Heapsort

Prepared by Suk Jin Lee

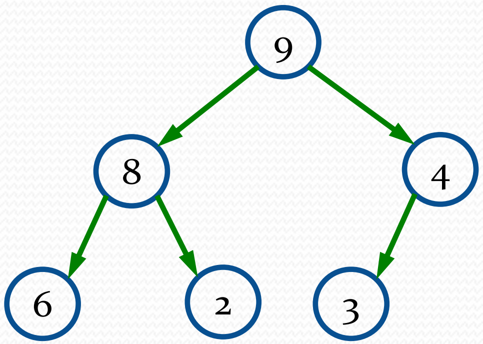
Heaps

- A **heap** is a binary tree with properties:
  - It is complete
    - Each level of tree completely filled
    - Except possibly bottom level (nodes in left most positions)
  - It satisfies heap-order property (two kinds of heaps)
    - Max-heap: for all node i, excluding the root
      - $A[\text{Parent}(i)] \geq A[i]$
    - Min-heap: for all node i, excluding the root
      - $A[\text{Parent}(i)] \leq A[i]$

Heaps

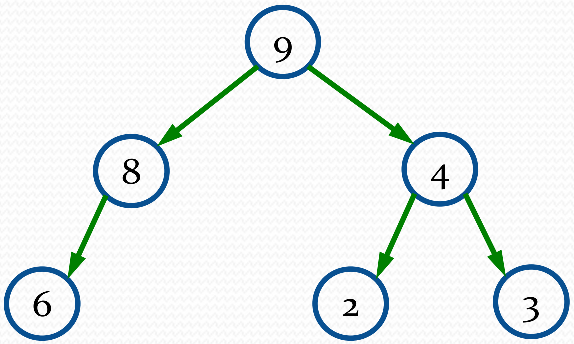
- Which of the following are heaps?

A



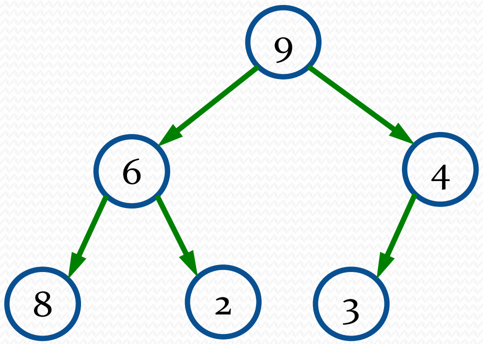
Yes, it is a heap…!

B



No, it is not, b/c it is not complete…!

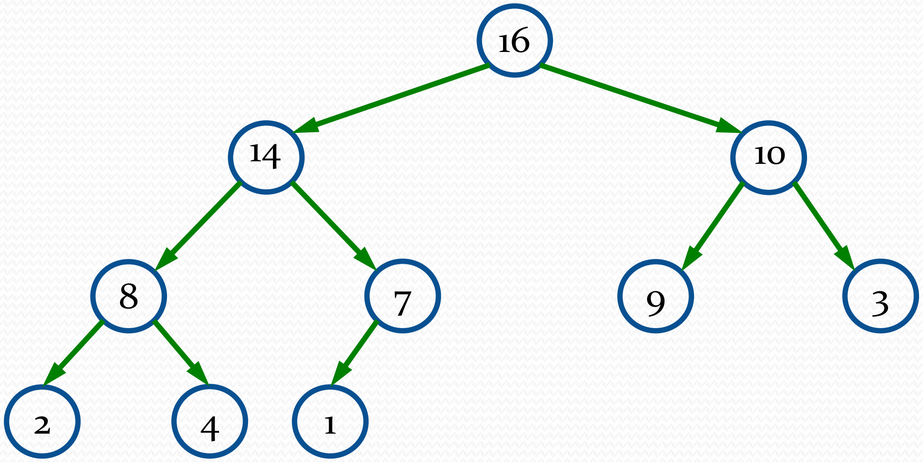
C



Complete! But, it is not, b/c heap-order condition is not satisfied…!

Heaps

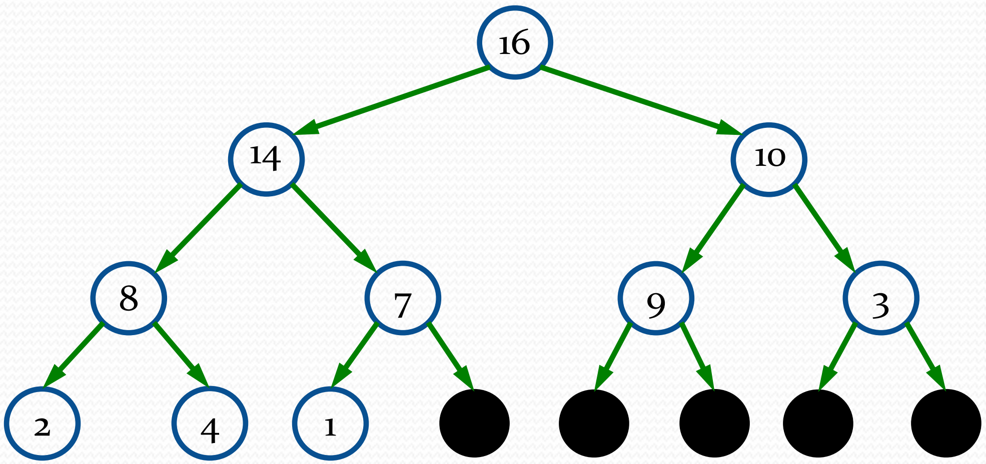
- A heap can be seen as a complete binary tree:



  - What makes a binary tree complete?
  - Is the example above complete?

Heaps

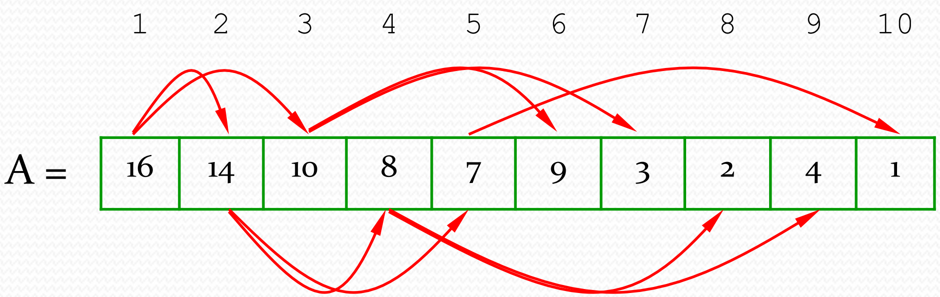
- A heap can be seen as a complete binary tree:

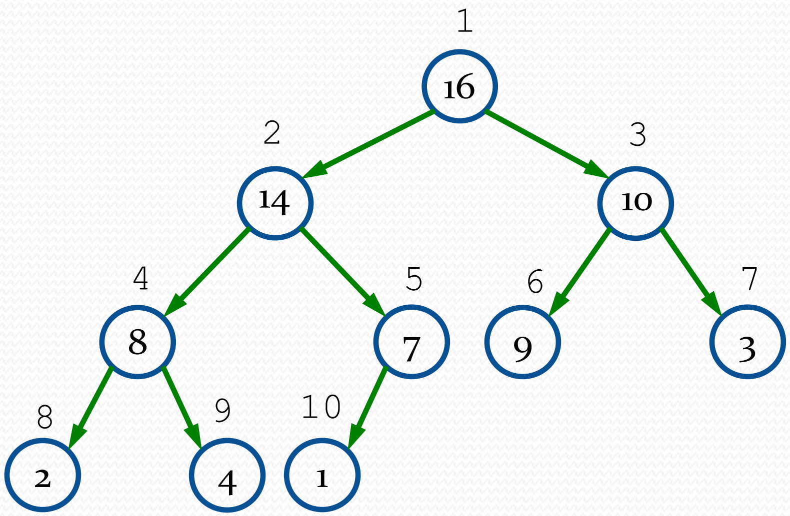


  - The book calls them "nearly complete" binary trees; can think of unfilled slots as null pointers

Heaps

- In practice, heaps are usually implemented as arrays:





Heaps

- To represent a complete binary tree as an array:
  - The root node is A[`1`]
  - Node *i* is A[`i`]
  - The parent of node *i* is A[`i/2`] (note: integer divide)
  - The left child of node *i* is A[`2i`]
  - The right child of node *i* is A[`2i + 1`]

Referencing Heap Elements

- So, we can get

PARENT(i)

1. return $\lfloor i/2 \rfloor$

LEFT(i)

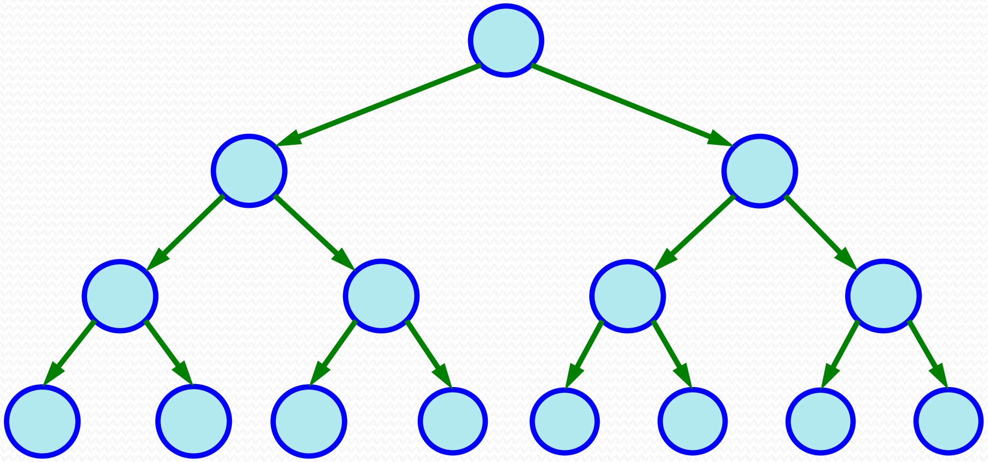
1. return $2 \times i$

RIGHT(i)

