CS217 - Data Structures & Algorithm Analysis (DSAA)

Lecture #4

➢ **HeapSort**

Prof. Pietro S. Oliveto
Department of Computer Science and Engineering
Southern University of Science and Technology (SUSTech)

olivetop@sustech.edu.cn
https://faculty.sustech.edu.cn/olivetop

Reading: Chapter 6

Aims of this lecture

- To introduce the **HeapSort** algorithm.
- To show how a **clever data structure**, a **heap**, can lead to a **fast** and **in place** sorting algorithm
  - In place: *O(1)* additional space.
- To **practice the design and analysis of algorithms**.
- To introduce the **Priority Queue** data structure

Idea behind HeapSort

- Idea:
  - Find the largest element.
  - Move it to the end of the array (put another one in its place).
  - Repeat with remaining elements.
- Like SelectionSort but …
  - SelectionSort compares lots of elements to find the largest.
  - Can we store knowledge gained from these comparisons for the future?
  - Use this knowledge to make future iterations faster!

Use your imagination…

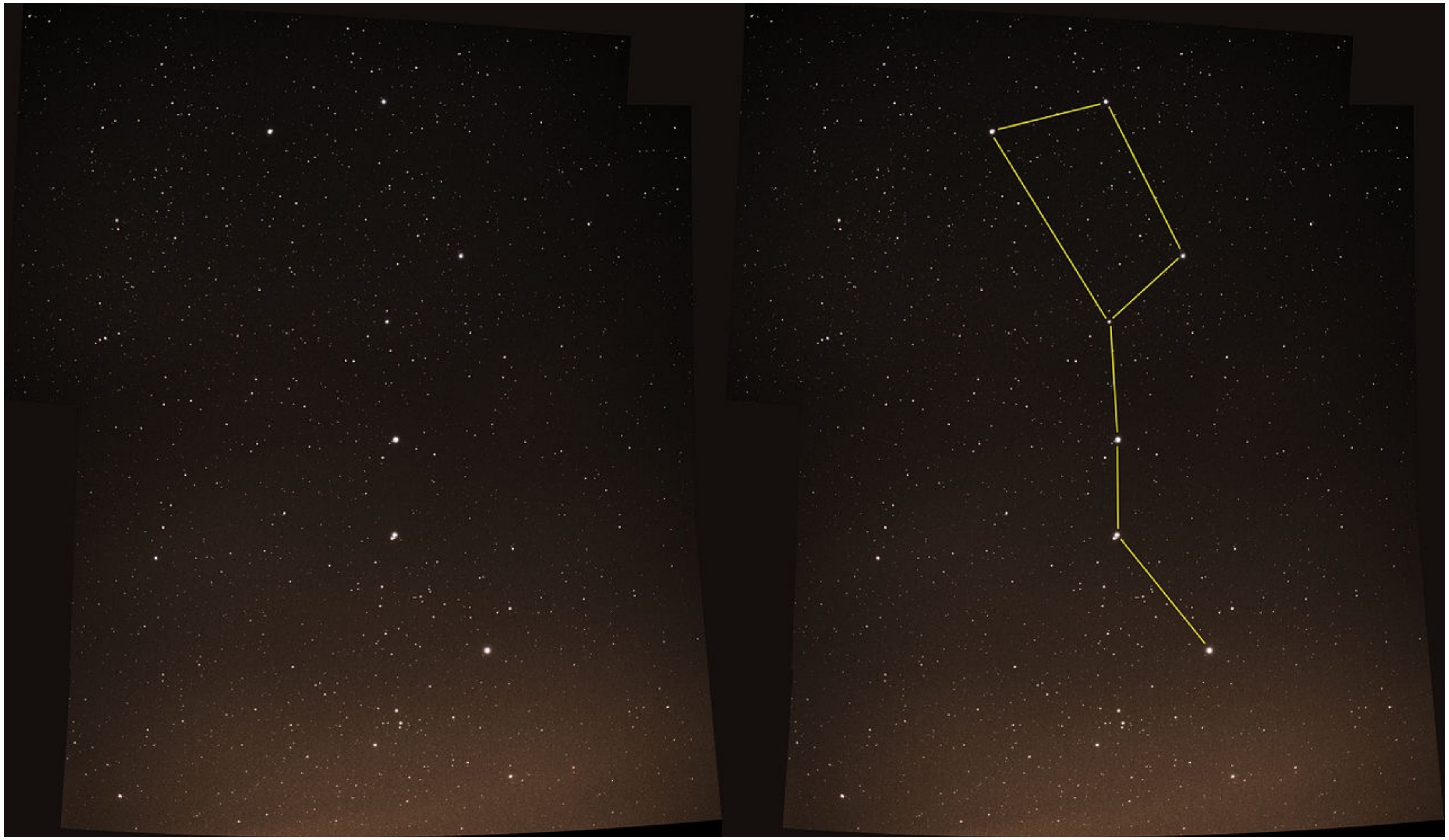
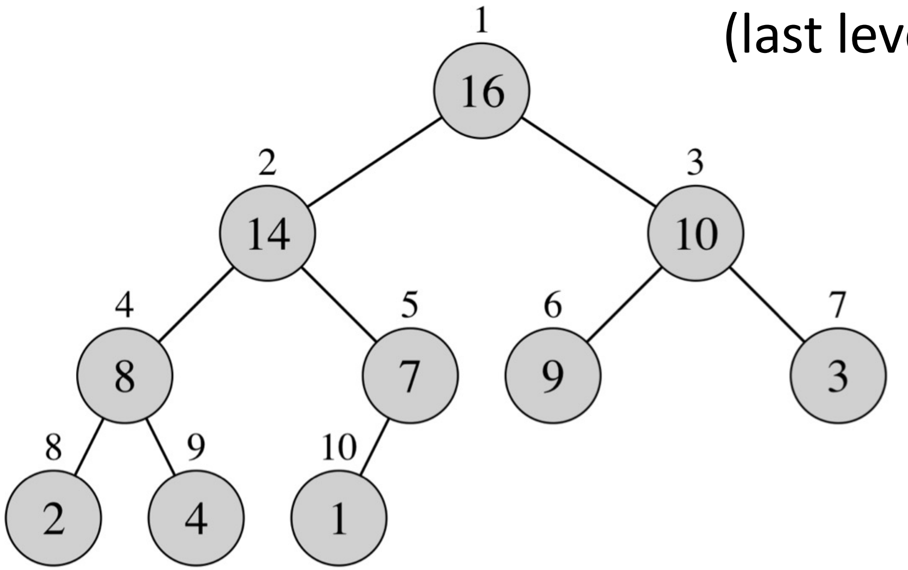


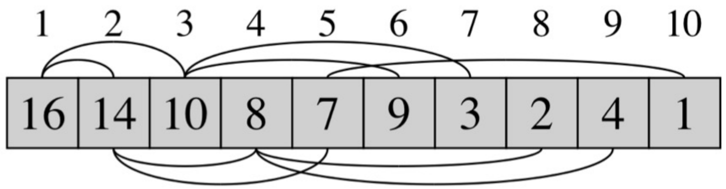
Photo : Thomas Bresson

➢ A Heap

- Essentially an array **imagined** as being a **binary tree**!
- Elements are arranged row by row from left to right.
  (last level may be incomplete)



