TRANSFORM AND CONQUER

Group of techniques to solve a problem by first transforming the problem into one of:

1. a simpler/more convenient instance of the same problem (**instance simplification**)
2. a different representation of the same instance (**representation change**)
3. a different problem for which an algorithm is already available (**problem reduction**)

INSTANCE SIMPLIFICATION - PRESORTING

Main idea:

- Many problems involving lists are easier when list is sorted:
  - 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.

Efficiency of sorting

- Efficiency of algorithms with pre-sorts 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, quicksort)

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: $\Theta(n\log n) + O(\log n) = \Theta(n\log n)$
- Linear search (without presorting):

Element uniqueness with presorting

- Problem: are all the elements of an array unique?
- Brute force algorithm : Compare all pairs of elements
  Efficiency: $O(n^2)$
- Presorting-based algorithm
  - Stage 1: sort by efficient sorting algorithm (e.g. mergesort)
  - Stage 2: scan array to check pairs of adjacent elements
- Efficiency: $\Theta(n\log n) + O(n) = \Theta(n\log n)$

Computing a mode with presorting

- Definition: a **mode** is a value that occurs most often in a given list of numbers: e.g. for 2, 6, 7, 8, 6, 1, 6, 3, 6, 2 the mode is 6
- Brute force algorithm:
  - Check the frequency of occurrence of each element and select element with highest frequency
  - Efficiency: $O(n^2)$
- Presorting-based algorithm
  - Stage 1: sort by efficient sorting algorithm (e.g. mergesort)
  - Stage 2: scan array counting equal adjacent elements
  - Efficiency: $\Theta(n\log n) + O(n) = \Theta(n\log n)$

INSTANCE SIMPLIFICATION - GAUSSIAN ELIMINATION

Background

- To solve a set of equations:

$$a_{11}x_1 + a_{12}x_2 = b_1$$
$$a_{21}x_1 + a_{22}x_2 = b_2$$

  - Transform the second equation as an equation with 1 variable:

$$a_{11}x_1 + a_{12}x_2 = b_1$$
$$c_{22}x_2 = d_2$$

  - Then second equation can be solved as $x_2 = d_2 / c_{22}$
  - And this value can be substituted back into first:

$$a_{11}x_1 + a_{12}(d_2 / c_{22}) = b_1$$

  - Which can be solved as: $x_1 = [\, b_1 - a_{12}(d_2 / c_{22}) \,] / a_{11}$

- How to transform:

$$a_{11}x_1 + a_{12}x_2 = b_1$$
$$a_{21}x_1 + a_{22}x_2 = b_2$$

into:

$$a_{11}x_1 + a_{12}x_2 = b_1$$
$$c_{22}x_2 = d_2$$

  - Multiply first equation by a_{21}/ a_{11}

$$a_{21}x_1 + (a_{12}a_{21} / a_{11})x_2 = b_1 a_{21}/ a_{11}$$

  - Subtract this equation from the second on both sides:

$$a_{22}x_2 - (a_{12}a_{21} / a_{11})x_2 = b_2 - b_1 a_{21}/ a_{11}$$

  - We get: $c_{22} = a_{22} - (a_{12}a_{21} / a_{11})$

$$d_2 = b_2 - b_1 a_{21}/ a_{11}$$

- Note that the matrix equivalent of

$$a_{11}x_1 + a_{12}x_2 = b_1$$
$$a_{21}x_1 + a_{22}x_2 = b_2$$

is:

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$$

- And the matrix equivalent of

$$a_{11}x_1 + a_{12}x_2 = b_1$$
$$c_{22}x_2 = d_2$$

is:

$$\begin{pmatrix} a_{11} & a_{12} \\ 0 & c_{22} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ d_2 \end{pmatrix}$$

Generalization

- Given: A system of n linear equations in n unknowns with an arbitrary coefficient matrix.

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

- Transform to: An equivalent system of n linear equations in n unknowns with an upper triangular coefficient matrix.

$$\begin{aligned} c_{11}x_1 + c_{12}x_2 + \ldots + c_{1n}x_n &= d_1 \\ c_{22}x_2 + \ldots + c_{2n}x_n &= d_2 \\ &\ldots \\ c_{nn}x_n &= d_n \end{aligned}$$

- Solve the latter by substitutions starting with the last equation and moving up to the first one.
- The transformation is accomplished by a sequence of elementary operations on the system's coefficient matrix (which don't change the system's solution):

for i ←1 to n-1 do
replace each of the subsequent rows (i.e., rows i+1, …, n) by
a difference between that row and an appropriate multiple
of the i-th row to make the new coefficient in the i-th column
of that row 0

Pseudocode

- Stage 1: Reduction to an upper-triangular matrix

```
for i ← 1 to n-1 do          // remove multiple of row i from others below
    for j ← i+1 to n do      // rows j from which row i is removed
        factor ← A[j, i] / A[i, i]
            for k ← i to n+1 do        // columns of row j
                A[j, k] ← A[j, k] - A[i, k] * factor
```

- Stage 2: Back substitutions

```
for j ← n downto 1 do                  //go from bottom row j to top
    t ← 0
    for k ← j +1 to n do               //substitute solved x's into equation
        t ← t + A[j, k] * x[k]
    x[j] ← (A[j, n+1] - t) / A[j, j]   // solve equation for row
```

Efficiency

- $\Theta(n^3) + \Theta(n^2) = \Theta(n^3)$

Example

- Solve

$$\begin{aligned} 2x_1 - 4x_2 + x_3 &= 6 \\ 3x_1 - x_2 + x_3 &= 11 \\ x_1 + x_2 - x_3 &= -3 \end{aligned}$$

$$\begin{array}{cccc} 2 & -4 & 1 & 6 \\ 3 & -1 & 1 & 11 \\ 1 & 1 & -1 & -3 \end{array}$$

