第三章　排序问题和离散集合的操作

3.1　合并排序

3.1.1　合并排序算法的实现

假定有 8 个元素，第一步，划分为四对，每一对两个元素，用 merge 算法合并成四个有序的序列；第二步，把四个序列划分成两对，用 merge 算法合并成两个有序的序列；最后，再利用 merge 算法合并成一个有序的序列。

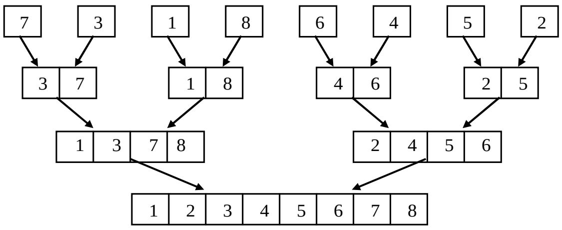


图 3.1　合并 8 个元素的过程

算法 3.1　合并排序算法

输入：具有 n 个元素的数组 A[]

输出：按递增顺序排序的数组 A[]

```
template <class Type>
void merge_sort(Type A[],int n)
{
   int  i,s,t = 1;
   while (t<n) {
       s = t; t = 2 * s;  i = 0;
       while (i+t<n) {
           merge(A,i,i+s-1,i+t-1,t);
           i = i + t;
       }
       if (i+s<n)
           merge(A,i,i+s-1,n-1,n-i);
   }
}
```

i：开始合并时第一个序列的起始位置；

s：合并前序列的大小；

t：合并后序列的大小；

i、$i+s-1$、$i+t-1$ 定义被合并的两个序列的边界。

例如，当 $n=11$ 时，算法的工作过程如图 3.2 所示，过程如下：

1. 在第一轮循环，$s=1$、$t=2$，有 5 对 1 个元素的序列进行合并，当 $i=10$ 时，$i+t=12>n$，退出内部的 while 循环。但 $i+s=11$，不小于 n，所以，不执行第 12 行的合并工作，余留一个元素没有处理。
2. 在第二轮，$s=2, t=4$，有两对两个元素的序列进行合并，在 $i=8$ 时，$i+t=12>n$，退出内部的 while 循环。但 $i+s=10<n$，所以执行第 12 行的合并工作，把一个大小为 2 的序列和另外一个元素合并，产生一个 3 个元素的有序序列。
3. 在第三轮，$s=4, t=8$，有一对四个元素的序列合并，在 $i=8$ 时，$i+t=16>n$，退出内部的 while 循环。而 $i+s=12>n$，所以，不执行第 12 行的合并工作，余留一个序列没有处理。
4. 在第四轮，$s=8, t=16$。在 $i=0$ 时，$i+t=16>n$，所以不执行内部的 while 循环，但 $i+s=8<n$，所以执行第 12 行的合并工作，产生一个大小为 11 的有序序列。
5. 在进入第五轮时，因为 $t=16>n$，所以退出外部的 while 循环，结束算法。

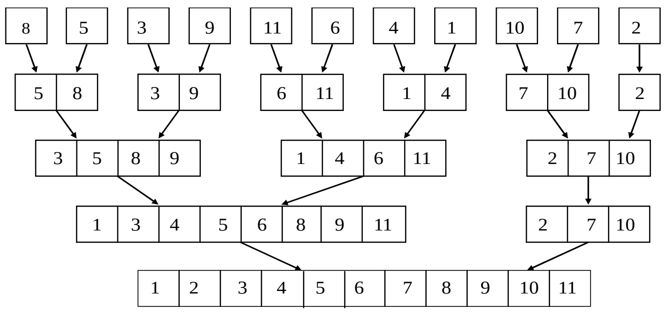


图 3.2　$n=11$ 时的合并排序的工作过程

3.1.2　合并排序算法的分析

一、时间复杂性

假定 n 是 2 的幂。

外部 while 循环的循环体的执行次数：$k=\log n$ 次。

	内部 while 循环 merge 执行次数	merge 执行的比较次数	所产生序列数	序列长度	元素比较总次数 最少	元素比较总次数 最多
第 1 轮	$n/2$	1	$n/2$	2	$(n/2)*1$	$(n/2)*1$
第 2 轮	$n/4=n/2^2$	2，4 – 1 = 3	$n/2^2$	4	$(n/2^2)*2^1$	$(n/2^2)*(2^2-1)$
第 3 轮	$n/2^3$	4，8 – 1 = 7	$n/2^3$	8	$(n/2^3)*2^2$	$(n/2^3)*(2^3-1)$
第 j 轮	$n/2^j$	2^{j-1}，2^j-1	$n/2^j$	2^j	$(n/2^j)*2^{j-1}$	$(n/2^j)*(2^j-1)$

