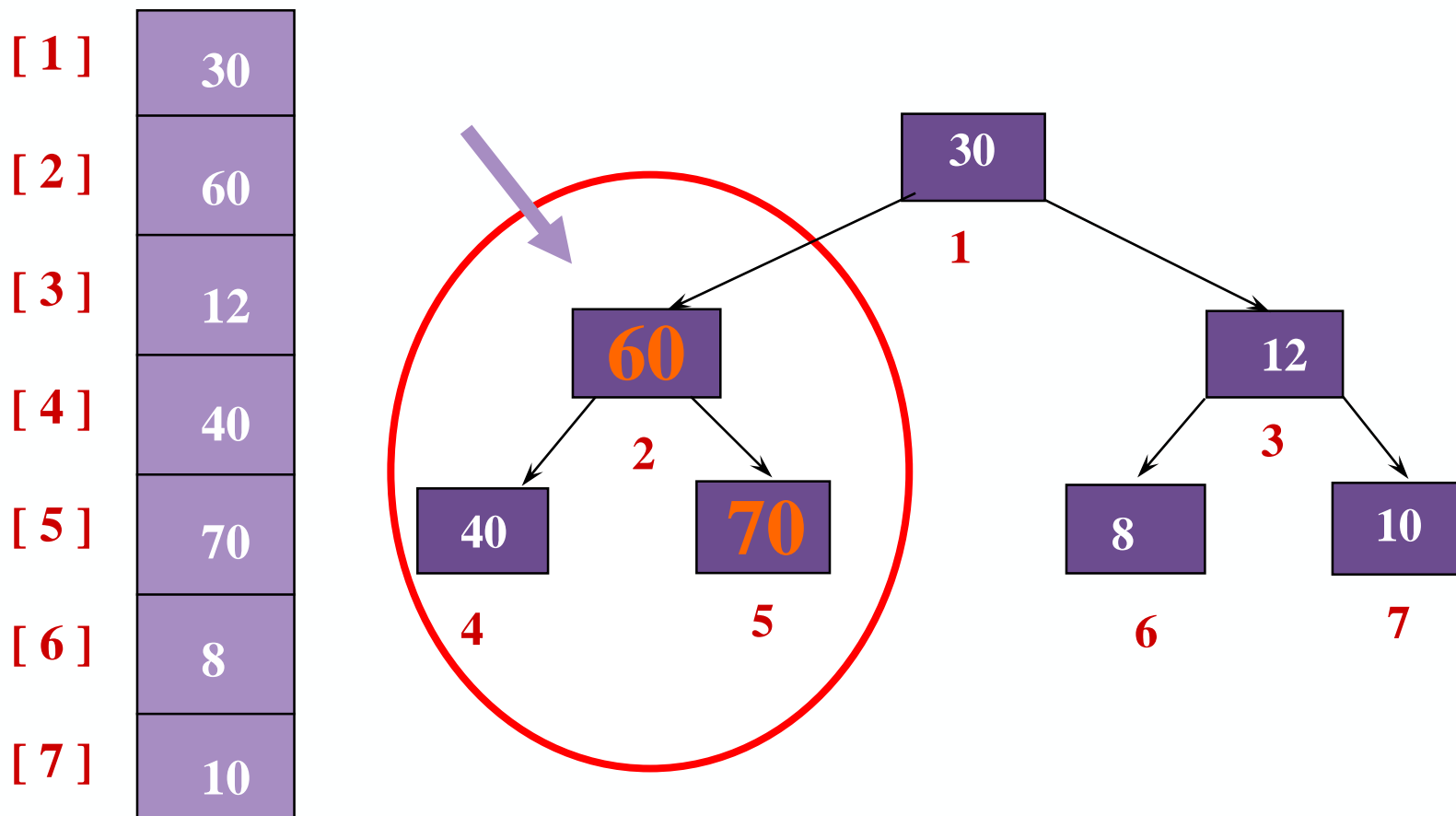


无序列建堆

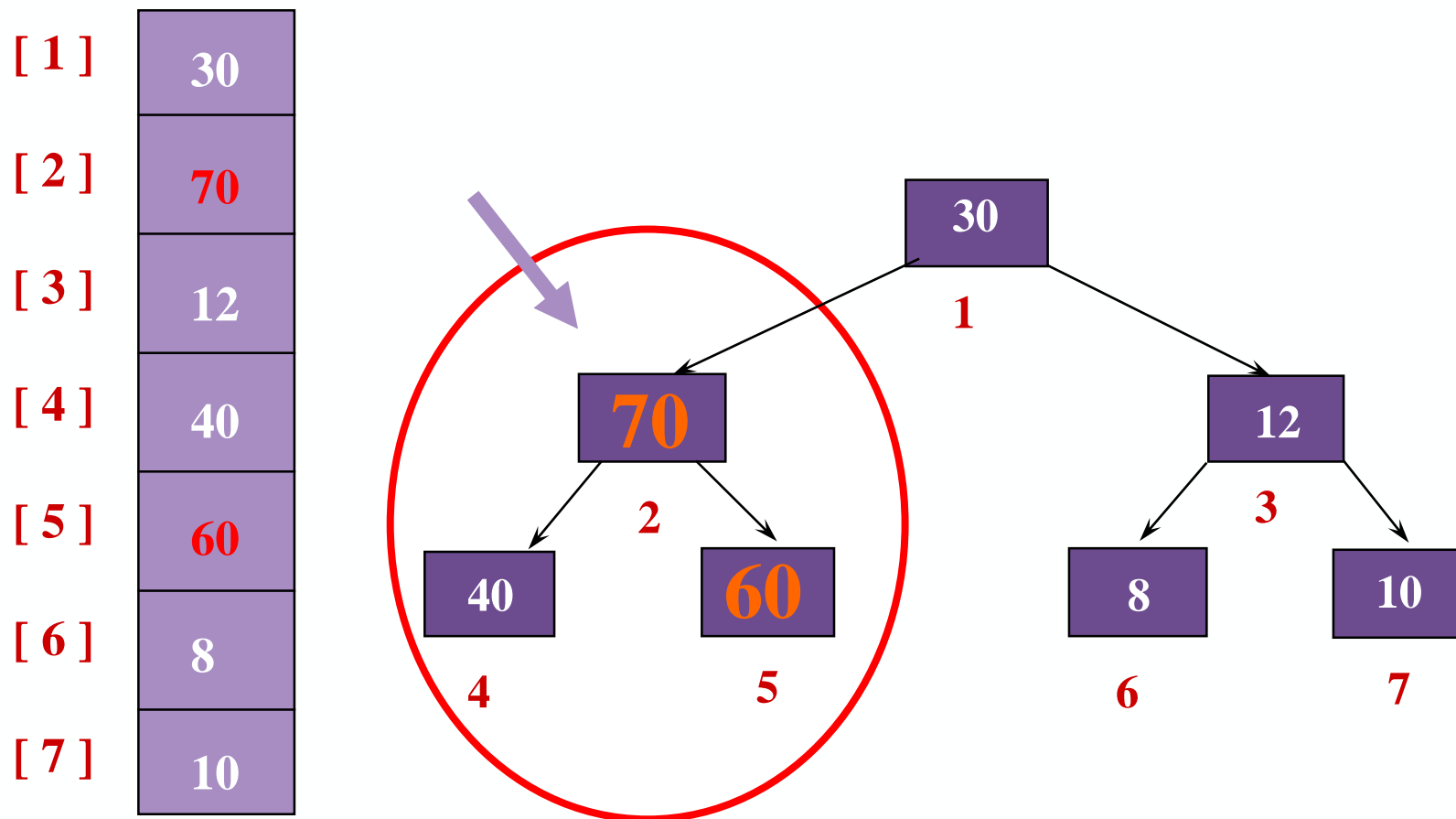
[1]	30
[2]	60
[3]	12
[4]	40
[5]	70
[6]	8
[7]	10



