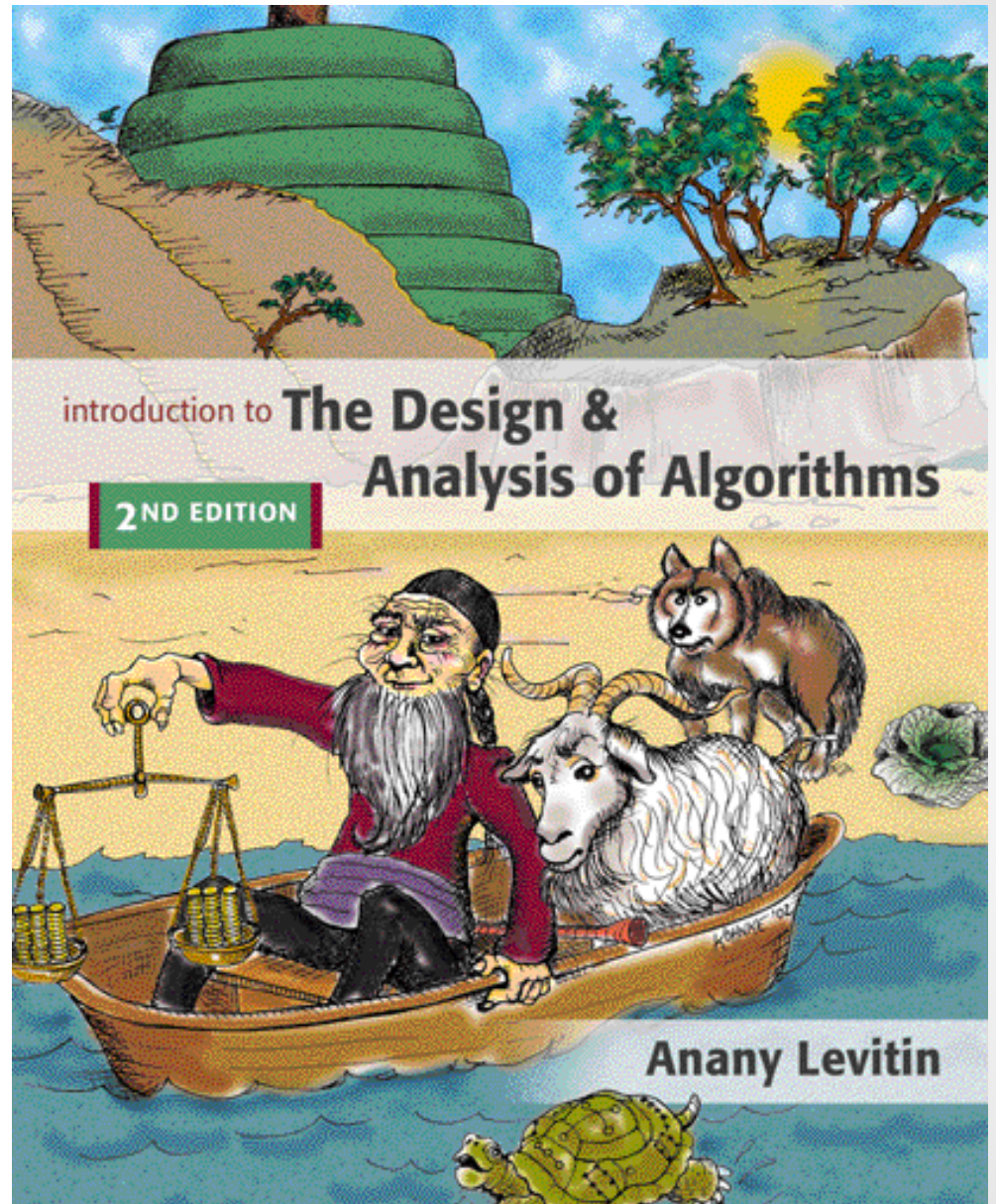


# Chapter 6

## Transform-and-Conquer



# Transform and Conquer

This group of techniques solves a problem by a transformation

*Methods work as two-stage procedures*

- *first: transformation to a more amenable solution*
- *second: problem is solved*
- Variations of idea by what we transfer a given instance to:
  - to a simpler/more convenient instance of the same problem (instance simplification)
    - Sorting, Gaussian elimination, Search trees
  - to a different representation of the same instance (representation change)
    - Heaps, Heapsort, Horner's rule, exponentiation
  - to a different problem for which an algorithm is already available (problem reduction)
    - Least common multiple, Counting path in a graph, Reduction of Optimization Problems, Linear Programming, Reduction to graph problems,

# Instance simplification - Presorting

Solve a problem's instance by transforming it into another simpler/easier instance of the same problem

## Presorting

Many problems involving lists are easier when list is sorted.

- searching
- computing the **median** (selection problem)
- checking if all elements are distinct (element uniqueness)

Also:

- Topological sorting helps solving some problems for DAGs
- Presorting is used in many geometric algorithms

# How fast can we sort ?

Efficiency of algorithms involving sorting depends on efficiency of sorting.

Theorem (see Sec. 11.2):  $\lceil \log_2 n! \rceil \approx n \log_2 n$  comparisons are necessary in the worst case to sort a list of size  $n$  by any comparison-based algorithm.

Note: About  $n \log_2 n$  comparisons are also sufficient to sort array of size  $n$  (by mergesort).

# Searching with presorting

Problem: Search for a given  $K$  in  $A[0..n-1]$

Presorting-based algorithm:

Stage 1 Sort the array by an efficient sorting algorithm

Stage 2 Apply binary search

Efficiency:  $\Theta(n \log n) + O(\log n) = \Theta(n \log n)$

Good or bad?

Why do we have our dictionaries, telephone directories, etc. sorted?

# Element Uniqueness with presorting

- Presorting-based algorithm

Stage 1: sort by efficient sorting algorithm (e.g. mergesort)

Stage 2: scan array to check pairs of adjacent elements

Efficiency:  $\Theta(n \log n) + O(n) = \Theta(n \log n)$

- Brute force algorithm  
Compare all pairs of elements

Efficiency:  $O(n^2)$

- Another algorithm? Hashing

# Computing a mode

- In descriptive statistics, mode is the value that has the largest number of observations i.e., the most frequent value or values. The mode is not necessarily unique, unlike the average or the median. It is especially useful when the values or observations are not numeric, since the mean and median may not be well defined.
- {maçã, banana, laranja, laranja, laranja, pêsego} -Mode: laranja
- {1, 3, 5, 5, 6, 6} - Modes: 5 e 6 (bimodal)
- {1, 3, 2, 5, 8, 7, 9} - No mode (amodal)
- Algorithms for computing mode – Which is best?
  - Brute Force: computing the frequencies
  - Transform & Conquer: presorting and finding the longest run

# Computing a mode

- Brute Force: computing the frequencies
- $C(n) = \sum_{1 \leq i \leq n} (i-1) = (n-1)n/2 \in \Theta(n^2)$
- Transform and Conquer: find the longest run of adjacent equal values in the sorted array
- $C(n) = C_{\text{sort}}(n) + C_{\text{flr}}(n)$   
     $= n \log n + n$  (*why?*)  
     $= O(n \log n)$



# Instance simplification

## Gaussian Elimination (1777-1855)

**Given:** A system of  $n$  linear equations in  $n$  unknowns with an arbitrary coefficient matrix.

**Transform to:** An equivalent system of  $n$  linear equations in  $n$  unknowns with an upper triangular coefficient matrix.

