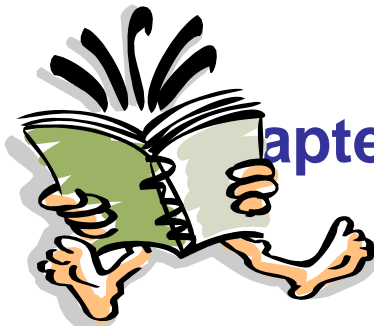


Analysis of Algorithms

CS 477/677

Heapsort

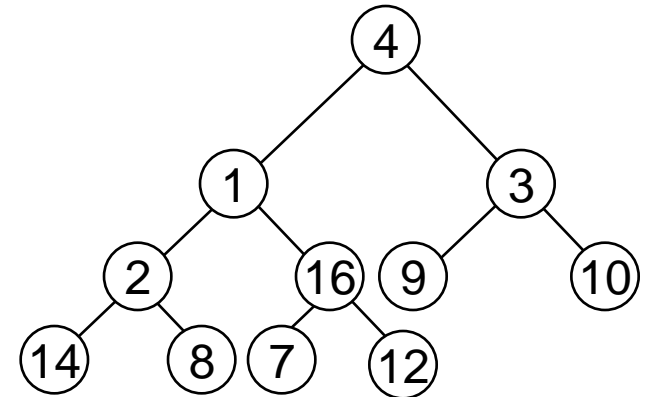
Instructor: George Bebis



Chapter 6, Appendix B.5)

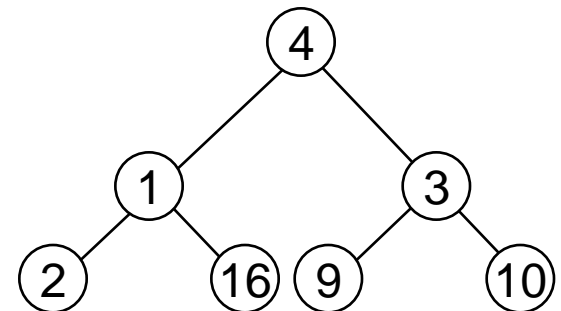
Special Types of Trees

- *Def:* **Full binary tree** = a binary tree in which each node is either a leaf or has degree exactly 2.



Full binary tree

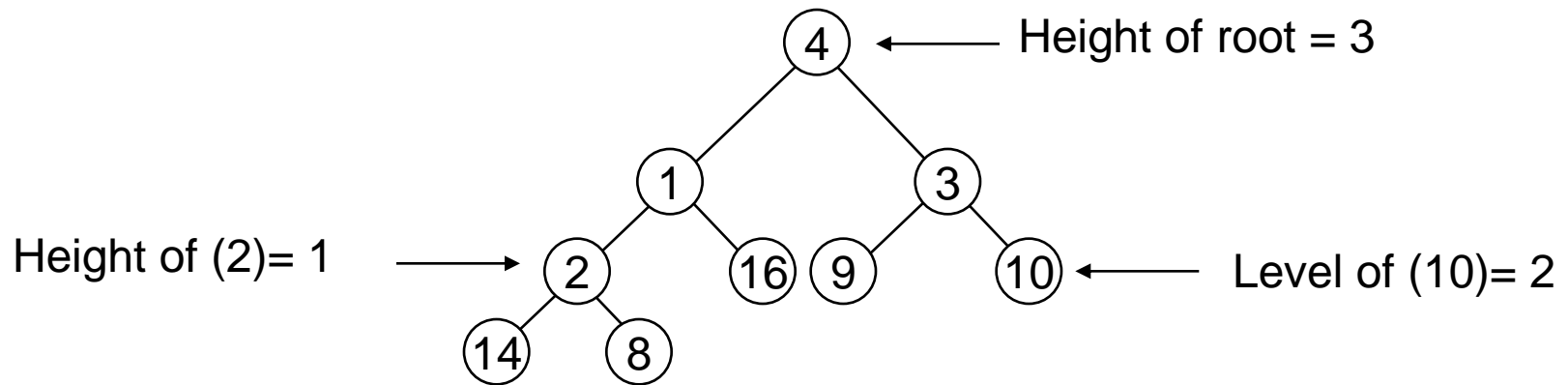
- *Def:* **Complete binary tree** = a binary tree in which all leaves are on the same level and all internal nodes have degree 2.



Complete binary tree

Definitions

- **Height of a node** = the number of edges on the longest simple path from the node down to a leaf
- **Level of a node** = the length of a path from the root to the node
- **Height of tree** = height of root node

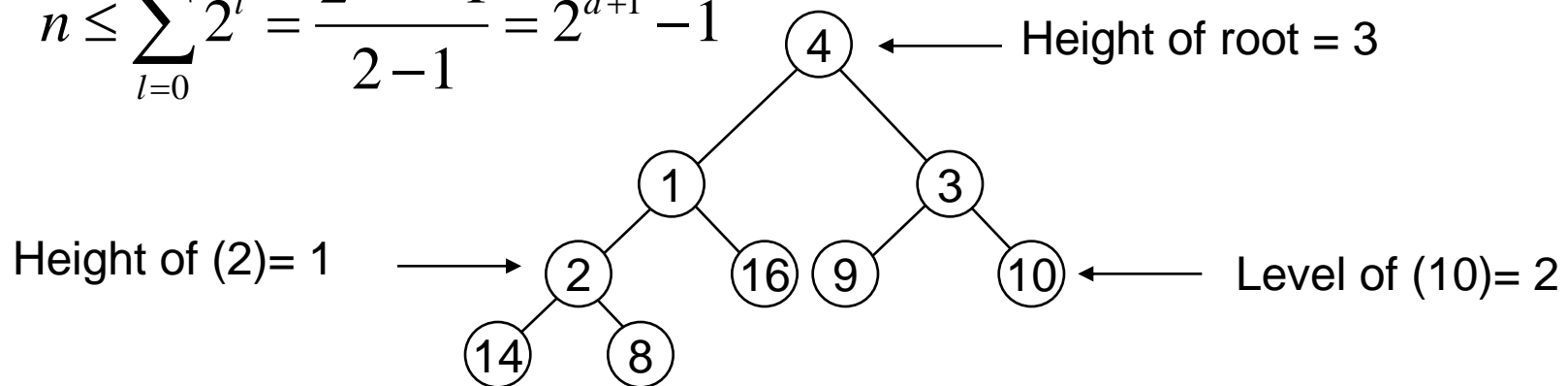


Useful Properties

- There are **at most** 2^l nodes at level (or depth) l of a binary tree
- A binary tree with height d has **at most** $2^{d+1} - 1$ nodes
- A binary tree with n nodes has height **at least** $\lceil \lg n \rceil$

