第10章　内部排序

1. 排序的基本概念

●排序：将一组杂乱无章的数据按一定的规律顺次排列起来，使之按关键字递增(或递减)有序排列。

3	10	5	78	36
3	5	10	36	78

排序:

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

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

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

$$K_{p_1} \le K_{p_2} \le K_{p_3} \le \ldots \le K_{p_n}$$

则原序列变为一个按关键字有序的序列:

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

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

排序的时间开销：算法执行中关键字比较次数和记录移动次数来衡量。

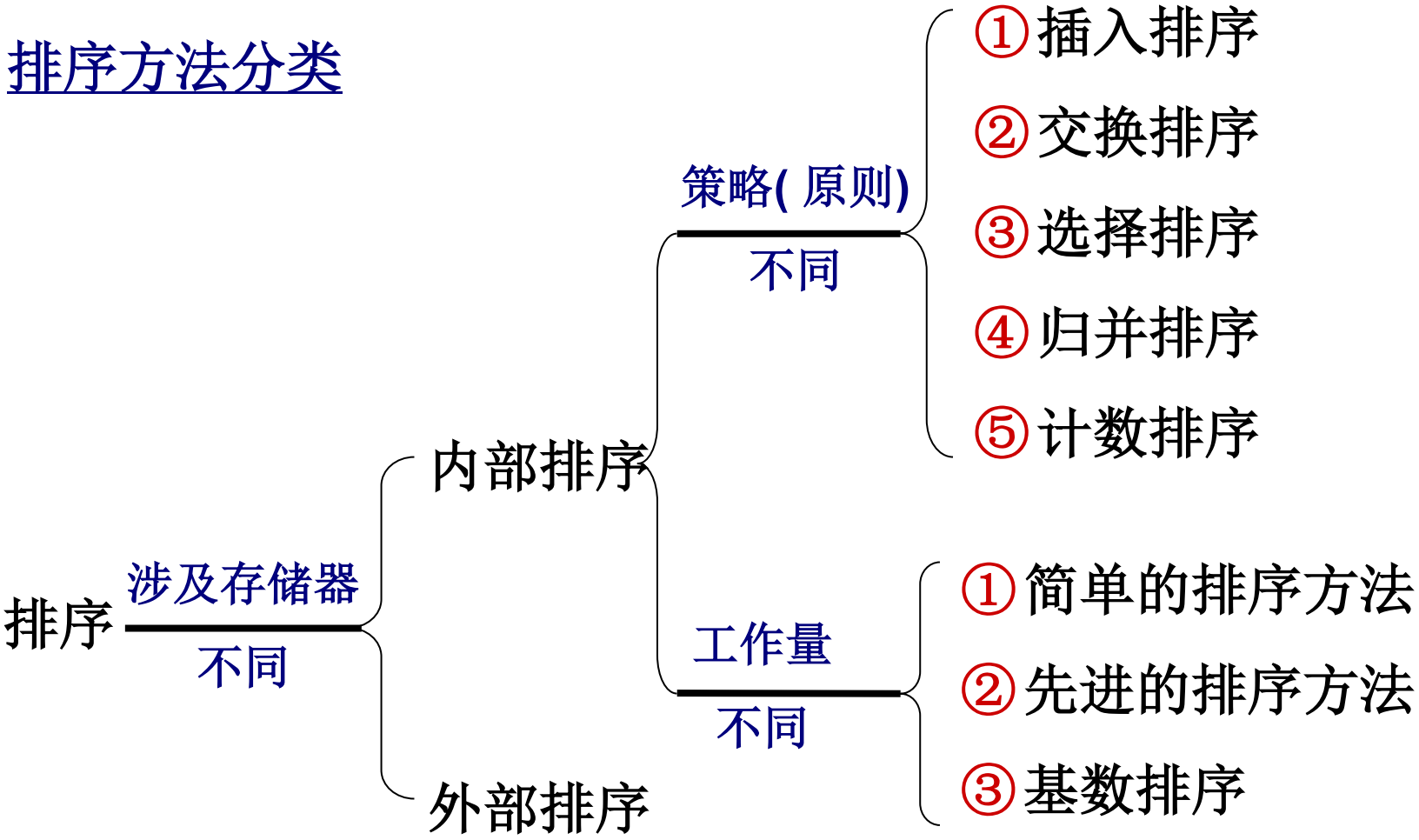
对于那些受关键字序列初始排列及记录个数影响较大的，需要按最好情况和最坏情况进行估算。

内部排序 与 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。

- 排序方法分类


排序
涉及存储器
不同
内部排序
外部排序
策略(原则)
不同
①插入排序
②交换排序
③选择排序
④归并排序
⑤计数排序
工作量
不同
①简单的排序方法
②先进的排序方法
③基数排序

- 待排序的顺序表类型的类型定义

```
#define MAXSIZE  20  //表长
typedef int KeyType;      //关键字类型
typedef struct {
     KeyType  key;          //关键字项
     InfoType otherType; //其它数据项
 } RecType;  //记录类型
 typedef struct{
     RecType   r[MAXSIZE+1];  //r[0]哨兵单元
     int           length;
 }SqList;
```

10.2 插入排序

插入排序(Insertion Sort)的基本思想是：每次将一个待排序的记录,按其关键字大小插入到前面已经排好序的子表中的适当位置,直到全部记录插入完成为止。

5种插入排序方法：

(1) 直接插入排序;
(2) 折半插入排序;
(3) 2-路插入排序;
(4) 表插入排序;
(5) 希尔排序。

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)
{
    for( i=2; i<= L.length; ++i )
        if( L.r[i].key < L.r[i-1].key){
            L.r[0] = L.r[i];
            L.r[i]  = L.r[i-1];
            for ( j=i-2; L.r[0].key < L.r[j].key; --j)
                  L.r[j+1] = L.r[j];
            L.r[j+1] = L.r[0];
        }
}
```

算法分析

- 需1个记录的辅助空间r[0];
- 记录个数为n，则n-1趟插入;
- 最好情况下(正序)

  $$\text{关键字比较次数} = \sum_{i=2}^{n} 1 = n-1$$

  记录不需移动。
- 最差情况下(逆序)

  $$\text{关键字比较次数} = \sum_{i=2}^{n} i = (n+2)(n-1)/2 \approx n^2/2$$

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

- 直接插入排序所需进行关键字间的比较次数和记录移动次数约为：$n^2/4$，算法时间复杂度为：$O(n^2)$。
- 直接插入排序是一个稳定的排序方法。

2. 折半插入排序

●基本思想：r[1..i-1] 已为有序序列，在插入 r[i] 时，利用折半查找法寻找 r[i] 的插入位置。

```
void BInsertSort( SqList &L)
{   for( i=2; i<=L.length; ++i){
        L.r[0] = L.r[i];   low = 1;  high = i-1;
        while( low <= high ){
            m = (low+high)/2;
            if( L.r[0].key < L.r[m].key)   high = m-1;
            else   low = m+1;
        }
        for( j=i-1; j>=high+1; --j)   L.r[j+1] = L.r[j];
        L.r[high+1] = L.r[0];
    }
}
```

●算法分析

◆关键字比较次数与待排序序列的初始状态无关，仅依赖于记录个数。在插入第 i 个记录时，需要经过 $\lfloor \log_2(i-1) \rfloor + 1$ 次关键字比较，才能确定它应插入的位置。

$$\sum_{i=1}^{n-1}\left(\lfloor \log_2 i \rfloor + 1\right) \approx n \log_2 n$$

- 折半插入排序的记录移动次数与直接插入排序相同，依赖于记录的初始排列。
- 折半插入排序的时间复杂度为 $O(n^2)$。
- 折半插入排序是一个稳定的排序方法。

3. 2-路插入排序

- 目的是减少记录移动的次数
- 以第一个记录为界，小于第一个记录的插入到前半部分，否则插入后半部分
- 第一个记录不动，视为中间位置
- 将整个数组视为一个循环队列，设置first和final两个指针指向第一个记录和最后一个记录

3. 2-路插入排序

初始关键字　49　38　65　97　76　13　27　49

i=1　(49)
↑↑ first final

i=2　(49)　(38)
↑final　↑first

i=3　(49　65)　(38)
↑final　↑first

i=4　(49　65　97)　(38)
↑final　↑first

….

↓first

i=8　(49　49　65　76　97　13　27　38)
↑final

算法分析

- 为了减少记录的移动次数，需n个记录的辅助空间。
- 移动记录的次数约为$n^2/8$。
- 2-路插入排序的时间复杂度为 $O(n^2)$。
- 2-路插入排序是一个稳定的排序方法。

4. 表插入排序

- 目的是不移动记录，只改变指针
- 每个记录维护一个next指针，构成一个循环链表
- 插入动作体现为改变next指针

4. 表插入排序

	0	1	2	3	4	5	6	7	8
初始状态	MAXINT	49	38	65	97	76	13	27	49
	1	0	-	-	-	-	-	-	-
i=2	2	0	1	-	-	-	-	-	-
i=3	2	3	1	0	-	-	-	-	-
i=4	2	3	1	4	0	-	-	-	-
i=5	2	3	1	5	0	4	-	-	-
i=6	6	3	1	5	0	4	2	-	-
i=7	6	3	1	5	0	4	7	2	-
i=8	6	8	1	5	0	4	7	2	3

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

	0	1	2	3	4	5	6	7	8
i=1, p=6	MAXINT	**13**	38	65	97	76	**49**	27	49
	6	**(6)**	1	5	0	4	**8**	2	3
i=2, p=7	MAXINT	13	**27**	65	97	76	49	**38**	49
	6	(6)	**(7)**	5	0	4	8	**1**	3
i=3, p=(2), 7	MAXINT	13	27	**38**	97	76	49	**65**	49
	6	(6)	(7)	**(7)**	0	4	8	**5**	3
i=4, p=(1), 6	MAXINT	13	27	38	**49**	76	**97**	65	49
	6	(6)	(7)	(7)	**(6)**	4	**0**	5	3
i=5, p=8	MAXINT	13	27	38	49	**49**	97	65	**76**
	6	(6)	(7)	(7)	(6)	**(8)**	0	5	**4**
i=6, p=(3), 7	MAXINT	13	27	38	49	49	**65**	**97**	76
	6	(6)	(7)	(7)	(6)	(8)	**(7)**	**0**	4
i=7, p=(5), 8	MAXINT	13	27	38	49	49	65	**76**	**97**
	6	(6)	(7)	(7)	(6)	(8)	(7)	**(8)**	**0**

●静态链表类型定义

```
#define SIZE  100
typedef struct{
    RcdType  rc;     //记录项
    int          next; //指针项
}SLNode;  //表结点类型
typedef struct{
    SLNode    r[SIZE];   //0号单元为表头结点
    int          length;
}SLinkListType;  //静态链表类型
```

```
void  Arrange( SLinkListType  &SL)
{
    p = SL.r[0].next;
    for( i=1; i<SL.length; ++I ){ //SL.r[1..i-1]已有序
        while( p<i)  p = SL.r[p].next; //找第i个记录
        q = SL.r[p].next; //指向尚未调整的表尾
        if( p != i )  { SL.r[p]←→SL.r[i];  SL.r[i].next = p;   }
        p = q;
    }
}
```

- 算法分析：
  - 修改2n次指针代替记录移动，关键字比较次数与直接插入排序相同；重排最坏情况下进行3(n-1)次记录移动。
  - 算法时间复杂度：$O(n^2)$.

5. 希尔排序(Shell's Sort) 1959年 “缩小增量排序”

- 基本思想：
  - 待排记录按关键字“基本有序”，即具有特性

L.r[i].key < Max{L.r[j].key} (1≤j＜i)的记录较少。

  - 先将整个待排序列分割成若干子序列分别进行直接插入排序，待整个序列“基本有序”时，再对全体序列进行一次直接插入排序。

  - 子序列的构成不是简单地“逐段分割”，而是将相隔某个“增量”的记录组成一个子序列。

思想：

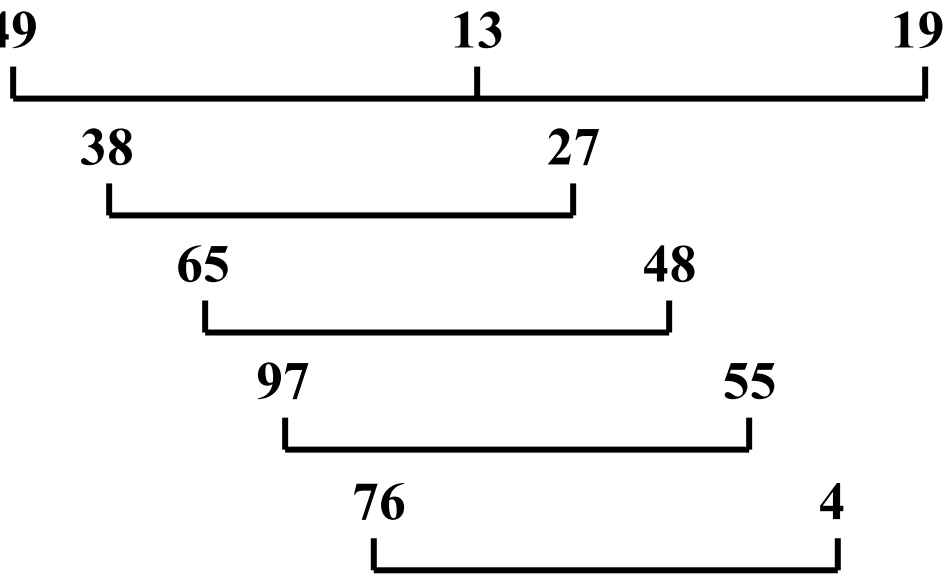
先将待排序记录序列分割成为若干子序列分别进行直接插入排序；

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

第一趟排序

Gap=5

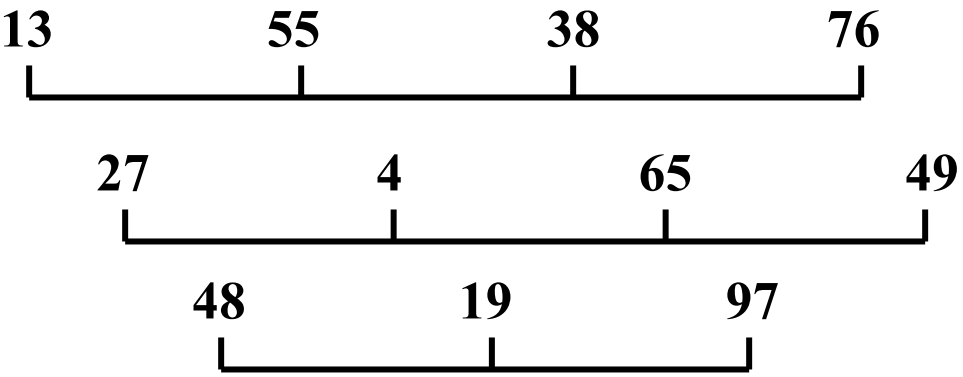


13 27 48 55 4 19 38 65 97 76 49

13 27 48 55 4 19 38 65 97 76 49

第二趟排序

Gap=3



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

Gap=1

```
void ShellInsert( SqList &L, int Gap)
{
    for( i=Gap+1; i<=L.length; ++i)
        if( L.r[i].key < L.r[i-Gap].key){ //L.r[i]插入有序增量子表
            L.r[0] = L.r[i];
            for( j=i-Gap; j>0&&(L.r[0].key<L.r[j].key); j-=Gap )
                L.r[j+Gap] = L.r[j];
            L.r[j+Gap] = L.r[0];
        }
}
```

