Introduction to Algorithms and Data Structures

Heapsort

Sorting algorithms so far

Sorting algorithms so far

- Insertsort (or Insertionsort)
  - Worst case: $\Theta(n^2)$
  - Best case: $\Theta(n)$

Sorting algorithms so far

- Insertsort (or Insertionsort)
  - Worst case: $\Theta(n^2)$
  - Best case: $\Theta(n)$
- Mergesort
  - Worst case: $\Theta(n \lg n)$
  - Best case: $\Theta(n \lg n)$

A simple sorting algorithm

A simple sorting algorithm

- Selectsort (or Selectionsort)

A simple sorting algorithm

- Selectsort (or Selectionsort)
  - “Scan” the array to find the maximum element.

A simple sorting algorithm

- Selectsort (or Selectionsort)
    - “Scan” the array to find the maximum element.
    - Put the maximum element at the end of the array.

A simple sorting algorithm

- Selectsort (or Selectionsort)
  - “Scan” the array to find the maximum element.
  - Put the maximum element at the end of the array.
  - Repeat for the part of the array that has not been sorted.

Selectsort Example

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

Selectsort Example

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

Selectsort Example

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

Selectsort Example

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

Selectsort Example

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

Selectsort Example

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

Selectsort Example

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

Selectsort Example

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

Selectsort Example

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

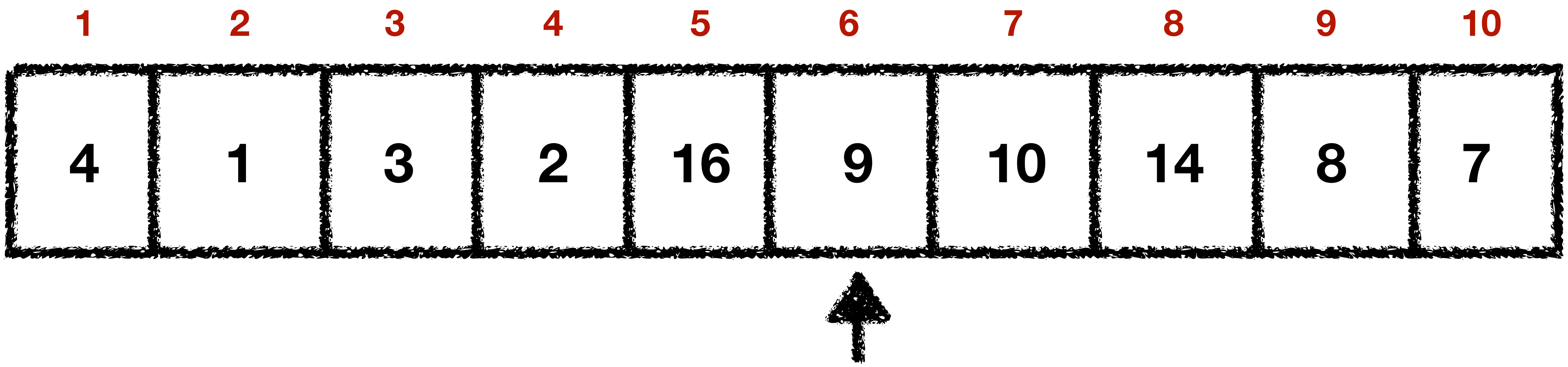
Selectsort Example

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

Selectsort Example

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

Selectsort Example



Selectsort Example

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

Selectsort Example

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

Selectsort Example

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

Selectsort Example

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

Selectsort Example

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

Selectsort Example

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

Selectsort Example

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

Selectsort Example

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

Selectsort Example

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

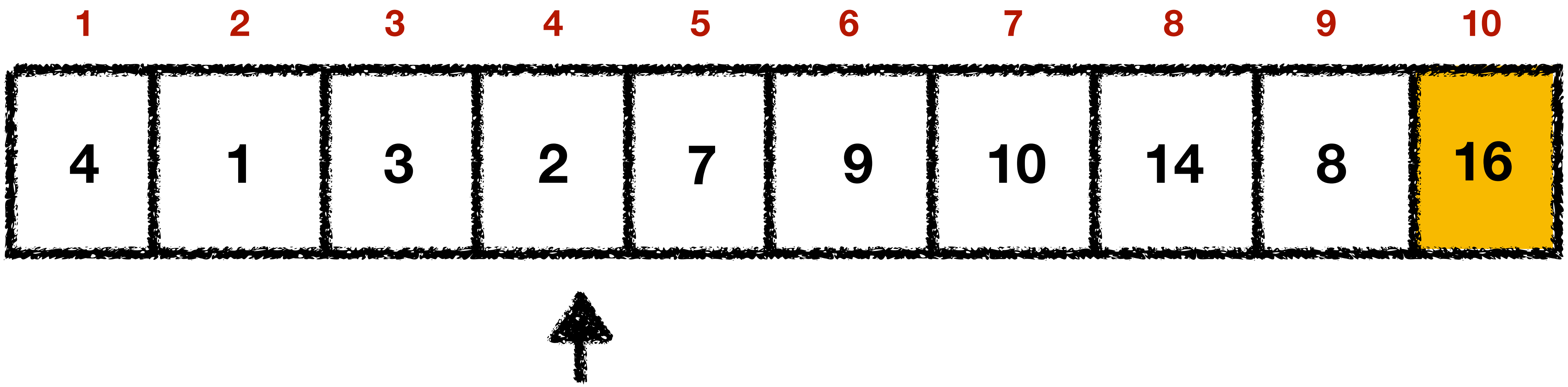
Selectsort Example

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

Selectsort Example

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

Selectsort Example



Selectsort Example

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

Selectsort Example

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

Selectsort Example

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

Selectsort Example

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

Selectsort Example

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

Selectsort Example

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

Selectsort Example

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

Selectsort Example

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

Selectsort Example

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

Selectsort Example

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

- What is the worst-case running time of Selectsort?

Selectsort Example

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

- What is the worst-case running time of Selectsort?
- What is the best case-running time of Selectsort?

Selectsort Example

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

- What is the worst-case running time of Selectsort?
- What is the best case-running time of Selectsort?
- Let's try it on wooclap!

Selectsort Running time

- Worst-case: $\Theta(n^2)$
- Average-case: $\Theta(n^2)$
- Best-case: $\Theta(n^2)$
- Why is this happening?

Selectsort Running time

- Worst-case: $\Theta(n^2)$
- Average-case: $\Theta(n^2)$
- Best-case: $\Theta(n^2)$
- Why is this happening?
- For $\Omega(n)$ iterations, we need $\Omega(n)$ comparisons to find the maximum element.

What if we could…

What if we could…

- …find the maximum element in $O(1)$ time?

What if we could…

- …find the maximum element in $O(1)$ time?
- That would give us a sorting algorithm with $O(n)$ running time…

What if we could…

- …find the maximum element in $O(1)$ time?
- That would give us a sorting algorithm with $O(n)$ running time…
- … which is in general not possible!

What if we could…

What if we could…

- …find the maximum element in $O(1)$ time…

What if we could…

- …find the maximum element in $O(1)$ time…
- … after we preprocess the array a little bit…

What if we could…

- …find the maximum element in $O(1)$ time…
- … after we preprocess the array a little bit…
- … and after we process it again after each iteration …

What if we could…

- …find the maximum element in $O(1)$ time…
- … after we preprocess the array a little bit…
- … and after we process it again after each iteration …
- … without using too many comparisons/operations?

Heapsort (very informally)

Heapsort (very informally)

1. Preprocess the array to become a *special array*.

Heapsort (very informally)

1. Preprocess the array to become a *special array*.

2. Find the maximum element of the special array in $O(1)$ time and move it to the last position.

Heapsort (very informally)

1. Preprocess the array to become a *special array*.

2. Find the maximum element of the special array in $O(1)$ time and move it to the last position.

3. Consider the remaining array and process it to become a special array again.

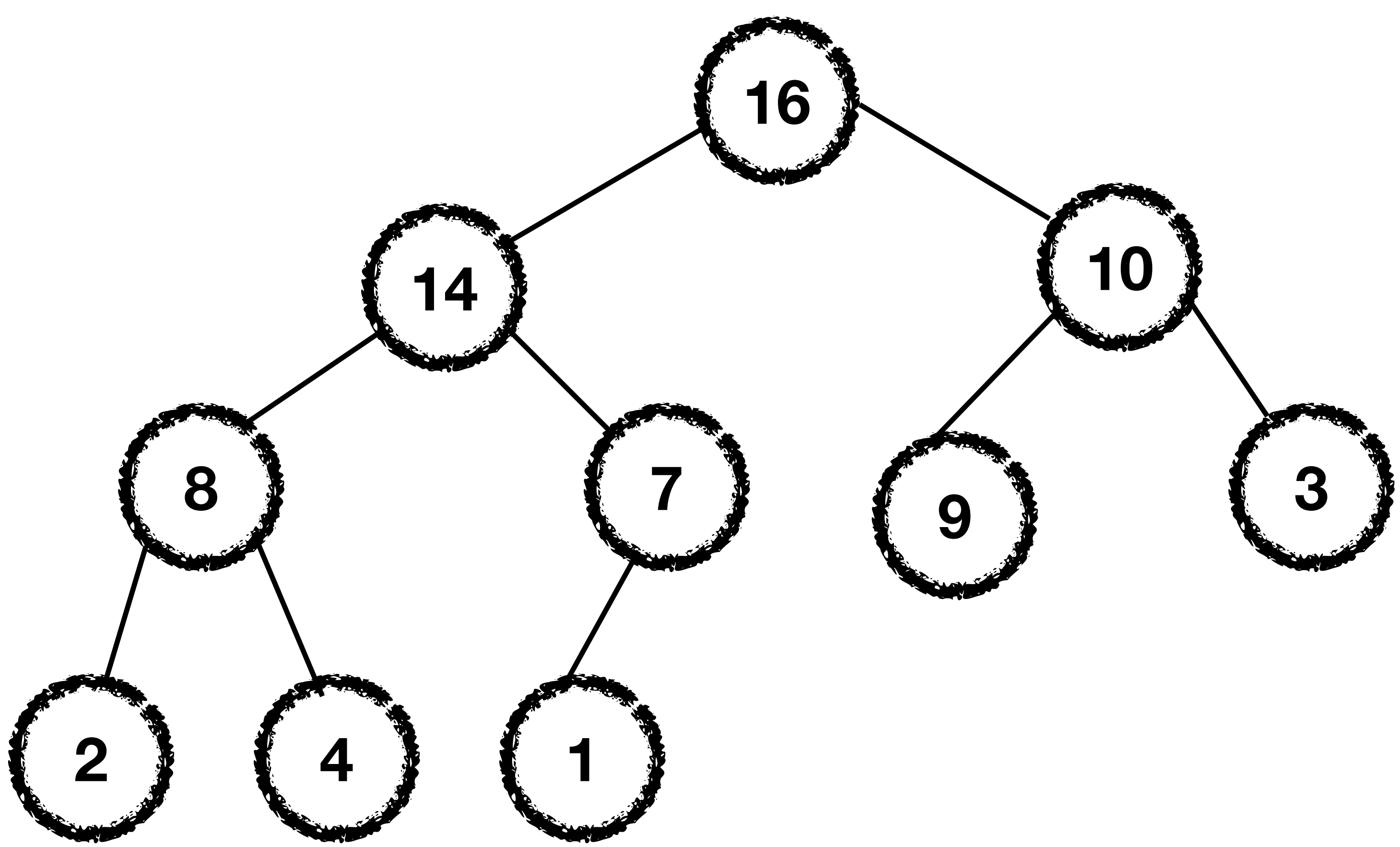
Heapsort (very informally)

1. Preprocess the array to become a *special array*.
2. Find the maximum element of the special array in $O(1)$ time and move it to the last position.
3. Consider the remaining array and process it to become a special array again.
4. Repeat Steps 2-4 for the remaining special array, until the remaining array has size 0.

The Heap Data Structure

An almost complete binary tree.

All levels completely filled, except possibly the last one, which is filled from the left to the right.



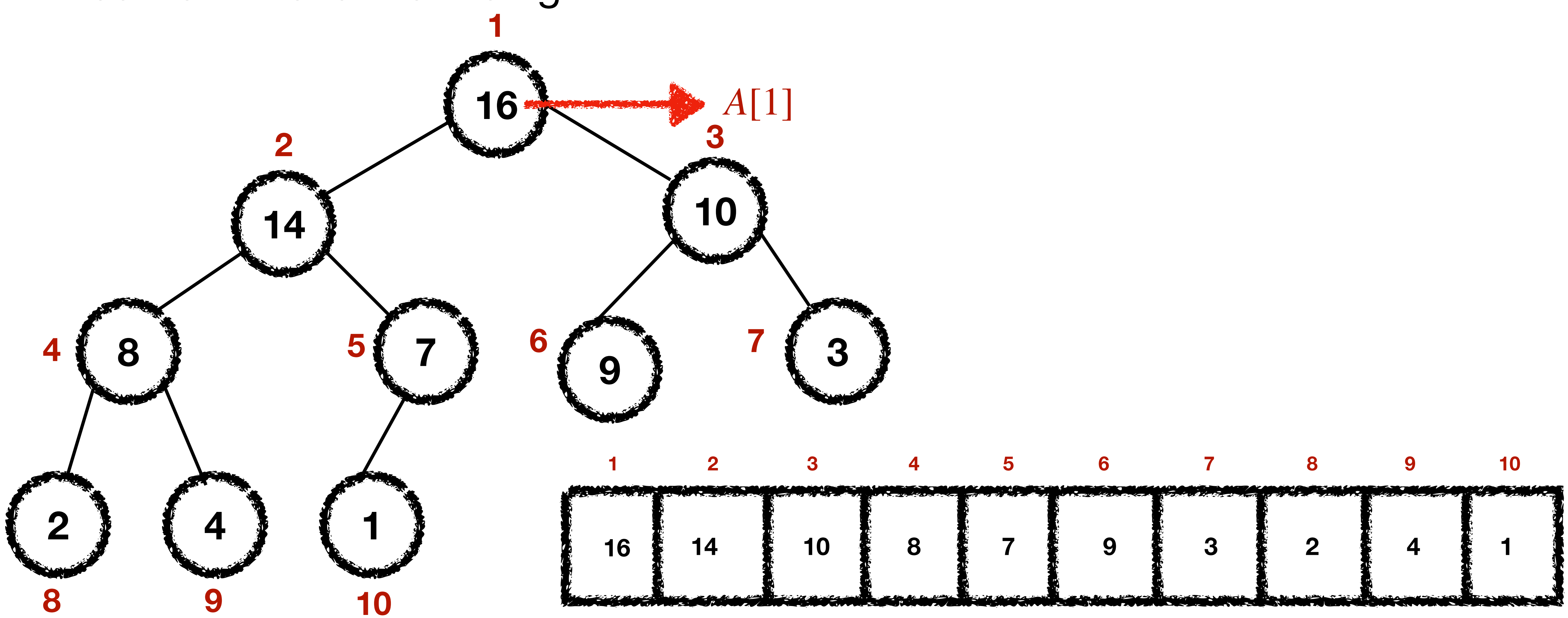
The Heap Data Structure

An almost complete binary tree.

All levels completely filled, except possibly the last one, which is filled from the left to the right.

Implemented via an array $A[1:n]$.

Attribute A.heap-size, so we can access $A[1:A\text{.heap-size}]$



The Heap Data Structure

An almost complete binary tree.

All levels completely filled, except possibly the last one, which is filled from the left to the right.

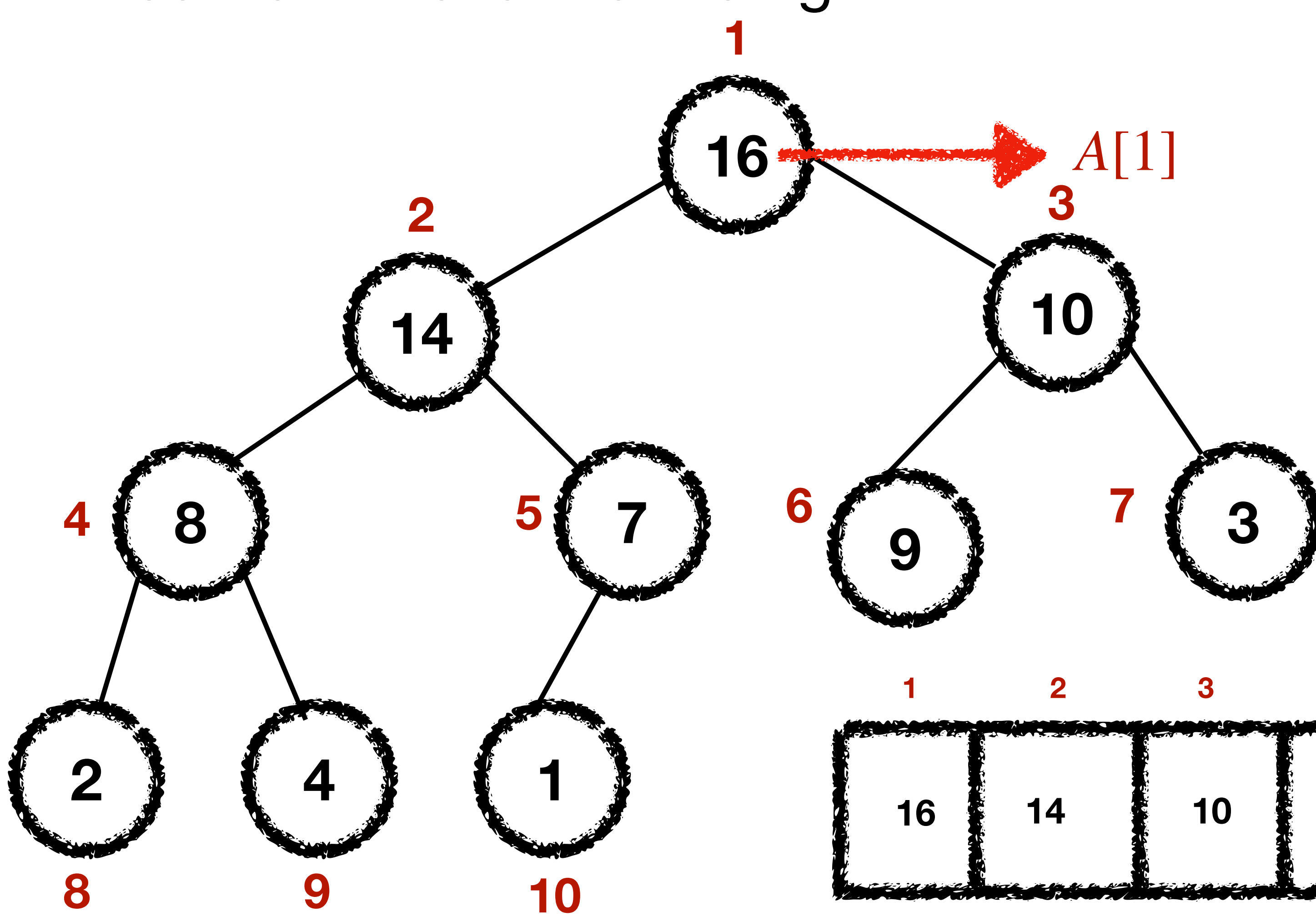
Implemented via an array $A[1 : n]$.

Attribute A.heap-size, so we can access $A[1 : A.\text{heap-size}]$

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

Left(i)
  return $2i$

Right(i)
  return $2i + 1$

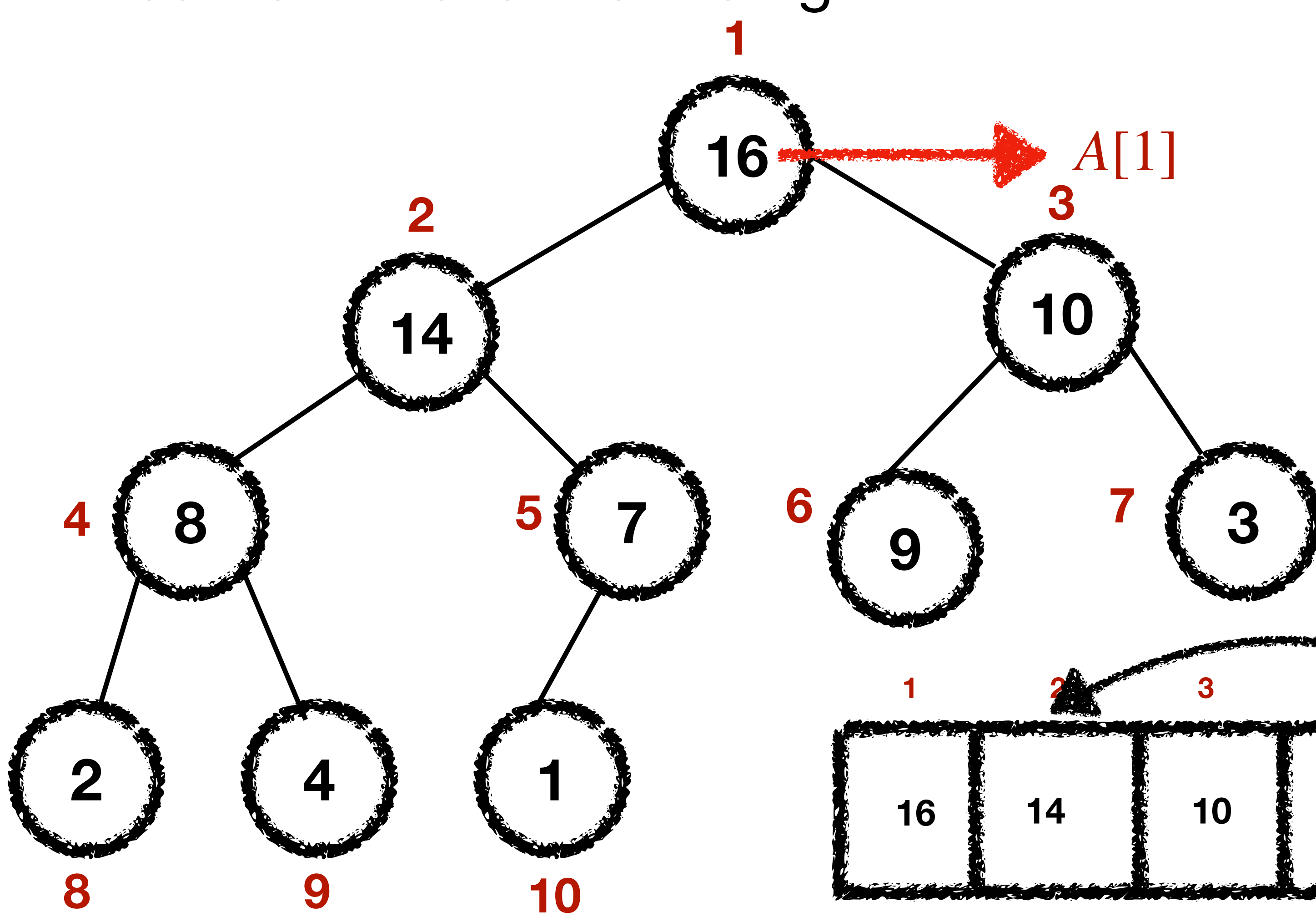


1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

The Heap Data Structure

An almost complete binary tree.

All levels completely filled, except possibly the last one, which is filled from the left to the right.



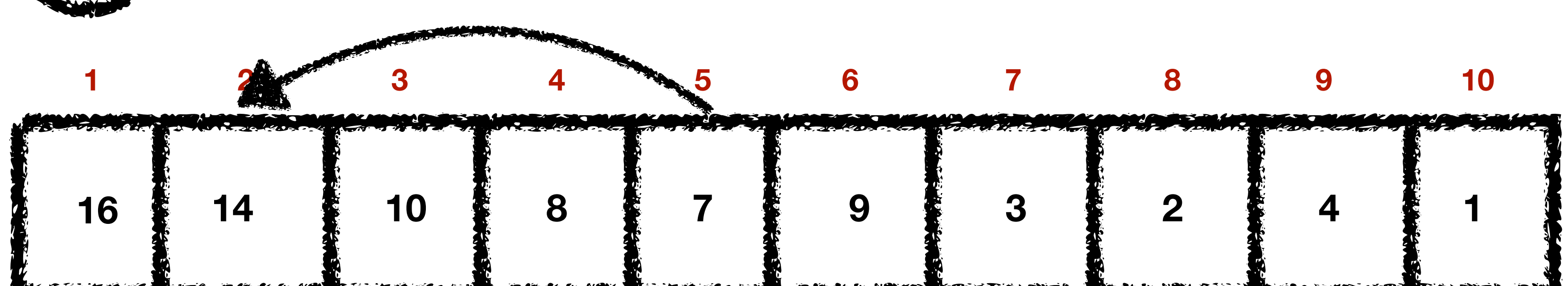
Implemented via an array $A[1:n]$.

Attribute A.heap-size, so we can access $A[1:A.\text{heap-size}]$

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

Left(i)
return $2i$

Right(i)
return $2i+1$



The Heap Data Structure

An almost complete binary tree.

All levels completely filled, except possibly the last one, which is filled from the left to the right.

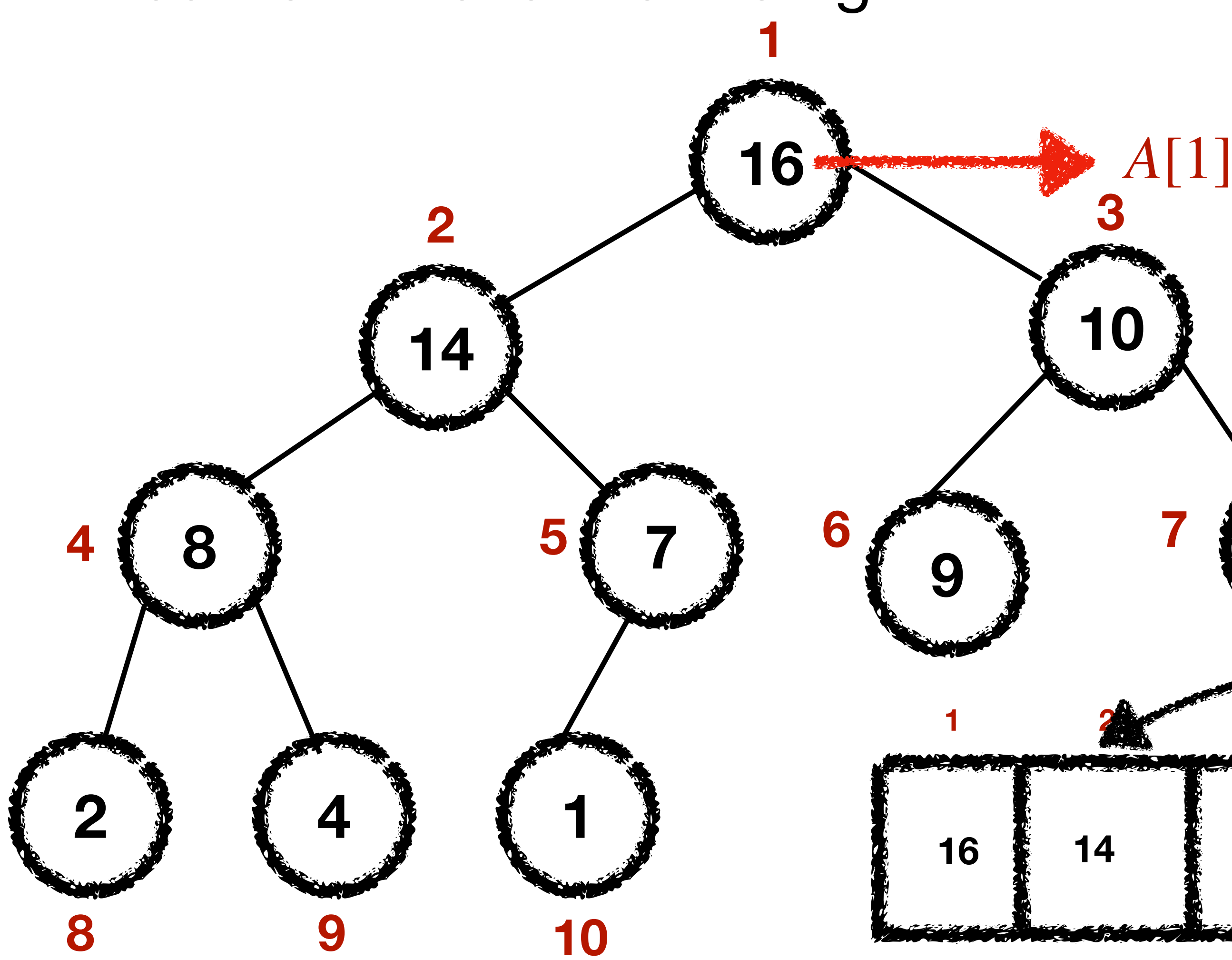
Implemented via an array $A[1:n]$.

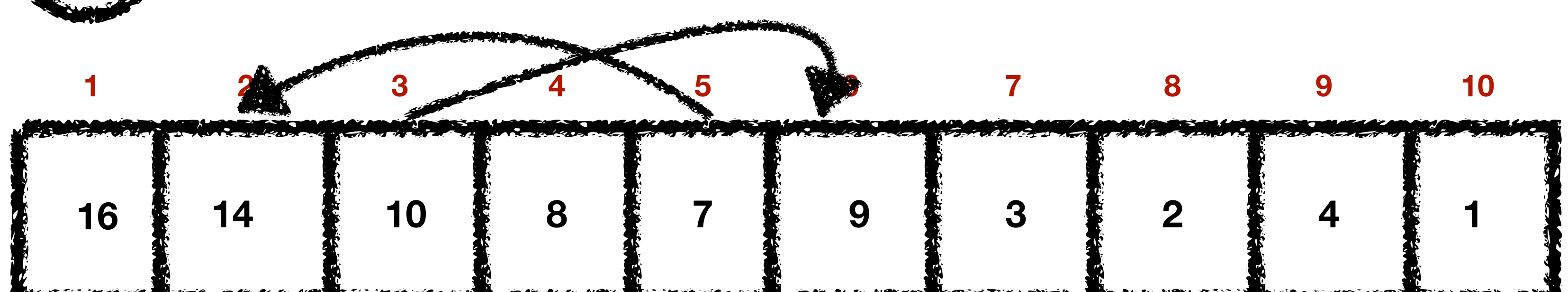
Attribute A.heap-size, so we can access $A[1:A.\text{heap-size}]$

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

Left(i)
 return $2i$

Right(i)
 return $2i+1$





The Heap Data Structure

An almost complete binary tree.

All levels completely filled, except possibly the last one, which is filled from the left to the right.

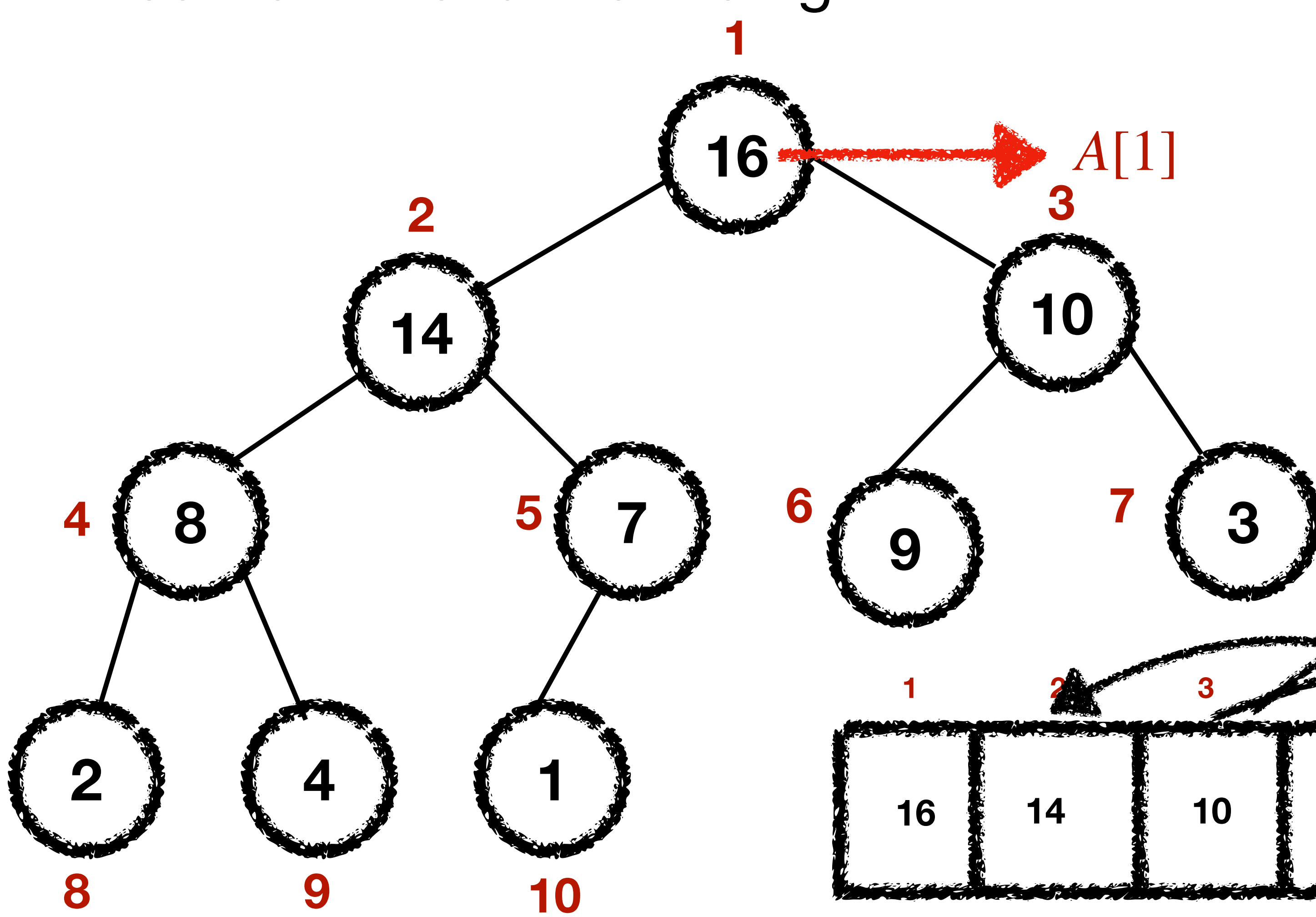
Implemented via an array $A[1:n]$.

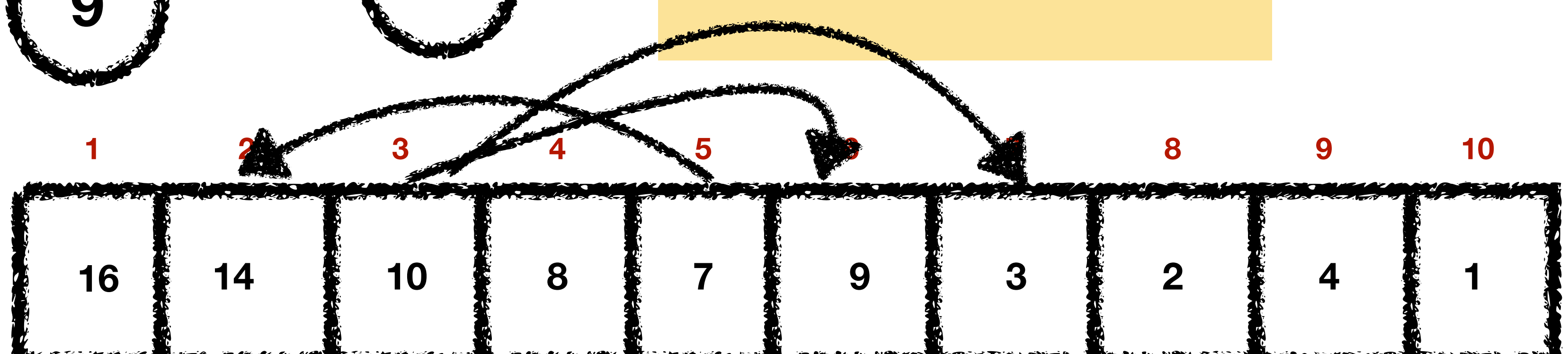
Attribute A.heap-size, so we can access $A[1:A.\text{heap-size}]$

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

Left(i)
    return $2i$

Right(i)
    return $2i + 1$

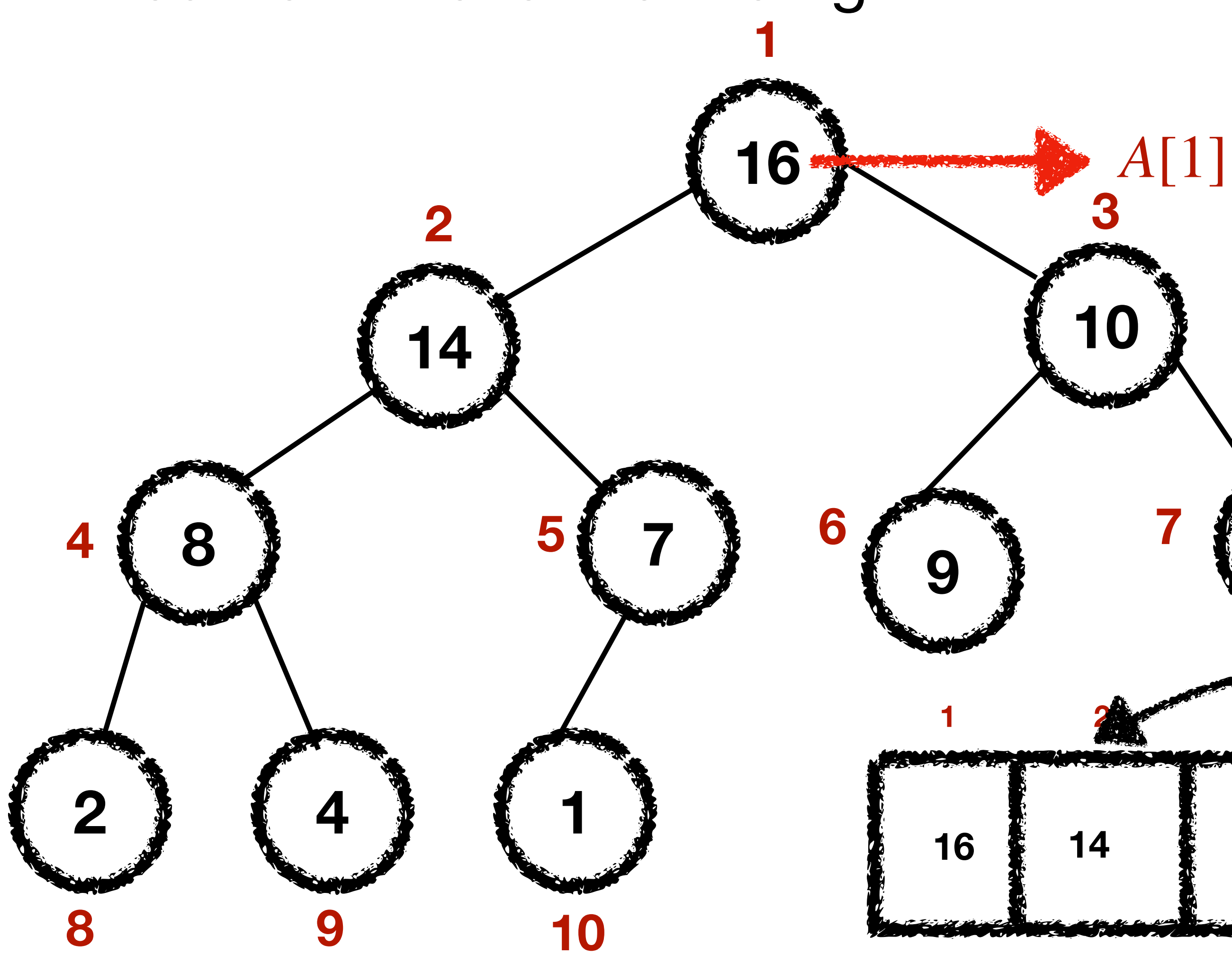




The Heap Data Structure

An almost complete binary tree.

All levels completely filled, except possibly the last one, which is filled from the left to the right.



Implemented via an array $A[1:n]$.

