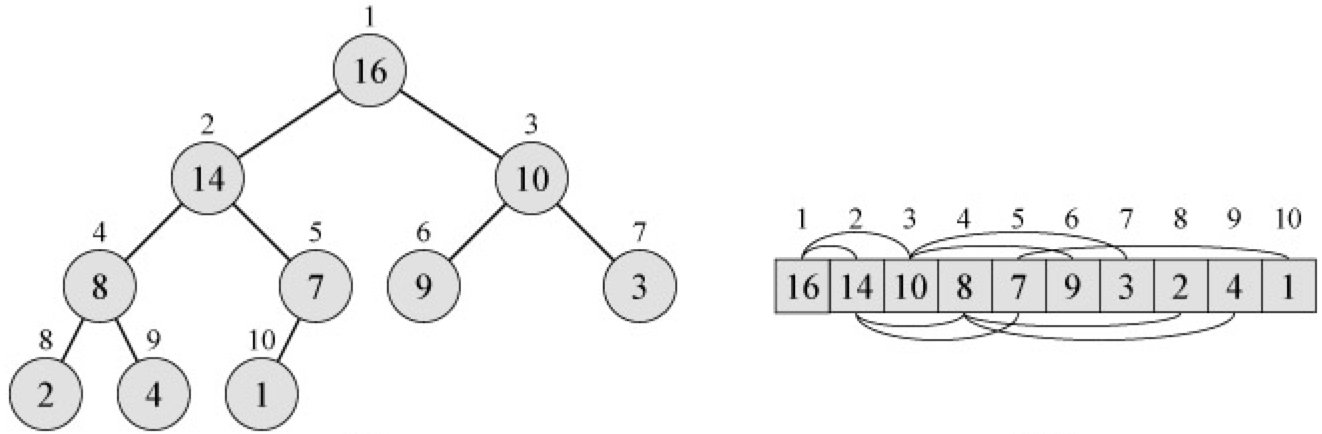
# Lecture 7: Heaps and Heapsort

An $O(n \log n)$-time in-place sorting algorithm

# (Binary) Heap

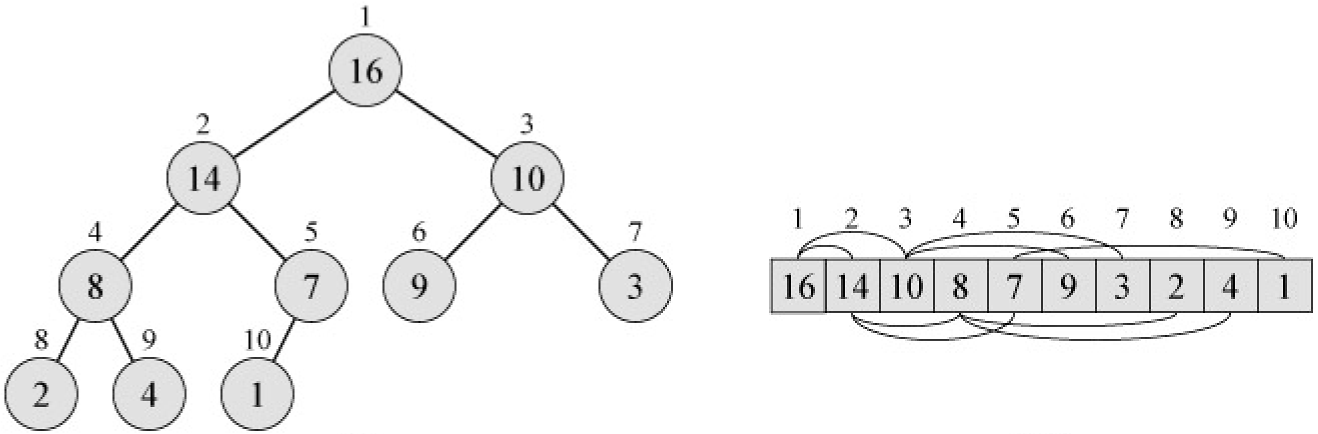



(a) (b)

Structure of a heap: An almost complete binary tree

- All levels are full except possibly the lowest level
- If the lowest level is not full, then nodes must be packed to the left
- Allows us to store the heap in an array (no pointers needed!)
  - For any element in array position $i$:
    Left child is in position $2i$
    Right child in position $2i + 1$
    The parent is in position $\lfloor i/2 \rfloor$
- The height of a heap with $n$ elements is $\Theta(\log n)$

# (Binary) Heap




(a) (b)

