UNIVERSITÀ DEGLI STUDI DI PADOVA

Data Structures
Heaps (Heapsort)

Gianmaria Silvello

Department of Information Engineering
University of Padua
gianmaria.silvello@unipd.it
http://www.dei.unipd.it/~silvello/

Outline

- Heaps
- Heapsort
  - Analysis of the complexity
- Priority queues

Reference: Chapter 6 of CLRS

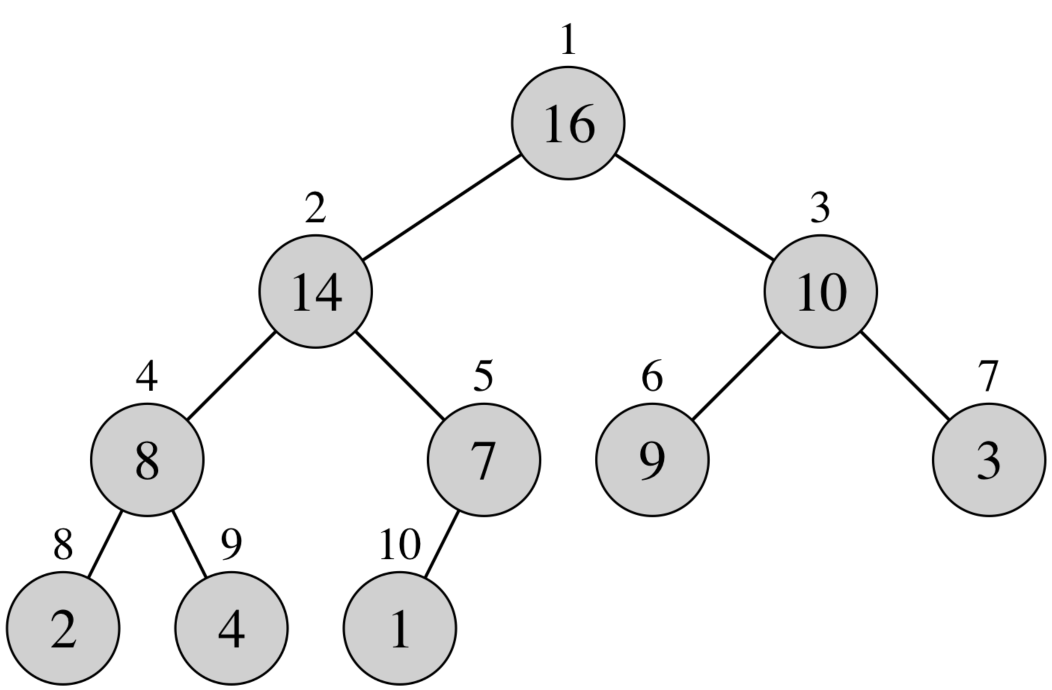
Reference: Chapter 9 of Goodrich, Tamassia and Goldwasser

Heapsort

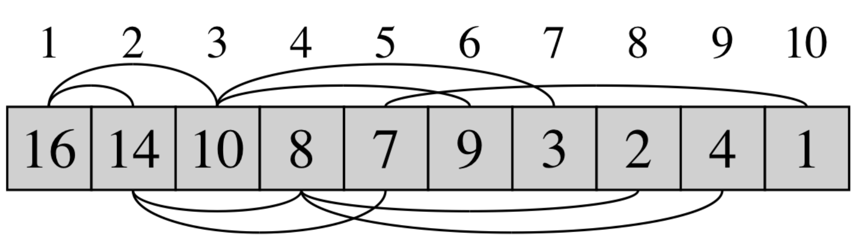
- It is a sorting algorithm that runs in O(n lg(n))
- It sorts "in-place"
  - Only a constant number of elements are stored outside the input array (in additional data structures)
- It introduces a new algorithmic design technique: the use of data structure (heap) to manage information during the execution of the algorithm
- Heaps are useful for heap sort but also to build other efficient data structures (e.g. priority queues)

Heap

- The heap data structure is an array that can be represented as a (nearly) complete binary tree
- The tree is filled up at all levels with possibly an exception at the lowest level which is filled from the left



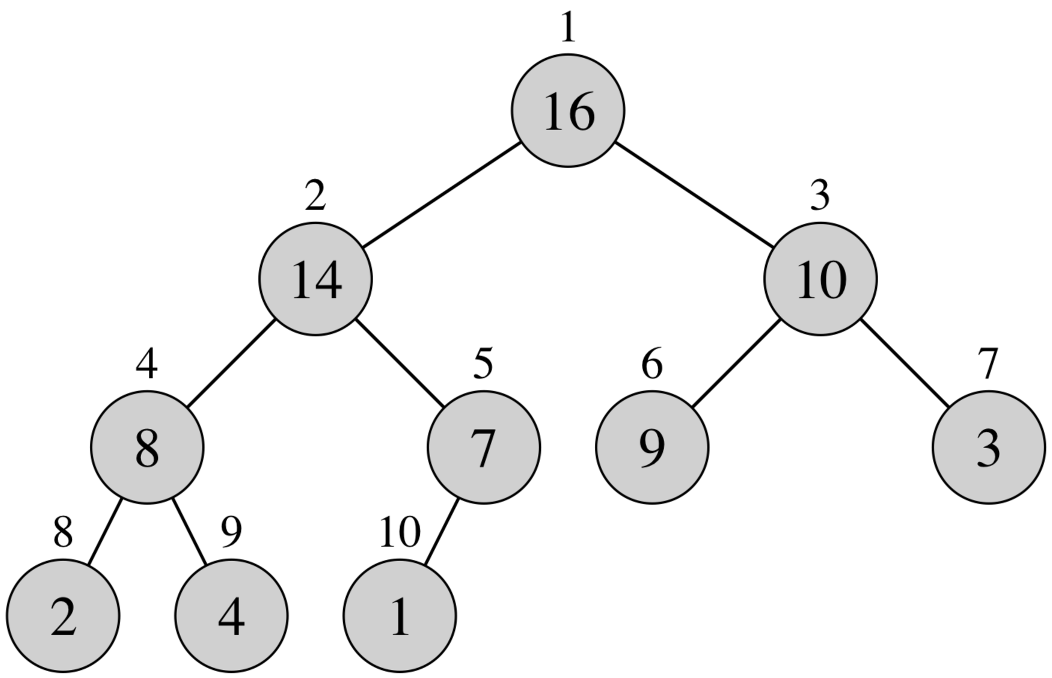
(a)

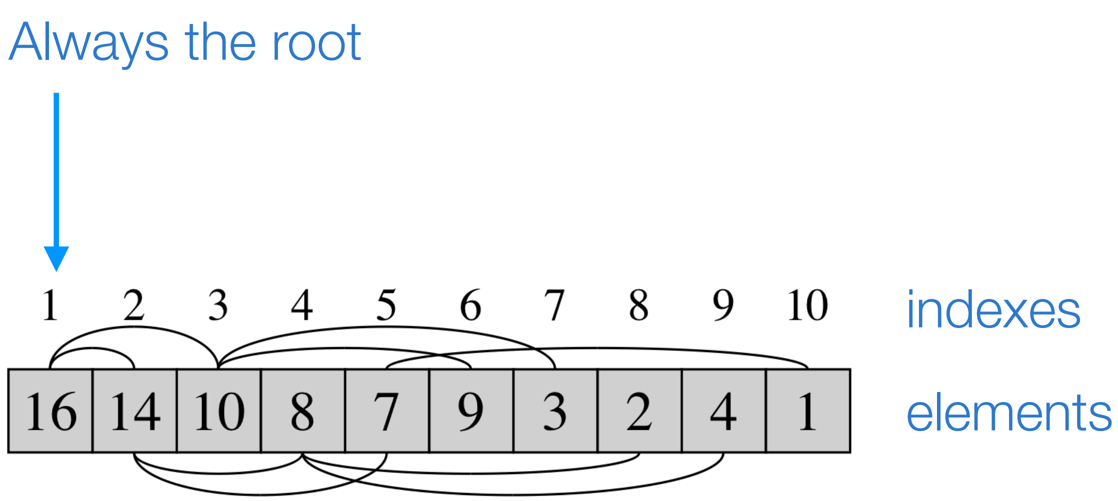


(b)

Heap

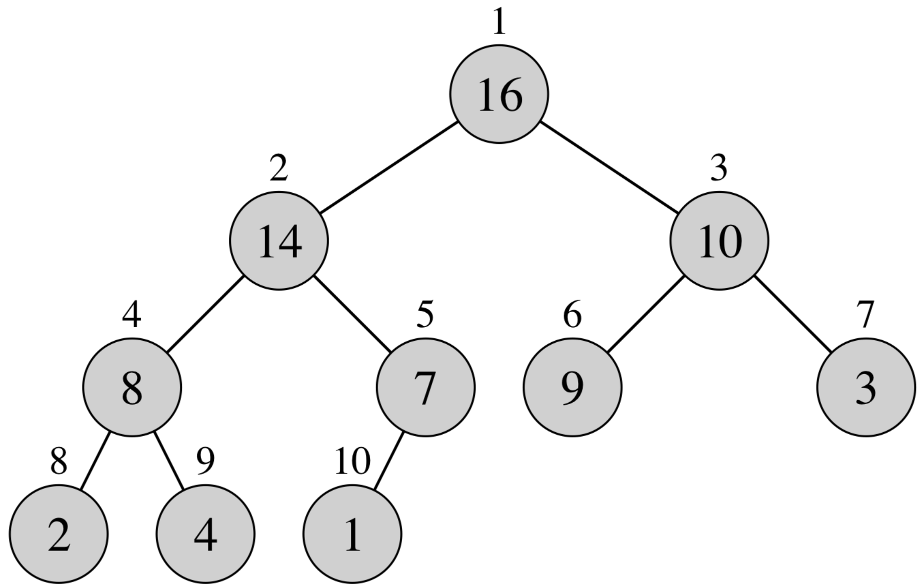
- An array `A` representing a heap is an object with two attributes: the `length[A]` of the array and the `heap-size[A]` which is the number of elements in the heap stored within the array
- `heap-size[A] <= length[A]`





Properties

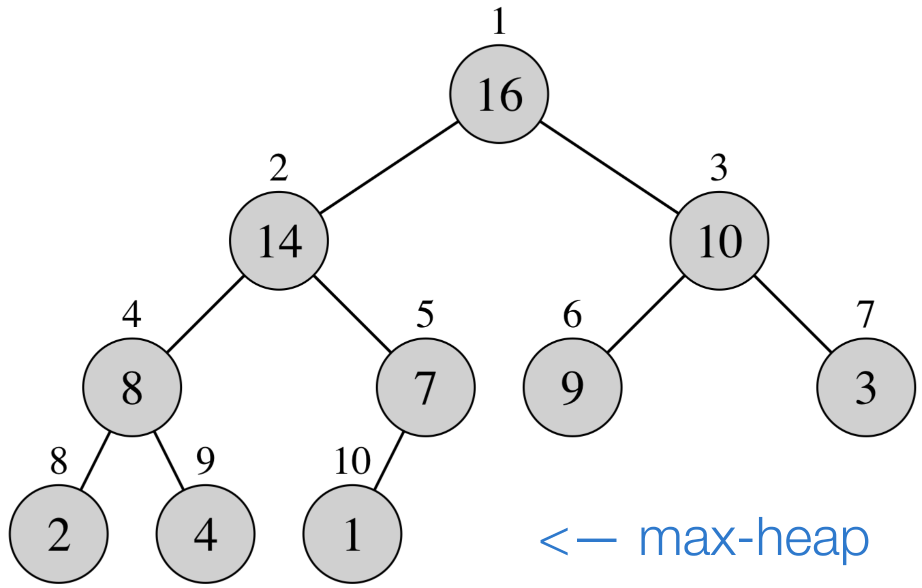
- The root of the heap A is always stored at A[1]
- Given an element with index *i*
  - The *parent* is stored at $\lfloor i/2 \rfloor$
  - The left child is stored at *2i*
  - The right child is stored at *2i+1*



- A max-heap respects the *max-heap property*
  - `A[parent(i)] ≥ A[i]`
  - The largest element is stored at the root
- A min-heap respects the *min-heap property*
  - `A[parent(i)] ≤ A[i]`

Properties

- The root of the heap A is always stored at A[1]
- Given an element with index *i*
  - The *parent* is stored at ⌊i/2⌋
  - The left child is stored at *2i*
  - The right child is stored at *2i+1*



<— max-heap

- A max-heap respects the *max-heap property*
  - `A[parent(i)] ≥ A[i]`
  - The largest element is stored at the root
- A min-heap respects the *min-heap property*
  - `A[parent(i)] ≤ A[i]`

Question

Question

Question: Is the sequence <23, 17, 14, 6, 13, 10, 1, 5, 7, 12> a max-heap?

Question

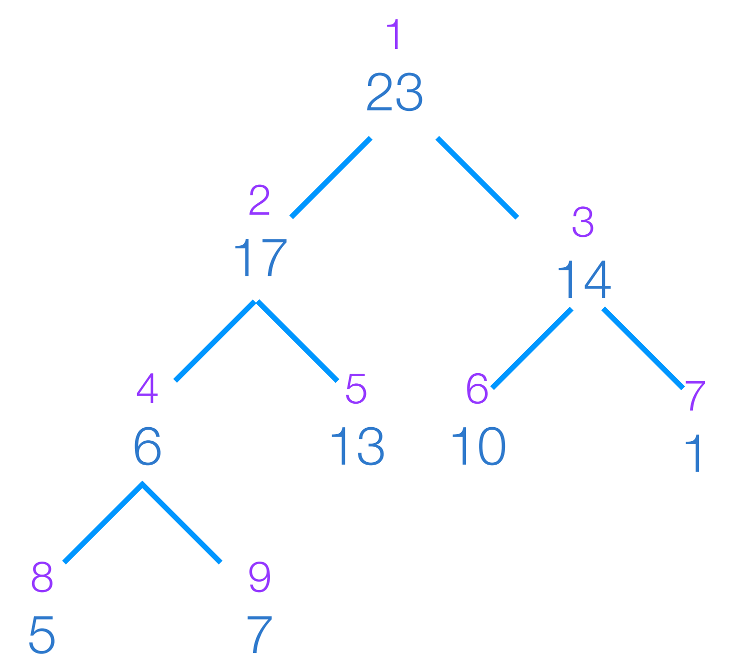
Question: Is the sequence <23, 17, 14, 6, 13, 10, 1, 5, 7, 12> a max-heap?

