Introduction to Algorithm design and Analysis

This lecture has been prepared using the book “Introduction to the Design and Analysis of Algorithms” by Anany Levitin.

Why Do You Need To Study Algorithms?

- There are both practical and theoretical reasons to study algorithms:
  - From a **practical standpoint**, you have to know a standard set of important algorithms from different areas of computing;
    - In addition, you should be able to design new algorithms and analyze their efficiency.
  - From the **theoretical standpoint**, the study of algorithms, sometimes called algorithmics, has come to be recognized as the cornerstone of computer science.
    - Algorithmics is more than a branch of computer science. It is the core of computer science
    - Computer programs would not exist without algorithms.

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.

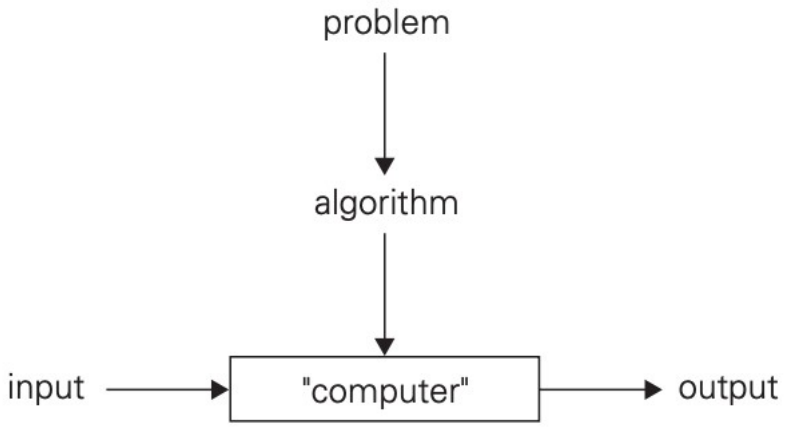


FIGURE 1.1 The notion of the algorithm.

What is an algorithm? (cont.)

- As examples illustrating the notion of the algorithm, we consider in three methods for solving the same problem: **computing the greatest common divisor of two integers**.
- These examples will help to illustrate several 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.
  - Algorithms for the same problem can be based on very different ideas and can solve the problem with dramatically different speeds.

Greatest Common Divisor

- **Greatest common divisor** of two nonnegative, not-both-zero integers m and n, denoted gcd(m, n), is defined as the largest integer that divides both m and n evenly, i.e., with a remainder of zero.
- Several Algorithms for solving the problem:
  - Euclid’s algorithm
  - Consecutive integer checking algorithm
  - Middle-school procedure

Euclid’s Algorithm

- **Euclid’s Algorithm** is based on repeated application of equality

  gcd(m, n) = gcd(n, m mod n)

until the second number becomes 0, which makes the problem trivial. the last value of m is also the greatest common divisor of the initial m and n.

- Example: gcd(60, 24) = gcd(24,12) = gcd(12,0) = 12

Euclid’s Algorithm (cont.)

- **Euclid’s algorithm** for computing gcd(m, n)
  - **Step 1:** If n = 0, return the value of m as the answer and stop; otherwise, proceed to Step 2.
  - **Step 2:** Divide m by n and assign the value of the remainder to r.
  - **Step3:** Assign the value of n to m and the value of r to n. Go to Step1.

Euclid's Algorithm (cont.)

- Alternatively, we can express the same algorithm in pseudocode:

ALGORITHM *Euclid*(m, n)

//Computes gcd(m, n) by Euclid's algorithm
//Input: Two nonnegative, not-both-zero integers m and n
//Output: Greatest common divisor of m and n
while $n \neq 0$ **do**
    $r \leftarrow m \bmod n$
    $m \leftarrow n$
    $n \leftarrow r$
return m

Euclid’s Algorithm (cont.)

- How do we know that Euclid’s algorithm eventually comes to a stop?
  - This follows from the observation that the second integer of the pair gets smaller with each iteration and it cannot become negative.
  - Indeed, the new value of n on the next iteration is m mod n, which is always smaller than n .
    - Hence, the value of the second integer eventually becomes 0, and the algorithm stops.

Consecutive integer checking algorithm

- Consecutive integer checking algorithm for computing gcd(m, n)
  - **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

- This algorithm starts by checking whether t divides both m and n: if it does, t is the answer; if it doesn’t, we simply decrease t by 1 and try again.

Consecutive integer checking algorithm (cont.)

- Note that unlike Euclid’s algorithm, this algorithm, in the form presented, does not work correctly when one of its input numbers is zero.
- This example illustrates why it is so important to specify the set of an algorithm’s inputs explicitly and carefully.

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)

- Example:
- gcd(60,24) can be computed as follows:

  60 = 2 . 2 . 3 . 5

  24 = 2 . 2 . 2 . 3

  gcd(60, 24) = 2 . 2 . 3 = 12.

Middle-school procedure (cont.)

- The middle- school procedure does not qualify, in the form presented, as a legitimate algorithm. Why?
  - Because the prime factorization steps are not defined unambiguously: they require a list of prime numbers,
  - Step 3 is also not defined clearly enough.
  - Much more complex and slower than Euclid’s Algorithm

Fundamentals of Algorithmic Problem Solving

- We can consider **algorithms** to be **procedural solutions to problems**.
- These **solutions are not answers but specific instructions for getting answers**.
- **It is this emphasis on precisely defined constructive procedures that makes computer science distinct from other disciplines.**
- In particular, this distinguishes it from theoretical mathematics, whose practitioners are typically satisfied with just proving the existence of a solution to a problem and, possibly, investigating the solution’s properties.

Algorithm design and analysis process

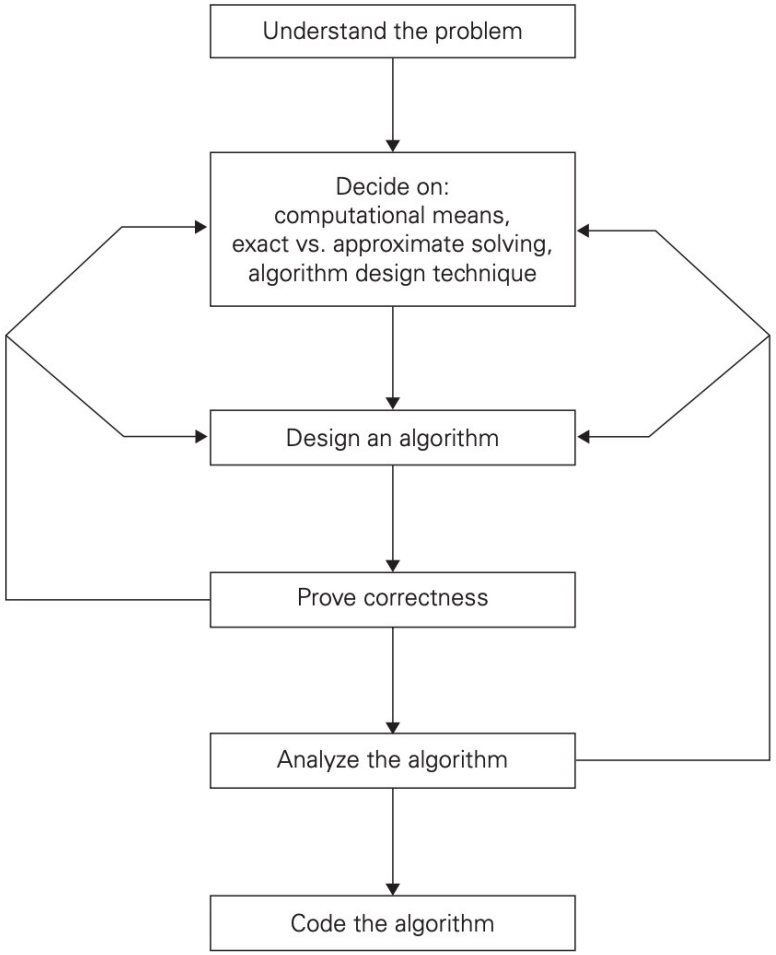


FIGURE 1.2 Algorithm design and analysis process.

