Introduction to Algorithms and Data Structures

Heap Operations and Priority Queues

Last lecture

- Max Heap data structure
- Operations:
  - Max-Heapify
  - Bulid-Max-Heap
- Using these, we presented Heapsort, a sorting algorithm with worst-case running time O(n lg n).

This lecture

- More Max Heap operations:
  - Max-Heap-Extract-Max(A)
  - Max-Heap-Insert(A, v)
- Using Heaps to implement Priority Queues

Heap operations

- Max-Heap-Extract-Max(A):
  Extract and return the maximum element of the heap, and also remove it from the heap.

- Max-Heap-Insert(A, v):
  Insert a new element v to the heap.

Max-Heap-Extract-Max(A):

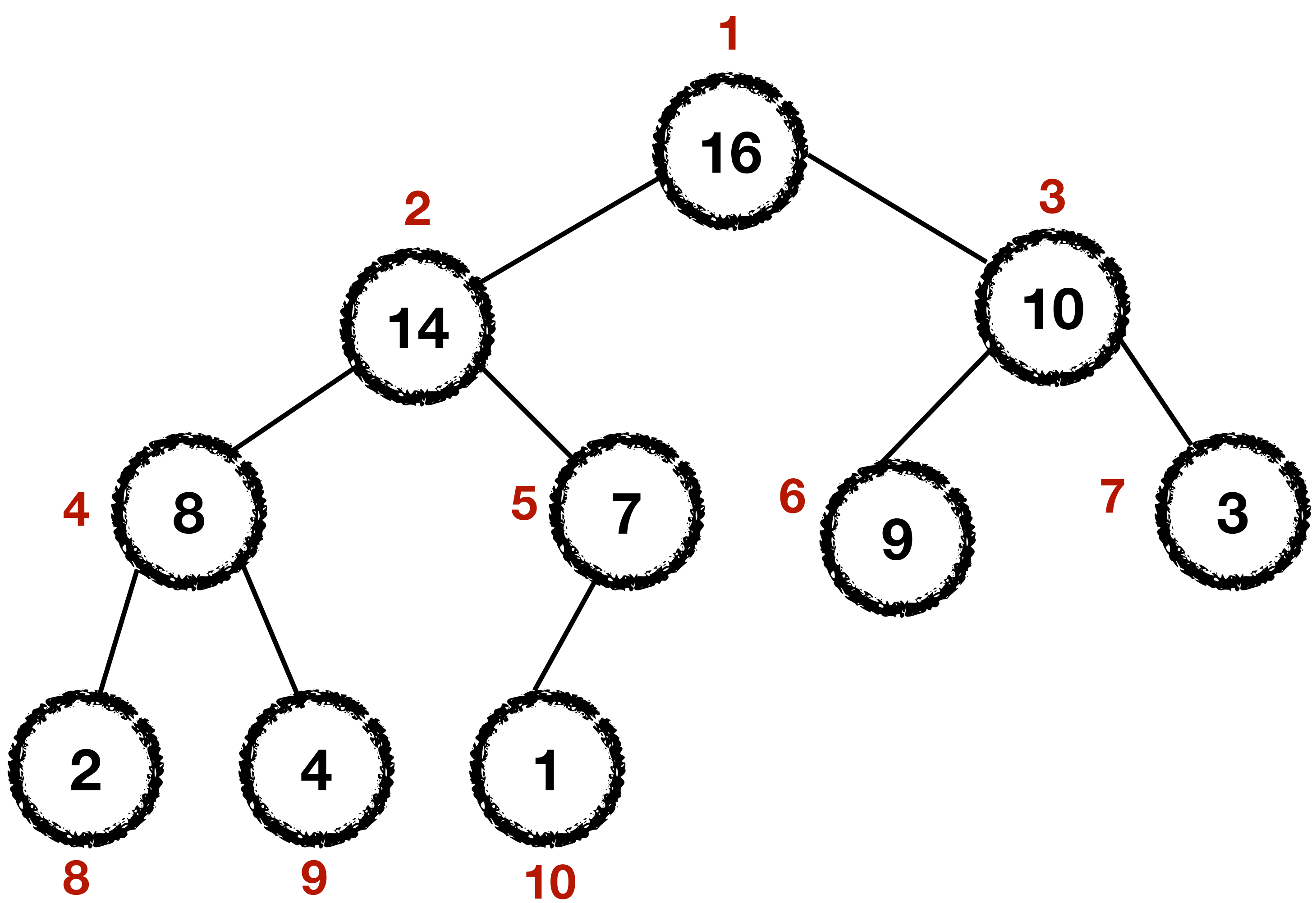
- Max-Heap-Extract-Max(A):
  Extract and return the maximum element of the heap, and also remove it from the heap.

Max-Heap-Extract-Max(A):

- Max-Heap-Extract-Max(A):
  Extract and return the maximum element of the heap, and also remove it from the heap.

- How can we do this?

Heapsort

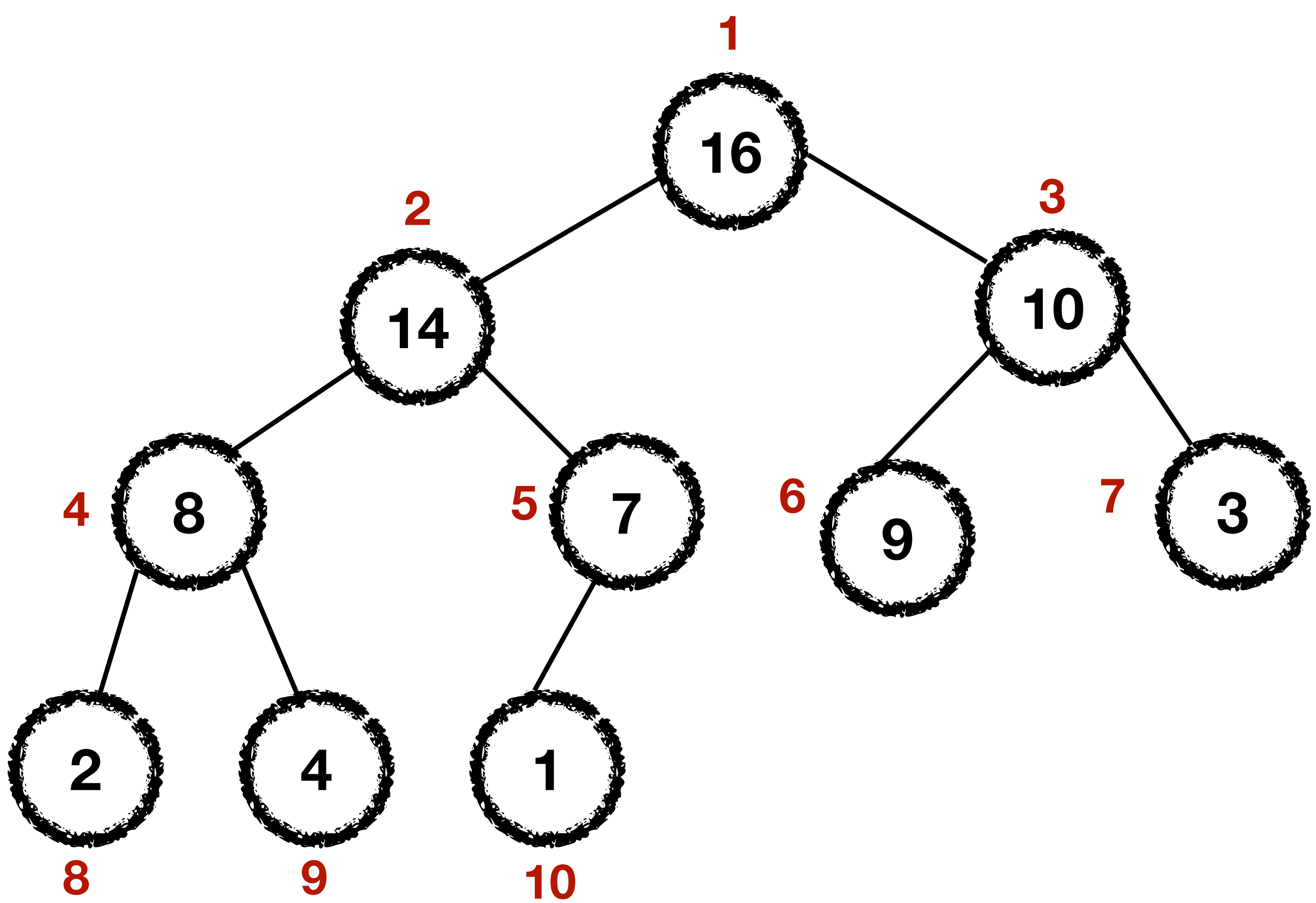



```
HEAPSORT(A, n)
BUILD-MAX-HEAP(A, n)
for i = n downto 2
    exchange A[1] with A[i]
    A.heap-size = A.heap-size − 1
    MAX-HEAPIFY(A, 1)
```

CLRS pp 170

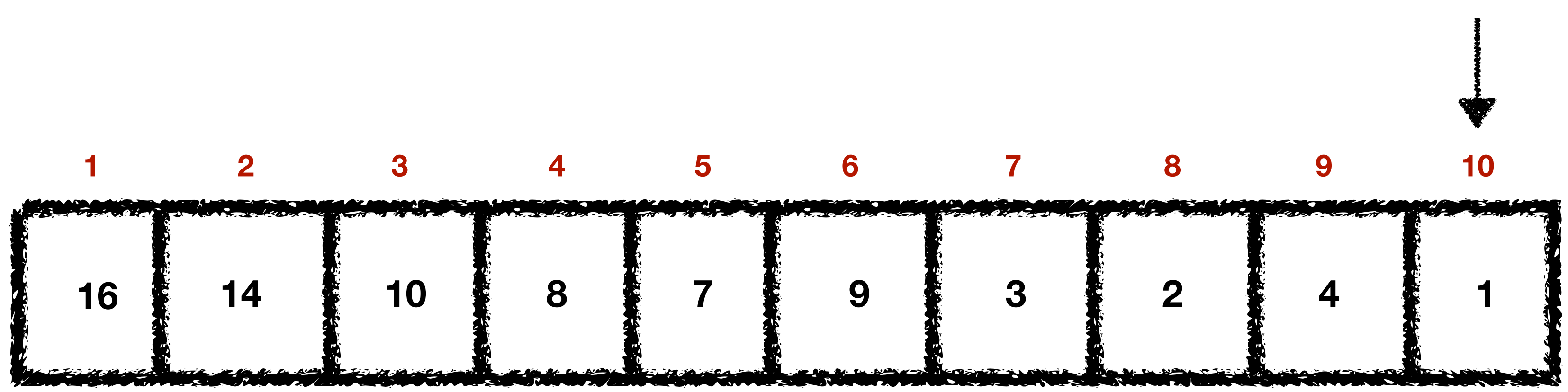
1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

Heapsort

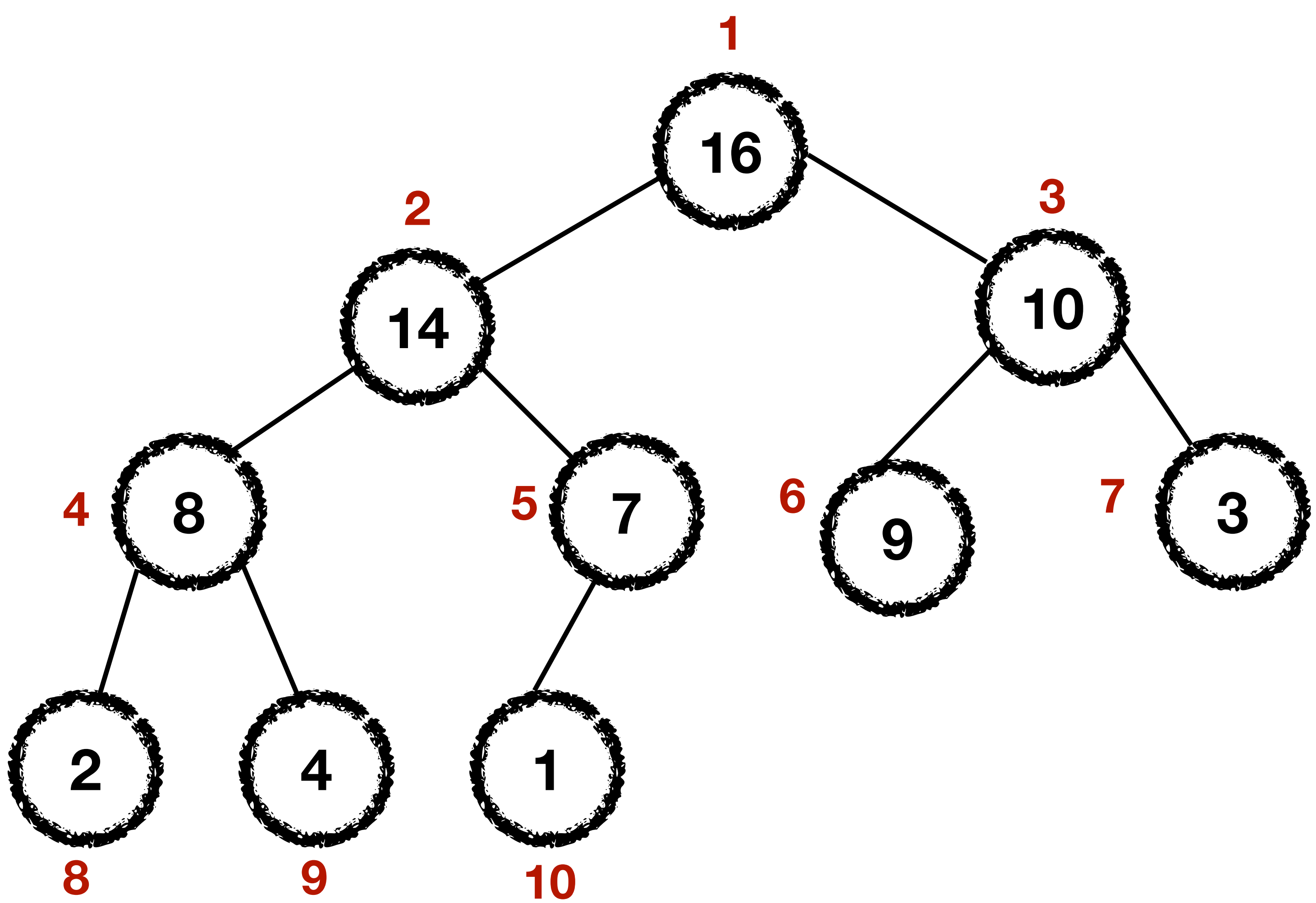



```
HEAPSORT(A, n)
    BUILD-MAX-HEAP(A, n)
    for i = n downto 2
        exchange A[1] with A[i]
        A.heap-size = A.heap-size − 1
        MAX-HEAPIFY(A, 1)
```

CLRS pp 170



Heapsort

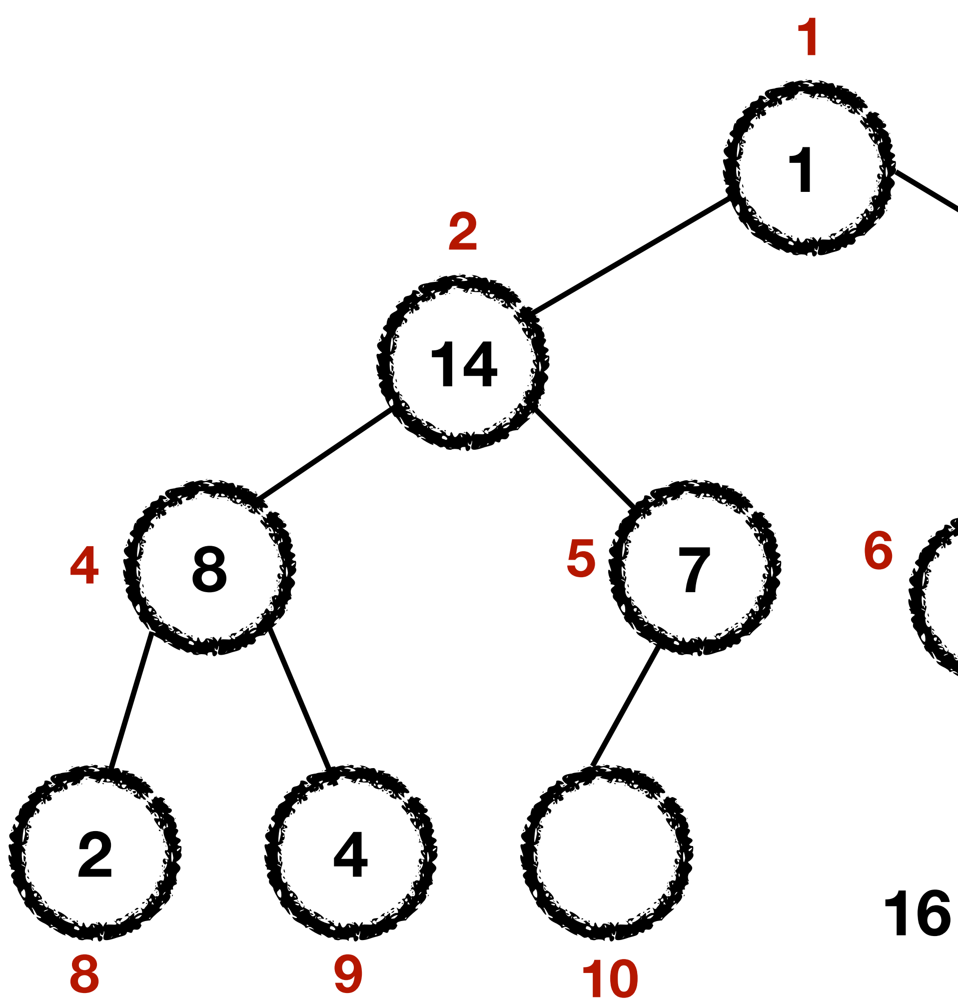



```
HEAPSORT(A, n)
BUILD-MAX-HEAP(A, n)
for i = n downto 2
    exchange A[1] with A[i]
    A.heap-size = A.heap-size − 1
    MAX-HEAPIFY(A, 1)
```

