Topic 4

Transform-and-Conquer

مشكلة مانعرف نحلها
نحولها لمشكلة أخرى
أسهل حلها.

Transform and Conquer

تحويلها

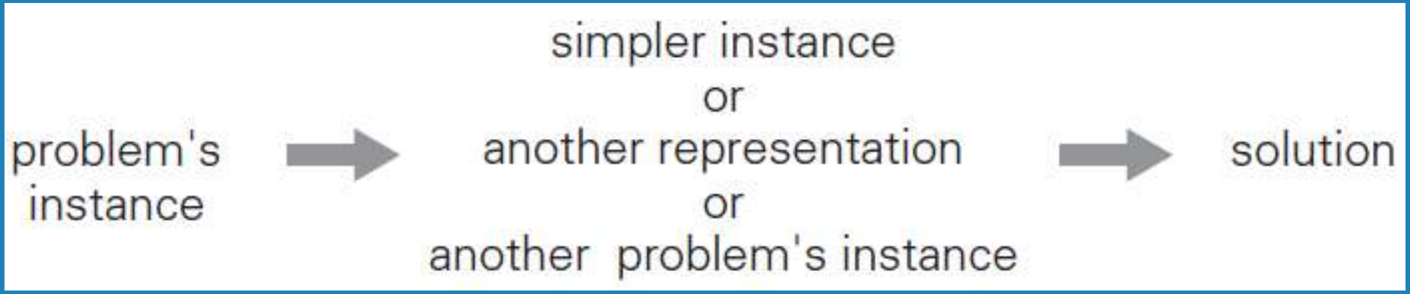
This group of techniques solves a problem by a *transformation*

عن طريق :

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

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

اسهل خطوة الترتيب .

Presorting

Many problems involving lists are easier when list is sorted.

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

التكرار

Also:

- Topological sorting helps solving some problems for directed acyclic graphs (DAGs).
- Presorting is used in many geometric algorithms.
- Presorting is a special case of preprocessing.

How fast can we sort ?

Efficiency of algorithms involving sorting depends on efficiency of sorting.

ما في ترتيب بقل عن nlogn

<u>Theorem</u> (see Sec. 11.2): $\lceil \log_2 n! \rceil \approx \boxed{n \log_2 n}$ comparisons are necessary in the worst case to sort a list of size n by <u>any</u> 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

Sort binary

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

لأن رح نبحث أكثر من مرة

Good or bad?

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

لو مرة وحدة ما بيحتاج ترتيب

تكرار

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

  Sort   scan

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

- **Brute force algorithm**

  Compare all pairs of elements

  2

  Efficiency: $O(n^2)$

- **Another algorithm? Hashing, which works well on average.**

Searching Problem

Problem: Given a 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:
  1. • file size (internal vs. external)
  2. • dynamics of data (static vs. dynamic)

- Dictionary operations (dynamic data):
  1. • find (search)
  2. • insert
  3. • delete

Taxonomy of Searching Algorithms

- List searching (good for static data)
  - 1 • sequential search – linear
  - 2 • binary search
  - 3 • interpolation search

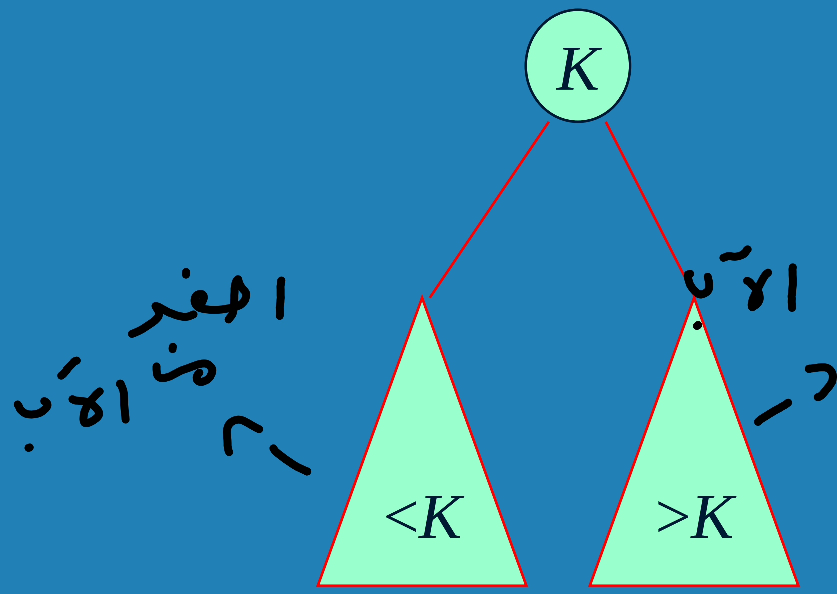
- Tree searching (good for dynamic data)
  - 1 • binary search tree
  - 2 • binary balanced trees: AVL trees 1, red-black trees 2
  - 3 • multiway balanced trees: 2-3 trees 1, 2-3-4 trees 2, B trees 3

