Chapter 6. Heapsort

6.1-1.
2^h, $2^{h+1}-1$ respectively.

6.1-2.
By the previous exercise, $2^h \le n \le 2^{h+1}-1 \Rightarrow h = \lfloor \lg n \rfloor$.

6.1-3.
By the definition, it's trivial.

6.1-4.
Anywhere in the leaf nodes, because it cannot have a child node.

6.1-5.
Yes, because for all i we have $A[\text{PARENT}(i)] \le A[i]$. The converse is not true.

6.1-6.
No. The parent of 7 is 6, violating the max-heap property.

6.1-7.
SInce $2*(\lfloor n/2 \rfloor + 1) \ge n+1$, the nodes indexed by $\lfloor n/2 \rfloor + 1, \cdots, n$ are leaves.
For node indexed by $i \le \lfloor n/2 \rfloor$, it cannot be a leaf because its left child $2*i \le n$ exists in the node.

6.2-1.
Swap $3-10$ and $3-9$.

6.2-2.

Algorithm 1: MIN-HEAPIFY(A, i)

```
$l = \text{LEFT}(i)$;
$r = \text{RIGHT}(i)$;
if $l \leq A.\text{heap} - \text{size}$ and $A[l] < A[i]$ then
    smallest = l;
else
    smallest = i;
end
if $r \leq A.\text{heap} - \text{size}$ and $A[r] < A[i]$ then
    smallest = r;
end
if $smallest \neq i$ then
    exchange $A[i]$ with $A[smallest]$;
    $\text{MIN} - \text{HEAPIFY}(A, smallest)$;
end
```

The running time of MIN-HEAPIFY is exactly same with MAX-HEAPIFY.

6.2-3.
There is no effect because it does not perform swap operation with child node.

6.2-4.
There is no effect because it is a leaf node.

6.2-5.

Algorithm 2: MIN-HEAPIFY-ITERATIVE(A)

```
$i = 1$;
while $i \leq A.\text{heap} - \text{size}/2$ do
    $l = \text{LEFT}(i)$;
    $r = \text{RIGHT}(i)$;
    if $l \leq A.\text{heap} - \text{size}$ and $A[l] < A[i]$ then
        smallest = l;
    else
        smallest = i;
    end
    if $r \leq A.\text{heap} - \text{size}$ and $A[r] < A[i]$ then
        smallest = r;
    end
    if $smallest \neq i$ then
        exchange $A[i]$ with $A[smallest]$;
        $i = smallest$;
    else
        return;
    end
end
```

6.2-6.
MAX-HEAPIFY calls down every node on a simple path from the root down to a leaf, which has length $\lfloor \lg n \rfloor = \Omega(\lg n)$.

6.3-1.
Swap $10 - 22, 17 - 19, 3 - 84, 5 - 84, 5 - 22, 5 - 10$.

6.3-2.
Because doing so does not satisfty the underlying assumption of MAX-HEAPIFY that all subtrees of the root is a max-heap.

6.3-3.
The maximum number of nodes can exist at height h is 2^{H-h}, where $H = \lfloor \lg n \rfloor$ is the height of the heap. Since $2^{H-h} \leq \lceil n/2^{h+1} \rceil$, the argument holds.

6.4-1.
Swap $2 - 20, 13 - 25, 5 - 25, 5 - 13, 5 - 8, 4 - 25, 4 - 20, 4 - 17, 5 - 20, 5 -$

$17, 2-17, 2-13, 2-8, 4-13, 4-8, 4-7, 4-8, 4-7, 2-7, 2-5, 2-5, 2-4.$
6.4-2.
Initialization : For $i = 0$, it's trivial.
Maintenance : When we swap $A[1]$ with $A[i]$, since $A[1]$ is the largest element in $A[1, \cdots, i]$ and smaller than any element in $A[i+1, \cdots, n]$, $A[i]$ becomes the $n-i+1$-th largest element of $A[1, \cdots, n]$, combining with the previous hypothesis maintains the loop invariant.
Termination : $A[2, \cdots, n]$ is sorted and $A[1]$ is the smallest element in the array, which means the array is sorted.

6.4-3.
Both of them is still $\Theta(n \lg n)$.

6.4-4.
BUILD-MAX-HEAP(A) is $\Omega(n)$, but MAX-HEAPIFY is n time called and each has time complexity $\Omega(\lg k)$. Hence the lower bound for total time complexity is

$$\sum_{k=1}^{n-1} \Omega(\lg k) = \Omega(\lg(n/2-1)!\,) = \Omega((n/2)\lg(n/2)) = \Omega(n \lg n).$$

