第六章 堆排序

堆排序($heapsort$):像合并排序而不像插入排序，堆排序的运行时间为$O(n \lg n)$；像插入排序而不像合并排序，它是一种原地($in\ place$)排序算法：在任何时候，数组中只有常数个元素在输入数组外。它结合了插入排序与合并排序两种算法的优点。

（二叉）堆数据结构是一种数组对象，它可以被视为一棵完全二叉树。

表示堆的数组A是一个具有两个属性的对象：$length[A]$是数组中的元素个数；$heap-size[A]$是存放在A中的堆的元素个数。虽然$A[1..length[A]]$中可以包含有效值，但是$A[heap-size[A]]$之后的元素都不属于相应的堆。此处$heap-size[A] \leqslant length[A]$，树的根为$A[1]$，对于给定了的某个结点的下标$i$，其父结点为$PARENT(i)$、左儿子为$LEFT(i)$、右儿子为$RIGHT(i)$。

计算方法：

```
PARENT(i)
    return ⌊i/2⌋
LEFT(i)
    return 2i
RIGHT(i)
    return 2i + 1
```

二叉堆有两种：最大堆（大根堆）和最小堆（小根堆）。

最大堆中除了根以外的每个结点i有$A[PARENT(i)] \geqslant A[i]$，最大元素在根结点中；

最小堆中除了根以外的每个结点i有$A[PARENT(i)] \leqslant A[i]$，最小元素在根结点中。

练习

6.1-1

最多的时候是当这棵完全二叉树为满二叉树的时候：$2^{h+1}-1$

最少的时候是当这棵完全二叉树的最后一层只有一个叶子的时候：$2^h-1+1=2^h$

6.1-2

根据 6.1-1：$2^h \leqslant n \leqslant 2^{h+1}-1$

1.$2^h \leqslant n \implies h \leqslant \lg n$

2.$n \leqslant 2^{h+1}-1 \implies n \geqslant \lg(n+1)-1 > \lg n - 1$

$\therefore \lg n - 1 < h < \lg n$

即$h = \lfloor \lg n \rfloor$

6.1-3

如果不是的话会违反$A[PARENT(i)] \geqslant A[i]$，因此构造任何实例用反证法证明即可。

6.1-4

二叉树最底层的最右边的叶子。

6.1-5

不一定，也可能是最大堆。

6.1-6

不是。（值为 5、6、7 的三个元素违反了最大堆的性质）

6.1-7

设度为 0、1、2 的结点数量分别是n_0、n_1、n_2。$n = n_0 + n_1 + n_2$，n_0是叶子结点数。

由堆的性质可以知道：$n_1 = 0$或$n_1 = 1$，$n_0 = n_2 + 1$

$\therefore n_0 = \frac{n}{2}\ or\ n_0 = \frac{n+1}{2}$

1.$n_1 = 0 \implies n_1 + n_2 = \frac{n-1}{2}$

2.$n_1 = 1 \implies n_1 + n_2 = \frac{n}{2}$

即$n_1 + n_2 = \frac{n-1}{2}\ or\ n_1 + n_2 = \frac{n}{2}$，故$n_1 + n_2 = \lfloor n/2 \rfloor$

故叶子结点的下标从$\lfloor n/2 \rfloor + 1$开始（因为前面有$\lfloor n/2 \rfloor$个结点）

```
MAX-HEAPIFY(A, i)
 1  l ← LEFT(i)
 2  r ← RIGHT(i)
 3  if l ⩽ heap − size[A] and A[l] > A[i]
 4      then largest ← l
 5      else  largest ← i
 6  if r ⩽ heap − size[A] and A[r] > A[largest]
 7      then largest ← r
 8  if largest ≠ i
 9      then exchangeA[i] ↔ A[largest]
10           MAX − HEAPIFY(A, largest)
```

当$MAX-HEAPIFY$作用在一棵以结点i为根、大小为n的子树上时，其运行时间为调整元素$A[i]$、$A[LEFT(i)]$和$A[RIGHT(i)]$的关系所用时间$\Theta(1)$，再加上对以i的某个子结点为根的子树递归调用$MAX-HEAPIFY$所需的时间。i结点的子树大小至多为$2n/3$。

