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 $A[\textsc{Parent}(i)] \leq A[i]$.

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

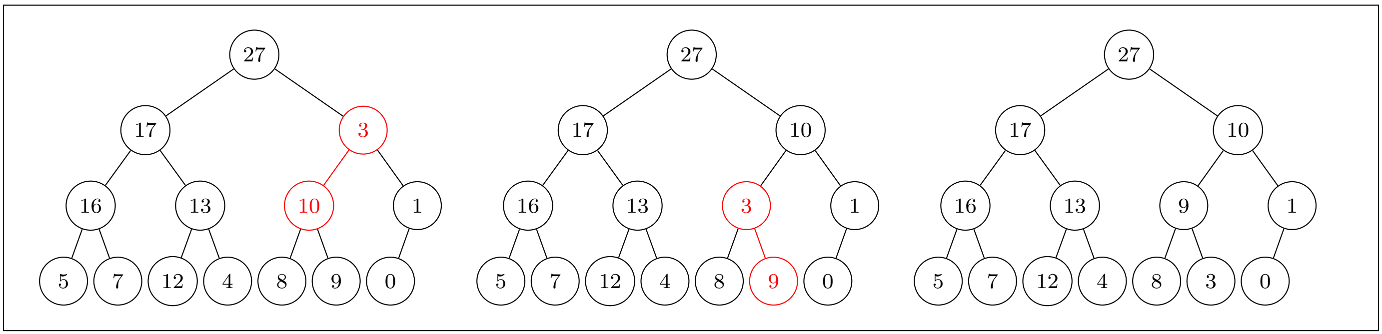
> No. The element 6 is the parent of the element 7 and $6 < 7$, which violates the min-heap property.

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)
    l = Left(i)
    r = Right(i)
    if l ≤ A.heap-size and A[l] < A[i] then
        smallest = l
    else
        smallest = i
    if r ≤ A.heap-size and A[r] < A[smallest] then
        smallest = r
    if smallest ≠ i then
        exchange A[i] with A[smallest]
        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 $i > A.\textit{heap-size}/2$?

Every node $i > A.\textit{heap-size}/2$ 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.

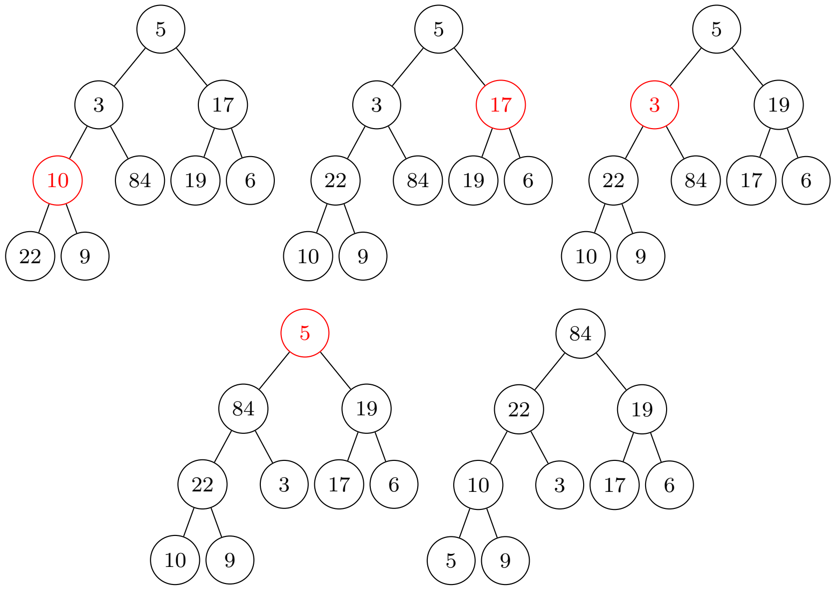
```
Max-Heapify-Iterative(A, i)
    solved = False
    current-node = i
    while not solved do
        l = Left(current-node)
        r = Right(current-node)
        if l ≤ A.heap-size and A[l] > A[current-node] then
            largest = l
        else
            largest = current-node
        if r ≤ A.heap-size and A[r] > A[largest] then
            largest = r
        if largest ≠ current-node then
            exchange A[current-node] with A[largest]
            current-node = largest
        else
            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 $\lfloor \lg n \rfloor \cdot \Theta(1) = \Theta(\lg n) = \Omega(\lg n)$.

Section 6.3 – Building a heap

6.3-1 Using Figure 6.3 as a model, illustrate the operation of BUILD-MAX-HEAP on the array $A = \langle 5, 3, 17, 10, 84, 19, 6, 22, 9 \rangle$.



6.3-2 Why do we want the loop index i in line 2 of BUILD-MAX-HEAP to decrease from $\lfloor A.length/2 \rfloor$ to 1 rather than increase from 1 to $\lfloor A.length/2 \rfloor$?

When we use MAX-HEAPIFY in a bottom-up manner, before each call to MAX-HEAPIFY(A, i), we can be sure that the subtrees rooted on its children are max-heaps and thus after exchanging $A[i]$ with $\max(A[\text{LEFT}(i)], A[\text{RIGHT}(i)])$, $A[i]$ will be the largest node among the nodes of the subtree rooted at i. In contrast, when we use MAX-HEAPIFY in a top-down manner, we can not be sure of that. For instance, if in a call to MAX-HEAPIFY(i), LEFT$(i) >$ RIGHT(i) and the largest node of the subtree rooted on i is on the subtree rooted on RIGHT(i), this largest element will never reach the position i, which will then violate the max-heap property.

6.3-3 Show that there are at most $\lceil n/2^{h+1} \rceil$ nodes of heigh h in any n-element heap.

From 6.1-7, we know that the leaves of a heap are the nodes indexed by

$$\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n.$$

Note that those elements corresponds to the second half of the heap array (plus the middle element if n is odd). Thus, the number of leaves in any heap of size n is $\lceil n/2 \rceil$. Lets prove by induction. Let n_h denote the number of nodes at height h. The upper bound holds for the base since $n_0 = \lceil n/2^{0+1} \rceil = \lceil n/2 \rceil$ is exactly the number of leaves in a heap of size n. Now assume is holds for $h-1$. We shall prove that it also holds for h. Note that if n_{h-1} is even each node at height h has exactly two children, which implies $n_h = n_{h-1}/2 = \lceil n_{h-1}/2 \rceil$. If n_{h-1} is odd, one node at height h has one child and the remaining has two children, which also implies $n_h = \lfloor n_{h-1}/2 \rfloor + 1 = \lceil n_{h-1}/2 \rceil$. Thus,

$$n_h = \left\lceil \frac{n_{h-1}}{2} \right\rceil \leq \left\lceil \frac{1}{2} \cdot \left\lceil \frac{n}{2^{(h-1)+1}} \right\rceil \right\rceil = \left\lceil \frac{1}{2} \cdot \left\lceil \frac{n}{2^h} \right\rceil \right\rceil = \left\lceil \frac{n}{2^{h+1}} \right\rceil,$$

which shows that it also holds for h.