بركز على ؟

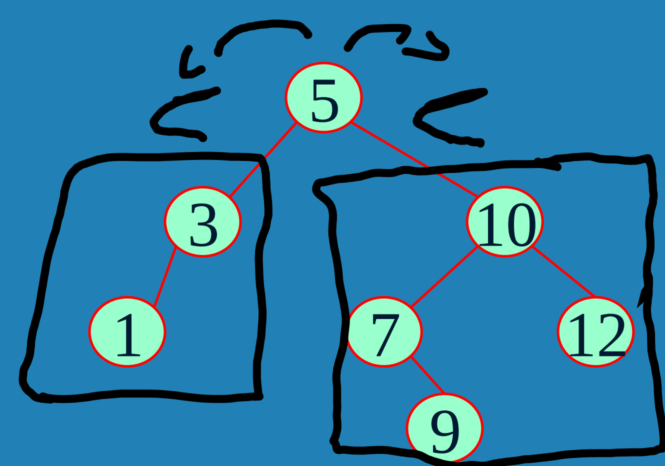
- Hashing (good on average case)
  - open hashing (separate chaining)
  - closed hashing (open addressing)

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:

1 deleting key at a leaf

2 deleting key at node with single child

3 deleting key at node with two children

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

Thus all three operations have

- worst case efficiency: $\Theta(n)$
- average case efficiency: $\Theta(\log n)$ (CLRS, Ch. 12)

Bonus: inorder traversal produces sorted list

Balanced Search Trees

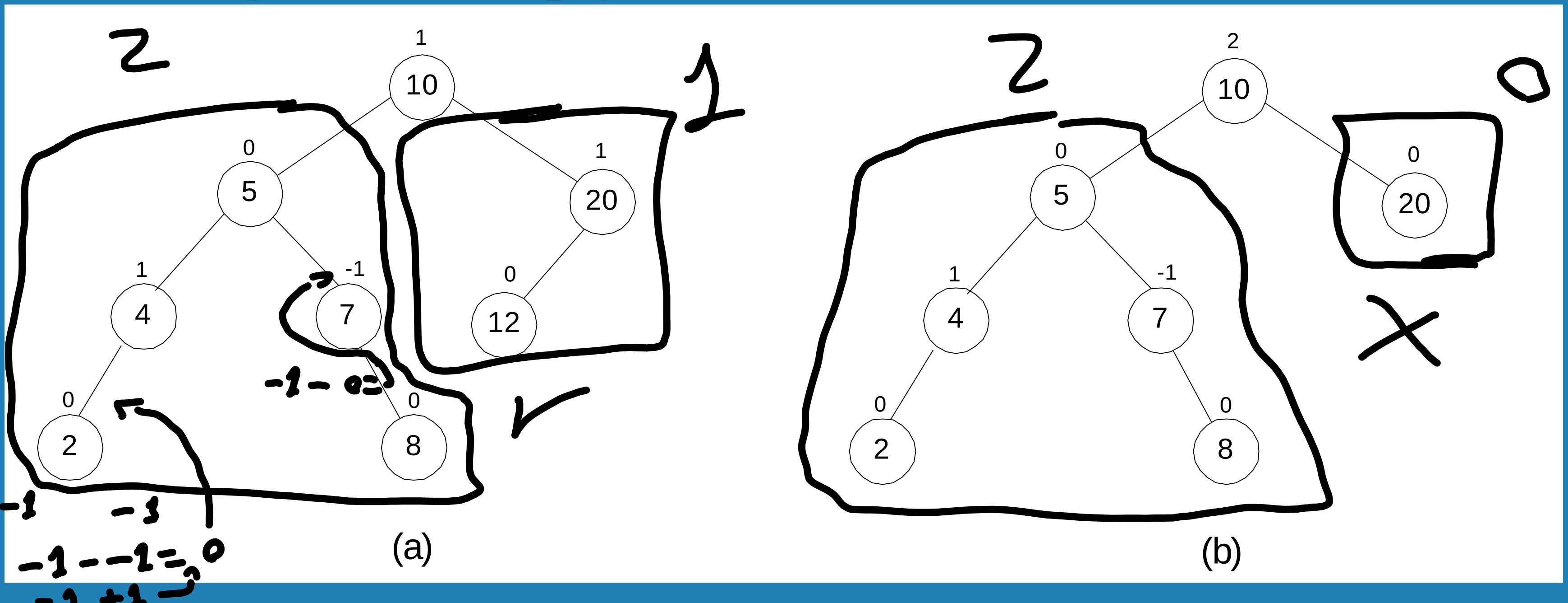
Attractiveness of *binary search tree* is marred by the bad (linear) worst-case efficiency. Two ideas to overcome it are:

- to rebalance binary search tree when a new insertion makes the tree "too unbalanced"
  1. *AVL trees*
  2. *red-black trees*

- to allow more than one key and two children
  1. *2-3 trees*
  2. *2-3-4 trees*
  3. *B-trees*

