## 第6章 堆排序

#### 练习 6.1-1



**问题分析**

在高度为*h*的堆中，第一层有1个元素，第二层有2个元素，第三层有4个元素，以此类推，除最后一层外，第*i*层有2*i*-1个元素。最后一层为第*h*+1层，最少有1个元素，最多有2*h*个元素。因此，高度为*h*的堆中，最少元素个数为



最多元素个数为



#### 练习 6.1-2



**问题分析**

利用练习6.1-1的结论，高度为*h*的堆中元素最少有2*h*个，最多有2*h*+1-1个，即2*h* ≤ *n* ≤ 2*h*+1-1。显然有。

#### 练习 6.1-3



**问题分析**

略。

#### 练习 6.1-4



**问题分析**

最小元素可以位于任意叶子结点上。

#### 练习 6.1-5



**问题分析**

如果数组已按照非降序排好序，那么它是一个最小堆。

#### 练习 6.1-6



**问题分析**

该数组不是最大堆，如下图所示。

#### 练习 6.1-7



**问题分析**

考虑最后一个结点，其下标为*n*，它的父结点*i* = 必然是最后一个非叶结点，而其后的结点都为叶结点。所以叶结点下标为, , … , *n*。

#### 练习 6.2-1



**问题分析**

#### 练习 6.2-2



**问题分析**

MIN\_HEAPIFY(*A*, *i*)

*l* = LEFT(*i*)

*r* = RIGHT(*i*)

**if** *l* <= *A*.*heap\_size* && *A*[*l*] < *A*[*i*]

*smallest* = *l*

**else**

*smallest* = *i*

**if** *r* <= *A*.*heap\_size* && *A*[*r*] < *A*[*smallest*]

*smallest* = *r*

**if** *smallest* != *i*

exchange *A*[*i*] with *A*[*smallest*]

MIN\_HEAPIFY(*A*, *smallest*)

MIN\_HEAPIFY的运行时间与MAX\_HEAPIFY一样，都为O(lg*n*)。

#### 练习 6.2-3



**问题分析**

当元素*A*[*i*]比其孩子都大时，*A*[*i*]满足最大堆的性质，调用MAX-HEAPIFY(*A*, *i*)不会有任何改变。

#### 练习 6.2-4



**问题分析**

当*i* > *A*.*heap\_size*/2时，*A*[*i*]一定为叶结点，故调用MAX-HEAPIFY(*A*, *i*)不会有任何改变。

#### 练习 6.2-5



**问题分析**

MAX\_HEAPIFY(*A*, *i*)

*l* = LEFT(*i*)

**while** *l* <= *A*.*heap\_size*

**if** *A*[*l*] > *A*[*i*]

*largest* = *l*

**else**

*largest* = *i*

*r* = RIGHT(*i*)

**if** *r* <= *A*.*heap\_size* && *A*[*r*] > *A*[*largest*]

*largest* = *r*

**if** *largest* != *i*

exchange *A*[*i*] with *A*[*largest*]

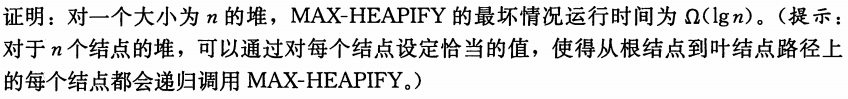
*i* = *largest*

*l* = LEFT(*i*)

**else**

**break**

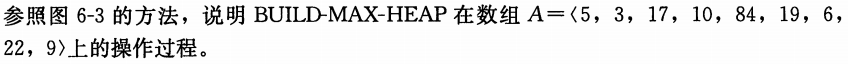
#### 练习 6.2-6



**问题分析**

利用练习6.1-2的结论，含有*n*个元素的堆的高度为。MAX-HEAPIFY的运行时间在最坏情况下与堆的高度成正比，因此它的最坏情况运行时间为Ω(lg*n*)。

#### 练习 6.3-1



**问题分析**

#### 练习 6.3-2



**问题分析**

因为调用MAX\_HEAPIFY(*A*, *i*)的先决条件是，结点*i*的子树必须满足最大堆条件。而节点*i*的子树中的结点的下标都比*i*要大，因此BUILD-MAX-HEAP中的循环控制变量*i*必须递减。而如果从1开始递增，节点1的子树还不满足最大堆条件。

