第三章 排序问题和离散集合的操作

3.1 合并排序

3.2 基于堆的排序

3.3 基数排序

3.4 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

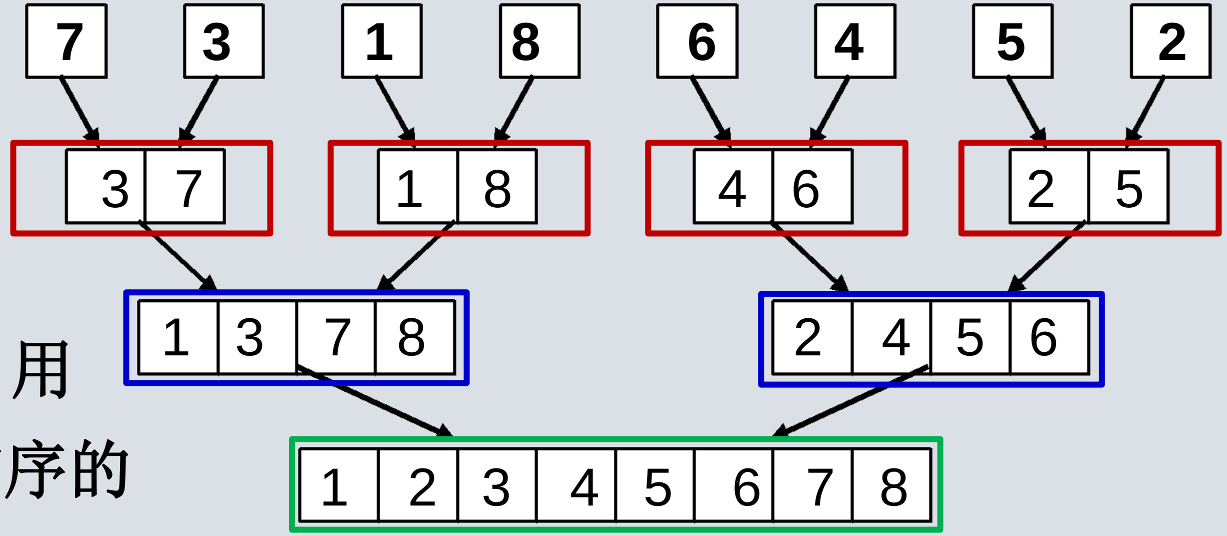
3.1.2 合并排序算法的分析

- 合并排序（**merge sort**）：就是将两个（或两个以上）有序表合并成一个新的有序表。
- 合并排序基本思路：
  - 首先将含有*n*个节点的待排序序列看作有*n*个长度为1的有序子表组成，将他们依次两两合并，得到长度为2的若干有序子表；
  - 然后再对这些子表进行两两合并，得到长度为4的若干有序子表；
  - …
  - 重复上述过程，一直重复到最后的子表长度为*n*，从而完成排序过程。

3.1.1 合并排序算法的实现

1. 合并排序算法的思想方法

- 划分为四对，两两合并，用 merge 算法合并成四个有序的序列；



- 把四个序列划分成两对，用 merge 算法合并成两个有序的序列;

- 再利用merge算法合并成一个有序的序列。

2. 合并排序算法的描述

算法3.1 合并排序算法

```
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);
            i = i + t;
        }
        if (i+s<n)
            merge(A,i,i+s-1,n-1);
    }
}
```

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

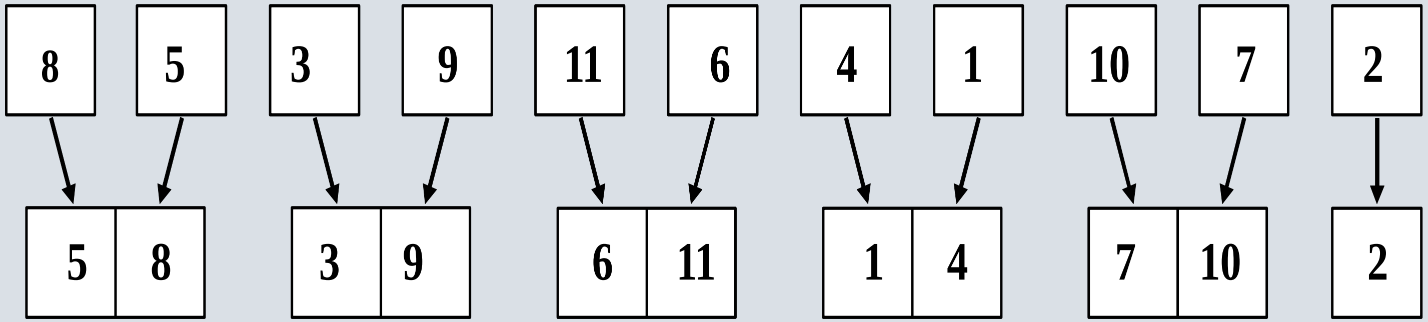
i,i+s-1,i+t-1定义被合并的两个序列的边界。

```
void merge(int A[],int p,int q,int
r)
{
  int *bp=new int[m];
  int i,j,k;
  i=p;j=q+1;k=0;
  while(i<=q&&j<=r){
    if(A[i]<=A[j])
     bp[k++]=A[i++];
    else bp[k++]=A[j++];
 }
```

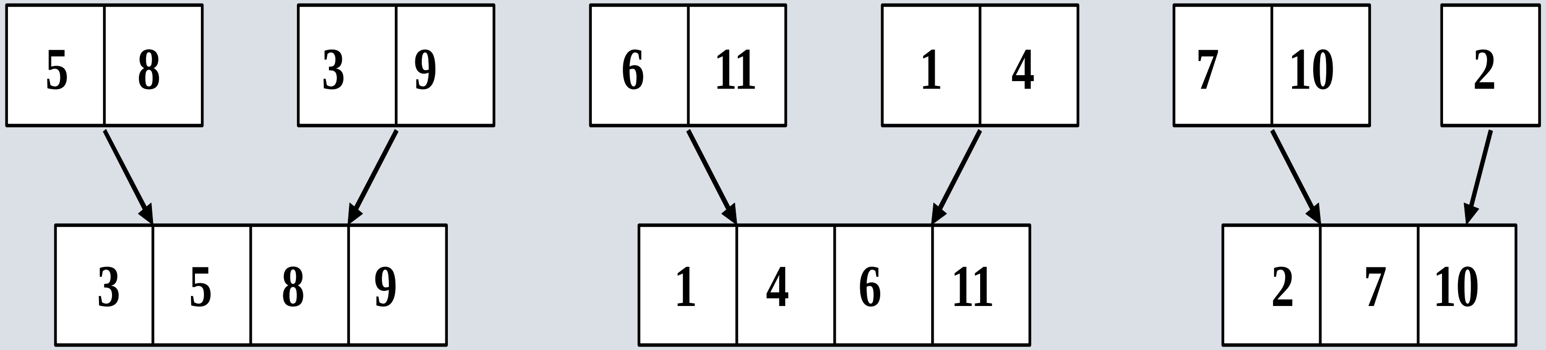
```
if(i==q+1){
  for(;j<=r;j++)
    bp[k++]=A[j++];}
else
   {for(;i<=q;i++)
     bp[k++]=A[i++];}
k=0;
for(i=p;i<=r;i++)
 A[i++]=bp[k++];
delete bp;
}
```