Balanced trees: AVL trees

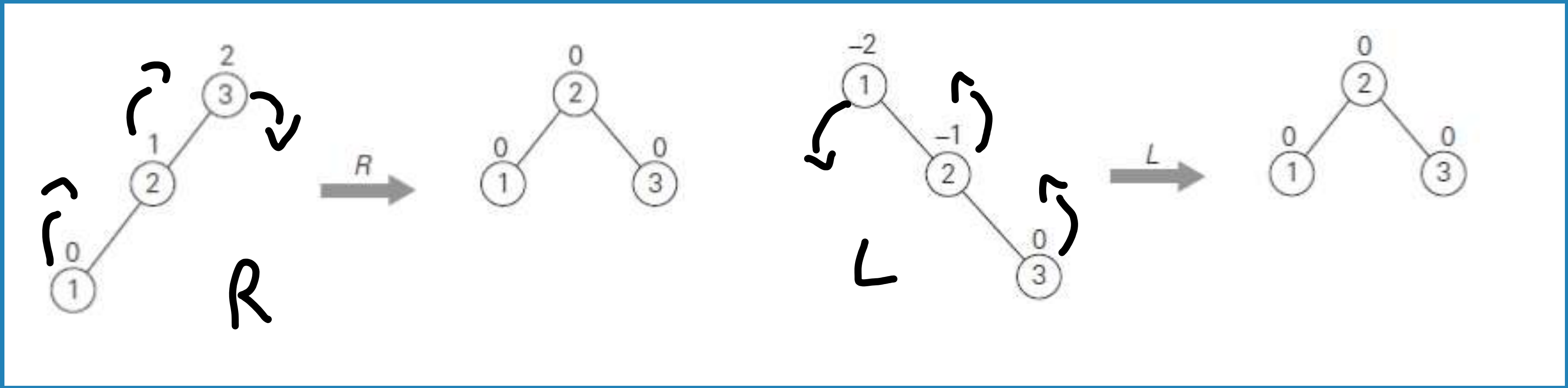
Definition An *AVL tree* is a binary search tree in which, for every node, the difference between the heights of its left and right subtrees, called the *balance factor*, is at most 1 (with the height of an empty tree defined as -1)



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

Rotations

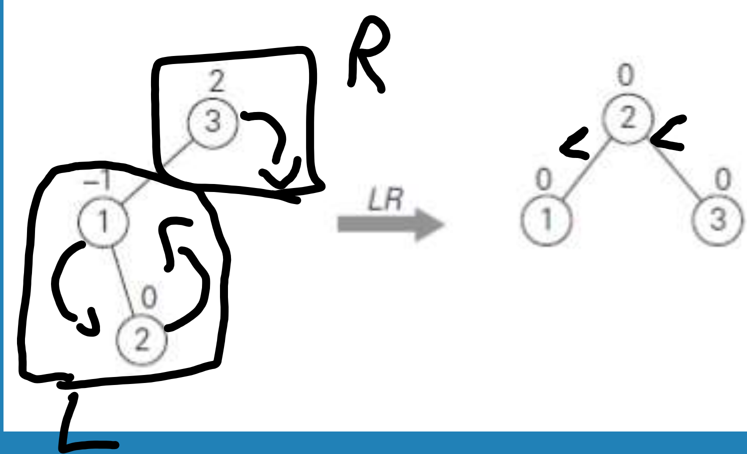
If a key insertion violates the balance requirement at some node, the subtree rooted at that node is transformed via one of the four *rotations*. (The rotation is always performed for a subtree rooted at an "unbalanced" node closest to the new leaf.)



