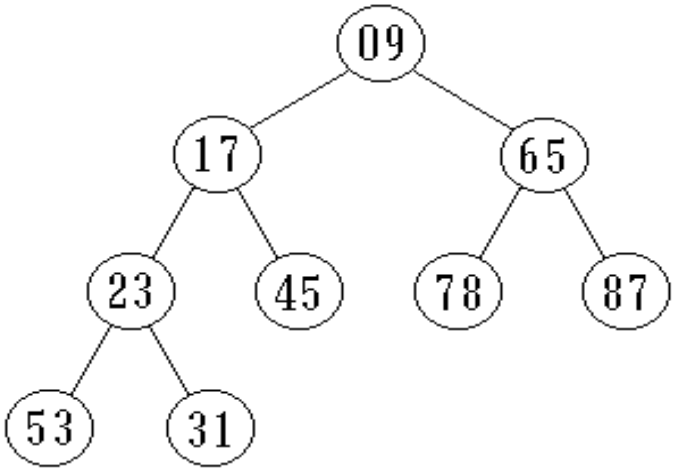
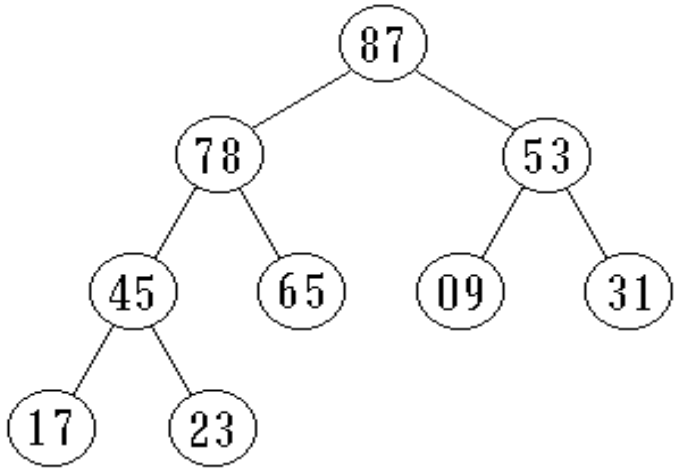
10.4 选择排序

10.4.3 堆排序(Heap Sort)

- 堆的定义：N个元素的序列($K_1,K_2,...,K_n$)当且仅当满足关系$K_i \le K_{2i}$ && $K_i \le K_{2i+1}$时，称为最小堆（小根堆）； N个元素的序列($K_1,K_2,...,K_n$)当且仅当满足关系$K_i \ge K_{2i}$ && $K_i \ge K_{2i+1}$时，称为最大堆（大根堆）。
- 堆与二叉树有密切的关系。

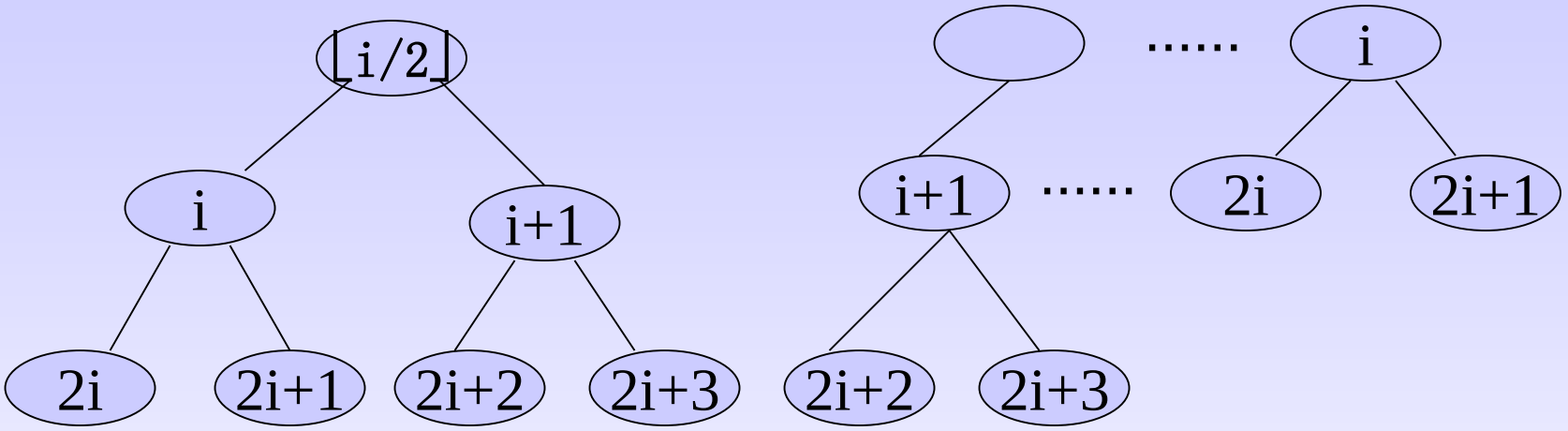


(a) 最小堆



(b) 最大堆

6.2 完全二叉树



(a) i和i+1结点在同一层　　(b) i和i+1结点不在同一层

性质5：如果对一棵有n个结点的完全二叉树的结点按层序编号（从上到下，从左到右),则对任一结点i（1<=i<=n),有：

（1）如果i＝1，则结点i无双亲，是二叉树的根；如果i>1，则其双亲是结点⌊i/2⌋。

（2）如果2i>n，则结点i无左孩子；否则其左孩子是结点2i。

（3）如果2i＋1>n，则结点i无右孩子；否则其右孩子是结点2i＋1。

10.4 选择排序

堆排序的第一个工作是建堆，即把整个记录数组r[1]到r[n]调整为一个堆。显然, 只有一个结点的树是堆，而在完全二叉树中，所有序号i >= ⌊n/2⌋的结点都是叶子，因此以这些结点为根的子树都已是堆。这样，只需依次将序号为⌊n/2⌋，⌊n/2⌋-1，...，1的结点作为根的子树都调整为堆即可。

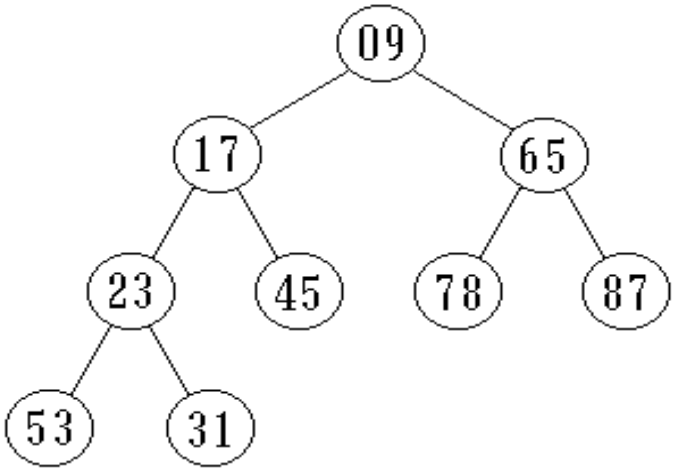
设序列r[s..m]中除第一个元素r[s]外均满足堆定义，函数HeapAdujst(r, s, m)将r[s..m]调整为大顶堆。则上述过程可以表示为：for(i=n/2; i>0; i--) HeapAdjust(r, i, n);

```
void HeapAdujst(SqList &H, int s, int m)
//已知H.r[s..m]中除H.r[s]外均满足堆定义, 本函数将H.r[s..m] 调整为大顶堆.
```

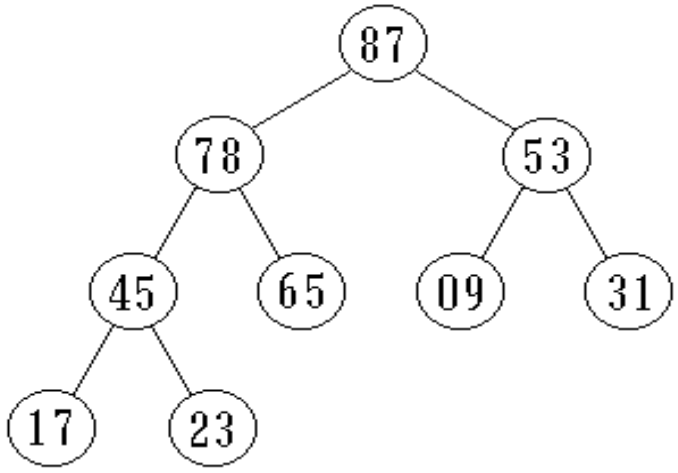
10.4 选择排序

10.4.3 堆排序(Heap Sort)

- 堆的定义：N个元素的序列($K_1,K_2,...,K_n$)当且仅当满足关系$K_i \le K_{2i}$ && $K_i \le K_{2i+1}$时，称为最小堆（小根堆）； N个元素的序列($K_1,K_2,...,K_n$)当且仅当满足关系$K_i \ge K_{2i}$ && $K_i \ge K_{2i+1}$时，称为最大堆（大根堆）。
- 堆与二叉树有密切的关系。



