第 19 章

Introduction to Algorithms, Third Edition

斐波那契堆

斐波那契堆数据结构有两种用途。第一种，它支持一系列操作，这些操作构成了所谓的“可合并堆”。第二种，斐波那契堆的一些操作可以在常数摊还时间内完成，这使得这种数据结构非常适合于需要频繁调用这些操作的应用。

可合并堆

可合并堆(mergeable heap)是支持以下 5 种操作的一种数据结构，其中每个元素都有一个关键字：

MAKE-HEAP()：创建和返回一个新的不含任何元素的堆。

INSERT(H，x)：将一个已填入关键字的元素 x 插入堆 H 中。

MINIMUM(H)：返回一个指向堆 H 中具有最小关键字元素的指针。

EXTRACT-MIN(H)：从堆 H 中删除最小关键字的元素，并返回一个指向该元素的指针。

UNION(H_1,H_2)：创建并返回一个包含堆 H_1 和堆 H_2 中所有元素的新堆。堆 H_1 和堆 H_2 由这一操作“销毁”。

除了以上可合并堆的操作外，斐波那契堆还支持以下两种操作：

DECREASE-KEY(H，x，k)：将堆 H 中元素 x 的关键字赋予新值 k。假定新值 k 不大于当前的关键字。⊖

505 DELETE(H，x)：从堆 H 中删除元素 x。

如图 19-1 所示，如果没有 UNION 操作，如同堆排序(第 6 章)中使用的普通二项堆，其操作性能相当好。除了 UNION 操作外，二项堆的其他操作均可在最坏情况时间为 $O(\lg n)$ 下完成。但是，如果需要支持 UNION 操作，则二项堆的性能就很差。通过把两个分别包含要被合并二项堆的数组进行链接，然后运行 BUILD-MIN-HEAP(参考 6.3 节)的方式来实现 UNION 操作，其最坏情形下需要 $\Theta(n)$时间。

操　作	二项堆（最坏情形）	斐波那契堆（摊还）
MAKE-HEAP	$\Theta(1)$	$\Theta(1)$
INSERT	$\Theta(\lg n)$	$\Theta(1)$
MINIMUM	$\Theta(1)$	$\Theta(1)$
EXTRACT-MIN	$\Theta(\lg n)$	$O(\lg n)$
UNION	$\Theta(n)$	$\Theta(1)$
DECREASE-KEY	$\Theta(\lg n)$	$\Theta(1)$
DELETE	$\Theta(\lg n)$	$O(\lg n)$

图 19-1　可合并堆的两种实现方式下各操作的运行时间。在操作时堆中的项数用 n 表示

另一方面，斐波那契堆对于操作 INSERT、UNION 和 DECREASE-KEY，比起二项堆有更好的渐近时间界，而对于剩下的几种操作，它们有相同的渐近运行时间。然而，注意，图 19-1中斐波那契堆的运行时间是摊还时间界，而不是每个操作的最坏情形时间界。UNION 操作在斐波那契堆中仅仅需要常数摊还时间，这比二项堆的最坏情形下的线性时间要好得多(当然，假定为一个摊还时间界)。

理论上的斐波那契堆与实际中的斐波那契堆

从理论角度来看，当 EXTRACT-MIN 和 DELETE 数目相比于其他操作小得多的时候，斐波那契堆尤其适用。这种情形出现在许多应用中。例如，一些图问题算法可能每条边调用一次 DECREASE-KEY。对于有很多边的稠密图，每次调用 DECREASE-KEY 需要 $\Theta(1)$摊还时间，相比起二项堆最坏情况时间 $\Theta(\lg n)$，其积累起来是个很大的改进。一些问题(如计算最小生成树

⊖ 正如在第五部分的导言中提到的，我们默认的可合并堆是可合并最小堆，因此，使用操作 MINIMUM、EXTRACT-MIN 和 DECREASE-KEY。同样，我们可以定义一个**可合并最大堆**(mergeable max-heap)，具有操作 MAXIMUM、EXTRACT-MAX 和 INCREASE-KEY。

(第 23 章)和寻找单源最短路径(第 24 章))的快速算法必不可少地要用到斐波那契堆。 506

然而从实际角度来看，除了某些需要管理大量数据的应用外，对于大多数应用，斐波那契堆的常数因子和编程复杂性使得它比起普通二项(或 k 项)堆并不那么适用。因此，对斐波那契堆的研究主要出于理论兴趣。如果能开发出一个简单得多的数据结构，而且它的摊还时间界与斐波那契堆相同，那么它将非常实用。

二项堆和斐波那契堆对于 SEARCH 操作的支持均比较低效；可能需要花费一段时间才能找到具有给定关键字的元素。为此，涉及给定元素的操作(如 DECREASE-KEY 和 DELETE)均需要一个指针指向这个元素，并且指针作为输入的一部分。正如 6.5 节对优先级队列的讨论中所述，当在应用中使用一个可合并堆时，通常在可合并堆的每个元素中存储一个句柄指向相关应用对象，同样在每个应用对象中也存储一个句柄指向可合并堆中相关元素。这些句柄的确切作用依赖于应用和它的实现。

如同所看到的一些其他数据结构，斐波那契堆也是基于有根树的。我们把每一个元素表示成树中的一个结点，每个结点具有一个 *key* 属性。在这一章的剩下部分，将使用“结点”来代替“元素”。我们也将忽略结点插入之前和删除之后的内存分配和释放问题，而不是假定调用堆操作的代码来处理这些细节问题。

19.1 节将定义斐波那契堆，讨论如何表示它，并给出用于分析摊还时间的势函数。19.2 节展示怎样实现可合并堆操作和如何得到图 19-1 中所述的摊还时间界。剩下的两个操作 DECREASE-KEY 和 DELETE 是 19.3 节的重点。最后，19.4 节完成理论分析的主要环节，并解释这个数据结构名字的由来。

19.1　斐波那契堆结构

一个斐波那契堆是一系列具有**最小堆序**(min-heap ordered)的有根树的集合。也就是说，每棵树均遵循**最小堆性质**(min-heap property)：每个结点的关键字大于或等于它的父结点的关键字。图 19-2(a)是一个斐波那契堆的例子。

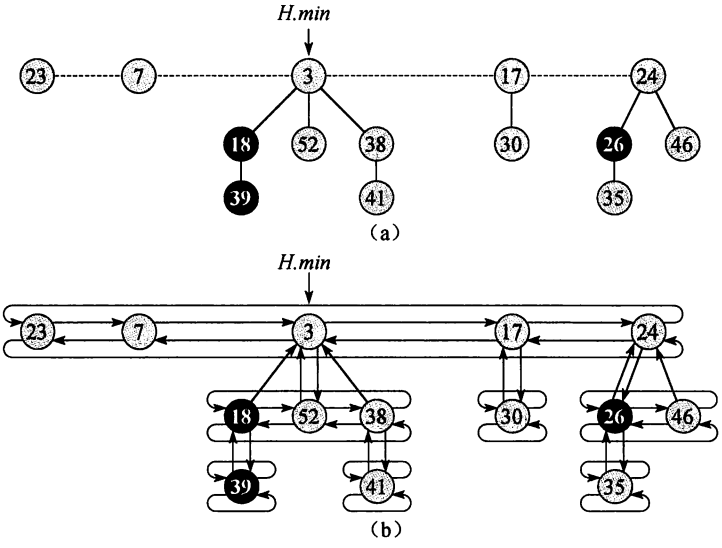

图 19-2　(a)一个包含 5 棵最小堆序树和 14 个结点的斐波那契堆。虚线标出了根链表。堆中最小的结点是包含关键字 3 的结点。黑色的结点是被标记的。这个斐波那契堆的势是5+2×3=11。(b)一个更加完整的表示，显示出了指针 p(向上箭头)、*child*(向下箭头)、*left* 和 *right*(横向箭头)。本章剩下的图省略了这些细节，因为该图中显示的所有信息均可从(a)图中推断出来

