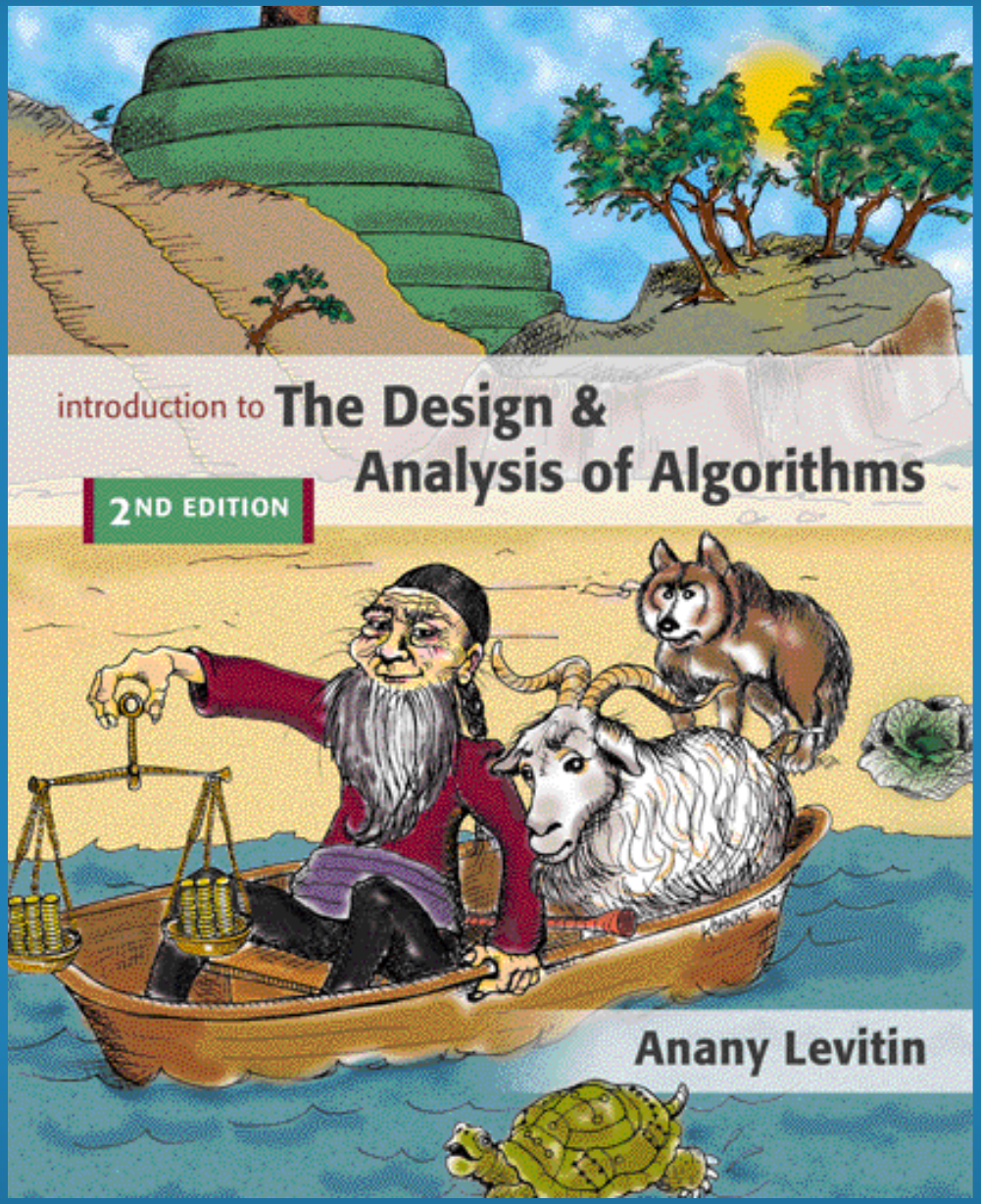
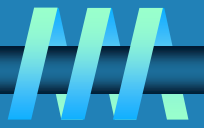
Chapter 6

Transform-and-Conquer



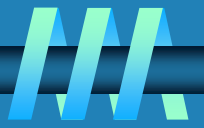
Transform and Conquer



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

- **to a simpler/more convenient instance of the same problem (*instance simplification*)**
- **to a different representation of the same instance (*representation change*)**
- **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

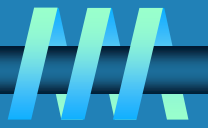
<u>Presorting</u>

Many problems involving lists are easier when list is sorted.

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

Efficiency of algorithms involving sorting depends on efficiency of sorting.

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

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

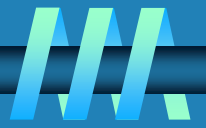
- **ALGORITHM** *PresortElementUniqueness*$(A[0..n-1])$

```
//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 ← 0 to n − 2 do
    if A[i] = A[i + 1] return false
return true
```

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
```

Taxonomy of Searching Algorithms



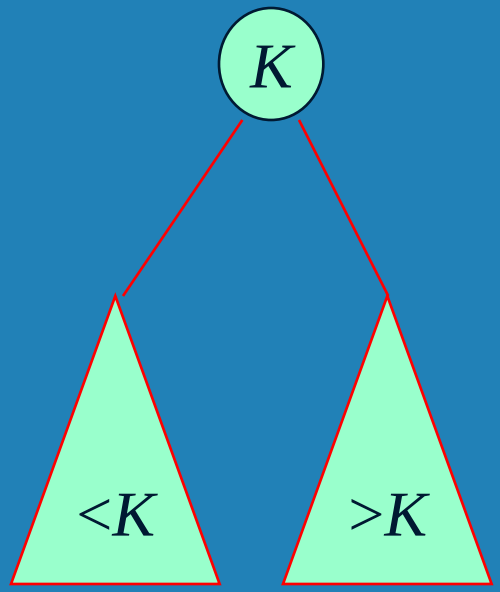
- List searching (good for static data)
  - sequential search
