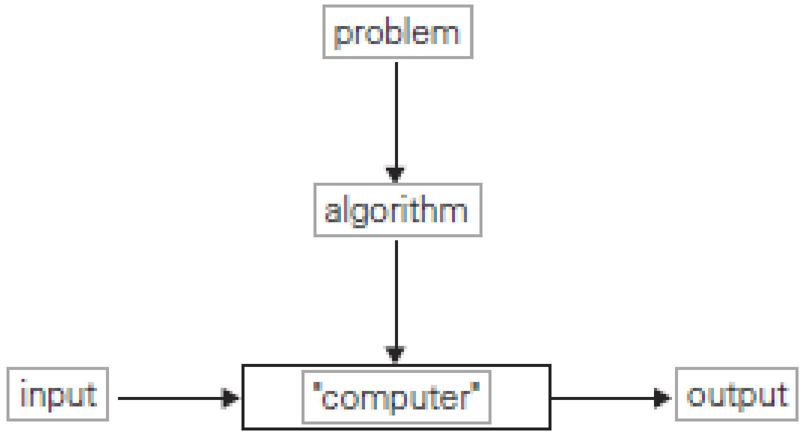
1-INTRODUCTION

1.1 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.
  - The nonambiguity requirement for each step of an algorithm cannot be com-promised.
  - 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.



1.2 Fundamentals of Algorithmic Problem Solving

- Understanding the Problem
- From a practical perspective, 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.

1.2 Fundamentals of Algorithmic Problem Solving

- Algorithm Design Techniques
- 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.

1.2 Fundamentals of Algorithmic Problem Solving

- Methods of Specifying an Algorithm
- 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.
- In the earlier days of computing, the dominant vehicle for specifying algo-rithms was a flowchart, a method of expressing an algorithm by a collection of connected geometric shapes containing descriptions of the algorithm’s steps.

1.2 Fundamentals of Algorithmic Problem Solving

- Proving an Algorithm’s Correctness
- Once an algorithm has been specified, you have to prove its correctness.
- That is, you have to 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.

1.2 Fundamentals of Algorithmic Problem Solving

- Proving an Algorithm’s Correctness
- For some algorithms, a proof of correctness is quite easy; for others, it can be quite complex.
- A common technique for proving correctness is to use mathemati-cal induction because an algorithm’s iterations provide a natural sequence of steps needed for such proofs.
- It might be worth mentioning that although tracing the algorithm’s performance for a few specific inputs can be a very worthwhile activ-ity, it cannot prove the algorithm’s correctness conclusively.
- But in order to show that an algorithm is incorrect, you need just one instance of its input for which the algorithm fails.

1.3 Important Problem Types

- In the limitless sea of problems one encounters in computing, there are a few areas that have attracted particular attention from researchers.
- By and large, their interest has been driven either by the problem’s practical importance or by some specific characteristics making the problem an interesting research subject; fortunately, these two motivating forces reinforce each other in most cases.

1.3 Important Problem Types

- Sorting
- Searching
- String processing
- Graph problems
- Combinatorial problems
- Geometric problems
- Numerical problems

1.3 Important Problem Types - 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.
- 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.

1.3 Important Problem Types - 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.

1.3 Important Problem Types - Searching

- For searching, too, there is no single algorithm that fits all situations best.
- Some algorithms work faster than others but require more memory; some are very fast but applicable only to sorted arrays; and so on.

1.3 Important Problem Types - 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}.
- It should be pointed out, however, that string-processing algorithms have been important for computer science for a long time in conjunction with computer languages and compiling issues.

1.3 Important Problem Types - Graph Problems

- One of the oldest and most interesting areas in algorithmics is graph algorithms.
- Informally, a graph can be thought of as a collection of points called vertices, some of which are connected by line segments called edges.
- Graphs are an interesting subject to study, for both theoretical and practical reasons.
- Graphs can be used for modeling a wide variety of applications, including transportation, communication, social and economic networks, project scheduling, and games.
- Studying different technical and social aspects of the Internet in particular is one of the active areas of current research involving computer scientists, economists, and social scientists.

