Heap sort - Introduction

Data Structure

- A **data structure** is a way to store and organize data in order to facilitate access and modifications
- No single data structure works well for all purposes
- It is important to know the strengths and limitations of each data structure
- To make the algorithm efficient - choose an appropriate data structure

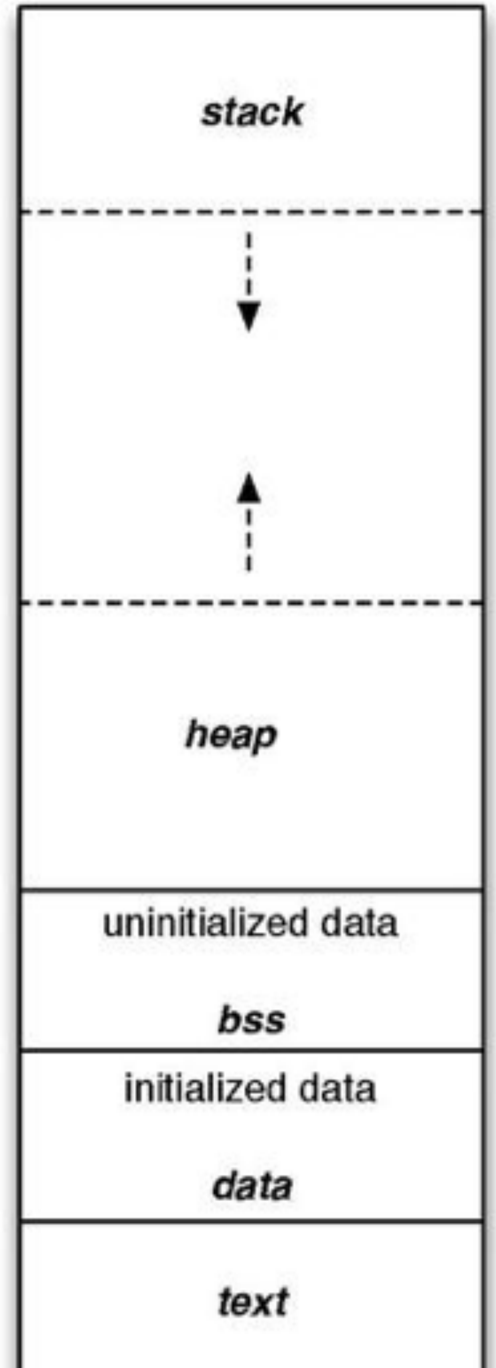
Data Structures

- Array
- Linked list
- Stack
- Queue
- Binary Tree
- Binary Search Tree

Heap

Program Memory Layout

- Heap is **a data structure**
- **Heap sort algorithm** – Use Heap to manage information
- Used to implement **efficient priority queue**
- Not a garbage collected storage



Binary heap

- Binary heap data structure is an **array object**

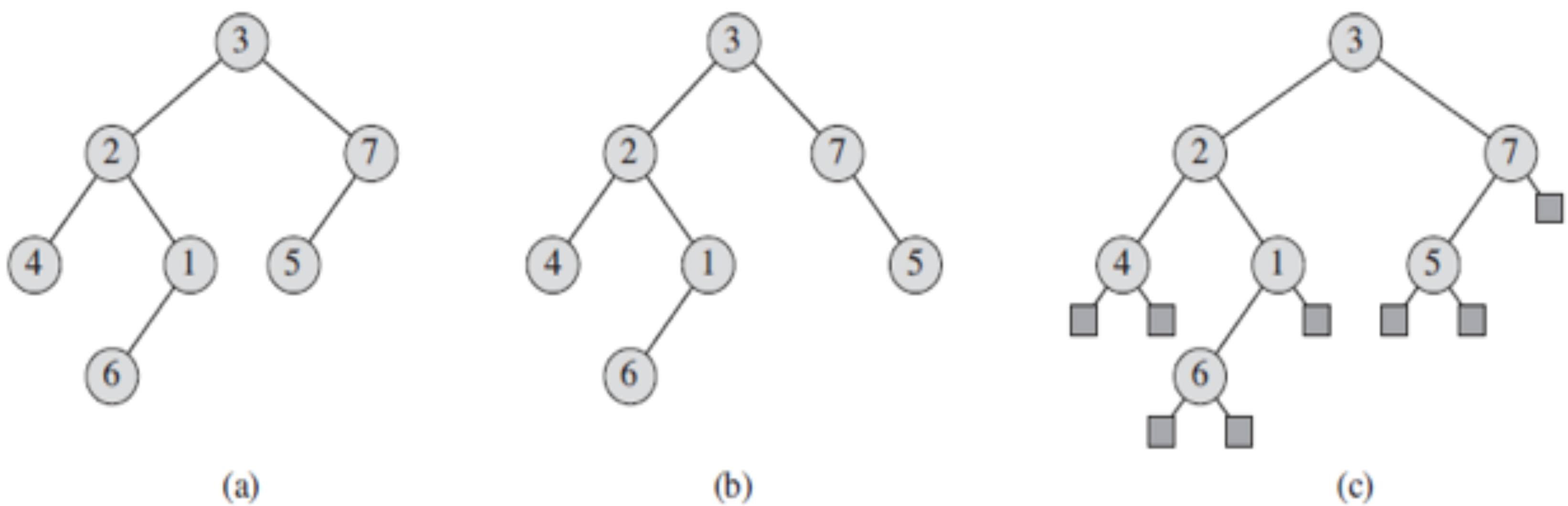
- Viewed as a **nearly complete binary tree**

- **Binary tree:** A binary tree is defined recursively.

- A binary tree T is a structure defined on finite set of nodes that either
  - Contains no nodes (**the *empty tree*** or ***null tree)*** denoted NIL or
  - Composed of three disjoint set of nodes:
    - a **root node**
    - a binary tree called its **left subtree**
    - a binary tree called its **right subtree**

Binary tree - examples

In the following example,
root node:
Node 3
left subtree:
Nodes 2,4,1 and 6 together form Left subtree
right subtree:
Nodes 7 and 5 together form Right subtree



Few terms….