Property of a max-heap: Parent $\geq$ child

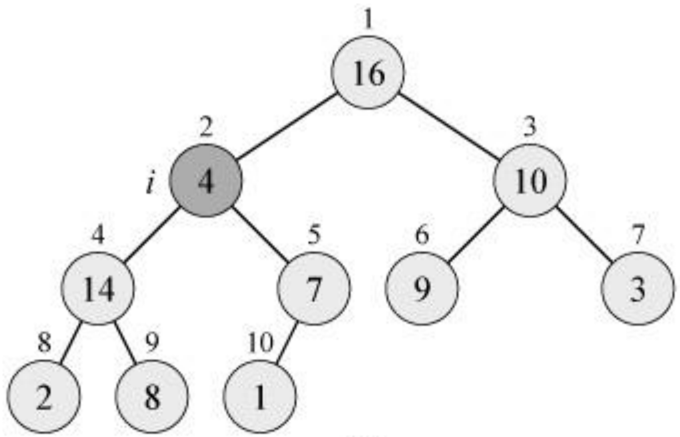
- Consequence: root is the maximum element in the heap

Property of a min-heap: Parent $\leq$ child

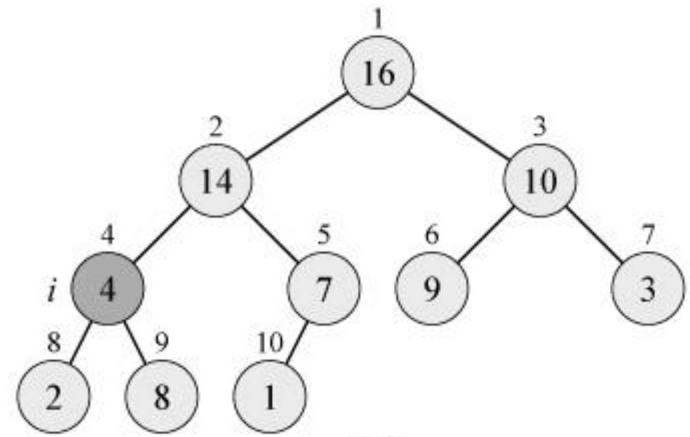
- Consequence: root is the minimum element in the heap

Note: We will assume a max-heap by default, but everything applies to a min-heap by symmetry.

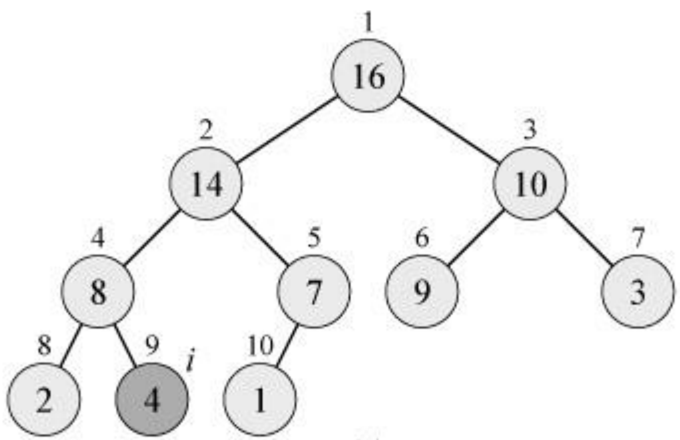
# Maintaining the heap property



(a)



(b)



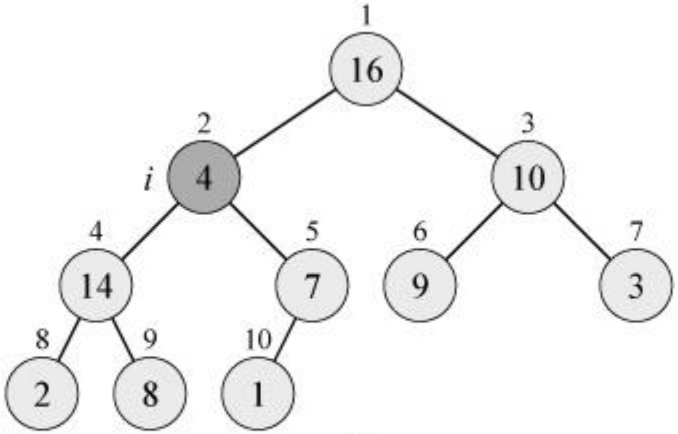
(c)

