Exercise: In Step 1 on page 37, we used $n/5$ groups of length 5.

- Use groups of length 7.

  This gives the recurrence

  $$T(n) = n + T\left(\frac{n}{7}\right) + T\left(\frac{5}{7}n\right),$$

  which solves to $T(n) = O(n)$.

- Use groups of length 3.

  This gives the recurrence

  $$T(n) = n + T(n/3) + T\left(\frac{2}{3}n\right),$$

  which solves to $T(n) = \Theta(n \log n)$

Heaps

Assume we have a data structure that

- stores numbers
- supports the operations
  - insert (x) ; insert the number x
  - extract_maximum ; return and delete the largest element

Then we can sort any sequence $a_1, a_2, \ldots, a_n$ of numbers:

```
for $i = 1, 2, \ldots, n$ : insert ($a_i$)
for $i = 1, 2, \ldots, n$ : extract_maximum
```

Data structure that does this : <u>heap</u>

Array $A[1..n]$ is called a <u>heap</u> if for all $i \geq 1$,

$$A[i] \geq A[2i] \qquad (\text{if } 2i \leq n)$$

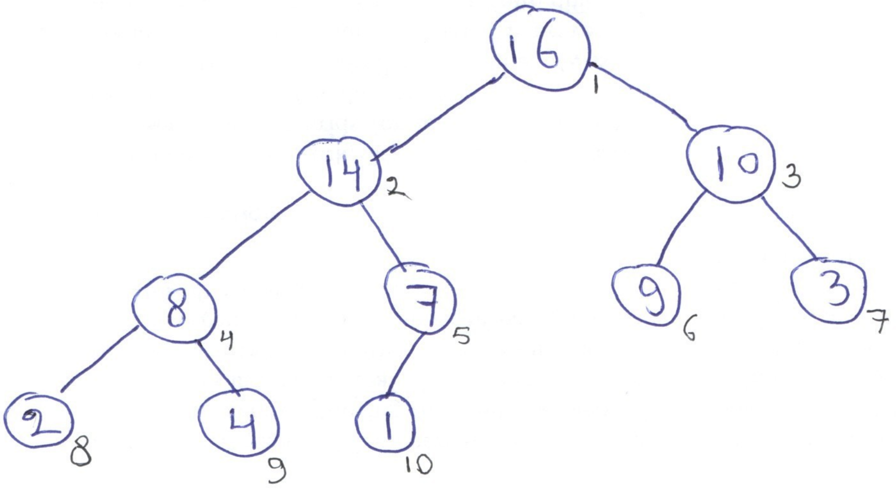
and

$$A[i] \geq A[2i+1] \qquad (\text{if } 2i+1 \leq n)$$

<u>Example</u>:

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

<u>Visualize</u> this array as a binary tree:



root of the tree : $A[1]$

node with index i :

- parent of i has index $\lfloor i/2 \rfloor$
- left child of i has index $2i$
- right child of i has index $2i+1$