Understanding the Problem

- The first thing you need to do before designing an algorithm is to understand completely the problem given:
  - Read the problem's description carefully and ask questions if you have any doubts about the problem.
  - Do a few small examples by hand, think about special cases, and ask questions again if needed.
  - Use a known algorithm for solving the problem. Of course, it helps to understand how such an algorithm works and to know its strengths and weaknesses, especially if you have to choose among several available algorithms.
  - Often you will not find a readily available algorithm and will have to design your own.
  - It is very important to specify exactly the set of instances the algorithm needs to handle.

Decide on Computational Means

- Once you completely understand a problem, you need to ascertain the capabilities of the computational device the algorithm is intended for.
- The vast majority of algorithms in use today are still destined to be programmed for a computer closely resembling the von Neumann machine—(a computer architecture). The essence of this architecture is captured by the so-called random-access machine (RAM).
- Its central assumption is that instructions are executed one after another, one operation at a time. Accordingly, algorithms designed to be executed on such machines are called **sequential algorithms**.
- The central assumption of the RAM model does not hold for some newer computers that can execute operations **concurrently, i.e., in parallel**.
  - Algorithms that take advantage of this capability are called **parallel algorithms**.

Decide on Computational Means (cont.)

- Studying the classic techniques for design and analysis of algorithms under the RAM model remains the cornerstone of algorithmics for the foreseeable future.

Decide on Computational Means (cont.)

- Sequential OR Parallel?
  - **Sequential Algorithm**
    - Instructions are executed one after another, one operation at time.
  - **Parallel Parallel**
    - Operations are executed concurrently.

Decide on Computational Means (cont.)

- Should you worry about the speed and amount of memory of a computer at your disposal?
  - If you are designing an algorithm as a scientific exercise, the answer is a qualified no., most computer scientists prefer to study algorithms in terms independent of specification parameters for a particular computer.
  - If you are designing an algorithm as a practical tool, the answer may depend on a problem you need to solve.
    - Even the “slow” computers of today are almost unimaginably fast. Consequently, in many situations you need not worry about a computer being too slow for the task.
    - There are important problems, however, that are very complex by their nature, or have to process huge volumes of data, or deal with applications where the time is critical. In such situations, it is imperative to be aware of the speed and memory available on a particular computer system.

Choosing between Exact and Approximate Problem Solving

- The next principal decision is to choose between solving the problem exactly (exact algorithm) or solving it approximately (approximation algorithm).

- Why would one opt for an approximation algorithm?
  - First, there are important problems that simply cannot be solved exactly for most of their instances;
    - Example: extracting square roots
  - Second, available algorithms for solving a problem exactly can be unacceptably slow because of the problem's intrinsic complexity.
    - This happens, in particular, for many problems involving a very large number of choices;
  - Third, an approximation algorithm can be a part of a more sophisticated algorithm that solves a problem exactly.

Decide on Algorithm Design Technique

- An **algorithm design technique** (or “strategy” or “paradigm”) is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing.
- **General algorithm design technique:**
  - Brute Force and Exhaustive Search
  - Decrease-and-Conquer
  - Divide-and-Conquer
  - Transform-and-Conquer
  - Space and Time Trade-Offs
  - Dynamic Programming
  - Greedy Technique
  - Iterative Improvement

Designing an Algorithm and Data Structures

- Designing an algorithm for a particular problem may still be a challenging task:
  - Some design techniques can be simply inapplicable to the problem in question.
  - Sometimes, several techniques need to be combined,
  - There are algorithms that are hard to pinpoint as applications of the known design techniques.
- Even when a particular design technique is applicable, getting an algorithm often requires a nontrivial ingenuity on the part of the algorithm designer.
- With practice, both tasks—choosing among the general techniques and applying them—get easier, but they are rarely easy.

Designing an Algorithm and Data Structures (Cont.)

- One should pay close attention to choosing data structures appropriate for the operations performed by the algorithm.
  - **For example**, performing a binary search on a sorted list would be inefficient if we used a **linked list** instead of an **array**, because accessing the middle element in a linked list takes linear time, whereas in an array it takes constant time.
  - This would increase the time complexity of each search step from O(1) to O(n), making the entire algorithm **O(n log n)** instead of the usual **O(log n)**.

Methods of Specifying an Algorithm

- Once you have designed an algorithm, you need to specify it in some fashion.
  - Using a **natural language** has an obvious appeal; however, the inherent ambiguity of any natural language makes a succinct and clear description of algorithms surprisingly difficult.
    - Nevertheless, being able to do this is an important skill that you should strive to develop in the process of learning algorithms.
  - **Pseudocode** is a mixture of a natural language and programming language- like constructs.
    - Pseudocode is usually more precise than natural language, and its usage often yields more succinct algorithm descriptions. Surprisingly, computer scientists have never agreed on a single form of pseudocode

Methods of Specifying an Algorithm (cont.)

- In the earlier days of computing, the dominant vehicle for specifying algorithms was a **flowchart**, a method of expressing an algorithm by a collection of connected geometric shapes containing descriptions of the algorithm's steps.
- This representation technique has proved to be inconvenient for all but very simple algorithms; nowadays, it can be found only in old algorithm books.

Proving an Algorithm's Correctness

- Once an algorithm has been specified, you have to prove its correctness.
- **Prove Correctness**: Prove that the algorithm yields a required result for every legitimate input in a finite amount of time.
  - For example, the correctness of Euclid's algorithm for computing the greatest common divisor stems from the correctness of the equality gcd(m, n) = gcd(n, m mod n), the simple observation that the second integer gets smaller on every iteration of the algorithm, and the fact that the algorithm stops when the second integer becomes 0.

Analyzing the Algorithm

- We usually want our algorithms to possess several qualities:
- After correctness, by far the most important is **efficiency**.
  - **Time efficiency**, indicating how fast the algorithm runs
  - **Space efficiency**, indicating how much extra memory it uses.
- Another desirable characteristic of an algorithm is **simplicity**.
  - Simpler algorithms are easier to understand and easier to program, consequently, the resulting programs usually contain fewer bugs.
- Yet another desirable characteristic of an algorithm is **generality**.
  - There are, in fact, two issues here: **generality of the problem the algorithm solves and the set of inputs it accepts.**

Analyzing the Algorithm (cont.)

- If you are not satisfied with the algorithm’s efficiency, simplicity, or generality, you must return to the drawing board and redesign the algorithm.

