

## 目录

## 排序

- 1 引言
- 2 合并排序（第六章分治法）
- 3 用比较法进行排序的时间下界
- 4 选择排序和堆排序（第三章）
- 5 插入排序和希尔排序
- 6 快速排序（第六章分治法6.6节）
- 7 基数排序（第五章归纳法）

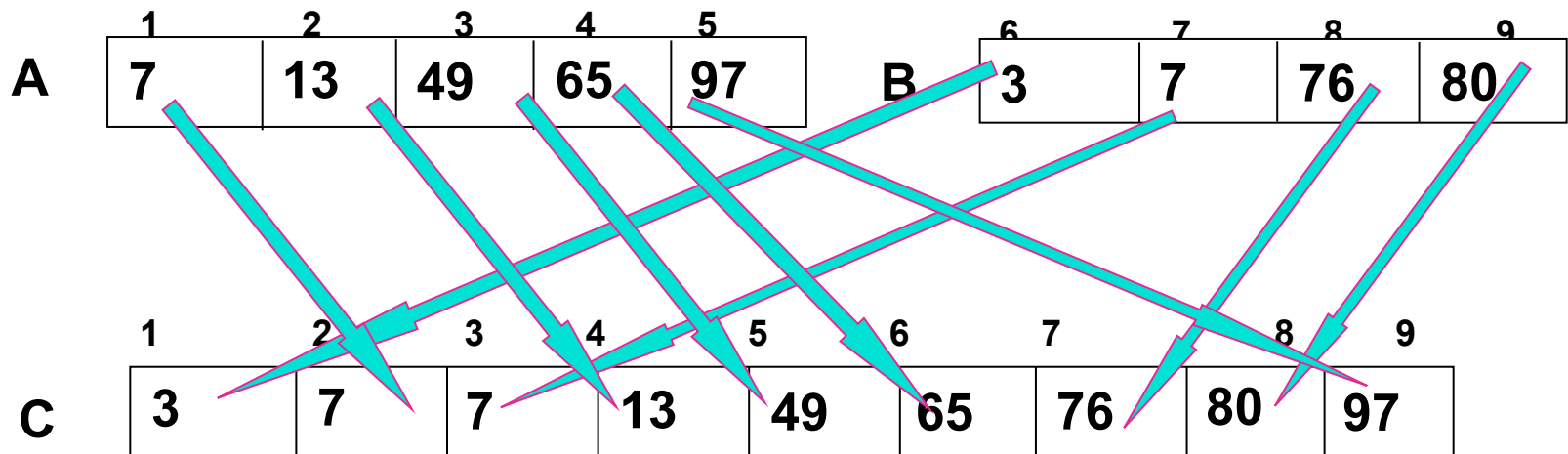
# 1、概述

- 定义：设有记录序列：{  $R_1$ 、 $R_2$  .....  $R_n$  } 其相应的关键字序列为：  
    {  $K_1$ 、 $K_2$  .....  $K_n$  }; 若存在一种确定的关系：  $K_x \leq K_y \leq \dots \leq K_z$   
    则将记录序列 {  $R_1$ 、 $R_2$  .....  $R_n$  } 排成按该关键字有序的序列：  
    {  $R_x$ 、 $R_y$  .....  $R_z$  } 的操作，称之为排序。  
    关系是任意的，如通常经常使用的小于、大于等关系或任意的关系。
- 稳定与不稳定：若记录序列中的任意两个记录  $R_x$ 、 $R_y$  的关键字  $K_x = K_y$  ；如果在  
    排序之前和排序之后，它们的相对位置保持不变，则这种排序方法  
    是稳定的，否则是不稳定的。

## 2、合并排序

合并两张表：

- **Merging sort:** 思想来源于合并两个已排序的顺序表。
- **e.g:** 将下属两个已排序的顺序表合并成一个已排序表。顺序比较两者的相应元素，小者移入另一表中，反复如此，直至其中任一表都移入另一表为止。



- 实现很简单，A、B、C 三表分别设置指针p、q、k 初值为 1, 6, 1，比较 p、q 指针指向的结点，将小者放入 C 表, 相应指针加1。如A、B表中一张表为空，则将另一张表的结点全部移入C表。
- 比较次数和移动次数和两张表的结点总数成正比。

## 2、合并排序

- 合并二张有序表：

```
template < class EType>
```

```
void Merge( EType a[ ], EType aa[ ], int n, int low, int up, int m ) {
```

```
    int p=low, q=low+m, r;
```

// p为被合并的二个表中的第一个表的首地址，m为其表长。up为第二张的末地址。q为

// 表的首地址。r 为数组 a 的指针。a[1], a[2 ], ...,a[n]为待排序序列。a[0]不用。

```
for (r=1; r<=n; r++) aa[r] = a[r];
```

```
r = low; // 数组 a 的初始指针。
```

```
while ( p < low + m && q < up + 1 ) {
```

```
    while ( p < low + m && aa[p] <= aa[q] ) a[r++] = aa[p++];
```

```
    if ( p < low + m ) while ( q < up + 1 && aa[q] < aa[p] ) a[r++] = aa[q++];
```

```
}
```

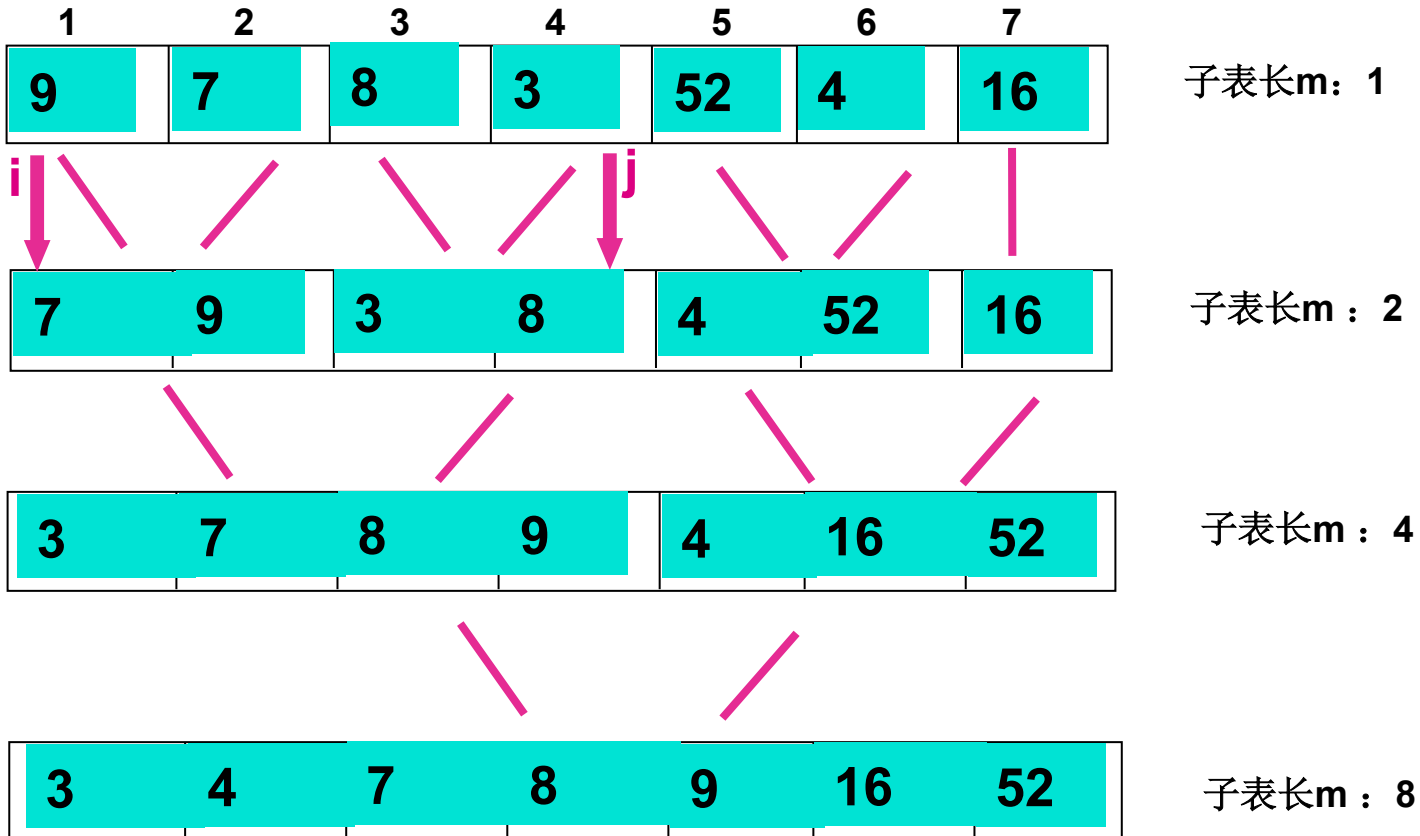
```
if ( p == low + m ) for ( ; q <= up; q++ ) a[r++] = aa[q];
```

```
if ( q == up + 1 ) for ( ; p <= low + m - 1; p++ ) a[r++] = aa[p];
```

```
}
```

## 2、合并排序

### 3、合并排序的基本思想：



分析：1、被合并的二张表中第一张表的起始地址设为  $low$ ，那么第二张表的起始地址应为  $low + m$ ，而终止地址通常为  $up = low + 2m - 1$ 。如果  $up > n$  (表长)，那么  $up = n$ ；即取二者小者作第二张表的终止地址。

2、当  $low + m \geq n$  时，意味着下一次被合并的二张表中的，第二张表不存在，意味着本趟结束。要过渡到下一趟合并； $m$  应增大一倍。

3、当子表长  $m \geq n$  时，合并排序结束。

## 2、合并排序

- 非递归合并排序法:

```

template < class EType>
void MergeSort( EType a[ ], int n ) {
    // 待排序的表为数组a, a[0] 不用, 待排序的数组元素为 a[1], a[2], ....., a[n]。
    int low=1; // 被合并的二个表中的第一个表的首地址。
    int up;    // 被合并的二个表中的第二个表的末地址。
    int m=1;   // 被合并的二个表中的第一个表的表长。初始时为1。
    EType * aa;
    aa = new EType[n+1]; // 用于合并的辅助数组, aa[0]仍然不用。
    while ( m < n ) {
        up = minmum(low + 2 * m -1, n);
        Merge(a, aa, n, low, up,m);    // 将a[low] 至 a[low+m-1] 和a[low+m]至a[up]进行合并。
        if ( up+m < n ) low = up + 1; // up+m ≥ n 意味着被合并的另一张表不存在。
        else { m *= 2;
                low = 1;
            } // 过渡到下一次合并, 子表长度增大一倍。
    }
    delete [ ]aa;
}

```

## 2、合并排序

·时间复杂性分析:

- 合并的过渡趟数:  $\lceil \log n \rceil$
- 每趟进行时, 调用 **merge** 过程  $\lceil n / (2 * len) \rceil$  次, 每次比较和数据移动都是  $(2 * len)$  次, 因此每趟代价为  $O(n)$ 。
- 总的代价为  $T(n) = O(n \log n)$
- 在递归的情况下: 可通过下式求出时间复杂性的级别。

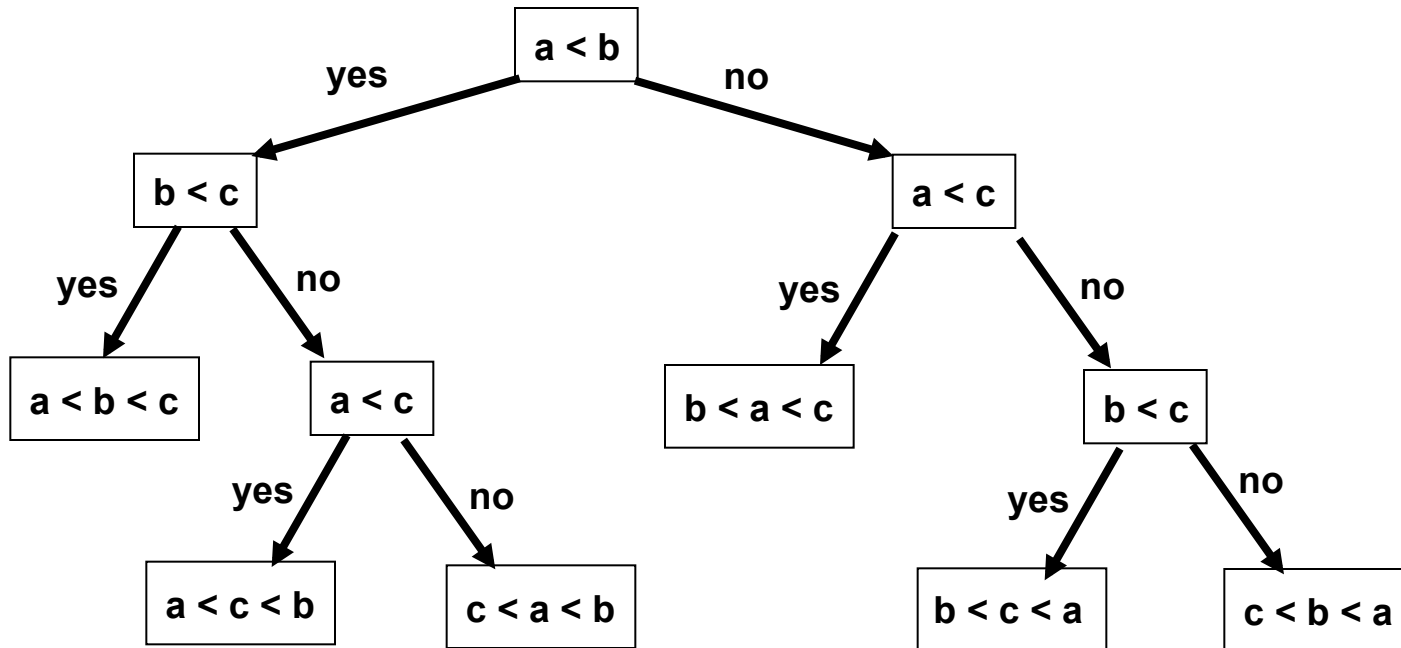
$$T(n) = \begin{cases} c & \text{for } n=2 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn & \text{else} \end{cases}$$

解上述的递归式, 可知时间复杂性为  $T(n) = O(n \log n)$ 。当  $n$  是 2 的整数幂时简化为  $T(n) = O(n \log n)$

### 3、用比较法进行排序的时间下界

- 比较法分类的下界：  $\Omega(n \log n)$

- 判定树模型：如下图所示，说明对  $a, b, c$  进行分类的一颗判定树。当判断条件成立时，转向左，否则转向右。



- 注意：树高代表比较的代价。因此只要知道了树高和结点数  $n$  的关系，就可以求出用比较法进行排序时的时间代价。另外， $n$  个结点的分类序列，其叶子结点共有  $n!$  片。



### 3、用比较法进行排序的时间下界

- 比较法分类的下界： $\Omega(n \log n)$

• 定理：高为  $H$  的二叉树，最多具有  $2^{(H-1)}$  片叶子。

证明： $H = 1$  时，最多 1 片叶子，定理成立。

$H = 2$  时，最多 2 片叶子，定理成立。

设  $H = n-1$  时公式成立。则当  $H = n$  时，  
根结点有具有叶子结点最多的高为  $n-1$  的左  
右子树；叶子结点总数为：

$$2^{(n-2)} \times 2 = 2^{(n-1)} \quad \text{公式成立。}$$

推论：具有  $N$  片叶子的二叉树，高最少为  $\lceil \log N \rceil + 1$ 。

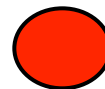
或比较次数最少为  $\lceil \log N \rceil$ 。

- 定理：把  $n$  个不同的结点进行分类的任一判定树的高度至少为  $\log(n!) + 1$ 。

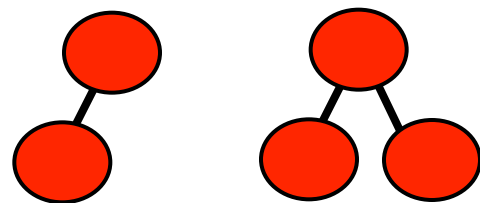
证明： $n$  个元素的排序序列的总数共有  $n!$  种，因此在相应的判定树上至少有  $n!$  片叶子。根据上述定理，可证。

推论：用比较的方法对  $n$  个元素的进行分类。在最坏情况下至少需要  $\mathbf{cn \log n}$  次比较，或  
 $\mathbf{\Omega(n \log n)}$

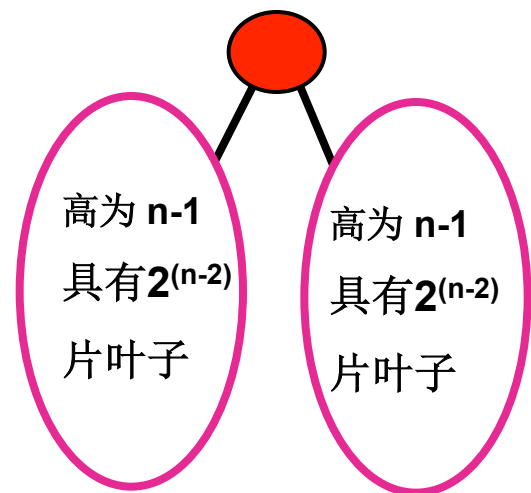
$H = 1$



$H = 2$



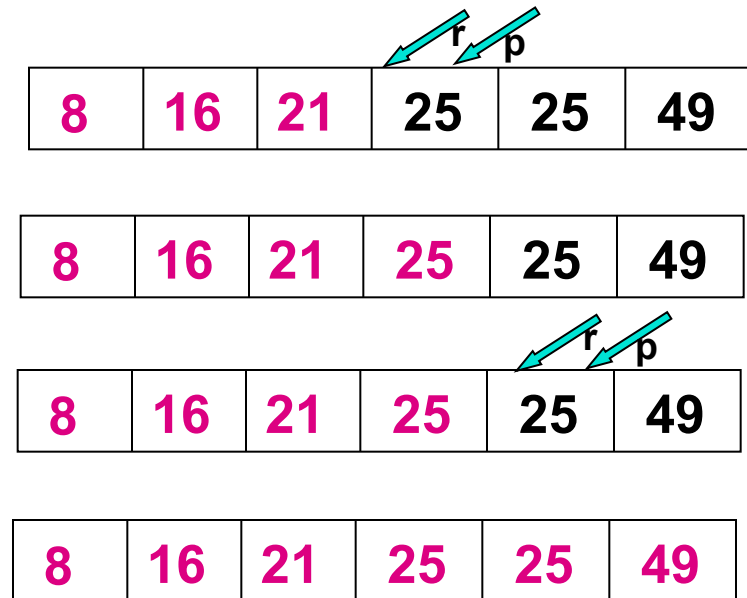
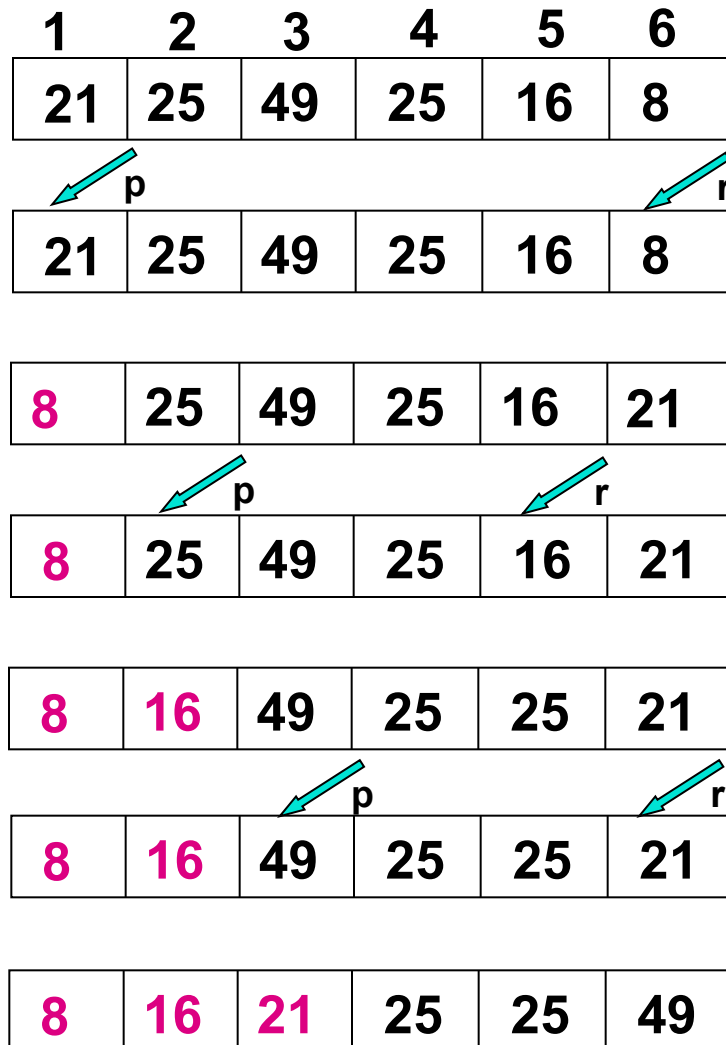
$H = n$



### 3、选择排序和堆排序

#### 1、简单选择排序

##### • 执行过程:



### 3、选择排序和堆排序

#### 1、简单选择排序

• 实现:

1	2	3	4	5	6
21	25	49	25	16	8

```

template < class EType>
void SelectSort( EType a[ ], int n ) {
    int p, q; // p、q 为数组a的指针。a[1], a[2 ], ...,a[n]为待排序序列。a[0]不用。
    int r;    // 暂时保存具有最小键值的结点地址。
    for ( p = 1; p < n; p++) {
        r = p;
        for ( q = p+1; q <= n; q++)
            if ( a[q ] < a[r] ) r = q;           // 记住本趟挑选的最小结点的地址。
        if ( r !=p ) { a[0] = a[p]; a[p] = a[r]; a[r] = a[0]; } // 若最小结点不是a[p], 则交换。
    }
}

```

## For Selection Sort in General

- The number of comparisons when the array contains  $N$  elements is

$$\begin{aligned}\text{Sum} &= (n-1) + (n-2) + \dots + 2 + 1 \\ &= O(n^2)\end{aligned}$$

- The number of moves when the array contains  $n$  elements is

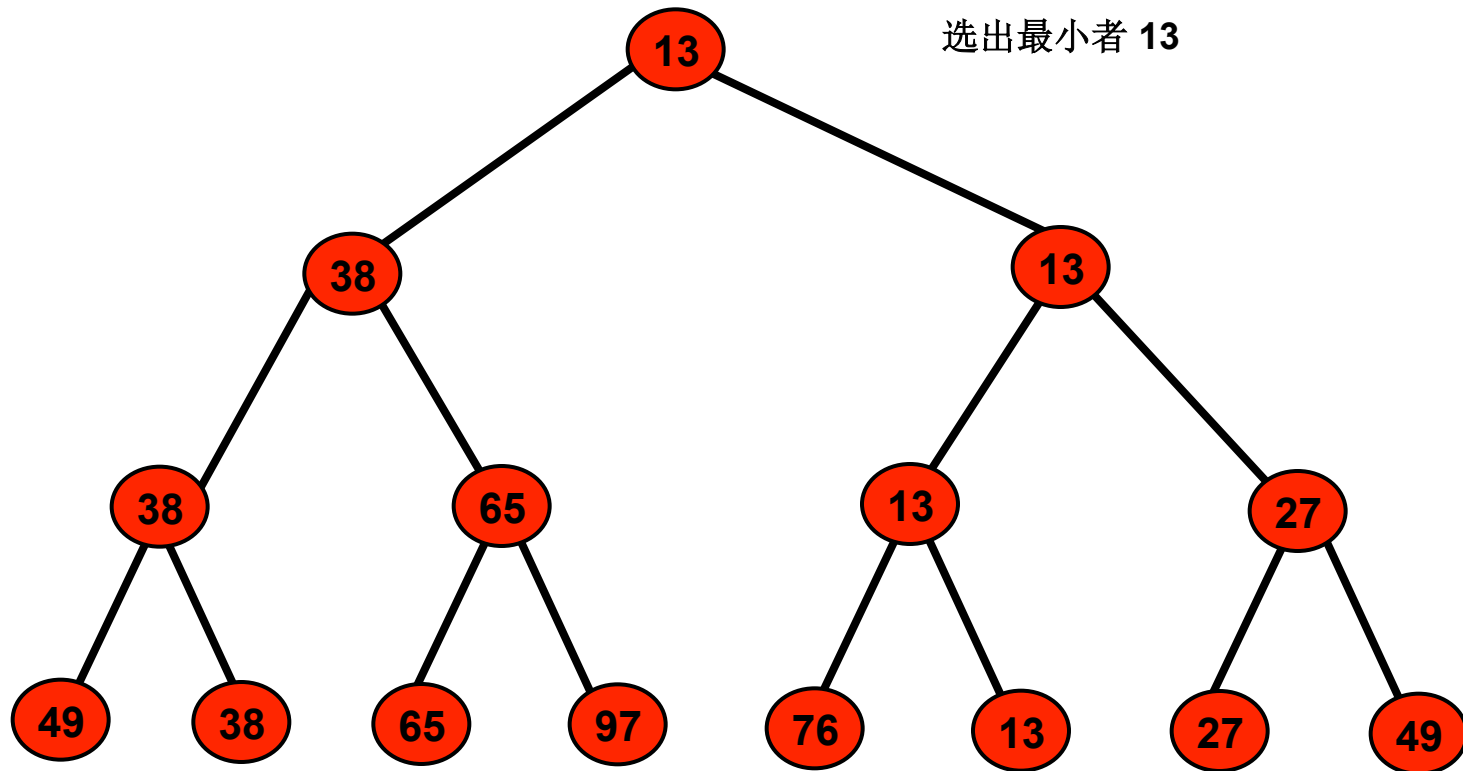
$$\text{Sum} = 3(n-1) \text{ 循环执行 } (n-1) \text{ 次, 每次都交换。}$$

总的时间复杂性代价为:  $O(n^2)$

## 4、选择排序和堆排序（注：算法设计与数据组织的相互关系）

### 2、树性选择排序

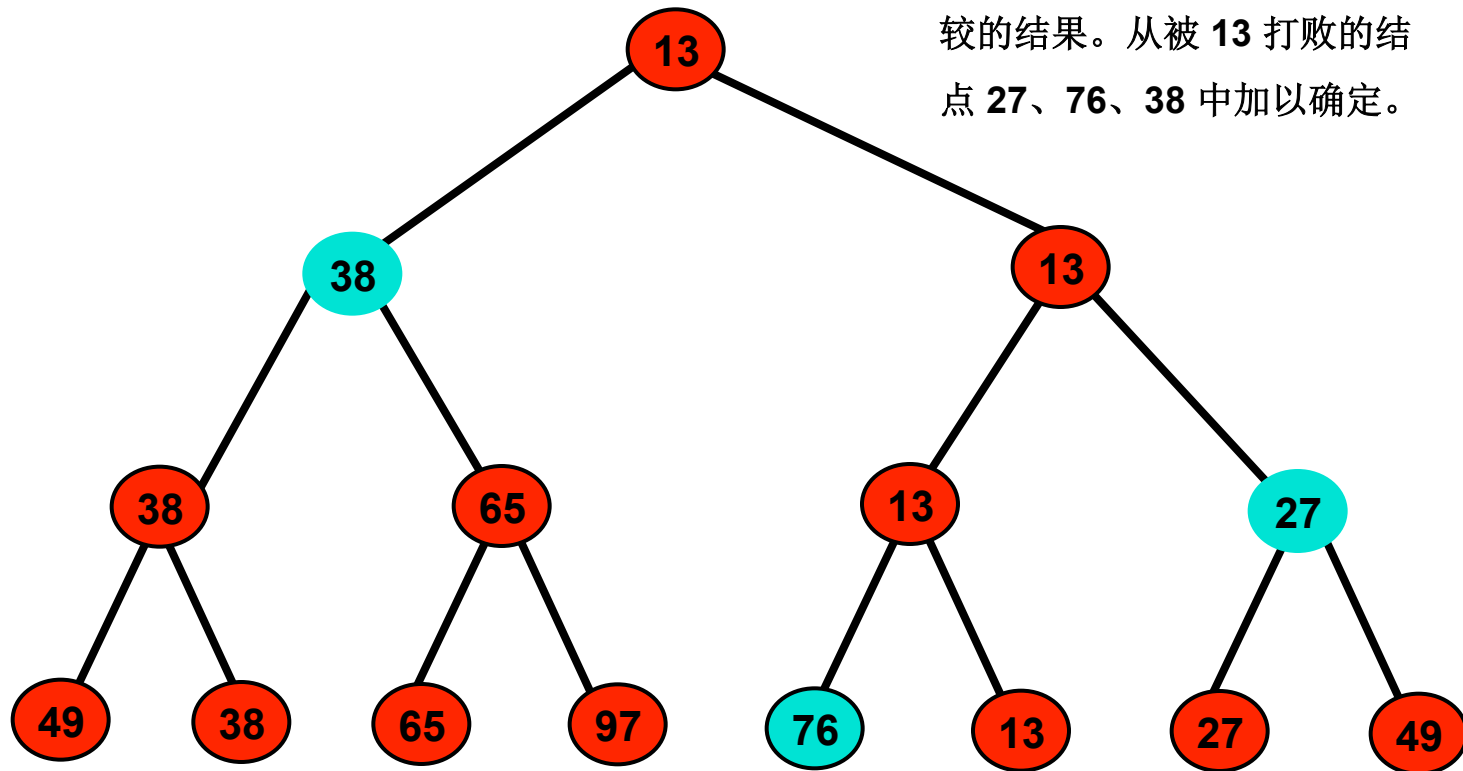
- 改进：简单选择排序没有利用上次选择的结果，是造成速度慢的主要原因。如果能够加以改进，将会提高排序的速度。



## 4、选择排序和堆排序

### 2、树性选择排序

- 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。由于有更好的排序方法，所以本方法用的很少。



## 4、选择排序和堆排序

### 3、堆排序

- 堆的定义：n 个元素的序列  $\{k_1, k_2, \dots, k_n\}$ ，当且仅当满足以下关系时，称之为堆：

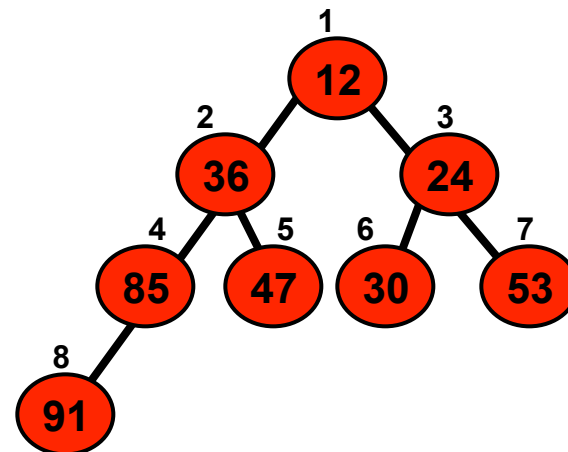
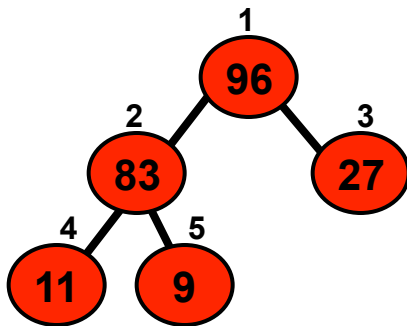
$$\begin{cases} k_i \leq k_{2i} \\ k_i \leq k_{2i+1} \end{cases} \quad \text{或} \quad \begin{cases} k_i \geq k_{2i} \\ k_i \geq k_{2i+1} \end{cases} \quad (i = 1, 2, \dots, \lfloor n/2 \rfloor)$$

