DSE 2256 DESIGN & ANALYSIS OF ALGORITHMS

Lecture 22, 23, 24

Transform-and-Conquer:
Introduction
Presorting
Binary Search Trees

Recap of L21

- Multiplication of Large Integers

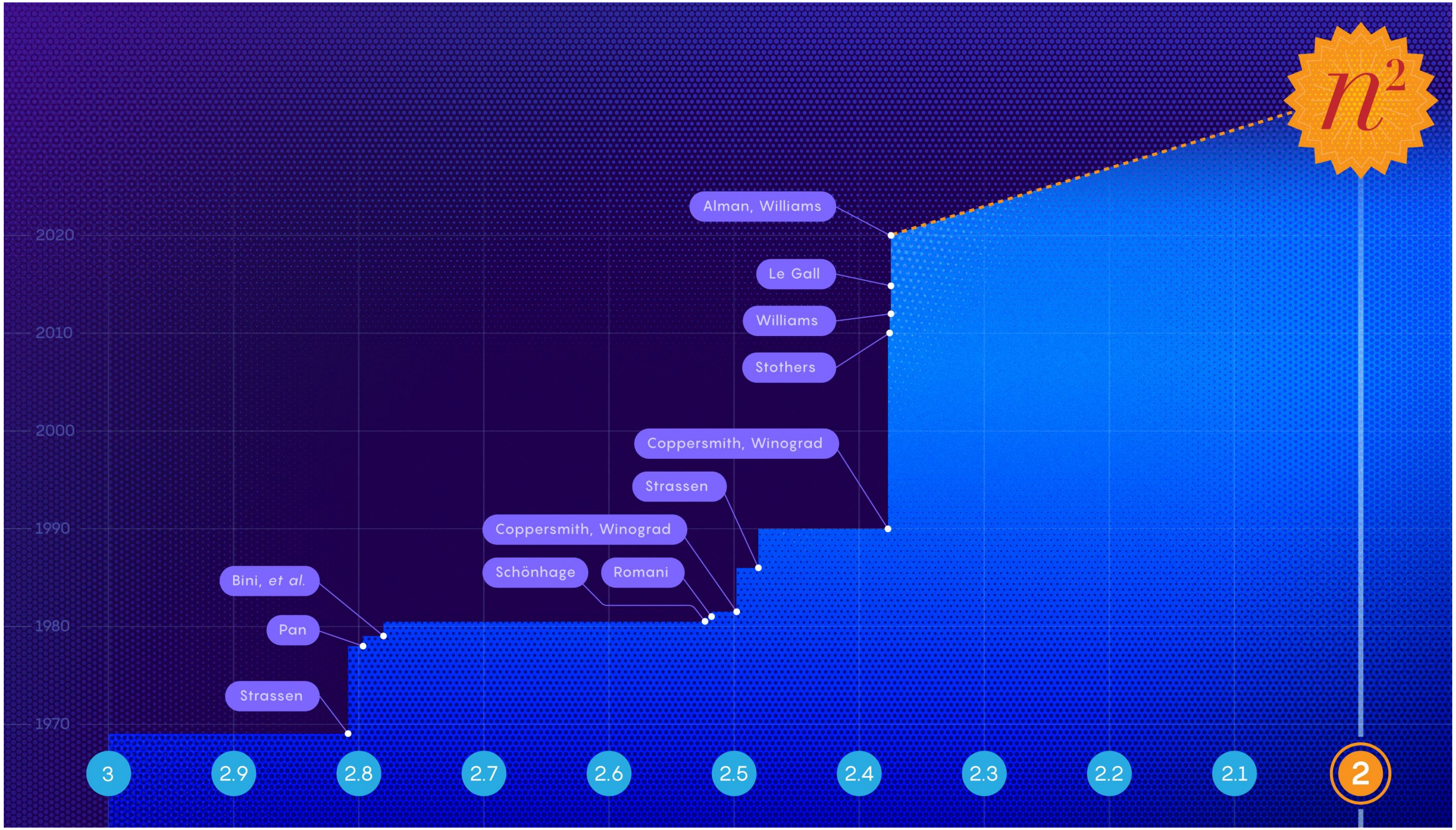
Recurrence Relation for multiplication operation:

$M(n) = 3M(n/2) \approx \Theta(n^{1.585})$

- Strassen's Matrix Multiplication

Recurrence Relation for multiplication operation:

$M(n) = 7M(n/2) \approx \Theta(n^{2.807})$



Source: https://www.quantamagazine.org/mathematicians-inch-closer-to-matrix-multiplication-goal-20210323/

Recap of L21: Solution to the exercise

$$\begin{bmatrix} 1 & 0 & 2 & 1 \\ 4 & 1 & 1 & 0 \\ 0 & 1 & 3 & 0 \\ 5 & 0 & 2 & 1 \end{bmatrix} * \begin{bmatrix} 0 & 1 & 0 & 1 \\ 2 & 1 & 0 & 1 \\ 2 & 0 & 1 & 1 \\ 1 & 3 & 5 & 0 \end{bmatrix}$$

For the matrices given, Strassen's algorithm yields the following:

$$C = \left[\begin{array}{c|c} C_{00} & C_{01} \\ \hline C_{10} & C_{11} \end{array}\right] = \left[\begin{array}{c|c} A_{00} & A_{01} \\ \hline A_{10} & A_{11} \end{array}\right] \left[\begin{array}{c|c} B_{00} & B_{01} \\ \hline B_{10} & B_{11} \end{array}\right]$$

where

$$A_{00} = \begin{bmatrix} 1 & 0 \\ 4 & 1 \end{bmatrix}, \quad A_{01} = \begin{bmatrix} 2 & 1 \\ 1 & 0 \end{bmatrix}, \quad A_{10} = \begin{bmatrix} 0 & 1 \\ 5 & 0 \end{bmatrix}, \quad A_{11} = \begin{bmatrix} 3 & 0 \\ 2 & 1 \end{bmatrix},$$

$$B_{00} = \begin{bmatrix} 0 & 1 \\ 2 & 1 \end{bmatrix}, \quad B_{01} = \begin{bmatrix} 0 & 1 \\ 0 & 4 \end{bmatrix}, \quad B_{10} = \begin{bmatrix} 2 & 0 \\ 1 & 3 \end{bmatrix}, \quad B_{11} = \begin{bmatrix} 1 & 1 \\ 5 & 0 \end{bmatrix}.$$