无序序列建堆

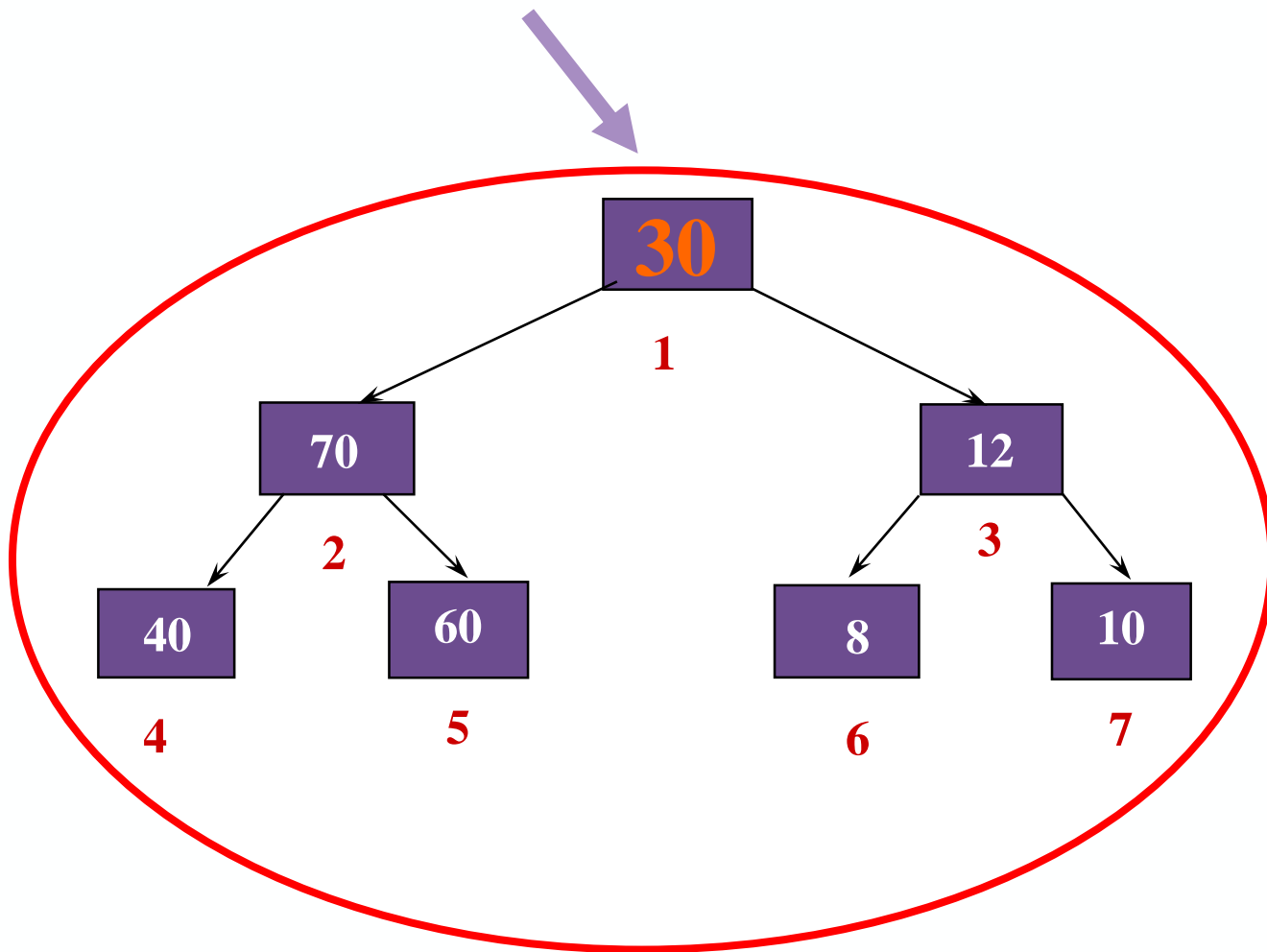


▶▶▶ 无序序列建堆



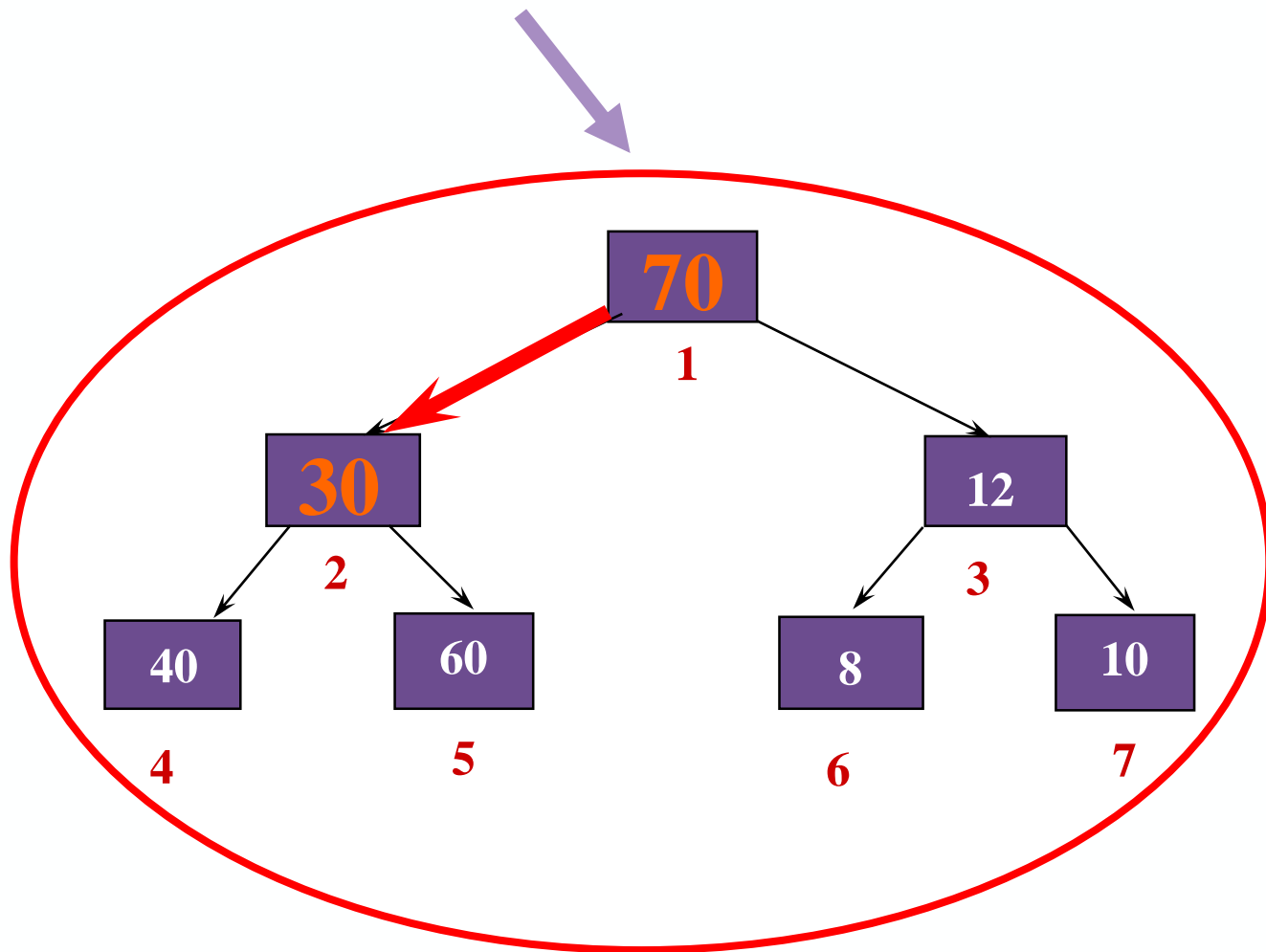
▶▶▶ 无序序列建堆