1 2 3 4 5 6 7 8 9 10
<23, 17, 14, 6, 13, 10, 1, 5, 7, 12>

Question

Question: Is the sequence <23, 17, 14, 6, 13, 10, 1, 5, 7, 12> a max-heap?

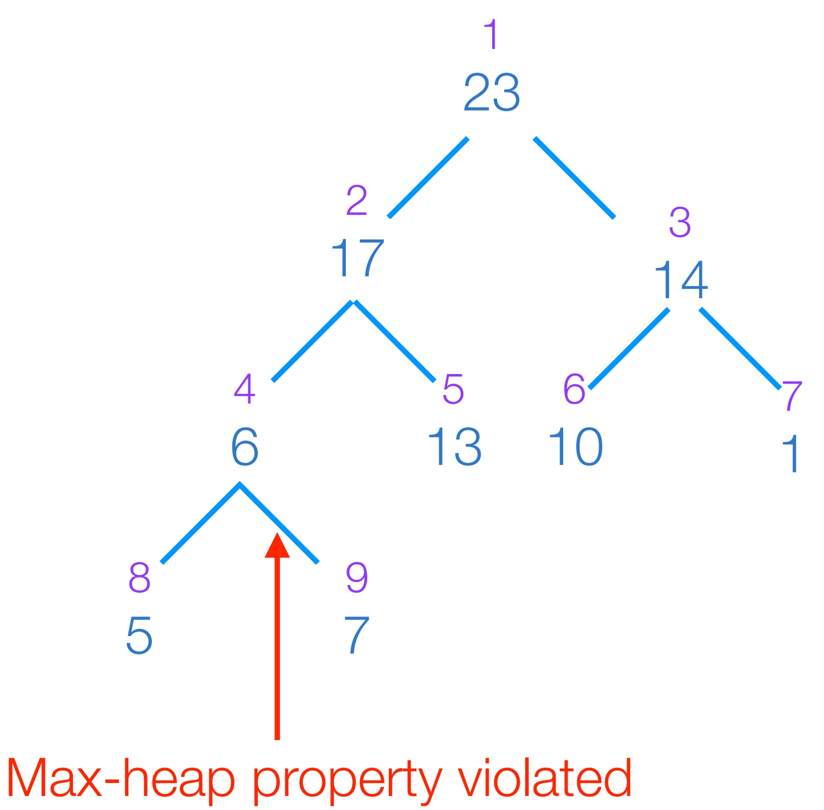
1 2 3 4 5 6 7 8 9 10
<23, 17, 14, 6, 13, 10, 1, 5, 7, 12>



Question

Question: Is the sequence <23, 17, 14, 6, 13, 10, 1, 5, 7, 12> a max-heap?

1 2 3 4 5 6 7 8 9 10
<23, 17, 14, 6, 13, 10, 1, 5, 7, 12>



Max-Heapify

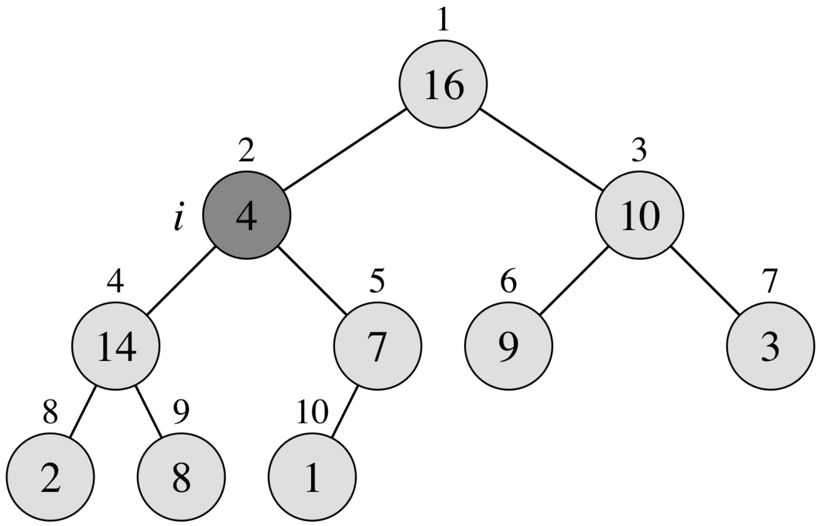
```
Input: A is an array and an index i

MAX-HEAPIFY(A,i)
l = left(i)
r = right(i)
if l≤heap-size[A] and A[l]>A[i] then
    largest = l
else largest = i
if r≤heap-size[A] and A[r]>A[largest] then
    largest = r
if largest!=i then
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A,largest)
```

Max-Heapify checks if the element at index i respect the max-heap property, if not it “floats down” the element in A[i] until the property is respected

Max-Heapify

```
Input: A is an array and an index i

MAX-HEAPIFY(A,i)
l = left(i)
r = right(i)
if l≤heap-size[A] and A[l]>A[i] then
    largest = l
else largest = i
if r≤heap-size[A] and A[r]>A[largest] then
    largest = r
if largest!=i then
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A,largest)
```

Max-Heapify checks if the element at index i respect the max-heap property, if not it "floats down" the element in A[i] until the property is respected

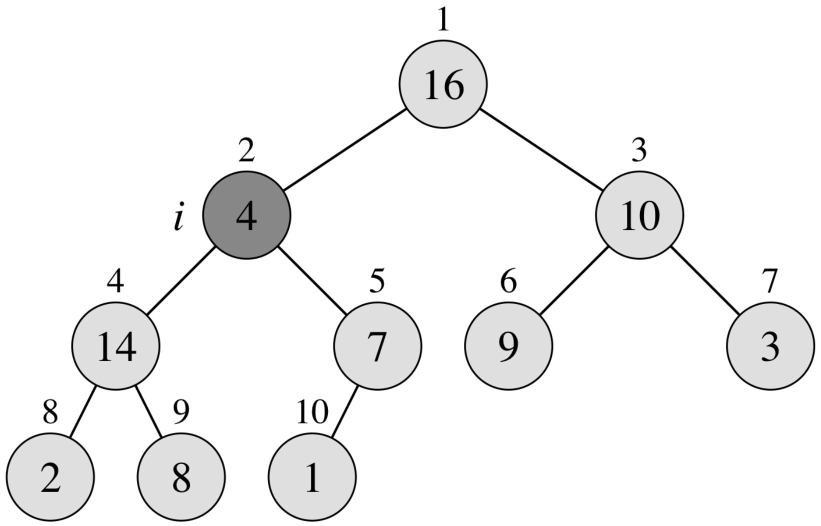


MAX-HEAPIFY(A,2)

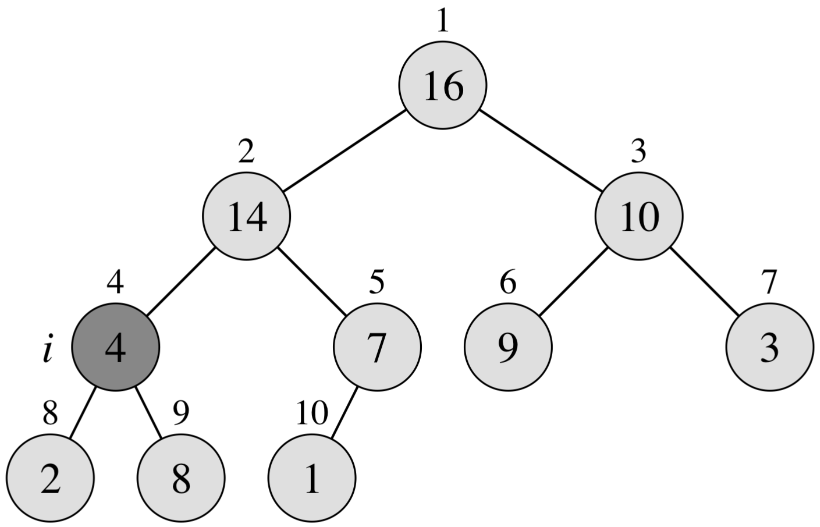
Max-Heapify

```
Input: A is an array and an index i

MAX-HEAPIFY(A,i)
l = left(i)
r = right(i)
if l≤heap-size[A] and A[l]>A[i] then
    largest = l
else largest = i
if r≤heap-size[A] and A[r]>A[largest] then
    largest = r
if largest!=i then
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A,largest)
```

Max-Heapify checks if the element at index i respect the max-heap property, if not it “floats down” the element in A[i] until the property is respected



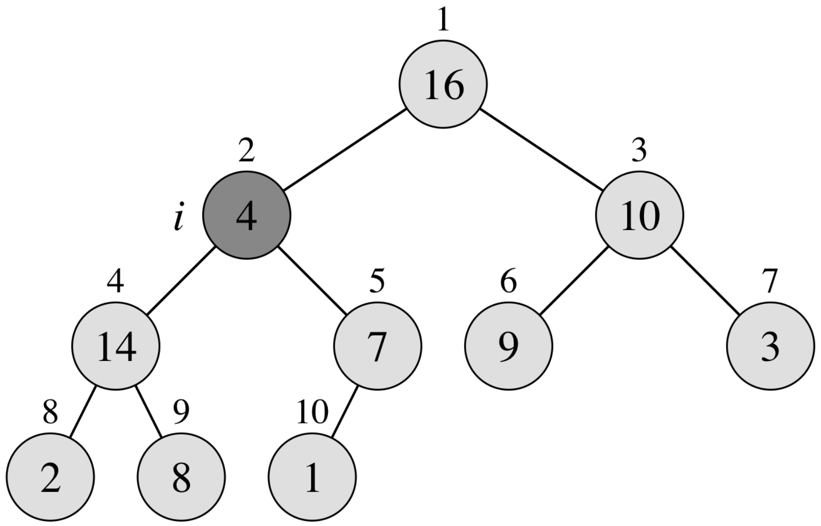
MAX-HEAPIFY(A,2)



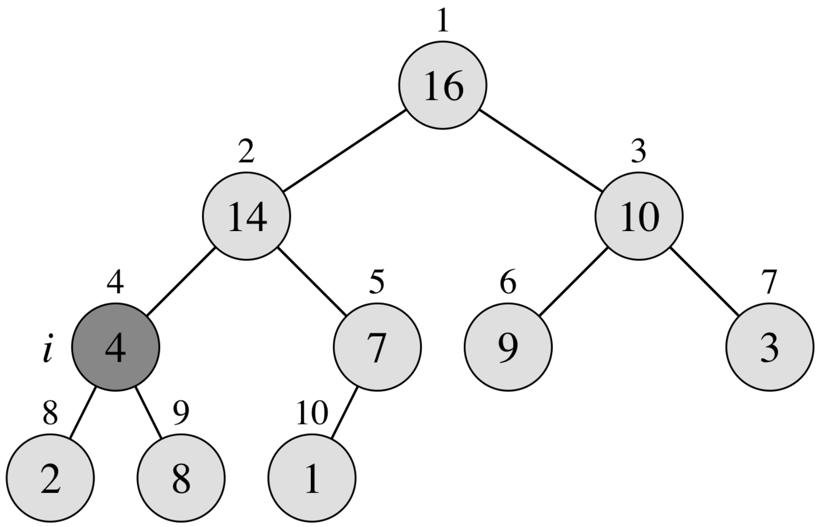
Max-Heapify

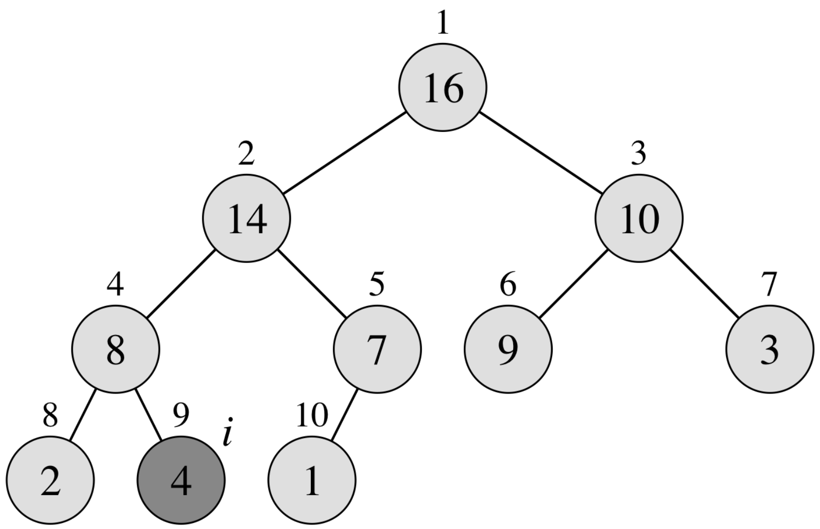
```
Input: A is an array and an index i

MAX-HEAPIFY(A,i)
l = left(i)
r = right(i)
if l≤heap-size[A] and A[l]>A[i] then
    largest = l
else largest = i
if r≤heap-size[A] and A[r]>A[largest] then
    largest = r
if largest!=i then
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A,largest)
```

Max-Heapify checks if the element at index i respect the max-heap property, if not it “floats down” the element in A[i] until the property is respected



MAX-HEAPIFY(A,2)



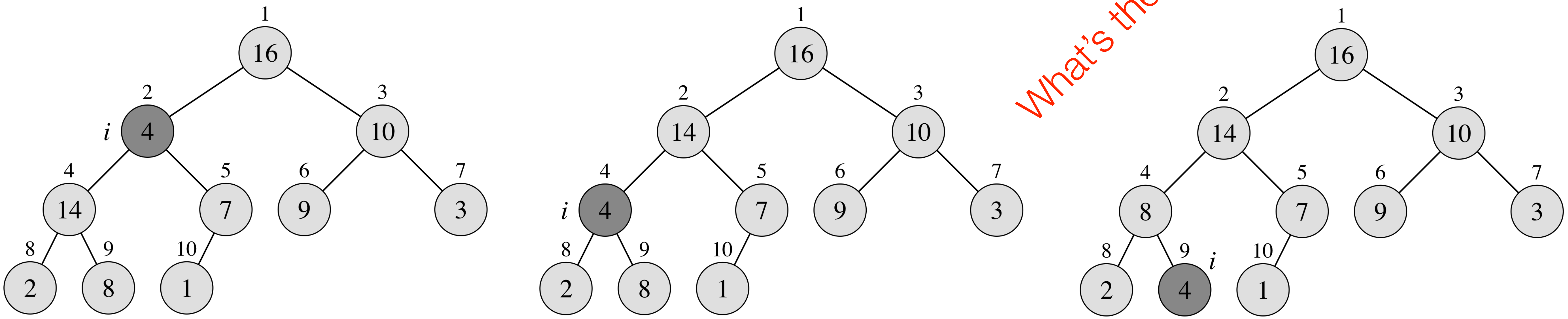


Max-Heapify

```
Input: A is an array and an index i

MAX-HEAPIFY(A,i)
l = left(i)
r = right(i)
if l≤heap-size[A] and A[l]>A[i] then
    largest = l
else largest = i
if r≤heap-size[A] and A[r]>A[largest] then
    largest = r
if largest!=i then
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A,largest)
```

Max-Heapify checks if the element at index i respect the max-heap property, if not it "floats down" the element in A[i] until the property is respected

What's the running time of Max-Heapify?



