HEAPSORT & PRIORITY QUEUES

Juan Mendivelso

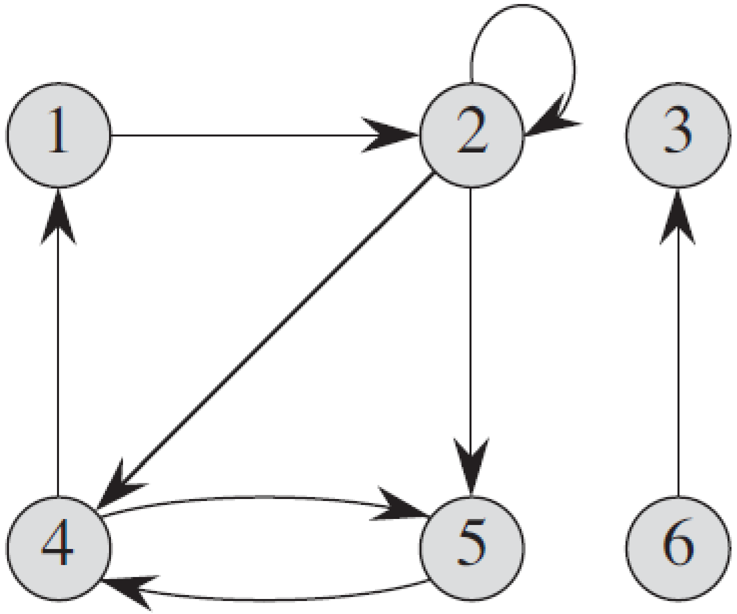
Presentation made by Juan Mendivelso. Contents and figures extracted from the book: Introduction to Algorithms, Third Edition. Cormen, Leiserson, Rivests and Stein. The MIT Press. 2009.

CONTENTS

1. TREES

Digraphs

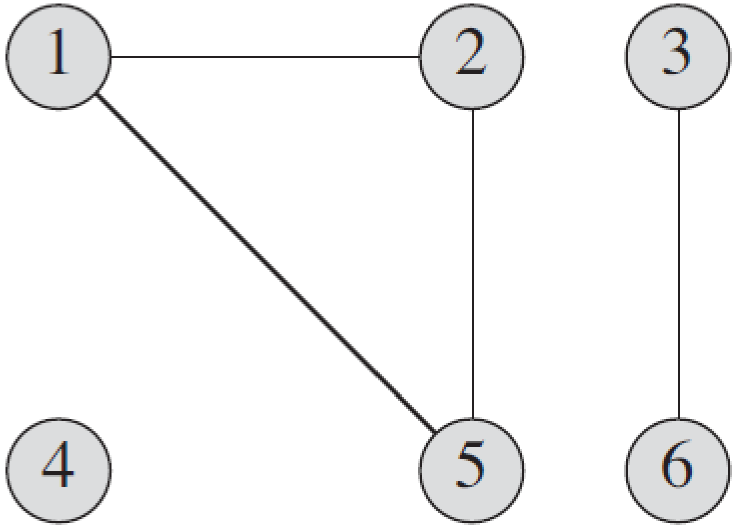
- A directed graph (digraph) G is a pair (V,E), where V is a finite set and E is a binary relation on V.
  - V is the vertex set.
  - E is the edge set.
- Self-loops are permitted.



Presentation made by Juan Mendivelso. Contents and figures extracted from the book: Introduction to Algorithms, Third Edition. Cormen, Leiserson, Rivests and Stein. The MIT Press. 2009.

Undirected Graphs

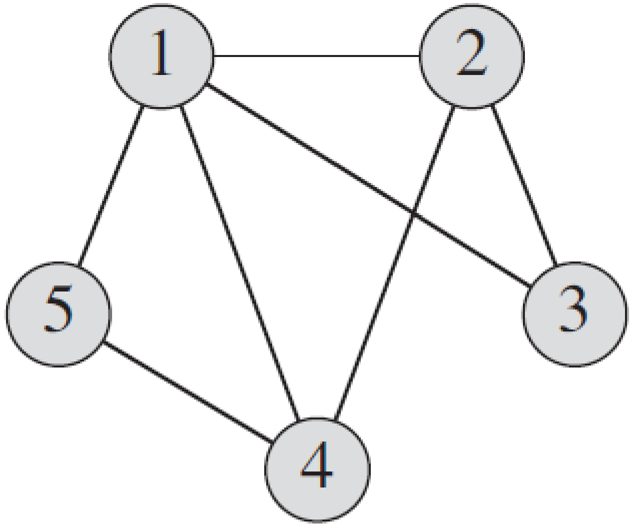
- In an undirected graph G=(V,E), the edge set consists of unordered pairs of vertices rather than ordered sets.
- An edge is a set {u,v}, where u,v are in V and u ≠ v.
- But, by convention, we use the notation (u,v).
- However, (u,v) and (v,u) refer to the same edge.
- Self-loops are not permitted.



Presentation made by Juan Mendivelso. Contents and figures extracted from the book: Introduction to Algorithms, Third Edition. Cormen, Leiserson, Rivests and Stein. The MIT Press. 2009.

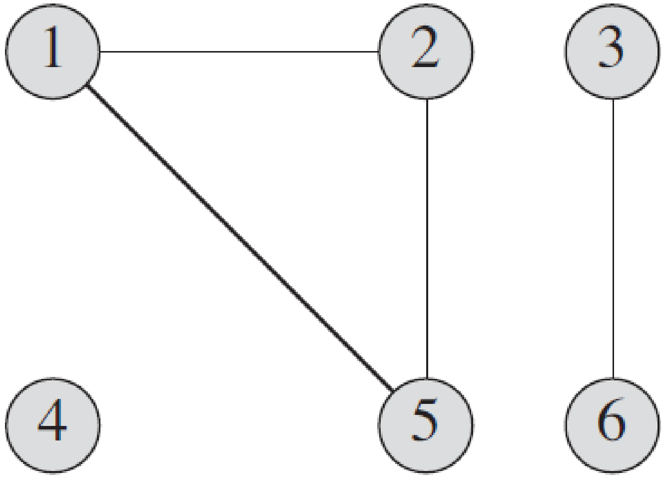
Paths & Cycles

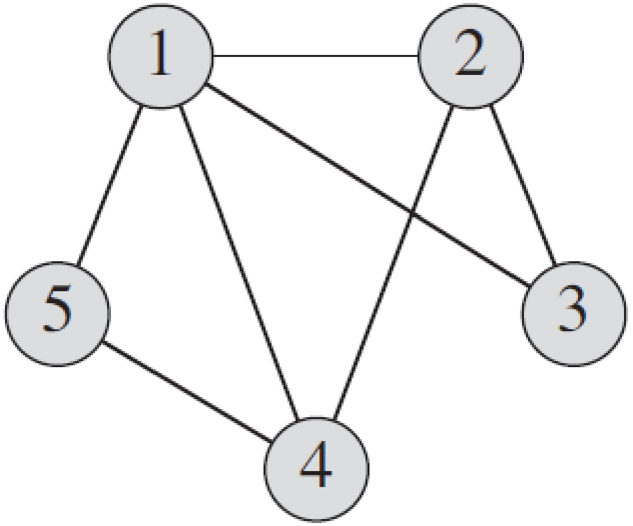
- A path of length k from vertex u to vertex u' in a graph $G=(V,E)$ is a sequence of vertices $v_0,v_1,\dots,v_k$ such that $u=v_0$, $u'=v_k$, and (v_{i-1},v_i) are in E for $i=1,2,\dots,k$.
- If there is a path p from u to u', we say that u' is reachable from u via p.
- A path forms a cycle if $v_0 = v_k$.
- A graph with no cycles is **acyclic**.



Connected Undirected Graph

- An undirected graph is connected if every vertex is reachable from all other vertices.

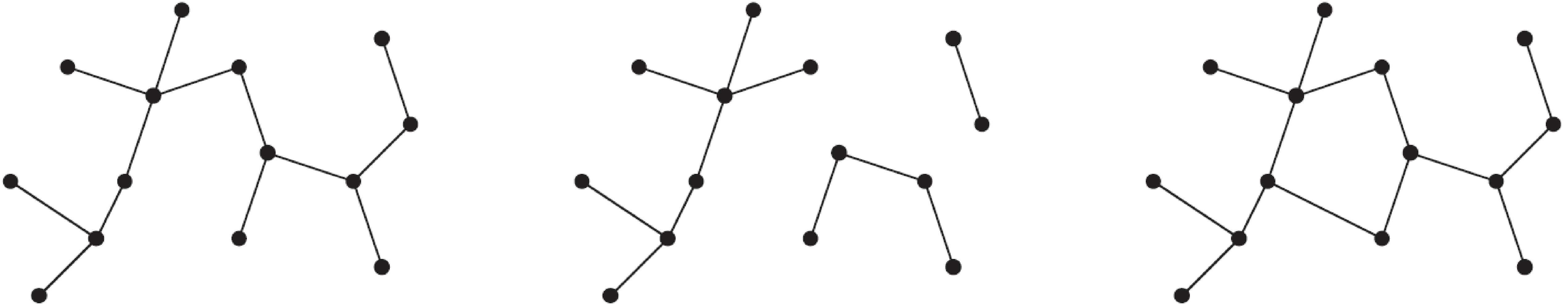




Presentation made by Juan Mendivelso. Contents and figures extracted from the book: Introduction to Algorithms, Third Edition. Cormen, Leiserson, Rivests and Stein. The MIT Press. 2009.

Free Trees & Forests

- A **free tree** is a connected, acyclic undirected graph.
- A **forest** is an acyclic undirected graph.



(a) (b) (c)

Properties of Free Trees

Theorem B.2 (Properties of free trees)
Let $G = (V, E)$ be an undirected graph. The following statements are equivalent.

1. G is a free tree.
2. Any two vertices in G are connected by a unique simple path.
3. G is connected, but if any edge is removed from E, the resulting graph is disconnected.
4. G is connected, and $|E| = |V| - 1$.
5. G is acyclic, and $|E| = |V| - 1$.
6. G is acyclic, but if any edge is added to E, the resulting graph contains a cycle.