Coding an Algorithm

- Most algorithms are destined to be ultimately implemented as computer programs.
- Programming an algorithm presents both a **peril** and an **opportunity**.
- The **peril** lies in the possibility of making the transition from an algorithm to a program either **incorrectly** or very **inefficiently**.

Coding an Algorithm (cont.)

- Some influential computer scientists strongly believe that unless the correctness of a computer program is proven with full mathematical rigor, the program cannot be considered correct.
- As a practical matter, the validity of programs is still established by testing.
  - Testing of computer programs is an art rather than a science, but that does not mean that there is nothing in it to learn.
  - Test and debug your program thoroughly whenever you implement an algorithm.
  - Provide verifications of the algorithm’s input
    - W assume that inputs to algorithms belong to the specified sets and hence require no verification. When implementing algorithms as programs to be used in actual applications, you should provide such verifications.

Coding an Algorithm (cont.)

- Of course, implementing an algorithm correctly is necessary but not sufficient: you would not like to diminish your algorithm’s power by an inefficient implementation.
  - Modern compilers do provide a certain safety net in this regard, especially when they are used in their code optimization mode. Still, you need to be aware of such standard tricks as: computing a loop’s invariant (an expression that does not change its value) outside the loop, collecting common subexpressions, replacing expensive operations by cheap ones, and so on.

Important Problem Types

- There are a few areas that have attracted particular attention from researchers:
  - Sorting
  - Searching
  - String processing
  - Graph problems
  - Combinatorial problems
  - Geometric problems
  - Numerical problems

- Their interest has been driven either by the problem’s practical importance or by some specific characteristics making the problem an interesting research subject;

Sorting

- The **sorting problem** is to rearrange the items of a given list in nondecreasing order.
- Of course, for this problem to be meaningful, the nature of the list items must allow such an ordering. (Mathematicians would say that there must exist a relation of total ordering.)
- As a practical matter, we usually need to sort lists of numbers, characters from an alphabet, character strings, and, most important, records similar to those maintained by schools about their students, libraries about their holdings, and companies about their employees.

Sorting (cont.)

- In the case of records, we need to choose a piece of information to guide sorting.
- For example, we can choose to sort student records in alphabetical order of names or by student number or by student grade-point average.
- Such a specially chosen piece of information is called a key.
- Computer scientists often talk about sorting a list of keys even when the list's items are not records but, say, just integers.

Sorting (cont.)

- Why would we want a sorted list?
  - Can be a required output of a task (example: Ranking students by their GPA scores)
  - Makes many questions about the list easier to answer (example: searching in dictionaries, class list, etc.)
  - Used as an auxiliary step in several important algorithms (example: The Greedy approach requires a sorted input)

Sorting (cont.)

- Although some sorting algorithms are indeed better than others, there is no algorithm that would be the best solution in all situations.
  - Some of the algorithms are simple but relatively slow, while others are faster but more complex;
  - Some work better on randomly ordered inputs, while others do better on almost-sorted lists;
  - Some are suitable only for lists residing in the fast memory, while others can be adapted for sorting large files stored on a disk;

Sorting (cont.)

- Two properties of sorting algorithms deserve special mention: stability and in-place operation.

Sorting (cont.)

- A sorting algorithm is called **stable** if it preserves the relative order of any two equal elements in its input.
    - In other words, if an input list contains two equal elements in positions i and j where i < j, then in the sorted list they have to be in positions i′ and j′, respectively, such that i′ < j′.
    - This property can be desirable if, for example, we have a list of students sorted alphabetically and we want to sort it according to student GPA: a stable algorithm will yield a list in which students with the same GPA will still be sorted alphabetically.
  - Algorithms that can exchange keys located far apart are not stable, but they usually work faster;

Sorting (cont.)

- The second notable feature of a sorting algorithm is the amount of extra memory the algorithm requires.
  - An algorithm is said to be **in-place** if it does not require extra memory, except, possibly, for a few memory units. There are important sorting algorithms that are in-place and those that are not.

Searching

- The searching problem deals with finding a given value, called a search key, in a given set (or a multiset, which permits several elements to have the same value).
- There are plenty of searching algorithms to choose from.
- They range from the straightforward sequential search to a spectacularly efficient but limited binary search and algorithms based on representing the underlying set in a different form more conducive to searching.
- The latter algorithms are of particular importance for real-world applications because they are indispensable for storing and retrieving information from large databases.

String Processing

- A string is a sequence of characters from an alphabet.
- Strings of particular interest are:
  - **Text strings**, which comprise letters, numbers, and special characters;
  - **Bit strings**, which comprise zeros and ones;
  - and **gene sequences**, which can be modeled by strings of characters from the four-character alphabet {A, C, G, T}.
- **String-processing algorithms** have been important for computer science for a long time in conjunction with computer languages and compiling issues.
  - One particular problem—that of searching for a given word in a text—has attracted special attention from researchers. They call it **string matching**.

Graph Problems

- A graph can be thought of as a collection of points called vertices, some of which are connected by line segments called edges.
- Graphs can be used for modeling a wide variety of applications including:
  - Transportation
  - Communication
  - Social and economic networks
  - Project scheduling
  - Games

Graph Problems (cont.)

- Some graph problems are computationally very hard; the most well-known examples are the traveling salesman problem and the graph-coloring problem.
- **Traveling salesman problem (TSP)** is the problem of finding the shortest tour through n cities that visits every city exactly once.
  - This problem arises in several applications, such as route planning,
- **Graph-coloring problem** seeks to assign the smallest number of colors to the vertices of a graph so that no two adjacent vertices are the same color.
  - This problem arises in several applications, such as event scheduling

Combinatorial Problems

- Combinatorial problems. These are problems that ask, explicitly or implicitly, to find a combinatorial object—such as a permutation, a combination, or a subset—that satisfies certain constraints.

Combinatorial Problems (cont.)

- Combinatorial problems are the most difficult problems in computing, from both a theoretical and practical standpoint.
- Their difficulty stems from the following facts:
  - The number of combinatorial objects typically grows extremely fast with a problem's size, reaching unimaginable magnitudes even for moderate-sized instances.
  - There are no known algorithms for solving most such problems exactly in an acceptable amount of time. Moreover, most computer scientists believe that such algorithms do not exist.

Combinatorial Problems (cont.)

- Some combinatorial problems can be solved by efficient algorithms, but they should be considered fortunate exceptions to the rule.
  - The shortest-path problem is among such exceptions.

Geometric Problems

- **Geometric algorithms** deal with geometric objects such as points, lines, and polygons.
- The ancient Greeks were very much interested in developing procedures for solving a variety of geometric problems, including problems of constructing simple geometric shapes—triangles, circles, and so on.
- We will discuss algorithms for only two classic problems of computational geometry:
  - The closest-pair problem (given n points in the plane, find the closest pair among them.)
  - The convex-hull problem (asks to find the smallest convex polygon that would include all the points of a given set)

Numerical Problems

- **Numerical problems** are problems that involve mathematical objects of continuous nature: solving equations and systems of equations, evaluating functions, and so on.

Fundamental Data Structures

- Since the vast majority of algorithms of interest operate on data, particular ways of organizing data play a critical role in the design and analysis of algorithms.
- A data structure can be defined as a particular scheme of organizing related data items.
- The nature of the data items is dictated by the problem at hand; they can range from elementary data types (e.g., integers or characters) to data structures (e.g., a one-dimensional array of one-dimensional arrays is often used for implementing matrices).
- There are a few data structures that have proved to be particularly important for computer algorithms.

