CSE 105:
Data Structures and Algorithms-I
(Part 2)

Instructor

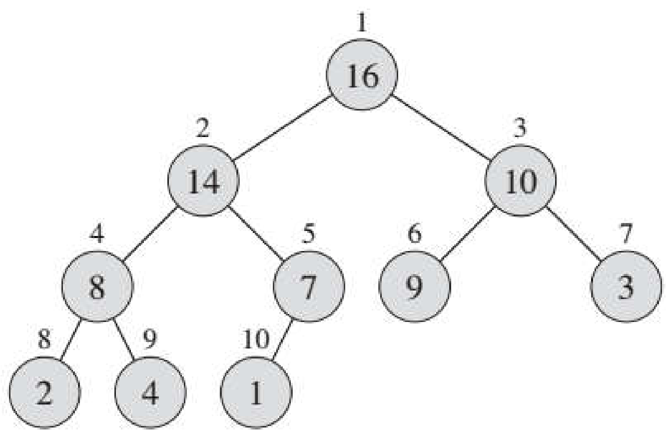
Dr Md Monirul Islam

Heap, Heapsort and Priority Queue

Example

Review

binary max-heap:



Review

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)
```

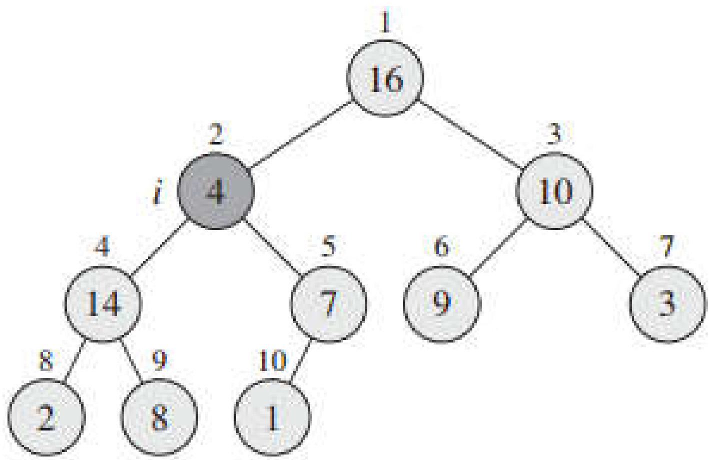
```
LEFT(i)
return 2i

RIGHT(i)
return 2i + 1
```

- Its inputs are an array A and an index i into the array.
- When it is called, MAX-HEAPIFY assumes
  - the binary trees rooted at LEFT(i) and RIGHT(i) are max-heaps,
  - but that $A[i]$ might be smaller than its children
    - thus violating the max-heap property.
- MAX-HEAPIFY lets the value at $A[i]$ "float down" in the max-heap so that the subtree rooted at index i obeys the max-heap property.

Review

MAX-HEAPIFY(A,2)

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

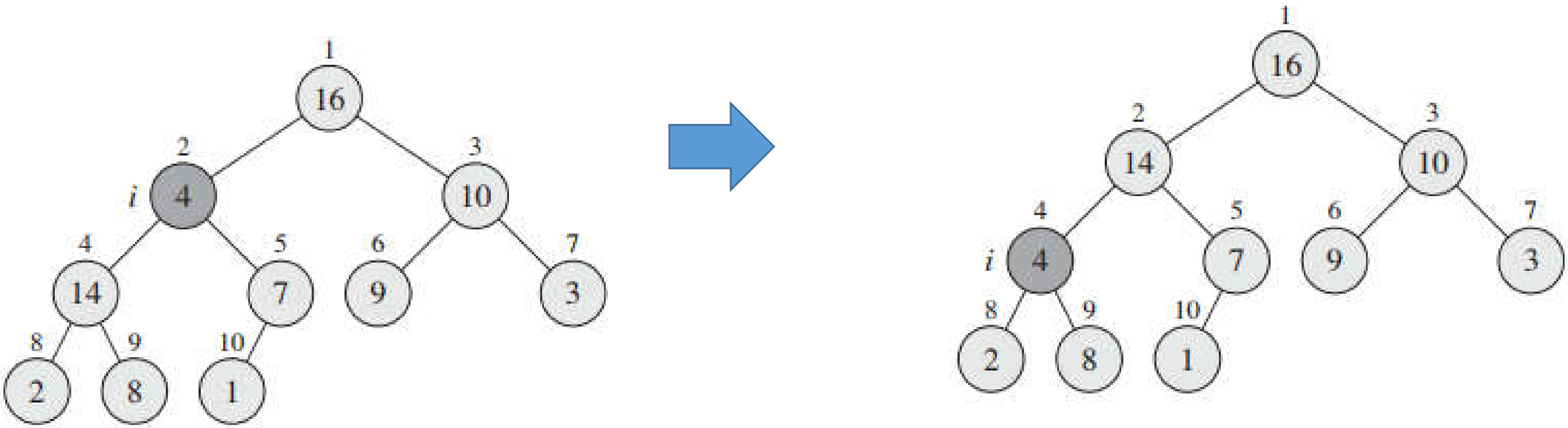



```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
    largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```

Review

MAX-HEAPIFY(A,2)

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

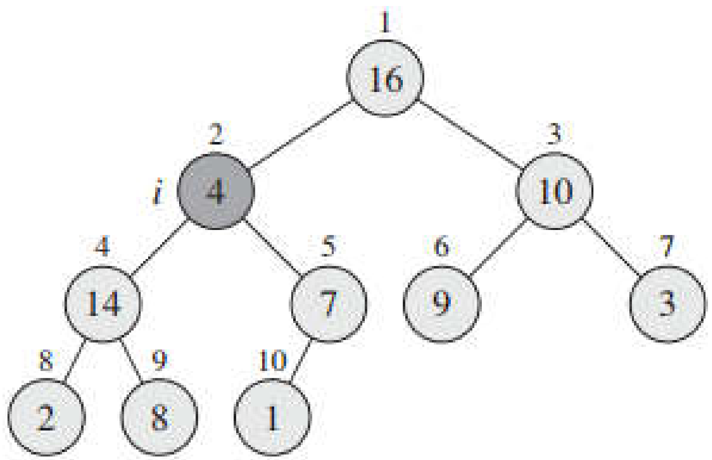


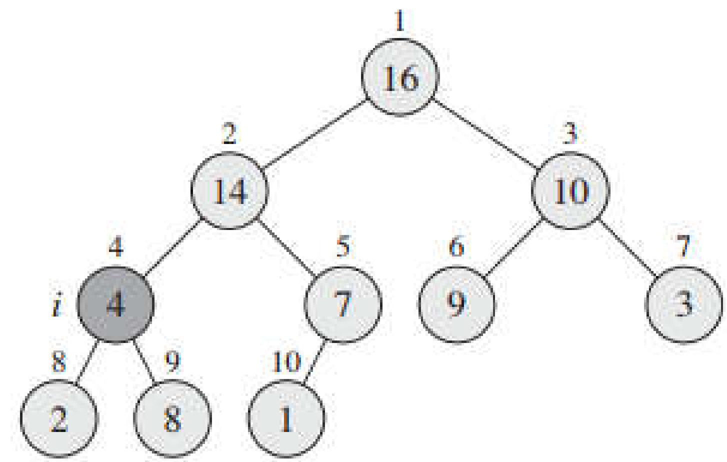
```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
    largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```

Review

MAX-HEAPIFY(*A*, 4)

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



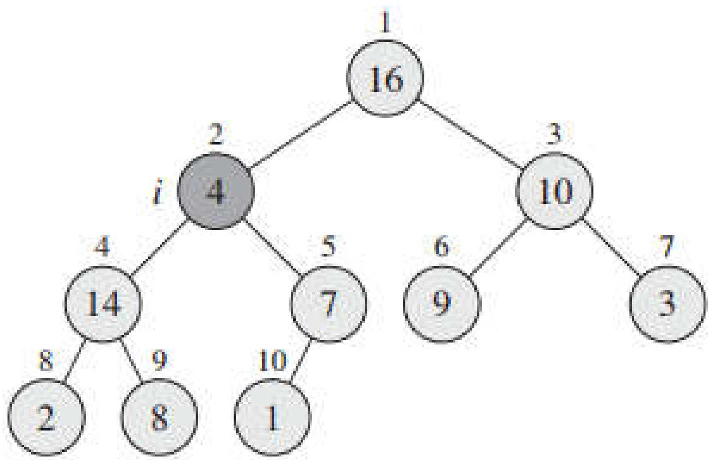


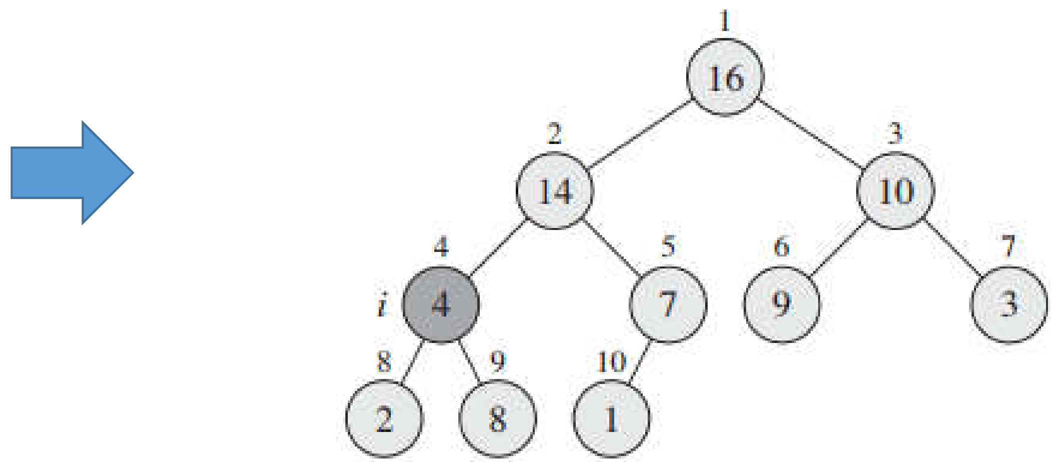

```
MAX-HEAPIFY(A, i)
  l = LEFT(i)
  r = RIGHT(i)
  if l ≤ A.heap-size and A[l] > A[i]
      largest = l
  else largest = i
  if r ≤ A.heap-size and A[r] > A[largest]
      largest = r
  if largest ≠ i
      exchange A[i] with A[largest]
      MAX-HEAPIFY(A, largest)
```

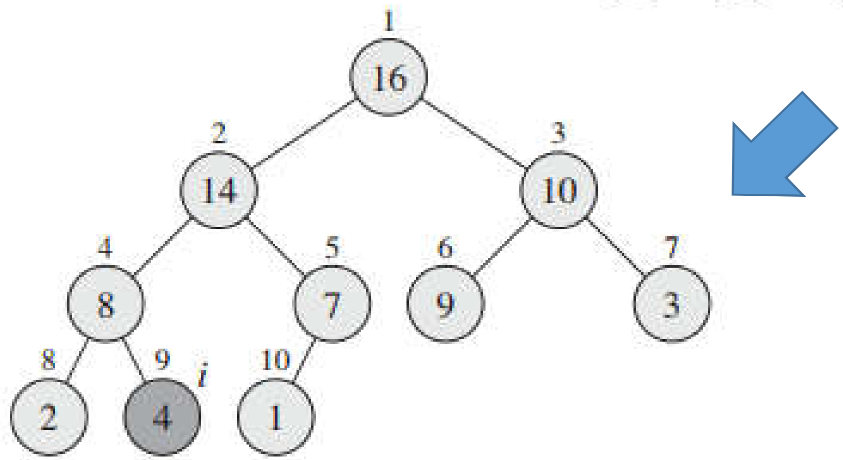
Review

MAX-HEAPIFY(A, 4)

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







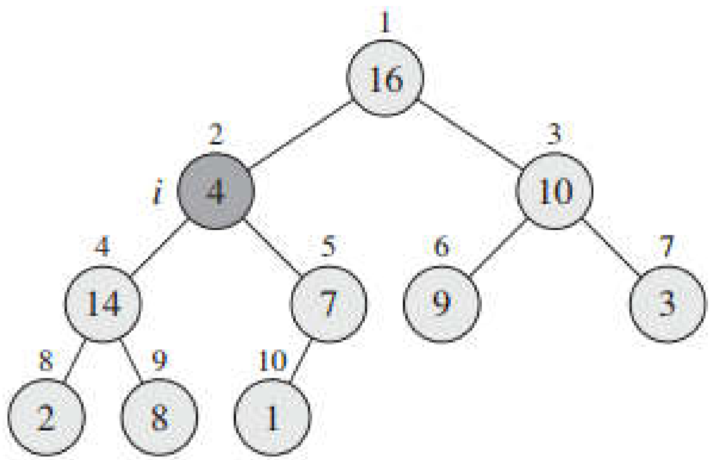
MAX-HEAPIFY(A, i)

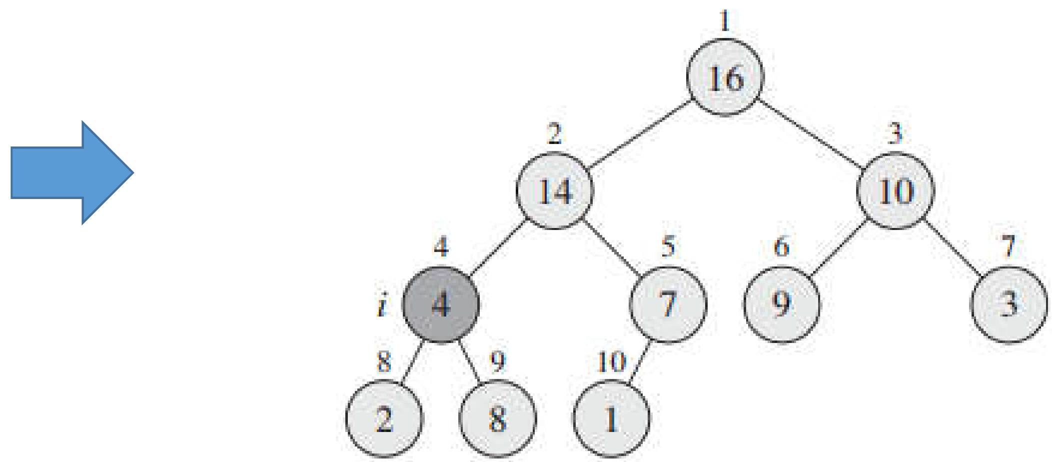
```
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
    largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```

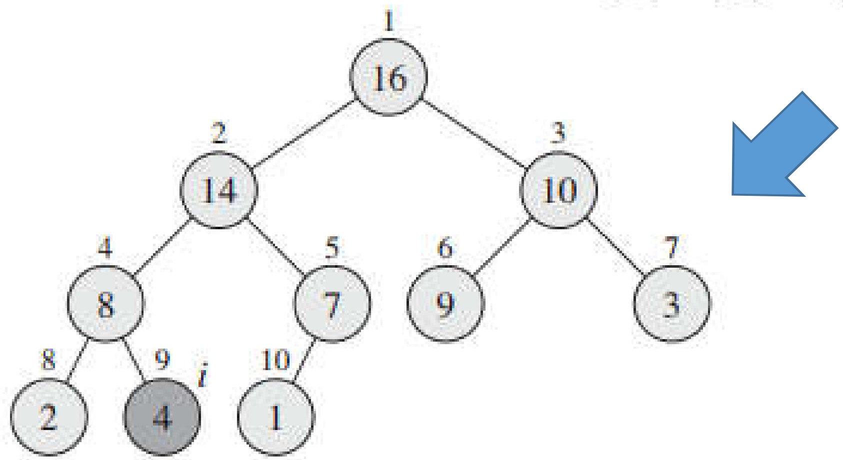
Review

MAX-HEAPIFY(A, 9)

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








```
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)
```

