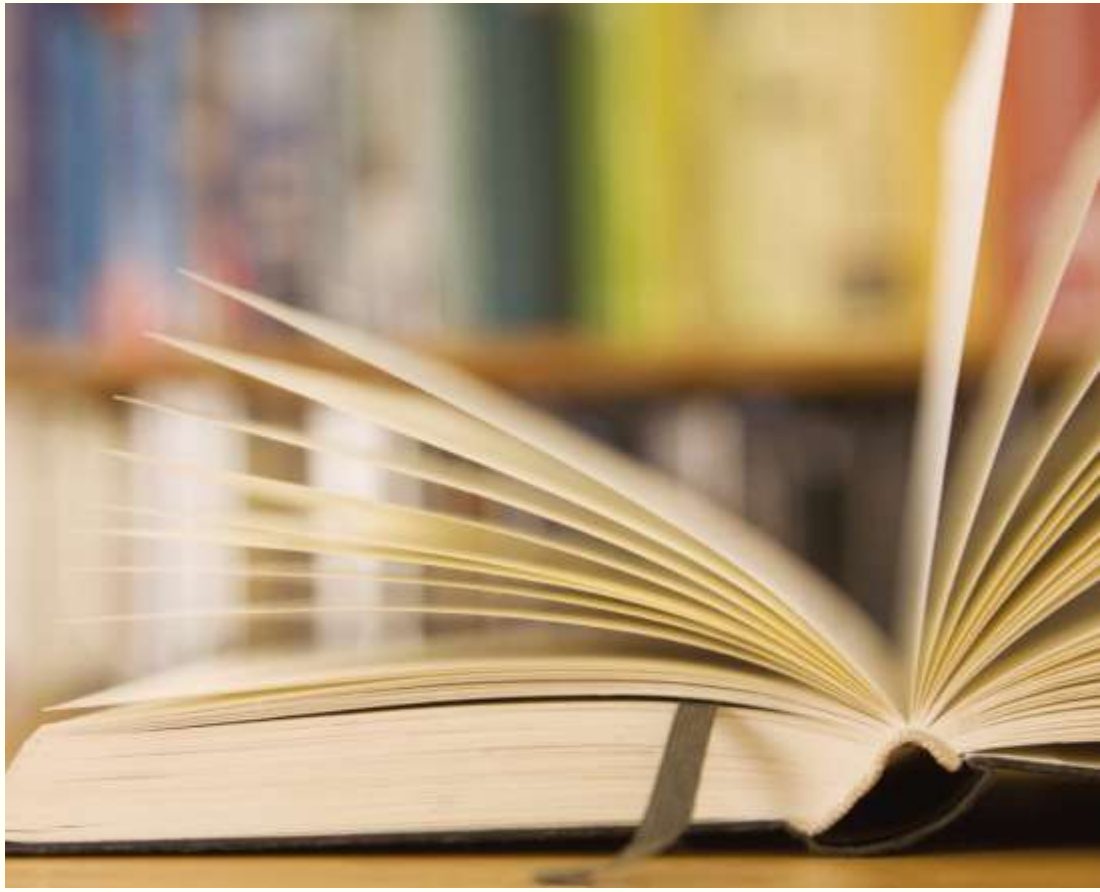
9. DATA STRUCTURES



FPT Education

FPT UNIVERSITY

Content

- 9.1 Arrays
- 9.2 Records
- 9.3 Linked List
- 9.4 Introduction: Stack, Queue, Tree, graph

Objectives

After studying this chapter, the student should be able to:

- Define a data structure.
- Define an array as a data structure and how it is used to store a list of data items.
- Distinguish between the name of an array and the names of the elements in an array.
- Describe operations defined for an array.
- Define a record as a data structure and how it is used to store attributes belonging to a single data element.
- Distinguish between the name of a record and the names of its fields.
- Define a linked list as a data structure and how it is implemented using pointers.
- Describe operations defined for a linked list.
- Compare and contrast arrays, records, and linked lists.
- Define the applications of arrays, records, and linked lists.

1 - ARRAYS

1. Introduction

- An array is a sequenced collection of elements, of the same data type. We can refer to the elements in the array as the first element, the second element, and so forth until we get to the last element.
- If we were to put our 100 scores into an array, we could designate the elements as scores[1], scores[2], and so on.
- The index indicates the ordinal number of the element, counting from the beginning of the array. The elements of the array are individually addressed through their subscripts (Figure 11.3).
- The array as a whole has a **name**, **scores**, but each score can be accessed individually using its subscript.

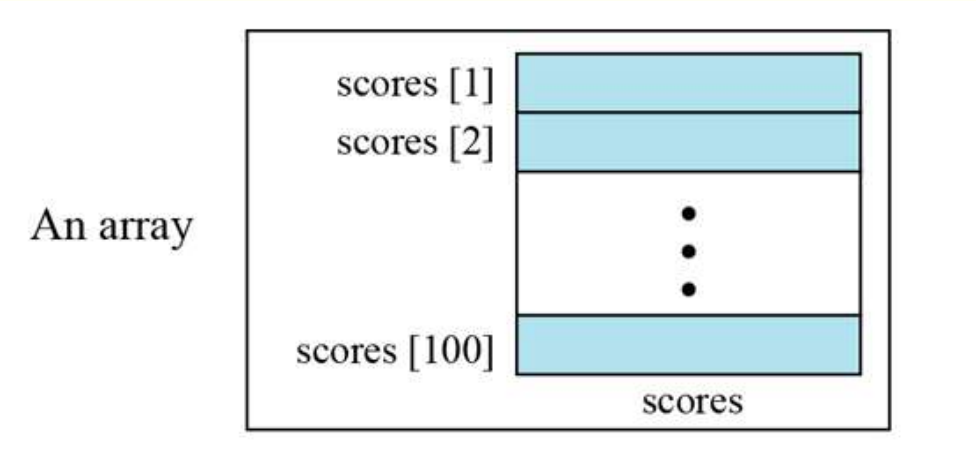


Figure 9.1 *Array with indexs*

2. Loops

- We can use loops to read and write the elements in an array. We can also use loops to process elements. Now it does not matter if there are 100, 1000, or 10,000 elements to be processed—loops make it easy to handle them all.
- We can use an integer variable to control the loop, and remain in the loop as long as the value of this variable is less than the total number of elements in the array (Figure 11.4).

