CSC 411
Design and Analysis of Algorithms

Chapter 6 Transfer and Conquer - Part 2

Instructor: Minhee Jun

Transfer-and-Conquer Examples

- Presorting (6.1)
- Gaussian Elimination (6.2)
- Balanced Search Trees (6.3)
  - AVL Trees
  - 2-3 Trees
- Heaps and Heapsort (6.4)
- Horner's Rule and Binary Exponentiation (6.5)

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

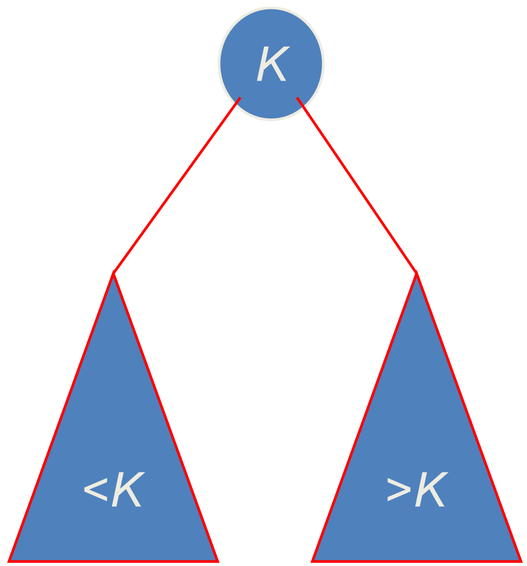
Taxonomy of Searching Algorithms

- 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)

Each approach needs a different data structure

Binary Search Tree

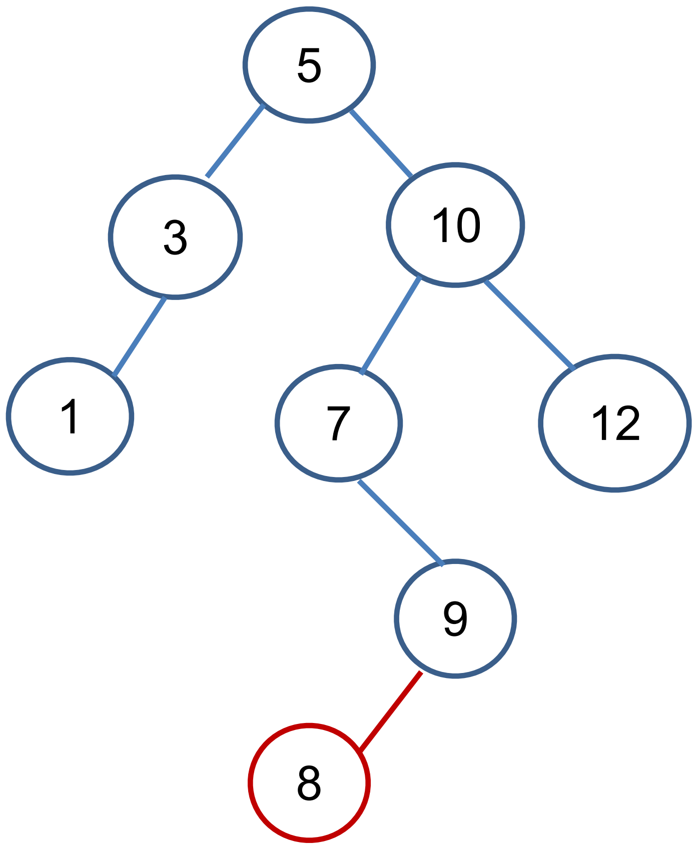
- Arrange keys in a binary tree with the binary search tree property:



- Example: construct a binary search tree with the following values as inputs: 5, 3, 1, 10, 12, 7, 9

Binary Search Tree: Example

- Example: construct a binary search tree with the following values as inputs: 5, 3, 1, 10, 12, 7, 9



Add 8?

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

Operations on Binary Search Trees

- 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: $\Theta(n)$
  - best case efficiency: $\Theta(\log n)$

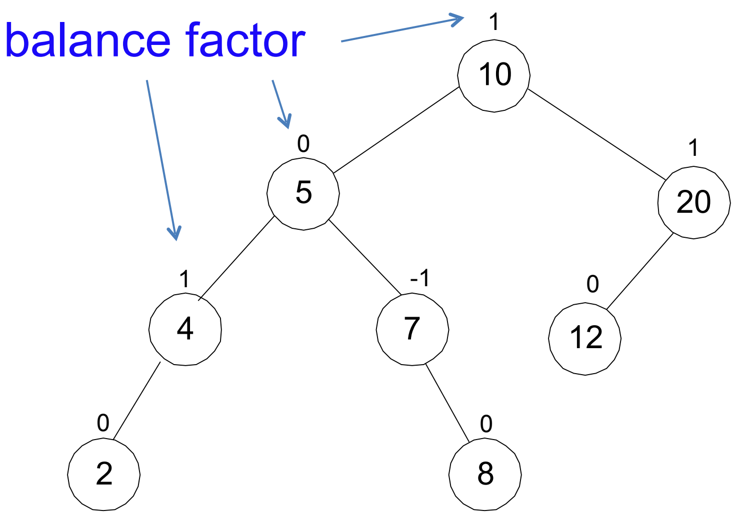
- <u>Bonus</u>: inorder traversal produces sorted list