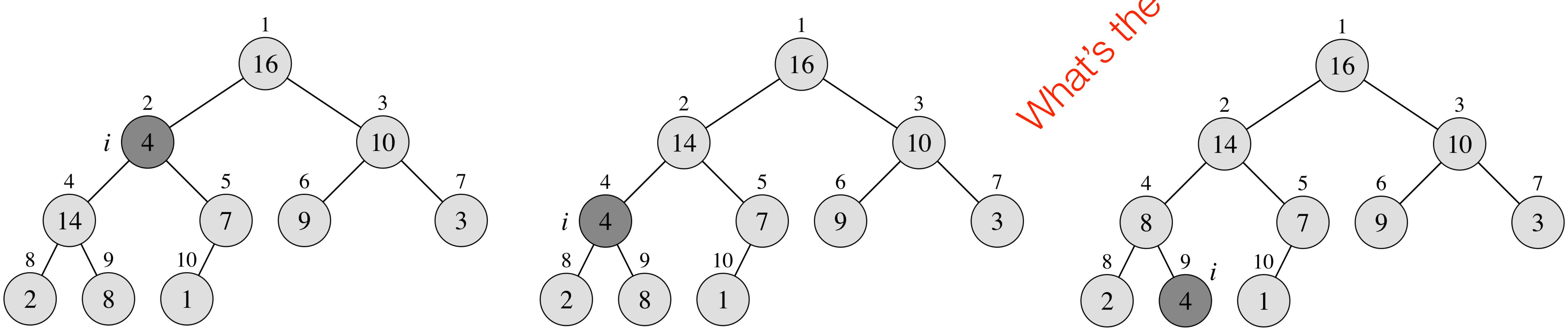
Max-Heapify

```
Input: A is an array and an index i

MAX-HEAPIFY(A,i)
l = left(i)
r = right(i)
if l≤heap-size[A] and A[l]>A[i] then
    largest = l
else largest = i
if r≤heap-size[A] and A[r]>A[largest] then
    largest = r
if largest!=i then
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A,largest)
```

Max-Heapify checks if the element at index i respect the max-heap property, if not it “floats down” the element in A[i] until the property is respected

What’s the running time of Max-Heapify?

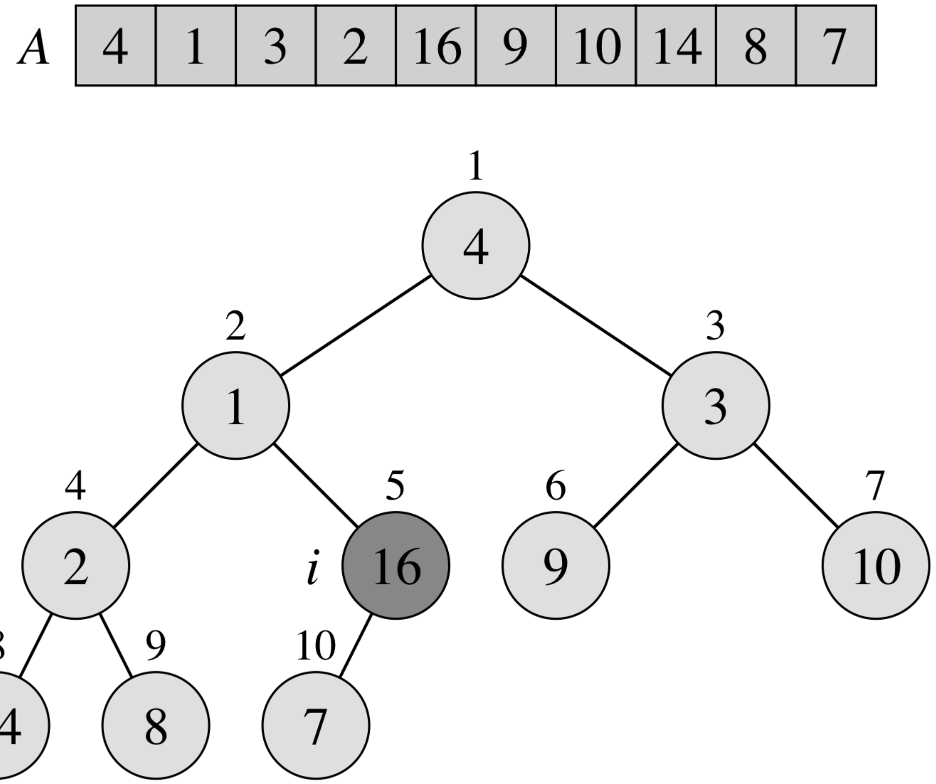
$\Theta(\lg n)$



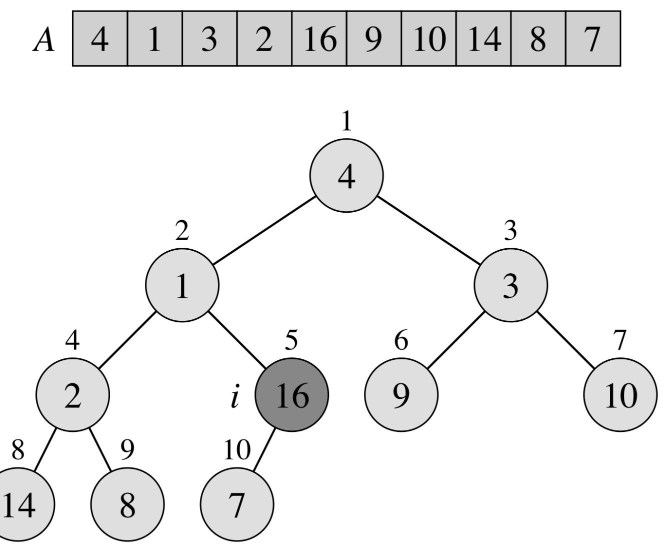
MAX-HEAPIFY(A,2)

Building a heap

- With an array representation for storing an n-element heap, the leaves are the nodes indexed by $\lfloor n/2 \rfloor+1, \lfloor n/2 \rfloor+2, \ldots, n$



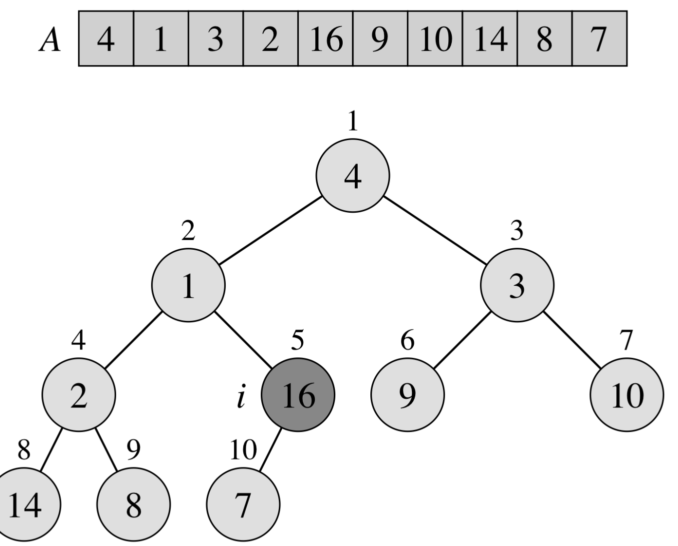
Building a (max) heap




```
Input: A is an array

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

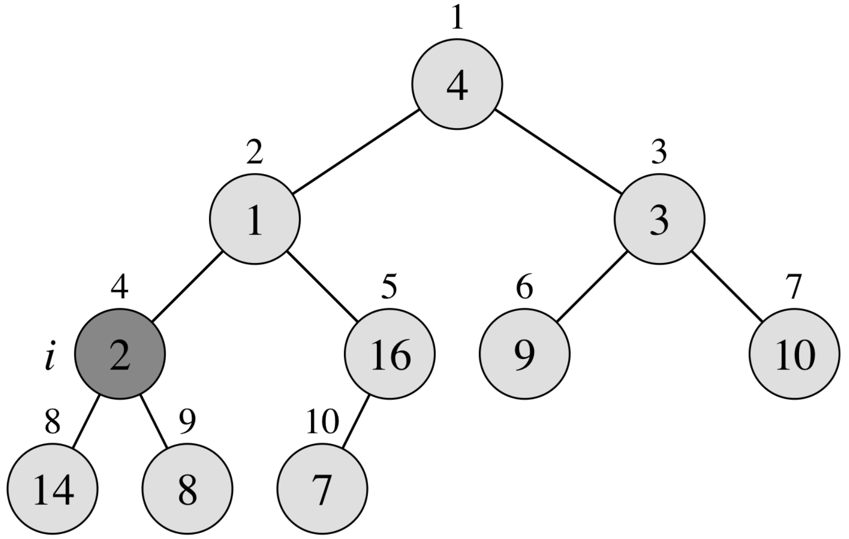
- With an array representation for storing an n-element heap, the leaves are the nodes indexed by $\lfloor n/2 \rfloor+1, \lfloor n/2 \rfloor+2, \ldots, n$

Building a (max) heap

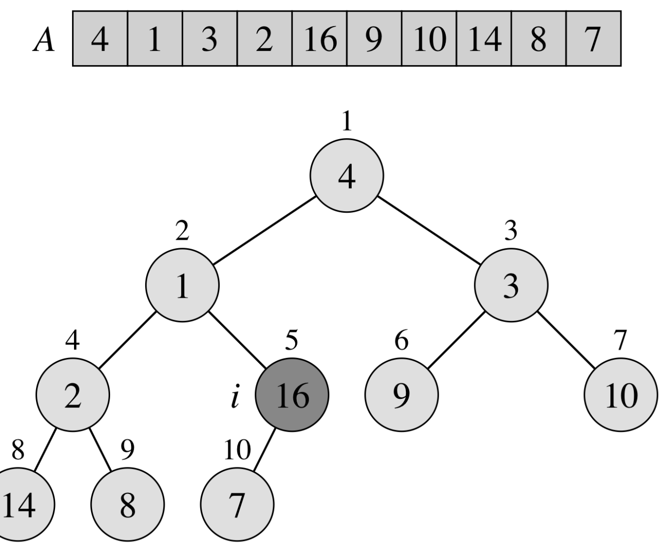



```
Input: A is an array

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

- With an array representation for storing an n-element heap, the leaves are the nodes indexed by ⌊n/2⌋+1, ⌊n/2⌋+2, …, n

