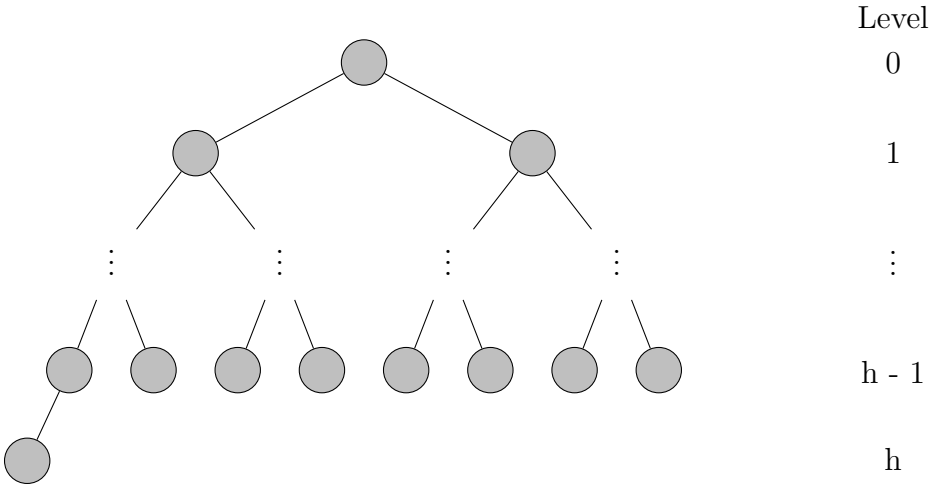
Exercises in Section 6.1

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

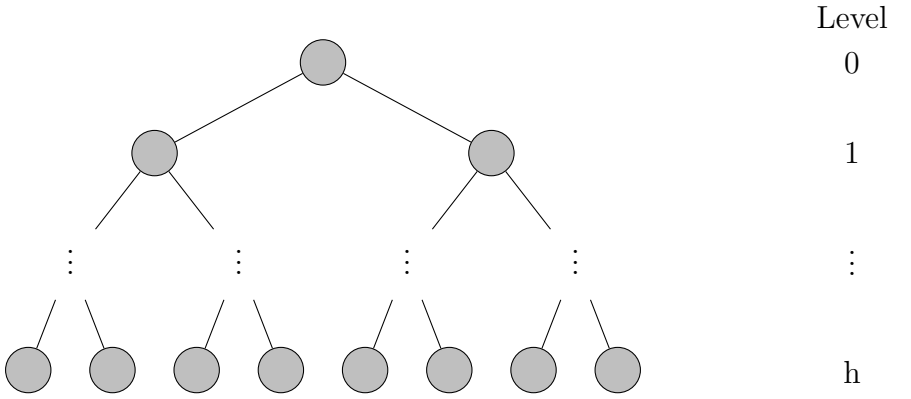
   Solution

   (i) Minimum: There is only one node at level h.



$$\textit{Minimum number of elements} = 1 + \sum_{i=0}^{h-1} 2^i = 2^h.$$

   (ii) Maximum: There are 2^h nodes, which is the maximally possible number, at level h.



$$\textit{Maximum number of elements} = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1.$$

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

   Solution In Exercise 6.1-1, we have found a set $\{2^h, 2^h + 1, ..., 2^{h+1} - 1\}$ consisting of all possible number of elements in a heap of height h. Notice that these sets (for each

nonnegative integer h) cover the set of positive integers, that is,

$$\sum_{h=0}^{\infty} \{2^h, 2^h + 1, ..., 2^{h+1} - 1\} = \mathbb{Z}^+.$$

Also, all these sets are disjoint, that is, do not coincide with each other. Therefore, for any given n-element heap, there must be a unique h such that

$$2^h \leq n \leq 2^{h+1} - 1.$$

Take floor of logarithm on all parts of this inequality, we get

$$h = \lfloor \lg 2^h \rfloor \leq \lfloor \lg n \rfloor \leq \lfloor \lg (2^{h+1} - 1) \rfloor = h,$$

where the last equality holds because

$$h = \lg 2^h \leq \lg (2^{h+1} - 1) < \lg 2^{h+1} = h + 1.$$

Hence, $h = \lfloor \lg n \rfloor$.

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

   Solution Note that a subtree of a max-heap is itself a max-heap. Suppose the largest value is not contained in an element other than the root, say e. Then, Parent(e), which represents the same element in both the subtree and the original max-heap, must have a smaller value than e, which contradicts the definition of max-heaps.

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

   Solution By the definition of max-heaps, the smallest element should have no children. Therefore, the smallest element resides only on the leaves of the max-heap.

5. At which levels in a max-heap might the k-th largest element reside, for $2 \leq k \leq \lfloor n/2 \rfloor$, assuming that all elements are distinct?

   Solution First, we find the upper bound of the levels a k-th largest element may reside. Exercise 6.1-2 gives a trivial bound $h = \lfloor \lg n \rfloor$, the height of an n-element heap, and of course, an element cannot have a level bigger than the height of the heap. Another trivial upper bound is $k-1$, because there are at most $k-1$ elements in the path from the root to the k-th largest element. Therefore, we get a upper bound $\min\{\lfloor \lg n \rfloor, k-1\}$. To achieve this upper bound, we put the 2-nd largest element as the left child to the root, the 3-rd largest element as the left child to the 2-nd largest element, and repeat this procedure until the level reaches $\min\{\lfloor \lg n \rfloor, k-1\}$, where we put the k-th largest element.

The lower bound is trivial too, and it is 1. It is easy to see this bound is tight, because we can always put the k-th largest element as a child of the root, and since $k \leq \lfloor n/2 \rfloor$, there are at least $\lceil n/2 \rceil$ elements that are smaller than the k-th largest element, and we can safely put the smallest $\lceil n/2 \rceil$ elements under the k-th largest element as its descendants (the elements in the subtree induced by an element, except the element itself).

Hence, the levels in a max-heap the k-th largest element may reside are in the set $\{1, 2, \ldots, \min\{\lfloor \lg n \rfloor, k-1\}\}$.

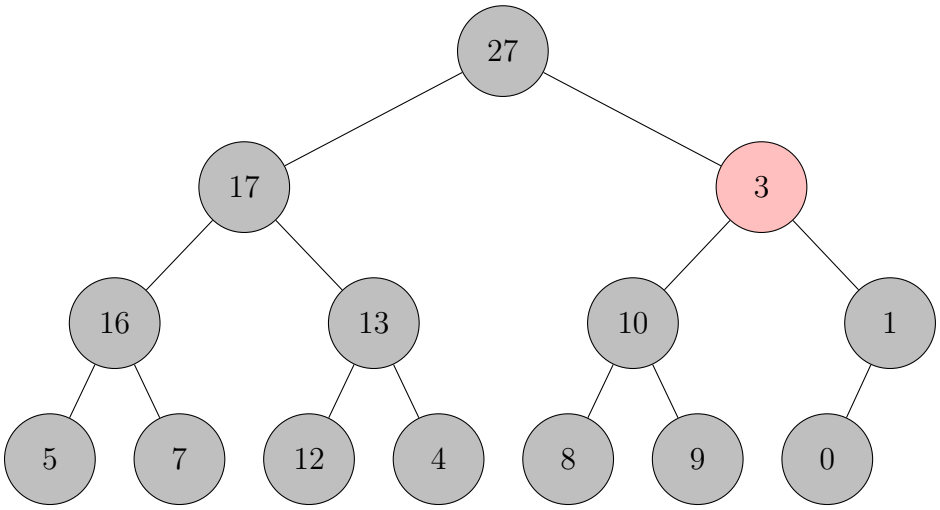
6. Is an array that is in sorted order a min-heap?
   Solution Yes. Let A be an array that is sorted in ascending order. $A[1]$ is the root, and for each $i = 2, 3, \ldots, n$, we have
   $$\text{Parent}(i) = \lfloor \frac{i}{2} \rfloor < \frac{i}{2} + 1 \leq \frac{i}{2} + 1 + (\frac{i}{2} - 1) = i.$$
   Since A is sorted in ascending order,
   $$A[\text{Parent}(i)] < A(i).$$
   Therefore, A is a min-heap.

7. Is the array with values $\langle 33, 19, 20, 15, 13, 10, 2, 13, 16, 12 \rangle$ a max-heap?
   Solution No, because
   $$A[\text{Parent}(9)] = A[4] = 15 < 16 = A[9].$$
   This contradicts the definition of max-heaps.

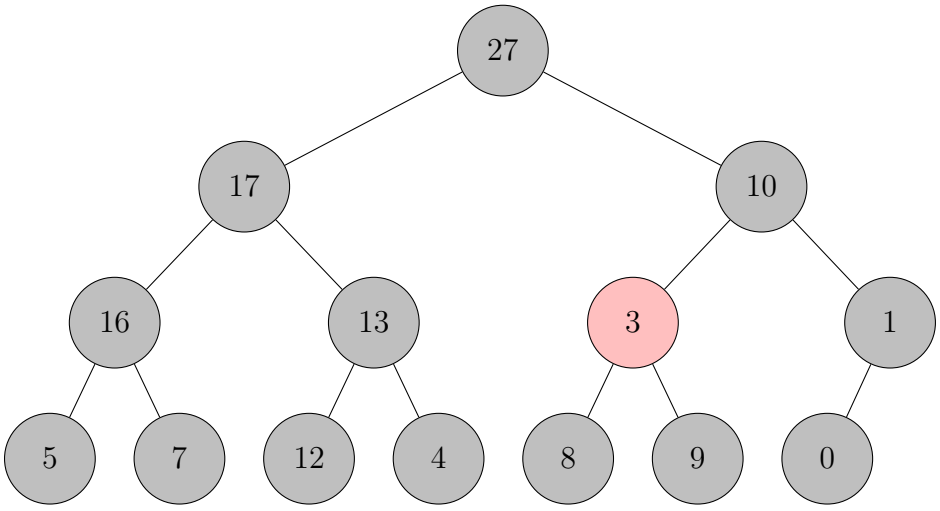
8. Show that, with the array representation for storing an n-element heap, the leaves are the nodes indexed by $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$.
   Solution For each node of index $i = 1, 2, \ldots, \lfloor n/2 \rfloor$, it has a left child indexed $2i \leq 2\lfloor n/2 \rfloor \leq n$. For each node of index $j = \lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$, we have
   $$2j \geq 2(\lfloor \frac{n}{2} \rfloor + 1) > 2\left((\frac{n}{2} - 1) + 1\right) = n.$$
   So, they should not have a left child, and therefore, they are leaves.