Solve the latter by substitutions starting with the last equation and moving up to the first one.

$$a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n = b_1$$

$$a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n = b_2$$



$$a'_{11}x_1 + a'_{12}x_2 + \dots + a'_{1n}x_n = b'_1$$

$$a'_{22}x_2 + \dots + a'_{2n}x_n = b'_2$$

$$a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nn}x_n = b_n$$

$$a'_{nn}x_n = b'_n$$

# Gaussian Elimination (cont.)

The transformation is accomplished by a sequence of elementary operations on the system's coefficient matrix (which don't change the system's solution):

Elementary operations:

- Exchanging two equations of the system
- Replacing an equation with its nonzero multiple
- Replacing an equation with a sum or difference of this equation and some multiple of another equation

for  $i \leftarrow 1$  to  $n-1$  do

    replace each of the subsequent rows (i.e., rows  $i+1, \dots, n$ ) by a difference between that row and an appropriate multiple of the  $i$ -th row to make the new coefficient in the  $i$ -th column of that row 0

# Example of Gaussian Elimination

Solve

$$\begin{array}{rcl} 2x_1 - 4x_2 + x_3 & = & 6 \\ 3x_1 - x_2 + x_3 & = & 11 \\ x_1 + x_2 - x_3 & = & -3 \end{array}$$

Gaussian elimination

$$\begin{array}{cccc|cccc} 2 & -4 & 1 & 6 & & & & \\ 3 & -1 & 1 & 11 & \text{row2} - (3/2)*\text{row1} & 0 & 5 & -1/2 & 2 \\ 1 & 1 & -1 & -3 & \text{row3} - (1/2)*\text{row1} & 0 & 3 & -3/2 & -6 & \text{row3} - (3/5)*\text{row2} \end{array}$$

$$\begin{array}{cccc} 2 & -4 & 1 & 6 \\ 0 & 5 & -1/2 & 2 \\ 0 & 0 & -6/5 & -36/5 \end{array}$$

Backward substitution

$$\begin{aligned} x_3 &= (-36/5) / (-6/5) = 6 \\ x_2 &= (2 + (1/2)*6) / 5 = 1 \\ x_1 &= (6 - 6 + 4*1)/2 = 2 \end{aligned}$$

# Pseudocode and Efficiency of Gaussian Elimination

Stage 1: Reduction to the upper-triangular matrix

for  $i \leftarrow 1$  to  $n-1$  do

    for  $j \leftarrow i+1$  to  $n$  do

        for  $k \leftarrow i$  to  $n+1$  do

$$A[j, k] \leftarrow A[j, k] - A[i, k] * A[j, i] / A[i, i]$$

Stage 2: Backward substitution

for  $j \leftarrow n$  downto 1 do

$t \leftarrow 0$

    for  $k \leftarrow j+1$  to  $n$  do

$$t \leftarrow t + A[j, k] * x[k]$$

$$x[j] \leftarrow (A[j, n+1] - t) / A[j, j]$$

Efficiency:  $\Theta(n^3) + \Theta(n^2) = \Theta(n^3)$

# Searching Problem

Problem: Given a (multi)set  $S$  of keys and a search key  $K$ , find an occurrence of  $K$  in  $S$ , if any

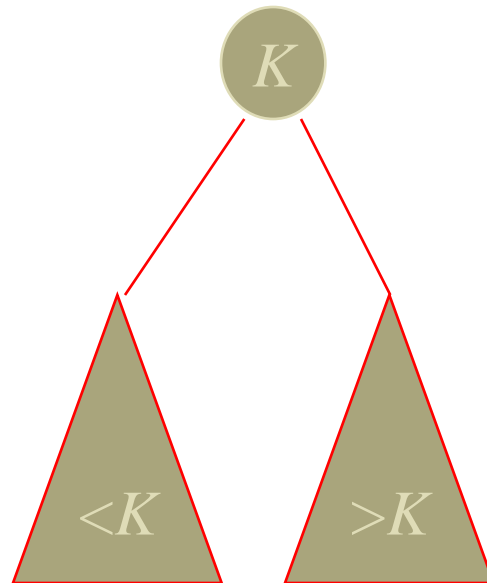
- Searching must be considered in the context of:
  - file size (internal vs. external)
  - dynamics of data (static vs. dynamic)
- Dictionary operations (dynamic data):
  - find (search)
  - insert
  - delete

# Taxonomy of Searching Algorithms

- List searching
  - sequential search
  - binary search
  - interpolation search (cap.5 – decrease and conquer)
    - Mimics human search for “Brown” and “Smith” in a dictionary
- Tree searching
  - binary search tree
  - binary balanced trees:
    - instance simplification: an unbalanced BT is transformed in a balanced one: AVL trees, red-black trees, splay trees
    - representation change: allow more than in element in a node: multiway balanced trees: 2-3 trees, 2-3-4 trees, B trees
- Hashing
  - open hashing (separate chaining)
  - closed hashing (open addressing)

# Binary Search Tree

Arrange keys in a binary tree with the *binary search tree property*:



**Example: 5, 3, 1, 10, 12, 7, 9**

# Dictionary Operations on Binary Search Trees

Searching – straightforward

Insertion – search for key, insert at leaf where search terminated

Deletion – 3 cases:

deleting key at a leaf

deleting key at node with single child

deleting key at node with two children

Efficiency depends of the tree's height:  $\lfloor \log_2 n \rfloor \leq h \leq n-1$ ,

with height average (random files) be about  $3\log_2 n$

Thus all three operations have

- worst case efficiency:  $\Theta(n)$
- average case efficiency:  $\Theta(\log n)$

Bonus: inorder traversal produces sorted list



# Balanced Search Trees

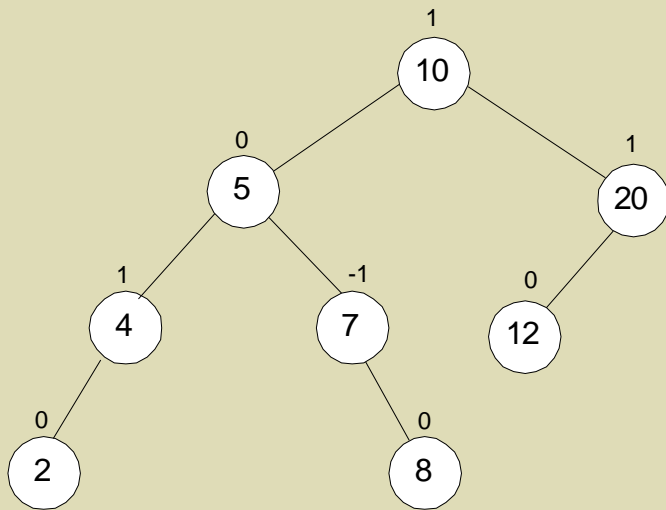
Attractiveness of *binary search tree* is marred by the bad (linear) worst-case efficiency. Two ideas to overcome it are:

- to rebalance binary search tree when a new insertion makes the tree “too unbalanced”
  - *AVL trees*
  - *red-black trees*
- to allow more than one key per node of a search tree
  - *2-3 trees*
  - *2-3-4 trees*
  - *B-trees*

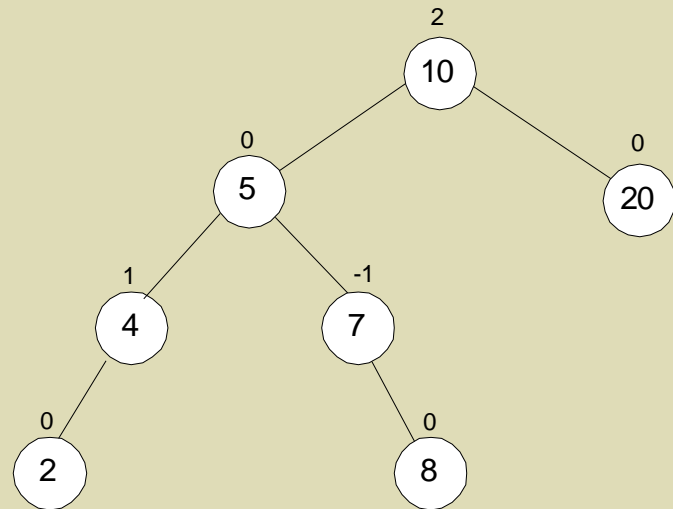
# Balanced trees: AVL trees

## (instance simplification)

Definition An *AVL tree* is a binary search tree in which, for every node, the difference between the heights of its left and right subtrees, called the *balance factor*, is at most 1 (with the height of an empty tree defined as -1)



(a)

