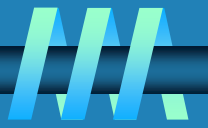
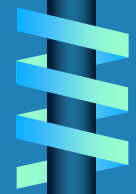
# Transform and Conquer

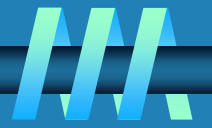


**This group of techniques solves a problem by a *transformation* to**

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

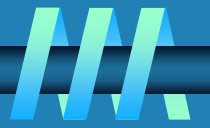
**Many problems involving lists are easier when list is sorted, e.g.**

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

**Also:**

- **Topological sorting helps solving some problems for dags.**
- **Presorting is used in many geometric algorithms.**

# How fast can we sort ?

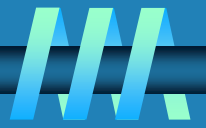


**Efficiency of algorithms involving sorting depends on efficiency of sorting.**

**Theorem (see Sec. 11.2): $\lceil \log_2 n! \rceil \approx n \log_2 n$ comparisons are necessary in the worst case to sort a list of size $n$ by any 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**

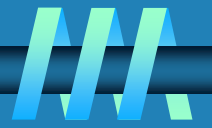
**Stage 2 Apply binary search**

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

**Good or bad?**

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

# 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**

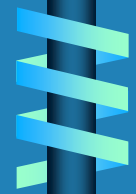
  **Efficiency: $\Theta(n \log n) + O(n) = \Theta(n \log n)$**

- **Brute force algorithm**

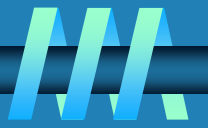
  **Compare all pairs of elements**

  **Efficiency: $O(n^2)$**

- **Another algorithm? Hashing**



# Searching Problem

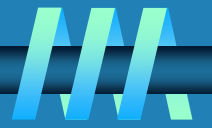


**Problem**: Given a (multi)set $S$ of keys and a search key $K$, find an occurrence of $K$ in $S$, if any

- 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



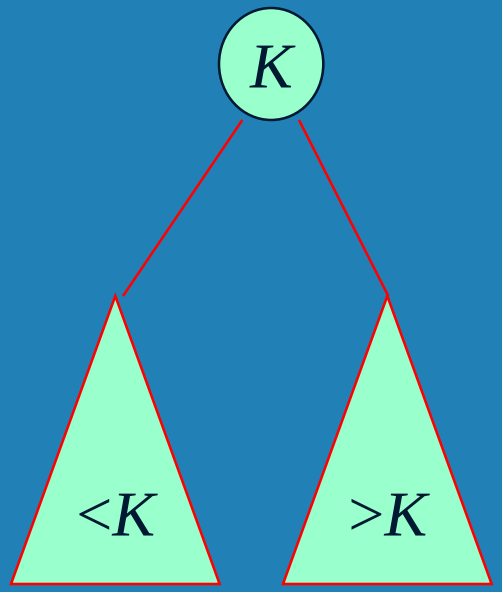
- List searching
  - sequential search
  - binary search
  - interpolation search

- 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)

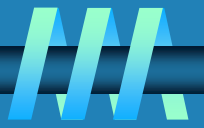
# Binary Search Tree

**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:**

- **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: $\lfloor \log_2 n \rfloor \le h \le n-1$, with height average (random files) be about $3\log_2 n$**

**Thus all three operations have**

- **worst case efficiency: $\Theta(n)$**
- **average case efficiency: $\Theta(\log n)$**

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

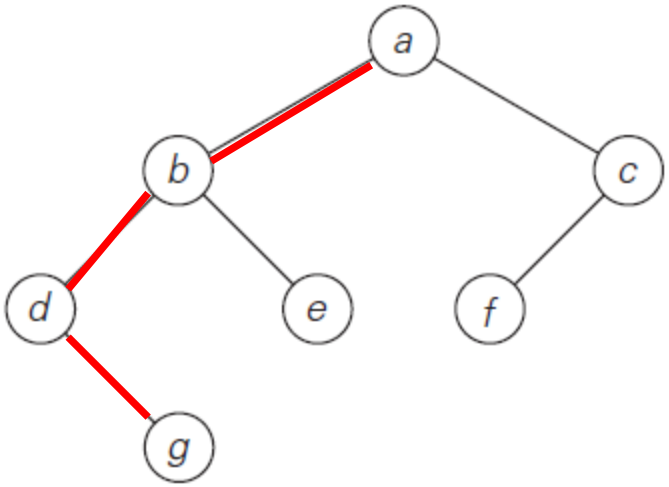
# Balanced Search Trees

**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***
- **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

**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)




H(a) = 3 (number of edges in the longest path)

**ALGORITHM** *Height*($T$)

//Computes recursively the height of a binary tree
//Input: A binary tree $T$
//Output: The height of $T$
**if** $T = \varnothing$ **return** $-1$
**else return** $\max\{Height(T_{left}), Height(T_{right})\} + 1$

