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?

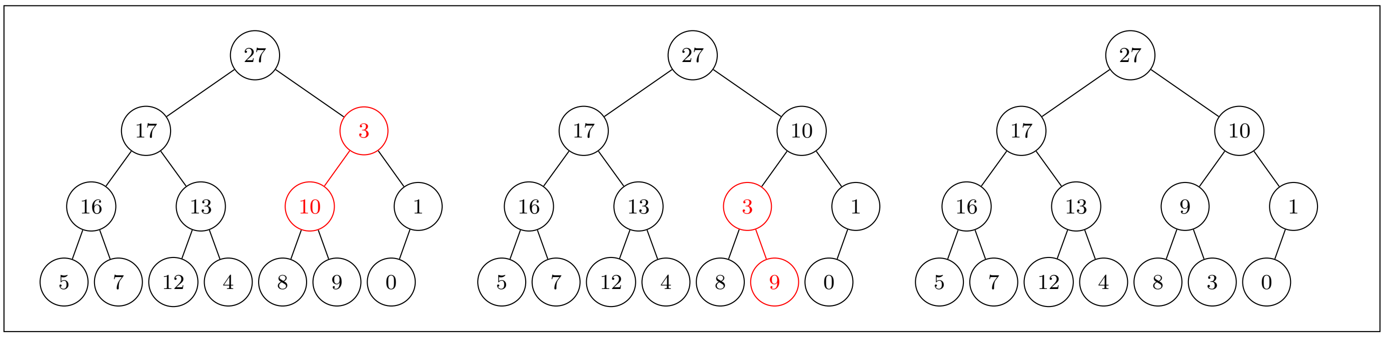
No. The element 6 is the parent of the element 7 and $6 < 7$, which violates the min-heap property.

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 $l \leq A$.heap-size and $A[l] < A[i]$ then
        $smallest = l$
    else
        $smallest = i$
    if $r \leq A$.heap-size and $A[r] < A[smallest]$ then
        $smallest = r$
    if smallest $\neq i$ 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 $i > A.heap\text{-}size/2$?

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.