  - binary search

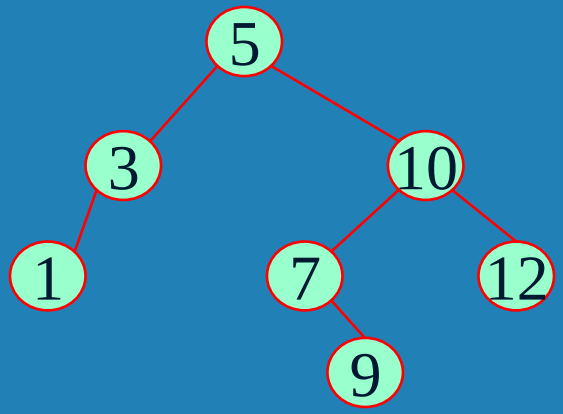
- Tree searching (good for dynamic data)
  - binary search tree
  - binary balanced trees: AVL trees, red-black trees
  - multiway balanced trees: 2-3 trees, 2-3-4 trees, B trees

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

Binary Search Tree

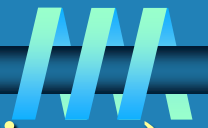
Arrange keys in a binary tree with the *binary search tree property*:





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

Balanced Search Trees



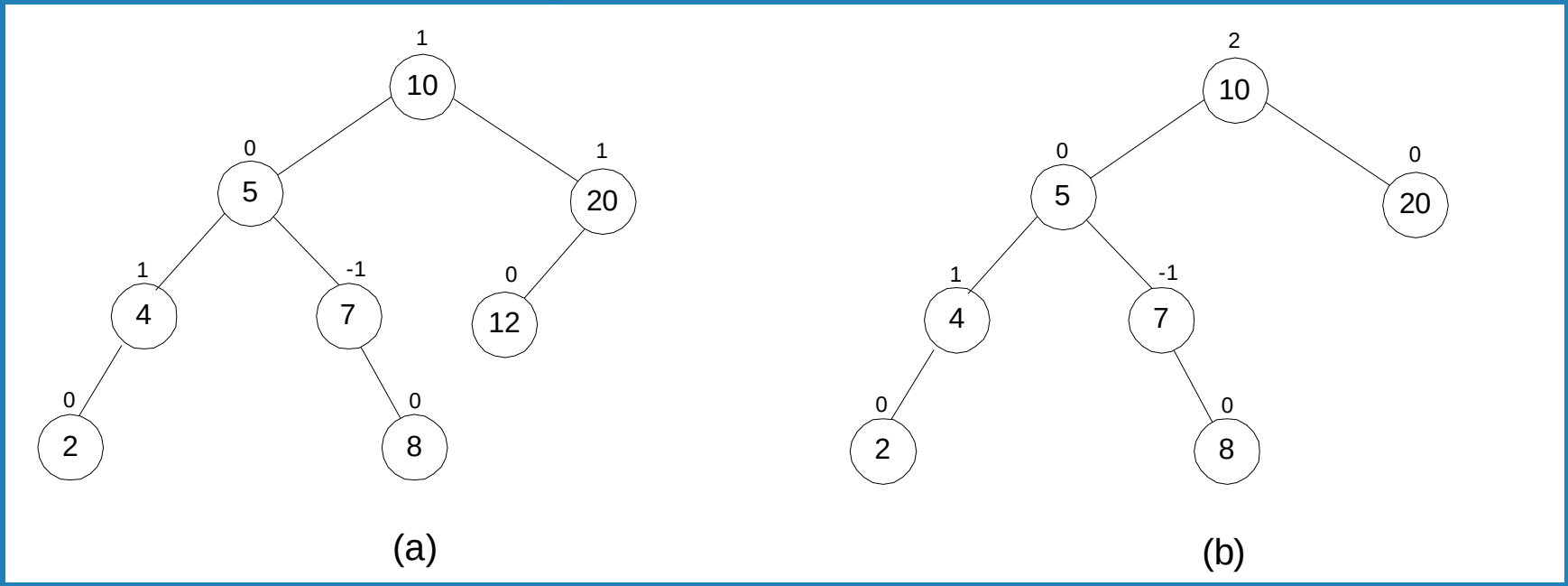
Attractiveness of *binary search tree* is marred by the bad (linear) worst-case efficiency. Two ideas to overcome it are:

- **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***

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

Balanced trees: AVL trees

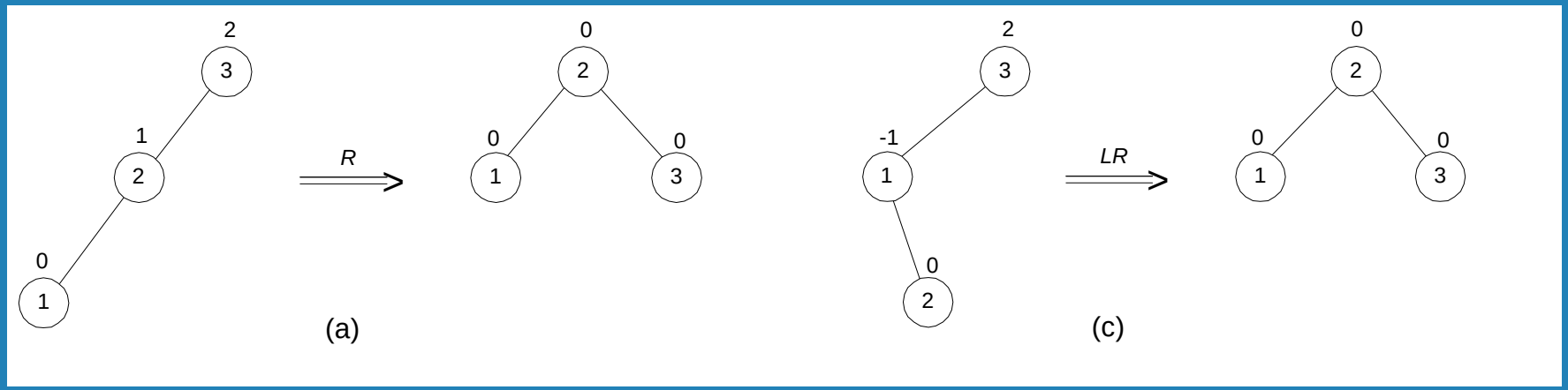
Definition An *AVL tree* is a binary search tree in which, for every node, the difference between the heights of its left and right subtrees, called the *balance factor*, is at most 1 (with the height of an empty tree defined as -1)



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

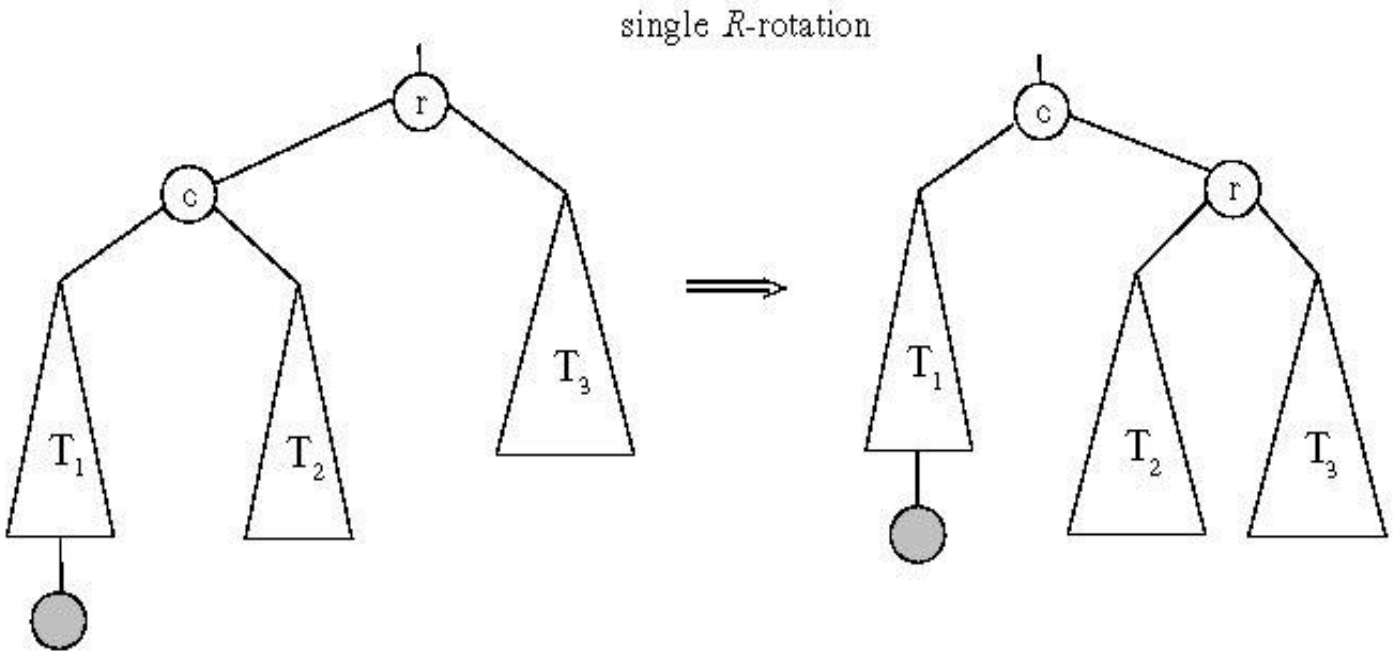
Rotations

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.)

