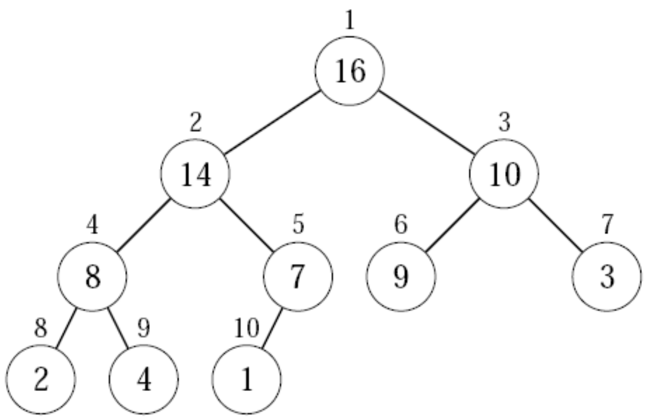
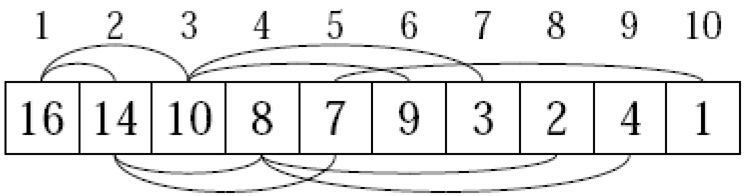
Chap 6 – Heapsort

6.1 Heaps

- Heap data structure
  - A (binary) heap is a nearly complete binary tree.





  - A heap can be stored as an array A.
    - Root of tree is $A[1]$
    - Parent of $A[i] = A[\lfloor i/2 \rfloor]$
    - Left child of $A[i] = A[2i]$
    - Right child of $A[i] = A[2i + 1]$

6.1 Heaps

- Height of heap

Height of an n-element heap

= height of the root of the heap

= # of edges on a longest simple path from the root to a leaf

= $\Theta(\lg n)$

To see this, let h be the height of an n-element heap, then

$$\sum_{i=0}^{h-1} 2^i < n \le \sum_{i=0}^{h} 2^i$$

$$\Rightarrow 2^h - 1 < n \le 2^{h+1} - 1$$

$$\Rightarrow 2^h \le n < 2^{h+1}$$

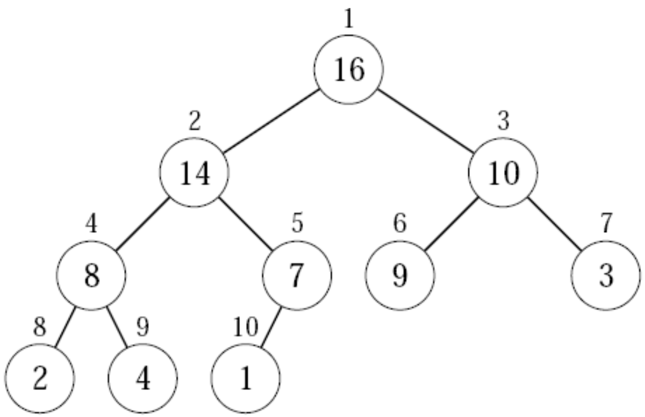
$$\Rightarrow h \le \lg n < h + 1 \Rightarrow h = \lfloor \lg n \rfloor$$ (Ex. 6.1-2)

6.1 Heaps

- Heap property

  For max-heaps (largest element at root), max-heap property:

  for all nodes i, excluding the root, $A[\textsc{Parent}(i)] \geq A[i]$



$\Leftarrow$ a max heap

  For min-heaps (smallest element at root), min-heap property:

  for all nodes i, excluding the root, $A[\textsc{Parent}(i)] \leq A[i]$

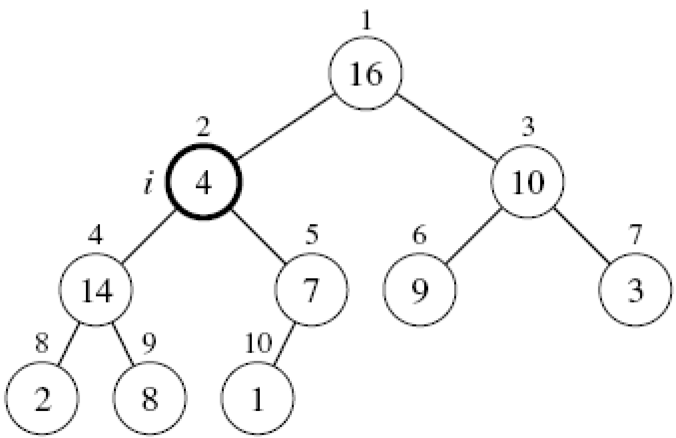
6.2 Maintaining the heap property

- Maintaining a max-heap
  - Before MAX-HEAPIFY, $A[i]$ may be smaller than its children.
  - Assume left and right subtrees of i are max-heaps.
  - After MAX-HEAPIFY, subtree rooted at i is a max-heap.

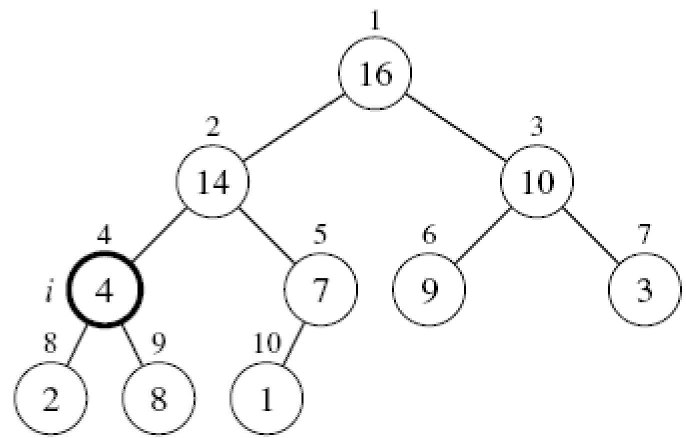
```
MAX-HEAPIFY(A, i, n)
l = LEFT(i)
r = RIGHT(i)
if l ≤ n and A[l] > A[i]
    largest = l
else largest = i
if r ≤ n and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i], A[largest]
    MAX-HEAPIFY(A, largest, n)
```

6.2 Maintaining the heap property

- Maintaining a max-heap



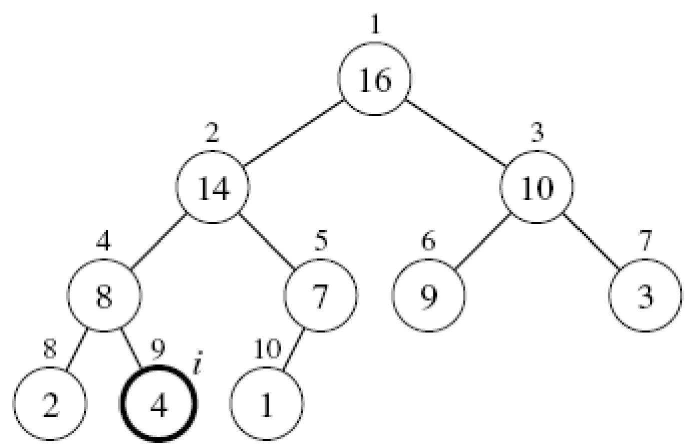
(a)



(b)

  - Example

    MAX-HEAPIFY$(A, 2, 10)$



(c)

6.2 Maintaining the heap property

- Running time of MAX-HEAPIFY

  Let $T(n)$ = the running time of MAX-HEAPIFY on a subtree of size n rooted at node i

  [As in book, this is NOT worst-case analysis.]

