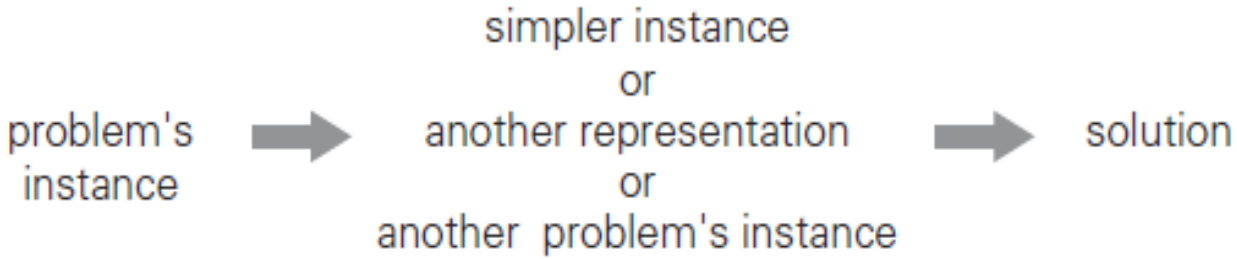
6- TRANSFORM AND CONQUER

6- Transform and Conquer

- Transform-and-conquer work as two-stage procedures. First, in the transformation stage, the problem's instance is modified to be, for one reason or another, more amenable to solution. Then, in the second or conquering stage, it is solved.

- Transformation to a simpler or more convenient instance of the same problem—we call it instance simplification.
- Transformation to a different representation of the same instance—we call it representation change.
- Transformation to an instance of a different problem for which an algorithm is already available—we call it problem reduction.

6- Transform and Conquer



6- Transform and Conquer - Presorting

- Presorting is an old idea in computer science. In fact, interest in sorting algorithms is due, to a significant degree, to the fact that many questions about a list are easier to answer if the list is sorted.

- Obviously, the time efficiency of algorithms that involve sorting may depend on the efficiency of the sorting algorithm being used.

- For the sake of simplicity, we assume throughout this section that lists are implemented as arrays, because some sorting algorithms are easier to implement for the array representation.

6- Transform and Conquer - Presorting

- Checking element uniqueness in an array is an example for this approach. The brute-force algorithm compared pairs of the array's elements until either two equal elements were found or no more pairs were left. Its worst-case efficiency was ???.

6- Transform and Conquer - Presorting

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 $i \leftarrow 0$ **to** $n-2$ **do**
    if $A[i] = A[i+1]$ **return false**
return true

6- Transform and Conquer – SOME EXERCISES

- Consider the problem of finding the distance between the two closest numbers in an array of n numbers. (The distance between two numbers x and y is computed as $|x - y|$.

  - a) Design a presorting-based algorithm for solving this problem and determine its efficiency class.
  - b) Compare the efficiency of this algorithm with that of the brute-force algorithm.

6- Transform and Conquer – SOME EXERCISES

- Consider the problem of finding the smallest and largest elements in an array of n numbers.

  - a) Design a presorting-based algorithm for solving this problem and determine its efficiency class.

  - b) Compare the efficiency of the three algorithms: (i) the brute-force algorithm, (ii) this presorting-based algorithm, and (iii) the divide-and-conquer algorithm .

6- Transform and Conquer – SOME EXERCISES

- Number placement given a list of n distinct integers and a sequence of n boxes with pre-set inequality signs inserted between them, design an algorithm that places the numbers into the boxes to satisfy those inequalities. For example, the numbers 4, 6, 3, 1, 8 can be placed in the five boxes as shown below:

$$\boxed{1} < \boxed{8} > \boxed{3} < \boxed{4} < \boxed{6}$$

6- Transform and Conquer - Gaussian Elimination

- You are certainly familiar with systems of two linear equations in two unknowns:

$$a_{11}x + a_{12}y = b_1$$
$$a_{21}x + a_{22}y = b_2.$$

6- Transform and Conquer - Gaussian Elimination

- Recall that unless the coefficients of one equation are proportional to the coefficients of the other, the system has a unique solution.
- The standard method for finding this solution is to use either equation to express one of the variables as a function of the other and then substitute the result into the other equation, yielding a linear equation whose solution is then used to find the value of the second variable.

6- Transform and Conquer - Gaussian Elimination

- In many applications, we need to solve a system of n equations in n unknowns:

$$
\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\
a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\
&\vdots \\
a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n
\end{aligned}
$$

- where n is a large number. Theoretically, we can solve such a system by generalizing the substitution method for solving systems of two linear equations (what general design technique would such a method be based upon?); however, the resulting algorithm would be extremely cumbersome.

6- Transform and Conquer - Gaussian Elimination

- Fortunately, there is a much more elegant algorithm for solving systems of linear equations called Gaussian elimination.

