CH08-320201
Algorithms and Data Structures

Lecture 7/8 — 27 Feb 2018

Prof. Dr. Michael Sedlmair

Jacobs University
Spring 2018

This & that

- “homework is too hard”
    - Strategies:
        - background reading (the Cormen book gives background to most of the things that we are talking about in class)
        - go to tutorials and ask questions
        - use the Moodle discussion forum
    - What will change:
        - Me: will try to have a closer connection between level of difficulty/rigor between class and homework
        - A little bit of independent thinking will remain to be necessary for assignments (important for learning effect)
    - in general —> tell me/us your feedback, it’s not supposed to be a “one man show”
- medical excuse policy refined:
    - http://vis.jacobs-university.de/teaching/ads/18s/

2. Sorting and Searching

Recall: Sorting Problem

- Input:
  - sequence $<a_1, a_2, \ldots., a_n>$ of numbers.
- Output:
  - permutation $<a'_1, a'_2, \ldots, a'_n>$
  - such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

Recall: Insertion & Merge Sort

Time complexity:

	Insertion Sort	Merge Sort
Best Case	$\Theta(n)$	$\Theta(n \lg n)$
Average Case	$\Theta(n^2)$	$\Theta(n \lg n)$
Worst Case	$\Theta(n^2)$	$\Theta(n \lg n)$

Visualizations:

http://www.sorting-algorithms.com/insertion-sort
http://www.sorting-algorithms.com/merge-sort

How about storage space complexity?

In situ

- Definition:
  - In-situ algorithms refer to algorithms that operate with $\Theta(1)$ memory.

- In-situ sorting:
  - Sorting algorithms that need only a constant number of additional storings.
- Insertion Sort:
  - In-situ sorting.
- Merge Sort:
  - Not in-situ sorting.

2.1 Heap Sort

Motivation

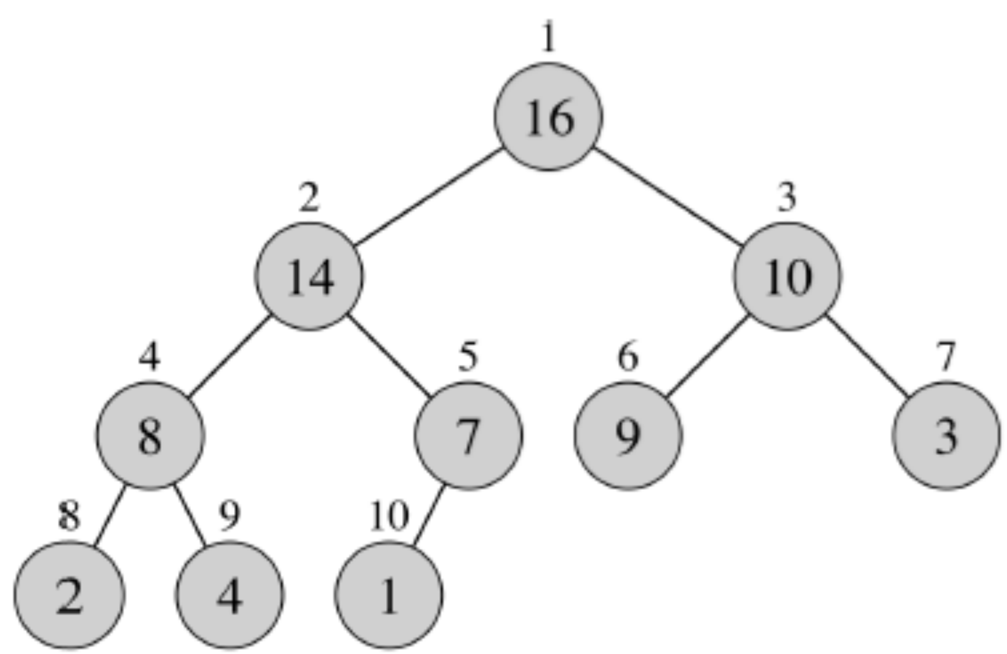
- Try to develop an in-situ sorting algorithm with asymptotic runtime $\Theta(n \lg n)$.
- Use a sophisticated data structure to support the computations.

Data structure: Heap

Defintion.

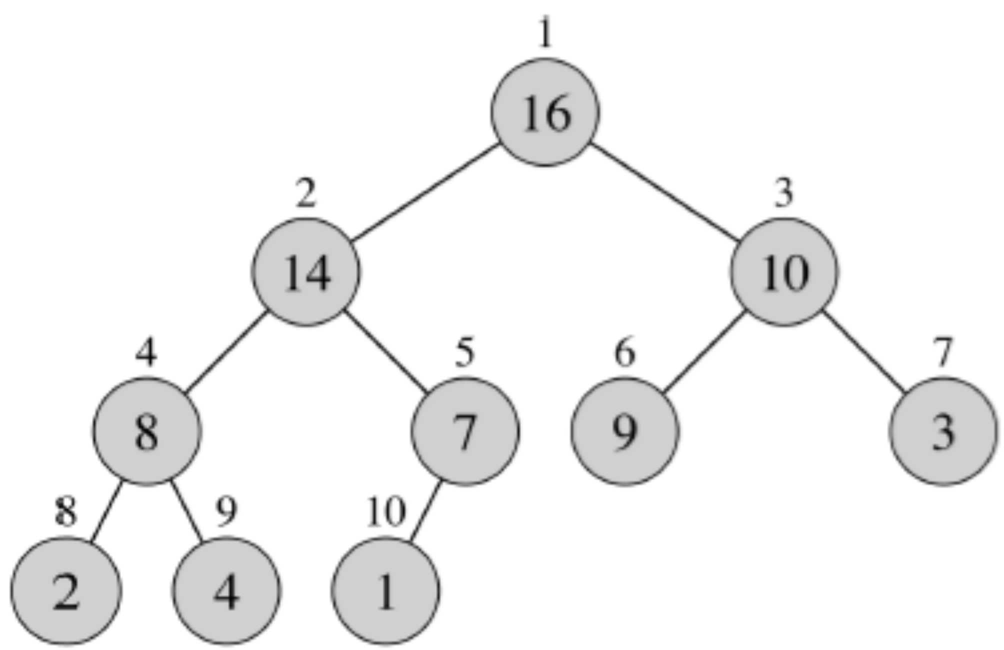
A (binary) heap data-structure is an array which can be viewed as a nearly complete binary tree:

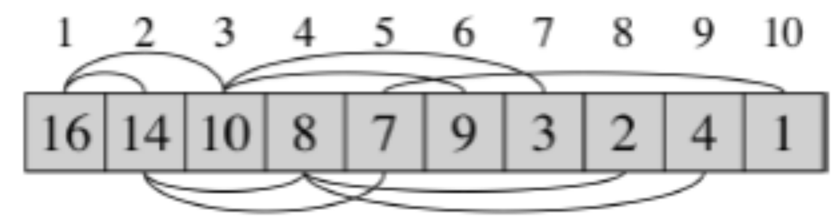
each level is completely full except possibly the last level, which is filled from left to right.



Heap as array

A heap can be stored as an array:





Heap as array

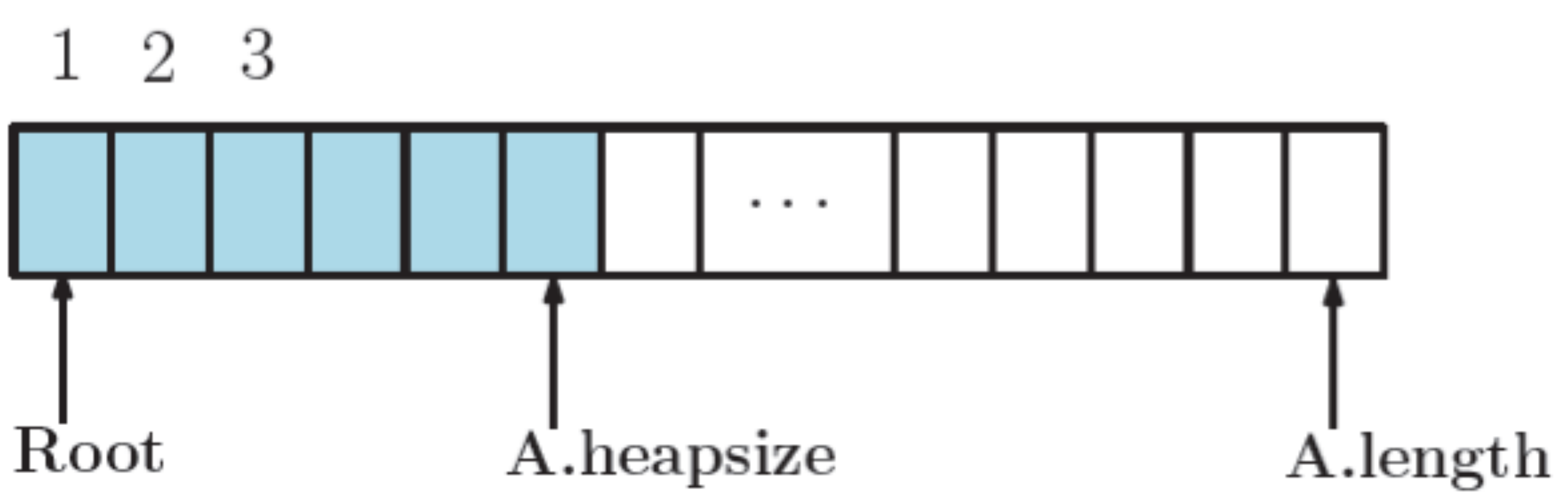
The array A representing the heap has two attributes:

– *A.length*

– *A.heapsize*

such that $0 \leq A.heapsize \leq A.length$.

There are only *A.heapsize* valid elements of the heap.

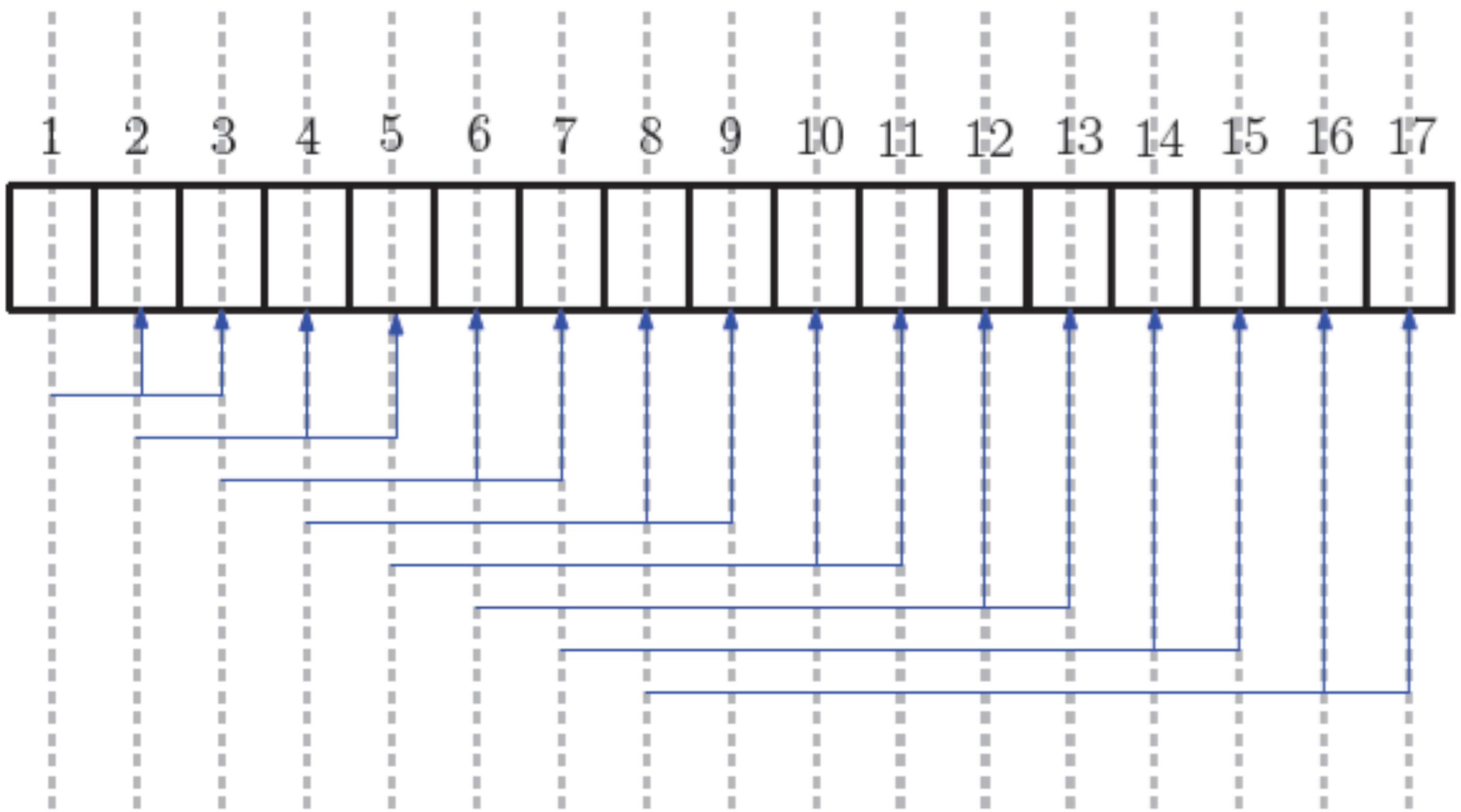


A[1] is the root of the heap (root of the binary tree).

