

# 数据结构

北京邮电大学 网络空间安全学院  
武 斌



# 上次课内容

## 上一次课程（内部排序(上)）内容：

- 理解排序的定义和各种排序方法的特点，并能加以灵活应用
- 了解排序方法有不同的分类方法，基于“关键字间的比较”进行排序的方法可以按排序过程所依据的不同原则分为插入排序、交换排序、选择排序、归并排序和计数排序等五类
- 掌握各种排序方法的时间复杂度的分析方法，能从“关键字间的比较次数”分析排序算法的平均情况和最坏情况的时间性能。





# 本次课程学习目标

学习完本次课程（内部排序(下)），  
您应该能够：

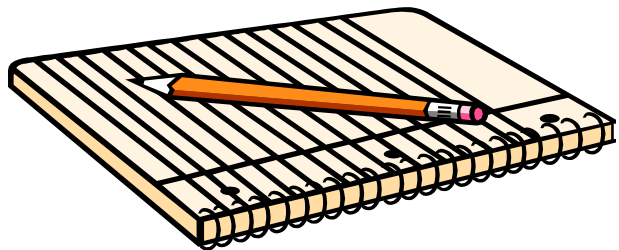
- 理解选择排序、归并排序和基数排序等的定义和各种排序方法的特点，并能加以灵活应用
- 掌握选择排序、归并排序和基数排序等方法的时间复杂度的分析方法
- 理解排序方法“稳定”或“不稳定”的含义，弄清楚在什么情况下要求应用的排序方法必须是稳定的
- 通过各种内部排序方法的比较讨论，领会各种方法的特点





# 本章课程内容（第十章 内部排序）

- 10.1 概述
- 10.2 插入排序
- 10.3 快速排序
- 10.4 选择排序
- 10.5 归并排序
- 10.6 基数排序
- 10.7 各种内部排序方法的比较讨论





# 选择排序

- **选择排序**(Selection Sort)的基本思想是：每一趟在 $n-i+1$  ( $i=1,2,\dots,n-1$ )个记录中选取关键字最小的记录作为有序序列中第 $i$ 个记录。其中最简单且为读者最熟悉的是**简单选择排序**(Simple Selection Sort)。

## → 1、简单选择排序

第  $i$  ( $i=1,2,\dots,n-1$ )趟的简单选择排序（序列中前  $i-1$  个记录的关键字均小于后  $n-i+1$  个记录的关键字）的作法是，在后  $n-i+1$  个记录中选出关键字最小的记录，并将它和第  $i$  个记录进行互换。如图所示。



# 选择排序

简单选择的基本思想为：

假设在排序过程中，  
记录序列 $R[1..n]$ 的状态为：





# 选择排序

## ● 算法10.11

```
void SelectSort (SqList &L)
```

```
{ // 对顺序表L作简单选择排序
```

```
    for (i=1; i<L.length; ++i) { // 选择第 i 小的记录，并交换到位
```

```
        j = SelectMinKey (L,i); // 在L.r[i..L.length]中选择关键字最小的记录
```

```
        if ( i!=j ) L.r[i]  $\longleftrightarrow$  L.r[j];    // 与第 i 个记录互换
```

```
    } // for
```

```
} // SelectSort
```

- 无论待排序列处于什么状态，选择排序所需进行的“比较”次数都相同，为  $\sum_{i=1}^{n-1} (n-i) = \frac{n(n-1)}{2}$ ，而“移动”次数在待排序列为“正序”时达最小为0，在“逆序”时达最大为  $3(n-1)$ 。

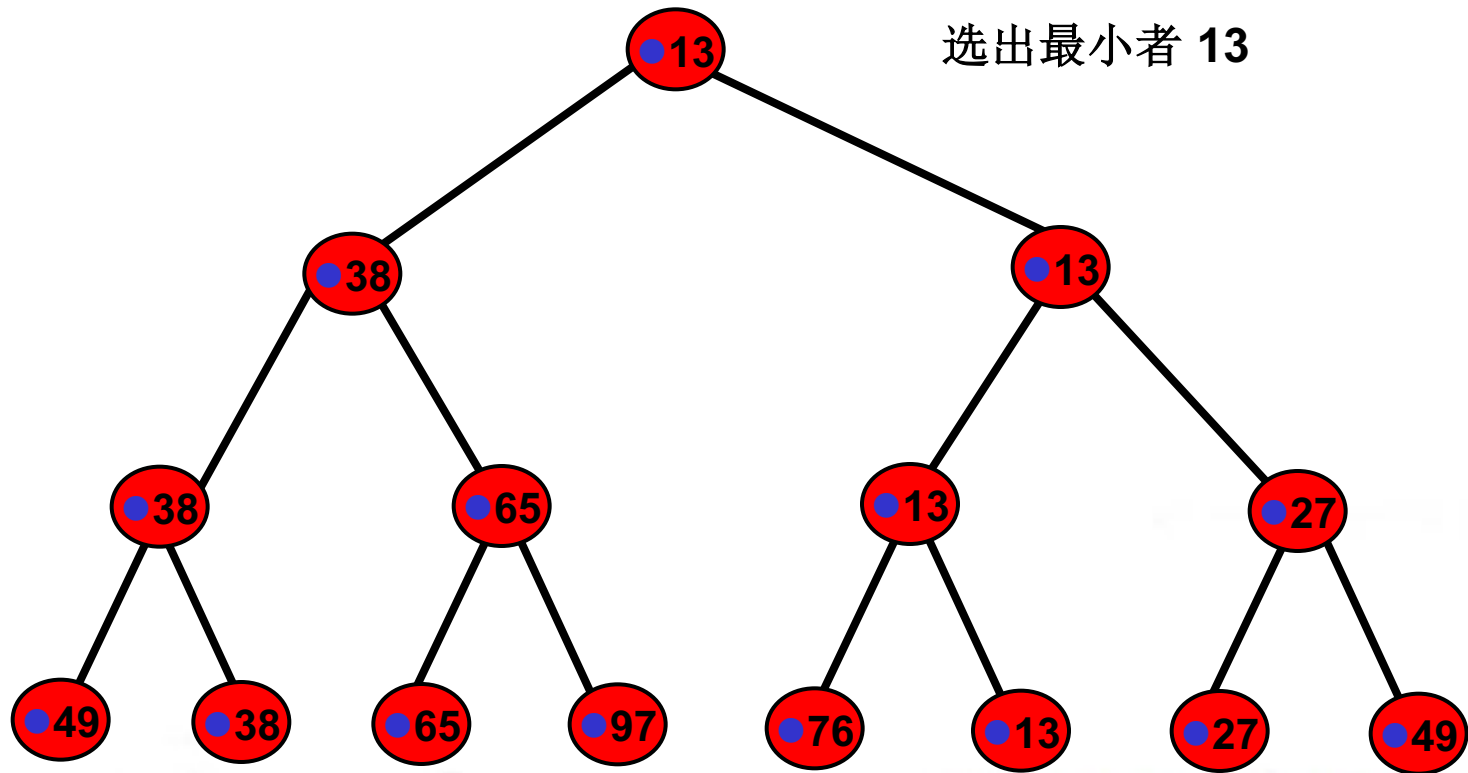
- 因此其时间复杂度为：  $O(n^2)$



# 树形选择排序

## ●2、树形选择排序

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。

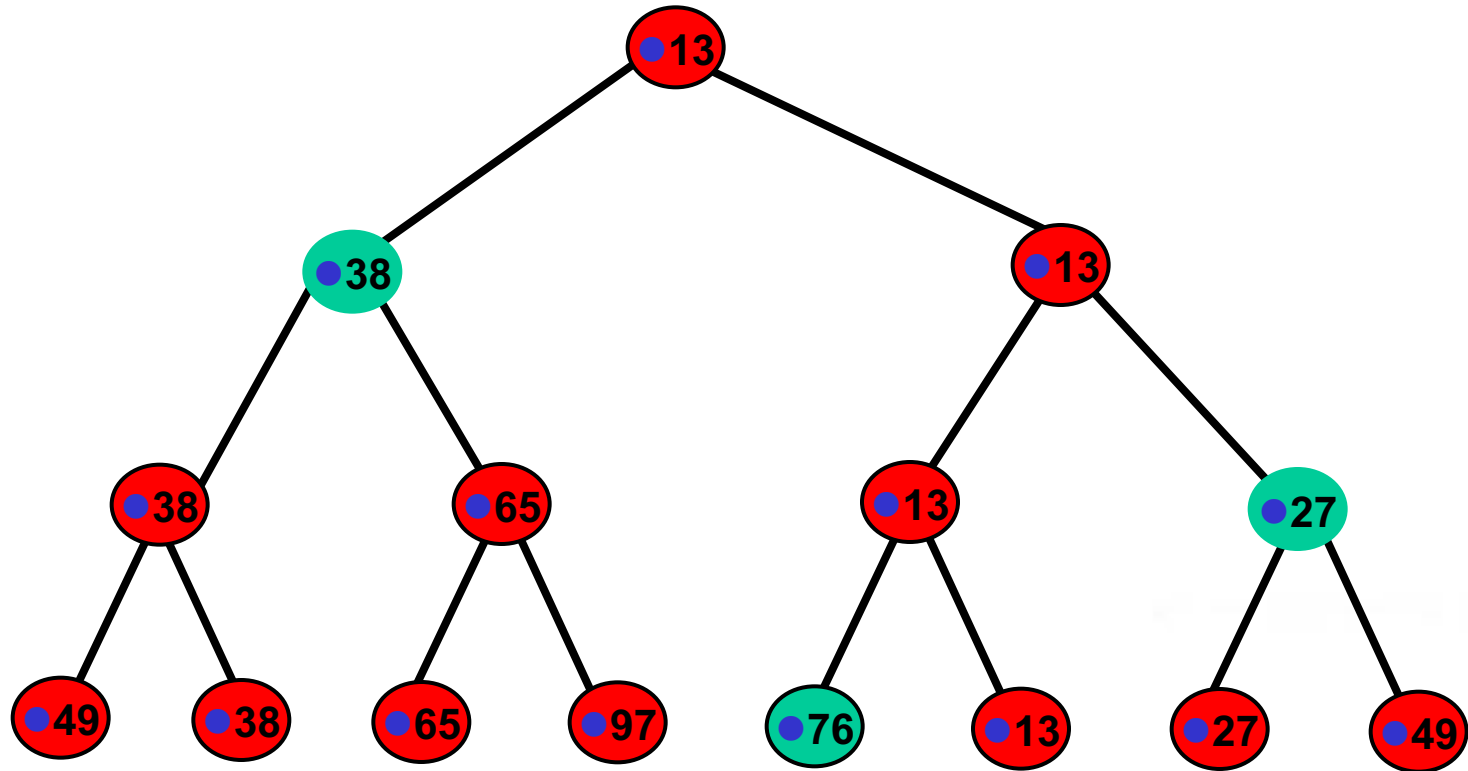






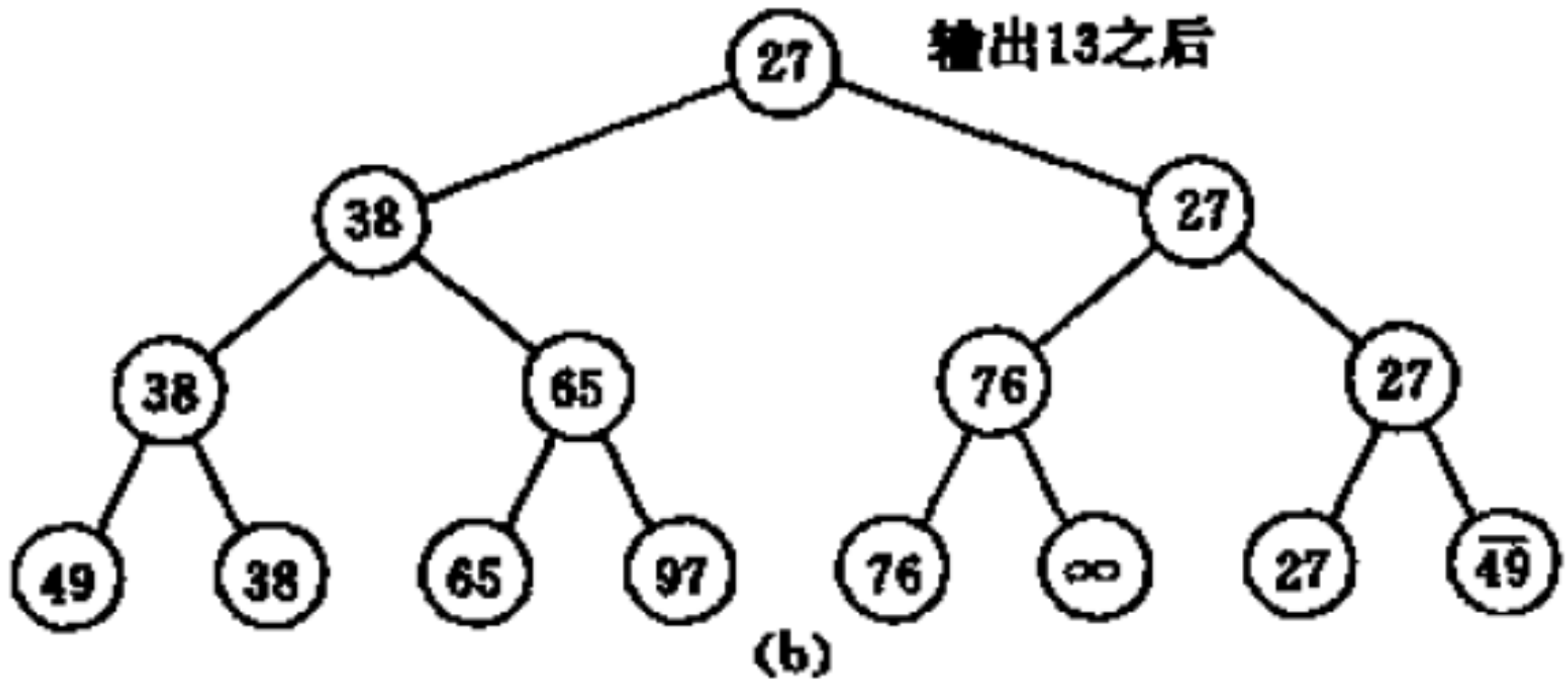
## 树形选择排序

- 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。





## 树形选择排序





# 选择排序

## → 3、堆排序

“堆排序”也是一种选择类的排序方法，每一趟从记录的无序序列中选出一个关键字最大或最小的记录，和简单选择所不同的是，**在第一趟选最大或最小关键字记录时先“建堆”**，从而减少之后选择次大或次小关键字等一系列记录时所需的比较和移动次数。

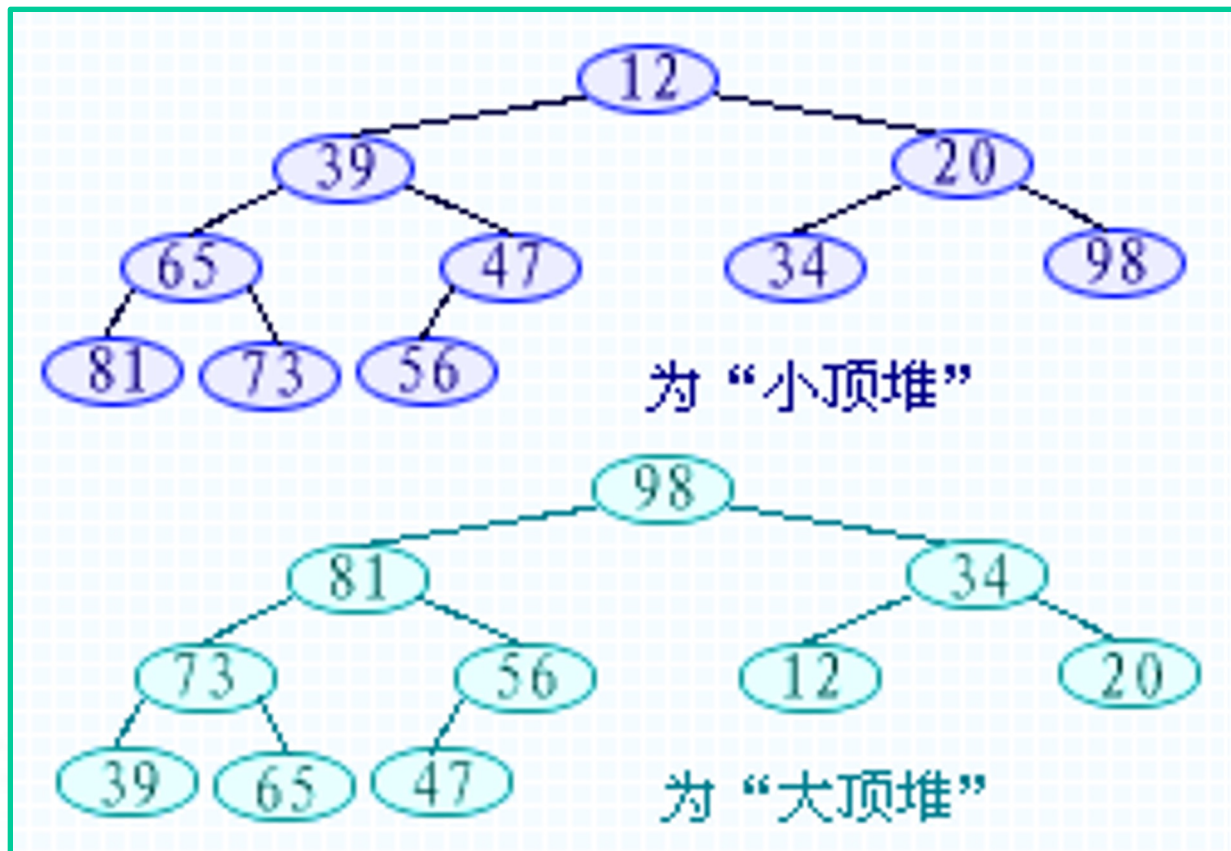
**定义：**若含 $n$ 个元素的序列  $\{k_1, k_2, \dots, k_n\}$  满足下列关系则称作“小顶堆”或“大顶堆”。“堆顶”元素为序列中的“最小值”或“最大值”。

$$\begin{cases} k_i \leq k_{2i} \\ k_i \leq k_{2i+1} \end{cases} \text{ 或 } \begin{cases} k_i \geq k_{2i} \\ k_i \geq k_{2i+1} \end{cases}, \text{ 其中 } i = 1, 2, \dots, \left\lfloor \frac{n}{2} \right\rfloor$$