#### 练习 6.3-3



**问题分析**

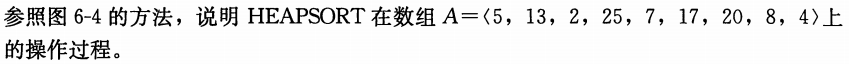
简略分析一下，不提供严格证明。

首先考虑叶结点，它们的高度*h* = 0，根据练习6.1-7的结论，含有*n*个元素的堆的叶子结点的个数为。

下面把原始堆*H*0中的叶结点去掉，剩下的元素仍然构成一个堆*H*1，并且*H*1中的叶结点就是*H*0中高度为1的结点。堆*H*1的大小为，故*H*1中的叶结点个数为。

以此类推，在一个大小为*n*的堆中，高度为*h*的结点至多有个。

#### 练习 6.4-1

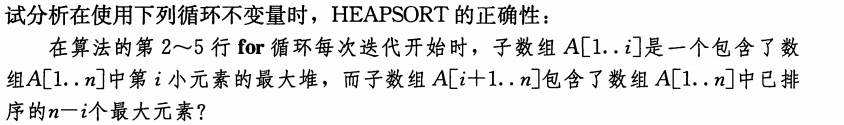


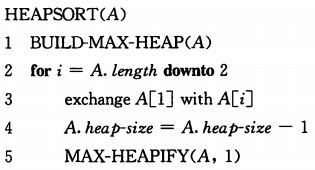
**问题分析**

先建堆，建堆过程省略，建好的堆如下图所示。

下面给出堆排序的过程。

#### 练习 6.4-2





**问题分析**

初始化：迭代开始时*i* = *n*，子数组*A*[*n*+1 .. *n*]为空，并且子数组*A*[1 .. *n*]包含了所有元素，循环不变式显然成立。

保持：假设在第*i*次迭代开始前，循环不变式成立，这意味着子数组*A*[1 .. *i*]是一个包含了最小的*i*个元素的最大堆，并且子数组*A*[*i*+1 .. *n*]包含了已排好序的最大的*n*−*i*个元素，于是有*A*[1 .. *i*] ≤ *A*[*i*+1 .. *n*]（这里的意思是*A*[1 .. *i*]中任一元素都小于或等于*A*[*i*+1 .. *n*]中的任一元素）。*A*[1]又是*A*[1 .. *i*]中的最大元素，于是又有*A*[2 .. *i*] ≤ *A*[1] ≤ *A*[*i*+1 .. *n*]。在第*i*次迭代中，将*A*[1]与*A*[*i*]交换。交换后，有*A*[1 .. *i*−1] ≤ *A*[*i*] ≤ *A*[*i*+1 .. *n*]。由于子数组*A*[*i*+1 .. *n*]本来已经有序，所以加入*A*[*i*]后，子数组*A*[*i* .. *n*]仍然是有序的，并且*A*[*i* .. *n*]包含了最大的*n*−(*i*−1)个元素。自然地，子数组*A*[1 .. *i*−1]包含了最小的*i*−1个元素，并且调用MAX\_HEAPIFY维护了*A*[1 .. *i*−1]的堆序性质。所以在进入第*i*−1迭代之前，循环不变式仍然成立。

终止：当*i* = 1时，循环终止。根据循环不变式，子数组*A*[2 .. *n*]包含已经排好序的最大的(*n*−1)个元素，并且*A*[1]是最小的元素。显然，整个数组*A*[1 .. *n*]包含了原数组中的所有元素，并且已经排好序。由此可以证明堆排序算法的正确。

#### 练习 6.4-3



**问题分析**

#### 练习 6.4-4



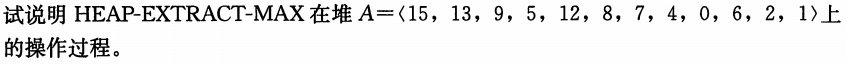
**问题分析**

#### 练习 6.4-5\*



**问题分析**

#### 练习 6.5-1



**问题分析**

#### 练习 6.5-2



**问题分析**

#### 练习 6.5-3



**问题分析**

HEAP\_MINIMUM(*A*)

**return** *A*[1]

HEAP\_EXTRACT\_MIN(*A*)

**if** *A*.*heap\_size* < 1

**error** “heap underflow”

*min* = *A*[1]

*A*[1] = *A*[*A*.*heap\_size*]

*A*.*heap\_size*--

MIN\_HEAPIFY(*A*, 1)