且分别称之为最小化堆和最大化堆。

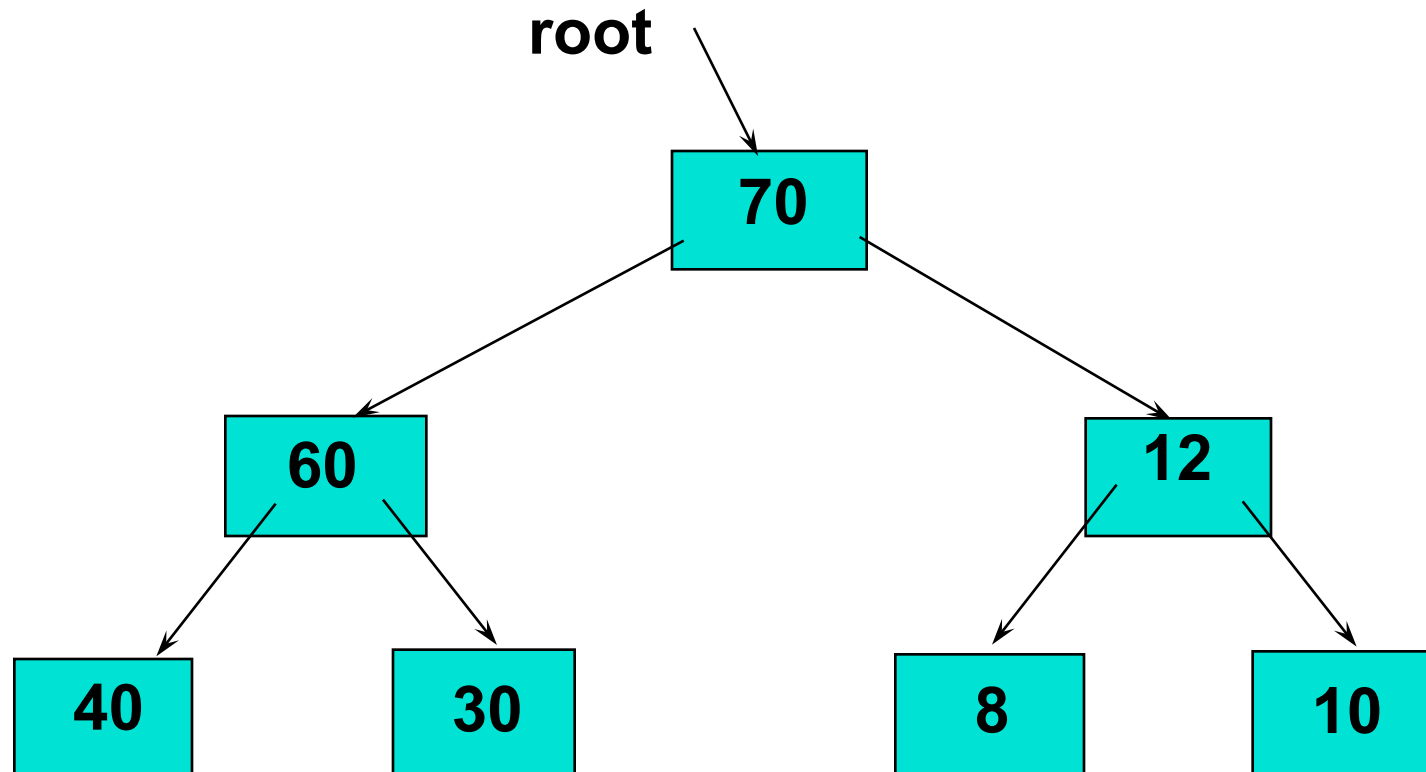
e.g: 序列  $\{96, 83, 27, 11, 9\}$

最大化堆

$\{12, 36, 24, 85, 47, 30, 53, 91\}$  最小化堆



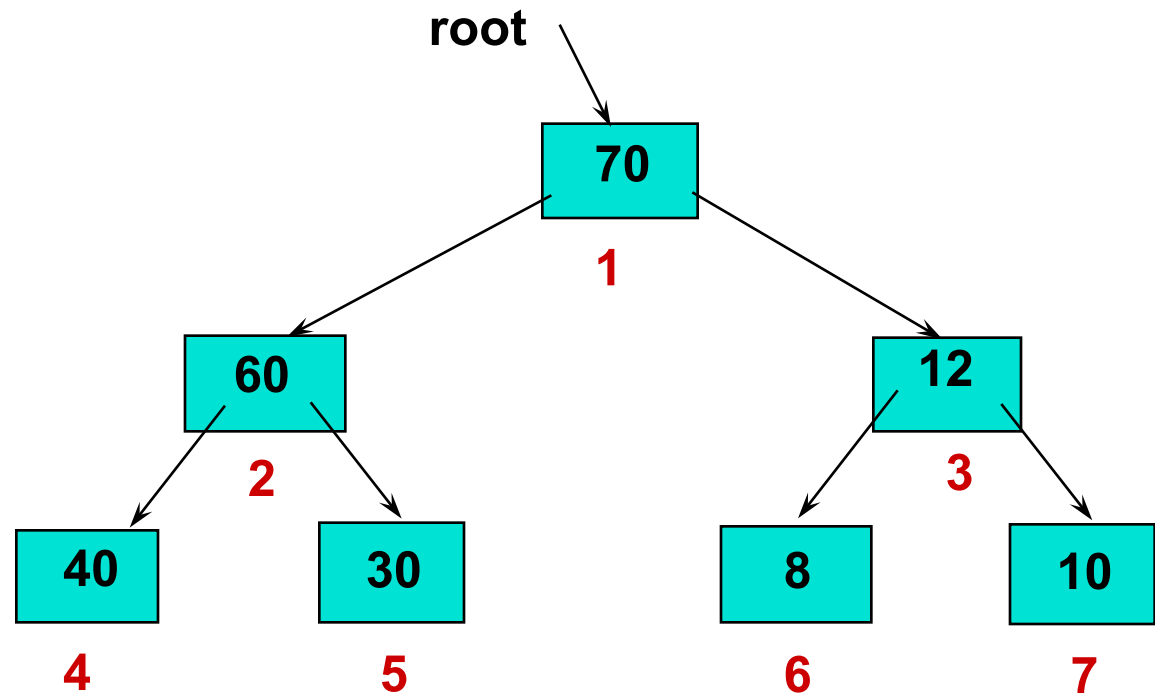
**The largest element  
in a maximum-heap is always found in the root node**





# The heap can be stored in an array

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



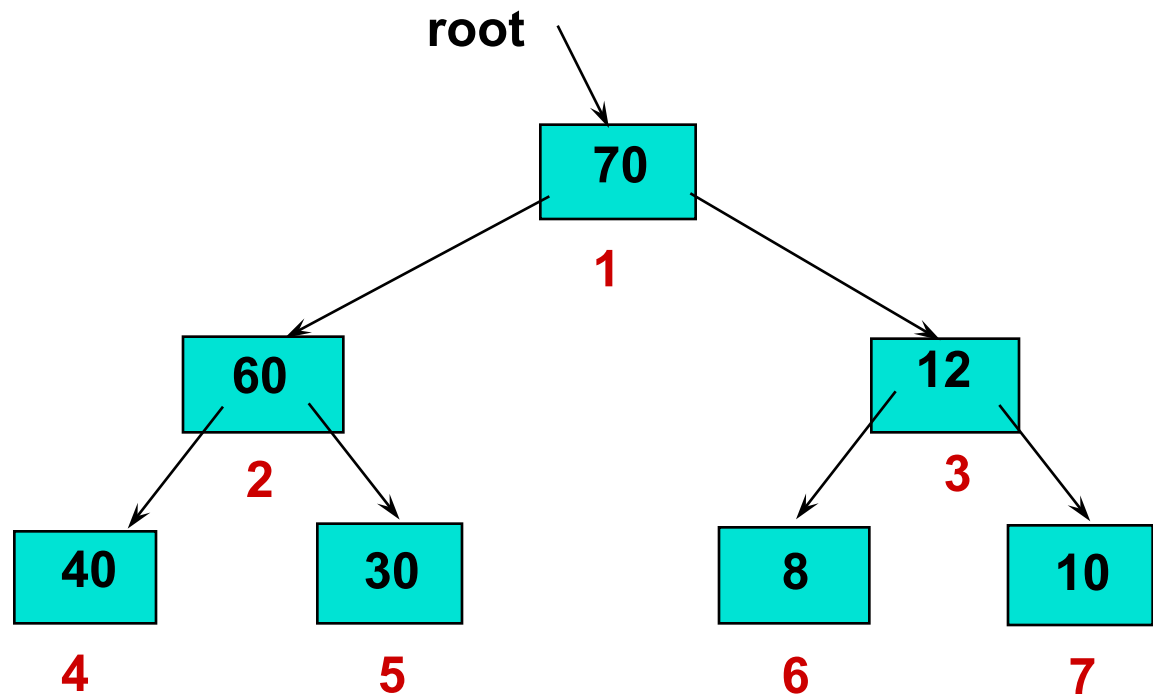
## 4、选择排序和堆排序

First, make the unsorted array into a heap by satisfying the order property. Then repeat the steps below until there are no more unsorted elements.

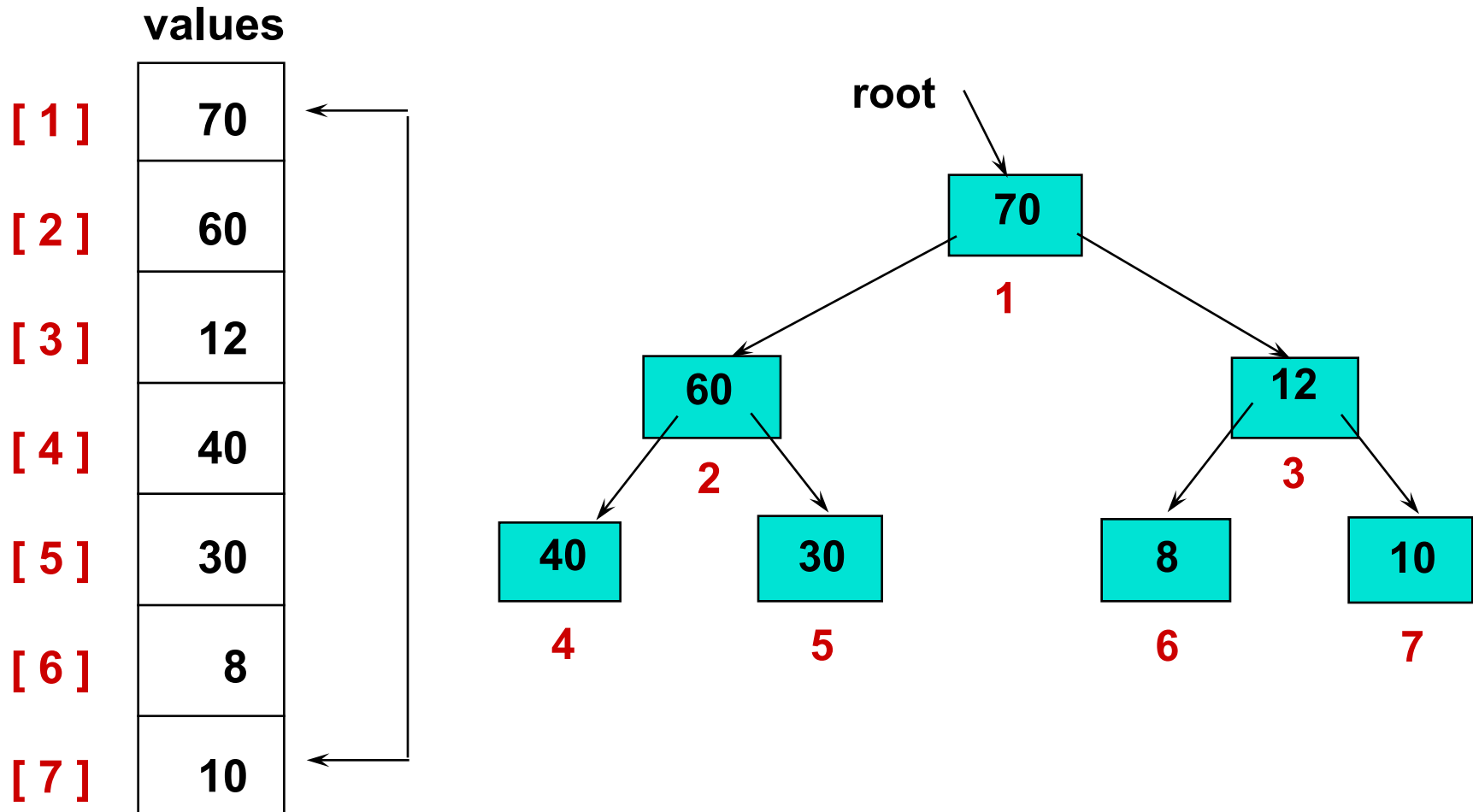
- **Take the root (maximum) element off the heap** by swapping it into its correct place in the array at the end of the unsorted elements.
- **Reheap the remaining unsorted elements.** (This puts the next-largest element into the root position).

# After creating the original heap

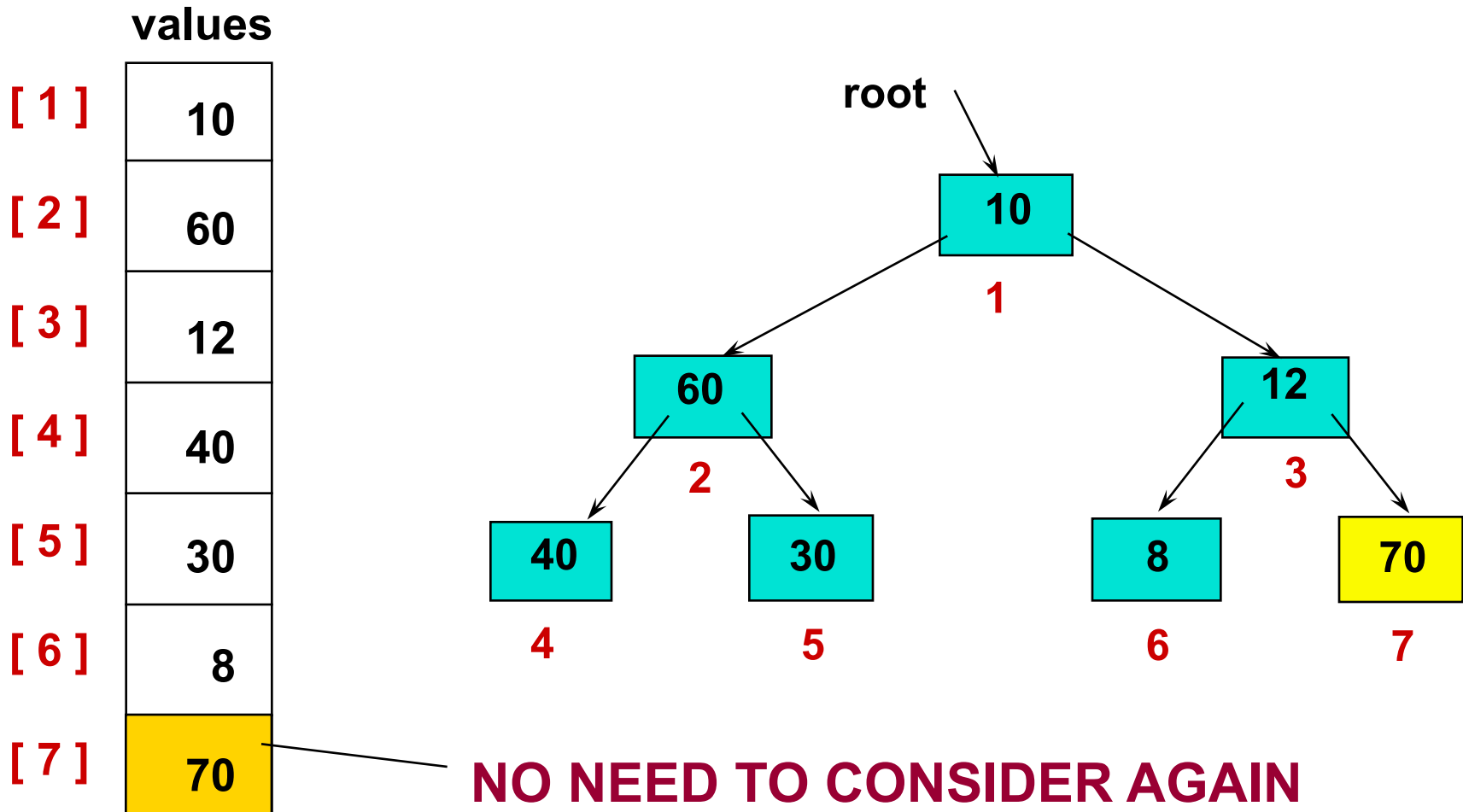
	values
[ 1 ]	70
[ 2 ]	60
[ 3 ]	12
[ 4 ]	40
[ 5 ]	30
[ 6 ]	8
[ 7 ]	10



# Swap root element into last place in unsorted array

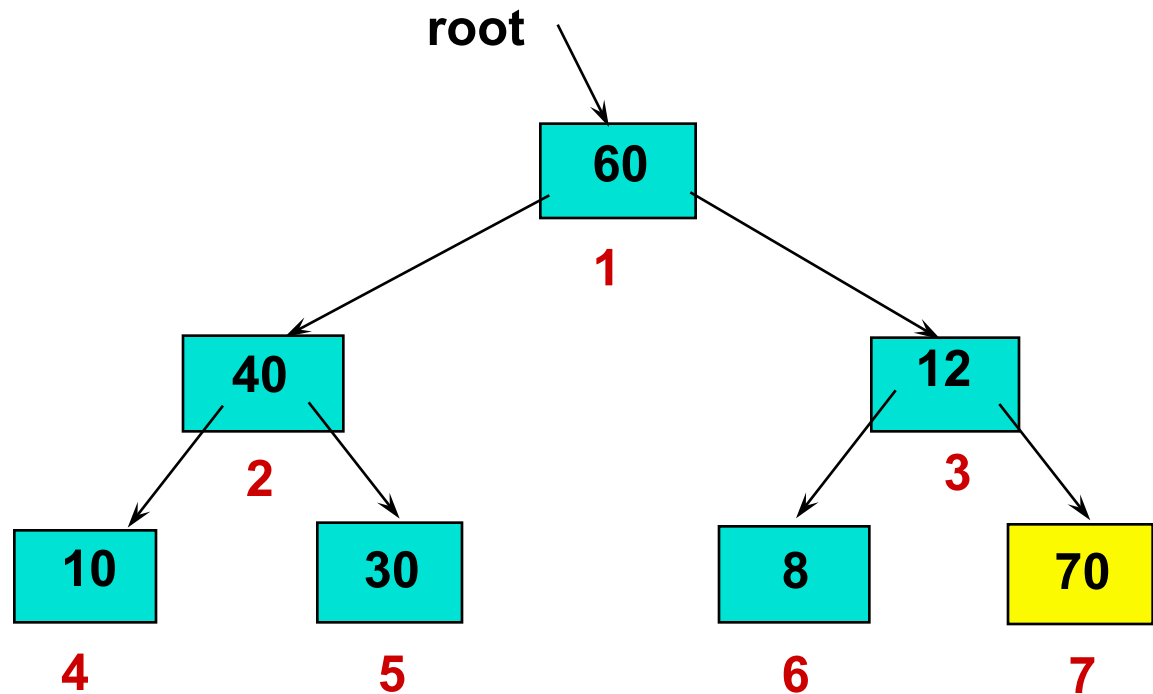


# After swapping root element into its place

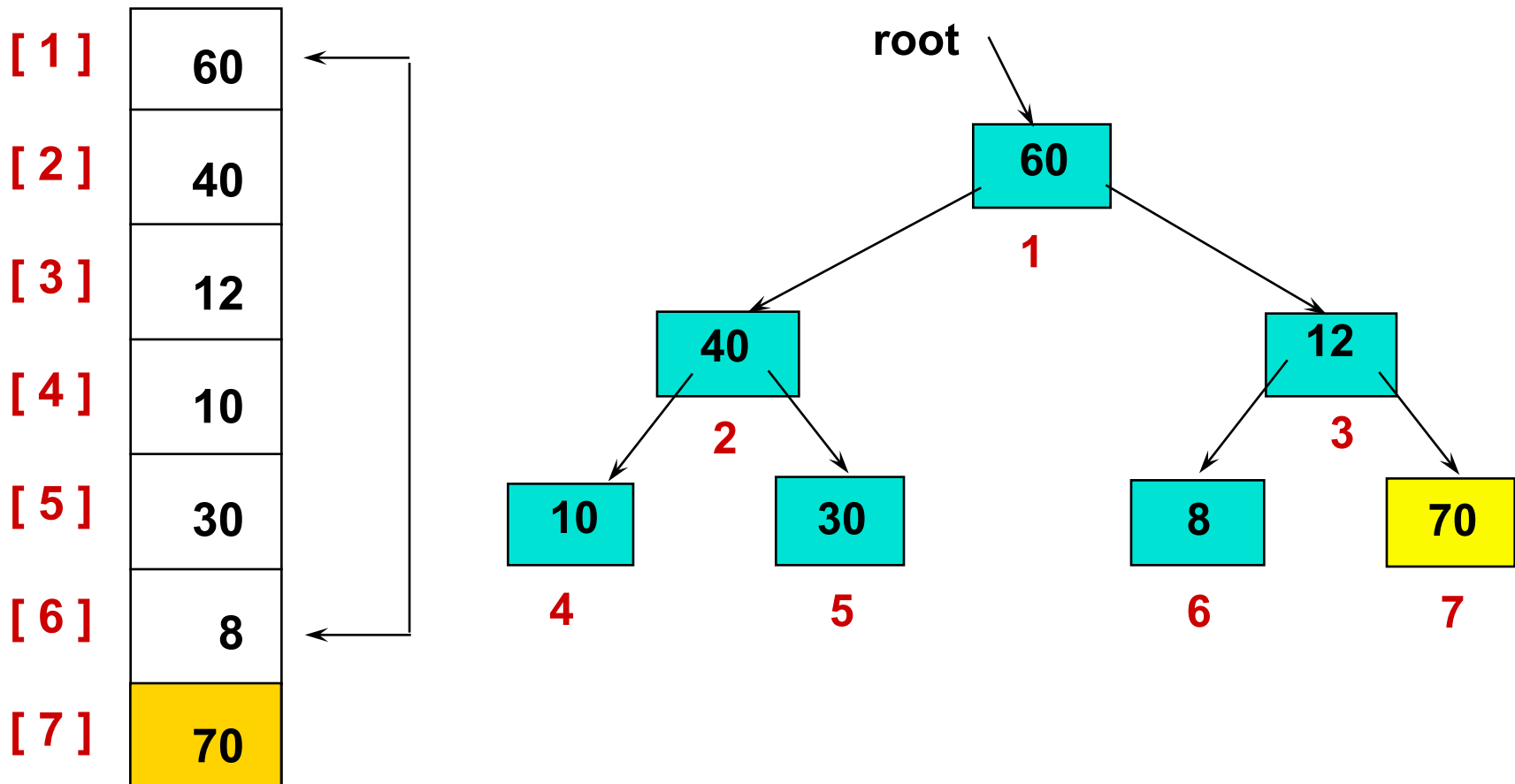


# After reheaping remaining unsorted elements

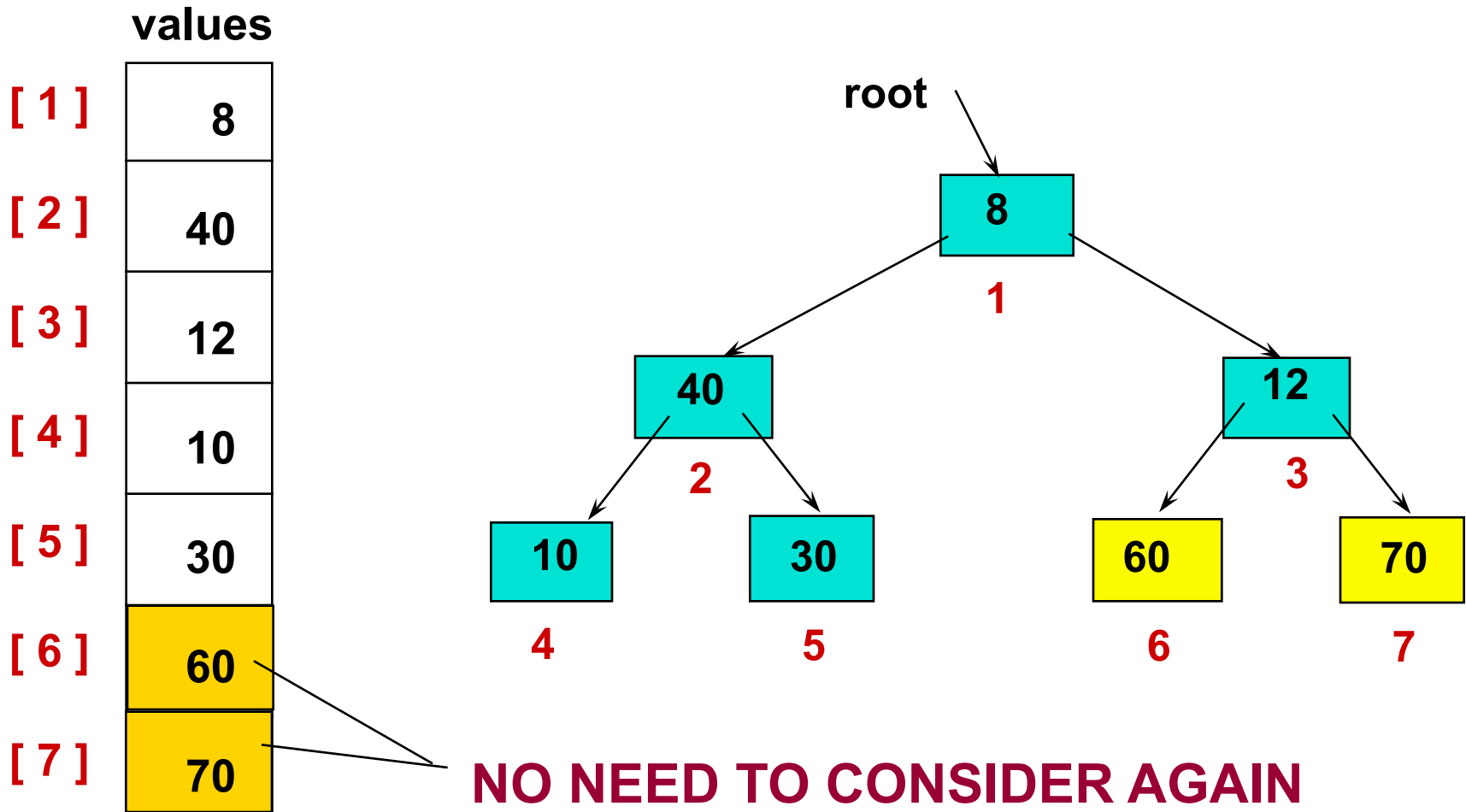
	values
[ 1 ]	60
[ 2 ]	40
[ 3 ]	12
[ 4 ]	10
[ 5 ]	30
[ 6 ]	8
[ 7 ]	70



# Swap root element into last place in unsorted array



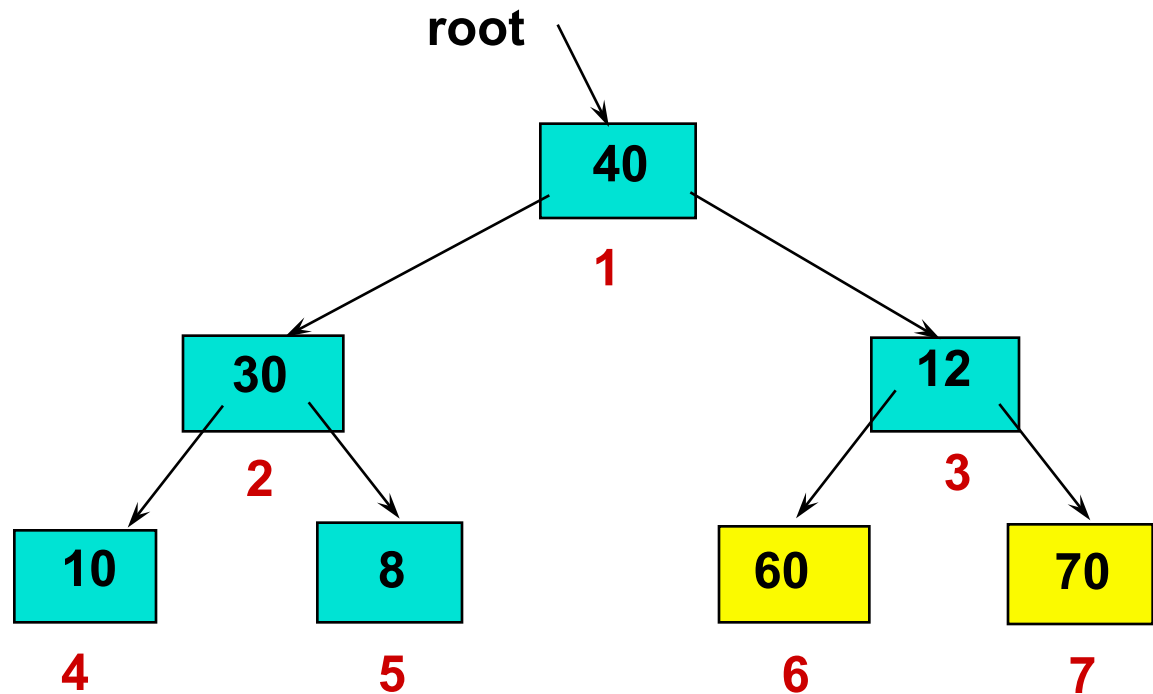
# After swapping root element into its place



