Computer Algorithms

Lecture 7: Transform-and-Conquer– Ch 6

Dr. Manal Helal, Spring 2014. http://moodle.manalhelal.com

Lecture Learning Objectives

1. Use a Transform-and-Conquer algorithm design strategy to transform a problem to a simpler form to solve such as sorting, Gaussian Elimination,
2. Use a Transform-and-Conquer algorithm design strategy to transform a problem to another easier representation such as Search Trees.

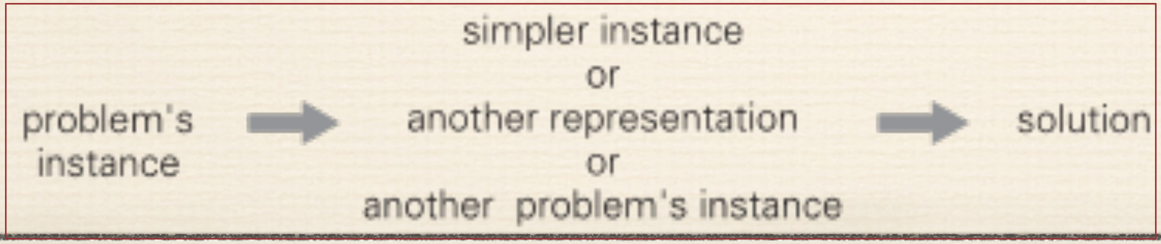
Transform and Conquer

This group of techniques solves a problem by a *transformation* to:

1. a simpler/more convenient instance of the same problem (*instance simplification*)

2. a different representation of the same instance (*representation change*)

3. a different problem for which an algorithm is already available (*problem reduction*)



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, e.g.

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 ?

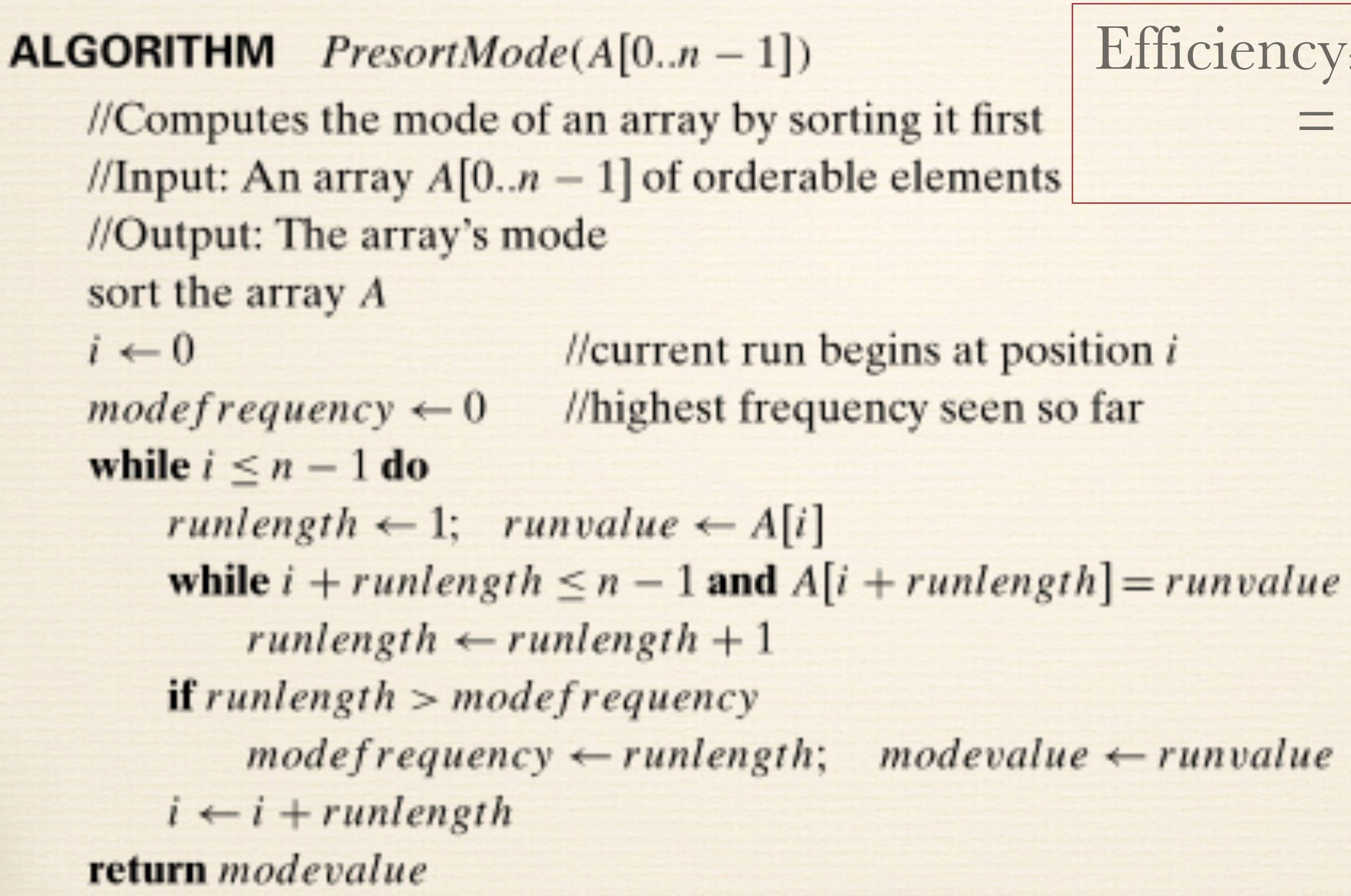
Efficiency of algorithms involving sorting depends on efficiency of sorting.

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).

Mode

A ***mode*** is a value that occurs most often in a given list of numbers. For example, for 5, 1, 5, 7, 6, 5, 7, the mode is 5.



```
ALGORITHM PresortMode(A[0..n − 1])
    //Computes the mode of an array by sorting it first
    //Input: An array A[0..n − 1] of orderable elements
    //Output: The array's mode
    sort the array A
    i ← 0                        //current run begins at position i
    modefrequency ← 0            //highest frequency seen so far
    while i ≤ n − 1 do
        runlength ← 1;   runvalue ← A[i]
        while i + runlength ≤ n − 1 and A[i + runlength] = runvalue
            runlength ← runlength + 1
        if runlength > modefrequency
            modefrequency ← runlength;   modevalue ← runvalue
        i ← i + runlength
    return modevalue
```

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

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

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?

Instance simplification – Gaussian Elimination

Solving 2 linear equations with 2 unknowns:

$2x - 3y = 3,\ 4x - 2y = 10$

$y = -(10 - 4x)/2$

$2x+3((10-4x)/2) = 3$

$x = 3,\ y = 1$

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.

Gaussian Elimination (cont.)

$$\begin{array}{l} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = b_n \end{array} \Longrightarrow \begin{array}{r} a'_{11}x_1 + a'_{12}x_2 + \cdots + a'_{1n}x_n = b'_1 \\ a'_{22}x_2 + \cdots + a'_{2n}x_n = b'_2 \\ \vdots \\ a'_{nn}x_n = b'_n. \end{array}$$

In matrix notations, we can write this as

$Ax=b \Rightarrow A'x=b'$,

where

$$A = \begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & & & \\ a_{n1} & a_{n2} & \ldots & a_{nn} \end{bmatrix}, \; b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}, \; A' = \begin{bmatrix} a'_{11} & a'_{12} & \ldots & a'_{1n} \\ 0 & a'_{22} & \ldots & a'_{2n} \\ \vdots & & & \\ 0 & 0 & \ldots & a'_{nn} \end{bmatrix}, \; b = \begin{bmatrix} b'_1 \\ b'_2 \\ \vdots \\ b'_n \end{bmatrix}$$

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

- Gaussian Elimination:
  - Make all x_1 coefficients zeros in the equations below the first one by a_{21}/a_{11}, a_{31}/a_{11}, … a_{n1}/a_{11}.
  - Make all x_2 coefficients zeros in the equations below the second one by a_{32}/a_{22}, a_{42}/a_{22}, … a_{n2}/a_{22}.
  - Repeat for each of the first n-1 variables to yield an upper-triangular coefficient matrix.
  - Back substitute from n-1 variable upwards.

