1. (CLRS 6.1-1) What are the minimum and maximum number of elements in a heap of height $h$?

**Solution:** The minimum number of elements is $2^h$ and the maximum number of elements is $2^{h+1} - 1$.

2. (CLRS 6.1-4) Where in a max-heap might the smallest element reside, assuming that all elements are distinct?

**Solution:** Since the parent is greater or equal to its children, the smallest element must be a leaf node.

3. (CLRS 6.2-4) What is the effect of calling MAX-HEAPIFY($A, i$) for $i > \text{size}[A]/2$?

**Solution:** Nothing, the elements are all leaves.

4. (CLRS 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.

**Solution:**

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

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

HEAP-DECREASE-KEY(A,i,key)
    if key > A[i]
        then error ```'new key is larger than current key'"
    A[i] <- key
    while i > 1 and A[parent(i)] > A[i]
        do exchange A[i] <-> A[parent(i)]
        i <- parent(i)

MIN-HEAP-INSERT(A,key)
    heap-size[A] <- heap-size[A] + 1
    A[heap-size[A]] <- +inf
    HEAP-DECREASE-KEY(A,heap-size[A],key)
```
5. (CLRS 6-2) Analysis of d-ary heaps
A d-ary heap is like a binary heap, but instead of 2 children, nodes have d 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**. Analyze its running time in terms of d and n.
d. Give an efficient implementation of **Insert**. Analyze its running time in terms of d and n.
e. Give an efficient implementation of **Heap-Increase-Key** (A, i, k), which sets \( A[i] \leftarrow \max(A[i], k) \) and updates the heap structure appropriately. Analyze its running time in terms of d and n.

**Solution:**


\[
\text{Children}(i) = \{di - d + 2, di - d + 3, ..., di, di + 1\}.
\]

The parent of \( A[1] \) is \( A[1] \). The parent of \( A[i] \) for \( 2 \leq i \leq d + 1 \) is \( A[1] \). The parent of \( A[i] \) for \( d + 2 \leq i \leq 2d + 1 \) is \( A[2] \). The general rule is:

\[
\text{Parent}(i) = \left\lceil \frac{i - 1}{d} \right\rceil.
\]

You can check for instance using the rule above that \( \text{Parent}(di - d + 2) \) is i and \( \text{Parent}(di - d + 1) \) is \( i - 1 \).

b. The number of nodes at level \( h \) is at most \( d^h \). The total number of nodes in a tree of height \( h \) is at most \( 1 + d + \ldots + d^h = \Theta(d^h) \). Setting \( d^h = n \) implies the height is \( \Theta(\log_d n) \).

c. **Extract-Max** is the same as for binary heaps. Its running time is given by the running time of **Heapify**. The **Heapify** operation on d-ary heaps works very similarly to the one on binary heaps:

\[
\text{Heapify}_d(A, i)
\]

i. find largest element \( l = \max\{A[i], \text{Children}(A[i])\} \)

ii. if \( l \neq i \) then exchange \( A[i] \leftarrow A[l] \) and \( \text{Heapify}_d(A, i) \)

The running time of **Heapify\_d** is \( \Theta(d \cdot \log_d n) \). The \( d \) term is because at each iteration a node compares its value and the values of its \( d \) children to find the maximum, which takes \( O(d) \) time.

d. **Insert** is the same as for binary heaps. The running time is \( \Theta(\text{height}) = \Theta(\log_d n) \).
e. The running time is \( O(\log_d n) \) if \( A[i] < k \).
\begin{algorithm}
\textbf{HEAP\_INCREASE\_KEY\_D}(A, i, k)
\begin{algorithmic}
\State i. if $A[i] < k$ then
\State \hspace{1em} $A[i] = k$
\State \hspace{1em} while $i > 1$ and $A[\text{Parent}(i)] < A[i]$ do
\State \hspace{2em} exchange $A[i] \leftrightarrow A[\text{Parent}(i)]$
\State \hspace{2em} $i = \text{Parent}(i)$
\end{algorithmic}
\end{algorithm}