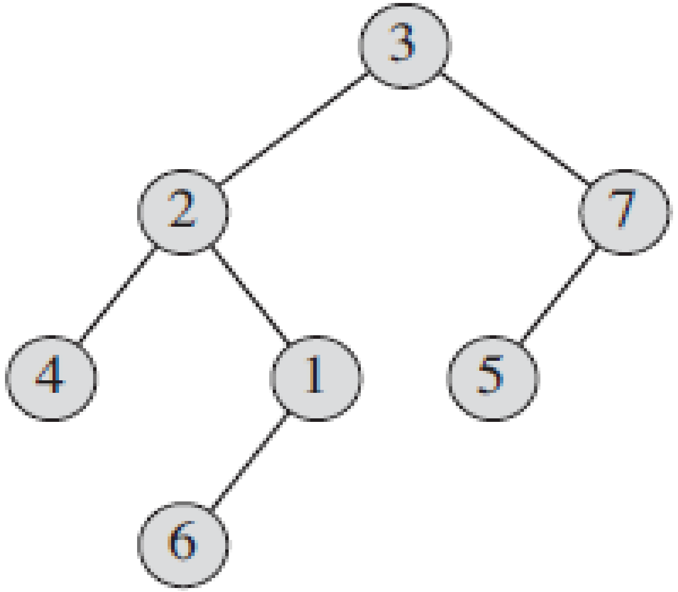
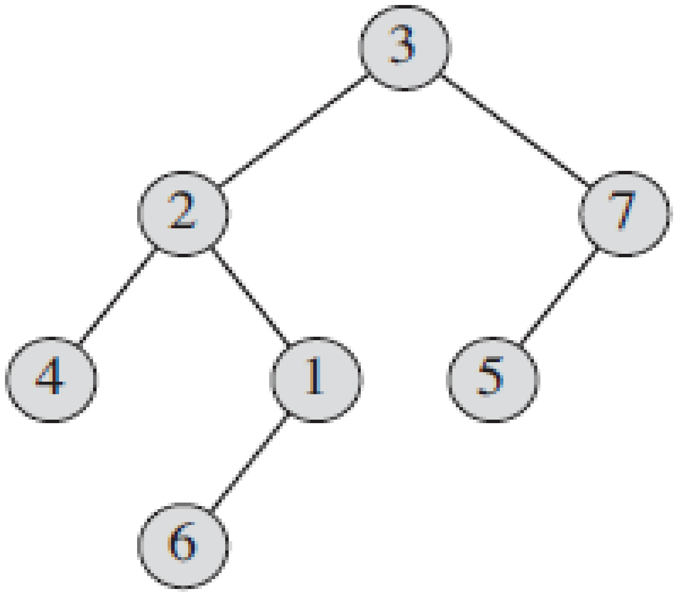
Trees – degree of a node

- The *degree* **of a node** x **is the number of children** of x
  - **Degree of node 3:** **2**
  - **Degree of node 7:** **1**
  - **Degree of node 6:** **0**