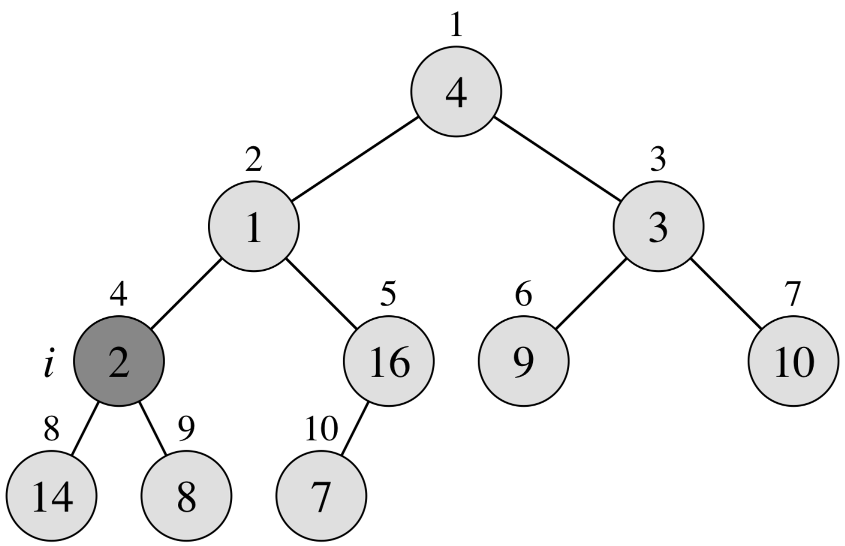


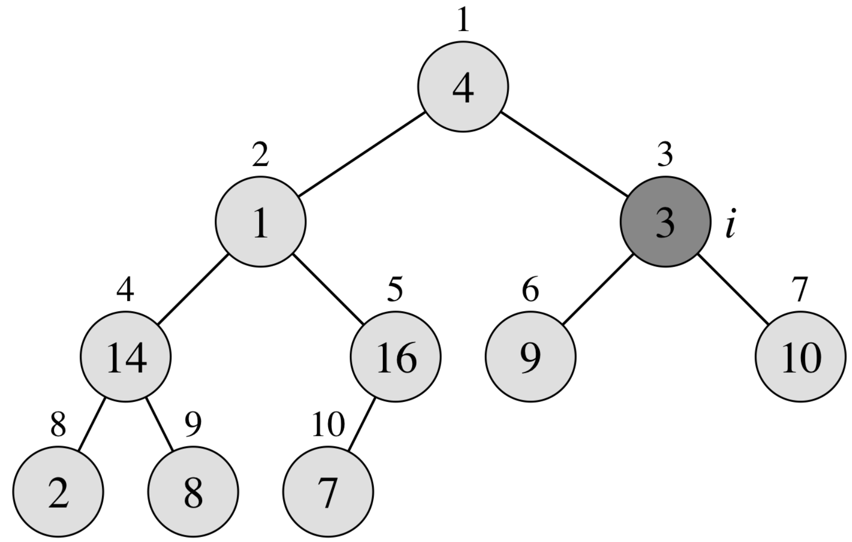
Building a (max) heap




```
Input: A is an array

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

- With an array representation for storing an n-element heap, the leaves are the nodes indexed by
⌊n/2⌋+1, ⌊n/2⌋+2, …, n



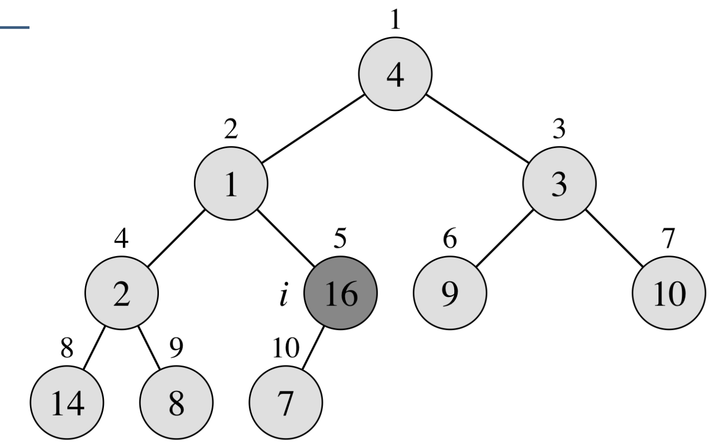


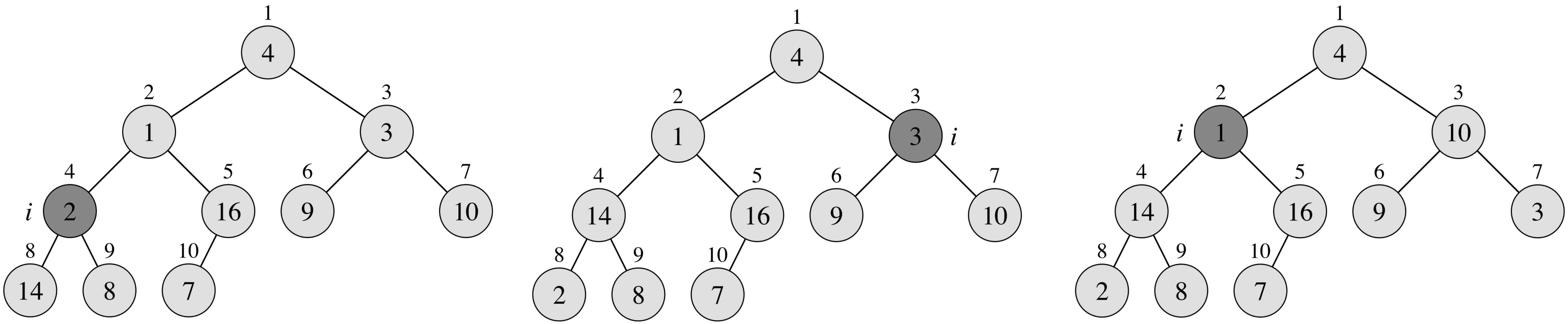
Building a (max) heap

```
Input: A is an array

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

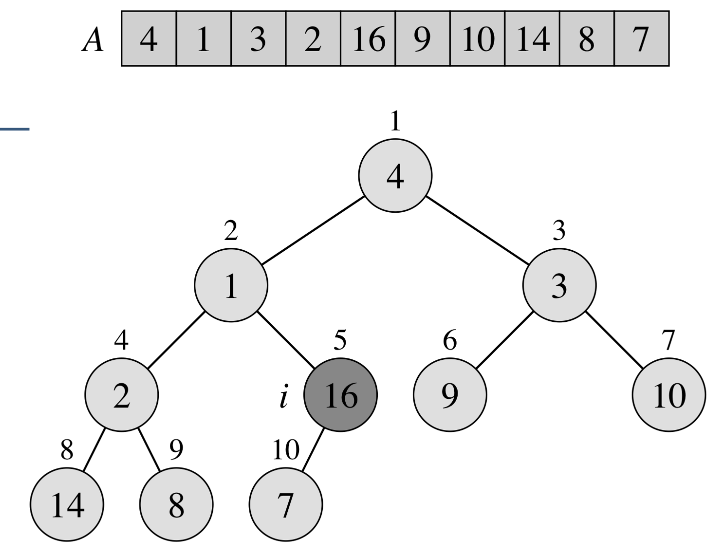
- With an array representation for storing an n-element heap, the leaves are the nodes indexed by $\lfloor n/2 \rfloor+1, \lfloor n/2 \rfloor+2, \ldots, n$