- Gaussian Elimination

$$\begin{array}{cccc} & & & \\ 2 & -4 & 1 & 6 \\ 0 & 5 & -1/2 & 2 \\ 0 & 3 & -3/2 & -6 \end{array}$$

row2 – (3/2)*row1:

row3 – (1/2)*row1:

$$\begin{array}{cccc} 2 & -4 & 1 & 6 \\ 0 & 5 & -1/2 & 2 \\ 0 & 0 & -6/5 & -36/5 \end{array}$$

row3–(3/5)*row2 :

- Which represents:

$$\begin{aligned} 2x_1 - 4x_2 + x_3 &= 6 \\ 5x_2 - 1/2\, x_3 &= 2 \\ -6/5 x_3 &= -36/5 \end{aligned}$$

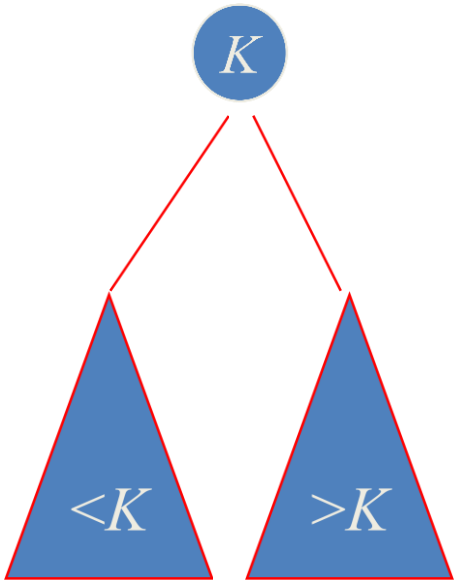
- Backward substitution

$$x_3 = (-36/5) / (-6/5) = 6$$
$$x_2 = (2+(1/2)*6) / 5 = 1$$
$$x_1 = (6 – 6 + 4*1)/2 = 2$$

INSTANCE SIMPLIFICATION - BALANCED SEARCH TREES

Review: Searching with BSTs

- Linear search for a key through an array is O(n)
- Instead arrange keys in a binary tree with the binary search tree property:



- If the tree is balanced, search for key is O(log n)
- **Bonus**: inorder traversal produces sorted list

Review: Operations on BSTs

- 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: $\Theta(\log n)$

Balanced Search Trees

- Attractiveness of binary search tree is marred by the bad (linear) worst-case efficiency. Two ideas to overcome it are:
- Instance simplification: rebalance binary search tree when a new insertion makes the tree "too unbalanced"
  - AVL trees: diff. between heights of L and R subtrees ≤ 1
  - red-black trees: height of 1 subtree ≤ 2* height of the other (see later)
- Representation change: allow more than one key per node of a search tree
  - 2-3 trees
  - 2-3-4 trees
  - B-trees

AVL Trees

- Definitions:
  - The **height** of a tree is the number of edges between the root and the lowest leaf. The height of an empty tree is -1.
  - The **balance factor** of the node of a tree is the difference between the heights of its left and right subtrees.
  - An **AVL tree** is a binary search tree in which the absolute value of the balance factor of any node is at most 1

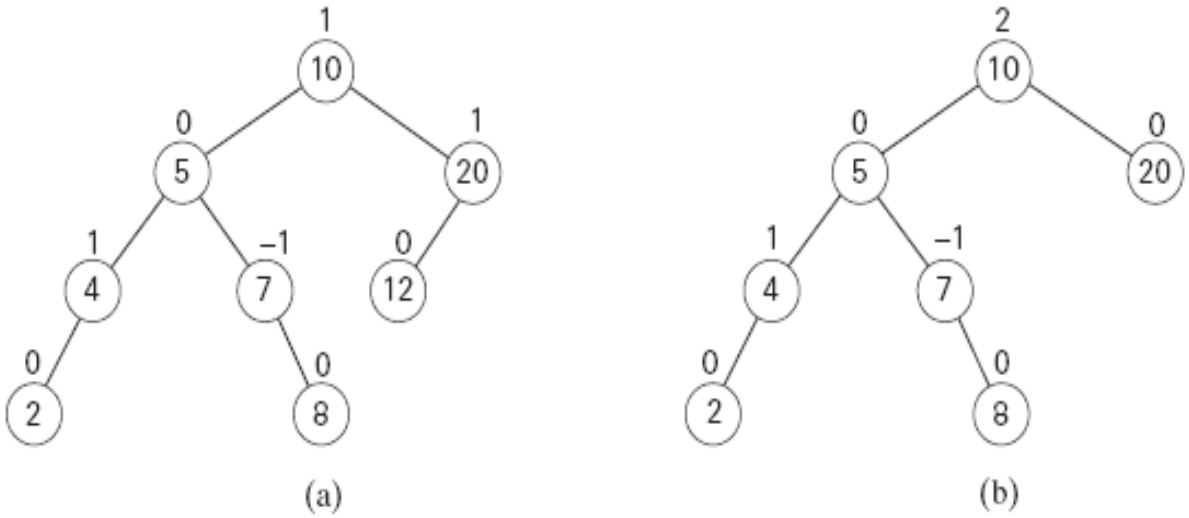
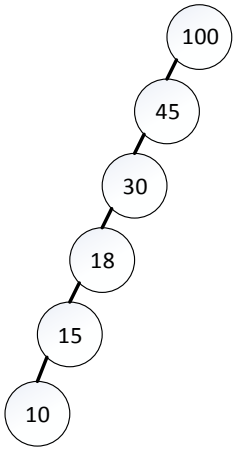