Balanced Search Trees

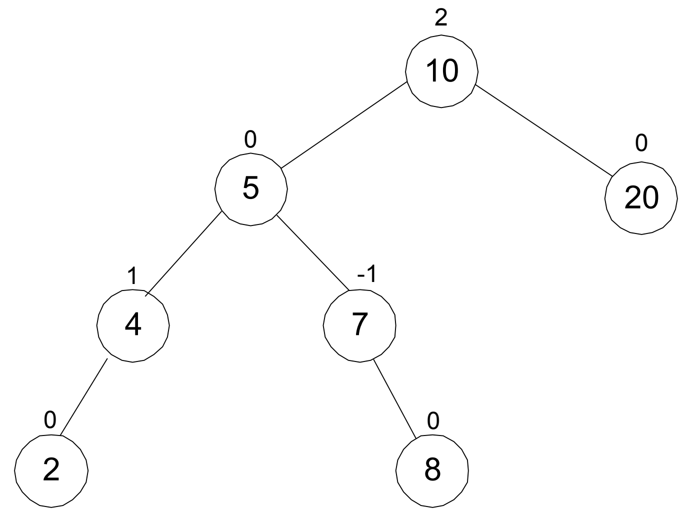
- Attractiveness of *binary search tree* can be hindered by the bad (linear) worst-case efficiency.
- Two ideas to overcome it are:
  - to rebalance binary search tree when a new insertion makes the tree "too unbalanced"
    - *AVL trees*
    - *red-black trees (will not cover)*
  - to allow more than one key per node of a search tree
    - *2-3 trees*
    - *2-3-4 trees*
    - *B-trees*

Balanced trees: AVL trees

- <u>Definition</u> 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

AVL trees: rotations

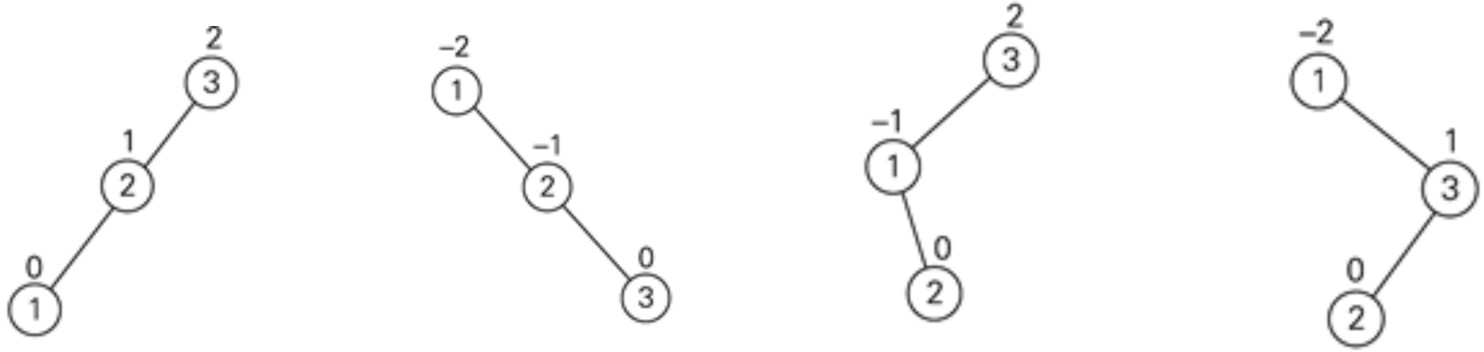
- If an insertion of a new node makes an AVL tree unbalanced, we transform the tree by rotation
- The subtree rooted at that node is transformed via one of the four *rotations*.
  - Single right rotation
  - Single left rotation
  - Double left-right rotation
  - Double right-left rotation
- A rotation in an AVL tree is a local transformation of its subtree rooted at the node whose balance factor has become either +2 or -2.
- If there are several nodes with the +2/-2 balance, the rotation is always performed for a subtree rooted at an “unbalanced” node closest to the new leaf.

AVL trees, 4 rotation types: Example

- Construct the binary tree (that satisfies the binary search tree property)
  - 3, 2, 1
  - 1, 2, 3
  - 3, 1, 2
  - 1, 3, 2

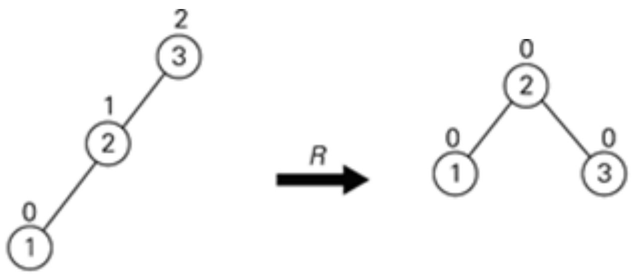
Balance factor = height of T(left) – height T(right)

In AVL tree, balance factor of every node is either 0 or 1 or -1.