CLRS pp 170

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

Heapsort

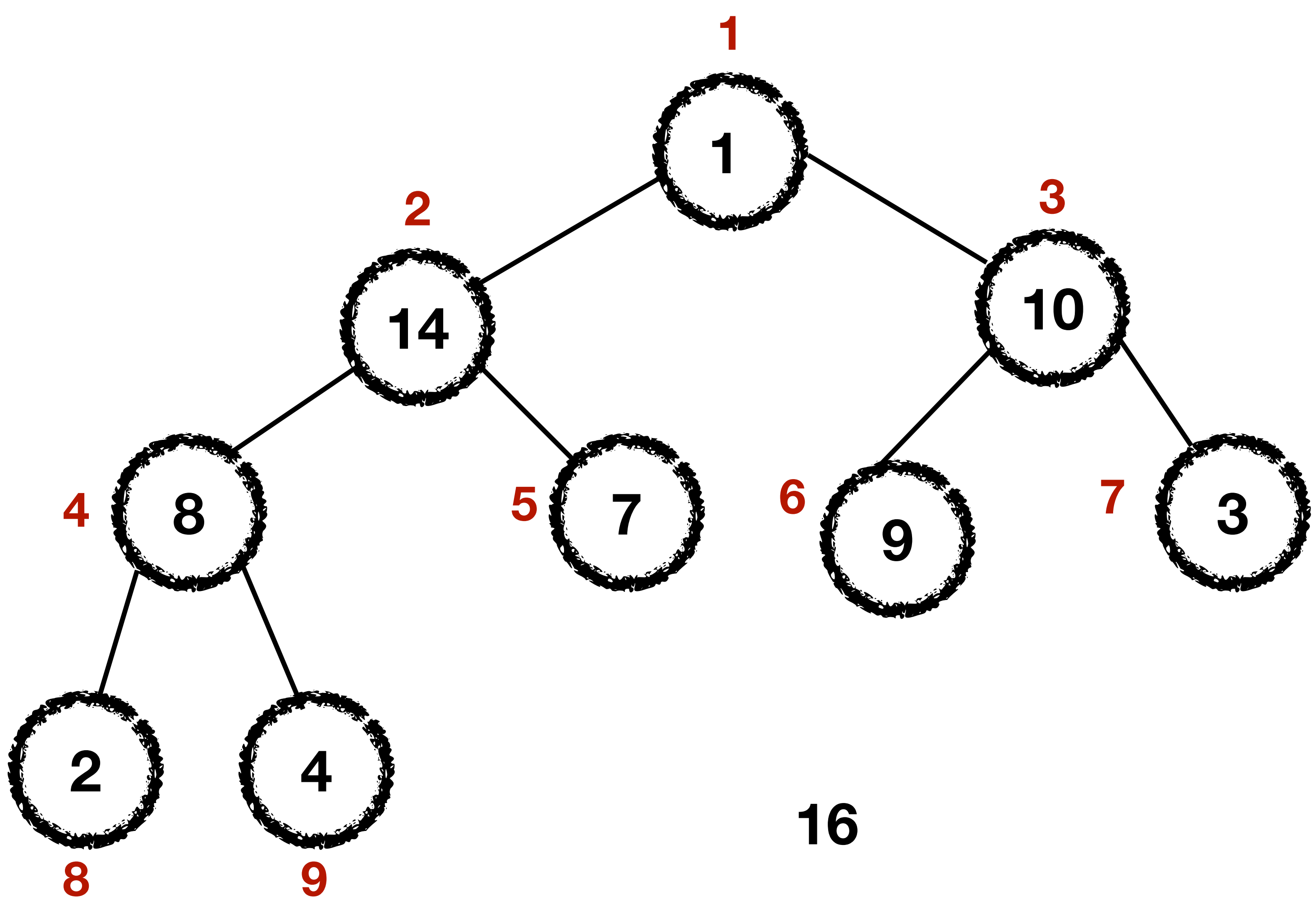



```
HEAPSORT(A, n)
    BUILD-MAX-HEAP(A, n)
    for i = n downto 2
        exchange A[1] with A[i]
        A.heap-size = A.heap-size − 1
        MAX-HEAPIFY(A, 1)
```

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

Max-Heap-Extract-Max(A):

- Max-Heap-Extract-Max(A):
  Extract and return the maximum element of the heap, and also remove it from the heap.

- How can we do this?

Max-Heap-Extract-Max (A)

max = $A[1]$
$A[1]$ = $A[A$. heap-size]
A.heap-size $=$ A.heap-size $- 1$
Max-Heapify$(A, 1)$
return max

Max-Heap-Extract-Max(A):

- Max-Heap-Extract-Max(A):
  Extract and return the maximum element of the heap, and also remove it from the heap.

- How can we do this?

Max-Heap-Extract-Max (A)

max = $A[1]$
$A[1] = A[A$. heap-size]
A.heap-size $= A$.heap-size $- 1$
Max-Heapify$(A, 1)$
return max

Running time?

Max-Heap-Extract-Max(A):

- Max-Heap-Extract-Max(A):
  Extract and return the maximum element of the heap, and also remove it from the heap.

- How can we do this?

Max-Heap-Extract-Max (A)

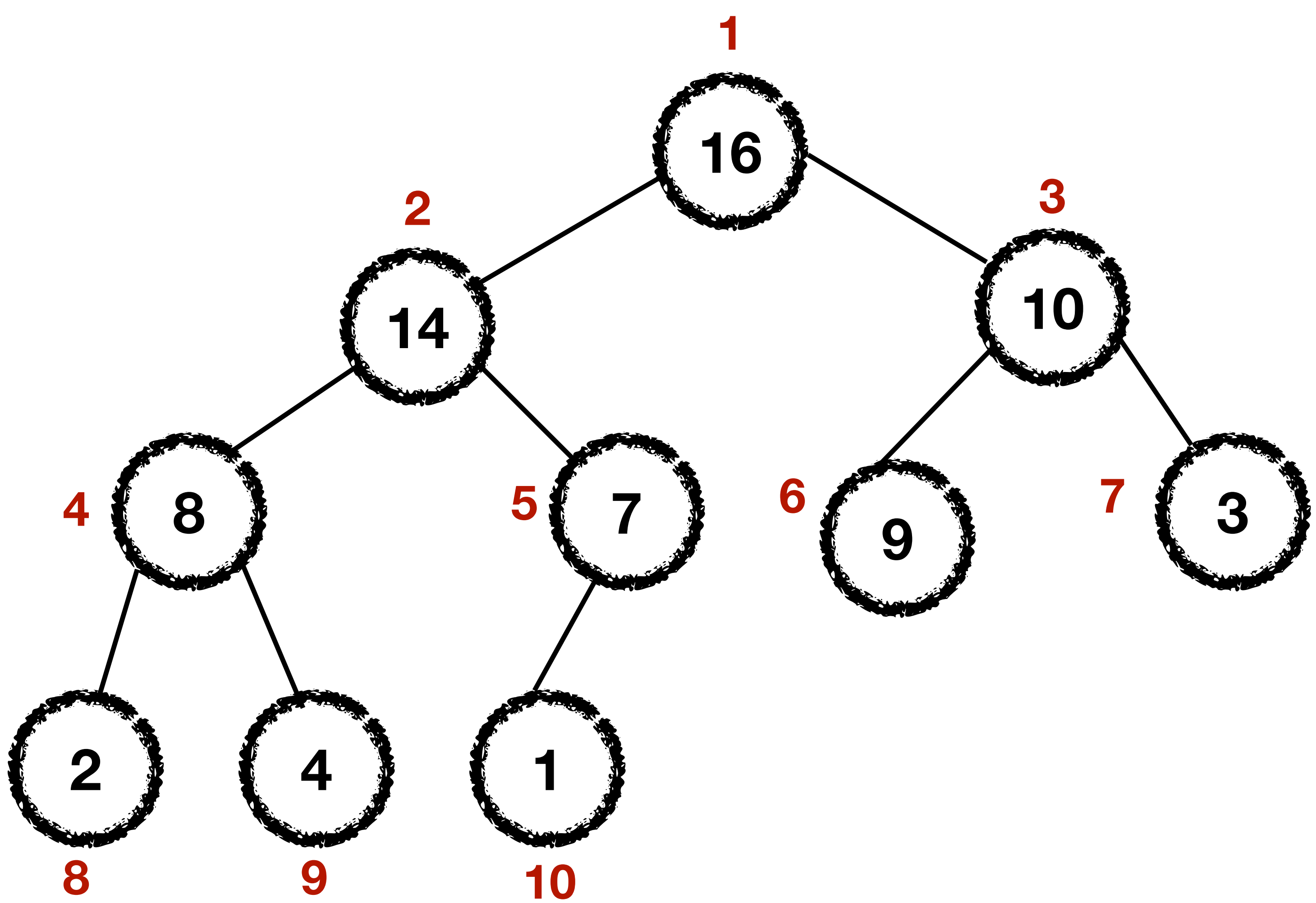
max = $A[1]$
$A[1] = A[A.\text{heap-size}]$
$A.\text{heap-size} = A.\text{heap-size} - 1$
Max-Heapify$(A, 1)$ $\Theta(\lg n)$
return max

Running time?

Max-Heap-Insert(A, v)

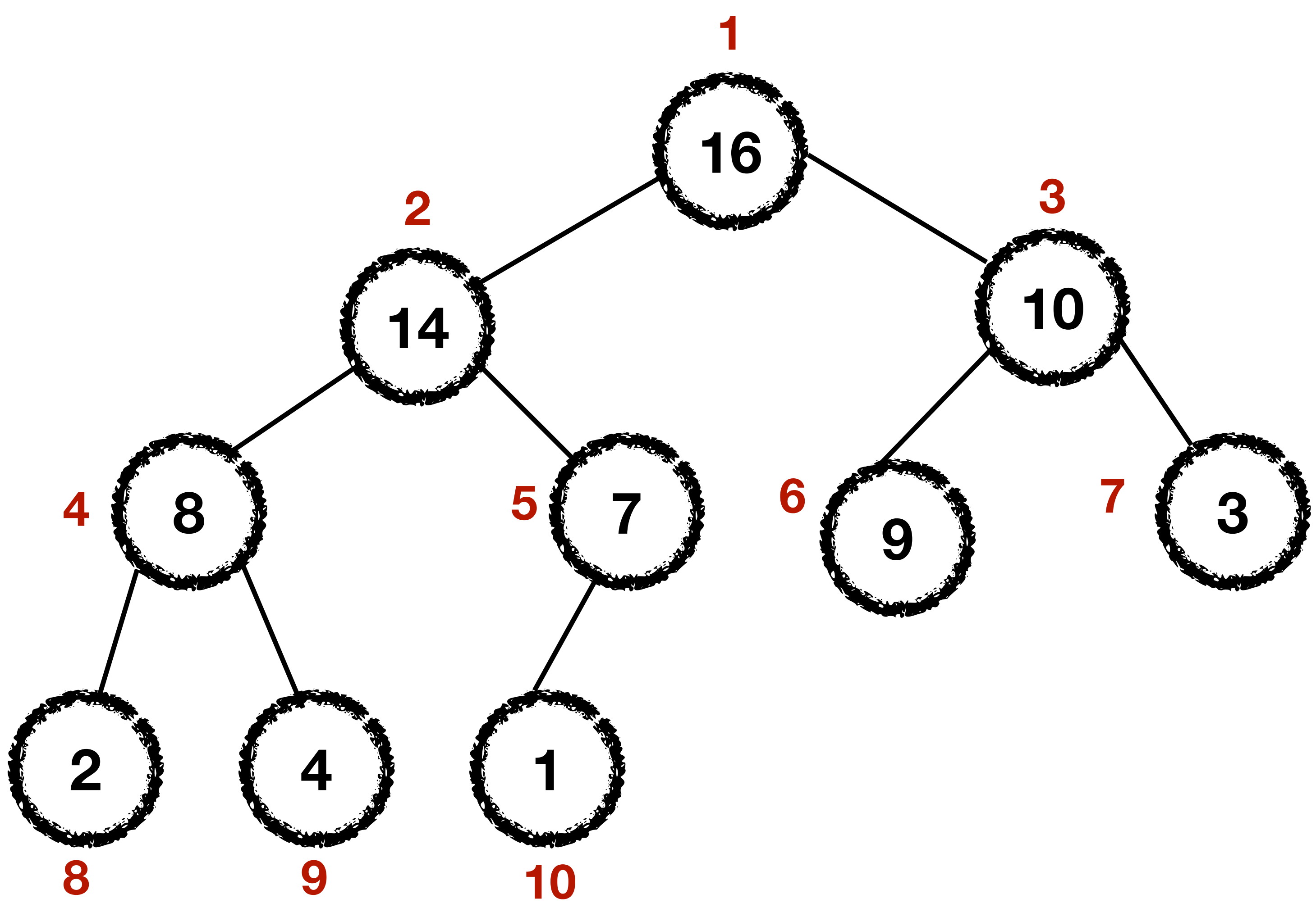
- Max-Heap-Insert(A, v):
  Insert a new element v to the heap.

Max-Heap-Insert $(A, 15)$



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

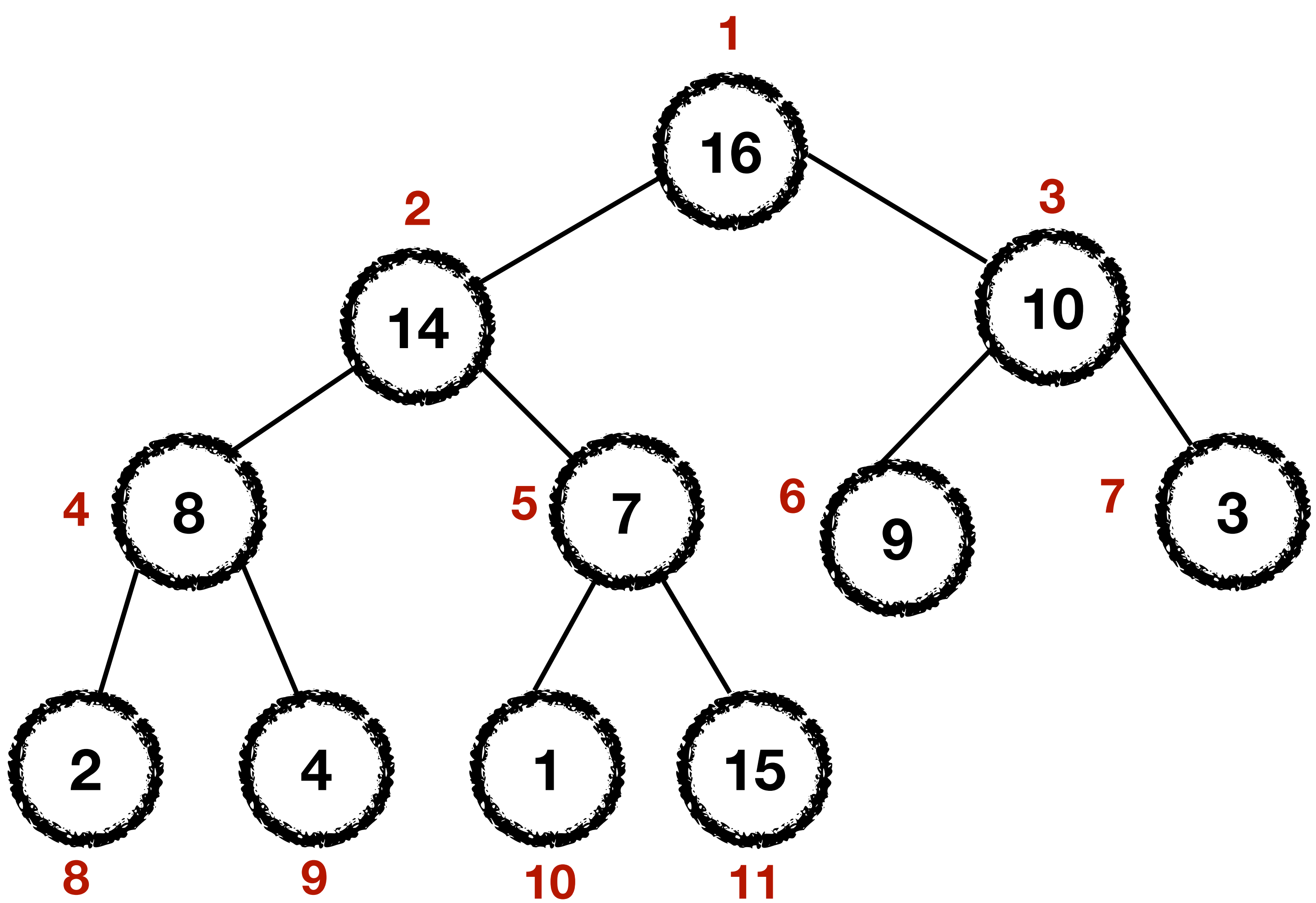
Max-Heap-Insert $(A, 15)$



Where should we add 15?