1. return $2 \times i + 1$

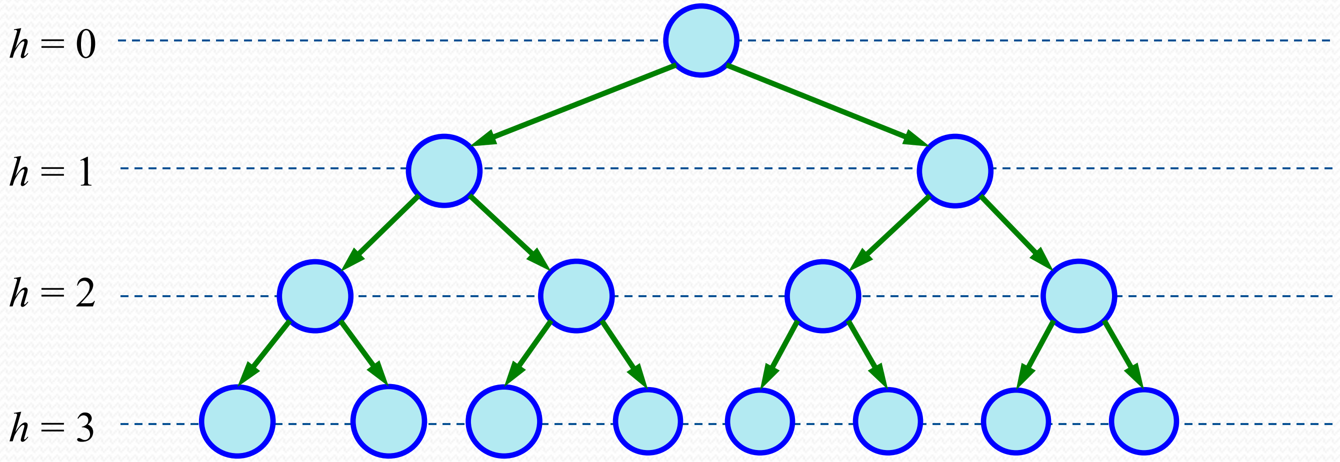
Quiz – 1

- What are the minimum and maximum numbers of elements in a heap of height *h*?



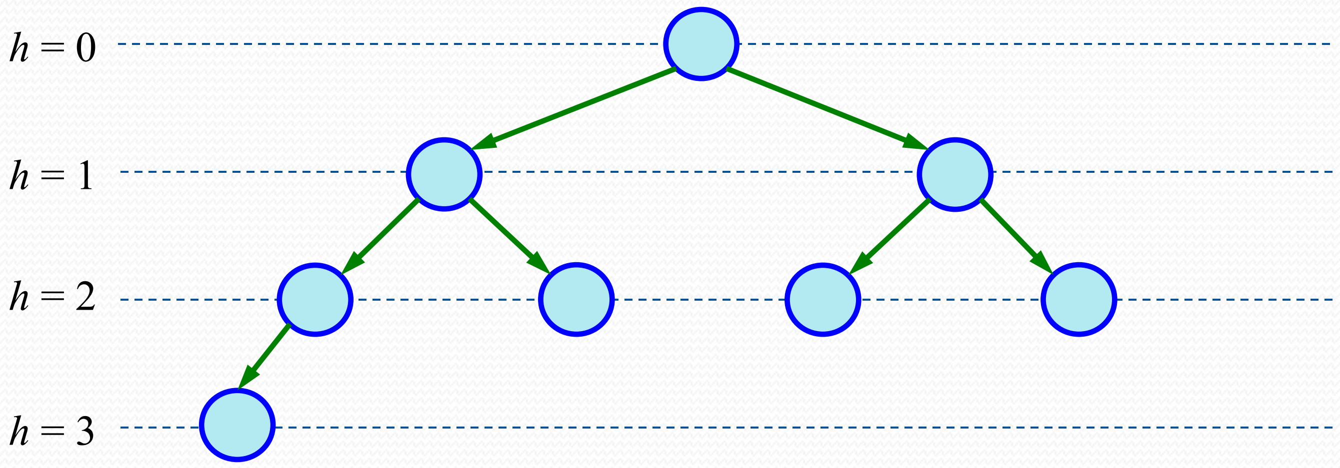
Quiz – 1

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



Quiz – 1

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



Quiz – 1

- What are the minimum and maximum numbers of elements in a heap of height h?
  - Since a heap is an almost-complete binary tree, it has at most $2^{h+1} - 1$ elements (if it is complete)
  - At least $2^h - 1 + 1 = 2^h$ elements
    - If the lowest level has just 1 element and the other levels are complete
  - Therefore

$$2^h \leq n \leq 2^{h+1} - 1$$

Quiz – 2

- Show that an n-element heap has height $\lfloor \lg n \rfloor$.

The Heap Property

- Heaps also satisfy the *heap property*:

  $A[\textsc{Parent}(i)] \geq A[i]$ 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?*
- 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

Heap Height

- *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

Maintaining the heap property

- Max-Heapify
  - Used to maintain the max-heap property
  - Before Max-Heapify, $A[i]$ may be smaller than its children
  - Assume left and right subtrees of i are max-heaps
  - After Max-Heapify, subtree rooted at i is a max-heap

