Section 6.1 – Heaps

6.1-1 What are the minimum and maximum numbers of elements in a heap of height h?

> Minimum is 2^h. Maximum is $2^{h+1} - 1$.

6.1-2 Show that an n-element heap has height $\lfloor \lg n \rfloor$.

> A heap of height $h + 1$ is a complete tree of height h plus one additional level with $1 \leq k \leq 2^h$ nodes. This additional level does not count to the height of the heap, which then explain the height of $\lfloor \lg n \rfloor$.

6.1-3 Show that in any subtree of a max-heap, the root of the subtree contains the largest value occurring anywhere in that subtree.

> Every node of the subtree has a path upwards to the root of the subtree. Therefore, the max-heap property assures that each of these nodes are no larger than the root of the subtree.

6.1-4 Where in a max-heap might the smallest element reside, assuming that all elements are distinct?

> In the leaves. Note that, since the bottom level may be incomplete, in addition to the nodes on level zero, some of the nodes on level one may also be leaves.

6.1-5 Is an array that is in sorted order a min-heap?

> Yes, since for each node i, we have $A[\text{PARENT}(i)] \leq A[i]$.

6.1-6 Is the array with values $\langle 23, 17, 14, 6, 13, 10, 1, 5, 7, 12 \rangle$ a max-heap?

> No. The element 6 is the parent of the element 7 and $6 < 7$, which violates the min-heap property.

6.1-7 Show that, with the array representation for storing an n-element heap, the leaves are the nodes indexed by $\lfloor n/2 \rfloor + 1, \lceil n/2 \rceil + 2, \ldots, n$.

> The parent of the last element of the array is the element at position $\lfloor n/2 \rfloor$, which implies that all elements after $\lfloor n/2 \rfloor$ has no children and are therefore leaves. Also, since the element at position $\lfloor n/2 \rfloor$ has at least one child (the element at position n), the elements before $\lfloor n/2 \rfloor$ also have and therefore can not be leaves.

Section 6.2 – Maintaining the heap property

6.2-1 Using Figure 6.2 as a model, illustrate the operation of MAX-HEAPIFY$(A, 3)$ on the array $A = \langle 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0 \rangle$.
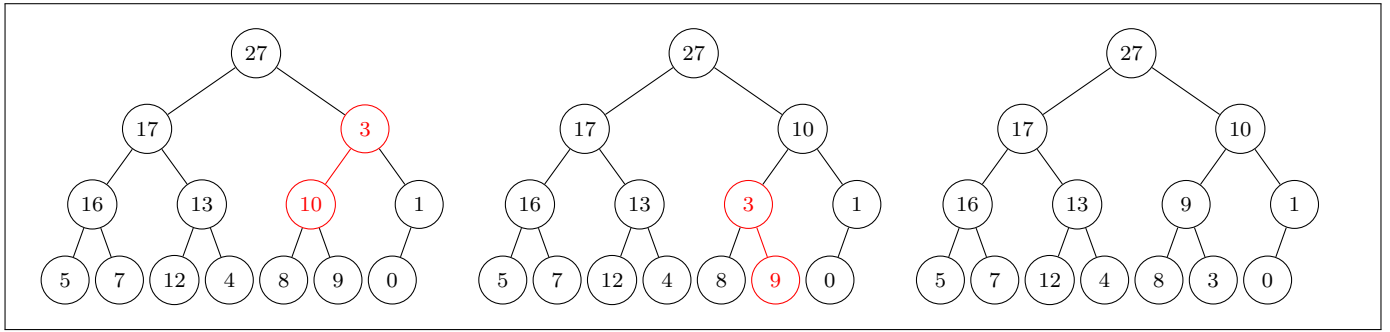
6.2-2 Starting with the procedure MAX-HEAPIFY, write pseudocode for the procedure MIN-HEAPIFY(A, i), which performs the corresponding manipulation on a min-heap. How does the running time of MIN-HEAPIFY compare to that of MAX-HEAPIFY?

> The pseudocode is stated below.
>
> ```
> Min-Heapify(A, i)
> ```
> 1 $l = \text{Left}(i)$
> 2 $r = \text{Right}(i)$
> 3   **if** $l \leq A.\text{heap-size}$ and $A[l] < A[i]$ **then**
> 4 $smallest = l$
> 5   **else**
> 6 $smallest = i$
> 7   **if** $r \leq A.\text{heap-size}$ and $A[r] < A[smallest]$ **then**
> 8 $smallest = r$
> 9   **if** smallest $\neq i$ **then**
> 10   exchange $A[i]$ with $A[smallest]$
> 11   Min-Heapify$(A, smallest)$
>
> The running time is the same.

6.2-3 What is the effect of calling MAX-HEAPIFY(A, i) when the element $A[i]$ is larger than its children?

> Node i and its children already satisfies the max-heap property. No recursion will be called and the array will keep the same.

6.2-4 What is the effect of calling MAX-HEAPIFY(A, i) for $i > A.heap\text{-}size/2$?

> Every node $i > A.heap\text{-}size/2$ is a leaf. No recursion will be called and the array will keep the same.

6.2-5 The code for MAX-HEAPIFY is quite efficient in terms of constant factors, except possibly for the recursive call in line 10, which might cause some compilers to produce inefficient code. Write an efficient MAX-HEAPIFY that uses an iterative control construct (a loop) instead of recursion.

The pseudocode is stated below.

Max-Heapify-Iterative(A, i)
```
 1    solved = False
 2    current-node = i
 3    while not solved do
 4        l = Left(current-node)
 5        r = Right(current-node)
 6        if l ≤ A.heap-size and A[l] > A[current-node] then
 7            largest = l
 8        else
 9            largest = current-node
10        if r ≤ A.heap-size and A[r] > A[largest] then
11            largest = r
12        if largest ≠ current-node then
13            exchange A[current-node] with A[largest]
14            current-node = largest
15        else
16            solved = True
```

6.2-6 Show that the worst-case running time of MAX-HEAPIFY on a heap of size n is $\Omega(\lg n)$. (Hint: For a heap with n nodes, give node values that cause MAX-HEAPIFY to be called recursively at every node on a simple path from the root down to a leaf.)

The worst-case occurs when $A[\text{LEFT}(i)] \geq A[\text{RIGHT}(i)] > A[i]$ in each level of the recursion, which will cause the node to be pushed to the leftmost position on the bottom level of the heap. There will be exactly $\lfloor \lg n \rfloor$ recursive calls (in addition to the first call). Since each call is $\Theta(1)$, the total running time is $\lfloor \lg n \rfloor \cdot \Theta(1) = \Theta(\lg n) = \Omega(\lg n)$.