Gaussian Elimination Example

$$\begin{aligned} 2x_1 - x_2 + x_3 &= 1 \\ 4x_1 + x_2 - x_3 &= 5 \\ x_1 + x_2 + x_3 &= 0. \end{aligned}$$

$$\begin{array}{cl} \begin{bmatrix} 2 & -1 & 1 & 1 \\ 4 & 1 & -1 & 5 \\ 1 & 1 & 1 & 0 \end{bmatrix} & \begin{array}{l} \\ \text{row } 2 - \frac{4}{2} \text{ row } 1 \\ \text{row } 3 - \frac{1}{2} \text{ row } 1 \end{array} \end{array}$$

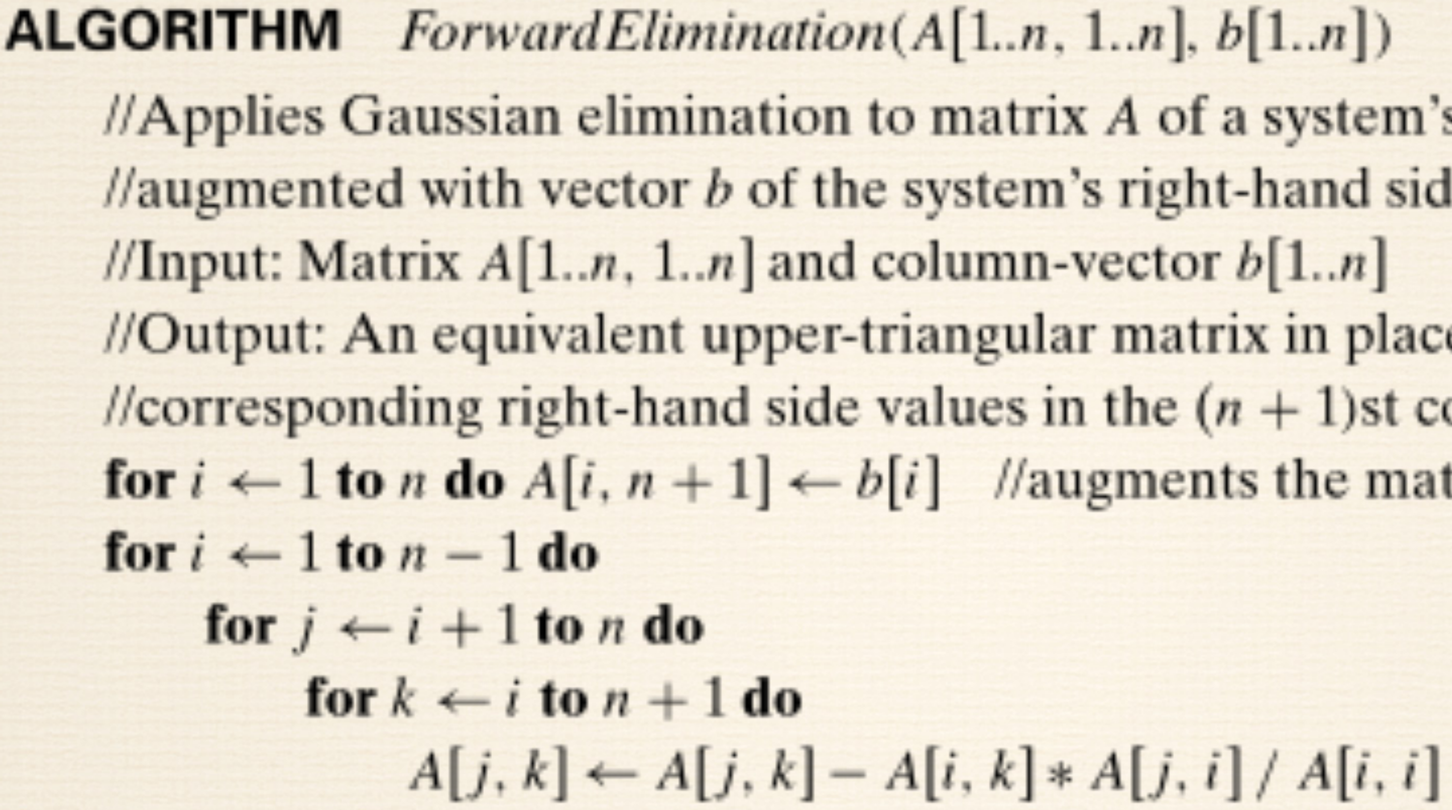
$$\begin{array}{cl} \begin{bmatrix} 2 & -1 & 1 & 1 \\ 0 & 3 & -3 & 3 \\ 0 & \frac{3}{2} & \frac{1}{2} & -\frac{1}{2} \end{bmatrix} & \begin{array}{l} \\ \\ \text{row } 3 - \frac{1}{2} \text{ row } 2 \end{array} \end{array}$$

$$\begin{bmatrix} 2 & -1 & 1 & 1 \\ 0 & 3 & -3 & 3 \\ 0 & 0 & 2 & -2 \end{bmatrix}$$

Now we can obtain the solution by back substitutions:

$x_3=(-2)/2=-1$, $x_2=(3-(-3)x_3)/3=0$, and $x_1=(1-x_3-(-1)x_2)/2=1$

Gaussian Elimination (Stage 1)



```
ALGORITHM ForwardElimination($A[1..n, 1..n], b[1..n]$)
    //Applies Gaussian elimination to matrix $A$ of a system's coefficients,
    //augmented with vector $b$ of the system's right-hand side values
    //Input: Matrix $A[1..n, 1..n]$ and column-vector $b[1..n]$
    //Output: An equivalent upper-triangular matrix in place of $A$ with the
    //corresponding right-hand side values in the $(n + 1)$st column
    for $i \leftarrow 1$ to $n$ do $A[i, n+1] \leftarrow b[i]$    //augments the 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
                $A[j, k] \leftarrow A[j, k] - A[i, k] * A[j, i] / A[i, i]$
```