# 选择排序

- 例如，{12, 39, 20, 65, 47, 34, 98, 81, 73, 56}为“小顶堆”；  
{98, 81, 34, 73, 56, 12, 20, 39, 65, 47}为“大顶堆”。
- 若将上述数列视为一个完全二叉树，则堆顶元素 $k_1$ 即为二叉树的根结点， $k_{2i}$ 和  $k_{2i+1}$ 分别为 $k_i$ 的“左子树根”和“右子树根”，如下图所示。





# 选择排序

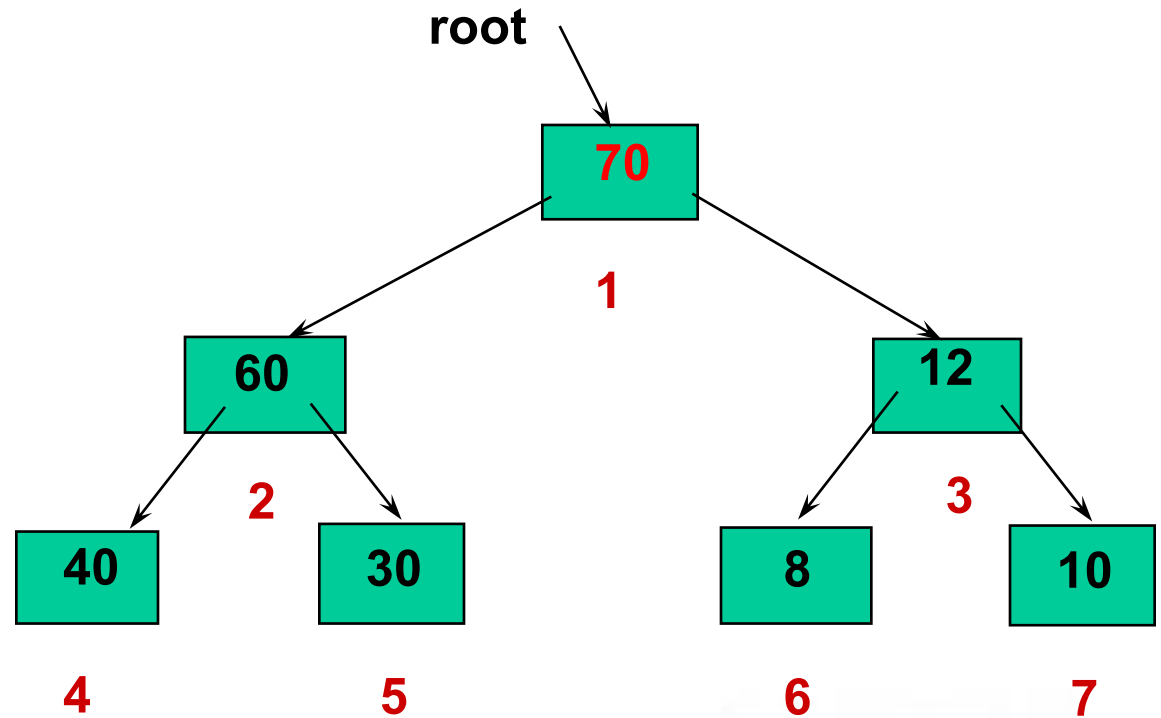
- 利用堆的特性进行的排序方法即为“**堆排序**”，其两个**关键问题**是：
  - 1) 如何将一个无序序列调整为堆？
  - 2) 如何在输出堆顶元素之后，调整剩余元素为一个新的堆？
- **首先看第二个问题**
  - 输出堆顶元素，以堆中最后一个元素代替之
  - 此时，堆顶元素的左右子树都为堆，仅需自上至下进行调整重新形成堆，这种自堆顶至叶子的调整过程叫做“**筛选**”。



# 堆排序

values

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

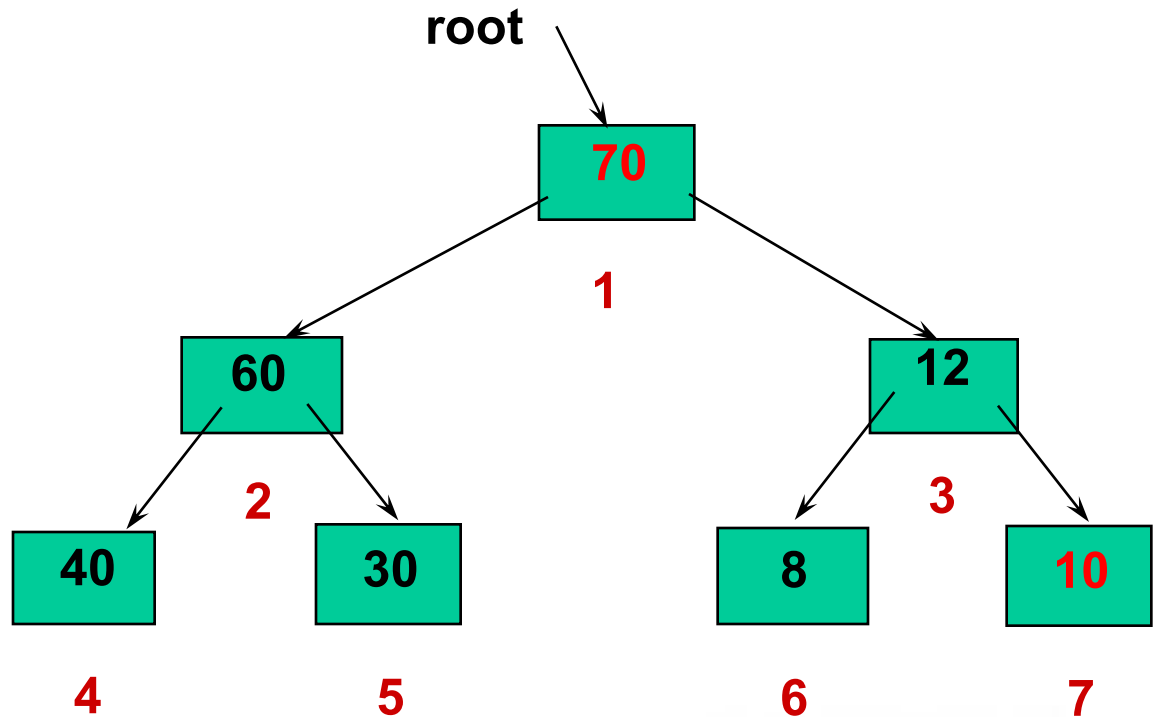




# 堆排序

values

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





# 堆排序

values

[ 1 ]

10

[ 2 ]

60

[ 3 ]

12

[ 4 ]

40

[ 5 ]

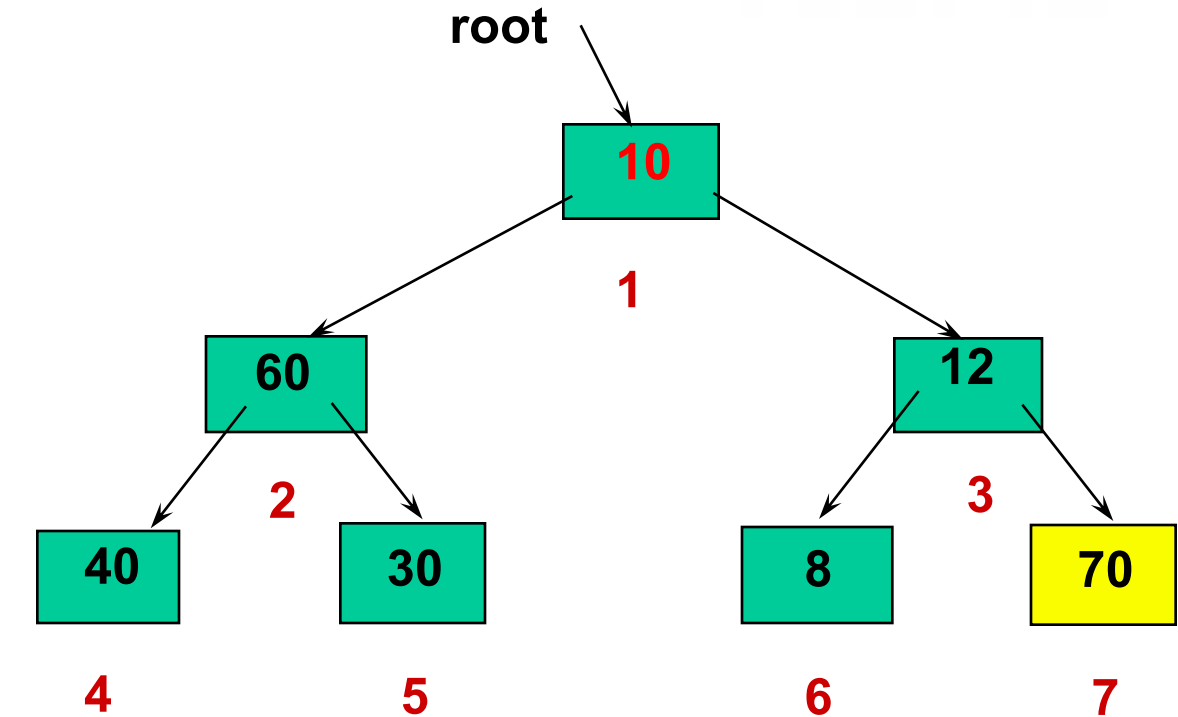
30

[ 6 ]

8

[ 7 ]

70



NO NEED TO CONSIDER AGAIN

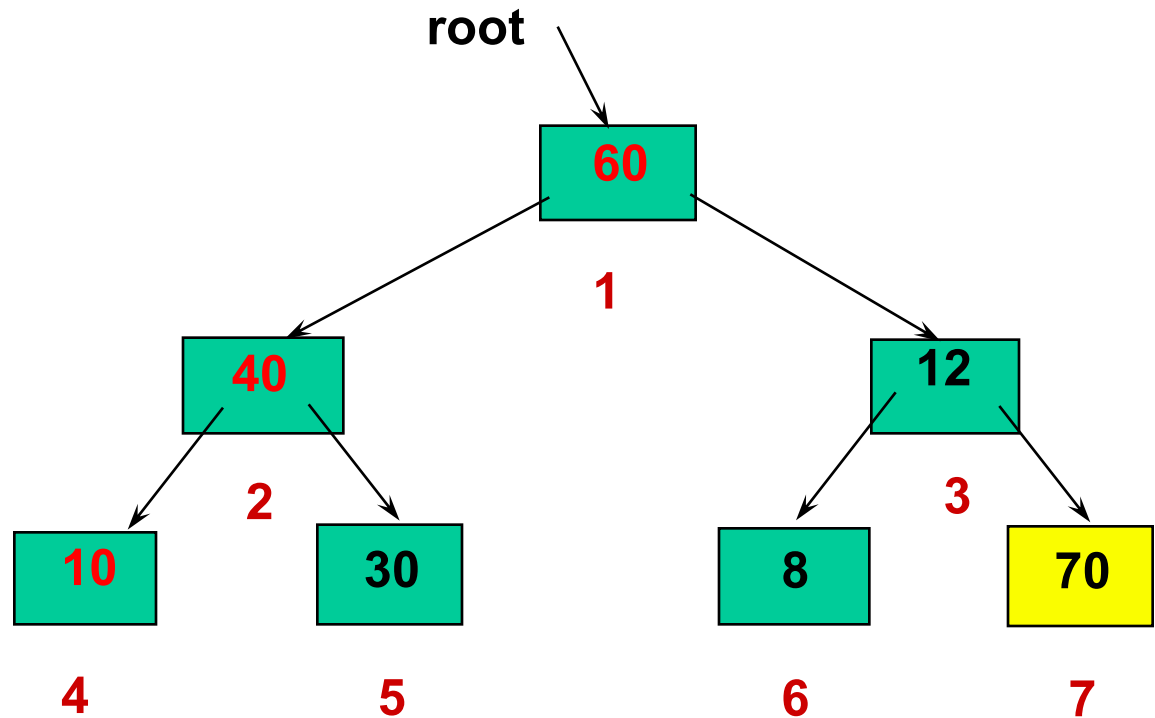




# 堆排序

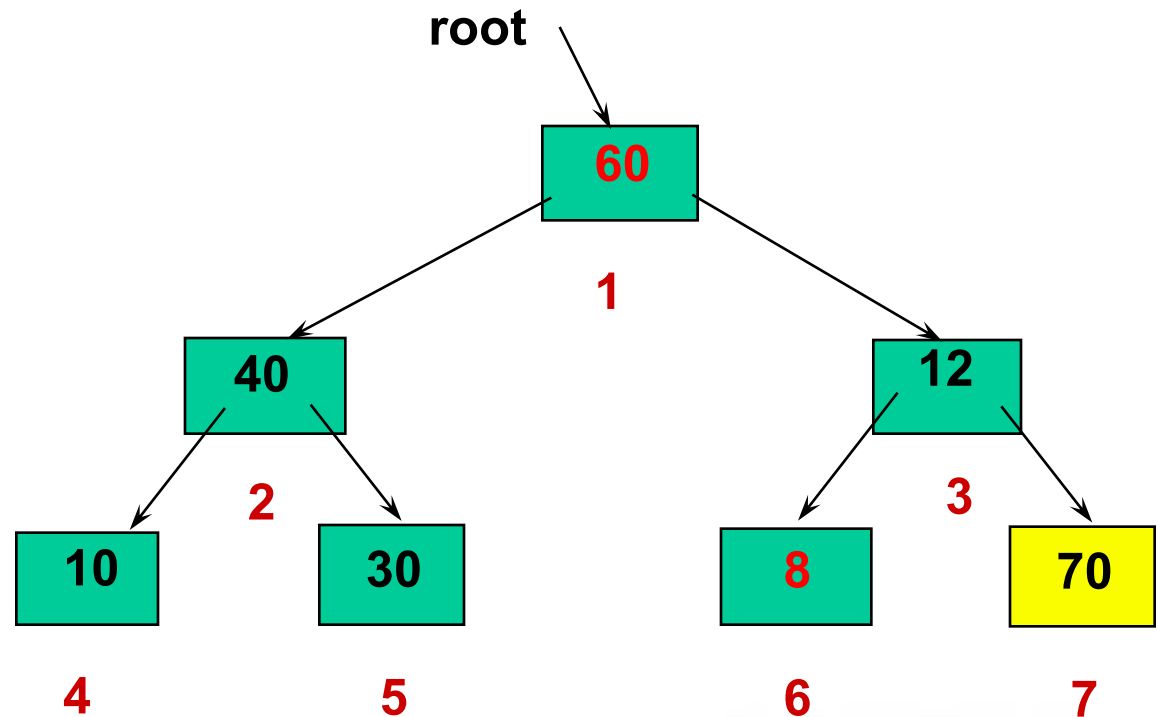
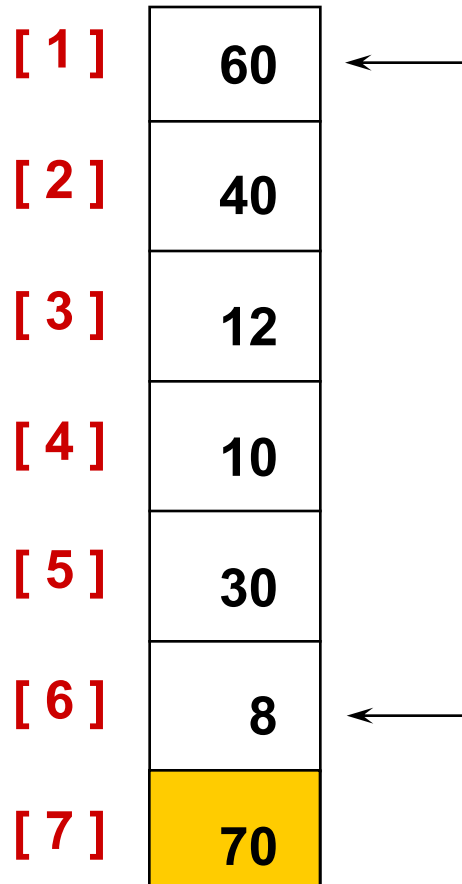
values