Review

(MAX/MIN)-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)
```

$\Theta(1)$

$$T(n) \le T\left(\frac{2n}{3}\right) + \Theta(1)$$

$\Rightarrow$ T(n) = O(log n)

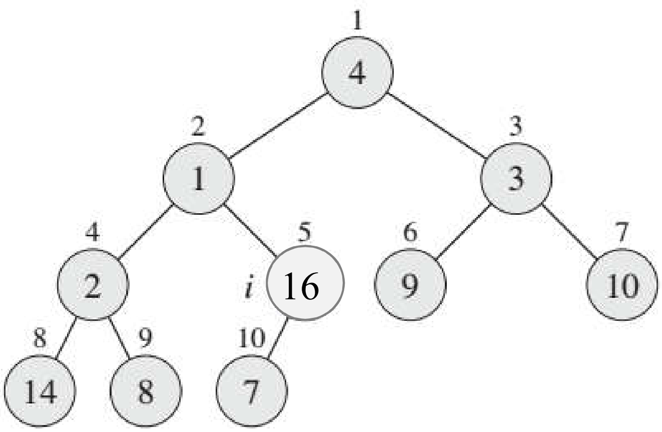
Master Theorem

BUILD-(MAX/MIN)-HEAP

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

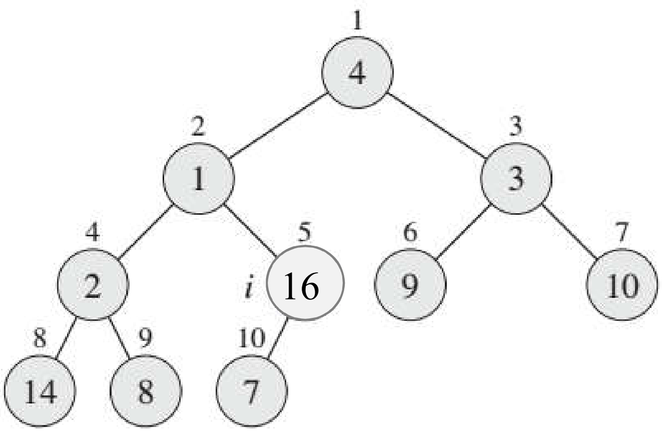
BUILD-MAXHEAP

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



BUILD-MAXHEAP

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

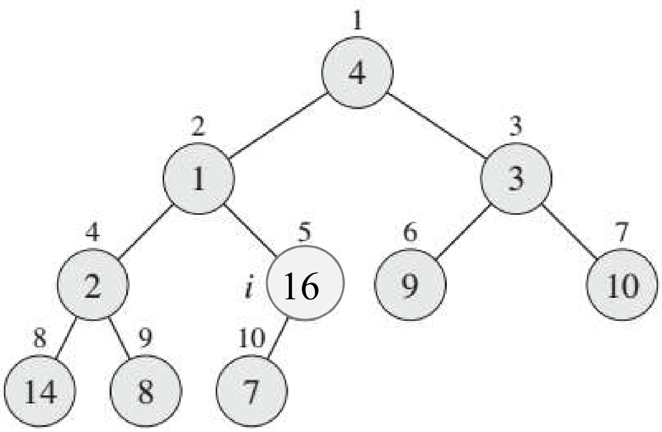


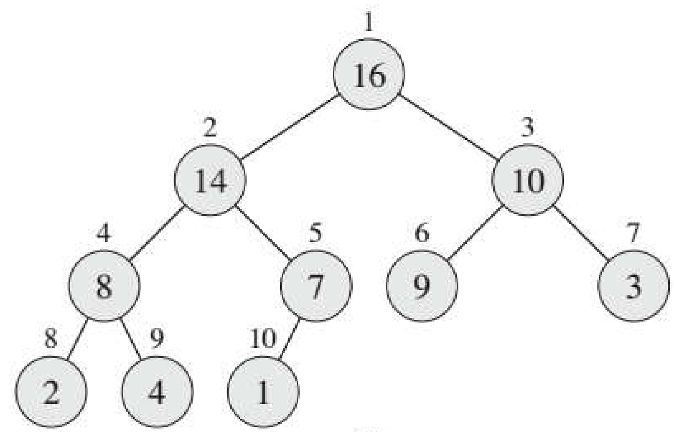
- We can use the procedure MAX-HEAPIFY in a bottom-up manner to convert an array into a max-heap.

BUILD-MAXHEAP

- We can use the procedure MAX-HEAPIFY in a bottom-up manner to convert an array into a max-heap.

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





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

BUILD-MAXHEAP

BUILD-MAX-HEAP(A)

A.*heap-size* = A.*length*

for i = $\lfloor$A.*length*/2$\rfloor$ **downto** 1

    MAX-HEAPIFY(A, i)

- We can use the procedure MAX-HEAPIFY in a bottom-up manner to convert an array into a max-heap.

BUILD-(MAX/MIN)-HEAP

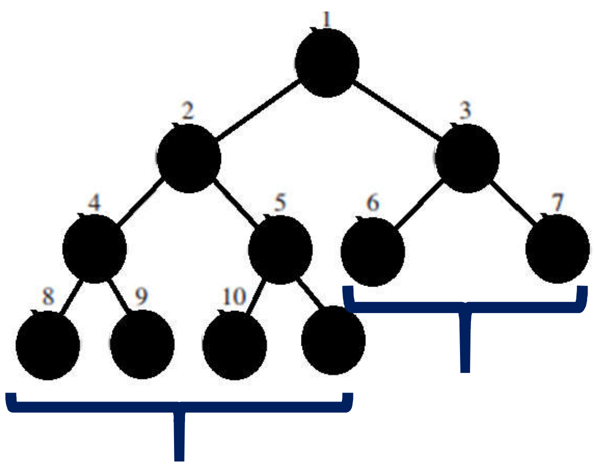
BUILD-MAX-HEAP(A)

A.*heap-size* = A.*length*

for i = $\lfloor$A.*length*/2$\rfloor$ **downto** 1

    MAX-HEAPIFY(A, i)

n = 11
n/2 + 1 = 6
∴ 6 onward all are leaves

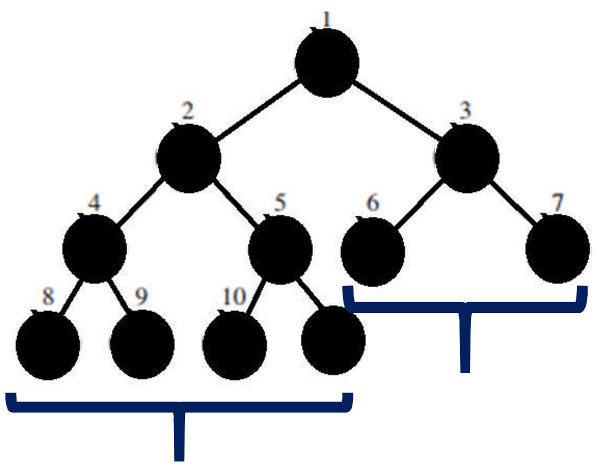


- We can use the procedure MAX-HEAPIFY in a bottom-up manner to convert an array into a max-heap.
- The elements in the subarray $A[\lfloor n/2 \rfloor + 1], \ldots, A[n]$ are all leaves.
  - Each is a 1-element heap.

BUILD-(MAX/MIN)-HEAP

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

n = 11
n/2 + 1 = 6
∴ 6 onward all are leaves

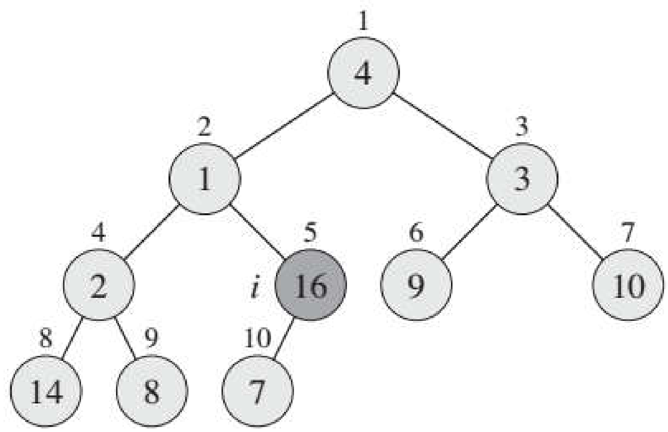


- We can use the procedure MAX-HEAPIFY in a bottom-up manner to convert an array into a max-heap.
- The elements in the subarray $A[\lfloor n/2 \rfloor + 1], \ldots, A[n]$ are all leaves.
  - Each is a 1-element heap.
- BUILD-MAX-HEAP goes through the remaining nodes *upward* and call MAX-HEAPIFY on each one.

BUILD-MAX-HEAP

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

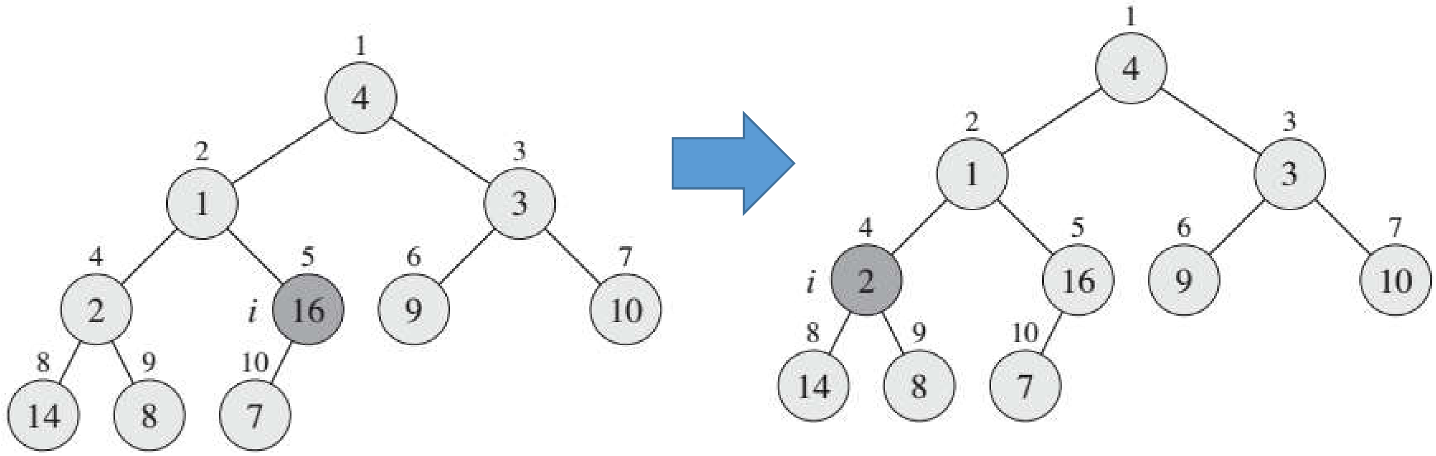
```
BUILD-MAX-HEAP(A)
A.heap-size = A.length
for i = ⌊A.length/2⌋ downto 1
        MAX-HEAPIFY(A, i)
```



BUILD-MAX-HEAP

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

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



BUILD-MAX-HEAP

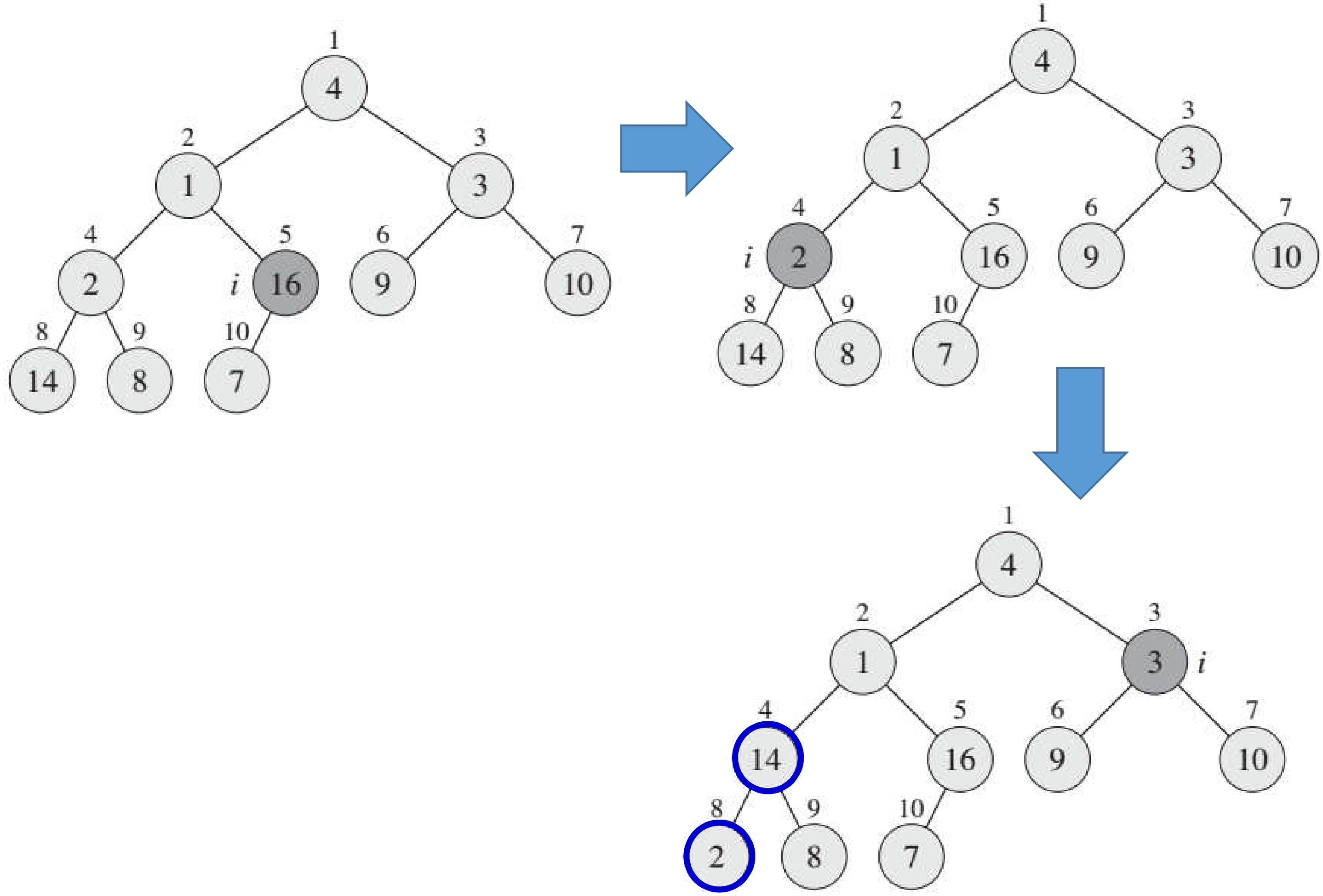
1	2	3	**4**	5	6	7	**8**	9	10
4	1	3	**14**	16	9	10	**2**	8	7

BUILD-MAX-HEAP(A)

A.*heap-size* = A.*length*

for i = ⌊A.*length*/2⌋ **downto** 1

MAX-HEAPIFY(A, i)



BUILD-MAX-HEAP

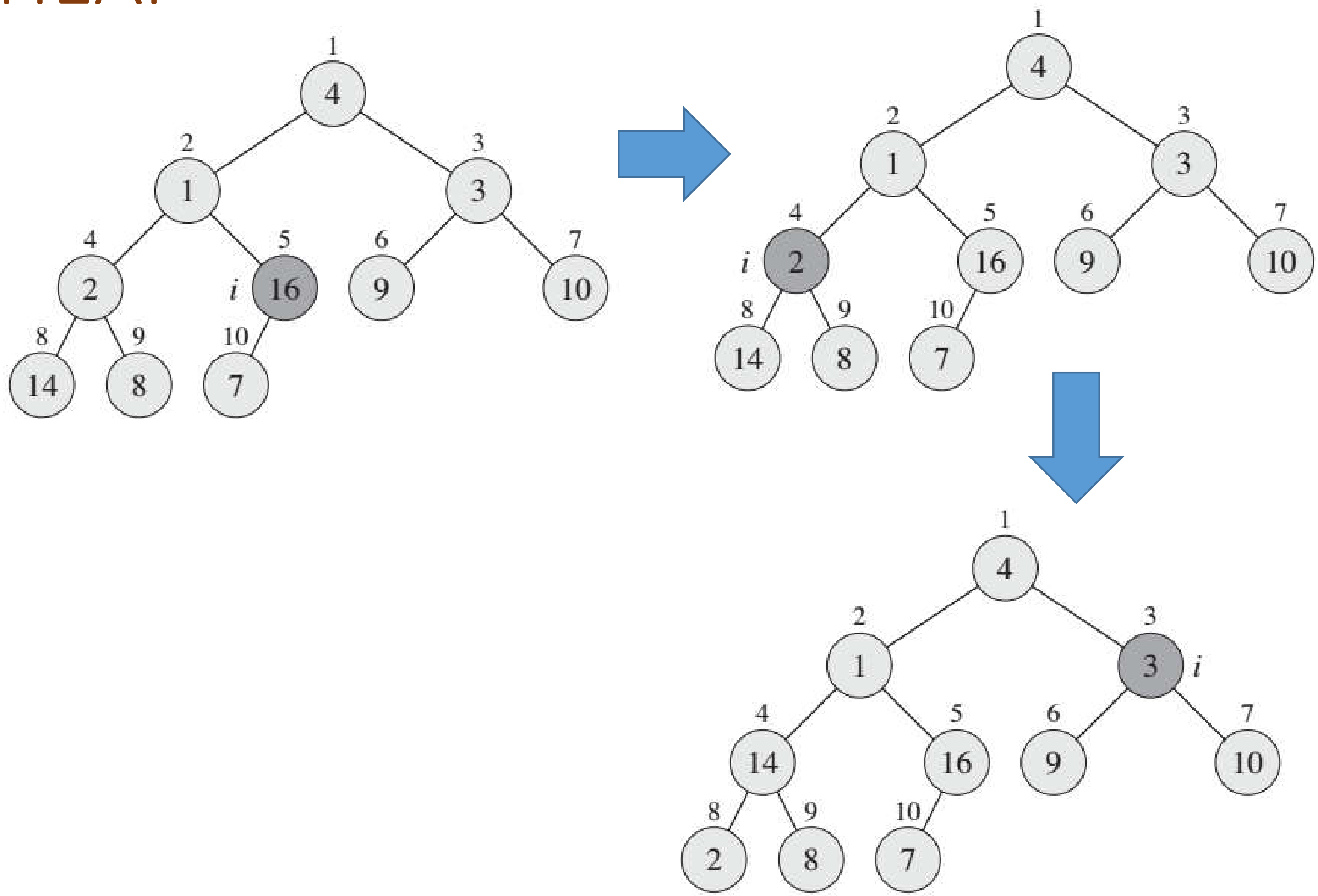
1	2	3	4	5	6	7	8	9	10
4	1	3	14	16	9	10	2	8	7

BUILD-MAX-HEAP(A)

A.*heap-size* = A.*length*

for i = ⌊A.*length*/2⌋ **downto** 1

MAX-HEAPIFY(A, i)



BUILD-MAX-HEAP

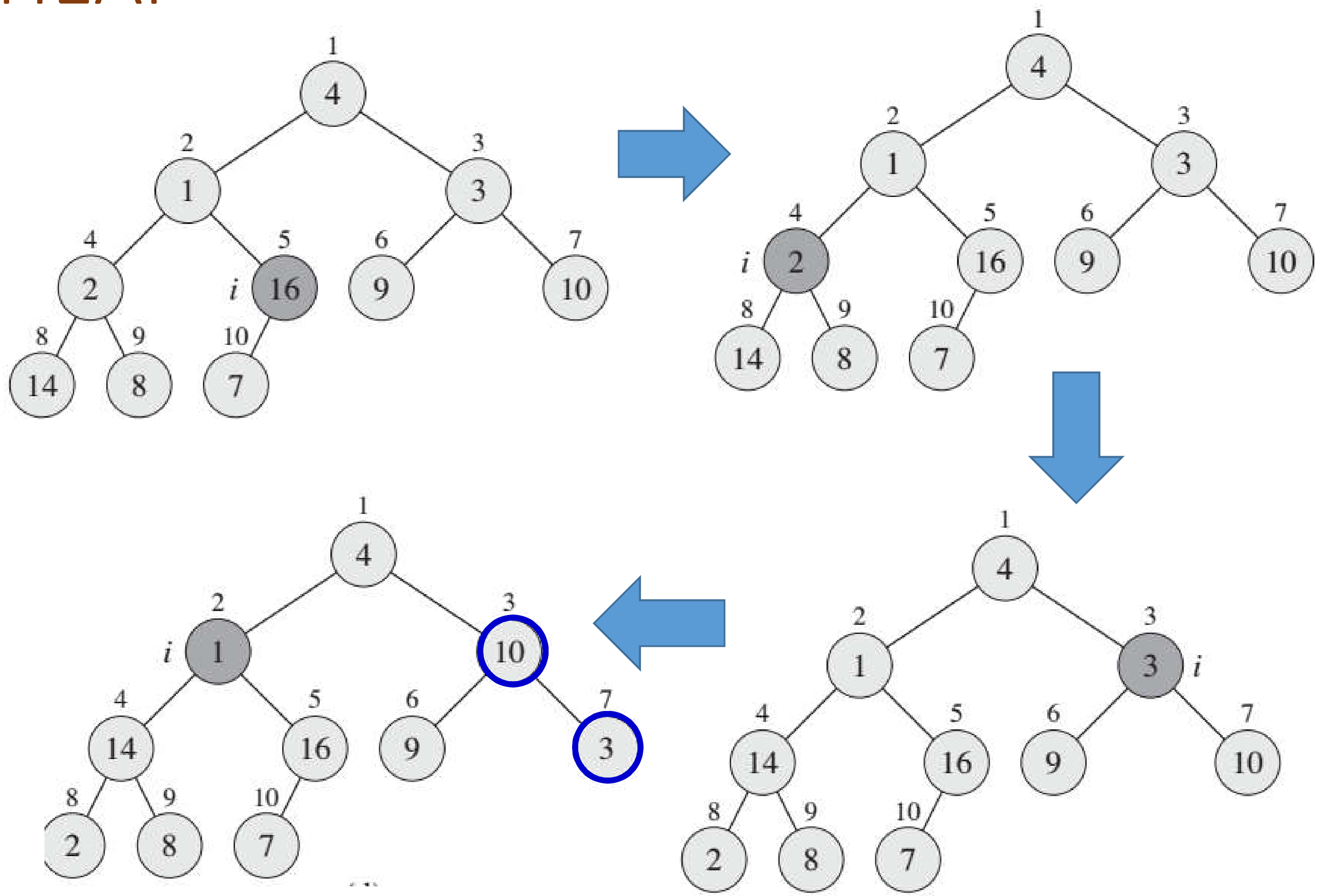
1	2	**3**	4	5	6	**7**	8	9	10
4	1	**10**	14	16	9	**3**	2	8	7

BUILD-MAX-HEAP(A)

A.*heap-size* = A.*length*

for i = ⌊A.*length*/2⌋ **downto** 1

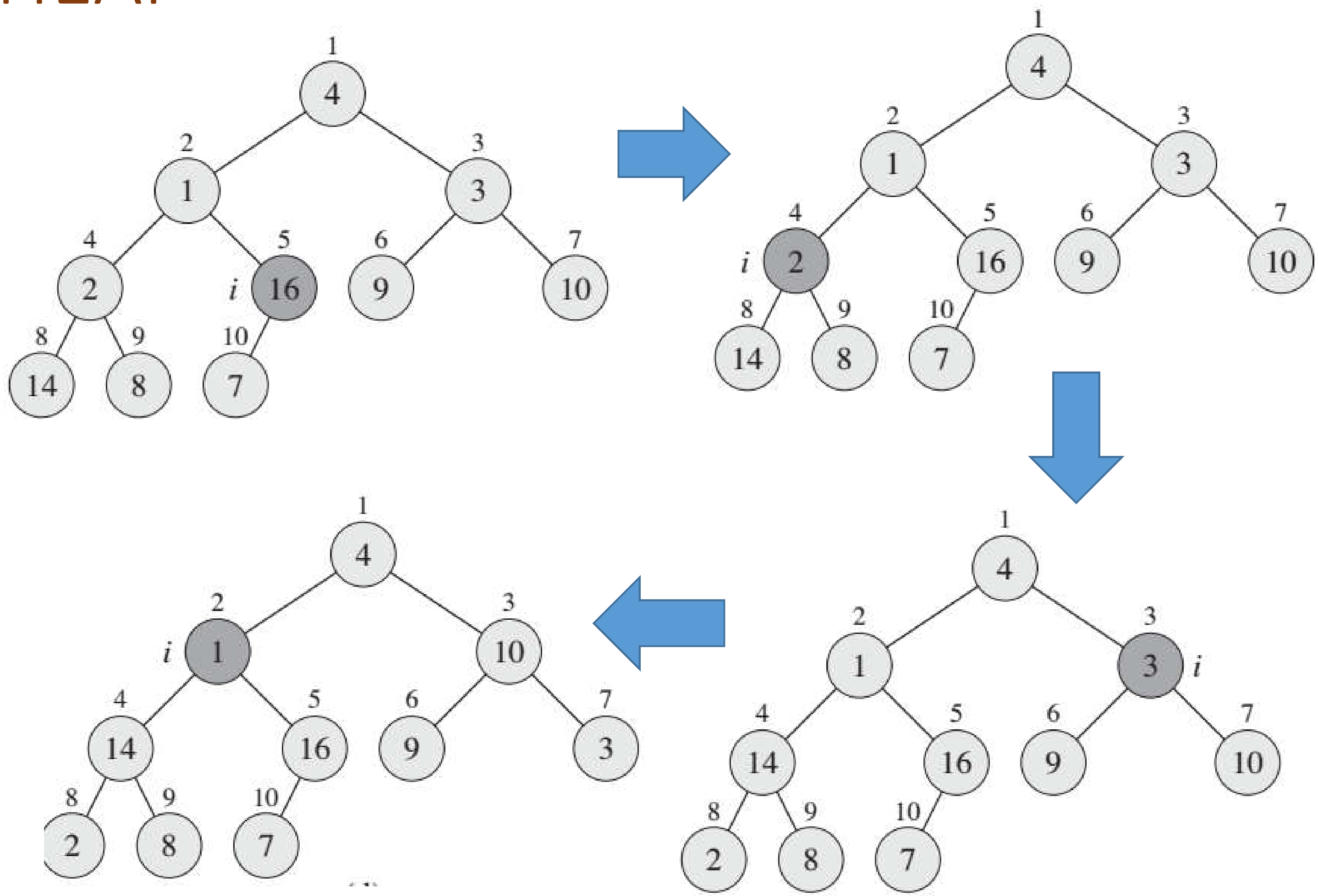
MAX-HEAPIFY(A, i)



BUILD-MAX-HEAP

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

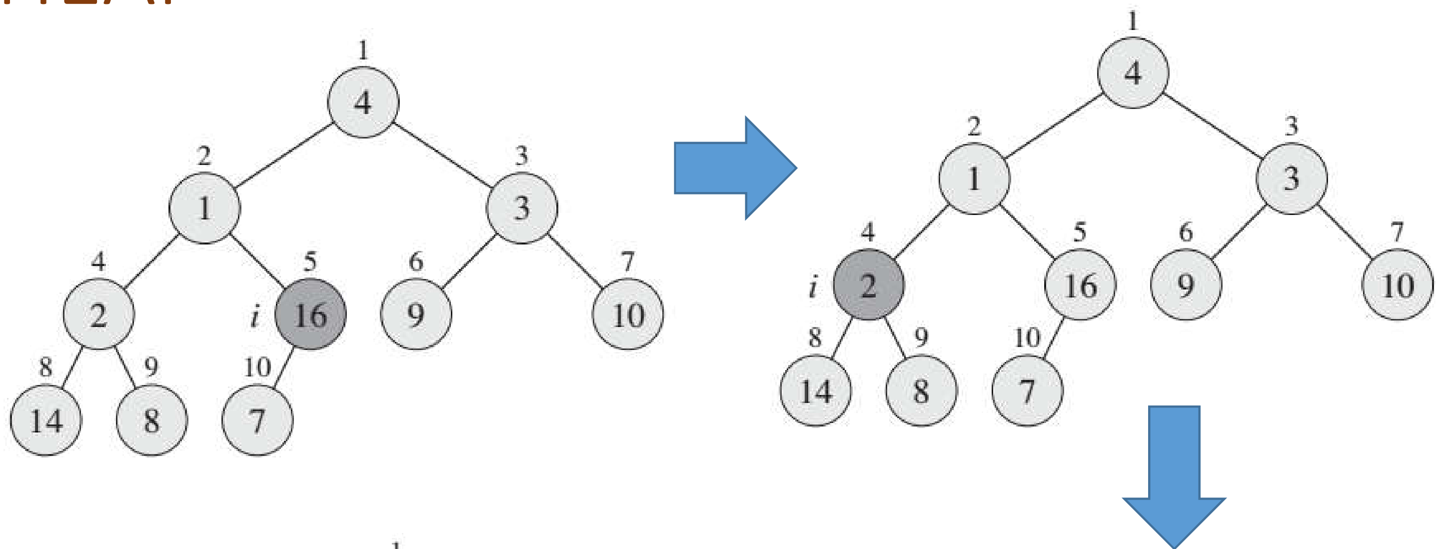
```
BUILD-MAX-HEAP(A)
A.heap-size = A.length
for i = ⌊A.length/2⌋ downto 1
        MAX-HEAPIFY(A, i)
```

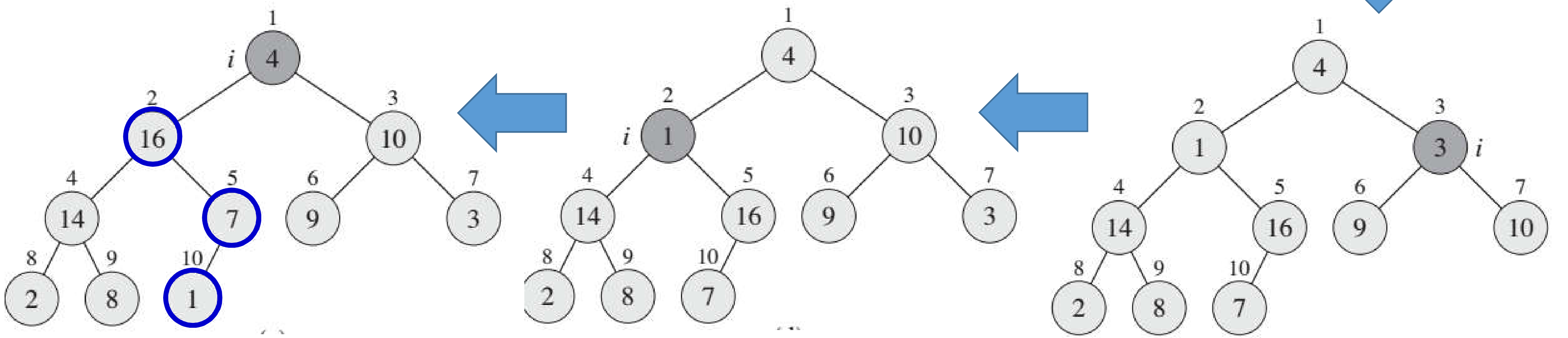


BUILD-MAX-HEAP

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

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

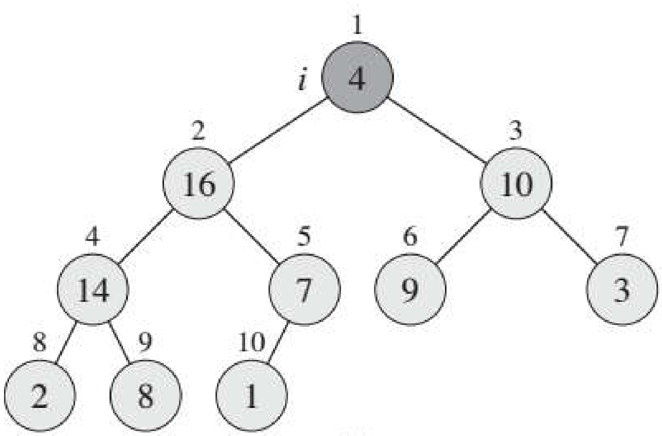




BUILD-MAX-HEAP

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

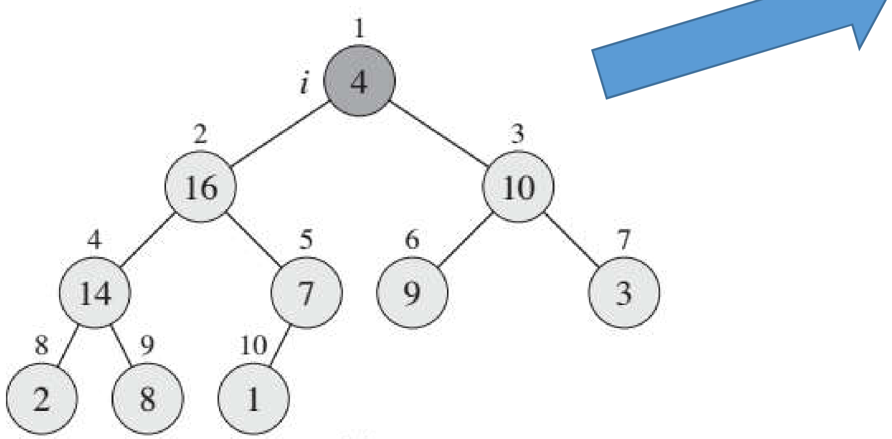
```
BUILD-MAX-HEAP(A)
A.heap-size = A.length
for i = ⌊A.length/2⌋ downto 1
        MAX-HEAPIFY(A, i)
```

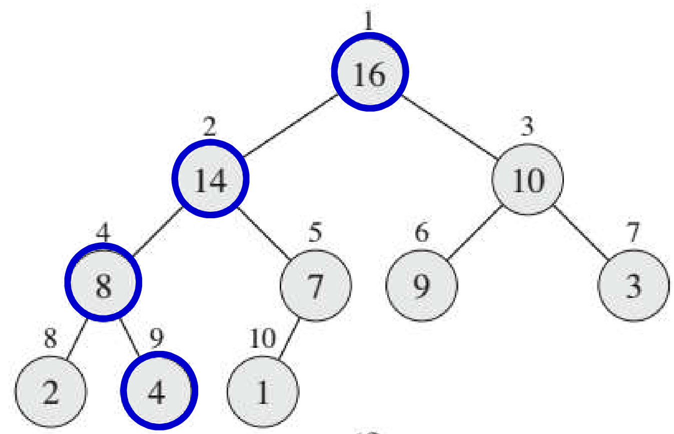


BUILD-MAX-HEAP

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

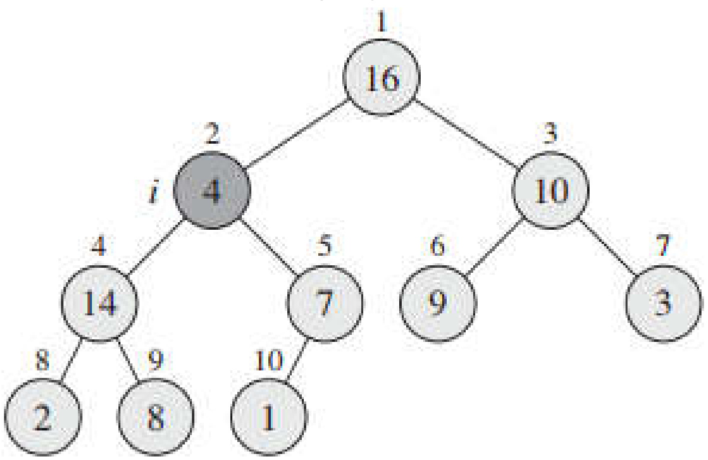
