

第十章 内部排序

10.1 排序

3 10 5 78 36

3 5 10 36 78

排序:

设 n 个记录的序列为 $\{ R_1, R_2, R_3, \dots, R_n \}$

其相应的关键字序列为 $\{ K_1, K_2, K_3, \dots, K_n \}$

若规定 $1, 2, 3, \dots, n$ 的一个排列 $p_1, p_2, p_3, \dots, p_n$,
使得相应的关键字满足如下非递减关系:

$$K_{p_1} \leq K_{p_2} \leq K_{p_3} \leq \dots \leq K_{p_n}$$

则原序列变为一个按关键字有序的序列:

$$\{ R_{p_1}, R_{p_2}, R_{p_3}, \dots, R_{p_n} \}$$

此操作过程称为**排序**。

稳定排序 与 不稳定排序

假设 $K_i = K_j$ ，且排序前序列中 R_i 领先于 R_j ；

若在排序后的序列中 R_i 仍领先于 R_j ，则称排序方法是稳定的。

若在排序后的序列中 R_j 领先于 R_i ，则称排序方法是不稳定的。

例，序列 3 15 8 8 6 9

若排序后得 3 6 8 8 9 15 稳定的

若排序后得 3 6 8 8 9 15 不稳定的

内部排序 vs. 外部排序

内部排序: 指的是待排序记录存放在计算机**随机存储器(内存)**中进行的排序过程。

外部排序: 指的是待排序记录的数量很大, 以致内存一次不能容纳全部记录, 在排序过程中尚需对**外存**进行访问的排序过程。

内部排序

按照排序过程中所依据的原则的不同可以分为:

- 插入排序(希尔排序)
- 交换排序(快速排序)
- 选择排序(堆排序)
- 归并排序
- 基数排序

1. 插入类

将无序子序列中的一个或几个记录“插入”到有序序列中，从而增加记录的有序子序列的长度。

2. 交换类

通过“**交换**”无序序列中的记录从而得到其中关键字最小或最大的记录，并将它加入到有序子序列中，以此方法增加记录的有序子序列的长度。

3. 选择类

从记录的无序子序列中 “选择”
关键字最小或最大的记录，并将它
加入到有序子序列中，以此方法增
加记录的有序子序列的长度。

4. 归并类



通过 “**归并**” 两个或两个以上的记录有序子序列，逐步增加记录有序序列的长度。

5. 其它方法



待排记录的数据类型定义:

```
#define MAXSIZE 1000    // 待排顺序表最大长度

typedef int KeyType;    // 关键字类型为整数类型

typedef struct {
    KeyType    key;      // 关键字项
    InfoType   otherinfo; // 其它数据项
} RecType;            // 记录类型

typedef struct {
    RecType    r[MAXSIZE+1]; // r[0]闲置
    int        length;        // 顺序表长度
} SqList;            // 顺序表类型
```

10.2 插入排序

10.2.1 直接插入排序

思想：利用有序表的插入操作进行排序

有序表的插入：将一个记录插入到已排好序的有序表中，从而得到一个新的有序表。

例，序列 13 27 38 65 76 97

插入 49

13 27 38 49 65 76 97

算法描述:

初始, 令第 **1** 个元素作为初始有序表;

依次插入第 **2, 3, ..., k** 个元素构造新的有序表;

直至最后一个元素;

例, 序列 **49** **38** **65** **97** **76** **13** **27**

 ↑ ↑ ↑ ↑ ↑ ↑

初始, $S = \{ 49 \}$;

 { **13** **27** **38** **49** **65** **76** **97** }

直接插入排序算法主要应用**比较**和**移动**两种操作。

```
void InsertSort ( SqList &L )  
{  
    // 对顺序表 L 作直接插入排序。  
    for ( i=2; i<=L.length; ++i )  
        if (L.r[i].key < L.r[i-1].key) {  
            L.r[0] = L.r[i];           // 复制为监视哨  
            for ( j=i-1; L.r[0].key < L.r[j].key; -- j )  
                L.r[j+1] = L.r[j];     // 记录后移  
            L.r[j+1] = L.r[0];         // 插入到正确位置  
        }  
} // InsertSort
```

10.2.2 希尔(shell)排序

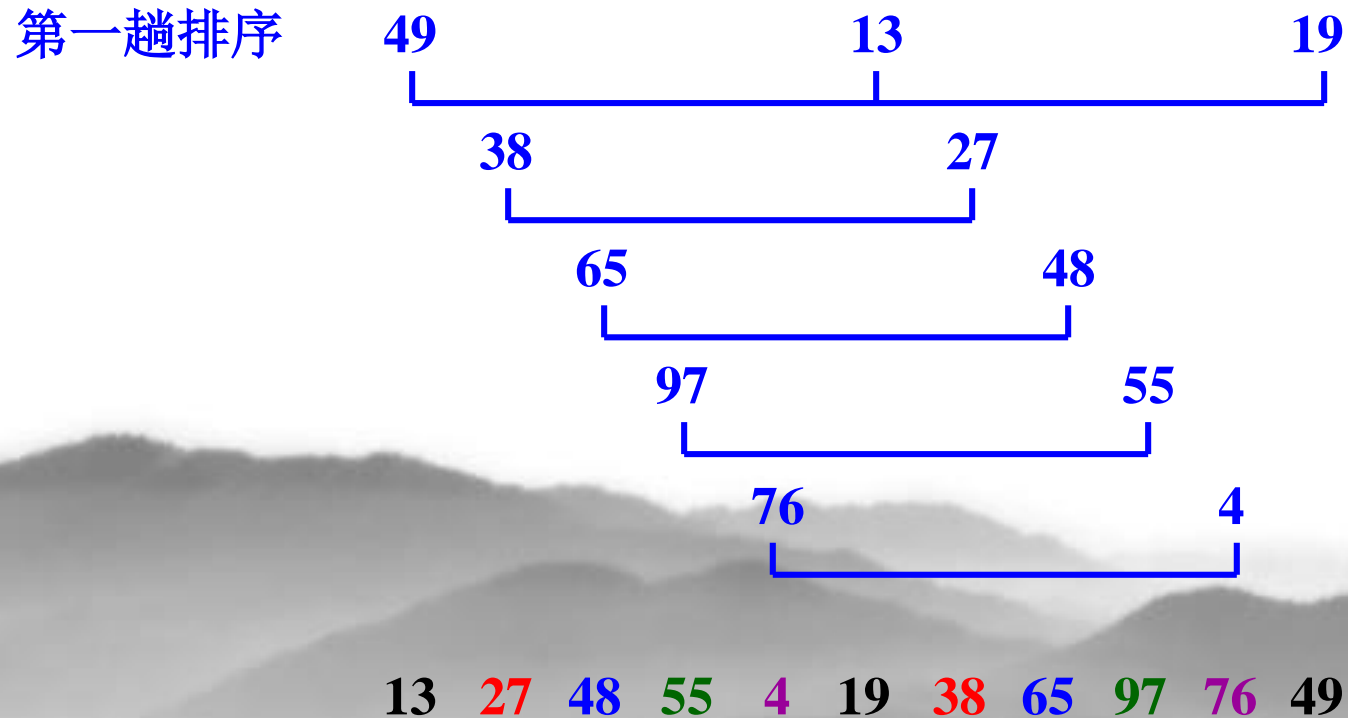
分析直接插入排序

1. 若待排序记录序列按关键字**基本有序**，则排序效率可大大提高；[移动较少]
2. 待排序记录总数越少，排序效率越高；

思想:

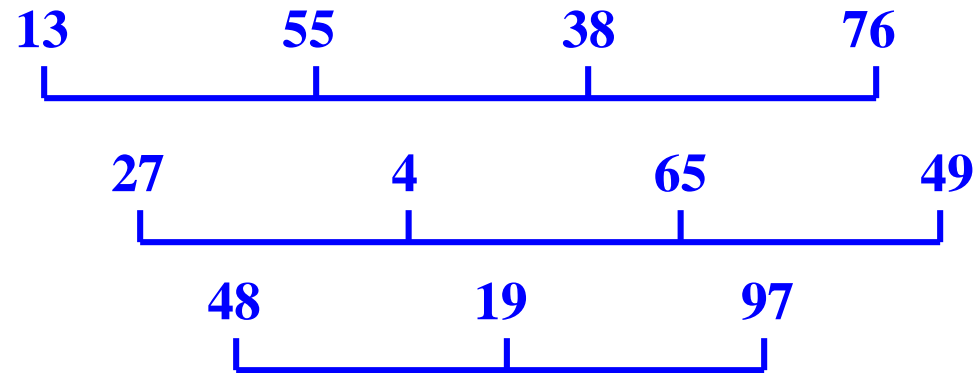
先将待排序记录序列分割成为若干子序列分别进行直接插入排序;
待整个序列中的记录基本有序后,再全体进行一次直接插入排序。