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





# 堆排序

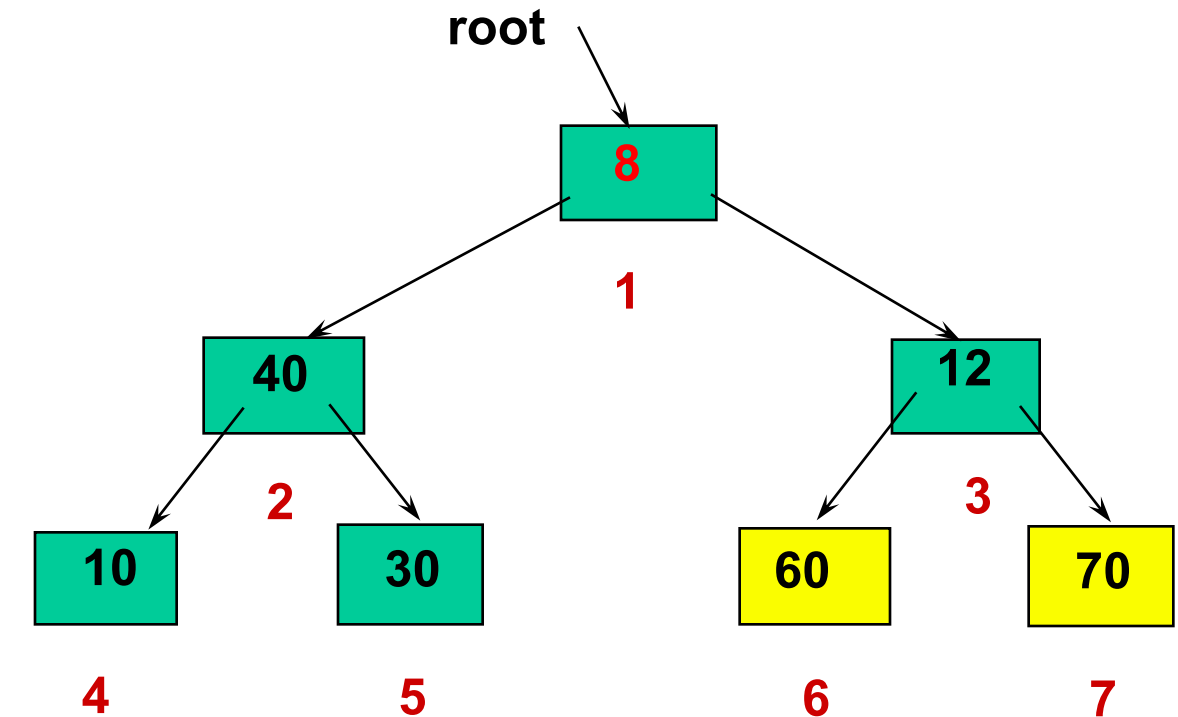




# 堆排序

values

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



**NO NEED TO CONSIDER AGAIN**



# 堆排序

values

[ 1 ]

40

[ 2 ]

30

[ 3 ]

12

[ 4 ]

10

[ 5 ]

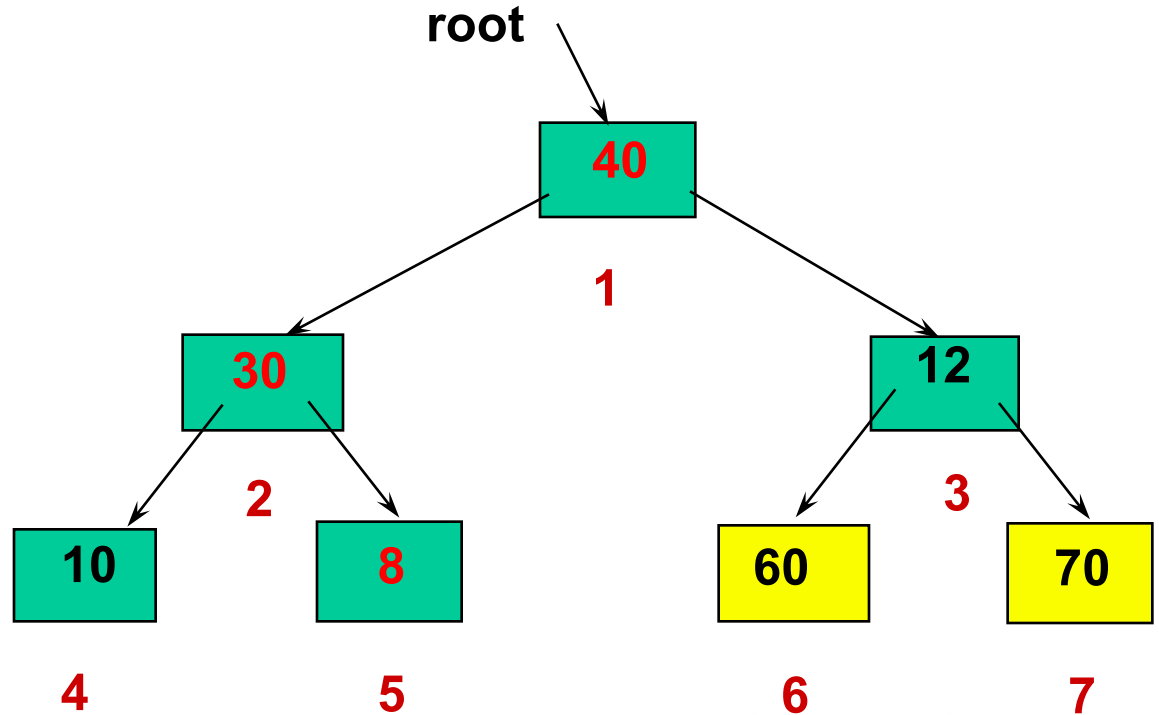
8

[ 6 ]

60

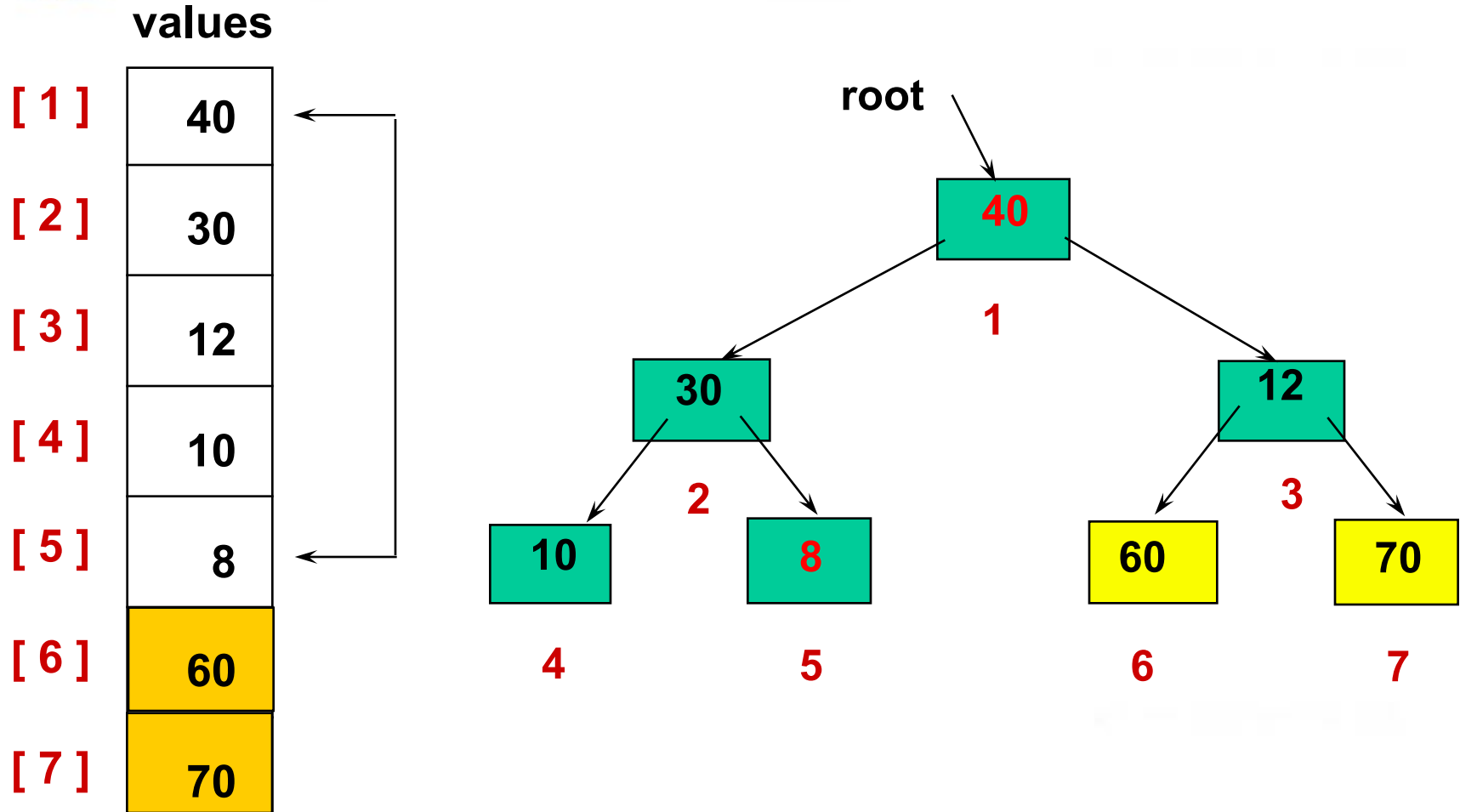
[ 7 ]

70





# 堆排序





# 堆排序

values

[1]

8

[2]

30

[3]

12

[4]

10

[5]

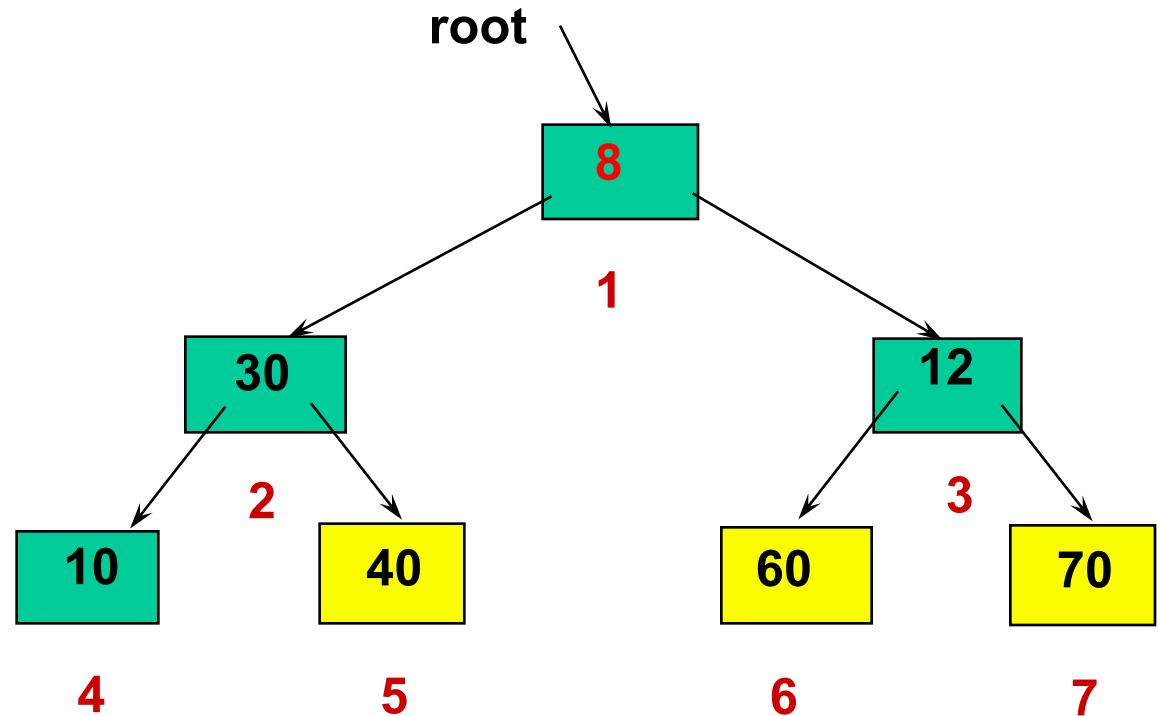
40

[6]

60

[7]

70



NO NEED TO CONSIDER AGAIN



# 堆排序

values

[ 1 ]

30

[ 2 ]

10

[ 3 ]

12

[ 4 ]

8

[ 5 ]

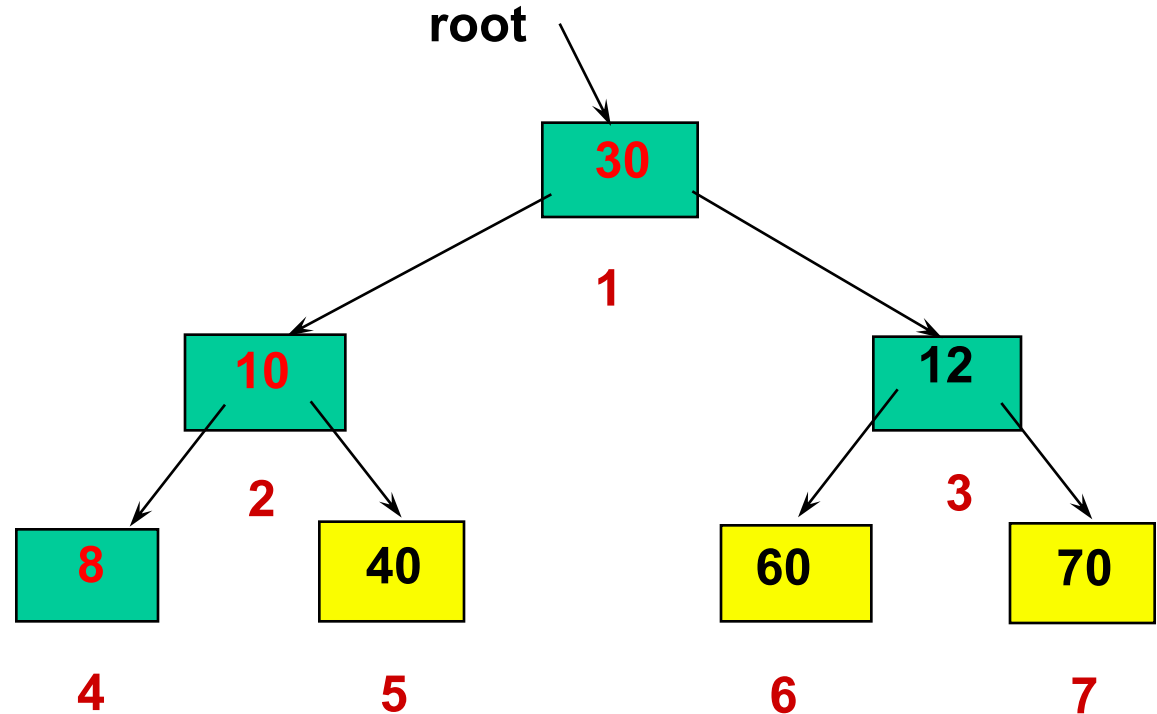
40

[ 6 ]