**return** *min*

HEAP\_DECREASE\_KEY(*A*, *i*, *key*)

**if** *key* > *A*[*i*]

**error** “the new key is larger than the current key”

*A*[*i*] = *key*

**while** *i* > 1 && *A*[PARENT(*i*)] > *A*[*i*]

exchange *A*[*i*] with *A*[PARENT(*i*)]

*i* = PARENT(*i*)

MIN\_HEAP\_INSERT(*A*, *key*)

*A.heap\_size* += 1

*A*[*A.heap\_size*] = ∞

HEAP-DECREASE-KEY(*A*, *A.heap\_size*, *key*)

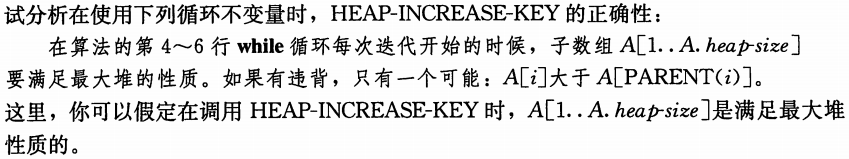
#### 练习 6.5-4

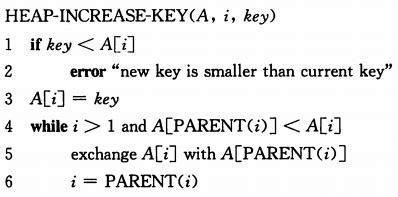


**问题分析**

这是为了可以调用HEAP\_INCREASE\_KEY来实现MAX\_HEAP\_INSERT。先把关键字设为−∞，可以保证在调用HEAP\_INCREASE\_KEY时，不会出现增加的值小于原始值的情况。

#### 练习 6.5-5





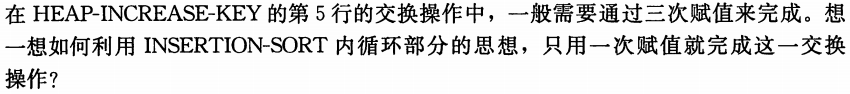
**问题分析**

初始化：初始时，在*A*[*i*]的值还未发生改变时，数组*A*[1 .. *A*.*heap\_size*]原本就满足最大堆性质。如果*A*[*i*]的值增加，*A*[*i*]仍然大于或等于它的子结点，以*A*[*i*]为根的子树仍然满足最大堆性质，而*A*[*i*]有可能会大于它的父结点。因此，初始时循环不变式成立。

保持：假设某次迭代之前，循环不变式成立。这意味着*A*[*i*] > *A*[PARENT(*i*)]，并且除此之外数组满足最大堆性质，这说明*A*[PARENT(*i*)]大于*A*[*i*]的子结点。进入迭代本次过程，由于*A*[PARENT(*i*)]大于*A*[*i*]的子结点，所以交换*A*[*i*]和*A*[PARENT(*i*)]后，由于*A*[*i*]也大于它的子结点，所以以*A*[*i*]为根的子树满足最大堆性质。进入下一次迭代之前，让*i* = PARENT(*i*)，这时*A*[*i*]有可能大于它的父结点。因此，下一次迭代之前，循环不变式仍然成立。

终止：当*i* = 1或*A*[PARENT(*i*)] ≥ *A*[*i*]时，循环终止。如果*i* = 1，根据循环不变式，整个数组*A*[1 .. *A*.*heap\_size*]满足最大堆性质，除了*A*[1]有可能大于*A*[PARENT(1)]之外，而*A*[1]已经是根结点，所以整个数组已经完全满足最大堆性质。如果*A*[PARENT(*i*)] ≥ *A*[*i*]，这时整个数组也已经完全满足最大堆性质。所以HEAP\_INCREASE\_KEY算法是正确的。

#### 练习 6.5-6



**问题分析**

HEAP-INCREASE-KEY(*A*, *i*, *key*)

**if** *key* < *A*[*i*]

error “the new key is smaller than the current key”

**while** *i* > 1 && *A*[PARENT(*i*)] < *key*

*A*[*i*] = *A*[PARENT(*i*)]

*i* = PARENT(*i*)