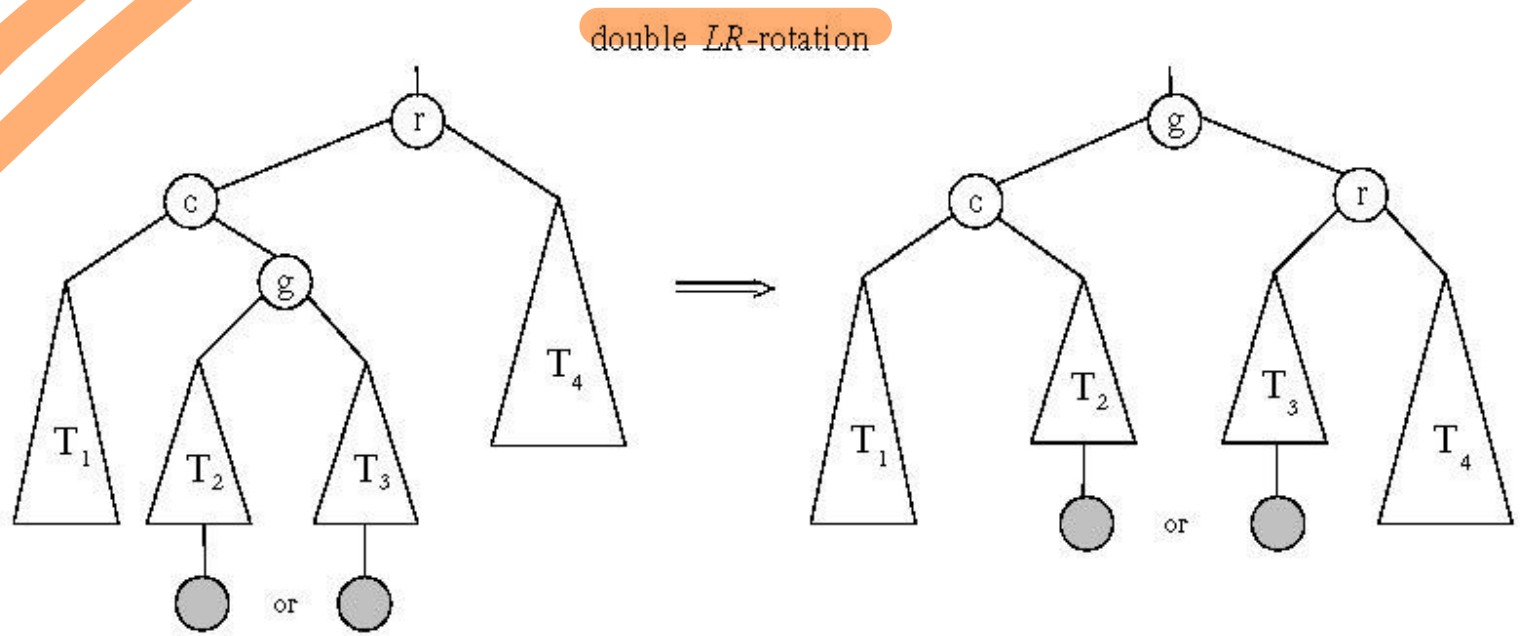


Single *R*-rotation **Double *LR*-rotation**

General case: Single R-rotation

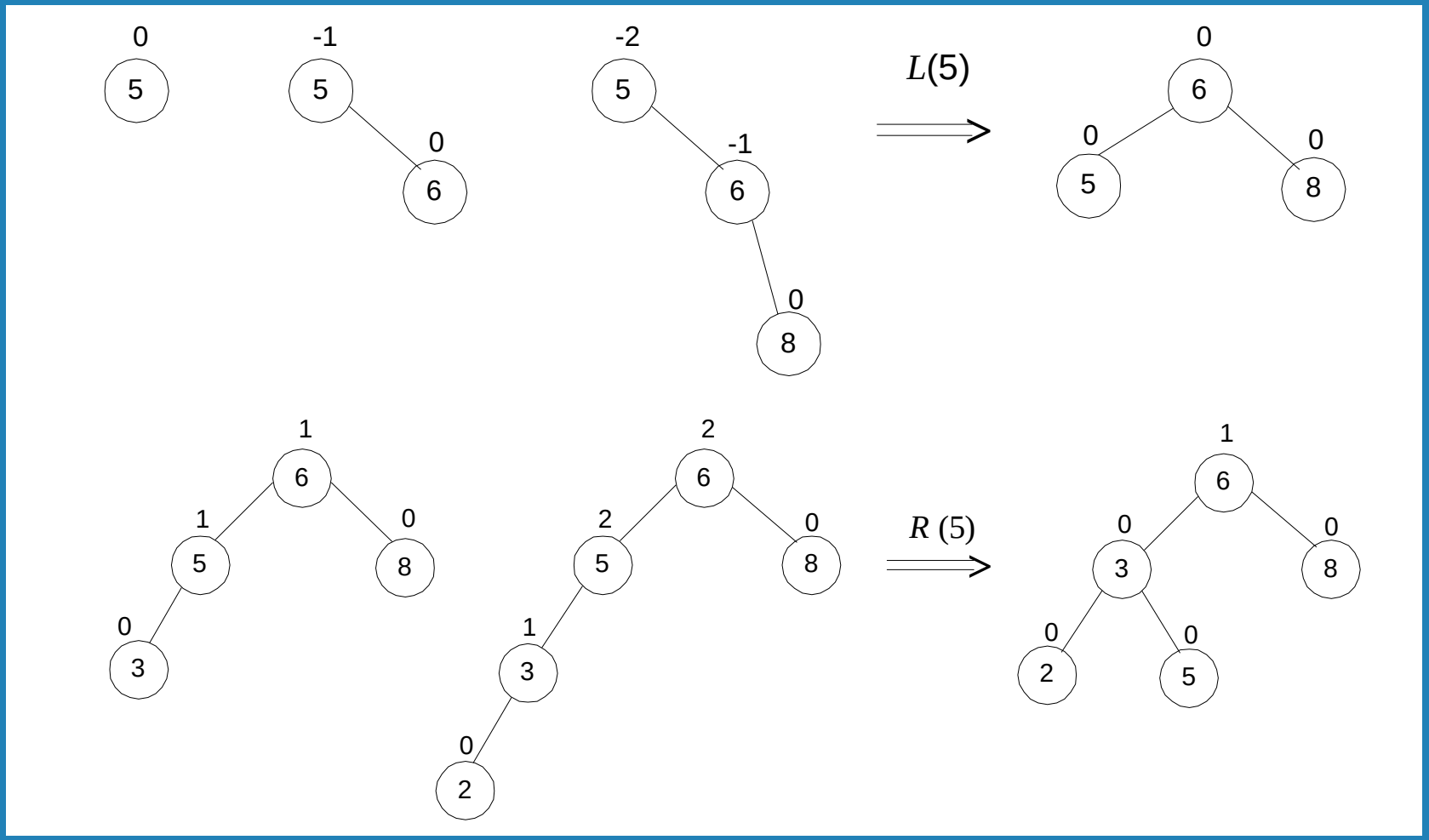


General case: Double LR-rotation