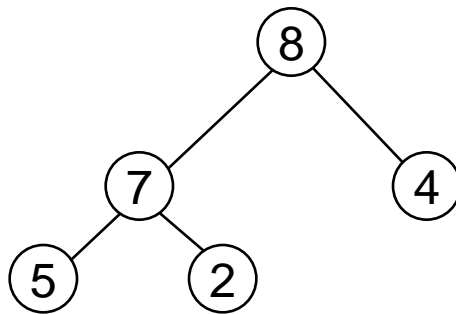
(see Ex 6.1-2, page 129)

$$n \leq \sum_{l=0}^d 2^l = \frac{2^{d+1} - 1}{2 - 1} = 2^{d+1} - 1$$



The Heap Data Structure

- *Def:* A **heap** is a nearly complete binary tree with the following two properties:
 - **Structural property:** all levels are full, except possibly the last one, which is filled from left to right
 - **Order (heap) property:** for any node x
$$\text{Parent}(x) \geq x$$



Heap

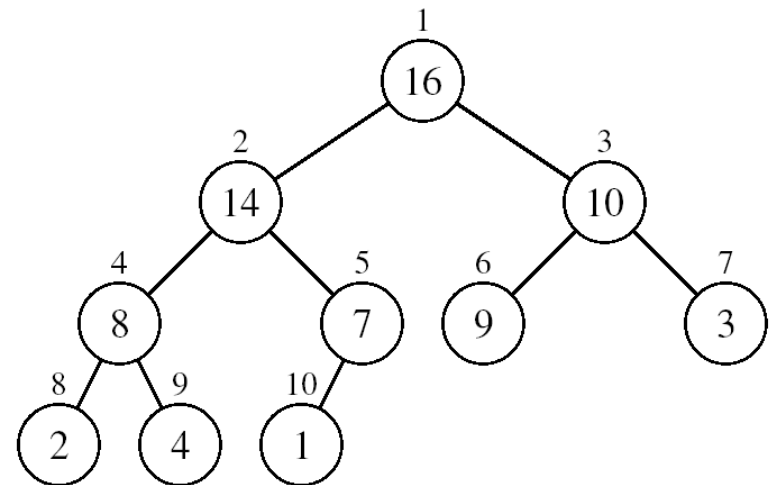
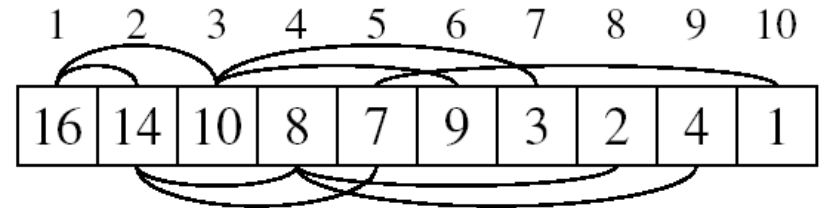
From the heap property, it follows that:

“The root is the maximum element of the heap!”

A heap is a binary tree that is filled in order

Array Representation of Heaps

- A heap can be stored as an array A .
 - Root of tree is $A[1]$
 - Left child of $A[i] = A[2i]$
 - Right child of $A[i] = A[2i + 1]$
 - Parent of $A[i] = A[\lfloor i/2 \rfloor]$
 - $\text{Heapsize}[A] \leq \text{length}[A]$
- The elements in the subarray $A[(\lfloor n/2 \rfloor + 1) .. n]$ are leaves



Heap Types

- **Max-heaps** (largest element at root), have the *max-heap property*:

- for all nodes i , excluding the root:

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

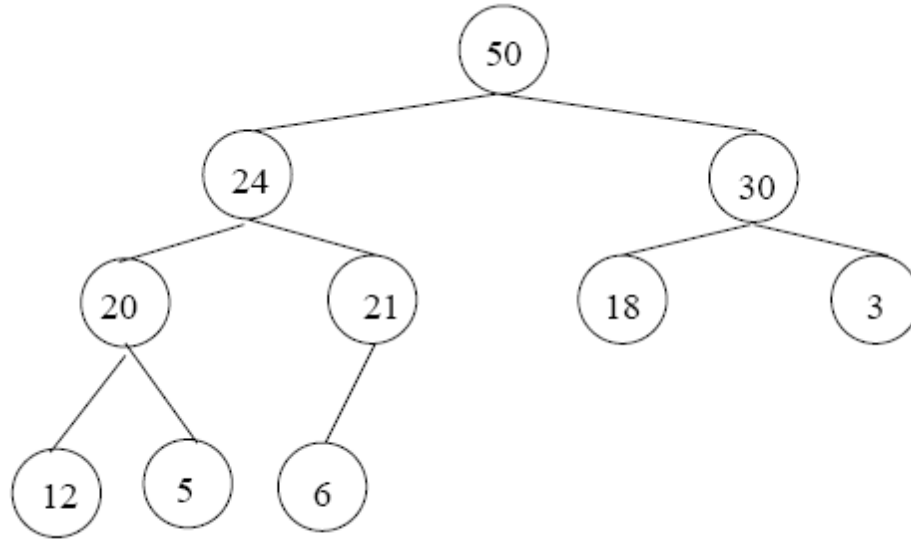
- **Min-heaps** (smallest element at root), have the *min-heap property*:

- for all nodes i , excluding the root:

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

Adding/Deleting Nodes

- New nodes are always inserted at the bottom level (left to right)
- Nodes are removed from the bottom level (right to left)

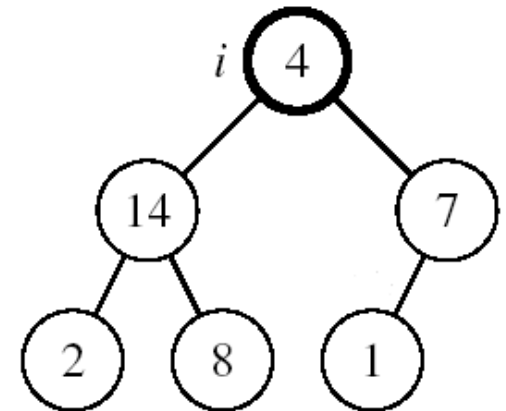


Operations on Heaps

- Maintain/Restore the max-heap property
 - MAX-HEAPIFY
- Create a max-heap from an unordered array
 - BUILD-MAX-HEAP
- Sort an array in place
 - HEAPSORT
- Priority queues

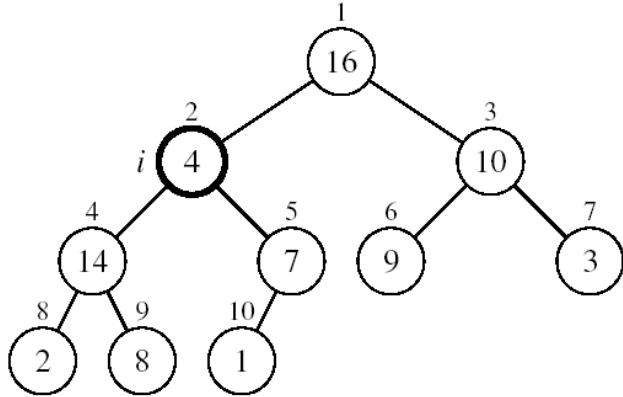
Maintaining the Heap Property

- Suppose a node is smaller than a child
 - Left and Right subtrees of i are max-heaps
- To eliminate the violation:
 - Exchange with larger child
 - Move down the tree
 - Continue until node is not smaller than children