- The idea of Gaussian elimination is to transform a system of n linear equations in n unknowns to an equivalent system (i.e., a system with the same solution as the original one) with an upper-triangular coefficient matrix, a matrix with all zeros below its main diagonal:

6- Transform and Conquer - Gaussian Elimination

$$
\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\
a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\
&\vdots \\
a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n
\end{aligned}
\quad \Longrightarrow \quad
\begin{aligned}
a'_{11}x_1 + a'_{12}x_2 + \cdots + a'_{1n}x_n &= b'_1 \\
a'_{22}x_2 + \cdots + a'_{2n}x_n &= b'_2 \\
&\vdots \\
a'_{nn}x_n &= b'_n.
\end{aligned}
$$

In matrix notations, we can write this as

$$Ax = b \quad \Longrightarrow \quad A'x = b',$$

where

$$
A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & & & \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix}, \; b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}, \; A' = \begin{bmatrix} a'_{11} & a'_{12} & \dots & a'_{1n} \\ 0 & a'_{22} & \dots & a'_{2n} \\ \vdots & & & \\ 0 & 0 & \dots & a'_{nn} \end{bmatrix}, \; b = \begin{bmatrix} b'_1 \\ b'_2 \\ \vdots \\ b'_n \end{bmatrix}
$$

6- Transform and Conquer - Gaussian Elimination

- How can we get from a system with an arbitrary coefficient matrix A to an equivalent system with an upper-triangular coefficient matrix A'?
- We can do that through a series of the so-called elementary operations.
  - exchanging two equations of the system
  - replacing an equation with its nonzero multiple
  - replacing an equation with a sum or difference of this equation and some multiple of another equation

6- Transform and Conquer - Gaussian Elimination

- Solve the system by Gaussian elimination.

$$
\begin{aligned}
2x_1 - x_2 + x_3 &= 1\\
4x_1 + x_2 - x_3 &= 5\\
x_1 + x_2 + x_3 &= 0.
\end{aligned}
$$

$$
\begin{bmatrix} 2 & -1 & 1 & 1 \\ 4 & 1 & -1 & 5 \\ 1 & 1 & 1 & 0 \end{bmatrix}
\begin{matrix} \\ \text{row } 2 - \frac{4}{2} \text{ row } 1 \\ \text{row } 3 - \frac{1}{2} \text{ row } 1 \end{matrix}
$$

$$
\begin{bmatrix} 2 & -1 & 1 & 1 \\ 0 & 3 & -3 & 3 \\ 0 & \frac{3}{2} & \frac{1}{2} & -\frac{1}{2} \end{bmatrix}
\begin{matrix} \\ \\ \text{row } 3 - \frac{1}{2} \text{ row } 2 \end{matrix}
$$

$$
\begin{bmatrix} 2 & -1 & 1 & 1 \\ 0 & 3 & -3 & 3 \\ 0 & 0 & 2 & -2 \end{bmatrix}
$$

Now we can obtain the solution by back substitutions:

$$x_3 = (-2)/2 = -1, \quad x_2 = (3 - (-3)x_3)/3 = 0, \quad \text{and} \quad x_1 = (1 - x_3 - (-1)x_2)/2 = 1.$$

6- Transform and Conquer - Gaussian Elimination

ALGORITHM *ForwardElimination*$(A[1..n, 1..n], b[1..n])$

```
//Applies Gaussian elimination to matrix A of a system's coefficients,
//augmented with vector b of the system's right-hand side values
//Input: Matrix A[1..n, 1..n] and column-vector b[1..n]
//Output: An equivalent upper-triangular matrix in place of A with the
//corresponding right-hand side values in the (n + 1)st column
for i ← 1 to n do A[i, n + 1] ← b[i]    //augments the matrix
for i ← 1 to n − 1 do
    for j ← i + 1 to n do
        for k ← i to n + 1 do
            A[j, k] ← A[j, k] − A[i, k] ∗ A[j, i] / A[i, i]
```

6- Transform and Conquer - Gaussian Elimination

Solve the following system by Gaussian elimination:

$$x_1 + x_2 + x_3 = 2$$
$$2x_1 + x_2 + x_3 = 3$$
$$x_1 - x_2 + 3x_3 = 8.$$

6- Transform and Conquer - Trees

- Binary Search Tree is a binary tree whose nodes contain elements of a set of orderable items, one element per node, so that all elements in the left subtree are smaller than the element in the subtree's root, and all the elements in the right subtree are greater than it.
- Note that this transformation from a set to a binary search tree is an example of the representation-change technique.

6- Transform and Conquer - Trees

- Computer scientists have expended a lot of effort in trying to find a structure that preserves the good properties of the classical binary search tree—principally, the logarithmic efficiency of the dictionary operations and having the set's elements sorted—but avoids its worst-case degeneracy.
- They have come up with two approaches.