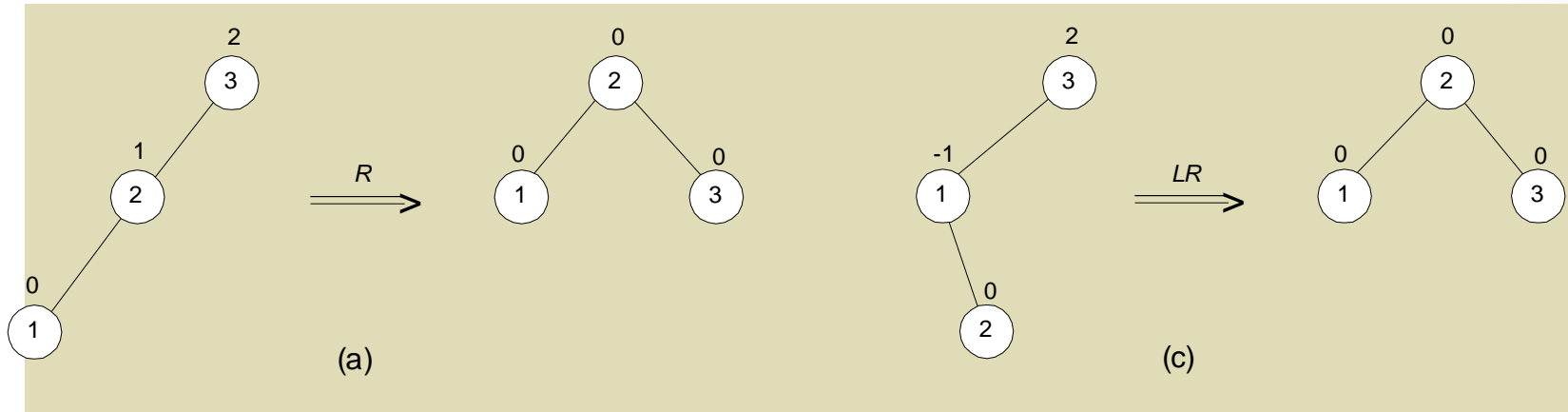


(b)

**Tree (a) is an AVL tree; tree (b) is not an AVL tree**

# Rotations

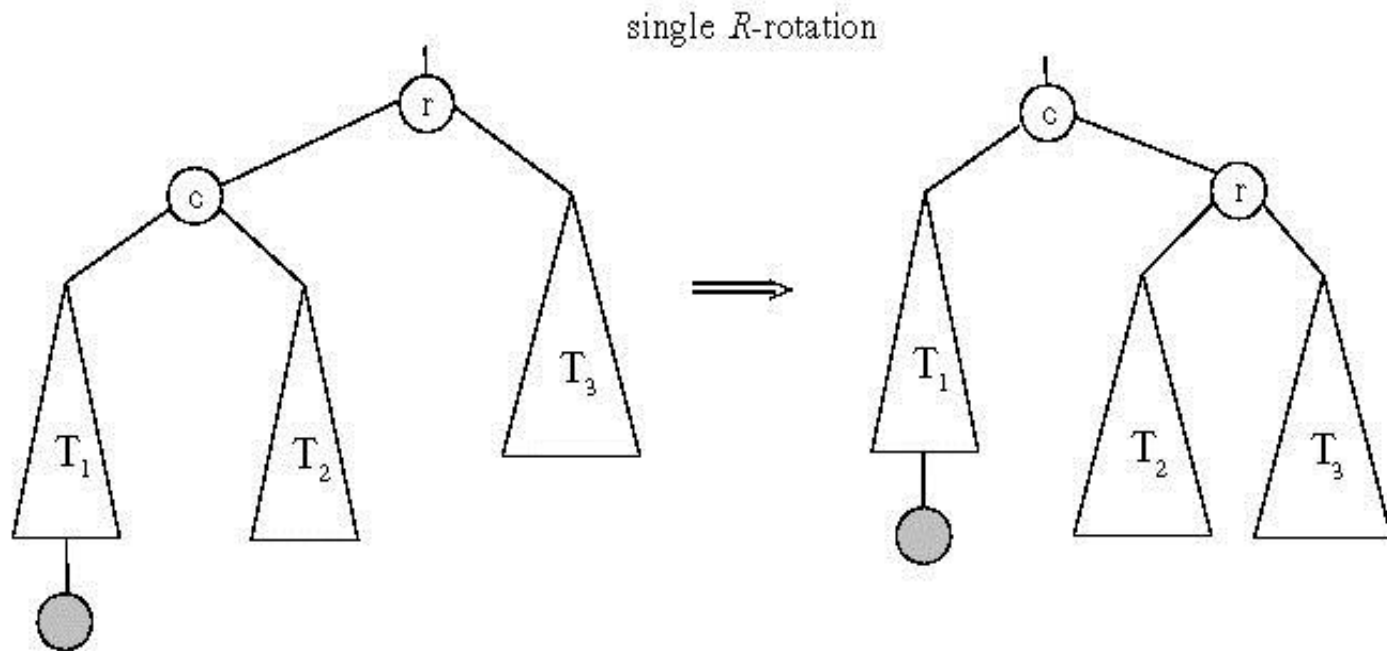
If a key insertion violates the balance requirement at some node, the subtree rooted at that node is transformed via one of the four *rotations*. (The rotation is always performed for a subtree rooted at an “unbalanced” node closest to the new leaf.)