Therefore,

$$\begin{aligned}
M_1 &= (A_{00} + A_{11})(B_{00} + B_{11}) = \begin{bmatrix} 4 & 0 \\ 6 & 2 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 7 & 1 \end{bmatrix} = \begin{bmatrix} 4 & 8 \\ 20 & 14 \end{bmatrix}, \\
M_2 &= (A_{10} + A_{11})B_{00} = \begin{bmatrix} 3 & 1 \\ 7 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 4 \\ 2 & 8 \end{bmatrix}, \\
M_3 &= A_{00}(B_{01} - B_{11}) = \begin{bmatrix} 1 & 0 \\ 4 & 1 \end{bmatrix} \begin{bmatrix} -1 & 0 \\ -5 & 4 \end{bmatrix} = \begin{bmatrix} -1 & 0 \\ -9 & 4 \end{bmatrix}, \\
M_4 &= A_{11}(B_{10} - B_{00}) = \begin{bmatrix} 3 & 0 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} 2 & -1 \\ -1 & 2 \end{bmatrix} = \begin{bmatrix} 6 & -3 \\ 3 & 0 \end{bmatrix}, \\
M_5 &= (A_{00} + A_{01})B_{11} = \begin{bmatrix} 3 & 1 \\ 5 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 5 & 0 \end{bmatrix} = \begin{bmatrix} 8 & 3 \\ 10 & 5 \end{bmatrix}, \\
M_6 &= (A_{10} - A_{00})(B_{00} + B_{01}) = \begin{bmatrix} -1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 0 & 2 \\ 2 & 5 \end{bmatrix} = \begin{bmatrix} 2 & 3 \\ -2 & -3 \end{bmatrix}, \\
M_7 &= (A_{01} - A_{11})(B_{10} + B_{11}) = \begin{bmatrix} -1 & 1 \\ -1 & -1 \end{bmatrix} \begin{bmatrix} 3 & 1 \\ 6 & 3 \end{bmatrix} = \begin{bmatrix} 3 & 2 \\ -9 & -4 \end{bmatrix}.
\end{aligned}$$

Recap of L21: Solution to the exercise

Accordingly,

$$\begin{aligned}
C_{00} &= M_1 + M_4 - M_5 + M_7 \\
&= \begin{bmatrix} 4 & 8 \\ 20 & 14 \end{bmatrix} + \begin{bmatrix} 6 & -3 \\ 3 & 0 \end{bmatrix} - \begin{bmatrix} 8 & 3 \\ 10 & 5 \end{bmatrix} + \begin{bmatrix} 3 & 2 \\ -9 & -4 \end{bmatrix} = \begin{bmatrix} 5 & 4 \\ 4 & 5 \end{bmatrix}, \\
C_{01} &= M_3 + M_5 \\
&= \begin{bmatrix} -1 & 0 \\ -9 & 4 \end{bmatrix} + \begin{bmatrix} 8 & 3 \\ 10 & 5 \end{bmatrix} = \begin{bmatrix} 7 & 3 \\ 1 & 9 \end{bmatrix}, \\
C_{10} &= M_2 + M_4 \\
&= \begin{bmatrix} 2 & 4 \\ 2 & 8 \end{bmatrix} + \begin{bmatrix} 6 & -3 \\ 3 & 0 \end{bmatrix} = \begin{bmatrix} 8 & 1 \\ 5 & 8 \end{bmatrix}, \\
C_{11} &= M_1 + M_3 - M_2 + M_6 \\
&= \begin{bmatrix} 4 & 8 \\ 20 & 14 \end{bmatrix} + \begin{bmatrix} -1 & 0 \\ -9 & 4 \end{bmatrix} - \begin{bmatrix} 2 & 4 \\ 2 & 8 \end{bmatrix} + \begin{bmatrix} 2 & 3 \\ -2 & -3 \end{bmatrix} = \begin{bmatrix} 3 & 7 \\ 7 & 7 \end{bmatrix}.
\end{aligned}$$

That is,

$$C = \begin{bmatrix} 5 & 4 & 7 & 3 \\ 4 & 5 & 1 & 9 \\ 8 & 1 & 3 & 7 \\ 5 & 8 & 7 & 7 \end{bmatrix}.$$

Transform-and-Conquer

- This group of techniques solves a problem by a transformation.

- Such a strategy works in two-stages:
  - Transformation stage: Here, the problem's instance is modified to be more amenable to solution.
  - Conquering stage: Here, the problem is solved.

