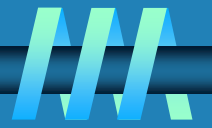
CS 395 – Analysis of Algorithms

Chapter 6 – Transform-and-Conquer

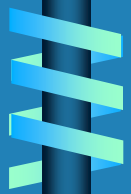
Read 6.3

- **Balanced Search Trees**
  - **AVL Trees**

Taxonomy of Searching Algorithms

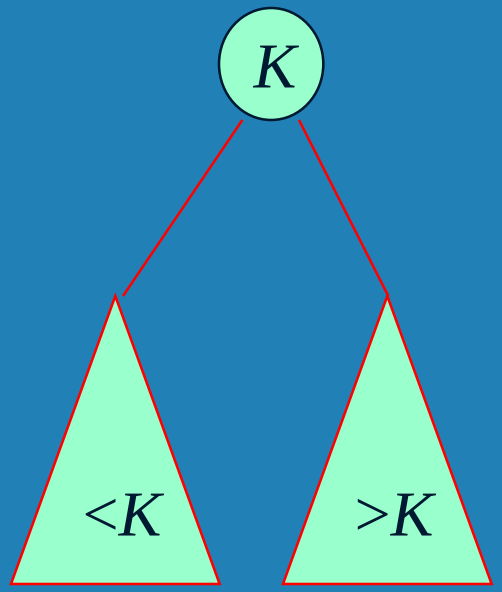


- List searching
  - sequential search
  - binary search
  - interpolation search
- Tree searching
  - binary search tree
  - binary balanced trees: AVL trees, red-black trees
  - multiway balanced trees: 2-3 trees, 2-3-4 trees, B trees
- Hashing
  - open hashing (separate chaining)
  - closed hashing (open addressing)

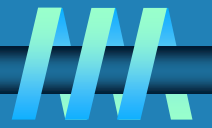


Binary Search Tree

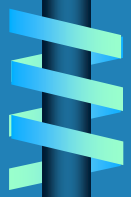
- Arrange keys in a binary tree with the binary search tree property:



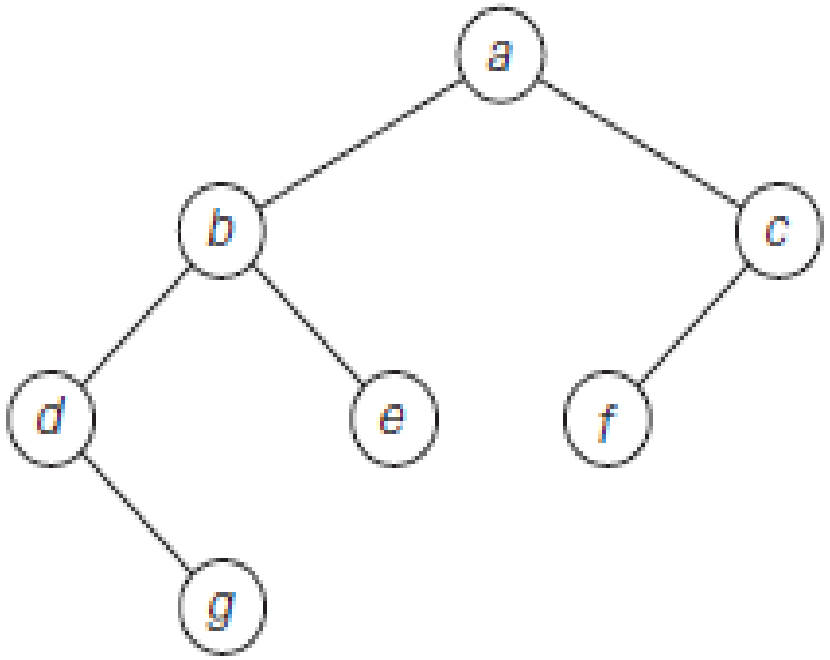
Binary Trees – Classical Traversals



- Three classic (depth-first) traversals:
  - preorder traversal
    - visit the parent first and then left and right children. (PLR)
  - inorder traversal
    - visit the left child, then the parent and the right child. (LPR)
  - In the postorder traversal
    - visit left child, then the right child and then the parent. (LRP)
- All three are recursive
- Differ only by the timing of parent's visit