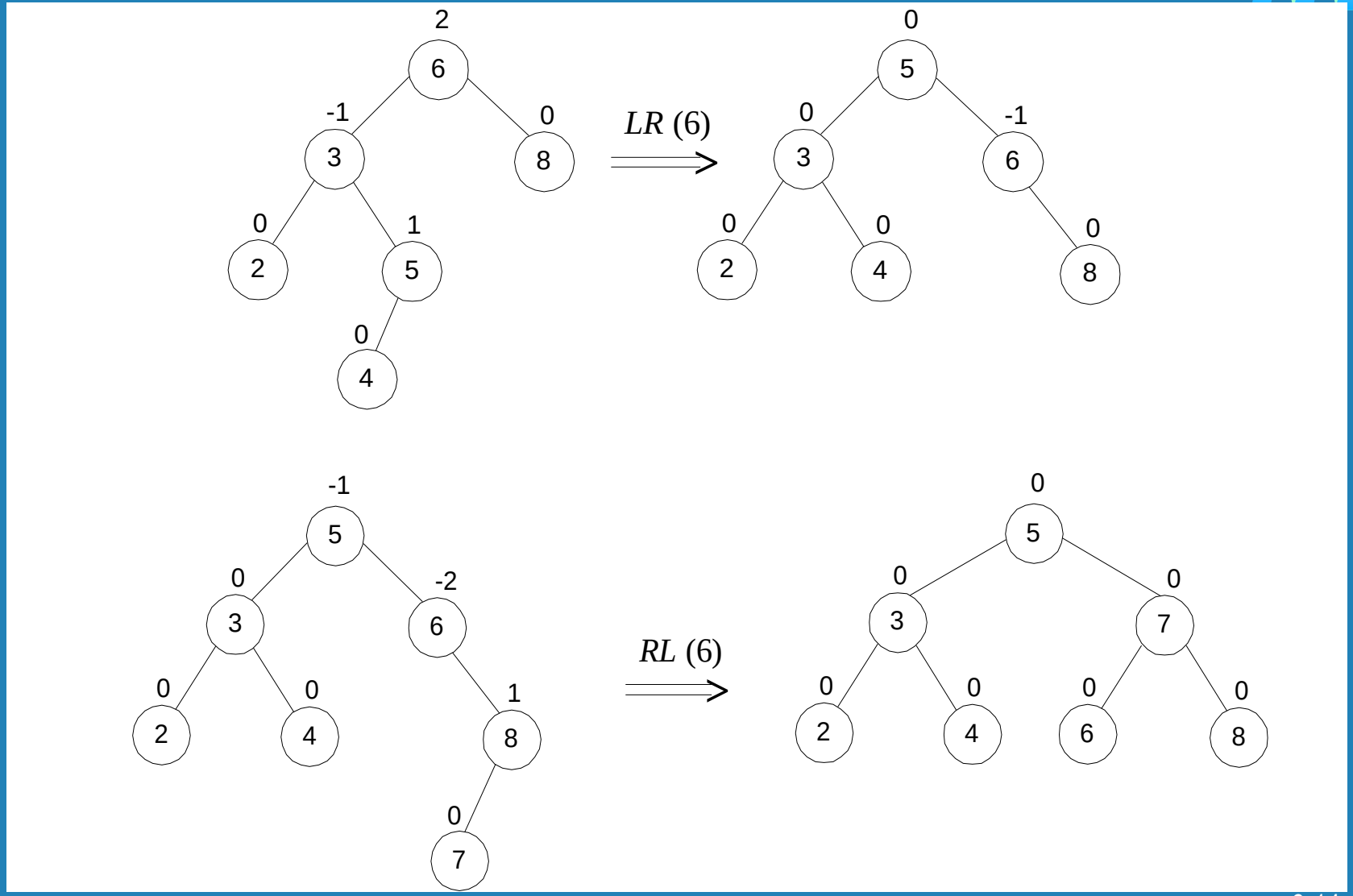


AVL tree construction - an example

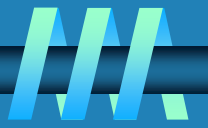
Construct an AVL tree for the list 5, 6, 8, 3, 2, 4, 7



AVL tree construction - an example (cont.)