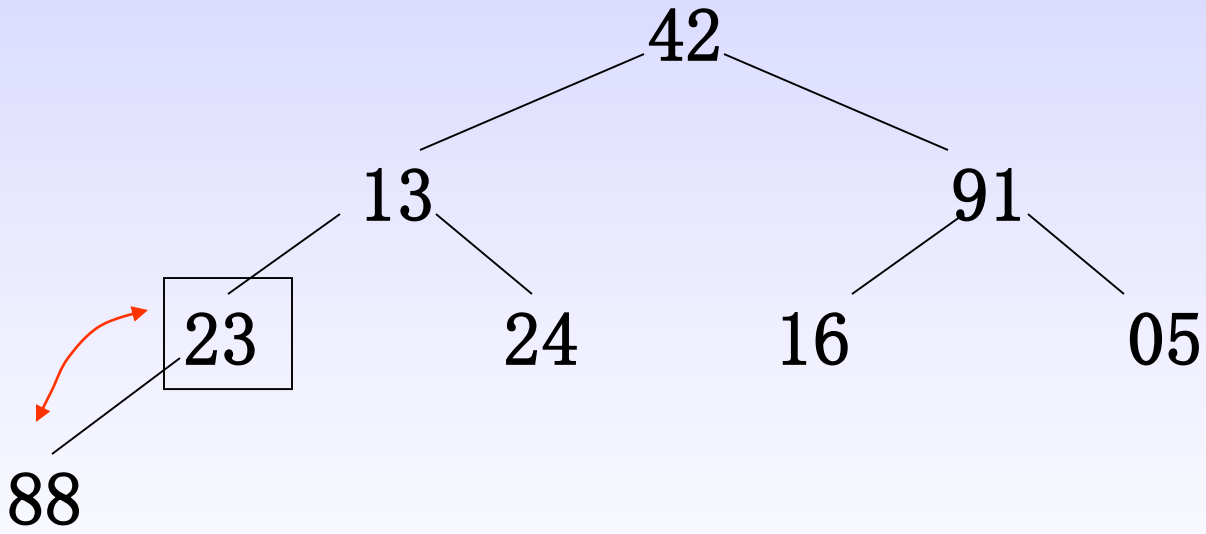
（a）最小堆

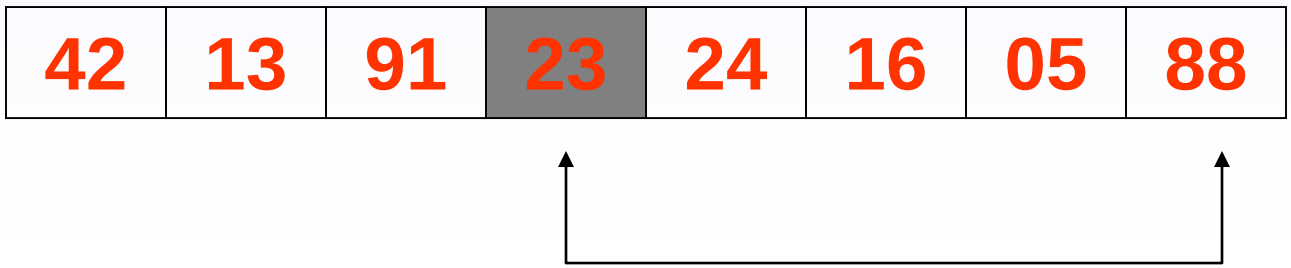


（b）最大堆

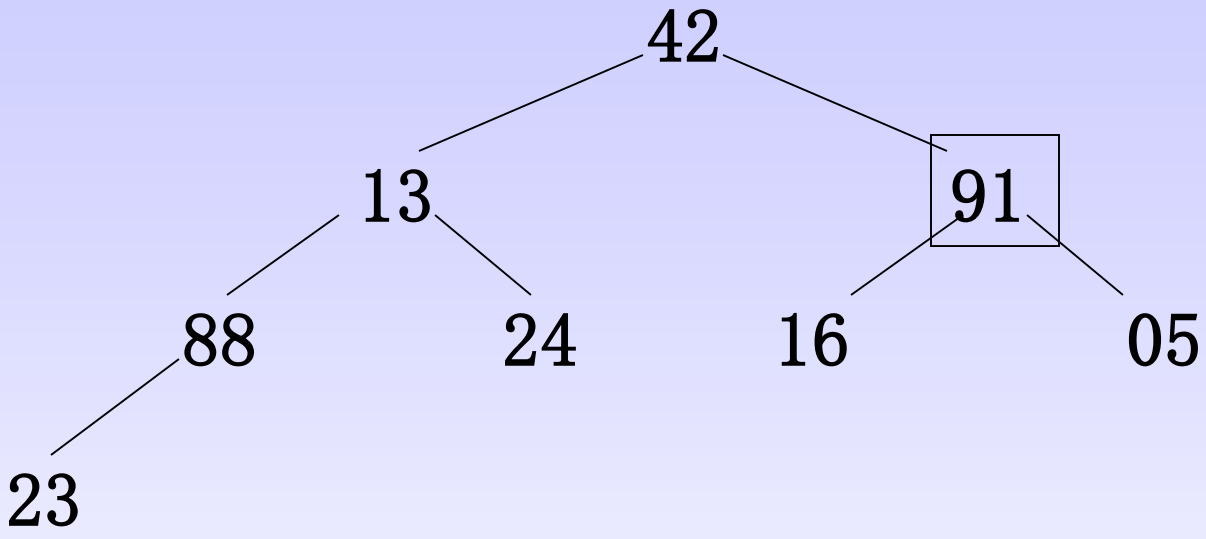
10.4 选择排序

例子：关键字序列为 42，13，91，23， 24，16，05，88，n=8，故从第四个结点开始调整





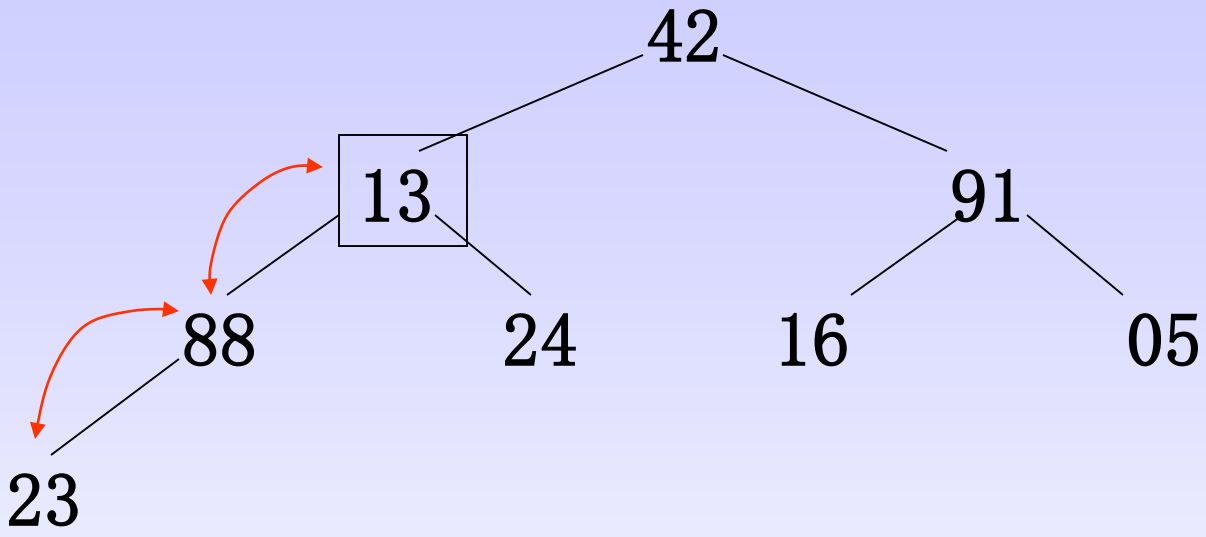
10.4 选择排序

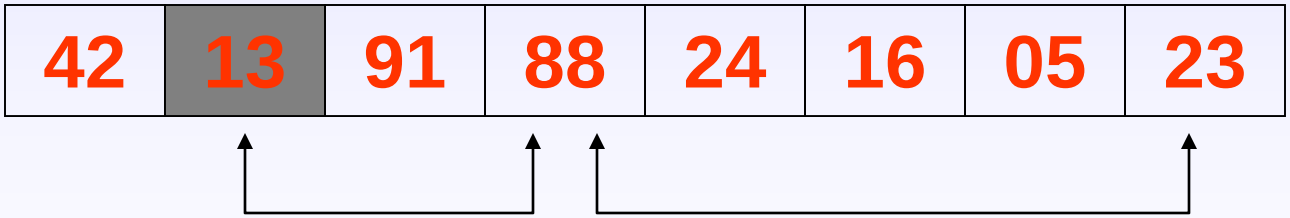


42	13	91	88	24	16	05	23

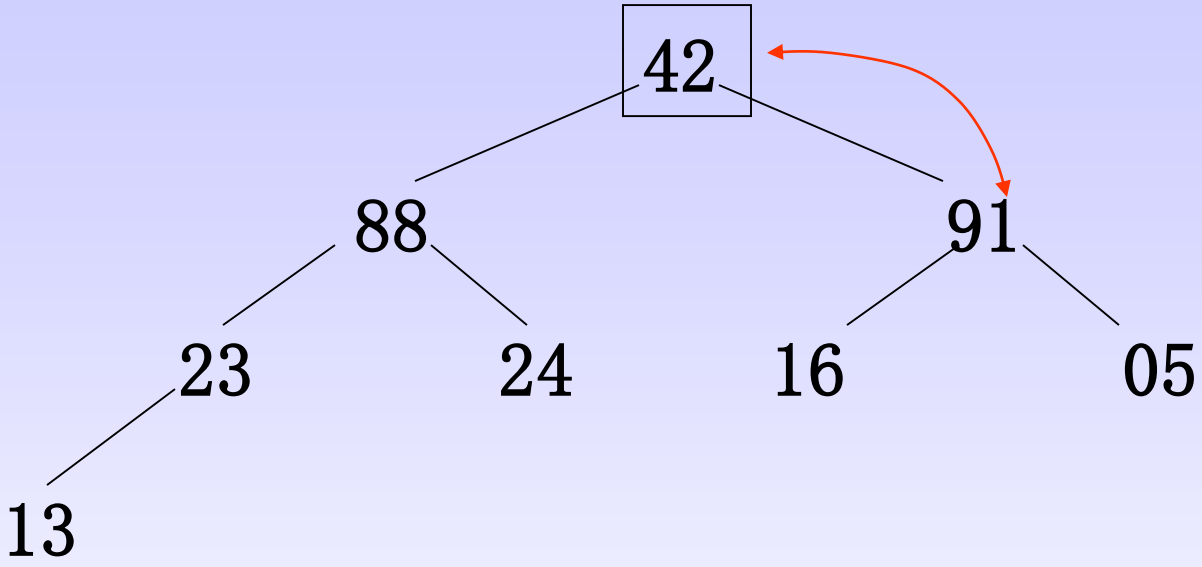
不调整

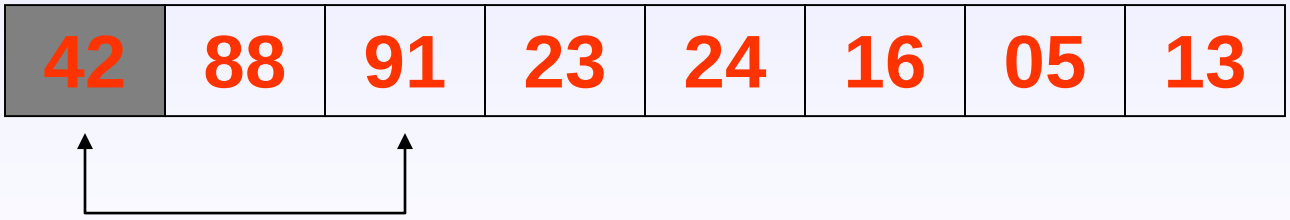
10.4 选择排序



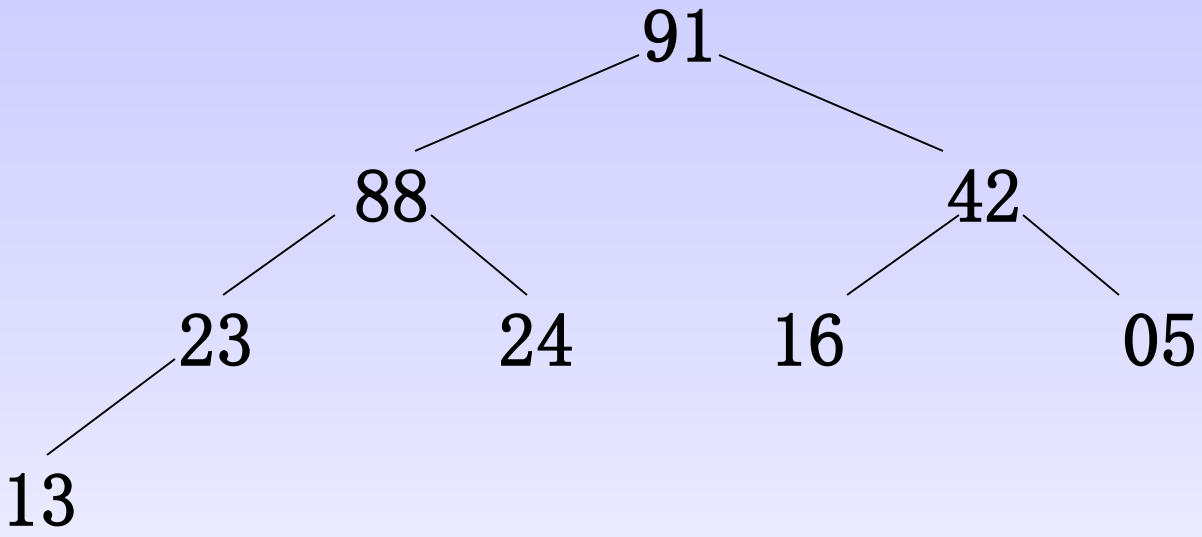


10.4 选择排序





10.4 选择排序

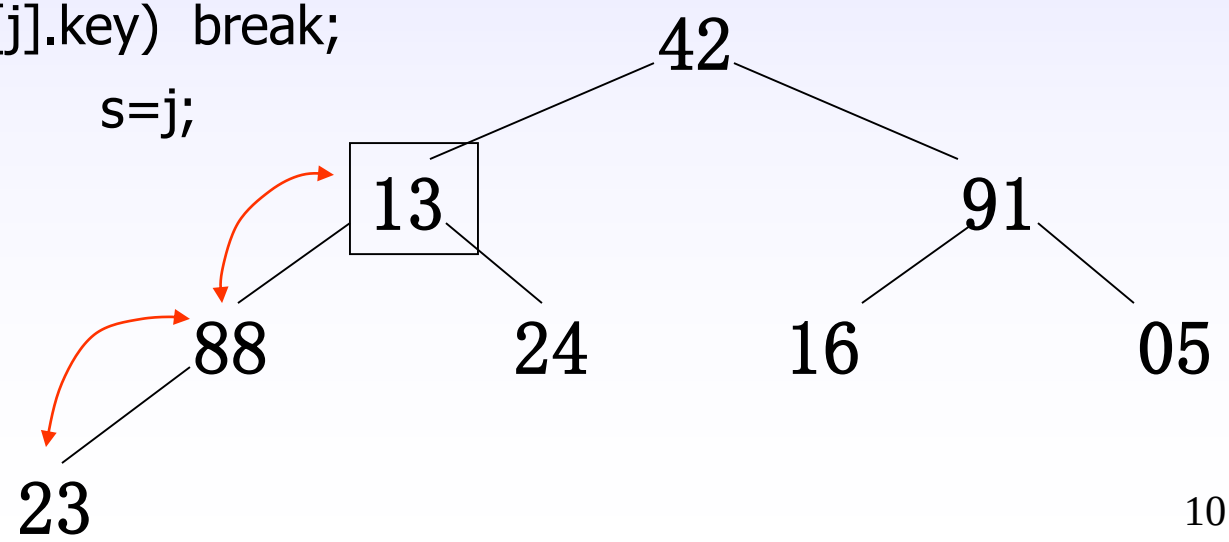


91	88	42	23	24	16	05	13

建成的堆

10.4 选择排序

```
void HeapAdujst(SqList &H, int s, int m)
{  //已知H.r[s..m]中除H.r[s]外均满足堆定义, 本函数将H.r[s..m]调整为大顶堆.
   rc=H.r[s];
   for(j=2*s; j<=m; j*=2)          //沿key较大的子结点向下筛选
   {   if(j<m && H.r[j].key<=H.r[j+1].key)