Exercises in Section 6.2

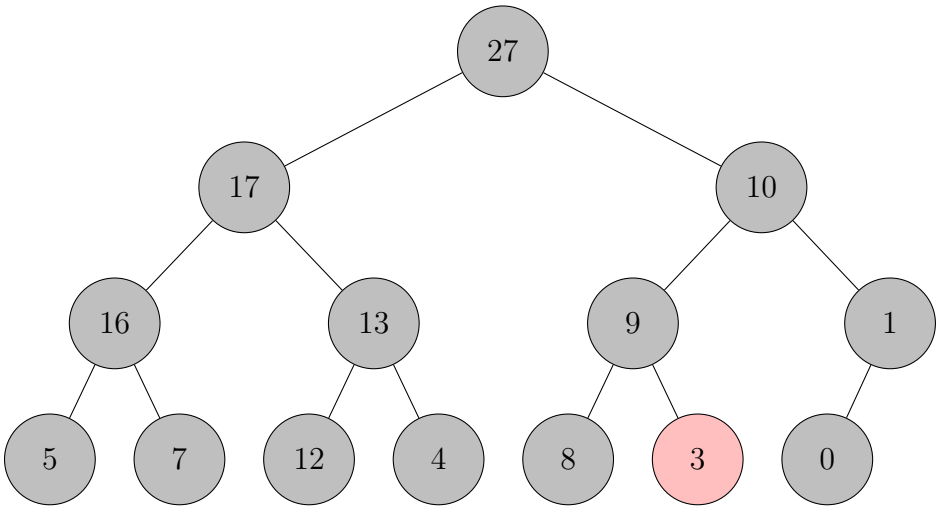
1. Using Figure 6.2 as a model, illustrate the operation of MAX-HEAPIFY$(A, 3)$ on the array $A = \langle 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0 \rangle$.
   Solution We start from the tree representation of A:



Since $A[3] = 3$, and its children $A[6] = 10$ and $A[7] = 1$, among which $A[6] = 10$ is the maximum, we exchange $A[3]$ with $A[6]$:



Similarly, since $A[6] = 3$, and its children $A[12] = 8$ and $A[13] = 9$, among which $A[13] = 9$ is the maximum, we exchange $A[6]$ with $A[13]$:

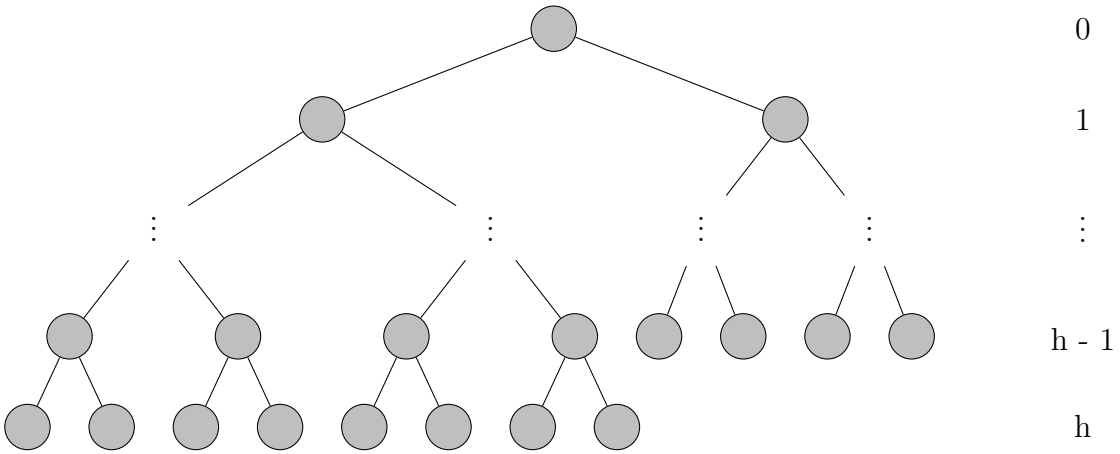


Now, we finished the procedure of MAX-HEAPIFY$(A, 3)$.

2. Show that each child of the root of an n-node heap is the root of a subtree containing at most $2n/3$ nodes. What is the smallest constant α such that each subtree has at most αn nodes? How does that affect the recurrence (6.1) and its solution?

Solution

(i) Consider the case where the left subtree contains the highest proportion of nodes, that is, this left subtree contains all possible leaves of level $h = \lfloor \lg n \rfloor$, while the right subtree contains no leaves of level $h = \lfloor \lg n \rfloor$. In tree representation,



The number of nodes of the whole heap is

$$n = \sum_{i=0}^{h} 2^i - 2^{h-1} = (2^{h+1} - 1) - 2^{h-1} = 3(2^{h-1} - 1/3).$$

The number of nodes of the left subtree is

$$n_l = \sum_{i=1}^{h} 2^{i-1} = 2^h - 1 < 2(2^{h-1} - 1/3) = 2n/3.$$

(ii) Every subtree of a heap must have a root other than the root of the heap. Therefore, it must be one of the left and right subtrees we discussed in (i), or a subtree of one of them. Therefore, every subtree has at most $2n/3$ nodes. Hence, $\alpha = 2/3$.

(iii) The value of α in (ii) tells us why the coefficient of n in recurrence (6.1) is $2/3$ (or, in other words, $b = 3/2$).

3. 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 with that of MAX-HEAPIFY?
Solution

Algorithm 1 MIN-HEAPIFY(A, i)

```
1: l = LEFT(i)
2: r = RIGHT(i)
3: if l ≤ A.heap-size and A[l] < A[i] then
4:     smallest = l
5: else
6:     smallest = i
7: end if
8: if r ≤ A.heap-size and A[r] < A[smallest] then
9:     smallest = r
10: end if
11: if smallest ≠ i then
12:     exchange A[i] with A[smallest]
13:     MAX-HEAPIFY(A, smallest)
14: end if
```

The running time of MIN-HEAPIFY is the same as MAX-HEAPIFY as they are essentially the same procedure except the order of elements are opposite.

4. What is the effect of calling MAX-HEAPIFY when the element $A[i]$ is larger than its children?
Solution Nothing but your computer would still compare the value of i with its children as programmed and decide there is no need to move $A[i]$ at the end.

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

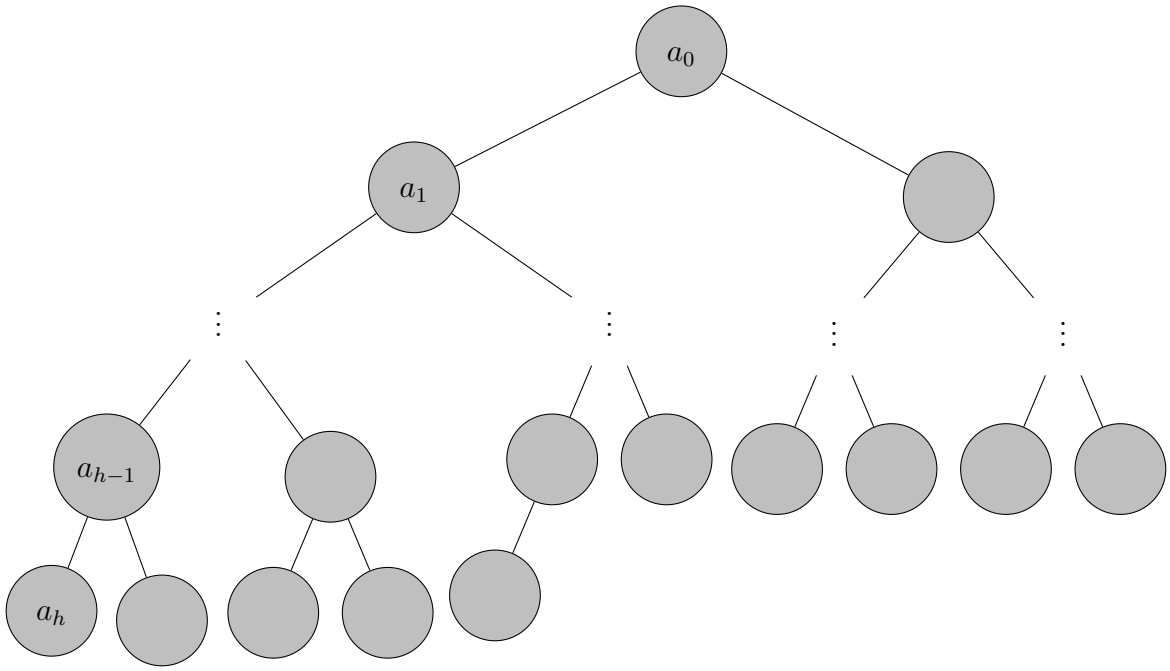
Solution For each $i > A.heap\text{-}size/2$, i does not have a child, that is, i is a leaf. Therefore, the variable $largest$ will be assigned the value i, and none of other if statements will be triggered. Therefore, nothing will happen to A.

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

Algorithm 2 MAX-HEAPIFY(A, i)

```
1: while True do
2:    l = LEFT(i)
3:    r = RIGHT(i)
4:    if l ≤ A.heap-size and A[l] > A[i] then
5:       largest = l
6:    else
7:       largest = i
8:    end if
9:    if r ≤ A.heap-size and A[r] > A[largest] then
10:      largest = r
11:   end if
12:   if largest = i then
13:      break
14:   end if
15:   exchange A[i] with A[largest]
16:   i = largest
17: end while
```

7. 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.)
   Solution Exercise 6.1-2 tells us the height of a heap with n nodes is $h = \lfloor \lg n \rfloor$. Consider the worst case where, in particular, the left most path consists of the $h+1$ largest elements that are in descending order (as shown in the graph below, assuming $a < a_0 < a_1 < \cdots < a_h$, where a stands for any other elements in this heap).



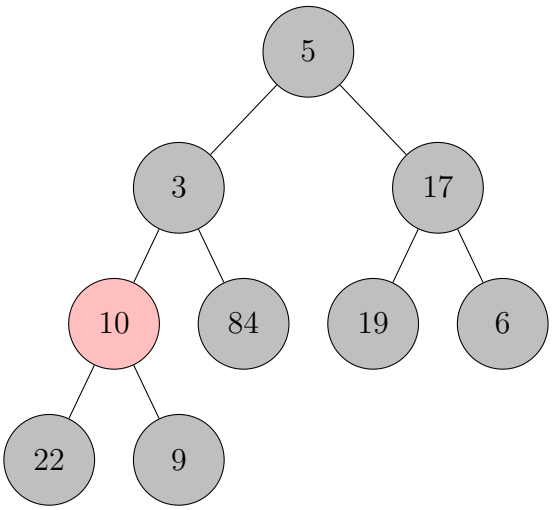
Therefore, if we call MAX-HEAPIFY$(A, 1)$, a_1 must be the largest element among a_0 and its children, and MAX-HEAPIFY$(A, 2)$ will be called as an iteration. Again, a_2 must be the largest among a_1 and its children, so MAX-HEAPIFY$(A, 4)$ will be called as an iteration. Repeat this process, the following iterations are MAX-HEAPIFY$(A, 8)$, MAX-HEAPIFY$(A, 16)$, $\ldots$, MAX-HEAPIFY$(A, 2^{h-1})$, with at least $h = \lfloor \lg n \rfloor$ iterations in total, and with running time $\Theta(1)$ for each. Hence, the running time of MAX-HEAPIFY is

$$\Omega(\lfloor \lg n \rfloor \Theta(1)) = \Omega(\lfloor \lg n \rfloor) = \Omega(\lg n).$$

