Kourosh Davoudi
kourosh@ontariotechu.ca

Lecture 3: Sort Algorithms I

CSCI 3070U: Design and Analysis of Algorithms

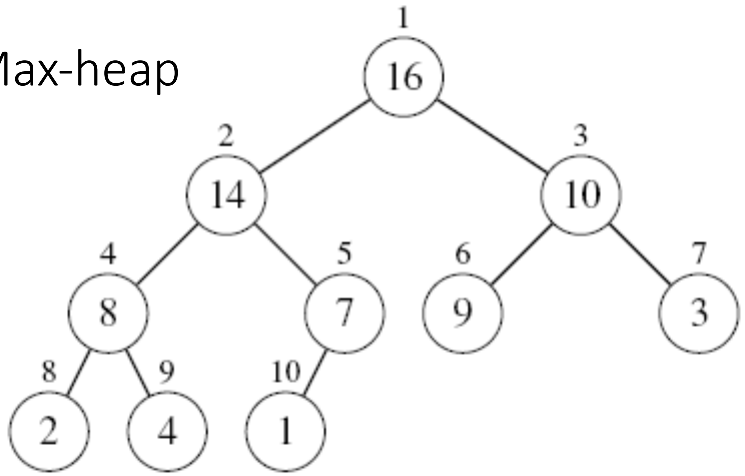
Learning Outcomes

- What is heap?
- Operations on heap
- Applications:
  - Heap Sort
  - Priority Queue

What is Heap?

- Heap properties:
  - A binary heap is an array which encodes a complete binary tree
  - The height of the tree is, thus, $\Theta(\log_2 n)$
  - Strict ordering:
    - A heap is ordered from parent to child, but not between siblings

Max-heap



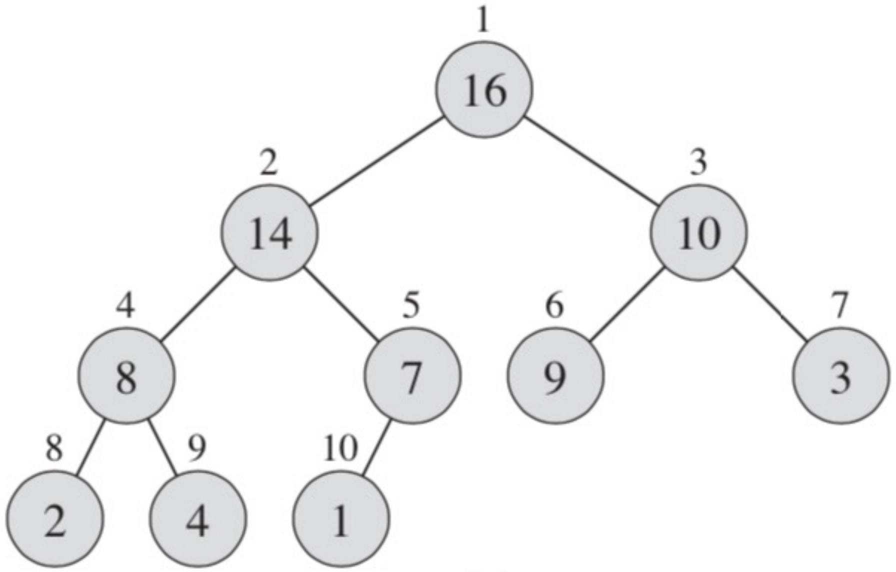
$A[\textsc{Parent}(i)] \geq A[i]$ In Max-heap

OR

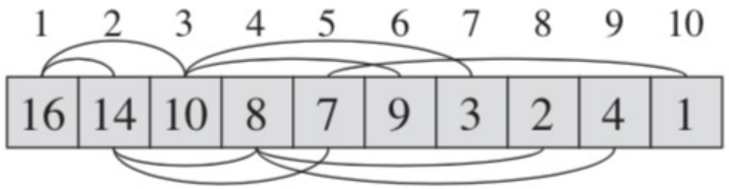
$A[\textsc{Parent}(i)] \leq A[i]$ In Min-heap

Data Structure for Heap Tree

- Complete binary tree can be easily stored in an array !



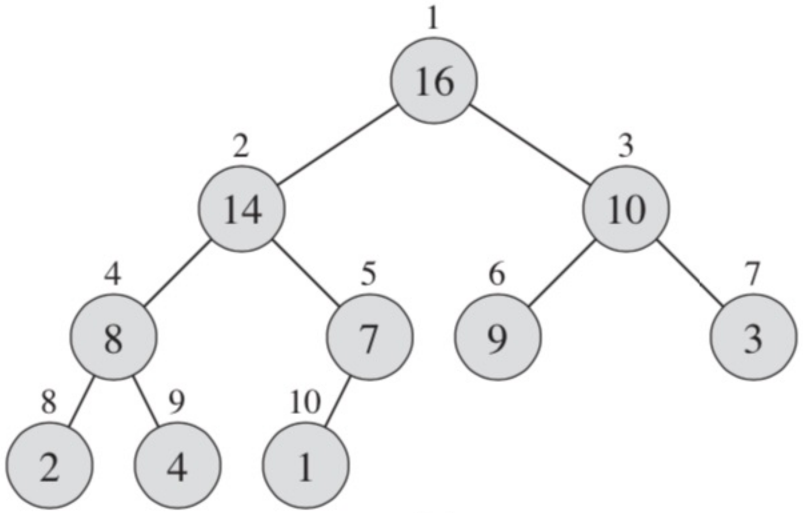
(a)



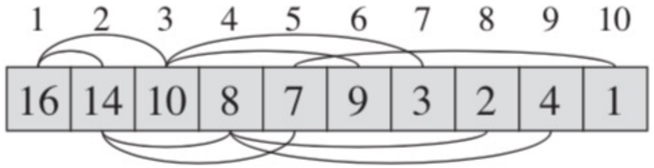
(b)

Data Structure for Heap Tree

- Complete binary tree can be easily stored in an array !