Example: $2x_1 - 4x_2 + x_3 = 6$

$3x_1 - x_2 + x_3 = 11$

$x_1 + x_2 - x_3 = -3$

Gaussian Elimination (Stage 2)

```
Algorithm BackSubstitutions
  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: $\Theta(n^3) + \Theta(n^2) = \Theta(n^3)$

Example of Gaussian Elimination

Solve

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

- **Gaussian elimination**

$$\begin{pmatrix} 2 & -4 & 1 & 6 \\ 3 & -1 & 1 & 11 \\ 1 & 1 & -1 & -3 \end{pmatrix}$$

row2 – (3/2)*row1
row3 – (1/2)*row1

$$\begin{pmatrix} 2 & -4 & 1 & 6 \\ 0 & 5 & -1/2 & 2 \\ 0 & 3 & -3/2 & -6 \end{pmatrix}$$

row3–(3/5)*row2

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

2. **Backward substitution:**

$x_3 = (-36/5) / (-6/5) = 6$
$x_2 = (2+((1/2)*6)) / 5 = 1$
$x_1 = (6 - (1*6) + (4*1))/2 = 2$

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

Tree searching

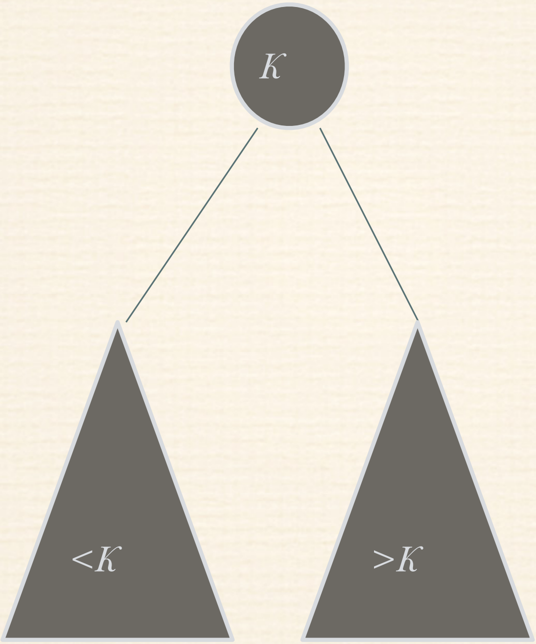
- binary search tree
- binary balanced trees: AVL trees, red-black trees
- multiway balanced trees: 2-3 trees, 2-3-4 trees, B trees

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

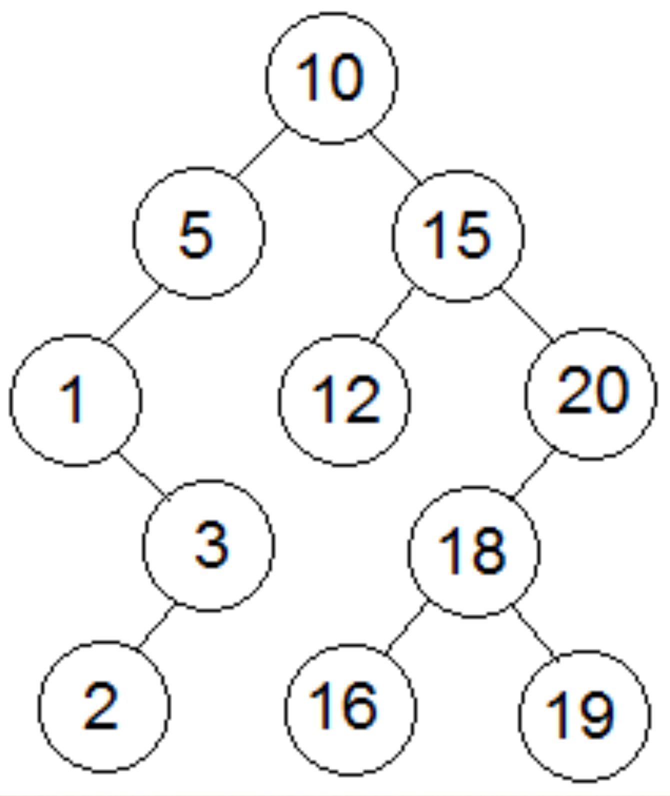
Efficiency depends of the tree's height: $\lfloor \log_2 n \rfloor \leq h \leq n\text{-}1$,
with height average (random files) be about $3\log_2 n$

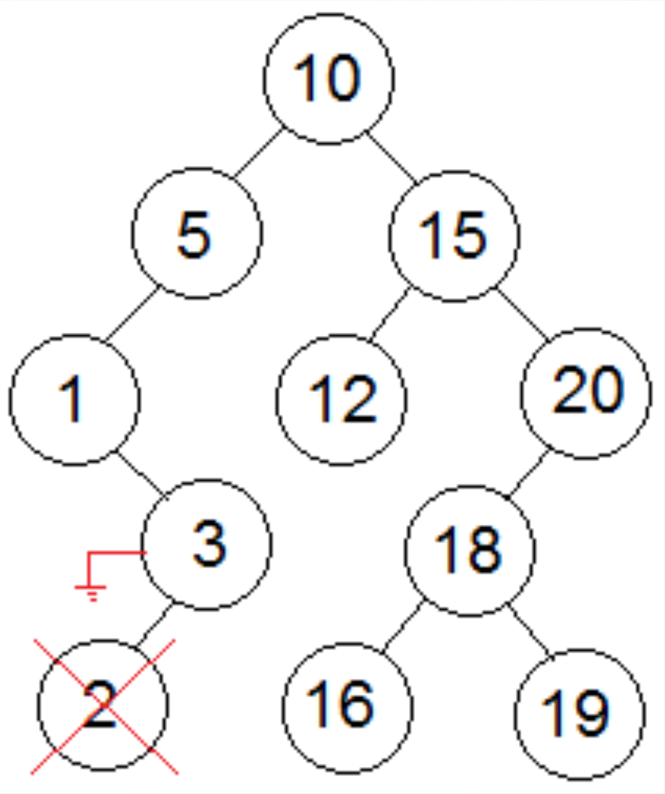
Thus all three operations have
worst case efficiency: $\Theta(n)$
average case efficiency: $\Theta(\log n)$

Bonus: inorder traversal produces sorted list

Deletion from a Binary Search Tree

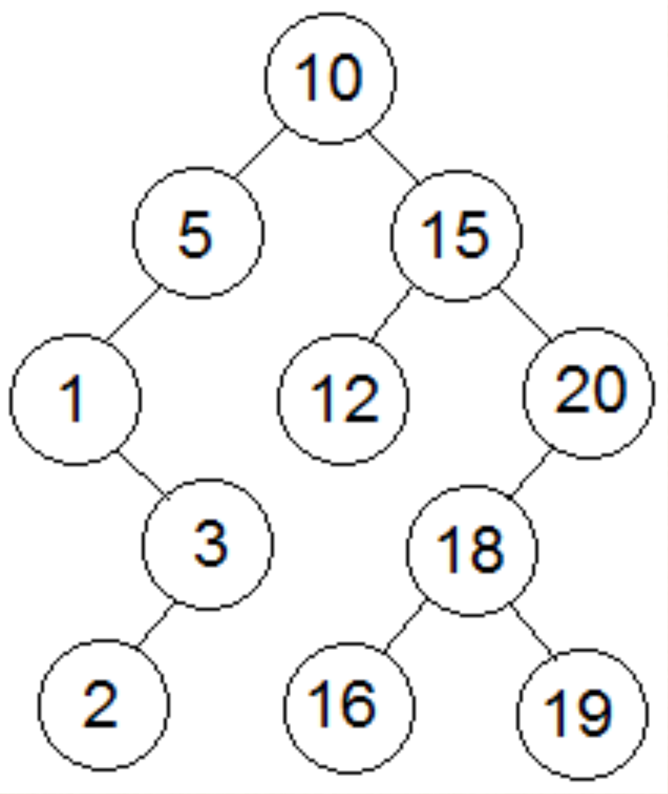
Delete node with no children

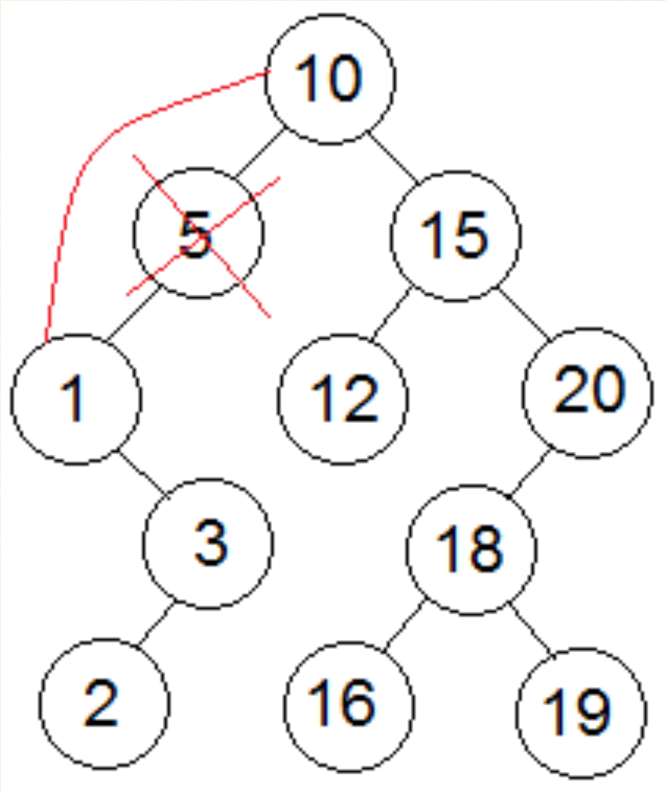




Deletion from a Binary Search Tree

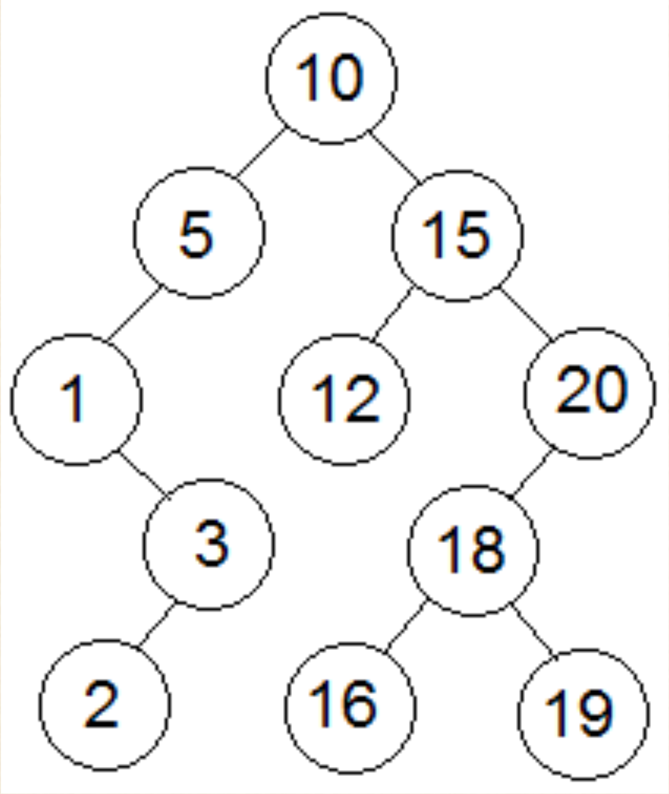