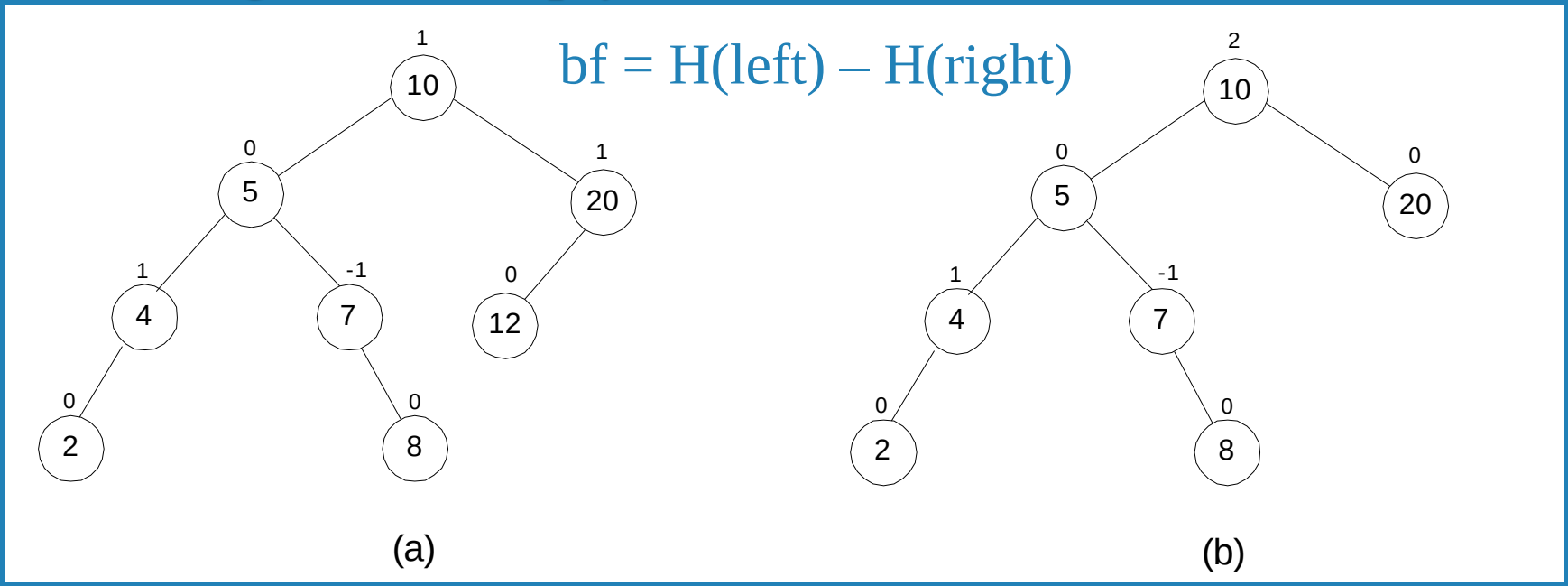
H(∅) = -1

H(Single Node) = 0

H(Node) = **max{**H(Node.left), H(Node.right)**}** + 1

# Balanced trees: AVL trees

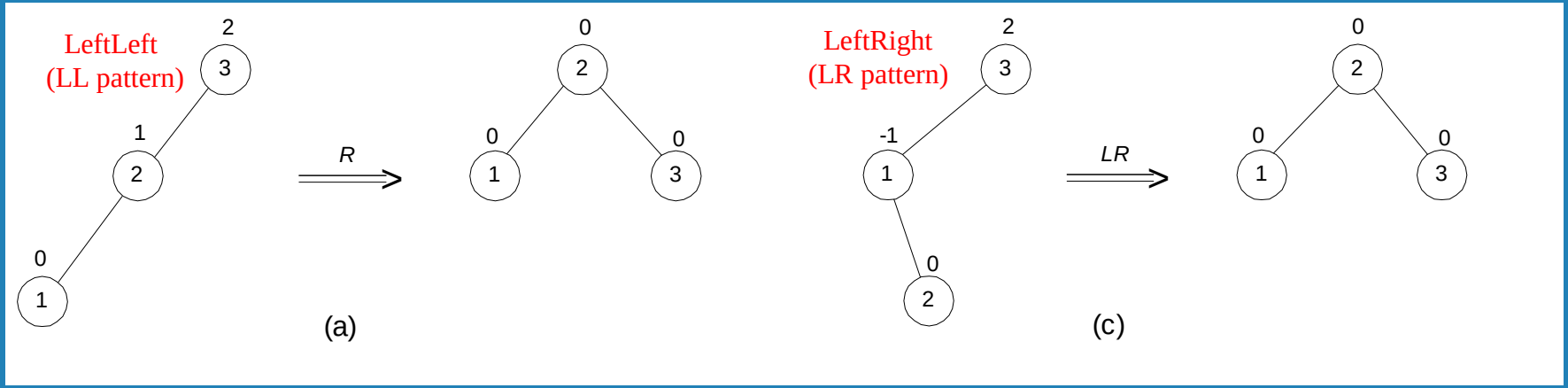
**Definition** An *AVL tree* is a binary search tree in which, for every node, the difference between the heights of its left and right subtrees, called the *balance factor*, is at most 1 (with the height of an empty tree defined as -1)