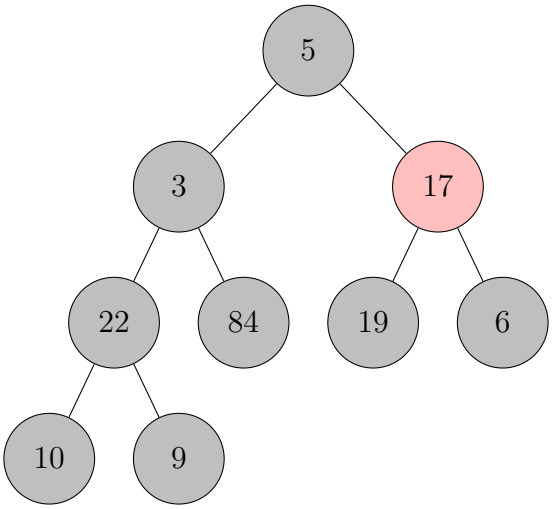
Exercises in Section 6.3

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

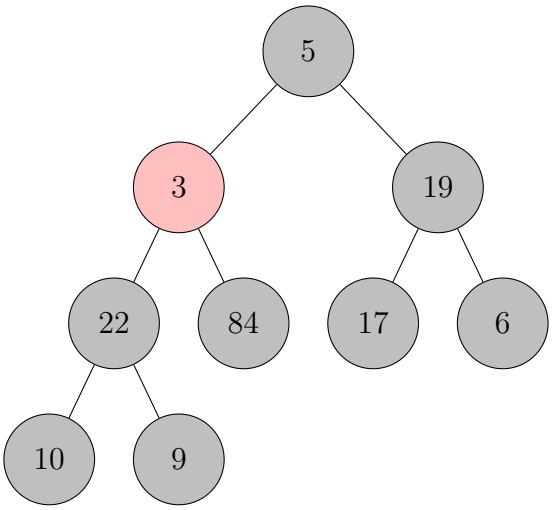
   Solution We start from the tree representation of A:



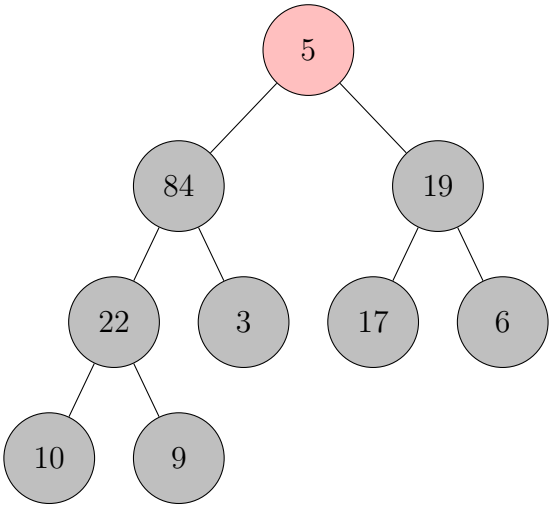
   MAX-HEAPIFY$(A, 4)$ exchanges $A[4] = 10$ and $A[8] = 22$:



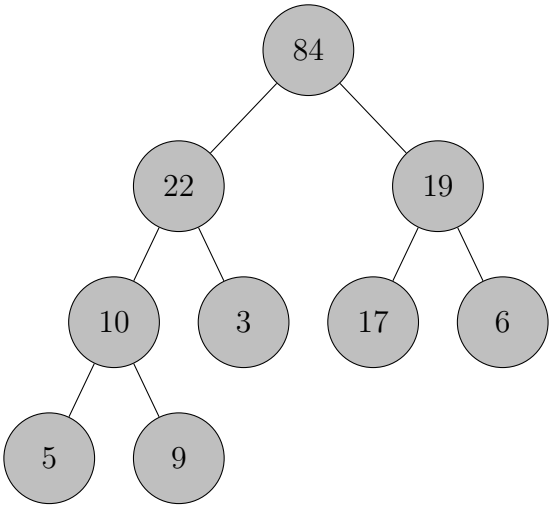
   MAX-HEAPIFY$(A, 3)$ exchanges $A[3] = 17$ and $A[6] = 19$:



MAX-HEAPIFY$(A, 2)$ exchanges $A[2] = 3$ and $A[5] = 84$:



MAX-HEAPIFY$(A, 1)$ exchanges $A[1] = 5$ and $A[2] = 84$, exchanges $A[2] = 5$ and $A[4] = 22$, and exchanges $A[4] = 5$ and $A[8] = 10$:

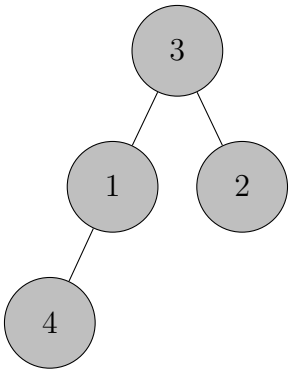


Congratulations! We have built a max-heap from a random array!

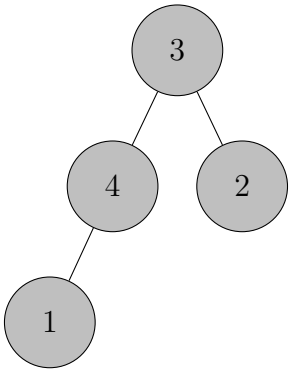
2. Show that $\lceil n/2^{h+1} \rceil \geq 1/2$ for $0 \leq h \leq \lfloor \lg n \rfloor$.
   Solution For any $0 \leq h \leq \lfloor \lg n \rfloor$, $n/2^{h+1}$ is positive. So,

$$\lceil n/2^{h+1} \rceil \geq 1 \geq 1/2.$$

3. Why does the loop index i in line 2 of BUILD-MAX-HEAP decrease from $\lfloor n/2 \rfloor$ to 1 rather than increase from 1 to $\lfloor n/2 \rfloor$?
   Solution If the loop index i was increasing from 1 to $\lfloor n/2 \rfloor$, BUILD-MAX-HEAP may fail to build a max-heap. Let us try to see it from an example. Consider a heap $A = \langle 3, 1, 2, 4 \rangle$, or in tree representation:



   If our i was to start from 1, this program would start by calling MAX-HEAPIFY$(A, 1)$, which does nothing to the heap because $A[i]$ is larger than both its children. Then, MAX-HEAPIFY$(A, 2)$ would exchange $A[2]$ with $A[4]$, and here is our result:



   Did you find the flaw of this procedure? Our program successfully compared $A[1]$ with its children, however, there might be larger elements (especially, the largest one) "hidden" in deeper levels, and a simple MAX-HEAPIFY$(A, 1)$ can do nothing about it. Even worse, once we missed the largest element, we could never get a chance to place it to the correct position (that is, the root) as only MAX-HEAPIFY$(A, 1)$ has the ability to change the value of $A[1]$! Now, we should pay more attention on the order of our loop indices.

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

   Solution Let A be an n-element heap, we denote n_h as the number of nodes of height h. We will prove the proposition desired by induction.

   The nodes of height 0 are the leaves. By the solution of Exercise 6.1-8, all leaves are indexed by $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$. Therefore,

$$n_0 = n - \lfloor n/2 \rfloor = \lceil n/2 \rceil,$$

   which verifies the base case. Now, suppose $n_{h-1} \leq \lceil n/2^h \rceil$. We consider two cases:

   (i) n_{h-1} is even. Then, each node of height h has exactly 2 children. Therefore,

$$n_h = n_{h-1}/2 = \lceil n_{h-1}/2 \rceil.$$

   (ii) n_{h-1} is odd. Then, all nodes of height h have 2 children except exactly one who has only 1 (left) child. Therefore,

$$n_h = (n_{h-1} + 1)/2 = \lceil n_{h-1}/2 \rceil.$$

   In both cases, we have $n_h = \lceil n_{h-1}/2 \rceil$. Therefore,

$$n_h \leq \lceil \lceil n/2^h \rceil / 2 \rceil = \lceil n/2^{h+1} \rceil,$$

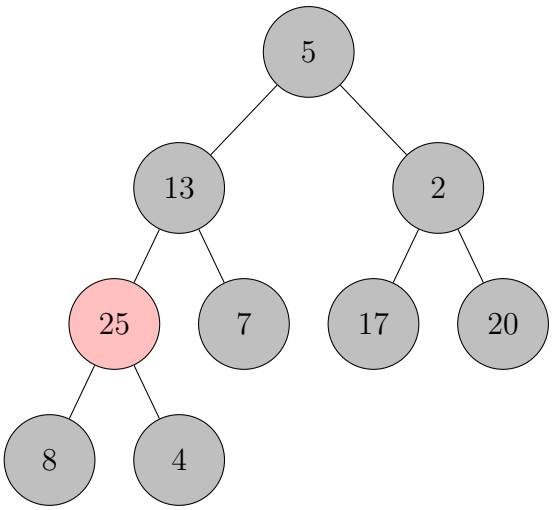
   where the last equality holds by equation (3.5).

Exercises in Section 6.4

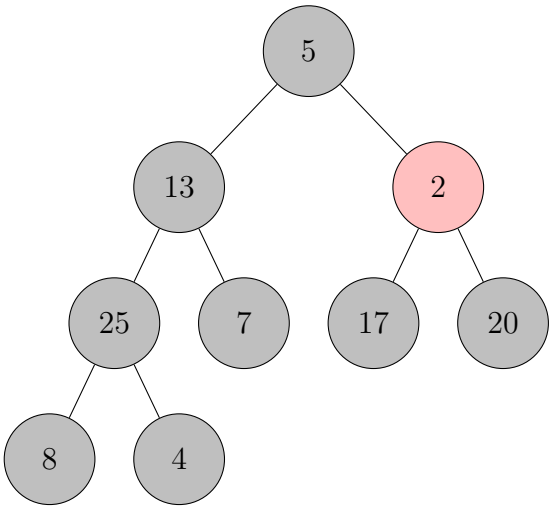
1. Using Figure 6.4 as a model, illustrate the operation of HEAPSORT on the array $A = \langle 5, 13, 2, 25, 7, 17, 20, 8, 4 \rangle$.

   Solution We apply heap sort in two steps. First, BUILD-MAX-HEAP(A, n) is called to produce a max-heap. Next, we repeatedly call MAX-HEAPIFY$(A, 1)$ and put $A[1]$ to the right place.