1.3 Important Problem Types - Graph Problems

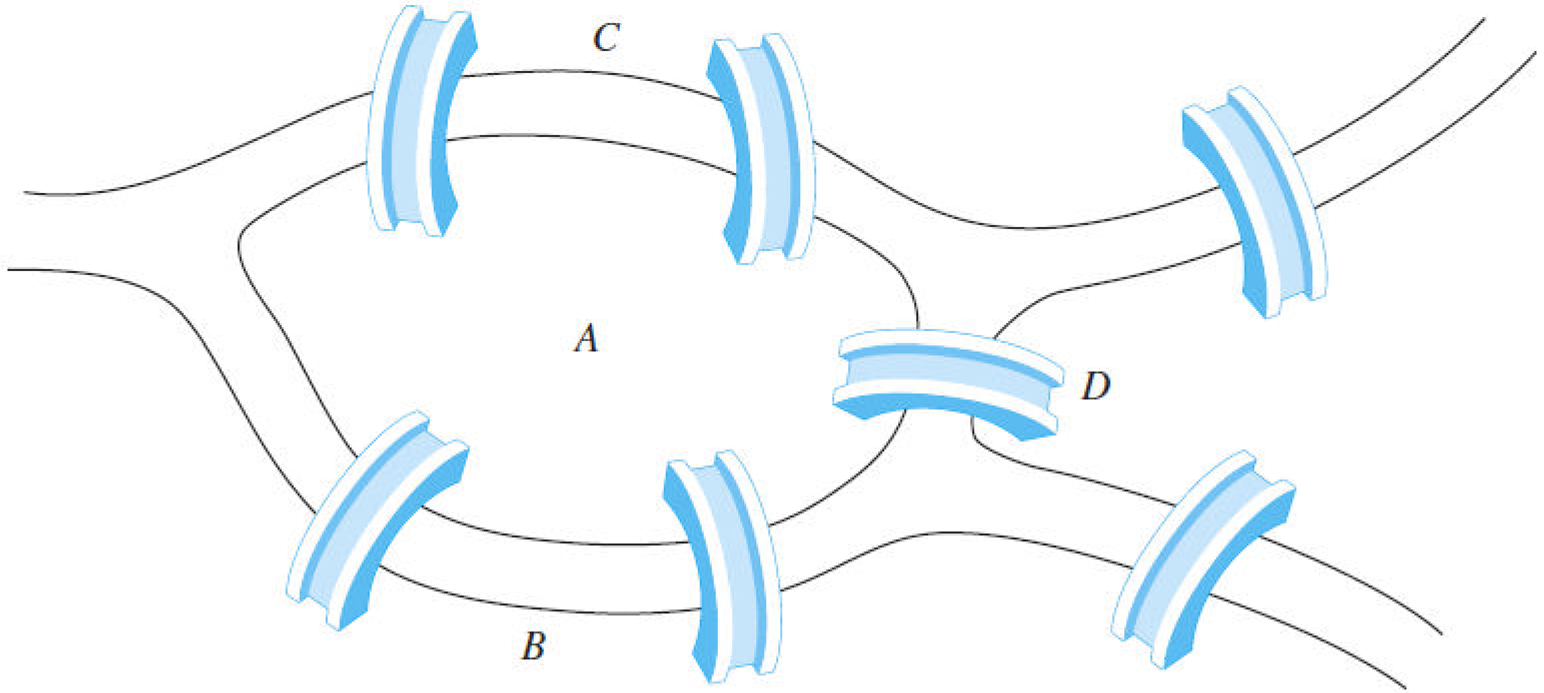
- Basic graph algorithms include graph-traversal algorithms (how can one reach all the points in a network?), shortest-path algorithms (what is the best route be-tween two cities?), and topological sorting for graphs with directed edges (is a set of courses with their prerequisites consistent or self-contradictory?).

- Fortunately, these algorithms can be considered illustrations of general design techniques; accordingly, you will find them in corresponding chapters of the book.

1.3 Important Problem Types - Graph Problems

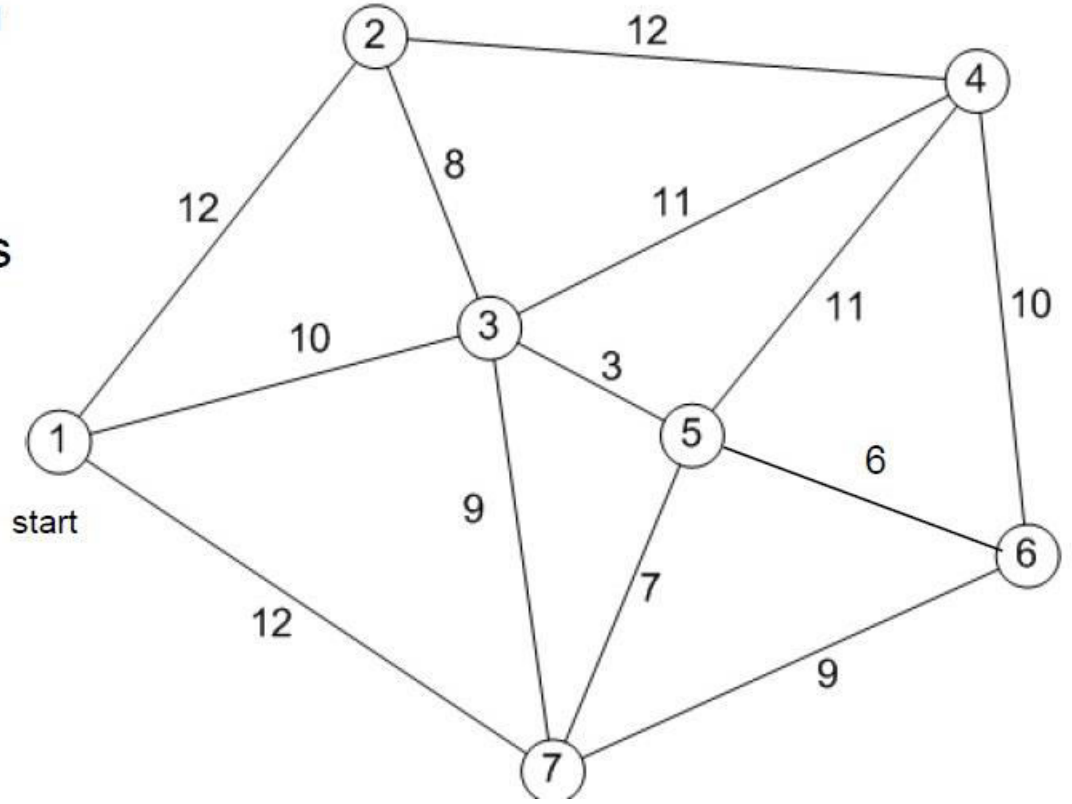
- Some graph problems are computationally very hard; the most well-known examples are the traveling salesman problem and the graph-coloring problem.
- The traveling salesman problem (TSP) is the problem of finding the shortest tour through n cities that visits every city exactly once.
- The graph-coloring 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: if the events are represented by vertices that are connected by an edge if and only if the corresponding events cannot be scheduled at the same time, a solution to the graph-coloring problem yields an optimal schedule.