3. 合并排序算法的实现过程

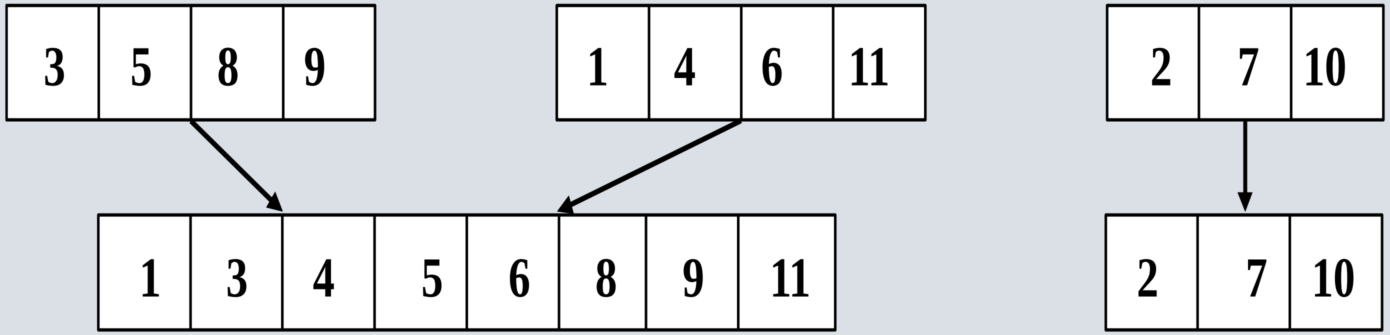
1）在第一轮循环，s = 1、t =2，有 5 对 1 个元素的序列进行合并，当 i = 10时，i + t = 12 > n ，退出内部的 while 循环。但 i + s = 11，不小于*n*，所以，不执行第 12 行的合并工作，余留一个元素没有处理。



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。

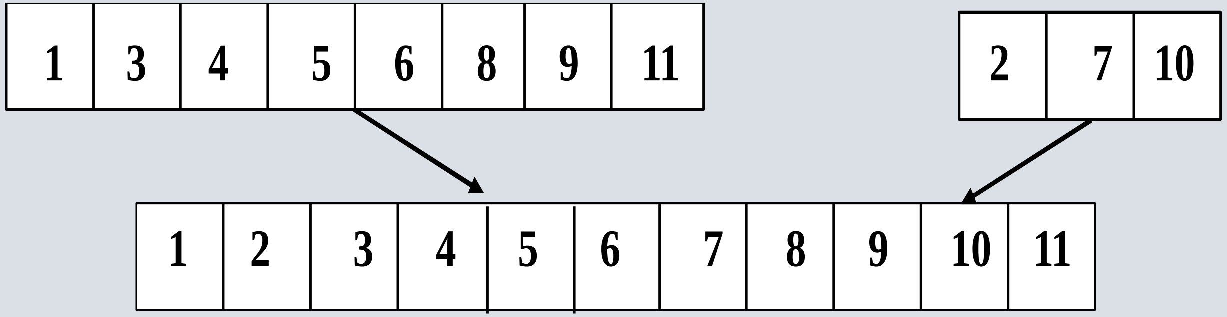


3）在第三轮 s = 4， t = 8, 有一对四个元素的序列进行合并，在 i = 8 时， i + 8 = 16 > n ， 退出内部的while循环。i + s = 12 > n， 不执行第12行的合并工作，余留一个序列没有处理 。



4）在第四轮，$s = 8, t = 16$。在 $i = 0$ 时，$i + t = 16 > n$，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）在第五轮，因为 $t = 16 > n$，所以退出外部的while循环，结束算法 。



3.1.2 合并排序算法的分析

1. 时间复杂性

- 假定 $n = 2^k$ ，外部 while 循环的循环体的执行次数为 $k = \log n$ 次；
- 合并排序算法由两个嵌套的循环组成，算法的执行时间取决于内部while循环merge算法的执行次数，以及每次执行merge算法时的元素比较次数；
- 外部while循环体的执行次数是 $k=\log n$；
- 内部while循环merge算法的比较次数，由2个比较的序列长度决定，如果待排序序列与排序结果相同是最好情况，比较次数至少是 $\min(n_1, n_2)$，如果是逆序是最差情况，比较次数最多是 n_1+n_2-1。

	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	最多
第1轮	$n/2$	1	$n/2$	2	$(n/2)*1$	$(n/2)*1$
第2轮	$n/2^2$	2^1, $2^2-1=3$	$n/2^2$	2^2	$(n/2^2)*2^1$	$(n/2^2)*(2^2-1)$
第3轮	$n/2^3$	2^2, $2^3-1=7$	$n/2^3$	2^3	$(n/2^3)*2^2$	$(n/2^3)*(2^3-1)$
	…	…	…	…	…	…
第j轮	$n/2^j$	2^{j-1}, 2^j-1	$n/2^j$	2^j	$\boldsymbol{(n/2^j)*2^{j-1}}$	$\boldsymbol{(n/2^j)*(2^j-1)}$

所有比较都最好情况执行时间至少是

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

所有比较都最差情况执行时间至多是

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

Ω(nlogn), O(nlogn), Θ(nlogn)

2. 空间复杂性

每调用一次**merge**便分配一个适当大小的缓冲区，退出**merge**便释放它。

最后一次调用**merge**所分配的缓冲区最大，此时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。