- The **number of children** of a node x in a rooted tree T equals the ***degree* of x**

Example:

- **Degree of node 3:**

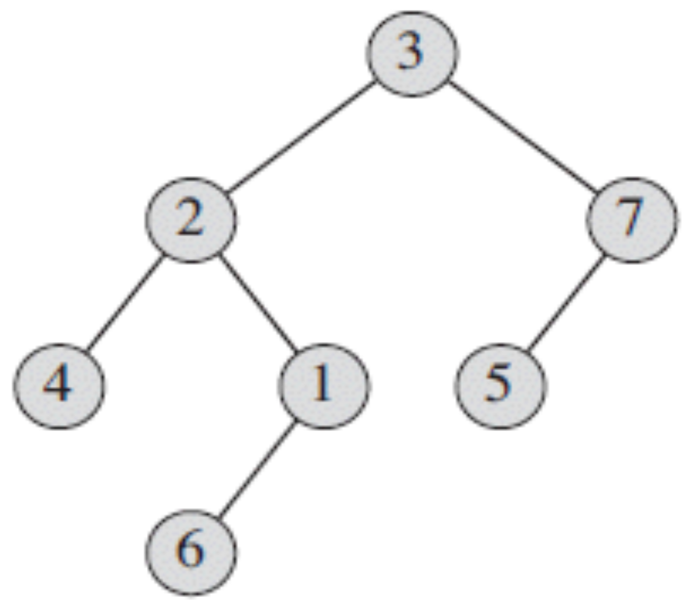
2

- **Degree of node 7:**

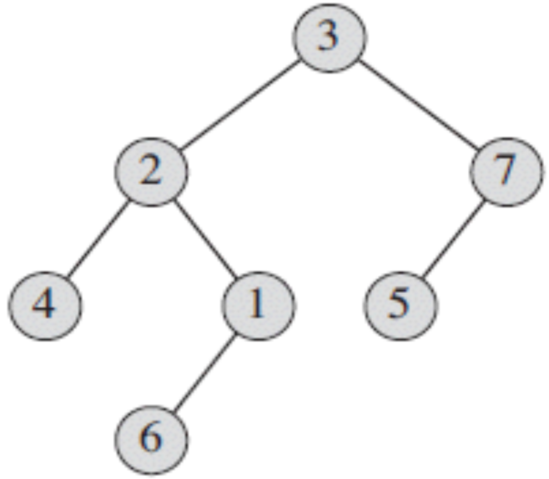
1

- **Degree of node 6:**

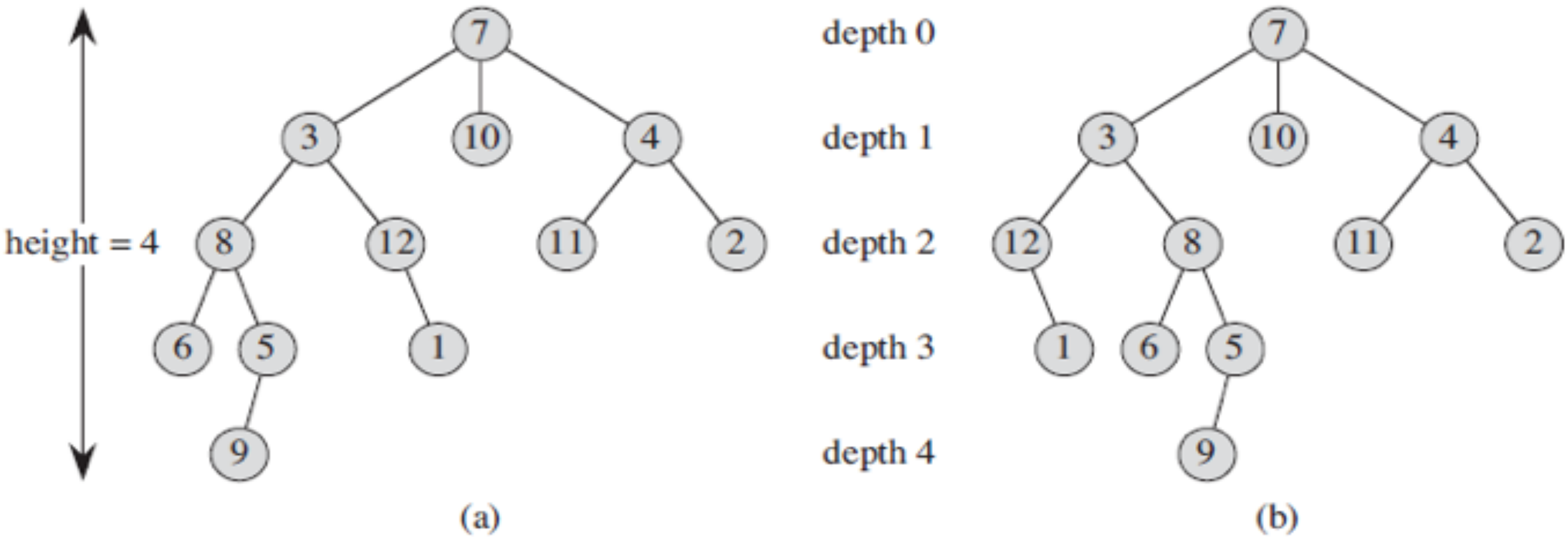
0



- The **length of the simple path** from the **root r** to a **node x** is the ***depth* of x** in T .
  - Eg: Depth of node 6:
  3
  - Depth of nodes 1, 4, 5:
  2
  - Depth of nodes 2 and 7:
  1
  - Depth of node 3:
  0



- A ***level* of a tree** consists of all nodes at the same depth.
  - **Nodes at level 2**:
    4, 1 and 5
- **Ex:** What are the nodes at level 0,1 and 3 in the above tree?



Few terms….