Königsberg Bridges

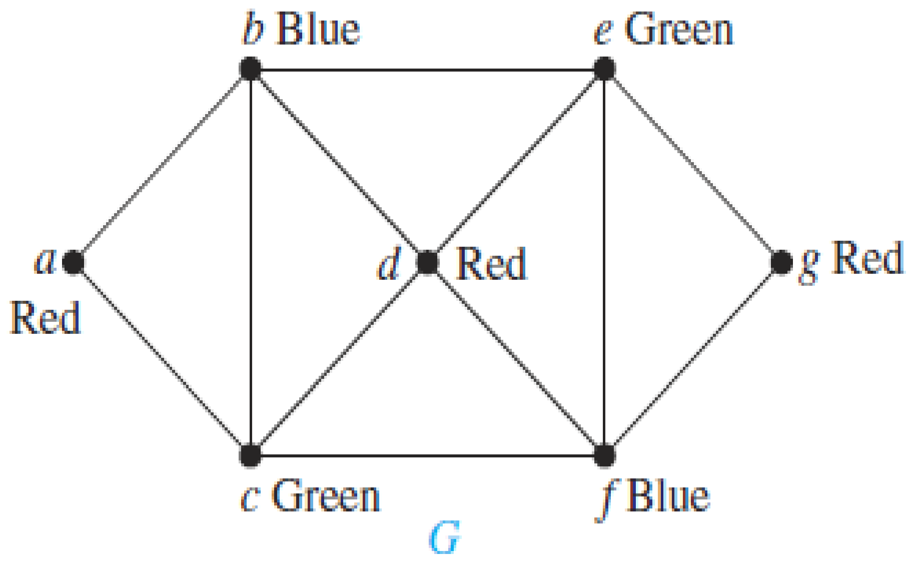


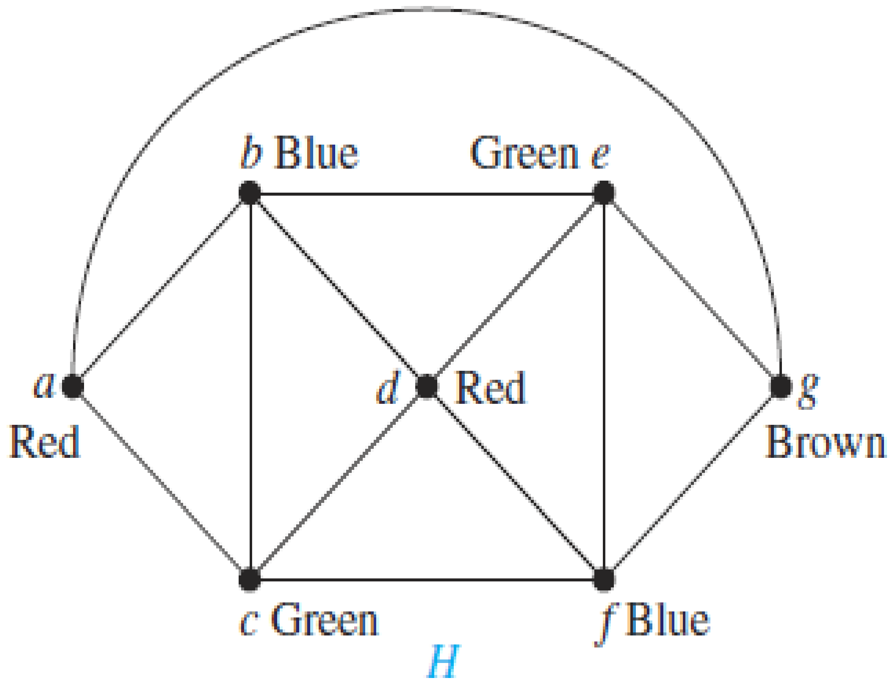
Travelling Salesman Problem

- Starting from city 1, the salesman must travel to all cities once before returning home
- The distance between each city is given, and is assumed to be the same in both directions
- Only the links shown are to be used
- Objective - Minimize the total distance to be travelled