Classical Traversals - Example



preorder: *a, b, d, g, e, c, f*
inorder: *d, g, b, e, a, f, c*
postorder: *g, d, e, b, f, c, a*

Preorder Traversal Algorithm

- **Algorithm Preorder(T)**

  if T ≠ ∅

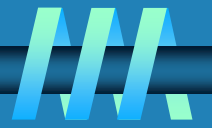
  print(root of T)

  Preorder(Tleft)

  Preorder(Tright)

- **Efficiency: Θ(n)**

Inorder Traversal Algorithm



- **Algorithm Inorder(T)**

  if T ≠ ∅

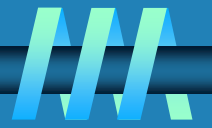
  Inorder(Tleft)

  print(root of T)

  Inorder(Tright)

- **Efficiency: Θ(n)**

Postorder Traversal Algorithm



- **Algorithm Postorder(T)**

  if T ≠ ∅

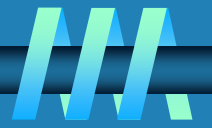
  Postorder(Tleft)

  Postorder(Tright)

  print(root of T)

- **Efficiency: Θ(n)**

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***

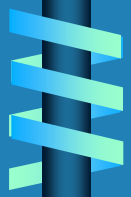
instance-simplification

- **to allow more than one key per node of a search tree**
  - ***2-3 trees***
  - ***2-3-4 trees***
  - ***B-trees***

Representation -change

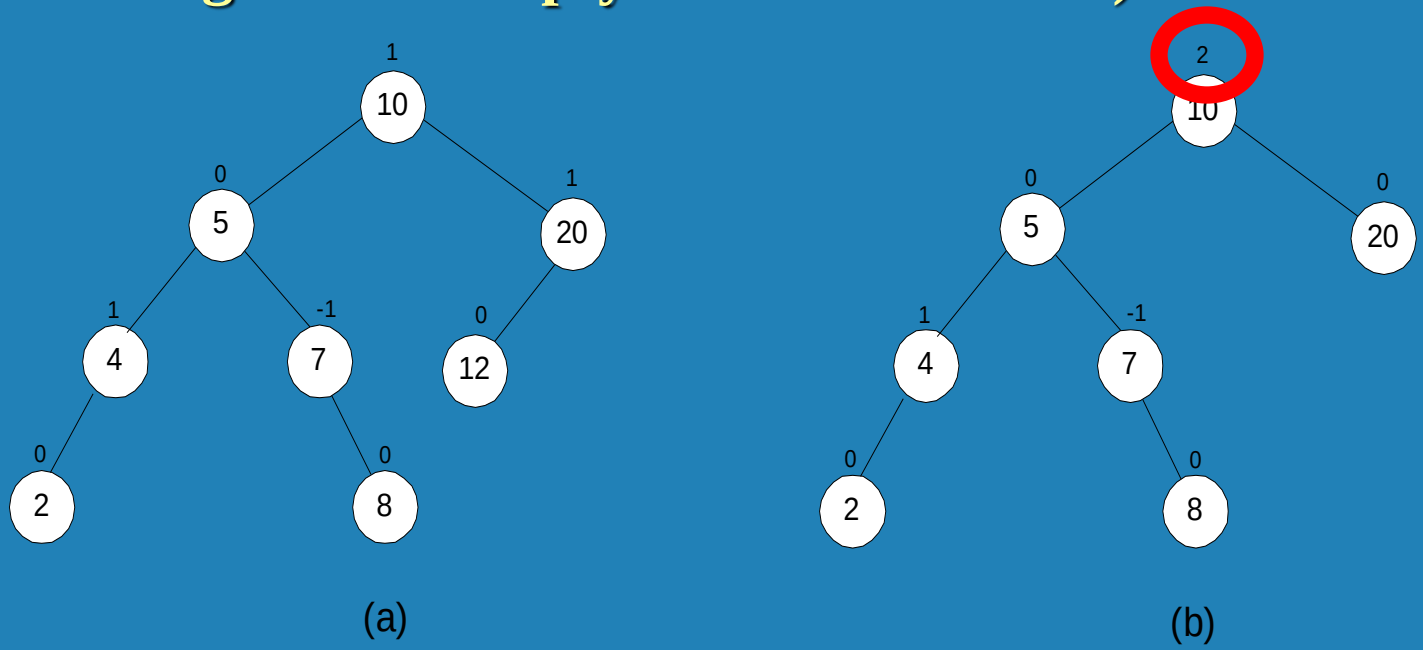
Transformation variations:

- **instance simplification:** a simpler/more convenient instance of the same problem
- **representation change:** a different representation of the same instance
- **problem reduction:** a different problem for which an algorithm is already available



Balanced trees: AVL trees

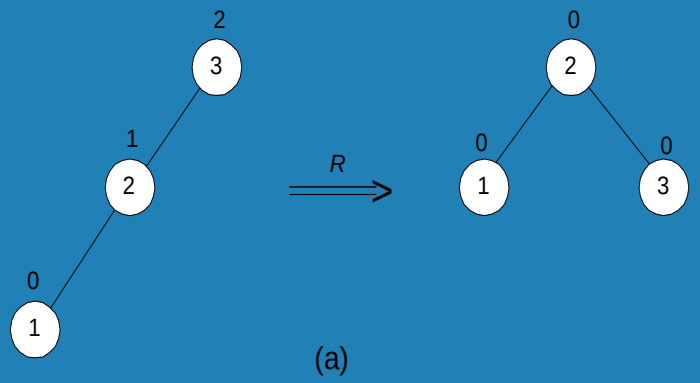
Definition An *AVL tree* is a binary search tree in which, for every node, the difference between the heights of its left and right subtrees, called the *balance factor*, is at most 1 (with the height of an empty tree defined as -1)



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

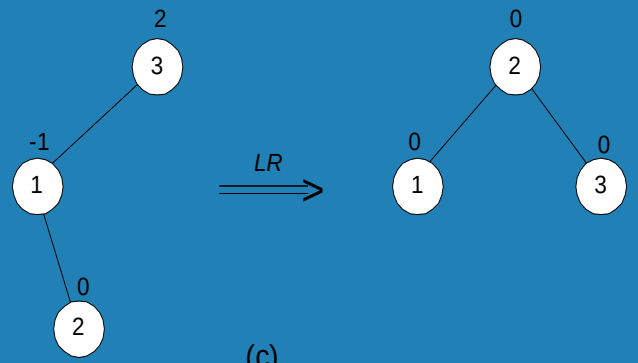
Rotations

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.)



(a)