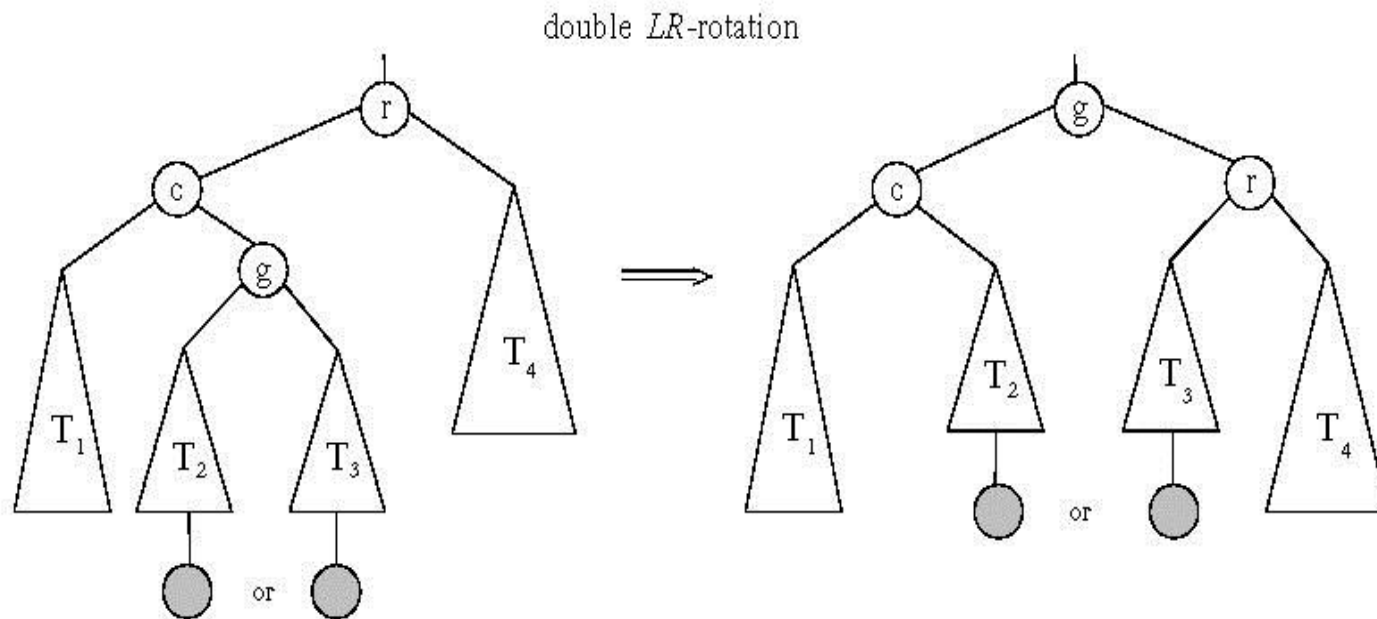
**Single *R*-rotation**

**Double *LR*-rotation**

# General case: Single R-rotation

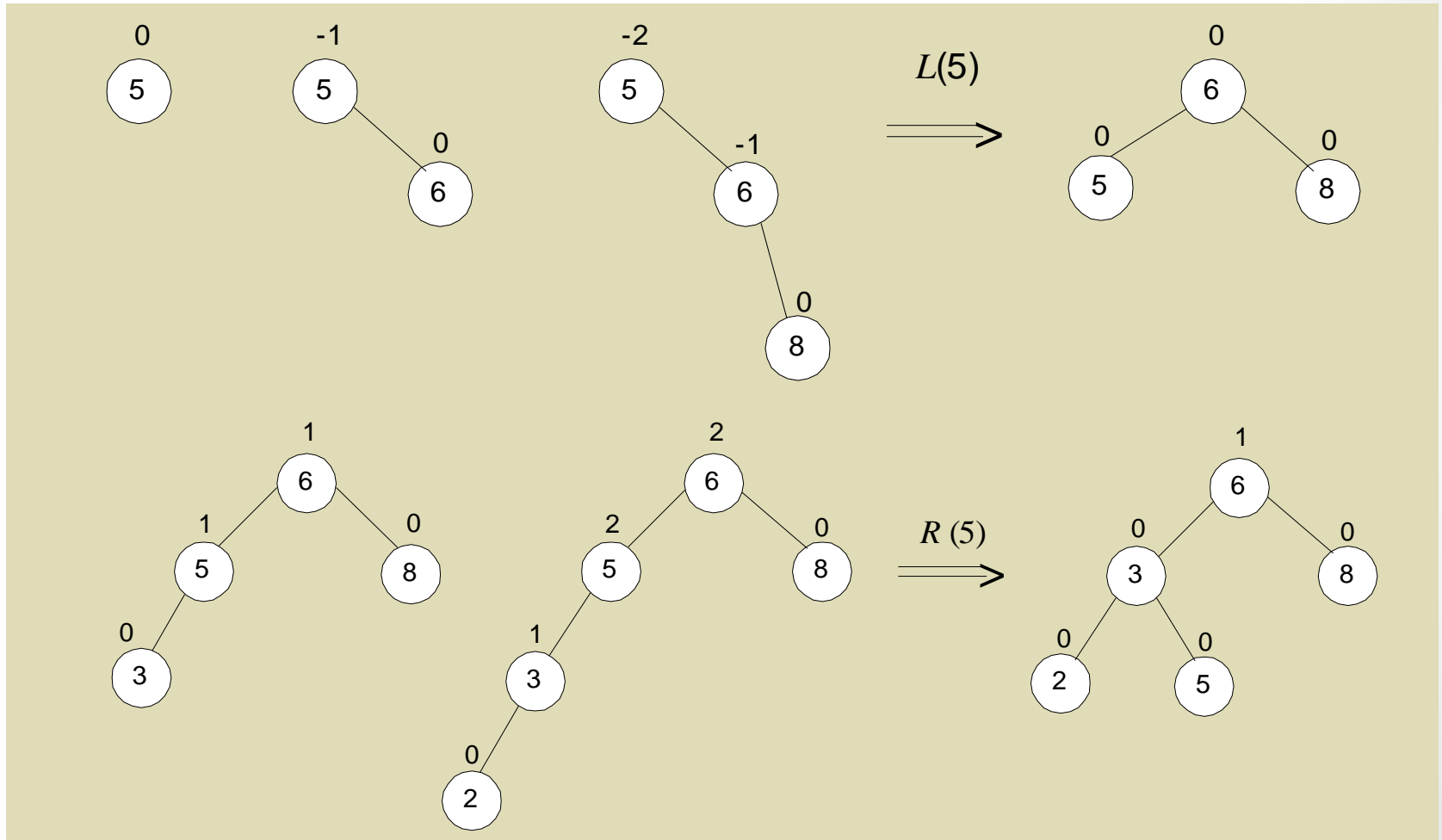


# General case: Double LR-rotation

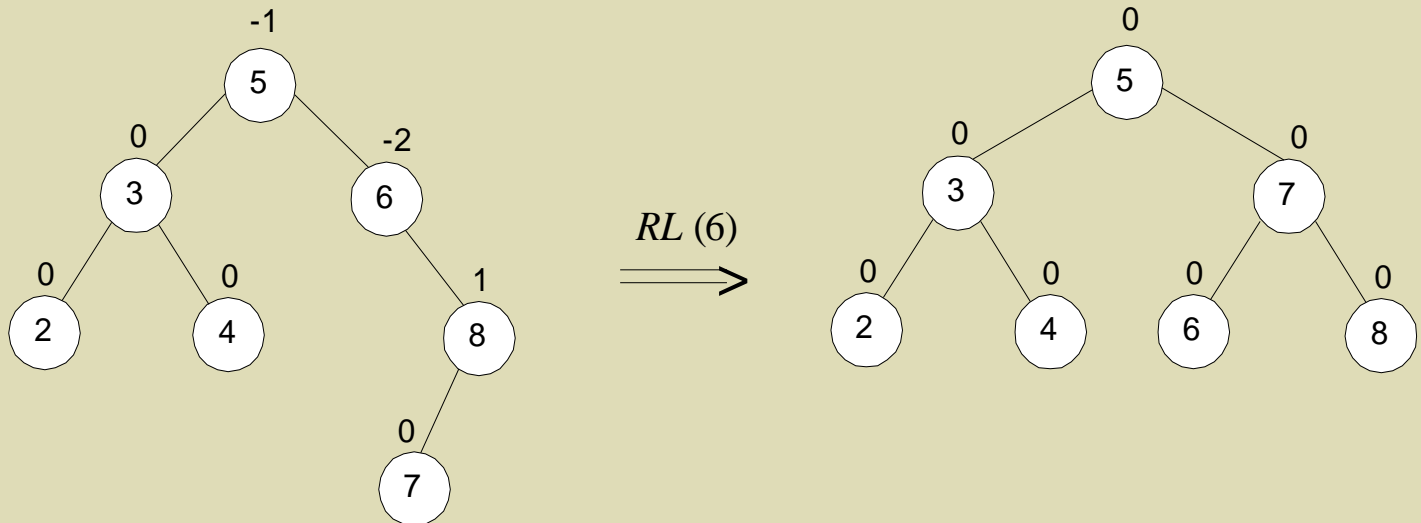
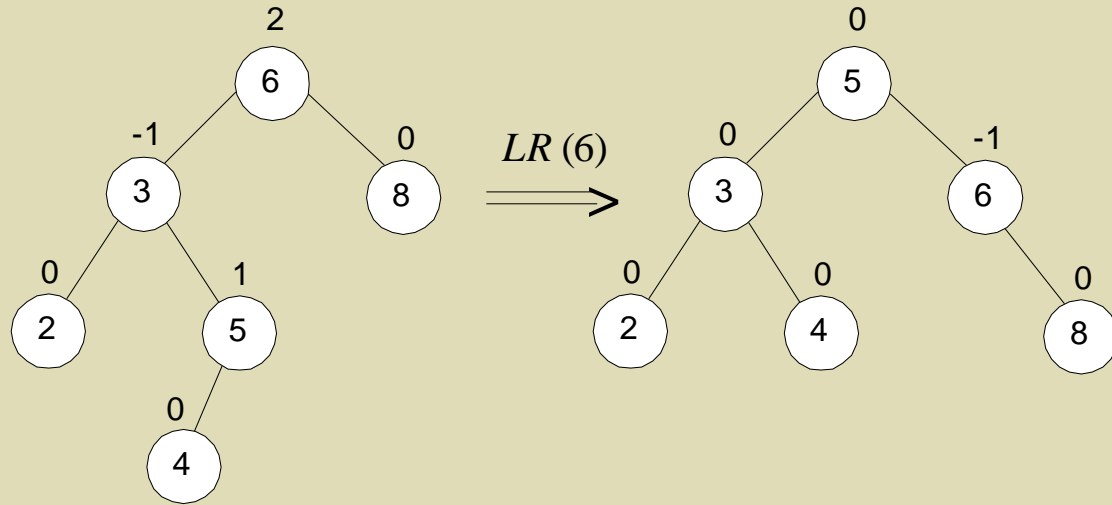


