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

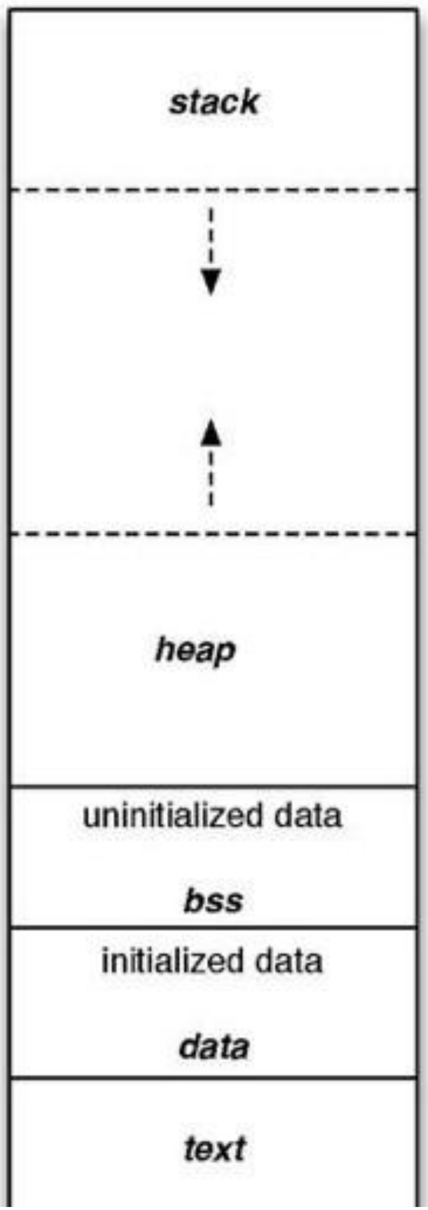
Data Structures

- 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**



Binary heap

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

  Viewed as a **nearly complete binary tree**

- **What is a tree?**
- **What is a binary tree?**
- **What is a complete binary tree?**
- **What is a nearly complete binary tree?**
- **We will have a brief introduction on binary trees now**

- **Details will be discussed later when binary tree and its traversals are discussed**

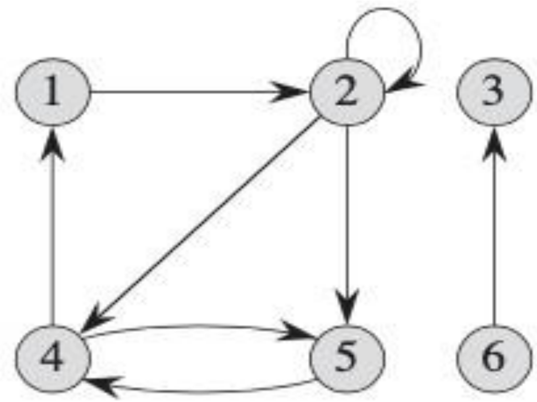
Tree

A Free tree is a connected, acyclic, undirected graph

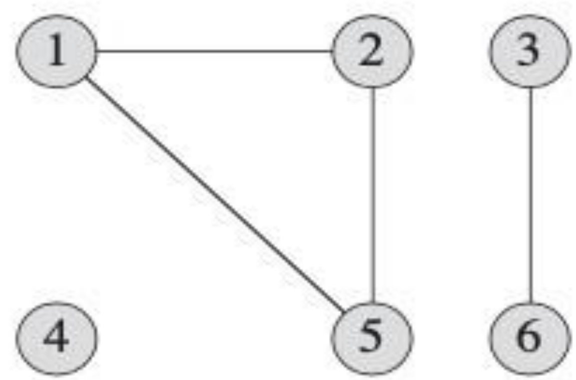
Graphs

Two kinds of Graphs

Directed Graph



Undirected Graph



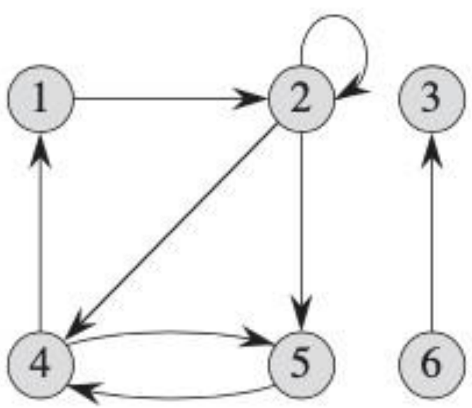
Directed Graph or Digraph

A **directed graph** (or digraph) G is a pair (V, E) where V is a finite set and E is a binary relation on V .

The set V - **vertex set** of G, elements - **vertices**

The set E - **edge set** of G, elements - **edges**.
Pictorial representation of a directed graph on the vertex set {1, 2, 3, 4, 5, 6 }

Vertices - circles, edges - arrows.



Self-loops: edges from a vertex to itself.

Undirected Graph

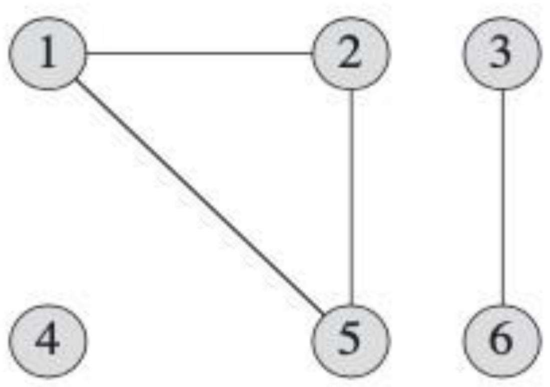
- In an **undirected graph** $G = (V, E)$, the edge set E consists of u, v **unordered pairs of vertices,** rather than ordered pairs.

- An edge is a set $\{u, v\}$, where $u, v \in V$ and $u \neq v$

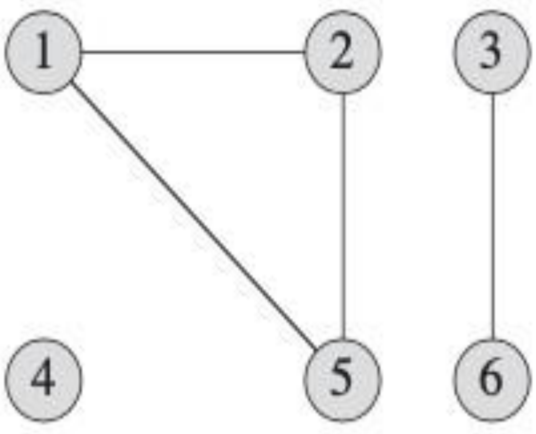
- Use the notation (u, v) for an edge, rather than the set notation $\{u, v\}$ and we consider (u, v) and (v, u) to be the same edge.