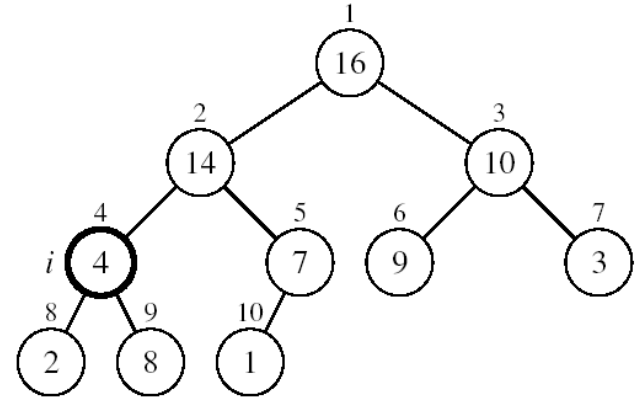
Example

MAX-HEAPIFY(A, 2, 10)



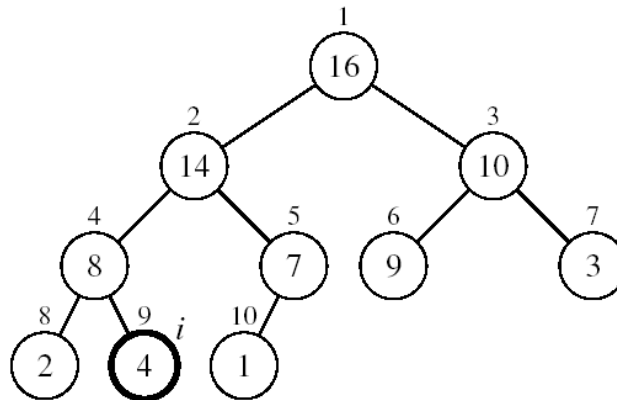
A[2] violates the heap property

$A[2] \leftrightarrow A[4]$



A[4] violates the heap property

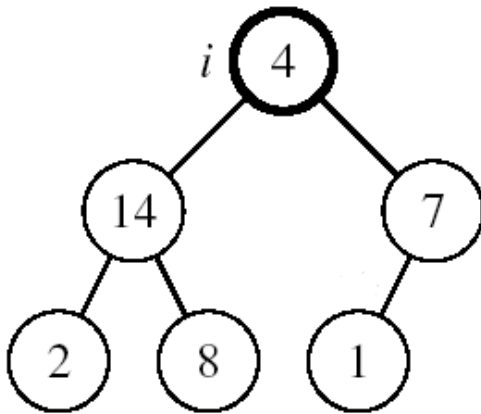
$A[4] \leftrightarrow A[9]$



Heap property restored

Maintaining the Heap Property

- Assumptions:
 - Left and Right subtrees of i are max-heaps
 - $A[i]$ may be smaller than its children



Alg: MAX-HEAPIFY(A, i, n)

1. $l \leftarrow \text{LEFT}(i)$
2. $r \leftarrow \text{RIGHT}(i)$
3. **if** $l \leq n$ and $A[l] > A[i]$
4. **then** $\text{largest} \leftarrow l$
5. **else** $\text{largest} \leftarrow i$
6. **if** $r \leq n$ and $A[r] > A[\text{largest}]$
7. **then** $\text{largest} \leftarrow r$
8. **if** $\text{largest} \neq i$
9. **then** exchange $A[i] \leftrightarrow A[\text{largest}]$
10. MAX-HEAPIFY($A, \text{largest}, n$)

MAX-HEAPIFY Running Time

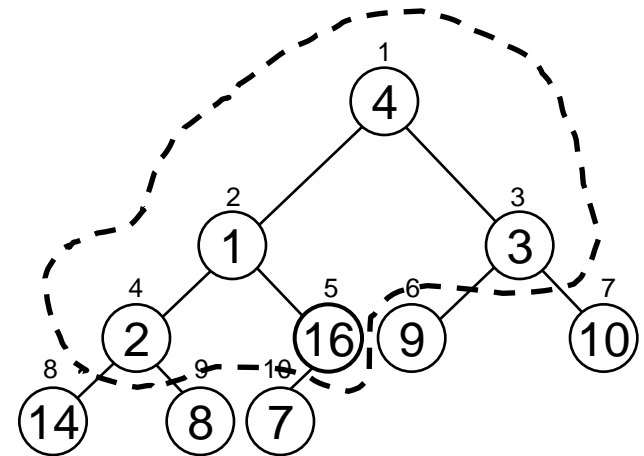
- Intuitively:
 - It traces a path from the root to a leaf (longest path length: h)
 - At each level, it makes exactly 2 comparisons
 - Total number of comparisons is $\leq 2h$
 - Running time is $O(h)$ or $O(\lg n)$
- Running time of MAX-HEAPIFY is $O(\lg n)$
- Can be written in terms of the height of the heap, as being $O(h)$
 - Since the height of the heap is $\lfloor \lg n \rfloor$

Building a Heap

- Convert an array $A[1 \dots n]$ into a max-heap ($n = \text{length}[A]$)
- The elements in the subarray $A[(\lfloor n/2 \rfloor + 1) \dots n]$ are leaves
- Apply MAX-HEAPIFY on elements between 1 and $\lfloor n/2 \rfloor$

Alg: BUILD-MAX-HEAP(A)

1. $n = \text{length}[A]$
2. **for** $i \leftarrow \lfloor n/2 \rfloor$ **downto** 1
3. **do** MAX-HEAPIFY(A, i, n)