Analysis of AVL trees

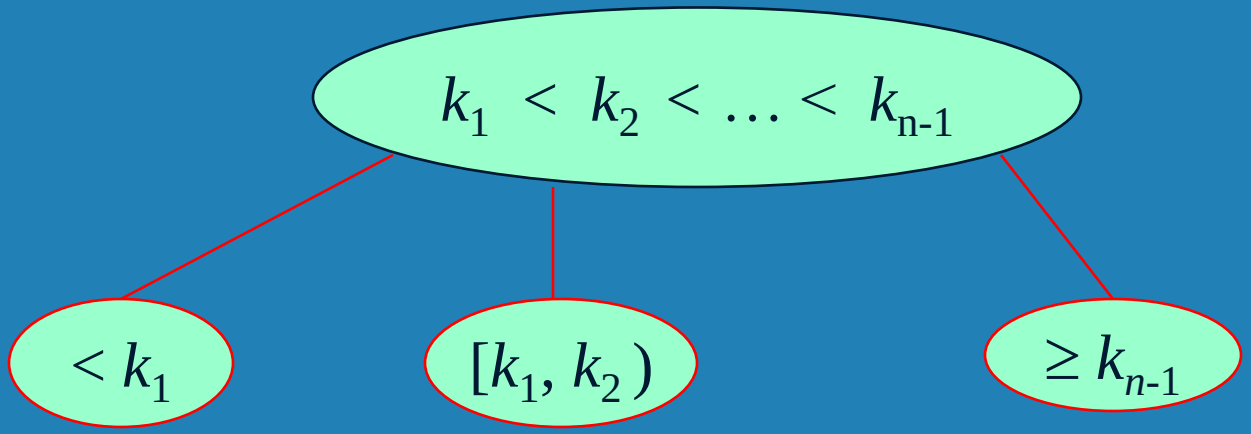


- Search and insertion are O(log *n*)
- Deletion is more complicated but is also O(log *n*)
- Disadvantages:
  - frequent rotations
  - complexity

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):

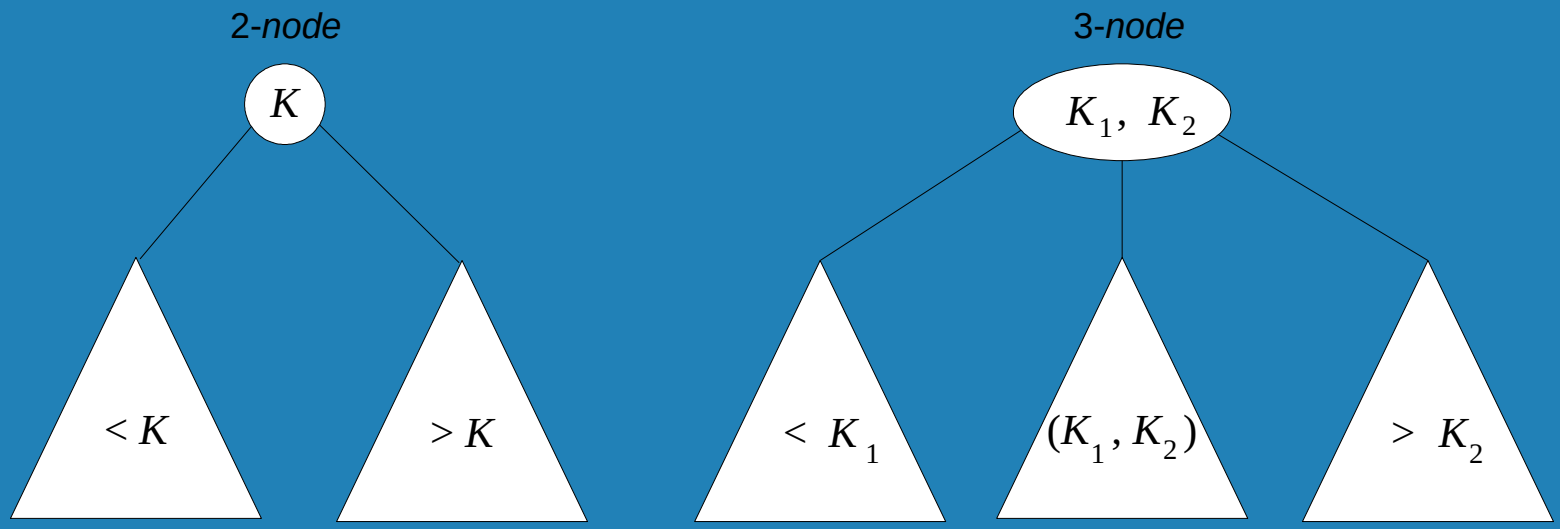


Note: Every node in a classical binary search tree is a 2-node

2-3 Tree

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

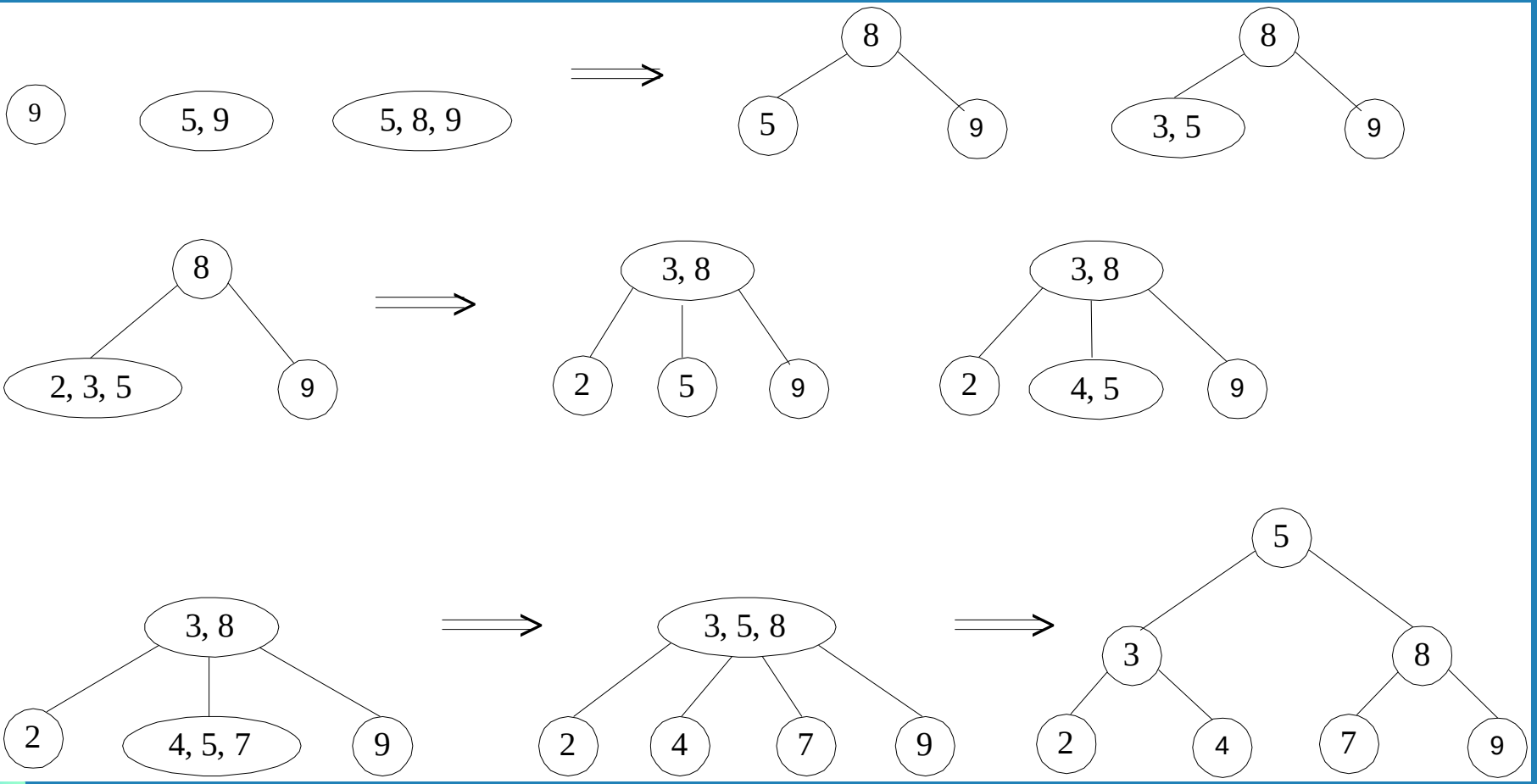
- may have 2-nodes and 3-nodes
- height-balanced (all leaves are on the same level)



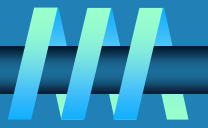
A 2-3 tree is constructed by successive insertions of keys given, with a new key always inserted into a leaf of the tree. If the leaf is a 3-node, it's split into two with the middle key promoted to the parent.