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

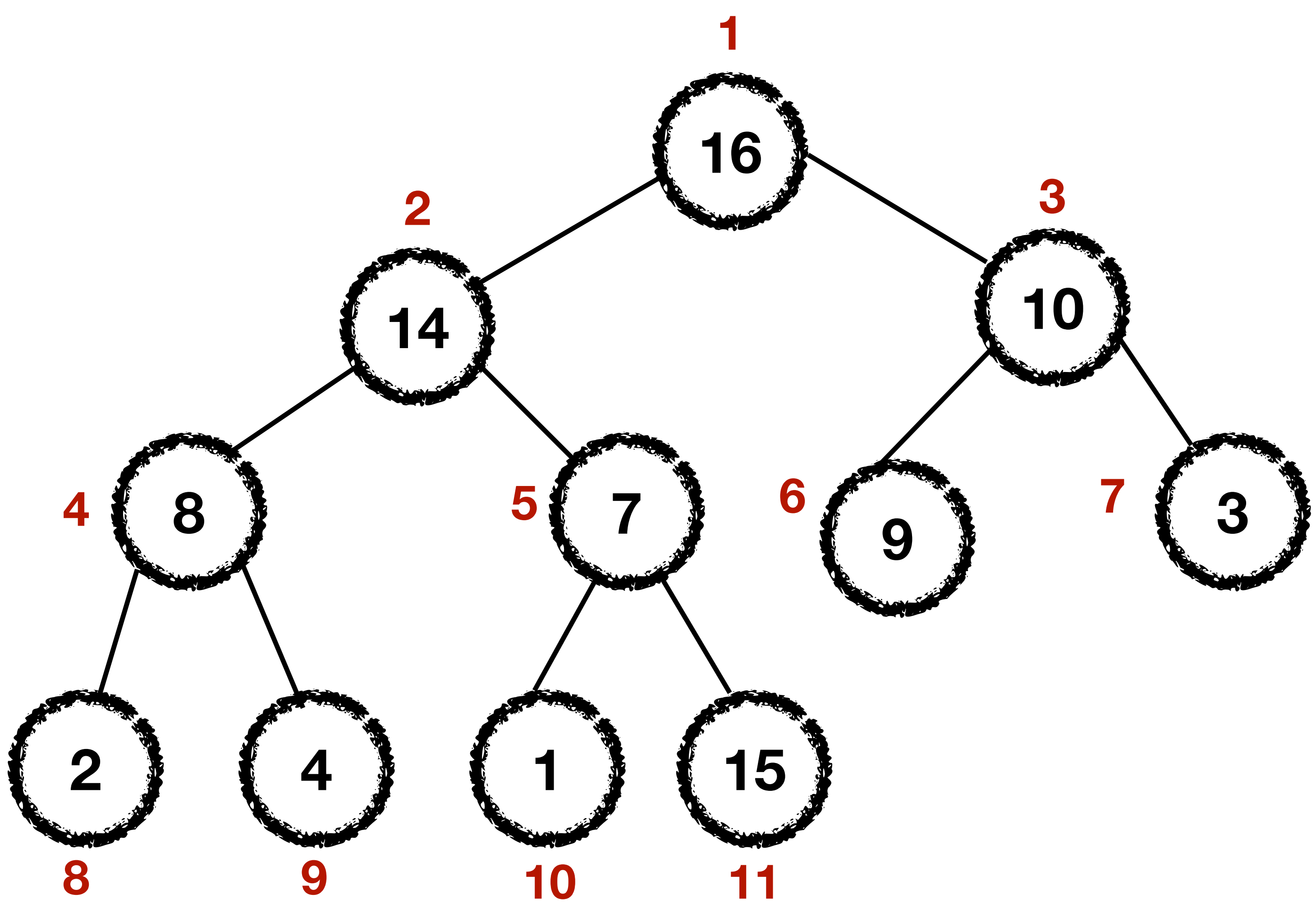
Max-Heap-Insert $(A, 15)$



Where should we add 15?

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

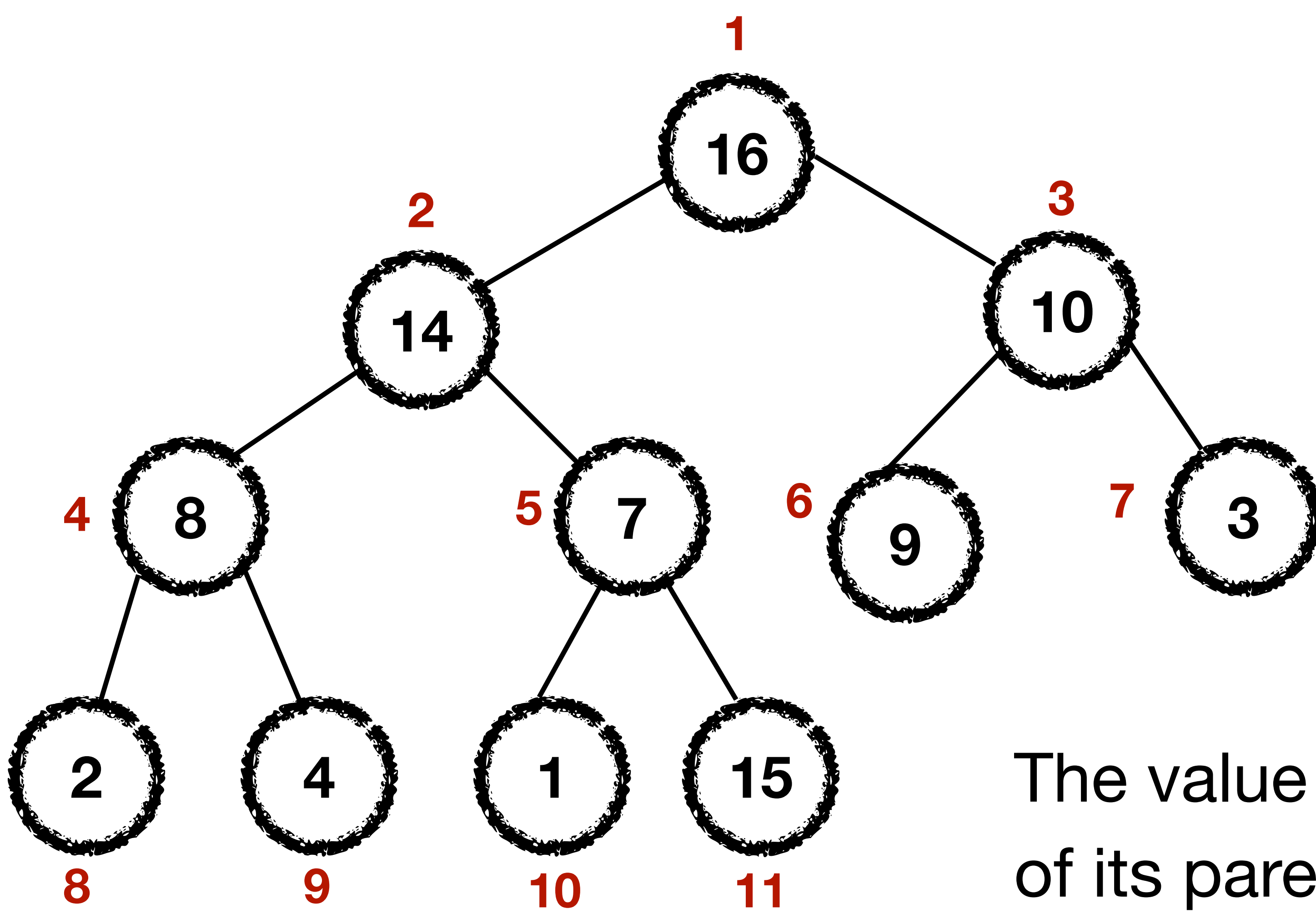
Max-Heap-Insert $(A, 15)$



Where should we add 15?

1	2	3	4	5	6	7	8	9	10	11
16	14	10	8	7	9	3	2	4	1	15

Max-Heap-Insert $(A, 15)$

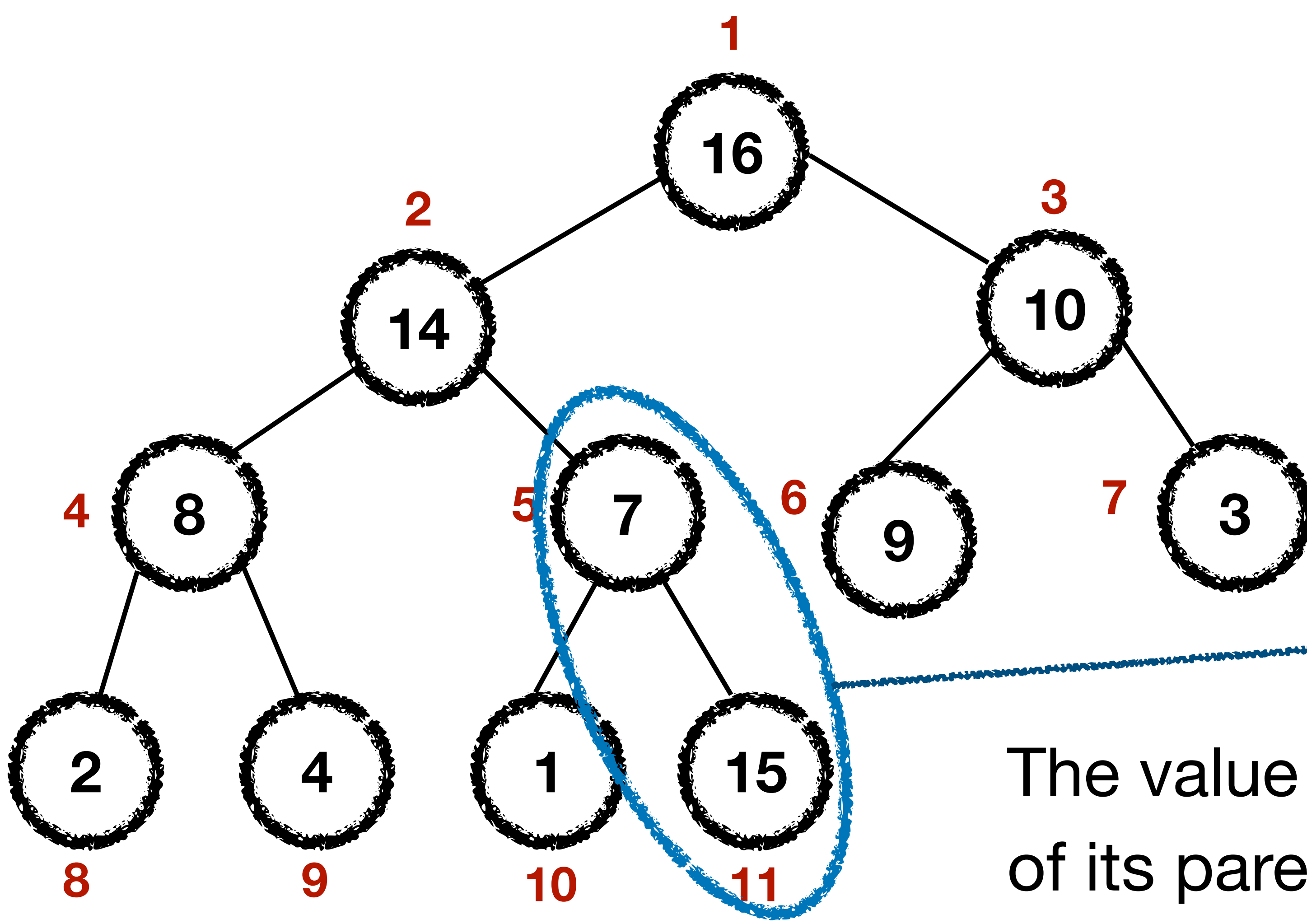


Where should we add 15?

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	11
16	14	10	8	7	9	3	2	4	1	15

Max-Heap-Insert $(A, 15)$



Where should we add 15?

Problem!

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	11
16	14	10	8	7	9	3	2	4	1	15

Fixing the problem

Fixing the problem

- How do we fix a tree that is “almost” a heap back to being a heap?

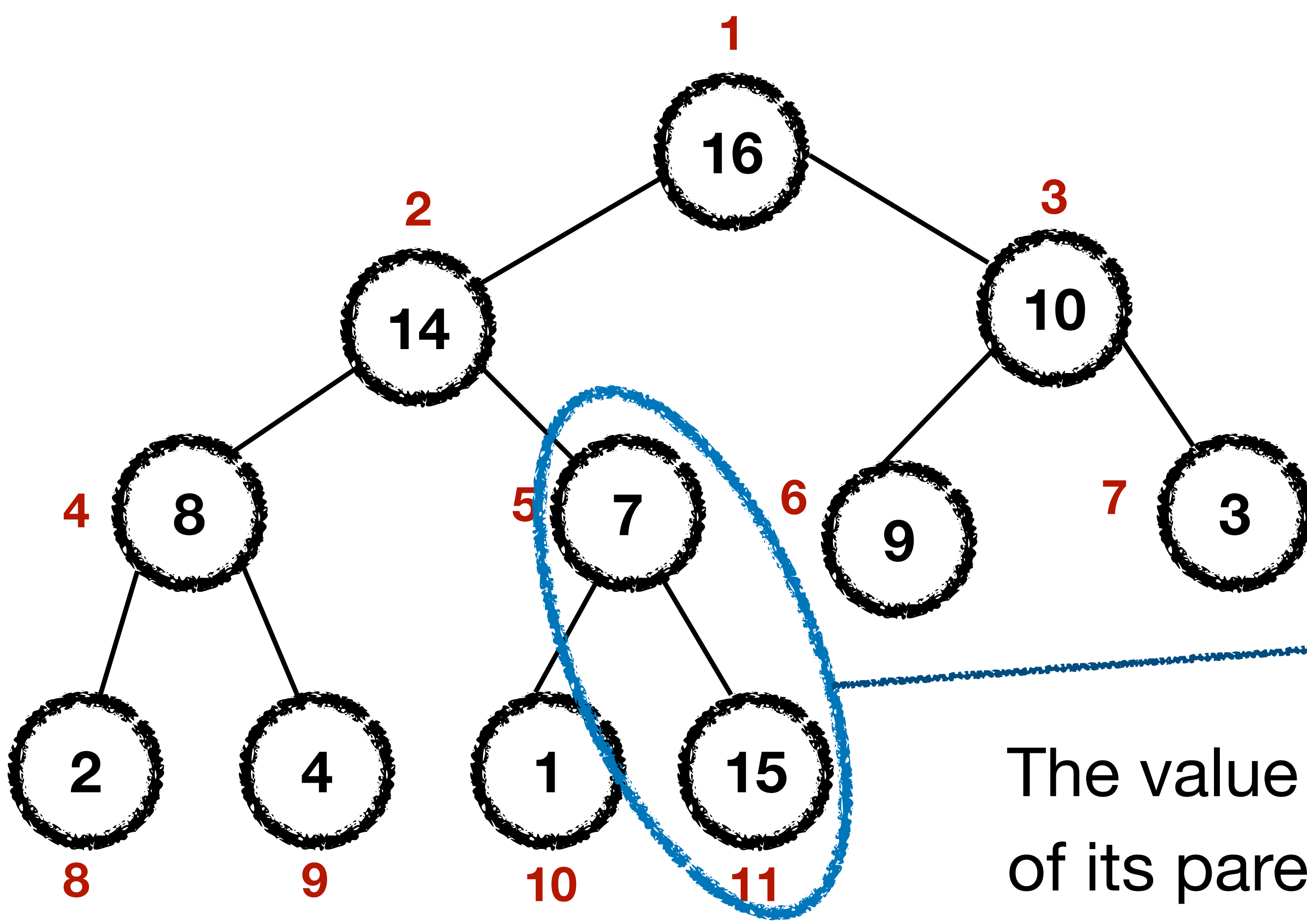
Fixing the problem

- How do we fix a tree that is “almost” a heap back to being 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-Heap-Insert $(A, 15)$



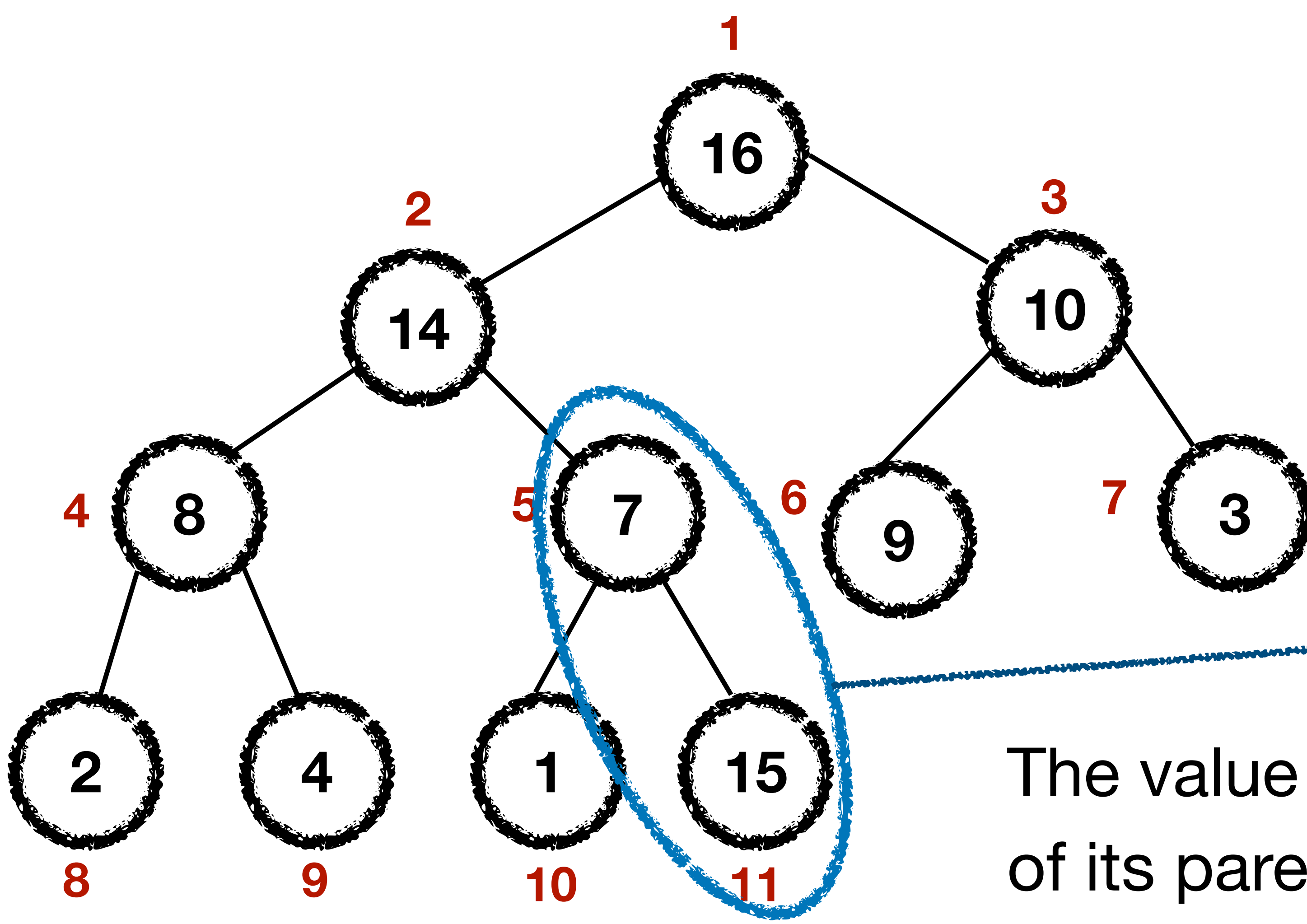
Can we use Max-Heapify here?