Heap as array

Given index i of an element of A, we can calculate:

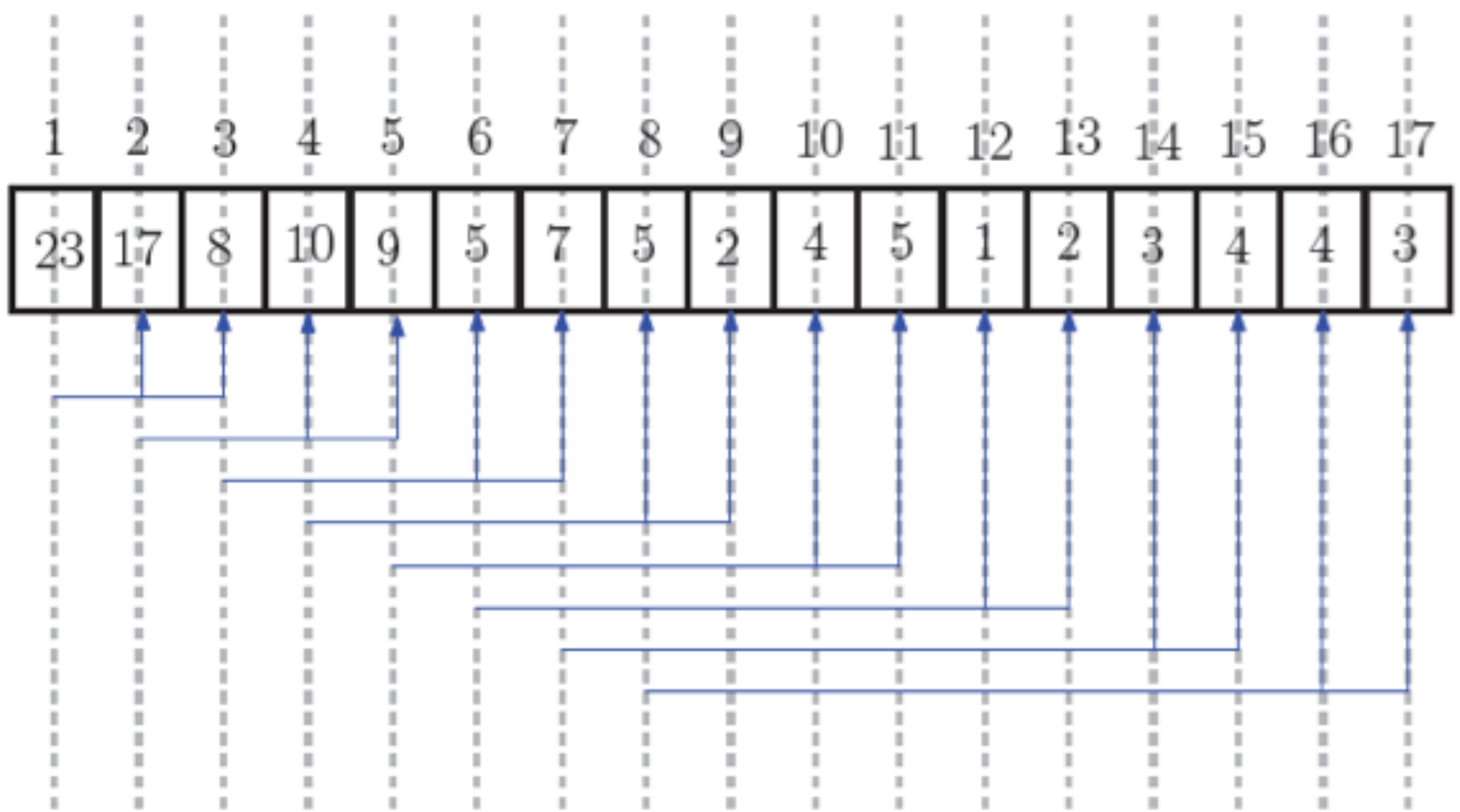
- Parent(i): return floor (i/2) ; // Right shift by 1 bit.
- Left(i): return 2i ; // Left shift by 1 bit.
- Right(i): return 2i + 1; // Left shift by 1 bit // and set LSB to 1

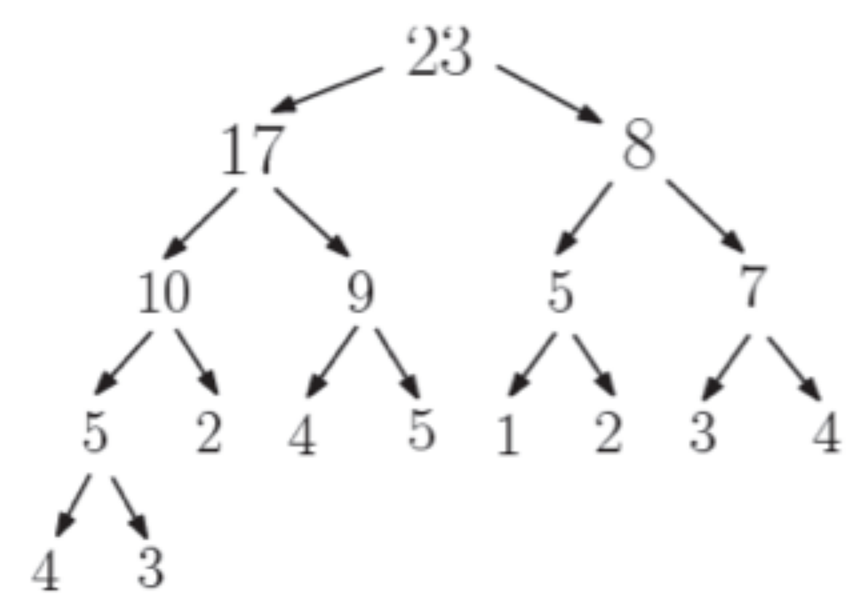


Max-heap property

In a max-heap, for every node i (other than the root),

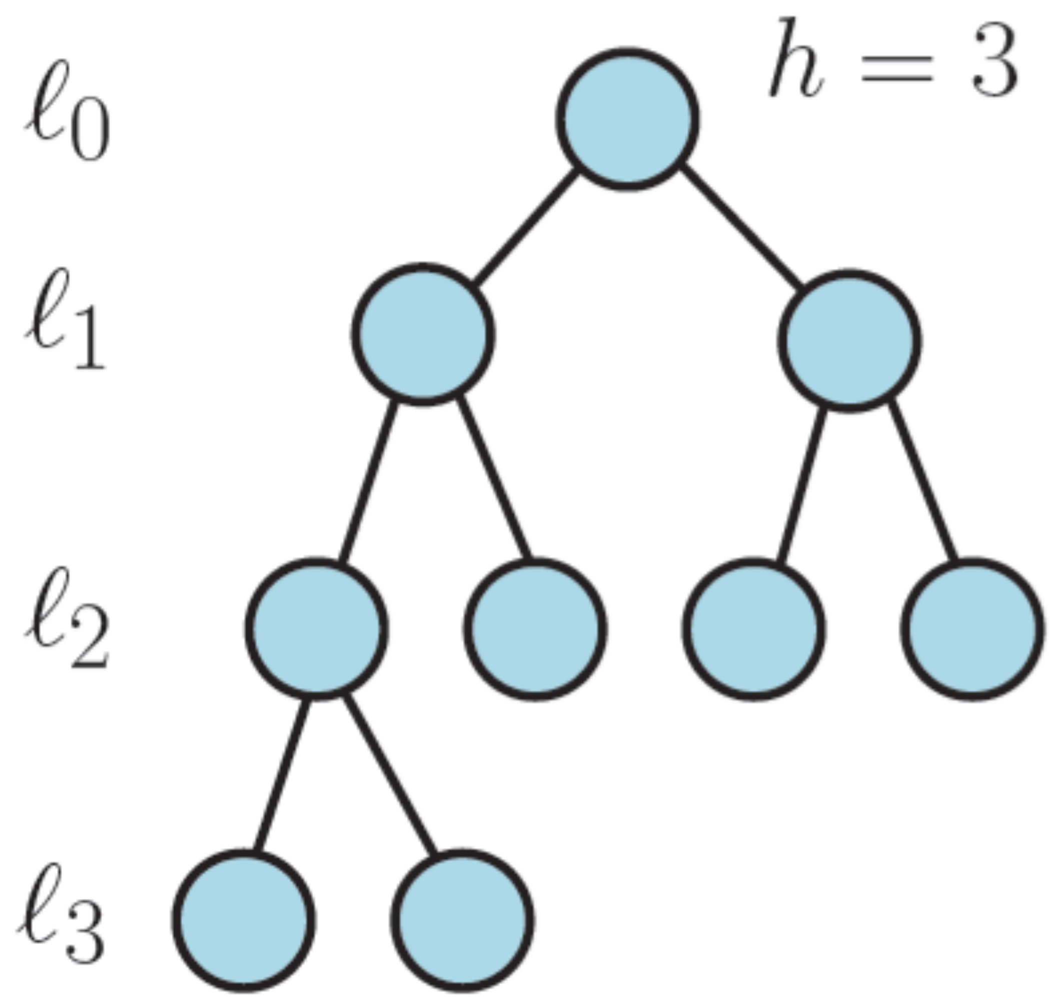
A[Parent(i)] ≥ A[i].





Recall: Height of a rooted tree

- The height of a node x is the length of the longest simple downward path from x to a leaf.
- The height of a rooted tree is the height of its root.



Heap height

Theorem:

A heap with n elements has height $h = \lfloor \lg n \rfloor$.

Heap height

Proof:

Heap height h implies that there are h + 1 levels (levels 0 to h).

As a heap is a nearly complete binary tree,
the last guaranteed complete level is level h - 1.

The level h may be incomplete but has at least one element.

Number of elements in complete levels 0 to h - 1 is

$$1 + 2 + 2^2 + \ldots + 2^{h-1} = 2^h - 1.$$

So, $n > 2^h - 1$ or (since it is an integer) $n \geq 2^h$.

If all levels 0 to h were complete, the number of elements would be $2^{h+1} - 1$. So, $n \leq 2^{h+1} - 1$.

Heap height

Proof (continued):

Combining the two inequalities:

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

As $2^{h+1} > 2^{h+1} - 1 \geq 2^h$ for $h \geq 0$,

$$h+1 > \lg(2^{h+1} - 1) \geq h$$

Thus, $\lg(2^{h+1} - 1) = h + \alpha$ with $\alpha \in [0,1)$,

which leads to $h \leq \lg n \leq h + \alpha$ with $\alpha \in [0,1)$.

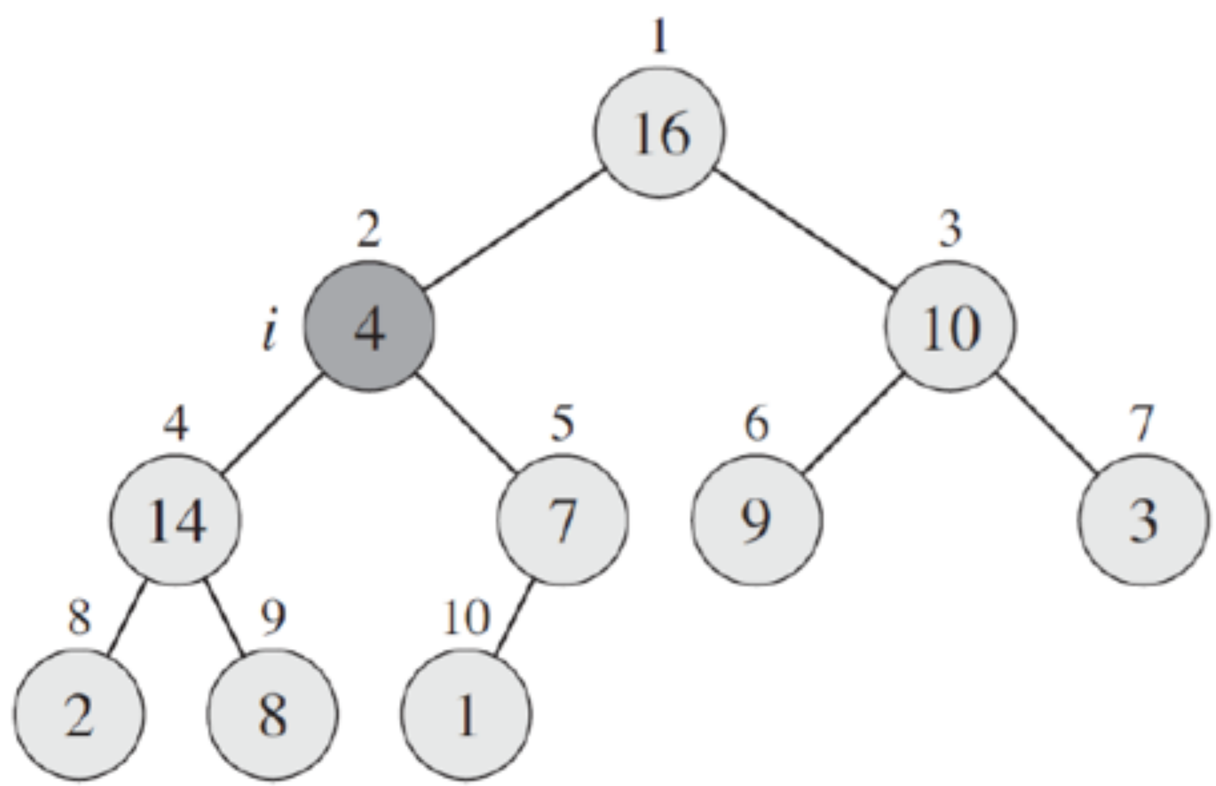
Hence, $h = \lfloor \lg n \rfloor$.

Max-Heapify (A,i)

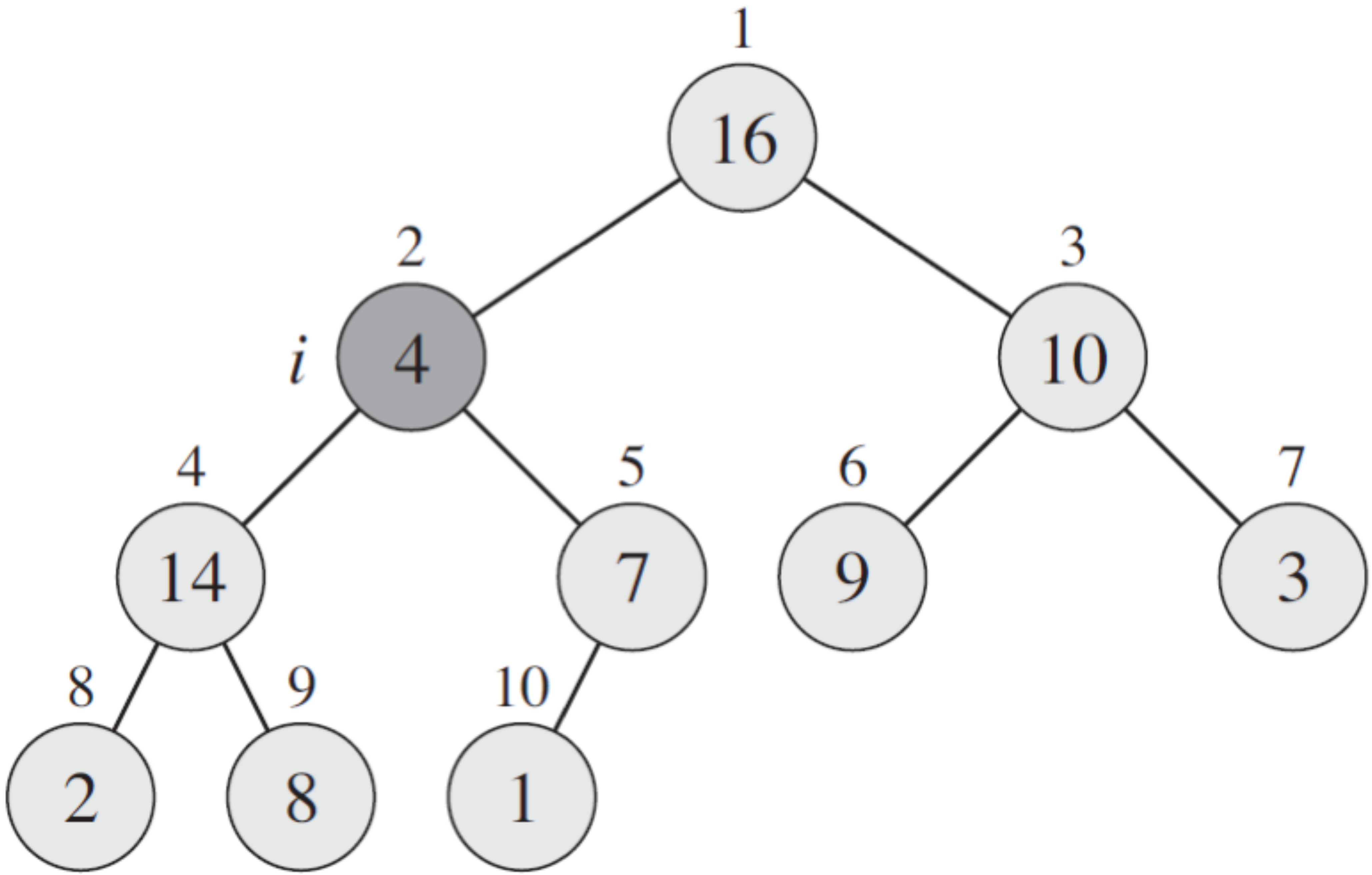
Precondition:

When Max-Heapify (A, i) is called,

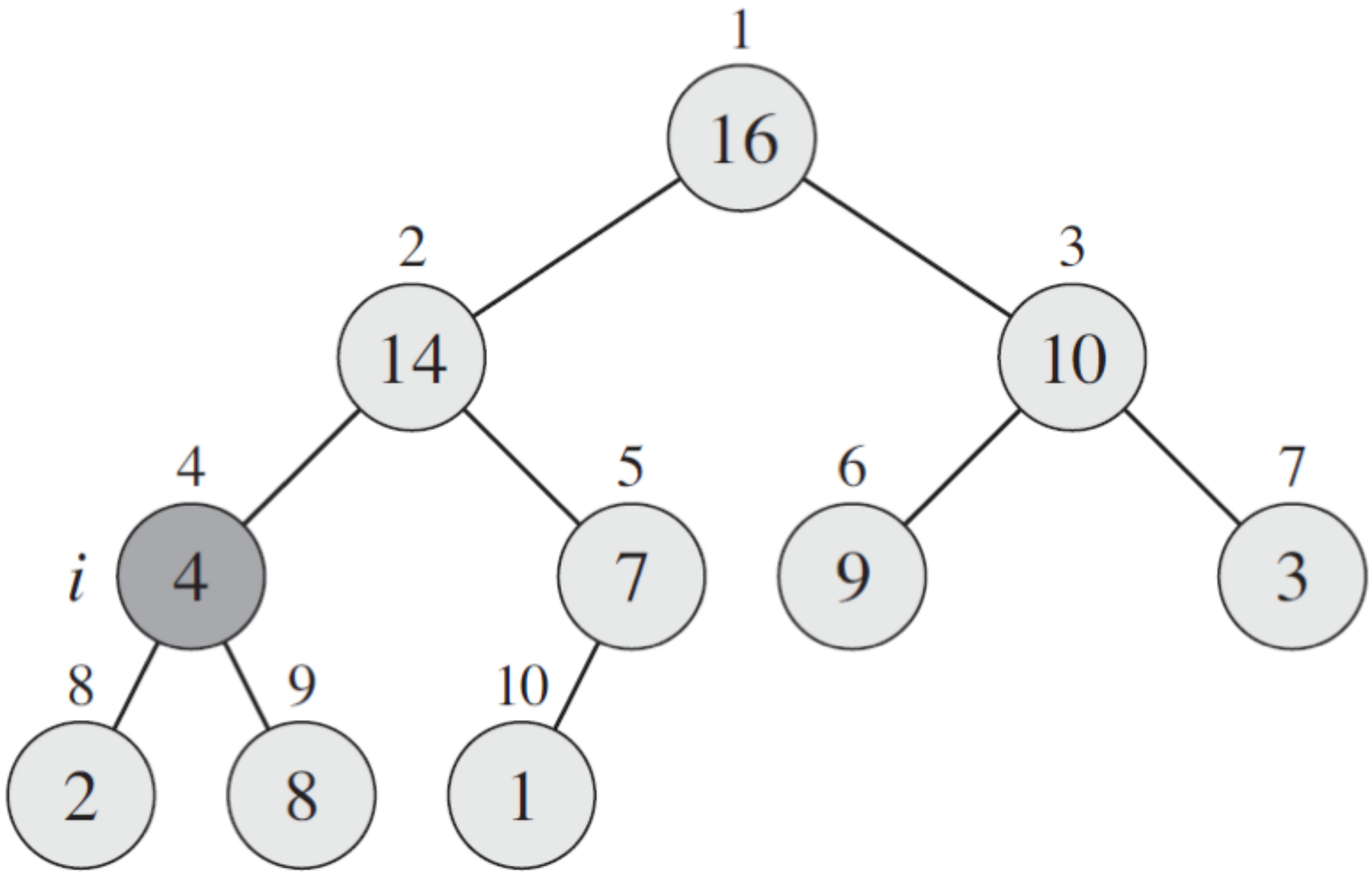
binary-trees rooted at Left(i) and Right(i) are valid max-heaps, but A[i] may be smaller than its children.



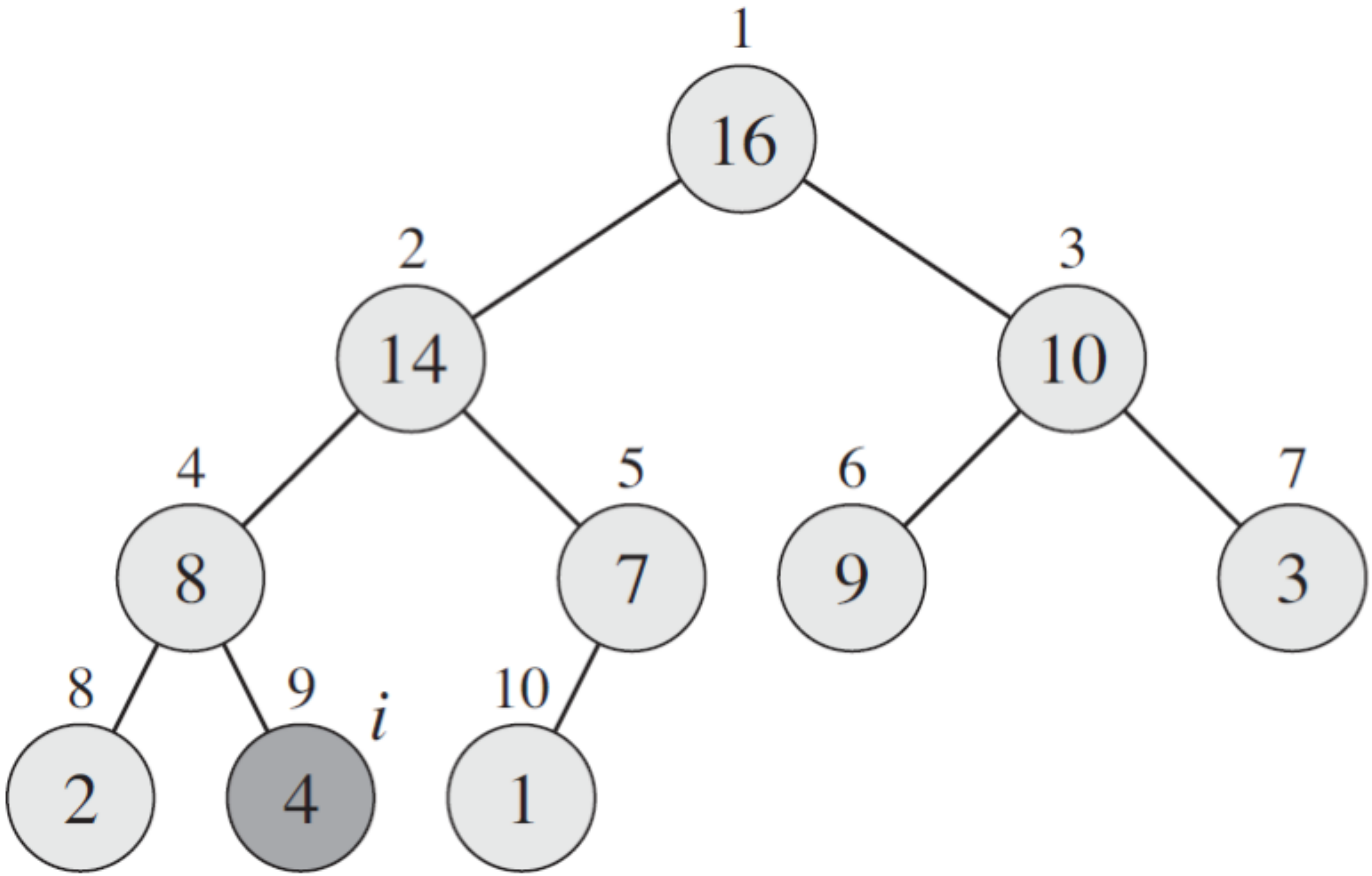
Max-Heapify (A,i)



Max-Heapify (A,i)



Max-Heapify (A,i)



Max-Heapify (A,i)

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 (A,i)

Complexity:

It is $O(h) = O(\lg n)$,

as in the worst case the new element has to go down all the way to the last level.

Converting an existing array A to a max-heap

- Call MAX-HEAPIFY, on what set?
  - all inner nodes
    —> float them down if necessary (MAX-HEAPIFY)
  - not necessary to call on leaves node

Leaves of a heap of size n

Let *n = A.heapsize.*

Where is the parent of the last element of a heap?

At index $\lfloor n/2 \rfloor$.

Therefore, the element at index $\lfloor n/2 + 1 \rfloor$

does not have a child in the heap, and hence is a leaf.

In a heap, there are $\lceil n/2 \rceil$ leaves:

from index $\lfloor n/2 + 1 \rfloor$ to n.

Each leaf is the root of a valid max-heap of size 1.

Converting an existing array A to a max-heap

BUILD-MAX-HEAP(A)

$A.heap\text{-}size = A.length$
for $i = \lfloor A.length/2 \rfloor$ **downto** 1
    MAX-HEAPIFY(A, i)

Loop invariant:

At the start of each iteration of the for loop,
each node i + 1, ..., n is the root of a max-heap.

Converting an existing array A to a max-heap

Initialization: $i = \lfloor n/2 \rfloor$.

Nodes with indices > *i* are leaves.

Maintenance:

At iteration i, nodes with indices *i* + 1, ..., *n* are roots of valid max-heaps, but the *i*-th node is not (necessarily).

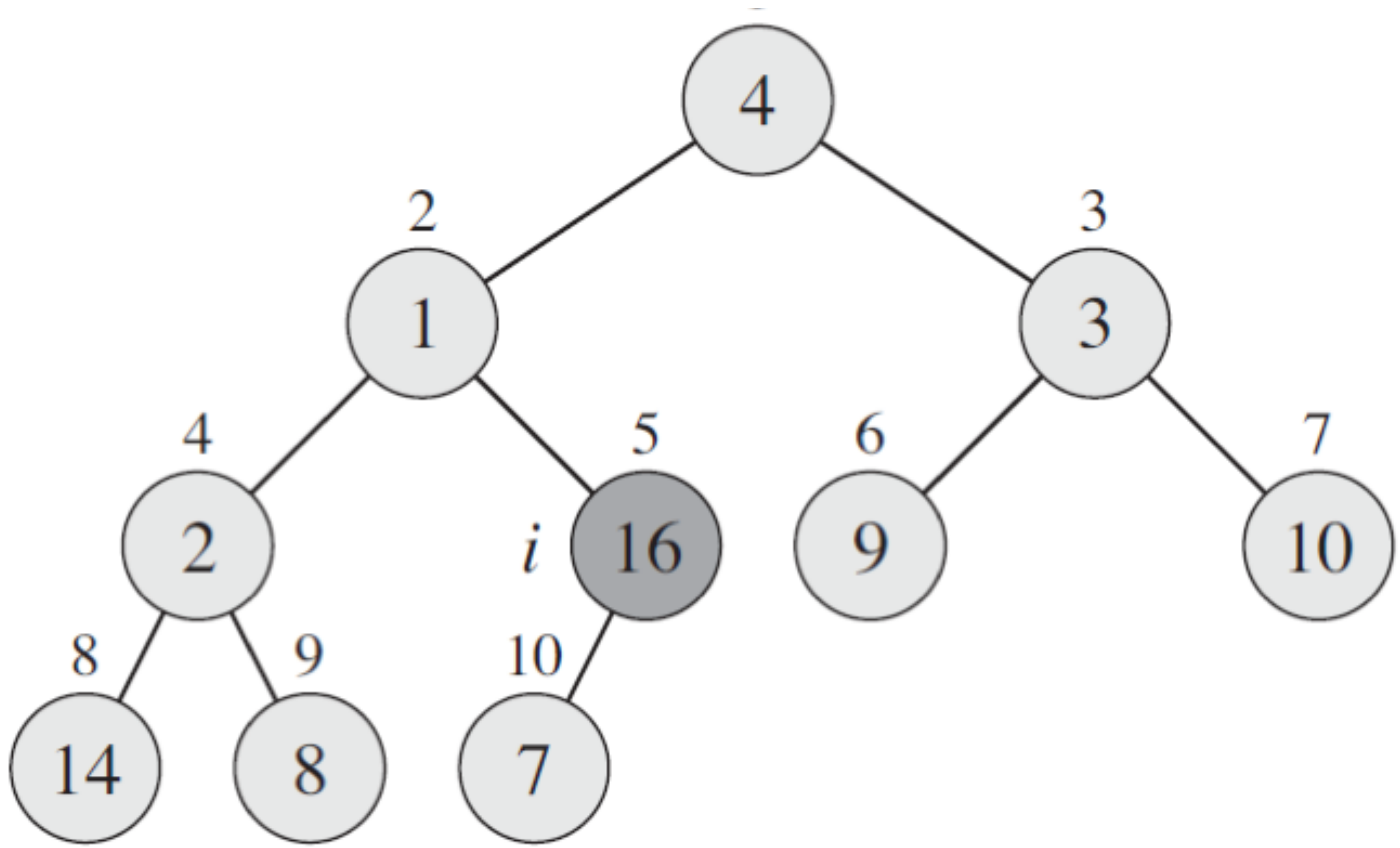
But this is exactly the precondition of Max-Heapify.

Termination: *i* = 0.

