Quick Sort
and
Bucket Sort

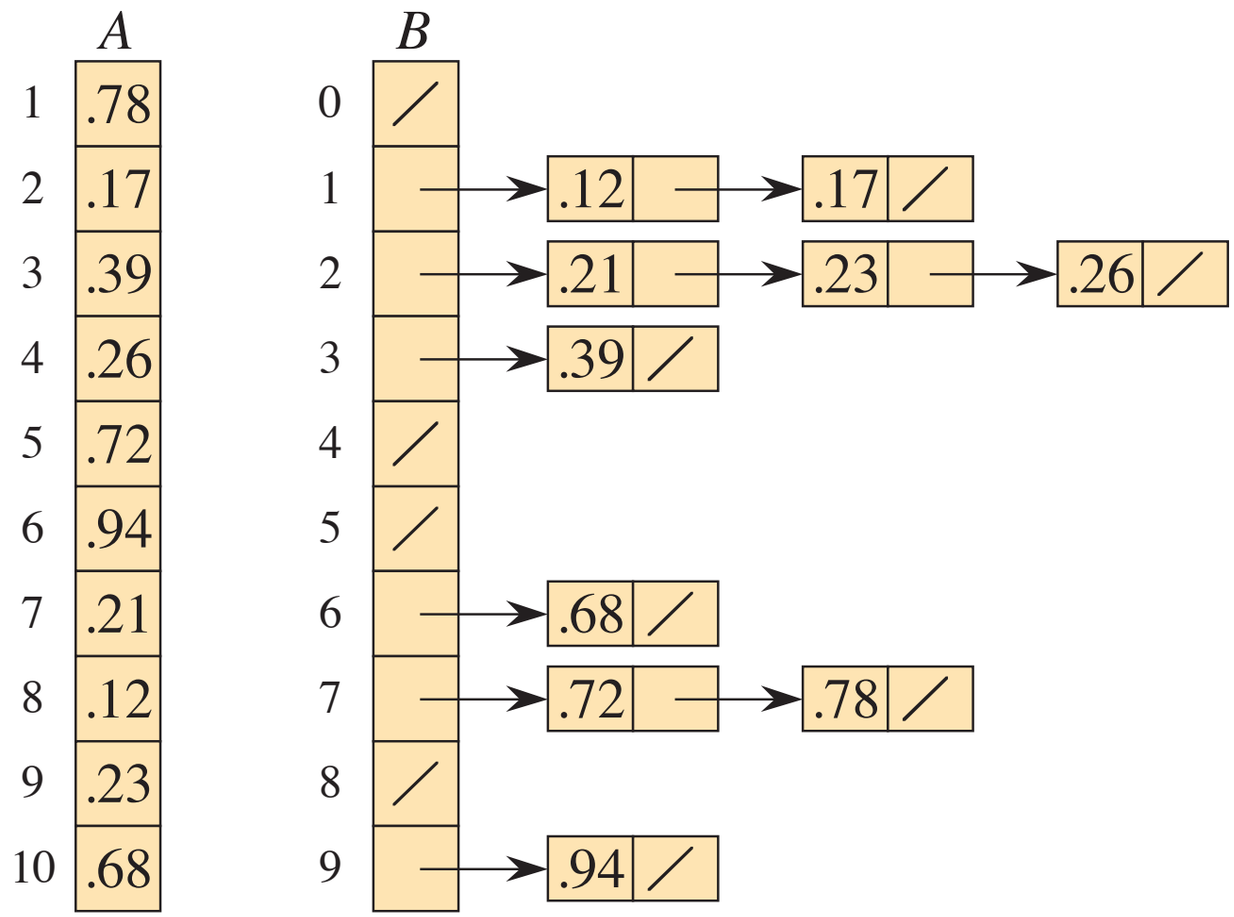
Lecture IV for course in
Data Structures & Algorithms for AI

Lectures by Hannah Santa Cruz Baur. Based on *Introduction to Algorithms*, Cormen et al.

Bucket Sort

- If we can assume the input array has elements that are uniformly distributed, we can design even faster algorithms.
- Bucket sorts assumes the input has values which are uniformly and independently sampled from the interval $[0,1)$.
- It runs in linear time.

Reminiscent of Hash tables :)



BUCKET SORT

```
BUCKET-SORT(A, n)
1  let B[0 : n − 1] be a new array
2  for i = 0 to n − 1
3      make B[i] an empty list
4  for i = 1 to n
5      insert A[i] into list B[⌊n · A[i]⌋]
6  for i = 0 to n − 1
7      sort list B[i] with insertion sort
8  concatenate the lists B[0], B[1], . . . , B[n − 1] together in order
9  return the concatenated lists
```

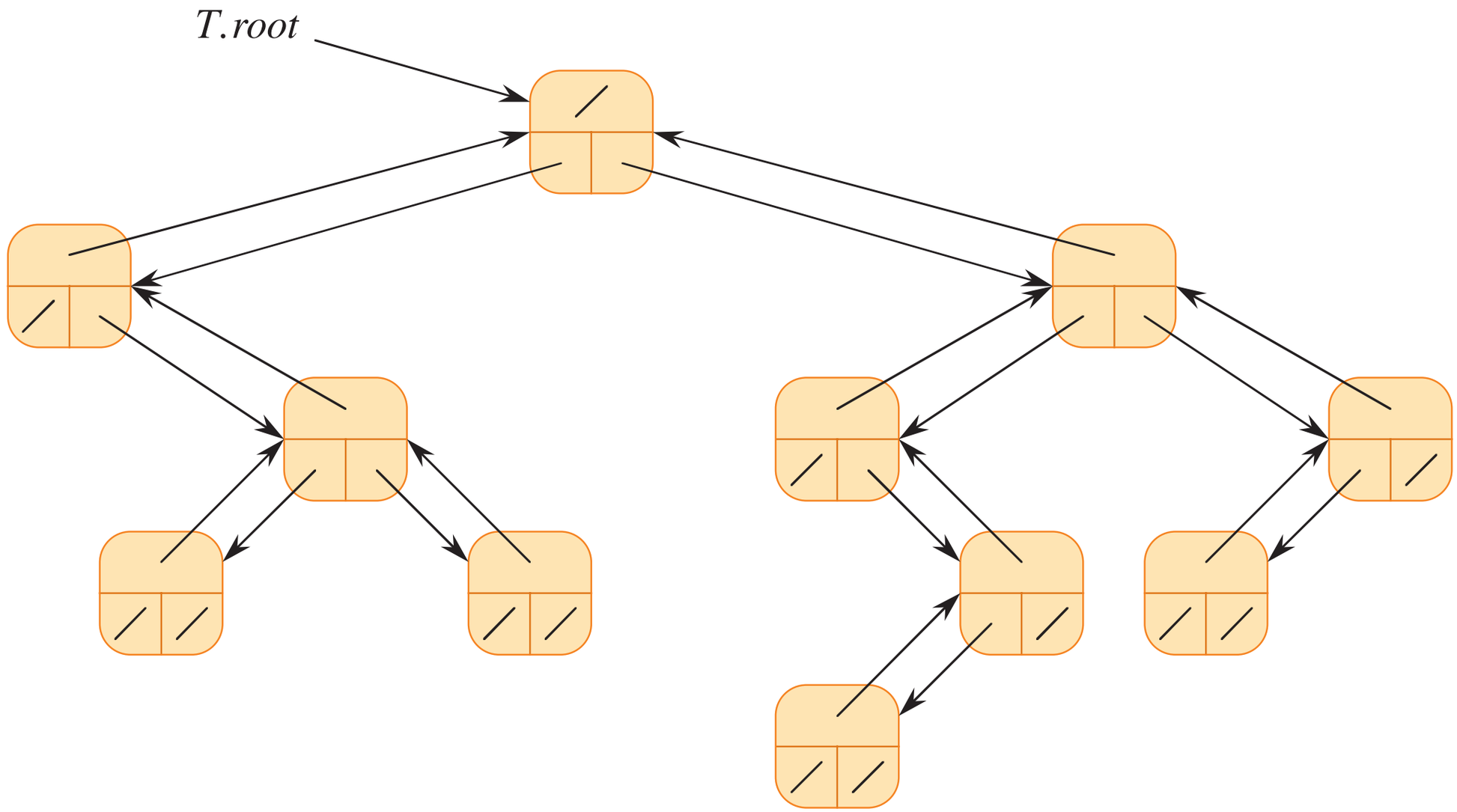
Linked lists, Rooted trees & Hash tables

Lecture III for course in Data Structures & Algorithms for AI

Lectures by Hannah Santa Cruz Baur. Based on *Introduction to Algorithms*, Cormen et al.

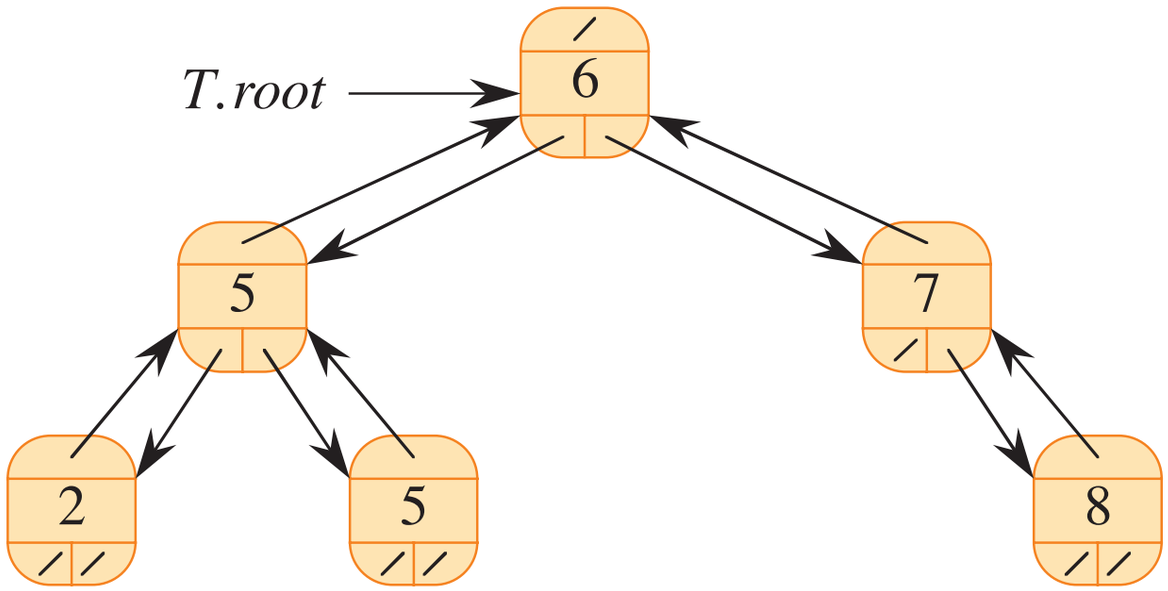
Rooted trees

- A binary tree is a non-linear structure, whose objects have a *key* attribute, along with pointer attributes, *p* (top), (lower) *left* and (lower) *right*.



Rooted trees

- Write a recursive procedure that prints out the key for each node in a n-node binary tree.



PRINT-TREE-WALK(x)

If $x \neq$ NIL

PRINT-TREE-WALK($x.left$)

print($x.key$)

PRINT-TREE-WALK($x.right$)