Problem!

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	11
16	14	10	8	7	9	3	2	4	1	15

Max-Heap-Insert $(A, 15)$



Can we use Max-Heapify here?

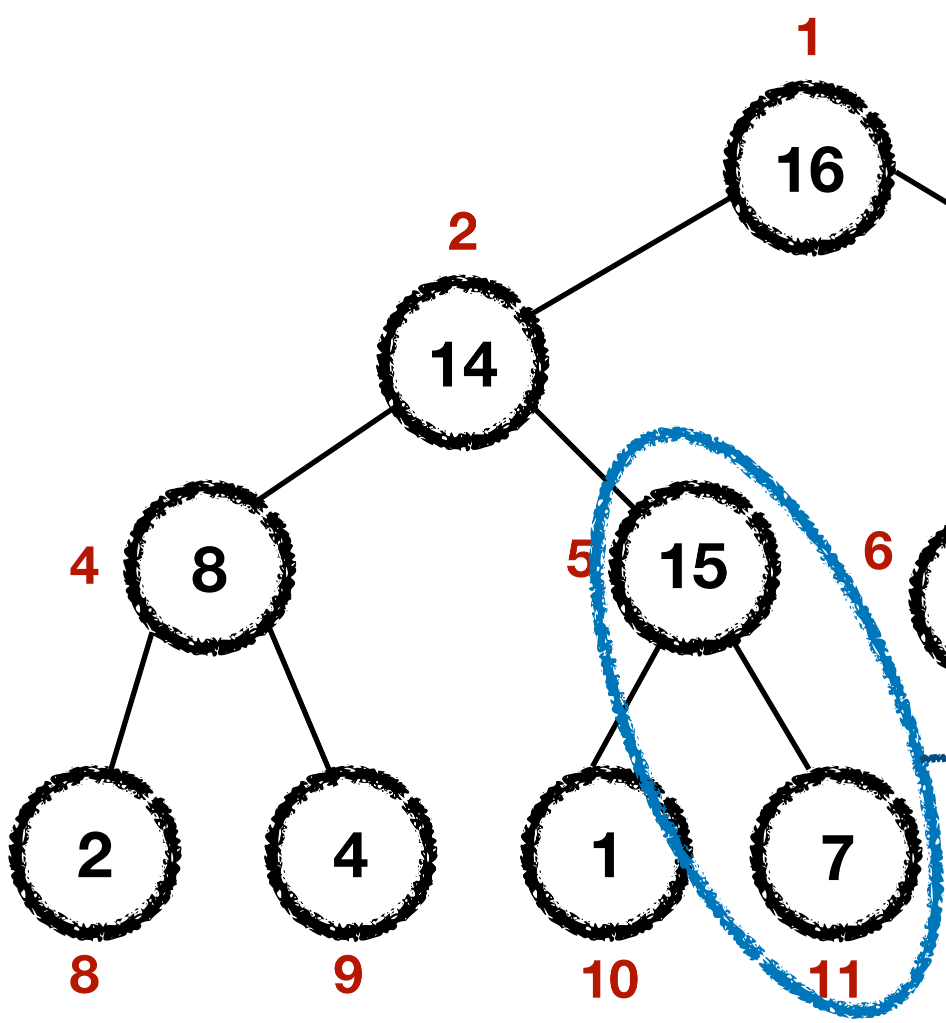
The value o

of its parent

```
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	11
16	14	10	8	7	9	3	2	4	1	15

Max-Heap-Insert $(A, 15)$



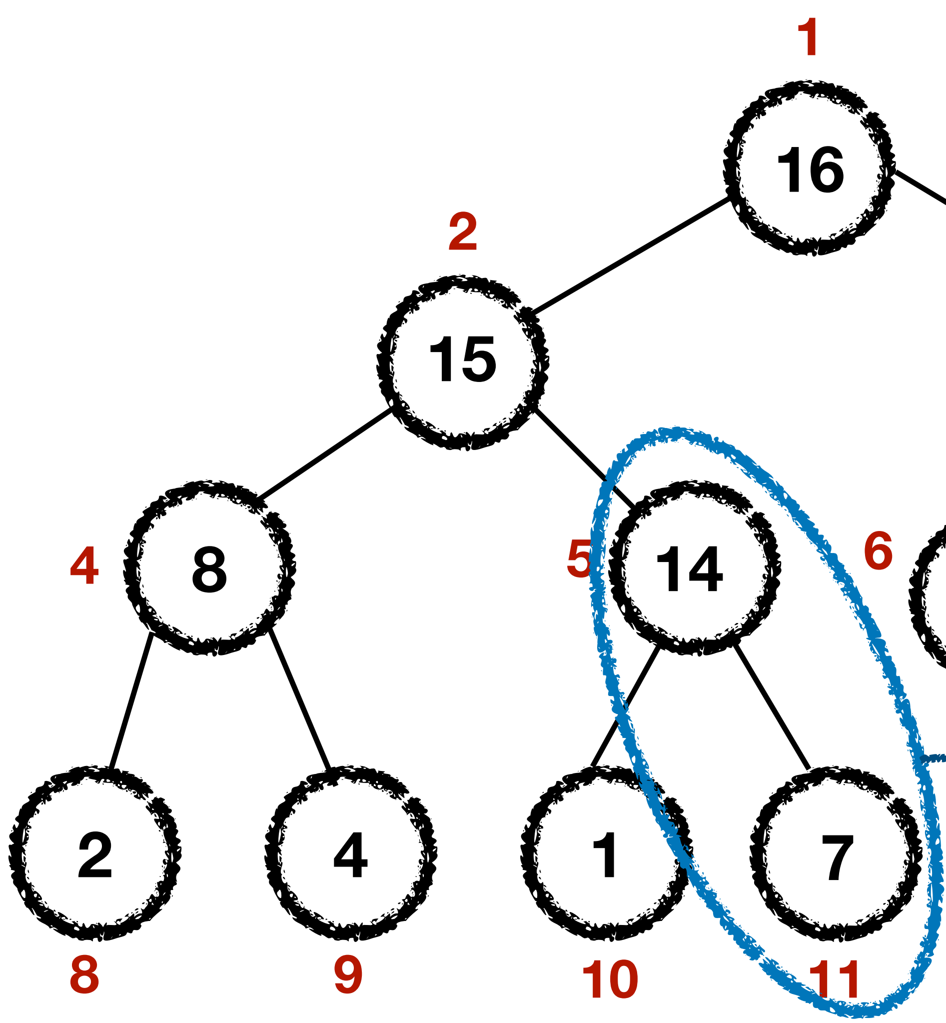
Can we use Max-Heapify here?

```
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)
```

The value o
of its parent

1	2	3	4	5	6	7	8	9	10	11
16	14	10	8	7	9	3	2	4	1	15

Max-Heap-Insert $(A, 15)$



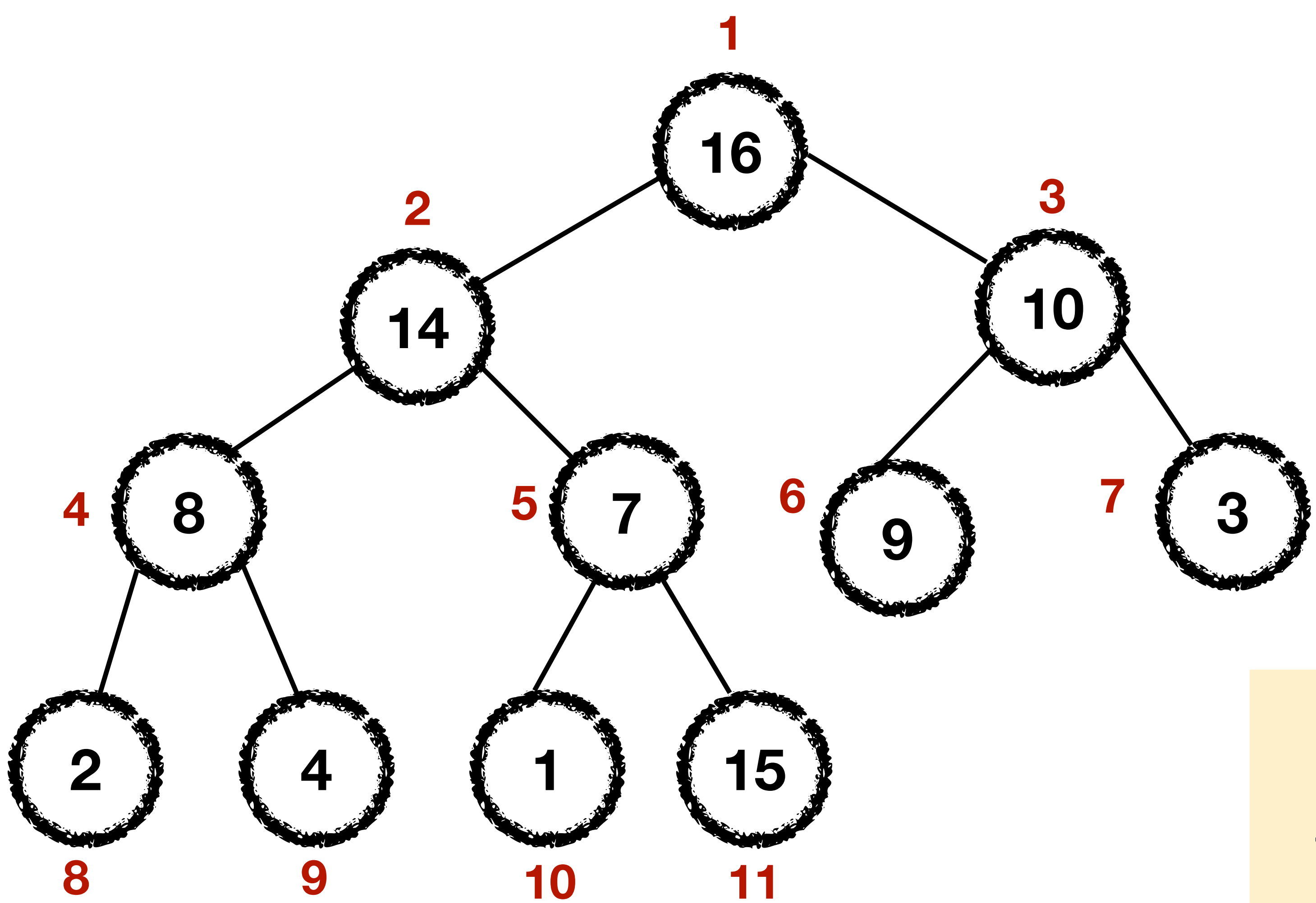
Can we use Max-Heapify here?

```
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)
```

The value o
of its parent

1	2	3	4	5	6	7	8	9	10	11
16	14	10	8	7	9	3	2	4	1	15

Max-Heapify-Up$(A, 11)$:



Max-Heapify-Up (A, i)

if $i > 1$ then

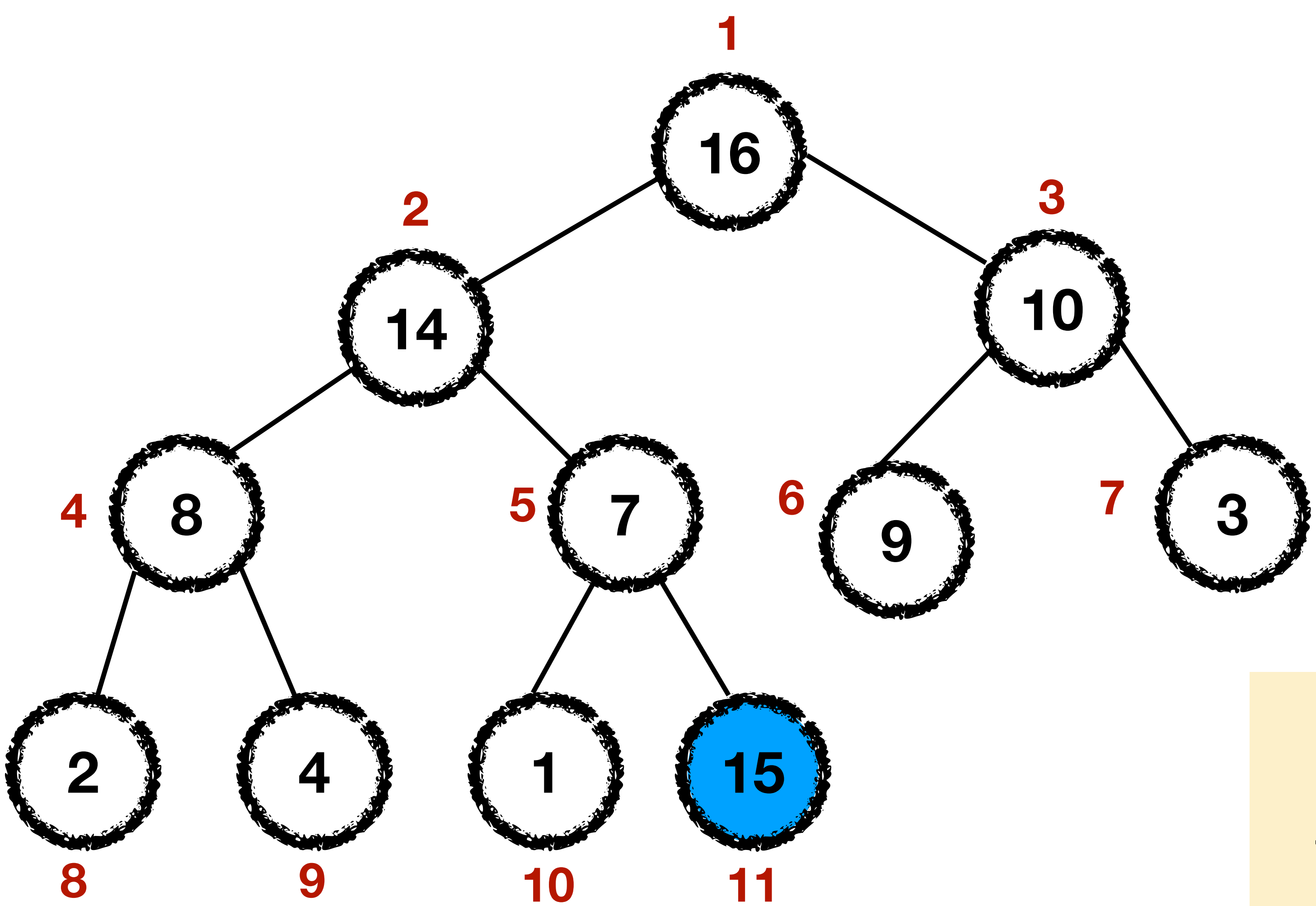
$j =$ Parent(i)

if $A[i] > A[j]$ then

exchange $A[i]$ with $A[j]$

Max-Heapify-Up (A, j)

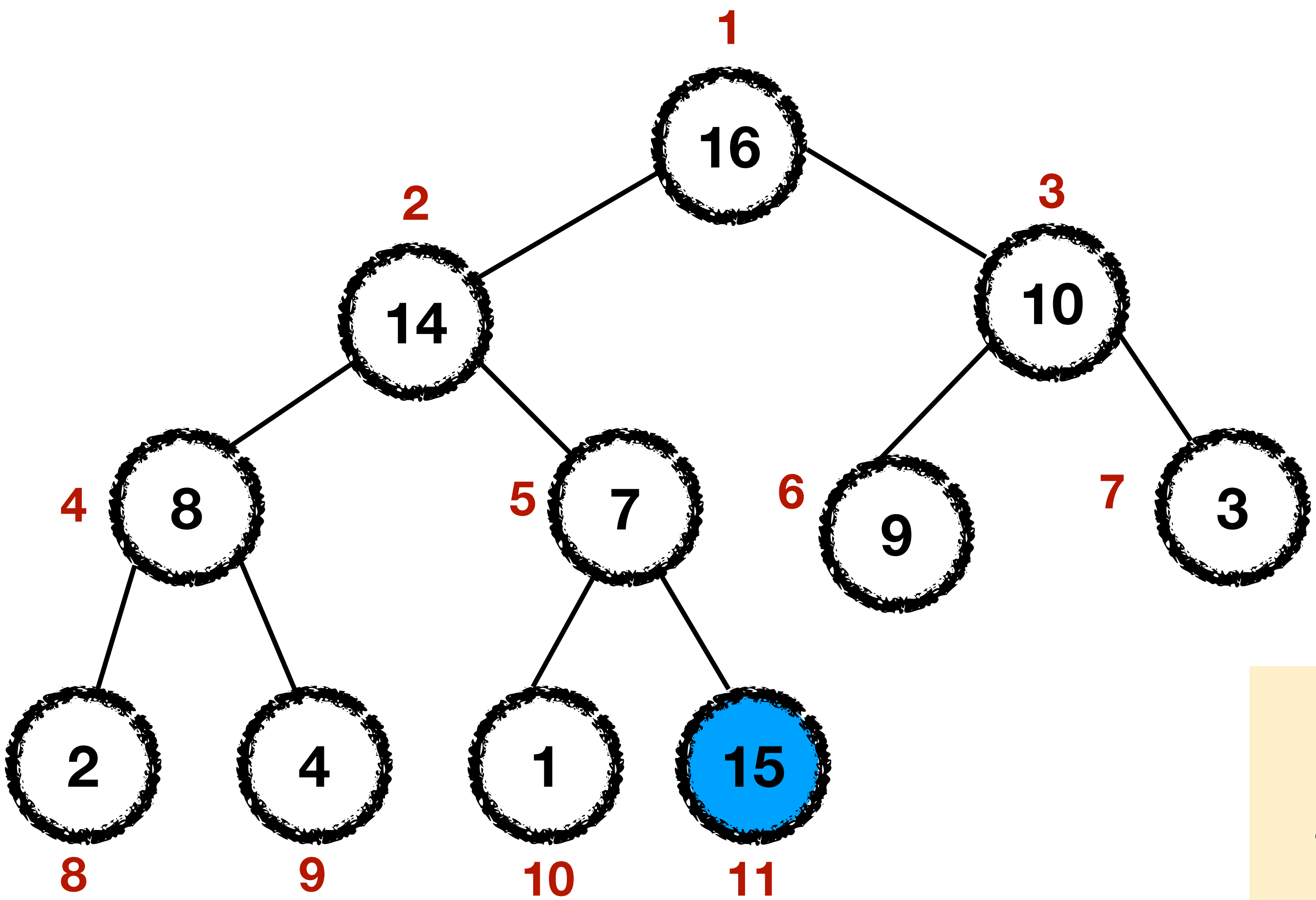
Max-Heapify-Up$(A, 11)$:



Max-Heapify-Up (A, i)

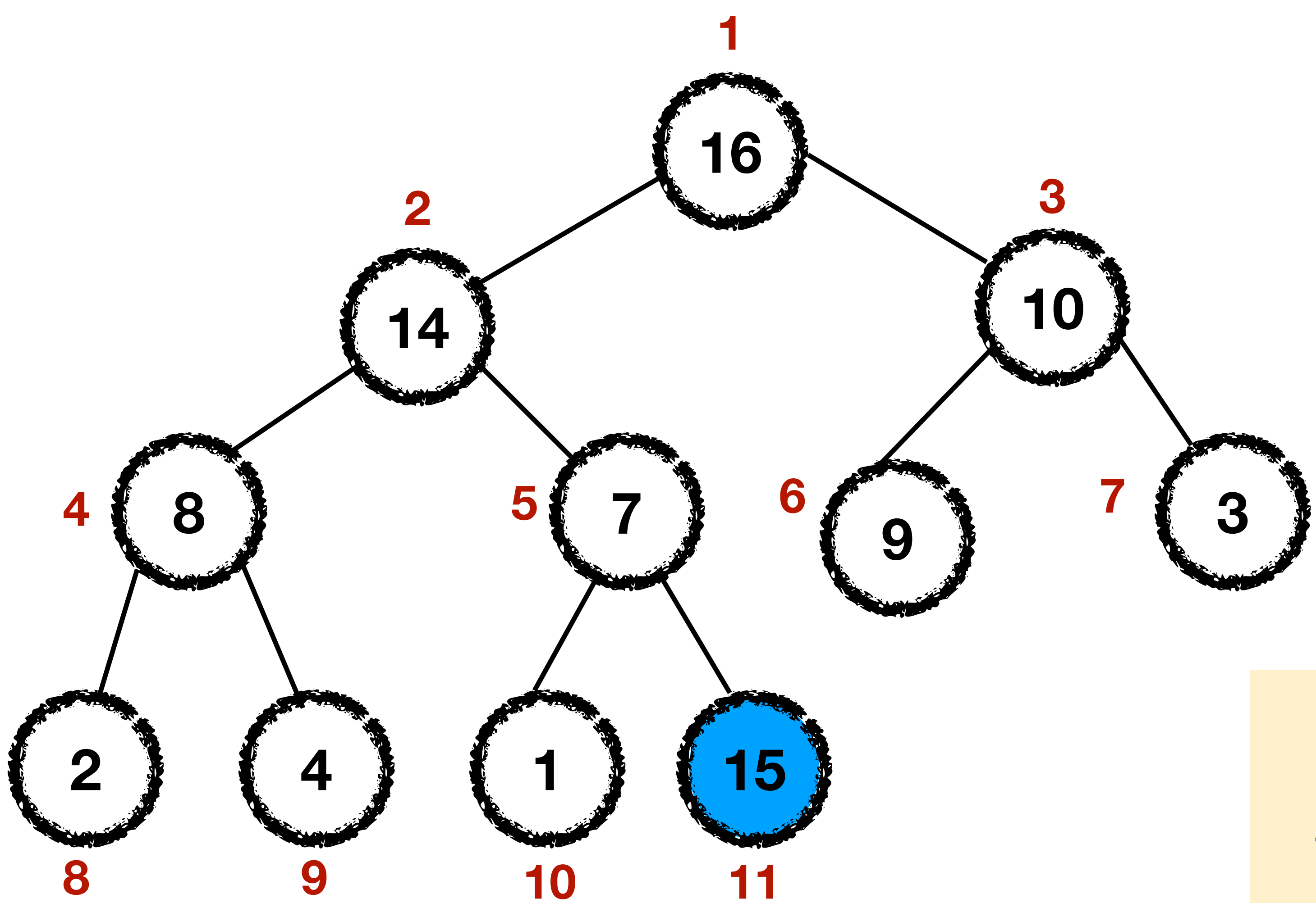
```
if i > 1 then
  j = Parent(i)
  if A[i] > A[j] then
    exchange A[i] with A[j]
    Max-Heapify-Up (A, j)
```

Max-Heapify-Up$(A, 11)$:




```
Max-Heapify-Up (A, i)
    if i > 1 then    true
  j = Parent(i)
  if A[i] > A[j] then
      exchange A[i] with A[j]
      Max-Heapify-Up (A, j)
```

Max-Heapify-Up$(A, 11)$:



Max-Heapify-Up (A, i)

if $i > 1$ then true

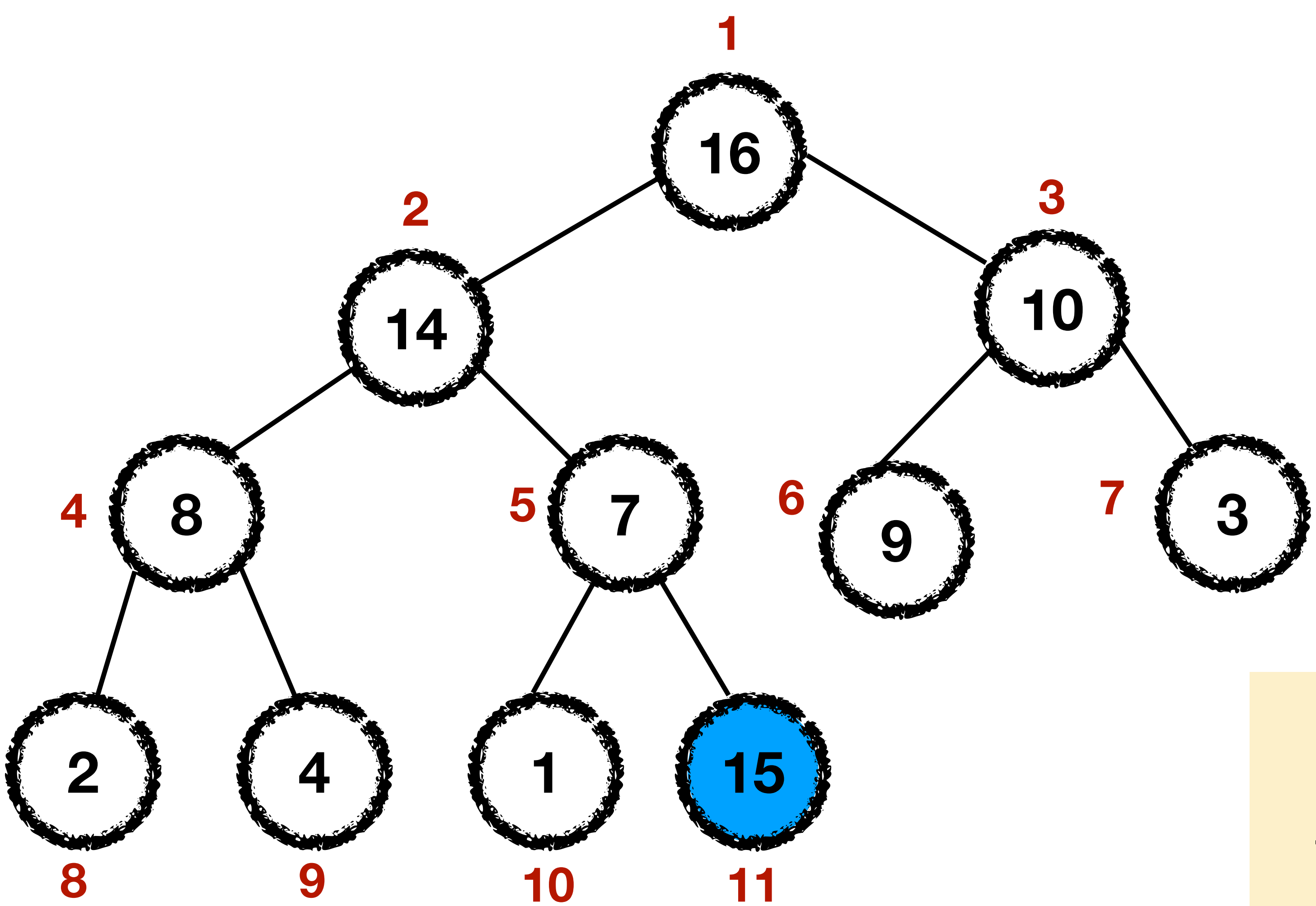
$j =$ Parent(i)

if $A[i] > A[j]$ then

exchange $A[i]$ with $A[j]$

Max-Heapify-Up (A, j)

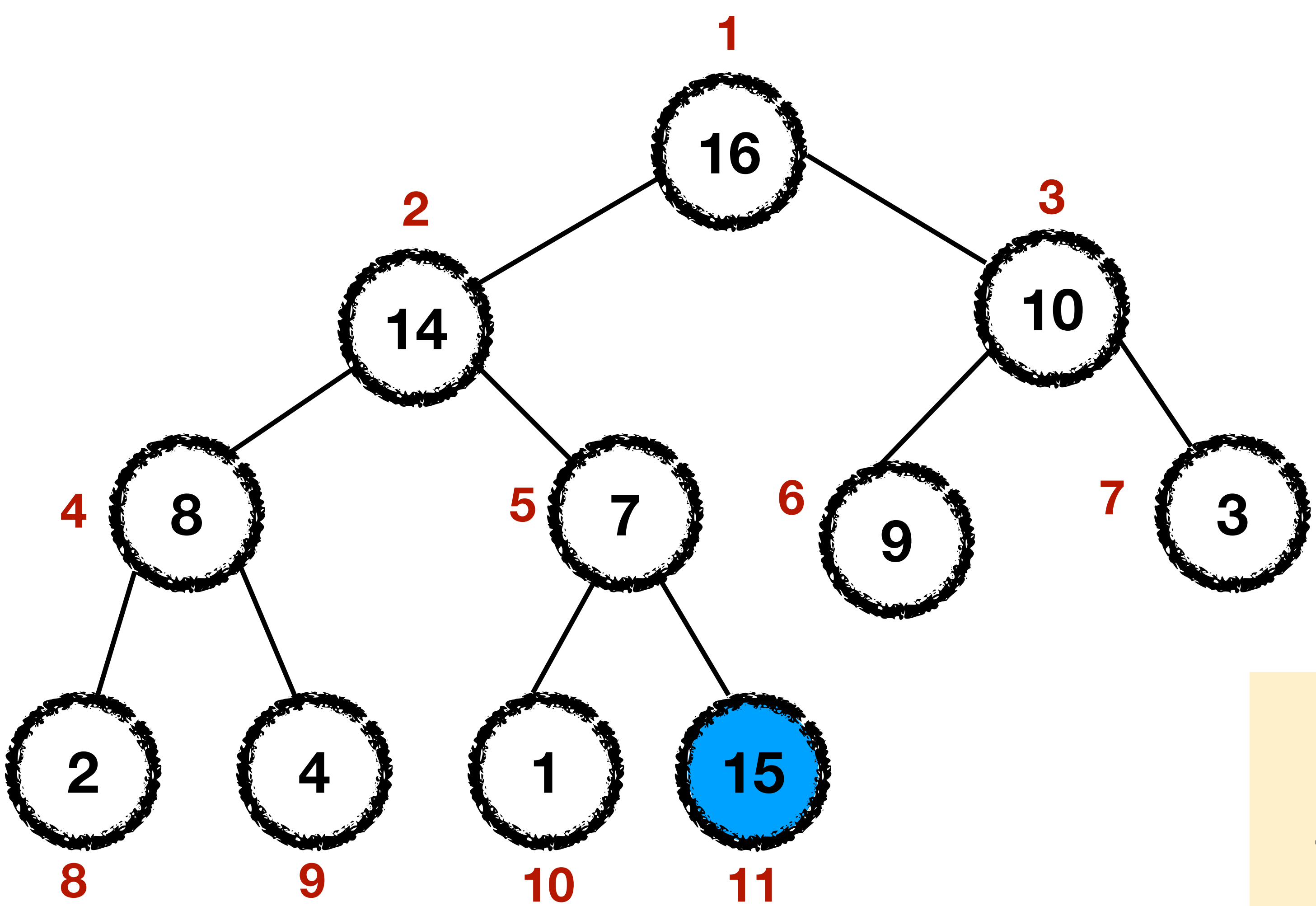
Max-Heapify-Up$(A, 11)$:



Max-Heapify-Up (A, i)

```
if i > 1 then                      true
  j = Parent(i)
  if A[i] > A[j] then              true
      exchange A[i] with A[j]
      Max-Heapify-Up (A, j)
```

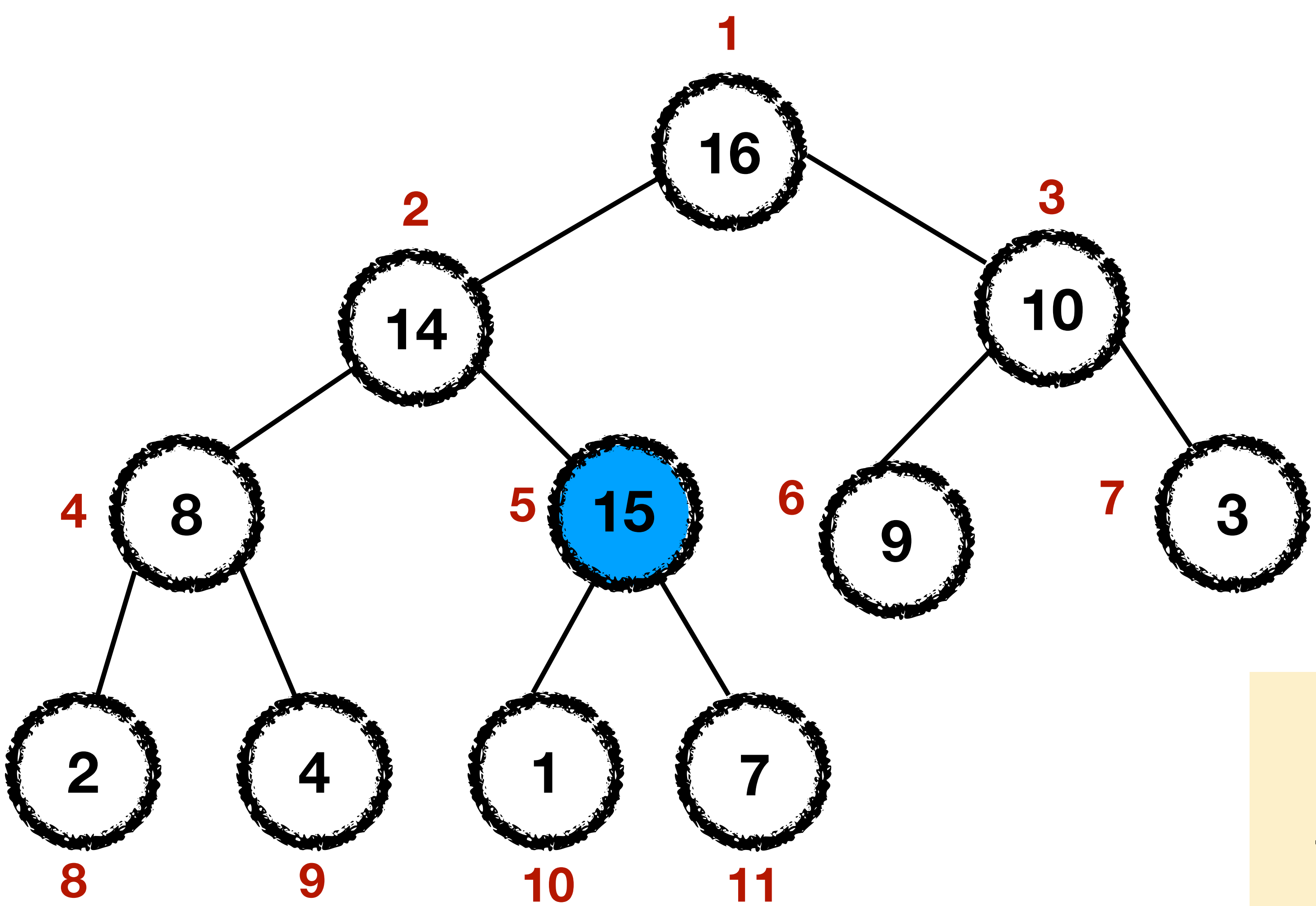
Max-Heapify-Up$(A, 11)$:



Max-Heapify-Up (A, i)