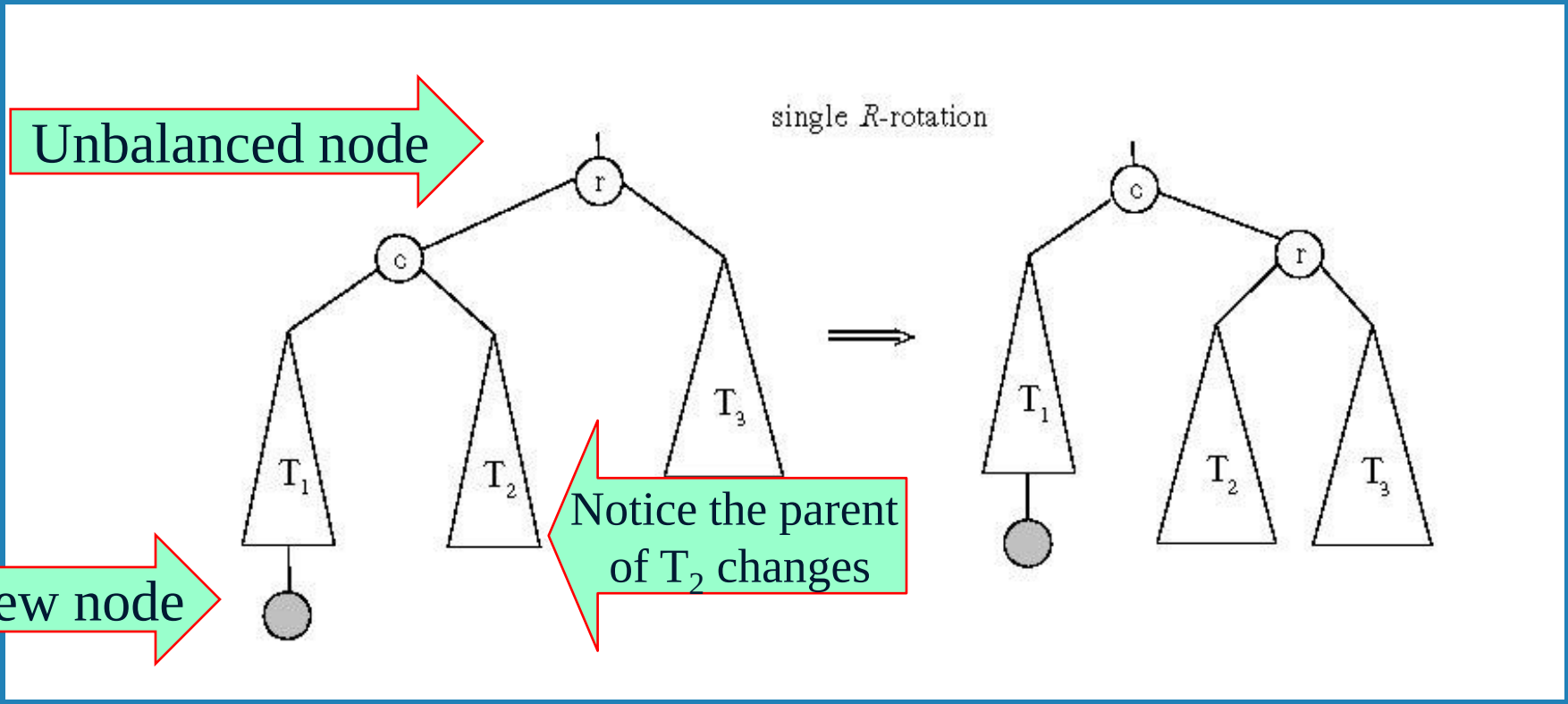
Single *R*-rotation



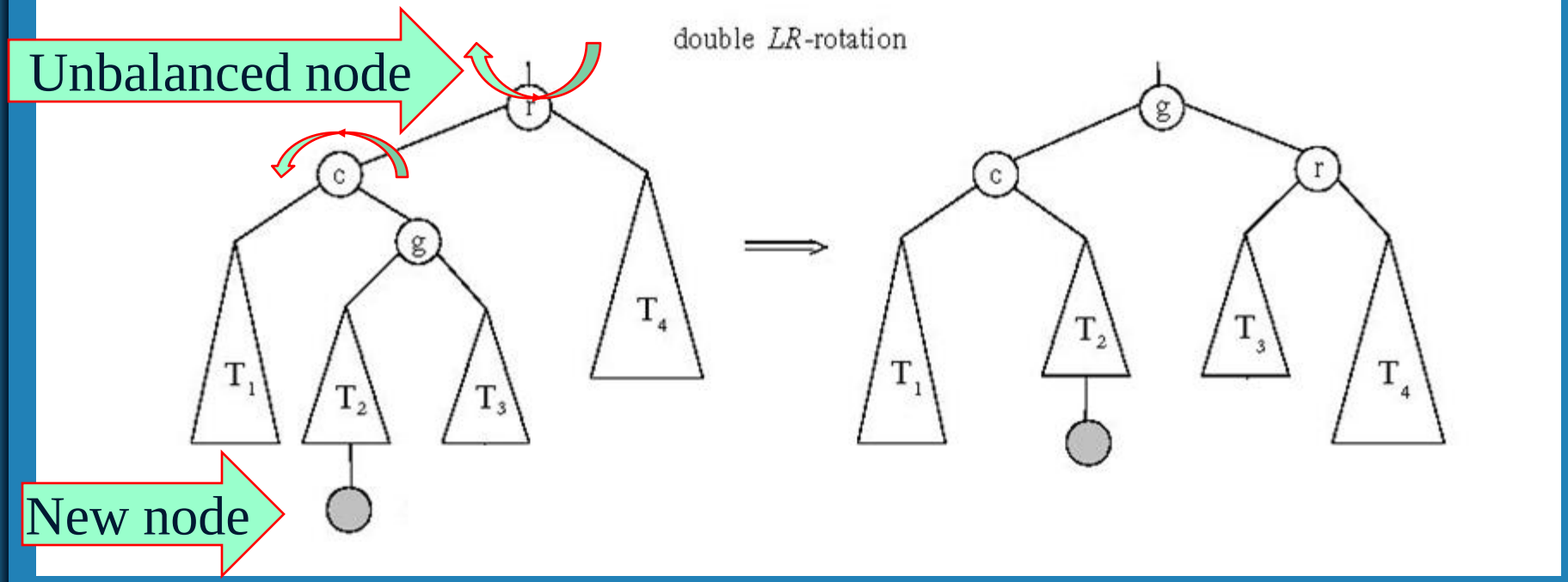
(c)