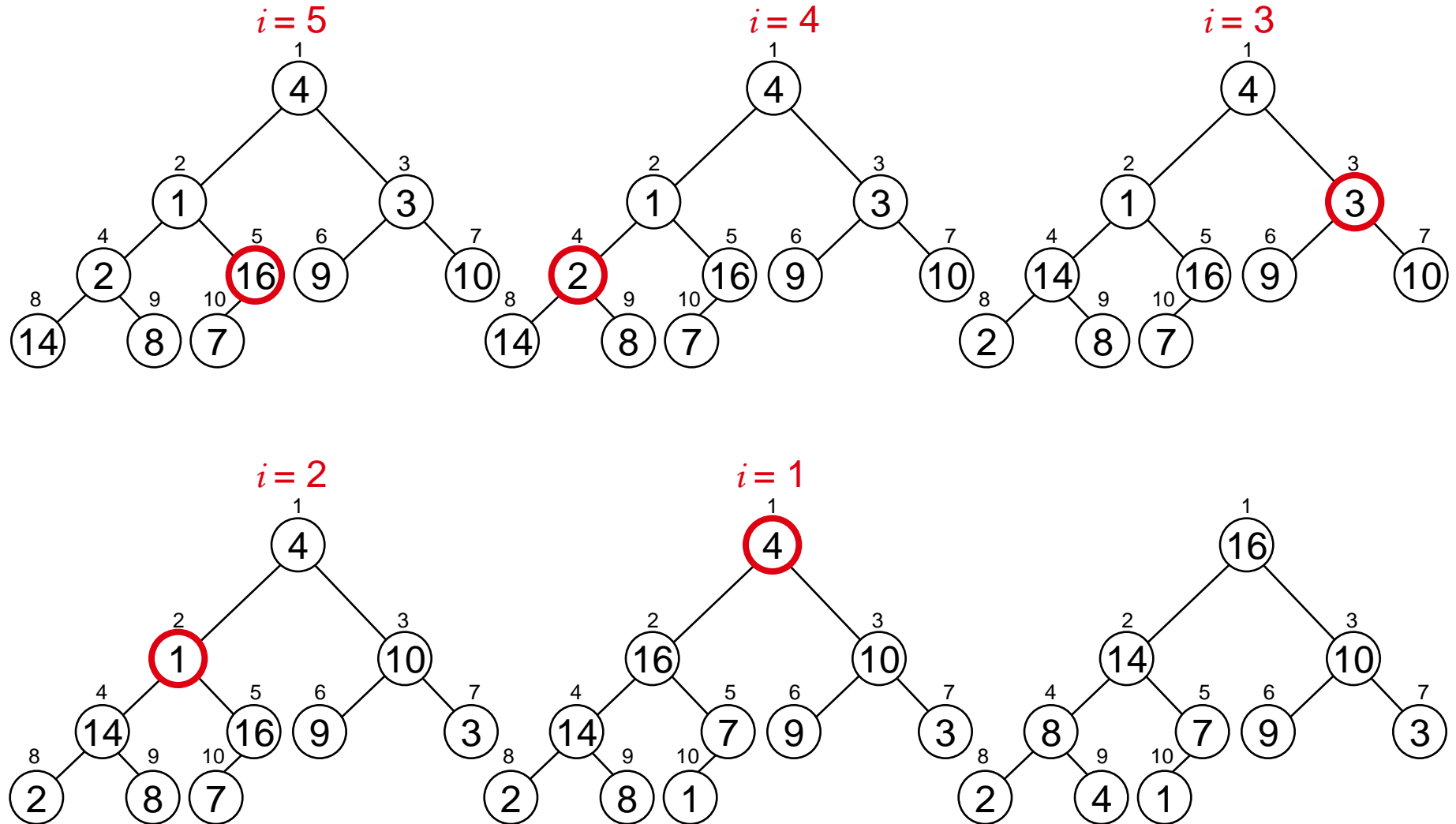
A:

4	1	3	2	16	9	10	14	8	7
---	---	---	---	----	---	----	----	---	---

Example:

A

4	1	3	2	16	9	10	14	8	7
---	---	---	---	----	---	----	----	---	---



Running Time of BUILD MAX HEAP

Alg: BUILD-MAX-HEAP(A)

1. $n = \text{length}[A]$
 2. **for** $i \leftarrow \lfloor n/2 \rfloor$ **downto** 1
 3. **do** MAX-HEAPIFY(A, i, n)
- $O(\lg n)$ } $O(n)$

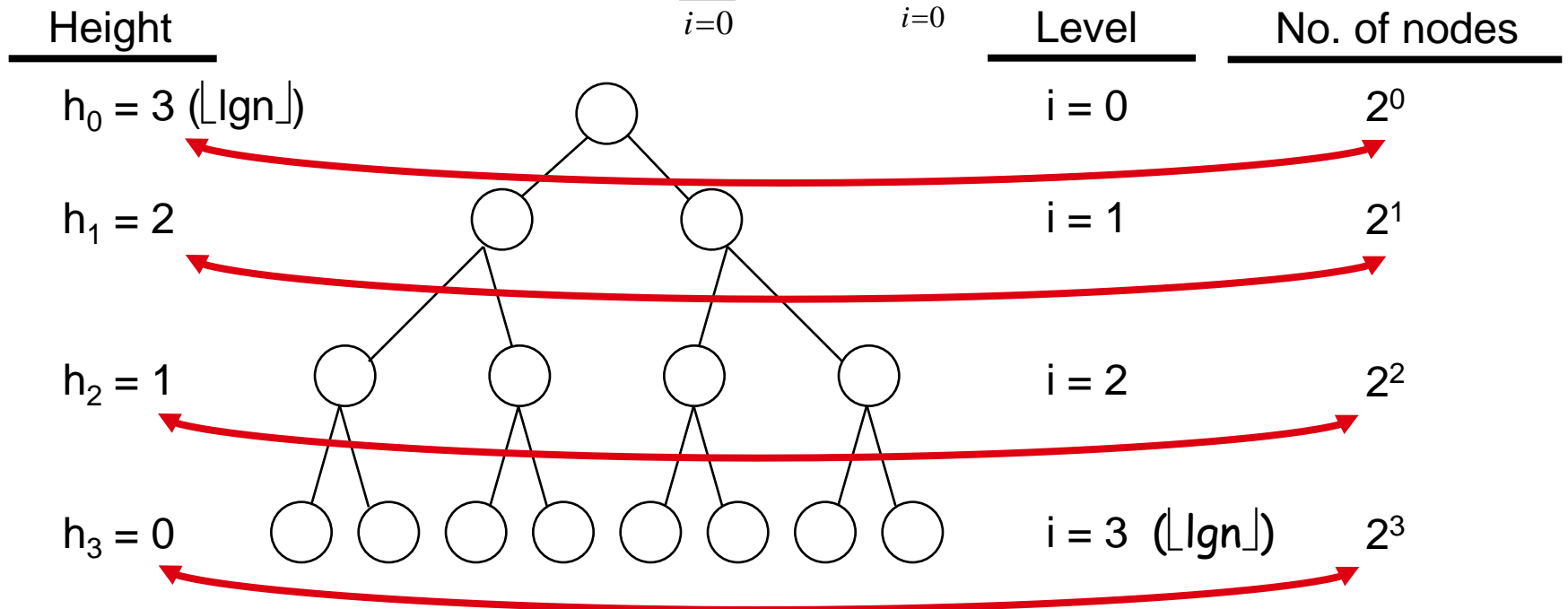
\Rightarrow Running time: $O(n \lg n)$

- This is not an asymptotically tight upper bound

Running Time of BUILD MAX HEAP

- HEAPIFY takes $O(h) \Rightarrow$ the cost of HEAPIFY on a node i is proportional to the height of the node i in the tree

$$\Rightarrow T(n) = \sum_{i=0}^h n_i h_i = \sum_{i=0}^h 2^i (h-i) = O(n)$$



$h_i = h - i$ height of the heap rooted at level i
 $n_i = 2^i$ number of nodes at level i

Running Time of BUILD MAX HEAP

$T(n) = \sum_{i=0}^h n_i h_i$	Cost of HEAPIFY at level i * number of nodes at that level
$= \sum_{i=0}^h 2^i (h - i)$	Replace the values of n_i and h_i computed before
$= \sum_{i=0}^h \frac{h - i}{2^{h-i}} 2^h$	Multiply by 2^h both at the nominator and denominator and write 2^i as $\frac{1}{2^{-i}}$
$= 2^h \sum_{k=0}^h \frac{k}{2^k}$	Change variables: $k = h - i$
$\leq n \sum_{k=0}^{\infty} \frac{k}{2^k}$	The sum above is smaller than the sum of all elements to ∞ and $h = \lg n$
$= O(n)$	The sum above is smaller than 2