Linear Data Structures

- The two most important elementary data structures are:
  - The array
  - The linked list.

Arrays

- A **(one-dimensional) array** is a sequence of n items of the same data type that are stored contiguously in computer memory and made accessible by specifying a value of the array's index

FIGURE 1.3 Array of n elements.

Arrays (cont.)

- In the majority of cases, the index is an integer either between 0 and n − 1 or between 1 and n.
- Some computer languages allow an array index to range between any two integer bounds low and high, and some even permit nonnumerical indices to specify, for example, data items corresponding to the 12 months of the year by the month names.
- Each and every element of an array can be accessed in the same constant amount of time regardless of where in the array the element in question is located. This feature positively distinguishes arrays from linked lists.

Arrays (cont.)

- Arrays are used for implementing a variety of other data structures.
- Prominent among them is the **string**, a sequence of characters from an alphabet terminated by a special character indicating the string's end.
- Strings composed of zeros and ones are called binary strings or bit strings.
- Strings are indispensable for processing textual data, defining computer languages and compiling programs written in them, and studying abstract computational models.

Arrays (cont.)

- Operations we usually perform on strings differ from those we typically perform on other arrays (say, arrays of numbers). They include:
  - Computing the string length,
  - Comparing two strings to determine which one precedes the other in lexicographic (i.e., alphabetical) order,
  - Concatenating two strings (forming one string from two given strings by appending the second to the end of the first).

Linked List

- A linked list is a sequence of zero or more elements called nodes, each containing two kinds of information: some data and one or more links called pointers to other nodes of the linked list.
- (A special pointer called “null” is used to indicate the absence of a node’s successor.)
- In a singly linked list, each node except the last one contains a single pointer to the next element.

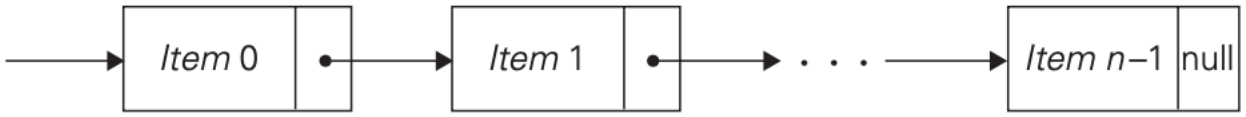


FIGURE 1.4 Singly linked list of n elements.

Linked List (cont.)

- To access a particular node of a linked list, one starts with the list's first node and traverses the pointer chain until the particular node is reached.
- Thus, the time needed to access an element of a singly linked list, unlike that of an array, depends on where in the list the element is located.
- On the positive side, linked lists do not require any preliminary reservation of the computer memory, and insertions and deletions can be made quite efficiently in a linked list by reconnecting a few appropriate pointers.

Linked List (cont.)

- We can exploit flexibility of the linked list structure in a variety of ways.
  - For example, it is often convenient to start a linked list with a special node called the header.
    - This node may contain information about the linked list itself, such as its current length; it may also contain, in addition to a pointer to the first element, a pointer to the linked list’s last element.

Linked List (cont.)

- Another extension is the structure called the doubly linked list , in which every node, except the first and the last, contains pointers to both its successor and its predecessor

Linked List (cont.)

- The array and linked list are two principal choices in representing a more abstract data structure called a **linear list** or simply a **list**.
- A **list** is a **finite sequence of data items**, i.e., a **collection of data items arranged in a certain linear order.**
- The basic operations performed on this data structure are searching for, inserting, and deleting an element.
- Two special types of lists, stacks and queues, are particularly important.

Stack

- A stack is a list in which insertions and deletions can be done only at the end.
- This end is called the **top** because a stack is usually visualized not horizontally but vertically—akin to a stack of plates whose “operations” it mimics very closely.
- As a result, when elements are added to (pushed onto) a stack and deleted from (popped off) it,
- The structure operates in a “last-in–first-out” (LIFO) fashion— exactly like a stack of plates if we can add or remove a plate only from the top.
- Stacks have a multitude of applications; in particular, they are indispensable for implementing recursive algorithms.

Queue

- A queue, on the other hand, is a list from which elements are deleted from one end of the structure, called the front (this operation is called dequeue), and new elements are added to the other end, called the rear (this operation is called enqueue).
- Consequently, a queue operates in a “first-in–first-out” (FIFO) fashion—akin to a queue of customers served by a single teller in a bank.
- Queues also have many important applications, including several algorithms for graph problems.

Queue (cont.)

- Many important applications require selection of an item of the highest priority among a dynamically changing set of candidates.
  - A data structure that seeks to satisfy the needs of such applications is called a **priority queue**.
- A **priority queue** is a collection of data items from a totally ordered universe (most often, integer or real numbers).
- The principal operations on a priority queue are finding its largest element, deleting its largest element, and adding a new element.
- Straightforward implementations of this data structure can be based on either an array or a sorted array, but neither of these options yields the most efficient solution possible.
- A better implementation of a priority queue is based on an ingenious data structure called the heap.

Graphs

- Graph is informally thought of as a collection of points in the plane called “vertices” or “nodes,” some of them connected by line segments called “edges” or “arcs.”
- Formally, a graph $G = \langle V , E \rangle$ is defined by a pair of two sets: a finite nonempty set V of items called vertices and a set E of pairs of these items called edges.
- If these pairs of vertices are unordered, i.e., a pair of vertices (u, v) is the same as the pair (v, u), we say that the vertices u and v are adjacent to each other and that they are connected by the undirected edge (u, v).
- We call the vertices u and v endpoints of the edge (u, v) and say that u and v are incident to this edge;
- Ww also say that the edge (u, v) is incident to its endpoints u and v.
- A graph G is called undirected if every edge in it is undirected.

Graphs (cont.)

- If a pair of vertices (u, v) is not the same as the pair (v, u), we say that the edge (u, v) is directed from the vertex u, called the edge's tail, to the vertex v, called the edge's head.
- We also say that the edge (u, v) leaves u and enters v.
- A graph whose every edge is directed is called directed.
- Directed graphs are also called digraphs.

Graphs (cont.)

- It is normally convenient to label vertices of a graph or a digraph with letters, integer numbers, or, if an application calls for it, character strings (Figure 1.6).
- The graph depicted in Figure 1.6a has six vertices and seven undirected edges:

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

- The digraph depicted in Figure 1.6b has six vertices and eight directed edges:

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

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

Graphs (cont.)

- Our definition of a graph does not forbid loops, or edges connecting vertices to themselves. Unless explicitly stated otherwise,
- We will consider graphs without loops. Since our definition disallows multiple edges between the same vertices of an undirected graph,
- We have the following inequality for the number of edges |E| possible in an undirected graph with |V | vertices and no loops:

$$0 \leq |E| \leq |V|(|V| - 1)/2.$$

- (We get the largest number of edges in a graph if there is an edge connecting each of its |V | vertices with all |V | − 1 other vertices.
- We have to divide product |V |(|V | − 1) by 2, however, because it includes every edge twice.)

Graphs (cont.)

