Overview and Introduction

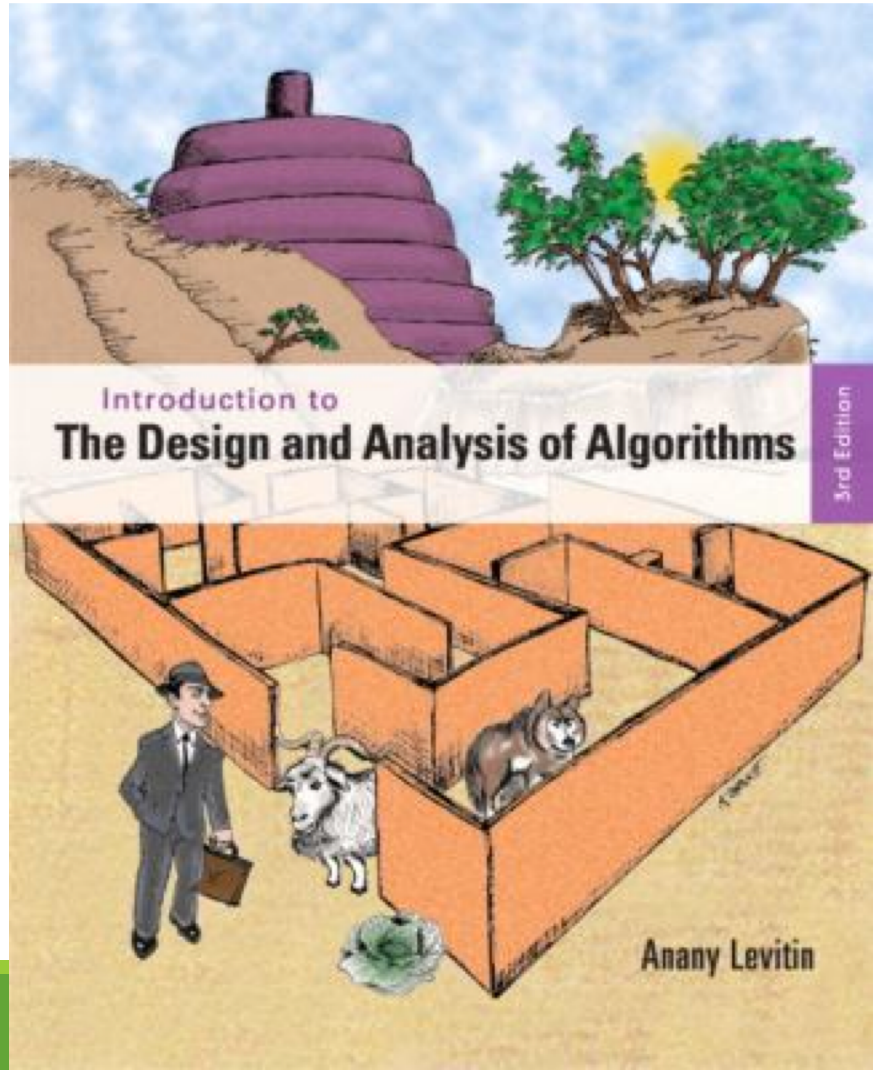
CS 3623-01

Design & Analysis of Algorithm

Instructor: Dr. Qingguo Wang

College of Computing & Technology

August 19, 2019



About me

Ph.D. in Computer Science, University of Missouri, 2011

Postdoctoral Fellow, Biomedical Informatics, Vanderbilt University, 2011-2014

VANDERBILT UNIVERSITY

Computational Engineer III, Memorial Sloan Kettering Cancer Center, 2014-2016

Memorial Sloan Kettering Cancer Center

Associate Professor of Data Science, Lipscomb University, 2016-present

LIPSCOMB UNIVERSITY 18 91

My contact

Email: qwang@lipscomb.edu

Office: Swang 120

Phone: (615) 966-5814

Office Hours: M-F 8:30-10:30am

Introduce you to the class

- Your name
- Major
- Interest

Prerequisites

Two courses with grades of ‘C’ or higher (see Syllabus in Canvas)

- CS 2233 - Data Structures and Algorithms
- MA 2903 - Logic, Proof & Math Modeling

Textbook

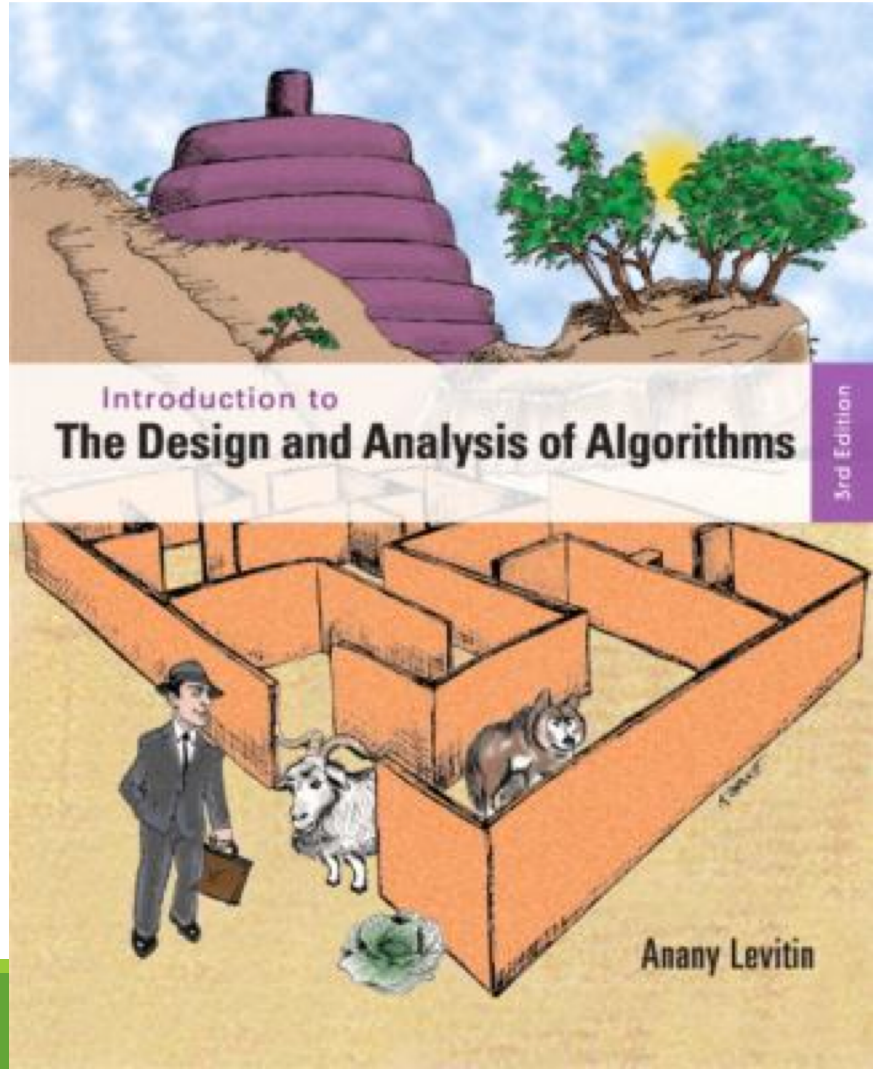
Introduction to the Desing & Analysis of Algorithm,

3rd Edition, Anany Levitin,

ISBN10: 0-13-231681-1, ISBN13: 978-0-13-231681-1

Free electronic version,

http://www.vgloop.com/_files/1394454921-126688.pdf



Syllabus

Course Objectives

- Know the major Algorithm design techniques
- Understand time and space efficiency and computational complexity
- Prove, design and program algorithms that meet the space and time requirement

Grading

- Homework 40 %
- Exam 1 15 %
- Exam 2 15 %
- Quizzes 10 %
- Final Exam 20 %

- **A** = 90+% points; **B** = 80 ~ 89%; **C** = 70 ~ 79%; **D** = 60 ~ 69%; **F** = below 60%

Policies

Late homework policy:

- assignment submitted after the due date without prior authorization will receive deduction of 10% of the potential points for that assignment for each day that the assignment is past due.

- No assignment is accepted during or after the week of final exam, i.e. after the study day on December 5th.

Exam policy:

- Make-up exams will be considered for only two reasons, sickness and unforeseen tragedy.

No makeup quizzes will be given.

Materials to be covered (tentative)

Lectures & Tests	Reading Assignment
Introduction	Chapter 1
Fundamentals of the Analysis	Chapter 2
Brute Force	Chapter 3
Decrease-and-Conquer	Chapter 4
Exam 1	Exam 1
Divide-and-Conquer	Chapter 5
Transform-and-Conquer	Chapter 6
Space and Time Trade-off	Chapter 7
Dynamic Programming	Chapter 8
Exam 2	Exam 2
Greedy Technique	Chapter 9
Iterative Improvement	Chapter 10
Limitation of Algorithm Power	Chapter 11
Copying with Limitation	Chapter 12
Final	Final

Syllabus (cont.)

Academic Integrity

- Do your own work on all tests and assignments
- Do not use external entities (friends, relatives, previous student's assignments, and especially internet sources) to obtain solutions to the homework and/or programming projects

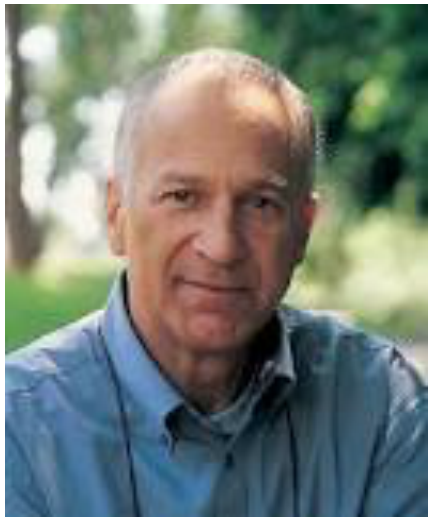
Student requiring accommodations

- Talk with me or email me

Why study algorithm?