```
void ShellSort( SqList &L, int dlta[], int t)
{
    for( k=0; k<t; ++k)
        ShellInsert(L, dlta[k]);
}
```

算法分析

- 开始时 Gap 的值较大，子序列中的记录较少，排序速度较快；随着排序进展，Gap 值逐渐变小，子序列中记录个数逐渐变多，由于前面工作的基础，大多数序列已基本有序，所以排序速度仍然很快。
- 关键字比较次数和记录移动次数与增量选择之间的依赖关系，并给出完整的数学分析，还没有人能够做到(难题）。
- Gap的取法有多种。最初 shell 提出取 gap = ⌊n/2⌋，gap = ⌊gap/2⌋，直到gap = 1。后来knuth 提出取gap = ⌊gap/3⌋ +1。还有人提出都取奇数为好，也有人提出各gap互质为好。
- Knuth利用大量的实验统计资料得出，当 n 很大时，关键字平均比较次数和记录平均移动次数大约在 $n^{1.25}$ 到 $1.6n^{1.25}$ 的范围内，当$n \to \infty$时，可减少到$n(\log_2 n)^2$。这是在利用直接插入排序作为子序列排序方法的情况下得到的。
- 希尔排序不是稳定的排序方法。

10.3 交换排序

10.3.1 起泡排序

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：

首先将第一个元素与第二个元素比较大小，若为逆序，则交换；

然后比较第二个元素与第三个元素的大小，若为逆序，则交换；

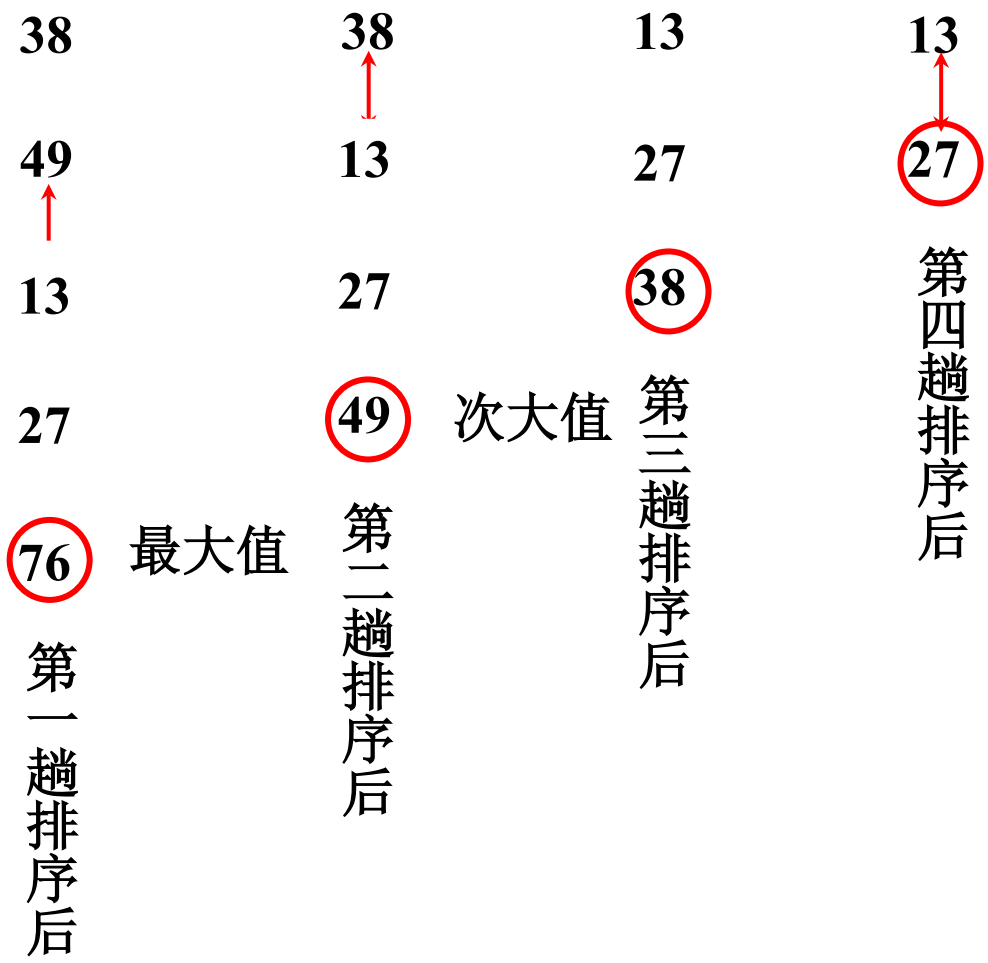
⋮

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



```
void BubbleSort( SqList &L)
{
    for( i=L.length, change=TRUE;
           i>1 && change; --i )
    {    change=FALSE;
         for (j=1;j<i;j++)
                if( L.r[j].key > L.r[j+1].key)
                { L.r[j] ←→ L.r[j+1];
                  change=TRUE;
                }
        }
}
```

- 算法分析
  - 正序：进行n-1次关键字的比较，且不移动记录；
  - 逆序：进行n-1趟排序

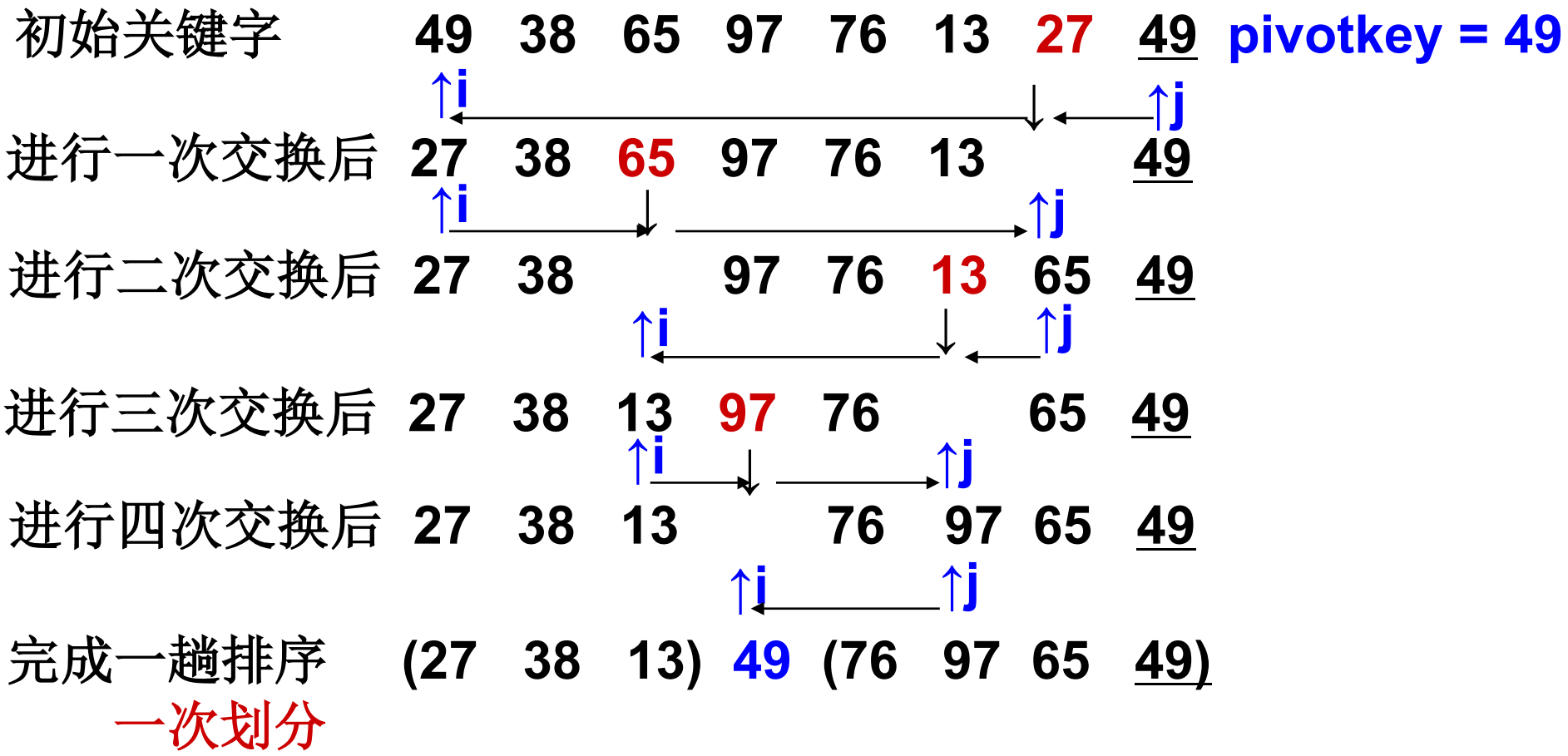
关键字比较次数：$\sum_{i=n}^{2}(i-1)=n(n-1)/2$

记录移动次数：$\sum_{i=n}^{2}3(i-1)=3n(n-1)/2$

- 起泡排序的时间复杂度：$O(n^2)$.
- 起泡排序是一种稳定的排序方法。

2. 快速排序

基本思想：通过一趟排序将待排记录分割成独立的两部分，其中一部分的关键字均比另一部分的关键字小，再对这两部分继续排序。



初始关键字　49　38　65　97　76　13　27　49

一次划分后　(27　38　13)　49　(76　97　65　49)

分别进行划分　(13)　27　(38)　49　(49　65)　76　(97)

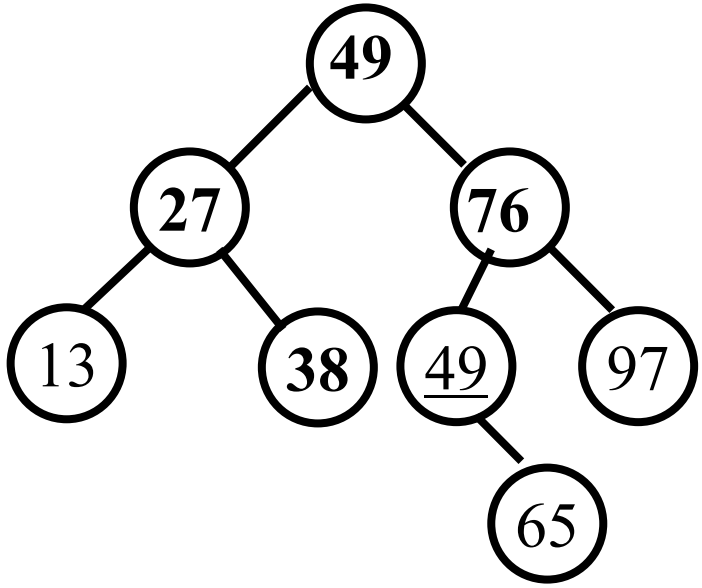
13　27　38　49　49　(65)　76　97

13　27　38　49　49　65　76　97

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