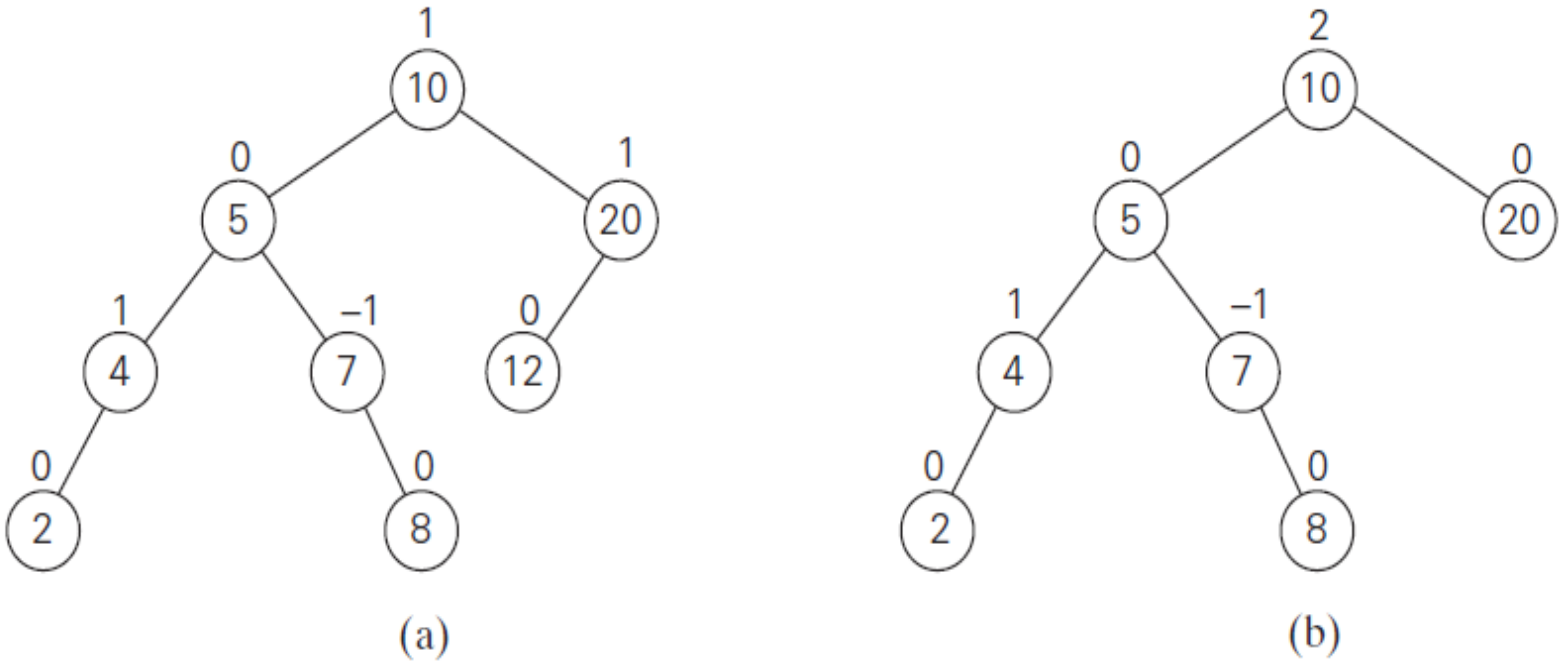
6- Transform and Conquer - Trees

- The first approach is of the instance-simplification variety: an unbalanced binary search tree is transformed into a balanced one. Because of this, such trees are called self-balancing.
- Specific implementations of this idea differ by their definition of balance.
- An AVL tree requires the difference between the heights of the left and right subtrees of every node never exceed 1. A red-black tree tolerates the height of one subtree being twice as large as the other subtree of the same node.

6- Transform and Conquer - Trees

- An AVL tree is a binary search tree in which the balance factor of every node, which is defined as the difference between the heights of the node's left and right subtrees, is either 0 or +1 or −1. (The height of the empty tree is defined as −1.
- Of course, the balance factor can also be computed as the difference between the numbers of levels rather than the height difference of the node's left and right subtrees.)

6- Transform and Conquer - Trees



(a) AVL tree. (b) Binary search tree that is not an AVL tree. The numbers above the nodes indicate the nodes' balance factors.