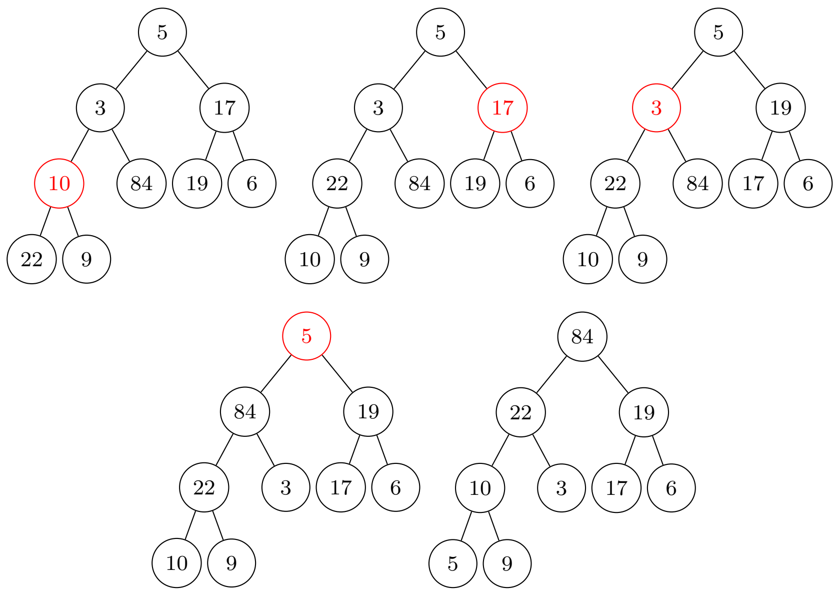
```
Max-Heapify-Iterative($A$, $i$)
    $solved$ = False
    $current\text{-}node = i$
    while not $solved$ do
        $l$ = Left($current\text{-}node$)
        $r$ = Right($current\text{-}node$)
        if $l \leq A.heap\text{-}size$ and $A[l] > A[current\text{-}node]$ then
            $largest = l$
        else
            $largest = current\text{-}node$
        if $r \leq A.heap\text{-}size$ and $A[r] > A[largest]$ then
            $largest = r$
        if $largest \neq current\text{-}node$ then
            exchange $A[current\text{-}node]$ with $A[largest]$
            $current\text{-}node = largest$
        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 $\lfloor A.\mathit{length}/2 \rfloor$ to 1 rather than increase from 1 to $\lfloor A.\mathit{length}/2 \rfloor$?

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

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

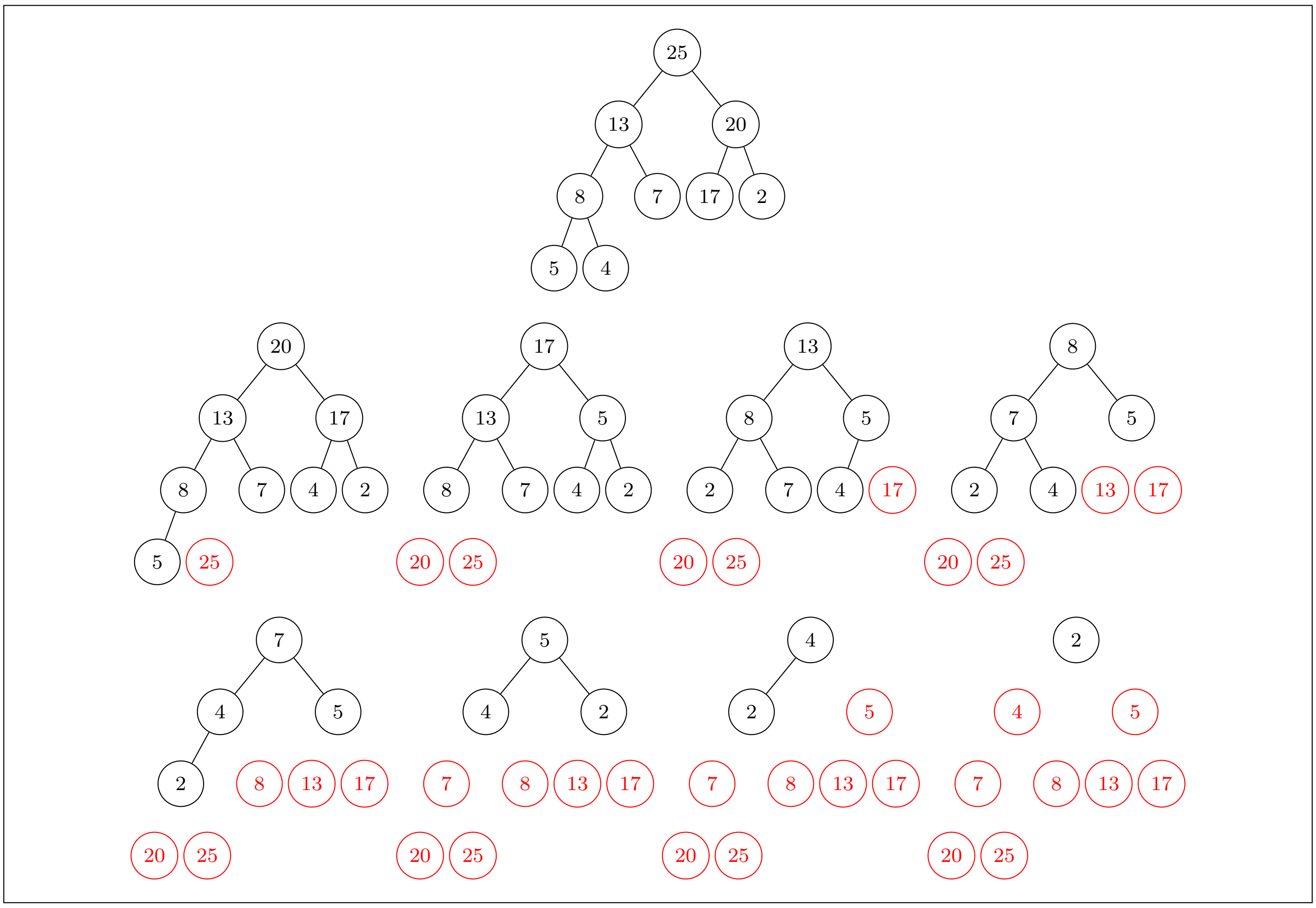
Note that those elements corresponds to the second half of the heap array (plus the middle element if n is odd). Thus, the number of leaves in any heap of size n is $\lceil n/2 \rceil$. Lets prove by induction. Let n_h denote the number of nodes at height h. The upper bound holds for the base since $n_0 = \lceil n/2^{0+1} \rceil = \lceil n/2 \rceil$ 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, we have