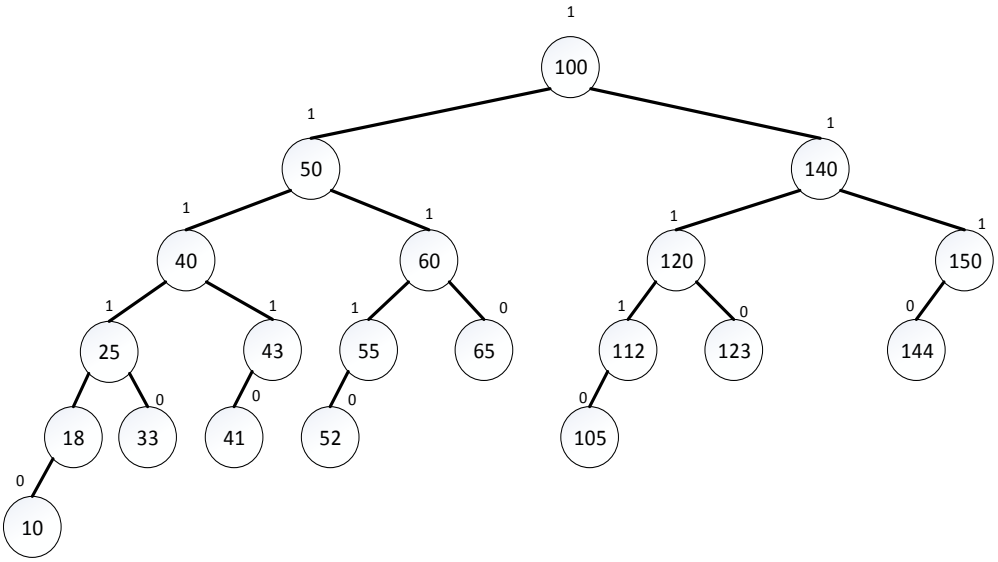
FIGURE 6.2 (a) AVL tree. (b) Binary search tree that is not an AVL tree. The numbers above the nodes indicate the nodes' balance factors.

Degenerate Trees

Degenerate BST



Degenerate AVL Tree
(half of the levels are fully balanced)



- Analysis of Efficiency
  - height $\leq$ 1.4404 $\log_2 (n + 2)$ - 1.3277 i.e. O(log n)
  - 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 (see later)
  - complexity

- 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.)

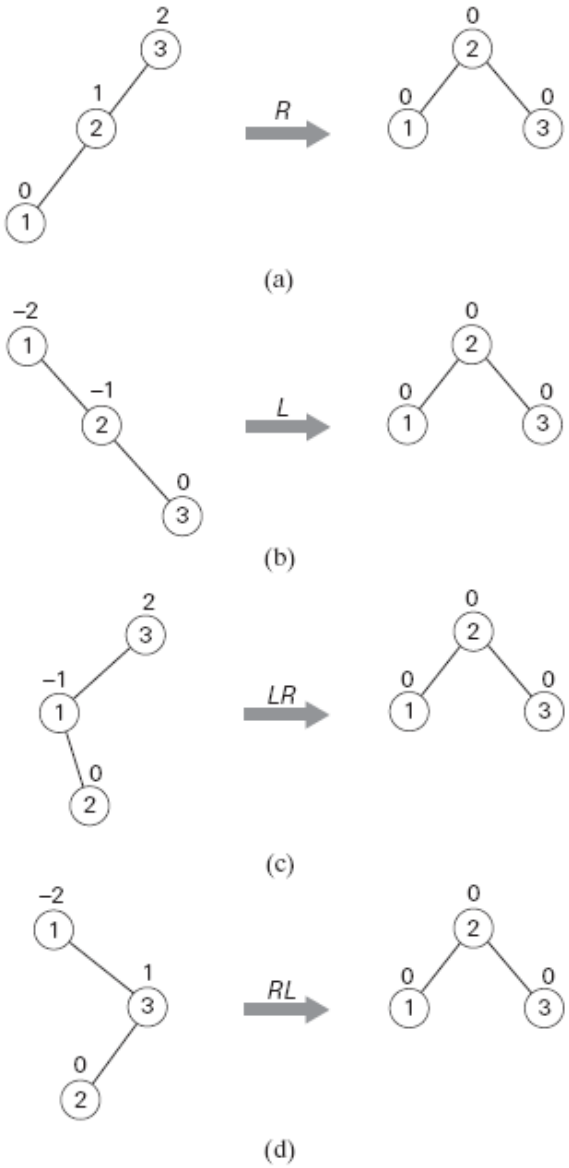


FIGURE 6.3 Four rotation types for AVL trees with three nodes. (a) Single *R*-rotation. (b) Single *L*-rotation. (c) Double *LR*-rotation. (d) Double *RL*-rotation.

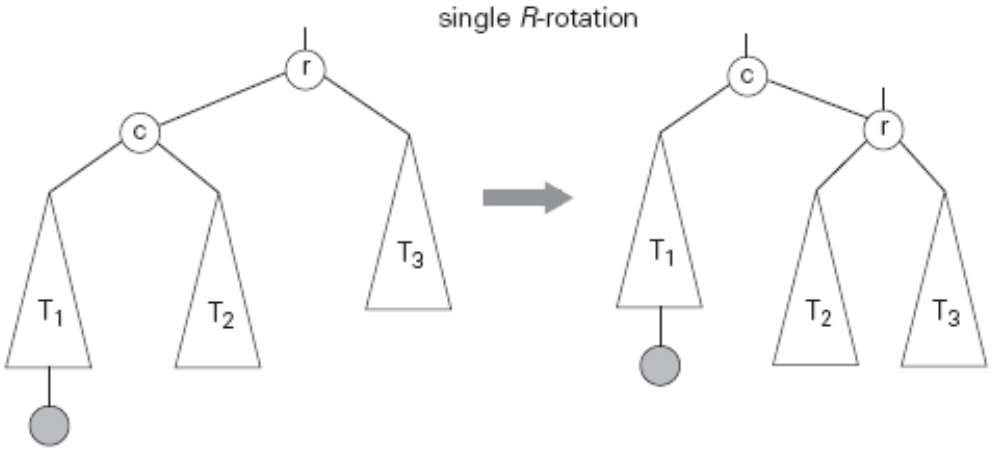


FIGURE 6.4 General form of the *R*-rotation in the AVL tree. A shaded node is the last one inserted.