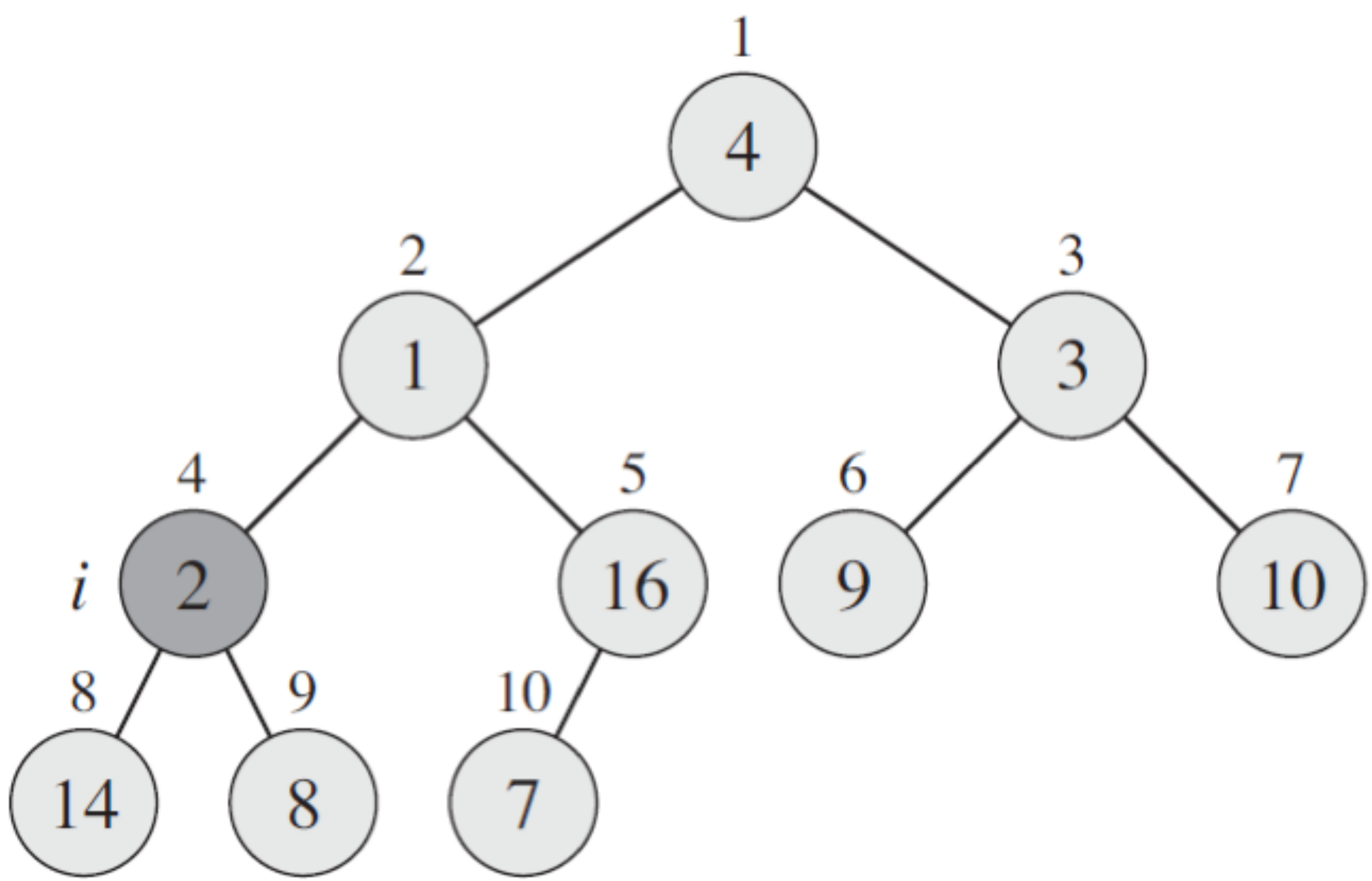
Node *i* = 1 is the root of a valid max-heap.