(a)



(b)

PARENT(i)
1 **return** $\lfloor i/2 \rfloor$ $\Theta(1)$

LEFT(i)
1 **return** $2i$ $\Theta(1)$

RIGHT(i)
1 **return** $2i + 1$ $\Theta(1)$

Heap Property

- The max heap property
  - No parent is smaller than its children

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

- The min heap property
  - No parent is greater than its children

$$A[\text{PARENT}(i)] \leq A[i]$$

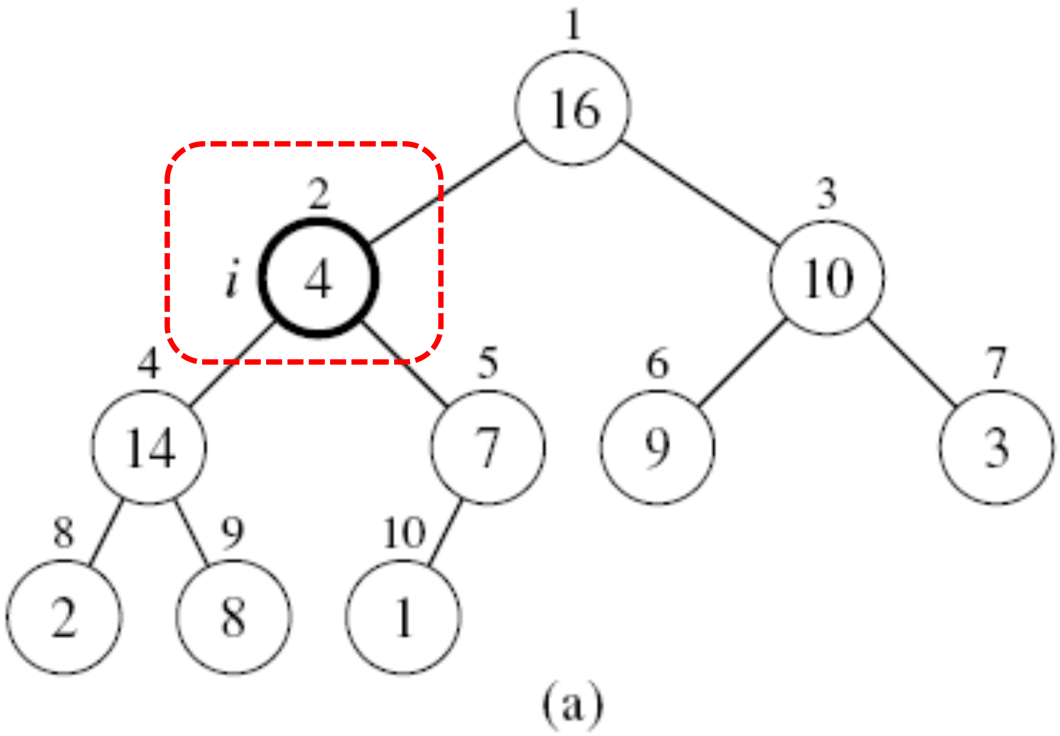
Basic Operations

A basic set of heap operations:

- MAX-HEAPIFY
- BUILD-MAX-HEAP
- HEAP-MAXIMUM
- HEAP-EXTRACT-MAX
- MAX-HEAP-INSERT

Max-Heapify

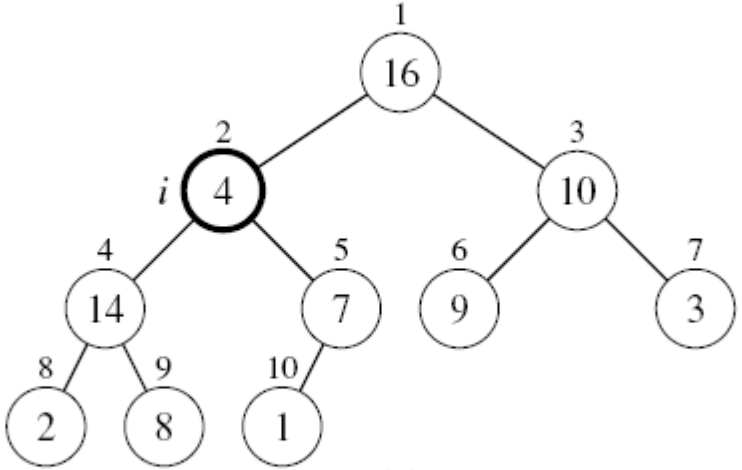
- Assume left and right subtrees of *i* are max-heaps.
- Suppose A[*i*] < max(A[2*i*], A[2*i*+1])
  - Heap property is violated
- After MAX-HEAPIFY, subtree rooted at *i* is a max-heap



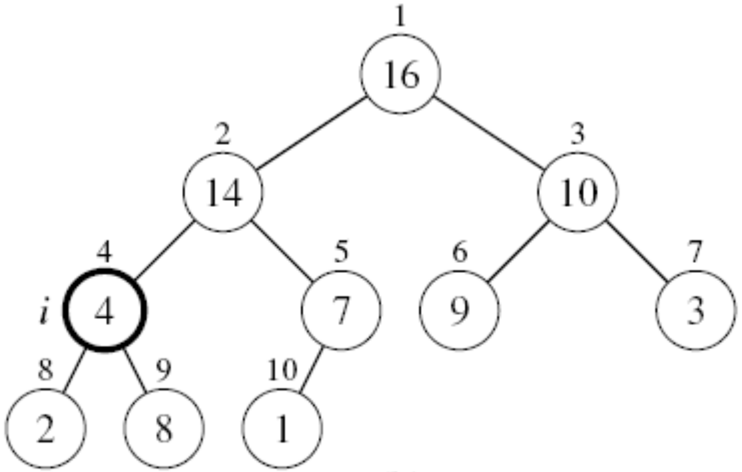
MAX-HEAPIFY(A, *i*, *n*) fixes the heap condition at *i*

How?