故合并排序算法的空间复杂度为$\Theta(n)$ 。

3.3 基数排序

- 基于比较的排序算法，下界Ω(***n*****log*****n***)；
- 基数排序方法是非比较型的排序，按此法设计的算法几乎都按线性时间运行。

3.3.1 基数排序算法的思想方法

3.3.2 基数排序算法的实现

3.3.3 基数排序算法的分析

3.3.1 基数排序算法的思想方法

令$L=\{a_1, a_2,\ldots, a_n\}$是一个具有$n$个元素的链表，每个元素关键字的值都由$k$个数字组成，即

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

排序过程：

- 1. 将这n个元素按照d_1的大小进行分布，分布到$L_0,L_1,\ldots,L_9$中去，分布好以后再链接成一个新的链表L.
- 2. 再继续做d_2的大小进行分布，一直到最高位d_k。最后生成的就是排好序的序列。

最低位优先 LSD

例：链表 *L* 中元素的关键字值分别为：

3097、3673、2985、1358、6138、9135、4782、1367、3684、0139

1. 按关键字中的数字 d1，把 L 中的元素分布到链表情况:

L0	*L1*	*L2*	*L3*	*L4*	*L5*	*L6*	*L7*	*L8*	*L9*
		4782	3673	3684	2985		3097	1358	0139
					9135		1367	6138	

把 *L0* ~*L9* 的元素顺序链接到 *L* 后，*L* 中的元素顺序如下

4782 3673 3684 2985 9135 3097 1367 1358 6138 0139

2. 按关键字中的数字 d_2，把 L 中的元素分布到链表情况:

4782　3673　3684　2985　9135　3097　1367　1358　6138　0139

$L0$	$L1$	$L2$	$L3$	$L4$	$L5$	$L6$	$L7$	$L8$	$L9$
			9135		1358	1367	3673	4782	3097
			6138					3684	
			0139					2985	

把 $L0$ ~$L9$ 的元素顺序链接到 L 后，L 中的元素顺序如下

9135　6138　0139　1358　1367　3673　4782　3684　2985　3097

3. 按关键字中的数字 d_3，把 L 中的元素分布到链表情况:

9135 6138 0139 1358 1367 3673 4782 3684 2985 3097

$L0$	$L1$	$L2$	$L3$	$L4$	$L5$	$L6$	$L7$	$L8$	$L9$
3097	9135		1358			3673	4782		2985
	6138		1367			3684			
	0139								

把 $L0$ ~$L9$ 的元素顺序链接到 L 后，L 中的元素顺序如下

3097 9135 6138 0139 1358 1367 3673 3684 4782 2985

4. 按关键字中的数字 d_4，把 L 中的元素分布到链表情况:

3097 9135 6138 0139 1358 1367 3673 3684 4782 2985

*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 ~*L*9 的元素顺序链接到 *L* 后，*L* 中的元素顺序如下

0139 1358 1367 2985 3097 3673 3684 4782 6138 9135

1. n 个元素的链表 $L=\{a_1,a_2,\ldots a_n\}$，元素关键字的值由 k 个数字组成。关键字的值形式如下：

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

2. 按照关键字的最低位数字d_1，把元素分布到 10 个链表$L_0,L_1,\ldots L_9$中，使得关键字的 $d_1=0$ 的元素，都分布在链表 L_0 中； $d_1=1$ 的元素，都分布在链表 L_1 中；如此等等。

在这一步结束之后，L_i 包含关键字最低位为 i的元素，其中，$0 \le i \le 9$。

3. 把这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。此时，新链表中的所有元素，都按关键字中最低位数字顺序排序。

4. 第二步，按照元素关键字的次低位数字 d_2，重复第一步工作。此时，所形成的新链表中，所有元素都按关键字最低两位数字顺序排序。

5. 如此类推，在第 k 步，按照元素关键字的最高位数字 d_k重复第一步工作。此时，所形成的新链表中，所有元素都按关键字的所有数字顺序排序。

练习：求递增排序：411，768，734，890，659，134，098

d1:

L0	L1	L2	L3	L4	L5	L6	L7	L8	L9
890	411			734				768	659
				134				098	

新的序列是：890，411，734，134，768，098，659

d2：

L0	L1	L2	L3	L4	L5	L6	L7	L8	L9
	411		734		659	768			890
			134						098

新的序列是：411，734，134，659，768，890，098

d3:

L0	L1	L2	L3	L4	L5	L6	L7	L8	L9
089	134			411		659	734	890	
							768		

新的序列是：089，134，411，659，734，768，890

3.3.2 基数排序算法的实现

1. 数据结构：

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

2. 相关操作：

1）Type ***del_entry**(Type *L)**

取下并删去双循环链表的第一个元素

2）void **add_entry(Type *L,Type *p)**

把一个元素插入双循环链表的表尾

3） int **get_digital(Type * p,int i)**

取*p*所指向元素关键字的第*i*位数字

4）void **append(Type *L,Type *L1)**

把链表*L*1附加到链表*L*的末端

取下并删去双向循环链表的第一个元素

```
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;
  return p;
}
```

将一个元素插入双向循环链表的表尾

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

取p所指向元素关键字的第i位数字(个位为0)

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

把链表L1附加到链表L的末端

```
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;
}}
```

3. 算法描述：

```
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个链表的头节点 */
        Lhead[i]  = new Type;
```

```
for (i=0,i<k,i++) {       /* k为关键字的数字位数*/
    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)  /* 取下L的第一个元素,使p指向该元素 */
        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);  /* p指向的元素加入链表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个链表的头节点 */
    delete(Lhead[i]);
}
```

3.3.3 基数排序算法的分析

1. 复杂性分析

算法的执行时间是$\Theta(kn)$;

工作单元为$\Theta(1)$。

2. 正确性证明

用归纳法证明：算法经k步(假定关键字位数为k)重新分布和重新链接后，序列中元素按顺序排列。

3.2 基于堆的排序

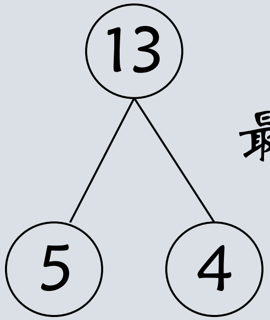
堆的定义

定义 n个元素构成一个堆，当且仅当它的关键字

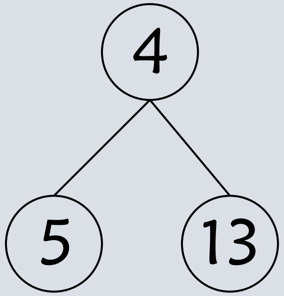
序列 $k_1, k_2, \cdots, k_n$，满足：

小顶堆（最小堆): $k_i \le k_{2i}, k_i \le k_{2i+1}, 1 \le i \le \lfloor n/2 \rfloor$

大顶堆（最大堆): $k_i \ge k_{2i}, k_i \ge k_{2i+1}, 1 \le i \le \lfloor n/2 \rfloor$



最大堆



最小堆

堆的性质

堆可以看做一棵完全二叉树，假设高度为d，具有如下性质：

- 1.所有叶结点不是处于第d层，就是处于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.对最大堆，每个结点的关键字都大于其子结点关键字; 对最小堆， 每个结点的关键字都小于其子结点关键字。