```
BUILD-MAX-HEAP(A)
A.heap-size = A.length
for i = ⌊A.length/2⌋ downto 1
        MAX-HEAPIFY(A, i)
```





BUILD-(MAX/MIN)-HEAP: Correctness

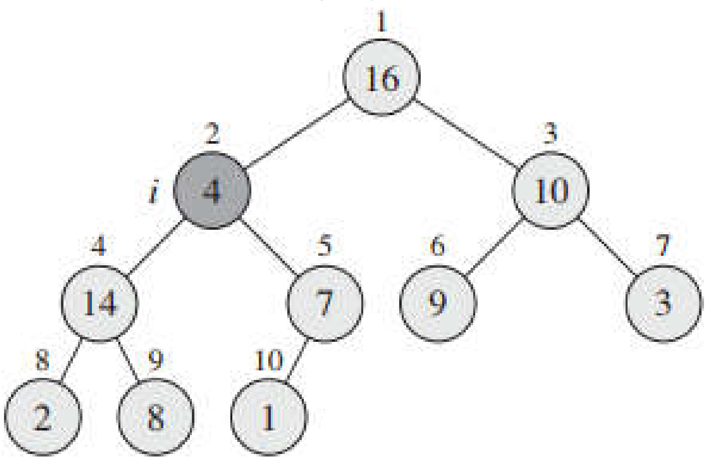
```
BUILD-MAX-HEAP(A)
A.heap-size = A.length
for i = ⌊A.length/2⌋ downto 1
        MAX-HEAPIFY(A, i)
```



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

BUILD-(MAX/MIN)-HEAP: Correctness

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



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

Observe the following loop invariant

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

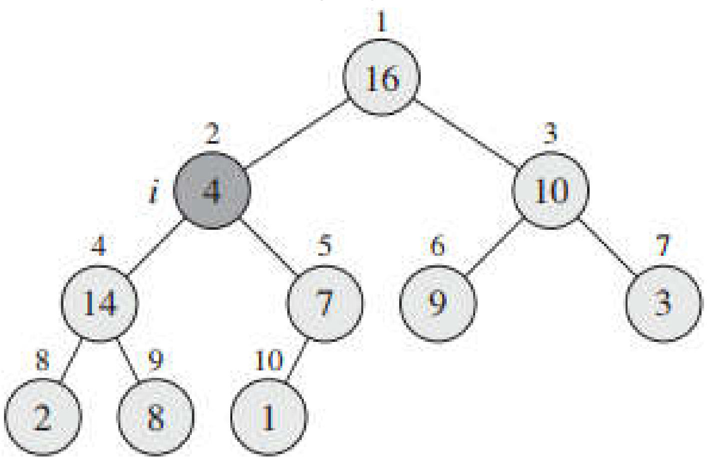
BUILD-(MAX/MIN)-HEAP: Correctness

BUILD-MAX-HEAP(A)

1 A.*heap-size* = A.*length*

2 **for** i = $\lfloor$A.*length*/2$\rfloor$ **downto** 1

3 MAX-HEAPIFY(A, i)



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

Observe the following loop invariant

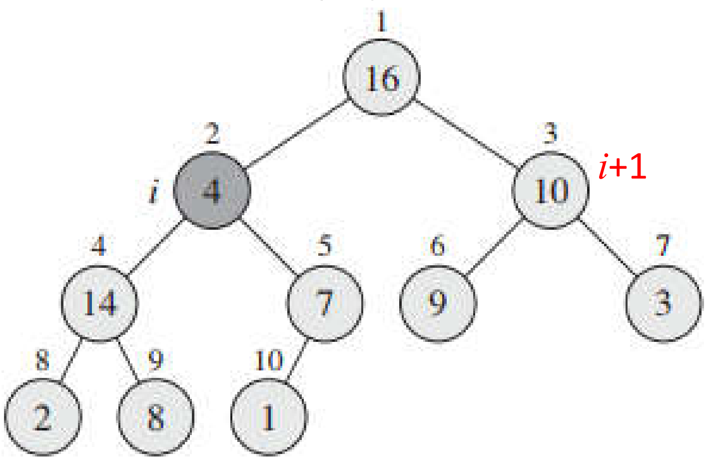
At the **start** of each iteration of the **for loop** of lines 2-3, each node $i+1$, $i+2$, …, n is the root of a max-heap

At Initialization

$i = \lfloor n/2 \rfloor$: all nodes afterwards, e.g., $\lfloor n/2 \rfloor +1$, …, n are leaf-nodes, therefore, root of max-heaps.

BUILD-(MAX/MIN)-HEAP: Correctness

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



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

Observe the following loop invariant

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

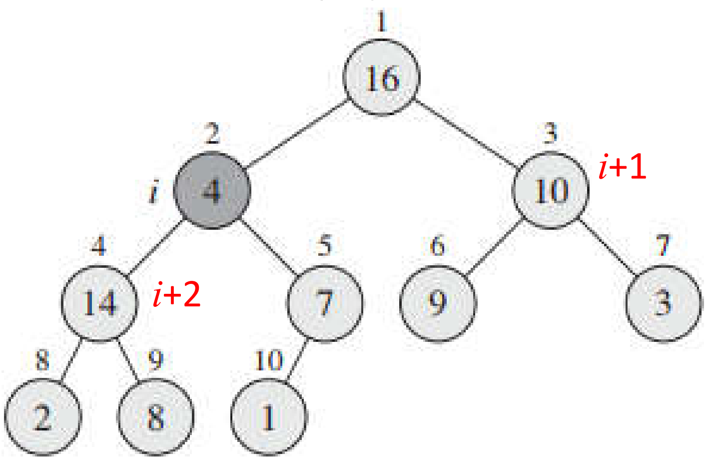
At Maintenance Steps

Let, it's true for $i+1$.

We have to show that it is true for i.

BUILD-(MAX/MIN)-HEAP: Correctness

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



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

Observe the following loop invariant

At the start of each iteration of the **for** loop of lines 2-3, each node i+1, i+2, …, n is the root of a max-heap

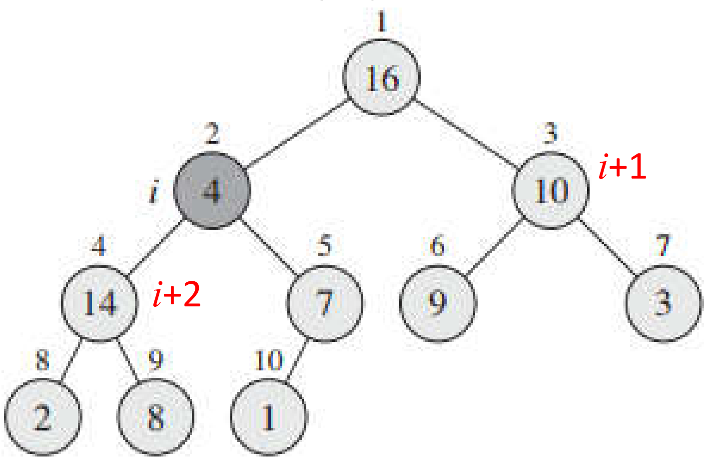
At Maintenance Steps

Let, it's true for i +1.

All nodes from i + 2 to n are roots of max-heaps.

BUILD-(MAX/MIN)-HEAP: Correctness

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



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

Observe the following loop invariant

At the start of each iteration of the **for** loop of lines 2-3, each node i+1, i+2, …, n is the root of a max-heap

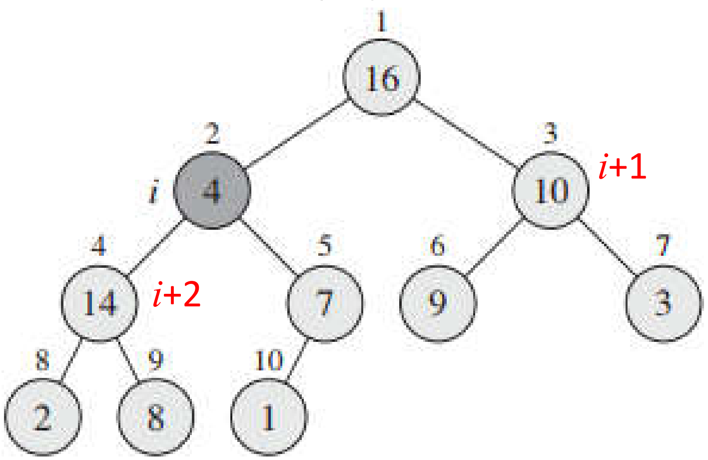
At Maintenance Steps

Let, it's true for i +1.
All nodes from i + 2 to n are roots of max-heaps.

Children of node i+1 are **higher than** i+1
e.g., They are in i + 2 to n

BUILD-(MAX/MIN)-HEAP: Correctness

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



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

Observe the following loop invariant

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

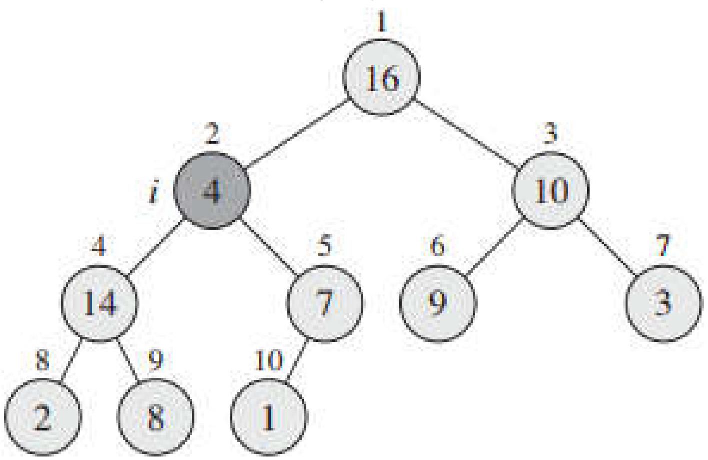
At Maintenance Steps

Let, it's true for $i+1$.
All nodes from $i+2$ to n are roots of max-heaps.

Therefore, max-heapify will correctly make node i+1 a root of a max-heap

BUILD-(MAX/MIN)-HEAP: Correctness

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



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

Observe the following loop invariant

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

At Maintenance Steps

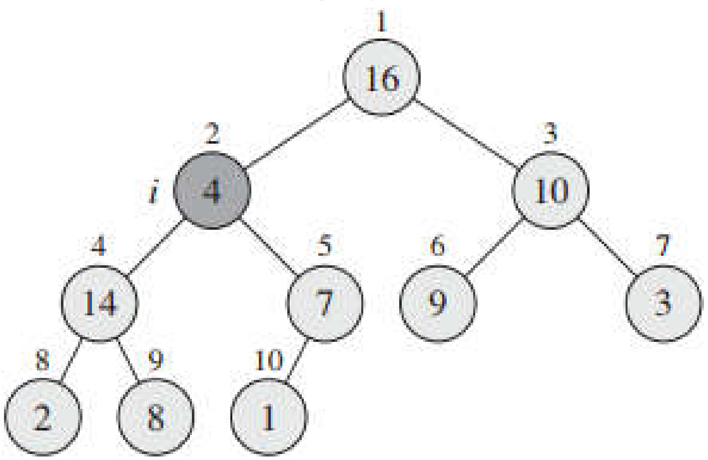
Let, it's true for $i+1$.
All nodes from $i+2$ to n are roots of max-heaps.

After that iteration, node $i+1$ will be root of a max-heap.

Therefore, all nodes from $i+1$ to n are roots of max-heaps.

BUILD-(MAX/MIN)-HEAP: Correctness

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



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

Observe the following loop invariant

At the start of each iteration of the **for** loop of lines 2-3, each node i+1, i+2, …, n is the root of a max-heap

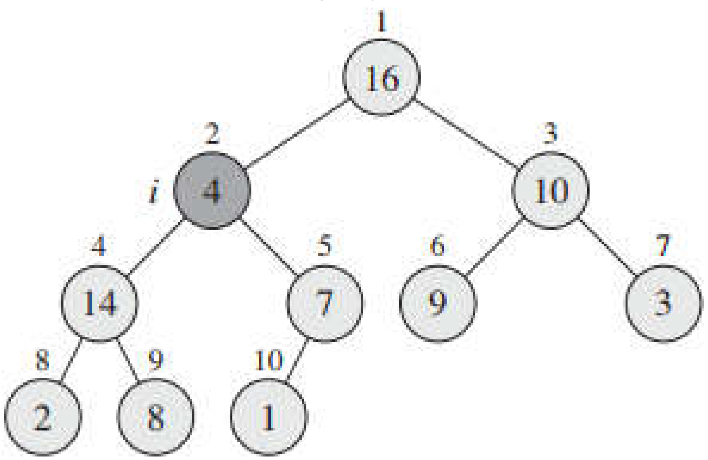
At Maintenance Steps

Let, it's true for i +1.

We get: All nodes from i + 1 to n are root of max-heaps.