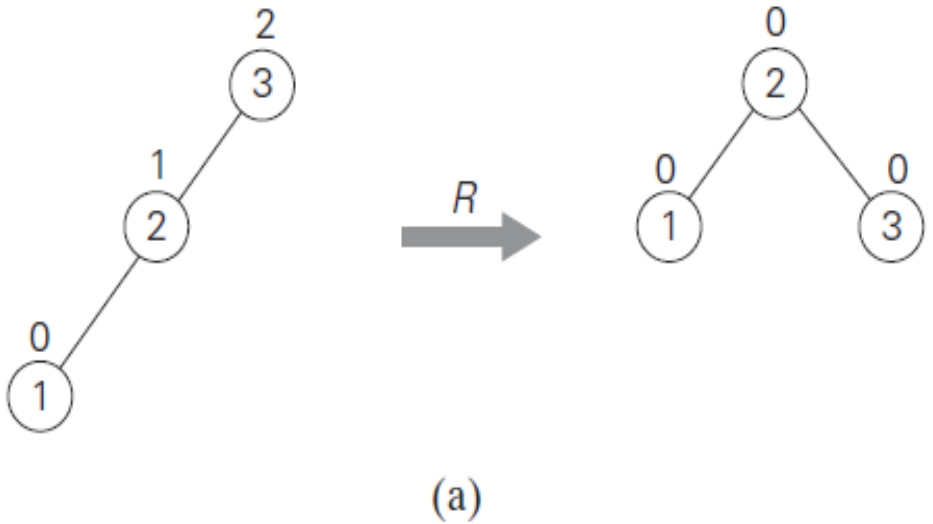
6- Transform and Conquer - Trees

- If an insertion of a new node makes an AVL tree unbalanced, we transform the tree by a rotation.
- A rotation in an AVL tree is a local transformation of its subtree rooted at a node whose balance has become either +2 or −2.
- If there are several such nodes, we rotate the tree rooted at the unbalanced node that is the closest to the newly inserted leaf.
- There are only four types of rotations; in fact, two of them are mirror images of the other two.

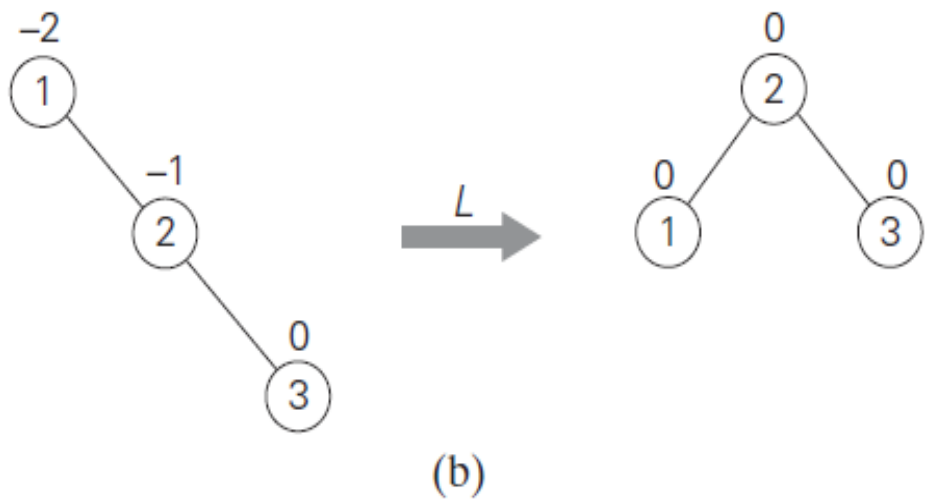
6- Transform and Conquer - Trees

- The first rotation type is called the single right rotation,or R-rotation. Note that this rotation is performed after a new key is inserted into the left subtree of the left child of a tree whose root had the balance of +1 before the insertion.
- The symmetric single left rotation,or L-rotation, is the mirror image of the single R-rotation. It is performed after a new key is inserted into the right subtree of the right child of a tree whose root had the balance of −1 before the insertion.

6- Transform and Conquer - Trees



(a)



(b)

6- Transform and Conquer - Trees

(c)