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

which shows that it also holds for h.

Section 6.4 – The heapsort algorithm

6.4-1 Using Figure 6.4 as a model, illustrate the operation of HEAPSORT on the array $A = \langle 5, 13, 2, 25, 7, 17, 20, 8, 4 \rangle$.



6.4-2 Argue the correctness of HEAPSORT using the following loop invariant:

> At the start of each iteration of the **for** loop of line 2–5, the subarray $A[1 \ldots i]$ is a max-heap containing the i smallest elements of $A[1 \ldots n]$, and the subarray $A[i+1 \ldots n]$ contains the $n - i$ largest elements of $A[1 \ldots n]$, sorted.

We need to show that this invariant is true prior to the first loop iteration, that each iteration of the loop maintains the invariant, and that the invariant provides a useful property to show correctness when the loop terminates.

- **Initialization.** Before the **for** loop, $i = n$ and line 1 assures A is a max-heap. Thus, $A[1, \ldots, i] = A$ is a max-heap containing the i smallest elements of A and $A[i+1, \ldots, n] = \emptyset$ contains the $n - i = 0$ largest elements of A, sorted.
- **Maintenance.** By the loop invariant, $A[1, \ldots, i]$ is a max-heap containing the i smallest elements of A, which implies that $A[1]$ is the ith smallest element of A. Since $A[i+1, \ldots, n]$ already contains the $n - i$ largest elements of A