CS302 – Analysis and Design of Algorithms

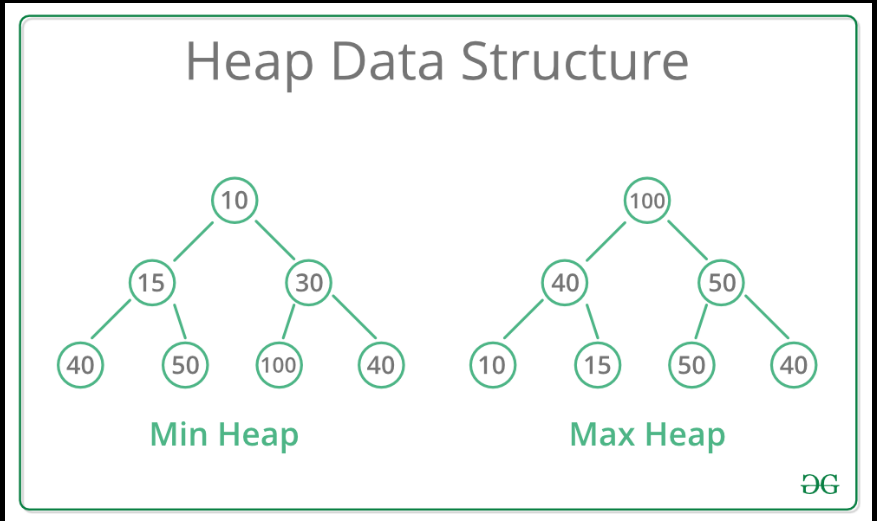
Heap Sort

Content

Content
Introduction
Maintaining the Heap Property
Building a Heap
The Heapsort Algorithm
Priority Queue
Applications
Exercises

Introduction

- **Heapsort:** a sorting algorithm based on the heap data structure.



- Has the advantages of merge-sort and insertion-sort:
  - Like merge-sort, runs in $O(n \lg n)$
  - Like insertion-sort, sorts in place.

Introduction

- The heap is an array object that we view as a <u>binary tree</u>.
  - Completely filled on all levels except possibly the lowest.

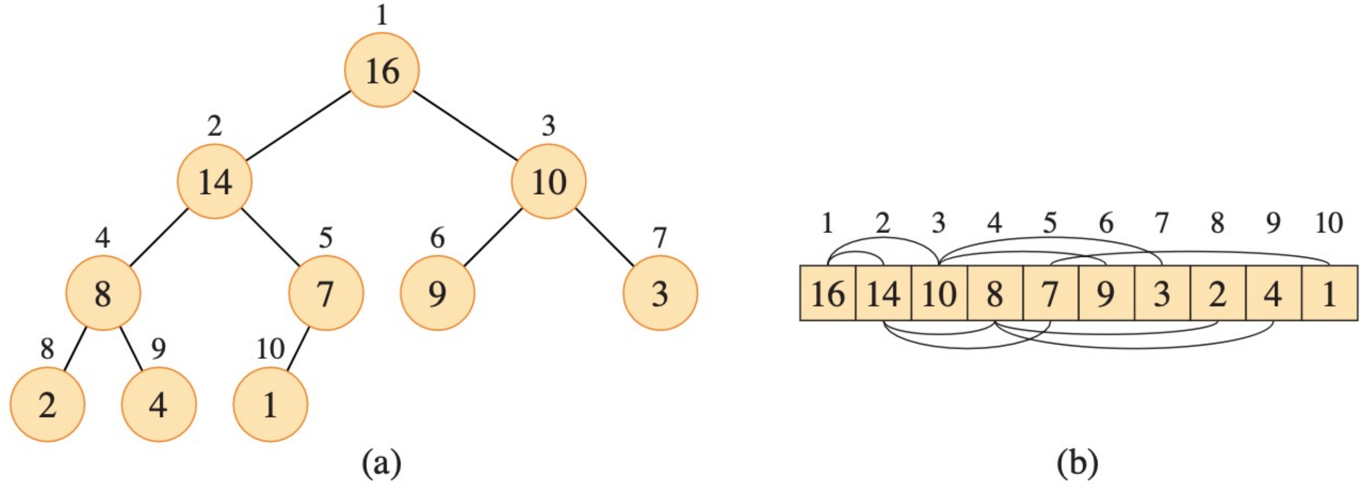


Figure 6.1 A max-heap viewed as **(a)** a binary tree and **(b)** an array. The number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships, with parents always to the left of their children. The tree has height 3, and the node at index 4 (with value 8) has height 1.

Introduction

- The heap:
  - Represented as an array $A[1{:}n]$
  - Its size is $A.heapSize$.
  - $1 \leq A.heapSize \leq n$ → not all the elements in the array are considered heap elements.
  - $A.heapSize = 0$ → empty heap
  - $A[1]$ → the root of the heap
  - Height of a **node** = number of edges on the longest path from the node to a leaf
  - Height of the heap = height from the root node
  - A heap of n elements is of height $\Theta(\lg n)$

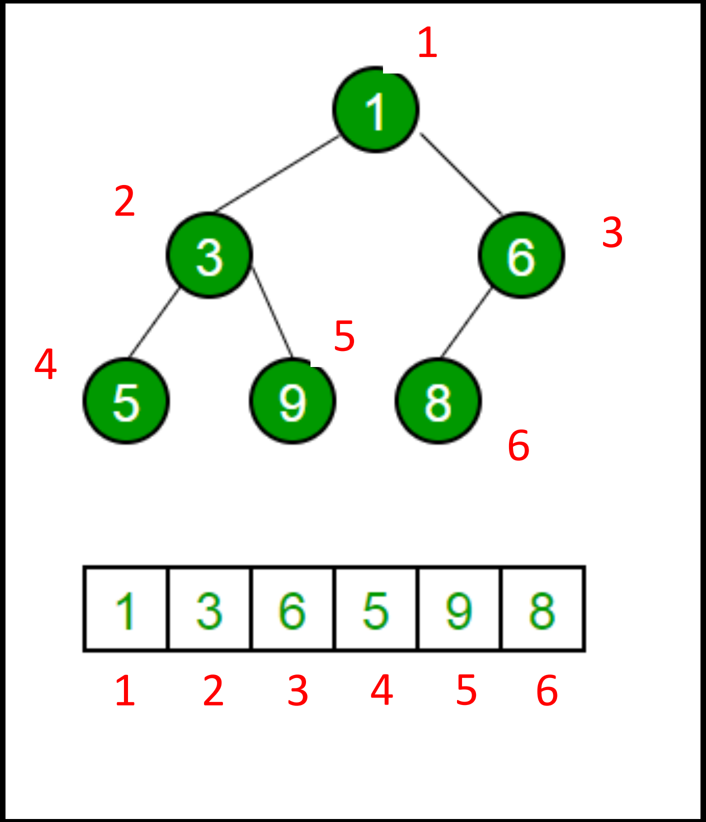
Introduction

- Given a node at index i, we can compute the index of the parent, left child, and right child.

PARENT(i)
return $\lfloor i/2 \rfloor$

LEFT(i)
return $2i$

RIGHT(i)
return $2i + 1$



Introduction

- Two kinds of heap

- Heaps
  - Max-heap
    - Property: $A[parent(i)] \geq A[i]$
    - The root is largest, its subtree is smaller than or equal to it
  - Min-heap
    - Property: $A[parent(i)] \leq A[i]$
    - The root is smallest, its subtree is larger than or equal to it

Commonly used for sorting

Implementing priority queues

Introduction

- Basic procedures for max-heap:

Procedure	Description	Time complexity
MAX-HEAPIFY	Maintaining the heap property.	$O(\lg n)$
BUILD-MAX-HEAP	Produces a max- heap from an unordered input array.	$O(n)$
HEAPSORT	Sorts an array in place.	$O(n \lg n)$

Content

Content
Introduction
Maintaining the Heap Property
Building a Heap
The Heapsort Algorithm
Priority Queue
Applications
Exercises

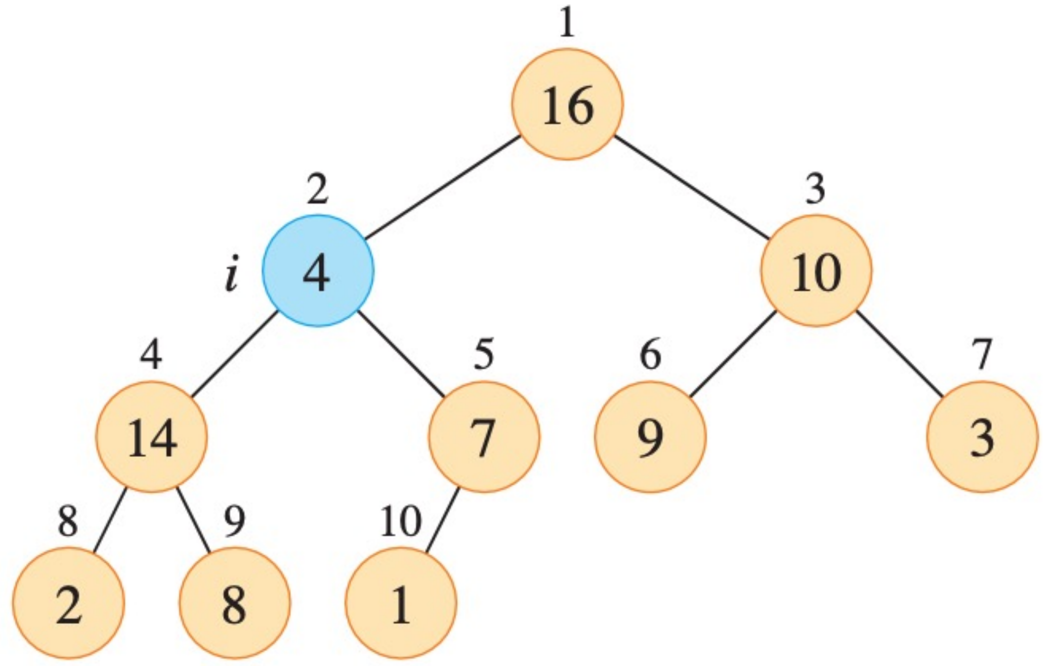
Maintaining the Heap Property

$MAX - HEAPIFY(A, i)$:

- A procedure that takes an array A and an index i and ensures that subtree rooted at index i obeys the max-heap property.
  - i.e., $A[i] \geq Right(i)$ and $A[i] \geq Left(i)$

Maintaining the Heap Property

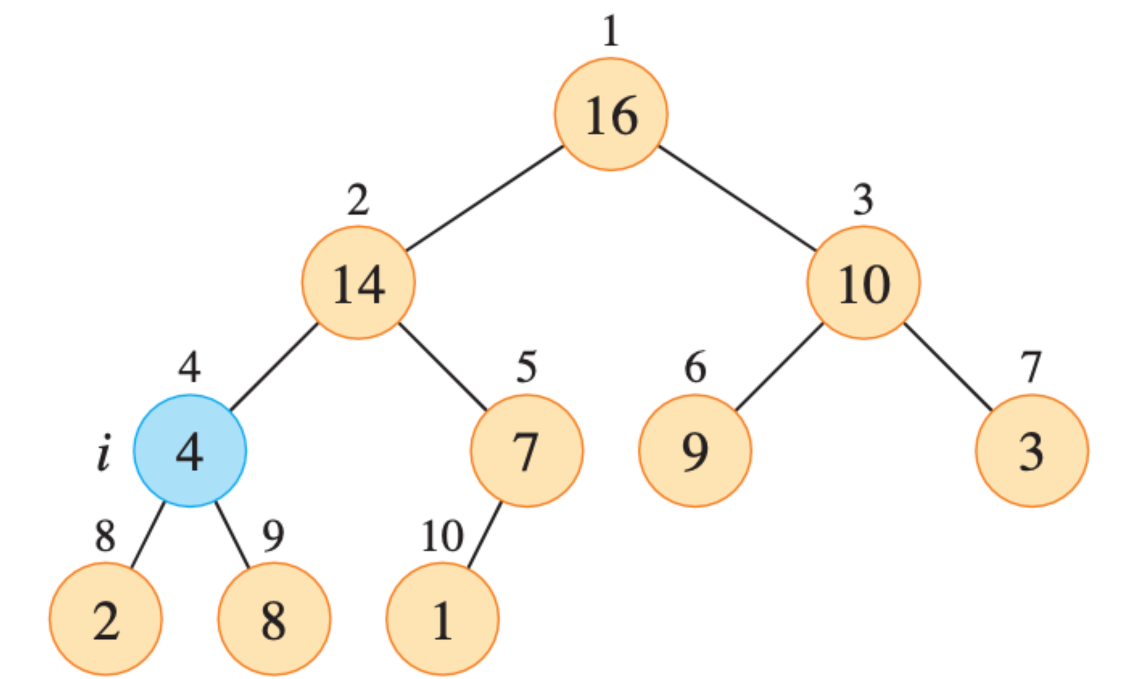
$MAX - HEAPIFY(A, 2)$:



(a)

Maintaining the Heap Property

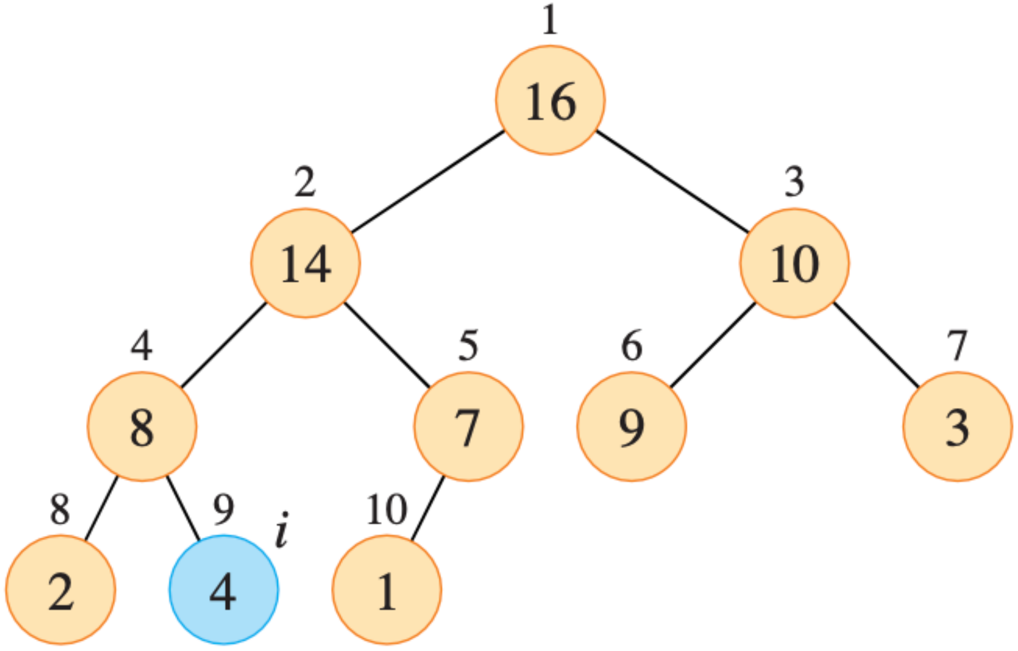
$MAX - HEAPIFY(A, 4)$:



(b)

Maintaining the Heap Property

$MAX - HEAPIFY(A, 9)$: no further change to the data structure.



(c)

Maintaining the Heap Property

- Algorithm

```
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)
```

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

Content

Building a Heap

$BUILD - MAX - HEAP(A, n)$

- Converts an array $A[1{:}\,n]$ into a max-heap by calling MAX-HEAPIFY in a bottom-up manner.

```
BUILD-MAX-HEAP(A, n)
  A.heap-size = n
  for i = ⌊n/2⌋ downto 1
      MAX-HEAPIFY(A, i)
```

- $i = \lfloor n/2 \rfloor$ because the indices $i + 1, i + 2, \ldots, n$ are leaf nodes.

