HeapSort

Heap

> **定义 7.1** 堆：一棵二叉树满足”堆结构特性“和”堆偏序特性“，则称它为一个堆
>
> - 堆结构特性：一颗二叉树要么是完美的，要么仅比一颗完美二叉树在最底层少若干节点，且最底层的结点从左向右紧挨着依次排列
> - 堆偏序特性：堆结点中存储的元素满足父结点的值大于所有子结点的值

堆的具体实现：数组

对于一个大小为 n 的堆，需要一个大小为 n 的数组（下标为 $1, 2, \ldots, n$），将堆中元素按深度从小到大，同一深度的元素从左到右，依次放入数组中。如此实现的堆中父子结点下标满足

- $\text{PARENT}(i) = \left\lfloor \frac{i}{2} \right\rfloor$
- $\text{LEFT}(i) = 2i$
- $\text{RIGHT}(i) = 2i + 1$

Fix-Heap

从堆顶取出一个元素后，修复其使其重新成为一个堆

- 结构特性修复：取出底层最右边的元素放在堆顶的位置
- 偏序特性修复：
    - 将父结点值与两个子结点值比较，假设左结点是其中值最大的，交换父结点和左结点的值
    - 递归地对左子树进行修复

每次修复比较两次，修复次数不会超过树的高度，故修复的最坏情况代价为 $O(\log n)$

基于数组实现可得到算法 FIX-HEAP (A, p):

```
l := LEFT(p);
r := RIGHT(p);
next := p;
if l <= heapSize and A[l] > A[p] then
    next := l;
if r <= heapSize and A[r] > A[next] then
    next := r;
if next != p then
    SWAP(A[p], A[next]);
    FIX-HEAP(A, next);
```

Construct-Heap

基于数组的堆实现已经完成了结构特性，故仅讨论偏序特性的构建，思想为：

- 从根开始构建，根的左右子树已经是堆，若根不符合偏序特性，则进行一次修复即可
- 递归地将左右子树构建为堆

堆的构建为线性时间

> Smart Guess:
>
> $$W(n) = 2W\left(\frac{n}{2}\right) + 2\log n$$
>
> 根据 Master Theorem
>
> $$W(n) = \Theta(n)$$

假设堆是完美二叉树，树中高度为 h 的结点个数为 $\left\lceil \frac{n}{2^{h+1}} \right\rceil$，所有结点的修复代价总和为

$$\begin{aligned} W(n) &= \sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) \\ &= O\left(n \sum_{h=1}^{\lfloor \log n \rfloor} \frac{h}{2^h}\right) \\ &= O(n) \end{aligned}$$

故堆的构建最坏情况时间复杂度为 $O(n)$

可基于数组实现得到算法 CONSTRUCT-HEAP (A[1...n]):

```
heapSize := n;
BUILD(1)
subroutine BUILD(p) begin
    l := LEFT(p);
    r := RIGHT(p);
    if l <= heapSize then
        BUILD(l);
    if r <= heapSize then
        BUILD(r);
    FIX-HEAP(A, p);
```

Understanding Heap

k^{th} element

寻找堆中第 k 大的元素，设堆大小为 n 且 $k \ll n$，要求算法代价只与 k 有关

Sum of heights

堆的所有结点的高度之和为 $n-1$，证明使用数学归纳法

HeapSort

Strategy

将输入的元素建立堆，取出堆顶的元素与下标为 n 的元素交换，然后将前 $n-1$ 个元素修复为堆

Algorithm

HEAP-SORT (A[1...n]):

```
CONSTRUCT-HEAP(A[1...n]);
for i := n downto 2 do
    SWAP(A[1], A[i]);
    heapSise := heapSize - 1;
    FIX-HEAP(A, 1)
```

Analysis

Worst-case

显然

$$W(n) = W_{cons}(n) + \sum_{k=1}^{n-1} W_{fix}(k)$$

而由之前的结论

$$W_{cons}(n) = \Theta(n) \text{ and } W_{fix}(k) \leqslant 2\log k$$

而

$$2\sum_{k=1}^{n-1}\lceil \log k \rceil \leqslant 2\int_1^n \log e \ln x \mathrm{d}x = 2\log e(n\ln n - n) = {\color{red}2(n\log n - 1.443n)}$$

所以

$$W(n) \leqslant 2n\log n + \Theta(n)$$
$$W(n) = \Theta(n\log n)$$

最坏情况时间复杂度为 $\Theta(n\log n)$

Average-case

平均情况时间复杂度同样为 $\Theta(n\log n)$

Accelerated HeapSort

减少修复堆时的比较次数:

- Bubbling Up，即由下向上修复堆
- 修复时仅比较一次，然后再上浮调整
- 使用 Divide & Conquer：即每次向下调整 $\frac{1}{2}$ ，若是超过了则向上浮动调整，否则继续向下调整 $\frac{1}{2}$

相关算法:

Bubble-Up Heap Algorithm:

```
void bubbleUpHeap(Element E[], int root, Element k, int vacant){
    if(vacant == root){
        E[vacant] = k;
    } else {
        int parent = vacant / 2;
        if (K.key <= E[parent].key){
            E[vacant] = K;
        } else {
            E[vacant] = E[parent];
            bubbleUpHeap(E, root, K, parent);
        }
    }
}
```

Depth Bounded Filtering Down:

```
int promote(Element E[], int hStop, int vacant, int h){
    int vacStop;
    if(h <= hStop){
        vacStop = vacant;
    } else if (E[2 * vacant].key <= E[2 * vacant +1].key){
        E[vacant] = E[2 * vacant + 1];
        vacStop = promote(E, hStop, 2 * vacant + 1, h - 1);
    } else {
        E[vacant] = E[2 * vacant];
        vacStop = promote(E, hStop, 2 * vacant, h - 1);
    }
    return vacStop;
}
```

Fix-Heap Using Divide-and-Conquer:

```
void fixHeapFast(Element E[], Element K, int vacant, int h){
    if(h <= 1){
        // Process heap of height 0 or 1
    } else {
        int hStop = h / 2;
        int vacStop = promote(E, hStop, vacant, h);
        int vacParent = vacStop / 2;
        if(E[vacParent].key <= K.key){
            E[vacStop] = E[vacParent];
```

```
                bubbleUpHeap(E, vacant, K, vacParent);
            } else {
                fixHeapFast(E, K, vacStop, hStop);
            }
        }
}
```

一次调整最多调用 t 次 promote 和 1 次 bubbleUpHaep，比较次数为

$$\sum_{k=1}^{t}\left\lceil \frac{h}{2^k} \right\rceil + \left\lceil \frac{h}{2^t} \right\rceil = h = \log(n+1)$$

且要执行 $\log h$ 次检查是否需要继续调整的比较

对于 Accelerated HeapSort

$$W(n) = n\log n + \Theta(n\log\log n)$$

More than Sorting

寻找第 k 大元素

寻找前 k 大元素

合并 k 个排好序的列表

动态中位数