**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**

# Rotations

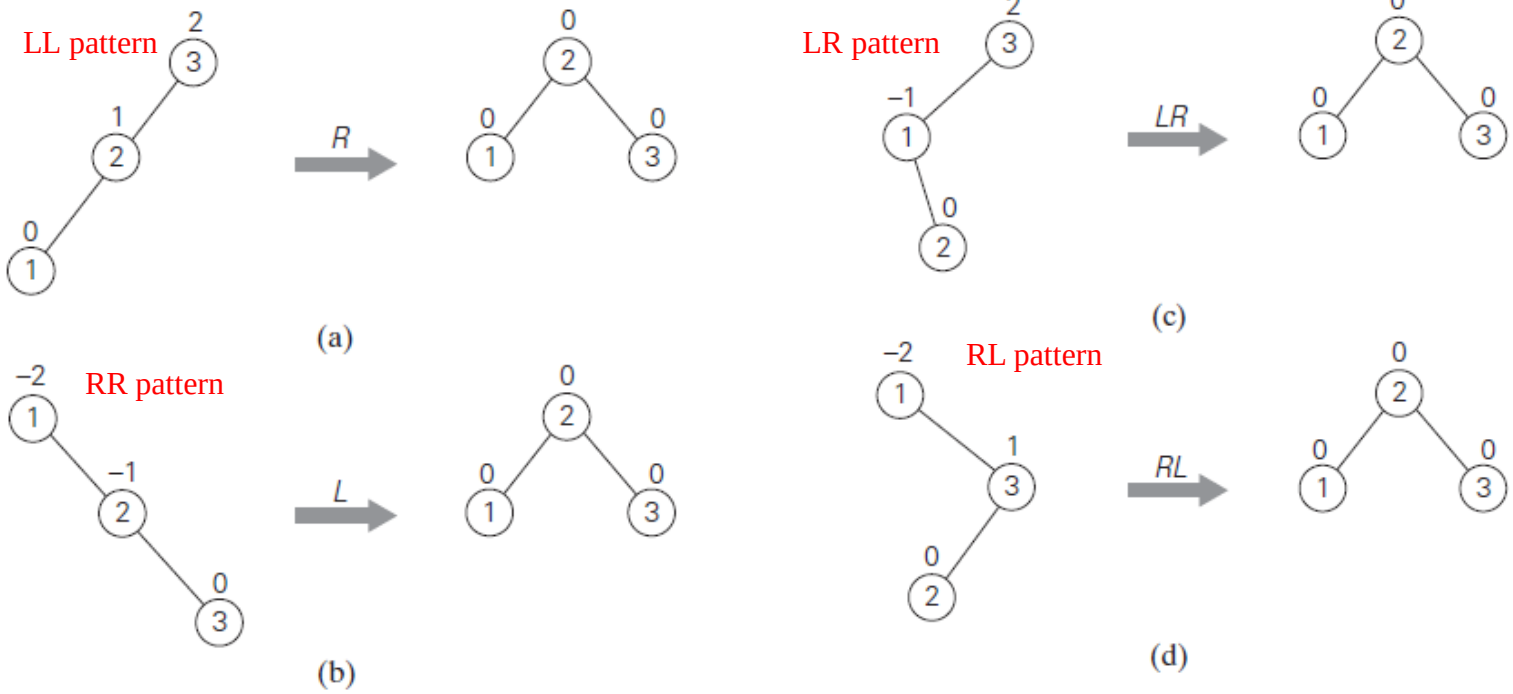



**FIGURE 6.3** Four rotation types for AVL trees with three nodes. (a) Single *R*-rotation. (b) Single *L*-rotation. (c) Double *LR*-rotation. (d) Double *RL*-rotation.