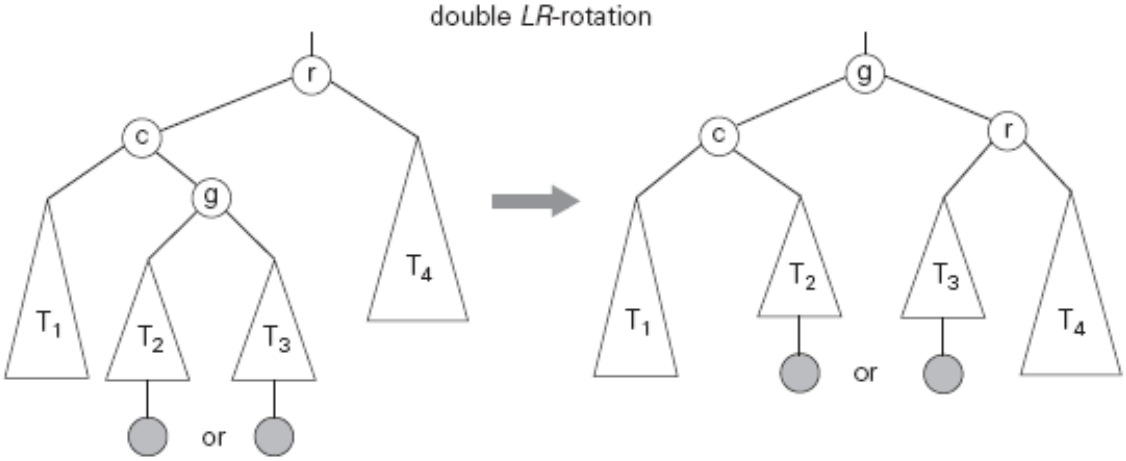


FIGURE 6.5 General form of the double *LR*-rotation in the AVL tree. A shaded node is the last one inserted. It can be either in the left subtree or in the right subtree of the root's grandchild.

- AVL Tree Construction

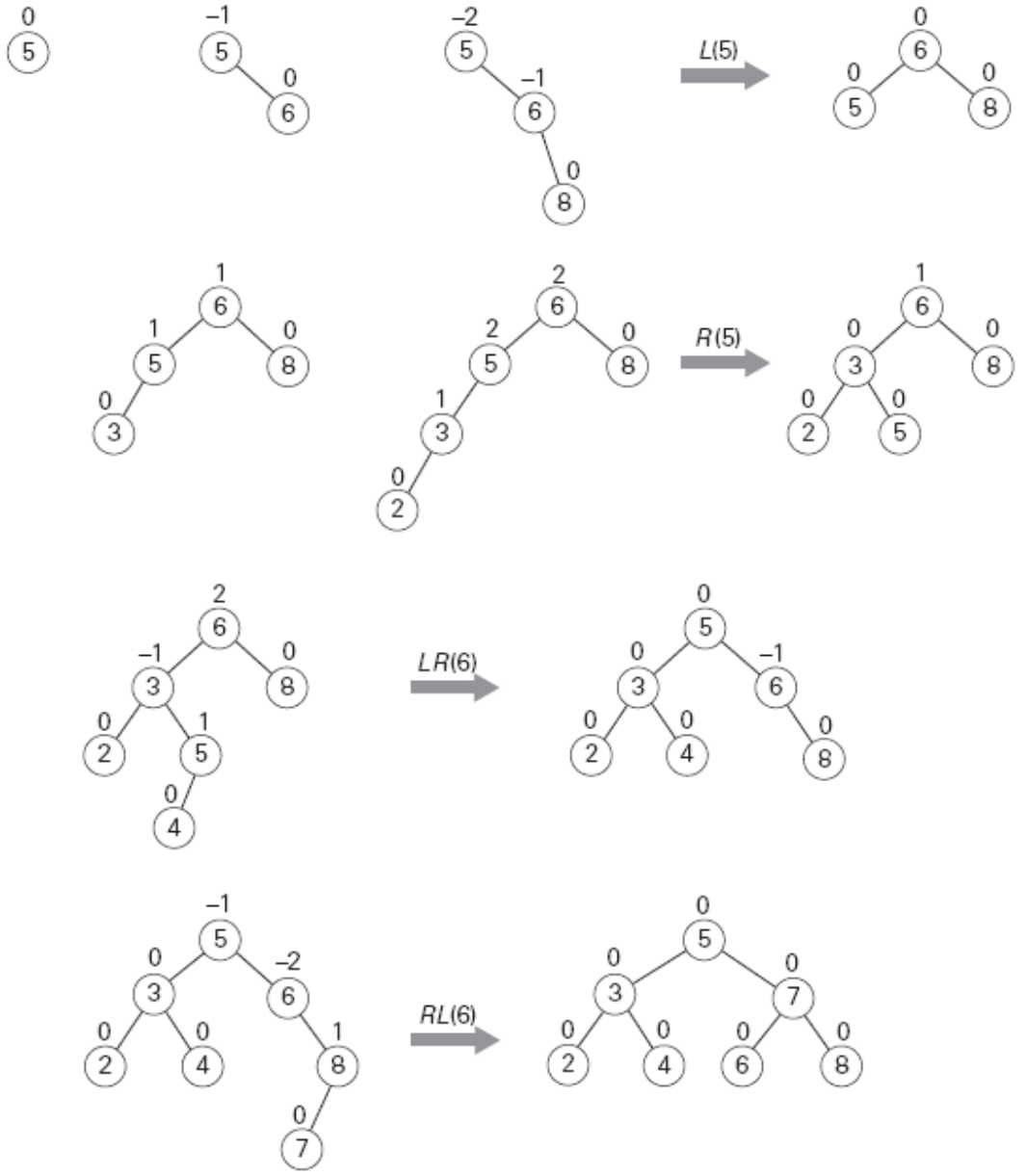
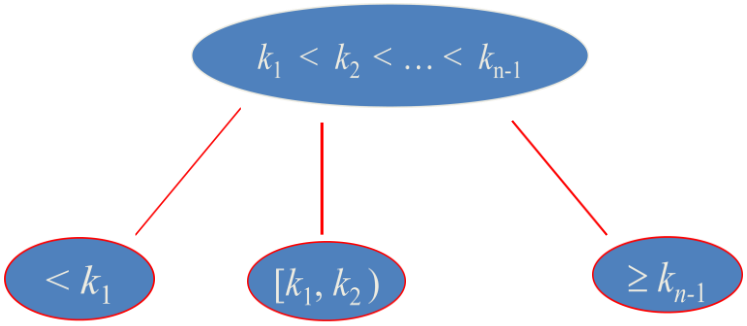