Max-Heapify

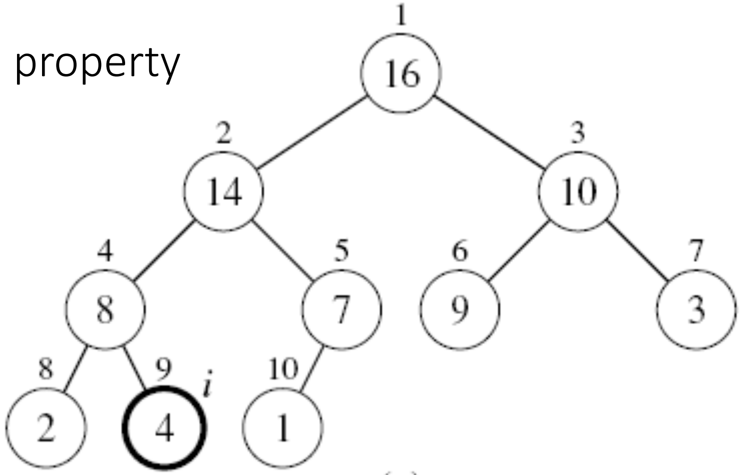


(a)



(b)

- Node 2 violates the max-heap property
- Compare node 2 with its children, and then swap it with the larger of the two children

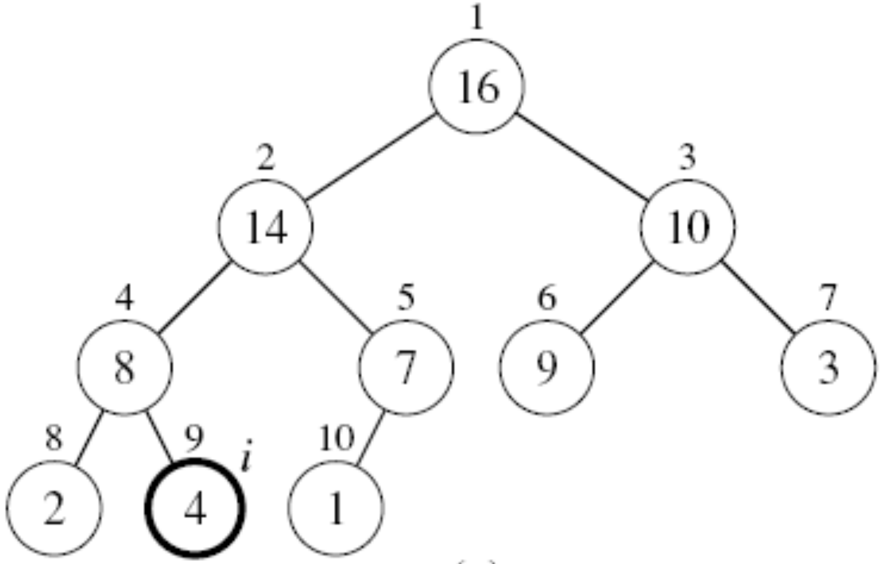


(c)

- Continue down the tree, swapping until the value is properly placed at the root of a subtree

Max-Heapify

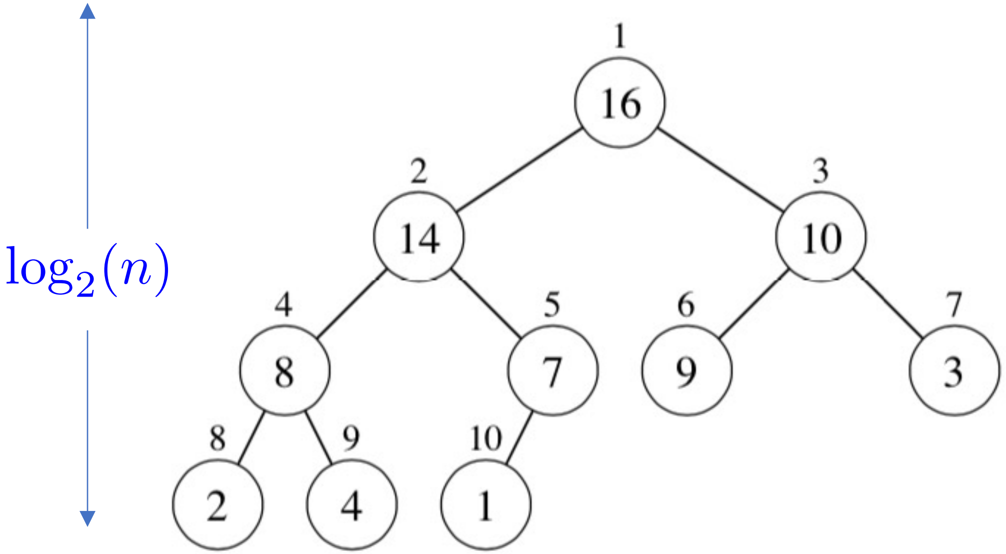
```
MAX-HEAPIFY(A, i)
  l = LEFT(i)
  r = RIGHT(i)
  if l ≤ A.heap-size and A[l] > A[i]
       largest = l
  else largest = i
  if r ≤ A.heap-size and A[r] > A[largest]
       largest = r
  if largest ≠ i
       exchange A[i] with A[largest]
       MAX-HEAPIFY(A, largest)
```



(c)

Max-Heapify Time Complexity

- An n-element heap has height $\log_2(n)$
- We only traverse down a single path
- We (at most) may have to traverse all the way to the leaves
- End conditions:
  - We encounter a subtree that already satisfies the heap property
  - We reach the leaves



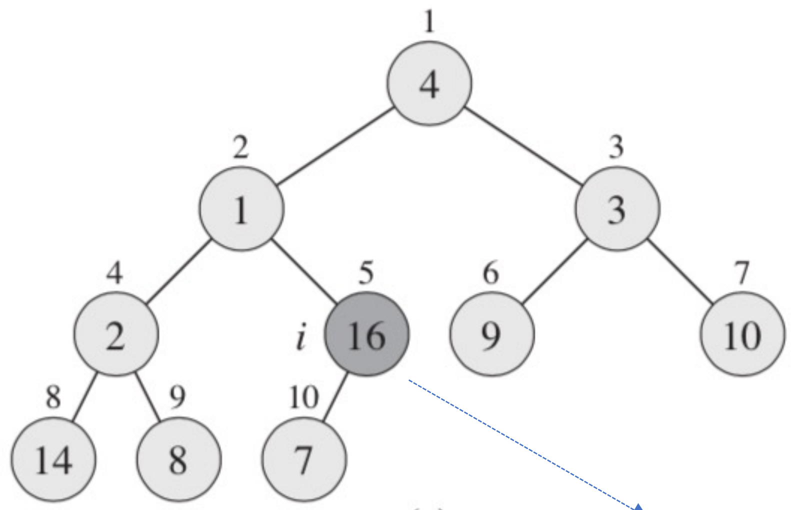
$$T(n) = O(\log n)$$

Build-Max-Heap

- This procedure 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

A										
	4	1	3	2	16	9	10	14	8	7

$n = 10$



(a)