Graph Colouring





1.3 Important Problem Types - Combinatorial Problems

- From a more abstract perspective, the traveling salesman problem and the graph-coloring problem are examples of 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.
- A desired combinatorial object may also be required to have some additional property such as a maximum value or a minimum cost.

1.3 Important Problem Types - Graph Problems

- Generally speaking, combinatorial problems are the most difficult problems in computing, from both a theoretical and practical standpoint.
- Their difficulty stems from the following fact
- First, the number of combinatorial objects typically grows extremely fast with a problem’s size, reaching unimaginable magnitudes even for moderate-sized instances.

1.3 Important Problem Types - Graph Problems

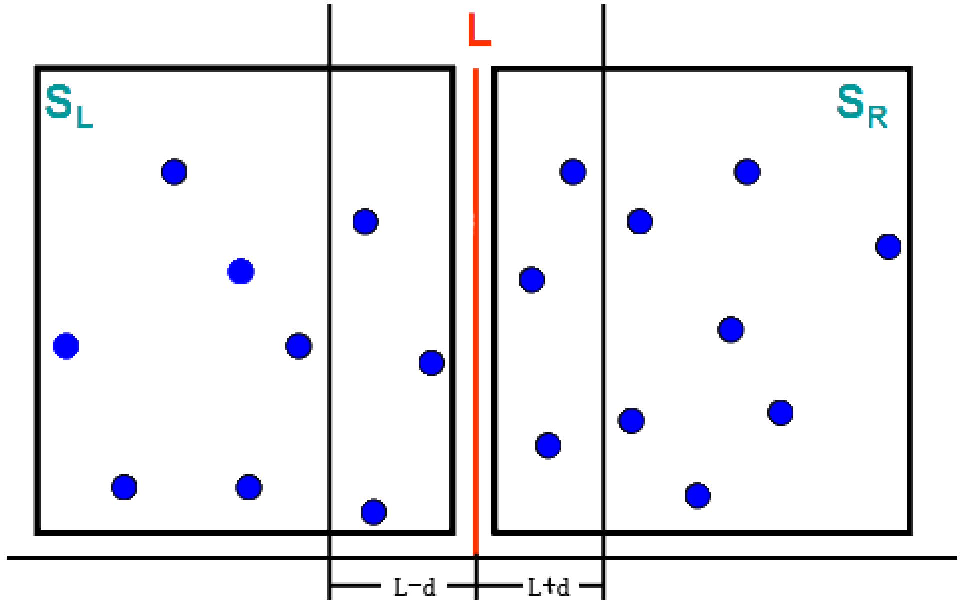
- Second, there are no known algorithms for solving most such problems exactly in an acceptable amount of time.
- Moreover, most computer scientists believe that such algorithms do not exist.
- This conjecture has been neither proved nor disproved, and it remains the most important unresolved issue in theoretical computer science.

1.3 Important Problem Types - 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—with an unmarked ruler and a compass.
- Then, for about 2000 years, intense interest in geometric algorithms disappeared, to be resurrected in the age of computers—no more rulers and compasses, just bits, bytes, and good old human ingenuity.
- Of course, today people are interested in geometric algorithms with quite different applications in mind, such as computer graphics, robotics, and tomography.