# AVL tree construction - an example

Construct an AVL tree for the list 5, 6, 8, 3, 2, 4, 7



# AVL tree construction - an example (cont.)



# Analysis of AVL trees

- $h \leq 1.4404 \log_2 (n + 2) - 1.3277$   
average height:  $1.01 \log_2 n + 0.1$  for large  $n$  (found empirically)
- Search and insertion are  $O(\log n)$
- Deletion is more complicated but is also  $O(\log n)$
- Disadvantages:
  - frequent rotations
  - complexity
- A similar idea: *red-black trees*  
(height of subtrees is allowed to differ by up to a factor of 2)

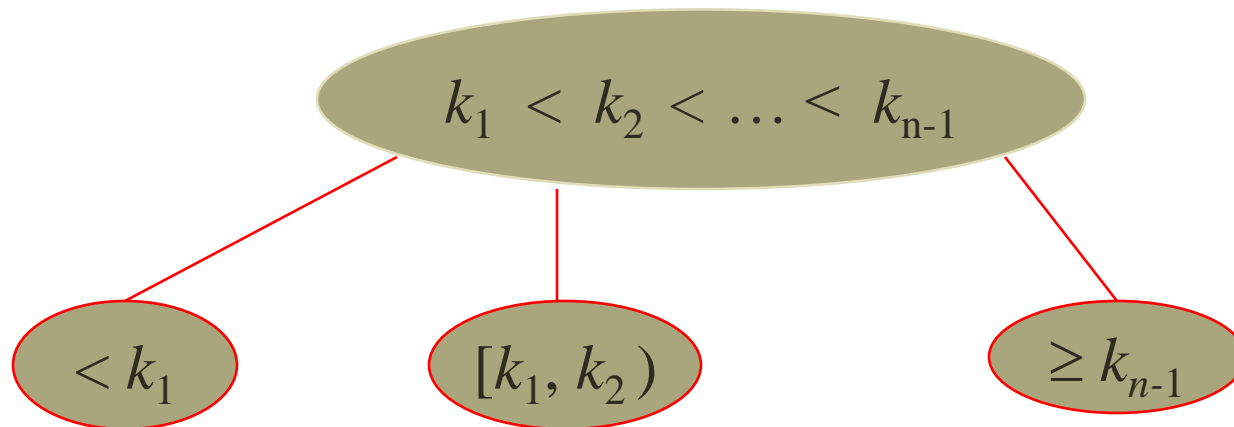


# Multiway Search Trees

## (representation change)

Definition A *multiway search tree* is a search tree that allows more than one key in the same node of the tree.

Definition A node of a search tree is called an *n-node* if it contains  $n-1$  ordered keys (which divide the entire key range into  $n$  intervals pointed to by the node's  $n$  links to its children):

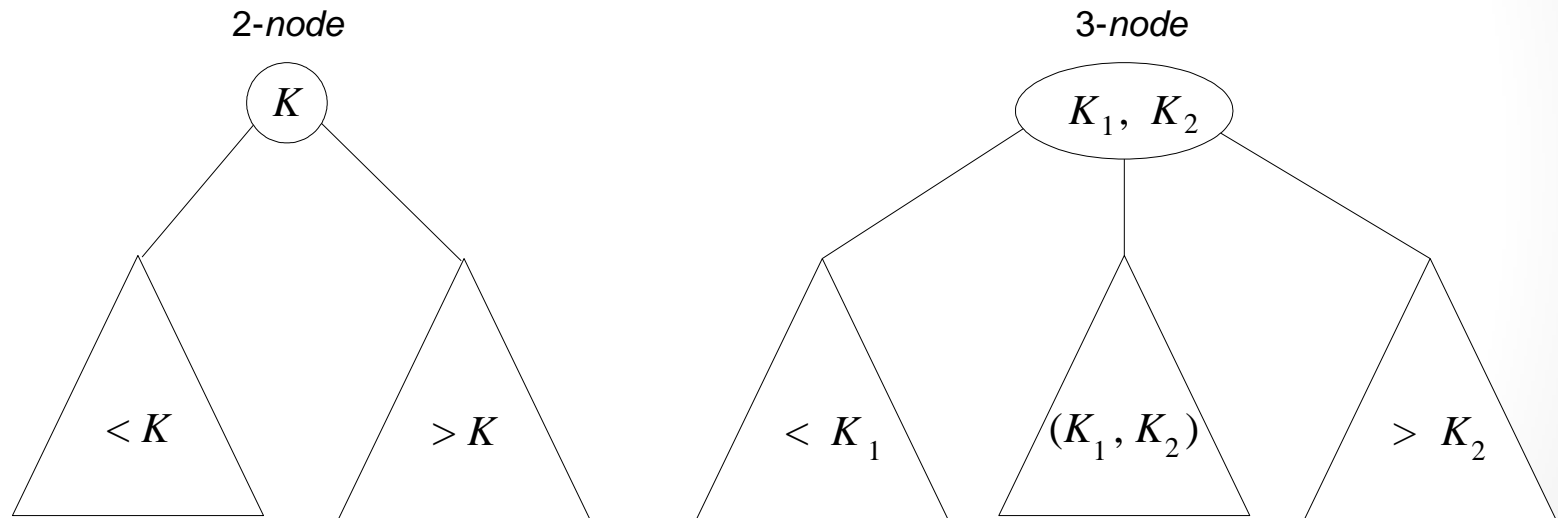


Note: Every node in a classical binary search tree is a 2-node

# 2-3 Tree

Definition A 2-3 tree is a search tree that

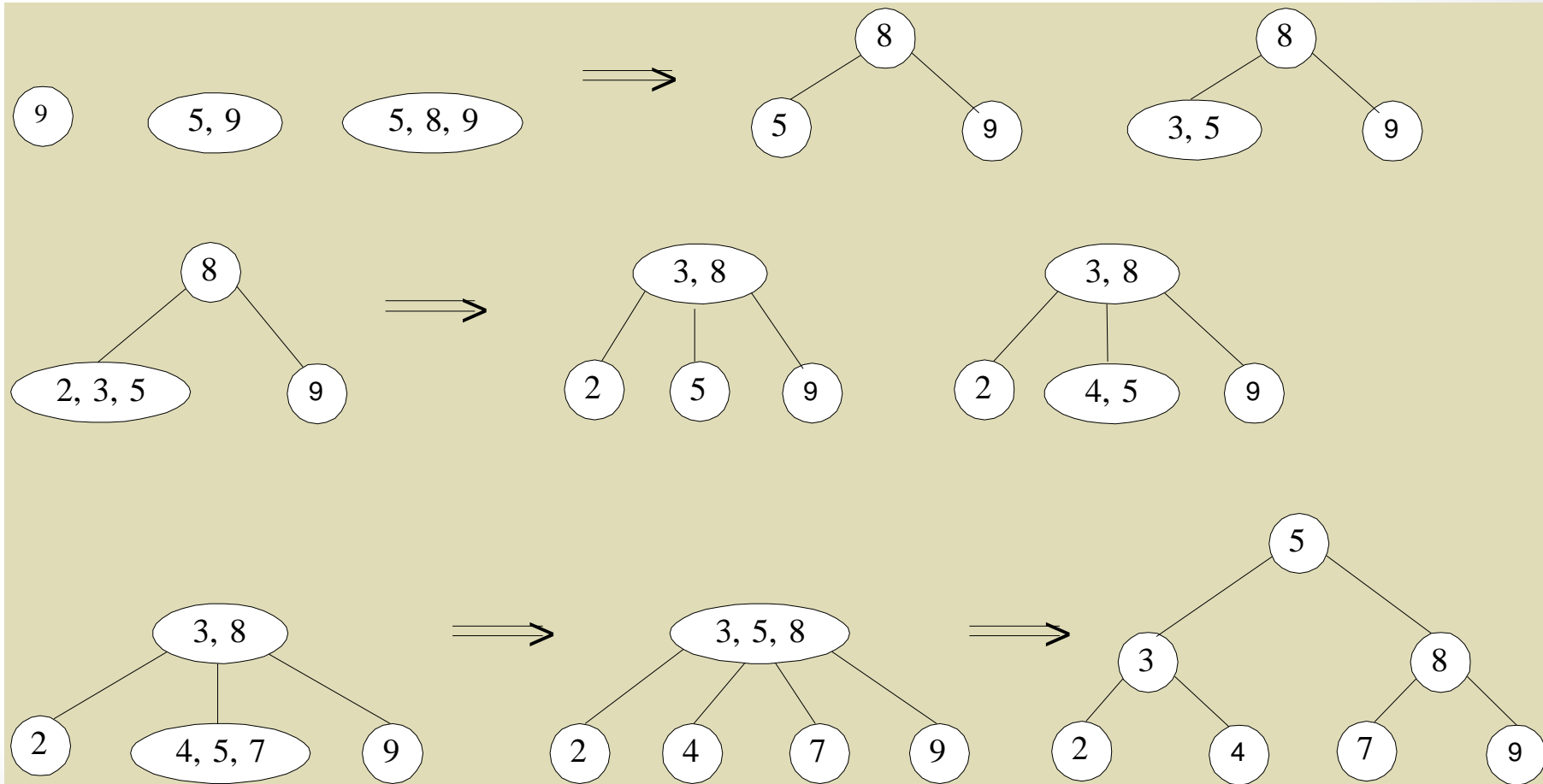
- ▶ may have 2-nodes (single key) and 3-nodes (two ordered keys)
- ▶ height-balanced (all leaves are on the same level)



A 2-3 tree is constructed by successive insertions of keys given, with a **new key always inserted into a leaf of the tree (except for empty tree)**. If the leaf is a 3-node, it's split into two with the middle key promoted to the parent.