             j++;                          //j为key较大的记录的下标
       if(rc.key>=H.r[j].key)  break;
       H.r[s]=H.r[j];        s=j;
    }
    H.r[s]=rc;
}
```

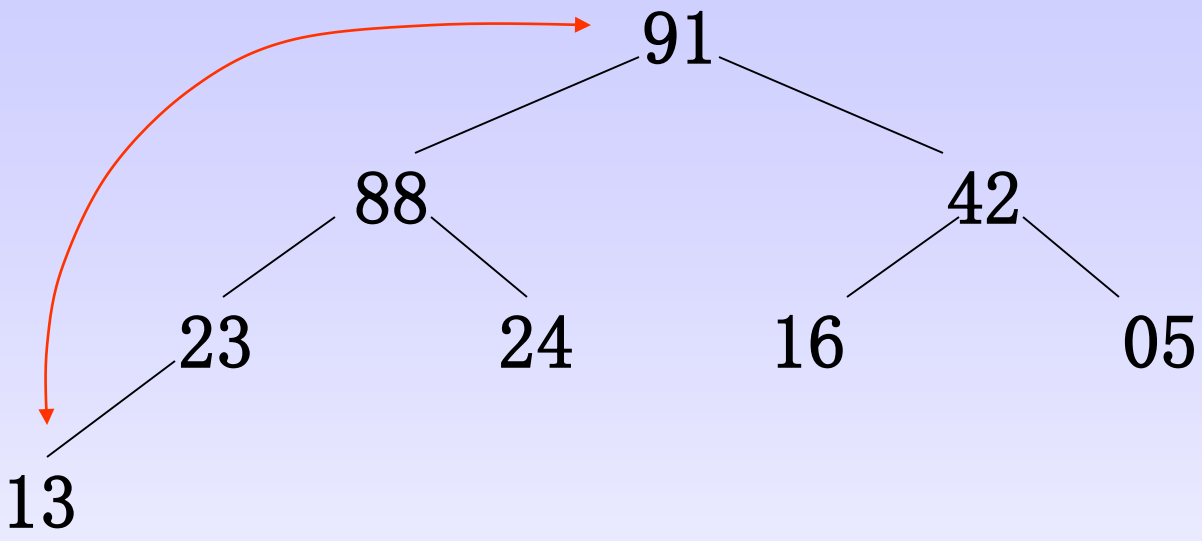


10.4 选择排序

基于最大堆的堆排序

- 堆排序分为两个步骤：第一步，根据初始输入数据，利用堆的调整算法建立初始堆，第二步，通过一系列的元素交换和重新调整堆进行排序。
- 最大堆的第一个元素r[1]具有最大的关键字，将r[1]与r[*n*]对调，把具有最大关键字的元素交换到最后，再对前面的*n*-1个元素，使用堆的调整算法*HeapAdjust*(r, 1, *n*-1)，重新建立最大堆。结果具有次最大关键字的元素又上浮到堆顶，即r[1]位置。再对调r[1]和r[*n*-1]，调用*HeapAdjust*(r, 1, *n*-2)，对前*n*-2个元素重新调整，…。
- 如此反复执行，最后得到全部排序好的元素序列。这个算法即堆排序算法，其细节在下面的程序中给出。

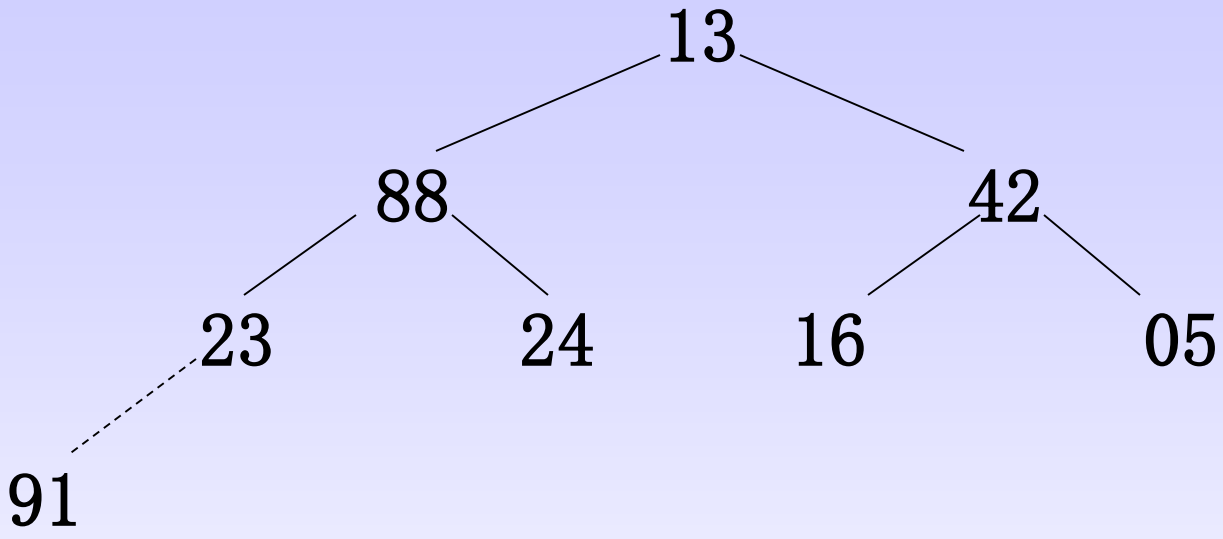
10.4 选择排序



91	88	42	23	24	16	05	13

（1）初始堆R[1]到R[8]

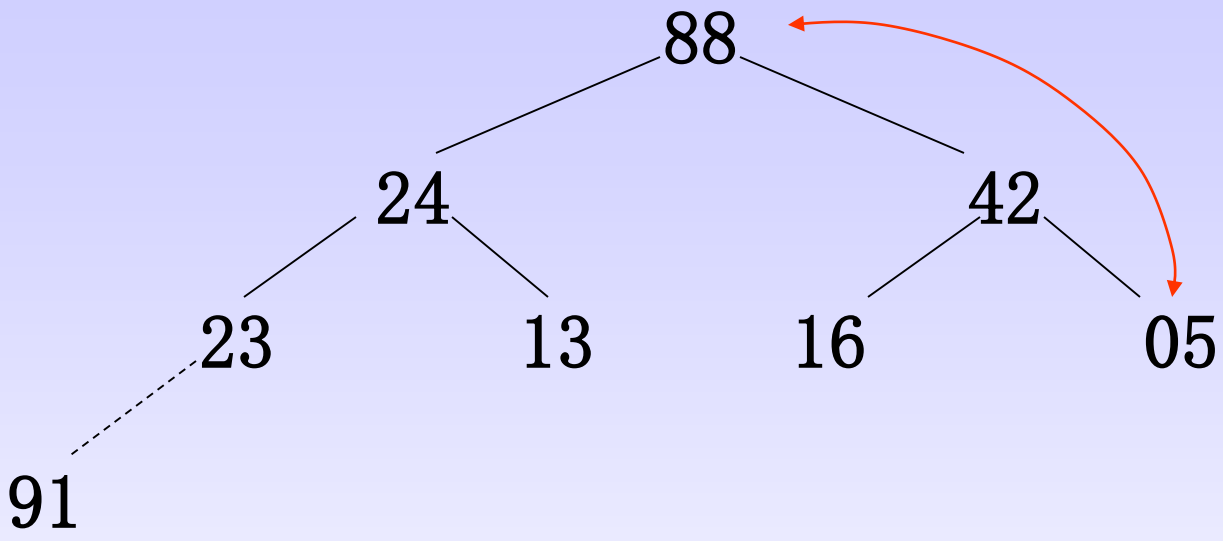
10.4 选择排序



13	88	42	23	24	16	05	91

（2）第一趟排序之后

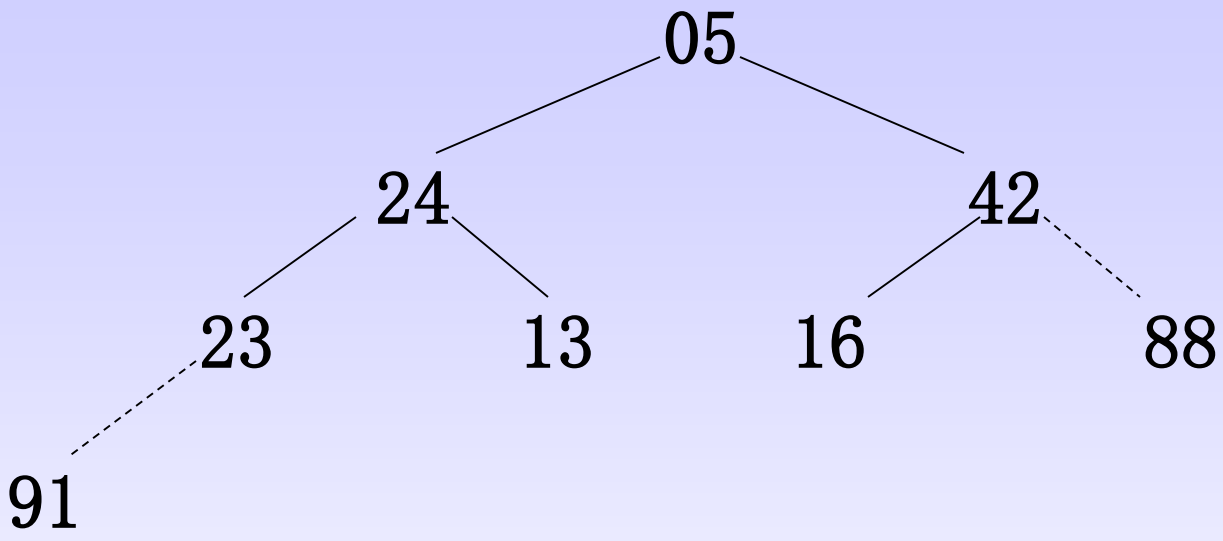
10.4 选择排序



88	24	42	23	13	16	05	91

（3）重建的堆R[1]到R[7]