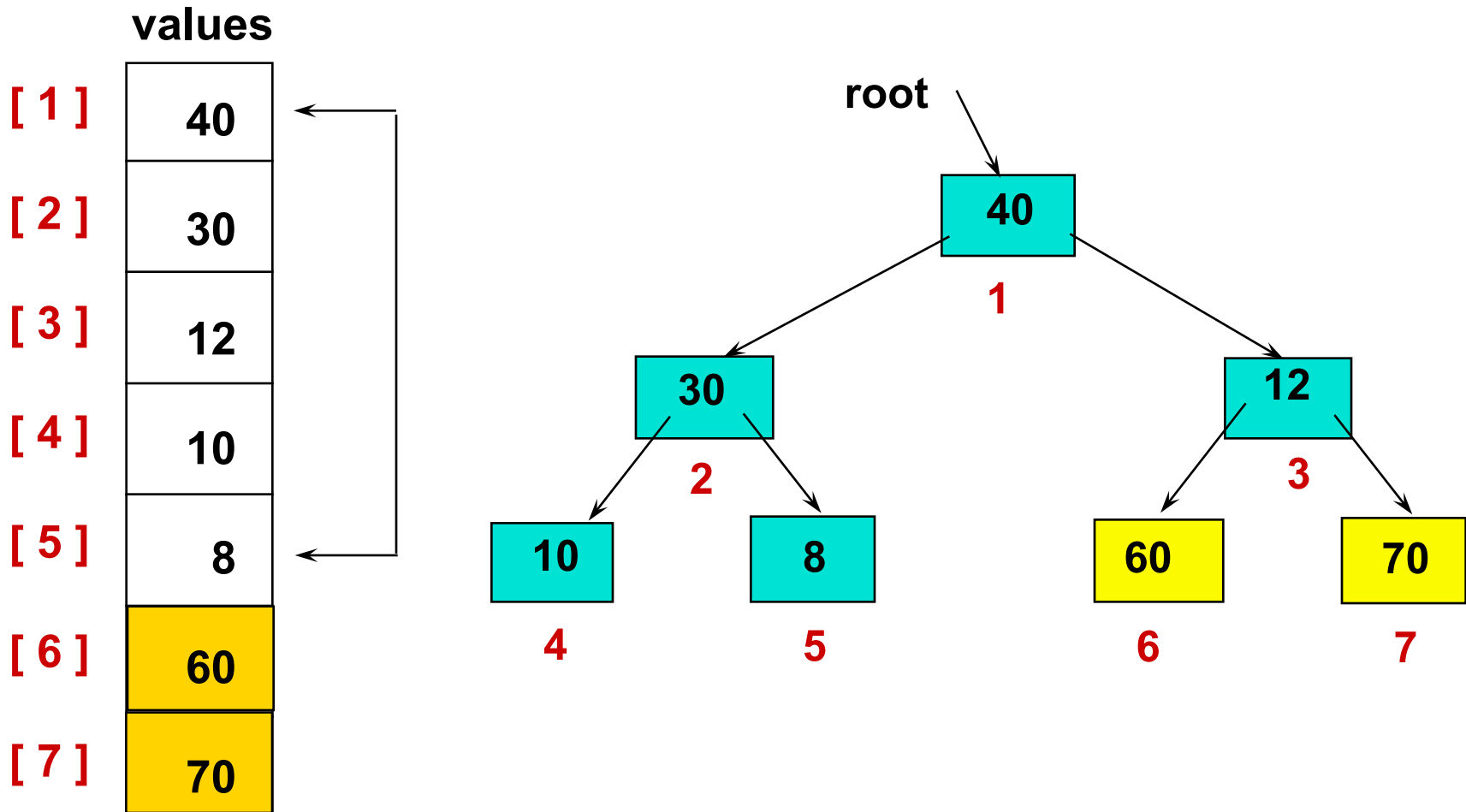


# After reheaping remaining unsorted elements

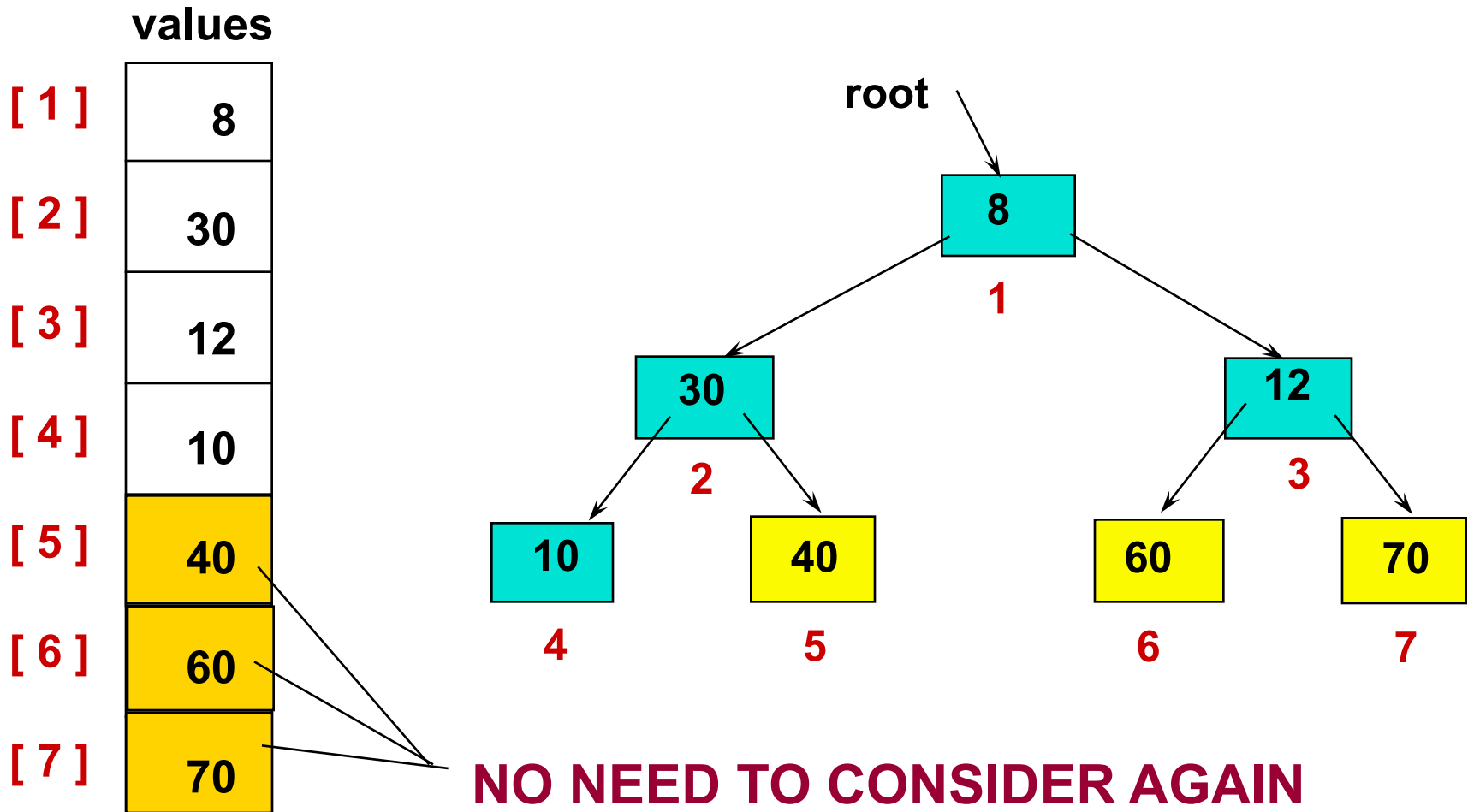
	values
[ 1 ]	40
[ 2 ]	30
[ 3 ]	12
[ 4 ]	10
[ 5 ]	8
[ 6 ]	60
[ 7 ]	70



# Swap root element into last place in unsorted array

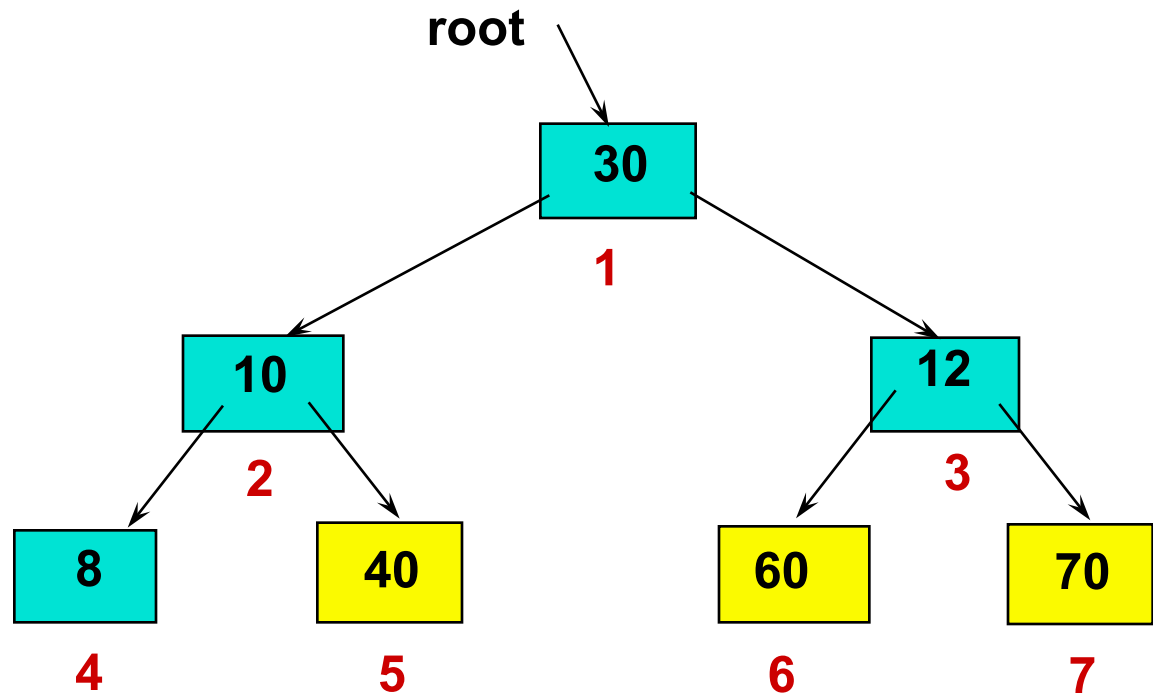


# After swapping root element into its place

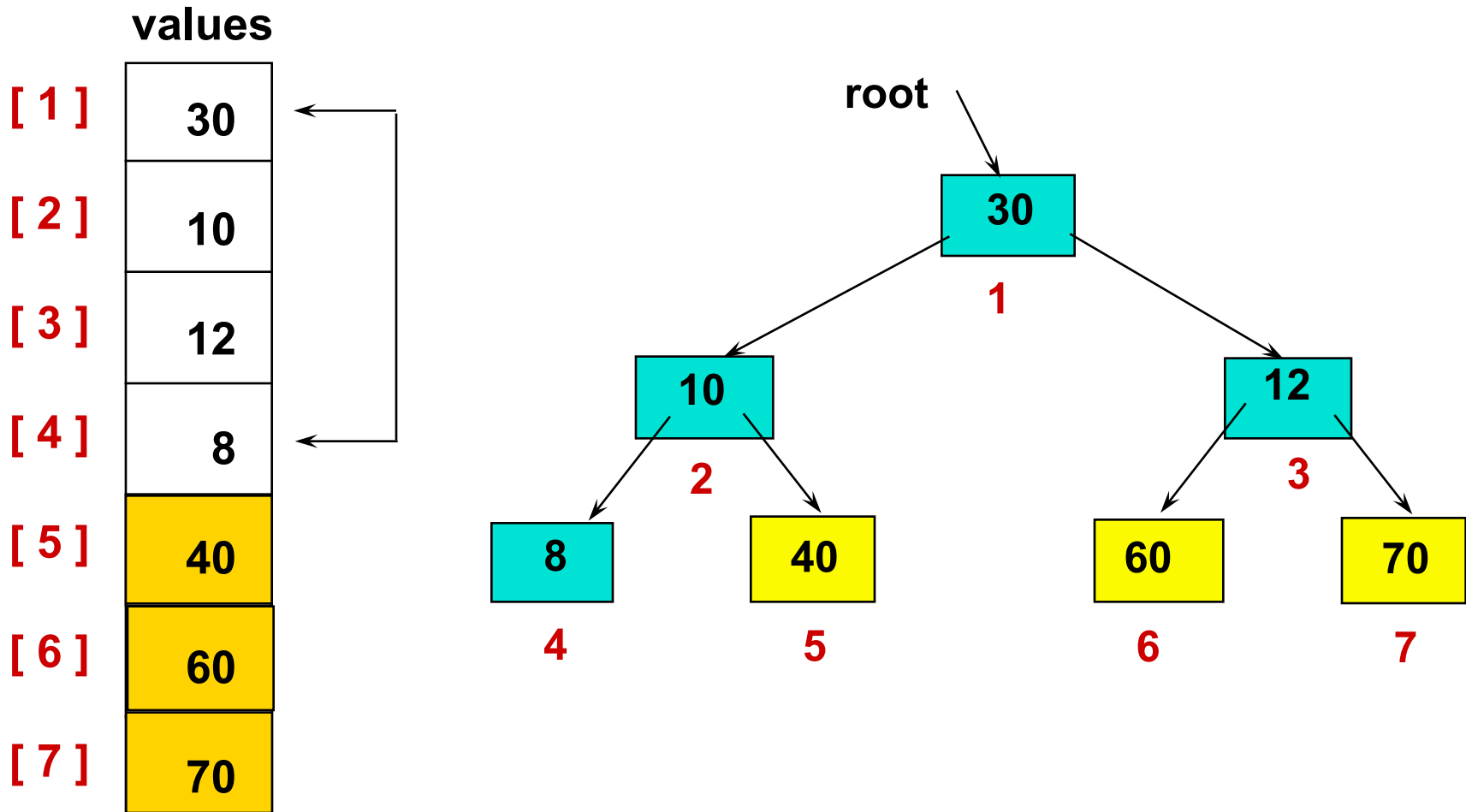


# After reheaping remaining unsorted elements

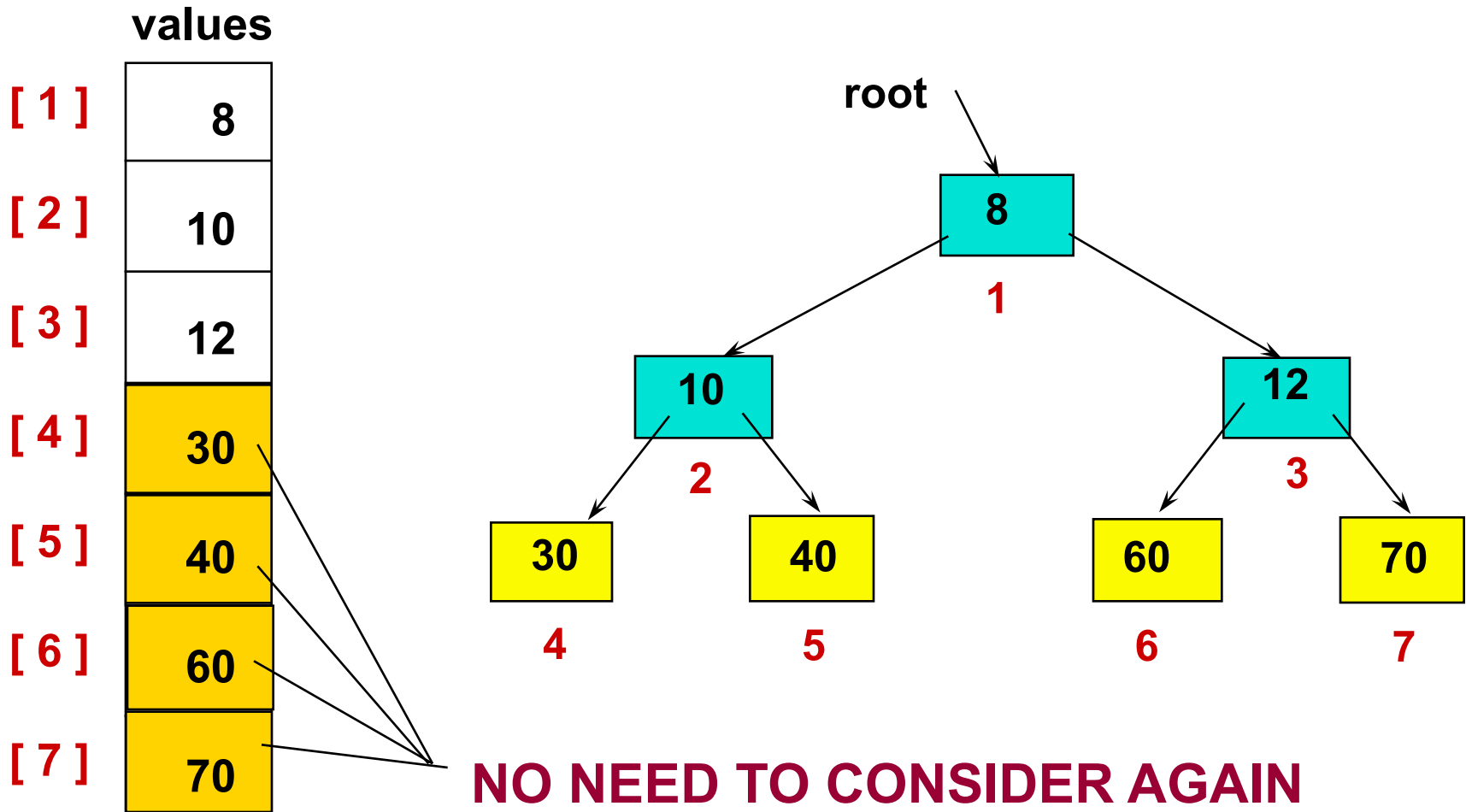
	values
[ 1 ]	30
[ 2 ]	10
[ 3 ]	12
[ 4 ]	8
[ 5 ]	40
[ 6 ]	60
[ 7 ]	70



# Swap root element into last place in unsorted array

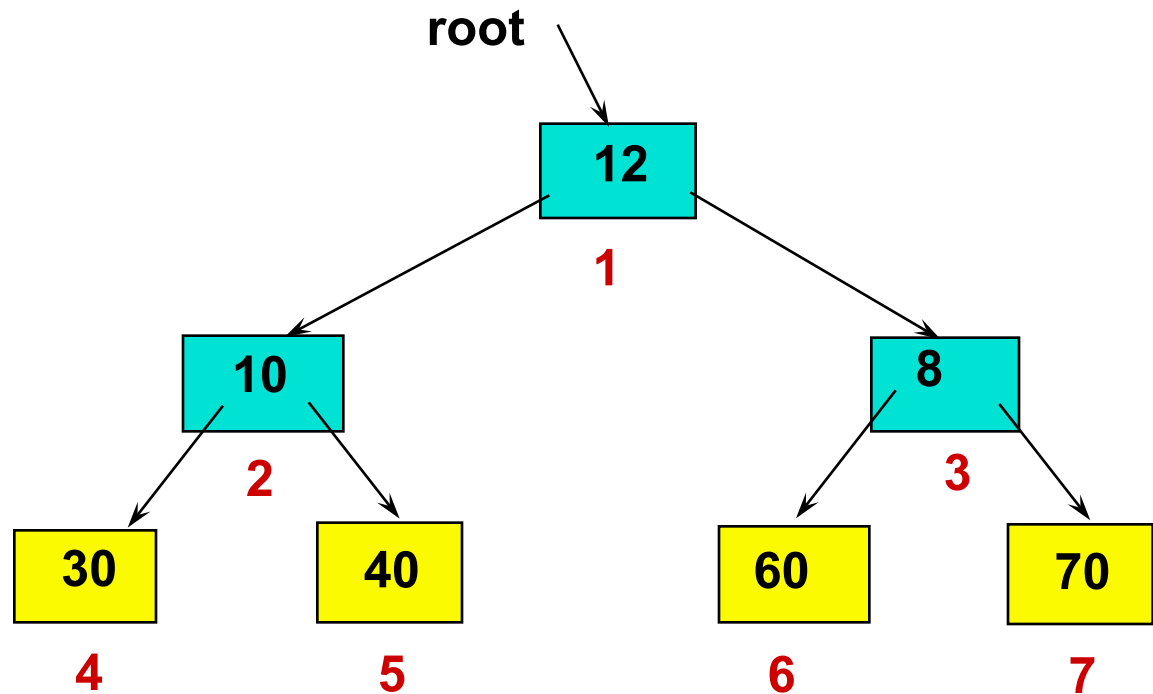


# After swapping root element into its place



# After reheaping remaining unsorted elements

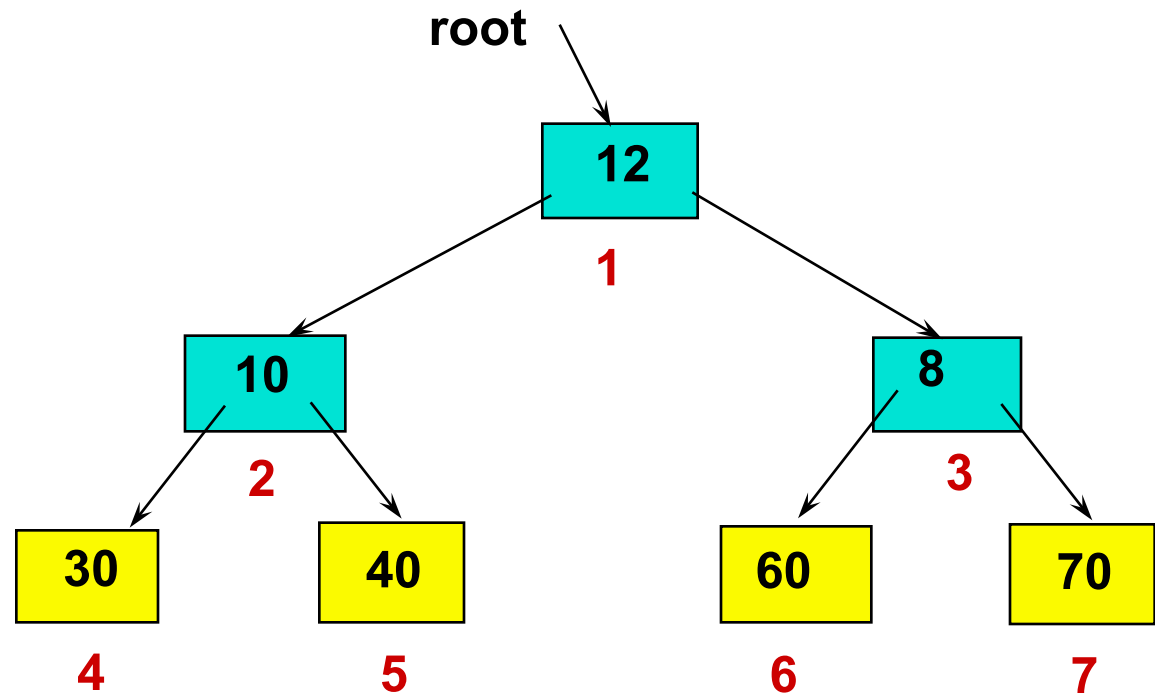

	values
[ 1 ]	12
[ 2 ]	10
[ 3 ]	8
[ 4 ]	30
[ 5 ]	40
[ 6 ]	60
[ 7 ]	70



# Swap root element into last place in unsorted array

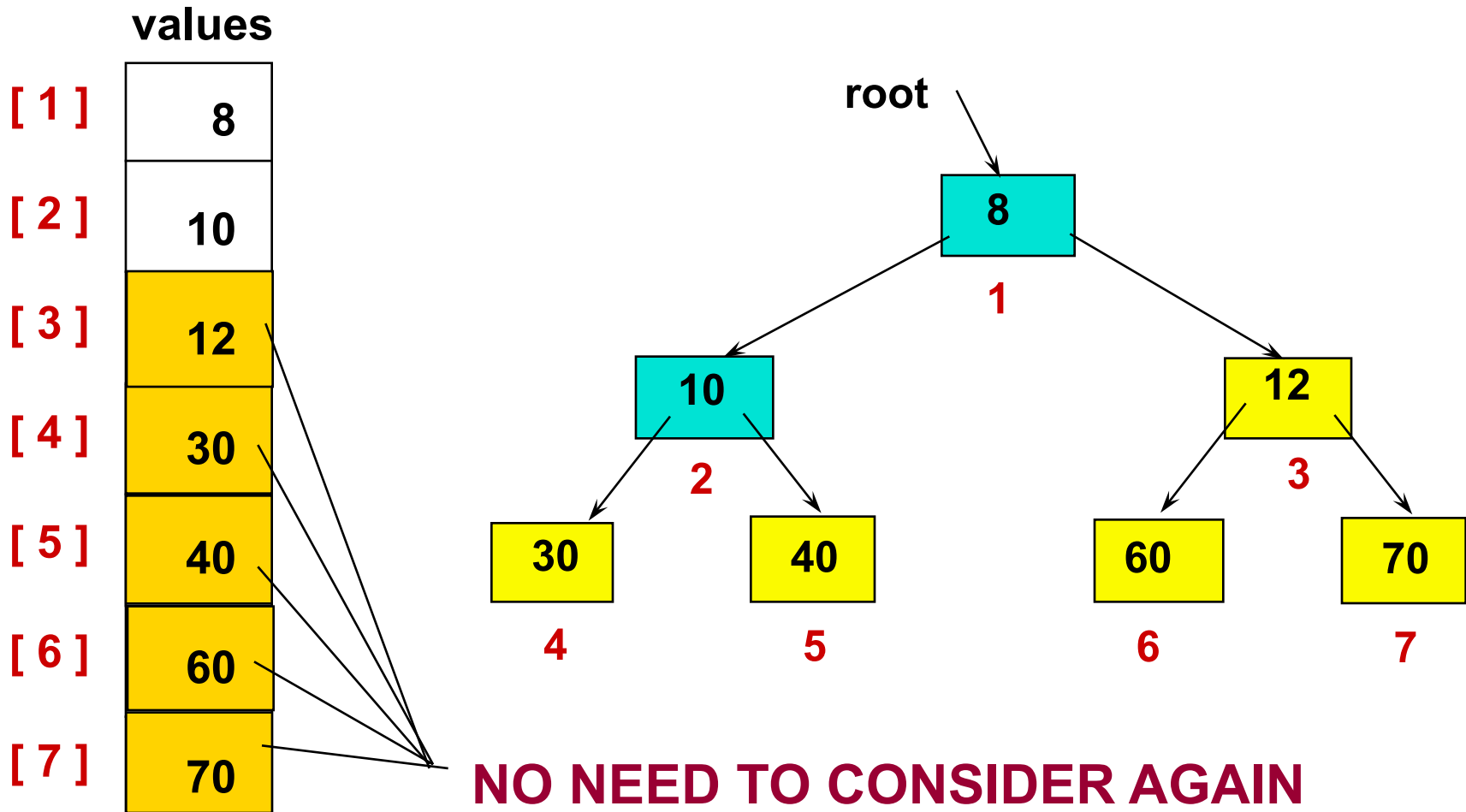
values

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



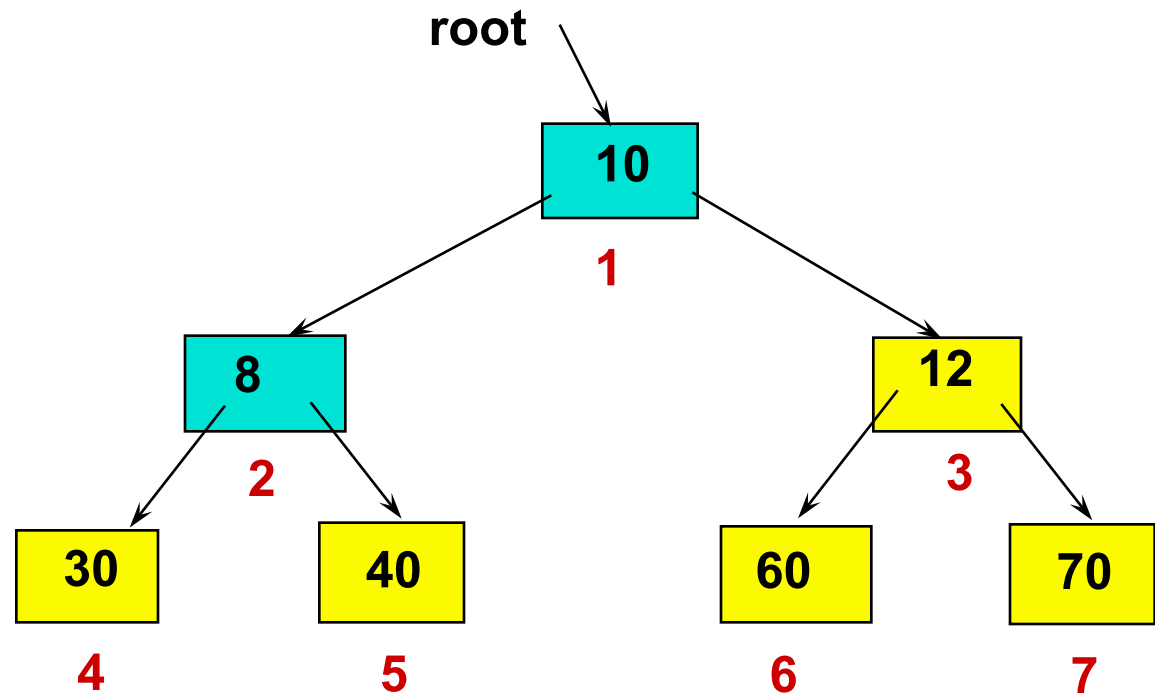


# After swapping root element into its place

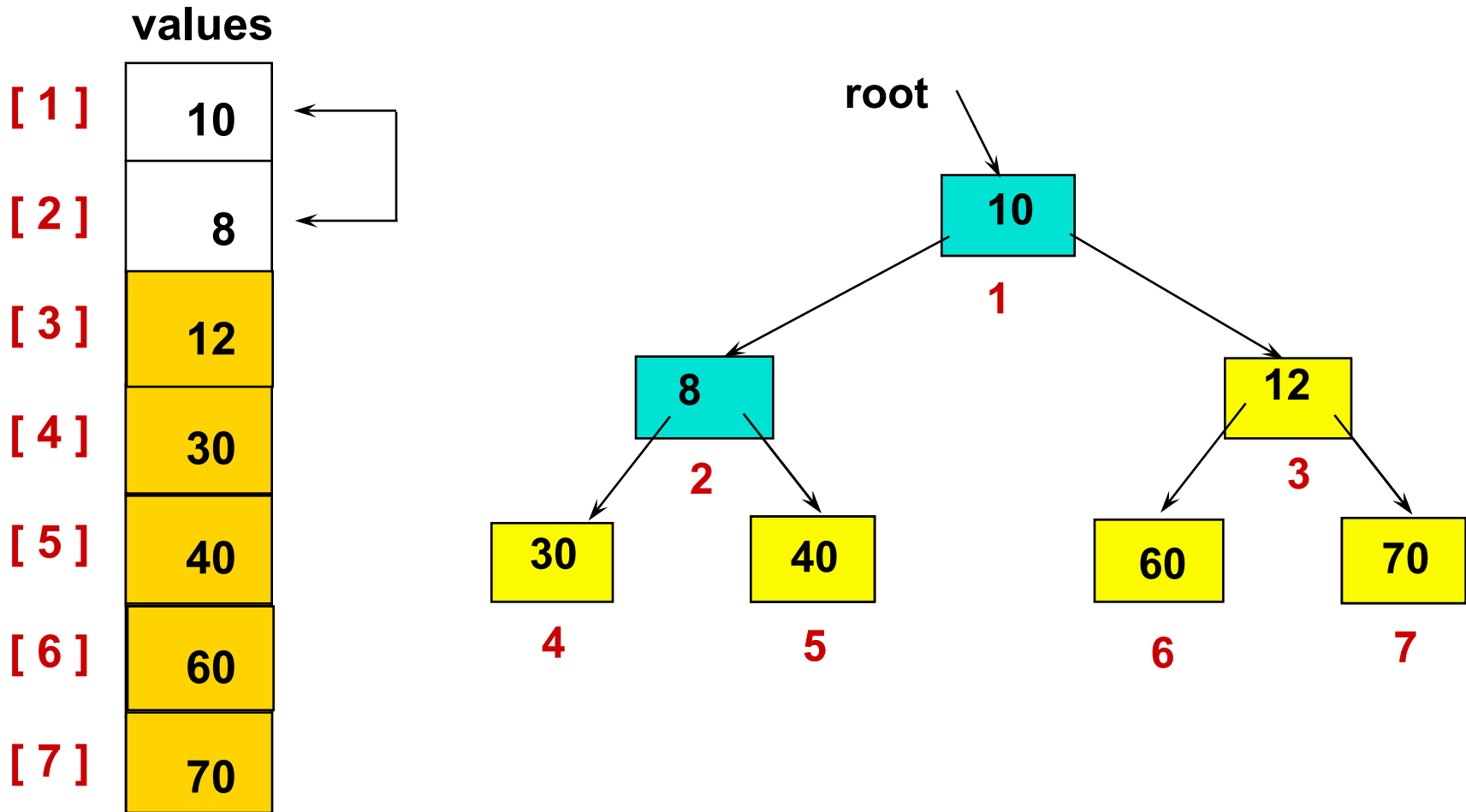


# After reheapifying remaining unsorted elements

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



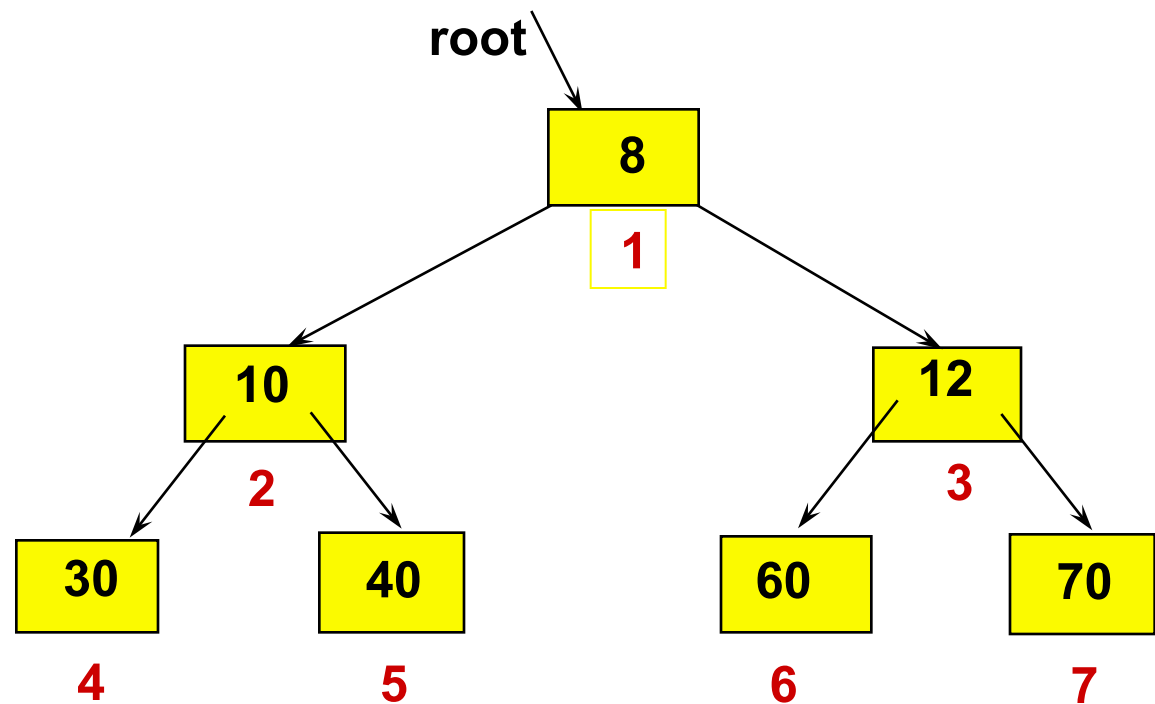
# Swap root element into last place in unsorted array



