Transform-and-Conquer

CS3230: Design and Analysis of Algorithms

Roger Zimmermann

National University of Singapore

Spring 2017

Chapter 6: Transform-and-Conquer

Topics: What we will cover today

- Presorting
- Gaussian elimination
- Balanced search trees
- Heaps and heapsort
- Decrease-by-a-constant-factor algorithms
- Horner's rule and binary exponentiation
- Problem reduction

The Transform-and-Conquer Design Strategy

- The strategy of transform-and-conquer attacks a problem in two stages. It first modifies the problem instance, and then solves the modified instance.
- There are three major variants of transformation:
  - Instance simplification
  - Representation change
  - Problem reduction

Presorting

- Some problems about a set of data items can be solved easily if the data items are sorted.
- Three examples given to illustrate the usefulness of presorting:
  - Checking element uniqueness
  - Computing a mode
    A mode of a set of values is a value that occurs most frequently.
  - The searching problem

Checking Element Uniqueness

- An algorithm

  ELEMENTUNIQUENESS
  // Input: n keys in $A[0..n-1]$
  sort $A[0..n-1]$
  for i from 0 to $n-2$ do
  if $A[i] = A[i+1]$ then return `false` fi
  od
  return `true`

- With a worst case sorting efficiency of $\Theta(n \log n)$, the worst case time efficiency of the algorithm is

$$C_{sort}(n) + n - 1 \in \Theta(n \log n) + \Theta(n) = \Theta(n \log n).$$

Computing a Mode: Algorithm

MODE
// Input: n keys in $A[0..n-1]$
sort $A[0..n-1]$
$i \leftarrow 0$; $fre \leftarrow 0$
while $i \leq n-1$ do
    $run \leftarrow 1$; $valu \leftarrow A[i]$
    while $i + run \leq n-1$ and $A[i+run] = valu$ do
        $run \leftarrow run + 1$
    od
    if $run > fre$ then $fre \leftarrow run$; $mode \leftarrow valu$ fi
    $i \leftarrow i + run$
od
return $mode$

Computing a Mode: Analysis

- The running time of the algorithm is dominated by the time spent on sorting, e.g., $\Theta(n \log n)$.
- The remainder of the algorithm takes $\Theta(n)$ time.
- Hence, the worst case time efficiency of the algorithm is

$$\Theta(n \log n) + \Theta(n) = \Theta(n \log n).$$

The Searching Problem

- For a single search, a linear search is faster than using pre-sorting.
- However, sorting for searching may be beneficial when multiple searches are performed.
- With binary search, the worst case time efficiency of m searches is $\Theta(m \log n)$ on a sorted array; but the worst case time efficiency of m searches on an unsorted array with sequential search is $\Theta(mn)$.

Solving Systems of Linear Equations

- An $m \times n$ system of linear equations has m linear equations in n variables:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ \vdots \quad &= \quad \vdots \\ a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n &= b_i \\ \vdots \quad &= \quad \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &= b_m \end{aligned}$$

Solving Systems of Linear Equations

- The system can be expressed concisely in matrix notation as

$$Ax = b,$$

where

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}, x = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}, b = \begin{bmatrix} b_1 \\ \vdots \\ b_m \end{bmatrix}.$$

- When $m = n$ and the determinant of the coefficient matrix $|A| \neq 0$, the solution is clearly

$$x = A^{-1}b.$$

- Gaussian elimination with backward substitution finds the solution without computing A^{-1} explicitly.

Gaussian Elimination

- Given a $n \times n$ system of linear equations $Ax = b$, Gaussian elimination transforms the coefficient matrix A to an upper triangular matrix A' such that the system of linear equations $A'x = b'$ gives the solution of the original system.
- An order n square matrix $A = [a_{ij}]$ is upper triangular if $a_{ij} = 0$ for all $1 \leq j < i \leq n$.
- The solution is easily obtained with backward substitution.

Upper Triangular Matrix

- In matrix notations, we can write

$$Ax = b \longrightarrow A'x = b'$$

where

$$A = \begin{bmatrix} a'_{11} & a'_{12} & \cdots & a'_{1n} \\ 0 & a'_{22} & \cdots & a'_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a'_{nn} \end{bmatrix}, x = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}, b = \begin{bmatrix} b'_1 \\ \vdots \\ b'_n \end{bmatrix}.$$

Backward Substitution

- Backward substitution gives

$$
\begin{aligned}
x_n &= \frac{b'_n}{a'_{nn}} \\
x_{n-1} &= \frac{b'_{n-1} - a'_{n-1,n} x_n}{a'_{n-1,n-1}} \\
\vdots &= \vdots \\
x_1 &= \frac{b'_1 - a'_{12} x_2 - \cdots - a'_{1,n} x_n}{a'_{11}}
\end{aligned}
$$

Elementary Operations

- Gaussian elimination uses elementary operations to transform the coefficient matrix A to an upper triangular matrix A'.
- There are three elementary operations:
  - exchanging two equations,
  - replacing an equation $E = 0$ with $cE = 0$ where $c \neq 0$ is some constant,
  - replacing an equation $E = 0$ with $E - cE' = 0$ where c is some constant and E' is another equation.
- Observe that elementary operations preserve the solution.