Delete node with one child

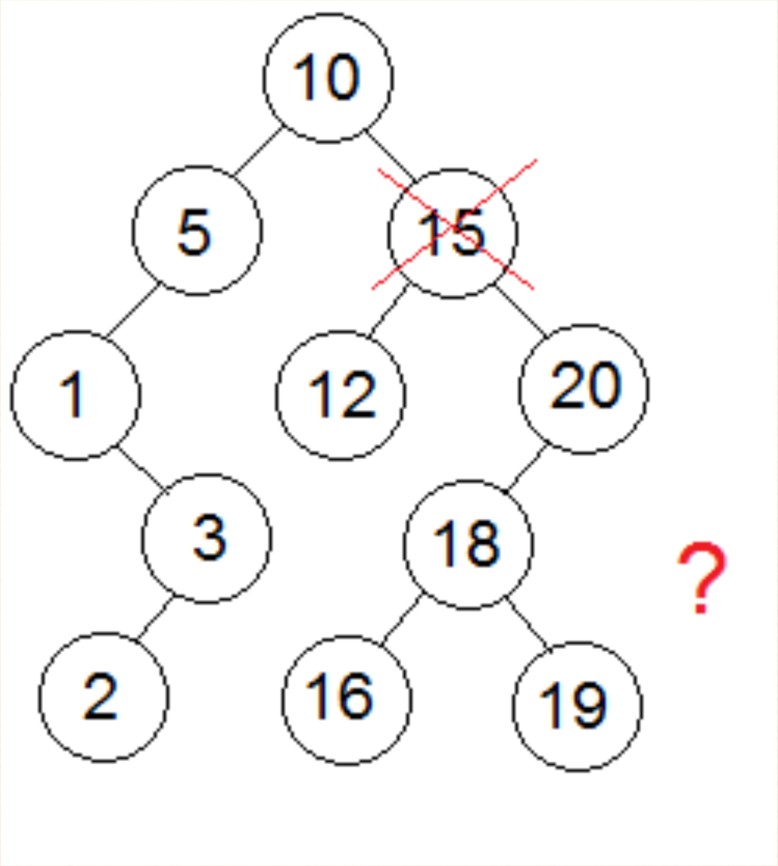




Deletion from a Binary Search Tree

Delete node with two children





Deletion from a Binary Search Tree

- How do we delete something from a binary search tree, ensuring that it remains a binary search tree? What is the complexity?
  - First we have to find the element to delete which is O(h), then it depends on how many children the node has.
    - 0 children (i.e. leaf) – Straightforward, just set pointer from parent to NULL and deallocate memory used for record of deleted node.
    - 1 child – Again straightforward, replace the deleted node with its single child.
    - 2 children – More complicated, replace the node with **its *in-order successor*** and delete the in-order successor (or replace the node with its ***in-order predecessor*** and delete the ***in-order predecessor***).