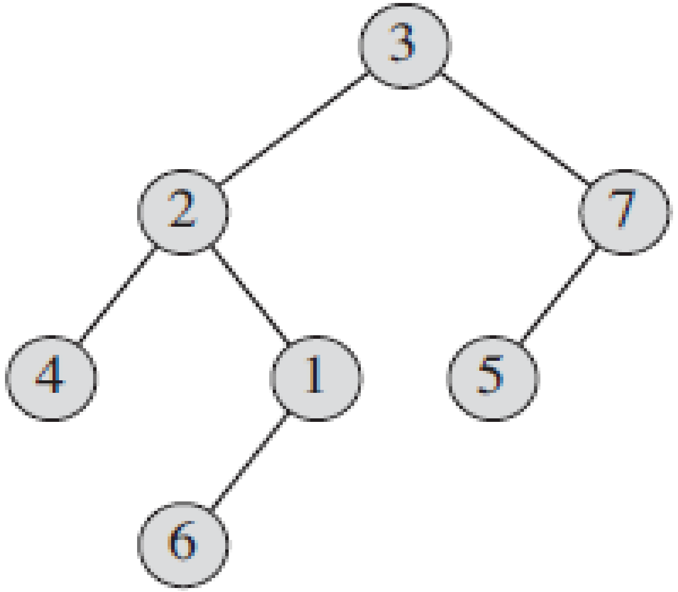
Trees – Depth of a node

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



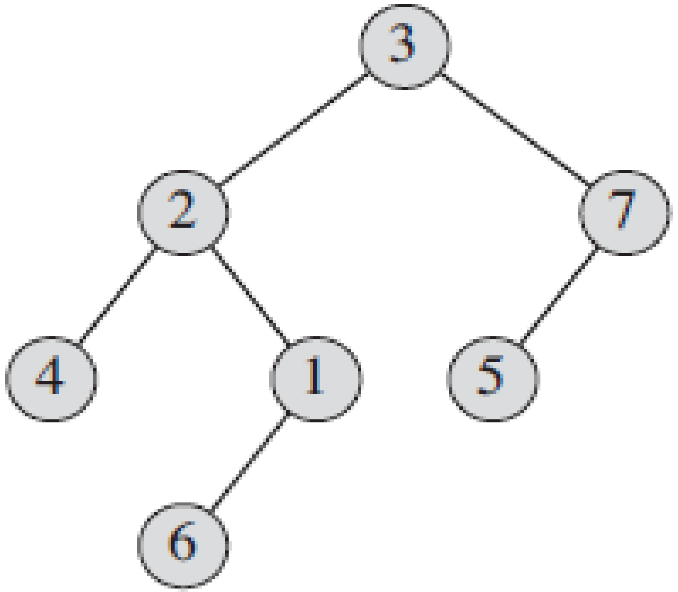
Trees – Levels

- A level of a tree consists of all nodes at the same depth.
  - Root node at level 0
  - Nodes at level 2: 4, 1, 5
  - Nodes at levels 1 ?
  - Nodes at levels 3 ?



Trees – Height of a node

- **Height of a node**
  - 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
  - Height of the tree shown is 3 = length of the path from root to node the node labeled 6



Complete Binary Trees

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

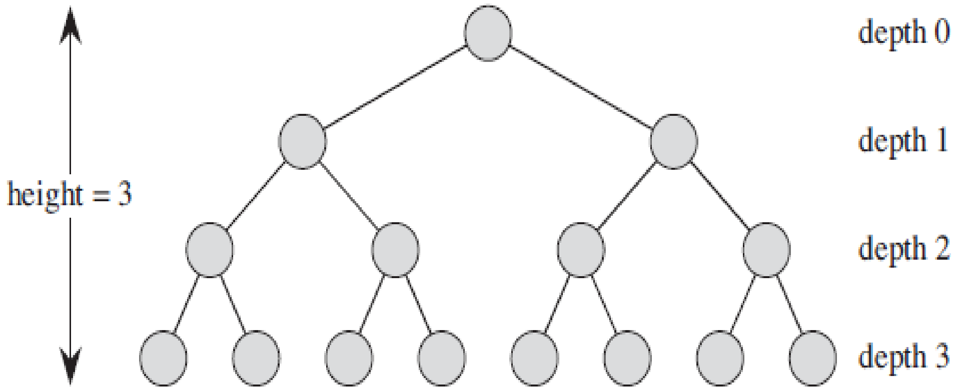
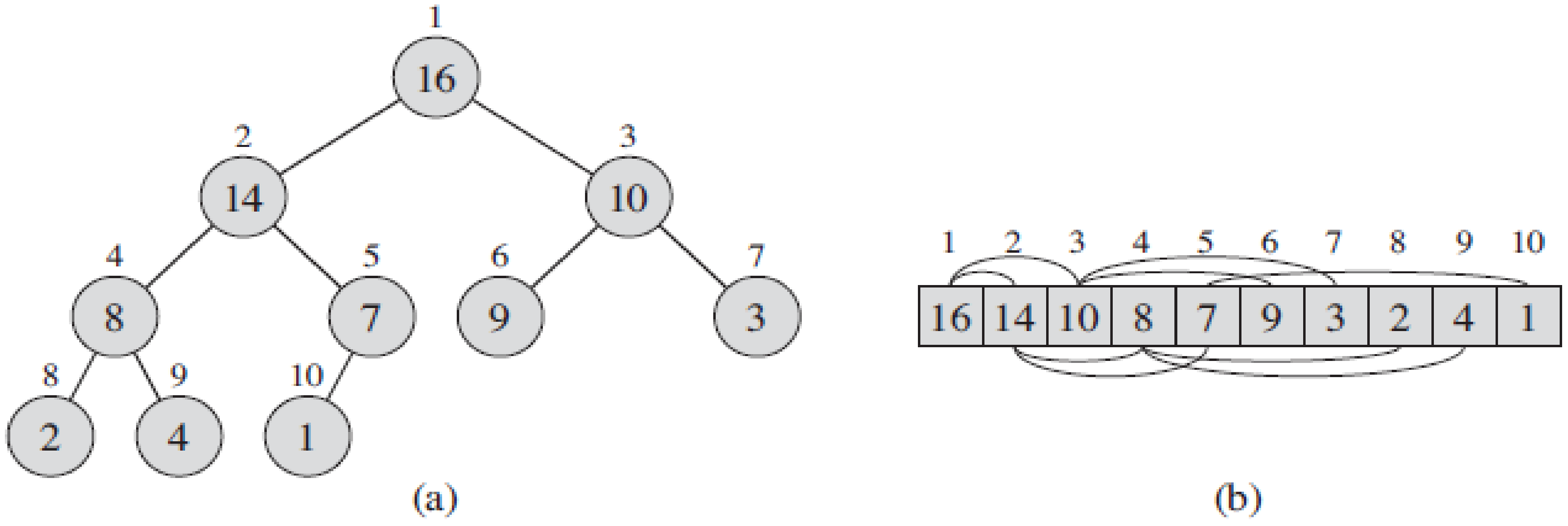


