Classic Data Structures

D. SAMANTA

School of Information Technology
Indian Institute of Technology Kharagpur

Prentice-Hall of India Private Limited
New Delhi - 110001
2006

Contents

Preface

Data structures are commonly used in many program designs. The study of data structures, therefore, rightly forms the central course of any curriculum in computer science and engineering. Today, most curricula in computer science courses cover topics such as "Introduction to Computing", "Principles of Programming Languages", "Programming Methodologies", "Algorithms", etc. The study of these topics is not possible without first acquiring a thorough knowledge of data structures.

Today's computer world has become unimaginably fast. To use the full strength of computing power to solve all sorts of complicated problems through elegant program design, a good knowledge of the data structures is highly essential. To be more precise, writing efficient programs and managing different types of real and abstract data is an art; and data structure is the only ingredient to promote this art. In-depth concepts of data structures help in mastering their applications in real software projects.

In the last few years, there has been a tremendous progress in the field of data structures and its related algorithms. This book offers a deep understanding of the essential concepts of data structures. The book exposes the reader to different types of data structures such as arrays, linked lists, stacks, queues, tables, trees, graphs, and sets for a good grounding in each area. These data structures are known as classic data structures, as with the help of these any abstract data which fits with real world applications can be implemented.

The study of data structures remains incomplete if their computer representations and operational details are not covered. The books currently available on data structures present the operational details with raw codes, which many readers, especially those new to the field, find difficult to understand. In the present title, operations on data structures are described in English-like constructs, which are easy to comprehend by students new to the computer science discipline.

This text is designed primarily for use in undergraduate engineering courses, but its clear analytic explanations in simple language also make it suitable for study by polytechnic students. The book can also be used as a self-study course on data structures.

The book is designed to be both *versatile* and *complete*, in the sense that for each type of data structure three important topics are elucidated. First, various ways of representing a structure are explained. Second, the different operations to manage a structure are presented. Finally, the applications of a data structure with focus on its engineering issues are discussed. More than 300 figures have been used to make the discussions comprehensive and lucid. There are numerous section-wise exercises as "Assignment" in each chapter so that the readers can test their understanding of the subject. Also, the problems under the heading "Problems to Ponder" in each chapter, are planned for the advanced readers who can judge their grasp of the subject. A few references are included at the end of each chapter for advanced study.

As prerequisites, the students are expected to have experience of a programming language,

a little understanding of recursive procedures, introductory concepts of compiler, operating system, etc. The students are advised to go for generic implementation of algorithms with C++ so that once the data structures are programmed, they can be subsequently used in many different applications.

It is very difficult to avoid errors completely from a book of this nature. In spite of the immense amount of effort and attention to minute details, some errors might have still crept in. The author would welcome and greatly appreciate suggestions from the readers on making improvements to the book.

I would like to express my sincere gratitude to my friends and colleagues who contributed in many ways towards completion of the present work. I am grateful to the staff of North Eastern Regional Institute of Science and Technology (NERIST), Nirjuli, and Indian Institute of Technology Kharagpur who extended their help during the preparation of this book. My most heartfelt thanks go to Y. Usha for drawing all the figures in the book, and to N. Chetri and P. Kuli for typing the text.

I also wish to express my gratitude to the staff at Prentice-Hall of India, New Delhi for a masterful job of producing the finished volume.

Finally, I thank my wife Monalisa, a faculty member in the Department of Computer Science and Engineering, NERIST, who checked the manuscript with painstaking attention. Her sincere cooperation and involvement has made this work a reality. Our daughter Ananya grudgingly allowed me to sit with the computer to do the work. I affectionately dedicate this work to them.

D. SAMANTA

1 Introduction and Overview

1.1 DEFINITIONS

Before going to study the actual subject, it will be a special advantage if we aware ourselves about the various terms involved in the subject. In the present section, basic terminologies and concepts are described with examples.

Data

Data means value or set of values. In both the singular and plural form, this term is data. Following are some examples of data:

(i) 34
(ii) 12/01/1965
(iii) ISBN 81-203-0000-0
(iv) | | |
(v) Pascal
(vi) Æ
(vii) 21, 25, 28, 30, 35, 37, 38, 41, 43

In all the above examples, some values are represented in different ways. Each value or collection of all such values is termed as data.

Entity

An *entity* is one that has certain attributes and which may be assigned values. For example, an employee in an organisation is an entity. The possible attributes and the corresponding values for an entity in the present example are:

Entity	:	EMPLOYEE			
Attributes	:	NAME	DOB	SEX	DESIGNATION
Values	:	RAVI ANAND	30/12/61	M	DIRECTOR

All the employees in an organisation constitute an *entity set*. Each attribute of an entity set has a range of values, and is called the *domain* of attribute. Domain is the set of all possible values that could be assigned to the particular attribute. For example, in the EMPLOYEE entity, the attribute SEX has domain as {M, F}. It can be noted that, an entity after assigning its values results a composite data.

Information

The term *information* is used for data with its attribute(s). In other words, information can be

defined as meaningful data or processed data. For example, a set of data is presented during defining the term data, all these data become information if we impose the meaning as mentioned below related to a person:

Data	*Meaning*
34	Age of the person
12/01/1965	Date of birth of the person
ISBN 81-203-0000-0	Book number, recently published by the person
I I I	Number of awards achieved by the person in tally mark
Pascal	Nick name of the person
Æ	Signature of the person
21, 25, 28, 30, 35, 37, 38, 41, 43	Important ages of the person

Difference between data and information

From the definition of data and information it is evident that these two are not the same, however, there is a relation between them. Figure 1.1 shows the interrelation between data and information:

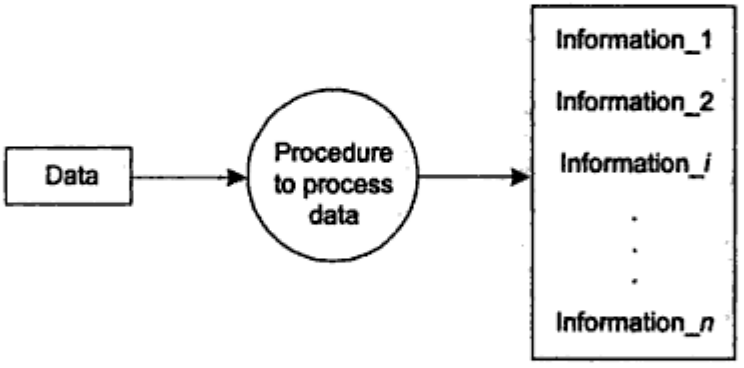


Fig. 1.1 Relation between data and information.

As an example, suppose there is a set of data comprising of the amount of milk consumed by a person in a month. From this given set of data information can be retrieved are as follows:

(i) What is the total amount of milk consumed?
(ii) In which day maximum milk is consumed?
(iii) In which day minimum milk is consumed?
(iv) What is the average amount of milk consumption per day?
(v) What amount of carbohydrate is assimilated?
(vi) What amount of protein is assimilated?
(vii) What amount of fat is assimilated?
(viii) What amount of mineral is assimilated?
(ix) What amount of average calorie is gained per day?

and so on. To get these information from the given set of data, we are to define the processes

and then to apply the corresponding process on the data. (In fact, the terms, *data*, *entity* and *information* have no standard definition in computer science and they are often used interchangeably.)

Data type

A *data type* is a term which refers to the kind of data that may appear in computation. Following are a few well-known data types (*see* table):

Data	*Data type*
34	Numeric (integer)
12/01/1965	Date
ISBN 81-203-0000-0	Alphanumeric
| | |	Graphics
Pascal	String
Æ	Image
21, 25, 28, 30, 35, 37, 38, 41, 43	Array of integers

Real, boolean, character, complex, etc., are also some more frequently used data types.

Built-in data type

With every programming language, there is a set of data types called *built-in data types*. For example, in C, FORTRAN and Pascal, the data types that are available as built-in are listed below:

C : int, float, char, double, enum etc.

FORTRAN : INTEGER, REAL, LOGICAL, COMPLEX, DOUBLE PRECISION, CHARACTER, etc.

Pascal : Integer, Real, Character, Boolean, etc.

With the built-in data types, programming languages allow users a lot of advantages regarding the processing of various types of data. For example, if a user declares a variable of type Real (say), then several things are automatically implied, such as how to store a value for that variable, what are the different operations possible on that type of data, what amount of memory is required to store, etc. All these things are taken care by the compiler or run-time system manager.

Abstract data type

When an application requires a special kind of data which is not available as built-in data type then it is the programmer's burden to implement his own kind of data. Here, the programmer has to give more effort regarding how to store value for that data, what are the operations that meaningfully manipulate variables of that kind of data, amount of memory require to store for a variable. The programmer has to decide all these things and accordingly implement it. Programmers' own data type is termed as *abstract data type*. Abstract data type is also alternatively termed as *user-defined data type*. For example, we want to process dates of the form dd/mm/yy. For

this, no built-in data type is known in C, FORTRAN, and Pascal. If a programmer wants to process dates, then an abstract data type, say Date, can be devised and various operations like: to add few days to a date to obtain another date, to find the days between two dates, to obtain a day for a given date, etc., have to be defined accordingly. Besides these, programmers should decide how to store the data, what amount of memory will be needed to represent a value of Date, etc. An abstract data type, in fact, can be built with the help of built-in data types and other abstract data type(s) already built by the programmer. Some programming languages allow facility to build abstract data type easily. For example, using struct/class in C/C++, and using record in Pascal, programmers can define their own data types.

1.2 CONCEPT OF DATA STRUCTURES

Digital computer can manipulate only primitive data, that is, data in terms of 0's and 1's. Manipulation of primitive data is inherent within the computer and need not require any extra effort from the user side. But in our real life applications, various kind of data other than the primitive data are involved. Manipulation of real-life data (can also be termed as user data) requires the following essential tasks:

1. Storage representation of user data: user data should be stored in such a way that computer can understand it.
2. Retrieval of stored data: data stored in a computer should be retrieved in such a way that user can understand it.
3. Transformation of user data: various operations which require to be performed on user data so that it can be transformed from one form to another.

Basic theory of computer science deals with the manipulation of various kinds of data, wherefrom the concept of data structures comes. In fact, data structure is the fundamentals in computer science. For a given kind of user data, its structure implies the following:

1. Domain ($\mathcal{D}$): This is the range of values that the data may have. This domain is also termed as data object.
2. Function ($\mathcal{F}$): This is the set of operations which may legally be applied to elements of data object. This implies that for a data structure we must specify the set of operations.
3. Axioms ($\mathcal{A}$): This is the set of rules with which the different operation belongs to $\mathcal{F}$ actually can be implemented.

Now we can define the term data structure.

A *data structure* D is a triplet, that is, D = ($\mathcal{D}$, $\mathcal{F}$, $\mathcal{A}$) where $\mathcal{D}$ is a set of data object, $\mathcal{F}$ is a set of functions and $\mathcal{A}$ is a set of rules to implement the functions. Let us consider an example.

We know for the integer data type (int) in C programming language the structure includes of the following type:

$\mathcal{D}$ = (0, ±1, ±2, ±3, ...)

$\mathcal{F}$ = (+, –, *, /, %)

$\mathcal{A}$ = (A set of binary arithmetics to perform addition, subtraction, division, multiplication, and modulo operations.)

It can be easily revealed that the triplet ($\mathcal{D}$, $\mathcal{F}$, $\mathcal{A}$) is nothing but an abstract data type. Also, the elements in set $\mathcal{D}$ are not necessarily from primitive data it may contain from some other abstract data type. Alternatively, an implementation of a data structure D is a mapping from $\mathcal{D}$ to a set of other data structures D_i, $i = 1, 2, ..., n$, for some n. More precisely, this mapping specifies how every object of $\mathcal{D}$ is to be represented by the objects of D_i, $i = 1, 2, ..., n$. Every function of D must be written using the function of the implementing data structures D_i, $i = 1, 2, ..., n$. The fact is that each of the implementing data structures is either primitive data type or abstract data type. We can conclude the discussion with another example.

Suppose, we want to implement a data type namely `complex` as abstract data type. Any variable of complex data type has two parts: real part and imaginary part. In our usual notation, if z is a complex number then $z = x + iy$, where x and y are real and imaginary parts respectively. Both x and y are of `Real` data type which is another abstract data type (available as built-in data type). So, abstract data type `complex` can be defined using the data structure `Real` as:

```
Complex z {
  x : Real
  y : Real
}
```

Now the set $\mathcal{D}$ of `Complex` can be realised from the domain of x and y which is `Real` in this case. Let us specify the set of operations for the `Complex` data type, which are stated as $\mathcal{F}$:

$$\mathcal{F} = (\oplus, \ominus, \otimes, \oslash, \nabla, |\,|)$$

Assume that $z_1 = x_1 + iy_1$ and $z_2 = x_2 + iy_2$ are two data of `Complex` data type. Then we can define the rules for implementing the operations in $\mathcal{F}$ giving thus axioms $\mathcal{A}$. In the current example, for `Complex` data type:

$\mathcal{A}$ = {

$z = z_1 \oplus z_2 = (x_1 + x_2) + i(y_1 + y_2)$ (complex addition)

$z = z_1 \ominus z_2 = (x_1 - x_2) + i(y_1 - y_2)$ (complex subtraction)

$z = z_1 \otimes z_2 = (x_1 \times x_2 - y_1 \times y_2) + i(x_1 \times y_2 + x_2 \times y_1)$ (complex multiplication)

$$z = z_1 \oslash z_2 = \frac{x_1 \times x_2 + y_1 \times y_2}{x_2^2 + y_2^2} + i\frac{x_2 \times y_1 - x_1 \times y_2}{x_2^2 + y_2^2}$$ (complex division)

$$z = \nabla z_1 = \frac{x_1}{x_1^2 + y_1^2} - i\frac{y_1}{x_1^2 + y_1^2}$$ (complex conjugate)

$$Z = |z_1| = \sqrt{x_1^2 + y_1^2}$$ (complex magnitude)

}

Note that, how different operations of `Complex` data type can be implemented using the operations +, –, ×, /, $\sqrt{\ }$, of implementing data structure, namely `Real`.

Assignment 1.1 Implement *Date* as an abstract data type which comprises of dd/mm/yy, where dd varies from 1 to 31, mm varies from 1 to 12 and yy is any integer with four digits.

Specify $\mathcal{F}$ consisting of all possible operations on variables of type *Date* and then define $\mathcal{A}$ to implement all the operations in $\mathcal{F}$. Assume the implementing data structure(s) which is/are necessary.

1.3 OVERVIEW OF DATA STRUCTURES

In computer science, several data structures are known depending on area of applications. Of them, few data structures are there which are frequently used almost in all application areas and with the help of which almost all complex data structure can be constructed. These data structures are known as *fundamental* data structures or *classic* data structures. Figure 1.2 gives a classification of all classic data structures.

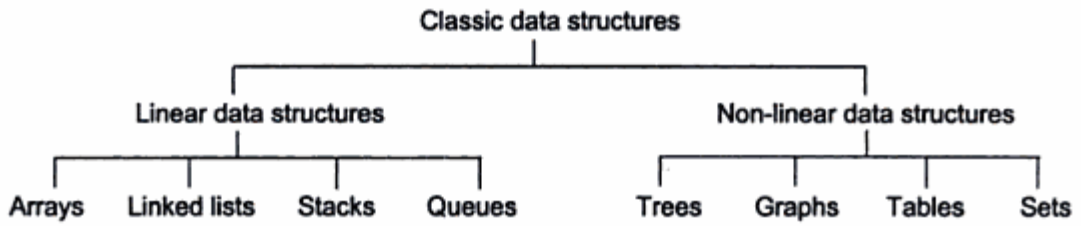


Fig. 1.2 Classification of classic data structures.

In addition to these classic data structures, other data structures such as lattice, Petri nets, neural nets, search graphs, semantic nets, etc., are known in various applications. These are known to be very complex data structures. (Discussion of all these complicated data structures is beyond the scope of this book.)

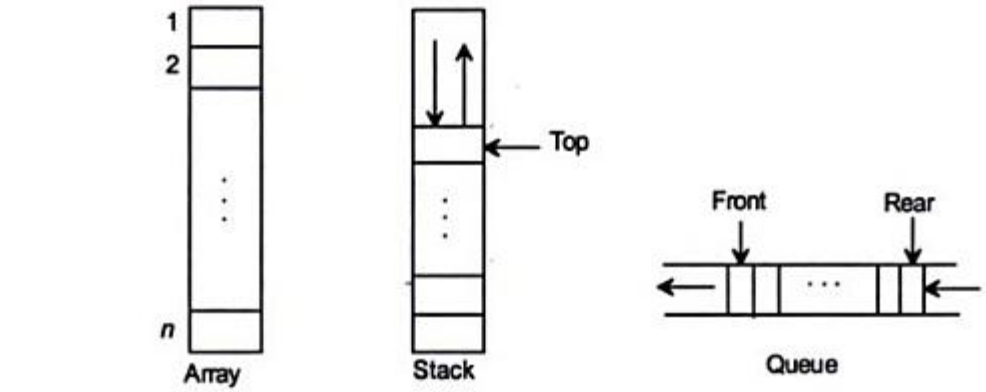
It can be observed that all the classic data structures are classified into two main classes: *linear data structures* and *non-linear data structures*. In case of linear data structures, all the elements form a sequence or maintain a linear ordering. On the other hand, no such sequence in elements, rather all the elements are distributed over a plane in case of non-linear data structures. Again within each classical data structure there are a number of variations as depicted in Figure 1.2. In this book, we have planned for eight chapters to discuss eight classic data structures separately. As an overview, all the classic data structures are primarily introduced which can be seen in Figure 1.3.

Assignment 1.2 Devise a data structure (say Graphic) to represent a graphics data as shown in Figure 1.4. Also, define the following operation on any data of this type:

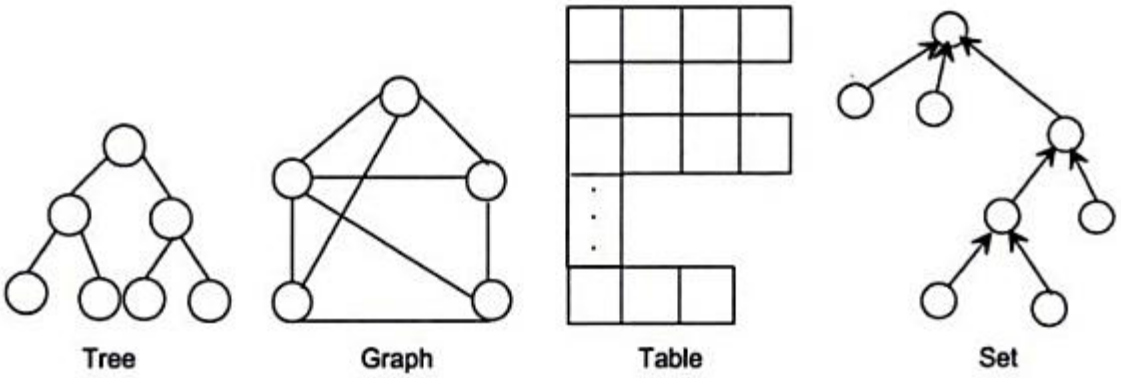
(a) ScaleUp(P): To magnify the image by certain per cent P

(b) ScaleDn(P): To reduce the image by certain per cent P

(c) Rotate(R, A): To rotate the image either clockwise (R = +) or anticlockwise (R = −) by a certain angle A.