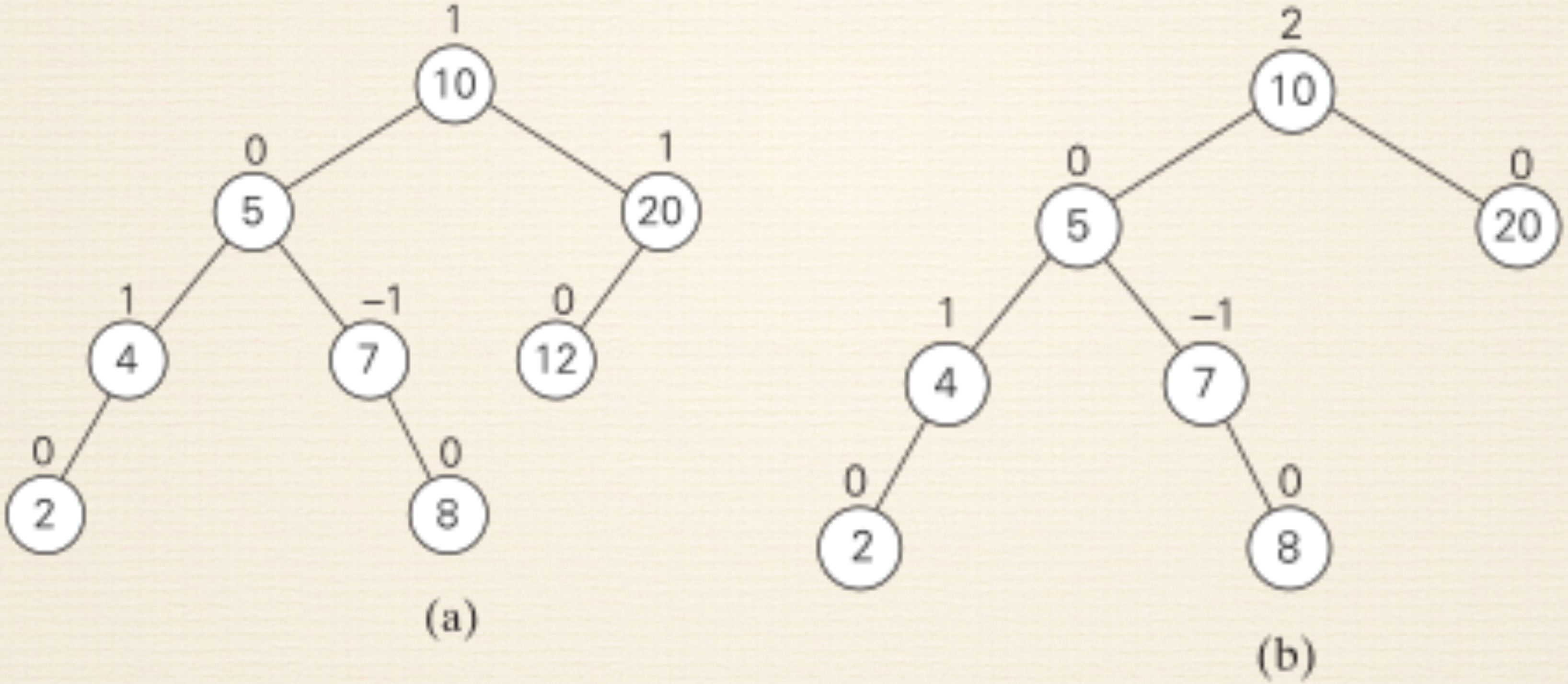
Balanced Search Trees

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

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)



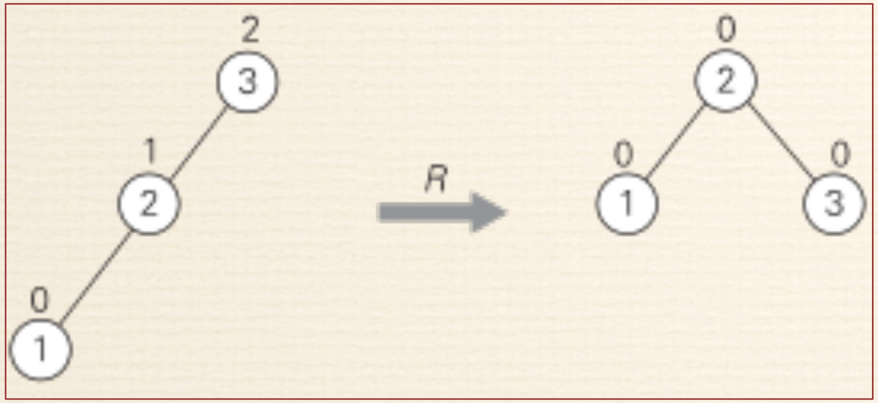
(a)

(b)

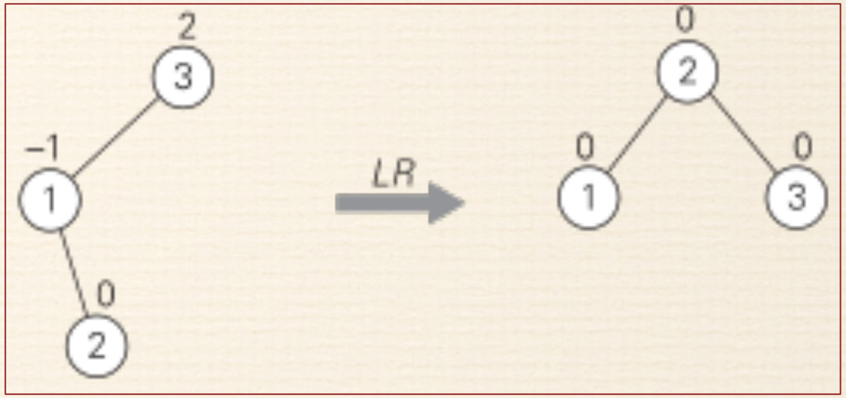
(a) AVL tree. (b) Binary search tree that is not an AVL tree. The numbers above the nodes indicate the nodes' balance factors.

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

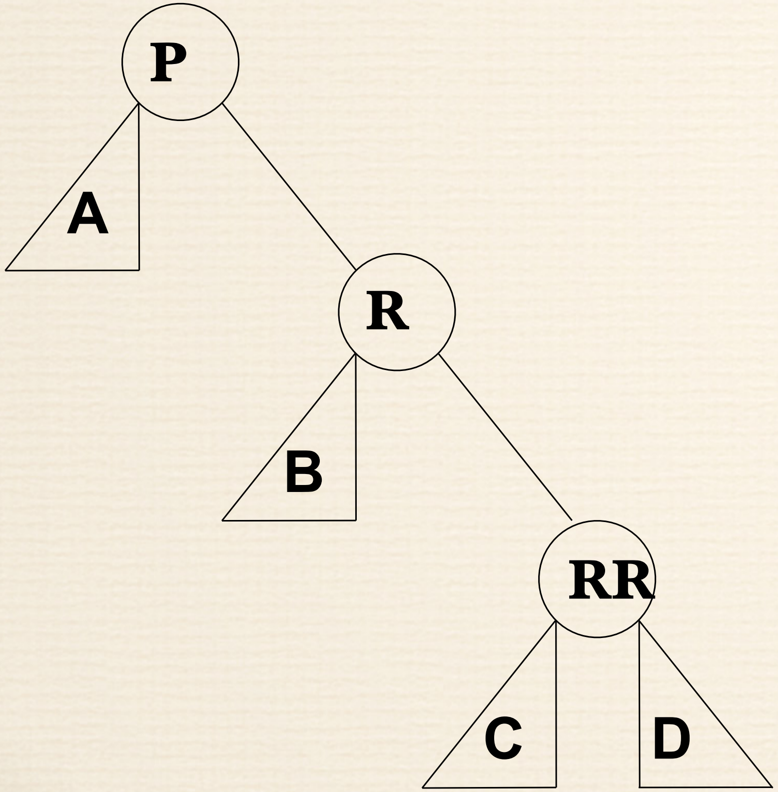


Double LR-rotation, L-rotation on the left subtree of the root, then R-rotation on the root

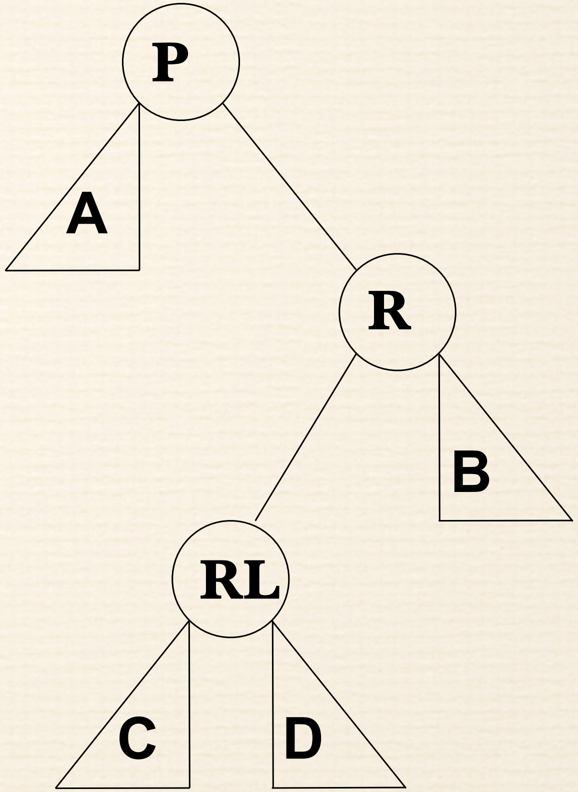
AVL Tree Rebalancing

Consider a subtree with root node P which has balance factor 2

Case 1 (Right right case):



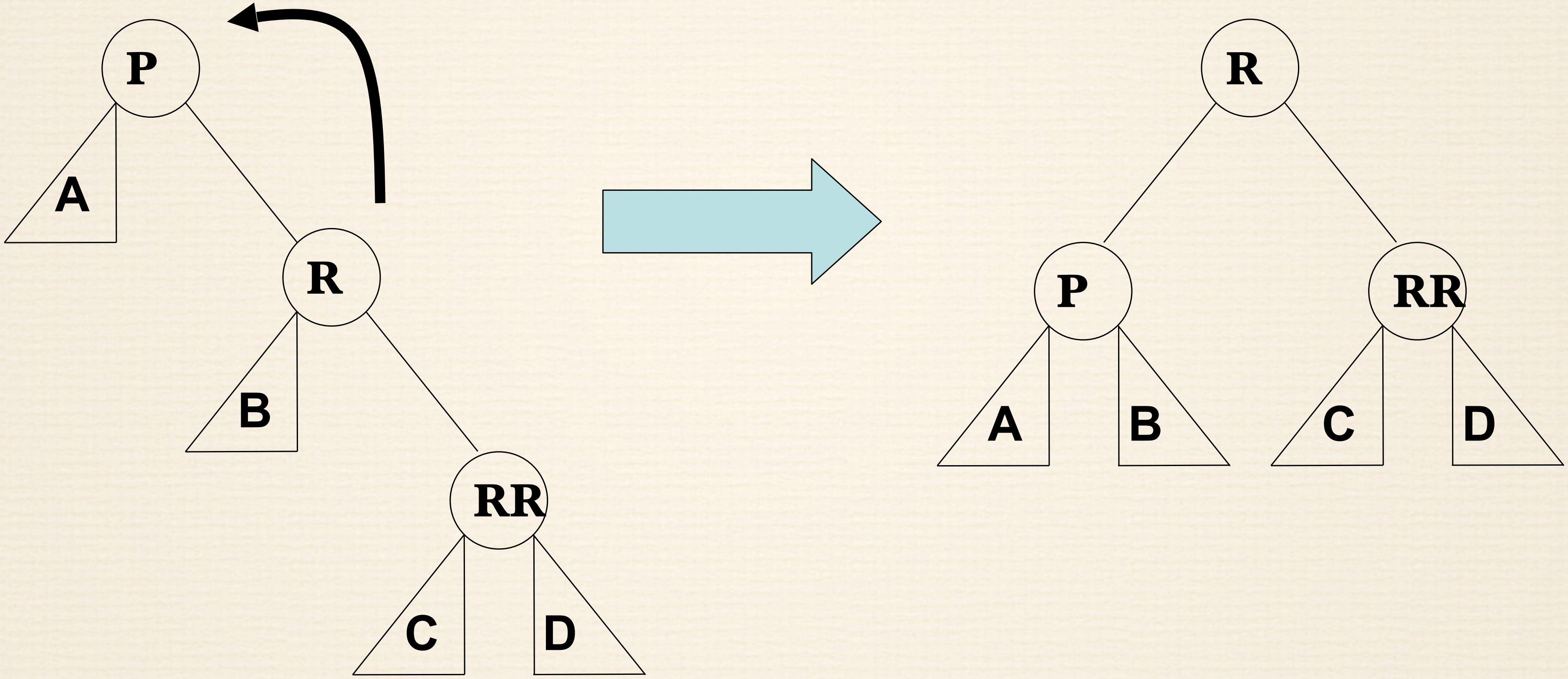
Case 2 (Right left case):