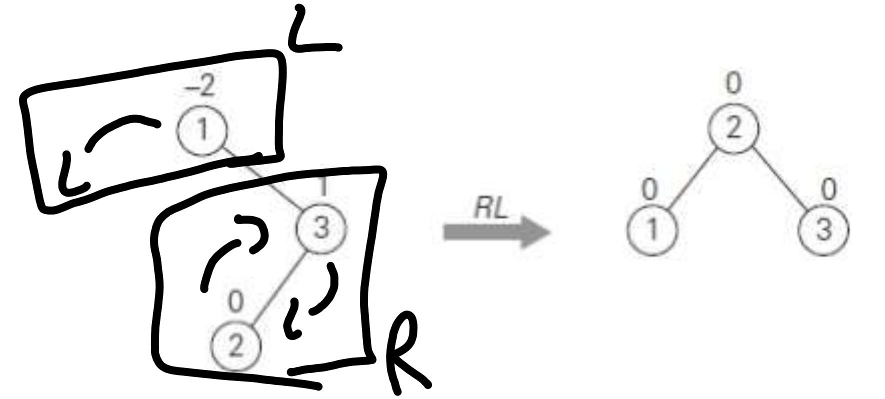
Single *R*-rotation

Single L-rotation

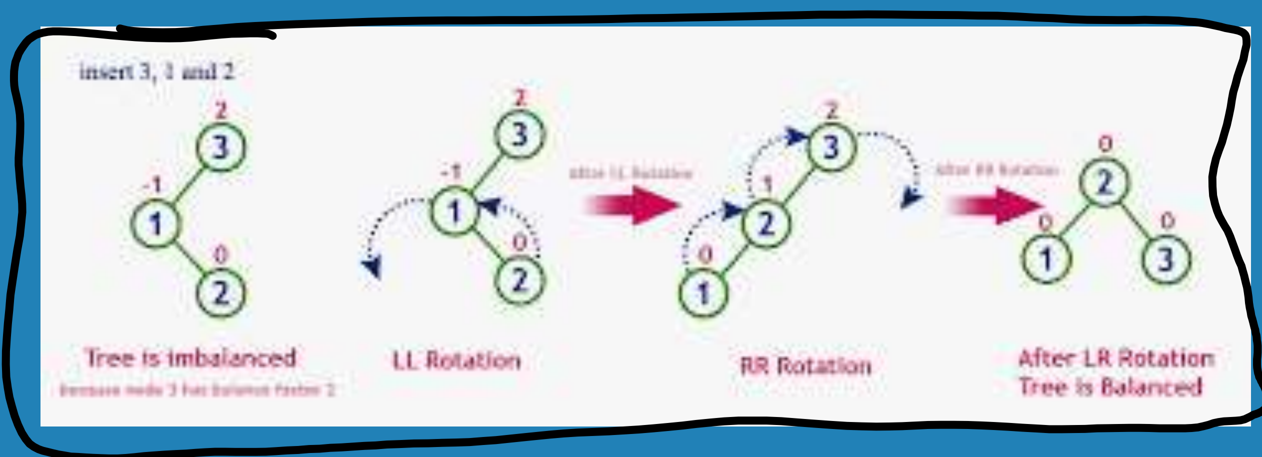
Rotations



Double *LR*-rotation



Double *RL*-rotation



Right Rotation:

a) Left Left Case

```
T1, T2, T3 and T4 are subtrees.
         z                                      y
        / \                                   /   \
       y   T4      Right Rotate (z)          x      z
      / \          - - - - - - - - ->      /  \    /  \
     x   T3                               T1  T2  T3  T4
    / \
  T1   T2
```

Left Rotation:

c) Right Right Case

```
  z                                y
 /  \                            /   \
T1   y     Left Rotate(z)       z      x
    /  \   - - - - - - - ->    / \    / \
   T2   x                     T1  T2 T3  T4
       / \
     T3  T4
```

LR Rotation:

b) Left Right Case

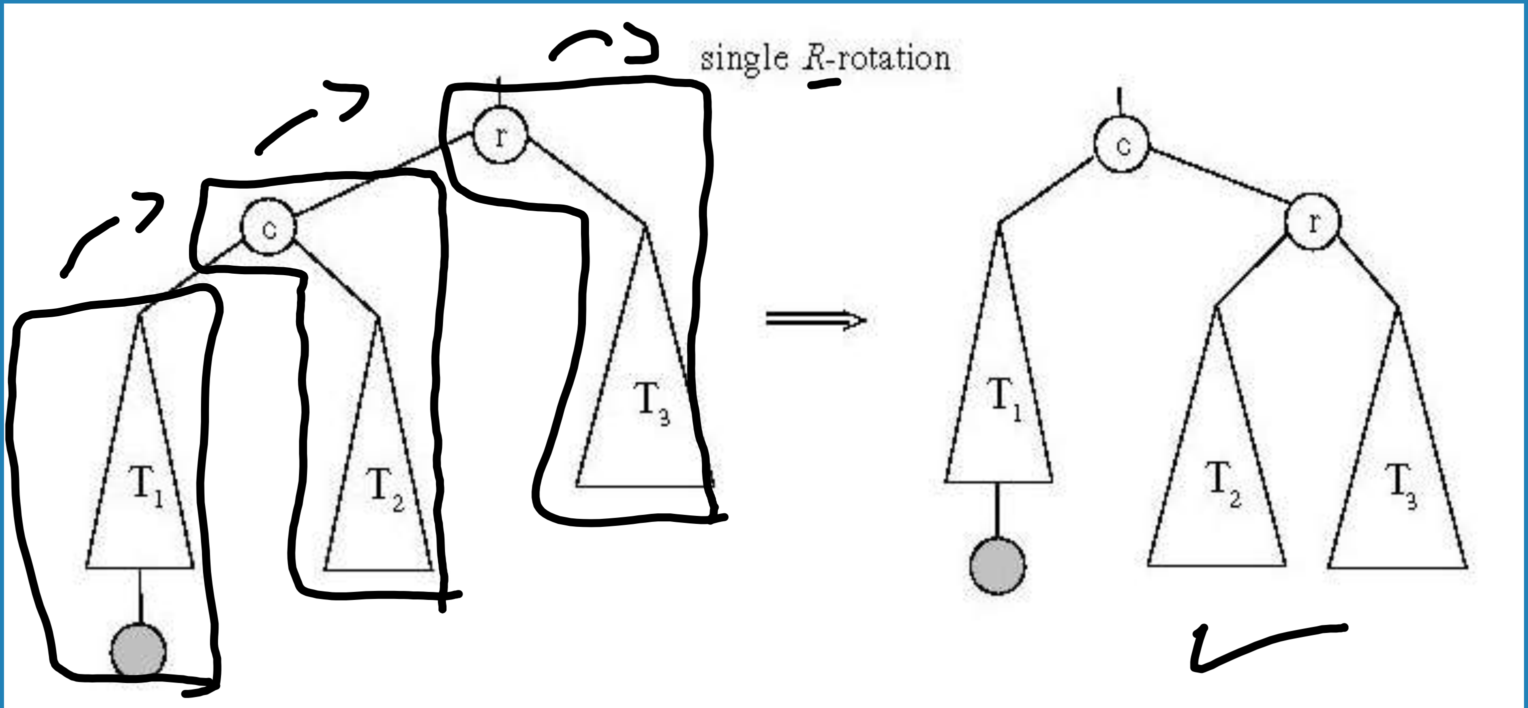
```
     z                               z                          x
    / \                            /   \                      /   \
   y   T4  Left Rotate (y)        x    T4  Right Rotate(z)   y     z
  / \      - - - - - - - - ->    /  \      - - - - - - - -> / \   / \
T1   x                          y    T3                   T1  T2 T3  T4
    / \                        / \
  T2   T3                    T1   T2
```

