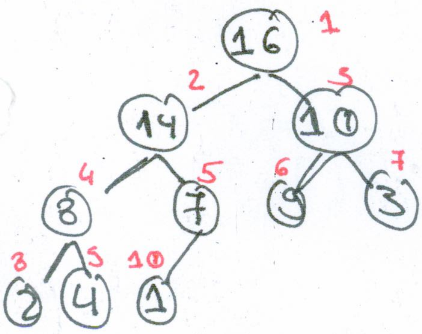
6. Heapsort [cormen2001]

- Running time $O(n \log n)$
- Heapsort sorts in place: only a constant number of array elements are stored outside the input array at any time.
- Uses the heap data structure to manage information during the execution of the algorithm.

6.1 Heaps

- Q: What is a heap?

  Array object that can be viewed as a nearly complete binary tree



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

- Each node of the tree corresponds to an element of the array that stores the value in the node
- The tree is completely filled on all levels except possibly the lowest (leaf nodes)

- An array $\underline{A}$ that represents a heap is an object with two attributes:

  Length [A] - number of elements in the array

  heap-size [A] - number of elements in the heap stored within array A. It will be used later for the Heap Sort algorithm.

- That is, although A [1 ... length [A]] may contain valid number, no element past A [heap-size [A]], where heap-size [A] $\leq$ length [A] is an element of the heap

- Root of the tree is A [1]

- Given the index $\underline{i}$ of a node then we can compute the following indexes:

  Parent(i) { return $\lfloor i/2 \rfloor$ }

  Left(i) { return $2i$ }

  Right(i) { return $2i+1$ }

Notes:

"Fast" implementations:

$2i = i << 1$

$2i+1 = (i << 1) + 1$

$\lfloor i/2 \rfloor = i >> 1$

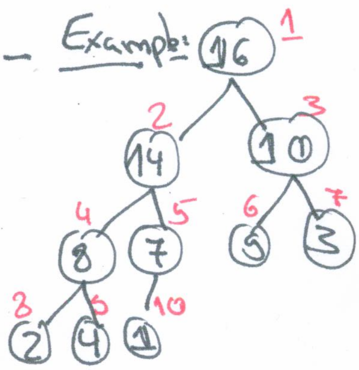
- There are two kinds of binary heaps:
  - max-heap: $A[\text{parent}(i)] \geq A[i] \Rightarrow$ largest element in the root
  - min-heap: $A[\text{parent}(i)] \leq A[i] \Rightarrow$ smallest element in the root

- Heapsort algorithm uses max-heaps

- Define the height of a node in a heap to be the number of edges on the longest simple downward path from the node to a leaf.

- Define the height of the heap to be the height of the root.

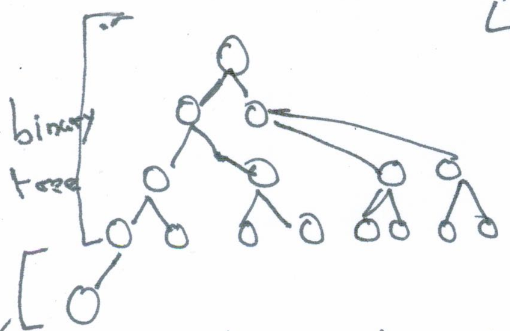
- Example:



height[1] = 3
height[2] = 2
height[3] = 1
height[4] = 1
height[5] = 1
height[6] = 0
height[7] = 0
height[8] = 0
height[9] = 0
height[10] = 0

- $\boxed{\text{Q: What is the height of a heap?}}$

Remember: binary tree that is almost complete



Level/Depth	Number of nodes
0	$1 = 2^0$
1	$2 = 2^1$
2	$4 = 2^2$
3	$8 = 2^3$
$\vdots$	
h	2^h

$\Rightarrow$ If we have n element in total, then:

$2^h = n \Leftrightarrow$

$\Leftrightarrow \boxed{h = \log_2 n}$

$\hookrightarrow$ We must have at least a leaf in the last level

If n is not a power of 2 then $h = \lfloor \log_2 n \rfloor$

- The rest of the notes focus on describing:

  Max Heapify - procedure running in $O(\log n)$ time that is key to maintain the max heap property

  Build Max Heap - procedure running in linear time that produces a heap from an unordered input array.