60

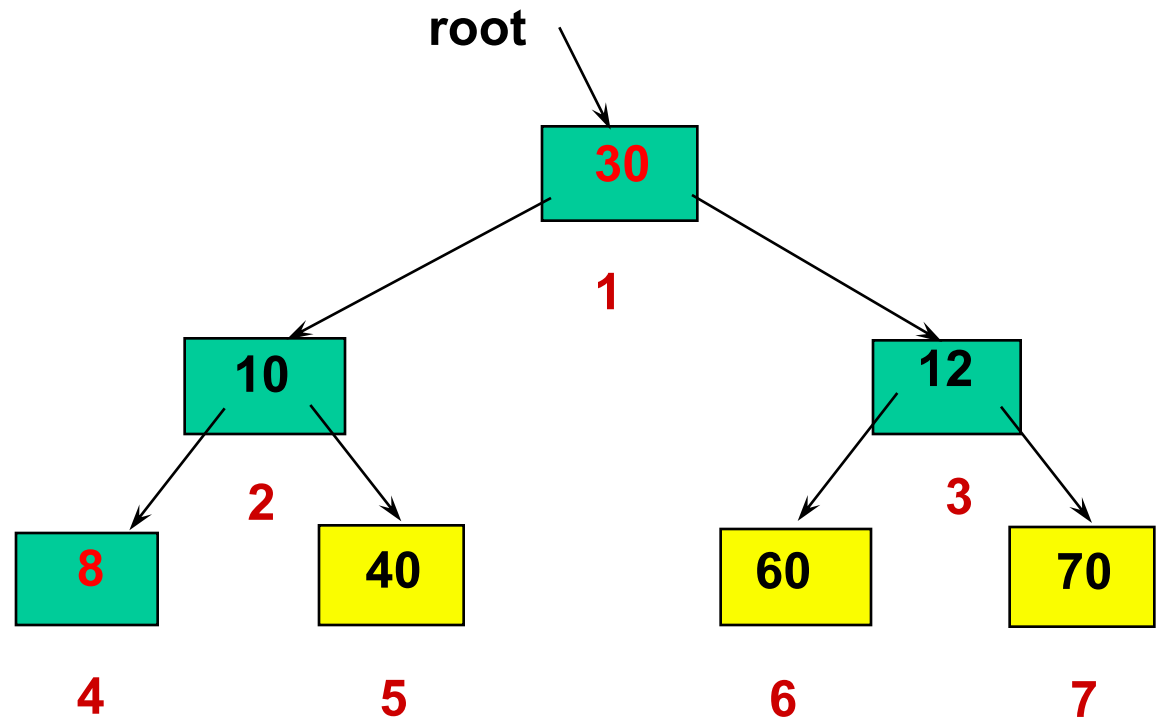
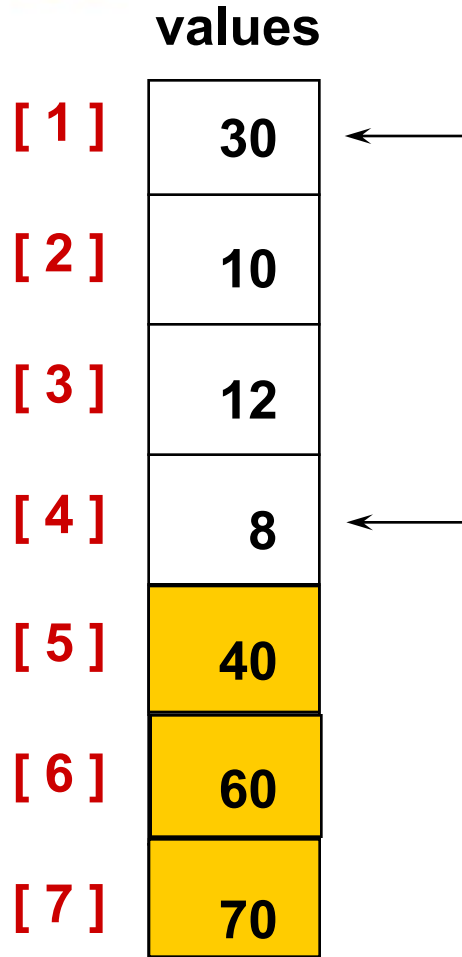
[ 7 ]

70





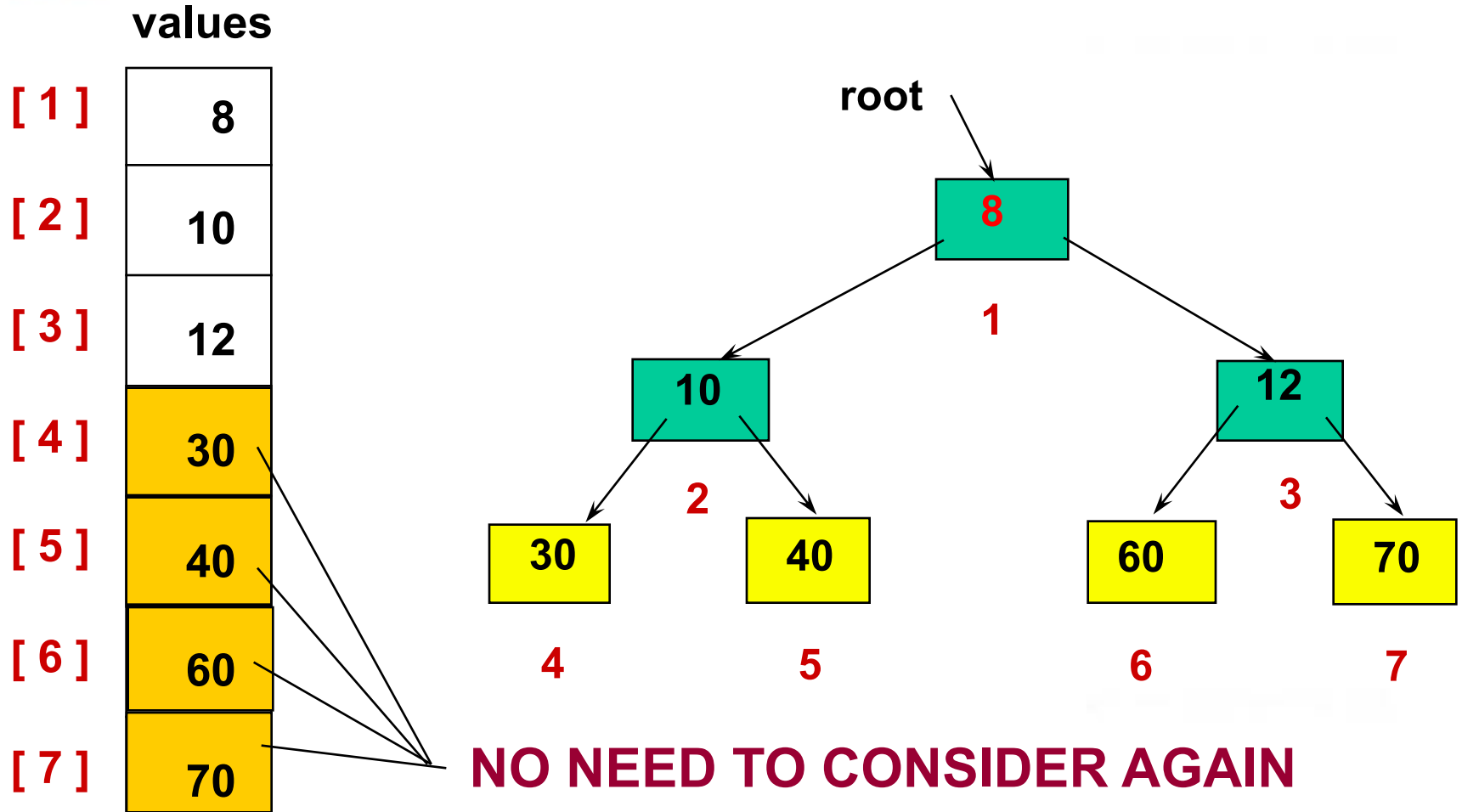
# 堆排序







# 堆排序

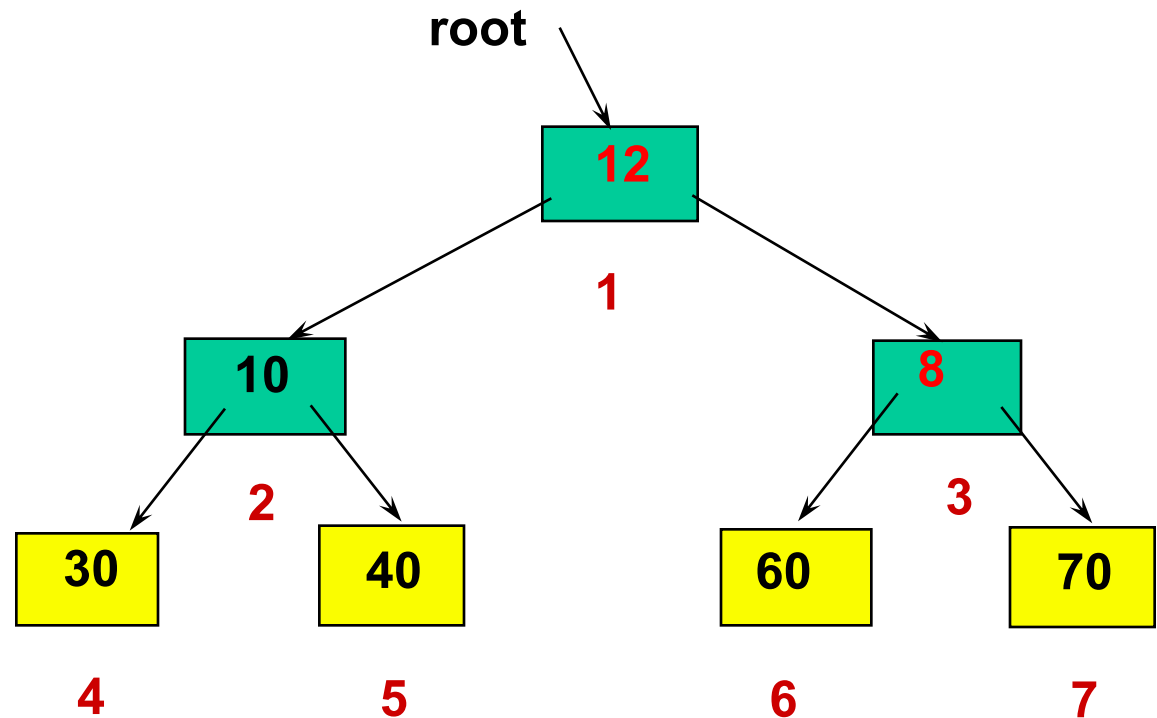




# 堆排序

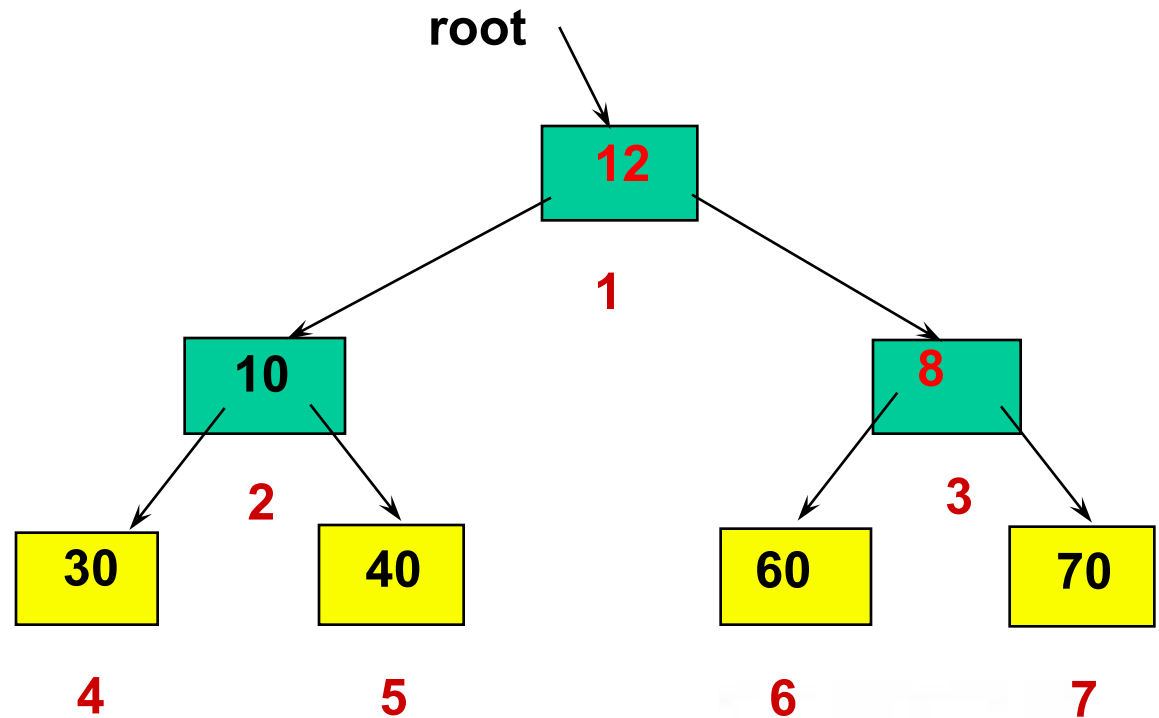
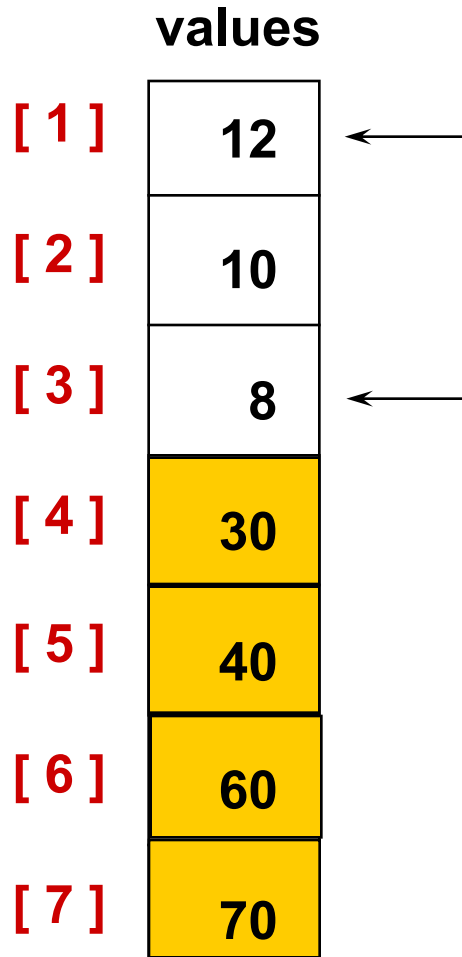
values