RL Rotation:

d) Right Left Case

```
     z                                z                          x
    / \                              / \                       /   \
  T1   y     Right Rotate (y)      T1   x      Left Rotate(z)  z      y
      / \    - - - - - - - - ->        /  \   - - - - - - - -> / \    / \
     x   T4                           T2   y                  T1  T2  T3  T4
    / \                                   /  \
  T2   T3                               T3    T4
```

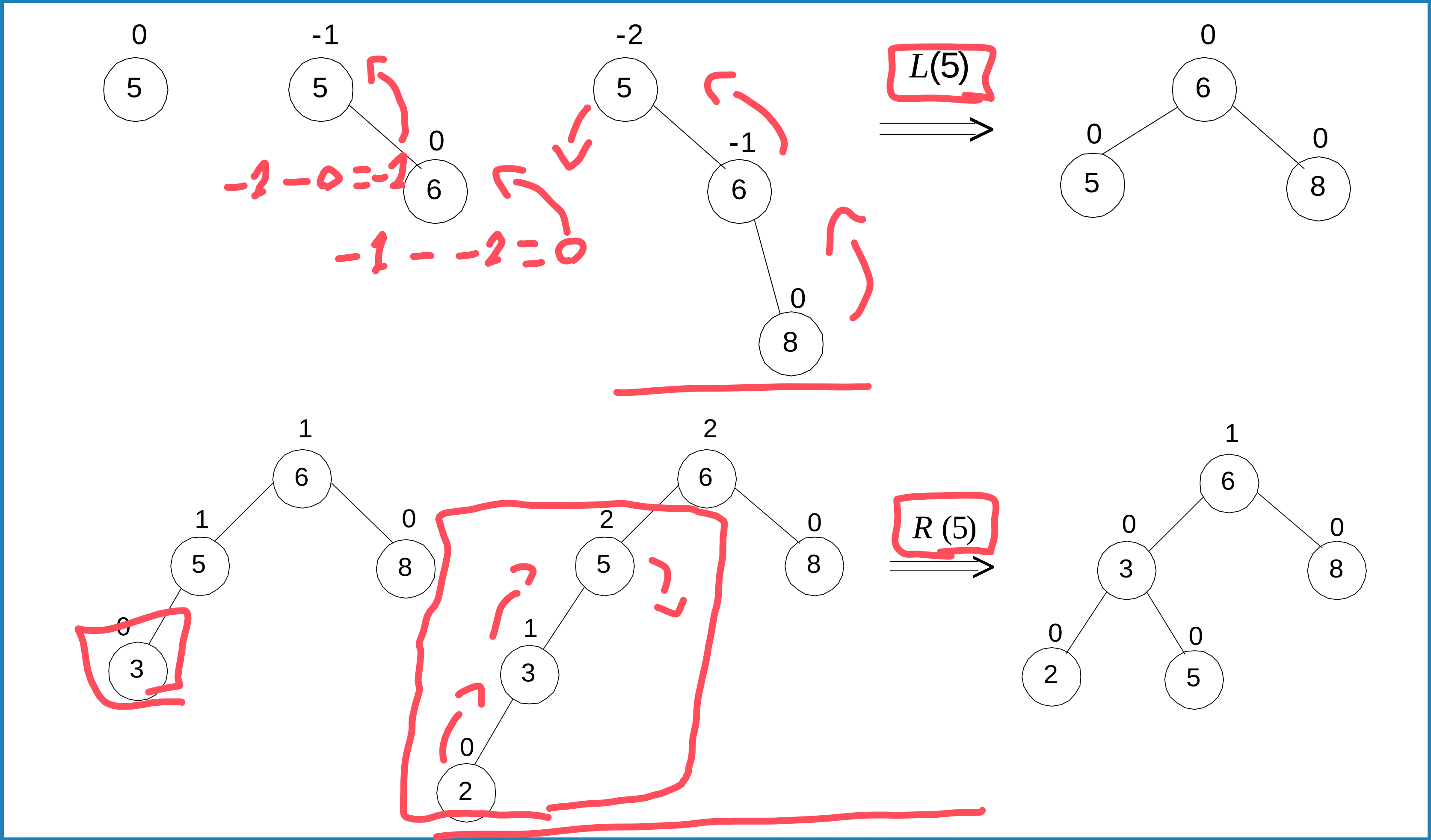
General case: Handling abandoned branches