(a) Linear data structures



(b) Non-linear data structures

Fig. 1.3 Overview of classic data structures.

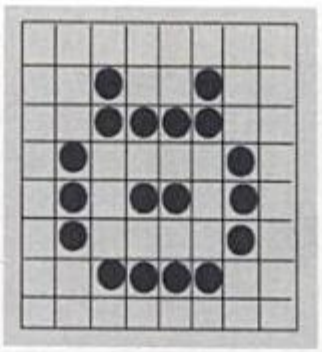


Fig. 1.4

1.4 IMPLEMENTATION OF DATA STRUCTURES

We will implement all the classic data structures as mentioned in the previous section in two phases:

Phase 1

Storage representation: Here we have decided how a data structure can be stored in computer memory. This storage representation, in general, is based on the use of other data structures.

Phase 2

Algorithmic notation of various operations of the classic data structure: Function for manipulating a data structure is expressed in terms of algorithms so that details of the operation can be understood easily and reader can implement them with the help of any programming language. It will be preferable to use C/C++ programming because of their versatile features and modern programming approaches.

In order to express the algorithm for a given operation, we will assume different control structures and notations as stated below:

Algorithm <Name of the operation>(Input parameters; Output parameters)
Input: <Specification of input data for the operation>
Output: <Specification of output after the successful performance of the operation>
Data structure: <If the operation assumes other data structure for its implementation>

Steps:

1. . . . . .
2. **If** <condition> **then**
   1. . . . . .
   2. . . . . .
   . . .
   . . .
3. **Else**
   1. . . . . .
   2. . . . . .
   . . .
   . . .
4. **EndIf**
5. . . .
6. **While** <condition> **do**
   1. . . . . .
   2. . . . . .
   . . .
   . . .
7. **EndWhile**
8. . . .

9. **For** <loop condition> **do**
   1. . . . . .
   2. . . . . .
   . . .
   . . .
10. **EndFor**
11. . . .
12. **Stop**

Different statements that will be used in the algorithms are very much similar to the statement in C language. Sufficient comment will be introduced side by side against statements to indicate what step that a statement will perform. The convention of putting comment is similar to the convention of placing comments in C/C++ programming language.

Another convention that we will assume is the naming of variables. The variable names are either in capital or small letters. The variable names in capital letters will be treated as either constant or as input to the algorithm. All the variable names in small letters are local to the algorithm and will be treated as temporary variables.

'Problems to Ponder' at the end of each chapter includes problems related to the topics discussed in the concerned chapter. In addition to, a number of 'Assignments' are also provided during the discussion in each chapter. Students can improve their level of learning if they practise these assignments and solve problems.

2 Arrays

If we want to store a group of data together in one place then array is the one data structure we are looking for. This data structure enables us to arrange more than one element, that is why it is termed as composite data structure. In this data structure, all the elements are stored in contiguous locations of memory. Figure 2.1 shows an array of data stored in a memory block starting at location 453.

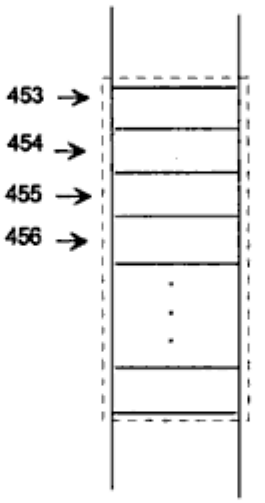


Fig. 2.1 Array of data.

2.1 DEFINITION

An *array* is a finite, ordered and collection of homogeneous data elements. Array is finite because it contains only limited number of elements; and ordered, as all the elements are stored one by one in contiguous locations of computer memory in a linear ordered fashion. All the elements of an array are of the same data type (say, integer) only and hence it is termed as collection of homogeneous elements. Following are some examples:

1. An array of integers to store the age of all students in a class.
2. An array of strings (of characters) to store the name of all villagers in a village.

Note: 1. An array is known as linear data structure because, all elements of the array are stored in a linear order.

2. The data structure is generally a built-in data type in all programming languages. Declaration syntax for an array A, say, of 100 integers in BASIC, FORTRAN, Pascal and C are as below:

BASIC : DIMENSION A[100]
FORTRAN : DIM A[100]
Pascal : A: ARRAY[1...100] of integer
C : int A[100]

2.2 TERMINOLOGY

Size. Number of elements in an array is called the *size* of the array. It is also alternatively termed as *length* or *dimension*.

Type. *Type* of an array represents the kind of data type it is meant for. For example, array of integers, array of character strings, etc.

Base. *Base* of an array is the address of memory location where the first element in the array is located. For example, 453 is the base address of the array as mentioned in Figure 2.1.

Index. All the elements in an array can be referenced by a subscript like A_i or A[i], this subscript is known as *index*. Index is always an integer value. As each array elements is identified by a subscript or index that is why an array element is also termed as *subscripted* or *indexed variable*.

Range of index. Indices of array elements may change from a lower bound (L) to an upper bound (U), which are called the boundaries of an array.

In a declaration of an array in FORTRAN (DIMENSION A[100]), range of index is 1 to 100. For the same array in C (int A[100]) the range of index is from 0 to 99. These are all default range of indices. However in Pascal, a user can define the range of index for any lower bound to upper bound, for example, for A: ARRAY[–5...19] of integer, the points of the range is –5, –4, –3, ..., 18, 19. Here, the index of i-th element is $-5 + i - 1$. In terms of L, the lower bound, this formula stands as:

$$\text{Index } (A_i) = L + i - 1$$

If the range of index varies from L ... U then the size of the array can be calculated as

$$\text{Size } (A) = U - L + 1$$

Word. Word denotes the size of an element. In each memory location, computer can store an element of word size w, say. This word size varies from machine to machine such as 1 byte (in IBM CPU-8085) to 8 bytes (in Intel Pentium PCs). Thus, if the size of an element is double the word size of a machine then to store such an element, it requires two consecutive memory locations.

2.3 ONE-DIMENSIONAL ARRAY

If only one subscript/index is required to reference all the elements in an array then the array will be termed as *one-dimensional* array or simply an array.

2.3.1 Memory Allocation for an Array

Memory representation of an array is very simple. Suppose, an array A[100] is to be stored in a memory as in Figure 2.2. Let the memory location where the first element can be stored is M. If each element requires one word then the location for any element say A[i] in the array can be obtained as:

$$\text{Address (A}[i]) = M + (i - 1)$$

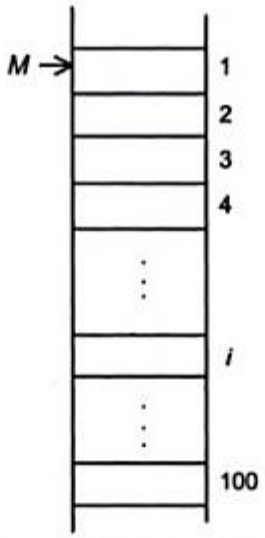


Fig. 2.2 Physical representation of a one-dimensional array.

Likewise, in general, an array can be written as A[L ... U], where L and U denote the lower and upper bounds for index. If it is stored starting from memory location M, and for each element it requires w number of words, then the address for A[i] will be

$$\text{Address (A}[i]) = M + (i - L) \times w$$

The above formula is known as *indexing formula*; which is used to map the logical presentation of an array to physical presentation. By knowing the starting address of an array M, the location of i-th element can be calculated instead of moving towards i from M, Figure 2.3.

Assignment 2.1 Suppose, an array A [–15 ... 64] is stored in a memory whose starting address is 459. Assume that word size for each element is 2. Then obtain the following:

(a) How many number of elements are there in the array A?

(b) If one word of the memory is equal to 2 bytes, then how much memory is required to store the entire array?

(c) What is the location for A[50]?

(d) What is the location of 10-th element?

(e) Which element is located at 589?

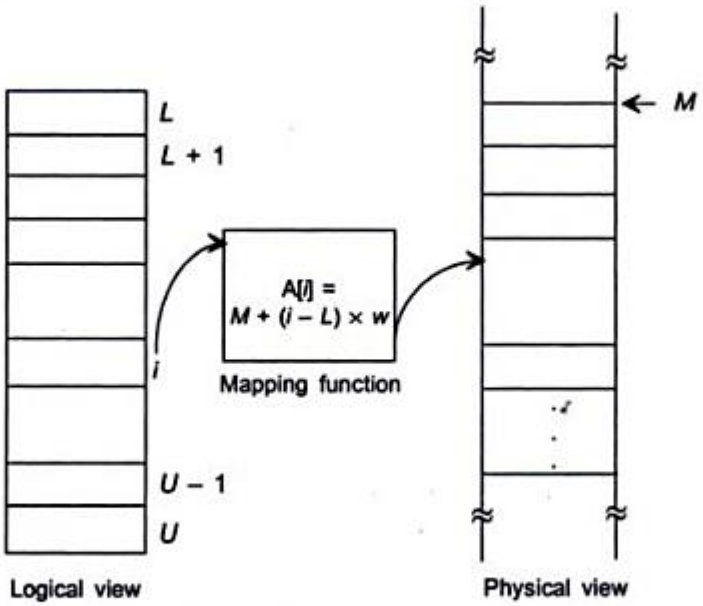


Fig. 2.3 Address mapping between logical and physical views of an array.

Assignment 2.2 In the indexing formula in Section 2.3.1, we have taken that the array is stored from lower region of the memory to the higher region. But in some computer the convention is just reverse, that is, starting an array i at higher region and hence Address (A_i) > Address (A_{i+1}) for all $L < i < U$. Modify the indexing formula for this case.

2.3.2 Operations on Arrays

Various operations that can be performed on an array are: traversing, sorting, searching, insertion, deletion, merging.

Traversing

This operation is used visiting all elements in an array. A simplified algorithm is presented as below:

Algorithm TRAVERSE_ARRAY( )
Input: An array A with elements.
Output: According to PROCESS( ).
Data structures: Array A[L ... U]. //L and U are the lower and upper bound //of array index

Steps:
1. $i = L$ //Start from first location L
2. While $i \leq U$ do
   1. PROCESS(A[i])
   2. $i = i + 1$ //Move to the next location
3. EndWhile
4. Stop

Note: Here PROCESS( ) is a procedure which when called for an element can perform an action. For example, display the element on the screen, determine whether A[*i*] is empty or not, etc. PROCESS( ) also can be used to manipulate some special operations like count the special element of interest (for example, negative numbers in an integer array), updating of each element (say, increase each by 10%) and so on.

Sorting

This operation, if performed on an array, will sort it in a specified order (ascending/descending). The following algorithm is used to store the elements of an integer array in ascending order.

Algorithm SORT_ARRAY( )
Input: An array with integer data.
Output: An array with sorted elements in an order according to ORDER( ).
Data structures: An integer array A[*L*...*U*]. //*L* and *U* are the lower and upper bound of //array index

Steps:
1. $i = U$
2. While $i \geq L$ do
   1. $j = L$ //Start comparing from first
   2. While $j < i$ do
      1. If ORDER(A [*j*], A[*j* + 1]) = FALSE //If A[*j*] and A[*j* + 1] are not in order
         1. SWAP(A[*j*], A[*j* + 1]) //Interchange the elements (*see* Figure 2.4)
      2. EndIf
      3. $j = j + 1$ //Go to next element
   3. EndWhile
   4. $i = i - 1$
3. EndWhile
4. Stop

Here, ORDER(...) is a procedure to test whether two elements are in order and SWAP(...) is a procedure to interchange the elements between two consecutive locations.

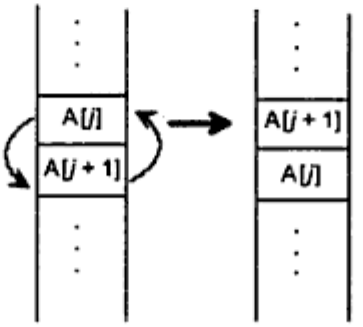


Fig. 2.4 Swapping of two elements in an array.

Assignment 2.3 Write algorithms to

(a) Sort an array of integers in ascending order.
(b) Sort an array of integers in descending order.
(c) Sort an array of string of characters in lexicographic order.
(d) Sort an array of an abstract data type.

Hint: You have to think of only little modification of procedure ORDER(...) in each case.

Searching

This operation is applied to search an element of interest in an array. The simplified version of the algorithm is as follows:

Algorithm SEARCH_ARRAY(KEY)
Input: KEY is the element to be searched.
Output: Index of KEY in A or a message on failure.
Data structures: An array A[*L* ... *U*]. //*L* and *U* are the lower and upper bound of array index

Steps:

1. $i = L$, found = 0, location = 0 //found = 0 indicates search is not finished and unsuccessful
2. While ($i \leq U$) and (found = 0) do // Continue if all or any one condition do(es) not satisfy
   1. If COMPARE(A[*i*], KEY) = TRUE then //If key is found
      1. found = 1 //Search is finished and successful
      2. location = *i*
   2. Else
      1. $i = i + 1$ //Move to the next
   3. EndIf
3. EndWhile
4. If found = 0 then
   1. Print "Search is unsuccessful : KEY is not in the array"
5. Else
   1. Print "Search is successful : KEY is in the array at location", location
6. EndIf
7. Return(location)
8. Stop

Assignment 2.4 In the algorithm discussed above assume that elements are randomly distributed. Modify the algorithm to give more faster algorithm when the elements are already in sorted (either ascending or descending) order.

Insertion

This operation is used to insert an element into an array provided that the array is not full (*see* Figure 2.5). Let us observe the algorithmic description for the insertion operation.

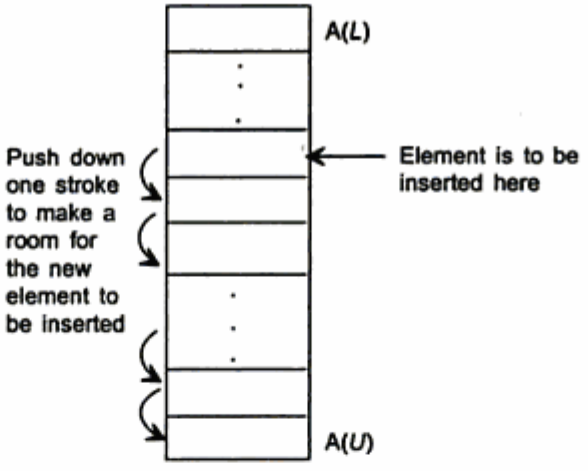


Fig. 2.5 Insertion of an element to an array.

Algorithm INSERT(KEY, LOCATION)
Input: KEY is the item, LOCATION is the index of the element where it is to be inserted.
Output: Array enriched with KEY.
Data structures: An array $A[L \ldots U]$. //L and U are the lower and upper bound of array index

Steps:
1. If A[U] ≠ NULL then
   1. Print "Array is full: No insertion possible"
   2. Exit //End of execution
2. Else
   1. $i = U$ //Start pushing from bottom
   2. While $i >$ LOCATION do
      1. A[$i + 1$] = A[i]
      2. $i = i - 1$
   3. EndWhile
   4. A[LOCATION] = KEY //Put the element at desired location
   5. $U = U + 1$ //Update the upper index of the array
3. EndIf
4. Stop

Assignment 2.5 The algorithm INSERT only checks the last element for vacancy. But an array may be empty from any i-th position ($L \leq i \leq U$); in that case number of push down can be reduced instead of pushing down the entire trailing part. Modify the above algorithm INSERT when the last element is at i-th location. ($i \leq U$).

Assignment 2.6 How can an empty array be defined? Verify that whether all the algorithms discussed so far in this chapter can work with empty array. If not, modify algorithm(s) so that they can work even for empty array(s).

Deletion

This operation is used to delete a particular element from an array. The element will be deleted by overwriting it with its subsequent element and this subsequent element then is to be deleted. In other words, push the tail (the part of the array after the elements which are to be deleted) one stroke up. Consider the following algorithm to delete an element from an array:

Algorithm DELETE(KEY)
Input: KEY the element to be deleted.
Output: Slimed array without KEY.
Data structures: An array A[*L*...*U*]. //*L* and *U* are the lower and upper bound of //array index

Steps:
1. i = SEARCH_ARRAY(A, KEY) //Perform the search operation on A and return
2. If (i = 0) then //the location
   1. Print "KEY is not found: No deletion"
   2. Exit //Exit the program
3. Else
   1. While $i < U$ do
      1. A[i] = A[$i + 1$] //Replace the element by its successor
      2. $i = i + 1$
   2. EndWhile
4. EndIf
5. A[U] = NULL //The bottom most element is made empty //(*see* Figure 2.6)
6. $U = U - 1$ //Updated the upper bound now
7. Stop

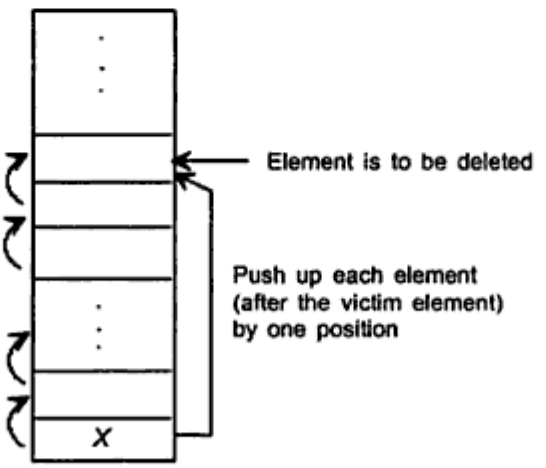


Fig. 2.6 Deletion of an element from an array.

Note: It is a general practice that no intermediate location will be made empty, that is, an array should be packed and empty locations are at the tail of an array.

Merging

Merging is an important operation when we need to compact the elements from two different arrays into a single array (*see* Figure 2.7).

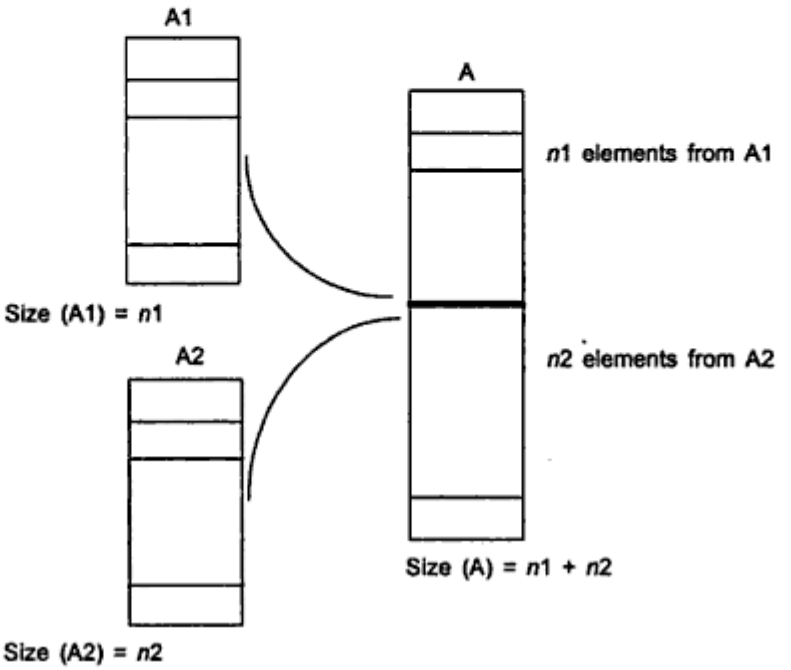


Fig. 2.7 Merging of A_1 and A_2 to A.