# General case: Single R-rotation

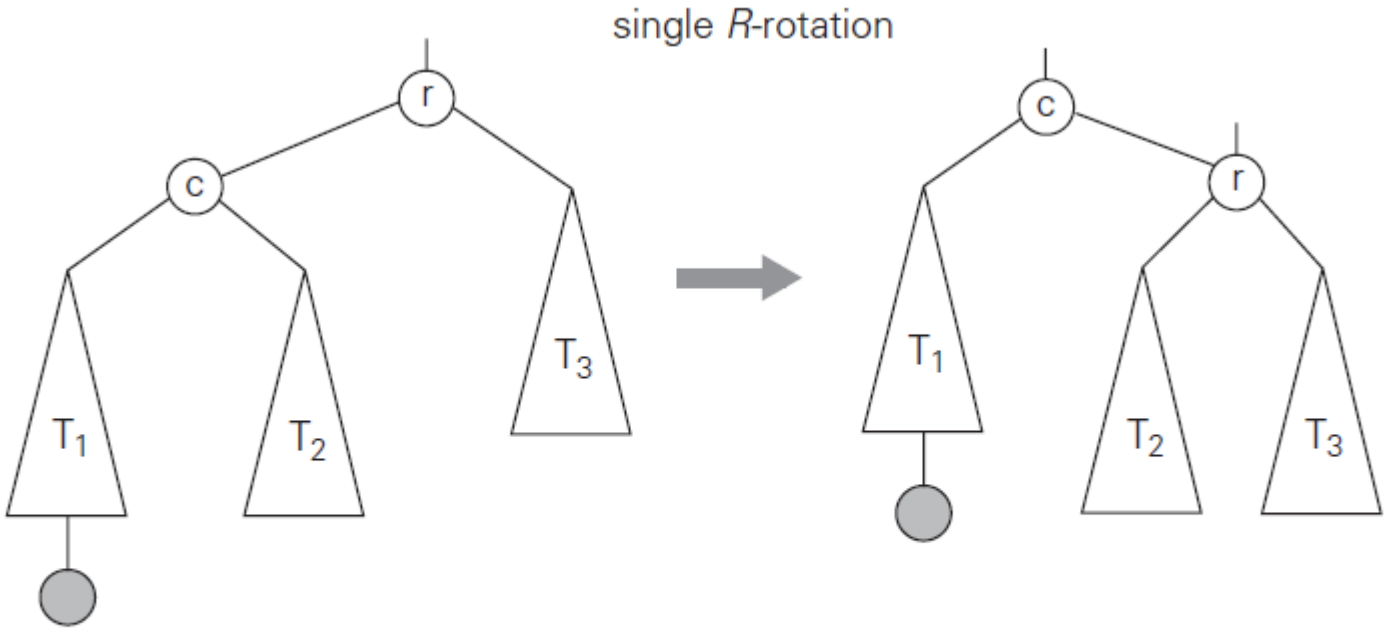



**FIGURE 6.4** General form of the *R*-rotation in the AVL tree. A shaded