最坏情况发生在最底层刚好半满：

$n = 2^h - 1 + 2^h \times \frac{1}{2}$，而最大子树结点数为$2^{1-1} + 2^{2-1} + \cdots + 2^{h-1} = 2^h - 1$

$$\frac{2^h - 1}{\frac{3}{2} \times 2^h - 1} \to \frac{2}{3}$$

所以有$T(n) \leqslant T(2n/3) + \Theta(1)$，用$T(n) = T(2n/3) + \Theta(1)$来计算上限(使用主方法)：

$\Theta(n^{\log_b a}) = \Theta(n^0) = \Theta(1)$，$f(n) = \Theta(1)$，因此可以解得$T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(\lg n)$

由于我们采用上限来计算，因此最终的结果应当是$T(n) = O(\lg n)$

也就是说，$MAX-HEAPIFY$作用于一个高度为h的结点所需的运行时间是$O(h)$

$MAX-HEAPIFY(A, i)$过程保证了以i结点为根的子树的最下层的一棵子树（3个结点组成的子树）是最大堆。

练习

6.2-1

$A = < 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0 >$

1. $A = < 27, 17, 10, 16, 13, 3, 1, 5, 7, 12, 4, 8, 9, 0 >$

2. $A =< 27, 17, 10, 16, 13, 9, 1, 5, 7, 12, 4, 8, 3, 0 >$

6.2-2

```
MIN-HEAPIFY(A, i)
 1  l ← LEFT(i)
 2  r ← RIGHT(i)
 3  if l ⩽ heap − size[A] and A[l] < A[i]
 4      then smallest ← l
 5      else  smallest ← i
 6  if r ⩽ heap − size[A] and A[r] < A[smallest]
 7      then smallest ← r
 8  if smallest ≠ i
 9      then exchangeA[i] ↔ A[smallest]
10           MIN − HEAPIFY(A, smallest)
```

对于随机的输入，两种算法的运行时间都是$O(h)$

6.2-3

不作任何改变。（顺序执行，结束$MAX - HEAPIFY(A, i)$过程）

6.2-4

对于$i > heap - size[A]/2$的结点都是叶子结点。所以调用$MAX - HEAPIFY$过程对原来的堆不做任何改变。

6.2-5

```
MAX-HEAPIFY(A, i)
 1  while TRUE
 2      do l ← LEFT(i)
 3         r ← RIGHT(i)
 4         if l ⩽ heap − size[A] and A[l] > A[i]
 5           then largest ← l
 6           else  largest ← i
 7         if r ⩽ heap − size[A] and A[r] > A[largest]
 8           then largest ← r
 9         if largest ≠ i
10           then exchangeA[i] ↔ A[largest]
11                i ← largest
12           else  break
```

6.2-6

题目有错误，应当是最优时间$\Omega(\lg n)$

对一个大小为n的堆，最优情况下经过的路径是涵盖了所有层最左边结点的路径。所以$\Theta(1) \times \Omega(\lg n) = \Omega(\lg n)$

这样就得到$MAX-HEAPIFY$过程运行时间的确界为$\Theta(\lg n)$

自底向上地用$MAX-HEAPIFY$来将一个数组$A[1..n]$（此处$n = length[A]$）变成一个最大堆。之数组$A[(\lfloor n/2 \rfloor + 1)..n]$都是叶子，因此对内部结点$A[1..\lfloor n/2 \rfloor]$调用$MAX-HEAPIFY$使得其满足最大堆的特性：

BUILD-MAX-HEAP(A)
1 $heap-size[A] \leftarrow length[A]$
2 **for** $i \leftarrow \lfloor length[A]/2 \rfloor$ **downto** 1
3 **do** MAX-HEAPIFY(A,i)

用循环不变式来证明此过程的正确性。

$BUILD-MAX-HEAP$过程使整棵树成为最大堆。

初始化：在第一轮循环迭代前，$i = \lfloor n/2 \rfloor$。结点$\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \cdots, n$都是叶结点，也是平凡最大堆的根。