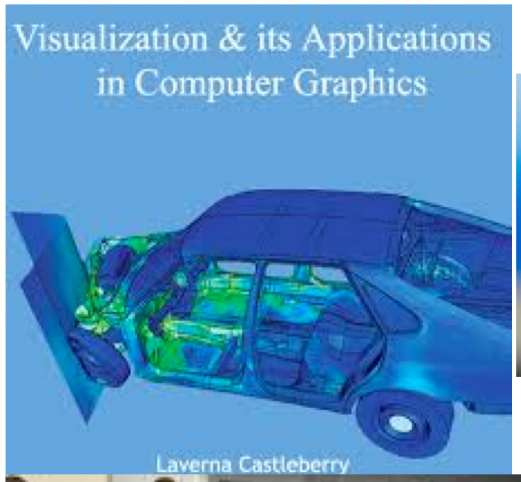
Algorithmics is more than a branch of computer science. It is the core of computer science, and, in all fairness, can be said to be relevant to most of science, business, and technology.

-- Algorithmics: the Spirit of Computing



David Harel, PhD.
Weizmann Institute of Science in Israel

Why study algorithm? (cont.)



Why study algorithm? (cont.)

Algorithms + Data Structures = Programs

Niklaus Emil Wirth, PhD.
Professor of Informatics at ETH Züric
Turing Award recipient (1984)

Word Origin

- The word algorithm comes from the name of a Persian mathematician. **Mohammed ibn Musa al-Khowarizmi**, latinized 'Algoritmi'.
- About 825, al-Khwarizmi wrote an Arabic language treatise on the Hindu–Arabic numeral system, which was translated into Latin during the 12th century under the title *Algoritmi de numero Indorum*. This title means "Algoritmi on the numbers of the Indians", where "Algoritmi" was the translator's Latinization of Al-Khwarizmi's name.

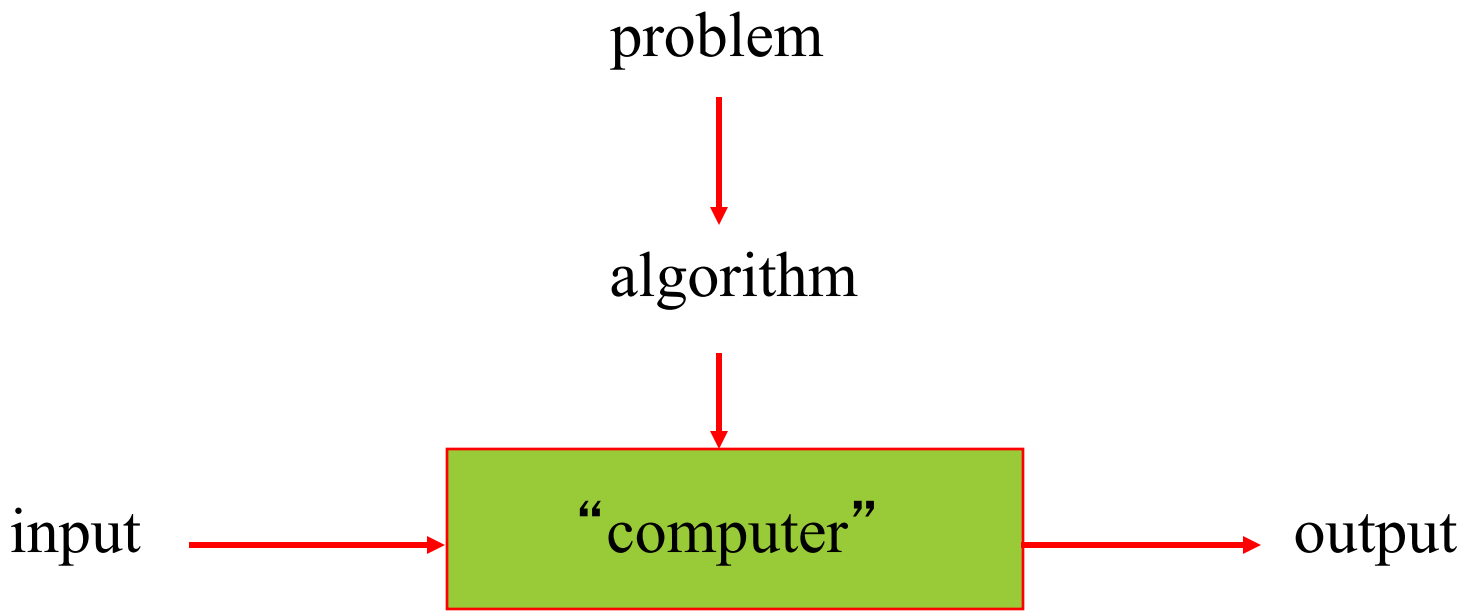
Statue of al-Khwārizmī in Amir Kabir University, Tehran

Born	c. 780
Died	c. 850

What is algorithm?

What is an algorithm?

An *algorithm* is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.



Important Points

- The nonambiguity requirement for each step of an algorithm cannot be compromised
- The range of inputs for which an algorithm works has to be specified carefully
- The same algorithm can be represented in several different ways.
- There may exist several algorithms for solving the same problem.

Greatest Common Divisor

The first algorithm "invented" in history was Euclid's algorithm for finding the greatest common divisor (GCD) of two natural numbers

Euclid's algorithm is named after the ancient Greek mathematician Euclid, who first described it in his *Elements* (c. 300 BC)

Definition: gcd(*x, y*) is the greatest common divisor of two nonnegative, not both zero integers *x* and *y*

Greatest Common Divisor

The GCD Problem:

- Input: two nonnegative, not both zero integers *x, y*
- Output: *GCD(x,y)* – their GCD

Examples: gcd(60,24) = ?

gcd(60, 0) = ?

gcd(0, 0) = ?

Euclid's Algorithm

Euclid's algorithm is based on repeated application of equality

$$\gcd(m,n) = \gcd(n, m \bmod n)$$

until the second number becomes 0, which makes the problem trivial.

Example: gcd(60,24) = ?

Two descriptions of Euclid's algorithm

Step 1 If $n = 0$, return m and stop; otherwise go to Step 2

Step 2 Divide m by n and assign the value of the remainder to r

Step 3 Assign the value of n to m and the value of r to n. Go to Step 1.

while $n \neq 0$ do

$r \leftarrow m \bmod n$

$m \leftarrow n$

$n \leftarrow r$

return m

Other methods for computing gcd(*m,n*)

Consecutive integer checking algorithm

Step 1 Assign the value of min{*m,n*} to *t*

Step 2 Divide *m* by *t*. If the remainder is 0, go to Step 3; otherwise, go to Step 4

Step 3 Divide *n* by *t*. If the remainder is 0, return *t* and stop; otherwise, go to Step 4

Step 4 Decrease *t* by 1 and go to Step 2

Does it work? No if given n=0

Other methods for gcd(*m,n*) [cont.]

Middle-school procedure

Step 1 Find the prime factorization of *m*

Step 2 Find the prime factorization of *n*

Step 3 Find all the common prime factors

Step 4 Compute the product of all the common prime factors and return it as gcd*(m,n*)

$$60 = 2 \cdot 2 \cdot 3 \cdot 5$$
$$24 = 2 \cdot 2 \cdot 2 \cdot 3$$
$$\gcd(60, 24) = 2 \cdot 2 \cdot 3 = 12$$

Is this an algorithm? The answer is NO according our definition due to the ambiguity of steps 1-3

Important Points

- The nonambiguity requirement for each step of an algorithm cannot be compromised
- The range of inputs for which an algorithm works has to be specified carefully
- The same algorithm can be represented in several different ways.
- There may exist several algorithms for solving the same problem.

Sieve of Eratosthenes to find prime numbers

1. Initializing a list of prime candidates with consecutive integers from 2 to n
2. On its first iteration, eliminate from the list all multiples of 2, i.e., 4, 6, and so on.
3. Move to next item

Sieve of Eratosthenes to find prime numbers

1. Initializing a list of prime candidates with consecutive integers from 2 to n
2. On its first iteration, eliminate from the list all multiples of 2, i.e., 4, 6, and so on.
3. Move to next item

As an example, consider the application of the algorithm to finding the list of primes not exceeding $n = 25$:

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
2	3		5		7		9		11		13		15		17		19		21		23		25
2	**3**		5		7				11		13				17		19				23		25
2	3		**5**		7				11		13				17		19				23		

Sieve Algorithm

Input: Integer $n \geq 2$

Output: List of primes less than or equal to n

```
for p ← 2 to n do  A[p] ← p          // Start with n-1 prime candidates
for p ← 2 to n do
    if A[p] ≠ 0                      //p hasn't been previously eliminated from the list
        j ← p * p                    //we can improve it!
        while j ≤ n  do
            A[j] ← 0                 //mark element as eliminated
            j ← j + p
```

Example: 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Sieve Algorithm

Input: Integer $n \geq 2$

Output: List of primes less than or equal to n

for $p \leftarrow 2$ to n do $A[p] \leftarrow p$ // Start with prime candidates

for $p \leftarrow 2$ to $\lfloor \sqrt{n} \rfloor$ do

  if $A[p] \neq 0$ //p hasn't been previously eliminated from the list

    $j \leftarrow p * p$

    while $j \leq n$ do

      $A[j] \leftarrow 0$ //mark element as eliminated

      $j \leftarrow j + p$

Sieve of Eratosthenes to find prime numbers

1. Initializing a list of prime candidates with consecutive integers from 2 to n
2. On its first iteration, eliminate from the list all multiples of 2, i.e., 4, 6, and so on.
3. Move to next item

As an example, consider the application of the algorithm to finding the list of primes not exceeding $n = 25$:

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
2	3		5		7		9		11		13		15		17		19		21		23		25
2	**3**		5		7				11		13				17		19				23		25
2	3		**5**		7				11		13				17		19				23		

Algorithm Design and Analysis Process

- Understand the problem: was the problem solved before?
  - If yes, how well can we improve over existing algorithms?
  - If no, how to solve the problem?