Transform-and-Conquer

- **There are three major variations of Transform-and-Conquer strategy that differ by what we transform a given instance to:**

    1. **To a simpler/more convenient instance of the same problem (Instance Simplification)**

    2. **To a different representation of the same instance (Representation Change)**

    3. **To a different problem for which an algorithm is already available (Problem Reduction)**

Instance simplification - Presorting

1. **Presorting**

- **Many problems involving lists are easier when list is sorted.**
  1. - **Searching**
  2. - **Computing the median (selection problem)**
  3. - **Checking if all elements are distinct (element uniqueness)**
- **Efficiency of algorithms involving sorting depends on efficiency of sorting.**

Searching with presorting

<u>Problem:</u> 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

<u>Time complexity:</u>

$$T(n) = T_{sort}(n) + T_{binary\_search}(n)$$

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

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 <u>adjacent</u> elements

<u>Time complexity:</u>

$T(n) = T_{sort}(n) + T_{scan}(n)$

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

Without sorting => Θ (n^2)

```
ALGORITHM  PresortElementUniqueness($A[0..n-1]$)
    //Solves the element uniqueness problem by sorting the array first
    //Input: An array $A[0..n-1]$ of orderable elements
    //Output: Returns “true” if $A$ has no equal elements, “false” otherwise
    sort the array $A$
    for $i \leftarrow 0$ to $n-2$ do
        if $A[i] = A[i+1]$ return false
    return true
```

```
ALGORITHM  UniqueElements($A[0..n-1]$)
    //Determines whether all the elements in a given array are distinct
    //Input: An array $A[0..n-1]$
    //Output: Returns “true” if all the elements in $A$ are distinct
    //          and “false” otherwise
    for $i \leftarrow 0$ to $n-2$ do
        for $j \leftarrow i+1$ to $n-1$ do
            if $A[i] = A[j]$ return false
    return true
```

Computing mode with presorting

```
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
    $i \leftarrow 0$                    //current run begins at position i
    $modefrequency \leftarrow 0$        //highest frequency seen so far
    while $i \le n - 1$ do
        $runlength \leftarrow 1$;   $runvalue \leftarrow A[i]$
        while $i + runlength \le n - 1$ and $A[i + runlength] = runvalue$
            $runlength \leftarrow runlength + 1$
        if $runlength > modefrequency$
            $modefrequency \leftarrow runlength$;   $modevalue \leftarrow runvalue$
        $i \leftarrow i + runlength$
    return modevalue
```

Without sorting => $\boxed{\Theta(n^2)}$

$$C(n) = \sum_{i=1}^{n}(i-1) = 0 + 1 + \cdots + (n-1) = \frac{(n-1)n}{2} \in \Theta(n^2).$$

Time complexity for computing mode with sorting:

$$T(n) = T_{sort}(n) + T_{mode}(n)$$

$$= \underline{\Theta(n \log n)} + \boxed{O(n)} = \underline{\Theta(n \log n)}$$

```
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
```

$4 > 1$

$3 > 4$ ×

$1 > 4$ ×

0	1	2	3	4	5	6	7	8	9	10	11
1	2	3	3	3	3	4	4	4	5	8	10

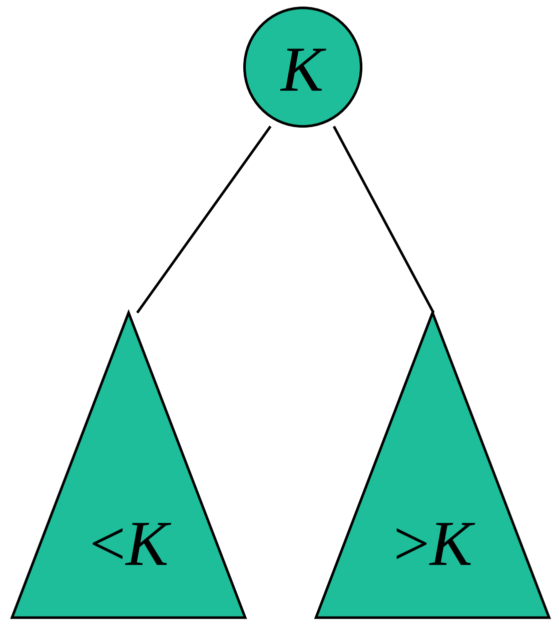
i	rl	rv	mf	mv
0	1	1	0	1
1	1	2	1	1
2	4	3	4	3
6	3	4	4	3
9	1	5	4	3
10	1	8	4	3
11	1	10	4	3

Taxonomy of Searching Algorithms

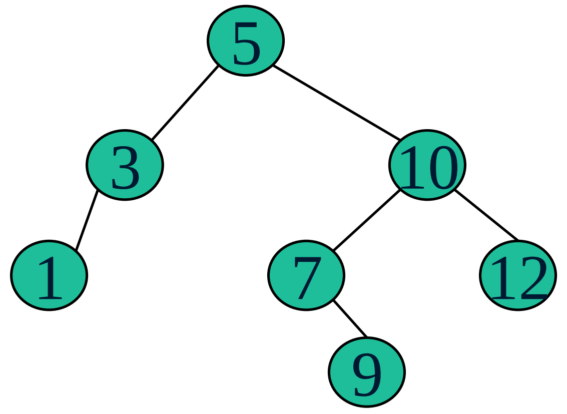
- **List searching (good for static data)**
  - Sequential search
  - Binary search
- **Tree searching (good for dynamic data)**
  - Binary search tree
  - Binary balanced trees: AVL trees, red-black trees
  - Multiway balanced trees: 2-3 trees, 2-3-4 trees, B trees
- **Hashing (good on average case)**
  - Open hashing (separate chaining)
  - Closed hashing (open addressing)

Binary Search Tree

- It is a binary tree whose nodes contain elements of a set of orderable items, one element per node, so that all elements in the left subtree are smaller than the element in the subtree's root, and all the elements in the right subtree are greater than it.



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



Balanced Search Trees

- Attractiveness of binary search tree is marred by the bad (linear) worst-case efficiency.