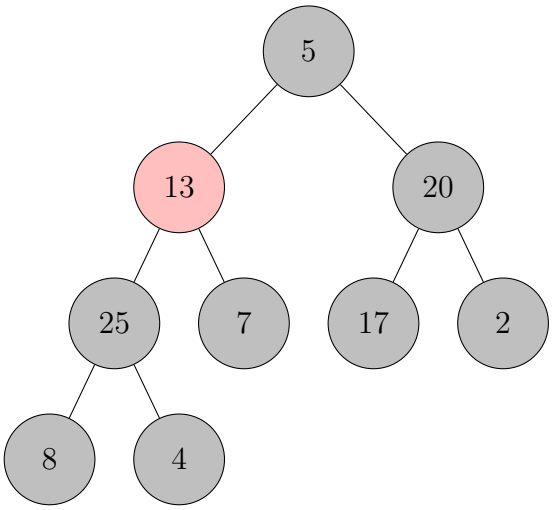
   Step 1. Build a max-heap. We start from the tree representation of A:



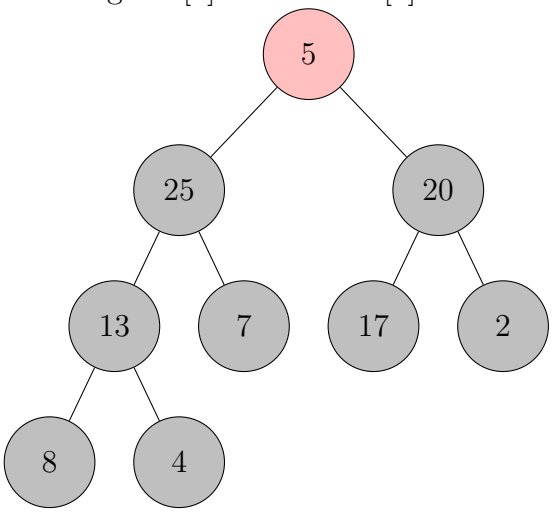
   MAX-HEAPIFY$(A, 4)$ does not exchange nodes:



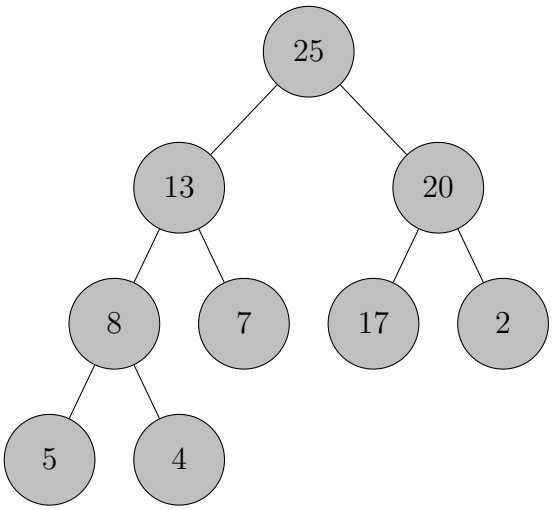
   MAX-HEAPIFY$(A, 3)$ exchanges $A[3] = 2$ and $A[7] = 20$:



MAX-HEAPIFY$(A, 2)$ exchanges $A[2] = 13$ and $A[5] = 25$:

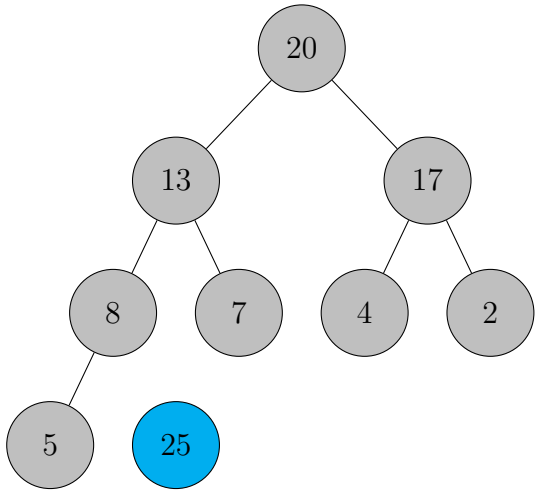


MAX-HEAPIFY$(A, 1)$ exchanges $A[1] = 5$ and $A[2] = 25$, exchanges $A[2] = 5$ and $A[4] = 13$, and exchanges $A[4] = 5$ and $A[8] = 8$:

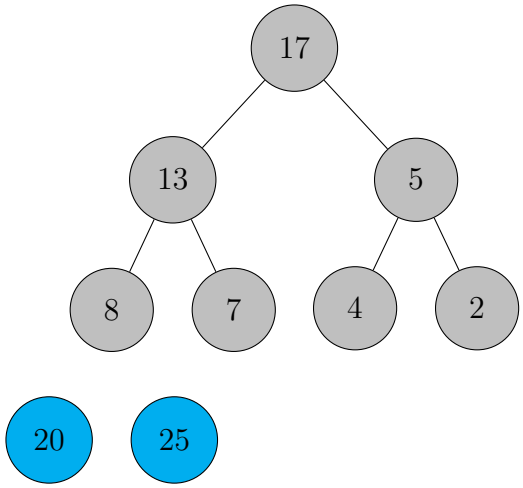


Now we have got a max-heap.

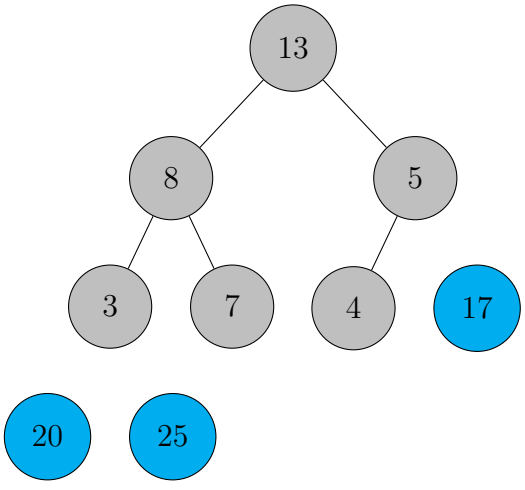
Step 2. Start sorting! Exchange $A[1] = 25$ with $A[9] = 4$; MAX-HEAPIFY$(A, 1)$ exchanges $A[1] = 4$ and $A[3] = 20$, and exchanges $A[3] = 4$ with $A[6] = 17$:



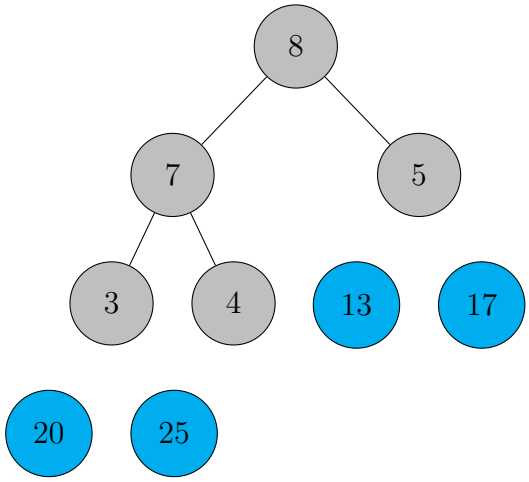
Exchange $A[1] = 20$ with $A[8] = 5$; MAX-HEAPIFY$(A, 1)$ exchanges $A[1] = 5$ and $A[3] = 17$:



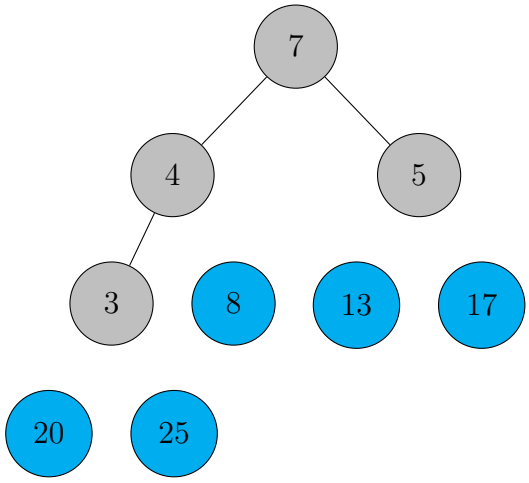
Exchange $A[1] = 17$ with $A[7] = 2$; MAX-HEAPIFY$(A, 1)$ exchanges $A[1] = 2$ and $A[2] = 13$, and exchanges $A[2] = 2$ and $A[4] = 8$:



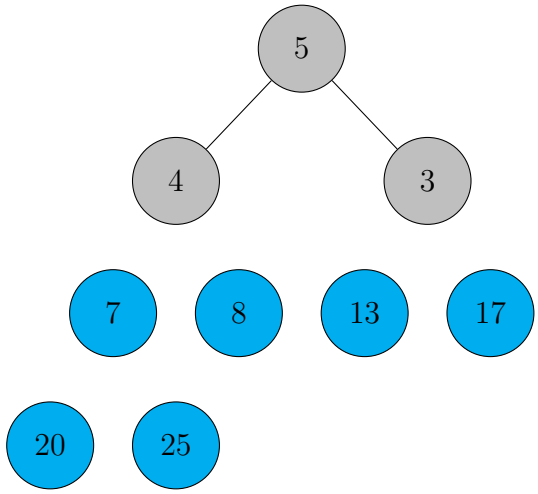
Exchange $A[1] = 13$ with $A[6] = 4$; MAX-HEAPIFY$(A, 1)$ exchanges $A[1] = 4$ and $A[2] = 8$, and exchanges $A[2] = 4$ and $A[5] = 7$:



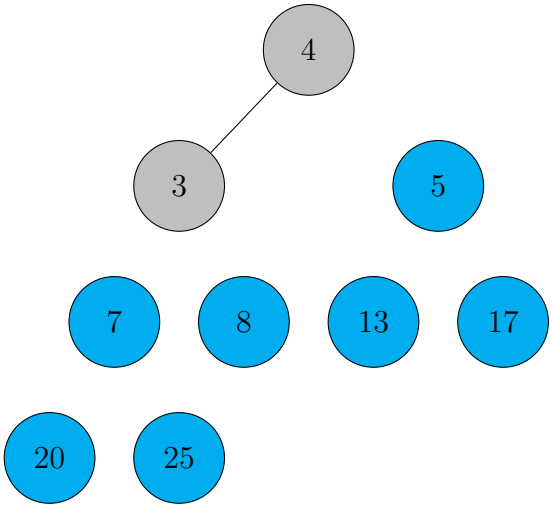
Exchange $A[1] = 8$ with $A[5] = 4$; MAX-HEAPIFY$(A, 1)$ exchanges $A[1] = 4$ and $A[2] = 7$:



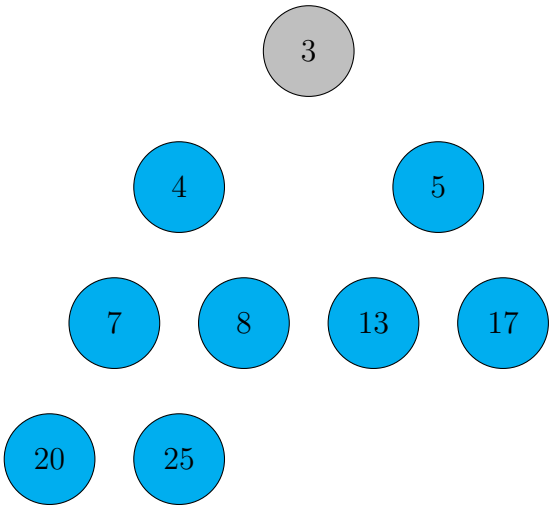
Exchange $A[1] = 7$ with $A[4] = 3$; MAX-HEAPIFY$(A, 1)$ exchanges $A[1] = 3$ and $A[3] = 5$:



Exchange $A[1] = 5$ with $A[3] = 3$; MAX-HEAPIFY$(A, 1)$ exchanges $A[1] = 3$ and $A[2] = 4$:



Exchange $A[1] = 4$ with $A[2] = 3$, and we are done!