例, 序列 49 38 65 97 76 13 27 48 55 4 19



13 27 48 55 4 19 38 65 97 76 49

第二趟排序



13 4 19 38 27 48 55 49 97 76 65

第三趟排序

4 13 19 27 38 48 49 55 65 76 97

```
void ShellInsert ( SqList &L, int dk )
```

```
{
```

```
// 对顺序表 L 作一趟希尔插入排序。和直接插入排序的差别:
```

```
// 1. 前后记录位置的增量是dk, 而不是1;
```

```
// 2. r[0]只是暂存单元, 而不是哨兵。当j<=0时, 插入位置已找到。
```

```
for ( i=dk+1; i<=L.length; ++i )
```

```
if ( L.r[i].key< L.r[i-dk].key) {
```

```
    L.r[0] = L.r[i];           // 暂存在R[0]
```

```
    for (j=i-dk; j>0 && (L.r[0].key<L.r[j].key); j-=dk)
```

```
        L.r[j+dk] = L.r[j]; // 记录后移, 查找插入位置
```

```
    L.r[j+dk] = L.r[0];      // 插入
```

```
    } // if
```

```
} // ShellInsert
```



```
void ShellSort (SqList &L, int delta[], int t)
{ // 增量为delta[]的希尔排序, delta[t-1]==1
    for (k=0; k<t; ++t)
        ShellInsert(L, delta[k]);
        //一趟增量为delta[k]的插入排序
} // ShellSort
```

10.3 交换排序

10.3.1 冒泡排序

思想：通过不断比较相邻元素大小，进行交换来实现排序。

第一趟排序：

首先将第1个元素与第2个元素比较大小，若为逆序，则交换；

然后比较第2个元素与第3个元素的大小，若为逆序，则交换；

⋮

直至比较第 $n-1$ 个元素与第 n 个元素的大小，若为逆序，则交换；

结果：

关键字最大的记录被交换至**最后**一个元素位置上。

例，序列 49 38 76 13 27

38

49

13

27

76

第一趟排序后

最大值

38

13

27

49

第二趟排序后

次大值

13

27

38

第三趟排序后

13

27

第四趟排序后

```
void BubbleSort(Elem R[ ], int n) {
```

```
    i = n;
```

```
    while (i > 1) {
```

```
        lastExchangeIndex = 1;
```

```
        for (j = 1; j < i; j++)
```

```
            if (R[j+1].key < R[j].key) {
```

```
                Swap(R[j], R[j+1]);
```

```
                lastExchangeIndex = j; //交换的记录位置
```

```
            } //if
```

```
        i = lastExchangeIndex; // 本趟进行过交换的
```

```
    } // while
```

```
        // 最后一个记录的位置
```

```
} // BubbleSort
```

注意:

1. 起泡排序的结束条件为,
最后一趟没有进行“记录交换”。
2. 一般情况下, 每经过一趟“起泡”,
“ i 减一”, 但并不是每趟都如此。

例如:



```
for (j = 1; j < i; j++) if (R[j+1].key < R[j].key) ...
```

时间分析:

最好的情况（关键字在记录序列中顺序有序）：
只需进行一趟起泡

“比较” 的次数：

$n-1$

“移动” 的次数：

0

最坏的情况（关键字在记录序列中逆序有序）：
需进行 $n-1$ 趟起泡

“比较” 的次数：

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

“移动” 的次数：

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

10.3.2 快速排序

冒泡排序的一种改进算法。

思想：

以首记录作为轴记录，从前、后双向扫描序列，通过交换，实现大值记录后移，小值记录前移，最终将轴记录安置在一个适当的位置。(小值记录在前、大值记录在后)

轴记录将原序列分割成两部分，依次对前后两部分重新设定轴记录，继而分别再进行快速排序。

直至整个序列有序。

例，序列 { 49 38 65 97 76 13 27 52 }

第一趟排序

13 38 27 49 76 97 65 52

 ↑ ↑
 j i

从前寻找大于轴记录的记录，从后寻找小于轴记录的记录；

交换大值记录与小值记录；

重复上述两步操作，直至 $i > j$ ；

交换轴记录和标识 j 指示的记录。

第一趟排序后 13 38 27 **49** 76 97 65 52

49 将序列分成两部分，分别进行新的快速排序；

第二趟排序 13 38 27 76 97 65 52

第二趟排序后 **13** 38 27 65 52 **76** **97**

第三趟排序 38 27 65 52

第三趟排序后 **27** **38** **52** **65**

最终有序序列为： 13 27 38 52 65 76 97

```
int Partition (RedType& R[], int low, int high)
{
    pivotkey = R[low].key;
    while (low<high) {
        while (low<high && R[high].key>=pivotkey)
            --high;
        R[low]↔R[high];
        while (low<high && R[low].key<=pivotkey)
            ++low;
        R[low]↔R[high];
    }
    return low;           // 返回枢轴所在位置
} // Partition
```

// 改进版

```
int Partition (RedType R[], int low, int high)
{
    R[0] = R[low]; pivotkey = R[low].key; // 枢轴
    while (low < high) {
        while (low < high && R[high].key >= pivotkey)
            -- high;    // 从右向左搜索
        R[low] = R[high];
        while (low < high && R[low].key <= pivotkey)
            ++ low;    // 从左向右搜索
        R[high] = R[low];
    }
    R[low] = R[0];
    return low;
} // Partition
```

```
void QSort (RedType & R[], int s, int t) {  
    // 对记录序列R[s..t]进行快速排序  
    if (s < t-1) { // 长度大于1  
        pivotloc = Partition(R, s, t); // 对 R[s..t] 进行一次划分  
        QSort(R, s, pivotloc-1); // 对低子序列递归排序,  
                                   // pivotloc是枢轴位置  
        QSort(R, pivotloc+1, t); // 对高子序列递归排序  
    }  
} // QSort  
  
void QuickSort(SqList & L)  
{ // 对顺序表进行快速排序, 调用QSort()  
    QSort(L.r, 1, L.length);  
} // QuickSort
```

10.4 选择排序

思想：每一趟都选出一个最大或最小的元素，并放在合适的位置。

- 简单选择排序
- 树形选择排序
- 堆排序

10.4.1 简单选择排序

思想：

第 1 趟选择：从 $1 \sim n$ 个记录中选择关键字最小的记录，并和第 1 个记录交换。

第 2 趟选择：从 $2 \sim n$ 个记录中选择关键字最小的记录，并和第 2 个记录交换。

⋮

第 $n-1$ 趟选择：从 $n-1 \sim n$ 个记录中选择关键字最小的记录，并和第 $n-1$ 个记录交换。

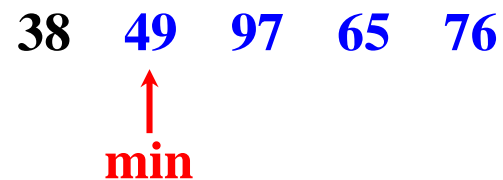
例，序列

49 38 97 65 76

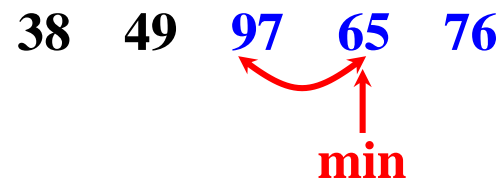
第 1 趟选择:

49 38 97 65 76
min


第 2 趟选择:

38 49 97 65 76
min

第 3 趟选择:

38 49 97 65 76
min

第 4 趟选择:

38 49 65 97 76
min

38 49 65 76 97

简单选择排序的算法描述如下：

```
void SelectSort (Elem R[], int n ) {  
    // 对记录序列R[1..n]作简单选择排序。  
    for (i=1; i<n; ++i) { // 选择第 i 小的记录  
        // 在 R[i..n] 中选择关键字最小的记录  
        j = SelectMinKey(R, i);  
        if (i != j) R[i]↔R[j]; // 与第 i 个记录交换  
    }  
} // SelectSort
```