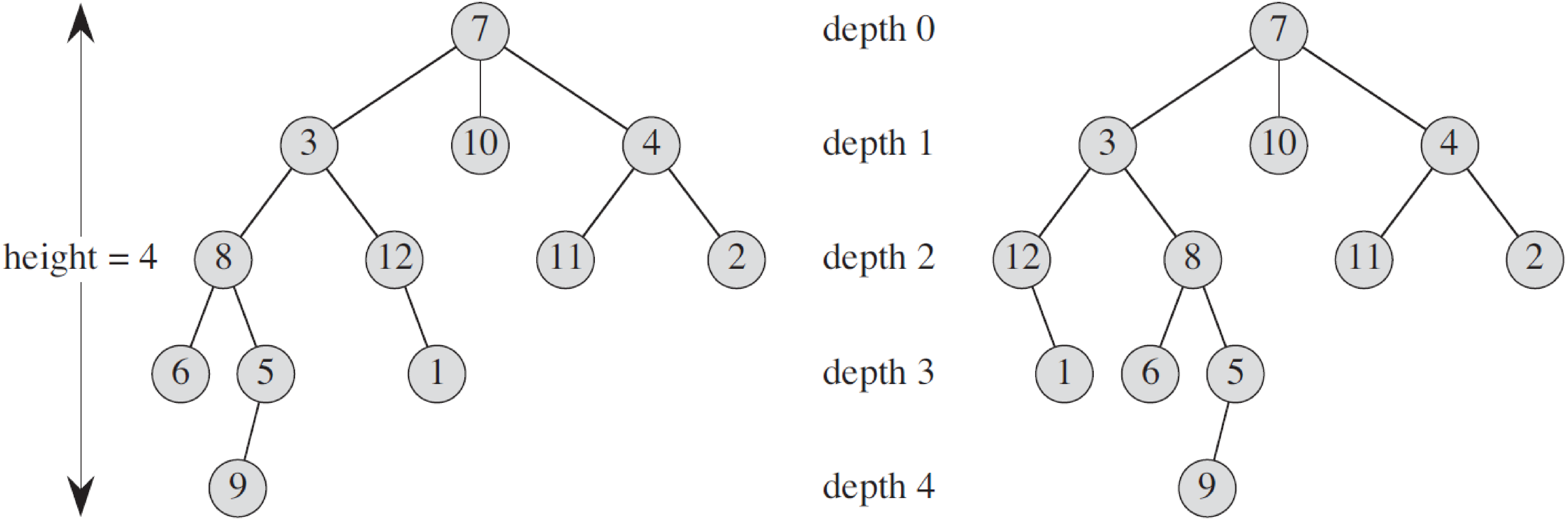
Rooted Trees

- A **rooted tree** is a free tree in which one of the vertices is distinguished from the others; it is called the **root**.
- A vertex in a rooted tree is called a **node**.
- Consider a node x in a rooted tree T with root r.
  - We call any node y on the unique simple path from r to x an **ancestor** of x; x is a **descendant** of y.
  - If the last edge on the simple path from r to x is (y, x), y is the **parent** of x and x is the **child** of y. The root is the only node with no parent.
- If two nodes have the same parents, they are **siblings**.

Rooted Trees

- A node with no children is a **leaf** (**external node**).
- A nonleaf node is an **internal node**.
- The **degree** of x is its number of children.
- The length of the simple path from r to x is the **depth** of x in T.
- A **level** of a tree consists of all the nodes at the same depth.
- The **height** of x is the number of edges on the longest simple downward path from x to a leaf. The height of the tree is the height of the root.
- An **ordered tree** is a rooted tree in which the children of each node are ordered.

Rooted Trees



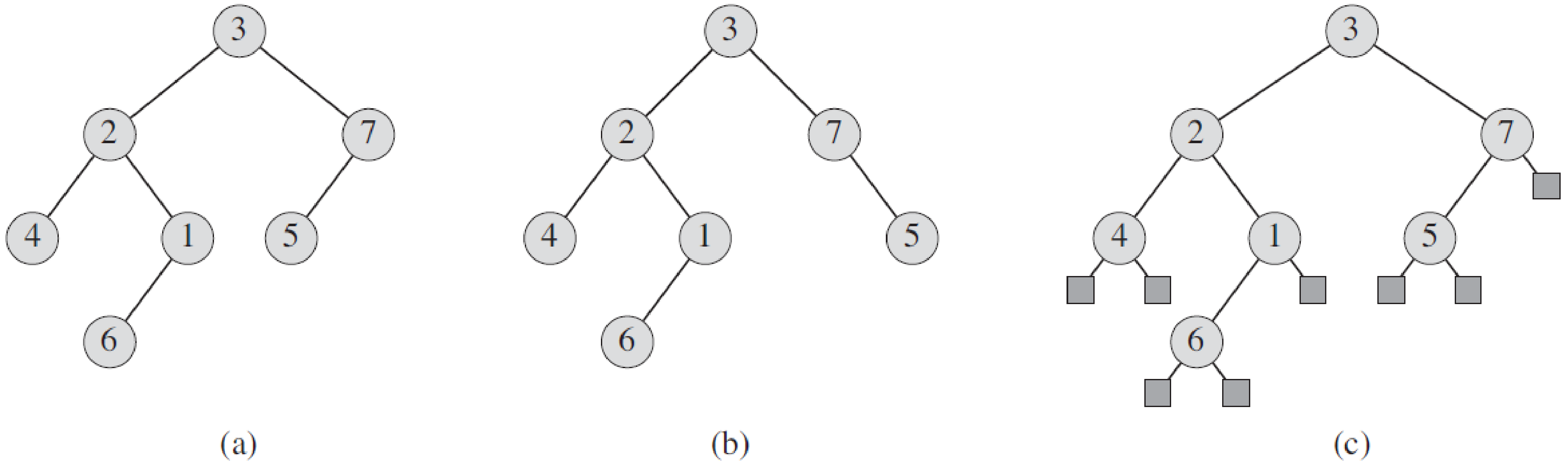
(a)
(b)

Binary Tree

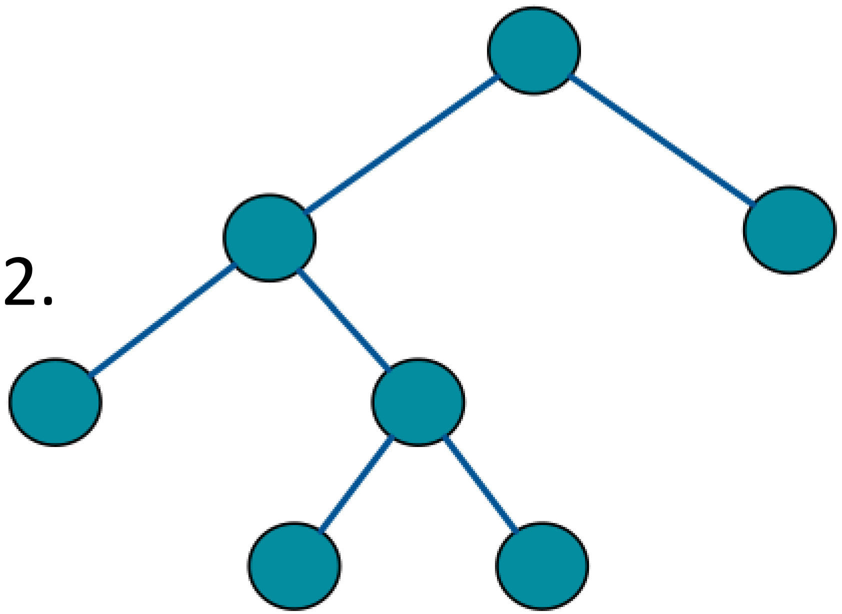
- A **binary tree** T is a structure defined on a finite set of nodes that either:
  - contains no nodes, or
  - is composed of three disjoint sets of nodes: a **root** node, a binary tree called its **left subtree**, and a binary tree called its **right subtree**.
- The binary tree that contains no nodes is called **empty tree** (**null tree**). It's often denoted by **NIL**.
- If the left (right) subtree is not empty, it's called **left** (**right**) **child**.
- If a subtree is the null tree, we say that the child is **absent** or **missing**.

Binary Tree

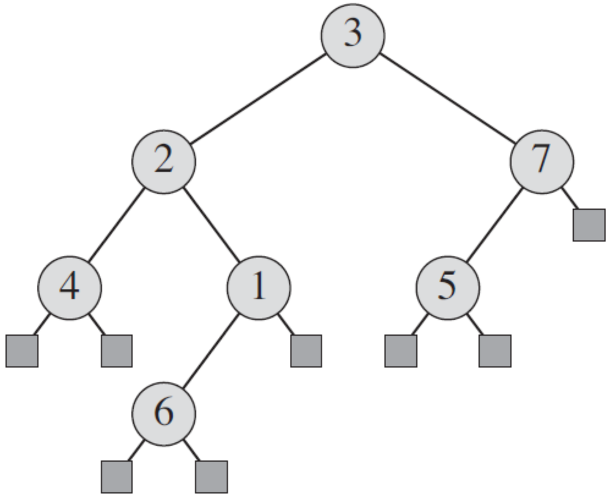
- It’s not simply an ordered tree where each node has degree at most 2: if a node has one child, the position of the child (left or right) matters!
- We can represent the positioning information in a binary tree by the internal nodes of an ordered tree.



Full Binary Tree

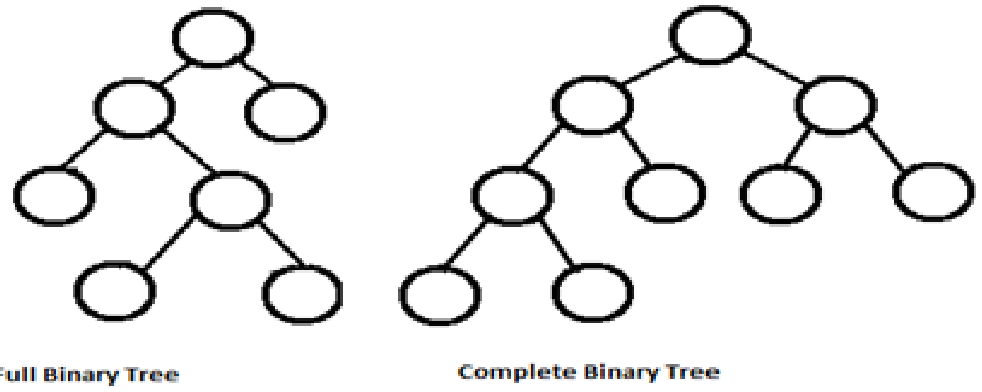


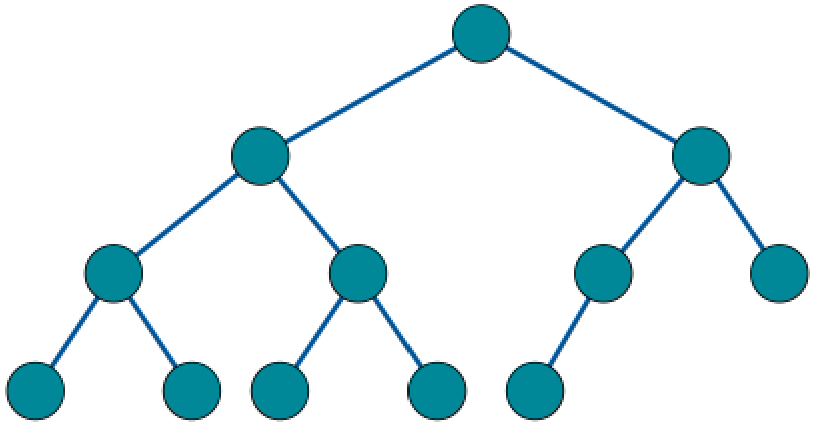
- Each node is either a leaf or has degree exactly 2.
- Because there is no degree-1 nodes, the order of the children of a node preserves positioning information.