- A node with no children is a ***leaf*** or ***external node***. A non-leaf node is an ***internal node***.
- The ***height*** **of a node** in a tree is the **number of edges on the longest simple downward path from the node to a leaf**
- **Height of a tree** is the height of its **root.**

Complete binary tree

- A ***complete binary tree*** is a binary tree in which all **leaves** have the **same depth** and all internal nodes have **degree 2**.

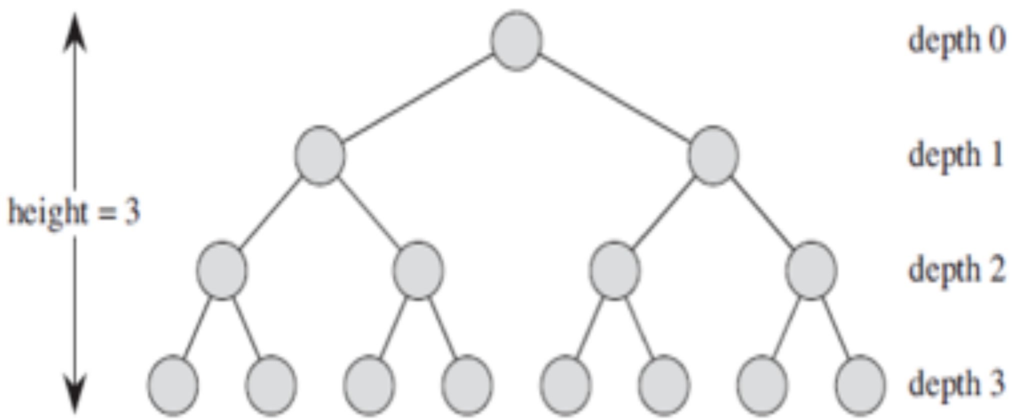


Figure B.8 A complete binary tree of height 3 with 8 leaves and 7 internal nodes.

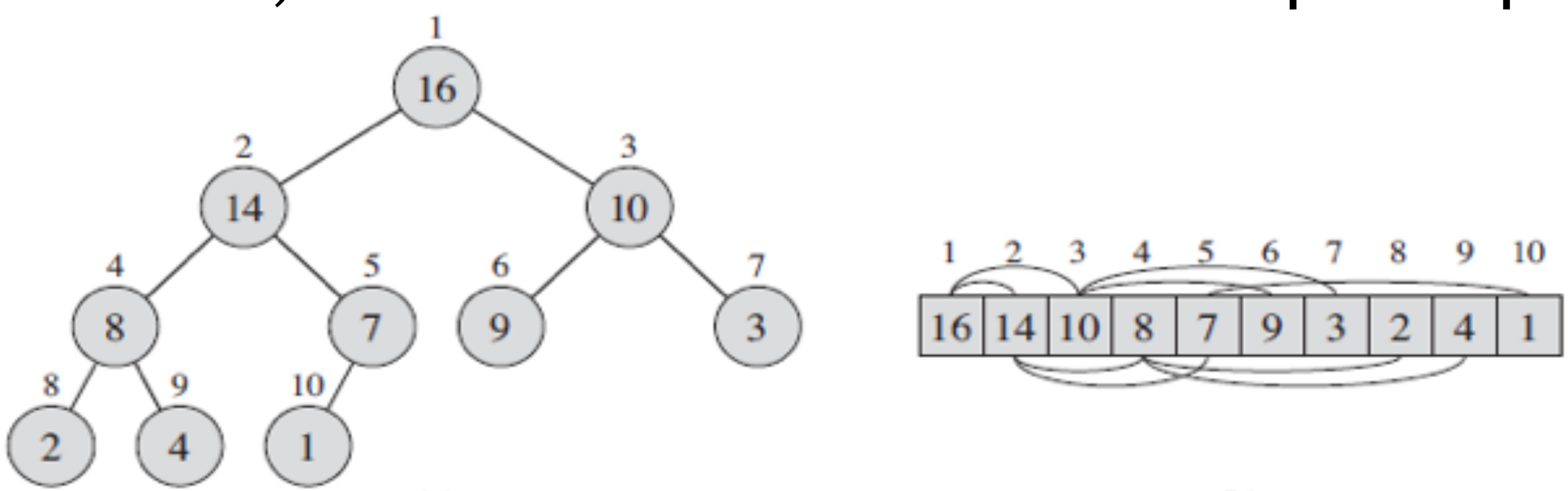
- Number of nodes in a complete binary tree of height h, $2^{h+1} - 1$

Nearly complete binary tree

Recall that Heap is viewed as a Nearly complete Binary tree

Each node in the tree – an element of the array

Tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point



(a) (b)

Two attributes of an array A

- **A.length:** Number of elements in the array
- **A.heapsize**: How many elements in the heap are stored within array A
- Although **A[1…A.*length*]** may contain numbers, only the elements in **A[1…A.*heapsize*]**, where **0<= A.*heapsize* <= A.*length*,** are valid elements of the heap.

Parent, left and right child - Binary heap

- The **root of the tree is A[1]**

Given the index *i* of a node,

Index of its parent:

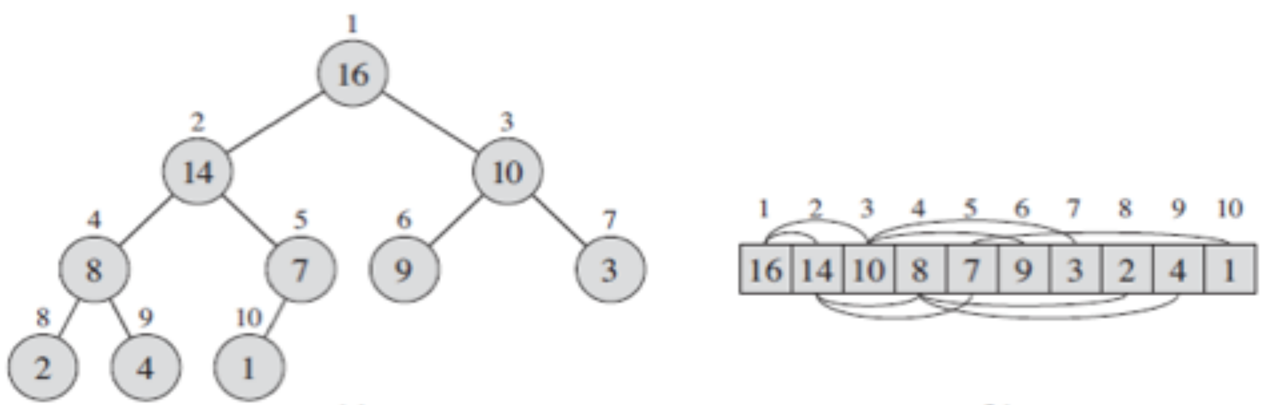
PARENT(i): **floor(i/2)**

Index of left child:

LEFT(i): **2*i**

Index of right child:

RIGHT(i): **2*i+1**



(a) (b)

Max and Min Heaps

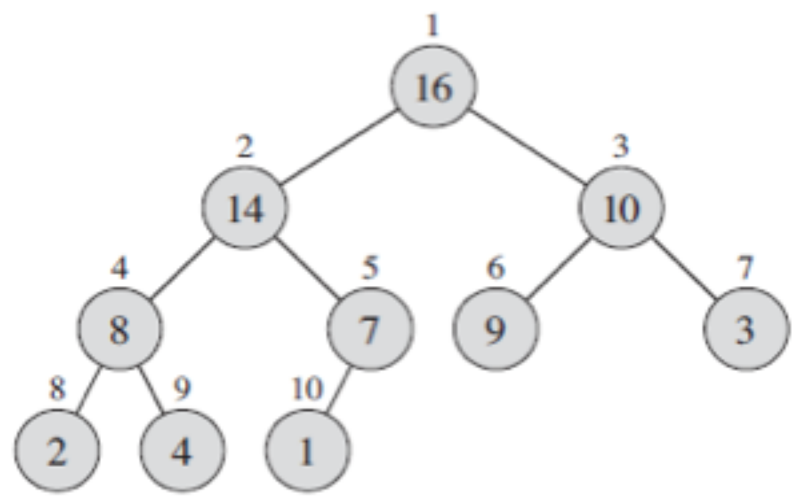
- There are **two kinds** of binary heaps:
  - Max-heaps
  - Min-heaps
- In both kinds, the values in the nodes satisfy a ***heap property***, the specifics of which depend on the kind of heap.

Max-heap

- ***Max-heap property:*** *F*or every node **i** other than the root,

  A[PARENT(i)]>= A[i];

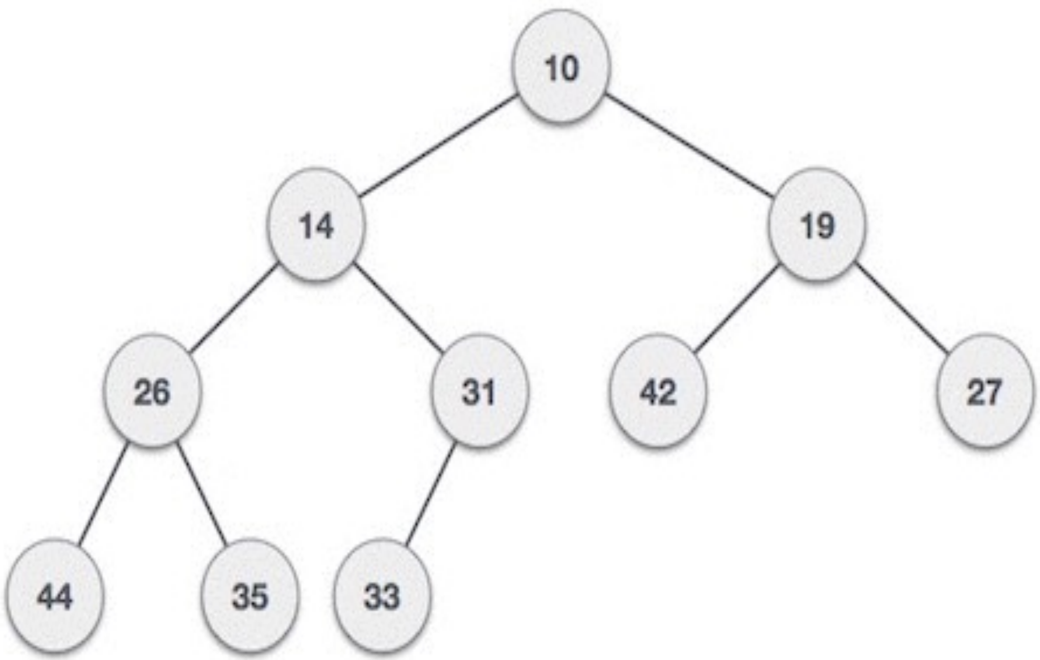
  Value of a node is at most the value of its parent.
- **Largest element** in a **max-heap** is stored at **the root.**
- Subtree rooted at a node contains values no larger than that contained at the node itself.



(a)

Min-heap

- **Min-heap *property*** is that for every node i other than the root, **A[PARENT(i)] <= A[i]**
- The **smallest element** in a min-heap is at **the root.**
- **Example:**



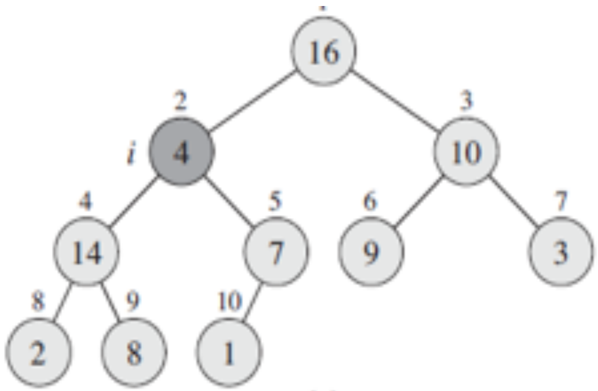
Max/Min Heapify

- How do you establish heap property (Max/ Min) in the given input array?
  - Apply Max/Min-HEAPIFY procedure to establish Max/Min-HEAP property
- Where to apply the Max/Min-HEAPIFY procedure?
  - On the ith element of an array, in which Max/ Min Heap property is violated