```
int Partition( SqList &L,  int low,  int high)
{
    L.r[0] = L.r[low];
    pivotkey = L.r[low].key;
    while( low < high){
        while( low < high && L.r[high].key >= pivotkey)  --high;
        L.r[low] = L.r[high];
        while( low < high && L.r[low].key <= pivotkey)  ++low;
        L.r[high] = L.r[low];
    }
    L.r[low] = L.r[0];
    return low;
}
```

■算法分析



◆最坏情况(有序)，递归树成为单支树，每次划分只得到一个比上一次少一个记录的子序列。总的关键字比较次数达到：

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

占用附加存储(即栈)将达到O(*n*)。

- 最好情况(基准为中值，左右子区间大致相等)：在n个元素的序列中，对一个记录定位所需时间为 $O(n)$， 设$T(n)$ 是对 n 个记录的序列进行排序所需的时间，总的计算时间为：

$$
\begin{aligned}
T(n) &\le cn + 2\,T(n/2) && \text{// } c\text{是一个常数} \\
&\le cn + 2\,(\,cn/2 + 2T(n/2^2)\,) = 2cn + 2^2T(n/2^2) \\
&\le 2cn + 4\,(\,cn/4 + 2t(n/2^3)\,) = 3cn + 2^3T(n/2^3) \\
&\cdots\cdots \\
&\le kcn + 2^kT(n/2^k) = cn\log_2 n + nT(1)
\end{aligned}
$$

递归调用层次数与递归树的深度一致 $\lfloor \log_2 n \rfloor + 1$

- 可以证明，QuickSort平均时间复杂度也是 $O(n\log_2 n)$。实验结果表明：就平均计算时间而言，快速排序是我们所讨论的所有内排序方法中最好的一个。

- 对于 n 较大的平均情况而言，快速排序是“快速”的，但是当 n 很小时，这种排序方法往往比其它简单排序方法还要慢。有一种改进办法：取每个待排序对象序列的第一个对象、最后一个对象和位置接近正中的3个对象，取其关键字居中者作为基准对象。

- 快速排序是一种不稳定的排序方法。

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—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—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—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:

38　49　97　65　76

第 2 趟选择:

38　49　97　65　76

第 3 趟选择:

38　49　65　97　76