  Two ideas to overcome it are:

  1. To rebalance binary search tree when a new insertion makes the tree "too unbalanced"
     - AVL trees (G. M. Adelson-Velsky and E. M. Landis -1962)
     - Red-black trees

     Instance simplification

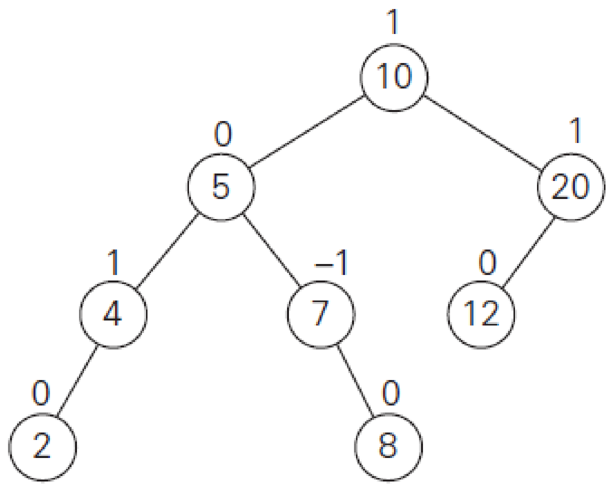
  2. To allow more than one element in a node of a search tree
     - 2-3 trees
     - 2-3-4 trees
     - B-trees

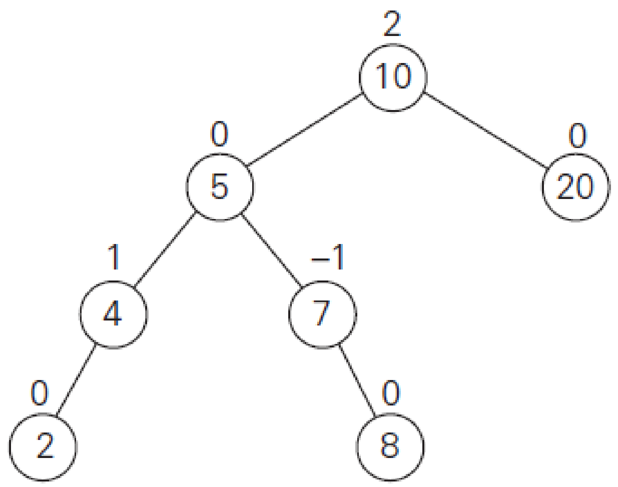
     Representation change

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 define

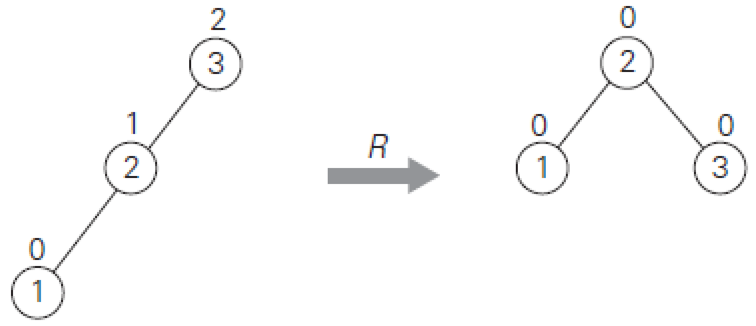




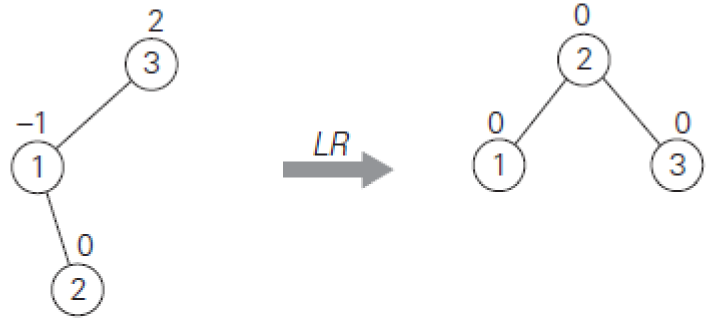
Tree (a) is an AVL tree; Tree (b) is not an AVL tree

Rotations

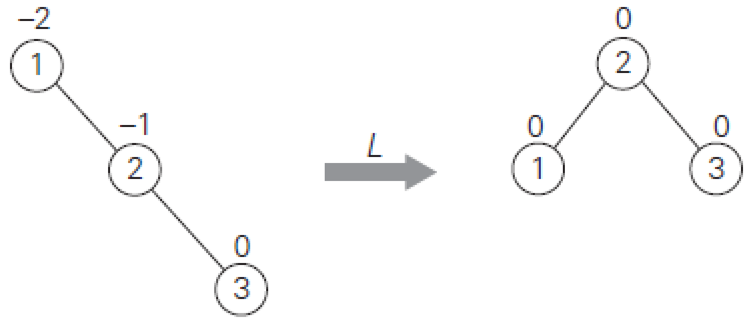
If a key insertion violates the balance requirement at some node, the subtree rooted at that node is transformed via one of the four rotations. (The rotation is always performed for a subtree rooted at an “unbalanced” node closest to the new leaf.)



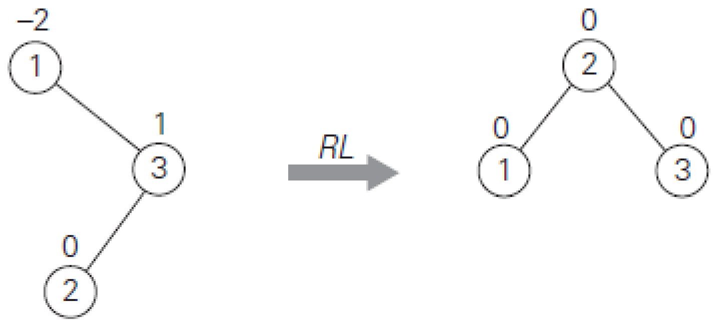
(a)



(c)