# After swapping root element into its place

数组 h

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

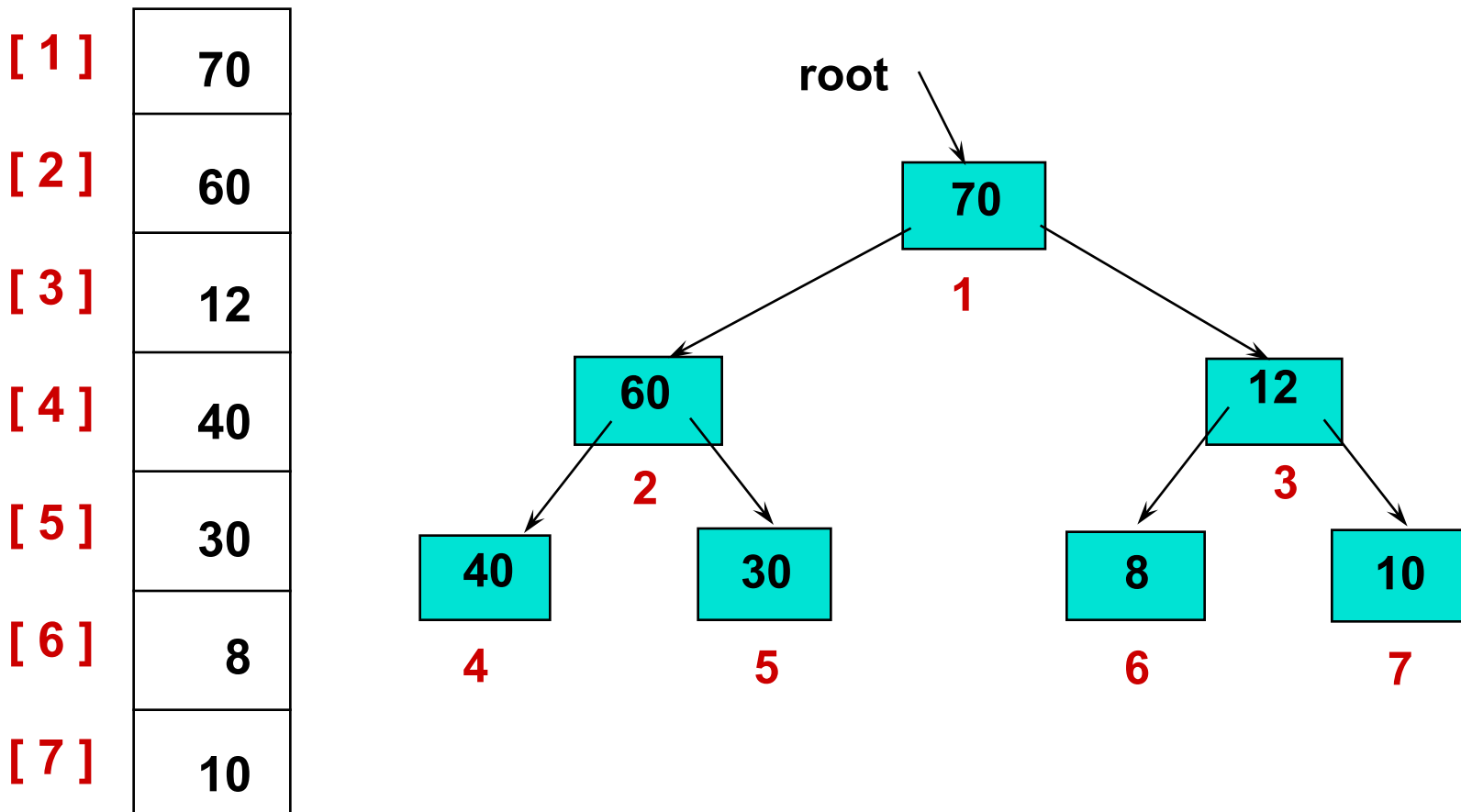


**ALL ELEMENTS ARE SORTED**

## 4、选择排序和堆排序

### 3、堆排序

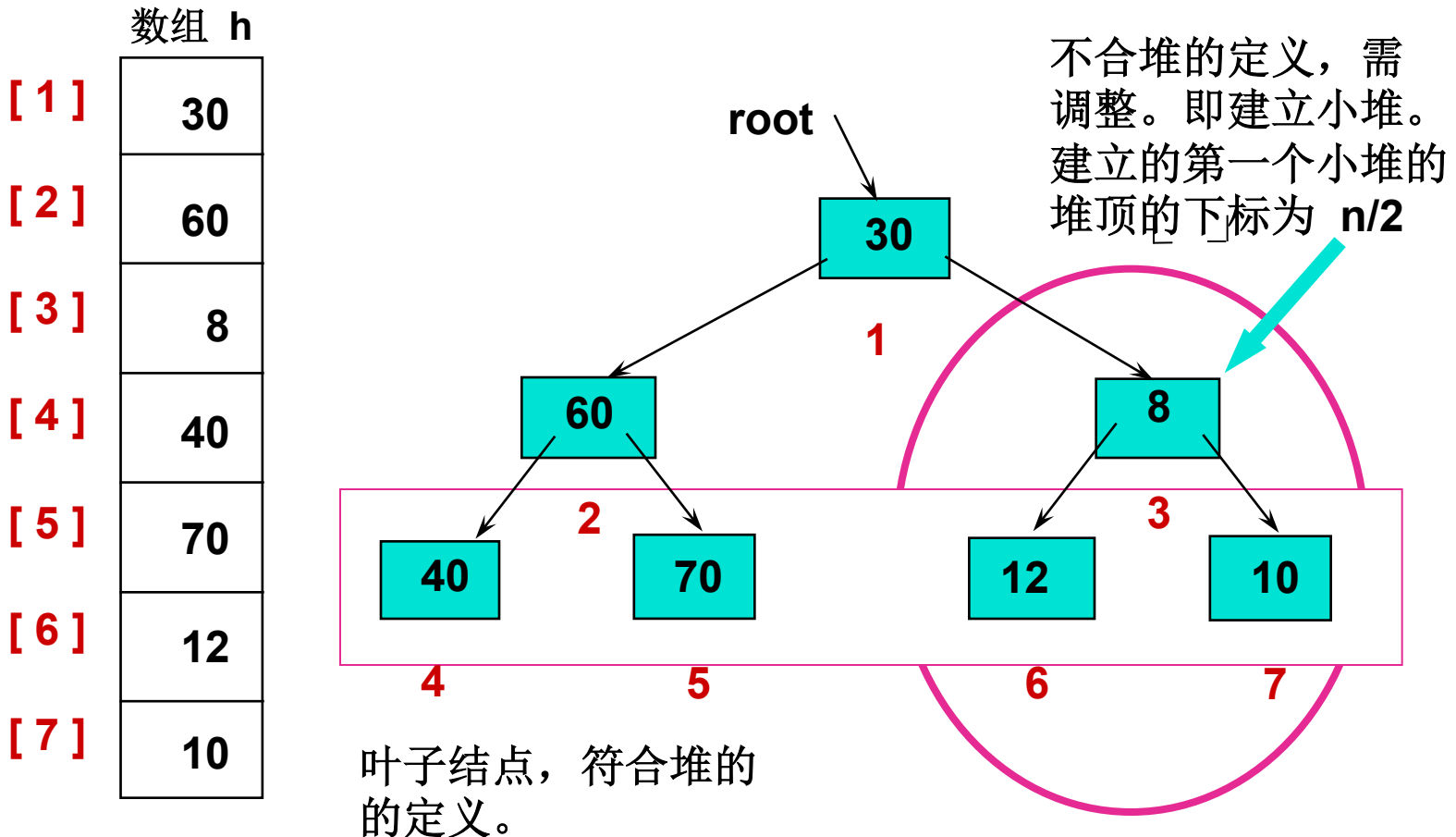
- 时间复杂性的分析：
- 建堆的时间耗费：比较次数  $4n$  次  $O(n)$  级



## 4、选择排序和堆排序

### 3、堆排序

- 时间复杂性的分析：
- 建堆的时间耗费：比较次数  $4n$  次  $O(n)$  级



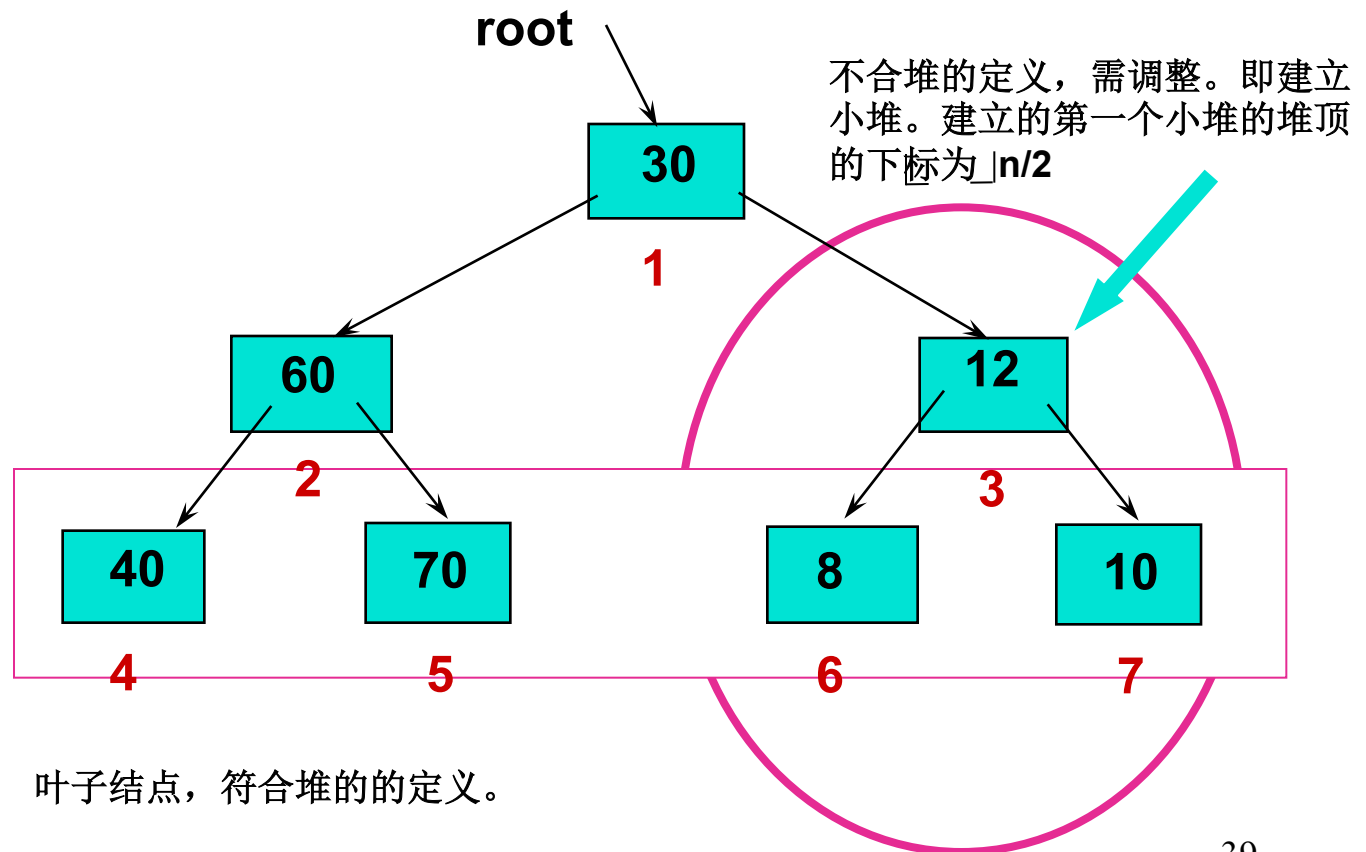
## 4、选择排序和堆排序

### 3、堆排序

- 时间复杂性的分析：
- 建堆的时间耗费：比较次数  $4n$  次  $O(n)$  级

数组 h

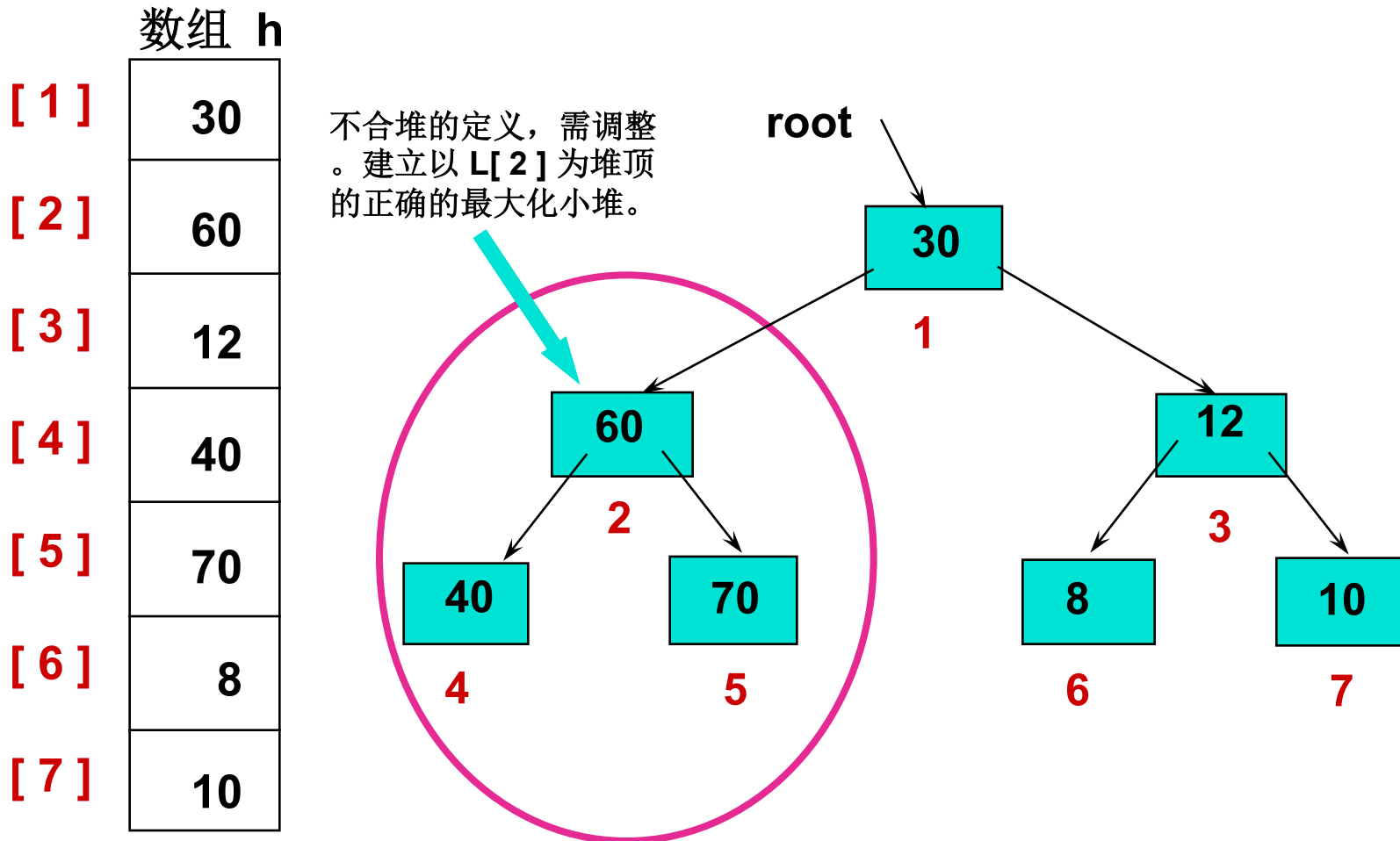
[ 1 ]	30
[ 2 ]	60
[ 3 ]	12
[ 4 ]	40
[ 5 ]	70
[ 6 ]	8
[ 7 ]	10



## 4、选择排序和堆排序

### 3、堆排序

- 时间复杂性的分析：
- 建堆的时间耗费：比较次数  $4n$  次  $O(n)$  级

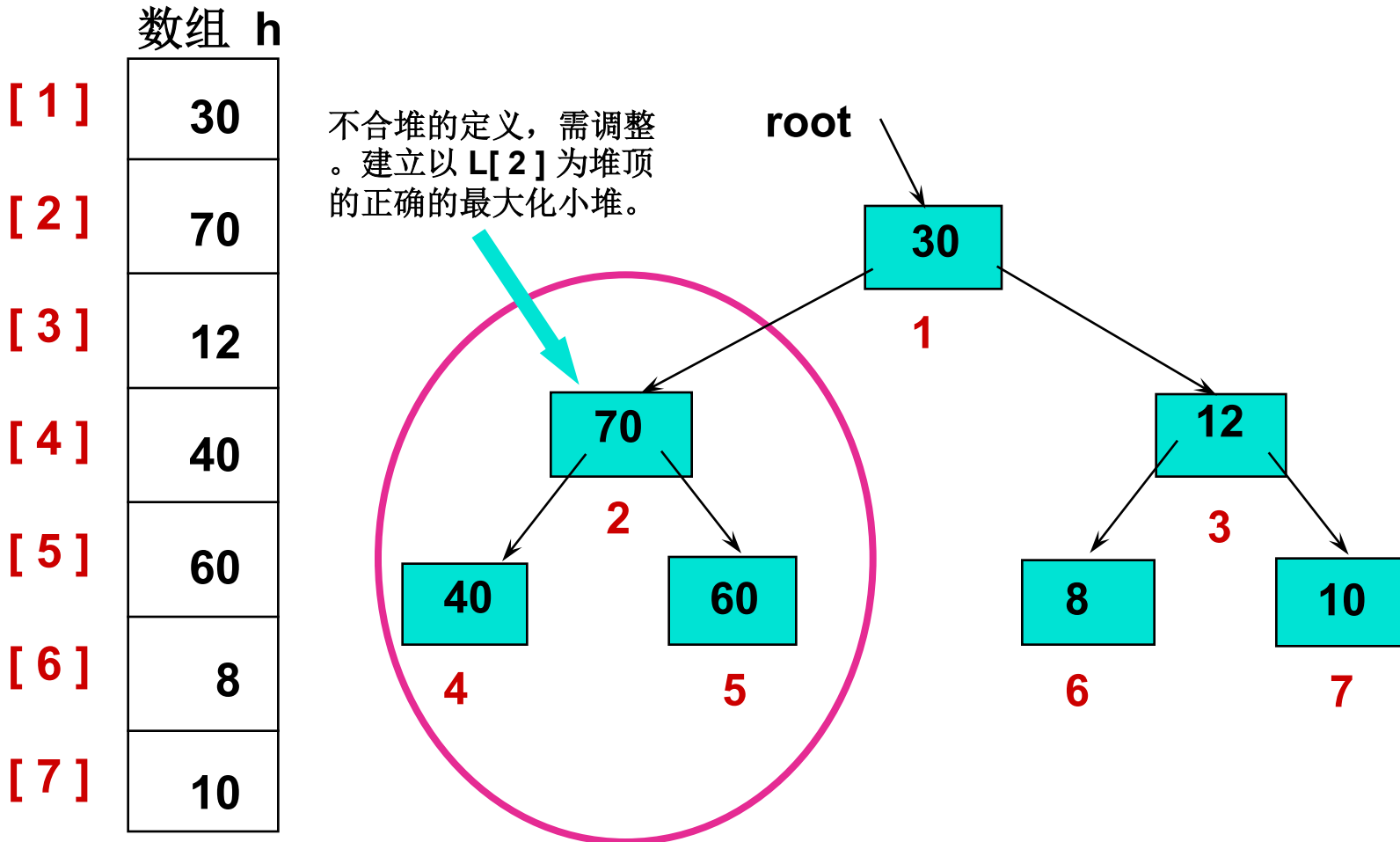




## 4、选择排序和堆排序

### 3、堆排序

- 时间复杂性的分析：
- 建堆的时间耗费：比较次数  $4n$  次  $O(n)$  级



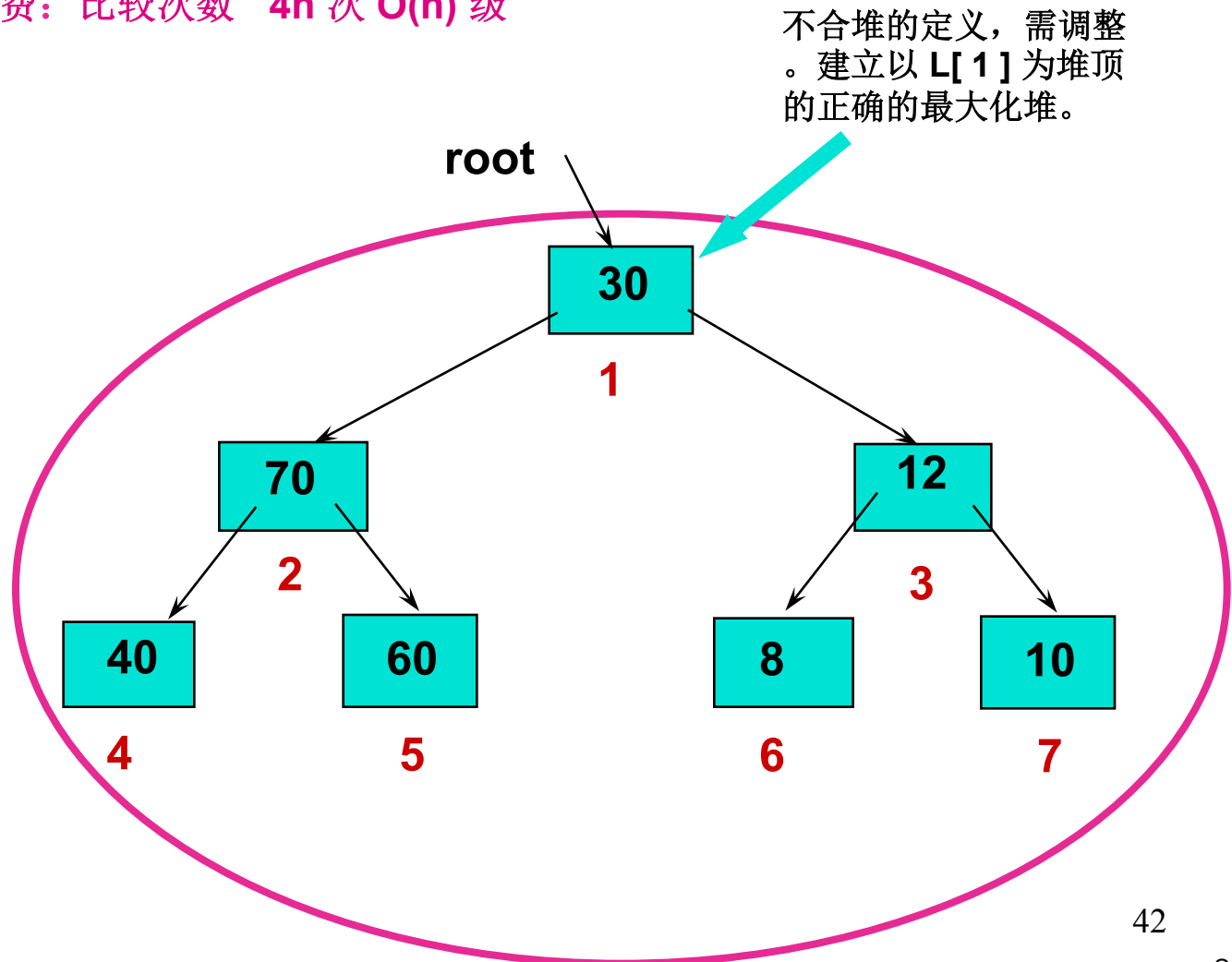
## 4、选择排序和堆排序

### 3、堆排序

- 时间复杂性的分析：
- 建堆的时间耗费：比较次数  $4n$  次  $O(n)$  级

数组 h

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



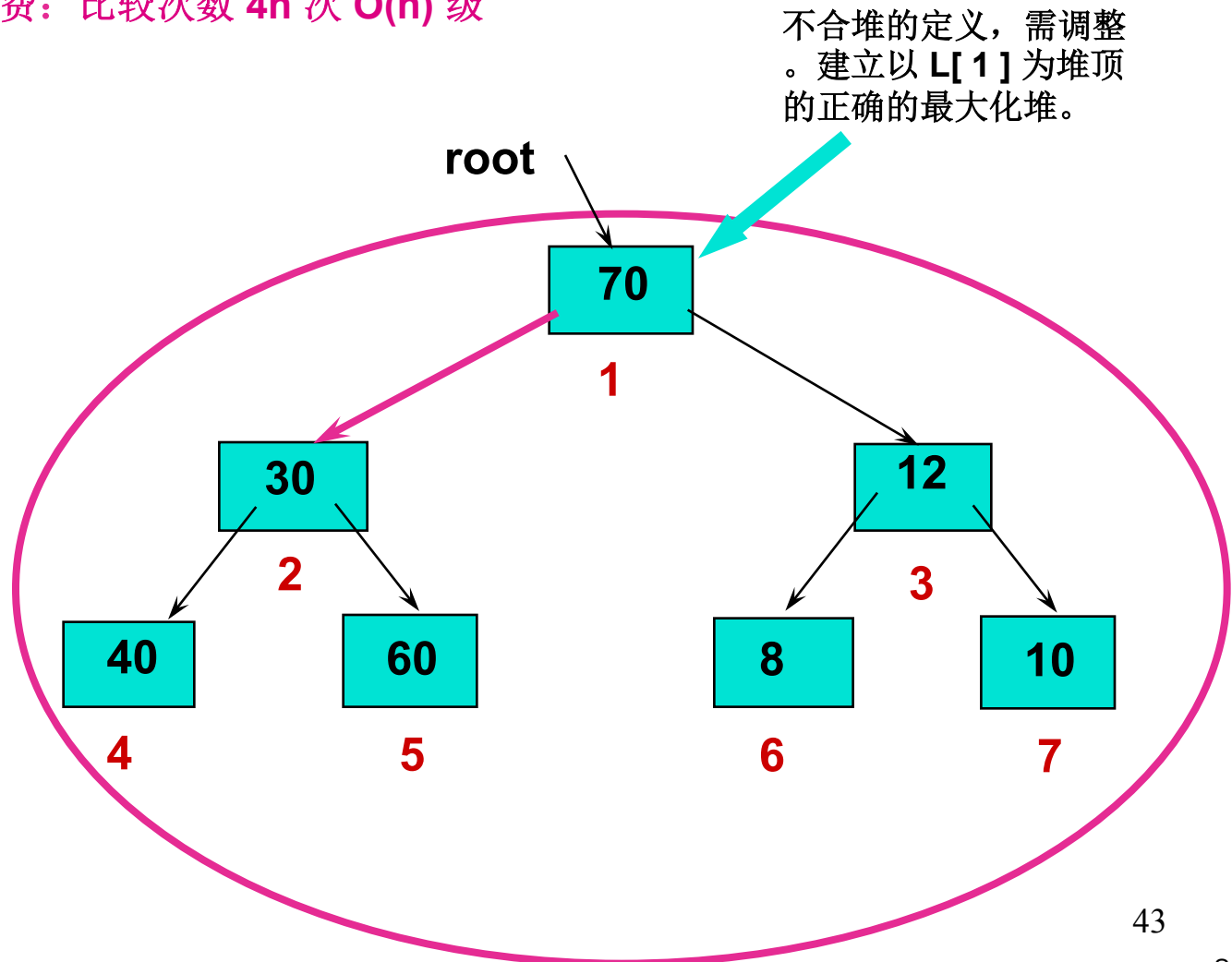
## 4、选择排序和堆排序

### 3、堆排序

- 时间复杂性的分析:
- 建堆的时间耗费: 比较次数  $4n$  次  $O(n)$  级

数组 h

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

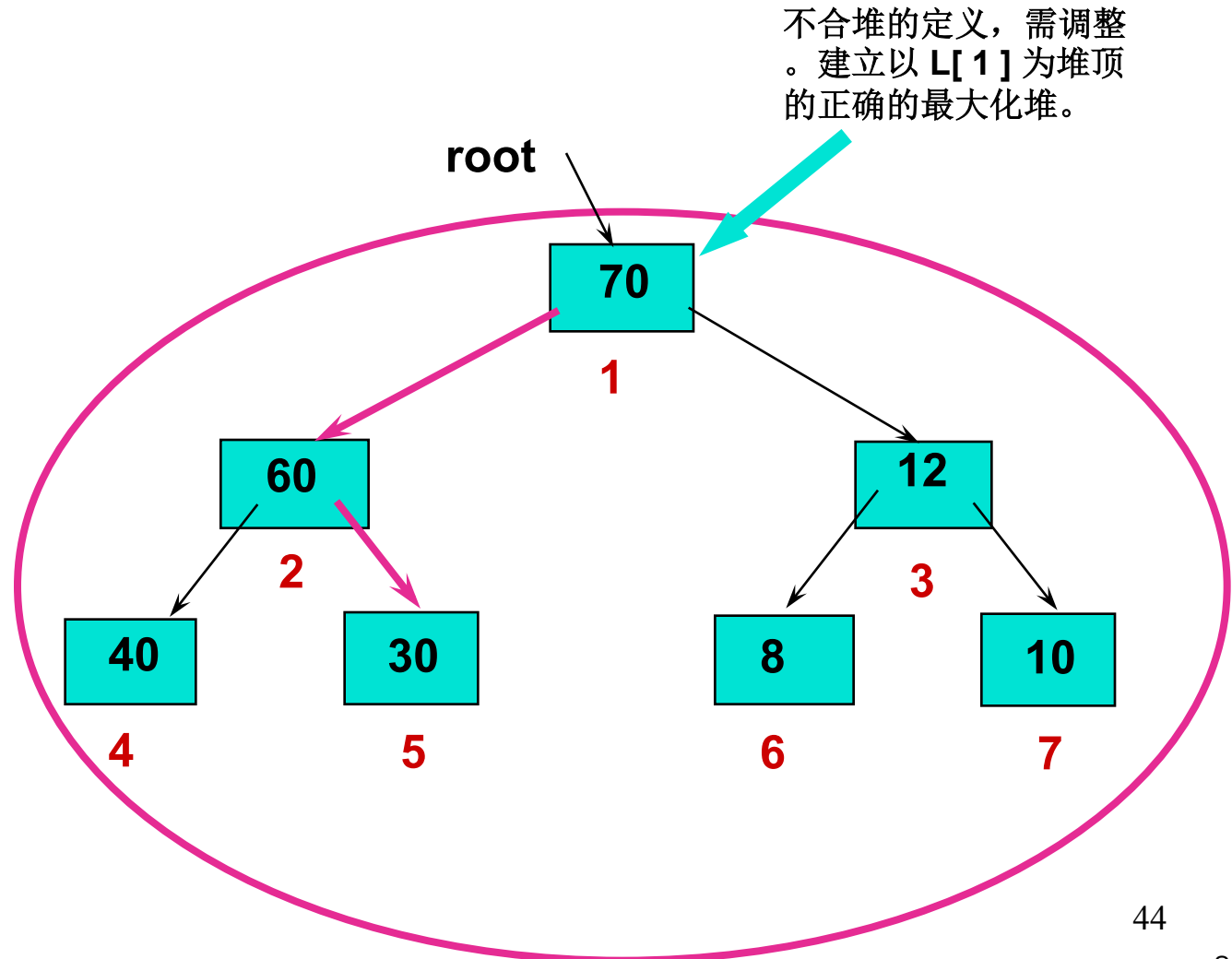


## 4、选择排序和堆排序

### 3、堆排序

- 建堆的时间耗费：比较次数  $4n$  次  $O(n)$  级

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



## 4、选择排序和堆排序

时间复杂性的分析:

- 时间耗费的代价: 建堆的时间耗费 + 排序的时间耗费
- 建堆的时间耗费: 设树根处于第 1 层, 该堆共有  $h$  层。建堆从第  $h-1$  层开始进行。只要知道了每一层的结点数 (建的小堆的个数), 和每建一个小堆所需的比较次数, 就可以求得建堆的时间耗费。
- 建的小堆的性质: 第  $i$  层上小堆的个数 = 第  $i$  层上结点的个数 = 最多  $2^{i-1}$   
 第  $i$  层上小堆的高度 =  $h-i+1$   
 建第  $i$  层上每个小堆最多所需的比较次数 =  $2 \times (h-i)$  次
- 建堆的时间耗费:

$$T(n) \leq \sum_{i=h-1}^1 2^{i-1} \times (h-i) \times 2 = \sum_{i=h-1}^1 2^i \times (h-i) = \sum_{j=1}^{h-1} 2^{h-j} \times j$$