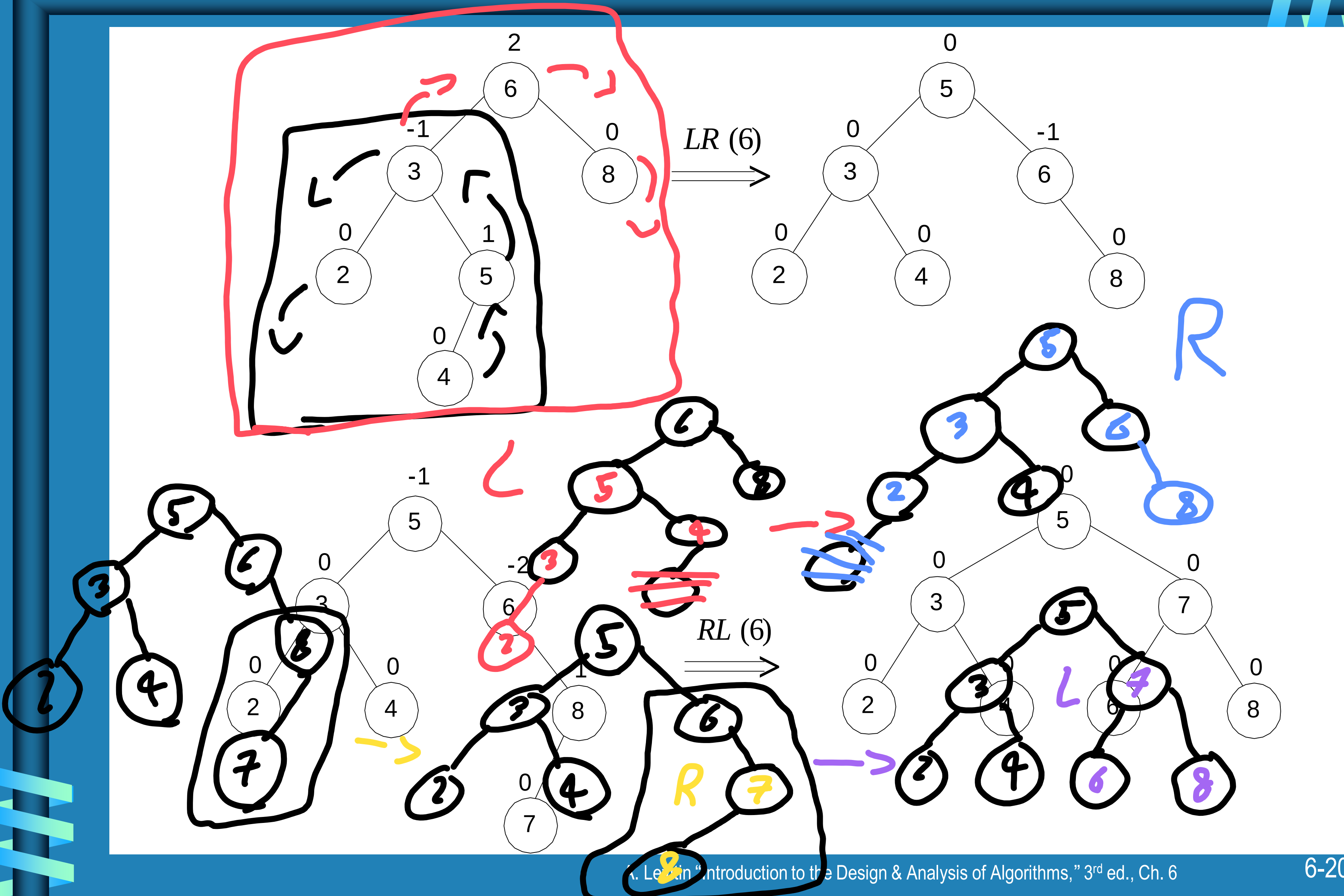
- When R-rotation is applied to node C, node T becomes its right-child.
- This leaves T2, which was the right-child before rotation, abandoned.
- This also leaves node r without a left-child, T2 takes this place. Note that T2 values are smaller than r so this is a legal operation.
- Similar procedure is used for left rotations.