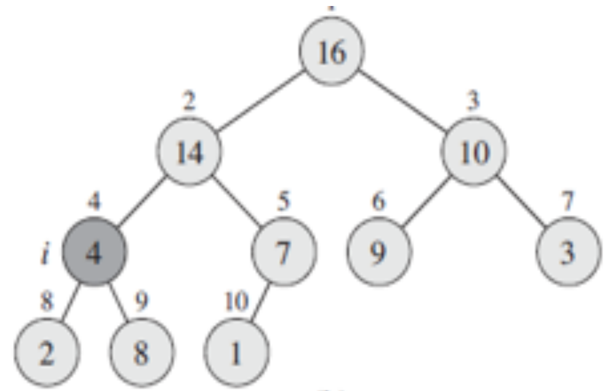
Maintaining the heap property

- To **maintain the max-heap property:** MAX-HEAPIFY
- Inputs are **an array A and an index i** into the array
- MAXHEAPIFY assumes that the binary trees rooted at **LEFT(i) and RIGHT(i) are max heaps**, but that **A[i] might be smaller than its children**
  - 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** satisfies the max-heap property

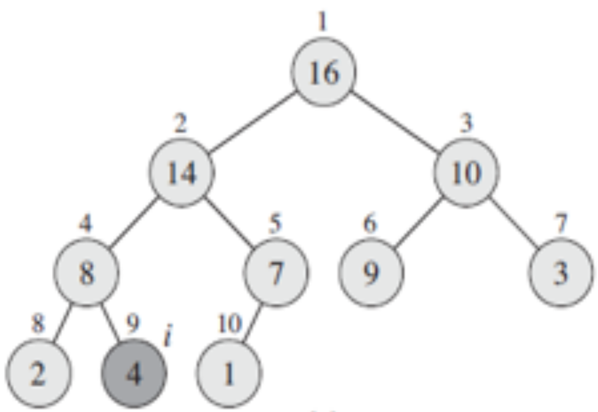
Figure 6.2 The action of MAX-HEAPIFY$(A, 2)$, where $A.heap\text{-}size = 10$. **(a)** The initial configuration, with $A[2]$ at node $i = 2$ violating the max-heap property since it is not larger than both children. The max-heap property is restored for node 2 in **(b)** by exchanging $A[2]$ with $A[4]$, which destroys the max-heap property for node 4. The recursive call MAX-HEAPIFY$(A, 4)$ now has $i = 4$. After swapping $A[4]$ with $A[9]$, as shown in **(c)**, node 4 is fixed up, and the recursive call MAX-HEAPIFY$(A, 9)$ yields no further change to the data structure.



(a)



(b)



(c)