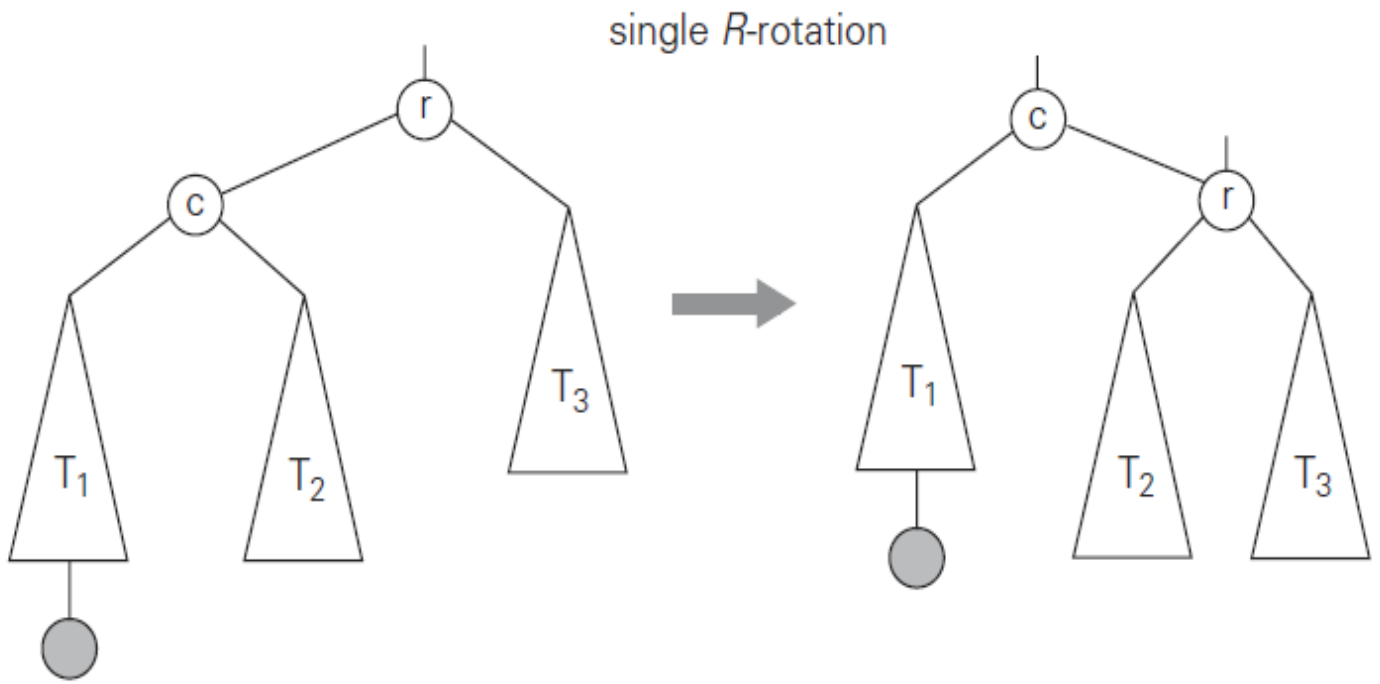
(d)

6- Transform and Conquer - Trees

- The second rotation type is called the double left-right rotation (LR-rotation). It is, in fact, a combination of two rotations: we perform the L-rotation of the left subtree of root r followed by the R-rotation of the new tree rooted at r.

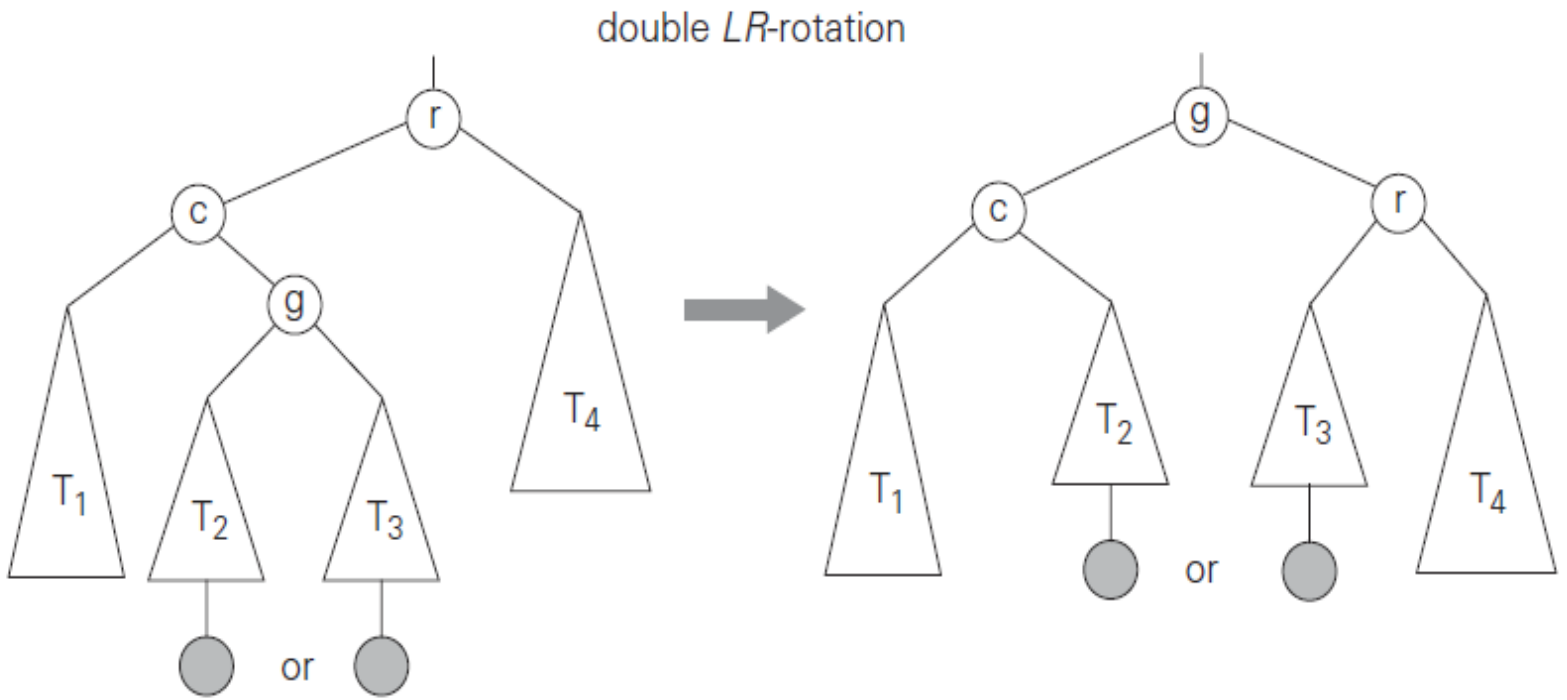
- It is performed after a new key is inserted into the right subtree of the left child of a tree whose root had the balance of +1 before the insertion.