时间性能分析

对 n 个记录进行简单选择排序，所需进行的
关键字间的比较次数 总计为：

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

移动记录的次数，最小为 0，最大为 $3(n-1)$ 。

选择排序的主要操作是进行关键字间的比较。

在 n 个关键字中选出最小值，至少需要 $n-1$ 次比较。

在剩余 $n-1$ 个关键字中选出最小值，至少需要 $n-2$ 次比较？

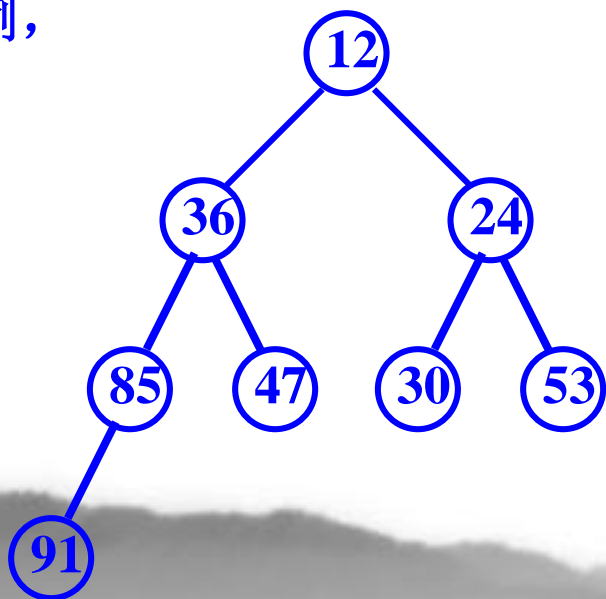
利用前 $n-1$ 次比较所得信息，可减少后面选择的比较次数。

体育比赛中的锦标赛

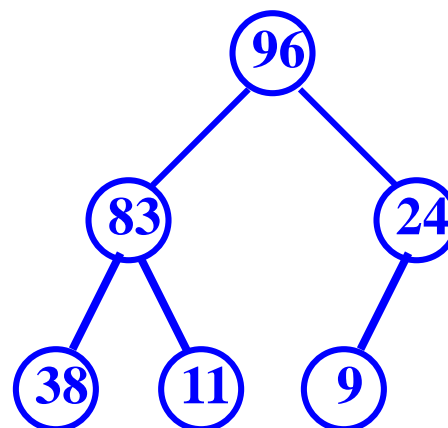
10.4.3 堆排序

堆：一棵完全二叉树，任一个非终端结点的值均小于等于(或大于等于)其左、右儿子结点的值。

例，



根结点为最小值



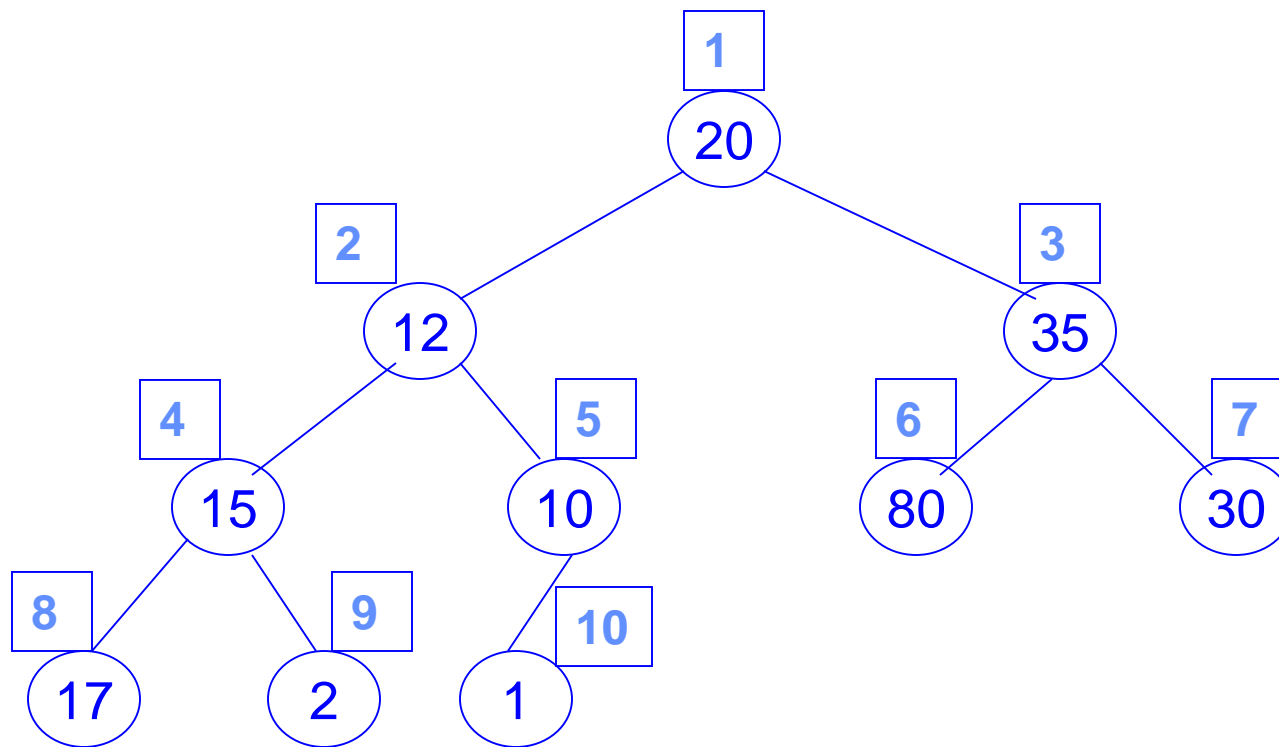
根结点为最大值

思想：

1. 将序列构造成一棵完全二叉树；
2. 把这棵普通的完全二叉树改造成堆，便可获取最小(或最大)值；
3. 输出最小(或大)值；
4. 删除根结点，继续改造剩余树成堆，便可获取次小(或大)值；
5. 输出次小(或大)值；
6. 重复改造，输出次次小(或大)值、次次次小(或大)值，直至所有结点均输出，便得到一个排序。

最大堆的初始化

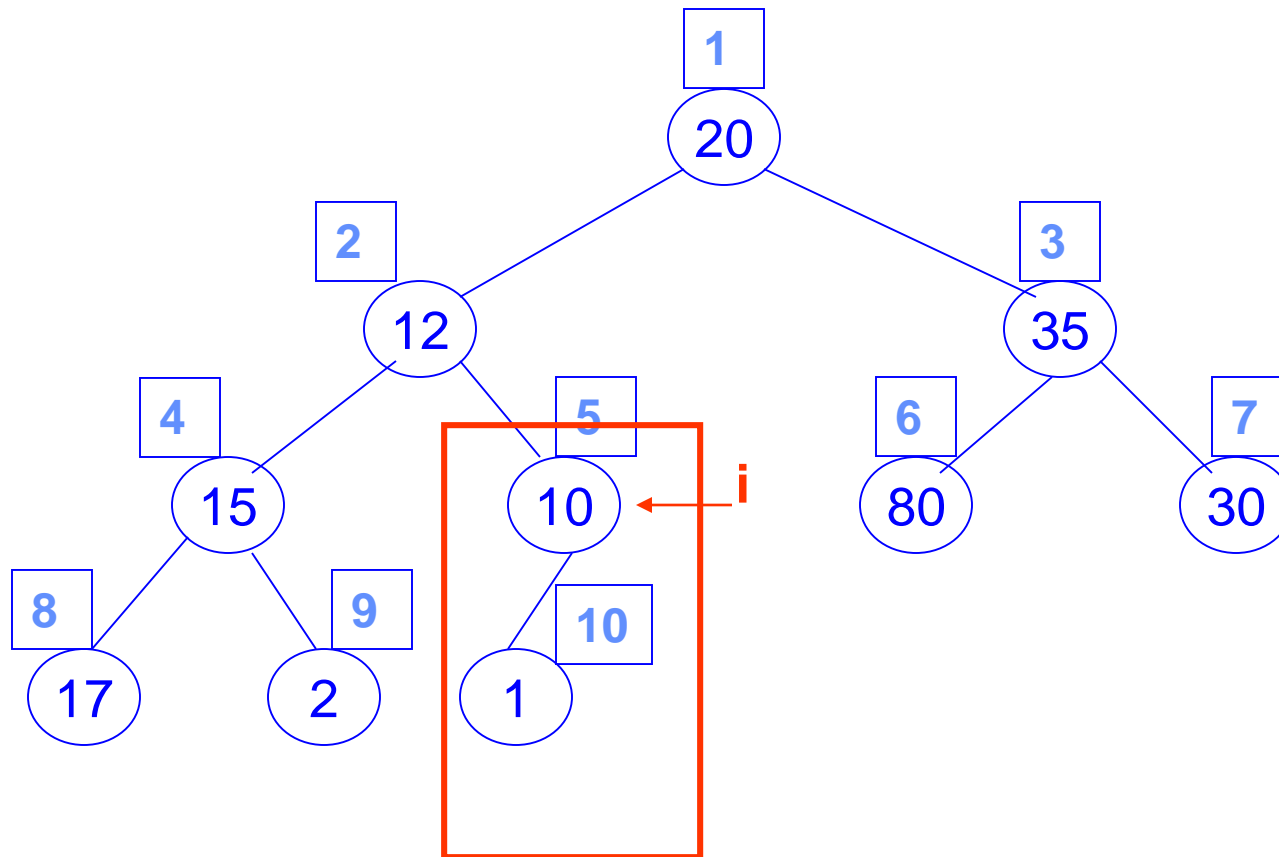
根据 $\text{int } a[10]=\{20,12,35,15,10,80,30,17,2,1\}$ 建立最大堆