$\lfloor A.length/2 \rfloor$ Parent of last leaf

$\underbrace{\qquad\qquad\qquad}_{A[(\lfloor n/2 \rfloor + 1)\,..\,n]}$

are all leaves of the tree

Build-Max-Heap

BUILD-MAX-HEAP(A)

$A.heap\text{-}size = A.length$

for $i = \lfloor A.length/2 \rfloor$ **downto** 1

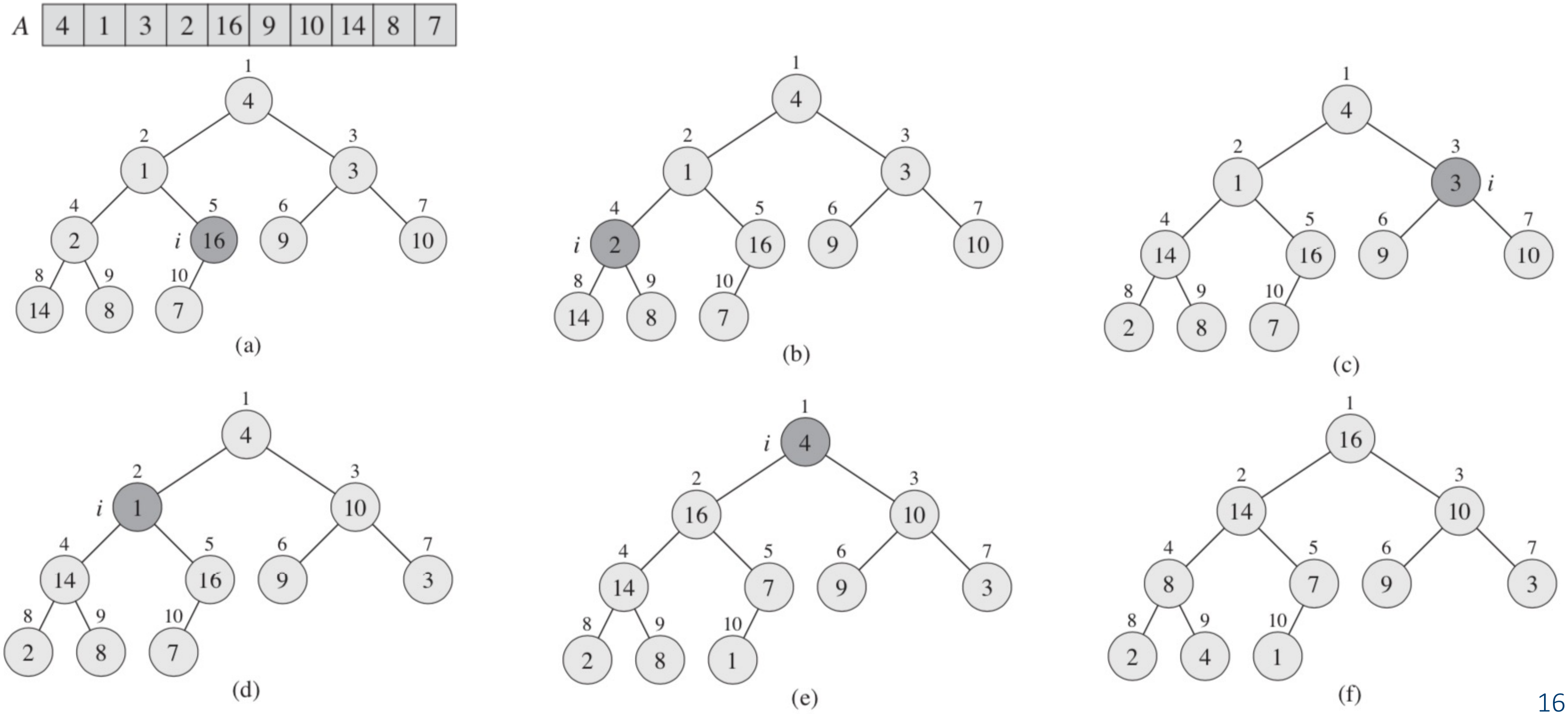
$\qquad \underbrace{\text{MAX-HEAPIFY}(A, i)}_{O(\log n)}$

<u>Time Complexity</u>

$$O(n \log n)$$

However, It is not tight enough !

Build-Max-Heap



Build-Max-Heap Complexity

Simple Bound: $O(n \log n)$

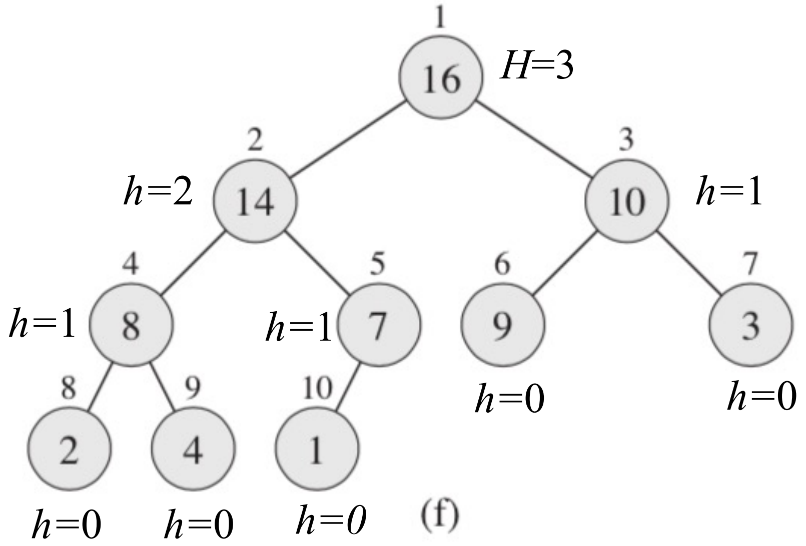
Tighter analysis:

Two facts:

(1) n-element heap has height $\lfloor \lg n \rfloor$

(2) there are at most $\lceil n/2^{h+1} \rceil$ nodes of height h in any n-element heap

$H = \lfloor \log 10 \rfloor = 3$



$$\#\,(h{=}0) : 5 \quad \le \left\lceil \frac{10}{2^{0+1}} \right\rceil = 5$$

$$\#\,(h{=}1) : 3 \quad \le \left\lceil \frac{10}{2^{1+1}} \right\rceil = 3$$

$$\#\,(h{=}2) : 1 \quad \le \left\lceil \frac{10}{2^{2+1}} \right\rceil = 2$$

$$\#\,(h{=}3) : 1 \quad \le \left\lceil \frac{10}{2^{3+1}} \right\rceil = 1$$

Build-Max-Heap Complexity

Simple Bound: $O(n \log n)$

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

for $|x| < 1$.

Tighter analysis:

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

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$$

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

Thus, the running time of BUILD-MAX-HEAP is $O(n)$

Heap-Maximum

- Returns the maximum element of heap
  - it’s the root !

$\textsc{Heap-Maximum}(A)$

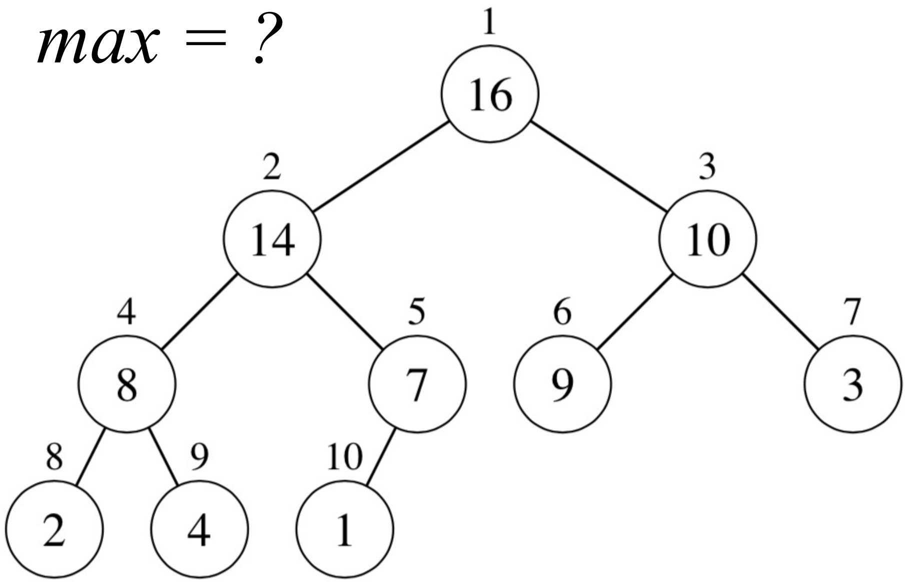
1 **return** $A[1]$

$$\Theta(1)$$

Heap-Extract-Max

- Returns the maximum element of heap and removes it

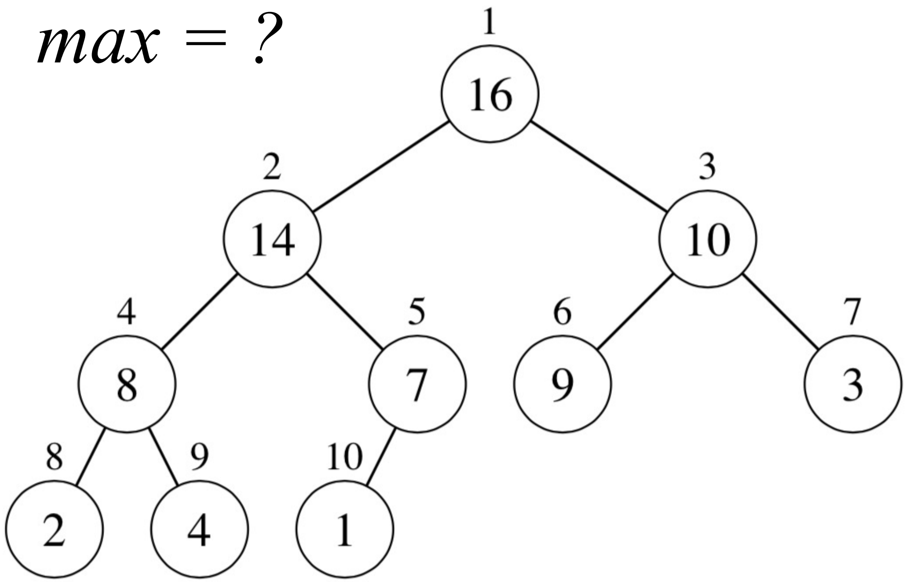
$max = ?$



Heap-Extract-Max

- Returns the maximum element of heap and removes it

$max = ?$

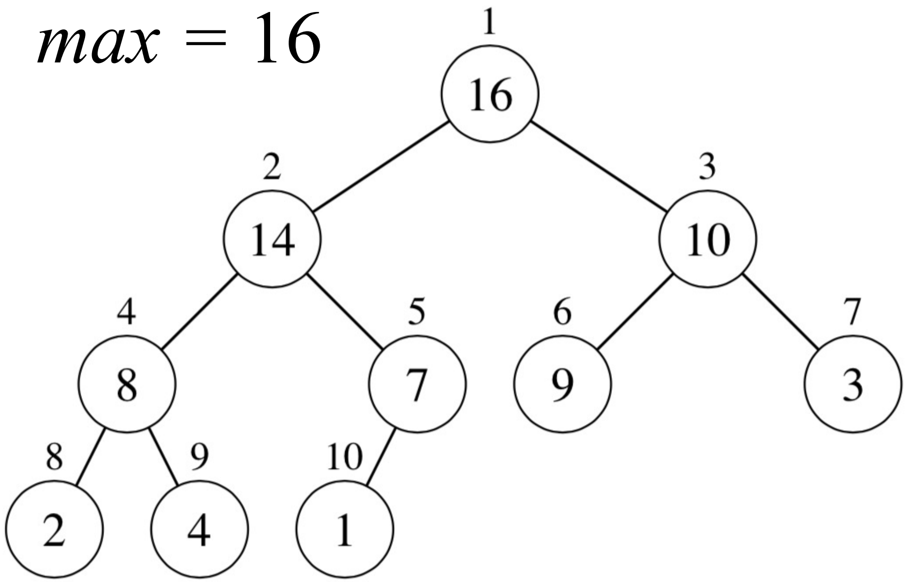


1) Take 16 out of node 1

Heap-Extract-Max

- Returns the maximum element of heap and removes it

$max = 16$

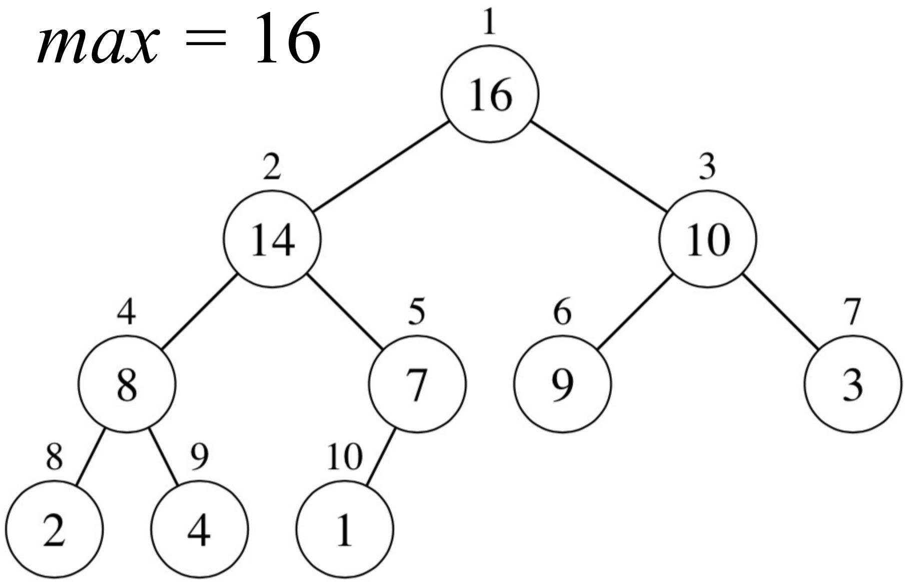


1) Take 16 out of node 1

Heap-Extract-Max

- Returns the maximum element of heap and removes it

$max = 16$

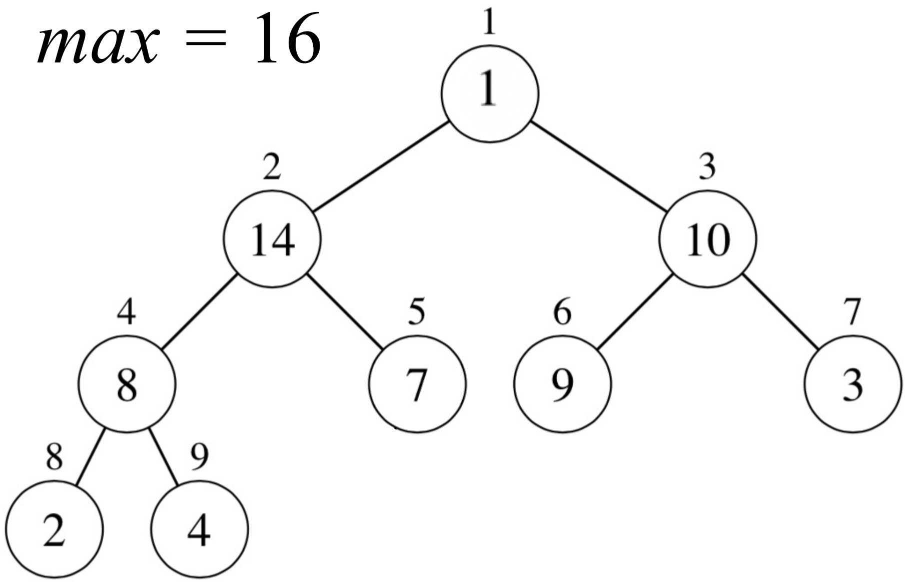


1) Take 16 out of node 1
2) Move 1 from node 10 to node 1 and erase node

Heap-Extract-Max