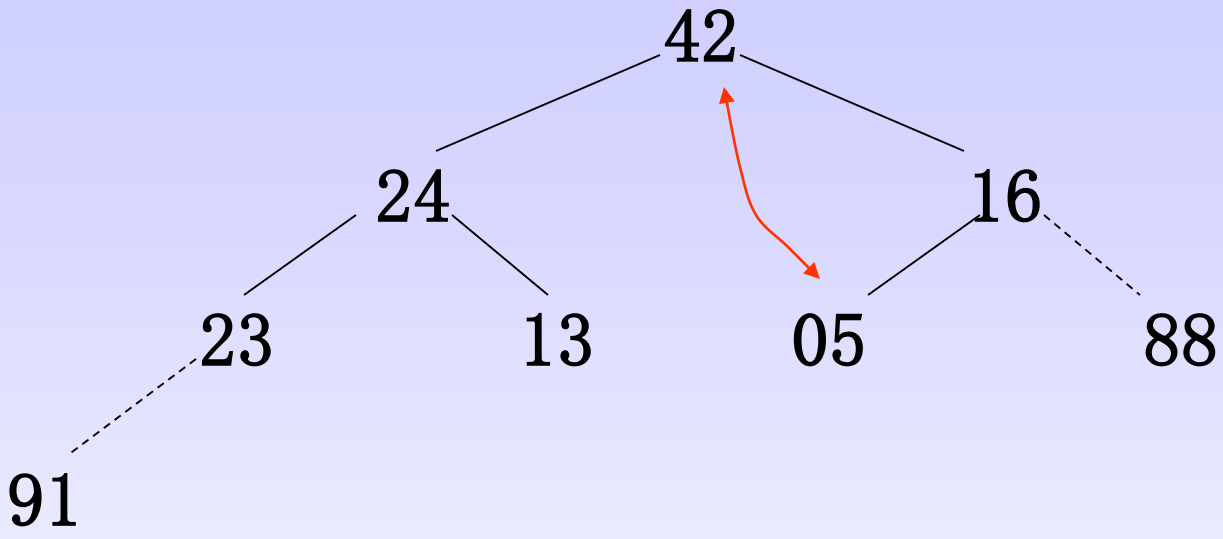
10.4 选择排序



05	24	42	23	13	16	88	91

（4）第二趟排序之后

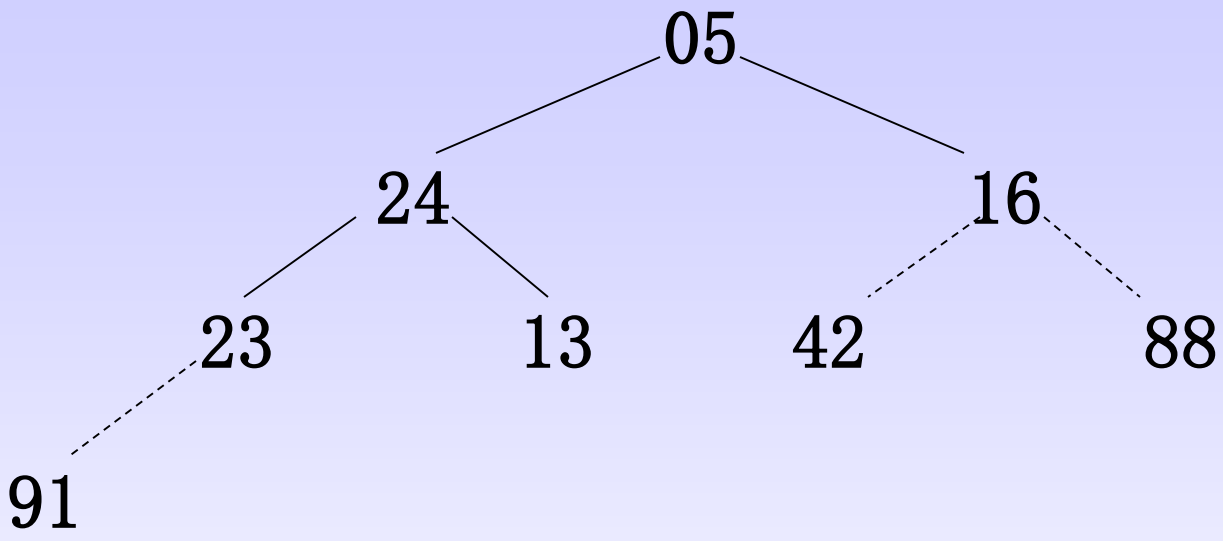
10.4 选择排序



42	24	16	23	13	05	88	91

（5）重建的堆R[1]到R[6]

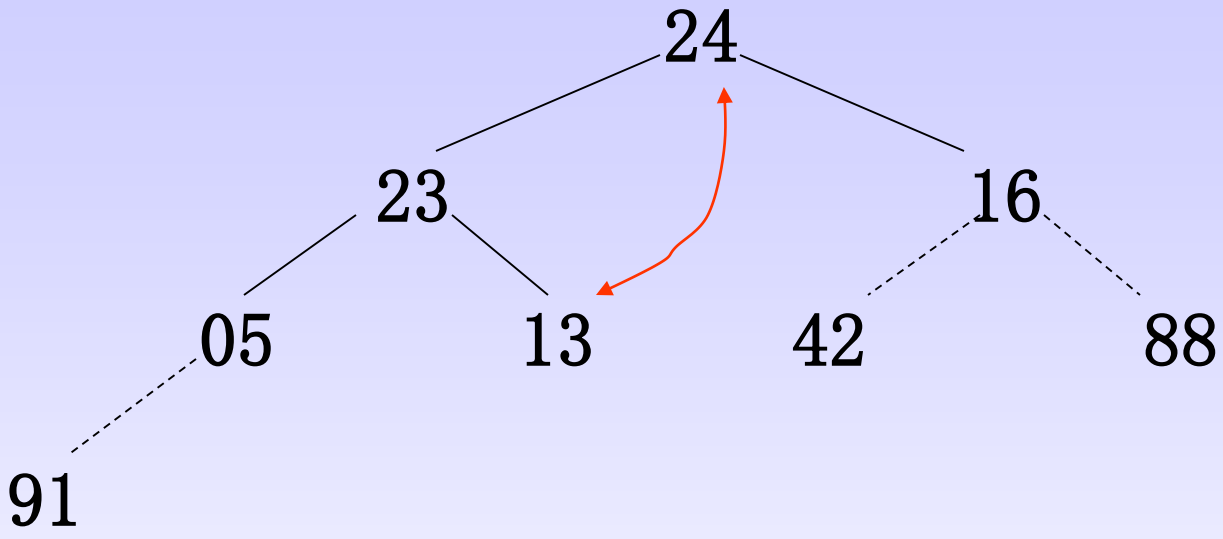
10.4 选择排序



05	24	16	23	13	42	88	91

（6）第三趟排序之后

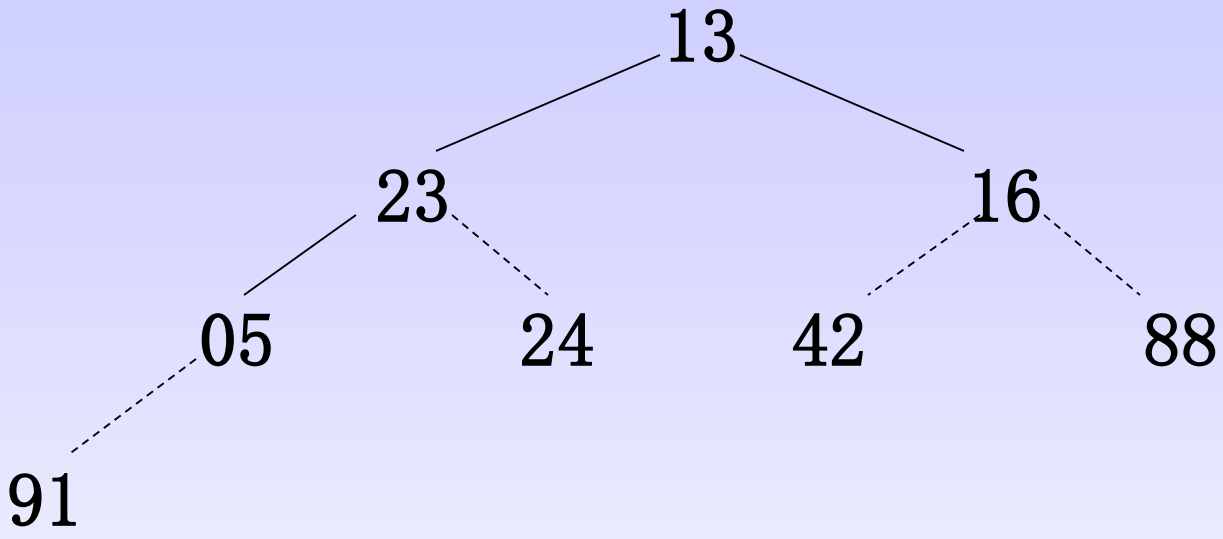
10.4 选择排序



24	23	16	05	13	42	88	91

（7）重建的堆R[1]到R[5]

