6 Heapsort

6.1 Heaps

6.1-1

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

At least 2^h and at most $2^{h+1} - 1$. Can be seen because a complete binary tree of depth $h - 1$ has $\sum_{i=0}^{h-1} 2^i = 2^h - 1$ elements, and the number of elements in a heap of depth h is between the number for a complete binary tree of depth $h - 1$ exclusive and the number in a complete binary tree of depth h inclusive.

6.1-2

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

Write $n = 2^m - 1 + k$ where m is as large as possible. Then the heap consists of a complete binary tree of height $m - 1$, along with k additional leaves along the bottom. The height of the root is the length of the longest simple path to one of these k leaves, which must have length m. It is clear from the way we defined m that $m = \lfloor \lg n \rfloor$.

6.1-3

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

If the largest element in the subtree were somewhere other than the root, it has a parent that is in the subtree. So, it is larger than it's parent, so, the heap property is violated at the parent of the maximum element in the subtree.

6.1-4

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

In any of the leaves, that is, elements with index $\lfloor n/2 \rfloor + k$, where $k \geq 1$ (see exercise 6.1-7), that is, in the second half of the heap array.

6.1-5

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

Yes. For any index i, both $\text{LEFT}(i)$ and $\text{RIGHT}(i)$ are larger and thus the elements indexed by them are greater or equal to $A[i]$ (because the array is sorted.)

6.1-6

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

No. Since $\text{PARENT}(7)$ is 6 in the array. This violates the max-heap property.

6.1-7

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

Let's take the left child of the node indexed by $\lfloor n/2 \rfloor + 1$.

$$\begin{aligned}\text{LEFT}(\lfloor n/2 \rfloor + 1) &= 2(\lfloor n/2 \rfloor + 1) \\ &> 2(n/2 - 1) + 2 \\ &= n - 2 + 2 \\ &= n.\end{aligned}$$

Since the index of the left child is larger than the number of elements in the heap, the node doesn't have childrens and thus is a leaf. Same goes for all nodes with larger indices.

Note that if we take element indexed by $\lfloor n/2 \rfloor$, it will not be a leaf. In case of even number of nodes, it will have a left child with index n and in the case of odd number of nodes, it will have a left child with index $n - 1$ and a right child with index n.

This makes the number of leaves in a heap of size n equal to $\lceil n/2 \rceil$.

6.2 Maintaining the heap property

6.2-1

> Using figure 6.2 as a model, illustrate the operation of $\text{MAX-HEAPIFY}(A, 3)$ on the array $A = \langle 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0 \rangle$.

$$\begin{gathered}\langle 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0 \rangle \\ \langle 27, 17, 10, 16, 13, 3, 1, 5, 7, 12, 4, 8, 9, 0 \rangle \\ \langle 27, 17, 10, 16, 13, 9, 1, 5, 7, 12, 4, 8, 3, 0 \rangle\end{gathered}$$

6.2-2

> Starting with the procedure MAX-HEAPIFY, write pseudocode for the procedure $\text{MIN-HEAPIFY}(A, i)$, which performs the corresponding manipulation on a min-heap. How does the running time of MIN-HEAPIFY compare to that of MAX-HEAPIFY?

```
MIN-HEAPIFY(A, i)
    l = LEFT(i)
    r = RIGHT(i)
    if l ≤ A.heap-size and A[l] < A[i]
```

```
        smallest = l
    else smallest = i
    if r ≤ A.heap-size and A[r] < A[smallest]
        smallest = r
    if smallest != i
        exchange A[i] with A[smallest]
        MIN-HEAPIFY(A, smallest)
```

The running time is the same. Actually, the algorithm is the same with the exceptions of two comparisons and some names.

6.2-3

> What is the effect of calling $\text{MAX-HEAPIFY}(A, i)$ when the element $A[i]$ is larger than its children?

No effect. The comparisons are carried out, $A[i]$ is found to be largest and the procedure just returns.

6.2-4

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

No effect. In that case, it is a leaf. Both LEFT and RIGHT return values that fail the comparison with the heap size and i is stored in largest. Afterwards the procedure just returns.

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.

```
MAX-HEAPIFY(A, i)
    while true
        l = LEFT(i)
        r = RIGHT(i)
        if l ≤ A.heap-size and A[l] > A[i]
            largest = l
        else largest = i
        if r ≤ A.heap-size and A[r] > A[largest]
            largest = r
        if largest == i
            return
        exchange A[i] with A[largest]
        i = largest
```

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.)

Consider the heap resulting from A where $A[1] = 1$ and $A[i] = 2$ for $2 \le i \le n$. Since 1 is the smallest element of the heap, it must be swapped through each level of the heap until it is a leaf node. Since the heap has height $\lfloor \lg n \rfloor$, MAX-HEAPIFY has worst-case time $\Omega(\lg n)$.

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 $A = \langle 5, 3, 17, 10, 84, 19, 6, 22, 9 \rangle$.

$$\begin{gathered}
\langle 5, 3, 17, 10, 84, 19, 6, 22, 9 \rangle \\
\langle 5, 3, 17, 22, 84, 19, 6, 10, 9 \rangle \\
\langle 5, 3, 19, 22, 84, 17, 6, 10, 9 \rangle \\
\langle 5, 84, 19, 22, 3, 17, 6, 10, 9 \rangle \\
\langle 84, 5, 19, 22, 3, 17, 6, 10, 9 \rangle \\
\langle 84, 22, 19, 5, 3, 17, 6, 10, 9 \rangle \\
\langle 84, 22, 19, 10, 3, 17, 6, 5, 9 \rangle
\end{gathered}$$

6.3-2

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

Otherwise we won't be allowed to call MAX-HEAPIFY, since it will fail the condition of having the subtrees be max-heaps. That is, if we start with 1, there is no guarantee that $A[2]$ and $A[3]$ are roots of max-heaps.

6.3-3

> Show that there are at most $\lceil n/2^{h+1} \rceil$ nodes of height 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 $\lceil n/2 \rceil$. Let's prove by induction. Let n_h denote the number of nodes at height h. The upper bound holds for the base since $n_0 = \lceil n/2 \rceil$ is exactly the number of leaves in a heap of size n.

Now assume it holds for $h - 1$. We have 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 $n_h = n_{h-1}/2 = \lfloor n_{h-1}/2 \rfloor$. 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

$$\begin{aligned} n_h &= \left\lceil \frac{n_{h-1}}{2} \right\rceil \\ &\le \left\lceil \frac{1}{2} \cdot \left\lceil \frac{n}{2^{(h-1)+1}} \right\rceil \right\rceil \\ &= \left\lceil \frac{1}{2} \cdot \left\lceil \frac{n}{2^h} \right\rceil \right\rceil \\ &= \left\lceil \frac{n}{2^{h+1}} \right\rceil , \end{aligned}$$

which implies that it holds for h.