如图 19-2(b)所示，每个结点 x 包含一个指向它父结点的指针 $x.p$ 和一个指向它的某一个孩子的指针 $x.child$。x 的所有孩子被链接成一个环形的双向链表，称为 x 的**孩子链表**(child list)。孩子链表中的每个孩子 y 均有指针 $y.left$ 和 $y.right$，分别指向 y 的左兄弟和右兄弟。如果 y 是仅有的一个孩子，则 $y.left = y.right = y$。孩子链表中各兄弟出现的次序是任意的。

环形双向链表(参考 10.2 节)应用在斐波那契堆中有两个优点。第一，可以在 $O(1)$ 时间内从一个环形双向链表的任何位置插入一个结点或删除一个结点。第二，给定两个这种链表，可以用 $O(1)$ 时间把它们链接(或把它们"捻接"在一起)成一个环形双向链表。在斐波那契堆操作的描述中，我们将非正式地提到这些操作，实现细节留给读者去补充。

每个结点有另外两个属性。把结点 x 的孩子链表中的孩子数目储存在 $x.degree$。布尔值属性 $x.mark$ 指示结点 x 自从上一次成为另一个结点的孩子后，是否失去过孩子。新产生的结点是未被标记的，并且当结点 x 成为另一个结点的孩子时，它便成为未被标记结点。直到 19.3 节的 DECREASE-KEY 操作，我们才把所有的 $mark$ 属性值设为 FALSE。

507 ~ 508

通过指针 $H.min$ 来访问一个给定的斐波那契堆 H，该指针指向具有最小关键字的树的根结点，我们把这个结点称为斐波那契堆的**最小结点**(minimum node)。如果不止一个根结点具有最小关键字，那么这些根结点中的任何一个都有可能成为最小结点。如果一个斐波那契堆 H 是空的，那么 $H.min$ 为 NIL。

在斐波那契堆中，所有树的根都用其 $left$ 和 $right$ 指针链成一个环形的双链表，该双链表称为斐波那契堆的**根链表**(root list)。因此，指针 $H.min$ 指向根链表中关键字最小的那个结点。根链表中的树次序可以任意。

我们还要用到斐波那契堆 H 的另一个属性：$H.n$，表示 H 中当前的结点数目。

势函数

正如上面提到的，将使用 17.3 节中的势方法来分析斐波那契堆操作的性能。对于一个给定的斐波那契堆 H，用 $t(H)$ 来表示 H 中根链表中树的数目，用 $m(H)$ 来表示 H 中已标记的结点数目。然后，定义斐波那契堆 H 的势函数 $\Phi(H)$ 如下：

$$\Phi(H) = t(H) + 2m(H) \tag{19.1}$$

(19.3 节会给出这样定义的一些直观解释。)例如，图 19-2 中所示的斐波那契堆的势为 $5+2\times3=11$。一系列斐波那契堆的势等于各个斐波那契堆势的和。假定势的一个单位可以支付常数数目的工作，该常数要足够大，能够支付我们可能遇到的任何特定的常数时间的工作。

假定斐波那契堆应用开始时，都没有堆。因此，势初始值为 0，而且根据公式(19.1)，势在随后的任何时间内均不为负。依据公式(17.3)，对于某一操作序列来说，总的摊还代价的上界就是其总的实际代价的上界。

最大度数

在本章剩下几节中，对于摊还分析均假定，在一个 n 个结点的斐波那契堆中任何结点的最大度数都有上界 $D(n)$。在此我们不证明这一假定，但是如果仅仅是支持可合并堆的操作，那么 $D(n) \leqslant \lfloor \lg n \rfloor$(思考题 19-2d 要求读者证明这一性质。)在 19.3 节和 19.4 节中，当支持 DECREASE-KEY 和 DELETE 操作时，也要求 $D(n) = O(\lg n)$。

509

19.2 可合并堆操作

斐波那契堆上的一些可合并堆操作要尽可能长地延后执行。不同的操作可以进行性能平衡。例如，用将一个结点加入根链表的方式来插入一个结点，这样仅需耗费常数时间。如果从空的斐波那契堆开始，插入 k 个结点，斐波那契堆将由一个正好包含 k 个结点的根链表组成。如果在斐波那契堆 H 上执行一个 EXTRACT-MIN 操作，在移除 $H.min$ 指向的结点后，将不得不遍历根链表中剩下的 $k-1$ 个结点来找出新的最小结点，这里便存在性能平衡问题。只要我们在执行 EXTRACT-

MIN 操作中遍历整个根链表，并且把结点合并到最小堆序树中以减小根链表的规模。下面将看到，不论根链表在执行 EXTRACT-MIN 操作之前是什么样子，执行完该操作之后，根链表中的每个结点要求有一个与根链表中其他结点均不同的度数，这使得根链表的规模最大是 $D(n)+1$。

创建一个新的斐波那契堆

创建一个空的斐波那契堆，MAKE-FIB-HEAP 过程分配并返回一个斐波那契堆对象 H，其中 $H.n=0$ 和 $H.min=$NIL，H 中不存在树。因为 $t(H)=0$ 和 $m(H)=0$，空斐波那契堆的势为 $\Phi(H)=0$。因此，MAKE-FIB-HEAP 的摊还代价等于它的实际代价 $O(1)$。

插入一个结点

下面的过程将结点 x 插入斐波那契堆 H 中，假定该结点已经被分配，$x.key$ 已经被赋值。

```
FIB-HEAP-INSERT(H,x)
 1  x.degree = 0
 2  x.p = NIL
 3  x.child = NIL
 4  x.mark = FALSE
 5  if H.min == NIL
 6       create a root list for H containing just x
 7       H.min = x
 8  else insert x into H's root list
 9       if x.key < H.min.key
10            H.min = x
11  H.n = H.n + 1
```

510

第 1～4 行初始化结点 x 的一些属性。第 5 行测试斐波那契堆 H 是否为空。如果为空，那么第 6～7 行使得 x 成为 H 的根链表中唯一的结点，并将 $H.min$ 指向 x；否则，第 8～10 行将结点 x 插入 H 的根链表中，如果有必要，就更新 $H.min$。最后，第 11 行 $H.n$ 增 1 来反映新结点的加入。图 19-3 展示了一个具有关键字 21 的结点插入图 19-2 所示的斐波那契堆中。

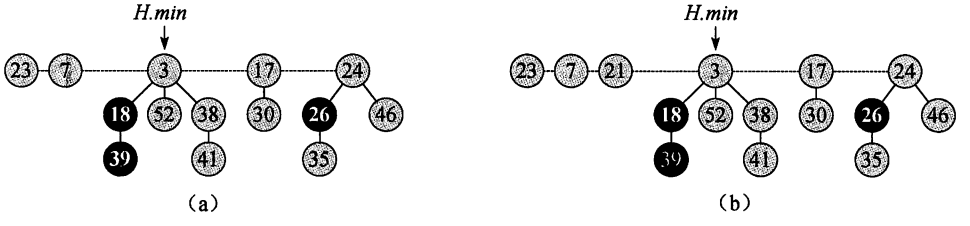

图 19-3　将一个结点插入斐波那契堆。(a)斐波那契堆 H。(b)插入关键字为 21 的结点后的斐波那契堆 H。该结点自成一棵最小堆序树，然后被插入根链表中，成为根的左兄弟