FIGURE 6.6 Construction of an AVL tree for the list 5, 6, 8, 3, 2, 4, 7 by successive insertions. The parenthesized number of a rotation's abbreviation indicates the root of the tree being reorganized.

REPRESENTATION CHANGE - 2-3 TREES

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 Trees

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

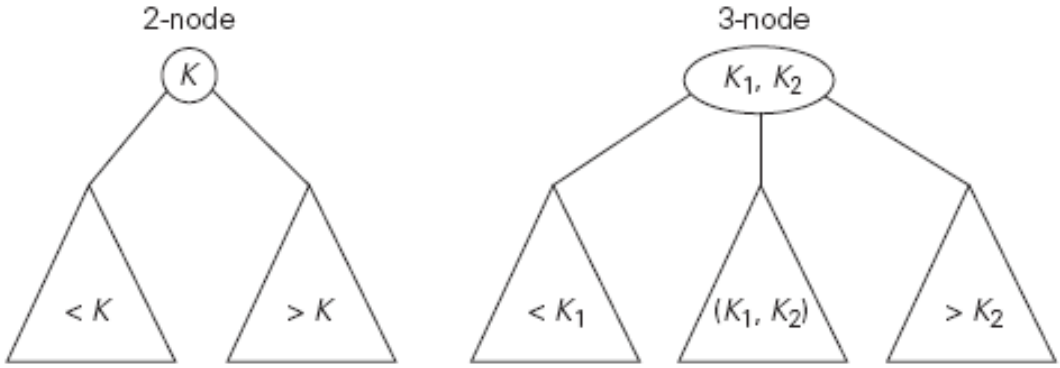
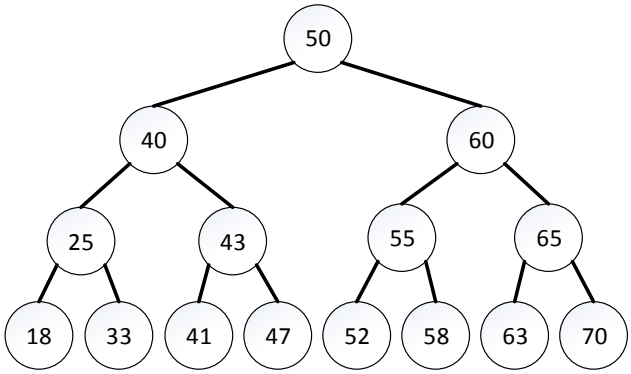


FIGURE 6.7 Two kinds of nodes of a 2-3 tree.

- Extreme 2-3 Trees

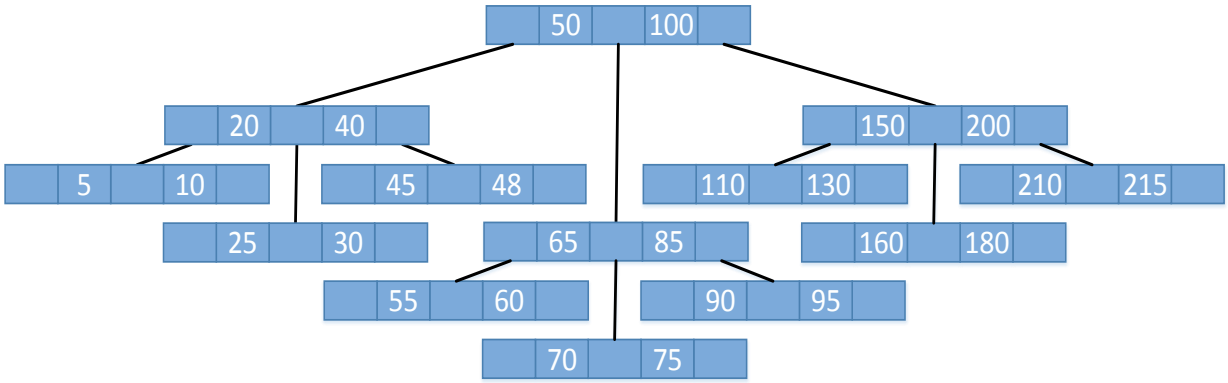
Max height when all nodes are 2-nodes



$$n = \sum_{i=0}^{height} 2^i = \frac{(2^{height+1} - 1)}{2 - 1} = (2^{height+1} - 1)$$

So height = $\log_2(n+1)-1$

Min height when all nodes are 3-nodes



$$n = 2.\sum_{i=0}^{height} 3^i = 2.\frac{(3^{height+1} - 1)}{3 - 1} = (3^{height+1} - 1)$$

So height = $\log_3(n+1)-1$

- Efficiency
  - $\log_3 (n + 1) - 1 \leq$ height $\leq \log_2 (n + 1) - 1$
  - Search, insertion, and deletion are in $\Theta(\log n)$

- Construction:

  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.