*A*[*i*] = *key*

#### 练习 6.5-7



**问题分析**

用最小堆来实现先进先出队列。需要记录每个元素入队的时间，入队时间早的先出队，入队时间晚的后出队。

用最大堆来实现栈。也需要记录每个元素的入栈时间，入栈时间晚的先栈，入栈时间早的后出栈。

这时所说的时间并非一定是实际的时间，例如可以用一个计数器来代器。

#### 练习 6.5-8



**问题分析**

将要删除的元素*A*[*i*]和堆的最后一个元素*A*[*A*.*heap\_size*]交换，再将*A*[*A*.*heap\_size*]删除。如此处理之后，堆序性质可能被破坏。根据*A*[*i*]和*A*[*A*.*heap\_size*]的大小关系分为3种情况：

(1) *A*[*A*.*heap\_size*] < *A*[*i*]

此时，交换*A*[*i*]和*A*[*A*.*heap\_size*]后，以*A*[*i*]为根的子树有可能不满足堆序性质，调用MAX-HEAPIFY即可。

(2) *A*[*A*.*heap\_size*] > *A*[*i*]

此时，交换*A*[*i*]和*A*[*A*.*heap\_size*]后，*A*[*i*]有可能比其父结点*A*[PARENT(*i*)]要大。可以采用类似HEAP-INCREASE-KEY的方式来将*A*[*i*]向上移动到正确的位置。或者，采用更直接的方法，不用先交换*A*[*i*]和*A*[*A*.*heap\_size*]，直接调用HEAP\_INCREASE\_KEY(*A*, *i*, *A*[*A*.*heap\_size*])。

(3) *A*[*A*.*heap\_size*] = *A*[*i*]

此时，交换交换*A*[*i*]和*A*[*A*.*heap\_size*]后，堆序性质仍然满足。

无论是情况(1)调用MAX-HEAPIFY，还是情况(2)调用HEAP-INCREASE-KEY(*A*, *i*, *A*[*A*.*heap\_size*])，都要花费O(lg*n*)时间。

**伪代码**

HEAP-DELETE(*A*, *i*)

**if** *A*[*A*.*heap\_size*] <= *A*[*i*]

*A*[*i*] = *A*[*A*.*heap\_size*]

*A.heap\_size*--

MAX-HEAPIFY(*A*, *i*)

**else**

HEAP-INCREASE-KEY(*A*, *i*, *A*[*A*.*heap\_size*])

*A.heap\_size*--

#### 练习 6.5-9



**问题分析**

(1) 从*k*个链表中分别取出第1个元素，构成一个大小为*k*的最小堆*H*。调用HEAP-EXTRACT-MIN取出*H*中的最小元素作为合并链表的第1个元素，并记住取出的元素来自哪个链表，从该链表中取出第2个元素插入到*H*中。

(2) 现在进入迭代。每次迭代调用HEAP-EXTRACT-MIN取出*H*中的最小元素，插入到合并链表的尾端；记住取出的元素来自哪个链表，从该链表取出首端元素插入到*H*中。如果在插入元素到*H*的过程中，相应的那个链表没有元素，则不执行插入，*H*的大小减1。执行迭代直到*H*中没有元素为止。

MERGE-LISTS(*L*, *k*)

create a min-heap *H* with the size of *k*

**for** *i* = 1 🡪 *k*

*H*[*i*] = *L*[*i*].*head*

*H*[*i*].*list\_index* = *i*

BUILD-MIN-HEAP(*H*)

create an empty list *LM*

**while** *H*.*heap\_size* > 0

*i* = *H*[1].*list\_index*

PUSH\_BACK(*LM*, HEAP-EXTRACT-MIN(*H*))

**if** *L*[*i*] is not empty

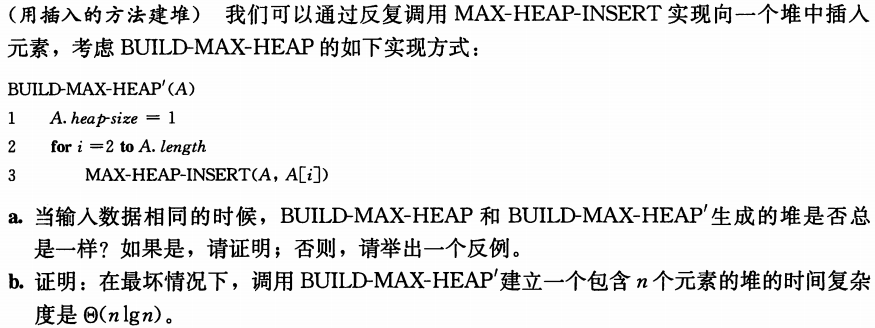
MIN-HEAP-INSERT(*H*, *L*[*i*].*head*)