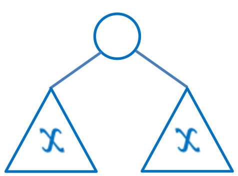
  Then,

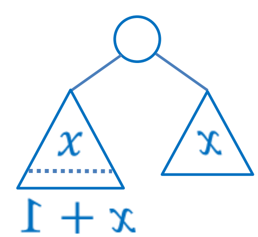
  $T(n) = \Omega(1)$

  $T(n) = O(\text{height of the subtree}) = O(\lg n)$

  Alternative analysis with recurrence

  Roughly, $n/2 \leq \#\text{ of nodes in the left subtree} \leq 2n/3$





6.2 Maintaining the heap property

- Running time of MAX-HEAPIFY

  Therefore,

  $$T(n) \le T(2n/3) + \Theta(1) \Rightarrow T(n) = O(\lg n)$$

  In details, let

  $$T'(n) = T'(2n/3) + \Theta(1)$$

  Then,

  $$T(n) \le T'(n) = \Theta(\lg n) \Rightarrow T(n) = O(\lg n)$$

6.2 Maintaining the heap property

- Running time of MAX-HEAPIFY

  As for the worst-case running time, let

  $T(n)$ = the worst-case running time of MAX-HEAPIFY on a subtree of size n rooted at node i, then

  $T(n) = \Theta(\text{height of the subtree}) = \Theta(\lg n)$

  Alternative analysis with recurrence

  Again, $n/2 \leq \#\text{ of nodes in the left subtree} \leq 2n/3$

  Therefore,

  $T(n) \geq T(n/2) + \Theta(1) \Rightarrow T(n) = \Omega(\lg n)$

  $T(n) \leq T(2n/3) + \Theta(1) \Rightarrow T(n) = O(\lg n)$

6.3 Building a heap

- Building a max-heap

  BUILD-MAX-HEAP(A, n)

  for $i = \lfloor n/2 \rfloor$ **downto** 1

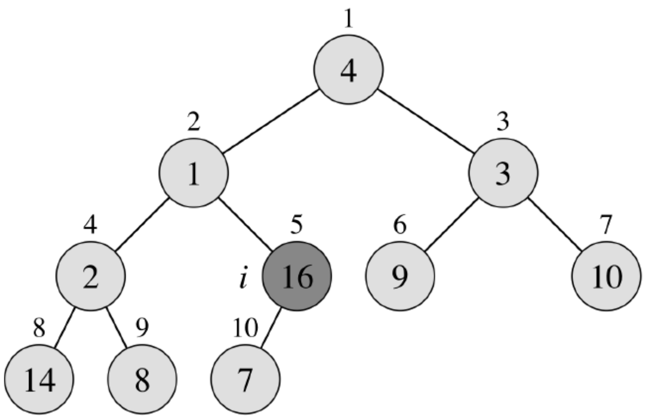
  MAX-HEAPYFY(A, i, n)

  N.B. $A[i], i = \lfloor n/2 \rfloor + k, 1 \leq k \leq \lceil n/2 \rceil$ are leaves.

  $\because 2(\lfloor n/2 \rfloor + k) \geq 2\left(\frac{n-1}{2} + k\right) \geq n + (2k - 1) > n$