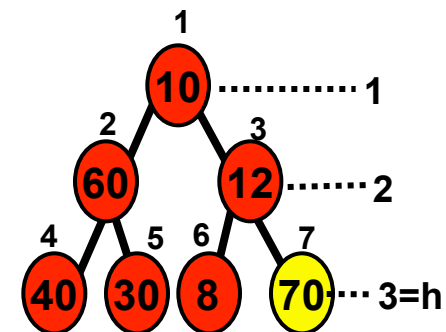
$$= 2^h \sum_{j=1}^{h-1} j/2^j$$

注意: 即使是 高度为  $h$  的完全的二叉树, 满足以下式子:

$$2^{h-1} \leq n \leq 2^h - 1 \quad \text{故: } 2^h \leq 2n$$

$$\text{又: } \sum_{j=1}^{\infty} j/2^j < 2$$

所以: 建堆的时间复杂性为  $4n = O(n)$  级



## 4、选择排序和堆排序

### 4、堆排序的实现: 建立最大化堆

```

template < class EType>
void MaxHeap<EType> :: SiftDown( int j ) {
    // heap[1], ..., heap[n]为存储堆的数组。heap[0]不用。
    int MaxSon; // 用于记录大儿子的下标地址。
        heap[0] = heap[j];
    for ( ; j*2 <= CurrentSize; j = MaxSon) {
        MaxSon = 2 * j;
        if ( MaxSon != CurrentSize && heap [MaxSon+1 ] > heap [MaxSon ] ) MaxSon++;
        if ( heap[MaxSon] > heap[0] ) heap[j] = heap[MaxSon];
        else break;
    }
    heap [j ] = heap[0];
}

template < class EType>
void MaxHeap<EType> :: BuildHeap( ) {
    for ( int j= CurrentSize/2; j >0; j-- ) SiftDown(j);
}

```

## 4、选择排序和堆排序

### 4、堆排序的实现

```

template < class EType>
void SiftDown( EType a[ ], int j, int Size ) {
    // a[1], ...,a[Size]为存储堆的数组。a[0]不用。注意：本函数用于排序。Size 为堆的最
    // 大下标，j 为当前要建立的最大化堆的堆顶结点的地址。
    int MaxSon; // 用于记录大儿子的下标地址。
    a[0] = a[j];
    for ( ; j*2 <= Size; j = MaxSon) {
        MaxSon = 2 * j;
        if ( MaxSon != Size && a[MaxSon+1 ] > a[MaxSon ] ) MaxSon++;
        if ( a[MaxSon] > a[0] ) a[j] = a[MaxSon];
        else break;
    }
    a[j] = a[0];
}

template < class EType>
void MaxHeapSort( EType a[ ], int Size ) {
    for ( int j = Size/2; j > 0; j- - ) // a[1], a[2 ], ...,a[Size]为待排序的数组。a[0]不用。
        SiftDown(a, j, Size);          // 将 a[1], a[2 ], ...,a[Size]建成最大化堆。
    for ( int k = Size; k > 1; k- - )
    {
        Swap( a[1], a[k]);             // 交换 a[1] 和 a[k ], 最大元素放在 a[k]。
        SiftDown(a, 1, k-1);           // 将 a[1], a[2 ], ...,a[k-1]调整为最大化堆。
    }
}

```

## 4、选择排序和堆排序

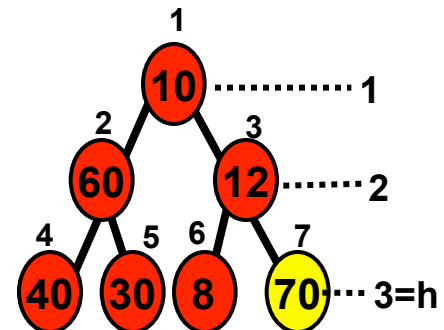
时间复杂性的分析：

- 时间复杂性的分析：建堆的时间耗费 + 排序的时间耗费
- 排序的时间耗费：

结点个数	高度	比较次数最多
$j=2$	$\lfloor \log 2 \rfloor + 1$	$\leq 2 \times \lfloor \log 2 \rfloor$
$j=3$	$\lfloor \log 3 \rfloor + 1$	$\leq 2 \times \lfloor \log 3 \rfloor$
$j=4$	$\lfloor \log 4 \rfloor + 1$	$\leq 2 \times \lfloor \log 4 \rfloor$
⋮		
$j=n-1$	$\lfloor \log(n-1) \rfloor + 1$	$\leq 2 \times \lfloor \log(n-1) \rfloor$

所以：  $T(n) \leq 2( \log 2 + \log 3 + \dots + \log(n-1) ) = 2\log(n-1)!$

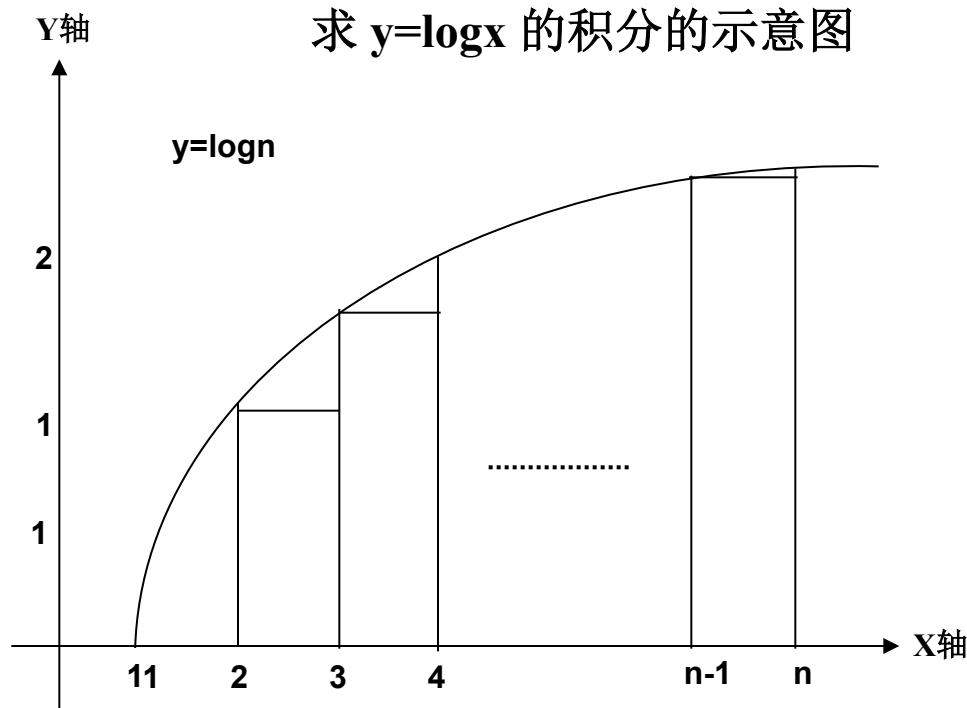
$T(n) = O( n \log n )$





## 4、选择排序和堆排序

- 堆排序的时间代价分析推导:  $O(n \log n)$



$$\log(n-1)! = \sum_{j=2}^{n-1} \log j \leq \int_1^n \log x \, dx$$

不难求出上述积分的值为:  $n \log n - n \log e + \log e \geq n \log n - n \log e$ 。注意到:  $\log e = 1.44$ 。

## 5、插入排序和希尔排序

### 1、直接插入排序

e.g: 36、24、10、6、12存放在  $r$  数组的下标为 1 至 5 的元素之中，用直接插入法将其排序。结果仍保存在下标为 1 至 5 的元素之中。

0	1	2	3	4	5
	36	24	10	6	12

$a[0]$  用作哨兵。共执行 5 遍操作。

每遍操作：先将元素复制内容放入  $a[0]$ ，再将本元素同已排序的序列，从尾开始进行比较。在已排序的序列中寻找自己的位置，进行插入。或者寻找不到，则一直进行到哨兵为止。意味着本元素最小，应该放在  $a[1]$ 。

每一遍，排序的序列将增加一个元素。如果序列中有  $n$  个元素，那么最多进行  $n$  遍即可。

## 5、插入排序和希尔排序


### 1、直接插入排序

e.g: 36、24、10、6、12存放在  $r$  数组的下标为 1 至 5 的元素之中，用直接插入法将其排序。结果仍保存在下标为 1 至 5 的元素之中。

0	1	2	3	4	5
	36	24	10	6	12

24	36	24			
----	----	----	--	--	--



## 5、插入排序和希尔排序


### 1、直接插入排序

e.g: 36、24、10、6、12存放在  $r$  数组的下标为 1 至 5 的元素之中，用直接插入法将其排序。结果仍保存在下标为 1 至 5 的元素之中。

0	1	2	3	4	5
	36	24	10	6	12

24		36			
----	--	----	--	--	--



## 5、插入排序和希尔排序


### 1、直接插入排序

e.g: 36、24、10、6、12存放在  $r$  数组的下标为 1 至 5 的元素之中，用直接插入法将其排序。结果仍保存在下标为 1 至 5 的元素之中。

0	1	2	3	4	5
	36	24	10	6	12

24	24	36			
----	----	----	--	--	--



## 5、插入排序和希尔排序


### 1、直接插入排序

e.g: 36、24、10、6、12存放在  $r$  数组的下标为 1 至 5 的元素之中，用直接插入法将其排序。结果仍保存在下标为 1 至 5 的元素之中。

0	1	2	3	4	5
	36	24	10	6	12

10	24	36	10		
----	----	----	----	--	--



## 5、插入排序和希尔排序


### 1、直接插入排序

e.g: 36、24、10、6、12存放在  $r$  数组的下标为 1 至 5 的元素之中，用直接插入法将其排序。结果仍保存在下标为 1 至 5 的元素之中。

0	1	2	3	4	5
	36	24	10	6	12

10	24		36		
----	----	--	----	--	--



## 5、插入排序和希尔排序


### 1、直接插入排序

e.g: 36、24、10、6、12存放在  $r$  数组的下标为 1 至 5 的元素之中，用直接插入法将其排序。结果仍保存在下标为 1 至 5 的元素之中。

0	1	2	3	4	5
	36	24	10	6	12

10		24	36		
----	--	----	----	--	--





## 5、插入排序和希尔排序


### 1、直接插入排序

e.g: 36、24、10、6、12存放在  $r$  数组的下标为 1 至 5 的元素之中，用直接插入法将其排序。结果仍保存在下标为 1 至 5 的元素之中。

0	1	2	3	4	5
	36	24	10	6	12

10	10	24	36		
----	----	----	----	--	--



## 5、插入排序和希尔排序


### 1、直接插入排序

e.g: 36、24、10、6、12存放在  $r$  数组的下标为 1 至 5 的元素之中，用直接插入法将其排序。结果仍保存在下标为 1 至 5 的元素之中。

0	1	2	3	4	5
	36	24	10	6	12

6	10	24	36	6	
---	----	----	----	---	--



## 5、插入排序和希尔排序


### 1、直接插入排序

e.g: 36、24、10、6、12存放在  $r$  数组的下标为 1 至 5 的元素之中，用直接插入法将其排序。结果仍保存在下标为 1 至 5 的元素之中。

0	1	2	3	4	5
	36	24	10	6	12

6	10	24		36	
---	----	----	--	----	--



## 5、插入排序和希尔排序


### 1、直接插入排序

e.g: 36、24、10、6、12存放在  $r$  数组的下标为 1 至 5 的元素之中，用直接插入法将其排序。结果仍保存在下标为 1 至 5 的元素之中。

0	1	2	3	4	5
	36	24	10	6	12

6	10		24	36	
---	----	--	----	----	--



## 5、插入排序和希尔排序


### 1、直接插入排序

e.g: 36、24、10、6、12存放在  $r$  数组的下标为 1 至 5 的元素之中，用直接插入法将其排序。结果仍保存在下标为 1 至 5 的元素之中。

0	1	2	3	4	5
	36	24	10	6	12

6		10	24	36	
---	--	----	----	----	--



## 5、插入排序和希尔排序


### 1、直接插入排序

e.g: 36、24、10、6、12存放在  $r$  数组的下标为 1 至 5 的元素之中，用直接插入法将其排序。结果仍保存在下标为 1 至 5 的元素之中。

0	1	2	3	4	5
	36	24	10	6	12

6	6	10	24	36	
---	---	----	----	----	--



## 5、插入排序和希尔排序

### 1、直接插入排序

e.g: 36、24、10、6、12存放在  $r$  数组的下标为 1 至 5 的元素之中，用直接插入法将其排序。结果仍保存在下标为 1 至 5 的元素之中。

0	1	2	3	4	5
	36	24	10	6	12
12	6	10	24	36	12

## 5、插入排序和希尔排序

### 1、直接插入排序

e.g: 36、24、10、6、12存放在  $r$  数组的下标为 1 至 5 的元素之中，用直接插入法将其排序。结果仍保存在下标为 1 至 5 的元素之中。

0	1	2	3	4	5
	36	24	10	6	12
12	6	10	24		36



## 5、插入排序和希尔排序

### 1、直接插入排序


e.g: 36、24、10、6、12存放在  $r$  数组的下标为 1 至 5 的元素之中，用直接插入法将其排序。结果仍保存在下标为 1 至 5 的元素之中。

0	1	2	3	4	5
	36	24	10	6	12
12	6	10		24	36

## 5、插入排序和希尔排序

### 1、直接插入排序

e.g: 36、24、10、6、12存放在  $r$  数组的下标为 1 至 5 的元素之中，用直接插入法将其排序。结果仍保存在下标为 1 至 5 的元素之中。

0	1	2	3	4	5
	36	24	10	6	12
					
12	6	10	12	24	36

## 5、插入排序和希尔排序

### 1、直接插入排序

0	1	2	3	4	5
	10	20	30	40	50

	50	40	30	20	10
--	----	----	----	----	----

分析: 移动、比较次数可作为衡量时间复杂性的标准

正序时: 比较次数  $\sum_{i=2}^n 1 = n-1$

移动次数  $2(n-1)$

逆序时: 比较次数  $\sum_{i=2}^n i = (n+2)(n-1)/2$

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

## 5、插入排序和希尔排序

### 1、直接插入排序

0	1	2	3	4	5
	36	24	10	6	12
	6	10	24	36	12
12	6	10	12	24	36

程序实现:

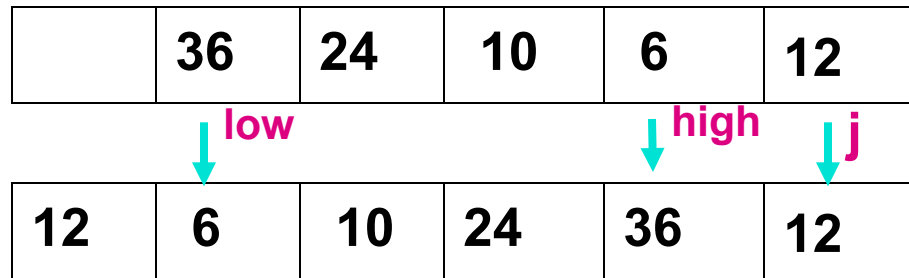
```

template < class EType>
void SimpleInsertSort( EType a[ ], int Size ) {
    int j, q;    // a[1], a[2 ], ...,a[Size]为待排序的数组。a[0] 为哨兵单元。
    EType temp=a[0];
    for ( j = 2; j <= Size; j++ )
    { a[0] = a[j];
      for ( q = j; a[0] < a[q-1]; q- -)a[q] =a[q-1];
      a[q] = a[0];
    }
    a[0] = temp;    // 恢复a[0]原来的值， 用于快速排序法中。
}

```

## 5、插入排序和希尔排序

### 2、折半插入排序



方法：在已排序的序列中查找下一元素的插入位置，将该元素插入进去，形成更长的排序序列。  
如：12 的插入位置为下标 3。

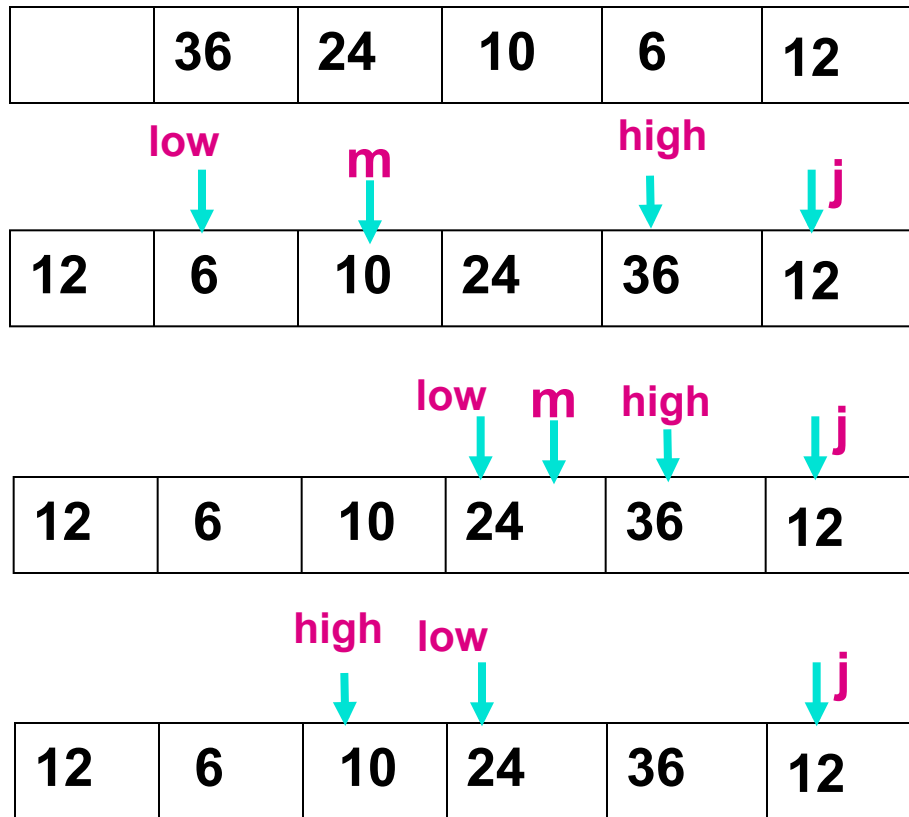
减少了比较次数，未降低时间复杂性。

程序实现：

```
template < class EType>
void BinaryInsertSort( EType a[ ], int Size )
{
    int low , high; // a[1], a[2 ], ...,a[Size]为待排序的数组。a[0]用哨兵单元。
    int j, k, mid;
    for ( j = 2; j <= Size; j++ ) {
        a[0]=a[j]; low = 1; high = j -1; // 在 a[ 1 ], ..., a[ j-1] 范围内寻找a[j] 的插入位置。
        while ( low <= high ) {
            mid = ( low + high ) / 2; // 中点下标。
            if ( a[0] < a[mid] ) high = mid - 1; // 小于中点，查找左段。
            else low = mid +1; // 大于等于中点，查找右段。
        }
        for ( k = j-1; k >= low; - - k ) a[ k+1] = a[ k ];
        a[ low ] = a[0];
    }
}
```

## 5、插入排序和希尔排序

### 2、折半插入排序



方法：在已排序的序列中查找下一元素的插入位置，将该元素插入进去，形成更长的排序序列。  
如：12 的插入位置为下标 3。

减少了比较次数，未降低时间复杂性。

注意：low 指针总是指向新关键字的下标地址，如：12 的插入位置为3。

## 5、插入排序和希尔排序

### 2、shell 排序

**e.g:** 将序列 **49、38、65、97、76、13、27、49、55、4** 用 **shell** 排序的方法进行排序

1、选定步长序列，如选为 **8、4、2、1**

2、针对步长序列进行排序，从最大的步长开始，逐步减少步长，最后一次选择的步长肯定为 **1**。

步长 8: **49、38、65、97、76、13、27、49、55、4**

步长 8: **49、38、65、97、76、13、27、49、55、4**



## 5、插入排序和希尔排序

### 2、shell 排序

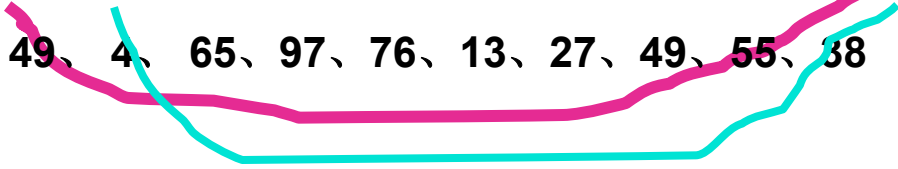
**e.g:** 将序列 **49、38、65、97、76、13、27、49、55、4** 用 **shell** 排序的方法进行排序

1、选定步长序列，如选为 **8、4、2、1**

2、针对步长序列进行排序，从最大的步长开始，逐步减少步长，最后一次选择的步长肯定为 **1**。

步长 **8**: **49、38、65、97、76、13、27、49、55、4**

步长 **8**: **49、4、65、97、76、13、27、49、55、38**



步长为 **8** 的序列的排序结束。



## 5、插入排序和希尔排序

### 2、shell 排序

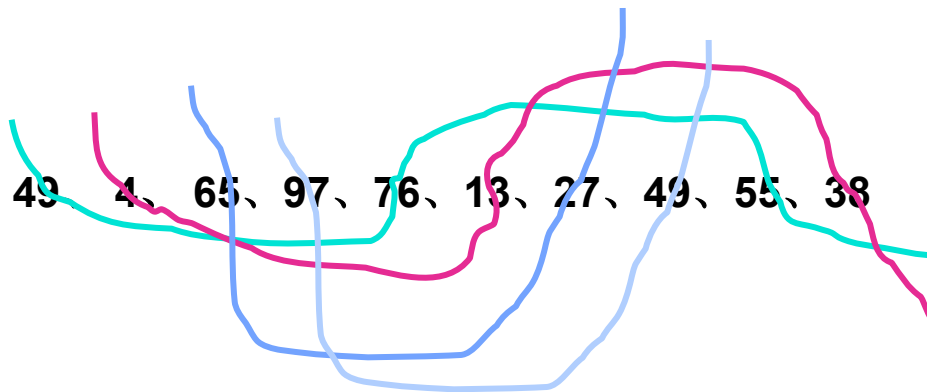
**e.g:** 将序列 **49、38、65、97、76、13、27、49、55、4** 用 **shell** 排序的方法进行排序

1、选定步长序列，如选为 **8、4、2、1**

2、针对步长序列进行排序，从最大的步长开始，逐步减少步长，最后一次选择的步长肯定为 **1**。

步长 4: **49、4、65、97、76、13、27、49、55、38**

步长 4: **49、4、65、97、76、13、27、49、55、38**



## 5、插入排序和希尔排序

### 2、shell 排序

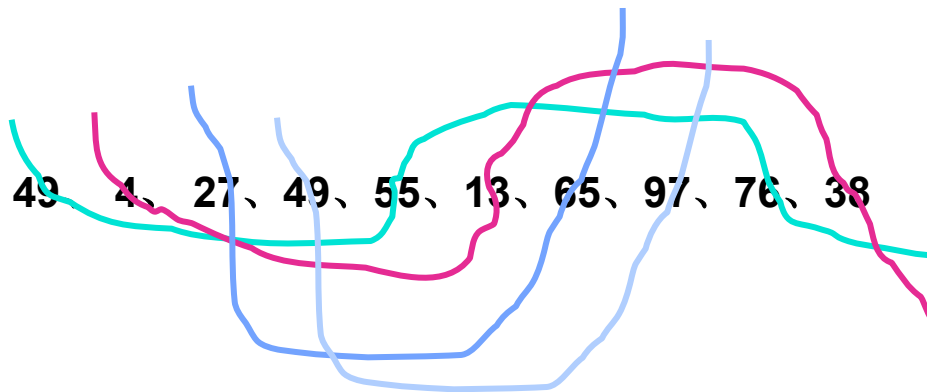
**e.g:** 将序列 **49、38、65、97、76、13、27、49、55、4** 用 **shell** 排序的方法进行排序

1、选定步长序列，如选为 **8、4、2、1**

2、针对步长序列进行排序，从最大的步长开始，逐步减少步长，最后一次选择的步长肯定为 **1**。

步长 4: **49、4、65、97、76、13、27、49、55、38**

步长 4: **49、4、27、49、55、13、65、97、76、38**



## 5、插入排序和希尔排序

### 2、shell 排序

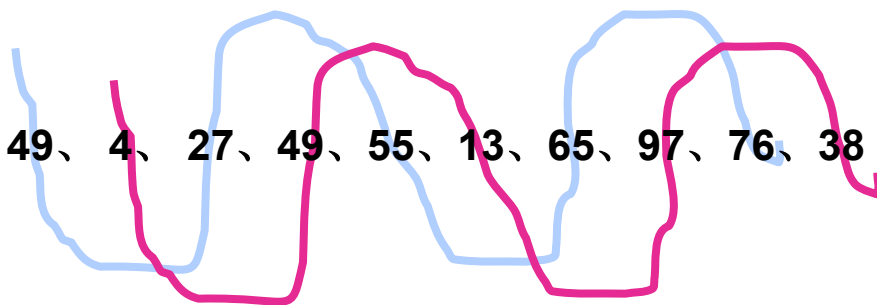
**e.g:** 将序列 **49、38、65、97、76、13、27、49、55、4** 用 **shell** 排序的方法进行排序

1、选定步长序列，如选为 **8、4、2、1**

2、针对步长序列进行排序，从最大的步长开始，逐步减少步长，最后一次选择的步长肯定为 **1**。

步长 2: **49、4、27、49、55、13、65、97、76、38**

步长 2: **49、4、27、49、55、13、65、97、76、38**



## 5、插入排序和希尔排序

### 2、shell 排序

**e.g:** 将序列 **49、38、65、97、76、13、27、49、55、4** 用 **shell** 排序的方法进行排序

1、选定步长序列，如选为 **8、4、2、1**

2、针对步长序列进行排序，从最大的步长开始，逐步减少步长，最后一次选择的步长肯定为 **1**。

步长 2: **49、4、27、49、55、13、65、97、76、38**

步长 2: **27、4、49、13、55、38、65、49、76、97**



## 5、插入排序和希尔排序

### 2、shell 排序

**e.g:** 将序列 **49、38、65、97、76、13、27、49、55、4** 用 **shell** 排序的方法进行排序

1、选定步长序列，如选为 **8、4、2、1**

2、针对步长序列进行排序，从最大的步长开始，逐步减少步长，最后一次选择的步长肯定为 **1**。

步长 **1:** **27、4、49、13、55、38、65、49、76、97**

步长 **1:** **4、13、27、38、49、49、55、65、76、97**      最后的排序结果

分析: **shell** 排序的分析非常困难，原因是何种步长序列最优难以断定。通常认为时间复杂性为:  $O(n^{3/2})$ 。

较好的步长序列: ..... **121、40、13、4、1**; 可由递推公式  $S_i = 3S_{i-1} + 1$  产生。

程序实现: 类似于直接插入排序的程序。

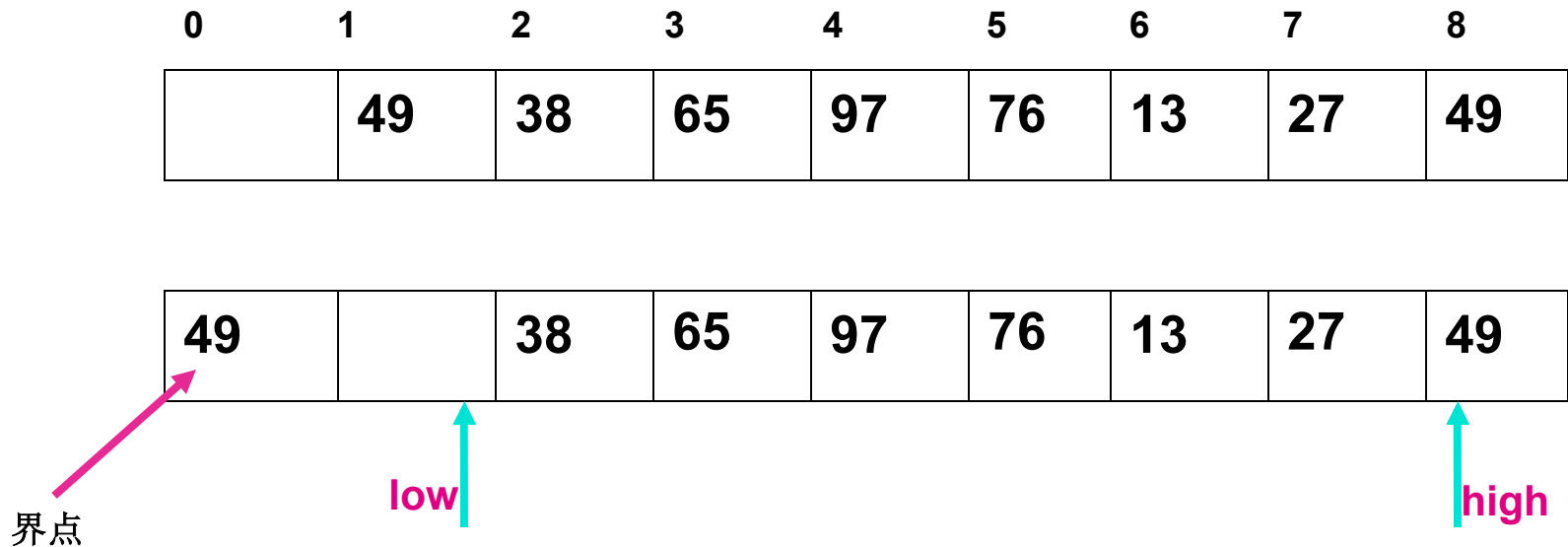
注意修改步长。

## 6、快速排序

### · 快速排序

- 原理: 若序列中有  $n$  个元素, 任选一个关键字作为界点, 将序列分成两部分。其中左半部分的结点的关键字小于等于界点, 右半部分的结点的关键字大于等于界点。然后, 对左右两部分分别进行类似的处理, 直至排好序为止。

e.g: 将序列 **49、38、65、97、76、13、27、49** 用 快速排序的方法进行排序。

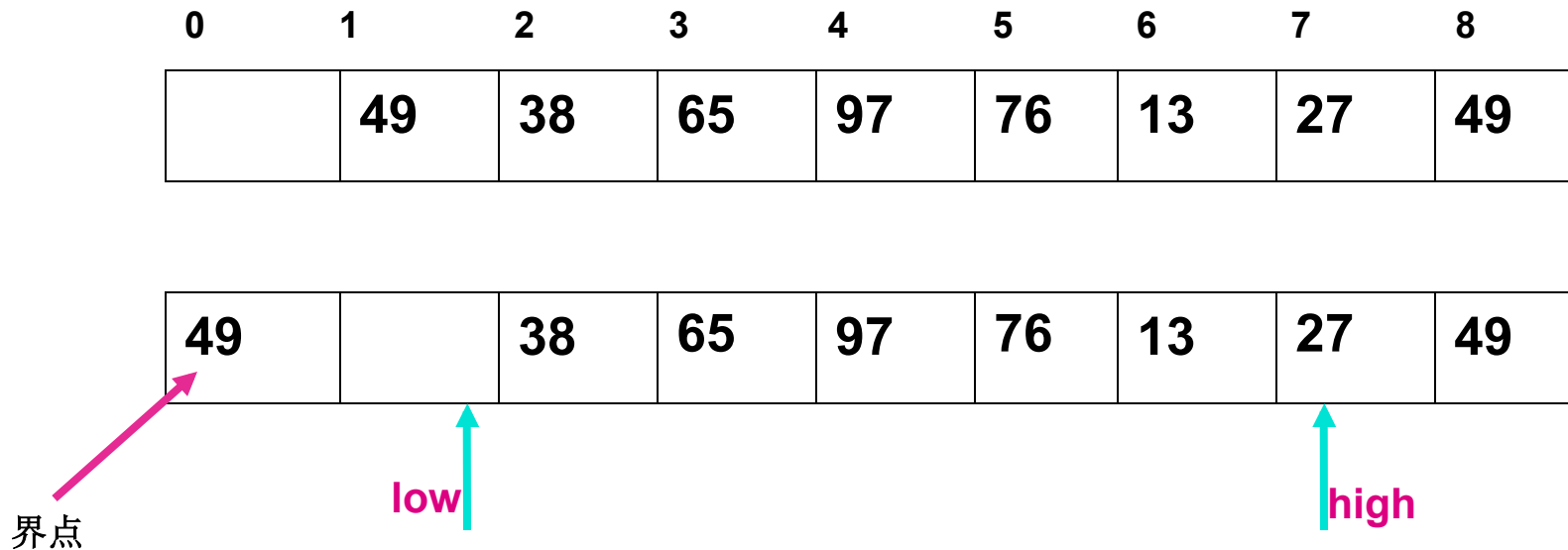


## 6、快速排序

### · 快速排序

- 原理: 若序列中有  $n$  个元素, 任选一个界点的关键字作为界点, 将序列分成两部分。其中左半部分的结点的关键字小于等于界点, 右半部分的结点的关键字大于等于界点。然后, 对左右两部分分别进行类似的处理, 直至排好序为止。