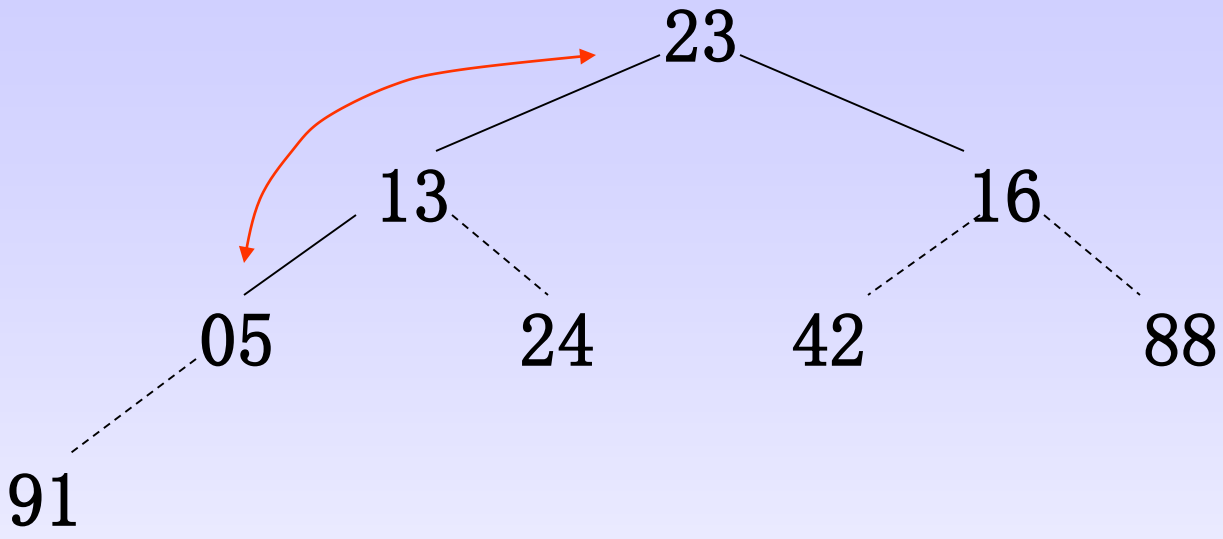
10.4 选择排序



13	23	16	05	24	42	88	91

（8）第四趟排序之后

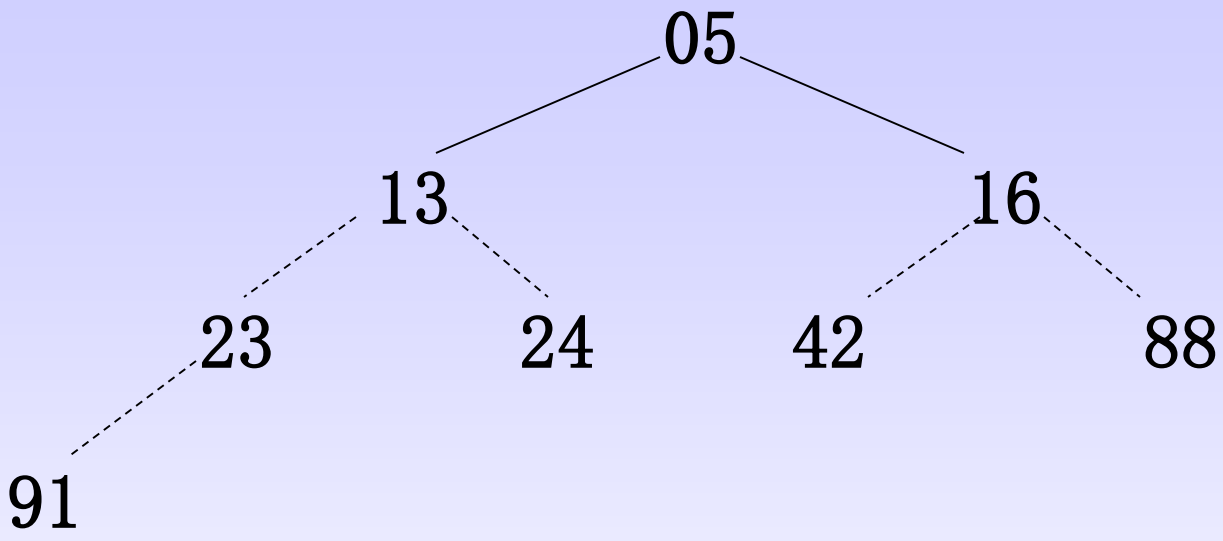
10.4 选择排序



23	13	16	05	24	42	88	91

（9）重建的堆R[1]到R[4]

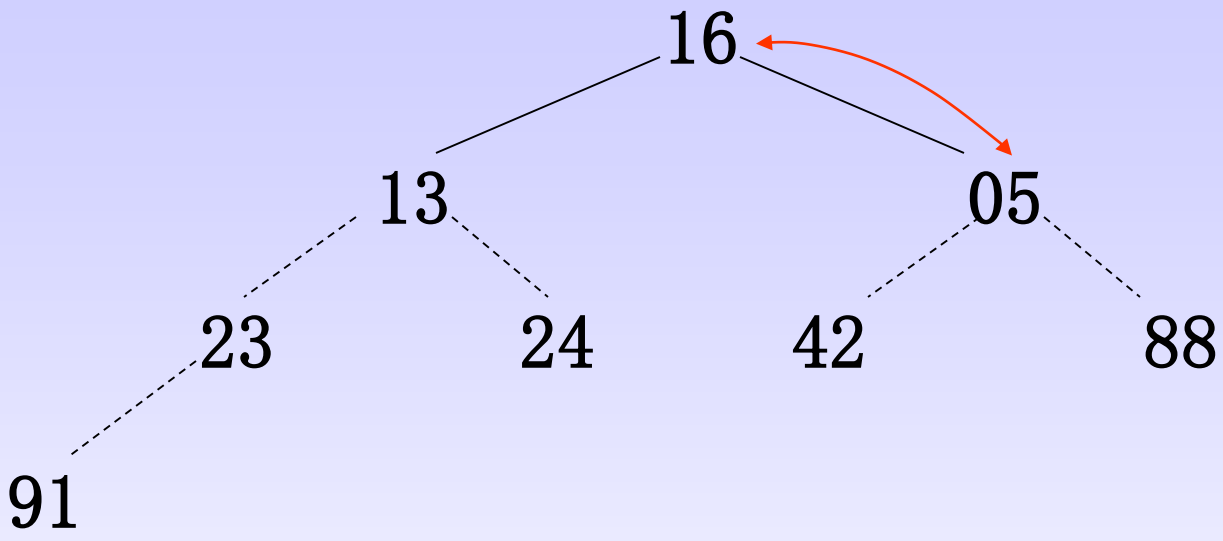
10.4 选择排序



05	13	16	23	24	42	88	91

（10）第五趟排序之后

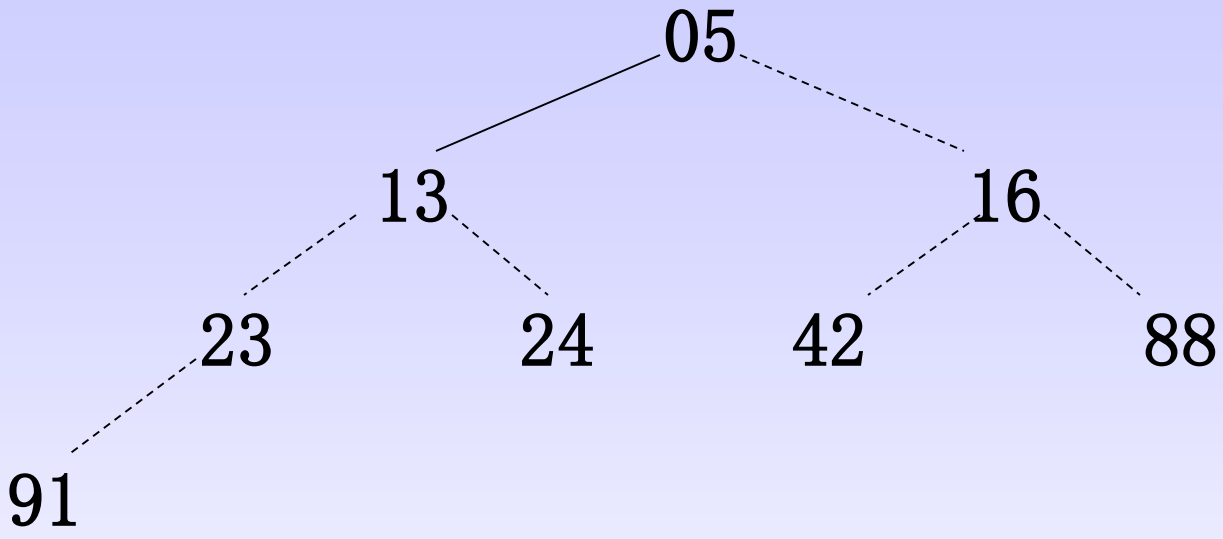
10.4 选择排序



16	13	05	23	24	42	88	91

（11）重建的堆R[1]到R[3]

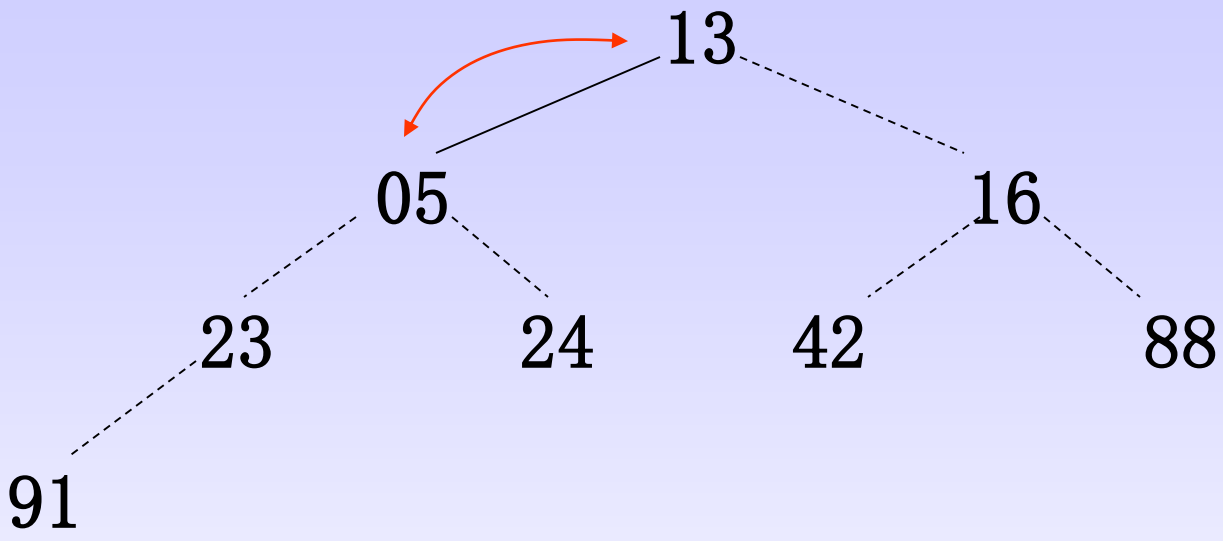
10.4 选择排序



05	13	16	23	24	42	88	91

（12）第六趟排序之后

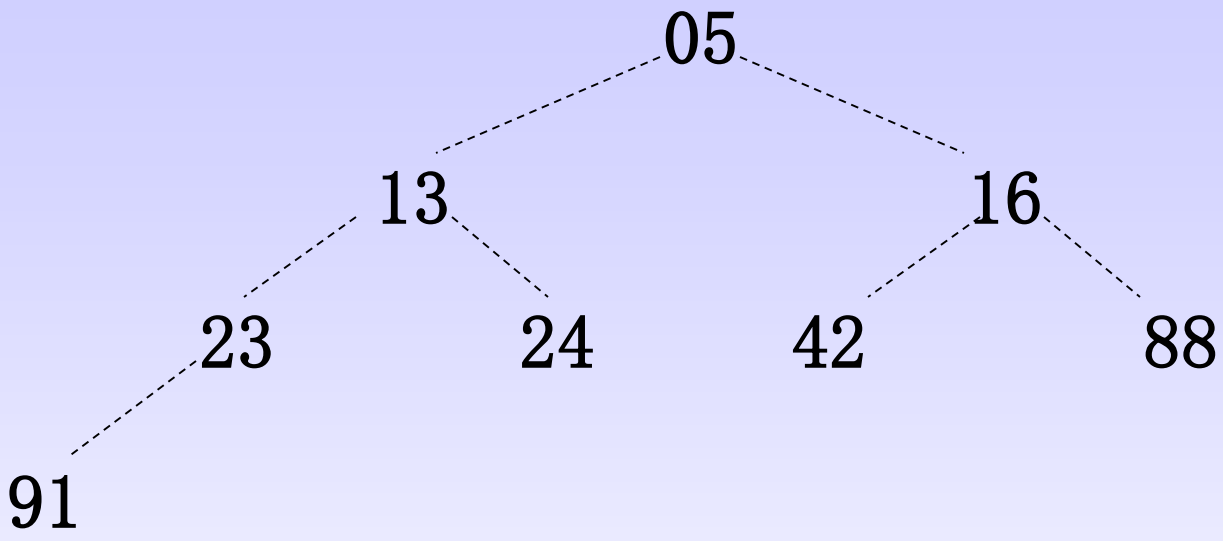
10.4 选择排序



13	05	16	23	24	42	88	91

（13）重建的堆R[1]到R[2]