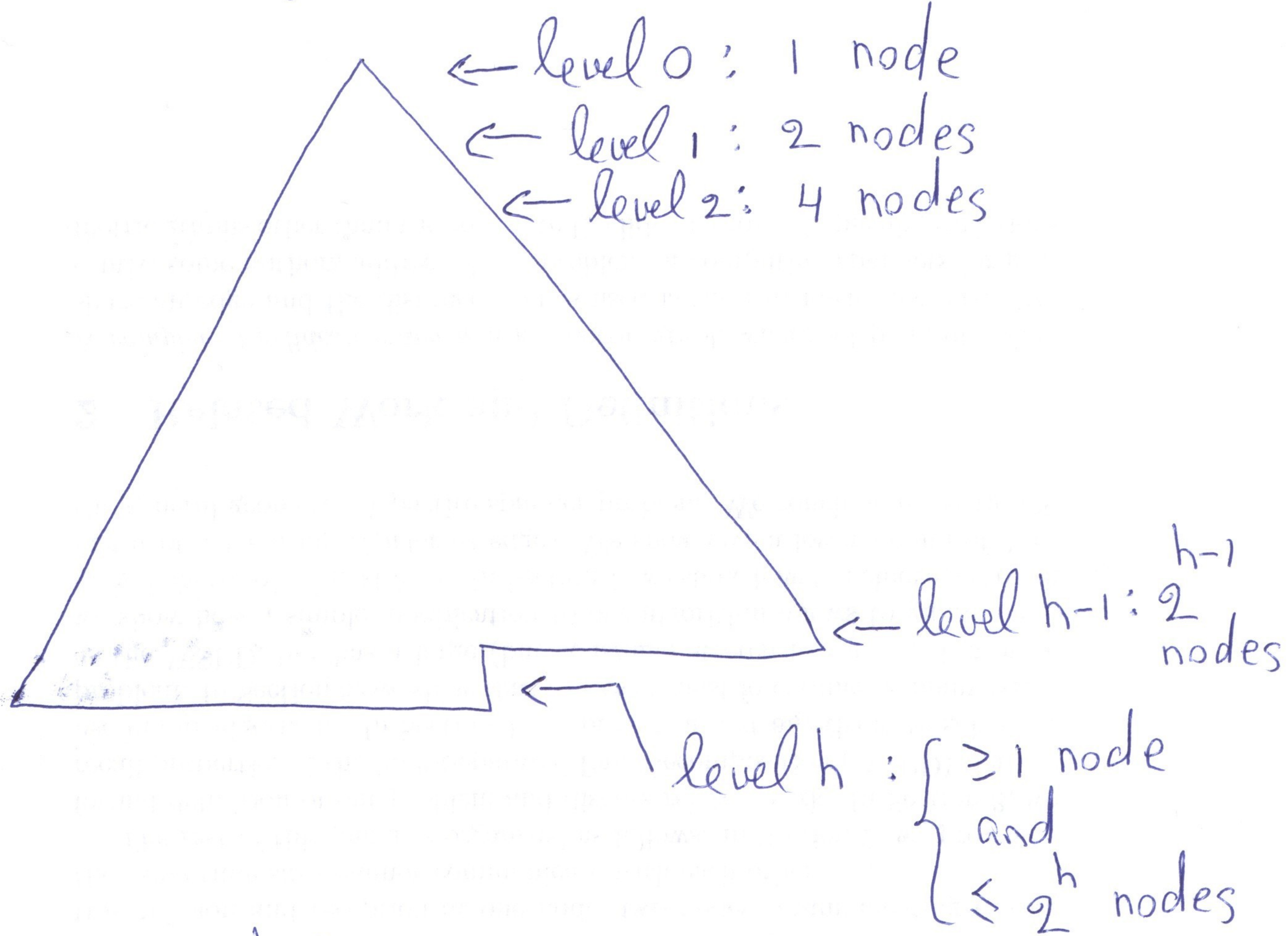
$A[1..n]$ is a heap if for all i with $1 < i \leq n$:

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

What is the $\underbrace{\text{height}}_{}$ of a heap $A[1..n]$?

= height of the tree in the visualization

Define h = height.



$$1+2+2^2+\dots+2^{h-1}+1 \le n \le 1+2+2^2+\dots 2^h$$

$$(2^h-1)+1 \le n \le 2^{h+1}-1 < 2^{h+1}$$

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

$$h \le \log n < h+1$$

$$\boxed{h = \lfloor \log n \rfloor}$$

Outline:

① maximum (A): return largest element.
$O(1)$ time

② increase_key (A, i, x):
assumes that $x \geq A[i]$.
this operation increases $A[i]$ to x and restores the heap property.
$O(\log n)$ time

③ insert (A, x): insert element x and restore the heap property; $O(\log n)$ time