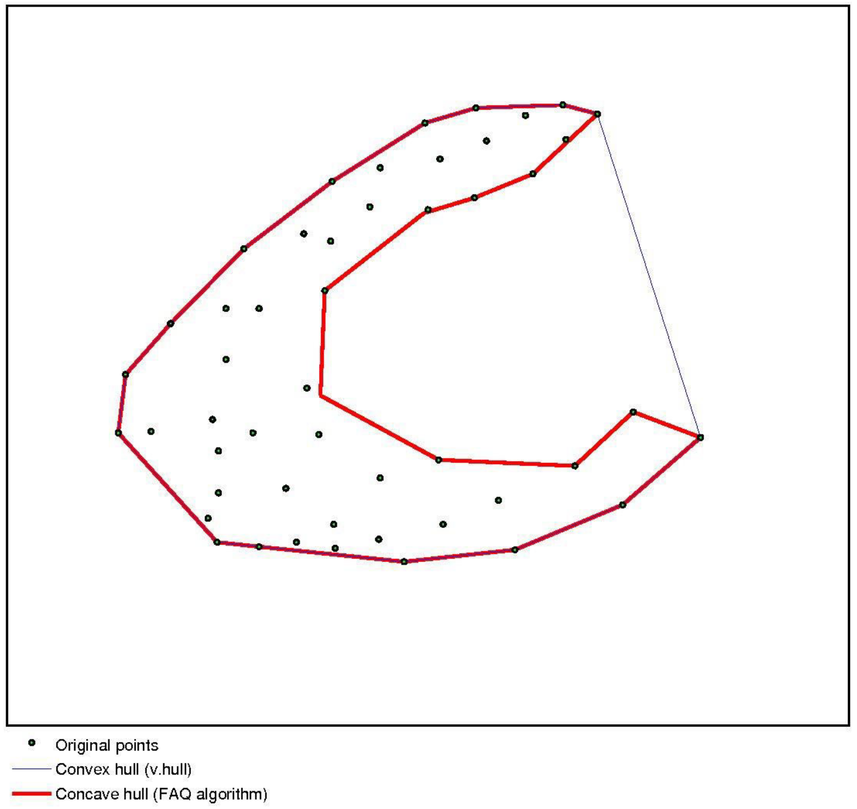
1.3 Important Problem Types - Geometric Problems

- The closest-pair problem is self-explanatory: given n points in the plane, find the closest pair among them.



1.3 Important Problem Types - Geometric Problems

- The convex-hull problem asks to find the smallest convex polygon that would include all the points of a given set.



1.3 Important Problem Types - Numerical Problems

- Numerical problems, another large special area of applications, are problems that involve mathematical objects of continuous nature: solving equations and systems of equations, computing definite integrals, evaluating functions, and so on.
- The majority of such mathematical problems can be solved only approximately.
- Another principal difficulty stems from the fact that such problems typically require manipulating real numbers, which can be represented in a computer only approximately.
- Moreover, a large number of arithmetic operations performed on approximately represented numbers can lead to an accumulation of the round-off error to a point where it can drastically distort an output produced by a seemingly sound algorithm.

1.4 Fundamental Data Structures

- 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.
- Since you are undoubtedly familiar with most if not all of them, just a quick review is provided here.

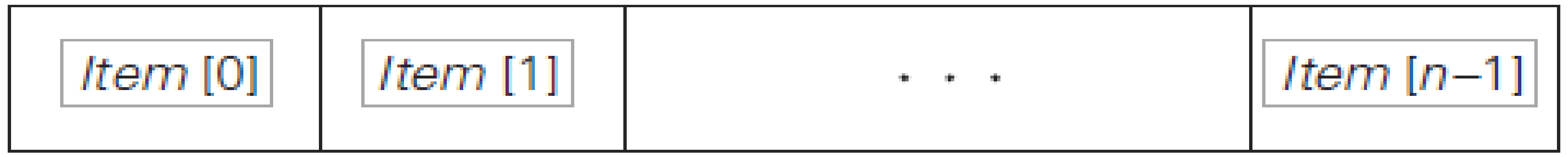
1.4 Fundamental Data Structures

- Linear Data Structures
- The two most important elementary data structures are the array and the linked list.
- 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.
- 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.

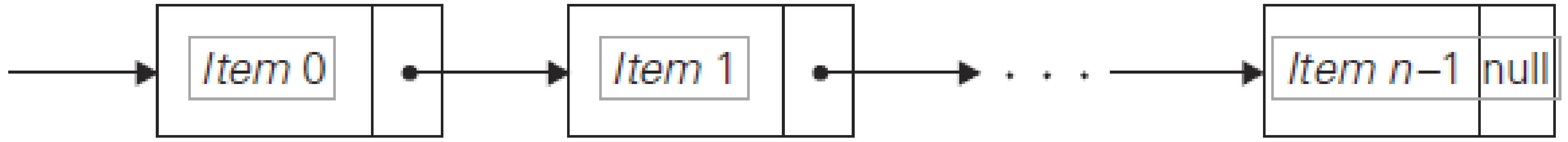
1.4 Fundamental Data Structures

- <u>Linear Data Structures</u>
- 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.

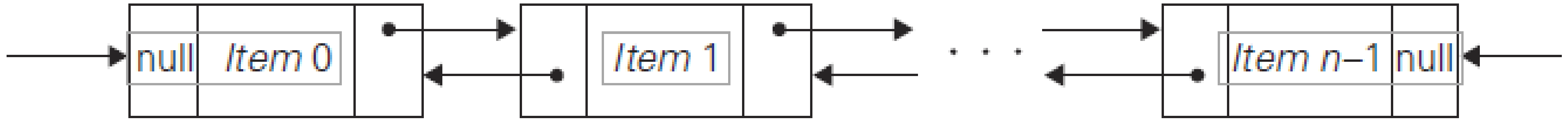
1.4 Fundamental Data Structures



Array of *n* elements.



Singly linked list of *n* elements.