10.4 选择排序



05	13	16	23	24	42	88	91

（14）第七趟排序之后

10.4 选择排序

堆排序的算法

```
void HeapSort( SqList &H )
{   for( i=H.length/2; i>0; i--)
        HeapAdjust( H, i, H.length );  //把H.r[1..H.length]建成最大堆
    for( i=H.length; i>1; i--)
    {   H.r[1]<->H.r[i];              //堆顶记录交换到H.r[1..i]的最后
        HeapAdjust( H, 1, i-1 );   //把H.r[1..i-1]建成最大堆
    }
}
```

10.4 选择排序

算法分析

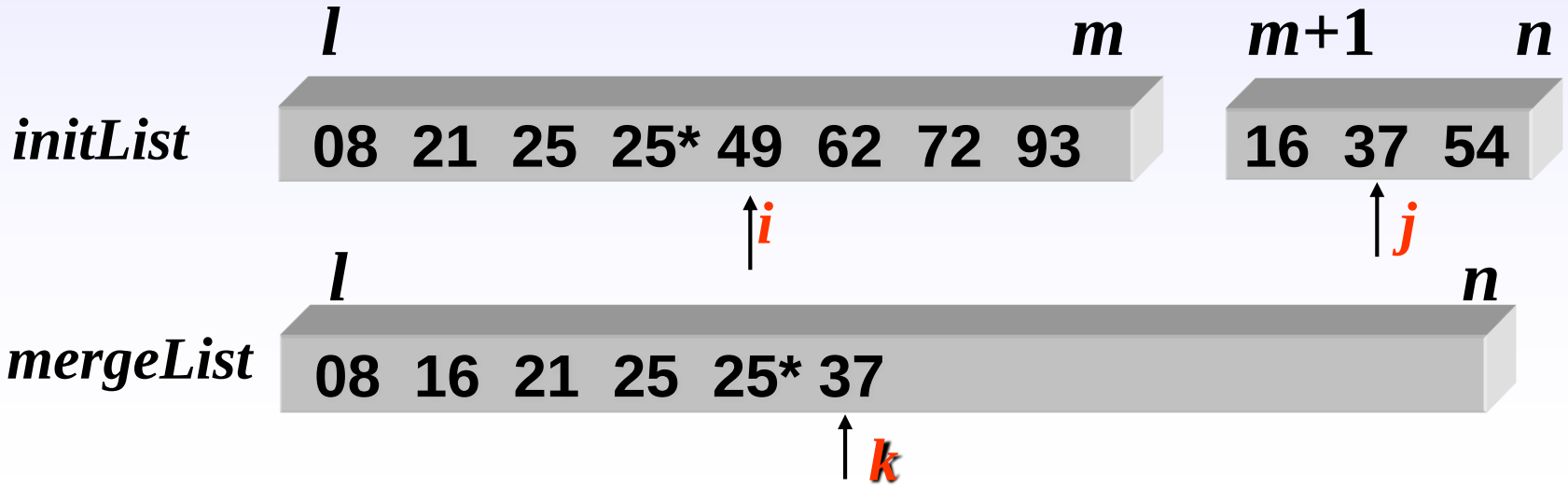
- 堆排序的时间复杂性为$O(n\log_2 n)$。
- 该算法的附加存储主要是在第二个for循环中用来执行元素交换时所用的一个临时元素。因此，该算法的空间复杂性为O(1)。
- 堆排序是一个不稳定的排序方法。

10.5 归并排序

- 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。
- 元素序列 *initList* 中有两个有序表***V*[*l*] …*V*[*m*]**和***V*[*m*+1] …*V*[*n*]**。它们可归并成一个有序表，存于另一元素序列 *mergedList* 的***V*[*l*] …*V*[*n*]**中。这种归并方法称为两路归并 **(2-way merging)**。

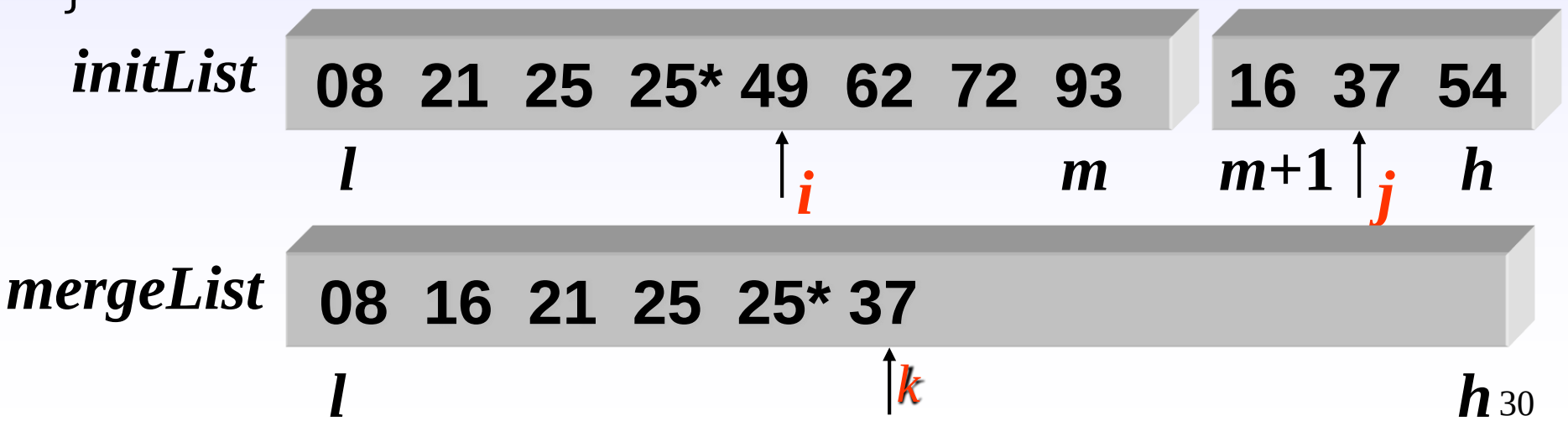
10.5 归并排序

- 归并的基本思想是：设有序表A和B分别表示$V[l]$ …$V[m]$和$V[m+1]$ …$V[n]$，变量 i 和 j 分别是表A和表B的当前检测指针。设表C是归并后的新有序表，变量 k 是它的当前存放指针。
  - 当 i 和 j 都在两个表的表长内变化时，根据A[i]与B[j]的关键字的大小，依次把关键字小的元素排放到新表C[k]中；
  - 当 i 与 j 中有一个已经超出表长时，将另一 个表中的剩余部分照抄到新表C[k]中



10.5 归并排序

```
void Merge(ElemType initList[], ElemType mergedList[], int l, int m, int h)
{ i = l;  j = m+1;  k = l;
  while ( i <= m && j <= h )                          //两两比较
     if(initList[i].key<=initList[j].key )  mergedList[k++] = initList[i++];
     else   mergedList[k++] = initList[j++];
  while( i<= m )  mergedList[k++] = initList[i++];
  while( j<= h ) mergedList[k++] = initList[j++];
}
```

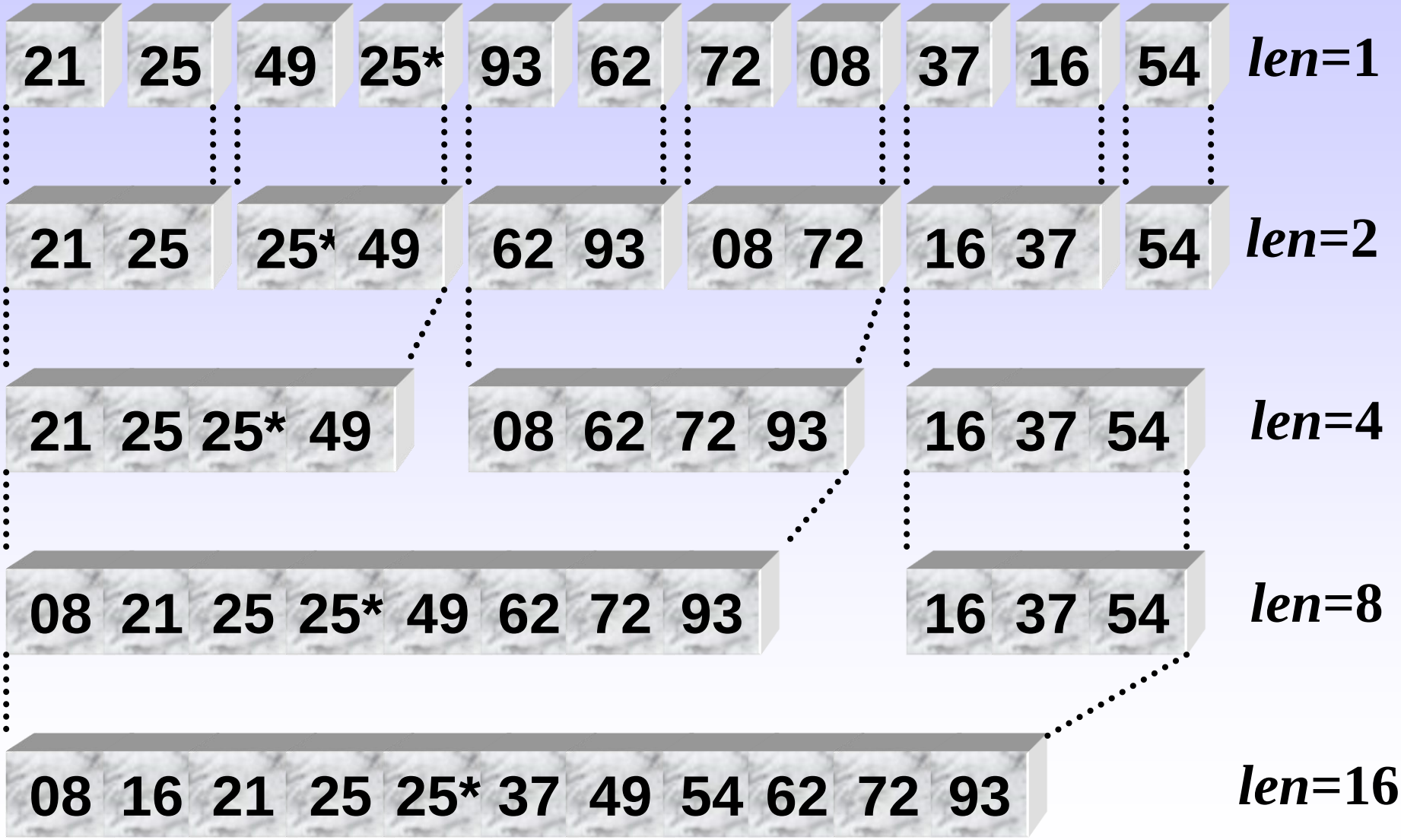


10.5 归并排序

迭代的归并排序算法

- 迭代的归并排序算法就是利用两路归并过程进行排序的。其基本思想是：设初始元素序列有 n 个元素，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。