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



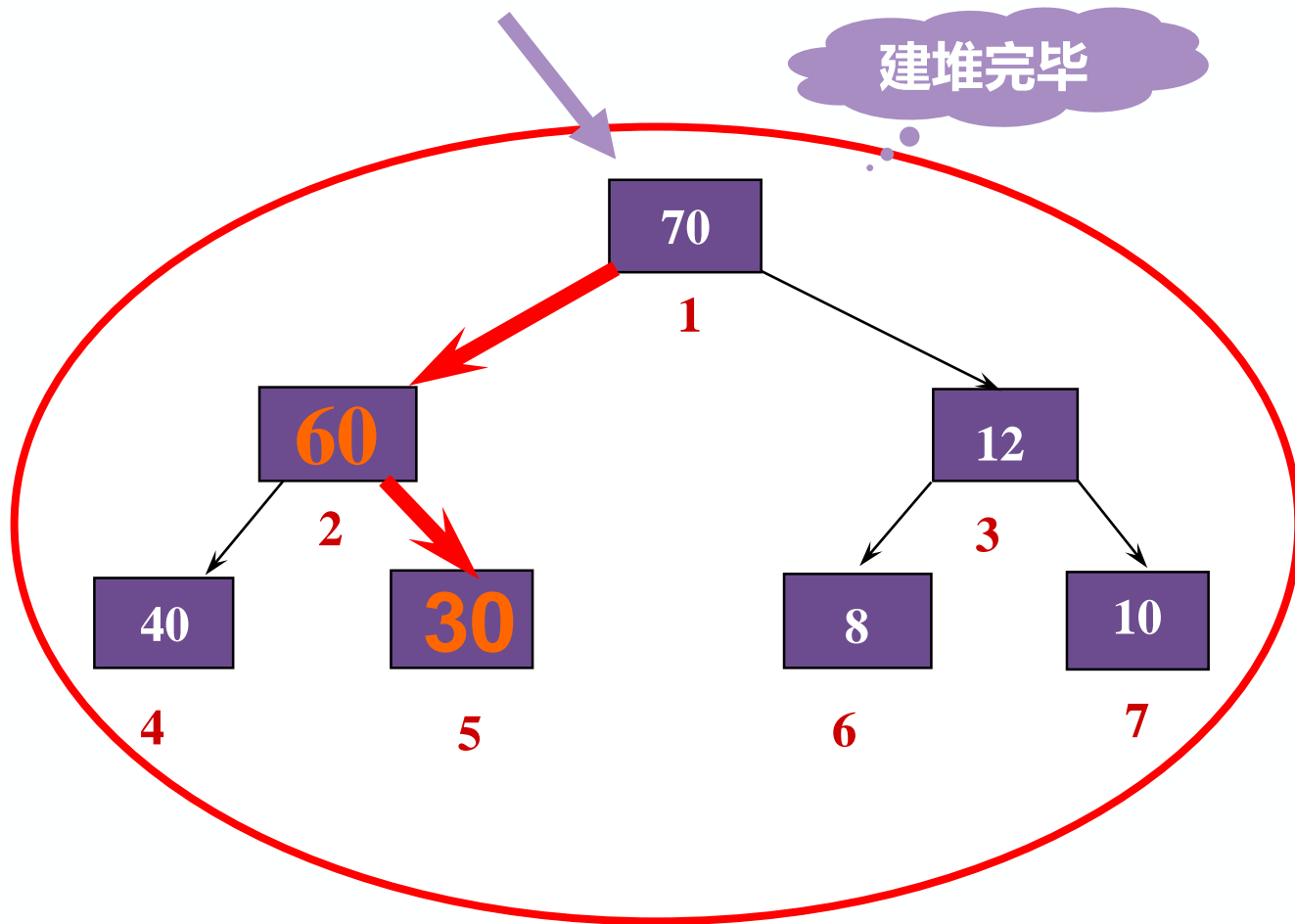
▶▶▶ 无序序列建堆