(A, B, C and D are perfectly balanced subtrees)

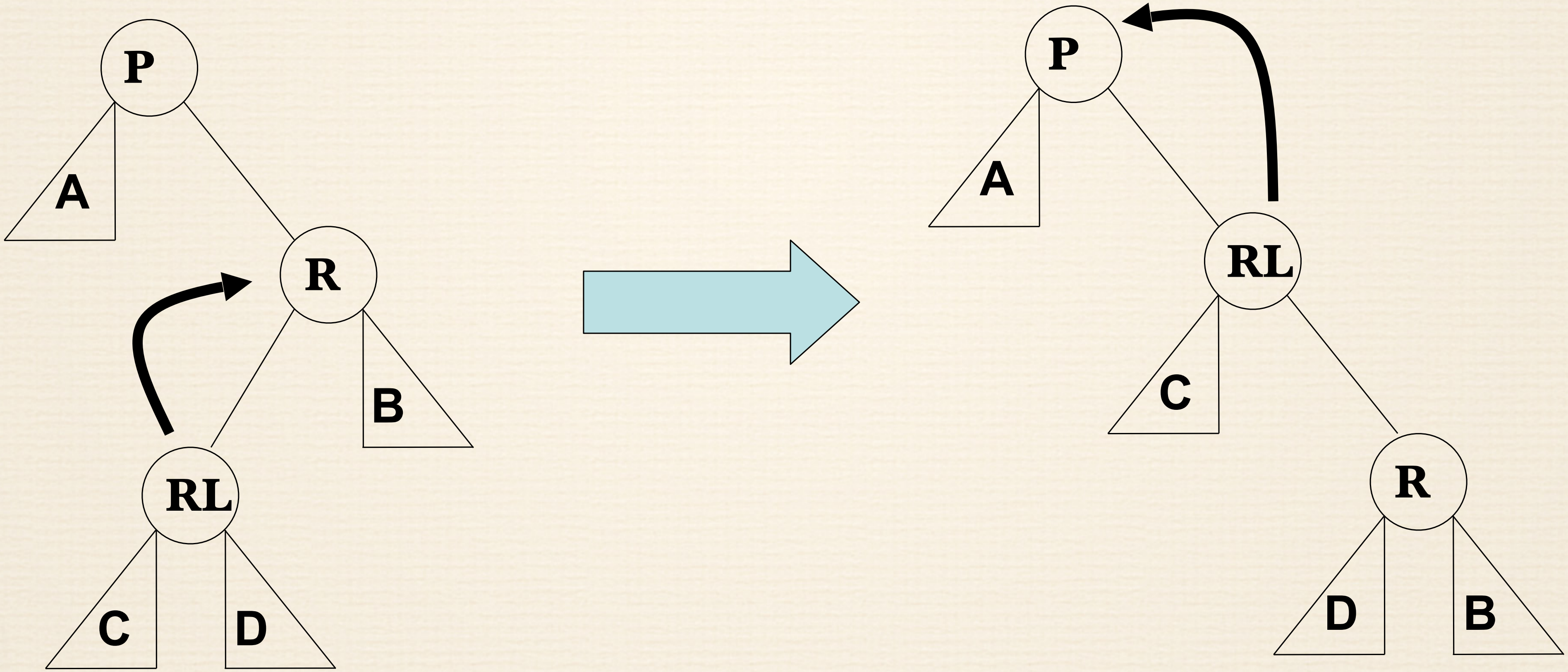
AVL Tree Rebalancing: Right Right Case

The right right case is fixed with a single left rotation about R:



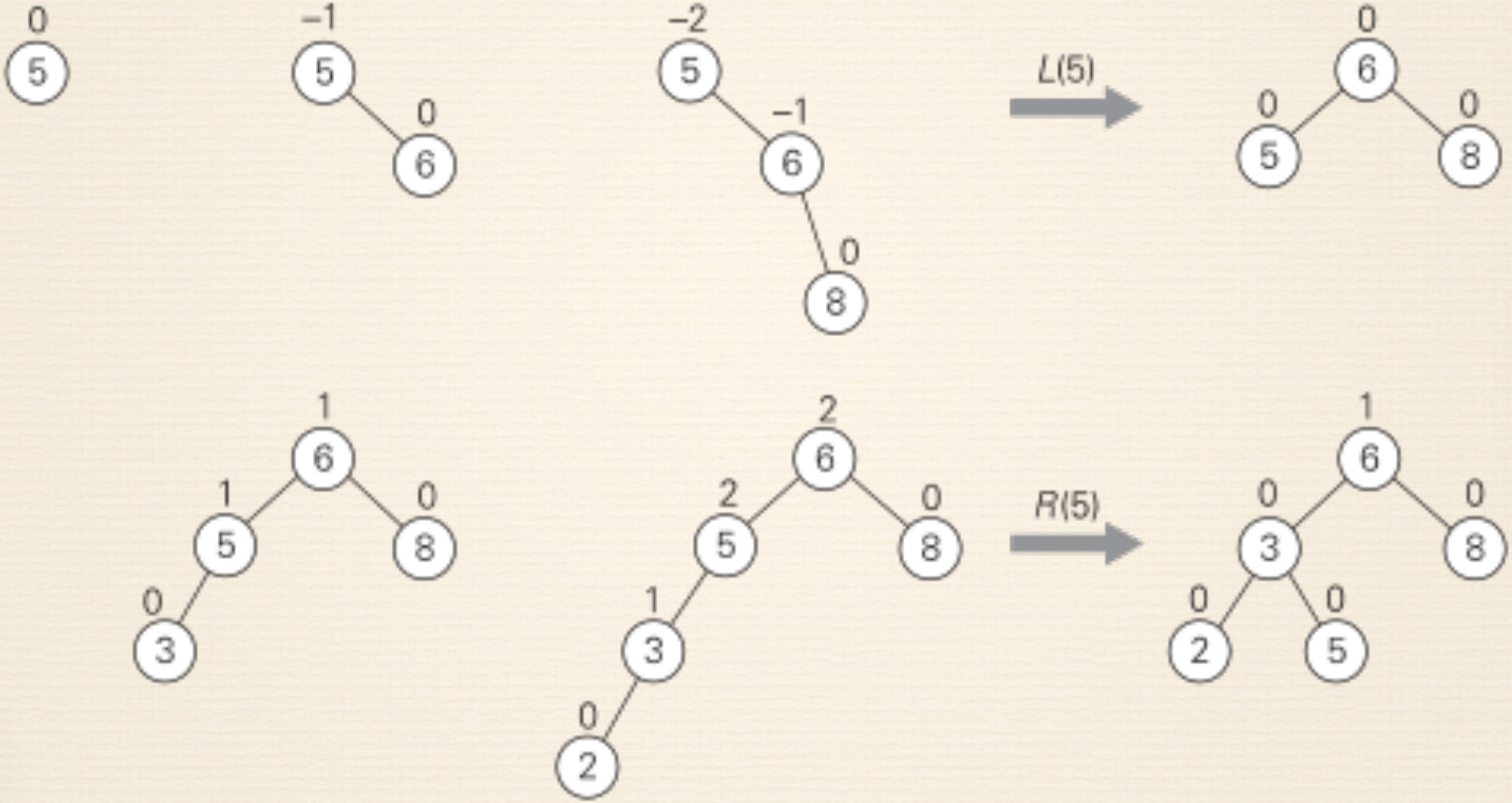
AVL Tree Rebalancing: Right left Case

The right left case is fixed with a right rotation about RL which converts the tree to a right right case, fixed as on previous slide by a further rotation:

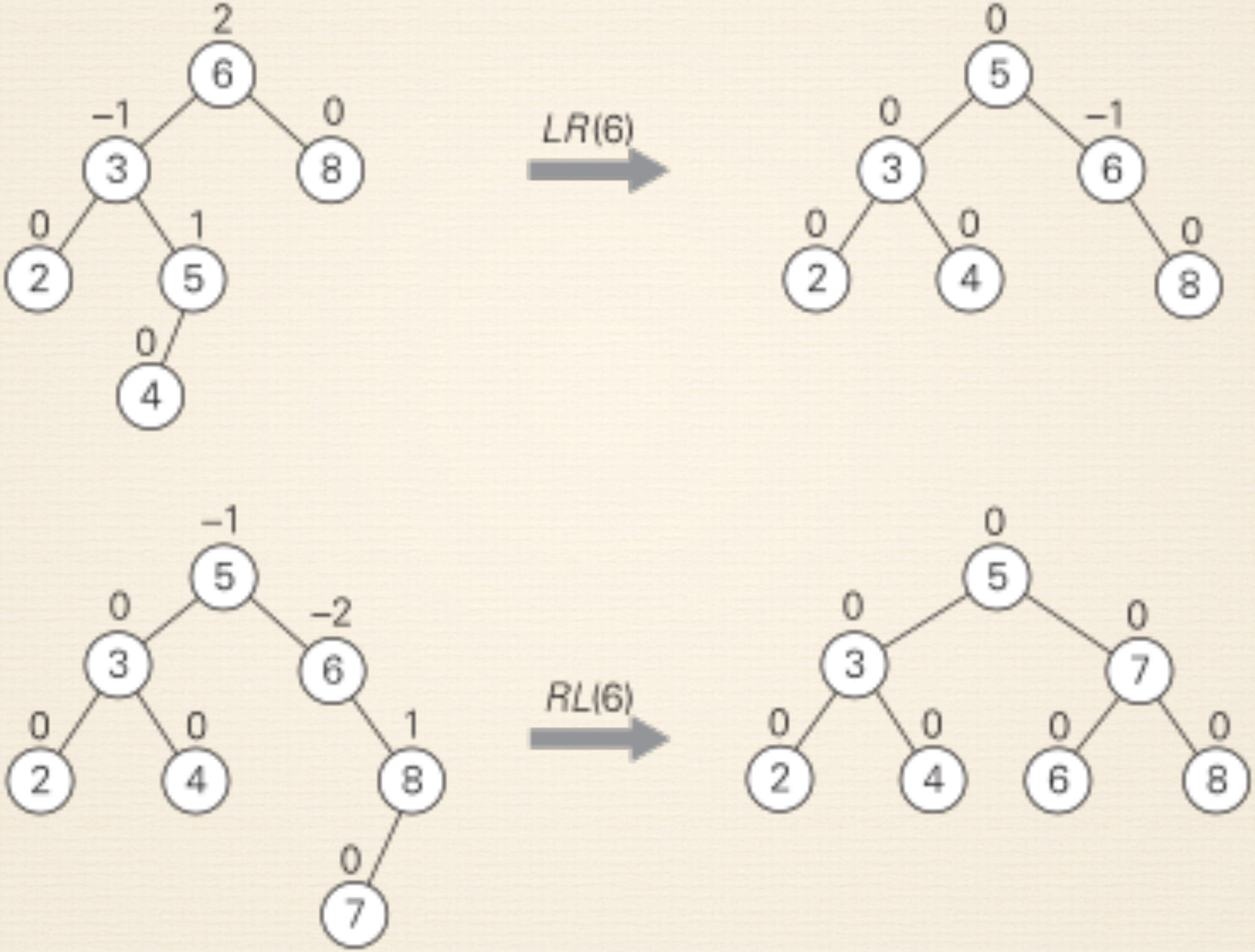


AVL tree construction - an example

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



AVL tree construction - an example (cont.)



Analysis of AVL trees

$\lfloor \log_2 n \rfloor \leq h < 1.4405 \log_2(n+2) - 1.3277$

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:

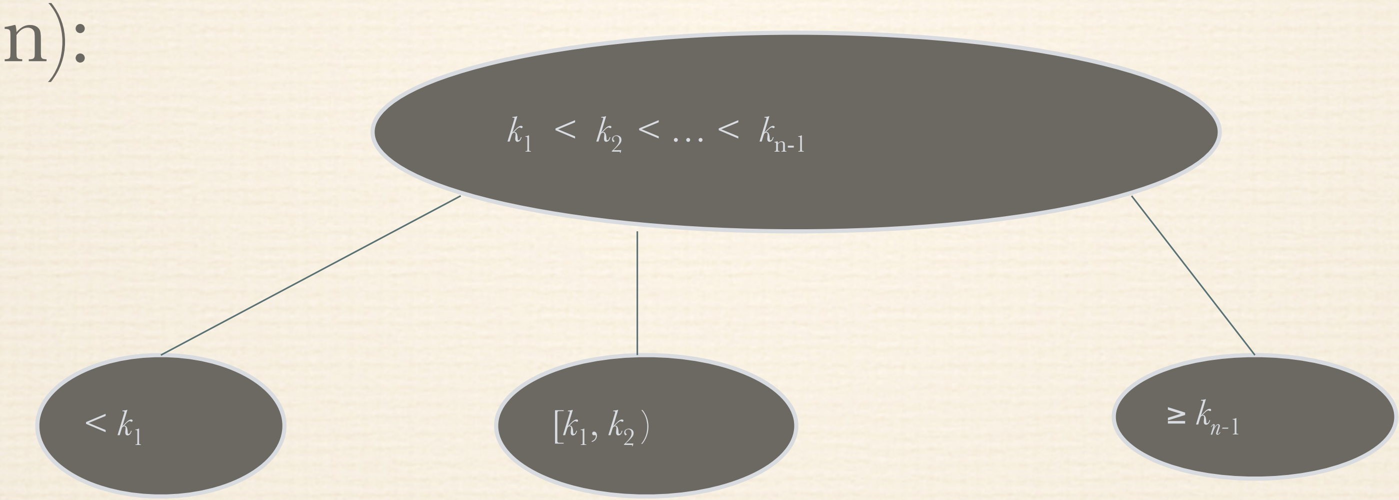
- frequent rotations