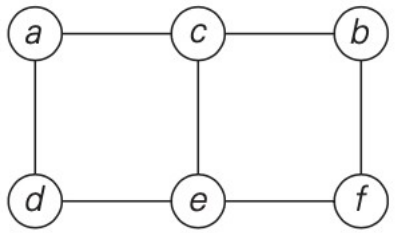
- A graph with every pair of its vertices connected by an edge is called **complete**.
- A graph with relatively few possible edges missing is called **dense**;
- a graph with few edges relative to the number of its vertices is called **sparse**.
- Whether we are dealing with a dense or sparse graph may influence how we choose to represent the graph and, consequently, the running time of an algorithm being designed or used.

Graph Representations

- Graphs for computer algorithms are usually represented in one of two ways:
  - Adjacency matrix
  - Adjacency lists.

Graph Representations (cont.)

- The adjacency matrix of a graph with n vertices is an n × n Boolean matrix with one row and one column for each of the graph's vertices, in which the element in the i_{th} row and the j_{th} column is equal to 1 if there is an edge from the i_{th} vertex to the j_{th} vertex, and equal to 0 if there is no such edge.
- Note that the adjacency matrix of an undirected graph is always symmetric, i.e., A[i, j] = A[j, i] for every 0 ≤ i, j ≤ n − 1

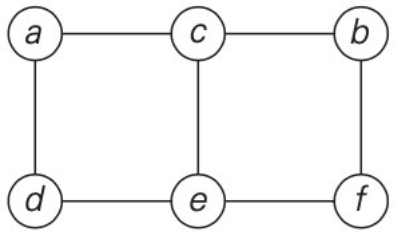


	a	b	c	d	e	f
a	0	0	1	1	0	0
b	0	0	1	0	0	1
c	1	1	0	0	1	0
d	1	0	0	0	1	0
e	0	0	1	1	0	1
f	0	1	0	0	1	0

Adjacency matrix

Graph Representations (cont.)

- The adjacency lists of a graph or a digraph is a collection of linked lists, one for each vertex, that contain all the vertices adjacent to the list's vertex (i.e., all the vertices connected to it by an edge).
- Usually, such lists start with a header identifying a vertex for which the list is compiled. For example,
- To put it another way, adjacency lists indicate columns of the adjacency matrix that, for a given vertex, contain 1's.



a	→	c	→	d		
b	→	c	→	f		
c	→	a	→	b	→	e
d	→	a	→	e		
e	→	c	→	d	→	f
f	→	b	→	e		

adjacency lists

Graph Representations (cont.)

- If a graph is sparse, the adjacency list representation may use less space than the corresponding adjacency matrix despite the extra storage consumed by pointers of the linked lists; the situation is exactly opposite for dense graphs.
- In general, which of the two representations is more convenient depends on the nature of the problem, on the algorithm used for solving it, and, possibly, on the type of input graph (sparse or dense).

Weighted Graphs

- A weighted graph (or weighted digraph) is a graph (or digraph) with numbers assigned to its edges.
- These numbers are called weights or costs.
- An interest in such graphs is motivated by numerous real-world applications, such as finding the shortest path between two points in a transportation or communication network or the traveling salesman problem mentioned earlier.

Weighted Graphs (cont.)

- Both principal representations of a graph can be easily adopted to accommodate weighted graphs.
- If a weighted graph is represented by its adjacency matrix, then its element A[i, j] will simply contain the weight of the edge from the i_{th} to the j_{th} vertex if there is such an edge and a special symbol, e.g., ∞, if there is no such edge.
  - Such a matrix is called the weight matrix or cost matrix.
- For some applications, it is more convenient to put 0's on the main diagonal of the adjacency matrix.) Adjacency lists for a weighted graph have to include in their nodes not only the name of an adjacent vertex but also the weight of the correspc

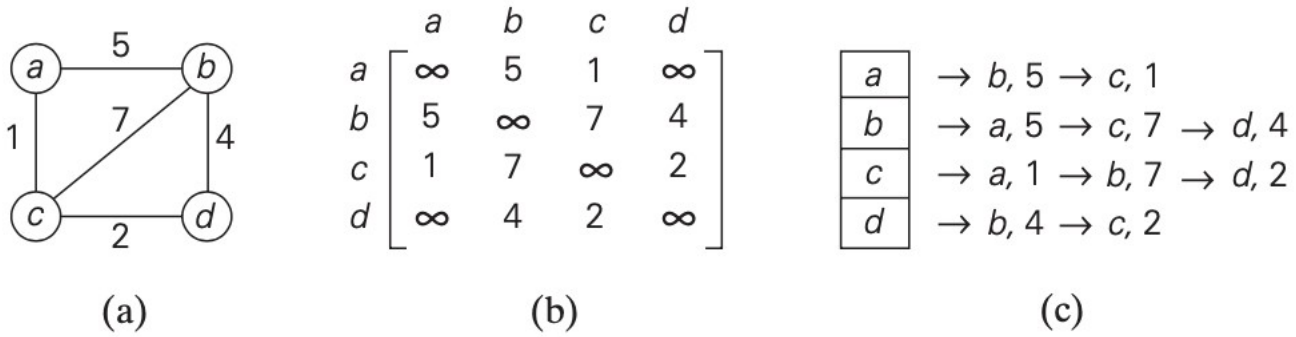
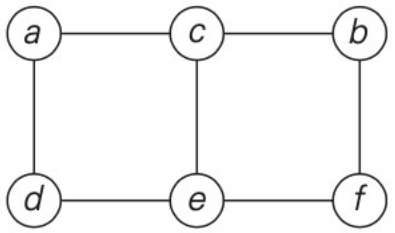


FIGURE 1.8 (a) Weighted graph. (b) Its weight matrix. (c) Its adjacency lists.

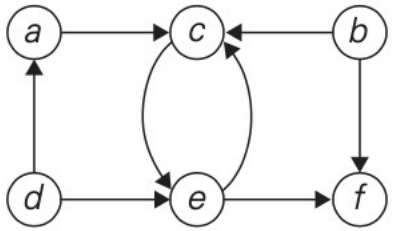
Paths

- Among the many properties of graphs, two are important for a great number of applications: **connectivity** and **acyclicity**. Both are based on the notion of a path.
- A path from vertex u to vertex v of a graph G can be defined as a sequence of adjacent (connected by an edge) vertices that starts with u and ends with v.
- If all vertices of a path are distinct, the path is said to be simple.
- The length of a path is the total number of vertices in the vertex sequence defining the path minus 1, which is the same as the number of edges in the path.
- For example, a, c, b, f is a simple path of length 3 from a to f in the graph in Figure 1.6a, whereas a, c, e, c, b, f is a path(not simple)of length 5 from a to f.



Paths

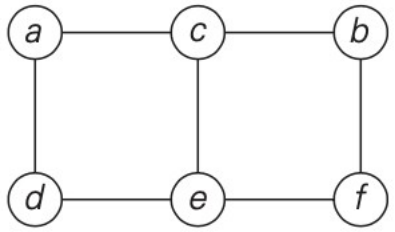
- In the case of a **directed graph**, we are usually interested in **directed paths**.
- A directed path is a sequence of vertices in which every consecutive pair of the vertices is connected by an edge directed from the vertex listed first to the vertex listed next.
- For example, a, c, e, f is a directed path from a to f in the graph in Figure 1.6b.