- What order would the keys be printed in? *2, 5, 5, 6, 7, 8*

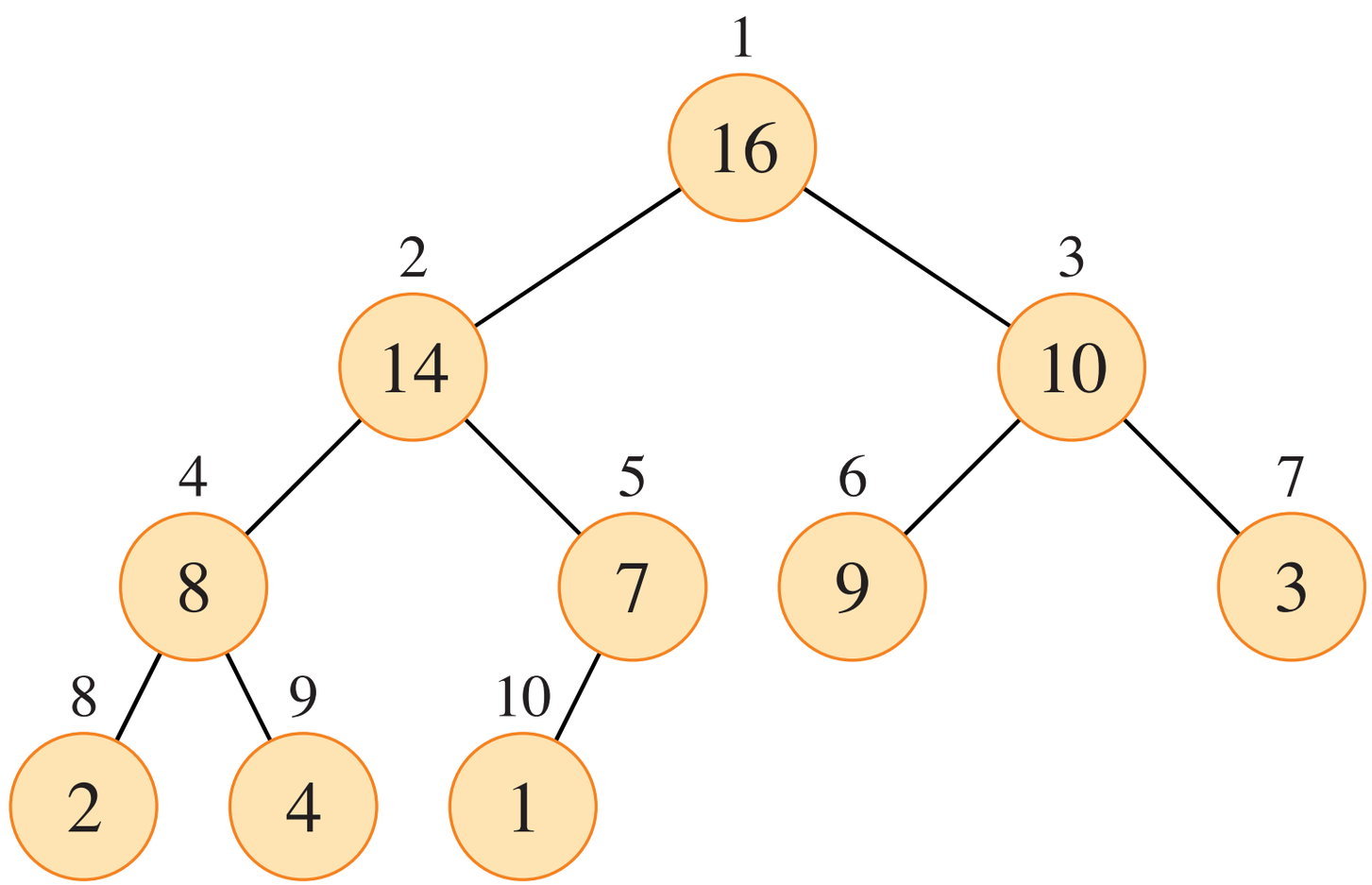
Heaps and Heap Sort

Lecture V for course in Data Structures & Algorithms for AI

Lectures by Hannah Santa Cruz Baur. Based on *Introduction to Algorithms*, Cormen et al.

Heaps

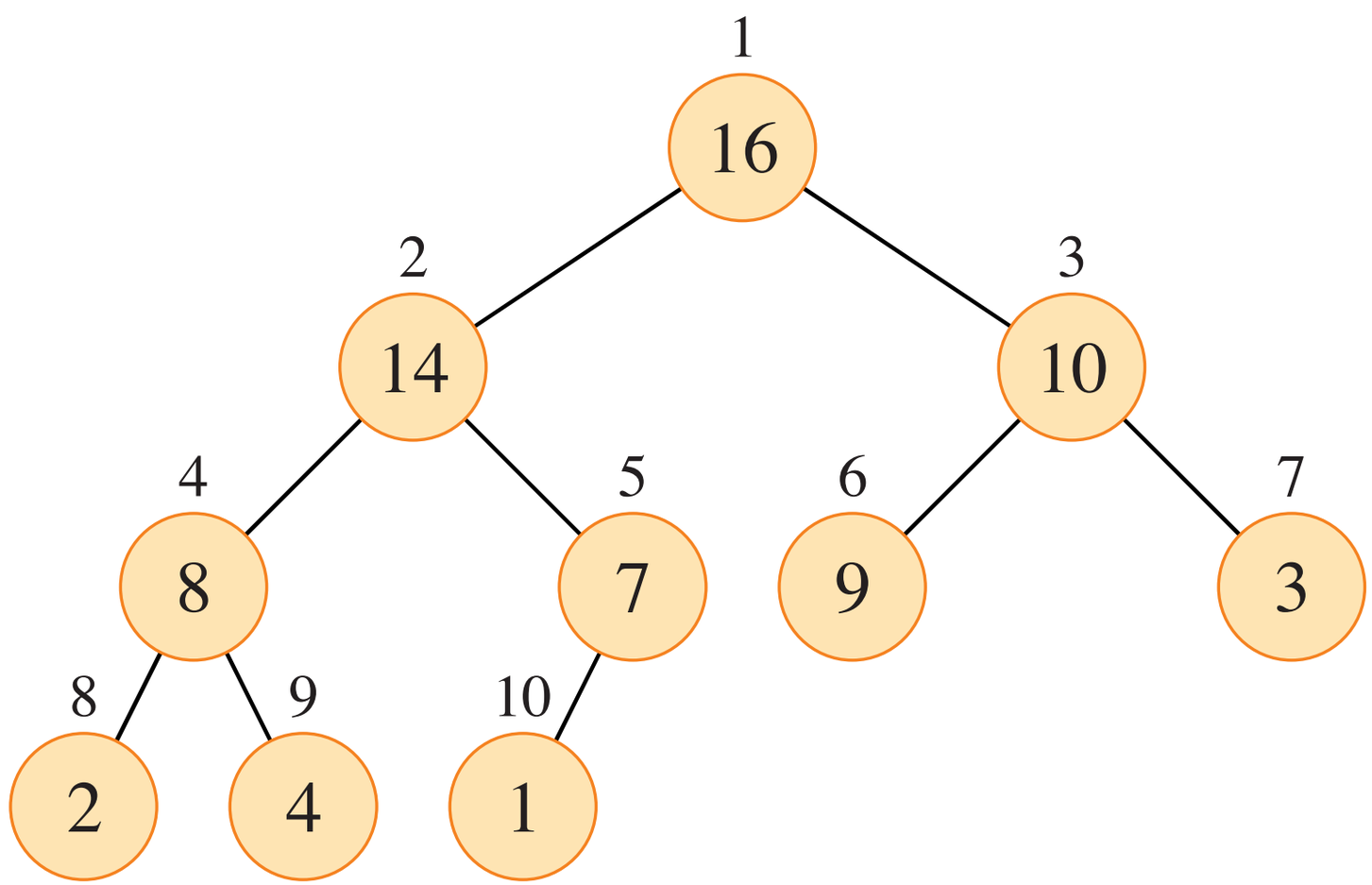
- A (binary) **heap** is a data structure which can be viewed as a *complete* binary tree, satisfying a key property.



Heaps

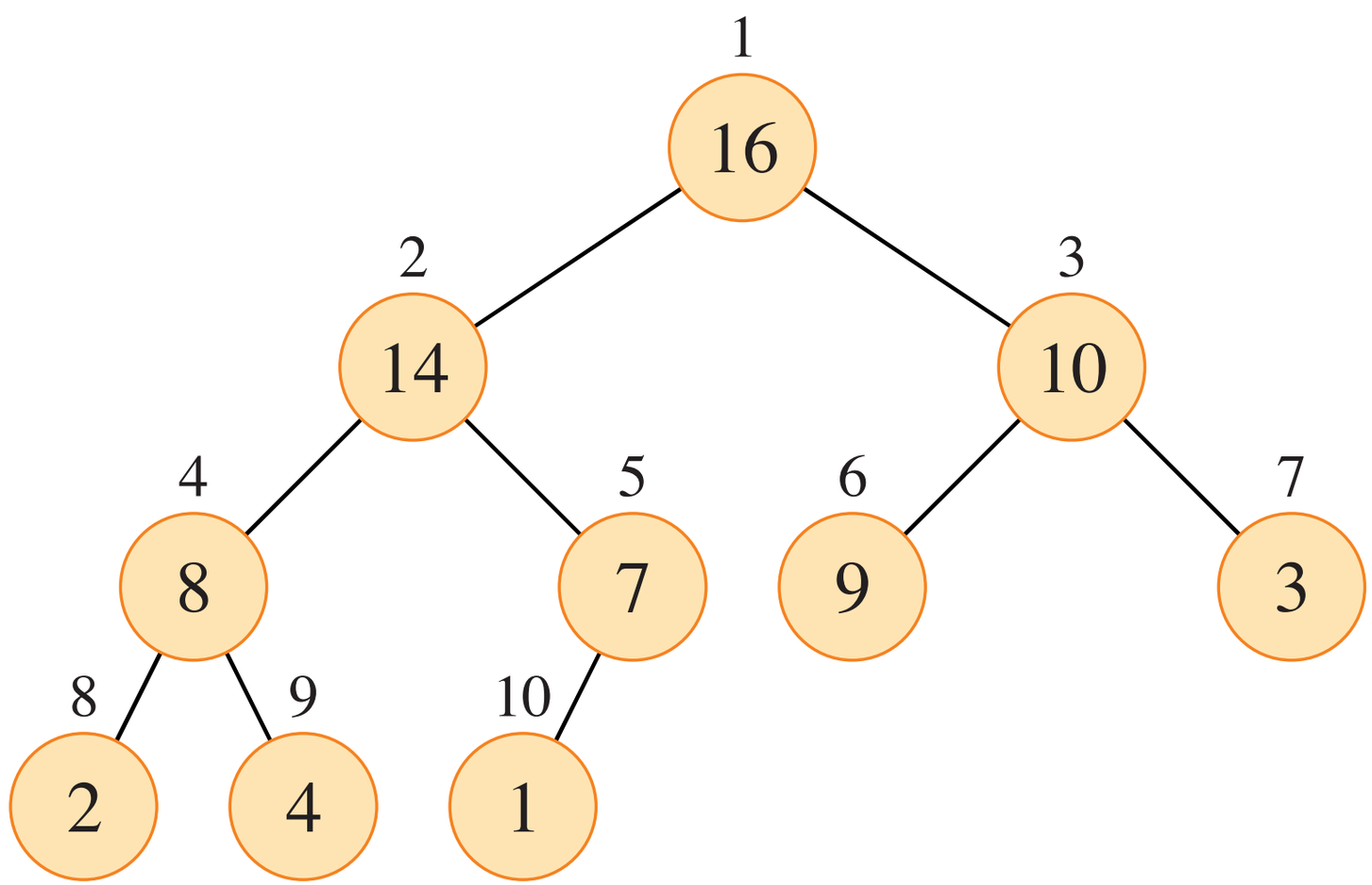
- A (binary) **heap** is a data structure which can be viewed as a *complete* binary tree, satisfying a key property.

  Every level (but possibly the last) is filled



Heaps

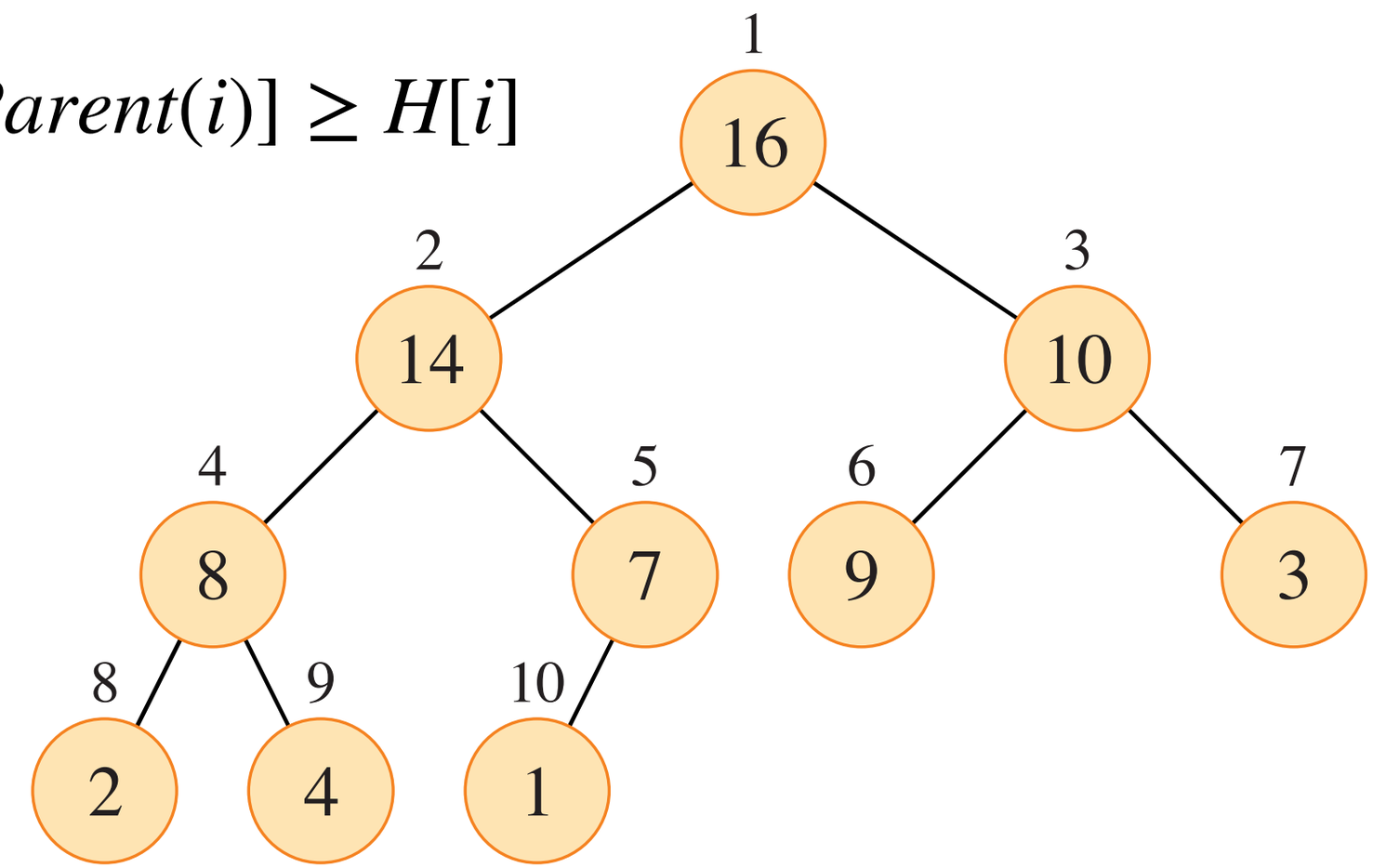
- A (binary) **heap** is a data structure which can be viewed as a *complete* binary tree, satisfying a key property.