- Returns the maximum element of heap and removes it

$max = 16$

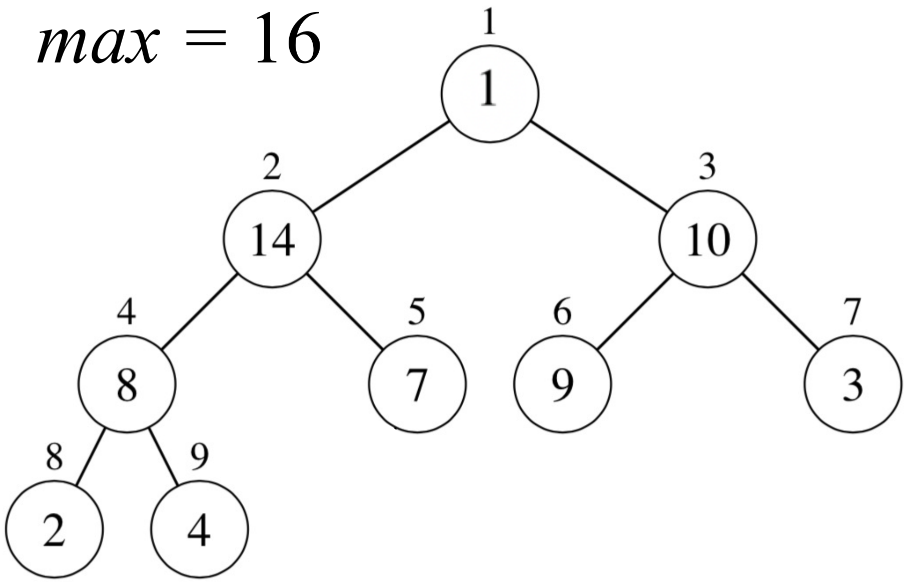


1) Take 16 out of node 1
2) Move 1 from node 10 to node 1 and erase node

Heap-Extract-Max

- Returns the maximum element of heap and removes it

$max = 16$

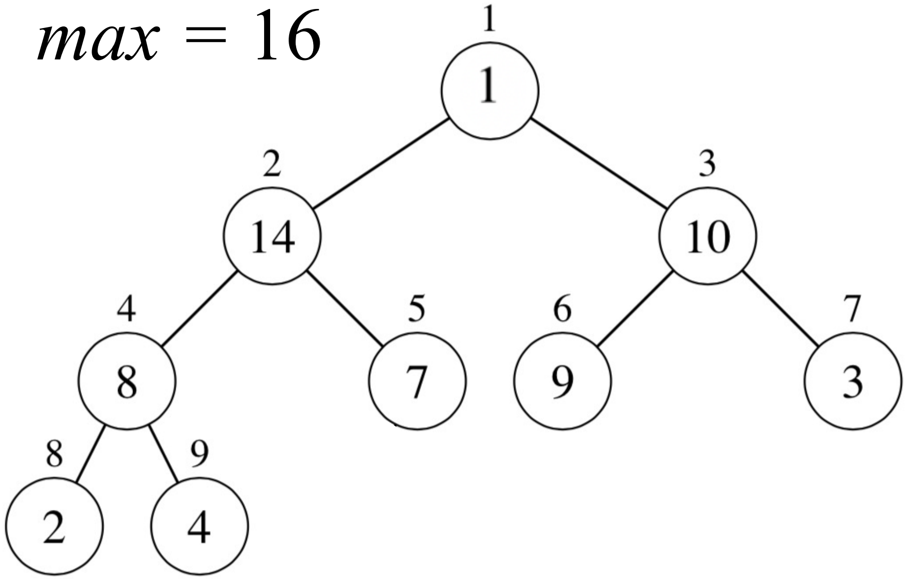


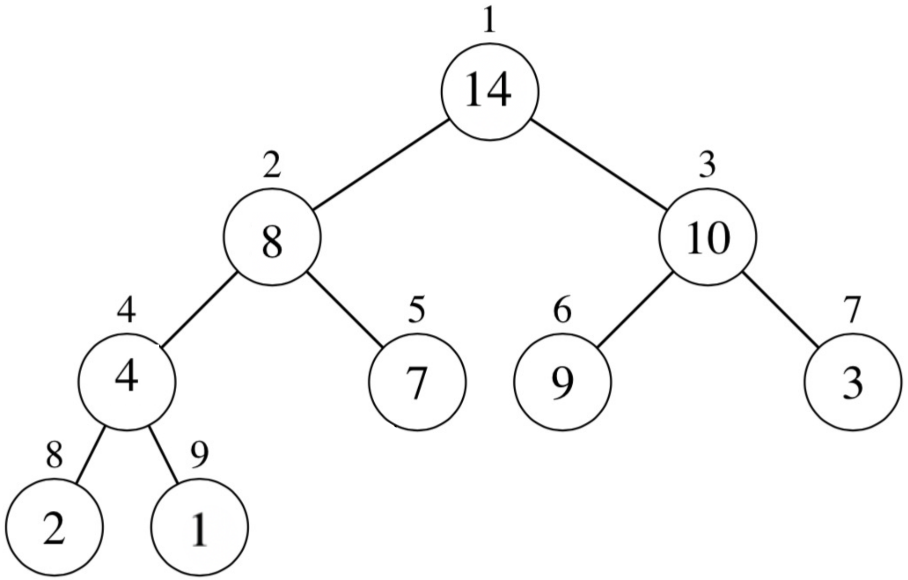
1) Take 16 out of node 1
2) Move 1 from node 10 to node 1 and erase node
3) MAX-HEAPIFY from the root to preserve max-heap property

Heap-Extract-Max

- Returns the maximum element of heap and removes it

$max = 16$





Heap-Extract-Max

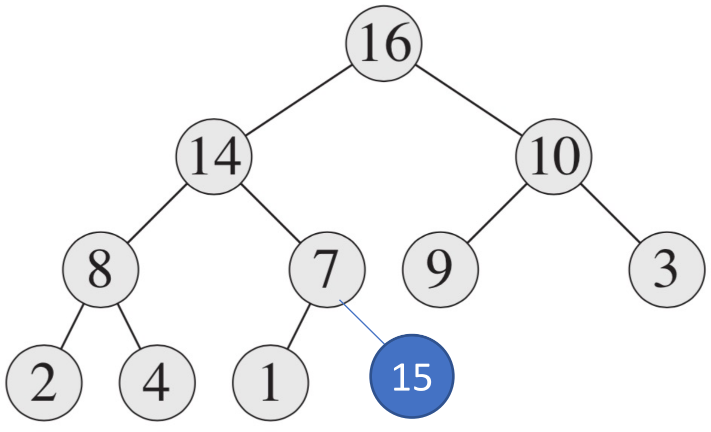
- Returns the maximum element of heap and removes it

```
HEAP-EXTRACT-MAX(A)
if A.heap-size < 1
        error "heap underflow"
max = A[1]
A[1] = A[A.heap-size]
A.heap-size = A.heap-size - 1
MAX-HEAPIFY(A, 1)
return max
```