Complete Binary Tree

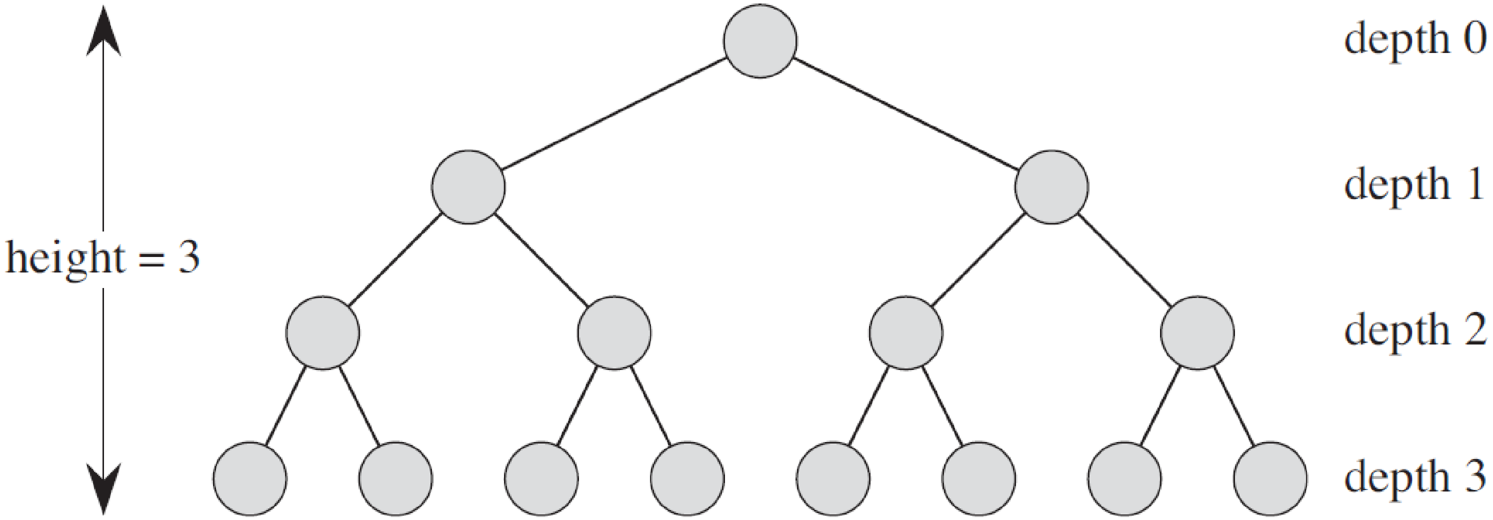
- Every level, except possibly the last, is completely filled and all the nodes in the last level are as far left as possible.





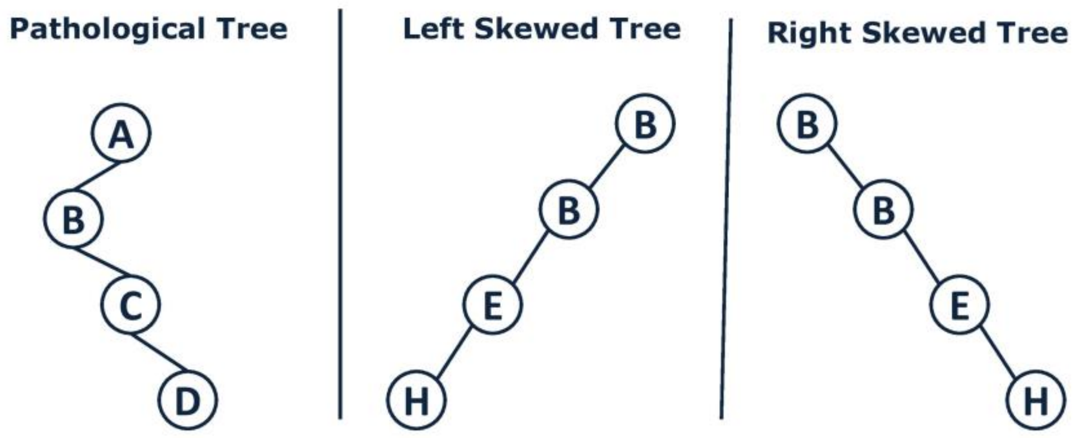
Perfect Binary Tree

- All internal nodes have degree two and all leaves have the same depth or (same level).
- Its number of internal nodes is 2^h-1, where h is its height. Why?



Degenerate (Pathological) Binary Tree

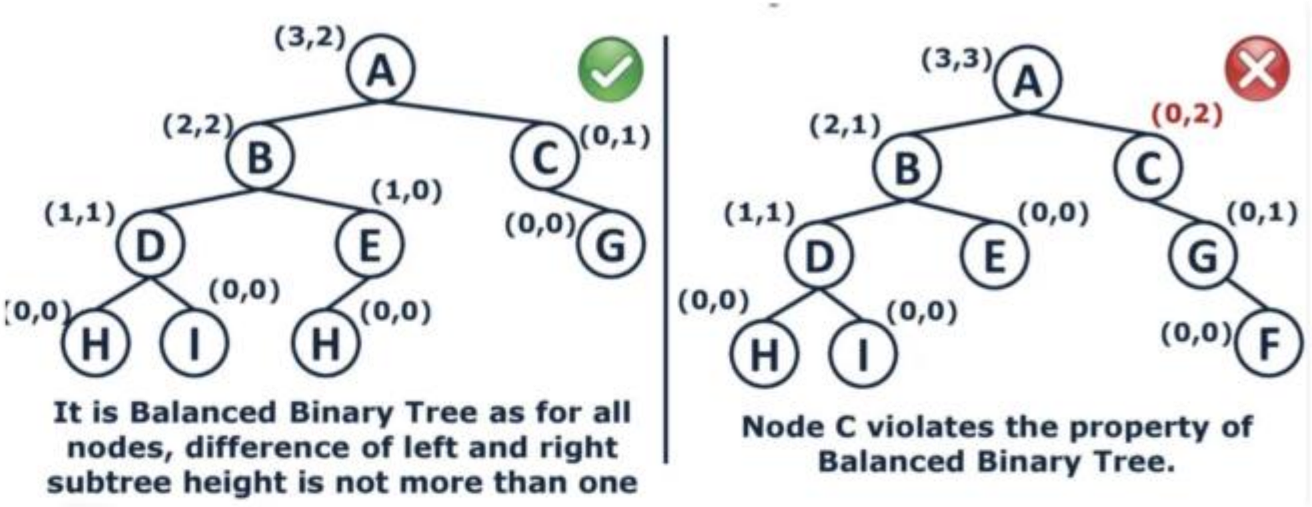
- In a **degenerate tree**, each parent node has only one associated child node. It will behave like a linked list data structure.
- A **skewed binary tree** is a degenerate tree which is solely dominated by left or right child nodes. All skewed binary trees are degenerate but not all degenerate trees are skewed.



https://anilgrgkarma.medium.com/binary-tree-and-its-types-8f2373b40837 Source: TarunChawla Tech

Balanced Binary Tree

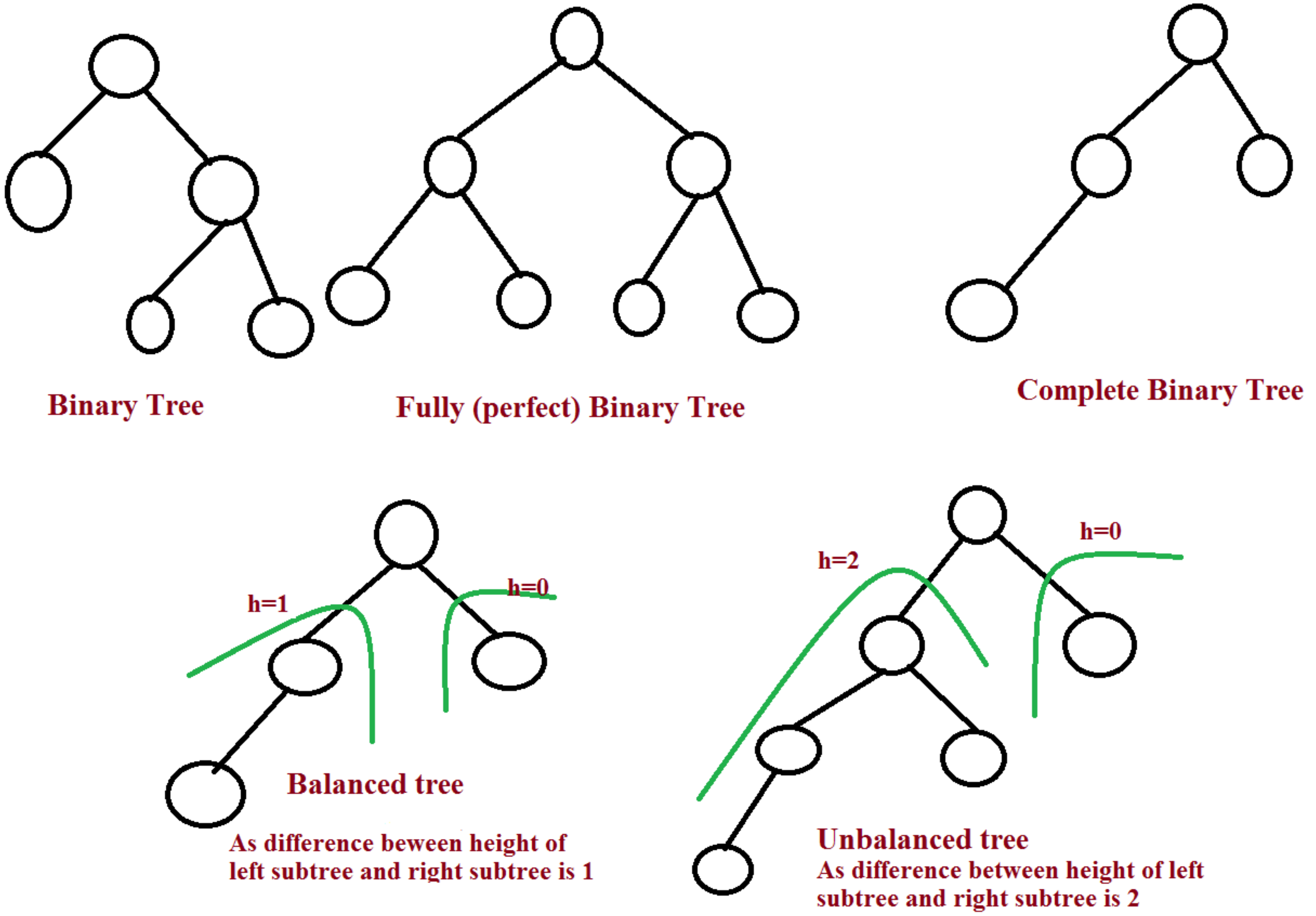
- The left and right subtrees of every node differ in height by no more than 1.