6.4-5.
First assume that $n = 2^k - 1$, that is, the heap is a complete binary tree. Removing the first 2^{k-1} keys, we can observe that the keys initially form a heap-ordered subtree of the total heap. In this subtree, the nodes at the bottom level are swapped wih the root during the process, and the nodes not at the bottom level are displaced by smaller keys from the higher position, sifting down by one step a time. Therefore, the number of data movements required is at least the sum of path length of the subtree composed of the largest 2^{k-1} keys except those at the bottom level. We observe that among the 2^{k-1} large keys, only at most 2^{k-2} of them can be at the bottom level. Otherwise, there will be at least $2^{i-2}+1$ large keys at level i and summing up these number gives $2^{k-1} + k - 2$, which is a contradiction since we have only 2^{k-1} large keys. Therefore, we have at least 2^{k-2} large keys in the levels other than the bottom level. Since the first $k-3$ levels contain exactly $2^{k-3} - 1$ elements, we have at least $2^{k-3} + 1$ large keys located in either level $k-1$ or $k-2$. These elements require at least $k-2$ sifts to get to the root, providing the lower bound $(k-2)2^{k-3} = \Omega(n \lg n)$.

6.5-1.
Extract 15, move 1 to the top, swap $1-13, 1-12, 1-6$.

6.5-2.
Insert 10, swap $8-10, 9-10$.

6.5-3

Algorithm 3: HEAP-MINIMUM(A)

```
return A[1];
```

Algorithm 4: HEAP-EXTRACT-MIN(A)

```
if $A.heap\text{-}size < 1$ then
  | error : heap − underflow;
end
$min = A[1]$;
$A[1] = A$[A.heap-size];
A.heap-size = A.heap-size - 1;
MIN-HEAPIFY(A, 1);
return min;
```

Algorithm 5: HEAP-DECREASE-KEY(A, i, key)

```
if $key > A[i]$ then
  | error : invalid − key;
end
A[i] = key;
while $i > 1$ AND $A[PARENT(i)] > A[i]$ do
  | exchange A[i] with A[PARENT(i)];
  | i = PARENT(i);
end
```

Algorithm 6: MIN-HEAP-INSERT(A, key)

```
A.heap-size = A.heap-size + 1;
A[A.heap-size] = $\infty$;
HEAP-DECREASE-KEY(A, A.heap-size, key);
```

6.5-4.
To pass the check $key < A[i]$.

6.5-5.
Initialization : At the time HEAP-INCREASE-KEY is called the array is the heap, and the check $key < A[i]$ is bypassed, there can be at most only one violation for heap property: A[i].
Maintenance : Whenever we exchange A[i] with its parent, the max-heap property can have only one violation for heap property: A[PARENT(i)].
Termination : At the loop termination, the max-heap property is not violated or the heap is consumed, concluding that the whole array becomes a max-heap.

6.5-6.

Algorithm 7: HEAP-INCREASE-KEY(A, i, key)

```
if key < A[i] then
    error : invalid − key;
end
while i > 1 AND A[PARENT(i)] < key do
    A[i] = A[PARENT(i)];
    i = PARENT(i);
end
A[i] = key;
```

6.5-7.
Constructing the priority queue in decreasing priority gives a first-in, first-out queue.
Constructing the priority queue in increasing priority gives a stack.

6.5-8.

Algorithm 8: HEAP-DELETE(A, i)

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

6.5-9.

Take on element of each list and put them together in a min-heap. Take the minimum element from the heap. Insert another element from the list the element taken came from unless the list is empty. Continue until the heap is empty. Since this does n insertion to the heap which takes $\lg k$ time, the total running time is $O(n \lg k)$.

6-1.
a. No. For $A = [1, 2, 3]$, BUILD-MAX-HEAP gives $[3, 2, 1]$ and BUILD-MAX-HEAP' gives $[3, 1, 2]$.
b. The worst case occurs when the elements are given in increasing order. The time complexity is $\sum_{k=1}^{n} \lg k = \lg(n!) = \Theta(n \lg n)$.

6-2.
a. We can represent a d-ary heap with the two following functions: d-ARY-PARENT(i) = $\lfloor \frac{i-2}{d} + 1 \rfloor$, d-ARY-CHILD(i, j) = $(i-1)d + j + 1$.
b. $\Theta(\log_d n)$.
c. Same with HEAP-EXTRACT-MAX. The running time is $O(d \log_d n)$.
d. Same with MAX-HEAP-INSERT. The running time is $O(d \log_d n)$.
e. Same with HEAP-INCREASE-KEY. The running time is $O(d \log_d n)$.

6-3.
a.

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

b. If $Y[1, 1] = \infty$, all element is ∞, making Y empty.
If $Y[m, n] < \infty$, all element is finite, making Y full,
c. Extract $A[1, 1]$, which is a smallest element. Replace it with ∞. Having the procedure similar to MAX-HEAPIFY, restore it into the Young tableau by continuously swapping the top-left element with its right or down neighbor recursively. The running time is $O(m + n)$.
d. The procedure is similar to c., but we start from the bottom-right corner instead.
e. We can just push elements into an empty tableau and put them back in the original array. The complexity is $O(2n) \cdot n^2 = O(n^3)$.
f. Start from the top-right corner. Each step, compare the key with the element in the tableau. If the key is equal to the element, we're done. If the

key is smaller, move left. If the key is bigger, move down. If we walk out from the tableau then the search has failed.