BUILD-(MAX/MIN)-HEAP: Correctness

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



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

Observe the following loop invariant

At the start of each iteration of the **for** loop of lines 2-3, each node i+1, i+2, …, n is the root of a max-heap

At Maintenance Steps

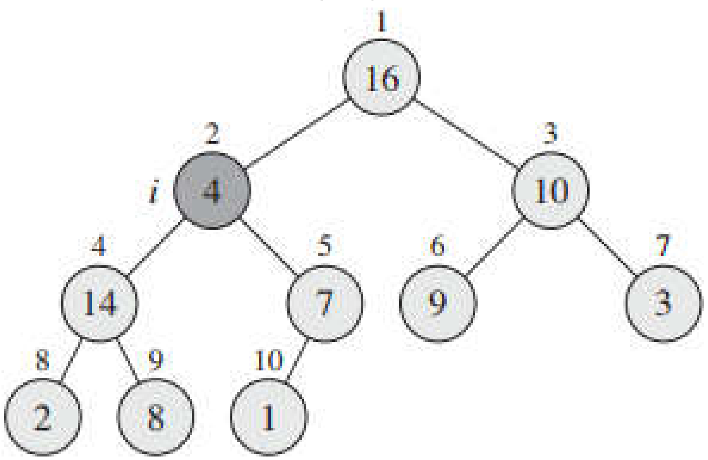
Let, it's true for i +1.

We get: All nodes from i + 1 to n are root of max-heaps.

Therefore, at the start of next iteration (e.g., at i) each node i+1, i+2, …, n is the root of a max-heap

BUILD-(MAX/MIN)-HEAP: Correctness

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



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

Observe the following loop invariant

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

After Termination

Value of $i = 0$.

All nodes from 1 to n are roots of max-heaps.

BUILD-(MAX/MIN)-HEAP: Running Time (Simple)

BUILD-MAX-HEAP(A)

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

BUILD-(MAX/MIN)-HEAP: Running Time (Tighter)

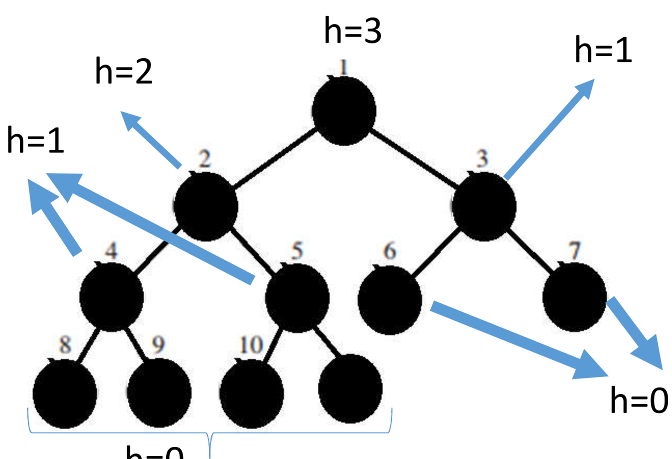
BUILD-MAX-HEAP(A)

1 A.*heap-size* = A.*length*

2 **for** i = $\lfloor$A.*length*/2$\rfloor$ **downto** 1

3 MAX-HEAPIFY(A, i)

- an n-element heap has height $\lfloor \log n \rfloor$
- There are at most $\lceil n/2^{h+1} \rceil$ nodes at any height h.
- Here height is the longest distance from a leaf
  - Somewhat opposite to depth



BUILD-(MAX/MIN)-HEAP: Running Time (Tighter)

BUILD-MAX-HEAP(A)

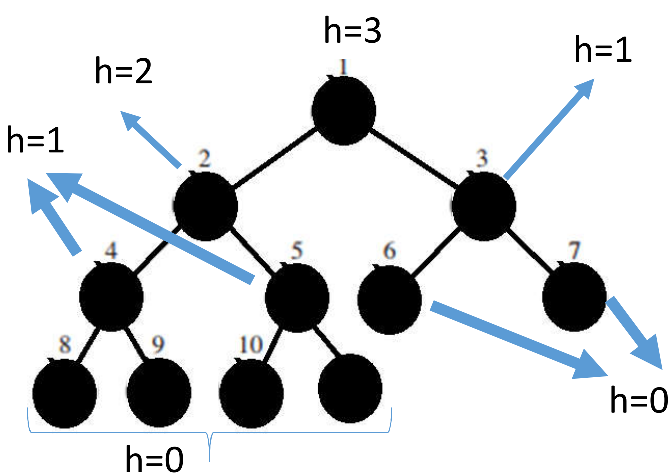
A.*heap-size* = A.*length*

for i = $\lfloor$A.*length*/2$\rfloor$ **downto** 1

MAX-HEAPIFY(A, i)

- an n-element heap has height $\lfloor \log n \rfloor$
- There are at most $\lceil n/2^{h+1} \rceil$ nodes at any height h.
- Here height is the longest distance from a leaf
    - Somewhat opposite to depth

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



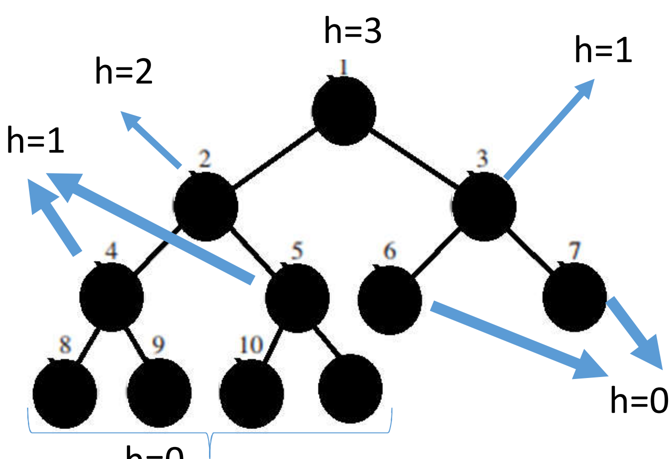
BUILD-(MAX/MIN)-HEAP: Running Time (Tighter)

BUILD-MAX-HEAP(A)

A.*heap-size* = A.*length*

for i = $\lfloor$A.*length*/2$\rfloor$ **downto** 1

  MAX-HEAPIFY(A, i)



- an n-element heap has height $\lfloor \log n \rfloor$
- There are at most $\lceil n/2^{h+1} \rceil$ nodes at any height h.
- Here height is the longest distance from a leaf
  - Somewhat opposite to depth

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

We know, asymptotically

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$$

BUILD-(MAX/MIN)-HEAP: Running Time (Tighter)

BUILD-MAX-HEAP(A)

A.*heap-size* = A.*length*

for i = $\lfloor$A.*length*/2$\rfloor$ **downto** 1

    MAX-HEAPIFY(A, i)

- an n-element heap has height $\lfloor \log n \rfloor$
- There are at most $\lceil n/2^{h+1} \rceil$ nodes at any height h.
- Here height is the longest distance from a leaf
  - Somewhat opposite to depth

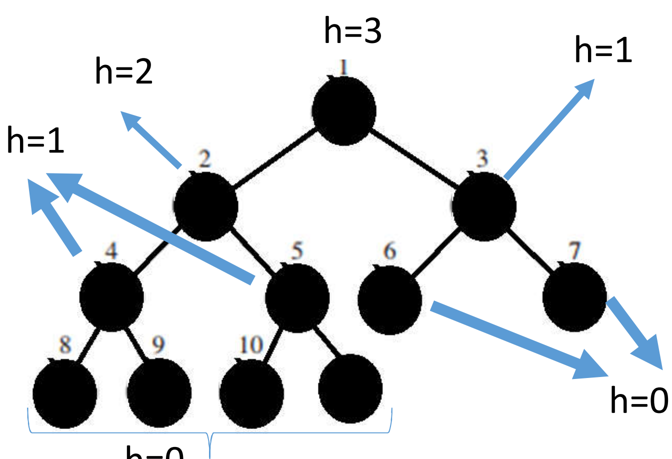
Complexity = $\displaystyle\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right)$

We know, asymptotically

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$$

Therefore,

$$\text{Complexity} = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) = O(n)\,.$$



Heapsort

4	1	3	2	16	9	10	14	8	7

Heapsort

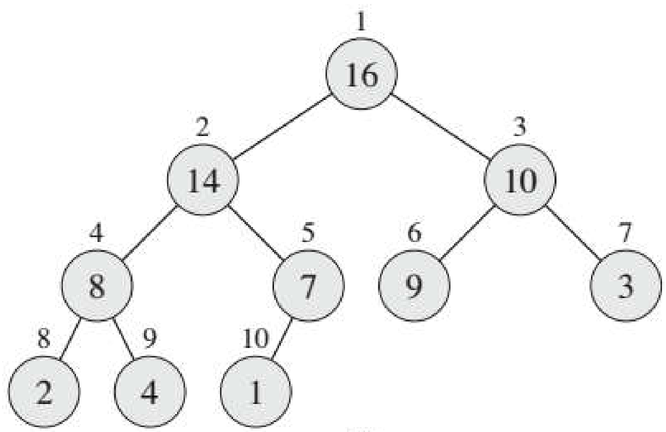
1	2	3	4	7	8	9	10	14	16

Heapsort

4	1	3	2	16	9	10	14	8	7

BuildMaxHeap

16	14	10	8	7	9	3	2	4	1



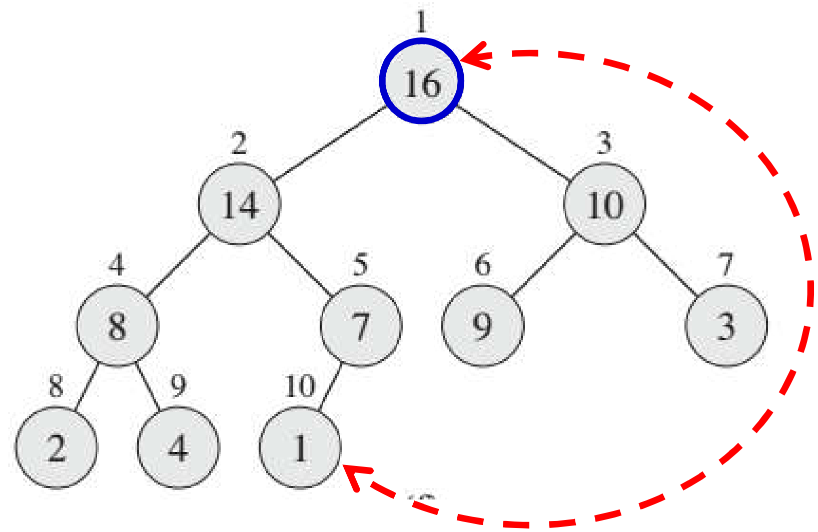
sort

1	2	3	4	7	8	9	10	14	16

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

Heapsort

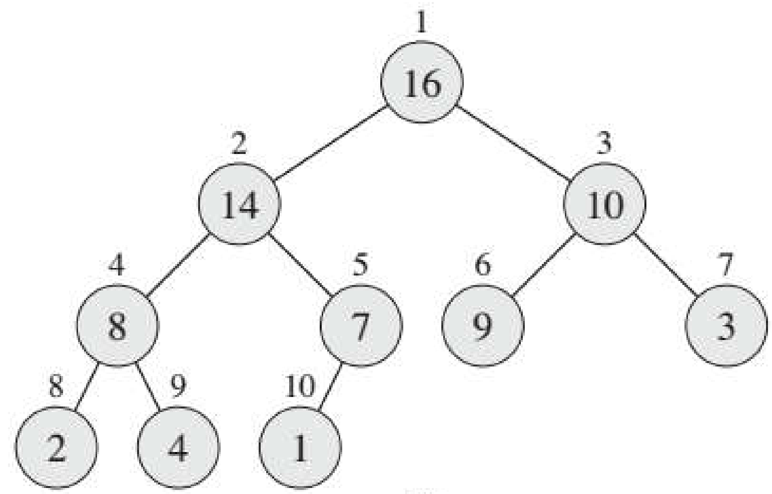
- Given **BuildHeap()**, a sorting algorithm can easily be constructed:
  - Maximum element is at $A[1]$
  - Discard by swapping with element at $A[n]$
    - Decrease heap_size[A]
    - $A[n]$ now contains correct value
  - Restore heap property at A[1] by calling **Heapify()**
  - Repeat, always swapping A[1] for A[heap_size(A)]



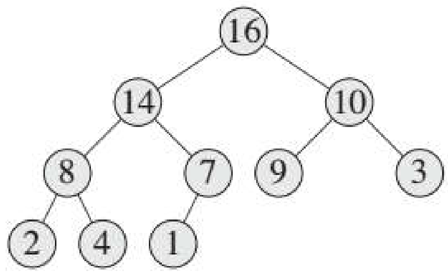
Heapsort

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