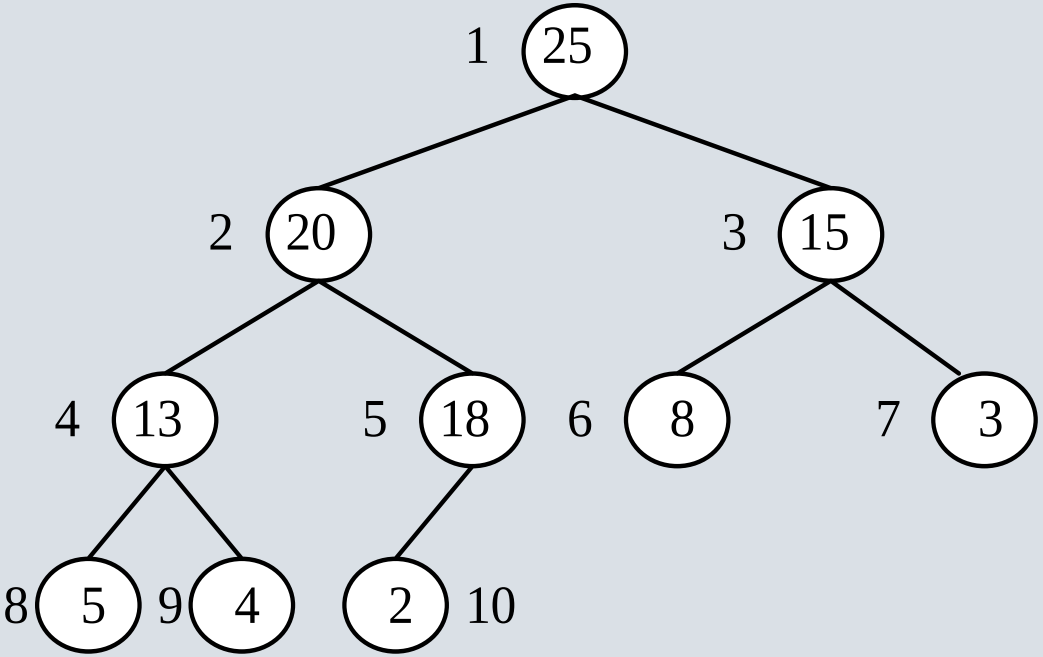
堆的数据结构

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，则其左儿子结点存放在 $H[\,2i\,]$；如果它有右儿子结点，则其右儿子结点存放在 $H[\,2i+1\,]$;

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



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

堆的操作

堆的操作如下：

Void sift_up(Type H[],int i) 把堆中的第i个元素上移

Void sift_down(Type H[],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) 删去堆中的第i个元素

Void delete_max(Type H[], int &n) 从非空最大堆中删除并回送最大元素

Void make_head(Type H[], int n) 使数组H中的元素按堆的结构重新组织

1. 元素上移操作

假定使用最大堆，沿 *H*[*i*] 到根的路线，把 *H*[*i*]向上移动。移动过程中，若大于其父结点，就与父结点交换位置。否则，操作结束。

算法3.2 元素上移操作

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

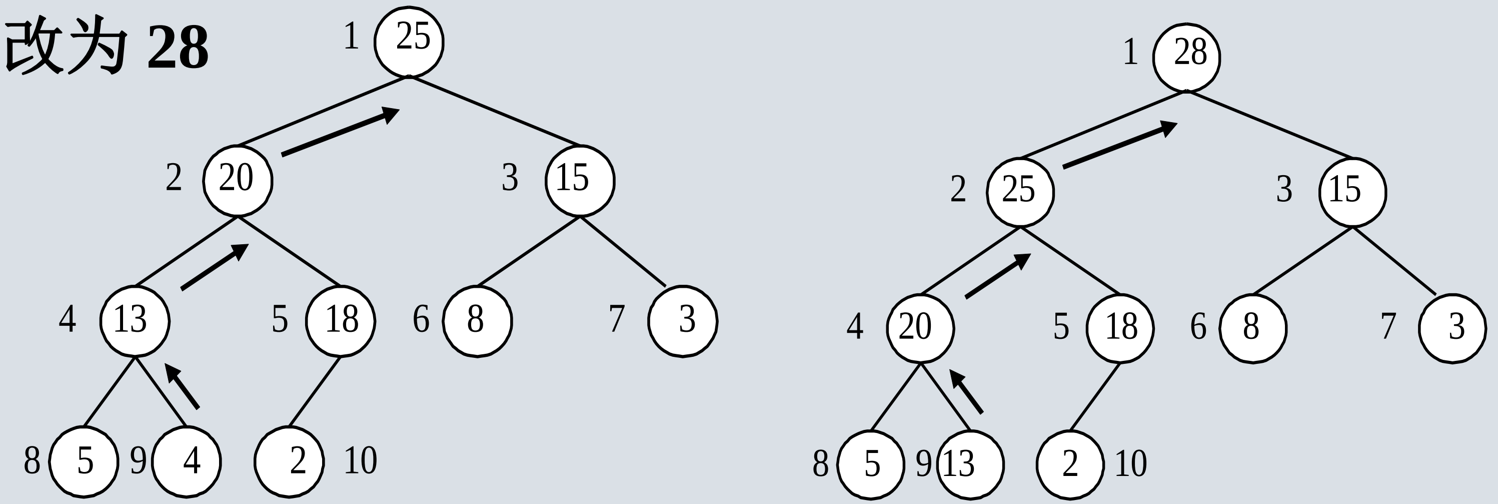
上移过程没有完成 &&
当前需要上移的元素不是根结点

当前结点移至其父结点

效率分析

- 元素每移动一次，就执行一次比较操作；
- 移动后，元素所在结点的层数减1；
- 执行时间：n 个元素共⌊ **logn** ⌋层结点，最多执行⌊**logn**⌋次元素比较操作，**sift_up**的执行时间是**O(logn)**；
- 工作单元：**Θ(1)**

例：把结点 **9** 中的 **4** 改为 **28**



2. 元素下移操作

在向下移动的过程中，下移元素的关键字和两个子结点中关键字大的子结点比较，若小于子结点的关键字，则与子结点交换位置。否则，操作结束 。

算法3.3 元素下移操作

```
void sift_down(Type H[], int n, int i)
{  Bool done = FALSE;
      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;  } }
```