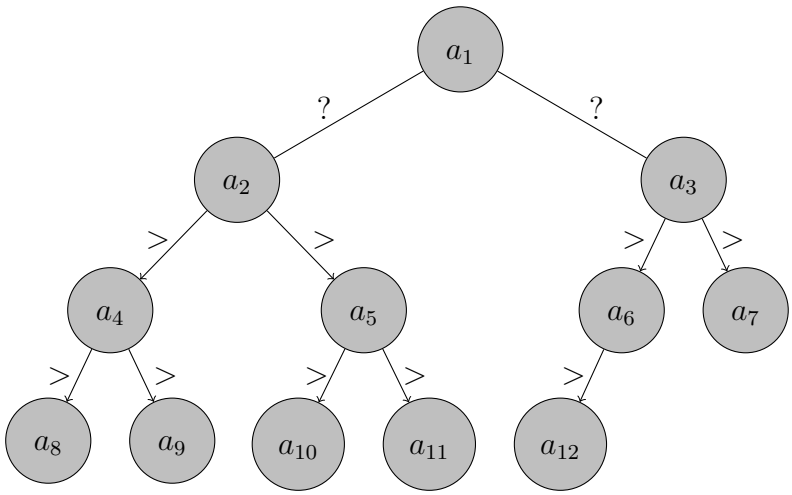
In array form, we get $A = \langle 3, 4, 5, 7, 8, 13, 17, 20, 25 \rangle$ sorted.

2. Argue the correctness of HEAPSORT using the following loop invariant: At the start of each iteration of the **for** loop of lines 2-5, the subarray $A[1 : i]$ us a max-heap containing the i smallest elements of $A[1 : n]$, and the subarray $A[i + 1 : n]$ contains the $n - i$ largest elements of $A[1 : n]$, sorted.

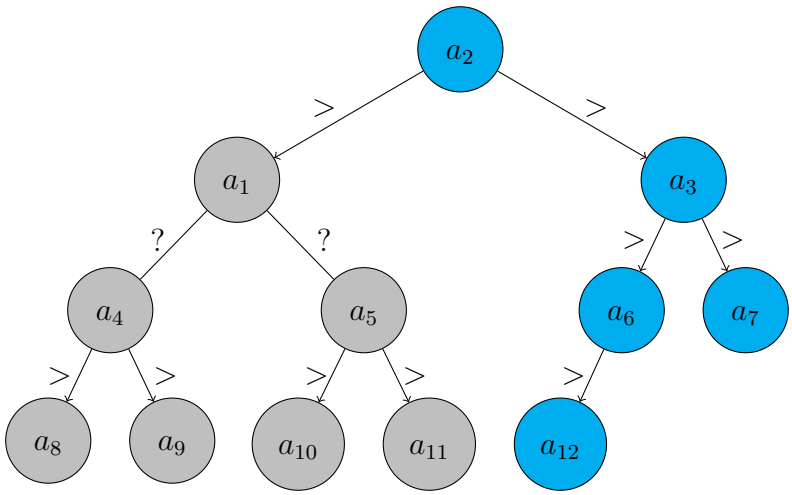
   Solution We denote the loop variant as $P(i)$, where $i = n, n - 1, \ldots, 2$, and prove it by induction. At the start of the first iteration, $i = n$, the correctness of $P(n)$ is guaranteed by BUILD-MAX-HEAP(A, n) before the loop.

   Now, we assume $P(i)$ is true, and show $P(i - 1)$ is true. At the start of the loop where the loop index is i, by our assumption, the subarray $A[1 : i]$ is a max-heap, thus $A[1]$ is the $n - i + 1$ largest element in A. Exchanging $A[1]$ with $A[i]$, the subarray $A[i, n]$ contains the

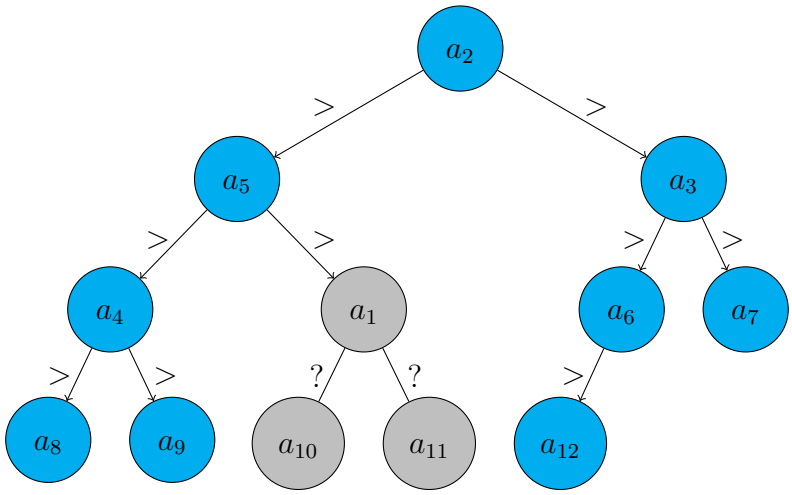
$n - i + 1$ largest elements of A. Now, we are left to show $A[1 : i - 1]$ is again a max-heap after calling MAX-HEAPIFY$(A[1 : i - 1], 1)$. To make our lives easier, we will use a silly proposition without proof: A heap is a max-heap if and only if the root is greater than both its children, and the subtrees rooted at these two children are max-heaps. After exchanging $A[1]$ and $A[i]$, the subtree $A[1 : i - 1]$ is almost a max-heap (the subtrees rooted at $A[2]$ and $A[3]$ are both max-heaps, except possibly the root is smaller than its children): (The nodes that has been sorted, that is, $A[i : n]$, are not shown in the sample graph below.)



Since MAX-HEAPIFY$(A, 1)$ is called, without loss of generality, it exchanges $A[1]$ with $A[2]$. Now, we have $A[1] > A[2]$, $A[1] > A[3]$, and the subtree rooted at $A[3]$ is a max-heap:



The cyan part (except the root) is the subtree rooted at $A[3]$, and is a max-heap. The root $A[1] = a_2$ is also colored because it is larger than both its children. Now, if the subtree rooted at $A[2] = a_1$ is a max-heap, then we are done. This subtree, is almost a max-heap except possibly the root $A[2]$ is smaller than its children. Note that this is exactly the situation we met just now! Again, MAX-HEAPIFY$(A, 1)$ will compare $A[2]$ with its children, and exchange them when necessary. For example, it exchanges $A[2]$ with $A[5]$:



Everything is done except the subtree rooted at $A[5] = a_1$. Now, it should be clear that this procedure will repeat until the uncolored subtree is trivial (that is, with only one node), or it turns that its root is already larger than both its children. In either case, we are done to say it is a max-heap, and conclude $A[1 : i-1]$ is indeed a max-heap before the next loop.

3. What is the running time on HEAPSORT on an array A of length n that is already sorted in increasing order? How about if the array is already sorted in decreasing order?
   Solution
   (i) If A is sorted in increasing order, it is not good news for BUILD-MAX-HEAP(A, n) (it is the worst, in fact) because every internal node is smaller than its children. We take the worst case running time $\Theta(n)$. The resulting max-heap, however, has no special structure to reduce the time complexity. The **for** loop requires $\Theta(n \lg n)$. The total running time is $\Theta(n \lg n)$. (it is hard to analyze the actual structure of the max-heap produced in line 1, one can see it as a corollary of Exercise 6.4.5.)
   (ii) If A is sorted in decreasing order, BUILD-MAX-HEAP(A, n) calls MAX-HEAPIFY(A, i) $\Theta(n)$ times, where each takes only constant time. So, the running time of BUILD-MAX-HEAP(A, n) is $\Theta(n)$, leaving A unchanged. In each loop, MAX-HEAPIFY$(A, 1)$ has to repeated exchange the smallest element with its children all way down from the root to the leaf. In particular, the $\lceil n/2 \rceil$ are of level $\lfloor \lg n \rfloor$ or $\lfloor \lg n \rfloor - 1$, therefore, MAX-HEAPIFY$(A, 1)$ makes at least $(\lceil n/2 \rceil)(\lfloor \lg n \rfloor - 1) = \Omega(n \lg n)$ exchanges. So, the total running time is $\Theta(n \lg n)$.

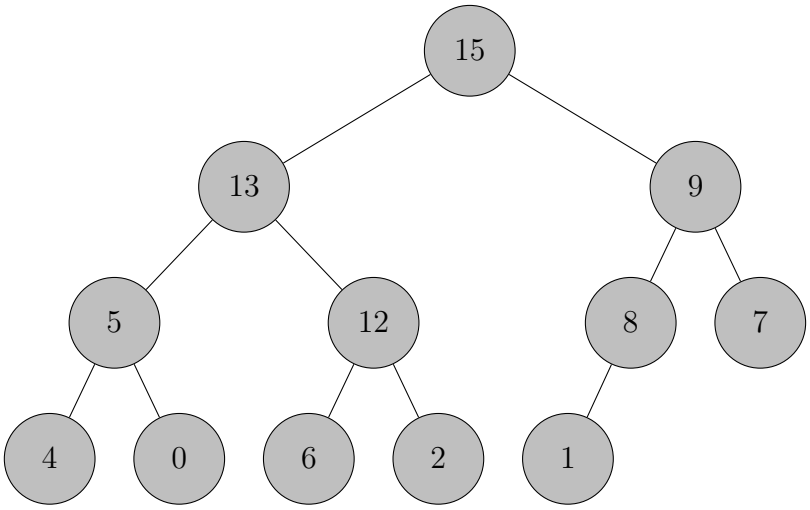
4. Show that the worst-case running time of HEAPSORT is $\Omega(n \lg n)$.
   Solution See Exercise 6.4-3 (ii).

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

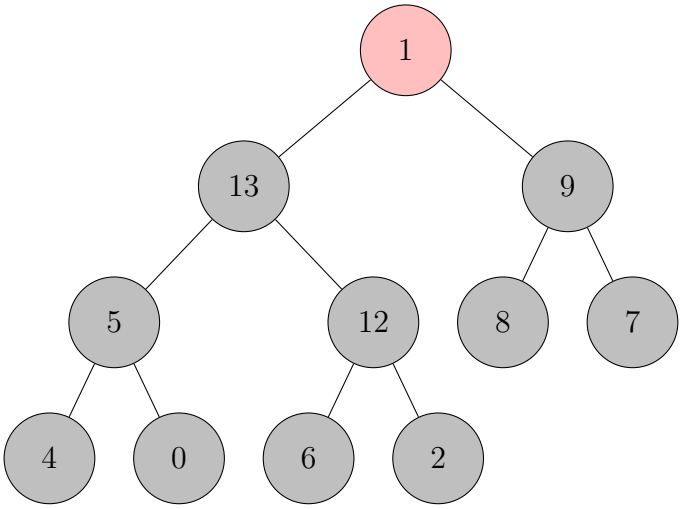
   Solution It suffices to show the **for** loop in HEAPSORT takes best-case running time of $\Omega(n \lg n)$. This is actually particularly hard and is waiting for a proof.