6- Transform and Conquer - Trees



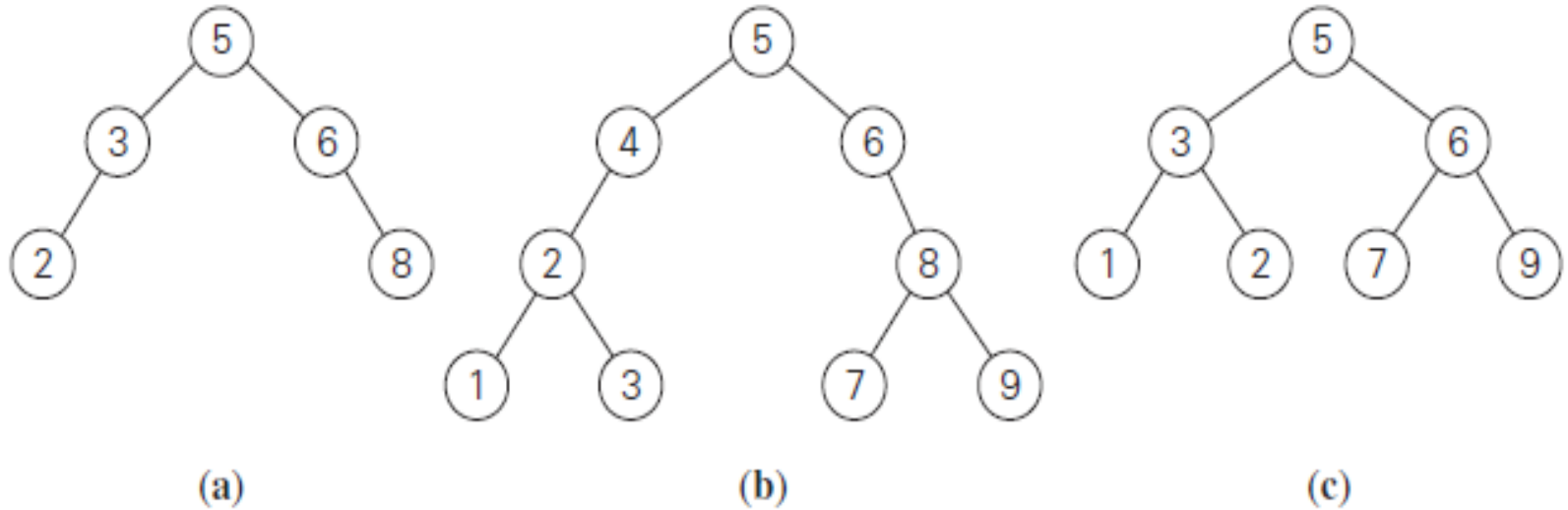
General form of the *R*-rotation in the AVL tree. A shaded node is the last one inserted.

6- Transform and Conquer - Trees



6- Transform and Conquer - Trees

Which of the following binary trees are AVL trees?



(a) (b) (c)

6- Transform and Conquer - Trees

- For n = 1, 2, 3, 4, and 5, draw all the binary trees with n nodes that satisfy the balance requirement of AVL trees.