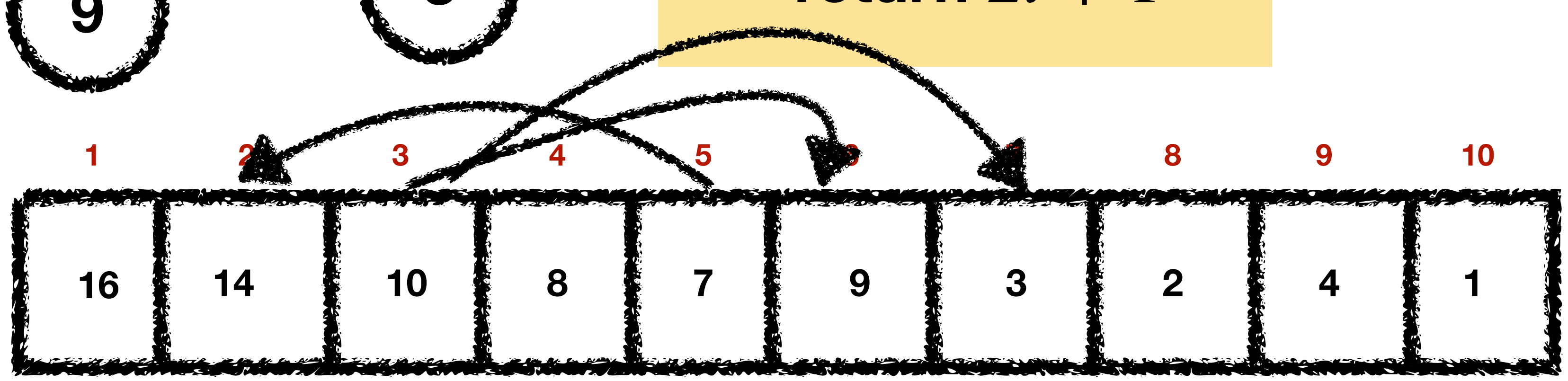
Attribute A.heap-size, so we can access $A[1 : A.\text{heap-size}]$

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

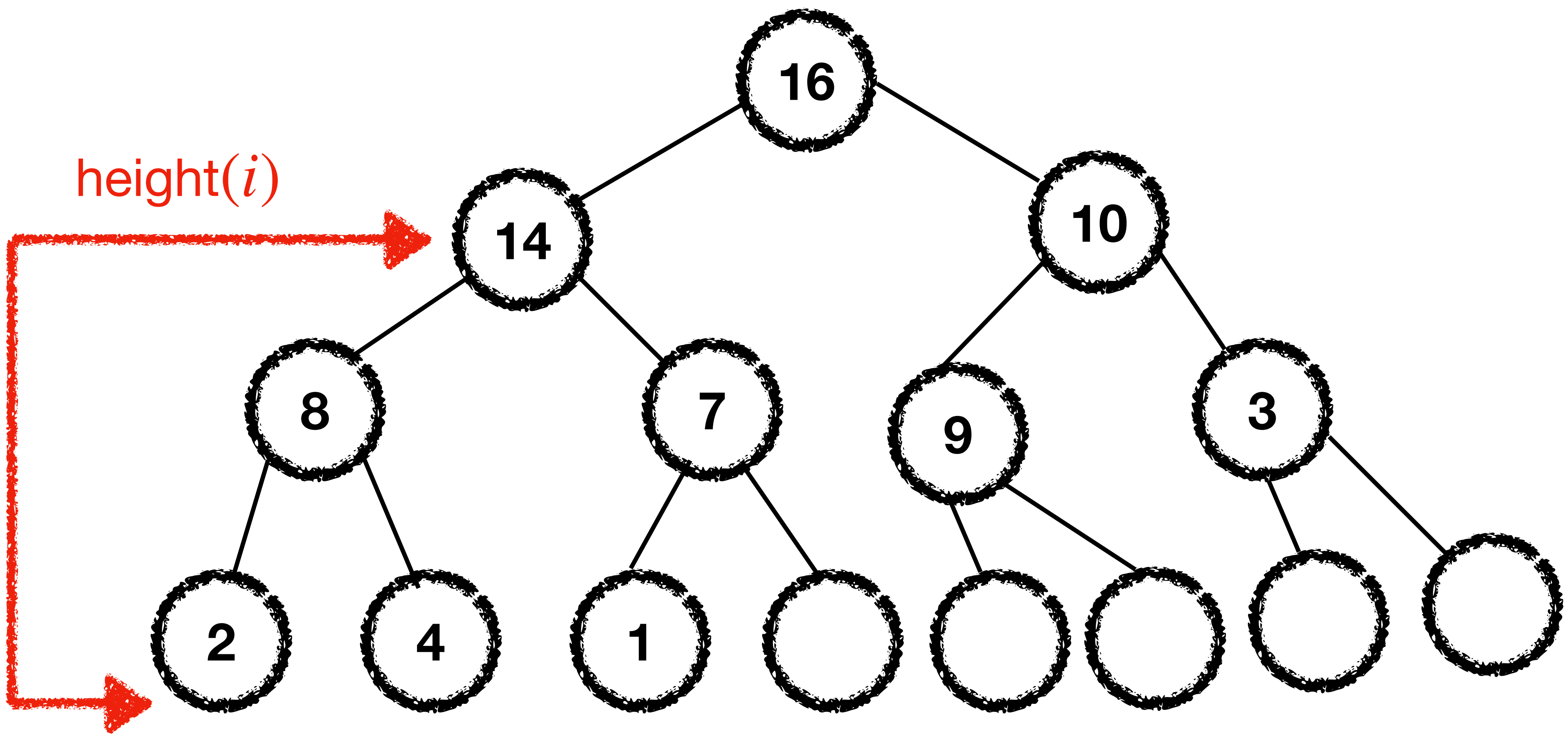
Left(i)
return $2i$

Right(i)
return $2i + 1$

These can be done in $O(1)$ time.



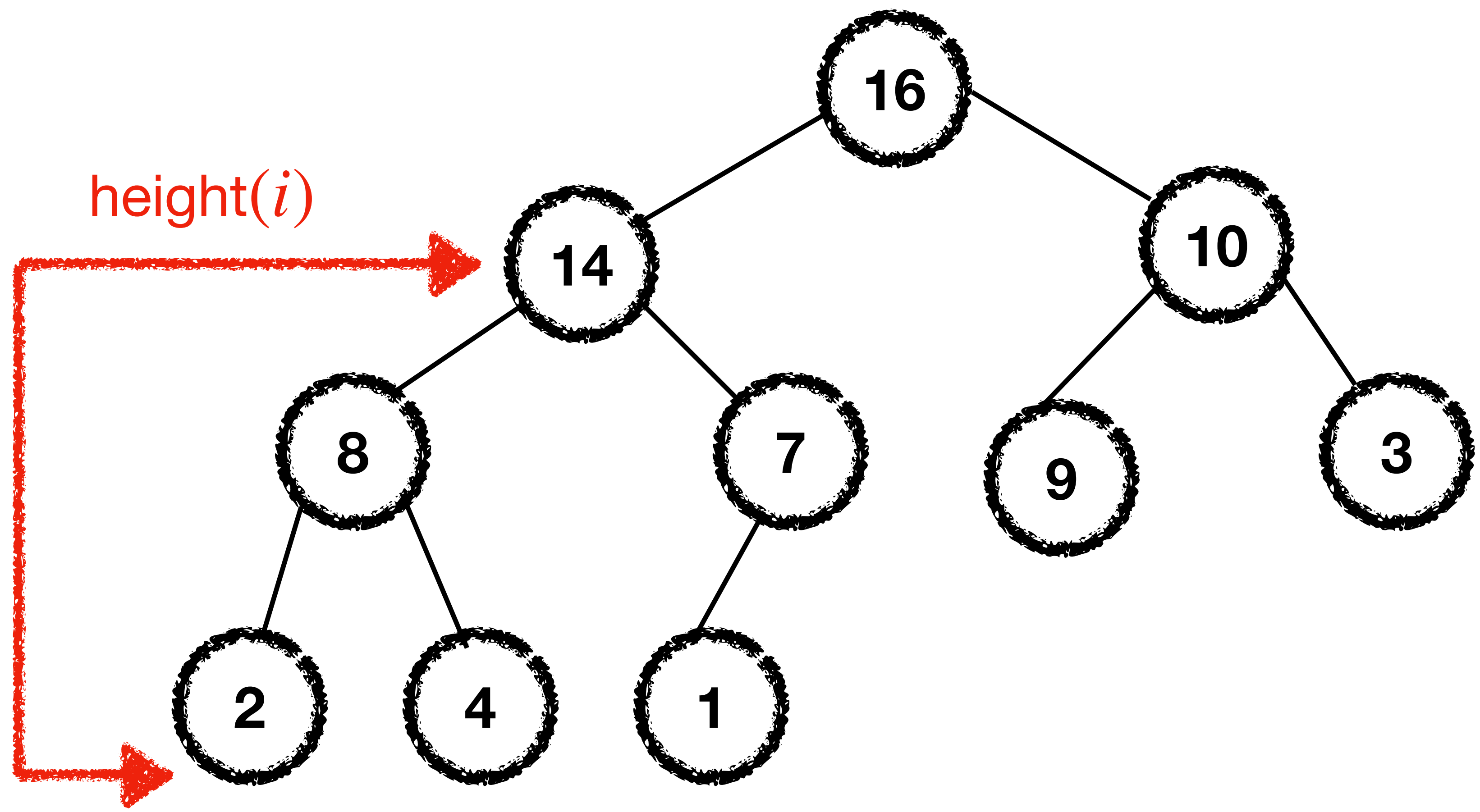
Height of a binary tree



height(i) = # edges on the longest simple path to a leaf

height(T) = height(root)

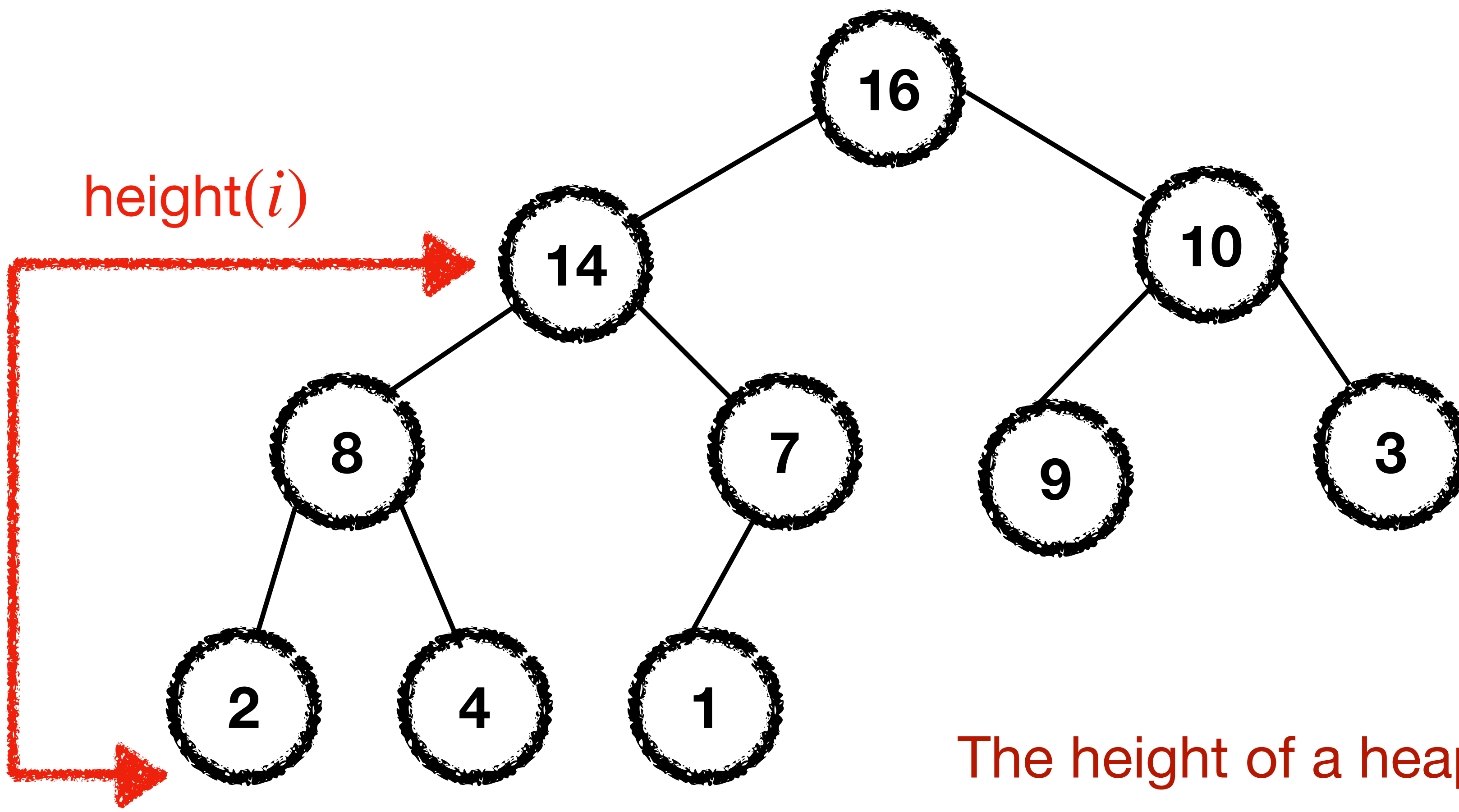
Height of a heap



height(i) = # edges on the longest simple path to a leaf

height($Tree$) = height(root)

Height of a heap



The height of a heap on n nodes is $\Theta(\lg n)$
(in particular, $\lfloor \lg n \rfloor$)

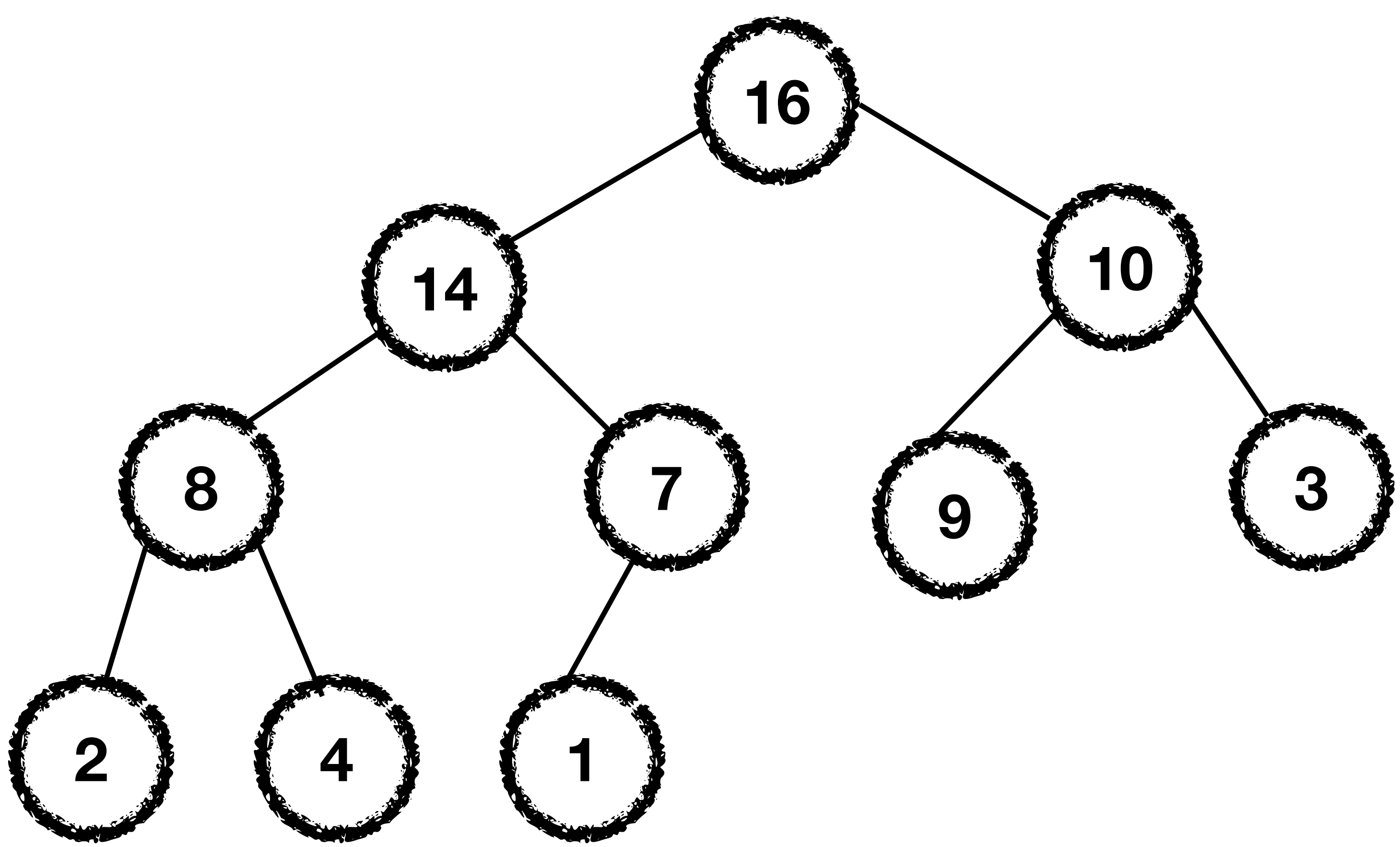
height(i) = # edges on the longest simple path to a leaf

height($Tree$) = height(root)

(Max) Heap Property

The value of a node is at most the value of its parent, i.e.,

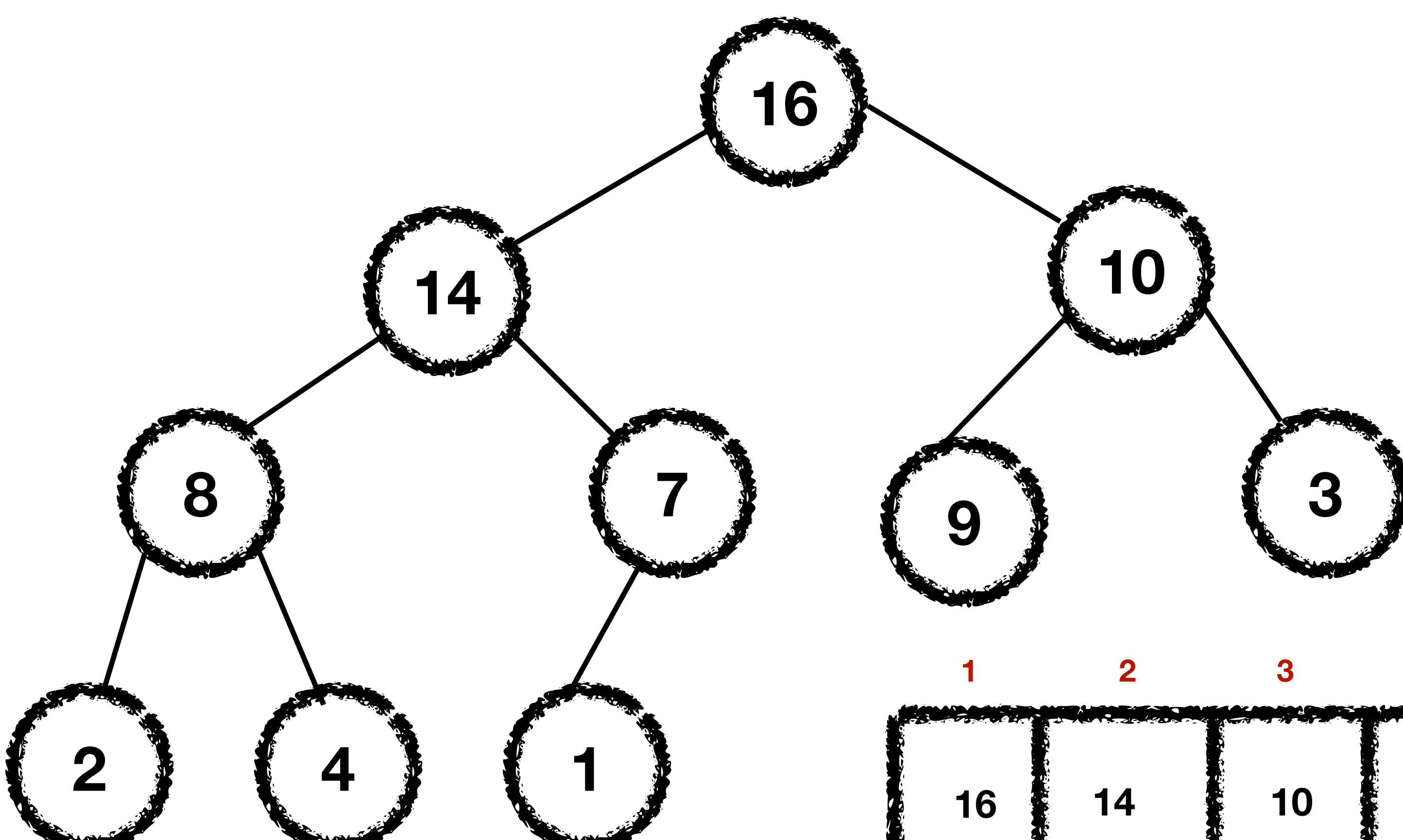
$$A[\text{Parent}(i)] \geq A[i].$$



(Max) Heap Property

The value of a node is at most the value of its parent, i.e.,

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

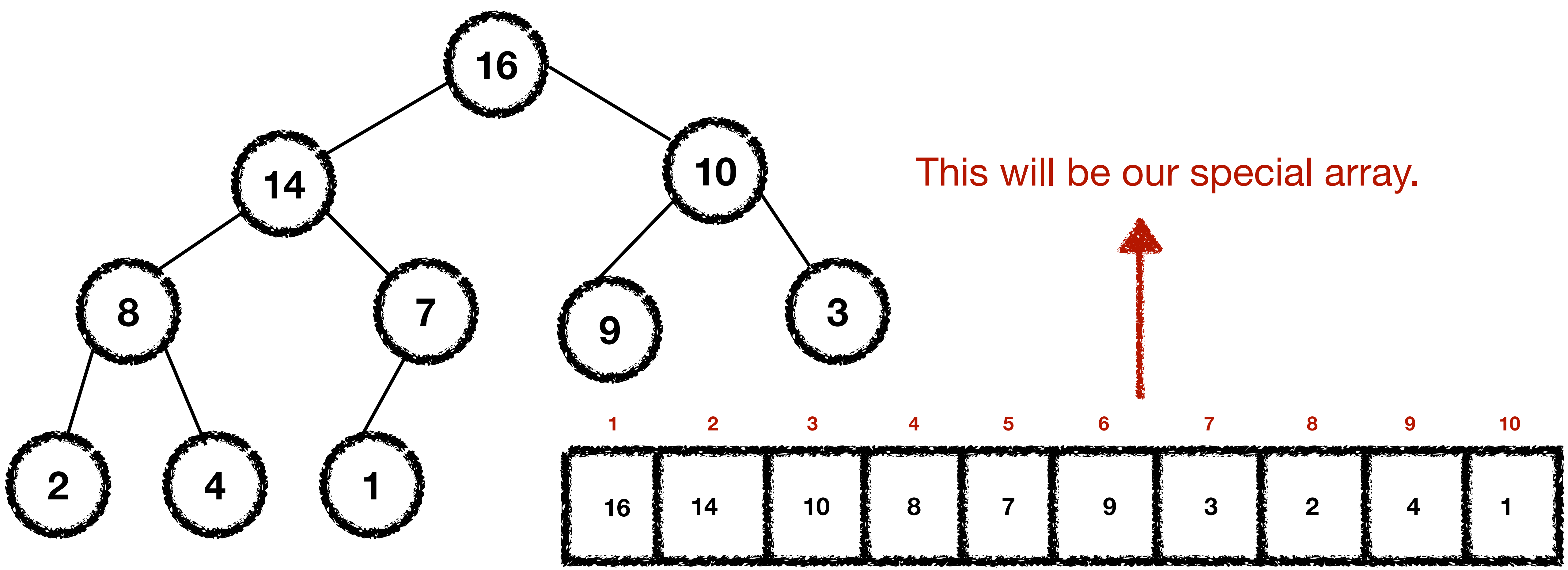


1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

(Max) Heap Property

The value of a node is at most the value of its parent, i.e.,

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



Heapsort (a bit less informally)

Heapsort (a bit less informally)

1. Preprocess the array to become a *heap*.

Heapsort (a bit less informally)

1. Preprocess the array to become a *heap*.

2. Find the maximum element of the heap in $O(1)$ time and move it to the last position (*how?*)

Heapsort (a bit less informally)

1. Preprocess the array to become a *heap*.

2. Find the maximum element of the heap in $O(1)$ time and move it to the last position (*how?*)

3. Consider the remaining array and process it to become a heap again.

Heapsort (a bit less informally)

1. Preprocess the array to become a *heap*.

2. Find the maximum element of the heap in $O(1)$ time and move it to the last position (*how?*)

3. Consider the remaining array and process it to become a heap again.

4. Repeat Steps 2-4 for the remaining special array, until the remaining array has size 0.

Max-Heapify

Max-Heapify

3. Consider the remaining array and process it to become a heap again.

Max-Heapify

3. Consider the remaining array and process it to become a heap again.

Max-Heapify(A, i).

Max-Heapify

3. Consider the remaining array and process it to become a heap again.

Max-Heapify(A, i).

Precondition: Trees rooted at Left(i) and Right(i) are heaps.

Max-Heapify

3. Consider the remaining array and process it to become a heap again.

Max-Heapify(A, i).

Precondition: Trees rooted at Left(i) and Right(i) are heaps.

Postcondition: The tree routed at i is a heap.

Max-Heapify

3. Consider the remaining array and process it to become a heap again.

Max-Heapify(A, i).

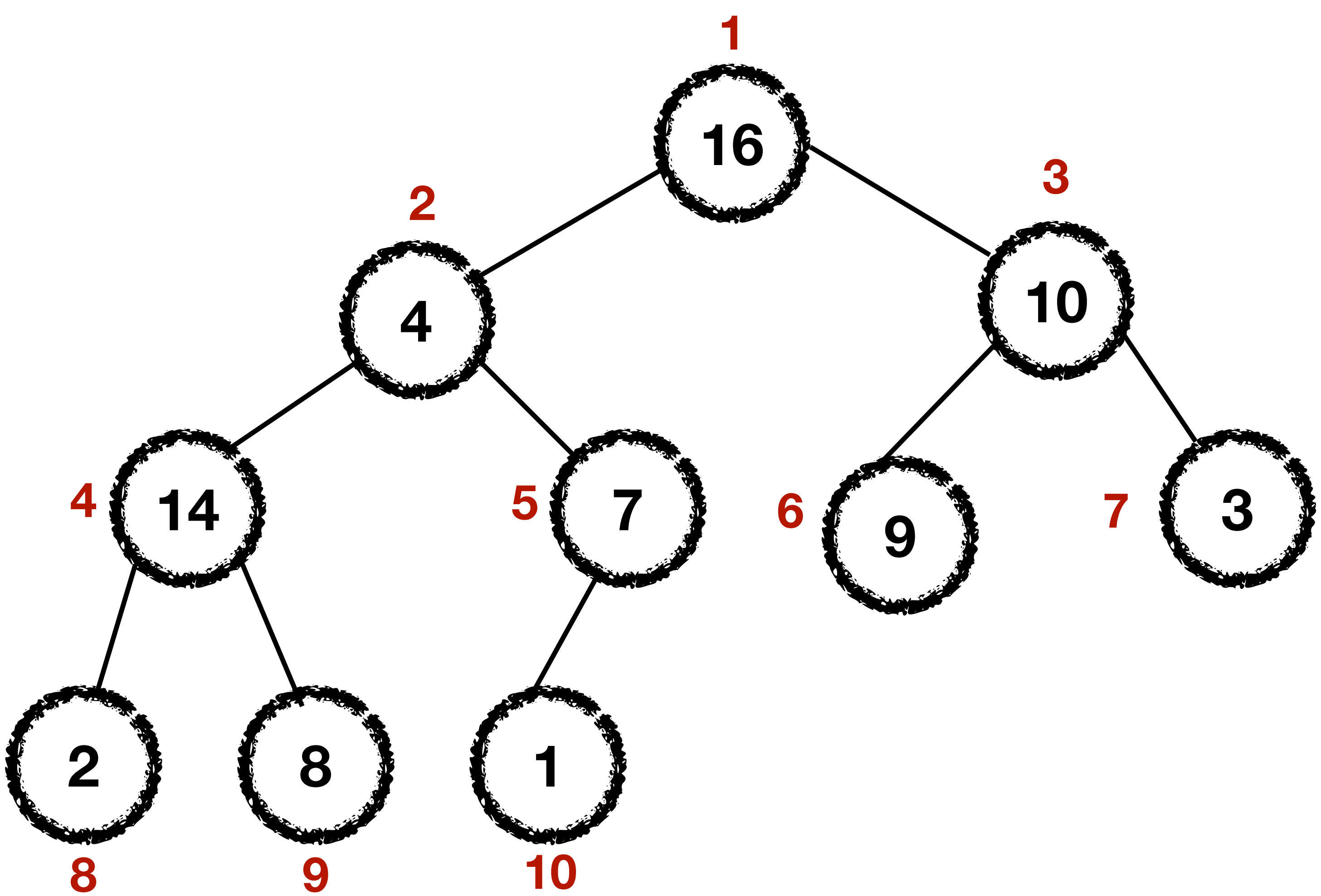
Precondition: Trees rooted at Left(i) and Right(i) are heaps.

Postcondition: The tree routed at i is a heap.

```
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)
```

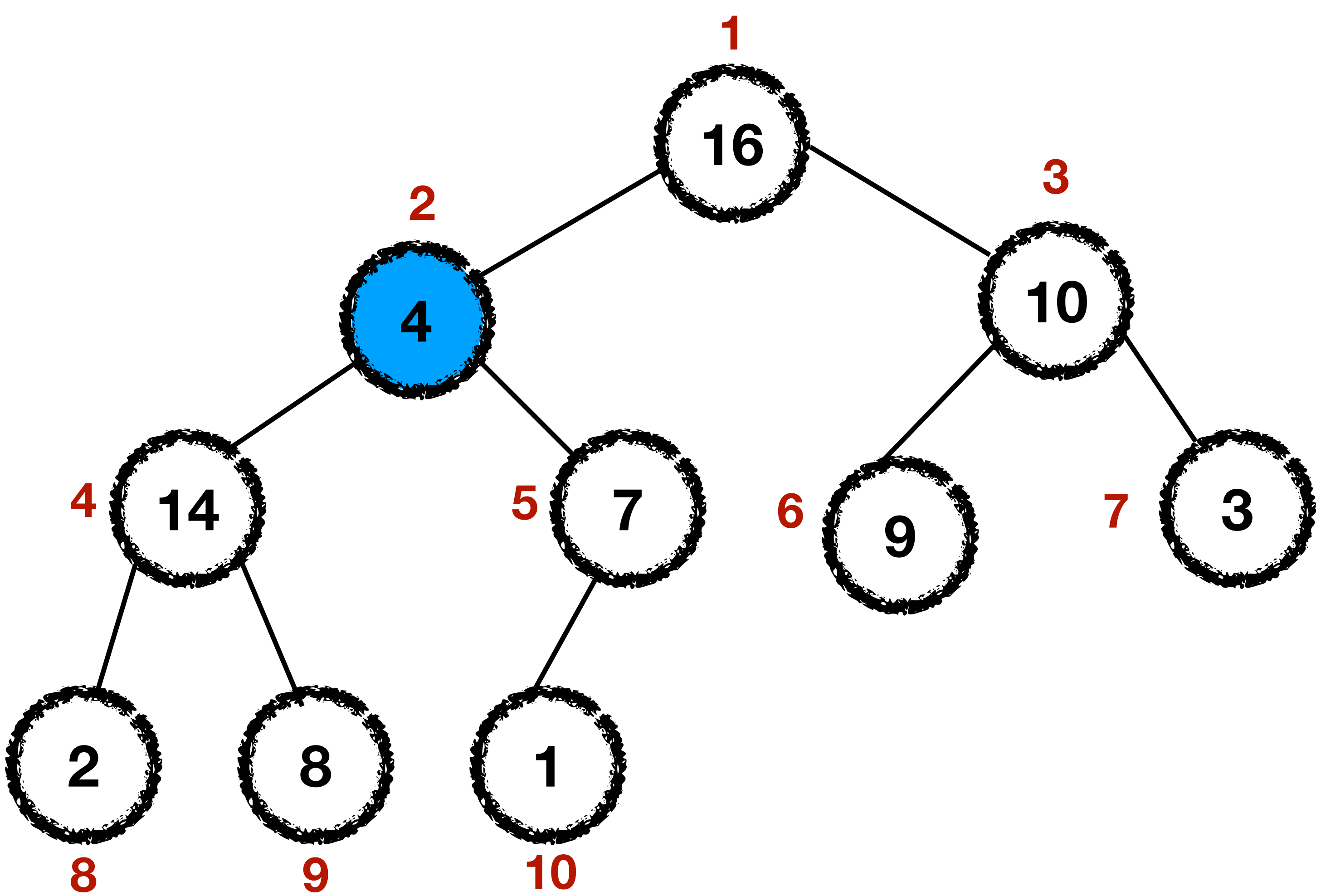
CLRS pp 165

Max-Heapify




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

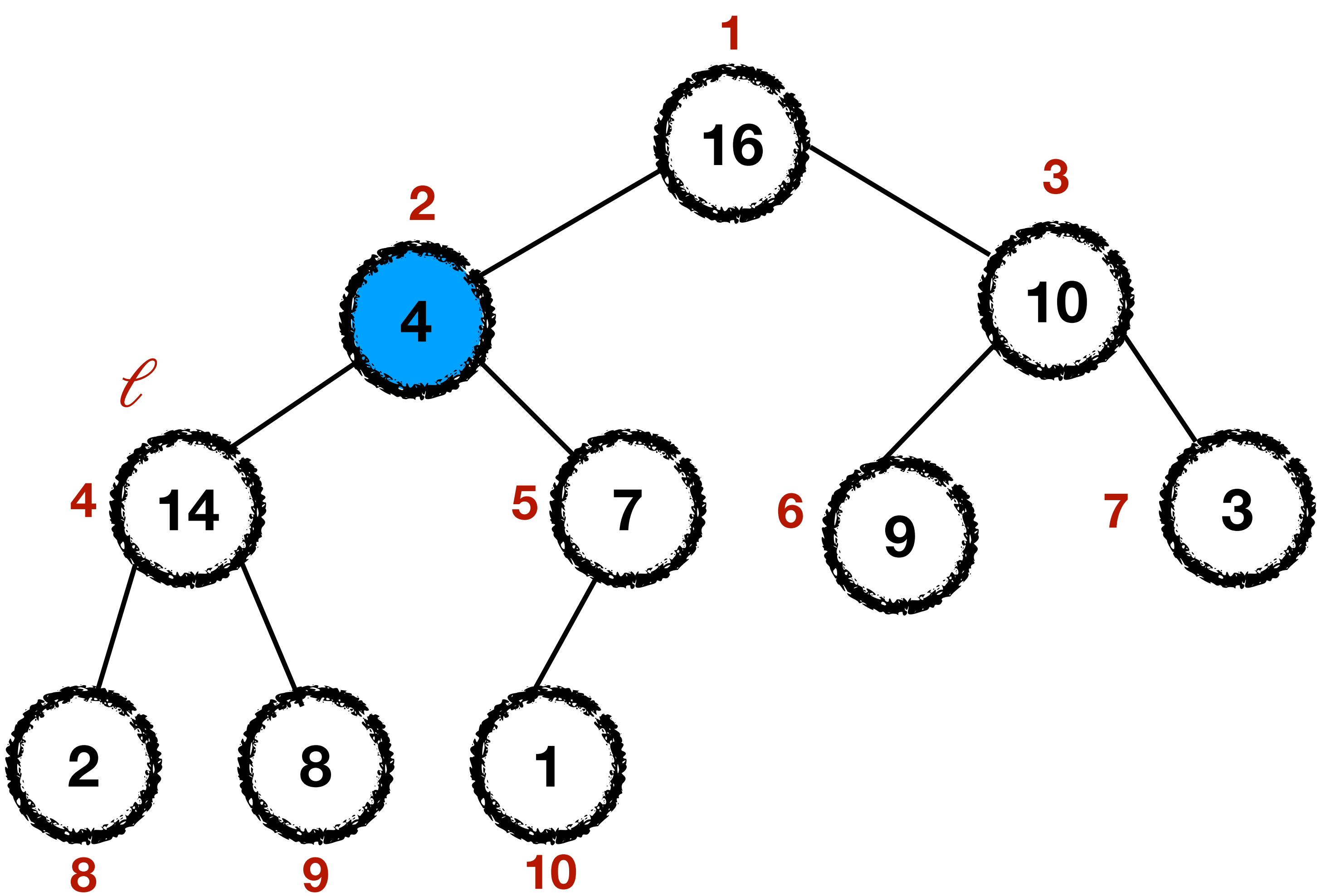
Max-Heapify



$i = 2, \quad A[i] = 4$

```
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)
```

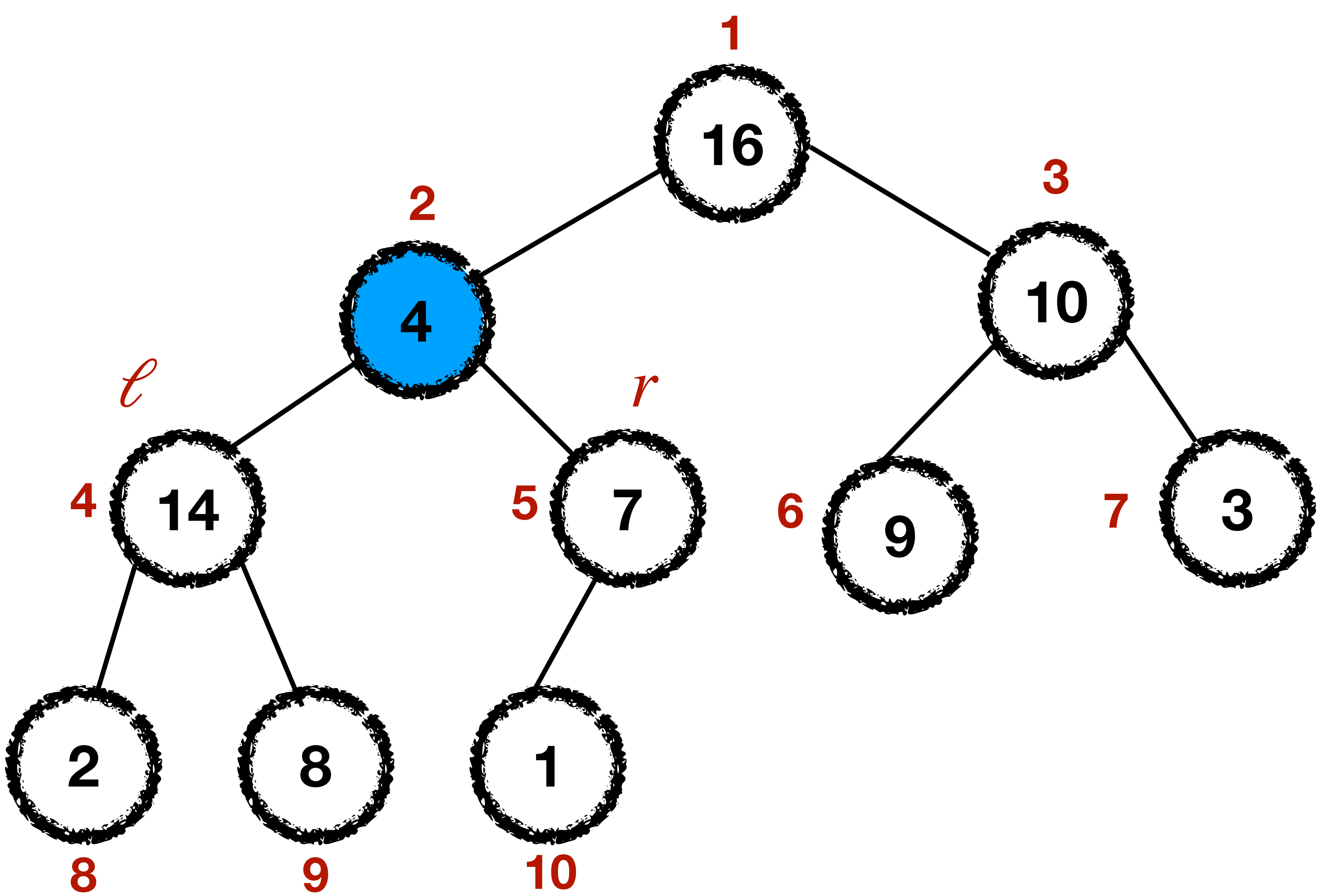
Max-Heapify




```
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)
```

$i = 2, \quad A[i] = 4$

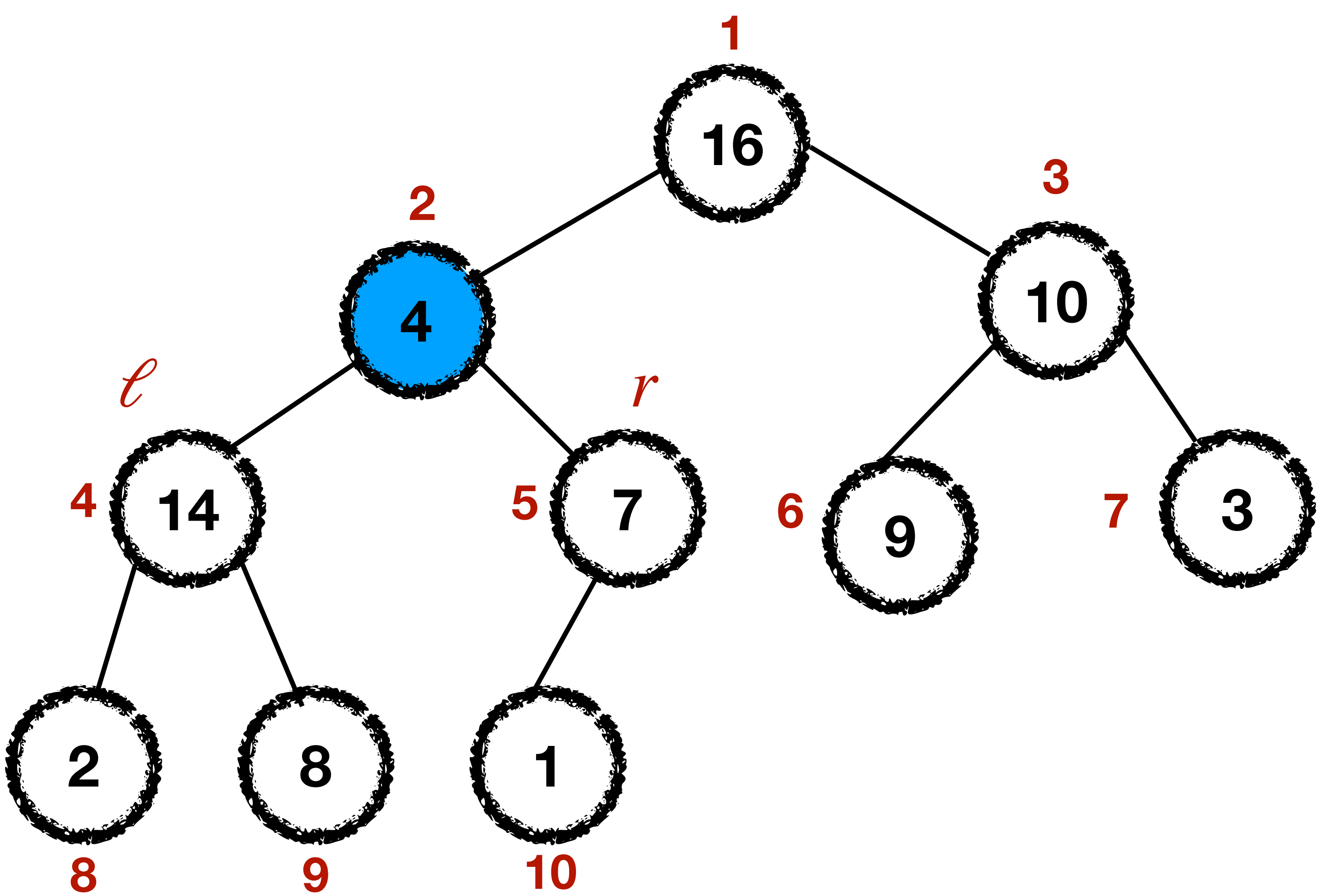
Max-Heapify




```
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)
```