```
if i > 1 then          true
    j = Parent(i)
    if A[i] > A[j] then    true
        exchange A[i] with A[j]
        Max-Heapify-Up (A, j)
```

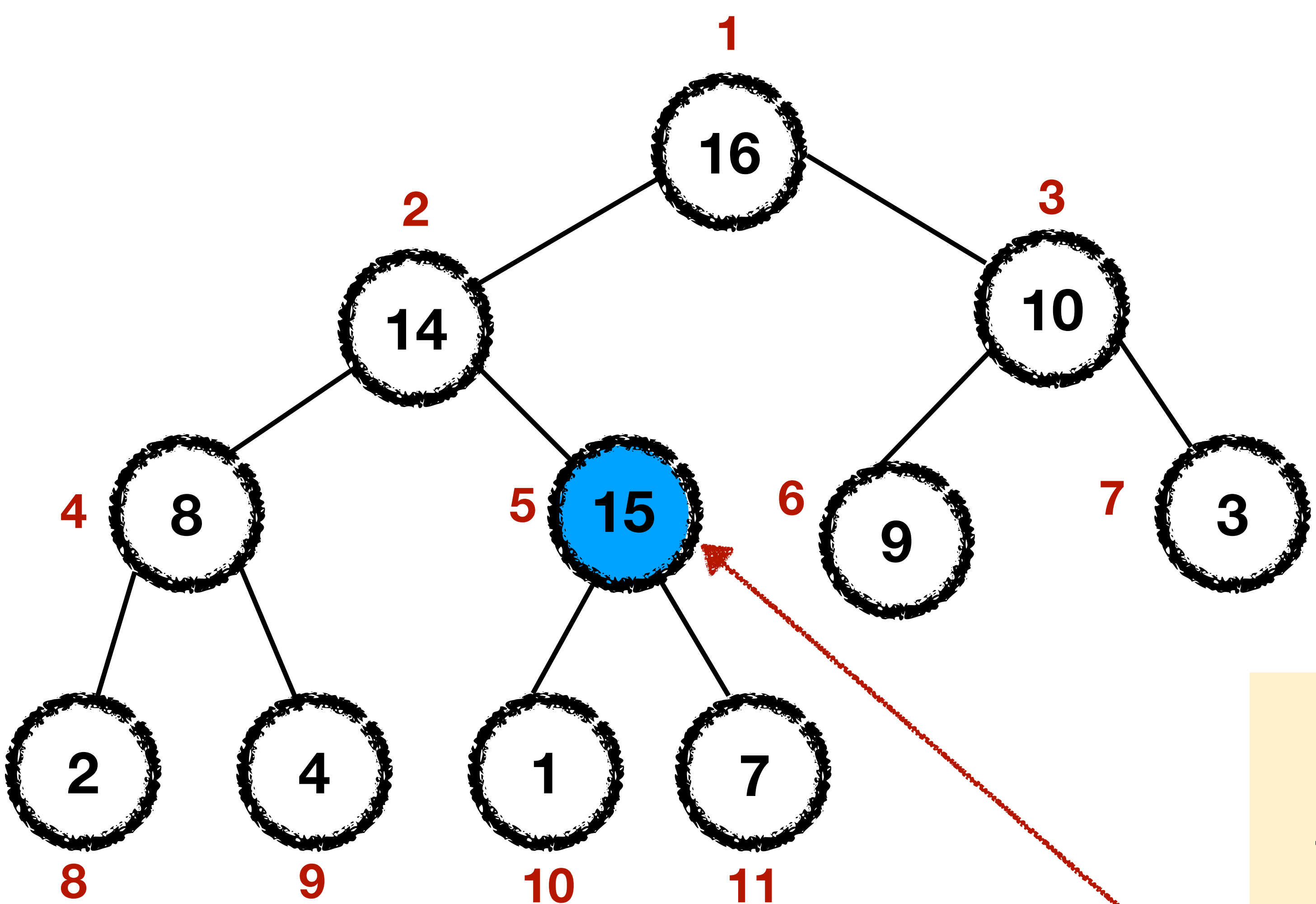
Max-Heapify-Up$(A, 11)$:



Max-Heapify-Up (A, i)

if $i > 1$ then true
$j =$ Parent(i)
if $A[i] > A[j]$ then true
exchange $A[i]$ with $A[j]$
Max-Heapify-Up (A, j)

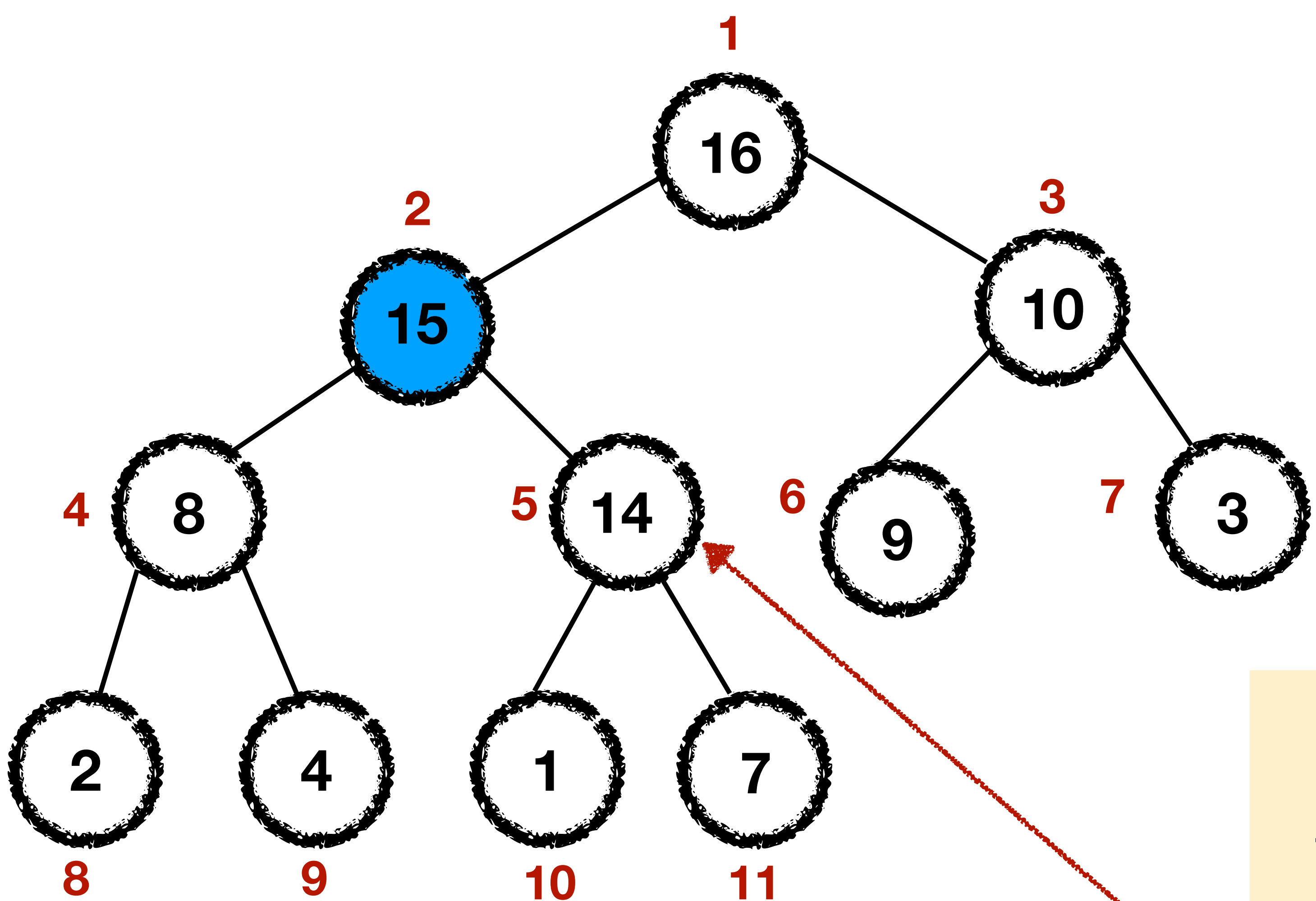
Max-Heapify-Up$(A, 11)$:



Max-Heapify-Up (A, i)

```
if i > 1 then   true
  j = Parent(i)
  if A[i] > A[j] then   true
    exchange A[i] with A[j]
    Max-Heapify-Up (A, j)
```

Max-Heapify-Up$(A, 11)$:



Max-Heapify-Up (A, i)

if $i > 1$ then true

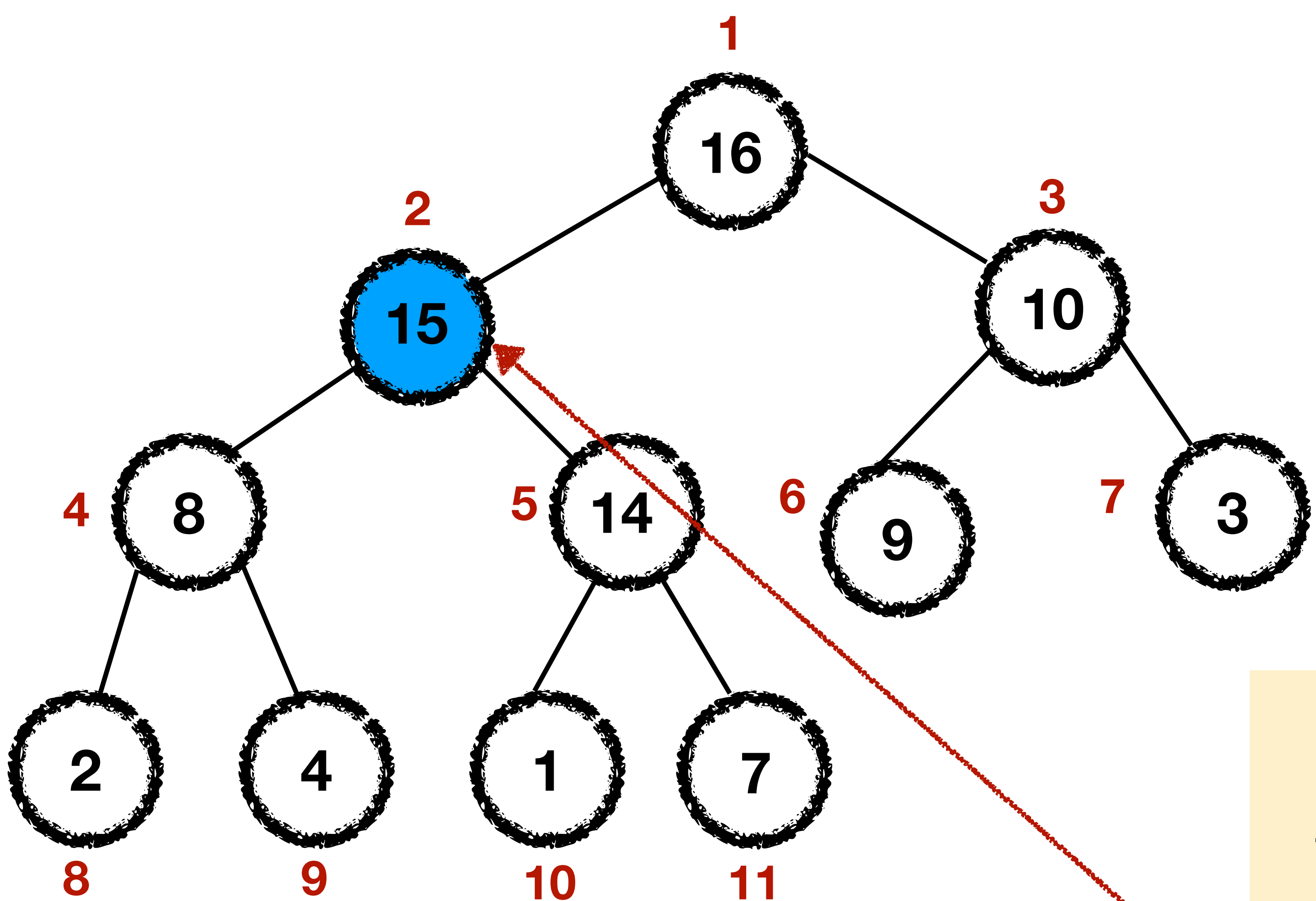
$j =$ Parent(i)

if $A[i] > A[j]$ then true

exchange $A[i]$ with $A[j]$

Max-Heapify-Up (A, j)

Max-Heapify-Up(A,11):



Max-Heapify-Up (A, i)

if $i > 1$ then true

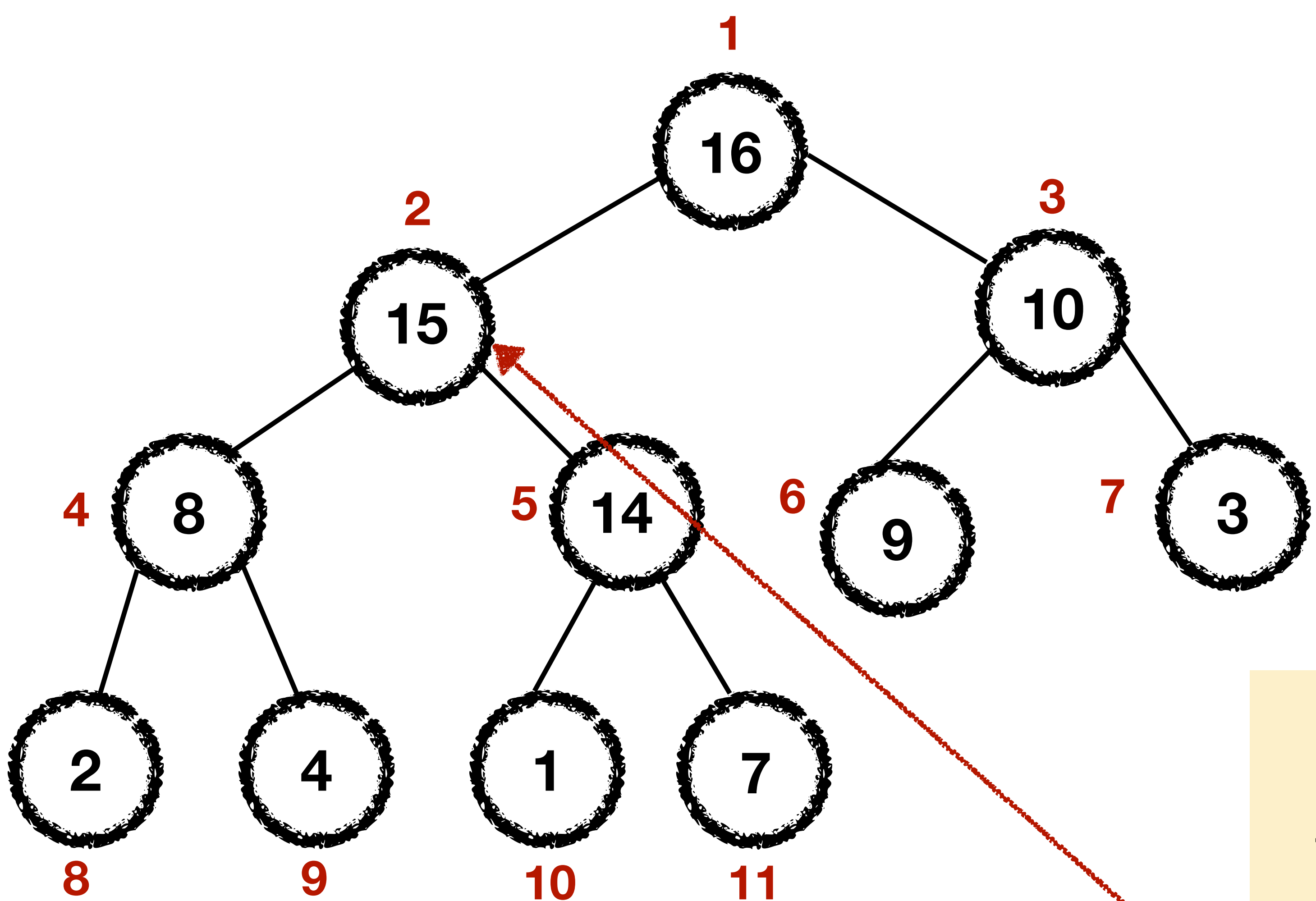
$j =$ Parent(i)

if $A[i] > A[j]$ then true

exchange $A[i]$ with $A[j]$

Max-Heapify-Up (A, j)

Max-Heapify-Up$(A, 11)$:



Max-Heapify-Up (A, i)