Build-max-heap

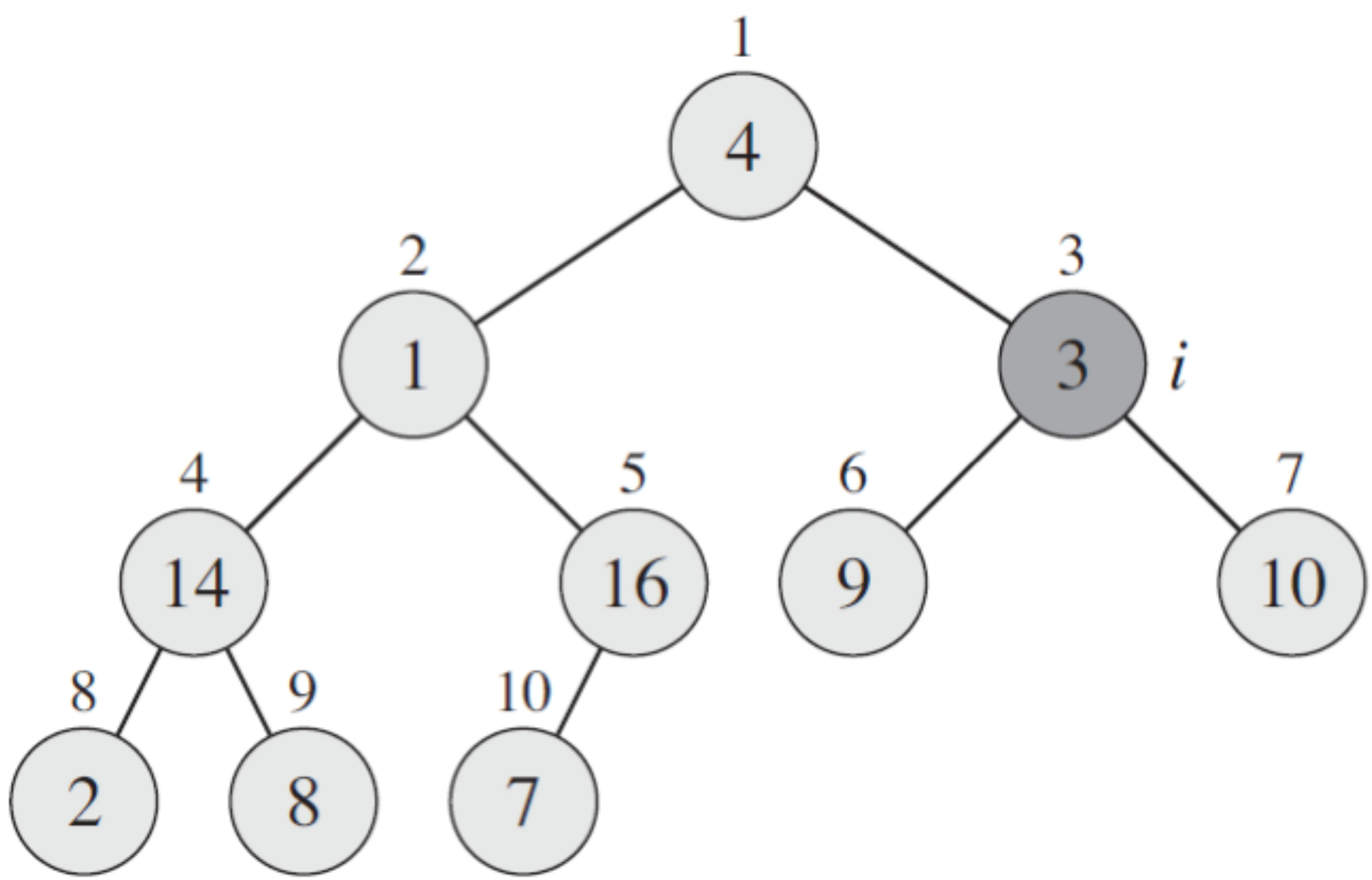
A	4	1	3	2	16	9	10	14	8	7



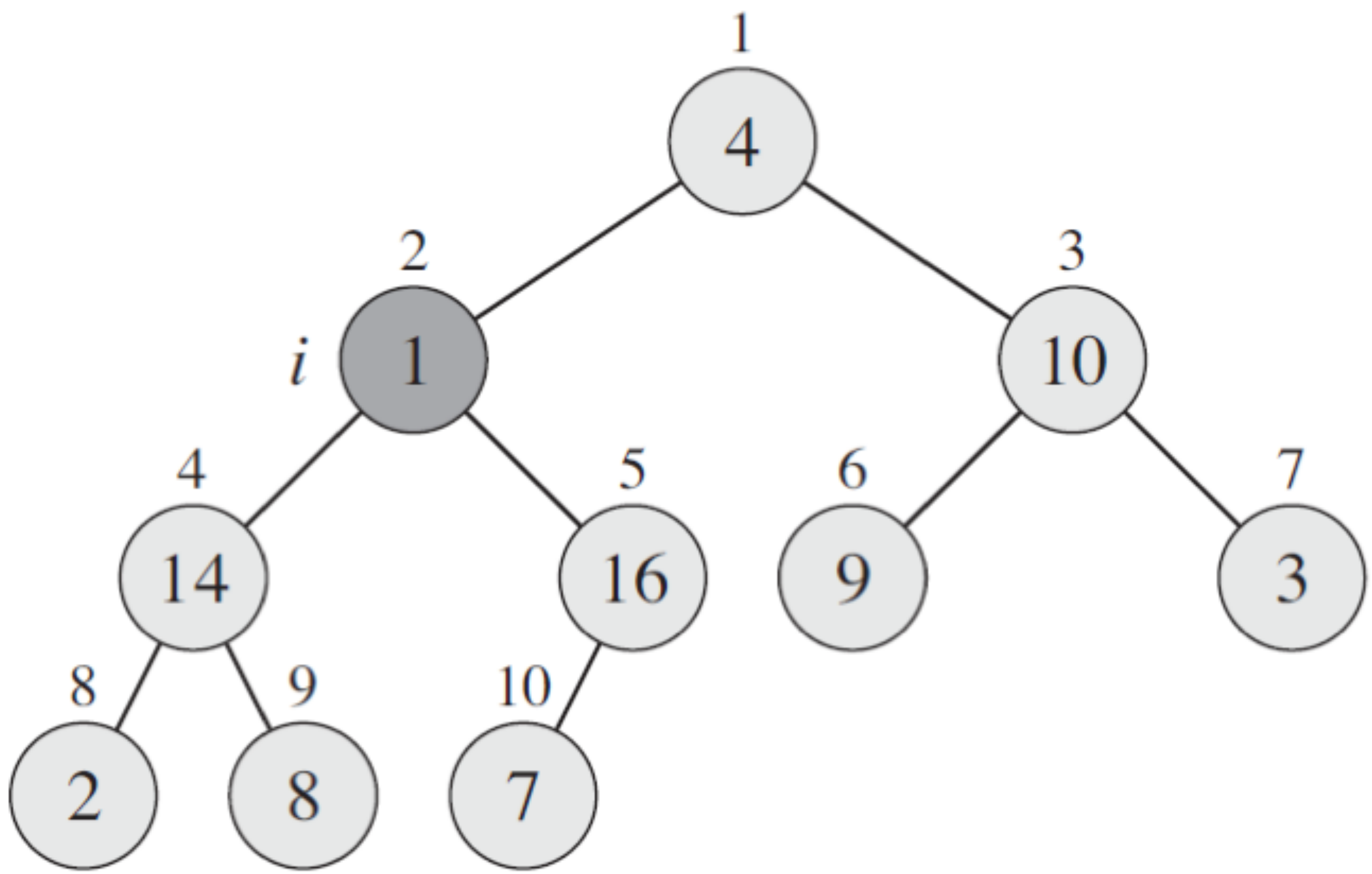
Build-max-heap



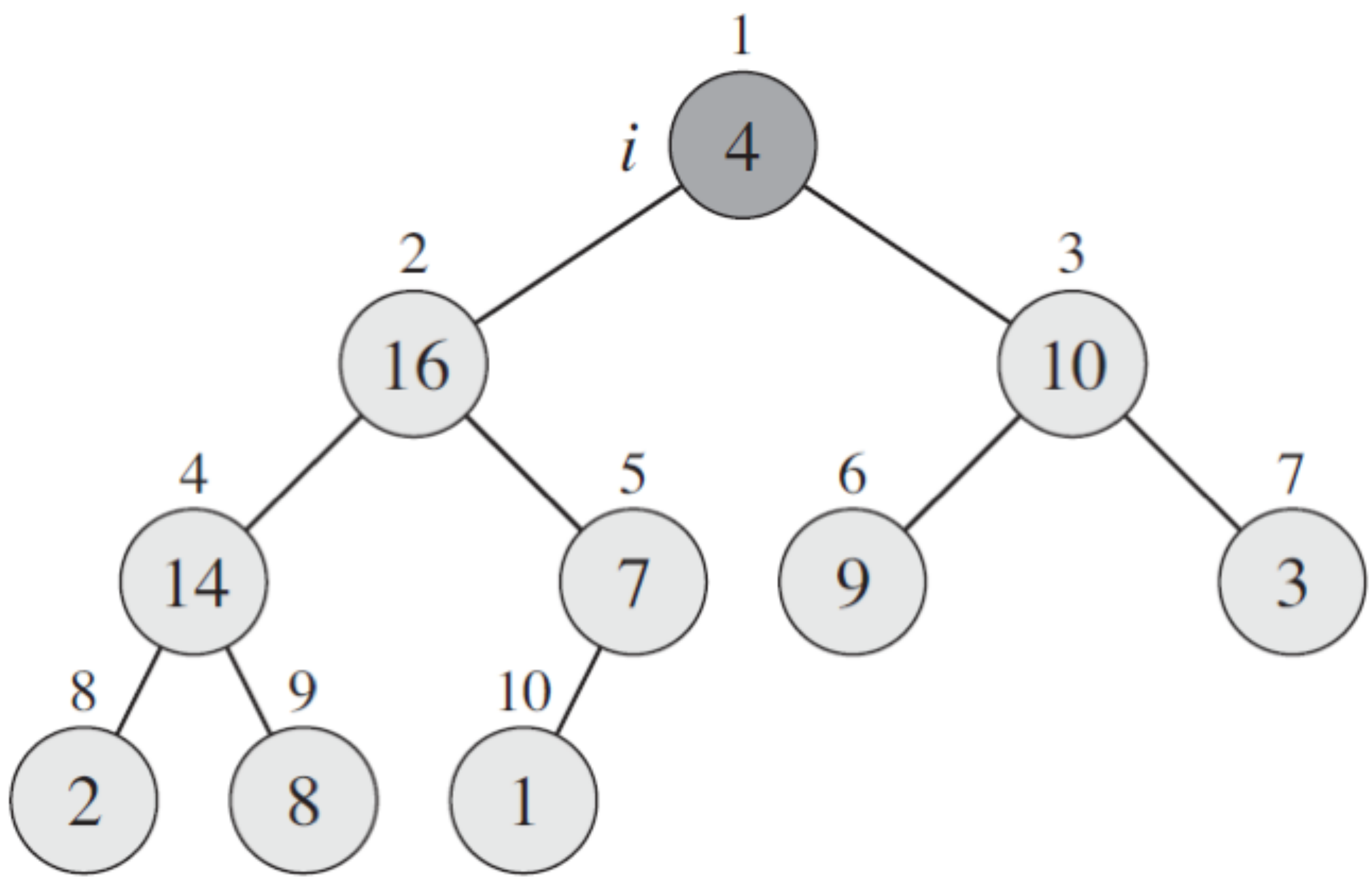
Build-max-heap



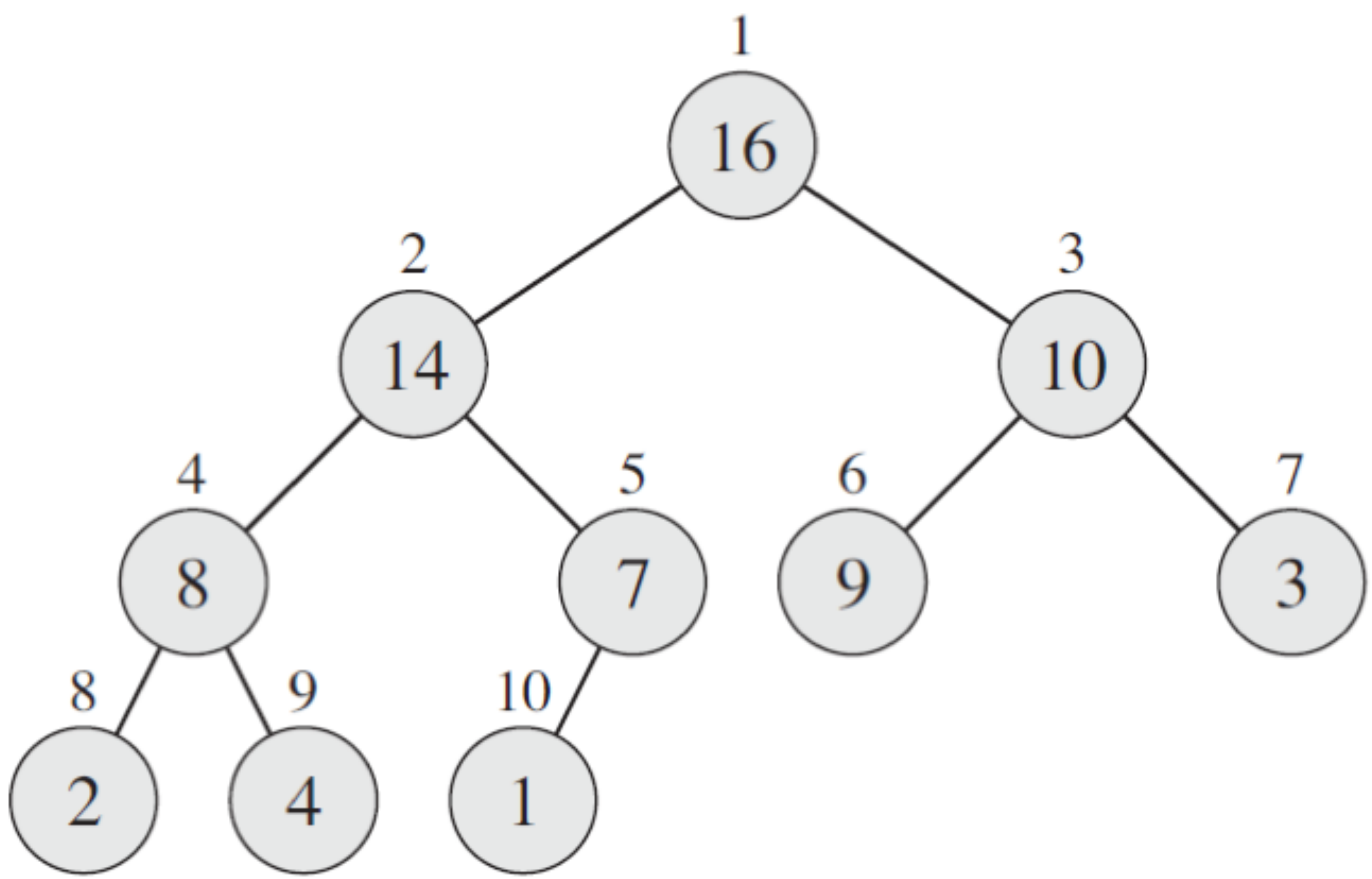
Build-max-heap



Build-max-heap



Build-max-heap



Build-max-heap

What is the time complexity?

Theorem:

Let m_h be the number of nodes of height h in any n element heap $T(n)$.

Then, $m_h(T, n) \leq \left\lceil \frac{n}{2^{h+1}} \right\rceil$

(Proof by induction over h)

Build-max-heap

Time complexity:

Time needed by Max-Heapify when called on a node of height *h* is O(*h*).

Therefore, total cost of Build-Max-Heap(A) is upper bounded by

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

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

$$= O(n).$$

using $\sum_{k=0}^{\infty} k\, x^k = \frac{x}{(1-x)^2}$ if $|x| < 1$.

Build-max-heap

Time complexity:

Conclusion:
We can convert an unordered array into a max-heap in linear time.

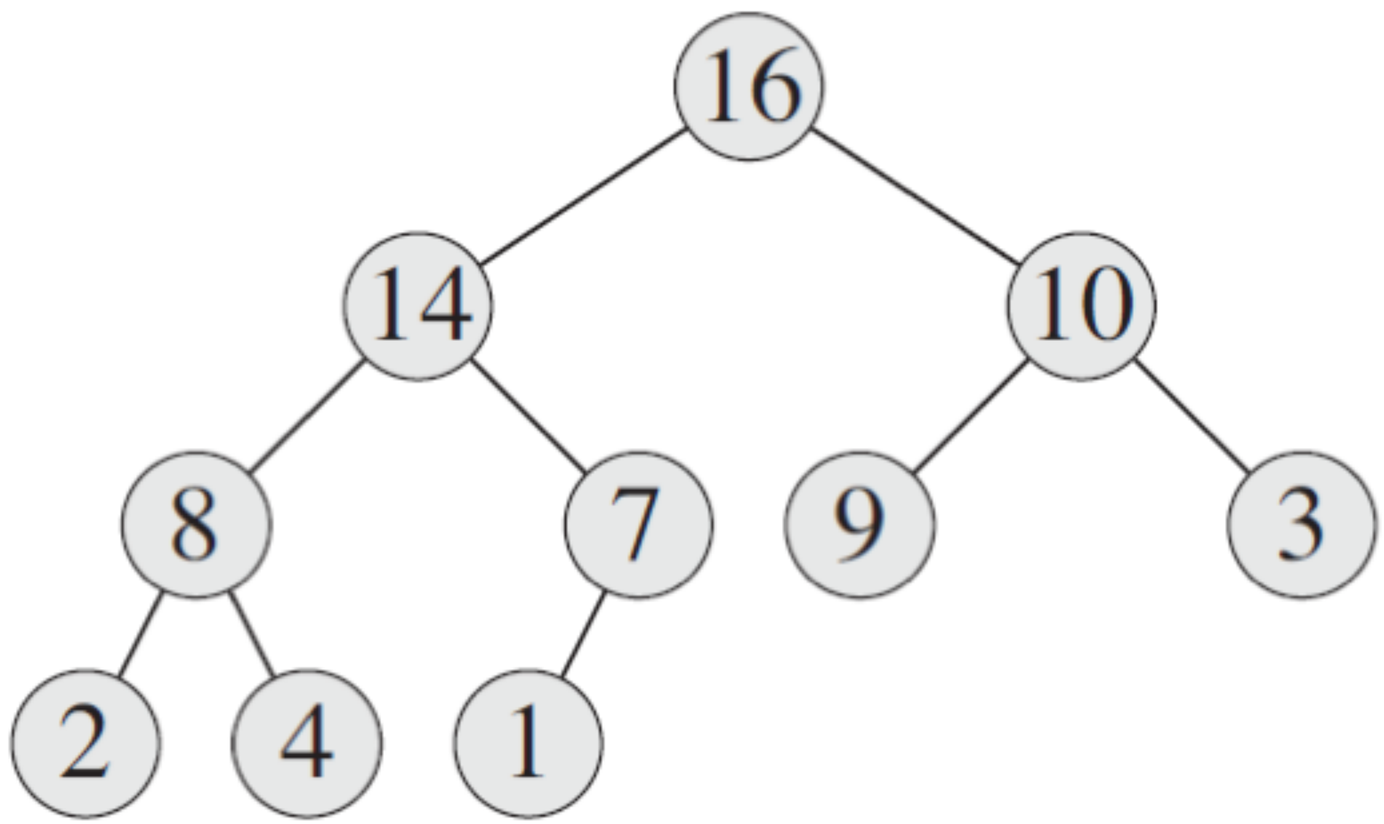
Heap Sort

- Start by generating a max-heap.
- The maximum element of a max-heap is at the root.
- Put it in its right sorted place at n = A.*heapsize* by swapping it with the last element A[n], which now becomes A[1].
- Decrement the heap size to create a smaller heap and thus implicitly remove the last element (the maximum) from the heap.
- The new A[1] may not satisfy the max-heap property, so float it down.
- Iterate.

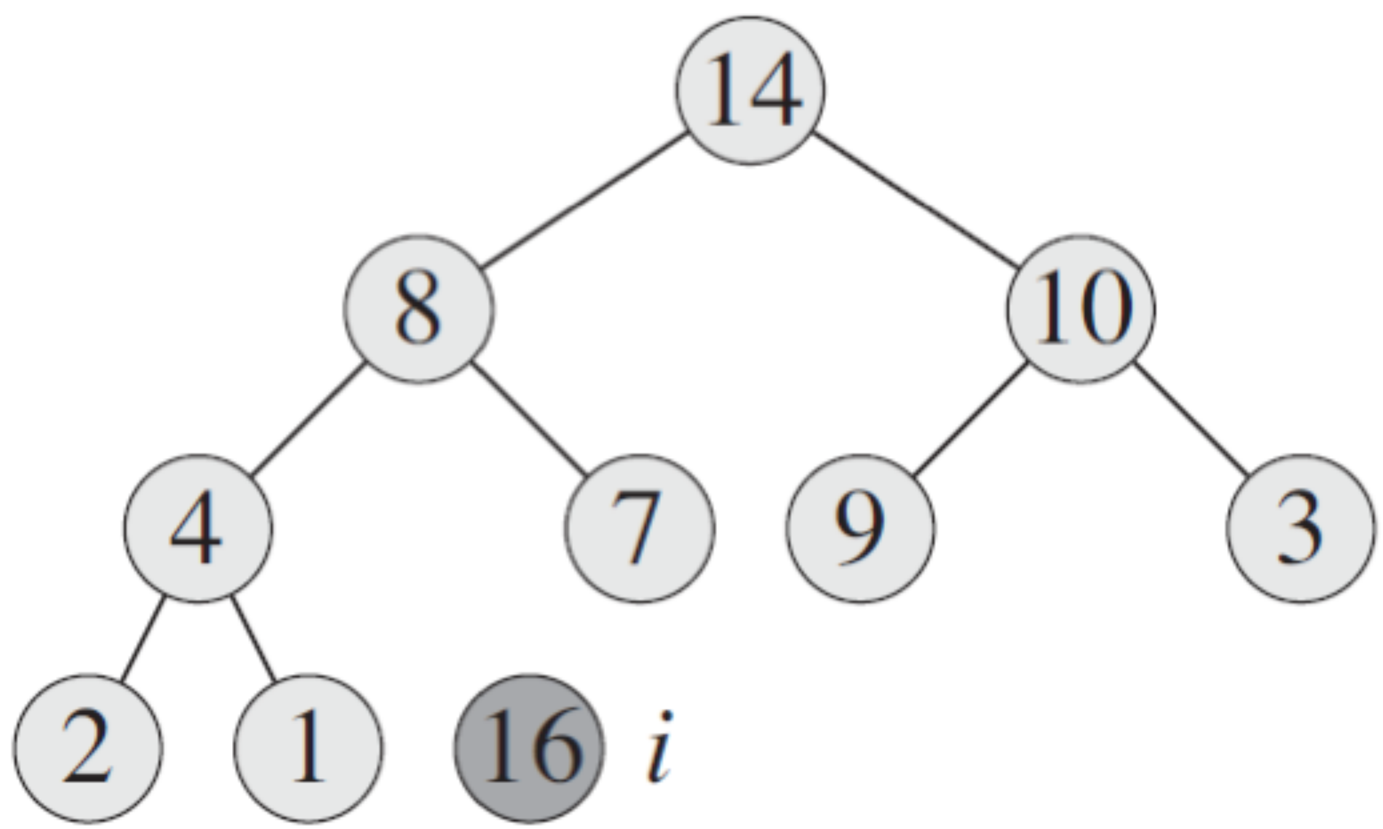
Heapsort

```
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)
```