$i = 2, \quad A[i] = 4$

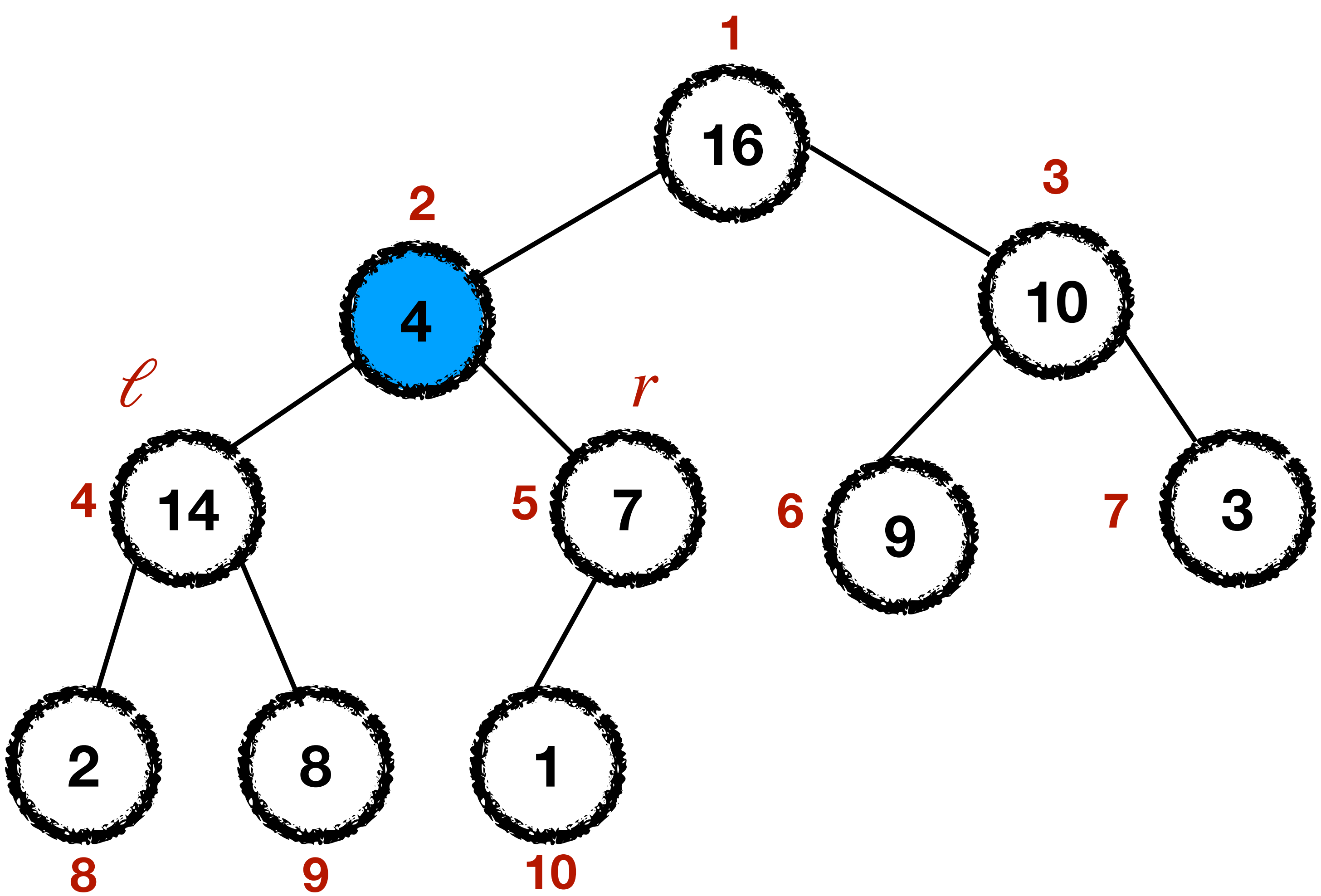
Max-Heapify



MAX-HEAPIFY(A, i) $\quad i = 2, \quad A[i] = 4$

l = LEFT(i)
r = RIGHT(i)
if $l \le A.heap\text{-}size$ and $A[l] > A[i]$ true
  $largest = l$
else $largest = i$
if $r \le A.heap\text{-}size$ and $A[r] > A[largest]$
  $largest = r$
if $largest \neq i$
  exchange $A[i]$ with $A[largest]$
  MAX-HEAPIFY$(A, largest)$

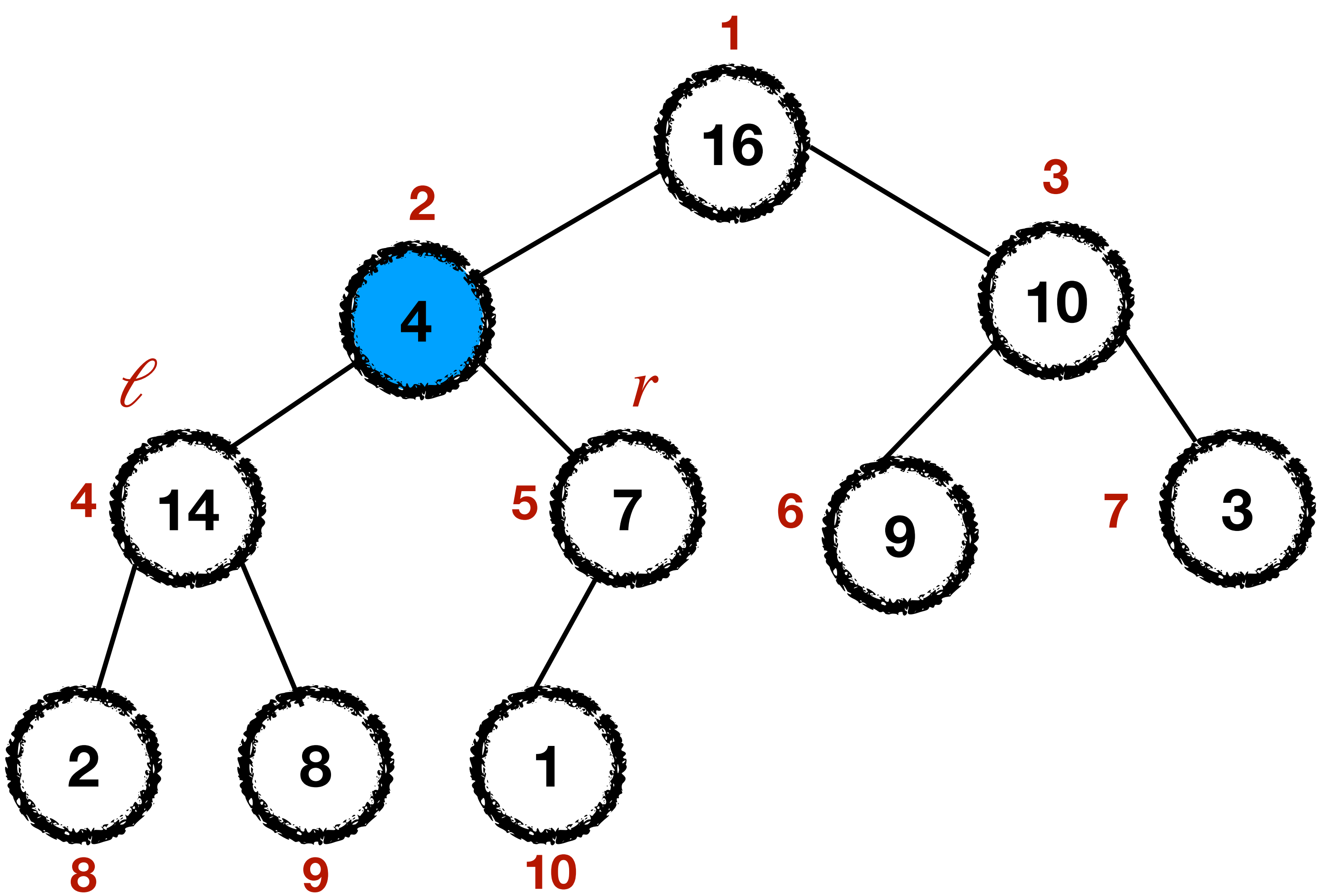
Max-Heapify



MAX-HEAPIFY(A, i) $\quad i = 2, \quad A[i] = 4$

```
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]      true
    largest = l                          largest = 4
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



MAX-HEAPIFY(A, i) $\quad i = 2, \quad A[i] = 4$

$l = \text{LEFT}(i)$

$r = \text{RIGHT}(i)$

if $l \le A.\textit{heap-size}$ and $A[l] > A[i]$ true

$\quad \textit{largest} = l$ largest = 4

else $\textit{largest} = i$

if $r \le A.\textit{heap-size}$ and $A[r] > A[\textit{largest}]$ false

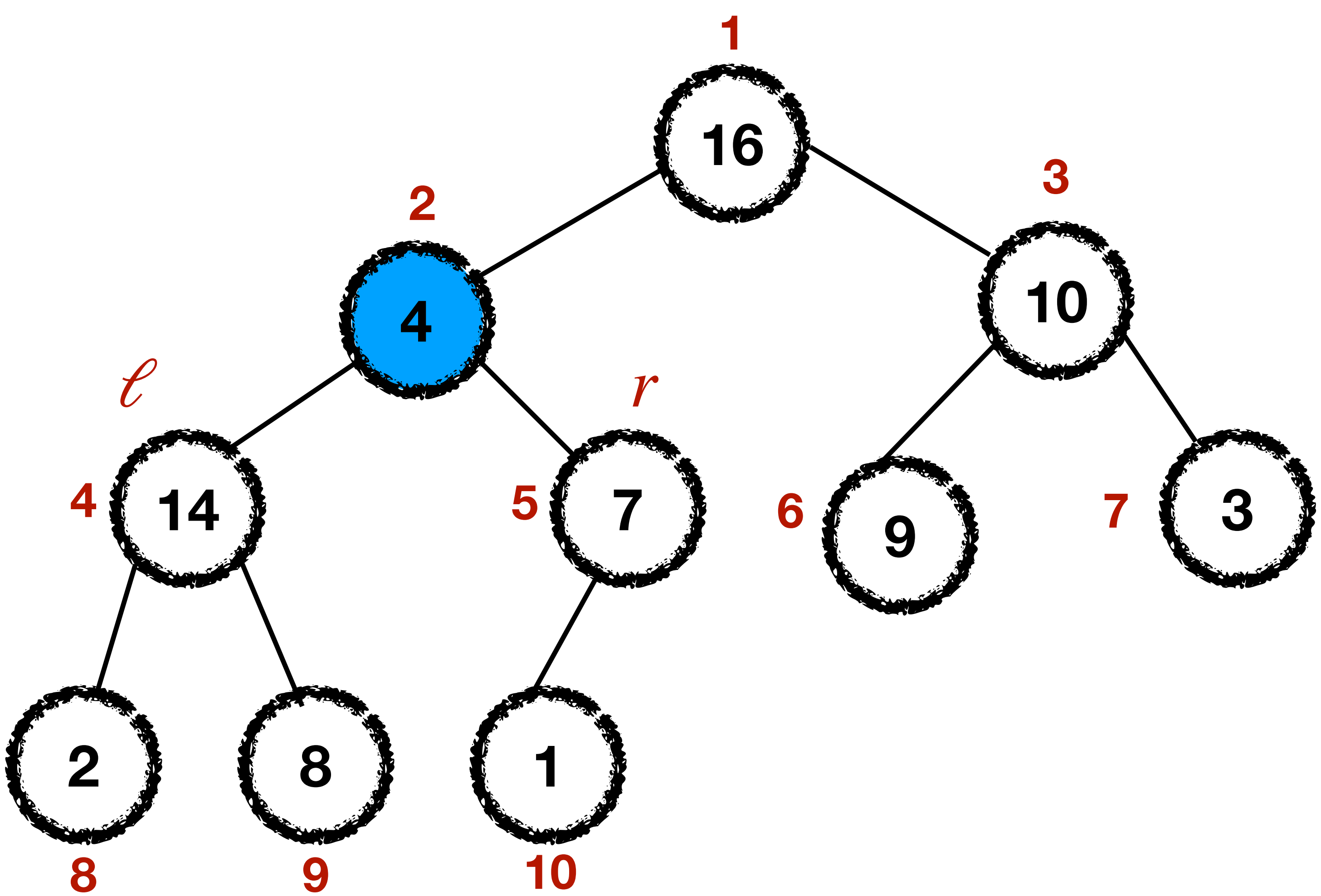
$\quad \textit{largest} = r$

if $\textit{largest} \ne i$

$\quad$ exchange $A[i]$ with $A[\textit{largest}]$

$\quad$ MAX-HEAPIFY$(A, \textit{largest})$

Max-Heapify



MAX-HEAPIFY(A, i) $\quad i = 2, \quad A[i] = 4$

$l = \text{LEFT}(i)$

$r = \text{RIGHT}(i)$

if $l \leq A.\textit{heap-size}$ and $A[l] > A[i]$ true

$\quad$ $\textit{largest} = l$ largest = 4

else $\textit{largest} = i$

if $r \leq A.\textit{heap-size}$ and $A[r] > A[\textit{largest}]$ false

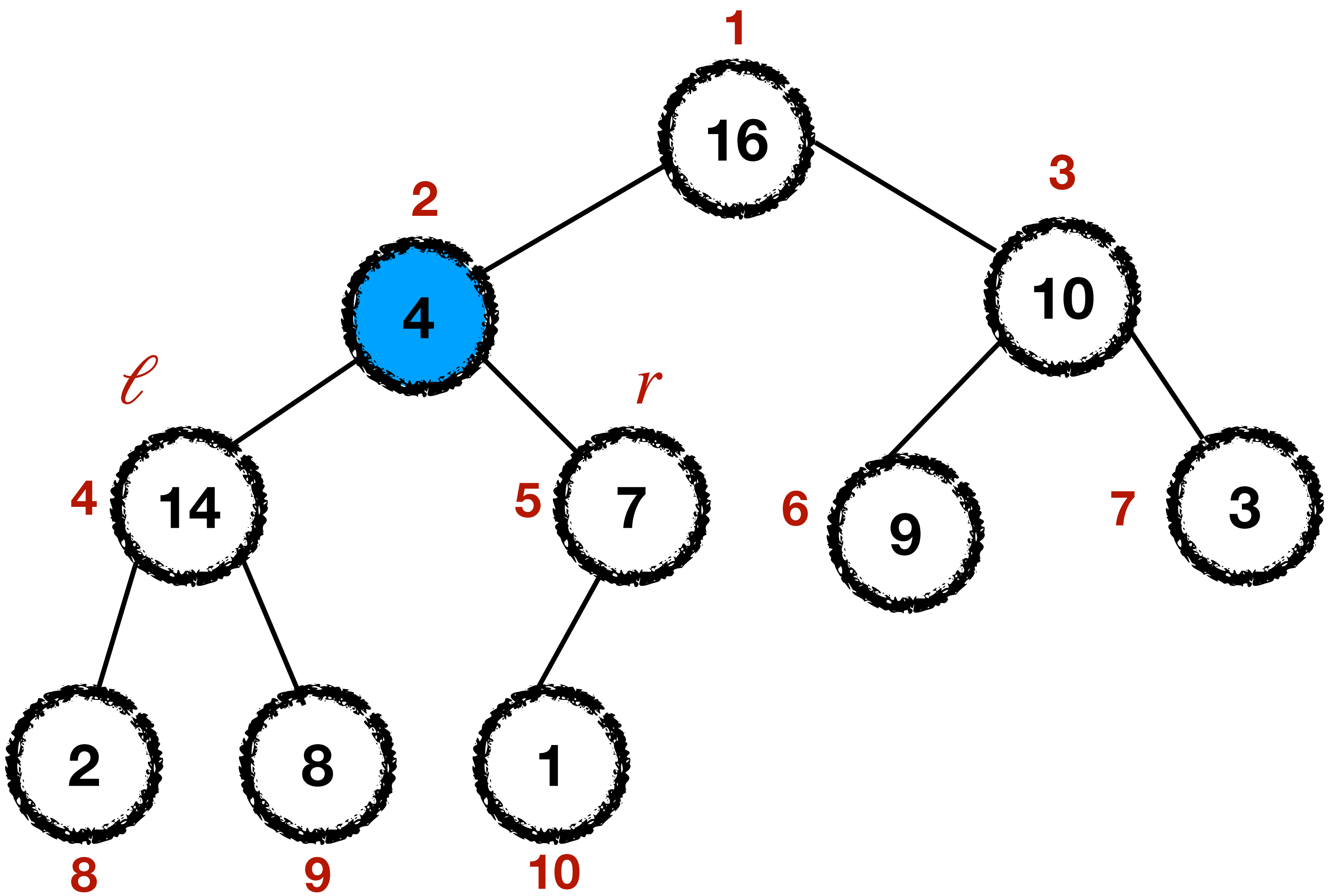
$\quad$ $\textit{largest} = r$

if $\textit{largest} \neq i$ true

$\quad$ exchange $A[i]$ with $A[\textit{largest}]$

$\quad$ MAX-HEAPIFY$(A, \textit{largest})$

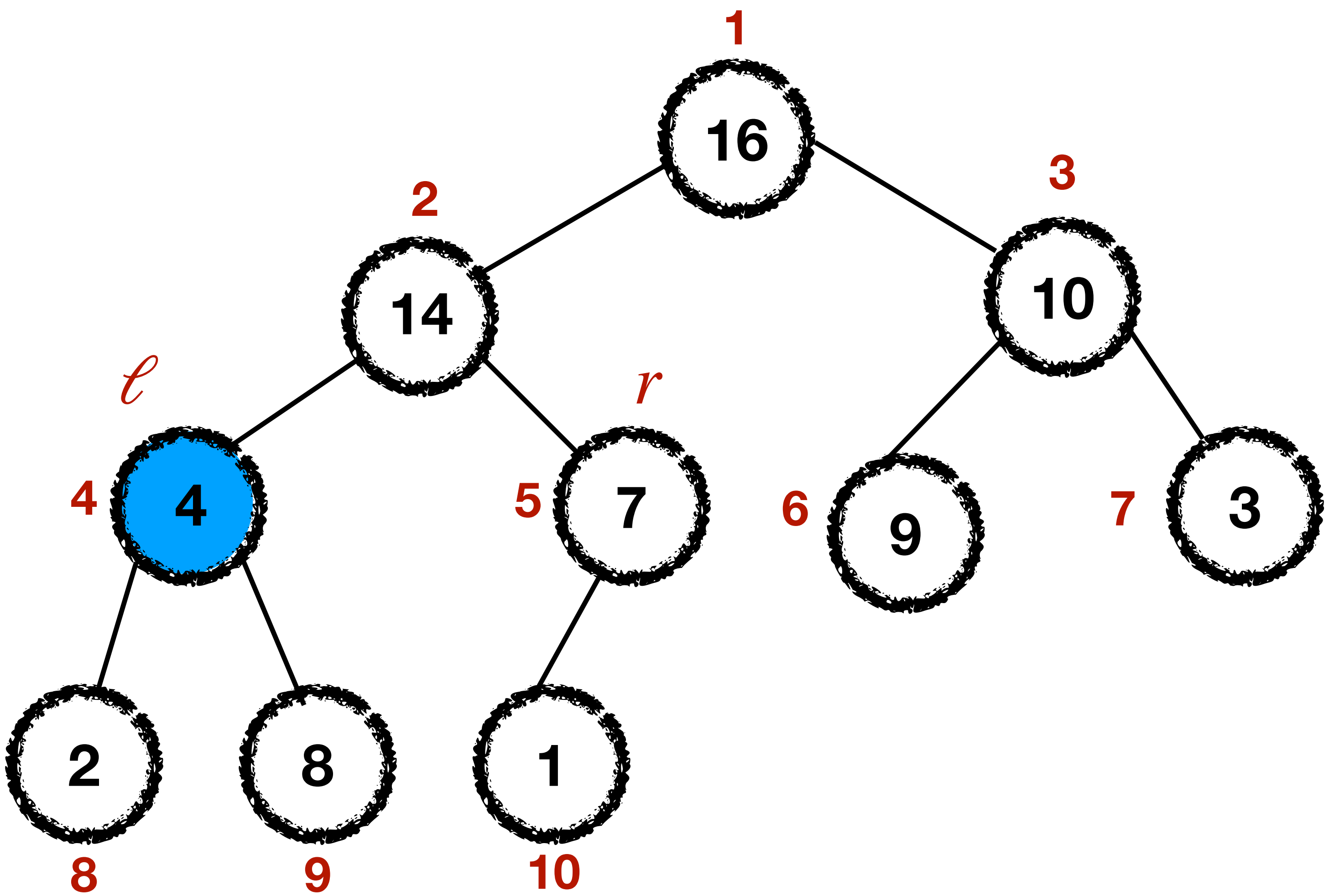
Max-Heapify



MAX-HEAPIFY(A, i) $\quad i = 2, \quad A[i] = 4$

```
l = LEFT(i)  →
r = RIGHT(i)  →
if l ≤ A.heap-size and A[l] > A[i]   true
    largest = l     largest = 4
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]   false
    largest = r
if largest ≠ i   true
    exchange A[i] with A[largest]  →
    MAX-HEAPIFY(A, largest)
```

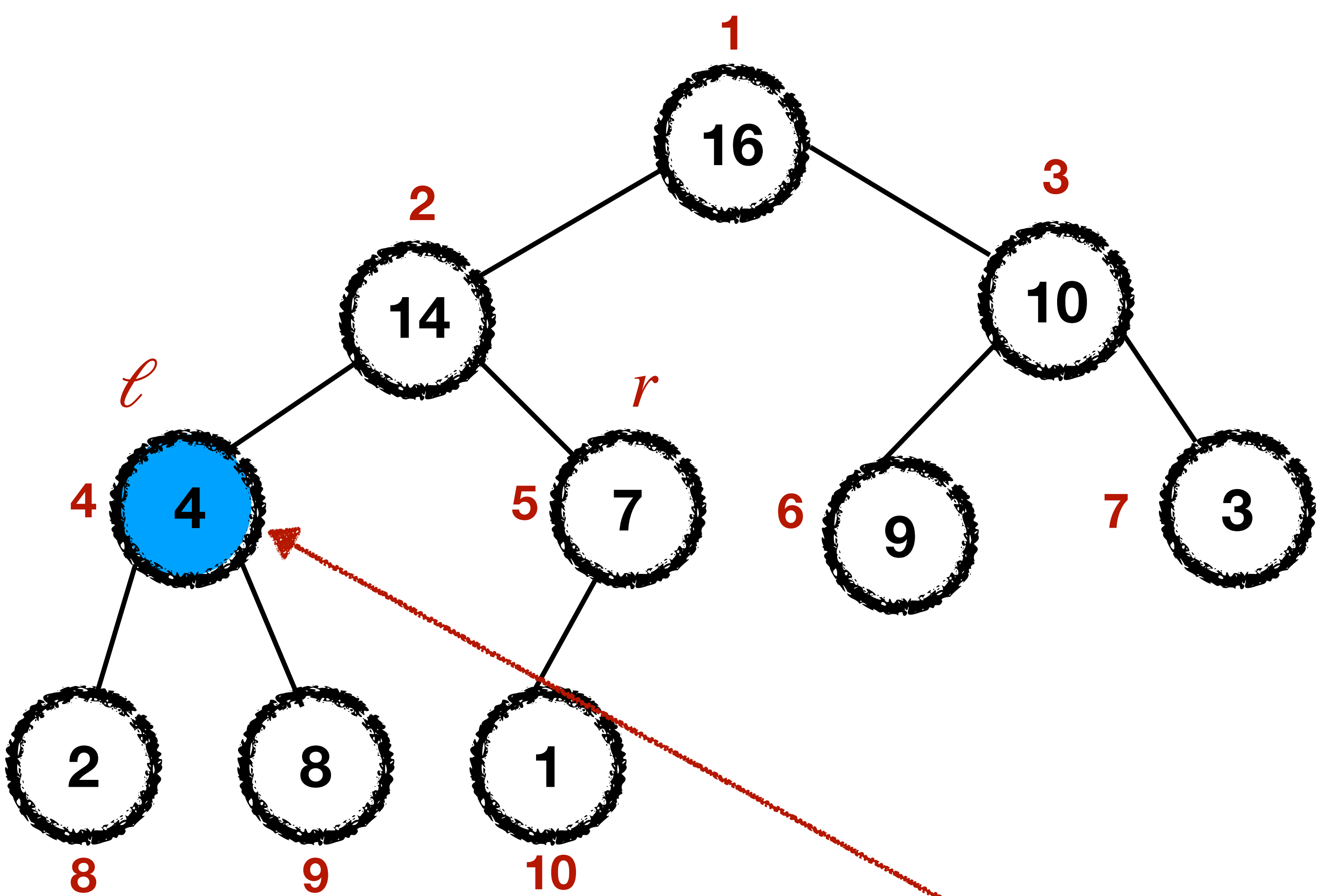
Max-Heapify



MAX-HEAPIFY(A, i) $i = 2, \quad A[i] = 4$

```
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]   true
    largest = l   largest = 4
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]   false
    largest = r
if largest ≠ i   true
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```

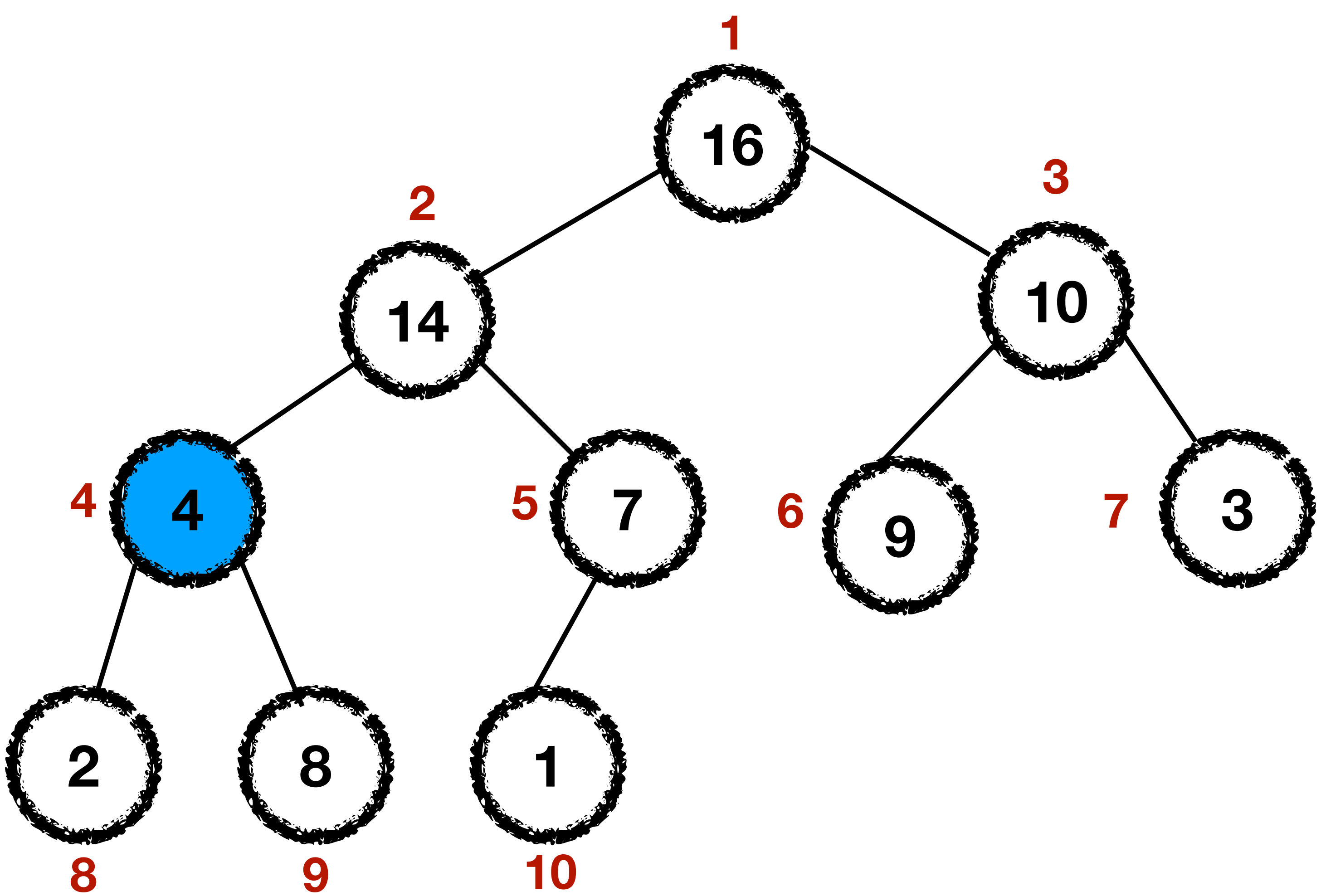
Max-Heapify



MAX-HEAPIFY(A, i) $\quad i = 2, \quad A[i] = 4$

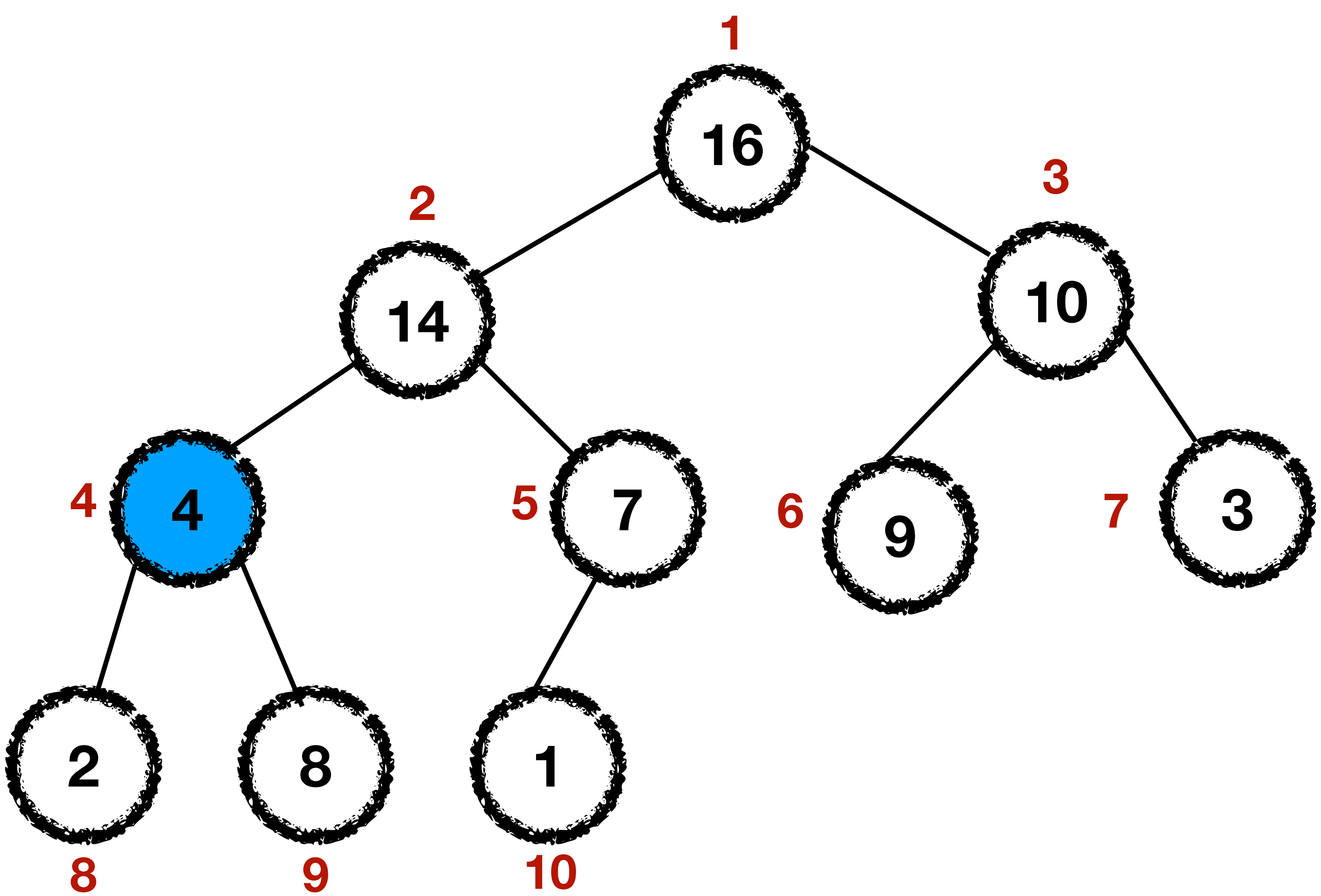
```
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]   true
    largest = l   largest = 4
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]   false
    largest = r
if largest ≠ i   true
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```

Max-Heapify (cont)




```
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)
```

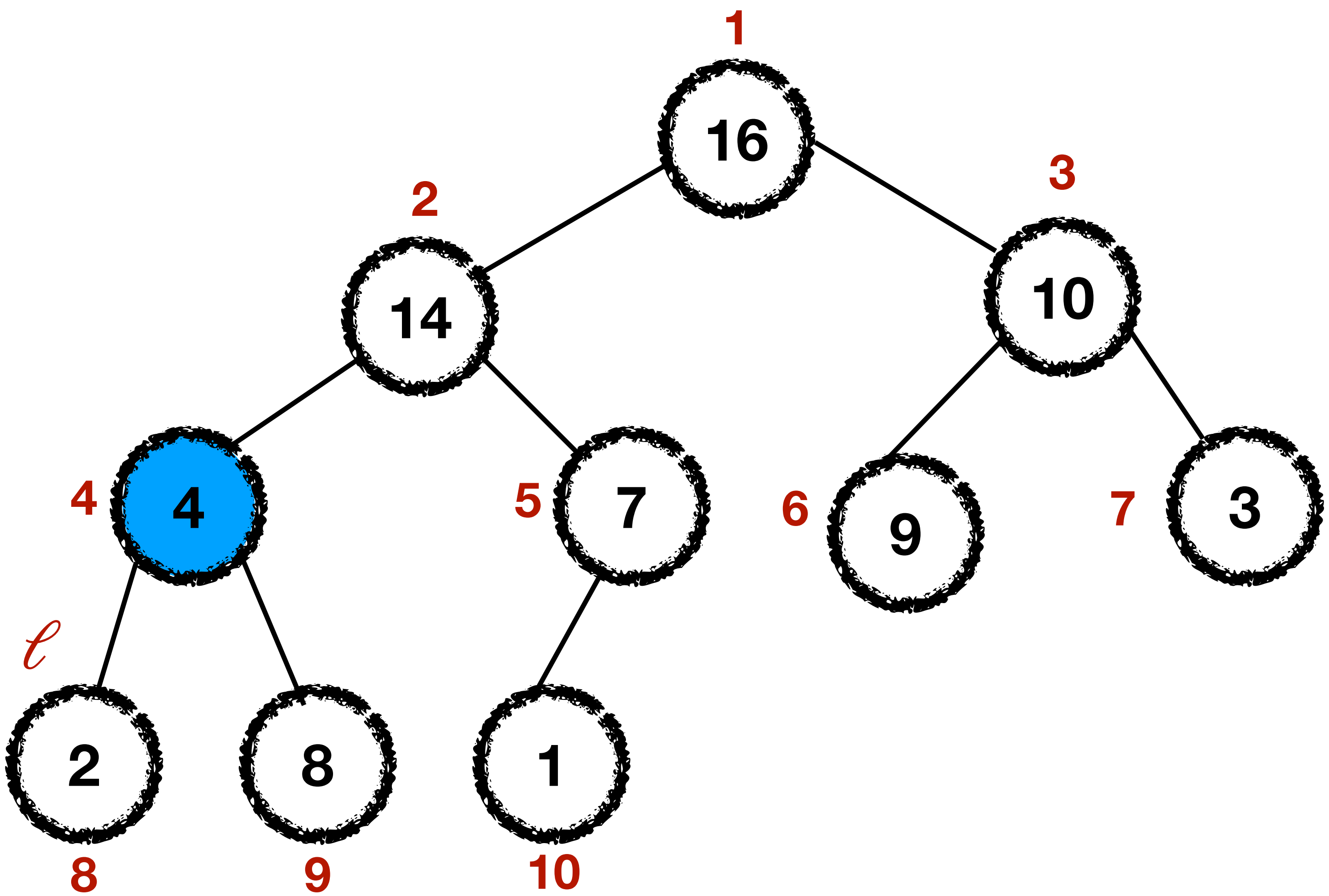
Max-Heapify (cont)



$i = 4, \quad A[i] = 4$

```
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)
```

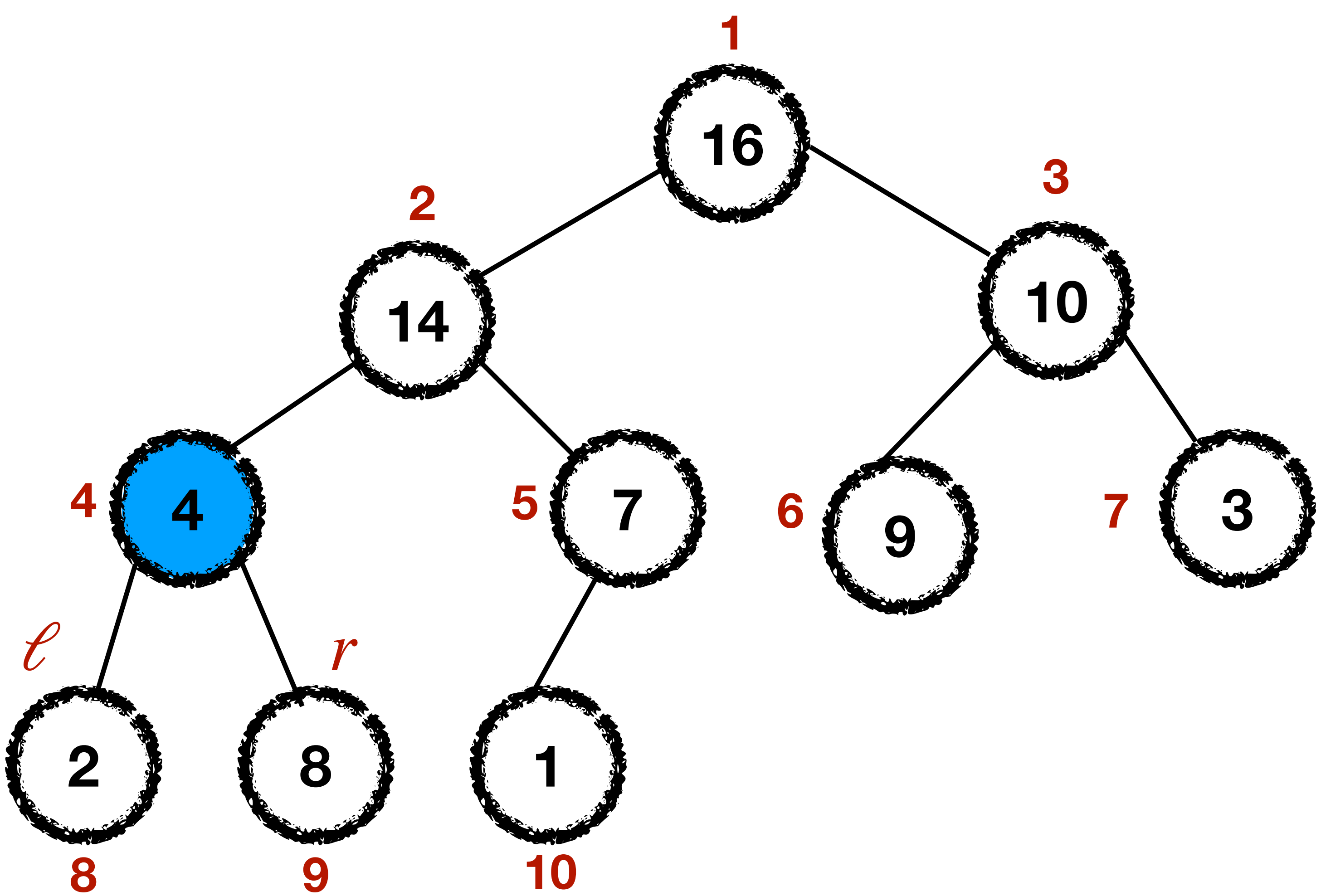
Max-Heapify (cont)



MAX-HEAPIFY(A, i) $\quad i = 4, \quad A[i] = 4$