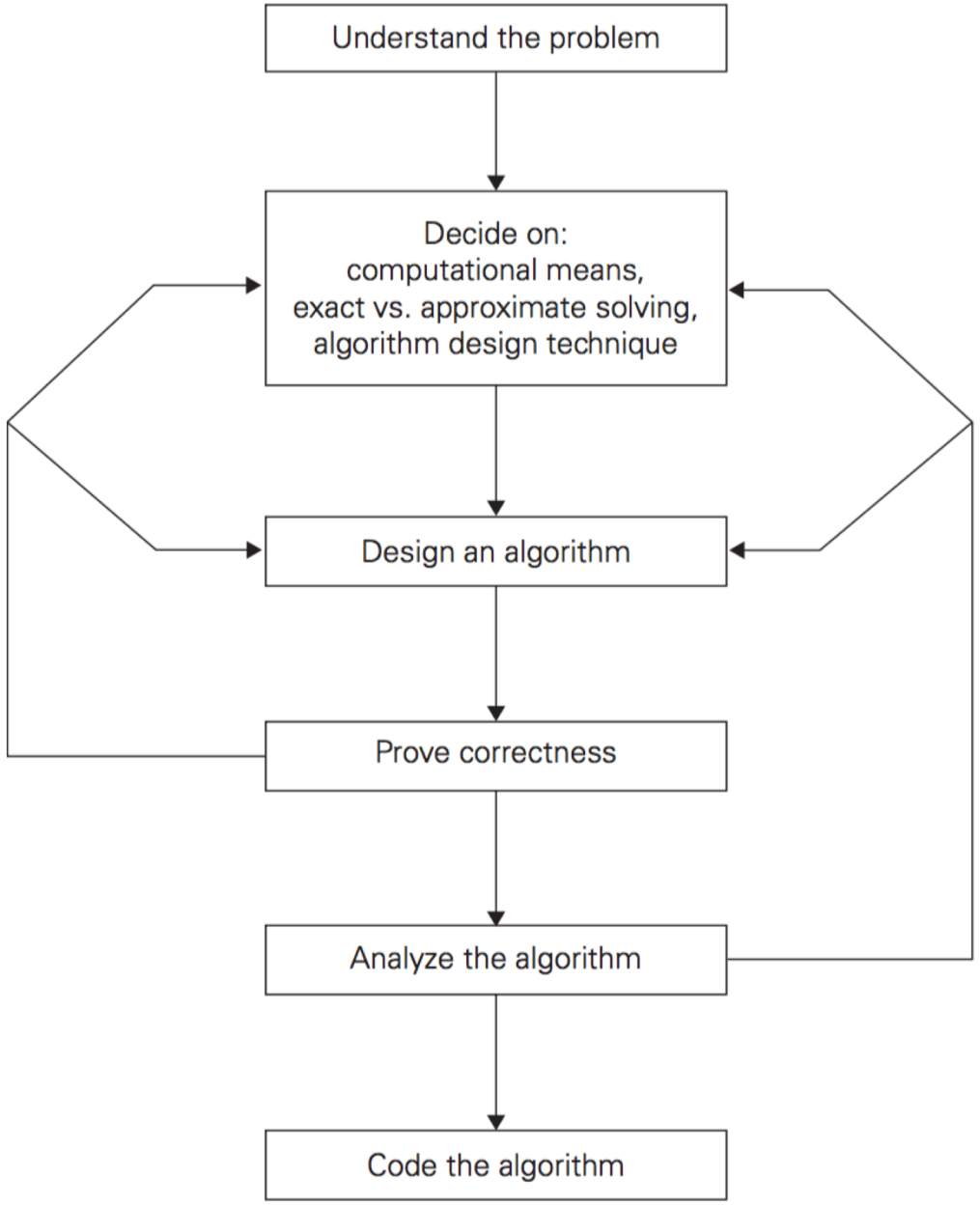


FIGURE 1.2 Algorithm design and analysis process.

Important problem types

sorting

searching

string processing

graph problems

combinatorial problems

geometric problems

numerical problems

Capabilities of computational Device

- Random-access machine (RAM)
  - Each ‘simple’ operation (+, *, -, =, if, call) takes exactly 1 time step.
  - we have as much memory as we need
  - Each memory access takes exactly one time step
  - Instructions are executed one after another
- Parallel algorithms

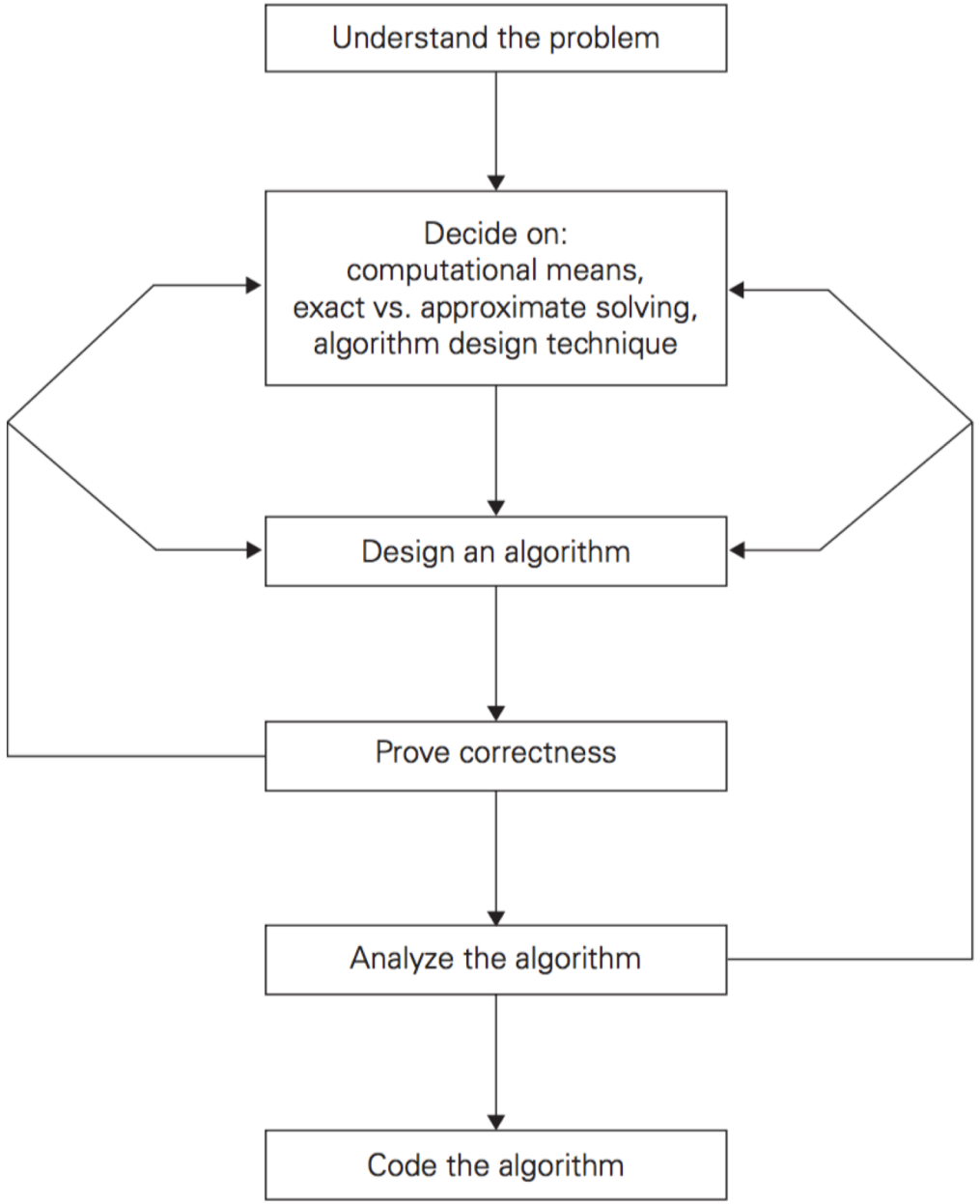


FIGURE 1.2 Algorithm design and analysis process.