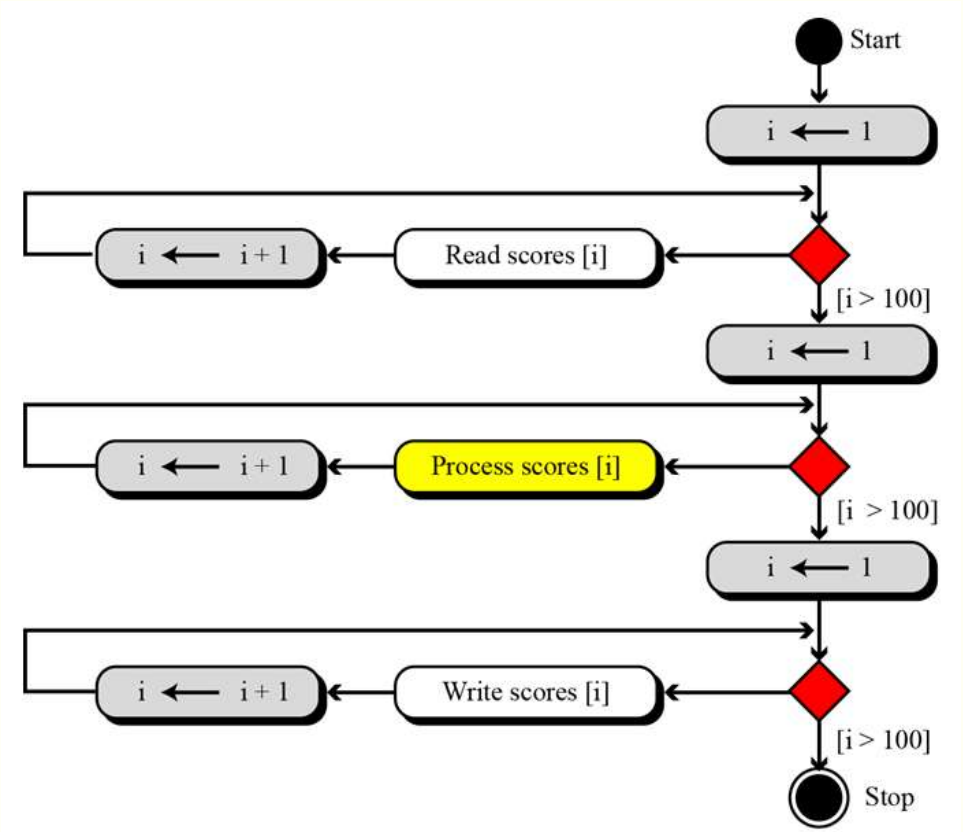


Figure 9.2 *Processing a array*

3. Multi-dimensional arrays

- The arrays discussed so far are known as one-dimensional arrays because the data is organized linearly in only one direction.
- Many applications require that data be stored in more than one dimension. Figure 11.5 shows a table, which is commonly called a two-dimensional array.

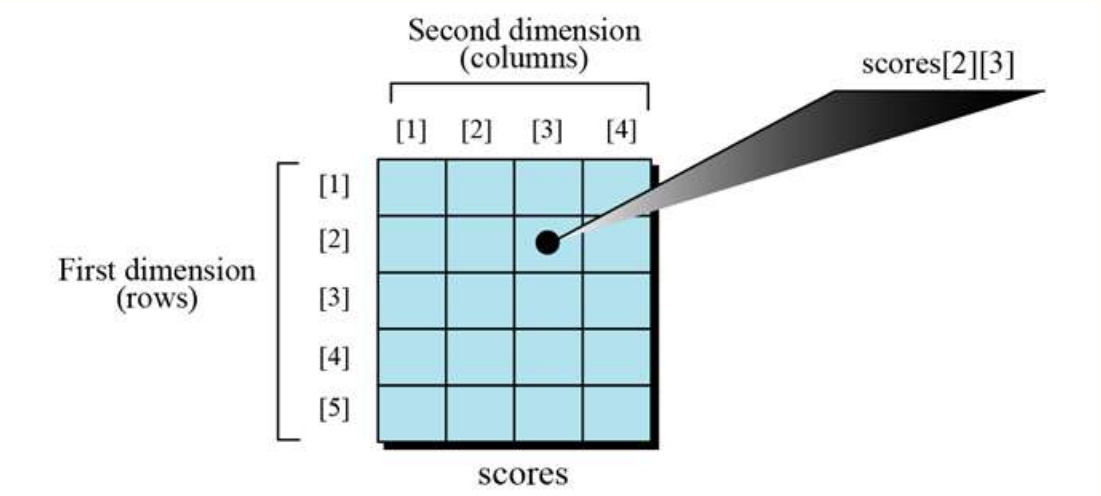


Figure 9.3 *A two-dimensional array*

4. Memory layout

- The indexes in a one-dimensional array directly define the relative positions of the element in actual memory. A two-dimensional array, however, represents rows and columns. How each element is stored in memory depends on the computer.

- Most computers use **row-major storage**, in which an entire row of an array is stored in memory before the next row. However, a computer may store the array using **column-major storage**, in which the entire column is stored before the next column.

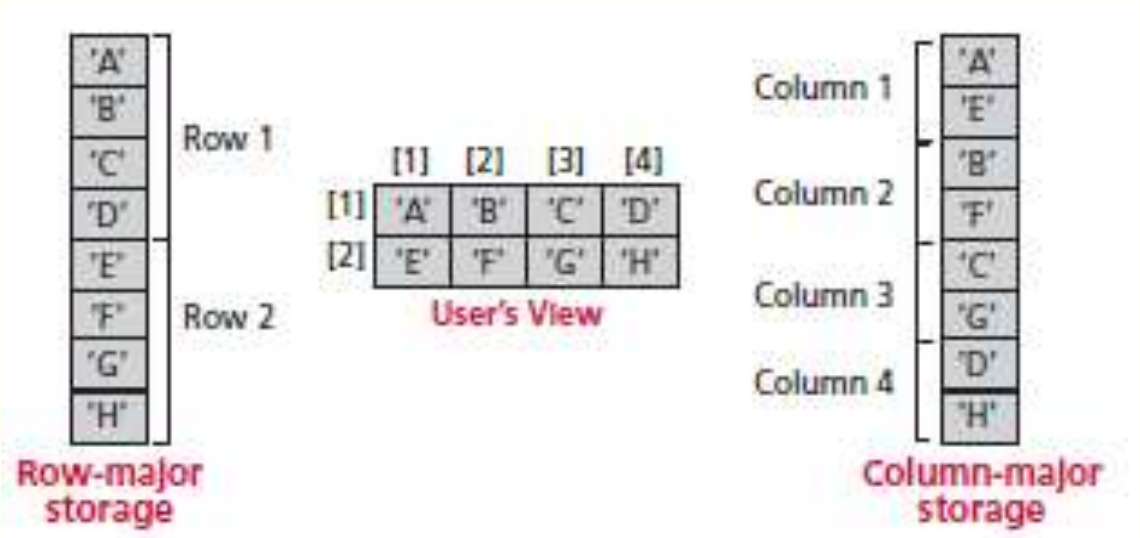


Figure 9.4 *Memory layout of arrays*

5. Operations on array

- Although we can apply conventional operations defined for each element of an array there are some operations that we can define on an array as a data structure.
- The common operations on arrays as structures are searching, insertion, deletion, retrieval, and traversal

```
i ← 30
while (i ≥ 9)
{
  array [i + 1] ← array [i]
  i ← i – 1
}
array [i] ← newValue
```

Figure 9.5 *Insert a value to array*

2- RECORDS

1. Introduction

- A **record** is a collection of related elements, possibly of different types, having a single name. Each element in a record is called a *field*.
- A **field** is the smallest element of named data that has meaning. A field has a type, and exists in memory. Fields can be assigned values, which in turn can be accessed for selection or manipulation. A field differs from a variable primarily in that it is part of a record.