合并排序算法的执行时间，至少为：

$$\sum_{j=1}^{k}\frac{n}{2^j}\cdot 2^{j-1}=\sum_{j=1}^{k}\frac{n}{2}$$

$$=\frac{1}{2}kn$$

$$=\frac{1}{2}n\log n$$

至多为：

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

$$=kn-n\sum_{j=1}^{k}\frac{1}{2^j}$$

$$=kn-n\left(1-\frac{1}{2^k}\right)$$

$$=kn-n\left(1-\frac{1}{n}\right)$$

$$=n\log n-n+1$$

合并排序算法的运行时间，是$\Omega(n\log n)$，也是$O(n\log n)$，因此，是$\Theta(n\log n)$。

二、空间复杂性

每调用一次 merge 算法，便分配一个适当大小的缓冲区，退出 merge 算法便释放它。在最后一次调用 merge 算法时，所分配的缓冲区最大，此时，它把两个序列合并成一个长度为n的序列，需要$\Theta(n)$个工作单元。所以，合并排序算法所使用的工作空间为$\Theta(n)$。

3.2 基于堆的排序

3.2.1 堆

一、堆的定义

定义 3.2 n个元素称为堆，当且仅当它的关键字序列$k_1,k_2,\cdots,k_n$满足：

$$k_i \le k_{2i} \quad k_i \le k_{2i+1} \qquad 1\le i\le \lfloor n/2 \rfloor \qquad (3.2.1)$$

或者满足：

$$k_i \ge k_{2i} \quad k_i \ge k_{2i+1} \qquad 1\le i\le \lfloor n/2 \rfloor \qquad (3.2.2)$$

把满足 (3.2.1) 式的堆称为最小堆 (min_heaps)； 把满足 (3.2.2) 式的堆称为最大堆

（max_heaps）。

二、堆的性质：可看成是一棵完全二叉树。如果树的高度为 d

1．所有的叶结点不是处于第 d 层，就是处于第 $d-1$ 层；

2．当 $d \geq 1$ 时，第 $d-1$ 层上有 2^{d-1} 个结点；

3．第 $d-1$ 层上如果有分支结点，则这些分支结点都集中在树的最左边；

4．每个结点所存放元素的关键字，都大于（最大堆）或小于（最小堆）它子孙结点所存放元素的关键字。

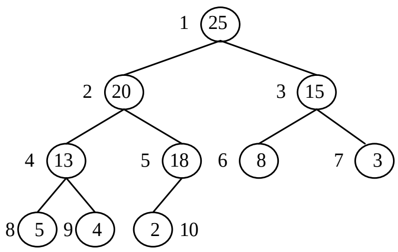
三、用数组 H 存放具有 n 个元素的堆

1．根结点存放在 $H[1]$；

2．假定结点 x 存放在 $H[i]$，如果它有左儿子结点，则它的左儿子结点存放在 $H[2i]$；如果它有右儿子结点，则它的右儿子结点存放在 $H[2i+1]$；

3．非根结点 $H[i]$ 的父亲结点存放在 $H[\lfloor i/2 \rfloor]$。

例：



1	25
2	20
3	15
4	13
5	18
6	8
7	3
8	5
9	4
10	2

图 3.3　堆及其数组表示

3.2.2　堆的操作

一般来说，对于堆这样的数据结构，需要下面几种操作：

- void sift_up（Type H[ ]，int i）；　　把堆中的第 i 个元素上移
- void sift_down（Type H[ ]，int n，int i）；　　把堆中的第 i 个元素下移
- void insert（Type H[ ]，int &n，Type x）；　　把元素 x 插入堆中
- void delete（Type H[ ]，int &n，int i）；　　删去堆中第 i 个元素
- Type delete_max（Type H[ ]，int &n）；　　从非空的最大堆中删除并回送关键字最大的元素
- void make_head（Type H[ ]，int n）；　　使数组 H 中的元素按堆的结构重新组织

3.2.2.1　元素上移操作

沿 $H[i]$ 到根的路线，把 $H[i]$ 向上移动。移动过程中，如果大于它的父亲结点，就与父亲结点交换位置。否则，操作结束。

算法 3.2 元素上移操作

输入：数组 H[]及被上移的元素下标 i

输出：维持堆的性质的数组 H[]

```
template <class Type>
void sift_up(Type H[],int i)
{
   BOOL done = FALSE;
   if (i!=1) {
      while (!done && i!=1) {
         if (H[i] > H[i/2])
            swap(H[i],H[i/2]);
         else  done = TRUE;
         i = i / 2;
      }
   }
}
```