**e.g:** 将序列 **49、38、65、97、76、13、27、49** 用 快速排序的方法进行排序。

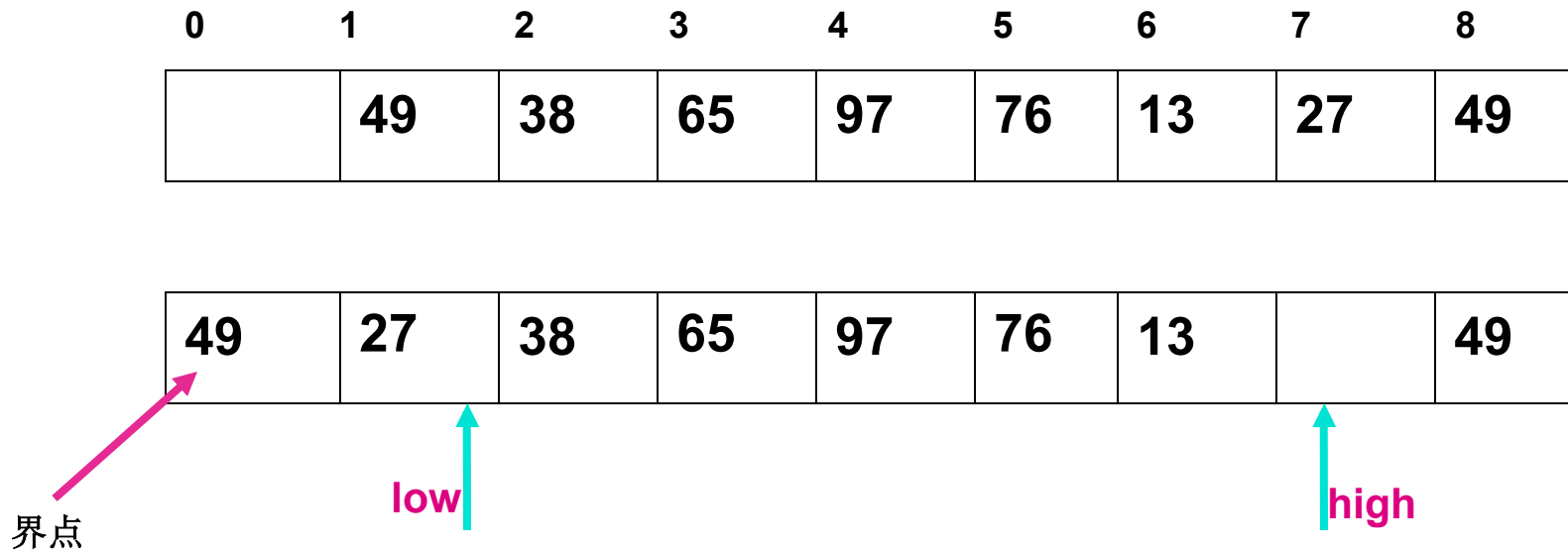


## 6、快速排序

### · 快速排序

- 原理: 若序列中有  $n$  个元素, 任选一个关键字作为界点, 将序列分成两部分。其中左半部分的结点的关键字小于等于界点, 右半部分的结点的关键字大于等于界点。然后, 对左右两部分分别进行类似的处理, 直至排好序为止。

e.g: 将序列 **49、38、65、97、76、13、27、49** 用 快速排序的方法进行排序。



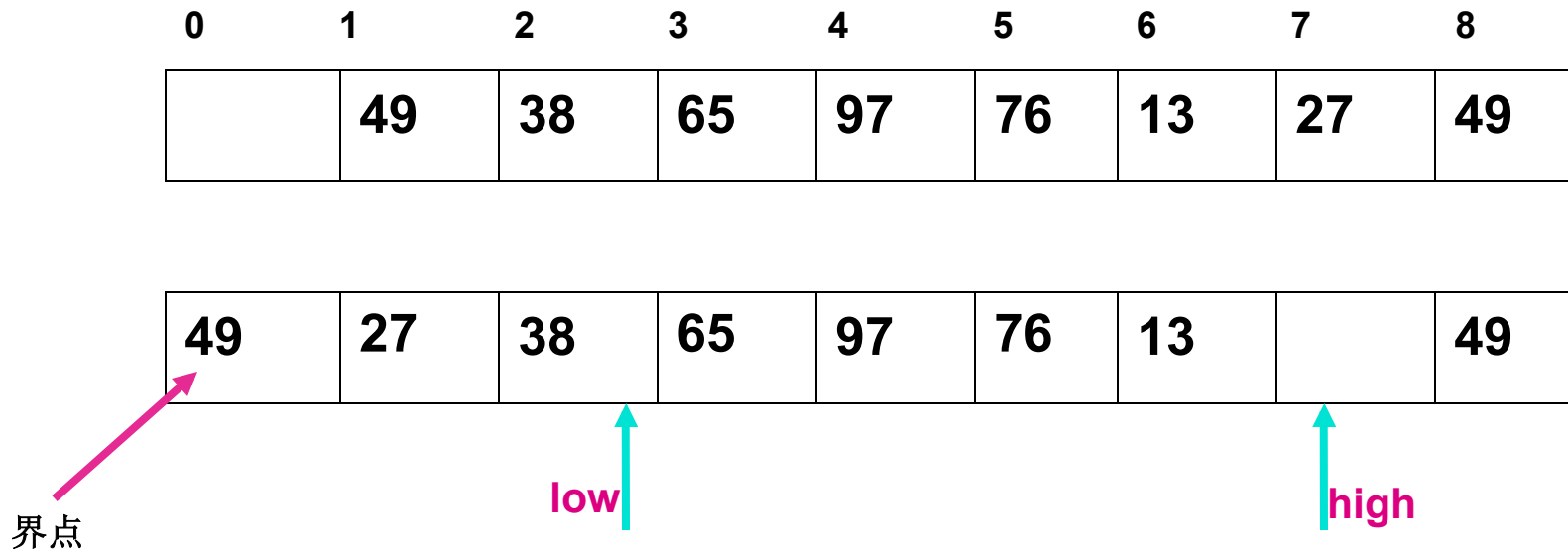


## 6、快速排序

### · 快速排序

- 原理: 若序列中有  $n$  个元素, 任选一个界点的关键字作为界点, 将序列分成两部分。其中左半部分的结点的关键字小于等于界点, 右半部分的结点的关键字大于等于界点。然后, 对左右两部分分别进行类似的处理, 直至排好序为止。

**e.g:** 将序列 **49、38、65、97、76、13、27、49** 用 快速排序的方法进行排序。

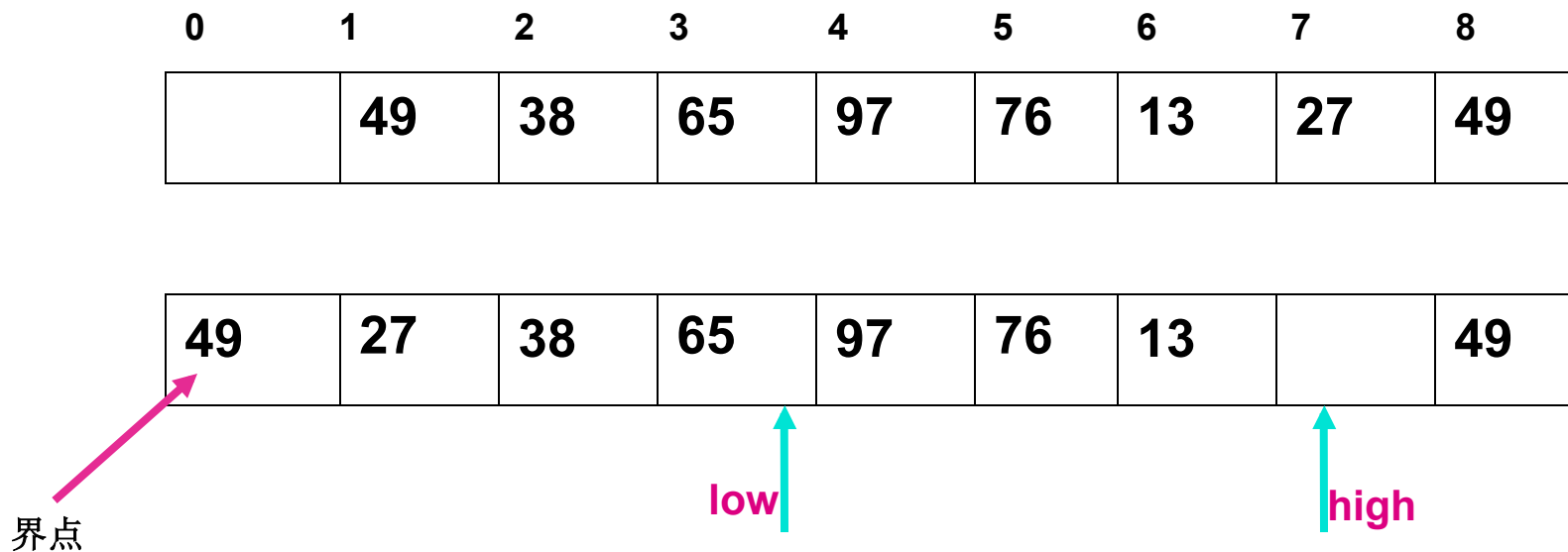


## 6、快速排序

### · 快速排序

- 原理: 若序列中有  $n$  个元素, 任选一个关键字作为界点, 将序列分成两部分。其中左半部分的结点的关键字小于等于界点, 右半部分的结点的关键字大于等于界点。然后, 对左右两部分分别进行类似的处理, 直至排好序为止。

e.g: 将序列 **49、38、65、97、76、13、27、49** 用 快速排序的方法进行排序。

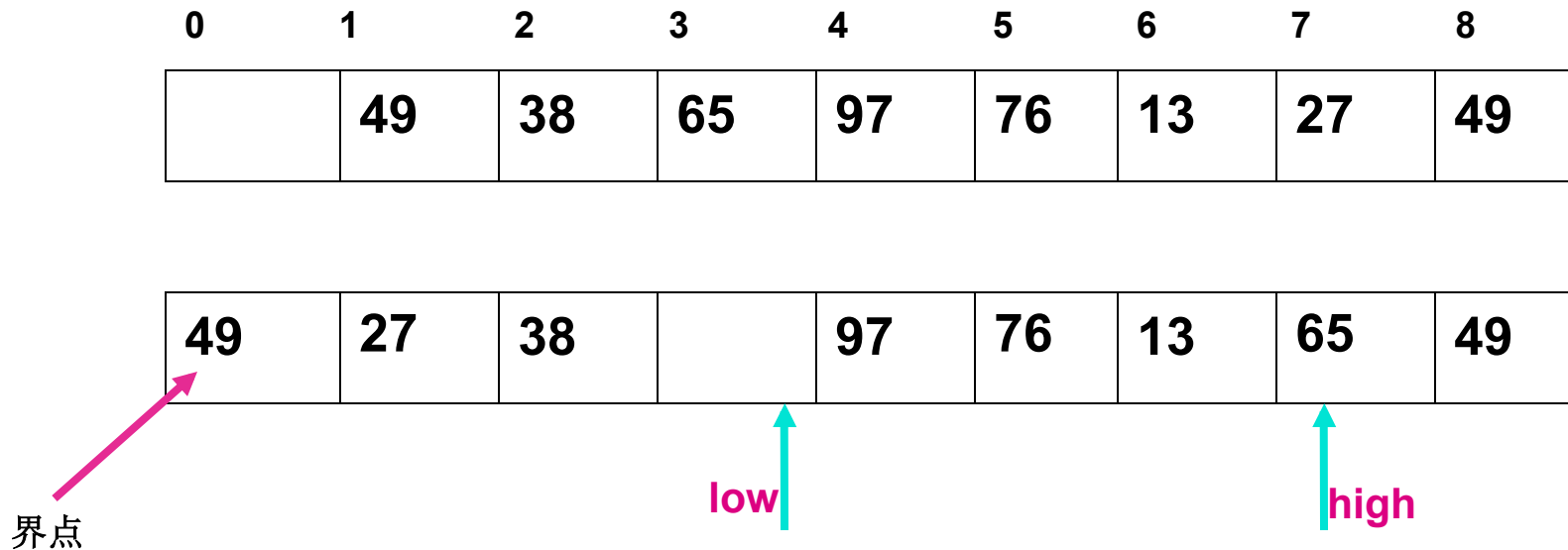


## 6、快速排序

### · 快速排序

- 原理: 若序列中有  $n$  个元素, 任选一个界点的关键字作为界点, 将序列分成两部分。其中左半部分的结点的关键字小于等于界点, 右半部分的结点的关键字大于等于界点。然后, 对左右两部分分别进行类似的处理, 直至排好序为止。

**e.g:** 将序列 **49、38、65、97、76、13、27、49** 用 快速排序的方法进行排序。

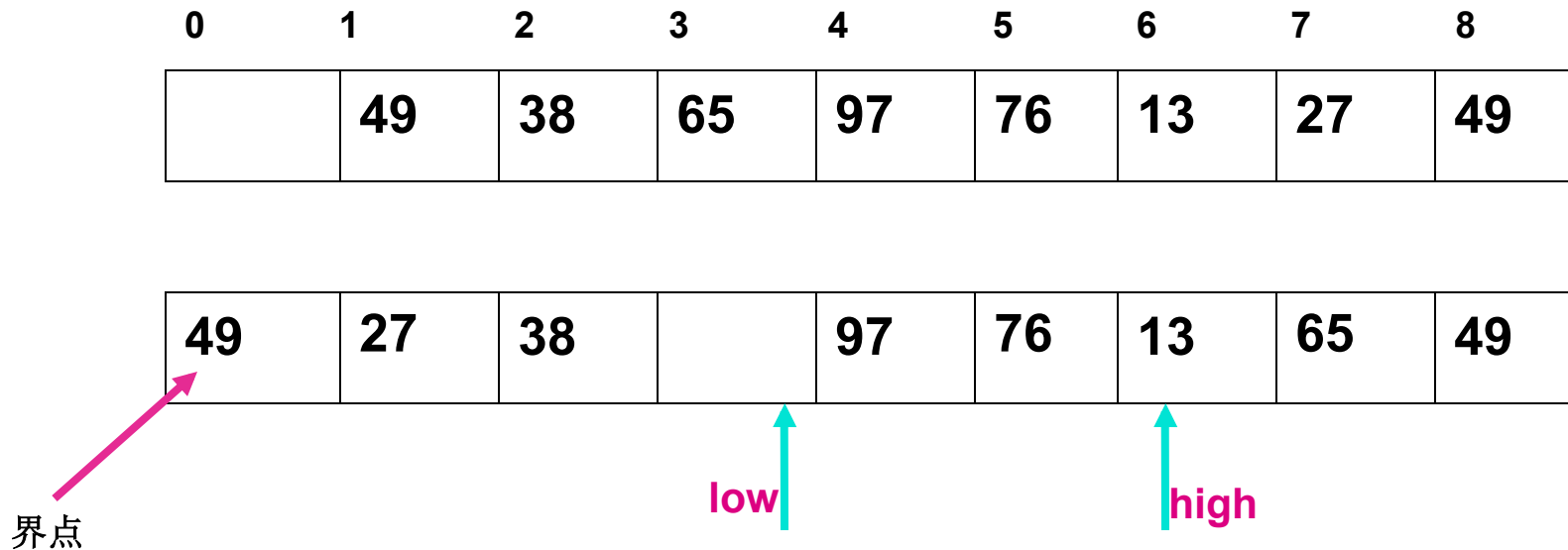


## 6、快速排序

### · 快速排序

- 原理: 若序列中有  $n$  个元素, 任选一个关键字作为界点, 将序列分成两部分。其中左半部分的结点的关键字小于等于界点, 右半部分的结点的关键字大于等于界点。然后, 对左右两部分分别进行类似的处理, 直至排好序为止。

e.g: 将序列 **49、38、65、97、76、13、27、49** 用 快速排序的方法进行排序。

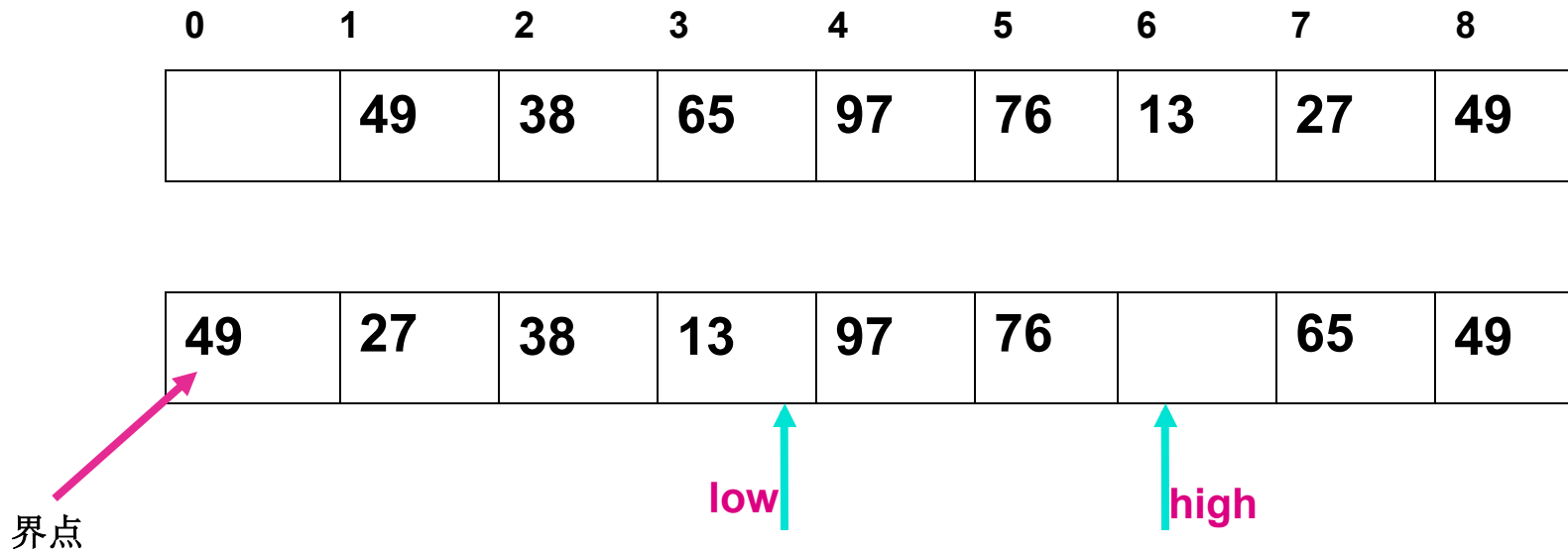


## 6、快速排序

### · 快速排序

- 原理: 若序列中有  $n$  个元素, 任选一个界点的关键字作为界点, 将序列分成两部分。其中左半部分的结点的关键字小于等于界点, 右半部分的结点的关键字大于等于界点。然后, 对左右两部分分别进行类似的处理, 直至排好序为止。

**e.g:** 将序列 **49、38、65、97、76、13、27、49** 用 快速排序的方法进行排序。

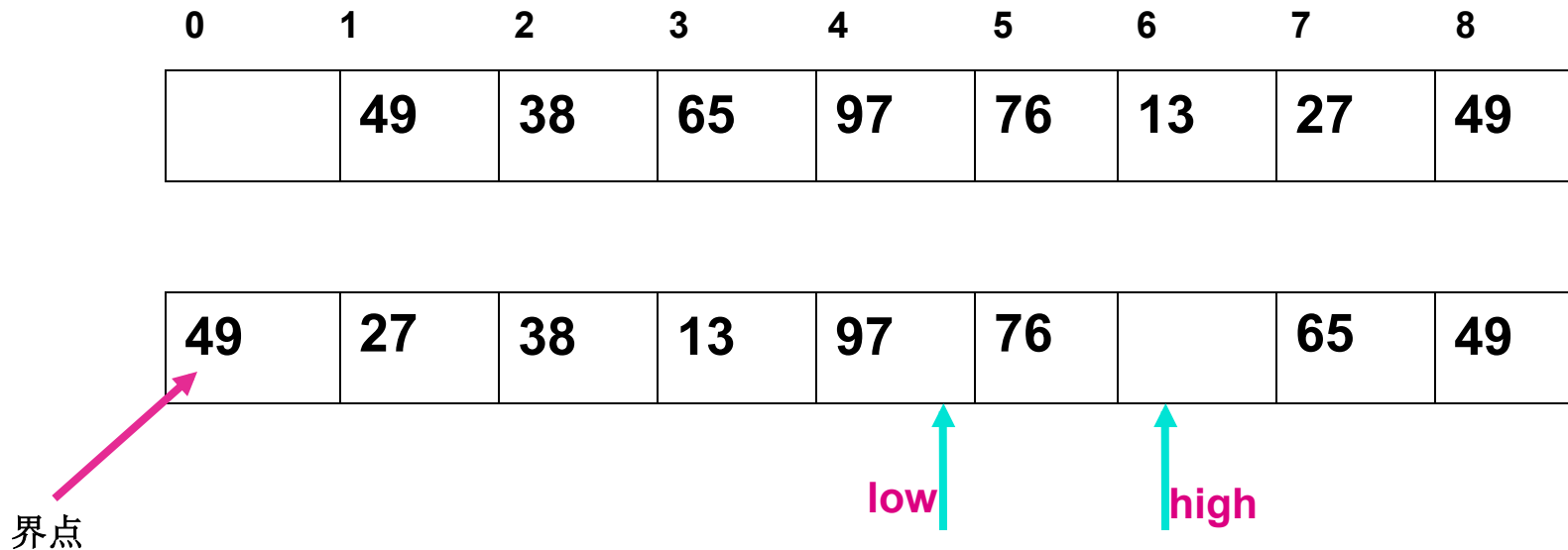


## 6、快速排序

### · 快速排序

- 原理: 若序列中有  $n$  个元素, 任选一个关键字作为界点, 将序列分成两部分。其中左半部分的结点的关键字小于等于界点, 右半部分的结点的关键字大于等于界点。然后, 对左右两部分分别进行类似的处理, 直至排好序为止。

**e.g:** 将序列 **49、38、65、97、76、13、27、49** 用 快速排序的方法进行排序。

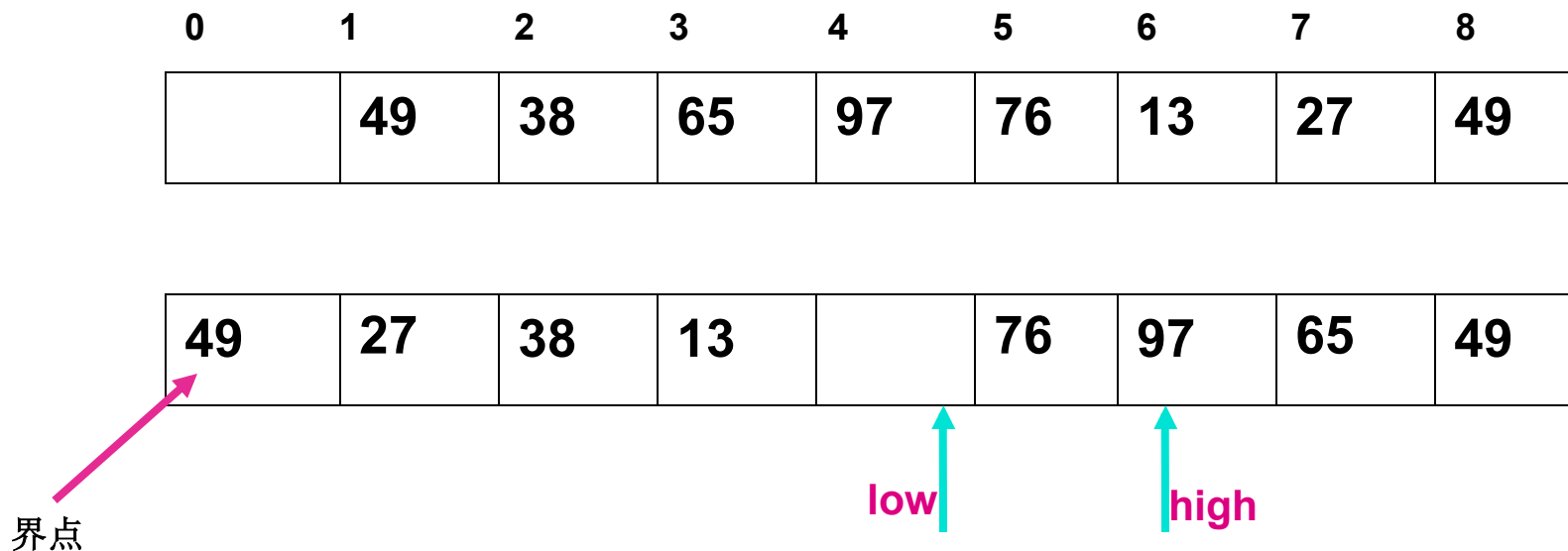


## 6、快速排序

### · 快速排序

- 原理: 若序列中有  $n$  个元素, 任选一个界点的关键字作为界点, 将序列分成两部分。其中左半部分的结点的关键字小于等于界点, 右半部分的结点的关键字大于等于界点。然后, 对左右两部分分别进行类似的处理, 直至排好序为止。

**e.g:** 将序列 **49、38、65、97、76、13、27、49** 用 快速排序的方法进行排序。

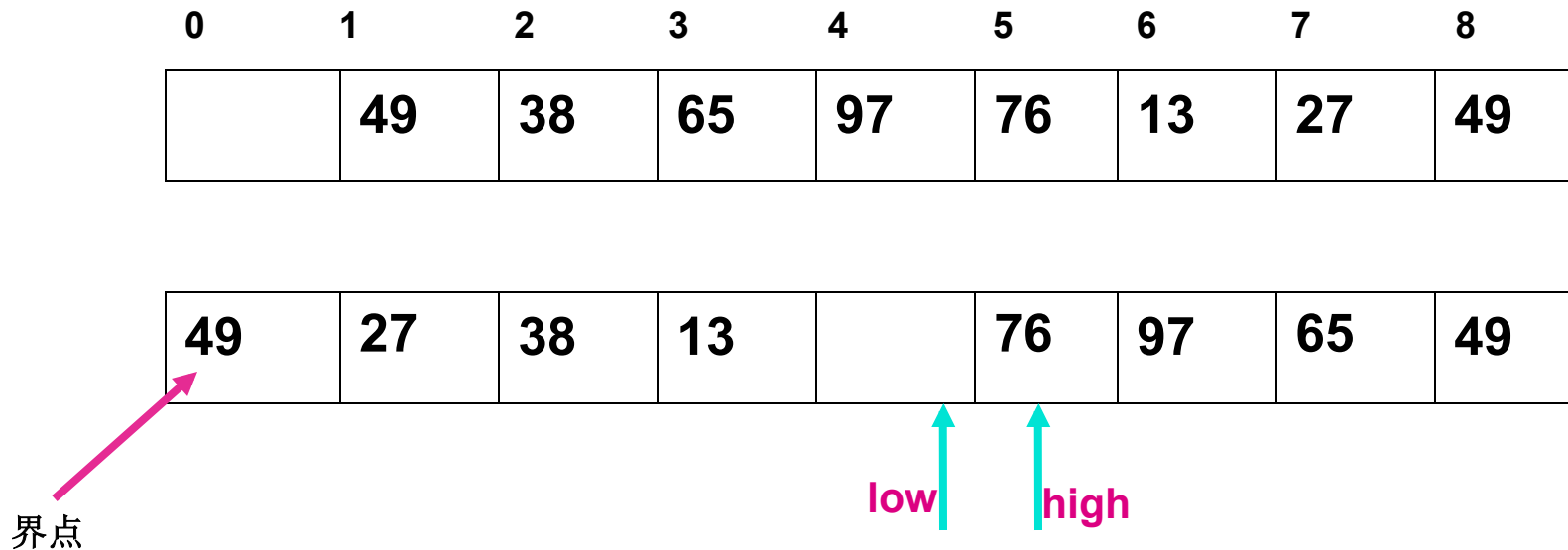


### 3、快速排序

#### · 快速排序

- 原理: 若序列中有  $n$  个元素, 任选一个关键字作为界点, 将序列分成两部分。其中左半部分的结点的关键字小于等于界点, 右半部分的结点的关键字大于等于界点。然后, 对左右两部分分别进行类似的处理, 直至排好序为止。

e.g: 将序列 **49、38、65、97、76、13、27、49** 用 快速排序的方法进行排序。





### 3、快速排序

#### · 快速排序

- 原理: 若序列中有  $n$  个元素, 任选一个界点的关键字作为界点, 将序列分成两部分。其中左半部分的结点的关键字小于等于界点, 右半部分的结点的关键字大于等于界点。然后, 对左右两部分分别进行类似的处理, 直至排好序为止。

e.g: 将序列 49、38、65、97、76、13、27、49 用 快速排序的方法进行排序。

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

## 6、快速排序

### · 快速排序

- 原理: 若序列中有  $n$  个元素, 任选一个关键字作为界点, 将序列分成两部分。其中左半部分的结点的关键字小于等于界点, 右半部分的结点的关键字大于等于界点。然后, 对左右两部分分别进行类似的处理, 直至排好序为止。

**e.g:** 将序列 **49、38、65、97、76、13、27、49** 用 快速排序的方法进行排序。

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

low high

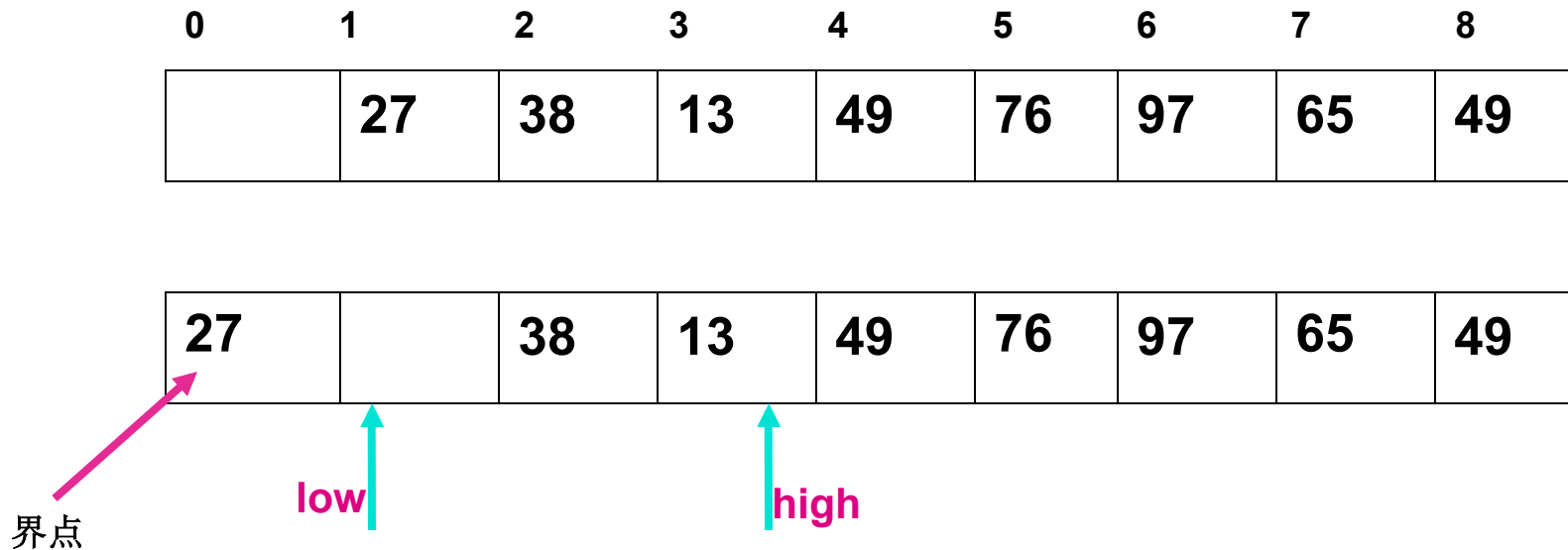
界点

## 6、快速排序

### · 快速排序

- 原理: 若序列中有  $n$  个元素, 任选一个界点的关键字作为界点, 将序列分成两部分。其中左半部分的结点的关键字小于等于界点, 右半部分的结点的关键字大于等于界点。然后, 对左右两部分分别进行类似的处理, 直至排好序为止。