Building a Heap

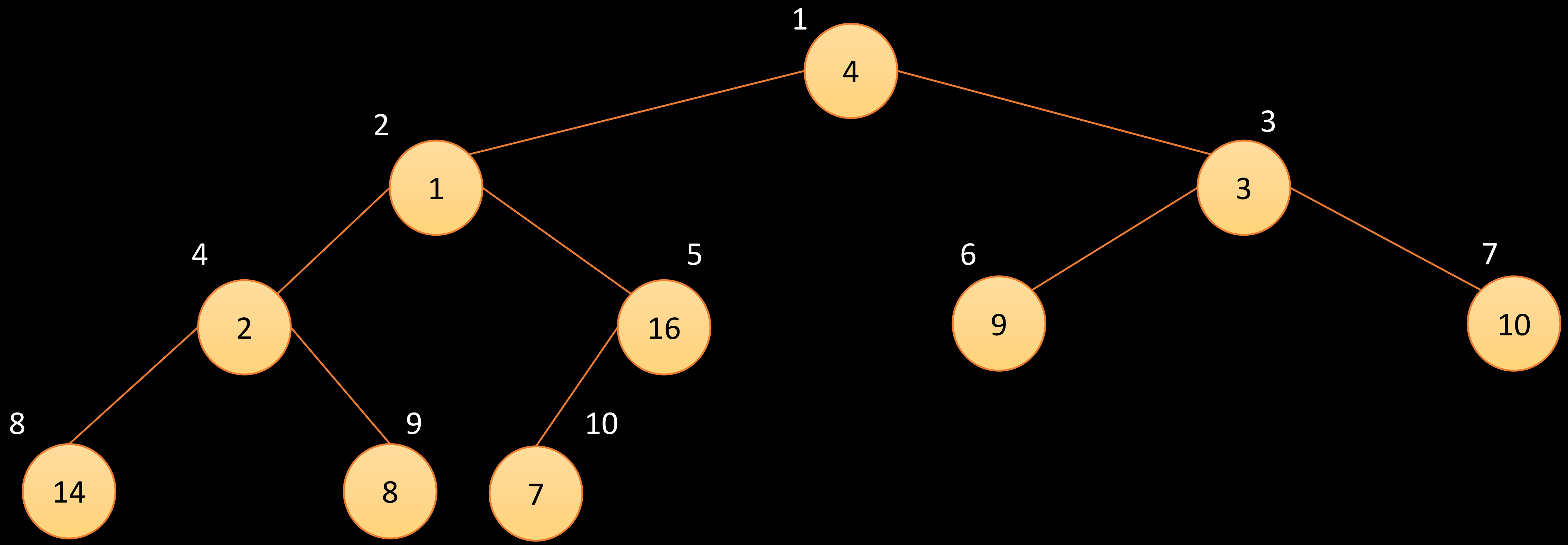
$BUILD-MAX-HEAP(A, 10)$

A	4	1	3	2	16	9	10	14	8	7

Building a Heap

$BUILD - MAX - HEAP(A, 10)$

A	4	1	3	2	16	9	10	14	8	7

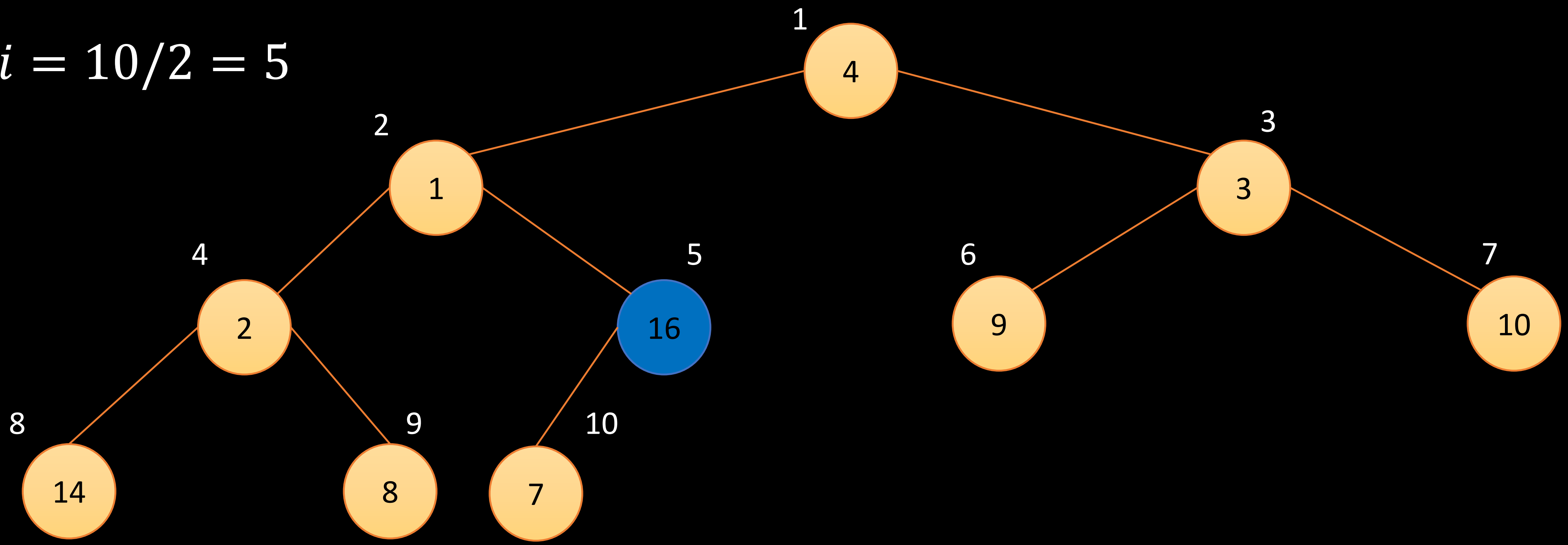


Building a Heap

$BUILD-MAX-HEAP(A, 10)$

A	4	1	3	2	16	9	10	14	8	7

- $i = 10/2 = 5$

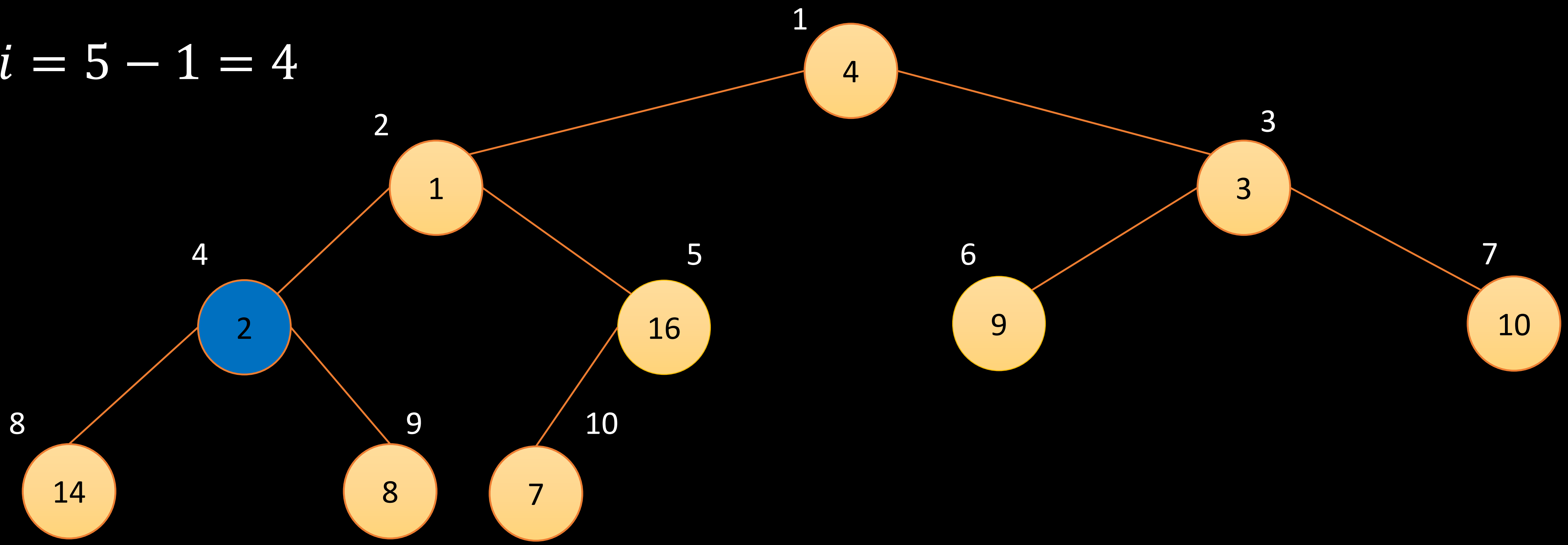


Building a Heap

$BUILD - MAX - HEAP(A, 10)$

A										
	4	1	3	2	16	9	10	14	8	7

- $i = 5 - 1 = 4$

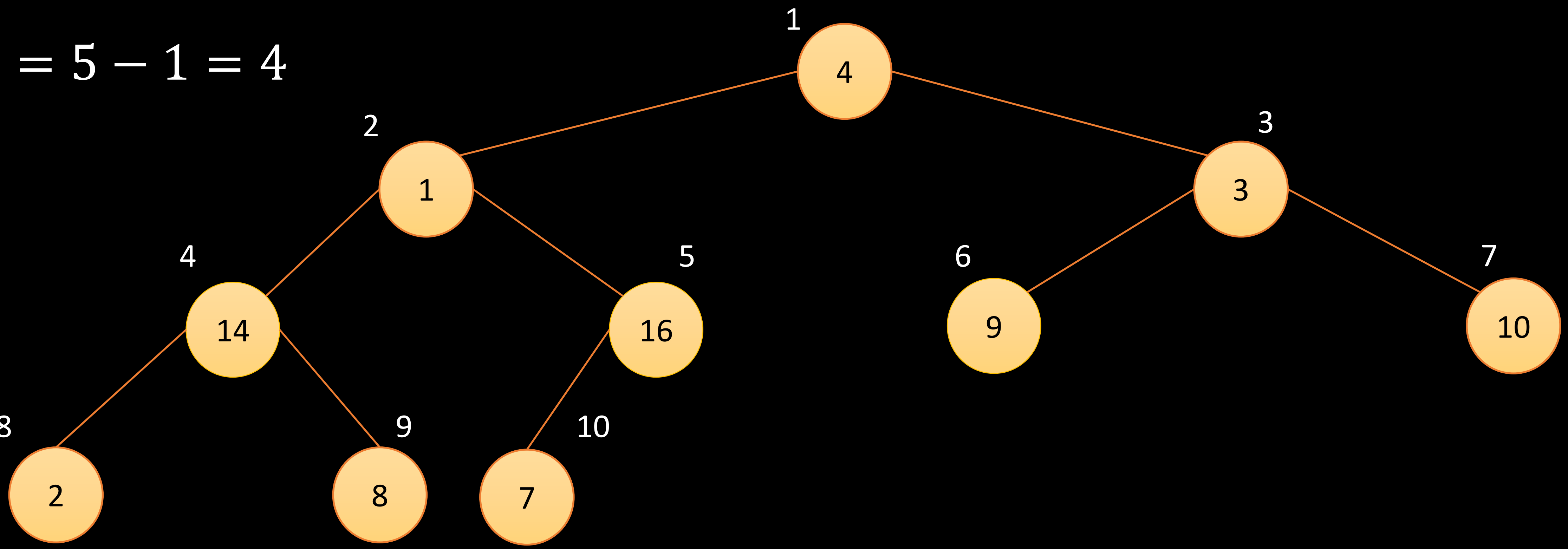


Building a Heap

$BUILD - MAX - HEAP(A, 10)$

A										
	4	1	3	2	16	9	10	14	8	7

- $i = 5 - 1 = 4$

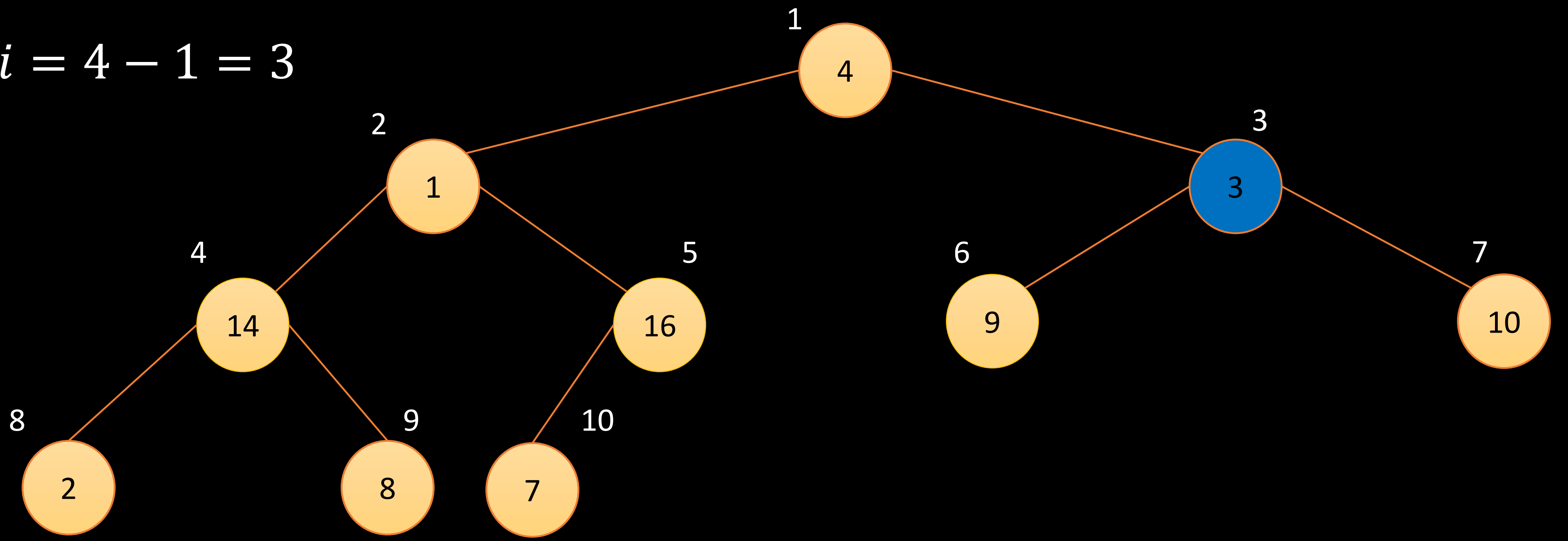


Building a Heap

$BUILD - MAX - HEAP(A, 10)$

A	4	1	3	2	16	9	10	14	8	7

- $i = 4 - 1 = 3$

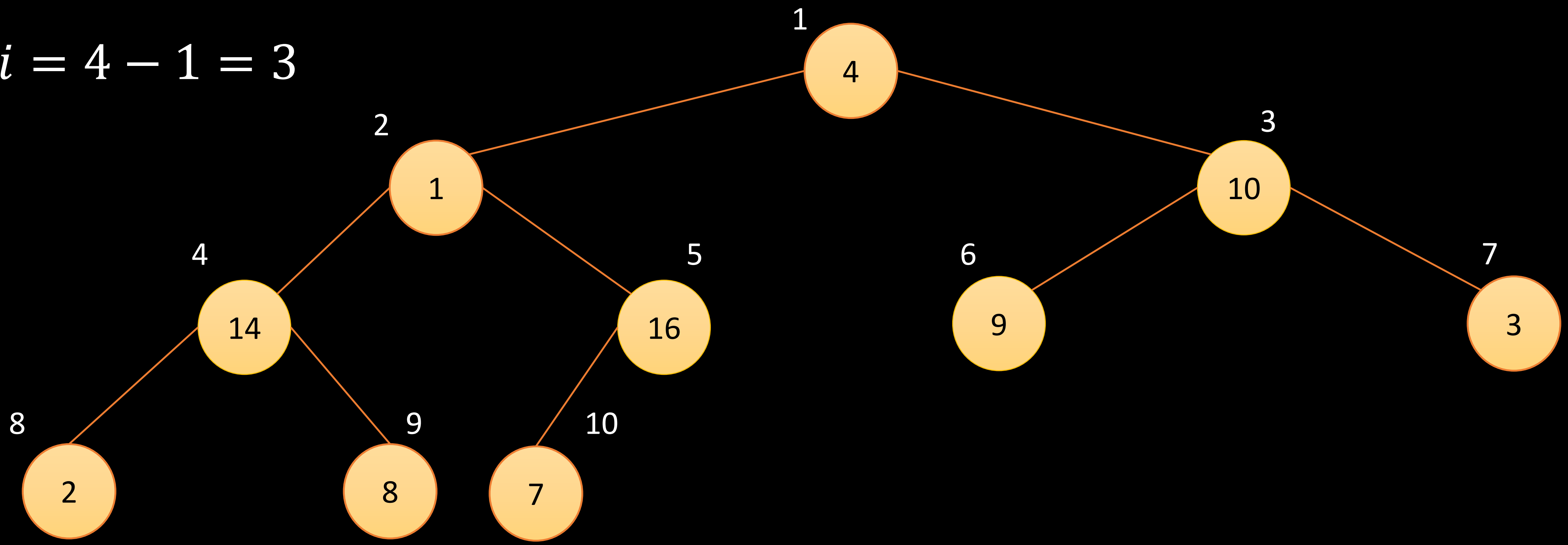


Building a Heap

$BUILD - MAX - HEAP(A, 10)$

A	4	1	3	2	16	9	10	14	8	7

- $i = 4 - 1 = 3$

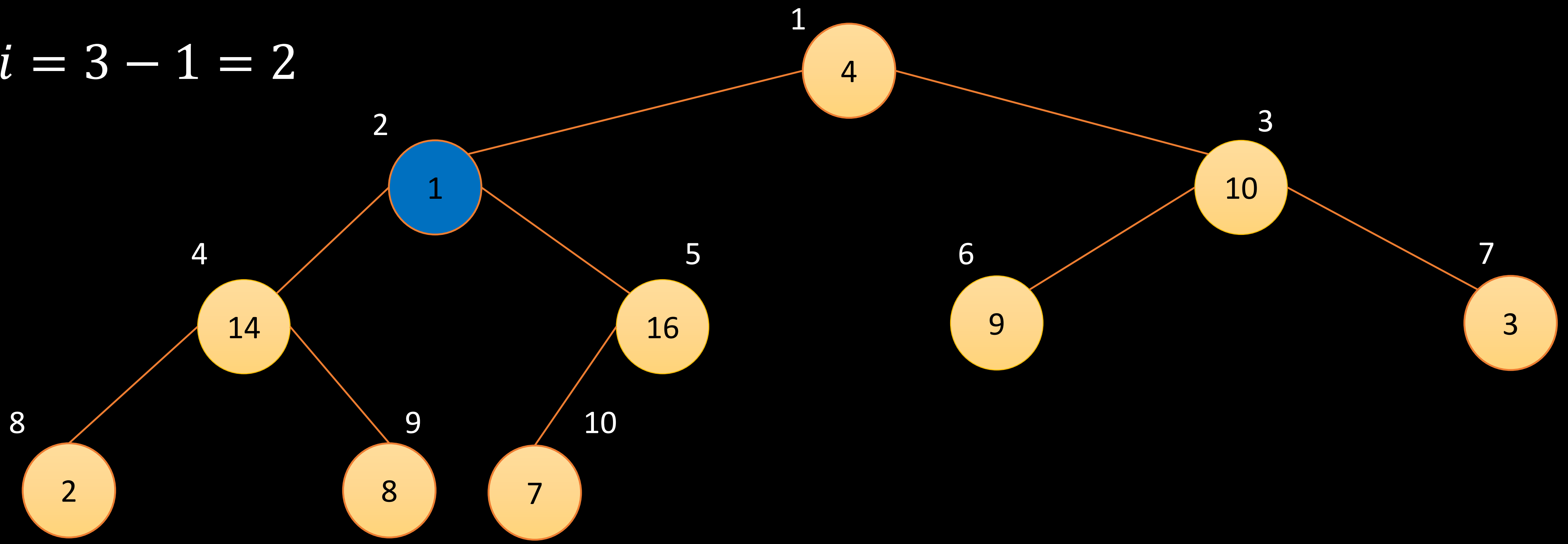


Building a Heap

$BUILD - MAX - HEAP(A, 10)$

A	4	1	3	2	16	9	10	14	8	7

- $i = 3 - 1 = 2$

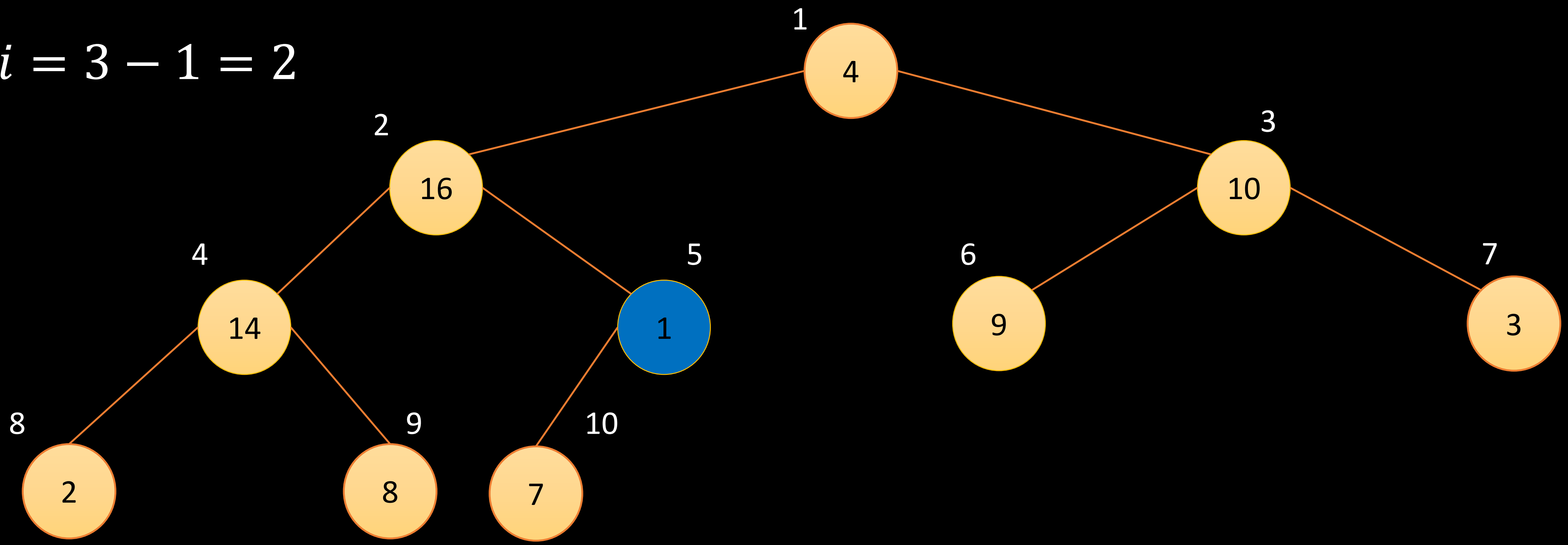


Building a Heap

$BUILD - MAX - HEAP(A, 10)$

A	4	1	3	2	16	9	10	14	8	7

- $i = 3 - 1 = 2$

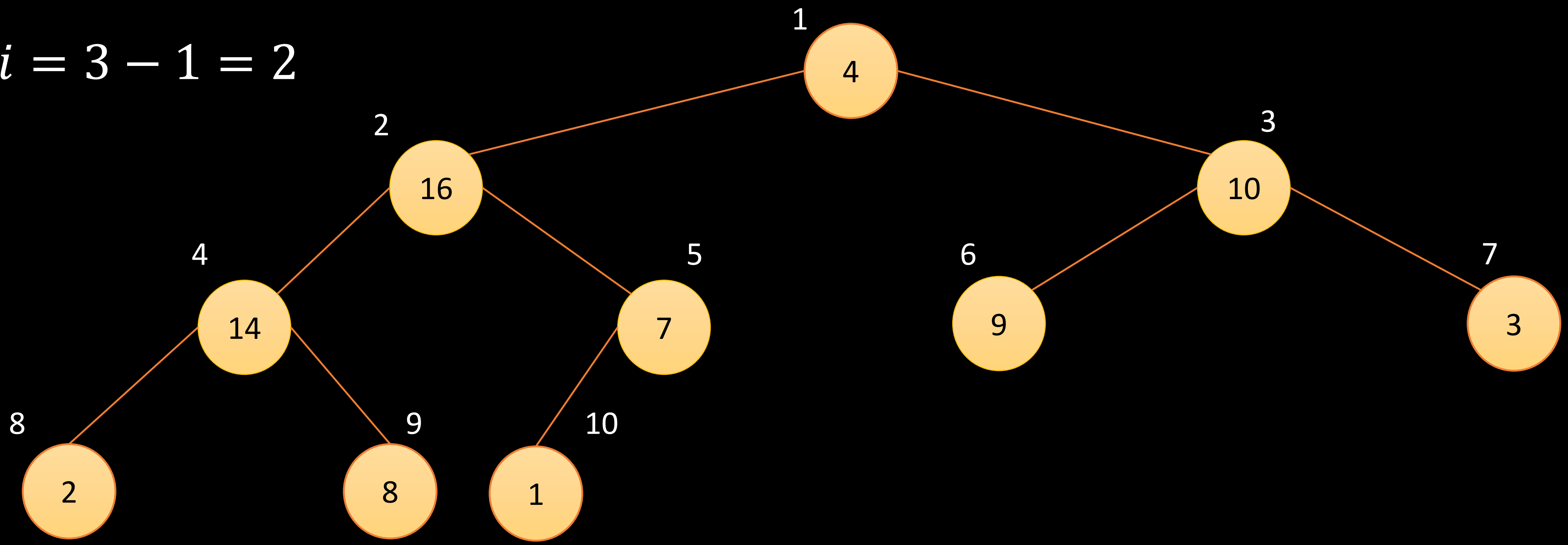


Building a Heap

$BUILD - MAX - HEAP(A, 10)$

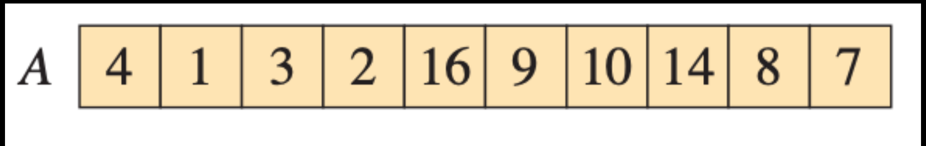
A	4	1	3	2	16	9	10	14	8	7

- $i = 3 - 1 = 2$

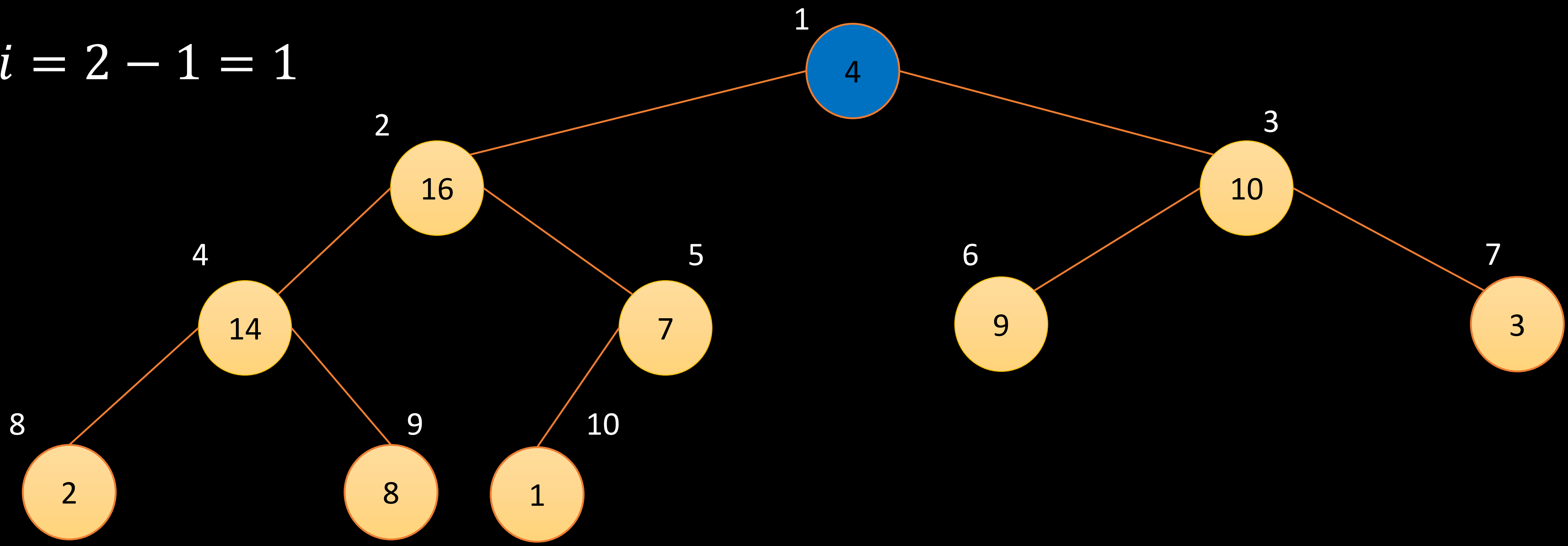


Building a Heap

$BUILD - MAX - HEAP(A, 10)$



- $i = 2 - 1 = 1$

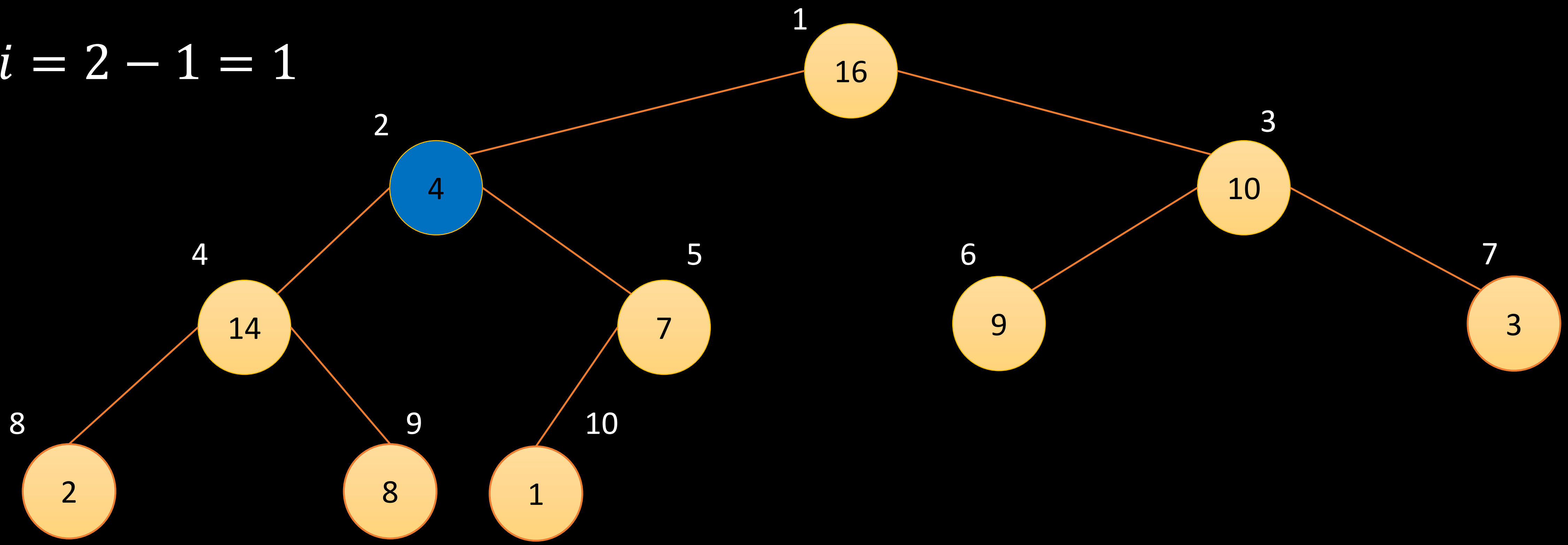


Building a Heap

$BUILD - MAX - HEAP(A, 10)$

A	4	1	3	2	16	9	10	14	8	7

- $i = 2 - 1 = 1$

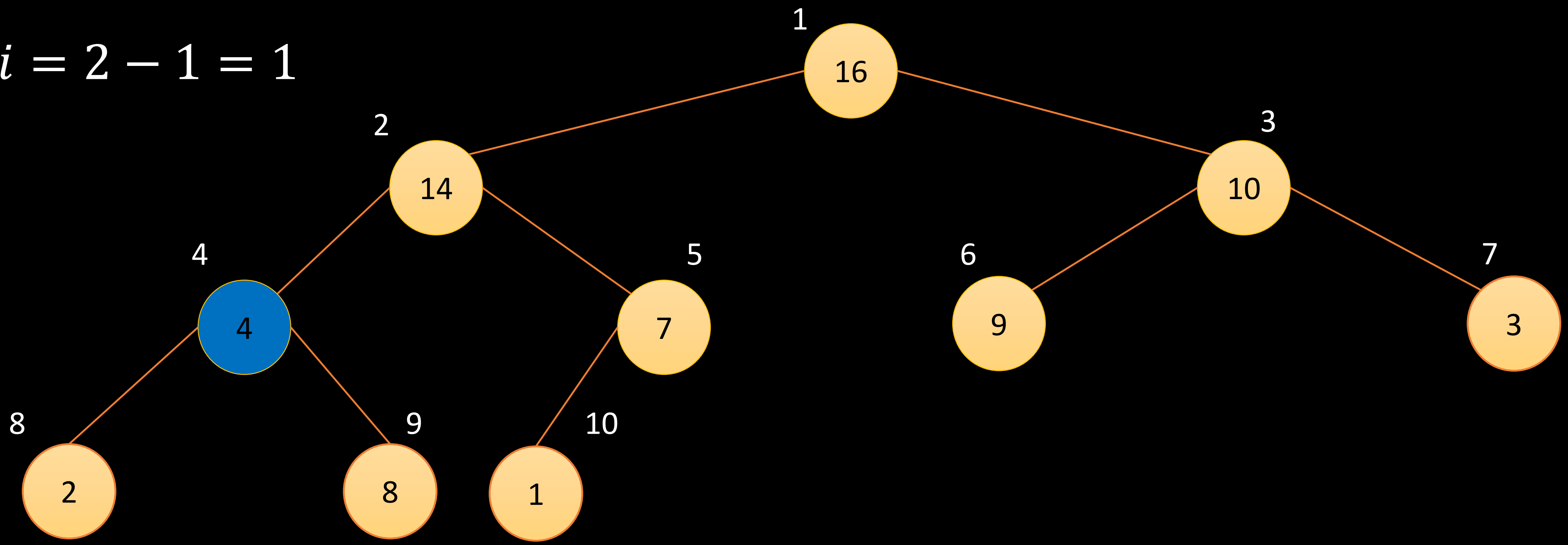


Building a Heap

$BUILD-MAX-HEAP(A,10)$

A	4	1	3	2	16	9	10	14	8	7

- $i = 2 - 1 = 1$

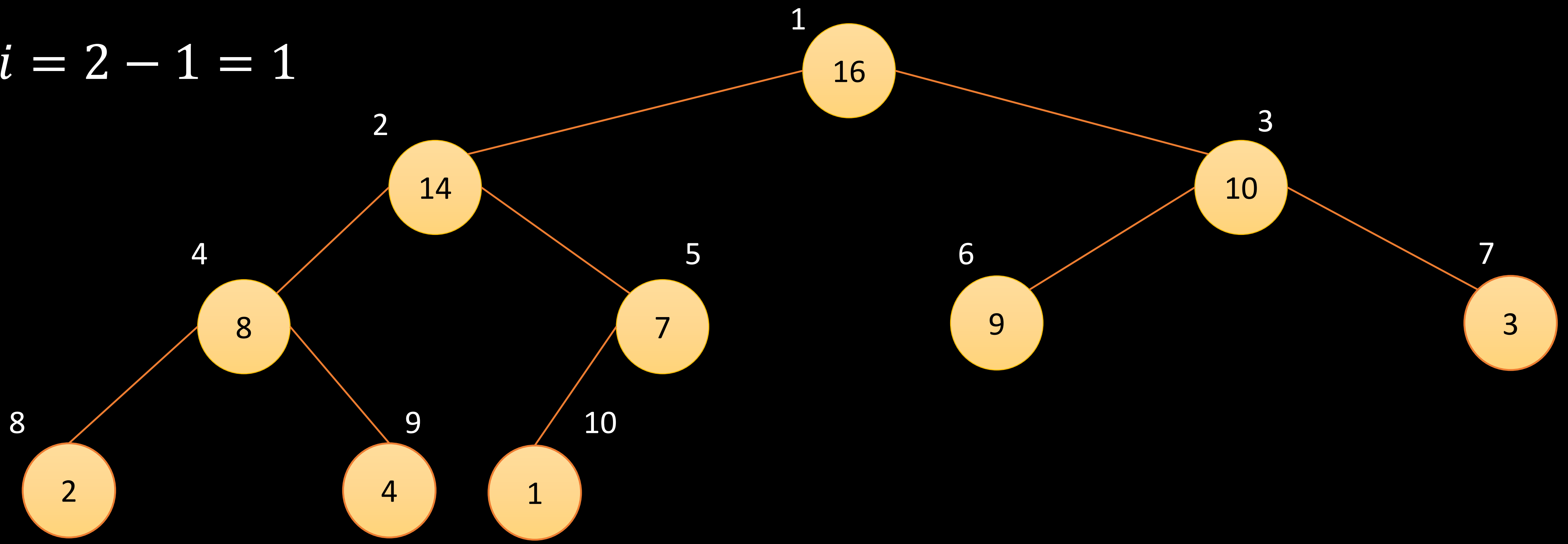


Building a Heap

$BUILD - MAX - HEAP(A, 10)$

A	4	1	3	2	16	9	10	14	8	7

- $i = 2 - 1 = 1$



Building a Heap

- Each call to MAX-HEAPIFY costs $O(\lg n)$ time.

- BUILD- MAX-HEAP makes $O(n)$ such calls.

- Thus, the running time (upper bound) is $O(n \lg n)$.

Content

The Heapsort Algorithm

$HEAP - SORT(A, n)$:

- Starts by calling BUILD-MAX-HEAP to build a max-heap on the array $A[1: n]$.
- At the end of BUILD-MAX-HEAP, the maximum element is at the root $A[1]$.
- Thus, HEAPSORT can place it into its correct final position by exchanging it with $A[n]$.
- The procedure then discards the last element by decrementing $A.HeapSize$.
- To restore the max-heap, the procedure calls $MAX - HEAPIFY(A, 1)$, which leaves a max-heap in $A[1: n - 1]$.

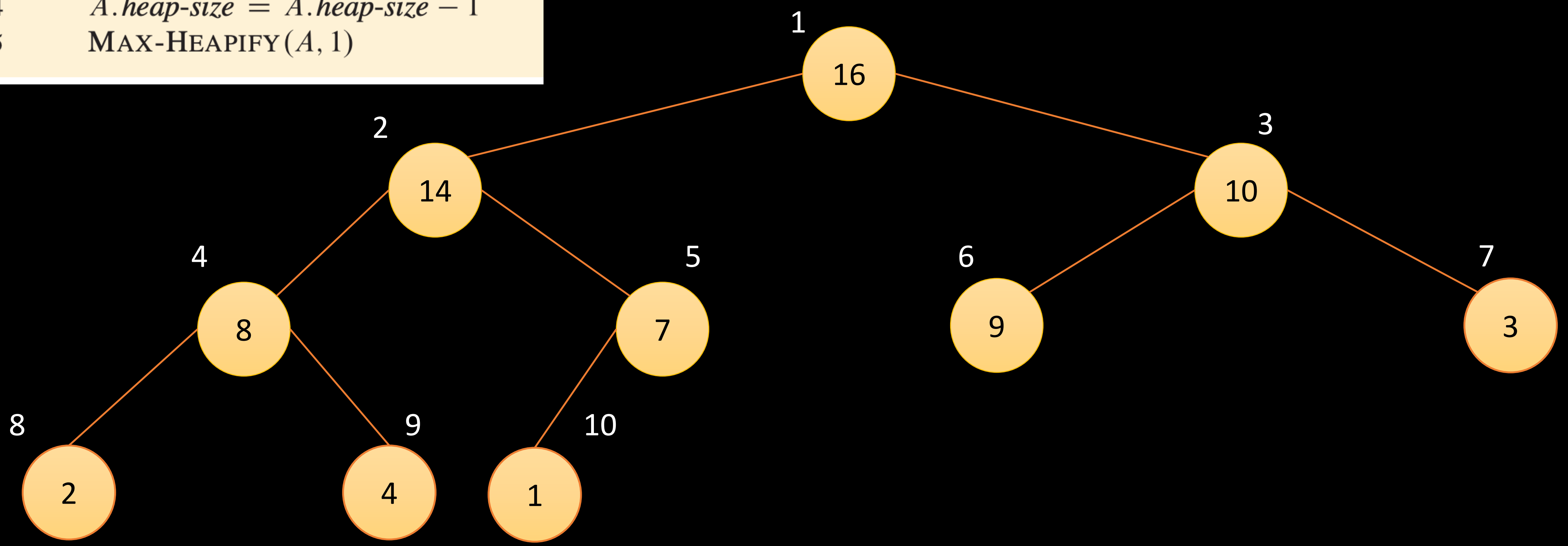
The Heapsort Algorithm

$HEAP - SORT(A, n)$:

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

The Heapsort Algorithm

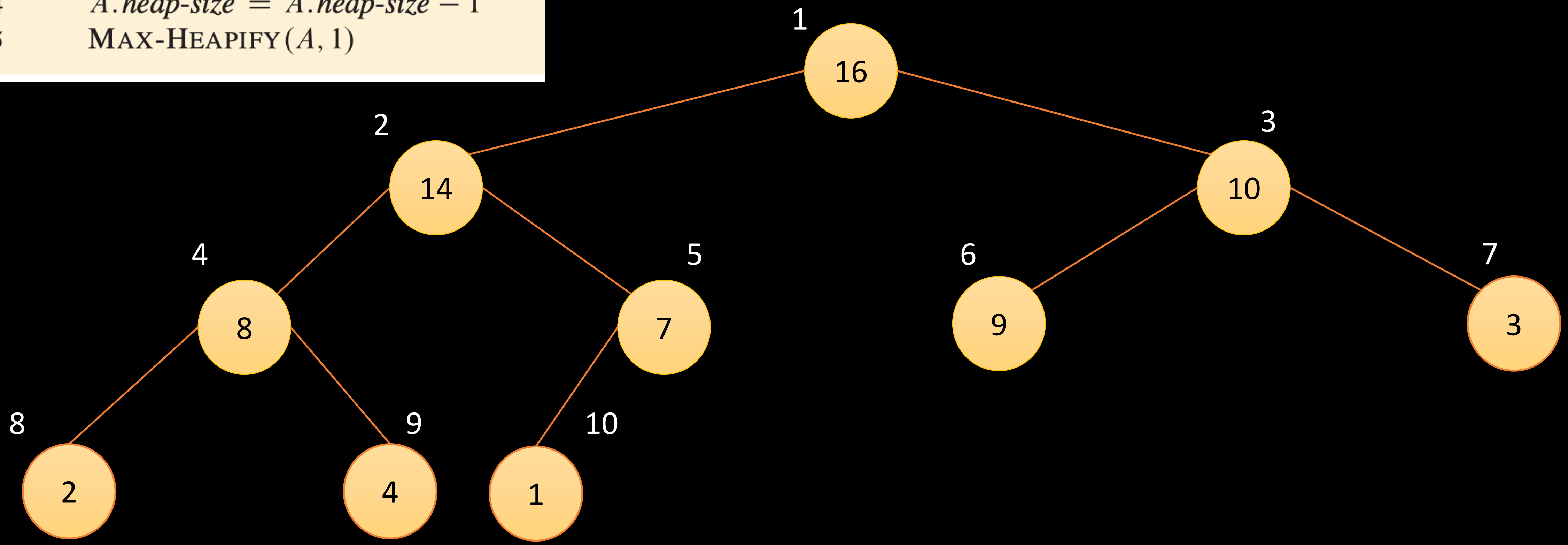
```
HEAPSORT(A, n)
BUILD-MAX-HEAP(A, n)
for i = n downto 2
    exchange A[1] with A[i]
    A.heap-size = A.heap-size − 1
    MAX-HEAPIFY(A, 1)
```