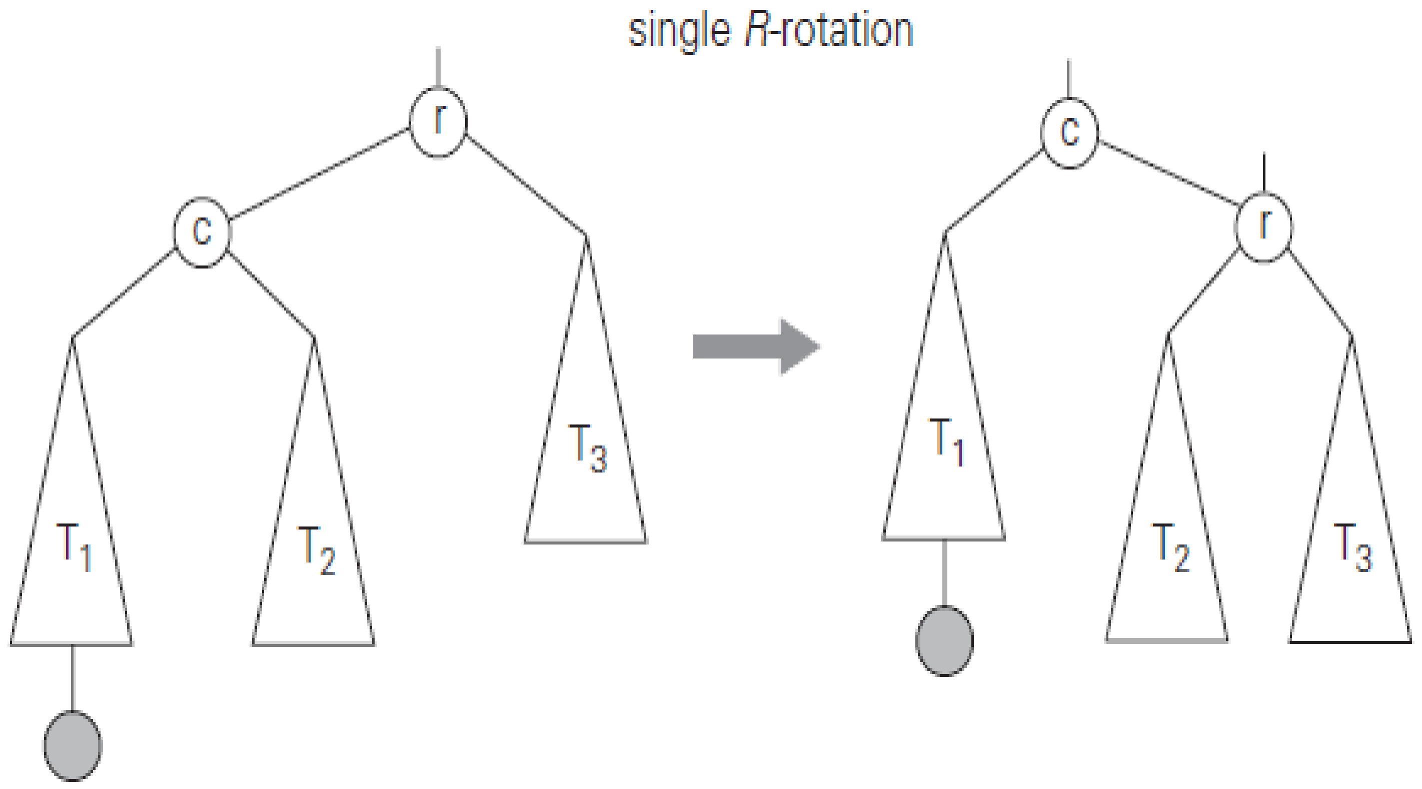


(b)