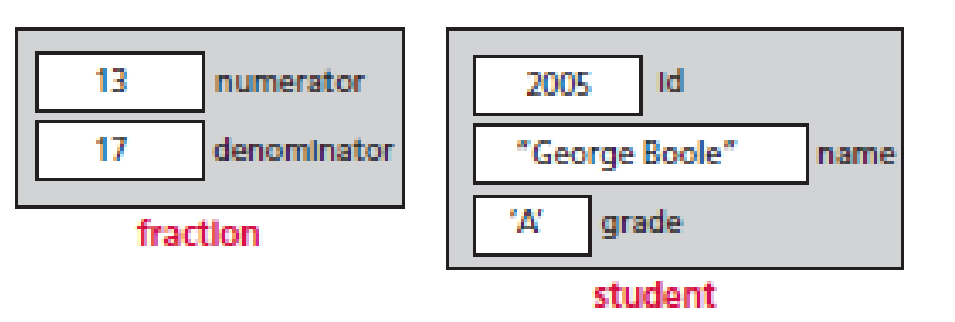


Figure 9.6 *Records*

The elements in a record can be of the same or different types, but all elements in the record must be related.

2. Record name versus field name

- Just like in an array, we have two types of identifier in a record: the name of the record and the name of each individual field inside the record. The name of the record is the name of the whole structure, while the name of each field allows us to refer to that field.
- For example, in the student record of Figure 11.7, the name of the record is student, the name of the fields are student.id, student.name, and student.grade

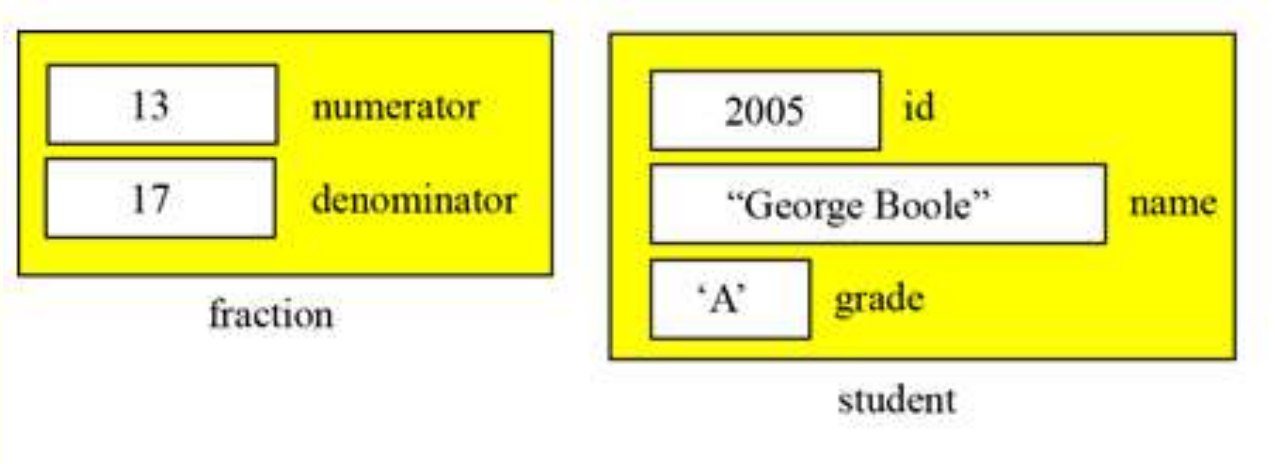


Figure 9.7 *Insert a value to array*

3. Comparison of records and arrays

- We can conceptually compare an array with a record. This helps us to understand when we should use an array and when a record. An array defines a combination of elements, while a record defines the identifiable parts of an element.
- For example, an array can define a class of students (40 students), but a record defines different attributes of a student, such as id, name, or grade.

```
ct java.util.ArrayList;

ic class RemoveArrayList

blic static void main(String arg[])

ArrayList<ArrayList<String>> details = new ArrayList<ArrayList<String>>(

//create student 1
ArrayList<String> s1 = new ArrayList<String>();
s1.add("Jane Doe");
s1.add("50");
details.add(s1);

//create student 2
ArrayList<String> s2 = new ArrayList<String>();
s2.add("John Doe");
s2.add("Part Time");
s2.add("70");
details.add(s2);

//output details
System.out.println("Output");
```

Figure 9.8 *Insert a record (name, status, grade) to array of students in Java*

4. Array of records

- If we need to define a combination of element and at the same time some attributes of each element, we can use an array of records. For example, in a class of 30 students, we can have an array of 30 records, each record representing a student. Figure 11.8 shows an array of 30 student records called students.

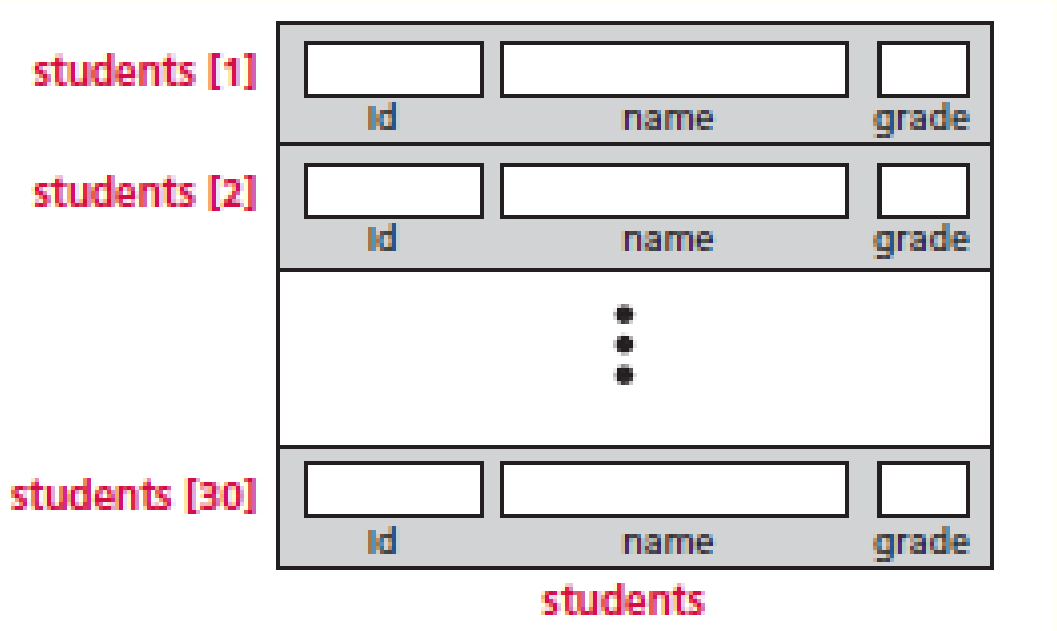


Figure 9.9 *Insert a value to array*

3 - LINKED LIST

1. Introduction

- A **linked list** is a collection of data in which each element contains the location of the next element—that is, each element contains two parts: data and **link**. The data part holds the value information: the data to be processed.

- The link is used to chain the data together, and contains a **pointer** (an address) that identifies the next element in the list. In addition, a pointer variable identifies the first element in the list. The name of the list is the same as the name of this pointer variable.