in sorted order, after exchanging $A[1]$ with $A[i]$, the subarray $A[i, \ldots, n]$ now contains the $n - i + 1$ largest elements of A in sorted order. Lines 4-5 maintains the max-heap property on the subarray $A[1, \ldots, i-1]$ and decrement i for the next iteration preserves the loop invariant.
- **Termination.** At termination $i = 1$ and the subarray $A[2, \ldots n]$ contains the $n - 1$ smallest elements of A in sorted order, which also implies that $A[1, \ldots, n]$ is fully sorted.

6.4-3 What is the running time of HEAPSORT on an array A of length n that is already sorted in increasing order? What about decreasing order?

Since an array that is sorted in increasing order is not a max-heap, BUILD-MAX-HEAP will break that ordering. The BUILD-MAX-HEAP procedure will take $\Theta(n)$ to build the max-heap and the **for** loop of lines 2–5 will take $O(n \lg n)$, which gives a total running time of $\Theta(n) + O(n \lg n) = O(n \lg n)$.

An array sorted in drecreasing order is already a max-heap, but even in that case BUILD-MAX-HEAP will take $\Theta(n)$. Note that, although the input is sorted in decreasing order, the intent of the algorithm is to sort in increasing order. In each iteration of the **for** loop of lines 2–5, it will exchange $A[1]$ with $A[i]$ and will call MAX-HEAPIFY on $A[1]$. Since $A[1]$ is not anymore the largest element of A, each