**return** *LM*

**时间分析**

该算法首先要建一个大小为*k*的堆，这一步所花时间为O(*k*)。然后进行*n*次迭代，每次迭代对大小最多为*k*的堆执行一次HEAP-EXTRACT-MIN和一次MIN-HEAP-INSERT，二者各花费O(lg*k*)时间。因此*n*次迭代总共花费O(*n*lg*k*)时间。

#### 思考题 6-1



**a-问题分析**

两种方法生成的堆有可能不一样。

例如，取*A* = {1, 2, 3, 4}。采用BUILD-MAX-HEAP生成的堆为

而采用BUILD-MAX-HEAP*’*生成的堆为

**b-问题分析**

最坏情况为*A*中的元素按升序排列。此时，每插入一个元素，都要将该元素移动到根结点，移动的步数为当前堆的深度。因此，总时间正比于插入每个元素所移动的步数之和，即堆中所有元素的深度总和。

简化分析，考虑建立的堆为完全二叉树，它的高度为。逐层分析每个节点的深度。

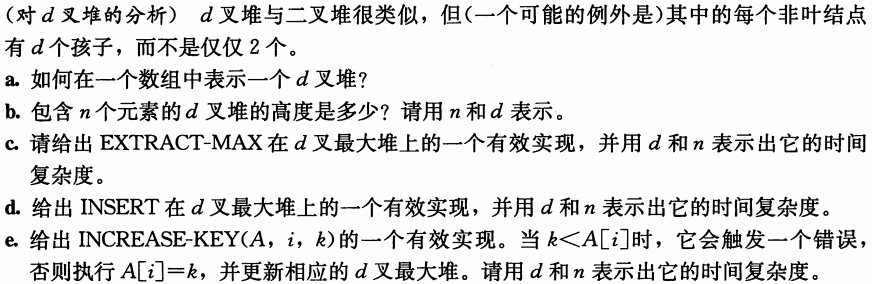
|  |  |
| --- | --- |
| 0 | 0 |
| 1 | 1∙21 |
| 2 | 2∙22 |
| 3 | 3∙23 |
| … … | … … |
|  |  |

因此，总时间为



因此，最坏情况下BUILD-MAX-HEAP*’*的时间复杂度为Θ(*n*lg*n*)。

#### 思考题 6-2



**a-问题分析**

与二叉堆类似，*d*叉堆的元素按照从上到下，从左到右的顺序在数组中排列。下面推导数组中每个元素父结点下标和子结点下标。

对于*d*叉堆中第*i*层的第*j*个元素(*i*和*j*都从1开始)，它在数组中的下标为



它的父结点为第*i*−1层的第个元素，该结点在数组中的下标为



*d*叉堆中第*i*层的第*j*个元素，其第一个子结点为第(*i*+1)层的第(*d*(*j*−1)+1)个结点，该结点在数组中的下标为



由于每个结点都有*d*个儿子，所以第*i*层的第*j*个元素的d个儿子的下标分别为*d*(*p*(*i*, *j*)-1)+2, *d*(*p*(*i*, *j*)-1)+3, *d*(*p*(*i*, *j*)-1)+4, …, *d*(*p*(*i*, *j*)-1)+*d*+1。

综上所述，*d*叉堆中数组下标为*i*的元素，其父结点的数组下标为，其第*j*个子结点的数组下标为。

**b-问题分析**

高为*h*的*d*叉堆，最少元素个数和最多元素个数分别为





所以，高为*h*的*d*叉堆的元素个数*n*(*h*)满足



即



显然有，*n*个元素的*d*叉堆，其高度。

**c-问题分析**

D\_MAX\_HEAPIFY(*A*, *d*, *i*)

*largest* = *i*

*child* = CHILD(*d*, *i*, 1)

**for** *j* = *child* 🡪 *child*+*d-*1

**if** *j* > *A*.*heap\_size*

**break**

**elseif** *A*[*j*] > *A*[*largest*]

*largest* = *j*

**if** *largest* != *i*

exchange *A*[*i*] with *A*[*largest*]

D\_MAX\_HEAPIFY(*A*, *d*, *largest*)

D\_HEAP\_EXTRACT\_MAX(*A*, *d*)

**if** *A*.*heap\_size* < 1

**error** “heap underflow”

*max* = *A*[1]