执行时间：共$\lfloor \log n \rfloor$层，每层一个元素比较操作，$O(\log n)$

工作单元：$\Theta(1)$

例 3.1 如果在图 3.3 中，把结点 9 的内容修改为 28 的工作过程。

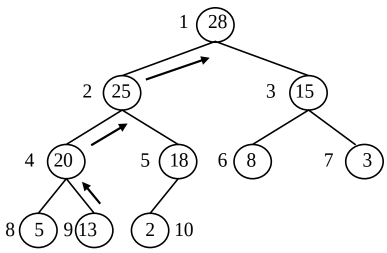


图 3.4 sift_up 操作的工作过程

3.2.2.2 元素下移操作

在向下移动的过程中，把它的关键字和它两个儿子中关键字大的儿子比较，如果小于它儿子结点的关键字，就与儿子结点交换位置。否则，操作结束。

算法 3.3 元素下移操作

输入：数组 H[],数组的元素个数 n,被下移的元素下标 i

输出：维持堆的性质的数组 H[]

```
template <class Type>
void sift_down(Type H[],int n,int i)
{
```

```
    BOOL done = FALSE;
    if ((2*i)<=n) {
       while (!done && (i=2*i<=n)) {
          if (i+1<=n && H[i+1]>H[i])
             i = i + 1;
          if (H[i/2] < H[i])
             swap(H[i/2],H[i]);
          else done = TRUE;
       }
    }
}
```

执行时间：共$\lfloor \log n \rfloor$层，每层两个元素比较操作$O(\log n)$

工作单元：$\Theta(1)$

例 3.2　如果在图 3.3 中，把结点 2 的内容由 20 改为 1 的工作过程。

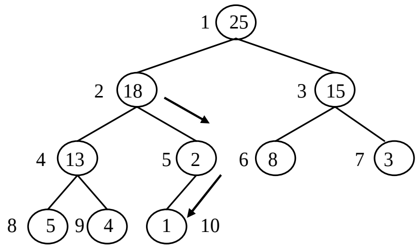


图 3.5　sift_down 操作的工作过程

3.2.2.3　元素插入操作

堆的大小增 1，把 x 放到堆的未端，对 x 做上移操作。借助于 sift_up 操作，既把元素插入堆中，又维持了堆的性质。

算法 3.4　元素插入操作

输入：数组 H[],数组的元素个数 n,被插入的元素 x

输出：维持堆的性质的数组 H[],及插入后的元素个数 n

```
template <class Type>
void insert(Type H[],int &n,Type x)
{
    n = n + 1;
    H[n] = x;
    sift_up(H,n);
}
```

执行时间： sift_up 操作的执行时间，$O(\log n)$

工作单元： $\Theta(1)$。

3.2.2.4 元素删除操作

为删除堆中的元素 $H[i]$，用堆中最后一个元素取代 $H[i]$，堆的大小减一。再根据被删除元素和取代它的元素的大小，确定对取代它的元素是做上移操作、还是做下移操作，

算法 3.5 元素删除操作

输入：数组 H[],数组的元素个数 n,被删除元素的下标 i

输出：维持堆的性质的数组 H[],及删除后的元素个数 n

```
template <class Type>
void delete(Type H[],int &n,int i)
{
   Type  x,y;
   x = H[i];   y = H[n];
   n = n - 1;
   if (i<=n) {
      H[i] = y;
      if (y>=x)
          sift_up(H,i);
      else
          sift_down(H,n,i);
   }
}
```

执行时间： sift_up 操作、或 sift_down 操作的执行时间， $O(\log n)$

工作单元： $\Theta(1)$

3.2.2.5 删除关键字最大的元素

在最大堆中，关键字最大的元素位于根结点，借助 delete 操作，既做删除操作，又维持堆的性质。

算法 3.6 删除关键字最大元素

输入：数组 H[],数组的元素个数 n

输出：维持堆的性质的数组 H[],被删除的元素、及删除后的元素个数 n

```
template <class Type>
Type delete_max(Type H[],int &n)
{
   Type  x;
   x = H[1];
```

```
    delete(H[],n,1);
    return x;
}
```

执行时间：$O(\log n)$

工作单元： $\Theta(1)$

3.2.3 堆的建立

一、建造堆的两种方法

1、用 insert 操作建造堆

算法 3.7 建造堆的第一种算法

输入：数组 A[],数组的元素个数 n

输出：n 个元素的堆 H[]

```
template <class Type>
void make_heap1(Type A[],Type H[],int n)
{
    int i,m = 0;
    for (i=0;i<n;i++)
        insert(H,m,A[i]);
}
```

执行时间：插入第 i 个元素需花费 $O(\log i)$，插入 n 个元素，需花费 $O(n\log n)$ 时间

工作单元：$\Theta(n)$

2、把数组本身构造成一个堆。

调整过程：从最后一片树叶找到它上面的分支结点，从这个分支结点开始作下移操作，一直到根结点为止。

算法 3.8 建造堆的第二种算法

输入：数组 H[],数组的元素个数 n

输出：n 个元素的堆 A

```
template <class Type>
void make_heap(Type A[],int n)
{
    int i;
    A[n] = A[0];
    for (i=n/2;i>=1;i--)
        sift_down(A,i);
```

```
8. }
```