Doubly linked list of *n* elements.

1.4 Fundamental Data Structures

- Linear Data Structures
- 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.
- 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.

1.4 Fundamental Data Structures

- Linear Data Structures
- 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.

- Stack - LIFO

1.4 Fundamental Data Structures

- Linear Data Structures
- 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).

- Queue - FIFO

1.4 Fundamental Data Structures

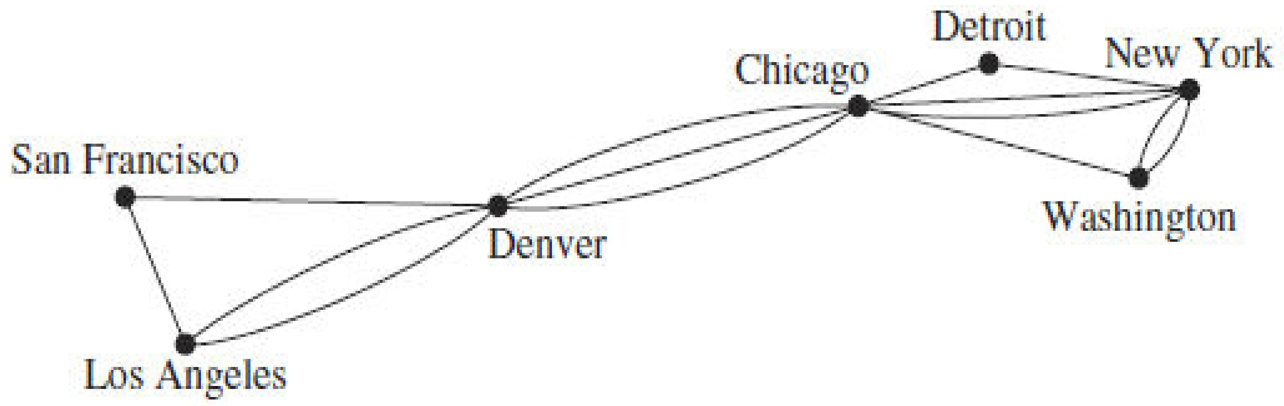
- Graphs
- A graph is informally thought of as a collection of points in the plane called “vertices” or “nodes,” some of them connected by line segments called “edges” or “arcs.”
- Formally, a graph G ={V, E} is defined by a pair of two sets: a finite nonempty set V of items called vertices and a set 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; we also say that the edge (u, v) is incident to its endpoints u and v.

1.4 Fundamental Data Structures

- <u>Graphs</u>
- 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.

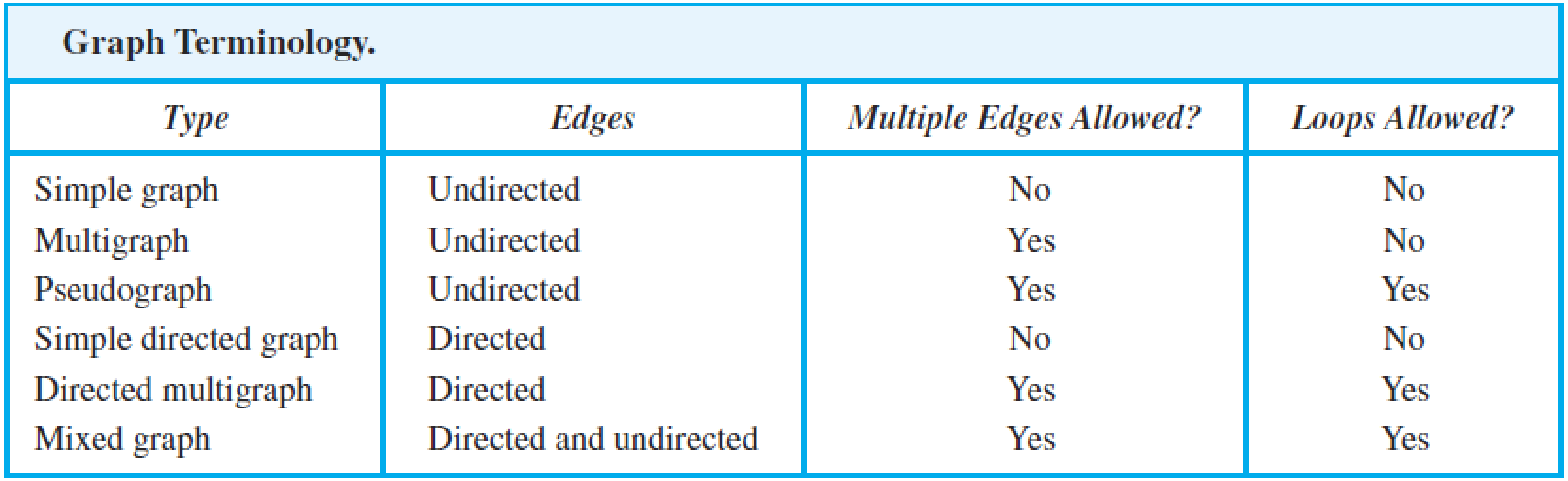
1.4 Fundamental Data Structures

- <u>Graphs</u>
- The set of vertices V of a graph G may be infinite.
- A graph with an infinite vertex set or an infinite number of edges is called an infinite graph, and in comparison, a graph with a finite vertex set and a finite edge set is called a finite graph.