The Heapsort Algorithm

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

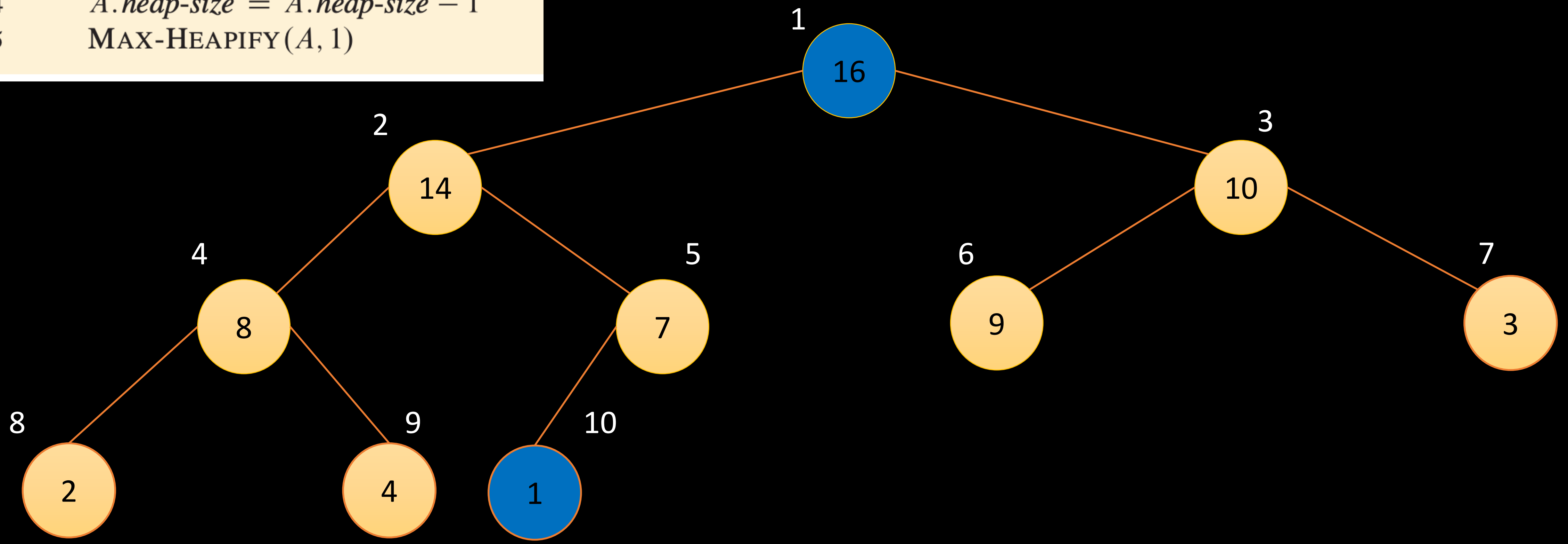
$$i = n = 10$$



The Heapsort Algorithm

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

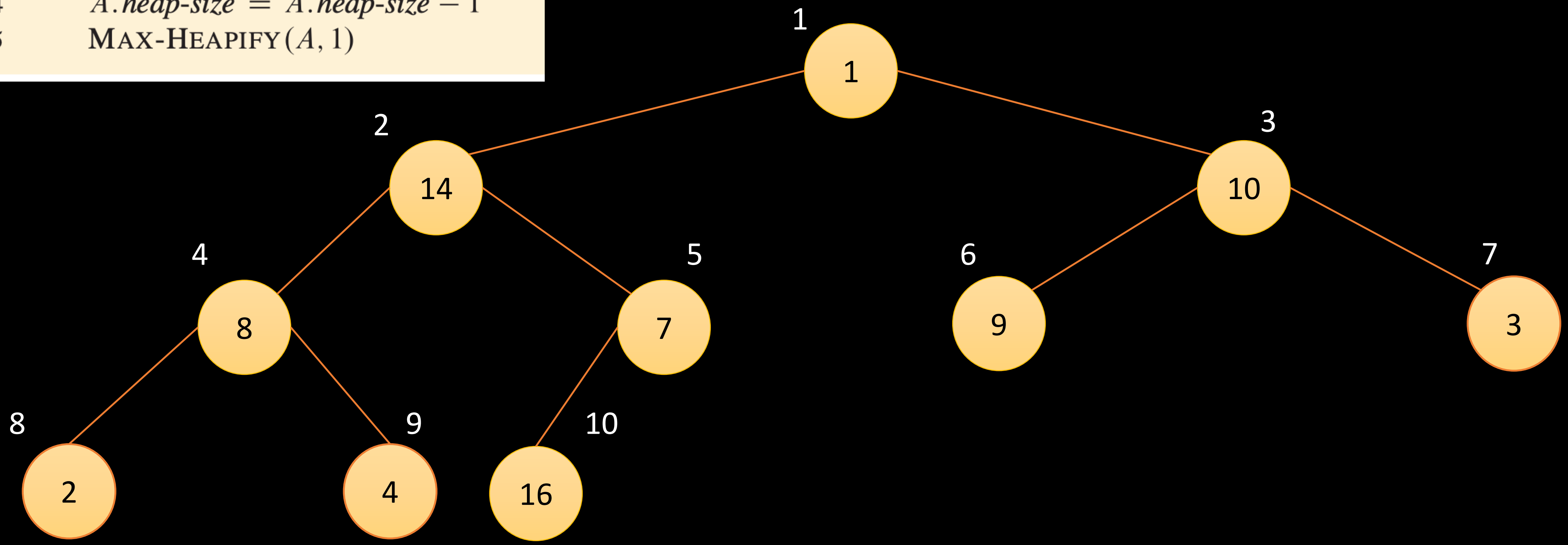
$i = n = 10$



The Heapsort Algorithm

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

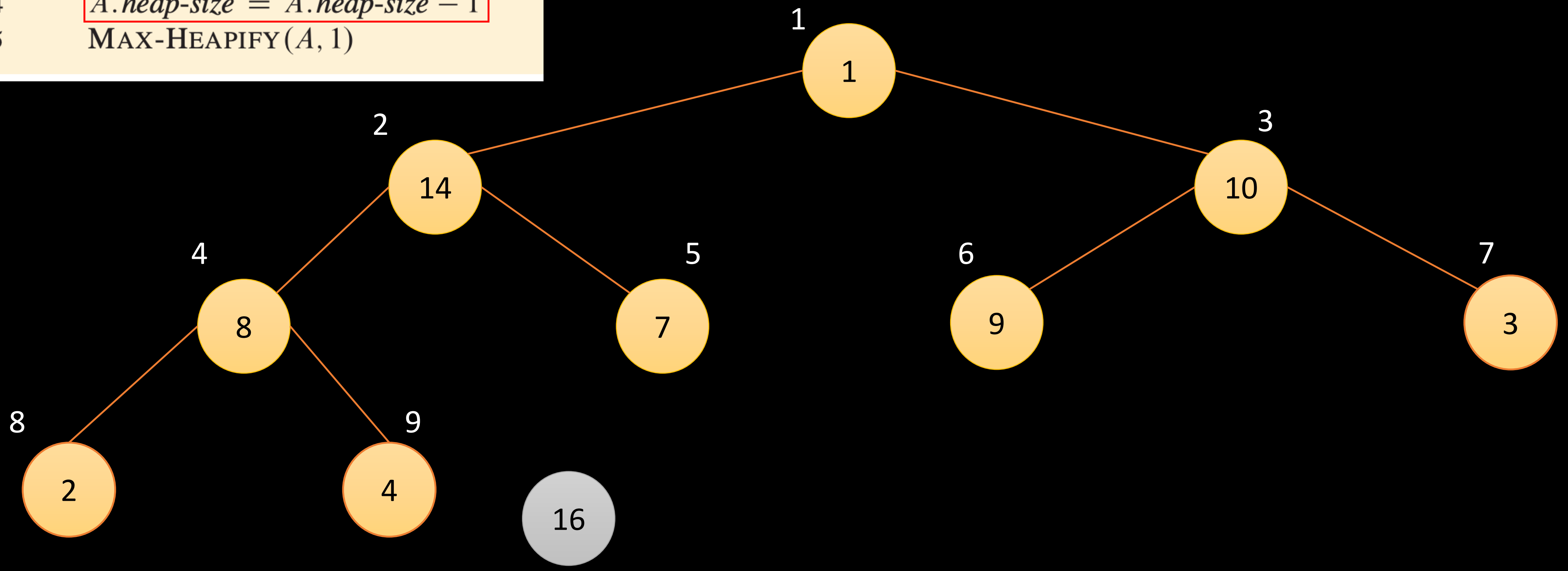
$$i = n = 10$$



The Heapsort Algorithm

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

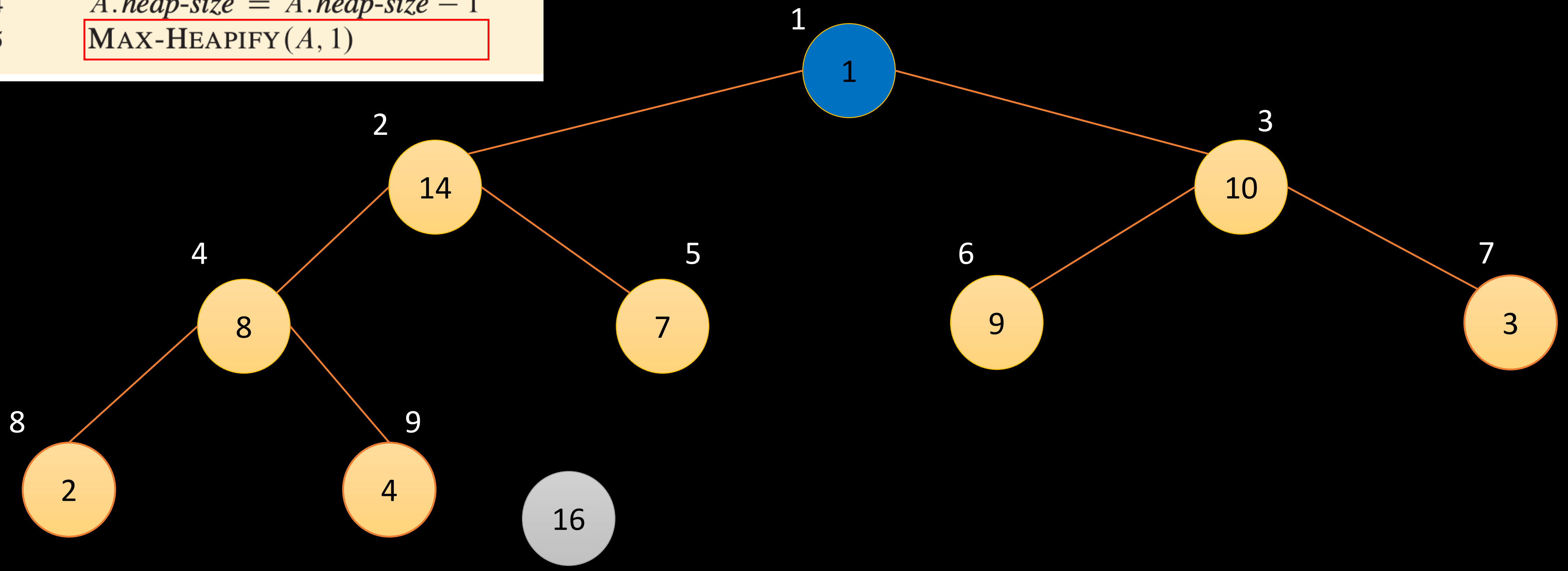
$$i = n = 10$$



The Heapsort Algorithm

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

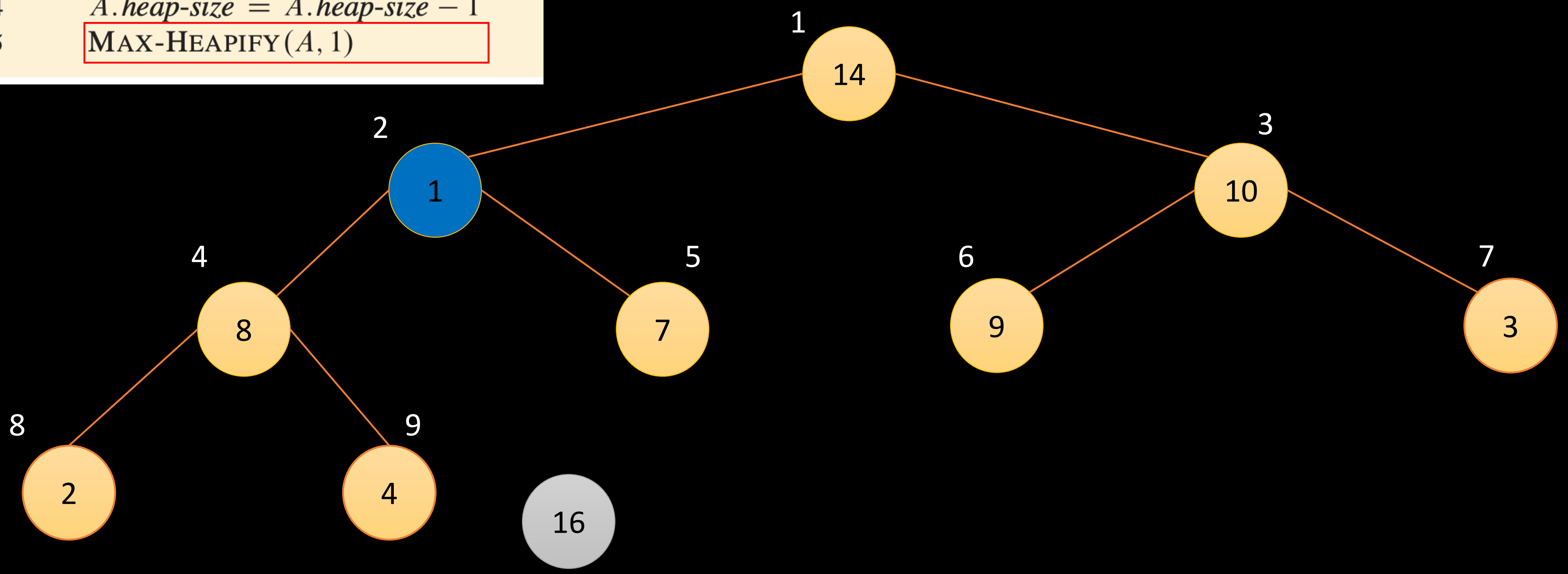
$i = n = 10$



The Heapsort Algorithm

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

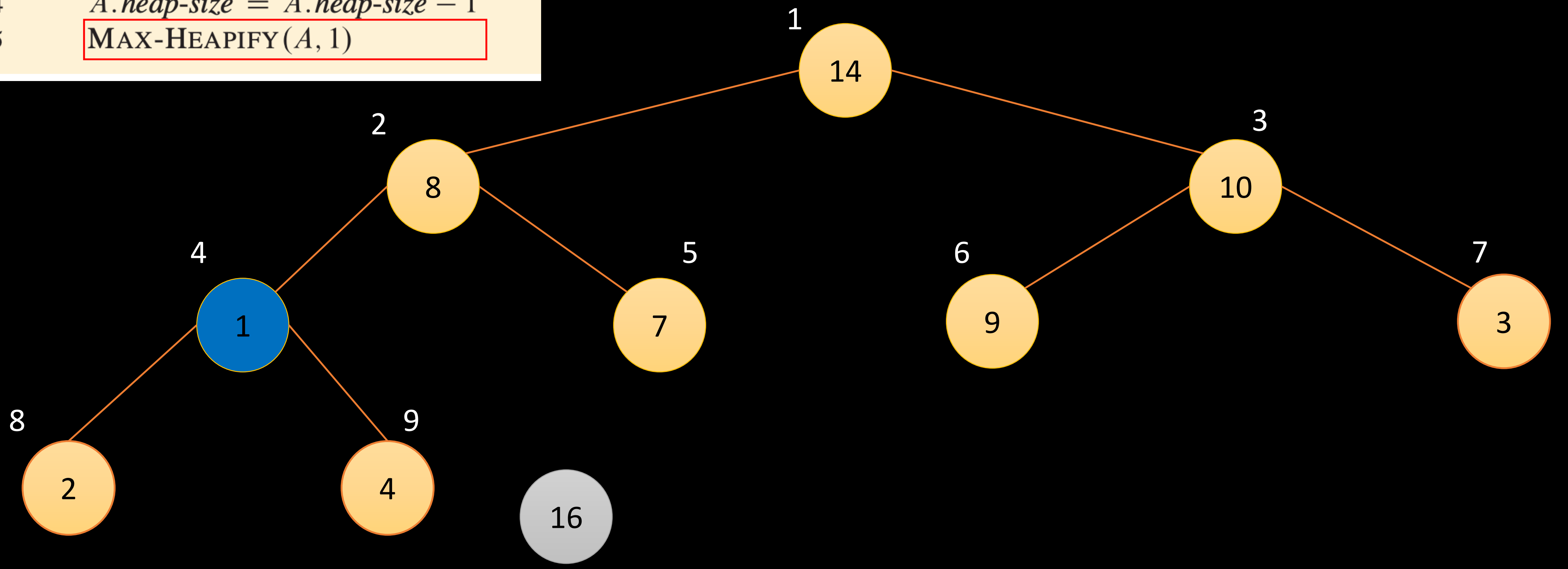
$$i = n = 10$$



The Heapsort Algorithm

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

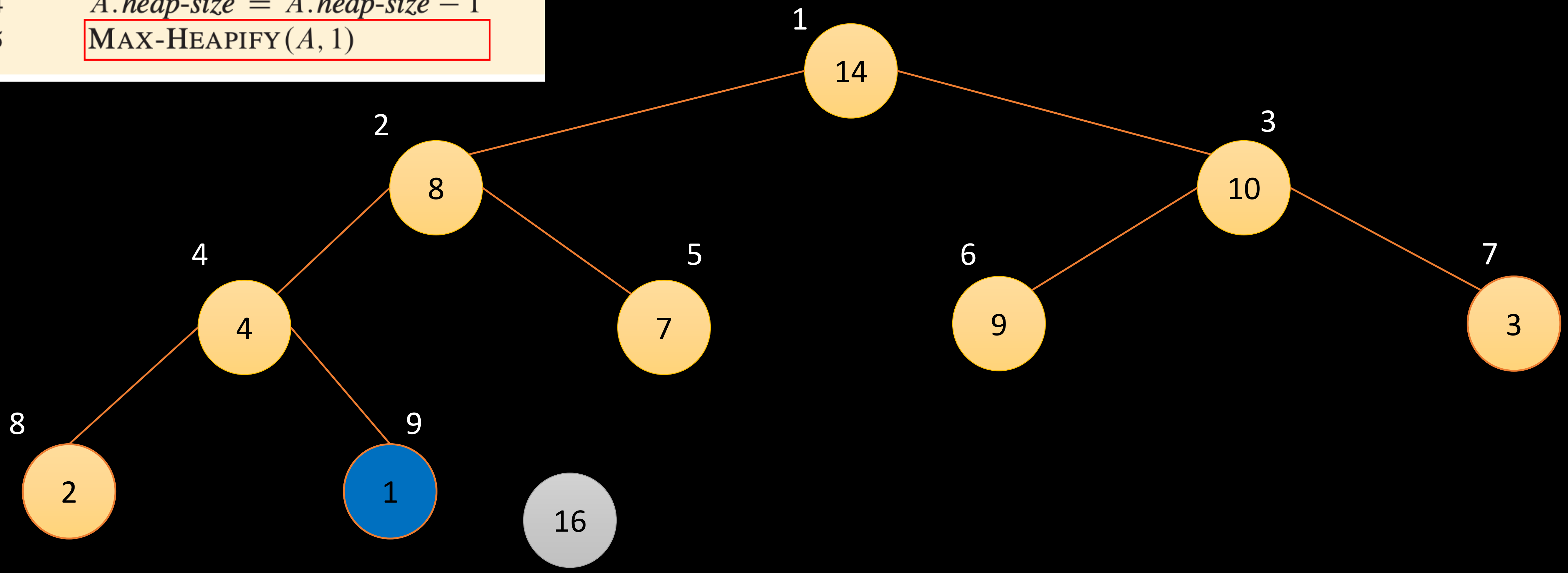
$i = n = 10$



The Heapsort Algorithm

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

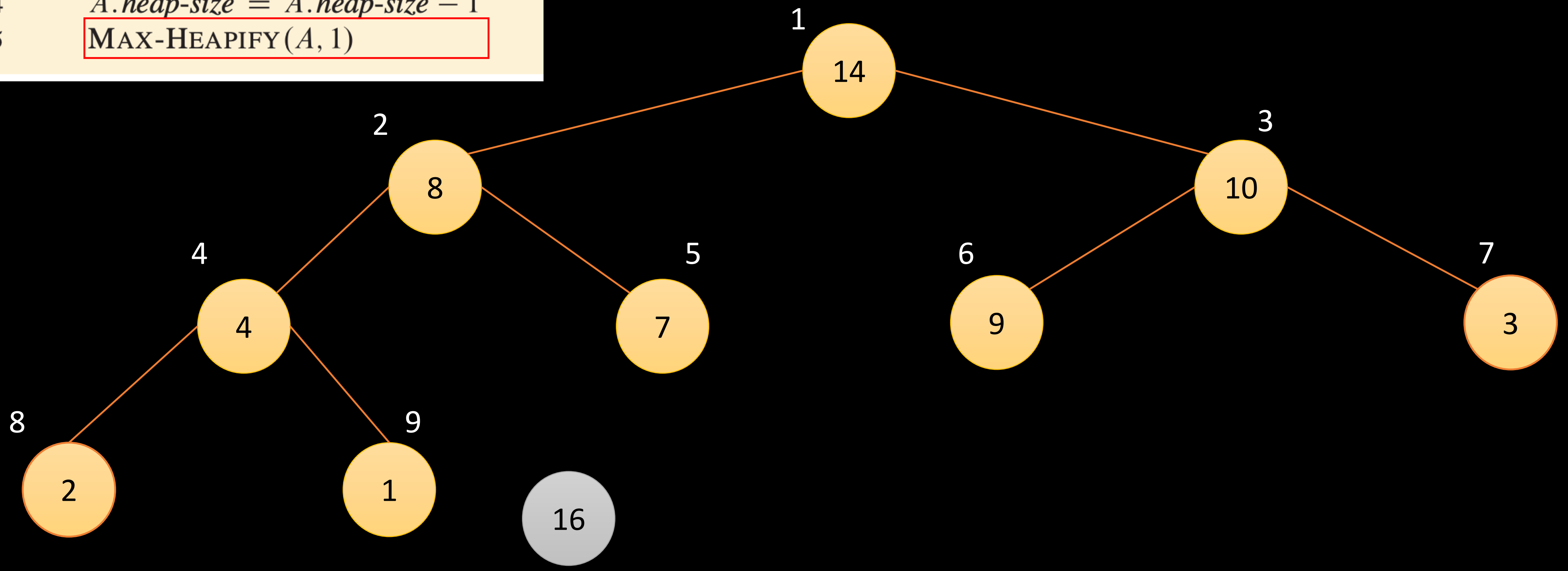
$i = n = 10$



The Heapsort Algorithm

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

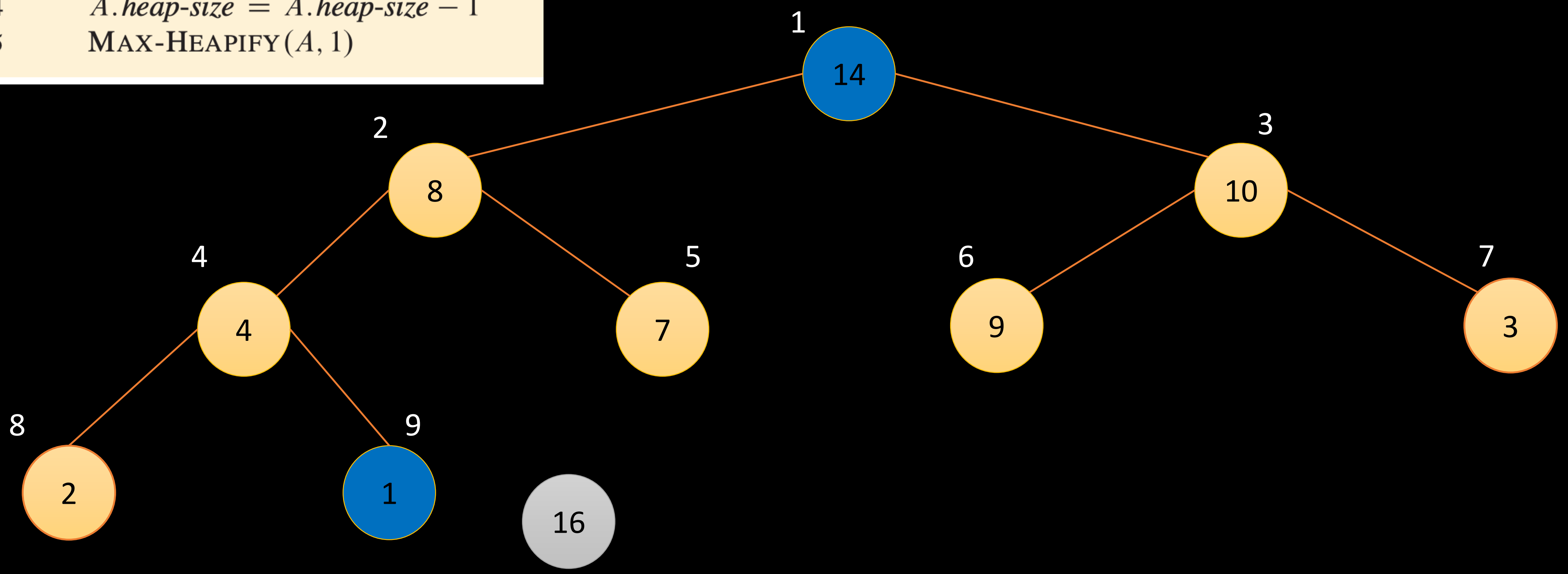
$$i = n = 10$$



The Heapsort Algorithm

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

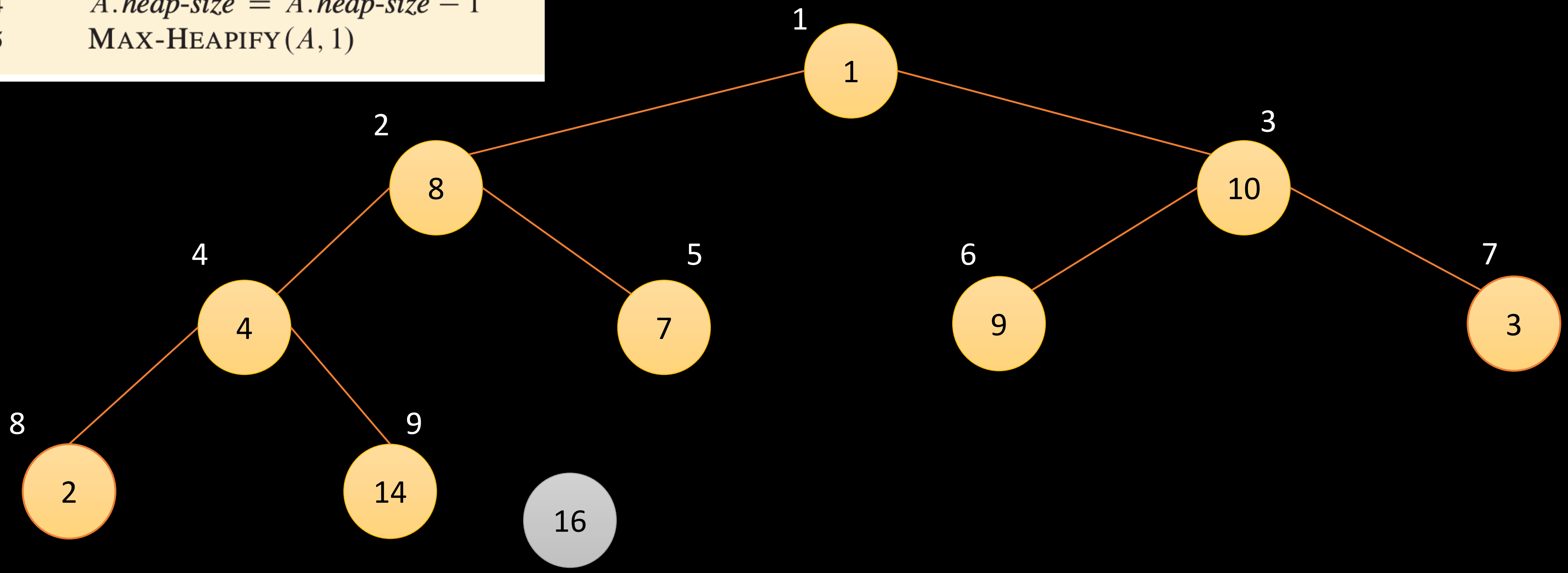
$$i = n = 9$$



The Heapsort Algorithm

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

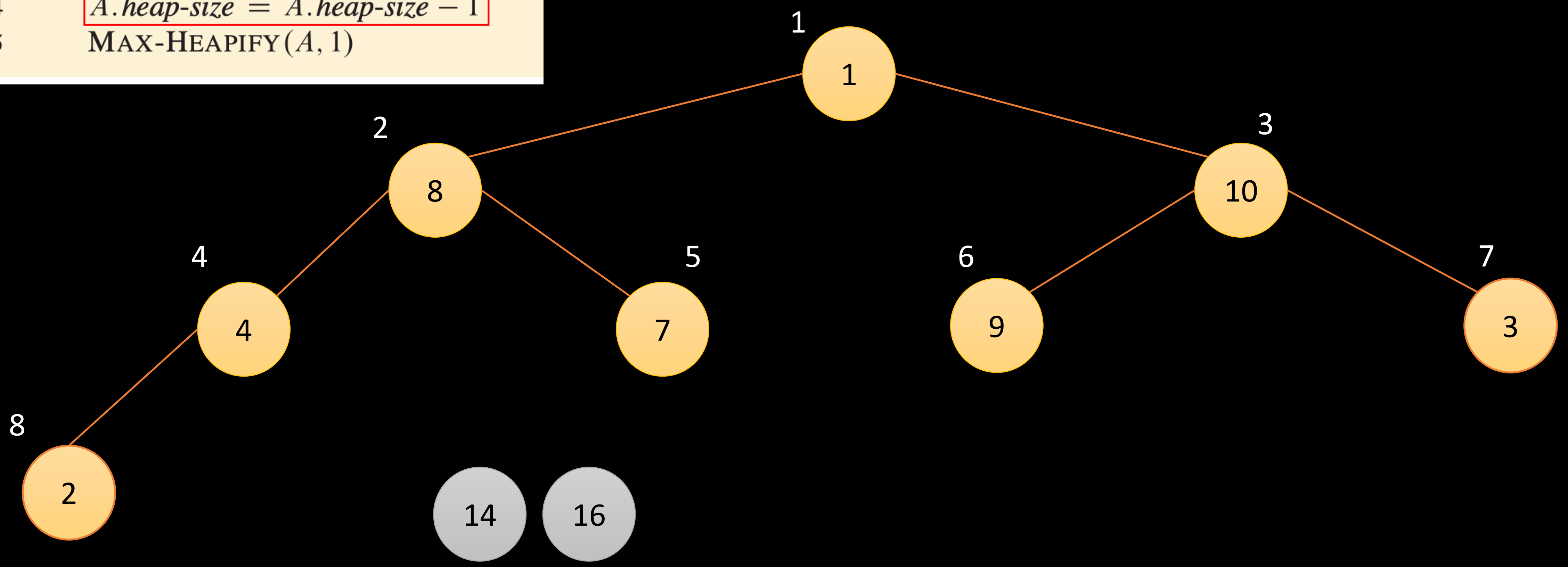
$i = n = 9$



The Heapsort Algorithm

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