Running time of BUILD-MAX-HEAP: $T(n) = O(n)$

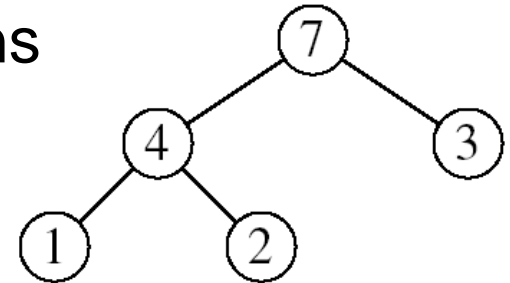
Heapsort

- Goal:

- Sort an array using heap representations

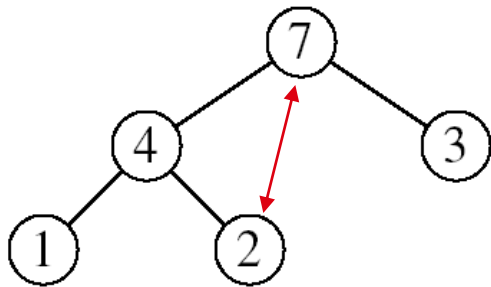
- Idea:

- Build a **max-heap** from the array
- Swap the root (the maximum element) with the last element in the array
- “Discard” this last node by decreasing the heap size
- Call MAX-HEAPIFY on the new root
- Repeat this process until only one node remains

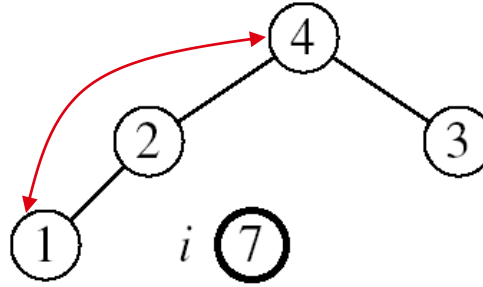


Example:

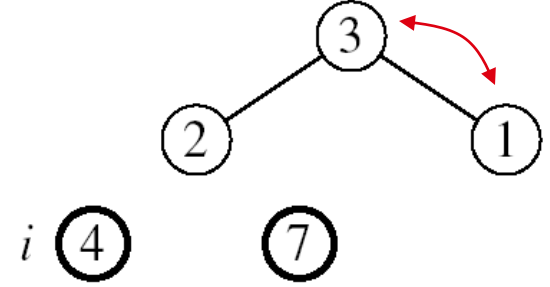
$A=[7, 4, 3, 1, 2]$



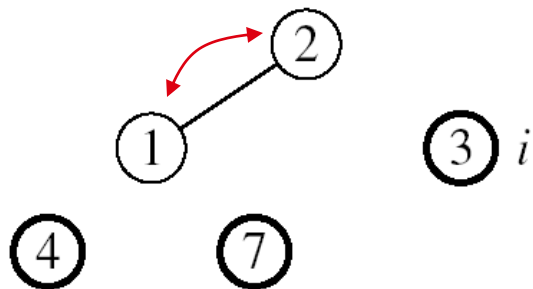
MAX-HEAPIFY(A, 1, 4)



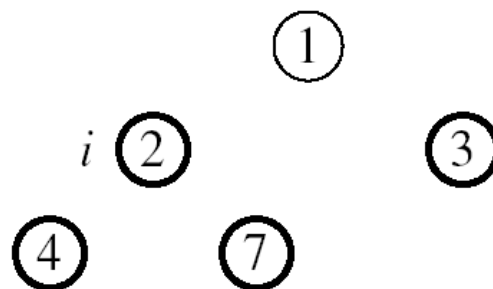
MAX-HEAPIFY(A, 1, 3)



MAX-HEAPIFY(A, 1, 2)



MAX-HEAPIFY(A, 1, 1)



A

1	2	3	4	7
---	---	---	---	---

Alg: HEAPSORT(*A*)

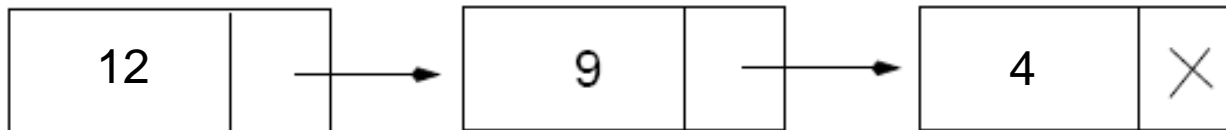
1. BUILD-MAX-HEAP(*A*) $O(n)$
 2. **for** $i \leftarrow \text{length}[A]$ **downto** 2
 3. **do** exchange $A[1] \leftrightarrow A[i]$
 4. MAX-HEAPIFY(*A*, 1, $i - 1$) $O(\lg n)$
- } $n-1$ times

- Running time: $O(n \lg n)$ --- Can be shown to be $\Theta(n \lg n)$

Priority Queues

Properties

- Each element is associated with a value (priority)
- The key with the highest (or lowest) priority is extracted first



Operations on Priority Queues

- Max-priority queues support the following operations:
 - $\text{INSERT}(S, x)$: inserts element x into set S
 - $\text{EXTRACT-MAX}(S)$: removes and returns element of S with largest key
 - $\text{MAXIMUM}(S)$: returns element of S with largest key
 - $\text{INCREASE-KEY}(S, x, k)$: increases value of element x 's key to k (Assume $k \geq x$'s current key value)

HEAP-MAXIMUM

Goal:

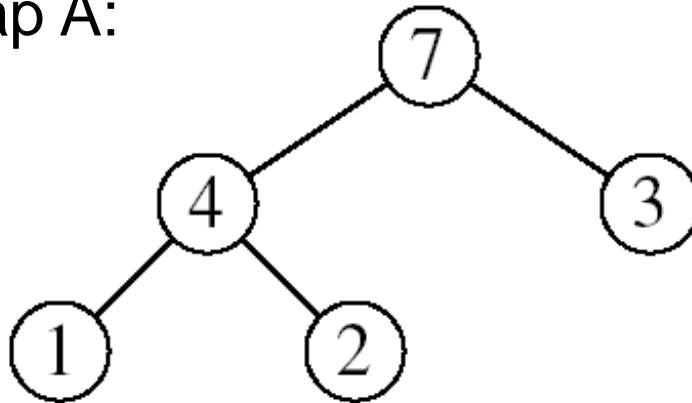
- Return the largest element of the heap

Alg: HEAP-MAXIMUM(A)

Running time: $O(1)$

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

Heap A:



Heap-Maximum(A) returns 7

HEAP-EXTRACT-MAX

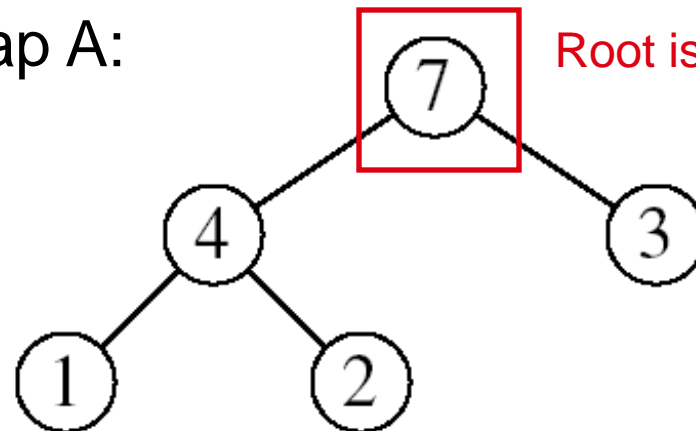
Goal:

- Extract the largest element of the heap (i.e., return the max value and also remove that element from the heap)

Idea:

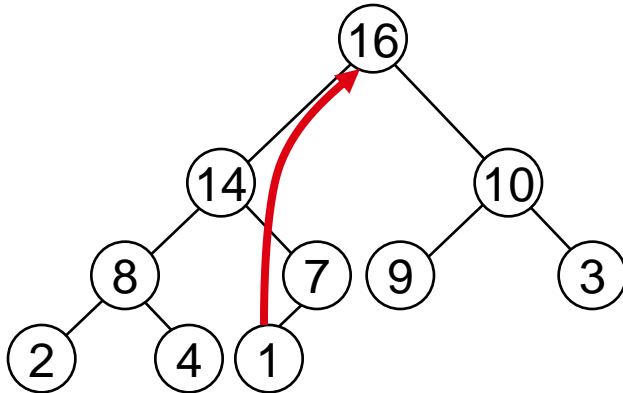
- Exchange the root element with the last
- Decrease the size of the heap by 1 element
- Call MAX-HEAPIFY on the new root, on a heap of size $n-1$

Heap A:

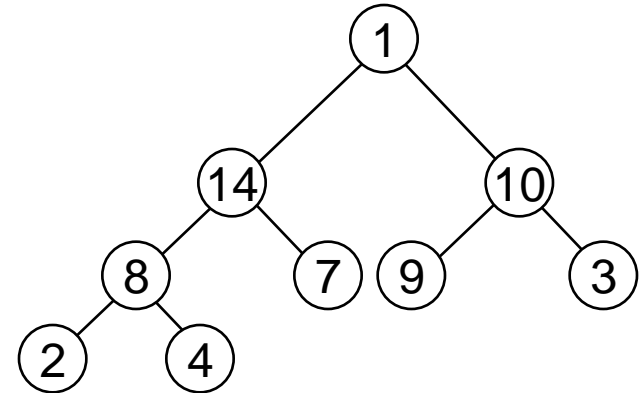


Root is the largest element

Example: HEAP-EXTRACT-MAX

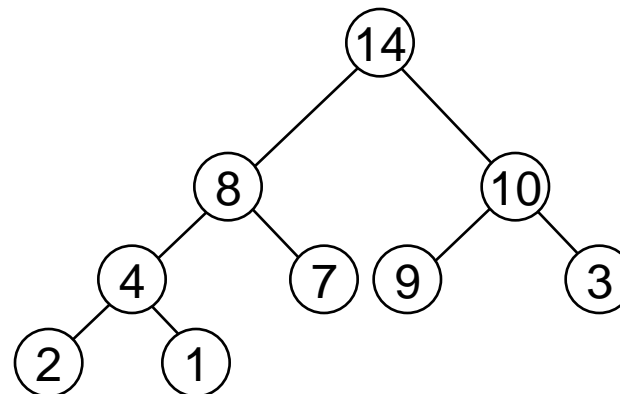


max = 16



Heap size decreased with 1

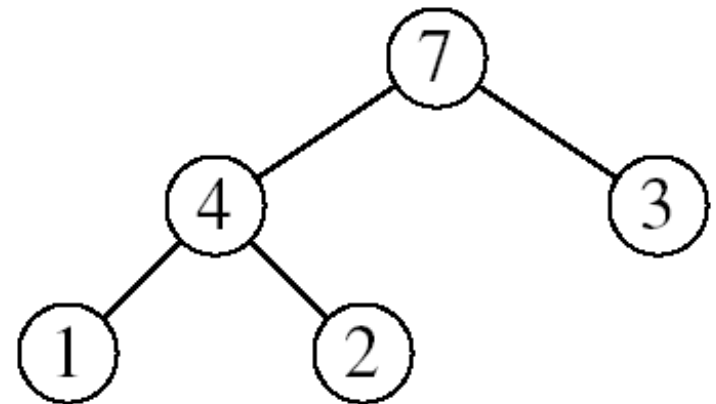
Call MAX-HEAPIFY(A, 1, n-1)



HEAP-EXTRACT-MAX

Alg: HEAP-EXTRACT-MAX(A, n)

1. **if** $n < 1$
2. **then error** “heap underflow”
3. $\text{max} \leftarrow A[1]$
4. $A[1] \leftarrow A[n]$
5. MAX-HEAPIFY($A, 1, n-1$)
6. **return** max