为了确定 FIB-HEAP-INSERT 的摊还代价，设 H 是输入的斐波那契堆，H' 是结果斐波那契堆。那么 $t(H')=t(H)+1$ 和 $m(H')=m(H)$，并且势的增加量为：

$$((t(H)+1)+2m(H))-(t(H)+2m(H))=1$$

由于实际代价为 $O(1)$，因此摊还代价为 $O(1)+1=O(1)$。

寻找最小结点

斐波那契堆的最小结点可以通过指针 $H.min$ 得到。因此，可以在 $O(1)$ 的实际代价内找到最小结点。由于 H 的势没有发生变化，因此该操作的摊还代价等于它的实际代价 $O(1)$。

两个斐波那契堆的合并

下面的过程合并斐波那契堆 H_1 和 H_2，并在该过程中销毁 H_1 和 H_2。它简单地将 H_1 和 H_2 的根链表链接，然后确定新的最小结点。之后，表示 H_1 和 H_2 的对象将不再使用。

511

```
FIB-HEAP-UNION(H1, H2)
1  H = MAKE-FIB-HEAP()
2  H.min = H1.min
3  concatenate the root list of H2 with the root list of H
4  if (H1.min = = NIL) or (H2.min ≠NIL and H2.min.key < H1.min.key)
5       H.min = H2.min
6  H.n = H1.n + H2.n
7  return H
```

第1～3行将 H_1 和 H_2 的根链表链接成为 H 的新根链表。第2、4、5行设定 H 的最小结点，第6行将 $H.n$ 设为所有结点的个数。第7行返回作为结果的斐波那契堆 H。与 FIB-HEAP-INSERT 过程相同，所有的根结点仍为根结点。

势函数的变化为：

$$\begin{aligned}\Phi(H)-(\Phi(H_1)+\Phi(H_2)) &= (t(H)+2m(H))-((t(H_1)+2m(H_1))+(t(H_2)+2m(H_2))) \\ &= 0\end{aligned}$$

因为 $t(H)=t(H_1)+t(H_2)$ 和 $m(H)=m(H_1)+m(H_2)$，所以 FIB-HEAP-UNION 的摊还代价等于它的实际代价 $O(1)$。

抽取最小结点

抽取最小结点的过程是本节所介绍的操作中最为复杂的一个。这里还要介绍前面提到的在根链表中合并树的延后工作。下面的伪代码是抽取最小结点的。为了简便该代码，假定当一个结点从链表中移除后，留在链表中的指针要被更新，但是抽取出的结点中的指针并不改变。该代码
512 还调用一个辅助过程 CONSOLIDATE，稍后将介绍。

```
FIB-HEAP-EXTRACT-MIN(H)
 1  z= H.min
 2  if z ≠NIL
 3       for each child x of z
 4            add x to the root list of H
 5            x.p = NIL
 6       remove z from the root list of H
 7       if z = = z.right
 8            H.min = NIL
 9       else H.min = z.right
10            CONSOLIDATE(H)
11       H.n = H.n−1
12  return z
```

如图19-4所示，FIB-HEAP-EXTRACT-MIN 首先将最小结点的每个孩子变为根结点，并从根链表中删除该最小结点。然后通过把具有相同度数的根结点合并的方法来链接成根链表，直到每个度数至多只有一个根在根链表中。

首先第1行保存一个指向最小结点的指针 z，该程序最后返回这个指针。如果 z 为 NIL，那么斐波那契堆为空，可以结束；否则，在第3～5行中让 z 的所有孩子成为 H 的根结点(把它们插入根链表)，并在第6行从根链表中移除 z，这样 z 便从 H 中删除了。执行完第6行之后，如果 z 是它自身的右兄弟，那么 z 是根链表中仅有的一个结点并且它没有孩子结点。这样所有剩下的工作是在返回 z 之前，第8行使斐波那契堆成为空堆。否则，把指针 $H.min$ 指向根链表中除 z 之外的某个根结点(这里是 z 的右兄弟)，该根结点没有必要一定是 FIB-HEAP-EXTRACT-MIN 执行完后的新的最小结点。图19-4(b)所示的是图19-4(a)执行完第9行之后的斐波那契堆。