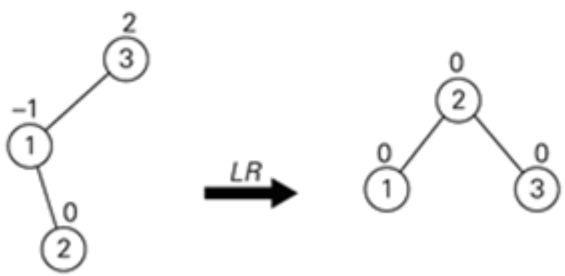
Not AVL tree

AVL trees, 4 Rotations



Single *R*-rotation

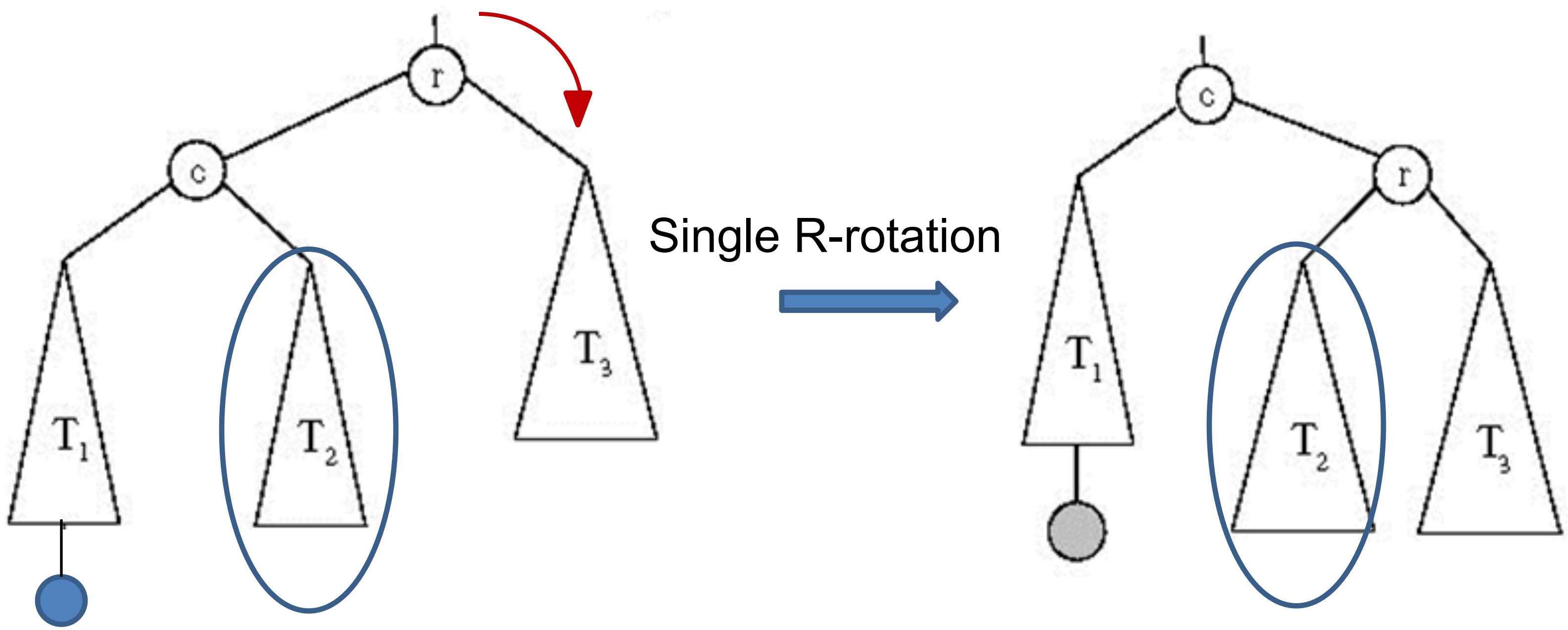
Single L-rotation



Double *LR*-rotation

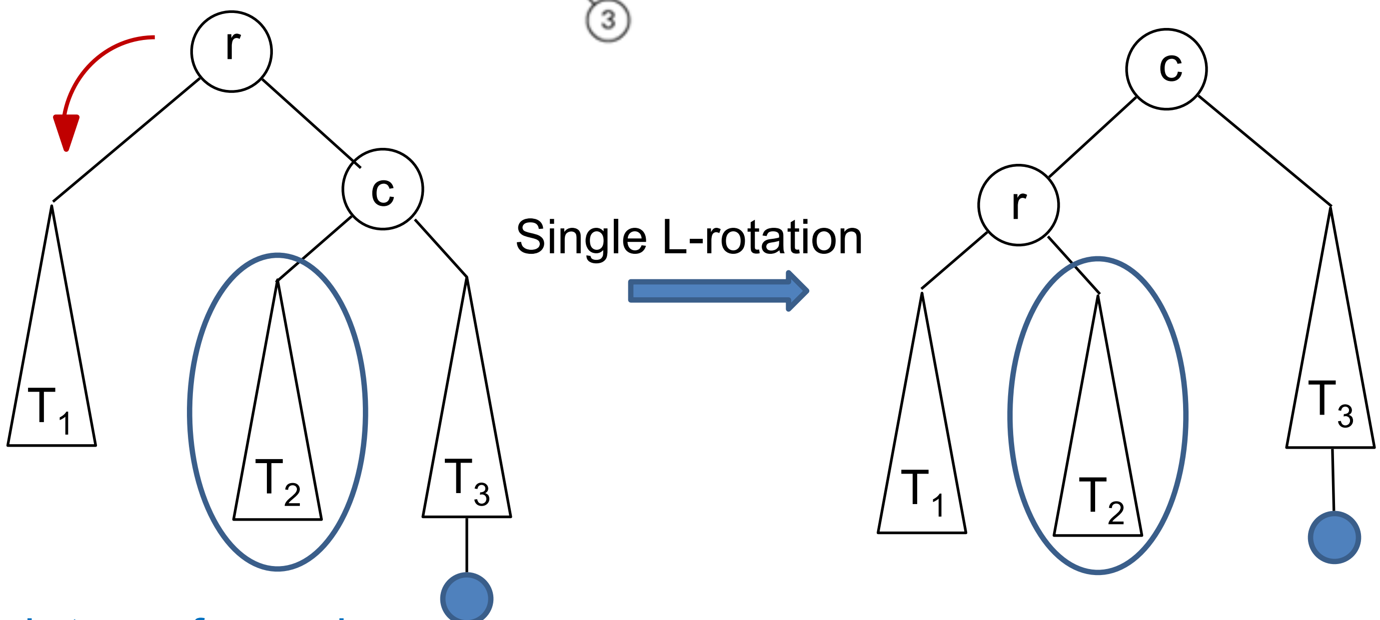
Double *RL*-rotation

AVL trees, 4 Rotations: general case
Single R-rotation



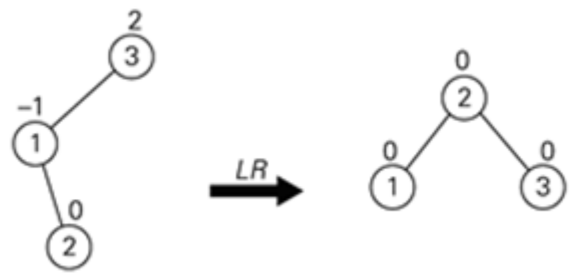
right sub-tree of c, and
the left sub-tree of r