6.4 The heapsort algorithm

6.4-1

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

$$\langle 5, 13, 2, 25, 7, 17, 20, 8, 4\rangle$$
$$\langle 5, 13, 20, 25, 7, 17, 2, 8, 4\rangle$$
$$\langle 5, 25, 20, 13, 7, 17, 2, 8, 4\rangle$$
$$\langle 25, 5, 20, 13, 7, 17, 2, 8, 4\rangle$$
$$\langle 25, 13, 20, 5, 7, 17, 2, 8, 4\rangle$$
$$\langle 25, 13, 20, 8, 7, 17, 2, 5, 4\rangle$$
$$\langle 4, 13, 20, 8, 7, 17, 2, 5, 25\rangle$$
$$\langle 20, 13, 4, 8, 7, 17, 2, 5, 25\rangle$$
$$\langle 20, 13, 17, 8, 7, 4, 2, 5, 25\rangle$$
$$\langle 5, 13, 17, 8, 7, 4, 2, 20, 25\rangle$$
$$\langle 17, 13, 5, 8, 7, 4, 2, 20, 25\rangle$$
$$\langle 2, 13, 5, 8, 7, 4, 17, 20, 25\rangle$$
$$\langle 13, 2, 5, 8, 7, 4, 17, 20, 25\rangle$$
$$\langle 13, 8, 5, 2, 7, 4, 17, 20, 25\rangle$$
$$\langle 4, 8, 5, 2, 7, 13, 17, 20, 25\rangle$$
$$\langle 8, 4, 5, 2, 7, 13, 17, 20, 25\rangle$$
$$\langle 8, 7, 5, 2, 4, 13, 17, 20, 25\rangle$$
$$\langle 4, 7, 5, 2, 8, 13, 17, 20, 25\rangle$$
$$\langle 7, 4, 5, 2, 8, 13, 17, 20, 25\rangle$$
$$\langle 2, 4, 5, 7, 8, 13, 17, 20, 25\rangle$$
$$\langle 5, 4, 2, 7, 8, 13, 17, 20, 25\rangle$$
$$\langle 2, 4, 5, 7, 8, 13, 17, 20, 25\rangle$$
$$\langle 4, 2, 5, 7, 8, 13, 17, 20, 25\rangle$$
$$\langle 2, 4, 5, 7, 8, 13, 17, 20, 25\rangle$$

6.4-2

> Argue the correctness of HEAPSORT using the following loop invariant:
>
> At the start of each iteration of the **for** loop of lines 2-5, the subarray $A[1..i]$ is 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.

Initialization: The subarray $A[i+1..n]$ is empty, thus the invariant holds.

Maintenance: $A[1]$ is the largest element in $A[1..i]$ and it is smaller than the elements in $A[i+1..n]$. When we put it in the ith position, then $A[i..n]$ contains the largest elements, sorted. Decreasing the heap size and calling MAX-HEAPIFY turns $A[1..i-1]$ into a max-heap. Decrementing i sets up the invariant for the next iteration.

Termination: After the loop $i = 1$. This means that $A[2..n]$ is sorted and $A[1]$ is the smallest element in the array, which makes the array 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?

Both of them are $\Theta(n \lg n)$.

If the array is sorted in increasing order, the algorithm will need to convert it to a heap that will take $O(n)$. Afterwards, however, there are $n - 1$ calls to MAX-HEAPIFY and each one will perform the full $\lg k$ operations. Since:

$$\sum_{k=1}^{n-1} \lg k = \lg((n-1)!) = \Theta(n \lg n).$$

Same goes for decreasing order. BUILD-MAX-HEAP will be faster (by a constant factor), but the computation time will be dominated by the loop in HEAPSORT, which is $\Theta(n \lg n)$.

6.4-4

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

This is essentially the first part of exercise 6.4-3. Whenever we have an array that is already sorted, we take linear time to convert it to a max-heap and then $n \lg n$ time to sort it.

6.4-5 ⋆

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

This proved to be quite tricky. My initial solution was wrong. Also, heapsort appeared in 1964, but the lower bound was proved by Schaffer and Sedgewick in 1992. It's evil to put this an exercise.

Let's assume that the heap is a full binary tree with $n = 2^k - 1$. There are 2^{k-1} leaves and $2^{k-1} - 1$ inner nodes.

Let's look at sorting the first 2^{k-1} elements of the heap. Let's consider their arrangement in the heap and color the leaves to be red and the inner nodes to be blue. The colored nodes are a subtree of the heap (otherwise there would be a contradiction). Since there are 2^{k-1} colored nodes, at most 2^{k-2} are red, which means that at least $2^{k-2} - 1$ are blue.

While the red nodes can jump directly to the root, the blue nodes need to travel up before they get removed. Let's count the number of swaps to move the blue nodes to the root. The minimal case of swaps is when

1. there are $2^{k-2} - 1$ blue nodes and
2. they are arranged in a binary tree.

If there are d such blue nodes, then there would be $i = \lg d$ levels, each containing 2^i nodes with length i. Thus the number of swaps is,

$$\sum_{i=0}^{\lg d} i2^i = 2 + (\lg d - 2)2^{\lg d} = \Omega(d \lg d).$$

And now for a lazy (but cute) trick. We've figured out a tight bound on sorting half of the heap. We have the following recurrence:

$$T(n) = T(n/2) + \Omega(n \lg n).$$

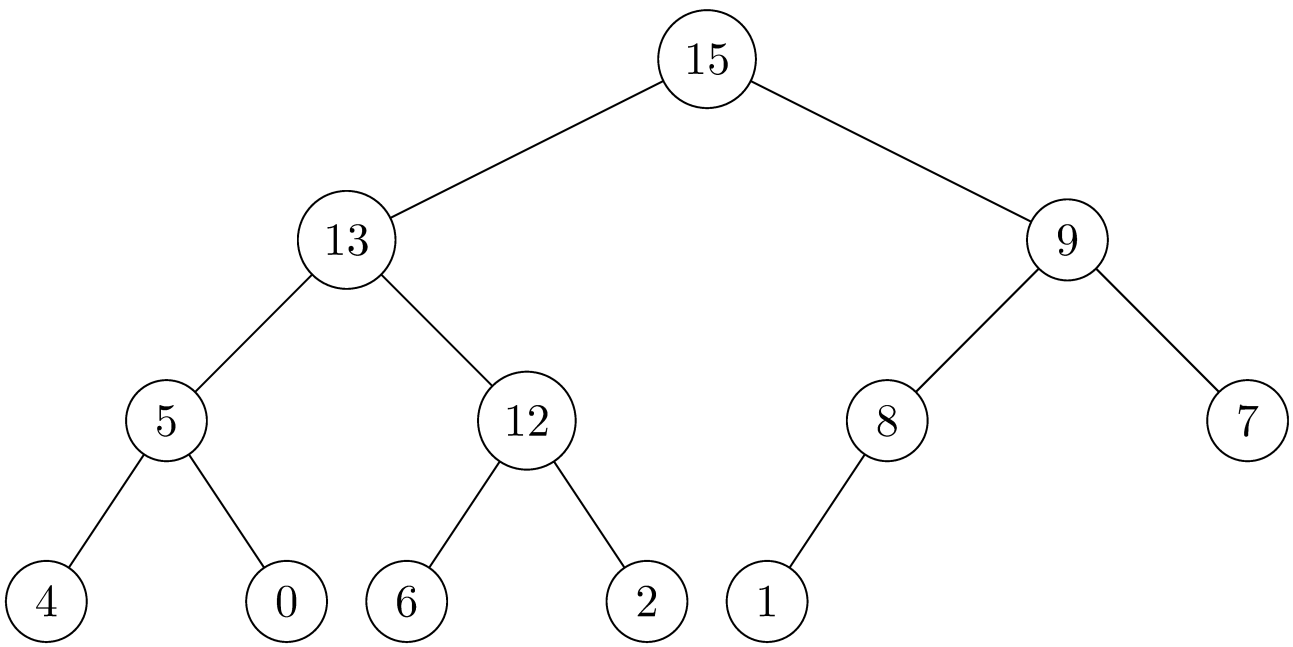
Applying the master method, we get that $T(n) = \Omega(n \lg n)$.

6.5 Priority queues

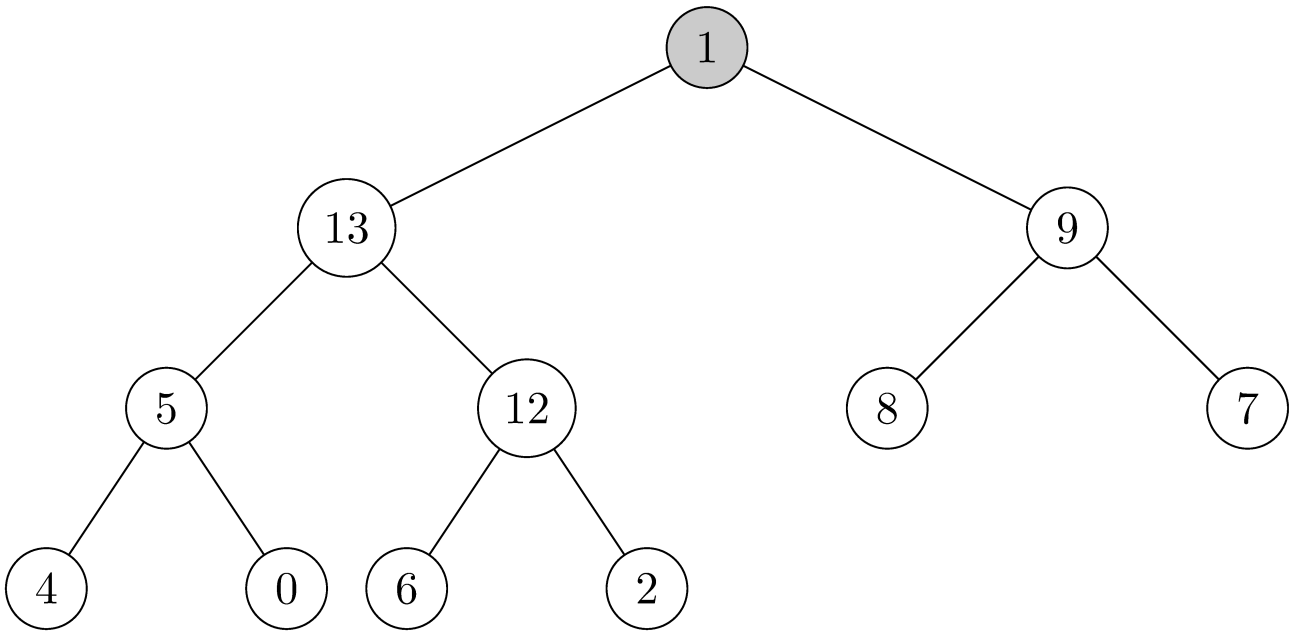
6.5-1

> Illustrate the operation HEAP-EXTRACT-MAX on the heap $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$.

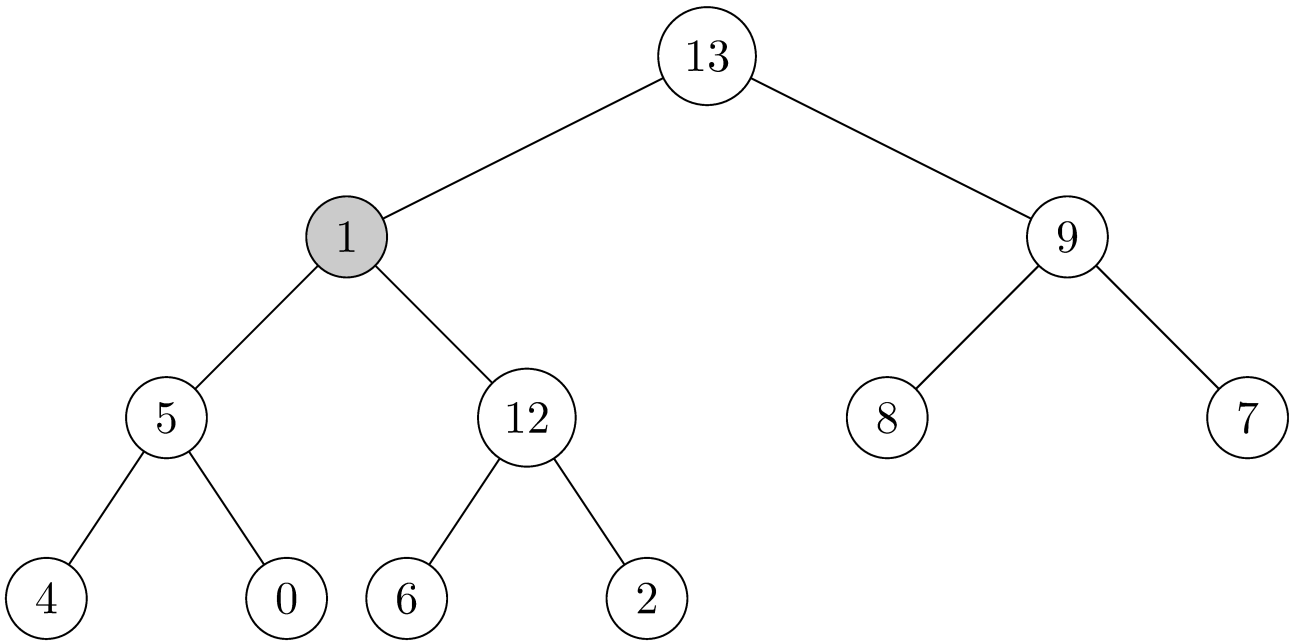
1. Original heap.