Given: A node $i$ in a heap such that

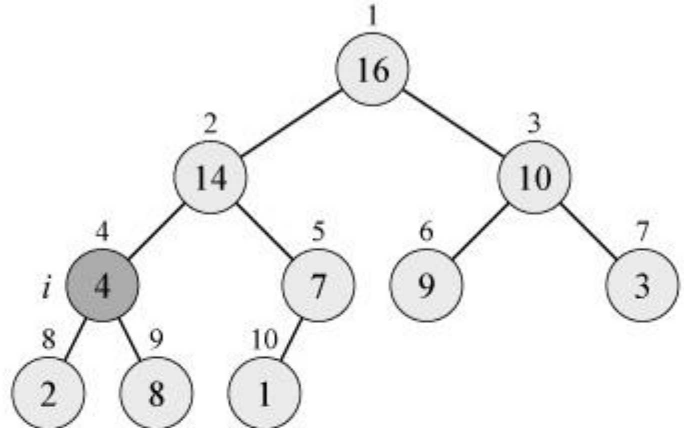
- the binary trees under **`Left(`**$i$**`)`** and **`Right(`**$i$**`)`** are max-heaps
- $A[i]$ might be smaller than **`Left(`**$i$**`)`** or **`Right(`**$i$**`)`**

Goal: Make the binary tree under $i$ a max-heap

# Maintaining the heap property

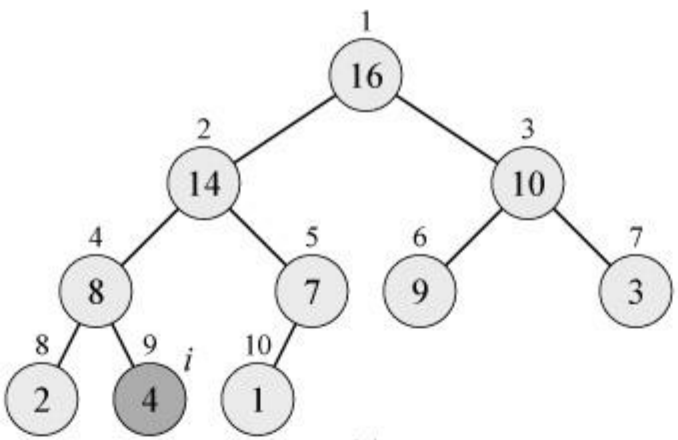



(a)




(b)