例 3.3 图 3.6 表示把一个具有 11 个元素的数组，调整成一个堆的过程。

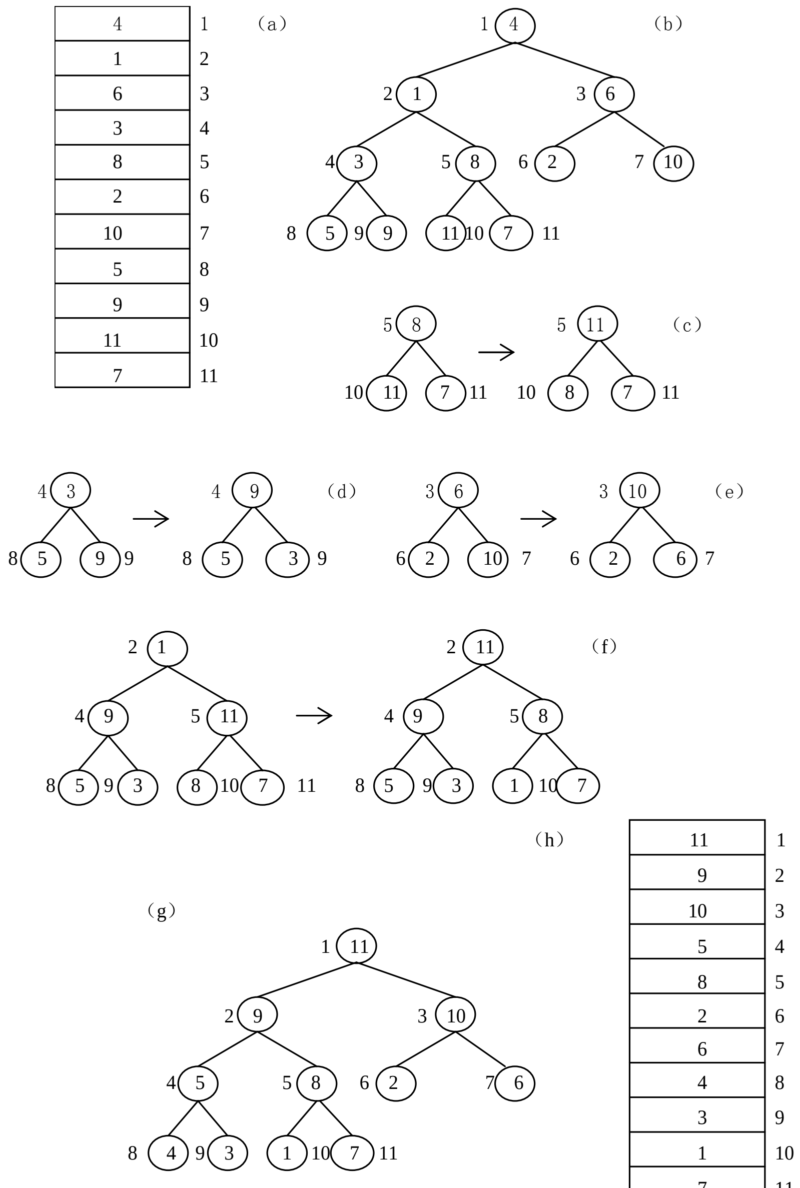


图 3.6 堆的建立过程

二、算法 make_heap 的运行时间分析

1. 数组有 n 个元素，所构成的二叉树的高度为 $k = \lfloor \log n \rfloor$；
2. 第 i 层的元素 $A[j]$ 最多下移 $k-i$ 层，最多执行 $2(k-i)$ 次元素比较；
3. 第 i 层上共有 2^i 个结点，第 i 层上所有结点最多执行 $2(k-i)2^i$ 次元素比较；
4. 第 k 层上的元素，都是叶子结点，无需执行下移操作。最多只需对第 0 层到第 $k-1$ 层

的元素执行下移操作。

由此，算法 make_heap 所执行的元素比较次数为：

$$\sum_{i=0}^{k-1} 2(k-i)2^i = 2k\sum_{i=0}^{k-1} 2^i - 2\sum_{i=0}^{k-1} i2^i$$

如果令 $n = 2^k$ ，即 $k = \log n$ 。由公式（2.1.20）及（2.1.23），有：

$$\begin{aligned}
\sum_{i=0}^{k-1} 2(k-i)2^i &= 2k(2^k - 1) - 2((k-1)2^{k+1} - (k-1)2^k - 2^k + 2) \\
&= 2(k2^k - k) - 2(k2^k - 2^{k+1} + 2) \\
&= 4 \cdot 2^k - 2k - 4 \\
&= 4n - 2\log n - 4 \\
&< 4n
\end{aligned}$$