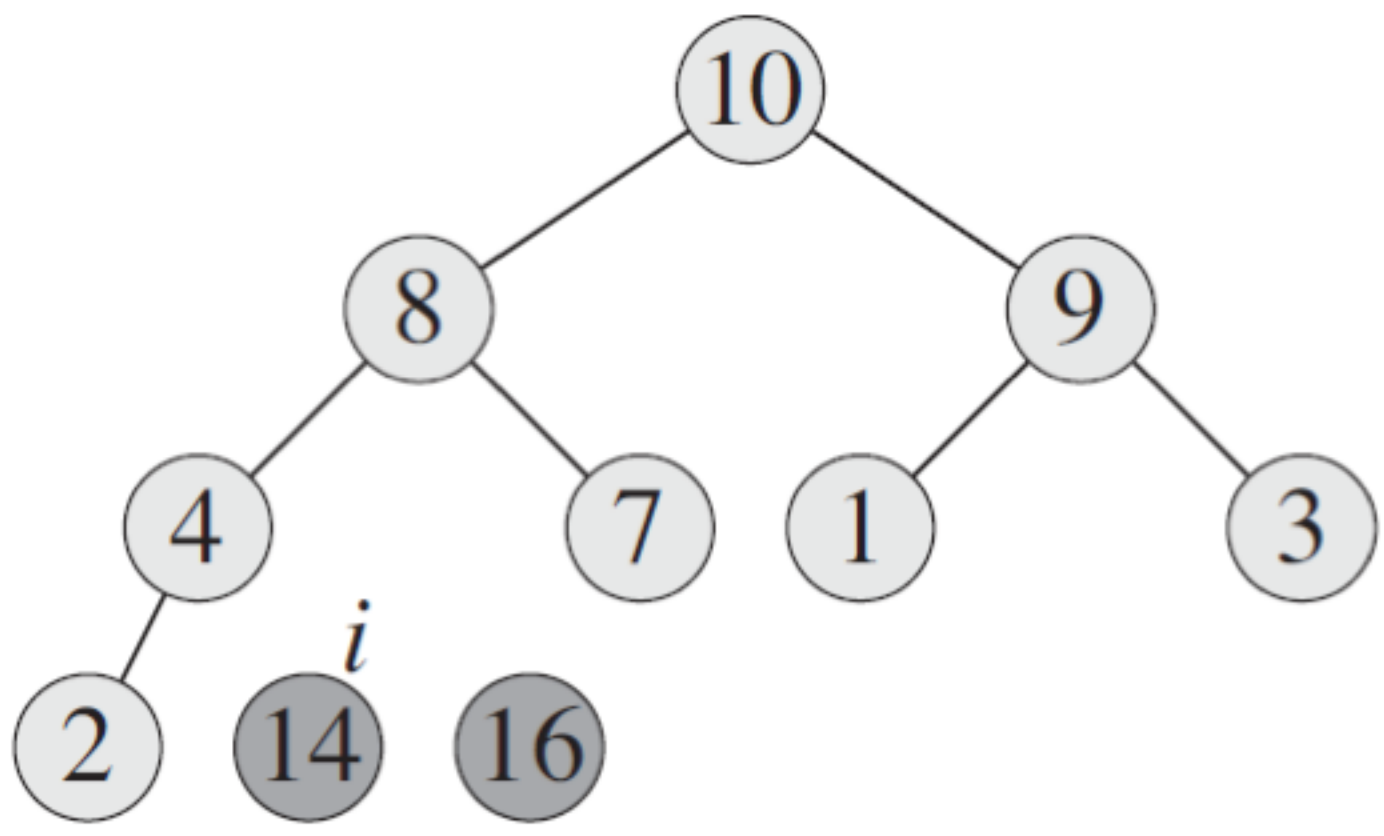
Heapsort



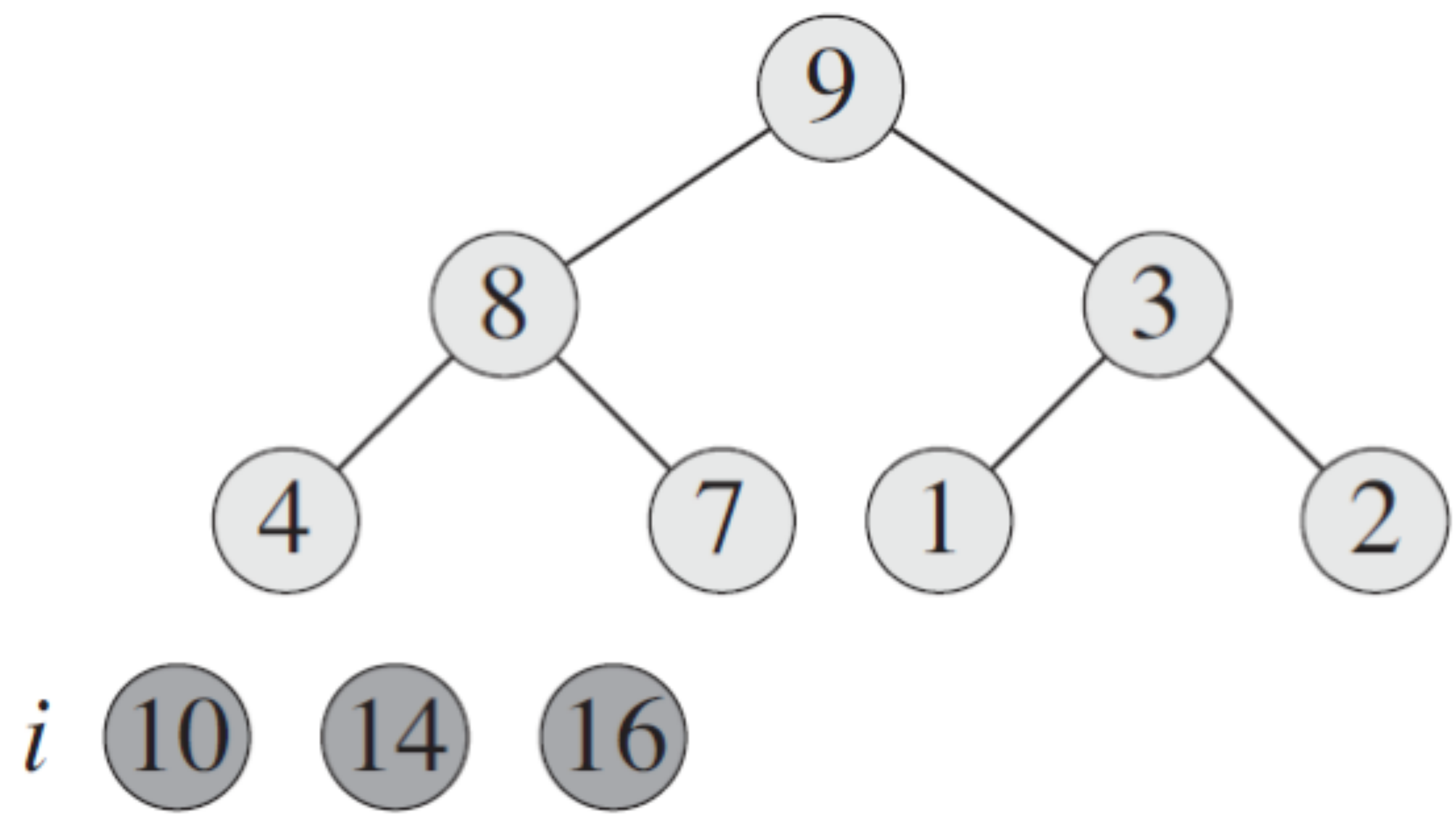
Heapsort



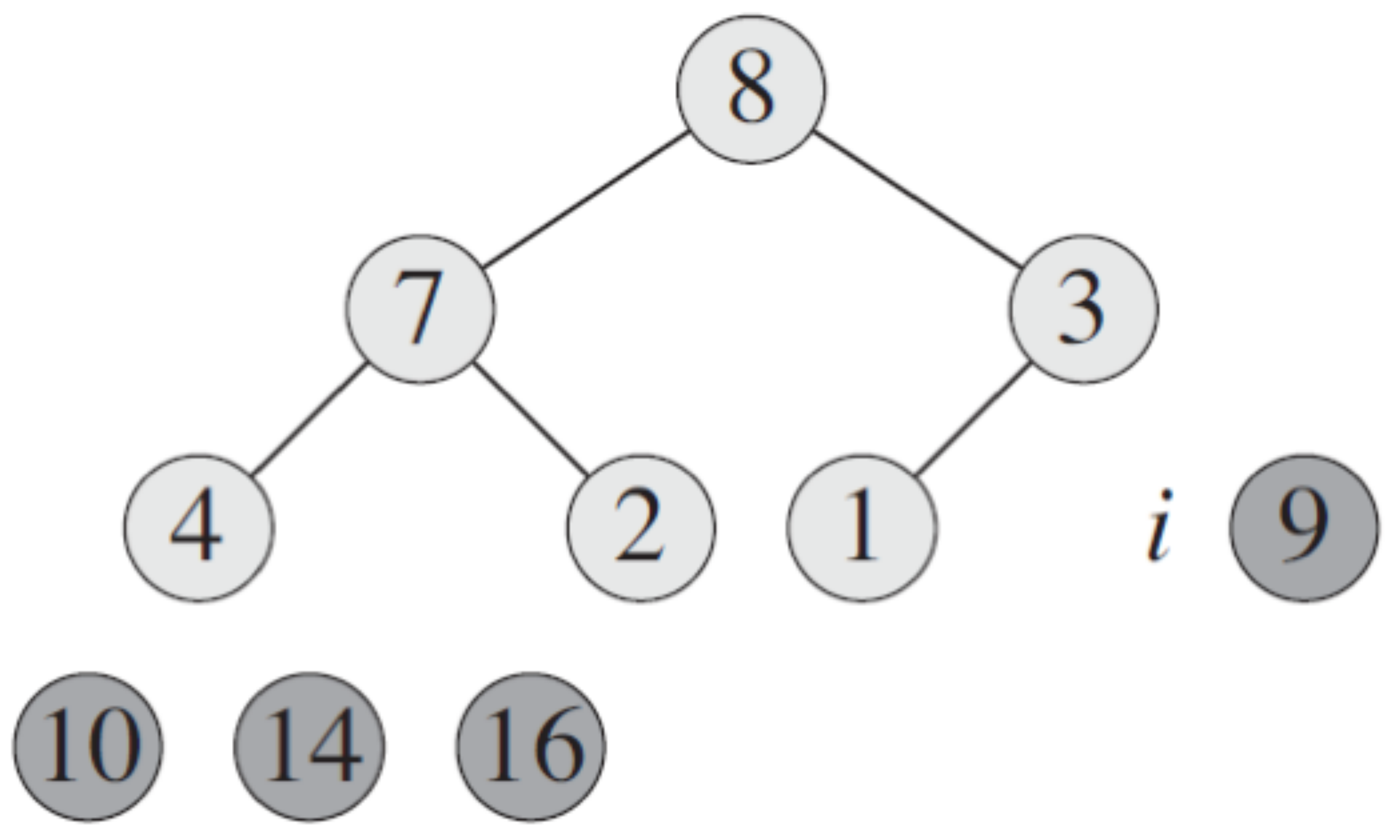
Heapsort



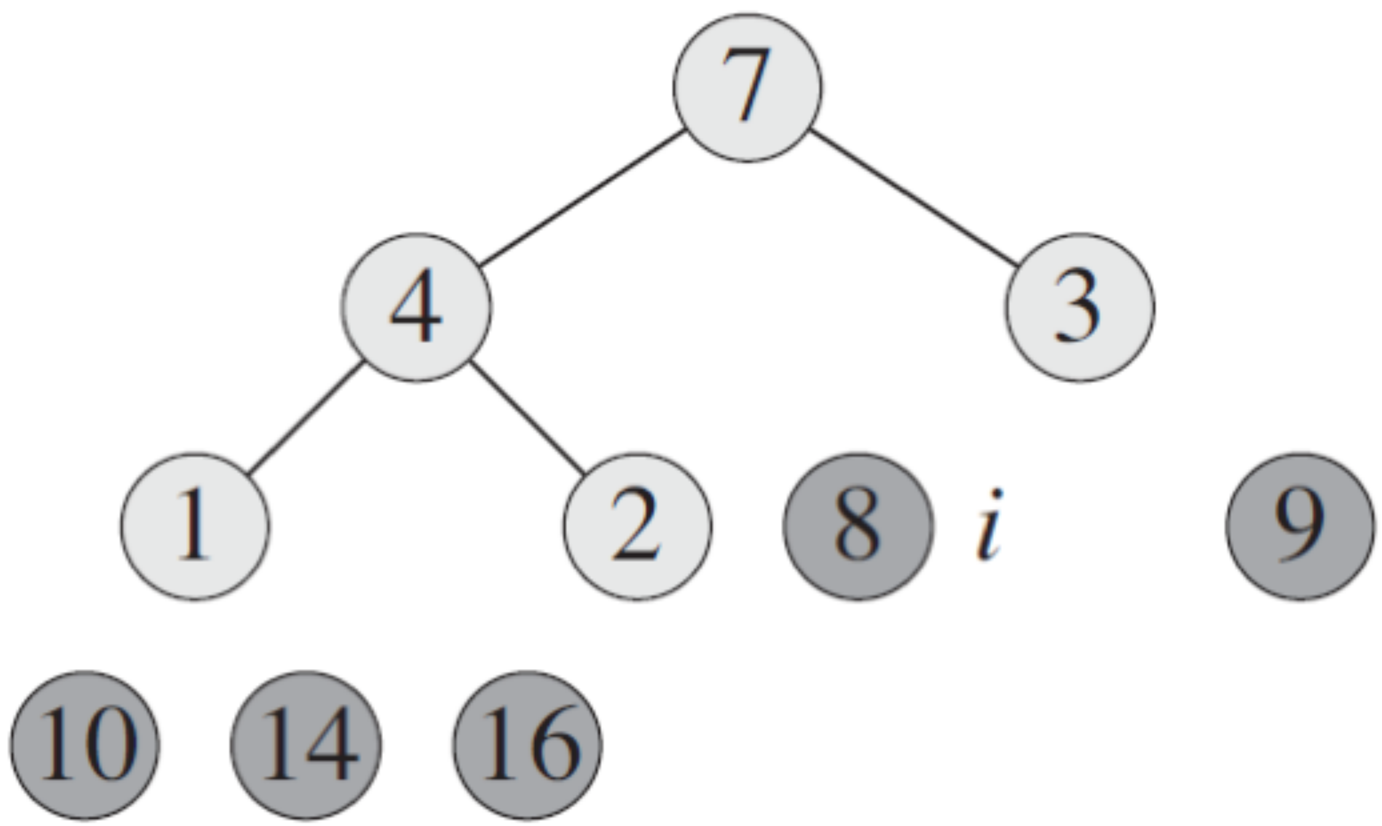
Heapsort



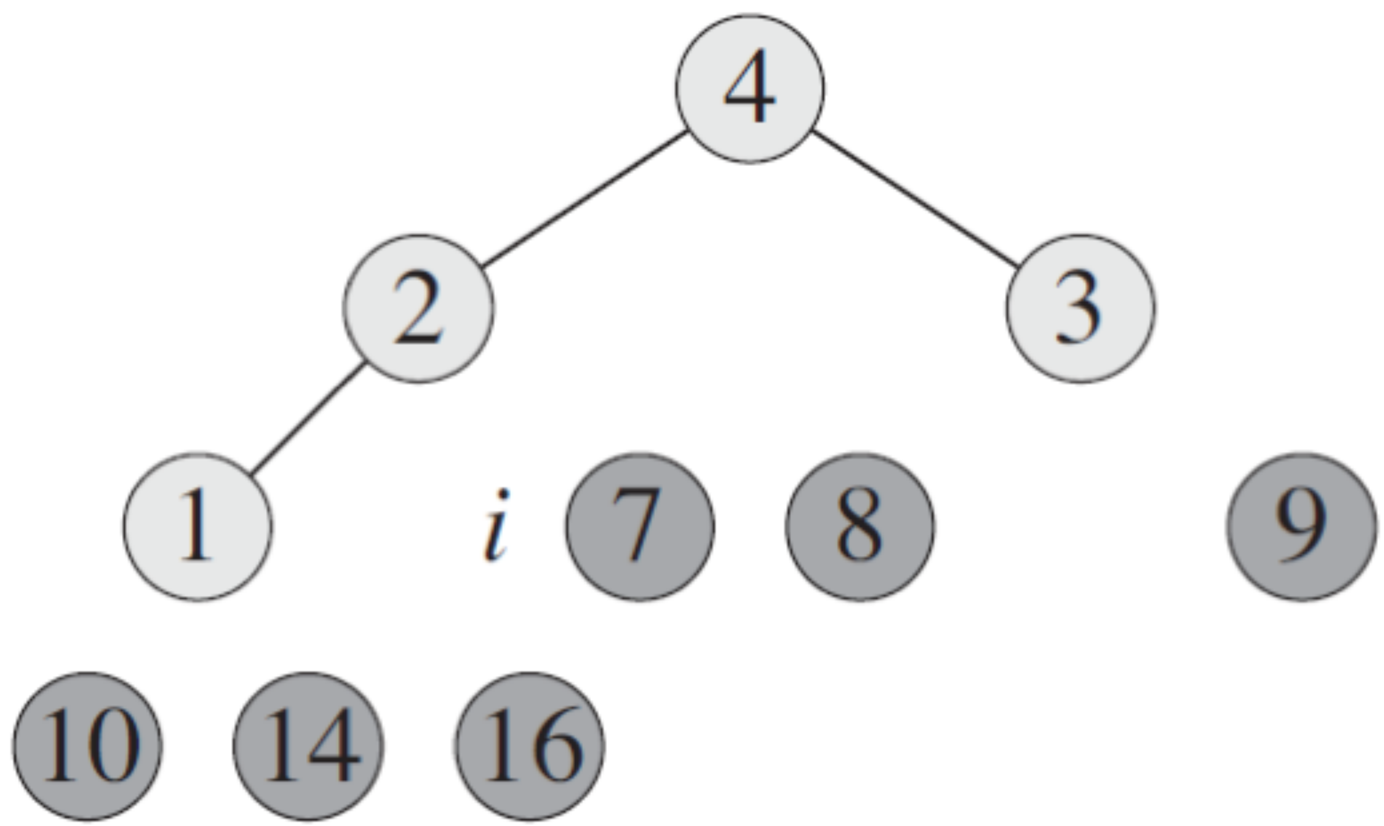
Heapsort



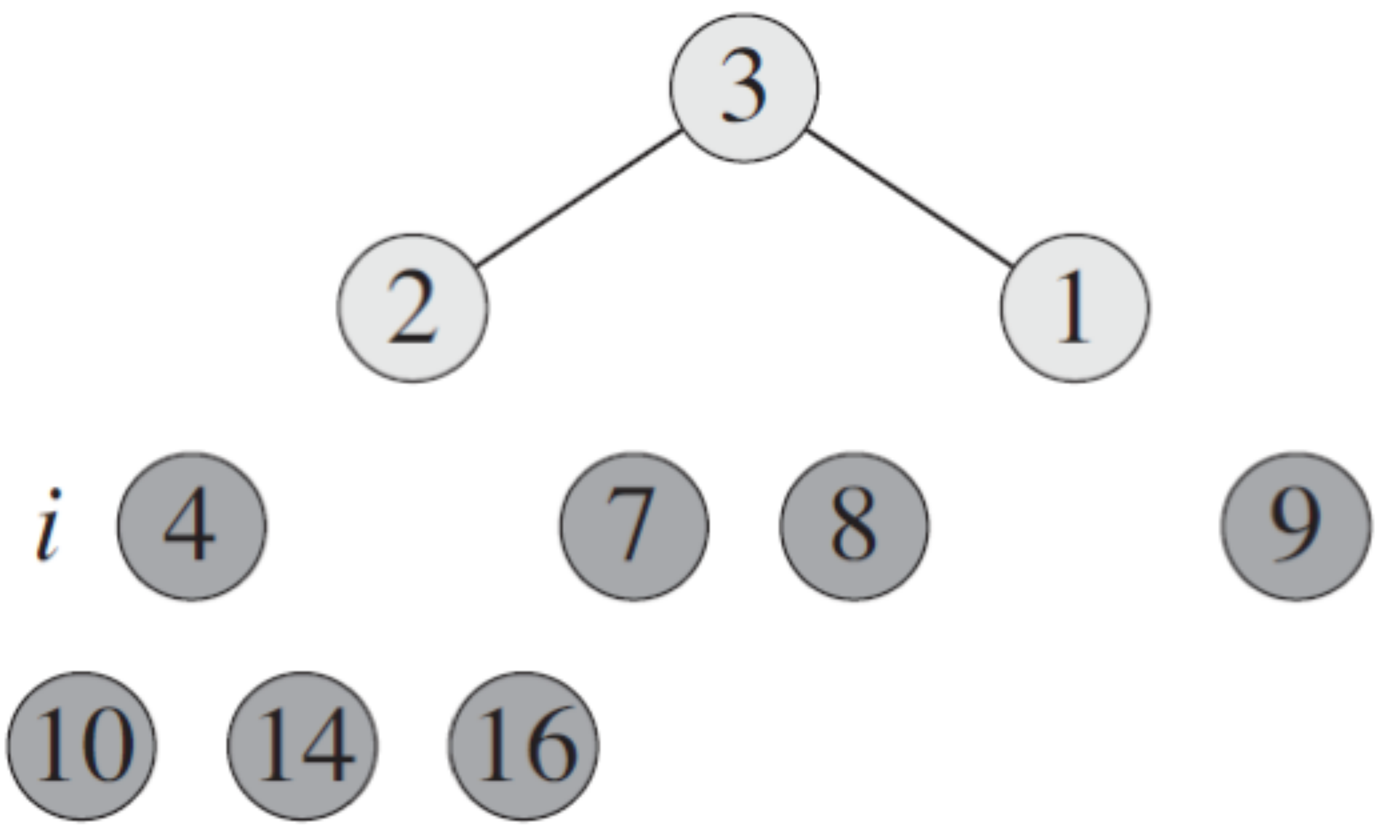
Heapsort



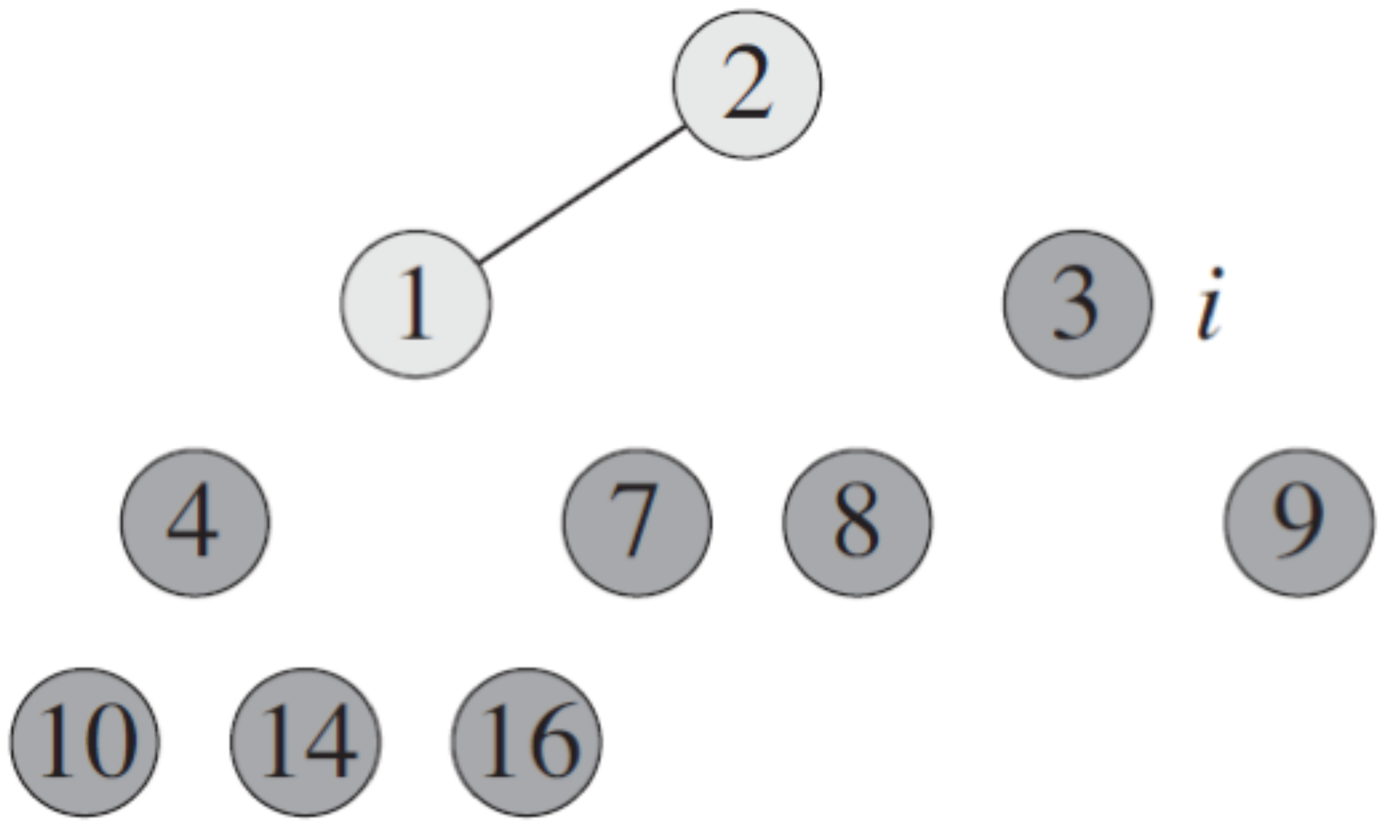
Heapsort



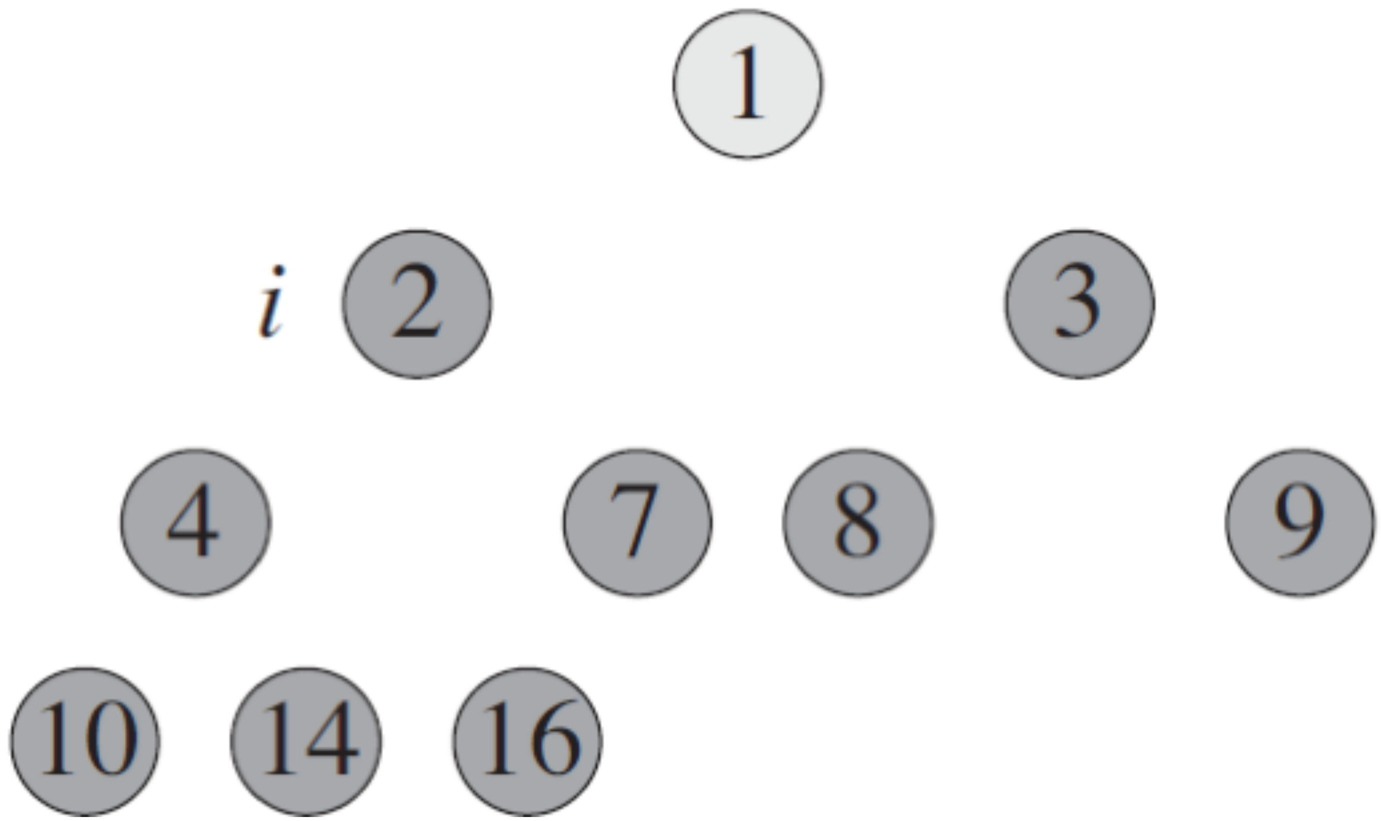
Heapsort



Heapsort



Heapsort



Runtime Analysis

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

- RuntimeCosts:
  O(n) + O(n lg n) = O(n lg n)
- MemoryCosts:
  O(1), i.e., in-situ sorting
- Visualization:
  http://www.sorting-algorithms.com/heap-sort

Heap as a data structure

- Heaps are a data structure that can be used for other purposes, as well.
- In particular, a max-heap is often used to build a max-priority queue.

Max-priority queues

Definition:

A priority queue is a data structure for maintaining a set S of elements, each with an associated value called a key.

Definition:

A max-priority queue is a priority queue that supports the following operations:

- Maximum(S): return element from S with largest key.
- Extract-Max(S): remove and return element from S with largest key.
- Increase-Key(S,x,k): increase the value of the key of element x to k, where k is assumed to be larger or equal than the current key.
- Insert(S,x): add element x to set S.