call to MAX-HEAPIFY may cover the entire height of the heap and thus will take $O(\lg n)$. Thus, the **for** loop of lines 2–5 will run in $O(n \lg n)$ and the algorithm will take $\Theta(n) + O(n \lg n) = O(n \lg n)$.

6.4-4 Show that the worst-case running time of HEAPSORT is $\Omega(n \lg n)$.

The worst-case is when every call to MAX-HEAPIFY covers the entire height of the heap. In that case, HEAPSORT will take

$$\sum_{i=1}^{n-1} \lfloor \lg i \rfloor \leq \sum_{i=1}^{n-1} \lg i = \lg((n-1)!) = \Theta((n-1)\lg(n-1)) = \Omega(n \lg n).$$

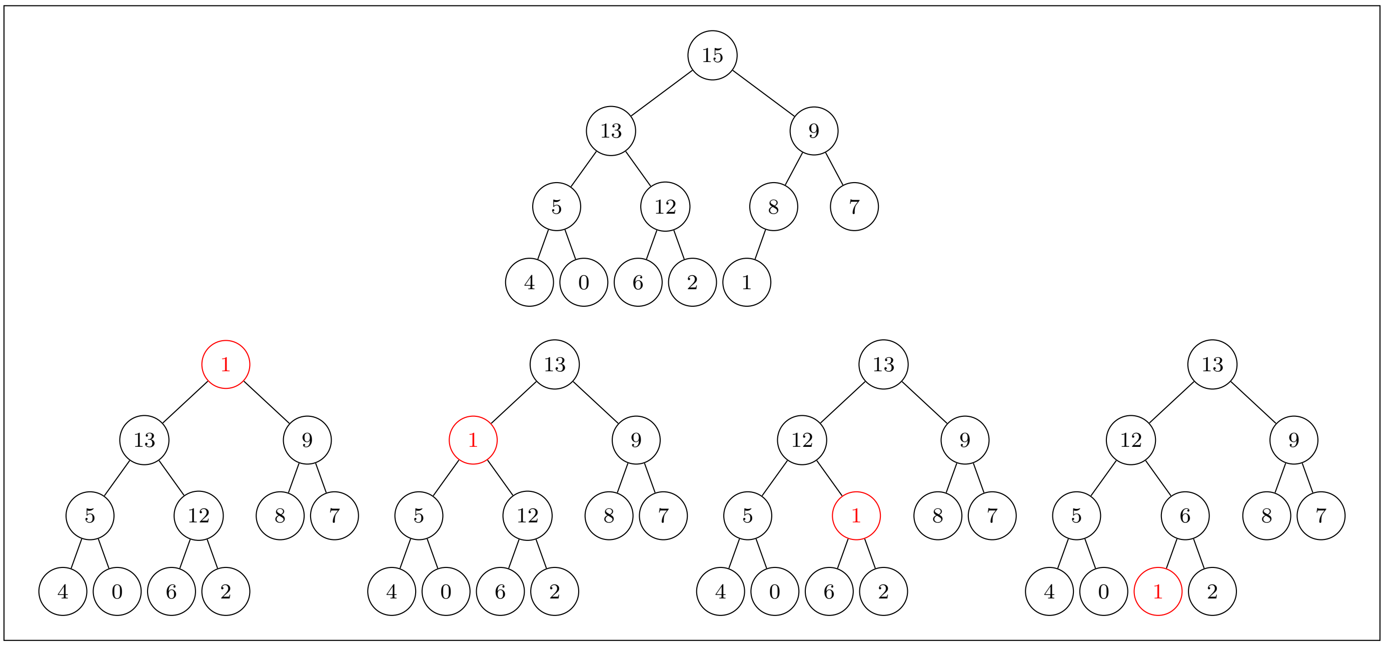
6.4-5 ($\star$) Show that when all elements are distinct, the best-case running time of HEAPSORT is $\Omega(n \lg n)$.

Proof on (Theorem 1, Page 86):

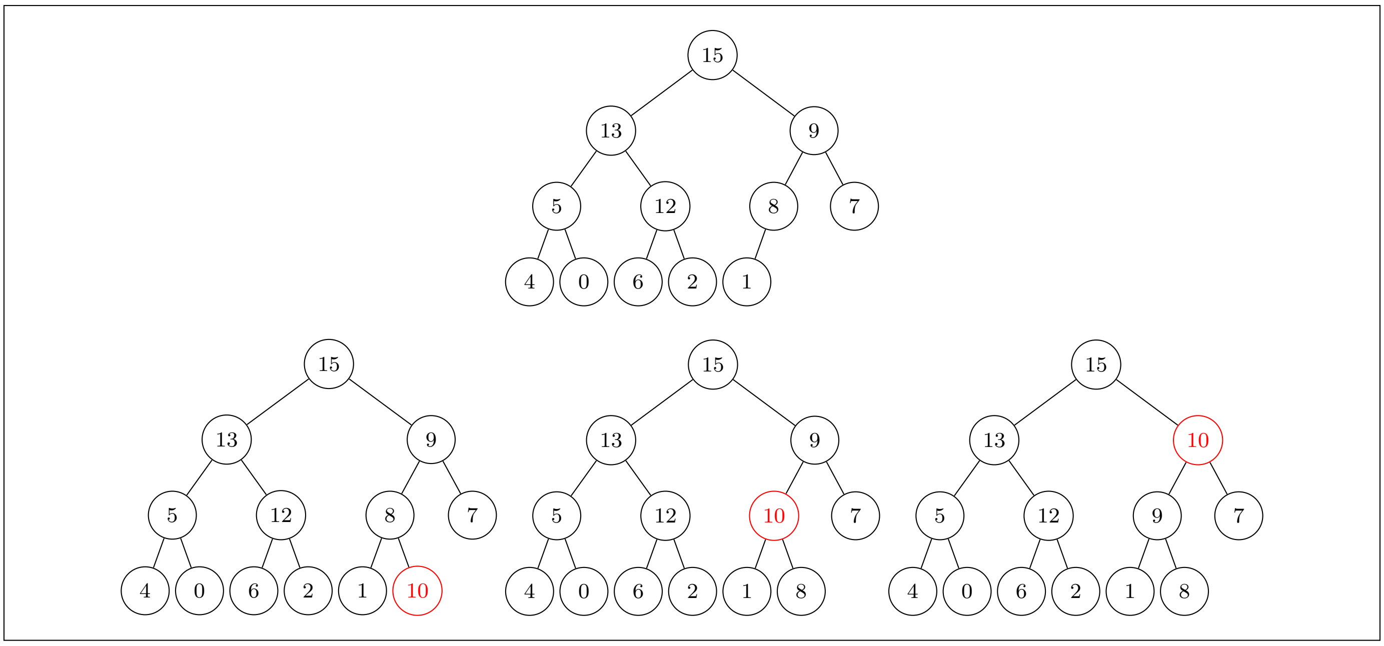
Schaffer, Russel, and Robert Sedgewick. "The analysis of heapsort." *Journal of Algorithms* 15.1 (1993): 76-100.

Section 6.5 – Priority queues

6.5-1 Illustrate the operation of Heap-Extract-Max on the heap $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$.



6.5-2 Illustrate the operation of Max-Heap-Insert$(A, 10)$ on the heap $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$.



6.5-3 Write pseudocode for the procedures HEAP-MINIMUM, HEAP-EXTRACT-MIN, HEAP-DECREASE-KEY, and MIN-HEAP-INSERT that implement a min-priority queue with a min-heap.

The pseudocodes are stated below.