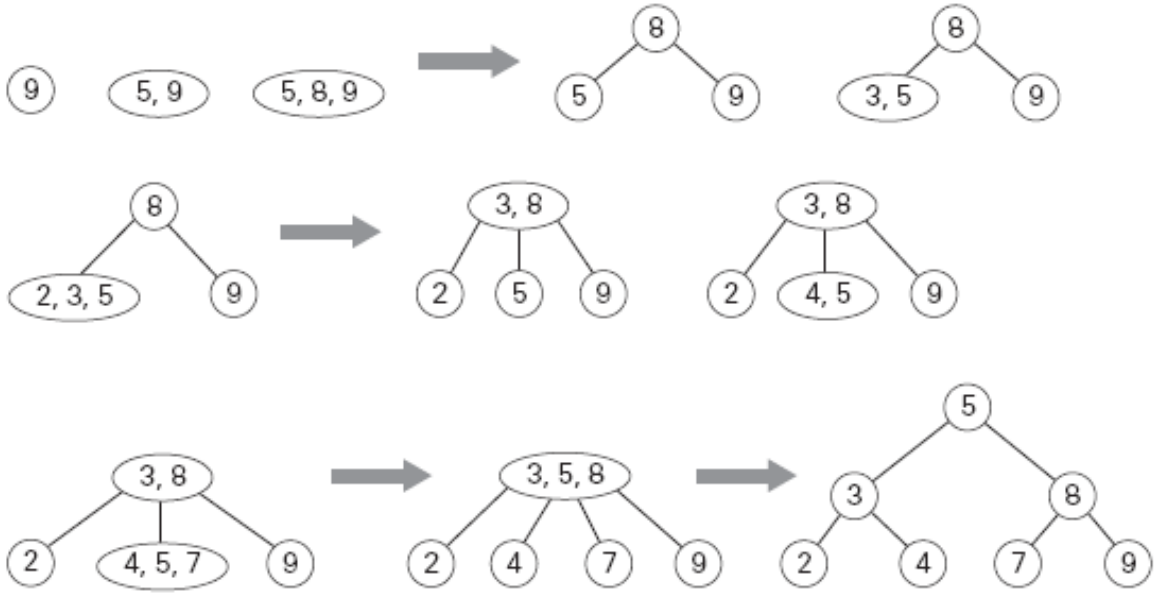


FIGURE 6.8 Construction of a 2-3 tree for the list 9, 5, 8, 3, 2, 4, 7.

- The idea of 2-3 tree can be generalized by allowing more keys per node
  - 2-3-4 trees
  - B-trees
- However, 2-3 tree implementation is complicated. These trees can instead be implemented as red-black BSTs

RED-BLACK TREES

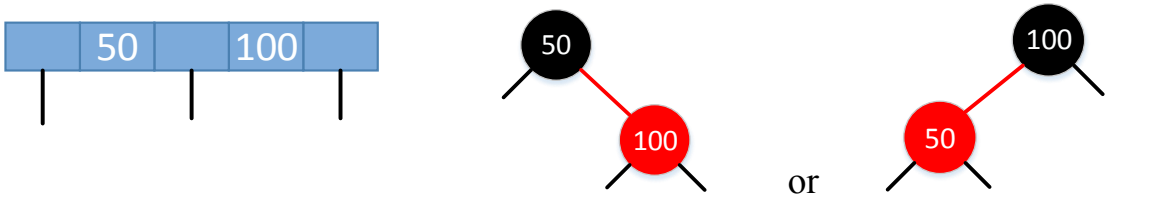
Definition

2-3 Trees can be represented as special BSTs using rules:

- Nodes and edges are either red or black
- A node is the same colour as its upper edge
- On any path from the root to a leaf the number of black edges is the same
- A red node that is not a leaf has 2 black children
- A black node that is not a leaf has either
  - Two black children, or
  - One child of each colour, or
  - A single child which is a red leaf

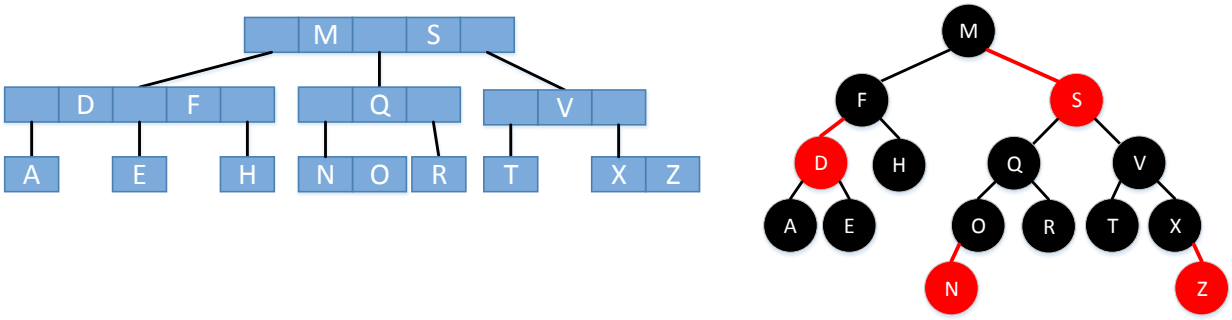
Conversion:

A 3-node: is converted into one of the two red-black trees



Examples

A 2-3 Tree: Is converted into:



Properties

- The height of the red-black tree is at most twice the height of the corresponding 2-3 tree.
- Therefore the logarithmic-time operations on 2-3 trees will be logarithmic time on the red-black implementations of those trees provided that each conversion operation (like splitting) can be done in constant time.