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



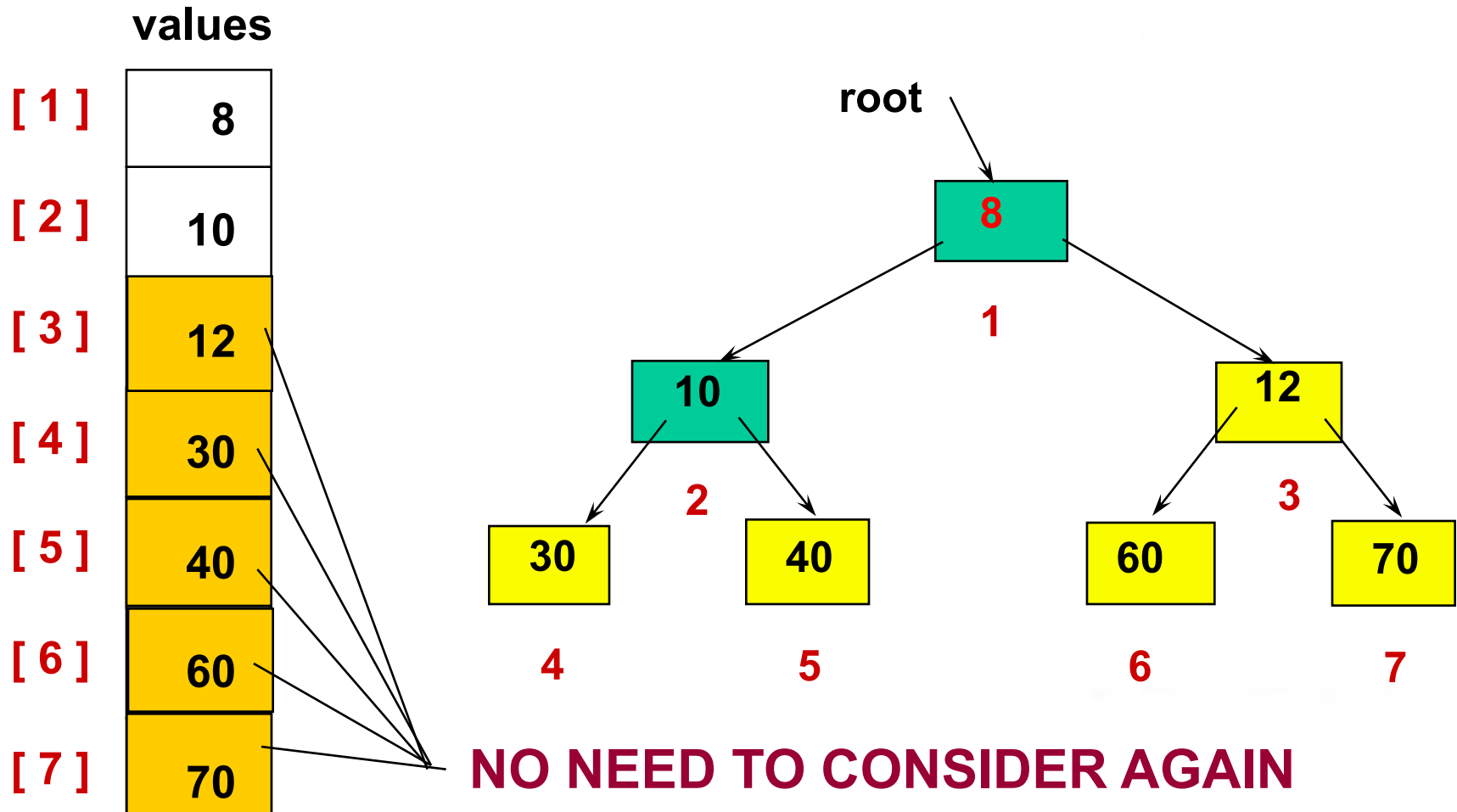


# 堆排序





# 堆排序

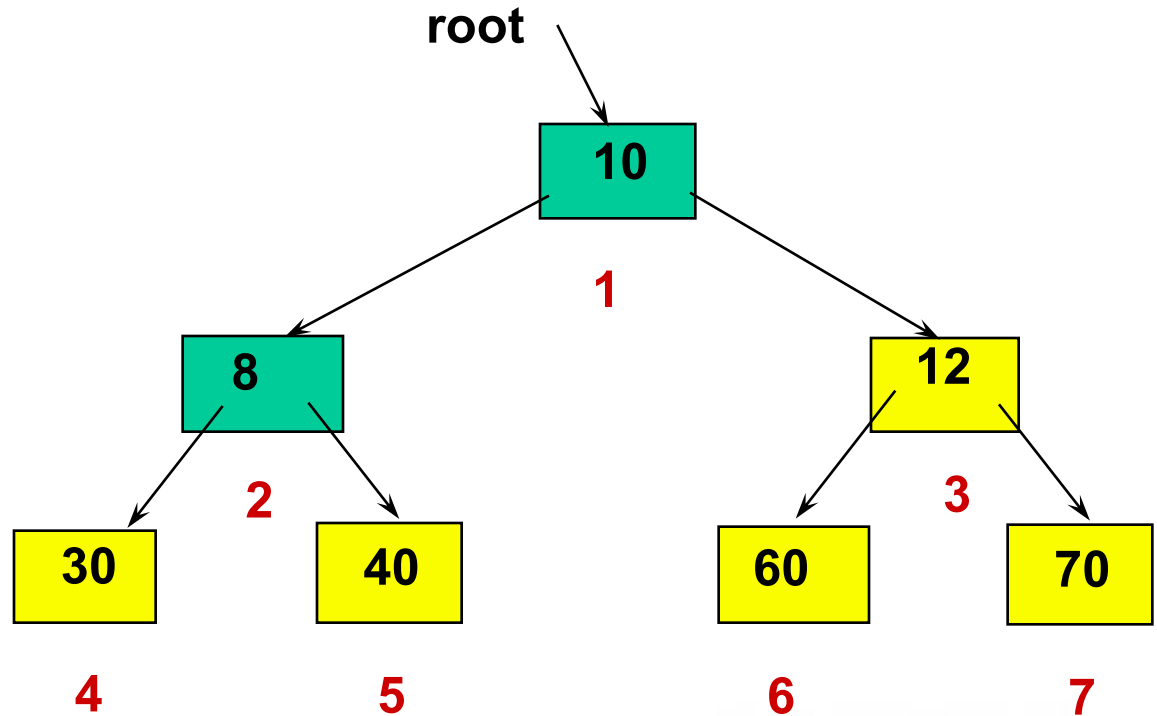




# 堆排序

values

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

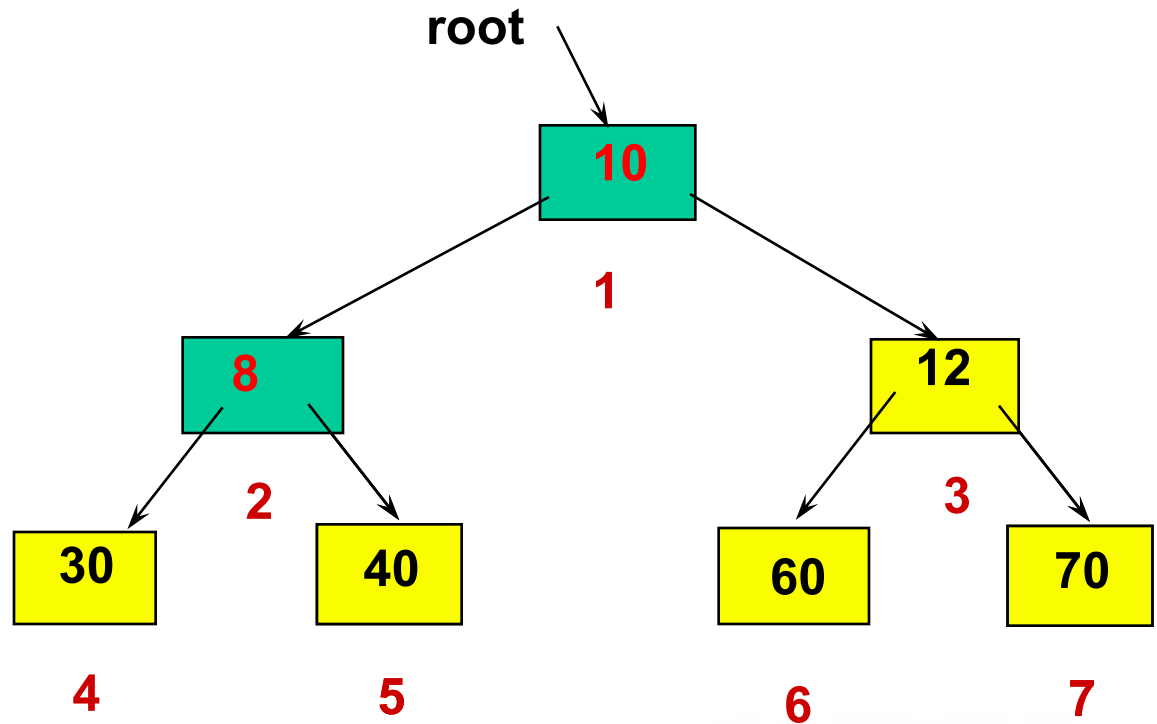




# 堆排序

values

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

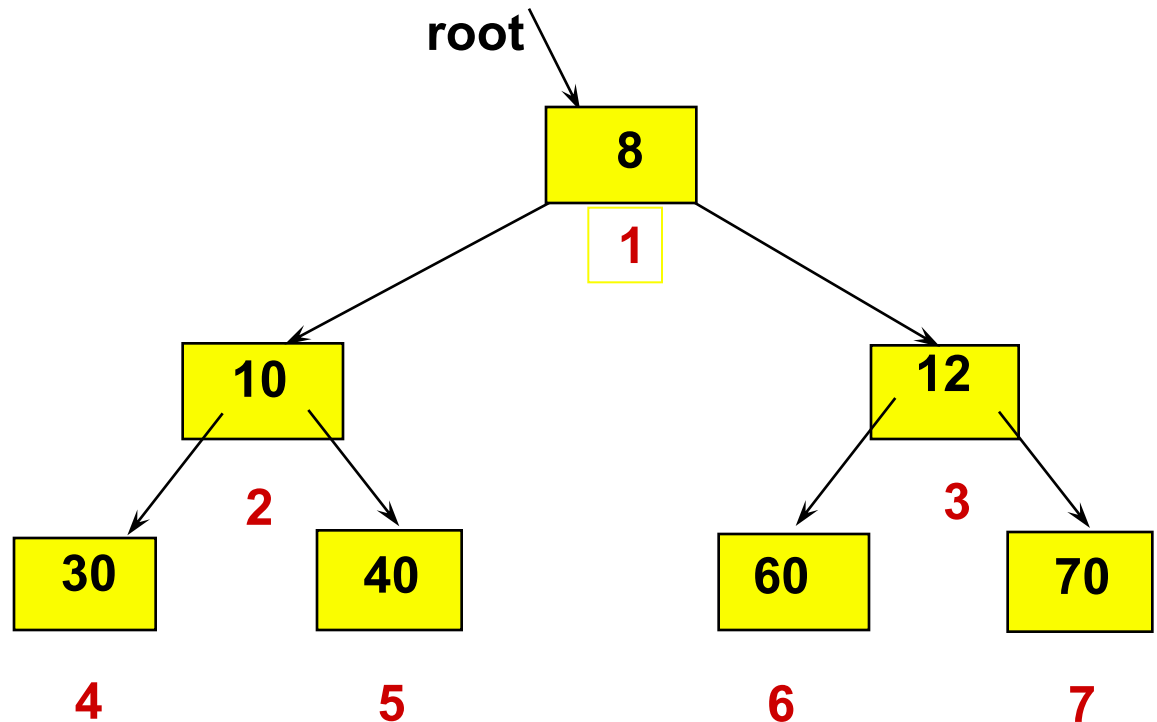




# 堆排序

数组 h

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



**ALL ELEMENTS ARE SORTED**



# 堆排序

## ● 算法10.12

```
typedef SqList HeapType // 堆采用顺序表存储表示
void HeapAdjust (SqList &H, int s, int m)
{ // 已知H.r[s..m]中记录的关键字除H.r[s].key之外均满足堆的定义,
  // 本函数依据关键字的大小对H.r[s]进行调整, 使H.r[s..m]成为一
  // 个大顶堆 (对其中记录的关键字而言)
  rc = H.r[s]; // 暂存根结点的记录
  for ( j=2*s; j<=m; j*=2 ) { // 沿关键字较大的孩子结点向下筛选
    if ( j<m &&LT(H.r[j].key,H.r[j+1].key) )
      ++j; // j 为关键字较大的孩子记录的下标
    if (! LT(H.r[0].key, H.r[j].key )) break; // rc应插入在位置s上
    H.r[s] = H.r[j]; s = j; //r[0]<r[j]
  } // for
  H.r[s] = rc; // 插入
} // HeapAdjust
```





# 堆排序

## ●再看第一个问题——建初始堆

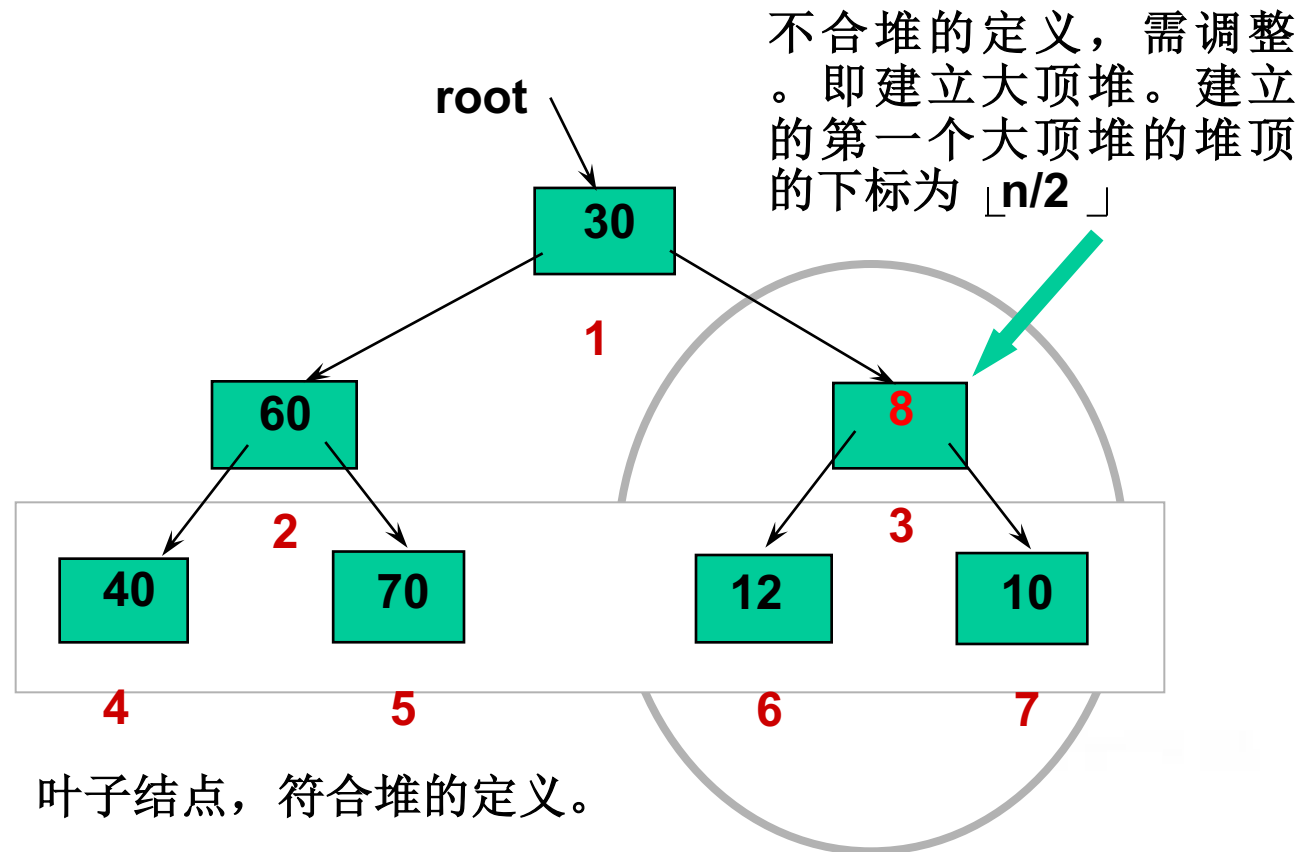
- 从一个无序序列建堆的过程就是一个反复“**筛选**”的过程
- 若将此序列看作一个完全二叉树，则最后一个非终端节点是第 $\lfloor n/2 \rfloor$ 个元素，由此“筛选”从第 $\lfloor n/2 \rfloor$ 个元素开始（因为终端节点符合堆的定义）
- 演示（10-4-4）



# 堆排序

数组 h

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





# 堆排序

数组 h

[1]

30

[2]

60

[3]

12

[4]

40

[5]

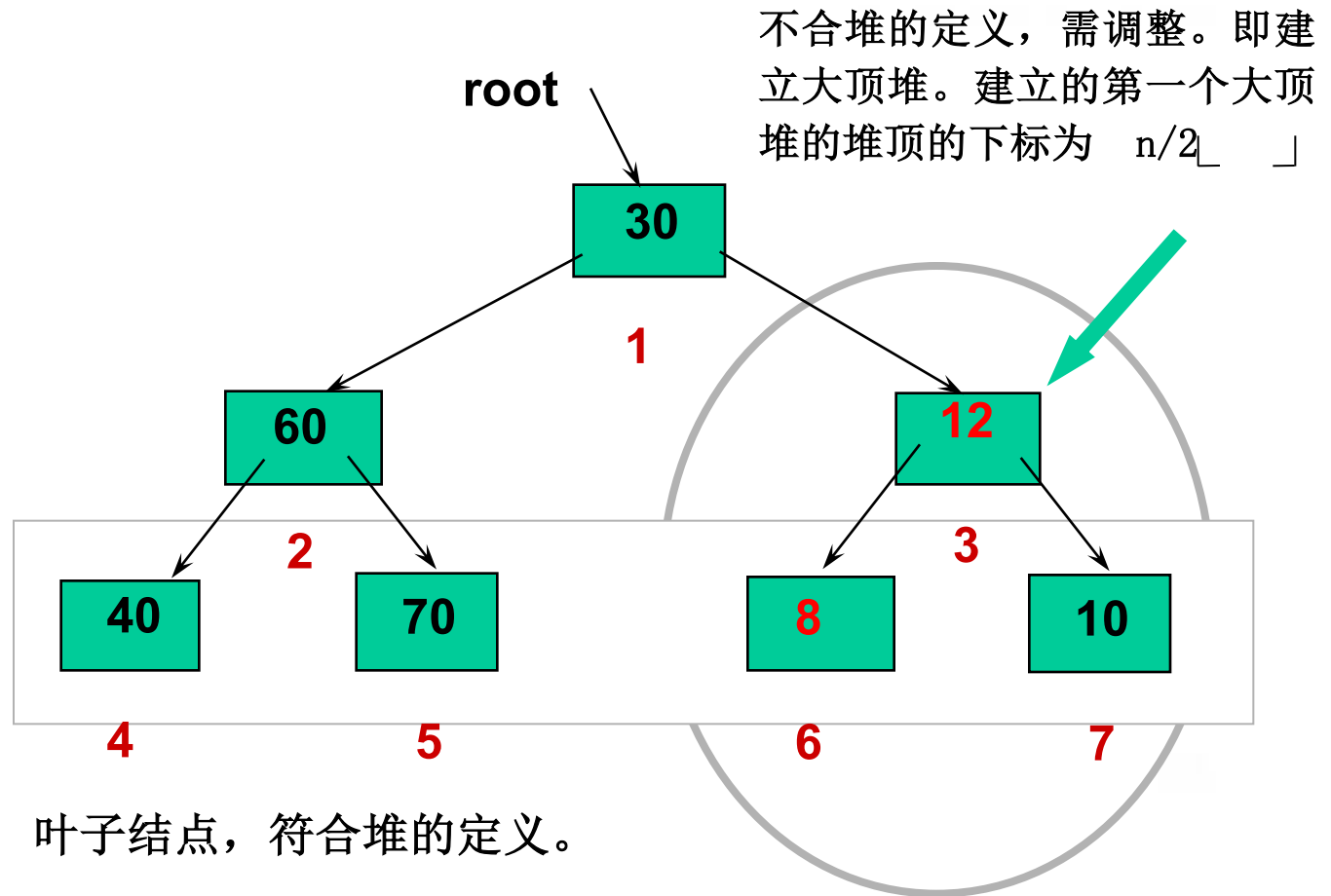
70

[6]

8

[7]