```
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)
```

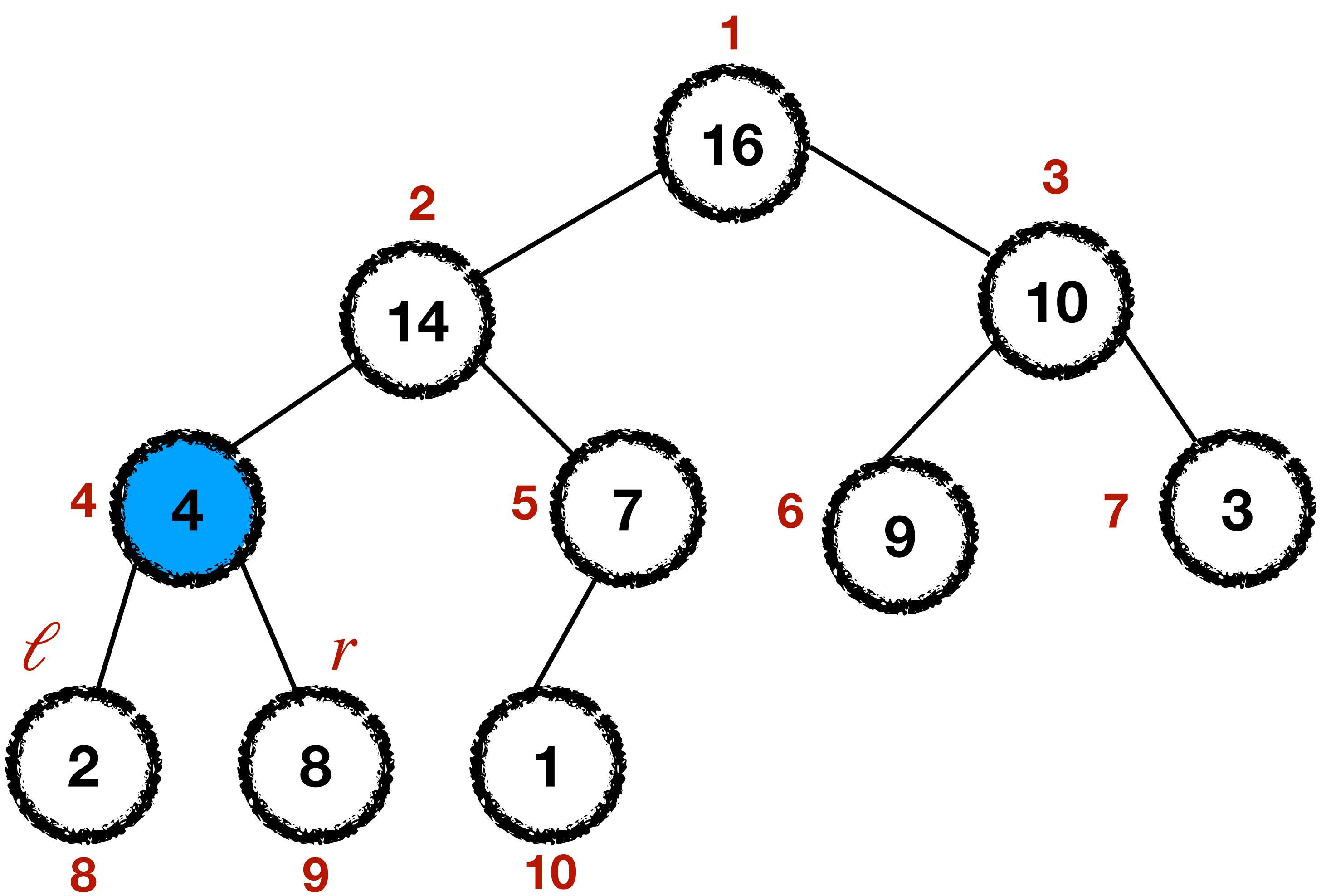
Max-Heapify (cont)



$i = 4, \quad A[i] = 4$

```
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)
```

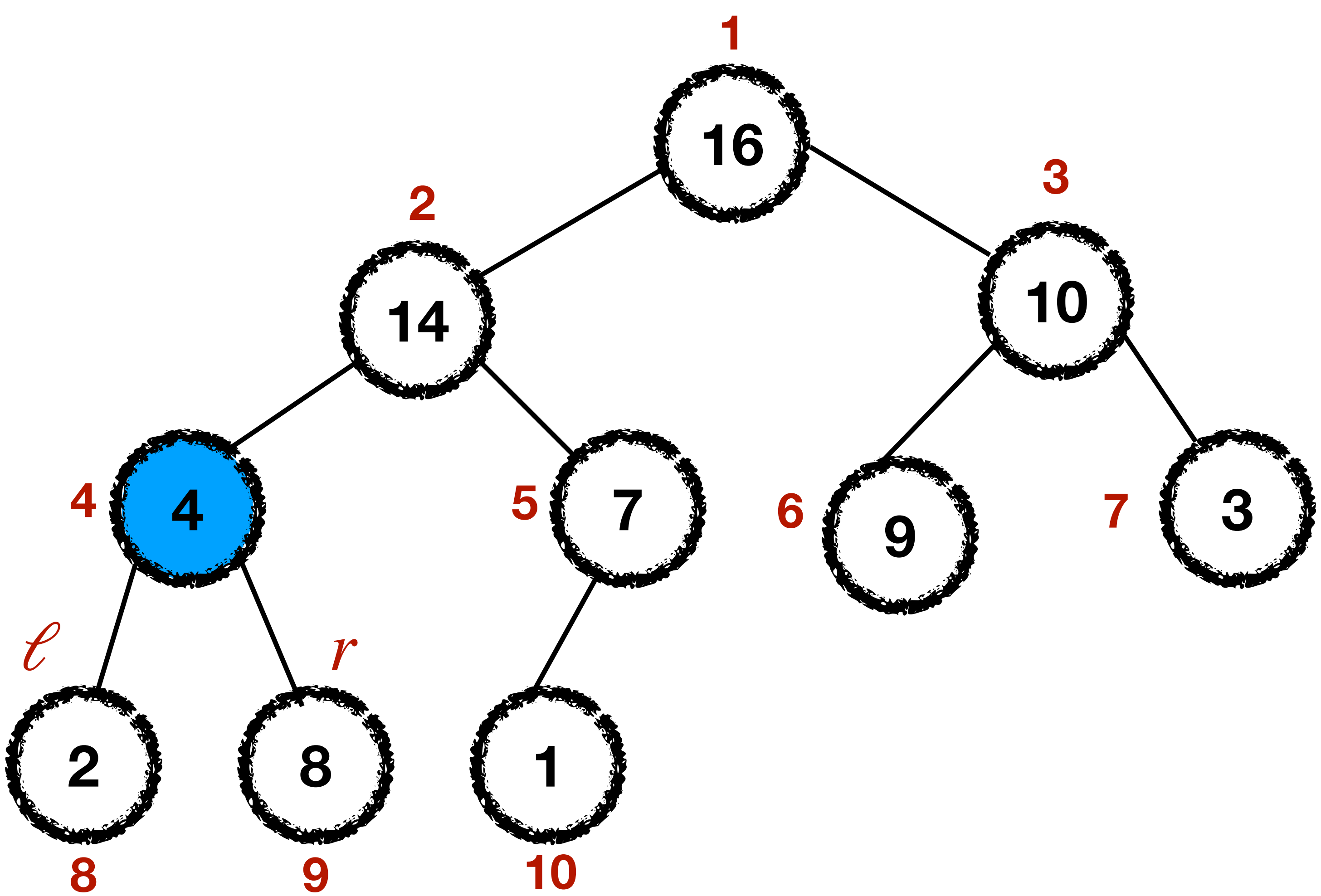
Max-Heapify (cont)



MAX-HEAPIFY(A, i) $i = 4, \quad A[i] = 4$

```
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]      false
    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)
```

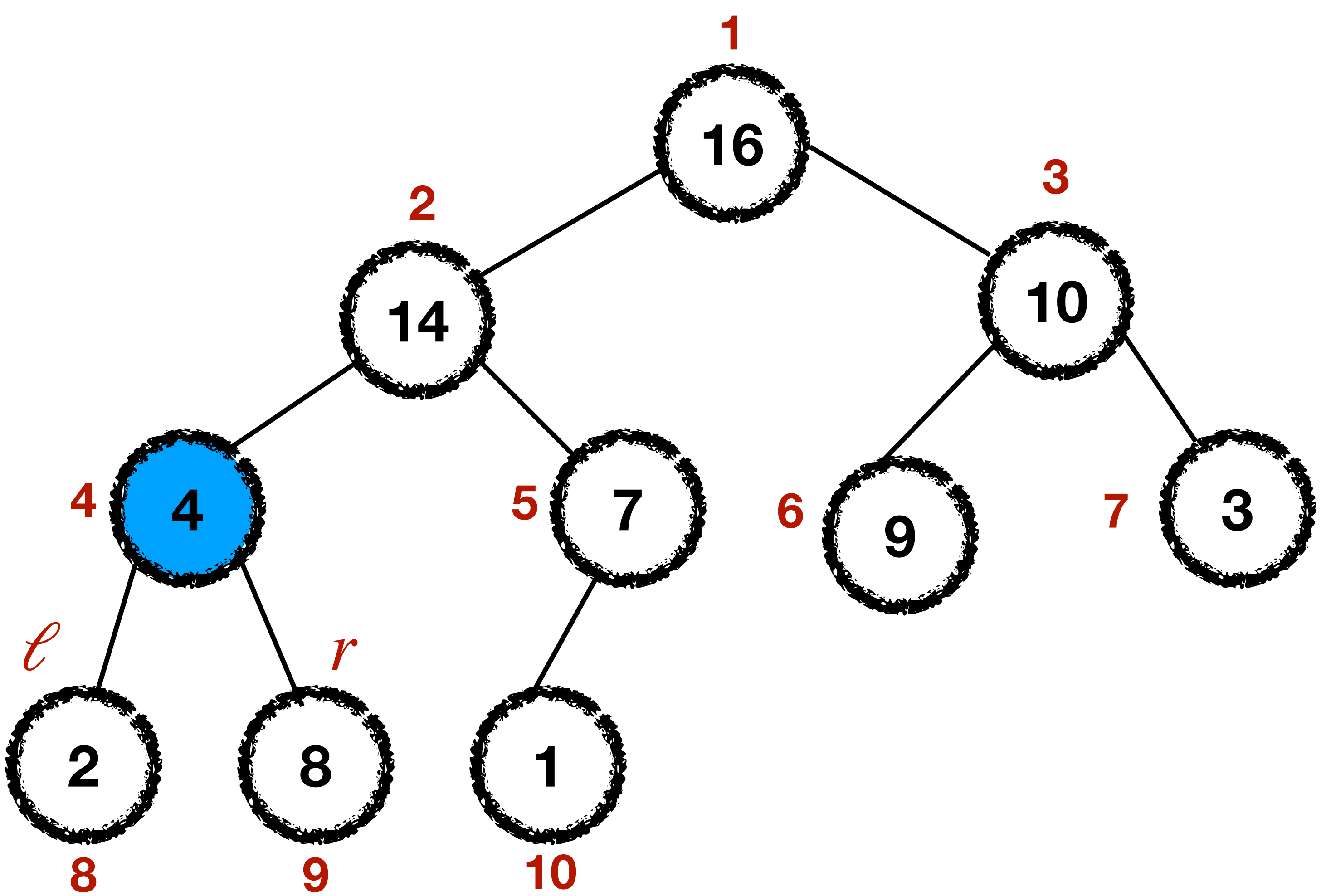
Max-Heapify (cont)



MAX-HEAPIFY(A, i) $\quad i = 4, \quad A[i] = 4$

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

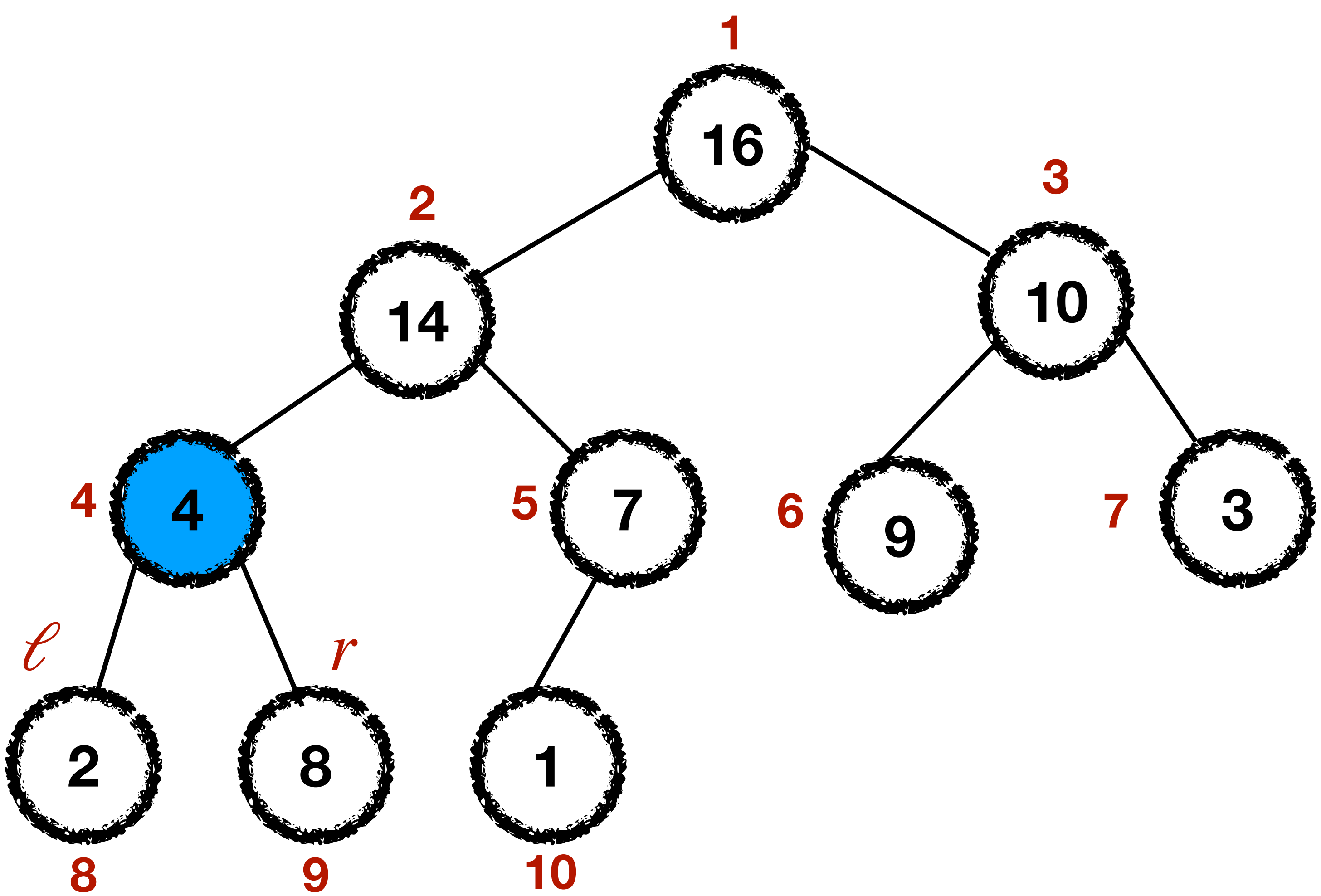
Max-Heapify (cont)



$i = 4, \quad A[i] = 4$

```
MAX-HEAPIFY(A, i)
    l = LEFT(i)
    r = RIGHT(i)
    if l ≤ A.heap-size and A[l] > A[i]        false
        largest = l
    else largest = i
    if r ≤ A.heap-size and A[r] > A[largest]  true
        largest = r                           largest = 9
    if largest ≠ i
        exchange A[i] with A[largest]
        MAX-HEAPIFY(A, largest)
```

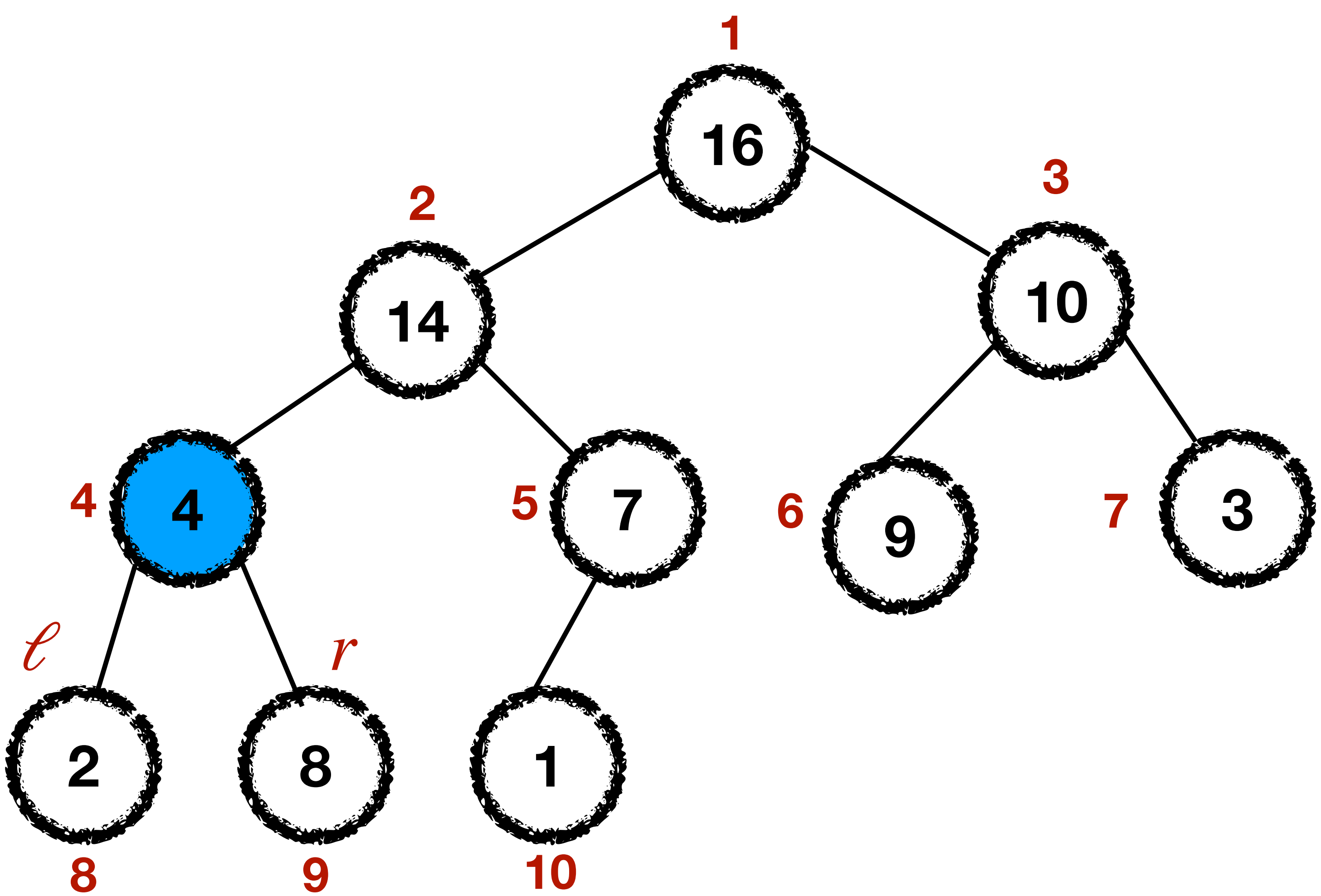
Max-Heapify (cont)



MAX-HEAPIFY(A, i) $\quad i = 4, \quad A[i] = 4$

$l = \text{LEFT}(i)$ $\rightarrow$
$r = \text{RIGHT}(i)$ $\rightarrow$
if $l \leq A.\textit{heap-size}$ and $A[l] > A[i]$ false
  $\textit{largest} = l$
else $\textit{largest} = i$
if $r \leq A.\textit{heap-size}$ and $A[r] > A[\textit{largest}]$ true
  $\textit{largest} = r$ largest = 9
if $\textit{largest} \neq i$ true
  exchange $A[i]$ with $A[\textit{largest}]$
  MAX-HEAPIFY$(A, \textit{largest})$

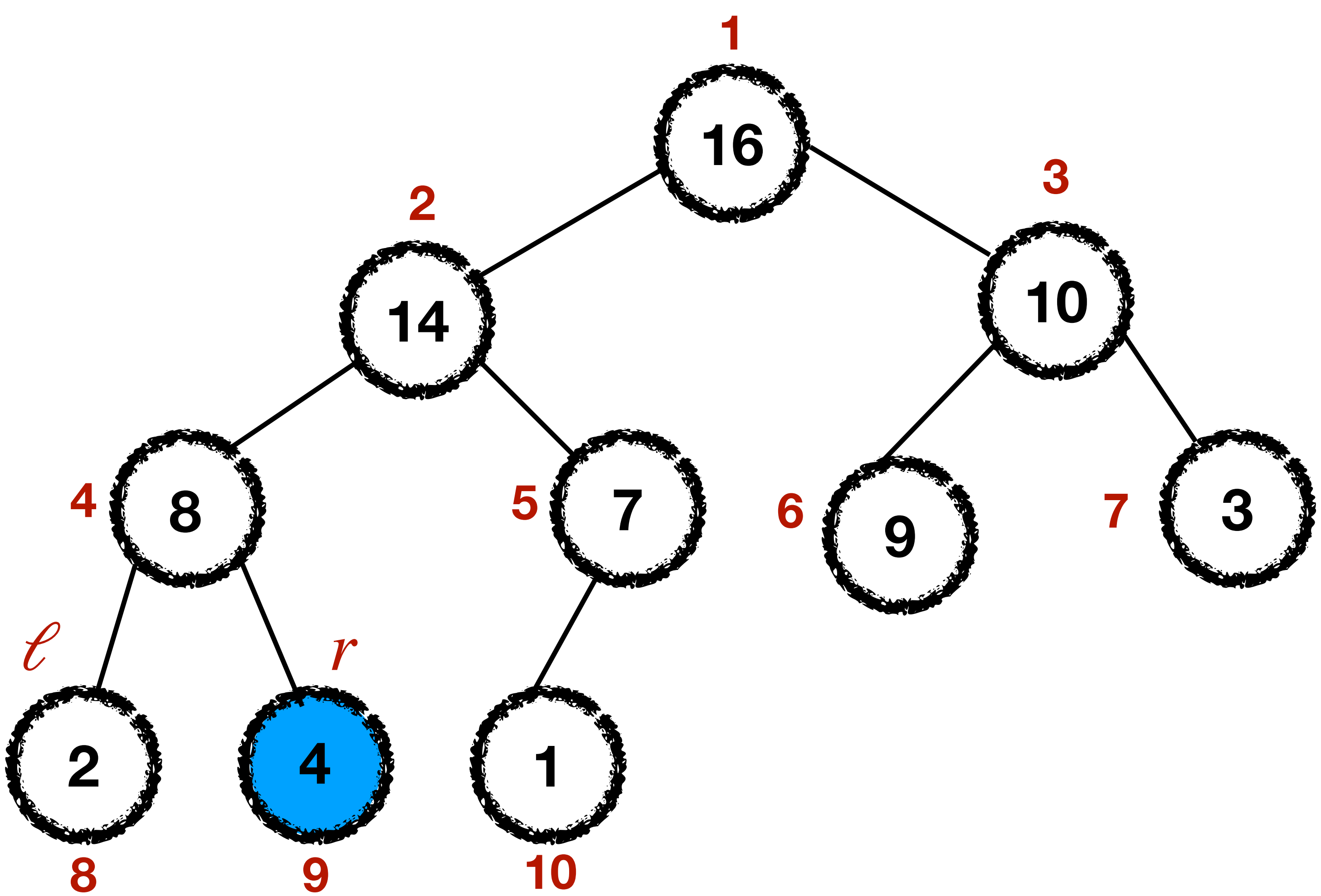
Max-Heapify (cont)



MAX-HEAPIFY(A, i) $i = 4, \quad A[i] = 4$

```
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]          false
    largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]    true
    largest = r                             largest = 9
if largest ≠ i                              true
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```

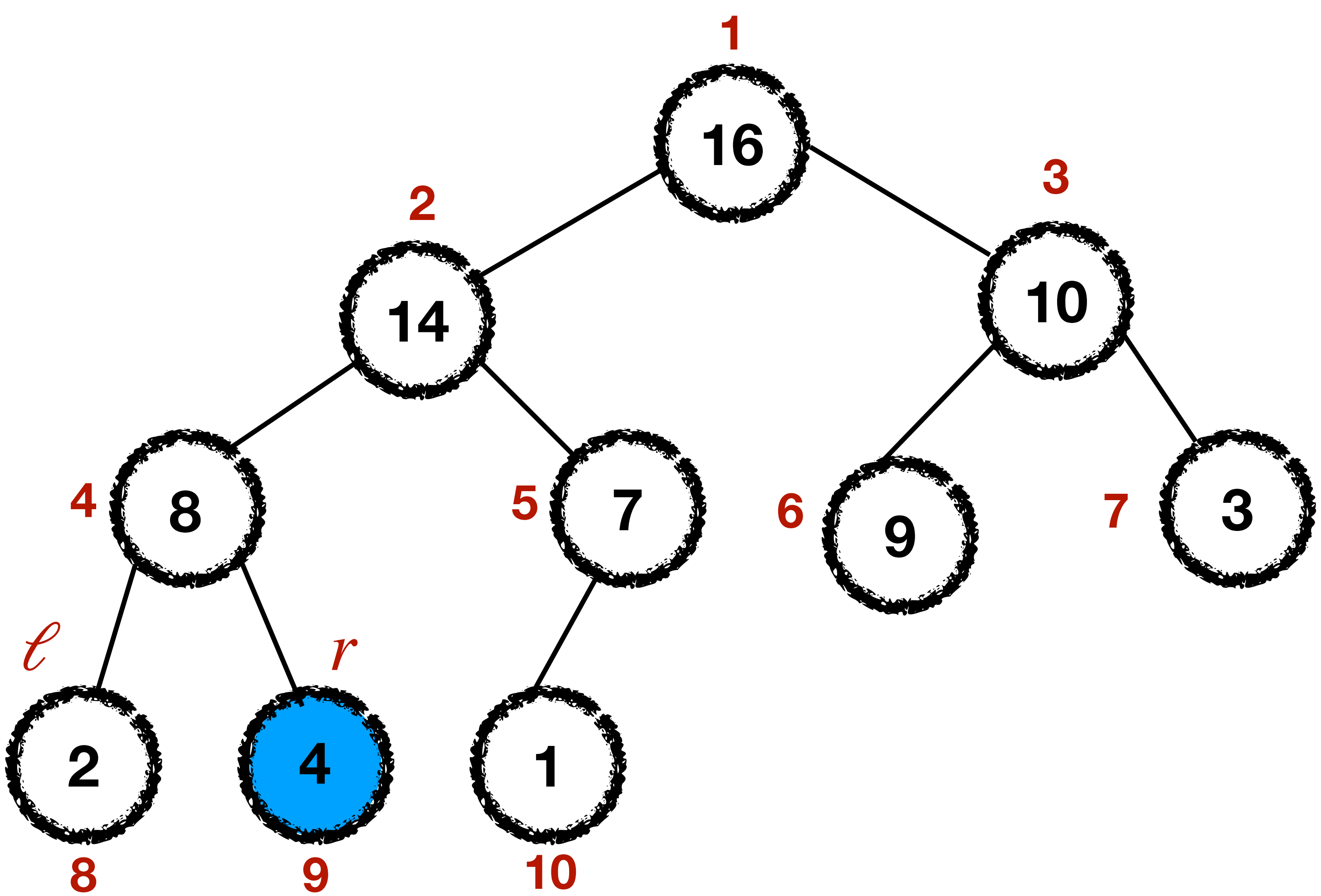
Max-Heapify (cont)



MAX-HEAPIFY(A, i) $\quad i = 4, \quad A[i] = 4$

```
l = LEFT(i)  →
r = RIGHT(i)  →
if l ≤ A.heap-size and A[l] > A[i]  false
    largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]  true
    largest = r   largest = 9
if largest ≠ i   true
    exchange A[i] with A[largest]  →
    MAX-HEAPIFY(A, largest)
```

Max-Heapify (cont)



Our tree is now a max-heap!

MAX-HEAPIFY(A, i) $\quad i = 4, \quad A[i] = 4$

l = LEFT(i) →
r = RIGHT(i) →
if $l \leq A.heap\text{-}size$ and $A[l] > A[i]$ false
  $largest = l$
else $largest = i$
if $r \leq A.heap\text{-}size$ and $A[r] > A[largest]$ true
  $largest = r$ largest = 9
if $largest \neq i$ true
  exchange $A[i]$ with $A[largest]$ →
  MAX-HEAPIFY$(A, largest)$

Max-Heapify running time

```
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 running time

What is the cost of an execution of Max-Heapify?

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

Max-Heapify running time

What is the cost of an execution of Max-Heapify?

```
MAX-HEAPIFY(A, i)
l = LEFT(i)        O(1)
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 running time

What is the cost of an execution of Max-Heapify?

```
MAX-HEAPIFY(A, i)
  l = LEFT(i)        O(1)
  r = RIGHT(i)       O(1)
  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 running time

What is the cost of an execution of Max-Heapify?

```
MAX-HEAPIFY(A, i)
l = LEFT(i)        O(1)
r = RIGHT(i)       O(1)
if l ≤ A.heap-size and A[l] > A[i]   O(1)
    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 running time

What is the cost of an execution of Max-Heapify?

```
MAX-HEAPIFY(A, i)
l = LEFT(i)    O(1)
r = RIGHT(i)   O(1)
if l ≤ A.heap-size and A[l] > A[i]   O(1)
    largest = l   O(1)
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 running time

What is the cost of an execution of Max-Heapify?

```
MAX-HEAPIFY(A, i)
  l = LEFT(i)                                 O(1)
  r = RIGHT(i)                                O(1)
  if l ≤ A.heap-size and A[l] > A[i]          O(1)
        largest = l                           O(1)
  else largest = i                            O(1)
  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 running time

What is the cost of an execution of Max-Heapify?

```
MAX-HEAPIFY(A, i)
  l = LEFT(i)    O(1)
  r = RIGHT(i)   O(1)
  if l ≤ A.heap-size and A[l] > A[i]  O(1)
      largest = l   O(1)
  else largest = i  O(1)
  if r ≤ A.heap-size and A[r] > A[largest]  O(1)
      largest = r
  if largest ≠ i
      exchange A[i] with A[largest]
      MAX-HEAPIFY(A, largest)
```

Max-Heapify running time

What is the cost of an execution of Max-Heapify?

```
MAX-HEAPIFY(A, i)
l = LEFT(i)    O(1)
r = RIGHT(i)   O(1)
if l ≤ A.heap-size and A[l] > A[i]  O(1)
    largest = l   O(1)
else largest = i  O(1)
if r ≤ A.heap-size and A[r] > A[largest]  O(1)
    largest = r   O(1)
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```

Max-Heapify running time

What is the cost of an execution of Max-Heapify?

```
MAX-HEAPIFY(A, i)
l = LEFT(i)    O(1)
r = RIGHT(i)   O(1)
if l ≤ A.heap-size and A[l] > A[i]  O(1)
        largest = l   O(1)
else largest = i   O(1)
if r ≤ A.heap-size and A[r] > A[largest]  O(1)
        largest = r   O(1)
if largest ≠ i   O(1)
        exchange A[i] with A[largest]
        MAX-HEAPIFY(A, largest)
```

Max-Heapify running time

What is the cost of an execution of Max-Heapify?

```
MAX-HEAPIFY(A, i)
l = LEFT(i)   O(1)
r = RIGHT(i)   O(1)
if l ≤ A.heap-size and A[l] > A[i]   O(1)
    largest = l   O(1)
else largest = i   O(1)
if r ≤ A.heap-size and A[r] > A[largest]   O(1)
    largest = r   O(1)
if largest ≠ i   O(1)
    exchange A[i] with A[largest]   O(1)
    MAX-HEAPIFY(A, largest)
```

Max-Heapify running time

What is the cost of an execution of Max-Heapify?

All “standard” operations can be done in $O(1)$ time.

MAX-HEAPIFY(A, i)

$l = \text{LEFT}(i)$ $O(1)$
$r = \text{RIGHT}(i)$ $O(1)$
if $l \leq A.\mathit{heap\text{-}size}$ and $A[l] > A[i]$ $O(1)$
    $\mathit{largest} = l$ $O(1)$
else $\mathit{largest} = i$ $O(1)$
if $r \leq A.\mathit{heap\text{-}size}$ and $A[r] > A[\mathit{largest}]$ $O(1)$
    $\mathit{largest} = r$ $O(1)$
if $\mathit{largest} \neq i$ $O(1)$
    exchange $A[i]$ with $A[\mathit{largest}]$ $O(1)$
    MAX-HEAPIFY$(A, \mathit{largest})$

Max-Heapify running time

What is the cost of an execution of Max-Heapify?

All “standard” operations can be done in $O(1)$ time.

```
MAX-HEAPIFY(A, i)
l = LEFT(i)    O(1)
r = RIGHT(i)    O(1)
if l ≤ A.heap-size and A[l] > A[i]    O(1)
    largest = l    O(1)
else largest = i    O(1)
if r ≤ A.heap-size and A[r] > A[largest]    O(1)
    largest = r    O(1)
if largest ≠ i    O(1)
    exchange A[i] with A[largest]    O(1)
    MAX-HEAPIFY(A, largest)
```

Max-Heapify running time

What is the cost of an execution of Max-Heapify?

All “standard” operations can be done in $O(1)$ time.

Plus the time needed for the recursive call of Max-Heapify on the child of node i.

```
MAX-HEAPIFY(A, i)
  l = LEFT(i)    O(1)
  r = RIGHT(i)   O(1)
  if l ≤ A.heap-size and A[l] > A[i]   O(1)
      largest = l   O(1)
  else largest = i   O(1)
  if r ≤ A.heap-size and A[r] > A[largest]   O(1)
      largest = r   O(1)
  if largest ≠ i   O(1)
      exchange A[i] with A[largest]   O(1)
      MAX-HEAPIFY(A, largest)
```

Max-Heapify running time

What is the cost of an execution of Max-Heapify?

All “standard” operations can be done in $O(1)$ time.

Plus the time needed for the recursive call of Max-Heapify on the child of node i.

```
MAX-HEAPIFY(A, i)   T(h)
  l = LEFT(i)   O(1)
  r = RIGHT(i)   O(1)
  if l ≤ A.heap-size and A[l] > A[i]   O(1)
      largest = l   O(1)
  else largest = i   O(1)
  if r ≤ A.heap-size and A[r] > A[largest]   O(1)
      largest = r   O(1)
  if largest ≠ i   O(1)
      exchange A[i] with A[largest]   O(1)
      MAX-HEAPIFY(A, largest)
```

Max-Heapify running time

What is the cost of an execution of Max-Heapify?

All "standard" operations can be done in $O(1)$ time.

Plus the time needed for the recursive call of Max-Heapify on the child of node i.

```
MAX-HEAPIFY(A, i)    T(h)
  l = LEFT(i)    O(1)
  r = RIGHT(i)    O(1)
  if l ≤ A.heap-size and A[l] > A[i]    O(1)
      largest = l    O(1)
  else largest = i    O(1)
  if r ≤ A.heap-size and A[r] > A[largest]    O(1)
      largest = r    O(1)
  if largest ≠ i    O(1)
      exchange A[i] with A[largest]    O(1)
      MAX-HEAPIFY(A, largest)    T(h − 1)
```

Max-Heapify running time

What is the cost of an execution of Max-Heapify?

All "standard" operations can be done in $O(1)$ time.

Plus the time needed for the recursive call of Max-Heapify on the child of node i.

```
MAX-HEAPIFY(A, i)   T(h)
l = LEFT(i)   O(1)
r = RIGHT(i)   O(1)
if l ≤ A.heap-size and A[l] > A[i]   O(1)
      largest = l   O(1)
else largest = i   O(1)
if r ≤ A.heap-size and A[r] > A[largest]   O(1)
      largest = r   O(1)
if largest ≠ i   O(1)
      exchange A[i] with A[largest]   O(1)
      MAX-HEAPIFY(A, largest)   T(h − 1)
```

$$T(h) \leq \begin{cases} T(h-1) + O(1), & \textbf{if} \quad h \geq 1 \\ O(1) & \textbf{if} \quad h = 0 \end{cases}$$

Max-Heapify running time

What is the cost of an execution of Max-Heapify?

All "standard" operations can be done in $O(1)$ time.

Plus the time needed for the recursive call of Max-Heapify on the child of node i.

$$T(h) \leq (h+1) \cdot O(1)$$
$$= O(h) = O(\lg n)$$