算法：从第一个具有孩子的结点 i 开始 ($i=\lfloor n/2 \rfloor$)，如果以这个元素为根的子树已是最大堆，则不需调整，否则需调整子树使之成为堆。继续检查 $i-1$ ， $i-2$ 等结点为根的子树，直到检查整个二叉树的根结点(其位置为 1)。

最大堆的初始化step1

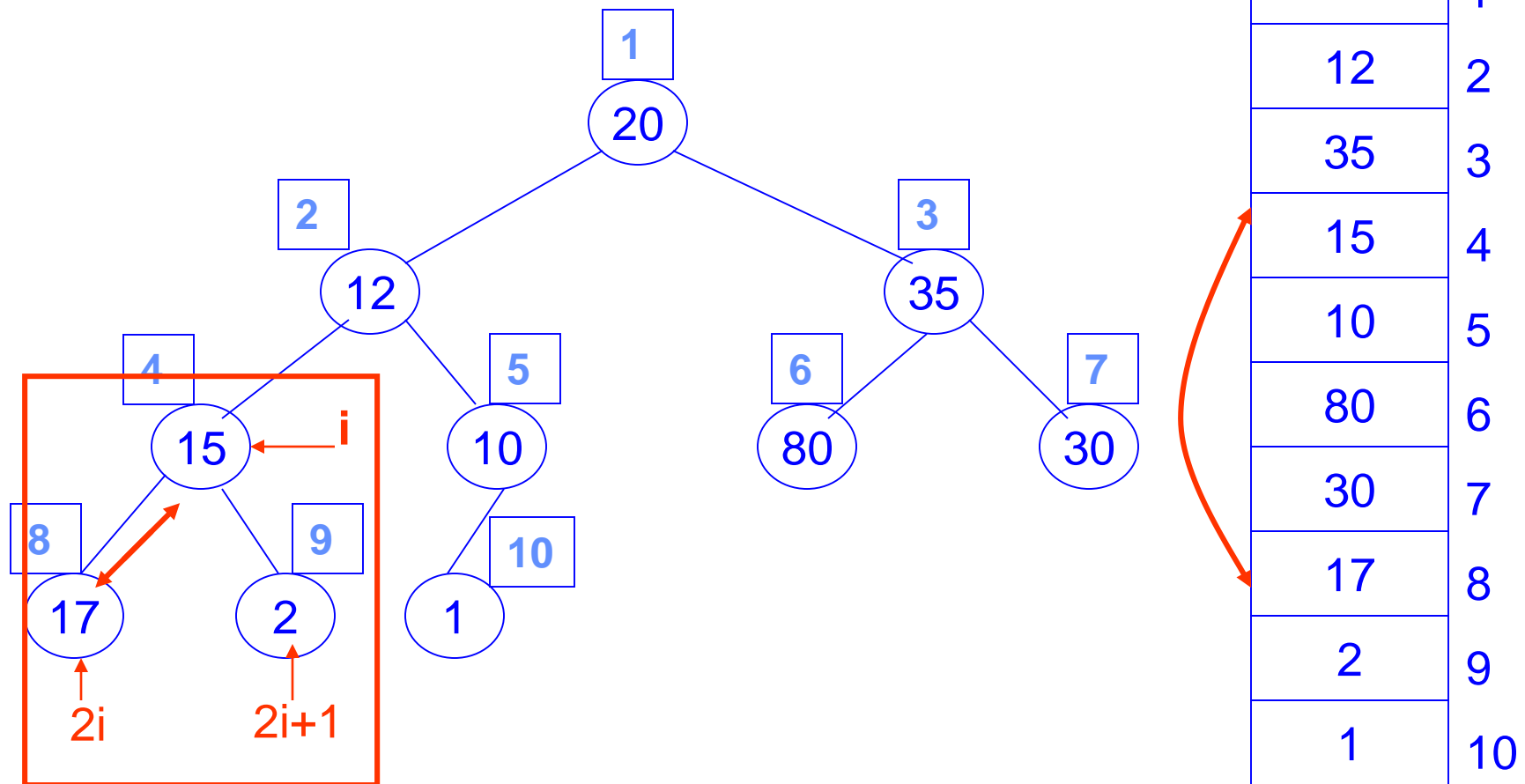
已知 $n=10$; $i=n/2=5$;