10.5 归并排序


21 25 49 25* 93 62 72 08 37 16 54
len=1
21 25 25* 49 62 93 08 72 16 37 54
len=2
21 25 25* 49 08 62 72 93 16 37 54
len=4
08 21 25 25* 49 62 72 93 16 37 54
len=8
08 16 21 25 25* 37 49 54 62 72 93
len=16

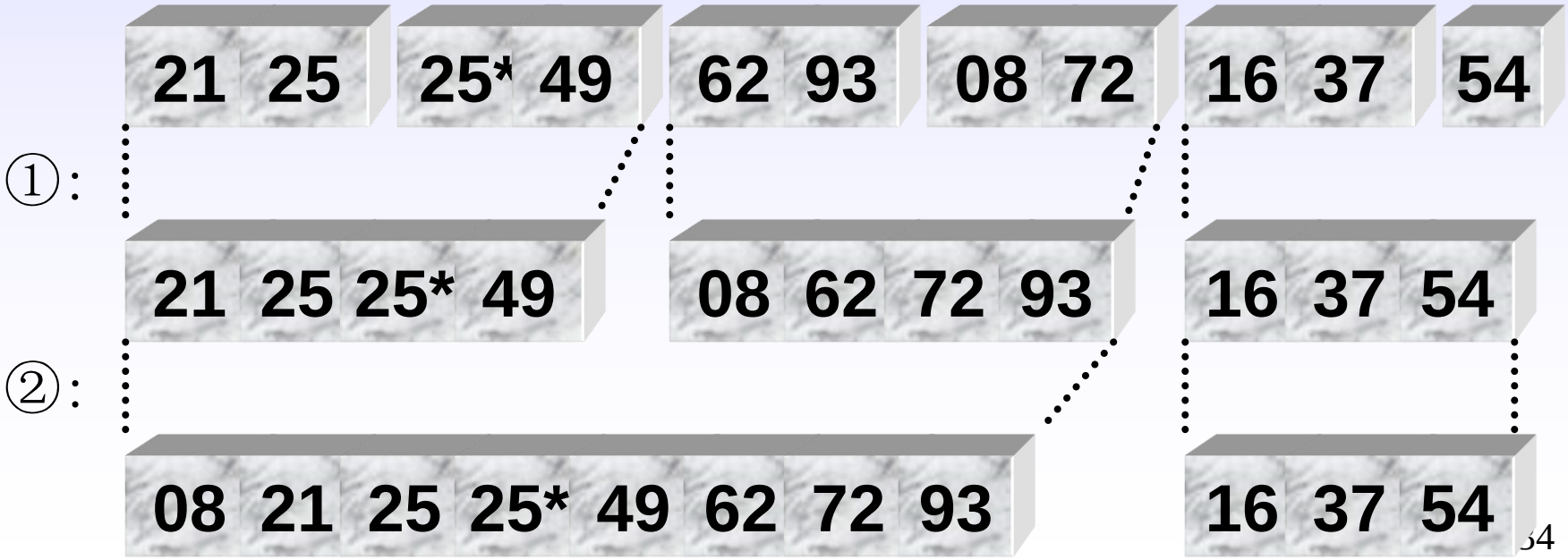
10.5 归并排序

一趟归并排序的情形

- 设*initList.r*[1..*n*]中的 *n*个元素已经分为一些长度为*len*的归并项，将其两两归并成一些长度为 2*len*的归并项，放到*mergedList.r* [1..*n*]中。
- 如果*n*不是2*len*的整数倍，则一趟归并到最后，可能遇到两种情形：
  - 剩下一个长度为*len*的归并项和另一个长度不足*len*的归并项，可用一次*merge*算法，将它们归并成一个长度小于2*len*的归并项。
  - 只剩下一个归并项，其长度小于或等于*len*，可将它直接抄到数组*mergedList.r* [1..*n*] 中。

10.5 归并排序

```
void MSort(ElemType initList[], ElemType mergedList[], int len, int n )
{  for( i=1; i <= n-2*len+1; i+=2*len )                    //两两归并
       Merge(initList, mergedList, i, i+len-1, i+2*len-1);
   if( i<n-len+1 )  Merge(initList, mergedList, i, i+len-1, n);  //①
   else while( i<= n ) mergedList[i] = initList[i++];            //②
}
```



10.5 归并排序

(两路)归并排序的主算法

```
void MergeSort ( SqList &L )
{   //按元素关键字非递减的顺序对表L中元素排序
    ElemType *L1;          //辅助数组
    L1=(ElemType *)malloc((L.length+1)*sizeof(ElemType));
    len = 1;
    while ( len < L.length)
    {       MSort ( L.r, L1, len, L.length );
            len *= 2;
            MSort ( L1, L.r, len, L.length );
            len *= 2;
    }
    free(L1);
}
```

10.5 归并排序

算法分析

- 在迭代的归并排序算法中，函数*MergeSort( )*调用*MSort( )*正好$\lceil \log_2 n \rceil = O(\log_2 n)$次，函数*MSort( )*做一趟两路归并排序，要调用*Merge( )*函数$\lceil n/(2*len) \rceil = O(n/len)$次，而每次*Merge( )*要执行比较$O(len)$次，所以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元素数组同样大小的辅助数组。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

10.6.1 多关键字排序

- 以扑克牌排序为例。每张扑克牌有两个“关键字”：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
- 如果把所有扑克牌排成以下次序：♣2, …, ♣A, ♦2, …, ♦A, ♥2, …, ♥A, ♠2, …, ♠A，这就是多关键字排序。
- 对于上例两关键字的排序，可以先按花色排序，之后再按面值排序；也可以先按面值排序，再按花色排序。

10.6 基数排序

- 假定有一个 n 个元素的序列 $\{V_0, V_1, \ldots, V_{n-1}\}$，且每个元素$V_i$ 中含有 d 个关键字：$\left(K_i^1, K_i^2, \ldots, K_i^d\right)$
- 如果对于序列中任意两个元素 V_i和 V_j ($0 \le i < j \le n-1$) 都满足：$\left(K_i^1, K_i^2, \ldots, K_i^d\right) < \left(K_j^1, K_j^2, \ldots, K_j^d\right)$
- 则称序列对关键字 $(K^1, K^2, \ldots, K^d)$ 有序。其中，K^1 称为最高位关键字，K^d 称为最低位关键字。
- 基数排序是借助多关键字排序思想对单关键字进行排序的方法。

10.6 基数排序

(1) 最高位优先法MSD (Most Significant Digit first)通常是一个递归的过程：

- 先根据最高位关键字K^1排序,得到若干元素组，元素组中每个元素都有相同关键字K^1。
- 再分别对每组中元素根据关键字K^2进行排序，按K^2值的不同，再分成若干个更小的子组，每个子组中的元素具有相同的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，把所有子组中的元素依次连接起来，就得到一个有序的元素序列。

10.6 基数排序

(2) 最低位优先法LSD (Least Significant Digit first):

- 首先依据最低位关键字K^d对所有元素进行一趟排序
- 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，而是整个元素组都参加排序。

10.6 基数排序

例如:学生记录含三个关键字:系别、班号和班内的序列号，其中以系别为最主位关键字。

LSD的排序过程如下:

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

10.6 基数排序

10.6.2 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，单关键字K_i看成是一个*d*元组：$\left(K_i^1, K_i^2, \ldots, K_i^d\right)$
- 其中的每一个分量 K_i^j ($1 \le j \le d$) 看成是一个关键字。
- 分量 K_i^j ($1 \le j \le d$) 有*radix*种取值，则称*radix*为基数。例如，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。

10.6 基数排序

- 针对*d*元组中的每一位分量，把元素序列中的所有元素，按 K_i^j 的取值，先“分配”到*radix*个队列中去。然后再按各队列的顺序，依次把元素从队列中“收集”起来，这样所有元素按取值 K_i^j 排序完成。
- 如果对于所有元素的关键字$K_0, K_1, \ldots, K_{n-1}$，依次对各位的分量，让 $j = d, d-1, \ldots, 1$，分别用这种“分配”、“收集”的运算逐趟进行排序，在最后一趟“分配”、“收集”完成后，所有元素就按其关键字的值从小到大排好序了。

10.6 基数排序

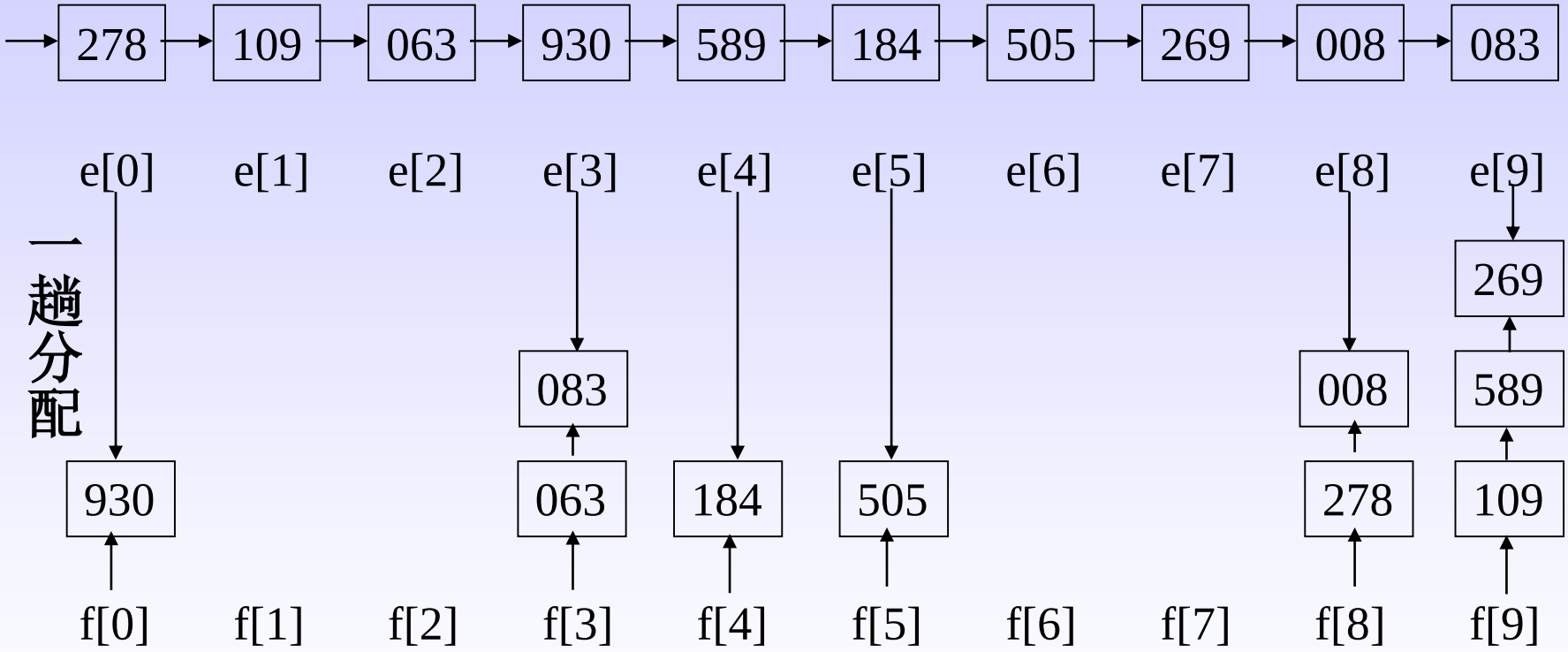
例如：对如下的一组关键字

{278,109,063,930,589,184,505,269,008,083 }

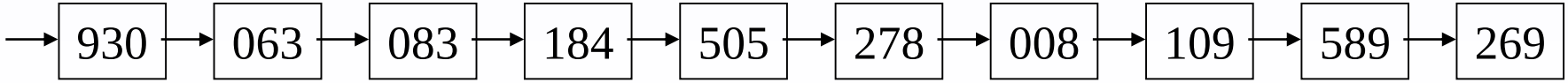
- 首先按其“个位数” 取值分别为 0, 1, …, 9 “分配”成 10 组，之后按从 0 至 9 的顺序将 它们 “收集” 在一起;
- 然后按其 “十位数” 取值分别为 0, 1, …, 9 “分配”成 10 组，之后再按从 0 至 9 的顺序将它们 “收集”在一起;
- 最后按其“百位数”重复一遍上述操作。