(a)



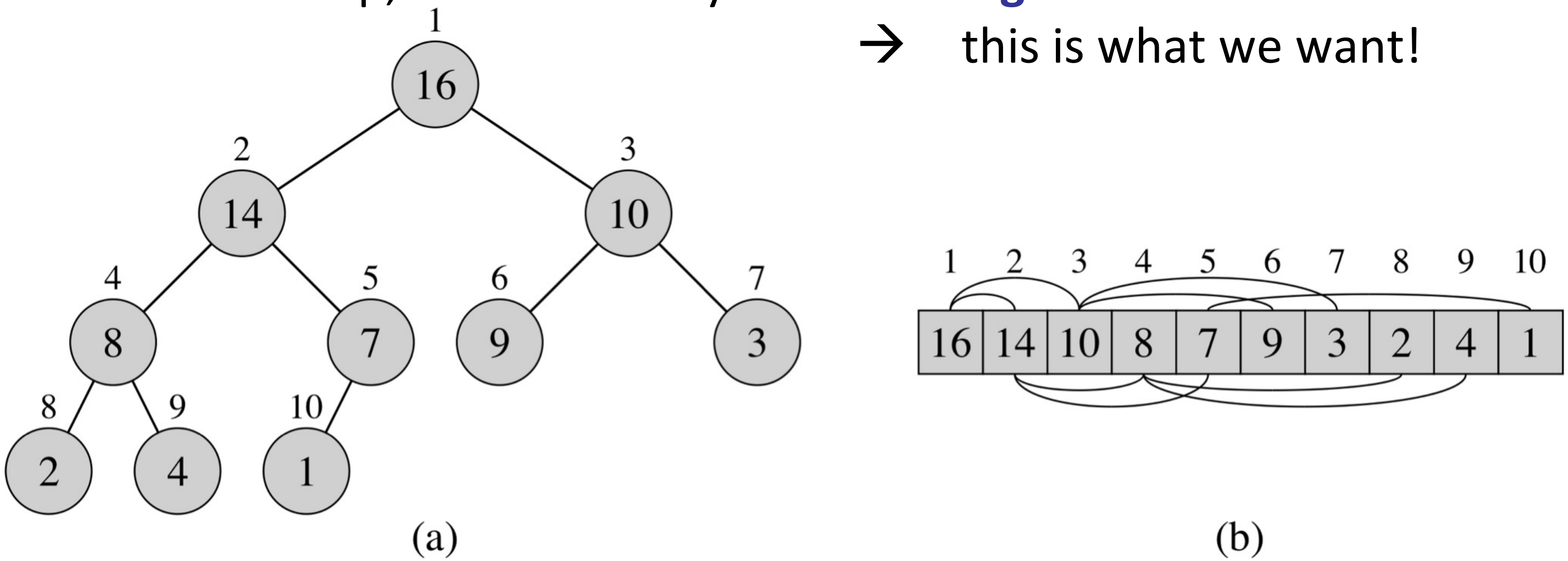
(b)

- Navigate through the array/imaginary tree using these operations:
- Parent$(i) = \left\lfloor \frac{i}{2} \right\rfloor$ ("floor of *i/2*"), Left$(i) = 2i$, Right$(i) = 2i + 1$

➢ Heap Properties

- **Max-heap property**: for every node other than the root, the parent is no smaller than the node, $A[Parent(i)] \geq A[i]$.
- In a max-heap, the **root** always stores a **largest** element.

→ this is what we want!



- **Min-heap property**: for every node other than the root, the parent is no larger than the node, $A[Parent(i)] \leq A[i]$.

➢ Procedures (what do we need)

1. **Build-Max-Heap:** produces a Max-Heap from an unordered array
2. **Max-Heapify**: maintains the max-heap property once the maximum has been removed
3. **HeapSort:** sorts an array in place

- New variable A.heap-size indicates how many elements of A are stored in a heap: $0 \leq$ A.heap-size $\leq$ A.length.
  - Decreasing A.heap-size by 1 effectively removes the last element from the heap (we imagine a heap without it)
- There are analogous operations for min-heaps: Min-Heapify and Build-Min-Heap.

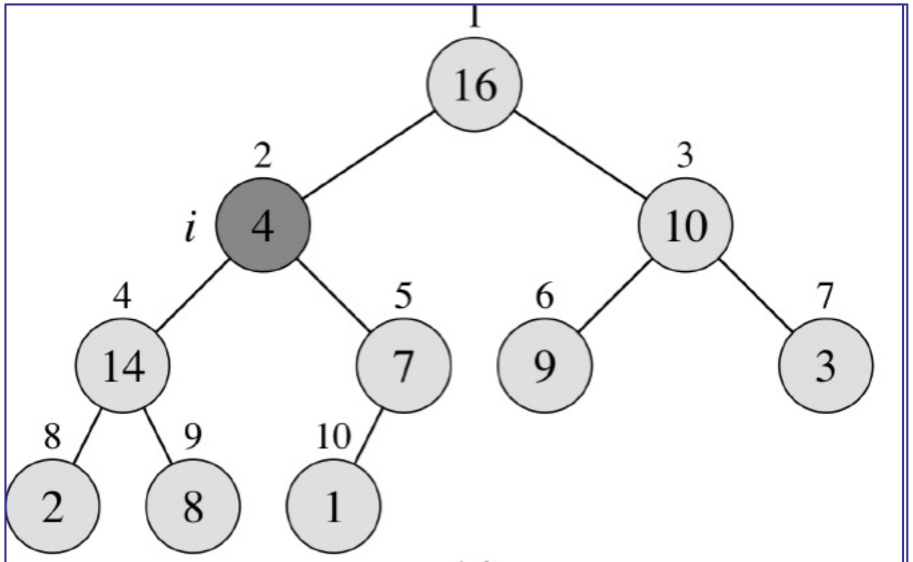
➢ Procedures (what do we need)

1. **Build-Max-Heap:** produces a Max-Heap from an unordered array
2. **Max-Heapify**: maintains the max-heap property once the maximum has been removed
3. **HeapSort:** sorts an array in place

- New variable A.heap-size indicates how many elements of A are stored in a heap: $0 \leq$ A.heap-size $\leq$ A.length.
  - Decreasing A.heap-size by 1 effectively removes the last element from the heap (we imagine a heap without it)
- There are analogous operations for min-heaps: Min-Heapify and Build-Min-Heap.