第 4 趟选择:

38　49　65　76　97

简单(直接)选择排序

$$\text{关键字比较次数} = \sum_{i=1}^{n-1}(n-i) = \frac{n(n-1)}{2}$$

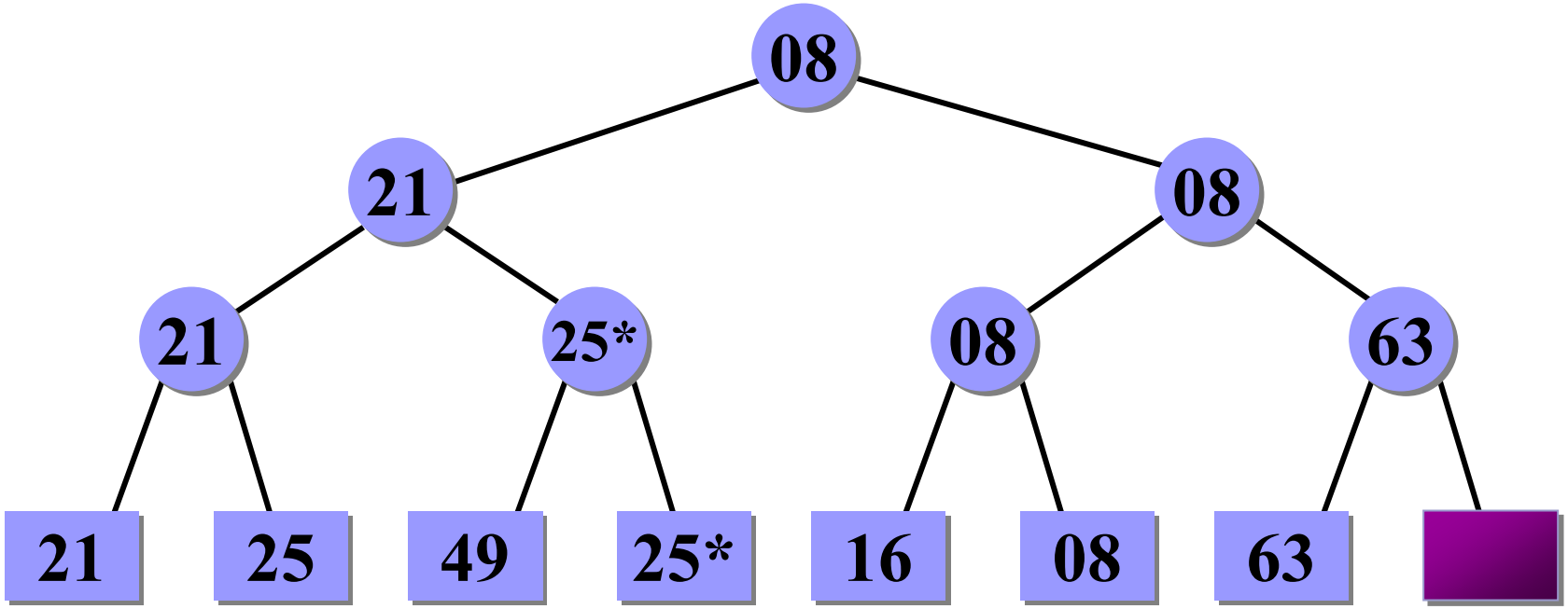
记录移动次数：最好：0次；

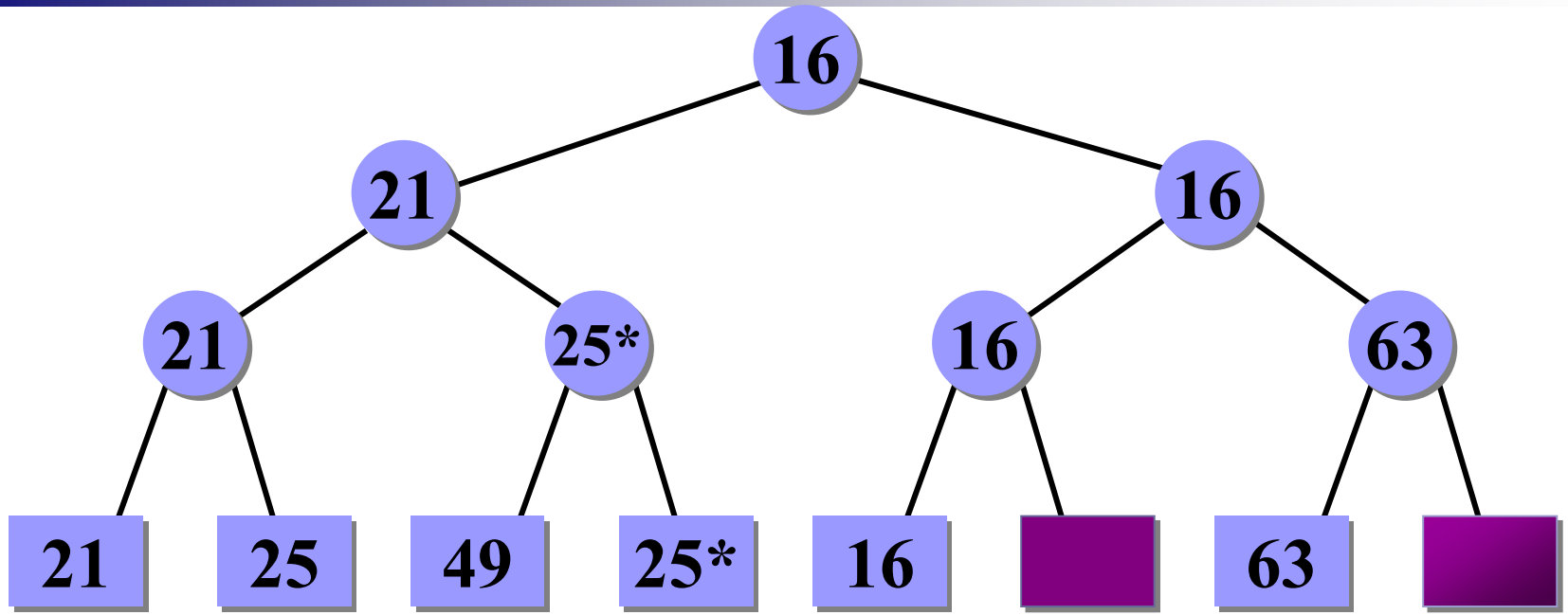
最坏：3(n-1).

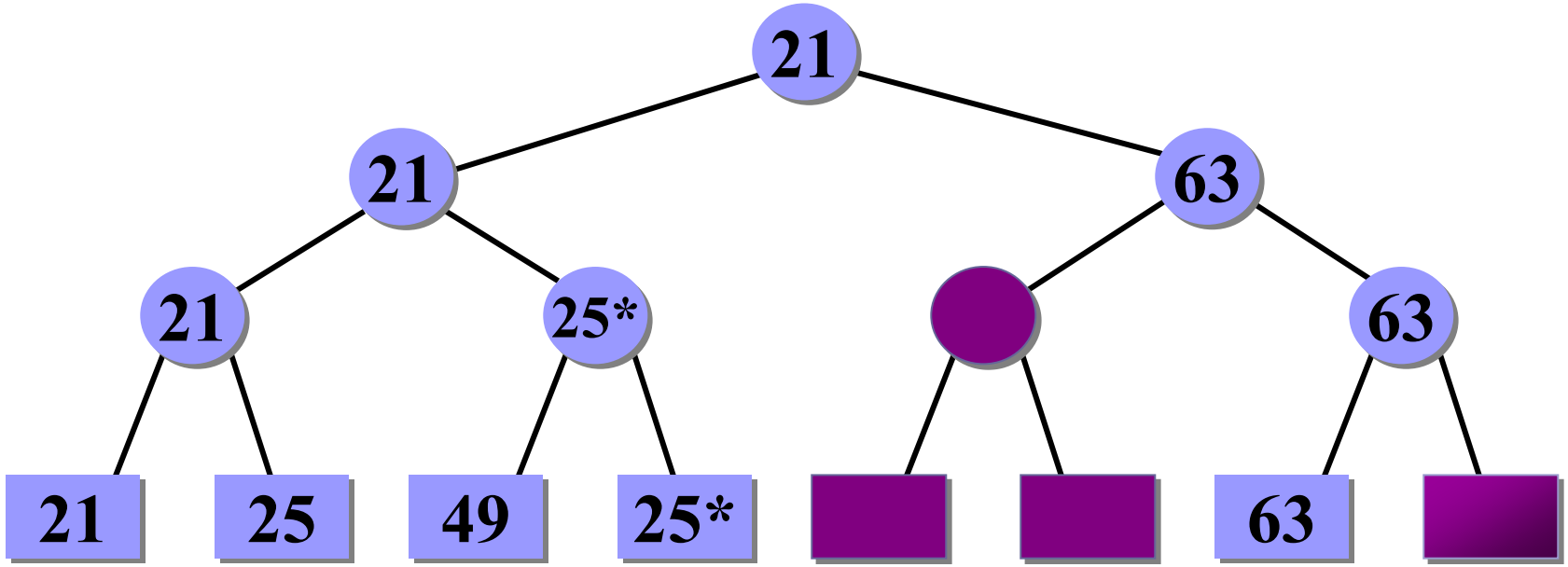
2. 树形选择排序(锦标赛排序)

n 个记录的关键字，进行两两比较，得到⌈*n*/2⌉个比较的优胜者(关键字小者)，作为第一步比较的结果保留下来。然后对这⌈*n*/2⌉个记录再进行关键字的两两比较，…，如此重复，直到选出一个关键字最小的记录为止。

{21, 25, 49, 25*, 16, 08, 63}




16
21
16
21
25*
16
63
21
25
49
25*
16
63


21
21
63
21
25*
63
21
25
49
25*
63

算法分析

- 锦标赛排序构成的树是满的完全二叉树，其深度为$\lceil \log_2 n \rceil + 1$，其中 n 为待排序记录个数。
- 除第一次选择具有最小关键字的记录需要进行 $n-1$ 次关键字比较外，选择具有次小、再次小关键字记录所需的关键字比较次数均为 $O(\log_2 n)$。总关键字比较次数为$O(n\log_2 n)$。
- 记录的移动次数不超过关键字的比较次数，所以锦标赛排序总的时间复杂度为$O(n\log_2 n)$。
- 这种排序方法虽然减少了许多排序时间，但是使用了较多的附加存储。
- 锦标赛排序是一个稳定的排序方法。

3. 堆排序(Heap Sort)

将r[1..n]看成是一棵完全二叉树的顺序存储结构,利用完全二叉树中双亲结点和孩子结点之间的内在关系,在当前无序区中选择关键字最大(或最小)的记录。

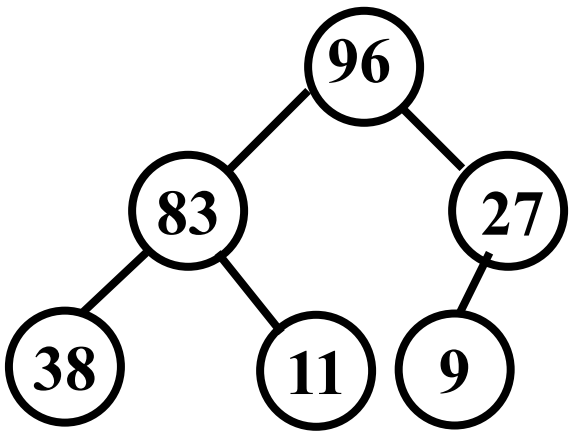
堆的定义：n个关键字序列$K_1,K_2,\ldots,K_n$称为堆，当且仅当该序列满足如下性质(简称为堆性质)：

(1) $K_i \le K_{2i}$ 且 $K_i \le K_{2i+1}$ 或

(2) $K_i \ge K_{2i}$ 且 $K_i \ge K_{2i+1}$ $(1 \le i \le \lfloor n/2 \rfloor)$

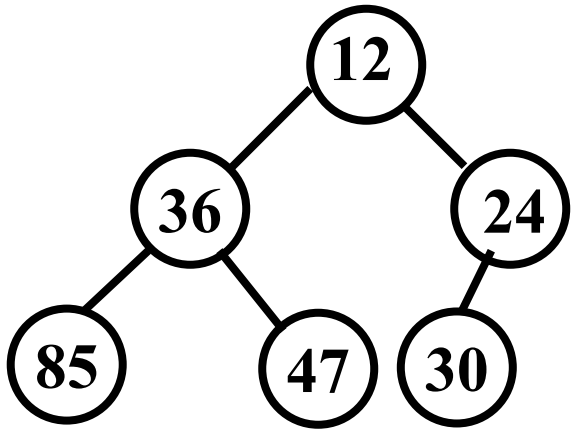
满足第(1)种情况的堆称为“小顶堆”,满足第(2)种情况的堆称为“大顶堆”。

{ 96, 83, 27, 38, 11, 9 }



大顶堆

{ 12, 36, 24, 85, 47, 30 }

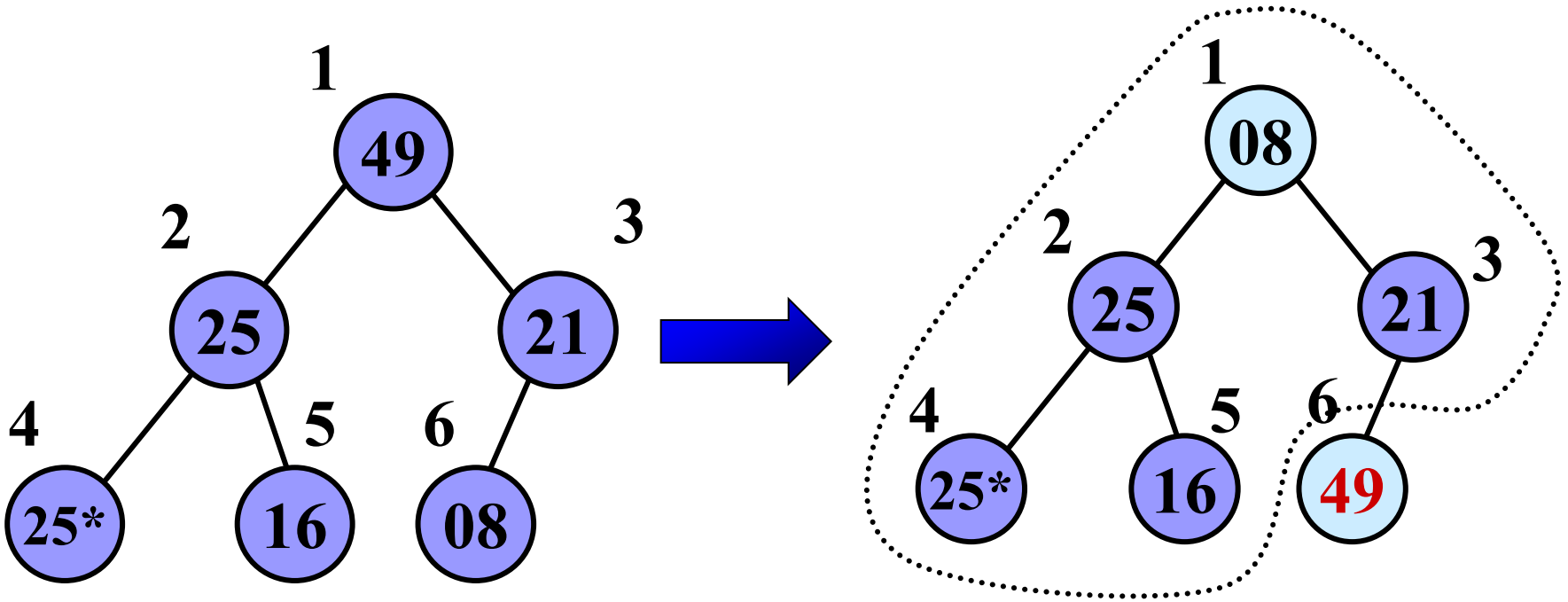


小顶堆

- 堆排序解决2个问题

①如何由一个无序序列建成一个堆？

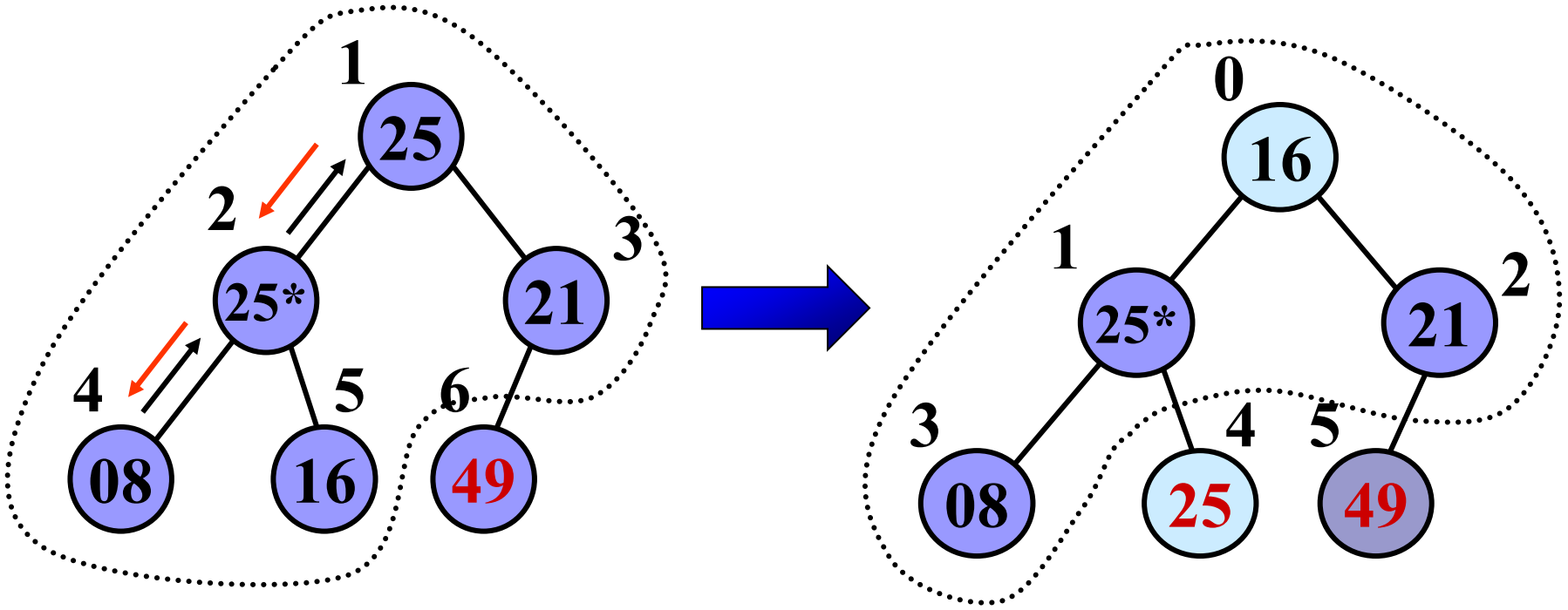
②如何在输出堆顶元素之后，调整剩余元素成为一个新的堆？



49	25	21	25*	16	08

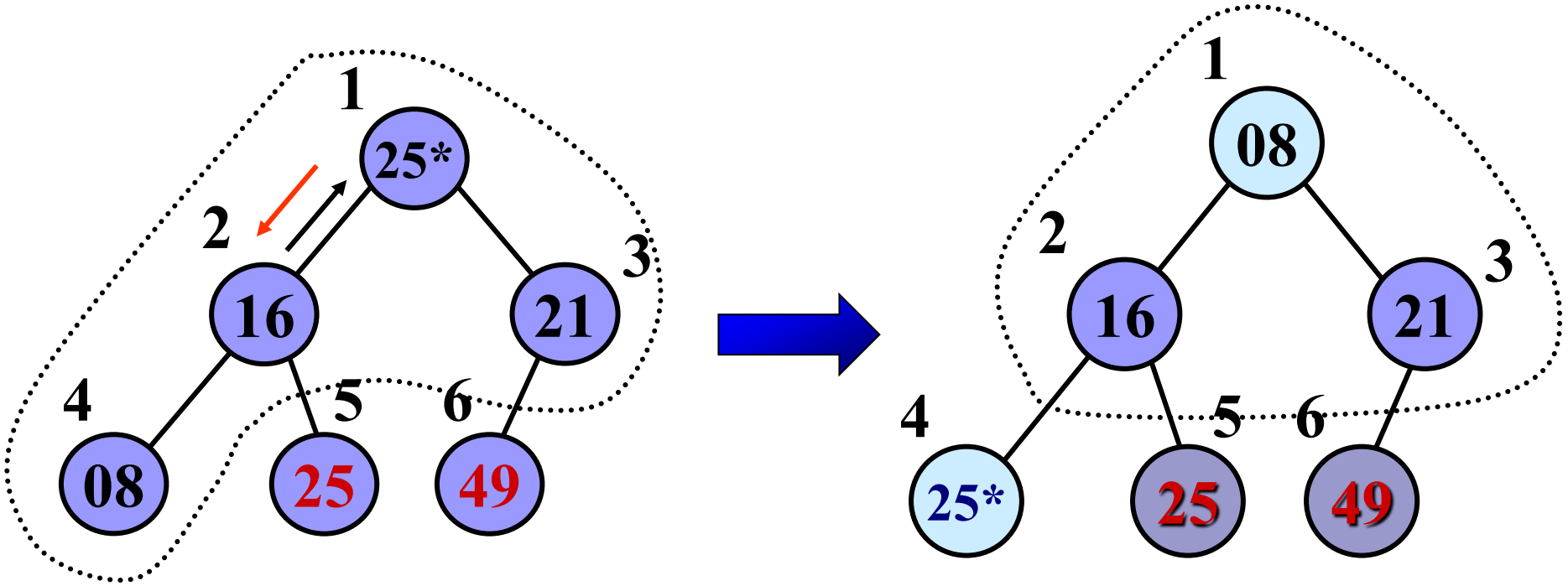