1.4 Fundamental Data Structures

- <u>Graphs</u>



Graph Terminology.

Type	*Edges*	*Multiple Edges Allowed?*	*Loops Allowed?*
Simple graph	Undirected	No	No
Multigraph	Undirected	Yes	No
Pseudograph	Undirected	Yes	Yes
Simple directed graph	Directed	No	No
Directed multigraph	Directed	Yes	Yes
Mixed graph	Directed and undirected	Yes	Yes

1.4 Fundamental Data Structures

- 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.

1.4 Fundamental Data Structures

- Graphs
- 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 ith to the jth 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.

1.4 Fundamental Data Structures

- <u>Graphs</u>
- 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.

- 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.

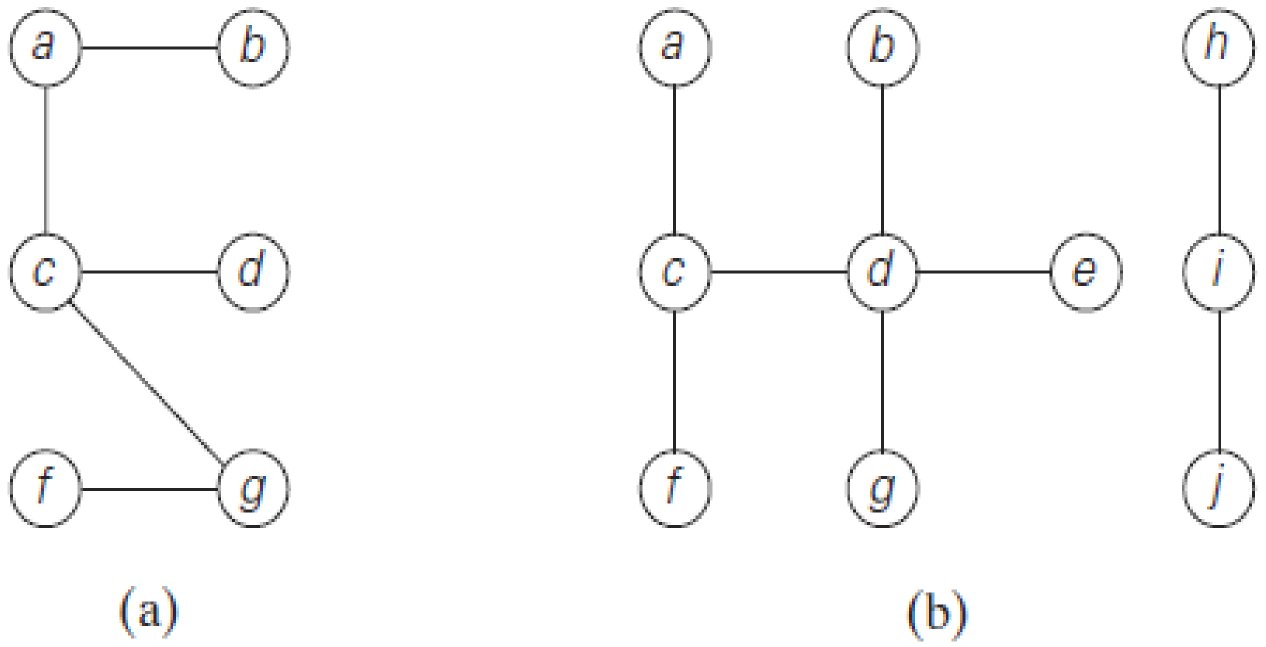
1.4 Fundamental Data Structures

- Trees

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

1.4 Fundamental Data Structures

- <u>Trees</u>



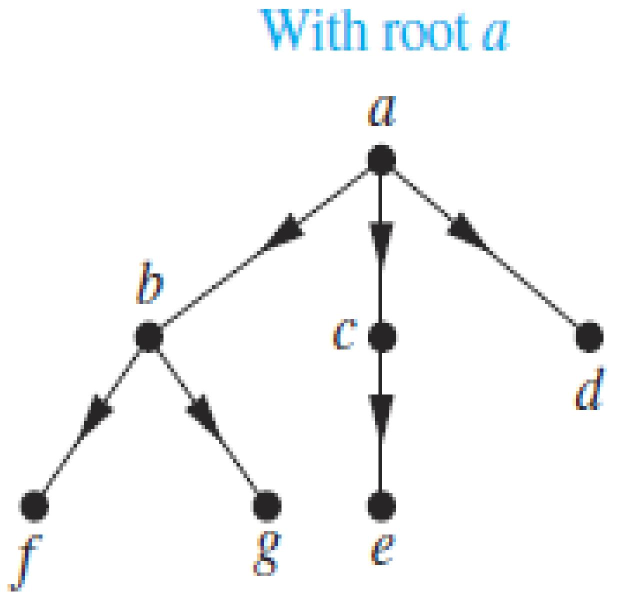
(a) Tree. (b) Forest.