- For each of the following lists, construct an AVL tree by inserting their elements successively, starting with the empty tree.
  - a) 1, 2, 3, 4, 5, 6
  - b) 6, 5, 4, 3, 2, 1
  - c) 3, 6, 5, 1, 2, 4

6- Transform and Conquer – Heap and Heapsort

- The data structure called the “heap” is definitely not a disordered pile of items as the word’s definition in a standard dictionary might suggest.
- Rather, it is a clever, partially ordered data structure that is especially suitable for implementing priority queues.
- Recall that a priority queue is a multiset of items with an orderable characteristic called an item’s priority, with the following operations:

6- Transform and Conquer – Heap and Heapsort

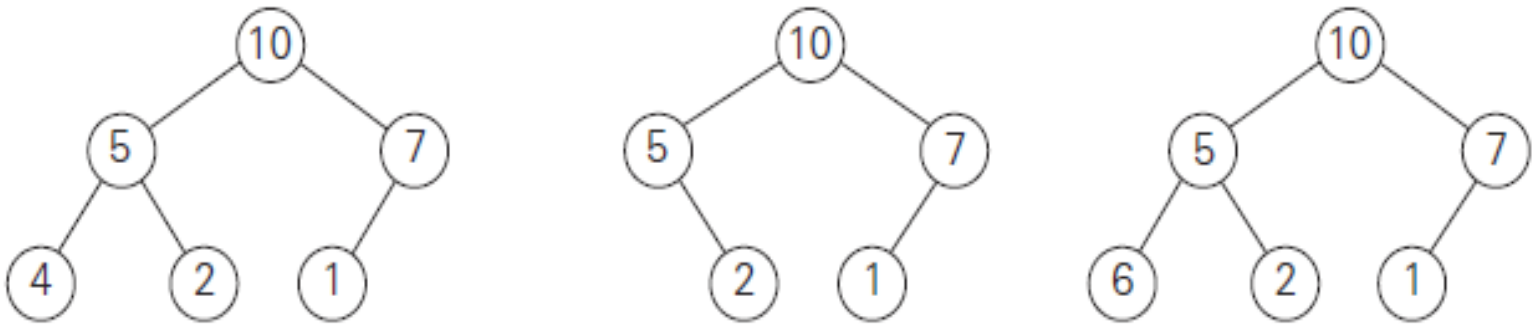


Illustration of the definition of heap: only the leftmost tree is a heap.

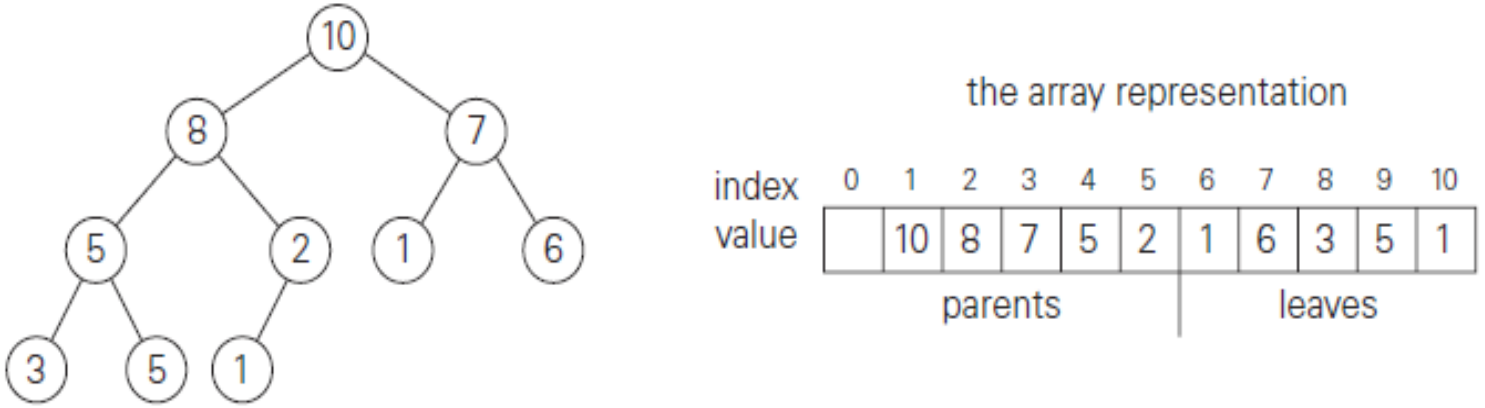
6- Transform and Conquer – Heap and Heapsort

- finding an item with the highest (i.e., largest) priority
- deleting an item with the highest priority
- adding a new item to the multiset

6- Transform and Conquer – Heap and Heapsort

- Notion of the Heap
- DEFINITION A heap can be defined as a binary tree with keys assigned to its nodes, one key per node, provided the following two conditions are met:
- The shape property—the binary tree is essentially complete (or simply complete), i.e., all its levels are full except possibly the last level, where only some rightmost leaves may be missing.
- The parental dominance or heap property—the key in each node is greater than or equal to the keys in its children. (This condition is considered automatically satisfied for all leaves.)