Gaussian Elimination by Elementary Operations

- Let the i^{th} equation be $E_i = 0$.
- That is,

$$E_i \equiv a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_x - b_i.$$

- If $a_{11} \neq 0$, the coefficients of x_1 in equations $E_i = 0$, $i > 1$ can be made zero by replacing E_i with $E_i - \frac{a_{i1}}{a_{11}}E_1 = 0$.
- a_{11} is called the pivot.
- The 2^{nd} step of Gaussian elimination considers the $(n-1)\times(n-1)$ system of linear equations $E_2 = 0, \cdots, E_n = 0$. (Note that we use the same notation for the modified equations.)

Gaussian Elimination by Elementary Operations

- Assume $a_{22} \neq 0$, the coefficients of x_2 in equations $E_i = 0$, $i > 2$ can be made zero by replacing E_i with $E_i - \frac{a_{i2}}{a_{22}} E_2 = 0$. (Note that we use the same notation for the modified equations and coefficients.)
- This process can be repeated until an upper triangular coefficient matrix is obtained.
- Note that if the given system of linear equations is non-singular, the assumption $a_{ii} \neq 0$ is always true by exchanging two equations when necessary. (Note that we use the same notation for the modified a_{ii}.)

Gaussian Elimination: Example (The Given System)

- Solve the 3×3 system of linear equations:

$$2x_1 - x_2 + x_3 = 1, \quad 4x_1 + x_2 - x_3 = 5, \quad x_1 + x_2 + x_3 = 0.$$

- The coefficient matrix, augmented with the right-hand-side is

$$\begin{bmatrix} 2 & -1 & 1 & 1 \\ 4 & 1 & -1 & 5 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

Gaussian Elimination: Example (Forward Elimination)

- Set $E_2 \leftarrow E_2 - \frac{4}{2}E_1, E_3 \leftarrow E_3 - \frac{1}{2}E_1$, the augmented matrix becomes

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

- Set $E_3 \leftarrow E_3 - \frac{3/2}{3}E_2$,

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

Gaussian Elimination: Example (Backward Elimination)

- x_3:

$$x_3 = \frac{-2}{2} = -1.$$

- x_2:

$$x_2 = \frac{3 - (-3)x_3}{3} = \frac{3 - (-3)(-1)}{3} = 0.$$

- x_1:

$$x_1 = \frac{1 - (1)x_3 - (-1)x_2}{2} = \frac{1 - (-1) + (0)}{2} = 1.$$

Gaussian Elimination: An Algorithm

GAUSSIANELIMINATION
    // Input: coefficient matrix $A[1..n, 1..n], b[1..n]$
    for i in $1..n$ do
        $A[i, n+1] \leftarrow b[i]$ // augment b to A
    od
    for i in $1..n-1$ do
        for j in $i+1..n$ do
            for k in $i..n+1$ do
                $A[j, k] \leftarrow A[j, k] - A[i, k] \times A[j, i]/A[i, i]$
            od
        od
    od

Partial Pivoting

- The i^{th} step of the Gaussian elimination may fail if $a_{ii} = 0$. This is possible even if the matrix A is non-singular.
- Even when $a_{ii} \neq 0$, it may be too small and the scaling factors $a_{ji}/a_{ii}, j = i+1, \cdots, n$, will become too large.
- The above problems can be avoided by partial pivoting.
- Partial pivoting means exchanging the i^{th} equation with the j^{th} equation with the largest $|a_{ji}|$ for $j = i, \cdots, n$. That is,

$$a_{ji} = \max\{a_{ii}, a_{i+1,i}, \cdots, a_{ni}\}.$$

Gaussian Elimination with Partial Pivoting: An Algorithm

// Input: coefficient matrix $A[1..n, 1..n], rhs[1..n]$
for i in $1..n$ do $A[i, n+1] \leftarrow b[i]$ od // augment b to A
for i in $1..n-1$ do
  $p \leftarrow i$ // pivot row
  for $j \leftarrow i+1$ to n do
    if $|A[j,i]| > |A[p,i]|$ then $p \leftarrow j$ fi; od
  for $k \leftarrow i$ to $n+1$ do
    swap($A[i,k]$, $A[p,k]$); od
  for j in $i+1..n$ do
    $temp \leftarrow A[j,i]/A[i,i]$
    for $k \leftarrow i$ to $n+1$ do $A[j,k] \leftarrow A[j,k] - temp \times A[i,k]$ od
  od
od

Gaussian Elimination with Partial Pivoting: An Analysis

- Let $C(n)$ be the number of multiplications and divisions.
- We have

$$C(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \sum_{k=i}^{n+1} 1 = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (n+1-i+1) = \frac{n^3}{3} + n^2 - \frac{4n}{3} \in \Theta n^3.$$

LU Decomposition and Other Applications

- An order n matrix $L = [l_{ij}]$ is lower triangular if $l_{ij} = 0$ for all $1 \leq i < j \leq n$.
- Let $A_0 = A$.
- Gaussian elimination can be seen as

$$A_1 = L_1 A_0, A_2 = L_2 A_1, \cdots, A_{n-1} = L_{n-1} A_{n-2}$$

where A_{n-1} is upper triangular.

LU Decomposition and Other Applications

- It can be verified that

$$L = L_1^{-1} \cdots L_{n-2}^{-1} L_{n-1}^{-1}$$

  is lower triangular.

- Expressing

$$A = LU$$

  as a product of a lower matrix and an upper triangular matrix is known as the LU decomposition of A.