```
Heapsort(A) {
  BuildHeap(A);
  for (i = length(A) downto 2) {
        Swap(A[1], A[i]);
        heap_size(A) = heap_size(A) – 1;
        Heapify(A, 1);
  }
}
```

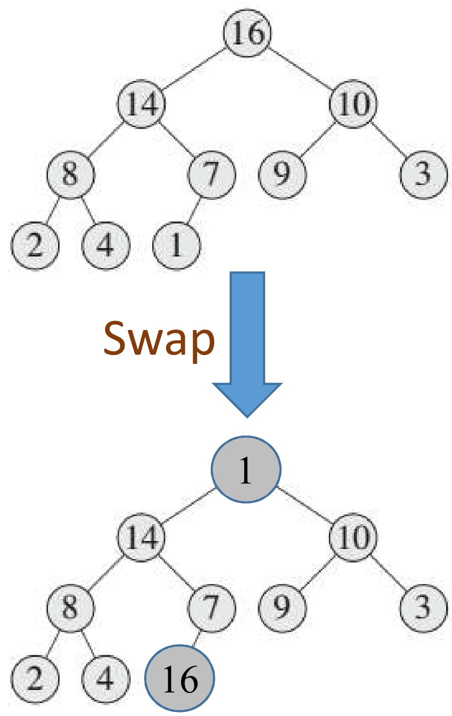


Heapsort Example



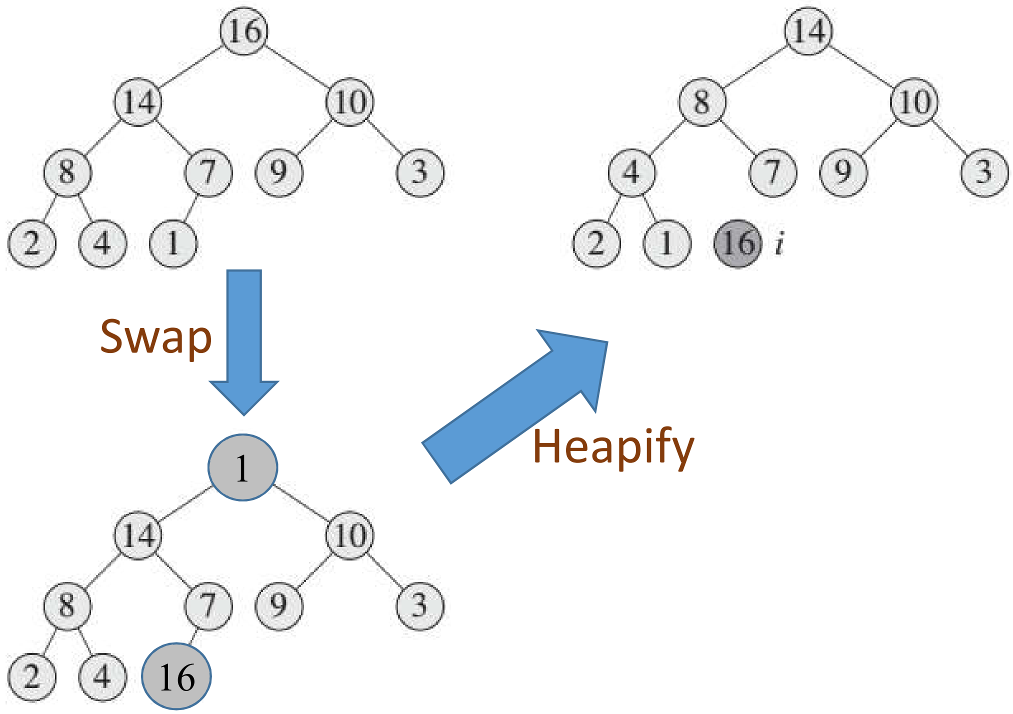
1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

Heapsort Example



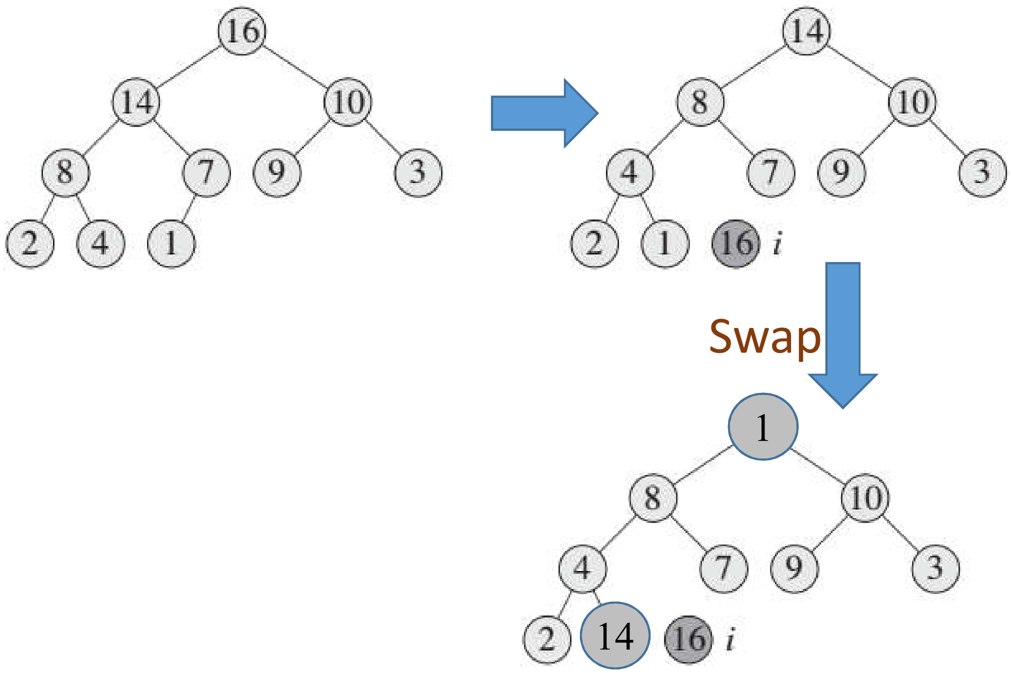
1	2	3	4	5	6	7	8	9	10
1	14	10	8	7	9	3	2	4	16

Heapsort Example



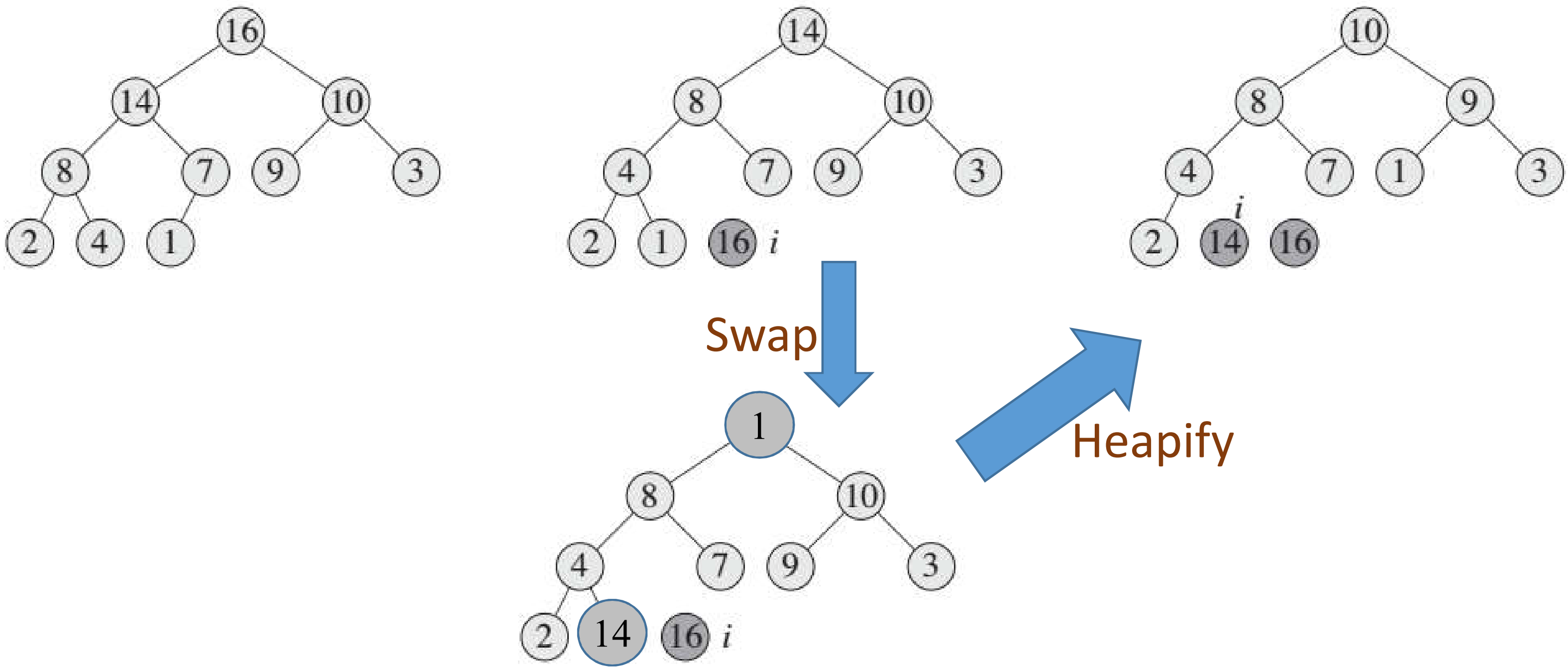
1	2	3	4	5	6	7	8	9	10
14	8	10	4	7	9	3	2	1	16

Heapsort Example



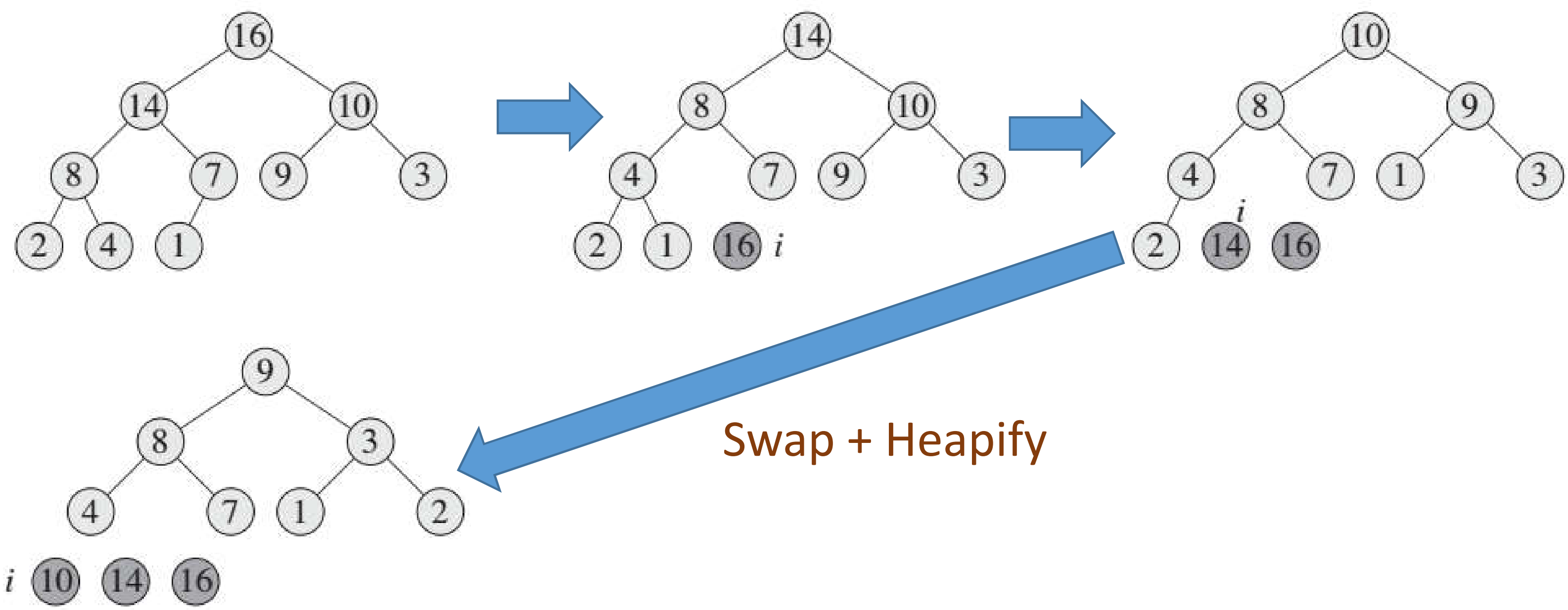
1	2	3	4	5	6	7	8	9	10
1	8	10	4	7	9	3	2	14	16

Heapsort Example



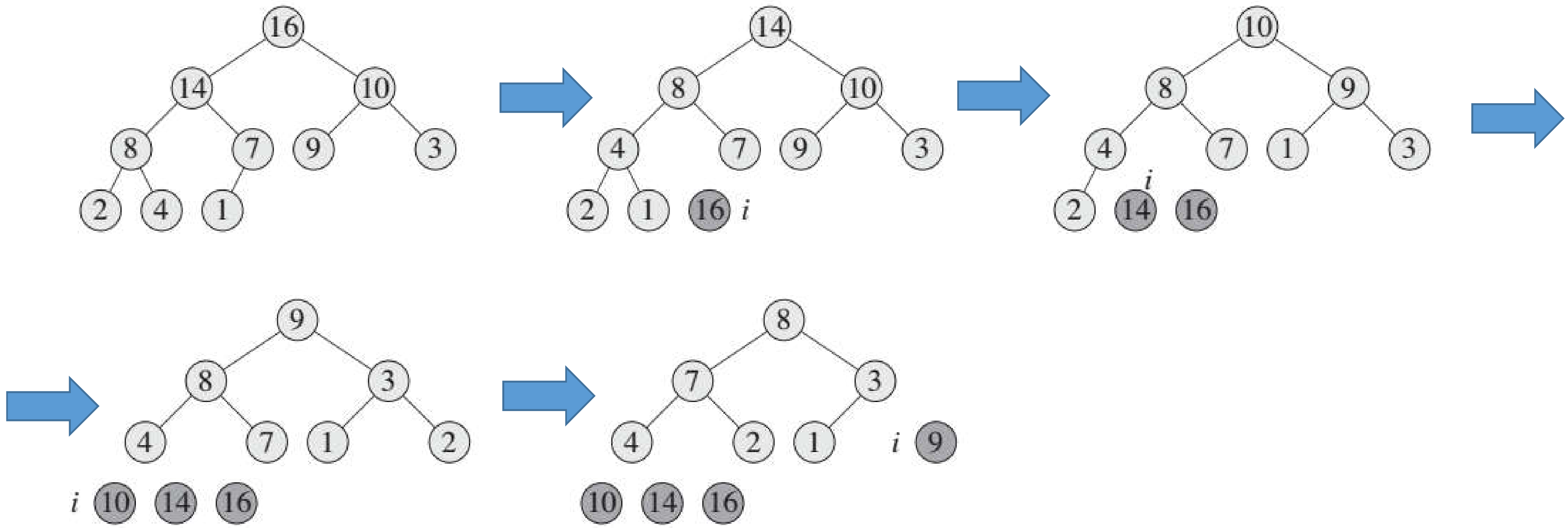
1	2	3	4	5	6	7	8	9	10
10	8	9	4	7	1	3	2	14	16

Heapsort Example



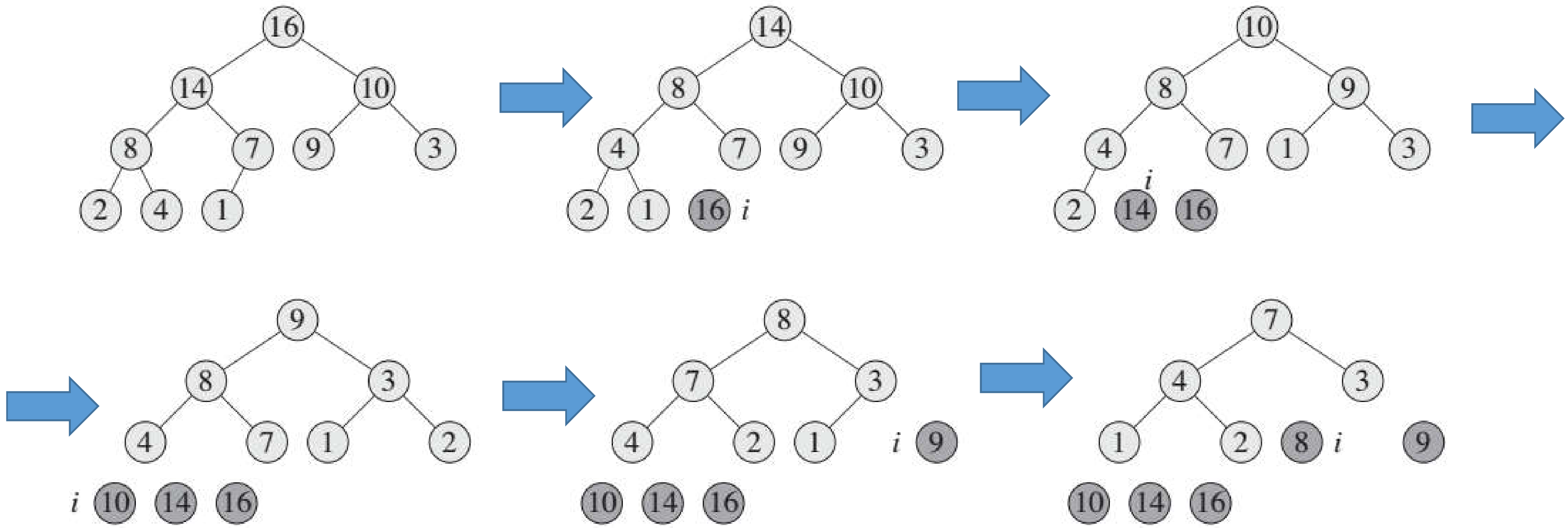
1	2	3	4	5	6	7	8	9	10
9	8	3	4	7	1	2	10	14	16

Heapsort Example



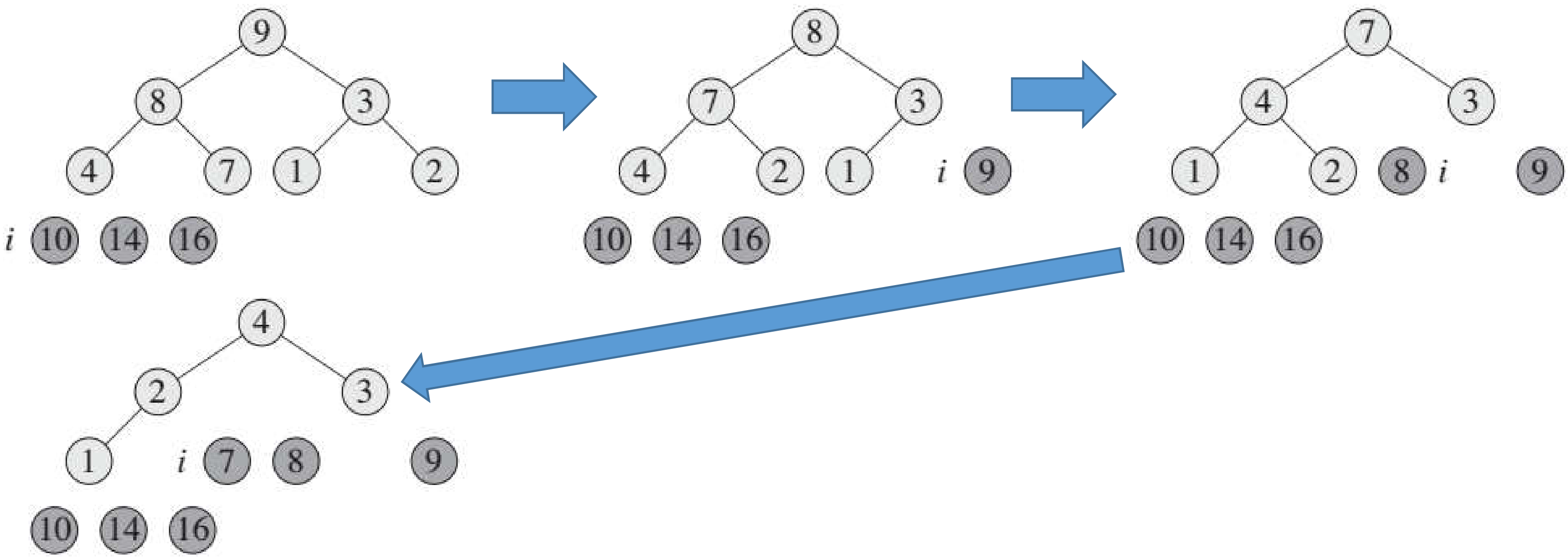
1	2	3	4	5	6	7	8	9	10
8	7	3	4	2	1	9	10	14	16

Heapsort Example



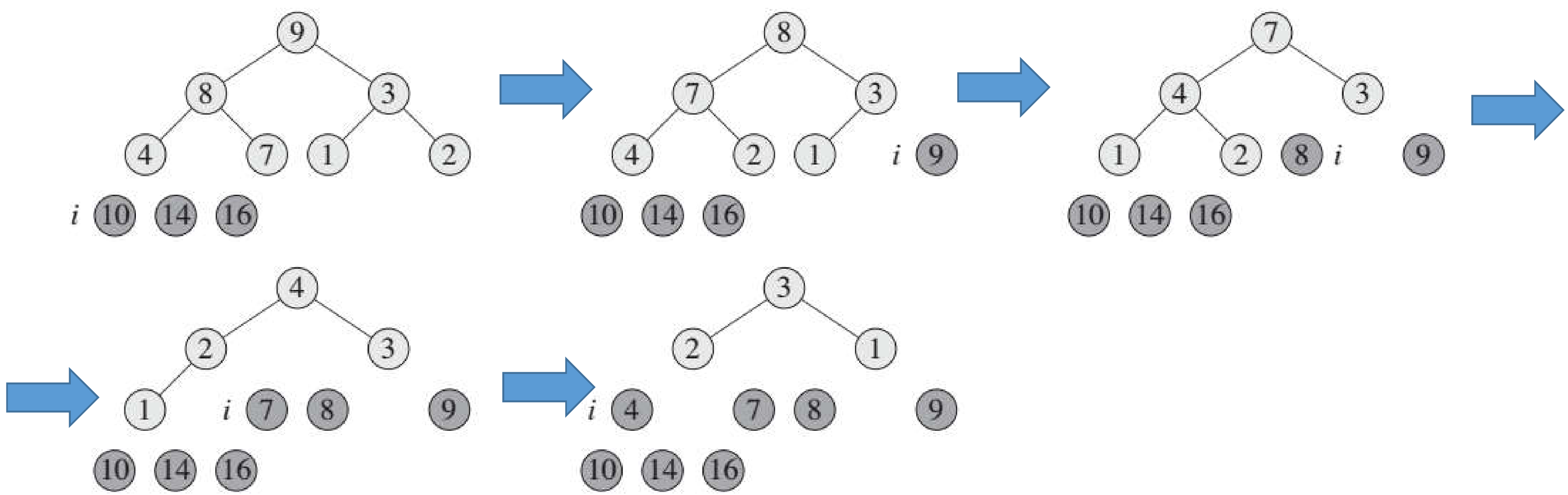
1	2	3	4	5	6	7	8	9	10
7	4	3	1	2	8	9	10	14	16

Heapsort Example



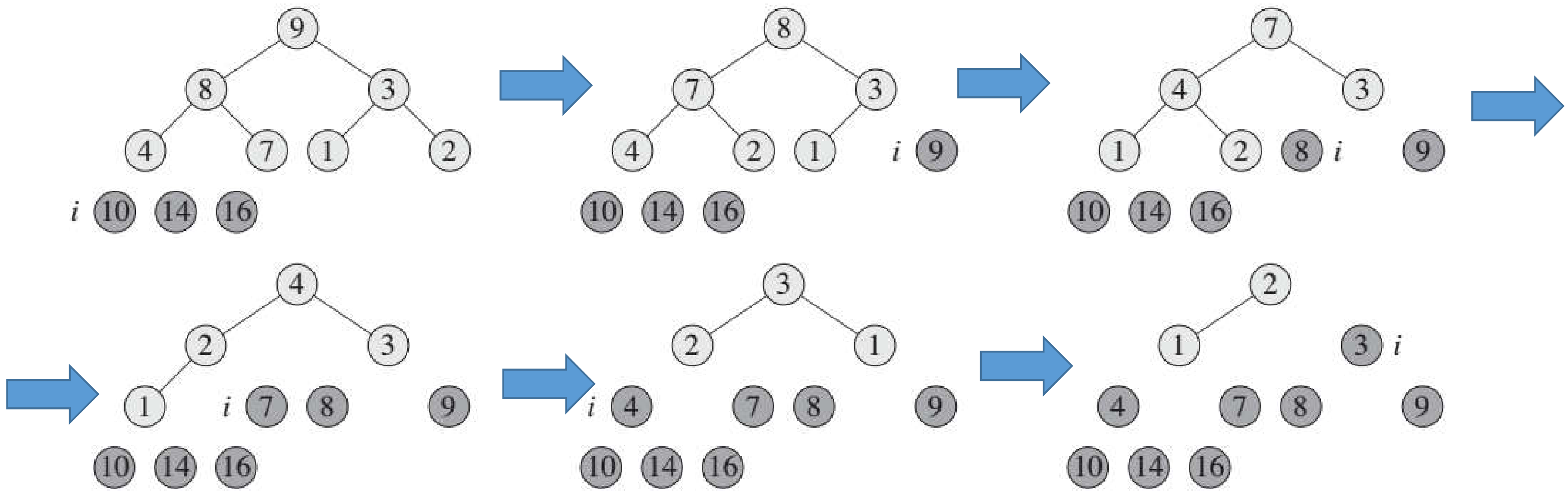
1	2	3	4	5	6	7	8	9	10
4	2	3	1	7	8	9	10	14	16

Heapsort Example



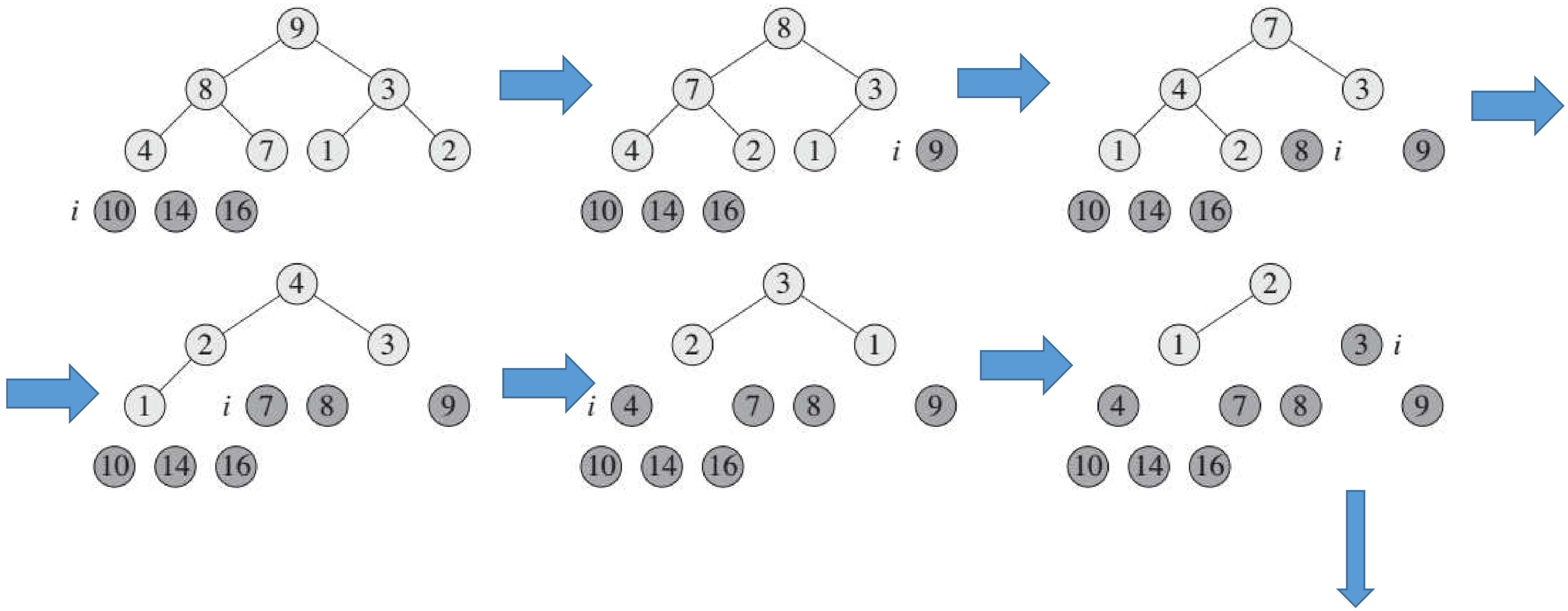
1	2	3	4	5	6	7	8	9	10
3	2	1	4	7	8	9	10	14	16

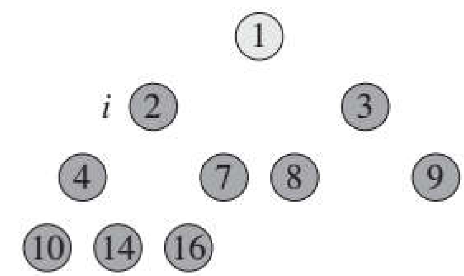
Heapsort Example



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

Heapsort Example





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

Analyzing Heapsort