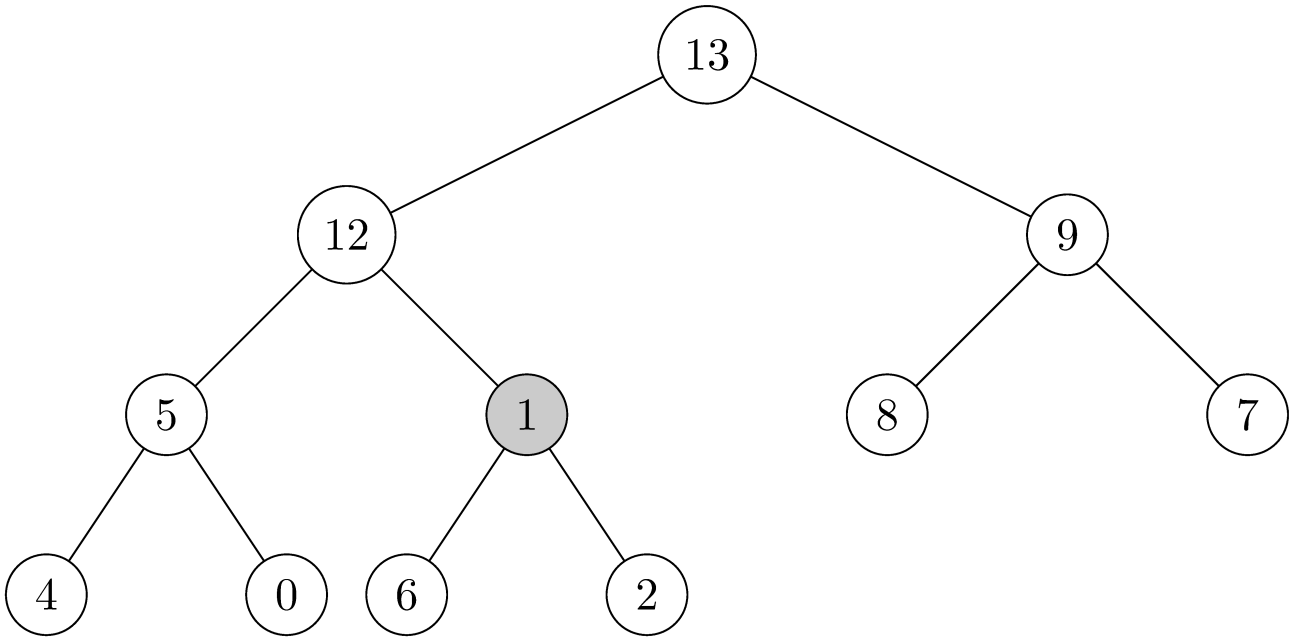
2. Extract the max node 15, then move 1 to the top of the heap.



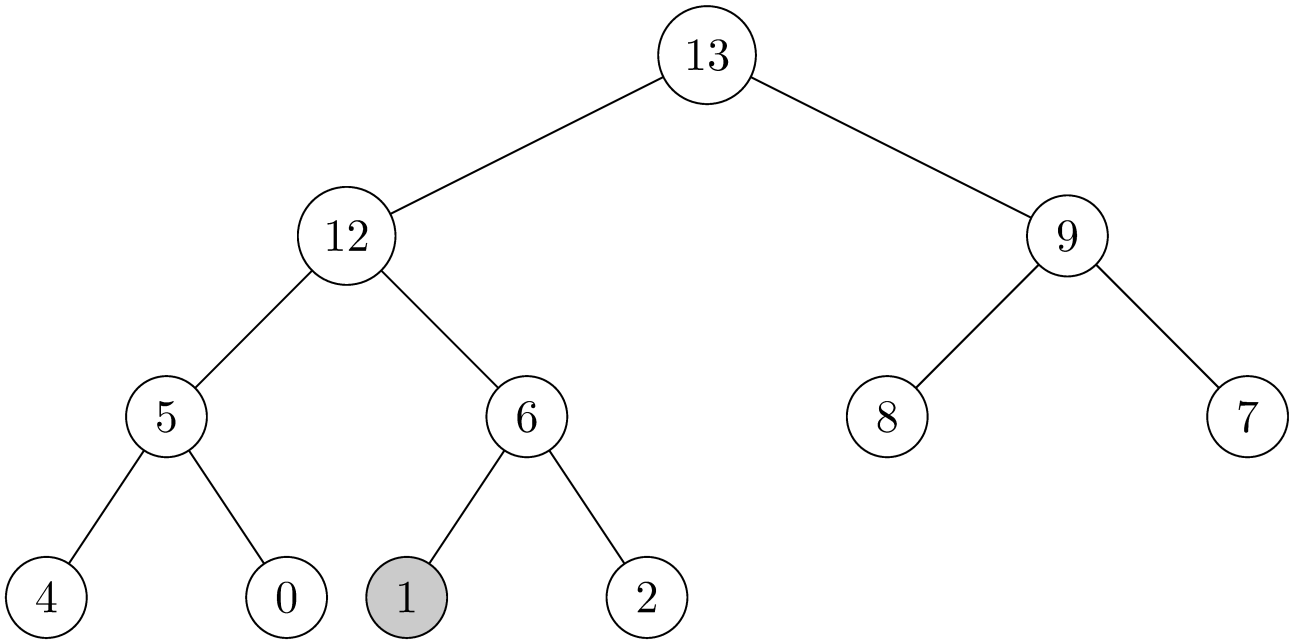
3. Since $13 > 9 > 1$, swap 1 and 13.