Heaps

- A (binary) **heap** is a data structure which can be viewed as a *complete* binary tree, satisfying a key property.
- A **max-heap**, H, satisfies the *max heap property*:

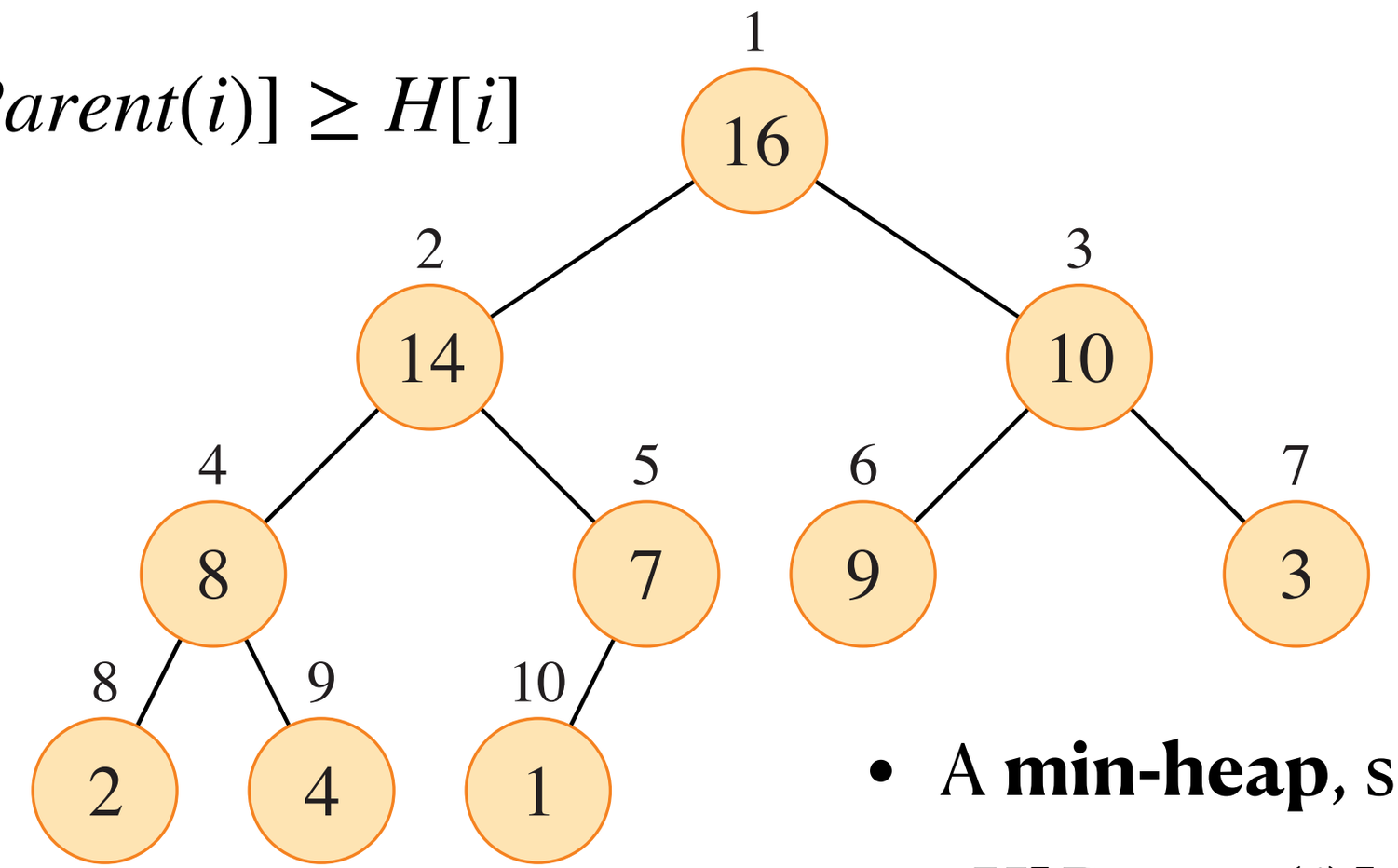
$$H[Parent(i)] \geq H[i]$$



Heaps

- A (binary) **heap** is a data structure which can be viewed as a *complete* binary tree, satisfying a key property.
- A **max-heap**, H, satisfies the *max heap property*:

$$H[Parent(i)] \geq H[i]$$

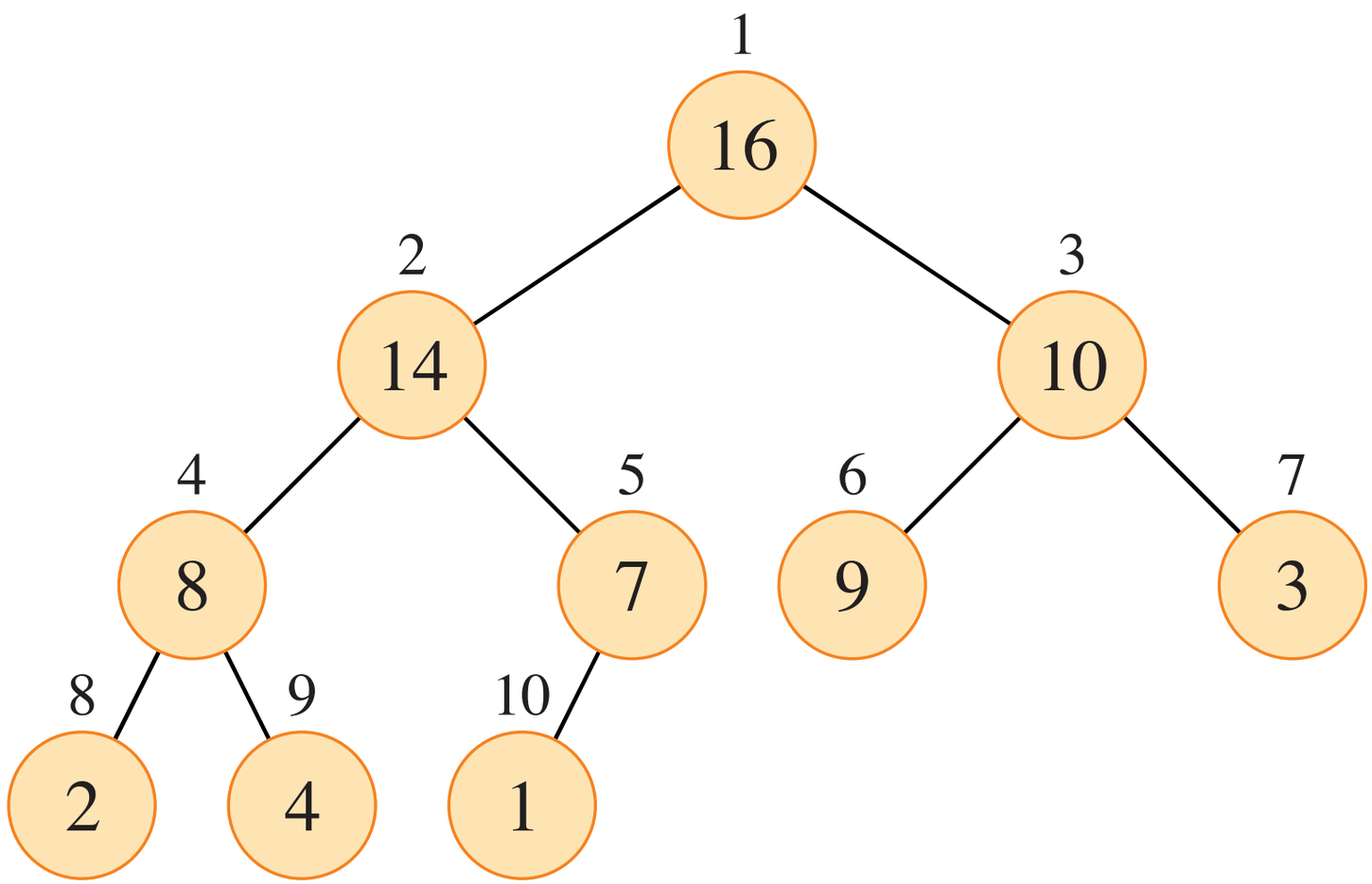


- A **min-heap**, satisfies:

$$H[Parent(i)] \leq H[i]$$

Exercise

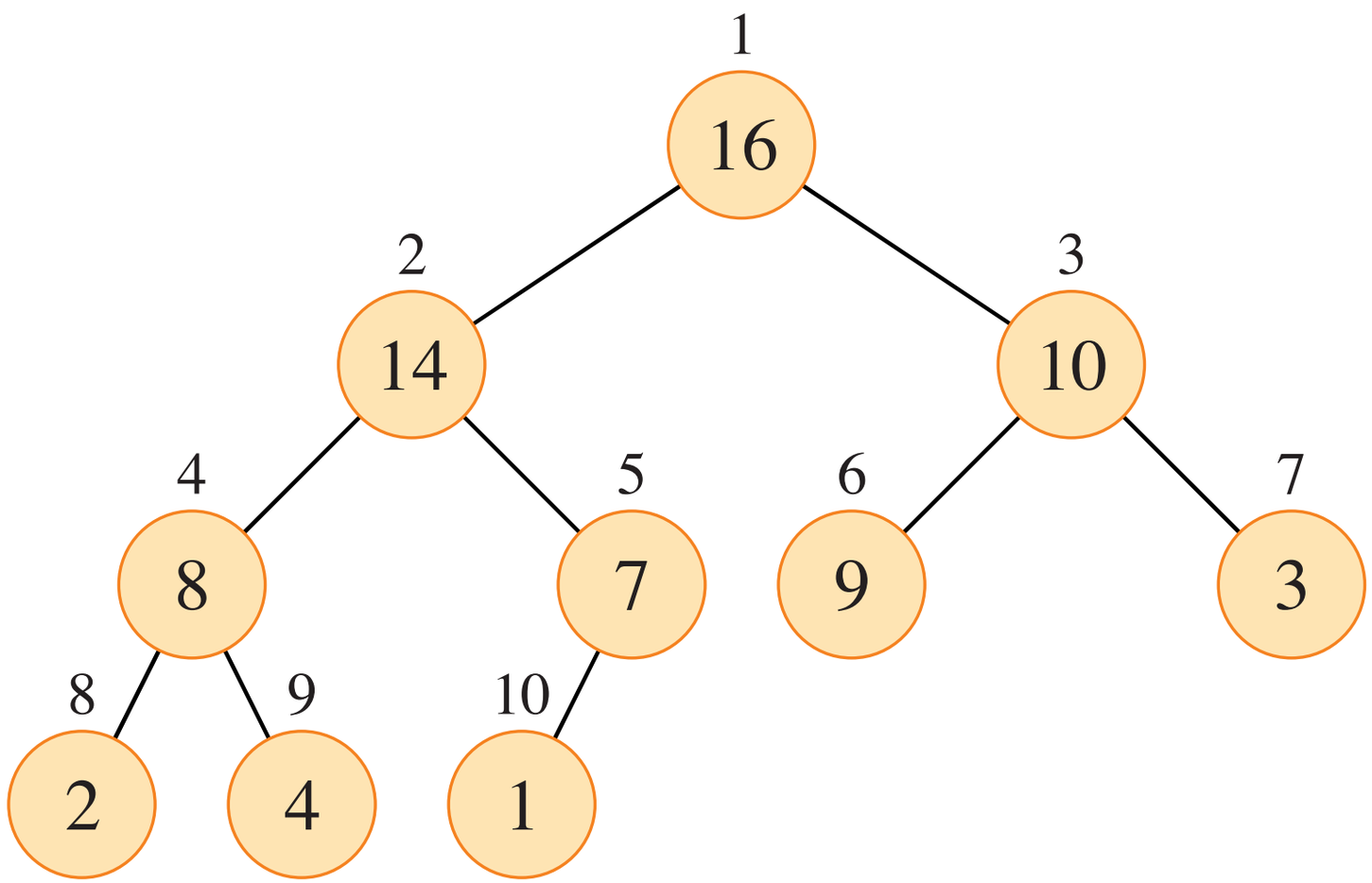
- What are the minimum and maximum number of elements in a heap of height h?