Double *LR*-rotation

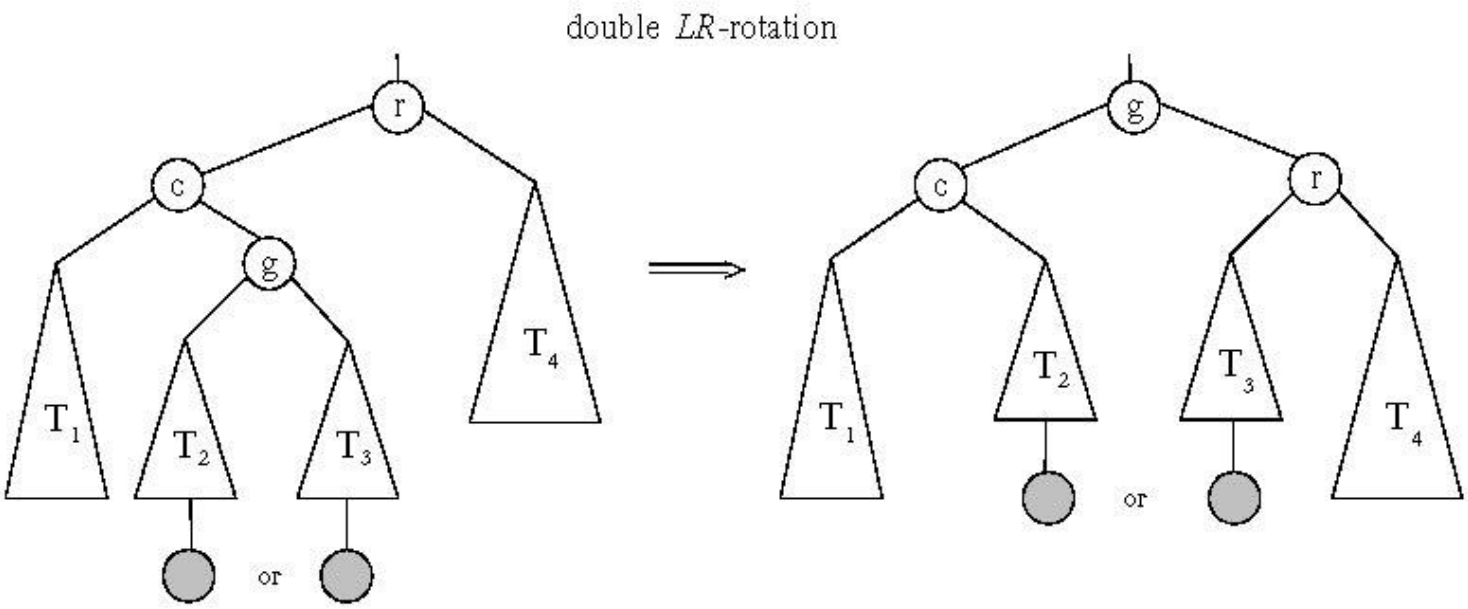
General case: Single R-rotation



General case: Double LR-rotation