Algorithm MERGE(A_1, A_2; A)
Input: Two arrays $A_1[L_1 \ldots U_1]$, $A_2[L_2 \ldots U_2]$.
Output: Resultant array $A[L \ldots U]$, where $L = L_1$, and $U = U_1 + (U_2 - L_2 + 1)$ when A_1 is appended after A_2.
Data structures: Array structure.

Steps:

1. $i_1 = L_1$, $i_2 = L_2$, //Initialization of control variables
2. $L = L_1$, $U = U_1 + U_2 - L_2 + 1$ //Initialization of lower and upper bounds of //resultant array A
3. $i = L$
4. Allocate memory for $A[L \ldots U]$
5. While $i_1 \leq U_1$ do //To copy array A_1 into the first part of A
   1. $A[i] = A_1[i_1]$
   2. $i = i + 1$, $i_1 = i_1 + 1$
6. EndWhile
7. While $i_2 \leq U_2$ do //To copy the array A_2 into last part of A
   1. $A[i] = A_2[i_2]$
   2. $i = i + 1$, $i_2 = i_2 + 1$
8. EndWhile
9. Stop

Assignment 2.7 Modify the algorithm MERGE for the following cases of (i) if both A_1 and A_2 are empty; (ii) either A_1 or A_2 is empty.

2.3.3 Application of Arrays

There are wide applications of arrays in computation. This is why almost every programming language include this data type as a built-in data type. A simple example that can be cited is to store some data before performing their manipulation. Another simple example is mentioned as below:

Suppose, we want to store records of all students in a class. The record structure is given by

STUDENTS

ROLL NO	MARK1	MARK2	MARK3	TOTAL	GRADE
(Alphanumeric)	(Numeric)	(Numeric)	(Numeric)	(Numeric)	(Character)

If sequential storage of records is not an objection, then we can store the records by maintaining 6 arrays whose size is specified by the total number of students in the class, as in Figure 2.8.

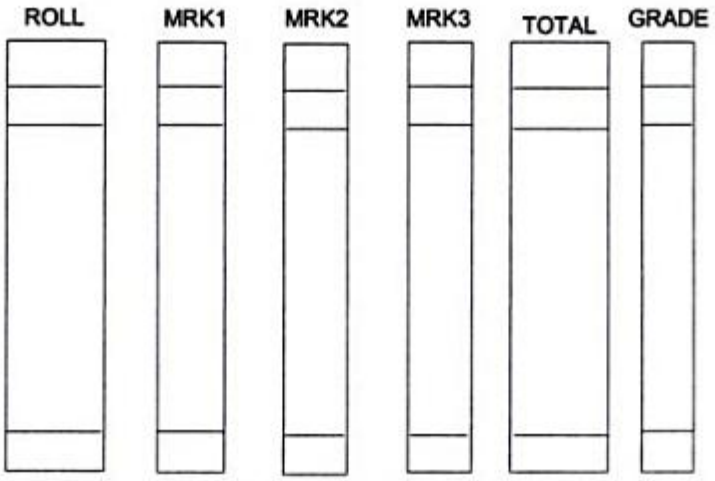


Fig. 2.8 Storing the students' records.

Assignment 2.8 (Practical)

1. Declare 6 arrays as required to store the students records each of size say, 30.
2. Initialize each array with appropriate data.
3. Use the following operations on your data structures
   (a) List all the records on the screen
   (b) Search for a record whose Roll No is say, X
   (c) Delete a record whose Roll No is say, X
   (d) Insert a new records just after the record, whose Roll No = X

(e) Sort all the records according to the descending order of TOTAL.
(f) Suppose, we want to merge the records of two classes. Obtain the merging in such a case.

Note: Carry out all the above tasks using the operations those already defined for array.

4. Give an idea how a polynomial with three variables, say X, Y and Z can be represented using a one-dimensional array.

2.4 MULTIDIMENSIONAL ARRAYS

So far we have discussed the one dimensional arrays. But multidimensional arrays are also important. Matrix (2- dimensional array), 3-dimensional array are two examples of multidimensional arrays. The following sections describe the multidimensional arrays.

2.4.1 Two-dimensional Arrays

Two-dimensional arrays (alternatively termed as matrices) are the collection of homogeneous elements where the elements are ordered in a number of rows and columns. An example of an $m \times n$ matrix where m denotes number of rows and n denotes number of columns is as follows:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & a_{24} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & a_{m3} & a_{m4} & \cdots & a_{mn} \end{bmatrix}_{m \times n}$$

The subscripts of any arbitrary element, say (a_{ij}) represent the i-th row and j-th column.

Memory representation of a matrix

Like one-dimensional array, matrices are also stored in contagious memory locations. There are two conventions of storing any matrix in memory:

1. Row-major order
2. Column-major order.

In row-major order, elements of a matrix are stored on a row-by-row basis, that is, all the elements in first row, then in second row and so on. On the other hand, in column-major order, elements are stored column-by-column, that is, all the elements in first column are stored in their order of rows, then in second column, third column and so on. For example, consider a matrix A of order 3 × 4:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \end{bmatrix}_{3 \times 4}$$

This matrix can be represented in memory as shown in Figure 2.9.

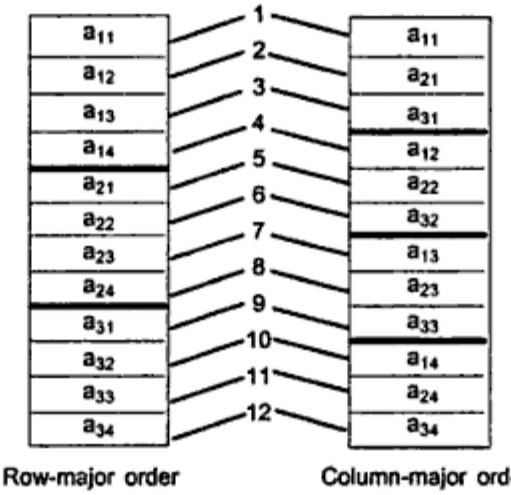


Fig. 2.9 Memory representation of $A_{3\times4}$ matrix.

Reference of elements in a matrix

Logically, a matrix appears as two-dimensional; but physically it is stored in a linear fashion. So, in order to map from logical view to physical structure, we need indexing formula. Obviously, the indexing formula for different order will be different. The indexing formulae, for different orders are stated as below:

Row-major order. Assume that the base address is the first location of the memory, that is, 1. So, the address of a_{ij} will be obtained as

$$\begin{aligned}\text{Address } (a_{ij}) &= \text{Storing all the elements in first } (i-1)\text{-th rows} \\ &\quad + \text{The number of elements in } i\text{-th row up to } j\text{-th column.} \\ &= (i-1)\times n + j\end{aligned}$$

So, for the matrix $A_{3\times4}$, the location of a_{32} will be calculated as 10 (see Figure 2.9). Instead of considering the base address to be 1, if it is at M, then the above formula can be easily modified as:

$$\text{Address } (a_{ij}) = M + (i-1)\times n + j - 1$$

Column-major order. Let us first consider the starting location of matrix is at memory location 1. Then

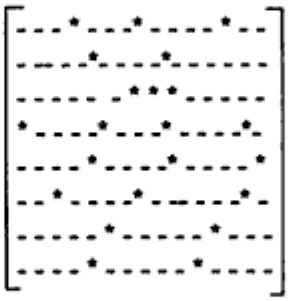
$$\begin{aligned}\text{Address } (a_{ij}) &= \text{Storing all the elements in first } (j-1)\text{-th columns} \\ &\quad + \text{The number of elements in } j\text{-th column up to } i\text{-th rows.} \\ &= (j-1)\times m + i\end{aligned}$$

And considering the base address at M instead of 1, the above formula will stand as

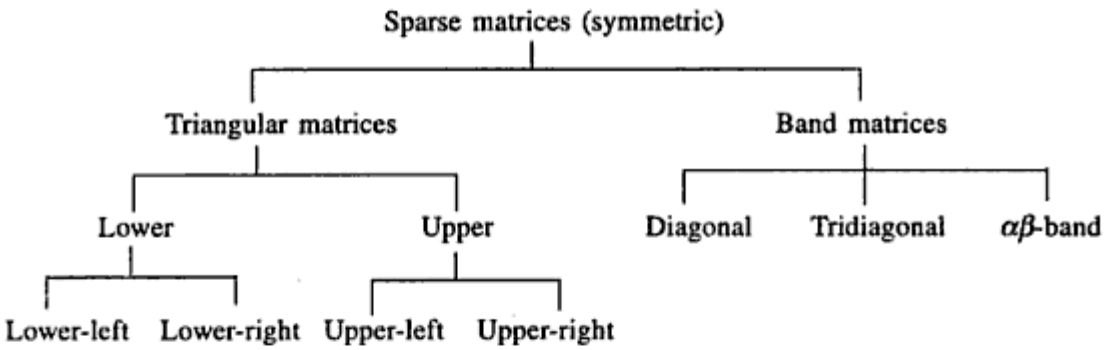
$$\text{Address } (a_{ij}) = M + (j-1)\times m + i - 1$$

2.4.2 Sparse Matrices

A *sparse* matrix is a two-dimensional array having the value of majority elements as null. Following is a sparse matrix where '*' denotes the elements having non-null values.



In large number of applications, sparse matrices are involved. So far storage of a sparse matrix is concerned, storing of null elements is nothing but wastage of memory. So we should devise a technique such that only non-null elements will be stored. One approach is to use the linked list (this will be discussed in Chapter 3, Section 3.6.1). Some well-known sparse matrices which are symmetric in form can be classified as follows:

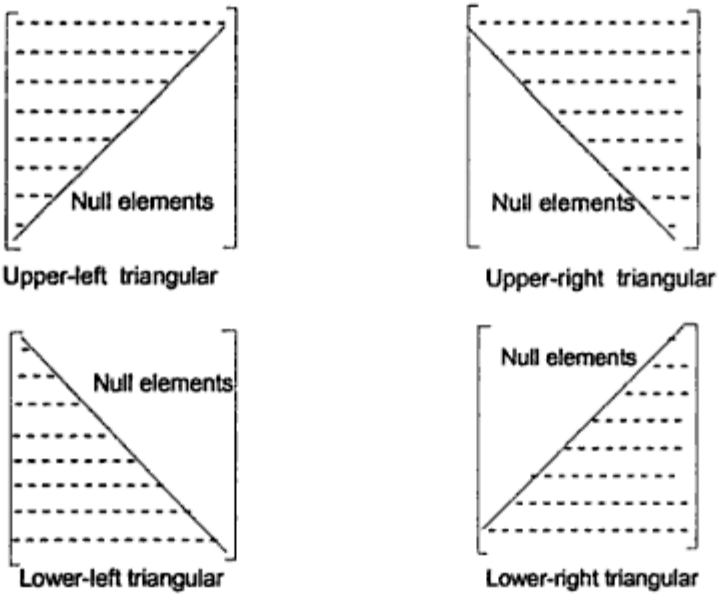


Figures 2.10(a) and 2.10(b) show the general views of the above mentioned sparse matrices. Let us discuss the memory representation some of the above forms.

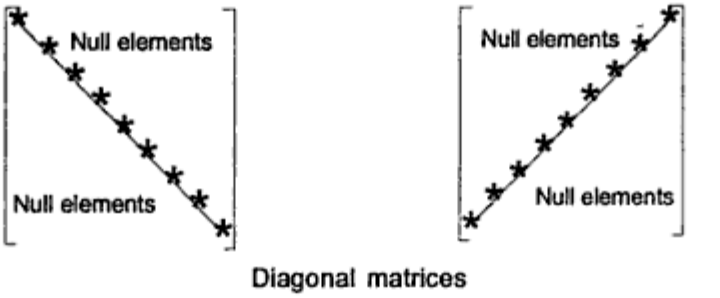
Memory representation of lower-triangular matrix

Consider the following lower-triangular matrix:

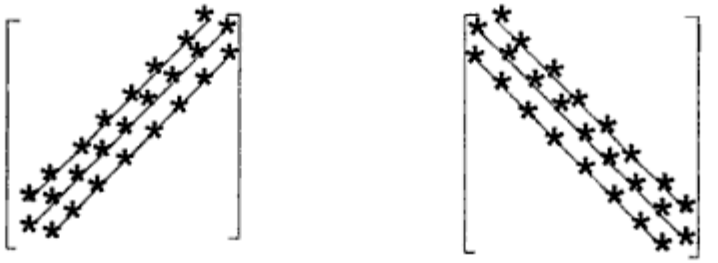
$$\begin{bmatrix} a_{11} & & & & \\ a_{21} & a_{22} & & & \\ a_{31} & a_{32} & a_{33} & & \\ \vdots & \vdots & \vdots & & \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{bmatrix}_{n \times n}$$



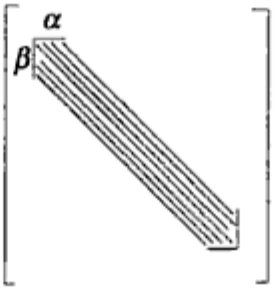
(a) Various triangular matrices



Diagonal matrices



Tridiagonal matrices



$\alpha\beta$-band matrix: non-null elements are only on α-upper diagonal and β-lower diagonal matrix

(b) Various diagonal matrices

Fig. 2.10 Different symmetric sparse matrices.

Row-major order. According to row-major order, the address of any element a_{ij}, $1 \le i, j \le n$ can be obtained as

$$\begin{aligned}\text{Address } (a_{ij}) &= \text{Number of elements up to } a_{ij} \text{ element} \\ &= \text{Total number of elements in first } i-1 \text{ rows} \\ &\quad + \text{Number of elements up to } j\text{-th column in the } i\text{-th row} \\ &= 1 + 2 + 3 + \ldots + (i-1) + j \\ &= \frac{i(i-1)}{2} + j\end{aligned}$$

If the starting location of the first element, that is, of a_{11} is M, then the address of a_{ij}, $1 \le i, j \le n$ will be

$$\text{Address } (a_{ij}) = M + \frac{i(i-1)}{2} + j - 1$$

Column-major order. According to column-major order, the address of any element a_{ij}, $1 \le i, j \le n$ can be obtained as

$$\begin{aligned}\text{Address } (a_{ij}) &= \text{Number of elements up to } a_{ij} \text{ element} \\ &= \text{Total number of elements in first } j-1 \text{ columns} \\ &\quad + \text{Number of elements up to } i\text{-th row in the } j\text{-th column} \\ &= [n + (n-1) + (n-2) + \ldots + (n-j+2)] + (i-j+1) \\ &= \{n \times (j-1) - [1 + 2 + 3 + \ldots + (j-2) + (j-1)] + i\} \\ &= n \times (j-1) - \frac{j(j-1)}{2} + i \\ &= (j-1) \times \left(n - \frac{j}{2}\right) + i\end{aligned}$$

If the starting location of the first element (that is, of a_{11}) is M then the address of a_{ij}, $1 \le i, j \le n$ will be

$$\text{Address } (a_{ij}) = M + (j-1) \times \left(n - \frac{j}{2}\right) + i - 1$$

Memory representation of upper-triangular matrix

As an another example, consider the following form of a upper-triangular matrix:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \ldots & a_{1n} \\ & a_{22} & a_{23} & \ldots & a_{2n} \\ & & a_{33} & \ldots & a_{3n} \\ & & & & \vdots \\ & & & & a_{nn} \end{bmatrix}_{n \times n}$$

Row-major order. According to row-major order, the address of any element a_{ij}, $1 \le i, j \le n$ can be obtained as

$$\begin{aligned}
\text{Address } (a_{ij}) &= \text{Number of elements up to } a_{ij} \text{ element} \\
&= \text{Total number of elements in first } (i-1) \text{ rows} \\
&\quad + \text{Number of elements up to } j\text{-th column in the } i\text{-th row} \\
&= n + (n-1) + (n-2) + \cdots + (n-i+2) + (j-i+1) \\
&= n \times (i-1) - [1 + 2 + 3 + \cdots + (i-2) + (i-1)] + j \\
&= n \times (i-1) - \frac{i(i-1)}{2} + i \\
&= (i-1) \times \left(n - \frac{i}{2}\right) + j
\end{aligned}$$

If the starting location of the first element, i.e. of a_{11} is M, then the address of a_{ij}, $1 \le i, j \le n$ will be

$$\text{Address } (a_{ij}) = M + (i-1) \times \left(n - \frac{i}{2}\right) + j - 1$$

Column-major order. According to column-major order, the address of any element a_{ij}, $1 \le i, j \le n$ can be obtained as

$$\begin{aligned}
\text{Address } (a_{ij}) &= \text{Number of elements up to } a_{ij} \text{ element} \\
&= \text{Total number of elements in first } (j-1) \text{ columns} \\
&\quad + \text{Number of elements up to } i\text{-th row in the } j\text{-th column} \\
&= [1 + 2 + 3 + \cdots + (j-1)] + i \\
&= \frac{j(j-1)}{2} + i
\end{aligned}$$

If the starting location of the first element, i.e. of a_{11} is M, then the address of a_{ij}, $1 \le i, j \le n$ will be

$$\text{Address } (a_{ij}) = M + \frac{j(j-1)}{2} + i - 1$$

Assignment 2.9

(a) How many elements are there in a triangular sparse matrix of order $n \times n$.

(b) Obtain the indexing formula for lower-right and upper-left triangular matrices using row-major and column-major order, consider the cases of

(i) square matrix of order $n \times n$

(ii) non-square matrix of order $m \times n$, $m \ne n$

Memory representation of diagonal matrix

In the sparse matrices having the elements only on diagonal following points are evident:

Number of elements in a $n \times n$ square diagonal matrix = n.

Any element a_{ij} can be referred in memory using the formula

$$\text{Address } (a_{ij}) = i[\text{or } j]$$

One can easily verify that the above formula is same in both row-major and column-major order.