④ Algorithm for building a heap: $O(n \log n)$ time

⑤ heapify: $O(\log n)$ time

⑥ Algorithm for building a heap: $O(n)$ time

⑦ heap-sort: $O(n \log n)$ time

⑧ extract_max (A): return and delete the largest element, restore the heap property.
$O(\log n)$

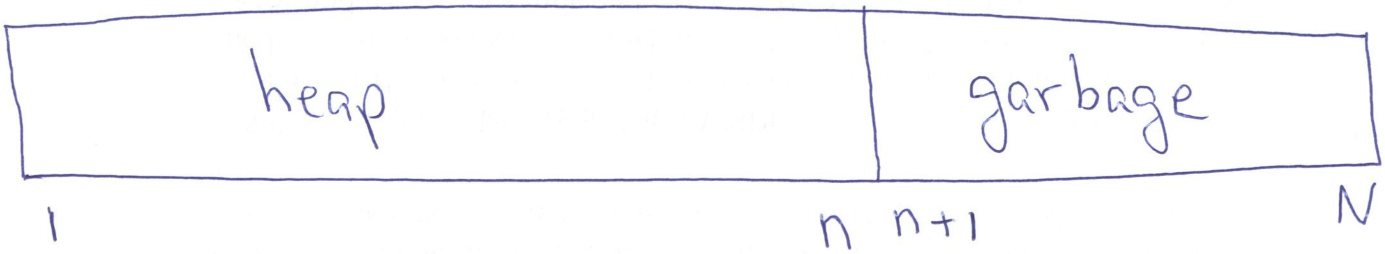
From now on;

array $A[1..N]$,

integer n, $1 \leq n \leq N$

Sequence of n numbers, stored in $A[1..n]$

$A[1..n]$ is a heap



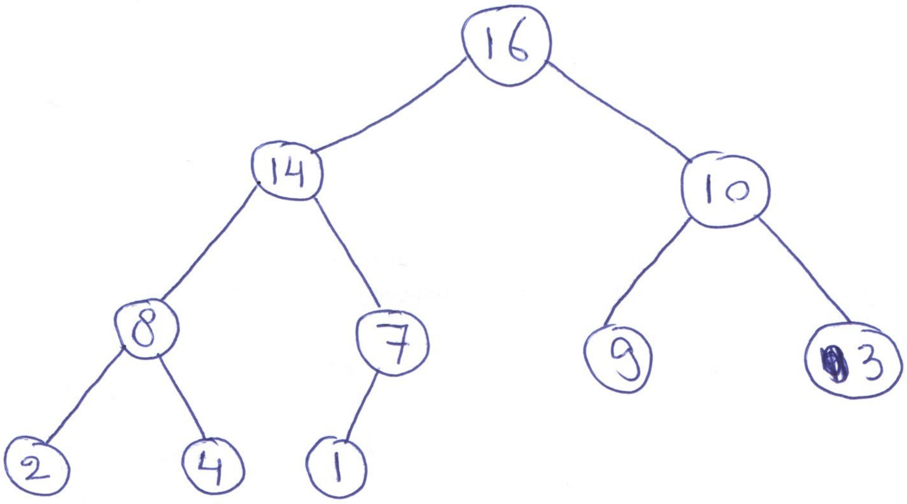
---

① maximum(A): return $A[1]$

time: $O(1)$

② increase_key(A, i, x):

this operation assumes that $x \geq A[i]$



→ increase this key to 15:
increase_key(A, 9, 15).
explain algorithm for this example.