4. Since $12 > 5 > 1$, swap 1 and 12.



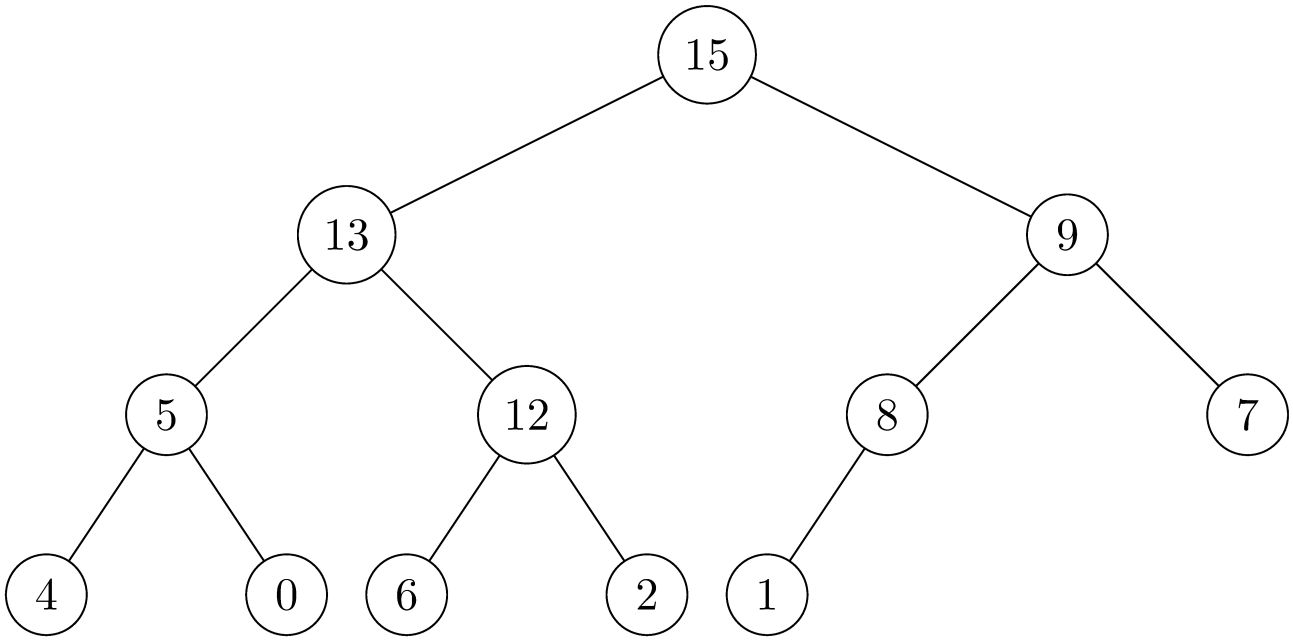
5. Since $6 > 2 > 1$, swap 1 and 6.



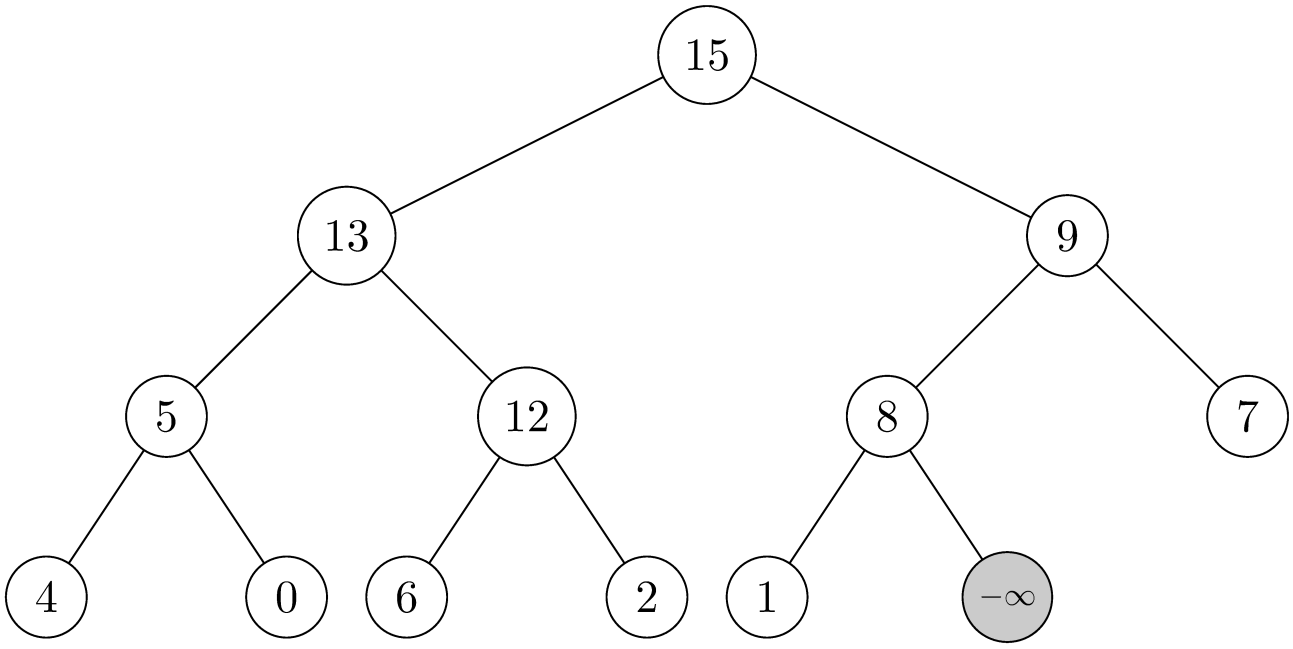
6.5-2

> Illustrate the operation of $\text{MAX-HEAP-INSERT}(A, 10)$ on the heap $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$.