AVL trees, 4 Rotations: general case
Single L-rotation

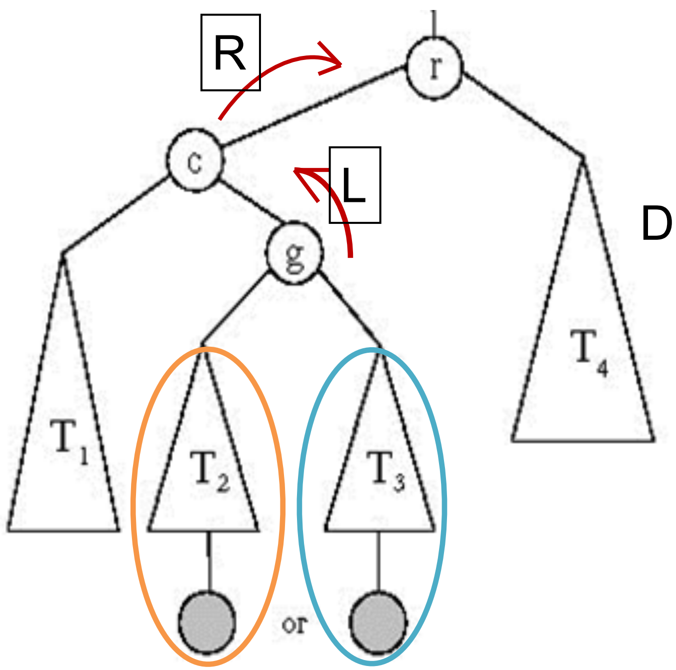


left sub-tree of c, and
the right sub-tree of r

AVL trees, 4 Rotations: general case
General case: Double LR-rotation

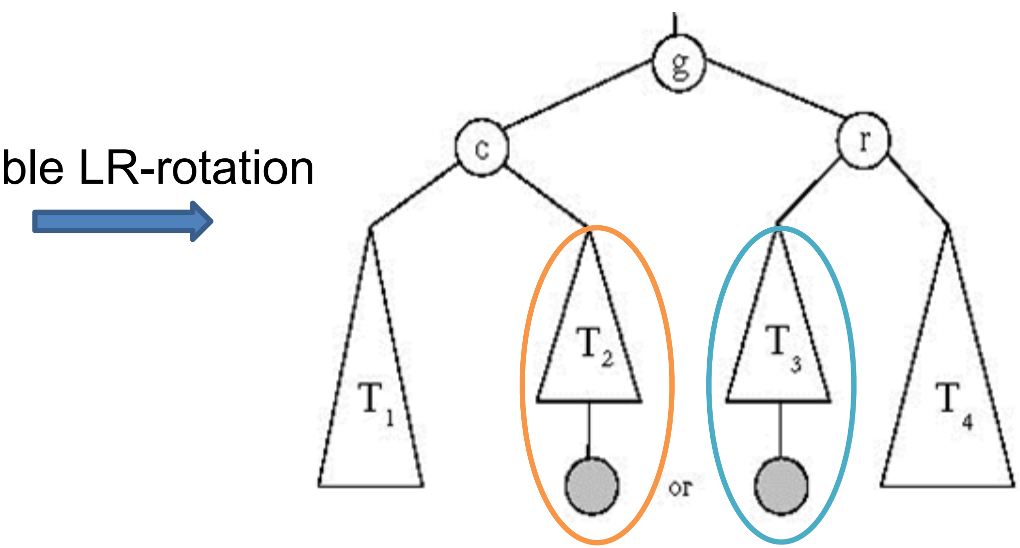