- **Self-loops are forbidden** - every edge consists of two distinct vertices

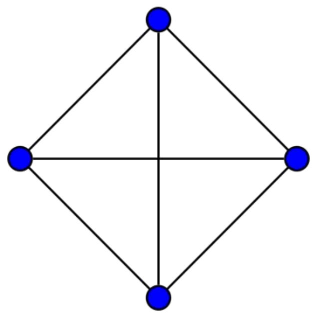
- Pictorial representation of an undirected graph on the vertex set $\{1,2,3,4,5,6\}$



Connected Graph



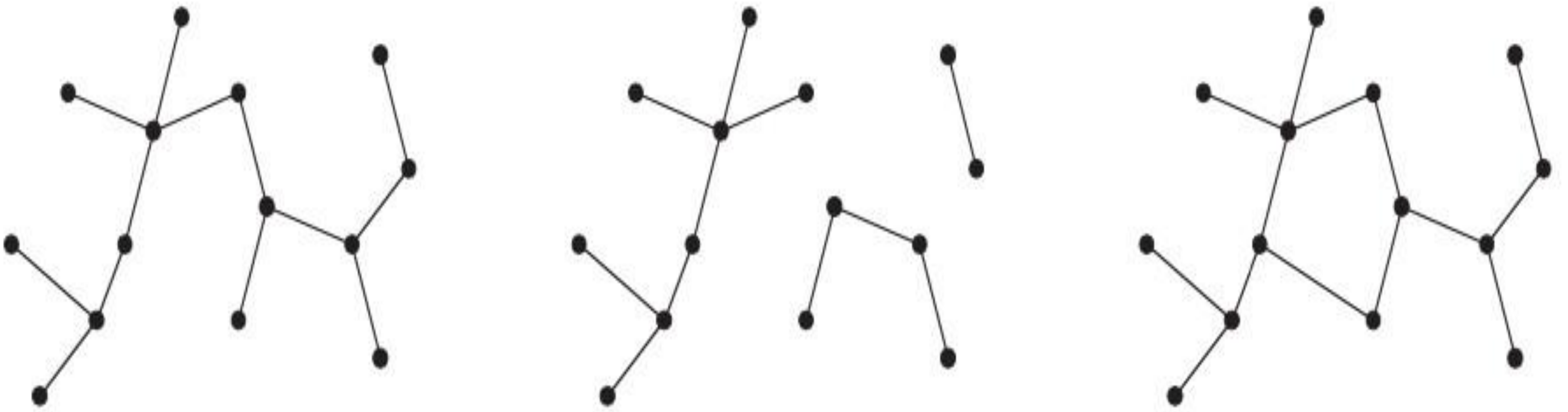
- An undirected graph is **connected** if every vertex is **reachable** from all other vertices.
- The ***connected components*** of a graph are the equivalence classes of vertices under the “is reachable from” relation.



- **Example:** Three components
- Every vertex in {1,2,5} is reachable from every other vertex in {1,2,5}
- An undirected graph is **connected** if it has exactly one connected component.
- A ***complete graph*** is an undirected graph in which every pair of vertices is adjacent.

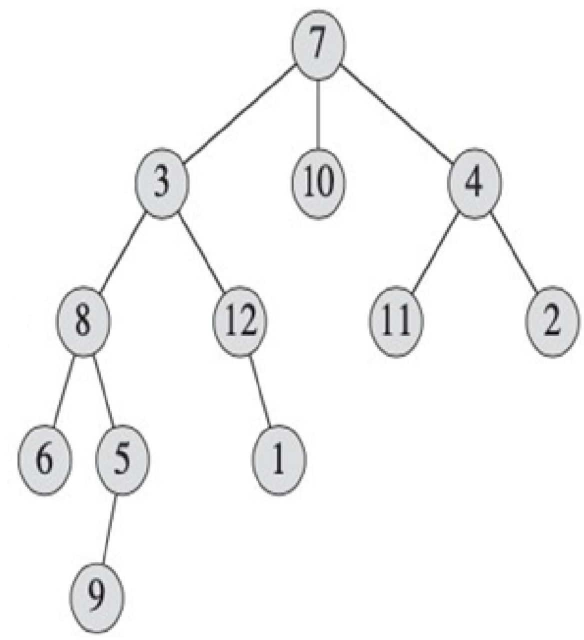
Free Tree

- A **Free tree** is a **connected, acyclic, undirected graph**
- Disconnected **acyclic** undirected graph - ***forest***.



Rooted Trees

- A **rooted tree** is a free tree in which one of the vertices is distinguished from the others.



- Distinguished vertex the **root** of the tree.
- **Vertex** of a rooted tree as a **node of the tree**.
- Figure shows a rooted tree on a **set of 12 nodes with root 7.**

Binary Trees

- 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

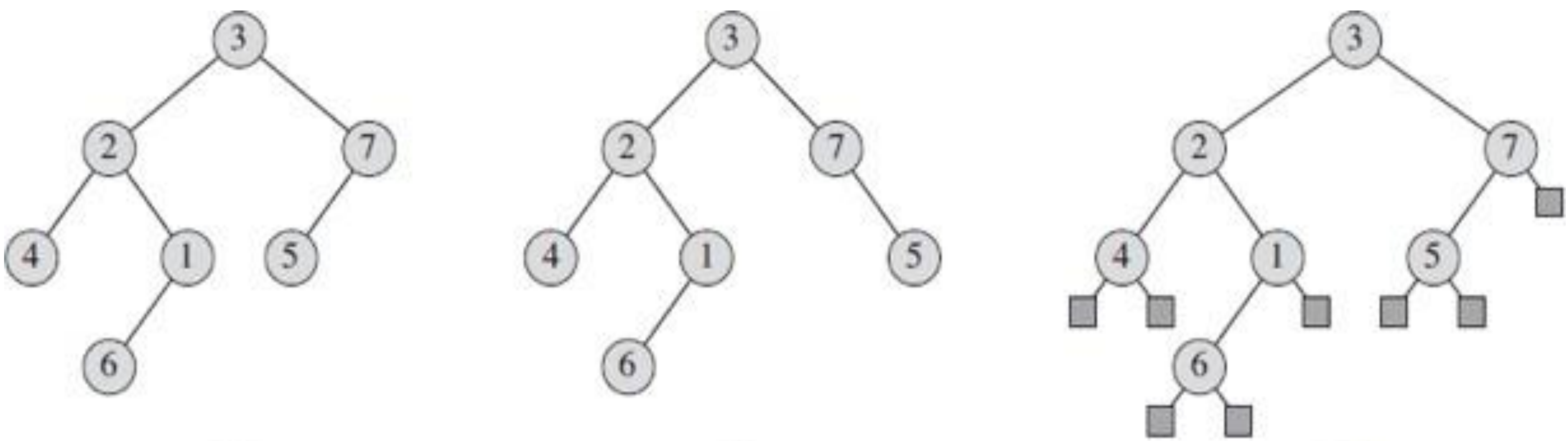
Binary tree - examples

In the following example:

rootnode: **Node 3**

left subtree: **Nodes 2,4,1 and 6** together form Left subtree

right subtree: **Nodes 7 and 5** together form Right subtree



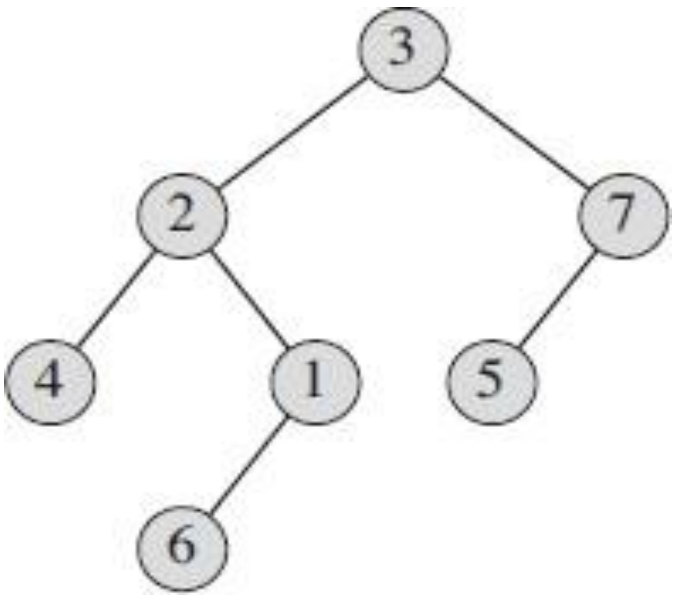
(a) (b) (c)

Few terms….

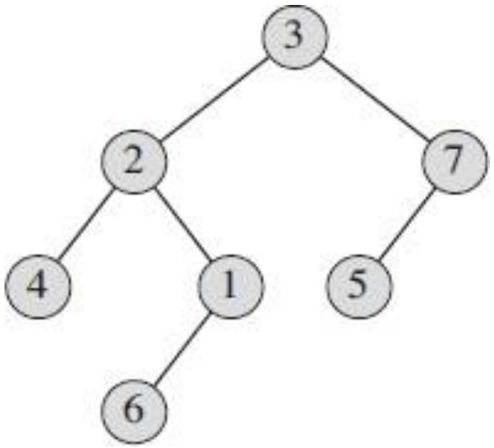
- The **number of children** of a node x in a rooted tree T equals the ***degree* of x**

Example:

- **Degree of node 3: 2**
- **Degree of node 7: 1**
- **Degree of node 6: 0**

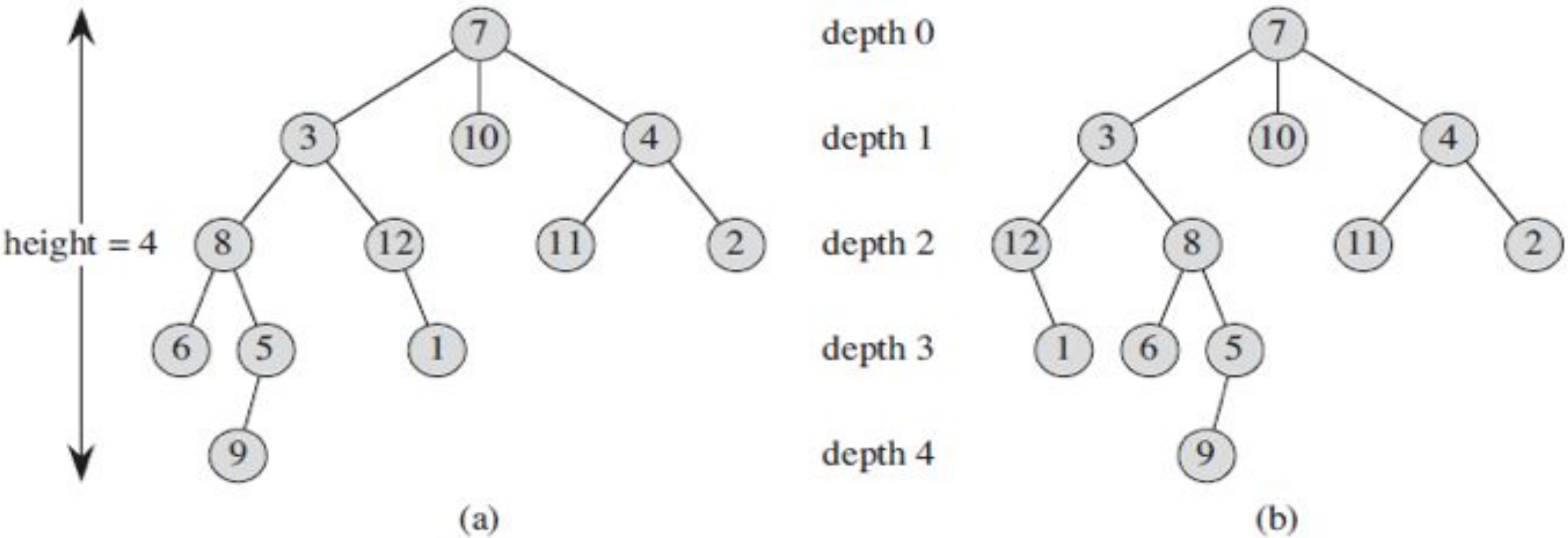


- The length of the simple path from the root r to a node x is the *depth* of x in T .
  - Eg: Depth of node 6:3
  - Depth of nodes 1, 4, 5:2
  - Depth of nodes 2 and 7:1
  - Depth of node 3:0



- 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?



- 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.**

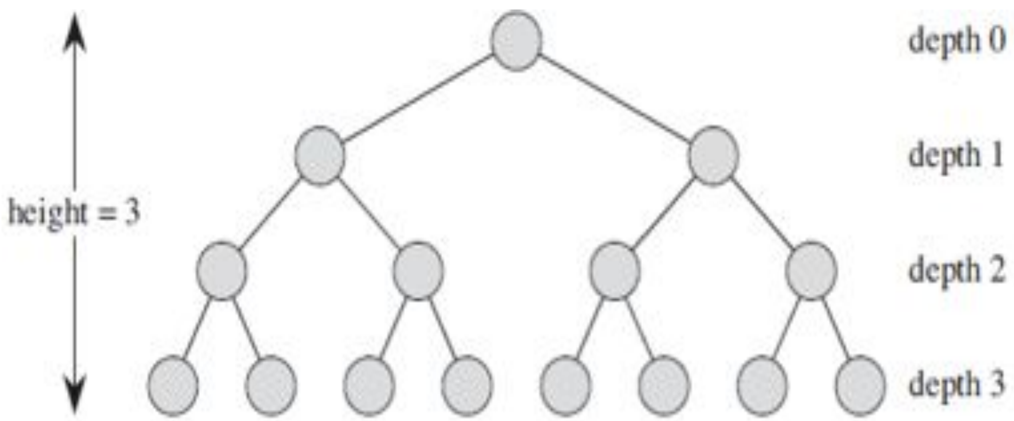
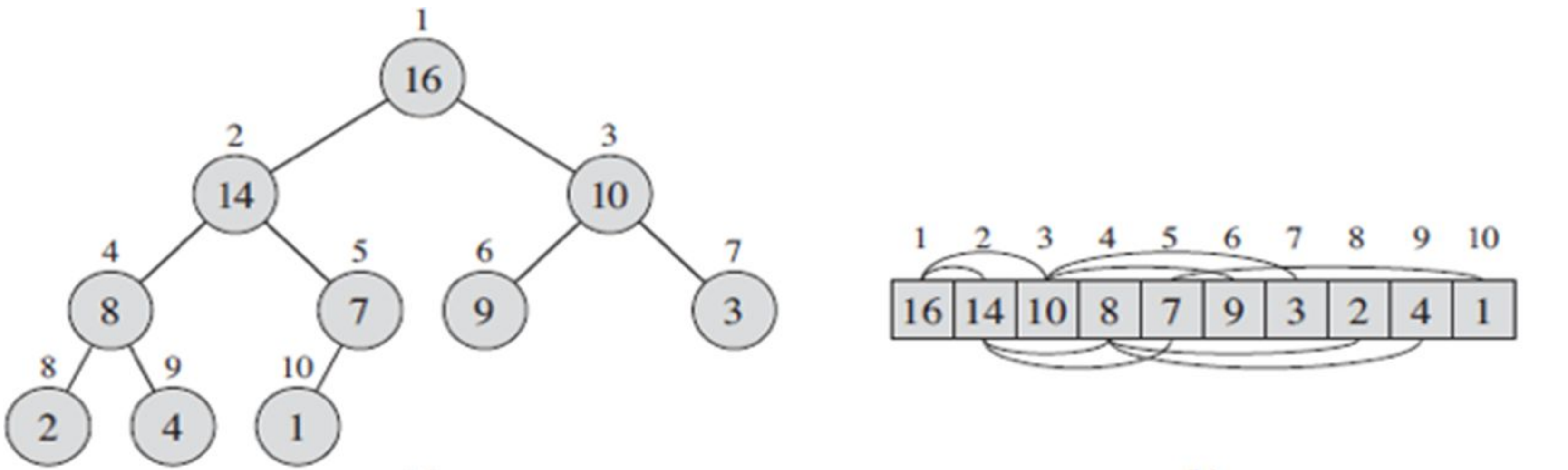