10.6 基数排序

例 初始状态：

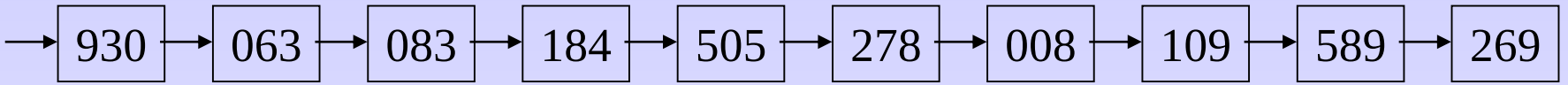


一趟收集：

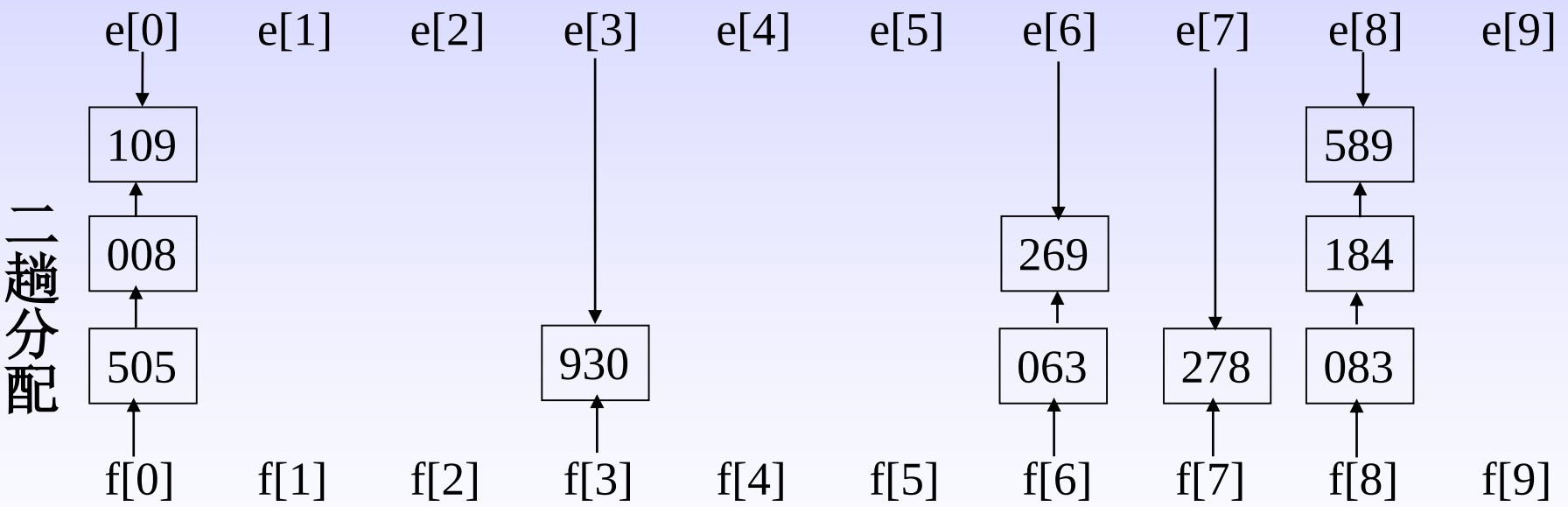


10.6 基数排序

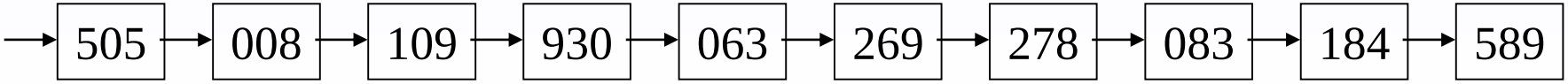
一趟收集：



二趟分配

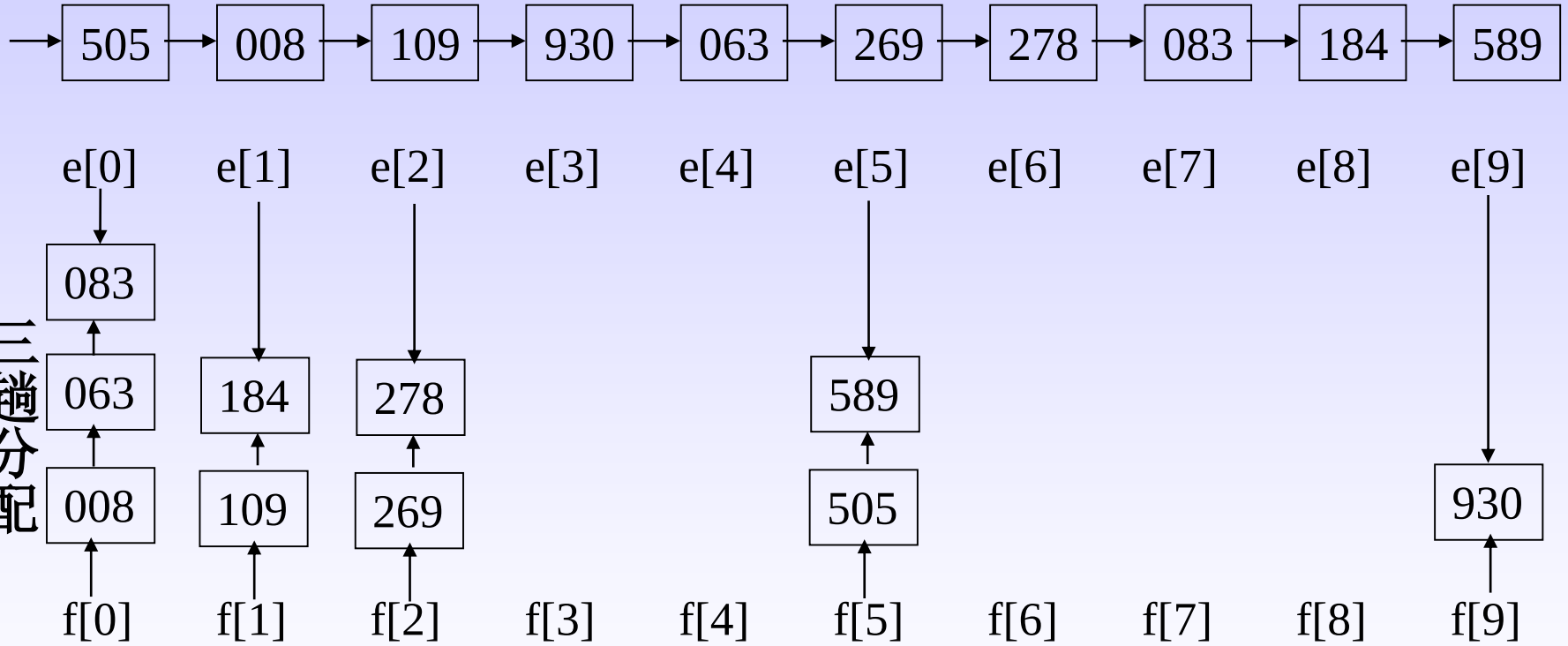


二趟收集：

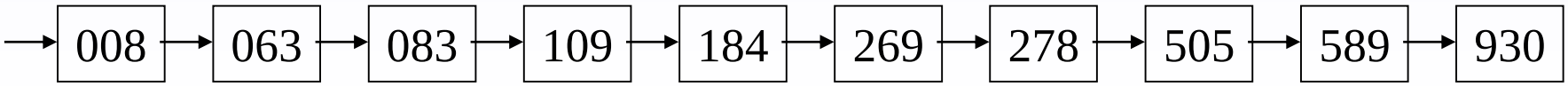


10.6 基数排序

二趟收集：



三趟收集：



10.6 基数排序

在计算机上实现基数排序时，为减少所需辅助存储空间，应采用链表作存储结构，即链式基数排序，具体作法为：

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”所取值，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 2）和 3）两步。

10.6 基数排序

数据类型声明

```
typedef struct{
    KeyType   keys[];
    InfoType  otheritems;        // 其它数据域
    int       next;              // 链域
} SLCell;
typedef struct{
    SLCell    r[MAX];            // r[0]为头结点
    int       length;            // 表长
} SLList;
```

10.6 基数排序

基数排序主算法

```
void RadixSort( SLList &L)
  {  int f[KEYNUM], e[KEYNUM];
     for( i=0;  i< L.length;  i++)
           L.r[i].next=i+1;  // 建立静态链表
     L.r[L.length].next=0;          // 链表的表尾
     for( i=0; i<keynum; i++)    // keynum为关键字位数
     {   Distribute(L.r, i, f, e);
         Collect (L.r, i, f, e);
      }
  }
```

10.6 基数排序

基数排序分配算法

```
void Distribute(SLCell r[], int i, int f[], int e[])
{  for(j=0; j<Radix; j++)
        f[j]=0;          // 清空所有队列, 依次处理链表中的每个元素
   for(p=r[0].next; p; p=r[p].next)
   {  j = ord(r[p].keys[i]);
      if( f[j]==0 ) f[j]=p;                    // 将r[p]加入到第j个队列
      else r[e[j]].next=p;
      e[j]=p;
   }
}
```

10.6 基数排序

基数排序收集算法

```
void Collect(SLCell r[], int i, int f[], int e[])
{  for(j=0; f[j]==0; j++);  // 找第1个非空队列
    r[0].next=f[j];
    t=e[j];                          // t保存前1个队列的队尾位置
    for(k=j+1; k<Radix; k++)        // 将所有队列首尾连接
      if( f[k]!=0 )
     {  r[t].next=f[k];
        t=e[k];
      }
    r[t].next=0;                     // 链表的表尾
}
```

10.6 基数排序

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 个元素进行“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，对于元素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。

各种排序方法的比较

排序方法	元素比较		元素移动		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	×	$\log_2 n$	
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	