Upper- and Lower-Triangular Matrix: Example

- Recall the upper-triangular matrix of our previous example:

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

- The lower-triangular matrix is:

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1/2 & 1/2 & 1 \end{bmatrix}$$

- Solving $Ax = b$ is equivalent to solving $LUx = b$. Solve $Ly = b$ and $y = Ux$.

Computing the Matrix Inverse

- Order n matrix B (denoted A^{-1}) is the inverse of matrix A if

$$AB = I$$

  where I is the order n identity matrix.
- Let the columns of B be $B_1, \cdots, B_n$ and the columns of I be $I_1, \cdots, I_n$.
- We can do Gaussian elimination n times to the augmented matrices $[A, I_1], \cdots, [A, I_n]$ to find $B_1, \cdots, B_n$.
- Alternatively, we do Gaussian elimination once to find the LU decomposition of $A = LU$.

Computing the Matrix Inverse

- We then use backward substitution to solve for Y_i in

$$LY_i = I_i.$$

and another backward substitution to solve for B_i in

$$UB_i = Y_i.$$

Computing a Determinant

- The determinant of an order n matrix A, can be computed by Laplace expansion recursively such as

$$|A| = \sum_{j=1}^{n} (-1)^{j+1} a_{1j} |A_{1j}|$$

  where $A_{1j}, j = 1, \cdots, n$, is the determinant of the $(n-1) \times (n-1)$ submatrix of A obtained by deleting the first row and the j^{th} column from A.
- But this is a $\Theta(n!)$ algorithm because there are $n!$ terms in $|A|$.

Computing a Determinant

- A better algorithm is to perform Gaussian elimination on A to obtain the upper triangular matrix A'.
- Since A' is triangular, so $|A'| = \prod_{i=1}^{n} a'_{ii}$.
- Furthermore, $|A| = (-1)^{2k+1}|A'|$ where k is the number of performing exchange of rows when partial pivoting is applied in the Gaussian elimination.

Balanced Search Trees

- Searching a skewed binary search tree of n vertices results in $\Theta(n)$ worst case time efficiency.
- To ensure a worst case time efficiency of $\Theta(\log n)$, it is worth trying to make the tree as balanced as possible when doing a deletion or insertion.
- AVL trees are binary search trees that are almost balanced.
- 2-3 trees are search trees that are perfectly balanced.

AVL (Binary Search) Trees

- AVL trees illustrate the instance-simplification variant of the transform and conquer strategy.

The AVL Subtree Height Constraint

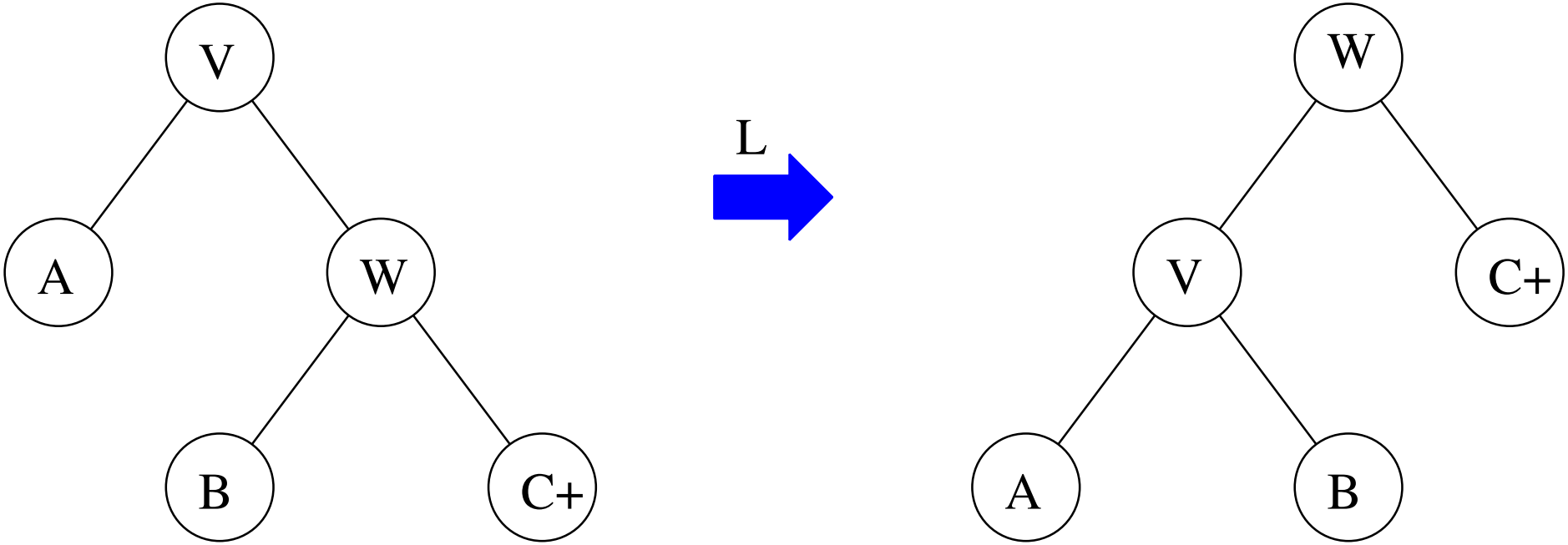
- An AVL tree is a binary search tree with a height constraint to maintain a certain degree of balance.
- Let $L(v)$, $R(v)$ be the left, right subtree of the vertex v and $h(T)$ be the height of the tree T with the convention that $h(\emptyset) = -1$.
- A binary search tree is an AVL tree if for any vertex v of the tree:

$$|h(L(v)) - h(R(v))| \leq 1.$$

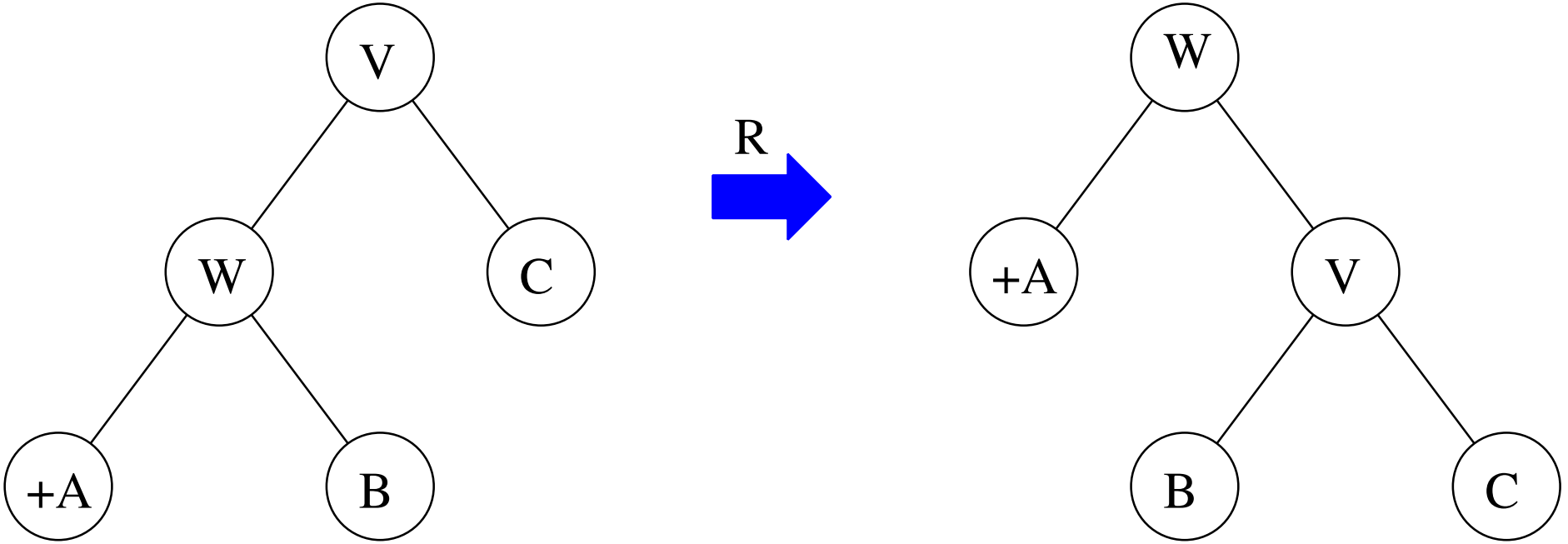
Maintaining the AVL Height Constraint

- The height constraint of an AVL tree may be violated after the insertion of a new vertex.
- Let v be the root of the lowest subtree that fails the height constraint. There are four cases to be considered.
  1. The insertion is to the left child's left subtree.
  2. The insertion is to the left child's right subtree.
  3. The insertion is to the right child's left subtree.
  4. The insertion is to the right child's right subtree.
- It turns out that a single rotation for Cases 1 and 4, and a double rotation for Cases 2 and 3, will restore the height constraint.