08	25	21	25*	16	49

自堆顶至叶子的调整过程为“筛选”。



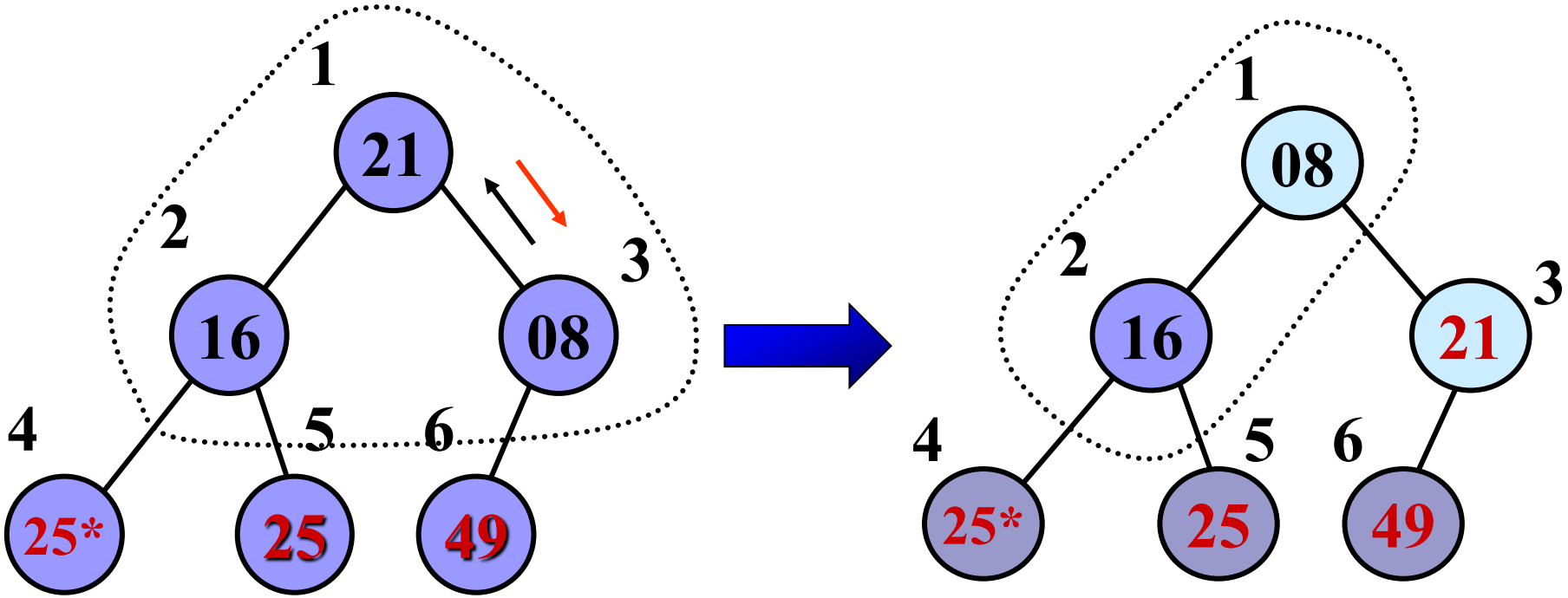
25	25*	21	08	16	49

16	25*	21	08	25	49



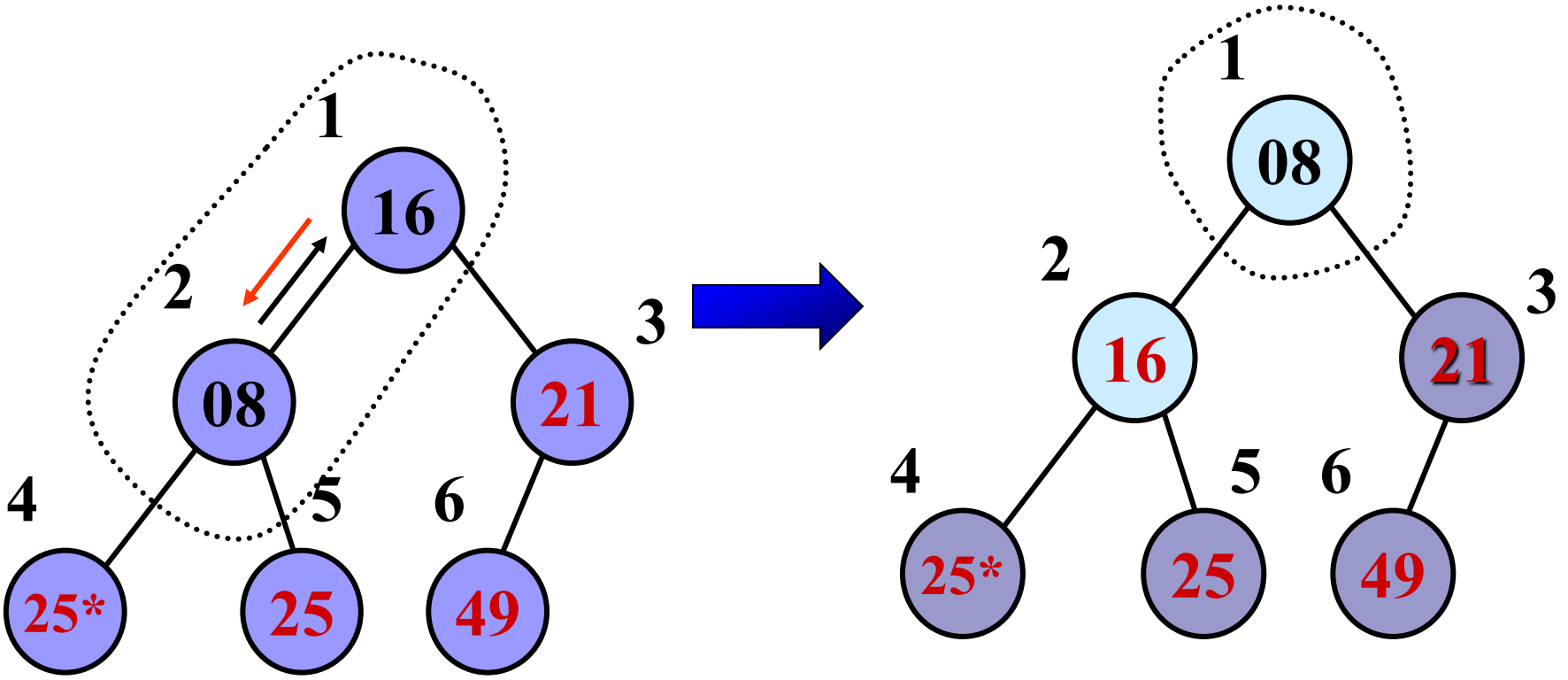
25*	16	21	08	25	49

08	16	21	25*	25	49



21	16	08	25*	25	49

08	16	21	25*	25	49

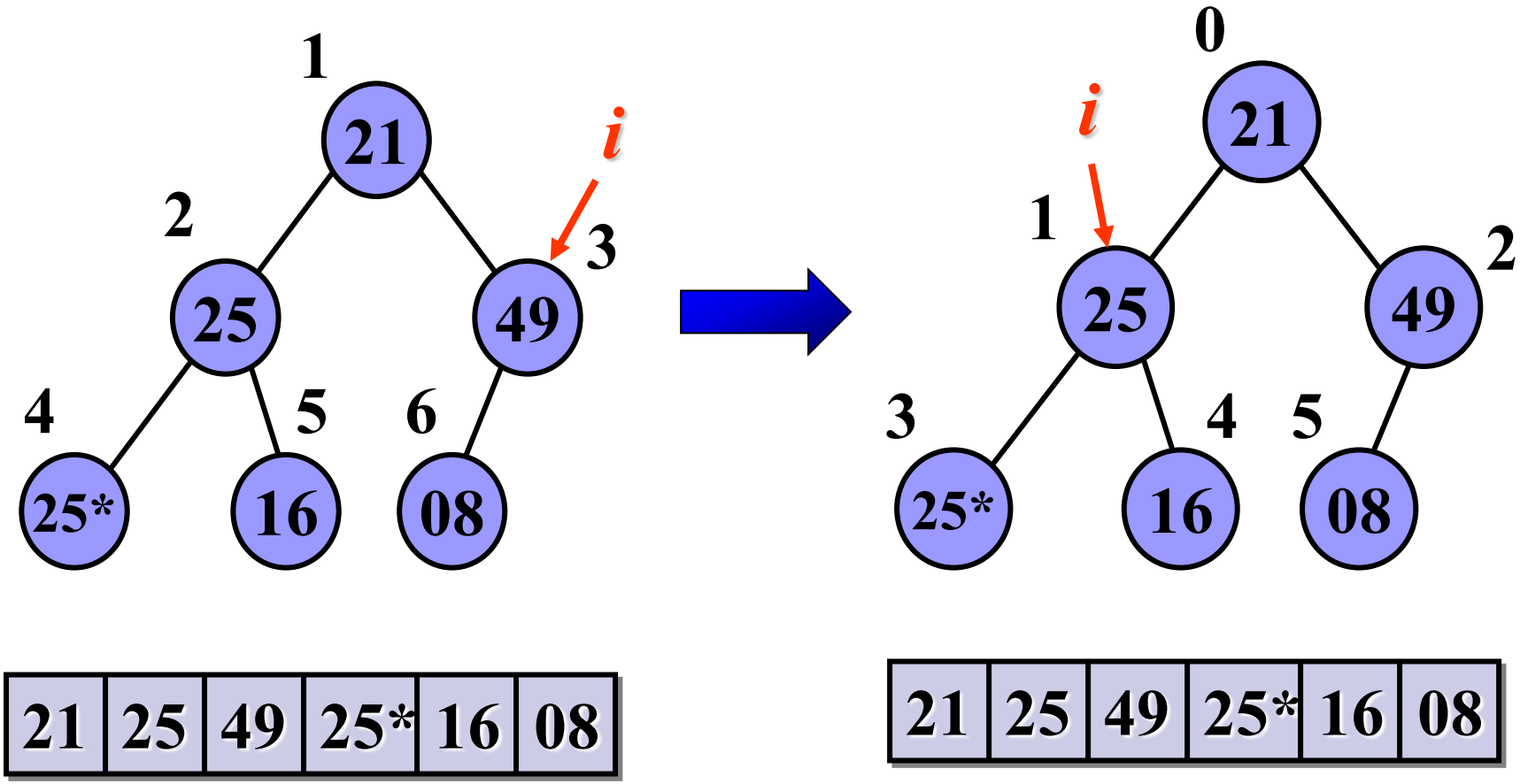


16	08	21	25*	25	49

08	16	21	25*	25	49

①如何由一个无序序列建成一个堆？反复“筛选”的过程。

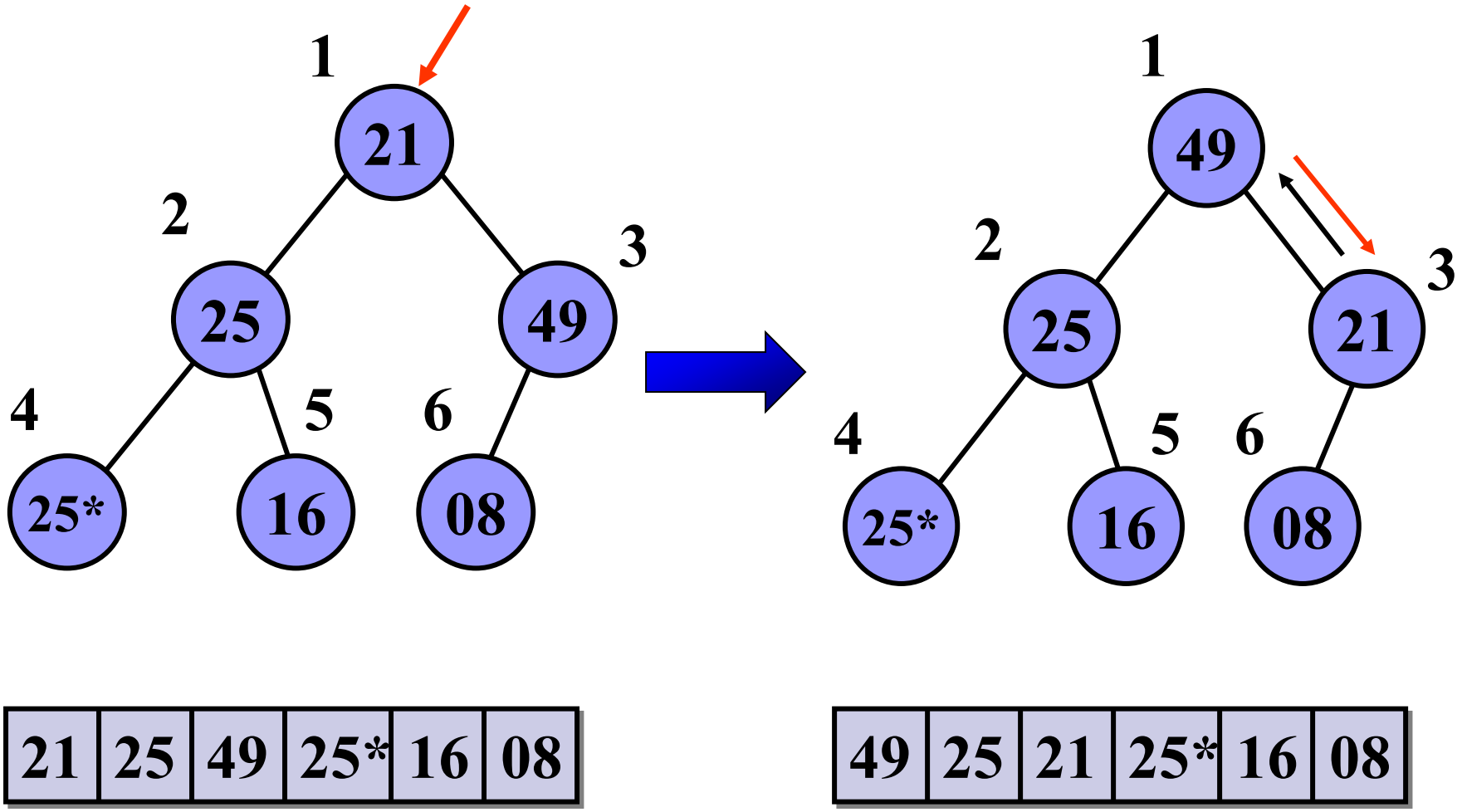
从i=⌊n/2⌋到1,反复“筛选”。



21	25	49	25*	16	08

初始关键字集合

21	25	49	25*	16	08

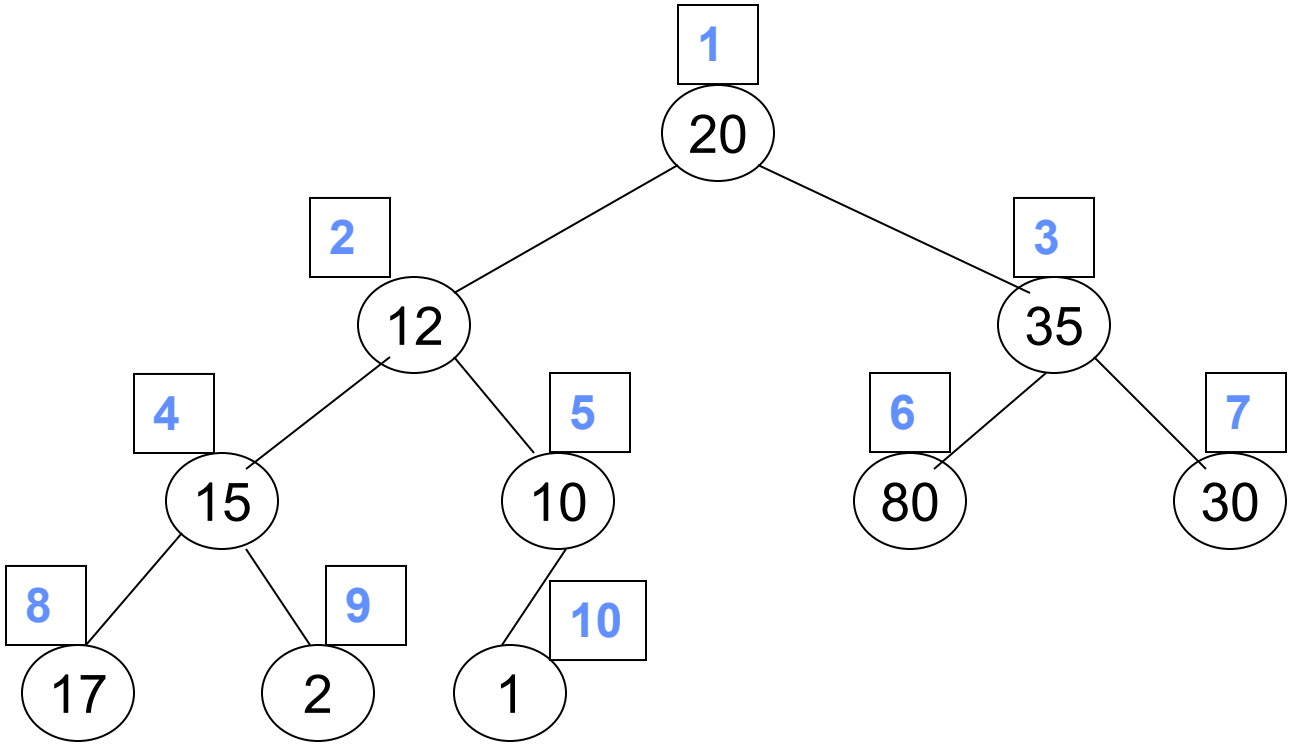


21	25	49	25*	16	08

49	25	21	25*	16	08

最大堆的初始化

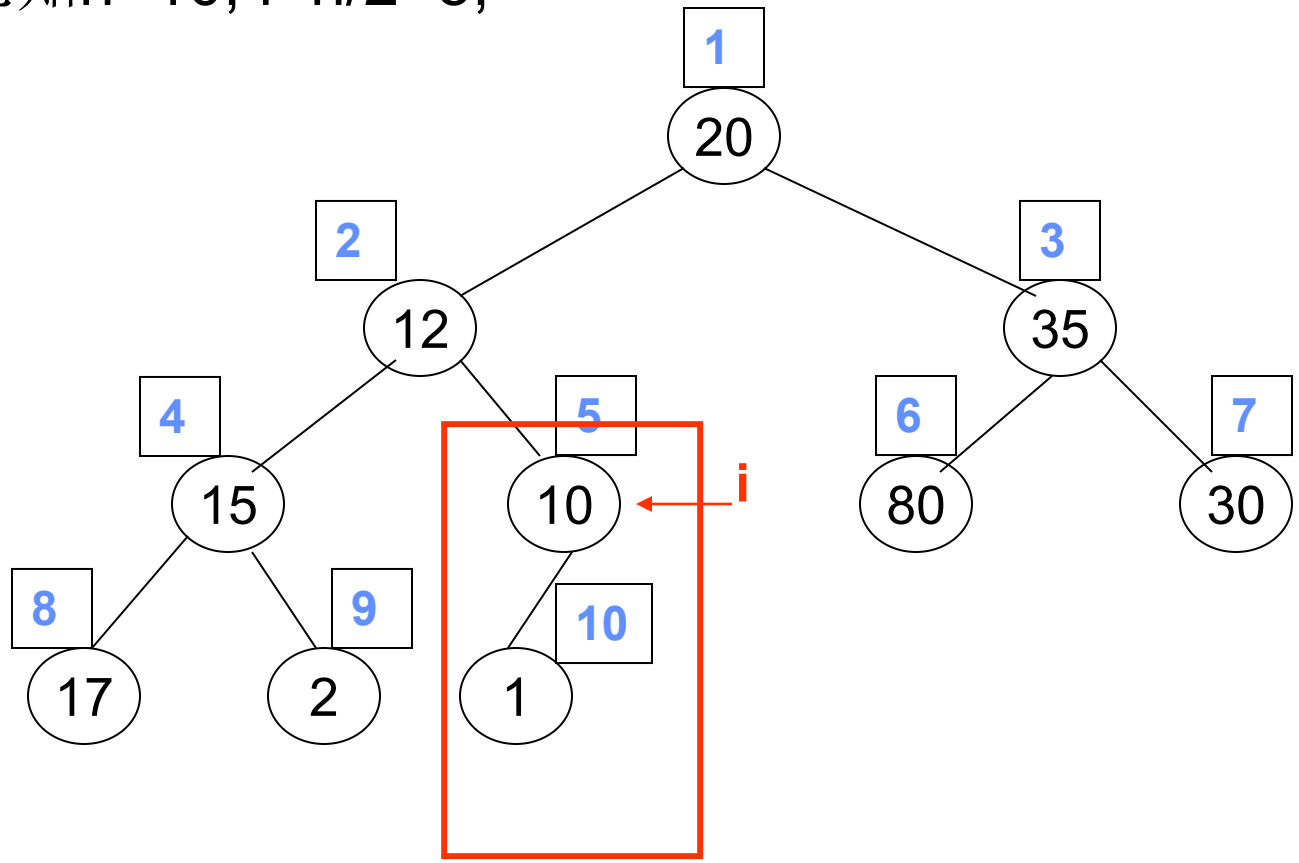
根据int a[10]={20,12,35,15,10,80,30,17,2,1}建立最大堆



算法：从第一个具有孩子的结点i开始（i=[n/2]），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

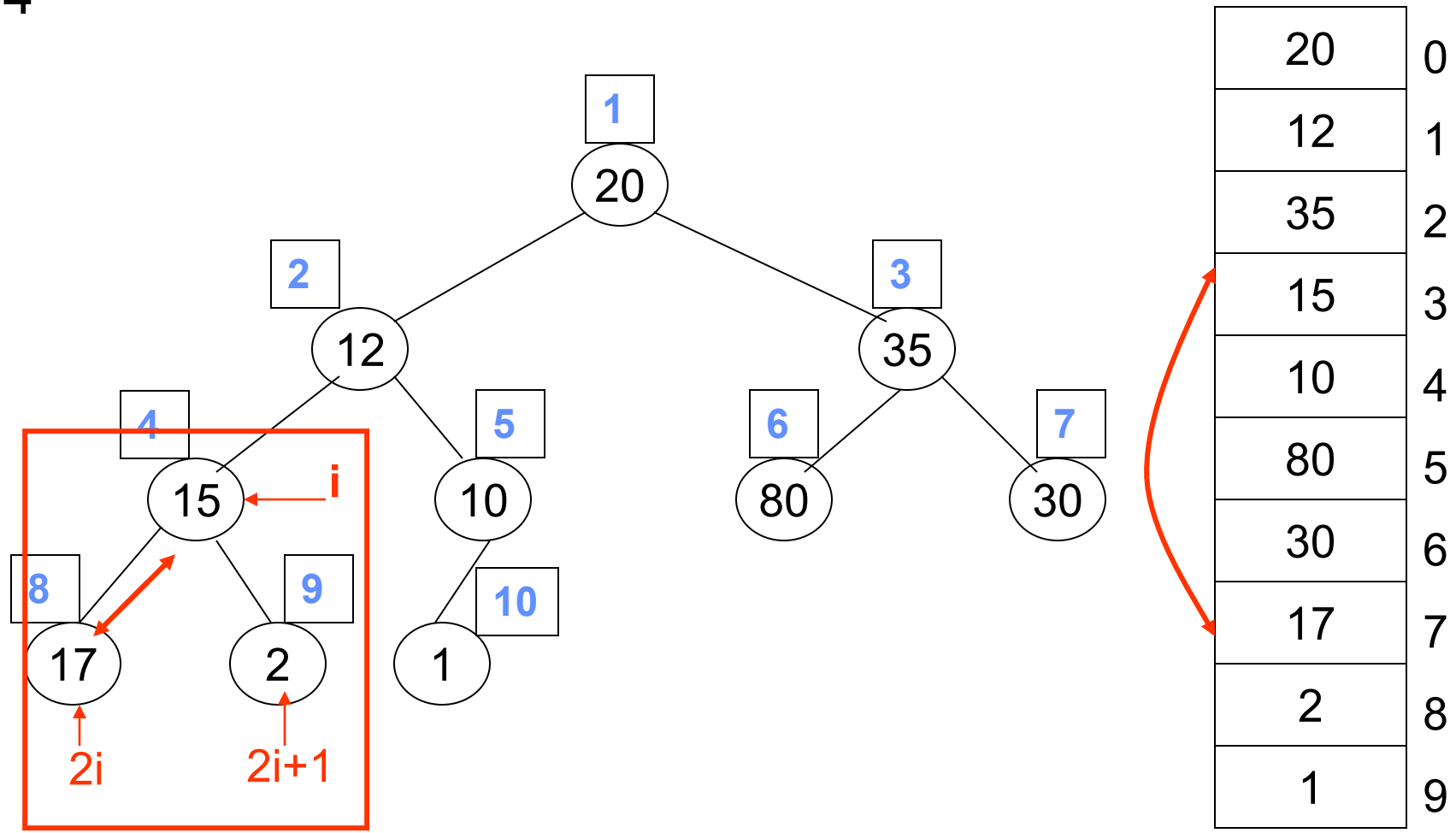
已知n=10; i=n/2=5;



20	0
12	1
35	2
15	3
10	4
80	5
30	6
17	7
2	8
1	9

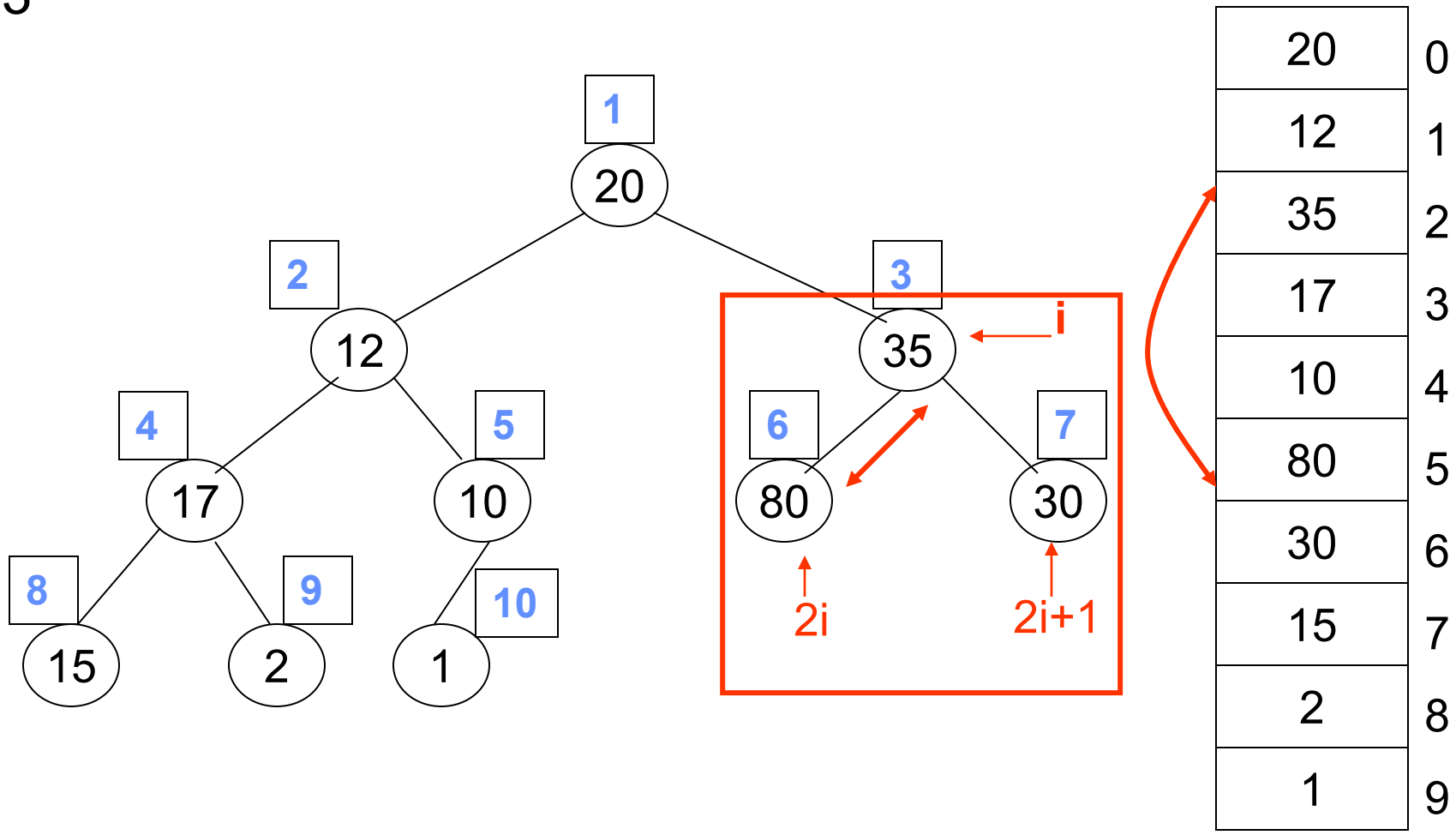
最大堆的初始化step2

i=4



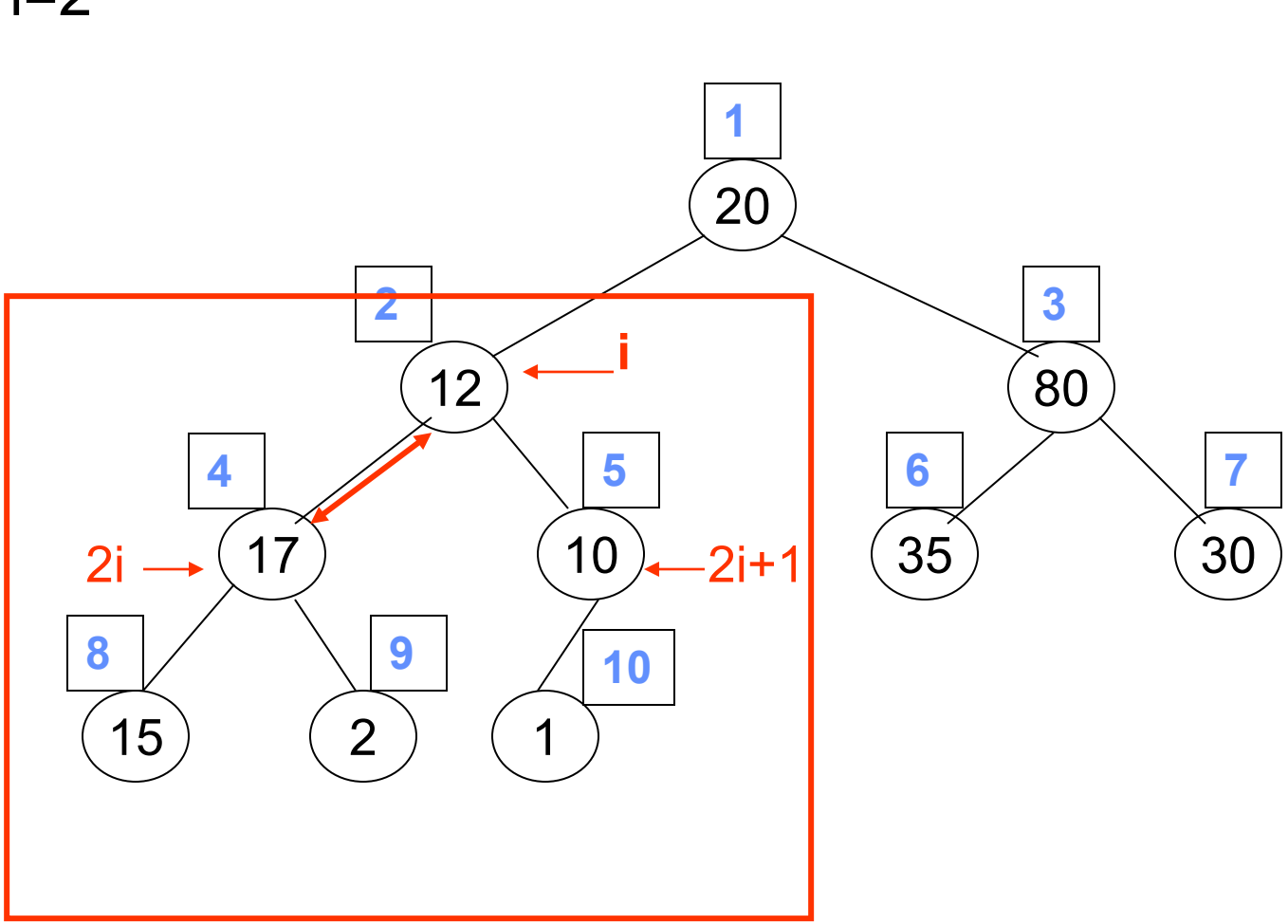
最大堆的初始化step3

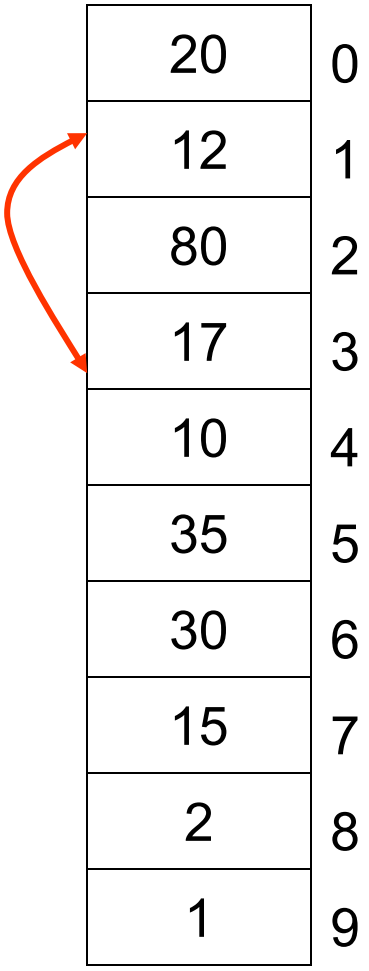
i=3



最大堆的初始化step4_0

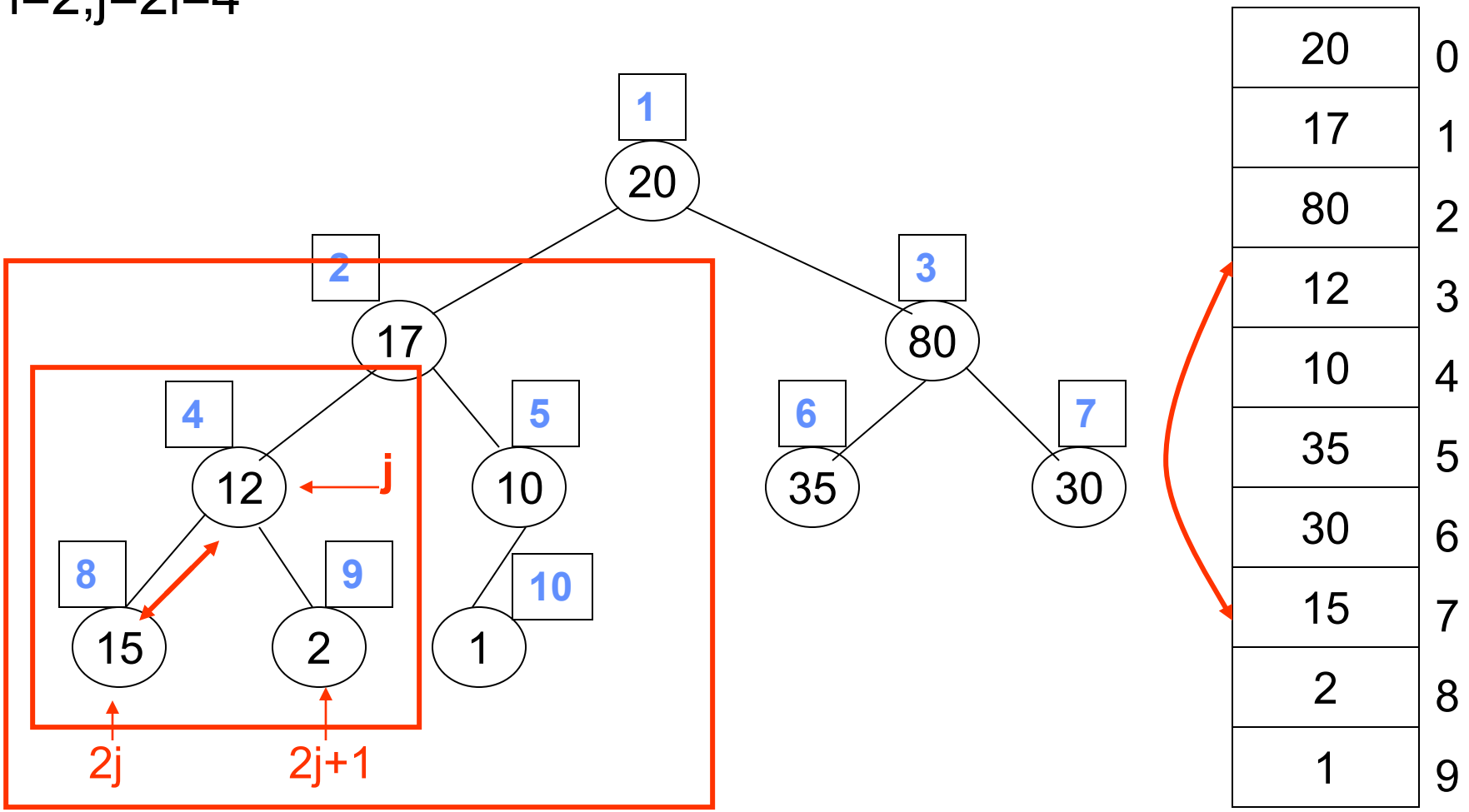
i=2





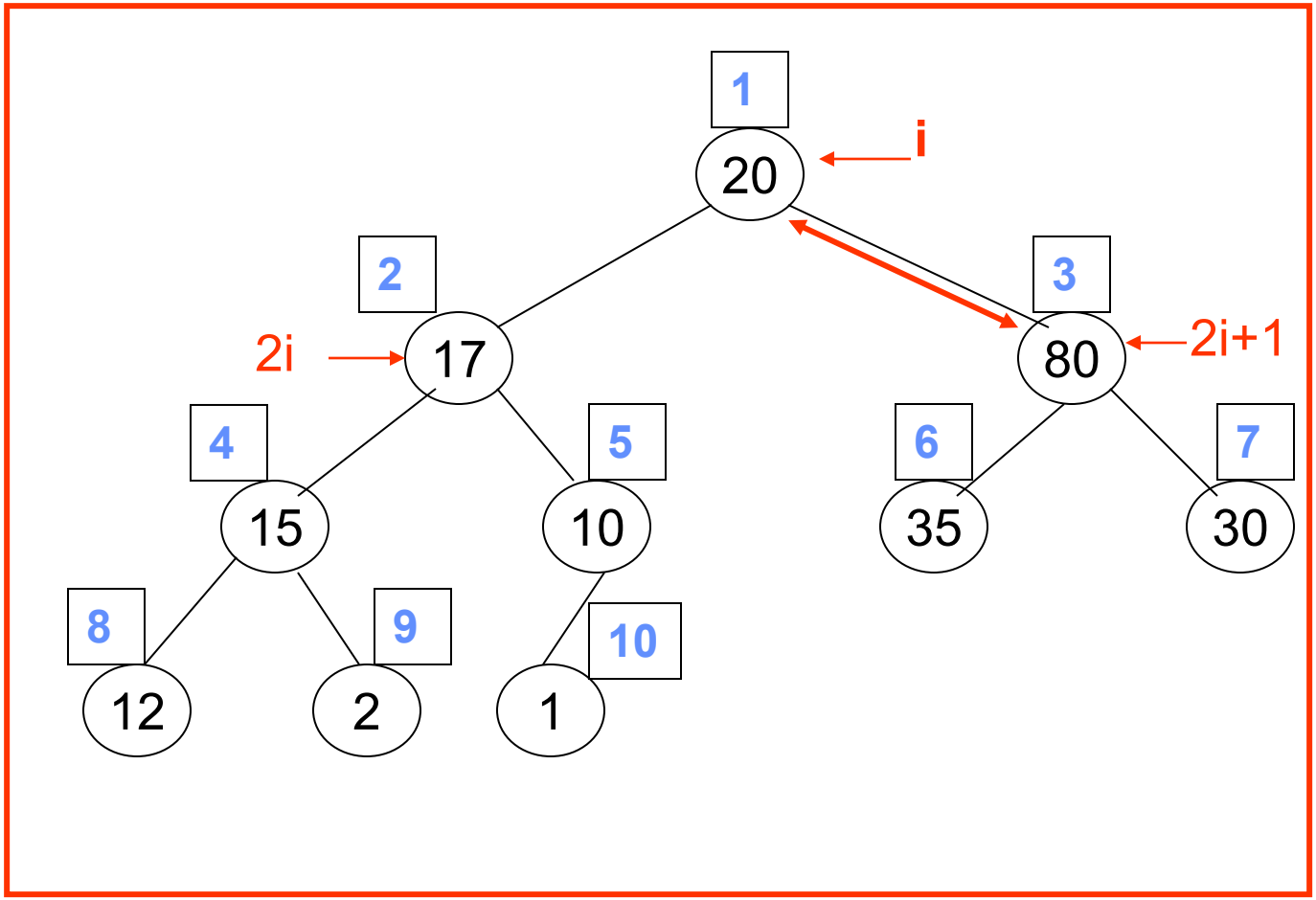
最大堆的初始化step4_1

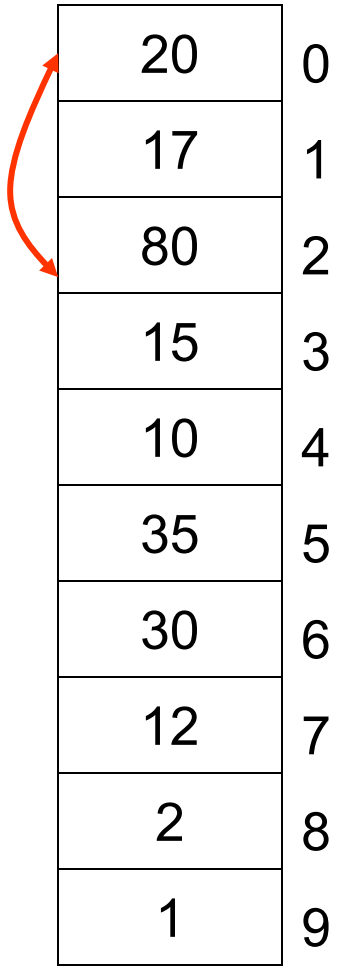
i=2,j=2i=4



最大堆的初始化step5_0

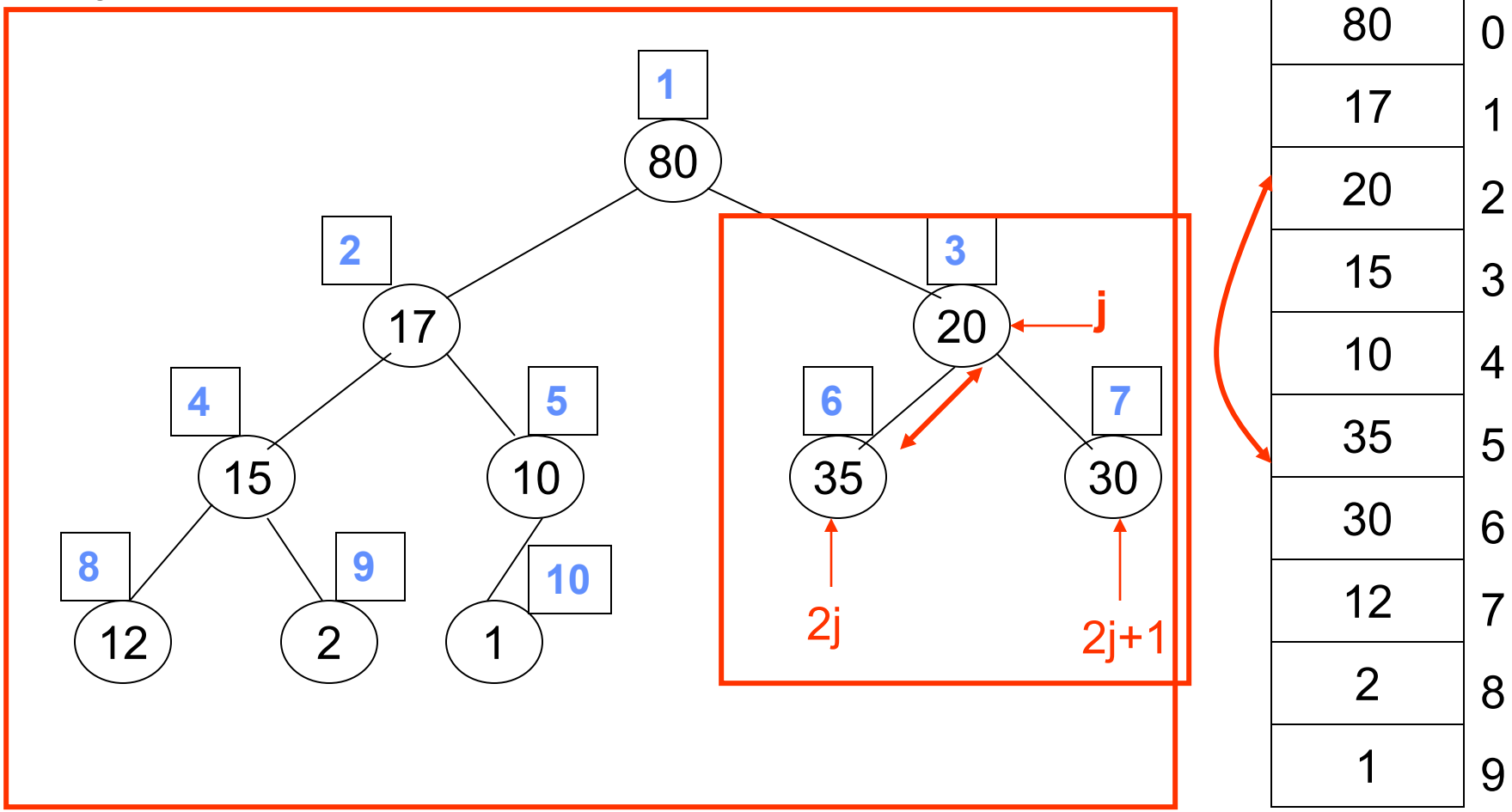
i=5



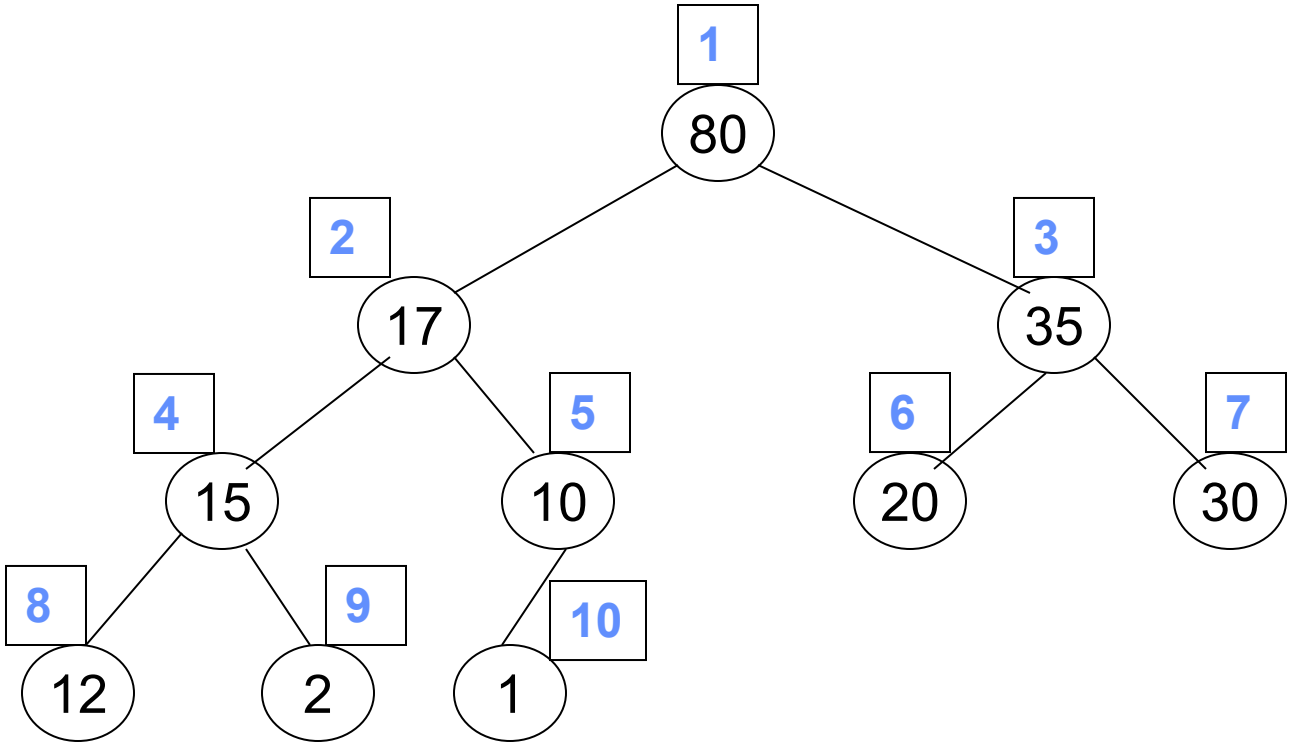


最大堆的初始化step5_1

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



最大堆的初始化step5_2



80	0
17	1
35	2
15	3
10	4
20	5
30	6
12	7
2	8
1	9