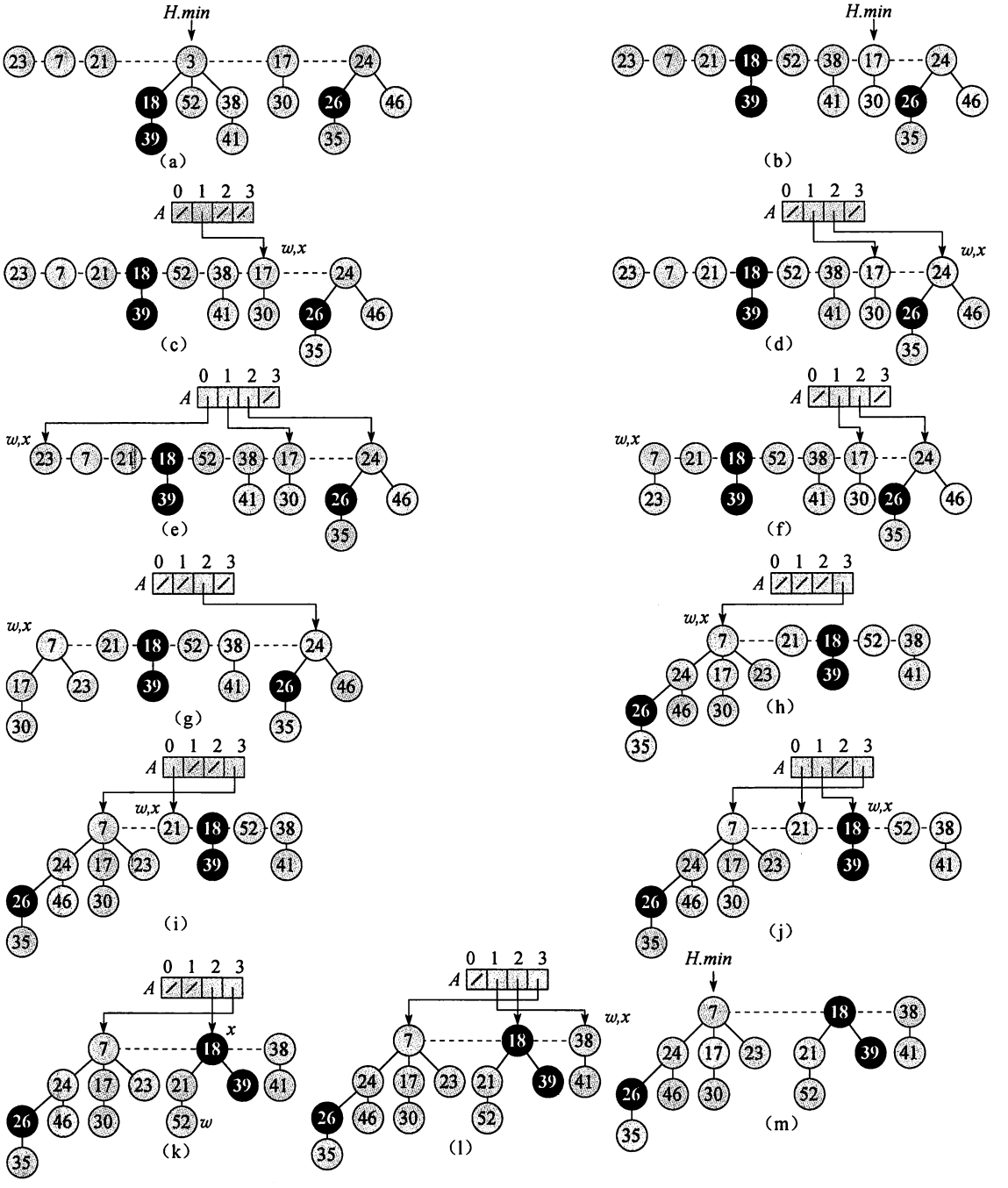

图 19-4　FIB-HEAP-EXTRACT-MIN 的执行过程。(a)斐波那契堆 H。(b)从根链表中移除最小结点 z 并把它的孩子加入根链表后的情形。(c)～(e)过程 CONSOLIDATE 中第 4～14 行 **for** 循环的前三次迭代中，每一次迭代完后的树以及数组 A 的情况。该过程对根链表的处理是从 $H.min$ 指向的结点开始的，并沿着 *right* 指针的方向进行。每个图都显示出了每一次迭代之后的 w 和 x 值。(f)～(h)**for** 循环接下来的一次迭代，显示了第 7～13 行的 **while** 循环每一次迭代之后的 w 和 x 值。图(f)展示了 **while** 循环第一次执行后的情形。关键字为 23 的结点被链接到关键字为 7 的结点，后者也是 x 当前所指向的结点。图(g)中，关键字为 17 的结点被链接到关键字为 7 的结点，后者仍由 x 所指向。图(h)中，关键字为 24 的结点被链接到关键字为 7 的结点。由于之前 $A[3]$没有指向任何结点，在 **for** 的这一次迭代后，$A[3]$被设为指向结果树的根结点。(i)～(l)**for** 循环剩下的 4 次迭代中每一次迭代后的情形。(m)依据数组 A 重构根链表以及确定新的 $H.min$ 指针后的斐波那契堆 H

下一步是**合并**(consolidating)H的根链表，通过调用CONSOLIDATE(H)来减少斐波那契堆中树的数目。合并根链表的过程为重复执行以下步骤，直到根链表中的每一个根有不同的度数。

1. 在根链表中找到两个具有相同度数的根x和y。不失一般性，假定$x.key \leqslant y.key$。

2. 把y链接到x：从根链表中移除y，调用FIB-HEAP-LINK过程，使y成为x的孩子。该过程将$x.degree$属性增1，并清除y上的标记。

过程CONSOLIDATE使用一个辅助数组$A[0..D(H.n)]$来记录根结点对应的度数的轨迹。如果$A[i]=y$，那么当前的y是一个具有$y.degree=i$的根。当然，为了分配数组必须知道如何计算最大度数的上界$D(H.n)$，但这些将在19.4节中介绍。

513 ~ 515

```
CONSOLIDATE(H)
 1  let A[0..D(H.n)] be a new array
 2  for i = 0 to D(H.n)
 3      A[i]=NIL
 4  for each node w in the root list of H
 5      x = w
 6      d = x.degree
 7      while A[d] ≠ NIL
 8          y = A[d]   // another node with the same degree as x
 9          if x.key > y.key
10              exchange x with y
11          FIB-HEAP-LINK(H,y, x)
12          A[d]= NIL
13          d = d + 1
14      A[d]= x
15  H.min = NIL
16  for i = 0 to D(H.n)
17      if A[i] ≠ NIL
18          if H.min == NIL
19              create a root list for H containing just A[i]
20              H.min = A[i]
21          else insert A[i] into H's root list
22              if A[i].key < H.min.key
23                  H.min = A[i]

FIB-HEAP-LINK(H, y, x)
1  remove y from the root list of H
2  make y a child of x, incrementing x.degree
3  y.mark = FALSE
```

具体地说，CONSOLIDATE过程工作如下。第1～3行分配数组A，并将数组A的每个元素初始化为NIL。第4～14行的**for**循环处理根链表中的每个根结点w。由于要把根链接起来，因此w可能被链接到其他的结点上，不再是一个根。然而，w必然在以某个结点x为根树内，x可能是也可能不是w本身。因为想要每个根都有不同的度数，所以查找数组A来确定是否有某个根y与x有相同度数。如果有，则把根x和y链接起来，并保证链接完后x仍然是一个根。也就是说，如果y的关键字小于x的关键字，则先交换指向这两个根的指针，再把y链接到x。在y链接到x以后，x的度数增加1，继续执行这个过程，把x和另一个与x的新度数相同的根链接，直到处理过的根没有与x有相同的度数。然后，将A的相应关元素指向x。这样处理后续根时，已经记录x是已处理过的根中具有该度数的唯一根。当这个**for**循环结束时，每个度数下至多只

516

有一个根，数组 A 指向每个剩下的根。

第 7～13 行的 **while** 循环重复地将包含结点 w 的以 x 为根的树与和 x 度数相同的根相链接，直到没有其他的根与 x 有相同的度数。这个 **while** 循环维持了如下的不变式：

在 **while** 循环的每次迭代开始处，$d=x.degree$

使用这一循环不变式如下：

初始化：第 6 行确保第一次进入该循环时，该循环不变式成立。

保持：在 **while** 循环的每一次迭代中，$A[d]$指向某个根 y。因为 $d=x.degree=y.degree$，因此要链接 x 和 y。不论 x 和 y 中哪个具有更小的关键字，链接操作之后，该结点成为另一个结点的父结点，因此如有必要，第 9～10 行交换指向 x 和 y 的指针。接下来，在第 11 行通过调用 FIB-HEAP-LINK(H，y，x)把 y 链接到 x。这个调用增加了 $x.degree$ 值，而 $y.degree$ 仍为 d。结点 y 不再是一个根结点，因此第 12 行从数组 A 中删除指向它的指针。由于调用 FIB-HEAP-LINK 增加了 $x.degree$ 的值，第 13 行恢复不变式 $d=x.degree$。

终止：重复 **while** 循环直到 $A[d]$=NIL，在这种情形下，没有其他的根与 x 有相同的度数。

while 循环结束后，在第 14 行将 $A[d]$设为 x，并执行 **for** 循环的下一轮迭代。

图 19-4(c)～(e)示出了第 4～14 行 **for** 循环前三轮迭代后的数组 A 和结果树。在 **for** 循环的下一轮迭代中，发生了三次链接，它们的结果如图 19-4(f)～(h)所示。图 19-4(i)～(l)展示了 **for** 循环接下来 4 轮迭代后的结果。

其余的工作就是清理。一旦第 4～14 行的 **for** 循环完成，第 15 行清空根链表，第 16～23 行依据数组 A 来重构根链表。最后得到的斐波那契堆如图 19-4(m)所示。根链表合并完后，FIB-HEAP-EXTARCT-MIN 在第 11 行减小 $H.n$，在第 12 行返回指向被删除的结点 z 的指针，然后结束程序。

现在来证明从一个 n 个结点的斐波那契堆中抽取最小结点的摊还代价为 $O(D(n))$。设 H 表 517
示执行 FIB-HEAP-EXTARCT-MIN 操作之前的斐波那契堆。

首先给出抽取最小结点的实际代价。FIB-HEAP-EXTRACT-MIN 最多处理最小结点的 $D(n)$ 个孩子，再加上 CONSOLIDATE 中第 2～3 行和第 16～23 行的工作，合计需要的时间代价为 $O(D(n))$。剩下的是分析 CONSOLIDATE 中第 4～14 行的 **for** 循环代价，这一部分我们使用聚合分析。因为原始的根链表中有 $t(H)$ 个结点，减去抽取出的结点，再加上抽取出的结点的孩子结点(至多为 $D(n)$)，所以调用 CONSOLIDATE 时根链表的大小最大为 $D(n)+t(H)-1$。给定第 4～14 行 **for** 循环的一轮迭代中，第 7～13 行的 **while** 循环的迭代次数取决于根链表。但是我们知道每次调用 **while** 循环，总有一个根结点被链接到另一个上，因此 **for** 循环的所有迭代中，**while** 循环的总次数最多为根链表中根的数目。因此，**for** 循环的总工作量最多与 $D(n)+t(H)$ 成正比。所以，抽取最小结点的总实际工作量为 $O(D(n)+t(H))$。