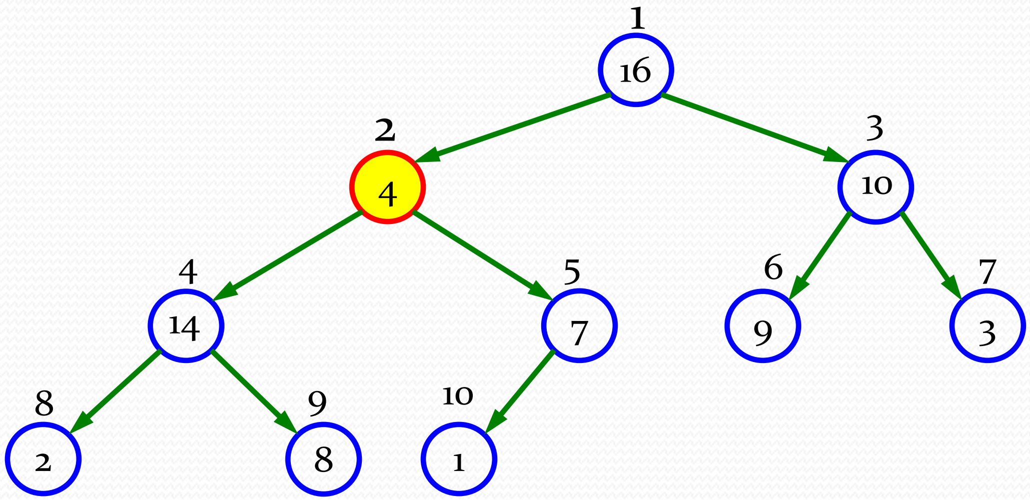
Maintaining the heap property

- 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] with A[largest]
        MAX-HEAPIFY(A, largest)
```

Max-Heapify() Example

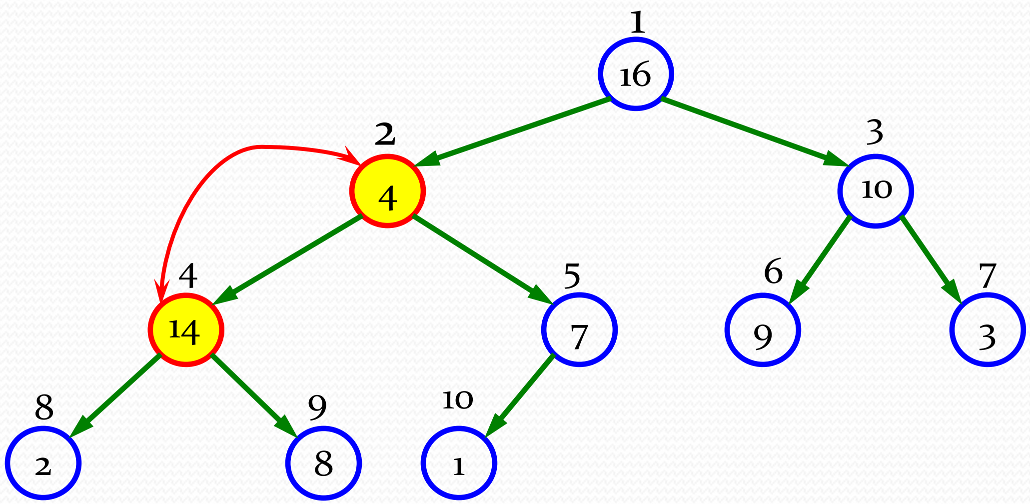
- **MAX-HEAPIFY**$(A, 2)$



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

Max-Heapify() Example

- **MAX-HEAPIFY**$(A, 2)$

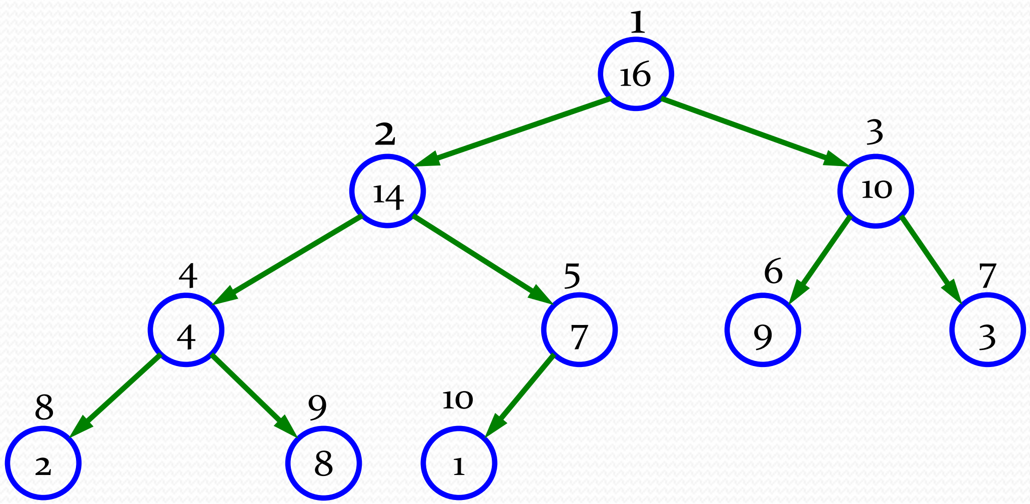


A =

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

Max-Heapify() Example

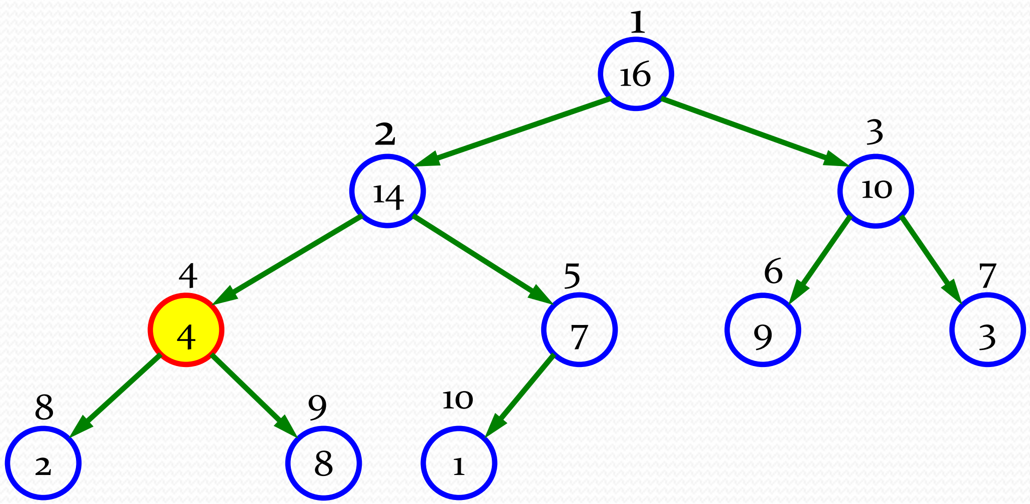
- **MAX-HEAPIFY**$(A, 2)$



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

Max-Heapify() Example

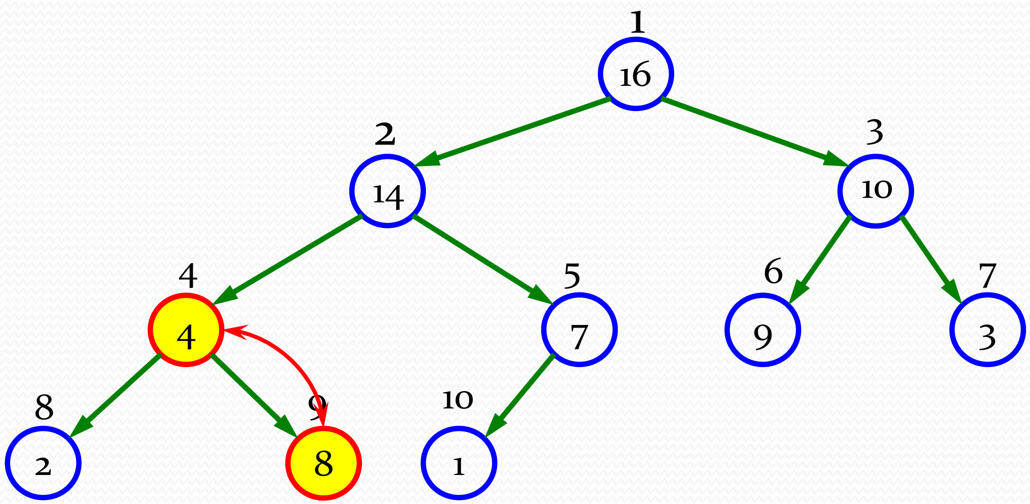
- **MAX-HEAPIFY**(A, 4)



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

Max-Heapify() Example

- **MAX-HEAPIFY**$(A, 4)$

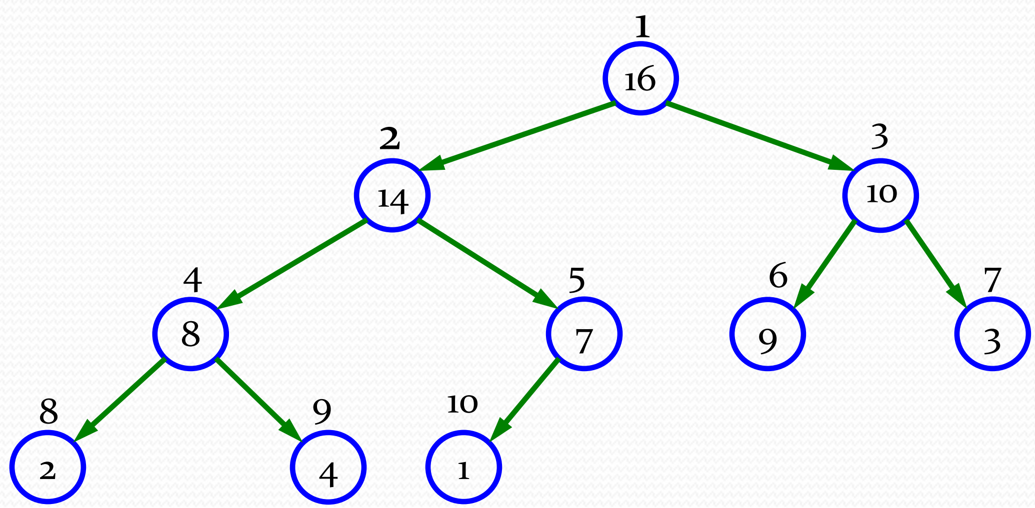


A =

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

Max-Heapify() Example

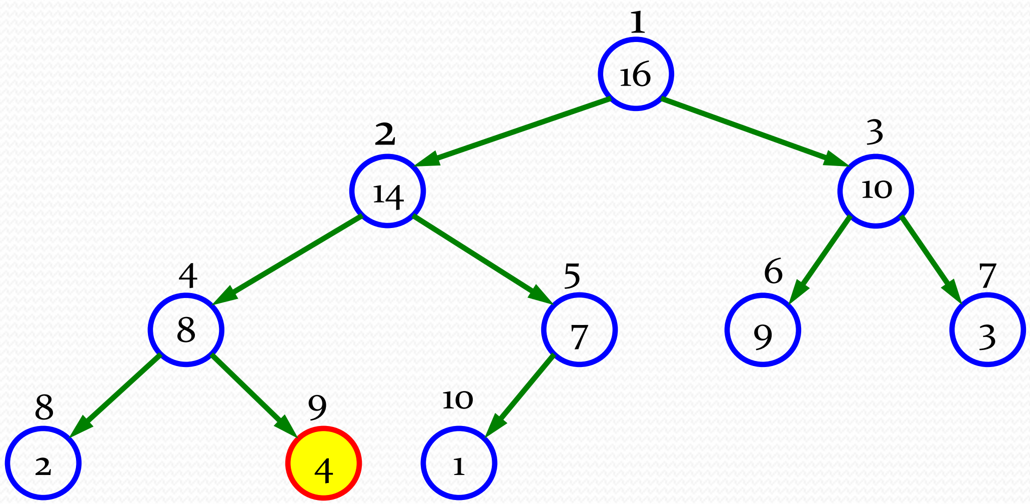
- **MAX-HEAPIFY**$(A, 4)$



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

Max-Heapify() Example

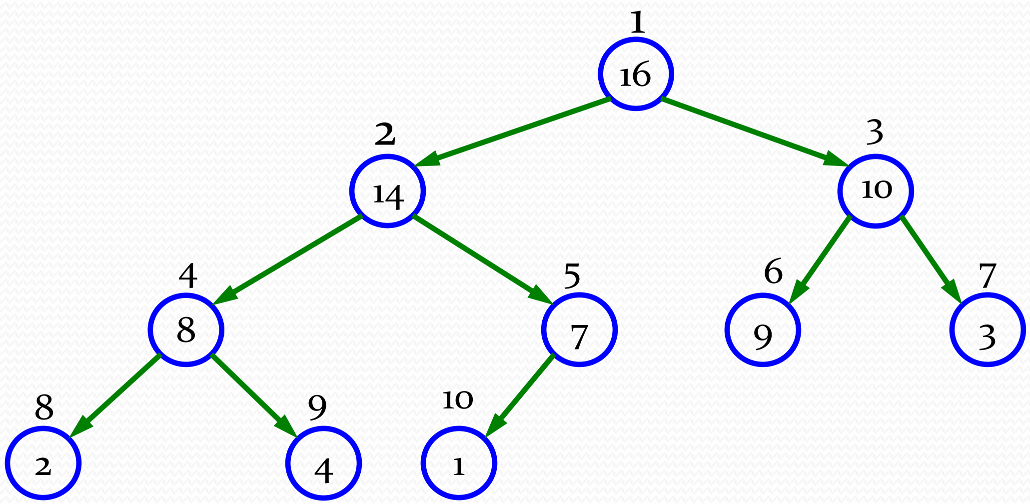
- **MAX-HEAPIFY**$(A, 9)$



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

Max-Heapify() Example