A										
	4	1	3	2	16	9	10	14	8	7



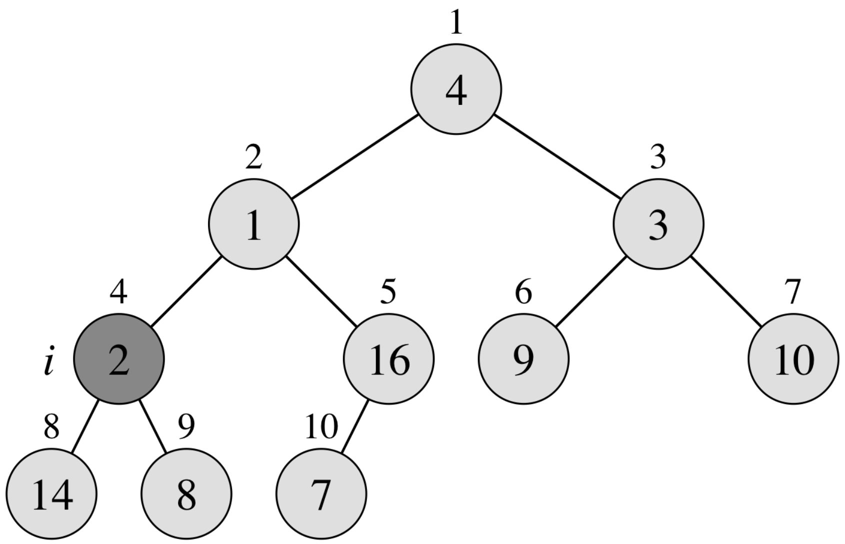


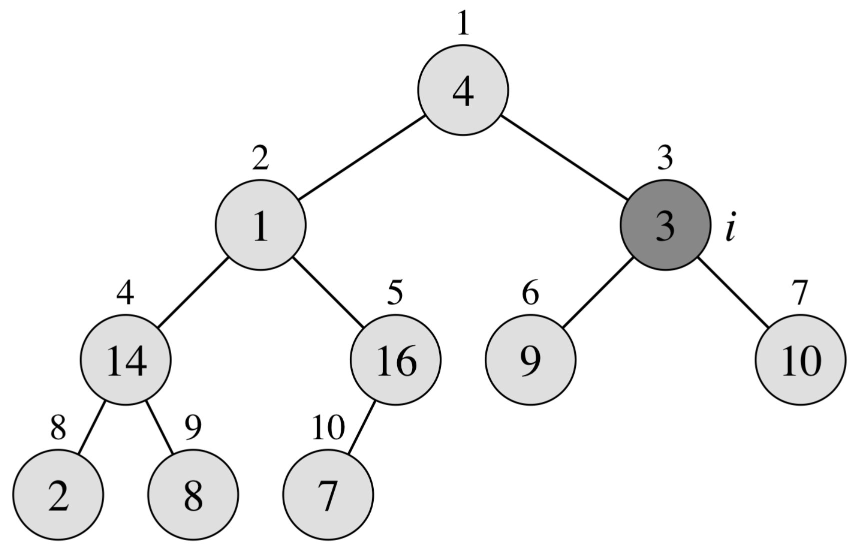
Building a (max) heap

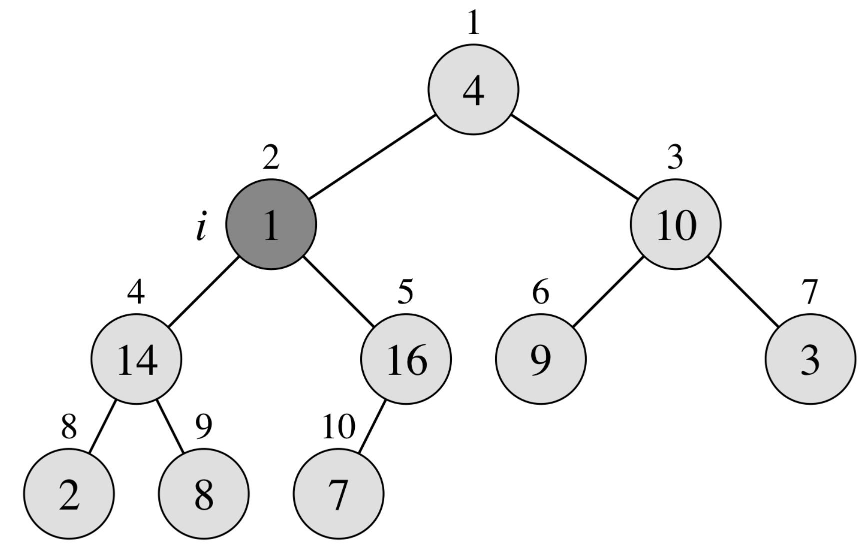


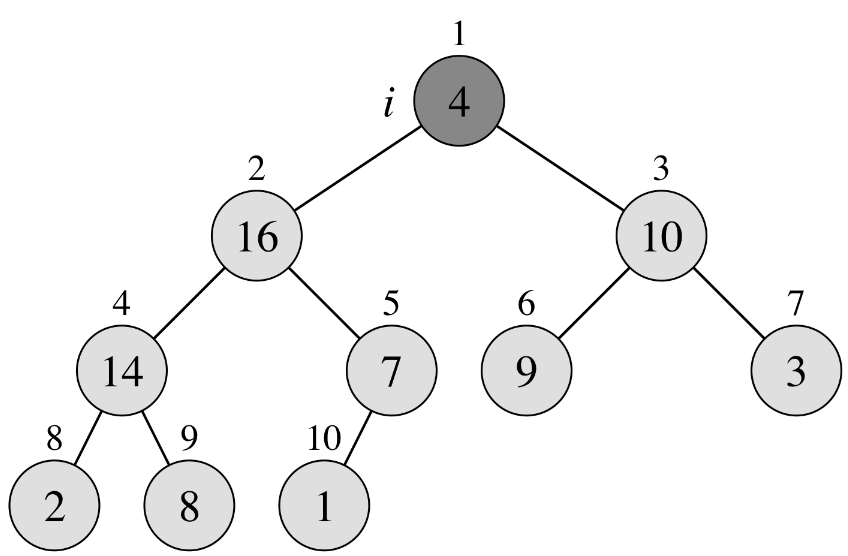

```
Input: A is an array

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

- With an array representation for storing an n-element heap, the leaves are the nodes indexed by $\lfloor n/2 \rfloor+1$, $\lfloor n/2 \rfloor+2$, …, n

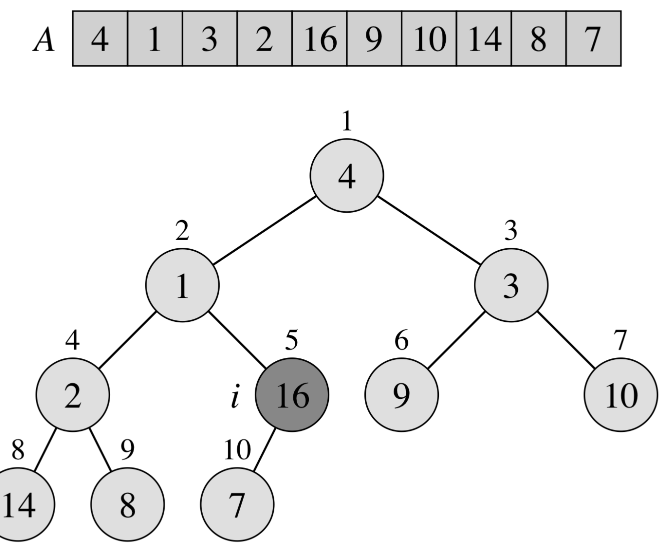






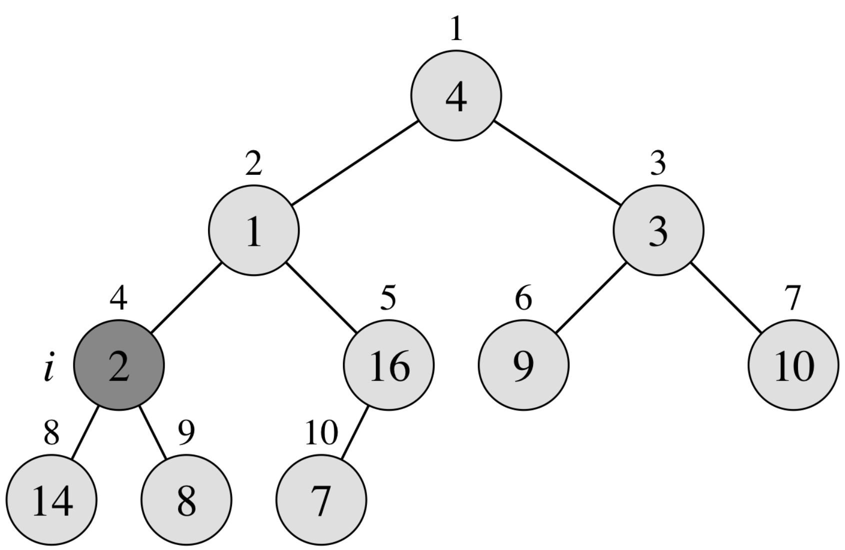


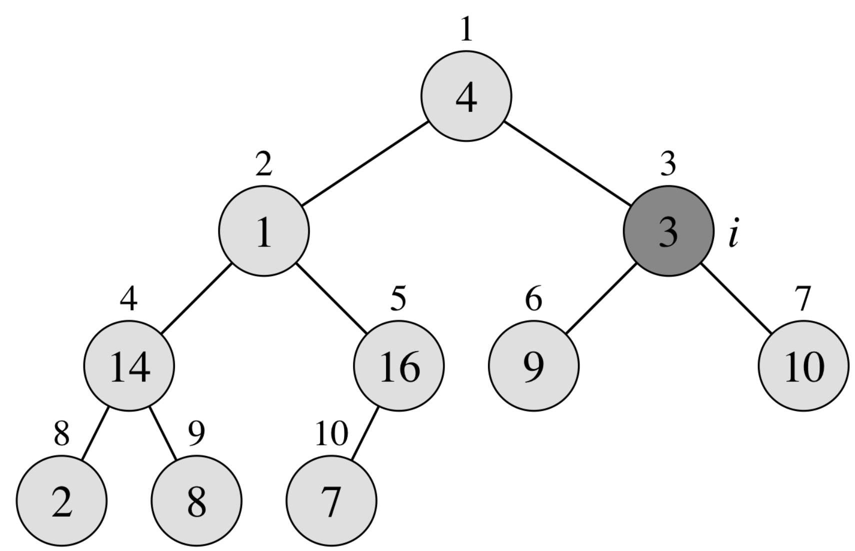
Building a (max) heap

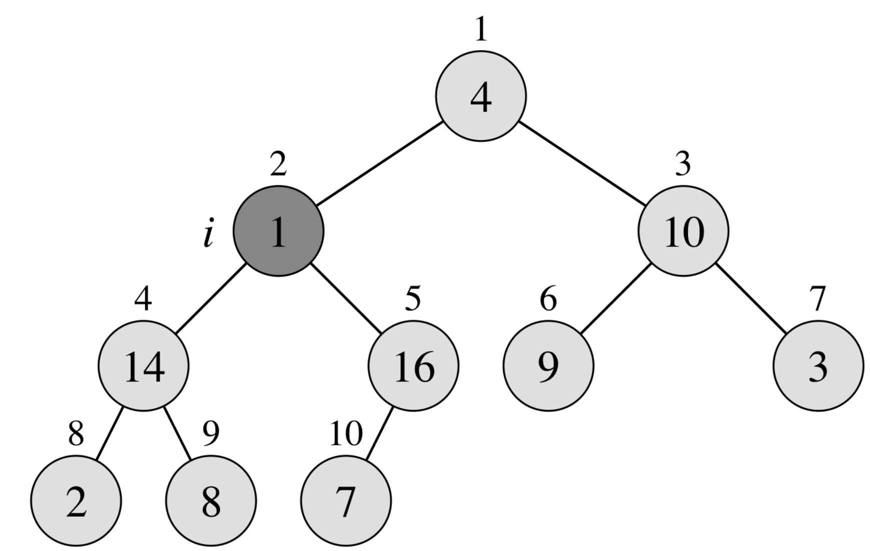


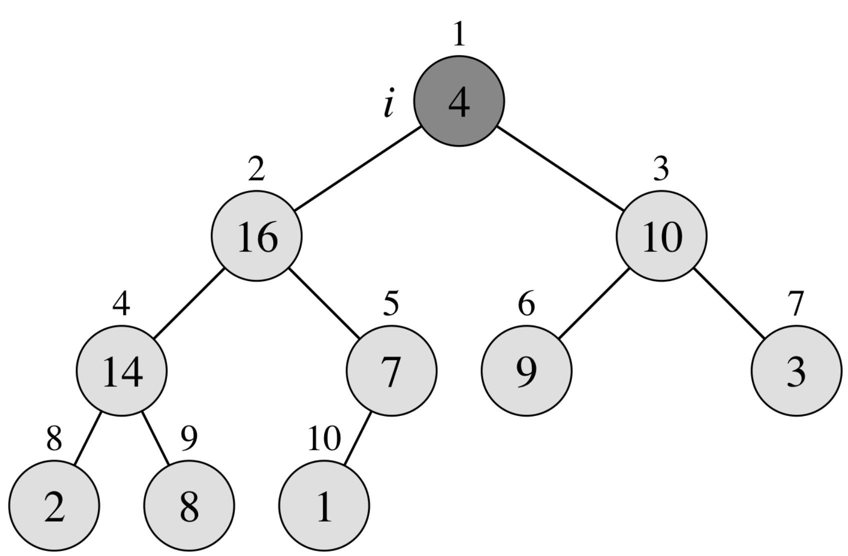

```
Input: A is an array

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

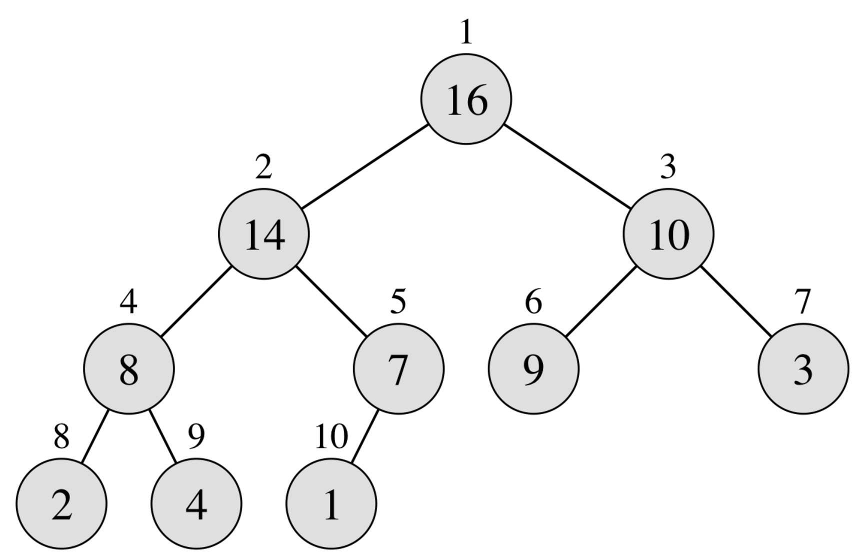
- With an array representation for storing an n-element heap, the leaves are the nodes indexed by ⌊n/2⌋+1, ⌊n/2⌋+2, …, n