抽取最小结点之前的势为 $t(H)+2m(H)$，因为最多有 $D(n)+1$ 个根留下且在该过程中没有任何结点被标记，所以在该操作之后势最大为 $(D(n)+1)+2m(H)$。所以摊还代价最多为：

$$\begin{aligned}&O(D(n)+t(H))+((D(n)+1)+2m(H))-(t(H)+2m(H))\\=\;&O(D(n))+O(t(H))-t(H)\\=\;&O(D(n))\end{aligned}$$

因为可以增大势的单位来支配隐藏在 $O(t(H))$ 中的常数。直观上讲，由于每次链接操作均把根的数目减小 1，因此每次链接操作的代价可以由势的减小来支付。我们将在 19.4 节中看到 $D(n)=O(\lg n)$，因此抽取最小结点的摊还代价为 $O(\lg n)$。

练习

19.2-1　给出图 19-4(m)中的斐波那契堆调用 FIB-HEAP-EXTRACT-MIN 后得到的斐波那契堆。

19.3 关键字减值和删除一个结点

本节介绍如何在 $O(1)$ 的摊还时间内减小斐波那契堆中某个结点的关键字的值，以及如何在
518 $O(D(n))$ 摊还时间内从一个 n 个结点的斐波那契堆中删除一个结点。19.4 节将证明最大度数 $D(n)$ 是 $O(\lg n)$，这可以推出 FIB-HEAP-EXTRACT-MIN 和 FIB-HEAP-DELETE 能在 $O(\lg n)$ 的摊还时间代价内完成。

关键字减值

在下面 FIB-HEAP-DECREASE-KEY 操作的伪代码中，与前面一样假定从一个链表中移除一个结点不改变被移除的结点的任何结构属性。

```
FIB-HEAP-DECREASE-KEY(H, x, k)
1  if k > x.key
2      error "new key is greater than current key"
3  x.key = k
4  y = x.p
5  if y ≠ NIL and x.key < y.key
6      CUT(H, x, y)
7      CASCADING-CUT(H, y)
8  if x.key < H.min.key
9      H.min= x

CUT(H, x, y)
1  remove x from the child list of y, decrementing y.degree
2  add x to the root list of H
3  x.p = NIL
4  x.mark = FALSE

CASCADING-CUT(H,y)
1  z = y.p
2  if z ≠ NIL
3      if y.mark = = FALSE
4          y.mark = TRUE
5      else CUT(H, y, z)
6          CASCADING-CUT(H,z )
```

FIB-HEAP-DECREASE-KEY 过程工作如下。第 1～3 行保证新的关键字不比 x 的当前关键字大，然后把新的关键字赋值给 x。如果 x 是根结点，或者如果 $x.key \geqslant y.key$，此处 y 是 x 的父结点，那么不需要进行结构上的任何改变，因为没有违反最小堆序。第 4～5 行即为测试这一条件。

如果违反了最小堆序，那么需要进行很多改变。首先在第 6 行**切断**(cuting)x。CUT 过程“切
519 断”x 与其父结点 y 之间的链接，使 x 成为根结点。

我们使用 *mark* 属性来得到需要的时间界。该属性记录了每个结点的一小段历史。假定下面的步骤已经发生在结点 x 上：

1. 在某个时刻，x 是根。
2. 然后 x 被链接到另一个结点(成为孩子结点)。
3. 然后 x 的两个孩子被切断操作移除。

一旦失掉第二个孩子，就切断 x 与其父结点的链接，使它成为一个新的根。如果发生了第 1 步和第 2 步且 x 的一个孩子被切掉，那么属性 $x.mark$ 为 TRUE。因此，由于 CUT 过程执行了

第 1 步，所以它在第 4 行清除 $x.mark$。(现在我们知道了为什么 FIB-HEAP-LINK 中第 3 行清除 $y.mark$：因为结点 y 正被链接到另一个结点上，即上面的第 2 步正被执行。下一次如果 y 的一个孩子被切掉，则 $y.mark$ 将被设为 TRUE。)

我们的工作还没有完成，因为 x 可能是其父结点 y 被链接到另一个结点后被切掉的第二个孩子。因此，FIB-HEAP-DECREASE-KEY 的第 7 行尝试在结点 y 上执行一次**级联切断**(cascading-cut)操作。如果 y 是一个根结点，那么 CASCADING-CUT 的第 2 行测试将使得该过程返回。如果 y 是未被标记的结点，既然它的第一个孩子已经被切掉，那么该过程在第 4 行标记它，并返回。然而，如果 y 是被标记过的，则 y 刚刚失去了它的第二个孩子，那么 y 在第 5 行被切掉，且第 6 行 CASCADING-CUT 递归调用它本身来处理 y 的父结点 z。CASCADING-CUT 过程沿着树一直递归向上，直到它遇到根结点或者一个未被标记的结点。

一旦所有的级联切断都完成，如果有必要，FIB-HEAP-DECREASE-KEY 的第 8～9 行就更新 $H.min$，然后结束程序。唯一一个关键字发生改变的结点是关键字被减小的结点 x。因此，新的最小结点要么是原来的最小结点，要么是结点 x。

图 19-5 展示了两次调用 FIB-HEAP-DECREASE-KEY 的执行过程，初始的斐波那契堆如图 19-5(a)所示。图 19-5(b)所示的是第一次调用，其中不涉及任何级联切断。图 19-5(c)～(e)中所示的是第二次调用，其中引发了两次级联切断。