所以执行时间为 $O(n)$ 。

共 $\lfloor n/2 \rfloor$ 个结点作下移操作，至少需要 $2\lfloor n/2 \rfloor$ 次元素比较。所以执行时间是 $\Omega(n)$ 。

make_heap 的执行时间是 $\Theta(n)$ 。

工作单元个数为 $\Theta(1)$ 。

3.2.4　堆的排序

一、算法描述

算法 3.9　基于堆的排序

输入：数组 H[],数组的元素个数 n

输出：按递增顺序排序的数组 A[]

```
template <class Type>
void heap_sort(Type A[],int n)
{
    int  i;
    make_heap(A,n);
    for (i=n,i>1;i--) {
        swap(A[1],A[i]);
        sift_down(A,i-1,1);
    }
}
```

二、算法分析

1、执行时间：make_heap 的执行时间为 $\Theta(n)$

sift_down 执行 $n-1$ 次，每次花费 $O(\log n)$ 时间，总花费时间 $O(n\log n)$。

所以， heap_sort 的运行时间是 $O(n\log n)$

3、工作空间：$\Theta(1)$。

3.3 基数排序

基于比较的排序算法，下界为 $\Omega(n\log n)$。

基数排序方法可以按线性时间运行。

3.3.1 基数排序算法的思想方法

n 个元素的链表 $L=\{a_1,a_2,\cdots\cdots,a_n\}$，每个元素关键字的值有如下形式：

$$d_k d_{k-1}\cdots\cdots d_1 \qquad 0\le d_i\le 9,\quad 1\le i\le k$$

1、 $m=1$

2、按关键字的数字 d_m，把元素分布到 10 个链表 $L_0,L_1,\cdots\cdots,L_9$，使得关键字的 $d_m=i$ 的元素，都分布在链表 L_i 中；

3、把 10 个链表，按照链表的下标由 0 到 9 的顺序重新链接成一个新的链表 L。

4、 $m=m+1$，若 $m\le k$，转 2，否则结束

例 3.4　假设链表 L 中有如下 10 个元素，其关键字值分别为：3097、3673、2985、1358、6138、9135、4782、1367、3684、0139。

第一步，按关键字中的数字 d_1，把 L 中的元素分布到链表 $L_0\sim L_9$ 的情况如下：

L_0	L_1	L_2	L_3	L_4	L_5	L_6	L_7	L_8	L_9
		4782	3673	3684	2985		3097	1358	0139
					9135		1367	6138	

把 $L_0\sim L_9$ 的元素顺序链接到 L 后，在 L 中的元素顺序如下：

L ： 4782 3673 3684 2985 9135 3097 1367 1358 6138 0139

第二步，按数字 d_2，把 L 中的元素分布到 $L_0\sim L_9$ 的情况如下：

L_0	L_1	L_2	L_3	L_4	L_5	L_6	L_7	L_8	L_9
			9135		1358	1367	3673	4782	3097
			6138					3684	
			0139					2985	

把 $L_0\sim L_9$ 的元素顺序链接到 L 后，在 L 中的元素顺序如下：

L ： 9135 6138 0139 1358 1367 3673 4782 3684 2985 3097

第三步，按数字 d_3，把 L 中的元素分布到 $L_0\sim L_9$ 的情况如下：

L_0	L_1	L_2	L_3	L_4	L_5	L_6	L_7	L_8	L_9
3097	9135		1358			3673	4782		2985
	6138		1367			3684			
	0139								

把 $L_0 \sim L_9$ 的元素顺序链接到 L 后，在 L 中的元素顺序如下：

L ： 3097 9135 6138 0139 1358 1367 3673 3684 4782 2985

第四步，按数字 d_4，把 L 中的元素分布到 $L_0 \sim L_9$ 的情况如下：

L_0	L_1	L_2	L_3	L_4	L_5	L_6	L_7	L_8	L_9
0139	1358	2985	3097	4782		6138			9135
	1367		3673						
			3684						

把 $L_0 \sim L_9$ 的元素顺序链接到 L 后，在 L 中的元素顺序如下：

L ：0139 1358 1367 2985 3097 3673 3684 4782 6138 9135

在第四步之后，链表中的所有关键字都已经排序了。

3.3.2 基数排序算法的实现

用双循环链表，用成员变量 prior 指向前一个元素，用成员变量 next 指向下一个元素。

算法 3.10 基数排序

输入：存放元素的链表 L，元素个数 n，及关键字的数字位数 k

输出：按递增顺序排序的链表 L

```
template <class Type>
void radix_sort(Type *L,int k)
{
   Type  *Lhead[10],*p;
   int  i,j;
   for (i=0;i<10;i++)            /* 分配 10 个链表的头结点 */
       Lhead[i]  = new Type;
   for (i=0;i<k;i++) {
       for (j=0;j<10;j++)        /* 把 10 个链表置为空表 */
           Lhead[j]->prior = Lhead[j]->next = Lhead[j];
       while (L->next!=L) {
           p = del_entry(L);      /* 取 L 的第一个元素于 p 并把它从 L 删去 */
           j = get_digital(p,i);  /* 从 p 所指向的元素关键字取第 i 个数字 */
           add_entry(Lhead[j],p); /* 把 p 加入链表 Lhead[j]的表尾 */
       }
```

```
        for (j=0;j<10;j++)
            append(L,Lhead[j]);     /* 把 10 个链表的元素链接到 L */
    }
    for (i=0;i<10;i++)              /* 释放 10 个链表的头结点 */
        delete(Lhead[i]);
}
```