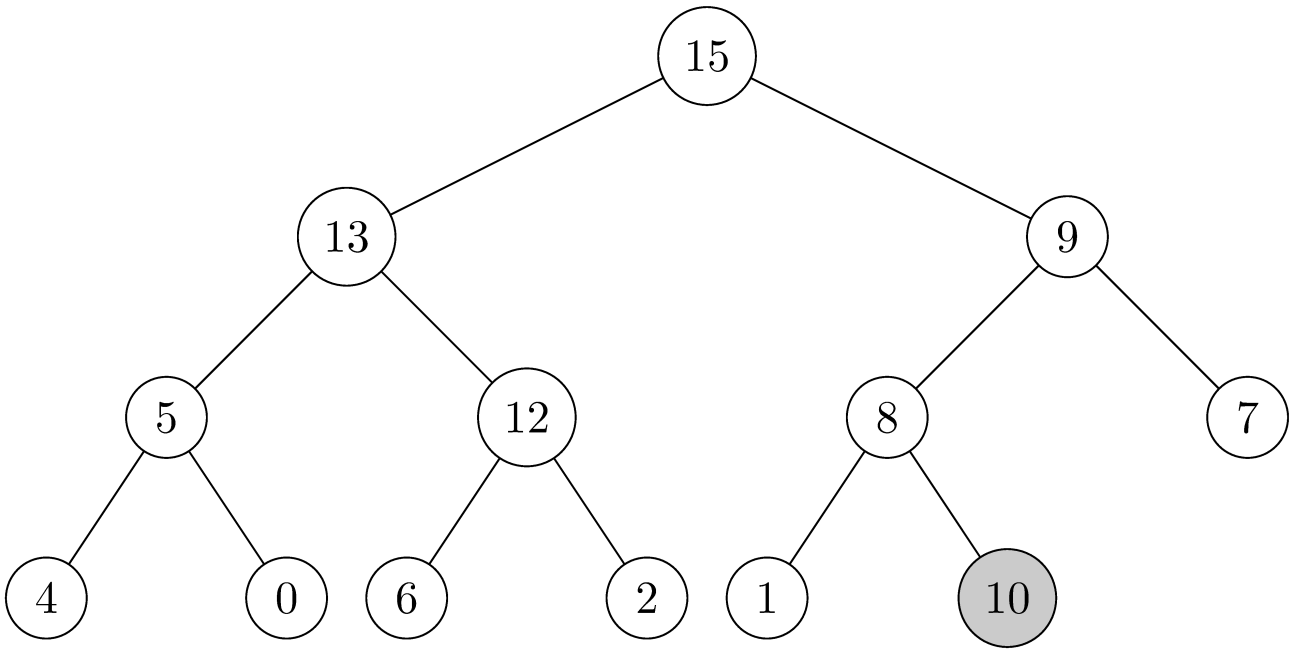
1. Original heap.



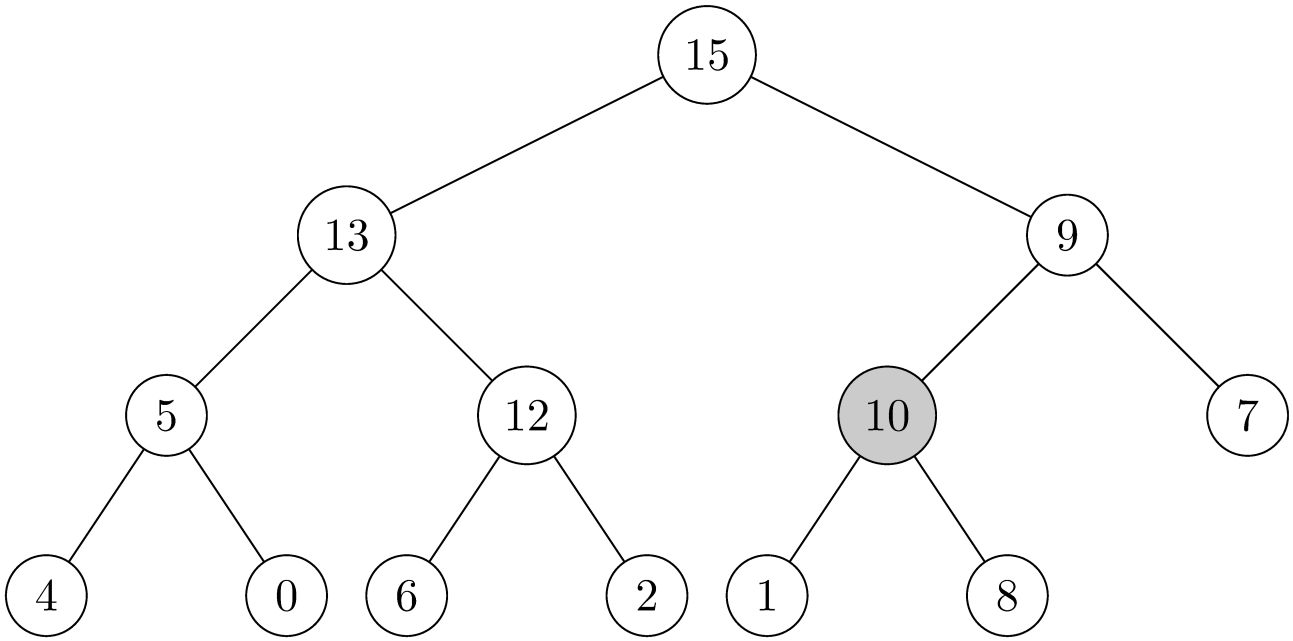
2. Since $\text{MAX-HEAP-INSERT}(A, 10)$ is called, we append a node assigned value $-\infty$.



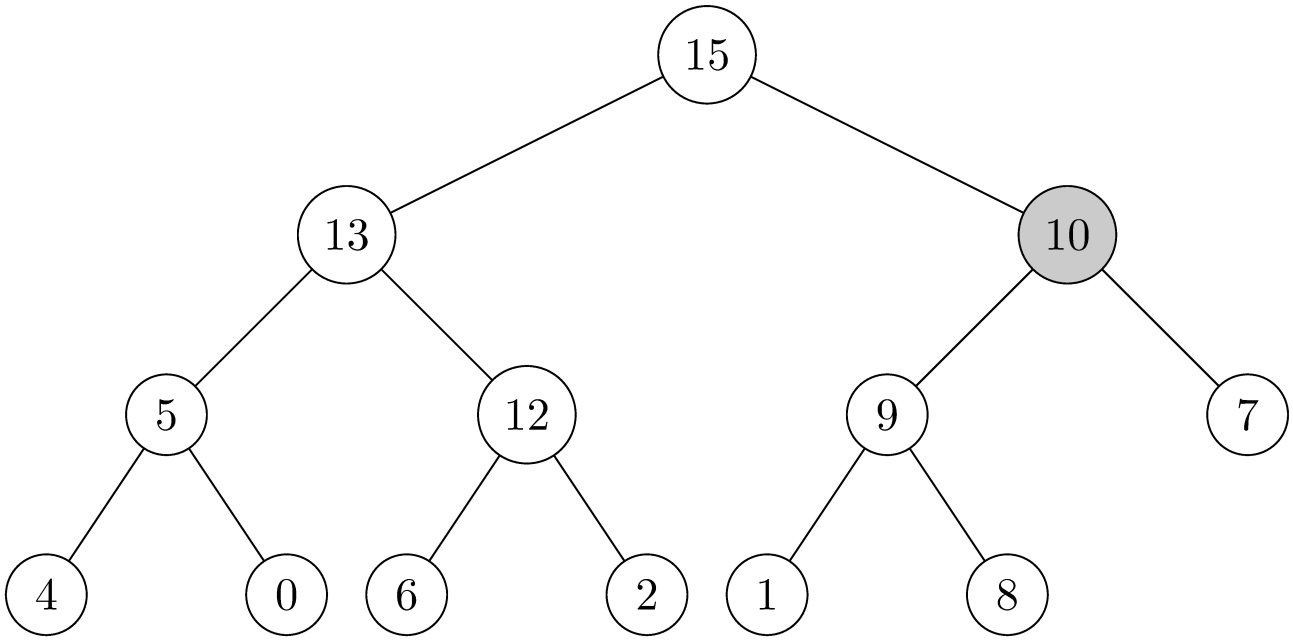
3. Update the key value of the new node.