# 2-3 tree construction – an example

Construct a 2-3 tree the list 9, 5, 8, 3, 2, 4, 7



# Analysis of 2-3 trees

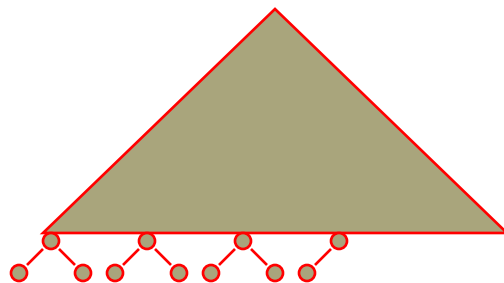
- $\log_3(n + 1) - 1 \leq h \leq \log_2(n + 1) - 1$   
*full of 3-nodes* *full of 2-nodes*
- Search, insertion, and deletion are in  $\Theta(\log n)$
- The idea of 2-3 tree can be generalized by allowing more keys per node
  - 2-3-4 trees
  - B-trees

# Heaps and Heapsort

## (representation change)

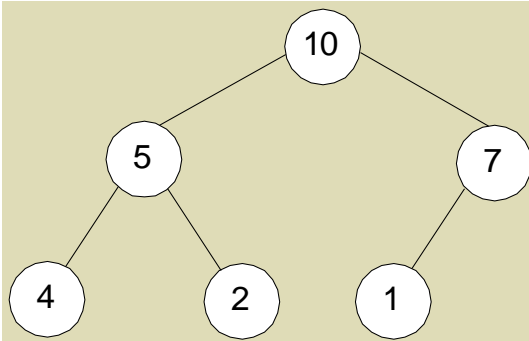
Definition A *heap* is a (data structure) binary tree with keys at its nodes (one key per node) such that:

- (the tree's shape requirement) It is essentially complete, i.e., all its levels are full except possibly the last level, where only some rightmost keys may be missing

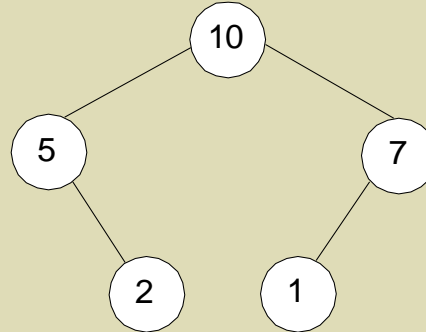


- (the parental's dominance requirement) The key at each node is  $\geq$  keys at its children

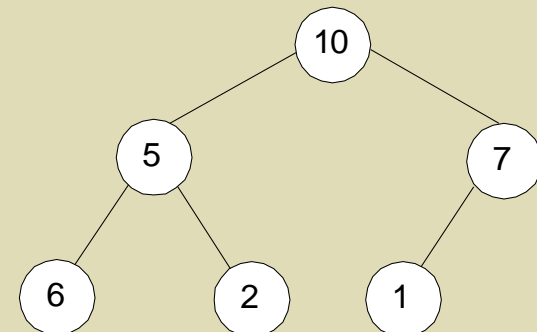
# Illustration of the heap's definition



**a heap**



**not a heap**



**not a heap**

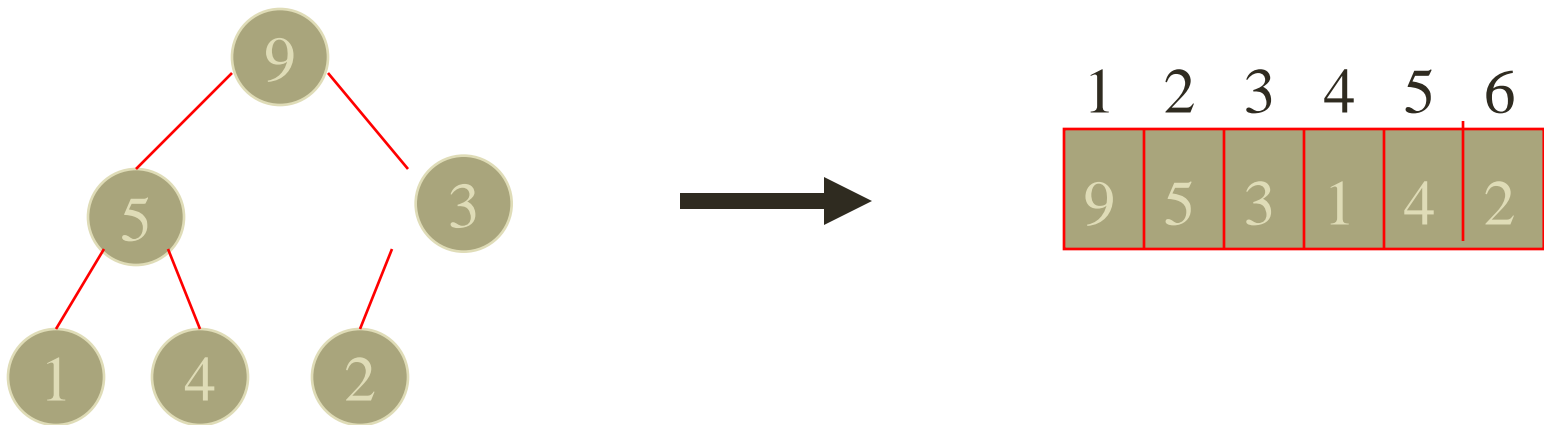
**Note: Heap's elements are ordered top down (along any path down from its root), but they are not ordered left to right**

# Some Important Properties of a Heap

- Given  $n$ , there exists a unique binary tree with  $n$  nodes that is essentially complete, with  $h = \lfloor \log_2 n \rfloor$
- The root contains the largest key
- The subtree rooted at any node of a heap is also a heap
- A heap can be represented as an array

# Heap's Array Representation

Store heap's elements in an array (whose elements indexed, for convenience, 1 to  $n$ ) in top-down left-to-right order



- Left child of node  $j$  is at  $2j$
- Right child of node  $j$  is at  $2j+1$
- Parent of node  $j$  is at  $\lfloor j/2 \rfloor$
- Parental nodes are represented in the first  $\lfloor n/2 \rfloor$  locations
- Algoritmos são mais fáceis de entender se Heap analisada como Binary Tree, mas a implementação é + simples e + eficiente com arrays!