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)
```

How to Build a Max/Min-Heap?

- Use the Max/Min-Heapify procedure

Building a Heap

- Each leaf node can be considered as a 1-element heap to begin with.
- Therefore, for building a max-heap it is sufficient to apply MAX-HEAPIFY on the remaining internal nodes of the tree
- i.e Apply MAX-HEAPIFY in a bottom-up manner to convert array A[1…A.length] into a max-heap
- Where are the leaves in the heap …. ?
- **Ex:** Leaves in the heap are appearing in the subarray $A[\lfloor n/2 \rfloor+1)\ldots n]$

BUILD-MAX-HEAP

Pseudocode for BUILD-MAX-HEAP

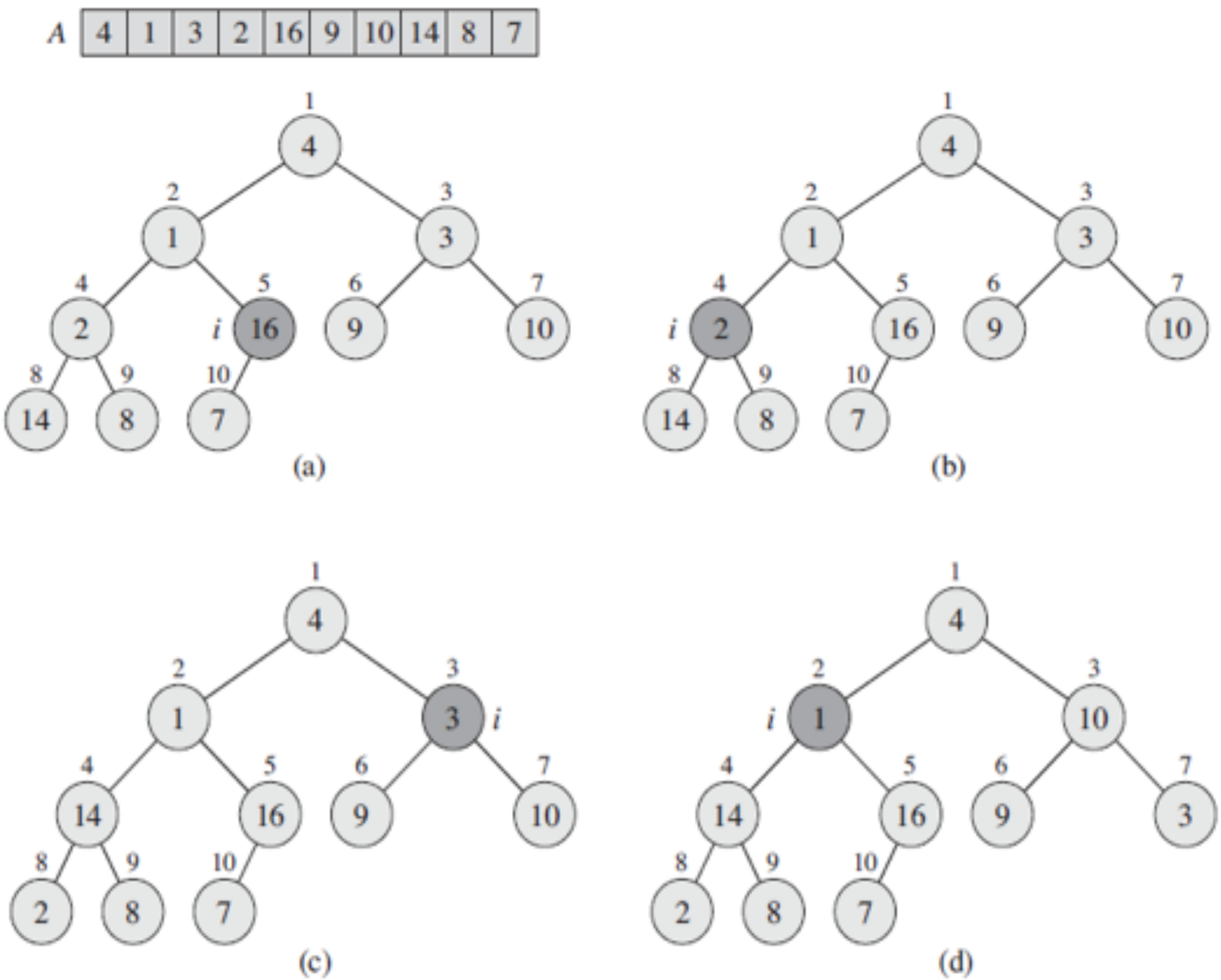
BUILD-MAX-HEAP(A)

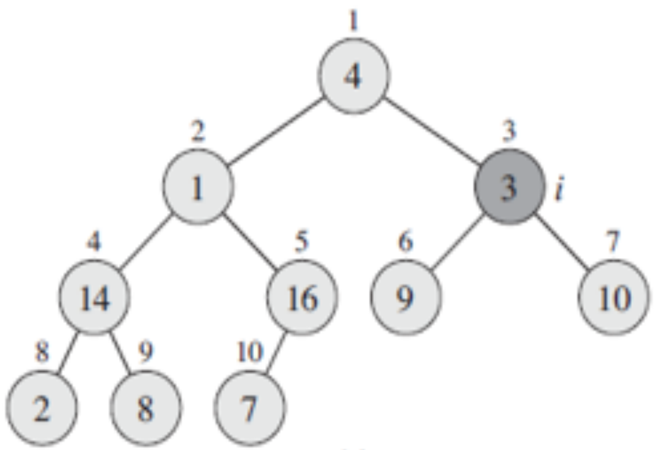
$A.heap\text{-}size = A.length$

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

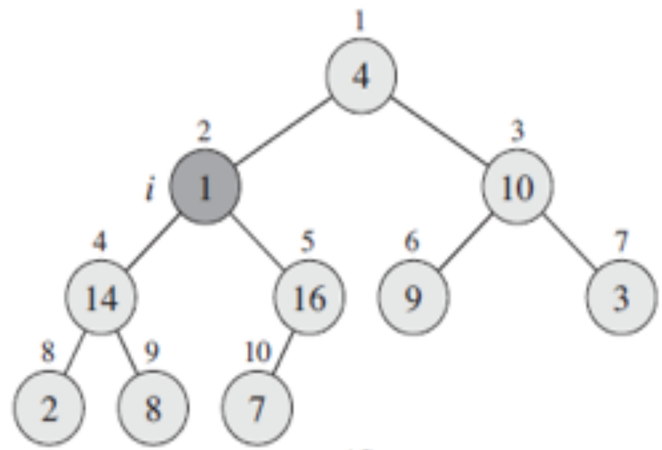
  MAX-HEAPIFY(A, i)

EXAMPLE: Working of BUILD-MAX-HEAP

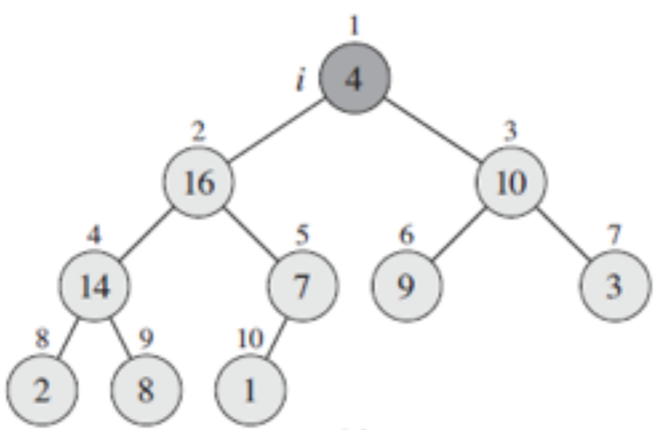




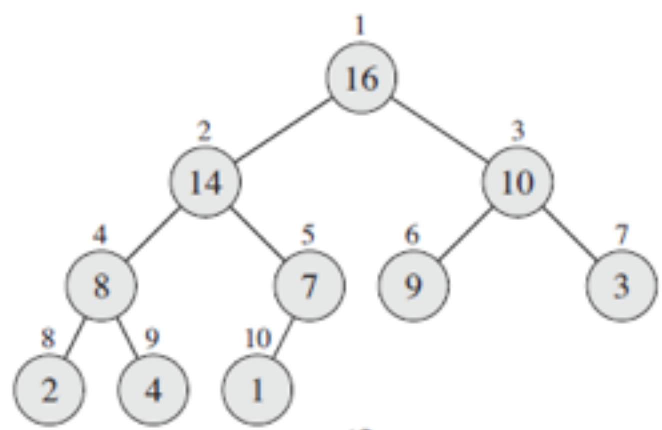
(c)



(d)



(e)



(f)

Idea: HEAP SORT

- Build a Max-heap on the input array A[1…n] where n is the length of the array

- Maximum element is found at the root of the Max-heap

- Exchange this element with the last position of the array i.e exchange A[1] with A[n]

