(6.5-5) Initialization: Just before the while loop begins, the invariant is true because we are able to assume that $A$ is a max-heap at the time Heap-Increase-Key is called.

Maintenance: Assume the invariant holds at the beginning of an iteration. Then we know $A$ is a max-heap except, possibly, $A[\text{Parent}(i)] < A[i]$, in which case these two values will be swapped, resulting in $A$ being a max-heap except, possibly, $A[\text{Parent}(i)] > A[\text{Parent}(\text{Parent}(i))]$. The last step in the loop body is to set $i = \text{Parent}(i)$, so the invariant holds at the start of the next iteration.

Termination: The loop terminates when either $i = 1$ or $A[\text{Parent}(i)] \geq A[i]$. If $i = 1$, we have from the invariant that $A$ is a max-heap, except possibly $A[1]$ may be larger than $A[\text{Parent}(1)]$, but $A[1]$ is the root, so $A$ is a max-heap. If $A[\text{Parent}(i)] \geq A[i]$, then from the invariant we can conclude that $A$ is a max-heap.

(6.5-9) Let $A_1, A_2, \ldots, A_k$ be $k$ sorted arrays. One way to solve this problem is to use a size $k$ min-heap $A$. Each node in the min-heap will be a value from one of the $k$ arrays along with an indicator of which array it came from (an index between 1 and $k$). Initially, fill $A$ with the first element of each of the sorted arrays using $k$ calls to Min-Heap-Insert. Next, extract the minimum node with a call to Heap-Extract-Min; note which array the minimum value came from and insert the next value from that array into the heap using Min-Heap-Insert. Continue in this fashion, extracting the minimum value, noting which array it came from, and inserting an element from that array into the heap (if there are no elements remaining in the array, do not insert anything). Each of the $n$ array elements will be inserted into the heap and extracted from the heap exactly one time. Since Min-Heap-Insert on a heap of size $k$ is $O(\log k)$ and, similarly, Min-Heap-Insert is $O(\log k)$, the entire procedure is $O(n \log k)$.

(6-2) (a) The representation of a $d$-ary array is a natural extension of a binary array. Store the heap in an array $A$. The root of the heap is stored in $A[1]$, and its $d$ children are in $A[2], \ldots, A[d + 1]$ in that order; their children are in $A[d + 2], \ldots, A[d^2 + d + 1]$, etc.

- The parent of node $i$ has index $\lfloor (i - 2)/d \rfloor$. 

• The $j$th child of node $i$ has index $d(i - 1) + j + 1$.

Note that the parent of the $j$th child of node $i$ is

$$
\begin{align*}
\lfloor (d(i - 1) + j + 1 - 2)/d + 1 \rfloor \\
= \lfloor (d(i - 1) + j - 1)/d + 1 \rfloor \\
= \lfloor (i - 1) + 1 \rfloor \\
= i
\end{align*}
$$

where the second equality follows because $0 \leq j - 1 < d$.

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

(e) The `Heap-Max-Extract` function given for binary heaps works for $d$-ary heaps with one change: instead of comparing the value of node $i$ to its two children, we have to compare it to all $d$ children. This change must be made in `Max-Heapify`. The running time for the modified `Max-Heapify` is proportional to the height of the heap times the number of children that must be examined, or $\Theta(d \log_d n) = \Theta(d \lg n/\lg d)$.

(d) The $d$-ary implementation of `Max-Heap-Insert` will call a modified `Heap-Increase-Key`, which has worst-case running time proportional to the height of the tree. For a $d$-ary heap, we have already seen that the height is $\Theta(\log_d n) = \Theta(\lg n/\lg d)$.

(e) `Heap-Increase-Key` only needs to be modified to call the $d$-ary `Parent` function. In the worst case, the height of the tree must be traversed to maintain the min-heap property, so again the worst case running time is $\Theta(h) = \Theta(\log_d n) = \Theta(\lg n/\lg d)$. 

2