Exercises in Section 6.5

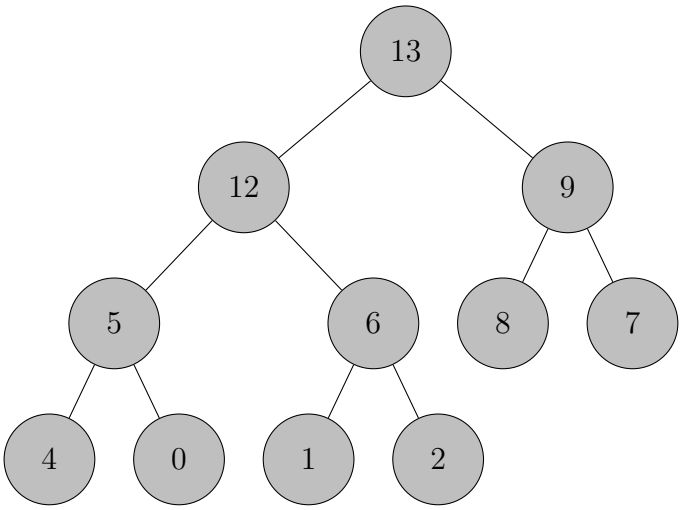
1. Suppose that the objects in a max-priority queue are just keys. Illustrate the operation of MAX-HEAP-EXTRACT-MAX on the heap $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$.
   Solution We start from the tree representation of A:



   Line 1 stores $A[1] = 15$ in the variable max. Then, line 2 revalues $A[1]$ to the last element $A[A.heapsize]$, and line 3 deducts the size of A by 1 (you can see it as we remove the last element from A):

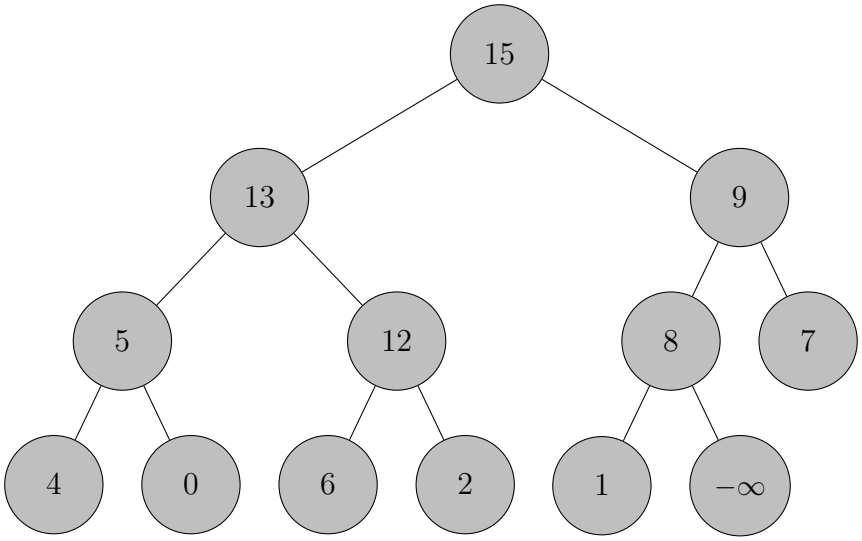


   MAX-HEAPIFY$(A, 1)$ exchanges $A[1] = 1$ with $A[2] = 13$, exchanges $A[2] = 1$ with $A[5] = 12$, and exchanges $A[5] = 1$ with $A[10] = 6$:

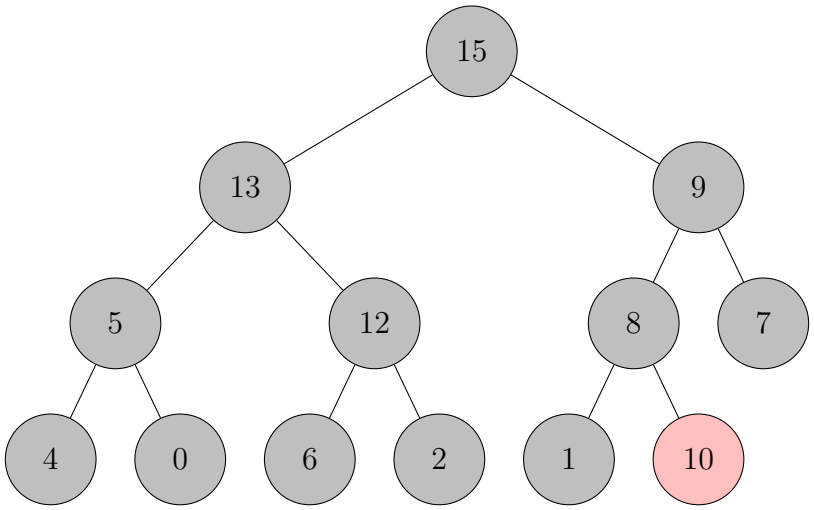


Finally, we get $max = 15$ returned and $A = \langle 13, 12, 9, 5, 6, 8, 7, 4, 0, 1, 2 \rangle$.

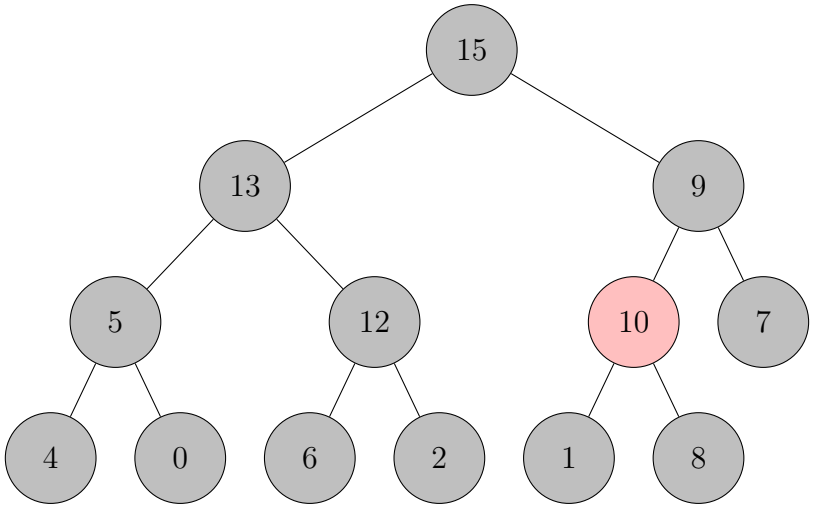
2. Suppose that the objects in a max-priority queue are just keys. Illustrate the operation of MAX-HEAP-INSERT$(A, 10)$ on the heap $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$.
Solution Line 3 will increase the size of A by 1 in order to store the extra element $x = 10$. Line 4 stores the value of x in k, that is, $k = 10$, before setting x to $-\infty$ and $A[A.heap - size] = x$:



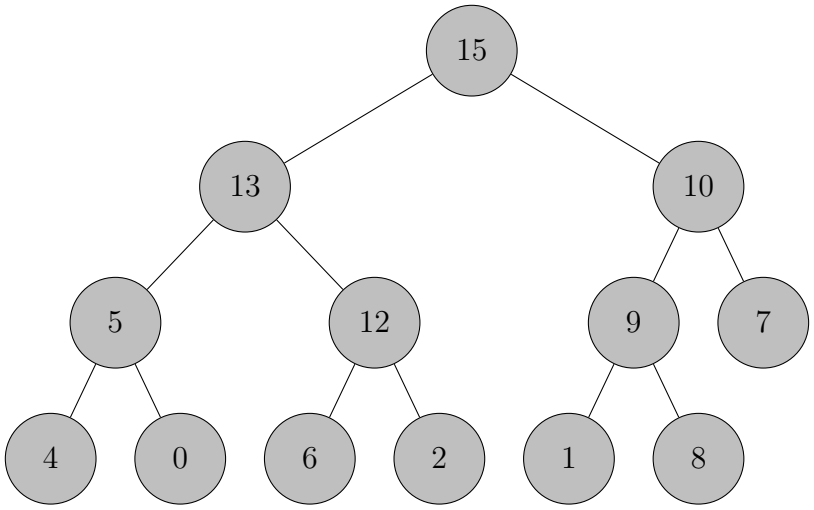
Then, MAX-HEAP-INCREASE-KEY(A, x, k) is called. Since $k = 10 > -\infty = x$, there will not be error. Line 3 changes x to $k = 10$:



We have $\textsc{Parent}(13) = 6$. Since $A[6] = 8 < 10 = A[13]$, exchange $A[13]$ with $A[6]$:



We have $\textsc{Parent}(6) = 3$. Since $A[3] = 9 < 10 = A[6]$, exchange $A[6]$ with $A[3]$:



We have $\textsc{Parent}(3) = 1$. Since $A[1] = 15 > 10 = A[3]$, we are done.

3. Write pseudocode to implement a min-priority queue with a min-heap by writing the procedures MIN-HEAP-MINIMUM, MIN-HEAP-EXTRACT-MIN, MIN-HEAP-DECREASE-KEY, and MIN-HEAP-INSERT.

   Solution

Algorithm 3 MIN-HEAP-MINIMUM(A)

```
1: if A.heap-size < 1 then
2:     error "heap underflow"
3: end if
4: return A[1]
```

Algorithm 4 MIN-HEAP-EXTRACT-MIN(A)

```
1: min = MIN-HEAP-MINIMUM(A)
2: A[1] = A[A.heap-size]
3: A.heap-size = A.heap-size − 1
4: MIN-HEAPIFY(A, 1)
5: return min
```

Algorithm 5 MIN-HEAP-DECREASE-KEY(A, x, k)

```
1: if k > x.key then
2:     error "new key is larger than current key"
3: end if
4: x.key = k
5: find the index i in array A where object x occurs
6: while i > 1 and A[PARENT(i)].key > A[i].key do
7:     exchange A[i] with A[PARENT(i)], updating the information that maps priority queue
       objects to array indices
8:     i = PARENT(i)
9: end while
```

Algorithm 6 MIN-HEAP-INSERT(A, x, n)

1: **if** $A.heap\text{-}size == n$ **then**
2: **error** "heap overflow"
3: **end if**
4: $A.heap\text{-}size = A.heap\text{-}size + 1$
5: $k = x.key$
6: $x.key = \infty$
7: $A[A.heap\text{-}size] = x$
8: map x to index *heap-size* in the array
9: MIN-HEAP-DECREASE-KEY(A, x, k)

4. Write pseudocode for the procedure MAX-HEAP-DECREASE-KEY(A, x, k) in a max-heap. What is the running time of your procedure?
Solution

Algorithm 7 MAX-HEAP-DECREASE-KEY(A, x, k)

1: **if** $k > x.key$ **then**
2: **error** "new key is larger than current key"
3: **end if**
4: $x.key = k$
5: find the index i in array A where object x occurs
6: MAX-HEAPIFY(A, i)

The running time of this procedure is a constant time plus the running time of MAX-HEAPIFY(A, i), therefore, is $O(\lg n)$.