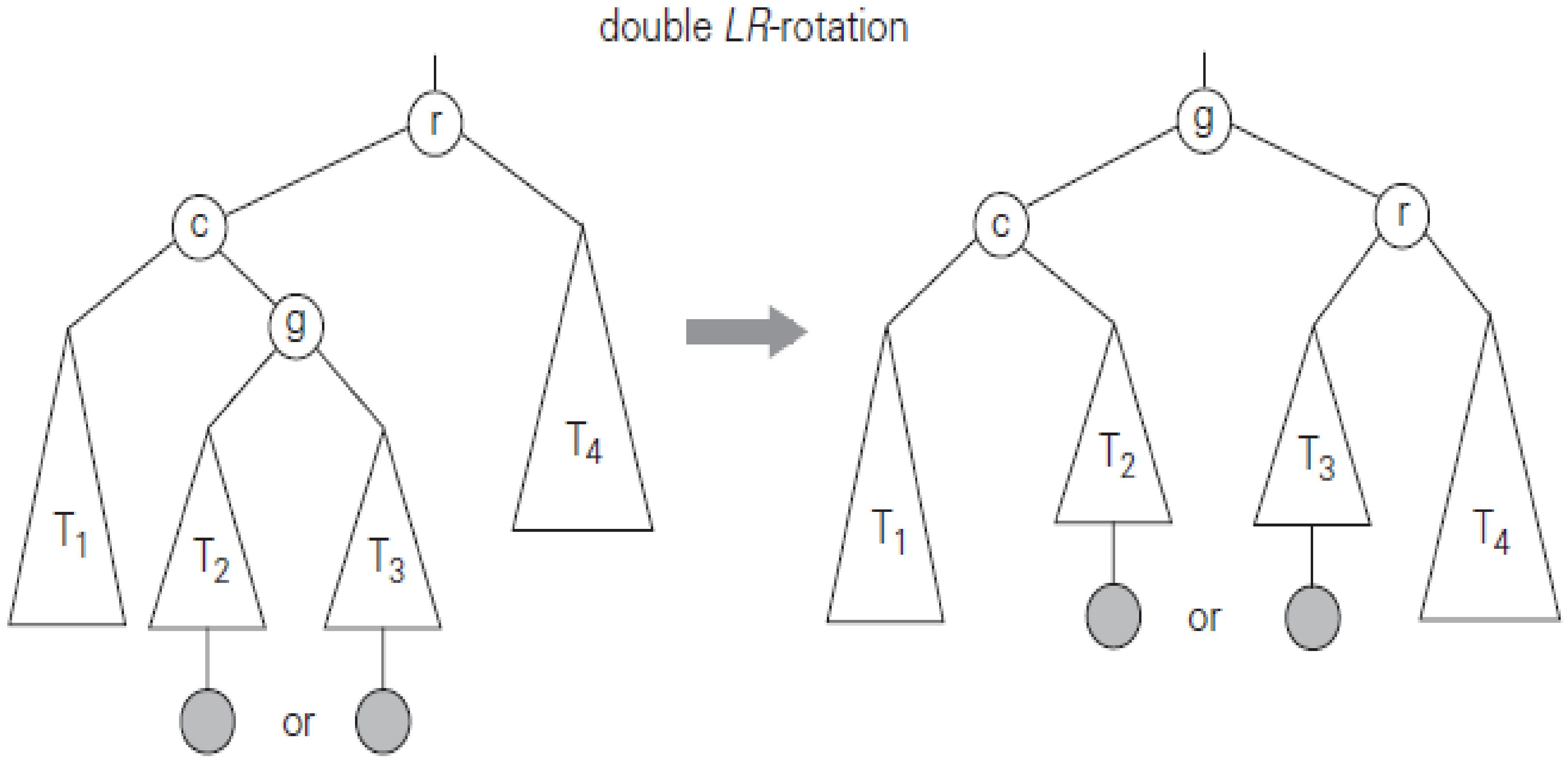


(d)