Balanced Binary Tree. (x,y): x is left sub tree height and y is right sub tree height.

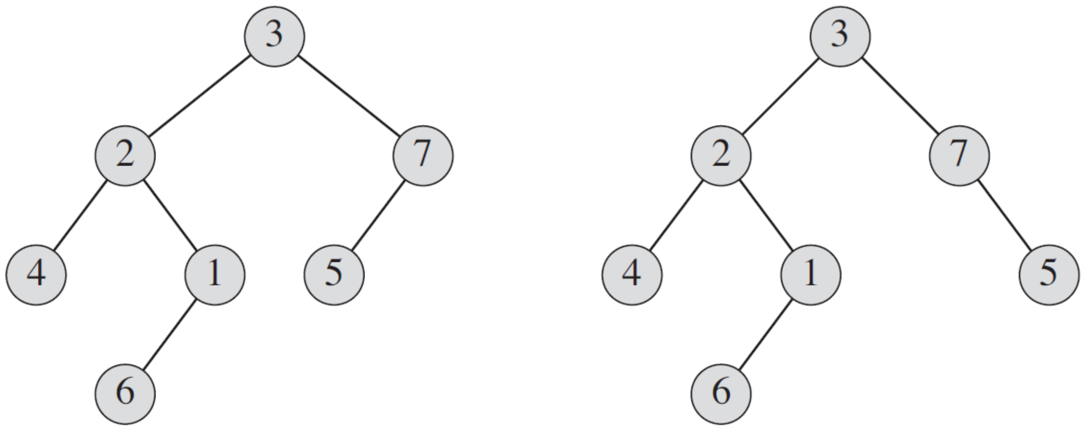
https://anilgrgkarma.medium.com/binary-tree-and-its-types-8f2373b40837 Source: TarunChawla Tech

Binary Tree Types



Positional Tree & k-ary Tree

- In a **positional tree**, the children of a node are labeled with distinct positive integers. The i-th child of a node is absent if no child is labeled with integer i.
- A **k-ary tree** is a positional tree in which, for every node, all children with labels greater than k are missing. Example: binary tree.



(a) (b)

Positional Tree & k-ary Tree

- A **perfect k-ary tree** is a k-ary tree in which all leaves have the same depth and all the internal nodes have degree k.
- The number of internal nodes of a perfect k-ary tree is

$$
\begin{aligned}
1 + k + k^2 + \cdots + k^{h-1} &= \sum_{i=0}^{h-1} k^i \\
&= \frac{k^h - 1}{k - 1}
\end{aligned}
$$

- A perfect binary tree has 2^h -1 internal nodes.

Properties of a Binary Tree

1. Every binary tree with n nodes, $n>0$, has exactly $n-1$ edges.

   Proof: Each element (except the root) has one parent. There is exactly one edge between each child and its parent. Hence, there are $n-1$ edges.

2. The number of nodes at level i is $\leq 2^i$, $i \geq 0$.

   Proof (By induction on i)

   Basis: The root is the only node at level $i=0$ and $2^0=1$.

   Inductive hypothesis: The number of nodes at level $i=k-1$ is $\leq 2^{k-1}$.

   We need to prove that at level $i=k$, we have $\leq 2^k$ nodes.

   The number of nodes at level k is $\leq 2 * 2^{k-1} = 2^k$ because each node at level $k-1$ has at most 2 children.

Properties of a Binary Tree

3. A binary tree of height h, h ≥ 0, has at least h+1 and at most 2^{h+1} -1 nodes, i.e. h < n < 2^{h+1}.

Proof: Let n be the number of nodes. There must be at least one node at each level and there are h+1 levels of nodes. This is the case of a degenerate tree. Therefore, n ≥ h+1.

Because of P2, there are at most 2^i nodes at level i. Thus, the maximum number of nodes in the whole tree is:

$$n \leq \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$$

This is the number of nodes in a perfect binary tree of height h.

Properties of a Binary Tree

4. Let h be the height of a binary tree with n nodes, n ≥ 0.
   Then, $\lceil \lg(n+1) \rceil < h < n$.

   Proof: There must be at least one node at each level. Because there are h+1 levels, the height should not exceed n-1, i.e. the number of nodes of a degenerate tree.

   The minimum possible height occurs when all the levels are completely filled, i.e. a perfect binary tree. Because such tree has 2^{h+1} -1 nodes (P3),

$$n \leq 2^{h+1} - 1$$
$$n + 1 \leq 2^{h+1}$$
$$\lceil \lg(n+1) \rceil \leq h + 1$$
$$h \geq \lceil \lg(n+1) \rceil - 1$$
$$h > \lceil \lg(n+1) \rceil$$

Indices in a Complete Binary Tree

Let i be the index assigned to a node u of a complete binary tree. Then, $1 \leq i \leq n$, is

a) If i=1, then u is the root.
b) If 2i > n, then u has no left child. Otherwise, its left child is labeled as 2i.
c) If 2i+1 > n, then u has no right child. Otherwise, its right child has been labeled as 2i+1.
d) If i>1, the parent of i is $\lfloor i/2 \rfloor$.

```
PARENT(i)
1  return ⌊i/2⌋

LEFT(i)
1  return 2i

RIGHT(i)
1  return 2i + 1
```

Properties of a Complete Binary Tree

1. The number of internal nodes of a length-n complete binary tree is $\lfloor n/2 \rfloor$ and its number of leaves is $\lceil n/2 \rceil$.

   Proof: Because of the way indices are assigned, the parent with highest index is $\lfloor n/2 \rfloor$. Thus, the nodes with indices 1, 2, …, $\lfloor n/2 \rfloor$ are the internal nodes. Then, the number of leaves is $n - \left\lfloor \frac{n}{2} \right\rfloor = \left\lceil \frac{n}{2} \right\rceil$.

2. The height h of a length-n complete binary tree is $\lfloor \lg n \rfloor$.

   Proof:

   - A perfect binary tree of height h-1 has 2^h-1 nodes.
   - A perfect binary tree of height h has 2^{h+1}-1 nodes.
   - Then, a complete binary tree of height h has $2^h \le n < 2^{h+1}$ nodes.
   - Consequently, h ≤ lg n < h+1, i.e. $\lfloor \lg n \rfloor = h$.

Properties of a Complete Binary Tree

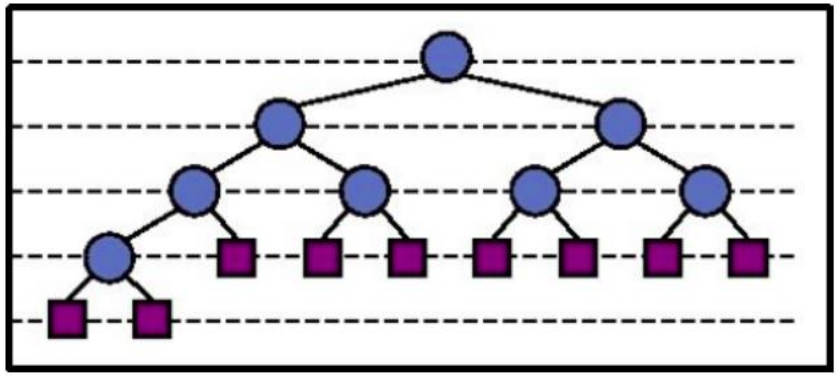
3. There are at most $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nodes of height h in a length-n tree.

   Proof (by induction)

   Basis: Because of P1, the number of leaves (h=0) is $\left\lceil \frac{n}{2} \right\rceil = \left\lceil \frac{n}{2^{0+1}} \right\rceil$.

   Inductive step: We assume it holds for h-1 and prove it holds for h.

   Even though all nodes of depth h have height 0, the nodes of depth h-1 can have height 0 or 1.



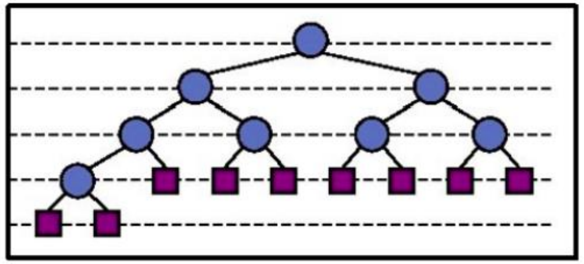
http://www.iitg.ac.in/psm/indexing ma252/y12/LectureNoteMA252Jan23.pdf

Properties of a Complete Binary Tree

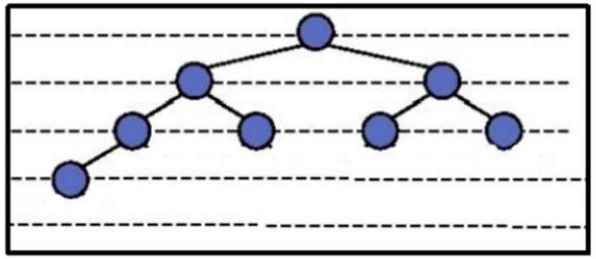
3. There are at most $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nodes of height h in a length-n tree.

Proof (by induction)

- Let T be the original tree and let T' be the tree resulting from removing the leaves of T.
- Let n (n') be the number of nodes in T (T').
- Let n_h (n_h') be the number of nodes of height h in T (T').
- Note that $n_h = n_{h-1}'$ and $n' = n - n_0 = n - \left\lceil \frac{n}{2} \right\rceil = \left\lfloor \frac{n}{2} \right\rfloor$.