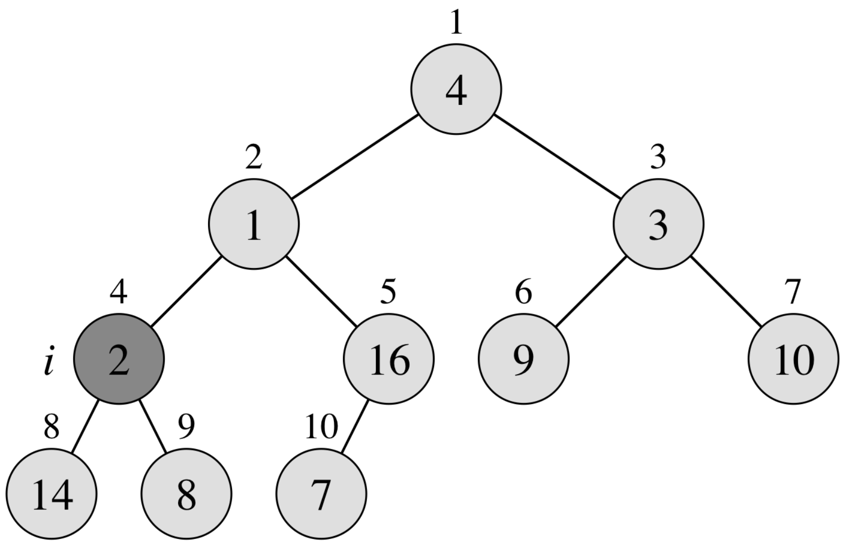


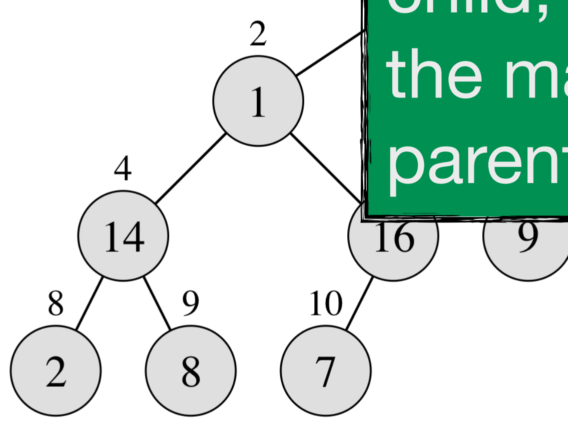
Building a (max) heap

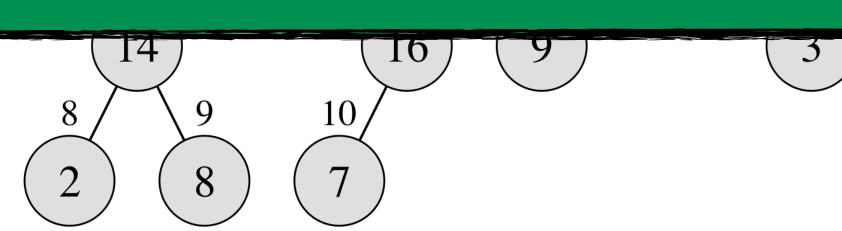
A									
4	1	3	2	16	9	10	14	8	7

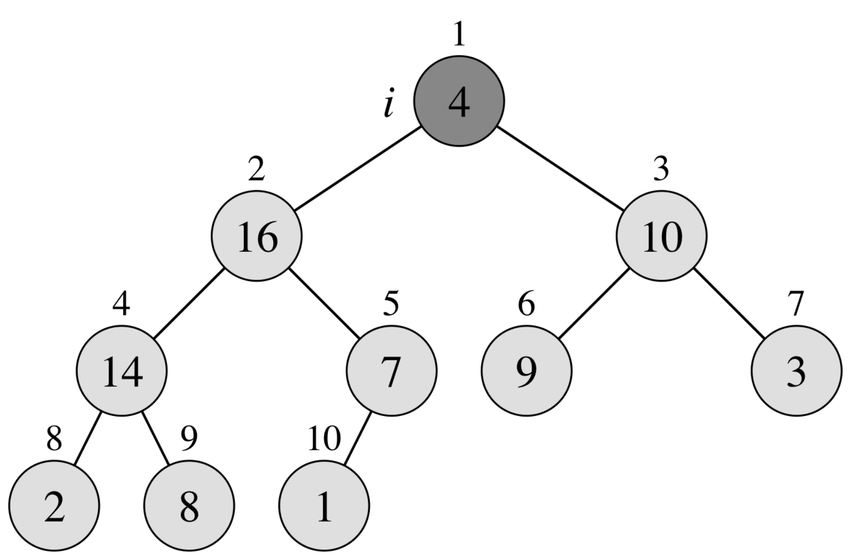
```
Input: A is an array

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

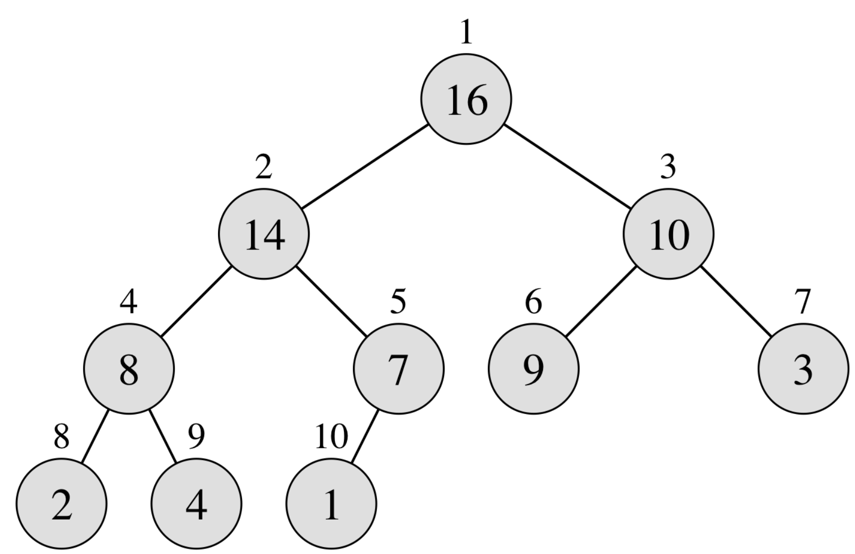
The loop could not go from 1 to ⌊length[A]/2⌋ because it could not guarantee the max-heap property.
E.g. A [2,1,1,3] then MAX-HEAPIFY won't exchange 2 with it's children (1's). However, when MAX-HEAPIFY is called on the left child, 1, it will swap 1 with 3. This violates the max-heap property because now 2 is the parent of 3.



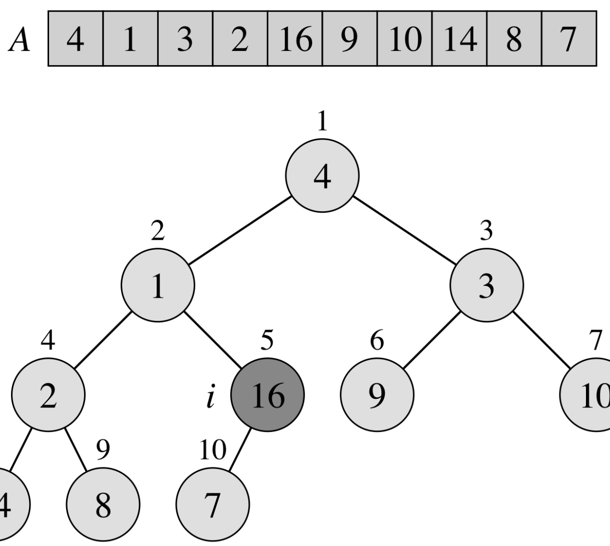






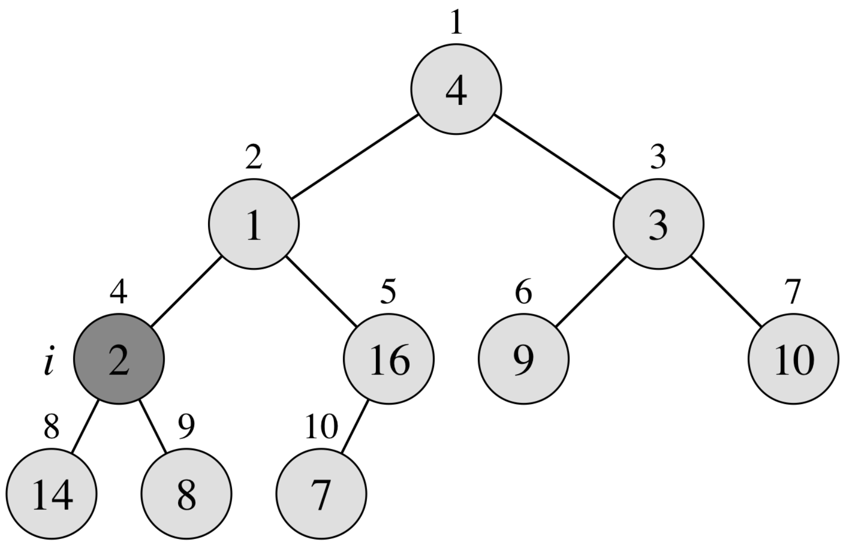


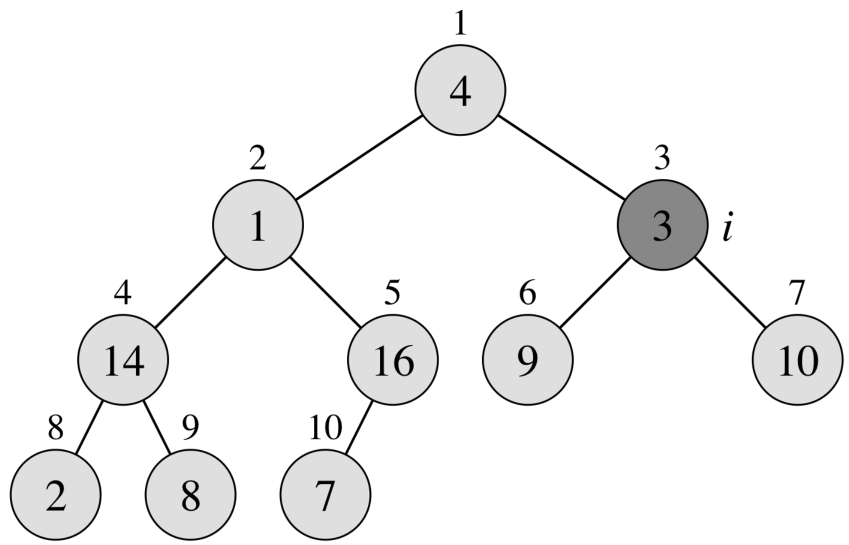
Building a (max) heap

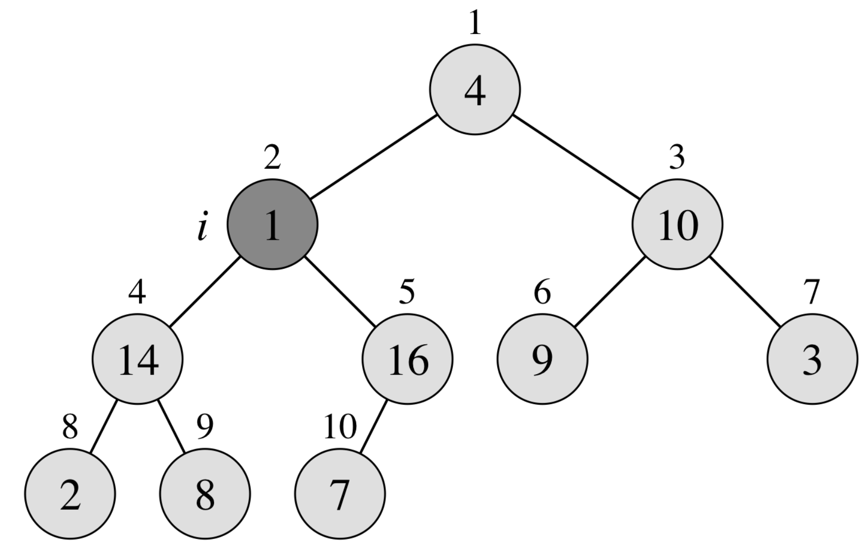


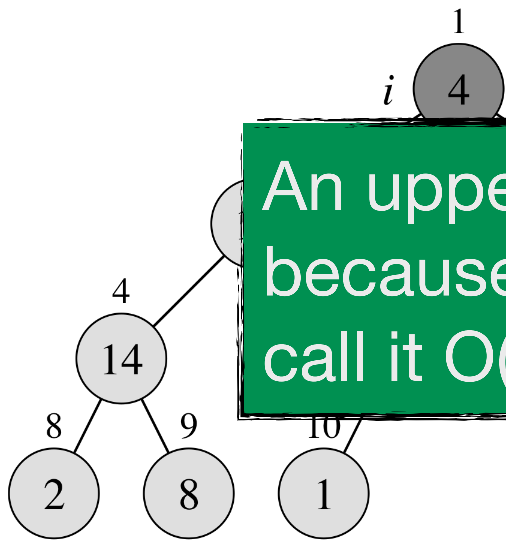

```
Input: A is an array

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

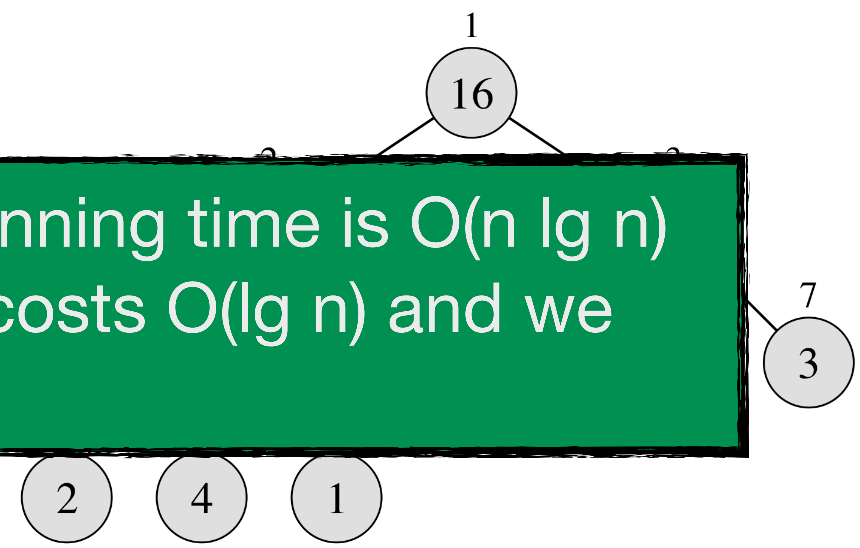
- With an array representation for storing an n-element heap, the leaves are the nodes indexed by $\lfloor n/2 \rfloor+1, \lfloor n/2 \rfloor+2, \ldots, n$





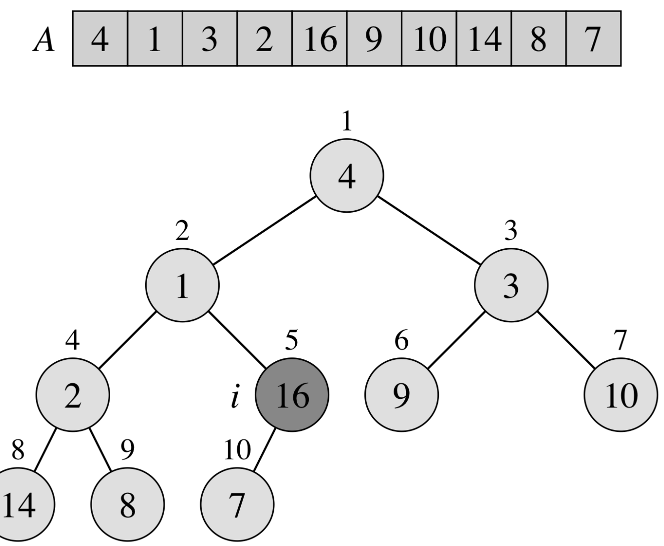






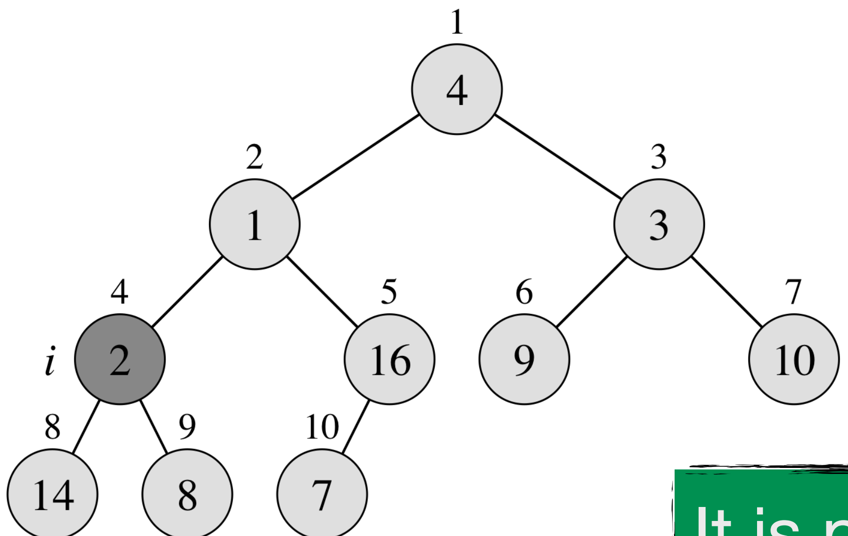
An upper bound of the running time is O(n lg n) because MAX-HEAPIFY costs O(lg n) and we call it O(n) times