  Heapsort - procedure running in $O(n \log n)$ time that sorts an array

6.2 Maintaining the heap property

- Max Heapify routine has as inputs the array A and an index i
- When Max Heapify is called it is assumed that the binary trees rooted at Left(i) and Right(i) are max heaps but that $A[i]$ may be smaller than its children, thus violating the max heap property
- Max Heapify places $A[i]$ in the correct position so that the subtree rooted at index i becomes a max heap

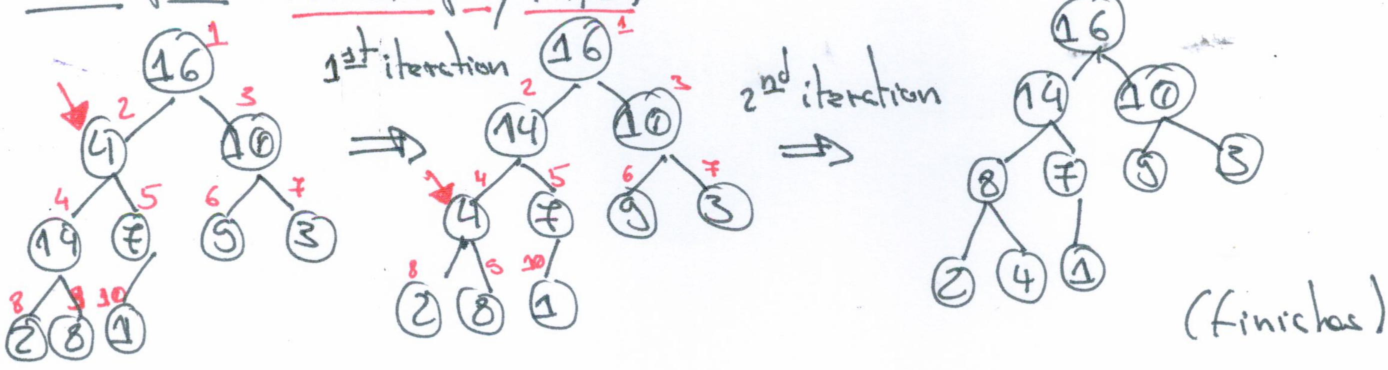
↪

```
MaxHeapify (A, i) {
   l = Left(i)
   r = Right(i)
   if l ≤ heap-size[A] and A[l] > A[i]
       then largest = l
       else largest = i
   if r ≤ heap-size[A] and A[r] > A[largest]
       then largest = r
   if largest ≠ i
       then { exchange A[i] <-> A[largest]
              MaxHeapify [A, largest]
            }
}
```

Determine the largest element (also check if we are within the bounds of the array)

The subtree "rooted" at the index largest may be violating the max heap property thus we need to recursively call MaxHeapify

Example: MaxHeapify (A, 2)



Question:

What is the running time for MaxHeapify?

- $\Theta(1)$ time to fix-up the relationships among the elements $A[i]$, $A[Left[i]]$, $A[Right[i]]$
- But then we call MaxHeapify recursively...
- We recursively call MaxHeapify on one of the children of node $\underline{i}$
- In the worst-case scenario we have to call MaxHeapify all the way from the root to one of the leaf nodes
- This implies $\underline{O(\log n)}$ time

Alternatively
- For a node of height $\underline{h}$ then $O(h)$ time.

6.3 – Building a heap –

Question:

So, how can we build a heap?