```
typedef SqList HeapType;
void  HeapAdjust( HeapType &H, int s, int m)
{
    rc = H.r[s];
    for( j=2*s; j<=m; j*=2){
        if( j<m && H.r[j].key < H.r[j+1].key)  ++j;
        if( rc.key >= H.r[j].key)  break;
        H.r[s] = H.r[j]; s = j;
    }
    H.r[s] = rc;
}
```

```
void HeapSort( HeapType &H )
{
    for( i=H.length/2; i>0; --i)  //建立初始堆
        HeapAdjust( H, i, H.length);
    for( i=H.length; i>1; --i){
        H.r[1] ←→ H.r[i];
        HeapAdjust( H, 1, i-1 );
    }
}
```

■算法分析

◆深度为k的堆，筛选算法中进行的关键字比较次数至多为2(k-1)。建初始堆进行了⌊n/2⌋次筛选，关键字比较次数至多为：

$$\sum_{i=h-1}^{1} 2^{i-1}\cdot 2(h-i)=\sum_{i=h-1}^{1} 2^{i}\cdot (h-i)=\sum_{j=1}^{h-1} 2^{h-j}\cdot j\le (2n)\sum_{j=1}^{h-1}\frac{j}{2^{j}}\le 4n$$

◆n个关键字，完全二叉树的深度$\lfloor \log_2 n\rfloor+1$。调整建立新堆时，调用HeapAdjust过程n-1次，关键字比较次数至多为：

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

◆堆排序的时间复杂度为$O(n\log_2 n)$，空间复杂性为O(1).

◆堆排序是一个不稳定的排序方法。

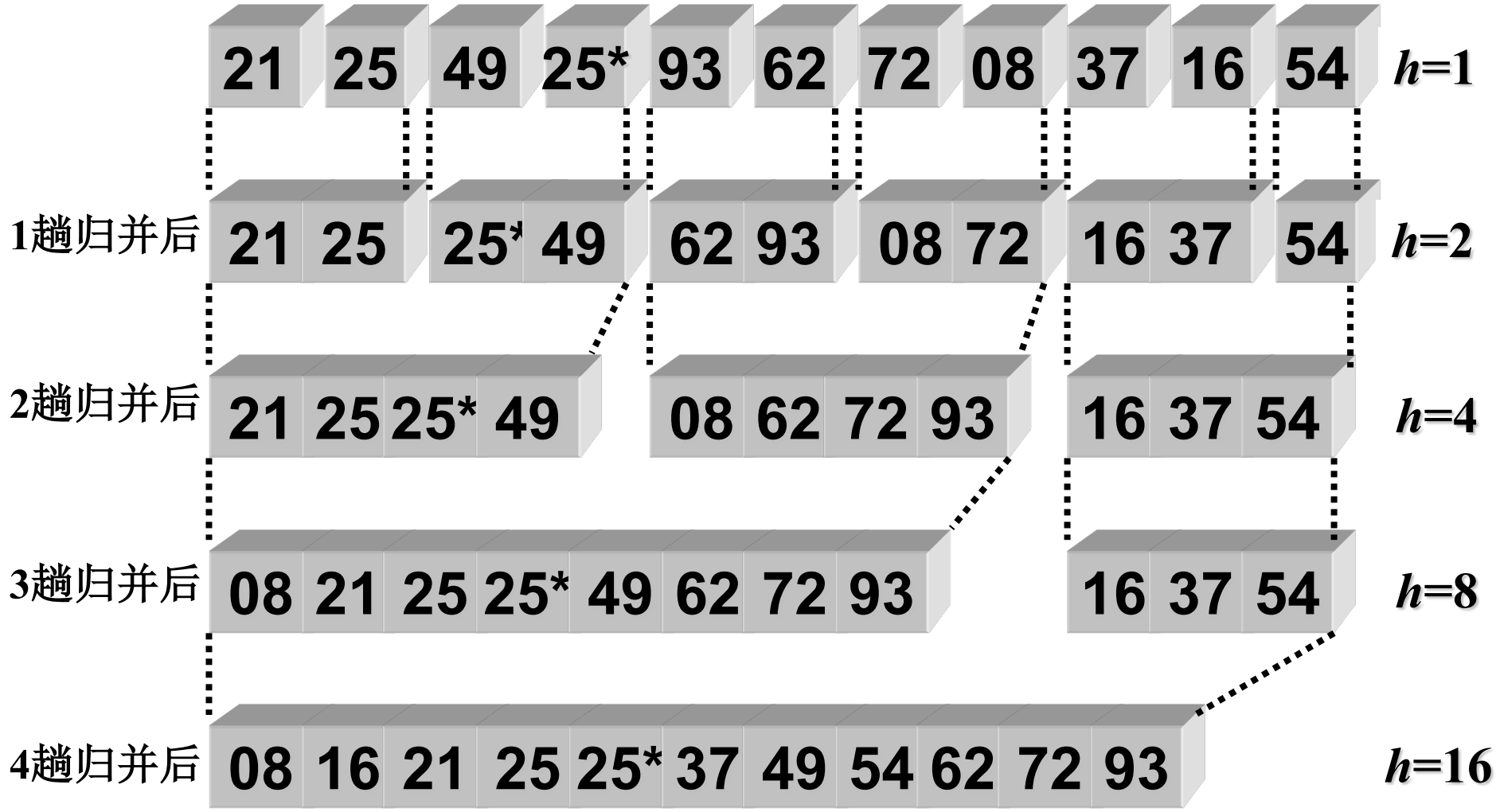
- 作业：

  10.12

10.5 归并排序 (Merging Sort)

- 归并: 是将两个或两个以上的有序表合并成一个新的有序表。
- 2-路归并：假设初始序列有 n 个记录，首先把它看成是 n 个长度为 1 的有序子序列 (归并项)，先做两两归并，得到$\lceil n/2 \rceil$个长度为 2 的归并项 (如果 n 为奇数，则最后一个有序子序列的长度为 1)；再做两两归并，…，如此重复，最后得到一个长度为 n 的有序序列。

采用2-路归并排序方法进行排序的过程 (11个记录)。



一趟归并进行$\lceil n/(2*h) \rceil$次两个有序子表的归并操作Merger.

将有序的SR[i..m]和SR[m+1..n]归并为有序的TR[i..n].