算法 3.11 取下并删去双循环链表的第一个元素

输入：链表的头结点指针 L

输出：被取下第一个元素的链表 L，指向被取下元素的指针，

```
template <class Type>
Type *del_entry(Type *L)
{
    Type  *p;
    p = L->next;
    if (p!=L) {
        p->prior->next = p->next;
        p->next->prior = p->prior;
    }
    else p = NULL;
    retuen p;
}
```

算法 3.12 把一个元素插入双循环链表的表尾

输入：链表头结点的指针 L，被插入元素的指针 p

输出：插入了一个元素的链表 L

```
template <class Type>
void add_entry(Type *L,Type *p)
{
    p->prior = L->prior;
    p->next = L;
    L->prior->next = p;
    L->prior = p;
}
```

算法 3.13 取 p 所指向元素关键字的第 i 位数字(最低位为第 0 位)

输入：指向某元素的指针 p，该元素关键字的的第 i 位数字

输出：该元素关键字的的第 i 位数字

```
template <class Type>
```

```
int get_digital(Type *p,int i)
{
    int key;
    key = p->key;
    if (i!=0)
        key = key / power(10,i);
    return key % 10;
}
```

算法 3.14 把链表 L1 附加到链表 L 的末端

输入：指向链表 L 及 L1 的头结点指针

输出：附加了新内容的链表 L

```
template <class Type>
void append(Type *L,Type *L1)
{
    if (L1->next!=L1) {
        L->prior->next = L1->next;
        L1->next->prior = L->prior;
        L1->prior->next = L;
        L->prior = L1->prior;
    }
}
```

算法 3.11、3.12、3.14 的执行时间是常数时间。

算法 3.13 的执行时间取决于函数 power(x,y) 的执行时间，power 函数计算以 x 为底的 y 次幂。假定，x 是有限长度的整数，后面将说明，该函数的执行时间将是 $\Theta(\log y)$，如果 y 是一个大于 0 的常整数，则该函数的执行时间也是常数。所以，它们都是 $\Theta(1)$。

3.3.3　基数排序算法的分析

一、复杂性分析

算法的执行时间是 $\Theta(kn)$。当 k 是常数时，它的执行时间是 $\Theta(n)$。

工作单元为 $\Theta(1)$。

二、正确性证明

用归纳法证明，算法经过 k 步（假定元素的关键字有 k 位数字）的重新分布和重新链接之后，序列中的元素是按顺序排列的：

$i=1$：L 中的元素按其关键字的最低位数字分布到 10 个链表，然后，再把这些链表按顺序链接成一个链表 L，则 L 中的元素将按其关键字的最低数字排序；

$i=2$：L 中的元素再按其关键字的十位数字分布到 10 个链表

令 x 和 y 是 L 序列中任意两个元素，

x 的关键字的最低两位数字分别为 a 、b ，

y 的关键字的最低两位数字分别为 c 、d 。

1）若 $a>c$ ，则 x 被分布到序号较高的链表，y 被分布到序号较低的链表。

重新链接到 L 去时，y 先于 x 被链接到 L ，

它们是按最低两位数字的顺序排序的。

2）若 $c>a$ 同理可证。

3）$a=c$ ，则它们分布在同一个链表。

这时，若 $b>d$ ，则 y 先于 x 被分布到这个链表。

重新链接到 L 去时，仍维持这个顺序，

它们也按最低两位数字的顺序排列。

x 和 y 是任意的，所以，链表中的元素都按最低两位数字的顺序排列。

归纳步的证明类似，留作练习。

3.4 离散集合的操作

例：对集合 $S=\{1,2,\cdots,8\}$ 定义如下的等价关系：

$$R=\{<x,y>|x\in S\wedge y\in S\wedge(x-y)\%3=0\}$$

求 S 关于 R 的等价类，

1. 初始化：$\{1\}\ \{2\}\ \{3\}\ \{4\}\ \{5\}\ \{6\}\ \{7\}\ \{8\}$；
2. $1R4$，有：$\{1,4\}\ \{2\}\ \{3\}\ \{5\}\ \{6\}\ \{7\}\ \{8\}$；
3. $4R7$，有：$\{1,4,7\}\ \{2\}\ \{3\}\ \{5\}\ \{6\}\ \{8\}$；
4. $2R5$，有：$\{1,4,7\}\ \{2,5\}\ \{3\}\ \{6\}\ \{8\}$；
5. $5R8$，有：$\{1,4,7\}\ \{2,5,8\}\ \{3\}\ \{6\}$；
6. $3R6$，有：$\{1,4,7\}\ \{2,5,8\}\ \{3,6\}$；

find 操作：把元素 x 和 y 所在的集合找出来，

union 操作：把两个集合合并成一个集合。

3.4.1 离散集合的数据结构

一、第一种数据结构

```
struct Tree_node {
    struee Tree_node *p;       /* 指向父亲结点的指针 */
    Type x;                    /* 存放在结点中的元素 */
}
```