4. Since the parent key is smaller than 10, the nodes are swapped.



5. Since the parent key is smaller than 10, the nodes are swapped.



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.

```
HEAP-MINIMUM(A)
    return A[1]
```

```
HEAP-EXTRACT-MIN(A)
    if A.heap-size < 1
        error "heap underflow"
    min = A[1]
    A[1] = A[A.heap-size]
    A.heap-size = A.heap-size - 1
    MIN-HEAPIFY(A, 1)
    return min
```

```
HEAP-DECREASE-KEY(A, i, key)
    if key > A[i]
        error "new key is larger than current key"
    A[i] = key
    while i > 1 and A[PARENT(i)] > A[i]
        exchange A[i] with A[PARENT(i)]
        i = PARENT(i)
```

```
MIN-HEAP-INSERT(A, key)
    A.heap-size = A.heap-size + 1
    A[A.heap-size] = ∞
    HEAP-DECREASE-KEY(A, A.heap-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?

In order to pass the guard clause. Otherwise we have to drop the check if $\text{key} < \text{A[i]}$.

6.5-5

> Argue the correctness of HEAP-INCREASE-KEY using the following loop invariant:
>
> At the start of each iteration of the **while** loop of lines 4-6, the subarray A[1..A.heap-size] satisfies the max-heap property, except that there may be one violation: A[i] may be larger than A[PARENT(i)].
>
> You may assume that the subarray A[1..A.heap-size] satisfies the max-heap property at the time HEAP-INCREASE-KEY is called.

Initialization: A is a heap except that A[i] might be larger that it's parent, because it has been modified. A[i] is larger than its children, because otherwise the guard clause would fail and the loop will not be entered (the new value is larger than the old value and the old value is larger than the children).

Maintenance: When we exchange A[i] with its parent, the max-heap property is satisfied except that now A[PARENT(i)] might be larger than its parent. Changing i to its parent maintains the invariant.

Termination: The loop terminates whenever the heap is exhausted or the max-heap property for A[i] and its parent is preserved. At the loop termination, A is a 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.

```
HEAP-INCREASE-KEY(A, i, key)
    if key < A[i]
        error "new key is smaller than current key"
    while i > 1 and A[PARENT(i)] < key
        A[i] = A[PARENT(i)]
```

```
        i = 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).

Both are simple. For a stack we keep adding elements in increasing priority, while in a queue we add them in decreasing priority. For the stack we can set the new priority to $\text{HEAP-MAXIMUM}(A) + 1$. For the queue we need to keep track of it and decrease it on every insertion.

Both are not very efficient. Furthermore, if the priority can overflow or underflow, so will eventually need to reassign priorities.

6.5-8

> The operation $\text{HEAP-DELETE}(A, i)$ 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.

```
HEAP-DELETE(A, i)
    if A[i] > A[A.heap-size]
        A[i] = A[A.heap-size]
        MAX-HEAPIFY(A, i)
    else
        HEAP-INCREASE-KEY(A, i, A[A.heap-size])
    A.heap-size = A.heap-size - 1
```

Note: The following algorithm is wrong. For example, given an array $A = [15, 7, 9, 1, 2, 3, 8]$ which is a max-heap, and if we delete $A[5] = 2$, then it will fail.

```
HEAP-DELETE(A, i)
    A[i] = A[A.heap-size]
    A.heap-size = A.heap-size - 1
    MAX-HEAPIFY(A, i)
```

- before:

```
        15
     /     \
    7       9
```

```
    / \     / \
   1   2   3   8
```

- after (which is wrong since $8 > 7$ violates the max-heap property):

```
        15
      /    \
     7      9
    / \    /
   1   8  3
```

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.)

We take one element of each list and put it in a min-heap. Along with each element we have to track which list we took it from. When merging, we take the minimum element from the heap and insert another element off the list it came from (unless the list is empty). We continue until we empty the heap.

We have n steps and at each step we're doing an insertion into the heap, which is $\lg k$.

Suppose that sorted lists on input are all nonempty, we have the following pseudocode.

```
def MERGE-SORTED-LISTS(lists)
    n = lists.length
    // Take the lowest element from each of lists together with an index of
the list and make list of such pairs.
    // Pairs are of "type" (element-value, index-of-list)
    let lowest-from-each be an empty array
    for i = 1 to n
        add (lists[i][0], i) to lowest-from-each
        delete lists[i][0]
    // This makes min-heap from list lowest-from-each.
    // We are sssuming that pairs of "type" (element-value, index-of-list)
are compared according to the values of elements.
    A = MIN-HEAP(lowest-from-each)
    let merged-lists be an empty array
    while not !min-heap.EMPTY()
        element-value, index-of-list = HEAP-EXTRACT-MIN(A)
        add element-value to merged-lists
        if lists[index-of-list].length > 0
            MIN-HEAP-INSERT(A, (lists[index-of-list][0], index-of-list))
            delete lists[index-of-list][0]
    return merged-lists
```

Problem 6-1 Building a heap using insertion

> We can build a heap by repeatedly calling MAX-HEAP-INSERT to insert the elements into the heap. Consider the following variation of the BUILD-MAX-HEAP procedure:
>
> ```
> BUILD-MAX-HEAP'(A)
>  A.heap-size = 1
>  for i = 2 to A.length
>  MAX-HEAP-INSERT(A, A[i])
> ```
>
> **a.** Do the procedures BUILD-MAX-HEAP and $\text{BUILD-MAX-HEAP}'$ always create the same heap when run on the same input array? Prove that they do, or provide a counterexample.
>
> **b.** Show that in the worst case, $\text{BUILD-MAX-HEAP}'$ requires $\Theta(n \lg n)$ time to build a n-element heap.

a. Consider the following counterexample.

