hw4

> 6.2-5; 6.3-1, 6.3-3; 6.4-4; 6.5-2, 6.5-8;

6.2-5

6.2-5 MAX-HEAPIFY 的代码效率较高，但第 10 行中的递归调用可能例外，它可能使某些编译器产生低效的代码。请用循环控制结构取代递归，重写 MAX-HEAPIFY 代码。

```
Iterative Max Heapify(A, i)
    while i < A.heap-size do
        l = LEFT(i)
        r = RIGHT(i)
        largest = i

        if l ≤ A.heap-size and A[l] > A[i] then
            largest = l
        end if

        if r ≤ A.heap-size and A[r] > A[largest] then
            largest = r
        end if

        if largest ≠ i then
            exchange A[i] and A[largest]
        else
            return A
        end if
    end while
    return A
```

6.3-1

6.3-1 参照图 6-3 的方法，说明 BUILD-MAX-HEAP 在数组 $A=\langle 5, 3, 17, 10, 84, 19, 6, 22, 9\rangle$上的操作过程。

5	3	17	10	84	19	6	22	9
5	3	17	22	84	19	6	10	9
5	3	19	22	84	17	6	10	9

5	3	17	10	84	19	6	22	9
5	84	19	22	3	17	6	10	9
84	5	19	22	3	17	6	10	9
84	22	19	5	3	17	6	10	9
84	22	19	10	3	17	6	5	9

6.3-3

6.3-3 证明：对于任一包含 n 个元素的堆中，至多有$\lceil n/2^{h+1} \rceil$个高度为 h 的结点?

高度为 h 的所有节点将叶子节点集合分成大小在 $2^{h-1}+1$ 和 2^h 之间的子集，其中除一个子集外，其大小都为 2^h。这是通过将每个节点的子节点放入分区的不同部分来实现的。6.1-2中提到的堆的高度是 $\lfloor \lg(n) \rfloor$，因此通过查看这个高度的一个元素（根节点），得到高度为 $\lfloor \lg(n) \rfloor$ 的叶子节点最多有 $2^{\lfloor \lg(n) \rfloor}$ 个。由于高度为 h 的每个节点将其分成至少大小为 $2^{h-1}+1$ 的部分，并且除一个外，其余部分都对应于大小为 2^h 的部分，可以将 k 表示为我们要限制的数量，因此，

$$(k-1)2^h + k\left(2^{h-1}+1\right) \leq 2^{\lfloor \lg(n) \rfloor} \leq n/2$$

所以

$$k \leq \frac{n+2^h}{2^{h+1}+2^h+1} \leq \frac{n}{2^{h+1}} \leq \left\lceil \frac{n}{2^{h+1}} \right\rceil$$