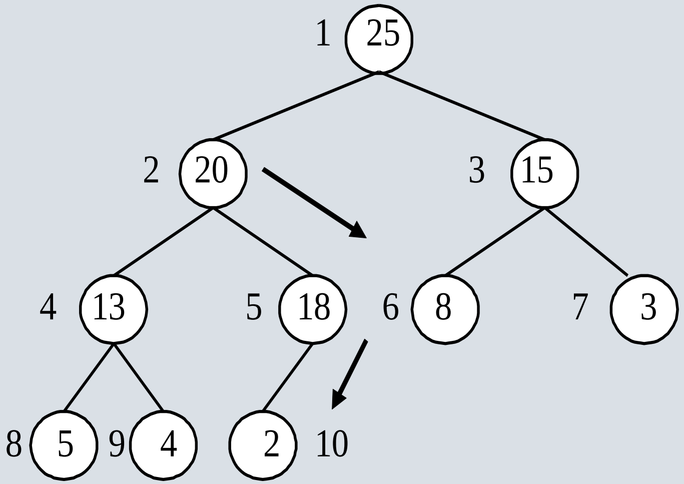
下移过程没有完成 &&
i指向左子结点且该结点不为空

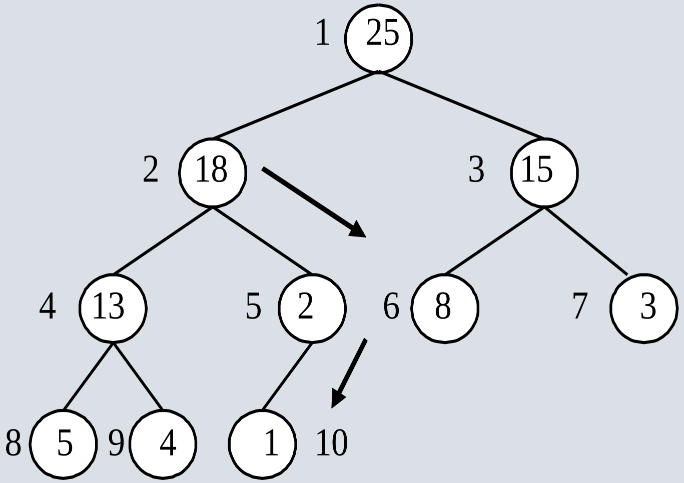
当前结点有右兄弟结点 &&
右兄弟结点值>当前结点

效率分析

- 元素每下移一次，就执行两次比较操作；
- 移动后，元素所在结点的层数增1；
- 执行时间：n 个元素共$\lfloor \log n \rfloor$层结点，最多执行$2\lfloor \log n \rfloor$次元素比较操作，sift_down的执行时间是$O(\log n)$
- 工作单元：$\Theta(1)$

例：把结点 2 中的 20 改为 1





3. 元素插入操作

堆的大小增 1，把 *x* 放到堆的末端，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

```
2. void insert(Type H[ ], int &n, Type x)
3. {
4.     n = n + 1;
5.     H[n] = x;
6.     sift_up(H,n);
7. }
```

执行时间：$O(\log n)$

工作单元：$\Theta(1)$

4. 元素删除操作

用堆中最后一个元素取代 *H*[*i*]，堆的大小减1。对取代它的元素做上移操作、或做下移操作。

算法3.5 元素删除操作

```
template <class Type>
void delete(Type H[ ],int &n,int i)
{
    Type  x;
    x = H[ i ];
    if (i<=n) {  待删除元素不是原先堆的最后一个元素
        H[ i ] = H[n];
        n = n - 1;
        if (H[ i ]>x)    sift_up(H,i);
        else
            sift_down(H,n,i);
    }
}
```