node is the last one inserted.

# General case: Double LR-rotation

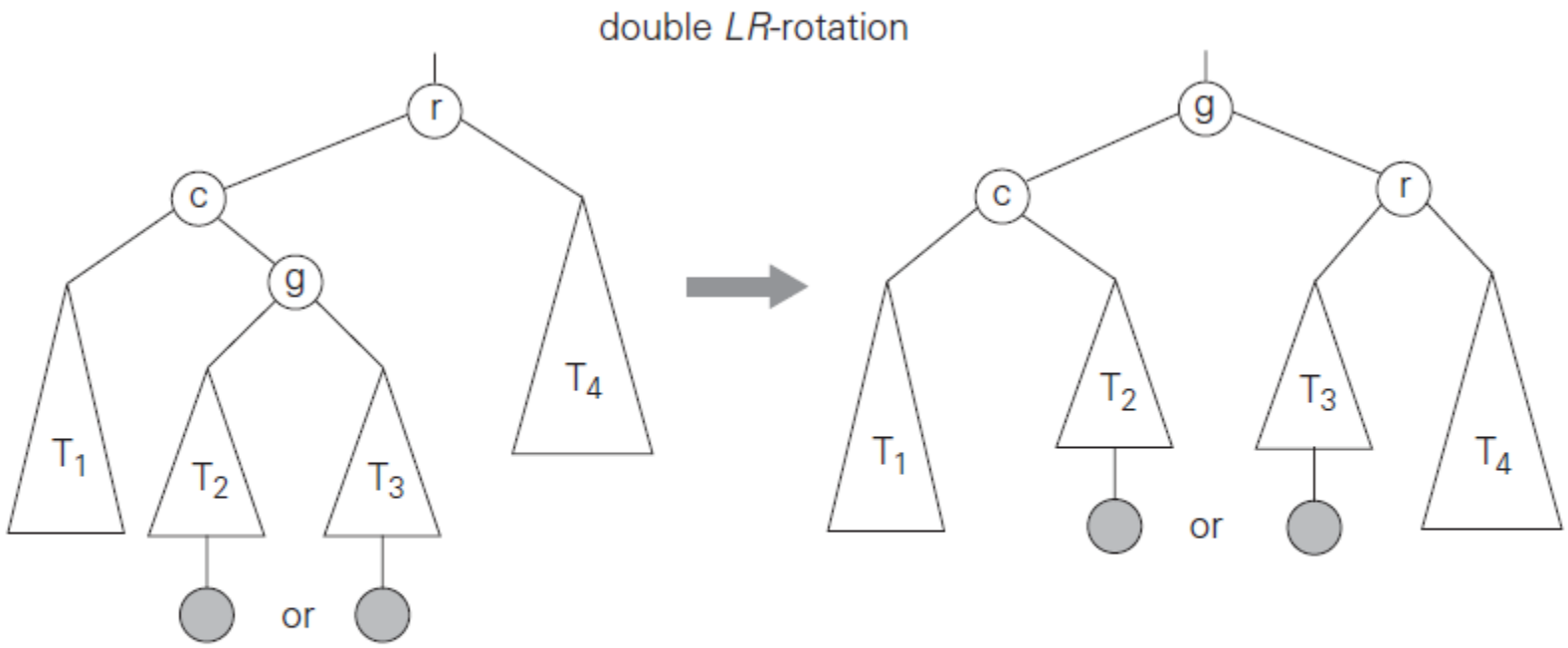
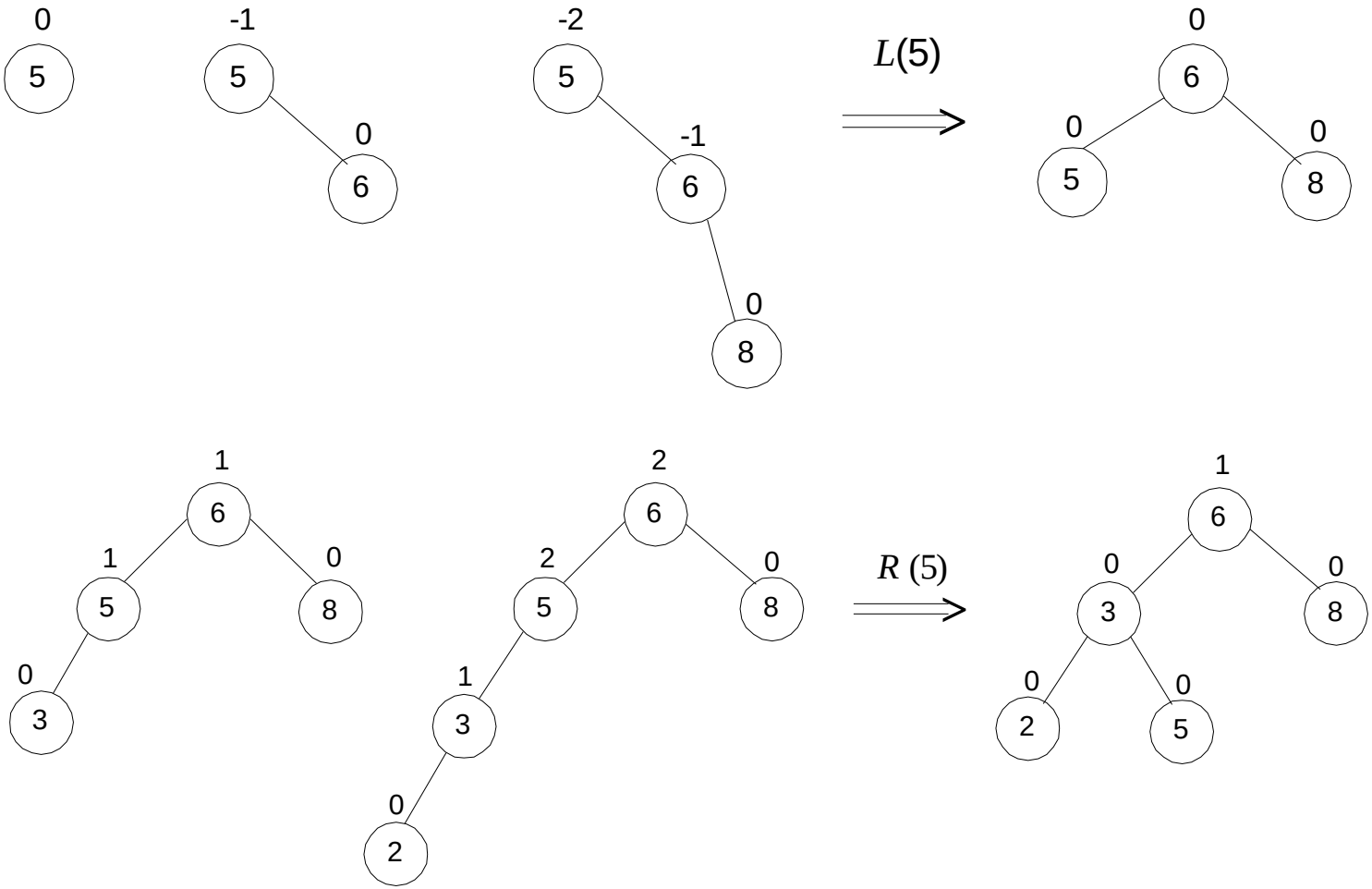