6.4-4

6.4-4 证明：在最坏情况下，HEAPSORT 的时间复杂度是 $\Omega(n \lg n)$。

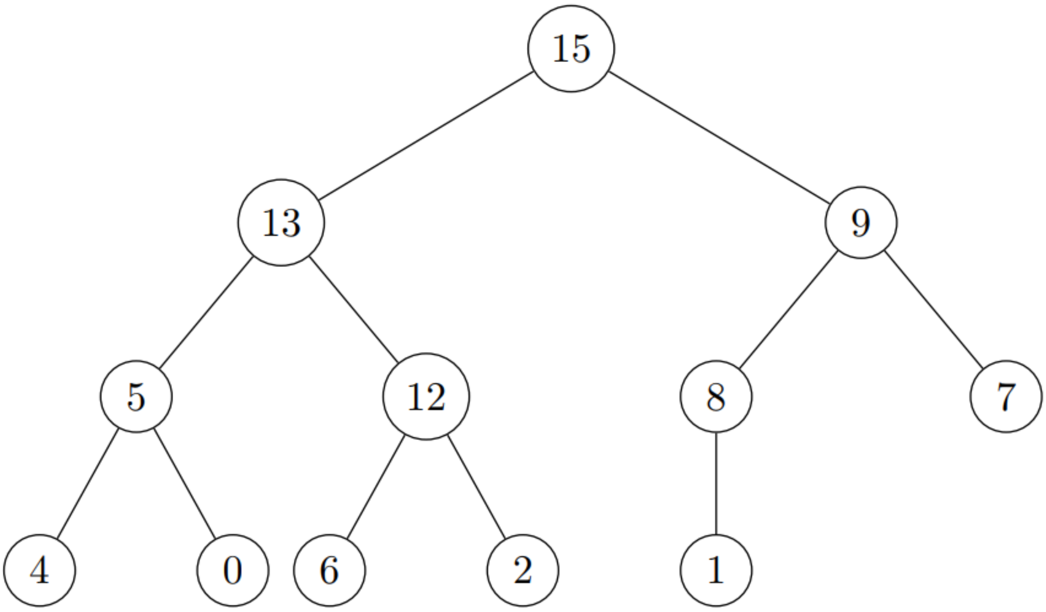
考虑在一个按递减顺序排列的数组上调用 HEAPSORT。每次 $A[1]$ 与 $A[i]$ 交换时，MAX-HEAPIFY 将递归调用一定次数，次数等于包含位置 1 到 $i-1$ 的元素的最大堆的高度 h，并且其运行时间为 $O(h)$。由于在高度 k 上有 2^k 个节点，因此运行时间的下限为

$$\sum_{i=1}^{\lfloor \lg n \rfloor} 2^i \log\left(2^i\right) = \sum_{i=1}^{\lfloor \lg n \rfloor} i2^i = 2 + (\lfloor \lg n \rfloor - 1)2^{\lfloor \lg n \rfloor} = \Omega(n \lg n).$$

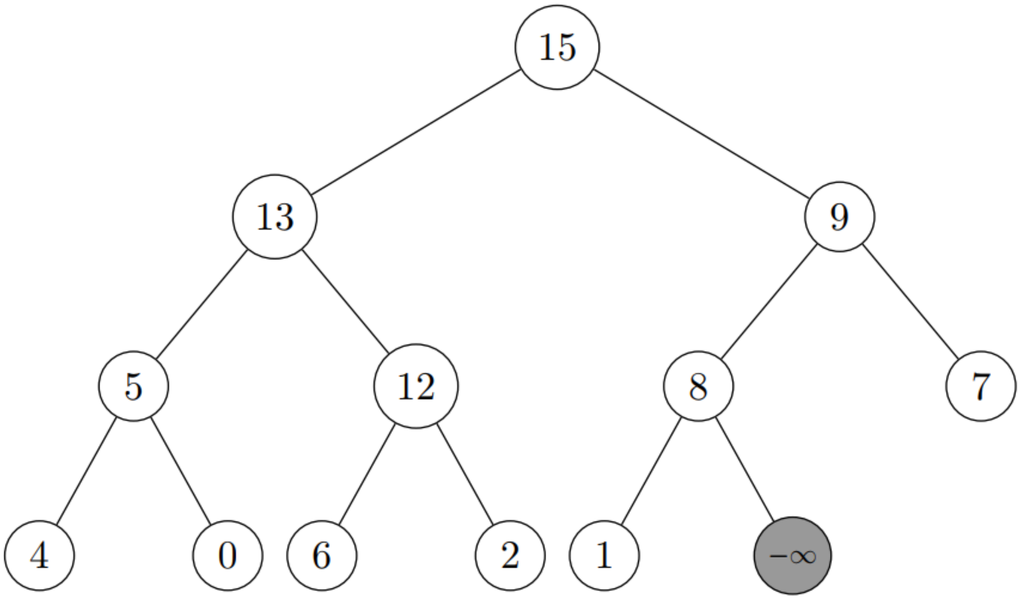
6.5-2

6.5-2 试说明 MAX-HEAP-INSERT(A, 10)在堆 $A=\langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$ 上的操作过程。