- **MAX-HEAPIFY**(A, 9)



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

Analyzing Heapify(): Informal

- Aside from the recursive call, what is the running time of Heapify()?
- How many times can Heapify() recursively call itself?
- What is the worst-case running time of Heapify() on a heap of size *n*?

Analyzing Heapify(): Formal

- Fixing up relationships between *i*, *l*, and *r* takes $\Theta(1)$ time
- If the heap at *i* has *n* elements, how many elements can the subtrees at *l* or *r* have?
  - Answer: $2n/3$ (worst case: bottom level of tree 1/2 full)
- So time taken by Heapify() is given by

$$T(n) \leq T(2n/3) + \Theta(1)$$

Analyzing Heapify(): Formal

- So we have

$$T(n) \leq T(2n/3) + \Theta(1)$$

  - $a = 1$, $b = 3/2$, $f(n) = \Theta(1)$
  - $f(n) = \Theta\left(n^{\log_b a}\right) = \Theta(n^{\log_{3/2} 1}) = \Theta(n^0) = \Theta(1)$
- By case 2 of the Master Theorem
  - $T(n) = \Theta\left(n^{\log_b a} \lg n\right) = \Theta(\lg n)$
- Thus, Heapify() takes linear time

Building a heap

- Use the procedure Max-Heapify in a bottom-up manner to convert an array $A[1..n]$, where $n = A.length$, into a max-heap.

Build-Max-Heap(A)

$A.heap\text{-}size$ = A.length
for $i = \lfloor A.length / 2 \rfloor$ **downto** 1 O(n)