Design an Algorithm

- This class will cover many algorithm design techniques.
- Design an algorithm and data structures
- Exact vs. approximate algorithm
- Do your best and keep a good record of your work

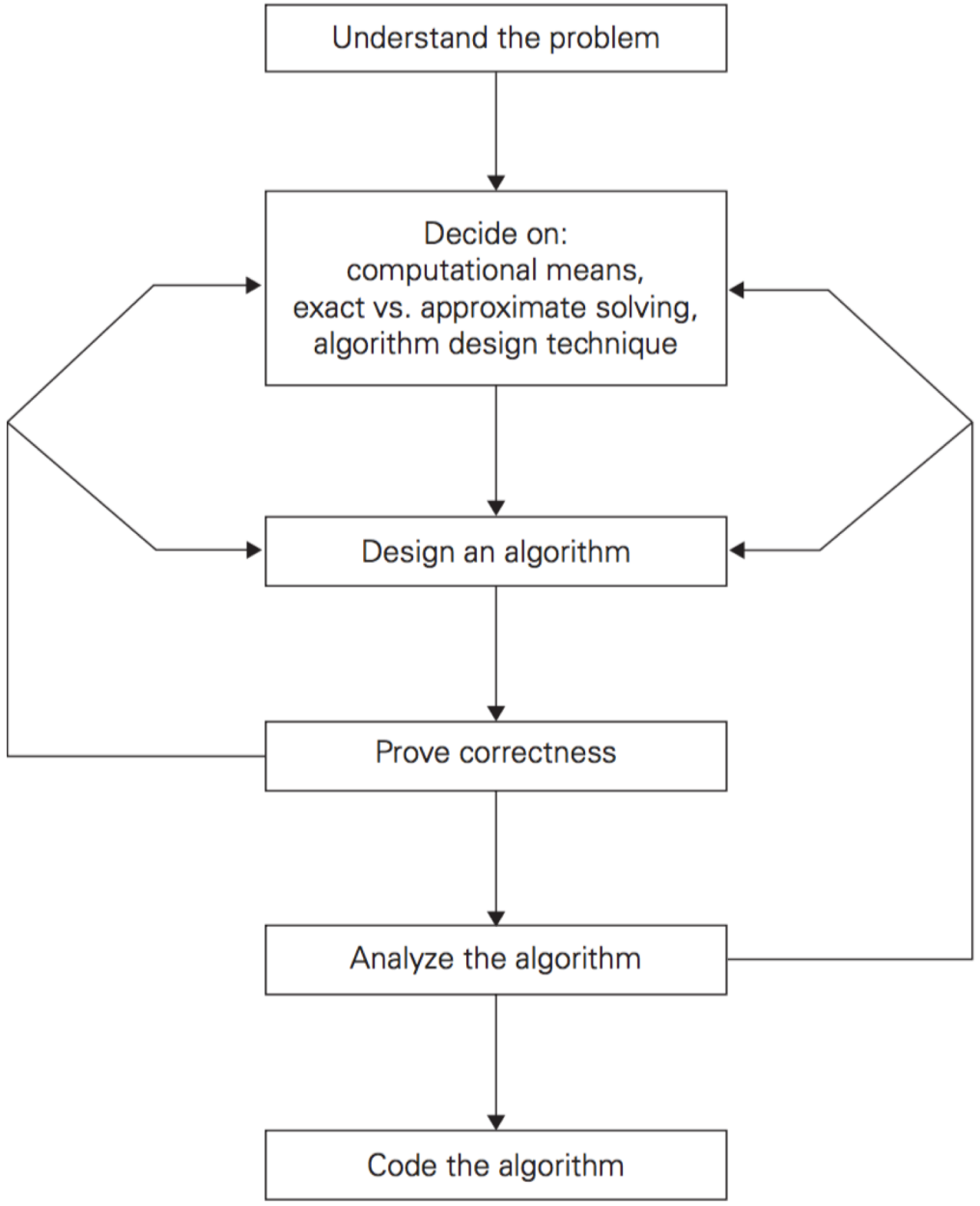


FIGURE 1.2 Algorithm design and analysis process.

Algorithm design techniques/strategies

- Brute force
- Divide and conquer
- Decrease and conquer
- Transform and conquer
- Space and time tradeoffs
- Greedy approach
- Dynamic programming
- Iterative improvement
- Backtracking
- Branch and bound

Prove correctness

Mathematical proof to demonstrate correctness

A counter-example to show an algorithm fail

For approximate algorithm, prove its upper or lower bound

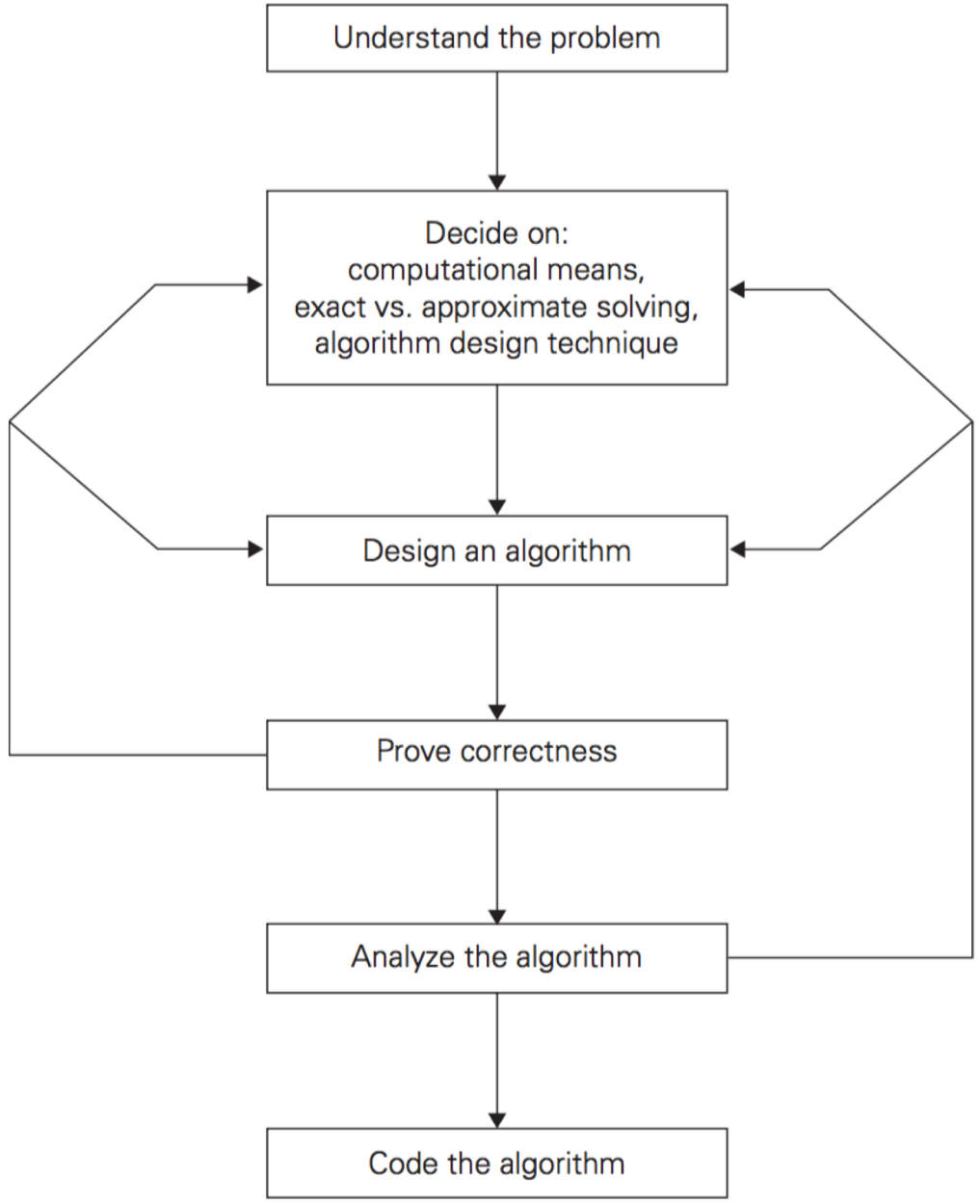


FIGURE 1.2 Algorithm design and analysis process.

Proof of Correctness

- After the design we need to prove that the algorithm is correct
  - Mathematical proof may be needed, For example prove that gcd(m,n) = gcd(n, m mod n)
  - Or use loop invariant analysis apply induction
- For some algorithms, a proof of correctness is quite easy. For others it can be quite complex