```
MAX-HEAPIFY(A, i)                                   T(h)
l = LEFT(i)                                         O(1)
r = RIGHT(i)                                        O(1)
if l ≤ A.heap-size and A[l] > A[i]                  O(1)
    largest = l                                     O(1)
else largest = i                                    O(1)
if r ≤ A.heap-size and A[r] > A[largest]            O(1)
    largest = r                                     O(1)
if largest ≠ i                                      O(1)
    exchange A[i] with A[largest]                   O(1)
    MAX-HEAPIFY(A, largest)                         T(h − 1)
```

$$T(h) \leq \begin{cases} T(h-1) + O(1), & \textbf{if} \quad h \geq 1 \\ O(1) \quad \textbf{if} \quad h = 0 \end{cases}$$

Build-Max-Heap

1. *Preprocess the array to become a heap.*

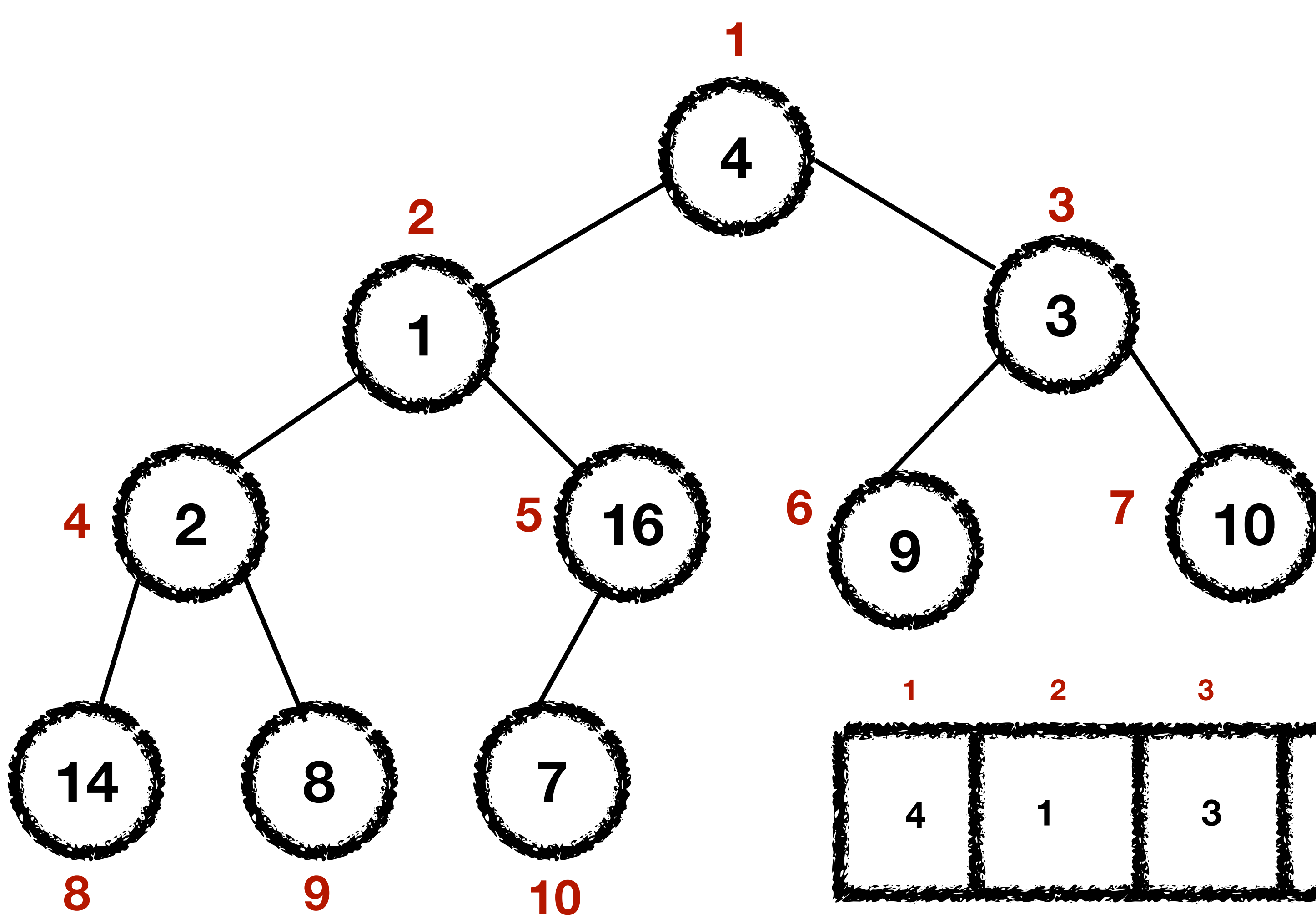
Build-Max-Heap

1. *Preprocess the array to become a heap.*

Idea: Apply Max-Heapify repeatedly until the tree becomes a heap.

Build-Max-Heap

Idea: Apply Max-Heapify repeatedly until the tree becomes a heap.

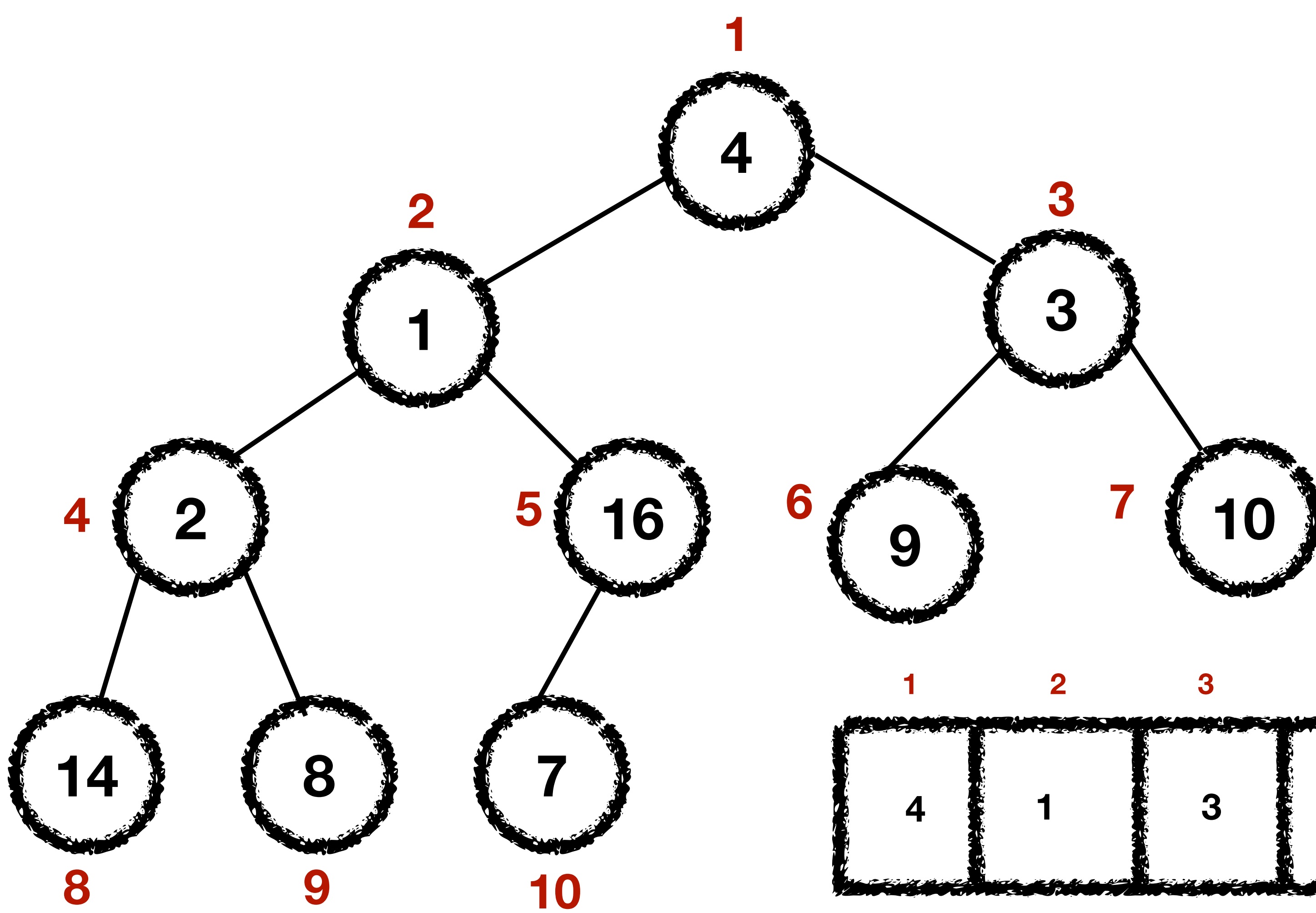


1	2	3	4	5	6	7	8	9	10
4	1	3	2	16	9	10	14	8	7

Build-Max-Heap

Idea: Apply Max-Heapify repeatedly until the tree becomes a heap.

Where do we start?



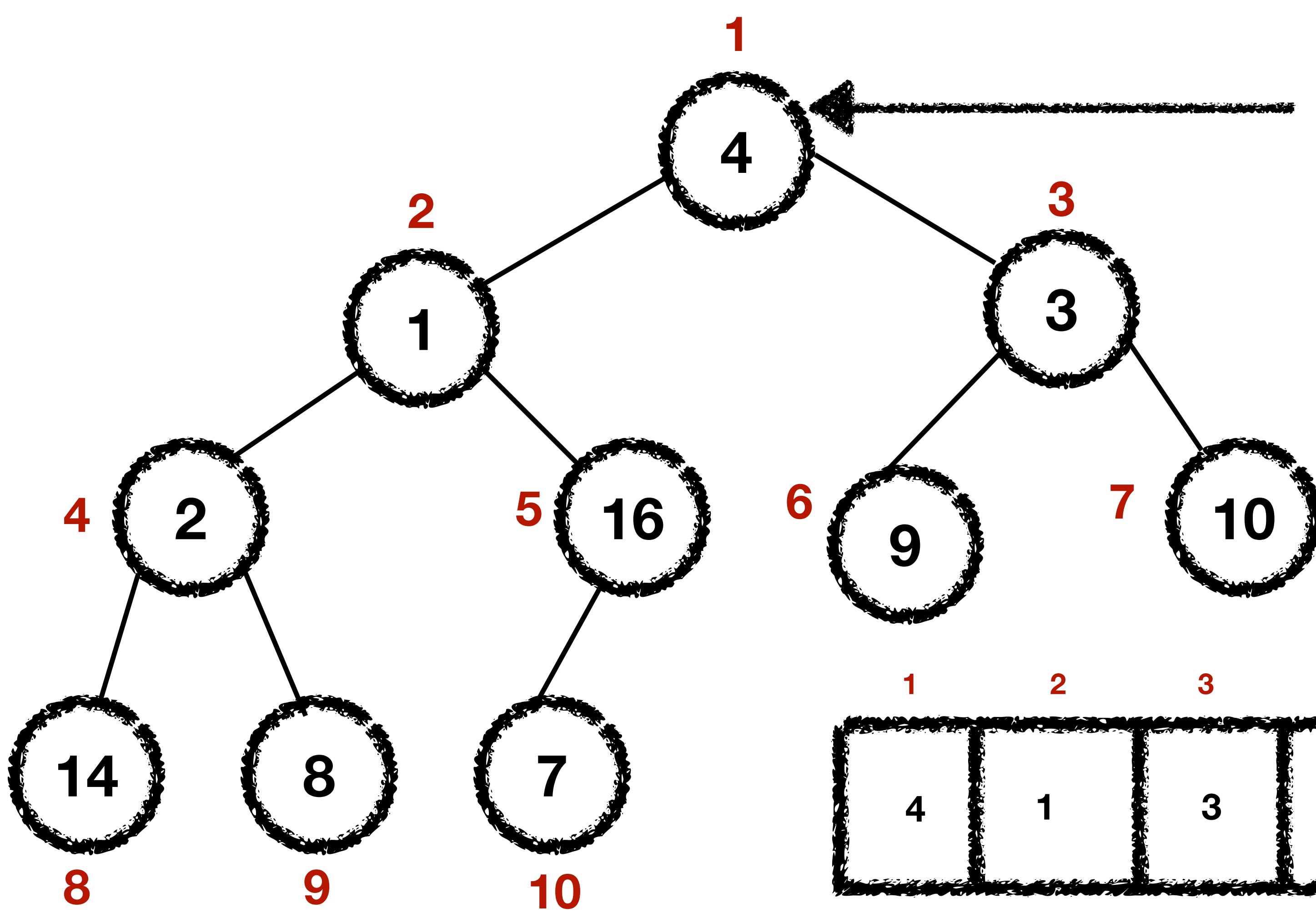
1	2	3	4	5	6	7	8	9	10
4	1	3	2	16	9	10	14	8	7

Build-Max-Heap

Idea: Apply Max-Heapify repeatedly until the tree becomes a heap.

Where do we start?

Can we start here?



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

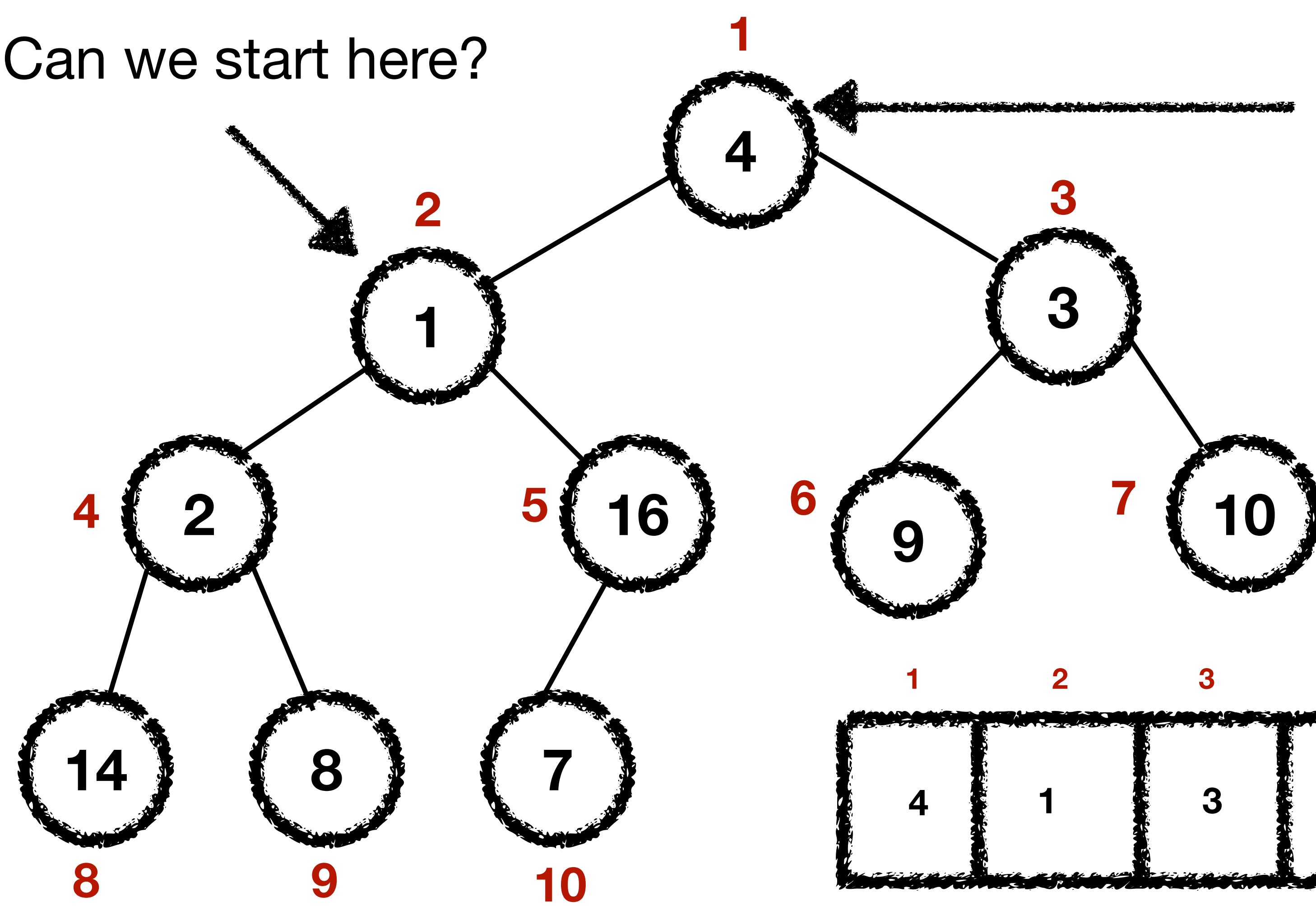
Build-Max-Heap

Idea: Apply Max-Heapify repeatedly until the tree becomes a heap.

Where do we start?

Can we start here?

Can we start here?



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

Max-Heapify

Max-Heapify(A, i).

Precondition: Trees rooted at Left(i) and Right(i) are heaps.

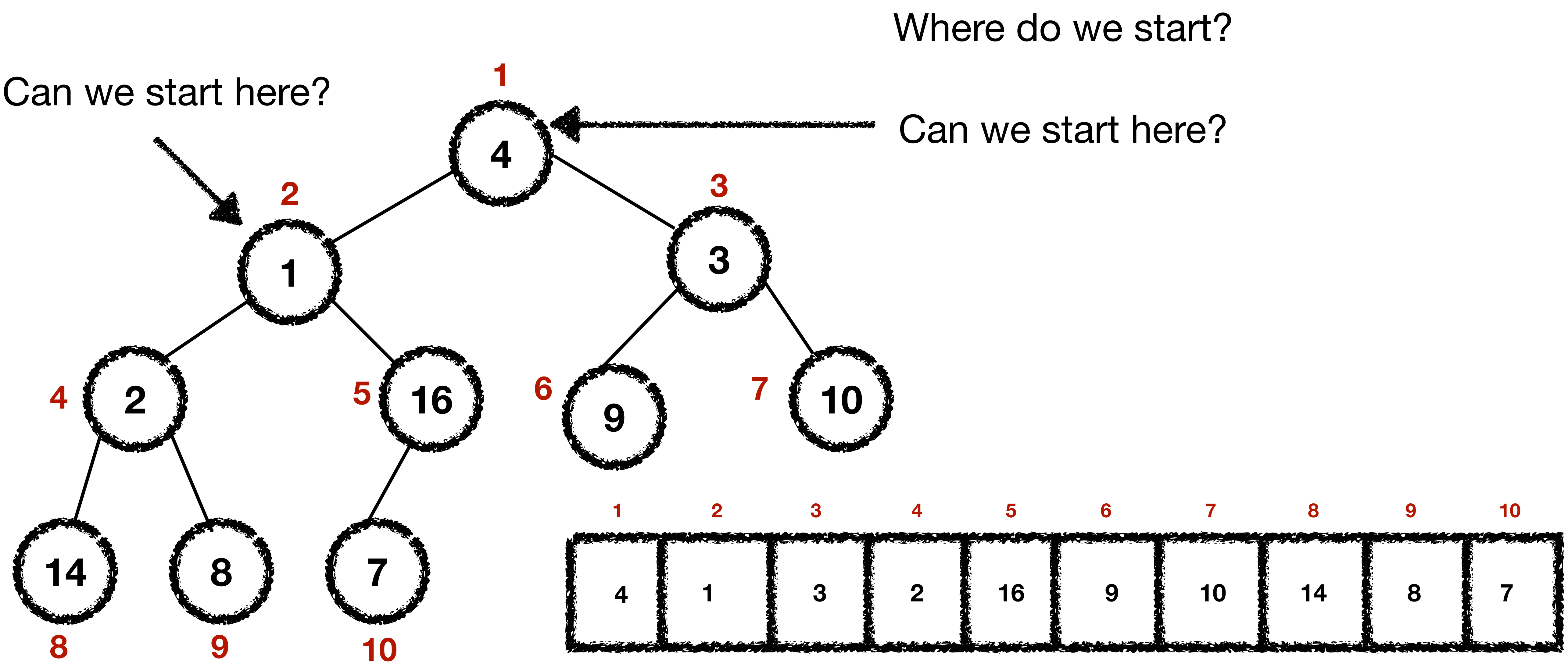
Postcondition: The tree routed at i is a heap.

```
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)
```

CLRS pp 165

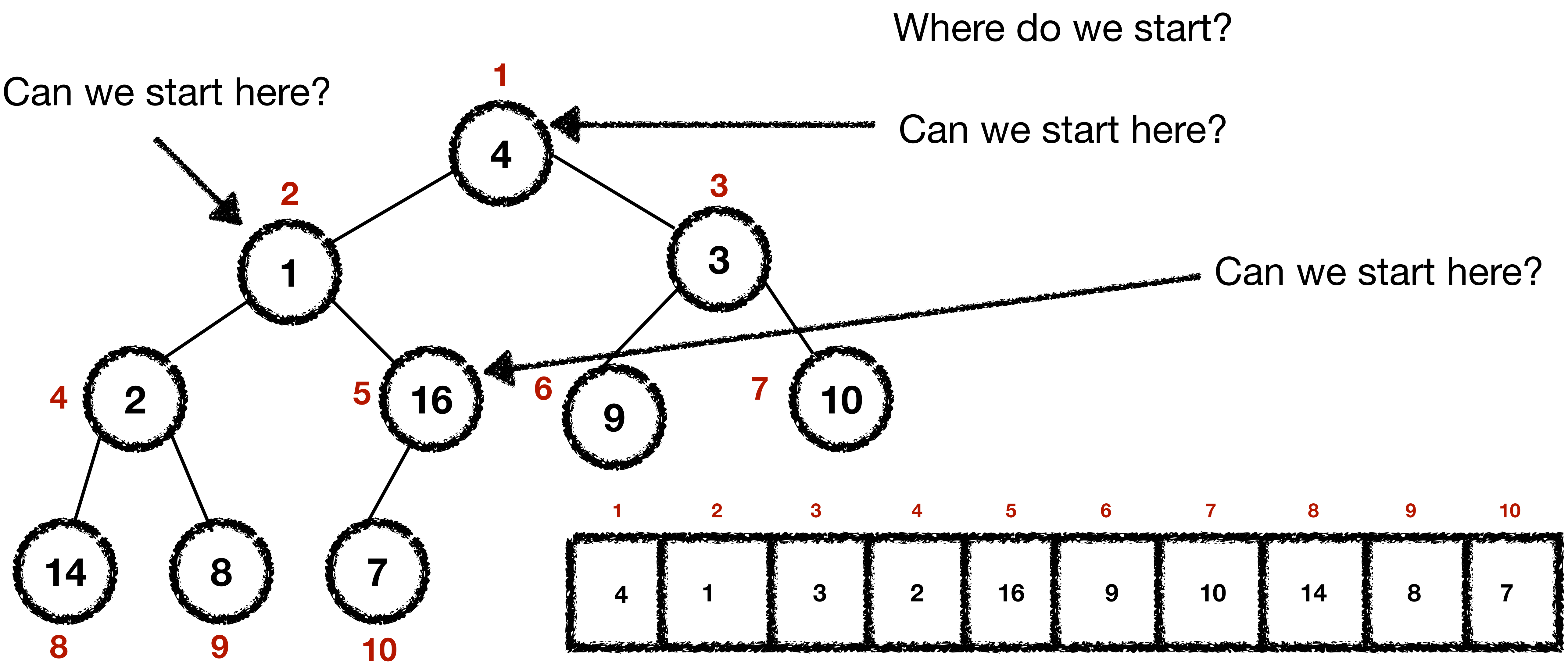
Build-Max-Heap

Idea: Apply Max-Heapify repeatedly until the tree becomes a heap.



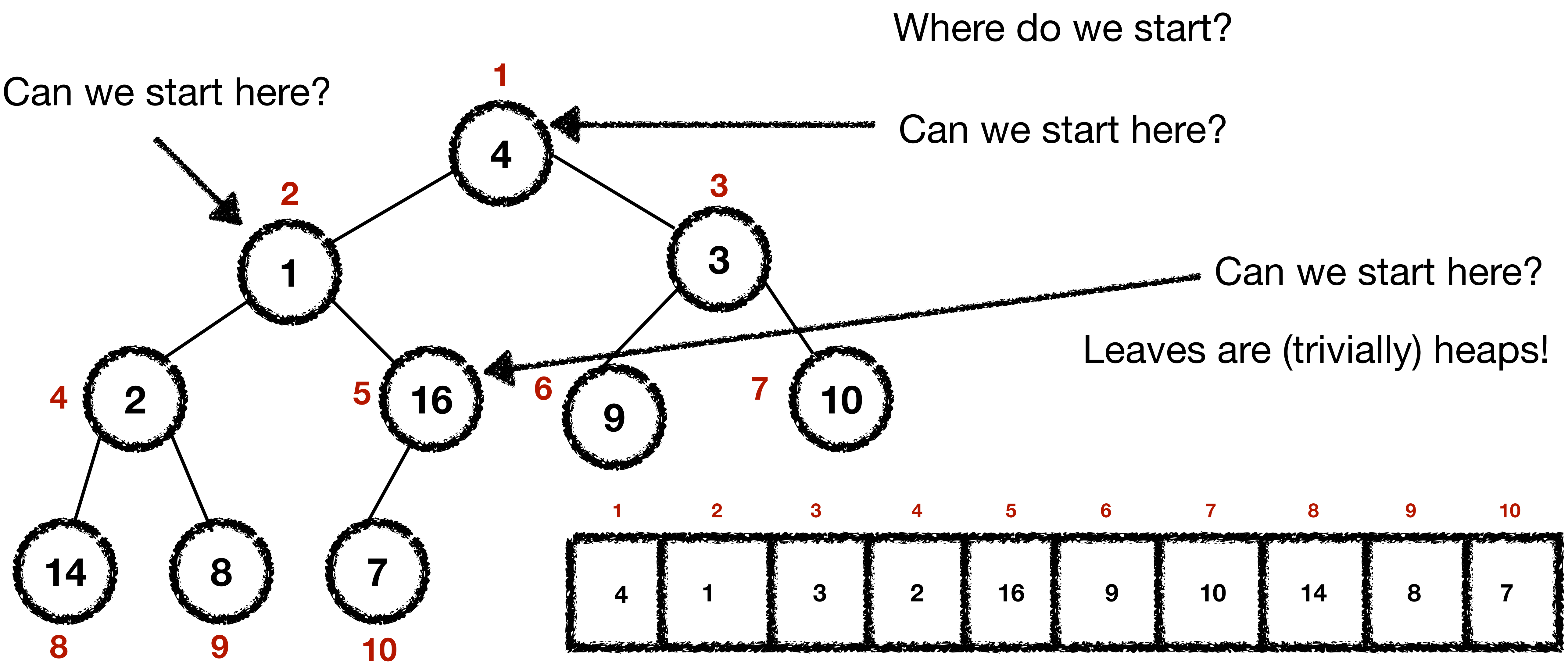
Build-Max-Heap

Idea: Apply Max-Heapify repeatedly until the tree becomes a heap.



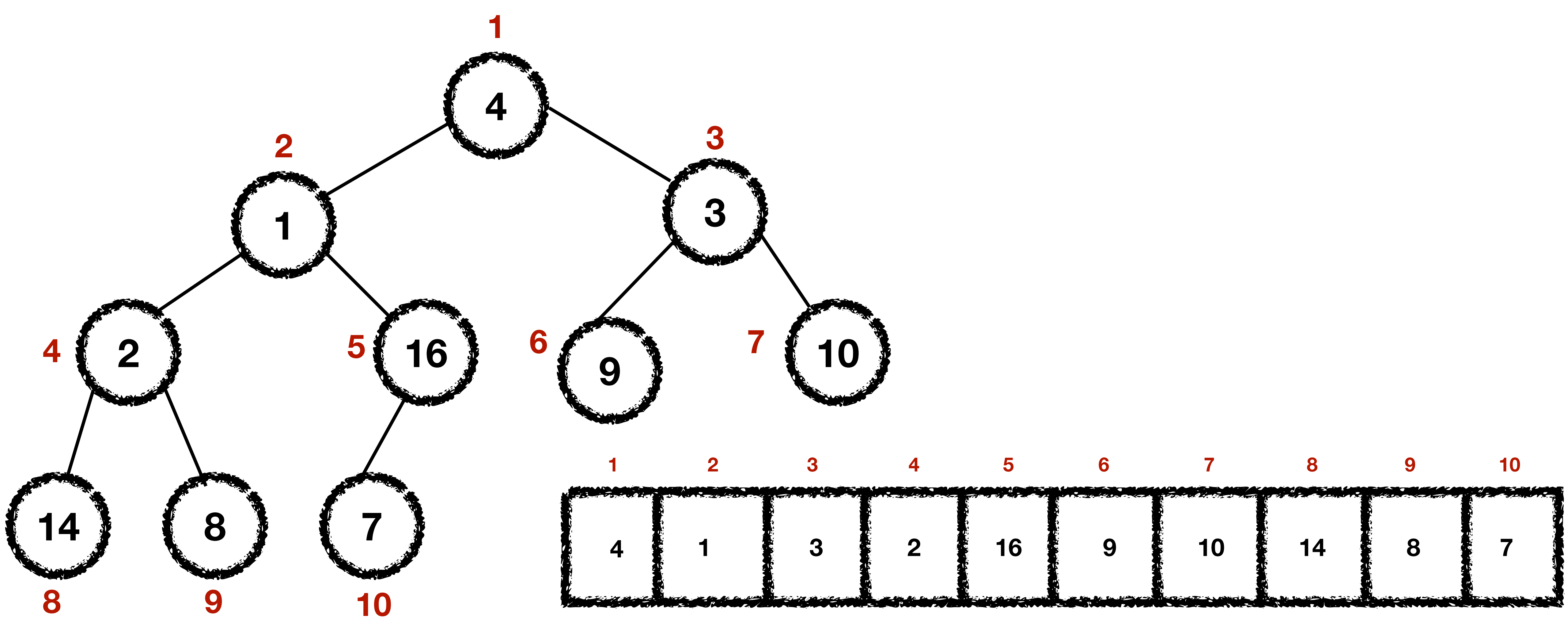
Build-Max-Heap

Idea: Apply Max-Heapify repeatedly until the tree becomes a heap.



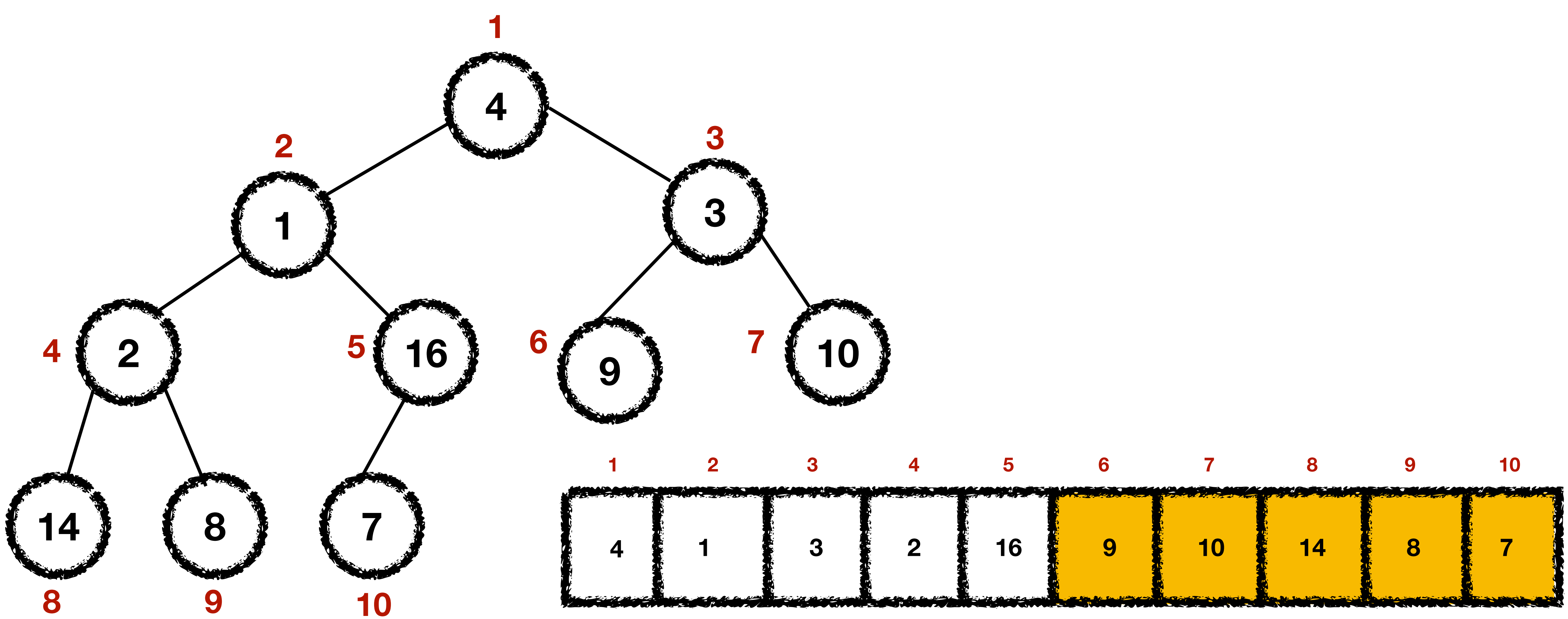
Build-Max-Heap

Observation: The elements of the subarray $A[\lfloor n/2 \rfloor + 1 : n]$ are leaves.



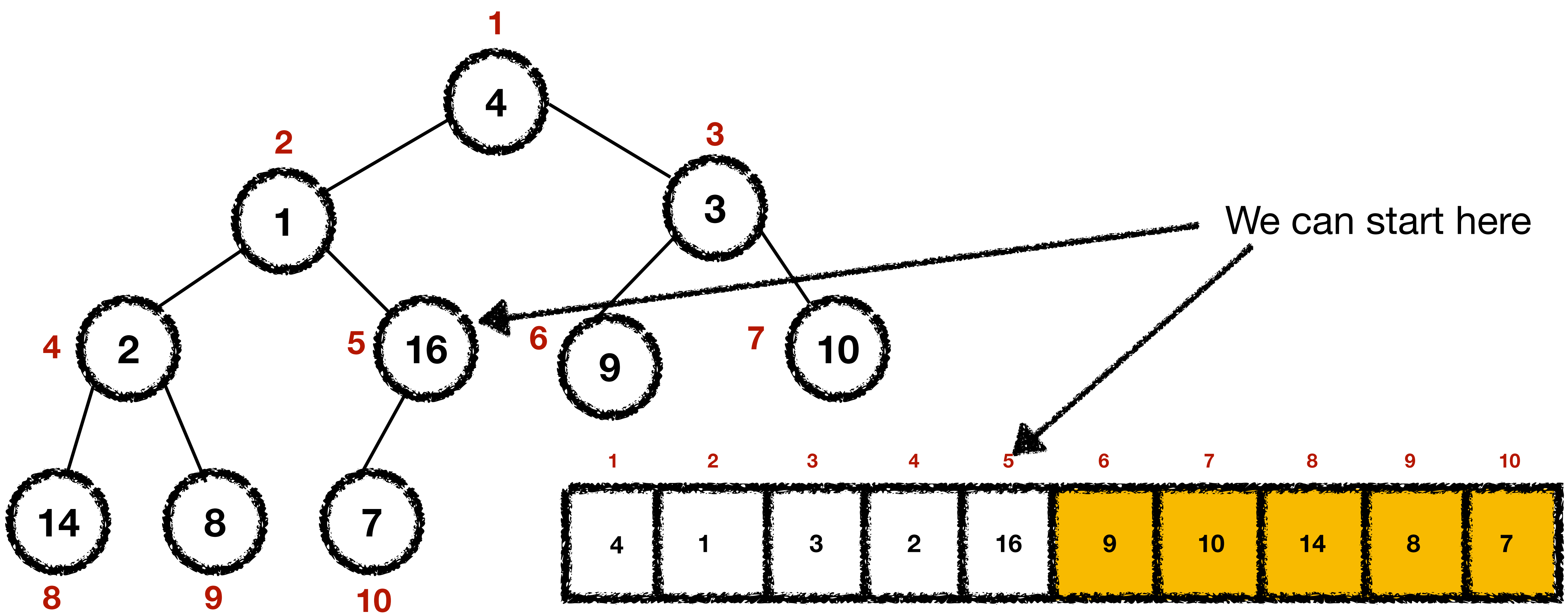
Build-Max-Heap

Observation: The elements of the subarray $A[\lfloor n/2 \rfloor + 1 : n]$ are leaves.

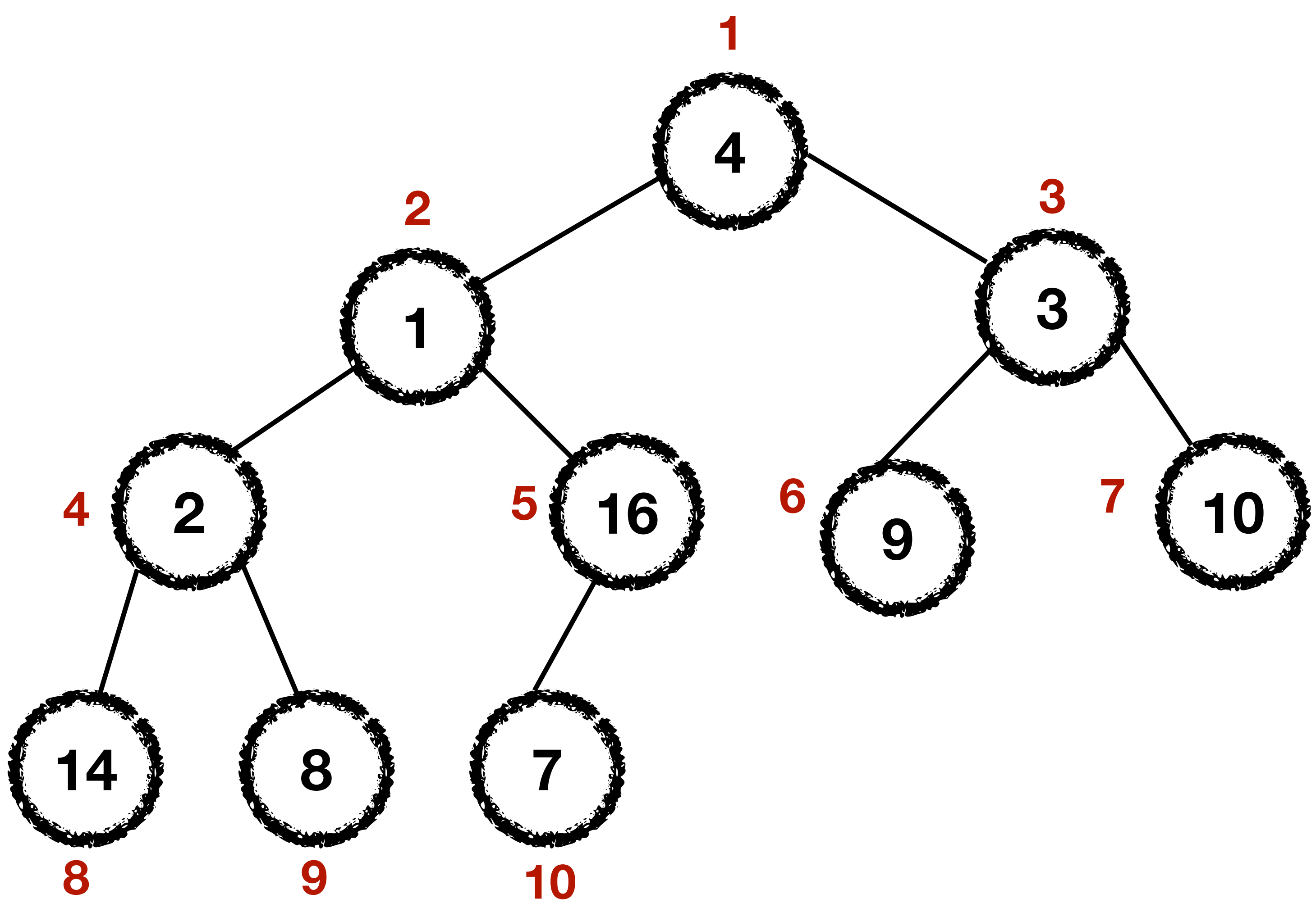


Build-Max-Heap

Observation: The elements of the subarray $A[\lfloor n/2 \rfloor + 1 : n]$ are leaves.



Build-Max-Heap



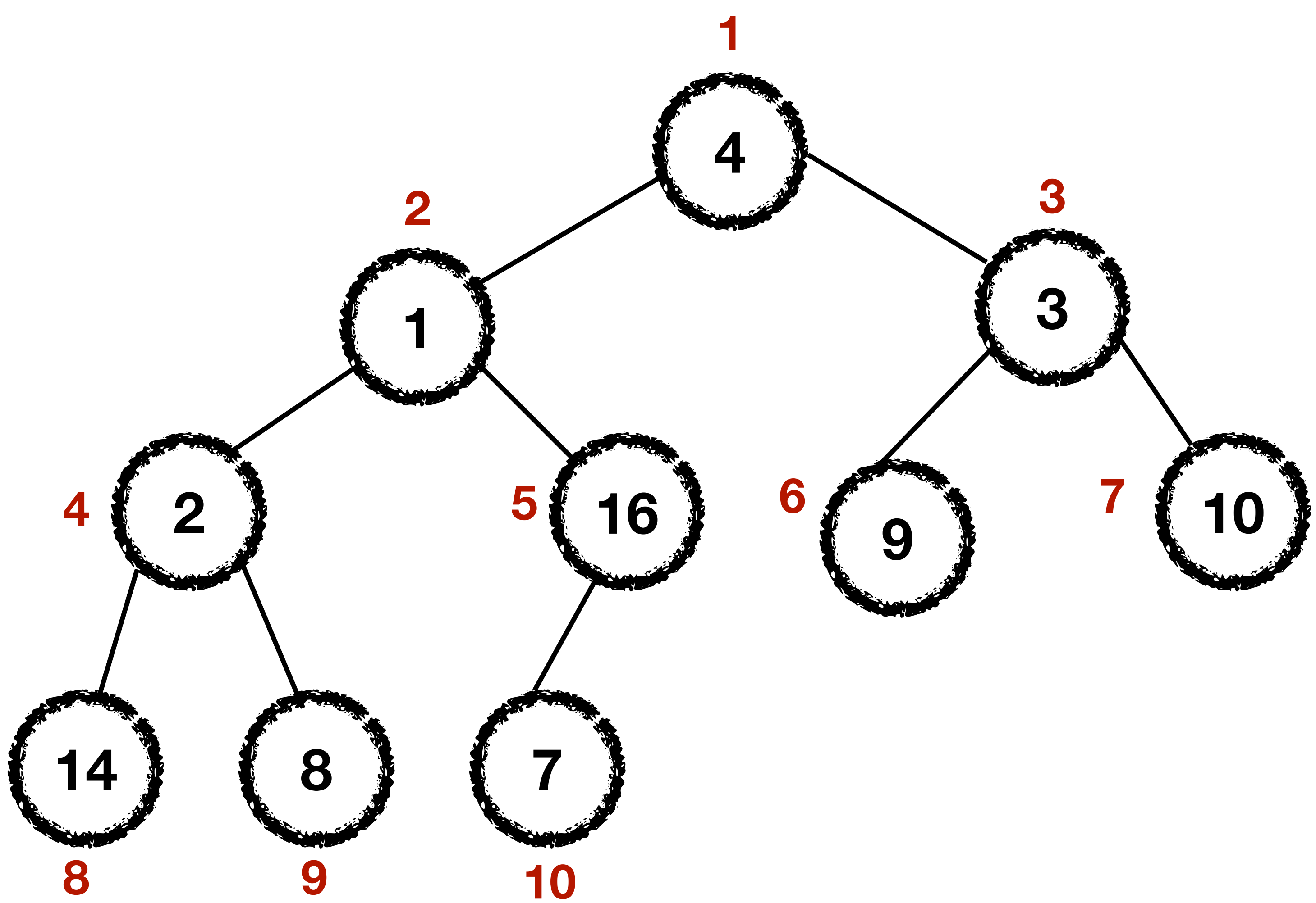

```
BUILD-MAX-HEAP(A, n)
  A.heap-size = n
  for i = ⌊n/2⌋ downto 1
      MAX-HEAPIFY(A, i)
```

CLRS pp 167

```
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)
```

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

Build-Max-Heap



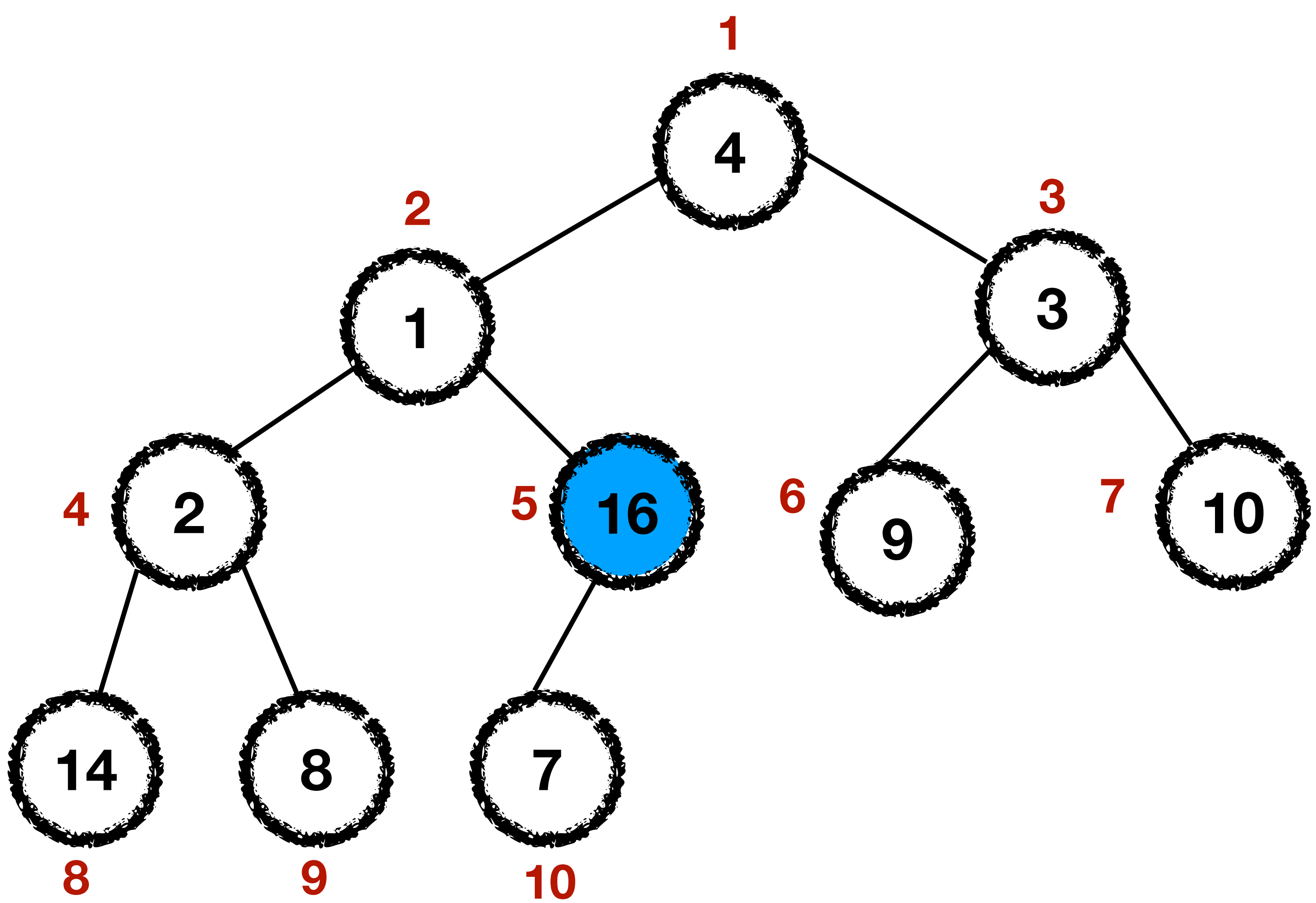