Maximum (S)

HEAP-MAXIMUM(A)

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

Costs: O(1)

Extract-Max (S)

```
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
```

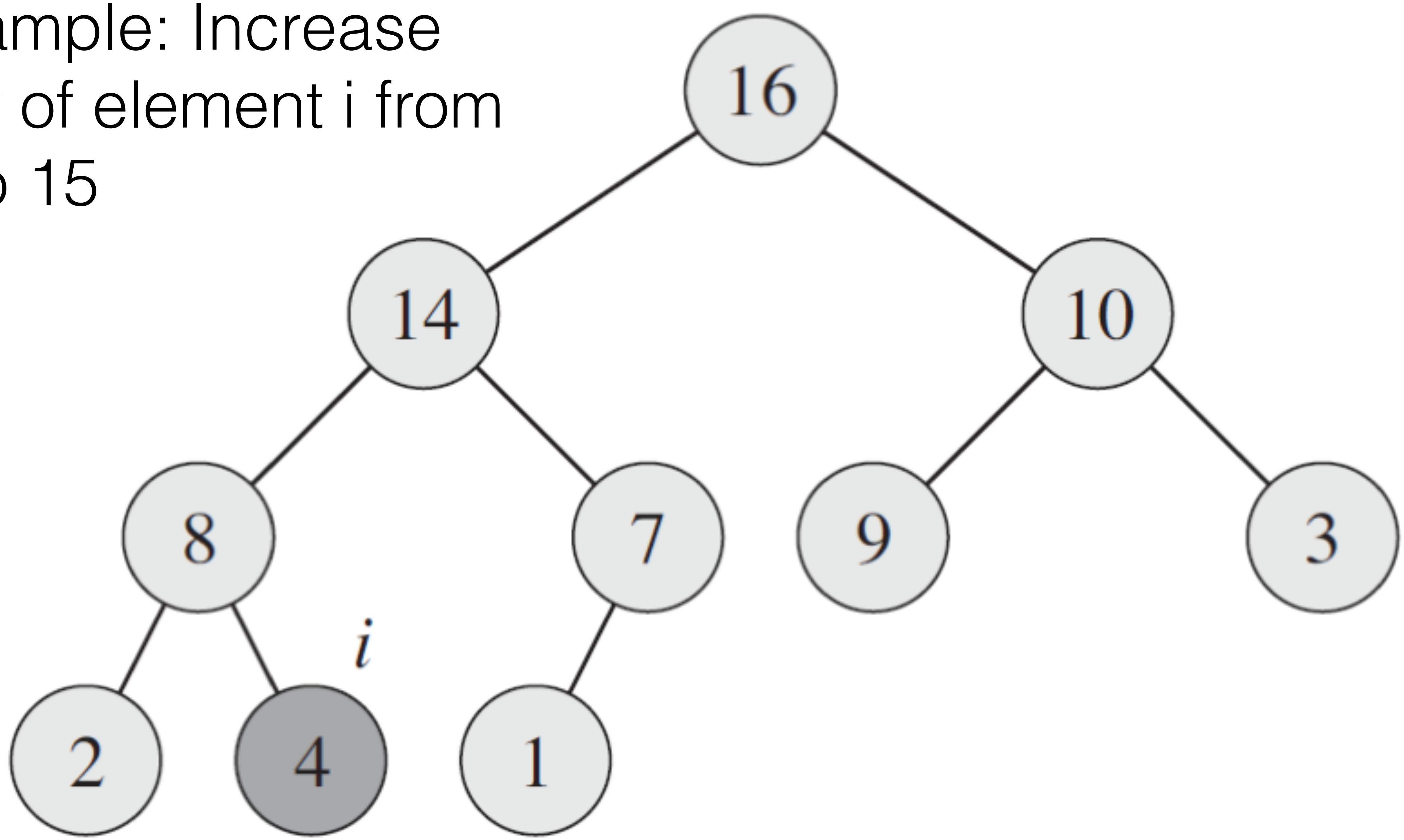
Costs: O(lg n)

Increase-Key (S,x,k)

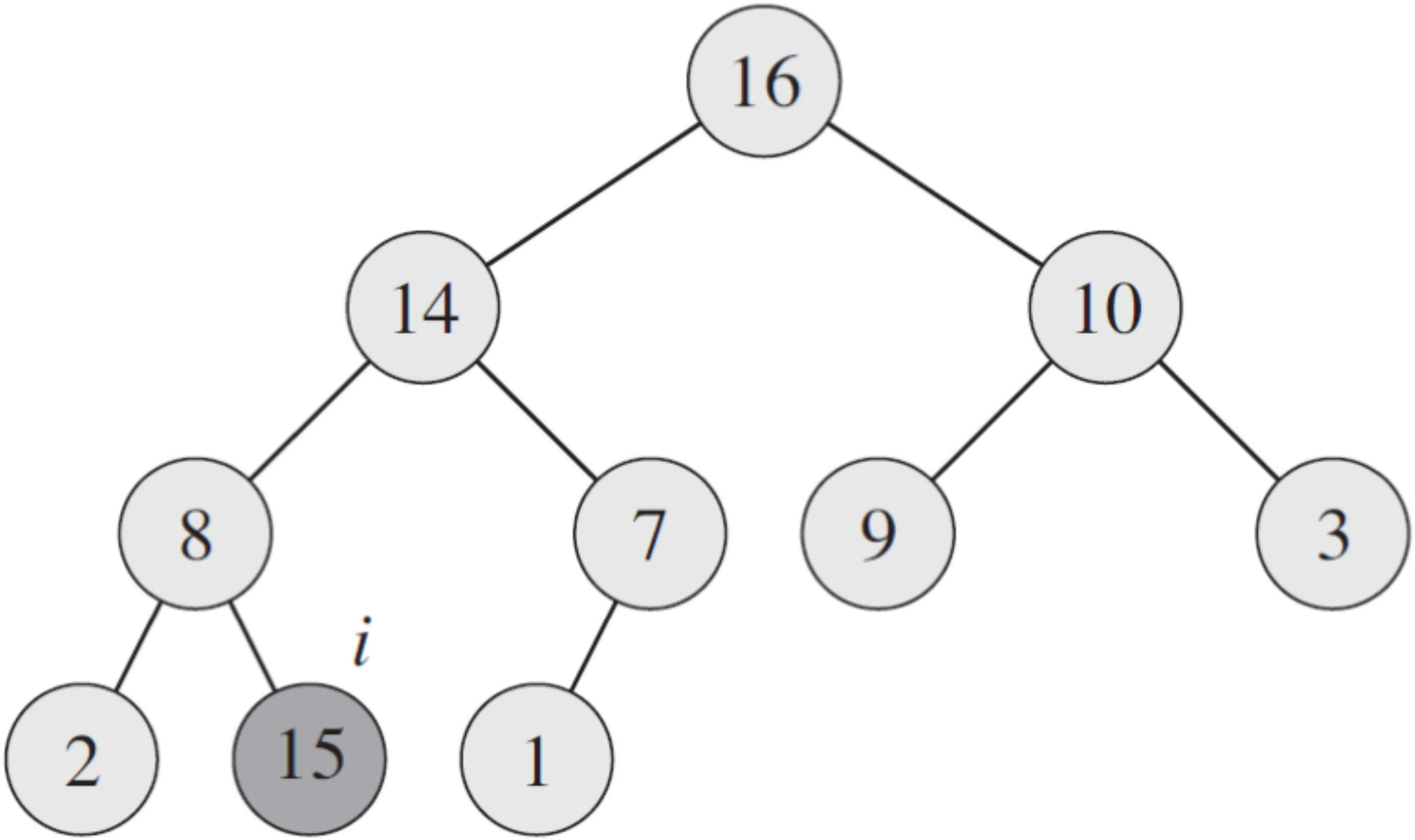
```
HEAP-INCREASE-KEY(A, i, key)
if key < A[i]
    error "new key is smaller than current key"
A[i] = key
while i > 1 and A[PARENT(i)] < A[i]
    exchange A[i] with A[PARENT(i)]
    i = PARENT(i)
```

Increase-Key (S,x,k)

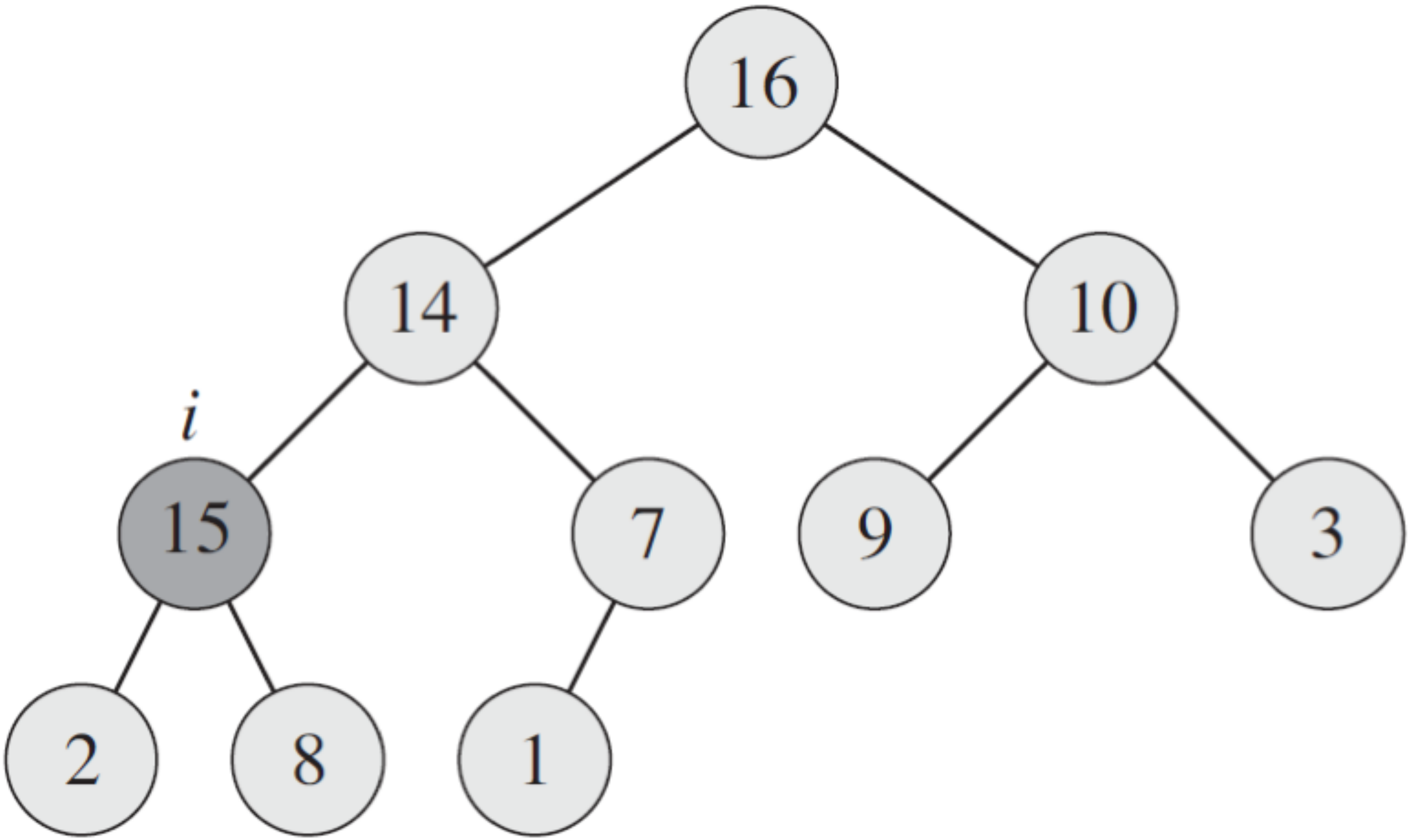
Example: Increase key of element i from 4 to 15



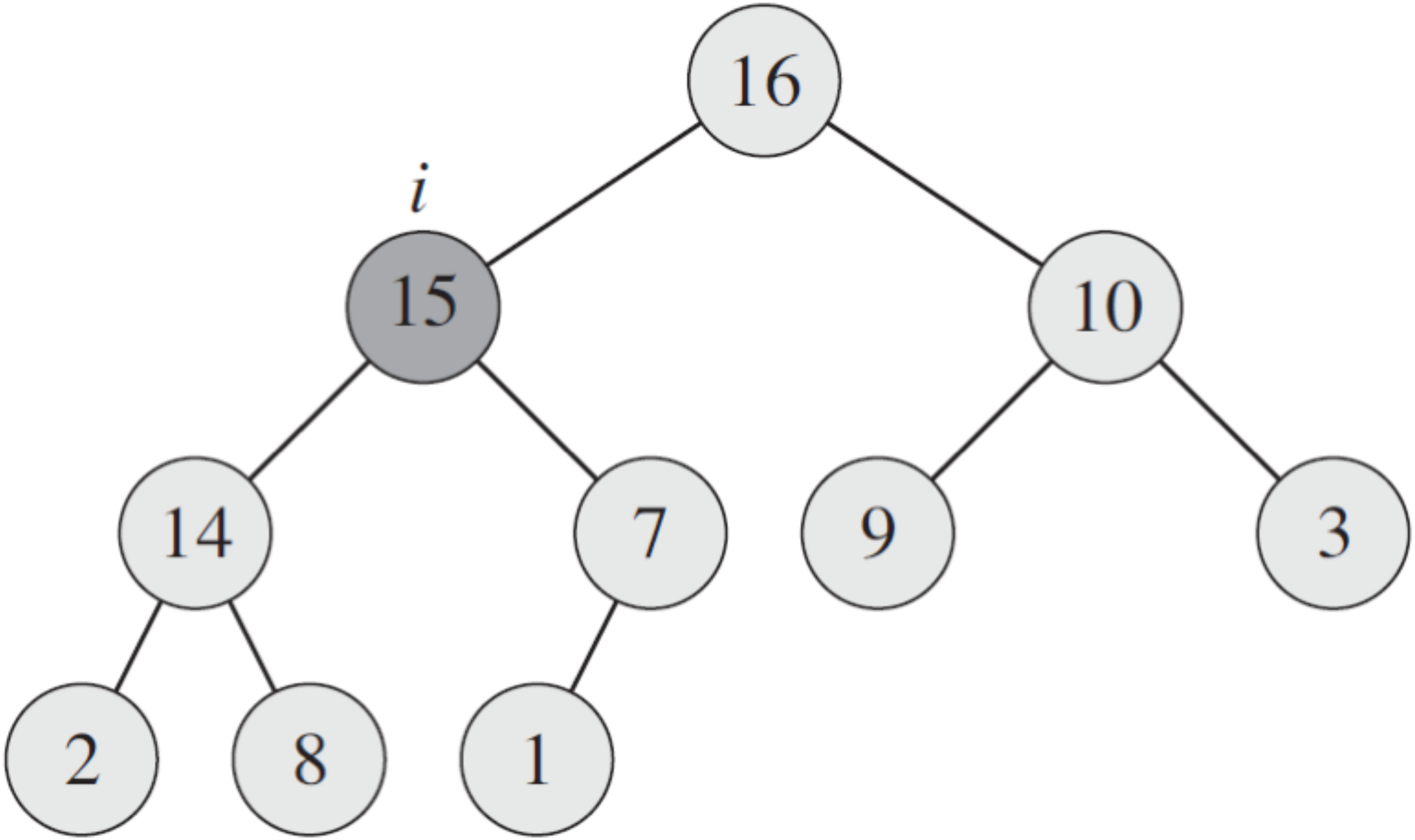
Increase-Key (S,x,k)



Increase-Key (S,x,k)



Increase-Key (S,x,k)



Increase-Key (S,x,k)

```
HEAP-INCREASE-KEY(A, i, key)
if key < A[i]
    error "new key is smaller than current key"
A[i] = key
while i > 1 and A[PARENT(i)] < A[i]
    exchange A[i] with A[PARENT(i)]
    i = PARENT(i)
```

Costs: O(lg n)

Insert (S,x)

MAX-HEAP-INSERT(A, key)

1 $A.heap\text{-}size = A.heap\text{-}size + 1$
2 $A[A.heap\text{-}size] = -\infty$
3 HEAP-INCREASE-KEY$(A, A.heap\text{-}size, key)$

Costs: O(lg n)