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

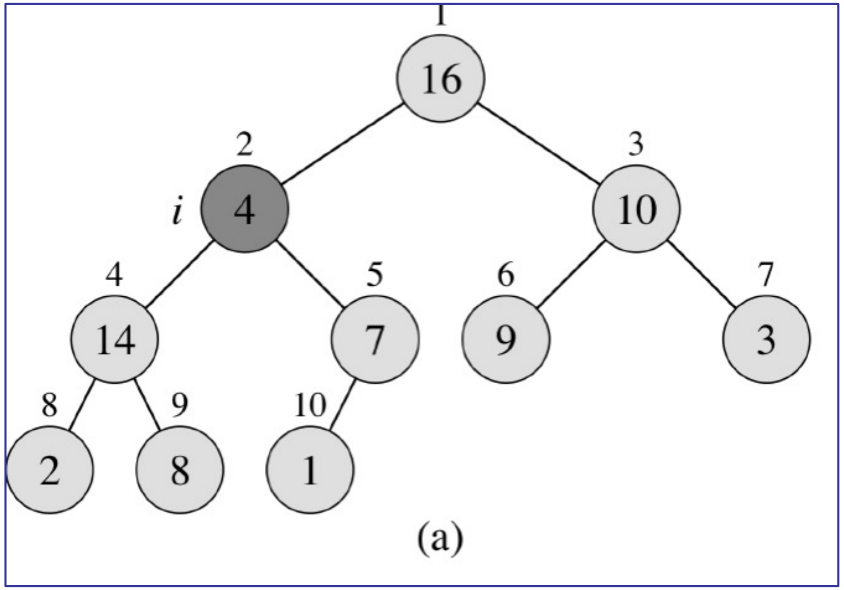
- **Assumes subtrees Left(*i*) and Right(*i*) are max-heaps**, but max-heap property might be violated in root of subtree at *i*.
  - "Subtree x": the part of the tree including x and everything below.
- Lets the value at *A[i]* "float down" if necessary, to restore max-heap property at *i*
- At the end of Max-Heapify the subtree at *i* is a max-heap.



(a)

➤ Max-Heapify: informal and in pseudocode

- Compare *A[i]* with all existing children
- If **largest child** is larger than *A[i]*, swap and recurse on child



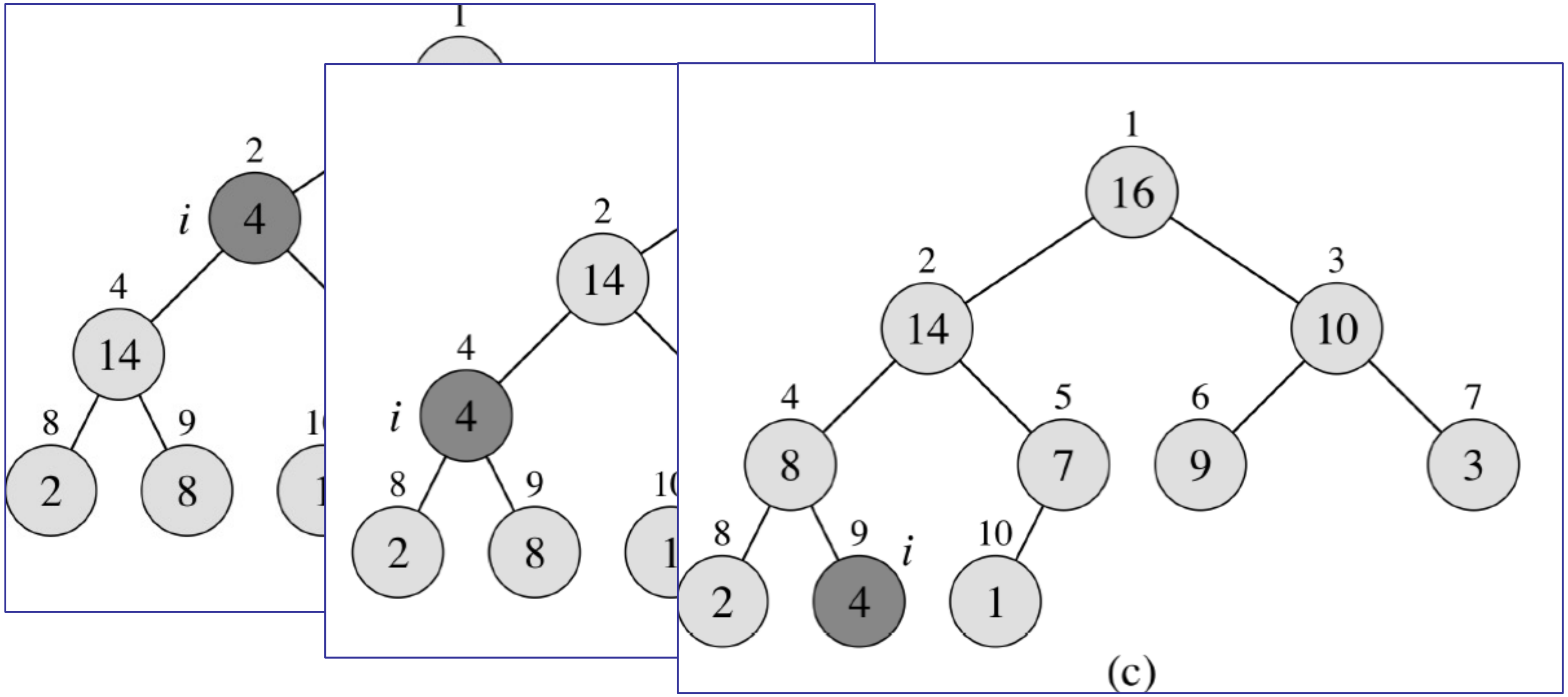
(a)

MAX-HEAPIFY(A, i)

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

Max-Heapify: Example

- Compare *A[i]* with all existing children
- If **largest child** is larger than *A[i]*, swap and recurse on child

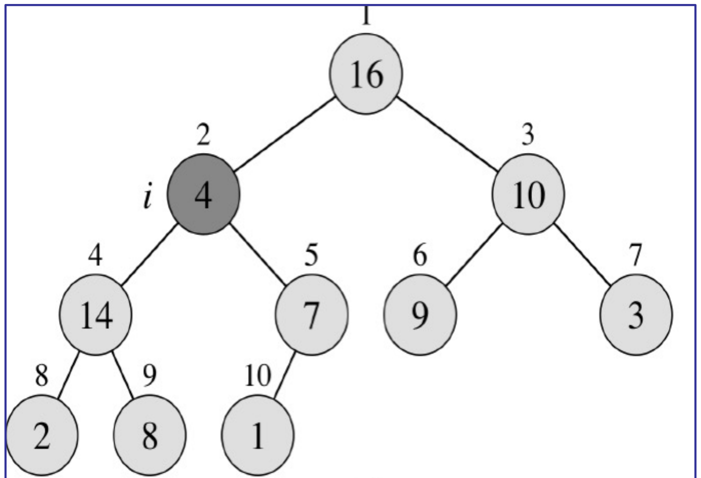


Runtime of Max-Heapify

- Define the height of a node as the longest number of simple downward edges from the node to a **leaf**.
- **Leaf**: a node without children.
- Max-Heapify takes constant time, $\Theta(1)$, on each level.
- Running time of Max-Heapify on a node of height h is $O(h)$.
- It's not $\Omega(h)$ as Max-Heapify may stop early, e.g. if heap-property holds at i.
- For leaves $h = 0$ and the time is $O(1)$.