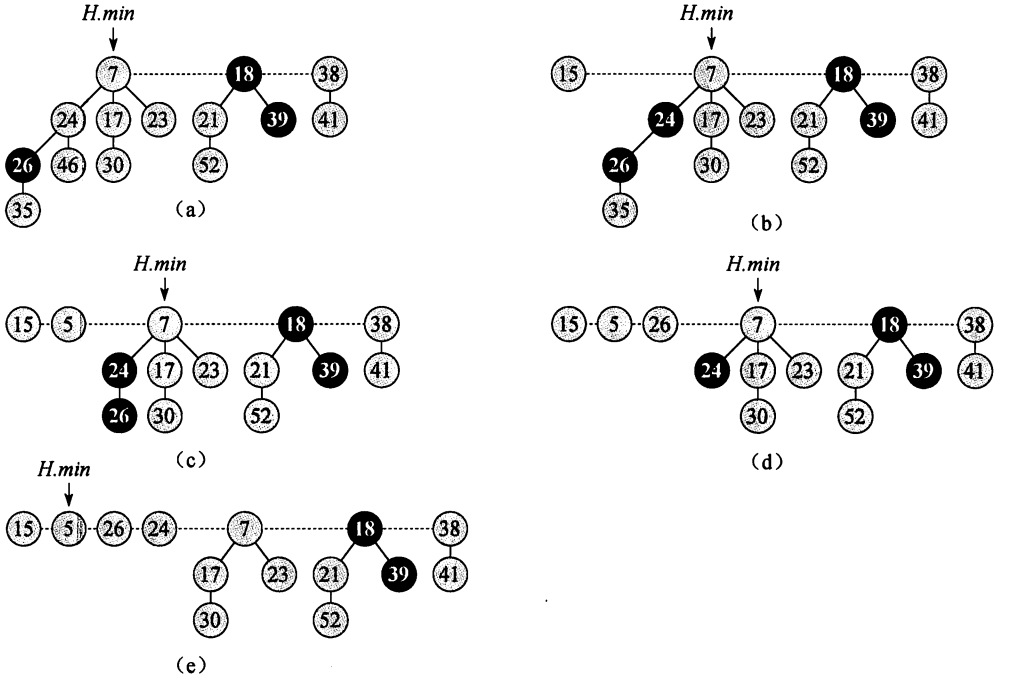

图 19-5　FIB-HEAP-DECREASE-KEY 的两次调用。(a)初始的斐波那契堆。(b)关键字为 46 的结点将关键字减小到 15。该结点成为一个根结点，它的父结点(具有关键字 24)之前没有被标记，现在被标记了。(c)～(e)关键字为 35 的结点将关键字减小到 5。图(c)中，该结点的关键字已经为 5，变为根结点。它的父结点(关键字为 26)是被标记过的，因此需要调用一个级联切断操作。关键字为 26 的结点被从父结点上剪切下来，成为图(d)中的一个未被标记的根。另一个级联切断操作需要执行，因为关键字为 24 的结点也已经被标记。该结点被从它的父结点上剪切下来，成为图(e)中的一个未被标记的根。因为关键字为 7 的结点是个根，所以级联切断操作在此结束。(即使这个结点不是根，级联操作也会结束，因为它未被标记。)图(e)展示了 FIB-HEAP-DECREASE-KEY 操作后的结果，其中 $H.min$ 指向了新的最小结点

现在来证明 FIB-HEAP-DECREASE-KEY 的摊还代价为 $O(1)$。先来推导它的实际代价。FIB-HEAP-DECREASE-KEY 过程需要 $O(1)$ 的时间，还需加上级联切断操作的时间。假定一个给定的 FIB-HEAP-DECREASE-KEY 调用中，要调用 c 次 CASCADING-CUT（FIB-HEAP-DECREASE-KEY 中第 7 行的调用引发了 CASCADING-CUT 的 $c-1$ 次递归调用）。CASCADING-CUT 的每一次调用(不包括递归调用)需要 $O(1)$ 的时间。因此，包含所有的递归调用后，FIB-HEAP-DECREASE-KEY 的实际代价为 $O(c)$。

接下来计算势的变化。设 H 是 FIB-HEAP-DECREASE-KEY 操作执行之前的斐波那契堆。FIB-HEAP-DECREASE-KEY 的第 6 行调用 CUT 创建了一棵以结点 x 为根的新树，并清除了 x 的标记位(该标记位可能已经是 FALSE)。除了最后一次调用，其他每一次调用 CASCADING-CUT，均切掉一个标记过的结点并清除该结点的标记位。此后，斐波那契堆包含 $t(H)+c$ 棵树(原来的 $t(H)$ 棵数，$c-1$ 棵被级联切断操作产生的树，以及以结点 x 为根的树)，而且最多有 $m(H)-c+2$ 个被标记的结点($c-1$ 个结点被级联切断操作清除标记，最后一次调用 CASCADING-CUT 可能又标记了一个结点)。因此势的变化最多为：

520
~
521

$$((t(H)+c)+2(m(H)-c+2))-(t(H)+2m(H))=4-c$$

因此，FIB-HEAP-DECREASE-KEY 的摊还代价至多是

$$O(c)+4-c=O(1)$$

因为可以将势的单位增大到能支配 $O(c)$ 中隐藏的常数。

现在读者应该清楚为什么在定义势函数时，要包含一个 2 倍于标记结点数目的项。当一个标记的结点 y 被一个级联切断操作切掉时，它的标记位被清空，这使得势减小 2。一个单位的势支付切断和标记位的清除，另一个单位补偿了因为结点 y 变成根而增加的势。

删除一个结点

下面的伪代码在 $O(D(n))$ 的摊还时间内从一个具有 n 个结点的斐波那契堆中删除一个结点。假定在斐波那契堆中任何关键字的当前值均不为 $-\infty$。

```
FIB-HEAP-DELETE(H, x)
1  FIB-HEAP-DECREASE-KEY(H, x, -∞)
2  FIB-HEAP-EXTRACT-MIN(H)
```

FIB-HEAP-DELETE 把唯一的最小关键字 $-\infty$ 赋予 x，将 x 变为斐波那契堆中最小的结点。然后 FIB-HEAP-EXTRACT-MIN 过程从斐波那契堆中移除 x。FIB-HEAP-DELETE 的摊还时间为 FIB-HEAP-DECREASE-KEY 的 $O(1)$ 摊还时间与 FIB-HEAP-EXTRACT-MIN 的 $O(D(n))$ 摊还时间之和。因为在 19.4 节中将证明 $D(n)=O(\lg n)$，所以 FIB-HEAP-DELETE 的摊还时间为 $O(\lg n)$。

练习

19.3-1 假定斐波那契堆中一个根 x 被标记了。解释 x 是如何成为一个被标记的根的。试说明 x 是否被标记对分析并没有影响，即使它不是一个先被链接到另一个结点，后又丢失了一个孩子的根。

19.3-2 使用聚合分析来证明 FIB-HEAP-DECREASE-KEY 的 $O(1)$ 摊还时间是每一个操作的平均代价。

522

19.4 最大度数的界

要证明 FIB-HEAP-EXTARCT-MIN 和 FIB-HEAP-DELETE 的摊还时间为 $O(\lg n)$，必需证明一个具有 n 个结点的斐波那契堆中任意结点的度数的上界 $D(n)$ 为 $O(\lg n)$。特别地，要证明

$D(n) \leqslant \lfloor \log_{\phi} n \rfloor$，这里 ϕ 是公式(3.24)中定义的黄金分割率：

$$\phi = (1+\sqrt{5})/2 = 1.61803\cdots$$

这个分析的关键如下。对于斐波那契堆中的每个结点 x，定义 size(x) 为以 x 为根的子树中包括 x 本身在内的结点个数(注意 x 并不是必须在根链表中，它可以是任意的结点。)我们将证明 size(x) 是 $x.degree$ 的幂。请记住：$x.degree$ 始终是 x 的度数的准确计数。

引理 19.1　*设 x 是斐波那契堆中的任意结点，并假定 $x.degree=k$。设 y_1，y_2，…，y_k 表示 x 的孩子，并以它们链入 x 的先后顺序排列，则 $y_1.degree \geqslant 0$，且对于 $i=2$，3，…，k，有 $y_i.degree \geqslant i-2$。*

证明　显然，$y_1.degree \geqslant 0$。对于 $i \geqslant 2$，注意到当 y_i 被链入 x 的时候，y_1，y_2，…，y_{i-1} 已经是 x 的孩子，因此一定有 $x.degree \geqslant i-1$。因为结点 y_i 只有在 $x.degree = y_i.degree$ 的时候，才会被链入 x(执行操作 CONSOLIDATE)，此时也一定有 $y_i.degree \geqslant i-1$。从这之后，结点 y_i 最多失去一个孩子，因为如果它失去了两个孩子，它将被从 x 中剪切掉(执行操作 CASCADING-CUT)。综上，$y_i.degree \geqslant i-2$。 ■

我们终于可以解释“斐波那契堆”这个名字的由来了。回顾 3.2 节，对于 $k=0$，1，2，…，第 k 个斐波那契数被定义为如下递归式：

$$F_k = \begin{cases} 0 & \text{如果 } k = 0 \\ 1 & \text{如果 } k = 1 \\ F_{k-1} + F_{k-2} & \text{如果 } k \geqslant 2 \end{cases}$$