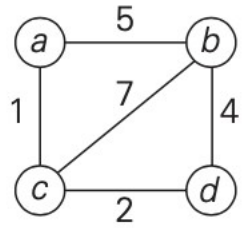
(b)

Connected Graph

- **A graph is said to be connected if for every pair of its vertices u and v there is a path from u to v.**
- If a graph is not connected, such a model will consist of several connected pieces that are called connected components of the graph.
- Formally, a connected component is a maximal (not expandable by including another vertex and an edge) connected subgraph of a given graph.



Connected Graph



Connected Graph

Connected Graph (Cont.)

- Whereas the graph in Figure 1.9 is not, because there is no path, for example, from a to f. The graph in Figure 1.9 has two connected components with vertices {a, b, c, d, e} and {f, g, h, i}, respectively.

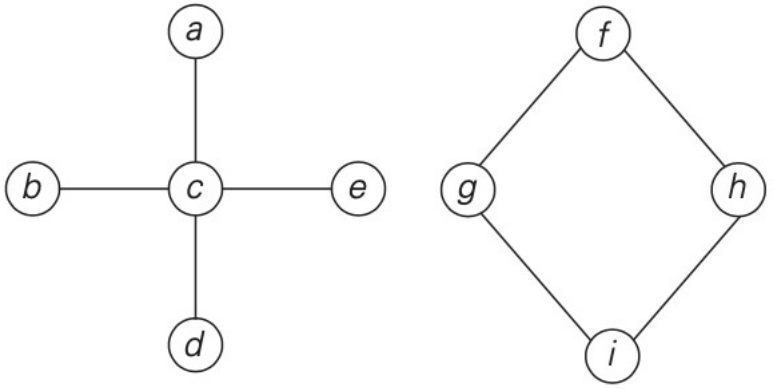


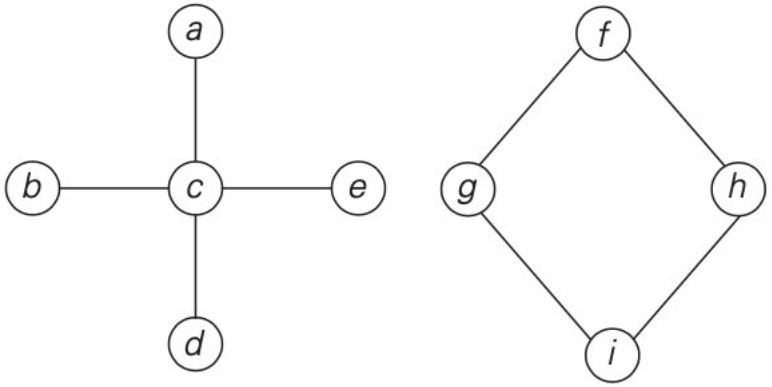
FIGURE 1.9 Graph that is not connected.

Connected Graph (Cont.)

- Graphs with several connected components do happen in real-world applications.
  - A graph representing the Interstate highway system of the United States would be an example

Cycles

- It is important to know for many applications whether or not a graph under consideration has cycles.
- A cycle is a path of a positive length that starts and ends at the same vertex and does not traverse the same edge more than once.
- For example, f, h, i, g, f is a cycle in the graph in Figure 1.9.
- A graph with no cycles is said to be acyclic.



Trees

- A tree (more accurately, a free tree) is a connected acyclic graph (Figure 1.10a).
- A graph that has no cycles but is not necessarily connected is called a forest: each of its connected components is a tree (Figure 1.10b).

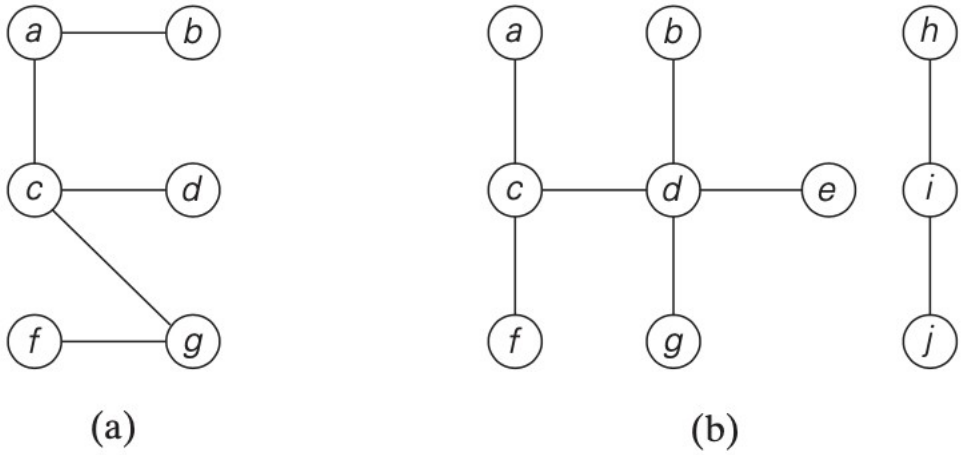


FIGURE 1.10 (a) Tree. (b) Forest.

Trees (cont.)

- Trees have several important properties other graphs do not have.
- Number of edges in a tree is always one less than the number of its vertices:

$$|E| = |V| - 1.$$

Trees (cont.)

- **Rooted Trees is another very important property of trees is the fact that for every two vertices in a tree, there always exists exactly one simple path from one of these vertices to the other.**
- **This property makes it possible to select an arbitrary vertex in a free tree and consider it as the root of the so-called rooted tree.**
- A **rooted tree** is usually depicted by placing its root on the top (level 0 of the tree), the vertices adjacent to the root below it (level 1), the vertices two edges apart from the root still below (level 2), and so on.
- Figure 1.11 presents such a transformation from a free tree to a rooted tree.

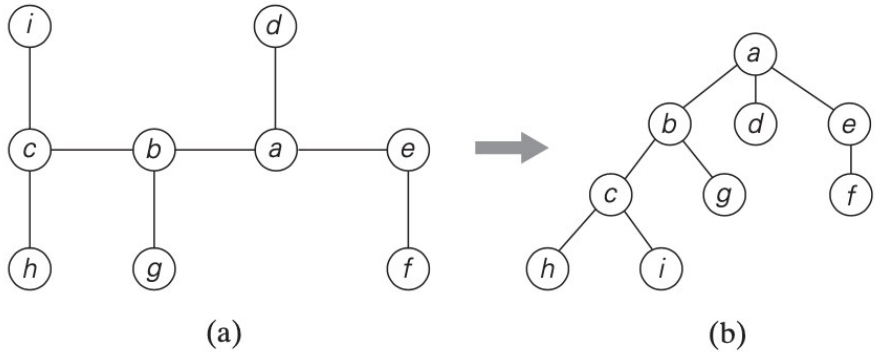


FIGURE 1.11 (a) Free tree. (b) Its transformation into a rooted tree.

Trees (cont.)

- **Rooted trees** play a very important role in computer science, a much more important one than free trees do; in fact, for the sake of brevity, they are often referred to as simply “trees.”
- An obvious application of trees is for describing hierarchies, from file directories to organizational charts of enterprises.

Trees (cont.)