Analyzing an algorithm

Quality of an algorithm

- Time & space efficiency
- Simplicity
- Generality

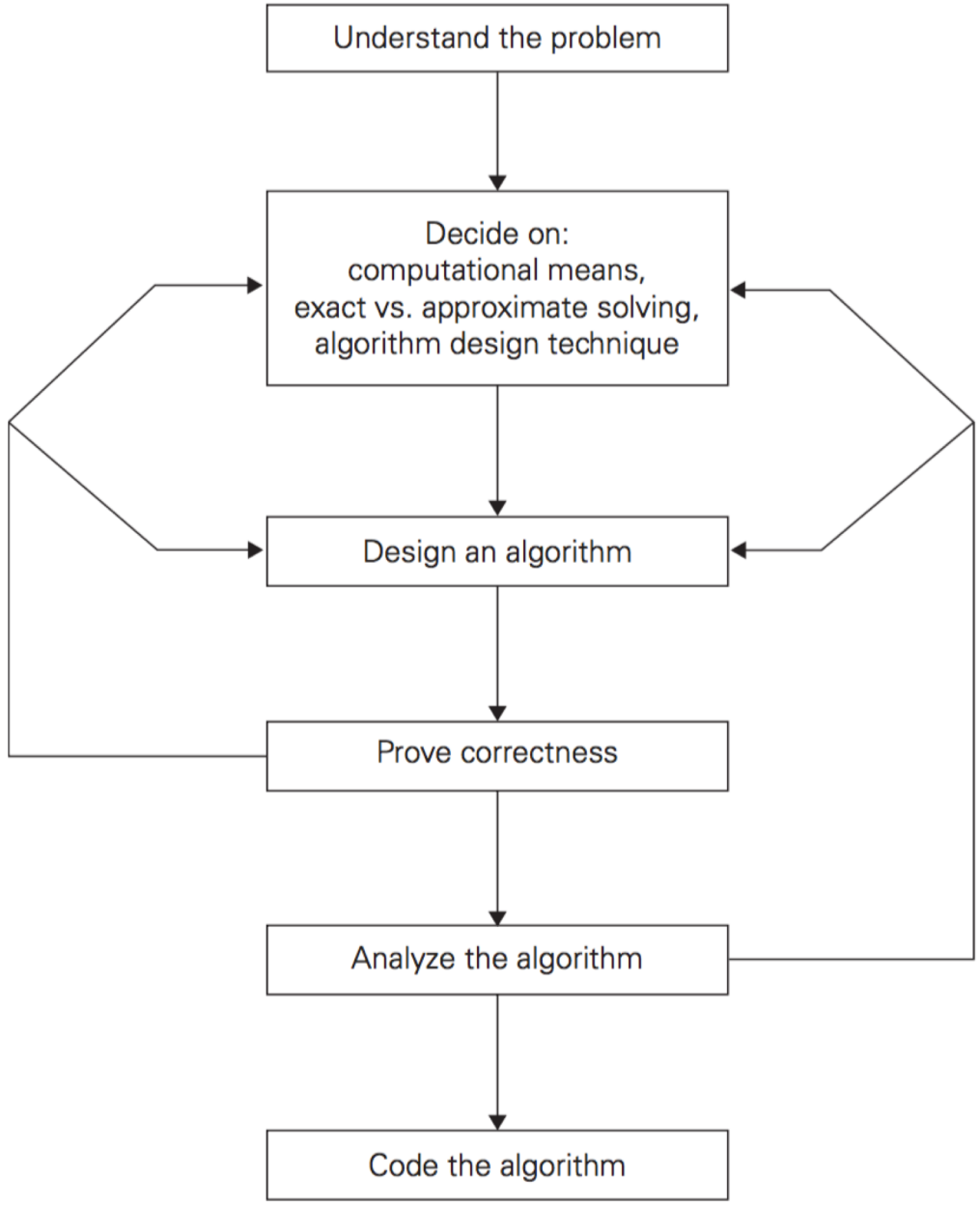


FIGURE 1.2 Algorithm design and analysis process.

Coding an algorithm

Efficient implementation is critical

Evaluating your algorithm on

- Simulation/real data
- Large data
- Test on different / representative data

Version management (GitHub, etc)

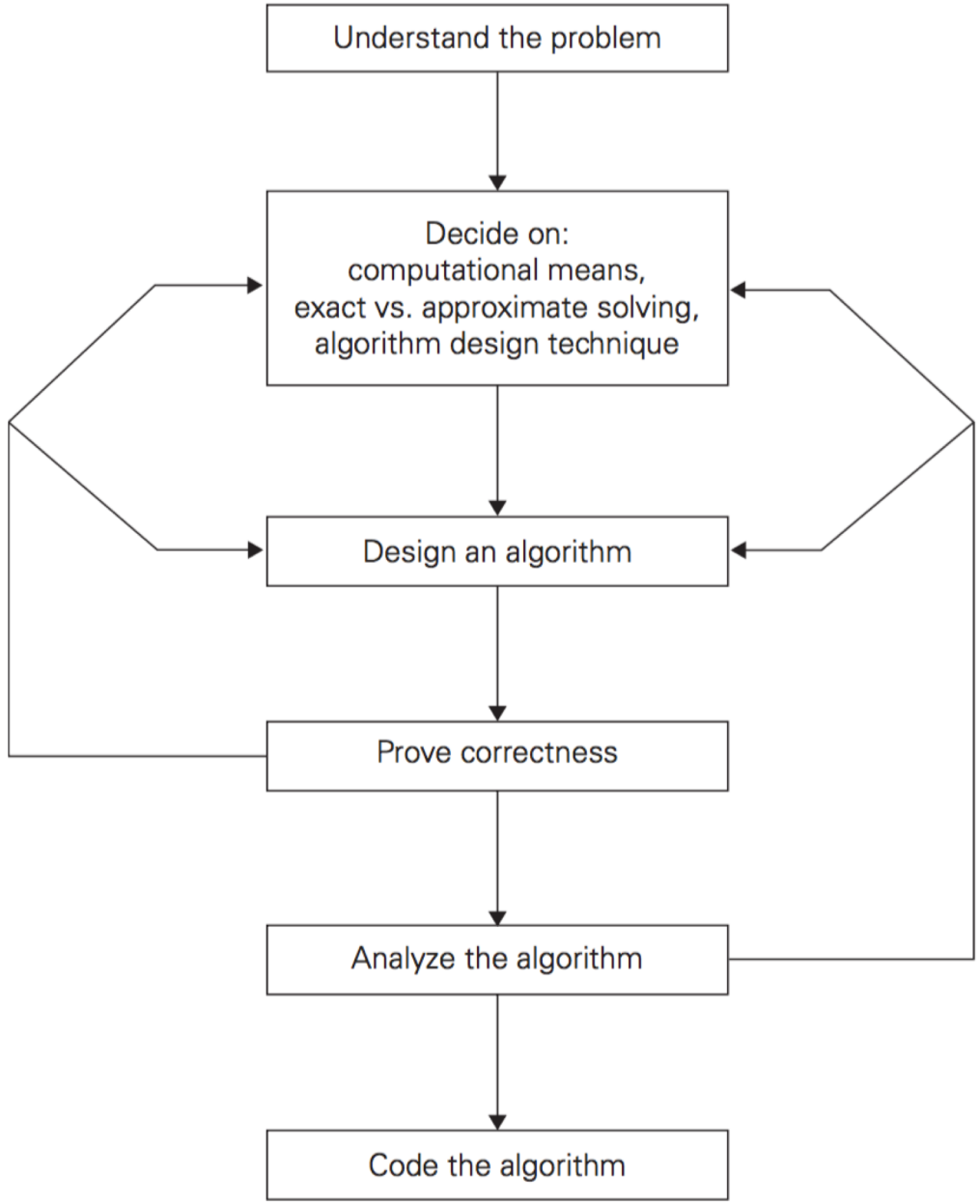


FIGURE 1.2 Algorithm design and analysis process.

Two themes in this class

- How to design algorithms
- How to analyze algorithm efficiency

Analysis of algorithms

How good is the algorithm?

- time efficiency
- space efficiency

Does there exist a better algorithm?

- lower bounds
- optimality

Fundamental data structures

Linear

Trees

Graphs

Sets and dictionaries

Linear Data Structures

Two most important

- Array
  - A (one-dimensional) ***array*** is a sequence of *n* items of the same data type stored contiguously
- Linked list
  - A sequence of zero or more items called nodes

Arrays

Items accessed by specifying the value of the array's ***index***

Index: integer between 0 and *n*-1 or 1 and *n*

All elements occupy same amount of storage, and are accessible in same constant amount of time