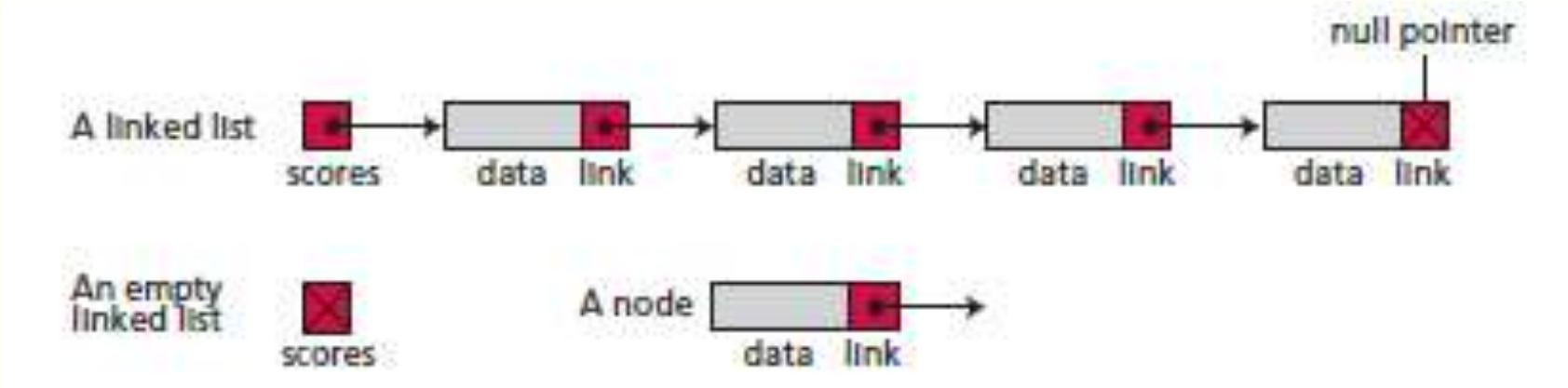


Figure 9.10 Linked lists

2. Arrays versus linked lists

- Both an array and a linked list are representations of a list of items in memory. The only difference is the way in which the items are linked together. In an array of records, the *linking tool* is the index.
- The element scores[3] is linked to the element scores[4] because the integer 4 comes after the integer 3. In a linked list, the ***linking tool*** is the link that points to the next element—the pointer or the address of the next element.
- Figure 11.11 compares the two representations for a list of five integers.

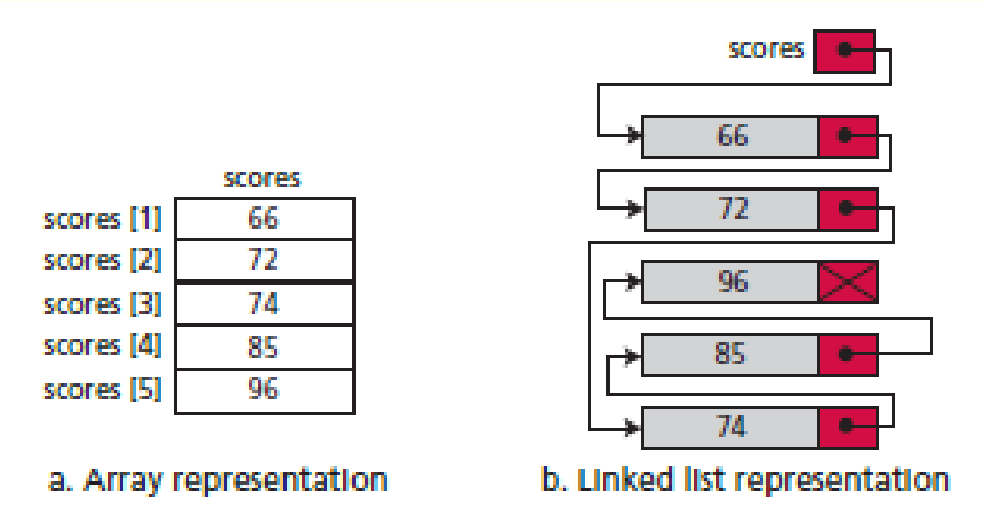


Figure 9.11 *Array* versus *linked list*

3. Linked list names versus nodes names

- As for arrays and records, we need to distinguish between the name of the linked list and the names of the nodes, the elements of a linked list. A linked list must have a name.

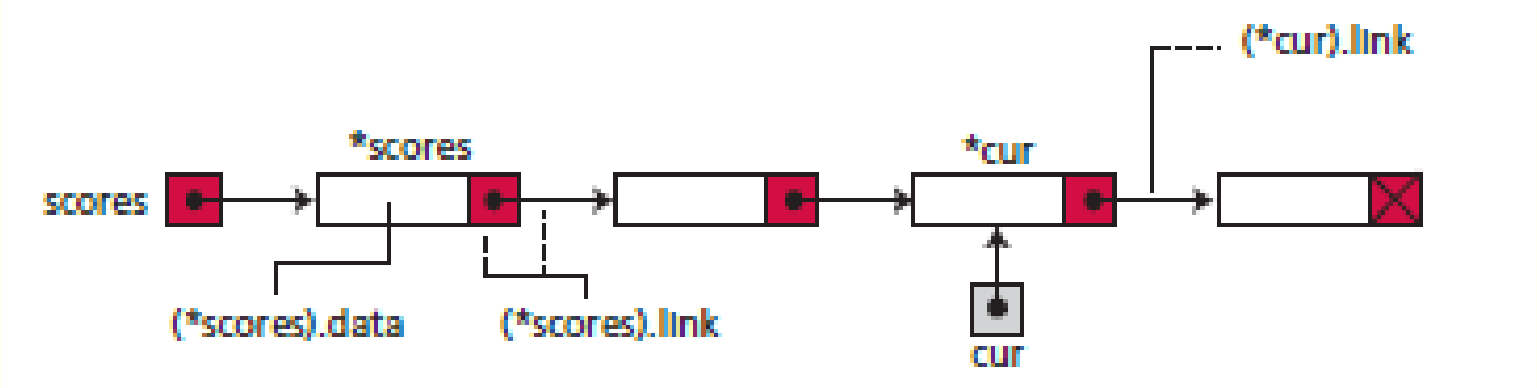


Figure 9.12 *The name of a linked list* versus *the names of nodes*

4. Operations on linked lists (searching)

- The search algorithm for a linked list can only be sequential because the nodes in a linked list have no specific names (unlike the elements in an array) that can be found using a binary search.
- However, since nodes in a linked list have no names, we use two pointers, pre (for previous) and cur (for current).

```
Algorithm: SearchLinkedList
Purpose: Search the list using two pointers: pre and cur.
Pre: The linked list (head pointer) and target value
Post: None
Return: The position of pre and cur pointers and the value of the flag
{
    pre ← null
    cur ← list
    while (target < (*cur).data)
    {
        pre ← cur
        cur ← (*cur).link
    }
    if ((*cur).data = target)   flag ← true
    else   flag ← false
}
```