20	1
12	2
35	3
15	4
10	5
80	6
30	7
17	8
2	9
1	10

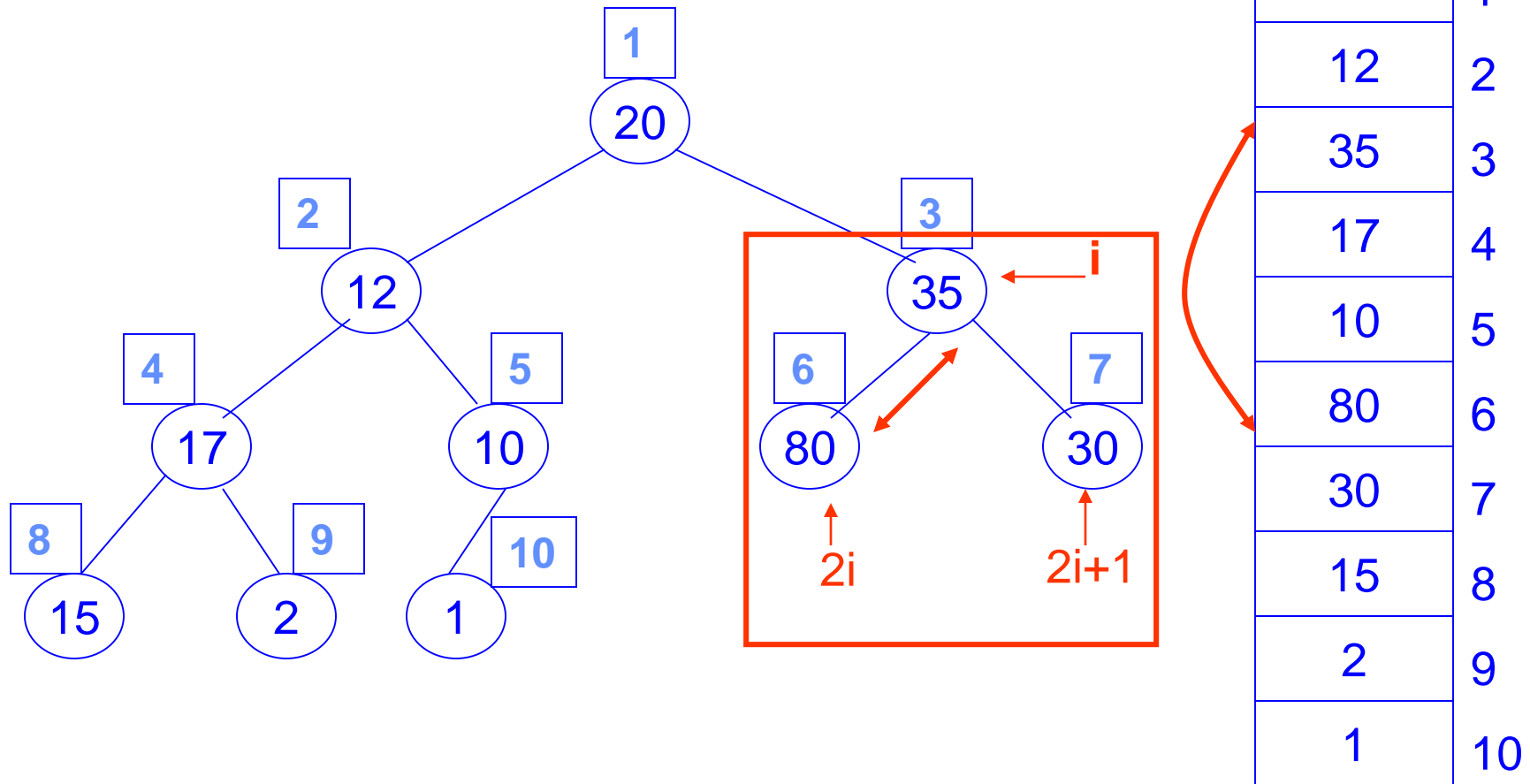
最大堆的初始化step2

$i=4$



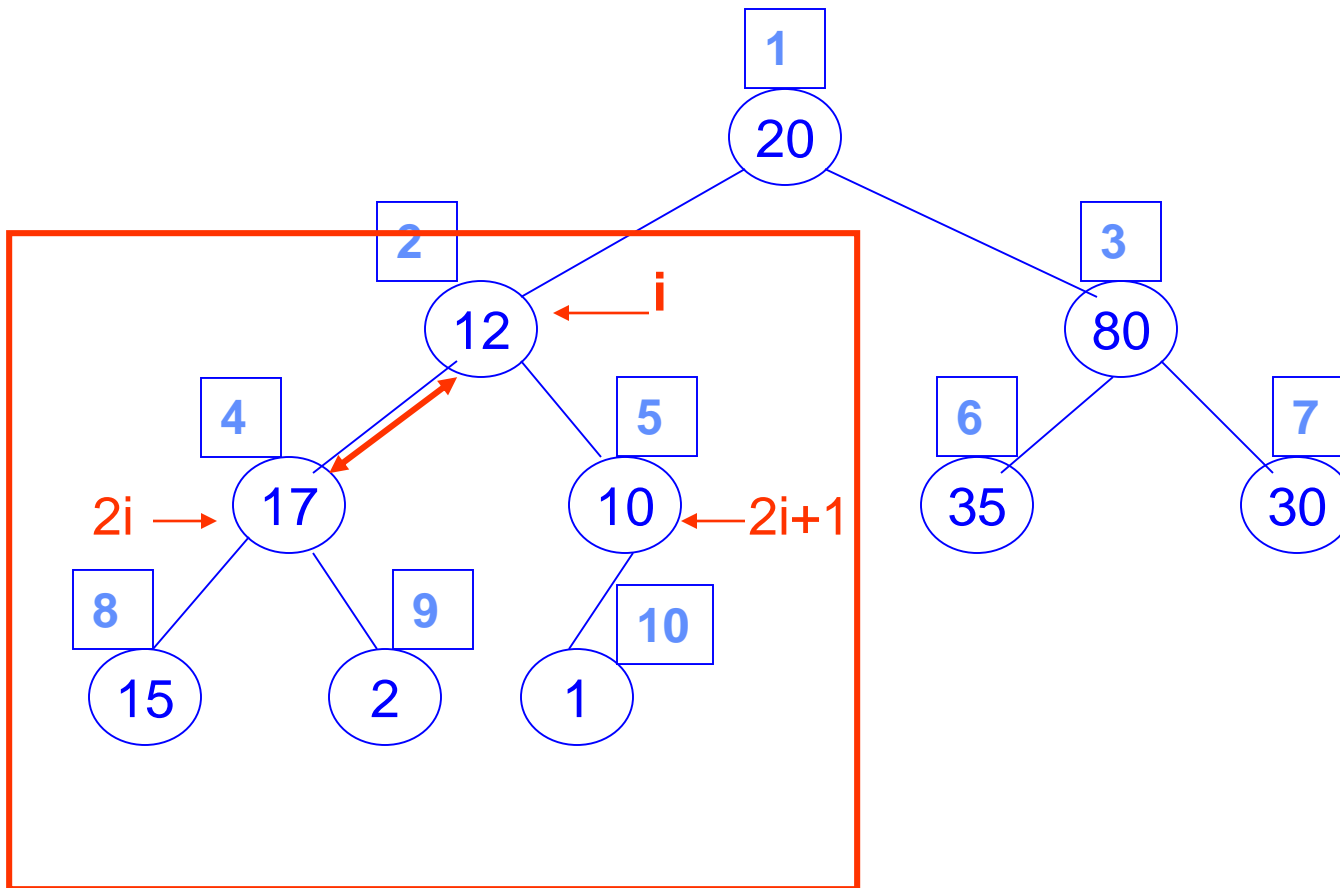
最大堆的初始化step3

$i=3$



最大堆的初始化step4_0

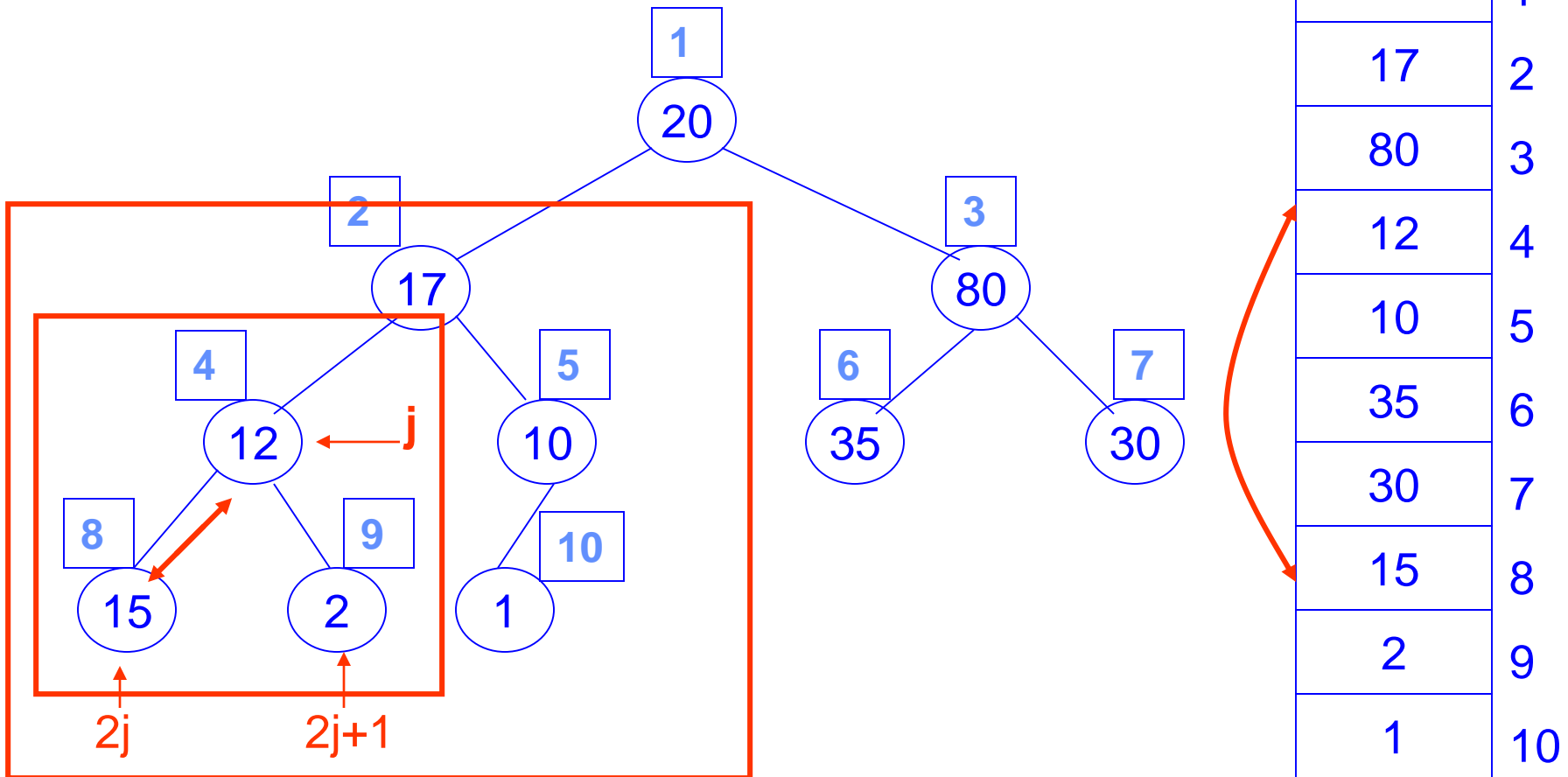
$i=2$



20	1
12	2
80	3
17	4
10	5
35	6
30	7
15	8
2	9
1	10

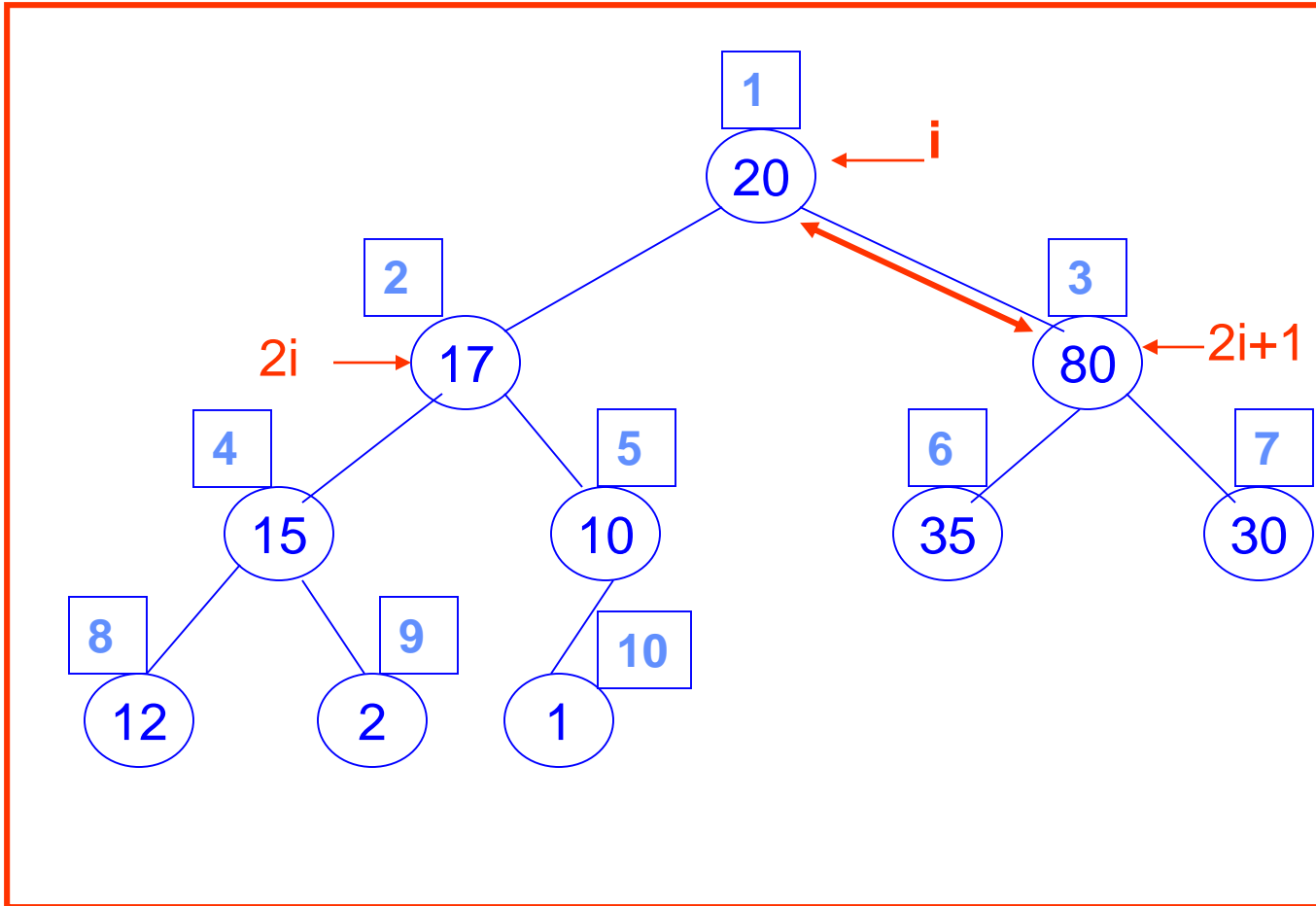
最大堆的初始化step4_1

$i=2, j=2i=4$



最大堆的初始化step5_0

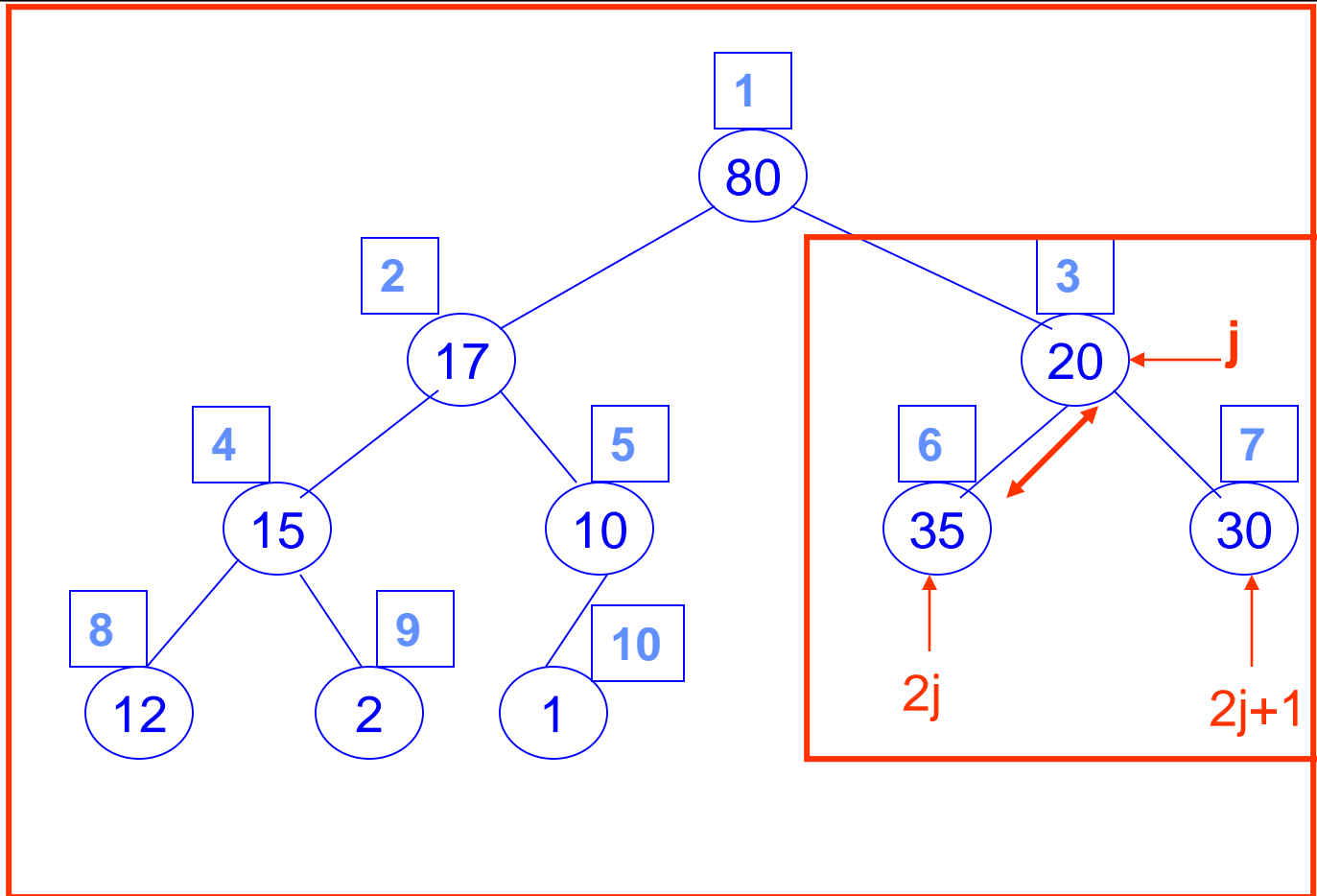
i=5



20	1
17	2
80	3
15	4
10	5
35	6
30	7
12	8
2	9
1	10

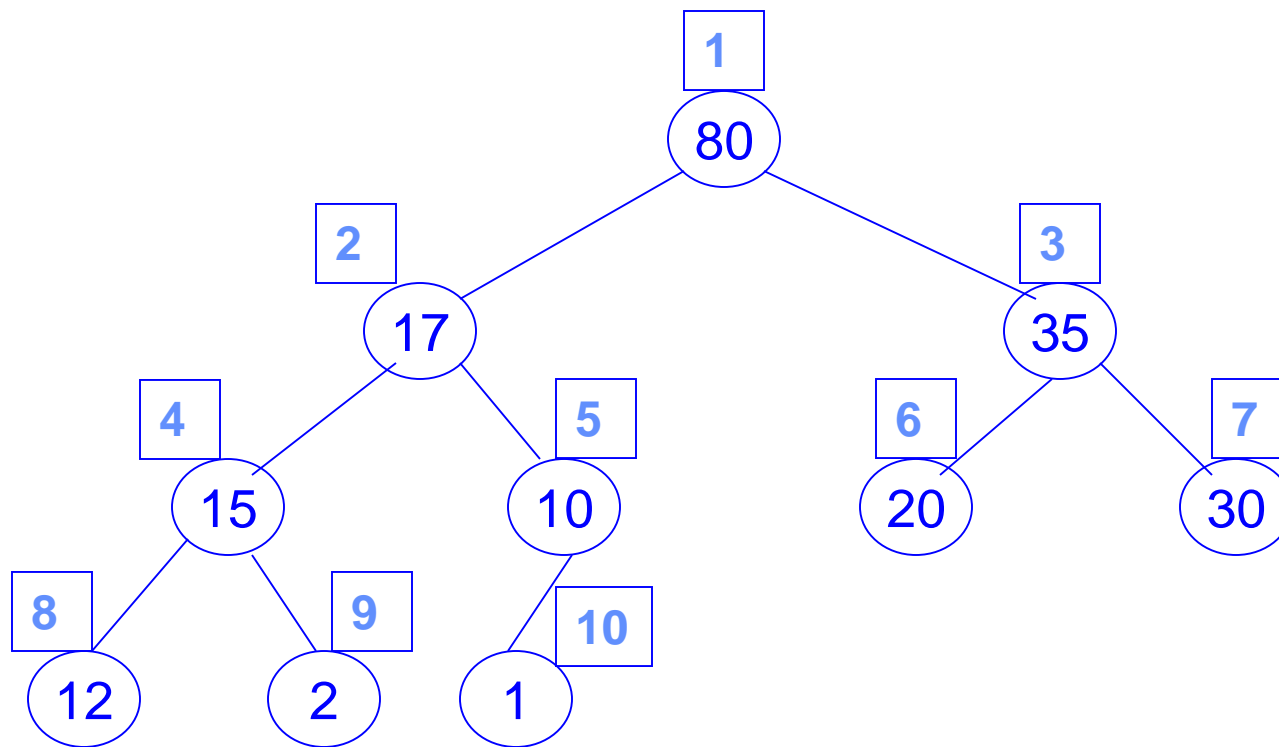
最大堆的初始化step5_1

$i=1, j=2i+1=3$



80	1
17	2
20	3
15	4
10	5
35	6
30	7
12	8
2	9
1	10

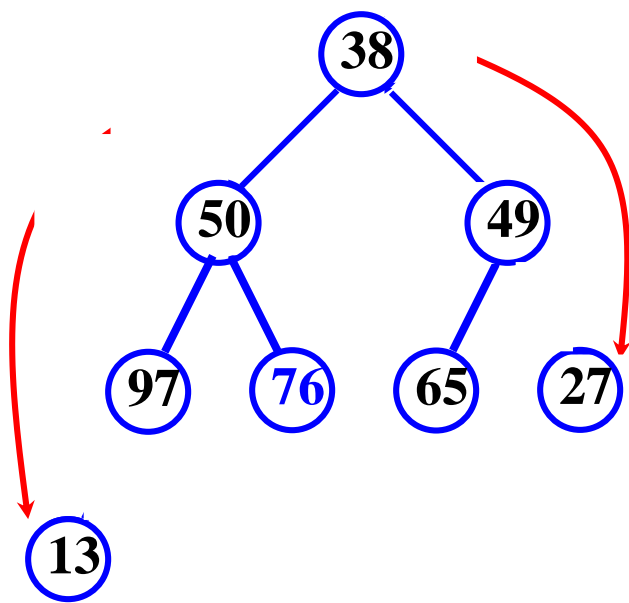
最大堆的初始化step5_2