Figure B.8 A complete binary tree of height 3 with 8 leaves and 7 internal nodes.

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

Nearly complete binary tree

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



(a) (b)

Recall that Heap is viewed as a Nearly complete Binary tree and Binary heap data structure is an array object

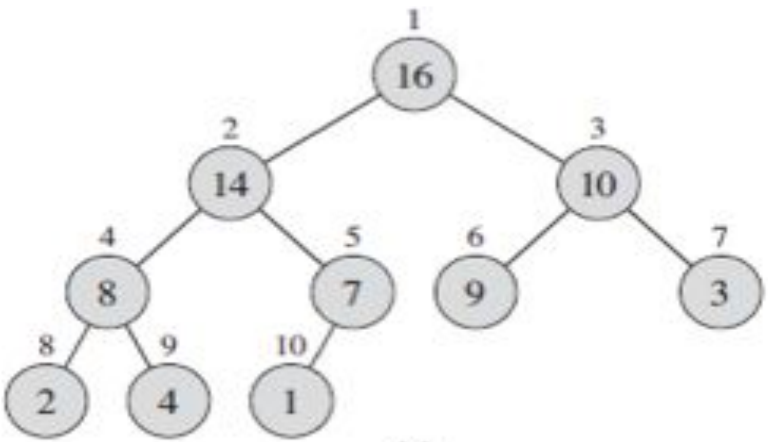
Each node in the tree – an element of the array

Two attributes of an array A

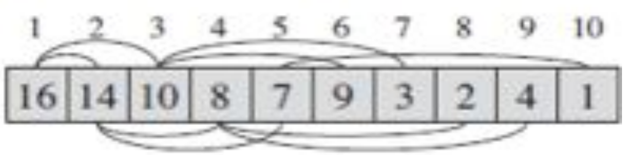
- **A.length:** Number of elements in the array
- **A.heapsize**: How many elements in the heap are stored within array A
- Although **A[1…A.*length*]** may contain numbers, only the elements in **A[1…A.*heapsize*]**, where **0<= A.*heapsize* <= A.*length*,** are valid elements of the heap.

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)

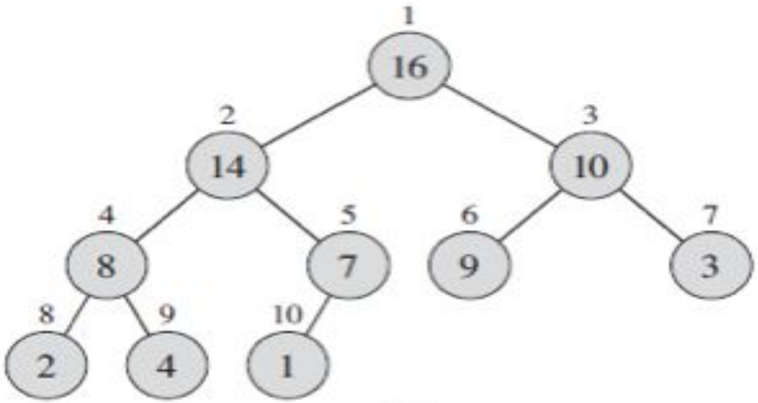
Max and Min Heaps

- 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];**

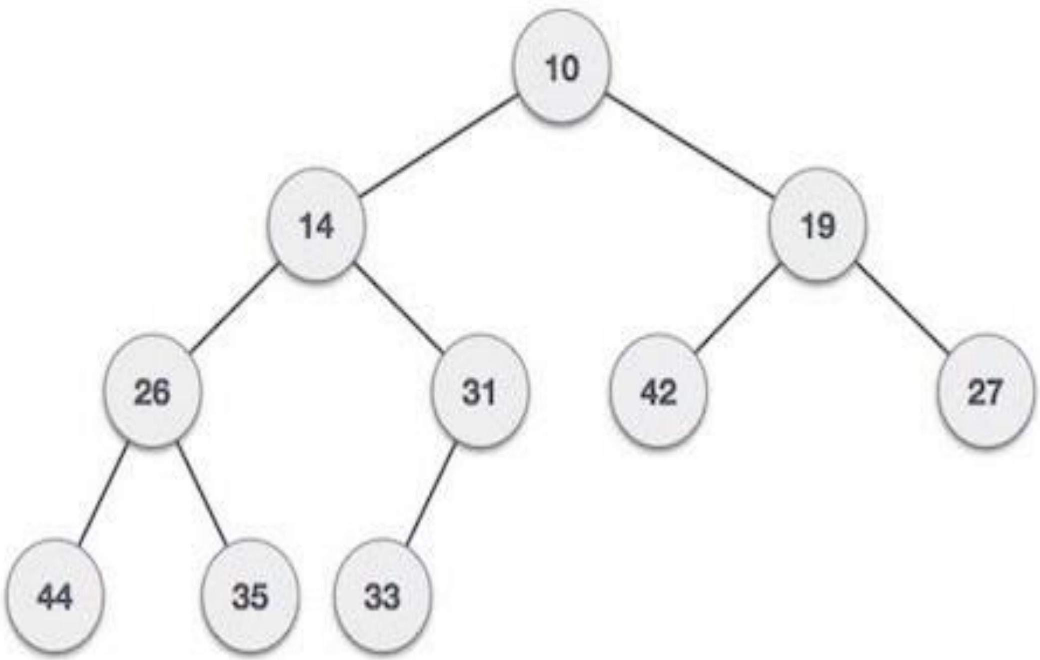
  Value of a node is at most the value of its parent.
- **Largest element** in a **max-heap** is stored at **the root.**
- Subtree rooted at a node contains values no larger than that contained at the node itself.