Algorithm 9.1 Searching a linked list

5. Operations on linked lists (inserting)

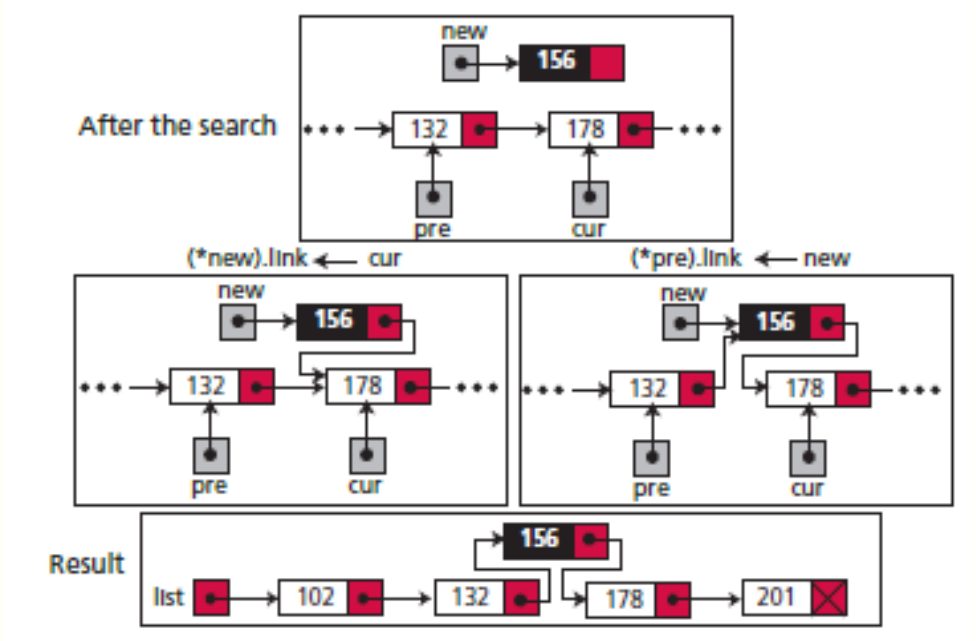
- Before insertion into a linked list need searching algorithm.
- If the flag returned from the searching algorithm is false, will allow insertion,
- otherwise abort the insertion algorithm Four cases can arise:

❑Inserting into an empty list.

❑Insertion at the beginning of the list.

❑Insertion at the end of the list.

❑Insertion in the middle of the list.



Algorithm 9.2 Inserting to a linked list

4- INTRODUCTION: STACK, QUEUE, TREE, GRAPH

1. BACKGROUND

- Problem solving with a computer means processing data. To process data, we need to define the data type and the operation to be performed on the data.
- For example, to find the sum of a list of numbers, we should select the type for the number (integer or real) and define the operation (addition). The definition of the data type and the definition of the operation to be applied to the data is part of the idea behind an **abstract data type**

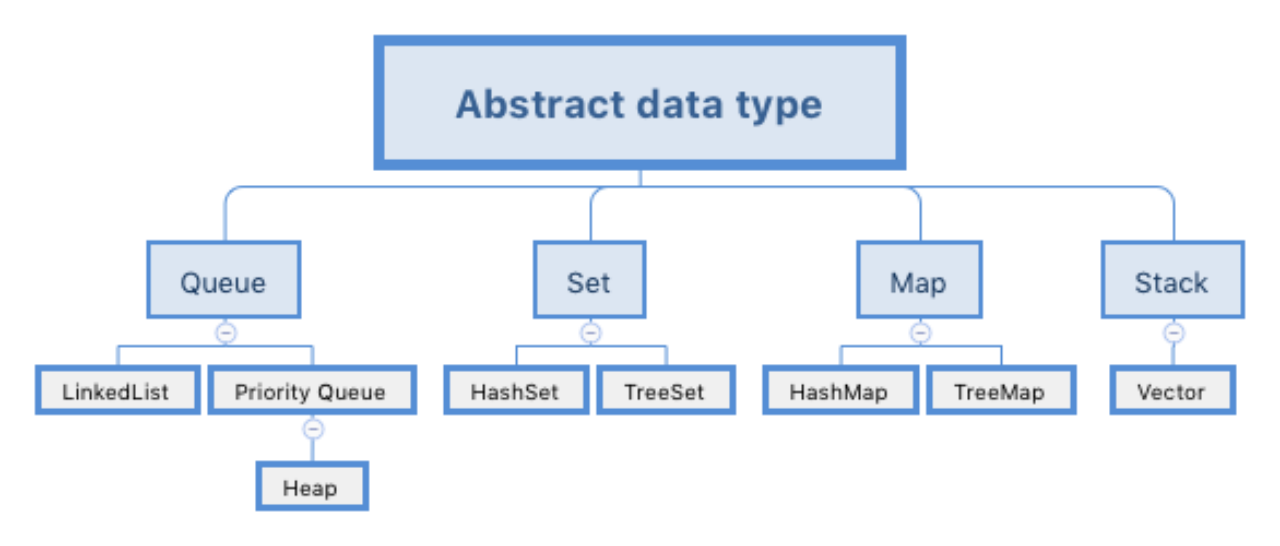


Figure 9.13 Structure of abstract data type

2. STACKS

- A stack is a restricted linear list in which all additions and deletions are made at one end, the top. If we insert a series of data into a stack and then remove it, the order of the data is reversed. Data input as 5, 10, 15, 20, for example, would be removed as 20, 15, 10, and 5.

Figure 9.14 *Three representations of stacks*

Operations on stacks

- Although we can define many operations for a stack, there are four basic operations, ***stack***, ***push***, ***pop***, and ***empty***, which we define in this chapter.

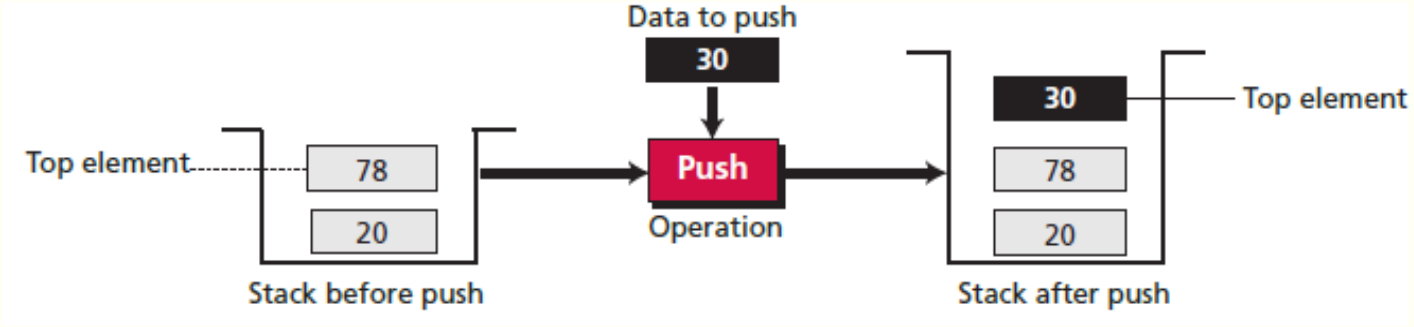


Figure 12.2 *Three representations of stacks*

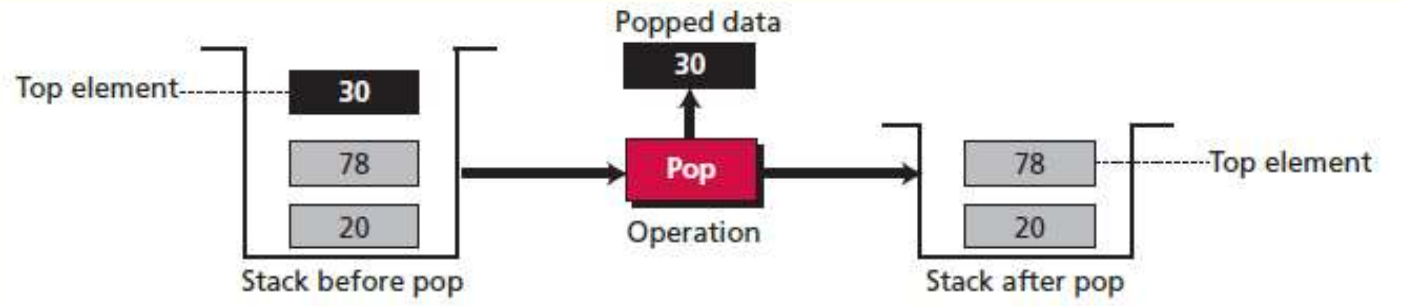


Figure 9.15 *Three representations of stacks*

3. QUEUES

- A queue is a linear list in which data can only be inserted at one end, called the rear, and deleted from the other end, called the front. These restrictions ensure that the data are processed through the queue in the order in which it is received. In other words, a queue is a first in, first out (FIFO) structure.