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

- $h \leq 1.4404 \log_2 (n + 2) - 1.3277$ $N(h\text{-}1) + N(h\text{-}2) \leq N(h)$
  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*)

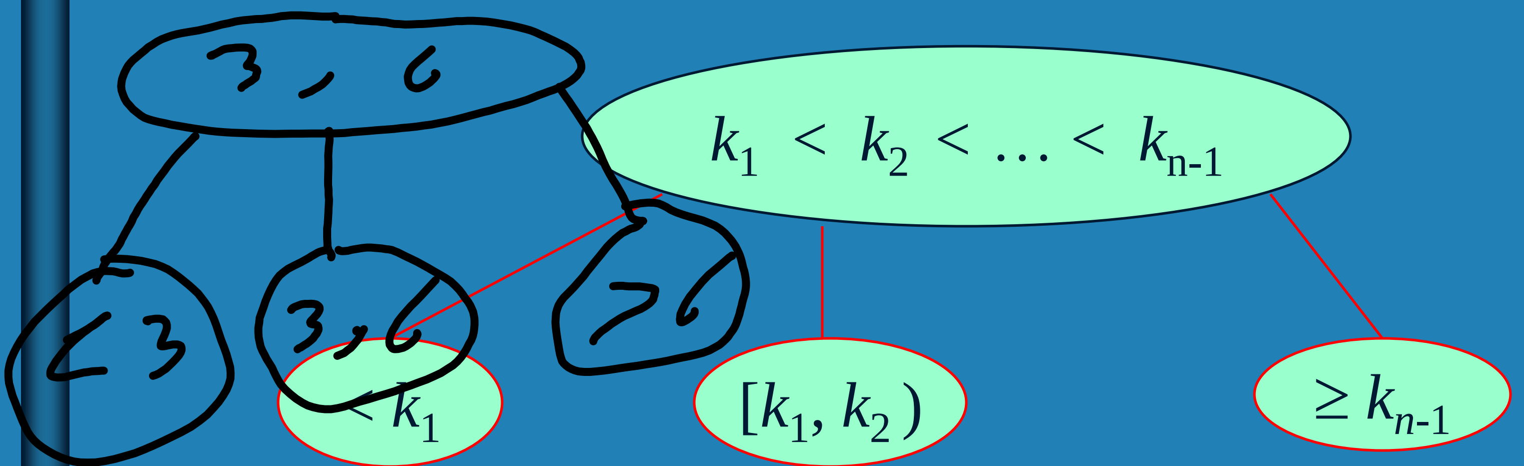
- Disadvantages:
  1 • frequent rotations
  2 • complexity

