The Heap Property

- Heaps also satisfy the *heap property*:

  $A[Parent(i)] \geq A[i]$ for all nodes $i > 1$

  - In other words, the value of a node is at most the value of its parent
  - *Where is the largest element in a heap stored?*

Heap Height

- Definitions:
  - The *height* of a node in the tree = the number of edges on the longest downward path to a leaf
  - The height of a tree = the height of its root
- *What is the height of an n-element heap? Why?*
- This is nice: basic heap operations take at most time proportional to the height of the heap

The Heap Property

- Max-heap : $A[Parent(i)] \geq A[i]$
- Min-heap : $A[Parent(i)] \leq A[i]$
- The height of a node in a tree: the number of edges on the longest simple downward path from the node to a leaf.
- The height of a tree: the height of the root
- The height of a heap: $O(\lg n)$.

Pop Quiz

1. What are the minimum and maximum numbers of elements in a heap of height h ?

- There is a most $2^{h+1} - 1$ vertices in a complete binary tree of height h. Since the lower level need not be filled we may only have 2^h vertices.

Pop Quiz

2. Show that an n-element heap has height lg n

- Since the height of an n-element heap must satisfy that $2^h \le n \le 2^{h+1} - 1 < 2^{h+1}$. we have $h \le \lg n < h + 1$. h is an integer so $h = \lg n$.

Pop Quiz

3. Is the array with values [23,17,14,6,13,10,1,5,7,12] a Max-Heap?

- No

Basic procedures on heap

- **`Max-Heapify`**
- **`Build-Max-Heap`**
- **`Heapsort`**
- **`Max-Heap-Insert`**
- **`Heap-Extract-Max`**
- **`Heap-Increase-Key`**
- **`Heap-Maximum`**

6.2 Maintaining the heap property

`Max-Heapify(A, i)`

- To maintain the max-heap property.
- Assume that the binary trees rooted at `Left(i)` and `Right(i)` are heaps, but that `A[i]` may be smaller than its children, thus violating the heap property.

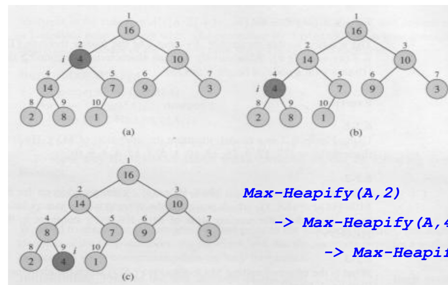
```
Max-Heapify(A,i)
   l = Left(i)
   r = Right(i)
   if l ≤ A.heap-size and A[l] > A[i]
       largest = l
   else largest = i
   if r ≤ A.heap-size and A[r] > A[largest]
       largest = r
   if largest ≠ i
       exchange A[i] ↔ A[largest]
       Max-Heapify(A, largest)
```

Assumption:
Left(i) and Right(i) are max-heaps.

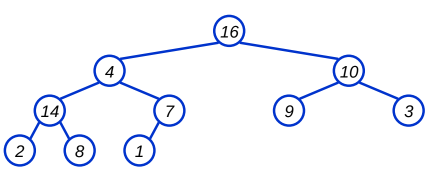
$$T(n) \le T(\frac{2n}{3}) + \Theta(1) \Rightarrow T(n) = O(\lg n)$$

Alternatively O(h) (h: height)

Max-Heapify(A,2)
heap-size[A] = 10

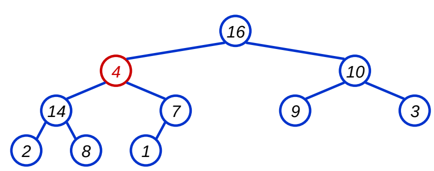


Max-Heapify(A,2)
-> Max-Heapify(A,4)
-> Max-Heapify(A,9)



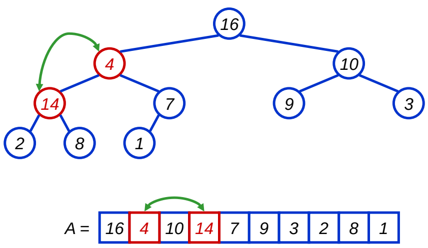
A =

16	4	10	14	7	9	3	2	8	1

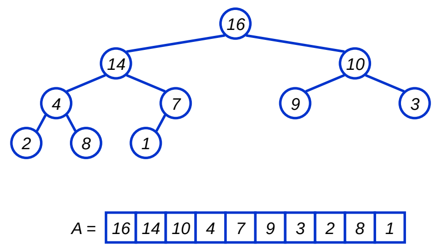


A =

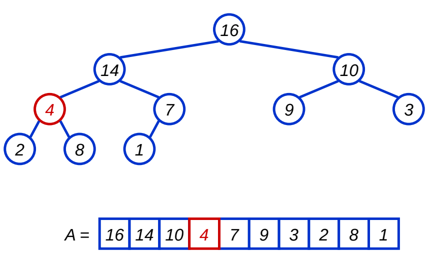
16	4	10	14	7	9	3	2	8	1



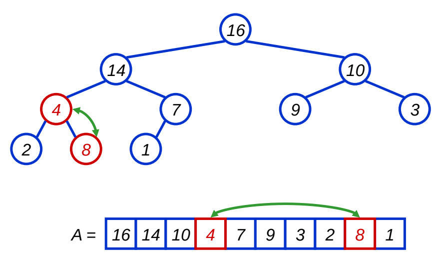
L6.13



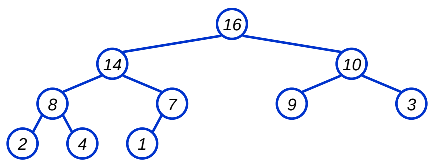
L6.14



L6.15



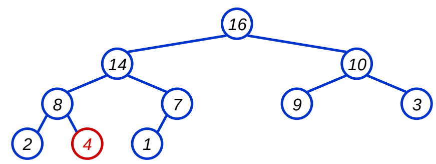
L6.16



A =

16	14	10	8	7	9	3	2	4	1

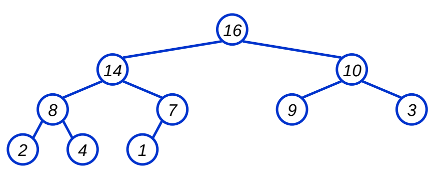
L6.17



A =

16	14	10	8	7	9	3	2	4	1

L6.18



A =

16	14	10	8	7	9	3	2	4	1

L6.19

Running Time for Max-Heapify

```
Max-Heapify(A,i)
 l = Left(i)
 r = Right(i)
 if l ≤ A.heap-size and A[l] > A[i]
        largest  = l
 else  largest = i
 if r ≤ A.heap-size and A[r] > A[largest]
        largest = r
 if largest ≠ i
       exchange A[i] ↔ A[largest]
       Max-Heapify(A, largest)
```

Time to fix node i and its children = ***Θ(1)***

PLUS

Time to fix the subtree rooted at one of i's children = T(size of subree at largest)

L6.20

Running Time for Max-Heapify(A, n)

- Fixing up relationships between i, l, and r takes $\Theta(1)$ time
- $T(n) = T(largest) + \Theta(1)$
- *If the heap at i has n elements, how many elements can the subtrees at l or r have?*
  - Draw it
- Answer: $2n/3$ (worst case: bottom row 1/2 full)
- $largest \leq 2n/3$ (worst case occurs when the last row of tree is exactly half full)
- So time taken by **`Max-Heapify()`** is given by
  $T(n) \leq T(2n/3) + \Theta(1)$

L6.21