```
void Merge(RcdType SR[], RcdType &TR[], int i, int m, int n)
{
    for( j=m+1, k=i; i<=m && j<=n; ++k){
        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];
    if( j<=n)   TR[k..n] = SR[j..n];
}
```

有序子表长度分别为：n,m. 则Merge的时间复杂度为：O(n+m).

迭代的归并排序算法（一趟归并排序的情形）

```
void MergePass(RcdType SR[], RcdType &TR[], int h, int n)
{
   for (i=1; i+2*h-1<=n; i=i+2*h)  //归并h长的两相邻子表
         Merge(SR, TR, i, i+h-1, i+2*h-1);
   if ( i+h-1<=n )  //余下部分
        Merge(SR, TR, i, i+h-1, n);
}
```

Merge调用⌈$n/(2*h)$⌉次

```
void MergeSort( SqList &L)   //自底向上的二路归并算法
{   RcdType TR[];
    for( h=1; h<L.length; h=2*h)
        { MergePass(L.r, TR, h, L.length);
          L.r[1..L.length] = TR[1..L.length];
        }
}
```

MergePass调用⌈$\log_2 n$⌉次

算法总的时间复杂度: $O(n\log_2 n)$

递归的归并排序算法

```
void MSort(RcdType SR[], RcdType &TR1[], int s, int t)
{
   if( s==t)  TR1[s] = SR[s];
   else{
        m = (s+t)/2;
        MSort( SR, TR2, s, m);
        MSort( SR, TR2, m+1, t);
        Merge( TR2, TR1, s, m, t);
   }
}