集合可以由集合中的元素来命名，这个元素就称为该集合的代表元。

集合中的所有元素，都有资格作为集合的代表元。

要把元素 x 所代表的集合，与元素 y 所代表的集合合并起来，只要分别找出元素 x 和元素 y 所在集合的根结点，使元素 y 的根结点的父指针指向元素 x 的根结点即可。

图 3.7（a）表示由集合$\{1, 3, 5, 8\}$, $\{2, 7, 10\}$, $\{4, 6\}$, $\{9\}$ 所组成的森林；

图 3.7（b）表示由元素 1 所代表的集合、与元素 7 所代表的集合合并的例子。

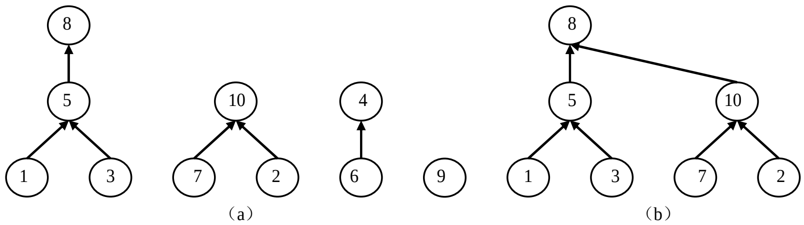


图 3.7　离散集合的表示形式

由此，可以把离散集合中 find 操作和 union 操作的含义定义如下：

- find(x)：寻找元素 x 所在集合的根结点；
- union(x, y)：把元素 x 和元素 y 所在集合合并成一个集合。

缺点：树的高度可能很大，变成退化树，成为线性表。如图 3.8(a)。

find 操作可能需要 $\Omega(n)$ 时间。

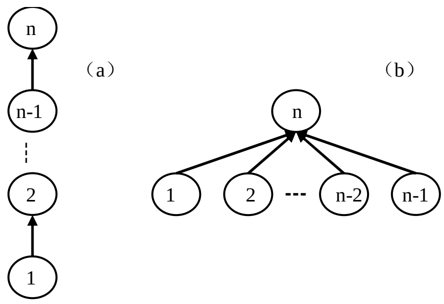


图 3.8　n 个集合合并的两种情况

二、改进的数据结构

```
struct Tree_node {
    struee Tree_node *p;          /* 指向父亲结点的指针 */
    int rank;                     /* 结点的秩 */
    Type x;                       /* 存放在结点中的元素 */
};
typedef struct Tree_node NODE;
```

结点的秩等于以该结点作为子树的根时，该子树的高度。

union(x, y)操作：令 x 和 y 是当前森林中两棵不同树的根结点，

如果 $rank(x) > rank(y)$，就把 x 作为 y 的父亲，并使 $rank(y)$ 加 1

例：图 3.8（b）表示采用这个方法对 n 个集合进行合并时的情况。

三、用数组存放元素

```
struct Tree_node {
    int index;                    /* 指向父亲结点的下标 */
    int rank;                     /* 结点的秩 */
    Type x;                       /* 存放在结点中的元素 */
};
struct Tree_node node[n];
```

这时，父结点的指针，用该结点在数组中的下标表示。

3.4.2　union、find 操作及路径压缩

一、路径压缩

find 操作时，找到根结点 y 之后，再沿着这条路径，改变路径上所有结点的父指针，使其直接指向 y 。如图 3.9 所示。

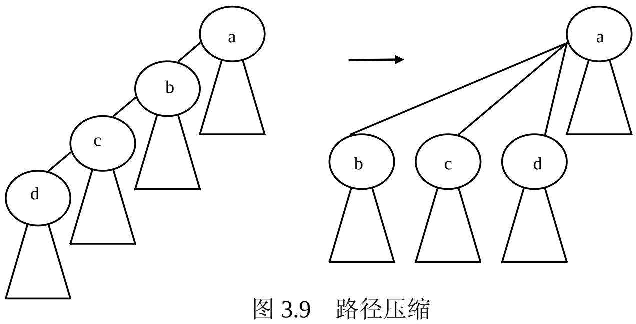


图 3.9　路径压缩

二、算法描述

算法 3.15　离散集合的 find 操作

输入：指向结点 x 的指针 xp

输出：指向结点 x 所在集合的根结点的指针 yp

```
NODE *find(NODE *xp)
{
    NODE *wp, *yp = xp, *zp = xp;
    while (yp->p!=NULL) {              /* 寻找 xp 所在集合的根结点 yp */
        yp = yp->p;
```

```
    while (zp->p!= NULL) {             /* 路径压缩 */
        wp = zp->p
        zp->p = yp;
        zp = wp;
    }
    return yp;
}
```