Running time: $O(\log n)$

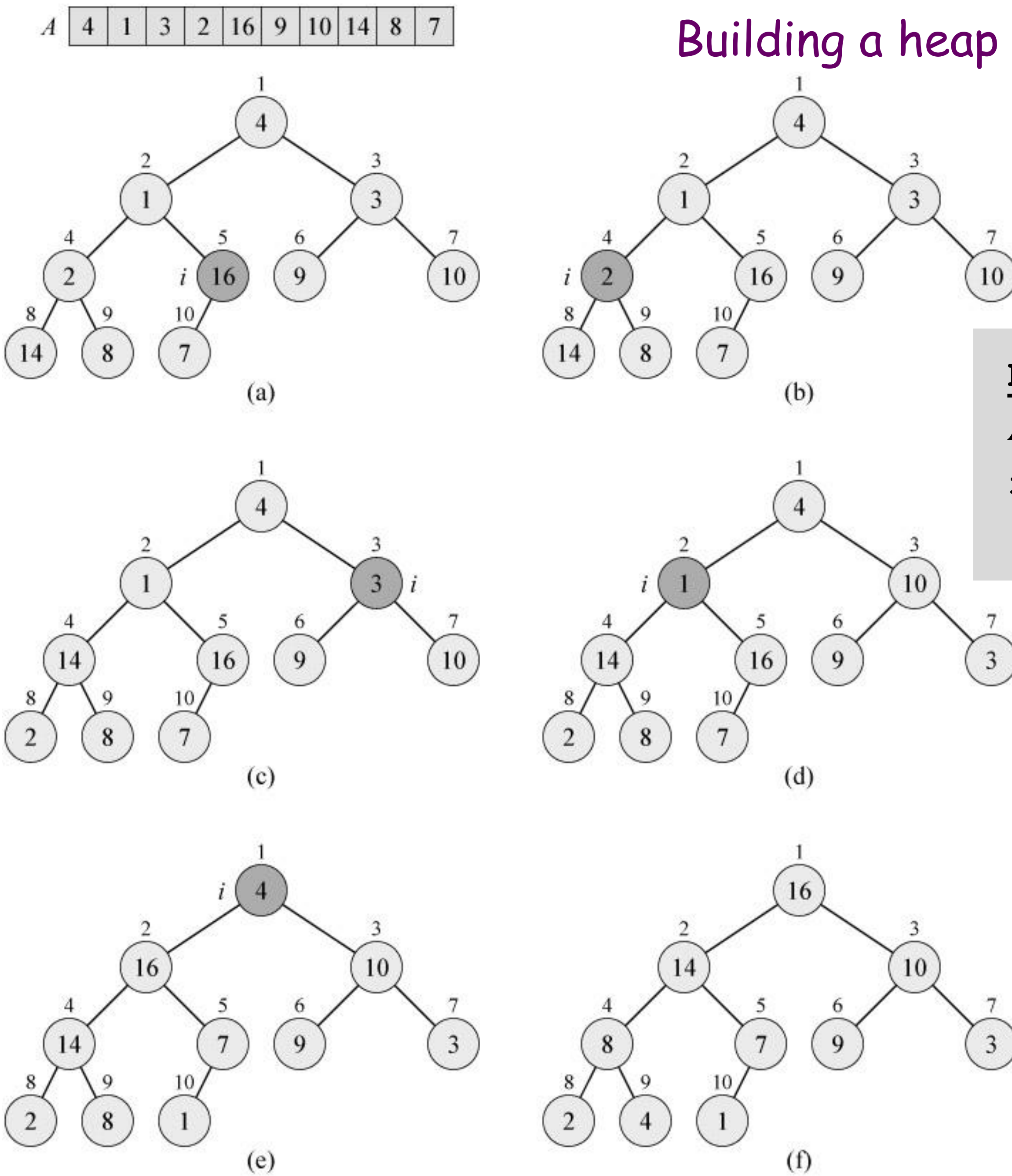



(c)

**Max-Heapify$(A, i)$:**

$l \leftarrow$ **Left(i)**, $r \leftarrow$ **Right**$(i)$
**if** $l \leq A.heapsize$ **and** $A[l] > A[i]$ **then**
  $largest \leftarrow l$
**else** $largest \leftarrow i$
**if** $r \leq A.heapsize$ **and** $A[r] > A[largest]$ **then**
  $largest \leftarrow r$
**if** $largest \neq i$ **then**
  **exchange** $A[i]$ **with** $A[largest]$
  **Max-Heapify**$(A, largest)$

# Building a heap




**Build-Max-Heap($A$):**

$A.heapsize \leftarrow A.length$

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

    **Max-Heapify($A, i$)**

# Analysis of heap-building

| level | # nodes | cost per `Max-Heapify` | total cost |
|---|---|---|---|
| $0$ | $1$ | $\leq \log n$ | $\leq 1 \cdot \log n$ |
| $1$ | $2$ | $\leq \log n - 1$ | $\leq 2 \cdot (\log n - 1)$ |
| $2$ | $4$ | $\leq \log n - 2$ | $\leq 4 \cdot (\log n - 2)$ |
| … | … | … | … |
| $\log n$ | $\leq n$ | $0$ | $0$ |

$O(n \log n)$ is an easy upper bound, but not tight, i.e., can't write $\Theta(n \log n)$

Theorem: It takes $\Theta(n)$ time to build a heap.

Proof: (Textbook method needs calculus. Here we show an elementary method.)

# Analysis of heap-building (continued)

| | | | | | |
|---|---|---|---|---|---|
| $1$ | $1$ | $1$ | … | $1$ | |
| | $2$ | $2$ | … | $2$ | |
| | | $4$ | … | $4$ | |
| | | | … | … | |
| | | | | $n/2$ | |
| $2 \cdot 1$ | $2 \cdot 2$ | $2 \cdot 4$ | … | $2 \cdot n/2$ | $= \Theta(n)$ |

# Heapsort

```
Heapsort(A):
Build-Max-Heap(A)
for i ← A.length downto 2
    exchange A[1] with A[i]
    A.heapsize ← A.heapsize − 1
    MaxHeapify(A, 1)
```

Running time: $O(n \log n)$, dominated by the $n$ **MaxHeapify** operations.

Q: Is it $\Theta(n \log n)$?

A: Yes, but difficult to show. (And we will see a stronger result later.)

Working space: $O(1)$

Q: Can we use a min-heap to implement heapsort?

# Summary of comparison-based sorting algorithms

| | Insertion sort | Merge sort | Quicksort | Heapsort |
|---|---|---|---|---|
| Running time | $\Theta(n^2)$ | $\Theta(n \log n)$ | $\Theta(n \log n)$ | $O(n \log n)$ |
| Working space | $\Theta(1)$ | $\Theta(n)$ | $\Theta(\log n)$ | $\Theta(1)$ |
| Randomized | No | No | Yes | No |
| Cache performance | Good | Good | Good | Bad |
| Parallelization | No | Excellent | Good | No |
| Stable | Yes | Yes | No | No |

Stable: The ordering of equal elements are preserved after sorting.