  Max-heapify(A, *largest*) O(lg n)

- *Simple upper bound*
  - Each call to *Max-Heapify* costs O(lg n) time, and *Build-Max-Heap* make O(n) such call.
    Thus, the running time is O(n lg n)

Building a heap

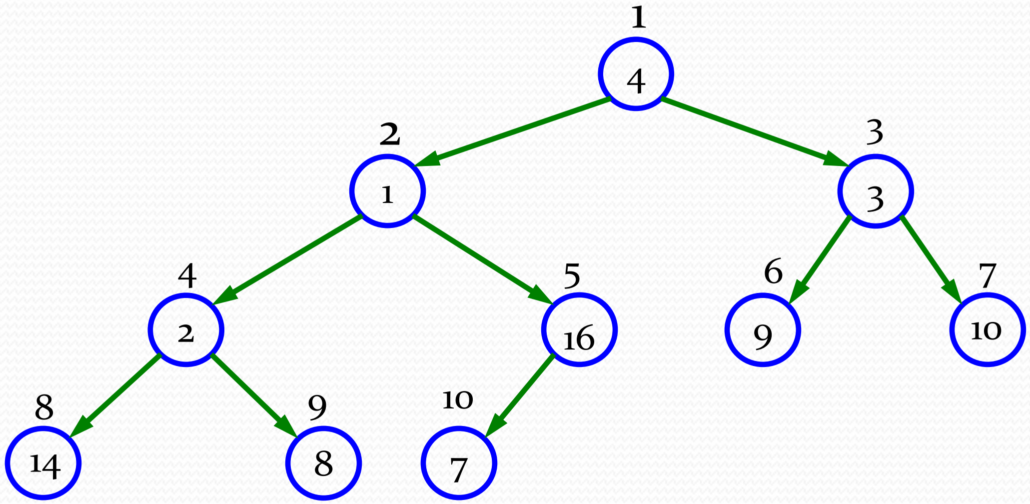
- Tight bound
  - n-element heap has height $\lfloor \lg n \rfloor$
  - At most $\lceil n/2^{h+1} \rceil$ nodes of any height
  - Time required by Max-Heapify when called on a node of any height h is $O(h)$
  - Total cost

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = O(n)$$

    - We can build a max-heap from an unordered array in linear time

Building a heap Example

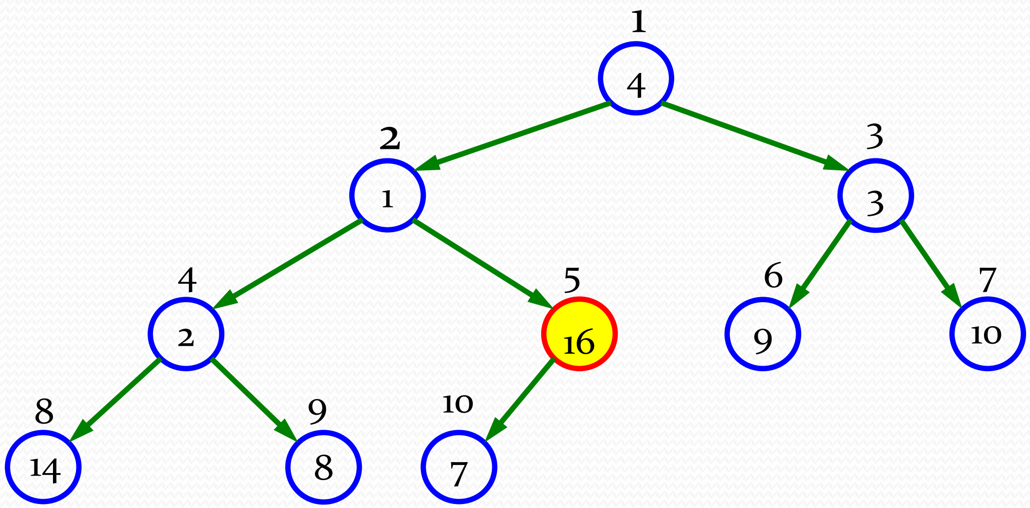
- Work through example: 10-element input array A



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

Building a heap Example

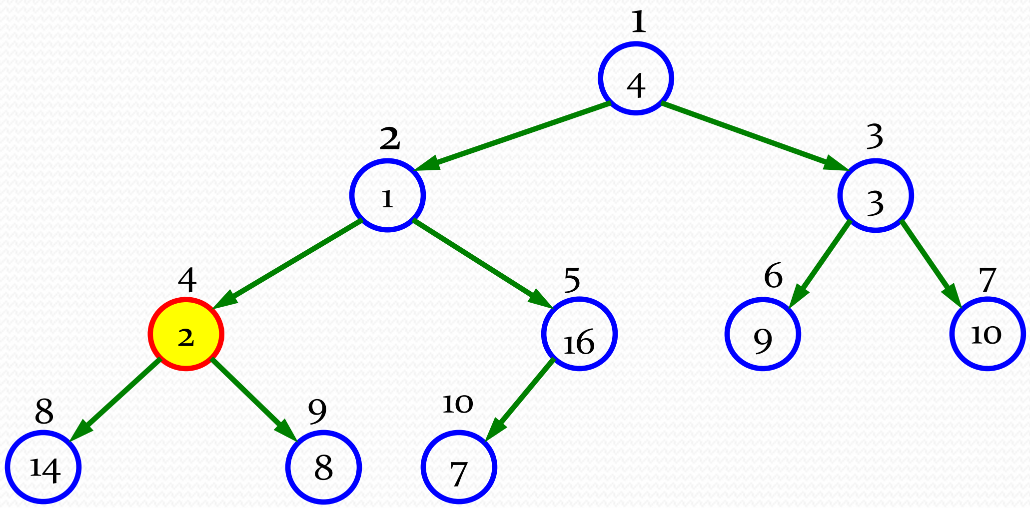
- i = 5; before the call MAX-HEAFIFY(A, i)



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

Building a heap Example

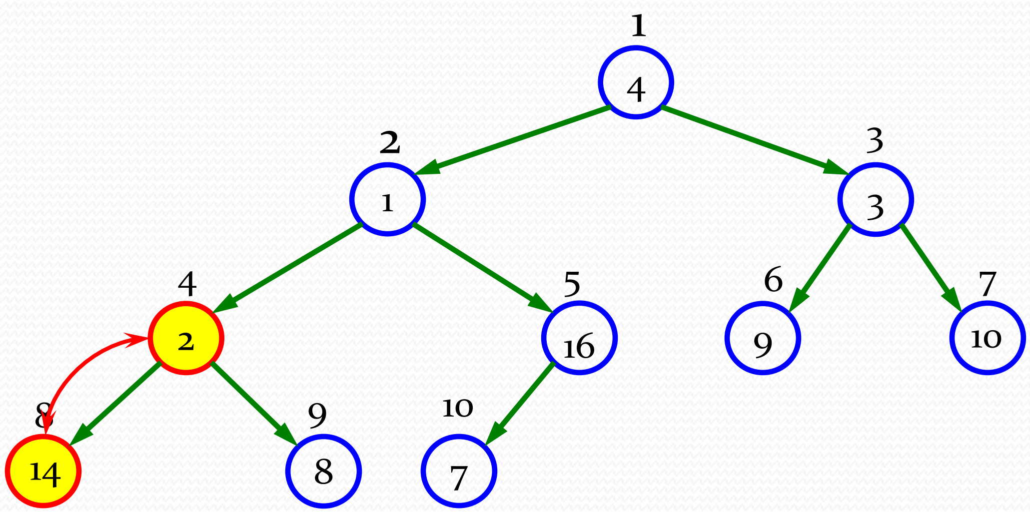
- $i = 4$



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

Building a heap Example

- i = 4; call MAX-HEAFIFY(A, i)

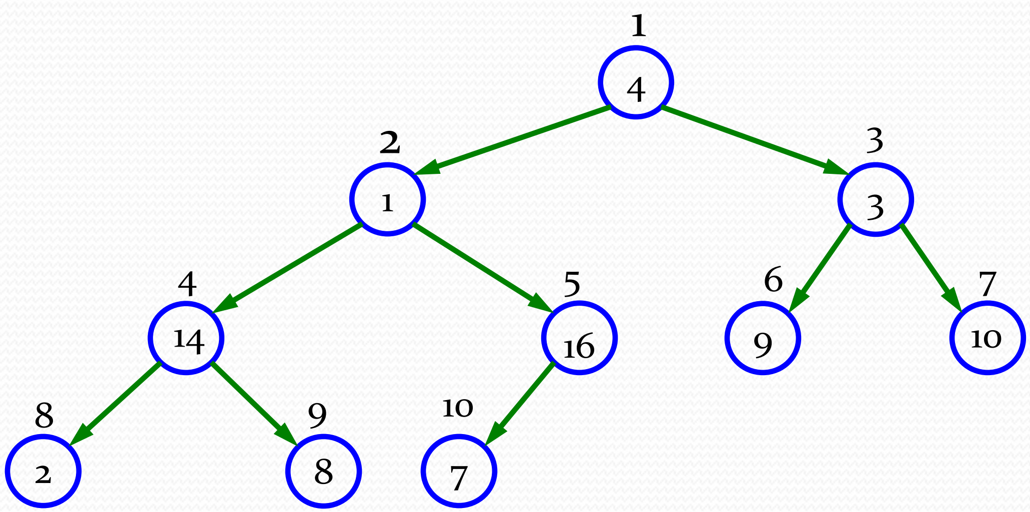


A =

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

Building a heap Example