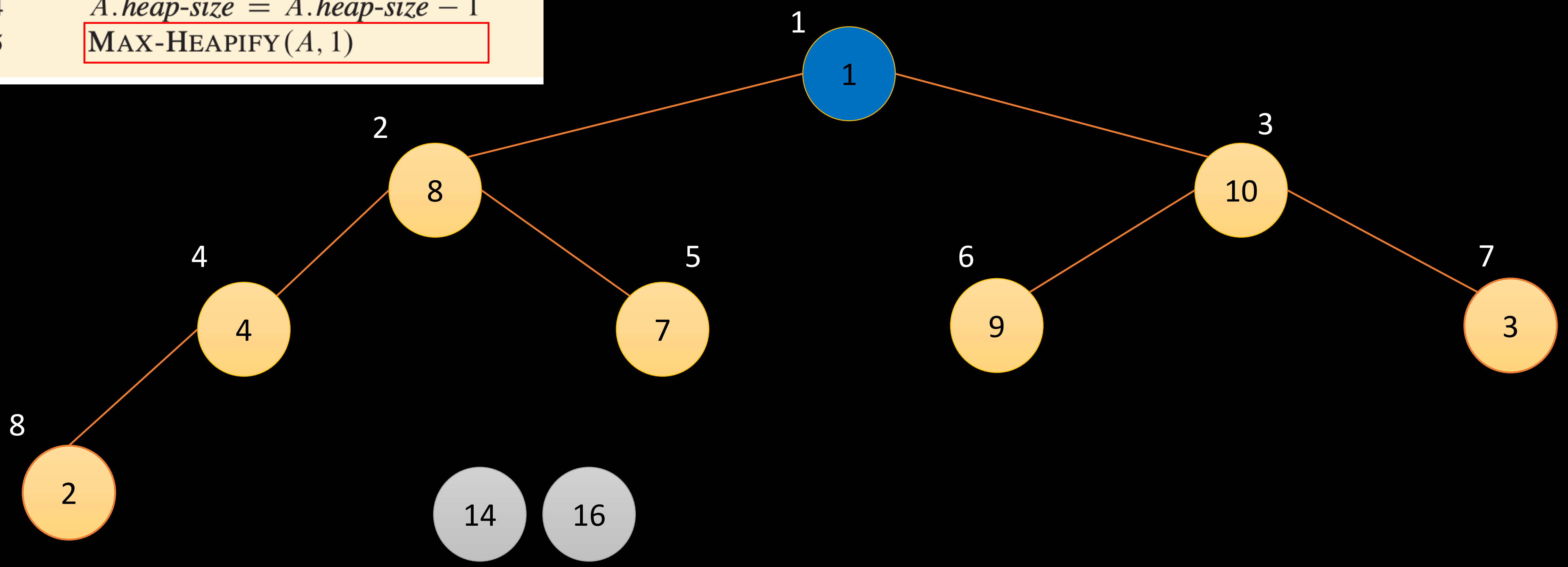
$$i = n = 9$$



The Heapsort Algorithm

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

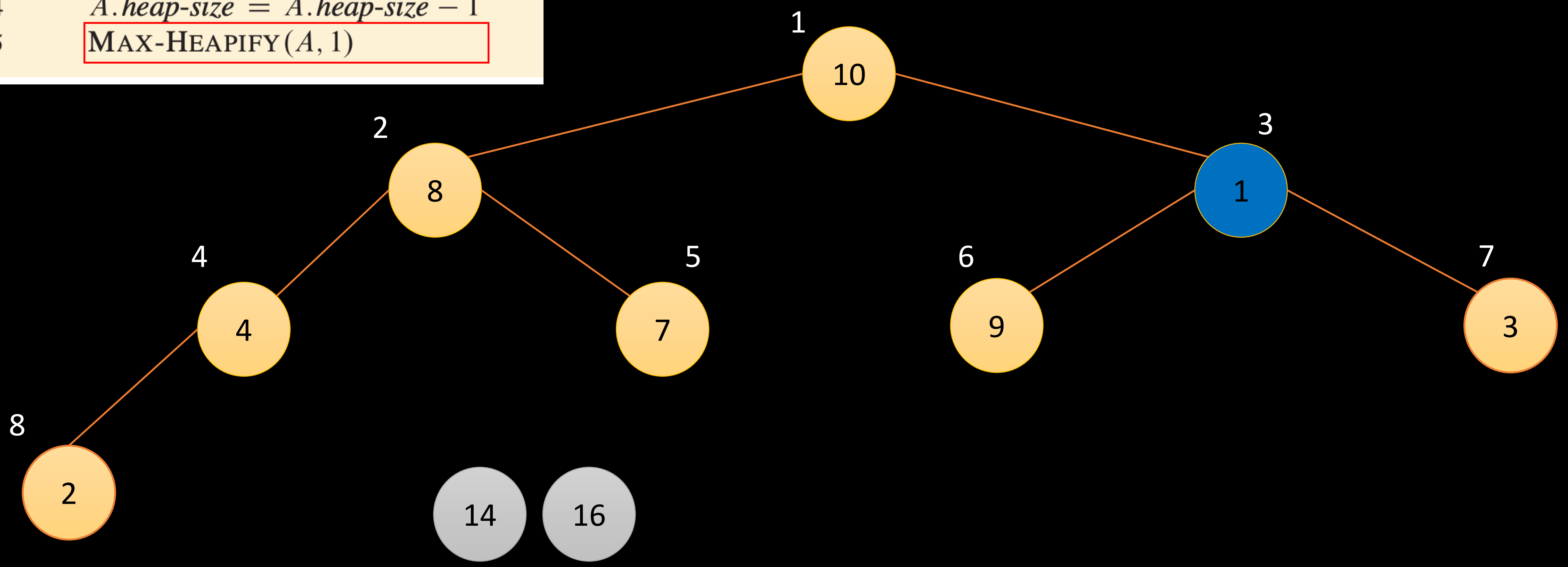
$i = n = 9$



The Heapsort Algorithm

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

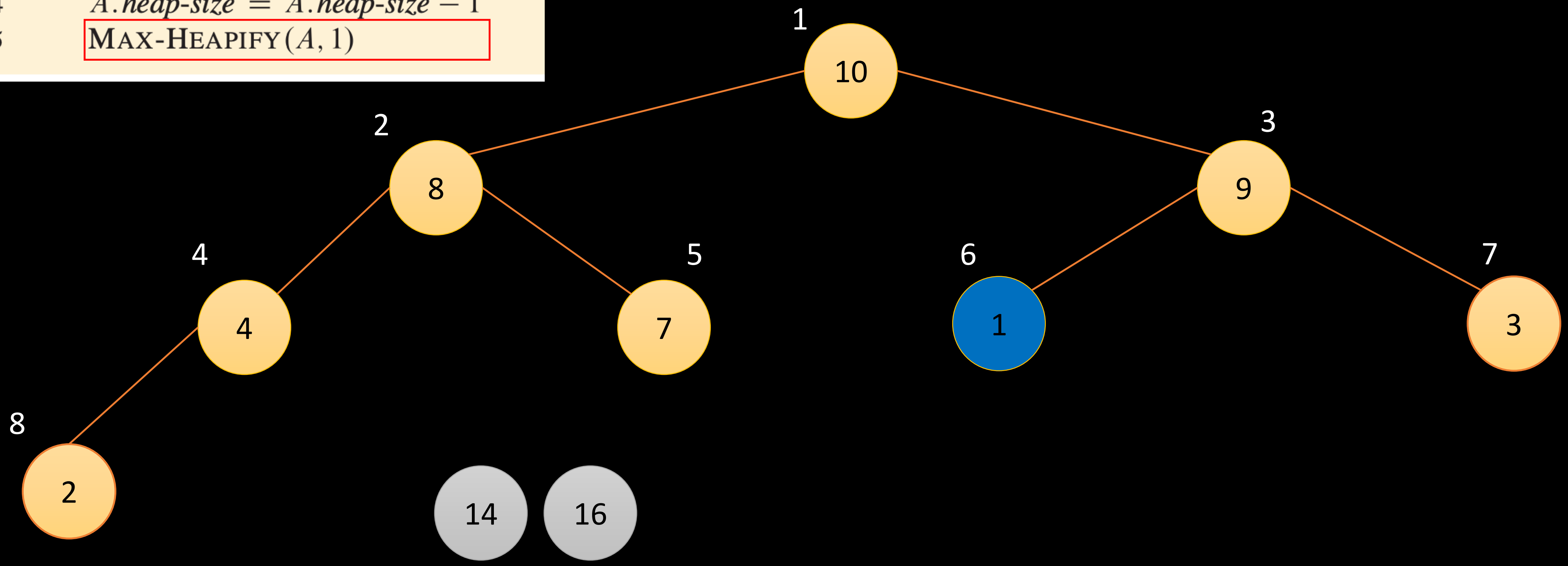
$$i = n = 9$$



The Heapsort Algorithm

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

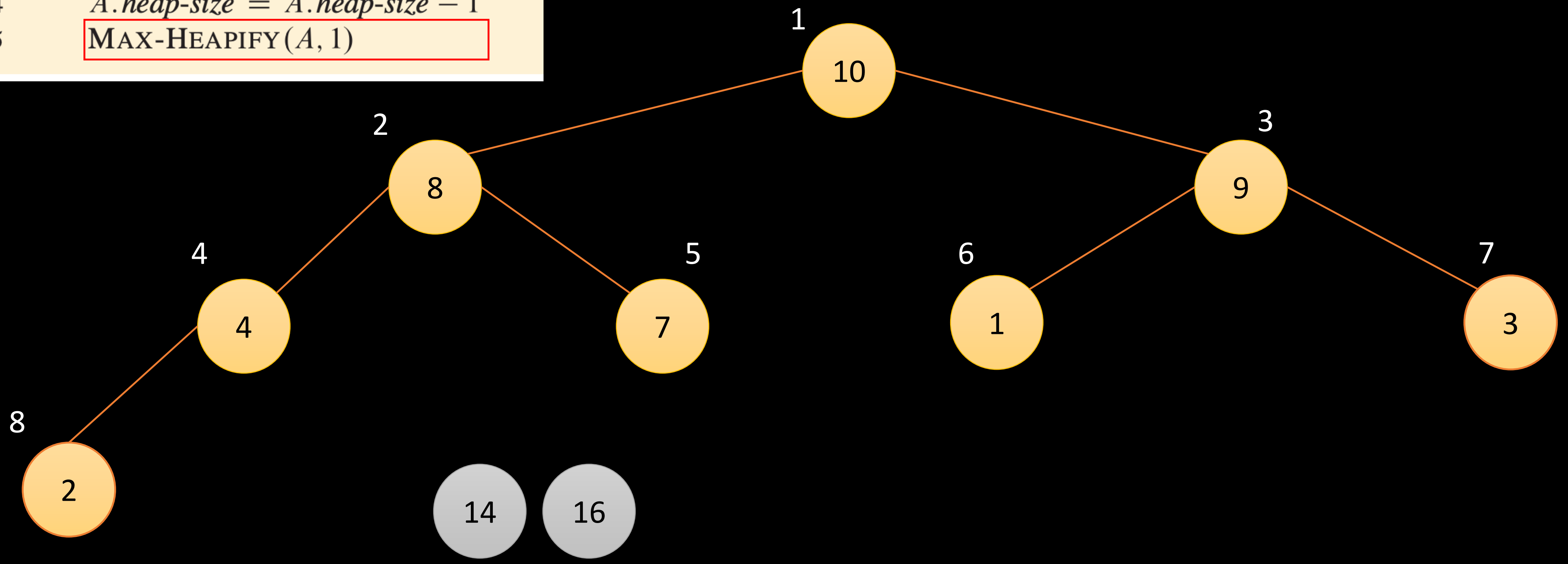
$i = n = 9$



The Heapsort Algorithm

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

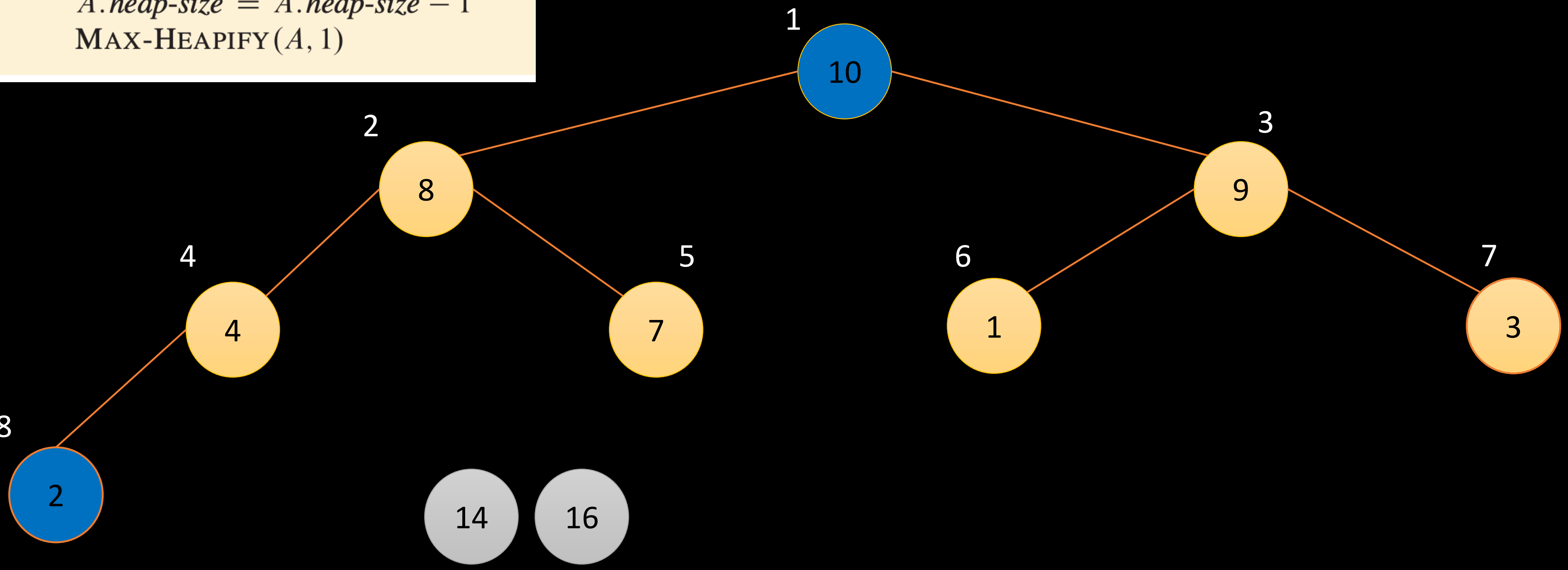
$$i = n = 9$$



The Heapsort Algorithm

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

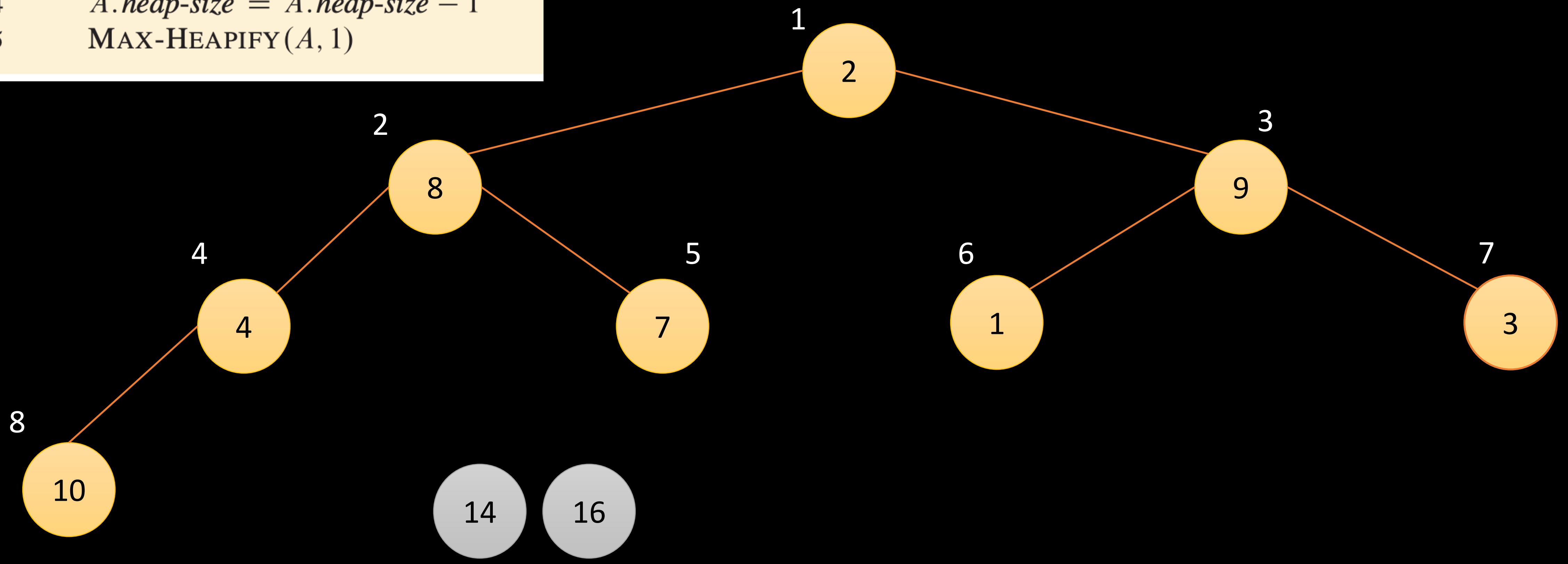
$$i = n = 8$$



The Heapsort Algorithm

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

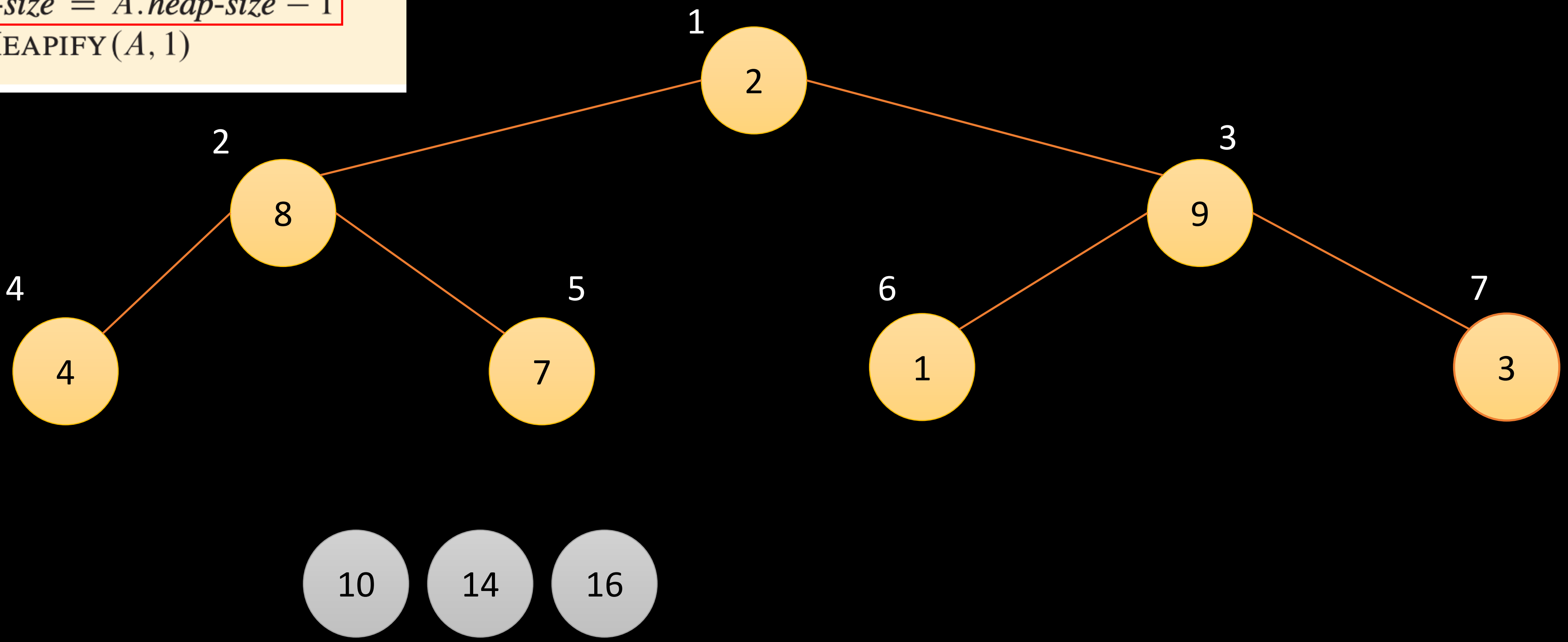
$i = n = 8$



The Heapsort Algorithm

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

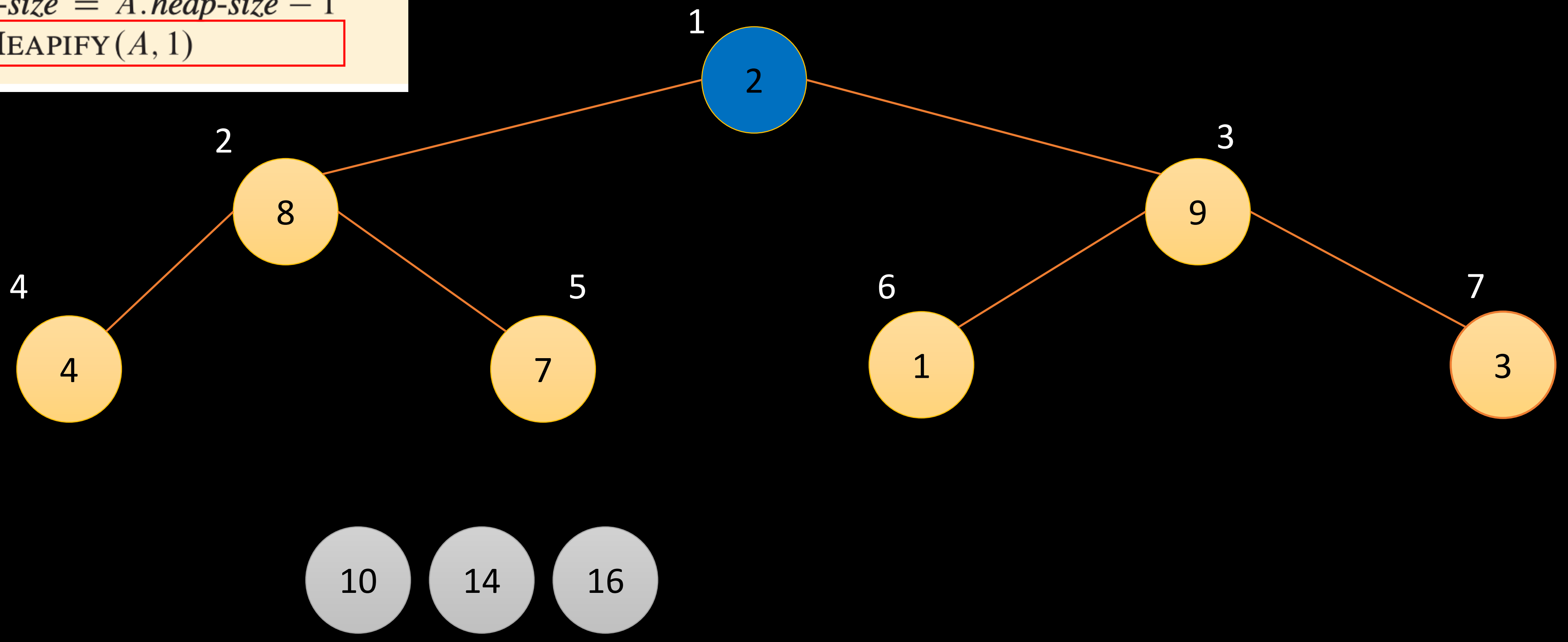
$i = n = 8$



The Heapsort Algorithm

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

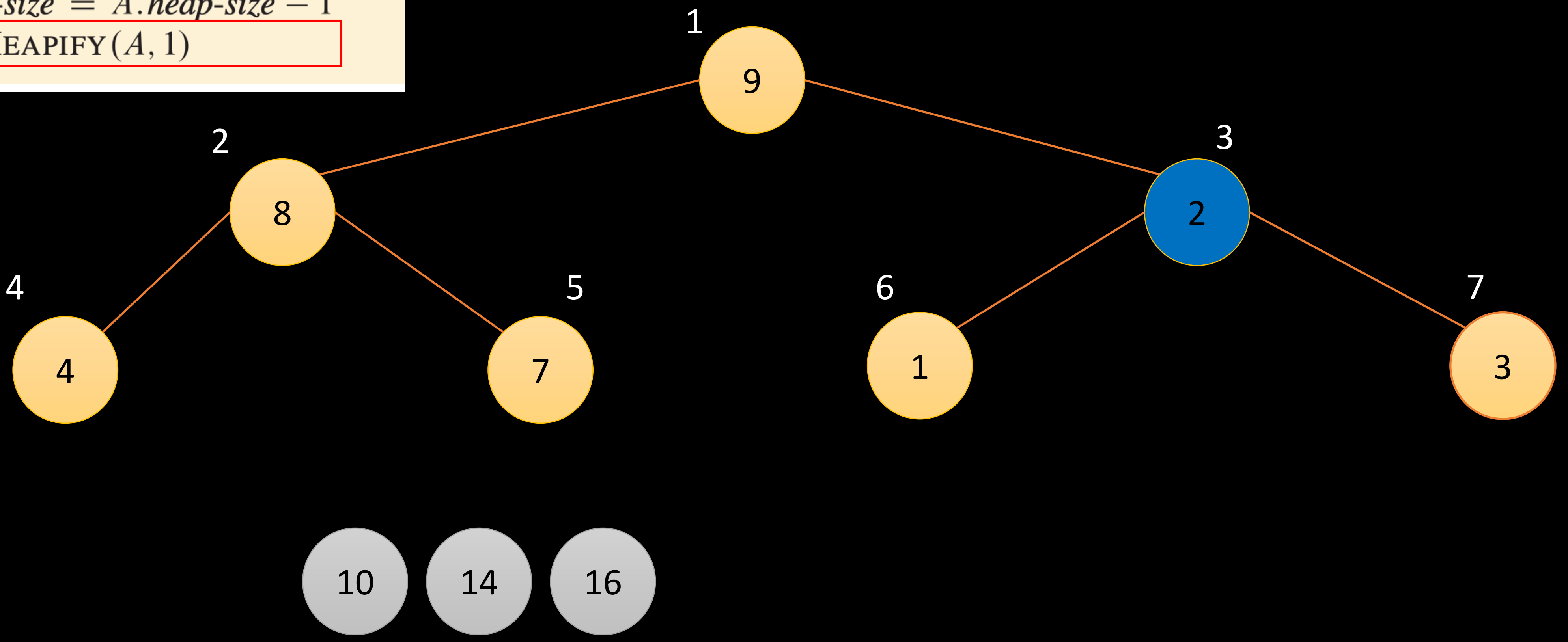
$$i = n = 8$$



The Heapsort Algorithm

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

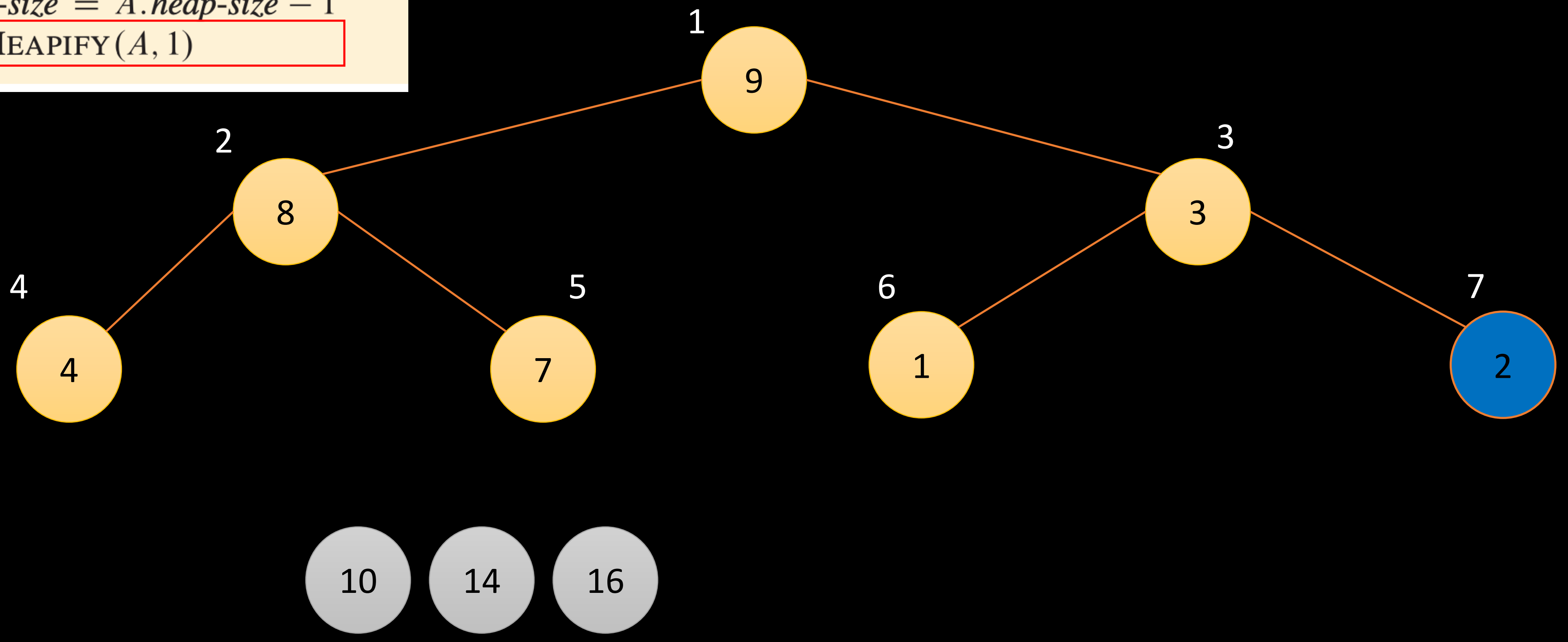
$$i = n = 8$$



The Heapsort Algorithm

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

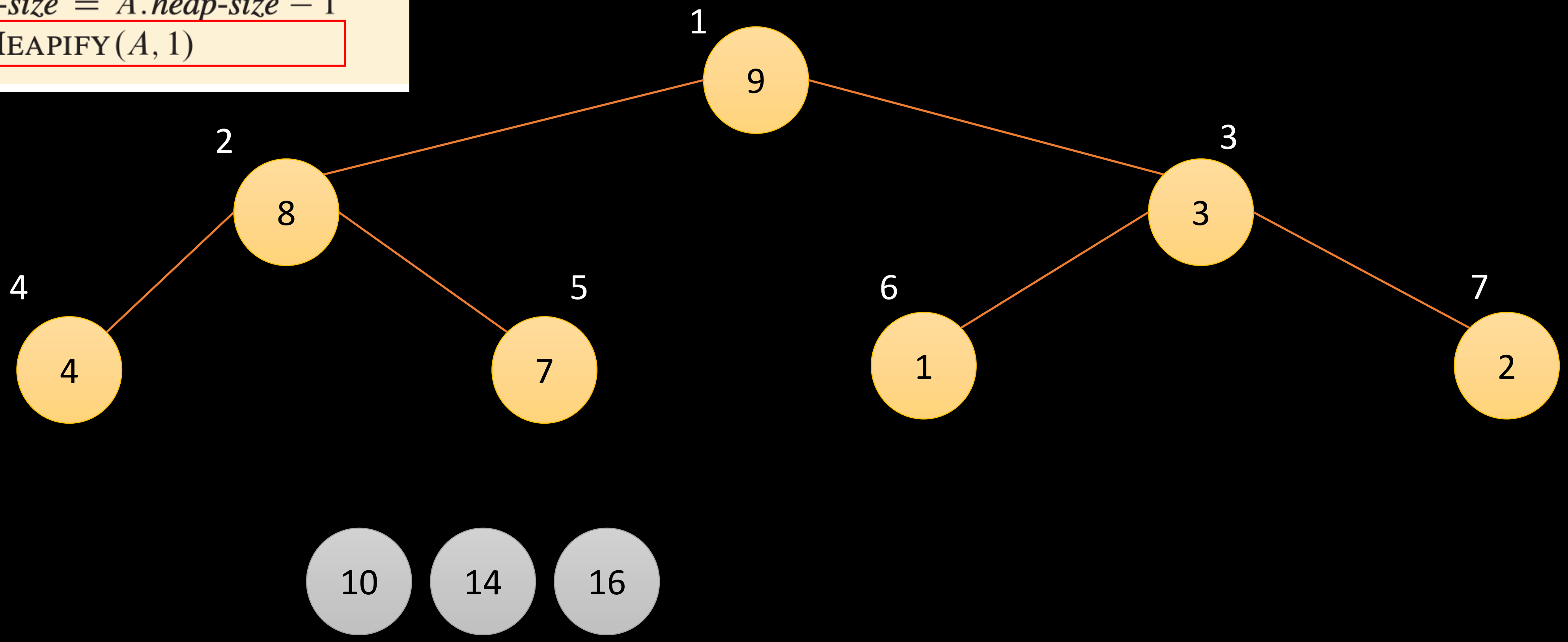
$$i = n = 8$$



The Heapsort Algorithm

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

$$i = n = 8$$

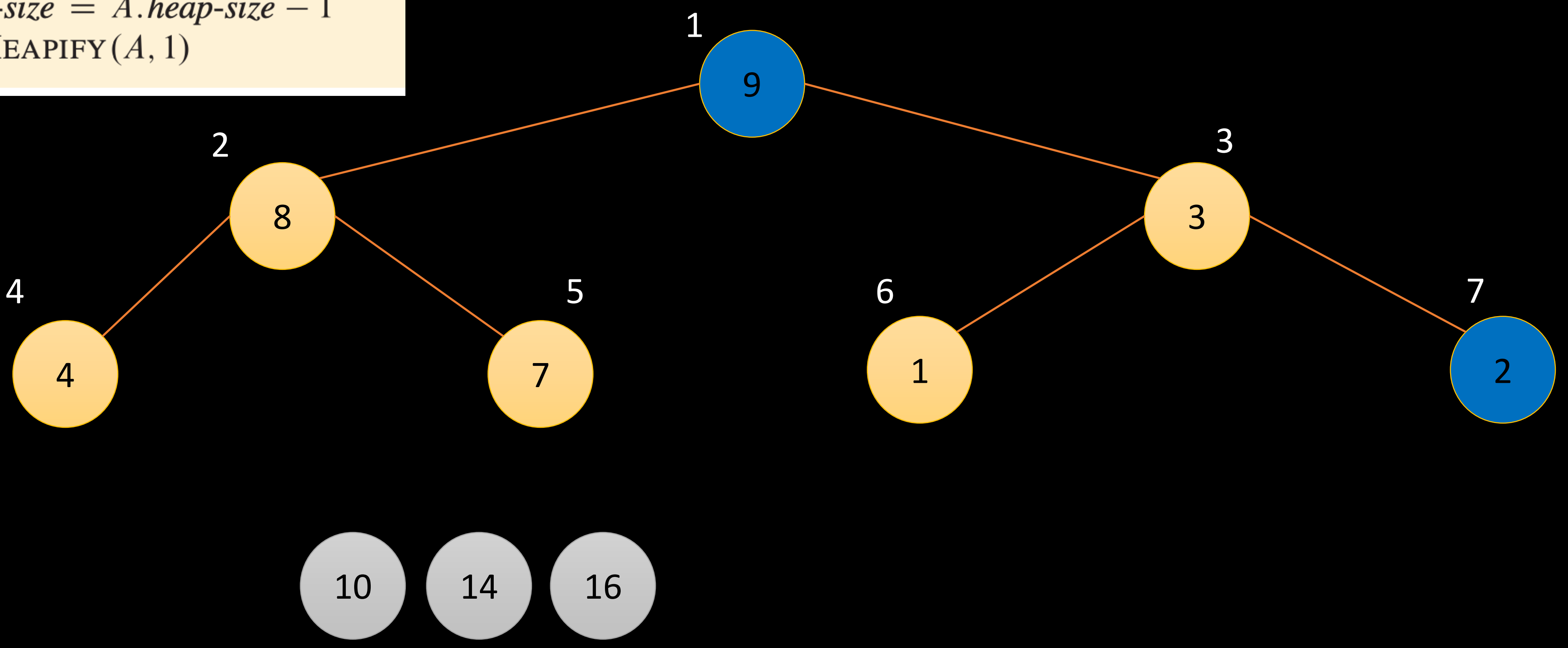


The Heapsort Algorithm

HEAPSORT(A, n)