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

---

time = $O(\text{height}) = O(\log n)$

③ insert(A, x):

this operation assumes that $A[1..n]$ is a heap and $1 \leq n < N$; thus, there is space for the new element x.

explain algorithm insert(A, 13) for example on page 48.

```
n = n+1;
A[n] = -∞;
// A[1..n] is a heap
increase_key(A, n, x)
```

$$\begin{aligned} \text{time} &= O(1) + \text{time for increase\_key} \\ &= O(1) + O(\log n) \\ &= O(\log n) \end{aligned}$$

④ How to build a heap: given array $A[1..N]$, permute the elements such that $A[1..N]$ is a heap.

```
$n = 1$;      // $A[1..n]$ is a heap
while $n < N$ : insert $(A, A[n+1])$
  // observe that insert increments $n$
```

$$\begin{aligned} \text{time} &\sim \log 1 + \log 2 + \log 3 + \dots + \log N \\ &\leq \log N + \log N + \log N + \dots + \log N \\ &= N \log N \\ &= O(N \log N) \end{aligned}$$

By the way:

$$\sum_{n=1}^{N} \log n \geq \sum_{n=N/2}^{N} \log n \geq \sum_{n=N/2}^{N} \log N/2$$

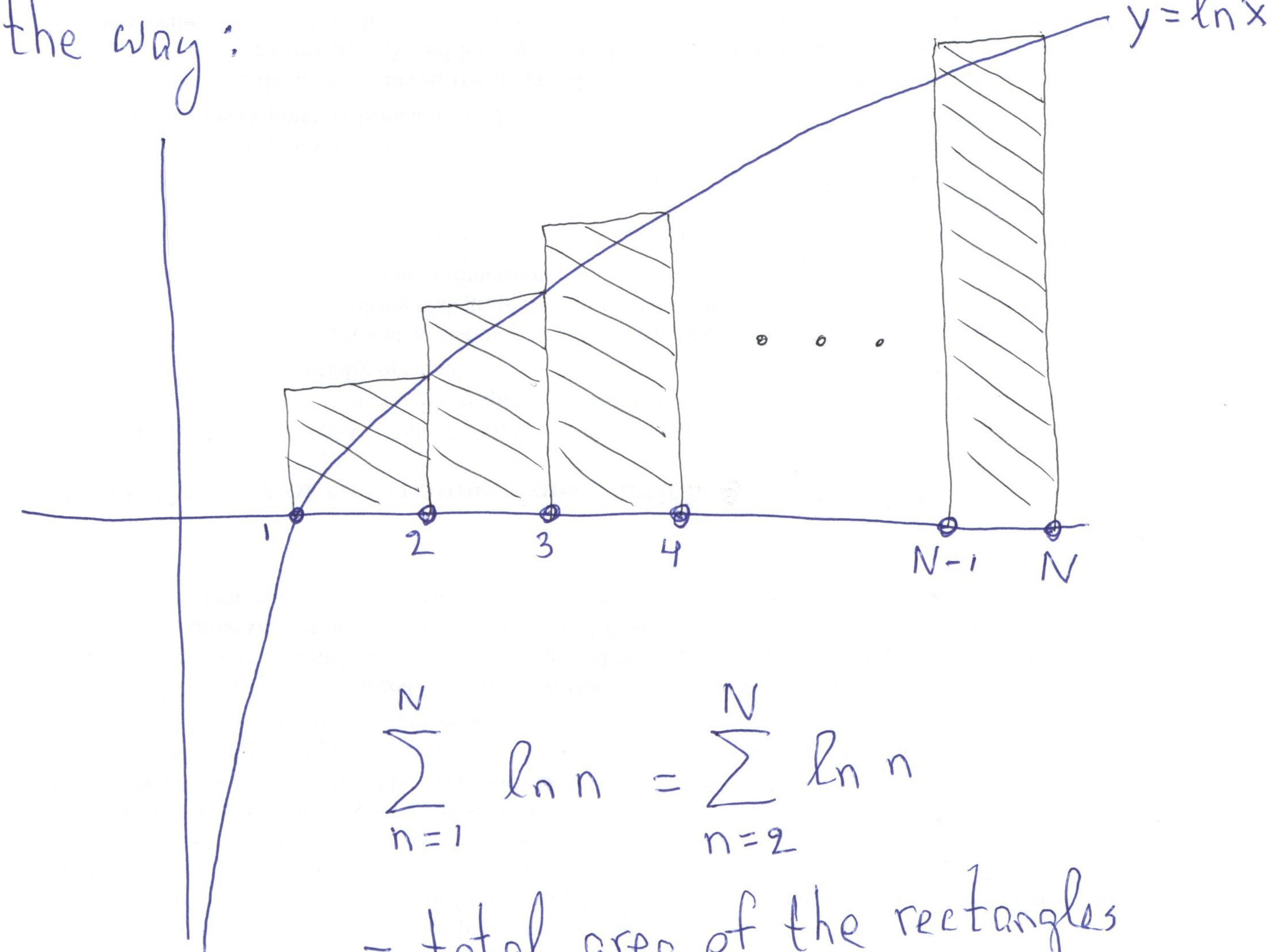
$$= \frac{N}{2} \log \frac{N}{2} = \frac{1}{2} N \log N - \frac{1}{2} N$$