80	1
17	2
35	3
15	4
10	5
20	6
30	7
12	8
2	9
1	10

例，序列 49 38 65 97 76 13 27 50

1. 按顺序依次构造成完全二叉树的结点；



2. 把完全二叉树改造成堆；

从下向上，父子交换；

3. 取得最小值 13

4. 删除 13，重新改造成新堆；

5. 取得次小值 27；

6. 删除 27，重新改造成新堆；

7. 取得次次小值 38；

```
void HeapAdjust (RcdType &R[], int s, int m)
{ // 已知 R[s..m]中记录的关键字除 R[s] 之外均
  // 满足堆的特征，本函数自上而下调整 R[s] 的
  // 关键字，使 R[s..m] 也成为一个大顶堆

  rc = R[s]; // 暂存 R[s]

  for ( j=2*s; j<=m; j*=2 ) { // j 初值指向左孩子
    自上而下的筛选过程;
  }

  R[s] = rc; // 将调整前的堆顶记录插入到 s 位置
} // HeapAdjust
```

自上而下的筛选过程

```
if ( j<m && R[j].key<R[j+1].key ) ++j;  
    // 左/右 “子树根” 之间先进行相互比  
    // 较令 j 指示关键字较大记录的位置  
if ( rc.key >= R[j].key ) break;  
    // 再作 “根” 和 “子树根” 之间的比较,  