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

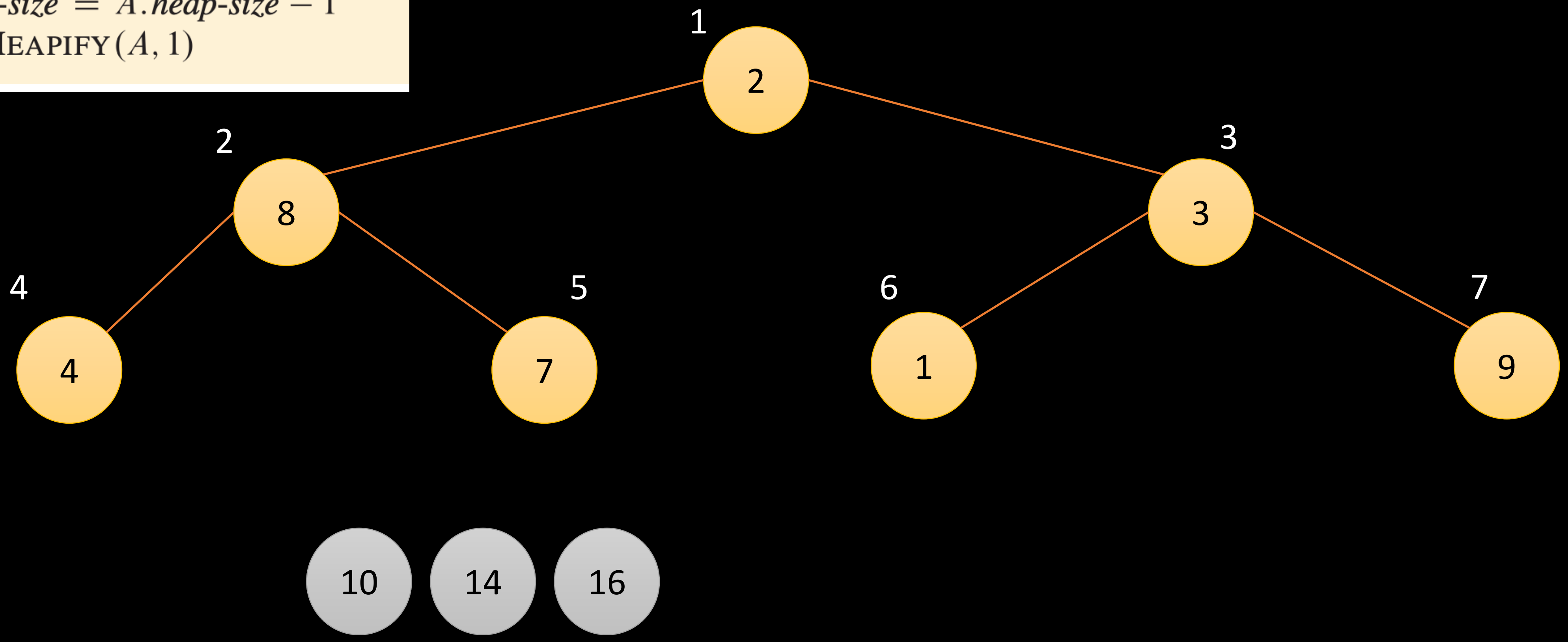
$i = n = 7$



The Heapsort Algorithm

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

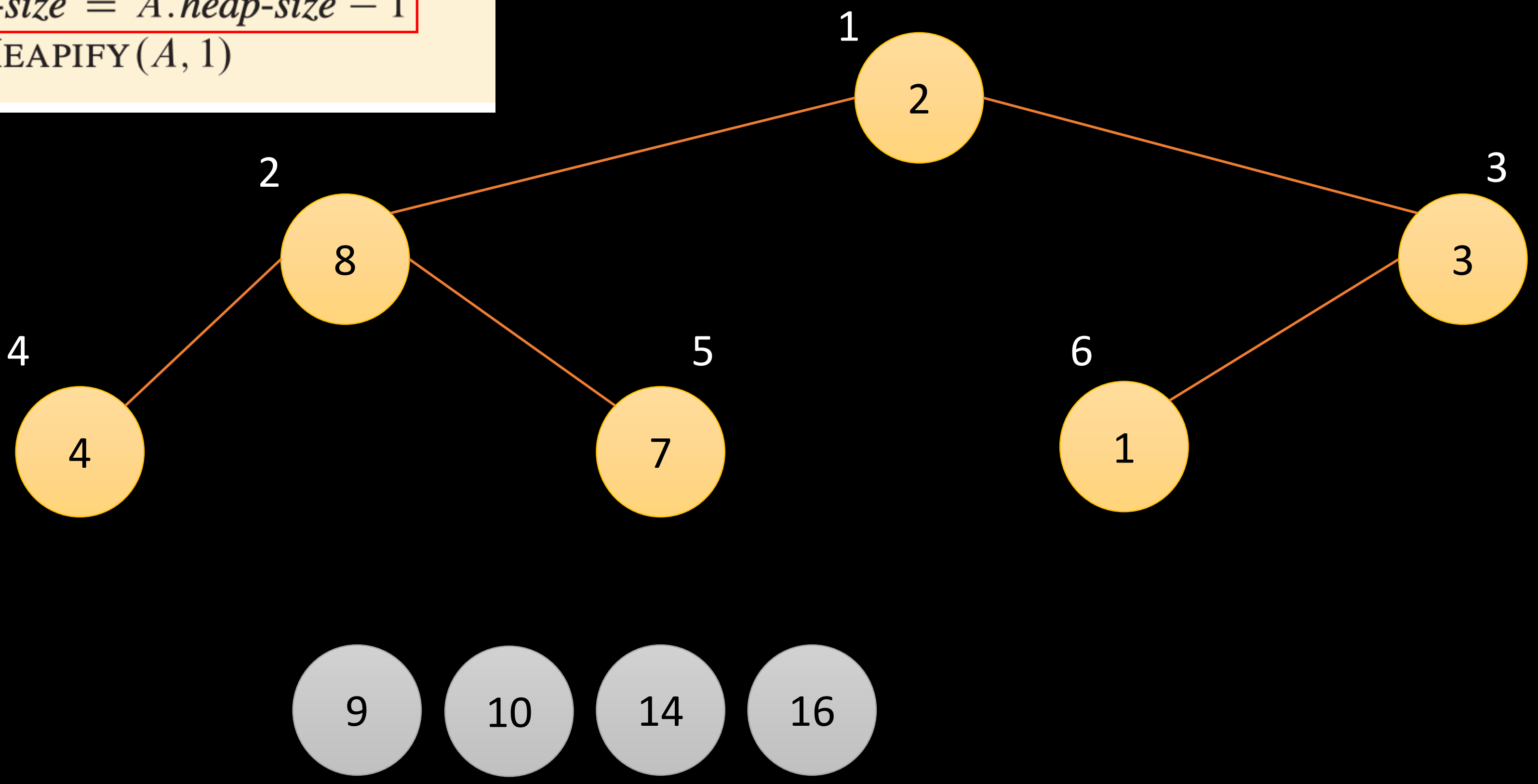
$i = n = 7$



The Heapsort Algorithm

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

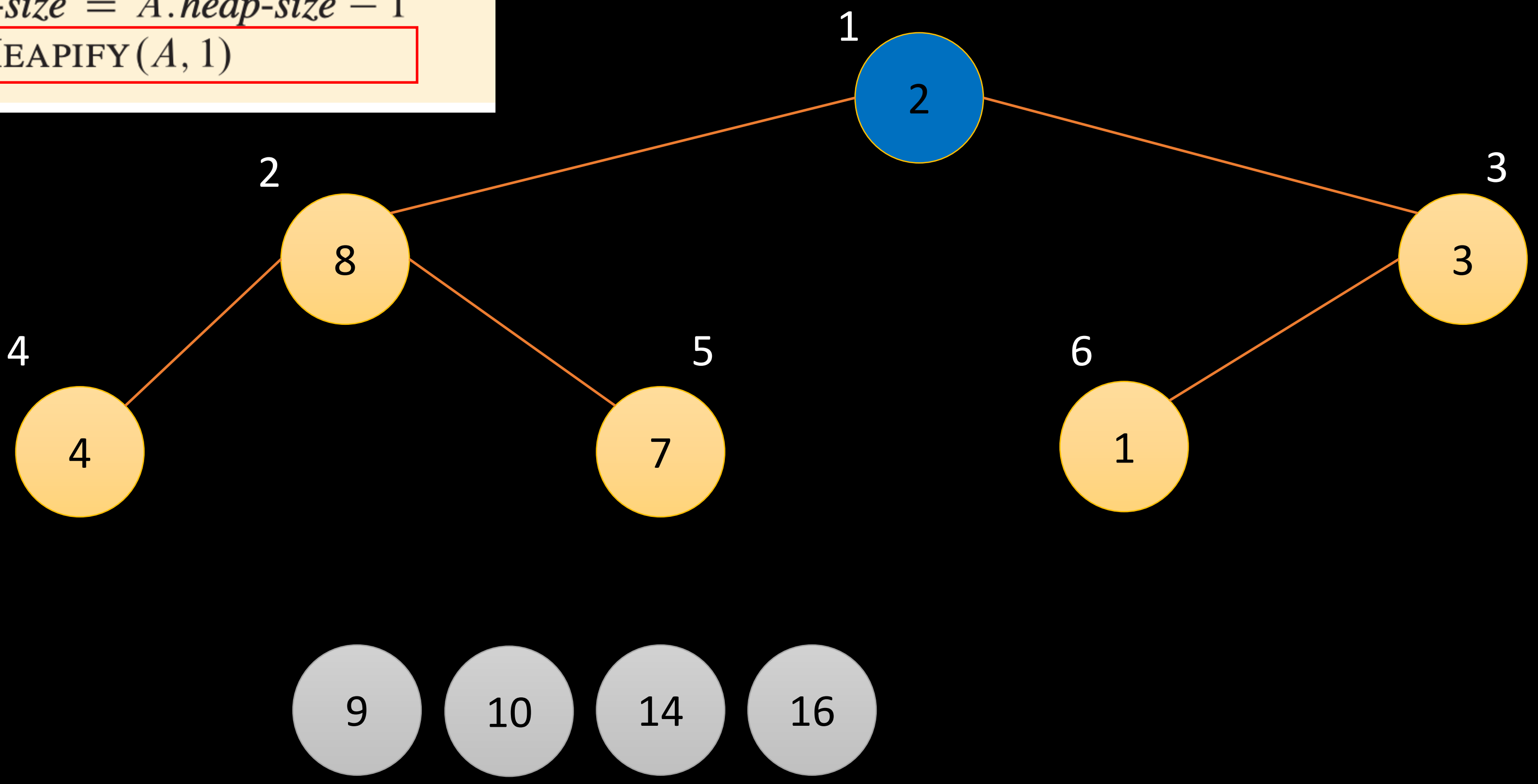
$i = n = 7$



The Heapsort Algorithm

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

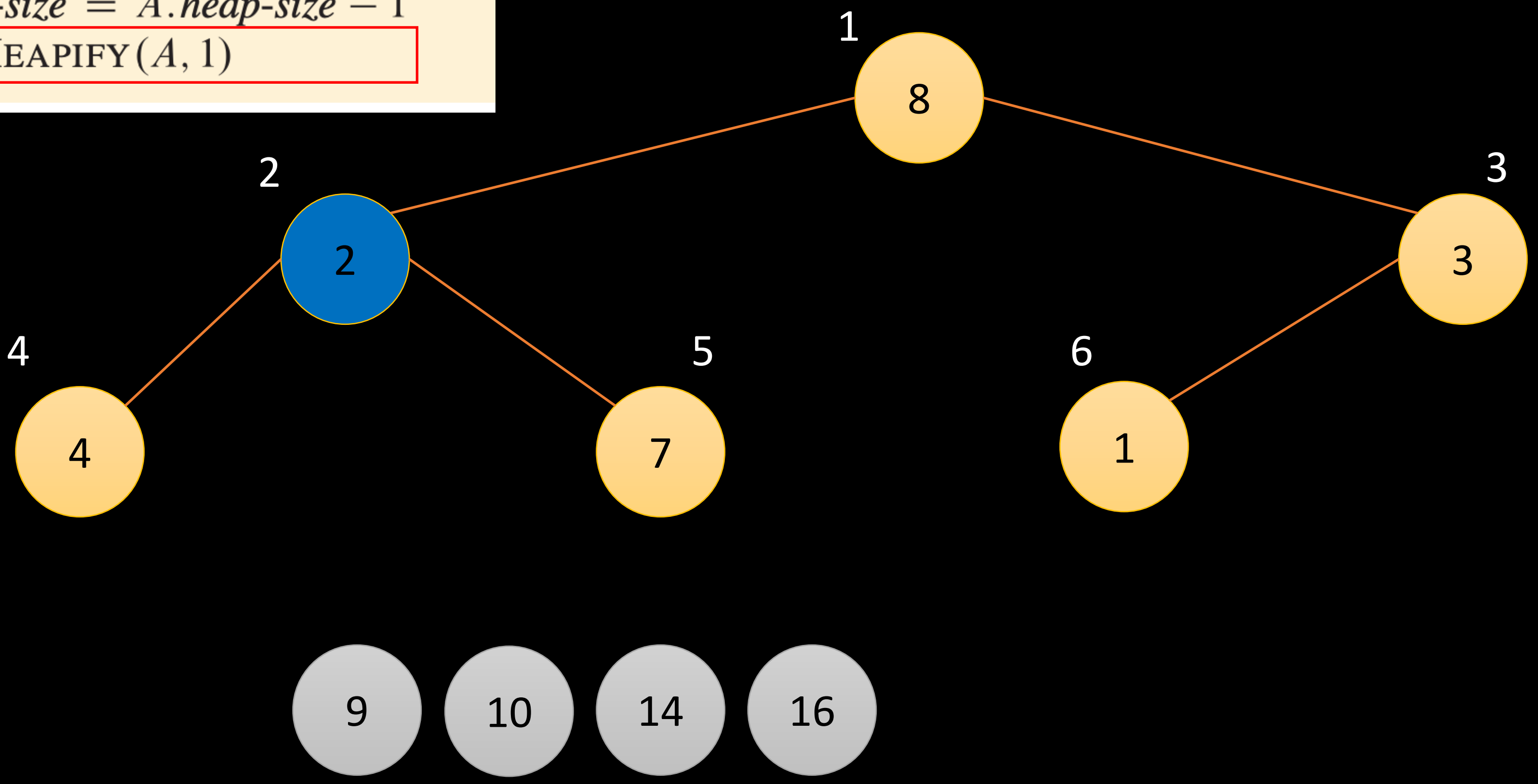
$$i = n = 7$$



The Heapsort Algorithm

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

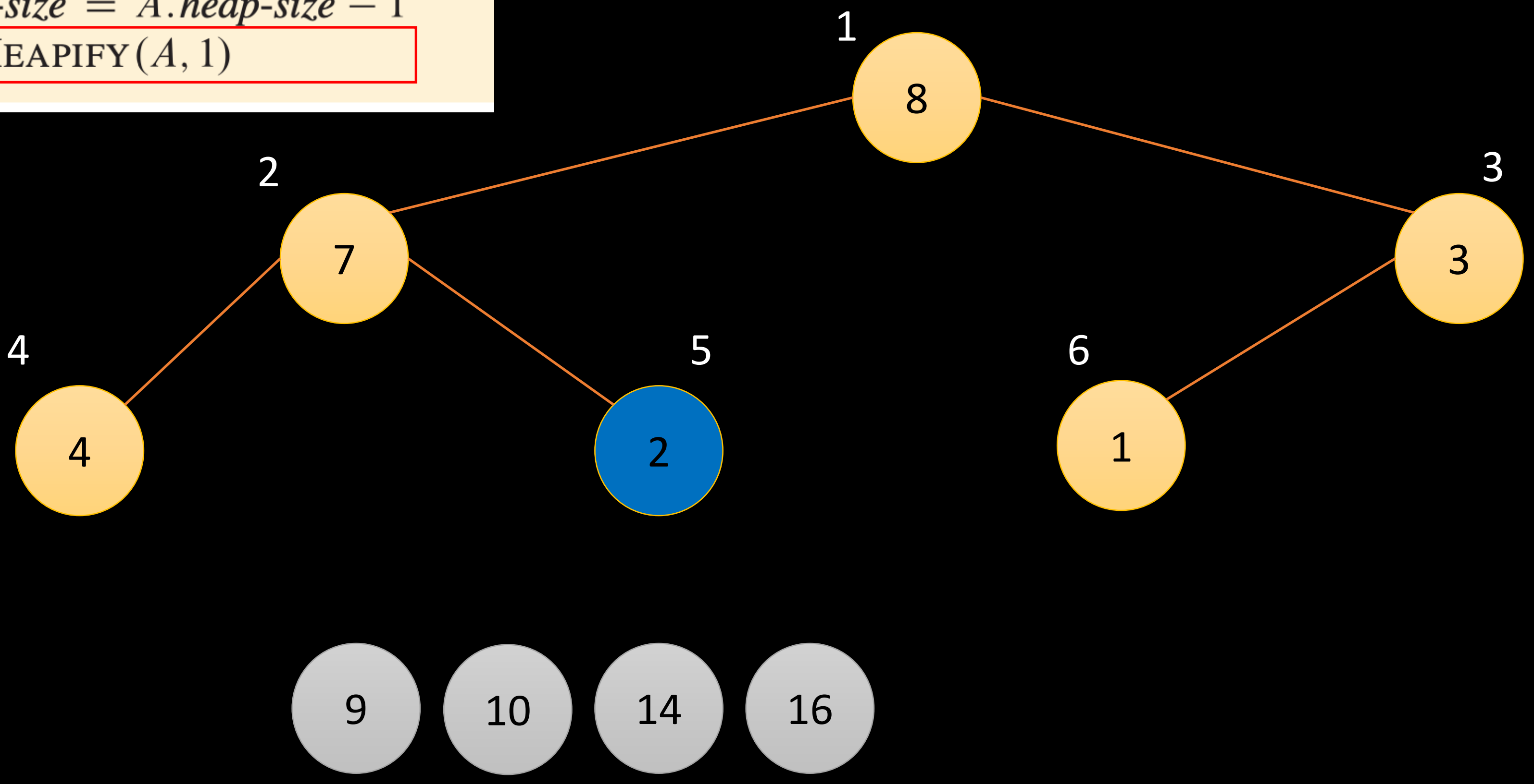
$i = n = 7$



The Heapsort Algorithm

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

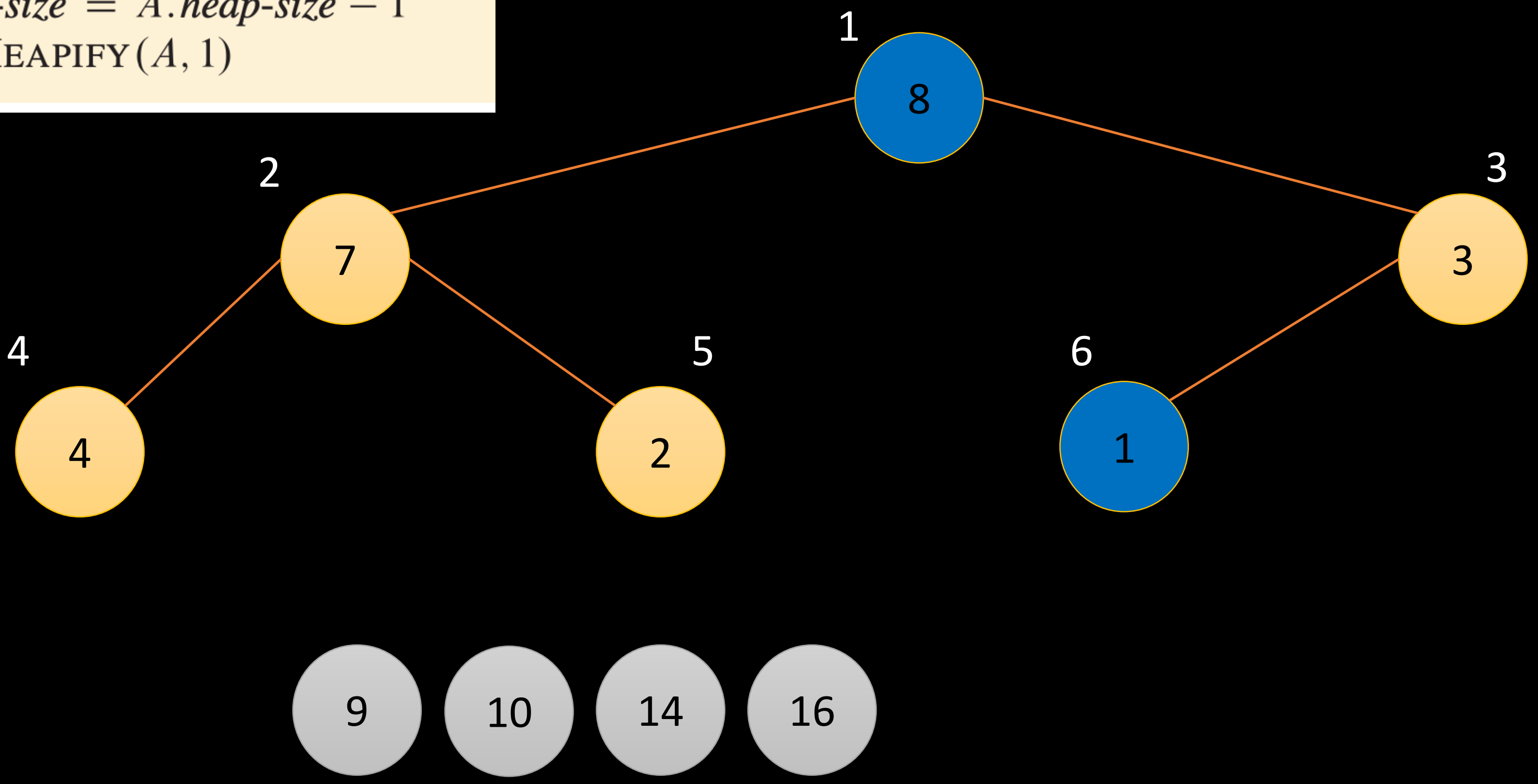
$$i = n = 7$$



The Heapsort Algorithm

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

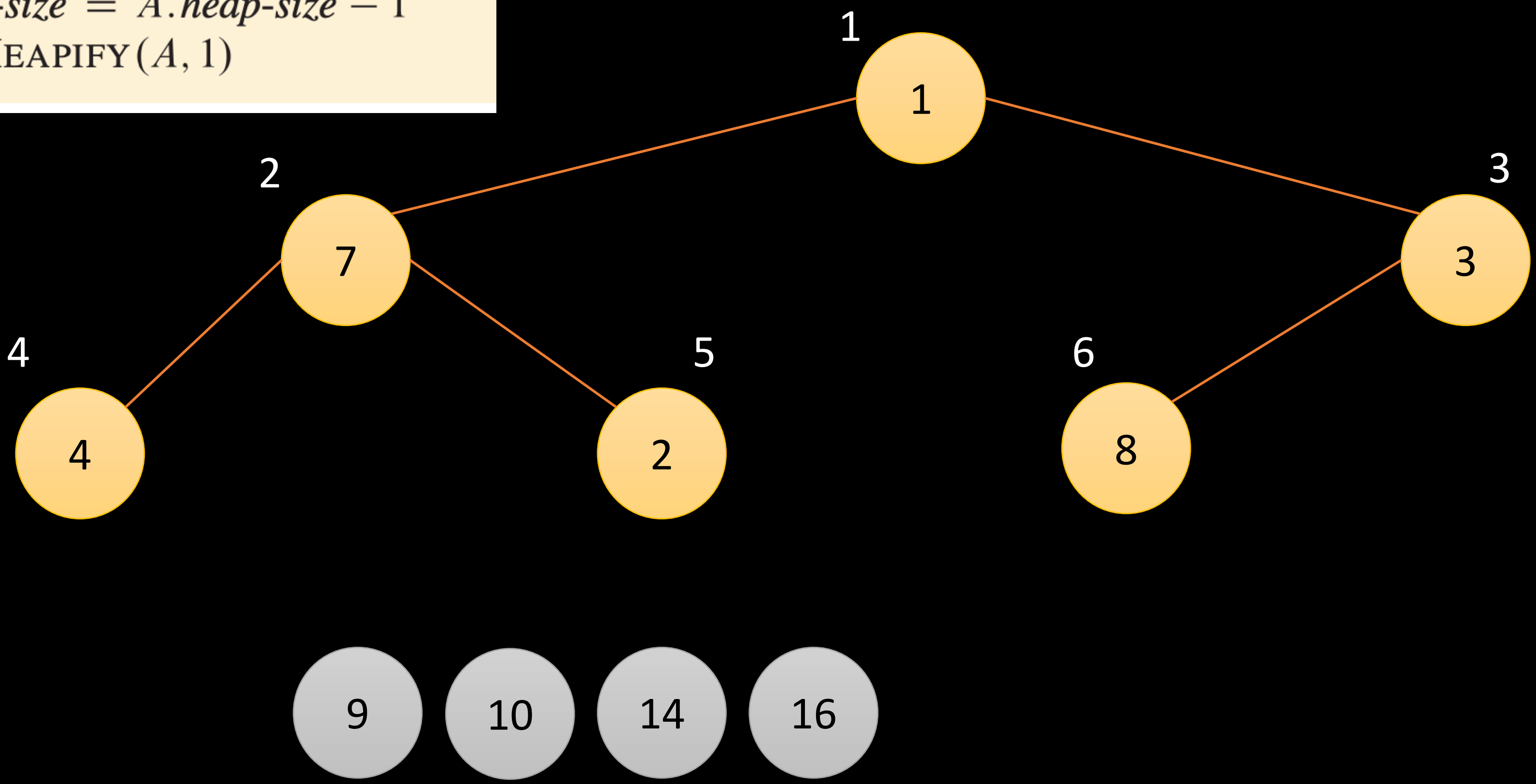
$$i = n = 6$$



The Heapsort Algorithm

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

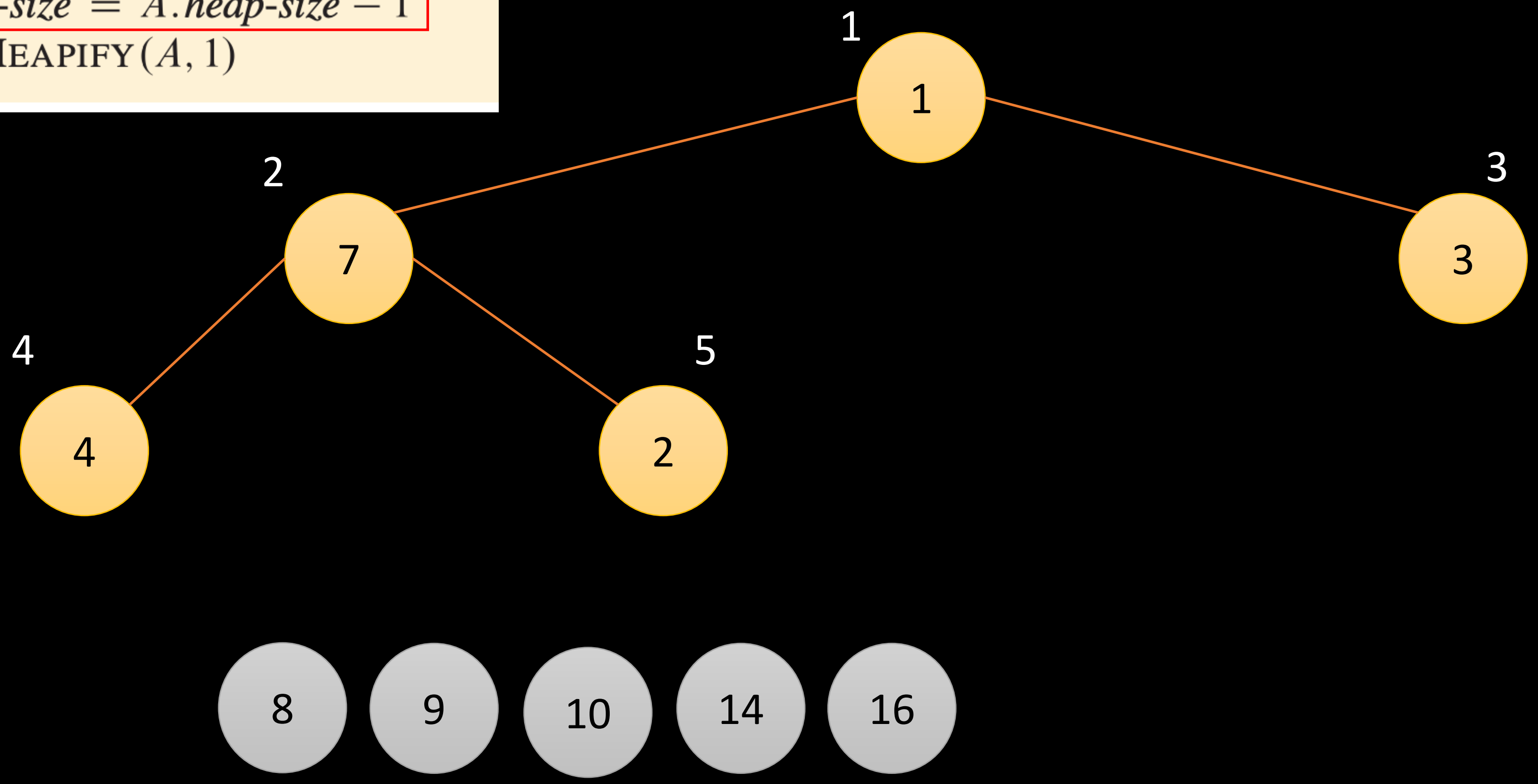
$$i = n = 6$$



The Heapsort Algorithm

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

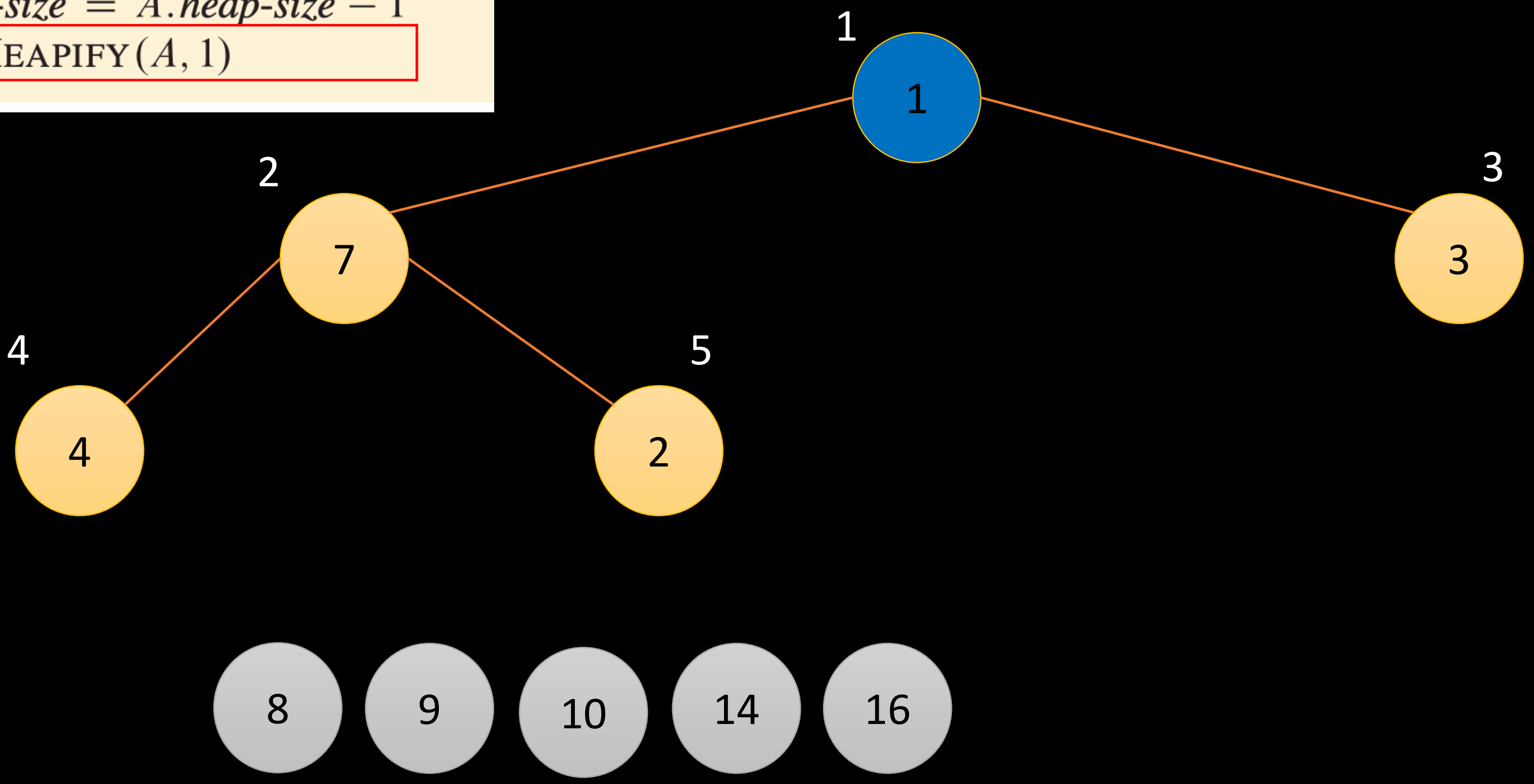
$i = n = 6$



The Heapsort Algorithm

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

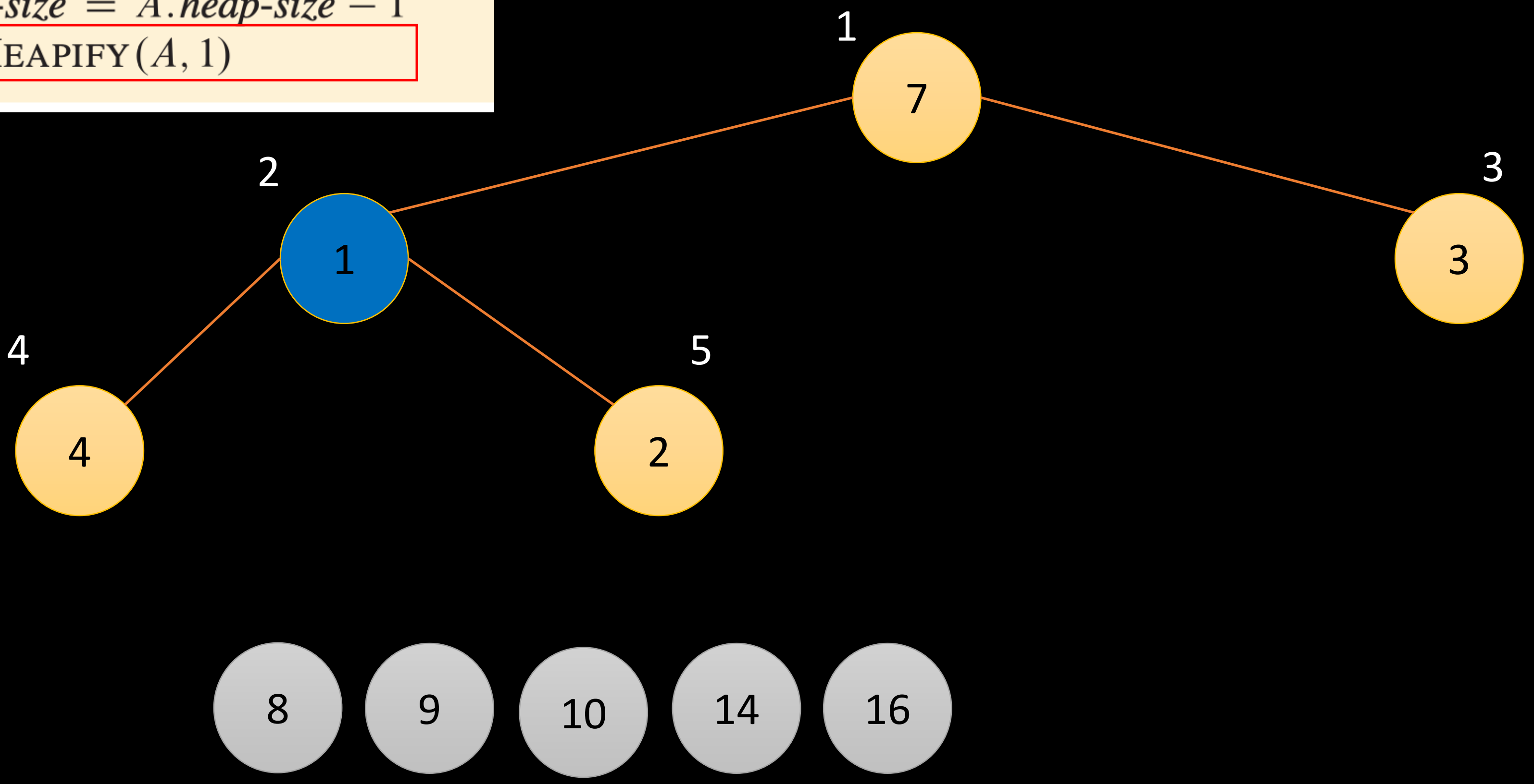
$$i = n = 6$$



The Heapsort Algorithm