REPRESENTATION CHANGE :
POLYNOMIAL EVALUATION USING HORNER'S RULE

Problem: (Handout 4)

Find the value of polynomial

$p(x) = a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x^1 + a_0$

at a point $x = x_0$

Horner's Rule

$$p(x) = a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x^1 + a_0$$
$$= [(((a_n x + a_{n-1})x + a_{n-2})x + a_{n-3})x + \ldots + a_1] x + a_0$$

Algorithm:

$p \leftarrow a[n]$
for $i \leftarrow n-1$ downto 0 do
  $p \leftarrow p * x + a[i]$
return p

Efficiency

#multiplications = #additions = n

Example

$$p(x) = 2x^4 - x^3 + 3x^2 + x - 5$$
$$= (2x^3 - x^2 + 3x + 1) x - 5$$
$$= ((2x^2 - x + 3) x + 1) x - 5$$
$$= (((2x - 1) x + 3) x + 1) x - 5$$

Arrange coefficients in table:

2	-1	3	1	-5

x=3

2x - 1 = 5
(2x - 1) x + 3 = 18
((2x - 1) x + 3) x + 1) = 55
(((2x - 1) x + 3) x + 1) x – 5 = 160

REPRESENTATION CHANGE : SYNTHETIC DIVISION = POLYNOMIAL DIVISION USING HORNER'S RULE

Problem:

Divide $p(x) = a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x^1 + a_0$ by $(x - x_0)$

Result: $p(x) = (x - x_0)\, q(x) + remainder$

where $q(x) = b_{n-1}x^{n-1} + \ldots + b_1 x^1 + b_0$

Remainder Theorem:

If the polynomial $p(x)$ is divided by $x - x_0$, then the remainder is $p(x_0)$.

Application (substitution of remainder): $p(x) = (x - x_0)\, q(x) + p(x_0)$

Find coefficients b_i's of $q(x)$

$$\begin{aligned} p(x) &= (x - x_0)\,(b_{n-1}x^{n-1} + \ldots + b_1 x^1 + b_0) + p(x_0) \\ &= x\,.(b_{n-1}x^{n-1} + \ldots + b_1 x^1 + b_0) - x_0\,.\,(b_{n-1}x^{n-1} + \ldots + b_1 x^1 + b_0) + p(x_0) \\ &= x\,.(b_{n-1}x^{n-1} + \ldots + b_1 x^1 + b_0) - x_0\,.\,(b_{n-1}x^{n-1} + \ldots + b_1 x^1) - b_0 x_0 + p(x_0) \\ &= a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x^1 + a_0 \end{aligned}$$

So $a_0 = p(x_0) - b_0 x_0$

So $b_0 = (p(x_0) - a_0)/x_0$

Apply Horner's Rule to $p(x_0)$:

$$\begin{aligned} p(x_0) &= a_n x_0^n + a_{n-1} x_0^{n-1} + \ldots + a_1 x_0^1 + a_0 \\ &= [(((a_n x_0 + a_{n-1}) x_0 + a_{n-2}) x_0 + a_{n-3}) x_0 + \ldots + a_1] x_0 + a_0 \end{aligned}$$

Substitute back into formula for b_0:

$$\begin{aligned} b_0 &= (p(x_0) - a_0)/x_0 \\ &= ([[(((a_n x_0 + a_{n-1}) x_0 + a_{n-2}) x_0 + a_{n-3}) x_0 + \ldots + a_1] x_0 + a_0] - a_0)/x_0 \\ &= ([[(((a_n x_0 + a_{n-1}) x_0 + a_{n-2}) x_0 + a_{n-3}) x_0 + \ldots + a_1] x_0)/x_0 \\ &= [[(((a_n x_0 + a_{n-1}) x_0 + a_{n-2}) x_0 + a_{n-3}) x_0 + \ldots + a_1] \end{aligned}$$

This is the one before last step in the calculation of $p(x_0)$ using Horner's rule!

Example:

divide $p(x) = 2x^4 - x^3 + 3x^2 + x - 5$ by $x-3$

The steps to calculate p(3) were: 2 (times 3^0), 5, 18, 55, 160

Result: $2x^4 - x^3 + 3x^2 + x - 5 = (x-3)(2x^3 + 5x^2 + 18x + 55) + 160$

Algorithm

```
p ← a[n]
for i ← n-1 downto 0 do
        b[i] ← p
        p ← p * x + a[i]
remainder ←p
```

PROBLEM REDUCTION - EXPLANATION

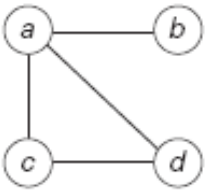
- This variation of transform-and-conquer solves a problem by a transforming it into different problem for which an algorithm is already available.
- To be of practical value, the combined time of the transformation and solving the other problem should be smaller than solving the problem as given by another method.