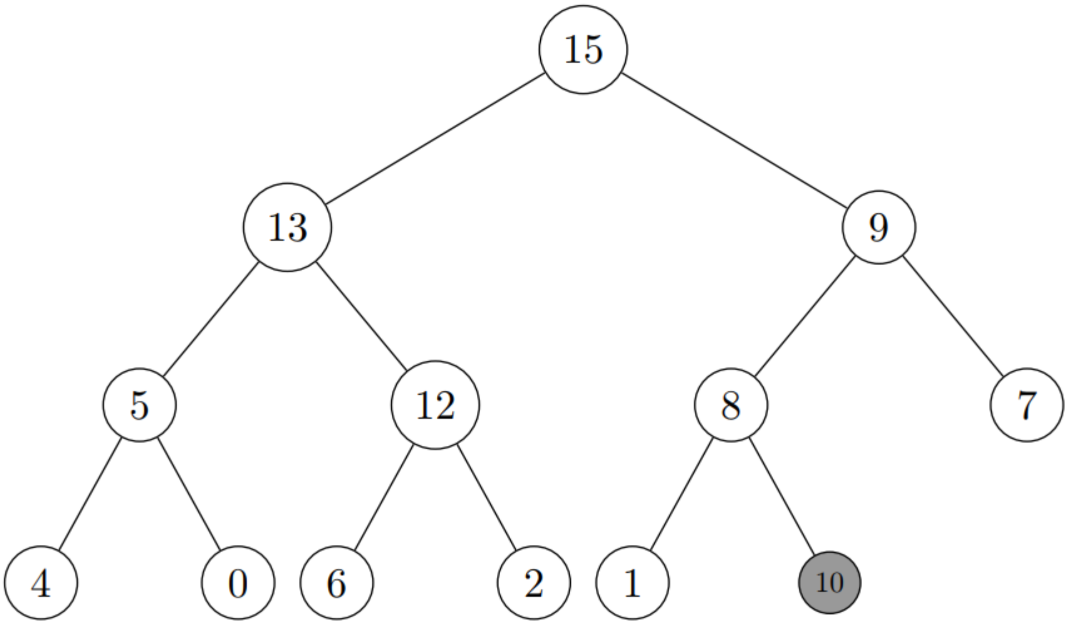
1. 初始堆



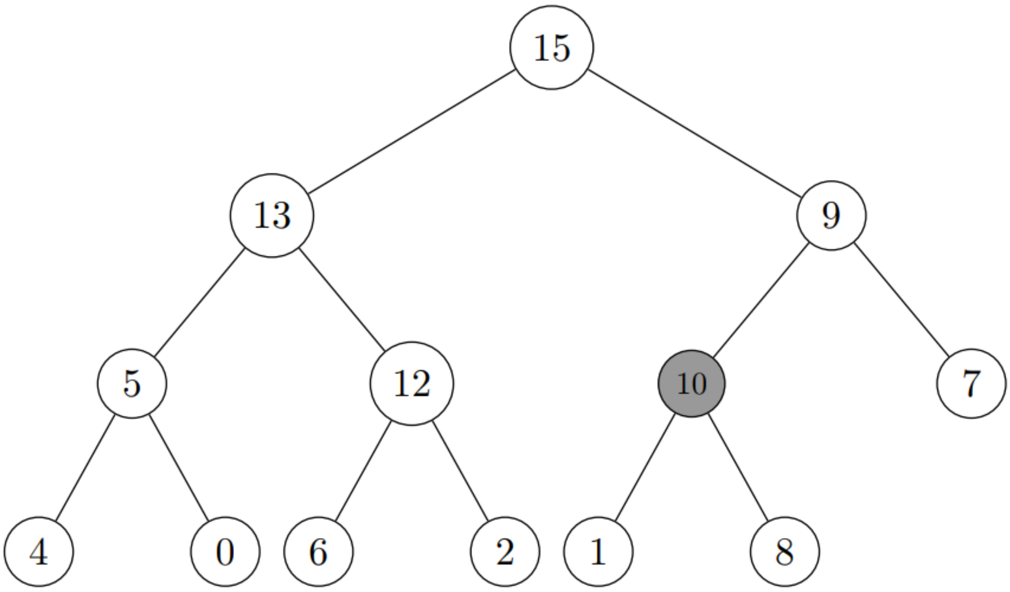
2. 调用 `MAX-HEAP-INSERT(A,10)`，因此先追加一个赋值为 -1 的节点：