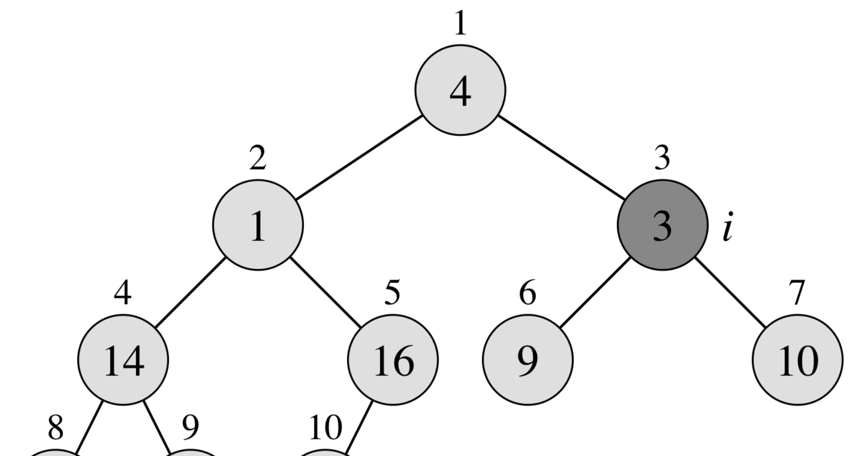
Building a (max) heap

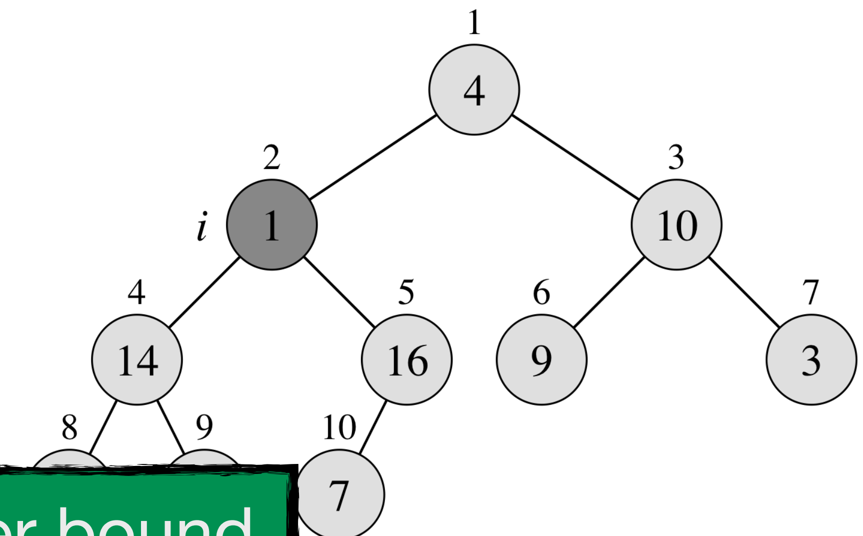


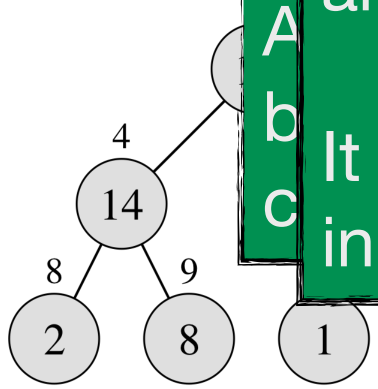

```
Input: A is an array

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

- With an array representation for storing an n-element heap, the leaves are the nodes indexed by $\lfloor n/2 \rfloor+1$, $\lfloor n/2 \rfloor+2$, …, n









It is possible to derive a tighter upper bound by observing that the time required by MAX-HEAPIFY varies with the height of the node and most heights are small…

It can be proved that BUILD-MAX-HEAP run in O(n)

Heapsort

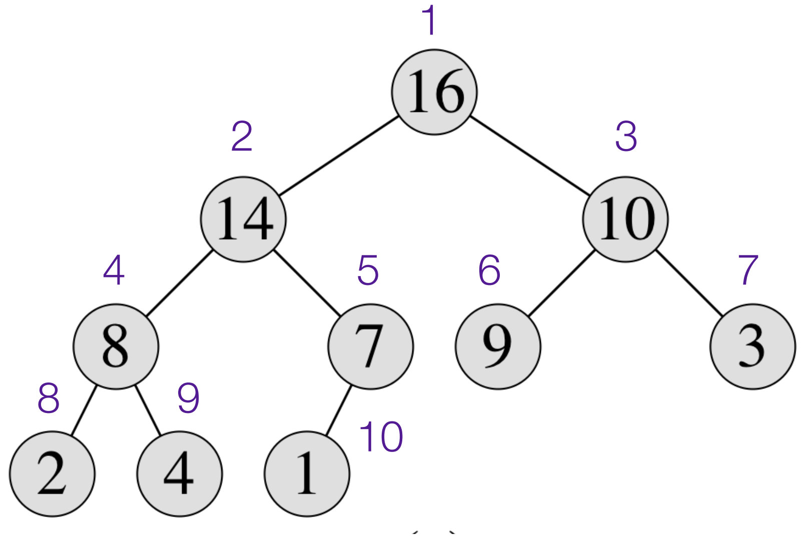
A <4 1 3 2 16 9 10 14 8 7> We need to sort this array

[Williams, J. Algorithm 232, CACM, 7(6), 1964]

Heapsort

A <4 1 3 2 16 9 10 14 8 7> We need to sort this array

We call BUILD-MAX-HEAP(A) A <16 14 10 8 7 9 3 2 4 1>



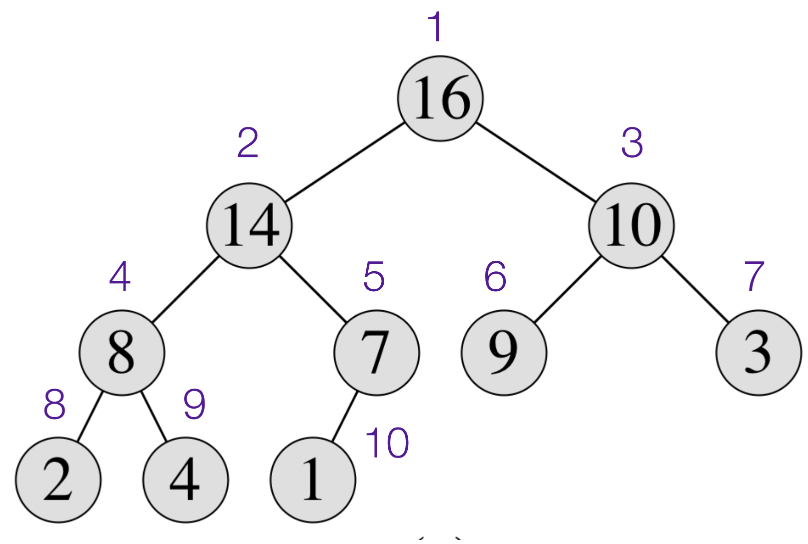
[Williams, J. Algorithm 232, CACM, 7(6), 1964]

Heapsort

A <4 1 3 2 16 9 10 14 8 7> We need to sort this array

We call BUILD-MAX-HEAP(A) A <16 14 10 8 7 9 3 2 4 1>

We exchange A[1] with A[i], i going from A.length down to 2 and we do heap-size=heap-size-1

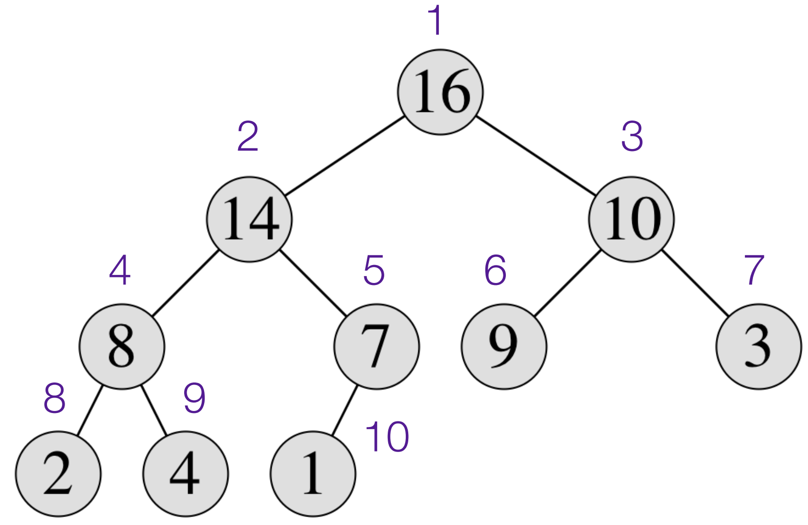


[Williams, J. Algorithm 232, CACM, 7(6), 1964]

Heapsort

A <4 1 3 2 16 9 10 14 8 7> We need to sort this array

We call BUILD-MAX-HEAP(A) A <16 14 10 8 7 9 3 2 4 1>



We exchange A[1] with A[i], i going from A.length down to 2 and we do heap-size=heap-size-1

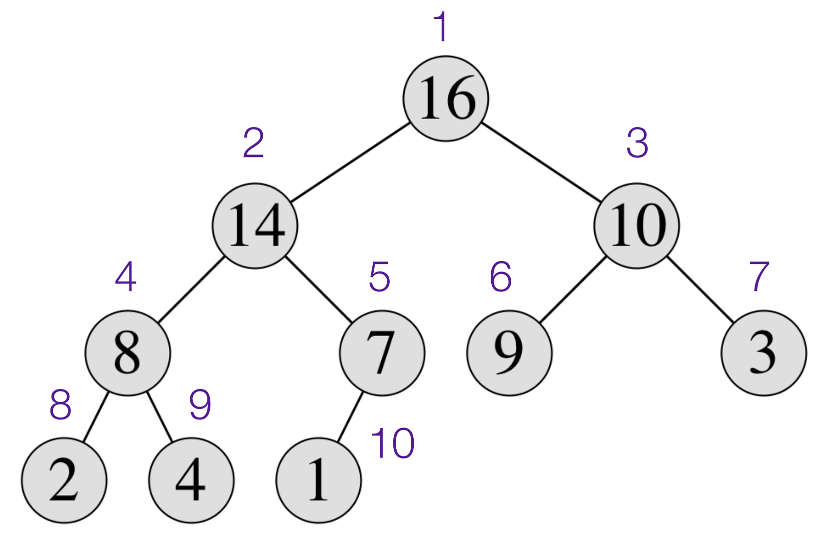
We exchange A[1] with A[10] and set heap-size to 9

A <1 14 10 8 7 9 3 2 4 16>

[Williams, J. Algorithm 232, CACM, 7(6), 1964]

Heapsort

A <4 1 3 2 16 9 10 14 8 7>   We need to sort this array



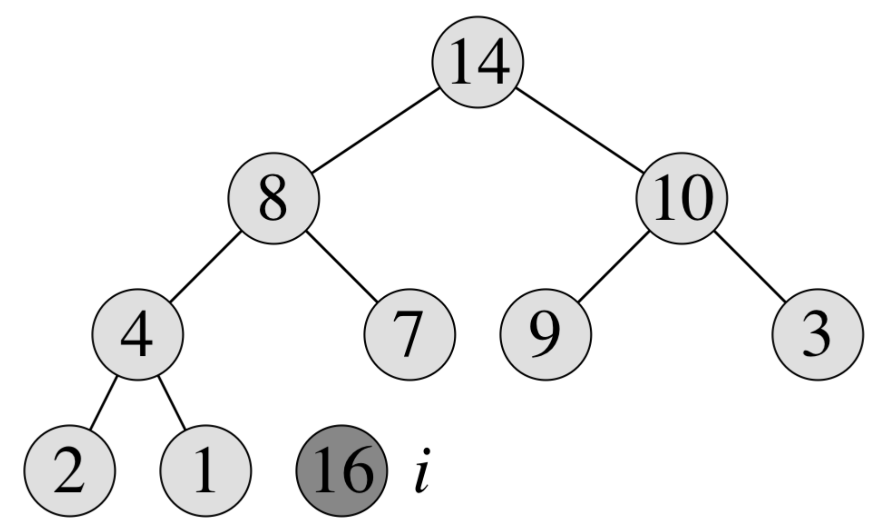
We call BUILD-MAX-HEAP(A)   A <16 14 10 8 7 9 3 2 4 1>

We exchange A[1] with A[i], i going from A.length down to 2
and we do heap-size=heap-size-1

We exchange A[1] with A[10] and set heap-size to 9

A <1 14 10 8 7 9 3 2 4 16>

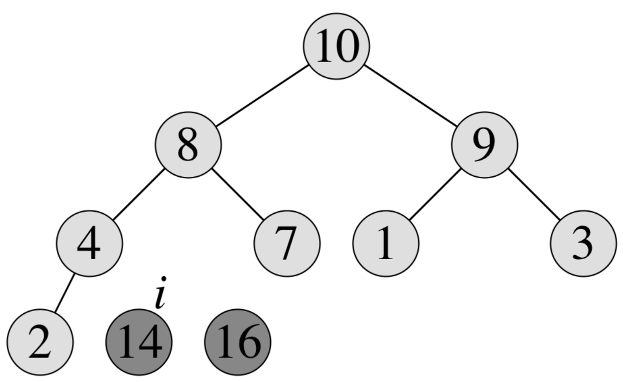
We call MAX-HEAPIFY(A,1)

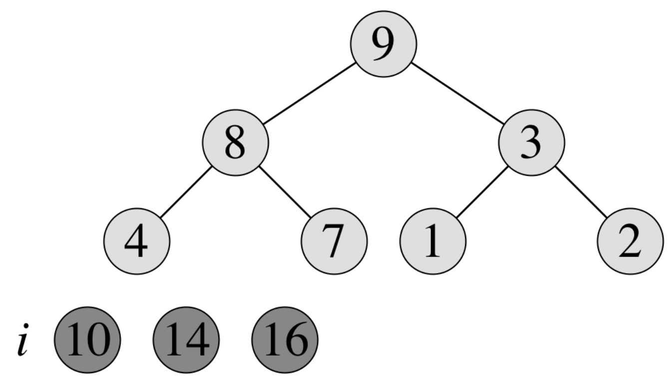


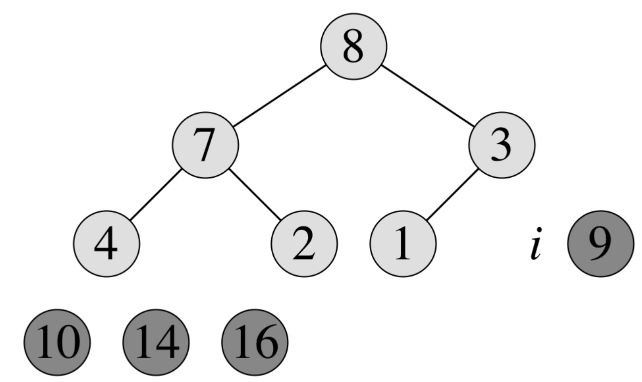
[Williams, J. Algorithm 232, CACM, 7(6), 1964]

Heapsort

We iterate the procedure down to i=2

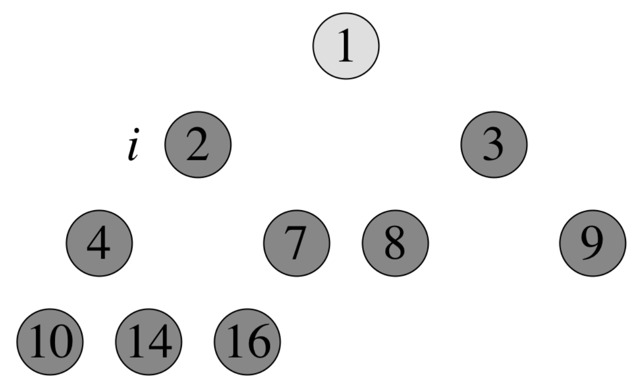






A <8 7 3 4 2 1 9 10 14 16>

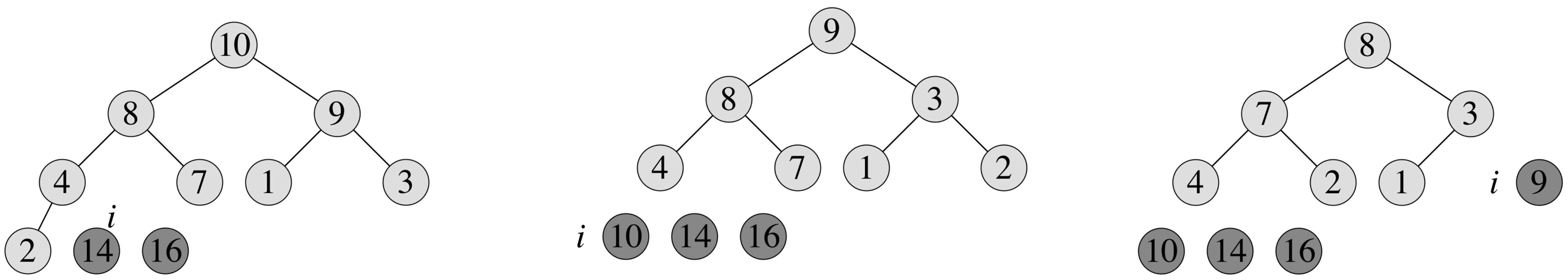
You keep going until you get



A <1 2 3 4 7 8 9 10 14 16>

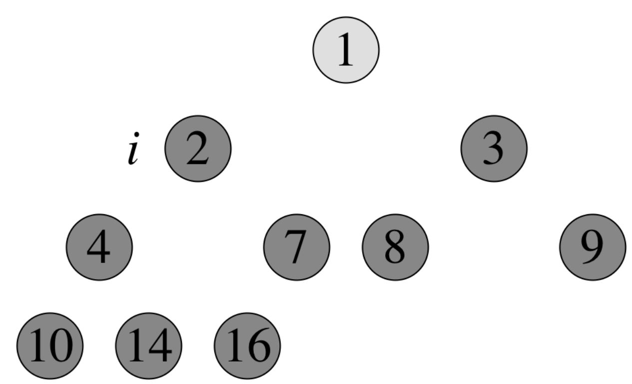
Heapsort

We iterate the procedure down to i=2



A <8 7 3 4 2 1 9 10 14 16>

You keep going until you get

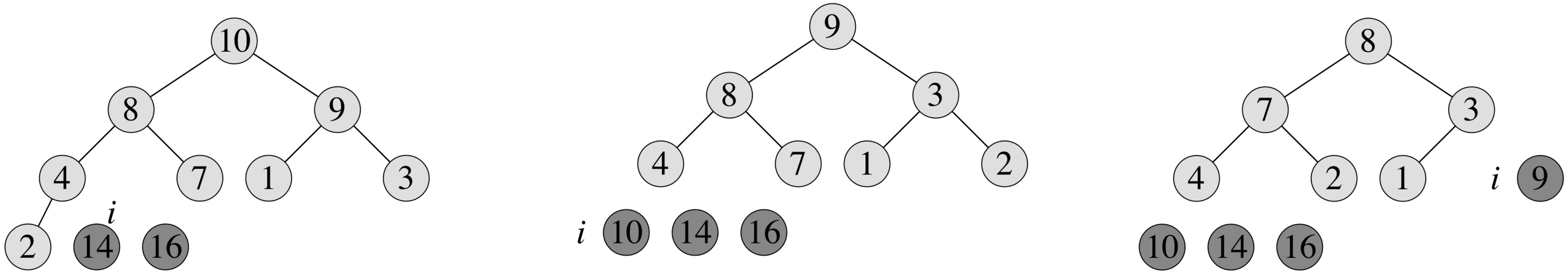


A <1 2 3 4 7 8 9 10 14 16>