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

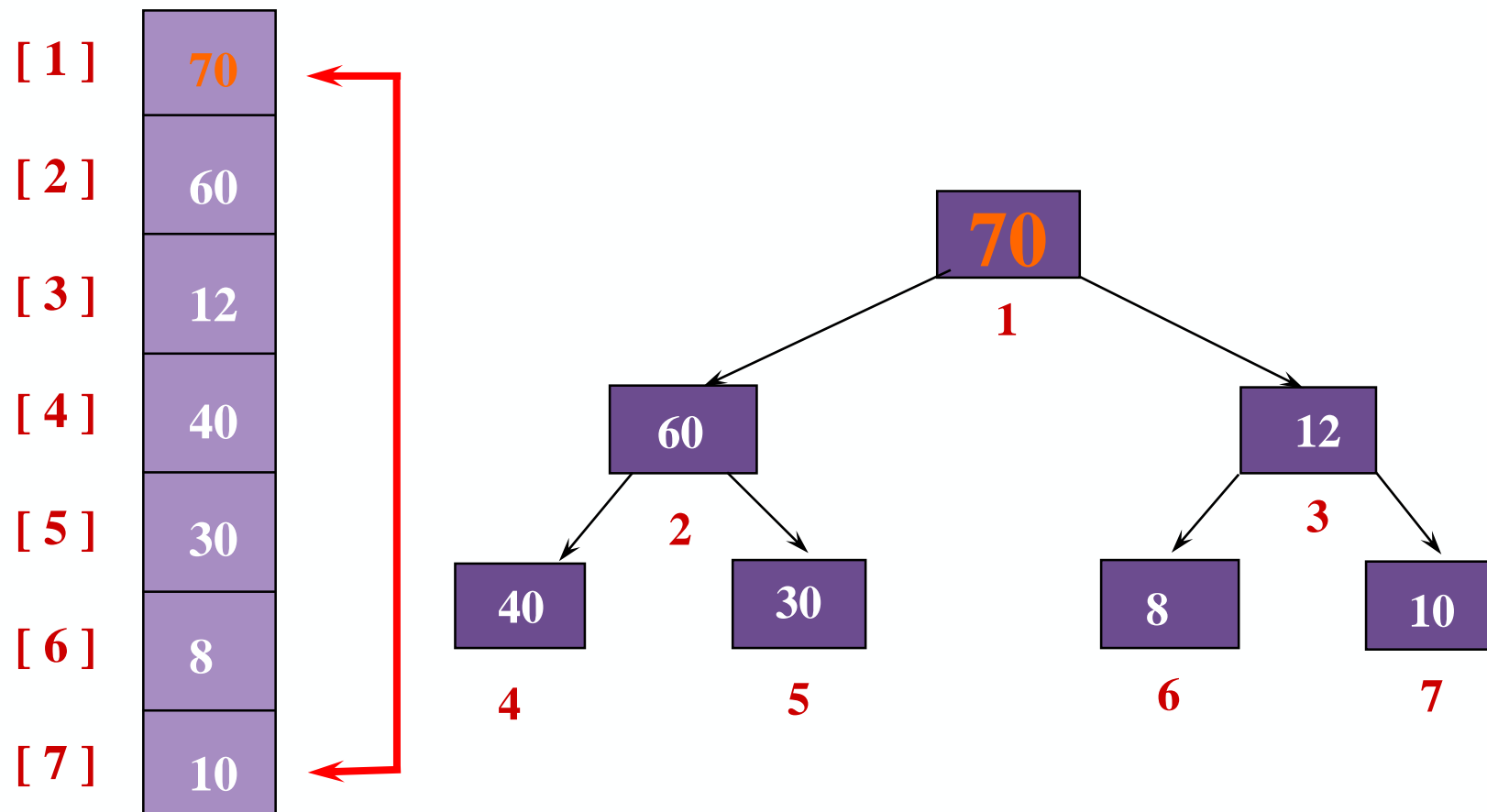


▶▶▶ 无序序列建堆