$$\geq \frac{1}{4} N \log N \quad (\text{if } N \geq 4)$$

$$= \Omega(N \log N)$$

∴ this algorithm for building a heap takes $\Theta(N \log N)$ time.

By the way:



$$\sum_{n=1}^{N} \ln n = \sum_{n=2}^{N} \ln n$$

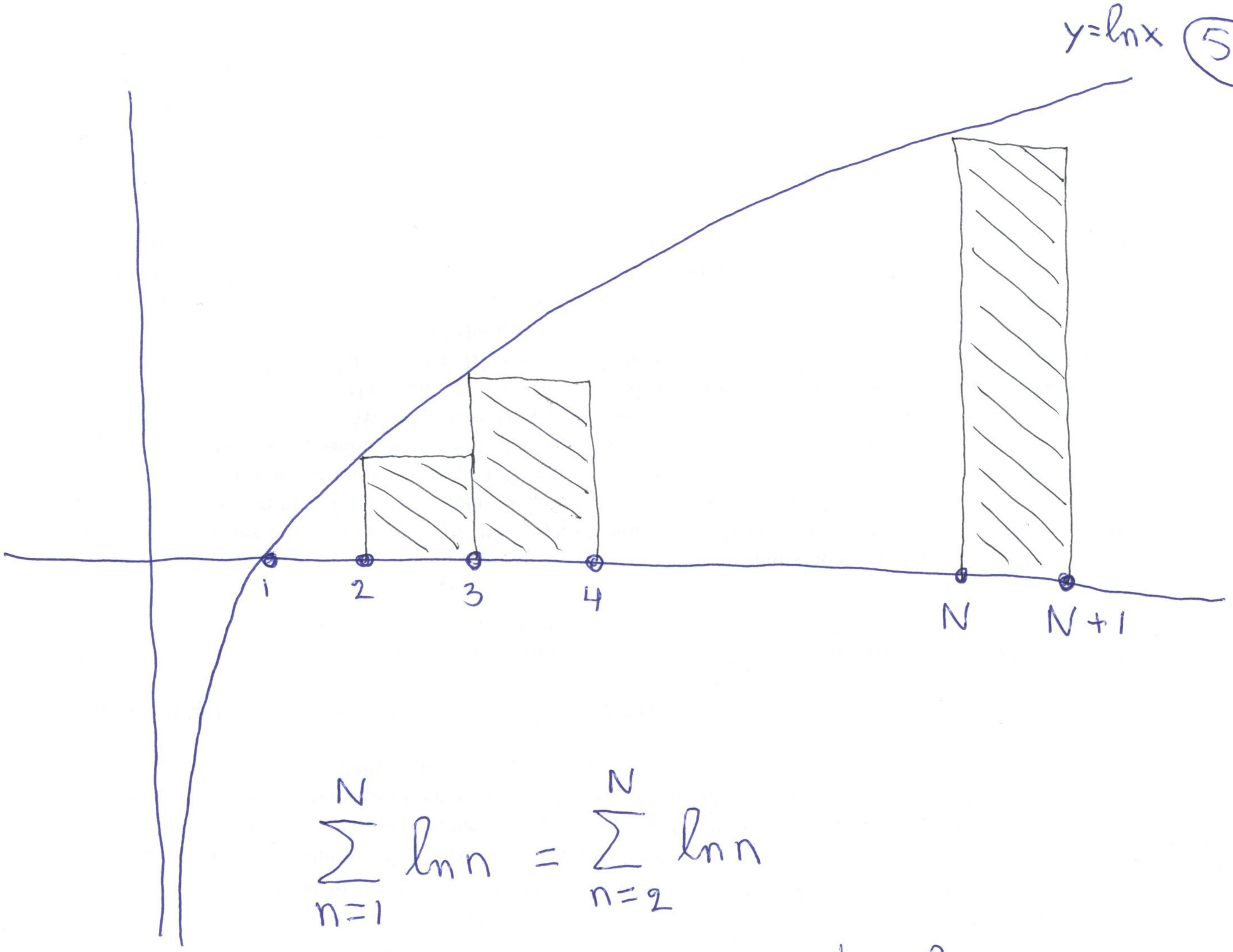
= total area of the rectangles

$$\geq \int_1^N \ln x \, dx = x \ln x - x \Big]_{x=1}^{x=N}$$

$= N \ln N - N + 1$

$= \Omega(N \ln N)$

$= \Omega(N \log N)$