执行时间：$O(\log n)$

工作单元：$\Theta(1)$

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

```
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： $\Theta(1)$

6. 堆的建立(1)

1） 用insert操作制造堆

算法3.7 建造堆的第一种算法

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

效率分析：

插入第i个元素花费 $O(\log i)$，插入n个元素，需花费 $O(n\log n)$

工作单元 $\Theta(n)$

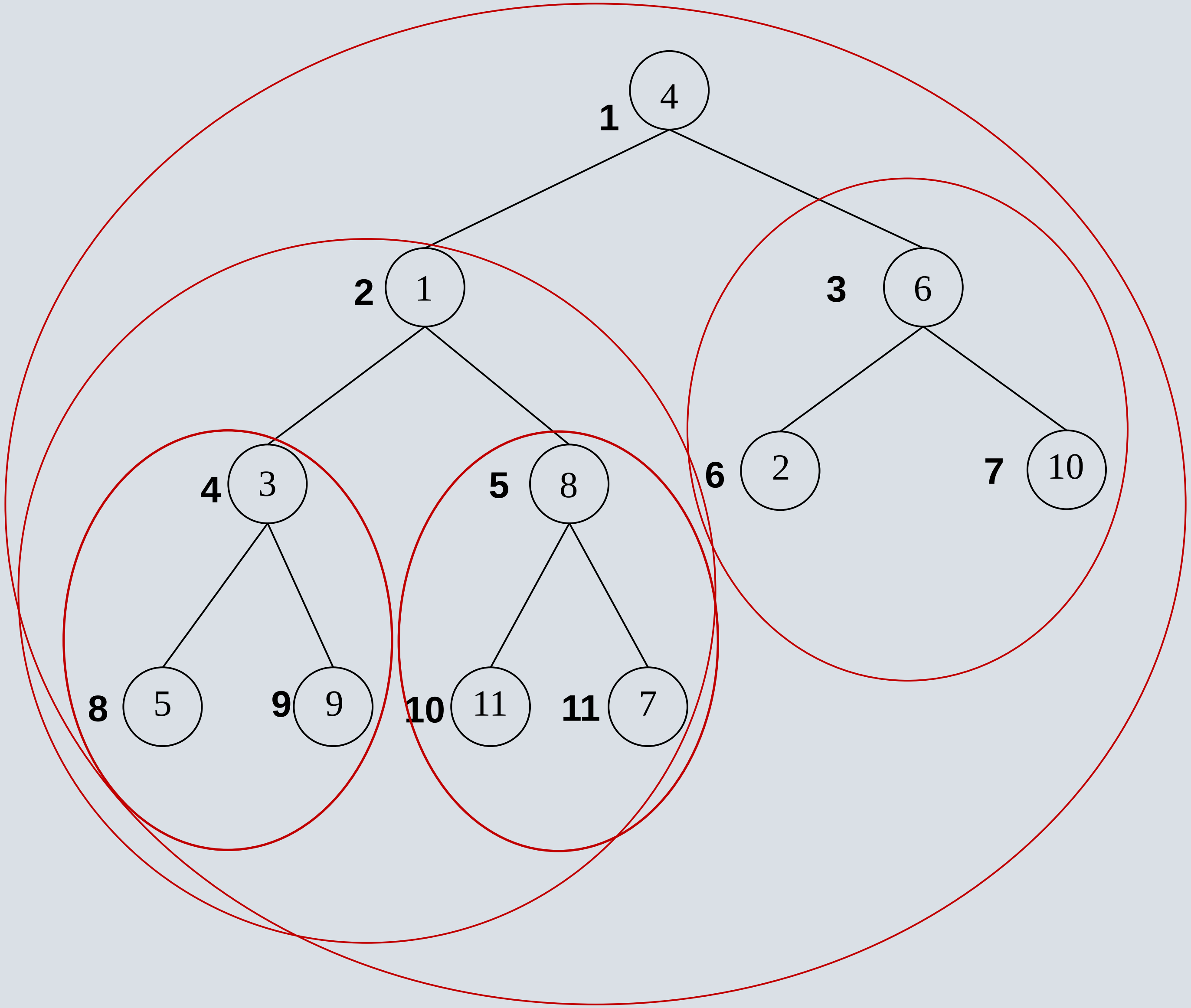
堆的建立(2)

2）把数组本身构造成一个堆

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

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


4
1
1
2
6
3
3
4
8
5
2
6
10
7
5
8
9
9
11
10
7
11

3）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 $k = \lfloor \log n \rfloor$

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 层元素都是叶子结点，无需下移，最多只需对第 0 层到第 $k - 1$ 层的元素执行下移操作。

算法 make_heap 所执行的元素比较次数为：

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

$$= 2k(2^k - 1) - 2((k-1)2^{k+1} - (k-1)2^k - 2^k + 2)$$

$$= 2(k2^k - k) - 2(k2^k - 2^{k+1} + 2)$$

$$= 4 \bullet 2^k - 2k - 4$$

$$= 4n - 2\log n - 4$$

$$< 4n$$

故 make_heap的执行时间为O(n)。

此外，每个结点作下移操作时，至少需做两次元素比较，共$\lfloor n/2 \rfloor$个节点作下移操作，至少需要$2\lfloor n/2 \rfloor$次元素比较，执行时间为$\Omega(n)$。

因此，**make_heap**执行时间为$\Theta(n)$。

工作单元：$\Theta(1)$

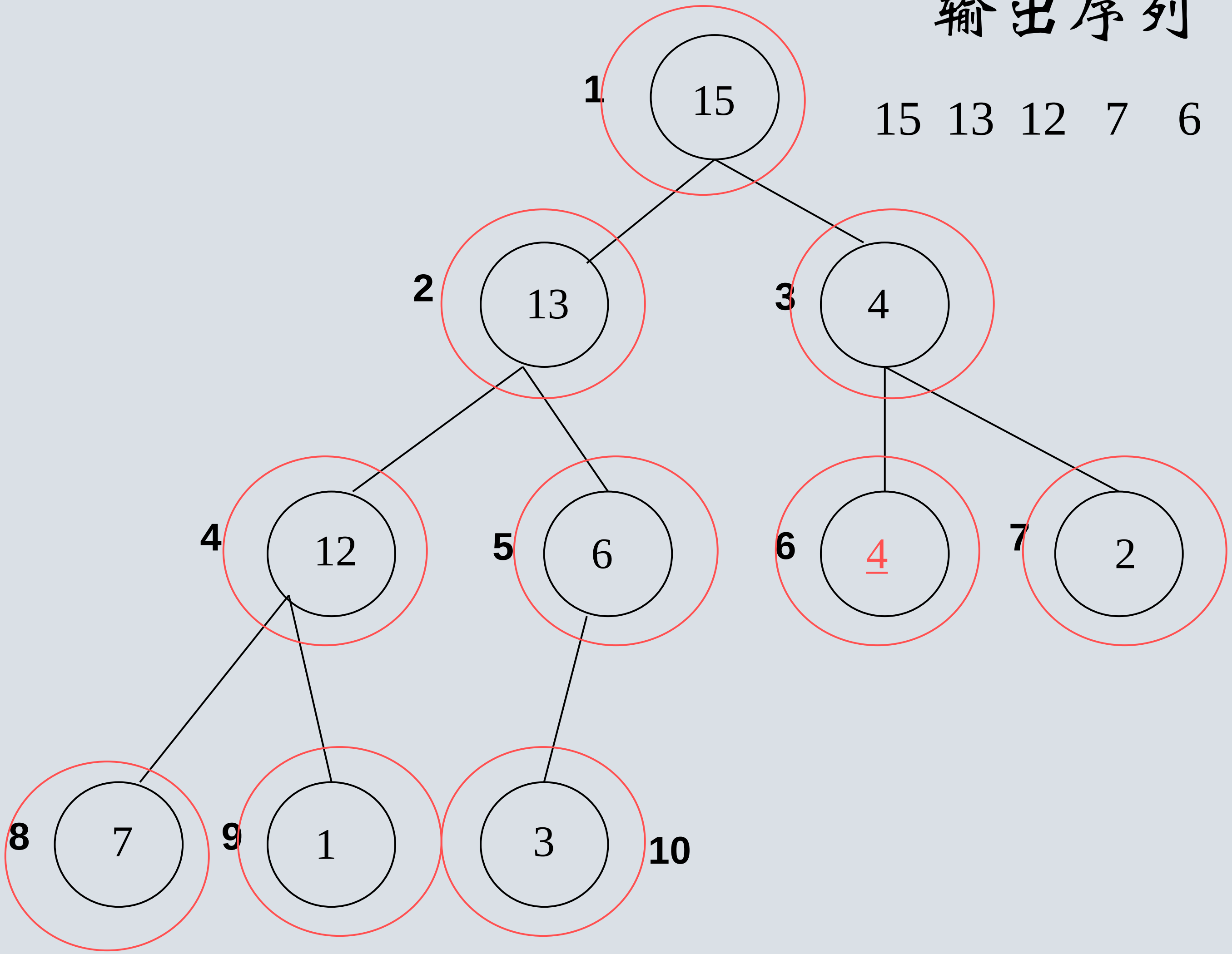
7. 堆的排序

- 假定数组的元素个数为 *n*，*A*[1]~*A*[*n*]为最大堆；
- 交换*A*[1]和*A*[*n*]， *A*[*n*]成为数组中关键字最大的元素；
- 把*A*[*n*]从堆中删去，使堆中的元素个数减1；
- 对*A*[1]做下移操作，使其恢复最大堆的结构；
- *A*[1]~*A*[*n*-1]成为新的最大堆，其元素个数为*n*-1；
- 继续对*A*[1]~ *A*[*n*-1]进行这种操作，则*A*[*n*-1]就成为数组中关键字次大的元素；
- *A*[1]~ *A*[*n*-2]成为新的最大堆，其元素个数为*n*-2；
- 如此继续进行，直到所构成的新堆的元素个数减少到1为止。

此时，数组中的元素就是由小到大排序的。

输出序列

15 13 12 7 6 4 4 3 2 1



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);
    }
}
```