Memory representation of tridiagonal matrix

Let us consider the following tridiagonal matrix:

$$\begin{bmatrix} a_{11} & a_{12} & & & & & \\ a_{21} & a_{22} & a_{23} & & & & \\ & a_{32} & a_{33} & a_{34} & & & \\ & & a_{43} & a_{44} & a_{45} & & \\ & & & \vdots & & & \\ & & & & \vdots & & \\ & & & & a_{(n-1)(n-2)} & a_{(n-1)(n-1)} & a_{(n-1)n} \\ & & & & & a_{n(n-1)} & a_{nn} \end{bmatrix}$$

Row-major order. According to row-major order, the address of any element a_{ij}, $1 \le i, j \le n$ can be obtained as

$$\begin{aligned} \text{Address } (a_{ij}) &= \text{Number of elements up to } a_{ij} \text{ element} \\ &= \text{Total number of elements in first } (i-1) \text{ rows} \\ &\quad + \text{Number of elements up to } j\text{-th column in the } i\text{-th row} \\ &= \{2 + [3 + 3 + \cdots + \text{up to } (i-2) \text{ terms}]\} + (j - i + 2) \\ &= 2 + (i-2) \times 3 + j - (i-2) \\ &= 2 + 2 \times (i-2) + j \end{aligned}$$

If the starting location of the first element, i.e. of a_{11} is M then the address of a_{ij}, $1 \le i, j \le n$ will be

$$\text{Address } (a_{ij}) = M + 2 \times (i-2) + j + 1$$

Column-major order. According to column-major order, the address of any element a_{ij}, $1 \le i, j \le n$ can be obtained as

$$\begin{aligned} \text{Address } (a_{ij}) &= \text{Number of elements up to } a_{ij} \text{ element} \\ &= \text{Total number of elements in first } (j-1) \text{ columns} \\ &\quad + \text{Number of elements up to } i\text{-th row in the } j\text{-th column} \\ &= \{2 + [3 + 3 + \cdots + \text{up to } (j-2) \text{ terms}]\} + (i - j + 2) \\ &= 2 + (j-2) \times 3 + i - (j-2) \\ &= 2 + 2 \times (j-2) + i \end{aligned}$$

If the starting location of the first element, i.e. of a_{11} is M then the address of a_{ij}, $1 \le i, j \le n$ will be

$$\text{Address } (a_{ij}) = M + 2 \times (j - 2) + i + 1$$

Note: The formulas for row-major/column-major order is symmetric and one can be obtained from the other by interchanging i and j.

Assignment 2.10

(a) Obtain the number of elements in a $(n \times n)$ square tridiagonal matrix.

(b) What will be the indexing formula for other form of tridiagonal matrix?

Memory representation for αβ-band matrix

Let us consider a matrix as in Figure 2.11:

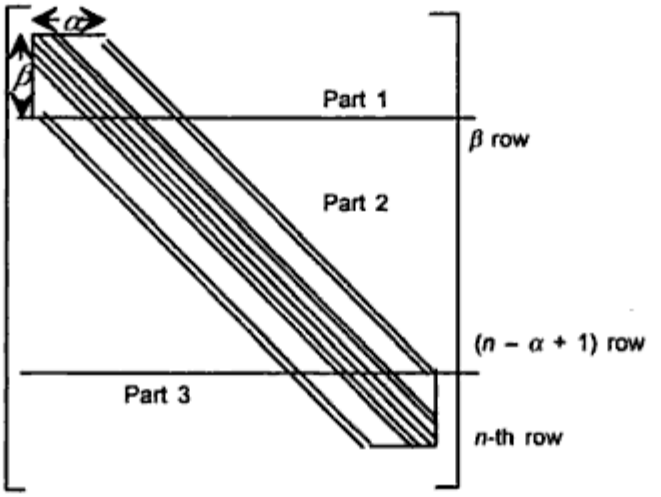


Fig. 2.11 $\alpha\beta$-band matrix.

Note that there are $(\alpha - 1)$ and $(\beta - 1)$ sub-diagonals above and below the main diagonal respectively. As there are three possible arrangements so elements in each arrangement can be referred by three indexing formulae corresponding to Part 1, Part 2, and Part 3 as shown in the Figure 2.11.

Considering the row-major ordering for the memory allocation, the indexing formula is explained as below:

Case 1: $1 \le i \le \beta$

$$\begin{aligned}\text{Address } (a_{ij}) &= \text{Number of elements in first } (i-1)\text{-th rows} \\ &\quad + \text{Number of elements in } i\text{-th row up to } j\text{-th column} \\ &= \alpha + (\alpha + 1) + (\alpha + 2) + \cdots + (\alpha + i - 2) + j \\ &= \alpha \times (i - 1) + [1 + 2 + 3 + \cdots + (i - 2)] + j \\ &= \alpha \times (i - 1) + \frac{(i-1)(i-2)}{2} + j\end{aligned}$$

Case 2: $\beta < i \leq n - \alpha + 1$

Address (a_{ij}) = Number of elements in first β rows
+ Number of elements between $(\beta + 1)$-th row and $(i - 1)$-th row
+ Number of elements in i-th row

$$= \alpha + (\alpha + 1) + (\alpha + 2) + \cdots + (\alpha + \beta - 1) + (\alpha + \beta - 1) \times (i - \beta - 1) + j - i + \beta$$

$$= \alpha\beta + \frac{\beta(\beta - 1)}{2} + (\alpha + \beta - 1)(i - \beta - 1) + j - i + \beta$$

Case 3: $n - \alpha + 1 < i$

Address (a_{ij}) = Number of elements in first $(n - \alpha + 1)$ rows
+ Number of elements after $(n - \alpha + 1)$-th row and up to $(i - 1)$-th row
+ Number of elements in i-th row and up to j-th column

$$= \alpha\beta + \frac{\beta(\beta - 1)}{2} + (\alpha + \beta - 1)(n - \alpha - \beta + 1) + (\alpha + \beta - 2) + (\alpha + \beta - 3) + \cdots + \{\alpha + \beta - [(i - 1) - (n - \alpha + 1)]\} + j - i + \alpha$$

$$= \alpha\beta + \frac{\beta(\beta - 1)}{2} + (\alpha + \beta - 1)(n - \alpha - \beta + 1) + (\alpha + \beta)(i - n + \alpha - 1) - \{1 + 2 + 3 + \cdots + [(i - 1) - (n - \alpha + 1)]\} + 1$$

$$= \alpha\beta + \frac{\beta(\beta - 1)}{2} + (\alpha + \beta - 1)(n - \alpha - \beta + 1) + (\alpha + \beta)(i - n + \alpha - 1) - \frac{(i - n + \alpha - 1) \times (i - n + \alpha - 2)}{2} + 1$$

Assignment 2.11

(a) Find the number of elements in the $\alpha\beta$-band matrix.
(b) Show that the indexing formula for $\alpha\beta$-band matrix is a general formula of tridiagonal matrix.
(c) Obtain the indexing formula for column-major ordering of $\alpha\beta$-band matrix.

2.4.3 Three-dimensional and *n*-dimensional Arrays

So far we have discussed one-dimensional and two-dimensional arrays. In some advanced applications, more than two-dimensional arrays are also used. Here we will discuss multidimensional arrays from the theoretical point of view.

Three-dimensional (3-D) array

A three-dimensional array will look as shown in Figure 2.12.

A three-dimensional array can be compared with a book whereas two-dimensional and one-dimensional arrays can be compared with a page and a line respectively. Here, three major dimensions can be termed as row, column and page. Let, for a three-dimensional array, the following specifications are known:

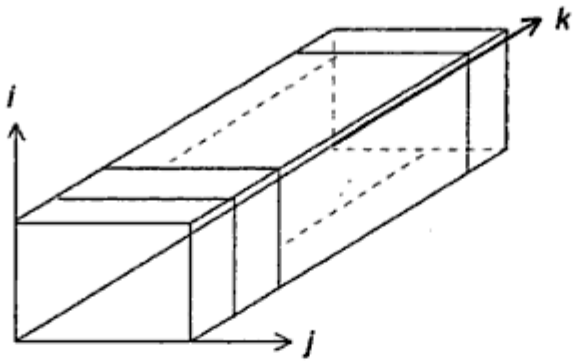


Fig. 2.12 A three-dimensional array.

Number of rows = x (number of elements in a column)
Number of columns = y (number of elements in a row) and
Number of pages = z

Assume that a_{ijk} is an element in i-th row, j-th column and k-th page. Now, we are interested to find the memory location of element a_{ijk} in the array.

Storing a 3-D array means, storing the pages one by one. Again storing a page is same as storing a 2-D array. Thus, if the elements in a page stored in row-major order then we term that 3-D array is also in row-major order. The following formula is taking into consideration of row-major order of a 3-D array:

$$\begin{aligned}\text{Address } (a_{ijk}) &= \text{Number of elements in first } (k-1) \text{ pages} \\ &\quad + \text{Number of elements in } k\text{-th page up to } (i-1) \text{ rows} \\ &\quad + \text{Number of elements in } k\text{-th page, in } i\text{-th row up to } j\text{-th column} \\ &= xy\,(k-1) + (i-1)y + j\end{aligned}$$

Instead of index starting from l for all indices (as assumed in the above mentioned formula), if we assume that i changes between l_x to u_x, j changes between l_y to u_y and k changes between l_z to u_z. So that

$$x = u_x - l_x + 1, \qquad y = u_y - l_y + 1 \qquad \text{and} \qquad z = u_z - l_z + 1$$

Also assuming the word size of each element is w instead of l then the above indexing formula for 3-D array can be restated in general form as:

$$\text{Address } (a_{ijk}) = M + [xy\,(k - l_z) + (i - l_x)y + (j - l_y)] \times w$$

where, M denotes the base address of the array.

n-dimensional array

From the representation of 2-D and 3-D, we can extend our idea to store an n-dimensional array. Here, to identify an element, we need n indices, $i_1, i_2, \ldots, i_n$.

Let x_j be the number of elements in j-th dimension and range of index for i_j, say, varies between $l_j \ldots u_j$, where $1 \le j \le n$. So, the total number of elements in the array is

$$\prod_{j=1}^{n} x_j = \prod_{j=1}^{n} (u_j - l_j + 1), \qquad 1 \le j \le n$$

Storing this n-dimensional array in memory, any element can be referenced using the formula:

$$\text{Address } (a_{i1\ i2\ \cdots\ in}) = (i_n - l_n)\, x_{n-1}\, x_{n-2}\, x_{n-3} \cdots x_3 x_2 x_1 + (i_{n-1} - l_{n-1}) x_{n-2} x_{n-3} \cdots x_3 x_2 x_1 + \cdots + (i_2 - l_2)\, x_1 + (i_1 - l_1)$$

in row-major order and

$$\text{Address } (a_{i1\ i2\ \cdots\ in}) = (i_1 - l_1)\, x_2 x_3 \cdots x_{n-2} x_{n-1} x_n + (i_2 - l_2)\, x_3 x_4 \cdots x_{n-2} x_{n-1} x_n + (i_{n-1} - l_{n-1})\, x_{n-2}\, x_{n-1}\, x_n + (i_n - l_n)$$

in column-major order.

2.5 POINTER ARRAYS

Some integer values, real numbers, string of characters, etc., are simple elements that arrays may contain. In addition to these, another kind of array is also well known in the theory of computer science which is in use to store address of some memory variables or the address of another arrays. Address of memory variable or array is known as *pointer* and an array containing pointers as its elements is known as *pointer array*. Let us consider the following example to illustrate a pointer array.

Example: Suppose, we want to store the marks of all students of Computer Science and Engineering (CSE) department for a year. There are six classes, say, CS–1, CS–2, ..., CS–6. And for each class, there are at most 5 courses; for example, course of i-th class can be denoted as CSij (where j varies from 1 to 5). We will assume that there is at most 30 students in each class.

We should maintain an array (one-dimensional) of size $6 \times 5 \times 30$ (= 900) to store the marks. We will use pointer arrays to keep track of the mark of i-th year student in a course $j(s_{ij})$ and the starting location where the marks of the course CSi begin. The idea is illustrated in Figure 2.13.

Here, we have maintained 3 arrays, viz. CLASSES, COURSES, and MARKS having sizes 6, 30 and 900 respectively. The MARKS array stores the marks obtained by the various students in different courses. The COURSES array stores the starting location for the marks of different courses. In Figure 2.13, starting location for all the marks of CS41 course is shown by the pointed arrow. The CLASSES array points the starting location (at COURSES) for various courses under a class. For example, the location of CS41 (the first course) under the 4-th year class (CS4) is shown by the pointed arrow.

Thus, if we want to retrieve the marks obtained by the k-th student in j-th course of class CSi, then we have to search first the array CLASSES to obtain the starting location of all courses under class CSi. After knowing the starting location of class CSi we shall move to that location in array COURSES. The information for j-th course can be obtained by shifting sequentially j steps in the array COURSES, where the starting location (at MARKS) for the j-th course will be obtained. With that address, one can move to the referred location at MARKS array and again sequentially moving k steps the desired marks will be retrieved.

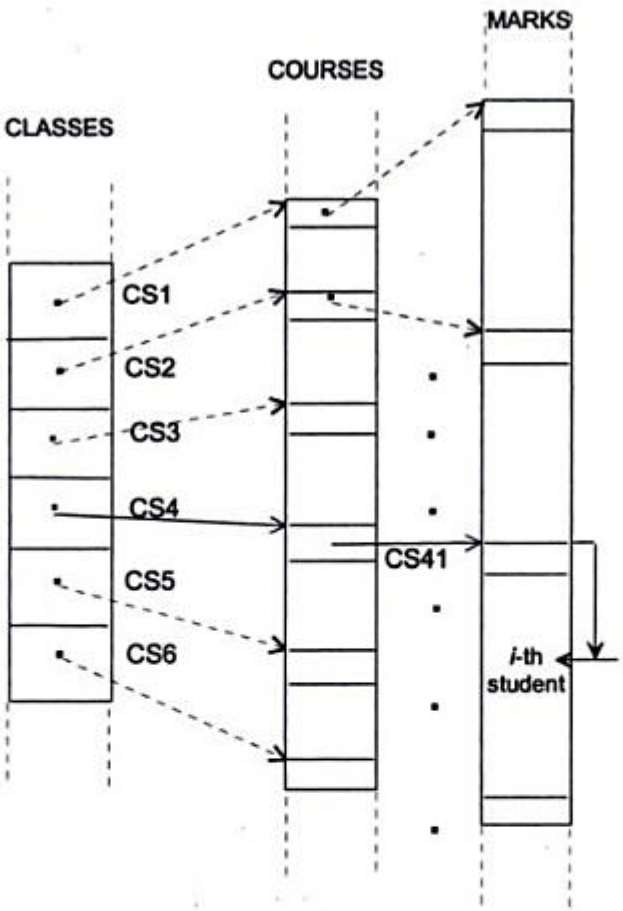


Fig. 2.13 Pointer array.

Assignment 2.12

(a) Using only a single array of marks having size 900, give an idea how the same information can be maintained. You must consider the case that there may be less than 5 courses in a class or may be less than 30 students in a course.

(b) Other than one-dimensional array, 2-D or 3-D can be employed? How?

PROBLEMS TO PONDER

2.1 Following are the three known searching methods when data are stored in an array.

(a) Sequential search

(b) Binary search

(c) Interpolation search

(d) Address calculation search (closed hashing)

(i) Write generic implementation for the above searching methods so that they can be used with any type of data stored in the array.

(ii) Obtain a time comparison of the above methods and then conclude which is/are to be taken as best searching methods.

2.2 Following are the important sorting algorithms which can be applied on data stored in an array.

(a) Bubble sort
(b) Selection sort
(c) Insertion sort
(d) Shell sort
(e) Quick sort

(i) Store a set of data into a *generic array*. Obtain a sorted order of data either in ascending or descending order using the above sorting methods.

(ii) For a given set of data, apply the above sorting methods and then note the time required for sorting in each case. Then conclude which is/are the best sorting method(s).

Note: An array is generic means it can hold any data irrespective of its type.

2.3 A multiplication table is a matrix of order $m \times n$ where an entry in i-th row and j-th column is the product $x \times y$, where x and y are the numbers in i-th row and j-th column respectively. Figure 2.14 shows a multiplication table from 3 to 7.

	3	4	5	6	7
3	9	12	15	18	21
4	12	16	20	24	28
5	15	20	25	30	35
6	18	24	30	36	42
7	21	28	35	42	49

Fig. 2.14 A multiplication table from 3 to 7.

Write a program to display multiplication table from x to y.

2.4 A magic square is a square matrix of integers such that the sum of every row, the sum of every column and sum of each of the diagonal are equal. Such a magic square is shown in Figure 2.15.

4	15	14	1
9	6	7	12
5	10	11	8
16	3	2	13

Fig. 2.15 A magic square with SUM = 34.

(a) Write a program to read a set of integers for a square matrix and then decide whether the matrix represents a magic square or not.

(b) Write a game program as follows:

(i) Read the size of the square matrix, $N \times N$

(ii) Display a square matrix (now it is blank) of $N \times N$

(iii) Allow the player to insert data into the matrix as displayed (you should give a chance to the user to confirm the entry and to alter the previous entries, if desire).

(iv) After the completion of all the entries from the player, count the score as follows:

Score = 0 (zero) if it is not a magic square. Otherwise score = $T + P*100$, where T is the time required in second and P is the number of alteration of entries.

The player's performance will be judged by the minimum score achieved, other than zero.

REFERENCES

1. Ellis Horowitz and Sartaj Sahni, *Fundamentals of Data Structures*, Computer Science Press, Rockville, Maryland, 1985.
2. Gotlieb, C.C. and Gotlieb, L.R., *Data Types and Structures*, Prentice Hall, Englewood Cliffs, New Jersey, 1986.
3. Jean Paul Tremblay and Paul G. Sorenson, *An Introduction to Data Structures with Applications*, McGraw-Hill, New York, 1987.
4. Robert L. Kruse, Bruce P. Leung and Clovis L. Tondo, *Data Structures and Program Design in C*, Prentice Hall, Englewood Cliffs, New Jersey, 1995.

3 Linked Lists

Array is a data structure where elements are stored in consecutive memory locations. In order to occupy the adjacent space, block of memory that is required for the array should be allocated before hand. Once memory is located it cannot be extended any more. This is why array is known as *static data structure*. In contrast to this, linked list is called *dynamic data structure* where amount of memory required can be varied during its use. In linked list, adjacency between the elements are maintained by means of *links* or *pointers*. A link or pointer actually is the address (memory location) of the subsequent element. Thus, in a linked list, data (actual content) and link (to point to the next data) both are required to be maintained. An element in a linked list is specially termed as *node*, which can be viewed as shown in Figure 3.1. A node consists of two fields: DATA (to store the actual information) and LINK (to point to the next node).

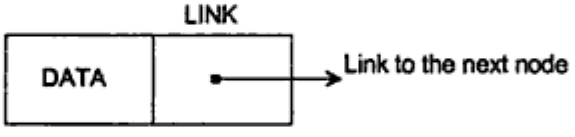


Fig. 3.1 Node: an element in a linked list.

3.1 DEFINITION

A *linked list* is an ordered collection of finite, homogeneous data elements called *nodes* where the linear order is maintained by means of links or pointers.

Depending on the requirements the pointers are maintained, and accordingly linked list can be classified into three major groups: single linked list, circular linked list, and double linked list.

3.2 SINGLE LINKED LIST

In a single linked list each node contains only one link which points the subsequent node in the list. Figure 3.2 shows a linked list with 6 nodes.