10



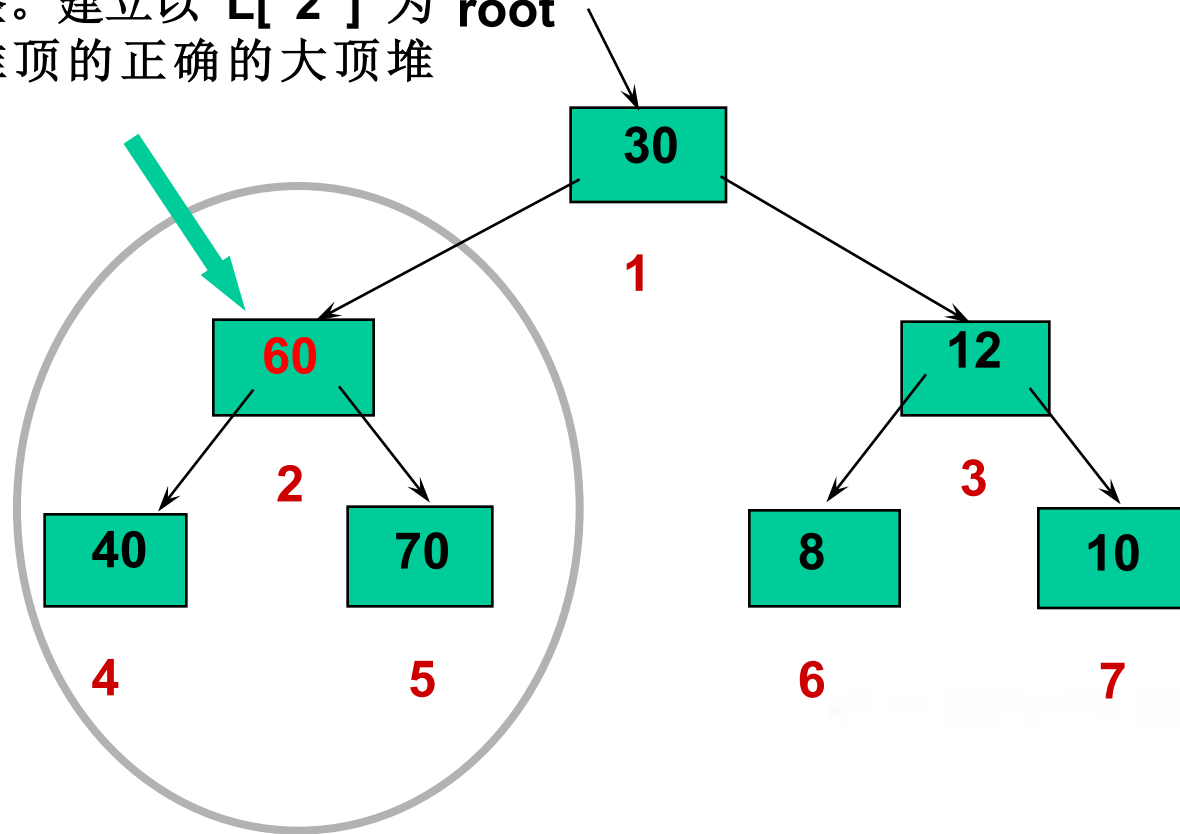


# 堆排序

数组 h

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

不合堆的定义，需调整。建立以 L[ 2 ] 为 root 的堆顶的正确的大顶堆。



36

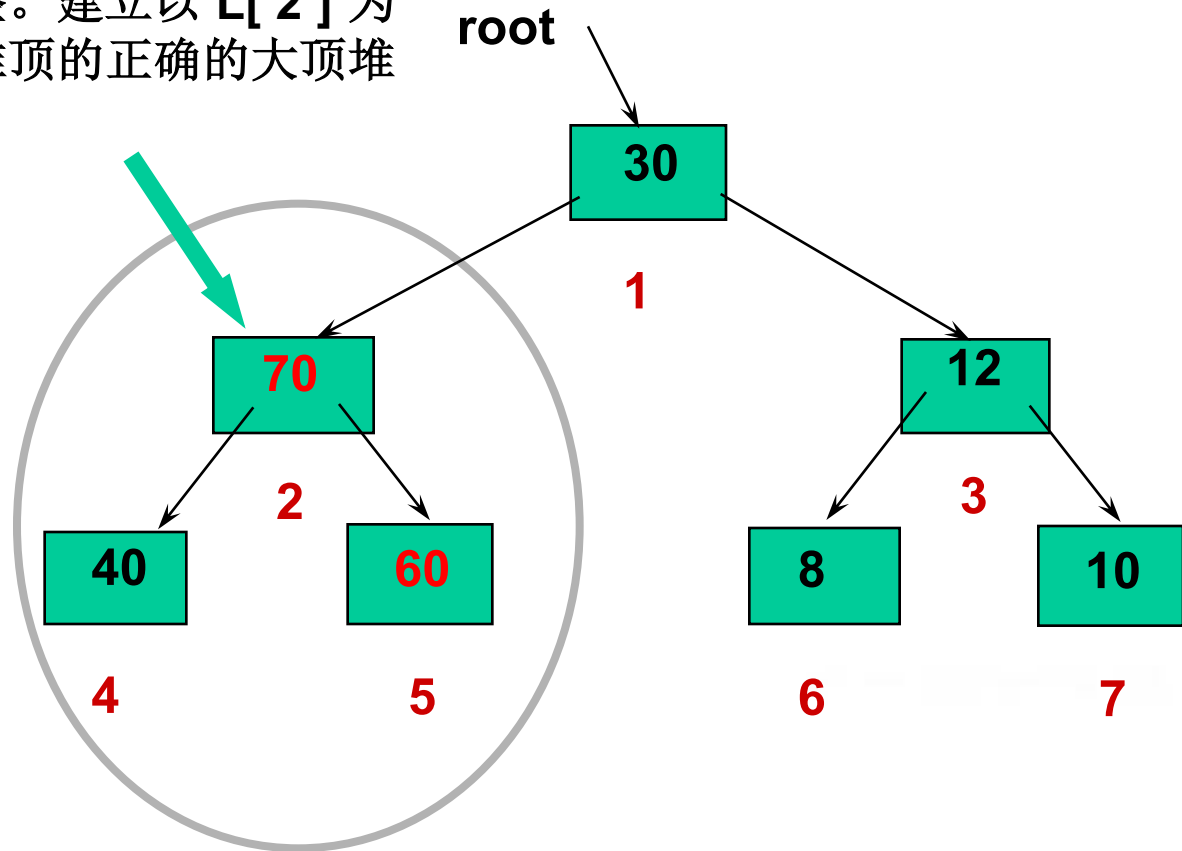


# 堆排序

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

不合堆的定义，需调整。建立以 L[2] 为堆顶的正确的大顶堆。

。





# 堆排序

数组 h

[ 1 ]

30

[ 2 ]

70

[ 3 ]

12

[ 4 ]

40

[ 5 ]

60

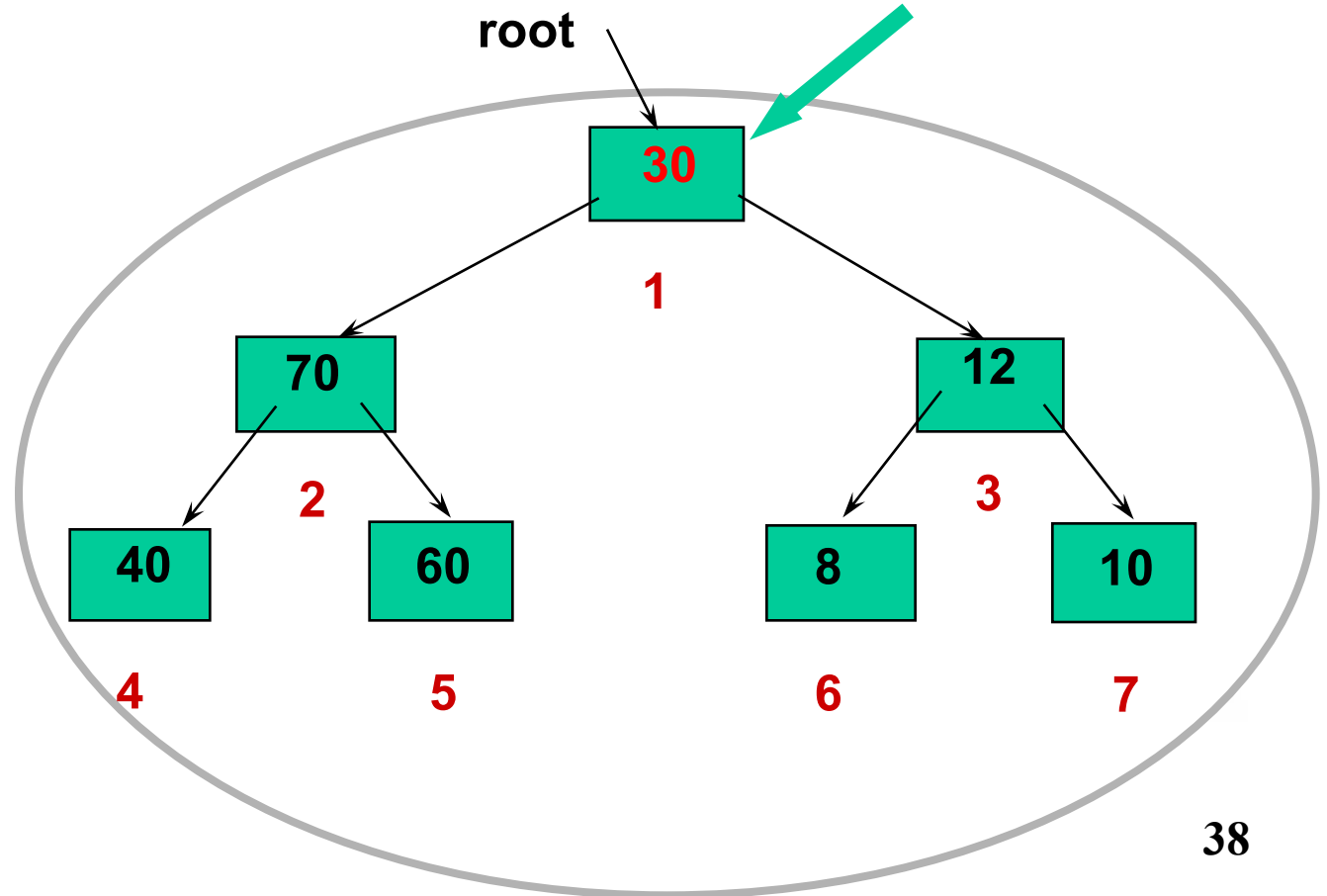
[ 6 ]

8

[ 7 ]

10

不合堆的定义，需调整。  
建立以 L[ 1 ] 为堆顶的正确的  
大顶堆。



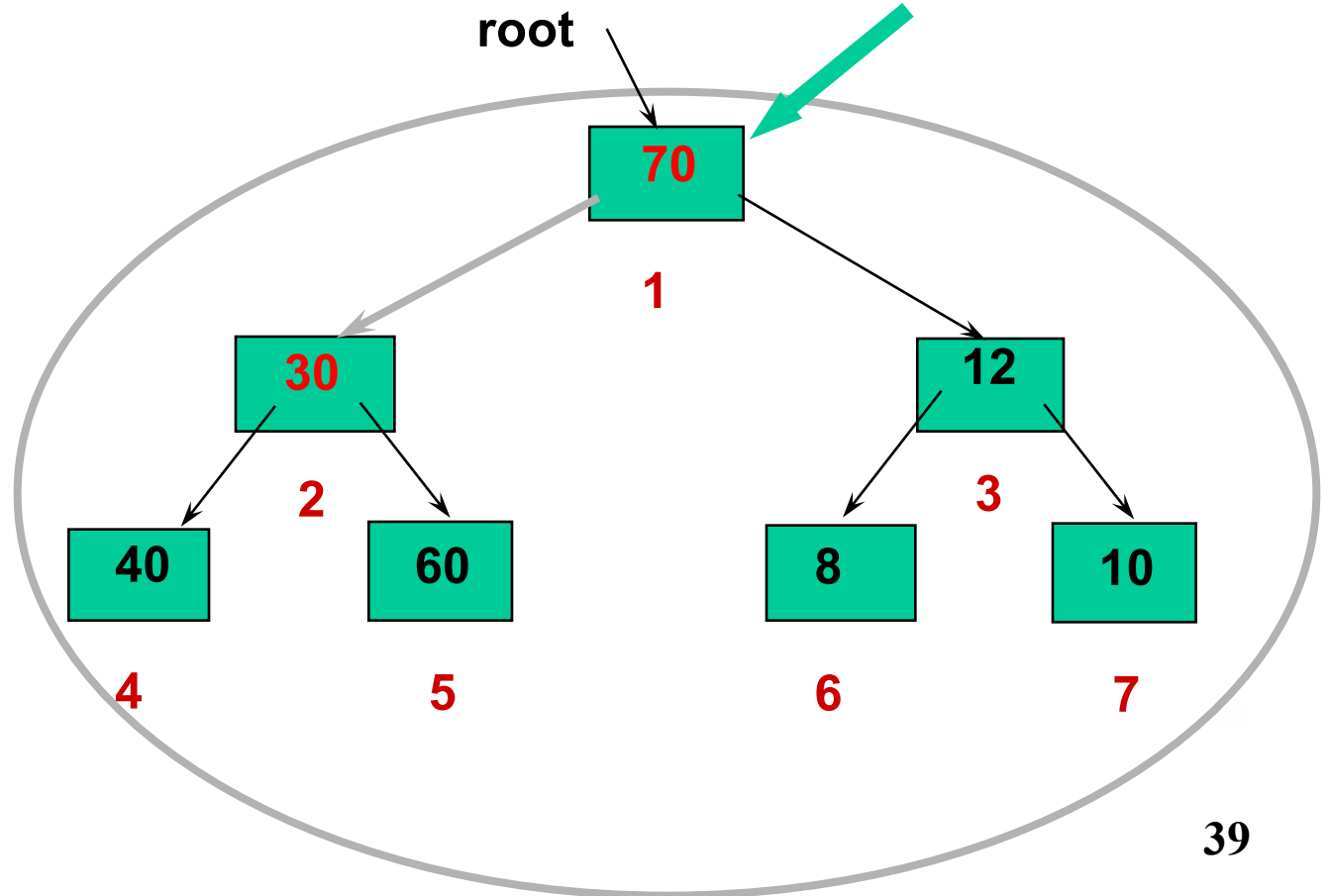


# 堆排序

数组 h

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

不合堆的定义，需调整。  
建立以  $L[1]$  为堆顶的正确的大顶堆。



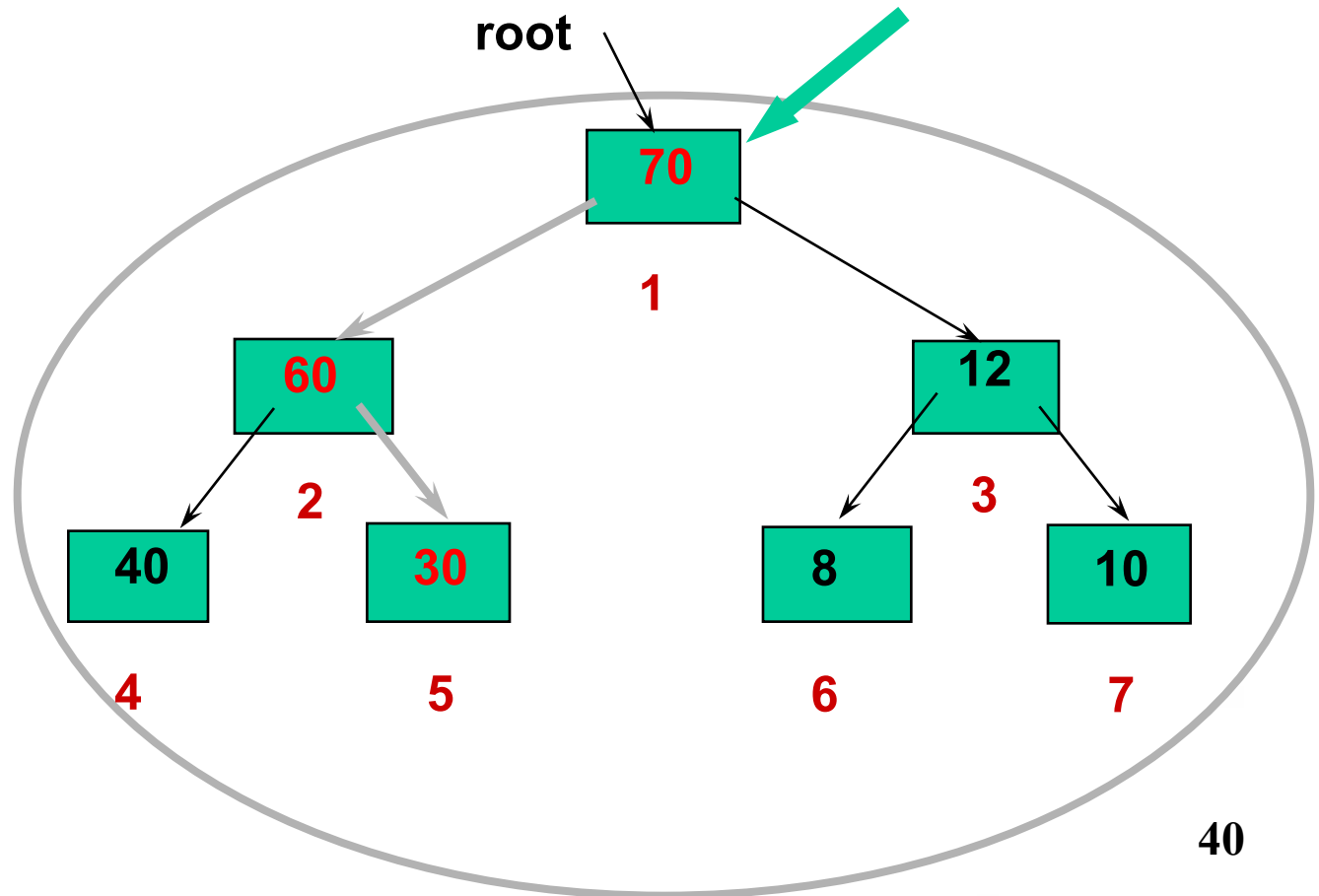


# 堆排序

不合堆的定义，需调整。  
建立以  $L[1]$  为堆顶的正确的大顶堆。

数组 h

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







# 堆排序

## ●算法 10.13

```
void HeapSort ( SqList &H )
{   // 对顺序表H进行堆排序
    for ( i=H.length/2; i>0; --i )    // 将 H.r[1..H.length] 建成大顶堆
        HeapAdjust ( H, i, H.length );
    for ( i=H.length-1; i>1; --i ) {
        H.r[1]  $\longleftrightarrow$  H.r[i]; // 将堆顶记录和当前未经排序子序列H.r[1...i]中
                                   // 最后一个记录相互交换
        HeapAdjust(H, 1, i-1); // 将H.r[1...i-1]重新调整为大顶堆
    } // for
} // HeapSort
```