- Input array $A = \langle 1, 2, 3 \rangle$:
- $\text{BUILD-MAX-HEAP}(A)$: $A = \langle 3, 2, 1 \rangle$.
- $\text{BUILD-MAX-HEAP}'(A)$: $A = \langle 3, 1, 2 \rangle$.

b. Each insert step takes at most $O(\lg n)$, since we are doing it n times, we get a bound on the runtime of $O(n \lg n)$.

Problem 6.2 Maintaining the heap property

> A d***-ary heap*** is like a binary heap, but (with one possible exception) non-leaf nodes have d children instead of 2 children.
>
> **a.** How would you represent a d-ary heap in an array?
>
> **b.** What is the height of a d-ary heap of n elements in terms of n and d?
>
> **c.** Give an efficient implementation of EXTRACT-MAX in a d-ary max-heap. Analyze its running time in terms of d and n.
>
> **d.** Give an efficient implementation of INSERT in a d-ary max-heap. Analyze its running time in terms of d and n.
>
> **e.** Give an efficient implementation of $\text{INCREASE-KEY}(A, i, k)$, which flags an error if $k < A[i]$, but otherwise sets $A[i] = k$ and then updates the d-ary max-heap structure appropriately. Analyze its running time in terms of d and n.

a. We can use those two following functions to retrieve parent of i-th element and j-th child of i-th element.

```
d-ARY-PARENT(i)
    return floor((i - 2) / d + 1)
```

```
d-ARY-CHILD(i, j)
    return d(i - 1) + j + 1
```

Obviously $1 \leq j \leq d$. You can verify those functions checking that

$$\text{d-ARY-PARENT}(\text{d-ARY-CHILD}(i, j)) = i.$$

Also easy to see is that binary heap is special type of d-ary heap where $d = 2$, if you substitute d with 2, then you will see that they match functions PARENT, LEFT and RIGHT mentioned in book.

b. Since each node has d children, the height of a d-ary heap with n nodes is $\Theta(\log_d n)$.

c. $\text{d-ARY-HEAP-EXTRACT-MAX}(A)$ consists of constant time operations, followed by a call to $\text{d-ARY-MAX-HEAPIFY}(A, i)$.

The number of times this recursively calls itself is bounded by the height of the d-ary heap, so the running time is $O(d \log_d n)$.

```
d-ARY-HEAP-EXTRACT-MAX(A)
    if A.heap-size < 1
        error "heap under flow"
    max = A[1]
    A[1] = A[A.heap-size]
    A.heap-size = A.heap-size - 1
    d-ARY-MAX-HEAPIFY(A, 1)
    return max
```

```
d-ARY-MAX-HEAPIFY(A, i)
    largest = i
    for k = 1 to d
        if d-ARY-CHILD(k, i) ≤ A.heap-size and A[d-ARY-CHILD(k, i)] > A[i]
            if A[d-ARY-CHILD(k, i)] > A[largest]
                largest = d-ARY-CHILD(k, i)
    if largest != i
        exchange A[i] with A[largest]
        d-ARY-MAX-HEAPIFY(A, largest)
```

d. The runtime is $O(\log_d n)$ since the **while** loop runs at most as many times as the height of the d-ary array.

```
d-ARY-MAX-HEAP-INSERT(A, key)
    A.heap-size = A.heap-size + 1
    A[A.heap-size] = key
    i = A.heap-size
    while i > 1 and A[d-ARY-PARENT(i)] < A[i]
        exchange A[i] with A[d-ARY-PARENT(i)]
        i = d-ARY-PARENT(i)
```

e. The runtime is $O(\log_d n)$ since the **while** loop runs at most as many times as the height of the d-ary array.

```
d-ARY-INCREASE-KEY(A, i, key)
    if key < A[i]
        error "new key is smaller than current key"
    A[i] = key
    while i > 1 and A[d-ARY-PARENT(i)] < A[i]
        exchange A[i] with A[d-ARY-PARENT(i)]
        i = d-ARY-PARENT(i)
```

Problem 6.3 Building a heap

> 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 4×4 tableau containing the elements $\{9, 16, 3, 2, 4, 8, 5, 14, 12\}$.
>
> **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 $Y[m, n] < \infty$.
>
> **c.** Give an algorithm to implement EXTRACT-MIN on a nonempty $m \times n$ 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.) Define $T(p)$ where $p = m + n$, to be the maximum running time of EXTRACT-MIN on any $m \times n$ Young tableau. Give and solve a recurrence relation for $T(p)$ that yields the $O(m + n)$ time bound.
>
> **d.** Show how to insert a new element into a nonfull $m \times n$ Young tableau in $O(m + n)$ time.
>
> **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.

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

a.

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

b. If the top left element is ∞, then all the elements on the first row need to be ∞. But if this is the case, all other elements need to be ∞ because they are larger than the first element on their column.

If the bottom right element is smaller than ∞, all the elements on the bottom row need to be smaller than ∞. But so are the other elements in the tableau, because each is smaller than the bottom element of its column.

c. The $A[1,1]$ is the smallest element. We store it, so we can return it later and then replace it with ∞. This breaks the Young tableau property and we need to perform a procedure, similar to MAX-HEAPIFY, to restore it.

We compare $A[i,j]$ with each of its neighbours and exchange it with the smallest. This restores the property for $A[i,j]$ but reduces the problem to either $A[i,j+1]$ or $A[i+1,j]$. We terminate when $A[i,j]$ is smaller than its neighbours.

The relation in question is

$$T(p) = T(p-1) + O(1) = T(p-2) + O(1) + O(1) = \cdots = O(p).$$

d. The algorithm is very similar to the previous, except that we start with the bottom right element of the tableau and move it upwards and leftwards to the correct position. The asymptotic analysis is the same.

e. We can sort by starting with an empty tableau and inserting all the n^2 elements in it. Each insertion is $O(n+n) = O(n)$. The complexity is $n^2 O(n) = O(n^3)$. Afterwards we can take them one by one and put them back in the original array which has the same complexity. In total, its $O(n^3)$.

We can also do it in place if we allow for "partial" tableaus where only a portion of the top rows (and a portion of the last of them) is in the tableau. Then we can build the tableau in place and then start putting each minimal element to the end. This would be asymptotically equal, but use constant memory. It would also sort the array in reverse.

f. We start from the lower-left corner. We check the current element current with the one we're looking for key and move up if $\text{current} > \text{key}$ and right if $\text{current} < \text{key}$. We declare success if $\text{current} = \text{key}$ and otherwise terminate if we walk off the tableau.