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

Multiway Search Trees

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

<u>Definition</u> **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):**

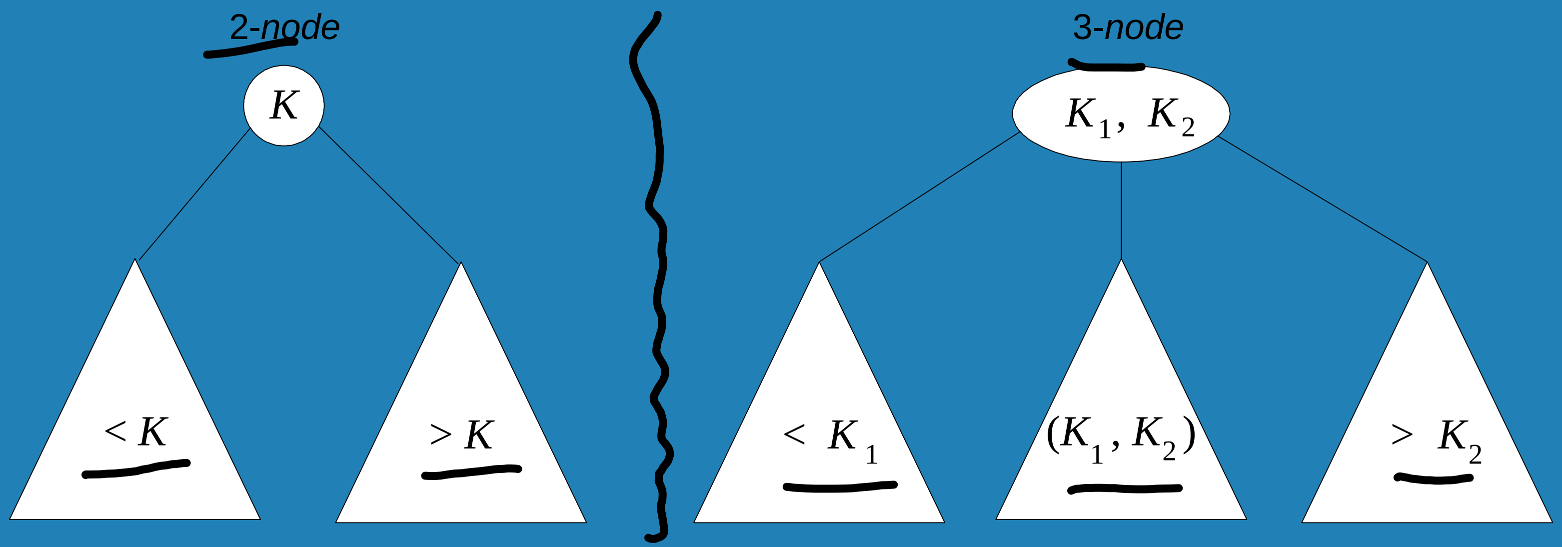


Note: Every node in a classical binary search tree is a 2-node

2-3 Tree

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

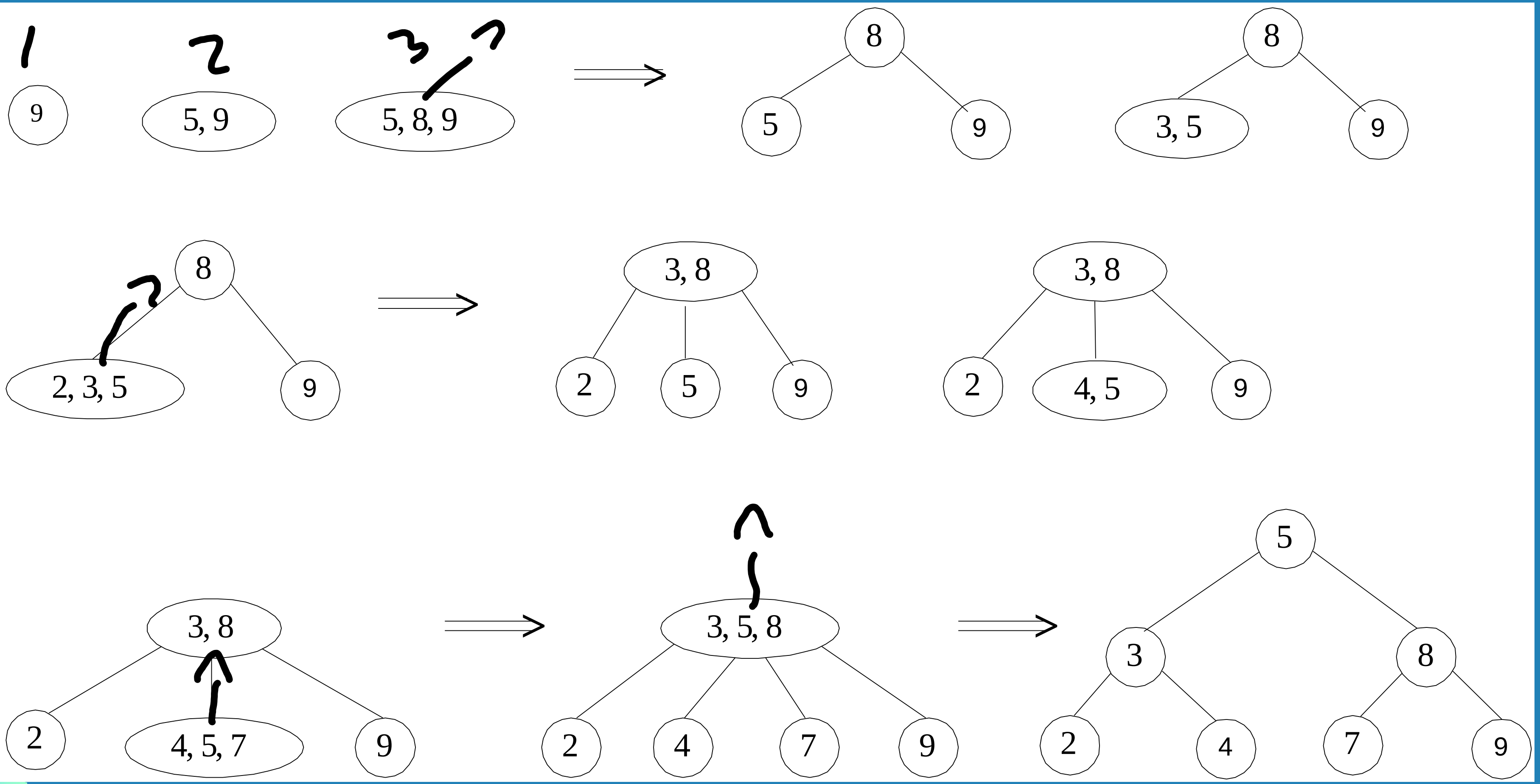
- **may have 2-nodes and 3-nodes**
- **height-balanced (all leaves are on the same level)**



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



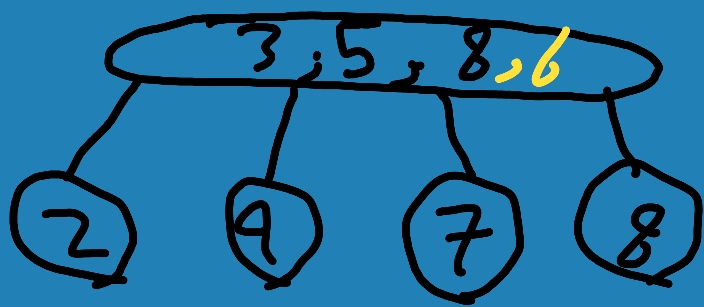
Analysis of 2-3 trees

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

- **Search, insertion, and deletion are in $\Theta(\log n)$**

- **The idea of 2-3 tree can be generalized by allowing more keys per node**
  1. • **2-3-4 trees**
  2. • **B-trees**

ADD = 6



كبس