2-3 tree construction – an example

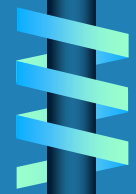
Construct a 2-3 tree the list 9, 5, 8, 3, 2, 4, 7



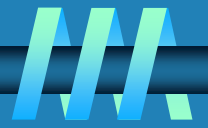
Analysis of 2-3 trees



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

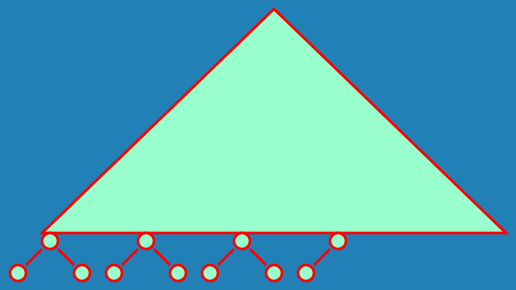


Heaps and Heapsort



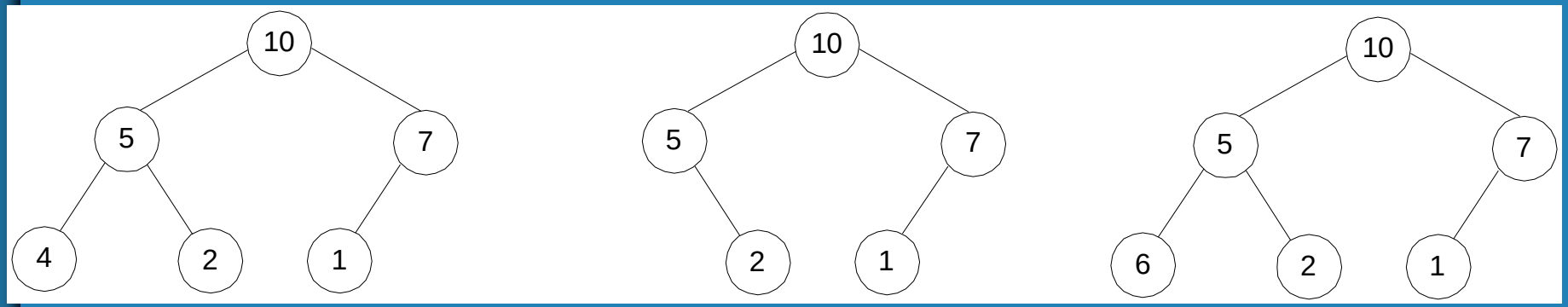
Definition A *heap* is a binary tree with keys at its nodes (one key per node) such that:

- It is essentially complete, i.e., all its levels are full except possibly the last level, where only some rightmost keys may be missing



- The key at each node is ≥ keys at its children (this is called a *max-heap*)

Illustration of the heap's definition



a heap **not a heap** **not a heap**

Note: Heap's elements are ordered top down (along any path down from its root), but they are not ordered left to right

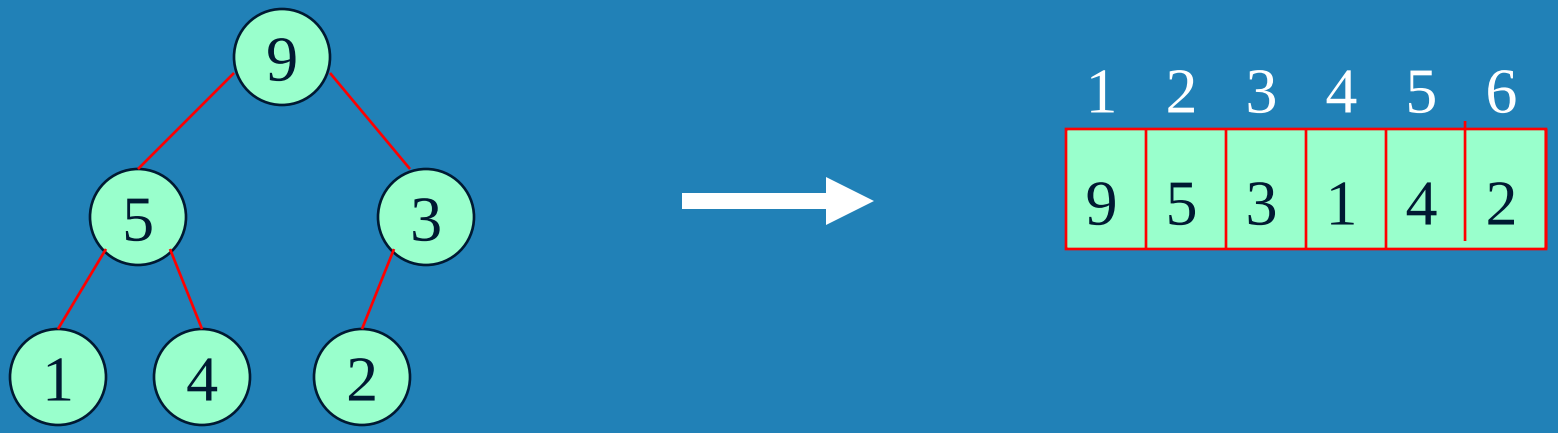
Some Important Properties of a Heap

1. There exists exactly one essentially complete binary tree with n nodes. Its height is equal to $\lfloor \log_2 n \rfloor$.