MAX-HEAPIFY(A, i)

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



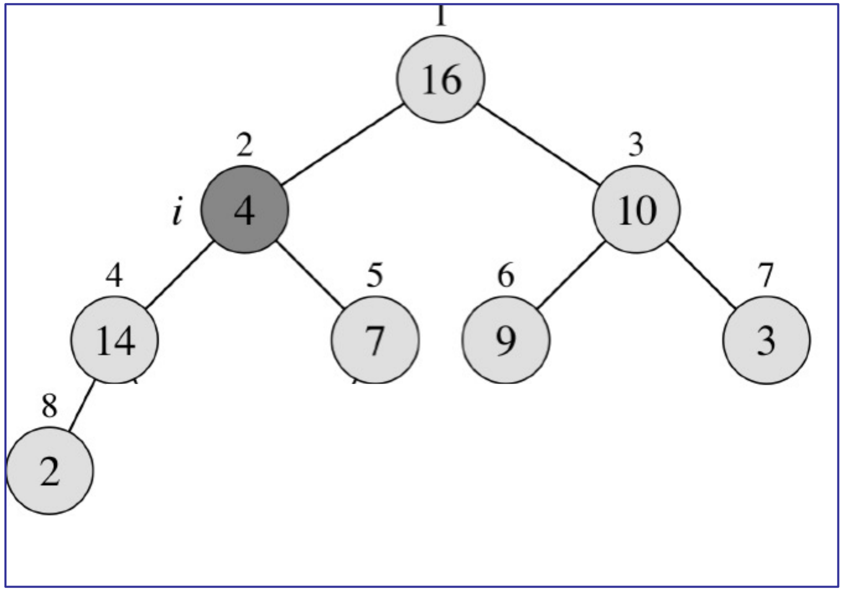
(a)

Bounding the height of a heap

- **Claim**: the **height of a heap** = height of the root is at most *log n*.
- **Proof**: the number *n* of elements in a heap of height *h* is
  - Doubling on each level
  - At least 1 node on the last level
  - Hence in total at least

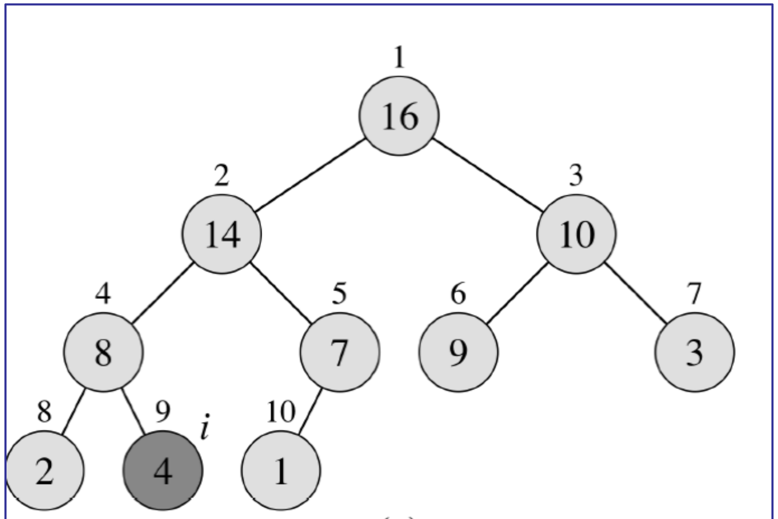
$$1 + 2 + 4 + \cdots + 2^{h-1} + 1 = 2^h$$

(we used $\sum_{i=0}^{k-1} 2^i = 2^k - 1$)



- So size and height are related as $n \geq 2^h \Leftrightarrow \log n \geq h$
- So the runtime of Max-Heapify is O(log n)

Max-Heapify: Correctness

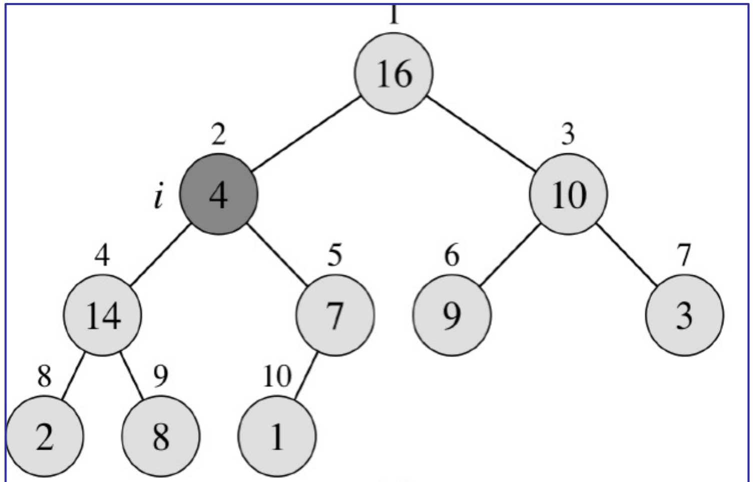


(c)

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

- By induction (on the height):
- **Base case:** height = 0 (*i* is a leaf)
- Then left(i) and right(i) are larger than A.heap-size and the algorithm returns a heap!

Max-Heapify: Correctness



(a)

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

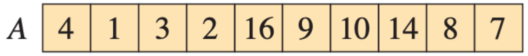
- By induction (on the height):
- **Inductive case:** assume it works for height h=*i-1* and show it works for h=i
- Then the algorithm swaps A[i] with the larger between Left(i) and Right(i) (if any) and one subtree was already a heap and the other will be by inductive hypothesis.