**e.g:** 将序列 **49、38、65、97、76、13、27、49** 用 快速排序的方法进行排序。

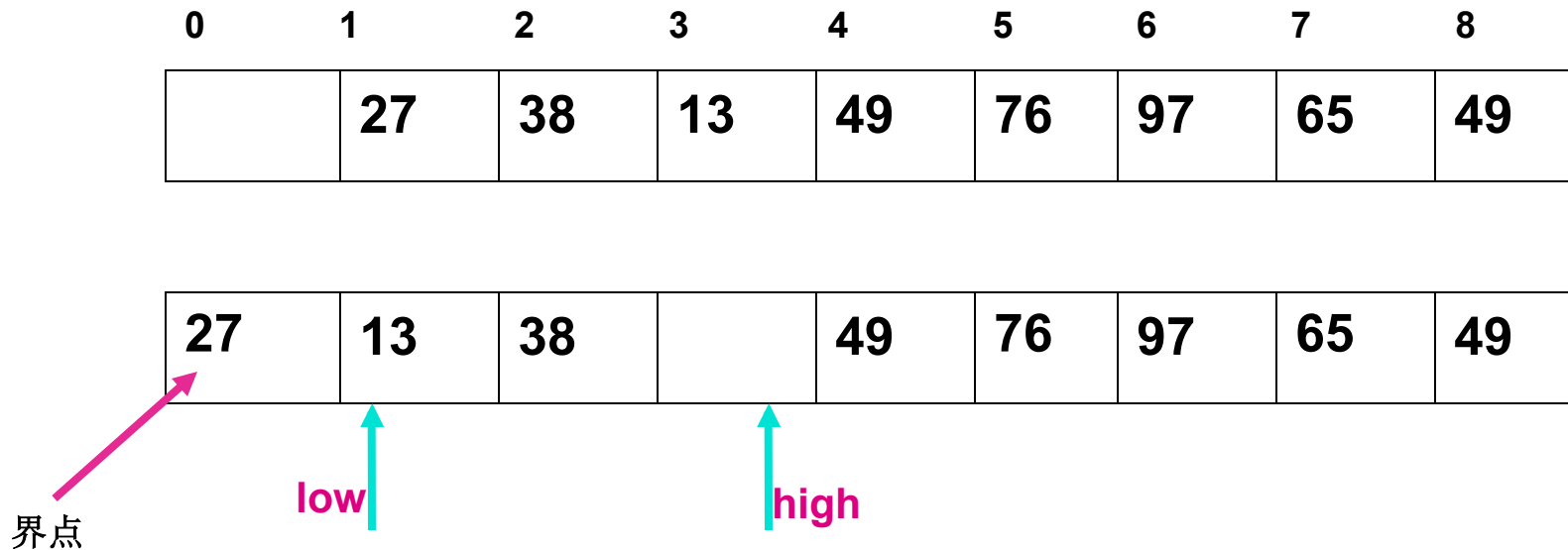


### 3、快速排序

#### · 快速排序

- 原理: 若序列中有  $n$  个元素, 任选一个关键字作为界点, 将序列分成两部分。其中左半部分的结点的关键字小于等于界点, 右半部分的结点的关键字大于等于界点。然后, 对左右两部分分别进行类似的处理, 直至排好序为止。

e.g: 将序列 **49、38、65、97、76、13、27、49** 用 快速排序的方法进行排序。

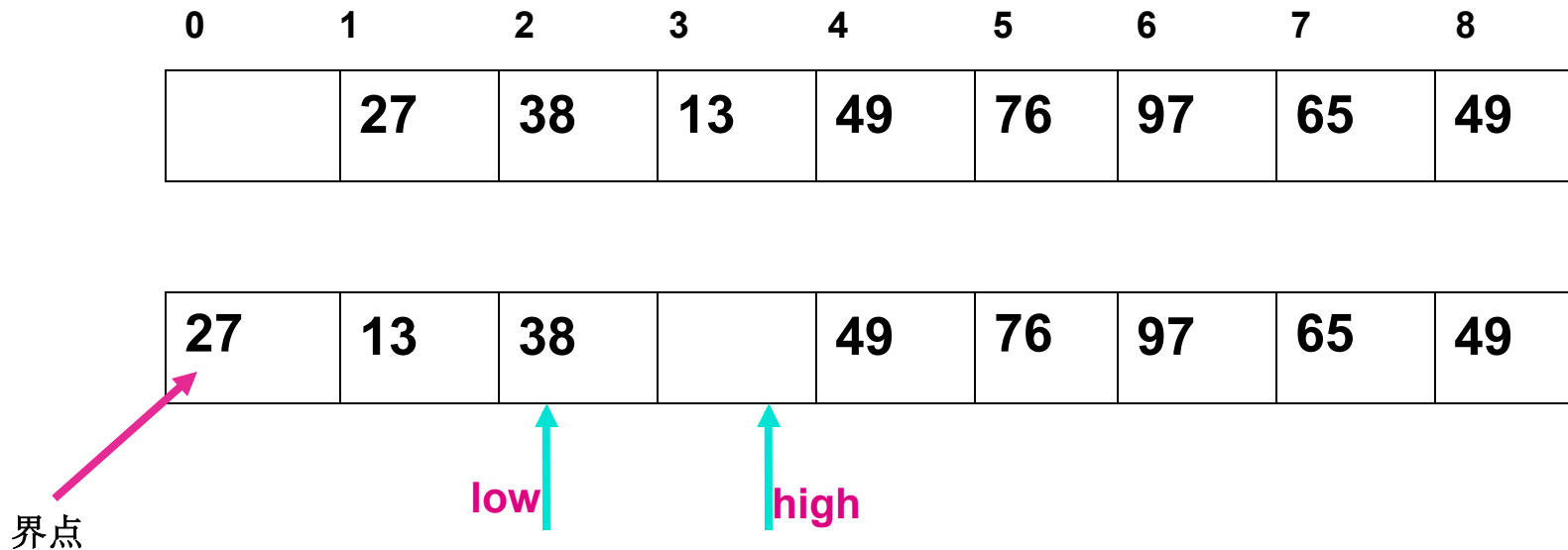


## 6、快速排序

### · 快速排序

- 原理: 若序列中有  $n$  个元素, 任选一个界点的关键字作为界点, 将序列分成两部分。其中左半部分的结点的关键字小于等于界点, 右半部分的结点的关键字大于等于界点。然后, 对左右两部分分别进行类似的处理, 直至排好序为止。

e.g: 将序列 **49、38、65、97、76、13、27、49** 用 快速排序的方法进行排序。

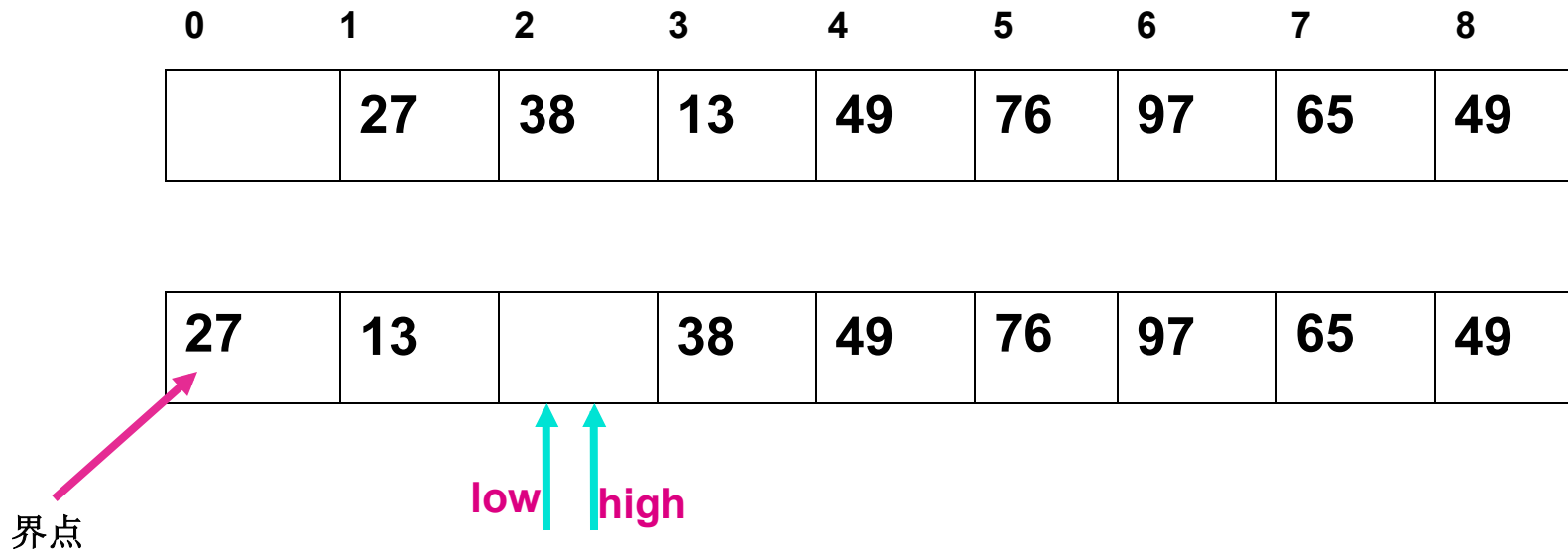


## 6、快速排序

### · 快速排序

- 原理: 若序列中有  $n$  个元素, 任选一个关键字作为界点, 将序列分成两部分。其中左半部分的结点的关键字小于等于界点, 右半部分的结点的关键字大于等于界点。然后, 对左右两部分分别进行类似的处理, 直至排好序为止。

**e.g:** 将序列 **49、38、65、97、76、13、27、49** 用 快速排序的方法进行排序。

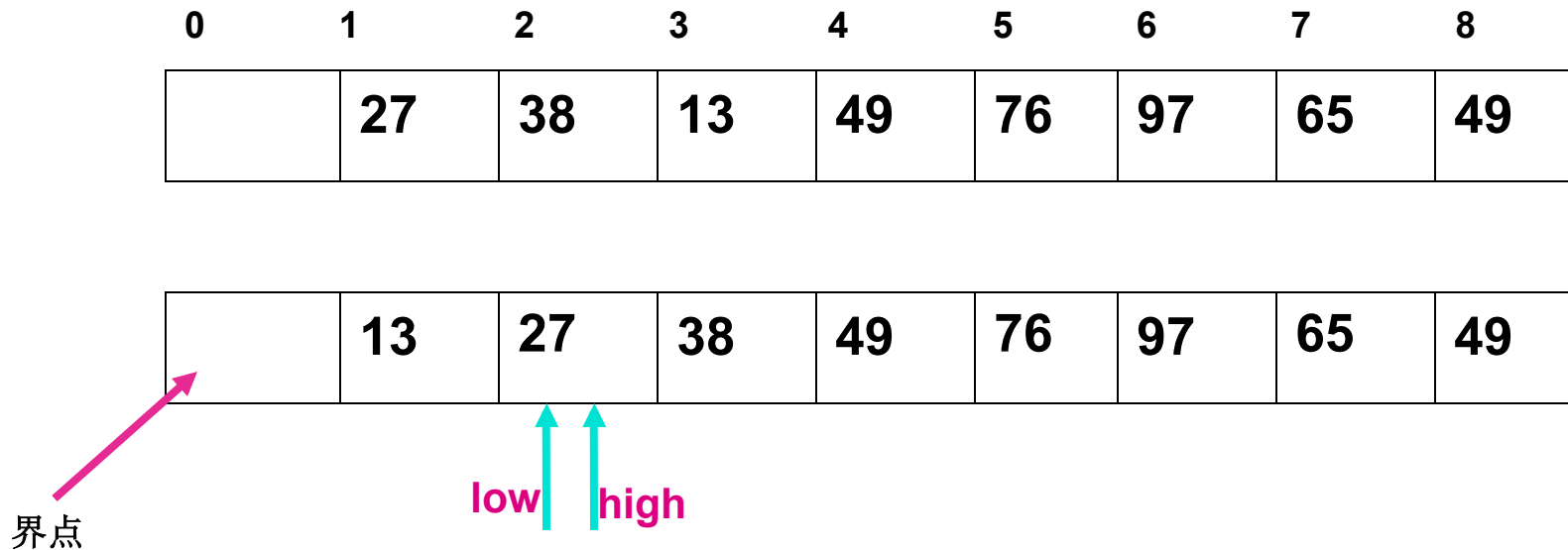


## 6、快速排序

### · 快速排序

- 原理: 若序列中有  $n$  个元素, 任选一个界点的关键字作为界点, 将序列分成两部分。其中左半部分的结点的关键字小于等于界点, 右半部分的结点的关键字大于等于界点。然后, 对左右两部分分别进行类似的处理, 直至排好序为止。

**e.g:** 将序列 **49、38、65、97、76、13、27、49** 用 快速排序的方法进行排序。

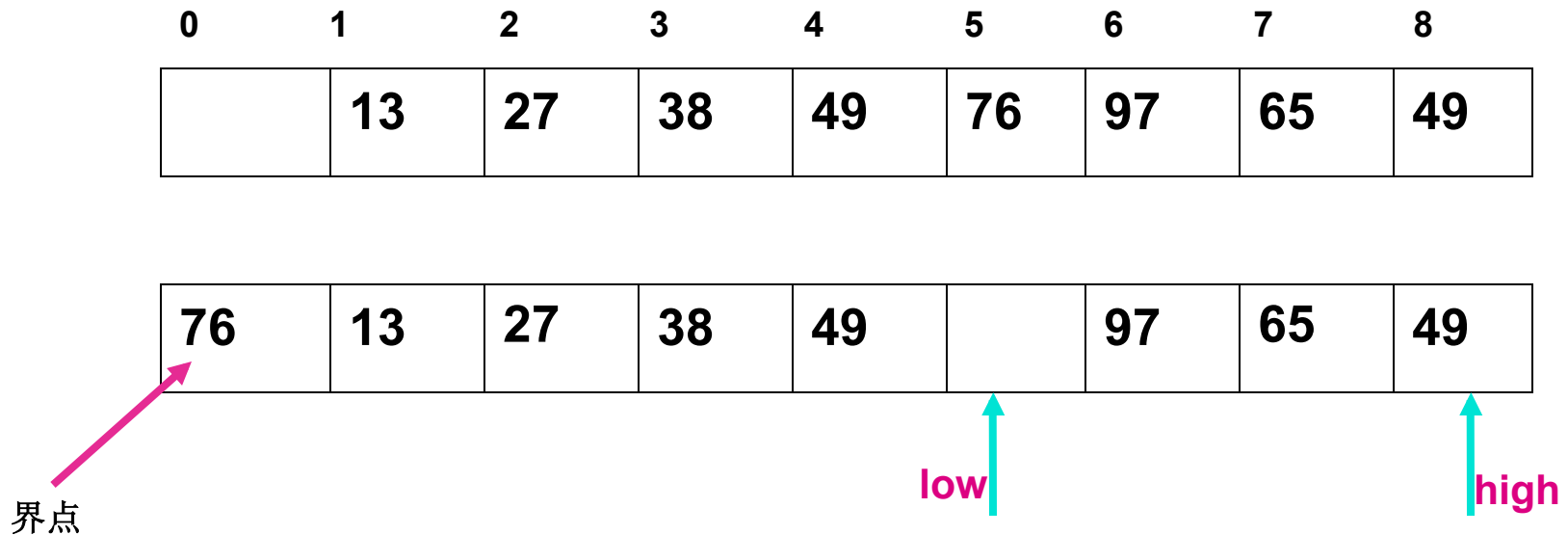


## 6、快速排序

### · 快速排序

- 原理: 若序列中有  $n$  个元素, 任选一个关键字作为界点, 将序列分成两部分。其中左半部分的结点的关键字小于等于界点, 右半部分的结点的关键字大于等于界点。然后, 对左右两部分分别进行类似的处理, 直至排好序为止。

e.g: 将序列 **49、38、65、97、76、13、27、49** 用 快速排序的方法进行排序。





## 6、快速排序

### · 快速排序

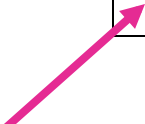
- 原理: 若序列中有  $n$  个元素, 任选一个界点的关键字作为界点, 将序列分成两部分。其中左半部分的结点的关键字小于等于界点, 右半部分的结点的关键字大于等于界点。然后, 对左右两部分分别进行类似的处理, 直至排好序为止。

**e.g:** 将序列 **49、38、65、97、76、13、27、49** 用 快速排序的方法进行排序。

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

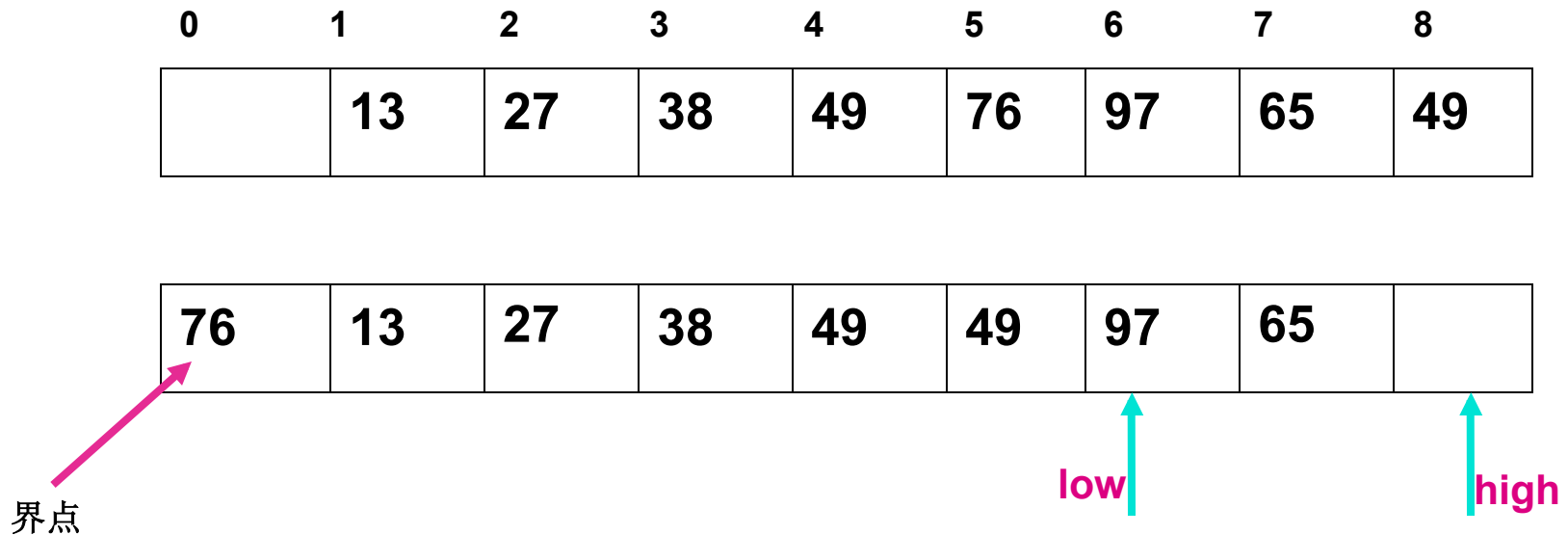


## 6、快速排序

### · 快速排序

- 原理: 若序列中有  $n$  个元素, 任选一个关键字作为界点, 将序列分成两部分。其中左半部分的结点的关键字小于等于界点, 右半部分的结点的关键字大于等于界点。然后, 对左右两部分分别进行类似的处理, 直至排好序为止。

**e.g:** 将序列 **49、38、65、97、76、13、27、49** 用 快速排序的方法进行排序。

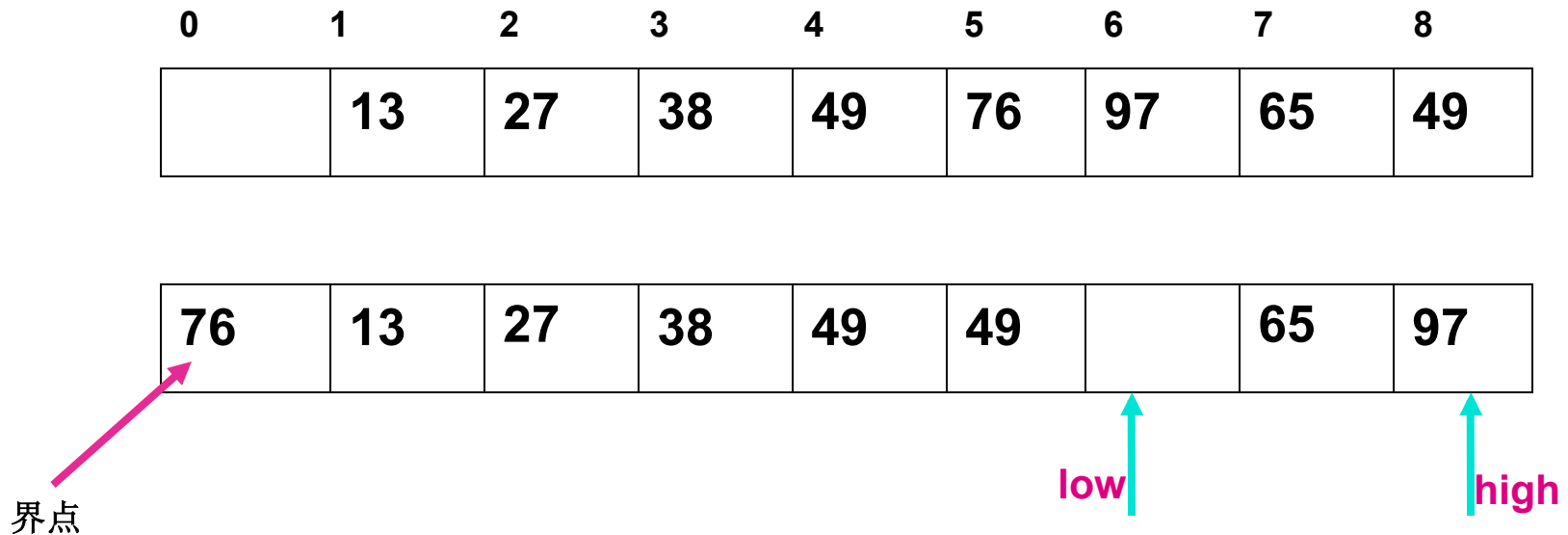


## 6、快速排序

### · 快速排序

- 原理: 若序列中有  $n$  个元素, 任选一个界点的关键字作为界点, 将序列分成两部分。其中左半部分的结点的关键字小于等于界点, 右半部分的结点的关键字大于等于界点。然后, 对左右两部分分别进行类似的处理, 直至排好序为止。

**e.g:** 将序列 **49、38、65、97、76、13、27、49** 用 快速排序的方法进行排序。



## 6、快速排序

### · 快速排序

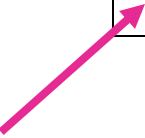
- 原理: 若序列中有  $n$  个元素, 任选一个关键字作为界点, 将序列分成两部分。其中左半部分的结点的关键字小于等于界点, 右半部分的结点的关键字大于等于界点。然后, 对左右两部分分别进行类似的处理, 直至排好序为止。

e.g: 将序列 **49、38、65、97、76、13、27、49** 用 快速排序的方法进行排序。

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



## 6、快速排序

### · 快速排序

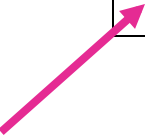
- 原理: 若序列中有  $n$  个元素, 任选一个界点的关键字作为界点, 将序列分成两部分。其中左半部分的结点的关键字小于等于界点, 右半部分的结点的关键字大于等于界点。然后, 对左右两部分分别进行类似的处理, 直至排好序为止。

**e.g:** 将序列 **49、38、65、97、76、13、27、49** 用 快速排序的方法进行排序。

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



## 6、快速排序

### · 快速排序

- 原理: 若序列中有  $n$  个元素, 任选一个关键字作为界点, 将序列分成两部分。其中左半部分的结点的关键字小于等于界点, 右半部分的结点的关键字大于等于界点。然后, 对左右两部分分别进行类似的处理, 直至排好序为止。

**e.g:** 将序列 **49、38、65、97、76、13、27、49** 用 快速排序的方法进行排序。

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

## 6、快速排序

### · 快速排序

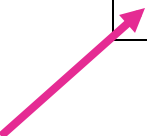
- 原理: 若序列中有  $n$  个元素, 任选一个界点的关键字作为界点, 将序列分成两部分。其中左半部分的结点的关键字小于等于界点, 右半部分的结点的关键字大于等于界点。然后, 对左右两部分分别进行类似的处理, 直至排好序为止。

**e.g:** 将序列 **49、38、65、97、76、13、27、49** 用 快速排序的方法进行排序。

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

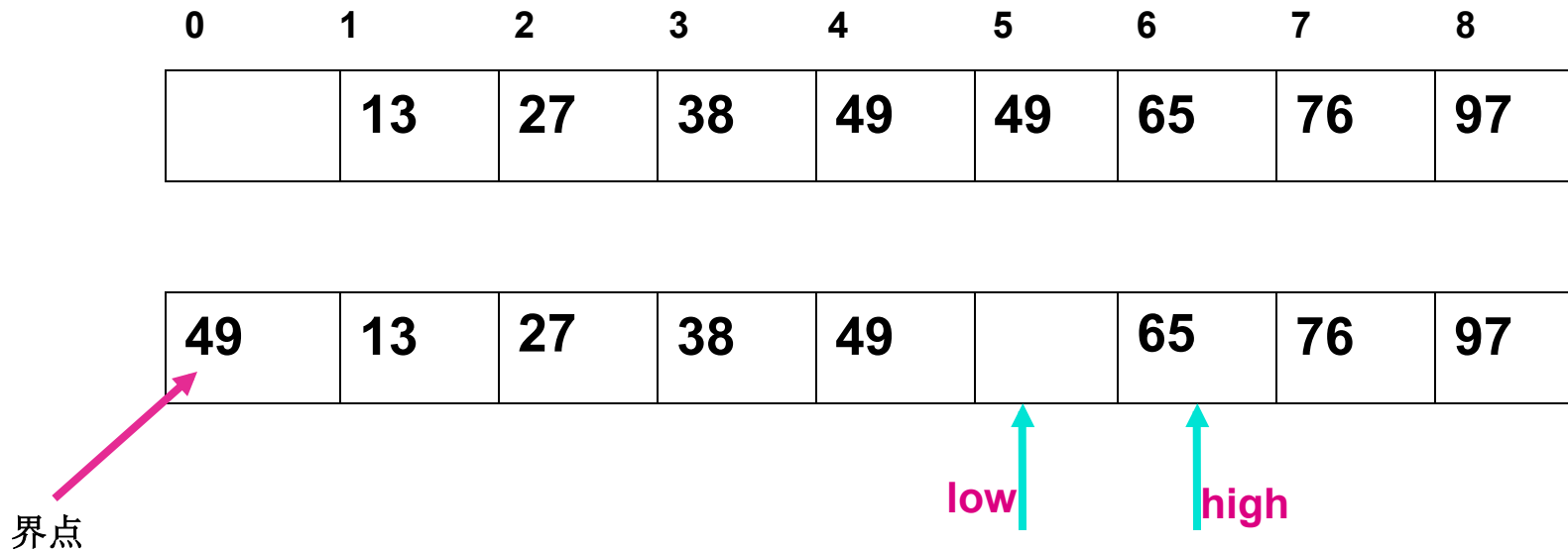


## 6、快速排序

### · 快速排序

- 原理: 若序列中有  $n$  个元素, 任选一个关键字作为界点, 将序列分成两部分。其中左半部分的结点的关键字小于等于界点, 右半部分的结点的关键字大于等于界点。然后, 对左右两部分分别进行类似的处理, 直至排好序为止。

e.g: 将序列 **49、38、65、97、76、13、27、49** 用 快速排序的方法进行排序。



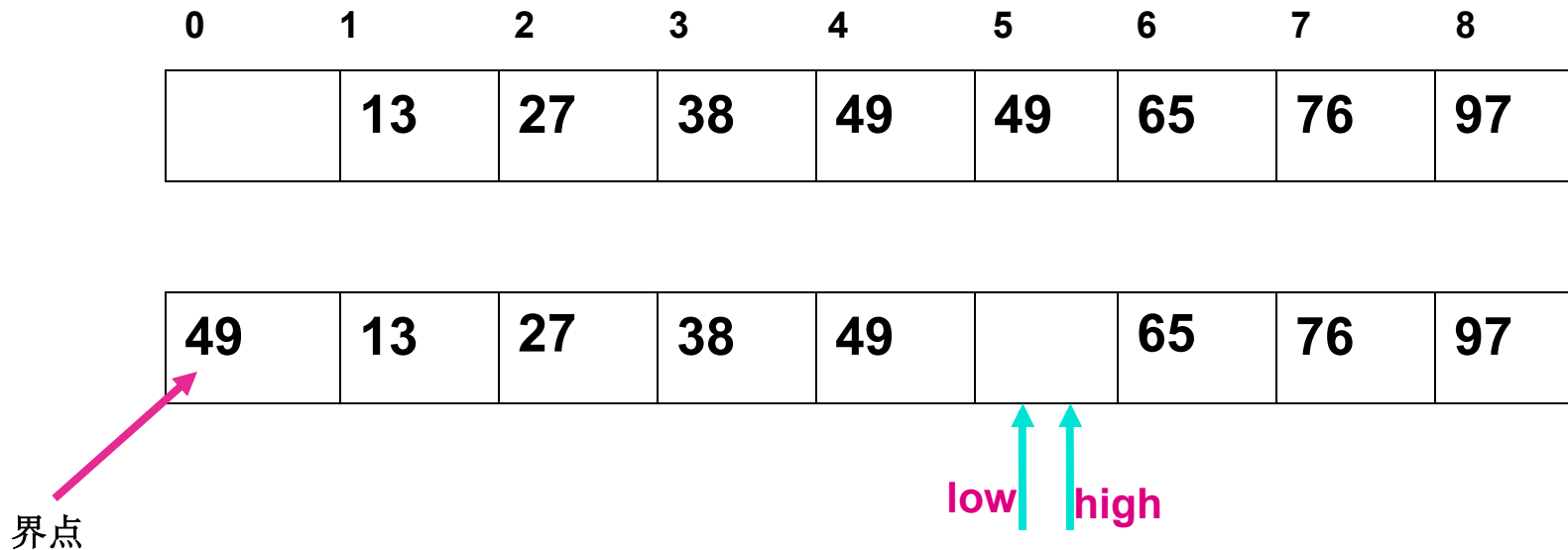


## 6、快速排序

### · 快速排序

- 原理: 若序列中有  $n$  个元素, 任选一个界点的关键字作为界点, 将序列分成两部分。其中左半部分的结点的关键字小于等于界点, 右半部分的结点的关键字大于等于界点。然后, 对左右两部分分别进行类似的处理, 直至排好序为止。

**e.g:** 将序列 **49、38、65、97、76、13、27、49** 用 快速排序的方法进行排序。

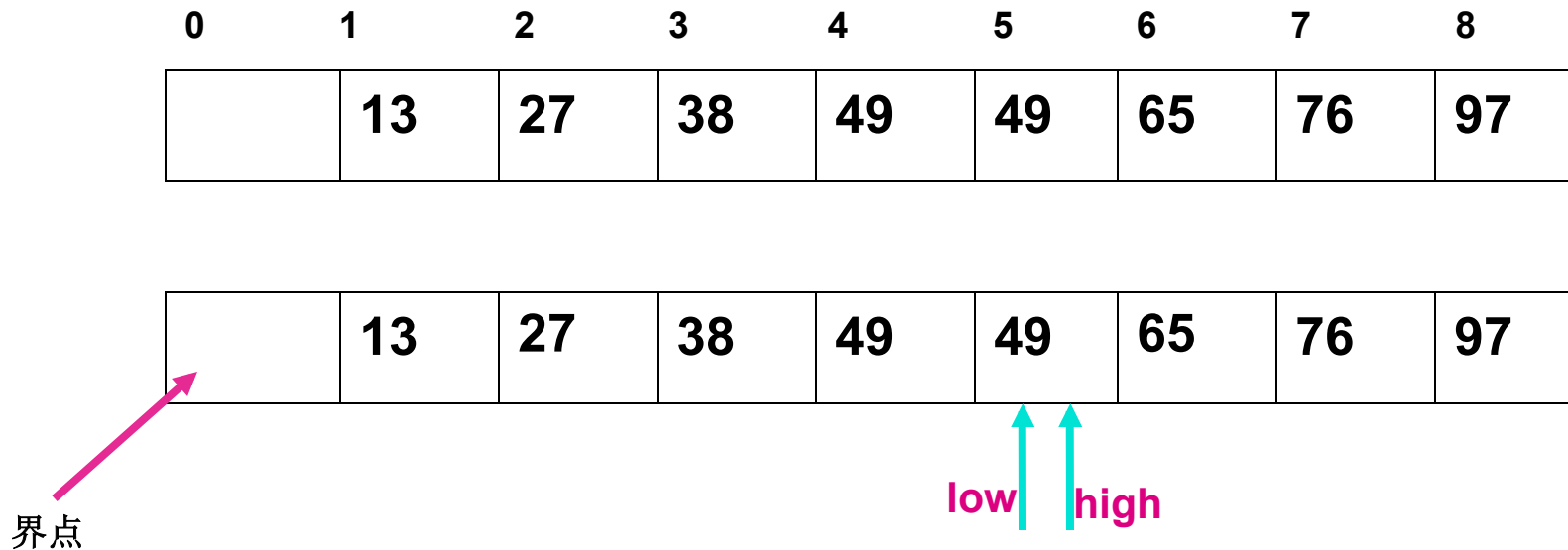


## 6、快速排序

### · 快速排序

- 原理: 若序列中有  $n$  个元素, 任选一个关键字作为界点, 将序列分成两部分。其中左半部分的结点的关键字小于等于界点, 右半部分的结点的关键字大于等于界点。然后, 对左右两部分分别进行类似的处理, 直至排好序为止。