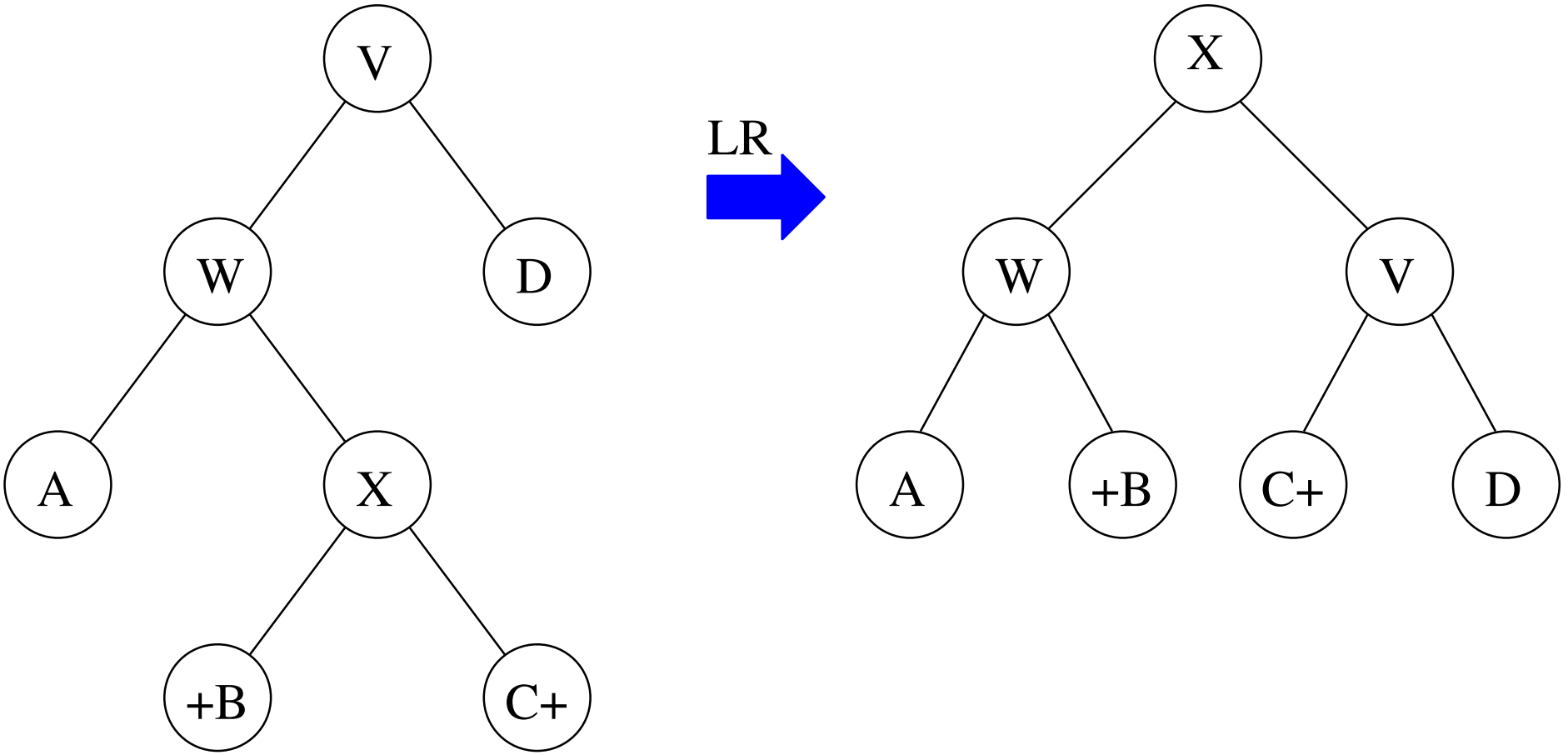
(Left-Left) Single Rotation: Two Vertices are Involved



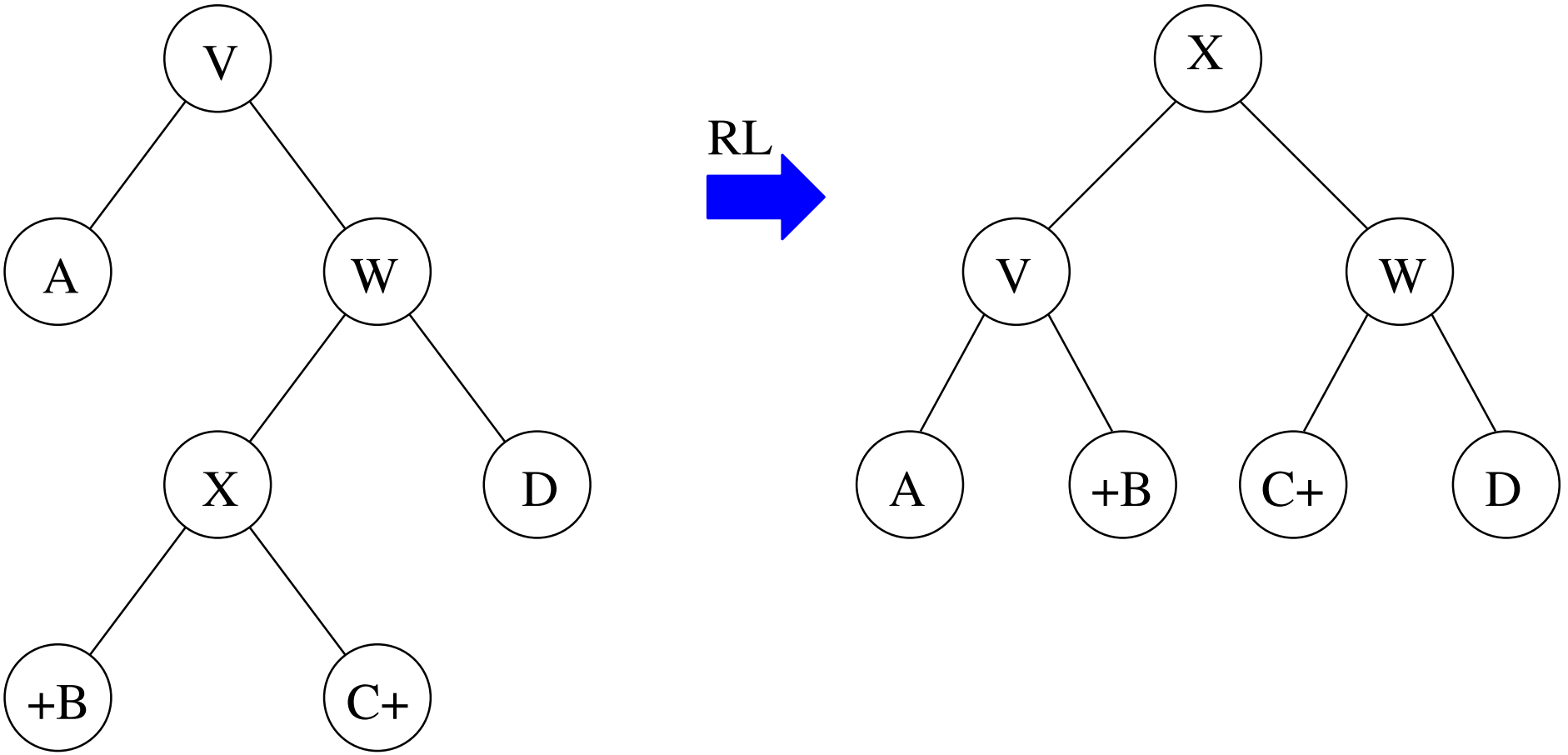
(Right-Right) Single Rotation: Two Vertices are Involved