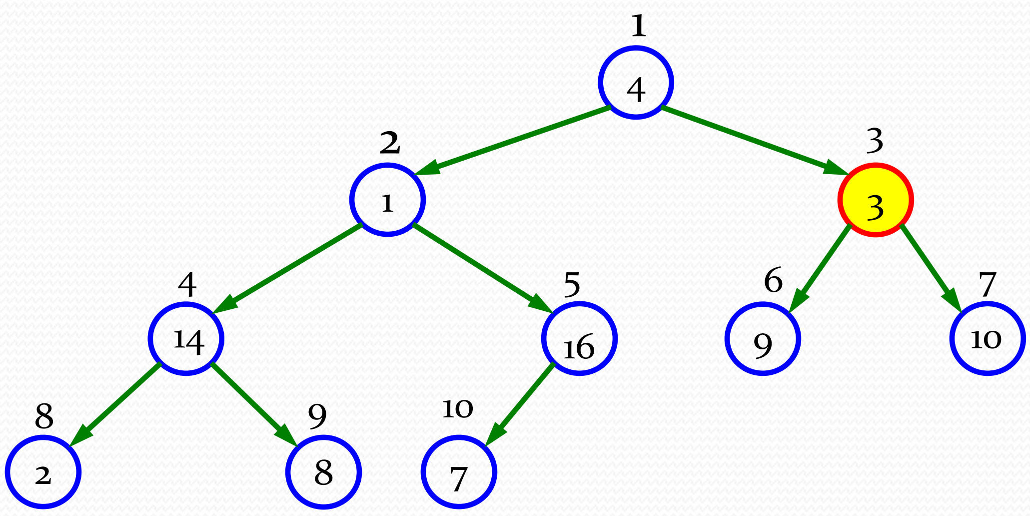
- i = 4; after the call MAX-HEAFIFY(A, i)



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

Building a heap Example

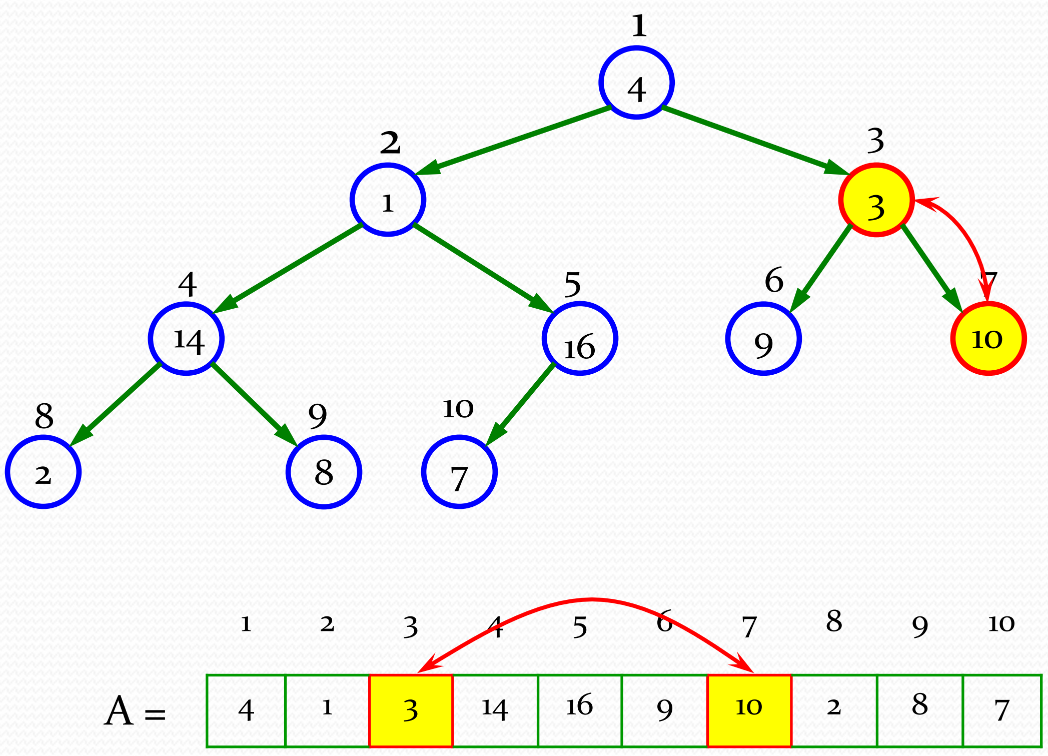
- $i = 3$



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

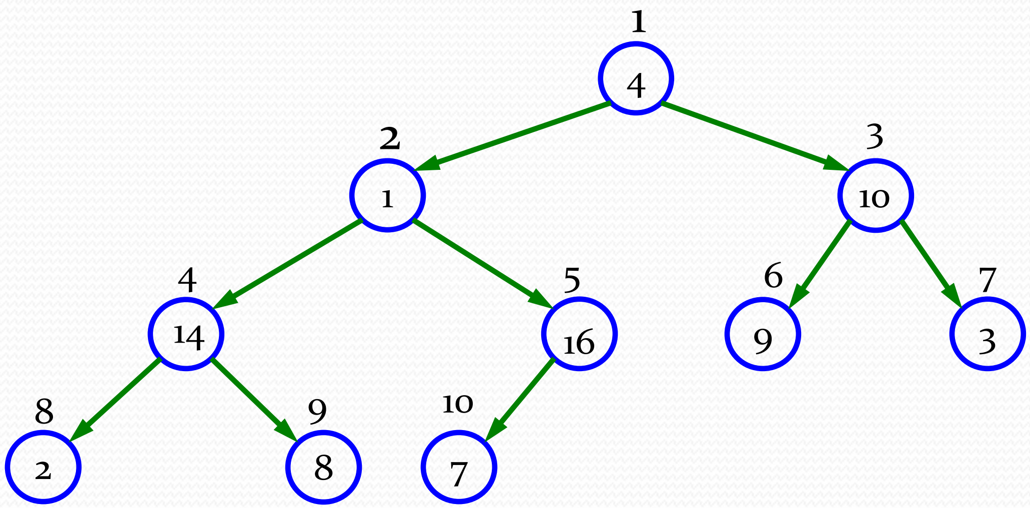
Building a heap Example

- i = 3; call MAX-HEAFIFY(A, i)



Building a heap Example

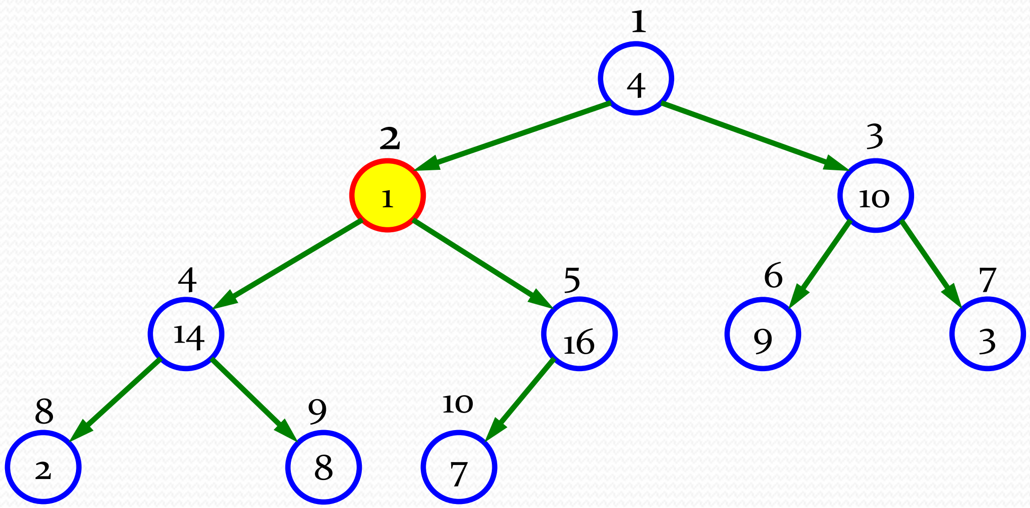
- *i* = 3; after the call MAX-HEAFIFY(A, *i*)



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

Building a heap Example

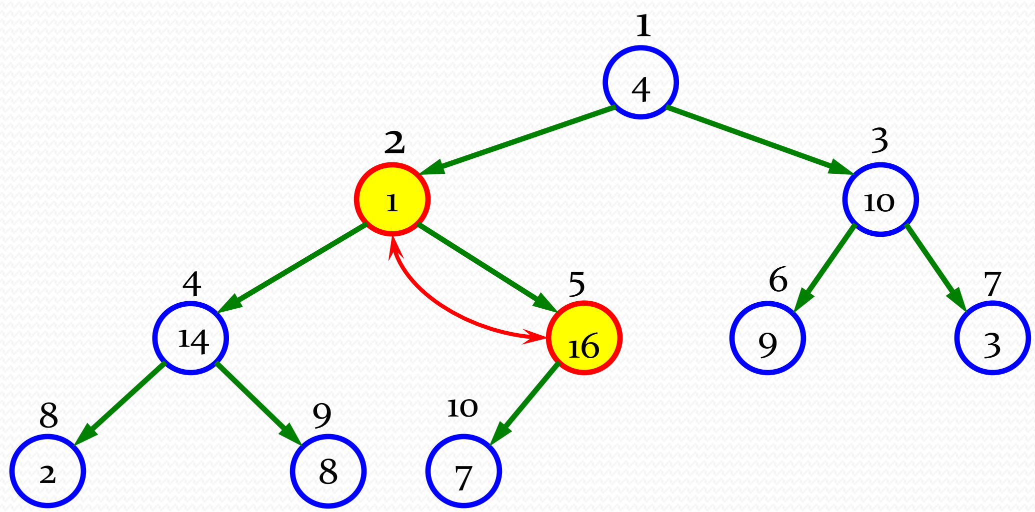
- $i = 2$



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

Building a heap Example

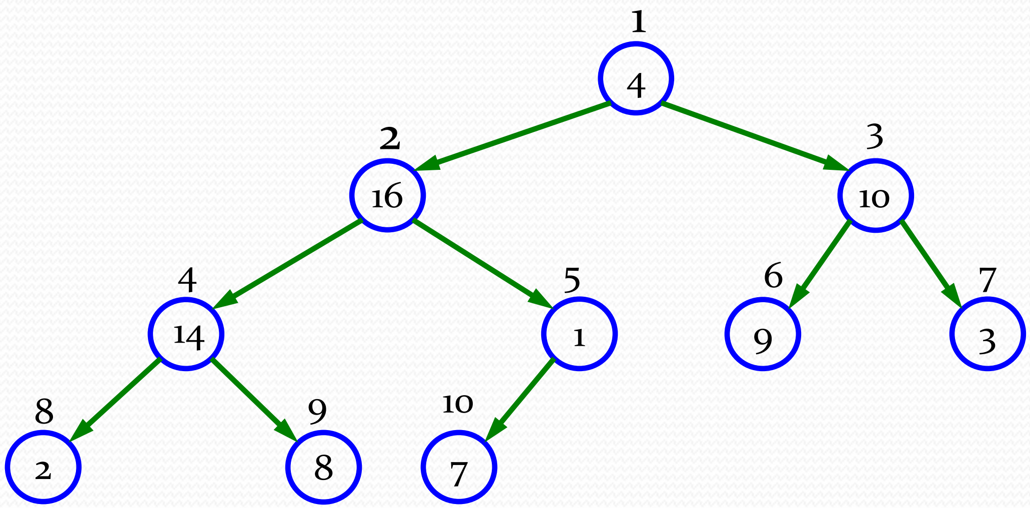
- i = 2; call MAX-HEAFIFY(A, i)



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

Building a heap Example

- $i = 2$

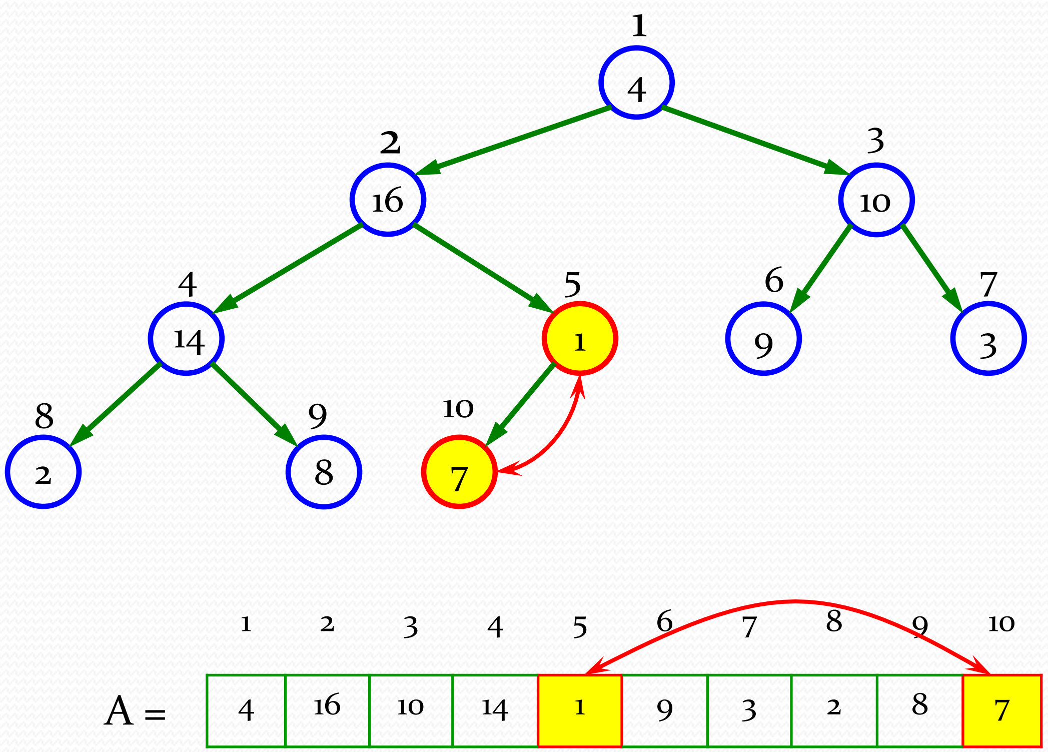


A =

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

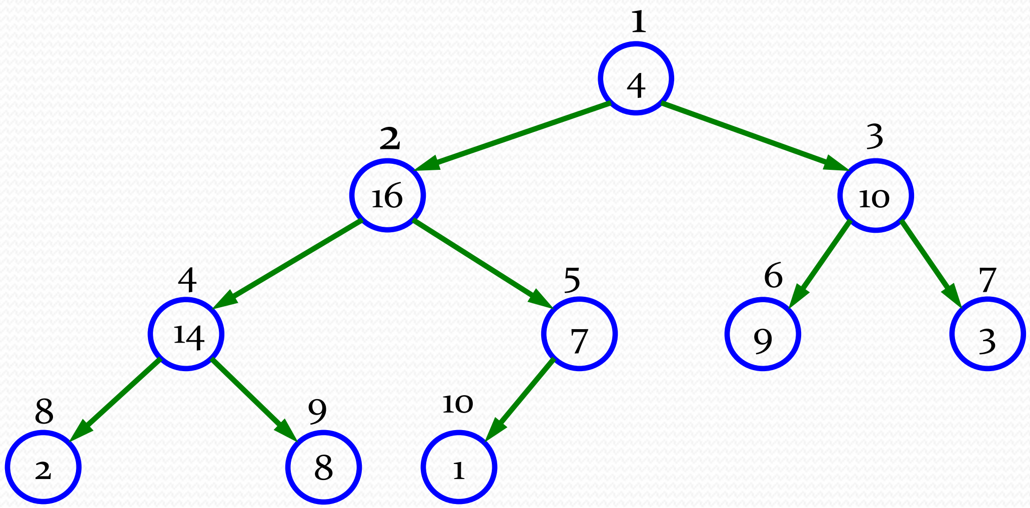
Building a heap Example

- i = 2; recursively call MAX-HEAFIFY(A, *largest*)



Building a heap Example

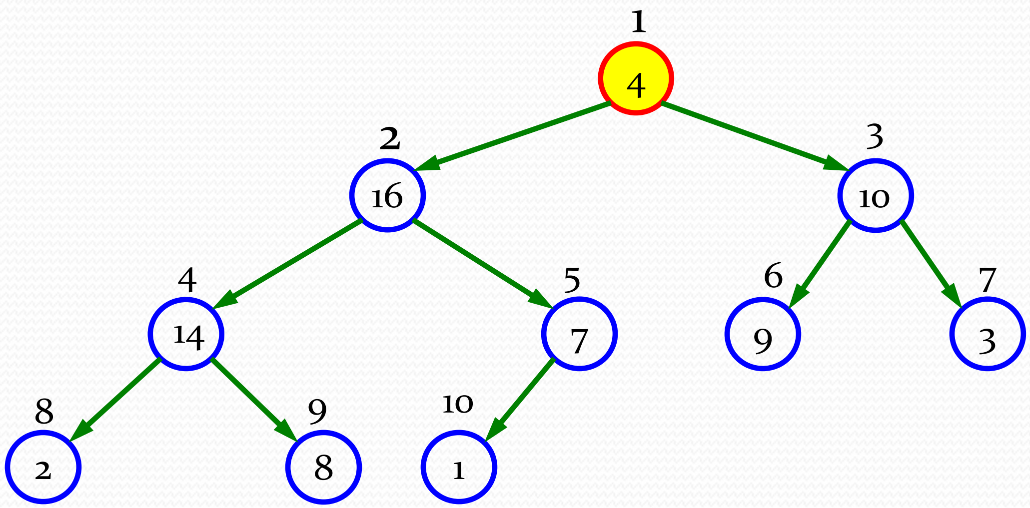
- $i = 2$



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

Building a heap Example