6- Transform and Conquer – Heap and Heapsort

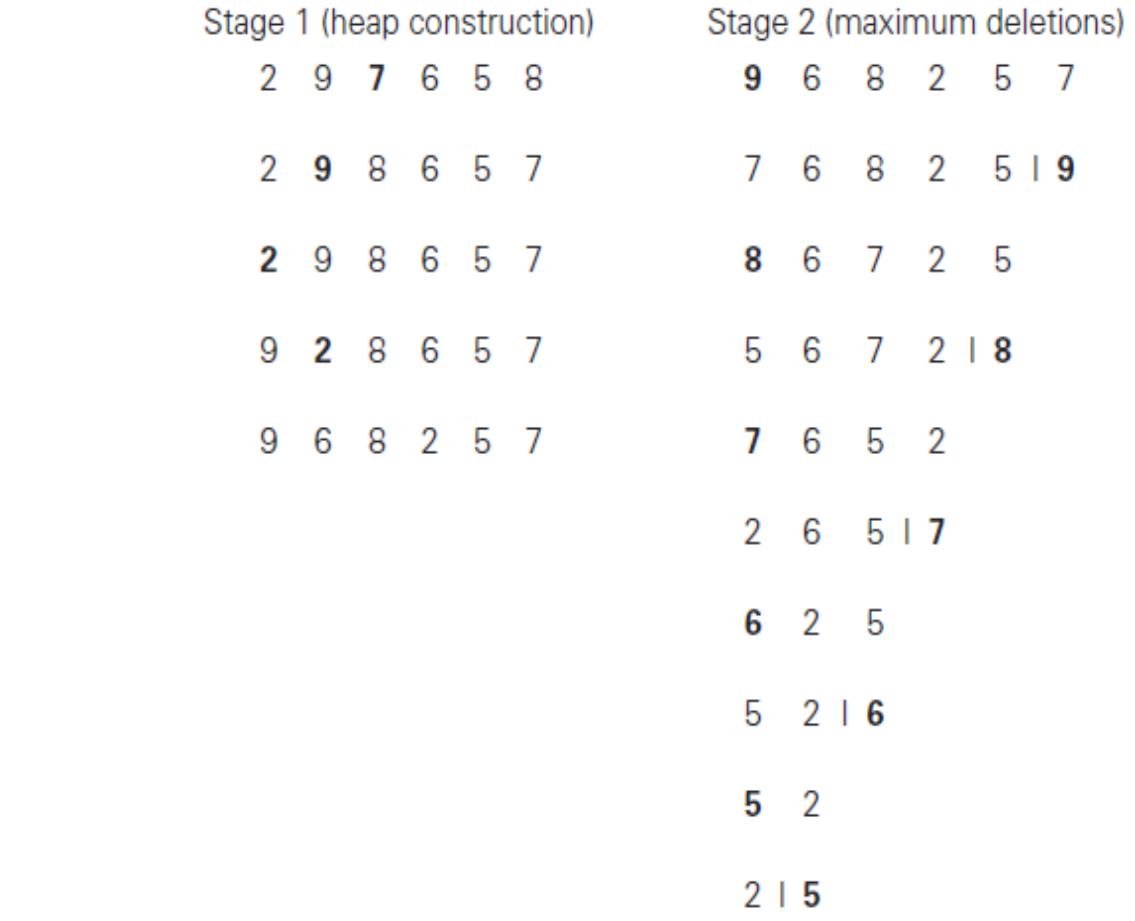


Heap and its array representation.

6- Transform and Conquer – Heapsort

- Now we can describe heapsort—an interesting sorting algorithm discovered by J. W. J. Williams [Wil64]. This is a two-stage algorithm that works as follows.
- Stage 1 (heap construction): Construct a heap for a given array.
- Stage 2 (maximum deletions): Apply the root-deletion operation n − 1 times to the remaining heap.
- As a result, the array elements are eliminated in decreasing order. But since under the array implementation of heaps an element being deleted is placed last, the resulting array will be exactly the original array sorted in increasing order.

6- Transform and Conquer – Heapsort



Sorting the array 2, 9, 7, 6, 5, 8 by heapsort.

6- Transform and Conquer – Heapsort

- Construct a heap for the list 1, 8, 6, 5, 3, 7, 4 by the bottom-up algorithm.
- Construct a heap for the list 1, 8, 6, 5, 3, 7, 4 by successive key insertions (top-down algorithm).
- Is it always true that the bottom-up and top-down algorithms yield the same heap for the same input?

6- Transform and Conquer – Heapsort

- Indicate the time efficiency classes of the three main operations of the priority queue implemented as
  - an unsorted array.
  - a sorted array.
  - a binary search tree.
  - an AVL tree.
  - a heap.