Here, N1, N2, . . ., N6 are the constituent nodes in the list. HEADER is an empty node (having data content NULL) and only used to store a pointer to the first node N1. Thus, if one knows the address of the HEADER node from the link field of this node, next node can be traced and so on. This means that starting from the first node one can reach to the last node whose link field does not contain any address rather a null value. Note that in a single linked list one can move from left to right only; this is why a single linked list is also alternatively termed as *one way* list.

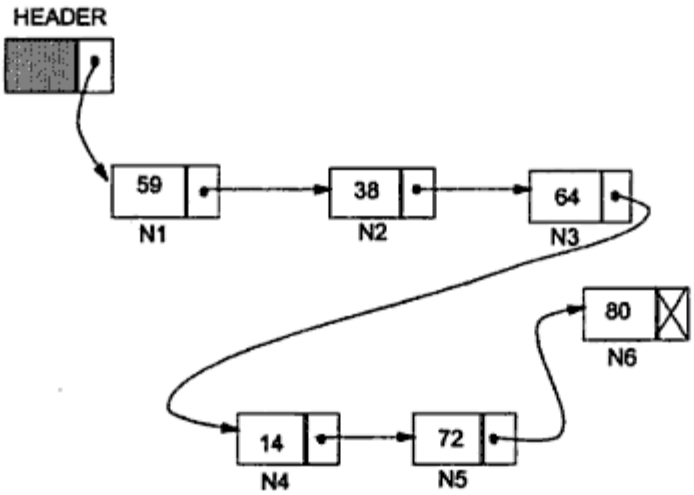


Fig. 3.2 A single linked list with 6 nodes.

3.2.1 Representation of a Linked List in Memory

There are two ways to represent a linked list in memory:

1. Static representation using array
2. Dynamic representation using free pool of storage

Static representation

Static representation of a single linked list maintains two arrays: one array for data and other for links. Static representation for the linked list as shown in the Figure 3.2 is given in Figure 3.3.

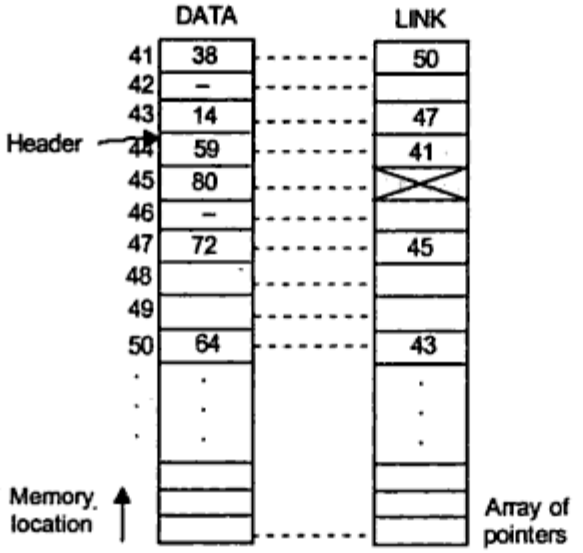


Fig. 3.3 Static representation of a single linked list using arrays.

Two parallel arrays of equal size are allocated which should be sufficient to store the entire linked list. Nevertheless this contradicts the idea of linked list (that is non-contagious location of elements). But in some programming languages, for example, ALGOL, FORTRAN, BASIC, etc., such a representation is the only representation to manage a linked list.

Dynamic representation

The efficient way of representing a linked list is using free pool of storage. In this method, there is a *memory bank* (which is nothing but a collection of free memory spaces), and a *memory manager* (a program, in fact). During the creation of linked list, whenever a node is required the request is placed to the memory manager; memory manager will then search the memory bank for the block requested and if found grants a desired block to the caller. Again, there is also another program called *garbage collector*, it plays whenever a node is no more in use; it returns the unused node to the memory bank. It may be noted that memory bank is also a list of memory space that is available to a programmer. Such a memory management is known as *dynamic* memory management. Dynamic representation of linked list uses the dynamic memory management policy.

The mechanism of dynamic representation of single linked list is illustrated as shown in Figures 3.4(a) and 3.4(b). A list of available memory space is there whose pointer is stored in AVAIL. For a request of a node, the list AVAIL is searched for the block of right size. If AVAIL is null or the block of desired size is not found memory manager will return a message accordingly. Suppose, the block is found and let it be *XY*. Then memory manager will return the pointer of *XY* to the caller in a temporary buffer, say NEW. The newly availed node *XY* then can be inserted at any position in the linked list by changing the pointers of the concerned nodes. In Figure 3.4(a), the node *XY* is inserted at the end and change of pointers are shown by the dotted arrows. Figure 3.4(b) explains the mechanism of how a node is to be returned to the memory bank.

The pointers that are required to be manipulated while returning a node is shown by dotted arrows. Note that such allocation or deallocation is carried out by changing the pointers only.

3.2.2 Operations on a Single Linked List

Possible operations on a single linked list are listed as below:

- *Traversing* a list
- *Insertion* of a node into a list
- *Deletion* of a node from a list
- *Copy* a linked list to make a duplicate
- *Merging* two linked lists into a larger list
- *Searching* for an element in a list.

We will assume the following convention in our present discussion: say *X* is a node. The values in the DATA field and LINK field will be denoted as X.DATA and X.LINK respectively. We will write NULL to imply that value in the field like DATA, LINK is nil.

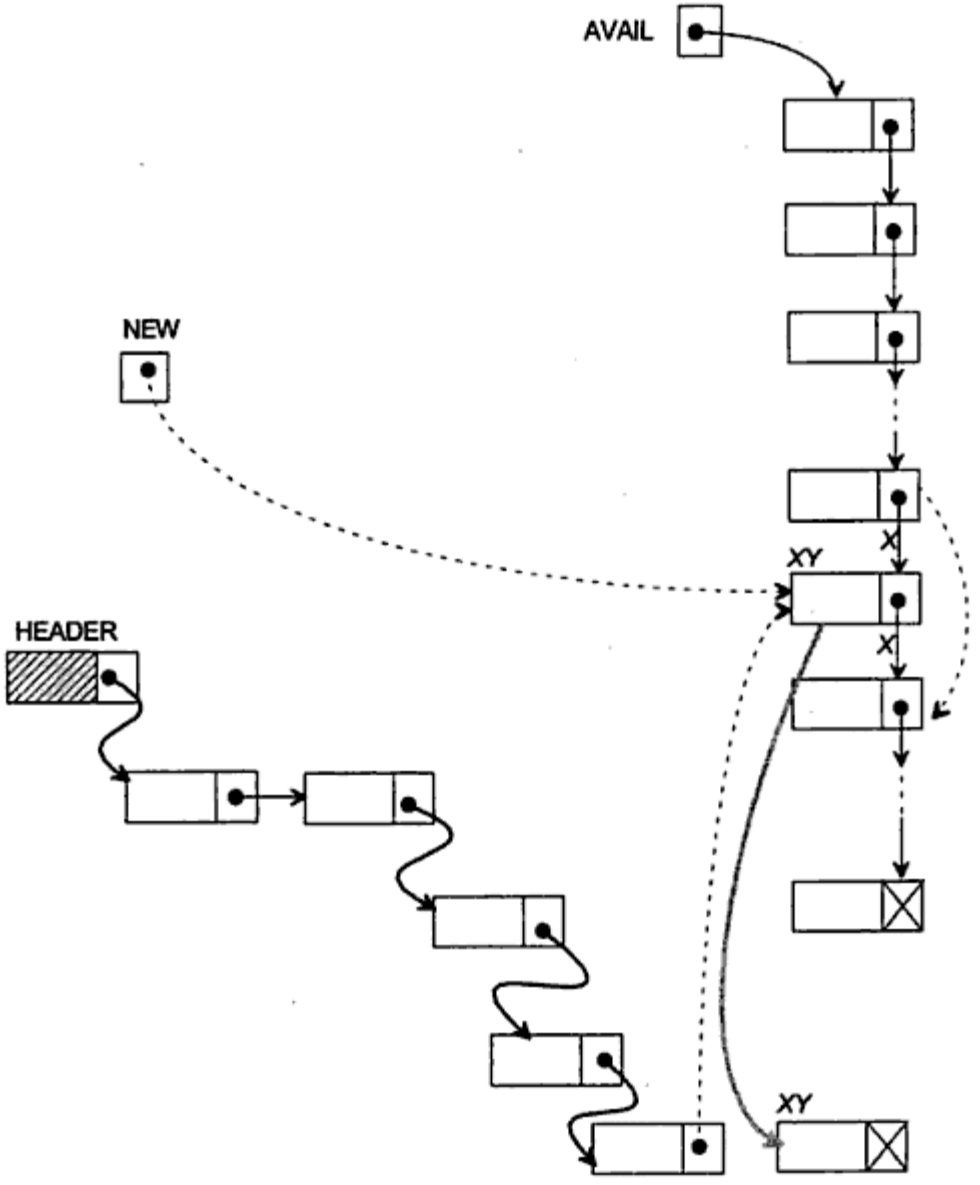


Fig. 3.4(a) Allocation of a node from memory bank to a linked list.

Traversing a single linked list

To traverse a single linked list we mean to visit every node in the list starting from the first node to the last node. Following is the algorithm TRAVERSE_SL for the same.

Algorithm TRAVERSE_SL (HEADER)
Input: HEADER is the pointer to the header node.
Output: According to the PROCESS( )
Data structures: A single linked list whose address of the starting node is known from HEADER.

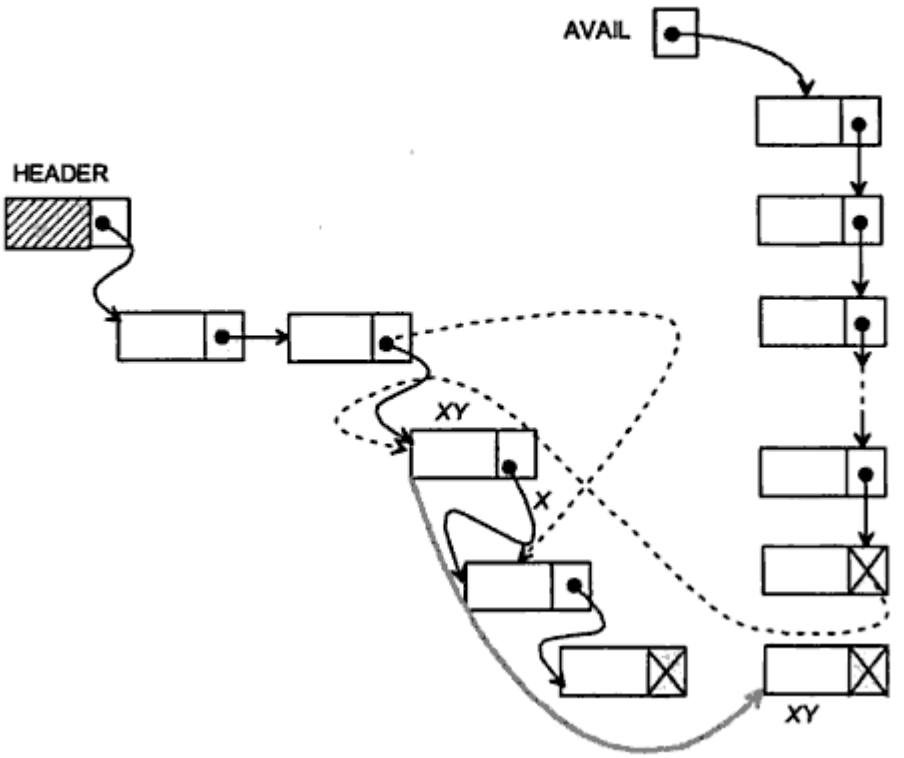


Fig. 3.4(b) Returning a node from list to memory bank.

Steps:

1. ptr = HEADER.LINK //ptr is to store the pointer to a current node
2. While (ptr ≠ NULL) do //Continue till the last node
   1. PROCESS(ptr) //Perform PROCESS( ) on the current node
   2. ptr = ptr.LINK //Move to the next node
3. EndWhile
4. Stop

Note: A simple operation of PROCESS( ) may be thought as to print the data content of a node.

Insertion of a node into a single linked list

There are various positions where a node can be inserted:

(i) Insert at front (as a first element)

(ii) Insert at end (as a last element)

(iii) Insert at any position.

Before we discuss these insertions, let us assume a procedure GETNODE(NODE) to get a pointer of a memory block which suits the type NODE. The procedure may be defined as below:

Procedure GETNODE(NODE)

1. If (AVAIL = NULL) //AVAIL is the pointer to the pool of free storage
   1. Return (NULL)
   2. Print "Insufficient memory: Unable to allocate memory"
2. Else //Sufficient memory is available
   1. ptr = AVAIL
   2. While (SIZEOF(ptr) ≠ SIZEOF(NODE)) and (ptr ≠ NULL)
      //Till the desired block is not found or search reaches
      //at the end of the pool
      1. ptr1 = ptr //To keep the track of the previous block
      2. ptr = ptr.LINK //Move to the next block
   3. EndWhile
   4. If (SIZEOF(ptr) = SIZEOF(NODE)) //Memory block of right size is found
      1. ptr1.LINK = ptr.LINK //Update the AVAIL List
      2. Return(ptr)
   5. Else
      1. Print "The memory block is too large to fit"
      2. Return(NULL)
   6. EndIf
3. EndIf
4. Stop

Insertion of a node into a single linked list at the front. The algorithm INSERT_SL_FRONT is to insert a node into a single linked list at the front of the list.

Algorithm INSERT_SL_FRONT(HEADER, *X*)

Input: HEADER is the pointer to the header node and *X* is the data of the node to be inserted.

Output: A single linked list with newly inserted node in the front of the list.

Data structures: A single linked list whose address of the starting node is known from HEADER.

Steps:

1. new = GETNODE(NODE) //Get a memory block of type NODE and store
   //its pointer in new
2. If (new = NULL) then //Memory manager returns NULL on searching
   //the memory bank
   1. Print "Memory underflow: No insertion"
   2. Exit //Quit the program
3. Else //Memory is available and get a node from
   //memory bank
   1. new.LINK = HEADER.LINK //Change of pointer 1 as shown in Figure 3.5(a)
   2. new.DATA = *X* //Copy the data *X* to newly availed node
   3. HEADER.LINK = new //Change of pointer 2 as shown in Figure 3.5(a)
4. EndIf
5. Stop

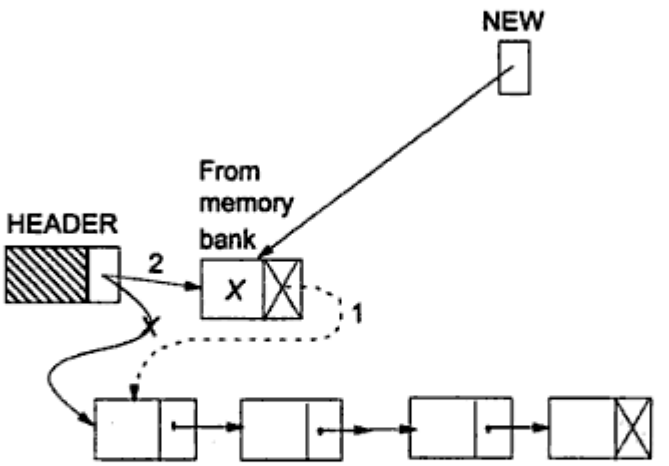


Fig. 3.5(a) Insertion of a node at the front of a single linked list.

Insertion of a node into a single linked list at the end. The algorithm INSERT_SL_END is to insert a node into a single linked list at the end of the list.

Algorithm INSERT_SL_END (HEADER, *X*)

Input: HEADER is the pointer to the header node and *X* is the data of the node to be inserted.

Output: A single linked list with newly inserted node having data *X* at end.

Data structures: A single linked list whose address of the starting node is known from HEADER.

Steps:

1. new = GETNODE(NODE) //Get a memory block of type NODE and store //its pointer in new
2. If (new = NULL) then //Unable to allocate memory for a node
   1. Print "Memory is insufficient: Insertion is not possible"
   2. Exit //Quit the program
3. Else //Move to the end of the given list and then insert
   1. ptr = HEADER //Start from the HEADER node
   2. While (ptr.LINK ≠ NULL) do //Move to the end
      1. ptr = ptr.LINK //Change pointer to the next node
   3. EndWhile
   4. ptr.LINK = new //Change the link field of last node: Pointer 1 as //in Figure 3.5(b)
   5. new.DATA = *X* //Copy the content *X* into new node
4. EndIf
5. Stop

Insertion of a node into a single linked list at any position in the list. The algorithm INSERT_SL_ANY is to insert a node into a single linked list at any position in the list.

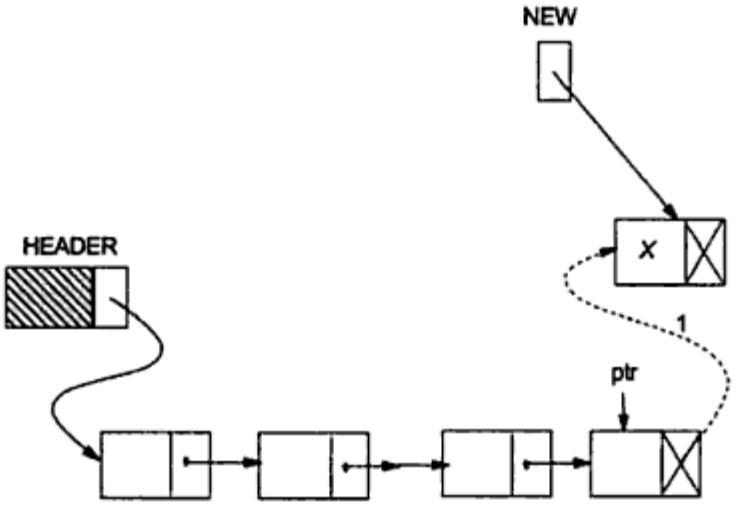


Fig. 3.5(b) Insertion at the end of a single linked list.

Algorithm INSERT_SL_ANY(HEADER, *X*, KEY)

Input: HEADER is the pointer to the header node, *X* is the data of the node to be inserted, and KEY being the data of the key node after which the node has to be inserted.

Output: A single linked list enriched with newly inserted node having data *X* after the node with data KEY.

Data structures: A single linked list whose address of the starting node is known from HEADER.

