Heaps [Cormen, Leiserson, Rivest]

Given an array (list) A with indices $1, \ldots, n$ we can think of it as a (nearly complete) *binary tree* with the following functions.

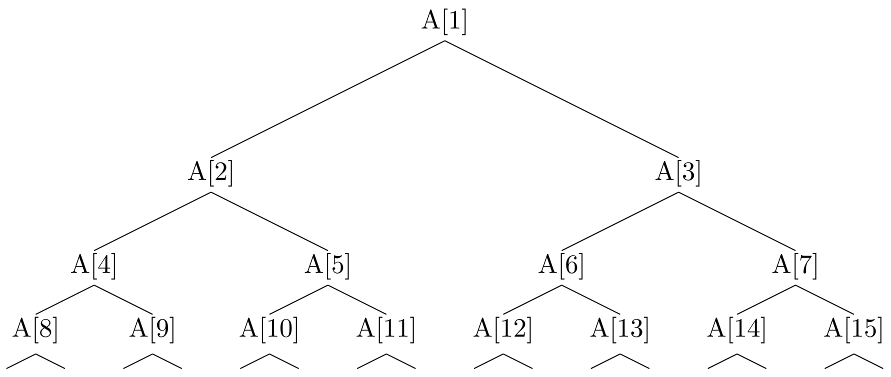
```
PARENT($k$)
    return $\lfloor k/2 \rfloor$

LEFT($k$)
    return $2k$

RIGHT($k$)
    return $2k + 1$
```

These three functions are extremely fast when implemented in binary.

We visualize the tree as follows:



Definition (Heap)**.** An array (list) A with indices $1, \ldots, n$ is a **heap** if

$$A[\,\textsc{parent}(k)\,] \geq A[\,k\,] \quad \text{for } k > 1. \tag{1}$$

That is, each parent is at least as large as its children.

Questions

(1) Show that a subtree of a heap is also a heap.
(2) Where is the largest element of a heap?
(3) Where might the smallest element of a heap reside?
(4) Is an array that is in reverse order a heap?
(5) Is the list $[23, 17, 14, 6, 13, 10, 1, 5, 7, 12]$ a heap?

Maintaining the heap property

HEAPIFY is an important subroutine for manipulating heaps. Its inputs are an array A and an index i into the array. When HEAPIFY is called, it is assumed that the binary trees rooted at LEFT(k) and RIGHT(k) are heaps, but that $A[k]$ may be smaller than its children, thus violating the heap property (1).

The idea behind HEAPIFY is that if the value at index k is larger than the values at either of its children (LEFT(k) and RIGHT(k)) then nothing needs to be done. Otherwise, the value at k is swapped with the value of the larger child, and then HEAPIFY is called again, this time with index equal to this larger child.

```
HEAPIFY($A, k$)

 1  $L \leftarrow$ LEFT($k$)
 2  $R \leftarrow$ RIGHT($k$)
 3  if $L \leq$ heap-size($A$) and $A[L] > A[k]$
 4      then largest $\leftarrow L$
 5      else largest $\leftarrow k$
 6  if $R \leq$ heap-size($A$) and $A[R] > A[$largest$]$
 7      then largest $\leftarrow R$
 8  if largest $\neq k$
 9      then exchange $A[k] \leftrightarrow A[$largest$]$
10           HEAPIFY($A$, largest)
```

Example. Run HEAPIFY($A, 2$) on $A = [16, 4, 10, 14, 7, 9, 3, 2, 8, 1]$. Make sure the assumptions for HEAPIFY are met.

Running Time. The running time of HEAPIFY(A, k) is $O(\text{height}(k))$.