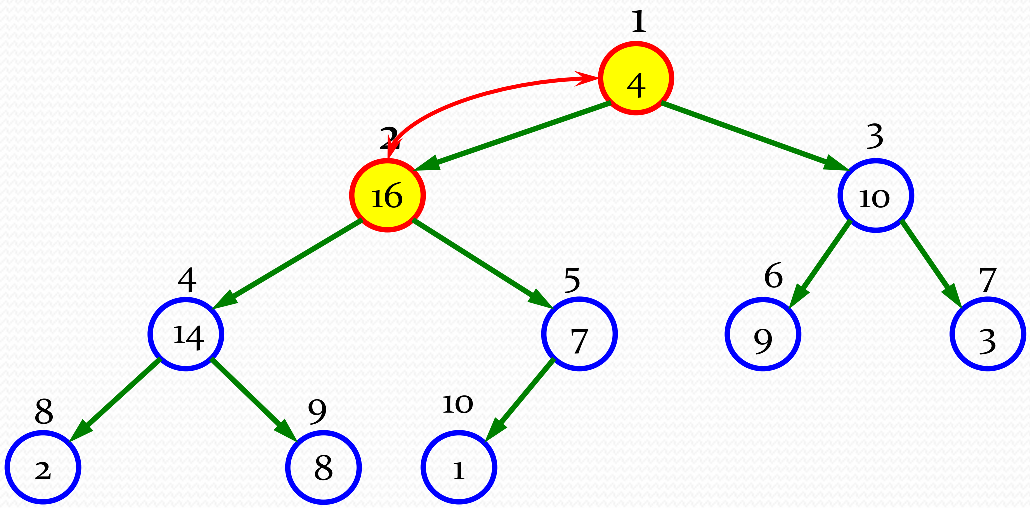
- $i = 1$



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

Building a heap Example

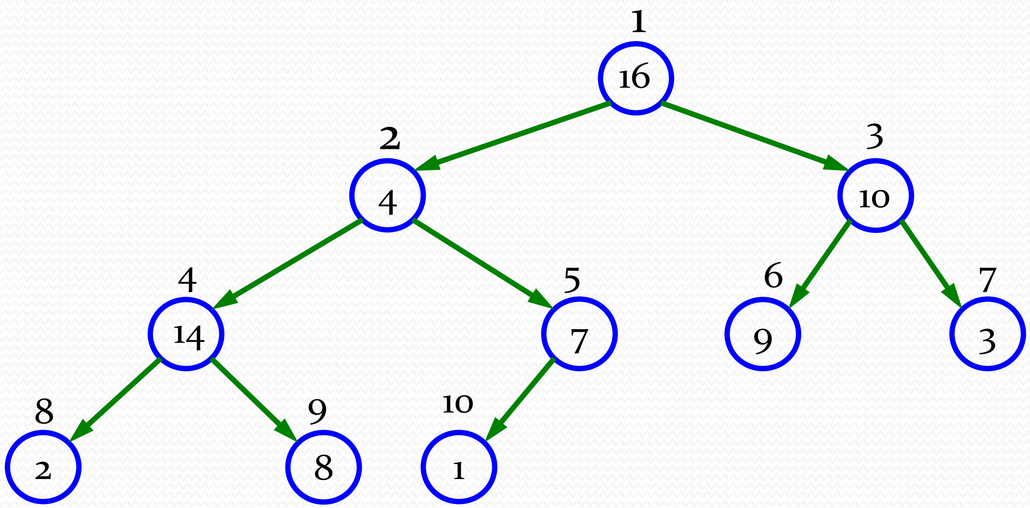
- i = 1; call MAX-HEAFIFY(A, i)



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

Building a heap Example

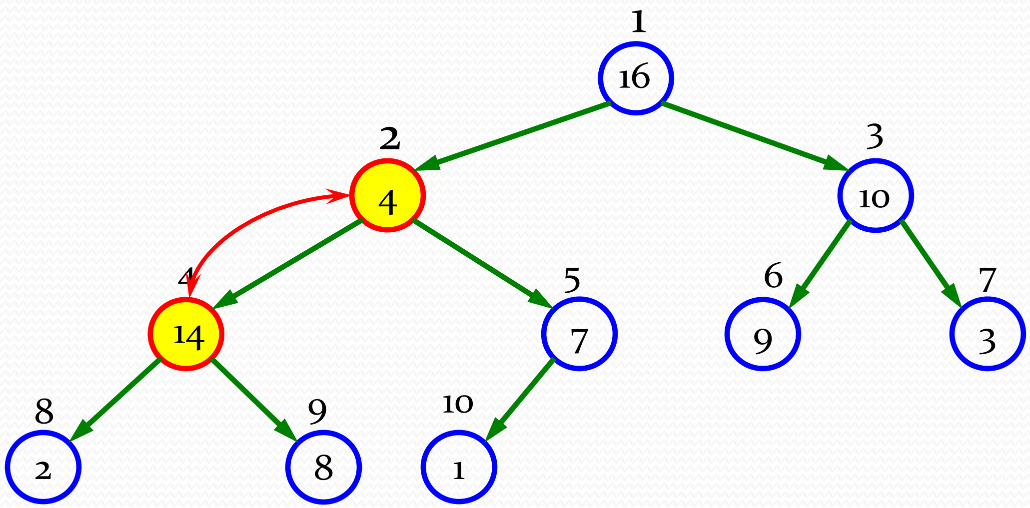
- $i = 1$



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

Building a heap Example

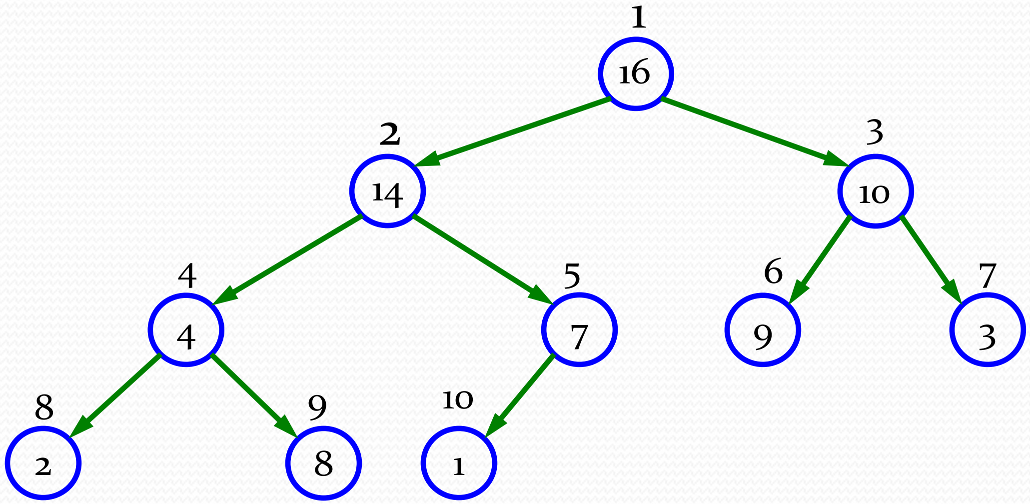
- *i* = 1; recursively call MAX-HEAFIFY(A, *largest*)



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

Building a heap Example

- $i = 1$

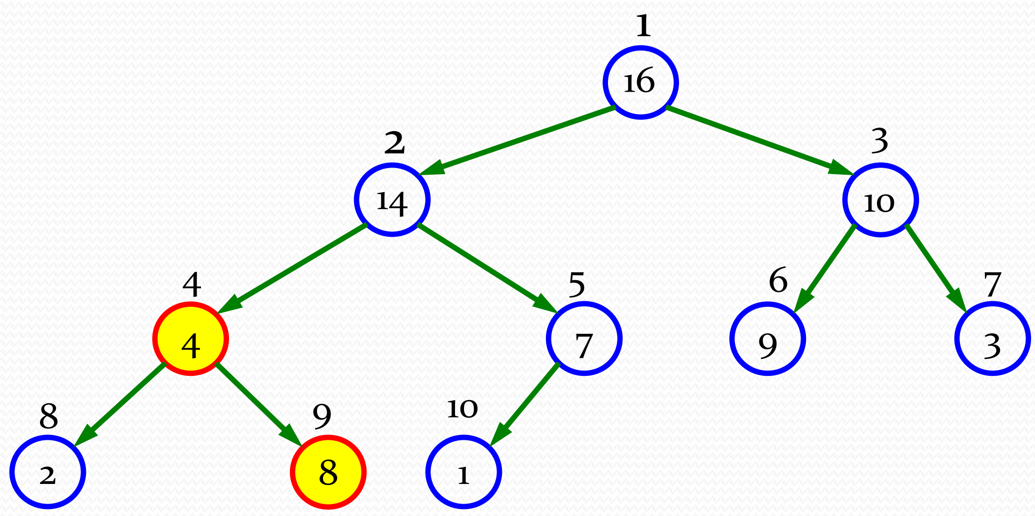


A =

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

Building a heap Example

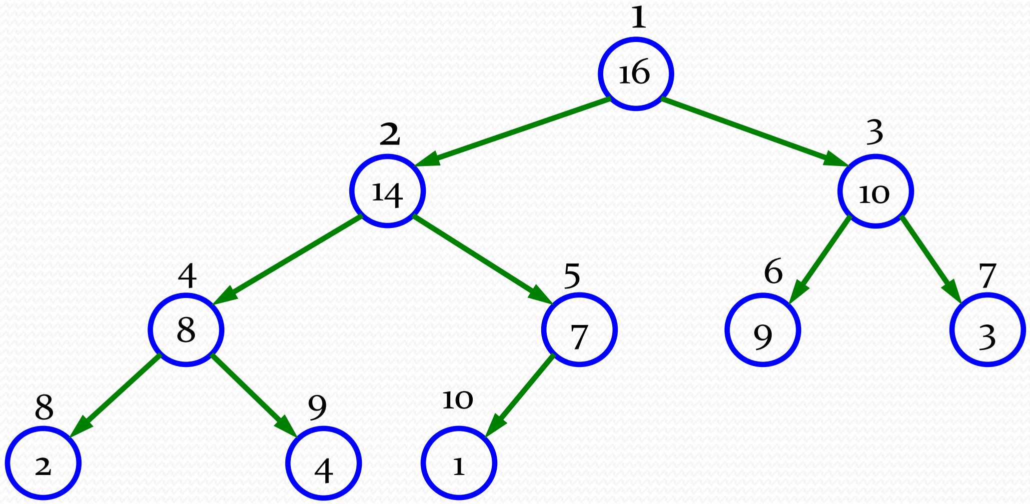
- i = 1; recursively call MAX-HEAFIFY(A, *largest*)



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

Building a heap Example

- The max-heap after BUILD-MAX-HEAP finishes



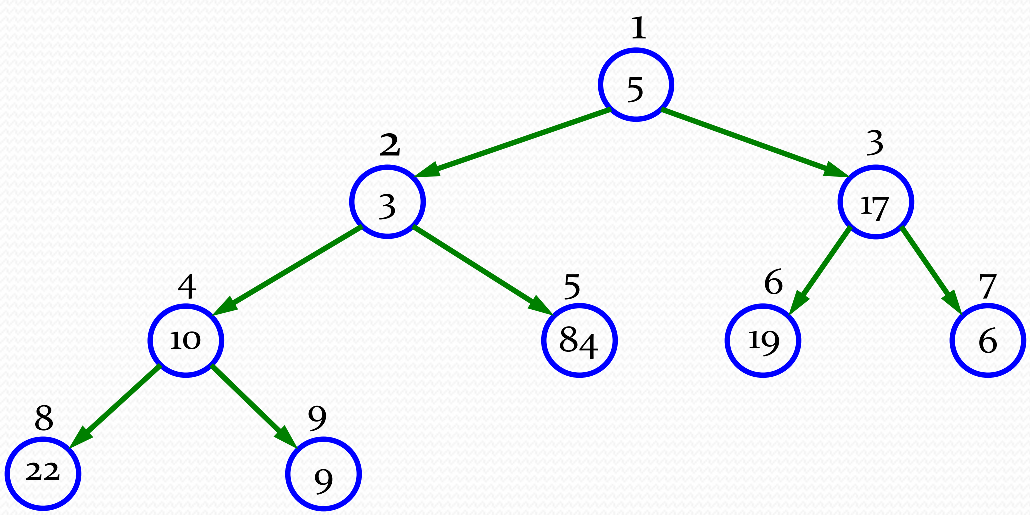
	1	2	3	4	5	6	7	8	9	10
A =	16	14	10	8	7	9	3	2	4	1

Quiz – 3

- Using the previous figure model, illustrate the operation of BUILD-MAX-HEAP on the array
  $A = \langle 5, 3, 17, 10, 84, 19, 6, 22, 9 \rangle$

Quiz – 3