*A*[1] = *A*[*A*.*heap\_size*]

*A*.*heap\_size*--

D\_MAX\_HEAPIFY(*A*, *d*, 1)

**return** *max*

该算法的运行时间主要取决于D\_MAX\_HEAPIFY。在D\_MAX\_HEAPIFY中，每次迭代需要进行*d*次比较，迭代次数与堆的高度成正比，因此时间复杂度为O(*d* log*d n*)。

**d-问题分析**

D\_MAX\_HEAP\_INSERT(*A*, *d*, *key*)

*A*.*heap\_size*++

*A*[*A*.*heap\_size*] = -∞

D\_HEAP\_INCREASE\_KEY(*A*, *d*, *A*.*heap\_size*, *key*)

该算法的运行时间与堆的高度成正比，时间复杂度为O(log*d n*)。

**e-问题分析**

D\_MAX \_HEAP\_INCREASE\_KEY(*A*, *d*, *i*, *key*)

**if** *key* < *A*[*i*]

error “the new key is smaller than the current key”

*A*[*i*] = *key*

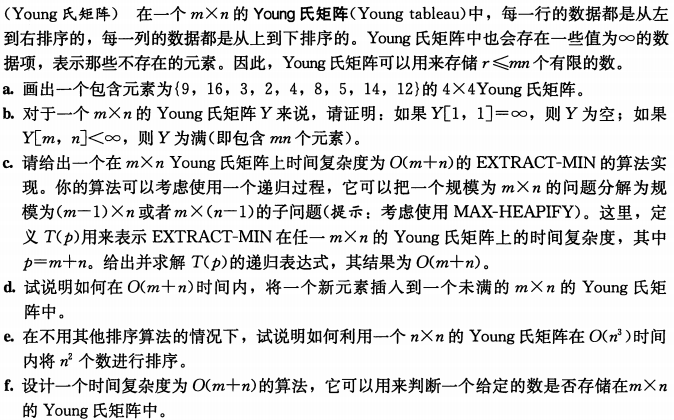
**while** *i* > 1 && *A*[PARENT(*d*, *i*)] < *A*[*i*]

exchange *A*[*i*] with *A*[PARENT(*d*, *i*)]

*i* = PARENT(*d*, *i*)

该算法的运行时间与堆的高度成正比，时间复杂度为O(log*d n*)。

#### 思考题 6-3



**a-问题分析**

2 3 8 ∞

4 5 12 ∞

9 14 16 ∞

∞ ∞ ∞ ∞

**b-问题分析**

如果*Y*[1, 1] = ∞，说明第一行全为∞，这也说明每一列都全为∞，故所有元素都为∞，即*Y*为空。

如果*Y*[*m*, *n*] < ∞，说明最后一行都不为∞，这也说明每一列都不为∞，故所有元素都不为∞，即*Y*为满。

**c-问题分析**

一个Young氏矩阵中，左上角元素一定是最小的，而与左上解元素最接近的两个元素（右边和下边的元素）之一肯定是次小的。EXTRACT-MIN算法首先把左上角元素提取出来，然后从与左上解元素最接近的两个元素中取较小者作为新的左上角元素。这样，在左上角元素的右边或下边会空出一个元素，可以采用同样的方法从这个空元素最接近的两个元素中取较小者来填充。这样可以递归解决EXTRACT-MIN问题。

在EXTRACT-MIN(*Y*, *m*, *n*, *i*, *j*)中，*Y*代表Young氏矩阵，*m*和*n*分别代表矩阵的行数和列数，*i*和*j*分别代表EXTRACT-MIN操作的起点。对一个矩阵调用EXTRACT-MIN(*Y*, *m*, *n*, 0, 0)即可提取最小元素。

EXTRACT\_MIN(*Y*, *m*, *n*, *i*, *j*)

**if** *i* > *m* || *j* > *n* || *Y*[*i*, *j*] == ∞

**return** ∞

*min* = *Y*[*i*, j]

**if** *i* == *m*

**if** *j* == *n*

*Y*[*i*, j] = ∞

**return** *min*

**else**

*smallest* == RIGHT

**else**

**if** *j* == *n*

*smallest* == BOTTOM

**else**

**if** *Y*[*i*, *j*+1] < *Y*[*i*+1, *j*]

*smallest* = BOTTOM

else

*smallest* = RIGHT

**if** *smallest* == RIGHT

*Y*[*i*, *j*] = EXTRACT-MIN(*Y*, *m*, *n*, *i*+1, *j*)