$$\sum_{n=1}^{N} \ln n = \sum_{n=2}^{N} \ln n$$

= total area of the rectangles

$$\leq \int_{2}^{N+1} \ln x \, dx = x \ln x - x \Big]_{x=2}^{x=N+1}$$

$$= (N+1)\ln(N+1) - (N+1) - 2\ln 2 + 2$$

$$= O(N \ln N)$$

$$= O(N \log N)$$

⑤ heapify(A, i):

this operation assumes:

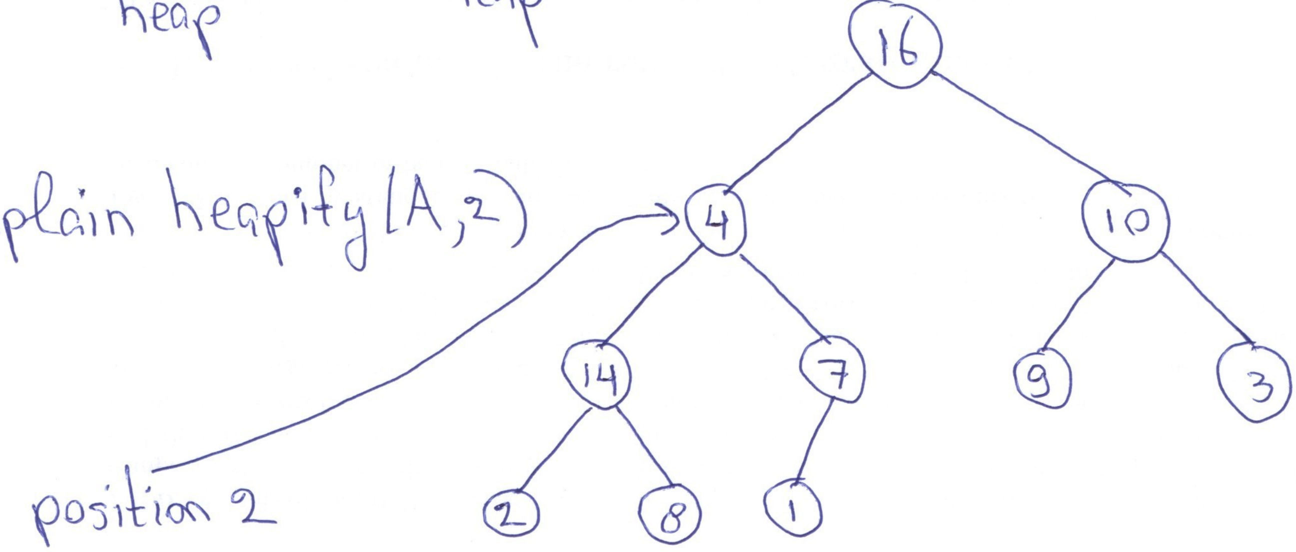
$1 \leq i \leq n$,

subtree rooted at left(i) is a heap,

subtree rooted at right(i) is a heap.