Exercise

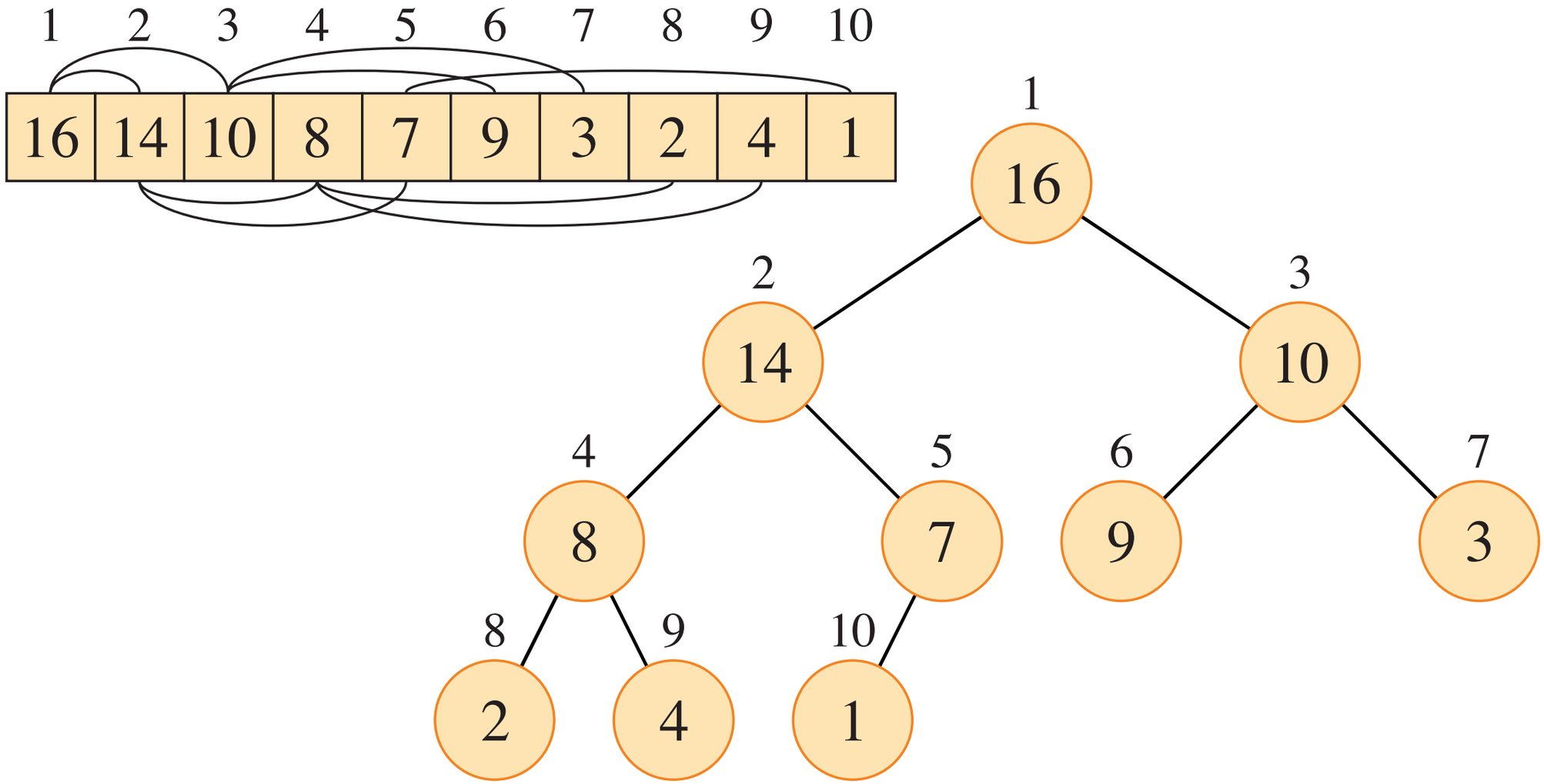
- What are the minimum and maximum number of elements in a heap of height h?

max= 1 + 2 + 4 + ... + 2^(h-1) = 2^(h) - 1



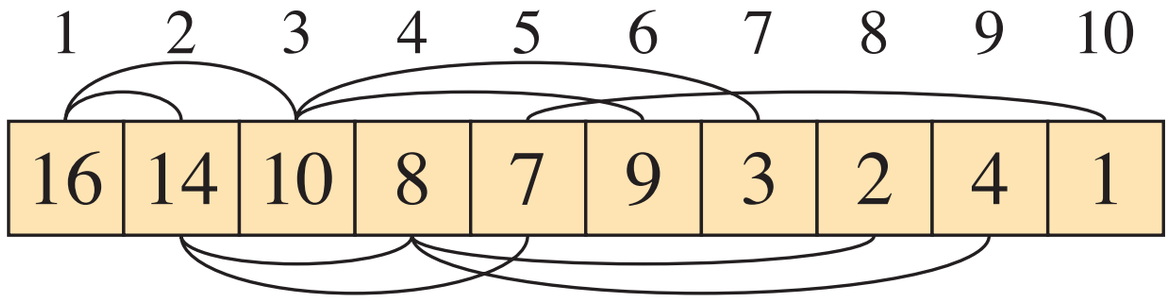
Exercise

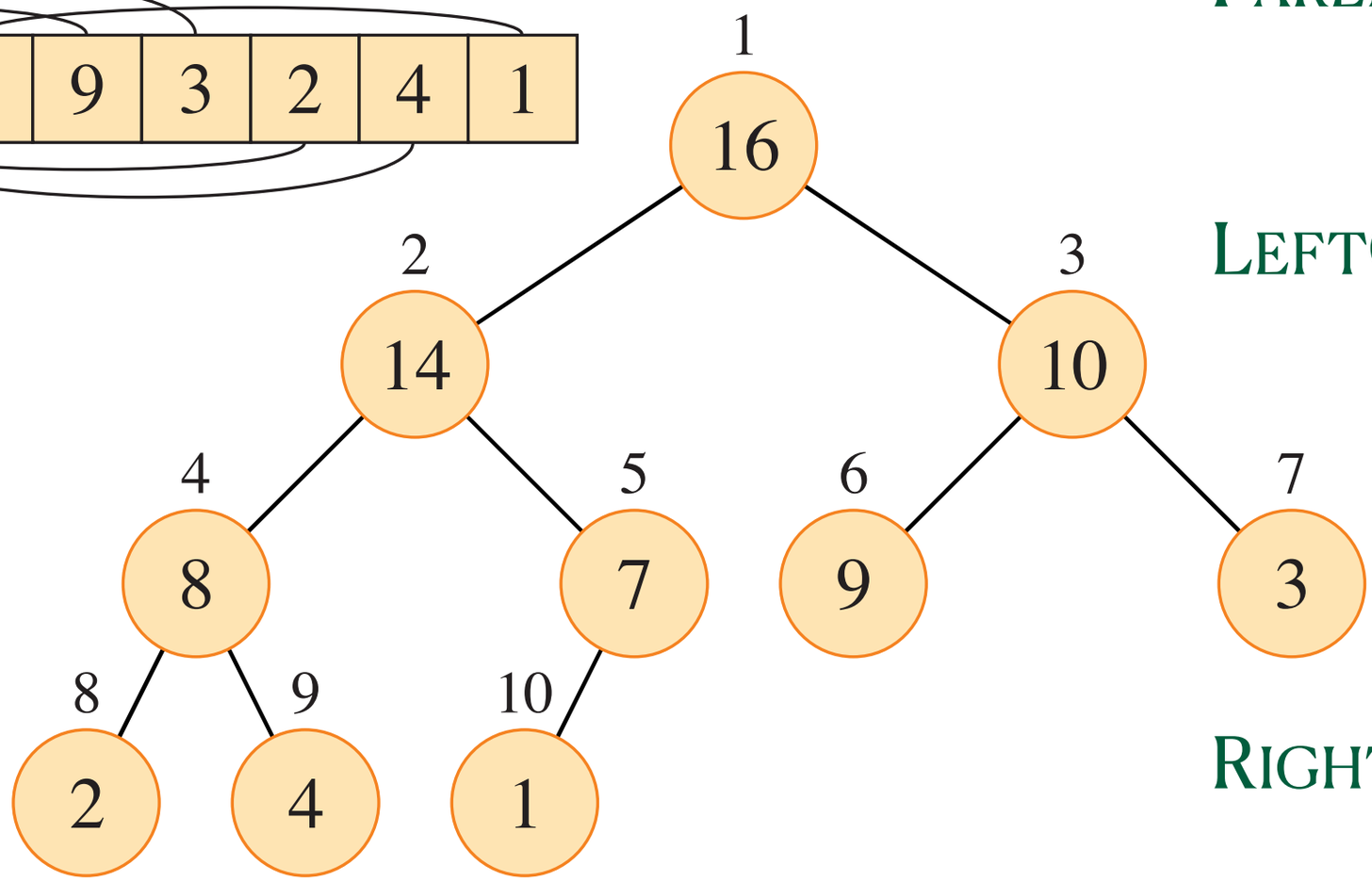
- Say you want to store a heap as an array, how would you access the parent, left child and right child of an element?



Exercise

- Say you want to store a heap as an array, how would you access the parent, left child and right child of an element?





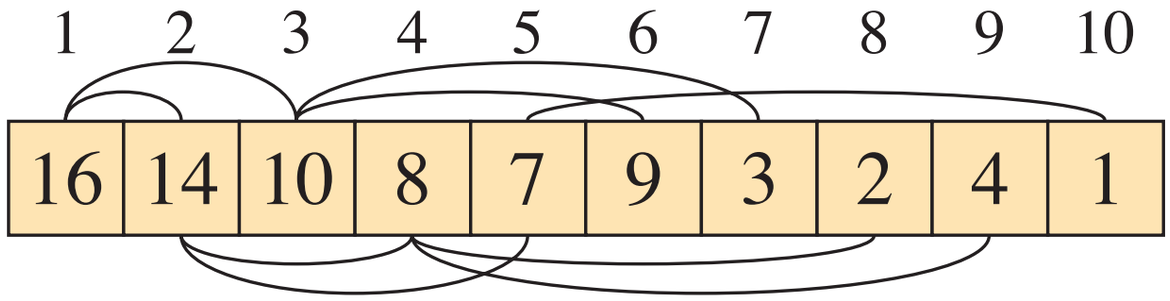
PARENT(i)

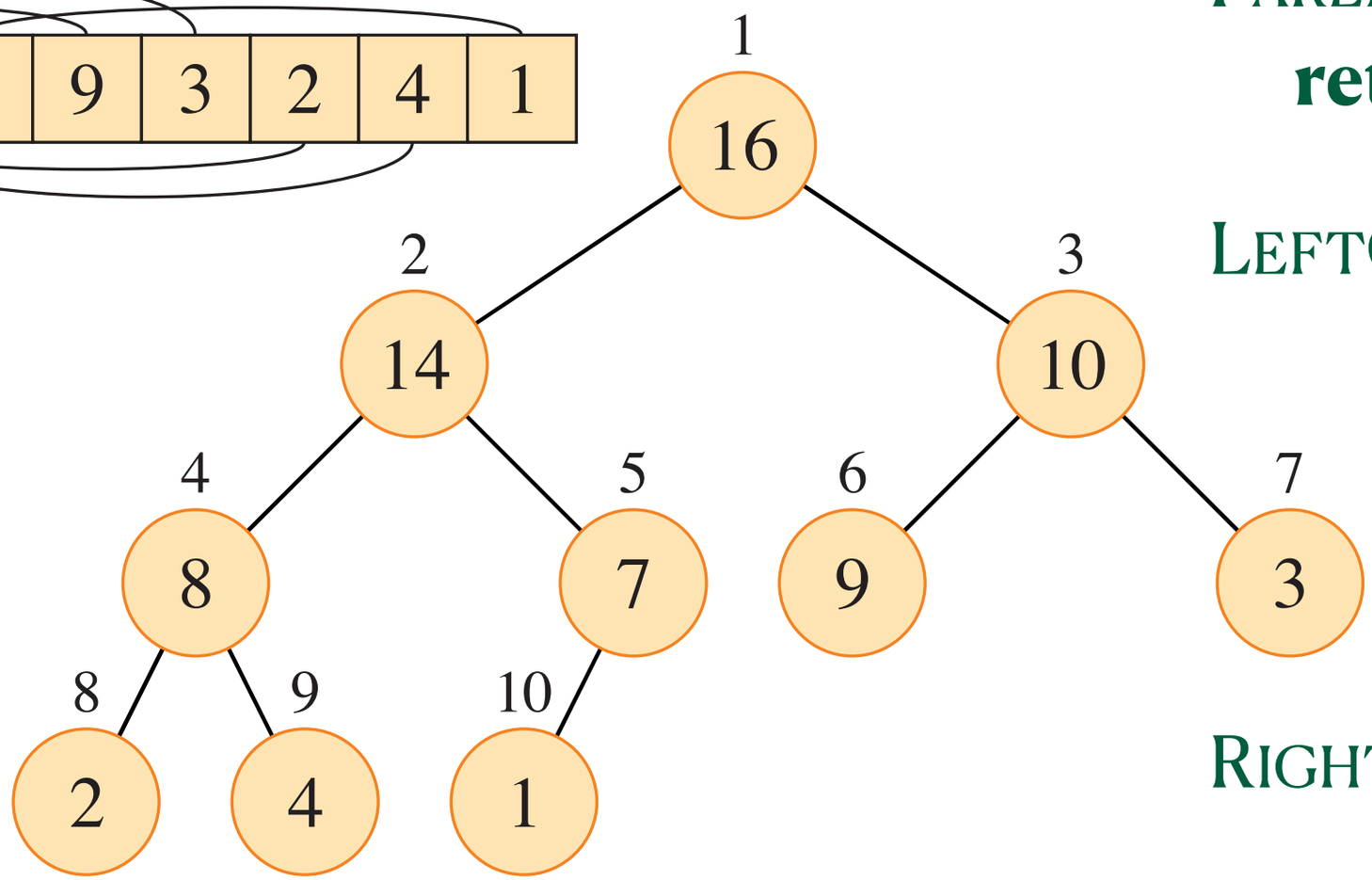
LEFTCHILD(i)

RIGHTCHILD(i)

Exercise

- Say you want to store a heap as an array, how would you access the parent, left child and right child of an element?