(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

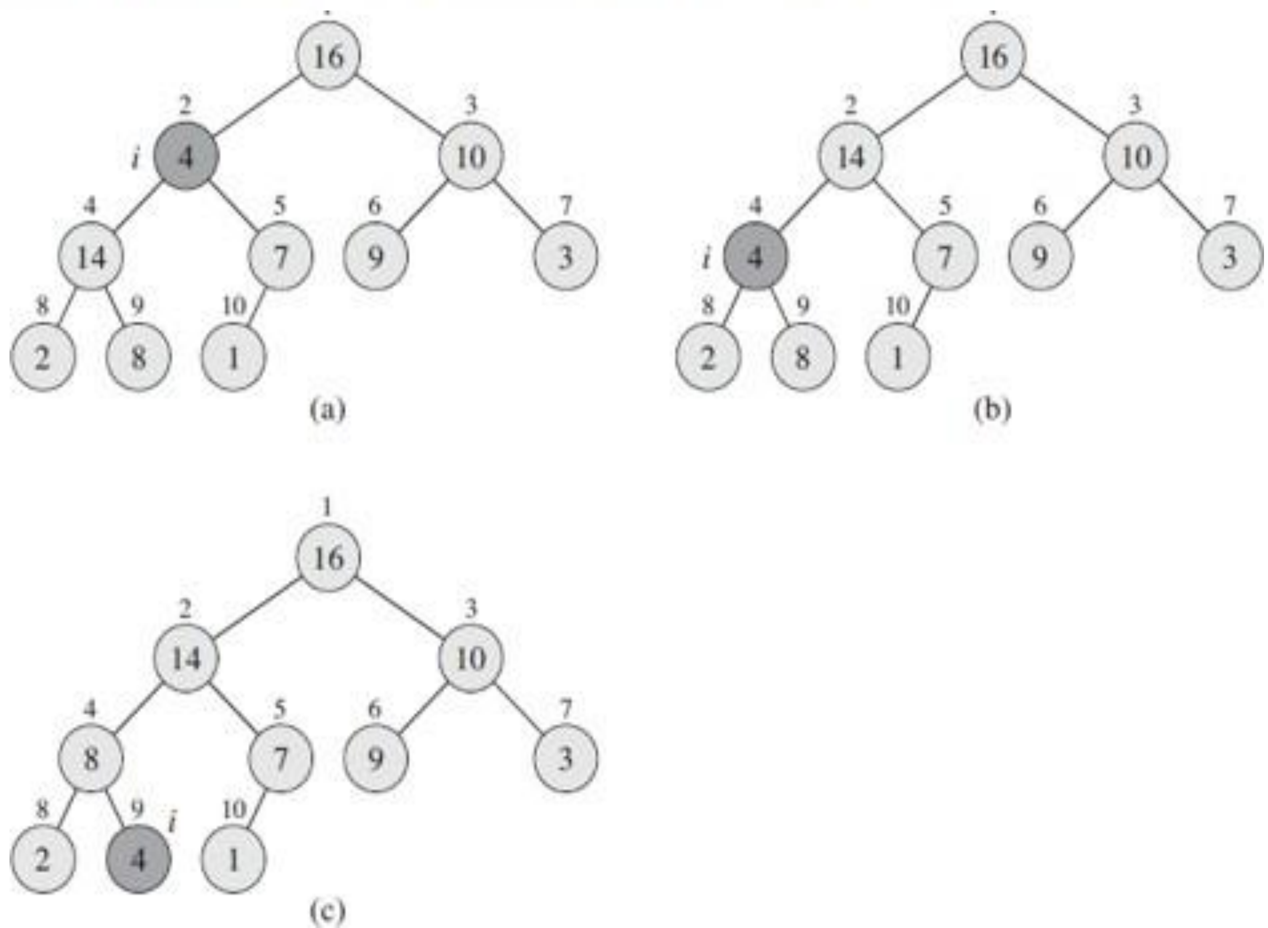
  LEFT(i) and RIGHT(i) are max heaps, but that **A[i] might be smaller than its children**

  _ 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

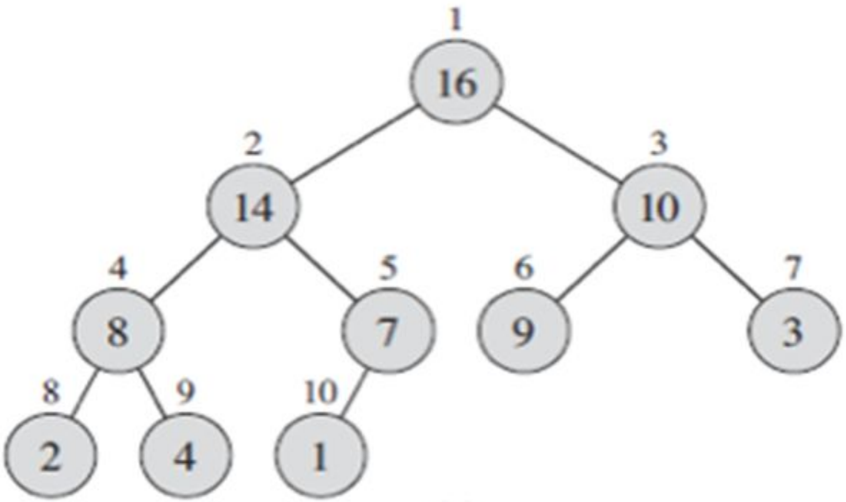
Figure 6.2 The action of MAX-HEAPIFY$(A, 2)$, where $A.heap\text{-}size = 10$. **(a)** The initial configuration, with $A[2]$ at node $i = 2$ violating the max-heap property since it is not larger than both children. The max-heap property is restored for node 2 in **(b)** by exchanging $A[2]$ with $A[4]$, 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.