General case: Single R-rotation

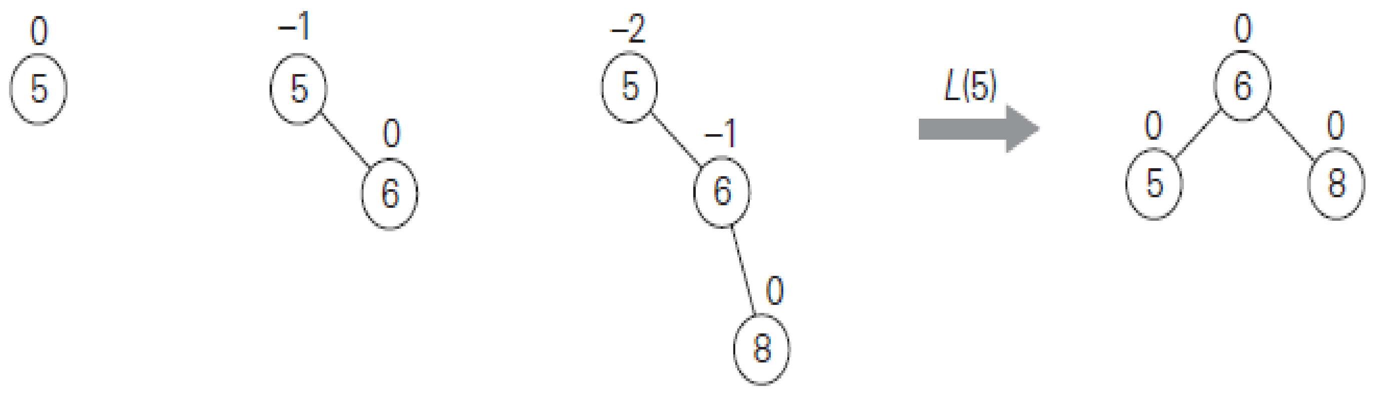


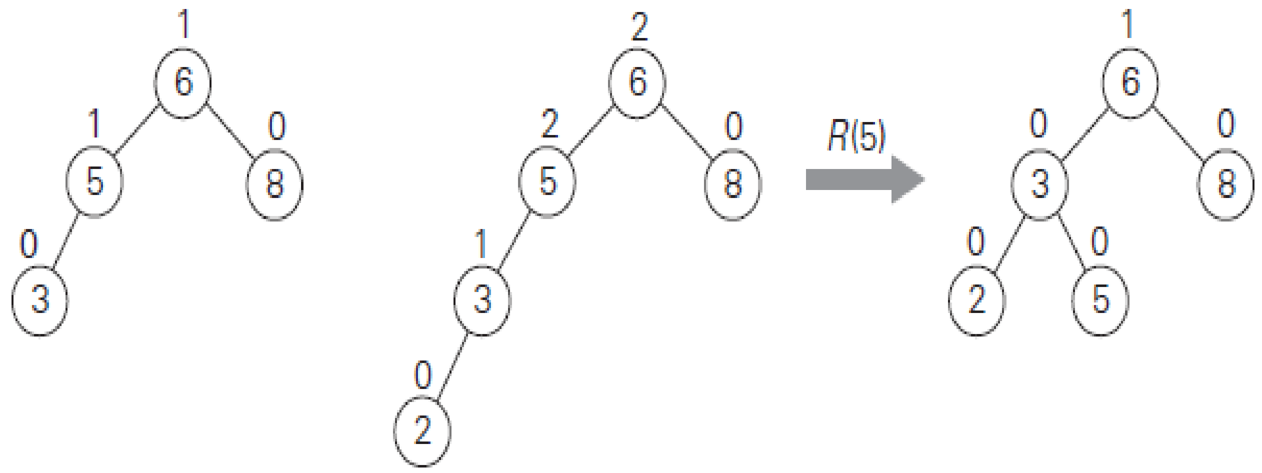
General case: Double LR-rotation



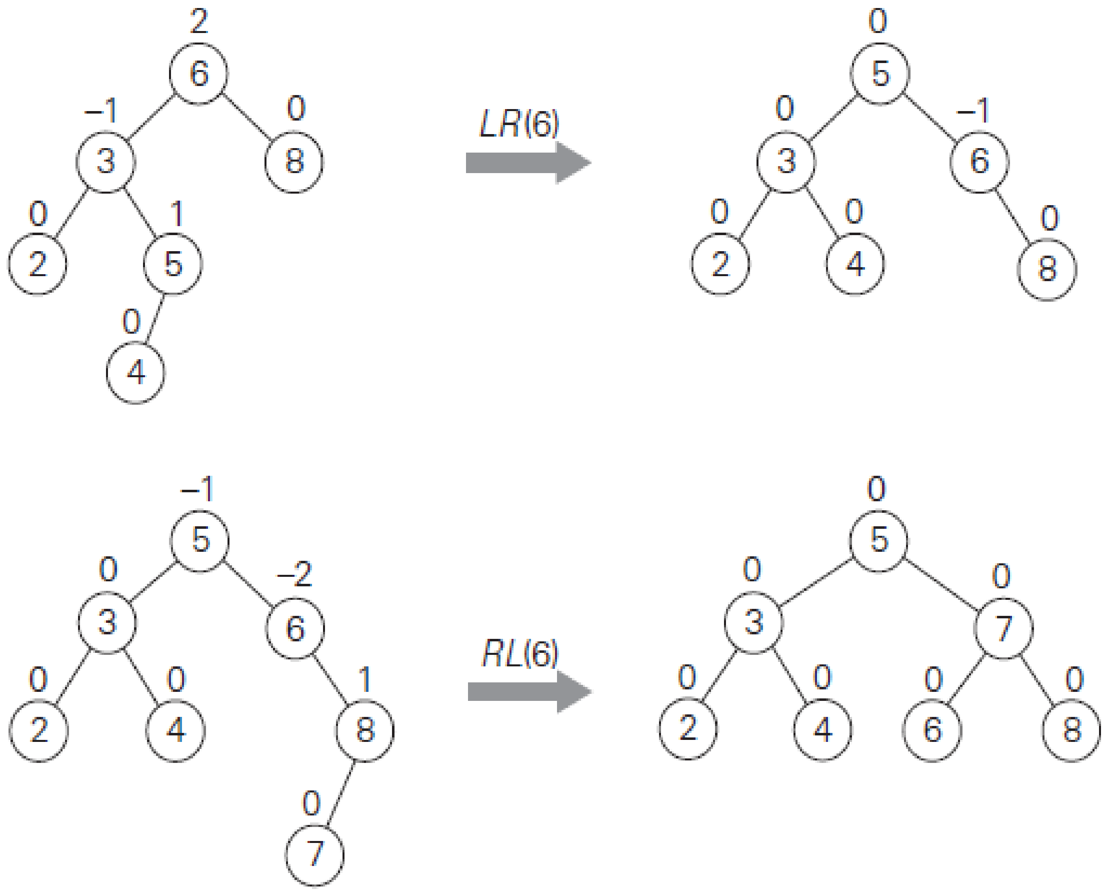
AVL tree construction – an example

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





AVL tree construction – an example



AVL tree: Analysis

- Height of an AVL tree with *n* nodes and height *h* satisfies the following :

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

- Search and insertion are O(log *n*)
- Deletion is more complicated but is also O(log *n*)

Disadvantages:

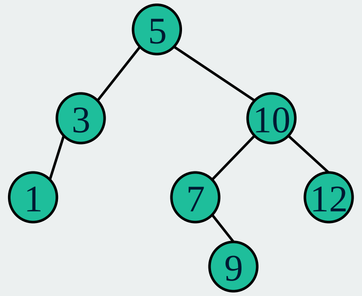
- Frequent rotations
- Complexity

Note:

- Height of a tree is:
    - The length of the longest path from the root to the leaf.

      OR
    - The maximum level of a tree.
- Example:

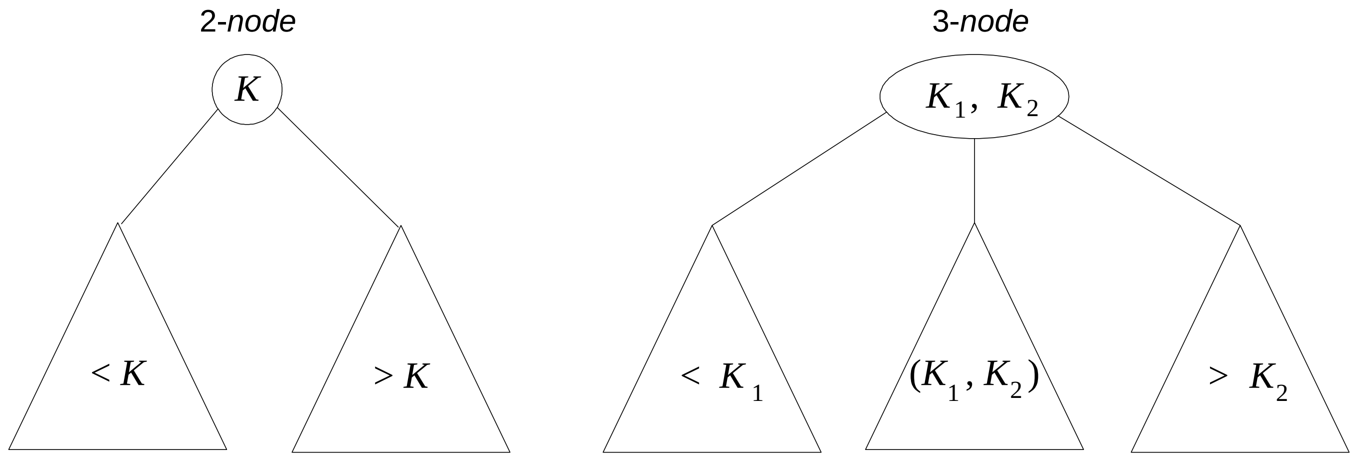
  Height of the following tree is **3**



2-3 Tree

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

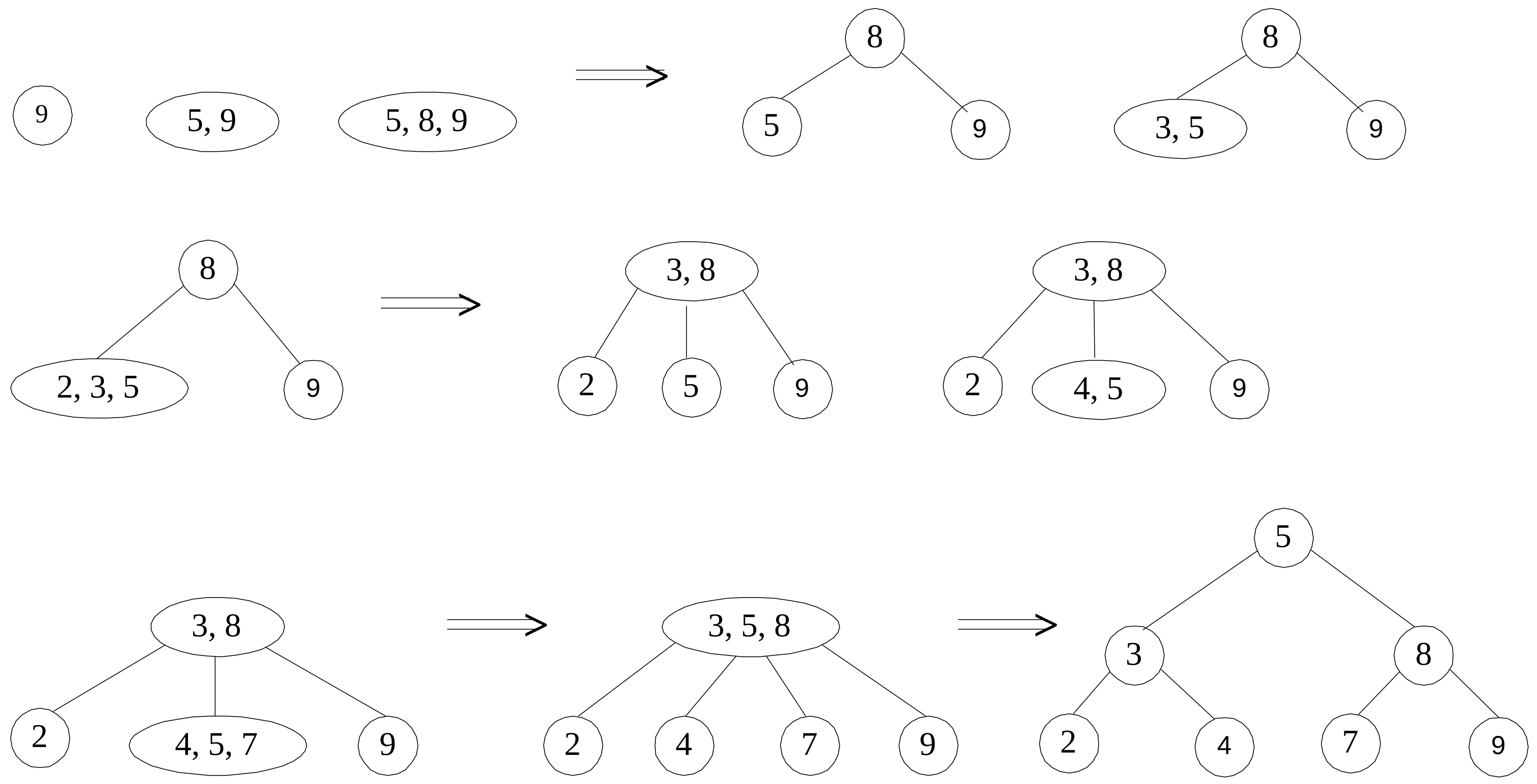
- **May have 2-nodes and 3-nodes.**
- **Height-balanced (all leaves are on the same level).**



- **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

Construction of a 2-3 tree for the list 9, 5, 8, 3, 2, 4, 7.



Analysis of 2-3 trees

- **A 2-3 tree of height h with the smallest number of keys is a full tree of 2-nodes. Therefore, for any 2-3 tree of height h with n nodes, we get the inequality:**

$$n \geq 1 + 2 + \cdots + 2^h = 2^{h+1} - 1 \quad \Rightarrow \quad h \leq \log_2(n+1) - 1$$

- **On the other hand, a 2-3 tree of height h with the largest number of keys is a full tree of 3-nodes, each with two keys and three children. Therefore, for any 2-3 tree with n nodes:**

$$n \leq 2 \cdot 1 + 2 \cdot 3 + \cdots + 2 \cdot 3^h = 2(1 + 3 + \cdots + 3^h) = 3^{h+1} - 1 \quad \Rightarrow \quad h \geq \log_3(n+1) - 1$$

- **Thus, the lower and upper bounds on height h :**

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

Time efficiencies

- **The time efficiencies of searching, insertion, and deletion are all in theta(log *n)* in both the worst and average case**

Analysis of 2-3 trees

- **Search, insertion, and deletion are in θ(log n)**

- **The idea of 2-3 tree can be generalized by allowing more keys per node**
  - **2-3-4 trees**
  - **B-trees**

Thank you!

Any queries?