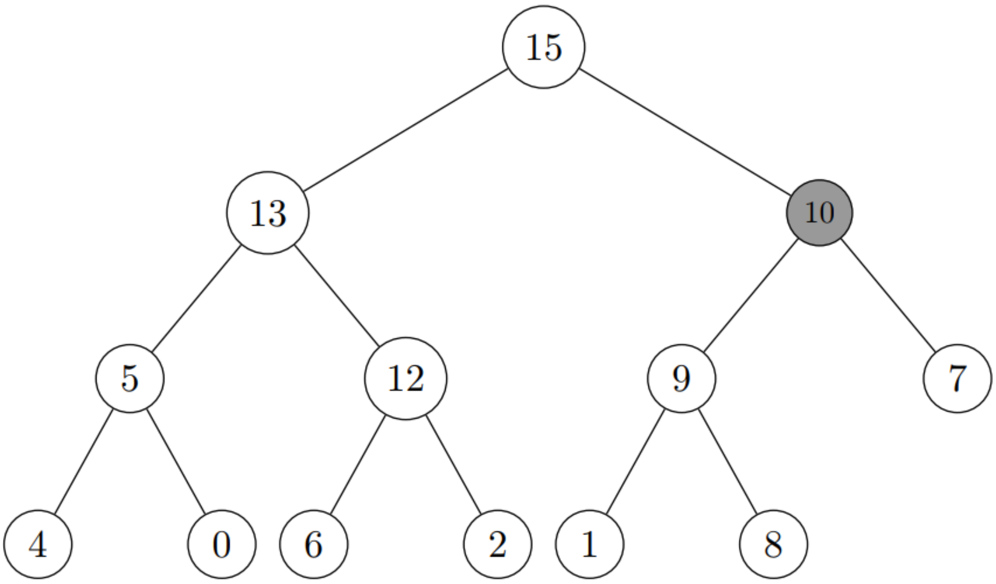
3. 更新新节点的 key value:



4. 由于父节点的键值小于10，节点被交换:



5. 由于父节点的键值小于10，节点被交换：



6.5-8

6.5-8 HEAP-DELETE(A, i)操作能够将结点 i 从堆 A 中删除。对于一个包含 n 个元素的堆，请设计一个能够在 $O(\lg n)$时间内完成的 HEAP-DELETE 操作。

首先，将要删除的节点的键替换为无穷大(∞)，即一个将被解释为大于所有存储在最大堆中的其他键的值。调用 HEAP-INCREASE-KEY 将该节点上浮到最大堆的顶部。然后，将根节点的值替换为堆中的最后一个元素的值，由于最大堆属性，我们知道该值小于 $A[1]$。更新堆的大小，然后调用 MAX-HEAPIFY 来恢复最大堆属性。这个算法的运行时间为 $O(\lg n)$，因为 INCREASE-KEY 运行时间为 $O(\lg n)$，而 MAX-HEAPIFY 递归调用的次数最多等于堆的高度，即 $\lfloor \lg n \rfloor$。

算法的伪代码：

```
Algorithm HEAP-DELETE(A, i)
1: HEAP-INCREASE-KEY(A, i, ∞)
2: A[1] = A[A.heap-size]
3: A.heap-size = A.heap-size - 1
4: MAX-HEAPIFY(A, 1)
```