**e.g:** 将序列 **49、38、65、97、76、13、27、49** 用 快速排序的方法进行排序。



## 6、快速排序

### · 快速排序

- 时间复杂性分析: 考虑平均情况下的时间复杂性。设元素个数为  $n$ , 则所有的排列形式共有  $n!$  种。每一个元素都可以充当界点, 所以充当界点的情况共计有  $n$  种。故平均时间复杂性  $T$  应为:

$$T_{(n)} = cn + \sum_{k=1}^n [T_{(k-1)} + T_{(n-k)}] / n$$

$$\text{设: } T_{(0)} = T_{(1)} = b$$

$$T_{(n)} = cn + 2 \sum_{i=0}^{n-1} T_{(i)} / n$$

$$nT_{(n)} = cn^2 + 2 \sum_{i=0}^{n-1} T_{(i)}$$

$$(n-1)T_{(n-1)} = c(n-1)^2 + 2 \sum_{i=0}^{n-2} T_{(i)}$$

上二式相减的结果为:

$$nT_{(n)} - (n-1)T_{(n-1)} = c(2n-1) + 2T_{(n-1)}$$

$$nT_{(n)} = c(2n-1) + (n+1)T_{(n-1)}$$

## 6、快速排序

### · 快速排序

所以：

$$T_{(n)} / (n + 1) = c(2n-1) / (n(n+1)) + T_{(n-1)} / n$$

$$T_{(n)} / (n + 1) \leq 2c/(n+1) + T_{(n-1)} / n$$

$$T_{(n-1)} / n \leq 2c/n + T_{(n-2)} / (n-1)$$

⋮

$$T_{(n-1)} / 3 \leq 2c/3 + T_{(1)} / 2$$

于是：

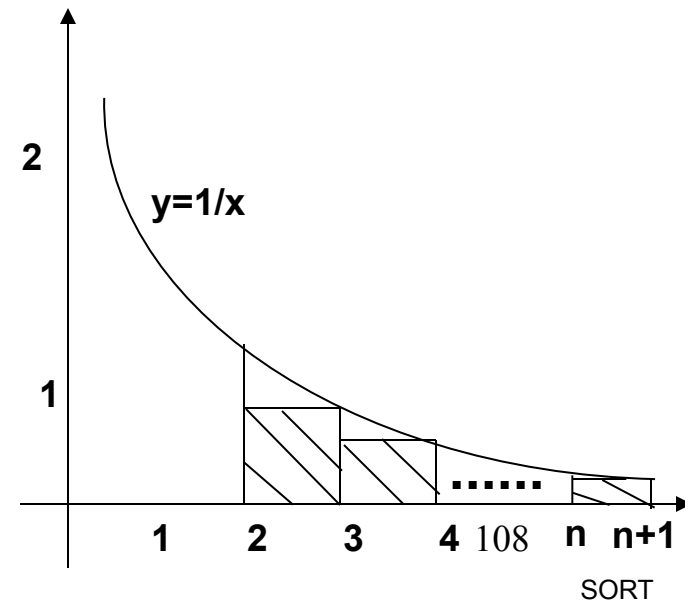
$$T_{(n)} \leq 2c(n+1)\ln(n+1) + b(n+1) / 2$$

注意：这里使用了一个公式：

$$\sum_{k=3}^{n+1} 1/k \leq \int_2^{n+1} 1/x \, dx < \ln(n+1)$$

结论：快速排序在平均情况下的时间复杂性为

**$O(n \log n)$**  级或阶的，通常认为是在平均情况下最佳的排序方法。



## 6、快速排序

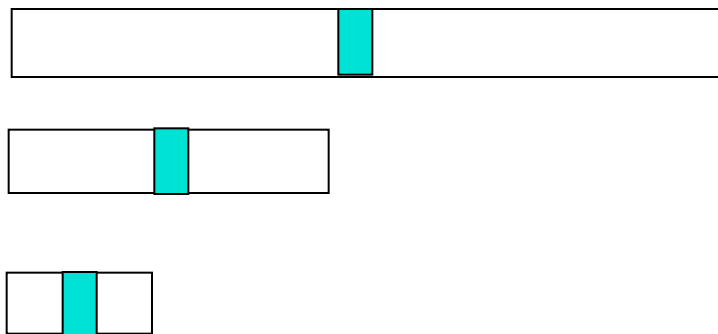
- 快速排序

- 快速排序使用的额外空间

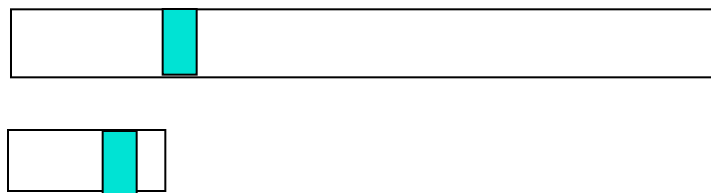
考虑最坏情况，栈的空间为  $O(n)$ 。可以考虑降低到  $O(\log n)$ 。

观察以下情况，每次优先处理短的那一段。则空间的使用可以降低到最小。

均匀分段



非均匀分段



在均匀分段的情况下，系统每一次将其中一段的上界、下界的下标压入堆栈，而去处理另外一段。当处理的一段元素个数为 **1** 时，将不必保存另一段的上、下界。这样堆栈的总的层数为  $\log n$ 。在非均匀分段时，由于优先处理短的一段，下降到只有 **1** 个元素的段的速度将更快，所以不会超过  $\log n$ 。

## 6、快速排序

- 快速排序

- 快速排序的不足和克服方法

- 1、在序列差不多排好序时，采用直接插入排序、起泡排序等排序方法。序列的个数通常取为 **10** 左右。本书使用了插入排序法用于差不多排好序时。
- 2、将递归改成非递归算法。避免进出栈、恢复断点等工作。速度加快。

## 6、快速排序

### · 快速排序

#### · 快速排序的不足和克服方法

3、最坏情况下 时间复杂性为  $O(n^2)$  级。如：在序列已是正序的情况下。

10、20、30、40、50、60、70、80 界点 10，共进行 7 次比较。

10、20、30、40、50、60、70、80 界点 20，共进行 6 次比较。

10、20、30、40、50、60、70、80 界点 60，共进行 2 次比较。

10、20、30、40、50、60、70、80 界点 70，共进行 1 次比较。

在最坏情况下：总的比较次数为： $(n-1) + (n-2) + \dots + 2 + 1 = n^2/2 = O(n^2)$

或

$$T_{(n)} = \begin{cases} 1 & \text{当 } n = 1 \text{ 时} \\ T_{(n-1)} + (n-1) & \text{当 } n = 1 \text{ 时} \end{cases}$$

原因：界点选择不当。改进：随机选取界点或最左、最右、中间三个元素中的值处于中间的作为界点，通常可以避免最坏情况。

## 7、基数排序

### 1、基数排序：

- 多关键字排序技术：

多关键字：（ $K_1, K_2, \dots, K_t$ ）；例如：关键字  $K_1$  小的结点排在前面。如关键字  $K_1$  相同，则比较关键字  $K_2$ ，关键字  $K_2$  小的结点排在前面，依次类推.....

基数排序实例：假定给定的是  $t = 2$  位十进制数，存放在数组  $B$  之中。现要求通过基数排序法将其排序。

方法：设置 十个口袋，因十进制数分别有数字：0, 1, 2, ..., 9，分别用  $B_0, B_1, B_2, \dots, B_9$  进行标识。

执行  $j = 1..t$  (这里  $t = 2$ ) 次循环，每次进行一次分配动作，一次收集动作。

分配：将右起第  $j$  位数字相同的数放入同一口袋。比如数字为 1 者，则放入口袋  $B_1$ ，余类推 .....

收集：按  $B_0, B_1, B_2, \dots, B_9$  的顺序进行收集。

e.g:  $B = 5, 2, 9, 7, 18, 17, 52$  用基数技术进行排序。



## 6、快速排序

- 快速排序
  - 程序实现:

```

const int insertnum = 10;
template < class EType>
void Swap( EType & x, EType & y ) { EType temp = x; x = y; y = temp; }
template < class EType>
void SimpleInsertSort( EType a[ ], int Size ) {
    int j, q; // a[1], a[2 ], ...,a[Size]为待排序的数组。a[0] 为哨兵单元。
    EType temp=a[0];
    for ( j = 2; j <= Size; j++ )
        { a[0] = a[j];
          for ( q = j; a[0] < a[q-1]; q- -)      a[q] =a[q-1];
          a[q] = a[0];
        }
    a[0] = temp; // 恢复a[0]原来的值，用于快速排序法中。
}
template < class EType>
void QuickSort( EType arr[ ], int n )      {
    // a[1], a[2 ], ...,a[n]为待排序的数组。a[0]不用。
    QuickSort( arr, 1, n);
}

```


## 7、基数排序

### 1、基数排序：

e.g: B = 5、2、9、7、18、17、52 用基数排序法进行排序。

B = 5、2、9、7、18、17、52



根据  所指向的数，进行分配动作，将其分配到相应的口袋。

口袋

$B_0$

$B_1$

$B_2$

$B_3$

$B_4$

$B_5$

$B_6$

$B_7$

$B_8$

$B_9$


## 7、基数排序

### 1、基数排序：

e.g: B = 5 、 2、 9、 7、 18、 17、 52 用基数排序法进行排序。

B = 5、 2、 9、 7、 18、 17、 52



根据  所指向的数，进行分配动作，将其分配到相应的口袋。

口袋

$B_0$

$B_1$

$B_2$

$B_3$

$B_4$

$B_5$      5

$B_6$

$B_7$

$B_8$

$B_9$

## 7、基数排序

### 1、基数排序：

e.g: B = 5 、 2、 9、 7、 18、 17、 52 用基数排序法进行排序。

B = 5、 2、 9、 7、 18、 17、 52



口袋

$B_0$

$B_1$

$B_2$

$B_3$

$B_4$


$B_5$      5

$B_6$

$B_7$

$B_8$

$B_9$

根据  所  
指向的  
数，进行分配动作，  
将其分配到相应的口  
袋。

## 7、基数排序

### 1、基数排序：

e.g: B = 5 、 2、 9、 7、 18、 17、 52 用基数排序法进行排序。

B = 5、 2、 9、 7、 18、 17、 52



口袋

$B_0$

$B_1$

$B_2$      2

$B_3$

$B_4$


$B_5$      5

$B_6$

$B_7$

$B_8$

$B_9$

根据  所  
指向的  
数，进行分配动作，  
将其分配到相应的口  
袋。

## 7、基数排序

### 1、基数排序：

e.g: B = 5 、 2、 9、 7、 18、 17、 52 用基数排序法进行排序。

B = 5、 2、 9、 7、 18、 17、 52



口袋

$B_0$

$B_1$

$B_2$      2

$B_3$

$B_4$


$B_5$      5

$B_6$

$B_7$

$B_8$

$B_9$

根据  所指向的数，进行分配动作，将其分配到相应的口袋。

## 7、基数排序

### 1、基数排序：

e.g: B = 5 、 2、 9、 7、 18、 17、 52 用基数排序法进行排序。

B = 5、 2、 9、 7、 18、 17、 52



口袋

$B_0$

$B_1$

$B_2$      2

$B_3$

$B_4$


$B_5$      5

$B_6$

$B_7$

$B_8$

$B_9$      9

根据  所指向的数，进行分配动作，将其分配到相应的口袋。

## 7、基数排序

### 1、基数排序：

e.g: B = 5 、 2、 9、 7、 18、 17、 52 用基数排序法进行排序。

B = 5、 2、 9、 7、 18、 17、 52



口袋

$B_0$

$B_1$

$B_2$      2

$B_3$

$B_4$


$B_5$      5

$B_6$

$B_7$

$B_8$

$B_9$      9

根据  所指向的数，进行分配动作，将其分配到相应的口袋。



## 7、基数排序

### 1、基数排序：

e.g: B = 5、2、9、7、18、17、52 用基数排序法进行排序。

B = 5、2、9、7、18、17、52



口袋

$B_0$

$B_1$

$B_2$      2

$B_3$

$B_4$


$B_5$      5

$B_6$

$B_7$      7

$B_8$

$B_9$      9

根据  所指向的数，进行分配动作，将其分配到相应的口袋。

## 7、基数排序

### 1、基数排序：

e.g: B = 5 、 2、 9、 7、 18、 17、 52 用基数排序法进行排序。

B = 5、 2、 9、 7、 18、 17、 52



口袋

$B_0$

$B_1$

$B_2$      2

$B_3$

$B_4$


$B_5$      5

$B_6$

$B_7$      7

$B_8$

$B_9$      9

根据  所指向的数，进行分配动作，将其分配到相应的口袋。

## 7、基数排序

### 1、基数排序：

e.g: B = 5 、 2、 9、 7、 18、 17、 52 用基数排序法进行排序。

B = 5、 2、 9、 7、 18、 17、 52



口袋

$B_0$

$B_1$

$B_2$      2

$B_3$

$B_4$


$B_5$      5

$B_6$

$B_7$      7

$B_8$      18

$B_9$      9

根据  所指向的数，进行分配动作，将其分配到相应的口袋。

## 7、基数排序

### 1、基数排序：

e.g: B = 5 、 2、 9、 7、 18、 17、 52 用基数排序法进行排序。

B = 5、 2、 9、 7、 18、 17、 52



口袋

$B_0$

$B_1$

$B_2$      2

$B_3$

$B_4$


$B_5$      5

$B_6$

$B_7$      7

$B_8$      18

$B_9$      9

根据  所  
指向的  
数，进行分配动作，  
将其分配到相应的口  
袋。

## 7、基数排序

### 1、基数排序：

e.g: B = 5、2、9、7、18、17、52 用基数排序法进行排序。

B = 5、2、9、7、18、17、52



口袋

$B_0$

$B_1$

$B_2$      2

$B_3$

$B_4$


$B_5$      5

$B_6$

$B_7$      7     17

$B_8$      18

$B_9$      9

根据  所指向的数，进行分配动作，将其分配到相应的口袋。


## 7、基数排序

### 1、基数排序：

e.g: B = 5、2、9、7、18、17、52 用基数排序法进行排序。

B = 5、2、9、7、18、17、52



根据  所指向的数，进行分配动作，将其分配到相应的口袋。

口袋

$B_0$

$B_1$

$B_2$      2

$B_3$

$B_4$

$B_5$      5

$B_6$

$B_7$      7     17

$B_8$      18

$B_9$      9


## 7、基数排序

### 1、基数排序：

e.g: B = 5、2、9、7、18、17、52 用基数排序法进行排序。

B = 5、2、9、7、18、17、52



根据  所  
指向的  
数，进行分配动作，  
将其分配到相应的口  
袋。

口袋

$B_0$

$B_1$

$B_2$      2     52

$B_3$

$B_4$

$B_5$      5

$B_6$

$B_7$      7     17

$B_8$      18

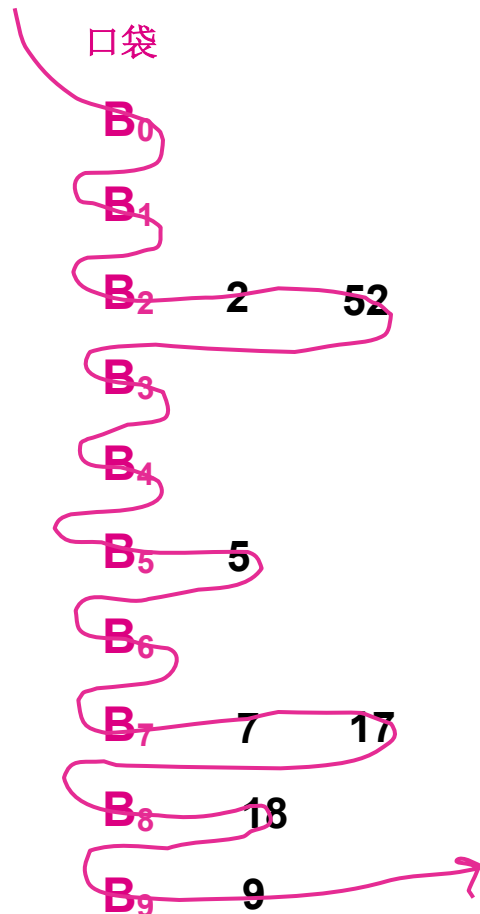
$B_9$      9

## 7、基数排序

### 1、基数排序：

e.g: B = 5、2、9、7、18、17、52 用基数排序法进行排序。

B = 5、2、9、7、18、17、52



分配完毕，按照红色箭头所指的方向进行收集动作。注意：收集后的序列已经按照右起第一位（个位数字）排好序了。

收集后的序列：2、52、5、7、17、18、9、



## 7、基数排序

### 1、基数排序：

e.g: B = 5 、 2、 9、 7、 18、 17、 52 用基数排序法进行排序。

B = 2、 52、 5、 7、 17、 18、 9 （第一次收集的结果）



口袋

$B_0$

$B_1$

$B_2$

$B_3$

$B_4$


$B_5$

$B_6$

$B_7$

$B_8$

$B_9$

根据  所指向的数，进行分配动作，将其分配到相应的口袋。和第一次不同，这次根据右起第二位数字（十位数字）进行分配。

## 7、基数排序

### 1、基数排序：

e.g: B = 5 、 2、 9、 7、 18、 17、 52 用基数排序法进行排序。

B = 2、 52、 5、 7、 17、 18、 9 （第一次收集的结果）



口袋

$B_0$  2

$B_1$

$B_2$

$B_3$

$B_4$


$B_5$

$B_6$

$B_7$

$B_8$

$B_9$

根据  所指向的数，进行分配动作，将其分配到相应的口袋。和第一次不同，这次根据右起第二位数字（十位数字）进行分配。

## 7、基数排序

### 1、基数排序：

e.g: B = 5 、 2、 9、 7、 18、 17、 52 用基数排序法进行排序。

B = 2、 52、 5、 7、 17、 18、 9 （第一次收集的结果）



口袋

$B_0$  2

$B_1$

$B_2$

$B_3$

$B_4$


$B_5$

$B_6$

$B_7$

$B_8$

$B_9$

根据  所  
指向的  
数，进行分配动作，  
将其分配到相应的口  
袋。和第一次不同，  
这次根据右起第二位  
数字（十位数字）进  
分配。


## 7、基数排序

### 1、基数排序：

e.g: B = 5 、 2、 9、 7、 18、 17、 52 用基数排序法进行排序。

B = 2、 52、 5、 7、 17、 18、 9 （第一次收集的结果）



根据  所指向的数，进行分配动作，将其分配到相应的口袋。和第一次不同，这次根据右起第二位数字（十位数字）进行分配。

口袋

$B_0$	2
$B_1$	
$B_2$	
$B_3$	
$B_4$	
$B_5$	52
$B_6$	
$B_7$	
$B_8$	
$B_9$	

## 7、基数排序


### 1、基数排序：

e.g: B = 5 、 2、 9、 7、 18、 17、 52 用基数排序法进行排序。

B = 2、 52、 5、 7、 17、 18、 9 （第一次收集的结果）



口袋

根据  所指向的数，进行分配动作，将其分配到相应的口袋。和第一次不同，这次根据右起第二位数字（十位数字）进行分配。

$B_0$       2  
 $B_1$   
 $B_2$   
 $B_3$   
 $B_4$   
 $B_5$       52  
 $B_6$   
 $B_7$   
 $B_8$   
 $B_9$

## 7、基数排序


### 1、基数排序：

e.g: B = 5 、 2、 9、 7、 18、 17、 52 用基数排序法进行排序。

B = 2、 52、 5、 7、 17、 18、 9 （第一次收集的结果）



口袋

根据  所指向的数，进行分配动作，将其分配到相应的口袋。和第一次不同，这次根据右起第二位数字（十位数字）进行分配。

$B_0$       2      5

$B_1$

$B_2$

$B_3$

$B_4$

$B_5$       52

$B_6$

$B_7$

$B_8$

$B_9$

## 7、基数排序


### 1、基数排序：

e.g: B = 5 、 2、 9、 7、 18、 17、 52 用基数排序法进行排序。

B = 2、 52、 5、 7、 17、 18、 9 （第一次收集的结果）



口袋

根据  所指向的数，进行分配动作，将其分配到相应的口袋。和第一次不同，这次根据右起第二位数字（十位数字）进行分配。

$B_0$       2      5

$B_1$

$B_2$

$B_3$

$B_4$

$B_5$       52

$B_6$

$B_7$

$B_8$

$B_9$

## 7、基数排序


### 1、基数排序：

e.g: B = 5 、 2、 9、 7、 18、 17、 52 用基数排序法进行排序。

B = 2、 52、 5、 7、 17、 18、 9 （第一次收集的结果）



口袋

根据  所指向的数，进行分配动作，将其分配到相应的口袋。和第一次不同，这次根据右起第二位数字（十位数字）进行分配。

**B<sub>0</sub>**      2      5      7

**B<sub>1</sub>**

**B<sub>2</sub>**

**B<sub>3</sub>**

**B<sub>4</sub>**

**B<sub>5</sub>**      52

**B<sub>6</sub>**

**B<sub>7</sub>**

**B<sub>8</sub>**

**B<sub>9</sub>**



## 7、基数排序

### 1、基数排序：

e.g: B = 5 、 2、 9、 7、 18、 17、 52 用基数排序法进行排序。

B = 2、 52、 5、 7、 17、 18、 9 （第一次收集的结果）



口袋

$B_0$      2     5     7

$B_1$

$B_2$

$B_3$

$B_4$


$B_5$      52

$B_6$

$B_7$

$B_8$

$B_9$

根据  所指向的数，进行分配动作，将其分配到相应的口袋。和第一次不同，这次根据右起第二位数字（十位数字）进行分配。

## 7、基数排序

### 1、基数排序：

e.g: B = 5 、 2、 9、 7、 18、 17、 52 用基数排序法进行排序。

B = 2、 52、 5、 7、 17、 18、 9 （第一次收集的结果）



口袋

$B_0$      2     5     7

$B_1$      17

$B_2$

$B_3$

$B_4$


$B_5$      52

$B_6$

$B_7$

$B_8$

$B_9$

根据  所指向的数，进行分配动作，将其分配到相应的口袋。和第一次不同，这次根据右起第二位数字（十位数字）进行分配。

## 7、基数排序

### 1、基数排序：

e.g: B = 5 、 2、 9、 7、 18、 17、 52 用基数排序法进行排序。

B = 2、 52、 5、 7、 17、 18、 9 （第一次收集的结果）



口袋

**B<sub>0</sub>**     2     5     7

**B<sub>1</sub>**     17

**B<sub>2</sub>**

**B<sub>3</sub>**

**B<sub>4</sub>**


**B<sub>5</sub>**     52

**B<sub>6</sub>**

**B<sub>7</sub>**

**B<sub>8</sub>**

**B<sub>9</sub>**

根据  所指向的数，进行分配动作，将其分配到相应的口袋。和第一次不同，这次根据右起第二位数字（十位数字）进行分配。

## 7、基数排序

### 1、基数排序：

e.g: B = 5 、 2、 9、 7、 18、 17、 52 用基数排序法进行排序。

B = 2、 52、 5、 7、 17、 18、 9 （第一次收集的结果）



口袋

$B_0$      2    5    7

$B_1$      17 18

$B_2$

$B_3$

$B_4$


$B_5$      52

$B_6$

$B_7$

$B_8$

$B_9$

根据  所指向的数，进行分配动作，将其分配到相应的口袋。和第一次不同，这次根据右起第二位数字（十位数字）进行分配。

## 7、基数排序

### 1、基数排序：

e.g: B = 5 、 2、 9、 7、 18、 17、 52 用基数排序法进行排序。

B = 2、 52、 5、 7、 17、 18、 9 （第一次收集的结果）



口袋

$B_0$     2    5    7

$B_1$     17   18

$B_2$

$B_3$

$B_4$


$B_5$     52

$B_6$

$B_7$

$B_8$

$B_9$

根据  所指向的数，进行分配动作，将其分配到相应的口袋。和第一次不同，这次根据右起第二位数字（十位数字）进行分配。

## 7、基数排序

### 1、基数排序：

e.g: B = 5 、 2、 9、 7、 18、 17、 52 用基数排序法进行排序。

B = 2、 52、 5、 7、 17、 18、 9 （第一次收集的结果）



口袋

$B_0$     2    5    7    9

$B_1$     17   18

$B_2$

$B_3$

$B_4$


$B_5$     52

$B_6$

$B_7$

$B_8$

$B_9$

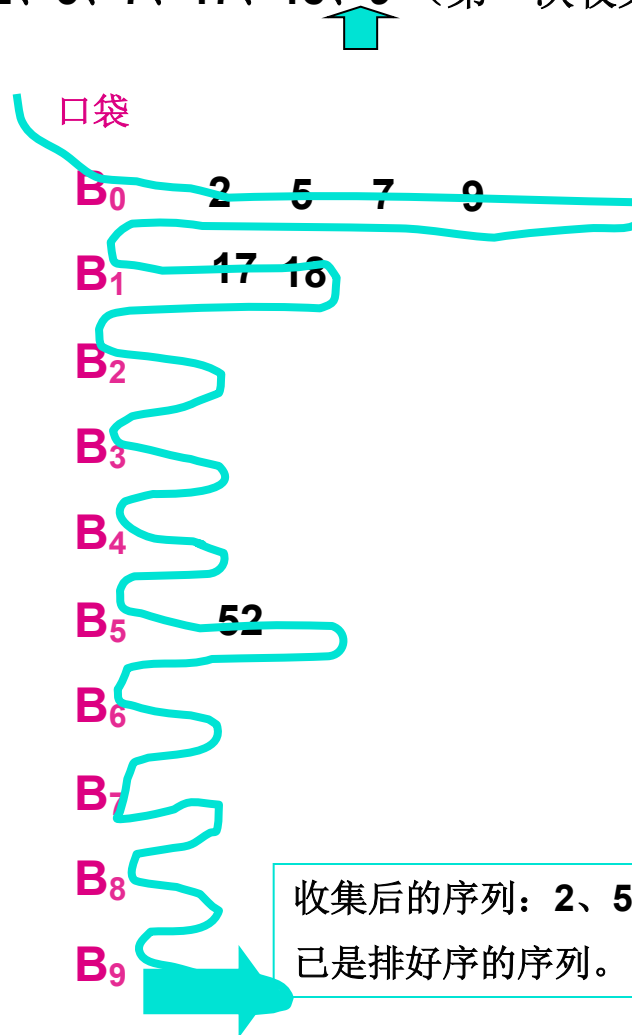
根据  所指向的数，进行分配动作，将其分配到相应的口袋。和第一次不同，这次根据右起第二位数字（十位数字）进行分配。

## 7、基数排序

### 1、基数排序：

e.g: B = 5 、 2、 9、 7、 18、 17、 52 用基数排序法进行排序。

B = 2、 52、 5、 7、 17、 18、 9 （第一次收集的结果）



分配完毕，按照红色箭头所指的方向进行第二次收集动作。注意：收集后的序列已经按照右起第一位（个位数字）、右起第二位（十位数字）排好序了。

收集后的序列：2、5、7、9、17、18、52  
已是排好序的序列。

## 7、基数排序

### 2、时空分析：

- 空间：采用顺序分配，显然不合适。由于每个口袋都有可能存放所有的待排序的整数。所以，额外空间的需求为  $10n$ ，太大了。采用链接分配是合理的。额外空间的需求为  $n$ ，通常再增加指向每个口袋的首尾指针就可以了。

在一般情况下，设每个键字的取值范围为  $d$ ，首尾指针共计  $2 \times \text{radix}$  个，总的空间为  $O(n + 2 \times \text{radix})$ 。

- 时间：上例中每个数计有  $t = 2$  位，因此执行  $t = 2$  次分配和收集就可以了。在一般情况下，每个结点有  $d$  位关键字，必须执行  $t = d$  次分配和收集操作。

每次分配的代价：  $O(n)$

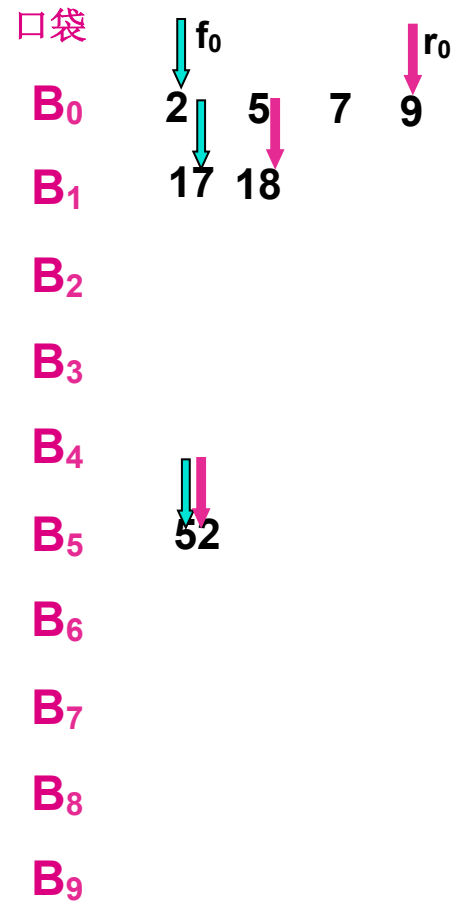
每次收集的代价：  $O(\text{radix})$

总的代价为：  $O(d \times (n + \text{radix}))$



## 7、基数排序

- 首尾指针的使用:



## 7、基数排序

- 程序实现:

```
const int radix = 10; // 如：十进制整数的基为 10。
```

```
const int t = 5;      // 整数的最大位数。
```

```
typedef struct node { int data ;
```

```
                int next; // 给出下一个结点的下标地址。
```

```
        } node;
```

```
void BucketSort( node arr[ ], int max ) {
```

```
// arr[1], arr[2 ], ..., arr[max]为待排序的整数数组，a[0]用作头结点。
```

```
for ( int j=0; j < max; j++ ) arr[j].next = j + 1; arr[max].next = 0; // 生成静态链表。
```

```
RadixSort( arr, max);
```

```
}
```

## 7、基数排序

- 程序实现:

```

void RadixSort( node a[ ], int max )      {
    int front[radix], tail[radix]; // 口袋的首尾指针。
    int p, last, j, k, d = 1;
    for ( j=1; j<= t; j++) {
        for ( k = 0; k < 10; ++k ) { front[ k ]=0; tail[ k ]=0; } // 口袋的首尾指针置初值。
        p = a[0].next;                // 头结点a[0] 给出链中的第一个整数结点的下标地址。
        while ( p ) {                  // 分配过程。
            k = a[p].data / d % radix; // 取出右起第j 位数字，将结点放入口袋Bk。
            if ( ! tail[k] ) front[ k ] = p;
            else a[ tail[k] ].next = p;
            tail[k] = p; p = a[p].next;
        }
        last=0;
        for ( k = 0; k < radix; ++k)      { // 收集过程，从口袋B0到最后一个口袋。
            if ( tail[k] ) { a[last].next = front[k];    last = tail[k];}
            a[last].next = 0;
        }
        d *= radix;
    }
}

```

# 作业

- 课后题目 **4.5 4.15**
- 阅读 **P113** 算法 **6.5 Split**
- 课后题目 **6.31 6.32 6.33 6.35**
- 请将快速排序递归算法改成非递归实现
- 实验比较归并排序、希尔排序、堆排序、快速排序，从时间、空间与稳定性方面，理论分析它们的差异
- 本章：对排序算法进行系统学习掌握，体会算法研究对一类特定问题的求解方法；体会算法设计与数据组织之间的相互关系