Steps:

1. new = GETNODE(NODE) //Get a memory block of type NODE and store //its pointer in new
2. If (new = NULL) then //Unable to allocate memory for a node
   1. Print "Memory is insufficient: Insertion is not possible"
   2. Exit //Quit the program
3. Else
   1. ptr = HEADER //Start from the HEADER node
   2. While (ptr.DATA ≠ KEY) and (ptr.LINK ≠ NULL) do //Move to the node having data as KEY or at //the end if KEY is not in the list
      1. ptr = ptr.LINK
   3. EndWhile
   4. If (ptr.LINK = NULL) then //Search fails to find the KEY
      1. Print "KEY is not available in the list"
      2. Exit
   5. Else
      1. new.LINK = ptr.LINK //Change the pointer 1 as shown in Figure 3.5(c)
      2. new.DATA = *X* //Copy the content into the new node
      3. ptr.LINK = new //Change the pointer 2 as shown in Figure 3.5(c)
   6. EndIf
4. EndIf
5. Stop

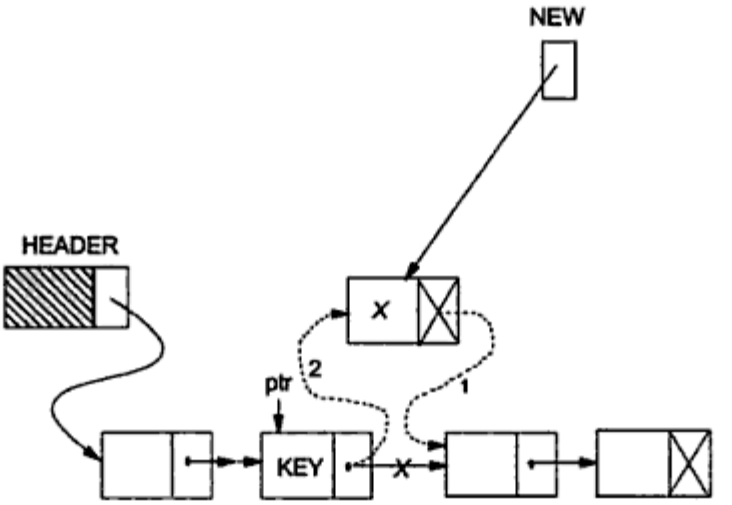


Fig. 3.5(c) Insert at any position in a single linked list.

Deletion of a node from a single linked list

Like insertions, there are also various cases of deletion:

(i) Deletion at the front of the list
(ii) Deletion at the end of the list
(iii) Deletion at any position in the list.

Let us consider the procedure for various cases of deletions. We will assume a procedure namely, RETURNNODE(PTR) which returns a node having pointer PTR to the free pool of storage. The procedure RETURNNODE(PTR) may be defined as follows:

```
Procedure RETURNNODE(PTR)          //PTR is the pointer of a node to be returned
1. ptr1 = AVAIL                    //Start from the beginning of the free pool
2. While (ptr1.LINK ≠ NULL) do
   1. ptr1 = ptr1.LINK
3. EndWhile
4. ptr1.LINK = PTR                 //Insert the node at the end
5. PTR.LINK = NULL                 //Node inserted is the last node
6. Stop
```

Note: The procedure RETURNNODE( ) inserts the free node at the end of the pool of free storage whose header address is AVAIL. Alternatively, we can insert the free node at the front of the AVAIL list which is left as an exercise.

Deletion of a node from a single linked list at the front. The algorithm DELETE_SL_FRONT is to delete a node from a single linked list at the front of the list.

Algorithm DELETE_SL_FRONT(HEADER)
Input: HEADER is the pointer to the header node.
Output: A single linked list eliminating the node at the front.
Data structures: A single linked list whose address of the starting node is known from HEADER.

Steps:

1. ptr = HEADER.LINK //Pointer to the first node
2. If (ptr = NULL) //If the list is empty
   1. Print "The list is empty: No deletion"
   2. Exit //Quit the program
3. Else //The list is not empty
   1. ptr1 = ptr.LINK //ptr1 is the pointer to the second node, if any
   2. HEADER.LINK = ptr1 //Next node becomes the first node as in Figure 3.6(a)
   3. RETURNNODE(ptr) //Deleted node is freed to the memory bank for //future use
4. EndIf
5. Stop

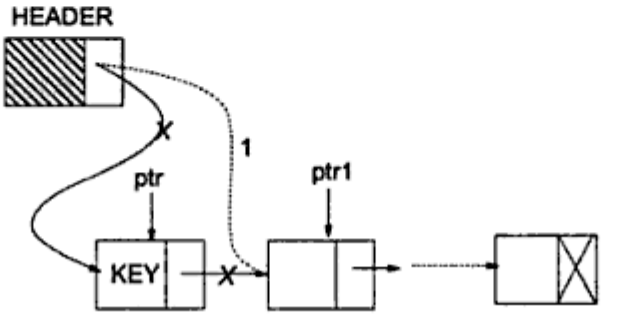


Fig. 3.6(a) Deletion of a node from a single linked list at the front.

Deletion of a node from a single linked list at the end. The algorithm DELETE_SL_END is to delete a node at the end from a single linked list.

Algorithm DELETE_SL_END(HEADER)

Input: HEADER is the pointer to the header node.

Output: A single linked list eliminating the node at the end.

Data structures: A single linked list whose address of the starting node is known from HEADER.

Steps:

1. ptr = HEADER //Move from the header node
2. If (ptr.LINK = NULL) then
   1. Print "The list is empty: No deletion possible"
   2. Exit //Quit the program
3. Else
   1. While (ptr.LINK ≠ NULL) do //Go to the last node
      1. ptr1 = ptr //To store the previous pointer
      2. ptr = ptr.LINK //Move to the next
   2. EndWhile
   3. ptr1.LINK = NULL //Last but one node become the last node as in //Figure 3.6(b)
   4. RETURNNODE(ptr) //Deleted node is returned to the memory bank //for future use
4. EndIf
5. Stop

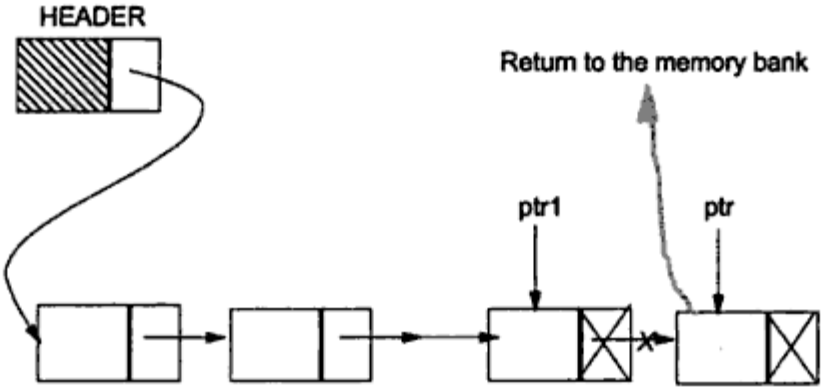


Fig. 3.6(b) Deletion of a node from a single linked list at the end.

Deletion of a node from a single linked list at any position in the list. The next algorithm DELETE_SL_ANY is to delete a node from any position in the single linked list.

Algorithm DELETE_SL_ANY

Input: HEADER is the pointer to the header node, KEY is the data content of the node to be deleted.

Output: A single linked list except the node with data content as KEY.

Data structures: A single linked list whose address of the starting node is known from HEADER.

Steps:

```
1. ptr1 = HEADER                    //Start from the header node
2. ptr = ptr1.LINK                  //This points to the first node, if any
3. While (ptr ≠ NULL) do
   1. If (ptr.DATA ≠ KEY) then      //If not found the key
      1. ptr1 = ptr                 //Keep a track of the pointer of the previous node
      2. ptr = ptr.LINK             //Shift to the next
   2. Else                          //The node is found
      1. ptr1.LINK = ptr.LINK       //Link field of the predecessor is to point
                                    //the successor of node under deletion, see Figure 3.6(c)
      2. RETURNNODE(ptr)            //Return the deleted node to the memory bank
      3. Exit                       //Exit the program
   3. EndIf
4. EndWhile
5. If (ptr = NULL) then             //When the desired node is not available in the list
   1. Print "Node with KEY does not exist: No deletion"
6. EndIf
7. Stop
```

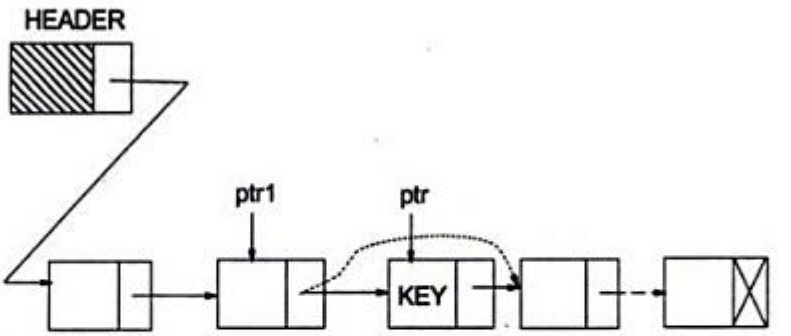


Fig. 3.6(c) Deletion of a node at any position in a single linked list.

Assignment 3.1

(a) It can be noted that in all our previous algorithms on single linked lists, we have maintained a header node whose data field contains null value and link field points the first node in the list. This simply means that even in an empty list there is a node. However, maintenance of this special node is just a convention. So, instead of making the data field to null value, if we store actual data, that is, header node itself becomes the first node then what modification you would suggest for all the algorithms as mentioned for insertion and deletion operations. [*Hint:* Only little change is required.]

(b) An alternative implementation for the algorithm DELETE_SL_ANY is proposed as given below:

Algorithm DELETE_SL_ANY_ALTERNATIVE(HEADER, KEY)

Input: HEADER is the pointer to the header node, KEY is the data content of the node to be deleted.

Output: A single linked list except the node with data content as KEY.

Data structures: A single linked list whose address of the starting node is known from HEADER.

Steps:

1. ptr = HEADER.LINK
2. While(ptr.DATA ≠ KEY) and (ptr ≠ NULL) do
   1. ptr1 = ptr
   2. ptr = ptr.LINK
3. EndWhile
4. If (ptr.DATA = KEY) then
   1. ptr1.LINK = ptr.LINK
   2. RETURNNODE(ptr)
5. Else
   1. Print "Node does not exist : Deletion is unsuccessful"
6. EndIf
7. Stop

With respect to the above algorithm, answer the following:

(i) Is the algorithm works for an empty list (the list containing only header node)?

(ii) If the key element is at the first position or last position then will it work correctly?

(iii) Find the bugs, if any, in it and then rectify for correct operation.

Following is the algorithm MERGE_SL to merge two single linked lists into one single linked list:

Algorithm MERGE_SL(HEADER1, HEADER2; HEADER)
Input: HEADER1 and HEADER2 are pointers to header nodes of lists (L1 and L2, respectively) to be merged.
Output: HEADER is the pointer to the resultant list.
Data structures: Single linked list structure.

Steps:

1. ptr = HEADER1
2. While (ptr.LINK ≠ NULL) do //Move to the last node in the list L1
   1. ptr = ptr.LINK
3. EndWhile
4. ptr.LINK = HEADER2.LINK //Last node in L1 points to the first node in L2
5. RETURNNODE(HEADER2) //Return the header node to the memory bank
6. HEADER = HEADER1 //HEADER becomes the header node of the merged list
7. Stop

Assignment 3.2 Perform the following tasks on linked lists using single linked list representation:

(a) Obtain insertion and deletion operations on a list which is stored in an array.
(b) Find the maximum, minimum and average for some numeric data as element stored in single linked list.
(c) For a given set of data, insert them in a sorted fashion.
(d) A list containing unsorted data is known, make it as a sorted list.
(e) Suppose a list is no more required, dispose the entire list into the pool of free storage.
(f) Suppose two lists are known; compare two lists to furnish the following:
   (i) Exact replica or not
   (ii) Number of matching occurs between the lists
   (iii) Two lists are sorted in the same order or opposite order etc.
(g) For a given list obtain decatenation (sublist). Decide yourself a suitable criteria for splitting.
(h) Store the element in reverse order that of a given list (don't make another list).

Searching for an element in a single linked list

The algorithm SEARCH_SL( ) is given below to search an item in a single linked list.

Algorithm SEARCH_SL(Key; LOCATION)
Input: KEY, the item to be searched.
Output: LOCATION, the pointer to a node where the KEY belongs to or an error messages.
Data structures: A single linked list whose address of the starting node is known from HEADER.

Step:

1. ptr = HEADER.LINK //Start from the first node
2. flag = 0, LOCATION = NULL

3. While (ptr ≠ NULL) and (flag = 0) do
   1. If (ptr.DATA = KEY) then
      1. flag = 1 //Search is finished
      2. LOCATION = ptr
      3. Return (LOCATION)
   2. Else
      1. ptr = ptr.LINK //Move to the next node
   3. EndIf
4. EndWhile
5. If (ptr = NULL) then
   1. Print "Search is unsuccessful"
6. EndIf
7. Stop

3.3 CIRCULAR LINKED LIST

In our previous discussion, we have noticed that in single linked list, the link field of the last node is null (hereafter a single linked list may be read as ordinary linked list), but a number of advantages can be gained if we utilize this link field to store the pointer of the header node. A linked list where the last node points the header node is called *circular* linked list. Figure 3.8 shows a pictorial representation of a circular linked list.

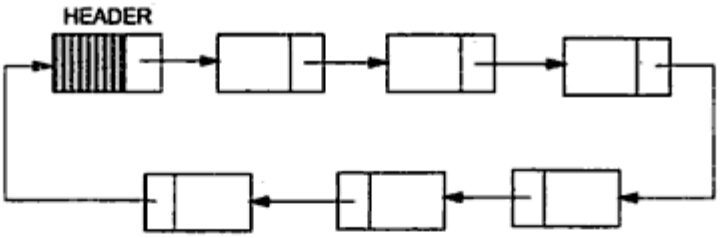


Fig. 3.8 A circular linked list.

Circular linked lists have certain advantages over ordinary linked lists. Few advantages of circular linked lists are discussed as below:

Accessibility of a member node in the list

In an ordinary list, a member node is accessible from a particular node, that is, from the header node only. But in a circular linked list, every member node is accessible from any node by merely chaining through the list.

Example: Suppose, we are manipulating some information which are stored in a list. Also, think of a case, where for a given data we want to find the earlier occurrence(s) as well as post occurrence(s). Post occurrence(s) can be traced out by chaining through the list from the current node irrespective of whether the list is maintained as circular linked or ordinary linked list.

In order to find all the earlier occurrences, in case of ordinary linked lists, we have to start our traversing from the header node at the cost of maintaining pointer for header in addition to the pointers for current node and another for chaining. But in case of circular linked list, one can trace out the same without maintaining the header information; rather maintaining only two pointers. Note that, in ordinary linked lists, one can chain through left to right only where it is virtually in the both direction for circular linked lists.

Null link problem

Null value in the link field may create some problem during the execution of programs if a proper care is not taken. This is illustrated by mentioning two algorithms to perform search on ordinary linked list and circular linked list.

Algorithm SEARCH_SL(KEY)
Input: KEY the item to be searched.
Output: Location, the pointer to a node where KEY belongs or an error message.
Data structures: A single linked list whose address of the starting node is known from HEADER.

Steps:

1. ptr = HEADER.LINK
2. While (ptr ≠ NULL) do
   1. If (ptr.DATA ≠ KEY) then
      1. ptr = ptr.LINK
   2. Else
      1. Return(ptr)
      2. Exit
      3. EndIf
3. EndWhile
4. If (ptr = NULL) then
   1. Print "The entire list has traversed but KEY is not found"
5. EndIf
6. Stop

Note that, here two tests in step 2 (which control the loop of searching) cannot be placed together as `While (ptr≠NULL) AND (ptr.DATA≠KEY) do` because in that case there will be an execution error for ptr.DATA since it is not defined when ptr = NULL. But with circular linked list very simple implementation is possible without any special care for NULL pointer. As an illustration the searching of an element in a circular linked list is given below:

Algorithm SEARCH_CL(KEY)
Input: KEY the item of search.
Output: Location, the pointer to a node where KEY belongs or an error message.
Data structures: A circular linked list whose address to the starting node is known from HEADER.

Steps:

1. ptr = HEADER.LINK
2. While (ptr.DATA ≠ KEY) and (ptr ≠ HEADER) do
   1. ptr = ptr.LINK
3. EndWhile
4. If (ptr.DATA = KEY)
   1. Return (ptr)
5. Else
   1. Print "Entire list is searched: KEY node is not found"
6. EndIf
7. Stop

Some easy-to-implement operations

Some operation can easily be implemented with circular linked list than ordinary linked list. Operations like merging (concatenation) splitting (decatenation), deletion, dispose of an entire list, etc., can easily be performed on circular linked list. The merging operation, as in Figure 3.9, is explained in algorithm MERGE_CL as under:

Algorithm MERGE_CL(HEADER1, HEADER2)
Input: HEADER1 and HEADER2 are the two pointers to header nodes.
Output: A larger circular linked list containing all the nodes from lists HEADER1 and HEADER2.
Data structures: Circular linked list structure.

Steps:

1. ptr1 = HEADER1.LINK
2. ptr2 = HEADER2.LINK
3. HEADER1.LINK = ptr2
4. HEADER2.LINK = ptr1
5. Stop

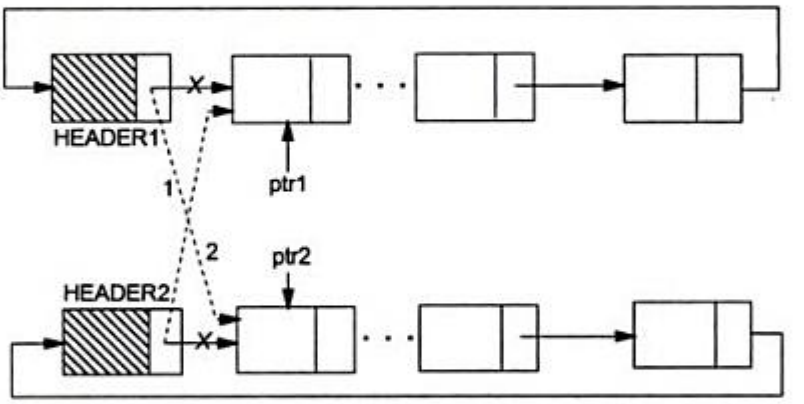


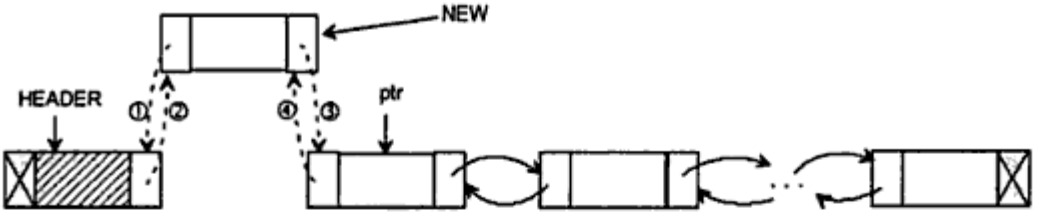
Fig. 3.9 Concatenation of two circular linked lists.

One can easily compare the algorithm MERGE_CL with MERGE_SL where an entire list is required to be traversed in order to locate the last node, and at the cost of some time.