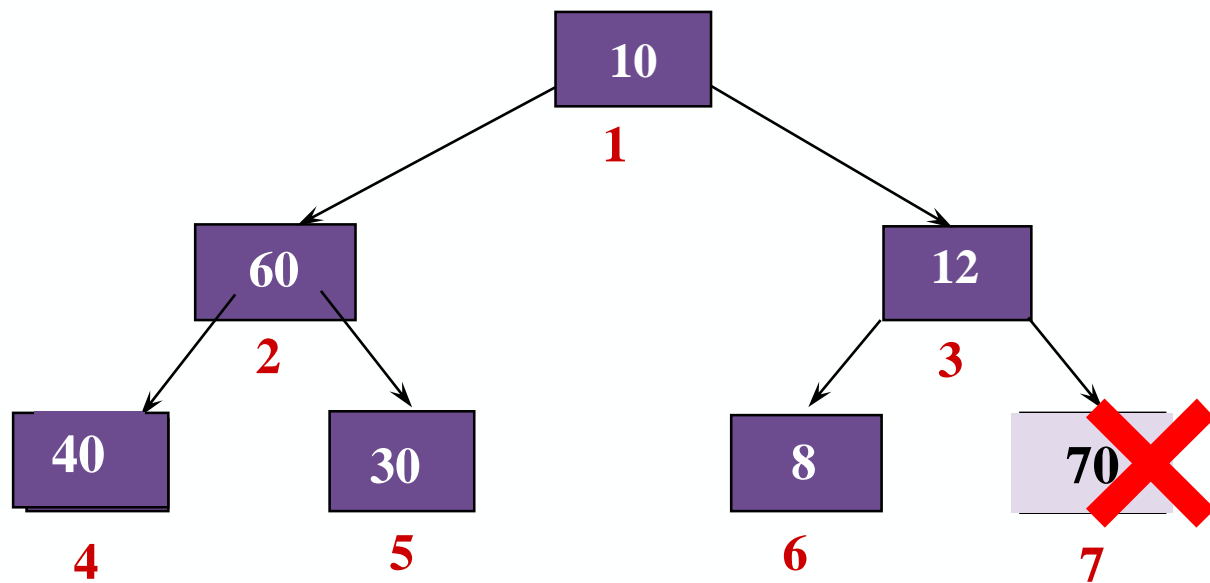
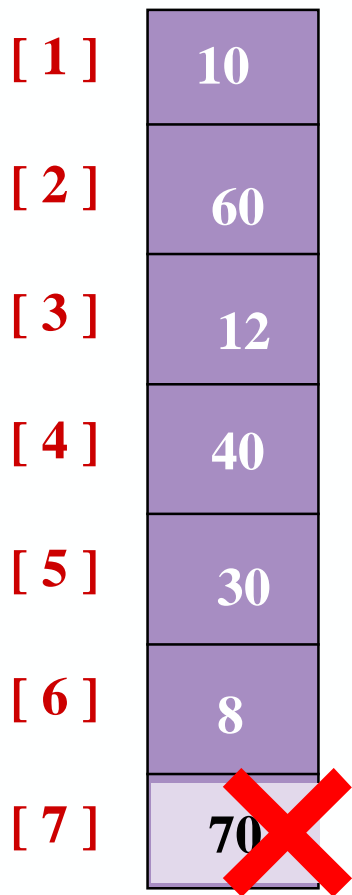
[1]	70
[2]	60
[3]	12
[4]	40
[5]	30
[6]	8
[7]	10