# Heap Construction (bottom-up)

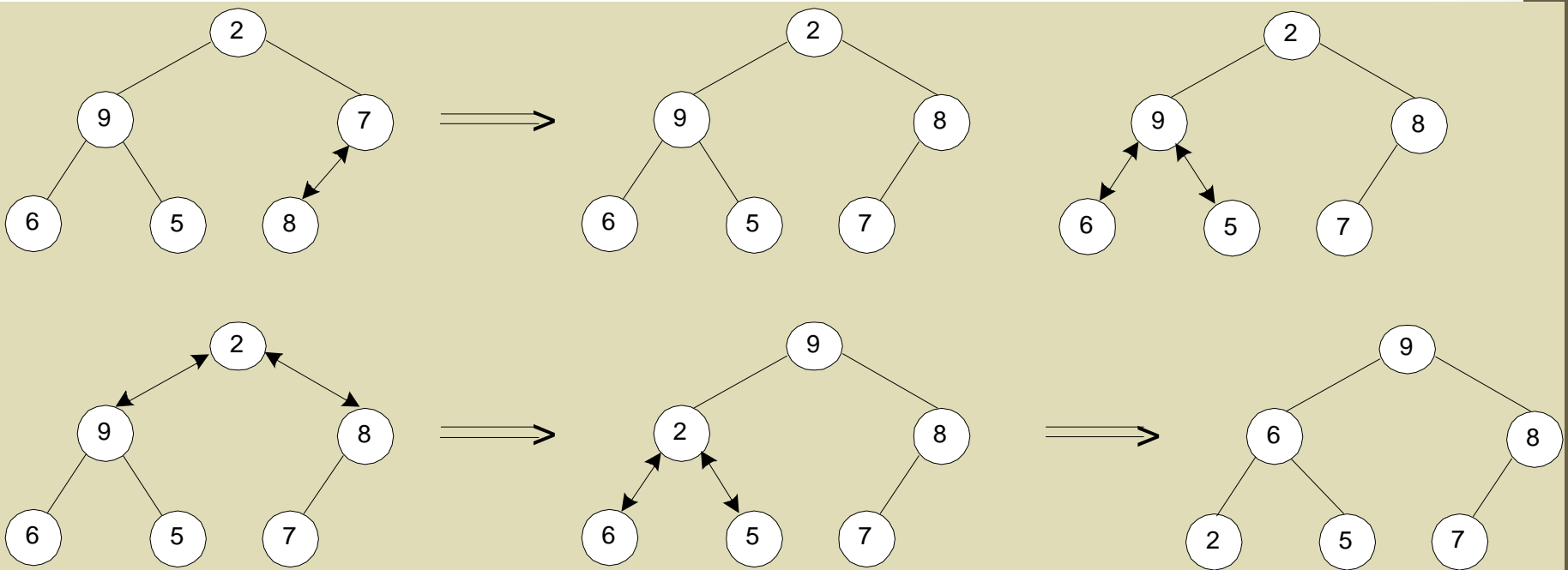
Step 0: Initialize the structure with keys in the order given

Step 1: Starting with the last (rightmost) parental node, fix the heap rooted at it, if it doesn't satisfy the heap condition: keep exchanging it with its largest child until the heap condition holds

Step 2: Repeat Step 1 for the preceding parental node

# Example of Heap Construction

**Construct a heap for the list 2, 9, 7, 6, 5, 8**



# Pseudocode of bottom-up heap construction

```
Algorithm HeapBottomUp( $H[1..n]$ )  
//Constructs a heap from the elements of a given array  
// by the bottom-up algorithm  
//Input: An array  $H[1..n]$  of orderable items  
//Output: A heap  $H[1..n]$   
for  $i \leftarrow \lfloor n/2 \rfloor$  downto 1 do  
     $k \leftarrow i$ ;  $v \leftarrow H[k]$   
     $heap \leftarrow \text{false}$   
    while not  $heap$  and  $2 * k \leq n$  do  
         $j \leftarrow 2 * k$   
        if  $j < n$  //there are two children  
            if  $H[j] < H[j + 1]$   $j \leftarrow j + 1$   
        if  $v \geq H[j]$   
             $heap \leftarrow \text{true}$   
        else  $H[k] \leftarrow H[j]$ ;  $k \leftarrow j$   
     $H[k] \leftarrow v$ 
```

# Analysis of HeapBottomUp

- Build heap for a given list of  $n$  keys

worst-case

$$C(n) = \sum_{i=0}^{h-1} \underbrace{2^{h-i}}_{\text{\# nodes at level } i} 2^i = 2(n - \log_2(n+1)) \in \Theta(n)$$

# Priority Queue

A *priority queue* is the ADT of a set of elements with numerical priorities with the following operations:

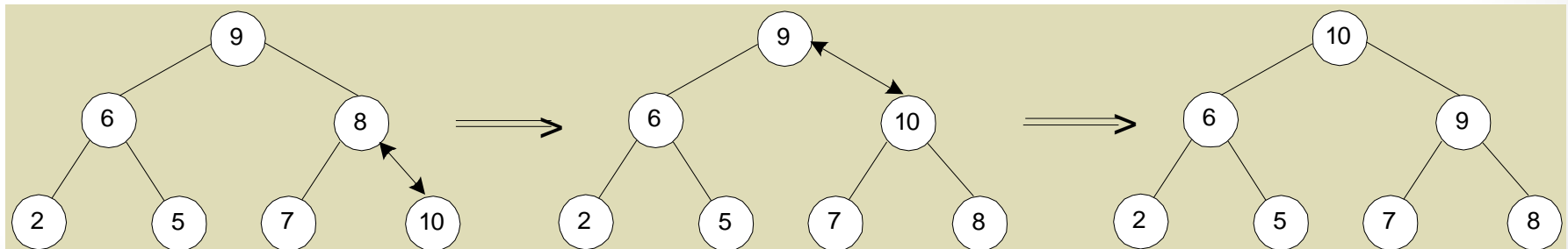
- find element with highest priority
  - delete element with highest priority
  - insert element with assigned priority (see below)
- 
- Heap is a very efficient way for implementing priority queues

# Insertion of a New Element into a Heap

## top-down heap construction

- ▶ Insert the new element at last position in heap
- ▶ Compare it with its parent and, if it violates heap condition, exchange them
- ▶ Continue comparing the new element with nodes up the tree until the heap condition is satisfied

Example: Insert key 10



- ▶ Efficiency of insertion :  $O(\log n)$  (mesmo para remoção)

# Polynomial Evaluation

Given a polynomial of degree  $n$

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$$

and a specific value of  $x$ , find the value of  $p$  at that point.

Two brute-force algorithms:

# Polynomial Evaluation

Given a polynomial of degree  $n$

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$$

and a specific value of  $x$ , find the value of  $p$  at that point.

Two brute-force algorithms:

$$C(n) = n^2$$

```
p ← 0
for i ← n downto 0 do
    power ← 1
    for j ← 1 to i do
        power ← power * x
    p ← p + ai * power
return p
```

$$C(n) = n$$

```
p ← a0; power ← 1
for i ← 1 to n do
    power ← power * x
    p ← p + ai * power
return p
```



# Horner's Rule For Polynomial Evaluation (representation change)

$$\begin{aligned}\text{Example: } p(x) &= 2x^4 - x^3 + 3x^2 + x - 5 = \\ &= x(2x^3 - x^2 + 3x + 1) - 5 = \\ &= x(x(2x^2 - x + 3) + 1) - 5 = \\ &= x(x(x(2x - 1) + 3) + 1) - 5\end{aligned}$$

Substitution into the last formula leads to a faster algorithm

Same sequence of computations are obtained by simply arranging the coefficient in a table and proceeding as follows:

coefficients	2	-1	3	1	-5
$x=3$	2	$3*2+(-1)=5$	$3*5+3=18$	$3*18+1=55$	$3*55+(-5) = 160$

# Horner's Rule pseudocode

**ALGORITHM** *Horner*( $P[0..n]$ ,  $x$ )

//Evaluates a polynomial at a given point by Horner's rule

//Input: An array  $P[0..n]$  of coefficients of a polynomial of degree  $n$

// (stored from the lowest to the highest) and a number  $x$

//Output: The value of the polynomial at  $x$

$p \leftarrow P[n]$

**for**  $i \leftarrow n - 1$  **downto** 0 **do**

$p \leftarrow x * p + P[i]$

**return**  $p$

Efficiency of Horner's Rule:

► # multiplications = # additions =  $n$

# Horner's Rule

- ▶ Considere  $p(x) = x^n$
- ▶ O que acontece ao avaliar  $P(x)$  para  $x=a$  (ou seja,  $a^n$ )?
- ▶  $P[i] = 0, i=n-1, n-2, \dots, 0$
- ▶ E o cálculo de  $a^n$  tem  $\Theta(n)$  – Bruta força

**ALGORITHM** *Horner*( $P[0..n], x$ )

//Evaluates a polynomial at a given point by Horner's rule

//Input: An array  $P[0..n]$  of coefficients of a polynomial of degree  $n$

// (stored from the lowest to the highest) and a number  $x$

//Output: The value of the polynomial at  $x$

$p \leftarrow P[n]$

**for**  $i \leftarrow n - 1$  **downto** 0 **do**

$p \leftarrow x * p + P[i]$

**return**  $p$

# Computing $a^n$ (revisited)

## Left-to-right binary exponentiation

Initialize product accumulator by 1.