```
BUILD-MAX-HEAP(A, n)
    A.heap-size = n
    for i = ⌊n/2⌋ downto 1
        MAX-HEAPIFY(A, i)
```

CLRS pp 167

```
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)
```

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

Build-Max-Heap



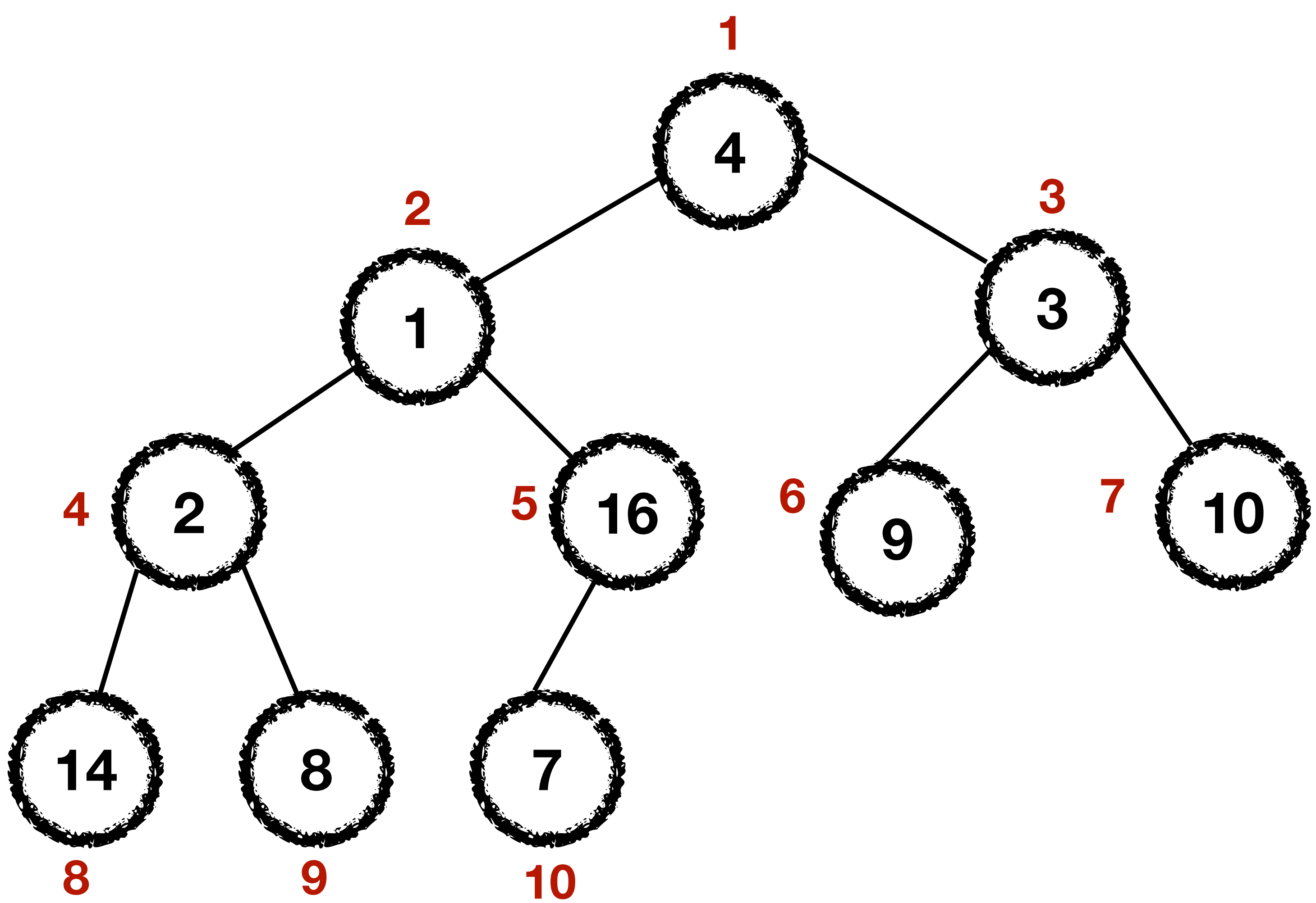

```
BUILD-MAX-HEAP(A, n)
  A.heap-size = n
  for i = ⌊n/2⌋ downto 1
      MAX-HEAPIFY(A, i)
```

CLRS pp 167

```
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)
```

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

Build-Max-Heap



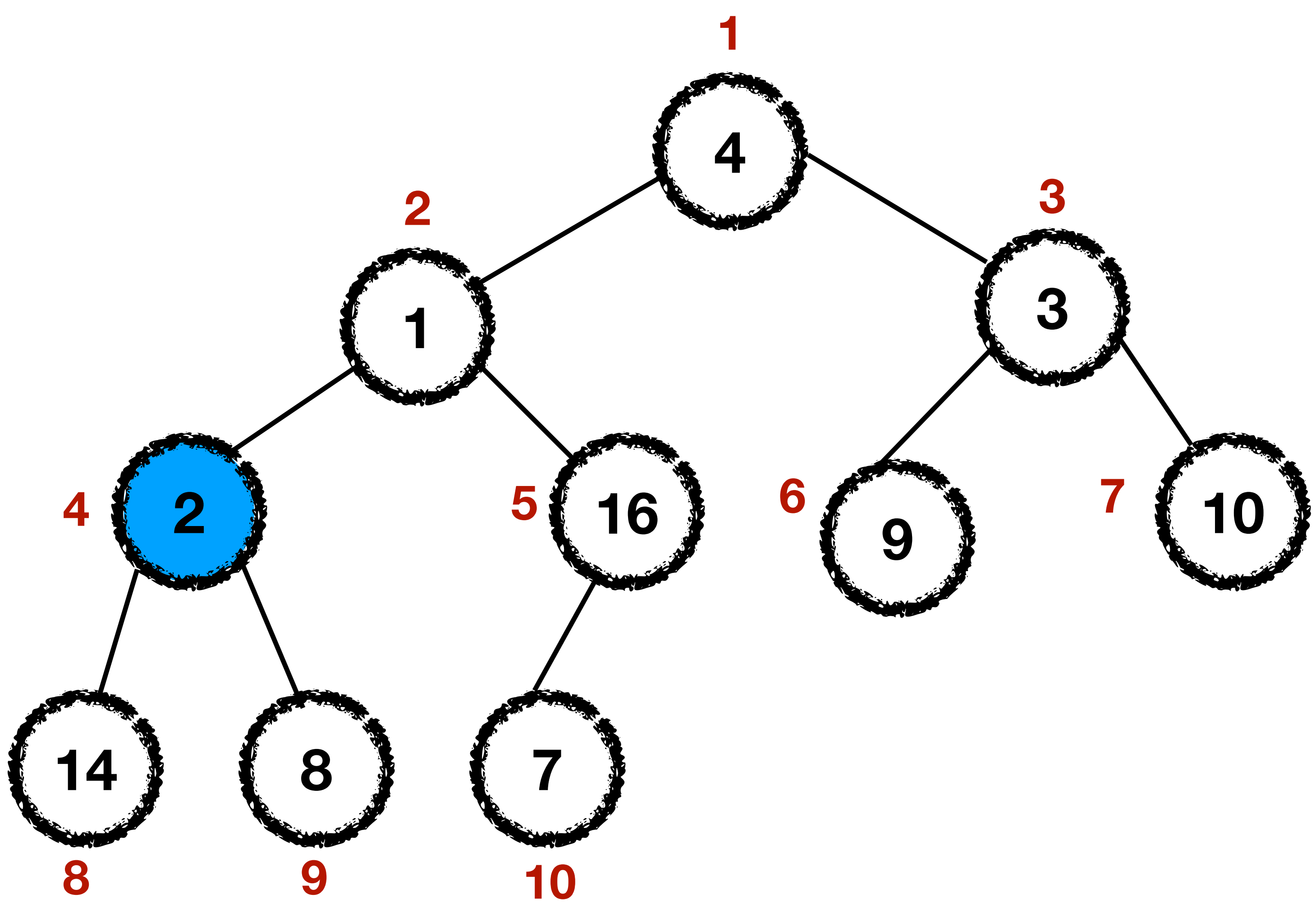

```
BUILD-MAX-HEAP(A, n)
  A.heap-size = n
  for i = ⌊n/2⌋ downto 1
      MAX-HEAPIFY(A, i)
```

CLRS pp 167

```
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)
```

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

Build-Max-Heap



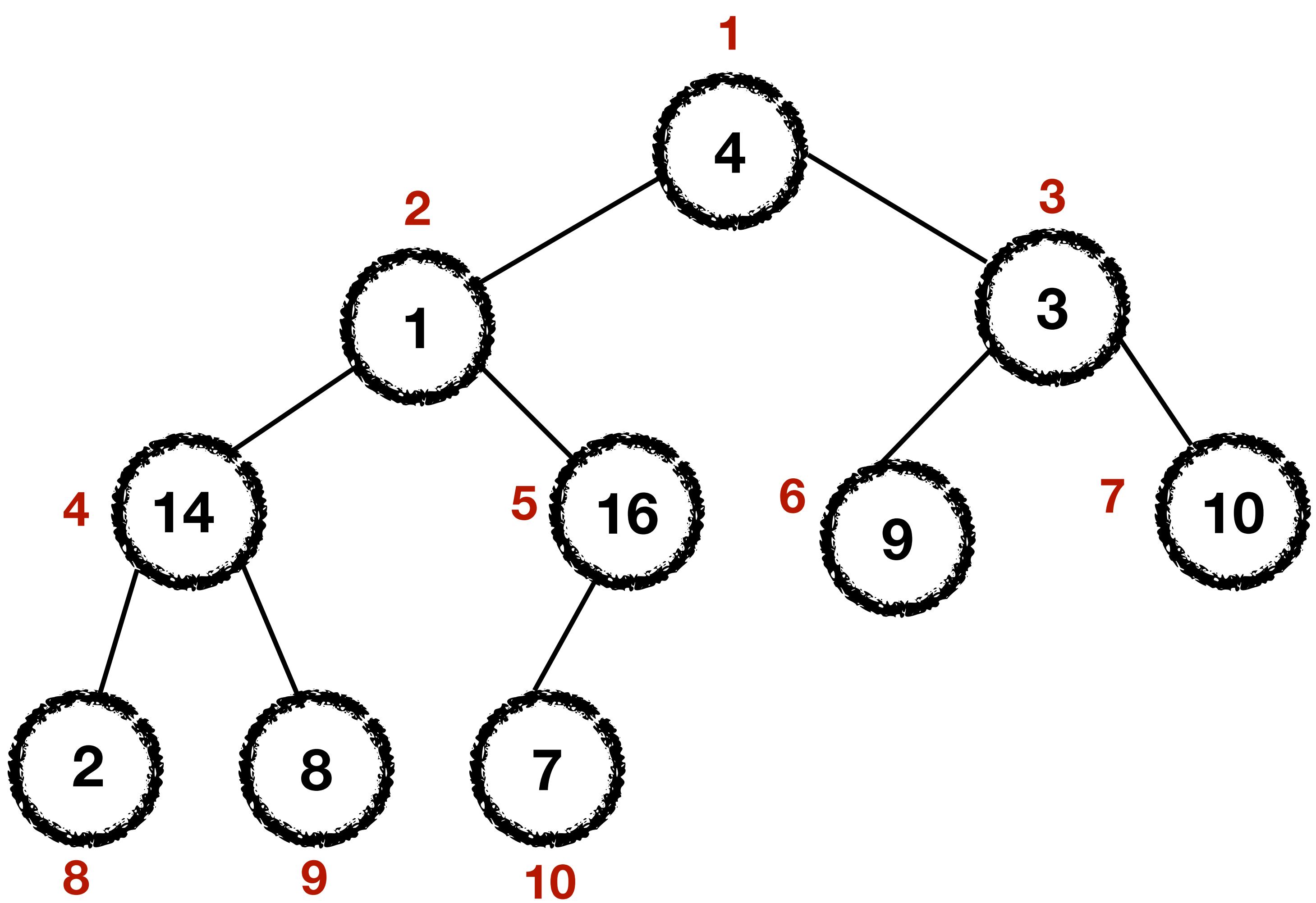

```
BUILD-MAX-HEAP(A, n)
  A.heap-size = n
  for i = ⌊n/2⌋ downto 1
      MAX-HEAPIFY(A, i)
```

CLRS pp 167

```
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)
```

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

Build-Max-Heap



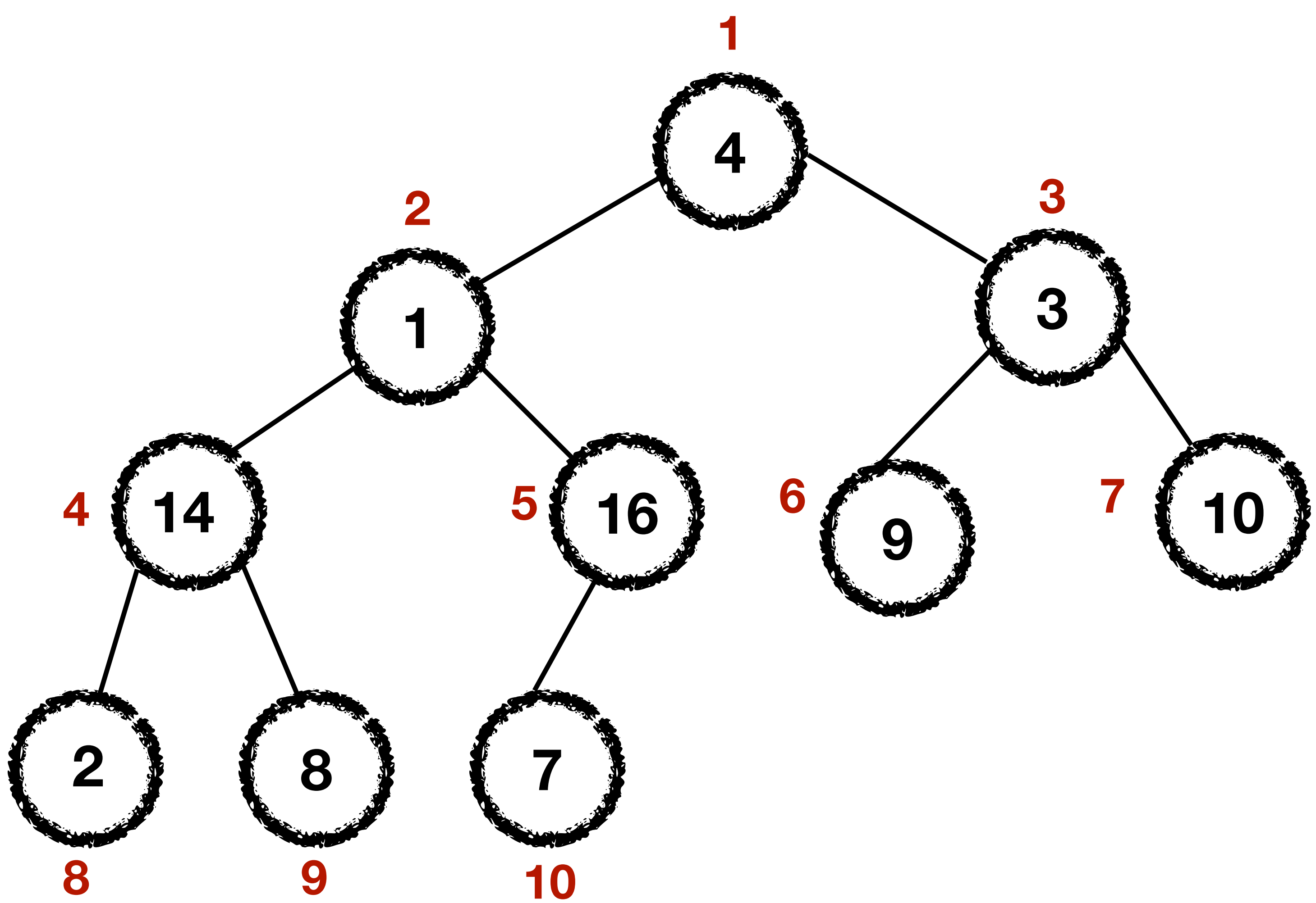

```
BUILD-MAX-HEAP(A, n)
  A.heap-size = n
  for i = ⌊n/2⌋ downto 1
      MAX-HEAPIFY(A, i)
```

CLRS pp 167

```
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)
```

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

Build-Max-Heap



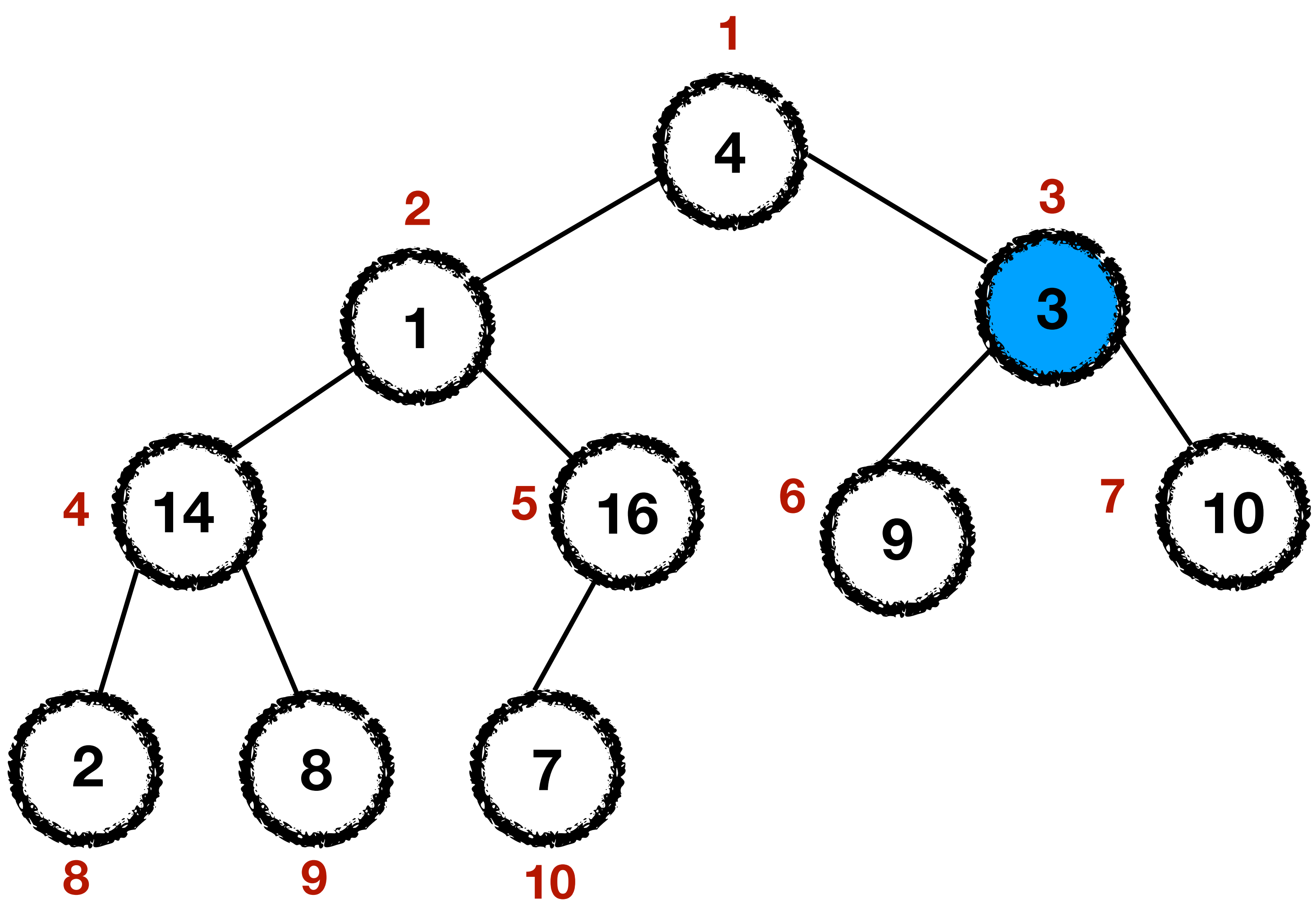

```
BUILD-MAX-HEAP(A, n)
  A.heap-size = n
  for i = ⌊n/2⌋ downto 1
      MAX-HEAPIFY(A, i)
```

CLRS pp 167

```
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)
```

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

Build-Max-Heap



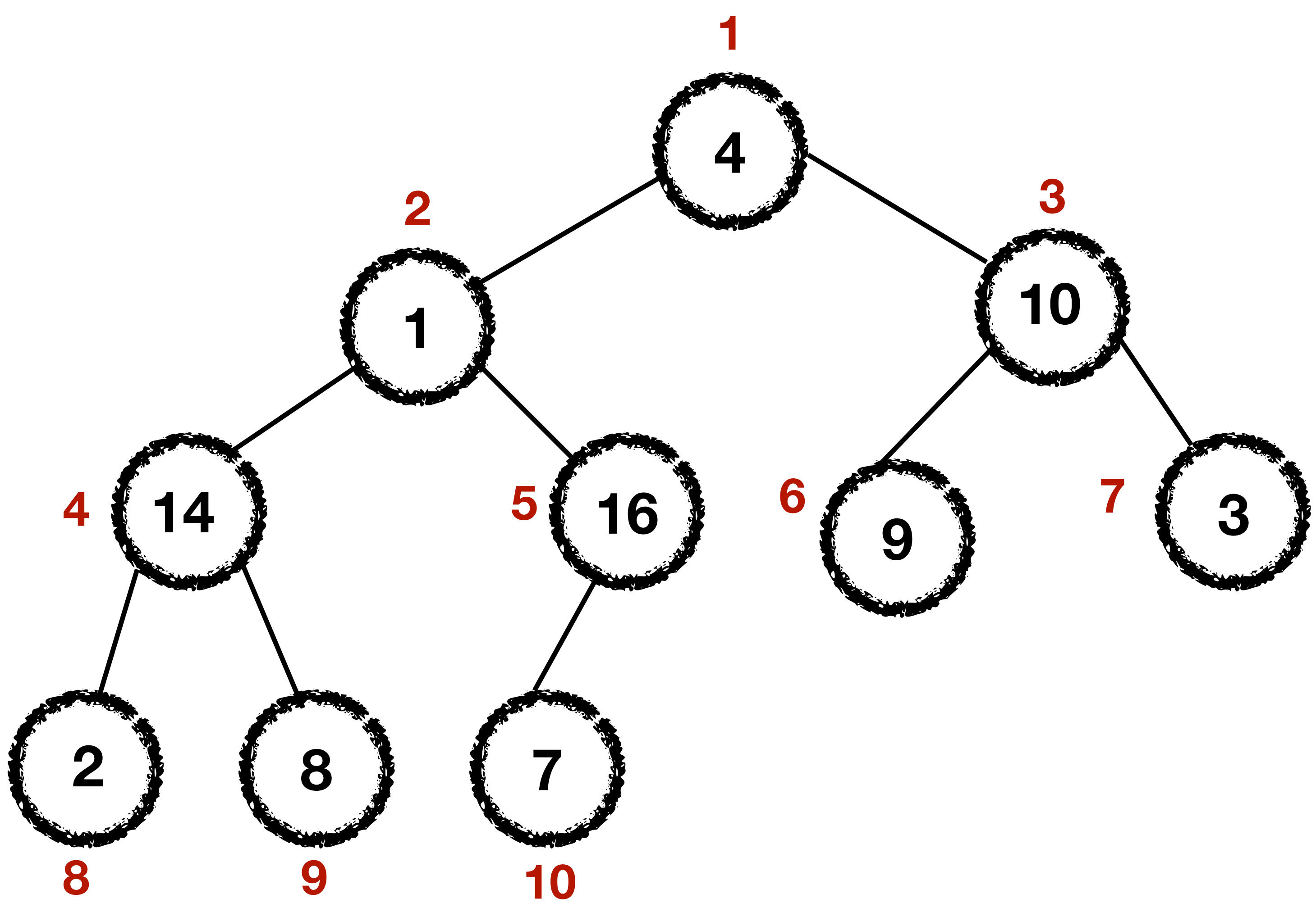

```
BUILD-MAX-HEAP(A, n)
    A.heap-size = n
    for i = ⌊n/2⌋ downto 1
        MAX-HEAPIFY(A, i)
```

CLRS pp 167

```
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)
```

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

Build-Max-Heap



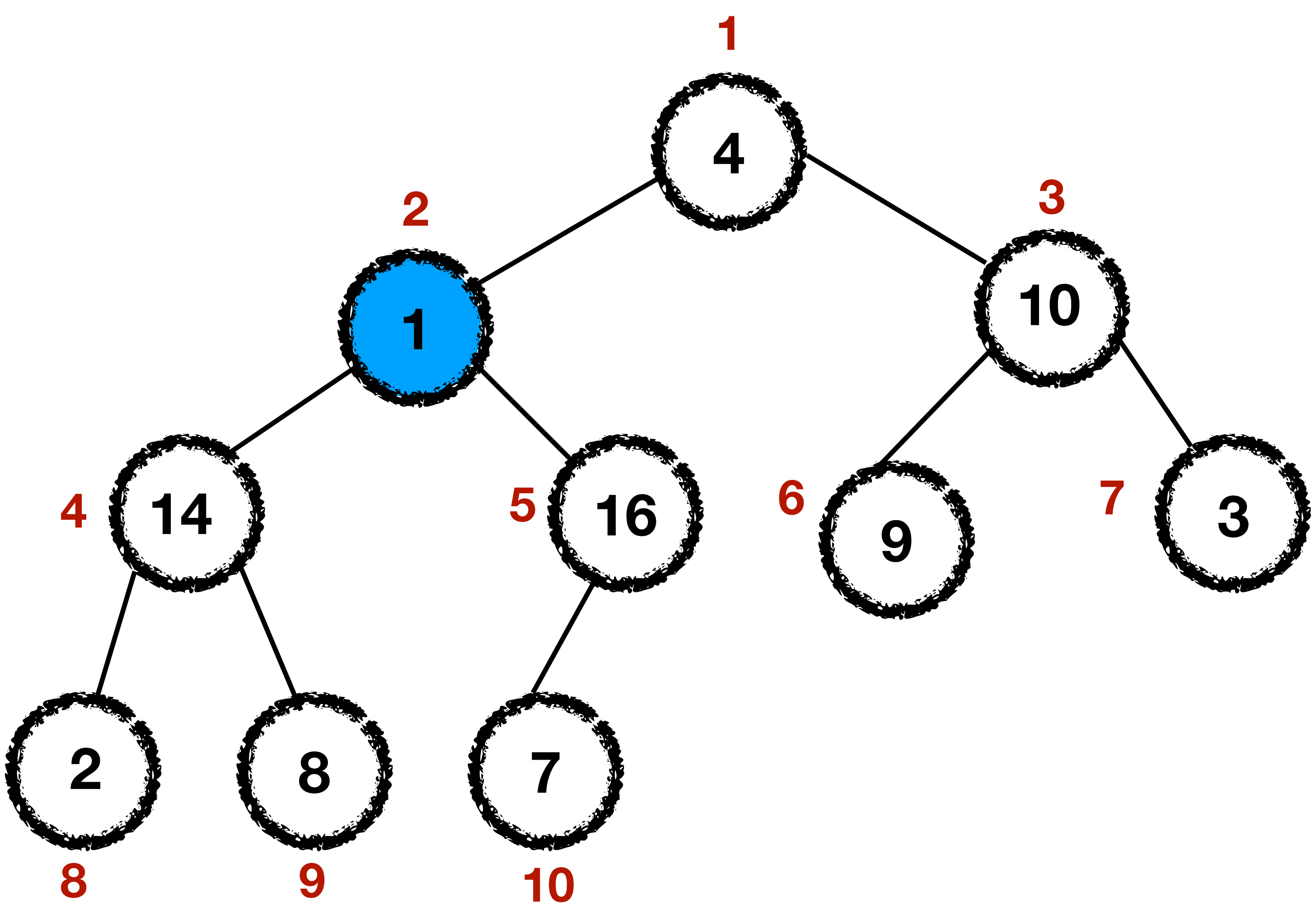

```
BUILD-MAX-HEAP(A, n)
  A.heap-size = n
  for i = ⌊n/2⌋ downto 1
      MAX-HEAPIFY(A, i)
```

CLRS pp 167

```
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)
```

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

Build-Max-Heap



BUILD-MAX-HEAP(A, n)

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

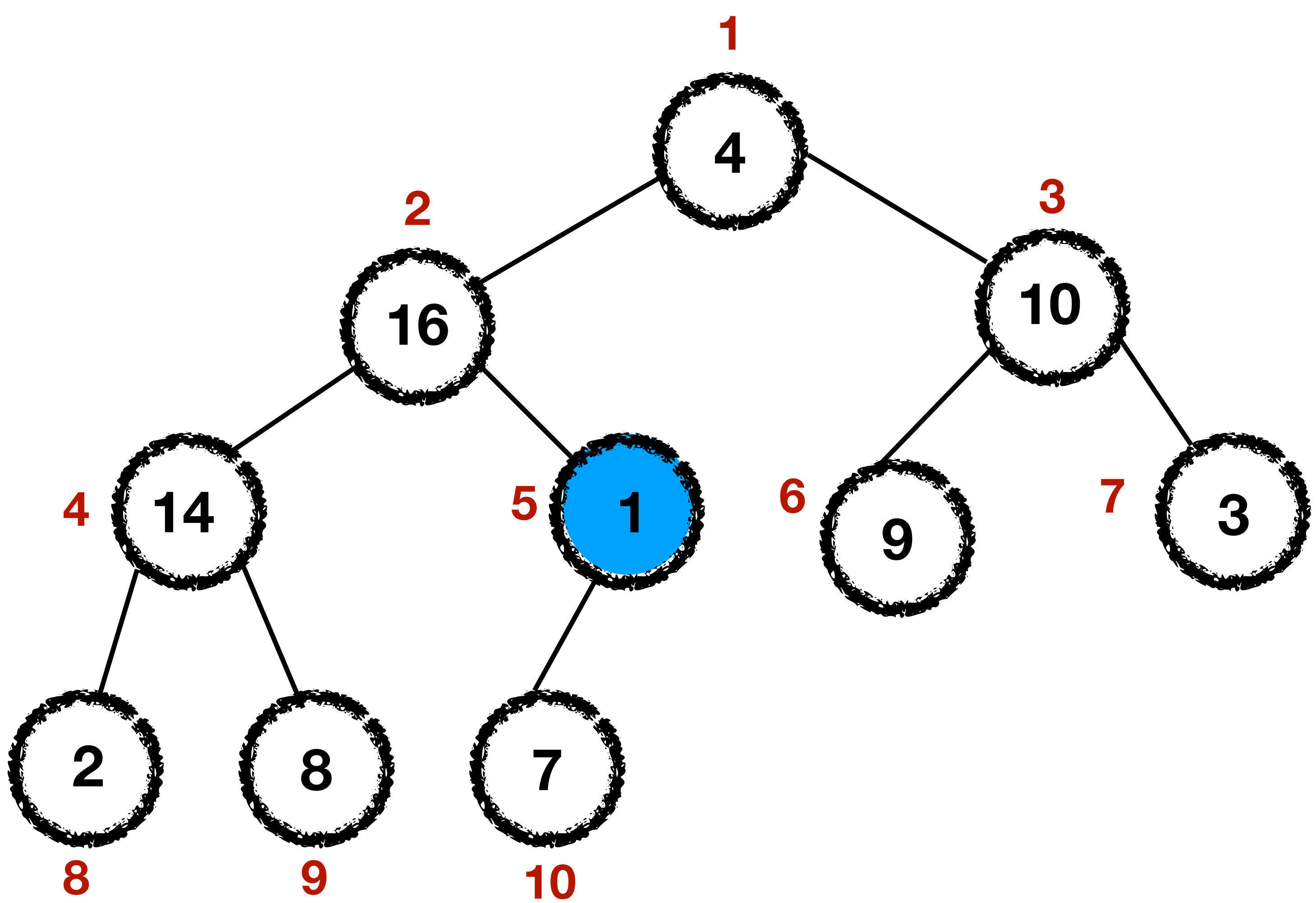
CLRS pp 167

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)
```

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

Build-Max-Heap



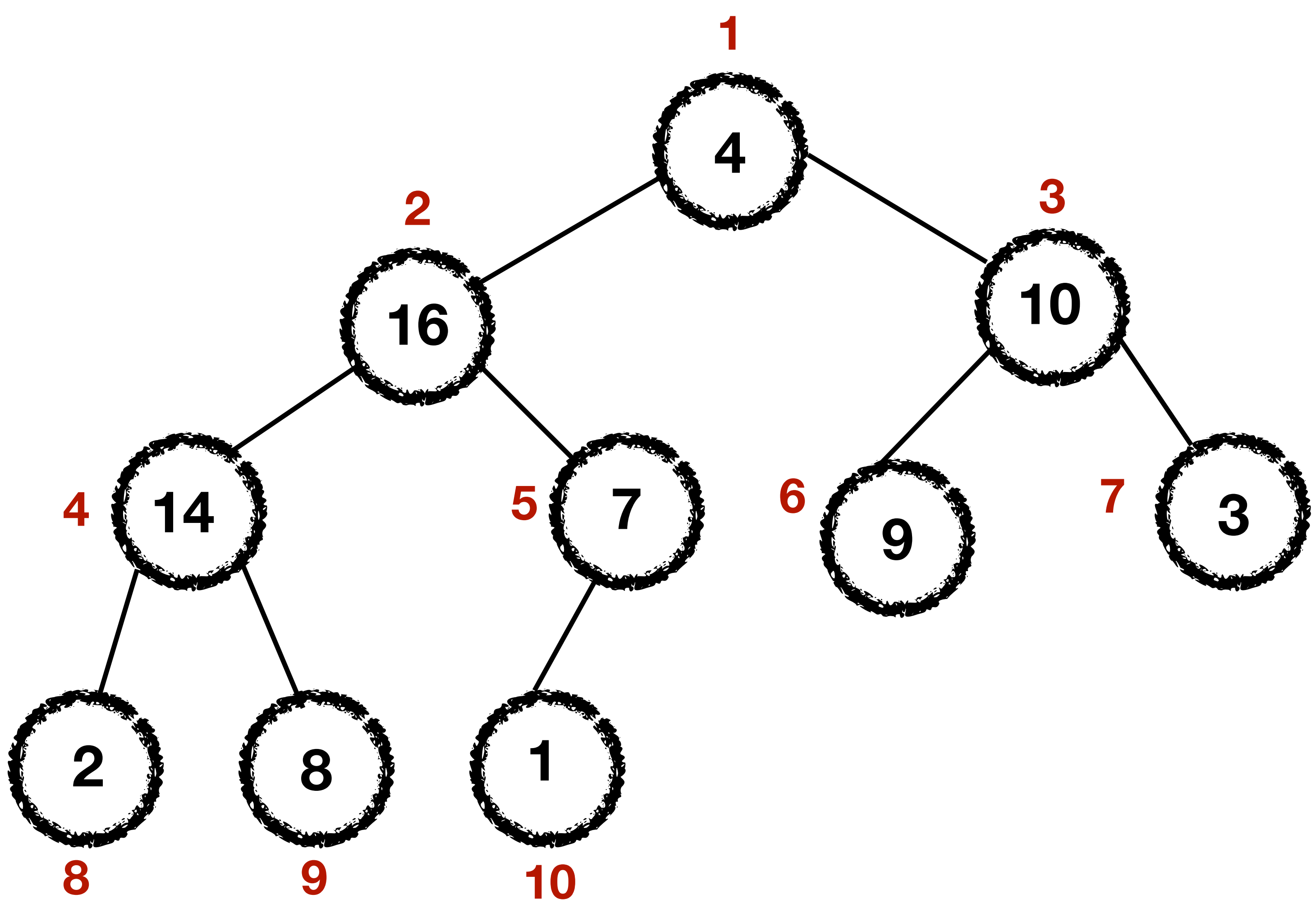

```
BUILD-MAX-HEAP(A, n)
  A.heap-size = n
  for i = ⌊n/2⌋ downto 1
      MAX-HEAPIFY(A, i)
```

CLRS pp 167

```
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)
```

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

Build-Max-Heap



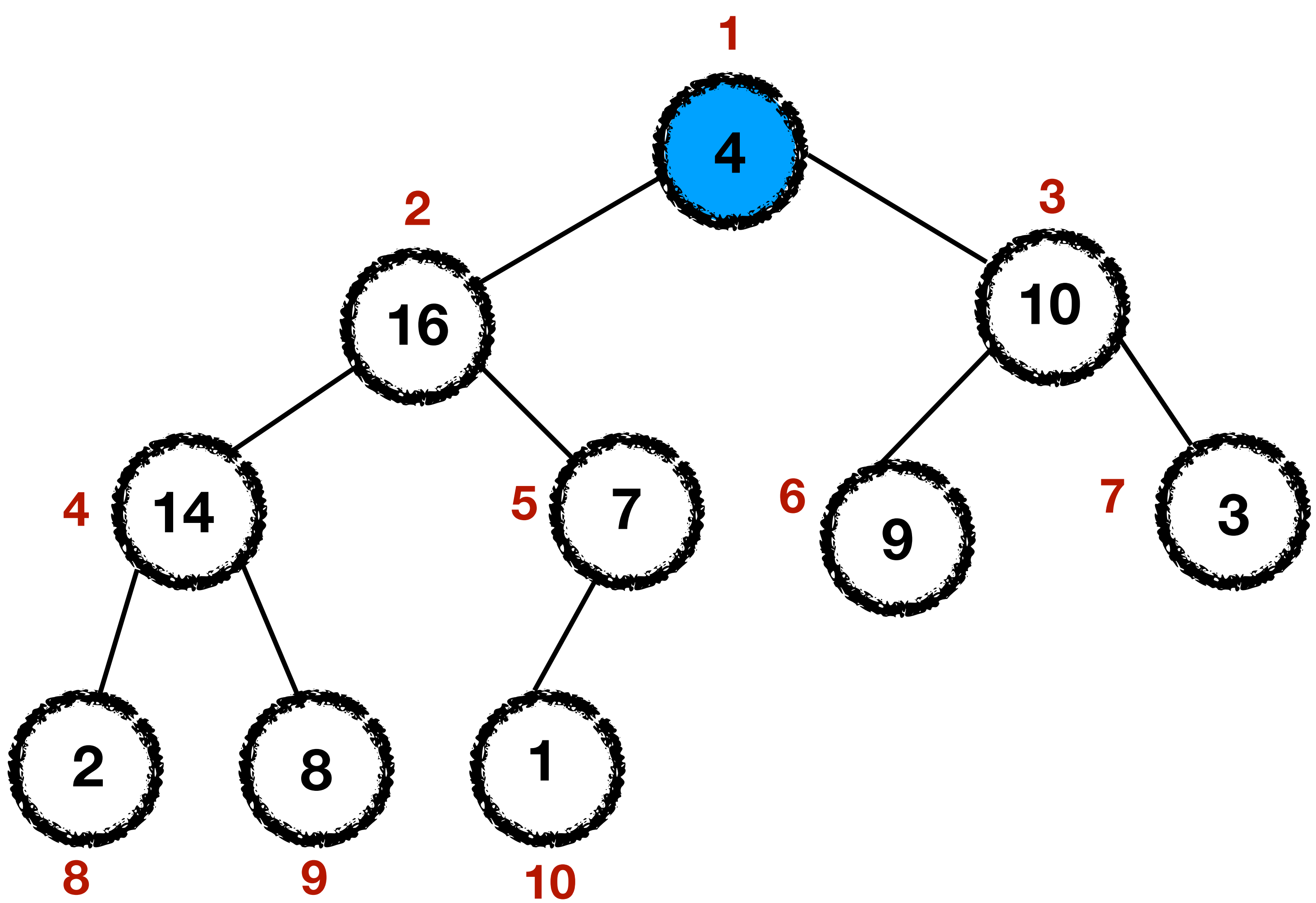

```
BUILD-MAX-HEAP(A, n)
    A.heap-size = n
    for i = ⌊n/2⌋ downto 1
        MAX-HEAPIFY(A, i)
```