Figure B.8 A complete binary tree of height 3 with 8 leaves and 7 internal nodes.

Nearly Complete Binary Trees

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



Binary heap

- Nearly Complete Binary Tree
- Each node satisfying the Heap property
- Height of a node in a heap?
- Height of a heap?
- Heap of **n** elements has height **$\Theta(\lg n)$**

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)
```

Running time of Max-Heapify

Running time of **MAX-HEAPIFY**

1) Time to fix up the relationships between **A[i], A[LEFT(i)]** and **A[RIGHT(i)]**

2) Time to run **MAX-HEAPIFY** on a subtree rooted at one of the children of node ***i*** (assuming that the recursive call occurs)

Running time of Max-Heapify

Running time of **MAX-HEAPIFY**

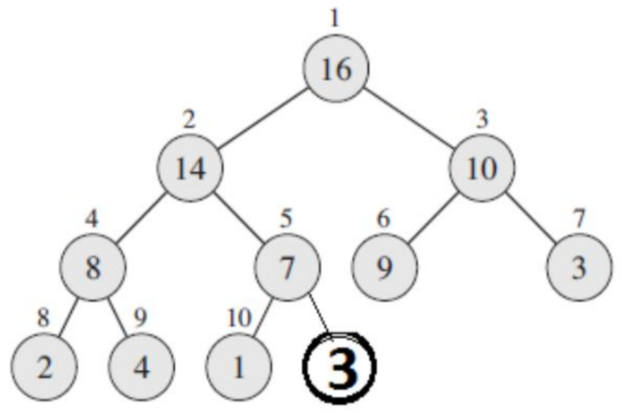
1) Time to fix up the relationships between **A[i], A[LEFT(i)]** and **A[RIGHT(i)] - $\Theta(1)$**

2) Time to run **MAX-HEAPIFY** on a subtree rooted at one of the children of node ***i*** - ?

Running time of MAX-HEAPIFY

Running time of MAX-HEAPIFY on a **subtree** rooted at one of the **children of node i:**

- Need to know the size of the subtree (# nodes) rooted at one of the children of node i.
- What is the worst case size of the subtree ….?
  - Since heap is a nearly complete binary tree, the worst case occurs when the bottom level of the tree is exactly half full
  - Hence, find the maximum number of nodes in the left subtree of the nearly complete binary tree



T**otal number** of nodes in a complete binary tree of height h is $\mathbf{2^{h+1} - 1}$

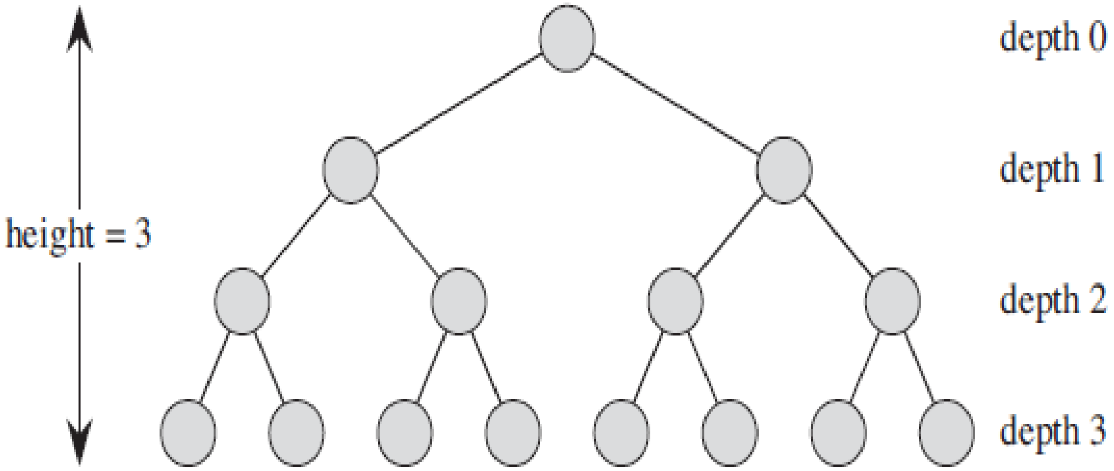
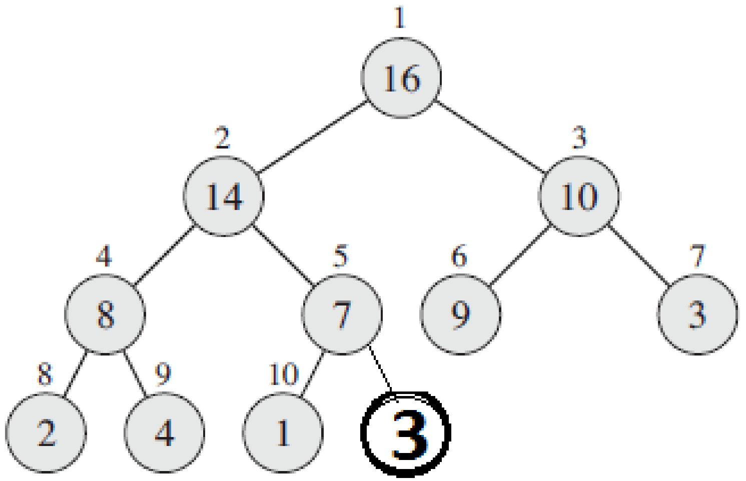


Figure B.8 A complete binary tree of height 3 with 8 leaves and 7 internal nodes.



- Total number of nodes in a nearly complete binary tree (if as in figure) of height h

  $(2^{h+1} - 1) - (2^{h-1})$

- Why do we subtract 2^{h-1} from the total number of nodes…..?

- If **h is the height** of the subtree, number of **leaf nodes is 2^h**
  - **Eg: h=3, #leaves=8**
- **Since it is a binary tree, $2^h/2$** is the number of leaf nodes in each of the **left and right subtree**

  i.e $2^h/2 = 2^{h-1}$
- Hence, total number of nodes in a nearly complete binary tree **$n = (2^{h+1} - 1) - (2^{h-1})$**

  (Writing 2^{h+1} in terms of 2^{h-1})

  $$n = (4 * 2^{h-1} - 1) - (2^{h-1})$$

  $$= 3 * 2^{h-1} - 1$$

  $$2^{h-1} = (n + 1)/3 \quad \text{-------------------}(1)$$