Buliding a heap

We can use the procedure HEAPIFY in a bottom-up manner to convert an array $A[1, \ldots, n]$, where $n = \text{length}(A)$, into a heap. Since the elements in the subarray $A[\lfloor n/2 \rfloor + 1, \ldots, n]$ are all leaves of the tree, each is a 1-element heap to begin with. The procedure BUILD-HEAP goes through the remaining nodes of the tree and runs HEAPIFY on each one. The order in which the nodes are processed gurantees that the subtrees rooted at children of a node i are heaps before HEAPIFY is run at that node.

```
BUILD-HEAP($A$)

 1  heap-size($A$) $\leftarrow$ length($A$)
 2  for $k \leftarrow \lfloor \text{length}(A)/2 \rfloor$ downto 1
 3      do HEAPIFY($A, k$)
```

Example. Run BUILD-HEAP(A) on $A = [4, 1, 3, 2, 16, 9, 10, 14, 8, 7]$.

Priority queues

A **priority queue** is a data structure for maintaining a set S of elements, each with an assoiciated value called a **key**. A priority queue supports the following operations.

- INSERT(S, x) inserts the element x into the set S.
- MAXIMUM(S) returns the element of S with the largest key.
- EXTRACT-MAX(S) removes and returns the element of S with the largest key.

A priority queue can be used in scheduling where a set S of jobs are to be performed in the order of most important to least important (as indicated by their keys).

Following are the heap based versions of the above three operations. In each it is assumed that the input A is a heap.

HEAP-INSERT(A, x)

```
1   heap-size($A$) $\leftarrow$ heap-size($A$) $+ 1$
2   $k \leftarrow$ heap-size($A$)
3   while $k > 1$ and $A[\text{PARENT}(k)] < x$
4       do $A[k] \leftarrow A[\text{PARENT}(k)]$
5           $k \leftarrow \text{PARENT}(k)$
6   $A[k] \leftarrow x$
```

Example. Run HEAP-INSERT$(A, 15)$ on $A = [16, 14, 10, 8, 7, 9, 3, 2, 4, 1]$. This array has already had BUILD-HEAP run on it.

Running Time. The running time of HEAP-INSERT(A, x) is $O(\log n)$ where $n =$ length(A). This follows since the height of the heap is $\log n$ and all we do is move *up* from a leaf to (possibly) the root of the tree.

Remark. Why don't we use HEAP-INSERT to build heaps by inserting elements from a list one at a time into an initially empty heap? The running time would be

$$\begin{aligned}\sum_{k=1}^{n} \log k &\approx \int_1^n \log x \, dx \qquad \text{(vague but actually OK)} \\ &= x \log x - x \Big|_1^n \\ &= n \log n - n + 1 \\ &= O(n \log n)\end{aligned}$$

Much slower than the $O(n)$ of BUILD-HEAP as we will see.

```
HEAP-MAX($A$)
1   return $A[1]$
```

```
HEAP-EXTRACT-MAX($A$)
1   if heap-size($A$) $< 1$
2       then error "heap underflow"
3   max $\leftarrow A[1]$
4   $A[1] \leftarrow A[$heap-size($A$)$]$
5   heap-size($A$) $\leftarrow$ heap-size($A$) $- 1$
6   HEAPIFY($A, 1$)
7   return max
```

Example. Run HEAP-EXTRACT-MAX(A) on $A = [16, 15, 10, 8, 14, 9, 3, 2, 4, 1, 7]$. This is the heap that resulted from the last example.

Running Time. The running time of HEAP-EXTRACT-MAX(A) is $O(\log n)$ where $n = \text{length}(A)$. This follows since there is only a constant amount of work done in addition to the single call to HEAPIFY which runs in $O(\log n)$ time.

...AND...BENEATH THE PROFESSOR'S WINDOW... THE MIXTURE CONTINUED TO PULSATE AND QUIVER WHERE IT HAD LAIN...PULSATED...QUIVERED...AND GREW!
GREW! STOOD UP! ERECT! A HORRIBLE STANDING GLOB OF SWAMP THING! THERE WAS NOTHING TO CALL IT BUT... HEAP!
4

– Will Elder, from MAD #5 (1953)