CLRS pp 167

```
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)
```

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

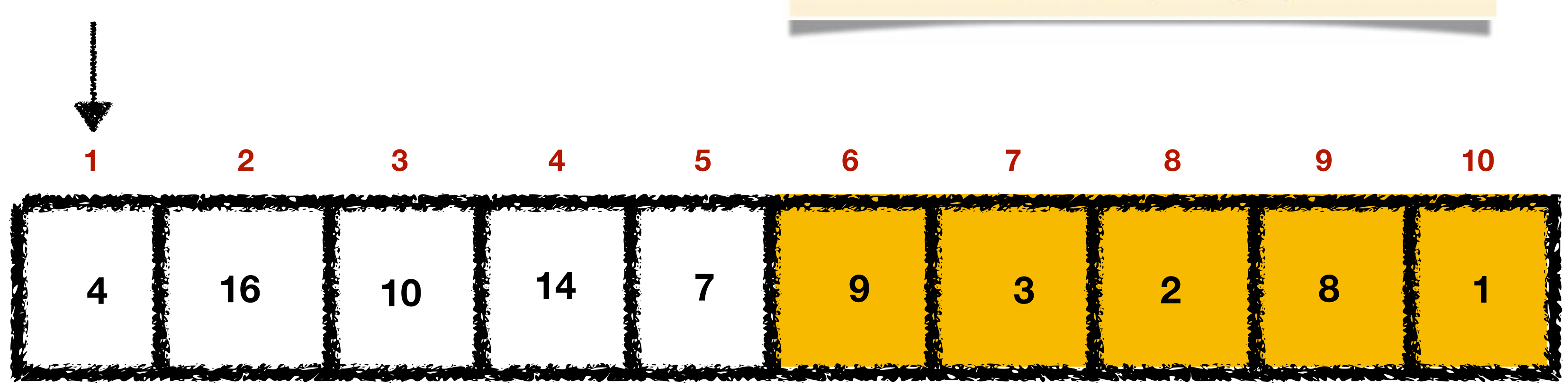
Build-Max-Heap



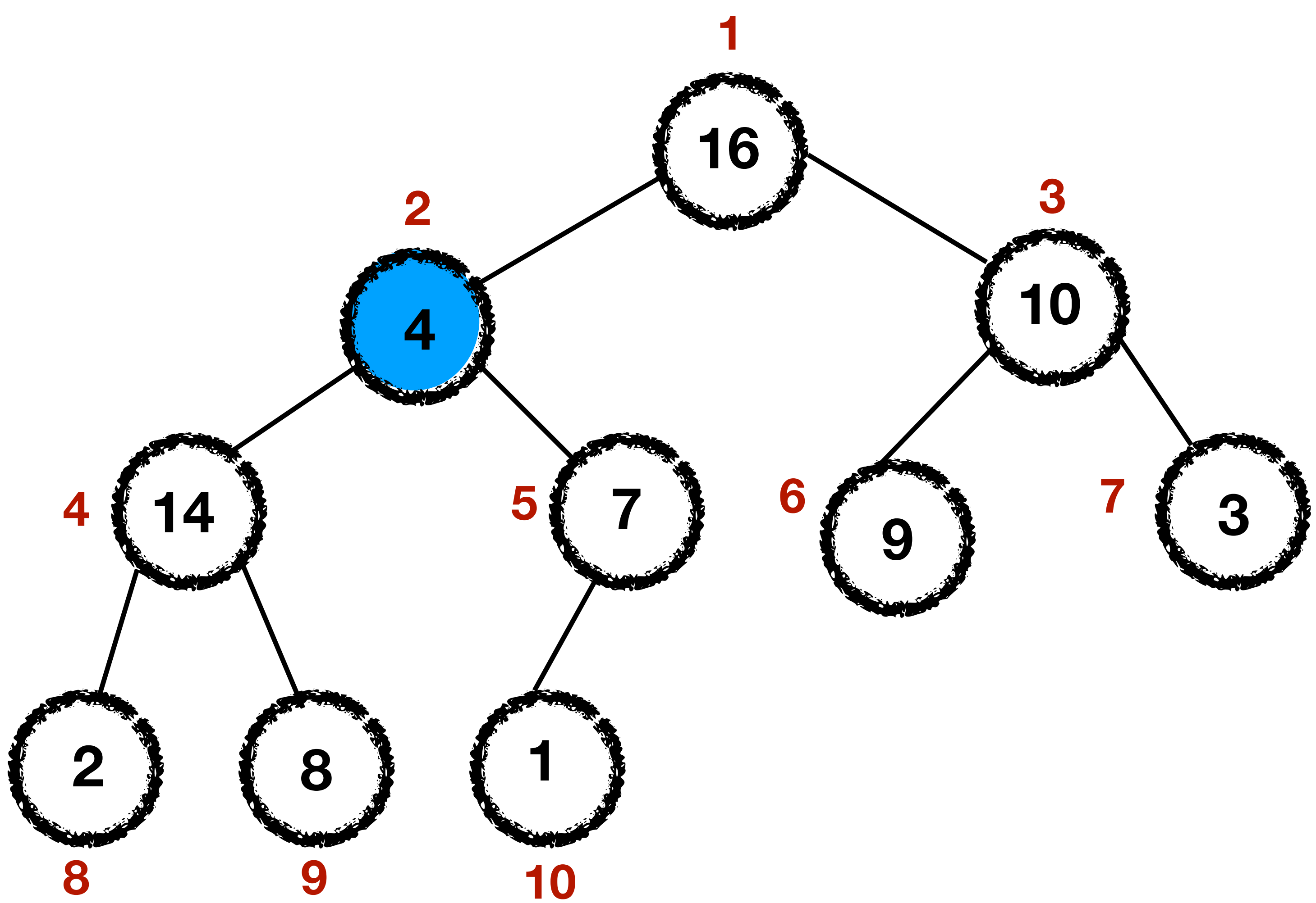

```
BUILD-MAX-HEAP(A, n)
  A.heap-size = n
  for i = ⌊n/2⌋ downto 1
      MAX-HEAPIFY(A, i)
```

CLRS pp 167

```
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)
```



Build-Max-Heap



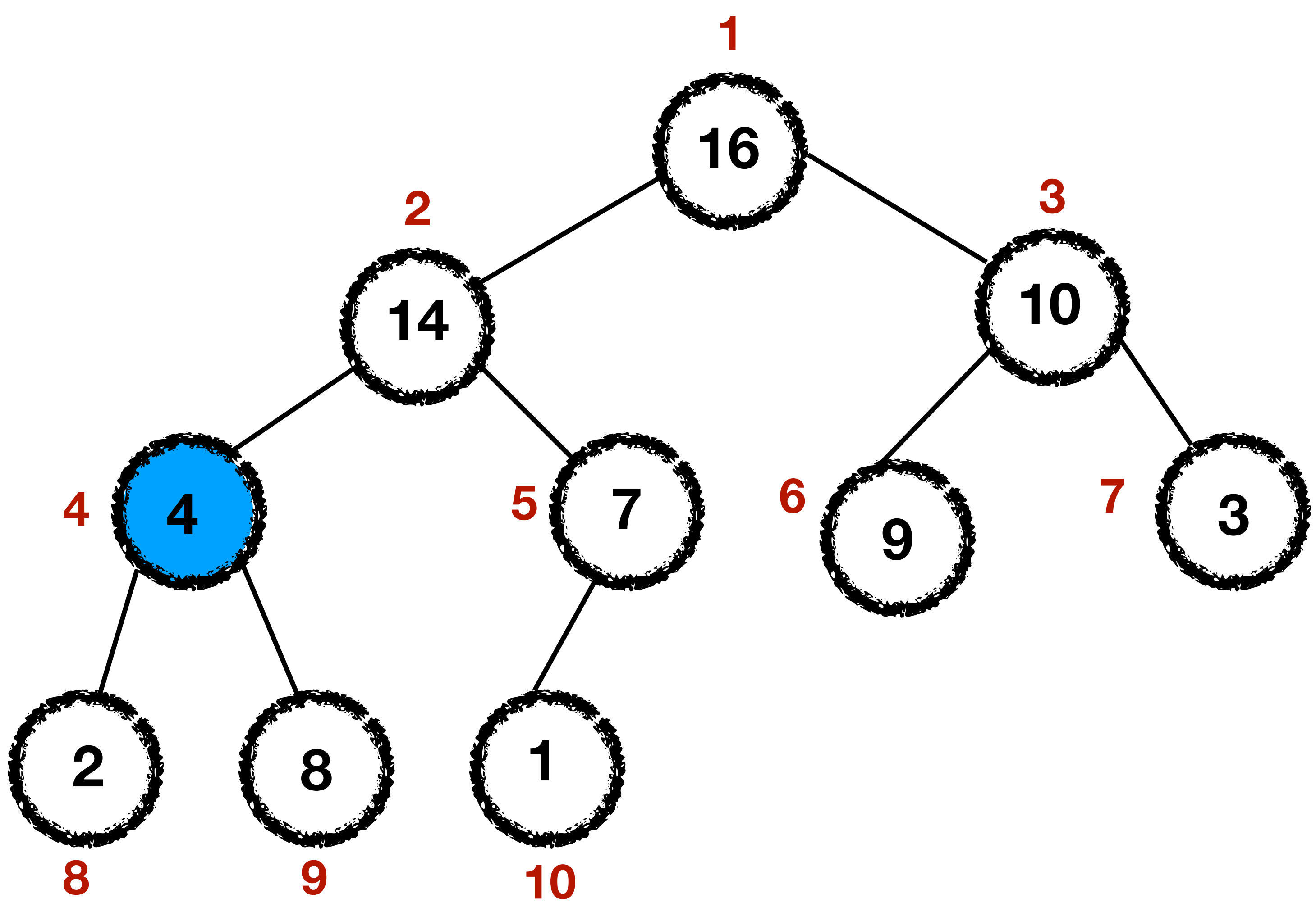

```
BUILD-MAX-HEAP(A, n)
   A.heap-size = n
   for i = ⌊n/2⌋ downto 1
       MAX-HEAPIFY(A, i)
```

CLRS pp 167

```
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)
```

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

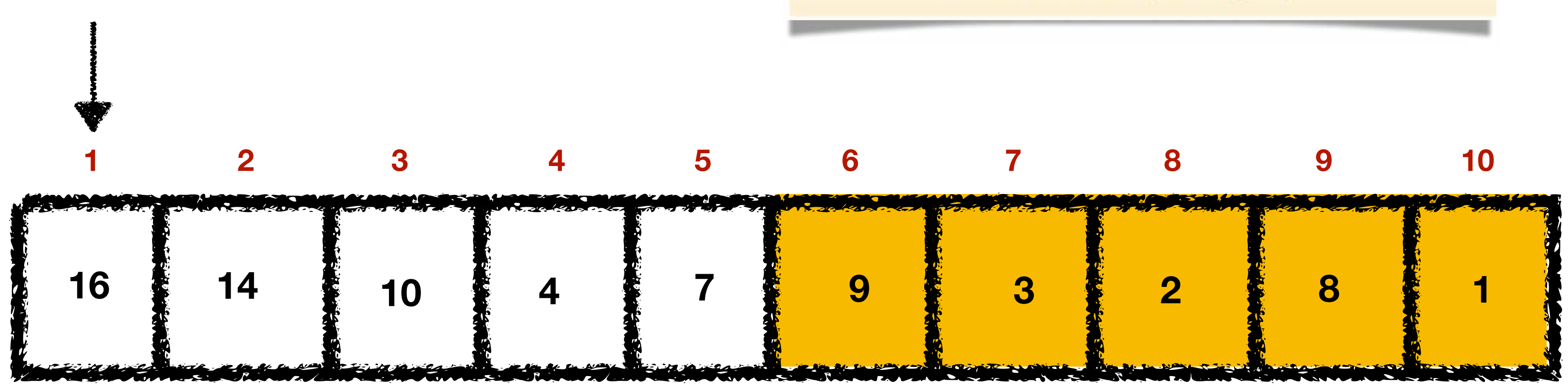
Build-Max-Heap



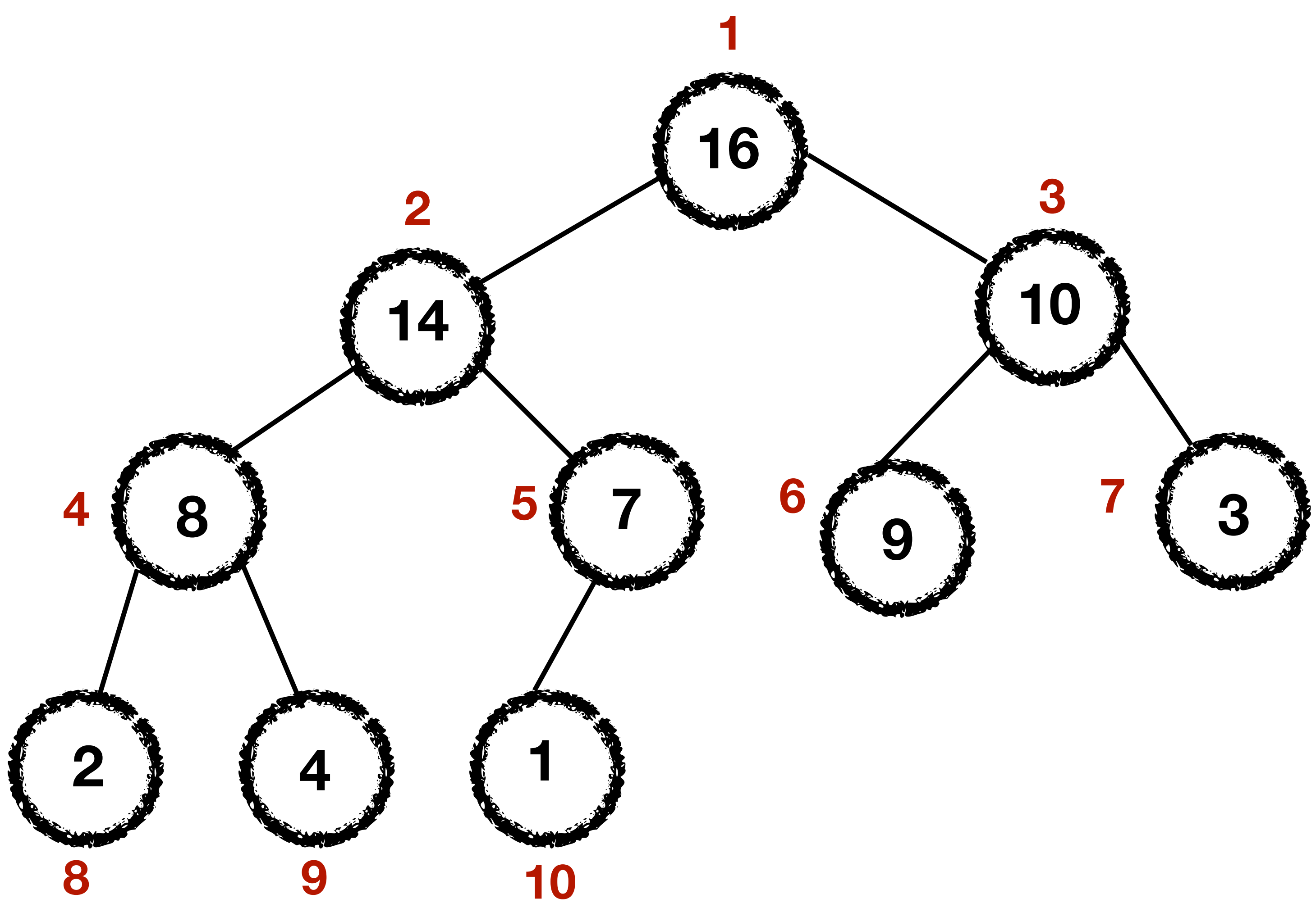

```
BUILD-MAX-HEAP(A, n)
  A.heap-size = n
  for i = ⌊n/2⌋ downto 1
      MAX-HEAPIFY(A, i)
```

CLRS pp 167

```
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)
```



Build-Max-Heap



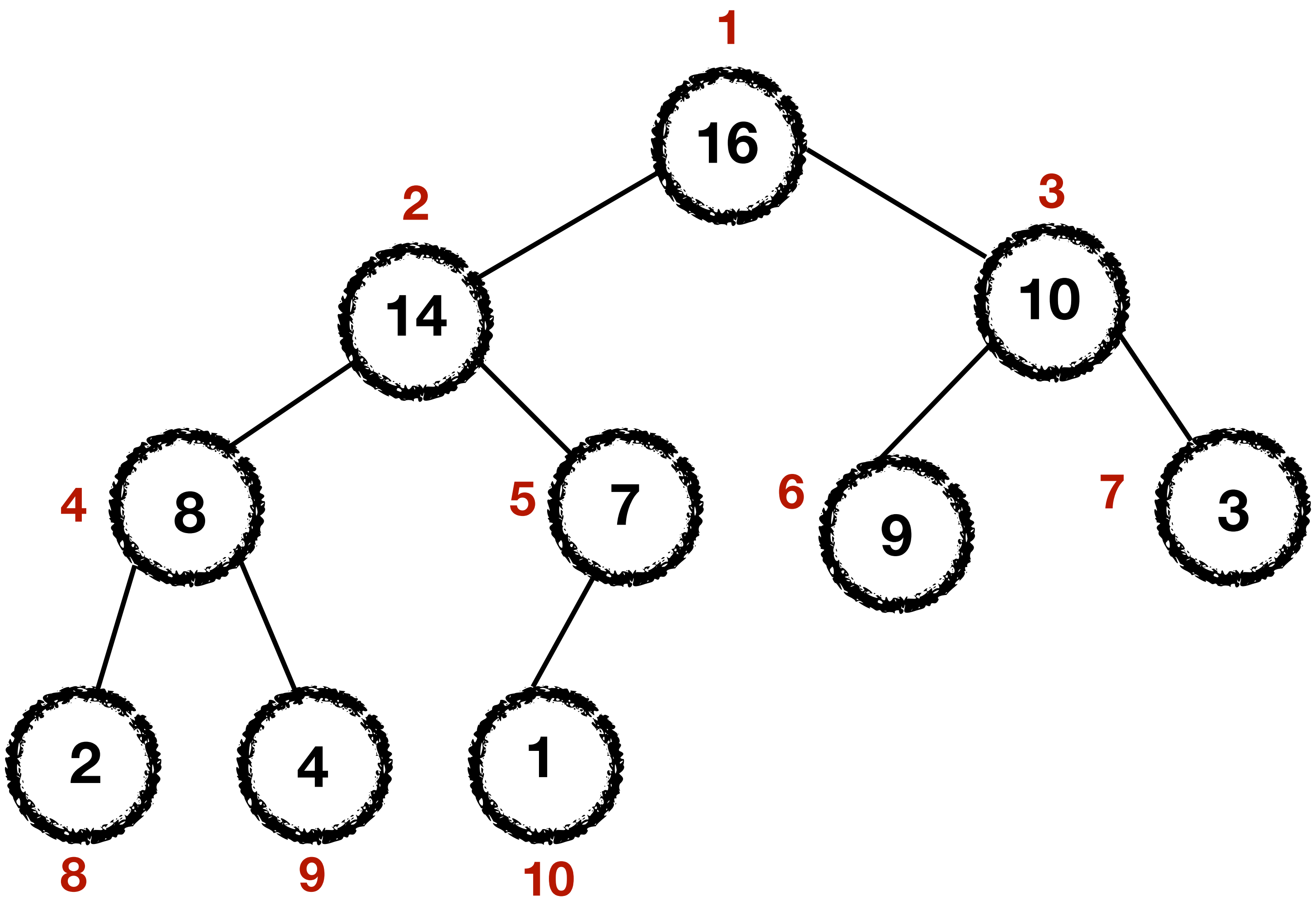

```
BUILD-MAX-HEAP(A, n)
  A.heap-size = n
  for i = ⌊n/2⌋ downto 1
      MAX-HEAPIFY(A, i)
```

CLRS pp 167

```
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)
```

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

Build-Max-Heap




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

CLRS pp 167

```
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)
```

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

Build-Max-Heap Correctness

Build-Max-Heap Correctness

We will argue via a loop invariant:

Build-Max-Heap Correctness

We will argue via a loop invariant:

At the start of each iteration of the for loop (lines 2-3), each node $i+1,\ i+2,\ \ldots,\ n$ *is the root of a max-heap.*

Build-Max-Heap Correctness

We will argue via a loop invariant:

At the start of each iteration of the for loop (lines 2-3), each node $i + 1,\ i + 2,\ \ldots,\ n$ is the root of a max-heap.

Initialisation: Prior to the first iteration of the loop $i = \lfloor n/2 \rfloor$. In that case the nodes $\lfloor n/2 \rfloor, \lfloor n/2 \rfloor + 1, \ldots, n$ are leaves, and hence trivially max-heaps.

Build-Max-Heap Correctness

We will argue via a loop invariant:

At the start of each iteration of the for loop (lines 2-3), each node $i+1,\ i+2,\ \ldots,\ n$ is the root of a max-heap.

Initialisation: Prior to the first iteration of the loop $i = \lfloor n/2 \rfloor$. In that case the nodes $\lfloor n/2 \rfloor, \lfloor n/2 \rfloor + 1, \ldots, n$ are leaves, and hence trivially max-heaps.

Maintenance:

Build-Max-Heap Correctness

We will argue via a loop invariant:

At the start of each iteration of the for loop (lines 2-3), each node $i+1,\ i+2,\ \ldots,\ n$ *is the root of a max-heap.*

Initialisation: Prior to the first iteration of the loop $i = \lfloor n/2 \rfloor$. In that case the nodes $\lfloor n/2 \rfloor, \lfloor n/2 \rfloor + 1, \ldots, n$ are leaves, and hence trivially max-heaps.

Maintenance:

Left(i) and Right(i) have higher indices than i. By the loop invariant, they are roots of max-heaps.

Build-Max-Heap Correctness

We will argue via a loop invariant:

At the start of each iteration of the for loop (lines 2-3), each node $i + 1,\ i + 2,\ \ldots,\ n$ is the root of a max-heap.

Initialisation: Prior to the first iteration of the loop $i = \lfloor n/2 \rfloor$. In that case the nodes $\lfloor n/2 \rfloor, \lfloor n/2 \rfloor + 1, \ldots, n$ are leaves, and hence trivially max-heaps.

Maintenance:

Left(i) and Right(i) have higher indices than i. By the loop invariant, they are roots of max-heaps.

The precondition of Max-Heapify(i) is thus satisfied. By the postcondition, i will be the root of a max-heap. Furthermore, nodes $i + 1,\ \ldots,\ n$ are still roots of max-heaps.

Build-Max-Heap Correctness

We will argue via a loop invariant:

At the start of each iteration of the for loop (lines 2-3), each node $i+1,\ i+2,\ \dots,\ n$ is the root of a max-heap.

Initialisation: Prior to the first iteration of the loop $i = \lfloor n/2 \rfloor$. In that case the nodes $\lfloor n/2 \rfloor, \lfloor n/2 \rfloor + 1, \dots, n$ are leaves, and hence trivially max-heaps.

Maintenance:

Left(i) and Right(i) have higher indices than i. By the loop invariant, they are roots of max-heaps.

The precondition of Max-Heapify(i) is thus satisfied. By the postcondition, i will be the root of a max-heap. Furthermore, nodes $i+1,\ \dots,\ n$ are still roots of max-heaps.

Termination: The loop obviously terminates, when $i = 0$. By the loop invariant, each node is the root of a max-heap, and so is the root.

Build-Max-Heap Running Time

Build-Max-Heap Running Time

Easy bound:

Build-Max-Heap Running Time

Easy bound:

Max-Heapify has running time $O(\lg n)$.

Build-Max-Heap Running Time

Easy bound:

Max-Heapify has running time $O(\lg n)$.

Max-Heapify is called $O(n)$ times.

Build-Max-Heap Running Time

Easy bound:

Max-Heapify has running time $O(\lg n)$.

Max-Heapify is called $O(n)$ times.

Build-Max-Heap has running time $O(n \lg n)$.

Heapsort

Heapsort

1. Preprocess the array to become a *heap*.
   Build-Max-Heap(A, n)

Heapsort

1. Preprocess the array to become a *heap*.

   Build-Max-Heap(A, n)

2. Find the maximum element of the heap in $O(1)$ time and move it to the last position.

   exchange $A[1]$ with $A[i]$ (where initially $i = n$).

Heapsort

1. Preprocess the array to become a *heap*.

    Build-Max-Heap(A, n)

2. Find the maximum element of the heap in $O(1)$ time and move it to the last position.

    exchange $A[1]$ with $A[i]$ (where initially $i = n$).

3. Consider the remaining array and process it to become a heap again.

    A.heap-size = A.heapsize -1

    Max-Heapify$(A,1)$

Heapsort

1. Preprocess the array to become a *heap*.

   Build-Max-Heap(A, n)

2. Find the maximum element of the heap in $O(1)$ time and move it to the last position.

   exchange $A[1]$ with $A[i]$ (where initially $i = n$).

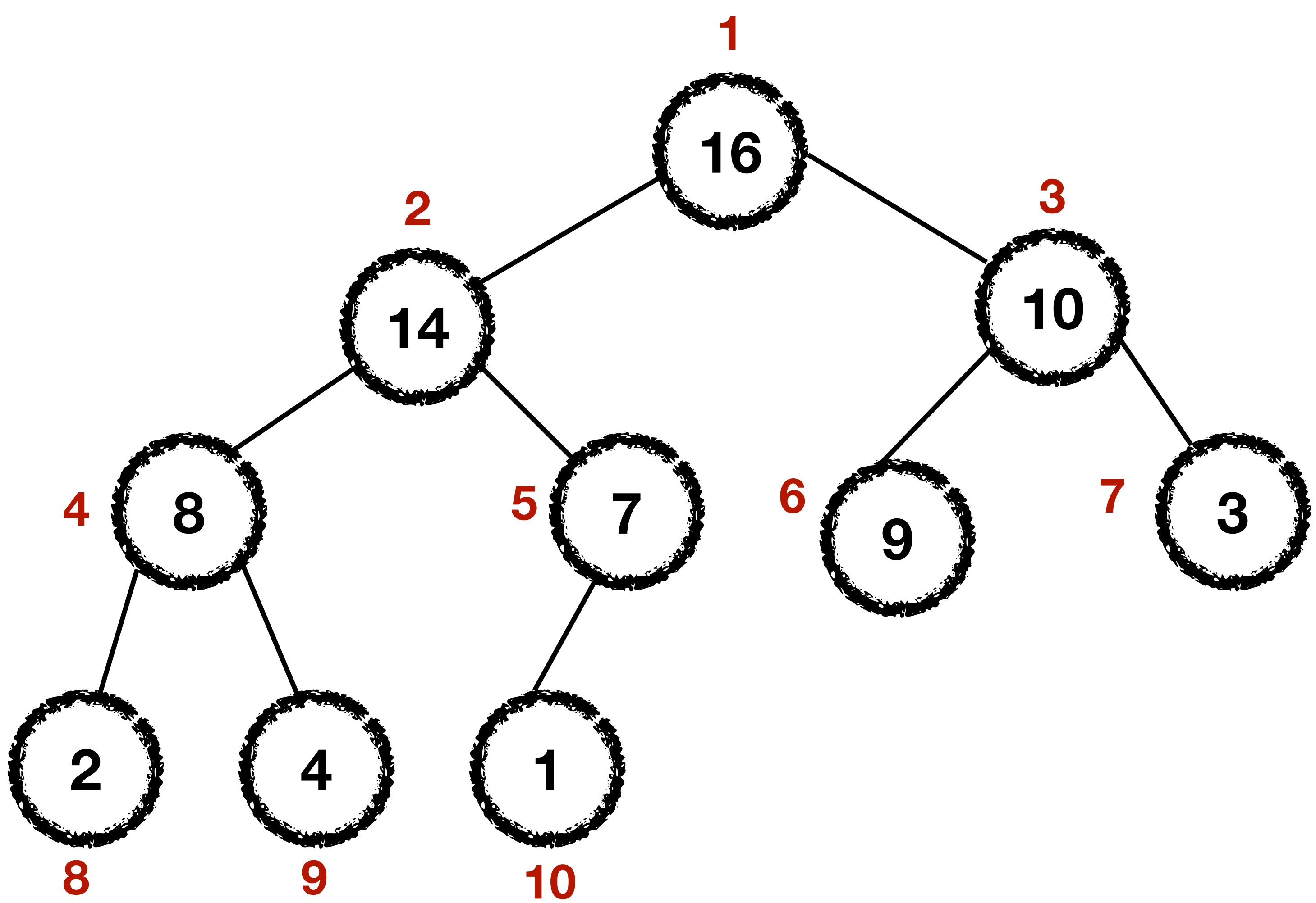
3. Consider the remaining array and process it to become a heap again.

   A.heap-size = A.heapsize -1

   Max-Heapify$(A,1)$

4. Repeat Steps 2-4 for the remaining special array, until the remaining array has size 0.

Heapsort

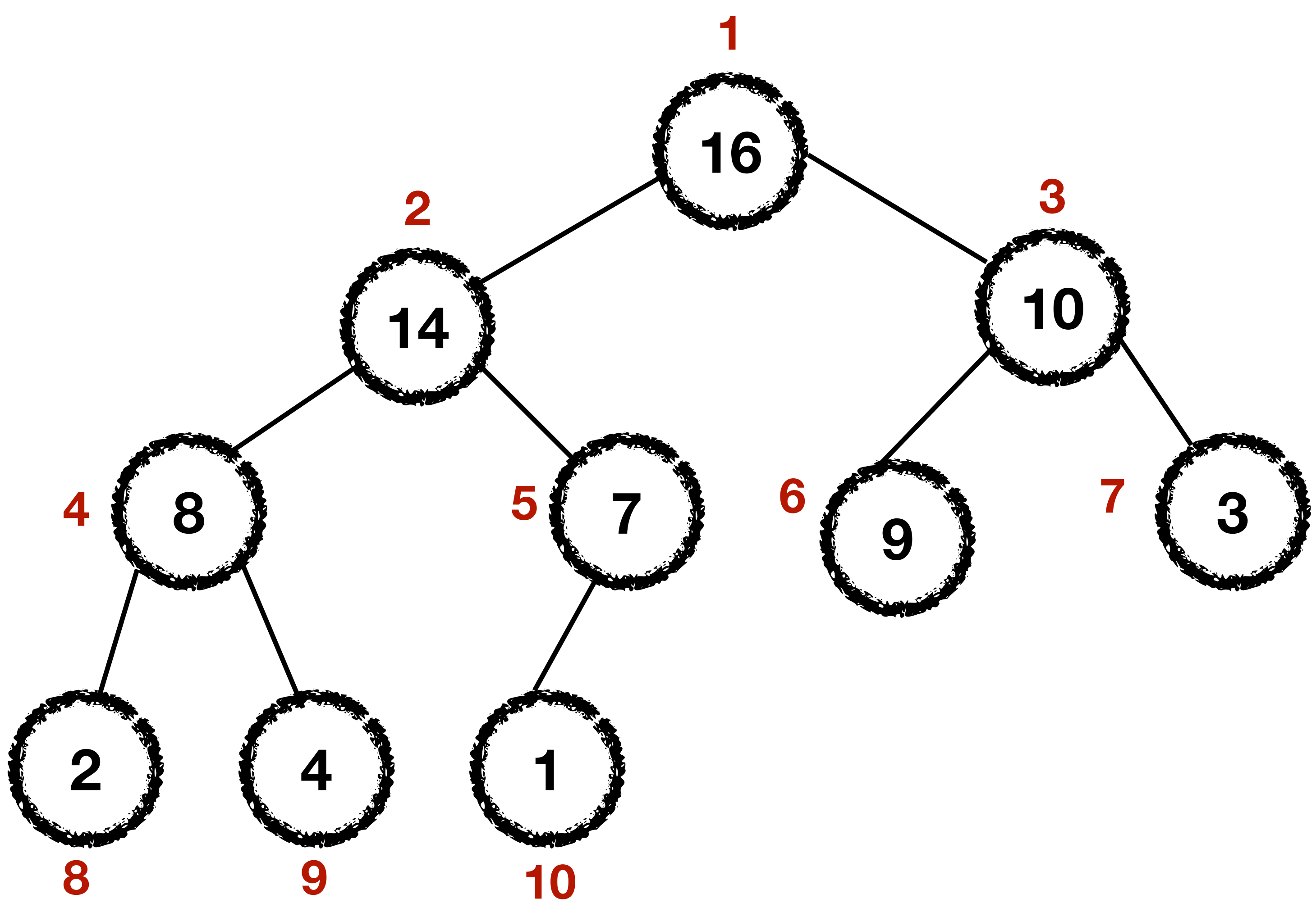



```
HEAPSORT(A, n)
BUILD-MAX-HEAP(A, n)
for i = n downto 2
    exchange A[1] with A[i]
    A.heap-size = A.heap-size − 1
    MAX-HEAPIFY(A, 1)
```

CLRS pp 170

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

Heapsort

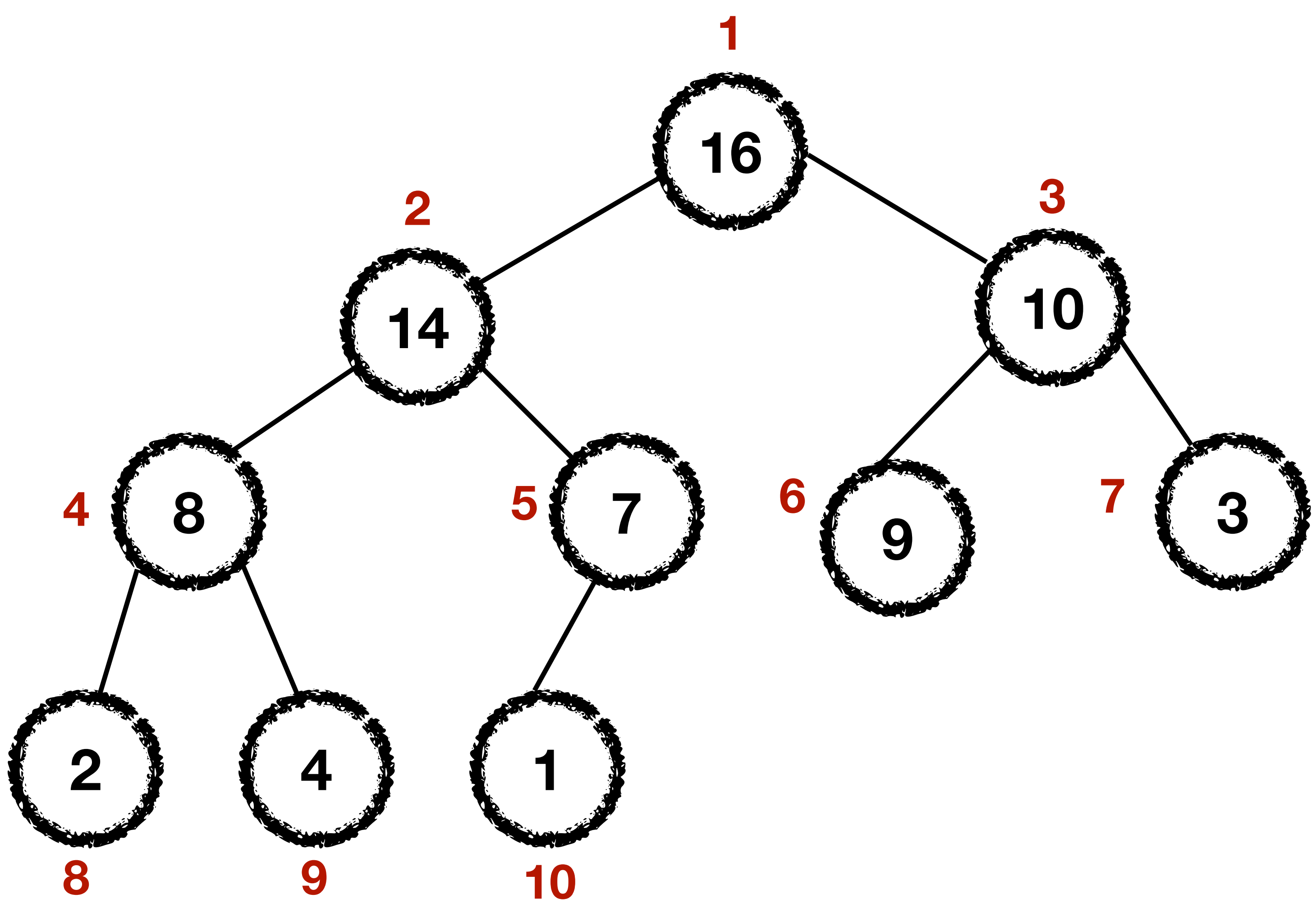



```
HEAPSORT(A, n)
BUILD-MAX-HEAP(A, n)
for i = n downto 2
    exchange A[1] with A[i]
    A.heap-size = A.heap-size − 1
    MAX-HEAPIFY(A, 1)
```

CLRS pp 170

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

Heapsort

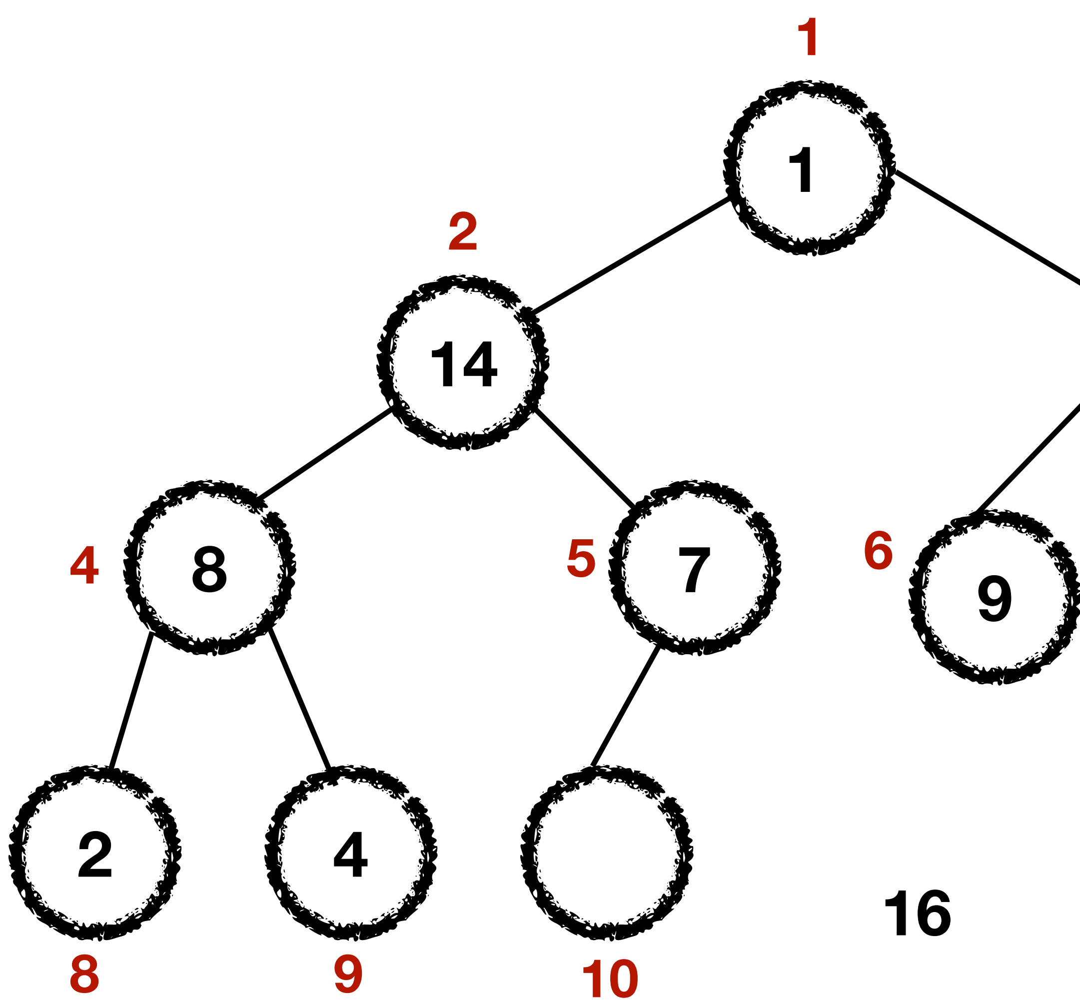



```
HEAPSORT(A, n)
BUILD-MAX-HEAP(A, n)
for i = n downto 2
    exchange A[1] with A[i]
    A.heap-size = A.heap-size − 1
    MAX-HEAPIFY(A, 1)
```

CLRS pp 170

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

Heapsort

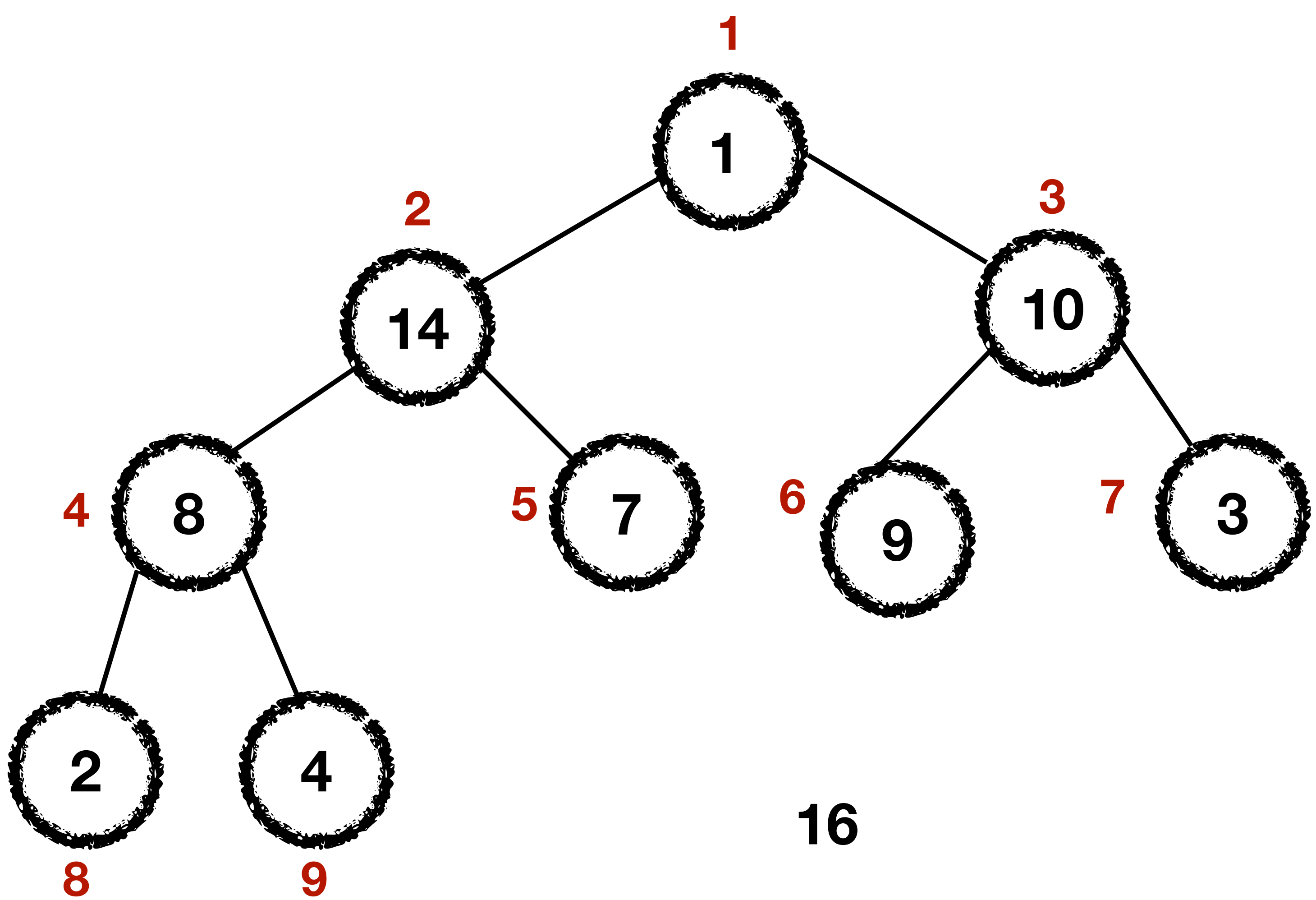



```
HEAPSORT(A, n)
  BUILD-MAX-HEAP(A, n)
  for i = n downto 2
      exchange A[1] with A[i]
      A.heap-size = A.heap-size − 1
      MAX-HEAPIFY(A, 1)
```