T



T'

http://www.iitg.ac.in/psm/indexing ma252/y12/LectureNoteMA252Jan23.pdf

Properties of a Complete Binary Tree

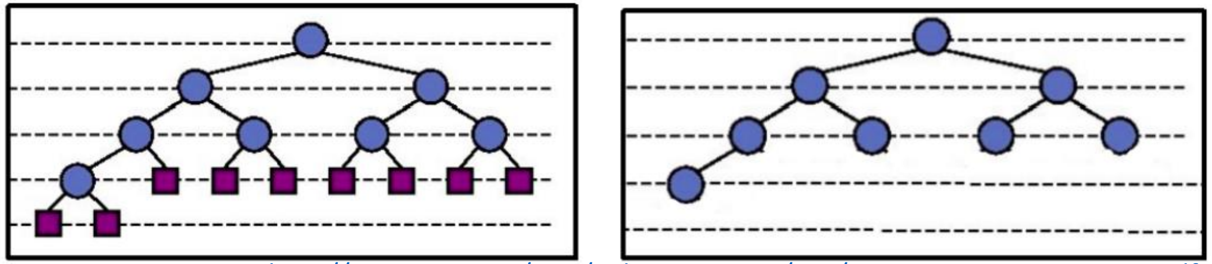
3. There are at most $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nodes of height h in a length-n tree.

Proof (by induction)

Recall that $n_h = n_{h-1}'$ and $n' = n - n_0 = n - \left\lceil \frac{n}{2} \right\rceil = \left\lfloor \frac{n}{2} \right\rfloor$.

Inductive hypothesis: A tree contains at most $\left\lceil \frac{n}{2^{(h-1)+1}} \right\rceil = \left\lceil \frac{n}{2^h} \right\rceil$ nodes of height h-1.

$$n_h = n_{h-1}' = \left\lceil \frac{n'}{2^h} \right\rceil = \left\lceil \frac{\lfloor n/2 \rfloor}{2^h} \right\rceil < \left\lceil \frac{n/2}{2^h} \right\rceil = \left\lceil \frac{n}{2^{h+1}} \right\rceil.$$



http://www.iitg.ac.in/psm/indexing ma252/y12/LectureNoteMA252Jan23.pdf

T T'

Properties of a Complete Binary Tree

4. A length-n complete binary tree with its last level half full has at most 2n/3 nodes in its left subtree. It is the greatest unbalance such tree can have.

Proof: Let h be the height of such tree, and n_L (n_R) be the number of nodes in the left (right) subtree. Then,

$$n_L = \sum_{i=0}^{h-1} 2^i = 2^h - 1 = 2.2^{h-1} - 1\,. \qquad n_R = \sum_{i=0}^{h-2} 2^i = 2^{h-1} - 1.$$

$$n = n_L + n_R + 1 = 3.\,2^{h-1} - 1$$

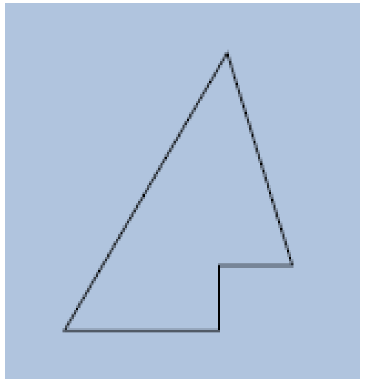
$$\frac{n_L}{n} = \frac{2.2^{h-1} - 1}{3.2^{h-1} - 1} < \frac{2}{3}.$$

Presentation made by Juan Mendivelso. Contents and figures extracted from the book: Introduction to Algorithms, Third Edition. Cormen, Leiserson, Rivests and Stein. The MIT Press. 2009.

2. HEAPS & HEAPSORT

Max Heaps & Min Heaps

- A **max heap** is a complete binary tree such that for every node i>1, A[Parent(i)] ≥ A[i].
- A **min heap** is a complete binary tree such that for every node i>1, A[Parent(i)] ≤ A[i].
- It's optimal to represent heaps as arrays.



Material Curso de Algoritmos – Julio López

PARENT(i)

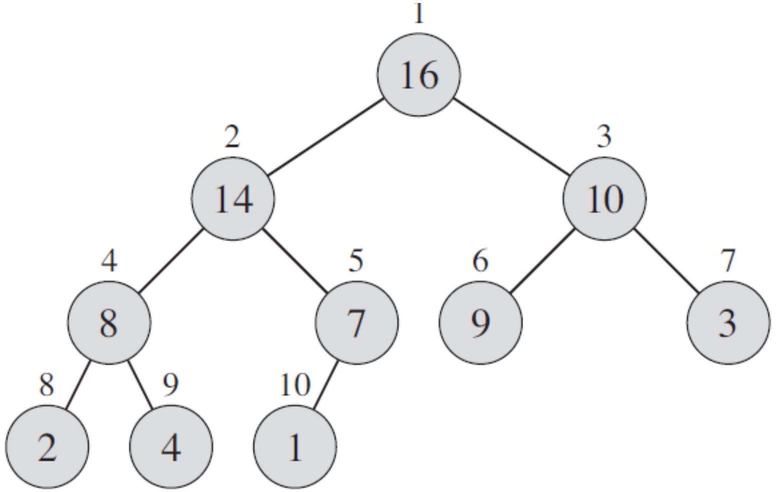
1 **return** $\lfloor i/2 \rfloor$

LEFT(i)

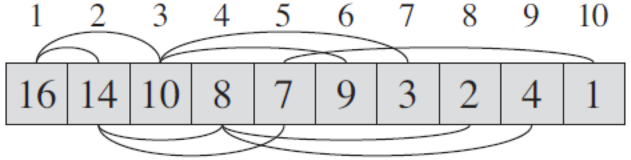
1 **return** $2i$

RIGHT(i)

1 **return** $2i + 1$



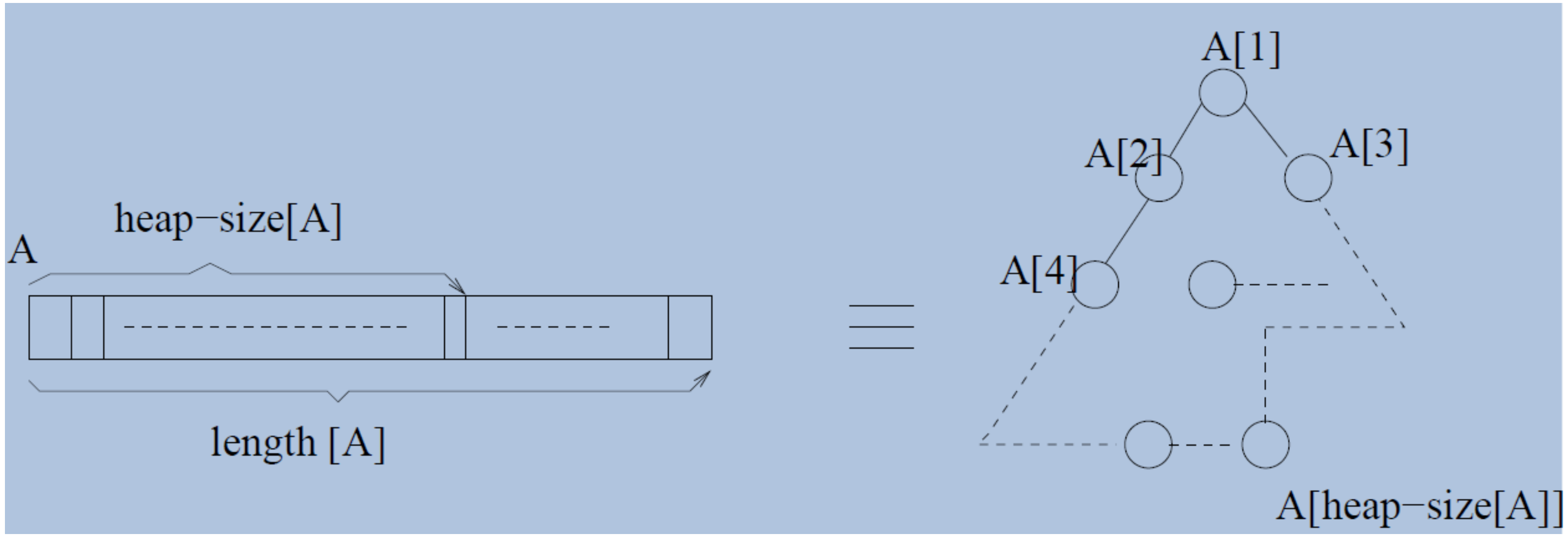
(a)



(b)

Max Heaps Representation

- Array A.
- A.length: size of the array.
- heapsize number of the current elements in the heap.
- Particularly, the heap is stored in A[1..heapsize].

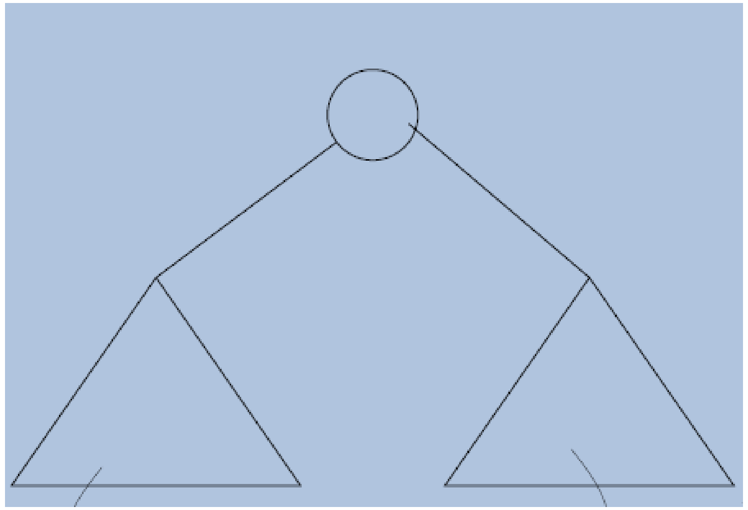


Important Routines for Max-Heaps

- Max-Heapify(A,i): O(lg n).
- Build-Heap(A): O(n).
- Heapsort: O(n lg n).
- Priority Queue Routines (O (lg n)):
  - Max-Heap-Insert()
  - Max-Heap-Extract-Max()
  - Max-Heap-Increase-Key()
  - Max-Heap-Maximum()

Max-Heapify

- Preconditions: The subtrees Left(i) and Right(i) are max heaps.
- Postconditions: The tree with root at i is a maxheap.