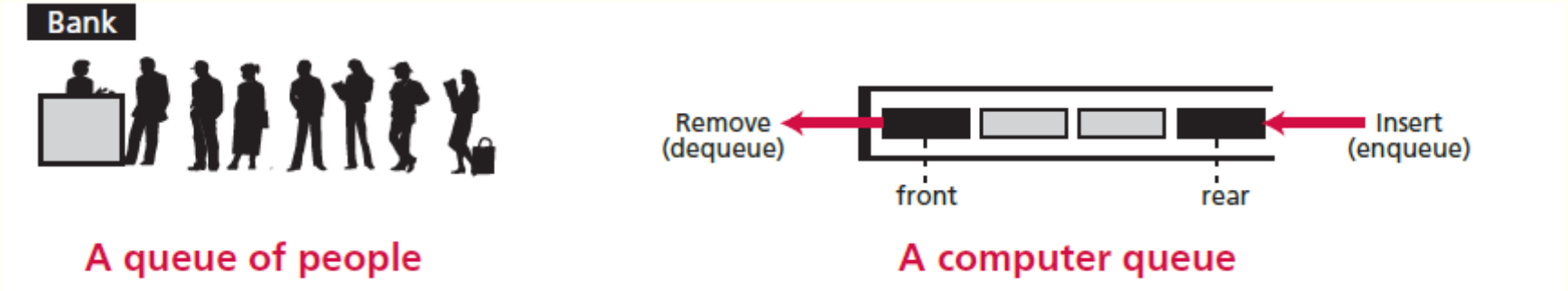


Figure 9.16 *Two representations of queues*

Operations on queues

- ***The queue operation***
- The *queue* operation creates an empty queue. The following shows the format:

```
queue (queueName)
```

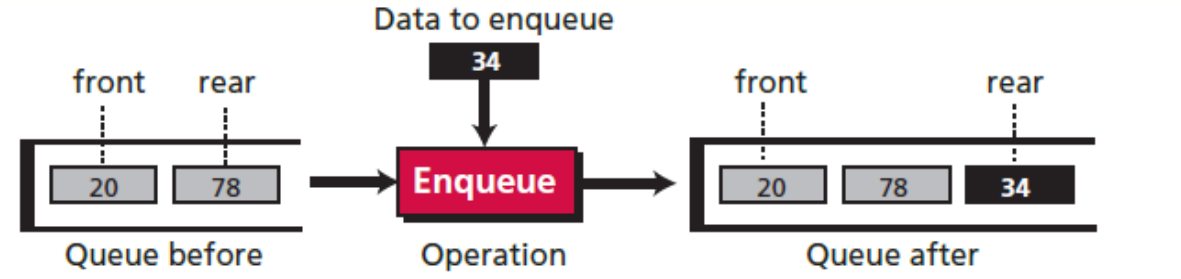


Figure 9.17 *The enqueue operation*

Operations on queues (cont)

- ***The dequeue operation***
- The **dequeue** operation deletes the item at the front of the queue. The following shows the format:

```
dequeue (queueName, dataItem)
```

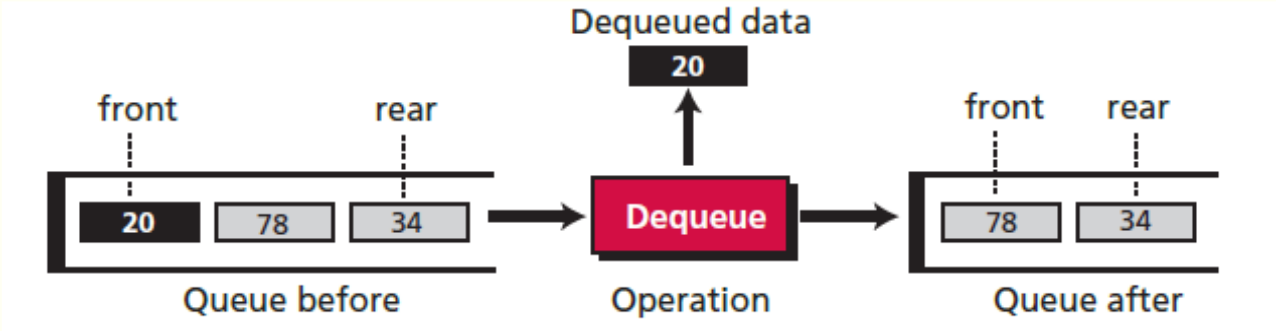


Figure 9.18 *The dequeue operation*

Operations on queues (cont)

- ***The empty operation***
- The *empty* operation checks the status of the queue. The following shows the format:

```
empty (queueName)
```

Queue ADT

- We define a queue as an ADT as shown below:

Queue ADT

Definition

A list of data items in which an item can be deleted from one end, called the front and an item can be inserted at the other end called the rear.

Operations

queue: Creates an empty queue.

enqueue: Inserts an element at the rear.

dequeue: Deletes an element from the front.

empty: Checks the status of the queue.

Queue implementation

- At the ADT level, we use the queue and its four operations (queue, enqueue, dequeue, and empty): at the implementation level, we need to choose a data structure to implement it.
- A queue ADT can be implemented using either an array or a linked list