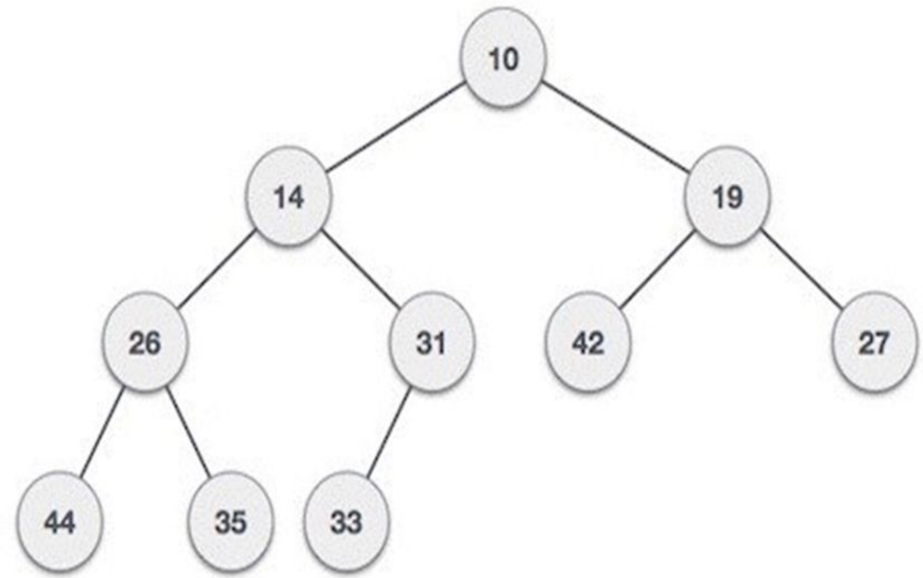
MAX-HEAPIFY

```
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?



(a)



- Use the Max/Min-Heapify procedure

Building a Heap

- Each leaf node can be considered as a 1-element heap to begin with.

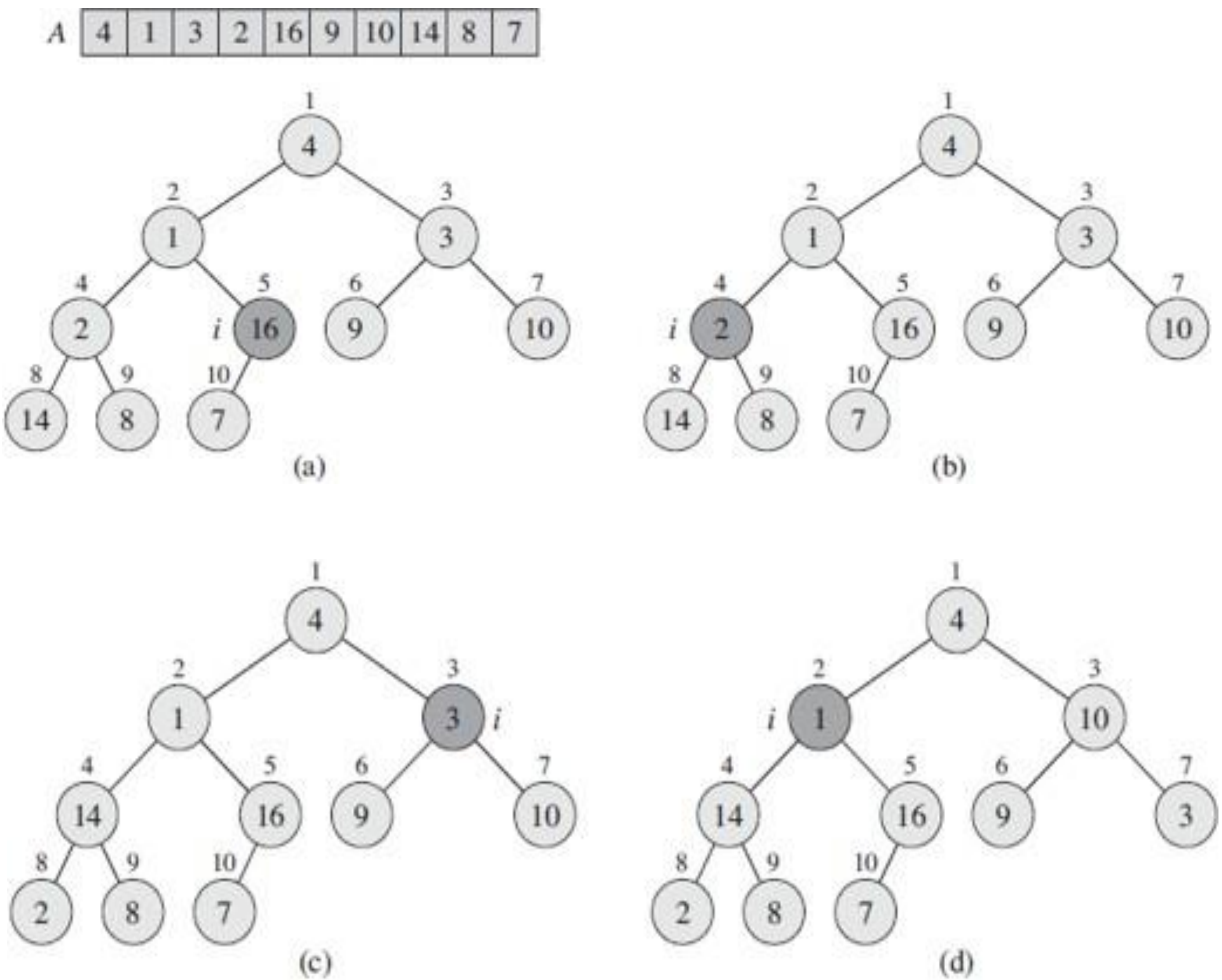
- Therefore, for building a max-heap it is sufficient to apply MAX-HEAPIFY on the remaining internal nodes of the tree

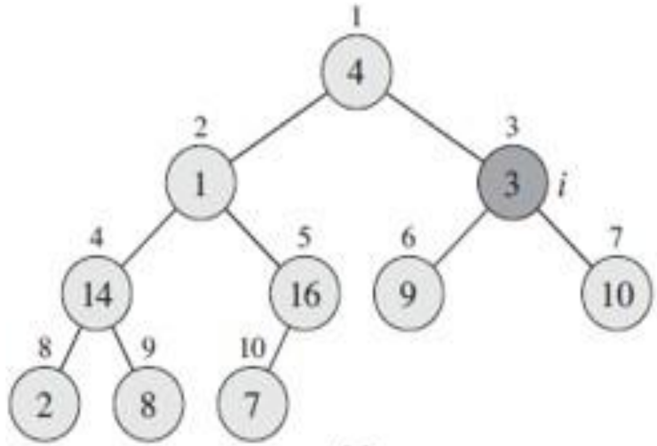
- i.e Apply MAX-HEAPIFY in a bottom-up manner to convert array A[1…A.length] into a max-heap

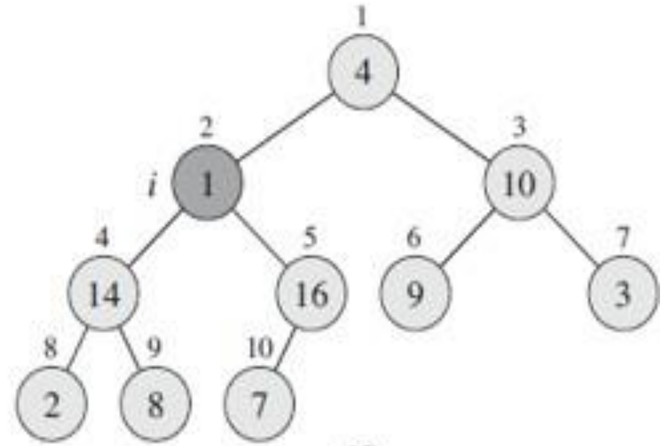
- Where are the leaves in the heap …. ?