- complexity

A similar idea: *red-black trees* (height of subtrees is allowed to differ by up to a factor of 2)

Multiway Search Trees

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):



$k_1 < k_2 < \ldots < k_{\text{n-1}}$

$< k_1$

$[k_1, k_2)$

$\geq k_{n\text{-}1}$

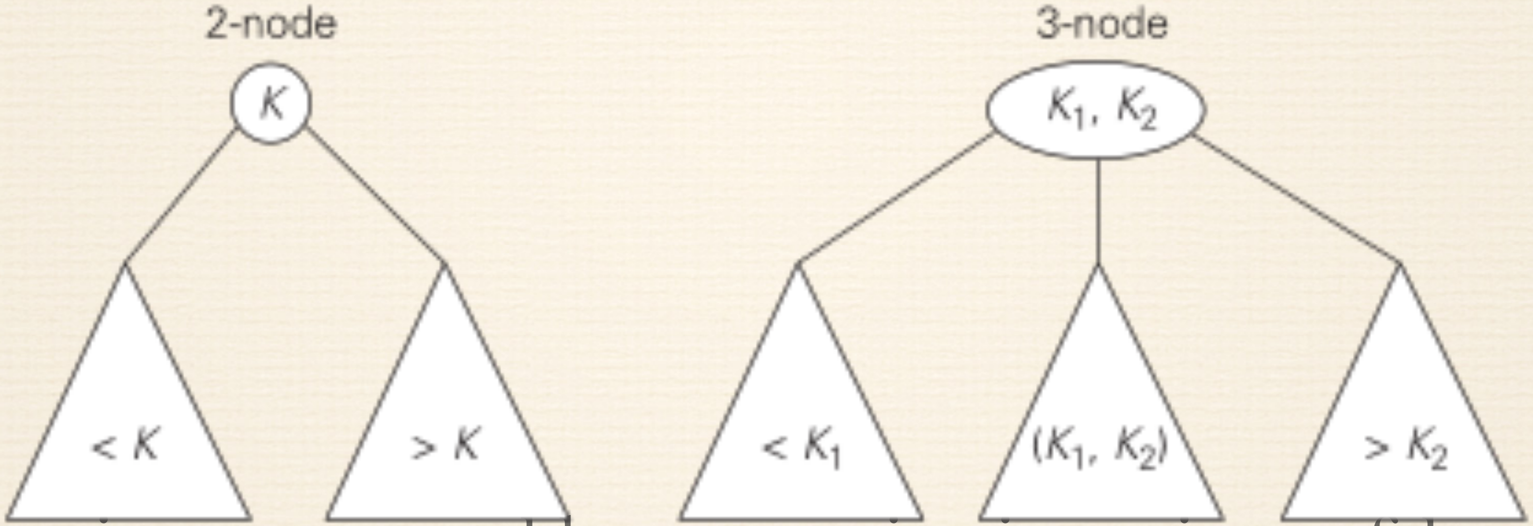
Note: Every node in a classical binary search tree is a 2-node

2-3 Tree

Definition A *2-3 tree* is a search tree that

may have 2-nodes and 3-nodes

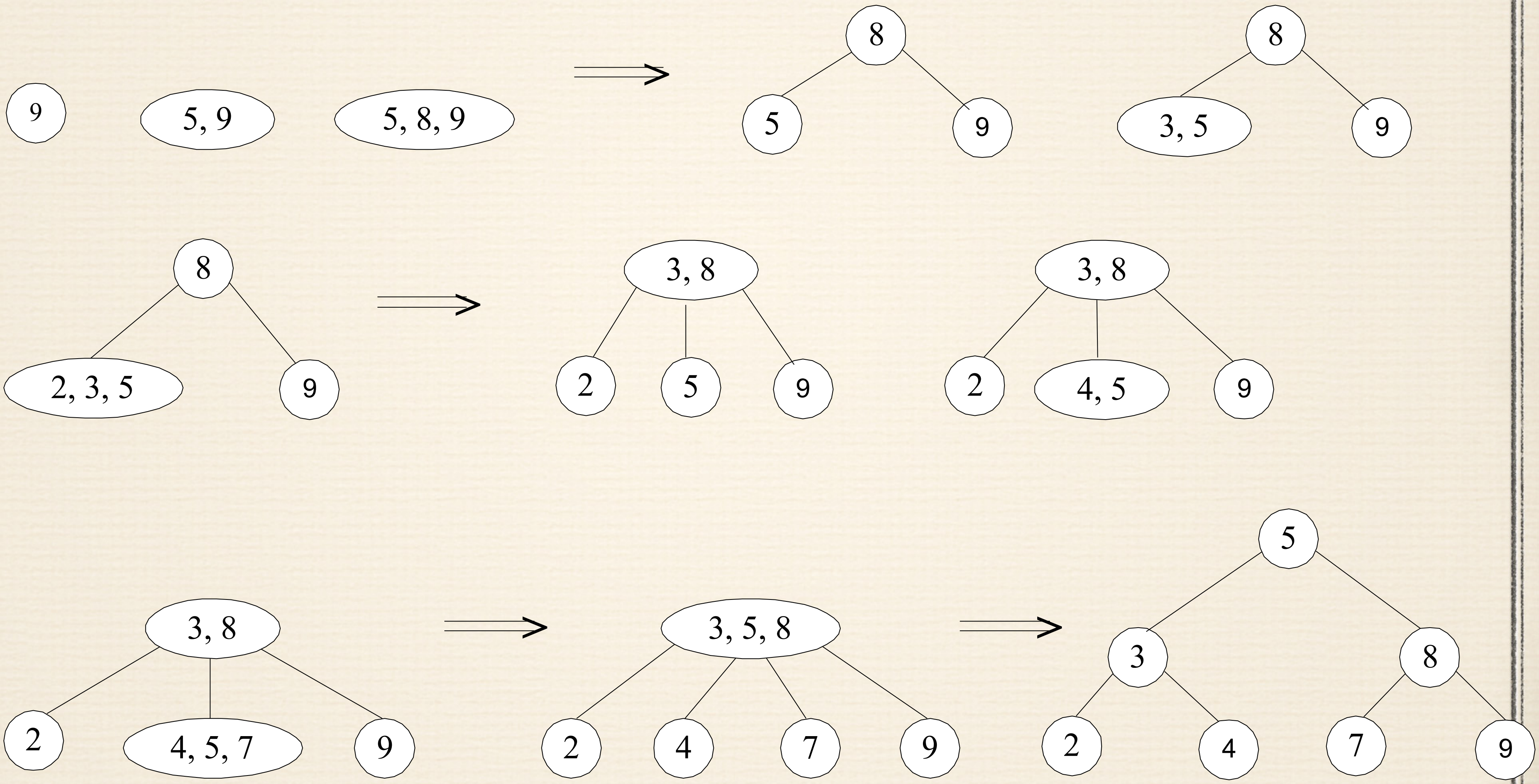
height-balanced (all leaves are on the same level)



A 2-3 tree is constructed by successive insertions of keys given, with a new key always inserted into a leaf of the tree. 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

$\log_3(n+1) - 1 \le h \le \log_2(n+1) - 1$

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***