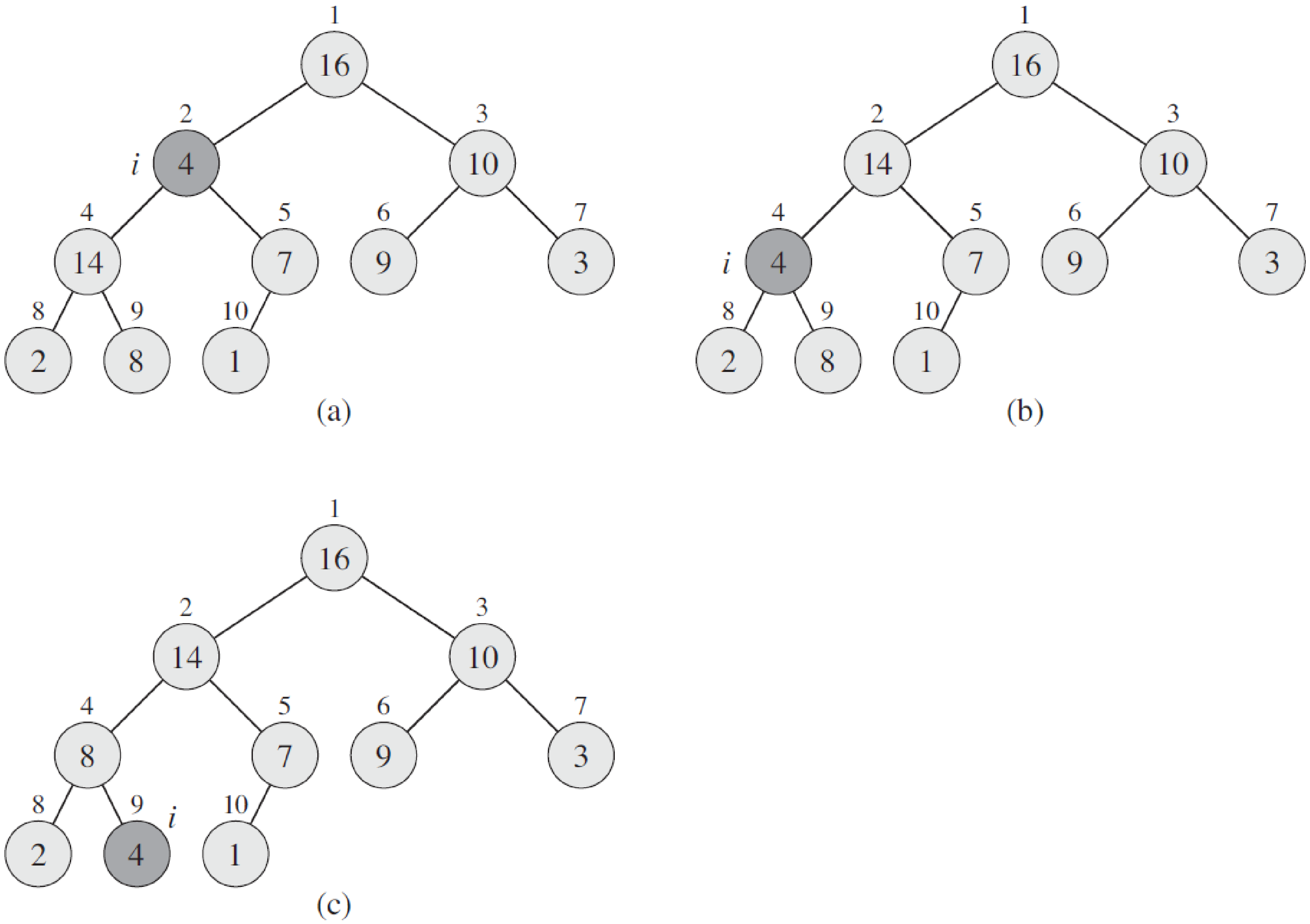
Material Curso de Algoritmos – Julio López

```
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)
```

Max-Heapify

- Max-Heapify(A,2)

```
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)
```