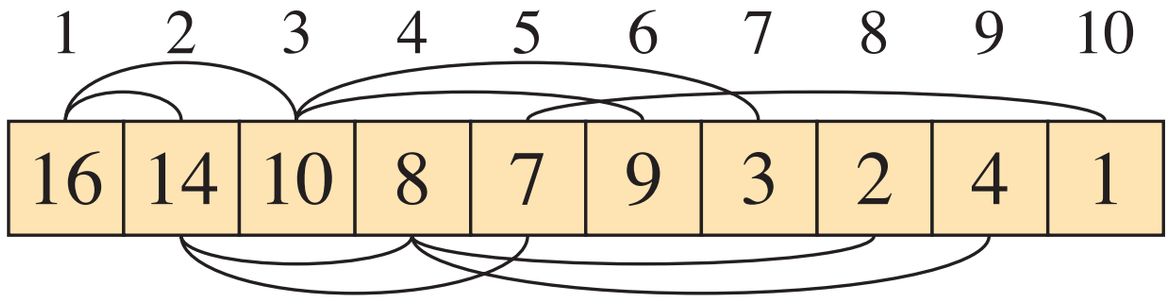

```
PARENT(i)
    return ⌊i/2⌋
```

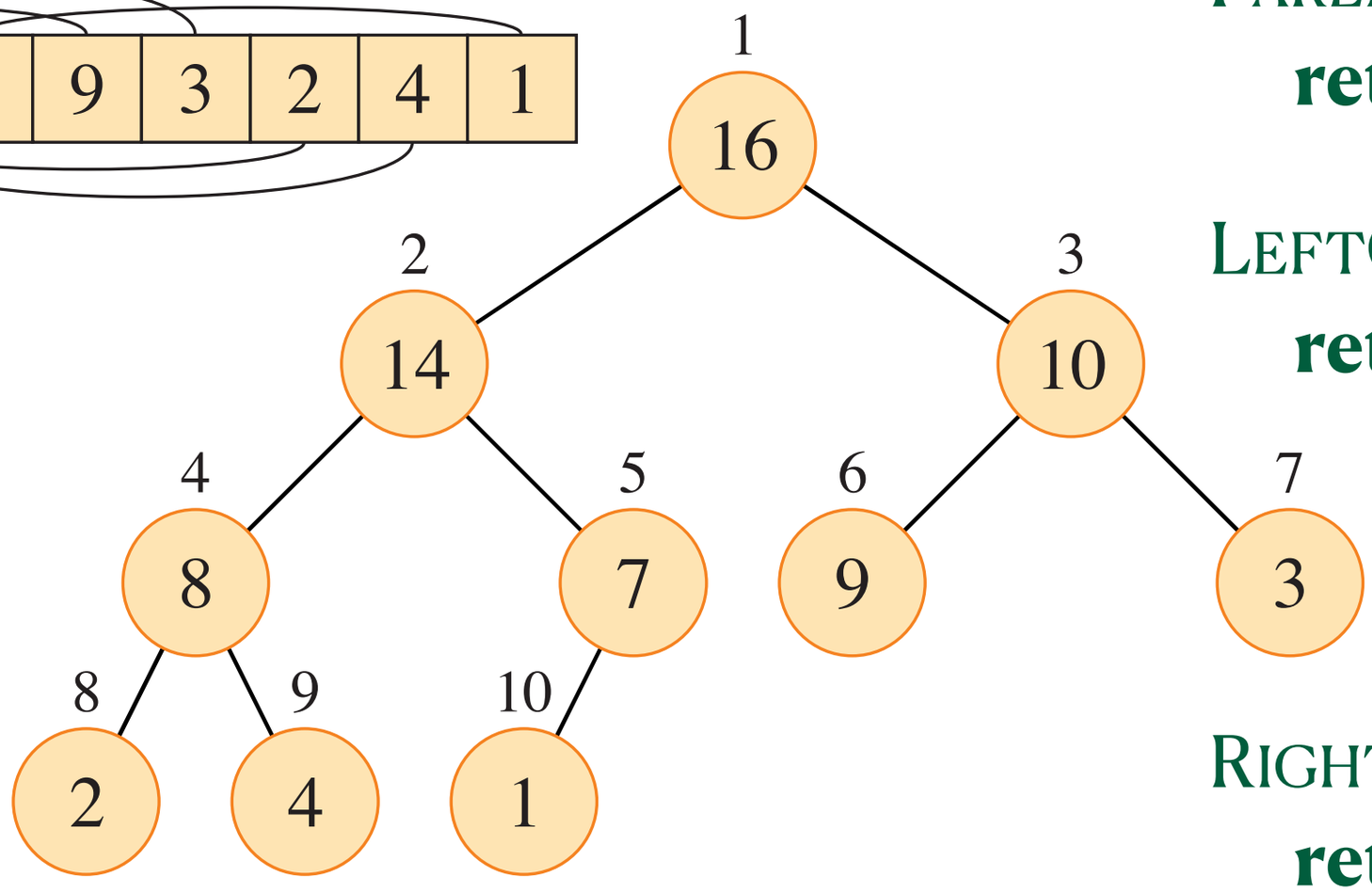
LEFTCHILD(i)

RIGHTCHILD(i)

Exercise

- Say you want to store a heap as an array, how would you access the parent, left child and right child of an element?





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

LEFTCHILD(i)
return $2i$

RIGHTCHILD(i)
return $2i + 1$

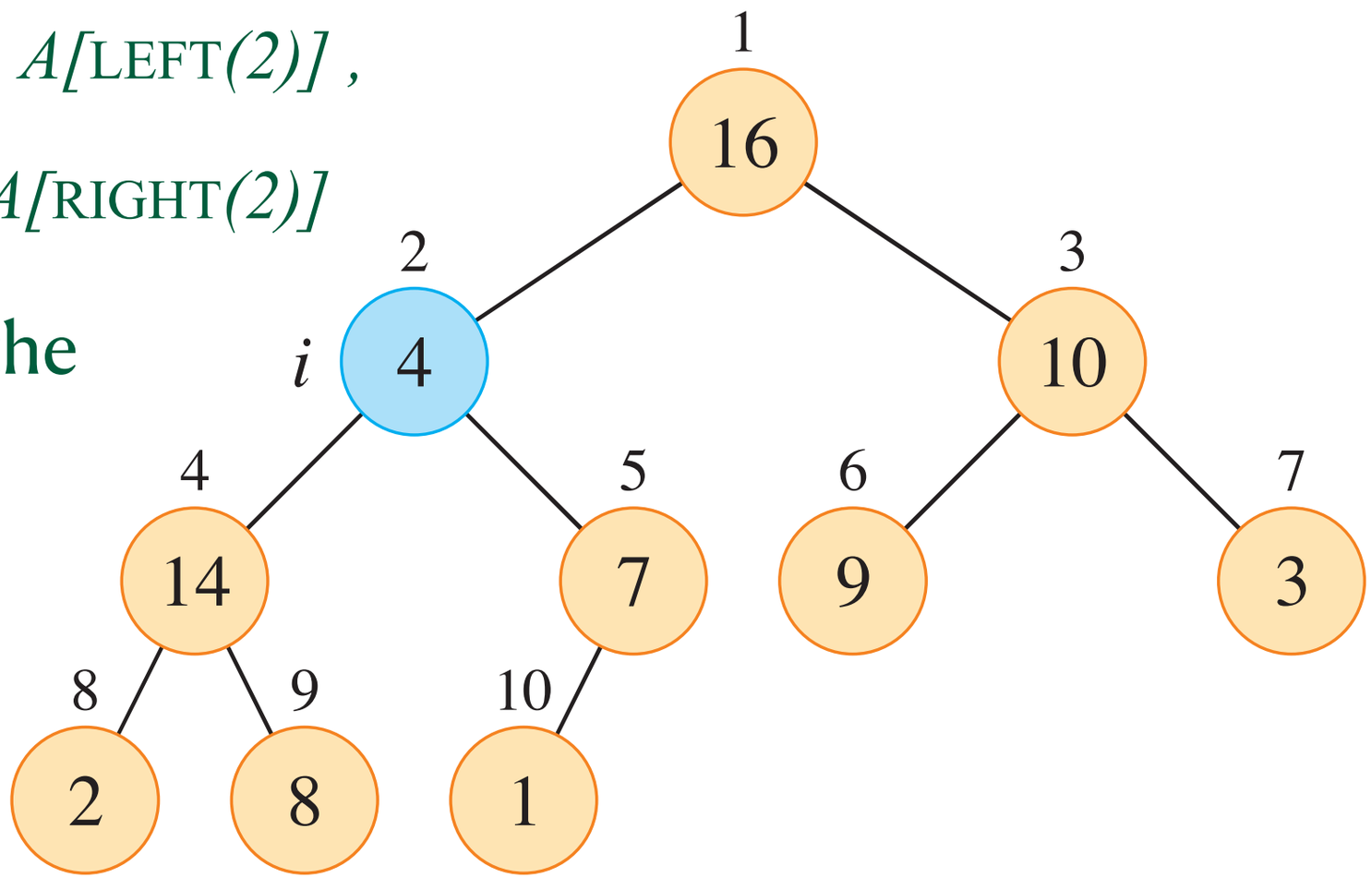
Basic operations for Heaps

- A crucial operation to define is MAX-HEAPIFY(A, i), which takes a heap A, where the max-heap property is violated with respect to node i, and restores the property at that node.

$A[2] < A[\text{LEFT}(2)]$,

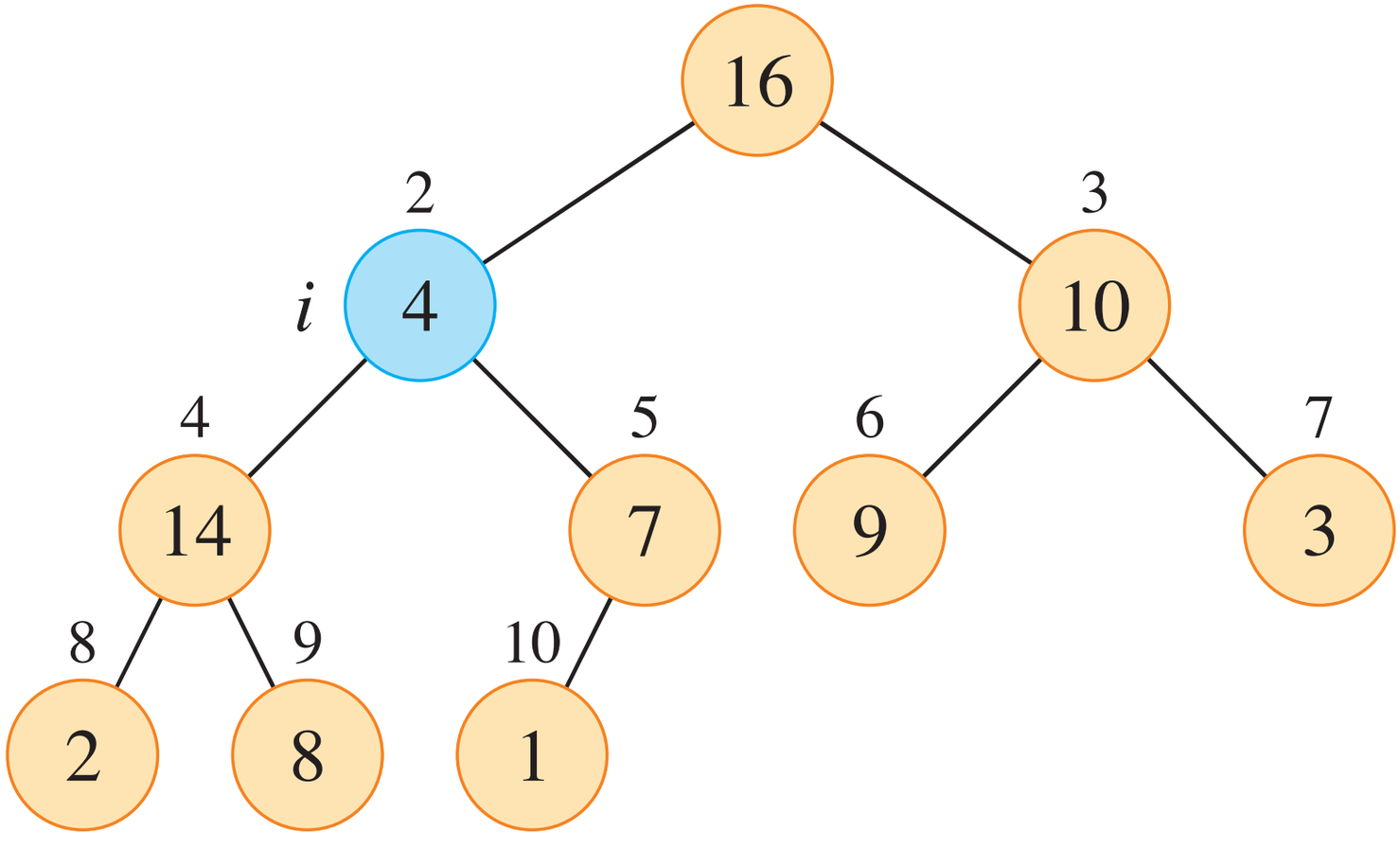
$A[\text{RIGHT}(2)]$

Violating the max-heap property.