Left-Right Double Rotation: Three Vertices are Involved



Right-Left Double Rotation: Three Vertices are Involved



The Lower Bound of the Height of an AVL Tree

- Consider an AVL tree with n vertices and height h.
- Since the tree is binary, we have

$$n \leq \sum_{l=0}^{h} 2^l = 2^{h+1} - 1.$$

- That is,

$$\lfloor \log_2 n \rfloor = \lceil \log_2 (n+1) \rceil - 1 \leq h.$$

The Upper Bound of the Height of an AVL Tree

- Let the minimum number of vertices of an AVL tree of height h be m_h.
- We have the recurrence

$$m_h = m_{h-1} + m_{h-2} + 1$$

  with $m_0 = 1$ and $m_1 = 2$.
- Thus

$$m_h + 1 = F_{h+3}, \quad h \geq 0$$

  where F_n is the n-term of the Fibonacci sequence.

Solving for the Upper Bound

- We have

$$m_h + 1 = F_{h+3} = \frac{\phi^{h+3} - \hat{\phi}^{h+3}}{\sqrt{5}} > \frac{\phi^{h+3}}{\sqrt{5}} - 1,$$

  or

$$\frac{\log_2 \sqrt{5}}{\log_2 \phi} + \frac{\log_2(m_h + 2)}{\log_2 \phi} - 3 > h.$$

- That is,

$$1.44 \log_2(m_h + 2) - 1.33 > h.$$

Solving for the Upper Bound

- Consequently, for an AVL tree with n vertices and height h, we have $n \geq m_h$ and hence

$$h < 1.44 \log_2(n+2) - 1.33.$$

Bounds on the Height of an AVL Tree

- For an AVL tree with n vertices and height h, we have

$$\lfloor \log_2 n \rfloor \leq h < 1.44 \log_2 (n+2) - 1.33.$$

2-3 Trees

- 2-3 trees illustrate the representation-change variant of the transform and conquer strategy.
- The vertex of a 2-3 tree holds either one or two keys.
- If a vertex holds one key k, then $l < k < r$ for all keys l of the left subtree and all keys r of the right subtree.
- If a vertex holds two keys $k_1 < k_2$, then $l < k_1 < m < k_2$ for all keys l of the left subtree, for all keys m of the middle subtree, and all keys r of the right subtree.
- All leaves of a 2-3 tree are at the same level.

Inserting a Key Into a 2-3 Trees

- When a key is inserted into a vertex holding one key, the two keys are sorted and stored in the vertex.
- When a key is inserted into a vertex holding two keys, the tree keys are sorted. The middle key is then deleted and inserted into the parent vertex, if there is one; otherwise, a new root is created to hold the middle key.

Bounds on the Height of 2-3 Trees

- For a 2-3 tree with n vertices and height h, we have

$$\sum_{l=0}^{h} 1 \times 2^l \leq n \leq \sum_{l=0}^{h} 2 \times 3^l, \text{ or } 2^{h+1} - 1 \leq n \leq 3^{h+1} - 1.$$

- Consequently,

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

- This ensures that the worst case time efficiency for searching, insertion, and deletion are in $\Theta(\log n)$.

Heaps and Heapsort

- A heap is a data structure that implements priority queues efficiently.
- The queue discipline of priority queues is:
  - enqueue and item according to its priority;
  - dequeue the item with the highest priority.
- A heap can be thought of as a partially sorted (or sorted by height) binary tree stored in an array.

The Heap Notion

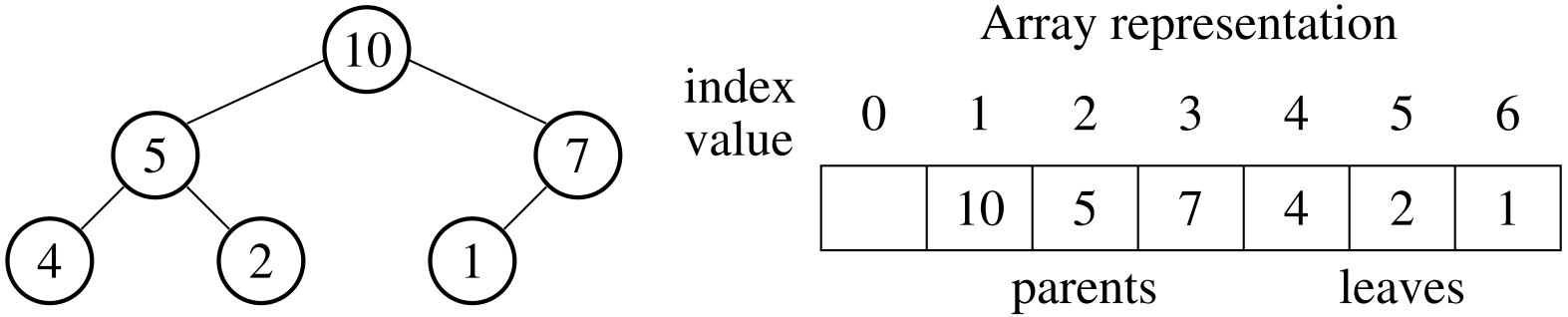
- A heap is a binary tree of keys that is essentially complete (the shape requirement) and parents dominate children (the parental dominance requirement).
- A binary tree is essentially complete if every level is full except perhaps the bottom level, where only some rightmost leaves may be absent.
- Parental dominance means the key of a parent is greater than or equal (or less than or equal) to the keys of its children.