→ <u>Idea:</u> Why not use the procedure MaxHeapify in a bottom-up manner to convert an array $A[1 \dots n]$ where $n = length[A]$ into a max-heap

Build Max Heap (A) {

  heapSize [A] = length [A]

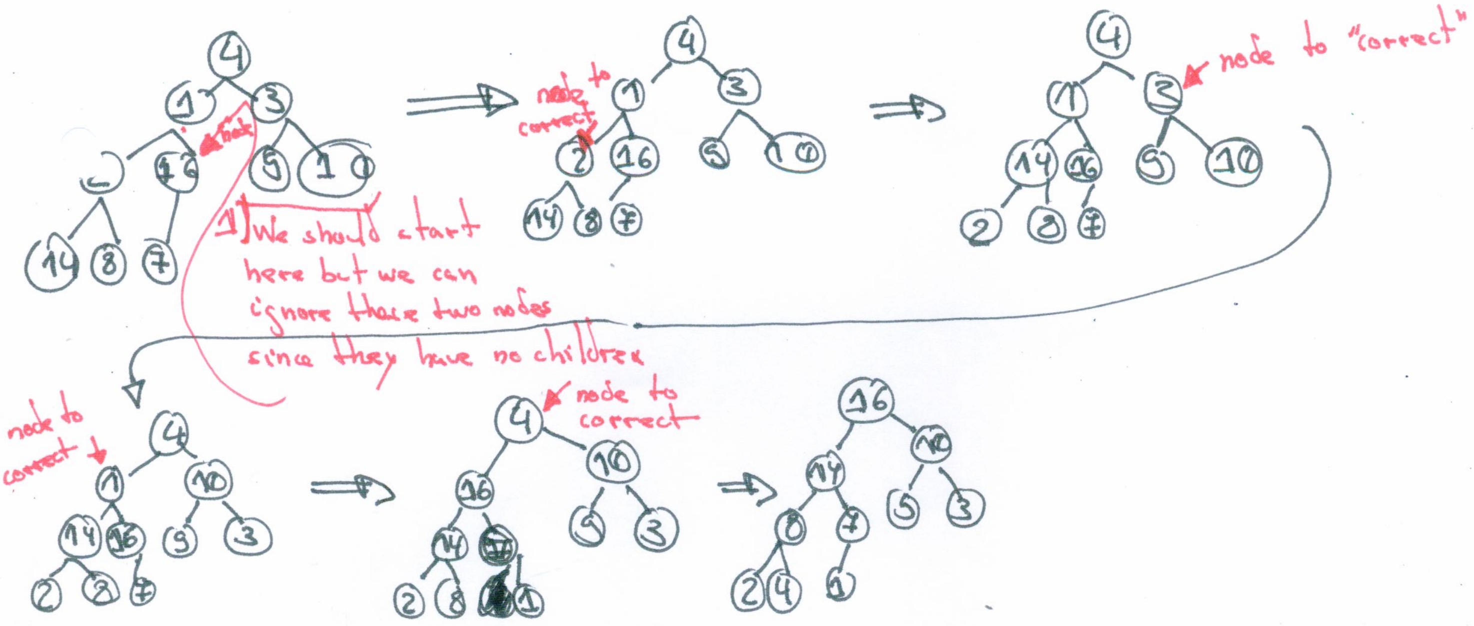
  for $i = \lfloor length[A]/2 \rfloor$ to 1

    Max Heapify (A, i)

the elements $A[\lfloor n/2 \rfloor + 1 \dots n]$ are all leaves of the tree. Therefore we do not need to perform Max Heapify on those

- Example:

A =	4	1	3	2	16	5	10	14	8	7

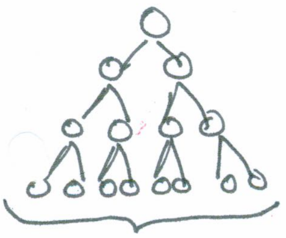


Question:
What is the execution time for Build Max Heap?

- For every single node we need to call Max Heapify
- There are $n/2$ nodes to evaluate, since the elements $A[\lfloor n/2 \rfloor + 1 \dots n]$ are leafs
- Each node call Max Heapify, therefore $O(n \log n)$
- Can we do better than $O(n \log n)$? I.e. a tighter bound?