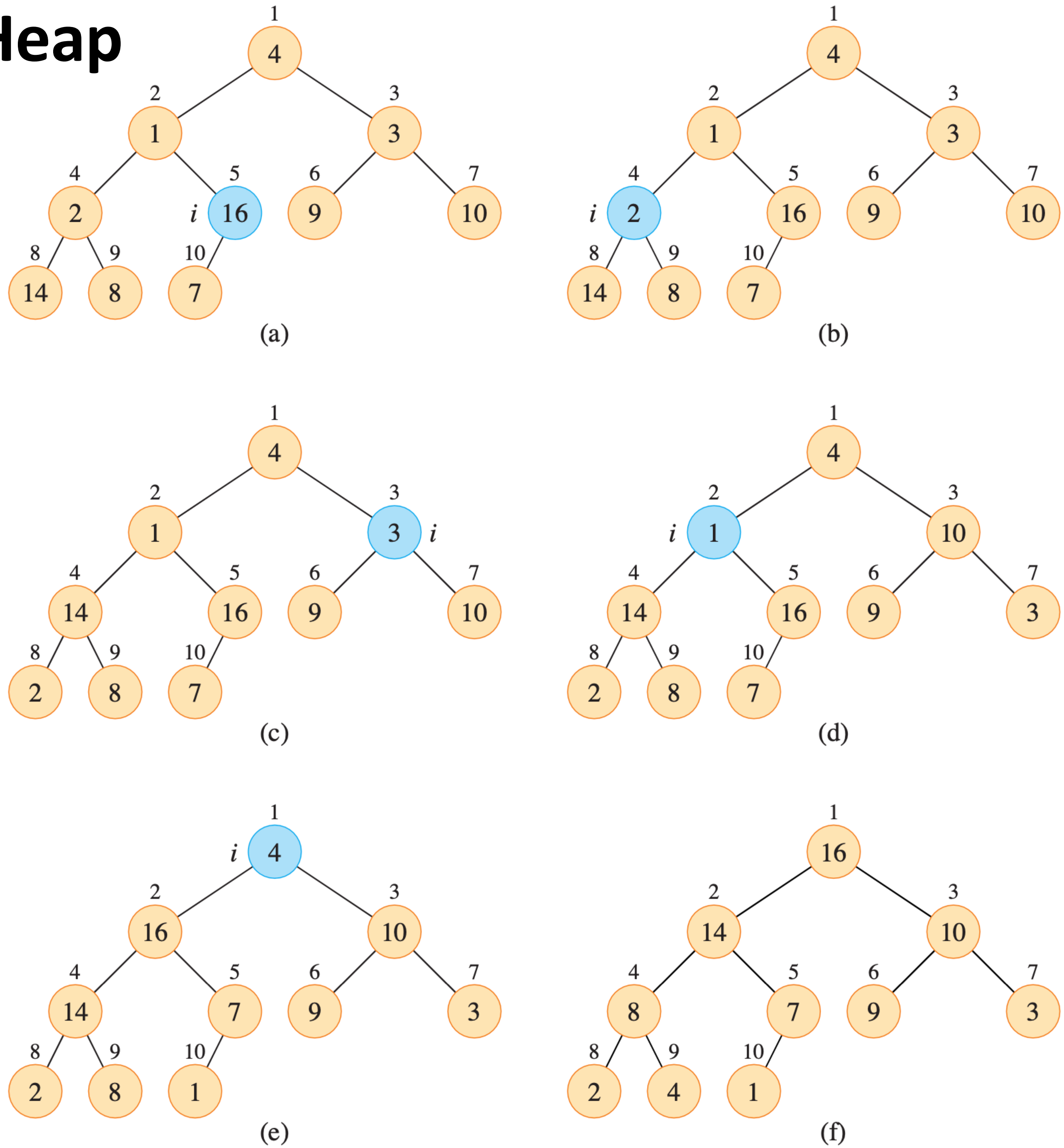
Building a Heap

- Idea: use Max-Heapify repeatedly to create a heap.
- Which order of nodes: top-down or bottom-up?
- Answer: **bottom-up** – Max-Heapify assumes Left(*i*) and Right(*i*) are heaps. Top-down wouldn't work, bottom-up does.
- Note: nodes in $A\left[\left(\left\lfloor\frac{n}{2}\right\rfloor + 1\right), \ldots, n\right]$ are all leaves. Leaves are max-heaps, so no work required.

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

➢ Build-Max-Heap

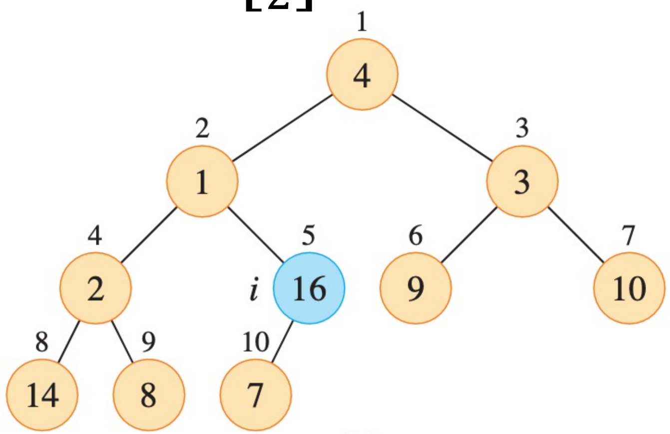




Correctness of Build-Max-Heap

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

- **Loop invariant:** At the start of each iteration of the for loop, each node $i+1, i+2, \ldots, n$ is the root of a max-heap.
- **Initialisation:** true for leaves $\left\lfloor \frac{n}{2} \right\rfloor + 1, \ldots, n$.

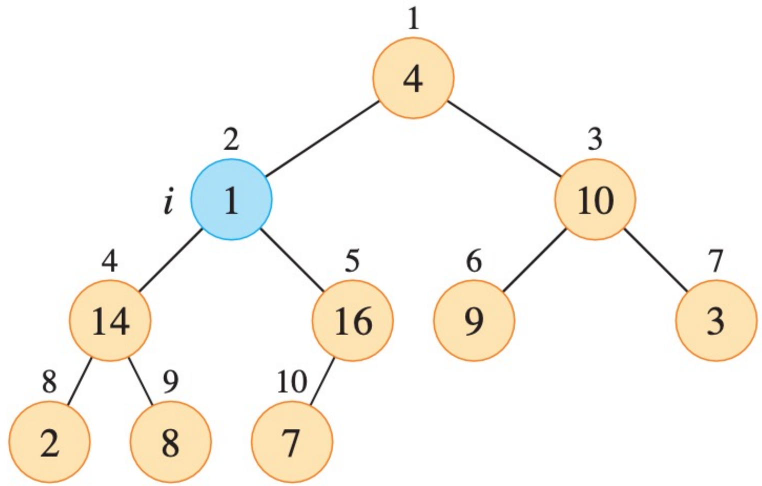


(a)

Correctness of Build-Max-Heap

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

- **Loop invariant:** At the start of each iteration of the for loop, each node $i+1, i+2, \ldots, n$ is the root of a max-heap.
- **Maintenance:** by loop invariant, all children of i are roots of max-heaps (as their numbers are larger than i).
  Then Max-Heapify(A, i) turns the subtree at i into a max-heap.

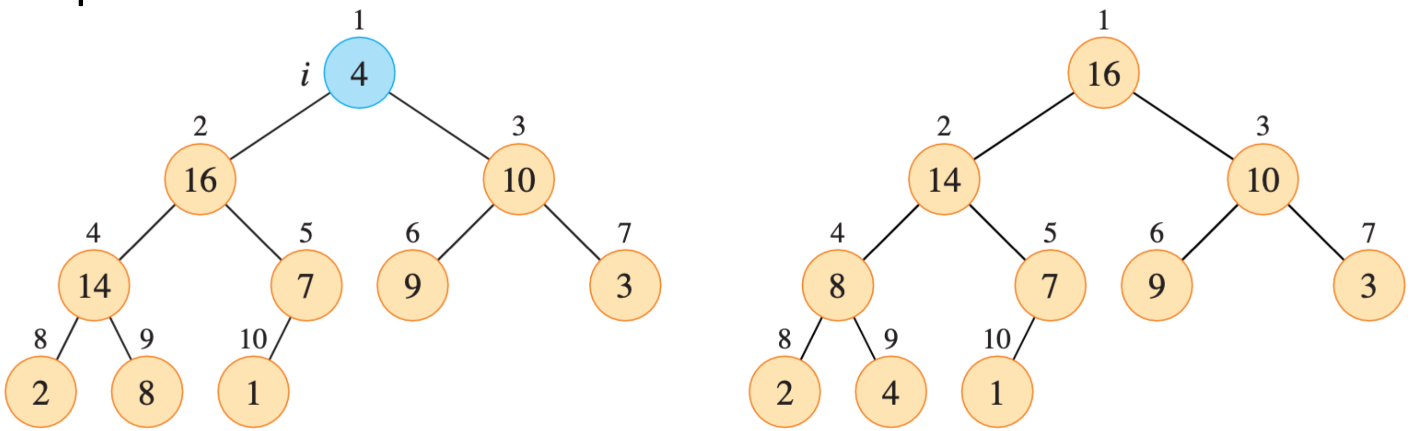


(d)

Correctness of Build-Max-Heap

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

- **Loop invariant:** At the start of each iteration of the for loop, each node $i+1, i+2, \ldots, n$ is the root of a max-heap.
- **Termination:** the loop terminates at $i = 0$, hence node 1 is the root of a max-heap.