2. The root of a heap always contains its largest element.
3. A node of a heap considered with all its descendants is also a heap.
4. A heap can be implemented as an array by recording its elements in the top-down, left-to-right fashion. It is convenient to store the heap's elements in positions 1 though n of such an array, leaving $H[0]$ either unused or putting there a sentinel whose value is greater than every element in the heap. In such a representation,
   a. the parental node keys will be in the first $\lfloor n/2 \rfloor$ positions of the array, while the leaf keys will occupy the last $\lceil n/2 \rceil$ positions;
   b. the children of a key in the array's parental position i $(1 \le i \le \lfloor n/2 \rfloor)$ will be in positions $2i$ and $2i + 1$, and, correspondingly, the parent of a key in position i $(2 \le i \le n)$ will be in position $\lfloor i/2 \rfloor$.

Heap's Array Representation

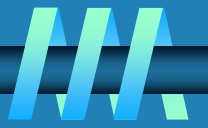
Store heap's elements in an array (whose elements indexed, for convenience, 1 to *n*) in top-down left-to-right order

Example:



- **Left child of node *j* is at 2*j***
- **Right child of node *j* is at 2*j*+1**
- **Parent of node *j* is at $\lfloor j/2 \rfloor$**
- **Parental nodes are represented in the first $\lfloor n/2 \rfloor$ locations**

Heap Construction (bottom-up)



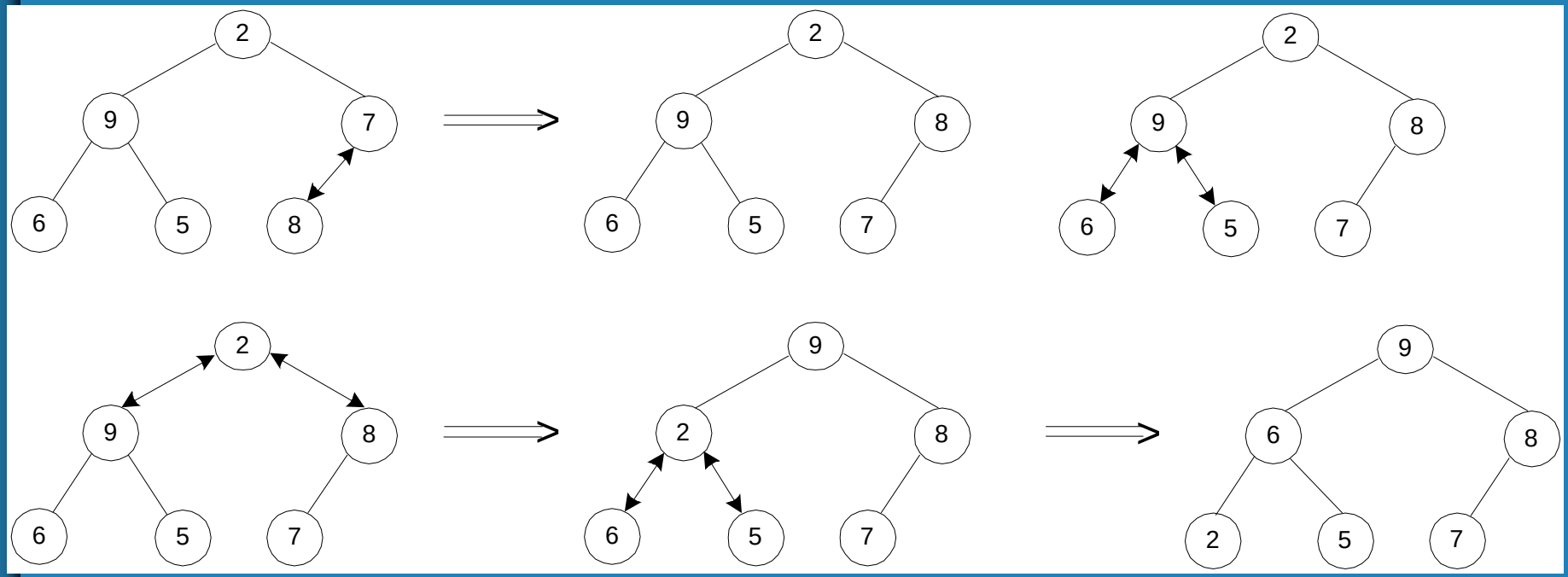
Step 0: Initialize the structure with keys in the order given

Step 1: Starting with the last (rightmost) parental node, fix the heap rooted at it, if it doesn't satisfy the heap condition: keep exchanging it with its larger child until the heap condition holds

Step 2: Repeat Step 1 for the preceding parental node

Example of Heap Construction

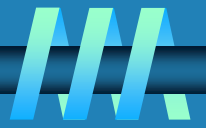
Construct a heap for the list 2, 9, 7, 6, 5, 8



Pseudocode of bottom-up heap construction

Algorithm *HeapBottomUp*$(H[1..n])$
//Constructs a heap from the elements of a given array
// by the bottom-up algorithm
//Input: An array $H[1..n]$ of orderable items
//Output: A heap $H[1..n]$
for $i \leftarrow \lfloor n/2 \rfloor$ **downto** 1 **do**
    $k \leftarrow i$; $v \leftarrow H[k]$
    $heap \leftarrow$ **false**
    while not $heap$ **and** $2 * k \leq n$ **do**
        $j \leftarrow 2 * k$
        if $j < n$ //there are two children
            if $H[j] < H[j+1]$ $j \leftarrow j+1$
        if $v \geq H[j]$
            $heap \leftarrow$ **true**
        else $H[k] \leftarrow H[j]$; $k \leftarrow j$
    $H[k] \leftarrow v$

Heapsort



Stage 1: Construct a heap for a given list of *n* keys

Stage 2:

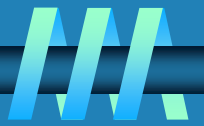
Maximum Key Deletion from a heap

Step 1 Exchange the root's key with the last key K of the heap.

Step 2 Decrease the heap's size by 1.

Step 3 "Heapify" the smaller tree by sifting K down the tree exactly in the same way we did it in the bottom-up heap construction algorithm. That is, verify the parental dominance for K: if it holds, we are done; if not, swap K with the larger of its children and repeat this operation until the parental dominance condition holds for K in its new position.