```
if i > 1 then        true
  j = Parent(i)
  if A[i] > A[j] then        true
    exchange A[i] with A[j]
    Max-Heapify-Up (A, j)
```

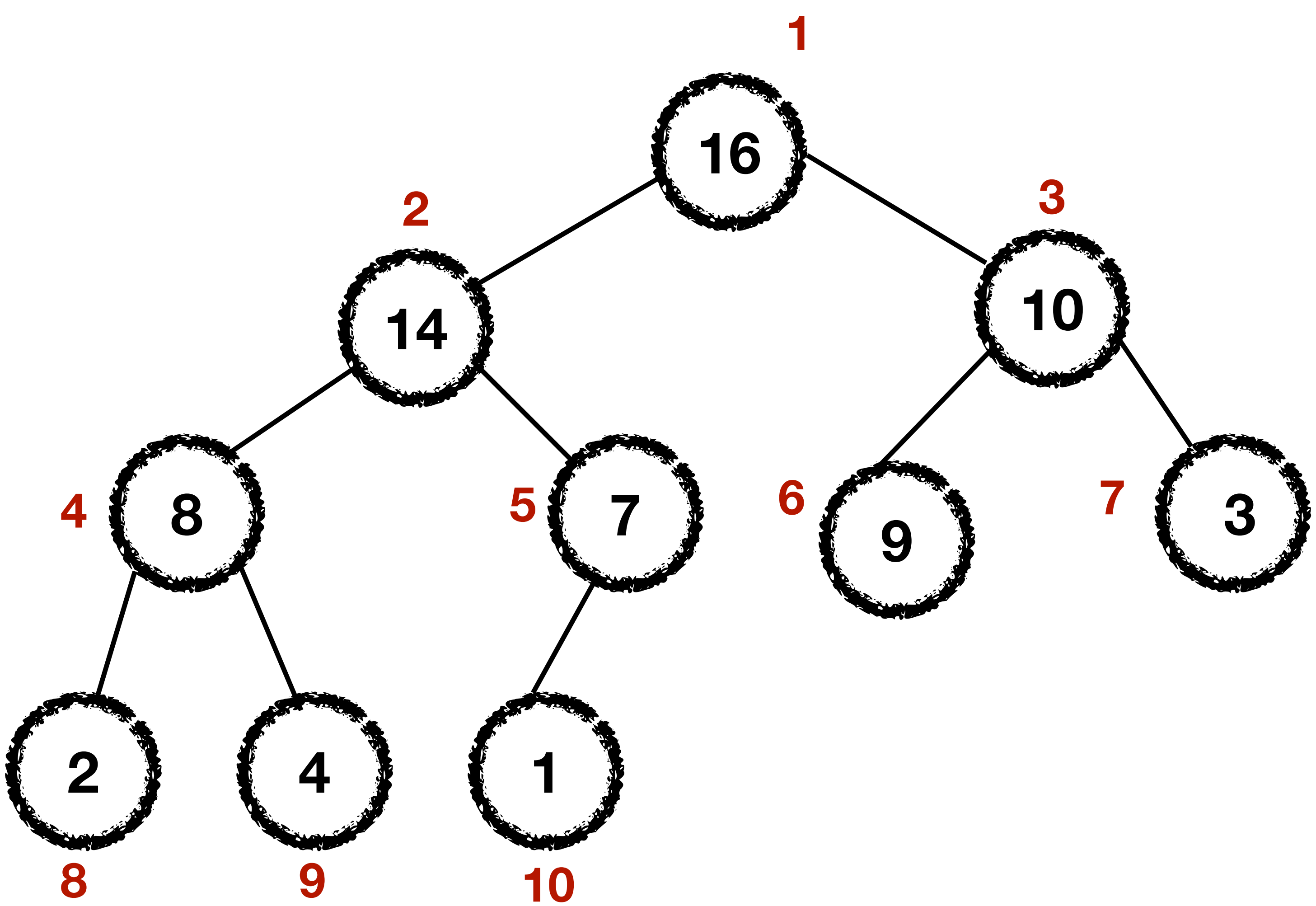
Max-Heap-Insert(A, v)

- Max-Heap-Insert(A, v):
  Insert a new element v to the heap.

Max-Heap-Insert(A, v)

A.heap-size $= A$.heap-size $+ 1$
$A[A$.heap-size$] = v$
$i = A$.heap-size
Max-Heapify-Up (A, i)

Max-Heap-Insert $(A, 15)$

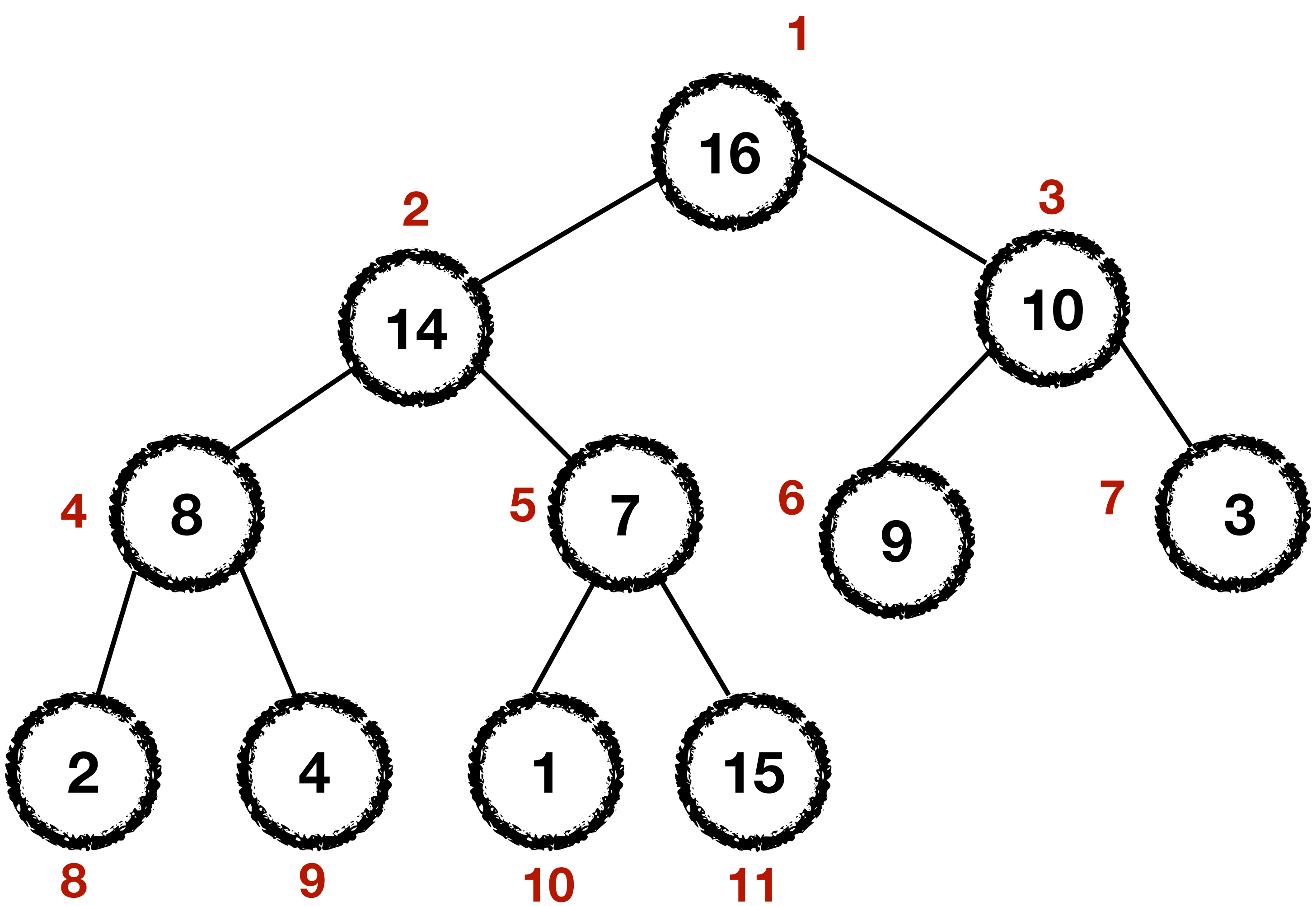


Max-Heap-Insert(A, v)

```
A.heap-size = A.heap-size + 1
A[A.heap-size] = v
i = A.heap-size
Max-Heapify-Up (A, i)
```

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

Max-Heap-Insert $(A, 15)$

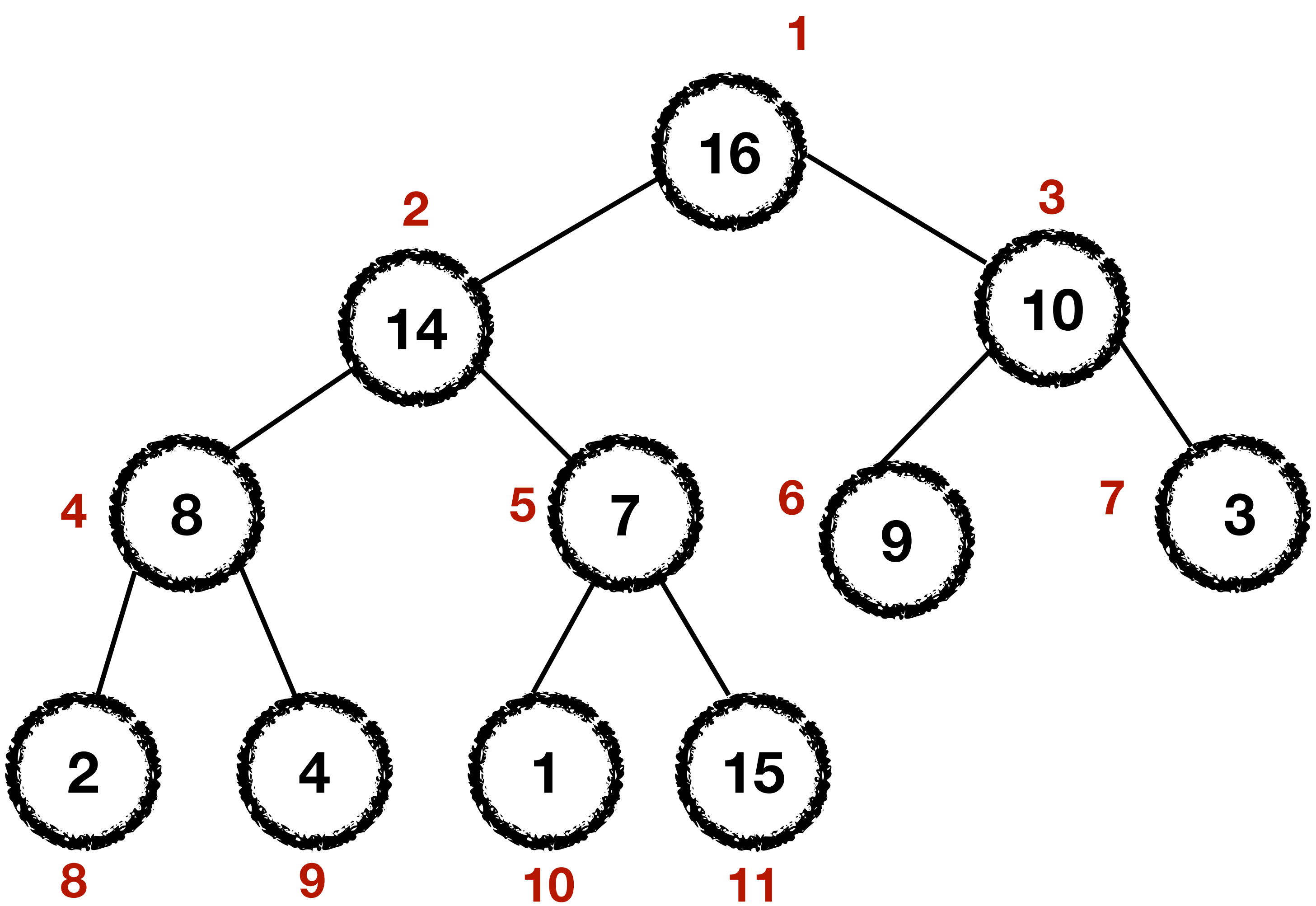


Max-Heap-Insert(A, v)

A.heap-size $= A$.heap-size $+ 1$
$A[A$.heap-size$] = v$
$i = A$.heap-size
Max-Heapify-Up (A, i)

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

Max-Heap-Insert $(A, 15)$

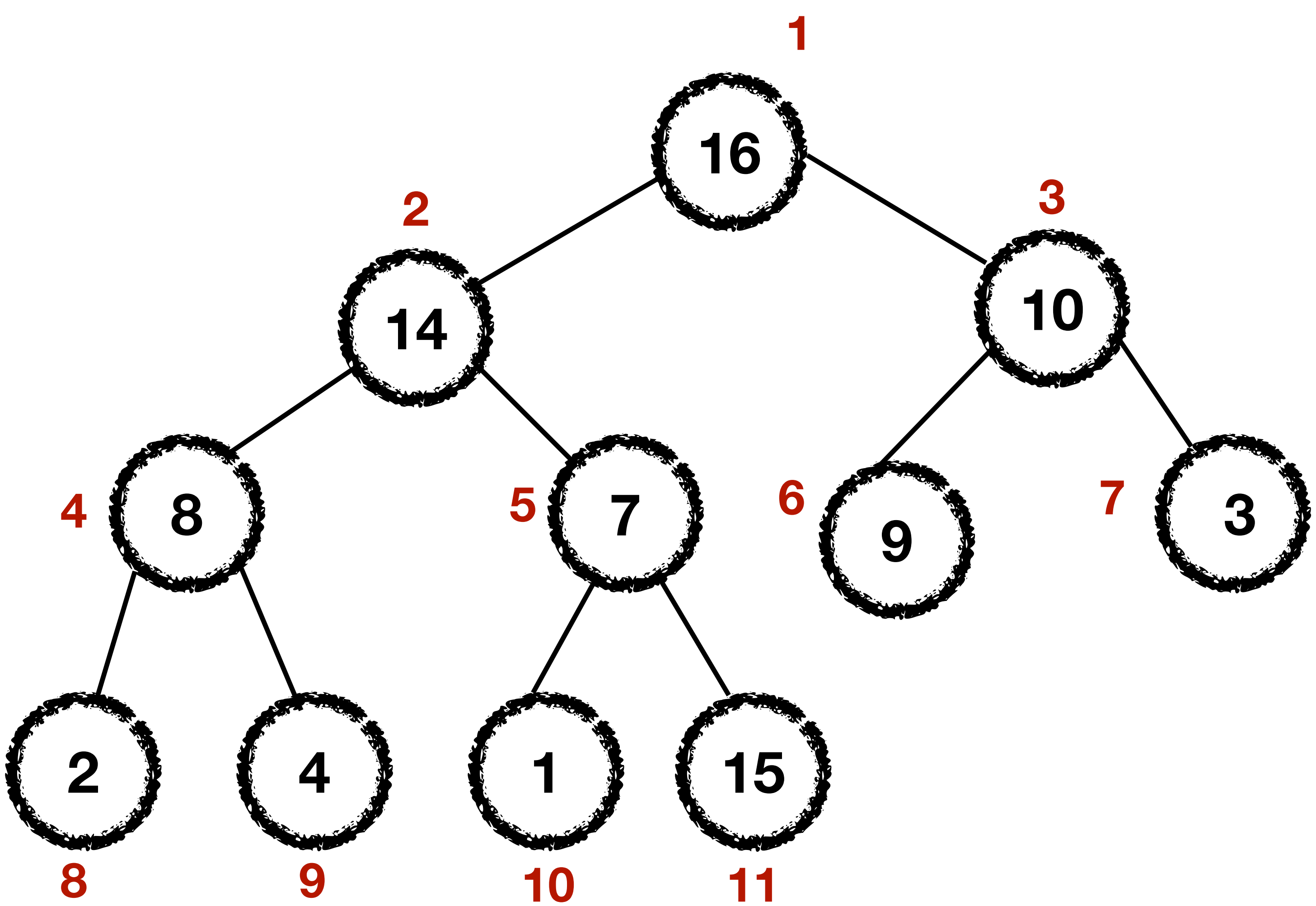


Max-Heap-Insert(A, v)

A.heap-size $= A$.heap-size $+ 1$
$A[A$.heap-size$] = v$
$i = A$.heap-size
Max-Heapify-Up (A, i)

1	2	3	4	5	6	7	8	9	10	11
16	14	10	8	7	9	3	2	4	1	15

Max-Heap-Insert $(A, 15)$

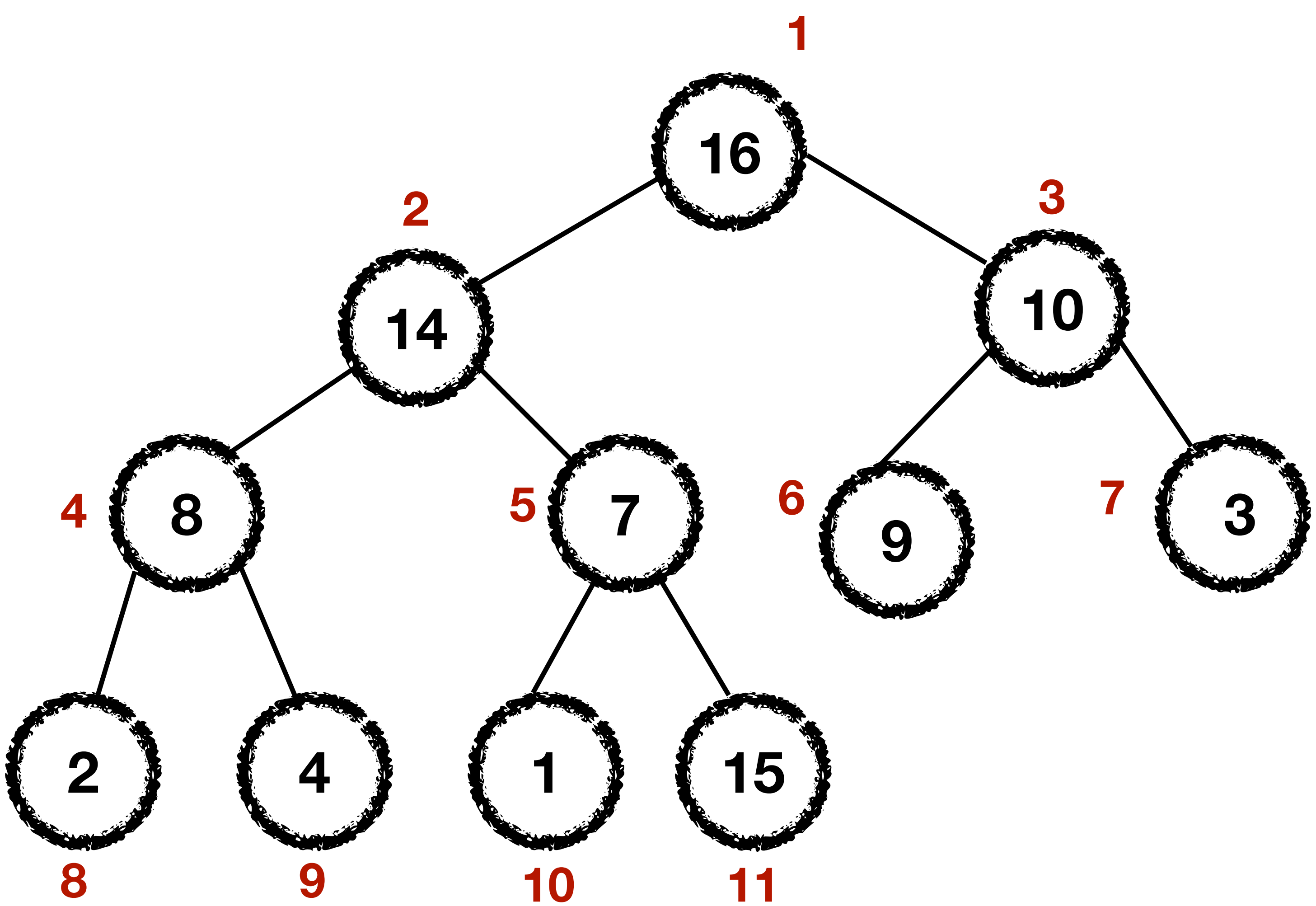


Max-Heap-Insert(A, v)