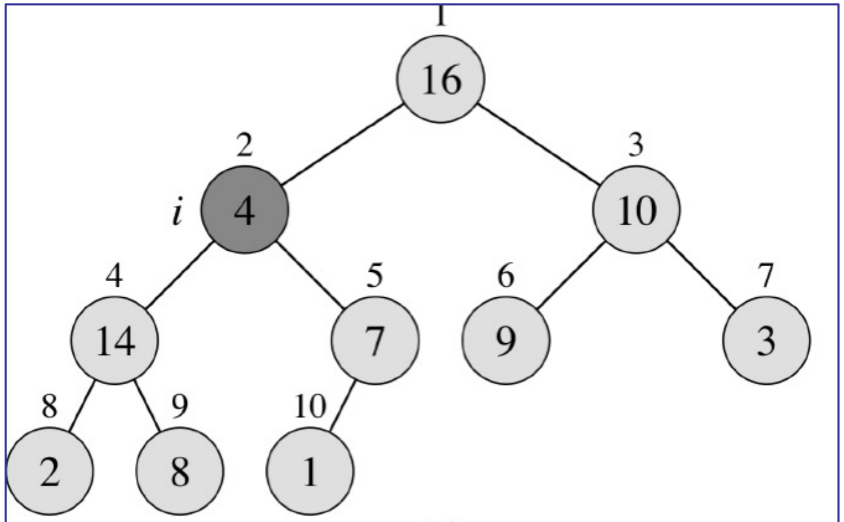
(e) (f)

Runtime of Build-Max-Heap

- The **height of a heap** = height of the root is at most *log n*.
- So all nodes have height at most *log n*.
- Every call to Max-Heapify takes time $O(\log n)$.
- Build-Max-Heap calls Max-Heapify $O(n)$ times.
- Total time is at most $O(n) \cdot O(\log n) = O(n \log n)$.
  - The time can be improved to *O(n)* since most nodes have small height.
  - $O(n \log n)$ is sufficient for us, though.

Refined Analysis of Build-Max-Heap

- **Observation: most nodes have small height!**
- One can show: there are at most $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nodes of height h.
- $O(\log n)$ time bound is correct, but crude for most nodes.
- A better bound:

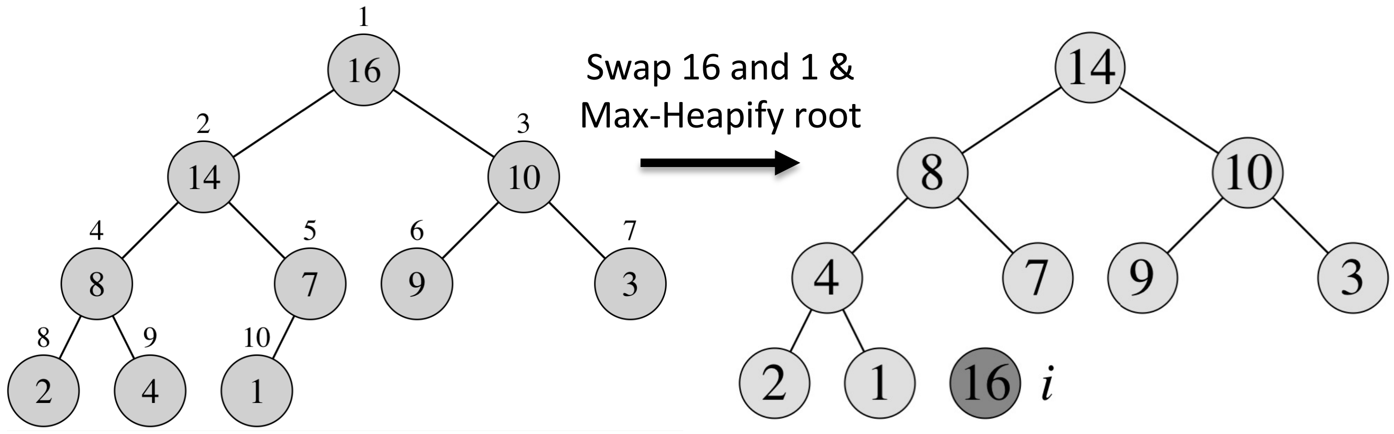


(a)

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

as the infinite series of $\frac{h}{2^h}$ is 2.