  Example – BUILD-MAX-HEAP$(A, 10)$

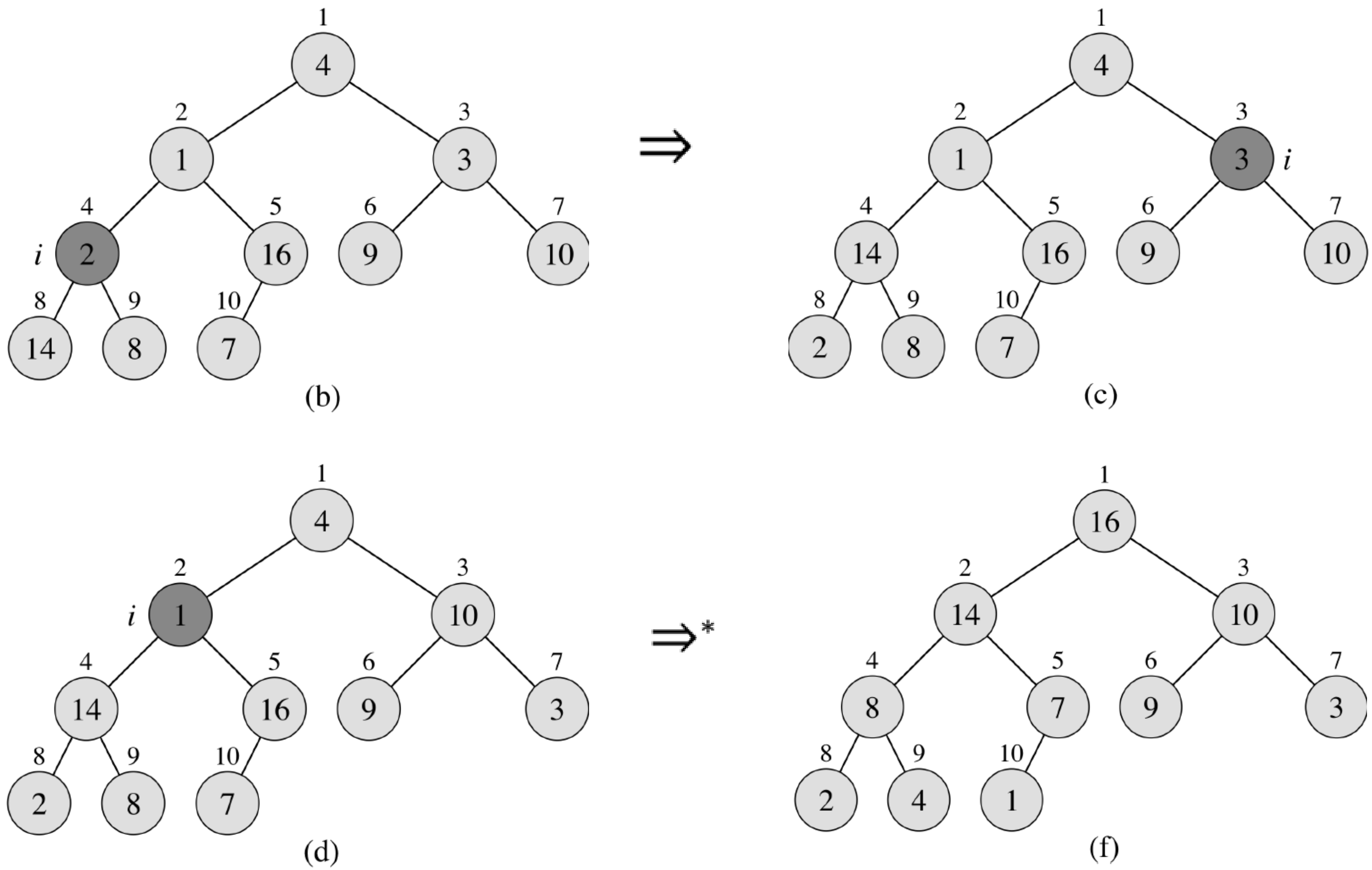
	1	2	3	4	5	6	7	8	9	10
A	4	1	3	2	16	9	10	14	8	7



(a)

6.3 Building a heap

- Building a max-heap



6.3 Building a heap

- Running time of BUILD-MAX-HEAP

  Let

  $T(n)$ = running time of BUILD-MAX-HEAP on an array of size n

  [As in book, this is NOT worst-case analysis.]

  Lower bound

  $T(n) = \Omega(n)$, due to the **for** loop

  Simple upper bound

  $O(n)$ calls to MAX-HEAPIFY, each of which takes $O(\lg n)$ time

  $\Rightarrow O(n \lg n)$ time in total

  For a tighter upper bound, we need

  LEMMA Height of an n-element heap $= \lfloor \lg n \rfloor$

6.3 Building a heap

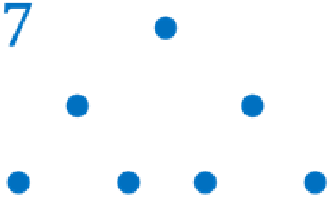
- Running time of BUILD-MAX-HEAP

  LEMMA (Ex. 6.3-3)

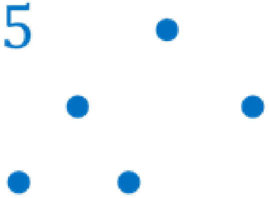
  # of nodes of height h in an n-element heap $\leq \lceil n/2^{h+1} \rceil$

  N.B. For a complete binary tree, it's easy to show that there are exactly $\lceil n/2^{h+1} \rceil$ nodes of height h.