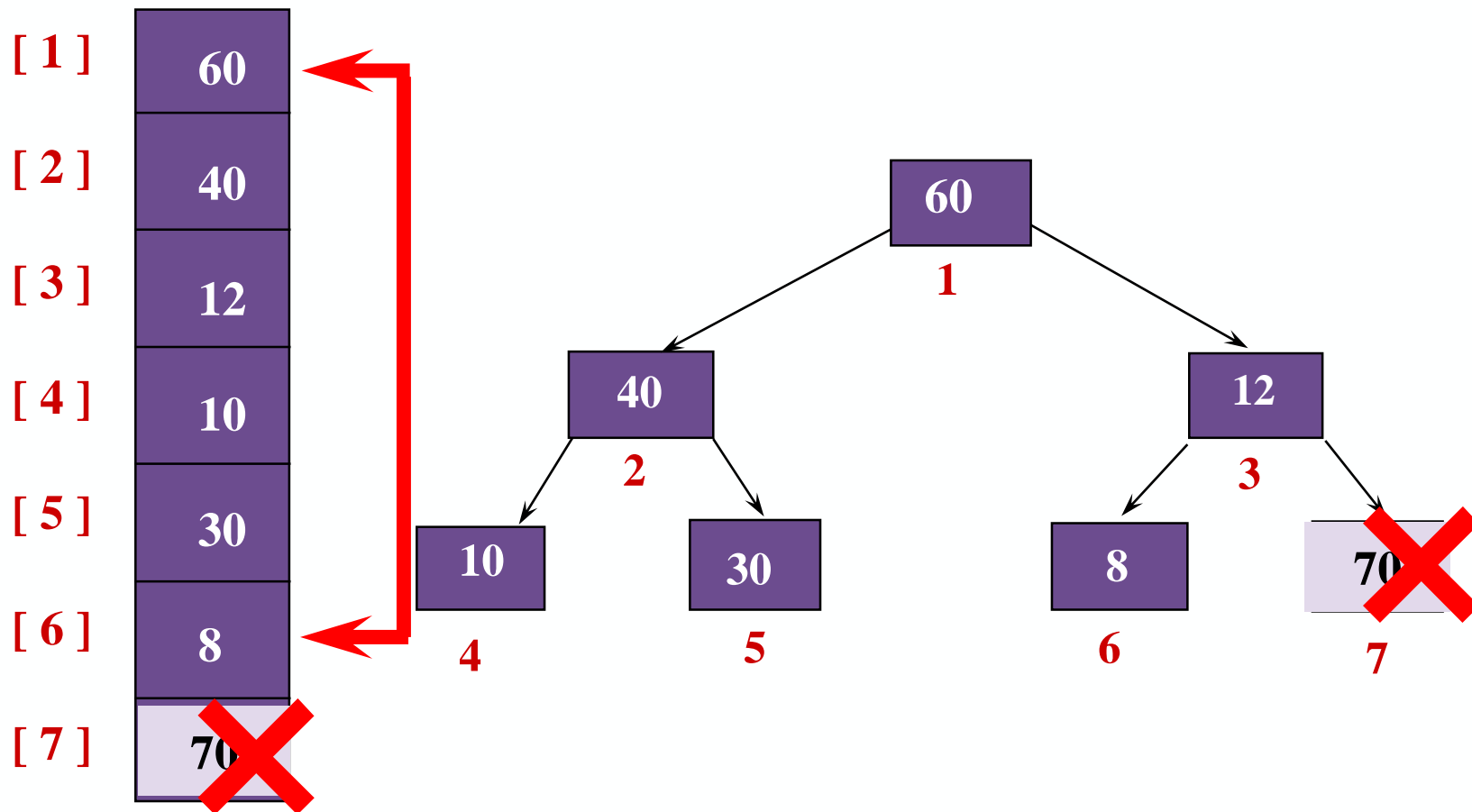
堆排序



堆排序

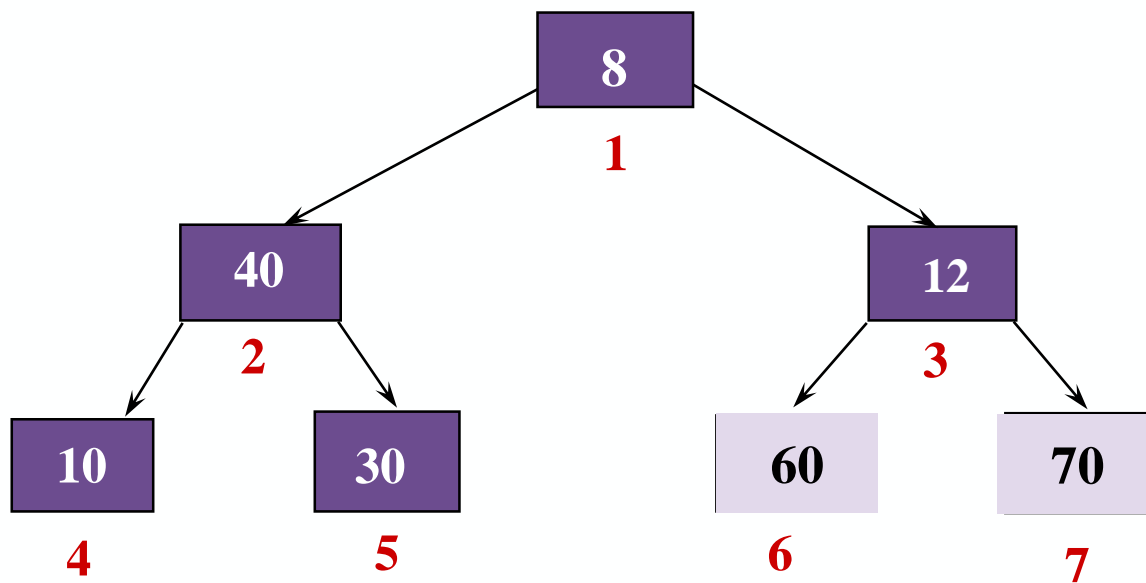


堆排序



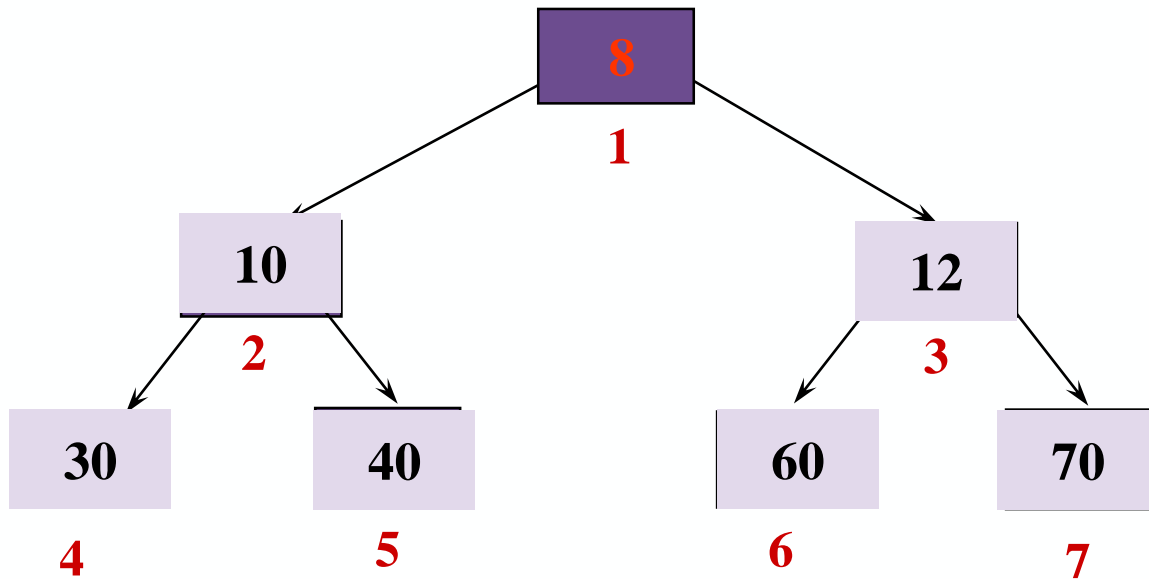
堆排序

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



堆排序

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



Two vertical bars, one red and one blue, are positioned on the left side of the slide.

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

四、归并排序

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

归并排序

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]



时间效率：
 $O(n \log 2n)$



空间效率：
 $O(n)$



稳定性：
稳定

五、基数排序

基数排序

前面的排序方法主要通过关键字值之间的比较和移动而基数排序不需要关键字之间的比较。

对52张扑克牌按以下次序排序：

♣ 2 < ♣ 3 < < ♣ A < ♦ 2 < ♦ 3 < < ♦ A <

♥ 2 < ♥ 3 < < ♥ A < ♠ 2 < ♠ 3 < < ♠ A

两个关键字：花色（♣ < ♦ < ♥ < ♠）

面值（2 < 3 < < A）

并且“花色”地位高于“面值”

多关键字排序



最高位优先**MSD** (**Most Significant Digit first**)



最低位优先**LSD** (**Least Significant Digit first**)

链式基数排序

链式基数排序：用**链表**作存储结构的基数排序

▶▶▶ 最高位优先法



先对**最高位**关键字 k_1
(如花色)排序，
将序列分成若干子
序列，每个子序列