You need to maintain the binary search tree property

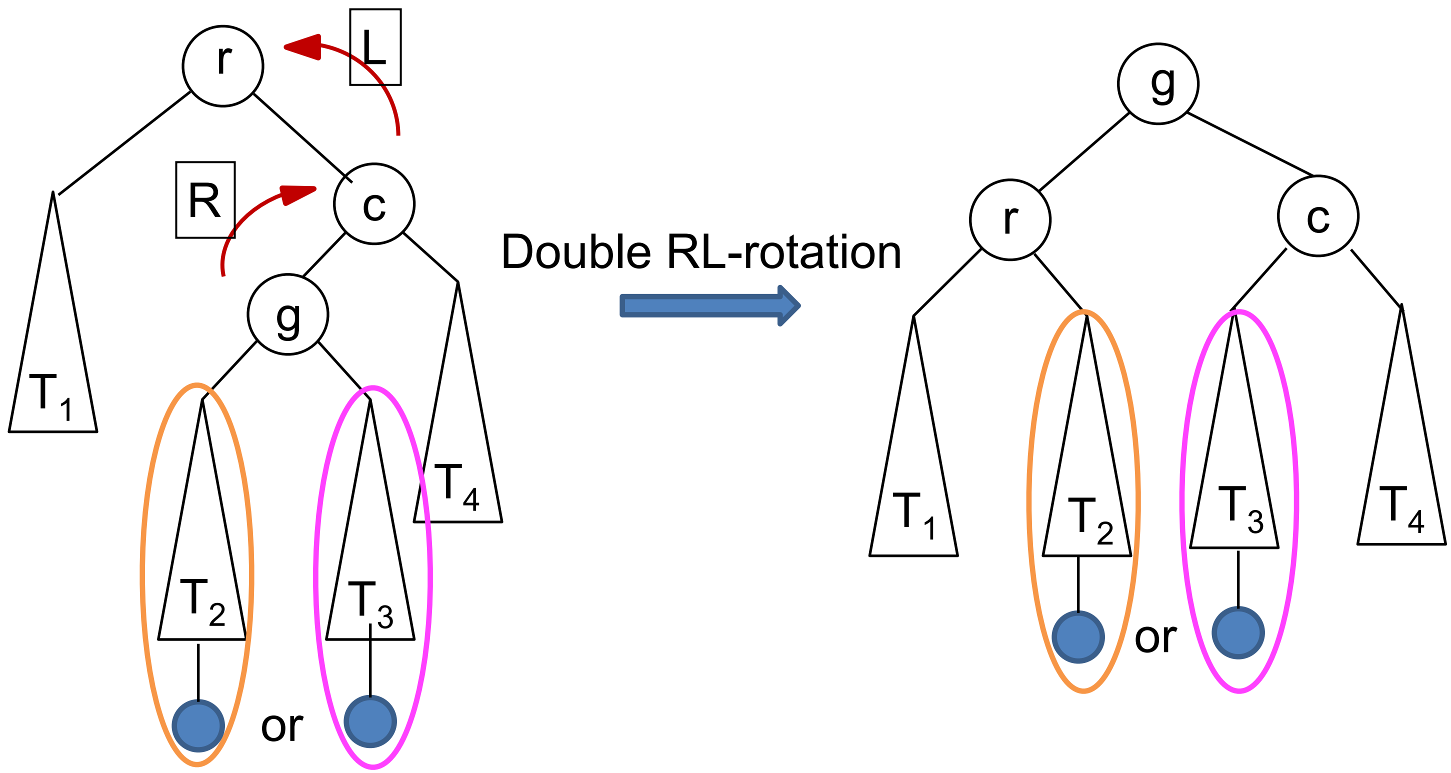


Added new node

Double LR-rotation

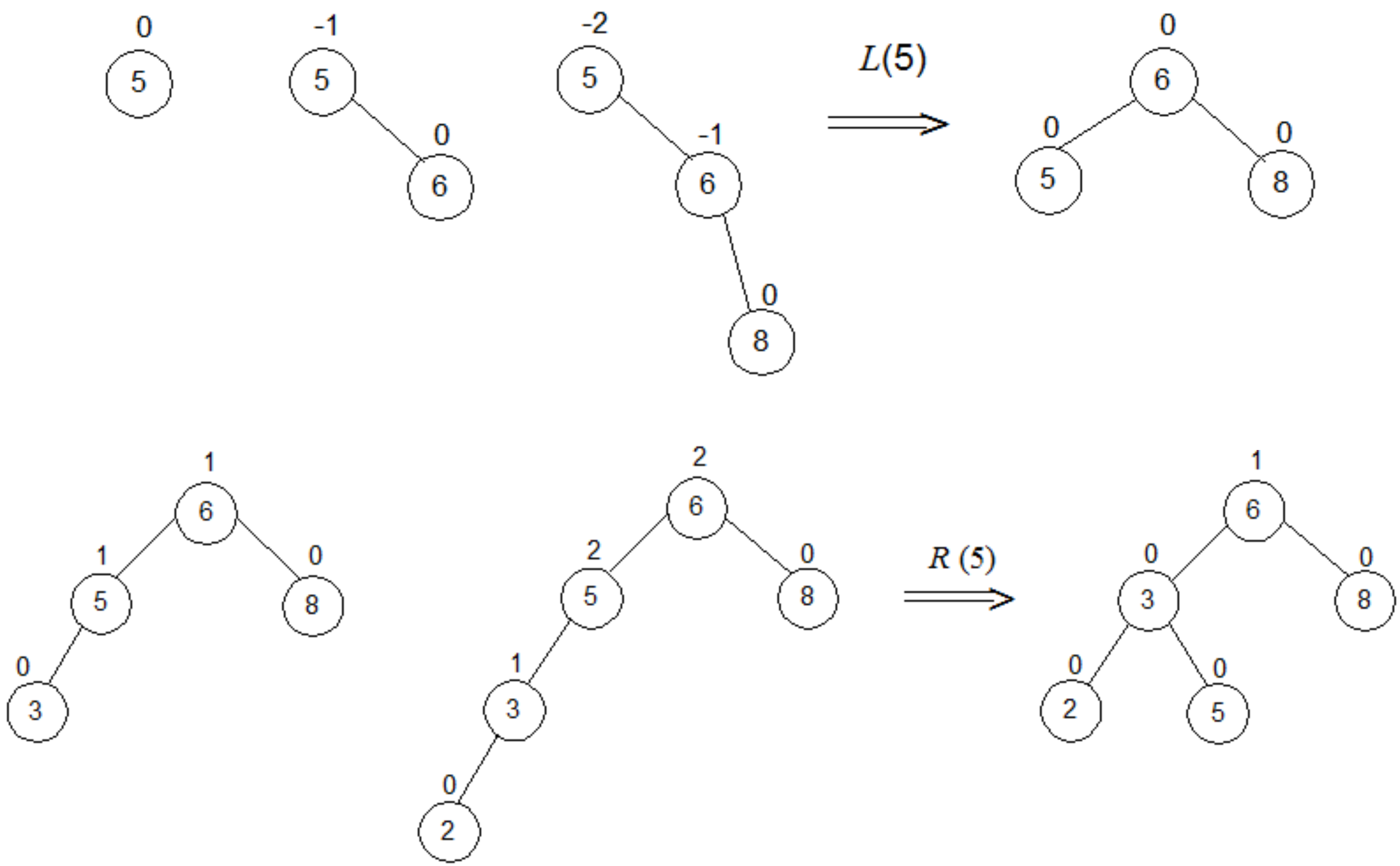


AVL trees, 4 Rotations: general case
Double RL-rotation



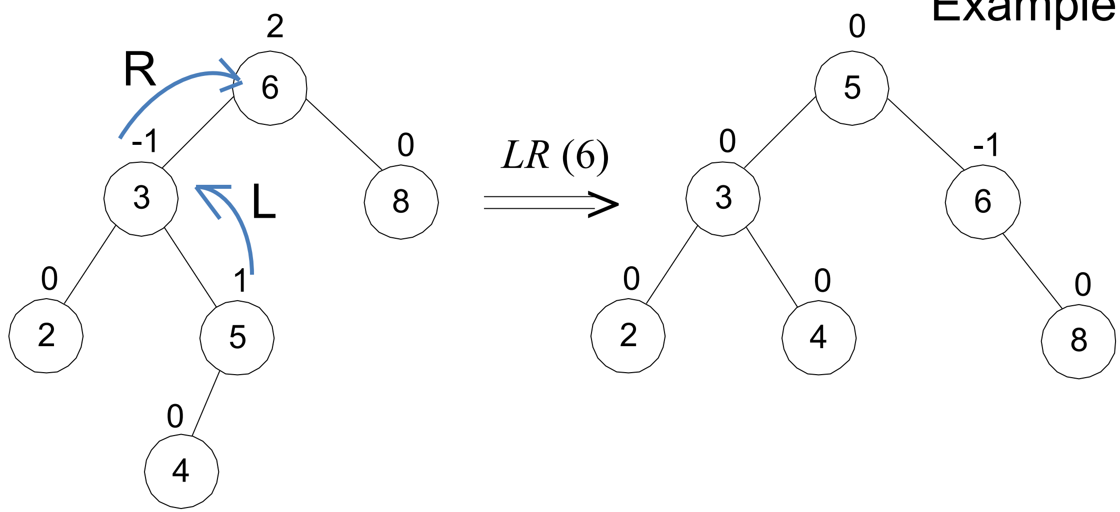