- Maximum number of nodes in the left subtree of a nearly complete binary tree of **height h**,

  = **$2^h - 1$ (writing in terms of 2^{h-1})**

  = $2 * 2^{h-1} – 1$ --------------------(2)

Substitute (1) in (2)

= 2 ((n + 1)/3) – 1

= (2n – 1)/ 3

<= 2n/3

Hence, the worst case size of the subtree rooted at i, is at most **2n/3**

Children's subtrees each have size at most $2n/3$

The Running time of Max-Heapify ,

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

How to solve $T(n) = T(2n/3) + \Theta(1)$

$T(n) = c + T(2/3 *n)$ ------- (1) (Replacing $\Theta(1)$ by constant c)

$T(2/3 * n) = c+ T(2/3 * 2/3 *n)$ ---(2)

Substitute (2) in (1)

$T(n) = c+c+ T(2/3 * 2/3 *n)$

$T(n) = 2c+ T(2^2/3^2 *n)$

Further :

$T(n) = 3c + T(2^3/3^3 * n)$

….

$T(n) = kc + T(2^k/3^k * n)$ -----(3)

Assume $n = (3/2)^k$ ------(4)

Substitute (4) in (3)

$T(n) = kc + T(1)$

$T(n) = c \log_{3/2} n + T(1)$

k is $\log_{3/2} n$

k can be written as $\log_2 n$. How?

$n = (3/2)^k$ --------(5)

Apply log on both sides of (5)

$\log_2 n = \log_2 (3/2)^k$

$\log_2 n = k * \log_2 (3/2)$

$k = \log_2 n / \log_2 (3/2)$

For MAX- HEAPIFY

$T(n) = kc + T(1)$

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

- Children’s subtrees each have size at most 2n/3

- The Running time of Max-Heapify ,

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

- $T(n) = O(\log n)$

- The **running time of Max-Heapify** on a node of **height h as O(h)**

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)

Running time of BUILD-MAX-HEAP

- Simple upper bound
  - Each call to MAX-HEAPIFY: $O(\log n)$
  - Number of calls to MAX-HEAPIFY: $O(n)$
  - Running time : $O(n \log n)$
- But this upper bound is not asymptotically tight

Tighter Analysis of BUILD-MAX-HEAP

Based on the following **observations** and the **property** given

Observations:

- Heights of most of the nodes are small
- Running time of MAX-HEAPIFY varies with the height of the node

Property: (Exercises: Prove the following)

- An n-element heap has height $\lfloor \mathbf{log\ n} \rfloor$
- At most $\lceil \mathbf{n / 2^{h+1}} \rceil$ number of nodes of height **h** in any **n**-element heap

Prove: At most $\lceil n / 2^{h+1} \rceil$ nodes of height h in any n-element heap

Proof by Induction on height h.

Base case:

h = 0

#nodes with height h=0 in an n-element heap $\lceil n/2 \rceil$

This is same as the #leaves in an n-element heap.

Hypothesis:

Given statement is true for height h-1.