Idea:

- Observe that the time for MaxHeapify to run at a node varies with the height of the node
- The heights of most nodes are small. This happens because we are dealing with a quasi-binary tree and thus most of the nodes will be on the bottom levels...
- $\boxed{\text{Question: How many nodes exist at height } h \text{ ?}}$



≈ 16 nodes

Level / Height	nodes
0 / 3	$1 = 2^0$
1 / 2	$2 = 2^1$
2 / 1	$4 = 2^2$
3 / 0	$8 = 2^3$

This is not leading anywhere, what if try to use n?

Height	Nodes
3	$1 \approx \frac{15}{16} = \frac{n}{2^{3+1}}$
2	$2 \approx \frac{15}{8} = \frac{n}{2^{2+1}}$
1	$4 \approx \frac{15}{4} = \frac{n}{2^{1+1}}$
0	$8 \approx \frac{15}{2} = \frac{n}{2^{0+1}}$

There are at most $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ ∴ nodes of height h

- Generalizing this for when the bottom level is not full requires a proof by induction which is beyond the scope of this class...
- Lets try again to determine the complexity
  - Time required for MaxHeapify on a node of height h = $O(h)$
  - Total cost of BuildMaxHeap: $\sum_{h=0}^{\boxed{\lceil \lg n \rceil}} \left\lceil \frac{n}{2^{h+1}} \right\rceil \cdot O(h)$

Here should be floor instead of ceiling

~~maximum~~ height

Each one of those has $O(h)$ cost

$$= O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right)$$

Notes: Geometric Series: $\sum_{k=0}^{\infty} k u^k = \frac{u}{(1-u)^2}, \quad |u|<1$

- $$\sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \le \sum_{h=0}^{\infty} \frac{h}{2^h} = \sum_{h=0}^{\infty} h \cdot \frac{1}{2^h} \overset{u=\frac{1}{2}}{=} \frac{1/2}{(1-1/2)^2} = 2$$

A property of the infinite geometric series

- Thus: $$O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right) = O\Big(n \underbrace{\sum_{h=0}^{\infty} \frac{h}{2^h}}_{2}\Big) = O(n)$$