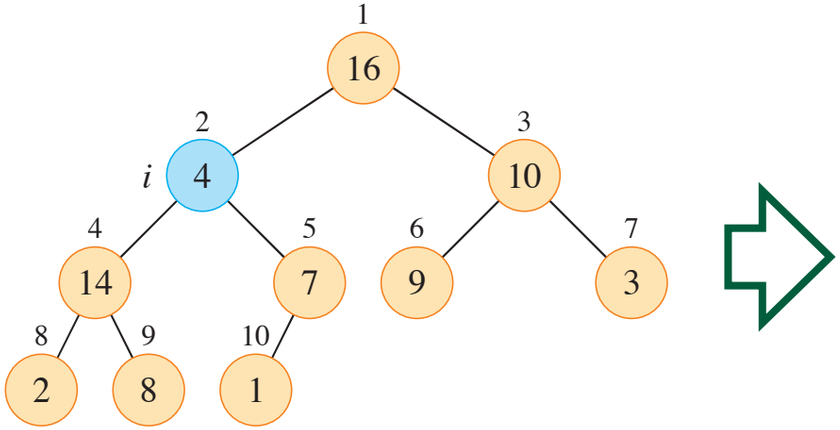
Basic operations for Heaps

- A crucial operation to define is MAX-HEAPIFY(A, i), which takes a heap A, where the max-heap property is violated with respect to node i, and restores the property at that node.



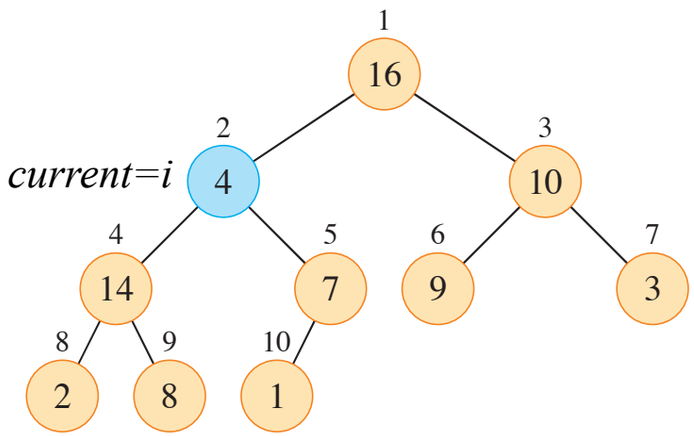
Basic operations for Heaps

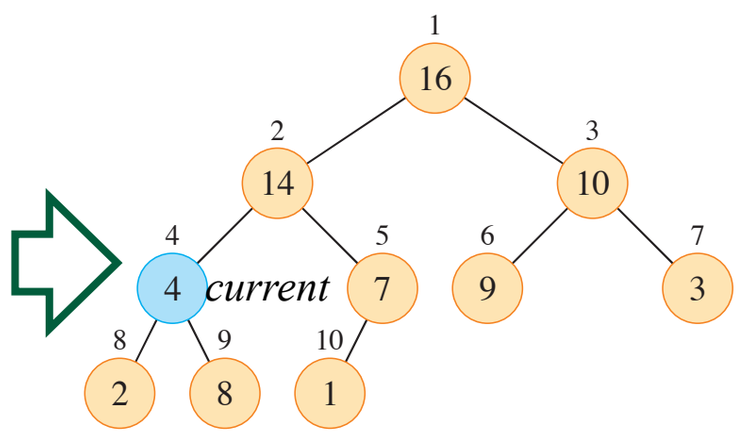
- A crucial operation to define is MAX-HEAPIFY(A, i), which takes a heap A, where the max-heap property is violated with respect to node i, and restores the property at that node.

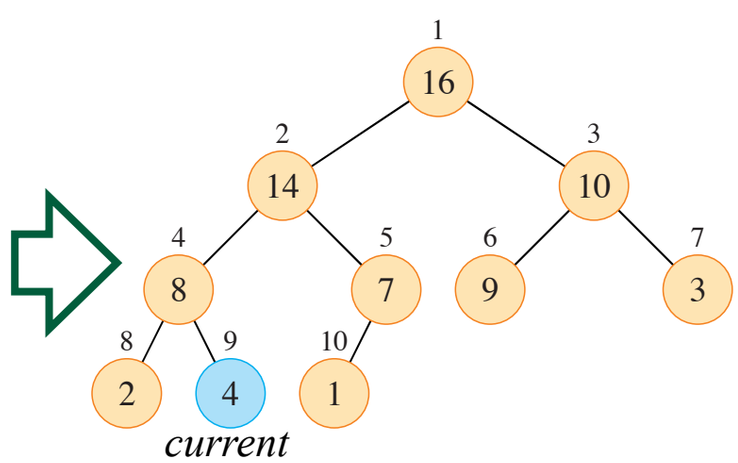


Basic operations for Heaps

- A crucial operation to define is MAX-HEAPIFY(A, i), which takes a heap A, where the max-heap property is violated with respect to node i, and restores the property at that node.







current=i,

A[current], $<$ *A[*LEFT*(current)]:*
*A[*RIGHT*(current)]*

swap &
current= LEFT*(current)*

A[current], $<$ *A[*RIGHT*(current)]:*
*A[*LEFT*(current)]*

swap &
current= RIGHT*(current)*

Basic operations for Heaps

- A crucial operation to define is MAX-HEAPIFY(A, i), which takes a heap A, where the max-heap property is violated with respect to node i, and restores the property at that node.

```
MAX-HEAPIFY(A, i)
1   l = LEFT(i)
2   r = RIGHT(i)
3   if l ≤ A.heap-size and A[l] > A[i]
4       largest = l
5   else largest = i
6   if r ≤ A.heap-size and A[r] > A[largest]
7       largest = r
8   if largest ≠ i
9       exchange A[i] with A[largest]
10      MAX-HEAPIFY(A, largest)
```

Find the largest amongst:

*A[i], A[*LEFT*(i)], A[*RIGHT*(i)]*

Basic operations for Heaps

- A crucial operation to define is MAX-HEAPIFY(A, i), which takes a heap A, where the max-heap property is violated with respect to node i, and restores the property at that node.

```
MAX-HEAPIFY(A, i)
1   l = LEFT(i)
2   r = RIGHT(i)
3   if l ≤ A.heap-size and A[l] > A[i]
4       largest = l
5   else largest = i
6   if r ≤ A.heap-size and A[r] > A[largest]
7       largest = r
8   if largest ≠ i
9       exchange A[i] with A[largest]
10      MAX-HEAPIFY(A, largest)
```

Find the largest amongst:
*A[i], A[*LEFT*(i)], A[*RIGHT*(i)]*

Exchange the largest with node i

Basic operations for Heaps

- A crucial operation to define is MAX-HEAPIFY(A, i), which takes a heap A, where the max-heap property is violated with respect to node i, and restores the property at that node.

```
MAX-HEAPIFY(A, i)
1   l = LEFT(i)
2   r = RIGHT(i)
3   if l ≤ A.heap-size and A[l] > A[i]
4       largest = l
5   else largest = i
6   if r ≤ A.heap-size and A[r] > A[largest]
7       largest = r
8   if largest ≠ i
9       exchange A[i] with A[largest]
10      MAX-HEAPIFY(A, largest)
```

Find the largest amongst:
*A[i], A[*LEFT*(i)], A[*RIGHT*(i)]*

Exchange the largest with node i

Recursively call the function, on new largest node.

Basic operations for Heaps

- The MAX-HEAPIFY(A, i), procedure can be used to build a heap from an un-ordered array.

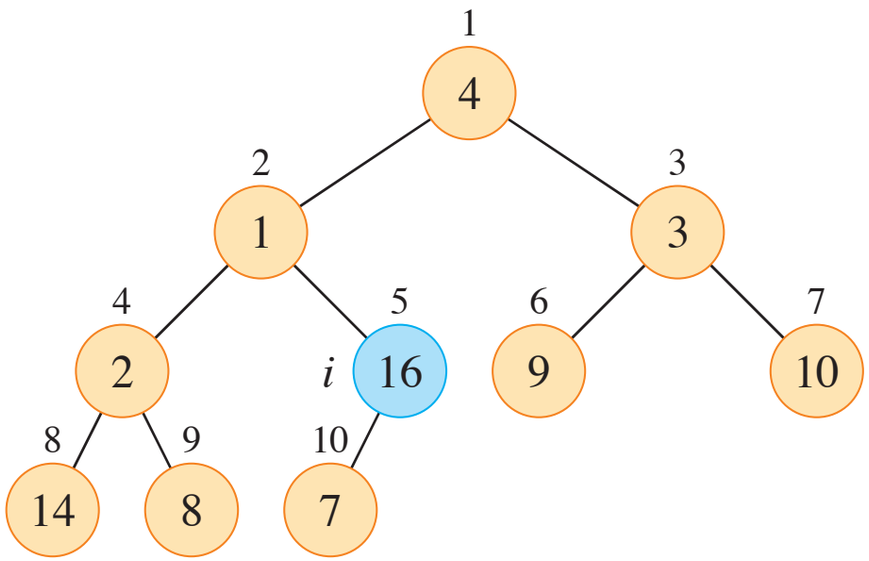
```
MAX-HEAPIFY(A, i)
 1  l = LEFT(i)
 2  r = RIGHT(i)
 3  if l ≤ A.heap-size and A[l] > A[i]
 4        largest = l
 5  else largest = i
 6  if r ≤ A.heap-size and A[r] > A[largest]
 7        largest = r
 8  if largest ≠ i
 9        exchange A[i] with A[largest]
10        MAX-HEAPIFY(A, largest)
```

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

Basic operations for Heaps

- The MAX-HEAPIFY(A, i), procedure can be used to build a heap from an un-ordered array, starting with the central node.