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

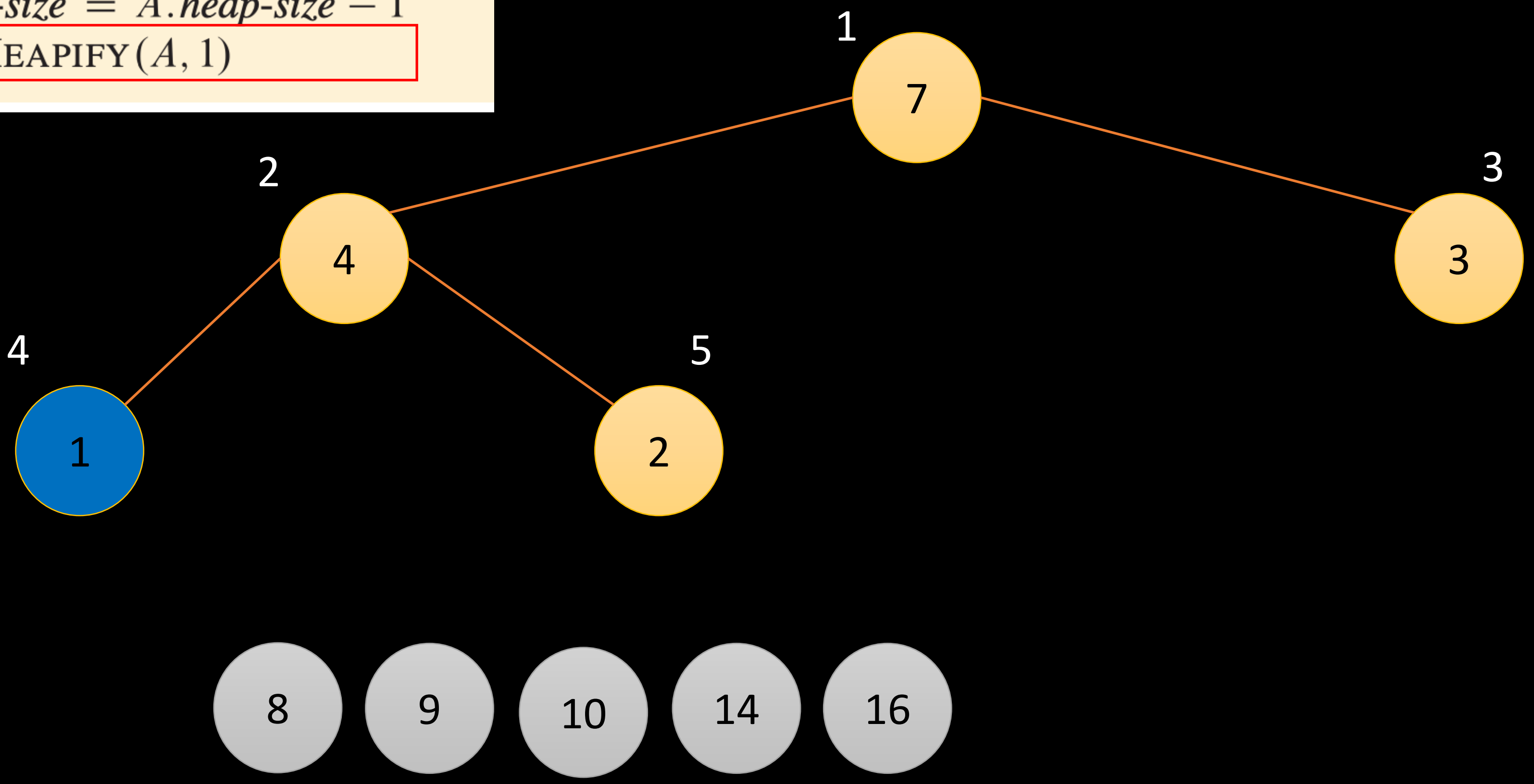
$i = n = 6$



The Heapsort Algorithm

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

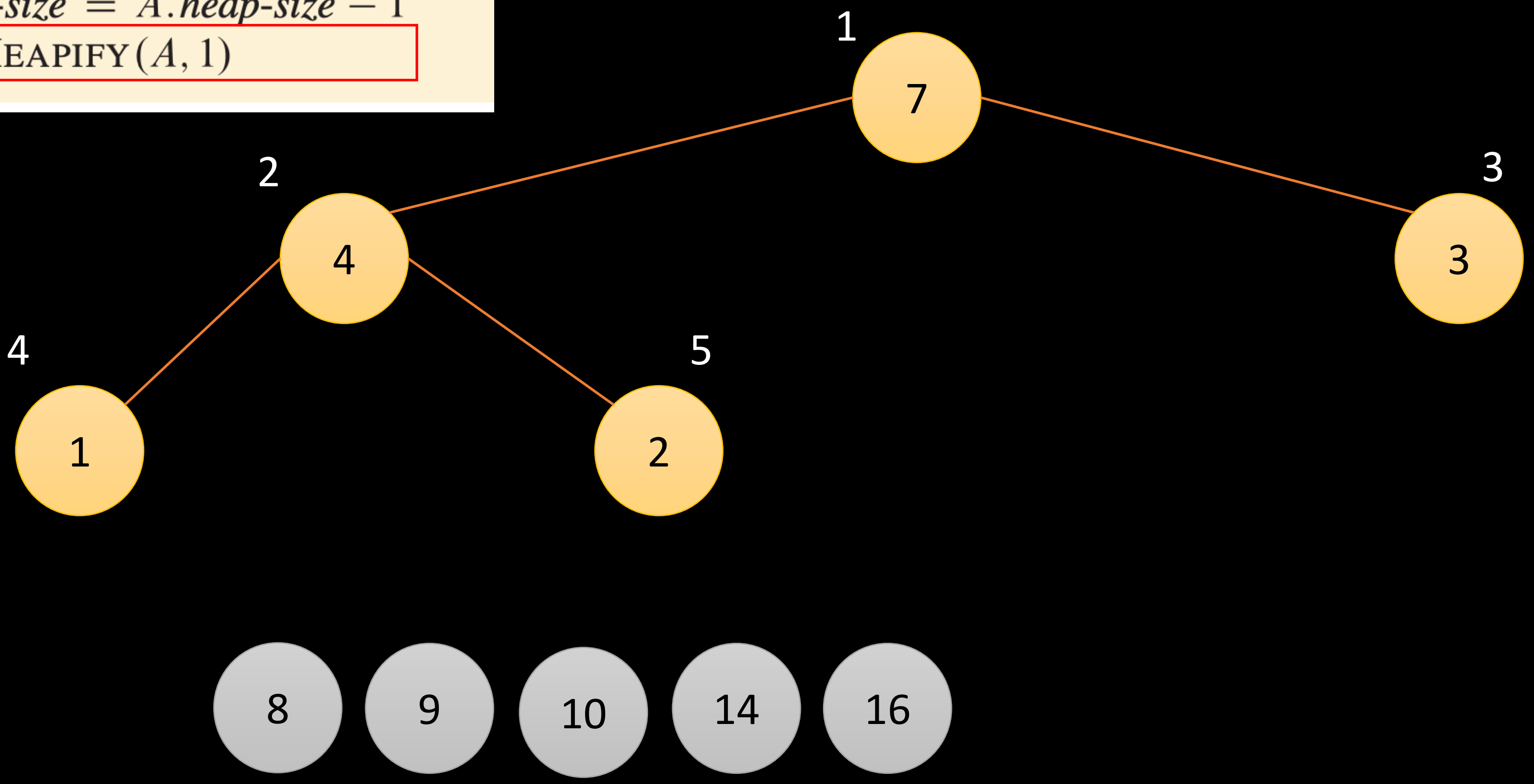
$$i = n = 6$$



The Heapsort Algorithm

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

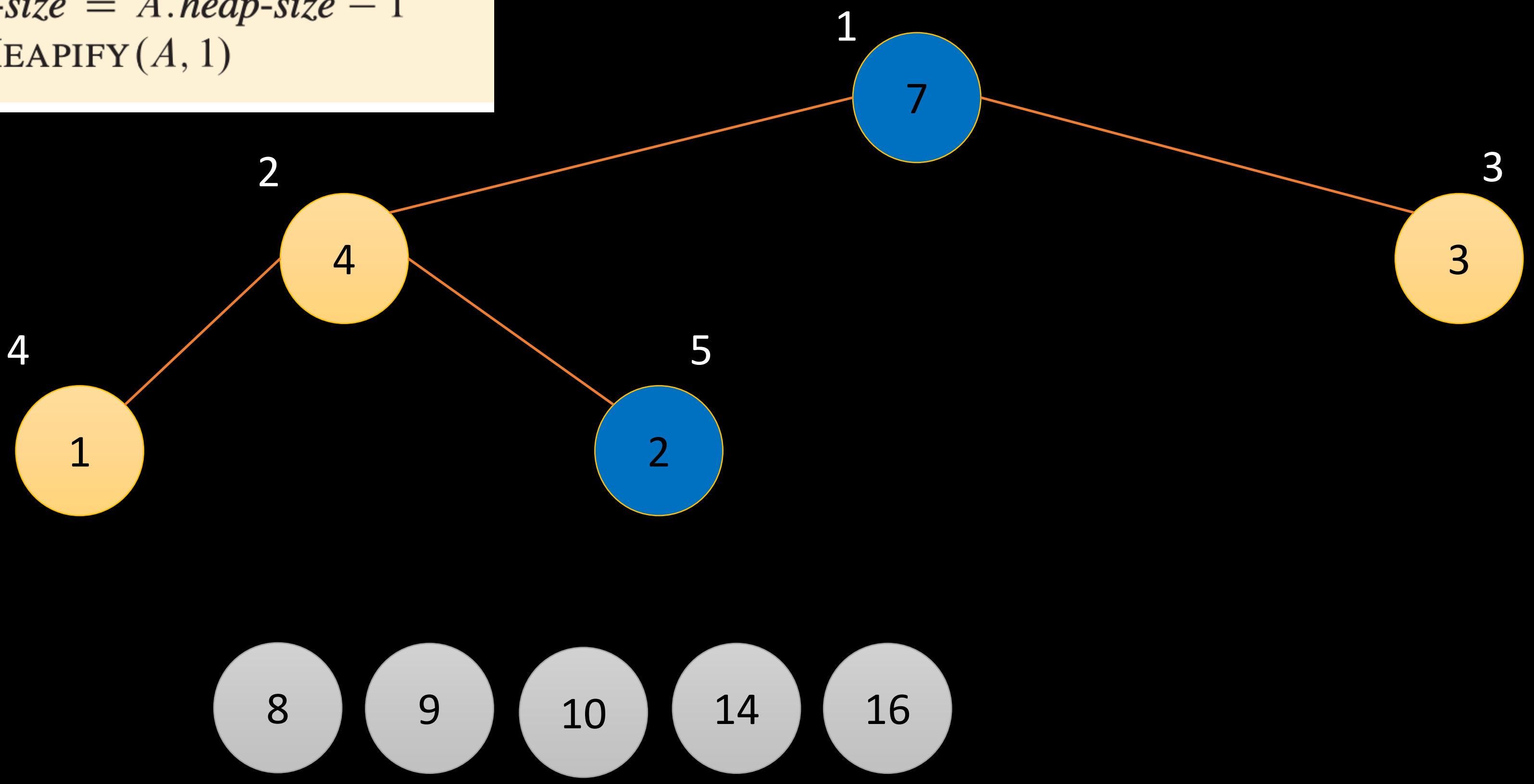
$$i = n = 6$$



The Heapsort Algorithm

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

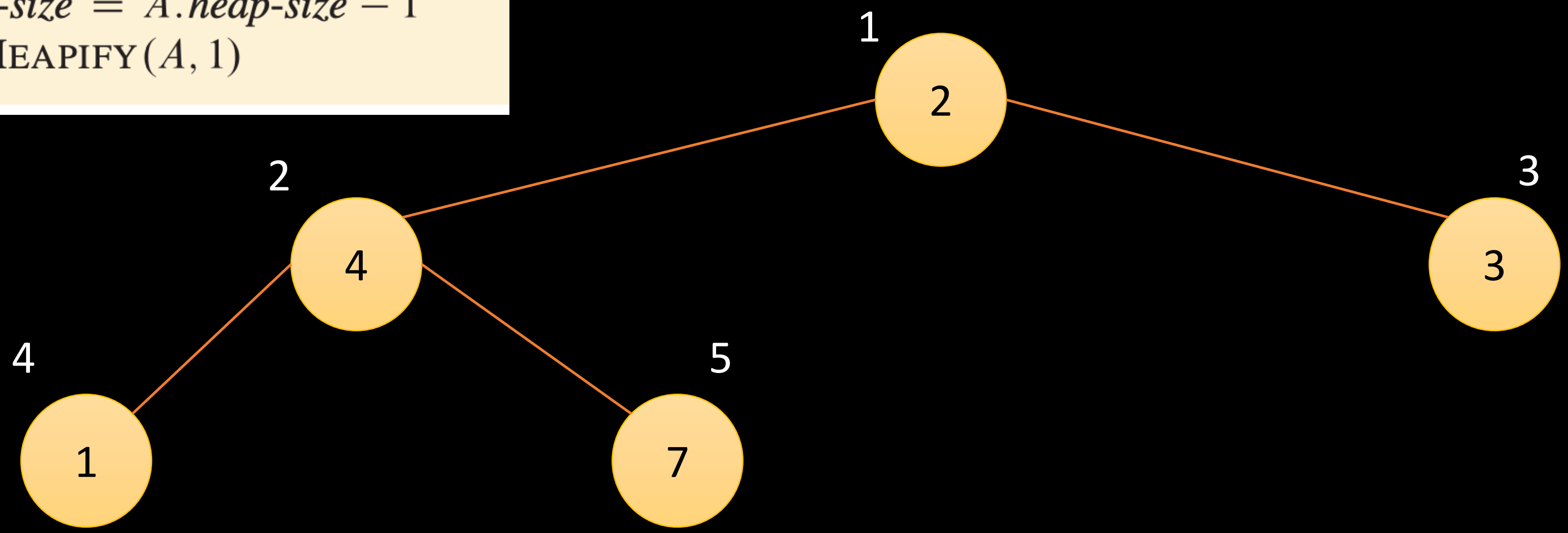
$$i = n = 5$$

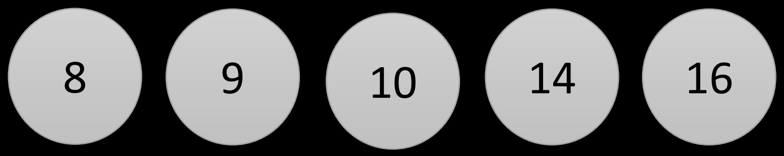


The Heapsort Algorithm

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

$$i = n = 5$$

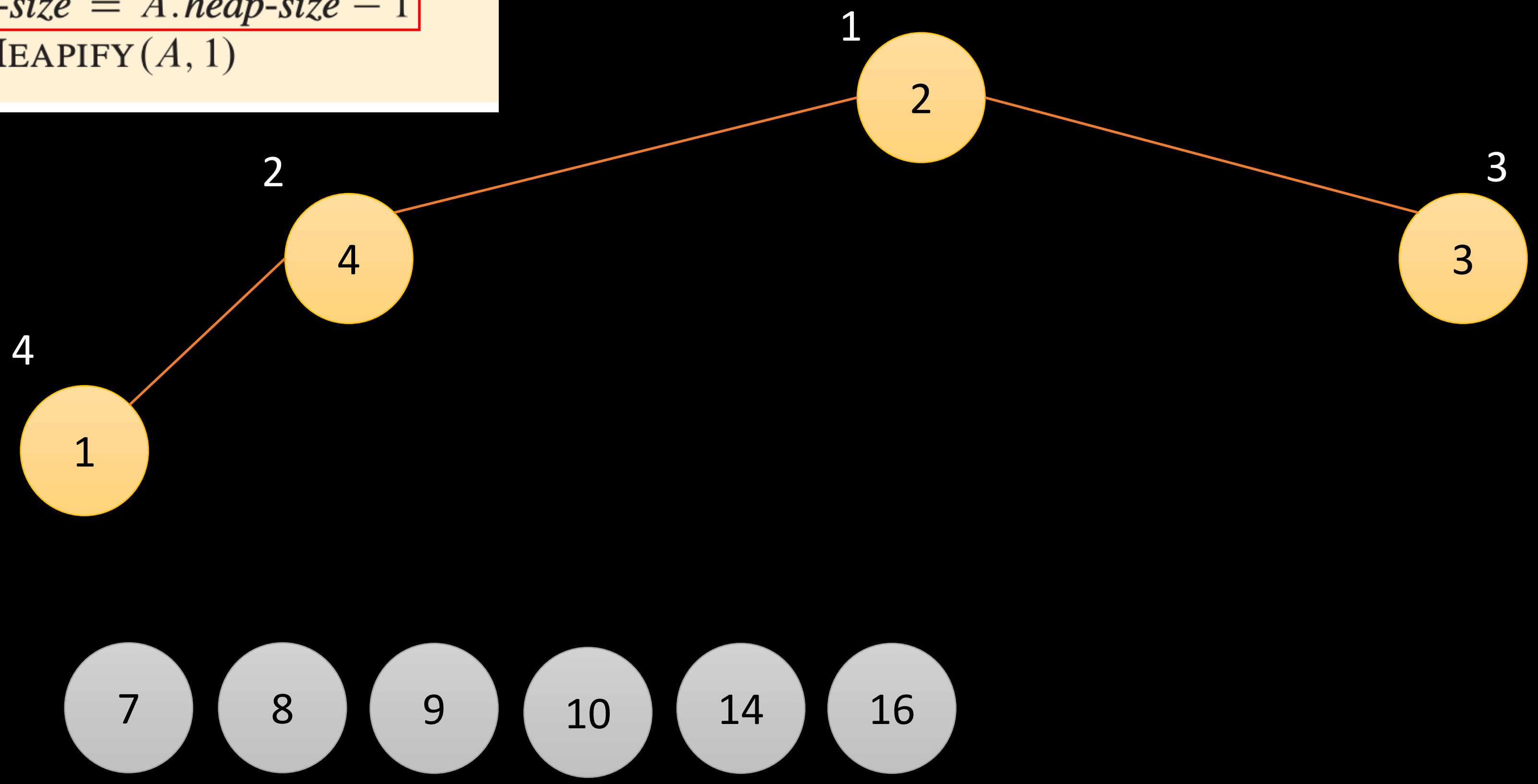




The Heapsort Algorithm

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

$$i = n = 5$$

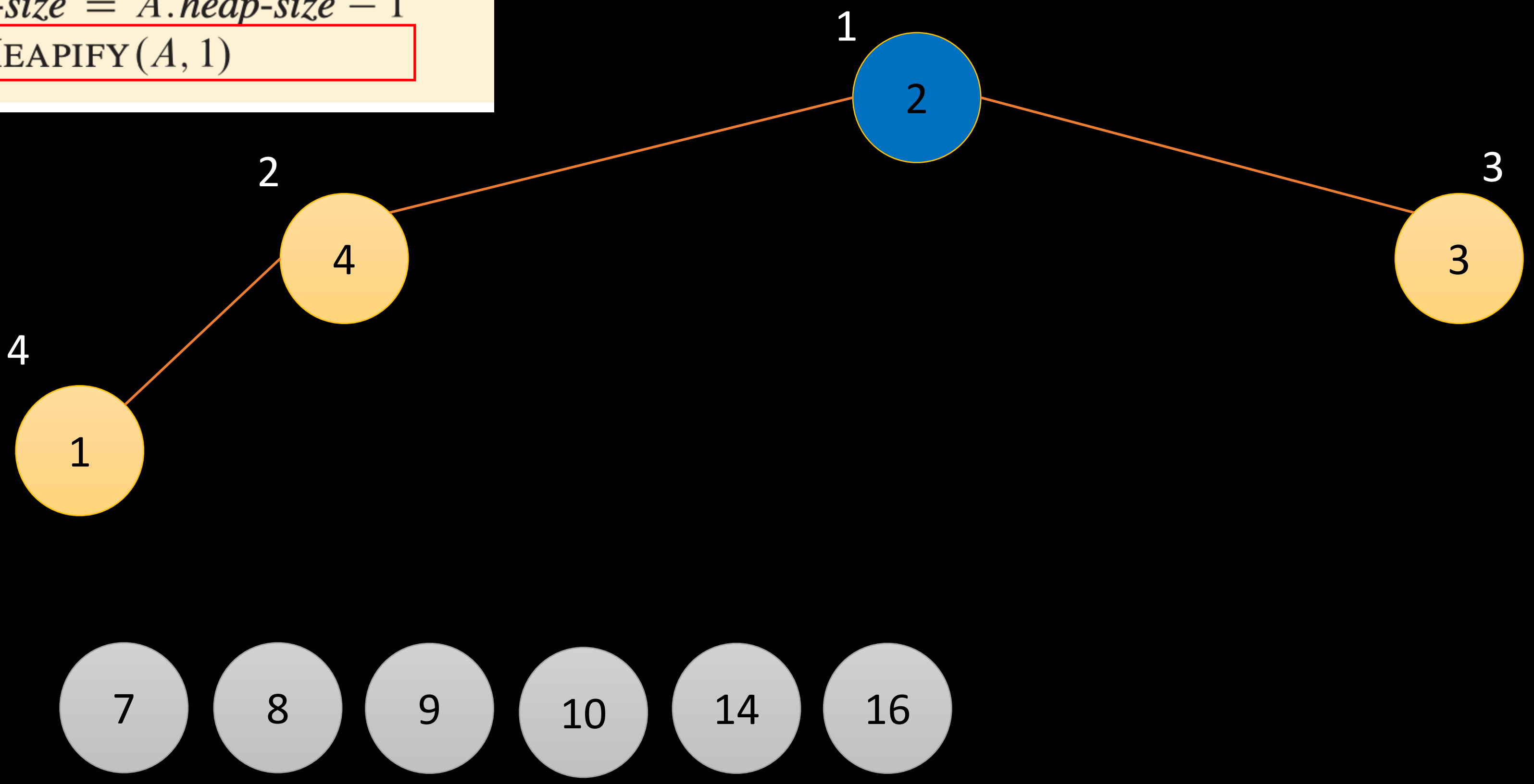


The Heapsort Algorithm

HEAPSORT(A, n)

BUILD-MAX-HEAP(A, n)
for $i = n$ **downto** 2
    exchange $A[1]$ with $A[i]$
    $A.heap\text{-}size = A.heap\text{-}size - 1$
    MAX-HEAPIFY$(A, 1)$

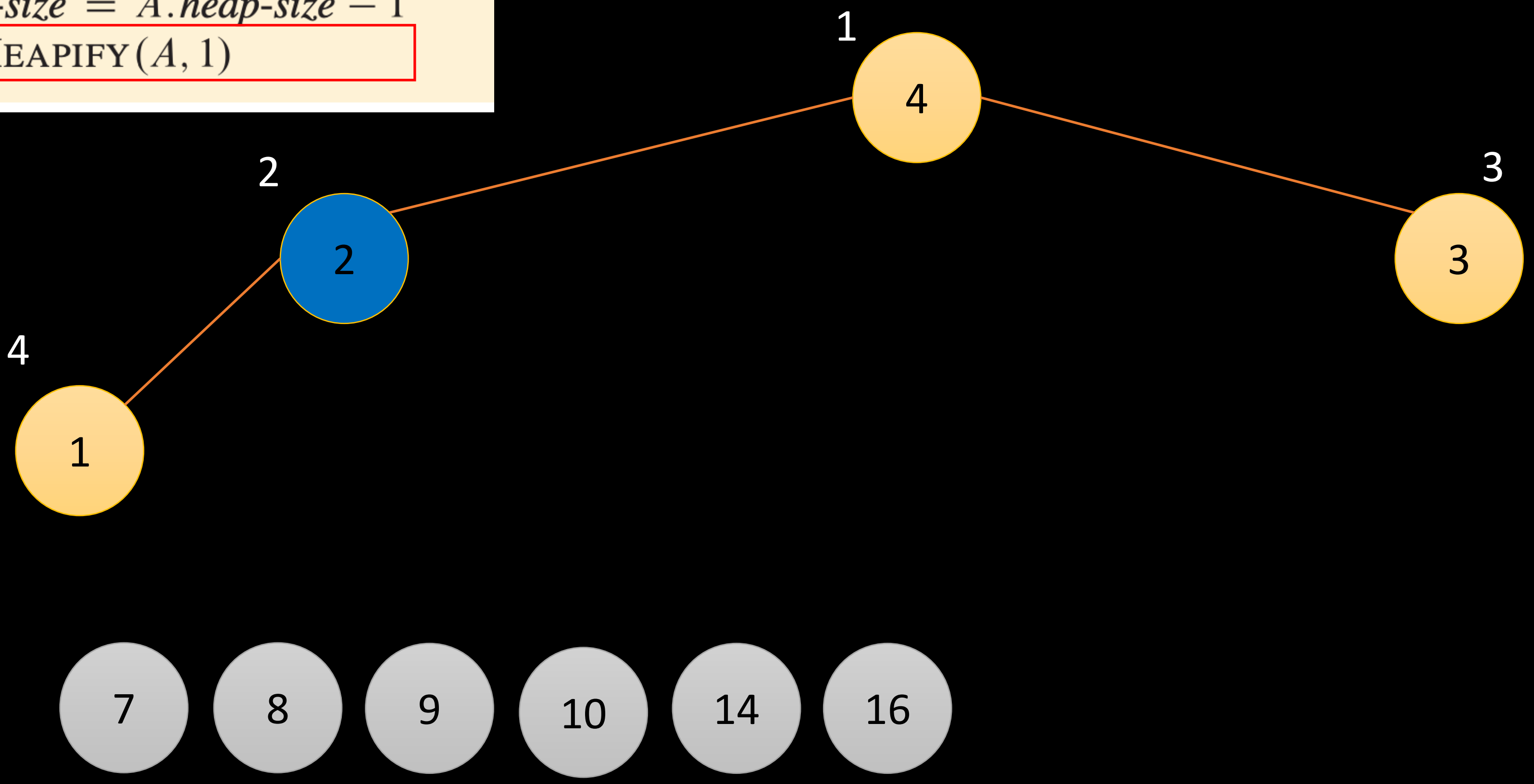
$$i = n = 5$$



The Heapsort Algorithm

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

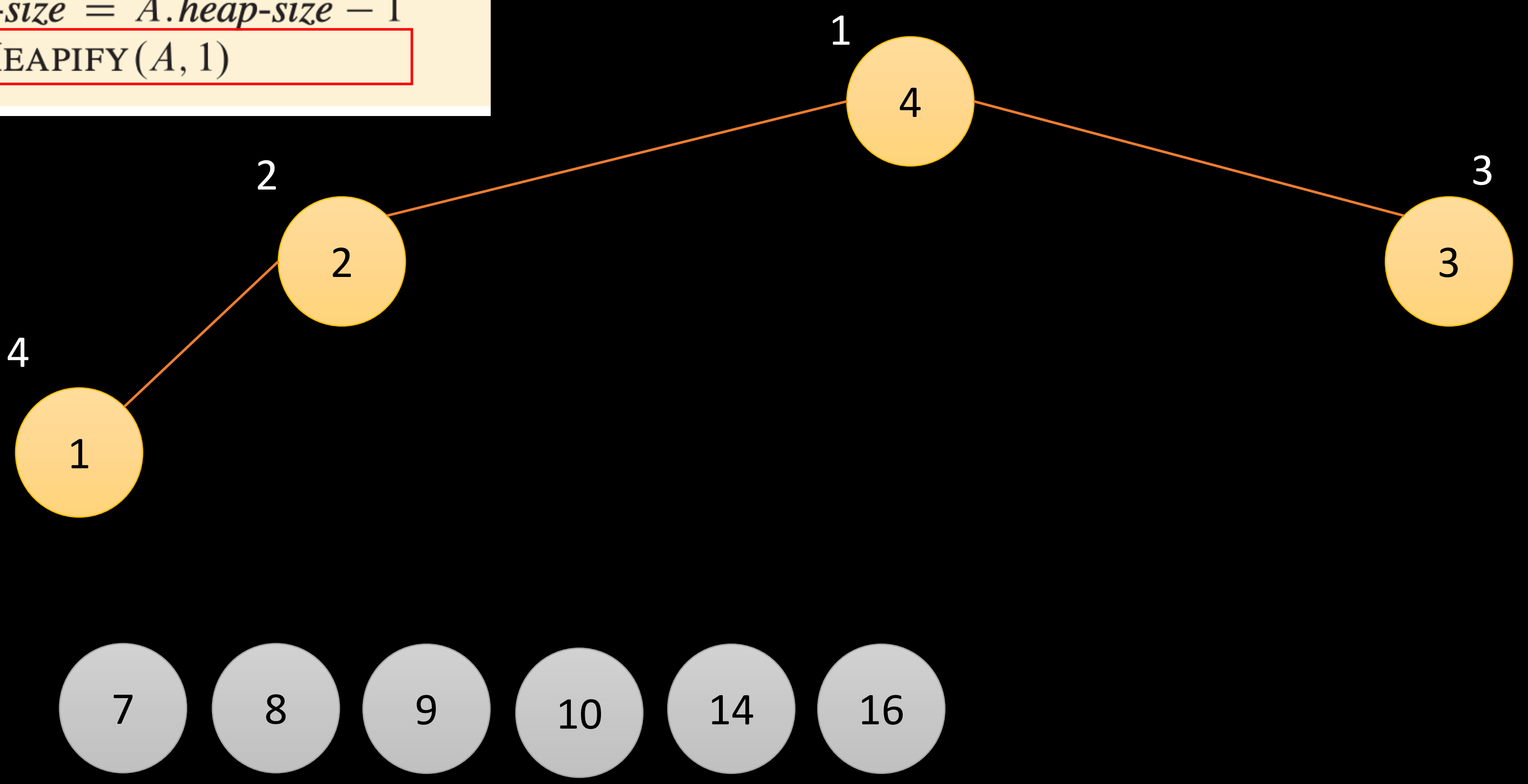
$$i = n = 5$$



The Heapsort Algorithm

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

$$i = n = 5$$

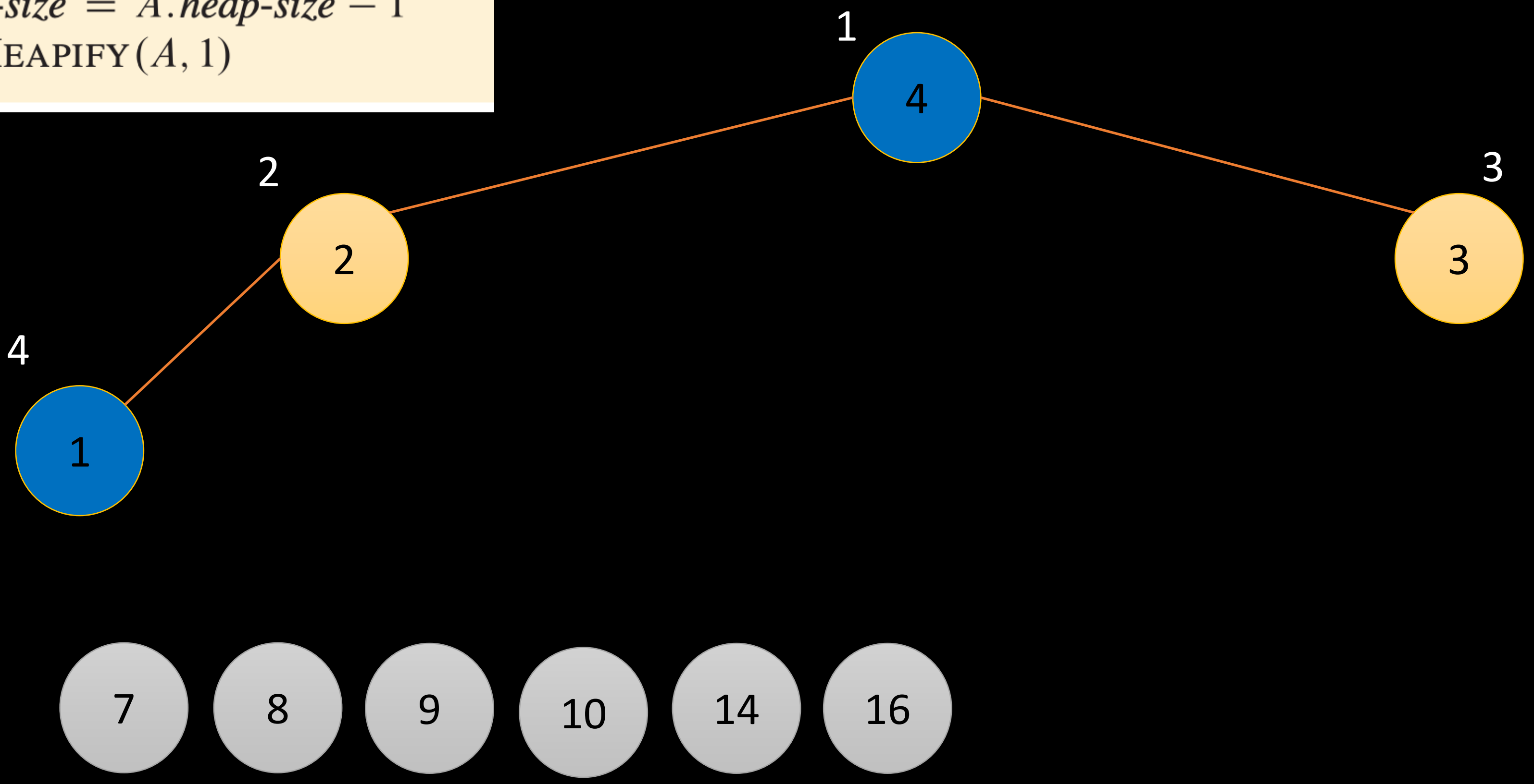


The Heapsort Algorithm

HEAPSORT(A, n)