Heap Properties

- There is exactly one essentially complete binary tree with n vertices. Its height is $\lfloor \log_2 n \rfloor$.
- The root of a heap contains the largest key.
- All nodes are either greater than or equal to (max heap or max root) or less than or equal to (min heap or min root) each of its children, according to a comparison predicate defined for the heap.
- Any vertex with all its descendents is again a heap.

Heap as An Array

- An n-element heap can be implemented as an array with indices $1..n$ by storing its elements top-down, left-right.
- If array $A[1..n]$ is a heap, then $A[1]$ is the root, the children of $A[i]$ are $A[2i]$ and $A[2i+1]$, the parent of $A[i]$ is $A[\lfloor i/2 \rfloor]$.
- Ex.:



Heapification

- An arbitrary array $A[1..n]$ can be transformed into a heap in a bottom-up manner called heapification.
- To heapify the array $A[1..n]$, we simply start from the last parental vertex and find the right place among its descendents, once the descendant is found, their keys are swapped.
- We then proceed to the next last parental vertex and repeat the procedure until the root has been similarly processed.

- Q: Can a heap contain duplicate values?

A Bottom-Up Heapification Algorithm

HEAPIFICATION

// Make an arbitrary array $A[1..n]$ into a heap
for $i \leftarrow n/2$ downto 1 do
    $p \leftarrow i$; $pv \leftarrow A[p]$; $heap \leftarrow$ `false`
    while not $heap$ and $2 \times p \leq n$ do
        $c \leftarrow 2 \times p$
        if $c < n$ and $A[c] < A[c+1]$ then $c \leftarrow c+1$ fi
        if $pv \geq A[c]$ then $heap \leftarrow$ `true`
            else $A[p] \leftarrow A[c]$; $p \leftarrow c$ fi
    od
    $A[p] \leftarrow pv$
od

Analysis of the Bottom-Up Heapification Algorithm

- In the while loop, there are two comparisons (which of the two children has the larger key and if the parent dominates the children).
- In the worst case, for the root of every subtree, all levels below it have to be examined.

Worst Case Time Efficiency

- Thus the worst case comparison count for a heap with n elements and height h is

$$\begin{aligned} C(n) &= \sum_{l=0}^{h-1} \sum \text{number of parents at level } l \; 2(h-l) \\ &\leq \sum_{l=o}^{h-1} \sum_{i=1}^{2^l} 2(h-l) \\ &= \sum_{l=0}^{h-1} 2(h-l)2^l \end{aligned}$$

Worst Case Time Efficiency (Cont'd)

- We have

$$\sum_{l=0}^{h-1} 2h2^l = 2h(2^h - 1) = h2^{h+1} - 2h,$$

- and

$$\sum_{l=0}^{h-1} 2l2^l = h2^{h+1} - 2^{h+2} + 4.$$

- That is,

$$C(n) \leq 42^h - 2h - 4 \leq 4n.$$

Heap Insertion

- Let array $A[1..n]$ be a heap.
- An item with key K can be inserted into the heap by storing it as $A[n+1]$ and then sift it up as high as is necessary.
- Observe that an n-item heap can be constructed by successively inserting the n items into an empty heap.

A Heap Insertion Algorithm

HEAPINSERT

// Insert key K into heap $A[1..n]$

$i \leftarrow n + 1$

while $i > 1$ and $K > A[i/2]$ do

$A[i] \leftarrow A[i/2]$; $i \leftarrow i/2$

od

$A[i] \leftarrow K$

Heap Deletion

- To delete the root $A[1]$ from the heap $A[1..n]$, we do the following:
  1. Exchange $A[1]$ and $A[n]$;
  2. Shrink the array to become $A[1..n-1]$.
  3. Sift down $A[1]$ to the right level.
- The cost of the sift operation is determined by the height of the heap binary tree.
- Thus the worst case time efficiency of deletion is $O(\log n)$.

A Heap Deletion Algorithm

HEAPDELETE // Delete $A[1]$ from the heap $A[1..n]$

$A[1] \leftarrow A[n]$; $n \leftarrow n - 1$

$p \leftarrow 1$; $pv \leftarrow A[p]$; $heap <- \texttt{false}$

while not $heap$ and $2 \times p \leq n$ do

  $c \leftarrow 2 \times p$

  if $c < n$ and $A[c] < A[c+1]$ then $c \leftarrow c + 1$ fi

  if $pv \geq A[c]$ then

    $heap \leftarrow \texttt{true}$

  else

    $A[p] \leftarrow A[c]$; $p \leftarrow c$

  fi

od

$A[p] \leftarrow pv$

Heapsort

- Heap deletion and heapification together give a $\Theta(n \log n)$ sorting algorithm.
- Heapsort first heapifies an arbitrary array into a heap and then repeats the following until the heap size is 1.
  1. Exchange the root with the last element.
  2. Reduce the heap size by 1.
  3. Heapify the remaining array into a heap.