下面的引理给出了另一种表示 F_k 的方法。 523

引理 19.2　*对于所有的整数 $k \geqslant 0$，*

$$F_{k+2} = 1 + \sum_{i=0}^{k} F_i$$

证明　对 k 进行归纳。当 $k=0$ 时，

$$1 + \sum_{i=0}^{0} F_i = 1 + F_0 = 1 + 0 = F_2$$

现做归纳假设 $F_{k+1} = 1 + \sum_{i=0}^{k-1} F_i$，那么有

$$F_{k+2} = F_k + F_{k+1} = F_k + \left(1 + \sum_{i=0}^{k-1} F_i\right) = 1 + \sum_{i=0}^{k} F_i$$ ■

引理 19.3　*对于所有的整数 $k \geqslant 0$，斐波那契数的第 $k+2$ 个数满足 $F_{k+2} \geqslant \phi^k$。*

证明　对 k 进行归纳。归纳基础是 $k=0$ 和 $k=1$ 的情形。当 $k=0$ 时，有 $F_2 = 1 = \phi^0$，并且当 $k=1$ 时，有 $F_3 = 2 > 1.619 > \phi^1$。归纳步是对于 $k \geqslant 2$，假定对于 $i=0$，1，…，$k-1$，有 $F_{i+2} > \phi^i$。回顾 ϕ 是等式(3.23) $x^2 = x+1$ 的正根。因此，有

$$\begin{aligned} F_{k+2} &= F_{k+1} + F_k \\ &\geqslant \phi^{k-1} + \phi^{k-2} && \text{(根据归纳假设)} \\ &= \phi^{k-2}(\phi+1) \\ &= \phi^{k-2} \cdot \phi^2 && \text{(根据等式(3.23))} \\ &= \phi^k \end{aligned}$$ ■

下面的引理和推论完成了整个分析。 524

引理 19.4　*设 x 是斐波那契堆中的任意结点，并设 $k=x.degree$，则有 size(x) $\geqslant F_{k+2} \geqslant \phi^k$，其中 $\phi = (1+\sqrt{5})/2$。*

证明　设 s_k 表示斐波那契堆中度数为 k 的任意结点的最小可能 size。平凡地，$s_0=1$，$s_1=2$。s_k 最大为 size(x)，且因为往一个结点上添加孩子不能减小该结点的 size，s_k 的值随着 k 单调递

增。在任意斐波那契堆中，考虑某个结点 z，有 $z.degree=k$ 和 $\text{size}(z)=s_k$。因为 $s_k \leqslant \text{size}(x)$，所以可以通过计算 s_k 的下界来得到 $\text{size}(x)$ 的一个下界。与引理 19.1 一样，用 y_1，y_2，…，y_k 表示结点 z 的孩子，并按照它们链入该结点的先后顺序排列。为了求 s_k 的界，把 z 本身和 z 的第一个孩子 y_1（$\text{size}(y_1) \geqslant 1$）各算一个，则有

$$\text{size}(x) \geqslant s_k \geqslant 2+\sum_{i=2}^{k} s_{y_i.degree} \geqslant 2+\sum_{i=2}^{k} s_{i-2}$$

其中最后一行由引理 19.1（因此有 $y_i.degree \geqslant i-2$），以及 s_k 的单调性（因此有 $s_{y_i.degree} \geqslant s_{i-2}$）得到。

现在对 k 进行归纳证明，对于所有的非负整数 k，有 $s_k \geqslant F_{k+2}$。归纳基础，$k=0$ 和 $k=1$ 时显然成立。对于归纳步，假定 $k \geqslant 2$ 且对于 $i=0$，1，…，$k-1$，均有 $s_k \geqslant F_{i+2}$，则有

$$\begin{aligned} s_k &\geqslant 2+\sum_{i=2}^{k} s_{i-2} \geqslant 2+\sum_{i=2}^{k} F_i = 1+\sum_{i=0}^{k} F_i \\ &= F_{k+2} \qquad \text{（根据引理 19.2）} \\ &\geqslant \phi^k \qquad \text{（根据引理 19.3）} \end{aligned}$$

525 于是就证明了 $\text{size}(x) \geqslant s_k \geqslant F_{k+2} \geqslant \phi^k$。 ■

推论 19.5 一个 n 个结点的斐波那契堆中任意结点的最大度数 $D(n)$ 为 $O(\lg n)$。

证明 设 x 是一个 n 个结点的斐波那契堆中的任意结点，并设 $k=x.degree$。依据引理 19.4，有 $n \geqslant \text{size}(x) \geqslant \phi^k$。取以 ϕ 为底的对数，得到 $k \leqslant \log_\phi n$。（实际上，因为 k 是整数，所以 $k \leqslant \lfloor \log_\phi n \rfloor$。）所以，任意结点的最大度数 $D(n)$ 为 $O(\lg n)$。 ■

练习

19.4-1 Pinocchio 教授声称一个 n 个结点的斐波那契堆的高度是 $O(\lg n)$ 的。对于任意的正整数 n，试给出经过一系列斐波那契堆操作后，可以创建出一个斐波那契堆，该堆仅仅包含一棵具有 n 个结点的线性链的树，以此来说明该教授是错误的。

19.4-2 假定对级联切断操作进行推广，对于某个整数常数 k，只要一个结点失去了它的第 k 个孩子，就将其从它的父结点上剪切掉（19.3 节中为 $k=2$ 的情形）。k 取什么值时，有 $D(n)=O(\lg n)$。

思考题

19-1 （删除操作的另一种实现） Pisano 教授提出了下面的 FIB-HEAP-DELETE 过程的一个变种，声称如果删除的结点不是由 $H.min$ 指向的结点，那么该程序运行得更快。

```
PISANO-DELETE(H, x)
1  if x == H.min
2      FIB-HEAP-EXTRACT-MIN(H)
3  else y = x.p
4      if y ≠ NIL
5          CUT(H, x, y)
6          CASCADING-CUT(H, y)
7      add x's child list to the root list of H
526    remove x from the root list of H
```

a. 该教授的声称是基于第 7 行可以在 $O(1)$ 实际时间内完成的这一假设，它的程序可以运行得更快。该假设有什么问题吗？

b. 当 x 不是由 $H.min$ 指向时，给出 PISANO-DELETE 实际时间的一个好上界。你给出的

上界应该以 $x.degree$ 和调用 CASCADING-CUT 的次数 c 这两个参数来表示。

c. 假定调用 PISANO-DELETE(H，x)，并设 H'是执行后得到的斐波那契堆。假定结点 x 不是一个根，用 $x.degree$、c、$t(H)$和 $m(H)$来表示 H'势的界。

d. 证明：PISANO-DELETE 的摊还时间渐近地不好于 FIB-HEAP-DELETE 的摊还时间，即使在 $x \neq H.min$ 时也是如此。

19-2 （二项树和二项堆） 二项树 B_k 是一棵递归定义的有序树(参看 B.5.2 节)。如图 19-6(a)所示，二项树 B_0 仅包含一个结点。二项树 B_k 是由两个二项树 B_{k-1}组成的，这两棵树按照一棵树的根是另一棵树根的最左孩子的方式链接。图 19-6(b)所示为二项树 B_0 到 B_4。