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

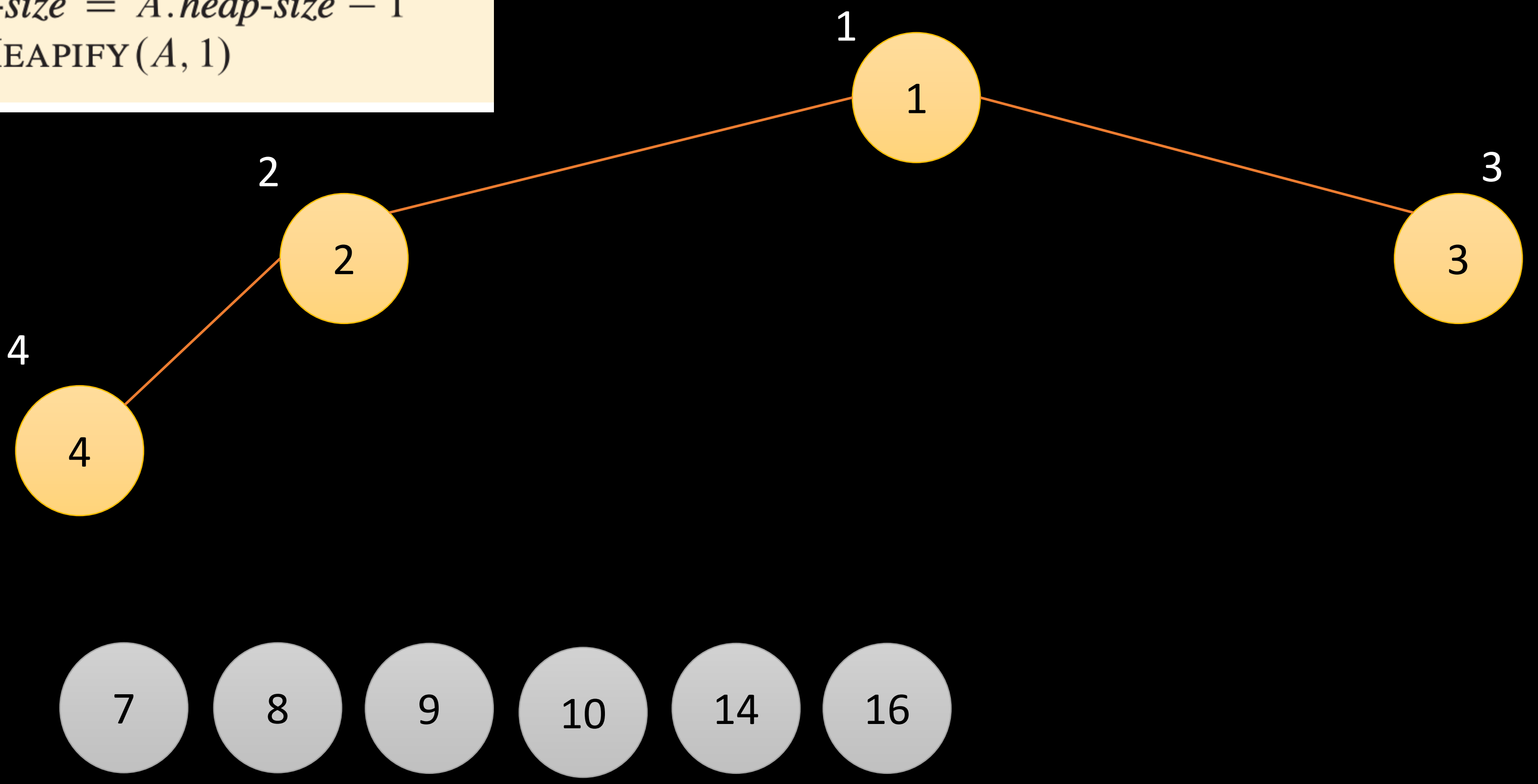
$$i = n = 4$$



The Heapsort Algorithm

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

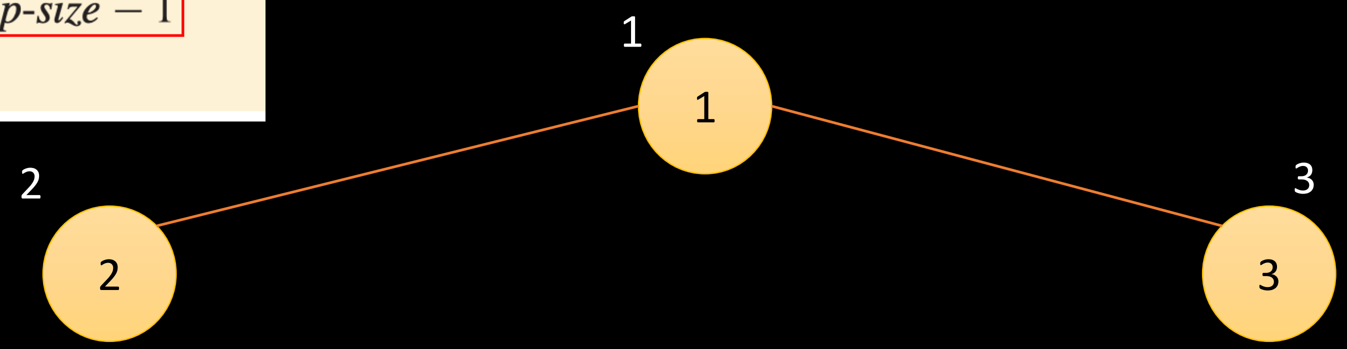
$$i = n = 4$$



The Heapsort Algorithm

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

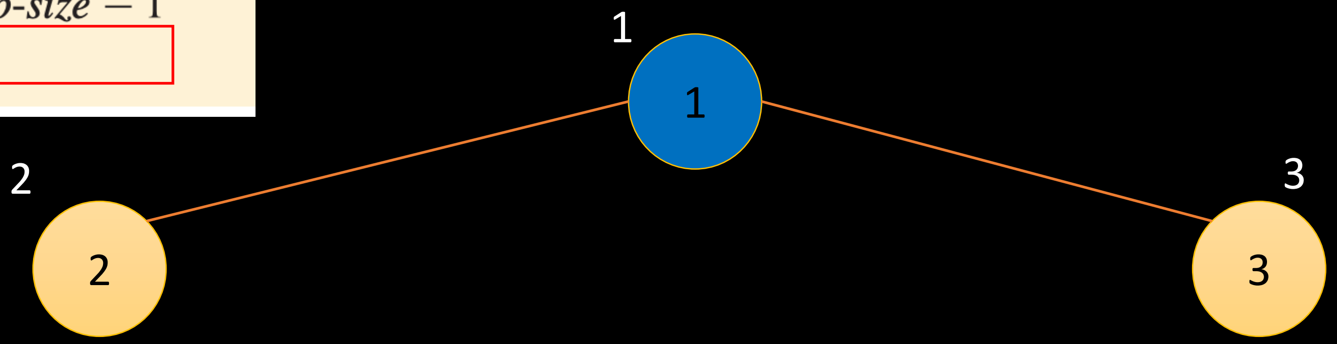
$i = n = 4$



The Heapsort Algorithm

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

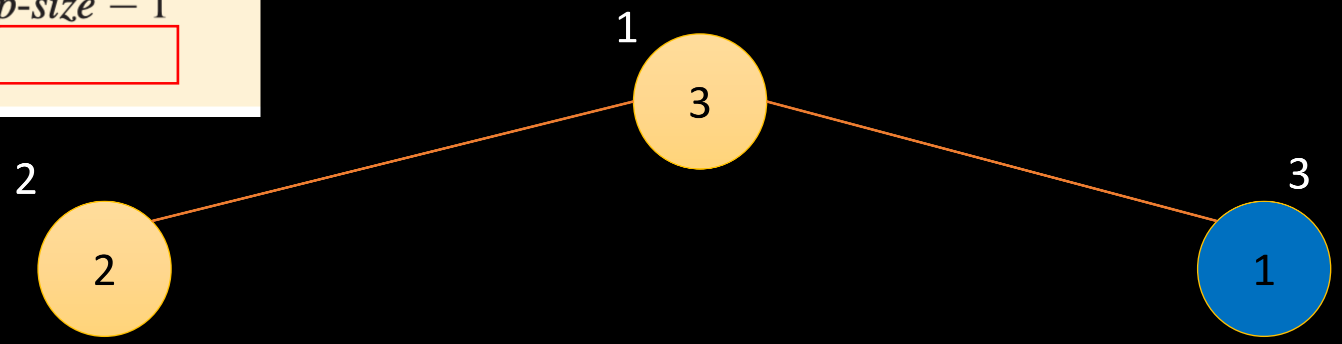
$$i = n = 4$$



The Heapsort Algorithm

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

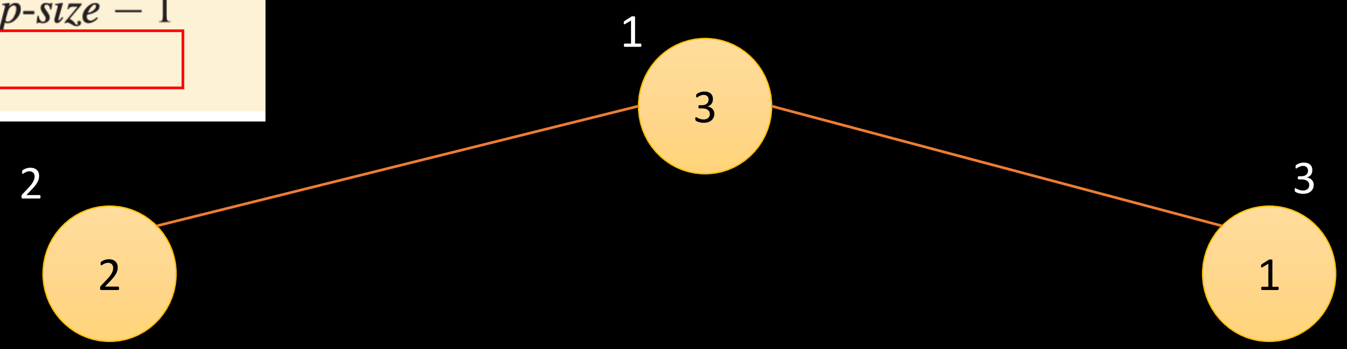
$$i = n = 4$$



The Heapsort Algorithm

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

$$i = n = 4$$

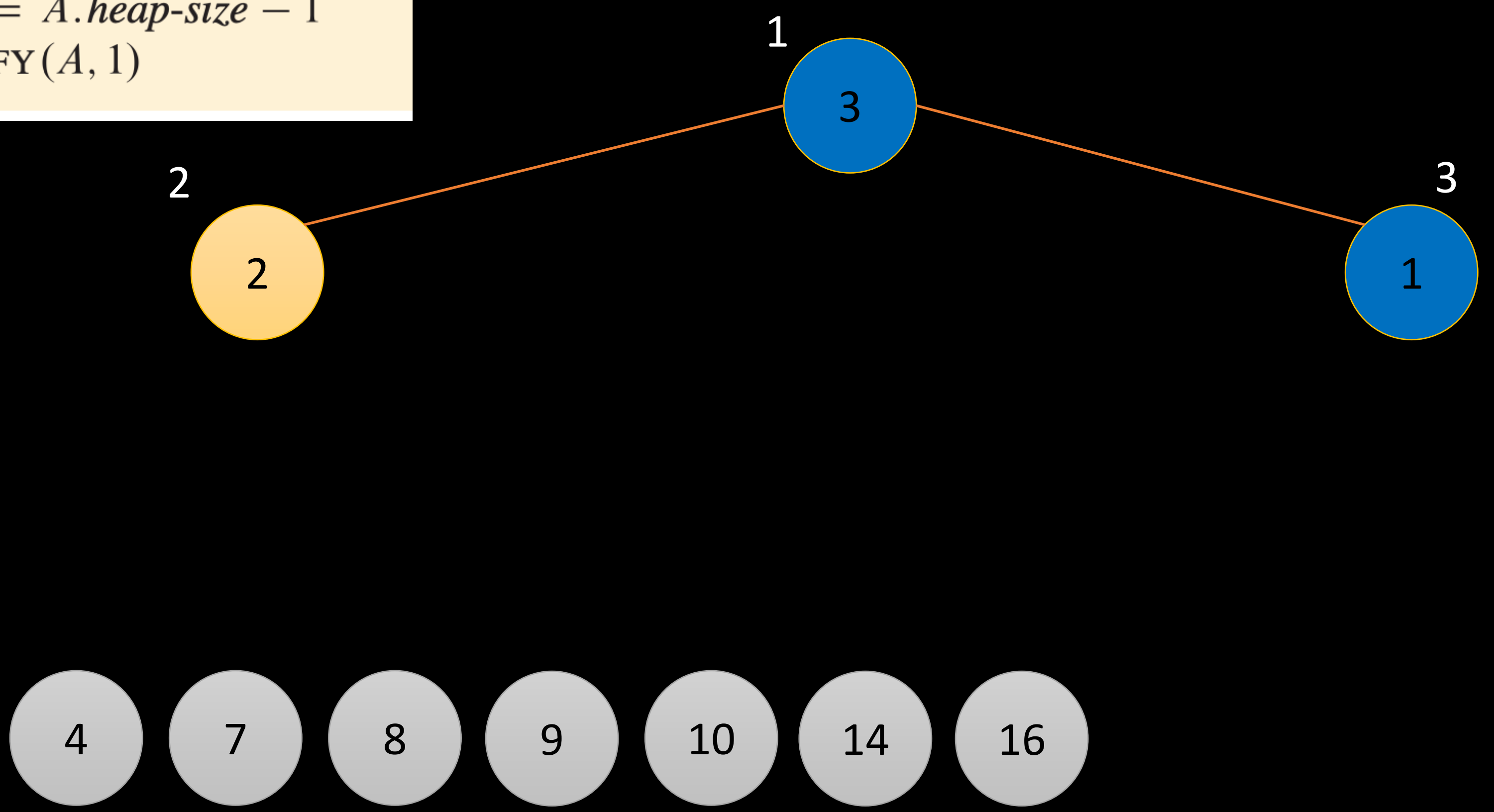


The Heapsort Algorithm

HEAPSORT(A, n)

BUILD-MAX-HEAP(A, n)
for $i = n$ **downto** 2
    exchange $A[1]$ with $A[i]$
    $A.heap\text{-}size = A.heap\text{-}size - 1$
    MAX-HEAPIFY$(A, 1)$

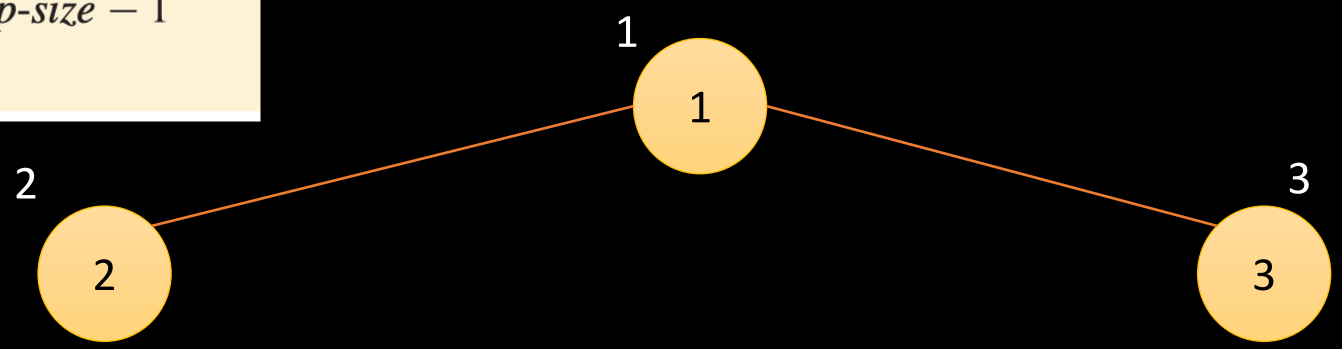
$i = n = 3$



The Heapsort Algorithm

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

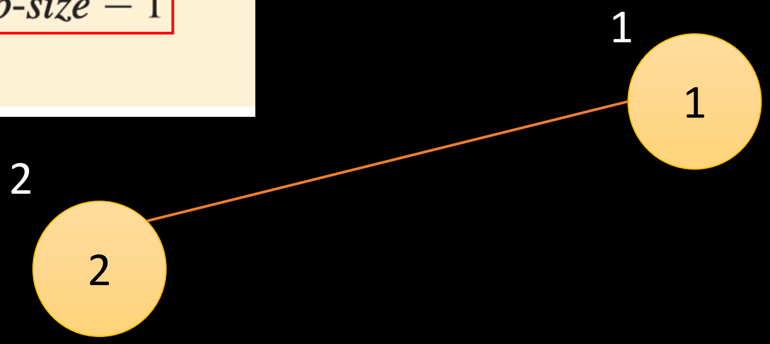
$$i = n = 3$$

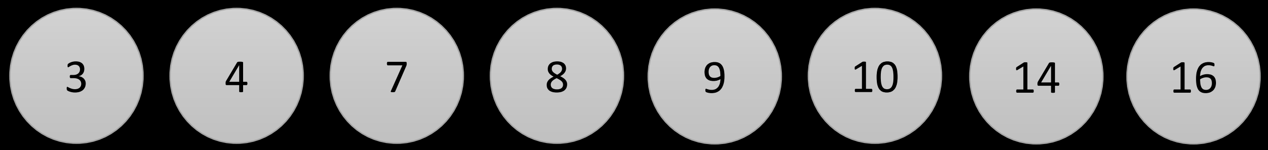


The Heapsort Algorithm

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

$$i = n = 3$$

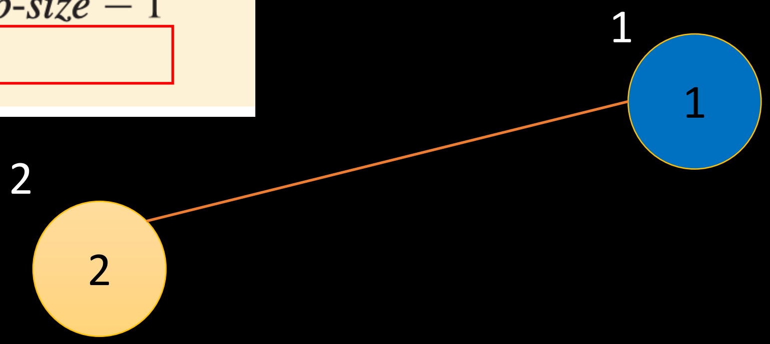


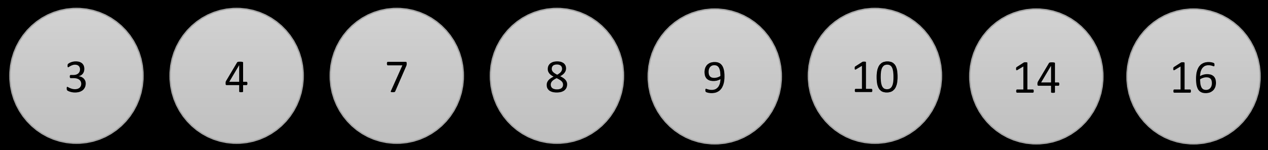


The Heapsort Algorithm

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

$i = n = 3$



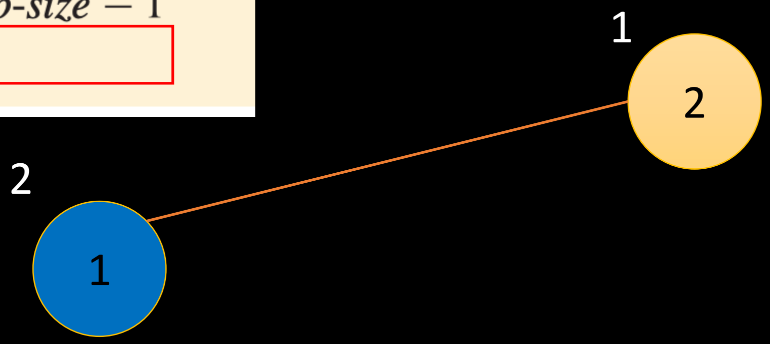


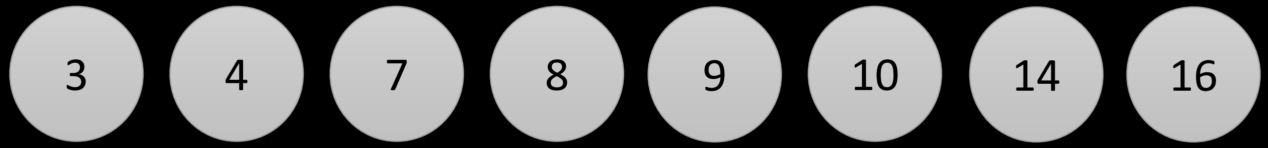
The Heapsort Algorithm

HEAPSORT(A, n)

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

$i = n = 3$

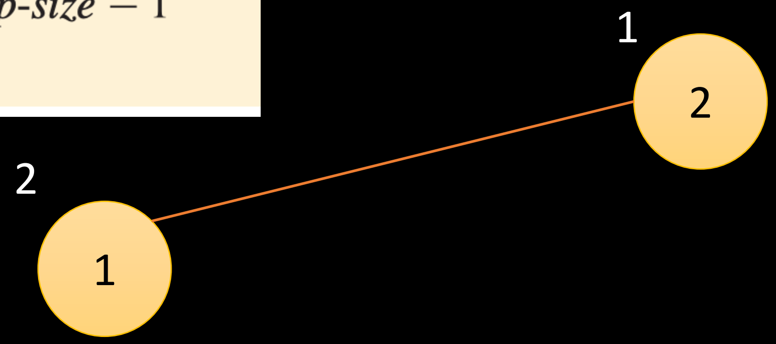


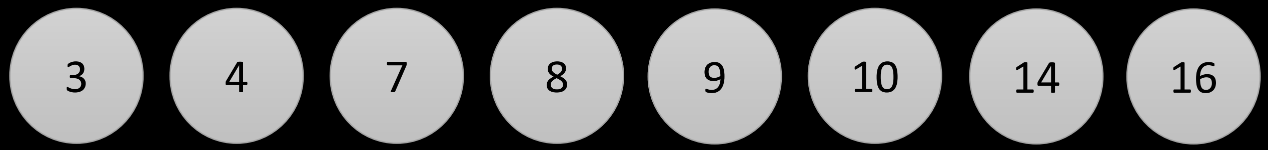


The Heapsort Algorithm

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

$$i = n = 2$$





The Heapsort Algorithm

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

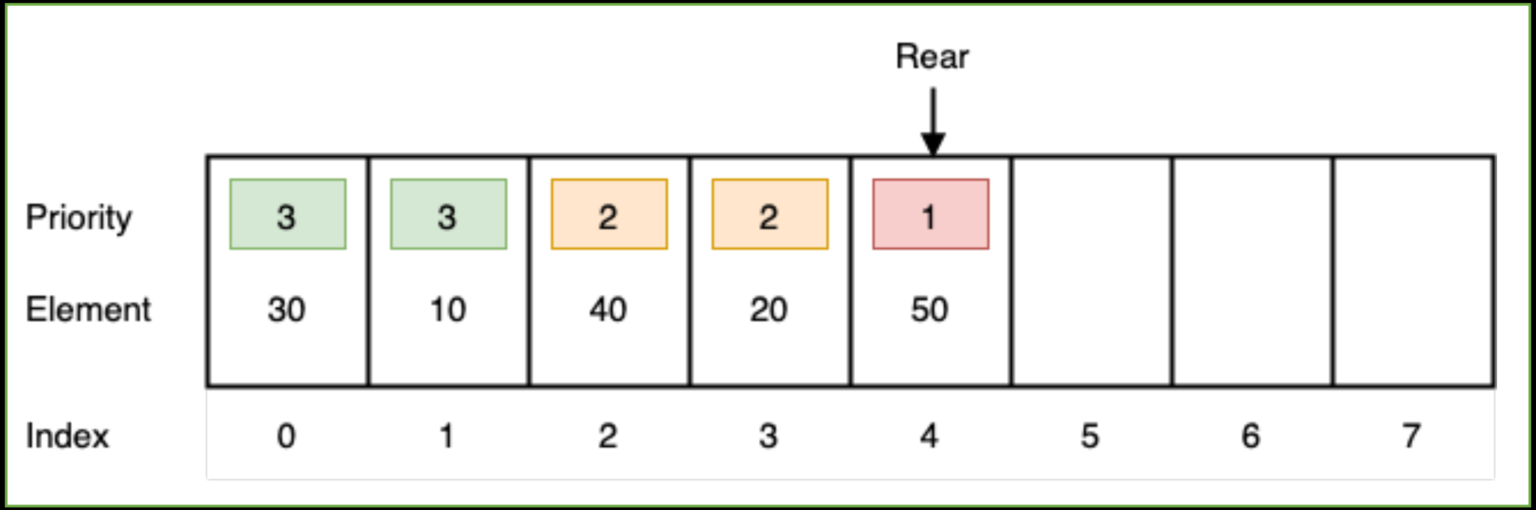
1 2 3 4 7 8 9 10 14 16

Content

Content
Introduction
Maintaining the Heap Property
Building a Heap
The Heapsort Algorithm
Priority Queue
Applications
Exercises

Priority Queue

- A type of queue that arranges elements based on their priority values.
- Each element in the queue has a **key**.
- A priority queue can be:
  - **Max-priority** queues – based on max-heaps.
  - **Min-priority** queues – based on min-heaps.



Priority Queue

- A max-priority queue supports the following operations:
  - $INSERT(S, x, k)$: inserts the element x with key k into the set S.
    - Equivalent to the operation $S \cup \{x\}$.
  - $MAXIMUM(S)$: returns the element of S with the largest key.
  - $EXTRACT - MAX(S)$: removes and returns the element of S with the largest key.
  - $INCREASE - KEY(S, x, k)$: increases x's key to the new value k
    - The new k is assumed to be at least as large as x's current key value.

Priority Queue

$MAXIMUM(A)$

- Takes $\Theta(1)$ time.

MAX-HEAP-MAXIMUM(A)

```
if A.heap-size < 1
    error "heap underflow"
return A[1]
```

Priority Queue

$EXTRACT - MAX(A)$

- Takes $O(\lg n)$ time.

```
MAX-HEAP-EXTRACT-MAX(A)
    max = MAX-HEAP-MAXIMUM(A)
    A[1] = A[A.heap-size]
    A.heap-size = A.heap-size − 1
    MAX-HEAPIFY(A, 1)
    return max
```

Priority Queue

$INCREASE - KEY(A, x, k)$

- Takes $O(\lg n)$ time.

```
MAX-HEAP-INCREASE-KEY(A, x, k)
if k < x.key
    error "new key is smaller than current key"
x.key = k
find the index i in array A where object x occurs
while i > 1 and A[PARENT(i)].key < A[i].key
    exchange A[i] with A[PARENT(i)], updating the information that maps
        priority queue objects to array indices
    i = PARENT(i)
```

Priority Queue

```
MAX-HEAP-INCREASE-KEY(A, x, k)
if k < x.key
    error "new key is smaller than current key"
x.key = k
find the index i in array A where object x occurs
while i > 1 and A[PARENT(i)].key < A[i].key
    exchange A[i] with A[PARENT(i)], updating the information that maps
        priority queue objects to array indices
    i = PARENT(i)
```

1. It first verifies that the new key k is larger than the current key.

Priority Queue

```
MAX-HEAP-INCREASE-KEY(A, x, k)
if k < x.key
    error "new key is smaller than current key"
x.key = k
find the index i in array A where object x occurs
while i > 1 and A[PARENT(i)].key < A[i].key
    exchange A[i] with A[PARENT(i)], updating the information that maps
        priority queue objects to array indices
    i = PARENT(i)
```

1. It first verifies that the new key k is larger than the current key.
2. Assigns the new key to x if it is valid.

Priority Queue

```
MAX-HEAP-INCREASE-KEY(A, x, k)
if k < x.key
    error "new key is smaller than current key"
x.key = k
find the index i in array A where object x occurs
while i > 1 and A[PARENT(i)].key < A[i].key
    exchange A[i] with A[PARENT(i)], updating the information that maps
        priority queue objects to array indices
    i = PARENT(i)
```

1. It first verifies that the new key k is larger than the current key.
2. Assigns the new key to x if it is valid.
3. Find the index i such that $A[i] = x$

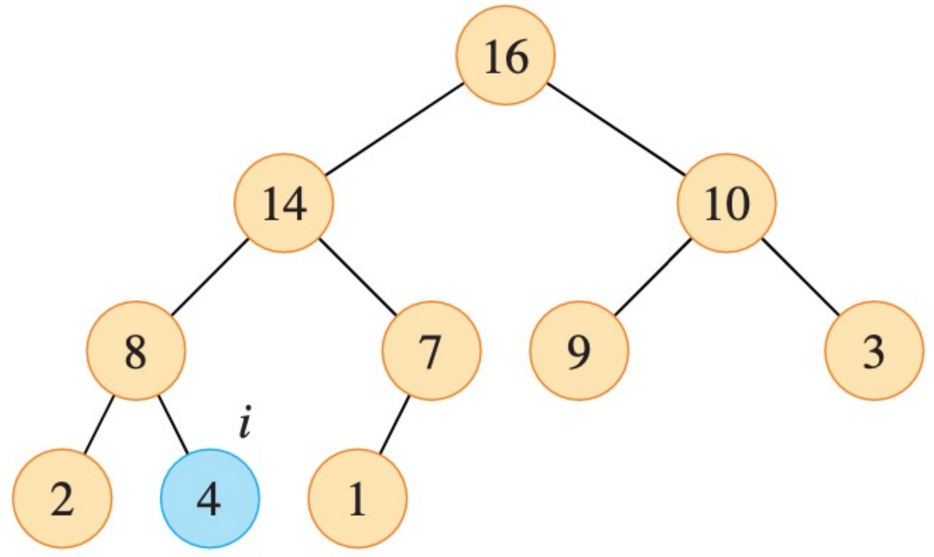
Priority Queue

```
MAX-HEAP-INCREASE-KEY(A, x, k)
if k < x.key
    error "new key is smaller than current key"
x.key = k
find the index i in array A where object x occurs
while i > 1 and A[PARENT(i)].key < A[i].key
    exchange A[i] with A[PARENT(i)], updating the information that maps
        priority queue objects to array indices
    i = PARENT(i)
```

1. It first verifies that the new key k is larger than the current key.
2. Assigns the new key to x if it is valid.
3. Find the index i such that $A[i] = x$
4. Traverse from the current node toward the root to find a proper place for the newly increased key.

Priority Queue

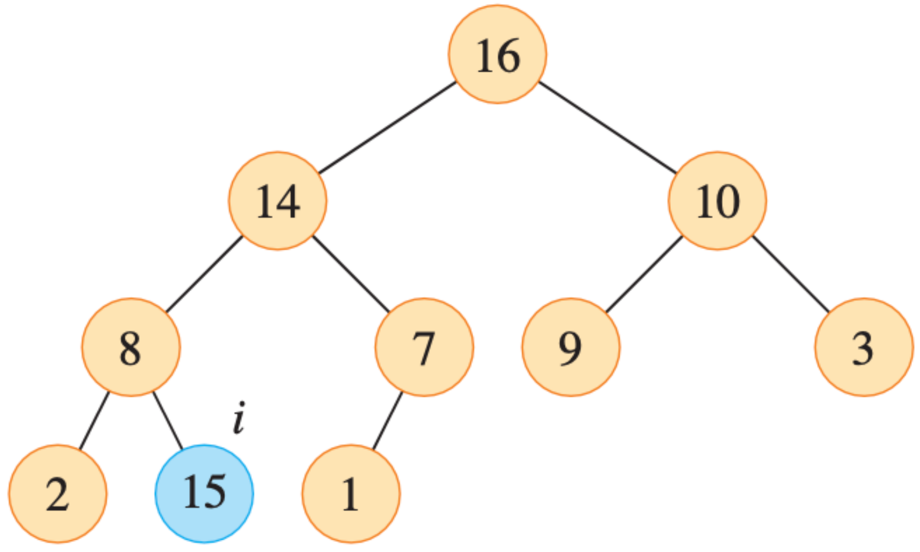
$INCREASE-KEY(A,4,15)$



(a)

Priority Queue

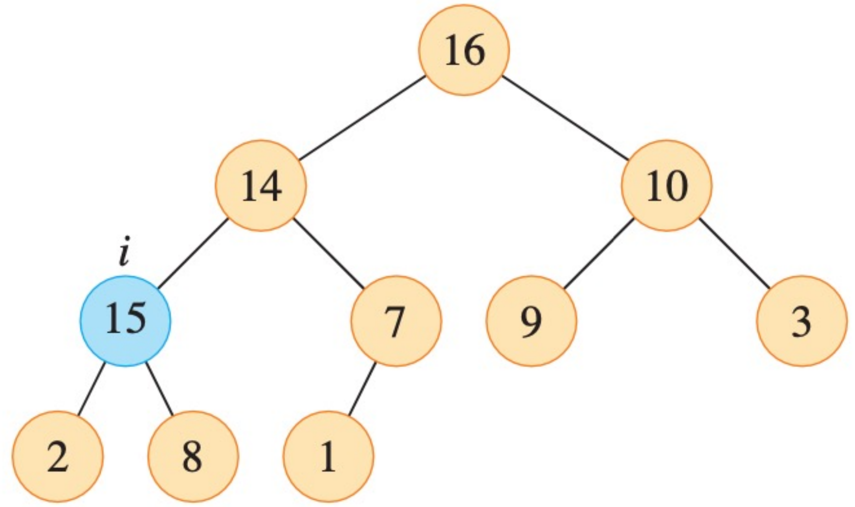
$INCREASE - KEY(A, 4, 15)$



(b)

Priority Queue

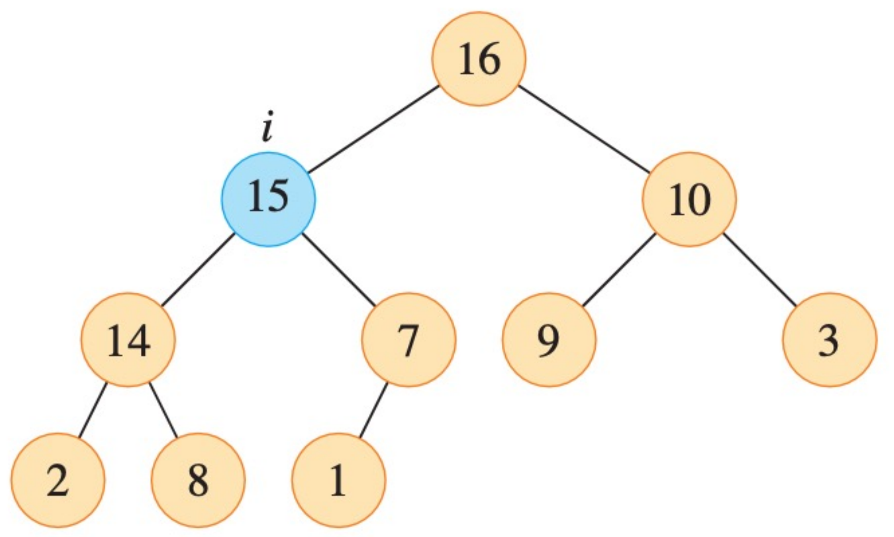
$INCREASE - KEY(A, 4, 15)$



(c)

Priority Queue

$INCREASE - KEY(A, 4, 15)$



(d)

Priority Queue

$INSERT(A, x, n)$ – assume that x has attribute key

- Takes $O(\lg n)$

MAX-HEAP-INSERT(A, x, n)

```
if A.heap-size == n
    error "heap overflow"
A.heap-size = A.heap-size + 1
k = x.key
x.key = −∞
A[A.heap-size] = x
map x to index heap-size in the array
MAX-HEAP-INCREASE-KEY(A, x, k)
```

Priority Queue

```
MAX-HEAP-INSERT(A, x, n)
if A.heap-size == n
    error "heap overflow"
A.heap-size = A.heap-size + 1
k = x.key
x.key = −∞
A[A.heap-size] = x
map x to index heap-size in the array
MAX-HEAP-INCREASE-KEY(A, x, k)
```

1. Verifies that the array has room for the new element.

Priority Queue

```
MAX-HEAP-INSERT(A, x, n)
if A.heap-size == n
    error "heap overflow"
A.heap-size = A.heap-size + 1
k = x.key
x.key = -∞
A[A.heap-size] = x
map x to index heap-size in the array
MAX-HEAP-INCREASE-KEY(A, x, k)
```

1. Verifies that the array has room for the new element.
2. Expands the max-heap by adding to the tree a new leaf whose key is $-\infty$.

Priority Queue

```
MAX-HEAP-INSERT(A, x, n)
if A.heap-size == n
    error "heap overflow"
A.heap-size = A.heap-size + 1
k = x.key
x.key = −∞
A[A.heap-size] = x
map x to index heap-size in the array
MAX-HEAP-INCREASE-KEY(A, x, k)
```

1. Verifies that the array has room for the new element.
2. Expands the max-heap by adding to the tree a new leaf whose key is $-\infty$.
3. Get the key associated with x

Priority Queue

```
MAX-HEAP-INSERT(A, x, n)
if A.heap-size == n
    error "heap overflow"
A.heap-size = A.heap-size + 1
k = x.key
x.key = −∞
A[A.heap-size] = x
map x to index heap-size in the array
MAX-HEAP-INCREASE-KEY(A, x, k)
```

1. Verifies that the array has room for the new element.
2. Expands the max-heap by adding to the tree a new leaf whose key is $-\infty$.
3. Get the key associated with x.
4. Add x to the end of the heap.

Priority Queue

MAX-HEAP-INSERT(A, x, n)

if $A.heap\text{-}size == n$
    error "heap overflow"
$A.heap\text{-}size = A.heap\text{-}size + 1$
$k = x.key$
$x.key = -\infty$
$A[A.heap\text{-}size] = x$
map x to index *heap-size* in the array
$\boxed{\text{MAX-HEAP-INCREASE-KEY}(A, x, k)}$

1. Verifies that the array has room for the new element.
2. Expands the max-heap by adding to the tree a new leaf whose key is $-\infty$.
3. Get the key associated with x.
4. Add x to the end of the heap.
5. Set the key of the new element to its correct value by increasing the $-\infty$.

Content

Content
Introduction
Maintaining the Heap Property
Building a Heap
The Heapsort Algorithm
Priority Queue
Applications
Exercises

Applications

- Applications of heaps:
  - **Priority Queues**: the heap structure guarantees constant-time extraction of the element with the highest priority.

  - **Dijkstra's Shortest Path Algorithm**: use heaps to determine the shortest path between two nodes.

  - **Memory management**: OSs use heap trees to allocate memory and dynamically guarantee the best use of resources.

  - **Data Compression**: heap trees are used in different data compression and encoding schemes (e.g., Huffman coding).

Applications

- Applications of priority queues:
  - **Job scheduling**: schedule jobs on a computer shared among multiple users.
  - **Process scheduling**: processes with highest priority allocated the resources first.
  - **Event-driven simulations**: executing an event at a precise time according to the event's priority. Some events may affect future events. Examples: computer networks and scientific research.
  - **Graph algorithms**: Prim's algorithm for finding the minimum spanning tree.
    - MST is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight.