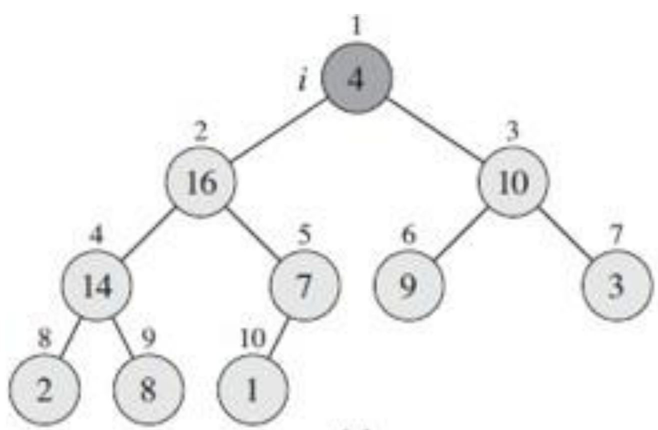
- **Ex:** Leaves in the heap are appearing in the subarray A [⌊ ⌊n/2⌋ +1) … n]

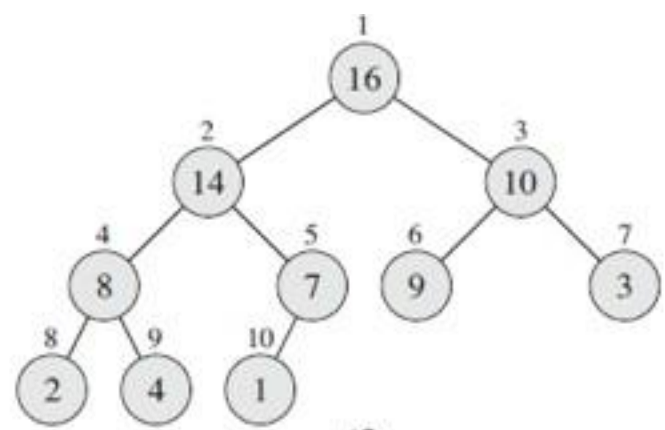
EXAMPLE: Working of BUILD-MAX-HEAP




1 4
2 1 3 3 i
4 14 5 16 6 9 7 10
8 2 9 8 10 7
(c)


1 4
i 2 1 3 10
4 14 5 16 6 9 7 3
8 2 9 8 10 7
(d)


i 1 4
2 16 3 10
4 14 5 7 6 9 7 3
8 2 9 8 10 1
(e)


1 16
2 14 3 10
4 8 5 7 6 9 7 3
8 2 9 4 10 1
(f)

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)

Heapsort

Input : an array
Output : sorted array

Eg: <12, 13, 18, 9, -7, 0, 6> ----Input
<-7, 0, 6, 9, 12, 13, 18> --- output

Note that the array might not be a max heap