- Using the previous figure model, illustrate the operation of BUILD-MAX-HEAP on the array $A = \langle 5, 3, 17, 10, 84, 19, 6, 22, 9 \rangle$



	1	2	3	4	5	6	7	8	9
A =	5	3	17	10	84	19	6	22	9

Heapsort algorithm

- Heapsort algorithm
  - Build a max-heap on the input array $A[1..n]$ ($n = A.length$)
  - Root $A[1]$: the maximum element of the array A
  - Put $A[1]$ into its correct final position $A[n]$
  - Discard node from the heap: $A.heap\text{-}size - 1$
  - Restore the max-heap property
  - Repeats this process until $A.heap\text{-}size = 2$

Heapsort algorithm

HEAPSORT(*A*)

BUILD-MAX-HEAP(*A*)	$O(n)$
for *i* = *A.length* **downto** 2	$n - 1$
exchange A[1] with A[i]	$n - 1$
A.heap-size = A.heap-size – 1	$n - 1$
MAX-HEAPIFY(*A*, 1)	$O(\lg n)$

Thus the total time taken by HeapSort()

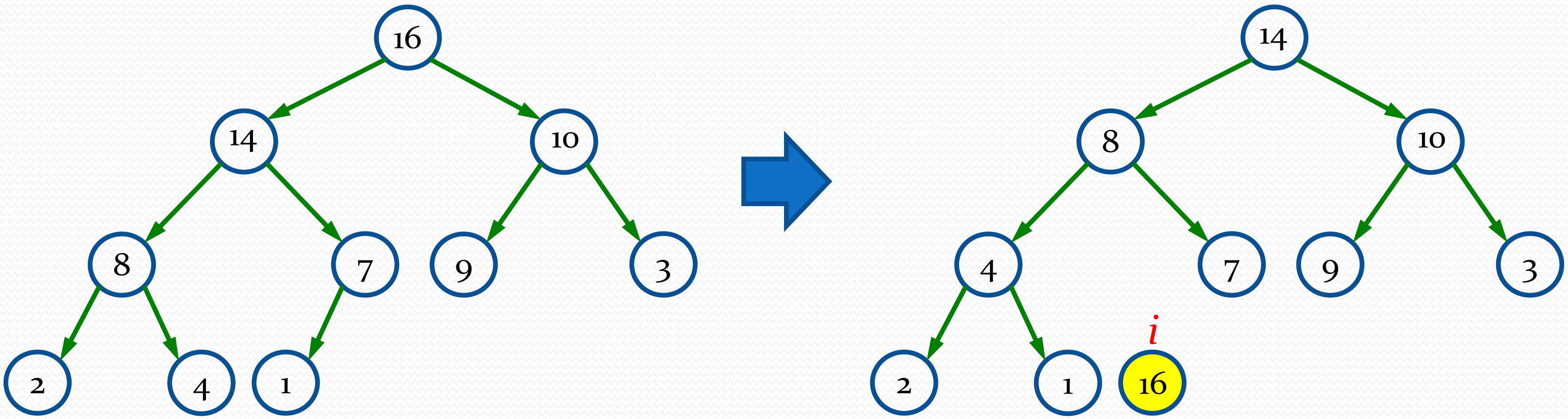
$= O(n) + (n - 1)\, O(\lg n)$

$= O(n) + O(n \lg n)$

$= \mathbf{O(n \lg n)}$

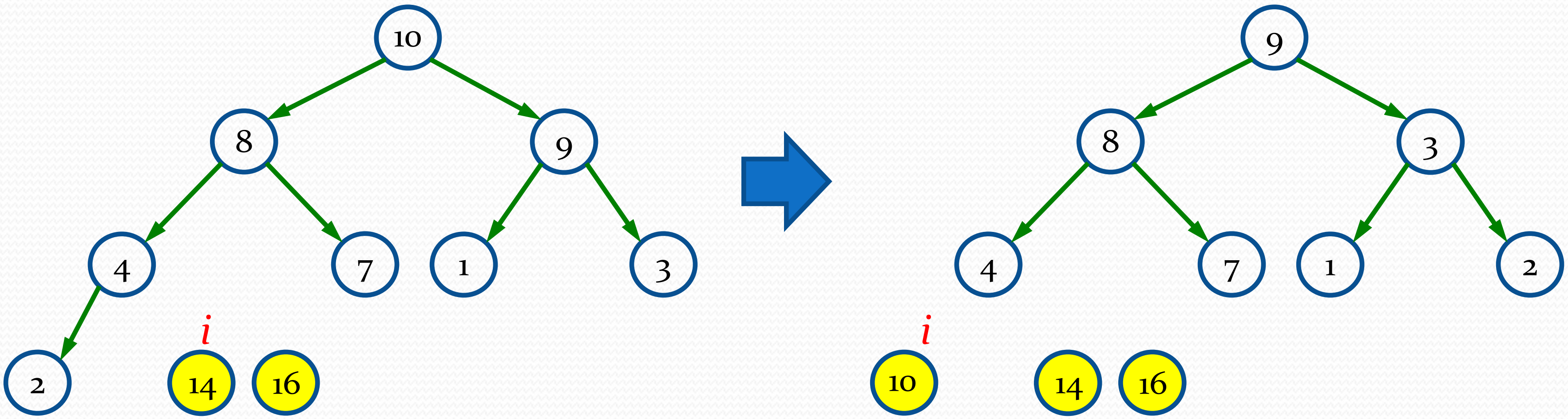
Example

- Operation of Heapsort



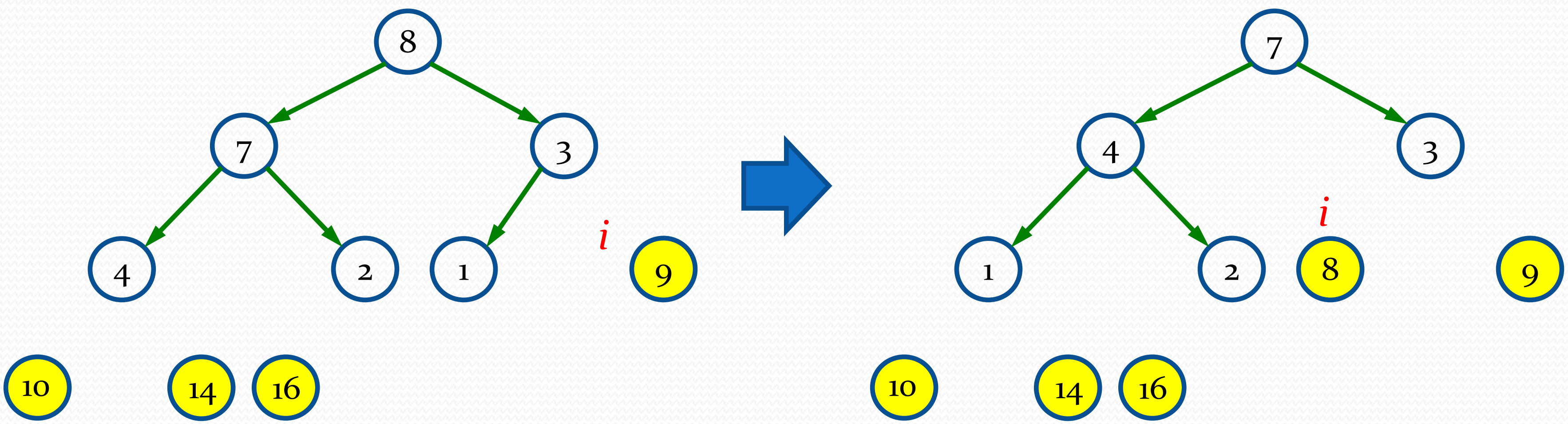
Example

- Operation of Heapsort



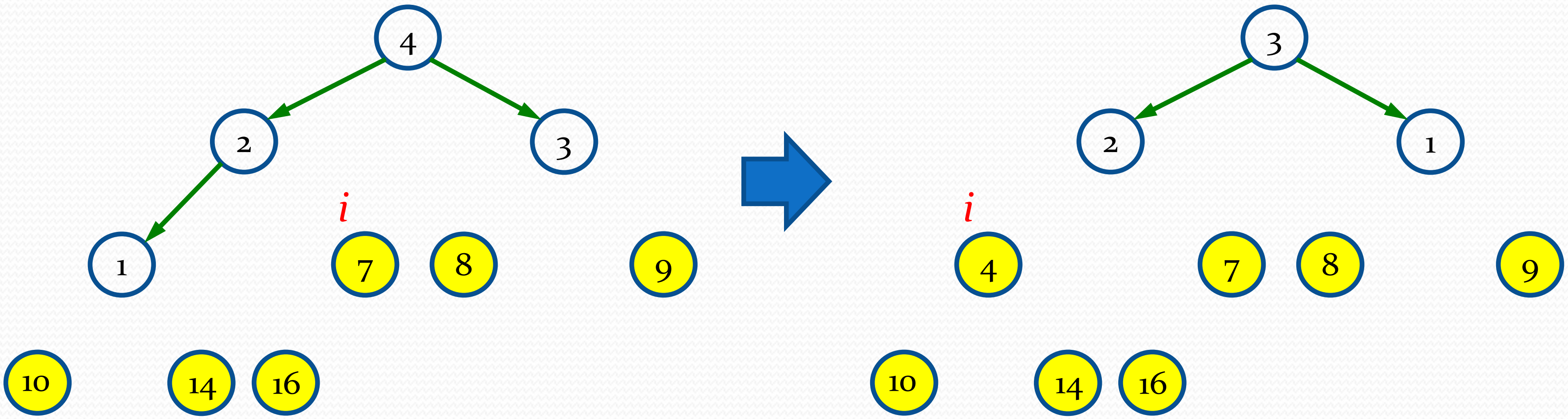
Example

- Operation of Heapsort



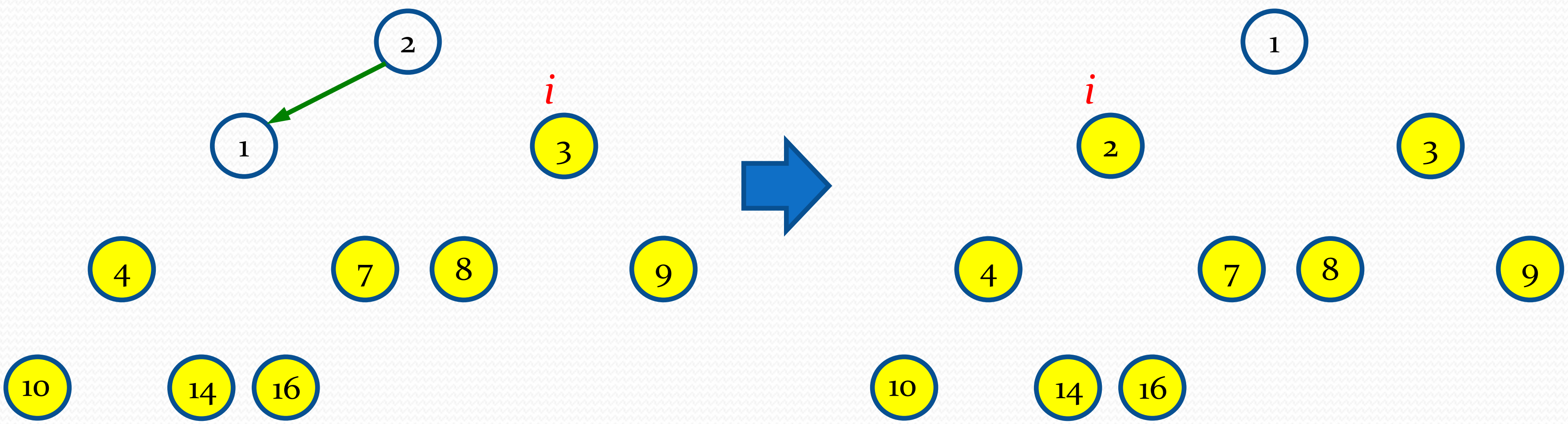
Example

- Operation of Heapsort



Example

- Operation of Heapsort



A	1	2	3	4	7	8	9	10	14	16