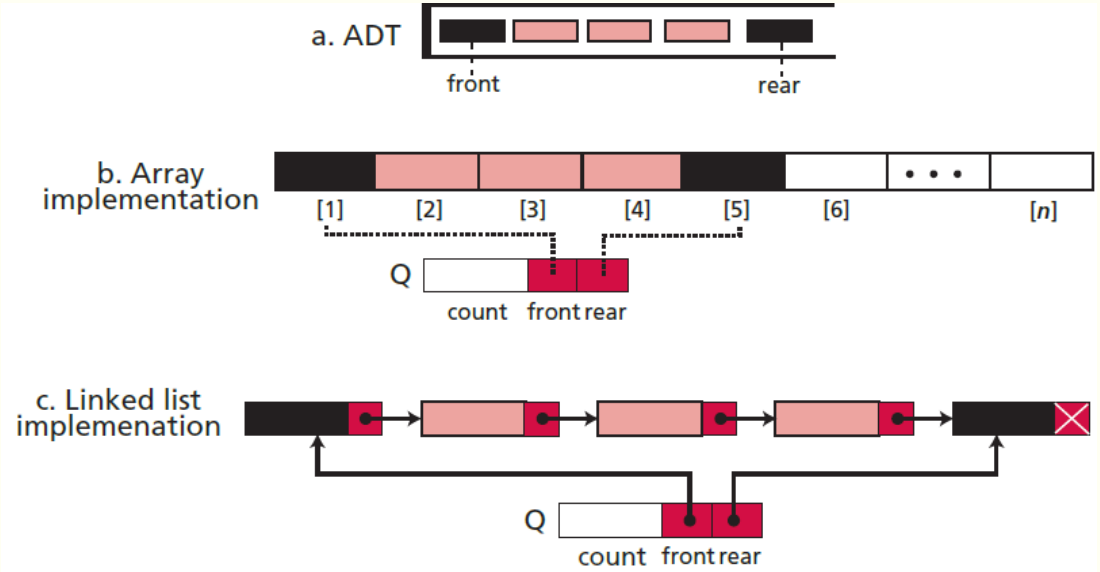


Figure 9.19 *Queue implementation*

3. TREES

- A tree consists of a finite set of elements, called nodes (or vertices), and a finite set of directed lines, called arcs, that connect pairs of the nodes. If the tree is not empty, one of the nodes, called the root, has no incoming arcs.
- The other nodes in a tree can be reached from the root by following a unique path, which is a sequence of consecutive arcs. Tree structures are normally drawn upside down with the root at the top

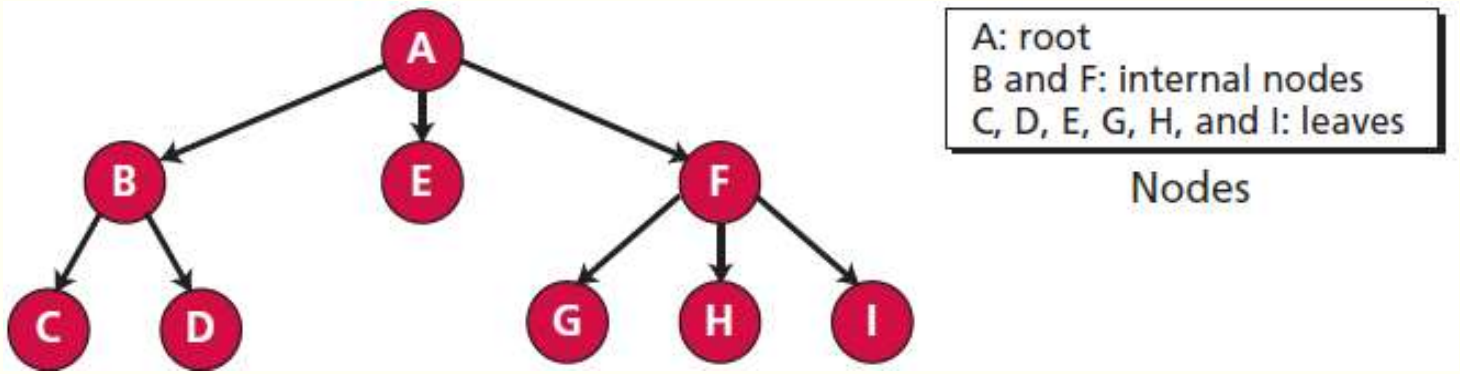


Figure 9.20 *Tree representation*

Binary trees

- A binary tree is a tree in which no node can have more than two subtrees. In other words, a node can have zero, one, or two subtrees. These subtrees are designated as the left subtree and the right subtree.
- Figure 12.22 shows a binary tree with its two subtrees

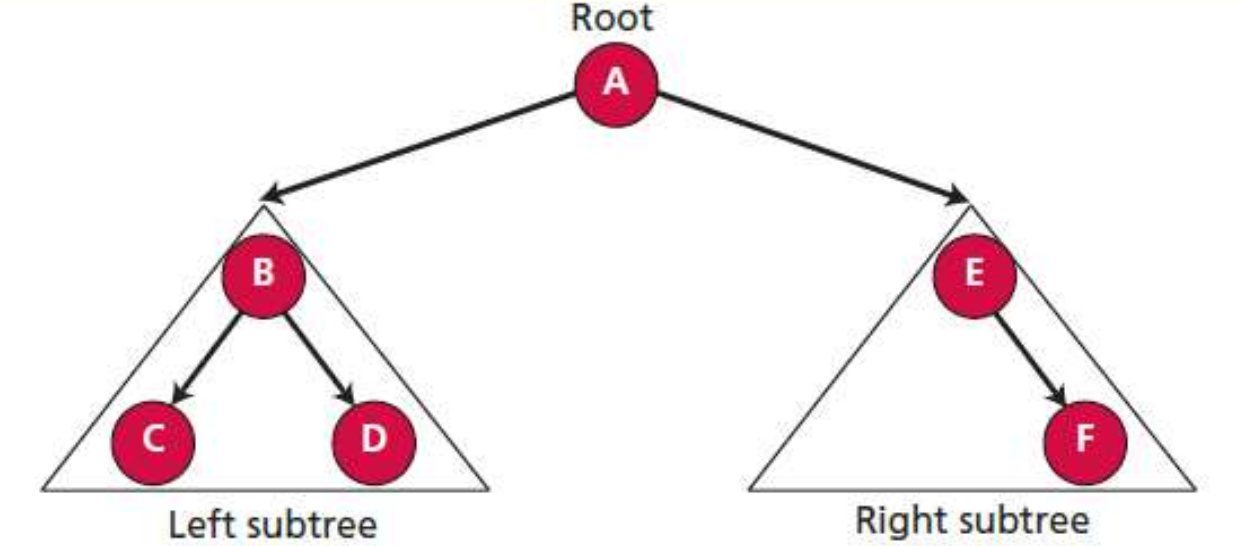


Figure 9.21 *A binary tree*

Operations on binary trees

- ***Binary tree traversals***

- A *binary tree traversal* requires that each node of the tree be processed once and only once in a predetermined sequence. The two general approaches to the traversal sequence are *depth-first* and *breadth-first* traversal

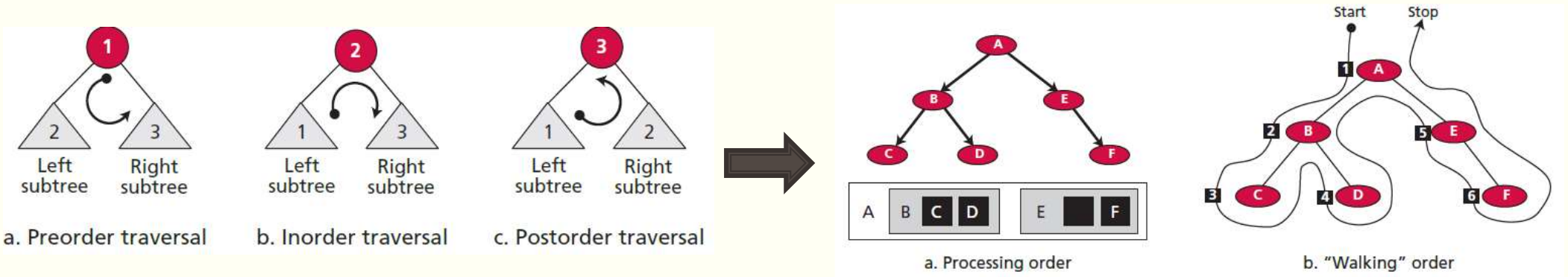


Figure 9.22 *Depth-first traversal of a binary tree*

Binary tree applications

- ***Huffman coding*** is a compression technique that uses binary trees to generate a variable length binary code from a string of symbols.