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.

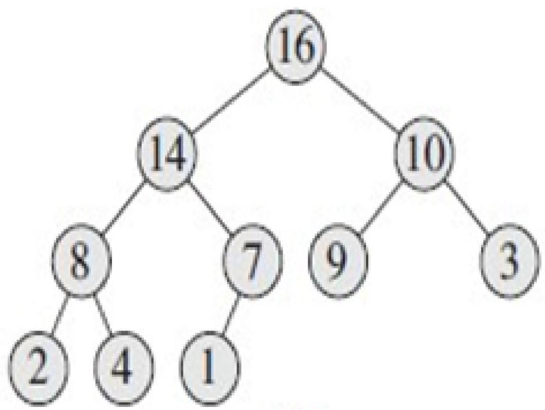
Example: Heap Sort

16	14	10	8	7	9	3	2	4	1

Input

1	2	3	4	7	8	9	10	14	16

Output



(a)

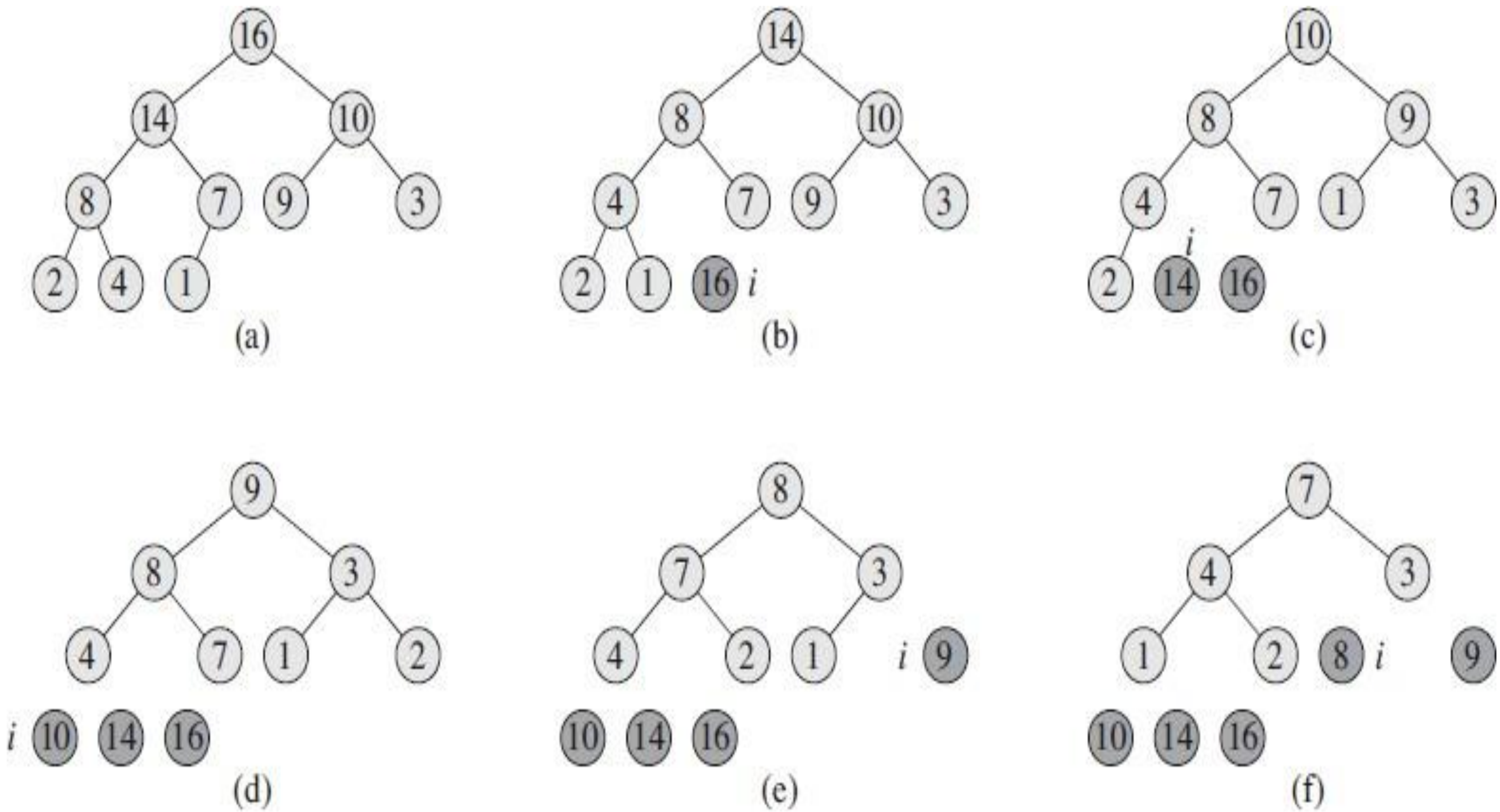
Example: Heap Sort

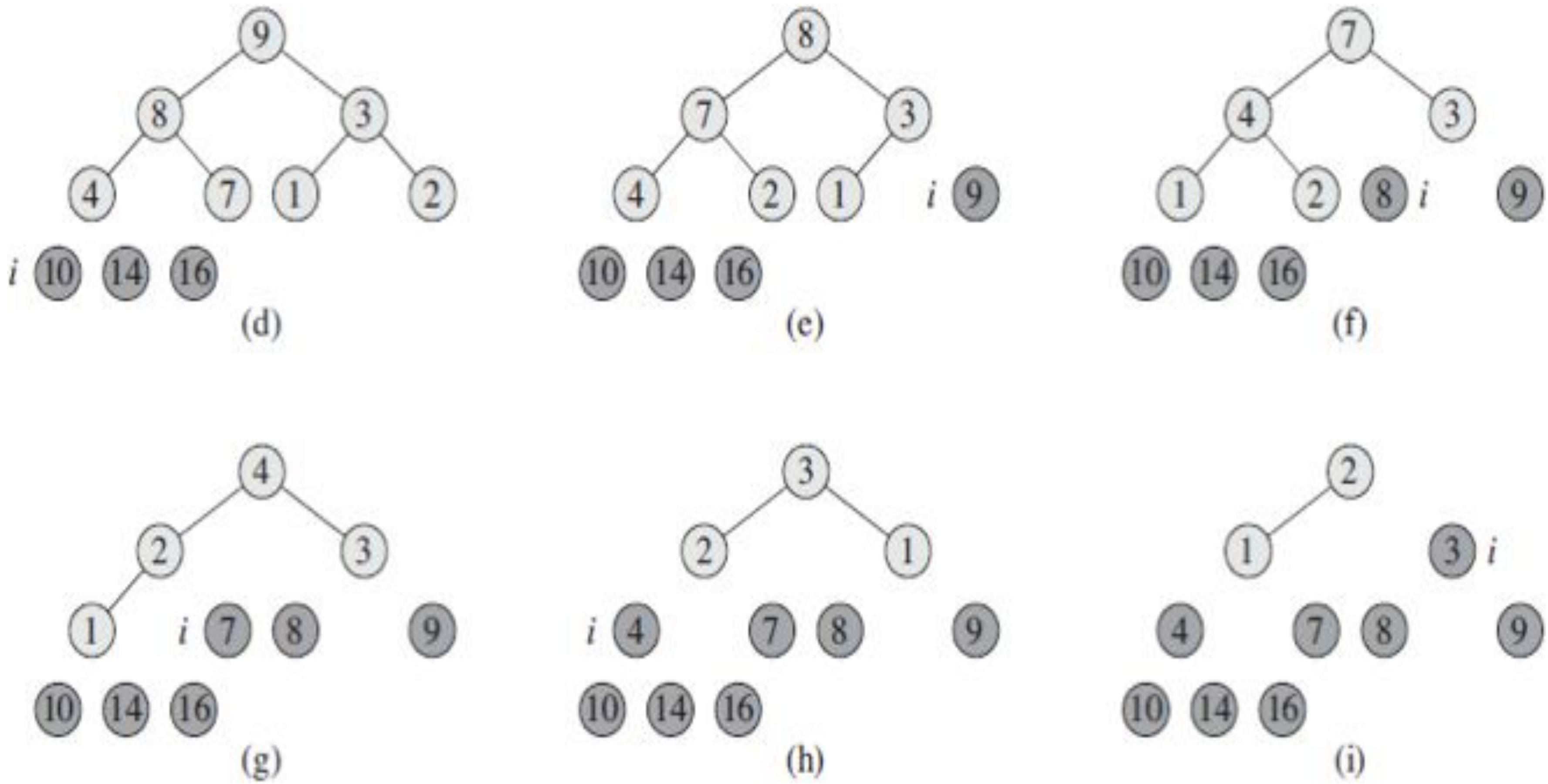
16	14	10	8	7	9	3	2	4	1

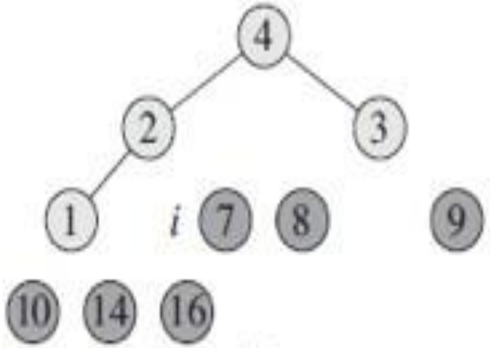
Input

1	2	3	4	7	8	9	10	14	16

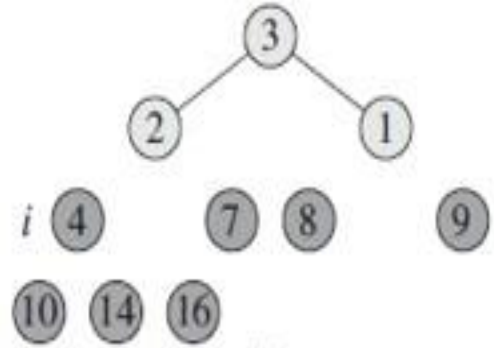
Output



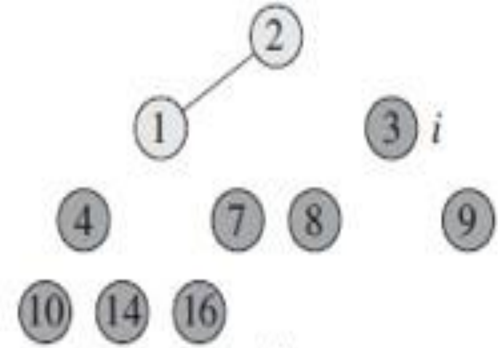

(d)
(e)
(f)
(g)
(h)
(i)



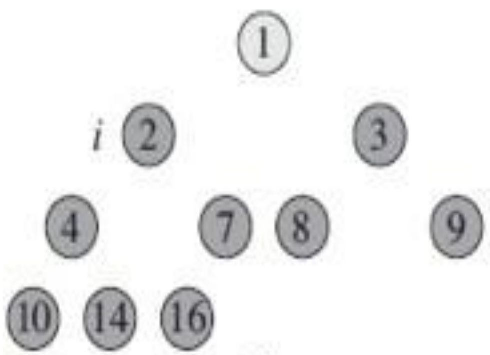
(g)



(h)



(i)



(j)

A	1	2	3	4	7	8	9	10	14	16

(k)

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)
```

Thank you