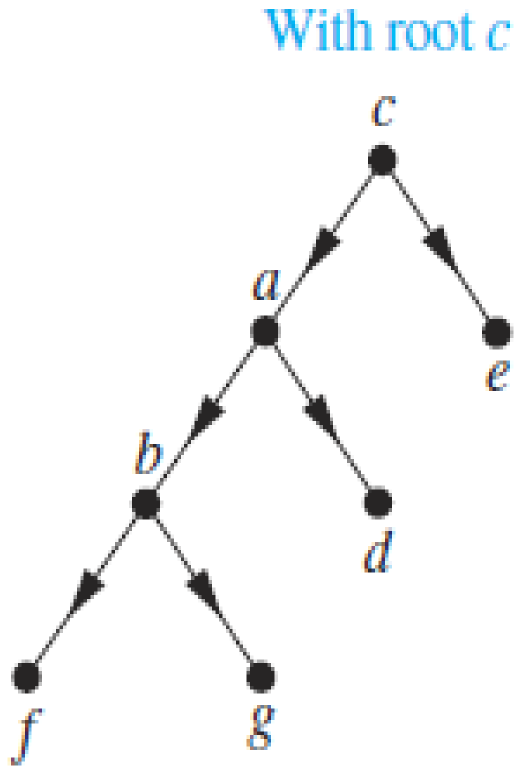
1.4 Fundamental Data Structures

- Trees
- 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.

1.4 Fundamental Data Structures

- Trees



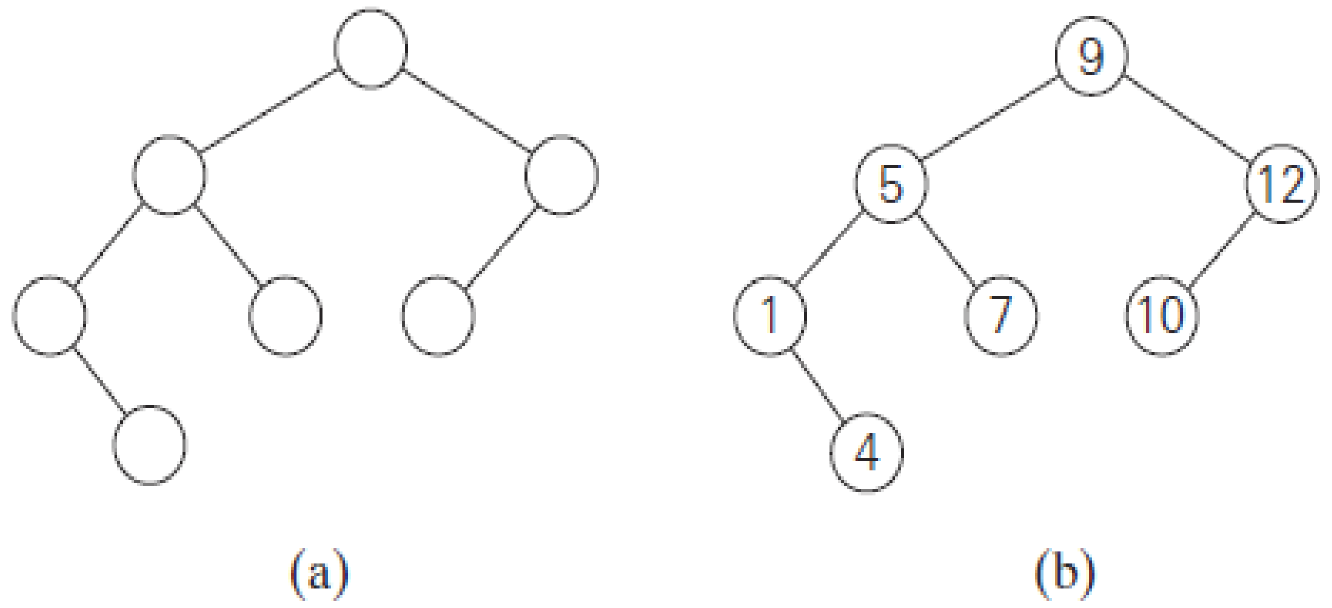


1.4 Fundamental Data Structures

- 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.
- 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.
- 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.

1.4 Fundamental Data Structures

- <u>Trees</u>



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

1.4 Fundamental Data Structures

- Sets and Dictionaries
- 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}).
- 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; and finding the intersection of two sets, which comprises all the common elements in the sets.

1.4 Fundamental Data Structures

- Sets and Dictionaries
- 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.