$n = 7$



$n = 5$



height	$n = 7$	$n = 5$
2	$\lceil 7/2^{2+1} \rceil = 1$	$\lceil 5/2^{2+1} \rceil = 1$
1	$\lceil 7/2^{1+1} \rceil = 2$	$\lceil 5/2^{1+1} \rceil = 2$
0	$\lceil 7/2^{0+1} \rceil = 4$	$\lceil 5/2^{0+1} \rceil = 3$

6.3 Building a heap

- Running time of BUILD-MAX-HEAP

$$T(n) = \sum_{i=1}^{\lfloor n/2 \rfloor} O(\text{height of node } i)$$

$$\leq \sum_{h=1}^{\lfloor \lg n \rfloor} \lceil n/2^{h+1} \rceil O(h)$$ Book uses $h = 0$ and $O(0)$

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

$$= O\left(\sum_{h=1}^{\lg n} (n/2^{h+1}) h \right)$$ Remove floor and ceiling

$$= O\left(n \sum_{h=1}^{\lg n} h/2^{h} \right)$$ Remove $1/2$

6.3 Building a heap

- Running time of BUILD-MAX-HEAP

$$T(n) = O\left(n \sum_{h=1}^{\lg n} h/2^h\right)$$
$$= O\left(n \sum_{h=1}^{\infty} h/2^h\right)$$
$$\because \sum_{h=1}^{\lg n} h/2^h \geq \sum_{h=\lg n+1}^{\infty} h/2^h \quad \forall n \geq 4$$
$$= O(n)$$
$$\because \sum_{h=1}^{\infty} h/2^h = \frac{1/2}{(1-1/2)^2} = 2 \text{ (Formula A.8)}$$

In conclusion, the running time of BUILD-MAX-HEAP is $\Theta(n)$.

It follows that the worst-case running time is also $\Theta(n)$.

6.4 The heapsort algorithm

- Heapsort

	As in book	Worst case
HEAPSORT(A, n)		
BUILD-MAX-HEAP(A, n)	$O(n)$	$\Theta(n)$
for $i = n$ **downto** 2		
exchange $A[1]$ and $A[i]$		
MAX-HEAPYFY$(A, 1, i-1)$	$O(\lg n)$	$\Theta(\lg i)$

Running time (as in book)