i.e At most $\lceil n / 2^{h} \rceil$ nodes of height h-1 in an n-element heap

Induction Step: Prove the given statement for **height h.**

Let n_h be the number of nodes at height h in n-node tree T

Consider tree T' by removing the leaves of T

Let n' be the number of nodes in T'.

$n' = n - n_0,$ where n_0 = #leaves in T

$= n - \lceil n/2 \rceil$

$= \lfloor n/2 \rfloor$ ------------------------------------- (1)

#nodes at **height h** in T = #nodes at **height h-1** in T'

(T': T in which leaves are removed)

Let n'_{h-1} be the number of nodes at height h-1 in T'

$$n_h = n'_{h-1}$$

From the induction hypothesis, we can bound n'_{h-1}

$n_h = n'_{h-1} \quad <= \lceil n' / 2^h \rceil$ ------------------------------- (2)

We saw that, $n' = \lfloor n/2 \rfloor$ in equation (1)

Substitute (1) in (2)

$$<= \lceil \lfloor n/2 \rfloor / 2^h \rceil$$

$$<= \lceil n / 2^{h+1} \rceil$$

Hence, the given statement is proved.

Tighter Analysis of Build-Max-Heap

- Time required for MAX-HEAPIFY when called on a node of height h is O (h)

- The total cost of BUILD-MAX-HEAP as being bounded from above by

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

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

for $|x| < 1$.

- Summing up the above series

by substituting x= ½

$$\begin{aligned} \sum_{h=0}^{\infty} \frac{h}{2^h} &= \frac{1/2}{(1-1/2)^2} \\ &= 2 . \end{aligned}$$

Thus, we can bound the running time of BUILD-MAX-HEAP as

$$\begin{aligned} O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right) &= O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) \\ &= O(n) . \end{aligned}$$

Linear time for Building a Max-heap

- From the tighter analysis, we can see that
  - Building a Max-heap from an unordered array takes linear time.
    - Linear in terms of the number of elements in the array

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)
```

Running time of Heap sort

- Call to BUILD-MAXHEAP takes time O(n)
- Each of the n-1 calls to MAX-HEAPIFY takes time O(lg n)
- Thus, HEAPSORT takes time O(n lg n)

```
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)
```

Priority Queue (implemented using heap) Running Time of operations

- HEAP-MAXIMUM() - $\Theta(1)$
- HEAP-EXTRACT-MAX() – $O(\lg n)$
- HEAP-INCREASE-KEY() - $O(\lg n)$
- MAX-HEAP-INSERT() – $O(\lg n)$

Reference

T H Cormen, C E Leiserson, R L Rivest, C Stein *Introduction to Algorithms*, 3rd ed., PHI, 2010