    // 若 “>=”成立, 则说明已找到 rc 的插  
    // 入位置 s , 不需要继续往下调整  
R[s] = R[j]; s = j;  
    // 否则记录上移, 尚需继续往下调整
```

```
typedef SqList HeapType; // 堆采用顺序表表示
void HeapSort ( HeapType &H ) {
    // 对顺序表 H 进行堆排序
    for ( i=H.length/2; i>0; --i )
        HeapAdjust ( H.r, i, H.length ); // 建大顶堆
    for ( i=H.length; i>1; --i ) {
        H.r[1]↔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.r, 1, i-1); // 对 H.r[1] 进行筛选
    }
} // HeapSort
```


10.5 归并排序

归并：将两个或两个以上的有序表合并成一个新的有序表。

有序线性表、有序链表的归并；

利用归并的**思想**实现排序：

初始，**n** 个记录看作是 **n** 个有序的子序列，长度为 **1**；

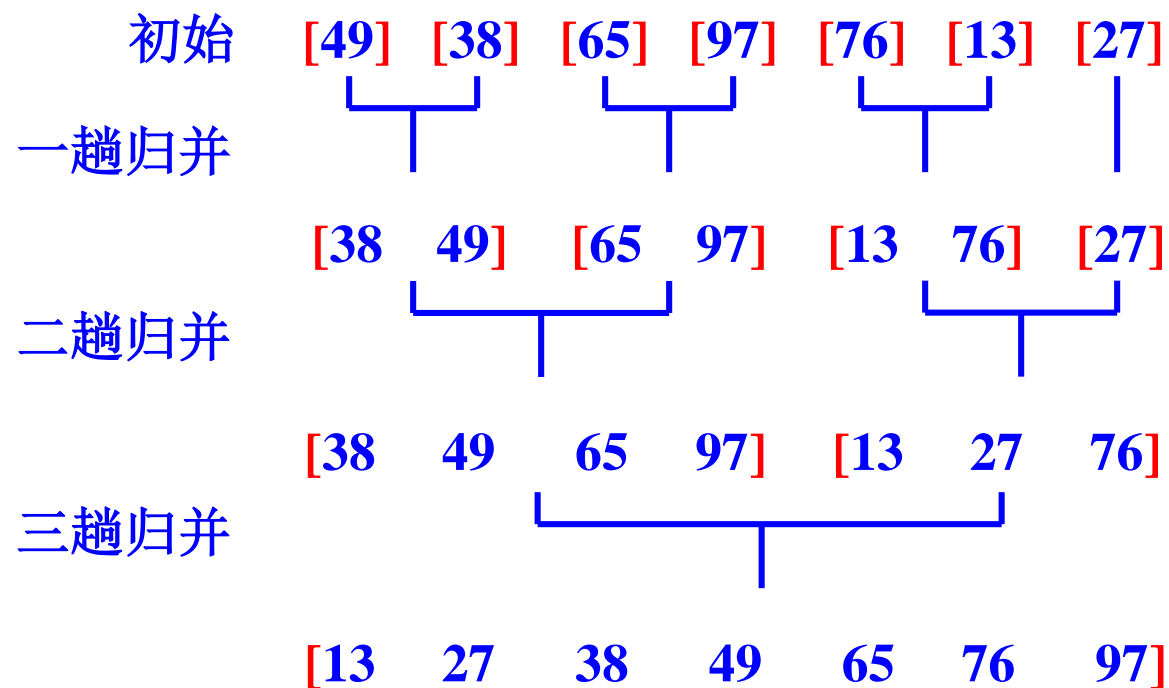
两两归并，得到 $\lceil n/2 \rceil$ 个长度为 **2** 或 **1** 的有序的子序列；