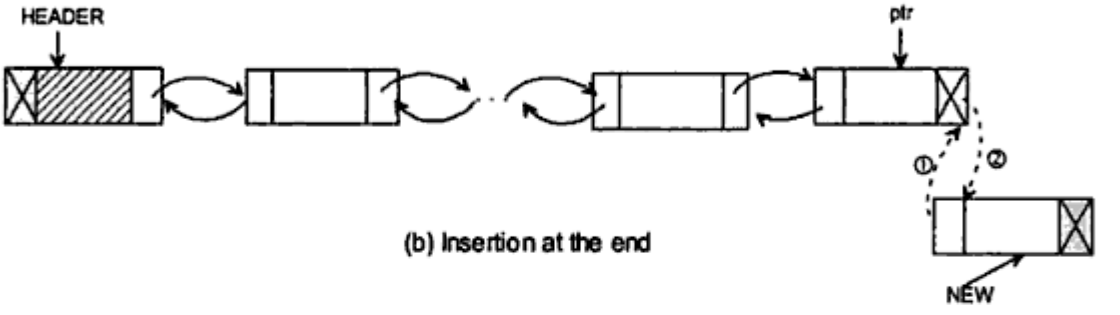
It can be noted that the algorithm MERGE_CL keeps two header node thus wasting some memory space; in fact the maintenance of header node in circular list has no significance (truly speaking it is a burden here) unlike ordinary linked lists.

Assignment 3.3

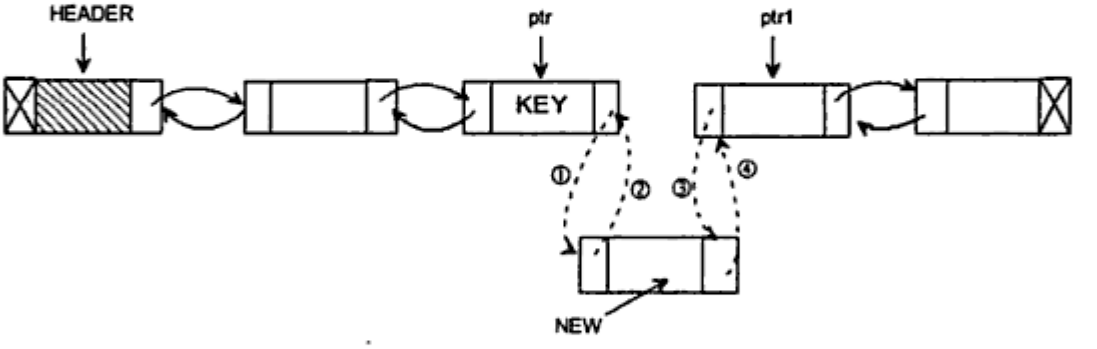
(a) Modify algorithm MERGE_CL so that it will contain only one header node instead of two header nodes.
(b) By writing algorithms for disposing the entire list with (i) ordinary linked list and (ii) circular linked list, show that the algorithm for circular linked list is better than ordinary linked list.
(c) Write a recursive algorithm to invert (pointer will point in reverse direction) a circular linked list.
(d) Write algorithms to convert an ordinary single linked list into a circular linked list and vice versa.



(a) Insertion at the front



(b) Insertion at the end



(c) Insertion at any intermediate position

Fig. 3.11 Insertion of a node at various positions in a double linked list.

Algorithm INSERT_DL_FRONT(*X*)
Input: *X* the data content of the node to be inserted.
Output: A double linked list enriched with a node containing data as *X* at the front.
Data structure: Double linked list structure whose pointer to the header node is HEADER.

Steps:

1. ptr = HEADER.RLINK //Points to the first node
2. new = GETNODE(NODE) //Avail a new node from the memory bank
3. If (new ≠ NULL) then //If new node is available
   1. new.LLINK = HEADER //Newly inserted node points the header as 1 in Figure //3.11(a)
   2. HEADER.RLINK = new //Header now points to the new node as 2 in Figure //3.11(a)

3. new.RLINK = ptr //See the change in pointer shown as 3 in Figure 3.11(a)
4. ptr.LLINK = new //See the change in pointer shown as 4 in Figure 3.11(a)
5. new.DATA = *X* //Copy the data into newly inserted node

4. Else
   1. Print "Unable to allocate memory: Insertion is not possible"
5. EndIf
6. Stop

Insertion of a node at the end. The algorithm INSERT_DL_END is to insert a node at the end into a double linked list.

Algorithm INSERT_DL_END(*X*)
Input: *X* the data content of the node to be inserted.
Output: A double linked list enriched with a node containing data as *X* at the end of the list.
Data structure: Double linked list structure whose pointer to the header node is HEADER.

Steps:

1. ptr = HEADER
2. While (ptr.RLINK ≠ NULL) do //Move to the last node
   1. ptr = ptr.RLINK
3. EndWhile
4. new = GETNODE(NODE) //Avail a new node
5. If (new ≠ NULL) then //If the node is available
   1. new.LLINK = ptr //Change the pointer shown as 1 in Figure 3.11(b)
   2. ptr.RLINK = new //Change the pointer shown as 2 in Figure 3.11(b)
   3. new.RLINK = NULL //Make the new node as the last node
   4. new.DATA = *X* //Copy the data into the new node
6. Else
   1. print "Unable to allocate memory: Insertion is not possible"
7. EndIf
8. Stop

Insertion of a node at any position in the list. The algorithm INSERT_DL_ANY is to insert a node into a double linked list at any position.

Algorithm INSERT_DL_ANY(*X*, KEY)
Input: *X* be the data content of the node to be inserted, and KEY the data content of the node after which the new node to be inserted.
Output: A double linked list enriched with a node containing data as *X* after the node with data KEY, if any.
Data structure: Double linked list structure whose pointer to the header node is HEADER.

Steps:

1. ptr = HEADER
2. While (ptr.DATA ≠ KEY) and (ptr.RLINK ≠ NULL) do // Move to the key node if the current //node is not the KEY node or list reaches at the end
   1. ptr = ptr.RLINK

3. EndWhile
4. new = GETNODE(NODE) //Get a new node from the pool of free storage
5. If (new = NULL) then //When the memory is not available
   1. Print "Memory is not available"
   2. Exit //Quit the program
6. EndIf
7. If (ptr.RLINK = NULL) //If the KEY node is at the end or not found in the list
   1. new.LLINK = ptr
   2. ptr.RLINK = new //Insert at the end
   3. new.RLINK = NULL
   4. new.DATA = *X* //Copy the information to the newly inserted node
8. Else //The KEY is available
   1. ptr1 = ptr.RLINK //Next node after the key node
   2. new.LLINK = ptr //Change the pointer shown as 2 in Figure 3.11(c)
   3. new.RLINK = ptr1 //Change the pointer shown as 4 in Figure 3.11(c)
   4. ptr.RLINK = new //Change the pointer shown as 1 in Figure 3.11(c)
   5. ptr1.LLINK = new //Change the pointer shown as 3 in Figure 3.11(c)
   6. ptr = new //This becomes the current node
   7. new.DATA = *X* //Copy the content to the newly inserted node
9. EndIf
10. Stop

Note: Observe that the algorithm INSERT_DL_ANY will insert a node even the key node does not exist. In that case the node will be inserted at the end of the list. Also, note that the algorithm will work even if the list is empty.

Deletion in a double linked list

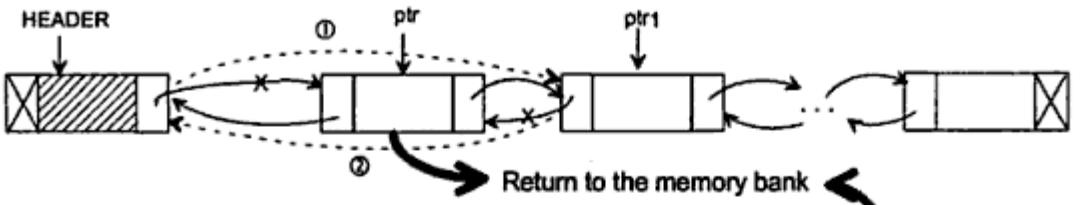
Deletion of a node from a double linked list may take place from any position in the list, which are depicted as shown in Figure 3.12. Let us consider each of the cases separately.

Deletion of a node at the front of a double linked list. The algorithm is as under:

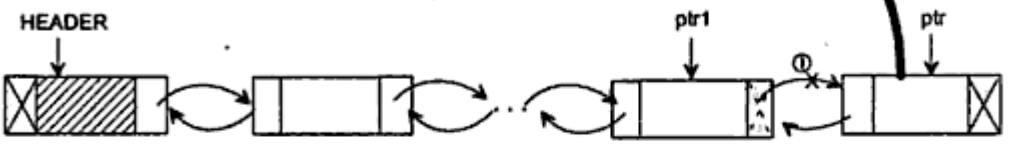
Algorithm DELETE_DL_FRONT( )
Input: A double linked list with data.
Output: A reduced double linked list.
Data structure: Double linked list structure whose pointer to the header node is HEADER.

Steps:

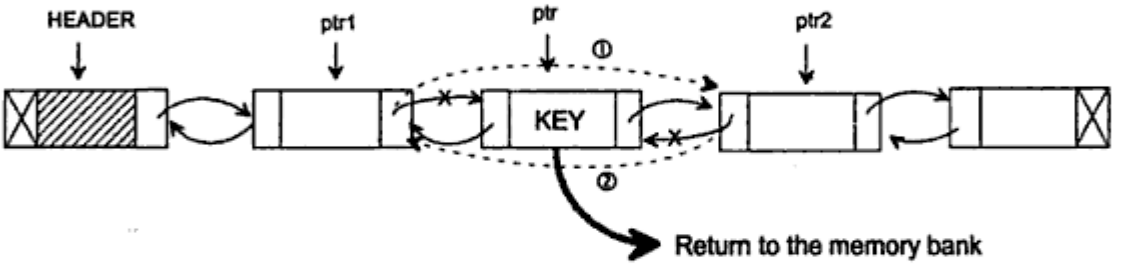
1. ptr = HEADER.RLINK //Pointer to the first node
2. If (ptr = NULL) then //If the list is empty
   1. Print "List is empty: No deletion is made"
   2. Exit
3. Else
   1. ptr1 = ptr.RLINK //Pointer to the second node
   2. HEADER.RLINK = ptr1 //Change the pointer shown as 1 in Figure 3.12(a)
   3. If (ptr1 ≠ NULL) //If the list contains a node after the first node //of deletion
      1. ptr1.LLINK = HEADER //Change the pointer shown as 2 in Figure 3.12(a)



(a) Deletion at the front



(b) Deletion at the end



(c) Deletion at an intermediate position

Fig. 3.12 Deletion at various position in a double linked list.

4. EndIf
5. RETURNNODE (ptr) //Return the deleted node to the memory bank

4. EndIf
5. Stop

Note that the above algorithm will work even if the list is empty.

Deletion of a node at the end of a double linked list. The algorithm is as under:

Algorithm DELETE_DL_END( )
Input: A double linked list with data.
Output: A reduced double linked list.
Data structure: Double linked list structure whose pointer to the header node is HEADER.

Steps:

1. ptr = HEADER
2. While (ptr.RLINK ≠ NULL) do //Move to the last node
   1. ptr = ptr.RLINK
3. EndWhile

4. If (ptr = HEADER) then //If the list is empty
   1. Print "List is empty: No deletion is made"
   2. Exit //Quit the program
5. Else
   1. ptr1 = ptr.LLINK //Pointer to the last but one node
   2. ptr1.RLINK = NULL //Change the pointer shown as 1 in Figure 3.12(b)
   3. RETURNNODE(ptr) //Return the node to the memory bank
6. EndIf
7. Stop

Deletion of a node at any position in a double linked list. The algorithm is as follows:

Algorithm DELETE_DL_ANY(KEY)
Input: A double linked list with data, and KEY, the data content of the key node to be deleted.
Output: A double linked list without a node having data content KEY, if any.
Data structure: Double linked list structure whose pointer to the header node is HEADER.

Steps:

1. ptr = HEADER.RLINK //Move to the first node
2. If (ptr = NULL) then
   1. Print "List is empty: No deletion is made"
   2. Exit
3. EndIf //Quit the program
4. While (ptr.DATA ≠ KEY) and (ptr.RLINK ≠ NULL) do
   //Move to the desired node
   1. ptr = ptr.RLINK
5. EndWhile
6. If (ptr.DATA = KEY) then //If the node is found
   1. ptr1 = ptr.LLINK //Track to the predecessor node
   2. ptr2 = ptr.RLINK //Track to the successor node
   3. ptr1.RLINK = ptr2 //Change the pointer as shown 1 in Figure 3.12(c)
   4. If (ptr2 ≠ NULL) then //If the deleted node is the last node
      1. ptr2.LLINK = ptr1 //Change the pointer shown as 2 in Figure 3.12(c)
   5. EndIf
   6. RETURNNODE(ptr) //Return the free node to the memory bank
7. Else
   Print "The node does not exist in the given list"
8. EndIf
9. Stop

Assignment 3.4 In a list, there may be several nodes having the data content as KEY (that is replication of information). Write an algorithm to delete all such node from a double linked list.

3.5 CIRCULAR DOUBLE LINKED LIST

The advantages of both double linked list and circular linked list are incorporated into an another list structure which is called circular double linked list and it is known to be the best of its kind. Figure 3.13 shows a schematic representation of a circular double linked list.

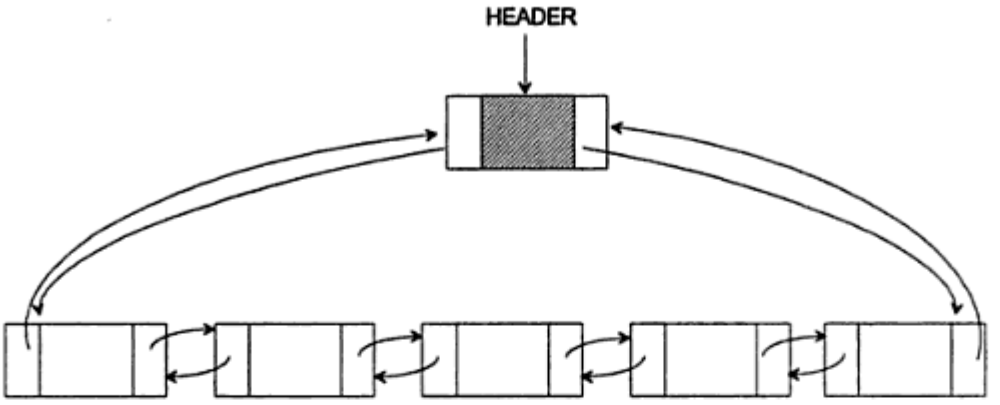


Fig. 3.13 A circular double linked list.

Here, note that the RLINK (right link) of the right most node and LLINK (left link) of the left most node contain the address of the header node; again the RLINK and LLINK of the header node contain the address of the right most and left most node, respectively. An empty circular double linked list is represented as shown in Figure 3.14. In case of an empty list, both LLINK and RLINK of the header node point to itself.

Fig. 3.14 An empty circular double linked list.

3.5.1 Operations on Circular Double Linked List

All the regular operations like insertion, deletion, traversing, searching, merging, splitting, disposing, etc., can be implemented very easily with circular linked list. Implementation of the said operations are left as an exercise for the reader. Here, only sorting operation is illustrated.

Sorting operation with a circular double linked list. The algorithm is as under:

Algorithm SORT_CDL( )
Input: A circularly double linked with elements.
Output: Sorted version of the circularly double linked list.
Data structures: Circular double linked list structure with HEADER being the pointer to the header node.

Steps:

1. ptr_beg = HEADER.LLINK //Pointer to the first node—the beginning node
2. ptr_end = HEADER.RLINK //Pointer to the last node—the ending node
3. While (ptr_beg ≠ ptr_end) do //To traverse the entire list—outer loop
   1. ptr1 = ptr_beg //ptr1 and ptr2 are
   2. ptr2 = ptr1.RLINK //Two variable pointers

```
3. While (ptr2 ≠ ptr_end) do          //For compare and interchange—inner loop
   1. If ORDER(ptr1.DATA, ptr2.DATA) = FALSE then
                                      //If keys are not in order
      1. SWAP (ptr1, ptr2)            //Interchange the data content at ptr1 and ptr2
   2. EndIf
   3. ptr1 = ptr1.RLINK               //Move the first pointer to the next
   4. ptr2 = ptr2.RLINK               //Move the second pointer to the next
4. EndWhile
5. ptr_end = ptr_end.LLINK            //Right most node is now sorted out
4. EndWhile
5. Stop
```

In the above algorithm, we have assumed the procedure ORDER(data1, data2). To test whether two data are in a required order or not; it will return TRUE if they are in order else FALSE. We also assume another procedure SWAP(ptr1, ptr2) to interchange the data content at the nodes pointed by the pointers ptr1 and ptr2. These procedures are very easy to implement.

The above algorithm uses the bubble sorting technique. Execution of each outer loop bubbles up the largest node towards the right end of sorting (say, in ascending order) and each inner loop is to compare between the successive nodes and push up the largest towards the right if they are not in order. Figure 3.15 illustrates the sorting procedure.

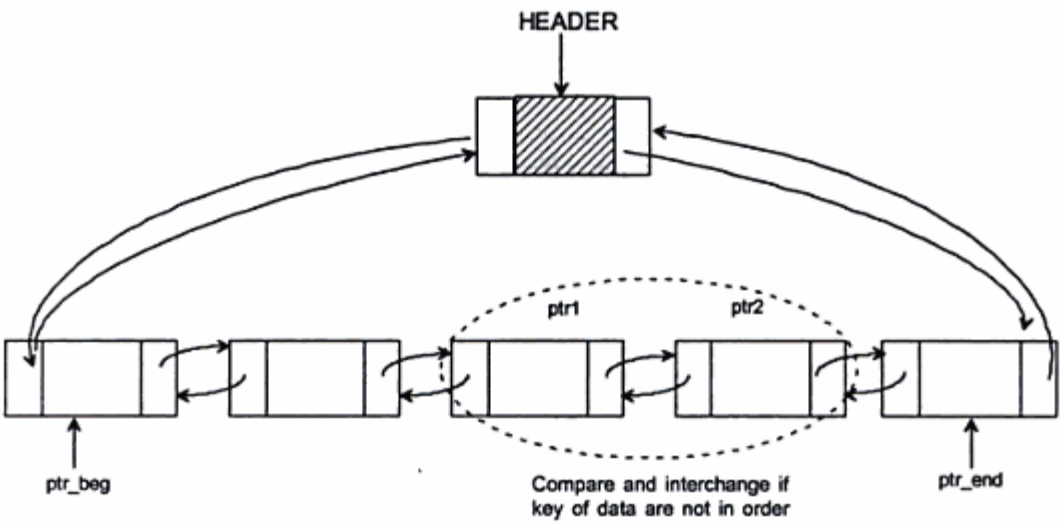


Fig. 3.15 Sorting operation and use of various pointers.

Assignment 3.5 A college uses the following structure for a graduate class:

```
1. Student (30)
   2. Name
      3. LAST (12 Characters)
      3. FIRST (12 Characters)
      3. MIDDLE (12 Characters)
   2. RollNo          (15 Alphanumeric characters)
```

2. SCORE
   3. COURSE1 (2.1Numeric)
   3. COURSE2 (2.1Numeric)
   3. COURSE3 (2.1Numeric)
   3. COURSE4 (2.1Numeric)
   3. COURSE5 (2.1Numeric)
2. GPA (2.2 N)
2. CGPA (2.2 N)

Each field having the usual meaning and GPA refers the Grade Point Average and CGPA denotes the Cumulative GPA. Give an idea how the above information can be managed by a data structure like circular double linked list. How many arrays and their type you have to consider for array representation of circular double linked list?