```
Input: A is an array

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

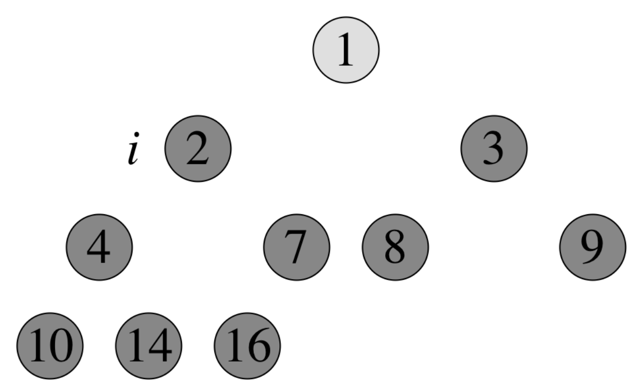
Heapsort

We iterate the procedure down to i=2



A <8 7 3 4 2 1 9 10 14 16>

You keep going until you get

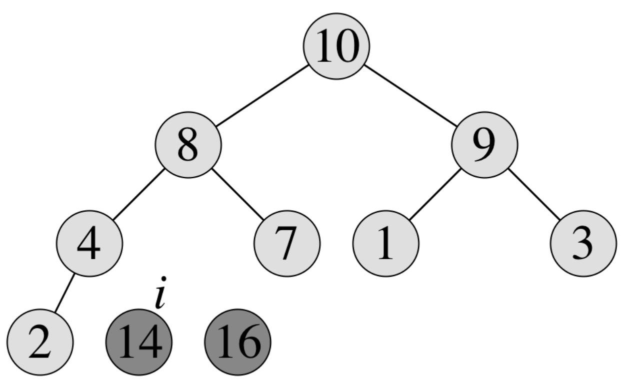


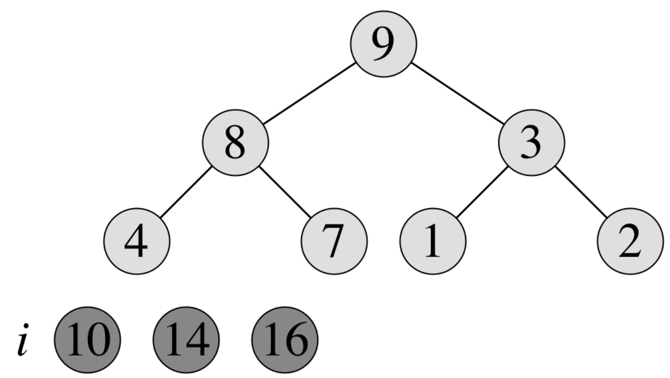
A <1 2 3 4 7 8 9 10 14 16>

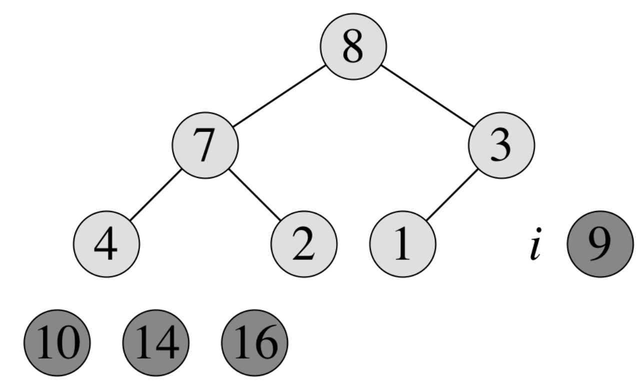
```
Input: A is an array

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

Heapsort

We iterate the procedure down to i=2

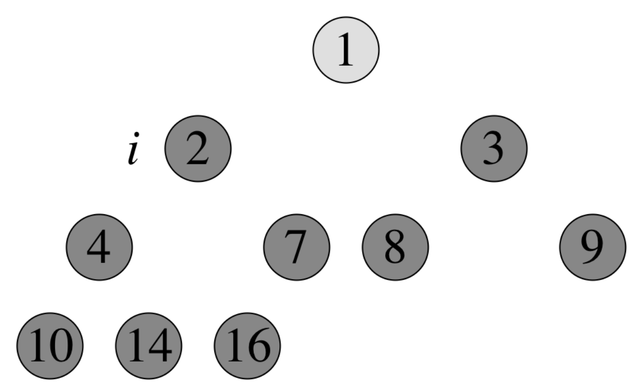






A <8 7 3 4 2 1 9 10 14 16>

You keep going until you get



A <1 2 3 4 7 8 9 10 14 16>

```
Input: A is an array

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

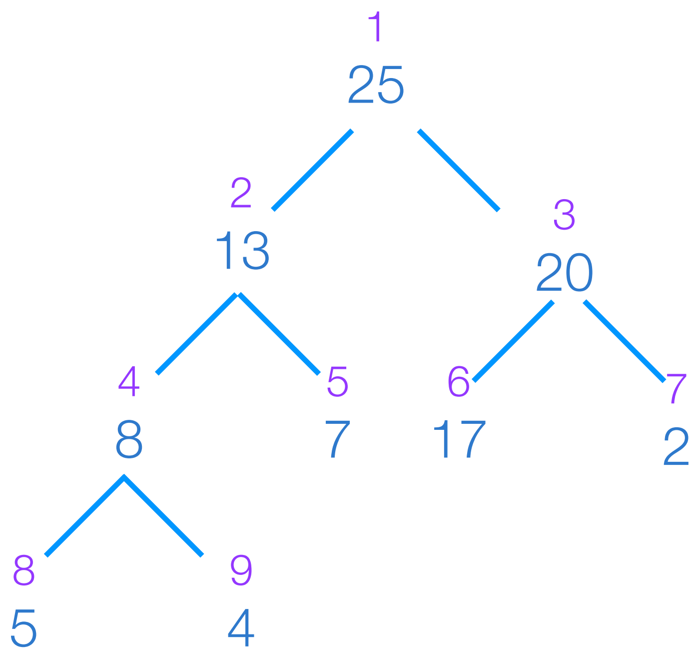
O(n) + O((n-1)(lg n)) = **O(n lg n)**

Exercise

Illustrate the operations of HEAPSORT on the array <5, 13, 2, 25, 7, 17, 20, 8, 4>

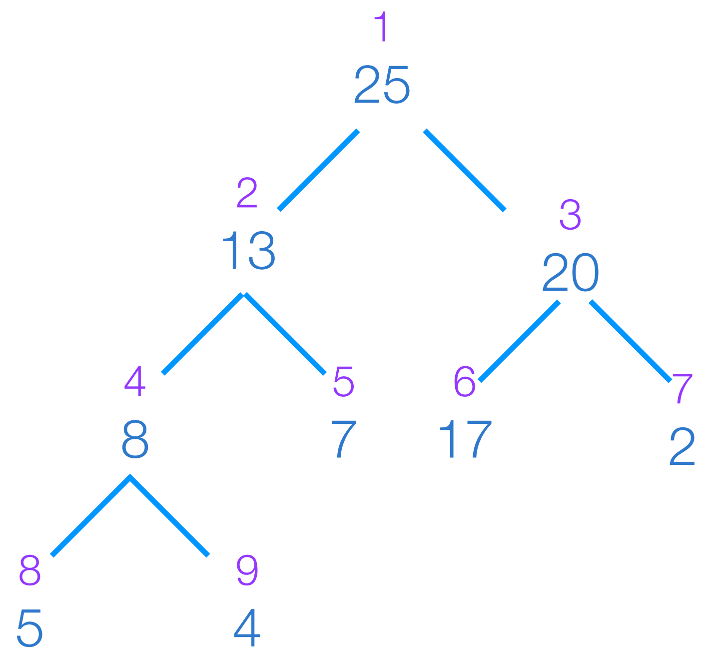
Exercise

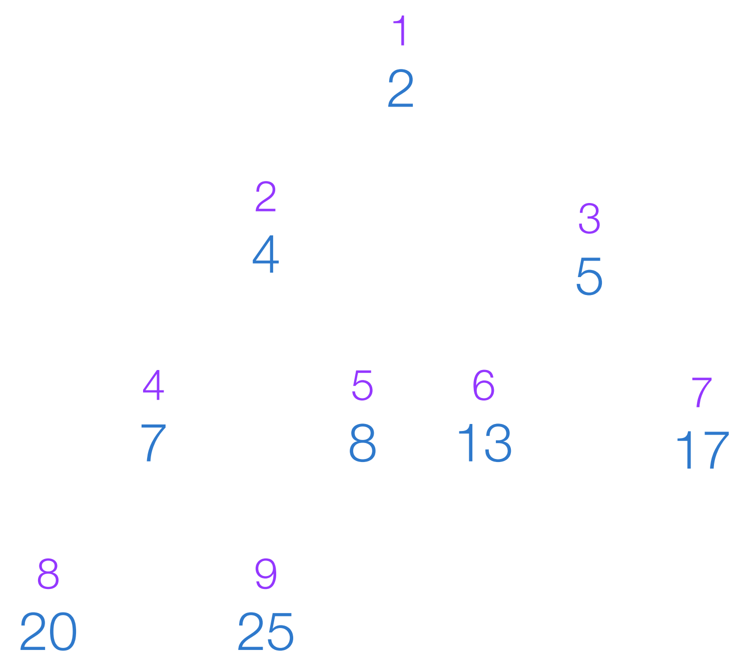
Illustrate the operations of HEAPSORT on the array <5, 13, 2, 25, 7, 17, 20, 8, 4>



Exercise

Illustrate the operations of HEAPSORT on the array <5, 13, 2, 25, 7, 17, 20, 8, 4>





Priority queues

- Heaps are used to implement efficient priority queues: max-priority queues and min-priority queues
- A *priority queue* is a data structure for maintaining a set S of elements, each with an associated *key*
- A **max-priority queue** maintaining a set of elements S supports the following operations:
  - INSERT(S,x): S = S U {x}
  - MAXIMUM(S): return the element with the max key in the queue
  - EXTRACT-MAX: return and remove the element with the max key in the queue
  - INCREASE-KEY(S, x, k): increase the value of x’s key to k (k is assumed to be larger than the current x’s key)

Priority queues

- Max-priority queues are used for instance for jobs scheduling in a computer where the key is the importance of the job
  - e.g. execute the job with maximum priority

- Min-priority queues are used for instance for event-driven simulations where the key of an event is its execution time
  - e.g. run the event requiring minimum time

Priority queues

- The costs of the operations in list-based priority queues are
  - INSERT(S,x): O(1)
  - MAXIMUM(S): O(n)
  - EXTRACT-MAX: O(n)

- If the priority queue is implemented with a heap:
  - INSERT(S,x): O(lg n)
  - MAXIMUM(S): O(1) <- it's the root of the max-heap
  - EXTRACT-MAX: O(lg n)