5. Why does MAX-HEAP-INSERT bother setting the key of the inserted object to $-\infty$ in line 5 given that line 8 will set the object's key to the desired value?
Solution Line 8, that is, MAX-HEAP-INCREASE-KEY(A, x, k) requires k is larger than $x.key$, the current key of x. By setting $x.key = -\infty$ in line 5, we can guarantee that $k > -\infty = x.key$.

6. Professor Uriah suggests replacing the **while** loop of lines 5-7 in MAX-HEAP-INCREASE-KEY by a call to MAX-HEAPIFY. Explain the flaw in the professor's idea.
Solution A call to MAX-HEAPIFY(A, i) will do nothing to the array A because the new key of x is larger than the current one, and therefore it must still be larger than its children.

What we are supposed to do is to put x to a higher height (possibly no movement) because it received a larger key, and line 5-7 in MAX-HEAP-INCREASE-KEY did this work perfectly because it repeatedly exchanged x with its parent while x is larger, preserving the overall max-heap structure.

7. Argue the correctness of MAX-HEAP-INCREASE-KEY using the following loop invariant:
   At the start of each iteration of the **while** loop of lines 5-7:
   a. If both nodes PARENT(i) and LEFT(i) exist, then $A[\text{PARENT}(i)].key \geq A[\text{LEFT}(i)].key$.
   b. If both nodes PARENT(i) and RIGHT(i) exist, then $A[\text{PARENT}(i)].key \geq A[\text{RIGHT}(i)].key$.
   c. The subarray $A[1 : A.heap\text{-}size]$ satisfies the max-heap property, except that there are may be one violation, which is that $A[i].key$ may be greater than $A[\text{PARENT}(i)].key$.

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

   Solution
   a. Since the input A is a max-heap, $A[\text{PARENT}(i)].key \geq A[i].key \geq A[\text{LEFT}(i)].key$.
   b. Similarly, $A[\text{PARENT}(i)].key \geq A[i].key \geq A[\text{RIGHT}(i)].key$.
   c. Since the input A is a max-heap, $x.key$ is larger than its children. In line 3, the value of $x.key$ is increased, so it is still larger than its children. It is possible that $x.key$ becomes larger than its parent, which violates the max-heap property. No other node is affected because a max-heap only requires the relation between adjacent nodes.

8. Each exchange operation on line 6 of MAX-HEAP-INCREASE-KEY typically requires three assignments, not counting the updating of the mapping from objects to array indices. Show how to use the idea of the inner loop of INSERTION-SORT to reduce the three assignments to just one assignment.

   Solution Instead of exchanging $A[i]$ with $A[\text{PARENT}(i)]$ in every loop, we save a copy of x, x_copy, before the loop, and just assign $A[\text{PARENT}(i)]$ to $A[i]$ in each loop. This method keeps the correctness of every node except $A[i]$ after the last iteration, and we assign it with x_copy. Also, in the condition of the **while** loop, $A[i].key$ should be replaced by $x\_copy.key$. The following pseudocode shows this adjustment of the algorithm MAX-HEAP-INCREASE-KEY with only one assignment in each iteration of the **while** loop:

Algorithm 8 MAX-HEAP-INCREASE-KEY(A, x, k)

1: **if** $k > x.key$ **then**
2: **error** "new key is smaller than current key"
3: **end if**
4: $x.key = k$
5: find the index i in array A where object x occurs
6: $x\_copy = x$
7: **while** $i > 1$ and $A[\text{PARENT}(i)].key < x\_copy.key$ **do**
8: Assign $A[\text{PARENT}(i)]$ to $A[i]$, updating the information that maps priority queue objects to array indices
9: $i = \text{PARENT}(i)$
10: **end while**
11: $A[i] = x\_copy$

9. 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 10.1.3.)
Solution Unfinished.

10. The operation MAX-HEAP-DELETE(A, x) deletes the object x from max-heap A. Give an implementation of MAX-HEAP-DELETE for an n-element max heap that runs in $O(\lg n)$ time plus the overhead for mapping priority queue objects to array indices.
Solution The key idea is to find the index i where x occurs, assign $A[A.heap\text{-}size]$ to $A[i]$, remove $A[A.heap\text{-}size]$ by decreasing the $A.heap\text{-}size$ by 1, and finally call MAX-HEAPIFY or MAX-HEAP-INCREASE-KEY to rebuild the max-heap. Below is the pseudocode:

Algorithm 9 MAX-HEAP-DELETE(A, x)

1: find the index i in array A where object x occurs
2: assign $A[A.heap\text{-}size]$ to $A[i]$, updating the information that maps priority queue objects to array indices
3: $A.heap\text{-}size = A.heap\text{-}size - 1$
4: **if** $A[i] < A[\text{parent}(i)]$ **then**
5: MAX-HEAPIFY(A, i)
6: **else**
7: MAX-HEAP-INCREASE-KEY$(A, A[i], A[i].key)$
8: **end if**

Since MAX-HEAPIFY(A, i) costs time $O(\lg n)$, the running time of MAX-HEAP-DELETE is $O(\lg n)$ plus the overhead for mapping priority queue objects to array indices.

11. 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.)
Solution First, we take the first elements of all the k sorted lists, and call BUILD-MIN-HEAP to build a max heap with these k elements. Then, we repeatedly call MIN-HEAP-EXTRACT-MIN (See Exercise 6.5-3) to collect the smallest element and put it to the sorted list, and call MIN-HEAP-INSERT (See Exercise 6.5-3) to insert the first element of the one among the k lists that the smallest element collected belongs to. Below is the pseudocode:

Algorithm 10 MERGE-SORTED-LISTS(L, x)

```
1: for i = 1 to k do
2:     H[i] = L[i][1]
3:     I[i] = 1
4: end for
5: BUILD-MIN-HEAP(H, k)
6: for i = 1 to n do
7:     A[i] = MIN-HEAP-EXTRACT-MIN(H)
8:     H.heap-size = H.heap-size − 1
9:     if L[A[i].group].size − I[A[i].group] ≥ 1 then
10:        new = L[A[i].group][I[A[i].group] + 1]
11:        MIN-HEAP-INSERT(H, new)
12:        I[A[i].group] = I[A[i].group] + 1
13:    end if
14: end for
15: return A
```

In this pseudocode, L is a list consisting of all the k sorted lists. We equip each x another attribute $x.group$, ranging from 1 to k, which shows where in the k lists x is come from. In other words, x is in $L[x.group]$. We denote $K.size$ as the number of elements in a list K. Now, let us compute the running time of our algorithm. The first **for** loop costs $O(k)$, followed by BUILD-MIN-HEAP(H, k) costing $O(k)$. In each of the n iterations of the second **for** loop, MIN-HEAP-EXTRACT-MIN(H) costs $O(\lg k)$. Inside the **if** statement, MIN-HEAP-INSERT(H, new) costs $O(\lg k)$, and every other line costs constant time. The second **for** loop therefore costs $O(n \lg k)$ in total. Hence, the running time of MERGE-SORTED-LISTS is $O(k) + O(k) + O(n \lg k) = O(n \lg k)$.

Problems in Chapter 6

1. **Building a heap using insertion**

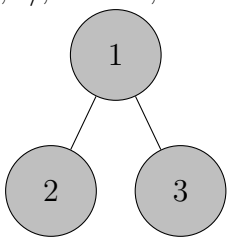
   One way to build a heap is by repeatedly calling MAX-HEAP-INSERT to insert the elements into the heap. Consider the procedure BUILD-MAX-HEAP' on the facing page. It assumes that the objects being inserted are just the heap elements.

Algorithm 11 BUILD-MAX-HEAP'

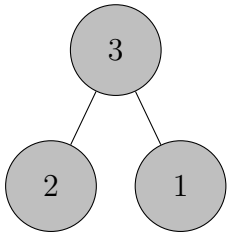
```
1: A.heap-size = 1
2: for i = 2 to n do
3:     MAX-HEAP-INSERT(A, A[i], n)
4: end for
```

   a. Do the procedure BUILD-MAX-HEAP and BUILD-MAX-HEAP' always create the same heap when run on the same input array? Prove that they do, or provide a counterexample.

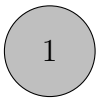
   Solution No, they may create different max-heaps even when run on the same input array. Consider the array $A = \langle 1, 2, 3 \rangle$, that is, in tree representation,



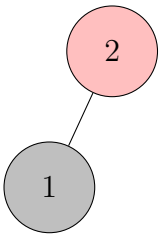
   Calling BUILD-MAX-HEAP(A) will do nothing but exchange $A[1]$ with $A[3]$, creating the max-heap $A = \langle 3, 2, 1 \rangle$:



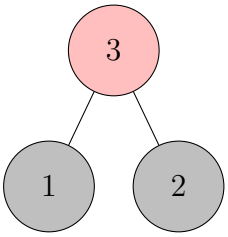
   On the other hand, if we call BUILD-MAX-HEAP(A)', the max-heap starts with only one element $A[1] = 1$:



   Then, MAX-HEAP-INSERT($A, A[2], n$) inserts $A[2] = 2$ and put it to the right place:



Lastly, MAX-HEAP-INSERT$(A, A[3], n)$ inserts $A[3] = 3$ and put it to the right place, resulting the max-heap $A = \langle 3, 1, 2 \rangle$:



b. Show that in the worst case, MAX-HEAP-INSERT' requires $\Theta(n \lg n)$ time to build an n-element heap.

Solution Consider the input array A is in increasing order, that is, in each iteration of the **for** loop, $A[i]$ is larger than any element existing in the max-array. Therefore, there will be $\lfloor \lg i \rfloor$ exchanges in each iteration. The total number of exchanges is $\sum_{i=2}^{n} \lfloor \lg i \rfloor$. We have

$$\sum_{i=2}^{n} \lfloor \lg i \rfloor \leq (n-1)(\lfloor \lg n \rfloor) = O(n \lg n),$$

and

$$\begin{aligned}
\sum_{i=2}^{n} \lfloor \lg i \rfloor &\geq \sum_{i=2^{\lfloor \lg n \rfloor - 1}}^{n} \lfloor \lg i \rfloor \\
&\geq (n - 2^{\lfloor \lg n \rfloor - 1} + 1) \lfloor \lg 2^{\lfloor \lg n \rfloor - 1} \rfloor \\
&\geq \frac{n}{2} (\lfloor \lg n \rfloor - 1) \\
&= \Omega(n \lg n).
\end{aligned}$$

Hence, the worst-case running case of MAX-HEAP-INSERT' is $\Theta(n \lg n)$.