AVL tree construction - an example

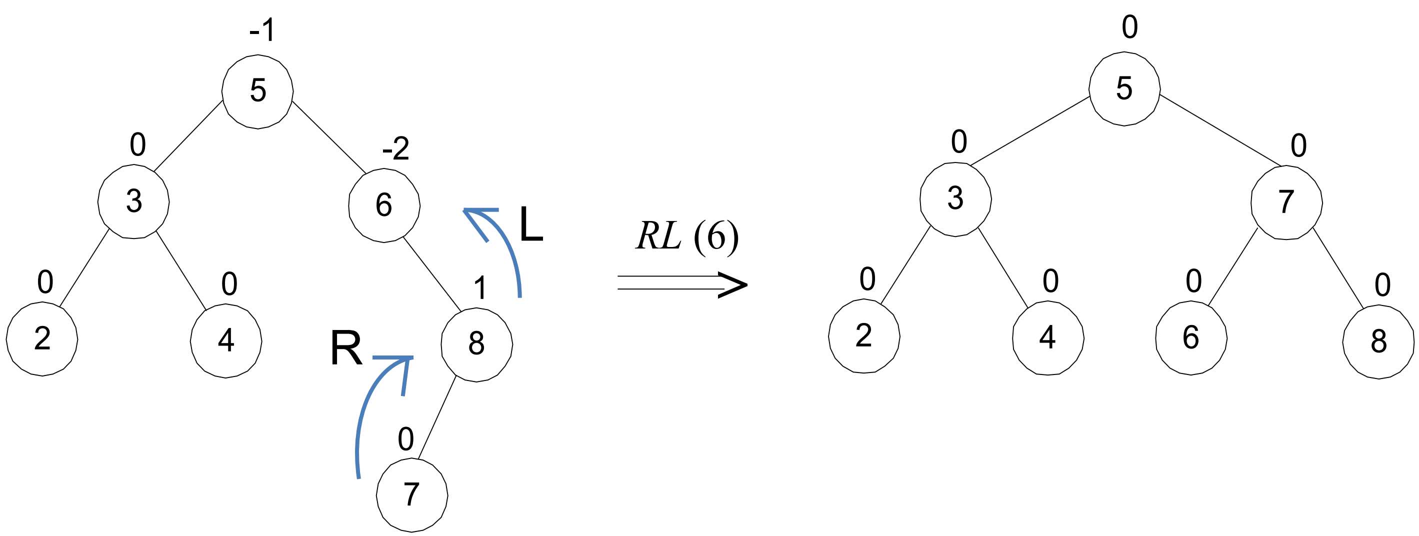
Example: 5, 6, 8, 3, 2, 4, 7



AVL tree construction - an example

Example: 5, 6, 8, 3, 2, 4, 7



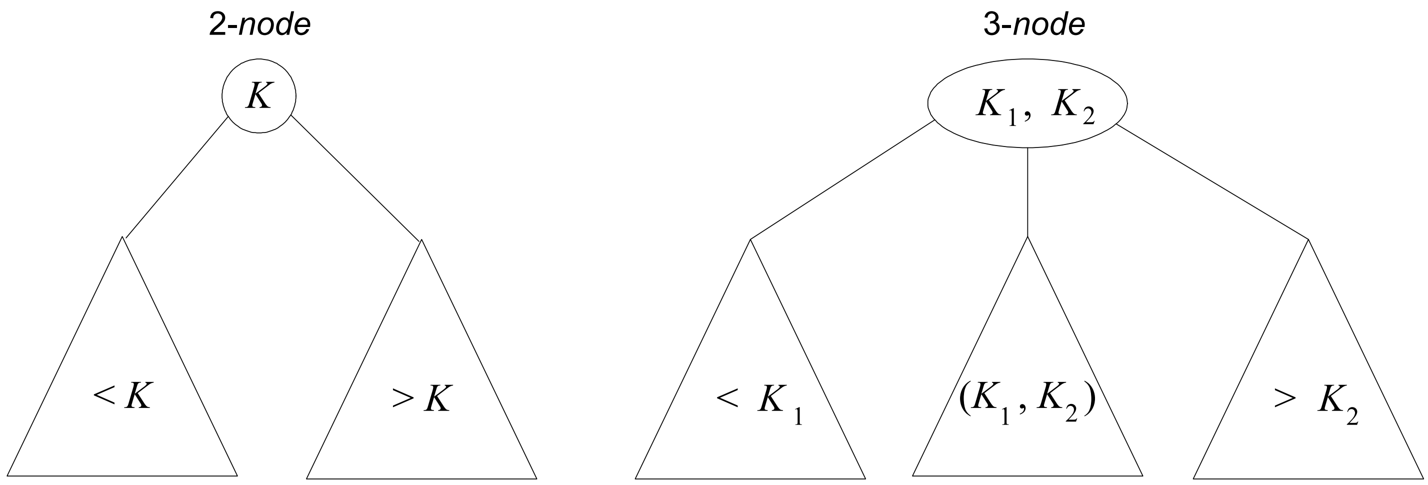


Analysis of AVL trees

- Efficiency:
  - $\lfloor \log_2 n \rfloor \leq h < 1.4405 \log_2(n+2) - 1.3277$
- average height: (found empirically)
  - 1.01 $\log_2 n$ + 0.1 for large n
- Search and insertion are O(log n)
- Deletion is more complicated but is also O(log n)
- Disadvantages:
  - frequent rotations
  - complexity