Used to implement ***strings*** among other data structures

FIGURE 1.3 Array of n elements

Singly Linked List

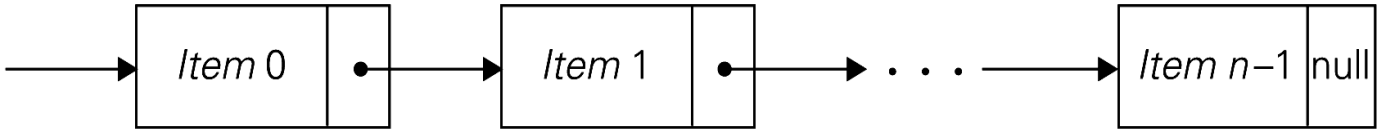


FIGURE 1.4 Singly linked list of n elements

Each node contains: ***data*** and a ***pointer***

Null (or ***Nil***) pointer indicates absence of node's successor

To access a node, start traversal from list's first node

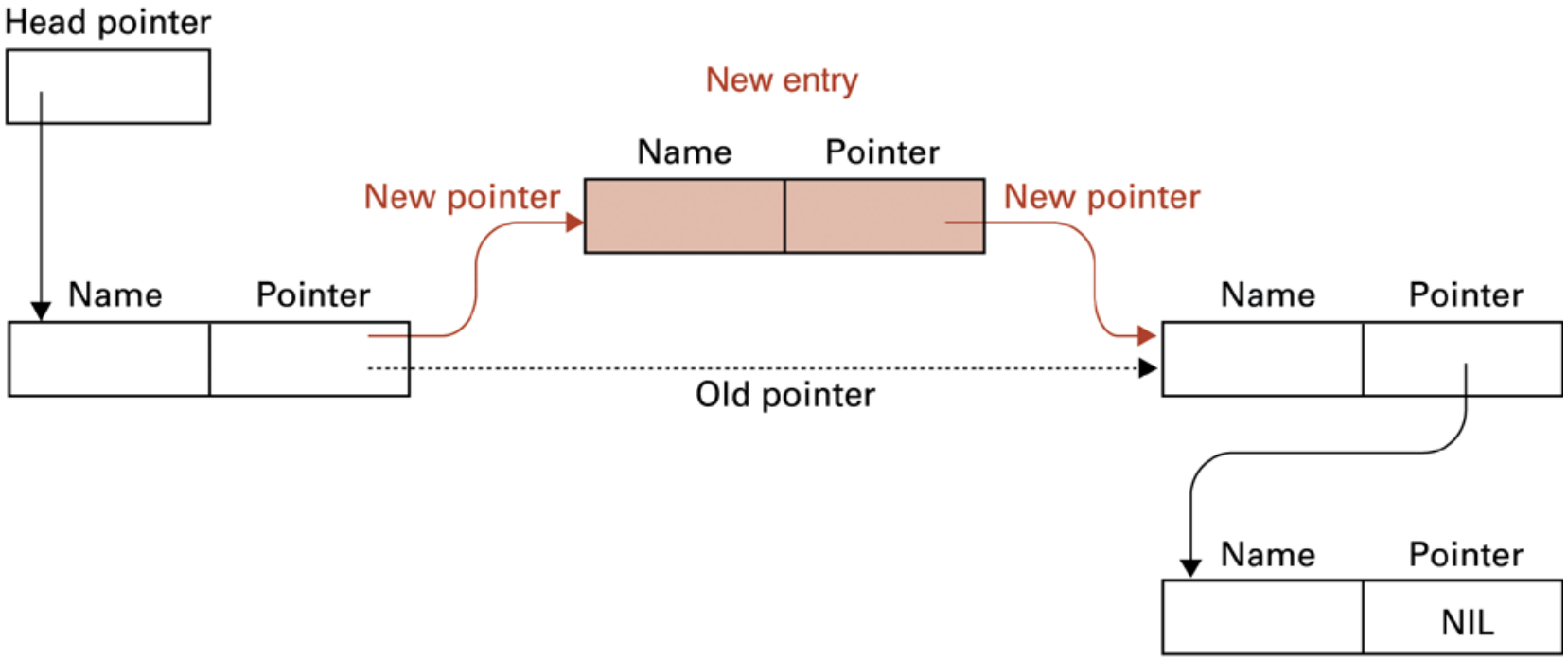
- Time to access a node depends on where it is in the list

No preliminary reservation of memory needed

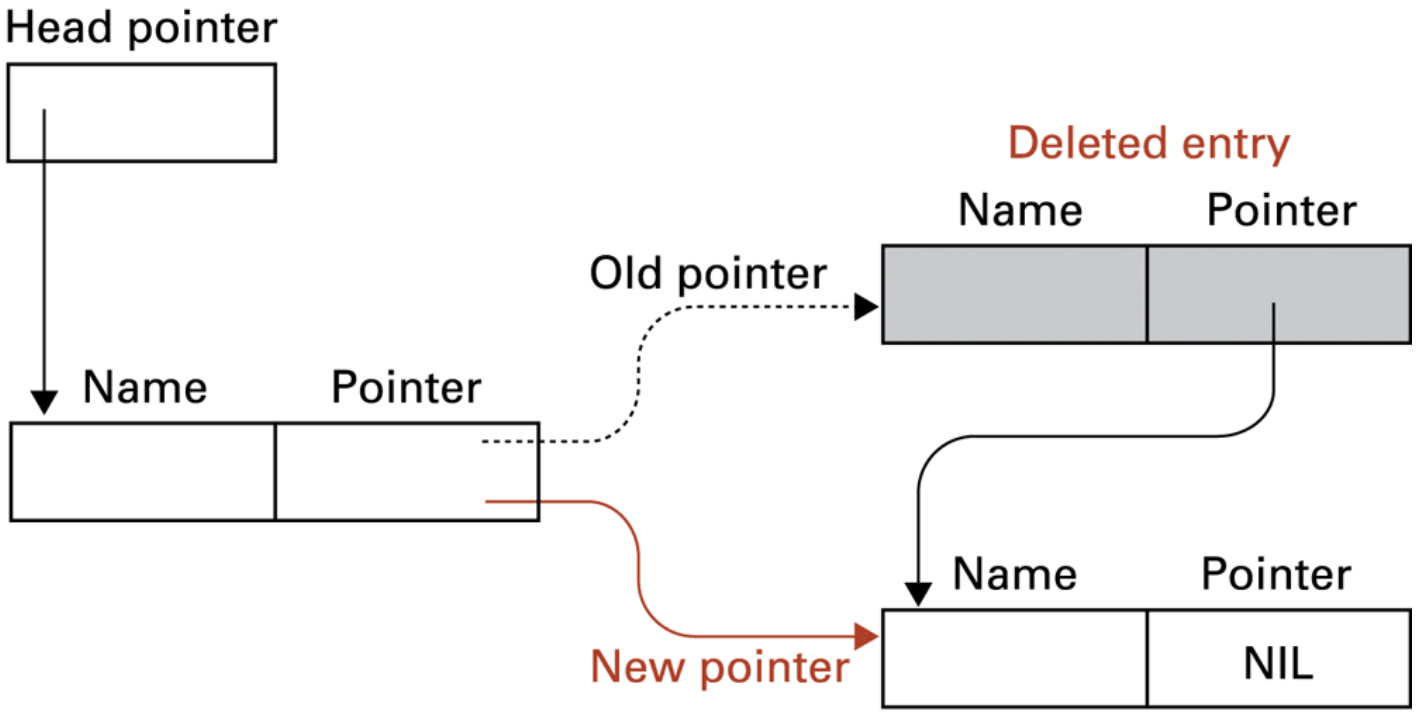
Header**:** starting node with info such as current length, pointer to first node, and pointer to last node

Efficient on ***insertions*** and ***deletions***

Inserting an entry into a Linked List



Deleting an entry from a Linked List



Doubly Linked List

Every node except first and last contains pointers to both its successor and predecessor

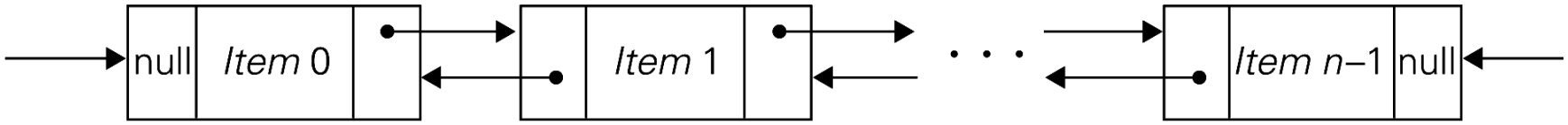


FIGURE 1.5 Doubly linked list of n elements

Linear List (or simply List)

Finite sequence of data items arranged in a certain linear order

Implementation represented by array or linked list

Basic operations: ***search***, ***insert***, ***delete***

Two special types of lists

- ***Stacks***
- ***Queues***

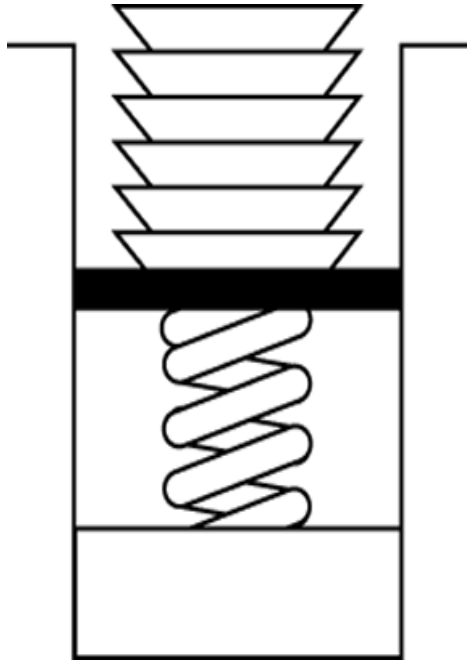
The Stack *Abstract* Data Type

Elements stored and retrieved one at a time.

Elements retrieved in reverse order of their time of storage, i.e., the latest element stored is the next element to be retrieved.

Sometimes referred to as a Last-In-First-Out (LIFO) or First-In-Last-Out (FILO) structure.

- Elements previously stored cannot be retrieved until the latest (top) element has been retrieved.



ADT Stack Operations

Only the top element of a stack is visible

- Number of operations performed by a stack are few

Need the ability to

- Inspect the top element
- Add (push) a new element on the stack
- Delete (pop) the top element
- Test for an empty stack

Some Applications of Stacks

Check to see if parentheses match

Reverse the line order of a text file

Palindrome finding: *Able was I ere I saw Elba*

Evaluation of complex expressions (intermediate values stored)

- e.g., postfix expressions

Recursive algorithms

Activation stack (method calls)

The Queue Abstract Data Type

New items enter at the back, rear, or tail of the queue

Items leave from the front or head of the queue

First-in, first-out (FIFO) property

- The first item inserted into a queue is the first item to leave

ADT Queue Operations

Create an empty queue

Destroy a queue

Determine whether a queue is empty

Add a new item to the queue: *enqueue*

Remove the item that was added earliest: *dequeue*

Retrieve the item that was added earliest

Some Applications of Queues

Print jobs

Virtually every real-life line

Phone calls to large companies

Waiting lists

(Rooted) Tree

A (***rooted***) ***tree*** is a collection of nodes. The collection can be empty; otherwise, a tree consists of a distinguished node r, called the ***root***, and zero or more nonempty (sub)trees $T_1, T_2, \ldots, T_k$, each of whose roots are connected by a directed edge from r

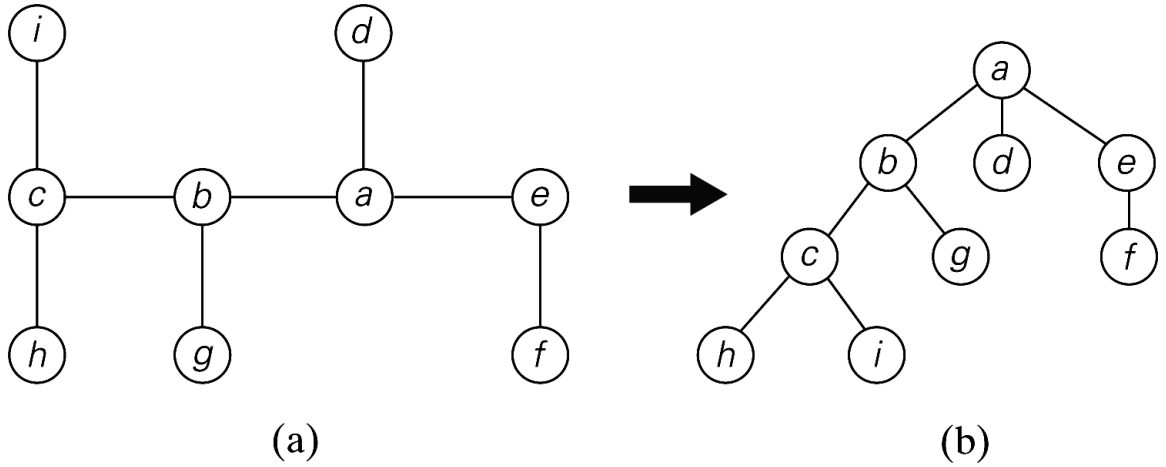


FIGURE 1.11 (a) Free tree. (b) Its transformation into a rooted tree.