Heapsort Analysis

- Let $C(n)$ be the worst case key comparisons count after heapification.
- We have (recall an i-element heap tree has height $\lfloor \log_2 i \rfloor$):

$$\begin{aligned} C(n) &\leq \sum_{i=2}^{n-1} 2\lfloor \log_2 i \rfloor \\ &\leq 2\sum_{i=2}^{n-1} \log_2 i \\ &\leq 2(n-1)\log_2(n-1) \\ &\leq 2n \log_2 n \end{aligned}$$

Horner's Rule

- Consider a degree n univariate polynomial

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

- Horner's rule evaluates $p(x)$ efficiently with the changed representation

$$p(x) = (\cdots (a_n x + a_{n-1})x + \cdots)x + a_0.$$

- Horner's rule requires n multiplications and n additions.

An Algorithm for Horner's Rule

HORNER

// $p[0..n]$ are the $n+1$ coefficients
// Find $v = p(x)$
$v \leftarrow p[n]$
for $i \leftarrow n-1$ downto 0 do
    $v \leftarrow v \times x + p[i]$
od

An Algorithm for the Remainder Theorem

REMAINDER

// $p[0..n]$ are the $n+1$ coefficients
// $p(x) = (x-y)q(x) + p(y)$
// $q[0..n-1]$ gives the quotient $p(x)/(x-y)$
$q[n-1] \leftarrow p[n]$
for $i \leftarrow n-1$ downto 1 do
    $q[i-1] \leftarrow q[i] \times y + p[i]$
od

Horner's Rule and Exponentiation

- The power a^n can be evaluated efficiently if
  1. The exponent n is represented in binary $n = b_l \cdots b_0$.
  2. Evaluate

$$n = \sum_{i=0}^{l} b_i 2^i$$

  using Horner's rule as the intermediate exponents of the value a.
- Since $l = \lfloor \log_2 n \rfloor$, so the time efficiency is $\Theta(\log n)$.

Exponentiation by Horner's Rule

EXPHORNER

// Array $b[0..l]$ gives the
// binary representation of n

$p \leftarrow a$
for i from $l-1$ downto 0 do
    $p \leftarrow p \times p$
    if $b[i] = 1$ then $p \leftarrow p \times a$ fi
od

Problem Reduction

- Problem reduction refers to the strategy of transforming a problem into another problem, solving the latter problem, and transforming the solution to become a solution of the former problem.
- Examples of this strategy include: computing the least common multiple, counting paths in a graph, reduction of optimization problems, linear programming, reduction to graph problems.

LCM from GCD

- Given two positive integers m, n it is well-known that

$$mn = \text{gcd}(m, n)\text{lcm}(m, n).$$

- Thus $lcm(m, n)$ can be computed as

$$\text{lcm}(m, n) = \frac{mn}{\text{gcd}(m, n)}$$

with any efficient algorithm to compute GCD.

Counting Paths in a Graph

- If the entry $A[i, j]$ of a matrix A gives the number of length 1 paths from vertex i to vertex j, then the entry $A^n[i, j]$ of the matrix A^n gives the number of length n paths from vertex i to vertex j.
- In other words, the problem of path counting has become the problem of matrix multiplication.

Reduction of Optimization Problems

- A minimization problem can be solved as a maximization problem by exploiting

$$\min f(x) = -\max(-f(x)).$$

- Similarly, a maximization problem can be solved as a minimization problem by exploiting

$$\max f(x) = -\min(-f(x)).$$

Linear Programming

- Linear programming concerns the optimization (minimization or maximization) of a linear function of several variables subject to linear constraints in the form of linear inequalities. That is, minimize or maximize

$$c_1 x_1 + \cdots + c_n x_n$$

subject to

$$a_{i1} x_1 + \cdots + a_{in} \leq b_i$$

for $i = 1, \cdots, m$.

The Continuous and the Discrete Knapsack Problems

- Given a knapsack of capacity W and n items of weights $w_1, \cdots, w_n$ and values $v_1, \cdots, v_n$, what is the most valuable subset of the items that fits in the knapsack?
- The continuous version: $\sum_{i=1}^{n} v_i x_i$ subject to $\sum_{i=1}^{n} w_i x_i \leq W$ and $0 \leq x_i \leq 1, i = 1, \cdots, n$.
- The discrete version: $\sum_{i=1}^{n} v_i x_i$ subject to $\sum_{i=1}^{n} w_i x_i \leq W$ and $x_i \in \{0, 1\}, i = 1, \cdots, n$.
- Clearly, the continuous version can be solved by linear programming by there is also a very simple solution.

Reduction to Graph Problems

- A problem can be visualized and the techniques of graphs can be utilized if it can be transformed to a graph problem. The general transformation strategy is as follows.
- Encode the possible state of the problem as vertices of a graph.
- There is an edge between two vertices if a transition is allowed between them.
- Identify the initial states and the goal states.

The Farmer, Cabbage, Goat, Wolf River Crossing Problem

- Possible states:

  pwgc/, wc/pg, pwc/g, c/pwg, pgc/w, w/pgc, pwg/c, g/pwc, pg/wc, /pwgc

- Initial state: pwgc/, goal state /pwgc.
- Edges:

  pg={ pwgc/, wc/pg }, p={ wc/pg, pwc/g }, pw={ pwc/g, c/pwg }, pc={ pwc/g, w/pgc }, etc.