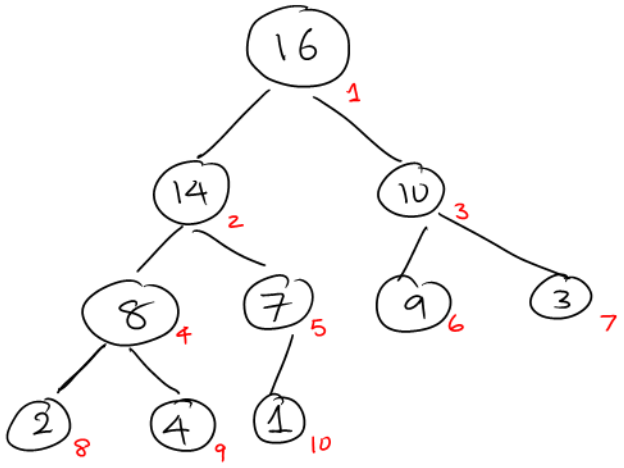
Heaps: An array $A[1..n]$ is called a heap if $\forall i \geq 1$: $A[i] \geq A[2i]$ (if $2i \leq n$) and $A[i] \geq A[2i+1]$ (if $2i+1 \leq n$).

Example $A =$

16	14	10	8	7	9	3	2	4	1
1	2	3	4	5	6	7	8	9	10

(INDEX)

is a heap and it can be visualized as a Binary Tree



root of tree: $A[1]$

Node with index i:

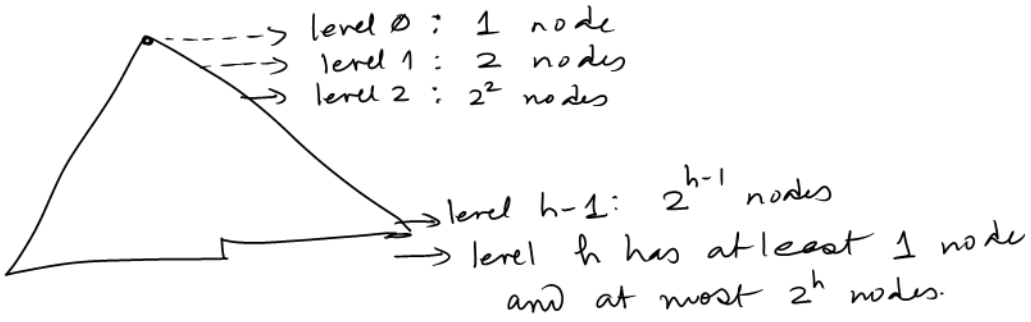
- parent of i has index $\lfloor i/2 \rfloor$.
  $parent(i) := \lfloor i/2 \rfloor$
- left child of i is at index $2i$ (if $2i \leq n$)
  $Left(i) = 2i$
- right child of i is at index $2i+1$ (if $2i+1 \leq n$).
  $Right(i) = 2i+1$.

Note that $A[1..n]$ is a heap if for all $1 < i \leq n$
$A[parent(i)] \geq A[i]$.

Q: What is the height of the heap?
≡ What is the height of the tree?
≡ # of edges on the longest root-to-leaf path

Define $h :=$ height of the Binary Tree T.

$T =$



$$\Rightarrow \quad 1+2+2^2+\cdots+2^{h-1}+1 \le n \le 1+2+2^2+\cdots+2^{h-1}+2^h$$

$$\text{or} \quad (2^h-1)+1 \le n \le 2^{h+1}-1$$

$$\text{or} \quad 2^h \le n < 2^{h+1}$$

$$\Rightarrow \quad h = \lfloor \log_2 n \rfloor .$$

Hence the height of heap h is $O(\log n)$.

Operations on a Heap

Assume we have a heap $A[1..N]$

an integer n, $1 \le n \le N$

Sequence of n numbers stored in $A[1..n]$ satisfying the heap property: $\forall i: 1 < i \le n$

$$A[\text{parent}(i)] \ge A[i].$$

#1: MAXIMUM(A): Return (A[1])

return the largest element.

Time $= O(1)$.

#2: Increase_key (A, i, x): assumes that $x \ge A[i]$, $1 \le i \le n$ and $A[1..n]$ is a heap.

increases A[i] to x and restores heap property

```
A[i] := x;
while i > 1 and A[parent(i)] < A[i] do
    [SWAP (A[parent(i)] ⇔ A[i]);
     i := parent(i);]
```

TIME $=$ $O($height$) = O(\log n)$.