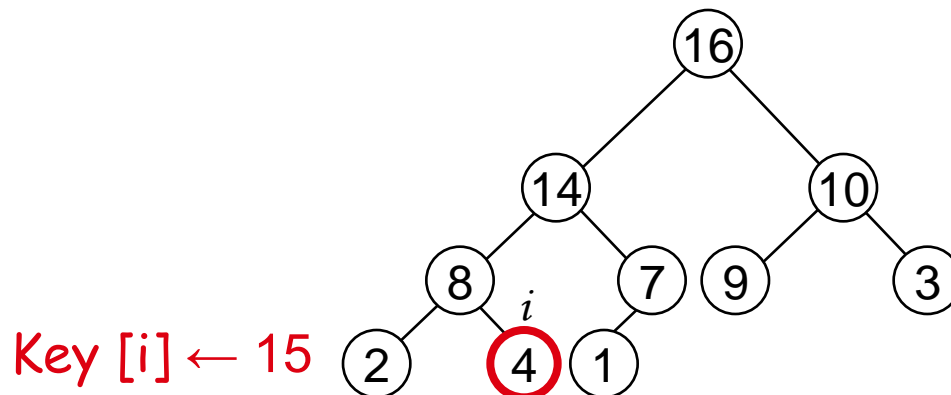


▷ remakes heap

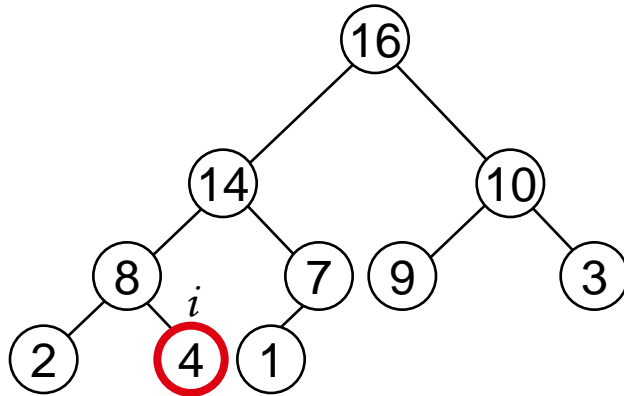
Running time: $O(\lg n)$

HEAP-INCREASE-KEY

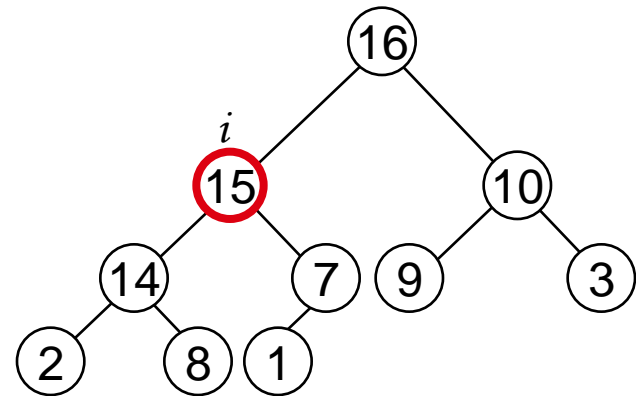
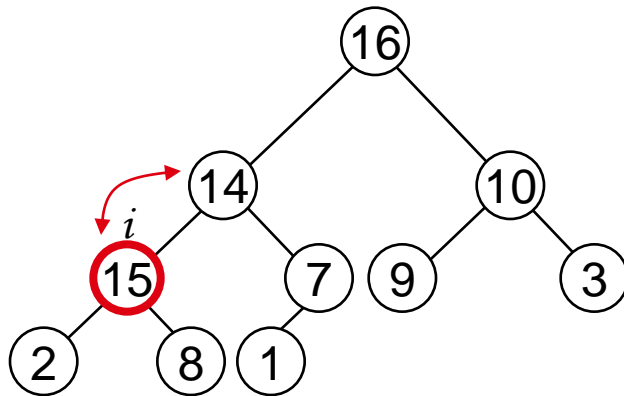
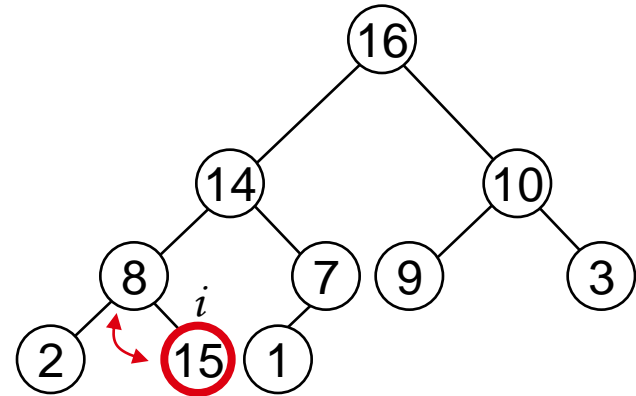
- Goal:
 - Increases the key of an element i in the heap
- Idea:
 - Increment the key of $A[i]$ to its new value
 - If the max-heap property does not hold anymore: traverse a path toward the root to find the proper place for the newly increased key



Example: HEAP-INCREASE-KEY



$Key[i] \leftarrow 15$

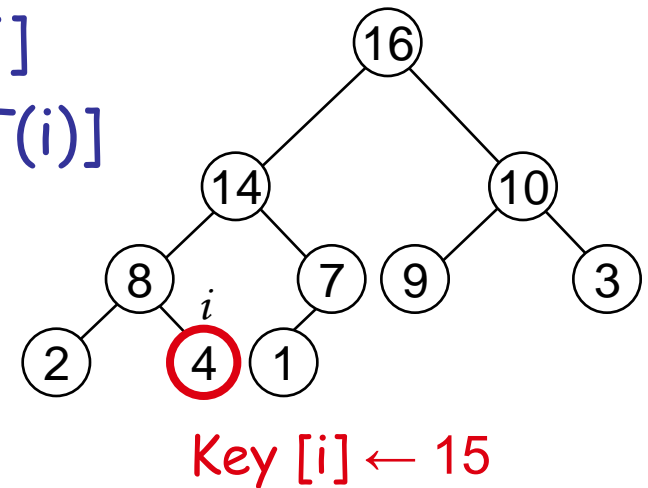


HEAP-INCREASE-KEY

Alg: HEAP-INCREASE-KEY(A, i, key)

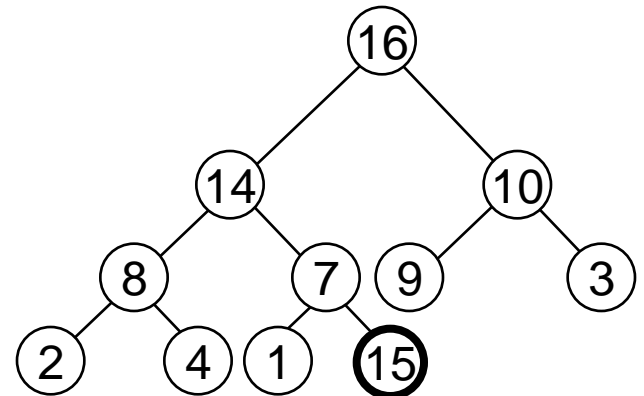
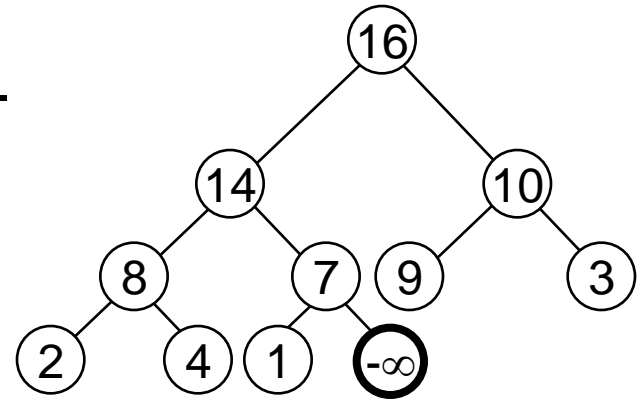
1. **if** $\text{key} < A[i]$
2. **then error** “new key is smaller than current key”
3. $A[i] \leftarrow \text{key}$
4. **while** $i > 1$ and $A[\text{PARENT}(i)] < A[i]$
5. **do** exchange $A[i] \leftrightarrow A[\text{PARENT}(i)]$
6. $i \leftarrow \text{PARENT}(i)$

- Running time: $O(\lg n)$



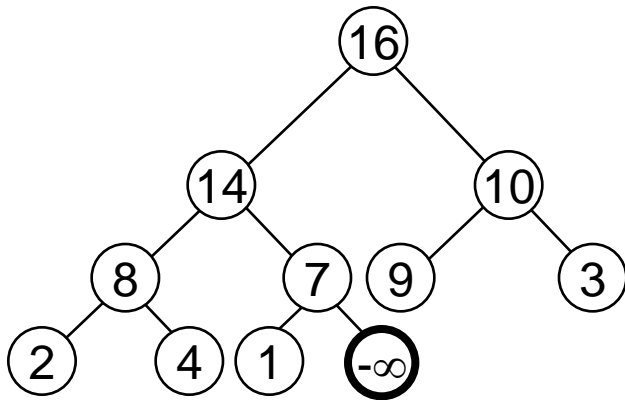
MAX-HEAP-INSERT

- Goal:
 - Inserts a new element into a max-heap
- Idea:
 - Expand the max-heap with a new element whose key is $-\infty$
 - Calls HEAP-INCREASE-KEY to set the key of the new node to its correct value and maintain the max-heap property