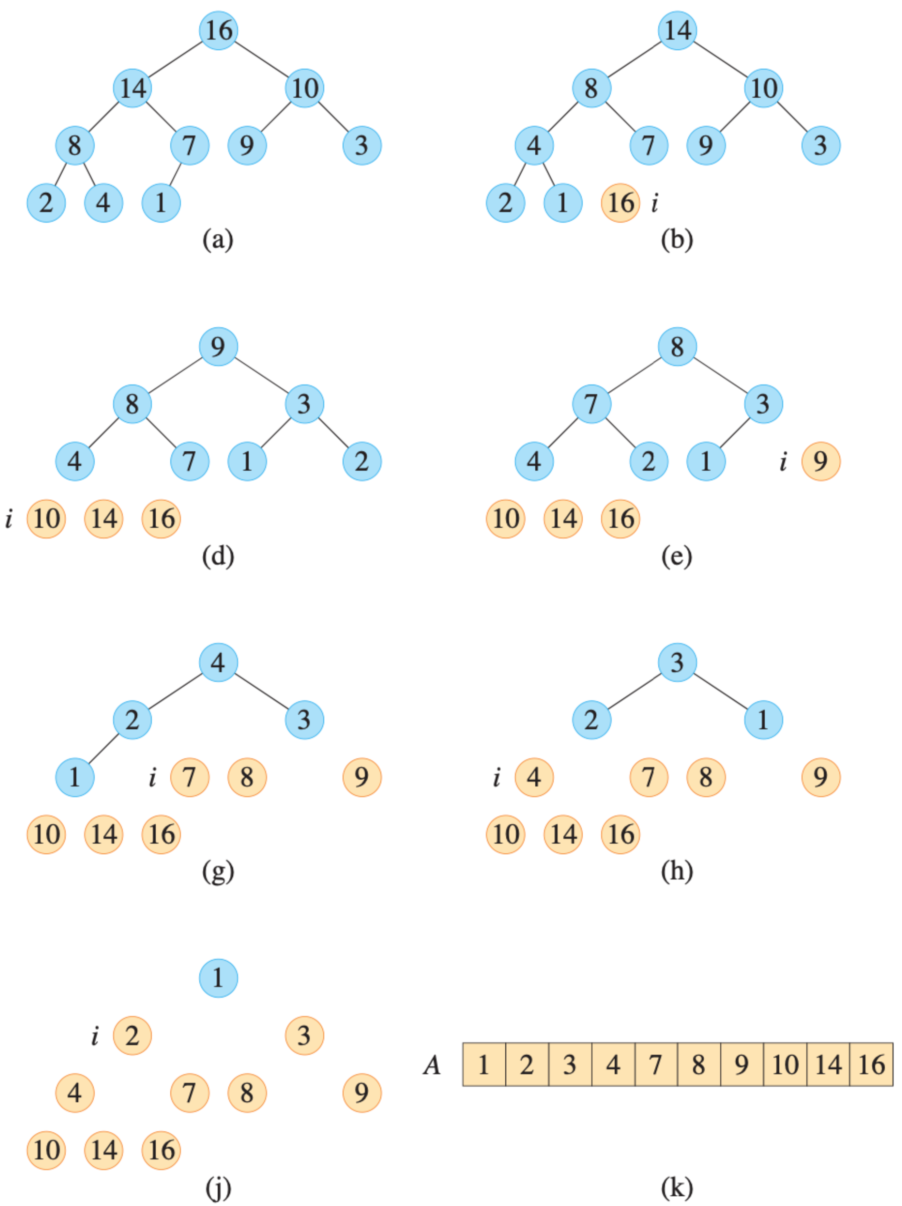
➢ HeapSort



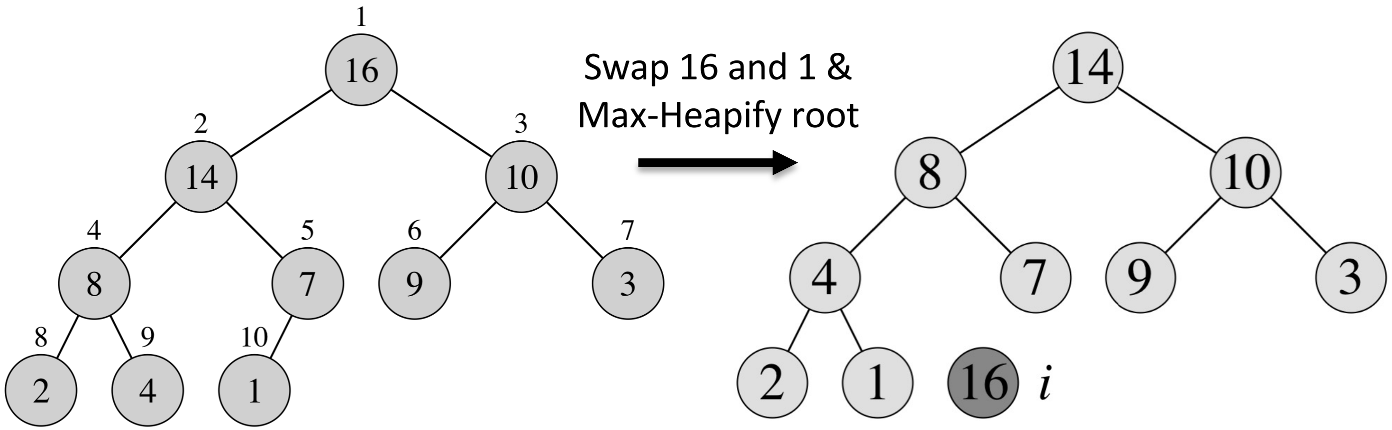
- Ideas:
  1. Build a max-heap, such that the root contains largest element.
  2. Swap the root with the last element of the heap/array.
  3. Discard the last element from the heap by reducing heap.size. (We simply imagine a smaller heap.)
  4. Call Max-Heapify(*A, 1*) to restore heap property at the root.

```
HeapSort(A)
Build-Max-Heap(A)
for i = A.length downto 2 do
    exchange A[1] with A[i]
    A.heap-size = A.heap-size −1
    Max-Heapify(A, 1)
```

➢ HeapSort: Example



HeapSort



- Ideas:
  1. Build a max-heap, such that the root contains largest element.
  2. Swap the root with the last element of the heap/array.
  3. Discard the last element from the heap by reducing heap.size. (We simply imagine a smaller heap.)
  4. Call Max-Heapify(*A, 1*) to restore heap property at the root.

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

Runtime:

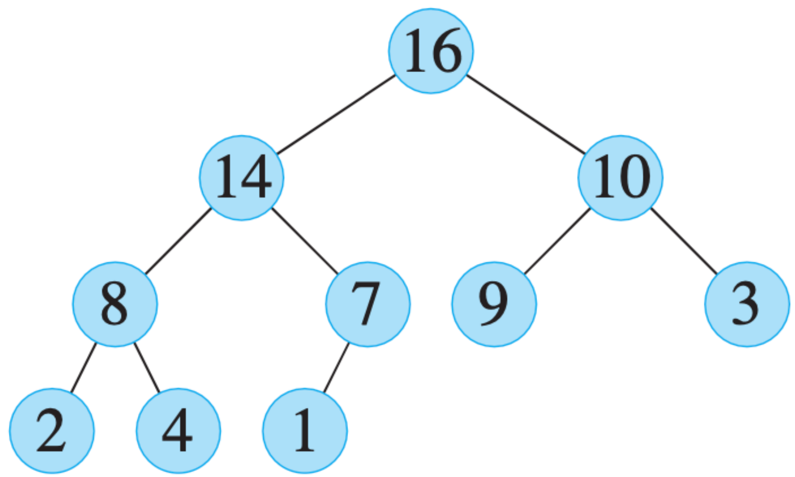
$$O(n \log n) + (n-1) \cdot O(\log n) = O(n \log n)$$

Correctness of HeapSort

Loop Invariant: "At the start of each iteration of the for loop of lines 2-5, the subarray $A[1..i]$ is a max-heap containing the i smallest elements of $A[1..n]$, and the subarray $A[i+1..n]$ contains the $n-i$ largest elements of $A[1..n]$, sorted."

- **Initialization:** The subarray $A[i+1..n]$ is empty, thus the invariant holds.

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



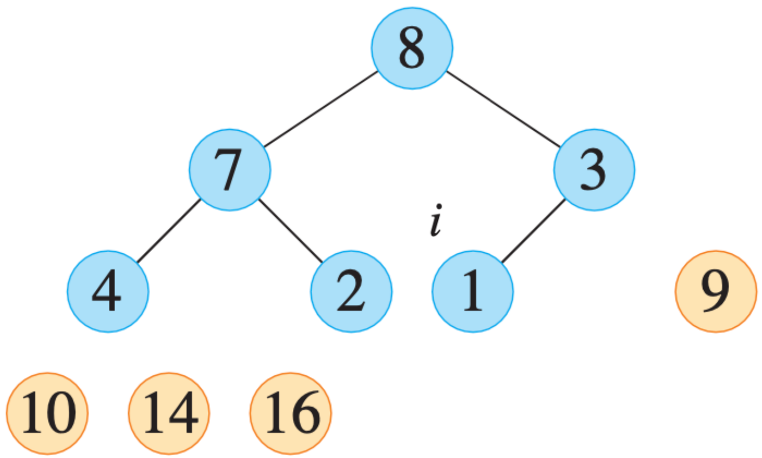
(a)

Correctness of HeapSort

Loop Invariant: “At the start of each iteration of the for loop of lines 2-5, the subarray $A[1..i]$ is a max-heap containing the i smallest elements of $A[1..n]$, and the subarray $A[i+1..n]$ contains the $n-i$ largest elements of $A[1..n]$, sorted.”

Maintenance: $A[1]$ is the largest element in $A[1..i]$ and it is smaller than the elements in $A[i+1..n]$. When we put it in the ith position, then $A[i..n]$ contains the largest elements, sorted. Decreasing the heap size and calling Max-Heapify turns $A[1..i-1]$ into a max-heap. Decrementing i sets up the invariant for the next iteration.