③ Insert (A, x) : Assume that $A[1..n]$ is a heap and $n < N$.

Inserts element x and restore the heap property.

```
n := n+1;
A[n] := -∞;
    (note that A[1..n] is a heap)
Increase-key (A, n, x)
```

$TIME \equiv O(1) +$ time for increase key

$= O(1) + O(\log n)$

$= O(\log n)$

4. First Method for constructing a heap on n-elements.

Input: An array $A[1..N]$ on elements

Output: Permute elements of A so that $A[1..N]$ is a heap.

```
n := 1;  (note that A[1..n] is a heap)
while n < N do
    [insert (A, A[n+1]);
```

Insert increments n. Also $A[1..n]$ is a heap.

Running Time: Cost for insert is sum total of cost for inserting the 1st element + cost of inserting 2nd element + cost of inserting 3rd element + ... + cost of inserting i^{th} element + ... + cost of inserting N^{th} element.

$\sim \log 1 + \log 2 + \log 3 + \cdots + \log N$

$\equiv \Theta(N \log N)$. [See Assignment 1].

5. Heapify (A, i)

Assumptions: Binary tree rooted at Left(i) and Right(i) are max-heaps, but $A[i]$ may be smaller than its children.

Max-heap property may be violated at the node i.

Objective: Restore the heap property.

The heap property can be restored as follows:

1. Compute MAX(A[i], A[left(i)], A[right(i)]);
   (If A[i] is largest, then heap property is satisfied)
2. If A[i] is not the MAX, then
   SWAP(A[i] $\leftrightarrow$ MAX(A[left(i)], A[right(i)])
3. Note that heap property is satisfied at A[i], but one of its children may not satisfy the property.
4. Recurse on the children, which does not satisfy the heap property.

TIME $\equiv \bigcirc$ (Height of the node i)

#6: An O(N) algorithm for BUILDING Heap.

Let A[1..N] be the array which needs to be permuted, so that A satisfies the heap property.

Note that $A[\frac{N}{2}+1, \frac{N}{2}+2, \ldots, N]$ satisfy heap property (since they correspond to leaves).

BUILD-MAX-HEAP(A)
For i := $\lfloor \frac{N}{2} \rfloor$ downto 1 do MAX-Heapify(A, i).

Cost of BUILDING THE Heap

$$= \sum_{h=0}^{\lfloor \log N \rfloor} \left(\frac{N}{2^{h+1}} \right) O(h)$$

$O(h)$ → Time for heapify for a node at height h

$\frac{N}{2^{h+1}}$ → # of nodes at height h.

$$= \bigcirc(N)$$

Note $$\sum_{h=0}^{\log n} \frac{h}{2^h} \leq \sum_{h=0}^{\infty} \frac{h}{2^h} = \sum_{h=0}^{\infty} h \left(\frac{1}{2}\right)^h = \frac{1/2}{(1-1/2)^2} = 2$$

Note for $|x| < 1$

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$$

$$\Leftrightarrow \sum_{k=0}^{\infty} k \cdot x^{k-1} = \frac{1}{(1-x)^2}$$

$$\Leftrightarrow x \sum_{k=0}^{\infty} k \cdot x^{k-1} = \frac{x}{(1-x)^2}$$

$$\Leftrightarrow \sum_{k=0}^{\infty} k x^k = \frac{x}{(1-x)^2}$$

Set $x = 1/2$, $k = h$.

#7 HeapSort :

In place sorting algorithm, doesn't require extra memory.

Input: An unsorted array A.
Output: Array A in sorted order.

HeapSort(A)

1. BUILD-MAX Heap (A). Heapsize := n;
2. For i := n downto 2 do
   [Exchange A[1] ↔ A[i];
   Heapsize := Heapsize − 1;
   Heapify (A, 1); ----> O(logN)/per call.

Correctness is obvious since root always contain the maximum value.

Complexity: Step 1 takes O(N) time.
Step 2 requires at most logN time per call, or in all O(NlogN) time.

#8 Extract-Max (A): Delete the largest element and restore the heap property.

Step 1: Return (A[1]) [max := A[1]]
Step 2: A[1] := A[Heap-size(A)]
Step 3: Heapsize(A) := Heapsize (A) − 1;
Step 4: Heapify (A, 1)

Heap-size denotes the number of elements in the heap.

TIME = O(logn)

Note that A[1.. Heapsize (A)] is a valid Heap.
New Heap has one fewer element than the old heap.

—x—