$O(\log n)$

Max-Heap-Insert

- We insert new element at the end, then move it up into its correct position



Max-Heap-Insert

- We insert new element at the end, then move it up into its correct position

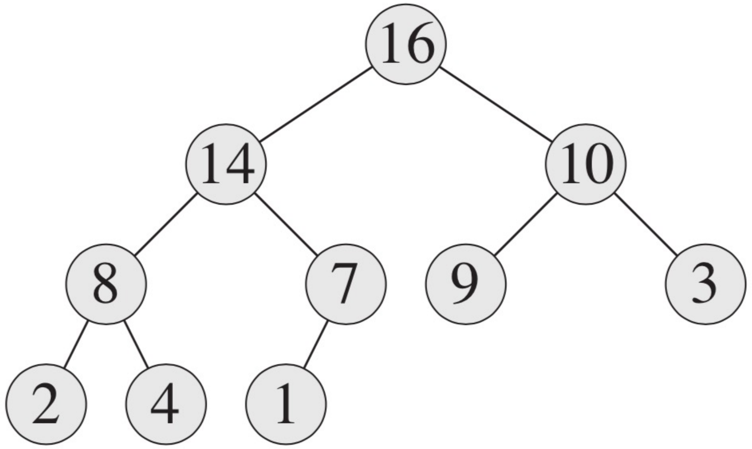
MAX-HEAP-INSERT(A, x)

$A.heap\text{-}size = A.heap\text{-}size + 1$
$A[A.heap\text{-}size] = x$ $\quad O(1)$
$i = A.heap\text{-}size$
while $A[i] > A[Parent(i)]$ and $i > 1$ $\quad O(\log n)$
  Exchange $A[i]$ with $A[Parent(i)]$ $\quad O(1)$
  $i = Parent(i)$

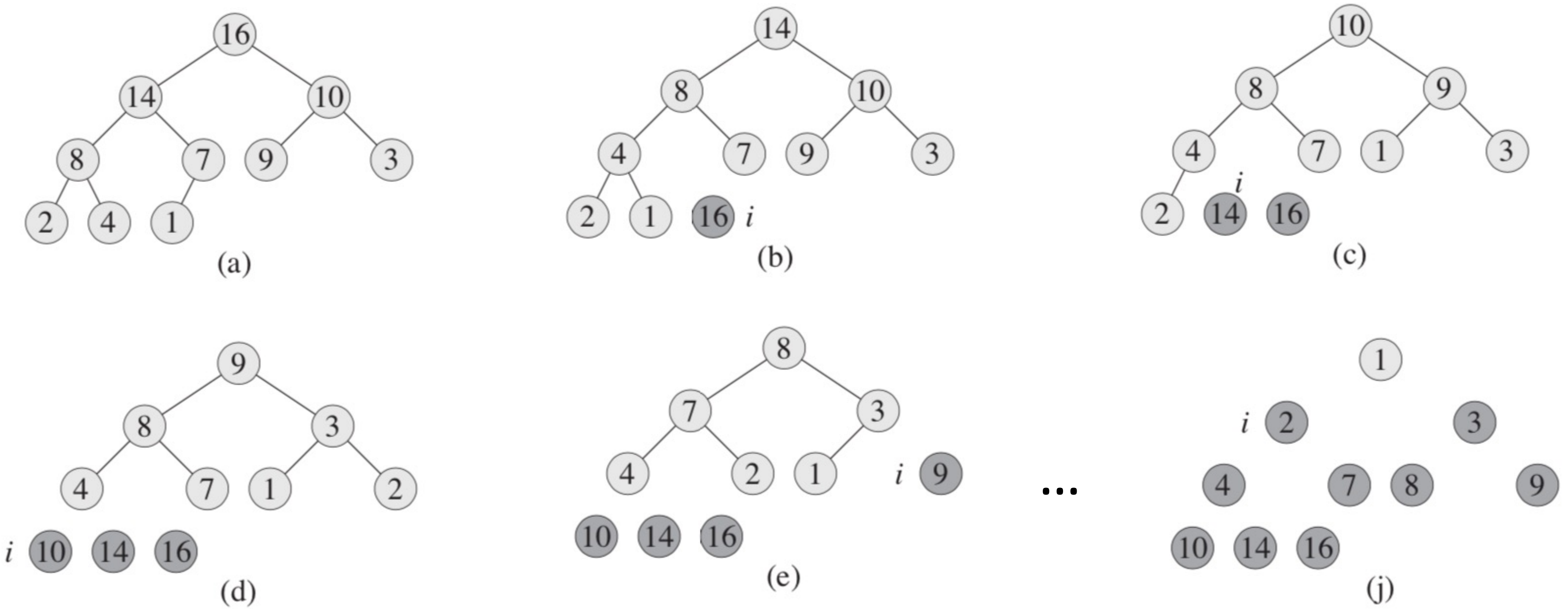
$$O(\log n)$$

Heap Applications

- Heapsort
  - The algorithm places the maximum element into the correct place in the array by swapping it with the element in the last position in the array.



Heap Sort



Heap Applications

- Heapsort

```
HEAPSORT(A)
BUILD-MAX-HEAP(A)
for i = A.length downto 2
    exchange A[1] with A[i]
    A.heap-size = A.heap-size - 1
    MAX-HEAPIFY(A, 1)
```

$O(n)$

$\underbrace{\text{MAX-HEAPIFY}(A, 1)}_{O(\log n)}$

Time Complexity

$$O(n \log n)$$

Heap Applications

- Priority queue is an Abstract Data Type (ADT) :
  - Maintains a dynamic set S of elements
  - Each set element has a ***key***—an associated value

- Priority support the following operations:
  - $\textsc{Insert}(S, x)$: inserts element x into set S
  - $\textsc{Maximum}(S)$: returns element of S with largest key
  - $\textsc{Extract-Max}(S)$: removes and returns element of S with largest key.

  …

Priority Queue Implementations

	Unsorted Array	Sorted Array	Heap
$\text{INSERT}(S, x)$	$\Theta(1)$ Add into the end	$\Theta(n)$ Shift	$\Theta(\log n)$ MAX-HEAP-INSERT
$\text{MAXIMUM}(S)$	$\Theta(n)$ Linear Search	$\Theta(1)$ Last Element	$\Theta(1)$ HEAP-MAXIMUM
$\text{EXTRACT-MAX}(S)$	$\Theta(n)$ Search + shift	$\Theta(1)$ Shorten	$\Theta(\log n)$ HEAP-EXTRACT-MAX

Wrap-up

- We learned
  - One interesting data structure called “Heap”
  - We study how heap can be used for:
    - Sort Problem
    - Priority Queue