at termination: subtree rooted at i is a heap.

explain heapify(A,2)



```
$\ell$ = left (i);
r = right (i);
if $\ell \le n$ and $A[\ell] > A[i]$
then max = $\ell$
else max = i
endif;
if $r \le n$ and $A[r] > A[max]$
then max = r
endif;
if $max \ne i$
then swap A[i] and A[max];
        heapify (A, max)
endif
```

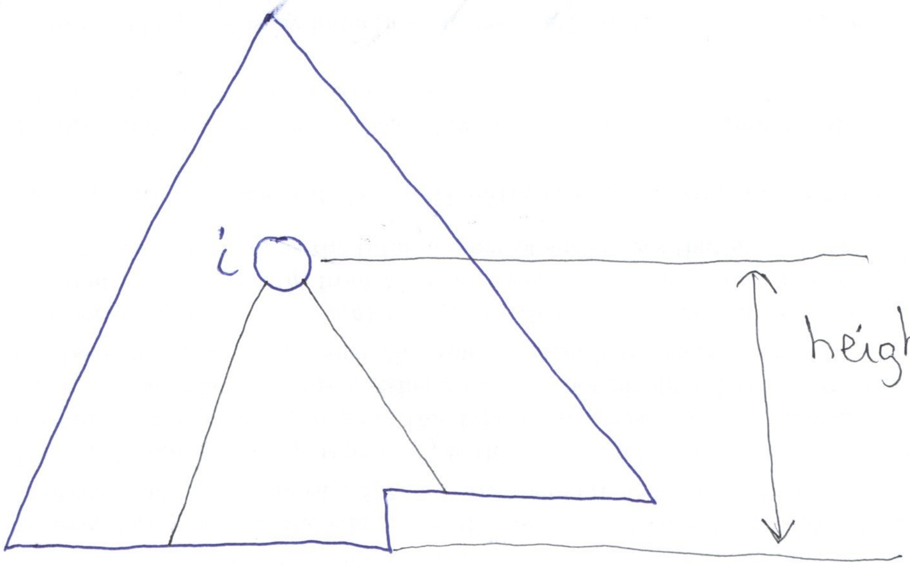
find the largest of A[i], A[ℓ], and A[r]; store its index in max.

---

time = O(1) + time for recursive call at a node one level lower in the tree

$\therefore$ time = O(height) = O(log n).

More precisely: time = O(height of node i)



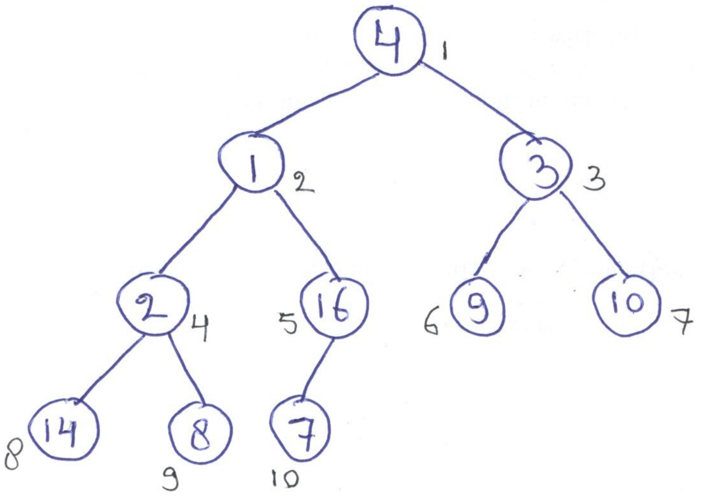
⑥ build_heap(A):

input: array $A[1..N]$.