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



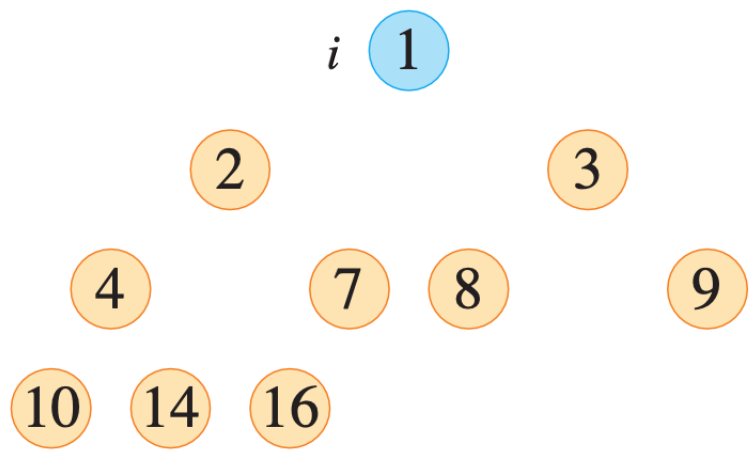
(e)

Correctness of HeapSort

Loop Invariant: “At the start of each iteration of the for loop of lines 2-5, the subarray *A*[1..*i*] is a max-heap containing the *i* smallest elements of *A*[1..*n*], and the subarray *A*[*i+1*..*n*] contains the *n−i* largest elements of *A*[1..*n*], sorted.”

- **Termination:** After the loop *i*=1. This means that *A*[2..*n*] is sorted and *A*[1] is the smallest element in the array, which makes the array sorted.

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



(j)

Priority Queues: Motivation

- Schedule jobs on a computer shared among multiple users
- A max-priority queue keeps track of the jobs to be performed and their relative priorities
- When a job is finished the scheduler selects the job with highest priority from those pending
- Jobs can be added to the scheduler at any time

Job	Owner	Priority (key)
Job 1	Yao Xin	35
Job 12	Oliveto Pietro	2
Job 24	Hao Qi	22
Job 25	Yu Shiqi	18
Job 72	Yao Xin	30

- **Use a heap!**

➢ Priority Queue based on max-heap

- A data structure for maintaining a set S of elements with an associated element called key (the priority).

Operation	Time
Insert(S, x, k) – inserts x with key k into S	$O(\log n)$
Maximum (S) – returns the element in S with the largest key	$O(1)$
Extract-Max(S) – removes and returns element in S with the largest key	$O(\log n)$
Increase-Key(S, x, k) – increases they key of x to a larger value k (element may float up in the heap)	$O(\log n)$

Job x: x.satellite_data; x.job_address x.priority (key)

(We need a way to map the position of job x in the heap (and update it as it moves in the heap) as well as the pointer to the job to execute it)

Min-priority queue based on min-heap also exist: we will use them in graph algorithms (eg., Djikstra, Prim)

Find and extract next job

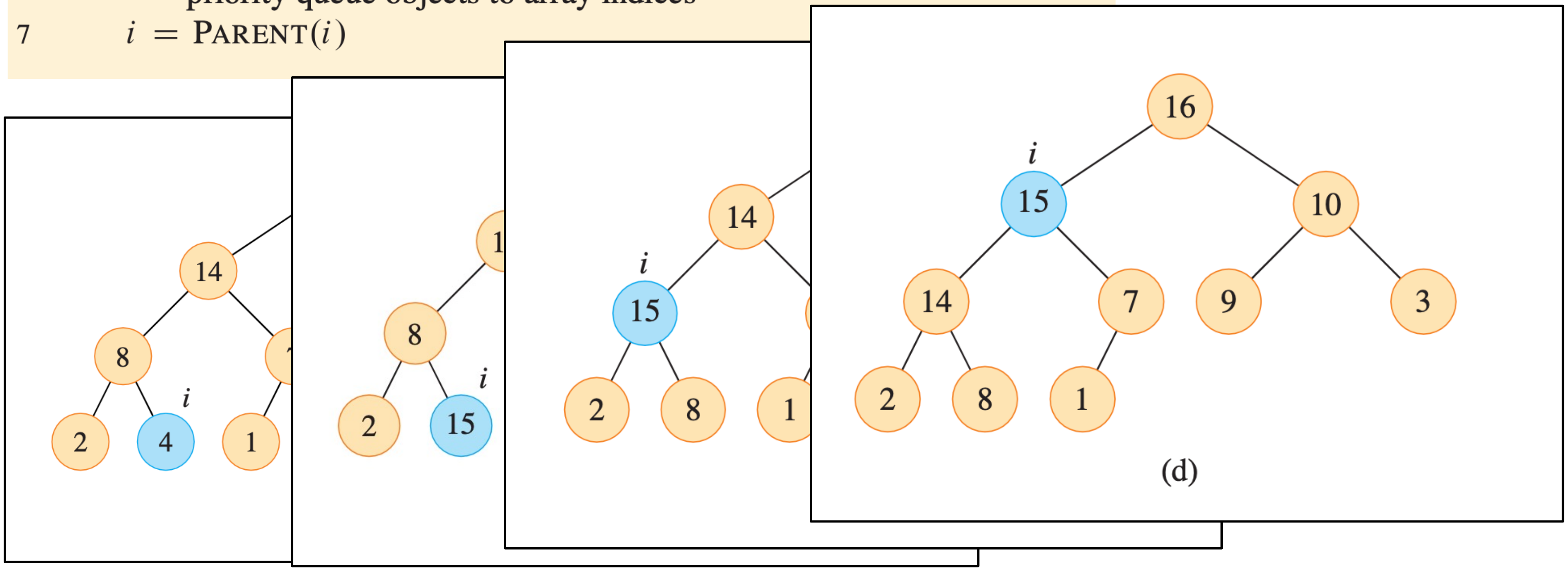
```
MAX-HEAP-MAXIMUM(A)
if A.heap-size < 1
    error "heap underflow"
return A[1]

MAX-HEAP-EXTRACT-MAX(A)
max = MAX-HEAP-MAXIMUM(A)
A[1] = A[A.heap-size]
A.heap-size = A.heap-size - 1
MAX-HEAPIFY(A, 1)
return max
```

Increase job priority

MAX-HEAP-INCREASE-KEY(A, x, k)

if $k < x.key$
    error "new key is smaller than current key"
$x.key = k$
find the index i in array A where object x occurs
while $i > 1$ and $A[\text{PARENT}(i)].key < A[i].key$
    exchange $A[i]$ with $A[\text{PARENT}(i)]$, updating the information that maps priority queue objects to array indices
    $i = \text{PARENT}(i)$



Insert new job

```
MAX-HEAP-INSERT(A, x, n)
if A.heap-size == n
    error "heap overflow"
A.heap-size = A.heap-size + 1
k = x.key
x.key = −∞
A[A.heap-size] = x
map x to index heap-size in the array
MAX-HEAP-INCREASE-KEY(A, x, k)
```

Summary

- HeapSort sorts in place in time $O(n \log n)$.
  - Building a Heap in time $O(n)$.
  - Extracting the largest element and restoring the heap-property in total time $O(n \log n)$.
- The use of appropriate **data structures** can speed up computation (in contrast to SelectionSort).
  - The heap "memorises" information about comparisons of elements.
  - The heap is imaginary, no objects/pointers required!
- Heaps also play a role in **Priority Queues**.