Ordered Trees

An ***ordered tree*** is a rooted tree in which all the children of each vertex are ordered (*conveniently left to right*)

A ***binary tree*** is an ordered tree in which every vertex has no more than two children and each child is designated either a ***left child*** or a ***right child***

The subtree with its root at the left (right) child of a vertex is called the ***left*** (***right***) ***subtree*** of that vertex

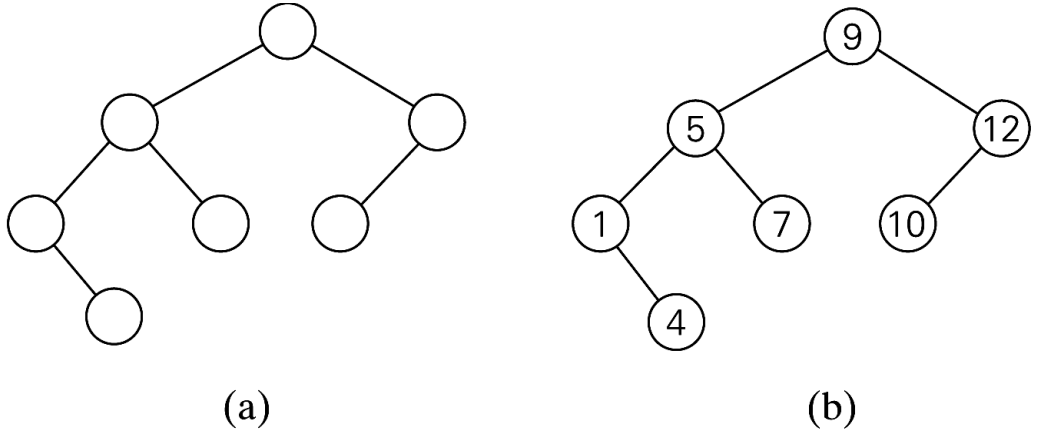


FIGURE 1.12 (a) Binary tree. (b) Binary search tree.

Binary Search Trees

Binary trees with ***labels*** (e.g., numbers) on vertices

- The label assigned to each parental vertex is ***larger*** than all the labels in its left subtree and ***smaller*** than all the labels in its right subtree

Efficiency of most BST algorithms depend on tree's height

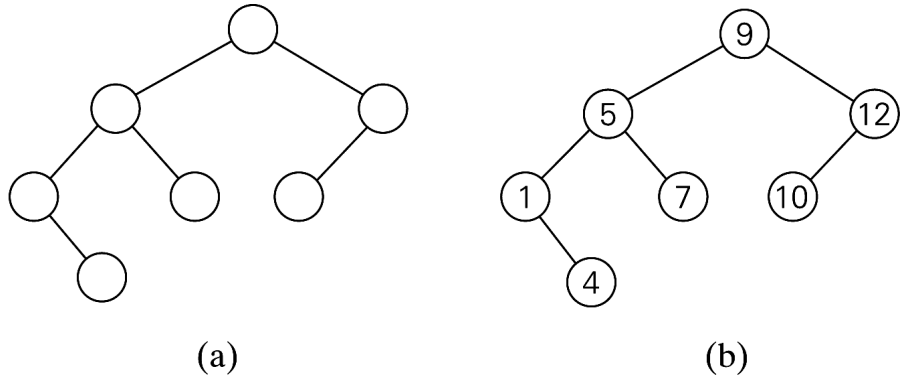


FIGURE 1.12 (a) Binary tree. (b) Binary search tree.

Implementing BSTs

Implemented by a collection of ***nodes*** representing vertices

Each node contains ***label***, ***left pointer*** representing left child, and ***right pointer*** representing right child

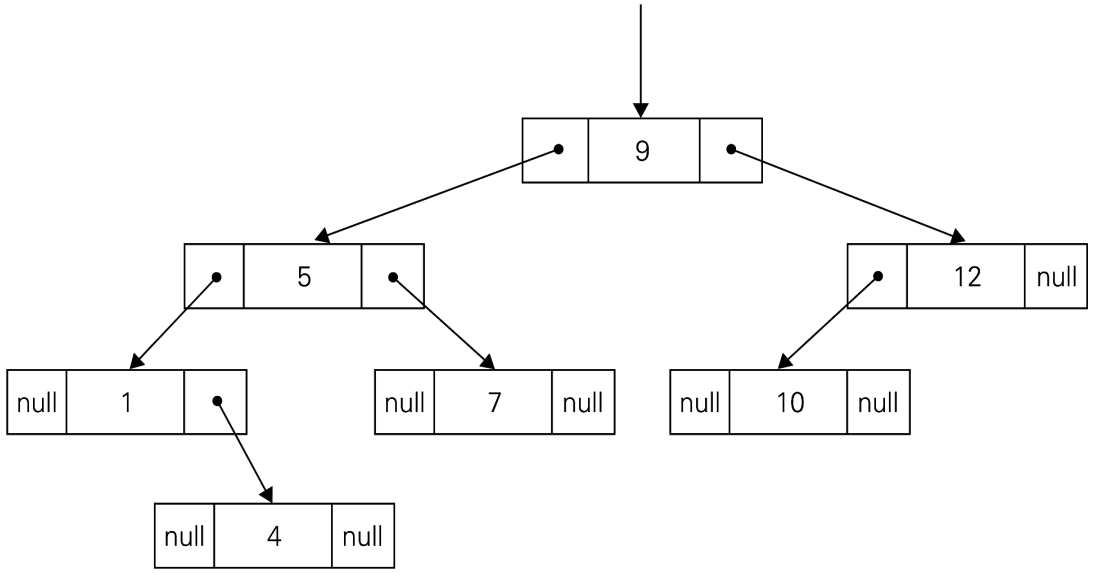


FIGURE 1.13 Standard implementation of the binary search tree in Figure 1.12b

(Free) Trees

A (***free***) ***tree*** is a connected acyclic graph

A ***forest*** is a graph with no cycles and not necessarily connected: each of its connected components is a tree

An important property of trees: $|E| = |V| - 1$

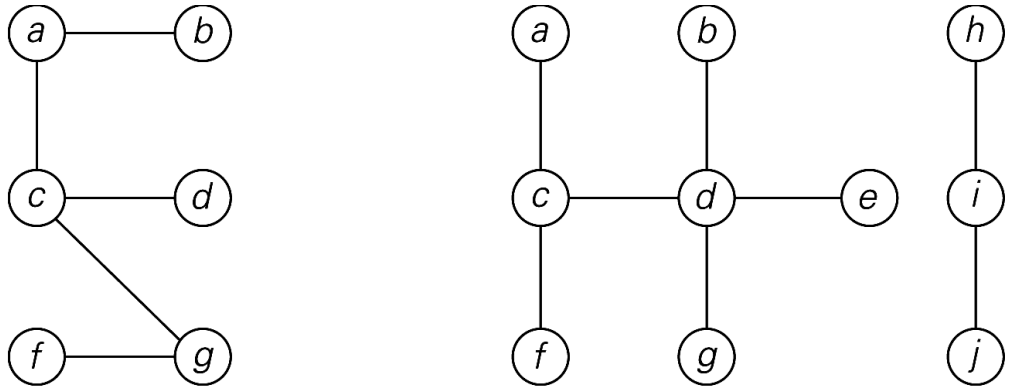


FIGURE 1.10 (a) Tree. (b) Forest.