Content

Content
Introduction
Maintaining the Heap Property
Building a Heap
The Heapsort Algorithm
Priority Queue
Applications
Exercises

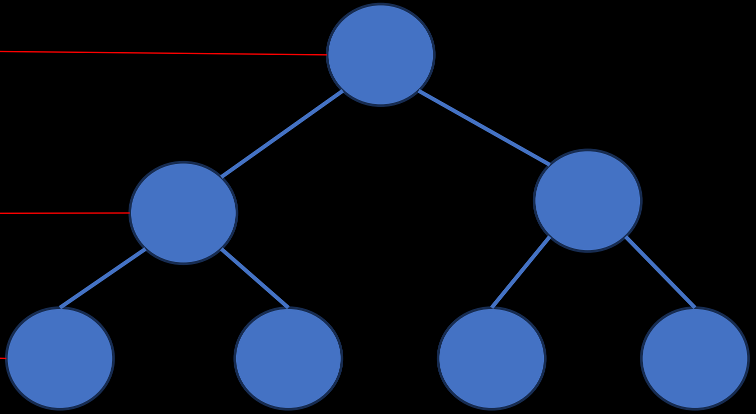
Exercises

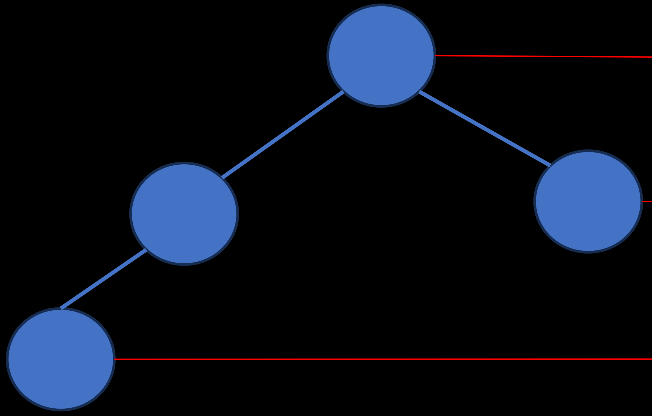
- What are the minimum and maximum numbers of elements in a heap of height h?

Exercises

- What are the minimum and maximum numbers of elements in a heap of height h? $h = \lg n$

Height 0 → 1 node
Height 1 → 3 node
Height 2 → 7 node





Height 0 → 1 node
Height 1 → 3 node
Height 2 → 4 node

Max number of elements: $2^{h+1} - 1$

Min number of elements: 2^h

Exercises

- Is the array with values [33;19;20;15;13;10;2;13;16;12] a max-heap?

Exercises

- Is the array with values [33;19;20;15;13;10;2;13;16;12] a max-heap?

33 is the root.

$left(1) = 2 * 1 = 2$→ 19 is the left child.

$right(1) = 2 * 1 + 1 = 3$→20 is the right child.

$left(2) = 2 * 2 = 4$→ 15 is the left child of 19.

$right(2) = 2 * 2 + 1 = 5$→13 is the right child of 19.

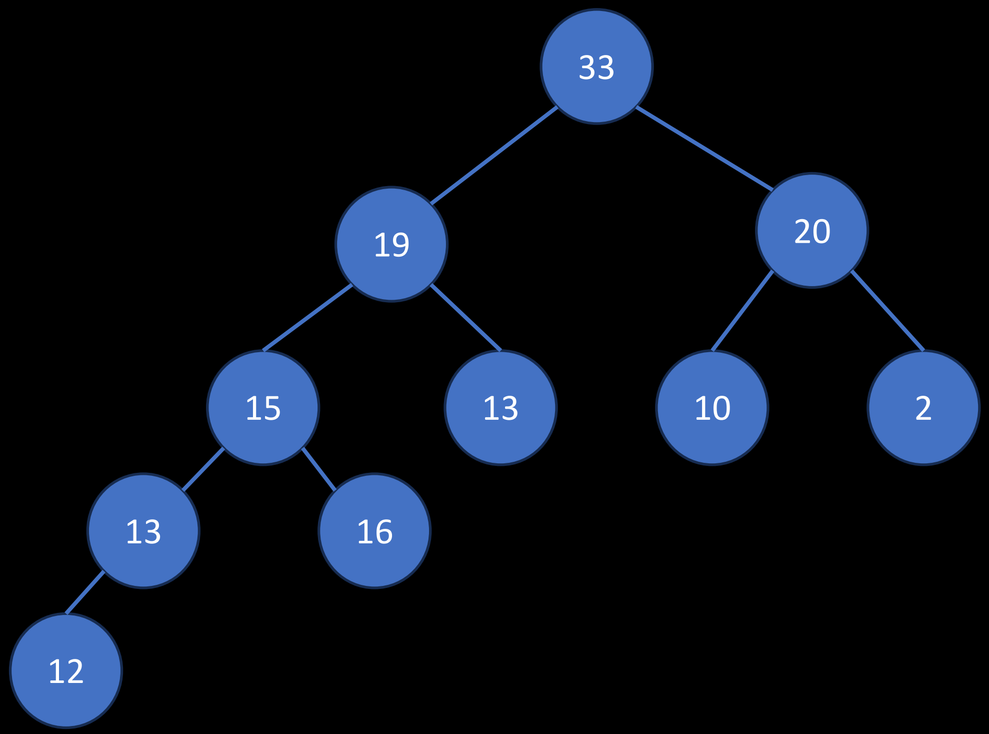
$left(3) = 3 * 2 = 6$→10 is the left child of 20.

$right(3) = 3 \ * 2 + 1 = 7$→ 2 is the right child of 20.

$right(4) = 2 * 4 = 8$→13 is the left child of 15.

$right(4) = 2 * 4 + 1$→16 is the right child of 15.

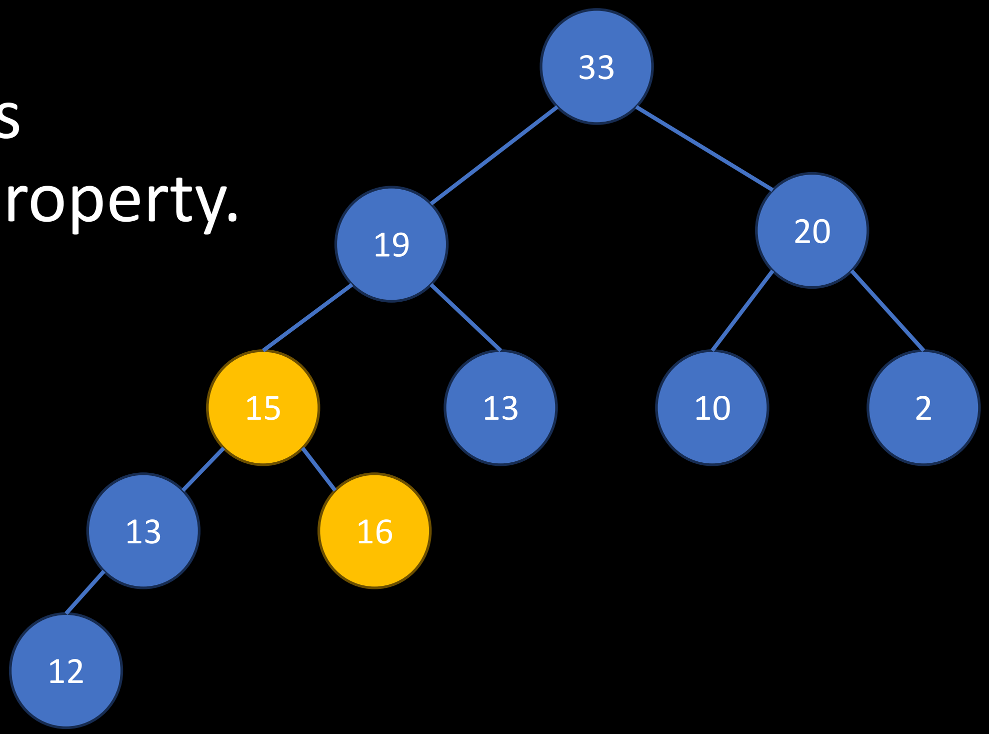
….



Exercises

- Is the array with values [33;19;20;15;13;10;2;13;16;12] a max-heap?

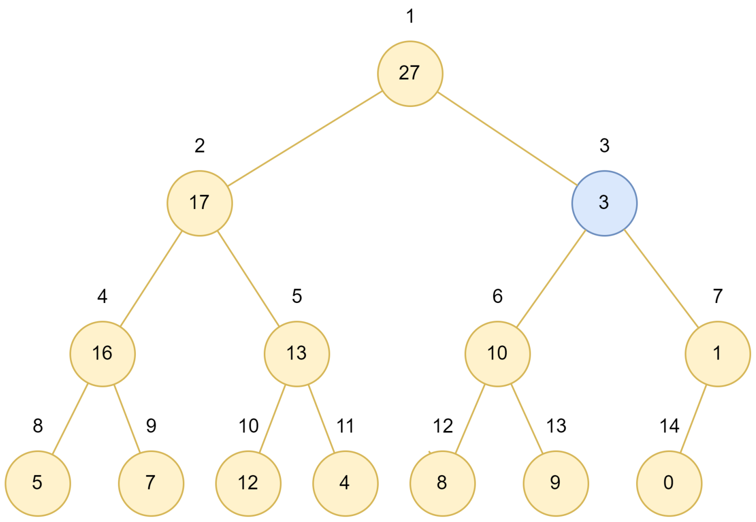
It is NOT a max-heap because node 15 has
a right child 16, which violates the heap property.



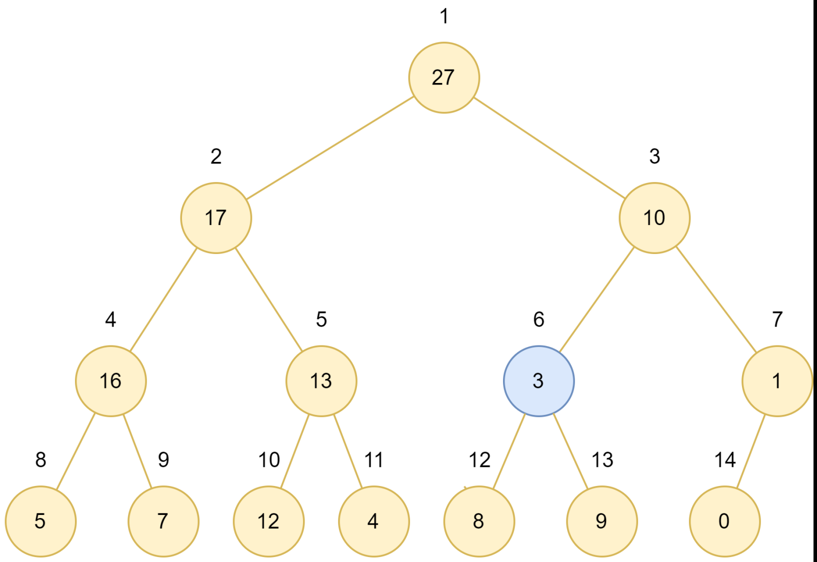
Exercises

- Illustrate the operation of MAX-HEAPIFY(A, 3) on the array A= [27;17;3;16;13;10;1;5;7;12;4;8;9;0]

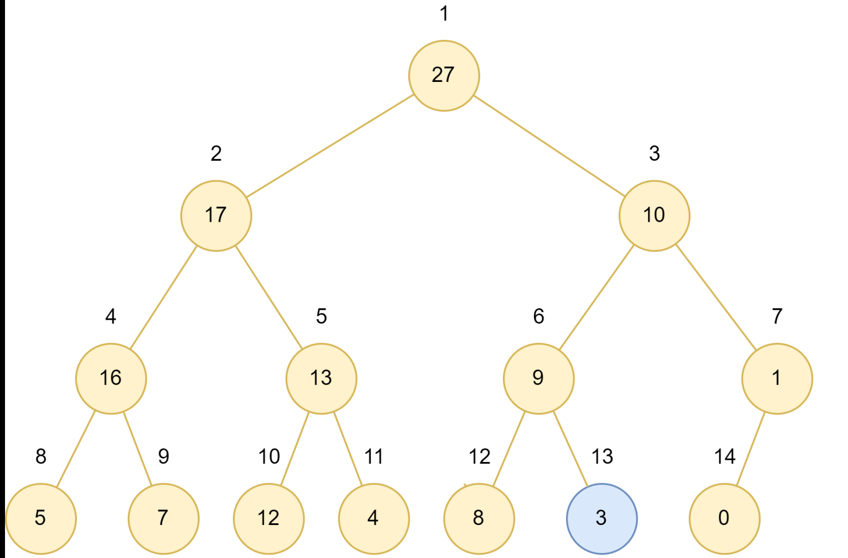
Exercises



a



b



c

Exercises

- What is the effect of calling MAX-HEAPIFY(A, i) when the element $A[i]$ is larger than its children?

Exercises

- What is the effect of calling MAX-HEAPIFY(A, i) when the element $A[i]$ is larger than its children?

No effect. A[i] is found to be largest, and the procedure just returns.

Exercises

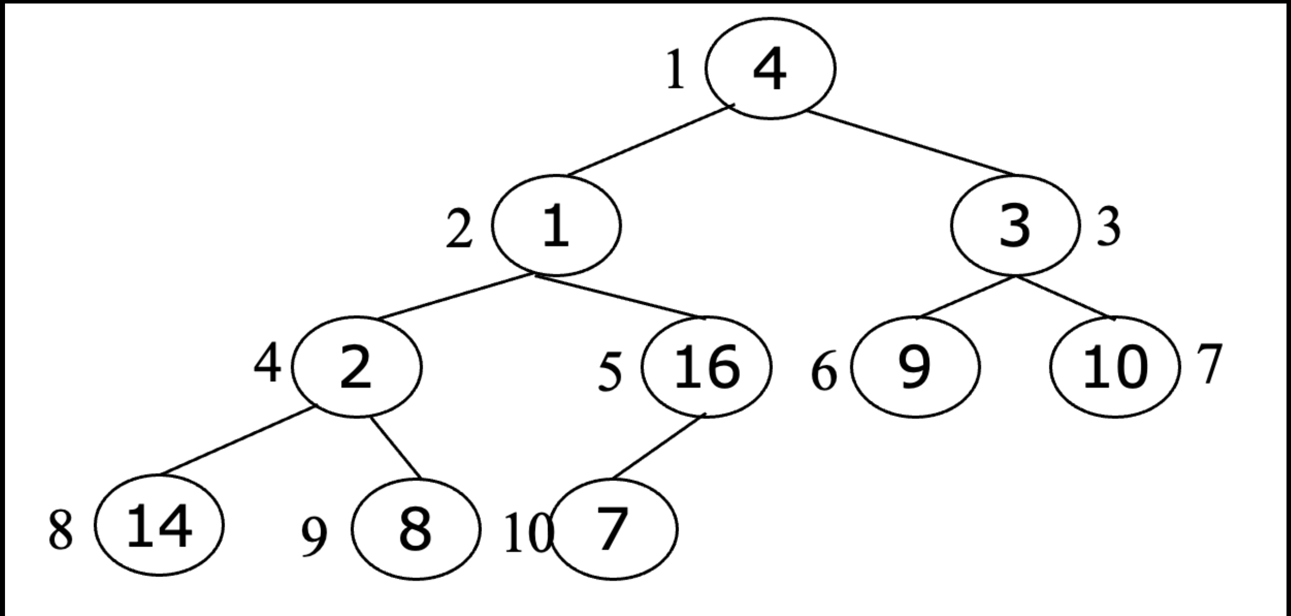
- What is the effect of calling MAX-HEAPIFY(A,i) for $i \ > \ A.heapSize/2$.

Exercises

Nodes at index $i > A.heapSize/2$ are either leaf nodes or nodes that have only one child at most.

So, if it is a leaf node, the procedure will have no effect.

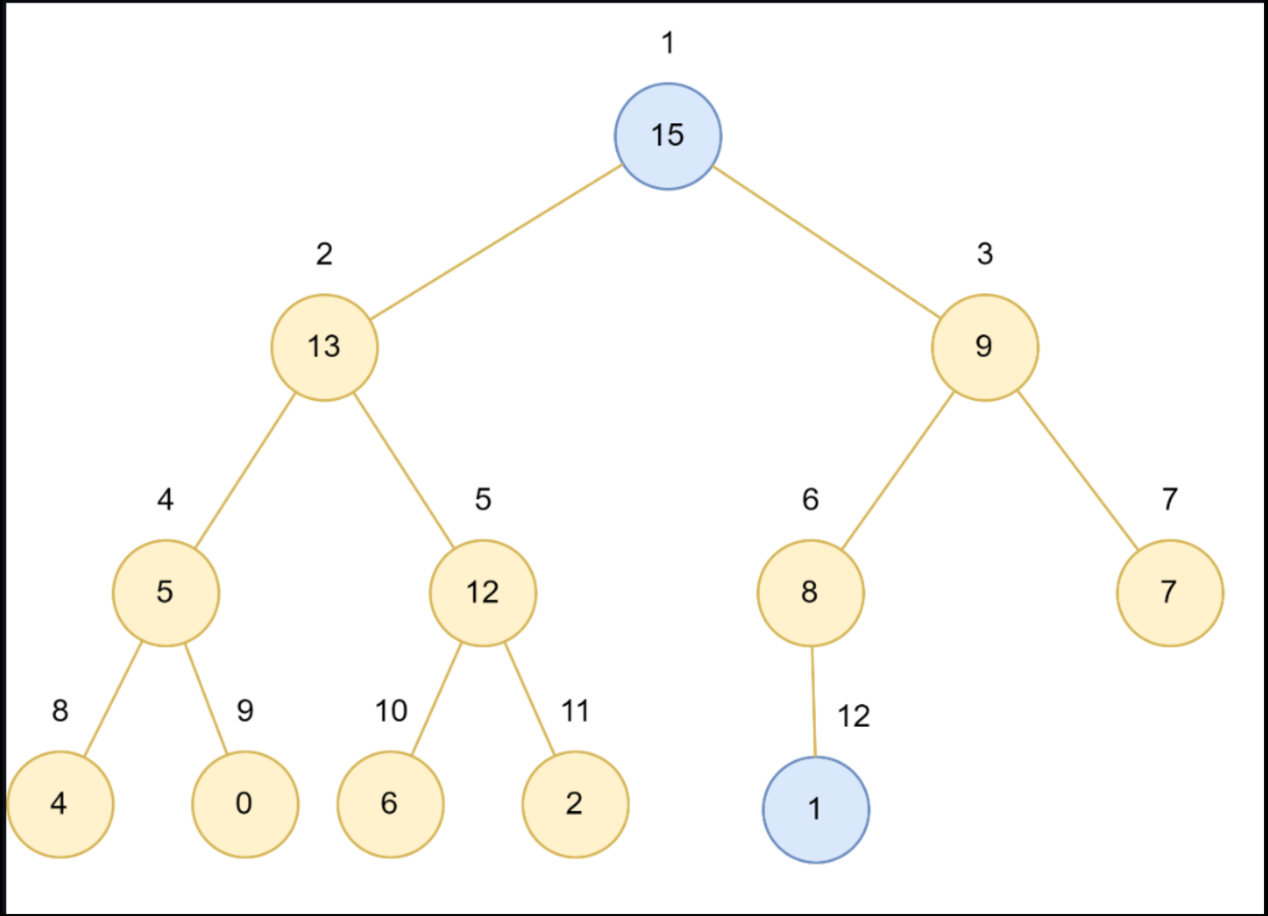
If i has a single child, MAX-HEAPIFY will compare the node with its one child, swap them if necessary.



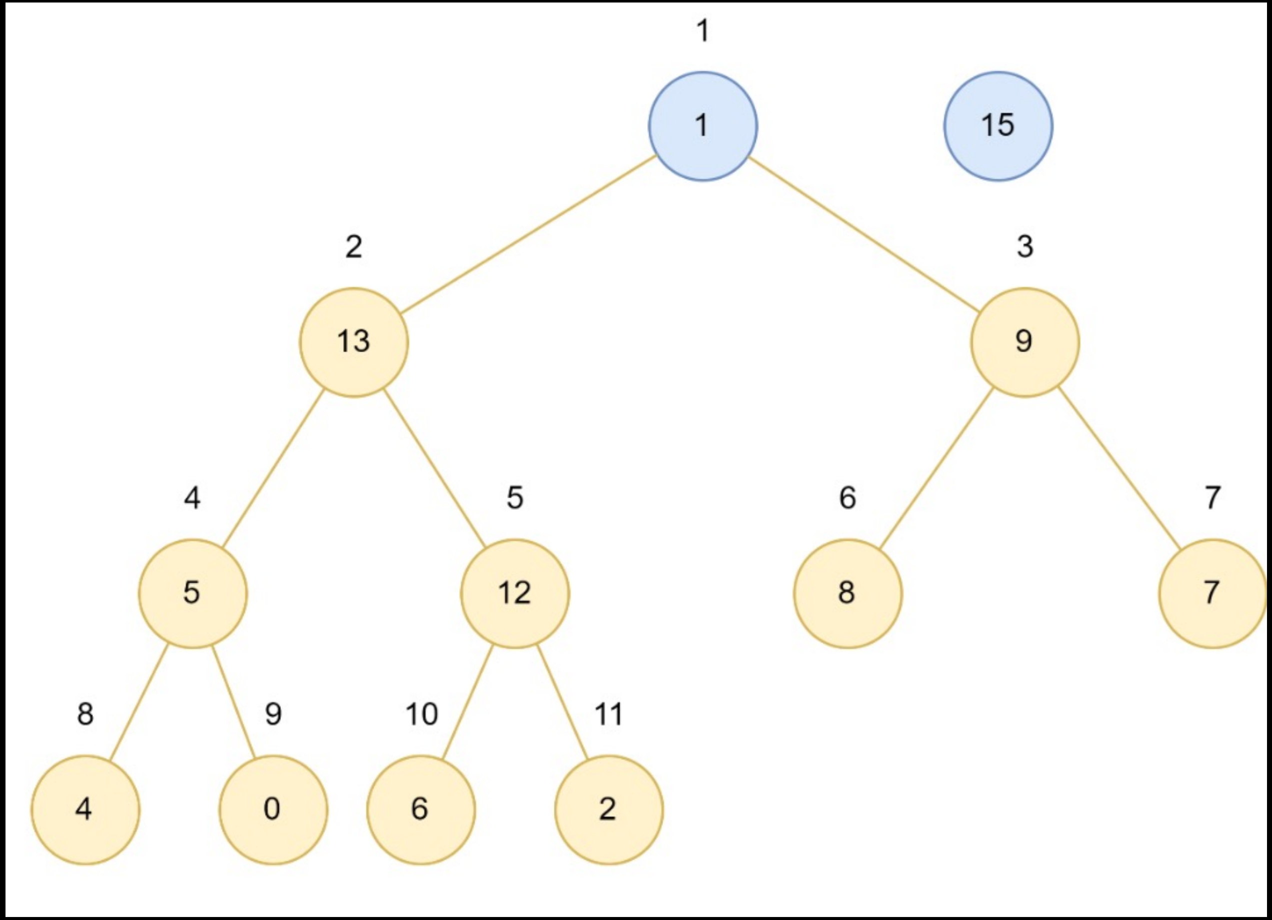
Exercises

- Suppose that the objects in a max-priority queue are just keys. Illustrate the operation of MAX-HEAP-EXTRACT-MAX on the heap A = <15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1>.

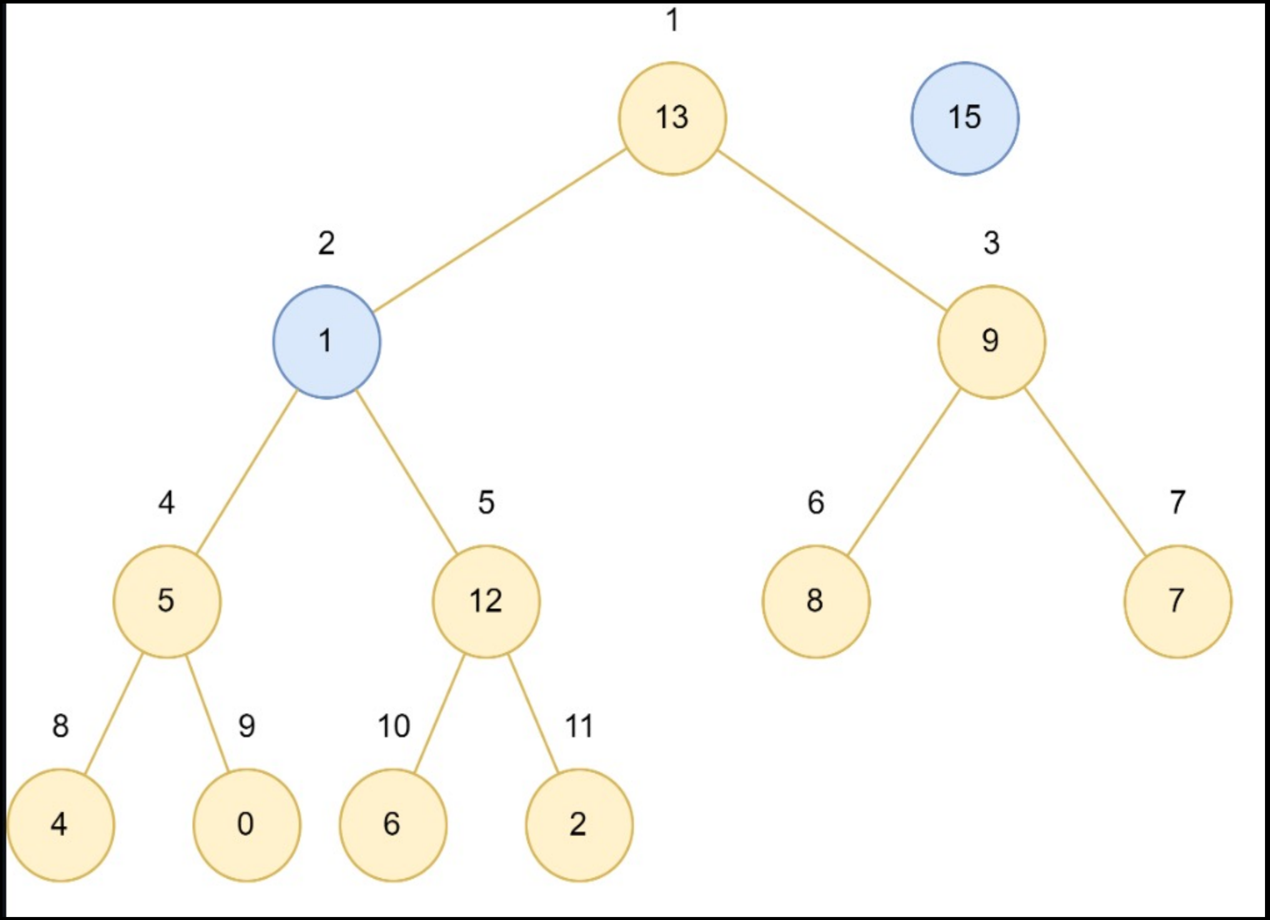
Exercises



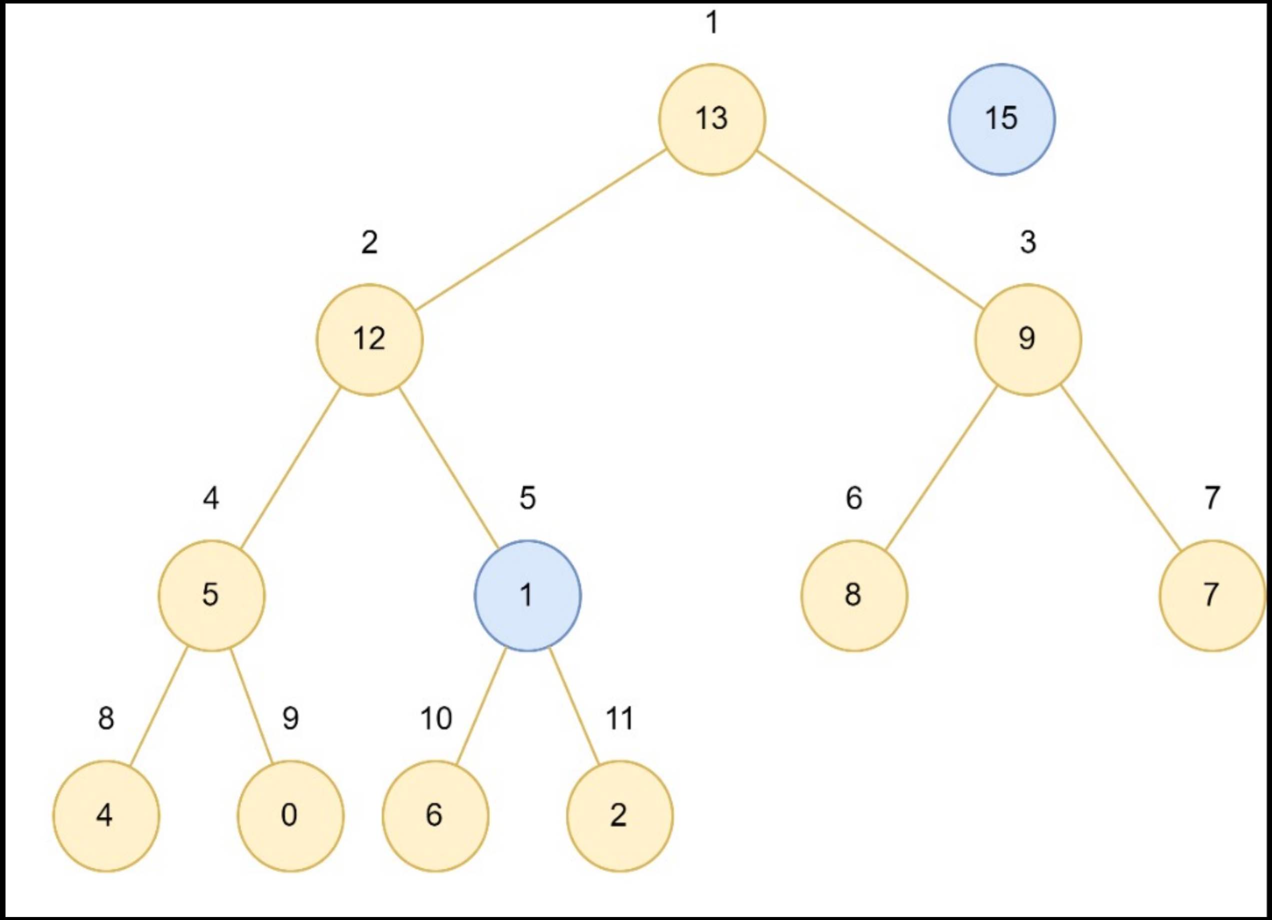
Exercises



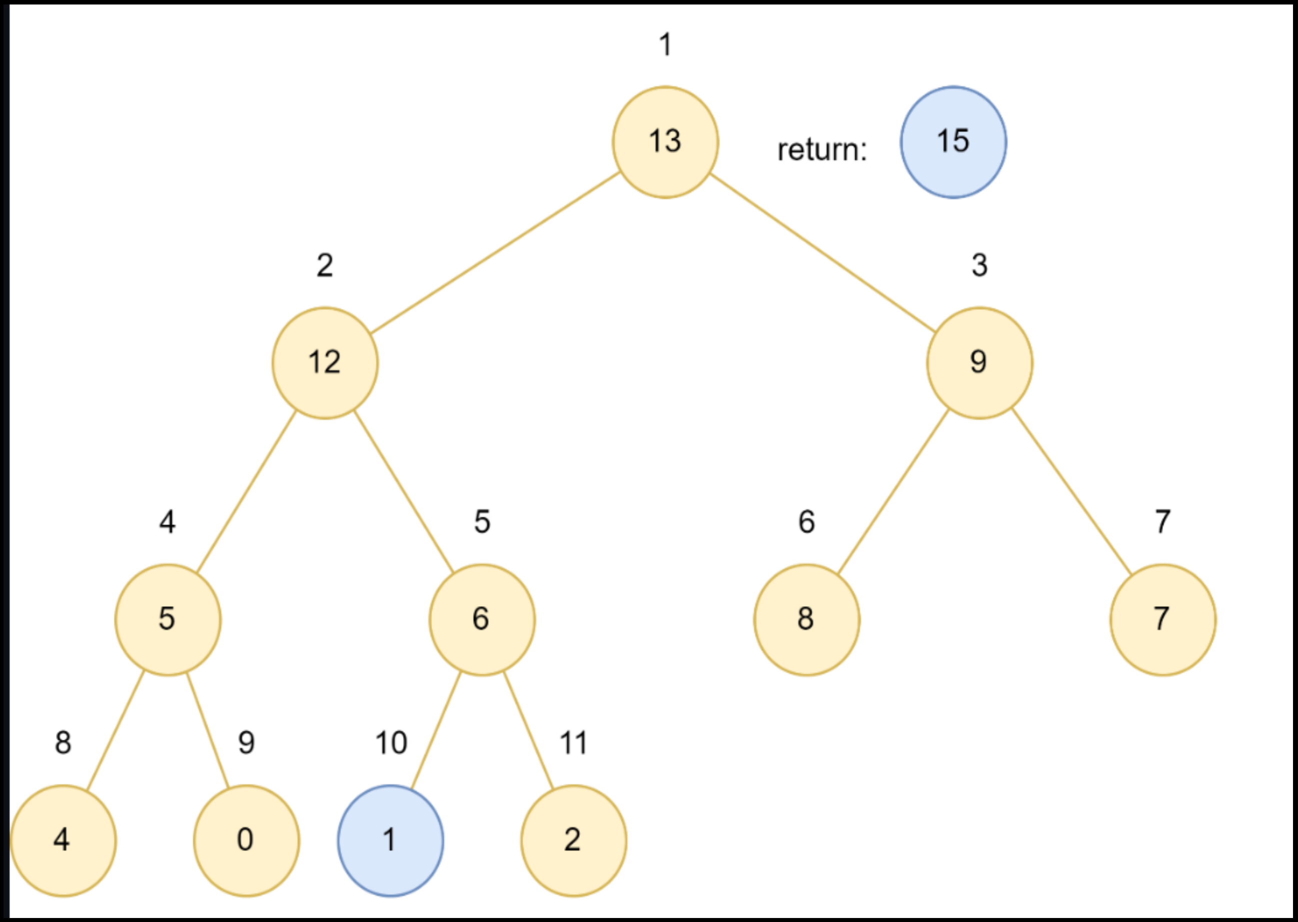
Exercises



Exercises



Exercises



Task

- Illustrate the operation of HEAPSORT on the array A=[5;13;2;25;7;17;20;8;4]
- Suppose that the objects in a max-priority queue are just keys. Illustrate the operation of MAX-HEAP-INSERT(A,10) on the heap A=<15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1>.