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)
```

- In the worst case, the max-heap is a complete binary tree with its last level half full, i.e. the most unbalanced a complete binary tree can get.
- Because of the corresponding property:

$$T(n) \leq T(2n/3) + \Theta(1)$$

$$T(n) = O(\lg n)$$

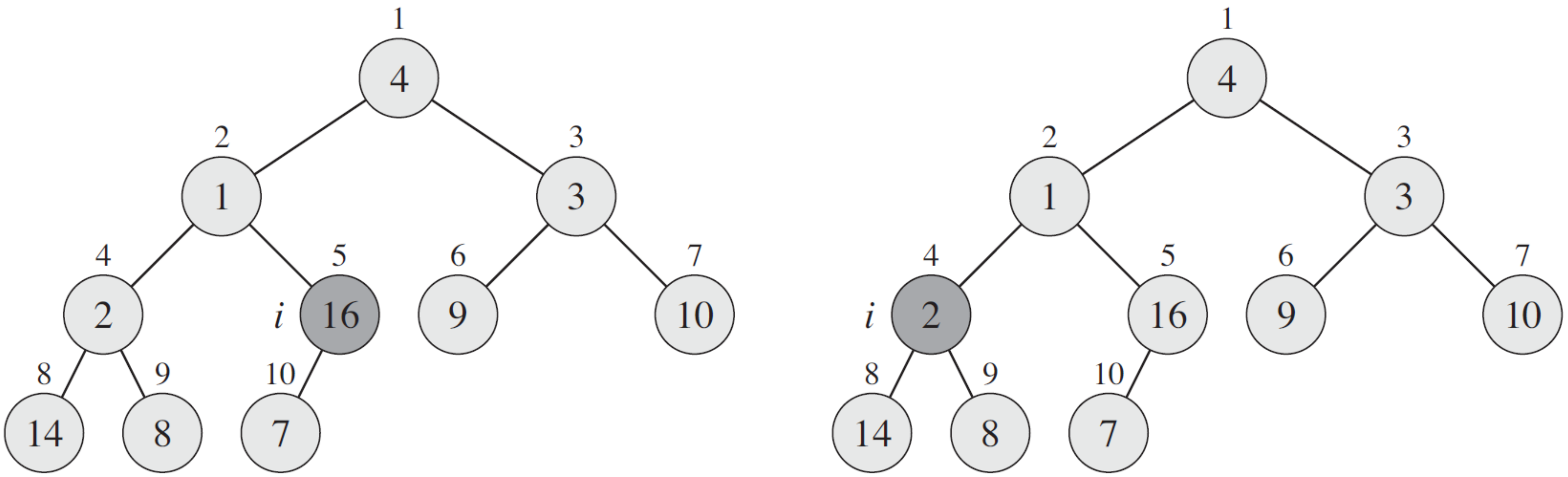
Build-Heap

BUILD-MAX-HEAP(A)

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

At the start of each iteration of the **for** loop of lines 2–3, each node $i + 1, i + 2, \ldots, n$ is the root of a max-heap.

A									
4	1	3	2	16	9	10	14	8	7



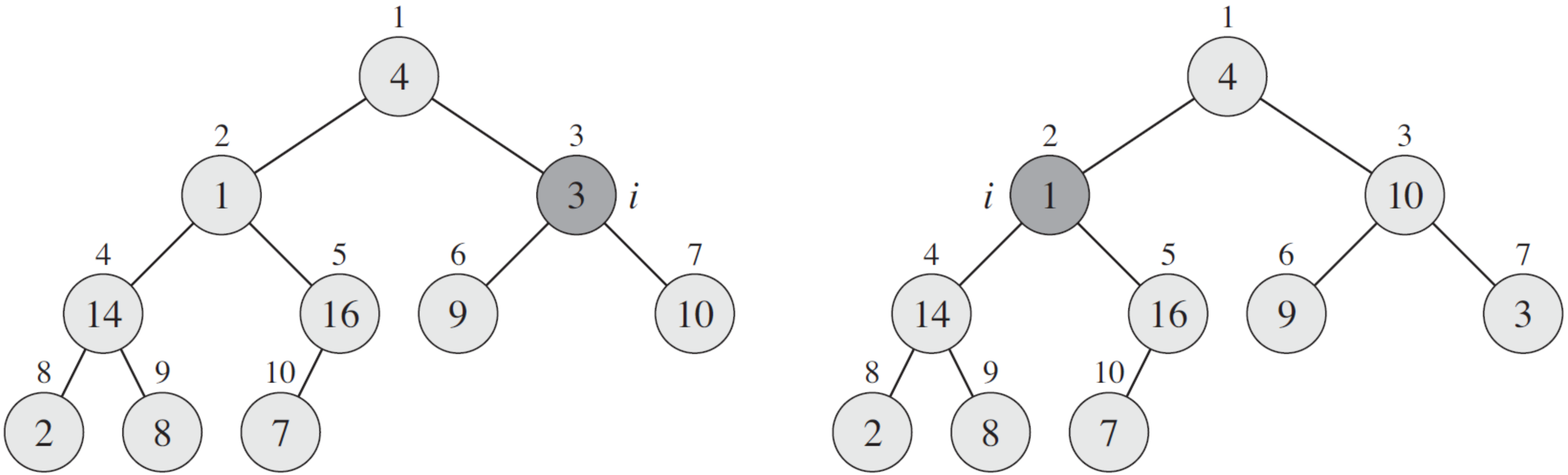
(a) (b)

Build-Heap

BUILD-MAX-HEAP(A)

1 $A.heap\text{-}size = A.length$
2 **for** $i = \lfloor A.length/2 \rfloor$ **downto** 1
3   MAX-HEAPIFY(A, i)

At the start of each iteration of the **for** loop of lines 2–3, each node $i + 1$, $i + 2, \ldots, n$ is the root of a max-heap.



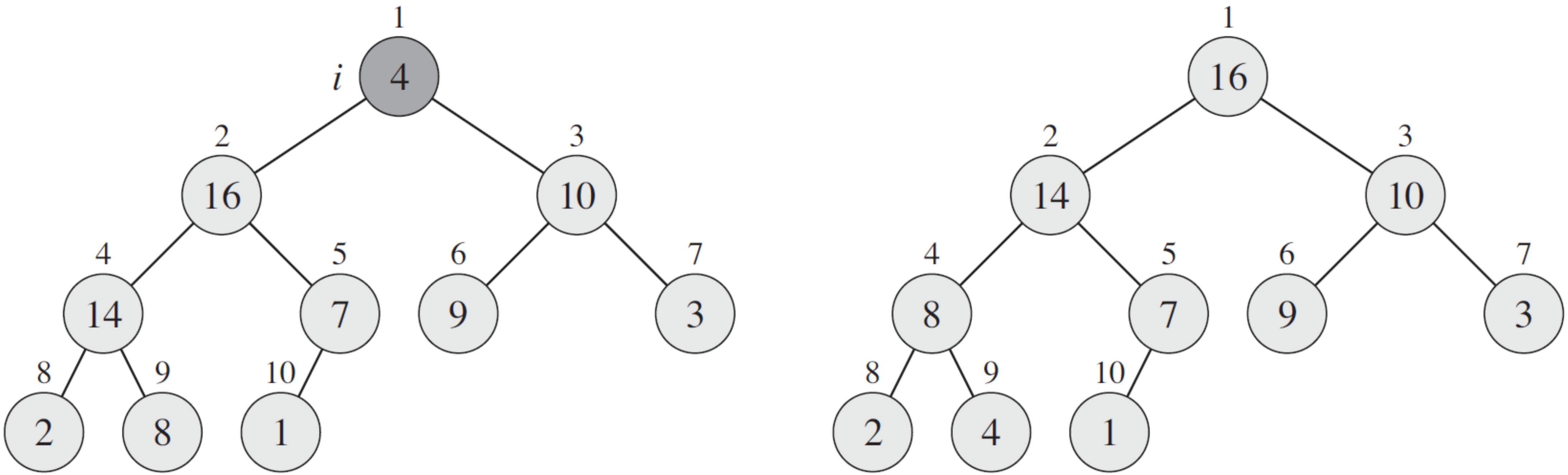
(c) (d)

Build-Heap

BUILD-MAX-HEAP(A)

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

At the start of each iteration of the **for** loop of lines 2–3, each node $i + 1, i + 2, \ldots, n$ is the root of a max-heap.



(e) (f)

Build-Heap Complexity

BUILD-MAX-HEAP(A)

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

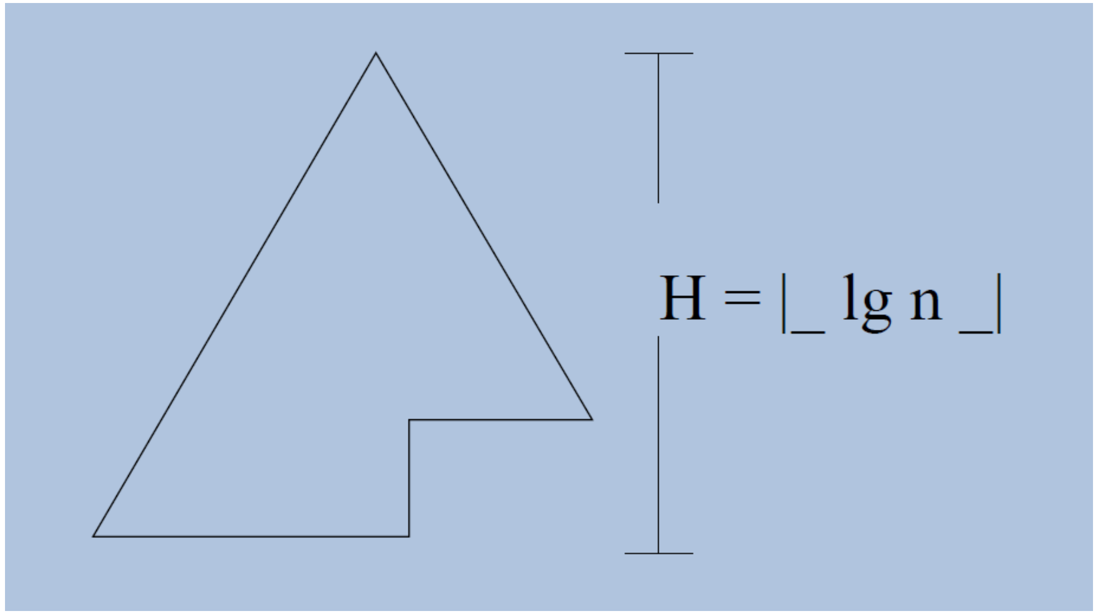
- The complexity of Max-Heapify(A,i) is O(h) where h is the height of node i.
- Because, the largest height, the root's height, is lg n, each call to Max-Heapify is O(lg n).
- There are n/2 calls of this procedure.
- Thus, Build-Heap is O(n lg n).
- However, we can find a tighter upper bound.

Build-Heap Complexity

BUILD-MAX-HEAP(A)

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

- If the heap has n nodes,



- The number of nodes of a given height h is at most $\left\lceil \frac{n}{2^{h+1}} \right\rceil$.

Build-Heap Complexity

BUILD-MAX-HEAP(A)

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

- So we can sum the complexity of each possible of the height h multiplied by the number of nodes of such height, i.e. $\left\lceil \frac{n}{2^{h+1}} \right\rceil$:

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = O(n)\,.$$

- This is because

$$\sum_{k=0}^{\infty} k x^k = \frac{x}{(1-x)^2}$$

for $|x| < 1$.

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2\,.$$

Presentation made by Juan Mendivelso. Contents and figures extracted from the book: Introduction to Algorithms, Third Edition. Cormen, Leiserson, Rivests and Stein. The MIT Press. 2009.

Heapsort

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

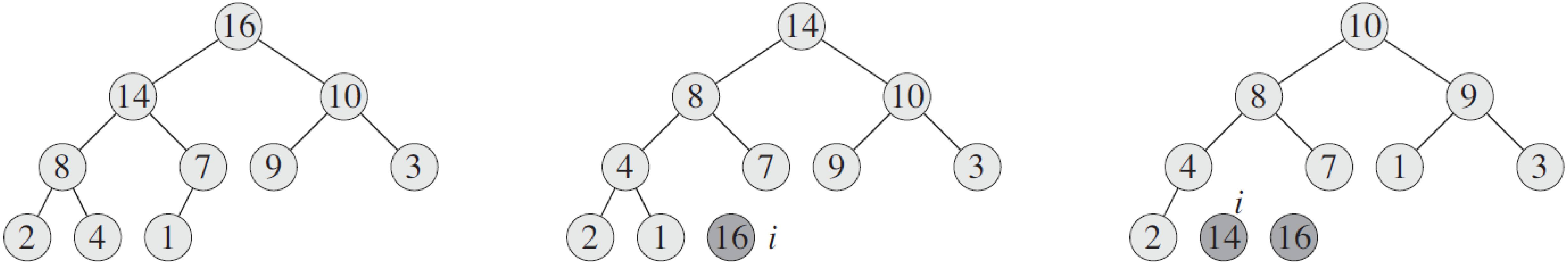
At the start of each iteration of the **for** loop of lines 2–5, the subarray $A[1..i]$ is a max-heap containing the i smallest elements of $A[1..n]$, and the subarray $A[i+1..n]$ contains the $n-i$ largest elements of $A[1..n]$, sorted.

Presentation made by Juan Mendivelso. Contents and figures extracted from the book: Introduction to Algorithms, Third Edition. Cormen, Leiserson, Rivests and Stein. The MIT Press. 2009.

Heapsort

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

At the start of each iteration of the **for** loop of lines 2–5, the subarray $A[1 \mathinner{..} i]$ is a max-heap containing the i smallest elements of $A[1 \mathinner{..} n]$, and the subarray $A[i+1 \mathinner{..} n]$ contains the $n-i$ largest elements of $A[1 \mathinner{..} n]$, sorted.



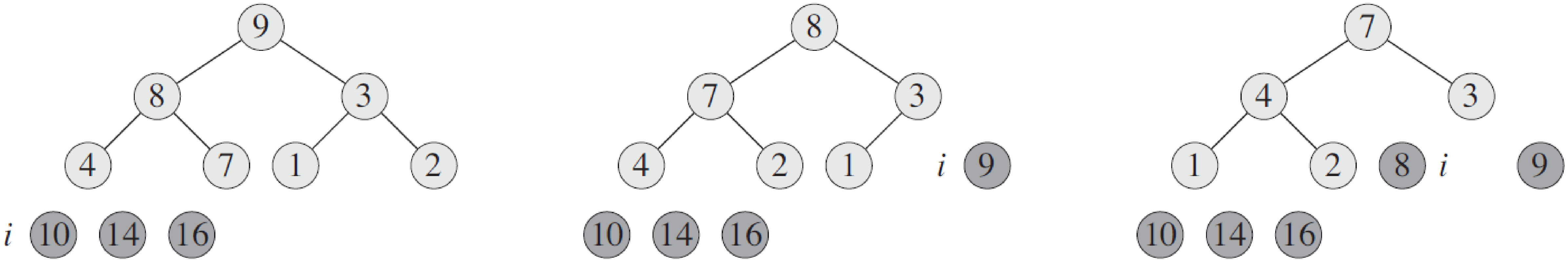
(a) (b) (c)

Presentation made by Juan Mendivelso. Contents and figures extracted from the book: Introduction to Algorithms, Third Edition. Cormen, Leiserson, Rivests and Stein. The MIT Press. 2009.

Heapsort

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

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.



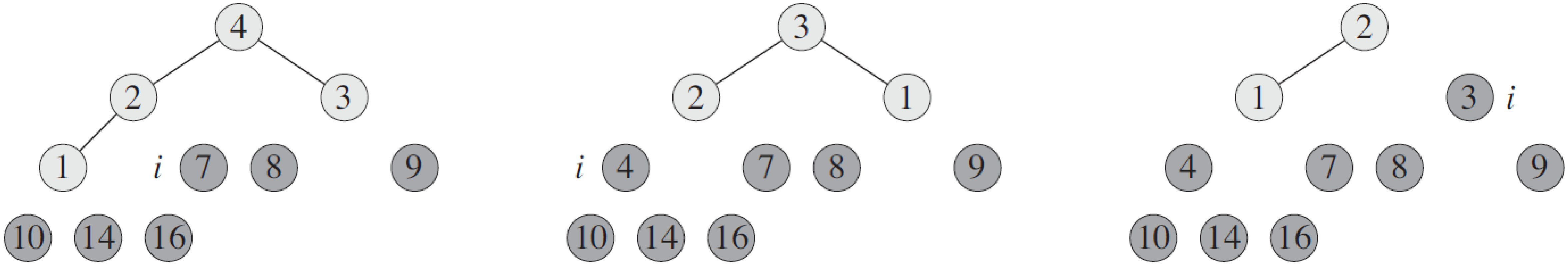
(d) (e) (f)

Presentation made by Juan Mendivelso. Contents and figures extracted from the book: Introduction to Algorithms, Third Edition. Cormen, Leiserson, Rivests and Stein. The MIT Press. 2009.

Heapsort

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

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.

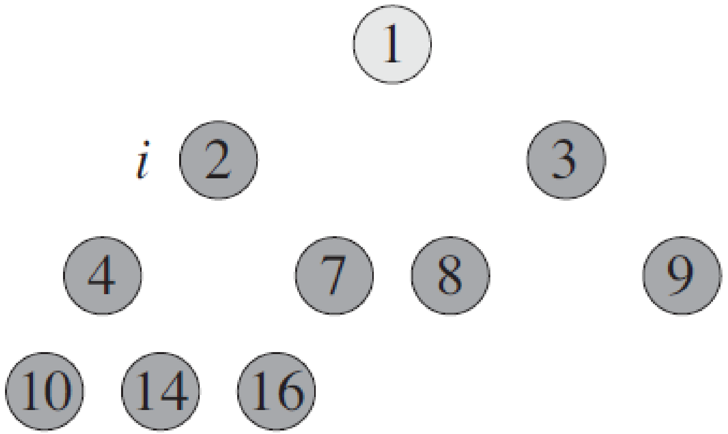


(g) (h) (i)

Heapsort

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

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.



(j)

A	1	2	3	4	7	8	9	10	14	16

(k)

Heapsort Complexity

$$\underbrace{\mathcal{O}(n)}_{\text{construir}} + (n-1)\underbrace{\mathcal{O}(\lg n)}_{Ordenar} = \underbrace{\mathcal{O}(n \lg n)}_{\text{HeapSort}}$$

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

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.

3. PRIORITY QUEUES

Priority Queues

- Heapsort is excellent but Quicksort is the usual choice because of its practical efficiency.
- However, heaps are very useful to implement priority queues.
- A **priority queue** is a data structure that allows the insertion and deletion of elements regarding priorities assigned to them.
- When an element is inserted, its priority is established. However, it can be updated later.
- The element that is deleted is the one with highest (lowest) priority in a **max-priority queue** (**min-priority queue**). Therefore, max heaps (**min-heaps**) are a good choice for this operation.

Operations for Max-Priority Queues

- Insert(S,x): Insert element x in the priority queue S.
- Maximum(S): It returns the element of S with the largest key.
- Extract-Max(S): It removes and returns the element of S with the largest key.
- Increase-Key(S,x,k): It increases the value of element x's key to the new value k, which is assumed to be at least as large as x's current key value.

Presentation made by Juan Mendivelso. Contents and figures extracted from the book: Introduction to Algorithms, Third Edition. Cormen, Leiserson, Rivests and Stein. The MIT Press. 2009.

Operations for Min-Priority Queues

- Insert(S,x): Insert element x in the priority queue S.
- Minimum(S): It returns the element of S with the smallest key.
- Extract-Min(S): It removes and returns the element of S with the smallest key.
- Decrease-Key(S,x,k): It increases the value of element x's key to the new value k, which is assumed to be at most as large as x's current key value.

Applications of Priority Queues

- Schedule jobs on a shared computer.
- Event-Driven Simulation.
- Agenda administration.

They allow to store a handle to the corresponding application object in each heap element.

Applications of Priority Queues

HEAP-MAXIMUM(A)