```
Heapsort(A) {
   1. BuildHeap(A);
   2. for (i = length(A) downto 2) {
   3.       Swap(A[1], A[i]);
   4.       heap_size(A) =   heap_size(A) – 1;
   5.       Heapify(A, 1);
   }
}
```

- Line #2 loops for n – 1 time
- So, Heapify() at Line #5 is called (from this procedure) n – 1 times.

- The call to **BuildHeap()** takes O(n) time (Line #1)
- Each of the n-1 calls to **Heapify()** takes O(lg$n$) time

Analyzing Heapsort

```
Heapsort(A) {
   1. BuildHeap(A);
   2. for (i = length(A) downto 2) {
   3.       Swap(A[1], A[i]);
   4.       heap_size(A) =   heap_size(A) – 1;
   5.       Heapify(A, 1);
   }
}
```

- Line #2 loops for $n - 1$ time
- So, Heapify() at Line #5 is called (from this procedure) $n - 1$ times.

- The call to **BuildHeap()** takes $O(n)$ time (Line #1)
- Each of the n-1 calls to **Heapify()** takes $O(\lg n)$ time
- Thus the total time taken by **HeapSort()**
  $= O(n) + (n - 1)\, O(\lg n)$
  $= O(n) + O(n \lg n)$
  $= O(n \lg n)$

Analyzing Heapsort