output: heap $A[1..N]$ containing the same elements.

```
n = N;
for i = ⌊N/2⌋ downto 1 : heapify(A, i)
```

$A = [4, 1, 3, 2, 16, 9, 10, 14, 8, 7]$, $N = 10$



i starts at $\lfloor \frac{N}{10} \rfloor = 5$.

why: because $A[6], \ldots, A[10]$ are leaves and the subtree rooted at a leaf is a heap.

In general: all $A[i]$, $\lfloor N/2 \rfloor + 1 \leq i \leq N$, are leaves.

running time: $\lfloor N/2 \rfloor$ calls to heapify, each call takes $O(\log N)$ time.

$\therefore$ total time $= O(N \log N)$.

Note: this is an upper bound.

Is there a better upper bound: yes.

Recall: heapify(A, i) takes time O(height of node i)

To simplify: assume the bottom level of the tree is full: $N = 1 + 2 + 2^2 + \dots + 2^h = 2^{h+1} - 1$, where h = height of the tree.

height	number of nodes
h	1
$h-1$	2
$h-2$	2^2
$h-3$	2^3
$\vdots$	$\vdots$
1	2^{h-1}
0	2^h

Running time of build_heap:

$$1 \cdot h + 2(h-1) + 2^2(h-2) + 2^3(h-3) + \dots +$$

$$2^{h-2} \cdot 2 + 2^{h-1} \cdot 1$$

$$= \sum_{i=1}^{h} 2^{h-i} \cdot i$$

$$= 2^h \sum_{i=1}^{h} i \left(\tfrac{1}{2}\right)^i$$

// $h = \lfloor \log N \rfloor \leq \log N$

// $\therefore 2^h \leq 2^{\log N} = N$

$$\leq N \cdot \underbrace{\sum_{i=1}^{\infty} i \cdot \left(\tfrac{1}{2}\right)^i}_{\text{constant}}$$

$$= O(N).$$

By the way: for $-1 < x < 1$:

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

differentiate: $$\sum_{i=1}^{\infty} i \cdot x^{i-1} = \frac{1}{(1-x)^2}$$

multiply by x: $$\sum_{i=1}^{\infty} i \cdot x^i = \frac{x}{(1-x)^2}$$

$x = 1/2$: $$\sum_{i=1}^{\infty} i \cdot \left(\frac{1}{2}\right)^i = \frac{1/2}{(1-1/2)^2} = 2.$$

⑦ heap-sort:

input: array $A[1..N]$ of numbers.

output: array $A[1..N]$ containing the same numbers in sorted order.

```
build_heap(A);
n = N;
while n ≥ 2
do  // A[1..n] is a heap, A[n+1..N] contains the
    // N-n largest elements in sorted order

    Swap A[1] and A[n];
    n = n-1;
    heapify(A,1)
endwhile
```

Running time:

build_heap: $O(N)$

while-loop: $O(\log N + \log(N-1) + \ldots + \log 3 + \log 2)$

$= O(N \log N)$

Total running time $= O(N \log N)$.

⑧ extract_max(A):

this operation assumes that A[1..n] is a heap and $n \geq 1$.

```
max = A[1];
A[1] = A[n];
n = n-1;
heapify(A,1);
return max
```

---

time = O(1) + time for heapify(A,1)

= O(1) + O(log n)

= O(log n)

We have discussed max-heaps.

Symmetric: min-heap

$$A[\text{parent}(i)] \le A[i], \quad 1 < i \le n$$

① → minimum(A)

② → decrease_key(A, i, x) ; assumes $x \le A[i]$

⑧ → extract_min(A)