**FIGURE 6.5** General form of the double *LR*-rotation in the AVL tree. A shaded node is the last one inserted. It can be either in the left subtree or in the right subtree of the root's grandchild.

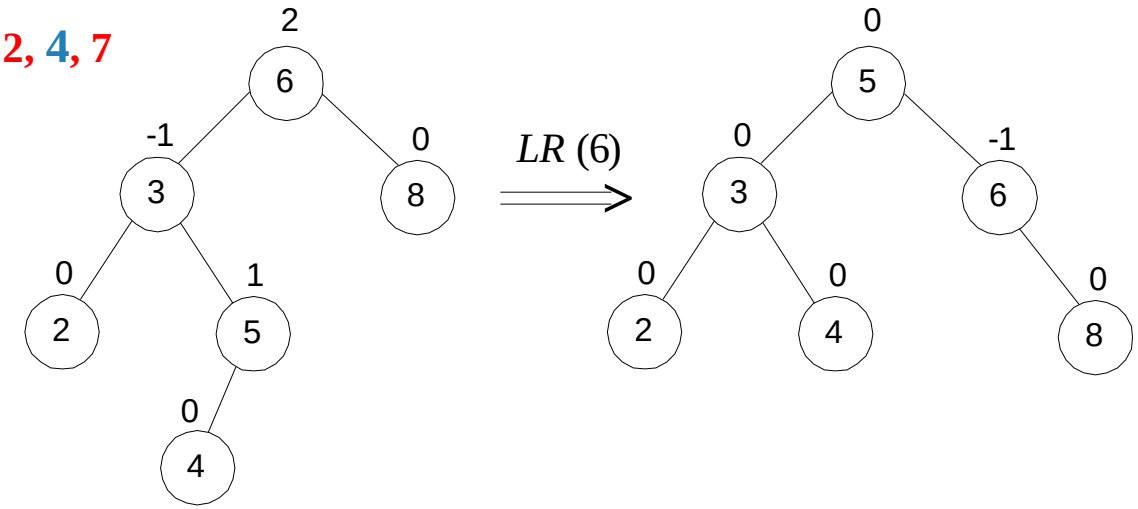
# AVL tree construction - an example

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

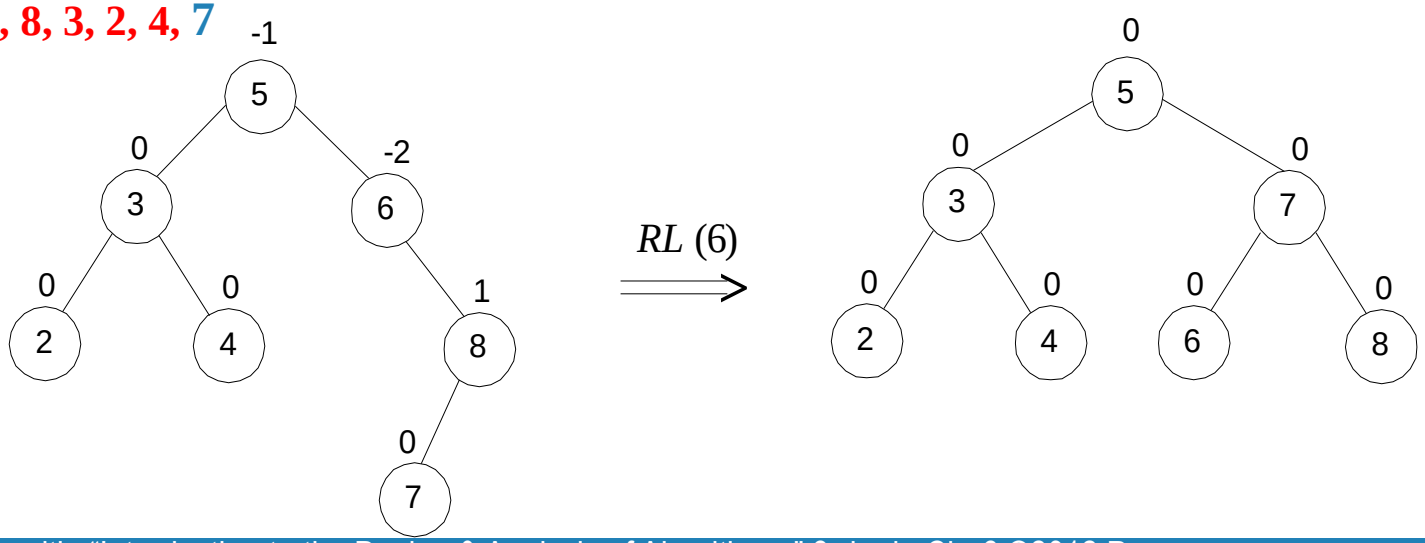


# AVL tree construction - an example (cont.)

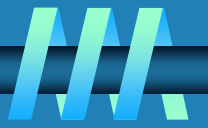
list: 5, 6, 8, 3, 2, 4, 7




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



# Analysis of AVL trees



- $h \leq 1.4404 \log_2 (n + 2) - 1.3277$

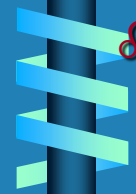
  average height: $1.01 \log_2 n + 0.1$ for large $n$ (found empirically)

- Search and insertion are O(log $n$)

- Deletion is more complicated but is also O(log $n$)

- Disadvantages:
  - frequent rotations
  - complexity

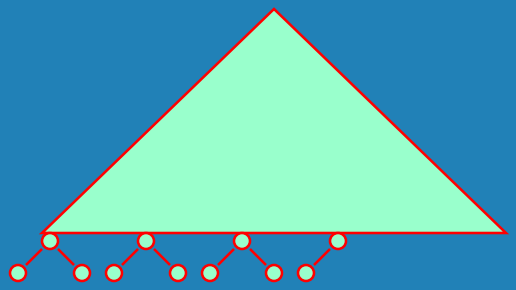
- A similar idea: *red-black trees* (height of subtrees is allowed to differ by up to a factor of 2)



# Heaps and Heapsort

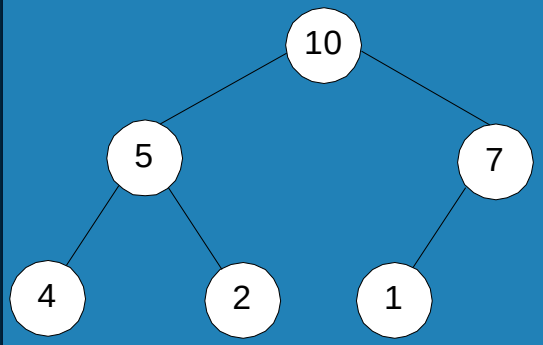
**Definition** A *heap* is a binary tree with keys at its nodes (one key per node) such that:

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

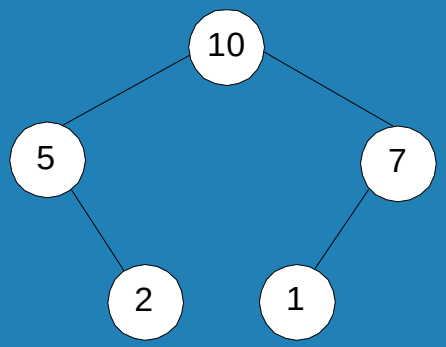


- The key at each node is ≥ keys at its children

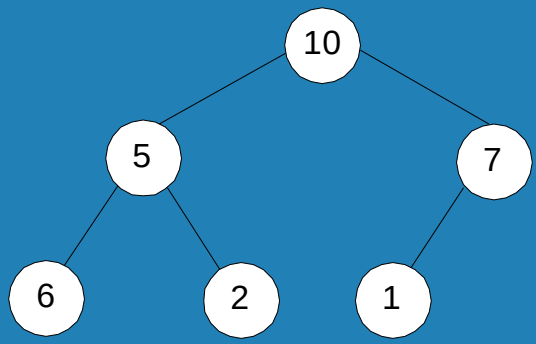
# Illustration of the heap's definition




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**

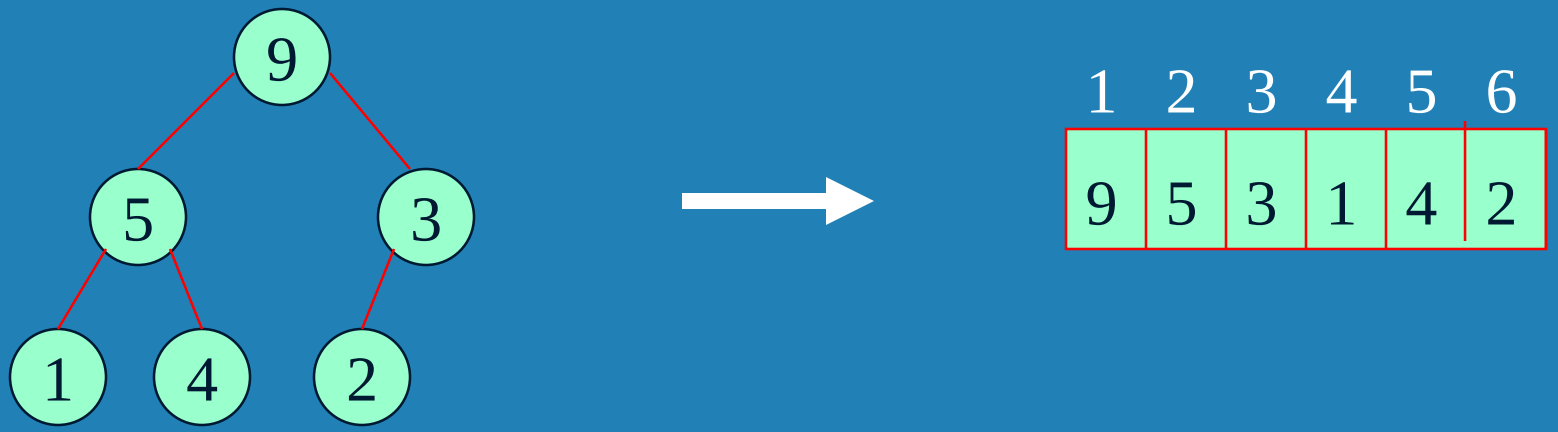
# Some Important Properties of a Heap

- Given $n$, there exists a unique binary tree with $n$ nodes that is essentially complete, with $h = \lfloor \log_2 n \rfloor$
- The root contains the largest key
- The subtree rooted at any node of a heap is also a heap
- A heap can be represented as an array

# Heap's Array Representation

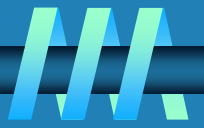
**Store heap's elements in an array (whose elements indexed, for convenience, 1 to *n*) in top-down left-to-right order**