 void MergeSort( SqList &L)
{
     MSort( L.r, L.r, 1, L.length);
}
```

算法分析

- 在迭代的归并排序算法中，函数 *MergePass*( ) 做一趟两路归并排序，要调用*merge* ( )函数$\lceil n/(2*h) \rceil$ 次，函数*MergeSort*( )调用*MergePass*( )正好$\lceil \log_2 n \rceil$次，而每次*merge*( )至多执行比较2*h*次，所以算法总的时间复杂度为$O(n\log_2 n)$。
- 递归的归并排序方法的递归深度为$\lceil \log_2 n \rceil$，算法总的时间复杂度为$O(n\log_2 n)$。
- 归并排序占用附加存储较多，需要另外一个与原待排序记录数组同样大小的辅助数组。$O(n)$ 这是这个算法的缺点。
- 归并排序是一个稳定的排序方法。

10.6 基数排序 (Radix Sort)

基数排序是通过“分配”和“收集”过程来实现排序, 是一种借助于多关键字排序的思想对单关键字排序的方法。

1. 多关键字排序

- 每张扑克牌有两个“关键码”：花色和面值。其有序关系为：
  - 花色：♣ < ♦ < ♥ < ♠
  - 面值：2 < 3 < 4 < 5 < 6 < 7 < 8 < 9 < 10 < J < Q < K < A
- 所有扑克牌排成以下次序：

  ♣ 2, …, ♣ A, ♦ 2, …, ♦ A, ♥ 2, …, ♥ A, ♠ 2, …, ♠ A

- 有 n 个记录的序列 $\{R_1, R_2, \ldots, R_n\}$，且每个记录$R_i$ 中含有 d 个关键字

$$\left(K_i^1,\ K_i^2,\ \cdots,\ K_i^d\right)$$

序列中任意两个对象 R_i 和 R_j ($1 \le i < j \le n$) 都满足：

$$\left(K_i^1,\ K_i^2,\ \cdots,\ K_i^d\right) < \left(K_j^1,\ K_j^2,\ \cdots,\ K_j^d\right)$$

则称序列对关键字 $(K^1, K^2, \ldots, K^d)$ 有序。其中，K^1 称为最主位关键字，K^d 称为最次位关键字。

- 实现多关键字排序有两种常用的方法
  - 最高位优先**MSD (Most Significant Digit first)**
  - 最低位优先**LSD (Least Significant Digit first)**

- 最高位优先法：
  - 先根据最高位关键字K^1排序,得到若干组，每组中每个记录都有相同关键字K^1。
  - 再分别对每组中记录根据关键字K^2进行排序，按K^2值的不同，再分成若干个更小的子组，每个子组中的记录具有相同的K^1和K^2值。
  - 依此重复，直到对关键字K^d完成排序为止。
  - 最后，把所有子组中的记录依次连接起来，就得到一个有序的序列。
- 最低位优先法：首先依据最低位关键字K^d对所有记录进行一趟排序，再依据次低位关键字K^{d-1}对上一趟排序的结果再排序，依次重复，直到依据关键字K^1最后一趟排序完成，就可以得到一个有序的序列。使用这种排序方法对每一个关键字进行排序时，不需要再分组，而是整个记录都参加排序。

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 。

2. 链式基数排序

- 基数排序是典型的LSD排序方法，利用“分配”和“收集”两种运算对单关键字进行排序。在这种方法中，把单关键字 K_i 看成是一个*d*元组：

$$\left(K_i^1,\ K_i^2,\ \cdots,\ K_i^d\right)$$

- 分量有*radix*种取值，则称*radix*为基数，即分量的取值范围。

  关键字984可以看成是一个3元组(9, 8, 4)，每一位有0, 1, …, 9等10种取值，基数*radix* = 10。关键字‘data’可以看成是一个4元组(d, a, t, a)，每一位有‘a’, ‘b’, …, ‘z’等26种取值，*radix* = 26。

- 记录的关键字$K^0, K1, \ldots, K^{d-1}$，依次对各位的分量，分别用“分配”、“收集”的运算逐趟进行排序，
- 各队列采用链式队列结构，分配到同一队列的关键字用指针链接起来。每一队列设置两 个指针: ***front*** [***radix***]指示队头，

  rear [***radix***] 指向队尾。
- 以静态链表作为***n***个记录的存储结构。在记录重排时不必移动记录，只需修改各记录的链接指针即可。

例，序列　278　109　063　930　589　184　505　269　008　083

第一趟分配

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

第一趟收集　930　063　083　184　505　278　008　109　589　269

第二趟分配

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

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

算法分析

- 若每个关键字有*d* 位，需要重复执行*d* 趟“分配”与“收集”。每趟对*n*个记录进行“分配”，对*radix*个队列进行“收集”。总时间复杂度为 $O(d(n+radix))$。
- 若基数*radix*相同，对于记录个数较多而关键字位数较少的情况，使用链式基数排序较好。
- 基数排序需要增加 $n+2*radix$ 个附加链接指针。
- 基数排序是稳定的排序方法。

- 作业
  - **10.1**

10.7 各种内部排序方法的比较讨论

1. 选择排序方法时需考虑的因素

- 待排序的记录数目；
- 记录本身信息量的大小；
- 关键字的结构及其分布情况；
- 对排序稳定性的要求；
- 语言工具的条件，辅助空间的大小。

2. 各种内部排序方法的性能

排序方法	比较次数		移动次数		稳定性	附加存储	
	最好	最差	最好	最差		最好	最差
直接插入排序	n	n^2	0	n^2	√	1	
折半插入排序	$n \log_2 n$		0	n^2	√	1	
起泡排序	n	n^2	0	n^2	√	1	
快速排序	$n\log_2 n$	n^2	$n \log_2 n$	n^2	×	$\log_2 n$	n
简单选择排序	n^2		0	n	×	1	
锦标赛排序	$n \log_2 n$		$n \log_2 n$		√	n	
堆排序	$n \log_2 n$		$n \log_2 n$		×	1	
归并排序	$n \log_2 n$		$n \log_2 n$		√	n	

3. 结论 没有哪一种排序方法是绝对好的，都有其优缺点

- 若n较小，可采用直接插入排序或简单选择排序
- 若序列的初始状态已按关键字基本有序，则选用直接插入或起泡排序为宜；
- 若n较大，采用$O(n\log_2 n)$的排序方法；
- 若n很大，记录的关键字位数较少且可以分解时，采用基数排序较好；
- 避免移动记录，用链表作存储结构，如表插入；
- 内部排序可能达到的最快速度是什么？时间下界？

  时间上界$O(n^2)$

  任何借助于“比较”的排序方法，至少需要$O(n\log_2 n)$的时间。

- 三个关键字：k1, k2, k3，则描述3个记录排序过程的判定树必有3!=6个叶子结点。
- n个记录的序列，初始状态有n!个，则描述n个记录排序过程的判定树必有n!个叶子结点。则判定树的树高至少为 $\lceil log_2 n! \rceil + 1$，$\log_2 n! \approx n\log_2 n$，
- 最坏情况下能达到的最好的时间复杂度为$O(n\log_2 n)$.

练习

1. 在堆排序、起泡排序、简单选择排序和快速排序中，时间复杂度不受初始状态影响，恒为$O(n\log_2 n)$的是：堆排序

恒为$O(n^2)$的是：简单选择排序

2. 在快速排序、希尔排序、起泡排序、堆排序中，当待排序的记录已经为有序时，花费时间最多的是：快速排序

3. 数据表中有5000个元素，如仅要求找出其中最大的10个元素，采用哪种排序方法最省时？

A 快速排序　B 堆排序　C 归并排序　D 基数排序

4. {13, 2, 17, 31, 9, 27, 5, 10, 20, 6, 18},

①写出增量为5的第一趟希尔排序的结果；

②写出以第一元素为枢轴的快速排序第一趟结果；

③判断该序列是否为堆，若不是画出调整其为小顶堆的过程；

④按照此序列顺序，画出二叉排序树的建立过程；

①{ 13, 2, 10, 20, 6, 18, 5, 17, 31, 9, 27}

② {6, 2, 10, 5, 9} 13 {27, 31, 20, 17, 18}

③小顶堆：**{2, 6, 5, 10, 9, 27, 17, 31, 20, 13, 18}**

④

5. {24, 19, 32, 43, 38, 6, 13, 22},

①写出快速排序第一趟的结果；

②堆排序时所建的初始堆；

③2-路归并排序的全过程；

④上述三种排序方法中，哪一种方法使用的辅助空间最少？

⑤在最坏情况下，哪种方法的时间复杂度最差？

①{ 22, 19, 13, 6, 24, 38, 43, 32}

②小顶堆：{6, 19, 13, 22, 38, 32, 24, 43}

大顶堆：{43, 38, 32, 22, 24, 6, 13, 19}

④辅助空间：堆O(*1*)，快速O(*logn*)， 归并O(*n*).

⑤堆O(*nlogn*)，快速O(n^2)，归并 O(*nlogn*).

第11章 外部排序

- 外部排序相对于内部排序而言，适用于大规模的数据排序，由于内存中一次不能容纳下全部数据，每次只能读入部分数据，其余数据依然在外存上存储，因此在外部排序的过程中需要进行多次内、外存之间的数据交换。

外部信息的存取

- 计算机的存储器分两种
  - 内存：随机访问，速度快，价格高，容量小
  - 外存(辅存)：顺序/随机访问，速度慢，价格低，容量大
- 外部存储器
  - 磁带：顺序存取设备
  - 磁盘：随机存取设备

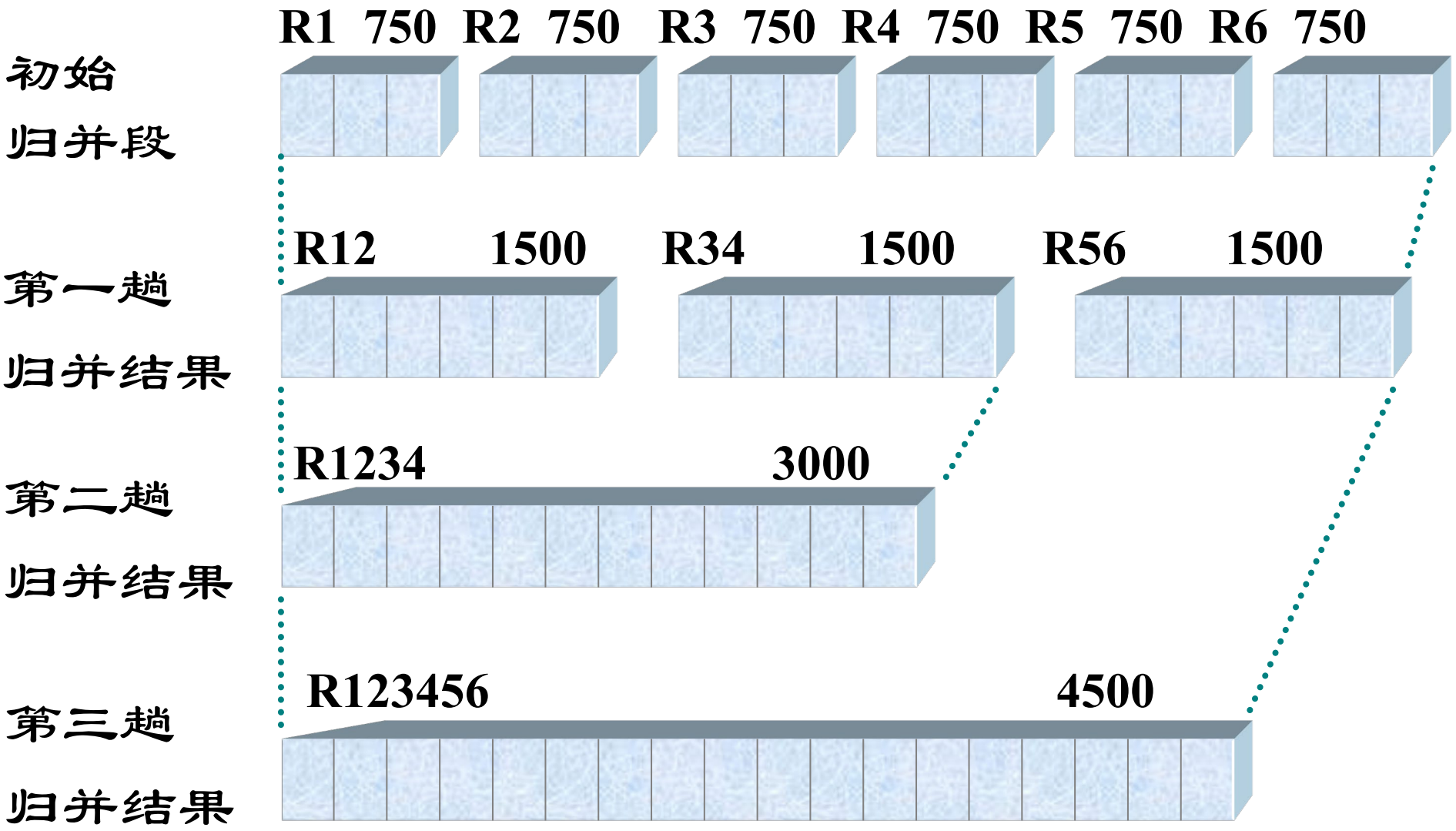
外部信息的存取

- 磁带
  - 读取数据时只能顺序读取，定位记录所花时间过多，信息检索和修改不方便。
  - 因此磁带主要用于处理变化少，只进行顺序存取大量数据的场合。
- 磁盘
  - 磁盘是一种直接存取设备，相对于磁带来说存取任意一个信息的时间变化不大。

- 外部排序过程主要分为两个阶段：
  - 第一个阶段：按内存大小将输入文件划分为若干段，用某种内排序方法对各段进行排序。初始归并段或初始顺串 (Run)。当它们生成后就被写到外存中去。
  - 第二个阶段：初始归并段加以归并，一趟趟地扩大归并段和减少归并段个数，直到最后归并成一个大归并段(有序文件)为止。

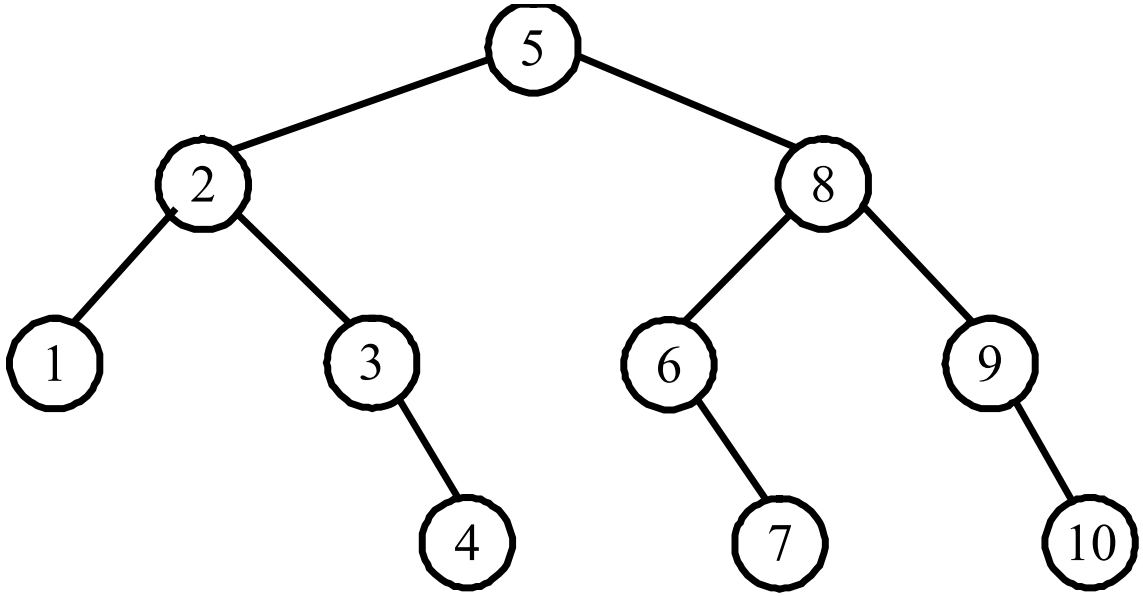
- 示例：含4500个记录的输入文件。内存至多可容纳750个记录的计算机。磁盘每块可容纳250个记录，全部记录可存储在 4500 / 250＝18 个块中。
- 内存中恰好能存3个块的记录。
- 第一阶段，把18块记录，每3块一组，读入内存。利用某种内排序方法进行内排序, 形成初始归并段, 再写回外存。
- 第二阶段，得到6个初始归并段。然后一趟一趟进行归并排序。

两路归并排序的归并树


R1 750
R2 750
R3 750
R4 750
R5 750
R6 750
初始
归并段
R12 1500
R34 1500
R56 1500
第一趟
归并结果
R1234 3000
第二趟
归并结果
R123456 4500
第三趟
归并结果

- 外部排序的优化思路：
  - 目标是减少总读写磁盘次数
  - 通过增大归并路数，减少归并趟数实现

- 9.3 画出对长度为10的有序表进行折半查找的判定树，并求等概率时查找成功的平均查找长度。

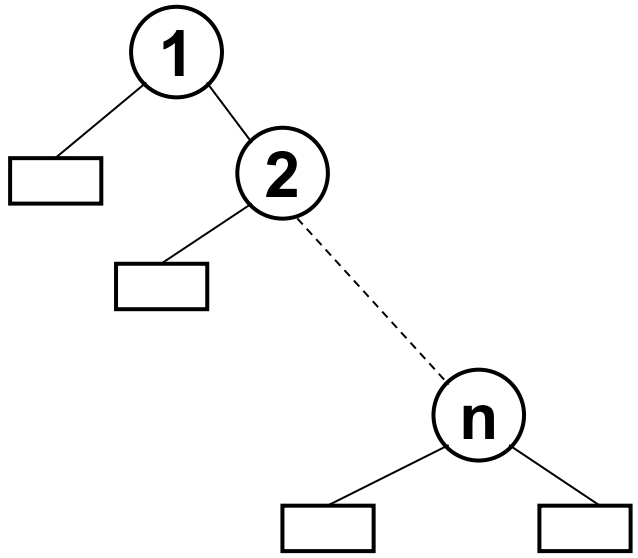


$$ASLsucc=\frac{1\times1+2\times2+4\times3+3\times4}{10}=2.9$$

- 9.25 假设顺序表按关键字自大至小有序，试改写教科书9.1.1节中的顺序查找算法，将监视哨设在高下标端。然后画出此查找过程的判定树，分别求出等概率情况下查找成功和不成功时的平均查找长度。

9.25

```
int Search_Sq(SSTable ST, int key)
{
    ST.elem[ST.length+1].key=key;
    for(i=1; ST.elem[i].key > key; i++);
    if( i>ST.length || ST.elem[i].key<key)
        return 0;
    return i;
}
```



$$ASL_{succ} = \frac{1}{n}\sum_{i=1}^{n} i = \frac{n+1}{2}$$

$$ASL_{unsucc} = \frac{1}{n+1}\left(\sum_{i=1}^{n} i + (n+1)\right) = \frac{n+2}{2}$$