```
Heapsort(A) {
   1. BuildHeap(A);
   2. for (i = length(A) downto 2) {
   3.       Swap(A[1], A[i]);
   4.       heap_size(A) =   heap_size(A) – 1;
   5.       Heapify(A, 1);
   }
}
```

- Line #2 loops for n – 1 time
- So, Heapify() at Line #5 is called (from this procedure) n – 1 times.

- The call to **BuildHeap()** takes O(n) time (Line #1)
- Each of the n-1 calls to **Heapify()** takes O(lg$n$) time
- Thus the total time taken by **HeapSort()**
  $= O(n) + (n - 1)\, O(\lg n)$
  $= O(n) + O(n \lg n)$
  $= O(n \lg n)$
- Heapsort is a nice algorithm, but in practice Quicksort (coming up) usually wins

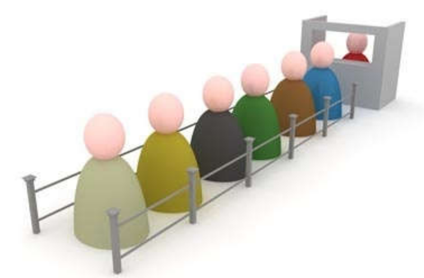
Queues

FIFO: First in, First Out

Restricted form of list: Insert at one end, remove from the other.

Notation:

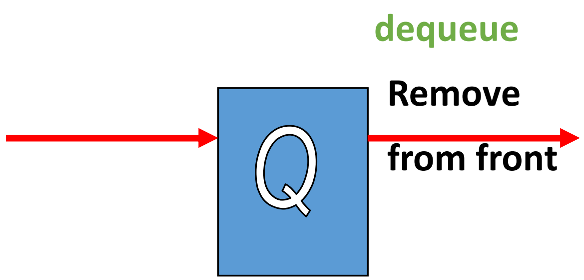
- Insert: Enqueue
- Delete: Dequeue
- First element: Front
- Last element: Rear



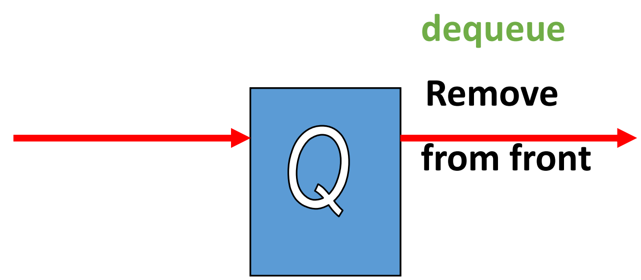
Priority Queues

A queue that is ordered according to some priority value

<u>Standard Queue</u>



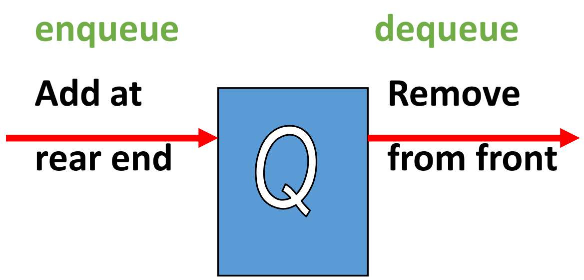
<u>Priority Queue</u>



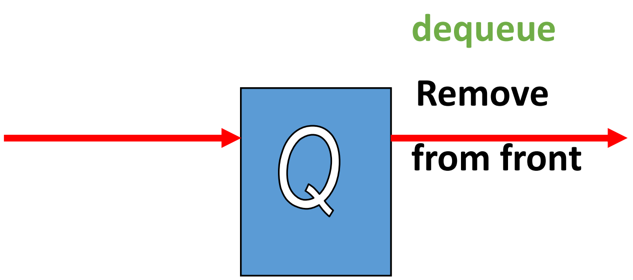
Priority Queues

A queue that is ordered according to some priority value

Standard Queue



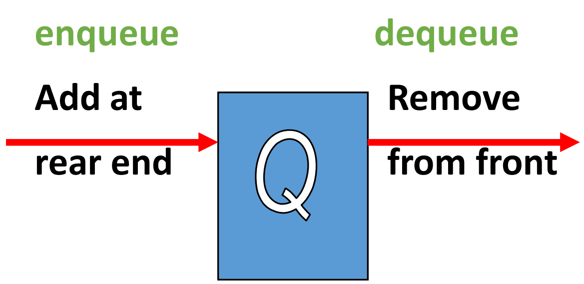
Priority Queue



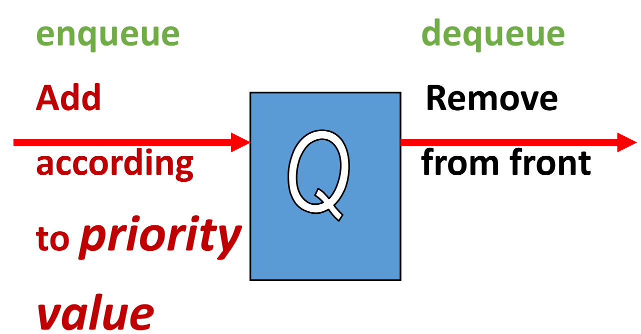
Priority Queues

A queue that is ordered according to some priority value

Standard Queue



Priority Queue



(MAX/MIN)-Priority Queues

- The heap data structure is incredibly useful for implementing *priority queues*
  - A data structure for maintaining a set *S* of elements, each with an associated value or *key*

- 2 classes
  - *max*-priority queue
  - *min*-priority queue

MAX-Priority Queues

- Applications:
  - we can use max-priority queues to schedule jobs on a shared computer.
    - The max-priority queue keeps track of the jobs to be performed and their relative priorities.
    - When a job is finished or interrupted, the scheduler selects the highest-priority job from among those pending.
    - The scheduler can add a new job to the queue at any time

MIN-Priority Queues

- Applications:
  - Simulating events
    - Events are simulated according to time of occurrence
    - The event with the next lowest time is to be generated first
    - One event can trigger multiple new events

Priority Queue Operations

Insert(*S, x*) – Inserts element *x* into set *S*, according to its priority

Maximum(*S*) – Returns, but does not remove, the element of *S* with the largest key

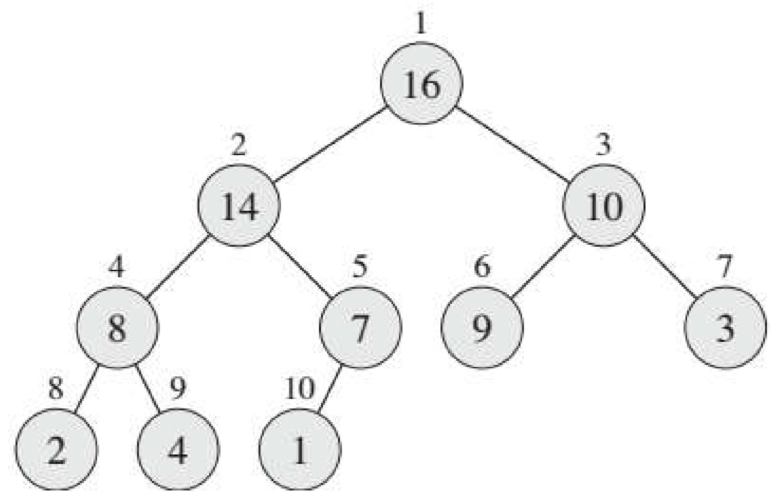
Extract-Max(*S*) – Removes and returns the element of *S* with the largest key

Increase-Key(*S, x, k*) – Increases the value of element *x*'s key to the new value *k*

How could we implement these operations using a heap?

Priority Queue Operations

HEAP-MAXIMUM(A)
1 **return** $A[1]$



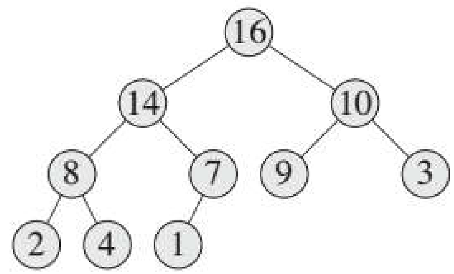
1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

Priority Queue Operations

```
HEAP-EXTRACT-MAX(A)
if A.heap-size < 1
    error "heap underflow"
max = A[1]
A[1] = A[A.heap-size]
A.heap-size = A.heap-size - 1
MAX-HEAPIFY(A, 1)
return max
```

Priority Queue Operations

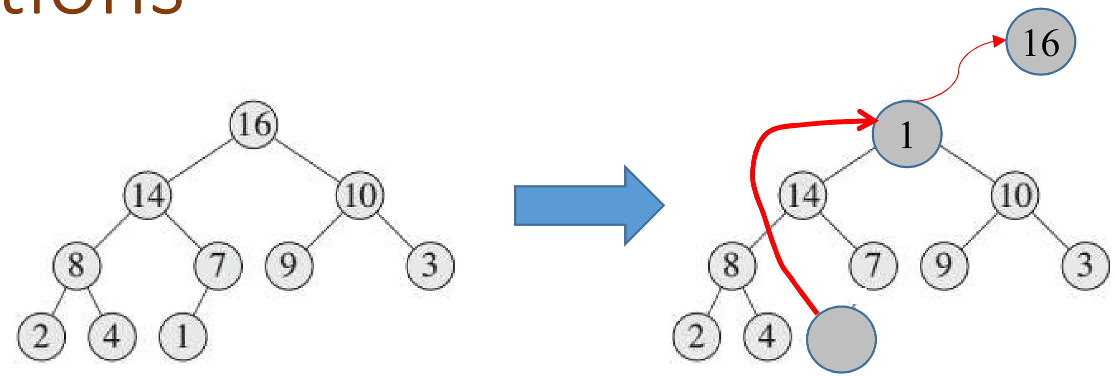
```
HEAP-EXTRACT-MAX(A)
if A.heap-size < 1
    error "heap underflow"
max = A[1]
A[1] = A[A.heap-size]
A.heap-size = A.heap-size - 1
MAX-HEAPIFY(A, 1)
return max
```



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

Priority Queue Operations

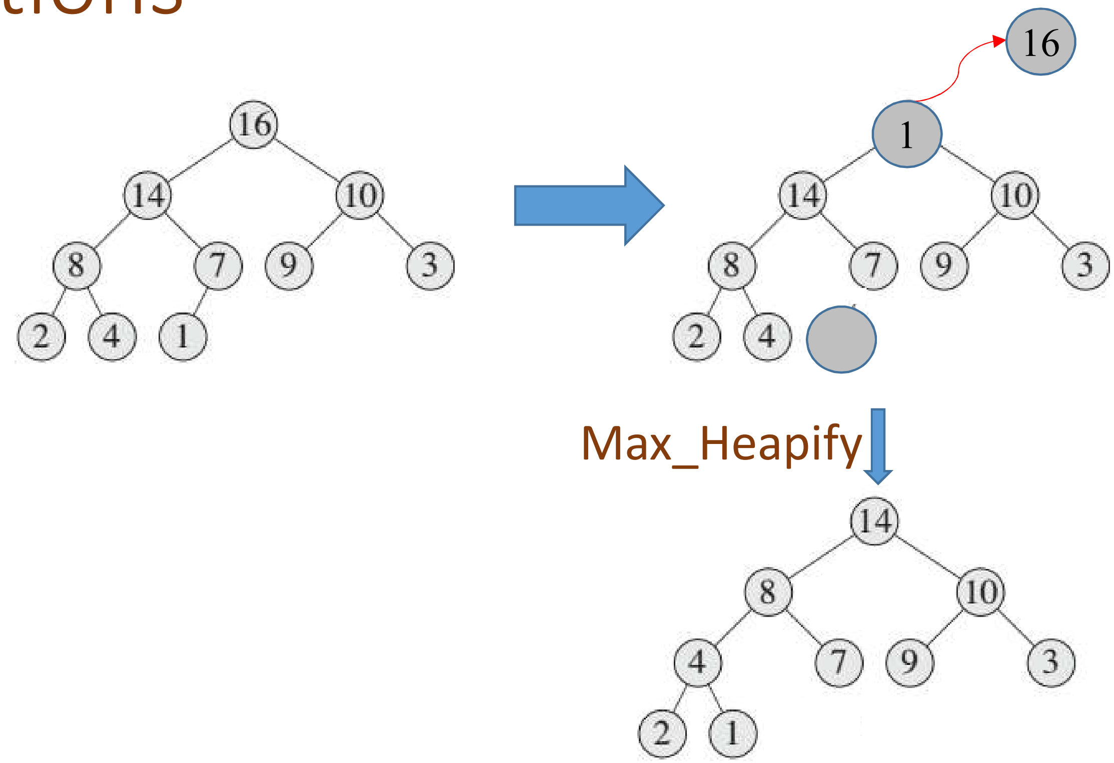
```
HEAP-EXTRACT-MAX(A)
if A.heap-size < 1
    error "heap underflow"
max = A[1]
A[1] = A[A.heap-size]
A.heap-size = A.heap-size - 1
MAX-HEAPIFY(A, 1)
return max
```



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

Priority Queue Operations

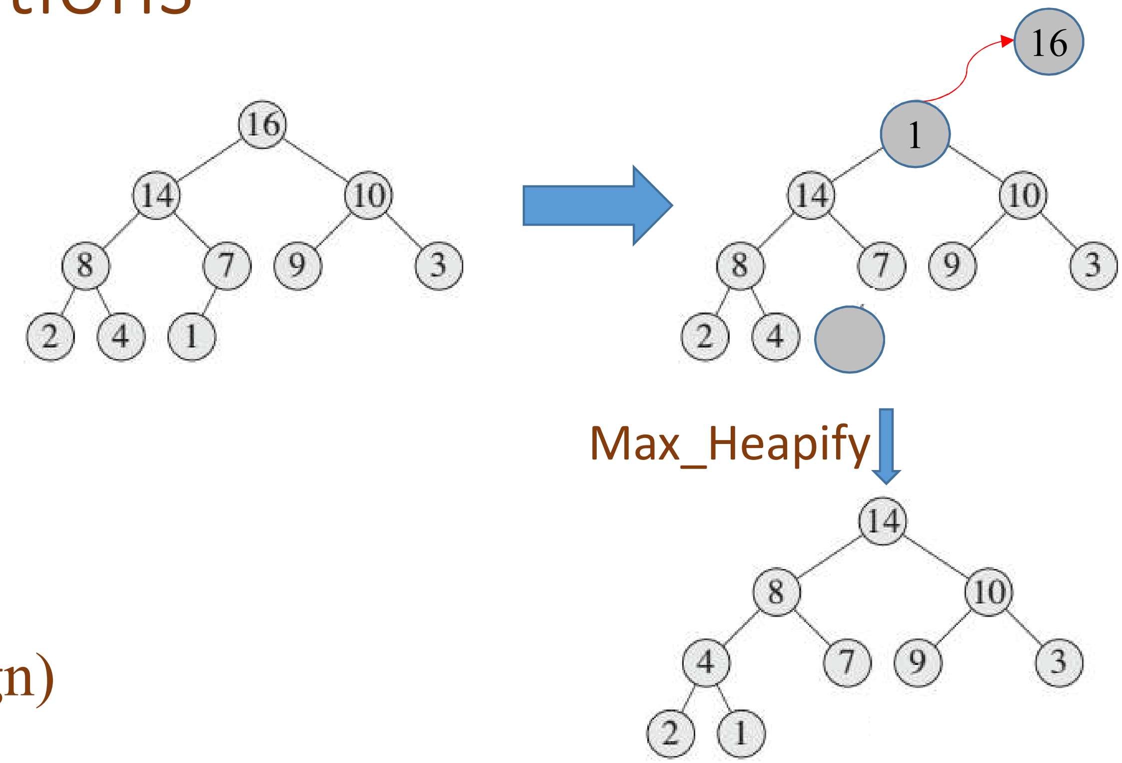
```
HEAP-EXTRACT-MAX(A)
if A.heap-size < 1
    error "heap underflow"
max = A[1]
A[1] = A[A.heap-size]
A.heap-size = A.heap-size - 1
MAX-HEAPIFY(A, 1)
return max
```



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

Priority Queue Operations

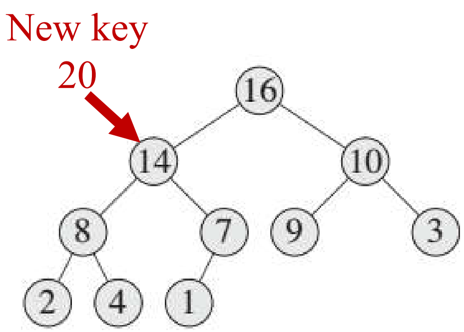
```
HEAP-EXTRACT-MAX(A)
if A.heap-size < 1
    error "heap underflow"
max = A[1]
A[1] = A[A.heap-size]
A.heap-size = A.heap-size − 1
MAX-HEAPIFY(A, 1)
return max
```



Complexity: O(logn)

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

Priority Queue Operations



Increase-Key(S, x, k) – Increases the value of element x's key to the new value k

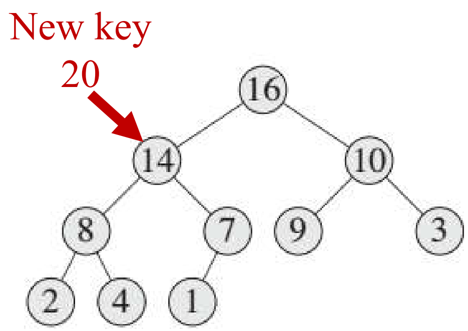
Heap relation

with parent may be violated

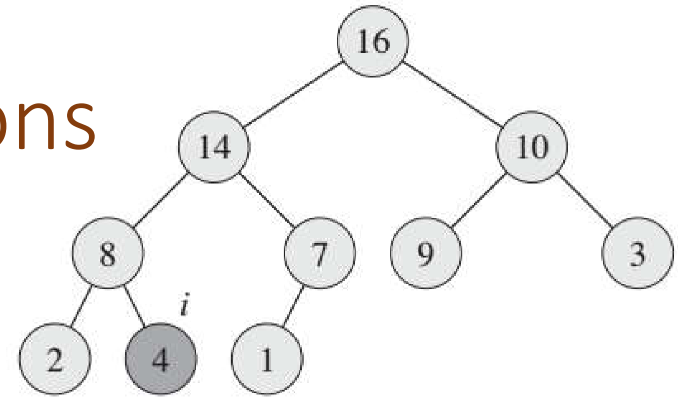
with children is maintained

Priority Queue Operations

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



Priority Queue Operations



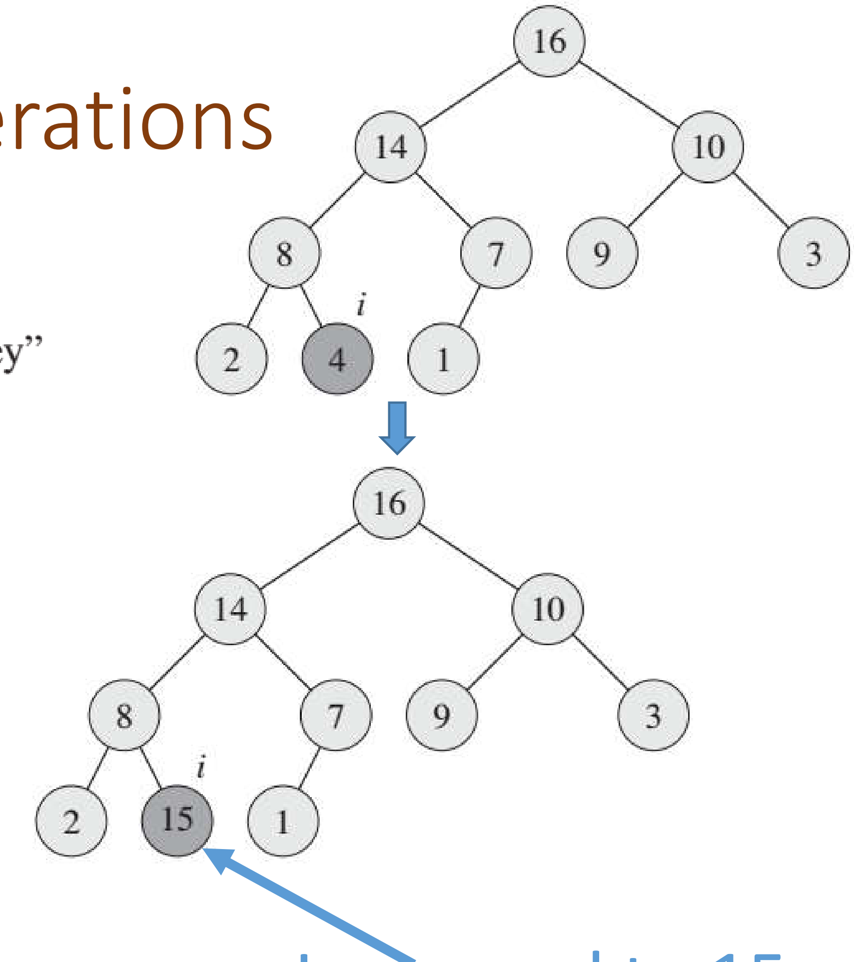
HEAP-INCREASE-KEY(A, i, key)

if $key < A[i]$
    error "new key is smaller than current key"
$A[i] = key$
while $i > 1$ and $A[\text{PARENT}(i)] < A[i]$
    exchange $A[i]$ with $A[\text{PARENT}(i)]$
    $i = \text{PARENT}(i)$

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

Priority Queue Operations

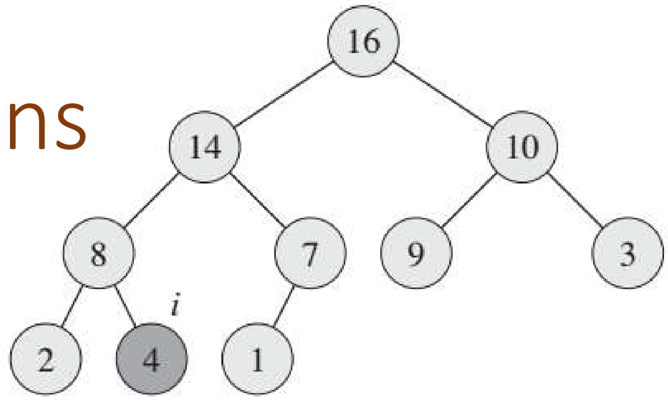
```
HEAP-INCREASE-KEY(A, i, key)
if key < A[i]
    error "new key is smaller than current key"
A[i] = key
while i > 1 and A[PARENT(i)] < A[i]
    exchange A[i] with A[PARENT(i)]
    i = PARENT(i)
```

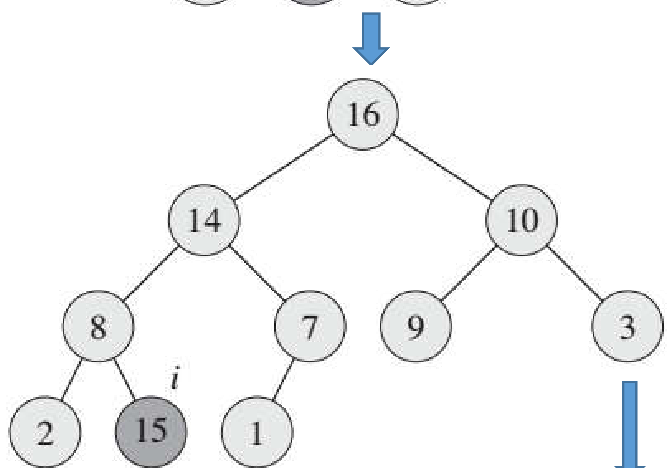


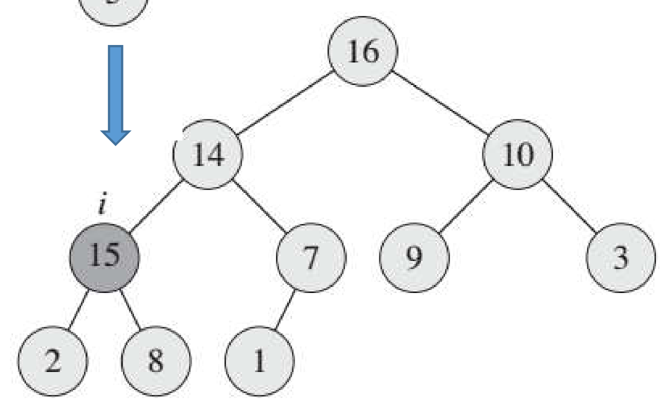
1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	15	1

Priority Queue Operations

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



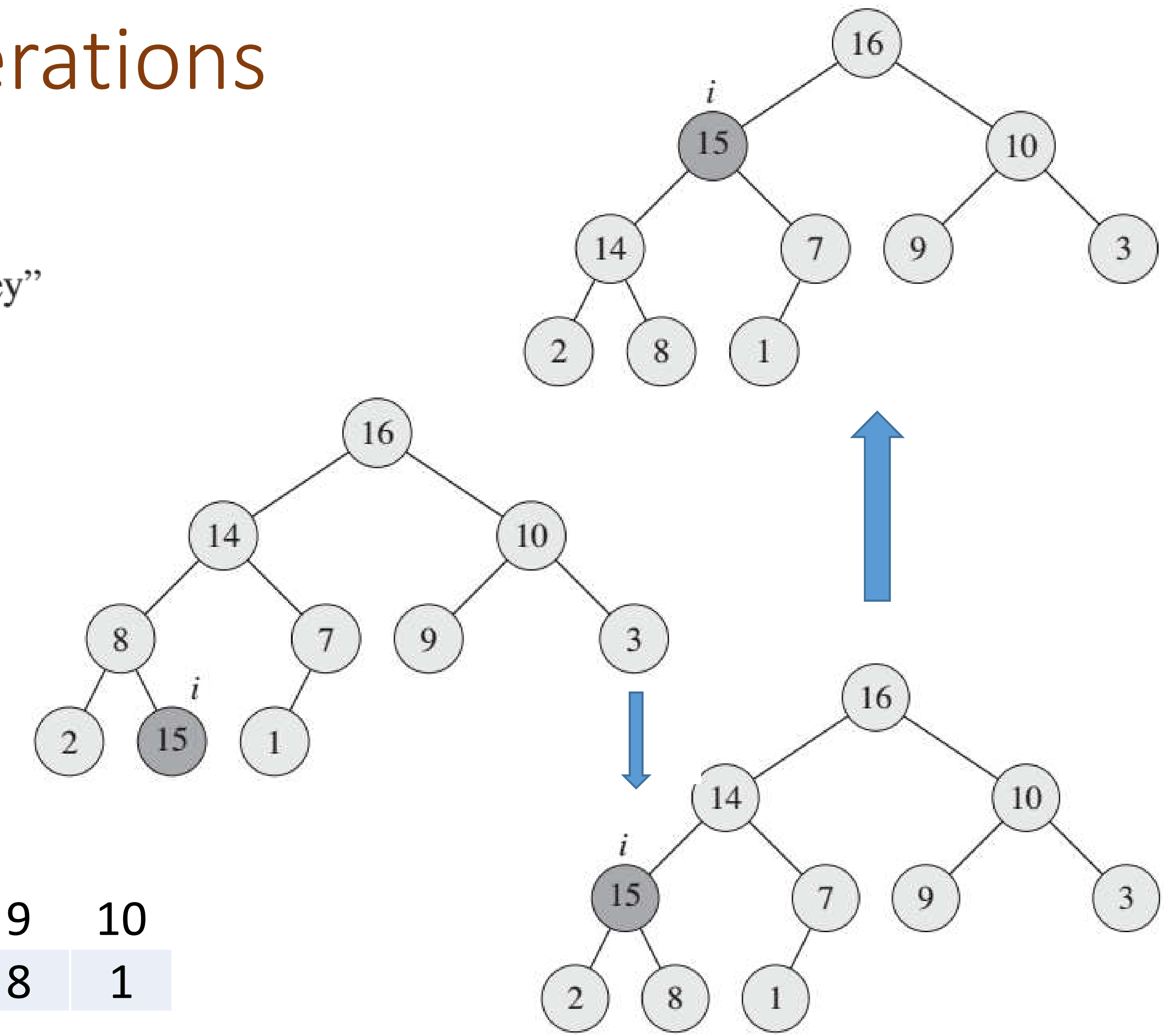




1	2	3	4	5	6	7	8	9	10
16	14	10	15	7	9	3	2	8	1

Priority Queue Operations

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

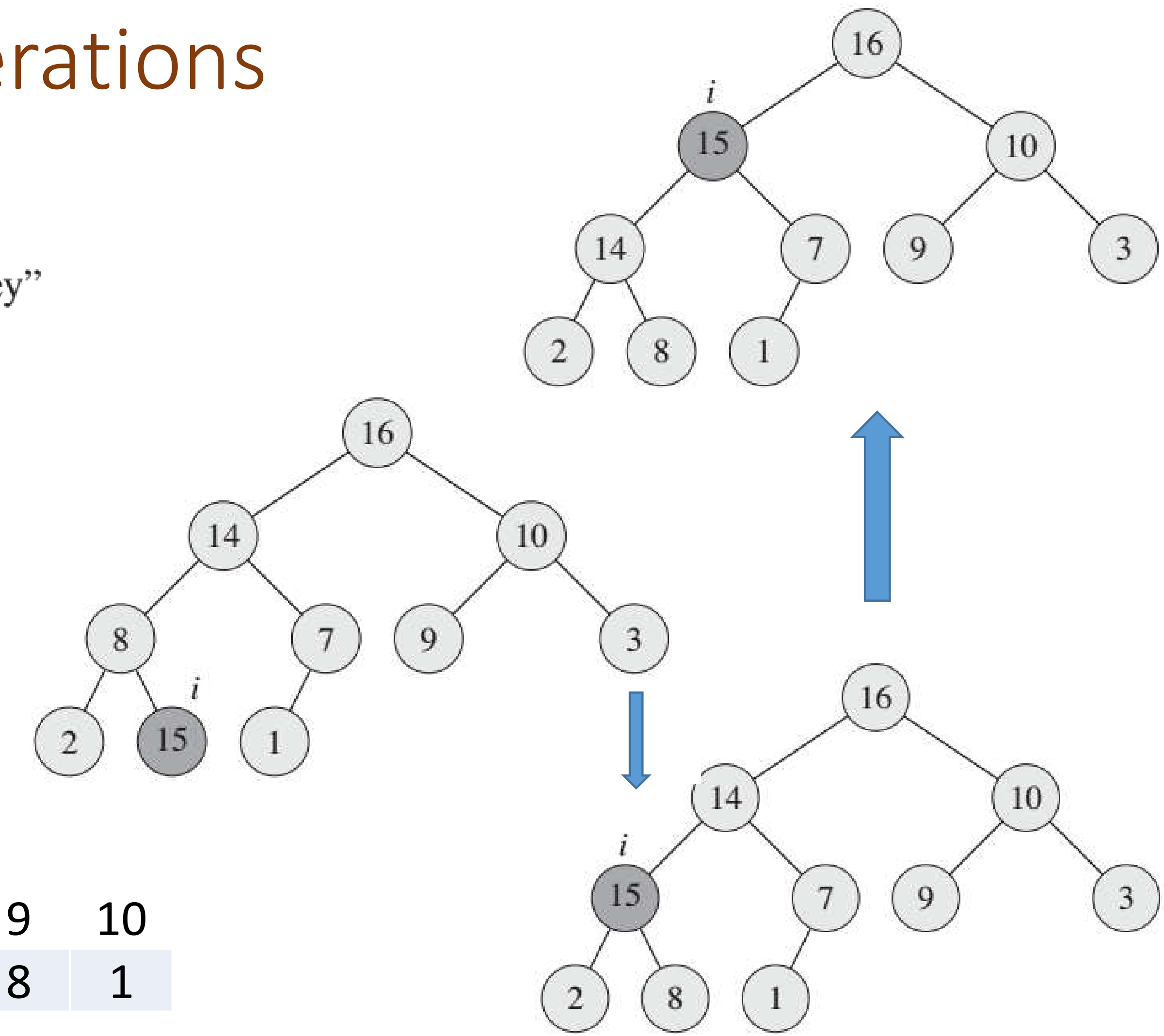


1	2	3	4	5	6	7	8	9	10
16	15	10	14	7	9	3	2	8	1

Priority Queue Operations

HEAP-INCREASE-KEY(A, i, key)

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



Complexity: O(logn)

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

Priority Queue Operations