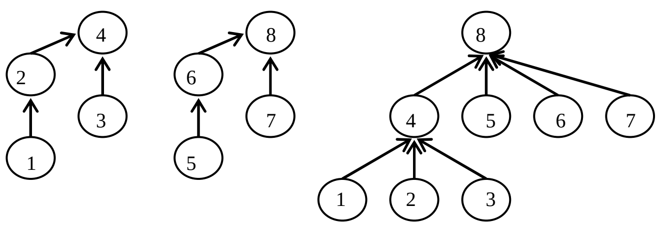
算法 3.16 离散集合的 union 操作

输入：指向结点 x 和结点 y 的指针 xp 和 yp

输出：结点 x 和结点 y 所在集合的并集,指向该并集根结点的指针

```
NODE *union(NODE *xp,NODE *yp)
{
    NODE *up,*vp;
    up = find(xp);
    vp = find(yp);
    if (up->rank<=vp->rank) {
        up->p = vp;
        if (up->rank==vp->rank)
            vp->rank++;
        up = vp;
    }
    else
        vp->p = up;
    return up;
}
```

例 3.5 集合{1, 2, 3, 4}, {5, 6, 7, 8}，如图 3.10（a）所示，在执行了 union（1, 5）之后，结果如图 3.10（b）所示。在 union 操作中，对结点 1 和 5 执行了 find 操作，结点 1 和 5 的路径都被压缩了。



(a) （b）

图 3.10 集合 union 操作的例子

三、算法分析

x 是树中的任意结点，$x.p$ 指向 x 的父亲结点。得到下面两个结论。

1、**结论 3.1** $x.p->rank \geq x.rank+1$。

2、**结论 3.2** $x.rank$ 的初始值为 0，在一系列的 union 操作中递增，直到 x 不再是树的根结点为止。一旦 x 变为另一个结点的儿子，它的秩就不再改变。

3、**引理 3.1** 若结点 x 的秩为 $x.rank$，则以 x 为根的树，其结点数至少为 $2^{x.rank}$。

（含义：结点数至少为 $n=2^{x.rank}$ 的树，其高度至多为 $\log n = x.rank$）

证明 用归纳法证明。

1. 开始时，x 本身是一棵数，其秩 $x.rank=0$，其结点数等于 $2^0=1$，引理成立。
2. 假定 x 和 y 分别是两棵树的根结点，其秩分别是 $x.rank$，和 $y.rank$。
   在 union(x, y)操作之前，x 和 y 为根的树，其结点数分别至少为 $2^{x.rank}$ 和 $2^{y.rank}$。
   在 union(x, y)操作之后，有三种情况：
   （1）若 $x.rank < y.rank$，在 union 操作之后，新的树以 y 为根结点，且 y 的秩不变，而树的结点数增加。因此，新树的结点数至少为 $2^{y.rank}$。引理成立。
   （2）若 $x.rank > y.rank$，同理可证。
   （3）若 $x.rank = y.rank$，则两棵树的结点数至少都是 $2^{y.rank}=2^{x.rank}$。
   在 union 操作之后，新树的结点数至少为 $2\cdot 2^{y.rank}=2^{y.rank+1}=2^{x.rank+1}$。
   若新树以 y 为根结点，则 y 的秩 $y.rank$ 增 1；
   否则，x 的秩 $x.rank$ 增 1；在这两种情况下，引理都成立。

4、**结论 3.3** find 操作的执行时间为 $O(\log n)$。

证明：如果 x 是树的根，x 的秩就是树的高度。

根据引理 3.1，结点数为 n，则该树的高度至多为 $\lfloor \log n \rfloor$。

find 操作最多执行 $\log n$ 次判断根结点的操作、

以及 $\log n$ 次对非根结点进行的路径压缩操作

5、**结论 3.4** union 操作的执行时间为 $O(\log n)$。

证明： union 操作除了执行两次 find 操作外，其余花费 $O(1)$ 时间。

6、**定理 3.1** 连续执行 m 次 union 和 find 操作，在最坏情况下，所需要的执行时间是 $O(m\log^* n) \approx O(m)$。

其中，$\log^* n$ 定义为：

$$\log^* n = \begin{cases} 0 & n=0,1 \\ \min\{i \geq 0 \mid \underbrace{\log\log\cdots\log n}_{i\text{ 次}} \leq 1\} & n \geq 2 \end{cases}$$

例如，$\log^* 2=1$，$\log^* 2^2=2$，$\log^* 2^4=3$，$\log^* 2^{16}=4$，$\log^* 2^{65536}=5$。在几乎所有的实际应用中，$\log^* n \leq 5$。所以，它所需要的执行时间实际上将是 $O(m)$。