CLRS pp 170

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

Heapsort

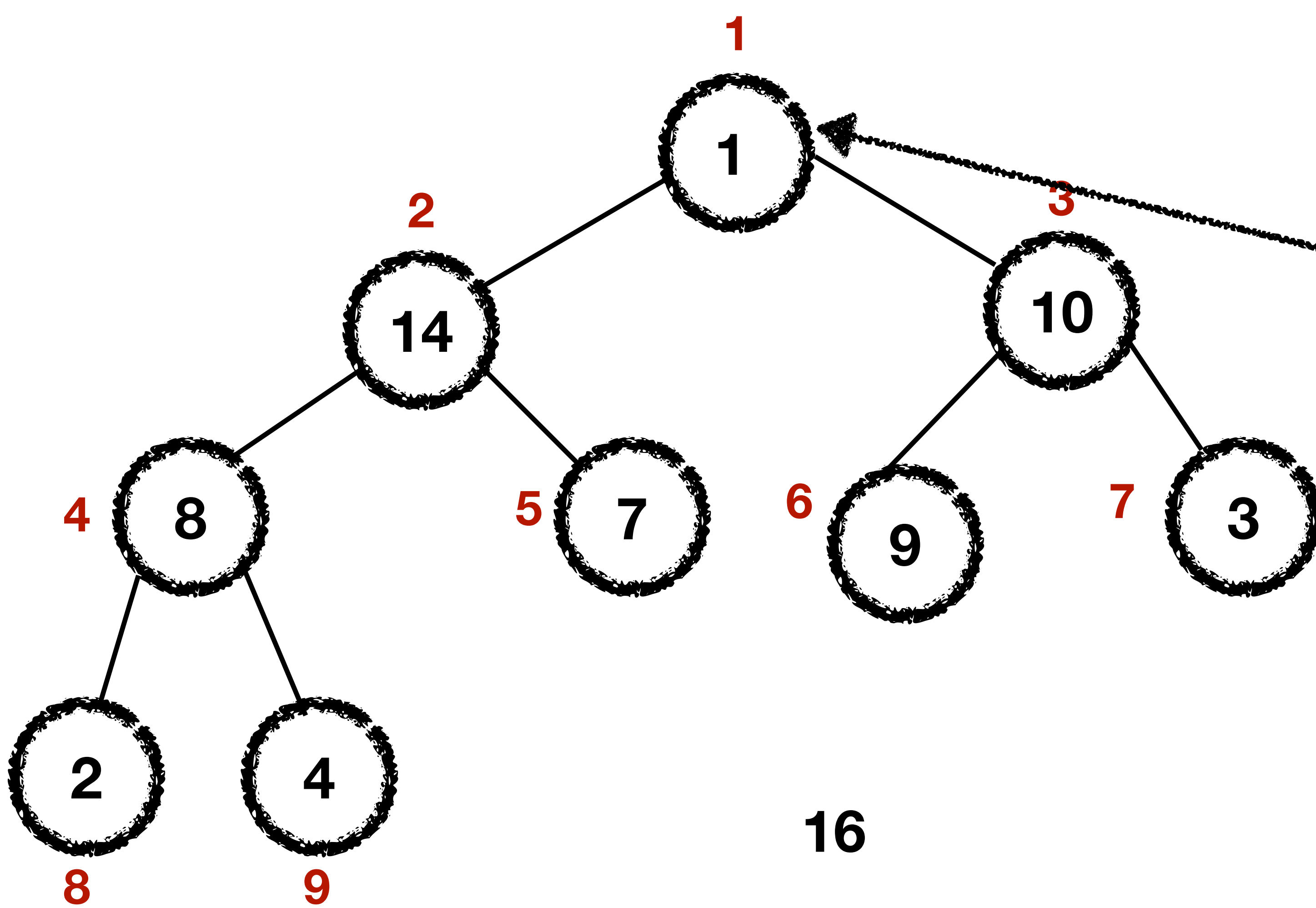



```
HEAPSORT(A, n)
BUILD-MAX-HEAP(A, n)
for i = n downto 2
      exchange A[1] with A[i]
      A.heap-size = A.heap-size − 1
      MAX-HEAPIFY(A, 1)
```

CLRS pp 170

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

Heapsort

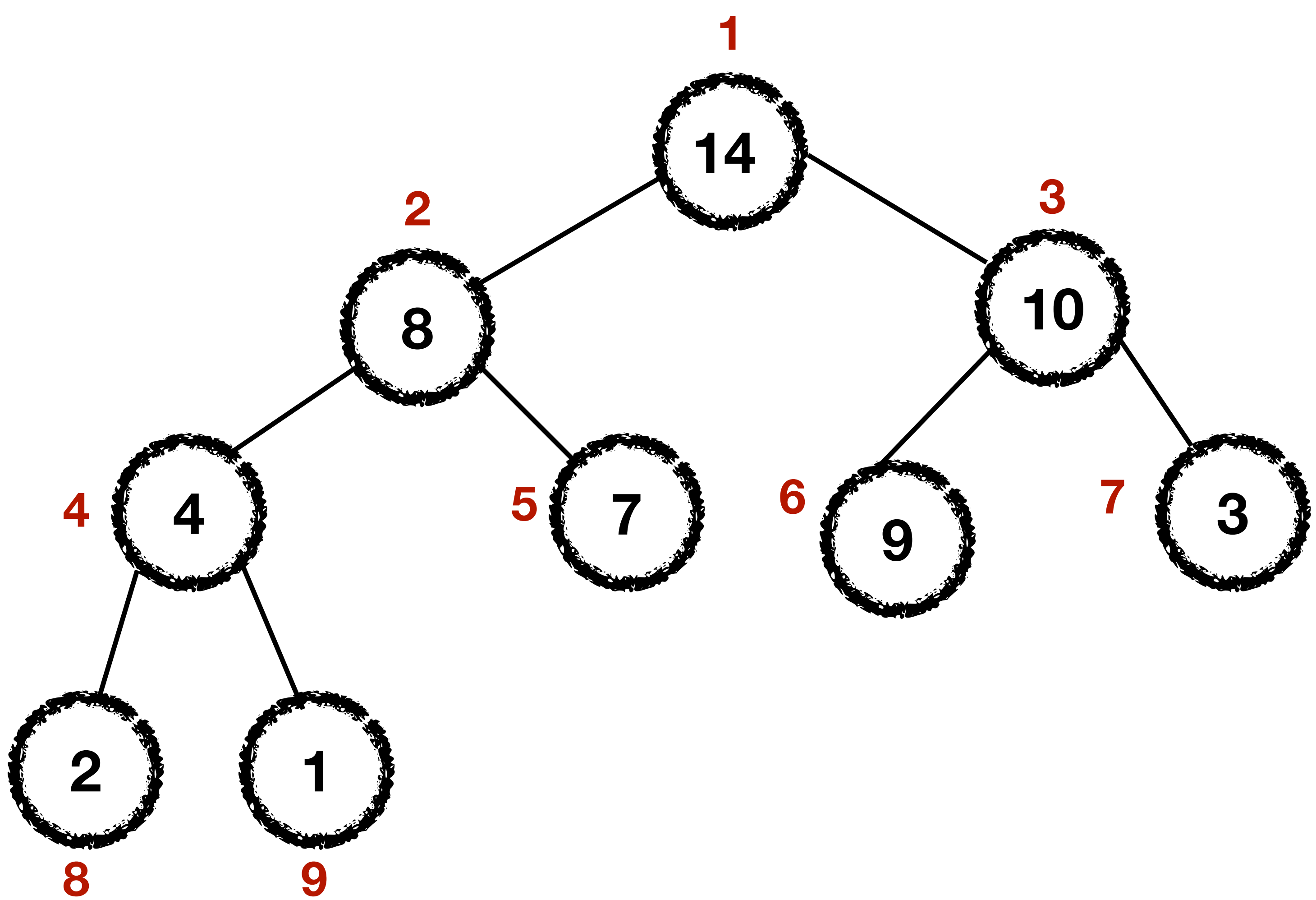



```
HEAPSORT(A, n)
BUILD-MAX-HEAP(A, n)
for i = n downto 2
    exchange A[1] with A[i]
    A.heap-size = A.heap-size − 1
    MAX-HEAPIFY(A, 1)
```

CLRS pp 170

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

Heapsort

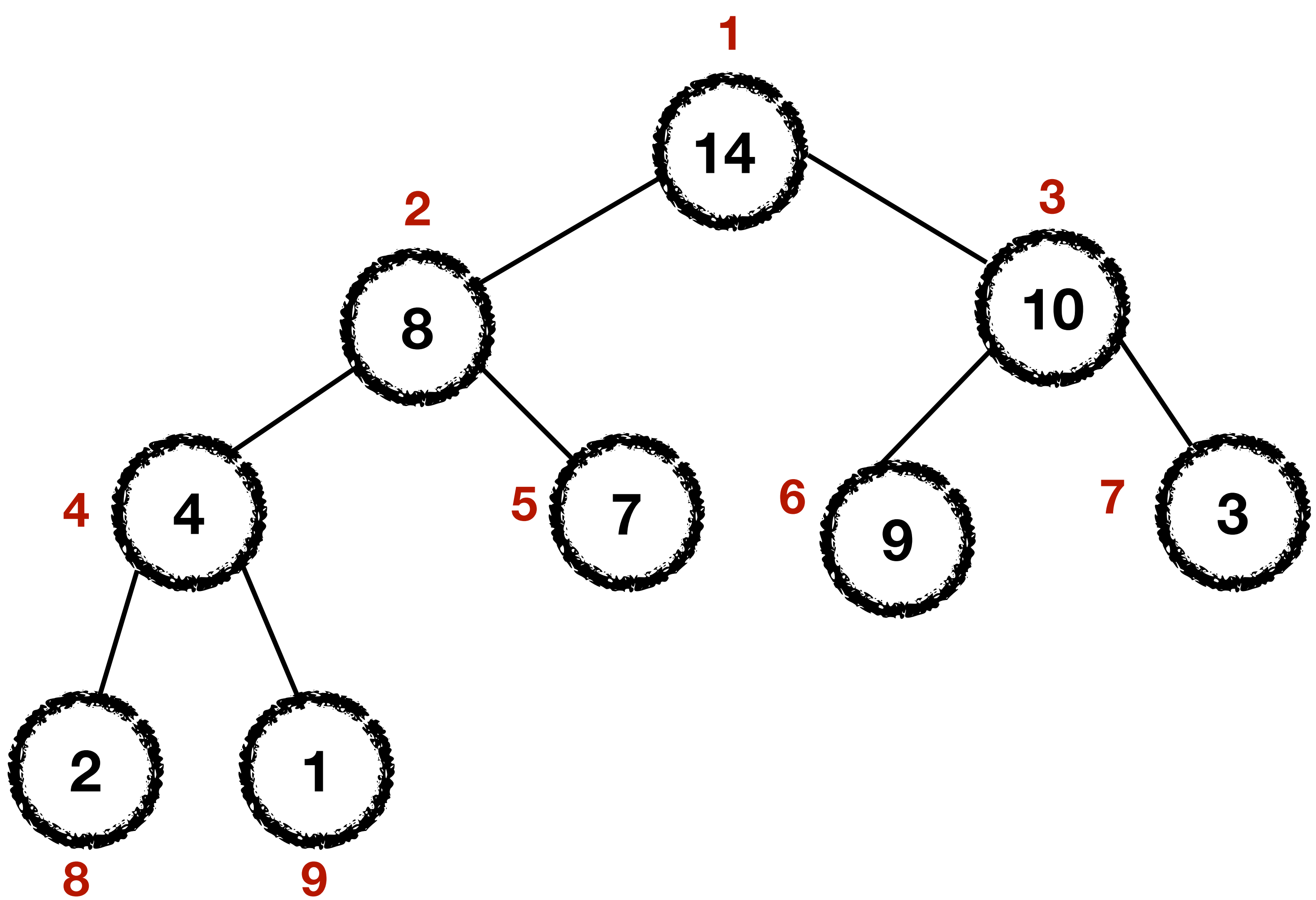



```
HEAPSORT(A, n)
  BUILD-MAX-HEAP(A, n)
  for i = n downto 2
      exchange A[1] with A[i]
      A.heap-size = A.heap-size − 1
      MAX-HEAPIFY(A, 1)
```

CLRS pp 170

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

Heapsort

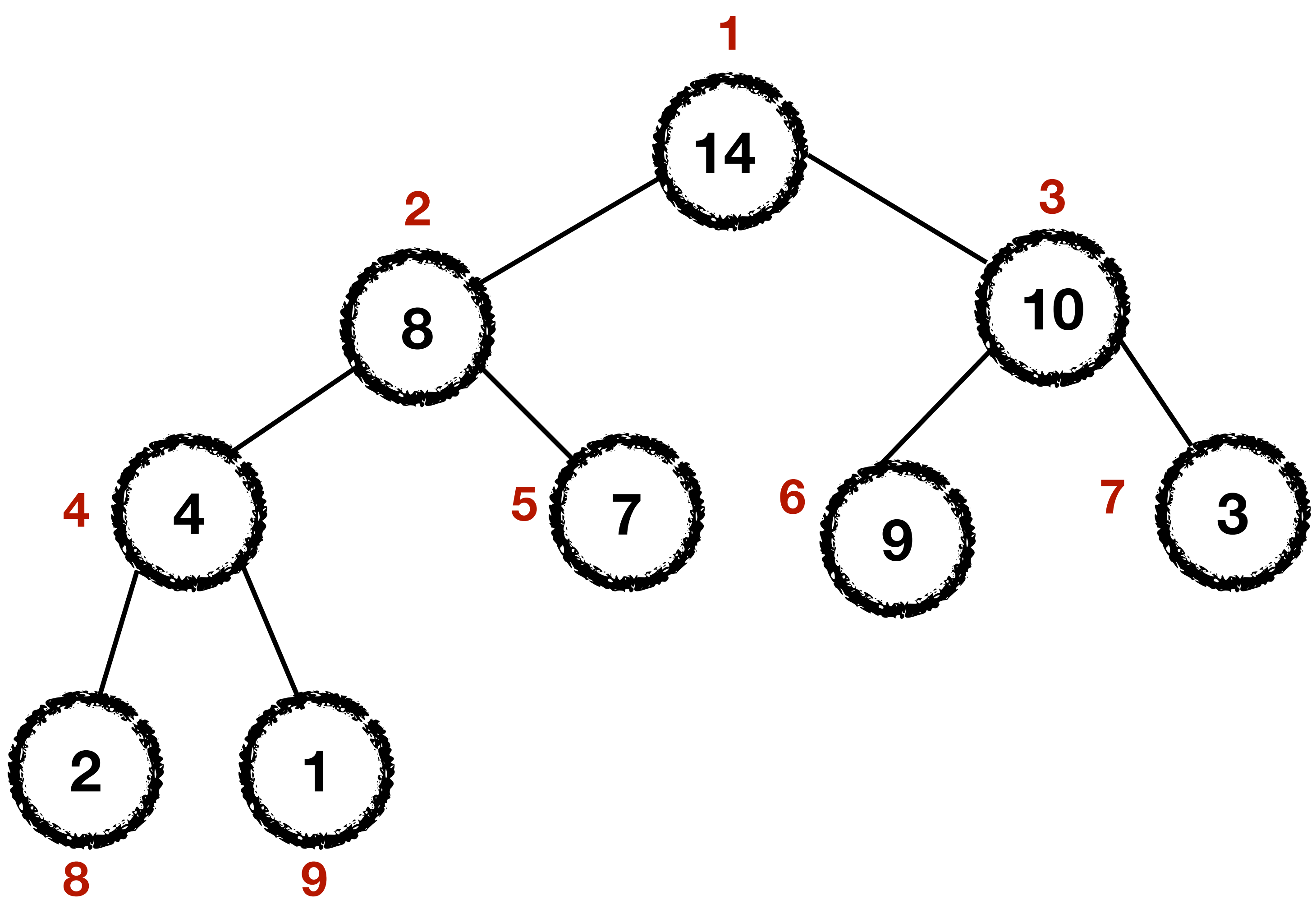



```
HEAPSORT(A, n)
BUILD-MAX-HEAP(A, n)
for i = n downto 2
    exchange A[1] with A[i]
    A.heap-size = A.heap-size − 1
    MAX-HEAPIFY(A, 1)
```

CLRS pp 170

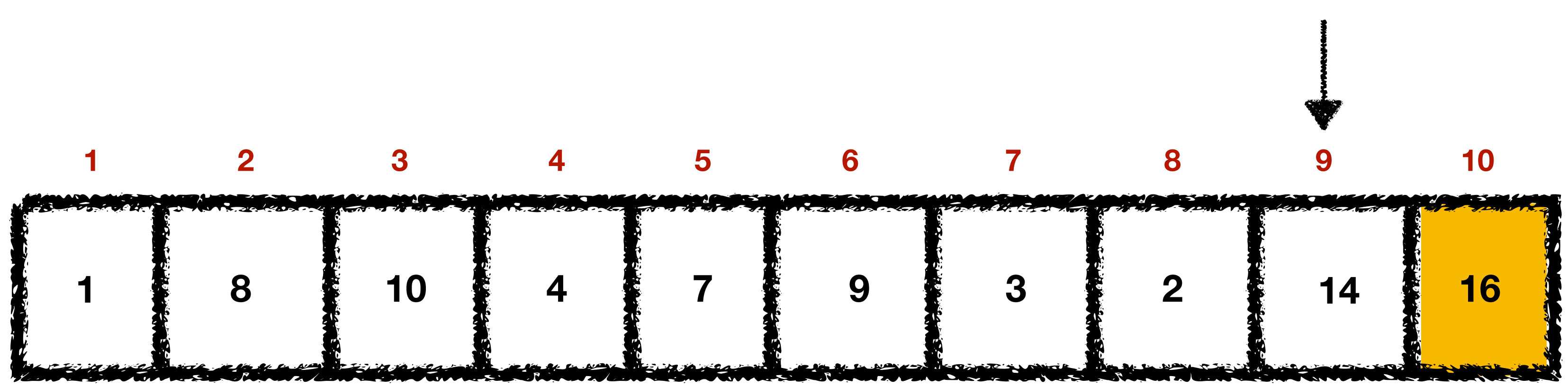
1	2	3	4	5	6	7	8	9	10
14	8	10	4	7	9	3	2	1	16

Heapsort

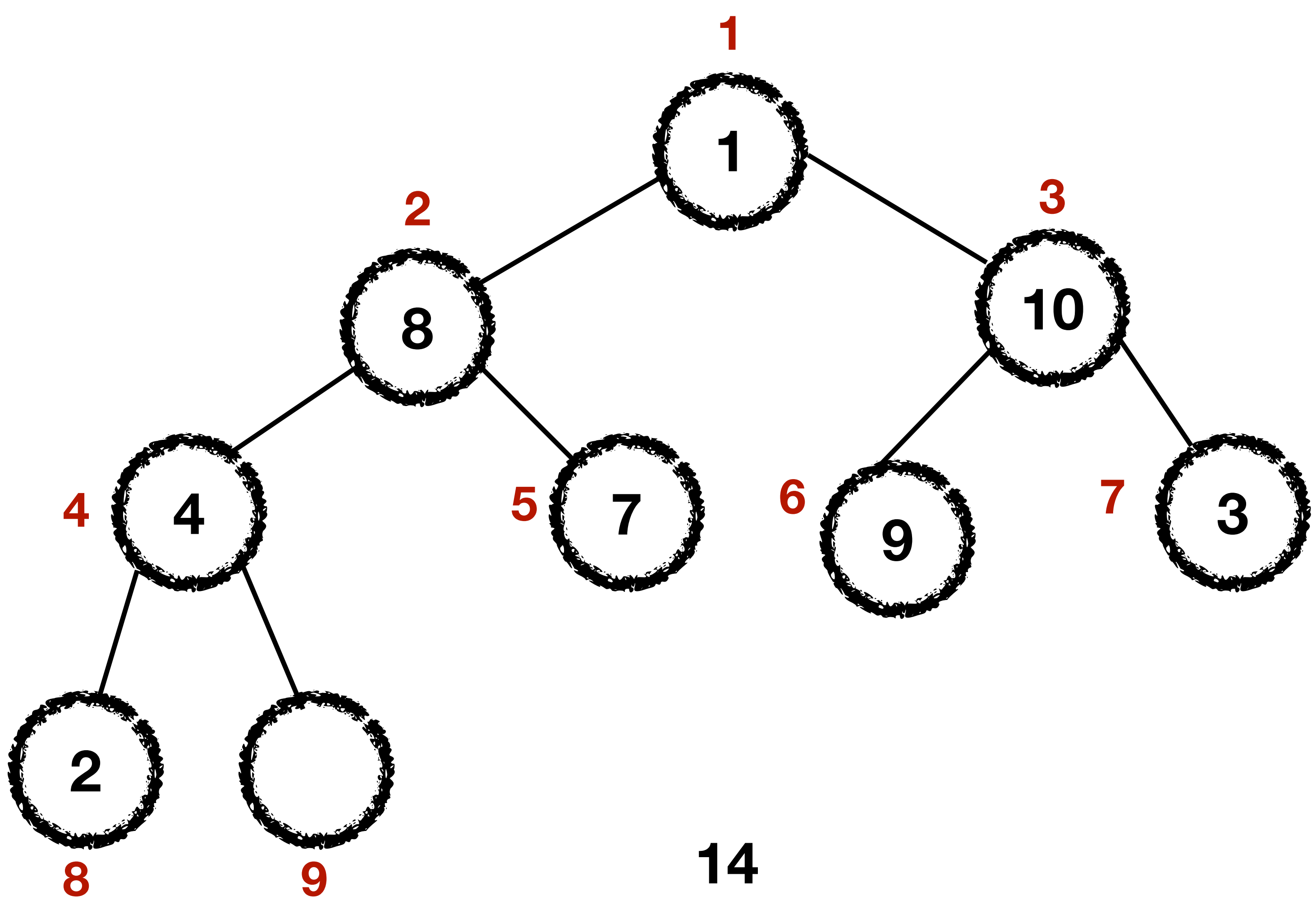



```
HEAPSORT(A, n)
    BUILD-MAX-HEAP(A, n)
    for i = n downto 2
        exchange A[1] with A[i]
        A.heap-size = A.heap-size − 1
        MAX-HEAPIFY(A, 1)
```

CLRS pp 170



Heapsort

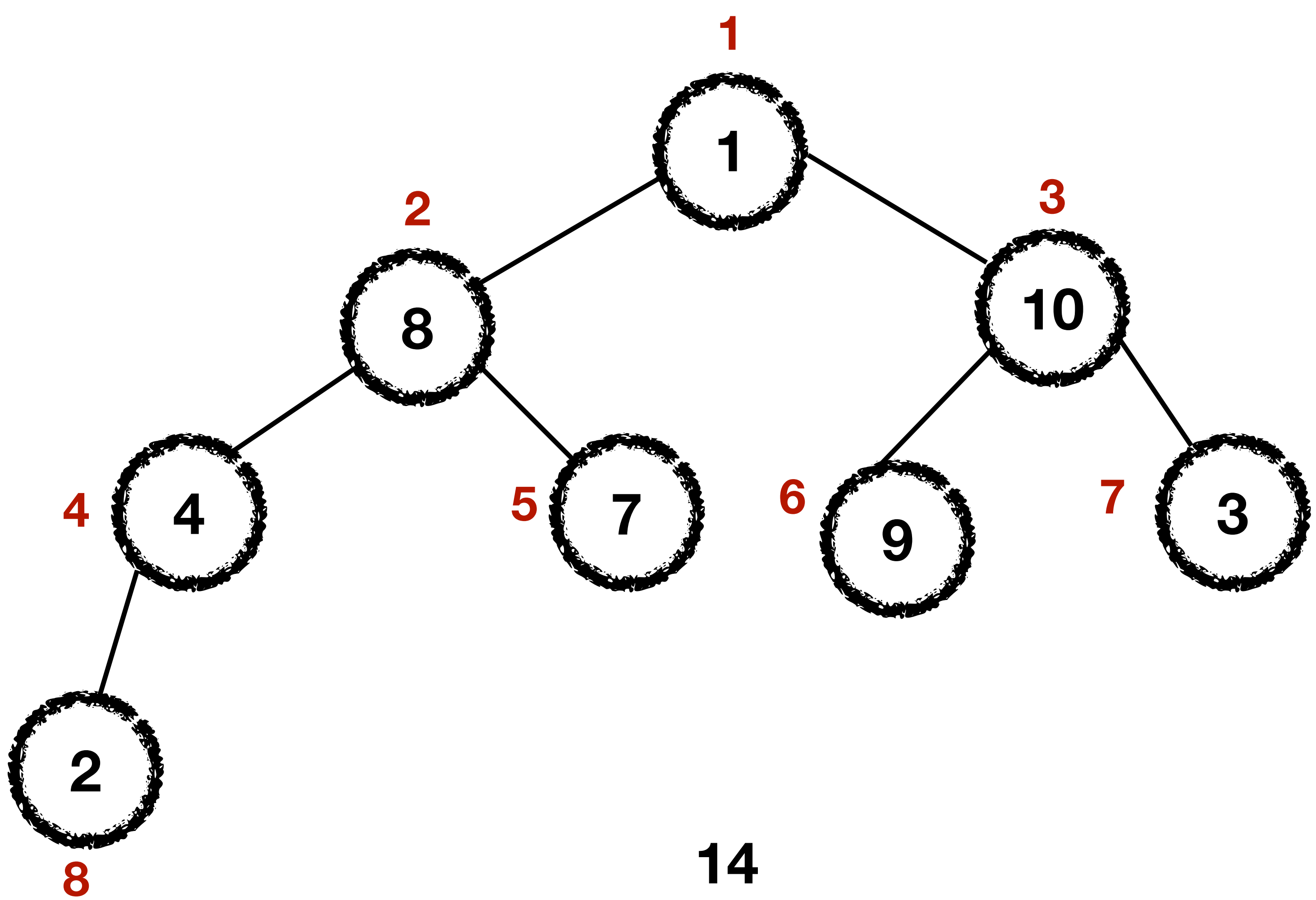



```
HEAPSORT(A, n)
  BUILD-MAX-HEAP(A, n)
  for i = n downto 2
      exchange A[1] with A[i]
      A.heap-size = A.heap-size − 1
      MAX-HEAPIFY(A, 1)
```

CLRS pp 170

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

Heapsort



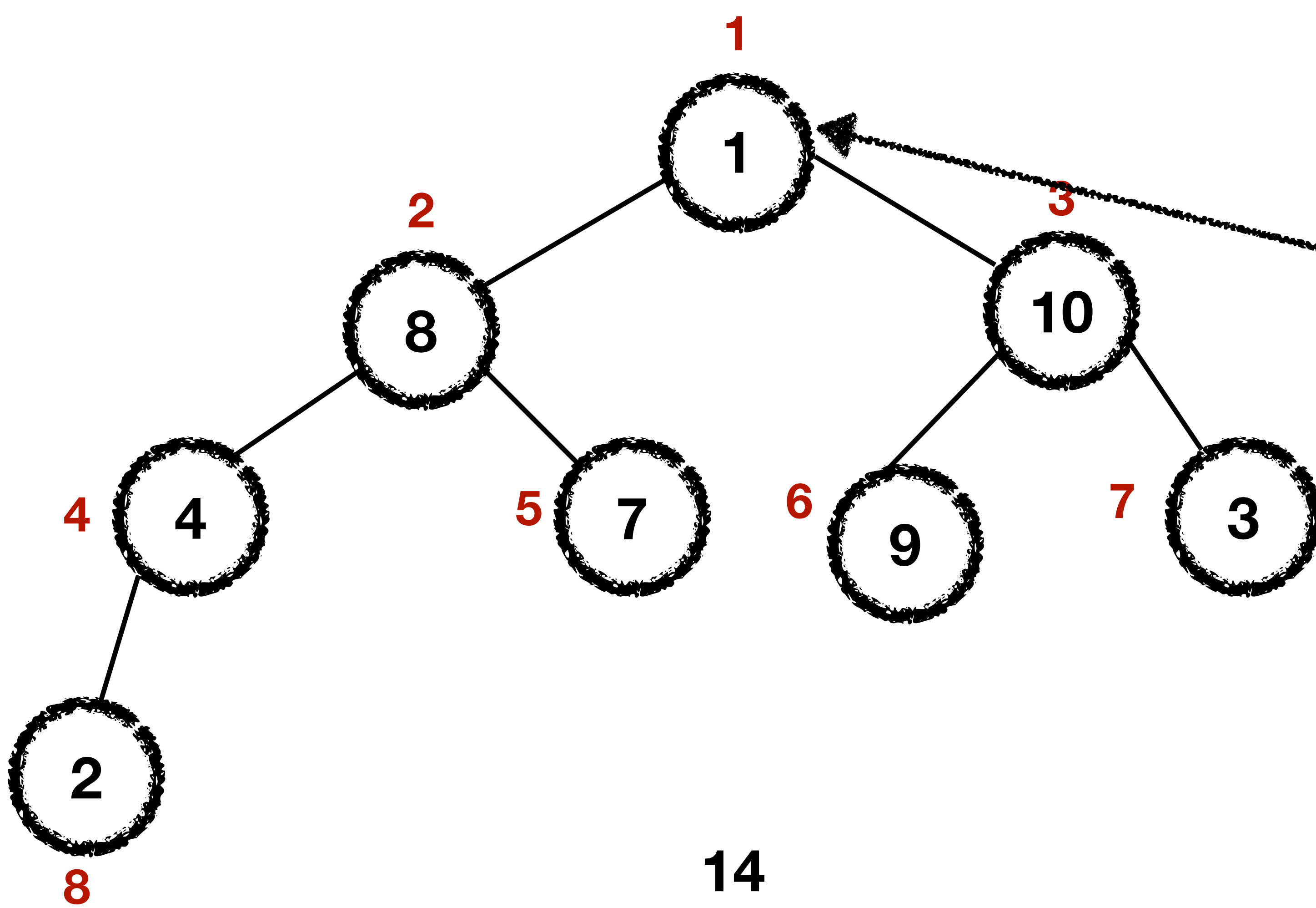
HEAPSORT(A, n)

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)

CLRS pp 170

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

Heapsort

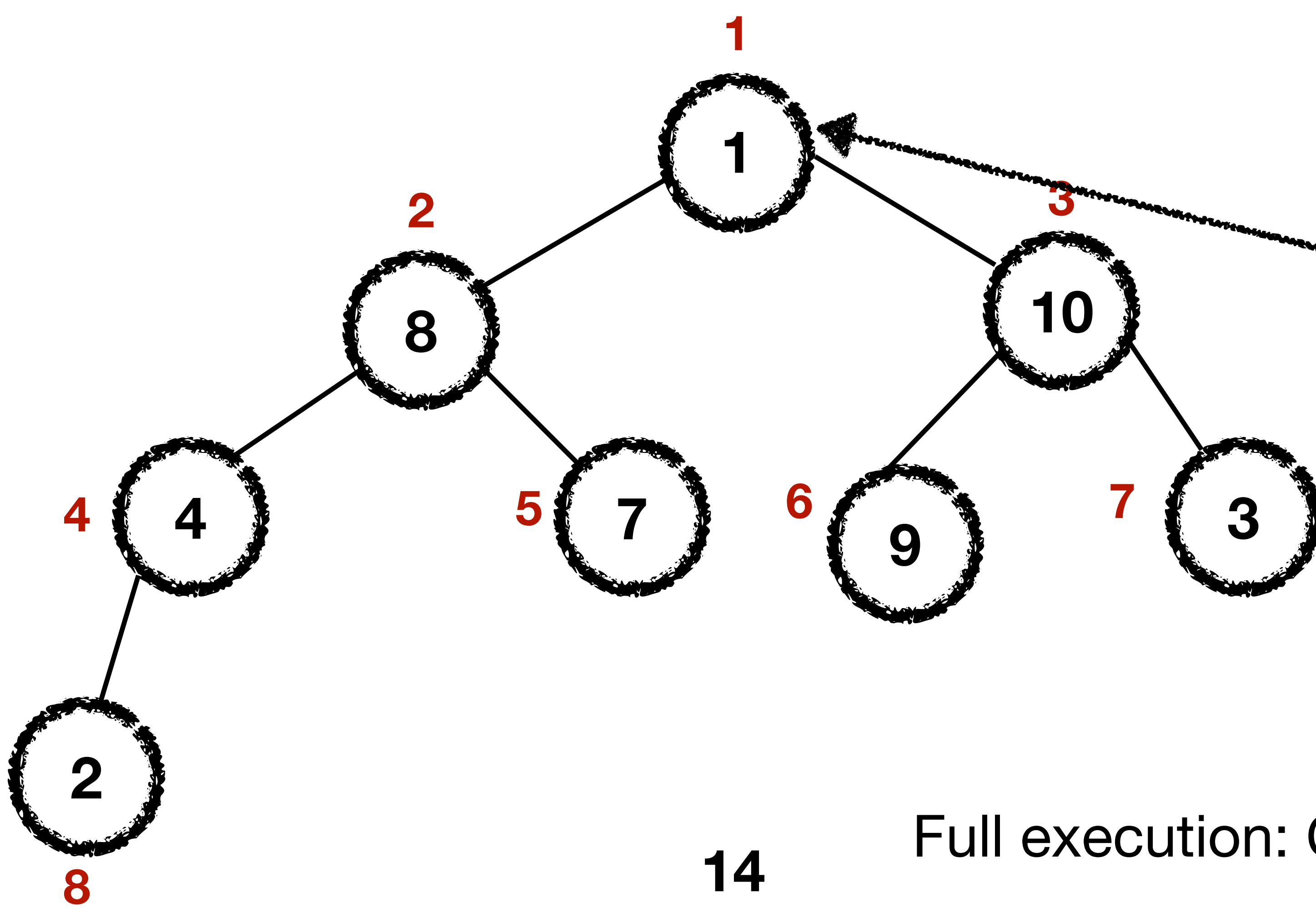



```
HEAPSORT(A, n)
    BUILD-MAX-HEAP(A, n)
    for i = n downto 2
        exchange A[1] with A[i]
        A.heap-size = A.heap-size − 1
        MAX-HEAPIFY(A, 1)
```

CLRS pp 170

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

Heapsort




```
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)
```

CLRS pp 170

14

Full execution: CLRS pp 171

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

Heapsort Running Time

```
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)
```

CLRS pp 170

Heapsort Running Time

```
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)
```

CLRS pp 170

Easy bound:

Max-Heapify has running time $O(\lg n)$.

Max-Heapify is called $O(n)$ times.

Build-Max-Heap has running time $O(n \lg n)$.

Heapsort Running Time

```
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)
```

CLRS pp 170

Running time of Heapsort:

Easy bound:

Max-Heapify has running time $O(\lg n)$.

Max-Heapify is called $O(n)$ times.

Build-Max-Heap has running time $O(n \lg n)$.

Heapsort Running Time

```
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)
```

CLRS pp 170

Running time of Heapsort:

$$\Theta(n \lg n)$$

Easy bound:

Max-Heapify has running time $O(\lg n)$.

Max-Heapify is called $O(n)$ times.

Build-Max-Heap has running time $O(n \lg n)$.

Build-Max-Heap Running Time

Easy bound:

Max-Heapify has running time $O(\lg n)$.

Max-Heapify is called $O(n)$ times.

Build-Max-Heap has running time $O(n \lg n)$.

Build-Max-Heap Running Time

Easy bound:

Max-Heapify has running time $O(\lg n)$.

Max-Heapify is called $O(n)$ times.

Build-Max-Heap has running time $O(n \lg n)$.

Is this really *tight*?

Build-Max-Heap Running Time, better analysis

Refined bound:

Build-Max-Heap Running Time, better analysis

Refined bound:

For Max-Heapify, we actually proved that

Build-Max-Heap Running Time, better analysis

Refined bound:

For Max-Heapify, we actually proved that

$$
\begin{aligned}
T(h) &\le (h+1)\cdot O(1) \\
&= \boxed{O(h)} = O(\lg n)
\end{aligned}
$$

Build-Max-Heap Running Time, better analysis

Refined bound:

For Max-Heapify, we actually proved that

$$
\begin{aligned}
T(h) &\leq (h+1) \cdot O(1) \\
&= O(h) = O(\lg n)
\end{aligned}
$$

In other words, there is a constant c such that $T(h) \leq ch$ for sufficiently large h.

Build-Max-Heap Running Time, better analysis

Refined bound:

For Max-Heapify, we actually proved that

$$\begin{aligned} T(h) &\le (h+1) \cdot O(1) \\ &= \boxed{O(h)} = O(\lg n) \end{aligned}$$

In other words, there is a constant c such that $T(h) \le ch$ for sufficiently large h.

Our cost depends on the height of the subtree we are considering!

Build-Max-Heap Running Time, better analysis

Refined bound:

$$
\begin{aligned}
T(\text{Build-Max-Heap}(A, n)) &\le \sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil ch \\
&\le \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{n}{2^h} ch \\
&\le cn \sum_{h=0}^{\infty} \frac{h}{2^h} \\
&\le cn \cdot \frac{1/2}{(1 - 1/2)^2} \\
&= O(n)
\end{aligned}
$$

Build-Max-Heap Running Time, better analysis

Refined bound:

$$
\begin{aligned}
T(\text{Build-Max-Heap}(A, n)) &\le \sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil ch \\
&\le \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{n}{2^h} ch \\
&\le cn \sum_{h=0}^{\infty} \frac{h}{2^h} \\
&\le cn \cdot \frac{1/2}{(1-1/2)^2} \\
&= O(n)
\end{aligned}
$$

straightforward

Build-Max-Heap Running Time, better analysis

Refined bound:

$$T(\text{Build-Max-Heap}(A, n)) \leq \sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil ch$$

$\lceil n/2^{h+1} \rceil \leq n/2^h$

$$\leq \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{n}{2^h} ch$$

straightforward

$$\leq cn \sum_{h=0}^{\infty} \frac{h}{2^h}$$

$$\leq cn \cdot \frac{1/2}{(1 - 1/2)^2}$$

$$= O(n)$$

Build-Max-Heap Running Time, better analysis

Refined bound:

$$
\begin{aligned}
T(\text{Build-Max-Heap}(A,n)) &\le \sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil ch \\
&\le \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{n}{2^h} ch \\
&\le cn \sum_{h=0}^{\infty} \frac{h}{2^h} \\
&\le cn \cdot \frac{1/2}{(1-1/2)^2} \\
&= O(n)
\end{aligned}
$$

$\lceil n/2^{h+1} \rceil \le n/2^h$

straightforward

Build-Max-Heap Running Time, better analysis

Refined bound:

$$T(\text{Build-Max-Heap}(A, n)) \leq \sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil ch$$

height(heap) = $\lfloor \lg n \rfloor$

$$\leq \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{n}{2^h} ch$$

$\lceil n/2^{h+1} \rceil \leq n/2^h$

$$\leq cn \sum_{h=0}^{\infty} \frac{h}{2^h}$$

straightforward

$$\leq cn \cdot \frac{1/2}{(1 - 1/2)^2}$$

$$= O(n)$$

Build-Max-Heap Running Time, better analysis

Refined bound:

$$T(\text{Build-Max-Heap}(A, n)) \le \sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil ch$$

height(heap) = $\lfloor \lg n \rfloor$

There are at most $\lceil n/2^{h+1} \rceil$ nodes of any height h

$$\le \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{n}{2^h} ch$$

$\lceil n/2^{h+1} \rceil \le n/2^h$

$$\le cn \sum_{h=0}^{\infty} \frac{h}{2^h}$$

straightforward

$$\le cn \cdot \frac{1/2}{(1 - 1/2)^2}$$

$$= O(n)$$

Build-Max-Heap Running Time, better analysis

Refined bound:

$$T(\text{Build-Max-Heap}(A, n)) \leq \sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil ch$$

height(heap) = $\lfloor \lg n \rfloor$

There are at most $\lceil n/2^{h+1} \rceil$ nodes of any height h

$\lceil n/2^{h+1} \rceil \leq n/2^h$

$$\leq \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{n}{2^h} ch$$

straightforward

$$\leq cn \sum_{h=0}^{\infty} \frac{h}{2^h}$$

$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$, **for** $|x| < 1$, **with** $x = 1/2$.

$$\leq cn \cdot \frac{1/2}{(1-1/2)^2}$$

$$= O(n)$$

Heapsort Properties

- Invented by J.W.J. Williams in 1964 (the Heap too!)
- It is an *in-place* algorithm (no auxiliary array).
- It is not a *stable* algorithm (i.e., it does not maintain the relative order between equal keys).
- Whether equal keys are allowed or not influences the best-case running time of the algorithm (think about it!).
- Next time: More about the heap - what else is it good for?