```
MAX-HEAP-INSERT(A, key)
A.heap-size = A.heap-size + 1
A[A.heap-size] = −∞
HEAP-INCREASE-KEY(A, A.heap-size, key)
```

Insert(*S*, *x*) – Inserts element *x* into set *S*, according to its priority

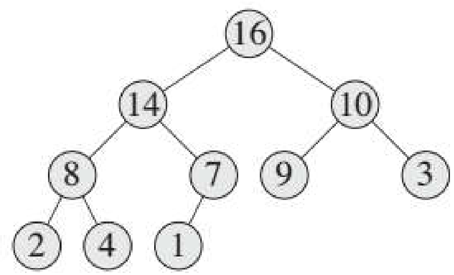
1. Create a **new leaf** with -∞

2. Then increase its value to *key*

Priority Queue Operations

MAX-HEAP-INSERT(A, key)

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

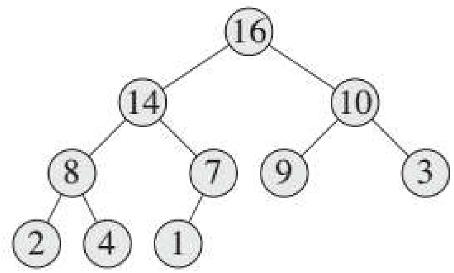


1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

Priority Queue Operations

MAX-HEAP-INSERT(A, key)

```
A.heap-size = A.heap-size + 1
A[A.heap-size] = -∞
HEAP-INCREASE-KEY(A, A.heap-size, key)
```



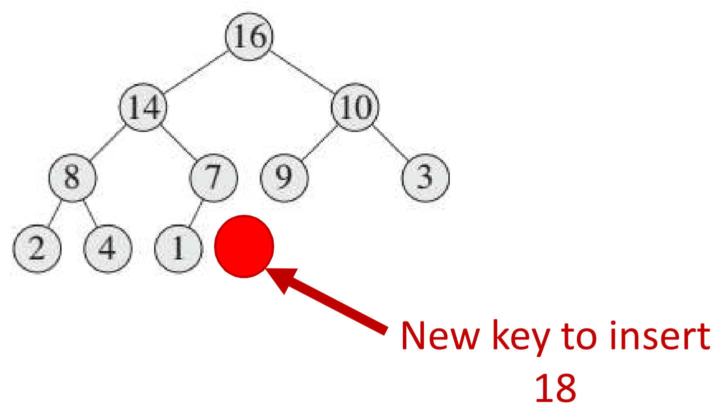
New key to insert
18

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

Priority Queue Operations

MAX-HEAP-INSERT(A, key)

```
A.heap-size = A.heap-size + 1
A[A.heap-size] = −∞
HEAP-INCREASE-KEY(A, A.heap-size, key)
```

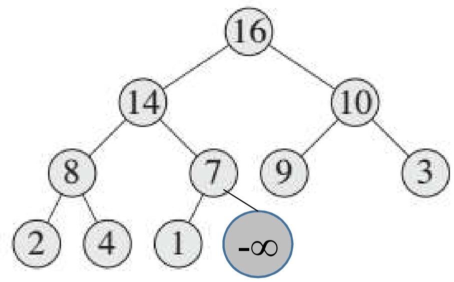


1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

Priority Queue Operations

MAX-HEAP-INSERT(A, key)

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

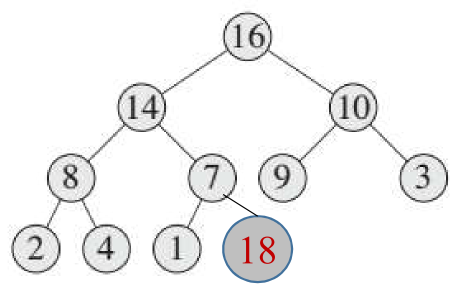


1	2	3	4	5	6	7	8	9	10	11
16	14	10	8	7	9	3	2	4	1	$-\infty$

Priority Queue Operations

MAX-HEAP-INSERT(A, key)

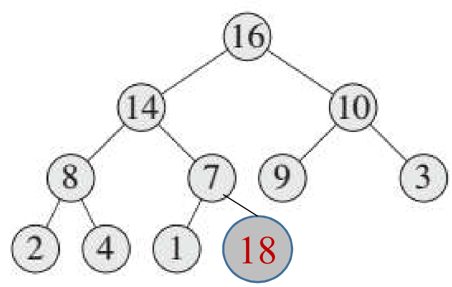
1 $A.heap\text{-}size = A.heap\text{-}size + 1$
2 $A[A.heap\text{-}size] = -\infty$
3 HEAP-INCREASE-KEY$(A, A.heap\text{-}size, key)$

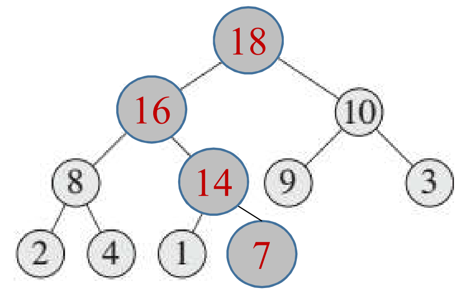


1	2	3	4	5	6	7	8	9	10	11
16	14	10	8	7	9	3	2	4	1	18

Priority Queue Operations

```
MAX-HEAP-INSERT(A, key)
    A.heap-size = A.heap-size + 1
    A[A.heap-size] = −∞
    HEAP-INCREASE-KEY(A, A.heap-size, key)
```



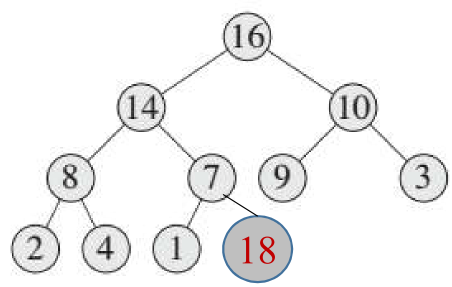


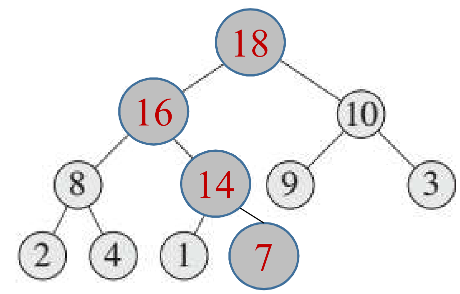
1	2	3	4	5	6	7	8	9	10	11
18	16	10	8	14	9	3	2	4	1	7

Priority Queue Operations

MAX-HEAP-INSERT(A, key)

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





Complexity: O(logn)

1	2	3	4	5	6	7	8	9	10	11
18	16	10	8	14	9	3	2	4	1	7