堆排序 Heapsort

堆

表示堆的 array A 是一个具有两个属性的对象：length[A]是 array 中元素个数，heap-size[A]是存放在 A 中的堆的元素个数。

堆与栈的区别

注意别于栈搞混，栈是一种先进后出的数据结构，而堆则可以被看成是一棵树。

堆与树的区别

堆是一种特殊的树,它每个结点都有一个值，堆的特点是根结点的值最小（或最大），且根结点的两个子树也是一个堆。就类似一堆东西一样，按照由大到小（或由小到大）“堆”起来。

在结构上：

二叉排序树：左子树小于根节点，根节点又小于右子树。

堆（小堆）：根节点小于左右子树，但是左右子树没有大小之分。

在作用上：

二叉排序树是用来做查找的。

而堆是用来做排序的。

堆的 5 个基本过程

- MAX-HEAPIFY 过程，运行时间为 O(lgn)，是保持最大堆性质的关键。

```
MAX-HEAPIFY(A, i)
l ← LEFT(i)
r ← RIGHT(i)
if l ≤ heap-size[A] and A[l] > A[i]
   then largest ← l
   else largest ← i
if r ≤ heap-size[A] and A[r] > A[largest]
   then largest ← r
if largest ≠ i
   then exchange A[i] ↔ A[largest]
        MAX-HEAPIFY(A, largest)
```

MAX-HEAPIFY 接收 array A 和下标 i，其中，i 为被指定的根。这一过程使子树最终全部符合最大堆的性质。花费的时间为 O(lgn)。若作用于一个高度为 h 的结点，其运行时间为 O(h)。

- BUILD-MAX-HEAP 过程，运行时间为 O(n)，可以在无序的输入数组的基础上构造出最大堆。

BUILD-MAX-HEAP，即建堆，过程对书中的每一个其他结点都调用一次 MAX-HEAPIFY。

```
BUILD-MAX-HEAP(A)
heap-size[A] ← length[A]
for i ← ⌊length[A]/2⌋ downto 1
     do MAX-HEAPIFY(A, i)
```

注：downto 1 意思是每次循环减 1，减少到 1 为止

i 从二分之一长度处开始，即，i 从第一个根出发，依次对剩下的全部根做 MAX-HEAPIFY 调用。每次调用都会让子树保持最大堆性质。直到所有调用结束，就得到了一个正常的最大堆。

可以看出，这一过程花费的时间为 O(n)。具体计算过程见算法导论中文版第 77 页。

- HEAPSORT 过程，运行时间为 O(nlgn), 对一个数组原地 in-place 进行排序。

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

首先对 array A 进行建堆。这时，array 的最大元素在根 A[1]处，通过将它与 A[n]互换即可让 A[n]的位置成为正确顺序的第一个元素。此时原堆由于 A[1]元素被置换而违背了最大堆性质，所以需要调用 MAX-HEAPIFY 来维持性质正确。维持完成后，A[1]位置的元素又成为了现有堆的最大元素，依此类推，最终得到一个成功排完序的 array。

- HEAP-MAXIMUM，返回最大值，即返回 A[1]即可。
- HEAP-EXTRACT-MAX，导出最大值并从原堆中移除该最大值。

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

首先，HEAP-EXTRACT-MAX 记录了最大值，接下来，函数将原堆的最后一位提前到第一位，即取代了原最大值。这时，堆中存在两个最后一位，所以，函数将 heapsize 缩减 1。而为了让现有的堆重新满足最大堆性质，函数进行了 MAX-HEAPIFY 过程。整个过程中的运行时间为 O(lgn)。

- HEAP-INCREASE-KEY，将 i 位置的值增加到 key，key 值不能小于原来的值。

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

由于值的增加，原有的堆性质被破坏，所以需要为 A[i]找到合适的位置。伪代码第 4 行判断 A[i]是否要与其父节点跟换位置。与树不同，在堆中无需为左右 child 大小担心，更换完子节点与父节点位置后即能保证子堆性质。这个过程的时间复杂度为 O(lgn)。

- MAX-HEAP-INSERT，将一个新的值插入到原有堆中

```
MAX-HEAP-INSERT(A, key)
heap-size[A] ← heap-size[A] + 1
A[heap-size[A]] ← -∞
HEAP-INCREASE-KEY(A, heap-size[A], key)
```

这个过程只需要 O(lgn)的时间即可完成！看到这里时，记得重新浏览 HEAP-INCREASE-KEY 过程。