$$O(n) + (n-1)O(\lg n) = O(n \lg n)$$

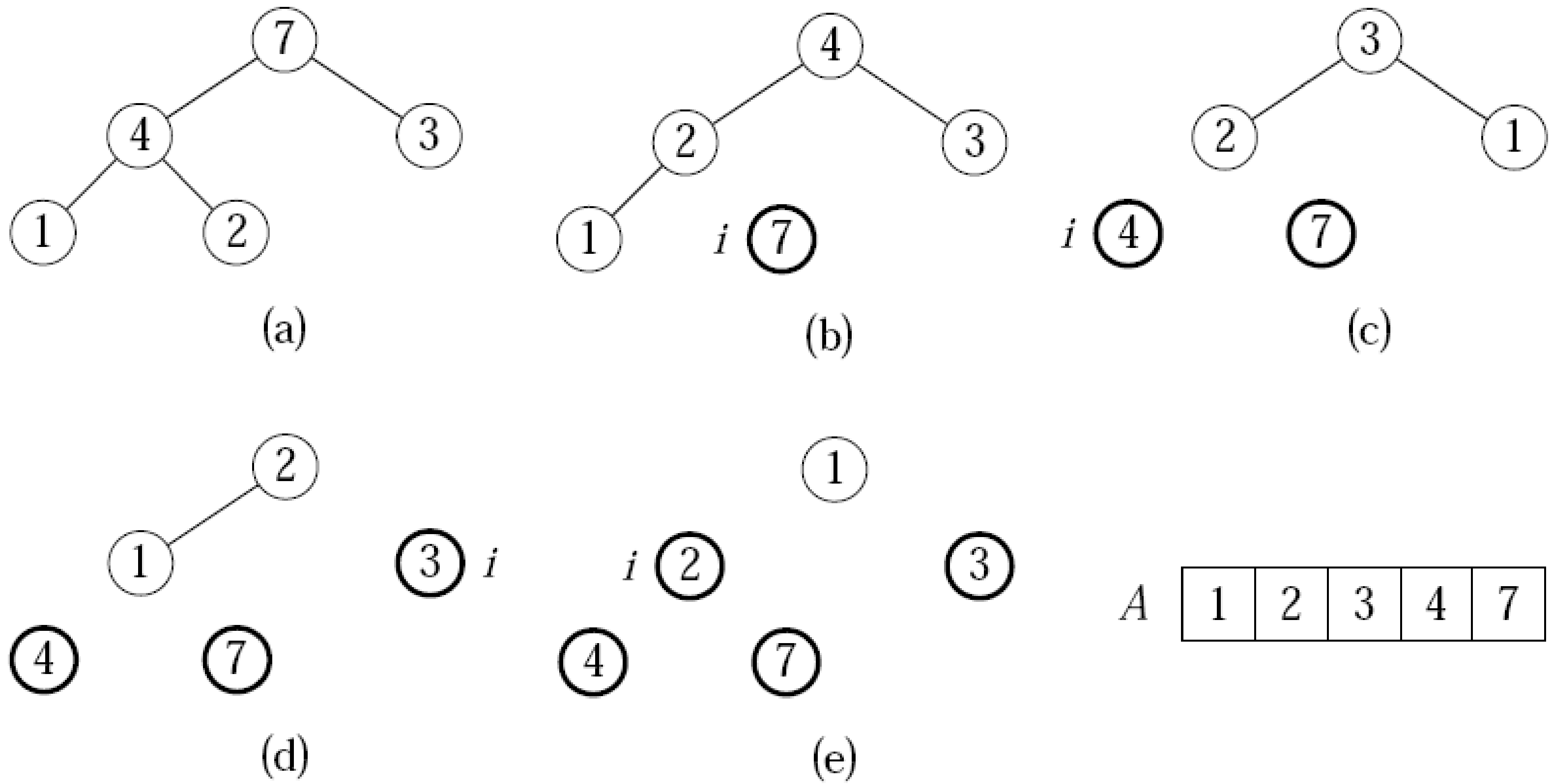
Worst-case running time

$$\Theta(n) + \sum_{i=2}^{n} \Theta(\lg i) = \Theta(n) + \Theta\left(\sum_{i=2}^{n} \lg i\right)$$
$$= \Theta(n) + \Theta(n \lg n) = \Theta(n \lg n)$$

6.4 The heapsort algorithm

- Heapsort

  Example – $\text{HEAPSORT}(A, 5)$



6.5 Priority queues

- Priority queue
  - A data structure that maintains a dynamic set of elements.
  - Each element has a key – an associated value.
- Max-priority queue

  Max-priority queue supports dynamic-set operations:
  - INSERT(S, x): inserts element x into set S
  - MAXIMUM(S): returns element of S with largest key
  - EXTRACT-MAX(S): removes and returns element of S with largest key
  - INCREASE-KEY(S, x, k): increases value of element x's key to k, assuming $k \geq x's$ current key value.

6.5 Priority queues

- Min-priority queue

  Min-priority queue supports similar operations:

  - INSERT(S, x) : inserts element x into set S
  - MINIMUM(S) : returns element of S with smallest key
  - EXTRACT-MIN(S) : removes and returns element of S with smallest key
  - DECREASE-KEY(S, x, k): decreases value of element x's key to k, assuming $k \leq x's$ current key value.

6.5 Priority queues

- Heap implementation of max-priority queue

HEAP-MAXIMUM(A) Time: $\Theta(1)$

return $A[1]$

HEAP-EXTRACT-MAX(A, n) Time: $O(\lg n)$

if $n < 1$ **error** "heap underflow" Worst case: $\Theta(\lg n)$

$max = A[1]$

$A[1] = A[n]$

MAX-HEAPIFY$(A, 1, n-1)$

return max

6.5 Priority queues

- Heap implementation of max-priority queue

HEAP-INCREASE-KEY(A, i, key) — Time: $O(\lg n)$

if $key < A[i]$ — Worst case: $\Theta(\lg i)$

    error "new key < current key"