Scan  $n$ 's binary expansion from left to right and do the following:

If the current binary digit is 0, square the accumulator (S);

if the binary digit is 1, square the accumulator and multiply it by  $a$  (SM).

Example: Compute  $a^{13}$ . Here,  $n = 13 = 1101_2$ .

binary rep. of 13:	1	1	0	1
		SM	SM	S
accumulator:	1	$1^2 * a = a$	$a^2 * a = a^3$	$(a^3)^2 = a^6$
(computed left-to-right)				$(a^6)^2 * a = a^{13}$

Efficiency:  $(b-1) \leq M(n) \leq 2(b-1)$  where  $b = \lfloor \log_2 n \rfloor + 1$

# Computing $a^n$ (cont.)

## Right-to-left binary exponentiation

Scan  $n$ 's binary expansion from right to left and compute  $a^n$  as the product of terms  $a^{2^i}$  corresponding to 1's in this expansion.

Example Compute  $a^{13}$  by the right-to-left binary exponentiation. Here,  $n = 13 = 1101_2$ .

1	1	0	1	
$a^8$	$a^4$	$a^2$	$a$	: $a^{2^i}$ terms
$a^5 * a^8 = a^{13}$	$a * a^4 = a^5$		$a$	: product

(computed right-to-left)

Efficiency: same as that of left-to-right binary exponentiation

# Problem Reduction

This variation of transform-and-conquer solves a problem by transforming it into different problem for which an algorithm is already available.

To be of practical value, the combined time of the transformation and solving the other problem should be smaller than solving the problem as given by another method.

# Examples of Solving Problems by Reduction

- computing  $\text{lcm}(m, n)$  via computing  $\text{gcd}(m, n)$
- counting number of paths of length  $n$  in a graph by raising the graph's adjacency matrix to the  $n$ -th power
- transforming a maximization problem to a minimization problem and vice versa (also, min-heap construction)
- linear programming
- reduction to graph problems (e.g., solving puzzles via state-space graphs)

# computing $\text{lcm}(m, n)$

- least common multiple

$$\text{lcm}(24, 60) = 120$$

$$24 = 2 * 2 * 2 * 3 \quad 60 = 2 * 2 * 3 * 5 \quad \text{lcm} = (2 * 2 * 3) * 2 * 5 = 120$$



prime  
factors

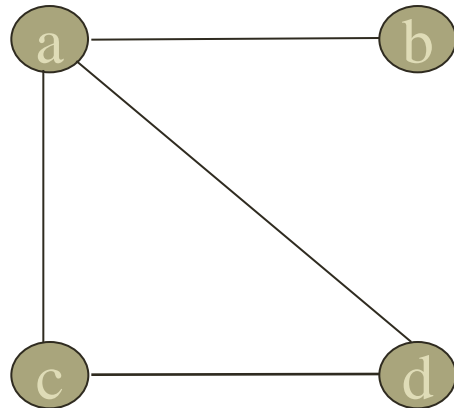
Problem reduction: via computing  $\text{gcd}(m, n)$

$$\text{lcm}(m, n) = m * n / \text{gcd}(m, n)$$



# Counting number of paths of length $n$ in a graph by raising the graph's adjacency matrix to the $n$ -th power

- Theorem: The  $ij$  entry in the  $n$ th power of the incidence matrix for any graph or digraph is exactly the number of different paths of length  $n$ , beginning at vertex  $i$  and ending at vertex  $j$ .

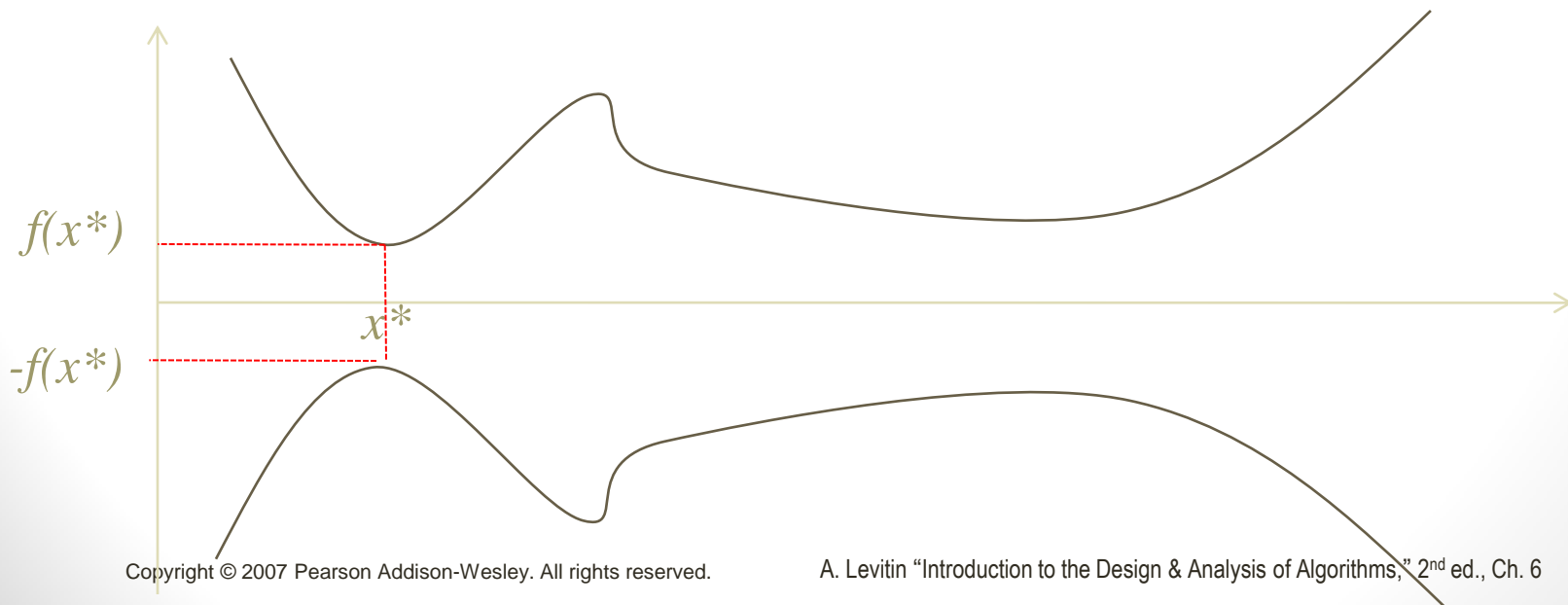


$$A = \begin{array}{ccccc} & a & b & c & d \\ \begin{array}{c} a \\ b \\ c \\ d \end{array} & \begin{array}{ccccc} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{array} \end{array} \quad A^2 = \begin{array}{ccccc} & a & b & c & d \\ \begin{array}{c} a \\ b \\ c \\ d \end{array} & \begin{array}{ccccc} 3 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 2 & 1 \\ 1 & 1 & 1 & 2 \end{array} \end{array}$$

- Proof by induction (e.g. Incidence is no Coincidence, By Nick Korevaar <http://www.math.utah.edu/mathcircle/notes/incidence.pdf>)

# Transforming a maximization problem to a minimization problem and vice versa

- $\min f(x) = - \max[-f(x)]$
- $\max f(x) = - \min[-f(x)]$



# Reduction to graph problems

e.g., solving puzzles via state-space graphs

- vértices representam possíveis estados do problema
- arestas representam possíveis transições entre estados
- *state-space-graph*
- Problema é reduzido para encontrar um caminho entre um vértice de estado inicial e chegar em um vértice de estado final
- Caso geral: assunto importante em IA
- Caso particular: Sec. 12.1 e 12.2.
- Exemplo fácil: Puzzle do Barqueiro, Lobo, Bode e Repolho