# 堆排序

- 堆排序的时间复杂度分析

- 1.对深度为K的堆，“筛选”所需进行的关键字比较的次数至多为 $2(K-1)$ ;

- 2.对n个关键字，建成深度为 $h=\lceil \log_2 n \rceil + 1$ 的堆，所需进行的关键字比较的次数至多为 $4n$ ;

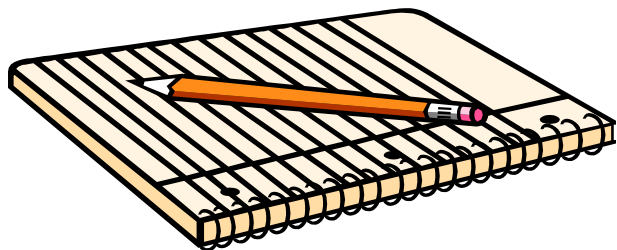
- 3.调整“堆顶” $n-1$ 次，总共进行的关键字比较次数不超过

$$2(\lfloor \log_2(n-1) \rfloor + \lfloor \log_2(n-2) \rfloor + \dots + \lfloor \log_2 2 \rfloor) < 2n \lfloor \log_2 n \rfloor$$

- 可以证明，堆排序的时间复杂度为 $O(n \log n)$ 。和选择排序相同，无论待排序列中的记录是正序还是逆序排列，都不会使堆排序处于“最好”或“最坏”的状态。



# 本章课程内容（第十章 内部排序）



- 10.1 概述
- 10.2 插入排序
- 10.3 快速排序
- 10.4 选择排序
- 10.5 归并排序
- 10.6 基数排序
- 10.7 各种内部排序方法的比较讨论



# 归并排序

- 归并排序的基本操作是将两个或两个以上的记录有序序列归并为一个有序序列。最简单的情况是，只含一个记录的序列显然是个有序序列，经过“逐趟归并”使整个序列中的有序子序列的长度逐趟增大，直至整个记录序列为有序序列止。
- 2-路归并排序的基本操作是将两个相邻的有序子序列“归并”为一个有序序列，如下侧所示。这个操作对顺序表而言是极其容易实现的，只要依关键字从小到大进行“复制”即可。





# 归并排序

## ● 算法10.14

```
void Merge (RcdType SR[], RcdType TR[], int i, int m, int n)
{    // 将有序的SR[i..m]和SR[m+1..n]归并为有序的TR[i..n]
    for (j=m+1, k=i; i<=m && j<=n; ++k)
    {    // 将SR中记录按关键字从小到大地复制到TR中
        if (SR[i].key<=SR[j].key) TR[k] = SR[i++];
        else TR[k] = SR[j++];
    }
    if (i<=m) TR[k..n] = SR[i..m];    // 将剩余的 SR[i..m] 复制到TR
    if (j<=n) TR[k..n] = SR[j..n];    // 将剩余的 SR[j..n] 复制到TR
} // Merge
```

- 因此，**2-路归并排序**首先要得到这两个子序列，然后进行一次复制归并操作。



# 归并排序

- 如果记录无序序列 $R[s..t]$ 的两部分 $R[s..(s+t)/2]$ 和 $R[(s+t)/2+1..t]$ 分别按关键字有序，则利用上述归并算法很容易将它们归并成整个记录序列是一个有序序列，由此，应该先分别对这两部分进行2-路归并排序。
- 执行2-路归并排序的一个例子。[演示](#)（10-5-2）



# 归并排序

## ● 算法 10.15

```
void Msort ( RcdType SR[], RcdType &TR1[], int s, int t )
{ // 对SR[s..t]进行归并排序, 排序后的记录存入TR1[s..t]
    if (s==t) TR1[s] = SR[s];
    else {
        m = (s+t)/2; // 将 SR[s..t] 平分为 SR[s..m] 和 SR[m+1..t]
        Msort (SR,TR2,s,m); // 递归地将 SR[s..m] 归并为有序的 TR2[s..m]
        Msort (SR,TR2,m+1, t); // 递归地将SR[m+1..t]归并为有序的TR2[m+1..t]
        Merge (TR2,TR1,s,m,t); // 将TR2[s..m]和TR2[m+1..t] 归并到 TR1[s..t]
    } // else
} // Msort
```



# 归并排序

- 算法10.16

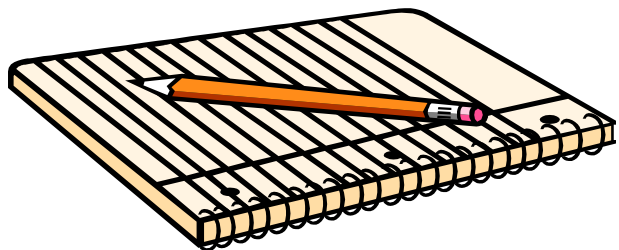
```
void MergeSort (SqList &L)
{ // 对顺序表L作2-路归并排序
    MSort(L.r, L.r, 1, L.length);
} // MergeSort
```

- 简单分析2-路归并排序的时间复杂度：对一个长度为  $n$  的记录序列需进行  $\lceil \log_2 n \rceil$  趟的归并，而每一趟需进行关键字比较和记录移动的时间和  $n$  成正比。
- 因此，2-路归并排序的时间复杂度为  $O(n \log n)$ 。





# 本章课程内容（第十章 内部排序）



- 10.1 概述
- 10.2 插入排序
- 10.3 快速排序
- 10.4 选择排序
- 10.5 归并排序
- 10.6 基数排序
- 10.7 各种内部排序方法的比较讨论



# 基数排序

---

- 借助“多关键字排序”的思想，实现“单关键字排序”的算法。
  - ➔ 多关键字排序
  - ➔ 链式基数排序



# 基数排序

## ●一、多关键字的排序

假设含有  $n$  个记录的序列为:  $\{R_1, R_2, \dots, R_n\}$ , 每个记录  $R_i$  中含有  $d$  个关键字  $(K_i^0, K_i^1, \dots, K_i^{d-1})$  则称该记录序列对关键字有序是指: 对于序列中任意两个记录  $R_i$  和  $R_j$  ( $1 \leq i < j \leq n$ ) 都满足下列有序关系:

$$(K_i^0, K_i^1, \dots, K_i^{d-1}) < (K_j^0, K_j^1, \dots, K_j^{d-1})$$

其中  $K^0$  被称作**最主位关键字**,  $K^{d-1}$  被称作**最次位关键字**。

→ 通常实现多关键字排序可以有两种策略: 一是**最高位优先(MSD法)**, 另一是**最低位优先(LSD法)**。



# 基数排序

## ●最高位优先的做法是：





- 先对最主位关键字 $K^0$ 进行排序，得到若干子序列，其中每个子序列中的记录都含相同的 $K^0$ 值
- 之后分别就每个子序列对关键字 $K^1$ 进行排序，致使 $K^1$ 值相同的记录构成长度更短的子序列
- 依次重复，直至就当前得到的各个子序列对 $K^{d-2}$ 进行排序之后得到的每个子序列中都具有相同的关键字  $(K_i^0, K_i^1, \dots, K_i^{d-2})$
- 分别对这些子序列按 $K^{d-1}$ 从小到大进行排序，最后由这些子序列依次相连所得序列便是排序的最后结果，即对关键字  $(K_i^0, K_i^1, \dots, K_i^{d-1})$ 有序



## ●最低位优先的做法是，在继续对“前一位”排序时不需要将当前所得对其后一位有序的序列分割成若干子序列进行，而是整个序列依次对 $K^{d-1}$ 、对 $K^{d-2}$ 直至对 $K^0$ 进行排序即可。



# 基数排序

● 假如你手中有一副扑克牌，若要将它们排列成一个有序序列，怎么做？

→ **方法一**：先按不同“花色” ( <  <  < ) 分成有次序的四堆，然后分别对每一堆按“面值”大小 ( $2 < 3 < \dots < A$ ) 排列有序。若将“花色”视作  $K^0$ ，将“面值”视作  $K^1$ ，这种整理方法即为“**MSD**”的作法。

→ **方法二**：先将手中的牌按“面值”分成13堆，然后将这13堆牌从大到小**收集**在一起（“A”在“K”之上，“K”在“Q”之上，……，最下面的是4张“2”），再重新按不同“花色”分成4堆，最后将这4堆牌按自小至大的次序**收集**在一起（ 在最上面， 在最下面），此时你手中的牌已是从上到下有序的了。显然，这种整理方法即为“**LSD**”的作法。

➤ 可见按最次位优先进行排序时，对每一位关键字的排序不一定要用前几节所述的内部排序方法进行，而是用先按类“分配”再依次“收集”的方法进行。



# 基数排序

## ●LSD举例

学生记录含三个关键字：系别、班号、班内序号，其中系别为最主位关键字，LSD排序过程如下：

无序序列	3,2,30	1,2,15	3,1,20	2,3,18	2,1,20
对 $K^2$ 排序	1,2,15	2,3,18	3,1,20	2,1,20	3,2,30
对 $K^1$ 排序	3,1,20	2,1,20	1,2,15	3,2,30	2,3,18
对 $K^0$ 排序	1,2,15	2,1,20	2,3,18	3,1,20	3,2,30



# 链式基数排序

## ●二、链式基数排序

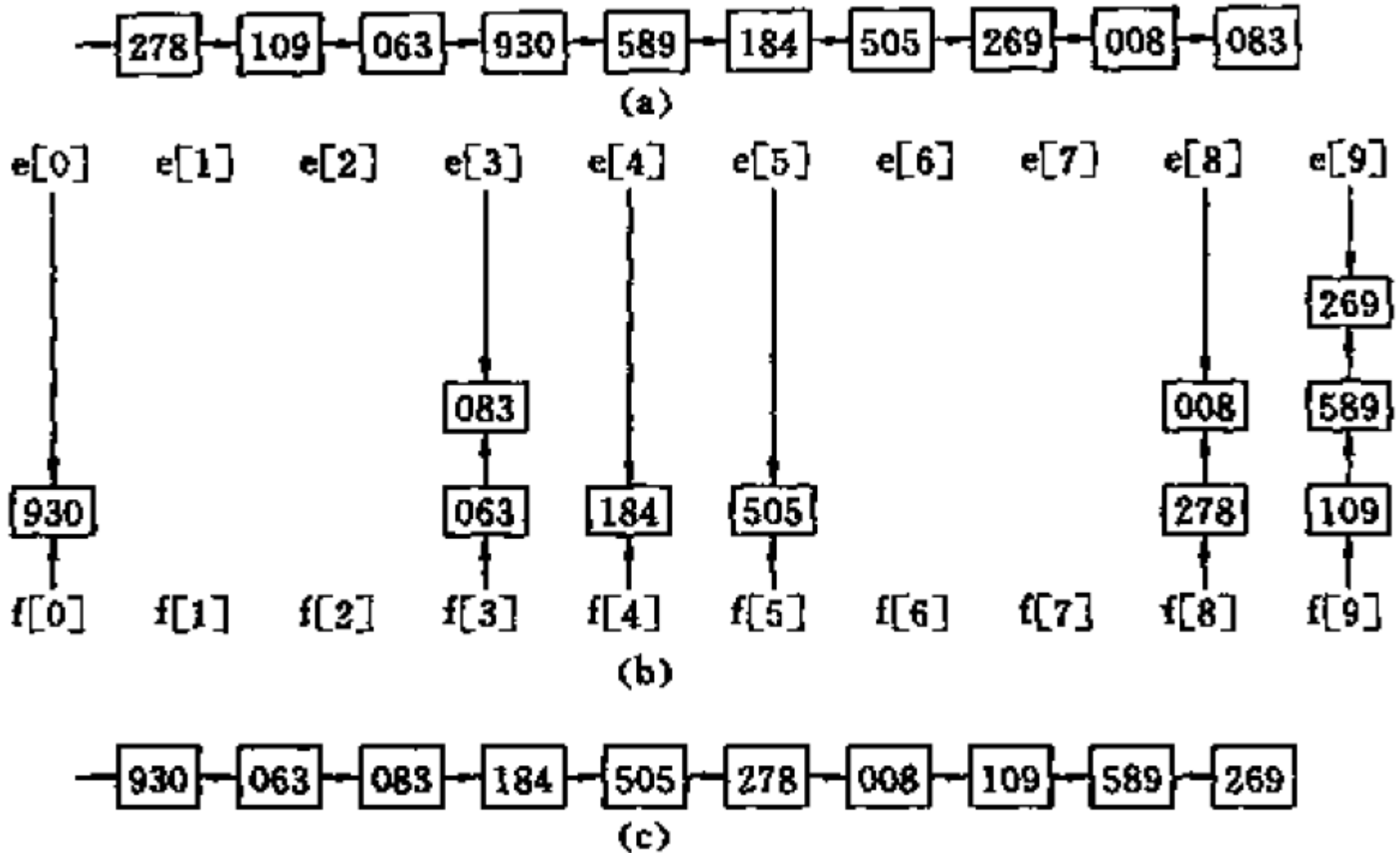
基数排序是借助 **“分配”** 和 **“收集”** 两种操作对单逻辑关键字进行排序的一种内部排序方法。

假如多关键字的记录序列中，每个关键字的取值范围相同，则按LSD法进行排序时，可以采用“分配-收集”的方法，其好处是不需要进行关键字之间的比较。

对于数字型或字符型的单关键字，可以看成是由多个数位或多个字符构成的多关键字，此时可以采用这种“分配-收集”的方法进行排序。



## 链式基数排序

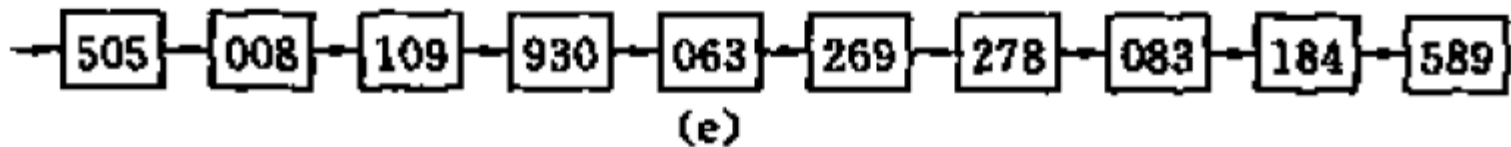
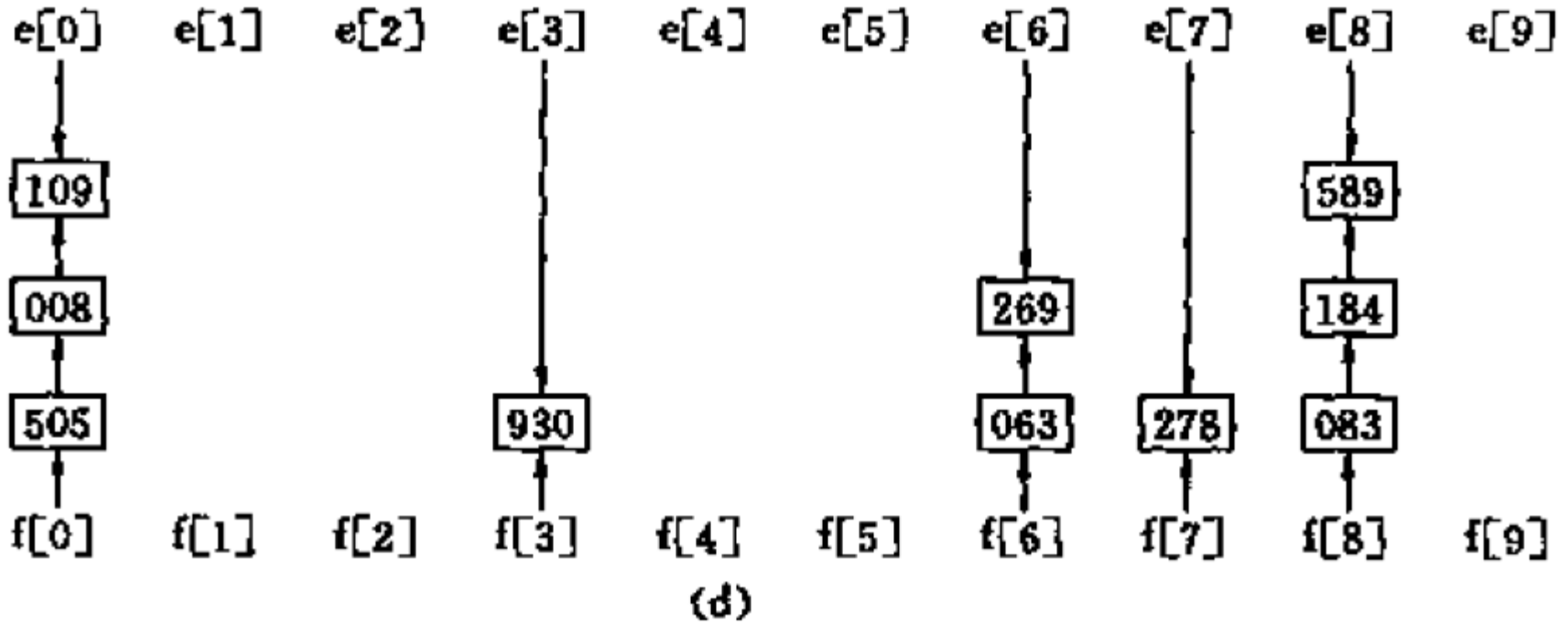