$A[i] = key$

while $i > 1$ and $A[\text{PARENT}(i)] < A[i]$

    exchange $A[i]$ with $A[\text{PARENT}(i)]$

    $i = \text{PARENT}(i)$

MAX-HEAP-INSERT(A, key, n) — Time: $O(\lg n)$

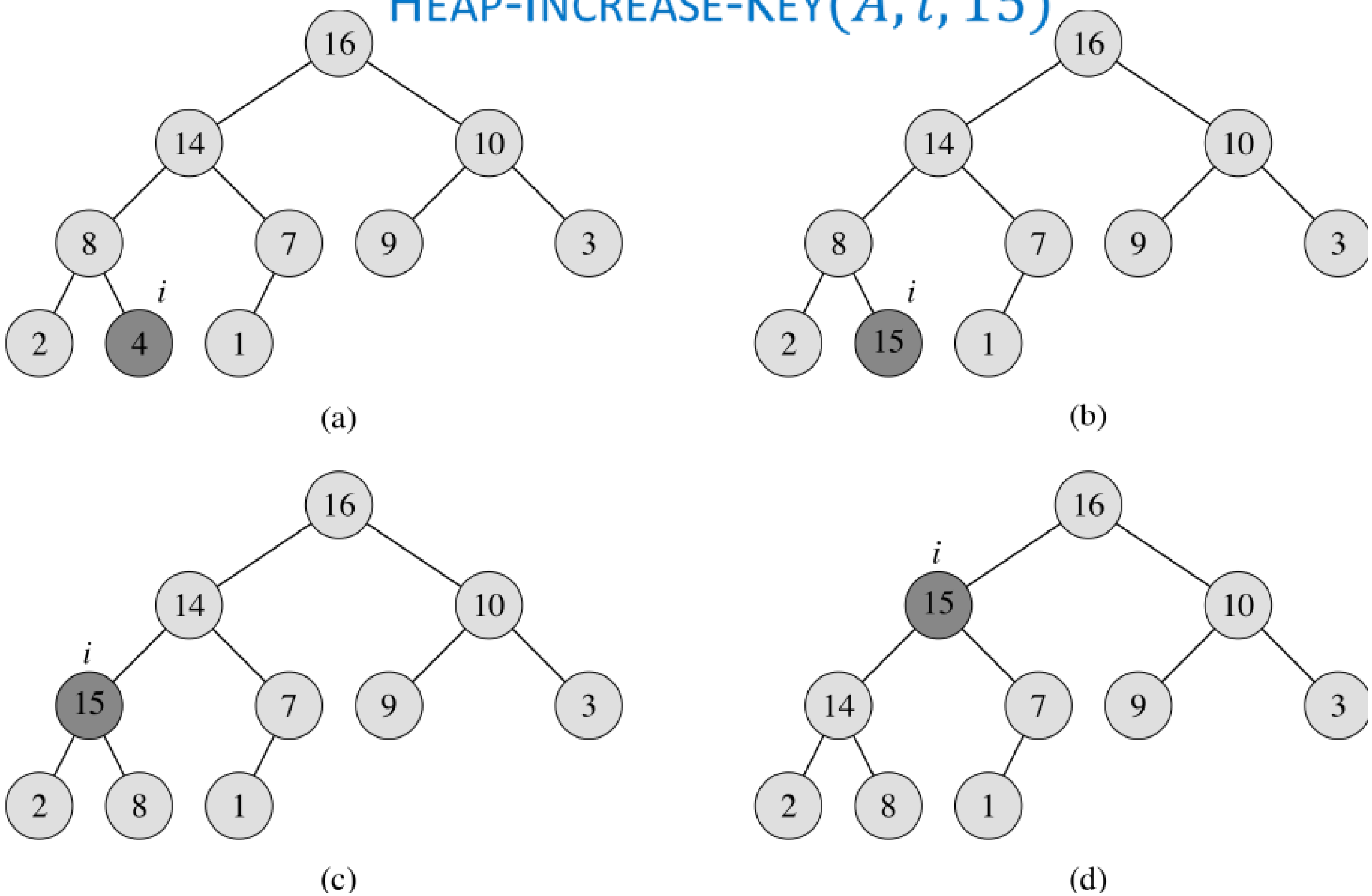
$A[n+1] = -\infty$ — Worst case: $\Theta(\lg n)$

HEAP-INCREASE-KEY$(A, n+1, key)$

6.5 Priority queues

- Heap implementation of max-priority queue

HEAP-INCREASE-KEY$(A, i, 15)$



(a) (b) (c) (d)