```
11 A.heap-size = A.heap-size + 1
   A[A.heap-size] = v
   i = A.heap-size
   Max-Heapify-Up (A, i)
```

1	2	3	4	5	6	7	8	9	10	11
16	14	10	8	7	9	3	2	4	1	15

Max-Heap-Insert $(A, 15)$

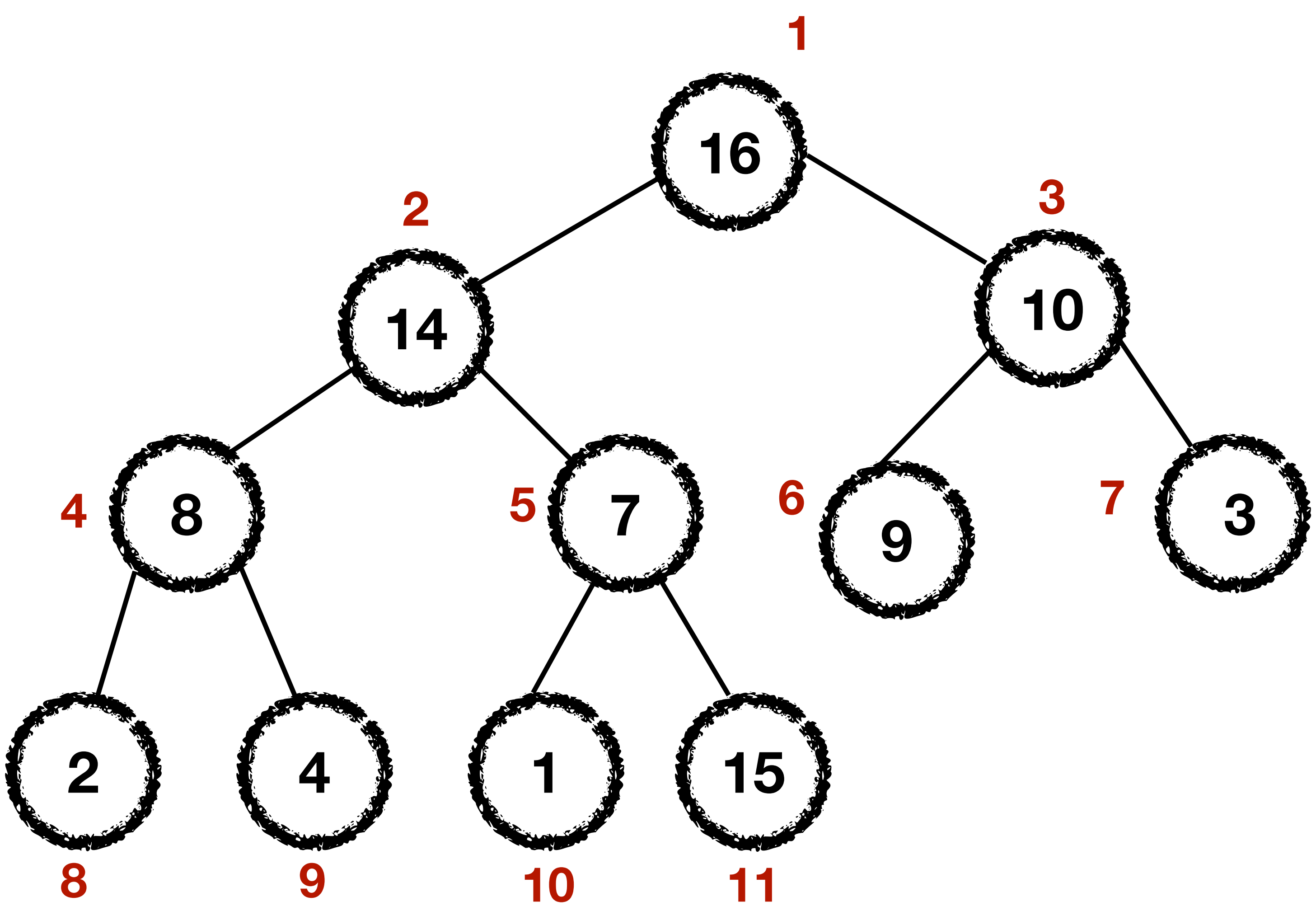


Max-Heap-Insert(A, v)

```
11 A.heap-size = A.heap-size + 1
   A[A.heap-size] = v    A[11] = 15
   i = A.heap-size
   Max-Heapify-Up (A, i)
```

1	2	3	4	5	6	7	8	9	10	11
16	14	10	8	7	9	3	2	4	1	15

Max-Heap-Insert $(A, 15)$

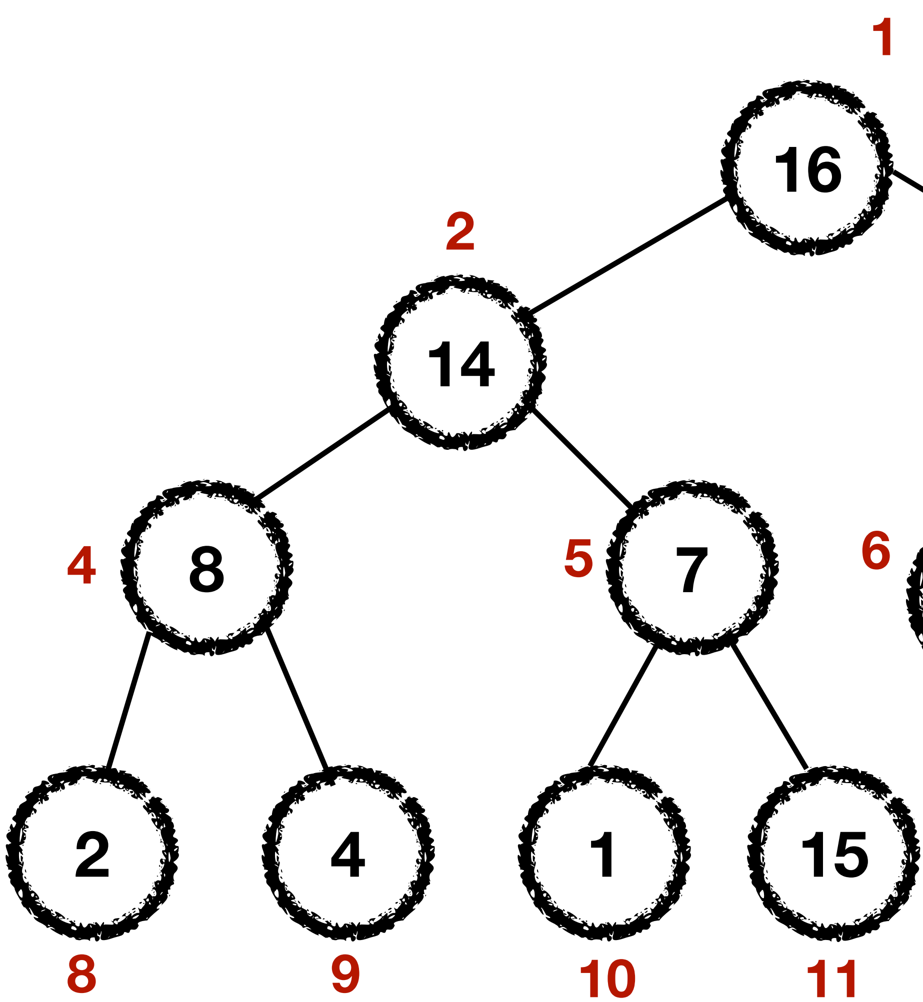


Max-Heap-Insert(A, v)

```
11  A.heap-size = A.heap-size + 1
    A[A.heap-size] = v    A[11] = 15
11  i = A.heap-size
    Max-Heapify-Up (A, i)
```

1	2	3	4	5	6	7	8	9	10	11
16	14	10	8	7	9	3	2	4	1	15

Max-Heap-Insert $(A, 15)$



Max-Heap-Insert(A, v)

```
11  A.heap-size = A.heap-size + 1
    A[A.heap-size] = v      A[11] = 15
11  i = A.heap-size
    Max-Heapify-Up (A, i)
```

1	2	3	4	5	6	7	8	9	10	11
16	14	10	8	7	9	3	2	4	1	15

Max-Heap-Insert(A, v)

- Max-Heap-Insert(A, v):
  Insert a new element v to the heap.

Max-Heap-Insert(A, v)

A.heap-size $= A$.heap-size $+ 1$
$A[A$.heap-size$] = v$
$i = A$.heap-size
Max-Heapify-Up (A, i)

Max-Heap-Insert(A, v)

- Max-Heap-Insert(A, v):
  Insert a new element v to the heap.

Max-Heap-Insert(A, v)

A.heap-size $= A$.heap-size $+ 1$
$A[A$.heap-size$] = v$
$i = A$.heap-size
Max-Heapify-Up (A, i)

What is the running time of Max-Heap-Insert?

Max-Heap-Insert(A, v)

- Max-Heap-Insert(A, v):
  Insert a new element v to the heap.

Max-Heap-Insert(A, v)

A.heap-size $= A$.heap-size $+ 1$
$A[A$.heap-size$] = v$
$i = A$.heap-size
Max-Heapify-Up (A, i)

What is the running time of Max-Heap-Insert?

What is the running time of Max-Heapify-Up?

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)
```

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

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)
```

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

Argument for Max-Heapify-Up almost identical!

Max-Heap-Insert(A, v) (without recursion)

- Max-Heap-Insert(A, v):
  Insert a new element v to the heap.

Max-Heap-Insert(A, v)

```
A.heap-size = A.heap-size + 1
A[A.heap-size] = v
i = A.heap-size
while (i ≠ 1 and A[i] > A[Parent(i)]) do
        exchange A[i] with A[Parent(i)]
        i = Parent(i)
```

Priority Queues

Priority Queues

- Priority queue: A data structure that maintains

Priority Queues

- Priority queue: A data structure that maintains
  - A set of elements S.

Priority Queues

- Priority queue: A data structure that maintains
  - A set of elements S.
  - Each with an associated value, $\text{key}(v)$.

Priority Queues

- Priority queue: A data structure that maintains
  - A set of elements S.
  - Each with an associated value, key(v).
  - The values denote *priorities*.

Priority Queues

- Priority queue: A data structure that maintains
  - A set of elements S.
  - Each with an associated value, key(v).
  - The values denote *priorities*.
    - For Max-Priority Queues, the elements with the largest values are those with the highest priority.

Priority Queues

Priority Queues

- Example: Scheduling processes on a computer.

Priority Queues

- Example: Scheduling processes on a computer.
    - Each process has a priority or urgency.

Priority Queues

- Example: Scheduling processes on a computer.
  - Each process has a priority or urgency.
  - Processes don't arrive in order of priorities.

Priority Queues

- Example: Scheduling processes on a computer.
  - Each process has a priority or urgency.
  - Processes don’t arrive in order of priorities.
  - From the set of active processes, we need to find that with the highest priority and run it.

Priority Queue Operations

Priority Queue Operations

- Insert(Q, v) inserts a new item v in the priority queue.

Priority Queue Operations

- Insert(Q, v) inserts a new item v in the priority queue.
- FindMax(Q) finds the element with the maximum priority (the highest value) in the priority queue and returns it (but does not remove it).

Priority Queue Operations

- Insert(Q, v) inserts a new item v in the priority queue.
- FindMax(Q) finds the element with the maximum priority (the highest value) in the priority queue and returns it (but does not remove it).
- ExtractMax(Q) finds the element with the maximum priority (highest value) in the priority queue, returns it, and deletes it from the queue.

Implementing Priority Queues

Implementing Priority Queues

- Approach 1: Store the elements in an array/list. Also maintain a pointer for the max element.

Implementing Priority Queues

- Approach 1: Store the elements in an array/list. Also maintain a pointer for the max element.
    - How long does it take to find the max element?

Implementing Priority Queues

- Approach 1: Store the elements in an array/list. Also maintain a pointer for the max element.
  - How long does it take to find the max element?
    - $O(1)$

Implementing Priority Queues

- Approach 1: Store the elements in an array/list. Also maintain a pointer for the max element.
  - How long does it take to find the max element?
    - $O(1)$
  - How long does it take to insert a new element?

Implementing Priority Queues

- Approach 1: Store the elements in an array/list. Also maintain a pointer for the max element.
  - How long does it take to find the max element?
    - $O(1)$
  - How long does it take to insert a new element?
    - Need to update the max pointer, hence $O(n)$.

Implementing Priority Queues

Implementing Priority Queues

- Approach 2: Store the elements in a *sorted* array/list.

Implementing Priority Queues

- Approach 2: Store the elements in a *sorted* array/list.
- How long does it take to find the max element?

Implementing Priority Queues

- Approach 2: Store the elements in a *sorted* array/list.
- How long does it take to find the max element?
  - $O(1)$

Implementing Priority Queues

- Approach 2: Store the elements in a *sorted* array/list.
- How long does it take to find the max element?
  - $O(1)$
- How long does it take to insert a new element?

Implementing Priority Queues

- Approach 2: Store the elements in a *sorted* array/list.
- How long does it take to find the max element?
    - $O(1)$
- How long does it take to insert a new element?
    - We need to find the right position to insert it in the array - $O(\lg n)$ using binary search.

Implementing Priority Queues

- Approach 2: Store the elements in a *sorted* array/list.
- How long does it take to find the max element?
    - $O(1)$
- How long does it take to insert a new element?
    - We need to find the right position to insert it in the array - $O(\lg n)$ using binary search.
    - We still need to insert it, which means moving all the later elements one position to the right - $O(n)$.

Implementing Priority Queues

Implementing Priority Queues

- Approach 3: Use a Max Heap.

Implementing Priority Queues

- Approach 3: Use a Max Heap.
- How long does it take to find the max element?

Implementing Priority Queues

- Approach 3: Use a Max Heap.
- How long does it take to find the max element?
    - $O(1)$

Implementing Priority Queues

- Approach 3: Use a Max Heap.
- How long does it take to find the max element?
  - $O(1)$
- How long does it take to insert a new element?

Implementing Priority Queues

- Approach 3: Use a Max Heap.
- How long does it take to find the max element?
    - $O(1)$
- How long does it take to insert a new element?
    - $O(\lg n)$ via Max-Heap-Insert.

Priority Queue Operations

- Insert(Q, v) inserts a new item v in the priority queue.
- FindMax(Q) finds the element with the maximum priority (the highest value) in the priority queue and returns it (but does not remove it).
- ExtractMax(Q) finds the element with the maximum priority (highest value) in the priority queue, returns it, and deletes it from the queue.

Priority Queue Operations

- Insert(Q, v) inserts a new item v in the priority queue. $O(\lg n)$
- FindMax(Q) finds the element with the maximum priority (the highest value) in the priority queue and returns it (but does not remove it).
- ExtractMax(Q) finds the element with the maximum priority (highest value) in the priority queue, returns it, and deletes it from the queue.

Priority Queue Operations

- Insert(Q, v) inserts a new item v in the priority queue. $O(\lg n)$
- FindMax(Q) finds the element with the maximum priority (the highest value) in the priority queue and returns it (but does not remove it). $O(1)$
- ExtractMax(Q) finds the element with the maximum priority (highest value) in the priority queue, returns it, and deletes it from the queue.

Priority Queue Operations

- Insert(Q, v) inserts a new item v in the priority queue. $O(\lg n)$
- FindMax(Q) finds the element with the maximum priority (the highest value) in the priority queue and returns it (but does not remove it). $O(1)$
- ExtractMax(Q) finds the element with the maximum priority (highest value) in the priority queue, returns it, and deletes it from the queue. $O(\lg n)$

Max-heaps notes

- Python `heapq` library.
- They use Min-heaps rather than Max-heaps
  - So do Roughgarden and KT.
  - Arrays indexed from 0 (these slides and CLRS/KT index from 1).
  - Names of operations are different
    - e.g., Their heapify is basically our Build-Max-Heap, our Max-Heapify is part of their Heapop (which is the equivalent of our Max-Heap-Extract-Max).

Reading

- **CLRS 6.5**
  - Notes: Uses max-heaps but presents the heap operations in the context of priority queues first, using an additional increase key operation.
- **KT 2.5.**
  - Notes: Very close to the exposition of these slides. Uses a min heap rather than a max heap, and further implements a general delete operation.
- **Roughgarden 10.2, 10.5**
  - Notes: Uses a min heap rather than a max heap. The operation heapify builds a heap from scratch, so it is like Build-Min-Heap. The operation that restores an “almost” heap into a heap is part of ExtractMin.