有相同的 k_1 值；



然后让每个子序列对**次
关键字** k_2 (如面值)排
序，又分成若干更小的
子序列；



依次重复，直至就
每个子序列对**最低
位关键字** k_d 排序，
就可以得到一个有
序的序列。

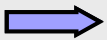
✓ 十进制数比较可以看作是一个多关键字排序

最高位优先法

278,109,063,930,184,589,269,008,083

按百位排序

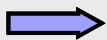
按十位排序

008,063,083 

008

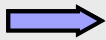
063

083

109,184 

109

184

269,278 

269

278

589

589

930

930

最低位优先法



首先依据最低位排序码 K_d 对所有对象进行一趟排序。



再依据次低位排序码 K_{d-1} 对上一趟排序结果排序。



依次重复，直到依据排序码 K_1 最后一趟排序完成，就可以得到一个有序的序列。

✓ 这种方法不需要再分组，而是整个对象组都参加排序；

最低位优先法

278, 109, 063, 930, 184, 589, 269, 008, 083

按个位排序

930, 063, 083, 184, 278, 008, 109, 589, 269

按十位排序

008, 109, 930, 063, 169, 278, 083, 184, 589

按百位排序

008, 063, 083, 109, 169, 184, 278, 589, 930,

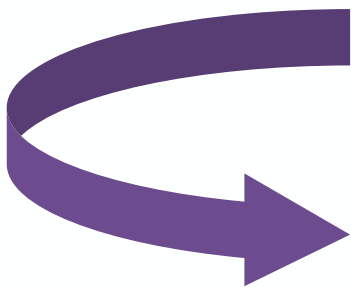
▶▶▶ 算法分析

n个记录

每个记录有 **d** 位关键字

关键字取值范围**rd**(如十进制为10)

- 重复执行**d**趟“分配”与“收集”
- 每趟对 **n** 个记录进行“分配”，对**rd**个队列进行“收集”
- 需要增加**n+2rd**个附加链接指针。



✓ 时间效率: $O(d(n+rd))$

✓ 空间效率: $O(n+rd)$

✓ 稳定性: 稳定