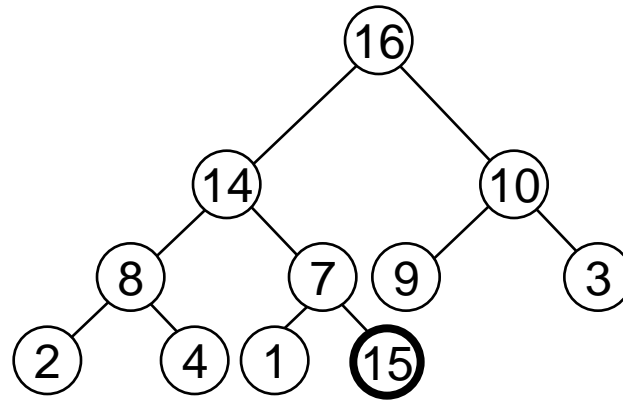


Example: MAX-HEAP-INSERT

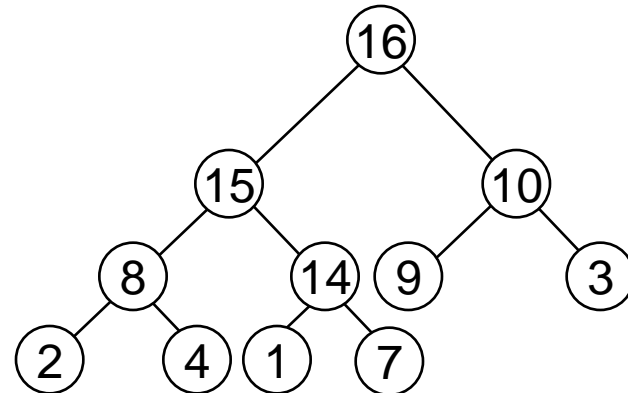
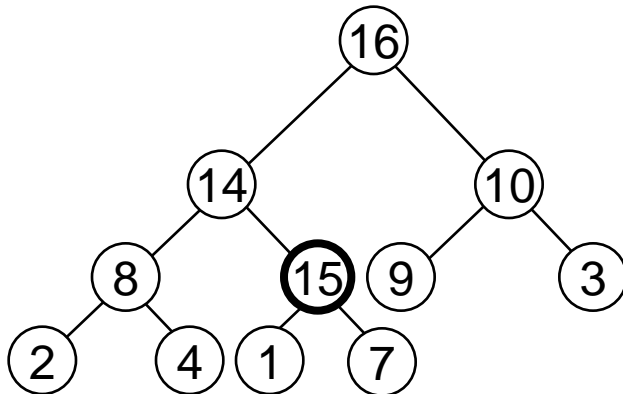
Insert value 15:
- Start by inserting $-\infty$



Increase the key to 15
Call HEAP-INCREASE-KEY on $A[11] = 15$



The restored heap containing
the newly added element



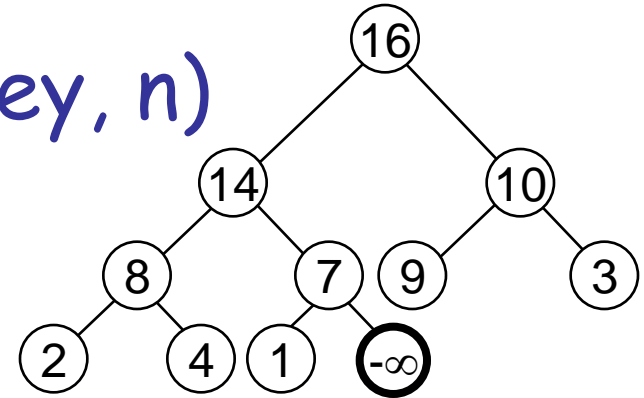
MAX-HEAP-INSERT

Alg: MAX-HEAP-INSERT(A, key, n)

1. $\text{heap-size}[A] \leftarrow n + 1$

2. $A[n + 1] \leftarrow -\infty$

3. HEAP-INCREASE-KEY($A, n + 1, \text{key}$)



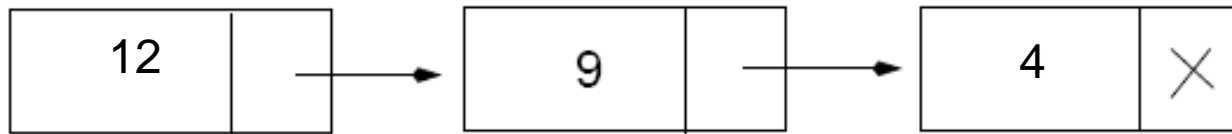
Running time: $O(\lg n)$

Summary

- We can perform the following operations on heaps:

– MAX-HEAPIFY	$O(\lg n)$	
– BUILD-MAX-HEAP	$O(n)$	
– HEAP-SORT	$O(n \lg n)$	
– MAX-HEAP-INSERT	$O(\lg n)$	} Average $O(\lg n)$
– HEAP-EXTRACT-MAX	$O(\lg n)$	
– HEAP-INCREASE-KEY	$O(\lg n)$	
– HEAP-MAXIMUM	$O(1)$	

Priority Queue Using Linked List



Remove a key: $O(1)$

Insert a key: $O(n)$

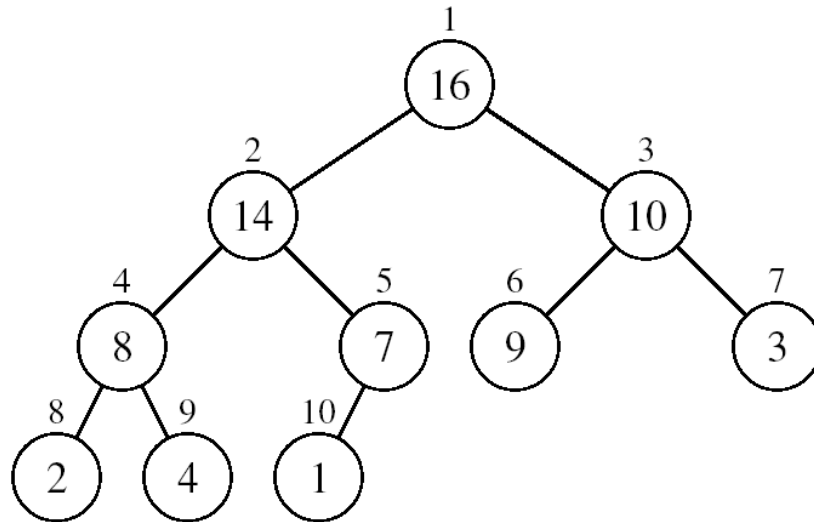
Increase key: $O(n)$

Extract max key: $O(1)$

Average: $O(n)$

Problems

Assuming the data in a max-heap are distinct, what are the possible locations of the second-largest element?



Problems

- (a) What is the maximum number of nodes in a max heap of height h ?
- (b) What is the maximum number of leaves?
- (c) What is the maximum number of internal nodes?

Problems

- Demonstrate, step by step, the operation of Build-Heap on the array

$A=[5, 3, 17, 10, 84, 19, 6, 22, 9]$

Problems

- Let A be a heap of size n . Give the most efficient algorithm for the following tasks:

(a) Find the sum of all elements

(b) Find the sum of the largest $\lg n$ elements