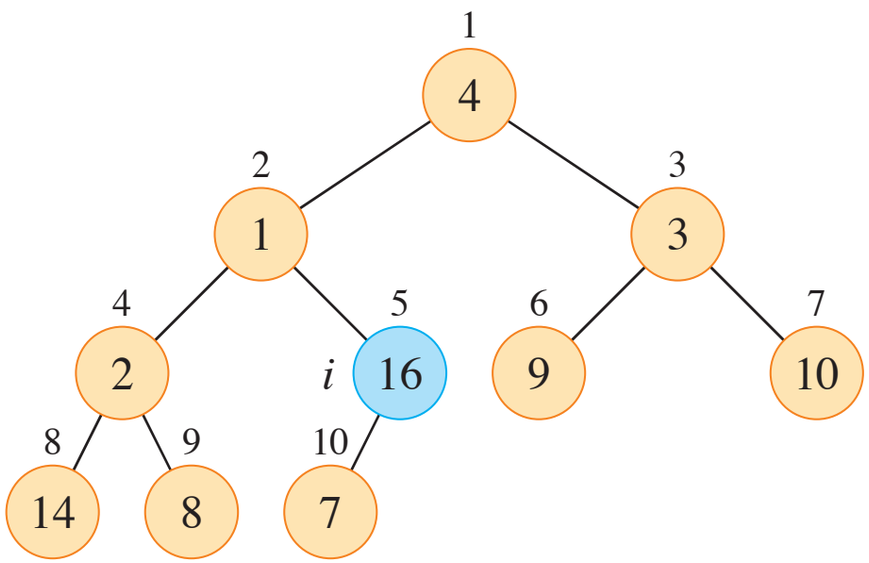
A	4	1	3	2	16	9	10	14	8	7



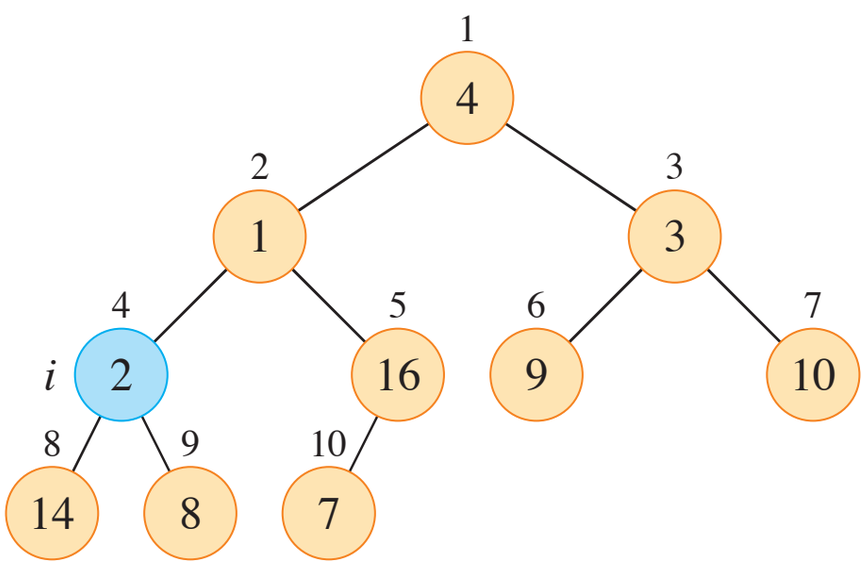

```
BUILD-MAX-HEAP(A, n)
1  A.heap-size = n
2  for i = ⌊n/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

Basic operations for Heaps

- The MAX-HEAPIFY(A, i), procedure can be used to build a heap from an un-ordered array, starting with the central node.



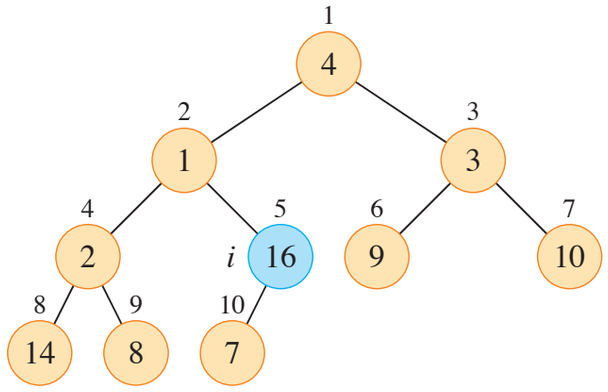
First call: MAX-HEAPIFY($A, 5$), changes nothing

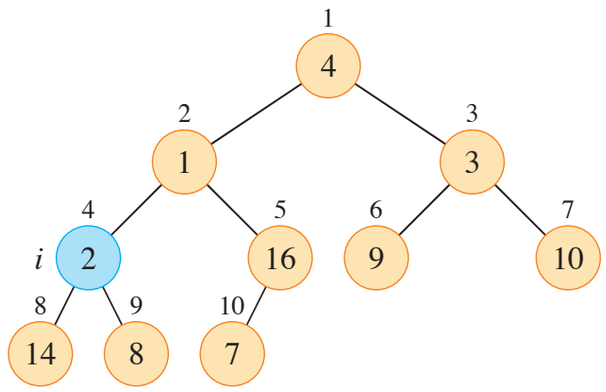


Now call MAX-HEAPIFY($A, 4$)

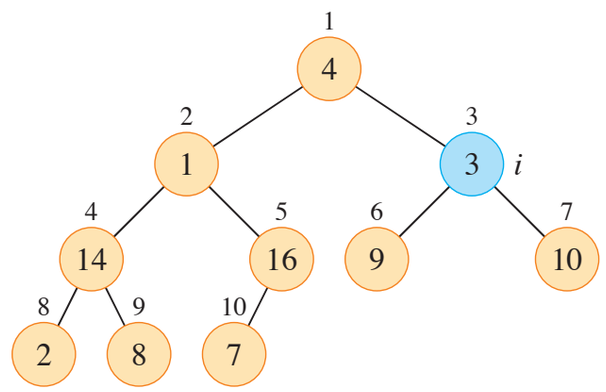
Basic operations for Heaps

- The MAX-HEAPIFY(A, i), procedure can be used to build a heap from an un-ordered array, starting with the central node.





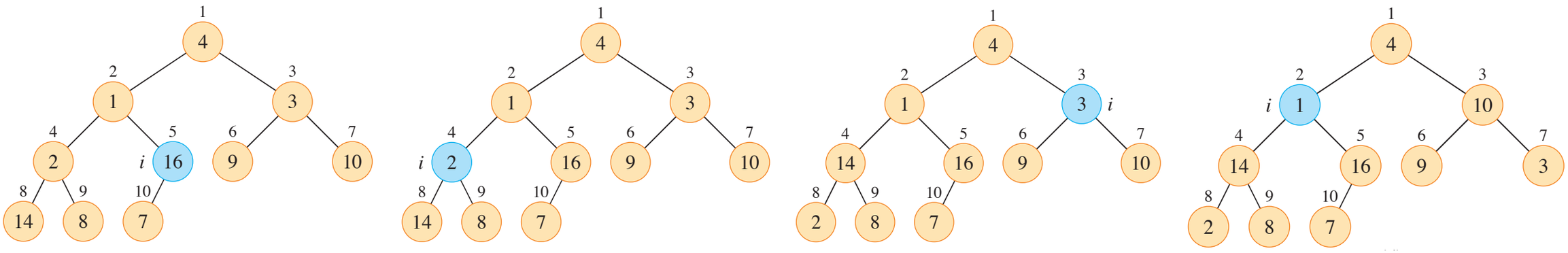
Call MAX-HEAPIFY*(A,4)*, make a swap with LEFT



Now call MAX-HEAPIFY*(A,3)*

Basic operations for Heaps

- The MAX-HEAPIFY(A, i), procedure can be used to build a heap from an un-ordered array, starting with the central node.

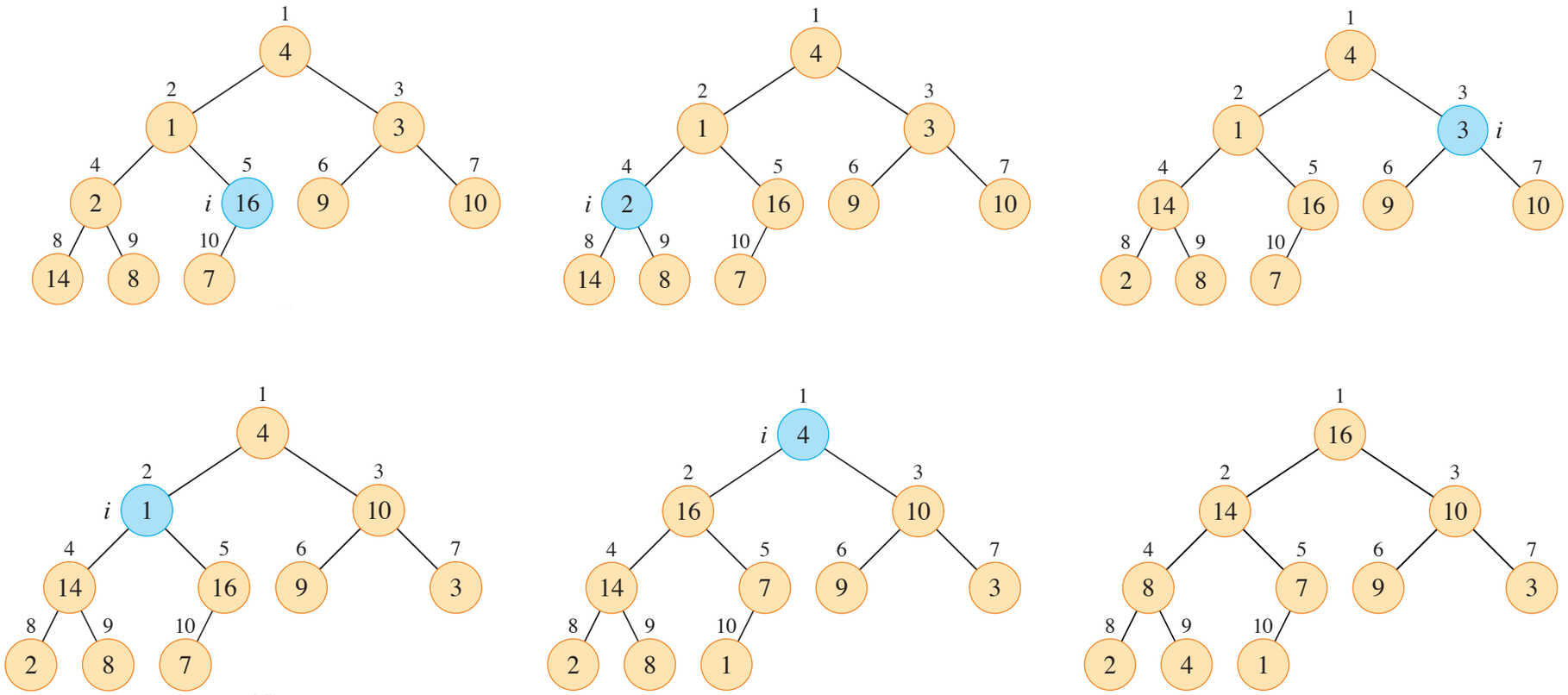


Call MAX-HEAPIFY*(A,3)*, make a swap with RIGHT

Continue...

Basic operations for Heaps

- The MAX-HEAPIFY(A, i), procedure can be used to build a heap from an un-ordered array, starting with the central node.



Finish with a max-heap.

Correctness of BUILD-MAX-HEAP

- The BUILD-MAX-HEAP(A,n), procedure satisfies the loop invariant:

Before each iteration, nodes $i+1,\ldots n$ act as root of a max-heap.

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

- **Initialization:**

Correctness of BUILD-MAX-HEAP

- The BUILD-MAX-HEAP(A,n), procedure satisfies the loop invariant:

Before each iteration, nodes $i+1,\ldots n$ act as root of a max-heap.

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

- **Initialization:**

  Nodes $\lfloor n/2 \rfloor, \ldots, n$ are leaves, thus max-heaps.

Correctness of BUILD-MAX-HEAP

- The BUILD-MAX-HEAP(A,n), procedure satisfies the loop invariant:

Before each iteration, nodes $i+1,\ldots n$ act as root of a max-heap.

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

- **Maintenance:**

Correctness of BUILD-MAX-HEAP

- The BUILD-MAX-HEAP(A,n), procedure satisfies the loop invariant:

Before each iteration, nodes $i+1,\ldots n$ act as root of a max-heap.

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

- **Maintenance:**

  When iteration at node i is called, it holds (from the previous iterations) that $i.left$ and $i.right$ are roots of max-heaps.

Correctness of BUILD-MAX-HEAP

- The BUILD-MAX-HEAP(A,n), procedure satisfies the loop invariant:

Before each iteration, nodes $i+1,...n$ act as root of a max-heap.

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

- **Maintenance:**

  When iteration at node i is called, it holds (from the previous iterations) that $i.left$ and $i.right$ are roots of max-heaps.

  Thus the heap, with based at node i, violate the max-heap property only at the root. When MAX-HEAPIFY is called on such a heap, it corrects the violation, while introducing no new ones.

Correctness of BUILD-MAX-HEAP

- The BUILD-MAX-HEAP(A,n), procedure satisfies the loop invariant:

Before each iteration, nodes $i+1,\ldots n$ act as root of a max-heap.

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

- **Maintenance:**

  When iteration at node i is called, it holds (from the previous iterations) that $i.left$ and $i.right$ are roots of max-heaps.

  Thus the heap, with based at node i, violate the max-heap property only at the root. When MAX-HEAPIFY is called on such a heap, it corrects the violation, while introducing no new ones.

  Thus, nodes $i,\ldots.,n$, are roots of max-heaps after the iteration.

Correctness of BUILD-MAX-HEAP

- The BUILD-MAX-HEAP(A,n), procedure satisfies the loop invariant:

Before each iteration, nodes $i+1,\ldots n$ act as root of a max-heap.

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

- **Termination:**

Correctness of BUILD-MAX-HEAP

- The BUILD-MAX-HEAP(A,n), procedure satisfies the loop invariant:

Before each iteration, nodes $i+1,\ldots n$ act as root of a max-heap.

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

- **Termination:**

  Terminates at $i=0$, with $i+1=1$ being the root of a max-heap, proving the whole heap is a max-heap.

Running time of BUILD-MAX-HEAP

- The BUILD-MAX-HEAP(A,n), procedure has linear running time.

Running time of BUILD-MAX-HEAP

- The BUILD-MAX-HEAP(A, i), procedure has linear running time.
- To prove it, we must analyze the running time of MAX-HEAPIFY, which depends on the height of the relevant node.

Running time of BUILD-MAX-HEAP

- The BUILD-MAX-HEAP(A, i), procedure has linear running time.
- To prove it, we must analyze the running time of MAX-HEAPIFY, which depends on the height of the relevant node.
    - What height does a heap of size n have?

Running time of BUILD-MAX-HEAP

- The BUILD-MAX-HEAP(A, i), procedure has linear running time.
- To prove it, we must analyze the running time of MAX-HEAPIFY, which depends on the height of the relevant node.
    - What height does a heap of size n have? $\lfloor lg(n) \rfloor$

Running time of BUILD-MAX-HEAP

- The BUILD-MAX-HEAP(A, i), procedure has linear running time.
- To prove it, we must analyze the running time of MAX-HEAPIFY, which depends on the height of the relevant node.
    - What height does a heap of size n have? $\lfloor lg(n) \rfloor$
    - What is the maximum amount of nodes at height h?

Running time of BUILD-MAX-HEAP

- The BUILD-MAX-HEAP(A, i), procedure has linear running time.
- To prove it, we must analyze the running time of MAX-HEAPIFY, which depends on the height of the relevant node.
    - What height does a heap of size n have? $\lfloor lg(n) \rfloor$
    - What is the maximum amount of nodes at height h?

$\lceil n/2 \rceil$ leaves (height 0)

Running time of BUILD-MAX-HEAP

- The BUILD-MAX-HEAP(A, i), procedure has linear running time.
- To prove it, we must analyze the running time of MAX-HEAPIFY, which depends on the height of the relevant node.
    - What height does a heap of size n have? $\lfloor lg(n) \rfloor$
    - What is the maximum amount of nodes at height h?

$\lceil n/2^2 \rceil$ nodes at height *1*

$\lceil n/2 \rceil$ leaves (height *0*)

Running time of BUILD-MAX-HEAP

- The BUILD-MAX-HEAP(A, i), procedure has linear running time.
- To prove it, we must analyze the running time of MAX-HEAPIFY, which depends on the height of the relevant node.
    - What height does a heap of size n have? $\lfloor lg(n) \rfloor$
    - What is the maximum amount of nodes at height h?

$\lceil n/2^3 \rceil$ nodes at height *2*

$\lceil n/2^2 \rceil$ nodes at height *1*

$\lceil n/2 \rceil$ leaves (height *0*)

Running time of BUILD-MAX-HEAP

- The BUILD-MAX-HEAP(A, i), procedure has linear running time.
- To prove it, we must analyze the running time of MAX-HEAPIFY, which depends on the height of the relevant node.
    - What height does a heap of size n have? $\lfloor lg(n) \rfloor$
    - What is the maximum amount of nodes at height h? $\lceil n/2^{h+1} \rceil$

$\lceil n/2^3 \rceil$ nodes at height *2*

$\lceil n/2^2 \rceil$ nodes at height *1*

$\lceil n/2 \rceil$ leaves (height *0*)

Running time of BUILD-MAX-HEAP

- The BUILD-MAX-HEAP(A, i), procedure has linear running time.
- To prove it, we must analyze the running time of MAX-HEAPIFY, which depends on the height of the relevant node.
    - What height does a heap of size n have? $\lfloor lg(n) \rfloor$
    - What is the maximum amount of nodes at height h? $\lceil n/2^{h+1} \rceil$
    - What is the cost of MAX-HEAPIFY at height h?

Running time of BUILD-MAX-HEAP

- The BUILD-MAX-HEAP(A, i), procedure has linear running time.
- To prove it, we must analyze the running time of MAX-HEAPIFY, which depends on the height of the relevant node.
    - What height does a heap of size n have? $\lfloor lg(n) \rfloor$
    - What is the maximum amount of nodes at height h? $\lceil n/2^{h+1} \rceil$
    - What is the cost of MAX-HEAPIFY at height h?

      h, will make at most one swap at each level

Running time of BUILD-MAX-HEAP

- The BUILD-MAX-HEAP(A, i), procedure has linear running time.
- To prove it, we must analyze the running time of MAX-HEAPIFY, which depends on the height of the relevant node.
    - What height does a heap of size n have? $\lfloor lg(n) \rfloor$
    - What is the maximum amount of nodes at height h? $\lceil n/2^{h+1} \rceil$
    - What is the cost of MAX-HEAPIFY at height h? h

Running time of BUILD-MAX-HEAP

- The BUILD-MAX-HEAP(A, i), procedure has linear running time.
- To prove it, we must analyze the running time of MAX-HEAPIFY, which depends on the height of the relevant node.
    - What height does a heap of size n have? $\lfloor lg(n) \rfloor$
    - What is the maximum amount of nodes at height h? $\lceil n/2^{h+1} \rceil$
    - What is the cost of MAX-HEAPIFY at height h? $c \cdot h$

$$\sum_{h=0}^{\lfloor lgn \rfloor} \lceil n/2^{h+1} \rceil c \cdot h \leq c \cdot n \sum_{h=0}^{\lfloor lgn \rfloor} \lceil h/2^h \rceil \leq c \cdot n \sum_{h=0}^{\infty} \lceil h/2^h \rceil$$

Running time of BUILD-MAX-HEAP

- The BUILD-MAX-HEAP(A, i), procedure has linear running time.
- To prove it, we must analyze the running time of MAX-HEAPIFY, which depends on the height of the relevant node.
    - What height does a heap of size n have? $\lfloor lg(n) \rfloor$
    - What is the maximum amount of nodes at height h? $\lceil n/2^{h+1} \rceil$
    - What is the cost of MAX-HEAPIFY at height h? $c \cdot h$

$$\sum_{h=0}^{\lfloor lgn \rfloor} \lceil n/2^{h+1} \rceil c \cdot h \leq c \cdot n \sum_{h=0}^{\lfloor lgn \rfloor} \lceil h/2^h \rceil \leq c \cdot n \sum_{h=0}^{\infty} \lceil h/2^h \rceil$$

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

Running time of BUILD-MAX-HEAP

- The BUILD-MAX-HEAP(A, i), procedure has linear running time.
- To prove it, we must analyze the running time of MAX-HEAPIFY, which depends on the height of the relevant node.
    - What height does a heap of size n have? $\lfloor lg(n) \rfloor$
        - What is the maximum amount of nodes at height h? $\lceil n/2^{h+1} \rceil$
        - What is the cost of MAX-HEAPIFY at height h? $c \cdot h$
- Proving we can build a max-heap in linear running time.

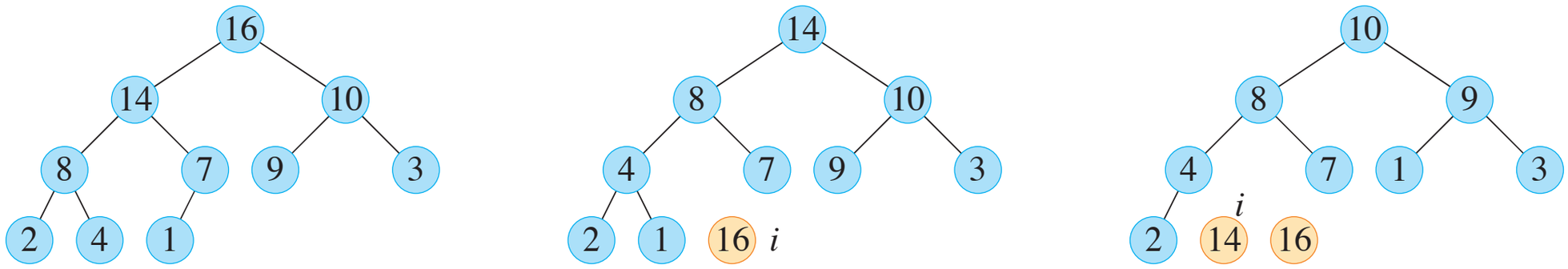
HEAPSORT

- The BUILD-MAX-HEAP, procedure can be used to sort an un-ordered array, in log-linear running time.

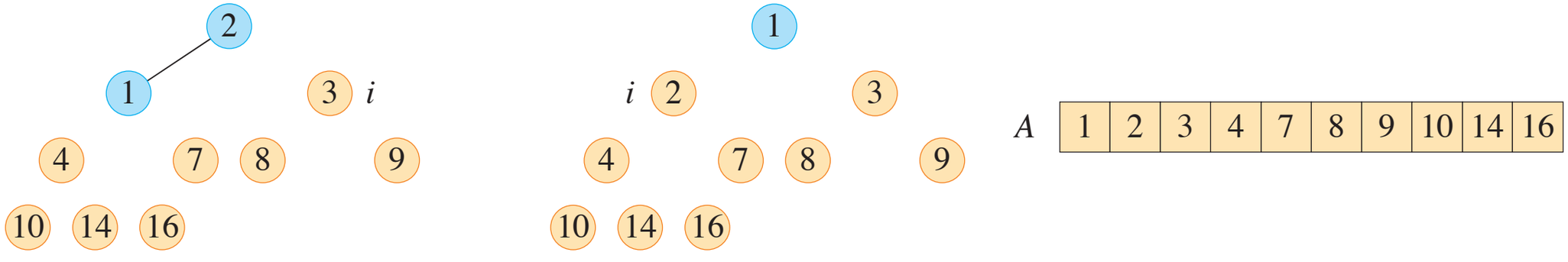
```
HEAPSORT(A, n)
1  BUILD-MAX-HEAP(A, n)
2  for i = n downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```