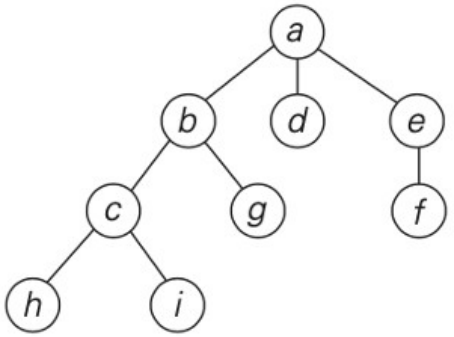
- For any vertex v in a tree T , all the vertices on the simple path from the root to that vertex are called ancestors of v.
- The vertex itself is usually considered its own ancestor;
- The set of ancestors that excludes the vertex itself is referred to as the set of proper ancestors.
- If (u, v) is the last edge of the simple path from the root to vertex v (and u ≠ v), u is said to be the parent of v and v is called a child of u;
- **vertices** that have the **same parent** are said to be **siblings**.

Trees (cont.)

- A vertex with no children is called a **leaf**
- a **vertex with at least one child** is called **parental**.
- **All the vertices for which a vertex v is an ancestor are said to be descendants of v; the proper descendants exclude the vertex v itself.**
- All the descendants of a vertex v with all the edges connecting them form the subtree of T rooted at that vertex.

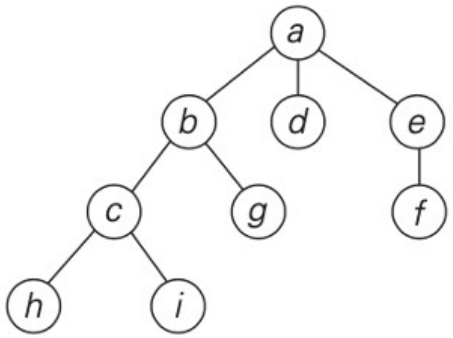
Trees (cont.)

- The root of the tree is a;
- Vertices d, g, f, h, and i are leaves, vertices a, b, e, and c are parental;
- The parent of b is a;
- The children of b are c and g;
- The siblings of b are d and e;
- the vertices of the subtree rooted at b are {b, c, g, h, i}.



Trees (cont.)

- The depth of a vertex v is the length of the simple path from the root to v.
- The height of a tree is the length of the longest simple path from the root to a leaf.
- For example, the depth of vertex c of the tree in Figure 1.11b is 2,
- and the height of the tree is 3.
- Thus, if we count tree levels top down starting with 0 for the root's level, the depth of a vertex is simply its level in the tree, and the tree's height is the maximum level of its vertices.



Ordered Trees

- An **ordered tree** is a rooted tree in which all the children of each vertex are ordered. It is convenient to assume that in a tree's diagram, all the children are ordered left to right.

Binary Tree

- A **binary tree** can be defined as an ordered tree in which every vertex has no more than two children and each child is designated as either a left child or a right child of its parent; a
- binary tree may also be empty.
- An example of a binary tree is given in Figure 1.12a.

Binary Tree (cont.)

- The binary tree with its root at the left (right) child of a vertex in a binary tree is called the left (right) subtree of that vertex.
- Since left and right subtrees are binary trees as well, a binary tree can also be defined recursively.
- This makes it possible to solve many problems involving binary trees by recursive algorithms.

Binary Tree (cont.)

- In Figure 1.12b, some numbers are assigned to vertices of the binary tree in Figure 1.12a.

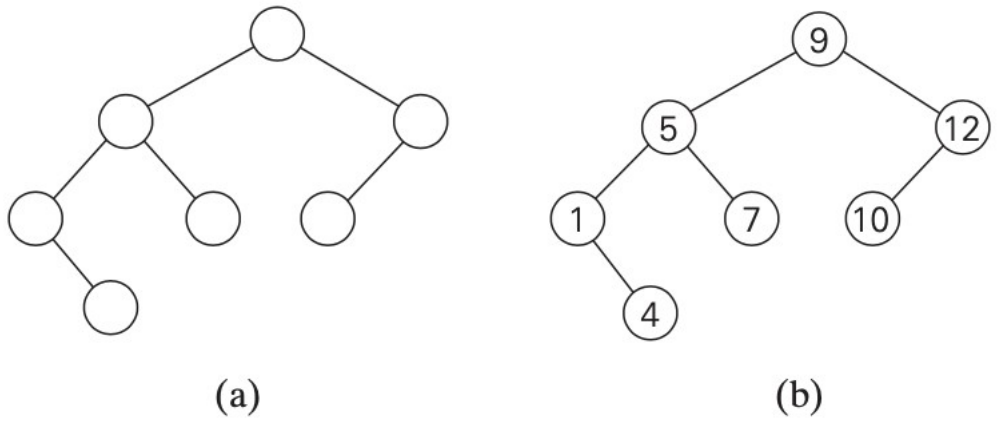


FIGURE 1.12 (a) Binary tree. (b) Binary search tree.

Binary Tree (cont.)

- Note that a number assigned to each parental vertex is larger than all the numbers in its left subtree and smaller than all the numbers in its right subtree. Such trees are called **binary search trees.**
- Binary trees and binary search trees have a wide variety of applications in computer science
- In particular, binary search trees can be generalized to more general types of search trees called multiway search trees, which are indispensable for efficient access to very large data sets.

Binary Tree (cont.)

- The efficiency of most important algorithms for binary search trees and their extensions depends on the tree's height.
- Therefore, the following inequalities for the height h of a binary tree with n nodes are especially important for analysis of such algorithms:

$$\lfloor \log_2 n \rfloor \leq h \leq n - 1.$$

Binary Tree (cont.)

- A **binary tree** is usually implemented for computing purposes by a collection of nodes corresponding to vertices of the tree.
  - Each node contains some information associated with the vertex (its name or some value assigned to it) and two pointers to the nodes representing the left child and right child of the vertex, respectively.
  - Figure 1.13 illustrates such an implementation for the binary search tree in Figure 1.12b.