2-3 Tree

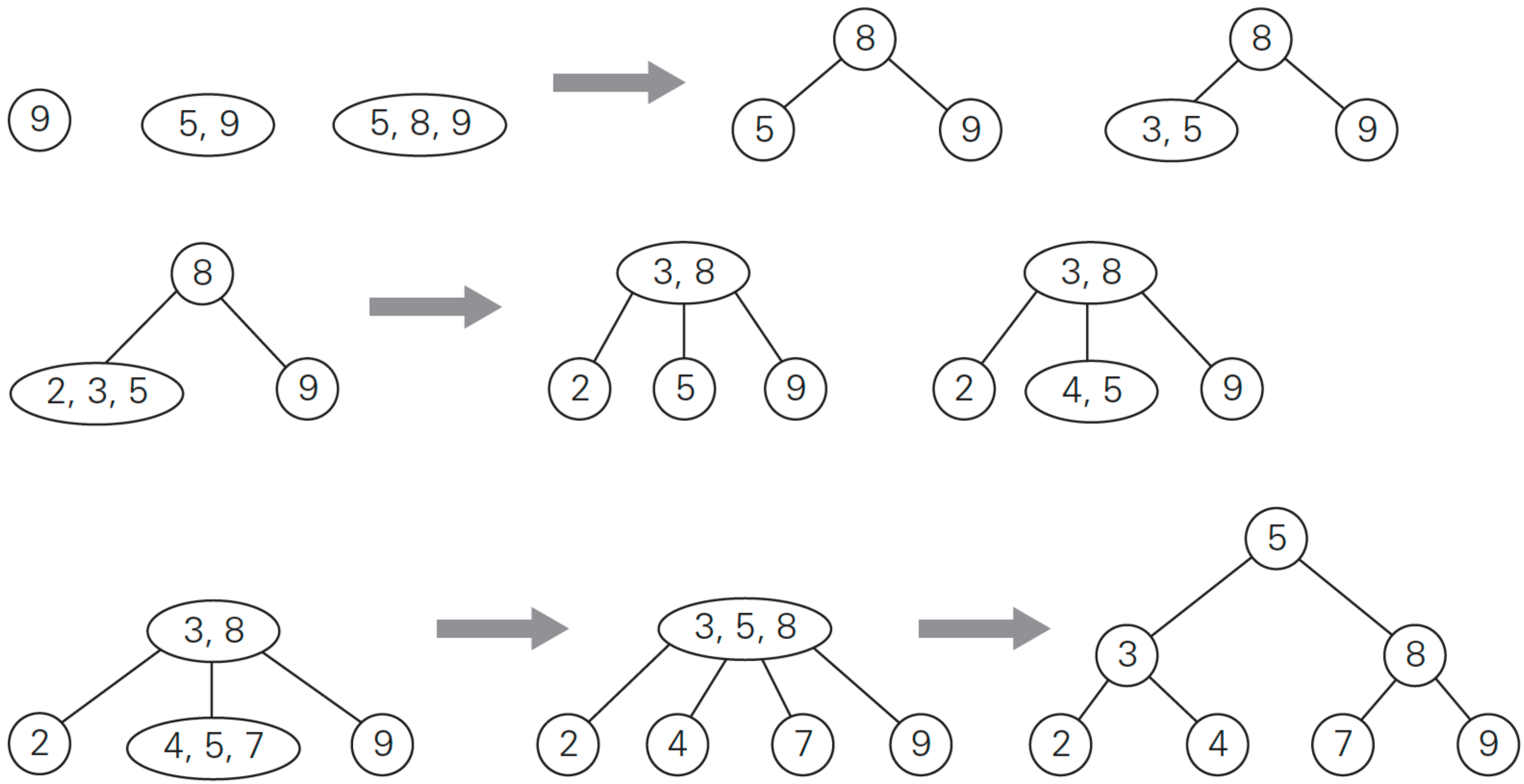
- <u>Definition</u> A *2-3 tree* is a search tree that
- may have 2-nodes and 3-nodes
- height-balanced (all leaves are on the same level)



- 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

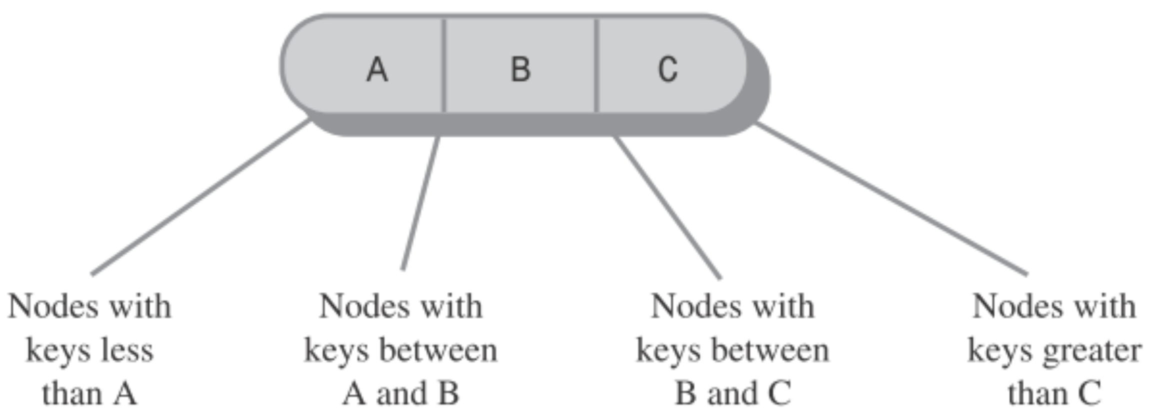


Analysis of 2-3 trees

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

What's in a Name (2-3-4)?

- How many links to child nodes can be potentially be contained in a given node.
  - A node with 1 data item always has 2 children.
  - A node with 2 data item always has 3 children.
  - A node with 3 data item always has 4 children.
- A 2-3-4 tree is a multiway tree of order=4.



< A   < A, B]   < B, C]   > C