- ***Expression trees*** An arithmetic expression can be represented in three different formats: **infix**, **postfix**, and **prefix**. In an infix notation, the operator comes between the two operands. In postfix notation, the operator comes after its two operands, and in prefix notation it comes before the two operands. These formats are shown below for the addition of two operands A and B.

Prefix: + A B **Infix: A + B** **Postfix: A B +**

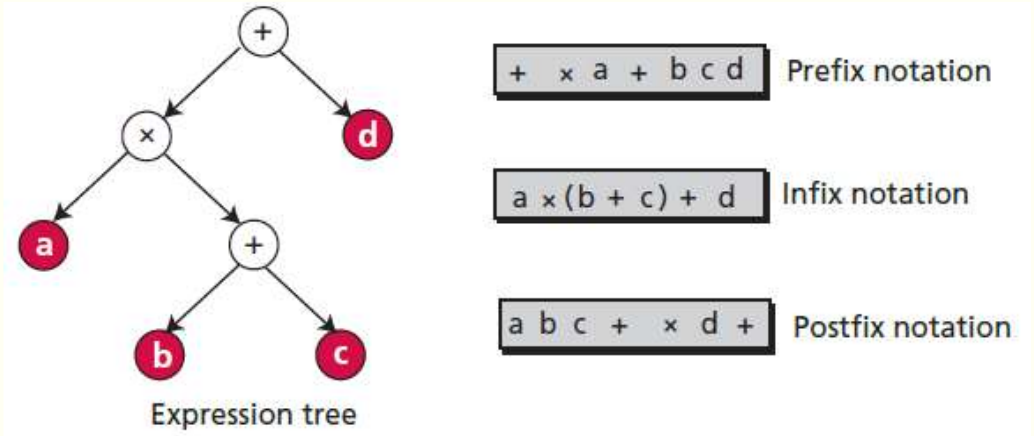


Figure 9.23 *Expression tree*

Binary search trees

- A binary search tree (BST) is a binary tree with one extra property: the key value of each node is greater than the key values of all nodes in each left subtree and smaller than the value of all nodes in each right subtree. Figure 12.28 shows the idea

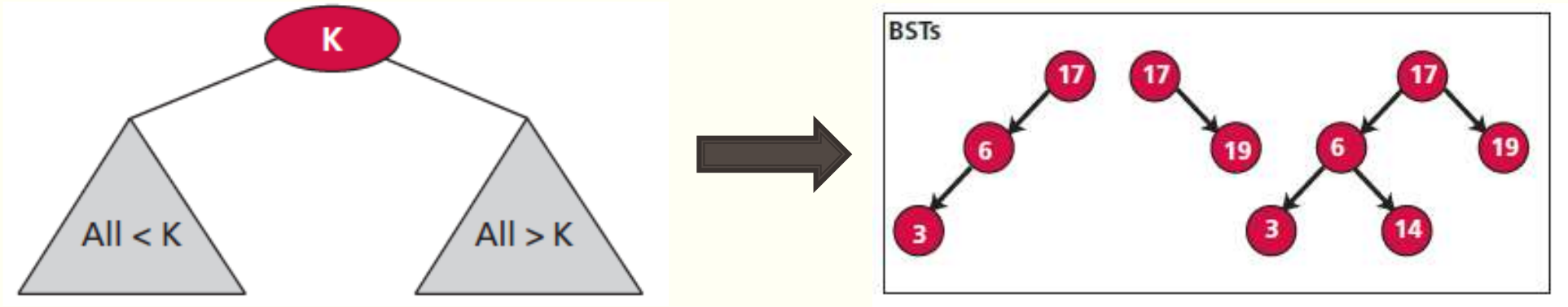


Figure 9.24 *Binary search tree (BST)*

4. GRAPHS

- A **graph** is an ADT made of a set of nodes, called vertices, and set of lines connecting the vertices, called **edges** or arcs. Whereas a tree defines a hierarchical structure in which a node can have only one single parent, each node in a graph can have one or more parents.

- Graphs may be either *directed* or *undirected*. In a **directed graph**, or **digraph**, each edge, which connects two vertices, has a direction (shown in the figure by an arrowhead) from one vertex to the other. In an **undirected graph**, there is no direction. Figure 12.32 shows an example of both a directed graph (a) and an undirected graph (b).

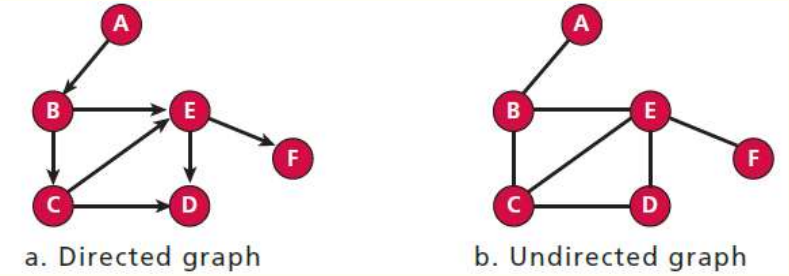


Figure 9.25 *Graph*

Example

- Graphs are directly applicable to real-world scenarios. For example, we could use graphs to model a transportation network where nodes would represent facilities that send or receive products and edges would represent roads or paths that connect them (see below).

- Use *weighted graphs*, in which each edge has a weight that represent the distance between two cities connected by that edge.

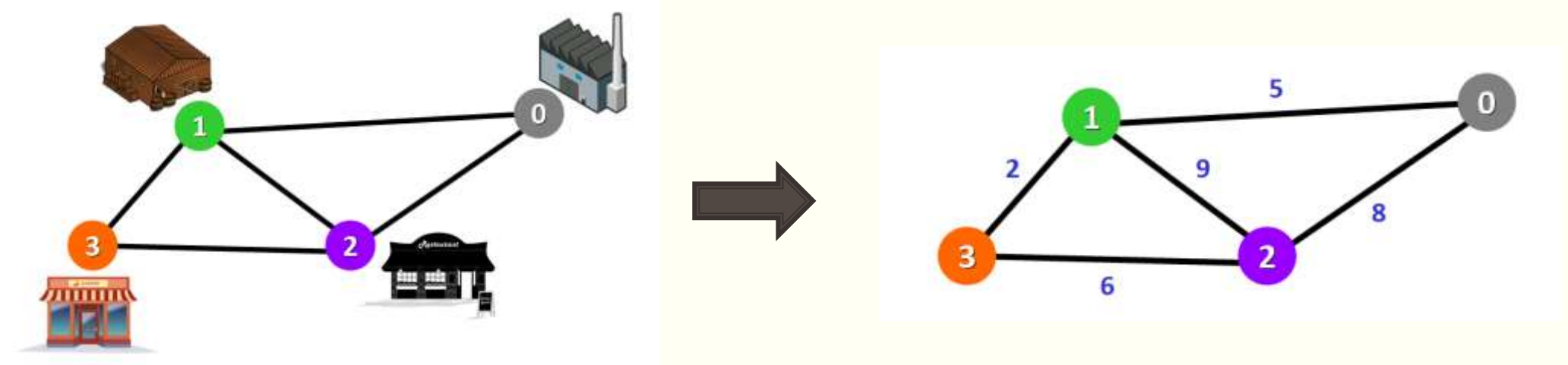


Figure 9.26 *Network represented with a graph*