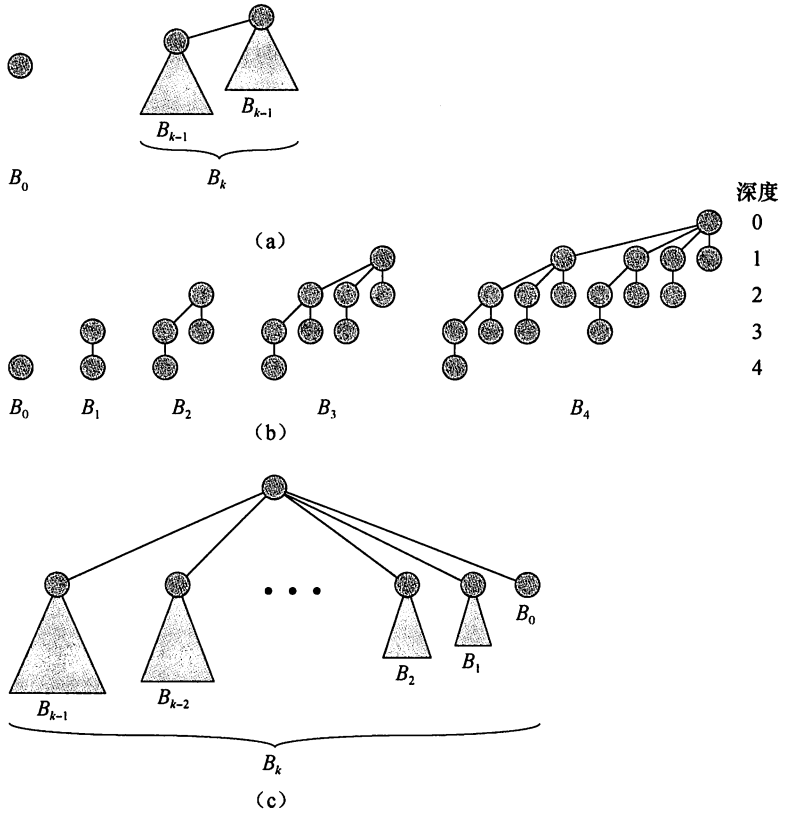

图 19-6　(a)递归定义的二项树 B_k。三角形表示有根子树。(b)二项树 B_0 到 B_4。B_4 中结点的深度数已经给出。(c)另一种角度看二项树 B_k

a. 对于二项树 B_k，证明：

1. 一共有 2^k 个结点。
2. 树的高度是 k。
3. 对于 $i=0, 1, \cdots, k$，深度为 i 的结点恰有 $\binom{k}{i}$ 个。
4. 根的度数为 k，它比其他任意结点的度数都要大。此外，如图 19-6(c)所示，如果把根的孩子从左到右编号为 $k-1, k-2, \cdots, 0$，那么孩子 i 是子树 B_i 的根。

二项堆(binomial heap)H 是具备如下性质的二项树的集合：

1. 每个结点具有一个关键字(与斐波那契堆相同)。
2. H 中的每个二项树遵循最小堆性质。
3. 对于任意的非负整数 k，H 中最多有一个二项树的根的度数为 k。

b. 假定一个二项堆 H 一共有 n 个结点。讨论 H 中包含的二项树与 n 的二进制表示之间的关系。并证明 H 最多由 $\lfloor \lg n \rfloor + 1$ 棵二项树组成。

假定按如下方式表述二项堆。用 10.4 节中的左孩子、右邻兄弟方案来表示二项堆中的每一棵二项树。每个结点包含一个关键字，指向它父结点的指针、指向它最左孩子的指针和指向与它右邻兄弟的指针(这些指针一些情况下是 NIL)，以及它的度数(如同斐波那契堆，表示为有多少个孩子)。这些根组成了一个单向链接的根链表，并以根的度数从小到大排列。可以通过一个指向根链表第一个结点的指针来访问二项堆。

527 ~ 528

c. 完整描述如何表示一个二项堆(例如，对属性进行命名，描述属性值什么时候为 NIL，定义根链表是怎么组织的)，并说明如何用与本章中实现斐波那契堆一样的方式来实现在二项堆上同样的 7 个操作。每一个操作的最坏时间应该为 $O(\lg n)$，其中 n 为二项堆中的结点数目(或对于 UNION 操作，为要被合并的两个二项堆中的结点数)。MAKE-HEAP 操作应为常数时间。

d. 假定仅仅要实现在一个斐波那契堆上的可合并堆操作(即并不实现 DECREASE-KEY 和 DELETE 操作)。斐波那契堆中的树与二项堆中的树有何相似之处？有什么区别？证明在一个 n 个结点的斐波那契堆中最大度数最多为 $\lfloor \lg n \rfloor$。

e. McGee 教授提出了一个基于斐波那契堆的新的数据结构。一个 McGee 堆具有与斐波那契堆相同的结构，并且只支持可合并堆操作。除了插入和合并在最后一步中合并根链表外，其他操作的实现方式均与斐波那契堆中的实现方式相同。McGee 堆上各操作的最坏情况运行时间是多少？

19-3 (*更多的斐波那契堆操作*) 想要扩展斐波那契堆 H 支持两个新操作，要求不改变斐波那契堆其他操作的摊还时间。

a. 操作 FIB-HEAP-CHANGE-KEY(H, x, k)将结点 x 中关键字的值改为 k。给出 FIB-HEAP-CHANGE-KEY 的一个有效实现，并分析当 k 大于、小于或等于 $x.key$ 时，各情形下的摊还运行时间。

b. 给出 FIB-HEAP-PRUNE(H, r)的一个有效实现，该操作从 H 中删除 $q = \min(r, H.n)$ 个结点。可以选择任意 q 个结点来删除。试分析你的实现的摊还运行时间。(*提示*：可能需要修改数据结构以及势函数。)

19-4 (2-3-4 堆) 第 18 章介绍了 2-3-4 树，树中每个内部结点(而不是根)有 2 个、3 个或 4 个孩子，且所有的叶结点有相同的深度。在本问题中，实现支持可合并堆操作的 2-3-4 堆。

2-3-4 堆以下几点与 2-3-4 树不同。在 2-3-4 堆中，仅仅叶结点存储关键字，并且每个叶结点 x 仅仅在属性 $x.key$ 中存储一个关键字。叶结点中的关键字可能以任意顺序存在。

529

每个内部结点 x 包含一个值 $x.small$，它等于以 x 为根的子树中叶结点存储的最小的关键字。根 r 包含一个属性 $r.height$，存储树的高度。最后，2-3-4 堆设计为存放在主存中，这样磁盘的读/写是不需要的。

实现下面的 2-3-4 堆操作。在一个具有 n 个元素的 2-3-4 堆上，(a)～(e)中每一个操作应该在 $O(\lg n)$ 时间内完成。(f)中的 UNION 操作应该在 $O(\lg n)$ 时间内完成，其中 n 为输入的两个堆元素个数之和。

a. MINIMUM，该操作返回一个指向具有最小关键字的叶结点的指针。

b. DECREASE-KEY，该操作将一个给定的叶结点 x 的关键字减小为给定的值 $k \leqslant x.key$。

c. INSERT，该操作插入一个关键字为 k 的叶结点 x。

d. DELETE，该操作删除一个给定的叶结点 x。

e. EXTRACT-MIN，该操作抽取具有最小关键字的叶结点。

f. UNION，该操作合并两个 2-3-4 堆，并返回一个单独的 2-3-4 堆，销毁掉输入的堆。

本章注记

Fredman 和 Tarjan[114]提出了斐波那契堆。他们的论文也描述了斐波那契堆在一些问题上的应用：单源最短路径、所有点对之间的最短路径、加权二分图匹配和最小生成树问题。

随后，Driscoll、Gabow、Shrairman 和 Tarjan[96]设计了有别于斐波那契堆的“松散堆”，该堆有两个变体。一个具有与斐波那契堆相同的摊还时间界。另一个允许 DECREASE-KEY 在 $O(1)$ 的最坏情况时间(不是摊还时间)内完成，DEXTACT-MIN 和 DELETE 在 $O(\lg n)$ 最坏情况时间内完成。松散堆在并行算法中也要比斐波那契堆更优越些。

在第 6 章的“本章注记”中，也提到一些其他数据结构，当 EXTRACT-MIN 调用的返回值序列随时间单调递增并且这些值在某一特定的范围内是整数时，这些数据结构支持更快的 DECREASE-KEY 操作。 530