保持：结点i的子结点的编号均比i大（二叉树结点编号的性质）。在循环中，使用的是从大到小的i（自底而上），因此这些子结点已经是最大堆根。这也是调用$MAX-HEAPIFY(A,i)$以使结点i成为最大堆的根的**前提**条件。而$MAX-HEAPIFY$的调用保持了结点$i+1, i+2, \cdots, n$为最大堆根的性质。（$MAX-HEAPIFY$过程的性质）。而当i变为$i-1$时，同样又使得$i-1$结点成为最大堆的根，并保持$i, i+1, \cdots, n$结点为最大堆的根。

终止：过程终止时，$i = 0$，此时$1, 2, \cdots, n$都是最大堆的根，特别地，结点1就是一个

最大堆的根。

因此$BUILD-MAX-HEAP$过程是正确的。

练习

6.3-1

$A =< 5,3,17,10,84,19,6,22,9 >$

1. $A =< 5,3,17,22,84,19,6,10,9 >$
2. $A =< 5,3,19,22,84,17,6,10,9 >$
3. $A =< 5,84,19,22,3,17,6,10,9 >$
4. $A =< 5,84,19,22,3,17,6,10,9 >$
5. $A =< 84,5,19,22,3,17,6,10,9 >$

6.3-2

这样是因为对当前结点进行操作时能够保证以当前结点为根的树的子树都已经是最大堆了。（降低操作时间）

6.3-3

区分：

结点的深度：与根节点的距离

结点的高度：与最远叶子的距离

题目应该是写错了，应当是至多有$\lceil n/2^{h+1} \rceil$

如果是求上限的话就当作他是一棵满二叉树来计算。

设树的深度为h（高度也是h），那么最底层共有2^h个结点，他们的高度是0，这棵二叉树共有$2^{h+1}-1$个结点。

$\left(2^{h+1}-1\right)/2^{0+1} \leqslant 2^h < \left(2^{h+1}-1\right)/2^{0+1}+1$

这个式子等价于

$$2^h - \frac{1}{2} \leqslant 2^h < 2^h + \frac{1}{2}$$

这个式子始终是成立的。

因此在含n个元素的堆中，至多有$\lceil n/2^{h+1} \rceil$个高度为h的结点。

堆排序算法：

首先使用$BUILD-MAX-HEAP$将输入数组$A[1..n]$构造成一个最大堆。数组中的最大元素在在根$A[1]$，通过与$A[n]$（数组的最后一个元素）互换，这时最大元素在数组的最后一个位置。去掉结点n不算（通过减小$heap-size[A]$实现），然后通过$MAX-HEAPIFY(A,1)$把$A[1..n-1]$建成最大堆。不断重复这个过程。使在每个过程中最大的元素从n结点开始从后往前排列。