Graphs

A graph $G = <V, E>$ where V is a finite set of ***vertices*** and E is a set of ***edges*** (pairs of vertices)

If the pairs of vertices are unordered, then $(u,v) = (v,u)$

- u and v are ***adjacent*** and are connected by the ***undirected*** edge (u,v)
- u and v are ***endpoints*** of the edge (u,v), and they are ***incident*** to this edge
- the edge (u,v) is also said to be ***incident*** to its endpoints u and v

A graph G is called ***undirected*** if every edge in it is undirected

If $(u,v) \neq (v,u)$, then (u,v) is said to be a directed edge from u (***tail***) to v (***head***)

A graph whose every edge is directed is called directed (***digraph***)

Graphs and Applications

Facebook Friends

Very-large-scale integration (VLSI)

Network Traffic flow

Biological analysis – Network biology

Undirected and Directed Graph

Undirected graph in 1.6 (a):

- $V = \{a, b, c, d, e, f\}$, $E = \{(a,c), (a,d), (b,c), (b,f), (c,e), (d,e), (e,f)\}$

Digraph in 1.6 (b):

- $V = \{a, b, c, d, e, f\}$, $E = \{(a,c), (b,c), (b,f), (c,e), (d,a), (d,e), (e,c), (e,f)\}$

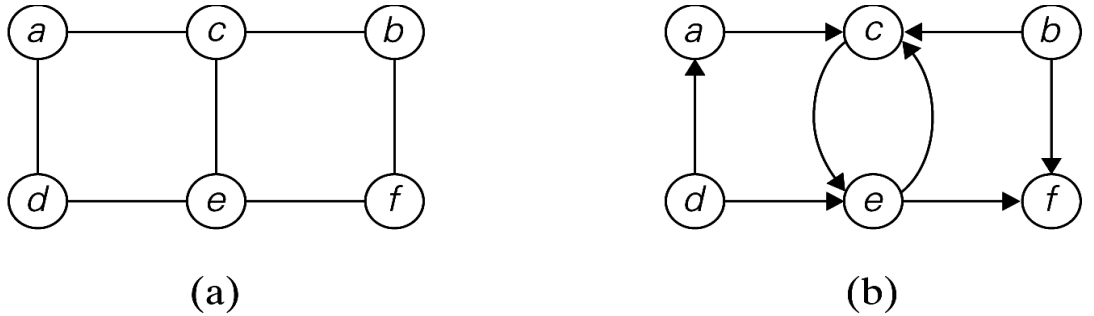


FIGURE 1.6 (a) Undirected graph. (b) Digraph.

Graph Representations

Adjacency matrix for G with *n* vertices: *n* x *n* boolean matrix

Adjacency lists is a collection of linked lists, one for each vertex, that contain all the vertices connected to it by an edge

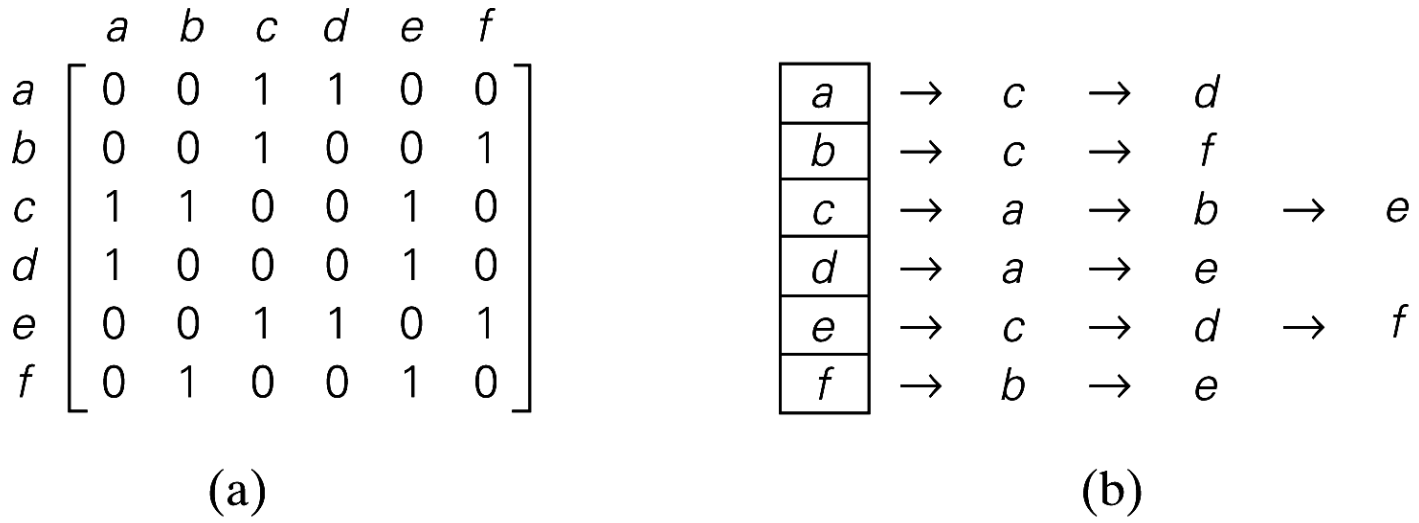


FIGURE 1.7 (a) Adjacency matrix and (b) adjacency lists of the graph in Figure 1.6a

Weighted Graphs

A ***weighted graph (***or weighted digraph) is a graph (or digraph) with numbers assigned to its edges

- The numbers are called ***weights*** or ***costs***

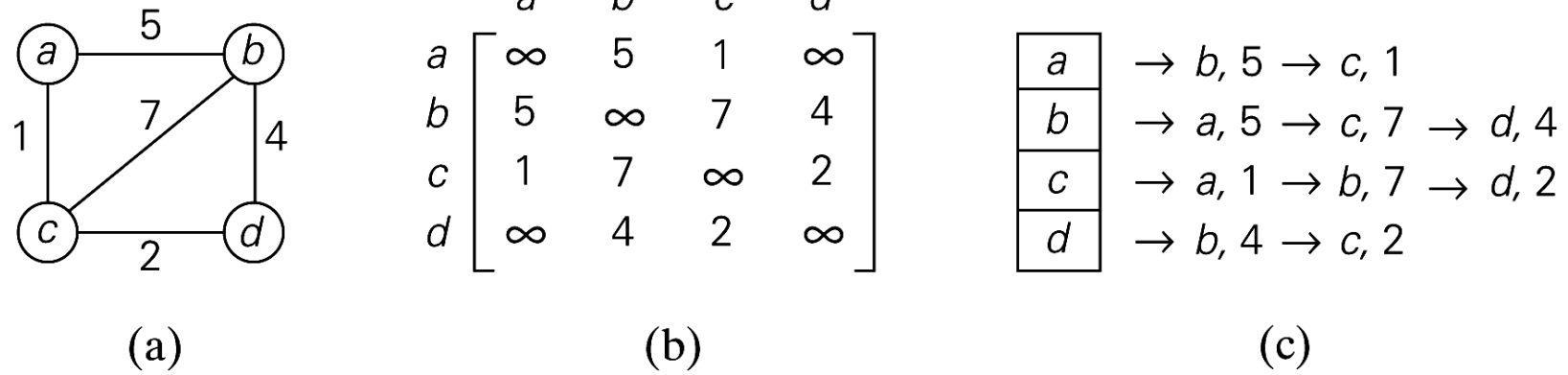


FIGURE 1.8 (a) Weighted graph. (b) Its weight matrix. (c) Its adjacency lists.

Paths and Cycles

A ***path*** from vertex *u* to vertex *v* of graph G is a sequence of adjacent (connected by an edge) vertices that start with *u* and ends with *v*

If all vertices are distinct, the path is ***simple***

The ***length*** of a path is the total number of vertices in the path minus one, or the total number of edges in the path

In the case of digraphs, the paths are called ***directed paths***

Connected Graphs

Graph G is ***connected*** if for every pair of its vertices u and v there's a path from u to v

A ***cycle*** is a path of positive length that starts and ends at the same vertex and does not traverse the same edge more than once

- A graph with no cycles is said to be ***acyclic***

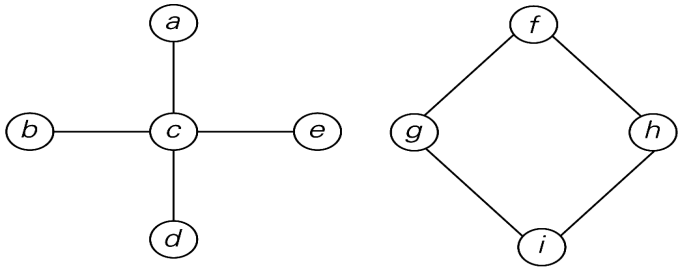


FIGURE 1.9 Graph that is not connected

Sets and Dictionaries

A set is an unordered collection (possibly empty) of distinct elements

Specification of a set

- Roster method: e.g., **S** = {2, 3, 5, 7}
- Set-builder method: e.g., $S = \{n | \ n \text{ is a prime number}, n < 10\}$

Set operations: $\in$, $\cup$, $\cap$, $-$

Implementation of Sets

Universal set approach

- Let U be a set of n elements. Then any subset S of U can be represented by a string of size n, called a ***bit vector***, in which the i^{th} element is 1 *iff* the i^{th} element of U is included in set S.

- Example: $U = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$. $S = \{2, 3, 5, 7\}$ is represented by bit string **0 1 1 0 1 0 1 0 0**

List structure approach

- List can have duplicates. Introduce a ***multiset*** or ***bag***, an unordered collection of not necessarily distinct items

- Relax the notion of a list being a collection of ordered items

Exercise

There are four people who want to cross a rickety bridge; they all begin on the same side. You have **17** minutes to get them all across to the other side. It is night, and they have one flashlight. A maximum of two people can cross the bridge at one time. Any party that crosses, either one or two people, must have the flashlight with them. The flashlight must be walked back and forth; it cannot be thrown, for example. Person 1 takes **1** minute to cross the bridge, person 2 takes **2** minutes, person 3 takes **5** minutes, and person 4 takes **10** minutes. A pair must walk together at the rate of the slower person's pace. (Note: According to a rumor on the Internet, interviewers at a well-known software company located near Seattle have given this problem to interviewees.)

Next week

Chapter 2 Fundamentals of the analysis of Algorithm Efficiency