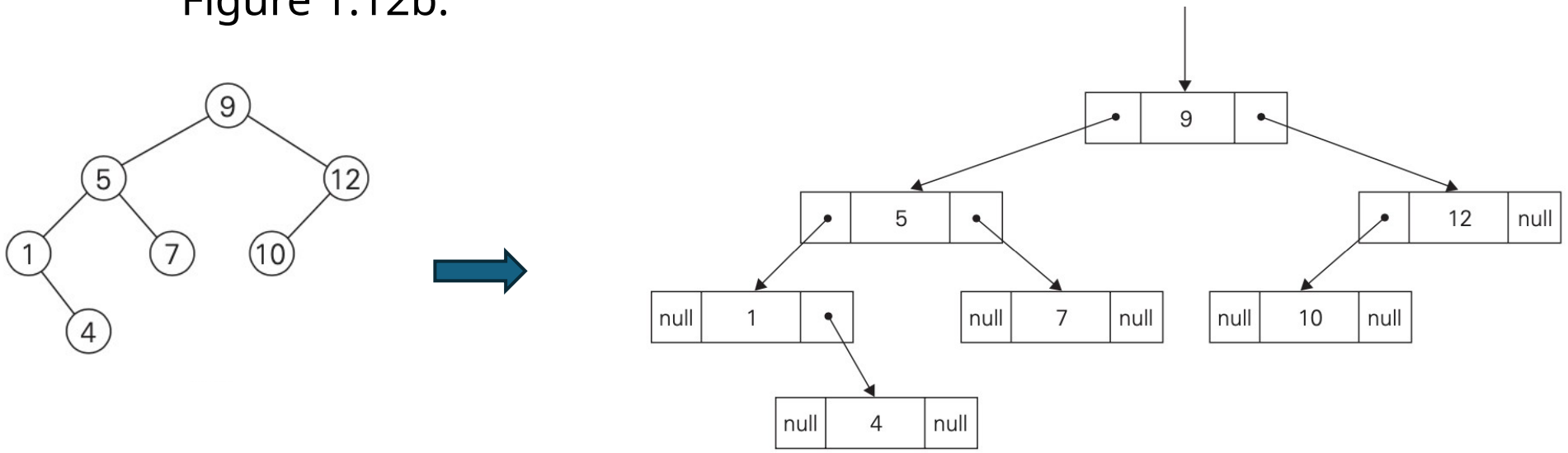


FIGURE 1.13 Standard implementation of the binary search tree in Figure 1.12b.

Binary Tree (cont.)

- A computer representation of an **arbitrary ordered tree** can be done by simply providing a parental vertex with the number of pointers equal to the number of its children.
  - This representation may prove to be inconvenient if the number of children varies widely among the nodes.
- We can avoid this inconvenience by using nodes with just two pointers, as we did for binary trees. Here, however, the left pointer will point to the first child of the vertex, and the right pointer will point to its next sibling.
  - Accordingly, this representation is called the **first child–next sibling representation**. Thus, all the siblings of a vertex are linked via the nodes' right pointers in a singly linked list, with the first element of the list pointed to by the left pointer of their parent.

Binary Tree (cont.)

- Figure 1.14a illustrates this representation for the tree in Figure 1.11b.
- It is not difficult to see that this representation effectively transforms an ordered tree into a binary tree said to be associated with the ordered tree. We get this representation by “rotating” the pointers about 45 degrees clockwise (see Figure 1.14b).

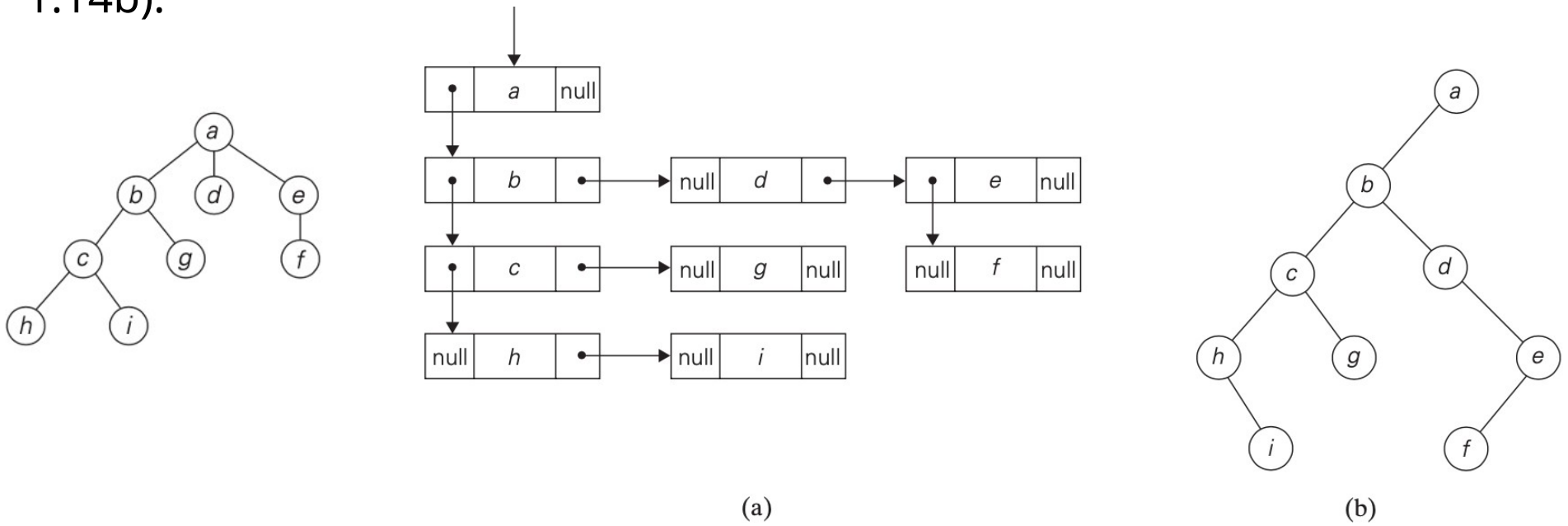


FIGURE 1.14 (a) First child–next sibling representation of the tree in Figure 1.11b. (b) Its binary tree

Sets

- The notion of a set plays a central role in mathematics.
- **A set can be described as an unordered collection (possibly empty) of distinct items called elements of the set.**
- A specific set is defined either by an explicit listing of its elements (e.g., S = {2, 3, 5, 7}) or by specifying a property that all the set's elements and only they must satisfy (e.g., S = {n: n is a prime number smaller than 10}).

Sets (cont.)

- The most important set operations are:
  - Checking membership of a given item in a given set;
  - Finding the union of two sets, which comprises all the elements in either or both of them;
  - Finding the intersection of two sets, which comprises all the common elements in the sets.

Sets (cont.)

- Sets can be implemented in computer applications in two ways.
  - The first considers only sets that are subsets of some large set U, called the universal set.
  - If set U has n elements, then any subset S of U can be represented by a bit string of size n, called a bit vector, in which the i_{th} element is 1 if and only if the ith element of U is included in set S.
  - Thus, to continue with our example, if U = {1, 2, 3, 4, 5, 6, 7, 8, 9}, then S = {2, 3, 5, 7} is represented by the bit string 011010100.
  - This way of representing sets makes it possible to implement the standard set operations very fast, but at the expense of potentially using a large amount of storage.

Sets (cont.)

- The second and more common way to represent a set for computing purposes is to use the list structure to indicate the set’s elements.
- Of course, this option, too, is feasible only for finite sets; fortunately, unlike mathematics, this is the kind of sets most computer applications need.

Sets (cont.)

- The two principal points of distinction between sets and lists.
  - **a set cannot contain identical elements; a list can.**
    - This requirement for uniqueness is sometimes circumvented by the introduction of a **multiset**, or **bag**, an unordered collection of items that are not necessarily distinct.
  - a set is an unordered collection of items; therefore, changing the order of its elements does not change the set. A list, defined as an ordered collection of items, is exactly the opposite.
    - This is an important theoretical distinction, but fortunately it is not important for many applications.
    - It is also worth mentioning that if a set is represented by a list, depending on the application at hand, it might be worth maintaining the list in a sorted order.

Sets (cont.)

- In computing, the operations we need to perform for a set or a multiset most often are searching for a given item, adding a new item, and deleting an item from the collection.
- A data structure that implements these three operations is called the dictionary.
- There are quite a few ways a dictionary can be implemented.
- They range from an unsophisticated use of arrays (sorted or not) to much more sophisticated techniques such as hashing and balanced search trees