2. **Analysis of d-ary heaps**

A **d-ary heap** is like a binary heap, but (with one possible exception) nonleaf nodes have d children instead of two children. In all parts of this problem, assume that the time to maintain the mapping between objects and heap elements is $O(1)$ per operation.

a. Describe how to represent a d-ary heap in an array.

Solution Given a d-ary heap A with n nodes, let $A[1]$ be the root of A, and let $A[2]$ to $A[d+1]$ be the children of $A[1]$ from left to right, let $A[d+2]$ to $A[2d+1]$ be the

children of $A[2]$ from left to right, repeat this procedure until all nodes are added into the array A, which is then an array representation of a d-ary heap.

b. Using Θ-notation, express th height of a d-ary heap of n elements in terms of n and d.
Solution Similar to Exercise 6.1.2, the height of a d-ary heap is $\Theta(\log_d n)$.

c. Give an efficient implementation of EXTRACT-MAX in a d-ary max-heap. Analyze its running time in terms of d and n.
Solution Note that the children of $A[i]$ are indexed from $(n-1)d+2$ to $nd+1$.

Algorithm 12 MAX-HEAP-MAXIMUM(A)

```
1: if A.heap-size < 1 then
2:    error "heap underflow"
3: end if
4: return A[1]
```

Algorithm 13 MAX-HEAPIFY(A, d, i)

```
1: largest = i
2: for j = (n − 1)d + 2 to nd + 1 do
3:    if j ≤ A.heap-size and A[j] > A[largest] then
4:       largest = j
5:    end if
6: end for
7: if largest ≠ i then
8:    exchange A[i] with A[largest]
9:    MAX-HEAPIFY(A, d, largest)
10: end if
```

Algorithm 14 EXTRACT-MAX(A, d)

```
1: max = MAX-HEAP-MAXIMUM(A)
2: A[1] = A[A.heap-size]
3: A[A.heap-size] = A[A.heap-size] - 1
4: MAX-HEAPIFY(A, d, 1)
5: return max
```

Every line in EXTRACT-MAX(A) takes constant time except for line 4, where MAX-HEAPIFY(A, i) is called. The **for** loop in lines 2-6 in MAX-HEAPIFY(A, i) takes $O(d)$, and there are $O(\log_d n)$ iterations for the recursion. Therefore, the running time of MAX-HEAPIFY(A, i) is $O(d \log_d n)$, so is the running time of EXTRACT-MAX(A).

d. Give an efficient implementation of INCREASE-KEY in a d-ary max-heap. Analyze its running time in terms of d and n.

Solution Note that the parent of $A[i]$ is indexed $\lceil (i-1)/d \rceil$.

Algorithm 15 PARENT(d, i)

```
1: return ⌈(i − 1)/d⌉
```

Algorithm 16 INCREASE-KEY(A, d, x, k)

```
1: if k < x.key then
2:     error "new key is smaller than current key"
3: end if
4: x.key = k
5: find the index i in A where object x occurs
6: while i > 1 and A[PARENT(d, i)].key < A[i].key do
7:     exchange A[i] and A[PARENT(d, i)], updating the information that maps priority queue
       objects to array indices.
8:     i = PARENT(d, i)
9: end while
```

The running time of INCREASE-KEY is $O(\log_d n)$ because the path from x to the root has length $O(\log_d n)$.

e. Give an efficient implementation of INSERT in a d-ary max-heap. Analyze its running time in terms of d and n.

Solution

Algorithm 17 INSERT(A, d, x, n)

```
1: if A.heap-size == n then
2:     error "heap overflow"
3: end if
4: A.heap-size = A.heap-size + 1
5: k = x.key
6: x.key = −∞
7: A[A.heap-size] = x
8: map x to index heap-size in the array
9: INCREASE-KEY(A, d, x, k)
```

The running time of INSERT equals the running time of INCREASE-KEY plus the time to map x to index *heap-size*, which we assumed as $O(1)$. Hence, the running time of INSERT is $O(\log_d n)$.

3. **Young tableaus**

An $m \times n$ **Young tableau** is an $m \times n$ matrix such that the entries of each row are in sorted order from left to right and the entries of each column are in sorted order from top to bottom. Some of the entries of a Young tableau may be ∞, which we treat as nonexistent elements. Thus, a Young tableau can be used to hold $r \leq mn$ finite numbers.

a. Draw a 4×4 Young tableau containing the elements $\{9, 16, 3, 2, 4, 8, 5, 14, 12\}$.

Solution

$$\begin{bmatrix} 2 & 3 & 5 & \infty \\ 4 & 9 & 14 & \infty \\ 8 & 12 & 16 & \infty \\ \infty & \infty & \infty & \infty \end{bmatrix}$$

b. Argue that an $m \times n$ Young tableau Y is empty if $Y[1,1] = \infty$. Argue that Y is full (contains mn elements) if $Y[m,n] < \infty$.

Solution It is easy to see that $Y[1,1]$ is the smallest element in a Young tableau, while $Y[m,n]$ is the largest element in a Young tableau. Therefore, if $Y[1,1] = \infty$, every entry must be ∞, that is, Y is empty. On the other hand, if $Y[m,n] < \infty$ is finite, then every entry must also be finite, and therefore Y is full.

c. Give an algorithm to implement EXTRACT-MIN on a non-empty $m \times n$ Young tableau that runs in $O(m+n)$ time. Your algorithm should use a recursive subroutine that solves an $m \times n$ problem by recursively solving either an $(m-1) \times n$ or an $m \times (n-1)$ subproblem. (*Hint:* Think about MAX-HEAPIFY.) Explain why your implementation of EXTRACT-MIN runs in $O(m+n)$ time.

Solution

Algorithm 18 MINIMUM(Y)

```
1: if Y[1,1] = ∞ then
2:     error "Young tableau underflow"
3: end if
4: return Y[1,1]
```

Algorithm 19 EXTRACT-MIN-RECURSION(Y, m, n, i, j)

```
1: if i < m and Y[i+1, j] < Y[i, j] then
2:     smallest = 1
3: else
4:     smallest = 0
5: end if
6: if j < n and Y[i, j+1] < Y[i, j] then
7:     smallest = 2
8: end if
9: if smallest = 0 then
10:    return
11: end if
12: if smallest = 1 then
13:    exchange Y[i, j] with Y[i+1, j]
14:    EXTRACT-MIN-RECURSION(Y, m, n, i+1, j)
15: else
16:    exchange Y[i, j] with Y[i, j+1]
17:    EXTRACT-MIN-RECURSION(Y, m, n, i, j+1)
18: end if
```

Algorithm 20 EXTRACT-MIN(Y, m, n)

```
1: min = MINIMUM(Y)
2: Y[1, 1] = Y[m, n]
3: EXTRACT-MIN-RECURSION(Y, m, n, 1, 1)
4: return min
```

In EXTRACT-MIN-RECURSION, we reduce the $m \times n$ matrix to an $(m-1) \times n$ or an $m \times (n-1)$ submatrix, and the recursion terminates only when the reduced submatrix is a Youth tableau.

There are at most $m + n - 1 = O(m + n)$ iterations in EXTRACT-MIN-RECURSION because if an iteration does not return, it increases either i or j by 1, that is, increases $i + j$ by 1. The initial value of $i + j$ is 2, and if $i + j = m + n$ at the beginning of an iteration, it must terminate in this iteration. Hence, the running time of EXTRACT-MIN is $O(m + n)$.

d. Show how to insert a new element into a nonfull $m \times n$ Young tableau in $O(m+n)$ time.
Solution If a Young tableau Y is nonfull, then $Y[m,n] = \infty$ (See b.), which means there is no element of index (m,n). So, we can safely set the new element x to $Y[m,n]$.

Algorithm 21 INSERT-RECURSION(Y, m, n, i, j)

```
1: if i > 1 and Y[i − 1, j] > Y[i, j] then
2:    largest = 1
3: else
4:    largest = 0
5: end if
6: if j > 1 and Y[i, j − 1] > Y[i, j] then
7:    largest = 2
8: end if
9: if largest = 0 then
10:   return
11: end if
12: if largest = 1 then
13:   exchange Y[i, j] with Y[i − 1, j]
14:   INSERT-RECURSION(Y, m, n, i − 1, j)
15: else
16:   exchange Y[i, j] with Y[i, j − 1]
17:   INSERT-RECURSION(Y, m, n, i, j − 1)
18: end if
```

Algorithm 22 INSERT(Y, m, n, x)

```
1: Y[m, n] = x
2: INSERT-RECURSION(Y, m, n, m, n)
```

This algorithm is very similar to the one in c. and it has the same running time $O(m+n)$.

e. Using no other sorting method as a subroutine, show how to use an $n \times n$ Young tableau to sort n^2 numbers in $O(n^3)$ time.
Solution First, we initialize an $n \times n$ Youth tableau Y with all entries ∞ (that is, an empty $n \times n$ matrix). Then, we call INSERT to insert all n^2 numbers into Y. Lastly, we call EXTRACT-MIN to build the sorted array.

Algorithm 23 YOUNG-TABLEAU-SORT(A)

```
1: for i = 1 to n^2 do
2:     Y[i] = ∞
3: end for
4: for i = 1 to n^2 do
5:     INSERT(Y, n, n, A[i])
6: end for
7: for i = 1 to n^2 do
8:     A[i] = EXTRACT-MIN(Y, n, n)
9: end for
```

Both INSERT($Y, n, n, A[i]$) and EXTRACT-MIN(Y, n, n) takes $O(n + n) = O(n)$ time. Therefore, the running time of YOUNG-TABLEAU-SORT(A), where A has size n^2, is $O(n^2 O(n)) = O(n^3)$.

f. Give an $O(m + n)$-time algorithm to determine whether a given number is stored in a given $m \times n$ Young tableau.

Solution The program starts from the upper right entry of the Young tableau Y, and eliminate either a row or a column from Y, depending on the relation between the upper right element and the given number.

Algorithm 24 YOUTH-TABLEAU-SEARCH-RECURSION(Y, m, n, x, i, j)

```
1: if Y[i, j] == x then
2:     return True
3: end if
4: if i == m and j == 1 then
5:     return False
6: end if
7: if Y[i, j] < x then
8:     return YOUTH-TABLEAU-SEARCH-RECURSION(Y, m, n, i + 1, j)
9: else
10:    return YOUTH-TABLEAU-SEARCH-RECURSION(Y, m, n, i, j − 1)
11: end if
```

Algorithm 25 YOUTH-TABLEAU-SEARCH(Y, m, n, x)

```
1: YOUTH-TABLEAU-SEARCH-RECURSION(Y, m, n, x, 1, n)
```

Since we eliminate a row or a column in each iteration of YOUTH-TABLEAU-SEARCH-RECURSION, there are $O(m+n)$ iterations before the given number is found (or unfound). Hence, the running time of YOUTH-TABLEAU-SEARCH is $O(m+n)$.