**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)



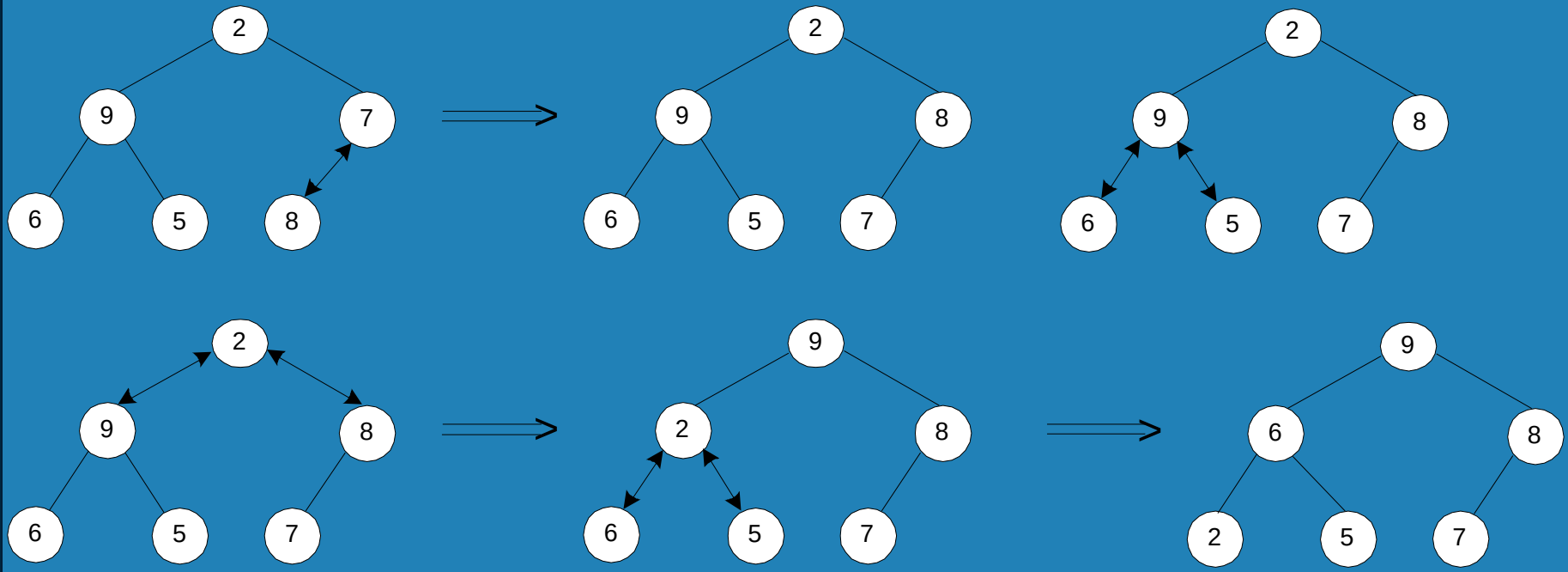
**Step 0: Initialize the structure with keys in the order given**

**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 largest child until the heap condition holds**

**Step 2: Repeat Step 1 for the preceding parental node**

# Example of Heap Construction

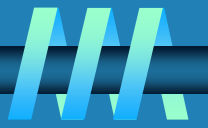
**Construct a heap for the list 2, 9, 7, 6, 5, 8**



# Pseudopodia 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**
    $k \leftarrow i; \quad v \leftarrow H[k]$
    $heap \leftarrow$ **false**
    **while not** $heap$ **and** $2 * k \leq n$ **do**
        $j \leftarrow 2 * k$
        **if** $j < n$ //there are two children
            **if** $H[j] < H[j+1] \quad j \leftarrow j+1$
        **if** $v \geq H[j]$
            $heap \leftarrow$ **true**
        **else** $H[k] \leftarrow H[j]; \quad k \leftarrow j$
    $H[k] \leftarrow v$

# Heapsort

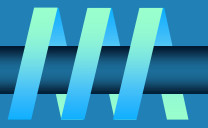


**Stage 1: Construct a heap for a given list of *n* keys**

**Stage 2: Repeat operation of root removal *n*-1 times:**

- **Exchange keys in the root and in the last (rightmost) leaf**
- **Decrease heap size by 1**
- **If necessary, swap new root with larger child until the heap condition holds**

# Example of Sorting by Heapsort



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

**Stage 1 (heap construction)**

1 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

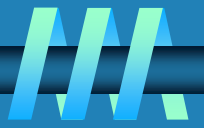
6 2 5 | 7 8 9

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)**

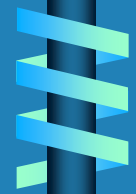
**worst-case**

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

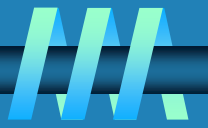
**Both worst-case and average-case efficiency: $\Theta(n \log n)$**

**In-place: yes**

**Stability: no (e.g., 1 1)**



# Priority Queue

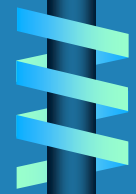


A *priority queue* is the ADT of a set of elements with numerical priorities with the following operations:

- find element with highest priority
- delete element with highest priority
- insert element with assigned priority (see below)

- Heap is a very efficient way for implementing priority queues

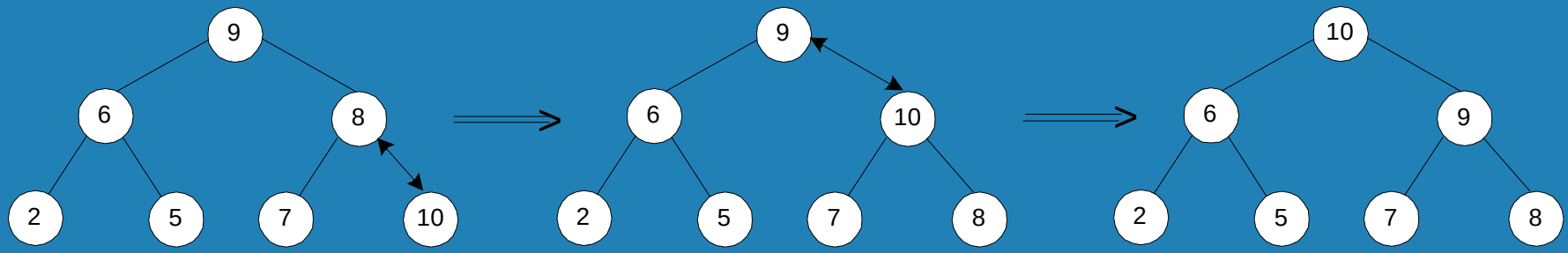
- Two ways to handle priority queue in which highest priority = smallest number



# 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*)**