- This may violate the heap property at the root, but its children are Max-heaps

Idea: HEAP SORT

- To restore the Max-heap property,

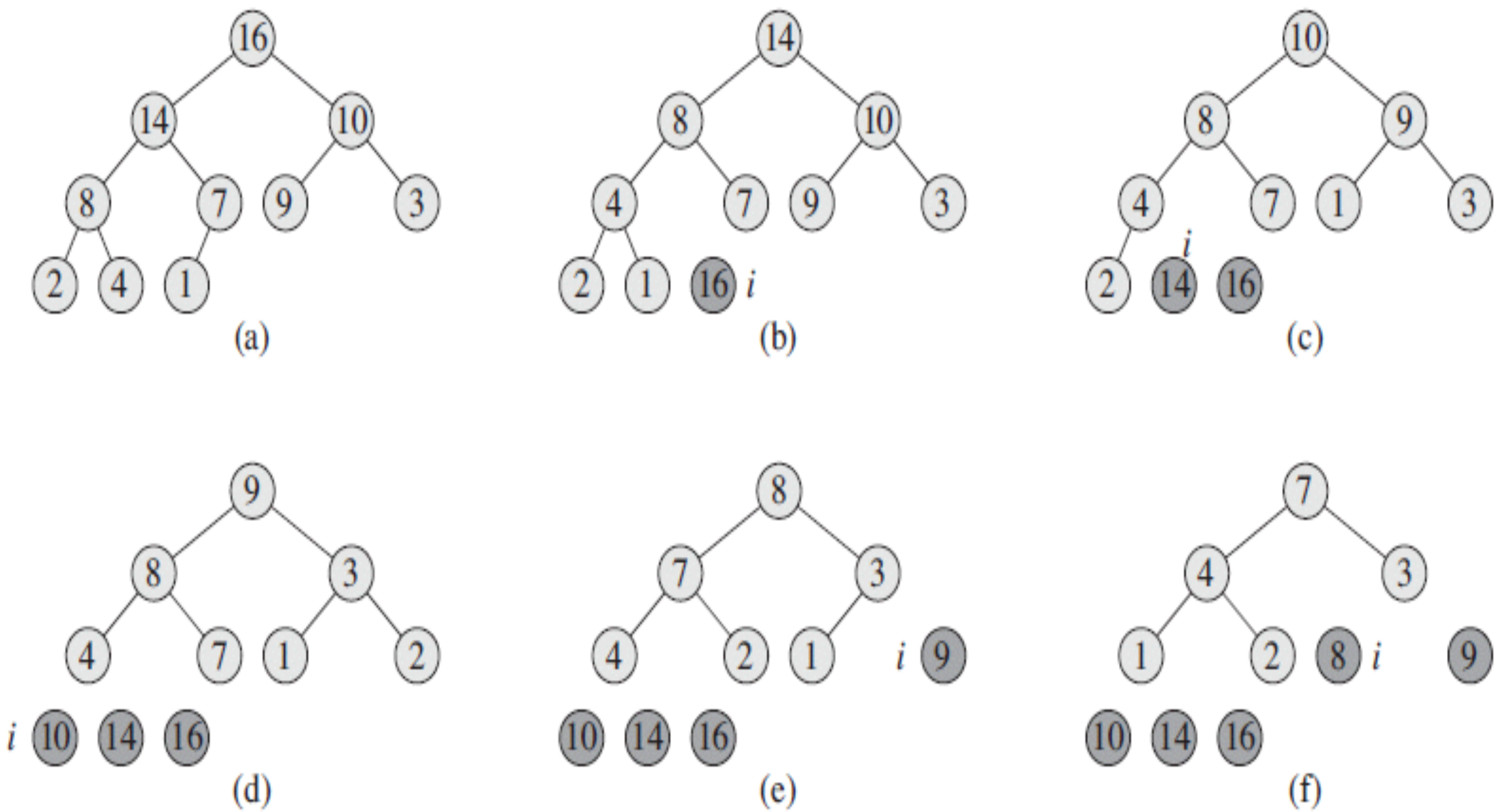
call Max-heapify(A,1) in the n-1 size heap

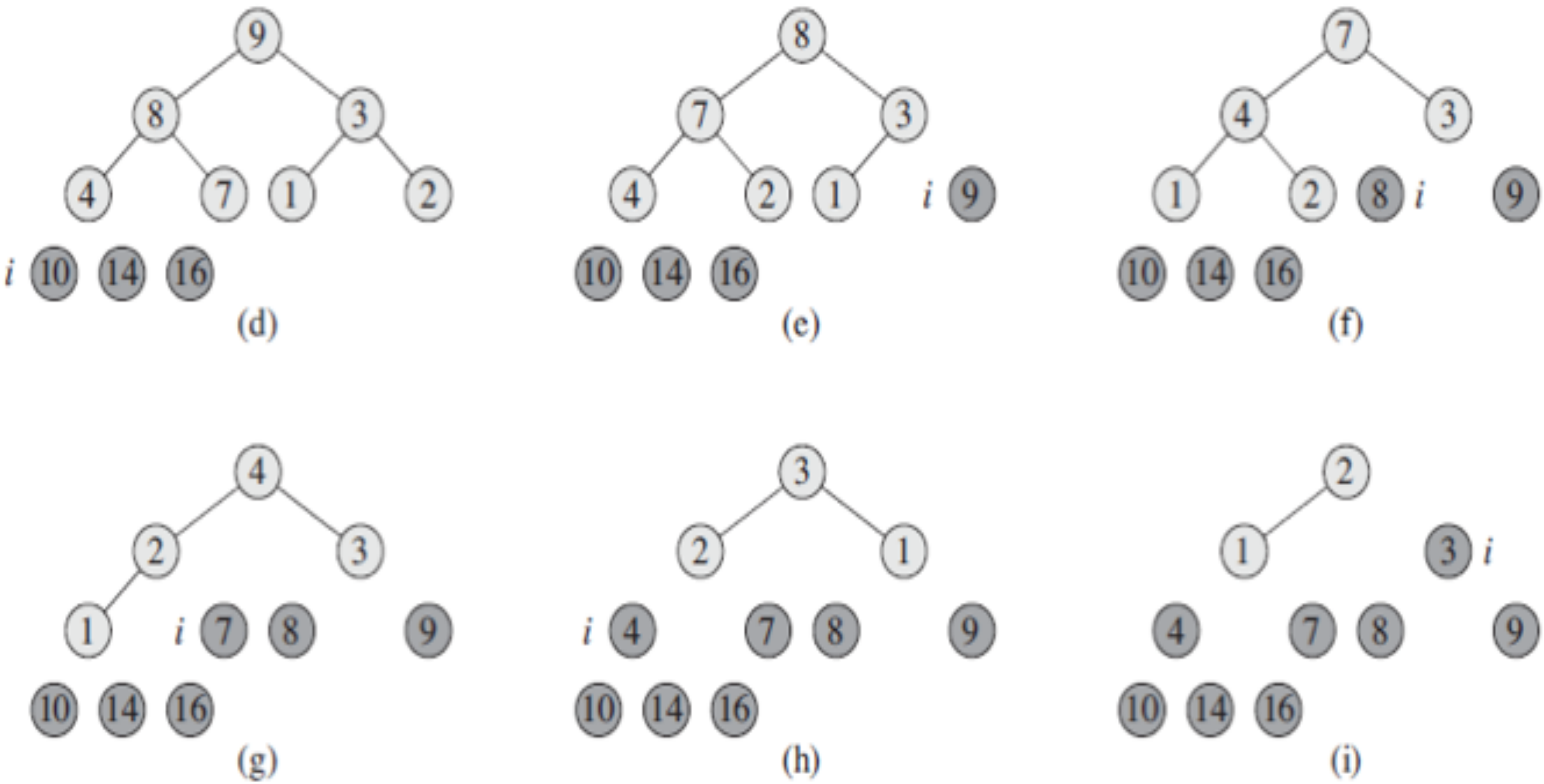
- The heapsort algorithm then repeats this process for the max-heap of size n-1 down to a heap of size 2.

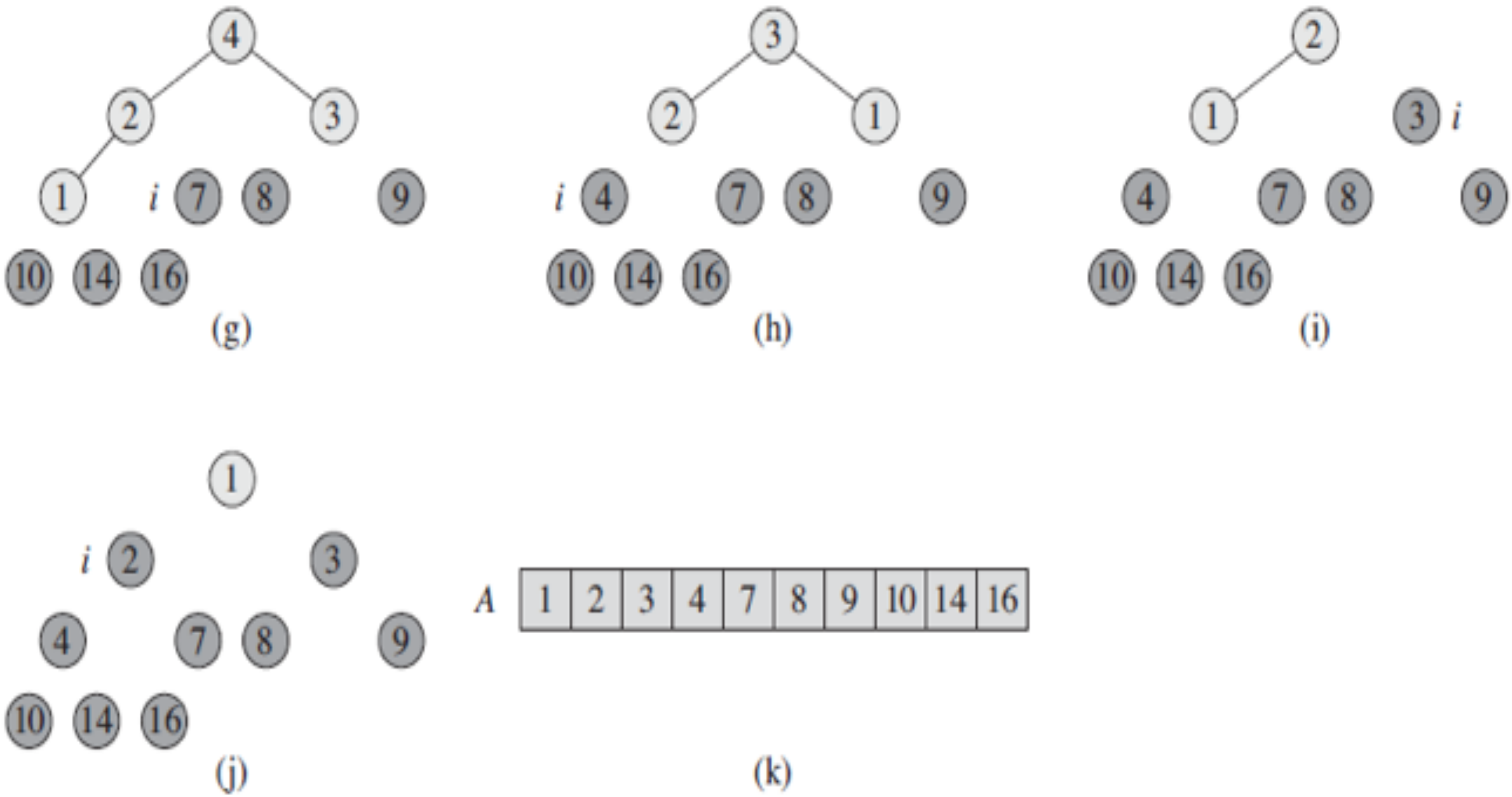
Heap Sort: ALGORITHM

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

Example: Heap Sort




(d)
(e)
(f)
(g)
(h)
(i)


(g)
(h)
(i)
(j)
(k)
A
1 2 3 4 7 8 9 10 14 16