堆排序的性能分析

- 执行时间:
  - **make_heap的执行时间为O(n)**
  - **sift_down执行*n*-1次，每次花费时间O(logn)，总花费时间O(nlogn)**
- **故heap_sort运行时间是O(nlogn)**
- 工作空间: Θ(1)

3.4 离散集合的 union_find 操作（自学）

一 离散集合的数据结构

二 union_find 操作及路径压缩

一 离散集合的数据结构

1. 离散集合的两个操作

- find(*x*)：寻找元素 *x* 所在集合
- union(*x*,*y*)：把元素 *x* 和元素 *y* 所在集合合并成一个集合

2. 离散集合的数据结构

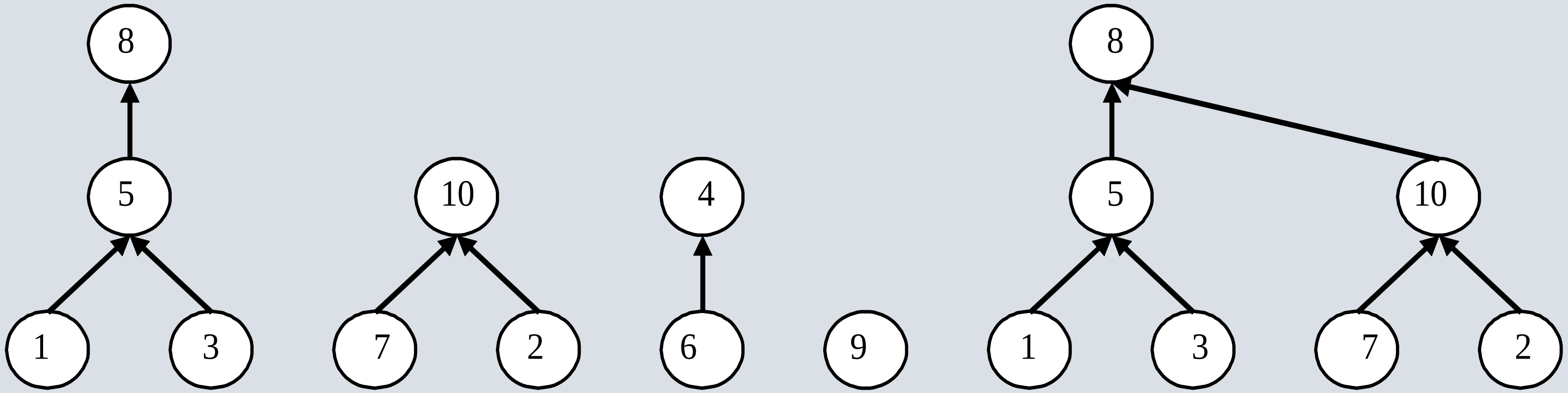
1）离散集合的一种数据结构:

```
struct Tree_node {
   struee Tree_node *p;     // 指向父亲节点的指针
   Type x;                  // 节点中的元素
}
```

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* 所在集合的根节点使元素 *y* 的根节点的父指针指向元素 *x* 的根节点即可。

2）利用该数据结构合并两个集合的情况

图 (a) 表示由集合 {1,3,5,8}, {2,7,10}, {4,6}, {9} 所组成的森林图 (b) 表示由元素 1 所在集合、与元素 7 所在集合合并的情况



（a）　　　　（b）

3）该数据结构的缺点

树的高度可能很大，变成退化树，成为线性表

4）改进的数据结构

```
struct Tree_node {
    struuee Tree_node *p;          // 指向父亲节点的指针
    int rank;                      // 节点的秩
    Type x;                        // 存放在节点中的元素
};
typedef struct Tree_node NODE;
```