HEAPSORT

- The BUILD-MAX-HEAP, procedure can be used to sort an un-ordered array, in log-linear running time.



Continue...



Exercise

- Illustrate Heapsort on *[5,13,2,25,7,17,20,8,4]*

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

Exercise

- Illustrate Heapsort on *[5,13,2,25,7,17,20,8,4]*

```
MAX-HEAPIFY(A, i)
1   l = LEFT(i)
2   r = RIGHT(i)
3   if l ≤ A.heap-size and A[l] > A[i]
4       largest = l
5   else largest = i
6   if r ≤ A.heap-size and A[r] > A[largest]
7       largest = r
8   if largest ≠ i
9       exchange A[i] with A[largest]
10      MAX-HEAPIFY(A, largest)
```

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

Exercise

- Illustrate Heapsort on *[5,13,2,25,7,17,20,8,4]*

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

Correctness of HeapSort

- The HeapSort(A,n), procedure satisfies the loop invariant:

  At the start of iteration i:
  The subarray $A[1{:}i]$ is a max-heap
  containing the smallest elements.
  The subarray $A[i{+}1{:}n]$ contains the rest
  and is sorted.

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

Correctness of HEAPSORT

- The HEAPSORT(A,n), procedure satisfies the loop invariant:

  At the start of iteration i:
  The subarray $A[1{:}i]$ is a max-heap
  containing the smallest elements.
  The subarray $A[i{+}1{:}n]$ contains the rest
  and is sorted.

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

- **Initialization:**

Correctness of HEAPSORT

- The HEAPSORT(A,n), procedure satisfies the loop invariant:

  At the start of iteration i:
  The subarray $A[1{:}i]$ is a max-heap
  containing the smallest elements.
  The subarray $A[i{+}1{:}n]$ contains the rest
  and is sorted.

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

- **Initialization:**

  $A[1{:}n]$ is a max-heap (from initial BUILD-MAX-HEAP).

  $A[n{+}1{:}n]$ is empty and trivially sorted.

Correctness of HEAPSORT

- The HEAPSORT(A,n), procedure satisfies the loop invariant:

  At the start of iteration i:
  The subarray $A[1{:}i]$ is a max-heap
  containing the smallest elements.
  The subarray $A[i{+}1{:}n]$ contains the rest
  and is sorted.

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

- **Maintenance:**

Correctness of HEAPSORT

- The HEAPSORT(A,n), procedure satisfies the loop invariant:

  At the start of iteration i:
  The subarray $A[1{:}i]$ is a max-heap
  containing the smallest elements.
  The subarray $A[i{+}1{:}n]$ contains the rest
  and is sorted.

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

- **Maintenance:**
  $A[1{:}i]$ was initially a max-heap, with the smallest i elements.
  The largest of which, $A[1]$, was put in position, $A[i]$ so now $A[1{:}i{-}1]$ contains the smallest elements.
  The max heap property was violated at the root, but this was corrected, leaving $A[1{:}i{-}1]$ a max heap.
  Subarray $A[i{:}n] = A[i]$ followed by $A[i{+}1{:}n]$ now has the largest values.

Correctness of HeapSort

- The HeapSort(A,n), procedure satisfies the loop invariant:

  At the start of iteration i:
  The subarray $A[1{:}i]$ is a max-heap
  containing the smallest elements.
  The subarray $A[i{+}1{:}n]$ contains the rest
  and is sorted.

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

- **Termination:**

Correctness of HEAPSORT

- The HEAPSORT(A,n), procedure satisfies the loop invariant:

  At the start of iteration i:
  The subarray $A[1{:}i]$ is a max-heap
  containing the smallest elements.
  The subarray $A[i{+}1{:}n]$ contains the rest
  and is sorted.

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

- **Termination:**

  $A[1{:}1]$ is trivially a max-heap.

  $A[2{:}n]$ is sorted and contains the larger elements.

  Implying $A[1{:}n]$ is sorted.