**else**

*Y*[*i*, *j*] = EXTRACT-MIN(*Y*, *m*, *n*, *i*, *j*+1)

**return** *min*

}

EXTRACT-MIN(*Y*, *m*, *n*, *i*, *j*)最多在矩阵的行方向和列方向各遍历一次，因此其时间复杂度为O(*m*+*n*)。

**d-问题分析**

如果一个Young氏矩阵未满，那么它的右下角元素*Y*[*m*, *n*]一定为空。先将要插入的元素置于矩阵的右下角。将右下角元素*Y*[*m*, *n*]与最接近的两个元素*Y*[*m*, *n*-1]和*Y*[*m*-1, *n*]（上边和左边的元素）进行比较：如果*Y*[*m*, *n*] ≥ *Y*[*m*, *n*-1]并且*Y*[*m*, *n*] ≥ *Y*[*m*-1, *n*]，那说明插入的元素*Y*[*m*, *n*]不破坏Young氏矩阵的性质，直接将插入的元素置于*Y*[*m*, *n*]即可；否则，插入的元素*Y*[*m*, *n*]会破坏Young氏矩阵的性质，此时取*Y*[*m*, *n*-1]和*Y*[*m*-1, *n*]之中的较大者，与*Y*[*m*, *n*]交换，*Y*[*m*, *n*]移动到新位置，再递归上面的过程即可。

在INSERT (*Y*, *m*, *n*, *i*, *j*, *x*)中，*Y*代表Young氏矩阵，*m*和*n*分别代表矩阵的行数和列数，*i*和*j*分别代表INSERT操作的起点。对一个矩阵调用INSERT (*Y*, *m*, *n*, *m*, *n*, *x*)即可提取插入一个元素。

INSERT(*Y*, *m*, *n*, *i*, *j*, *x*)

{

**if** *i* < 1 || *j* < 1 || *i* > *m* || *j* > *n* || *Y*[*i*, *j*] != ∞

**return**

*largest* = *x*

**if** *j* > 1 && *Y*[*i*, *j*-1] > *largest*

*largest* = *Y*[*i*, *j*-1]

**if** *i* > 1 && *Y*[*i*-1, *j*] > *largest*

*largest* = *Y*[*i*-1, *j*]

**if** *largest* == *x*

*Y*[*i*, *j*] = *x*

**elseif** *largest* == *Y*[*i*, *j*-1]

*Y*[*i*, *j*] = *Y*[*i*, *j*-1]

*Y*[*i*, *j*-1] = ∞

INSERT(*Y*, *m*, *n*, *i*, *j*-1, *x*)

else *largest* == *Y*[*i*-1, *j*]

*Y*[*i*, *j*] = *Y*[*i*-1, *j*]

*Y*[*i*-1, *j*] = ∞

INSERT(*Y*, *m*, *n*, *i*-1, *j*, *x*)

}

与EXTRACT-MIN类似，时间复杂度为O(*m*+*n*)。

**e-问题分析**

先调用INSERT建立一个Young氏矩阵，每插入一个元素花费O(*n*+*n*)=O(*n*)时间，一共有*n*2个元素，因此这一步所花费的时间为O(*n*3)。再调用EXTRACT-MIN依次提取出所有元素即可完成排序，这一步所花费的时间也为O(*n*3)。

**f-问题分析**

假设要寻找的元素为*x*，从左下角元素*Y*[*m*, 1]开始搜索。如果*x* > *Y*[*m*, 1]，说明*x*不可能出现在*Y*[*m*, 1]的上方，即第一列，只可能出现在第一列的右边，所以以*Y*[*m*, 2]作为左下角元素重新搜索。如果*x* < *Y*[*m*, 1]，说明*x*不可能出现在*Y*[*m*, 1]的右方，即最后一行，只可能出现在最后一行的上边，所以以*Y*[*m*-1, 1]作为左下角元素重新搜索。

SEARCH(*Y*, *m*, *n*, *x*)

**if** *m* < 1 || *n* < 1

**return** “*x* is not found”

*i* = *m*

*j* = 1

**while** *i* >= 1 && *j* <= *n*

**if**  *x* > *Y*[*i*, *j*]

*j*++

**elseif** *x* < *Y*[*i*, *j*]

*i*--

**else**

**return** *i* and *j*

**return** “*x* is not found”