节点的秩等于以该节点作为子树的根时，该子树的高度。

5）在改进的数据结构下， **union(x,y)** 的操作

union(x,y)操作：

若 x 和 y 是当前森林中两棵不同树的根节点，

如果 **rank(x) > rank(y)** 把 x 作为 y 的父亲

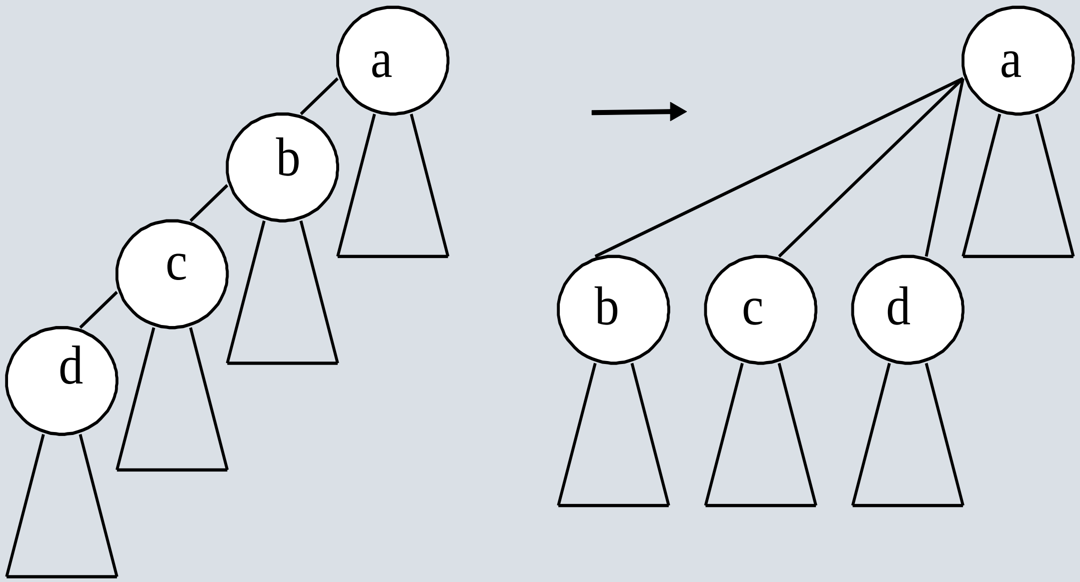
如果 **rank(x) < rank(y)** 把 y 作为 x 的父亲

如果 **rank(x) = rank(y)** 把 x 作为 y 的父亲，

rank(x) + 1

1. 路径压缩

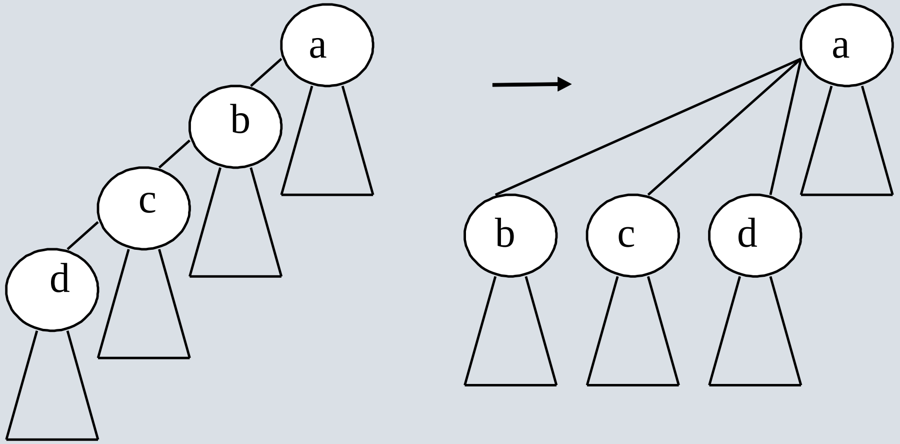
find(x) 操作时，找到 x 的根节点 y 之后，再沿着 x 到 y 的路径，改变路径上所有节点的父指针，使其直接指向 y。



2. 算法描述

*wp	*zp
	d
c	c
b	b
a	a

```
NODE *find(NODE *xp)
{
    NODE *wp, *yp = xp, *zp = xp;
    while (yp->p!=NULL) {           // 寻找 xp 所在集合的根
        yp = yp->p;                 // 节点 yp
    while (zp->p!= NULL) {          // 路径压缩
        wp = zp->p
        zp->p = yp;
        zp = wp;
    }
    return yp;
}
```



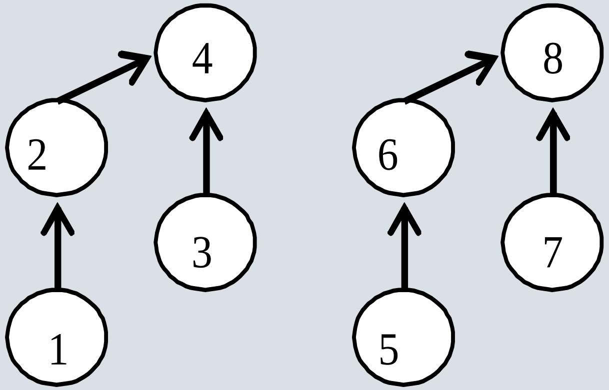
算法描述（续）

```
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;
}
```

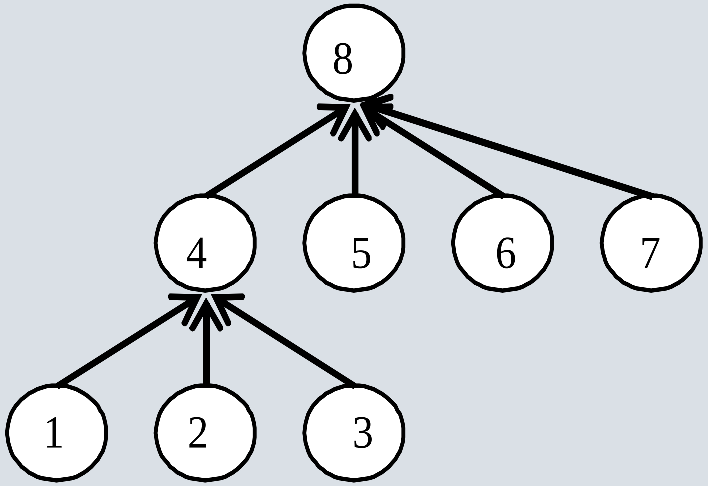
例：

图 (a) 表示集合 {1,2,3,4}, {5,6,7,8}

图 (b) 表示执行 union(1,5) 之后的结果



(a)



（b）

3. 复杂性分析

1）$x.p \rightarrow rank \geq x.rank + 1$

2）x.rank 的初始值为0，在一系列的union操作中递增，直到不再是树的根节点为止。一旦变为另一个节点的儿子，它的秩就不再改变 。

3）节点数为$n = 2^{x.rank}$的树，其高度至多为 $\log n = x.rank$

引理：若节点 x 的秩为 x.rank，则以 x 为根的树，其结点数至少为 $2^{x.rank}$

证明：用归纳法证明

(1) 开始时，x 是孤立顶点，x.rank = 0，节点数为 1，引理成立

(2) 设 x 和 y为根的树，秩分别是 x.rank 及 y.rank，节点数分别为 $2^{x.rank}$和 $2^{y.rank}$ ，用 union(x,y) 操作合并后，有三种情况：

① 若 x.rank < y.rank，在 union 操作之后，新的树以为 y 根节点，且 y 的秩不变，而树的节点数增加。新树的节点数至少为 $2^{y.rank}$。引理成立

节点数为 $n = 2^{x.rank}$ 的树，其高度至多为 logn = x.rank

② 若 x.rank > y.rank，同理可证

③ 若 x.rank = y.rank，两棵树的节点数至少都是

$$2^{x.rank} = 2^{y.rank}$$

在union操作之后，新树的节点数至少为。

$$2 \cdot 2^{y.rank} = 2^{y.rank+1} = 2^{x.rank+1}$$

若新树以 x 为根节点，则 x 的秩增1；

否则，y 的秩增1；在这两种情况下，引理都成立

4）find 操作的执行时间为 O(logn)

证明：如果 x 是树的根，x 的秩 x.rank 就是树的高度。

根据引理，节点数为 n，树的高度至多为 logn。

find 操作最多执行 logn 次判断根节点的操作、

以及 logn 次对非根节点进行的路径压缩操作

5）union操作的执行时间为 O(logn)

证明： union 操作除执行两次 find 操作外，

其余花费时间 O(1)

6）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 $O(m\log^* n) \approx O(m)$