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i ← length[A] downto 2
3        do exchange A[1] ↔ A[i]
4           heap-size[A] ← heap-size[A] − 1
5           MAX-HEAPIFY(A,1)
```

$HEAPSORT$过程的时间代价为$O(n \lg n)$。（调用$BUILD-MAX-HEAP$的时间为$O(n)$，$n-1$次$HEAP-MAX-HEAPIFY$调用中每一次的时间代价为$O(\lg n)$）。

练习

6.4-1

首先调用$BUILD-MAX-HEAP$过程：

$A =< 5, 13, 2, 25, 7, 17, 20, 8, 4 >$

1. $A =< 5, 13, 20, 25, 7, 17, 2, 8, 4 >$

2. $A =< 5, 13, 20, 25, 7, 17, 2, 8, 4 >$

3. $A =< 5, 25, 20, 13, 7, 17, 2, 8, 4 >$

4. $A =< 5, 25, 20, 13, 7, 17, 2, 8, 4 >$

5. $A =< 25, 5, 20, 13, 7, 17, 2, 8, 4 >$

6. $A =< 25, 13, 20, 5, 7, 17, 2, 8, 4 >$

7. $A =< 25, 13, 20, 5, 7, 17, 2, 8, 4 >$

8. $A =< 25, 13, 20, 8, 7, 17, 2, 5, 4 >$

然后

$A =< 13, 20, 8, 7, 17, 2, 5, 4 >$，$A' =< 25 >$，调用$MAX - HEAPIFY(A, 1)$：

1. $A =< 20, 13, 8, 7, 17, 2, 5, 4 >$

2. $A =< 20, 13, 8, 7, 17, 2, 5, 4 >$

3. $A =< 20, 17, 8, 7, 13, 2, 5, 4 >$

然后

$A =< 17, 8, 7, 13, 2, 5, 4 >$，$A' =< 20, 25 >$，调用$MAX - HEAPIFY(A, 1)$

然后

$A =< 8, 7, 13, 2, 5, 4 >$，$A' =< 17, 20, 25 >$，调用$MAX - HEAPIFY(A, 1)$：

1. $A =< 13, 7, 8, 2, 5, 4 >$

然后

$A =< 7, 8, 2, 5, 4 >$，$A' =< 13, 17, 20, 25 >$，调用$MAX - HEAPIFY(A, 1)$：

1. $A =< 8, 7, 2, 5, 4 >$

然后

$A =< 7, 2, 5, 4 >$，$A' =< 8, 13, 17, 20, 25 >$，调用$MAX - HEAPIFY(A, 1)$

然后

$A =< 2,5,4 >$，$A' =< 7,8,13,17,20,25 >$，调用$MAX-HEAPIFY(A,1)$：

1. $A =< 5,2,4 >$

然后

$A =< 5,2,4 >$，$A' =< 7,8,13,17,20,25 >$，调用$MAX-HEAPIFY(A,1)$

然后

$A =< 2,4 >$，$A' =< 5,7,8,13,17,20,25 >$，调用$MAX-HEAPIFY(A,1)$：

1. $A =< 4,2 >$

然后

$A =< 2 >$，$A' =< 4,5,7,8,13,17,20,25 >$，调用$MAX-HEAPIFY(A,1)$

最后

$A' =< 2,4,5,7,8,13,17,20,25 >$

6.4-2

假设A存放已排序数组

初始化：在循环开始前，A是空的，可以认为是已排序的。

保持：当A成为一个最大堆、$heap-size[A]=n$时，经过操作A中最大的元素$A[1]$放入A'，而$heap-size[A]=n-1$；而此时，经过$MAX-HEAPIFY(A,1)$过程后，$A[1..n-1]$成为最大堆，此时又将其中最大元素$A[1]$插入A'的最前面，然后$heap-size[A]=n-2$，而此时A'仍然是已排序的...

终止：$heap-size[A]=1$，A'中已经有$n-1$个元素，并且是已排序的，最后将原来A中的最小元素，即此时的$A[1]$插入A'前，整个过程结束，A'仍是已排序的。

优先级队列是一种用来维护由一组元素构成的集合S数据结构，这一组元素中的每一个都有一个关键字key。最个最大优先级队列支持以下操作：

$INSERT(S,x)$：把元素x插入集合S($S \leftarrow S \cup \{x\}$)

$MAXIMUM(S)$：返回S中具有最大关键字的元素

$EXTRACT-MAX(S)$：提取最大元素

$INCREASE-KEY(S,x,k)$:将元素x关键字值增加到k ,k值不能小于x原关键字值。

最大优先级队列的一个应用是在一台分时计算机上进行作业调度。

最小优先级队列支持的操作包括$INSERT$，$MINIMUM$，$EXTRACT-MIN$和$DECREASE-KEY$。这种队列可被用在基于事件驱动的模拟器中。

优先级队列可以用堆来实现。

```
HEAP-MAXIMUM(A)
1  return A[1]
```

```
HEAP-EXTRACT-MAX(A)
1  if heap-size[A] < 1
2      then error "heap underflow"
3  max ← A[1]
4  A[1] ← A[heap-size[A]]
5  heap-size[A] ← heap-size[A] − 1
6  MAX-HEAPIFY(A,1)
7  return max
```

```
HEAP-INCREASE-KEY(A, i, k)
1  if key < A[i]
2      then error "new key is smaller than current key"
3  A[i] ← key
4  while i > 1 and A[PARENT(i)] < A[i]
5        do exchange A[i] ↔ A[PARENT(i)]
6           i ← PARENT(i)
```

MAX-HEAP-INSERT(A, key)

1 $heap-size[A] \leftarrow heap-size[A]+1$
2 $A\left[heap-size[A]\right] \leftarrow -\infty$
3 $HEAP-INCREASE-KEY(A, heap-size[A], key)$