# Priority queues

A priority queue is a data structure that supports the following operations:

- Maximum($S$) returns the element of $S$ with the largest key.
- Insert($S, x$) inserts the element $x$ into the set $S$.
- Extract-Max($S$) removes and returns the element of $S$ with the largest key.
- Increase-Key($S, i, k$) increases $A[i]$ to the new value $k$.
- Decrease-Key($S, i, k$) decreases $A[i]$ to the new value $k$.

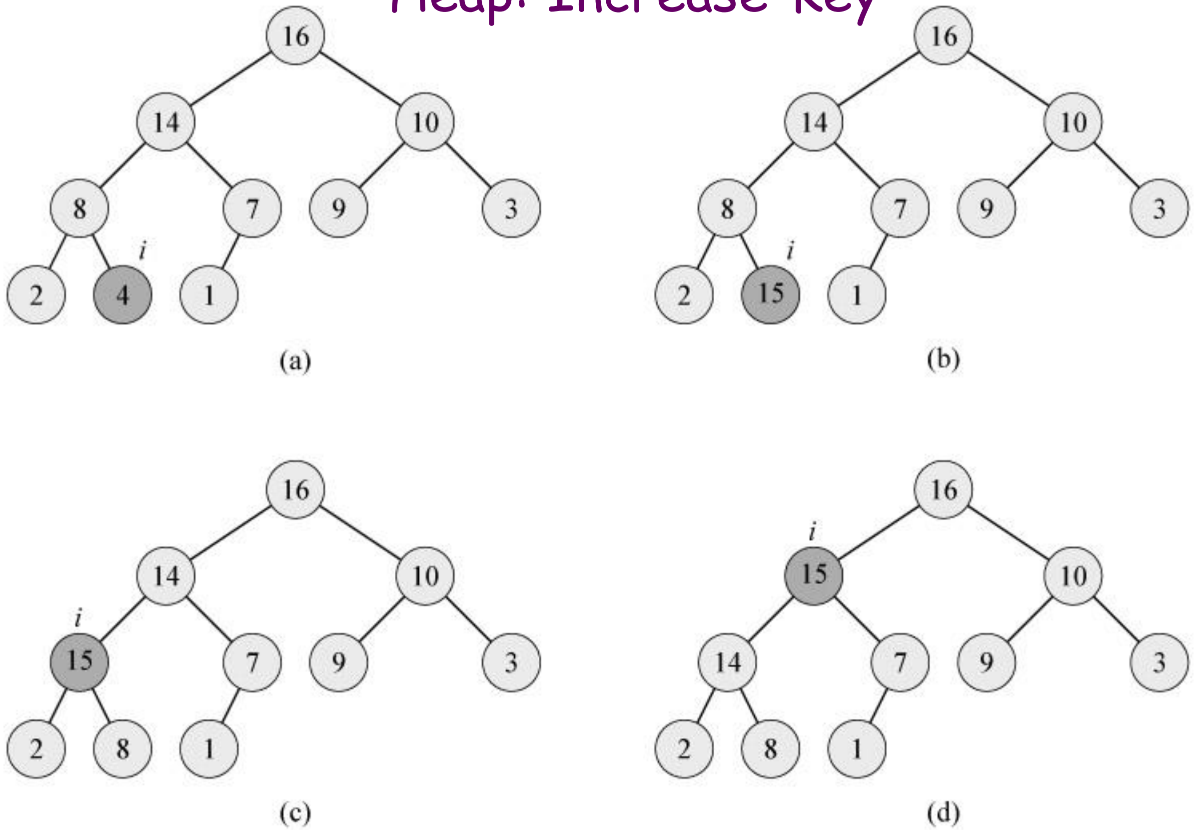
**Heap-Extract-Max($A$):**

**if** $A.heapsize < 1$ **then return error**
$max \leftarrow A[1]$
$A[1] \leftarrow A[A.heapsize]$
$A.heapsize \leftarrow A.heapsize - 1$
**Max-Heapify**($A, 1$)
**return** $max$

**Heap-Decrease-Key($A, i, k$):**

$A[i] \leftarrow k$
**Max-Heapify**($A, i$)

# Heap: Increase-Key




**Heap-Increase-Key** $(A, i, k)$ **:**

$A[i] \leftarrow key$

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

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

    $i \leftarrow \text{Parent}(i)$

Running time: $O(\log n)$

**Max-Heap-Insert** $(A, k)$

$A.heapsize \leftarrow A.heapsize + 1$

$A[A.heapsize] \leftarrow -\infty$

**Heap-Increase-Key** $(A, A.heapsize, k)$