再两两归并；

重复执行直至得到一个长度为 **n** 的有序序列为止。

这种排序方法称为 **2-路归并排序**。

例，序列 { 49 , 38 , 65 , 97 , 76 , 13 , 27 }



```
void Merge (RecType SR[], RecType &TR[], int i, int m, int n)
{
    // 将有序的记录序列 SR[i..m] 和 SR[m+1..n]
    // 归并为有序的记录序列 TR[i..n]
    for (j=m+1, k=i; i<=m && j<=n; ++k)
    {
        // 将SR中记录由小到大并入TR
        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
```

归并排序的算法

如果记录无序序列 $R[s..t]$ 的两部分

$R[s..\lfloor (s+t)/2 \rfloor]$ 和 $R[\lfloor (s+t)/2 \rfloor + 1..t]$

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。

例如:

52, 23, 80, 36, 68, 14 (s=1, t=6)

[52, 23, 80] [36, 68, 14]

[52, 23][80] [36, 68][14]

[52] [23]

[36][68]

[23, 52]

[36, 68]

[23, 52, 80] [14, 36, 68]

[14, 23, 36, 52, 68, 80]

```
void Msort ( RecType SR[], RecType &TR1[], int s, int t ) {  
    // 将SR[s..t] 归并排序为 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]  
    }  
} // Msort
```

```
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
```

容易看出，对 n 个记录进行归并排序的时间复杂度为 $O(n \log n)$ 。即：
每一趟归并的时间复杂度为 $O(n)$ ，
总共需进行 $\lceil \log_2 n \rceil$ 趟。





10.6 基数排序

是一种借助多关键字排序的思想对单逻辑关键字进行排序的方法。

10.6.1 多关键字排序

扑克牌问题：

已知扑克牌中52张牌面的次序关系定义为：

花色：  <  <  < 

花色优先级更高，
为主关键字，面
值为次关键字

面值： 2 < 3 < ... < A

例，  8 <  3

52张牌排序方法：

最高位优先法(MSDF)：

先按不同“花色”分成有次序的4堆，每一堆均具有相同的花色；
然后分别对每一堆按“面值”大小整理有序。

最低位优先法(LSDF)：

先按不同“面值”分成 13 堆；

然后将这 13 堆牌自小至大叠在一起(2, 3, ..., A)；

然后将这付牌整个颠倒过来再重新按不同的“花色”分成 4 堆；

最后将这 4 堆牌按自小至大的次序合在一起。

收集

分配

10.6.2 基数排序

基数排序借助“分配”和“收集”两种操作实现对单逻辑关键字的排序。

首先，单逻辑关键字通常都可以看作是由若干关键字复合而成。

例，若关键字是数值，且值域为 $0 \leq K \leq 999$ ，

故可以将 K 看作是由 3 个关键字 $K^0 K^1 K^2$ 组成，

例，603是由 6 0 3 组成。

其次，利用 LSDF 法实现对若干关键字的排序。(p285^①)

注意：分配过程必须使得 K^i 相同的元素保持稳定！

例，序列	278	109	063	930	589	184	505	269	008	083
第一趟分配	0 ↓ 930	1	2	3 ↓ 063 ↓ 083	4 ↓ 184	5 ↓ 505	6	7	8 ↓ 278 ↓ 008	9 ↓ 109 ↓ 589 ↓ 269
第一趟收集	930	063	083	184	505	278	008	109	589	269
第二趟分配	0 ↓ 505 ↓ 008 ↓ 109	1	2	3 ↓ 930	4	5	6 ↓ 063 ↓ 269	7 ↓ 278	8 ↓ 083 ↓ 184 ↓ 589	9
第二趟收集	505	008	109	930	063	269	278	083	184	589

第二趟收集 505 008 109 930 063 269 278 083 184 589

第三趟分配

0	1	2	3	4	5	6	7	8	9
↓	↓	↓			↓				↓
008	109	269			505				930
↓	↓	↓			↓				
063	184	278			589				
↓									
083									

第三趟收集 008 063 083 109 184 269 278 505 589 930

```
#define MAX_NUM_OF_KEY    8 // 关键字项数的最大值
#define RADIX              10 // 关键字基数，十进制整数的基数
#define MAX_SPACE         1000
typedef struct // 静态链表的结点类型
{
    KeysType keys[MAX_NUM_OF_KEY]; // 关键字
    InfoType otheritems; // 其它数据项
    int next;
}SLCell;

typedef struct // 静态链表类型
{
    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;

typedef int ArrType[RADIX];
```

```
void Distribute (SLCell &r, int i, ArrType &f, ArrType &e ) {  
    // 静态键表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将记录中第i个关键字映射到[0..RADIX-1]  
        if(!f[j]) f[j]=p;  
        else r[e[j]].next=p;  
        e[j]=p; // 将p所指的结点插入第j个子表中  
    }  
} // Distribute
```

```
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-1)  
    {  
        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  
    }  
    r[t].next=0; // t指向最后一个非空子表中的最后一个结点  
} // Collect
```

```
void RadixSort(SLCell &r) {
```

```
    // 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,f,e);    // 第i趟收集
```

```
    }
```

```
} // RadixSort
```


10.7 各种排序方法的综合比较(pp289)

	平均时间	最差	最佳	辅助空间	稳定性
直接插入	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
起泡排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
直接选择	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
希尔排序	$O(n^{1.5})$		$O(1)$		不稳定
快速排序	$O(n \log n)$	$O(n^2)$	同平均	$O(\log n)$	不稳定
堆排序	$O(n \log n)$	同平均	同平均	$O(1)$	不稳定
归并排序	$O(n \log n)$	同平均	同平均	$O(n)$	稳定
基数排序	$O(d(n+r))$	同平均	同平均	$O(n+r)$	稳定

一、时间性能

1. 平均的时间性能

时间复杂度为 $O(n\log n)$:

快速排序、堆排序和归并排序

时间复杂度为 $O(n^2)$:

直接插入排序、起泡排序和
简单选择排序

时间复杂度为 $O(n)$:

基数排序

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,

快速排序的时间性能蜕化为 $O(n^2)$ 。

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。

二、空间性能

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) 和**堆排序**的空间复杂度为 **$O(1)$** ;
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**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，则空间复杂度为 $O(rd)$ 。

✂ 一个对象序列的排序码为{46, 79, 56, 38, 40, 84}, 采用快速排序以位于最左位置的对象为基准而得到的第一次划分结果为 ()。

A. {38, 46, 79, 56, 40, 84}

B. {38, 79, 56, 46, 40, 84}

C. {40, 38, 46, 56, 79, 84}

D. {38, 46, 56, 79, 40, 84}

- ⌘ 设有5000个无序的元素，希望用最快的速度挑选出其中前50个最大的元素，最好选用()法。
 - A. 冒泡排序
 - B. 快速排序
 - C. 堆排序
 - D. 基数排序
- ⌘ 下列序列中()是执行第一趟快速排序后得到的序列（排序的关键字类型是字符串）。
 - A. [da,ax,eb,de,bb] ff [ba,gc]
 - B. [cd,eb,ax,da,bb] ff [ha,gc]
 - C. [gc,ax,cb,cd,bb] ff [da,ba]
 - D. [ax,bb,cd,da] ff [eb,gc,ba]

⌘ 假定有 k 个关键字互为同义词，若用线性探测法将这 k 个关键字存入散列表中，至少需要进行（ ）次探测。

A. $k-1$ B. k C. $k+1$ D. $k(k+1)/2$

⌘ 从一棵二叉排序树中查找一个元素时，其平均时间复杂度为（ ）。

A. $O(1)$ B. $O(n)$ C. $O(\log_2 n)$ D. $O(n^2)$

- ⌘ 分别采用堆排序、快速排序、插入排序和归并排序算法对初始状态递增序列的按递增顺序排序，最省时间的是算法_____，最费时间的是算法_____。
- ⌘ 假定一组记录的排序码为(46, 79, 56, 38, 40, 84, 50, 42)，利用堆排序方法画出初始最大堆顶（以树状表示）。
- ⌘ 对于一组记录的排序码为(465, 792, 562, 383, 401, 845, 502, 423)，写出基数排序（低位优先）进行一趟分配与回收后的结果。

以下为直接插入排序算法：

/*对记录数组r做直接插入排序， length为数组的长度*/

void InsSort(RecordType r[], int length)

{

for (i=2 ; i< length ; i++)

{

r[0] = r[i]; j=i-1; /*将待插入记录存放到r[0]中*/

while (r[0].key< r[j].key) /* 寻找插入位置 */

{

r[j+1] = r[j]; j=j-1;

}

r[j+1] = r[0]; /*将待插入记录插入到已排序的序列中*/

}

} /* InsSort */

约定待排记录数为n，给出该算法的最大比较次数和最大移动次数。