```
Heap-Minimum($A$)
    return $A[1]$

Heap-Extract-Min($A$)
    if $A.heap\text{-}size < 1$ then
        error "heap underflow"
    $min = A[1]$
    $A[1] = A[A.heap\text{-}size]$
    $A.heap\text{-}size = A.heap\text{-}size - 1$
    Min-Heapify($A, 1$)
    return $min$

Heap-Decrease-Key($A$, $i$, $key$)
    if $key > A[i]$ then
        error "new key is larger than current key"
    $A[i] = key$
    while $i > 1$ and $A[$Parent$(i)] > A[i]$ do
        exchange $A[i]$ with $A[$Parent$(i)]$
        $i =$ Parent$(i)$

Min-Heap-Insert($A$, $key$)
    $A.heap\text{-}size = A.heap\text{-}size + 1$
    $A[A.heap\text{-}size] = +\infty$
    Heap-Decrease-Key($A, A.heap\text{-}size, key$)
```

6.5-4 Why do we bother setting the key of the inserted node to $-\infty$ in line 2 of MAX-HEAP-INSERT when the next thing we do is increase its key to the desired value?

Beacause the HEAP-INCREASE-KEY procedure requires that the new key is greater than or equal to the current key.

6.5-5 Argue the correctness of HEAP-INCREASE-KEY using the folowing loop invariant:

At the start of each iteration of the **while** loop of lines 4-6, $A[\text{PARENT}(i)] \geq A[\text{LEFT}(i)]$ and $A[\text{PARENT}(i)] \geq A[\text{RIGHT}(i)]$, if these nodes exist, and the subarray $A[1, \dots, A.heap\text{-}size]$ satisfies the max-heap property, except that there may be one violation: $A[i]$ may be larger than $A[\text{PARENT}(i)]$.

You may assume that subarray $A[1, \dots, A.heap\text{-}size]$ satisfies the max-heap property at the time HEAP-INCREASE-KEY is called.

We need to show that this invariant is true prior to the first loop iteration, that each iteration of the loop maintains the invariant, and that the invariant provides a useful property to show correctness when the loop terminates.

- **Initialization.** Before the **while** loop, A is a valid max-heap with a possible change on the value of the element $A[i]$. Thus, the invariants $A[\text{PARENT}(i)] \geq A[\text{LEFT}(i)]$ and $A[\text{PARENT}(i)] \geq A[\text{RIGHT}(i)]$ holds (these values were not changed before the loop). Since the new value of $A[i]$ is equal or larger its previous value and its previous value was equal or larger than the value of its children (A was a max-heap at the time HEAP-INCRESE-KEY was called), the only possible violation on the heap is that $A[i]$ may be larger than $A[\text{PARENT}(i)]$, thus the second invariant also holds.
- **Maintenance.** By the loop invariant, the only possible violation is that $A[i]$ may be larger than $A[\text{PARENT}(i)]$. If there is no violation ($A[i]$ does not have a parent of $A[i] \leq A[\text{PARENT}(i)]$), the loop terminates and A is a valid max-heap. If there is a violation on $A[i]$, the positions of $A[i]$ and $A[\text{PARENT}(i)]$ are exchanged. From the loop invariant, before the exchange, $A[\text{PARENT}(i)] \geq A[\text{LEFT}(i)]$ and $A[\text{PARENT}(i)] \geq A[\text{RIGHT}(i)]$, which implies that, after the exchange, the new $A[i]$ will not violate the max-heap property anymore and the invariants $A[\text{PARENT}(i)] \geq A[\text{LEFT}(i)]$ and $A[\text{PARENT}(i)] \geq A[\text{RIGHT}(i)]$ will remain valid. The only possible violation after the exchange is that $A[\text{PARENT}(i)]$ may be larger than $A[\text{PARENT}(\text{PARENT}(i))]$, but setting $i = \text{PARENT}(i)$ preserves the loop invariant for the next iteration.
- **Termination.** At termination, either $i = 1$ or $A[i] \leq A[\text{PARENT}(i)]$. In both cases, $A[i]$ is not larger than $A[\text{PARENT}(i)]$, which implies that A is a valid max-heap.

6.5-6 Each exchange operation on line 5 of HEAP-INCREASE-KEY typically requires three assignments. Show how to use the idea of the inner loop of INSERTION-SORT to reduce the three assignments down to just one assignment.

The updated pseudocode is stated below.

```
Heap-Increase-Key($A$, $i$, $key$)
    if $key < A[i]$ then
        error "new key is smaller than current key"
    while $i > 1$ and $A[\texttt{Parent}(i)] < key$ do
        $A[i] = A[\texttt{Parent}(i)]$
        $i = \texttt{Parent}(i)$
    $A[i] = key$
```

6.5-7 Show how to implement a first-in, first-out queue with a priority queue. Show how to implement a stack with a priority queue. (Queues and stacks are defined in Section 10.1.)

A first-in, first-out queue can be implemented using a min-priority-queue, in such a way that each heap element is a tuple (key, handle) and the key of a new element is greater than the key of the current elements. The EXTRACT-MIN operation will always return the oldest element (minimum key value) and the INSERT operation will keep the min-heap property. A stack can be implemented similarly, but with a max-priority-heap instead of a min-priority-heap.

6.5-8 The operation HEAP-DELETE(A, i) deletes the item in node i from heap A. Give an implementation of HEAP-DELETE that runs in $O(\lg n)$ time for an n-element max-heap.

The pseudocode is stated below.

```
Heap-Delete($A$, $i$)
    $A[i] = A[heap\text{-}size]$
    $A.heap\text{-}size = A.heap\text{-}size - 1$
    Max-Heapify($A, i$)
```

6.5-9 Give an $O(n \lg k)$-time algorithm to merge k sorted lists into one sorted list, where n is the total number of elements in all the input lists. (Hint: Use a min-heap for k-way merging.)

Let T denote the final sorted list and S_i the ith input list. The pseudocode is stated below.

```
Merge-Lists-Min-Heap($S_1, S_2, \ldots, S_k$)
1   Let $T$ be a list
2   Let $H$ be a min-heap of tuples in the form $(key, j)$
3   for $i = 1$ to $k$ do
4       Insert($H, (S_i[1], i)$)
5       $p_i = 2$
6   while $H.heap\text{-}size > 0$ do
7       $(key, j) = $ Extract-Min($H$)
8       add $key$ to $T$
9       if $p_j \leq S_j.length$ then
10          Insert($H, (S_j[p_j], j)$)
11          $p_j = p_j + 1$
12  return $T$
```

The **for** loop of lines 3-5 runs in $O(k \lg k)$. The **while** loop of lines 6-11 will iterate n times and each iteration takes $O(\lg k)$. Since $n \geq k$, the algorithm runs in $O(n \lg k)$.