6-2. Analysis of d-ary heaps

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 MAX-HEAPIFY in a d-ary max-heap. Analyze its running time in terms of d and n.

d. Given an efficient implementation of BUILD-MAX-HEAP in a d-ary max-heap. Analyze its running time in terms of d and n.

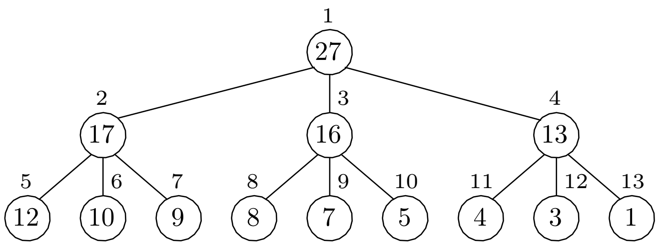
e. Give an efficient implementation of EXTRACT-MAX in a d-ary max-heap. Analyze its running time in terms of d and n.

f. Give an efficient implementation of 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.

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

Answer.

a. The d-ary heap can be represented by an array that corresponds to a nearly complete d-ary tree (see Section B.5.3). Figure 1 shows a triple max-heap viewed as a triple tree and an array. The



(a)

1	2	3	4	5	6	7	8	9	10	11	12	13
27	17	16	13	12	10	9	8	7	5	4	3	1

(b)

Figure 1. A max-heap viewed as **(a)** a triple tree and **(b)** an array.

root of the tree is $A[1]$, and given the index i of a node, we can compute the indices of its parent:

PARENT(i)
return $\lceil (i-1)/d \rceil$

Email address: informlarry@gmail.com

All children of node i are successors of the last child of node $i-1$, which has its index $d(i-1)+1$. So the jth child of node i is $d\,(i-1)+1+j$:

```
CHILD(i, j)
    return d (i − 1) + 1 + j
```

b. Since each non-leaf node contains at most d children, a tree of hieght h has total number of nodes bounded by:

$$1+d+\ldots+d^{h-1} < n \leq 1+d+\ldots+d^h$$

that is,

$$\frac{d^h-1}{d-1} < n \leq \frac{d^{h+1}-1}{d-1}$$

In other words, $n=\Theta\,(d^h)$. Thus, $h=\Theta\,(\log_d n)$.

c. The D-MAX-HEAPIFY procedure takes as its arguments an array A, an index i into the array and the amount of its children d. When it is called, D-MAX-HEAPIFY assumes that the d-ary rooted at CHILD(i, j), for each $1 \leq j \leq d$ are max-heaps, but that $A[i]$ might be smaller than its children. D-MAX-HEAPIFY lets the value at $A[i]$ "float down" in the max-heap in the same way as binary heap.

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

The running time of D-MAX-HEAPIFY on a subtree of size n rooted at a given node i is the $\Theta\,(d)$ time to fix up the relation ships amongs the elemnts $A[i], A[\text{CHILD}(i,1)], \ldots, A[\text{CHILD}(i,d)]$, plus the time to run D-MAX-HEAPIFY on a subtree rooted at one of the children of node i (assuming that the recursive call occurs). The children's subtrees each have depth at most $(\log_d n)-1$. Hence, the total running time for this algorithm is $O\,(d\log_d n)$.

d. We can use the procedure D-MAX-HEAPIFY in a bottom-up manner to build a d-ary max-heap from array $A[1\,..\,n]$, where $n=A.length$, into a max-heap. The last non-leaf node is the parent of the last node, and by procedure D-PARENT, it has it index $\lceil (n-1)/d \rceil$. So elements in the subarray $A[(\lceil (n-1)/d \rceil + 1)\,..\,n]$ are all leaves of the tree, an so each is a 1-element heap to begin with. The procedure D-BUILD-MAX-HEAP goes through the remaining nodes of the tree and runs D-MAX-HEAPIFY on each one.

```
D-BUILD-MAX-HEAP(A)
    A.heap-size = A.length
    for i = ⌈(A.length−1)/d⌉ downto 1
        D-MAX-HEAPIFY(A, i, d)
```

We can compute the running time of this procedure by observing that there are

$$d^{\log_d [n(d-1)]-(h+1)} = d^{\log_d \frac{n(d-1)}{d^{h+1}}} = \frac{n\,(d-1)}{d^{h+1}}$$

nodes on level of height h. As part (c) shows, each call to D-MAX-HEAPIFY on level of height h is $O\,(d\,h)$. So, the running time

$$\begin{aligned}
T\,(n) &\le \sum_{h=0}^{\log_d n(d-1)} d\,h\,\frac{n\,(d-1)}{d^{h+1}} \\
&< n\,(d-1)\sum_{h=0}^{\infty}\frac{h}{d^h} \\
&\le n\,(d-1)\,\frac{1/d}{(1-1/d)^2} && \textit{A.8} \\
&= n\,\frac{d}{d-1} \\
&= n\left(1+\frac{1}{d-1}\right)
\end{aligned}$$

For any $d \ge 2$, we have $1 \le 1+1/(d-1) \le 2$, so $T\,(n) \le \Theta\,(n)$.

e. The procedure D-HEAP-EXTRACT-MAX implements the EXTRACT-MAX operation in a d-ary max-heap. It is almost identical to HEAP-EXTRACT-MAX for binary heap which we saw in the text.

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

The running time of D-HEAP-EXTRACT-MAX is $O\,(d \log_d n)$, since it performs only a constant amount of work on top of the $O\,(d \log_d n)$ time for D-MAX-HEAPIFY.

f. The procedure D-HEAP-INCREASE-KEY implements the INCREASE-KEY operation in a d-ary max-heap. As increasing the key of $A[i]$ might violate the max-heap property, the procedure traverse a path from this node to the root and repeatedly move the element upward until it is smaller than its parent.

```
D-HEAP-INCREASE-KEY(A, i, k)
1  if k < A[i]
2      error "new key is smaller than current key"
3  A[i] = k
4  while i > 1 and A[PARENT(i)] < A[i]
5      exchange A[i] with A[PARENT(i)]
6      i = PARENT(i)
```

The running time of D-MAX-HEAP-INCREASE-KEY on an n-element heap is $O\,(\log_d n)$, since the path traced from the node updated in line 3 to the root has length $O\,(\log_d n)$.

g. The procedure D-MAX-HEAP-INSERT is the same to that of binary heap in the text.

```
D-MAX-HEAP-INSERT(A, k)
A.heap-size = A.heap-size + 1
A[A.heap-size] = -∞
D-HEAP-INCREASE-KEY(A, A.heap-size, k)
```

The running time of D-MAX-HEAP-INSERT on an n-element heap is $O\,(\log_d n)$.