(a) 初始状态 (b) 第一趟分配之后 (c) 第一趟收集之后





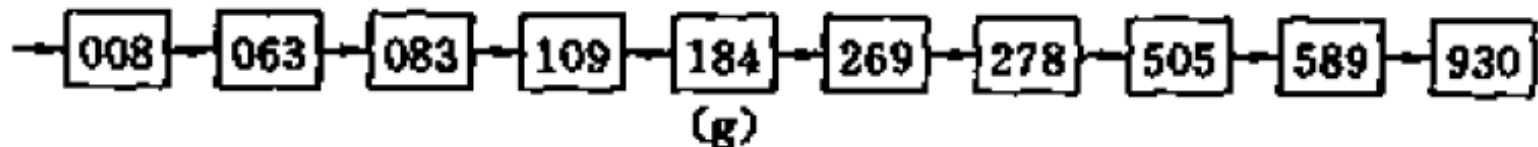
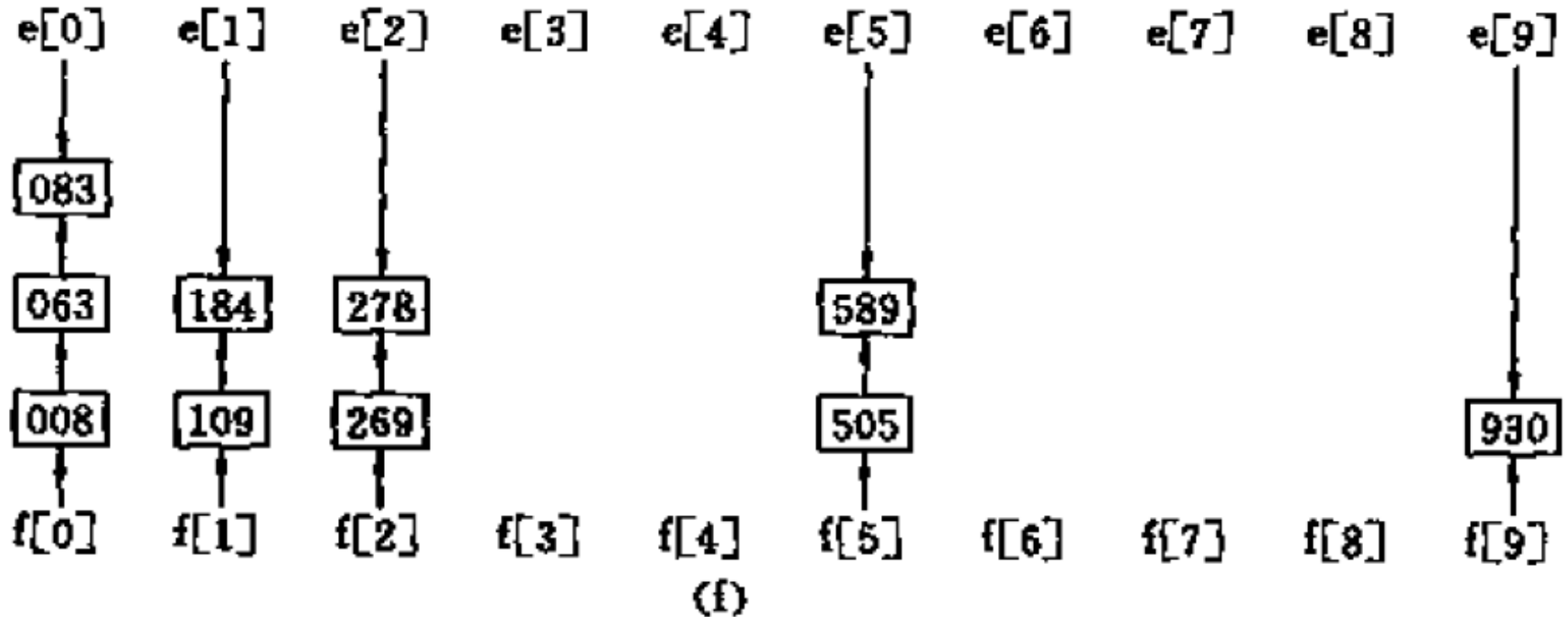
## 链式基数排序



(d) 第二趟分配之后 (e) 第二趟收集之后



## 链式基数排序



(f) 第三趟分配之后 (g) 第三趟收集之后



## 链式基数排序

1. 待排序记录以指针相链，构成一个链表；
2. 分配时，按当前“关键字位”取值，将记录**分配到不同的链队列中，每个队列中记录的“关键字位”相同**；
3. 收集时，按当前关键字位取值**从小到大将各队列首尾相链成一个链表**；
4. 对每个关键字位均重复上面2、3两步。

注意：

- 1、分配和收集的实际操作仅为修改链表中的指针和设置队列的头、尾指针；
- 2、为查找使用，该链表尚需应用算法Arrange调整为有序表。

链式基数排序的过程如动画所示。[演示](#)（10-6-3）



# 基数排序

- 重新定义记录和顺序表类型：

```
#define MAX_NUM_OF_KEY 8      // 关键字项数的最大值，暂定为8
#define RADIX 10              // 关键字基数，此为十进制整数的基数
#define MAX_SPACE 10000

typedef struct {
    KeysType keys[MAX_NUM_OF_KEY]; // 关键字
    InfoType otheritems; // 其它数据项
    int next;
} SLCell; // 静态链表的结点类型

typedef struct {
    SLCell r[MAX_SPACE]; // 静态链表的可利用空间，r[0]为头结点
    int keynum; // 记录的当前关键字个数
    int recnum; // 静态链表的当前长度
} SLList; // 静态链表类型

typedef int ArrType[RADIX]; // 指针数组类型
```



# 基数排序

## ● 算法10.19

**void** RadixSort(SLList &L)

{// L是采用静态链表表示的顺序表。对L作基数排序，

// 使得L成为按关键字自小到大的有序静态链表，L.r[0]为头结点。

for (i=0; i<L.recnum; ++i) L.r[i].next = i+1;

L.r[L.recnum].next = 0; // 将L改造为静态链表

for ( i=0; i<L.keynum; ++i) { // 按最低位优先依次对各关键字进行

//分配和收集

Distribute(L.r, i, f, e); // 第 i 趟分配

Collect(L.r, i, f, e); // 第 i 趟收集

}// for

} // LRadixSort



# 基数排序

## ● 算法10.17

**void** Distribute (SLCell &r, **int** i, **ArrType** &f, **ArrType** &e)

{// 静态链表L的r域中记录已按(keys[0],..., keys[i-1]) 有序，本算法按第i个关键字keys[i] 建立 **RADIX** 个子表，使同一子表中记录的keys[i] 相同。

f[0..**RADIX**-1] 和 e[0..**RADIX**-1] 分别指向各子表中第一个和最后一个记录

**for** (j=0; j<Radix; ++j) f[j] = 0; // 各子表初始化为空表

**for** (p=r[0].next; p; p=r[p].next) {

j = ord(r[p].keys[i]); //ord 将记录中的关键字keys[i]映射到[0..**RADIX**-1]中

**if** ( !f[j] ) f[j] = p;

**else** r[e[j]].next = p;

e[j] = p;

// 将 p 所指的结点插入相应子表

}// **for**

} // Distribute



# 基数排序

## ● 算法10.18

**void** Collect (SLCell &r, **int** i, ArrType f, ArrType e)

{// 本算法按 **keys[i]** 自小至大地将 **f[0..RADIX-1]** 所指各子表

// 依次链接成一个链表, **e[0..RADIX-1]** 为各子表的尾指针

**for** ( j=0; !f[j]; j=succ(j)); // 找第一个非空子表, **succ** 为求后继函数

r[0].next = f[j]; t = e[j]; // r[0].next指向第一个非空子表中第一个结点

**while** ( j<RADIX ) {

**for** (j=succ(j); j<RADIX-1 && !f[j]; j=succ(j) ); // 找下一个非空子表

**if** ( f[j] ) { r[t].next = f[j]; t = e[j]; } // 链接两个非空子表

}// while

r[t].next = 0; // t 指向最后一个非空子表中的最后一个结点

} // Collect



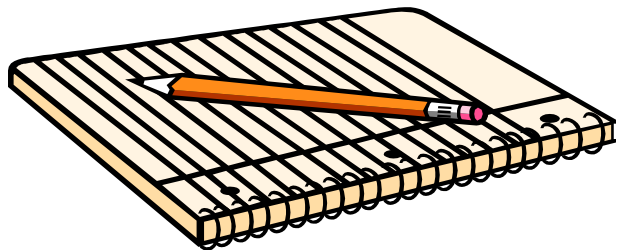
# 基数排序

- 分析**链式基数排序**的**时间复杂度**：假设  $n$  为记录个数， $rd$  为基数值， $d$  为构成逻辑关键字的关键字位数。
  - 一趟“分配”的时间复杂度是  $O(n)$
  - 一趟“收集”复杂度是  $O(rd)$
  - 因此一趟分配和收集的时间复杂度为  $O(n+rd)$
  - 对每一位关键字进行一趟分配和收集，因此总的时间复杂度为  $O(d(n+rd))$
  - 一般情况下，相比  $n$  而言， $rd$  要小得多，因此可简写为  **$O(dn)$** 。
  - 换句话说，当待排序序列中的记录数量很大，而逻辑关键字的“位数”较小，此时用基数排序法进行排序将比快速排序的效率更高。





# 本章课程内容（第十章 内部排序）



- 10.1 概述
- 10.2 插入排序
- 10.3 快速排序
- 10.4 选择排序
- 10.5 归并排序
- 10.6 基数排序
- 10.7 各种内部排序方法的比较讨论



# 各种内部排序方法的比较讨论

## ● 一、时间性能

→ 1. 按平均的时间性能来分，有三类排序方法：

- 时间复杂度为 $O(n \log n)$ 的方法有：快速排序、堆排序和归并排序，其中**快速排序目前被认为是最快的一种排序方法**，后两者与之比较，在  $n$  值较大的情况下，归并排序较堆排序更快；
- 时间复杂度为 $O(n^2)$ 的有：直接插入排序、起泡排序和简单选择排序，其中以插入排序为最常用，特别是对于已按关键字基本有序排列的记录序列尤为如此，选择排序过程中记录移动次数最少；
- 时间复杂度为 $O(n)$ 的排序方法只有基数排序一种。



## 各种内部排序方法的比较讨论

- 2. 当待排记录序列按关键字顺序有序时，直接插入排序和起泡排序能达到 $O(n)$ 的时间复杂度；而对于快速排序而言，这是最不好的情况，此时的时间性能蜕化为 $O(n^2)$ ，因此应尽量避免。
- 3. 简单选择排序、堆排序和归并排序的时间性能不随记录序列中关键字的分布而改变。
- 4. 以上对排序的时间复杂度的讨论主要考虑排序过程中所需进行的关键字间的比较次数，当待排序记录中其它各数据项比关键字占有更大的数据量时，还应考虑到排序过程中移动记录的操作时间，有时这种操作的时间在整个排序过程中占的比例更大，从这个观点考虑，简单排序的三种排序方法中起泡排序效率最低。



# 各种内部排序方法的比较讨论

## ●二、空间性能

指的是排序过程中所需的辅助空间大小。

- ➔ 1. 所有的简单排序方法（包括：直接插入、起泡和简单选择排序）和堆排序的空间复杂度均为 $O(1)$ 。
- ➔ 2. 快速排序为 $O(\log n)$ ，为递归程序执行过程中栈所需的辅助空间。
- ➔ 3. 归并排序和基数排序所需辅助空间最多，其空间复杂度为 $O(n)$ 。
- ➔ 4. 链式基数排序需附设队列首尾指针，空间复杂度 $O(rd+n)$



# 各种内部排序方法的比较讨论

## ●三、排序方法的稳定性能

- 1. 稳定的排序：两个关键字相等的记录在经过排序之后，不改变它们在排序之前在序列中的相对位置。

(56, 34, **47**, 23, 66, 18, 82, **47**)

(18, 23, 34, **47**, **47**, 56, 66, 82)

(18, 23, 34, **47**, **47**, 56, 66, 82)

- 2. 除**希尔排序、快速排序和堆排序是不稳定的排序方法**外，本章讨论的其它排序方法都是稳定的。
- 3. “稳定性”是由方法本身决定的。一般来说，排序过程中所进行的比较操作和交换数据仅发生在相邻的记录之间，没有大步距的数据调整时，则排序方法是稳定的。



## 各种内部排序方法的比较讨论

- 综合上述，可得下面列表所示结果。

排序方法	平均时间	最坏情况	最好情况	辅助空间	稳定性
直接插入排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	√
起泡排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	√
简单选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	√
快速排序	$O(n \log n)$	$O(n^2)$	$O(n \log n)$	$O(\log n)$	×
堆排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	×
2-路归并排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	√
基数排序	$O(dn)$	$O(dn)$	$O(dn)$	$O(n)$	√



## 各种内部排序方法的比较

排序方法	平均时间	最坏情况	辅助存储
简单排序	$O(n^2)$	$O(n^2)$	$O(1)$
快速排序	$O(n \log n)$	$O(n^2)$	$O(\log n)$
堆排序	$O(n \log n)$	$O(n \log n)$	$O(1)$
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n)$
基数排序	$O(d(n + rd))$	$O(d(n + rd))$	$O(rd)$





## 本章小结

- 本章主要讨论各种内部排序的方法。学习本章的目的是了解各种排序方法的原理以及各自的优缺点，以便在编制软件时能按照情况所需合理选用。
- 一般来说，在选择排序方法时，可有下列几种选择：
  - ➔ 若待排序的记录个数 $n$ 值较小（例如 $n < 30$ ），则可选用插入排序法，但若记录所含数据项较多，所占存储量大时，应选用选择排序法。
  - ➔ 反之，若待排序的记录个数 $n$ 值较大时，应选用快速排序法。但若待排序记录关键字有“有序”倾向时，就慎用快速排序，而宁可选用堆排序或归并排序，而后两者的最大差别是所需辅助空间不等。





## 本章小结

---

- 快速排序和归并排序在 $n$ 值较小时的性能不及直接插入排序，因此在实际应用时，可将它们和插入排序“混合”使用。如在快速排序划分子区间的长度小于某值时，转而调用直接插入排序；或者对待排记录序列先逐段进行直接插入排序，然后再利用“归并操作”进行两两归并直至整个序列有序为止。



## 本章小结

---

- 选择排序、归并排序和基数排序等的定义和各种排序方法的特点、时间复杂度的分析方法
- 理解排序方法“稳定”或“不稳定”的含义，弄清楚在什么情况下要求应用的排序方法必须是稳定的
- 通过各种内部排序方法的比较讨论，领会各种方法的特点



## 本章知识点与重点

### ● 知识点

直接插入排序、折半插入排序、表插入排序、希尔排序、起泡排序、快速排序、简单选择排序、堆排序、2-路归并排序、基数排序、排序方法的综合比较

### ● 重点和难点

希尔排序、快速排序、堆排序和归并排序等高效方法是本章的学习重点和难点