Example of Sorting by Heapsort



Sort the list 2, 9, 7, 6, 5, 8 by heapsort

Stage 1 (heap construction)

2 9 7 6 5 8

2 9 8 6 5 7

2 9 8 6 5 7

9 2 8 6 5 7

9 6 8 2 5 7

Stage 2 (root/max removal)

9 6 8 2 5 | 7

7 6 8 2 5 | 9

8 6 7 2 5 | 9

5 6 7 2 | 8 9

7 6 5 2 | 8 9

2 6 5 | 7 8 9

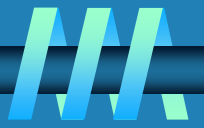
6 2 5 | 7 8 9

5 2 | 6 7 8 9

5 2 | 6 7 8 9

2 | 5 6 7 8 9

Analysis of Heapsort



Stage 1: Build heap for a given list of n keys

worst-case

$$C(n) = \sum_{i=0}^{h-1} 2(h-i)\, 2^i = 2\,(n - \log_2(n+1)) \in \Theta(n)$$

nodes at level i

Stage 2: Repeat operation of root removal n-1 times (fix heap)

worst-case

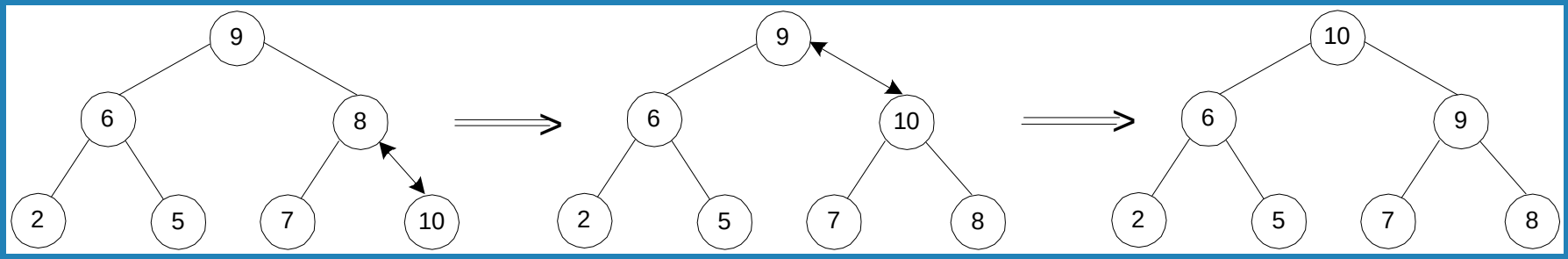
$$C(n) = \sum_{i=1}^{n-1} 2\log_2 i \in \Theta(n \log n)$$

Both worst-case and average-case efficiency: $\Theta(n \log n)$
In-place: yes
Stability: no (e.g., 1 1)

Insertion of a New Element into a Heap

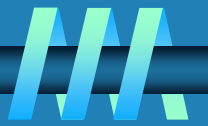
- Insert the new element at last position in heap.
- Compare it with its parent and, if it violates heap condition, exchange them
- Continue comparing the new element with nodes up the tree until the heap condition is satisfied

Example: Insert key 10



Efficiency: O(log *n*)

Exercise



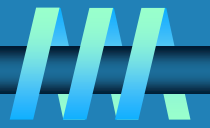
1. Sort the following lists by heapsort by using the array representation of heaps.

a) 1, 2, 3, 4, 5 (in increasing order)

b) 5, 4, 3, 2, 1 (in increasing order)

c) S, O, R, T, I, N, G (in alphabetical order)

Problem Reduction



This variation of transform-and-conquer solves a problem by transforming it into different problem for which an algorithm is already available.

To be of practical value, the combined time of the transformation and solving the other problem should be smaller than solving the problem as given by another method.

Examples of Solving Problems by Reduction

- **computing lcm(*m*, *n*) via computing gcd(*m*, *n*)**

  $$lcm(m,n)*gcd(m,n) = m*n$$

- **counting number of paths of length *n* in a graph by raising the graph's adjacency matrix to the *n*-th power**

- **transforming a maximization problem to a minimization problem and vice versa (also, min-heap construction)**

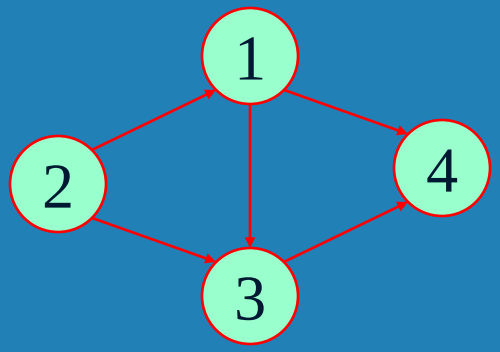
- **linear programming**

- **reduction to graph problems (e.g., solving puzzles via state-space graphs)**

Examples of Solving Problems by Reduction

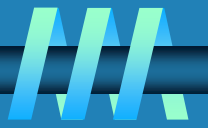
- **Counting number of paths of length *n* in a graph by raising the graph's adjacency matrix to the *n*-th power**

  If (directed) graph *G* has adjacency matrix *A*, then for any *k*, the (*i*,*j*) entry in *A*^*k* gives the number of paths from vertex *i* and *j* of length *k*. (Levitin, Ch. 6.6)



$$A = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \qquad A^2 = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

reduction to graph problems



A peasant finds himself on a river bank with a wolf, a goat, and a head of cabbage. He needs to transport all three to the other side of the river in his boat. However, the boat has room only for the peasant himself and one other item (either the wolf, the goat, or the cabbage). In his absence, the wolf would eat the goat, and the goat would eat the cabbage. Find a way for the peasant to solve his problem

reduction to graph problems

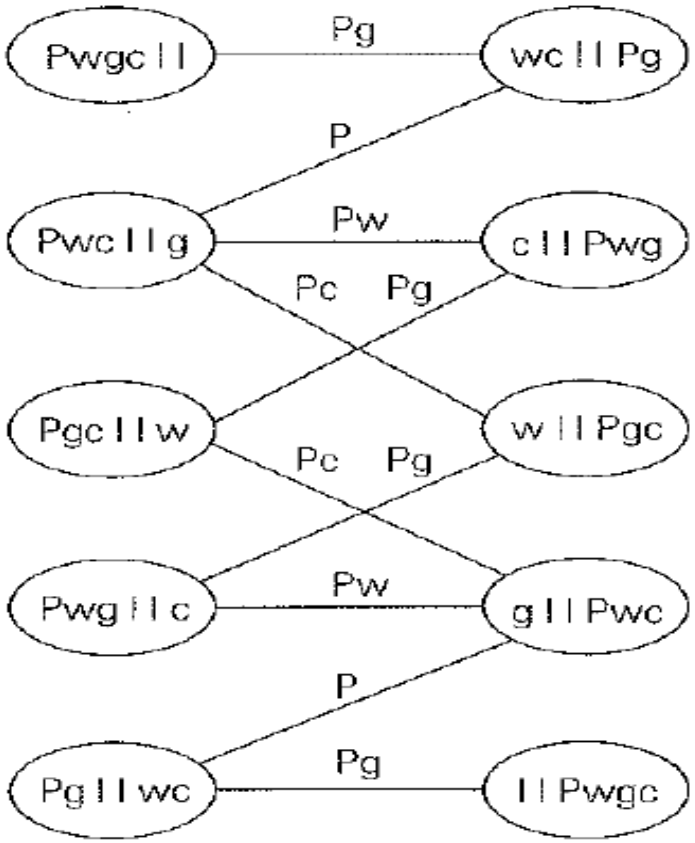


FIGURE 6.18 State-space graph for the peasant, wolf, goat, and cabbage puzzle