```
return A[1]
```

HEAP-EXTRACT-MAX(A)

```
if A.heap-size < 1
    error "heap underflow"
max = A[1]
A[1] = A[A.heap-size]
A.heap-size = A.heap-size - 1
MAX-HEAPIFY(A, 1)
return max
```

Complexity?

Applications of Priority Queues

```
HEAP-INCREASE-KEY(A, i, key)
if key < A[i]
    error "new key is smaller than current key"
A[i] = key
while i > 1 and A[PARENT(i)] < A[i]
    exchange A[i] with A[PARENT(i)]
    i = PARENT(i)
```

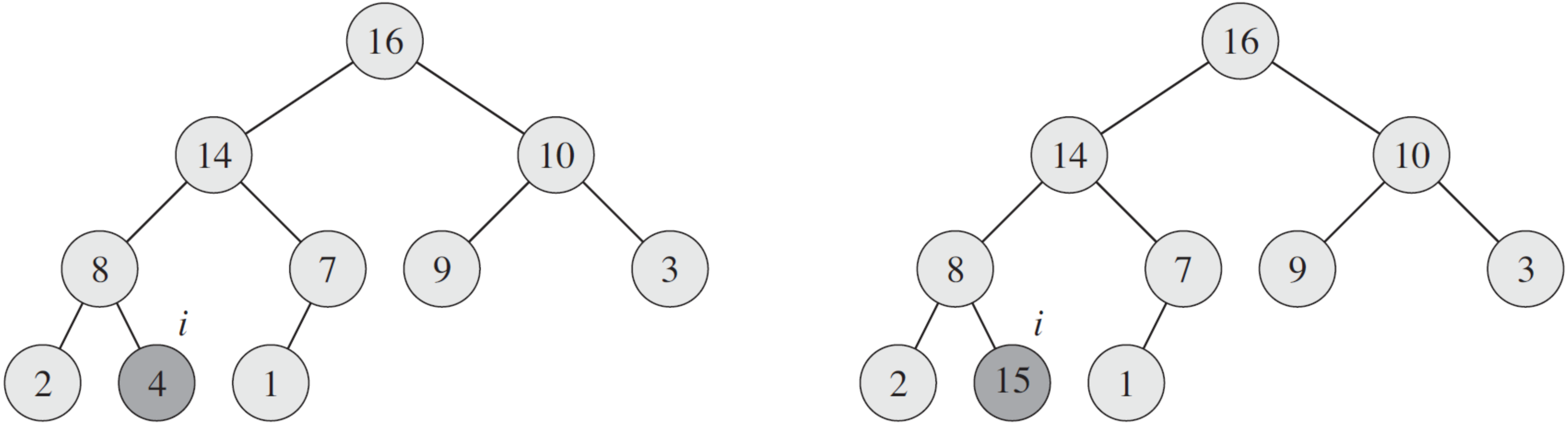
Applications of Priority Queues

```
HEAP-INCREASE-KEY(A, i, key)
if key < A[i]
    error "new key is smaller than current key"
A[i] = key
while i > 1 and A[PARENT(i)] < A[i]
    exchange A[i] with A[PARENT(i)]
    i = PARENT(i)
```

```
MAX-HEAP-INSERT(A, key)
A.heap-size = A.heap-size + 1
A[A.heap-size] = -∞
HEAP-INCREASE-KEY(A, A.heap-size, key)
```

Applications of Priority Queues

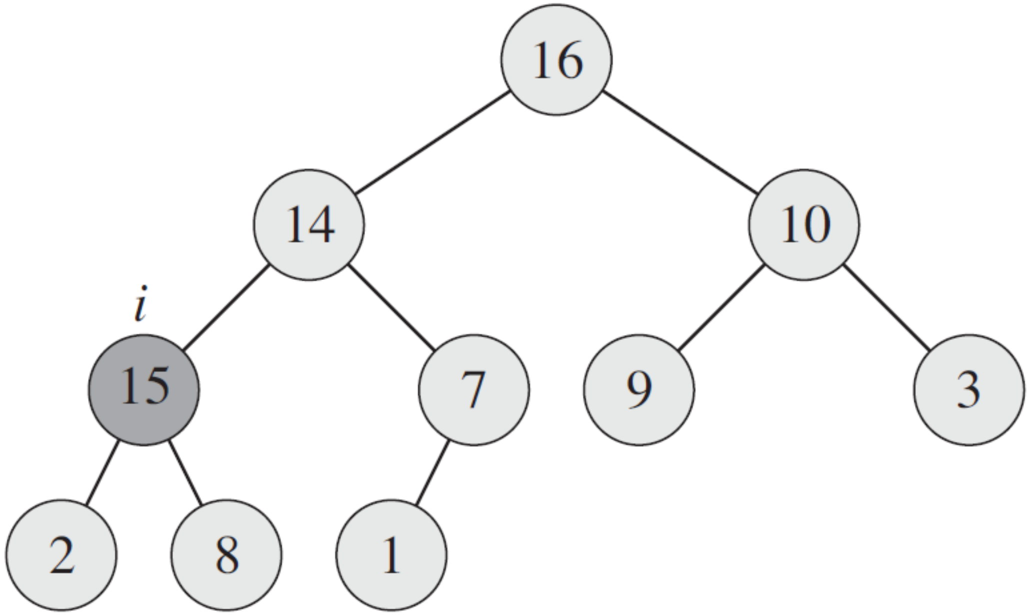
```
HEAP-INCREASE-KEY(A, i, key)
if key < A[i]
    error "new key is smaller than current key"
A[i] = key
while i > 1 and A[PARENT(i)] < A[i]
    exchange A[i] with A[PARENT(i)]
    i = PARENT(i)
```



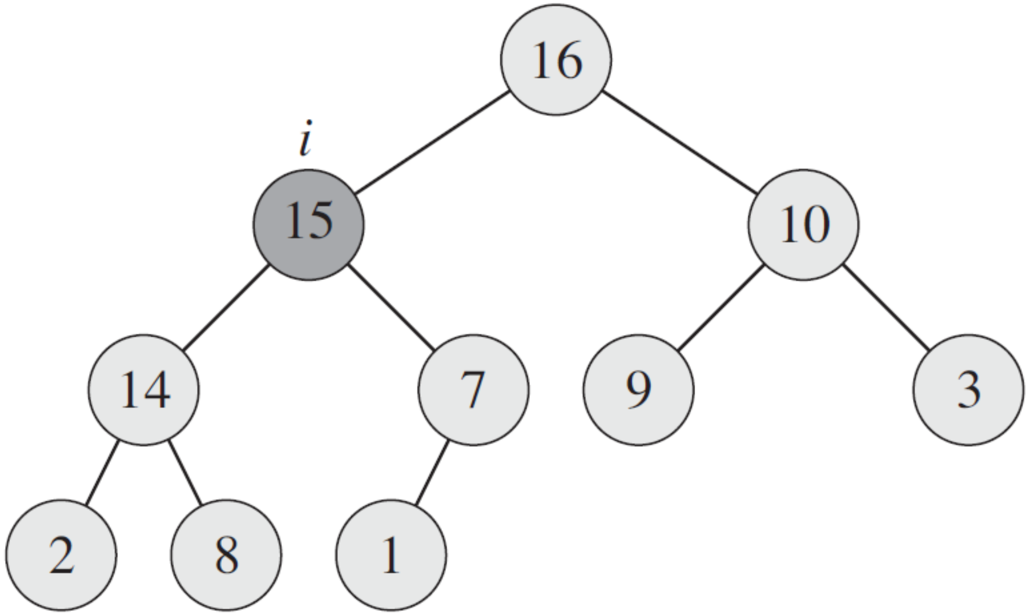
(a) (b)

Applications of Priority Queues

```
HEAP-INCREASE-KEY(A, i, key)
if key < A[i]
    error "new key is smaller than current key"
A[i] = key
while i > 1 and A[PARENT(i)] < A[i]
    exchange A[i] with A[PARENT(i)]
    i = PARENT(i)
```



(c)



(d)

BIBLIOGRAPHY

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Introduction to Algorithms, Third Edition. The MIT Press. 2009.
- https://codepumpkin.com/binary-tree-types-introduction/#FullBinaryTree
- https://anilgrgkarma.medium.com/binary-tree-and-its-types-8f2373b40837
- http://www.iitg.ac.in/psm/indexing ma252/y12/LectureNoteMA252Jan23.pdf
- https://www2.cs.sfu.ca/CourseCentral/307/petra/2009/SLN 2.pdf
- Images of Julio Cesar Lopez.