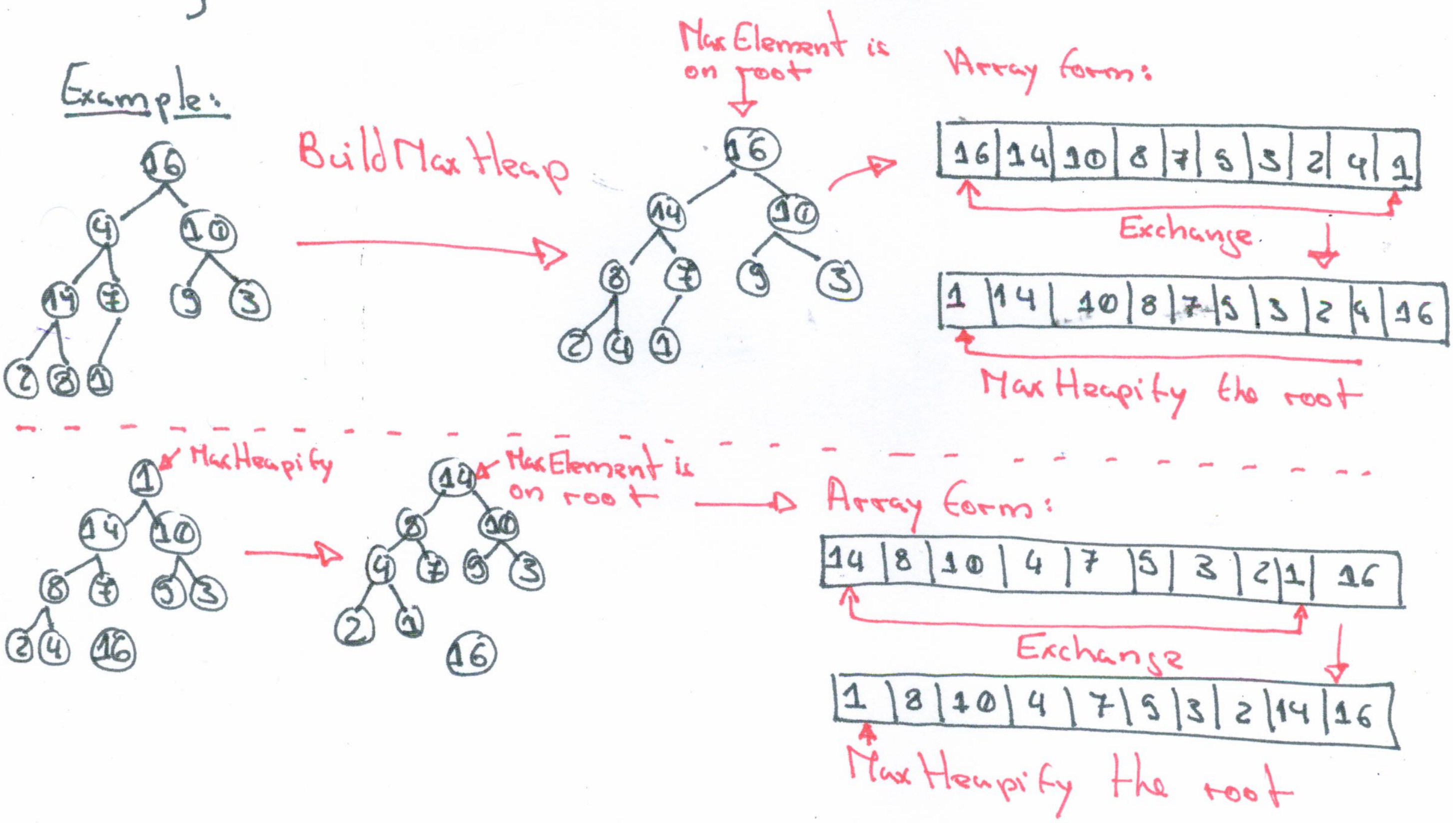
∴ BuildMaxHeap runs in linear time

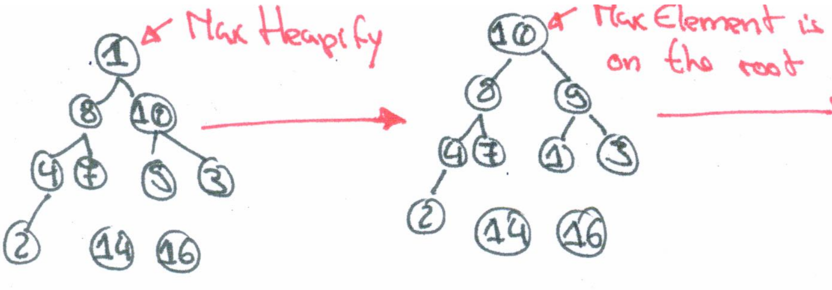
6.4 The Heapsort algorithm

Idea: 1) Build a Max Heap
2) The largest element will be in the root
3) Exchange the first element with the last element
4) Build a MaxHeap again, this time on A[1...length[A]-1]

```
Heapsort (A) {
    BuildMaxHeap(A)
    for i = length[A] to 2 {
        exchange A[1] <-> A[i]
        heapSize[A] = heapSize[A] - 1;
        MaxHeapify(A, 1)
}
```

Example:





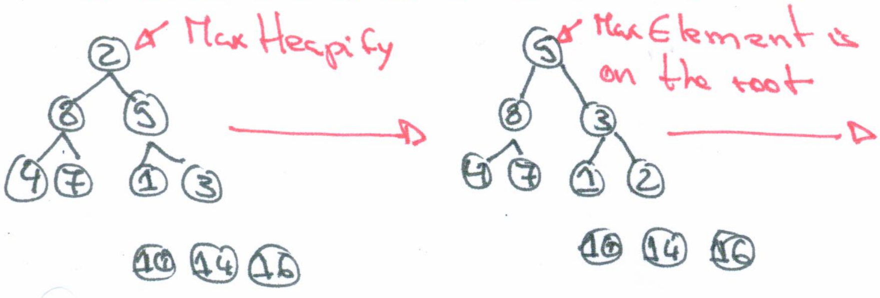
Array form:

10	8	9	4	7	1	3	2	14	16

Exchange these

2	8	9	4	7	1	3	10	14	16

Max Heapify the root



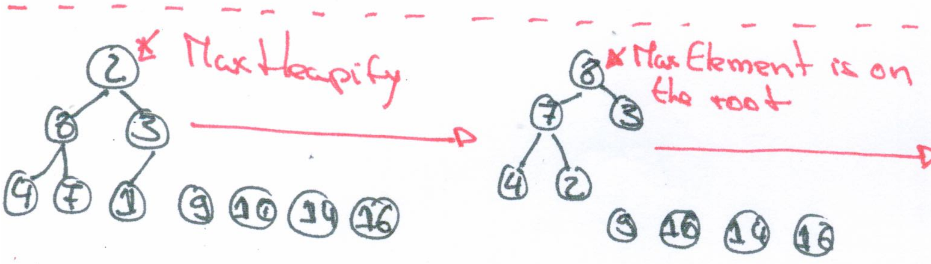
Array form:

9	8	3	4	7	1	2	10	14	16

Exchange these

2	8	3	4	7	1	9	10	14	16

Max Heapify the root



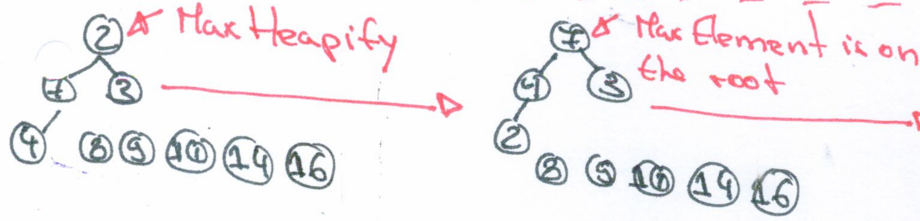
Array form:

8	7	3	4	2	9	10	14	16

Exchange these

2	7	3	4	8	9	10	14	16

Max Heapify the root



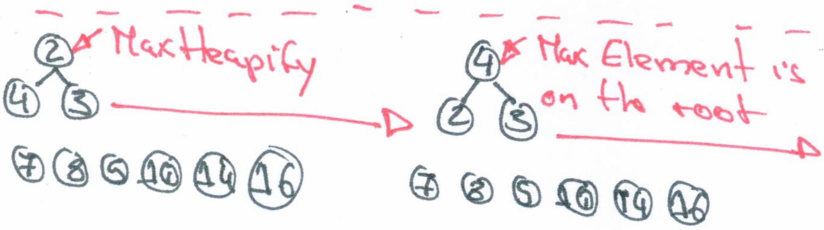
Array form:

7	4	3	2	8	9	10	14	16

Exchange these

2	4	3	7	8	9	10	14	16

Max Heapify the root



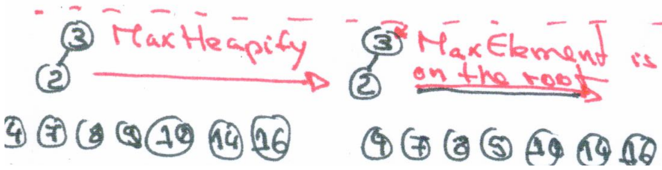
Array form:

4	2	3	7	8	9	10	14	16

Exchange these

3	2	4	7	8	9	10	14	16

Max Heapify the root



Array form:

3	2	4	7	8	9	10	14	16

Exchange

The end!!