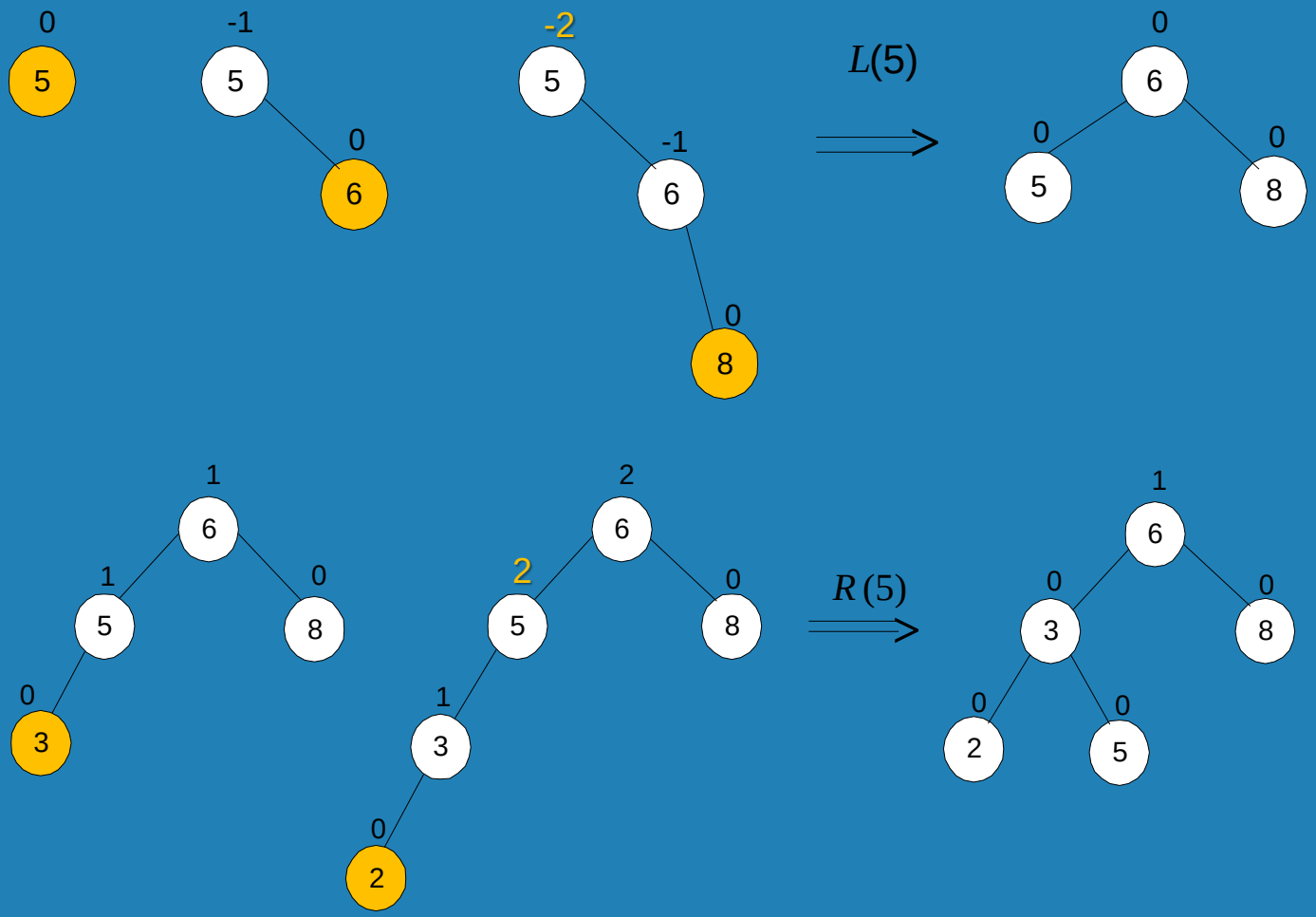


General case: Double LR-rotation

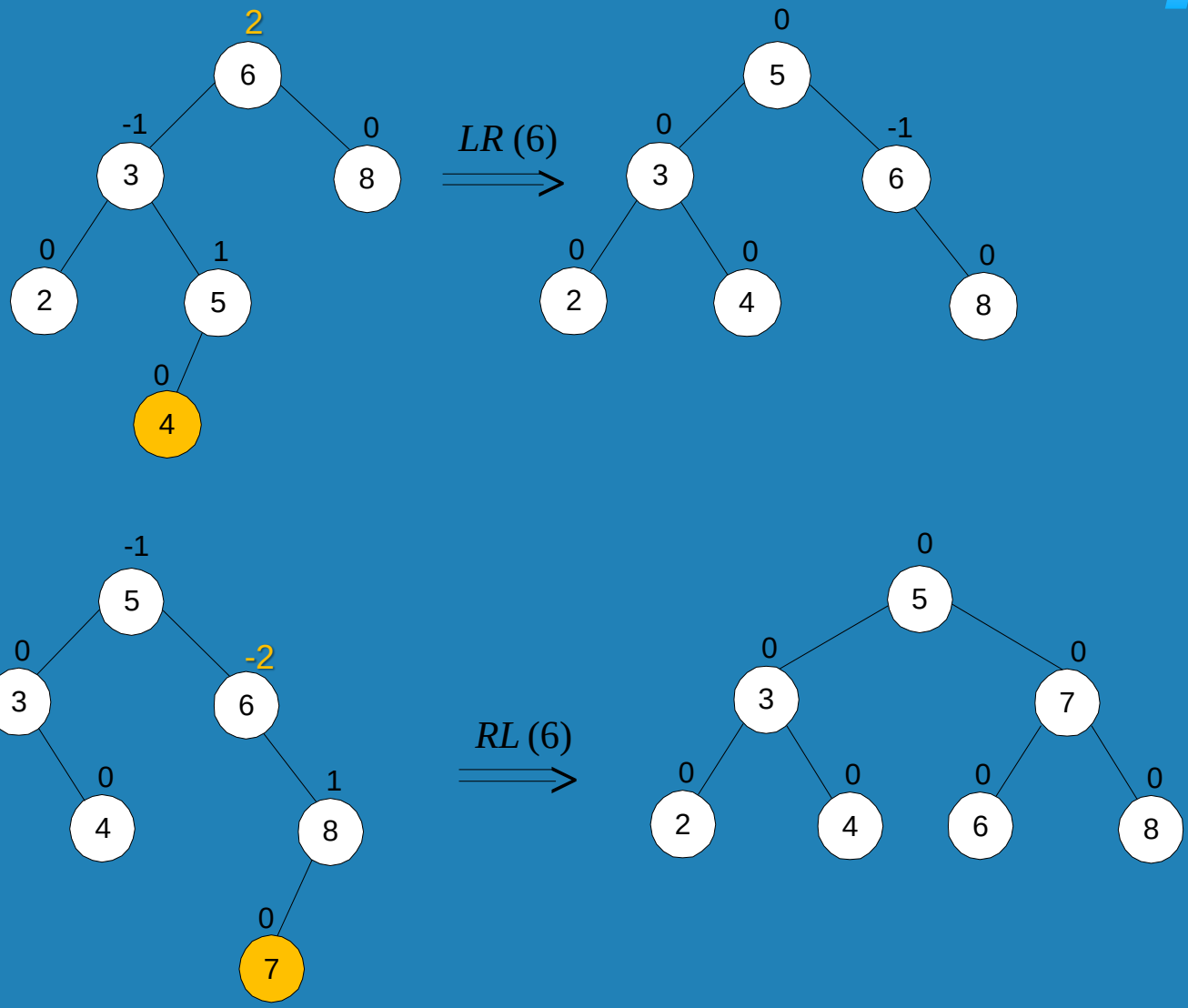


AVL tree construction - an example

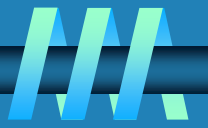
Construct an AVL tree for the list 5, 6, 8, 3, 2, 4, 7



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)