PROBLEM REDUCTION - LEAST COMMON MULTIPLE

- The least common multiple of two positive integers m and n, denoted lcm(m,n), is the smallest integer that is divisible by both m and n,
- E.g. lcm(25,60) = 300
- Theorem: m.n = gcd(m,n).lcm(m,n) and so $\boldsymbol{lcm(m,n) = \frac{m.n}{\mathbf{gcd}(m,n)}}$
- Algorithm:
  - Calculate gcd(m,n) first using Euclidian Algorithm
  - Apply formula above

PROBLEM REDUCTION - COUNTING PATHS IN A GRAPH

- Problem: given a graph, how many paths of size n does it have?
- Represent graph by adjacency matrix A
- Theorem: the number of paths of length n from vertex i to j is $A^n[i,j]$
- Algorithm: take the power of A n times - answers are in matrix A^n



$$A = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array} \qquad A^2 = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 3 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 2 & 1 \\ 1 & 1 & 1 & 2 \end{bmatrix} \end{array}$$

FIGURE 6.16 A graph, its adjacency matrix A, and its square A^2. The elements of A and A^2 indicate the numbers of paths of lengths 1 and 2, respectively.

PROBLEM REDUCTION - REDUCTION TO GRAPHS

River crossing puzzle:

- Peasant must take cabbage, goat, and wolf across river using one boat which can only take one passenger other than himself. In his absence, the wolf might eat the goat and the goat might eat the cabbage. Is there a way for the peasant to transport all 3 safely to the other side?
- **http://www.transum.org/software/River_Crossing/Level1.asp**

Reduction to graph:

- Draw a ***state-space*** graph: vertices are possible states of the problem, edges indicate how to move from one state to the other. This is the same as a ***state-transition diagram***.

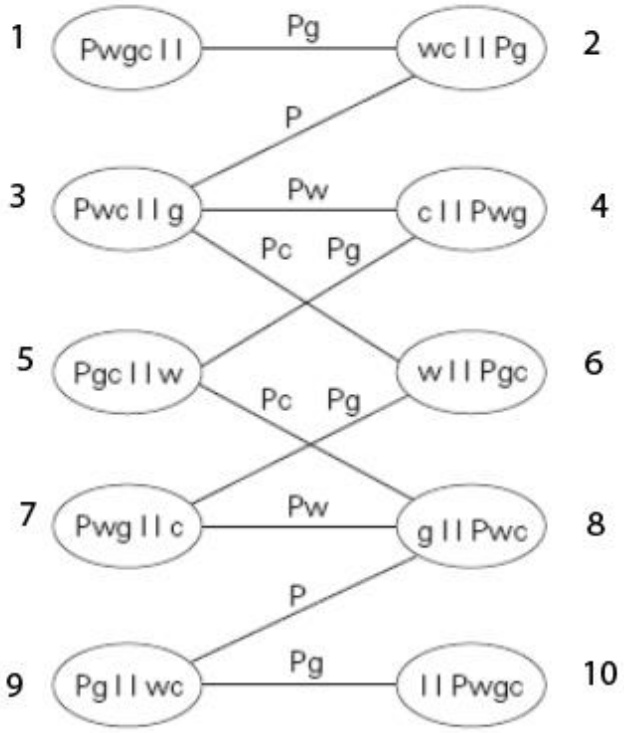


FIGURE 6.18 State-space graph for the peasant, wolf, goat, and cabbage puzzle.

Solution:

- find a minimal path from Pwgc|| to ||Pwgc using BFS

PROBLEM REDUCTION - OPTIMISATION PROBLEMS

- Problem: Find min (or max) of a function

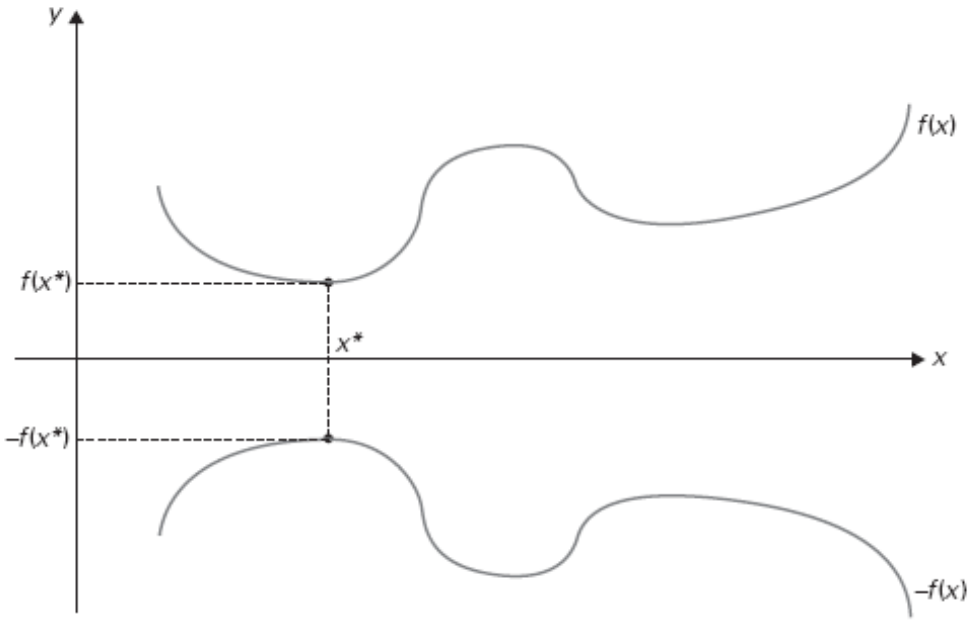


FIGURE 6.17 Relationship between minimization and maximization problems: $\min f(x) = -\max[-f(x)]$.

- How to find either of them in calculus? solve for f'(x)=0
- This is a problem reduction as well: min of a graph reduced to solving an equation