3.6 APPLICATION OF LINKED LISTS

In order to store and process data, linked lists are very popular data structures. This type of data structures hold certain advantages over arrays. First, in case of an array, data are stored in contiguous memory locations, so insertion and deletion operations are quite time consuming, as before in insertion we have to make room for the new element at a desired location by shifting down the trailing elements; similarly, in case of deletion, in order to fill the location of deleted element, all the trailing elements are required to shift upwards. But, in linked lists, it is a matter of only change in pointers. Second, array is based on the static allocation of memory: amount of memory required for an array must be known before hand, once it is allocated we cannot expand its size. This is why, for an array, general practice is to allocate memory, which is much more than the memory that actually will be used. But this is simply a wastage of memory space. This problem is not there in linked lists. Linked list uses dynamic memory management scheme; memory allocation is decided during the run-time as and when require. Also if a memory is no more required, it can be returned to the free storage space, so that other module or program can utilize it. Third, a program using an array may not execute although the memory required for the data are available but not in contiguous locations rather dispersed. As link structures do not necessarily require to store data in adjacent memory location, so the program of that kind, using linked list can then be executed.

However, there are obviously some disadvantages: one is the pointer business. Pointers, if not managed carefully, may lead to serious errors in execution. Next, linked lists consume extra space than the space for actual data as we are to maintain the links among the nodes. Frankly speaking these drawbacks are nothing compared to what gain we are achieving. This is why, use of this structure is appreciable, wherever it is required to manipulate data. In the next few sections, we illustrate how linked lists can be applied in various applications.

3.6.1 Sparse Matrix Manipulation

During the discussion of sparse matrices, in Chapter 2, Section 2.4.2, it was mentioned that, linked lists are the best solution to store them. Let us first decide what should be the node structure so that using that kind of node we can represent any sparse matrix. Figure 3.16 shows a node structure for the same.

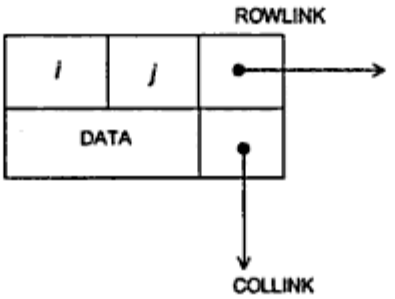


Fig. 3.16 Structure of a node to represent sparse matrices.

In Figure 3.16, fields i and j store the row and column numbers for a matrix element respectively. DATA field stores the matrix element at the i-th row and the j-th column, i.e., a_{ij}. The ROWLINK points to the next node in the same row and COLLINK points the next node in the same column. The principle is that, all the nodes particularly in a row (column) are circular linked with each other; each row (column) contains a header node. Thus, for a sparse matrix of order $m \times n$, we have to maintain m headers for all rows and n headers for all columns, plus one extra node use of which can be evident from Figure 3.17(b). For an illustration, a sparse matrix of order 6×5 is assumed as shown in Figure 3.17(a).

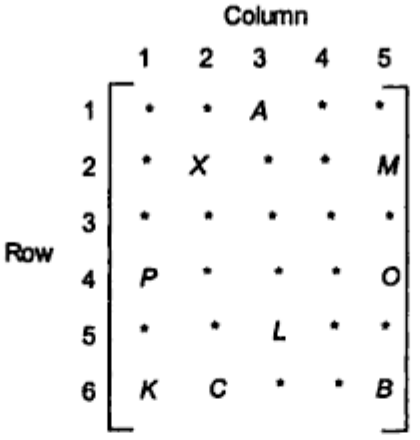
Figure 3.17(b) describes the representation of a sparse matrix. Here, CH1, CH2, ..., CH5 are the 5 headers heading 5 columns and RH1, RH2, ..., RH6 are the 6 headers heading 6 rows. HEADER is one additional header node keeping the starting address of the sparse matrix. Carefully observe the links among various nodes and compare them with the sparse matrix assumed.

Note that, with this representation, any node is accessible from any other node. Now let us consider the algorithm MAKE_SPARSE_&_INSERT to create a linked list to store a sparse matrix.

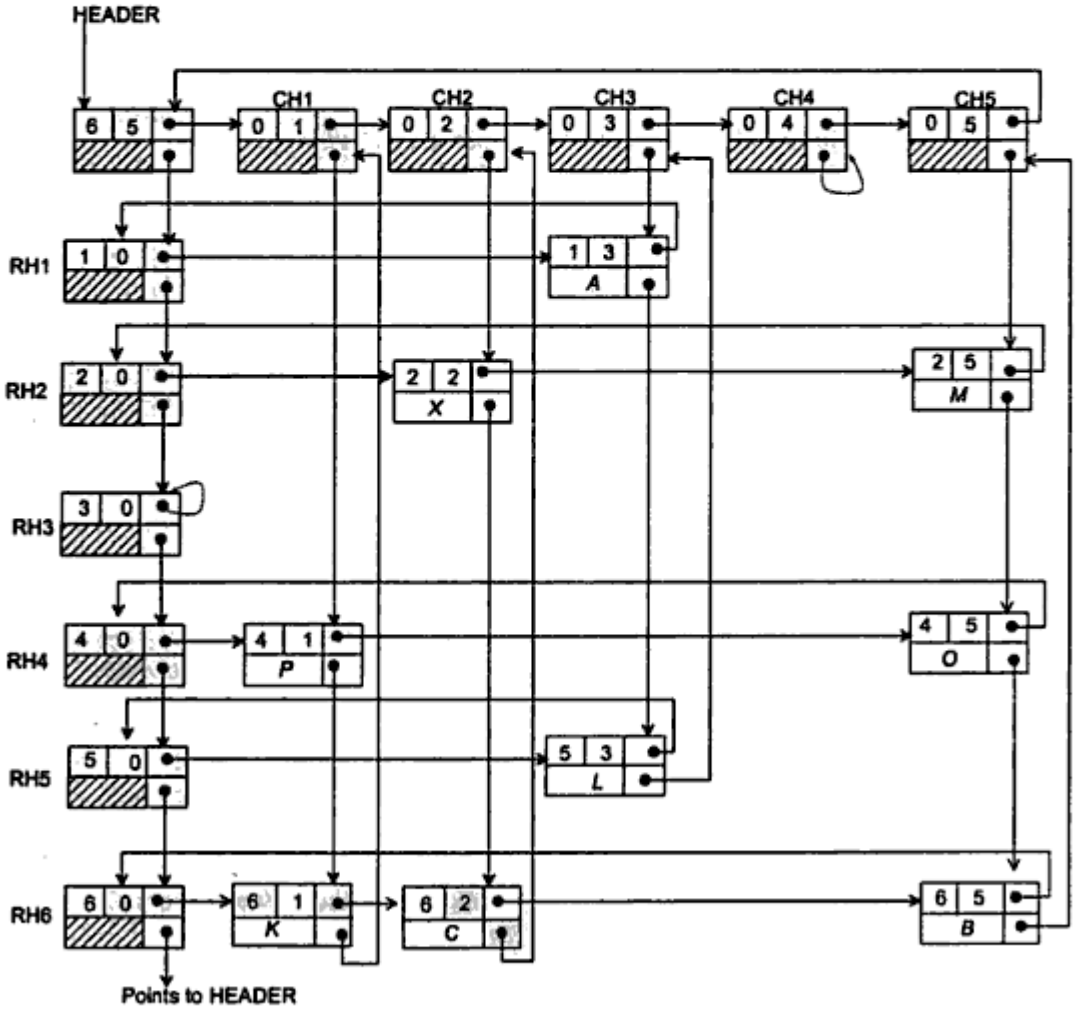
Algorithm MAKE_SPARSE_&_INSERT( )
Input: An $m \times n$ sparse matrix.
Output: Linked representation of the sparse matrix.
Data structures: Linked structure for sparse matrix.

Steps:

1. Read m, n //m—number of rows, n—number of columns
   //Get a node for header//
2. HEADER = GETNODE(NODE) //Get a node from free storage
3. If (HEADER = NULL) then
   1. Print "Non-availability of storage space: Quit"
   2. Exit //The program is aborted
4. Else
   //Initialize the node HEADER as the matrix is initially empty//
   1. HEADER.i = m
   2. HEADER.j = n
   3. HEADER.ROWLINK = HEADER
   4. HEADER.COLLINK = HEADER
   5. HEADER.DATA = NULL



(a) A sparse matrix of order 6 × 5 containing 9 elements only



(b) Linked list representation of a sparse matrix

Fig. 3.17 A sparse matrix and its linked list representation.

```
//Get header nodes for column header//
6.  ptr = HEADER
7.  For col = 1 to n do
    1. new = GETNODE(NODE)
    2. new.i = 0, new.j = col, new.DATA = NULL
    3. ptr.ROWLINK = new
    4. new.ROWLINK = HEADER, new.COLLINK = new
    5. ptr = new
8.  EndFor
    //Get header nodes for row header//
9.  ptr = HEADER
10. For row = 1 to m
    1. new = GETNODE(NODE)
    2. new.i = row, new.j = 0, new.DATA = NULL
    3. ptr.COLLINK = new
    4. new.COLLINK = HEADER, new.ROWLINK = new
    5. ptr = new
11. EndFor
    //Insertion of elements a_ij into the sparse matrix//
12. Read (data, row, col)          //Read the element to be inserted and its position
13. rowheader = HEADER.COLLINK, colheader = HEADER.ROWLINK
    //Go to the row header of the i-th row//
14. While (row < rowheader.i)
    1. rowheader = rowheader.COLLINK
15. EndWhile
    //Go to the column header of the j-th column//
16. While (col < colheader.j)
    1. colheader = colheader.ROWLINK
17. EndWhile
    //Find the position for insertion//
18. rowptr = rowheader             //ON ROW
19. While (rowptr.j < col) do      //This is to find the predecessor and successor
20. 1. ptr1 = rowptr               //ptr1 points the predecessor i.e. the node at
                                   //j-th column
    2. rowptr = rowptr.ROWLINK     //rowptr is the pointer to successor
    3. If (rowptr = rowheader) then
       1. Break                    //Exit the loop
    4. EndIf
21. EndWhile
22. colptr = colheader             //On Column
23. While (colptr.i < row)         //This is to find the predecessor and successor
    1. ptr2 = colptr               //ptr2 points the predecessor, i.e. the node in
                                   //i – 1-th row
    2. colptr = colptr. COLLINK    //colptr is the pointer to successor
    3. If (colptr = coolheaded)
       1. Break                    //Exit the loop
    4. EndIf
24. EndWhile
```

```
    //Insert the node//
25. new = GETNODE(NODE)          //Get a new node from the free storage
26. If (new = NULL) then
    1. Print "Non-availability of storage space: Quit"
    2. Exit                      //The program is aborted
27. Else                         //Initialize the node to be inserted
    1. ptr1.ROWLINK = new
    2. ptr2.COLLINK = new
    3. new.ROWLINK = rowptr
    4. new.COLLINK = colptr
    5. new.DATA = KEY
28. EndIf
29. If more insert then
    1. Go to Step 4.12
30. EndIf
5. EndIf
6. Stop
```

Assignment 3.6

(a) Write an algorithm to print a sparse matrix which is stored using a linked list.
(b) Delete an element from the sparse matrix in linked list form.
(c) For a given element, search a sparse matrix which is stored in a linked list, if the element is present then print its index, that is its row and column position.
(d) Write a program that will dispose a sparse matrix in linked list form to the pool of free storage.

3.6.2 Polynomial Representation

An important application of linked list is to represent polynomials and their manipulations. Main advantages of linked list for polynomial representation is that it can accommodate a number of polynomials of growing sizes so that their combined size does not exceed the total memory available. Methodology of representing polynomials and operations on them are discussed in this section. First, let us consider the case of representation of polynomials.

Polynomial with single variable

Let us consider the general form of a polynomial with single variable:

$$P(x) = a_n x^{e_n} + a_{n-1} x^{e_{n-1}} + \cdots + a_1 x^{e_1}$$

where $a_i x^{e_i}$ is a term in the polynomial so that a_i is a non-zero coefficient and e_i is the exponent. We will assume an ordering of the terms in the polynomial such that $e_n > e_{n-1} > \ldots > e_2 > e_1 \geq 0$. The structure of a node in order to represent a term can be decided as shown below:

Considering the single linked list representation, a node should have three fields: COEFF (to store the coefficient a_i), EXP (to store the exponent e_i) and a LINK (to store the pointer to the next node representing next term). It is evident that number of nodes required to represent a polynomial is the same as the number of terms in the polynomial. An additional node may be considered for a header. As an example, let us consider the single linked list representation of the polynomial $P(x) = 3x^8 - 7x^6 + 14x^3 + 10x - 5$ would be stored as shown in Figure 3.18.

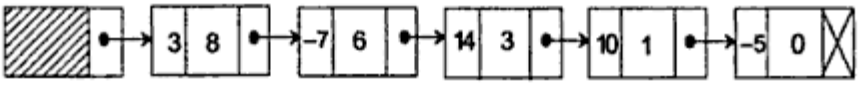


Fig. 3.18 Linked list representation of a polynomial (single variable).

Note that, the terms whose coefficients are zero, are not stored here. Next let us consider two basic operations, viz., addition and multiplication of two polynomials with this representation.

Polynomial addition. In order to add two polynomials, say, P and Q to get a resultant polynomial R, we have to compare their terms starting at the first nodes and moving towards the end one-by-one. Two pointers Pptr and Qptr are used to move along the terms of P and Q. There may be three cases during the comparison between terms in two polynomials.

(i) Case 1: Exponents of two terms are equal. In this case coefficients in two nodes are to be added and a new terms will be created with the values

$$\text{Rptr.COEFF} = \text{Pptr.COEFF} + \text{Qptr.COEFF}$$

and

$$\text{Rptr.EXP} = \text{Pptr.EXP}$$

(ii) Case 2: Pptr.EXP > Qptr.EXP, i.e. the exponent of the current term in P is greater than the exponent of the current term in Q. Then, a duplicate of the current term in P is created and to be inserted in the polynomial R.

(iii) Case 3: Pptr.EXP < Qptr.EXP, i.e. the case when exponent of the current term in P is less than the exponent of the current term in Q. In this case, a duplicate of the current term of Q is created and to be inserted in the polynomial R. The algorithm POLYNOMIAL_ADD is described as below:

Algorithm POLYNOMIAL_ADD(PHEADER, QHEADER; RHEADER)
Input: Two polynomials P and Q whose header pointers are PHEADER and QHEADER
Output: A polynomial R is the sum of P and Q having header as RHEADER.
Data structure: Single linked list structure for representing a term in single variable polynomial.

Steps:

1. Pptr = PHEADER.LINK, Qptr = QHEADER.LINK
   //Get a header node for resultant polynomial//
2. RHEADER = GETNODE(NODE)
3. RHEADER.LINK = NULL, RHEADER.EXP = NULL, RHEADER.COEFF = NULL
4. Rptr = RHEADER //Current pointer to the resultant polynomial *R*

In case of *static storage management* scheme, the net amount of memory required for various data for a program is allocated before the starting of the execution of the program. Once memory is allocated, it neither can be extended nor can be returned to the memory bank for the use of other programs at the same time. On the other hand, *dynamic storage management* scheme allows the user to allocate and deallocate as per the necessity during the execution of programs. This dynamic memory management scheme is suitable in multiprogramming as well as single-user environment where generally more than one program reside in the memory and their memory requirement can be known only during their execution. Operating systems (OS) generally provides the service of dynamic memory management. The data structure for implementing such a scheme is linked list. Following few sections discuss the various principles on which the dynamic memory management scheme is based on; these principles are listed below:

1. Allocation schemes: here we discuss how a request for a memory block will be serviced. There are two strategies:
   (a) Fixed block allocation
   (b) Variable block allocation. There are four strategies under this:
      (i) First fit and its variant
      (ii) Next fit
      (iii) Best fit
      (iv) Worst fit
2. Deallocation schemes: here we discuss how to return a memory block to the memory bank whenever it is no more required. Two strategies are known there:
   (i) Random deallocation
   (ii) Ordered deallocation.

We will discuss two more systems for the implementation of allocation and deallocation schemes:

1. Boundary tag system
2. Buddy system

There is, again, one more principle called *garbage collection*, to maintain a memory bank so that it can be utilized efficiently.

Now we discuss below the dynamic memory management schemes and possible use of linked list therein.

3.7 MEMORY REPRESENTATION

Memory bank or *pool* of free storages are often a collection of non-contiguous blocks of memory. Their linearity can be maintained by means of pointers between one block to the another or in other words, memory bank is a linked list where links are to maintain the adjacency of blocks. Regarding the size of the blocks there are two practices: fixed block storage and variable block storage. Let us discuss each of them individually.

3.7.1 Fixed Block Storage

This is the simplest storage maintenance method. Here each block is of the same size. The size

is determined by the system manager (user). Here, the memory manager (a program of OS) maintains a pointer AVAIL which points a list of non-contiguous memory blocks. Figure 3.19 shows a memory bank with fixed size blocks.

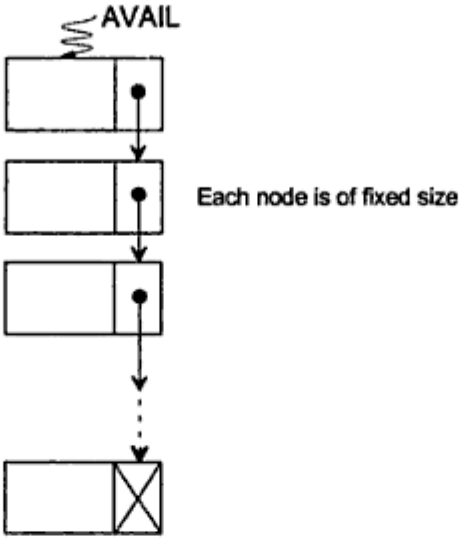


Fig. 3.19 Pool of free storages with fixed size blocks.

A user program communicates with the memory manager by means of two functions GETNODE(NODE) and RETURNNODE(ptr) which are discussed as below:

Procedure GETNODE(NODE) //This procedure avails a block from memory bank
1. If (AVAIL = NULL) then
   1. Print "The memory is insufficient"
2. Else
   1. ptr = AVAIL
   2. AVAIL = AVAIL.LINK
   3. Return(ptr) //Return pointer of the available block to the caller
3. EndIf
4. Stop

The procedure GETNODE is to get a memory block to store data of type NODE (by passing this as argument we are to mention the size of the memory block required). This procedure when invoked by a program, returns a pointer to the first block in the pool of free storage. The AVAIL then points to the next block. The link modification is shown (by the dotted line) in Figure 3.20. If AVAIL = NULL, it indicates that no more memory is available for allocation.

Similarly, whenever a memory block is no more required, it can be returned to the memory bank through a procedure RETURNNODE( ) which is stated as below:

Procedure RETURNNODE(PTR)
1. ptr1 = AVAIL
2. While (ptr1.LINK ≠ NULL) do //Move to the end of the list
   1. ptr1 = ptr1.LINK

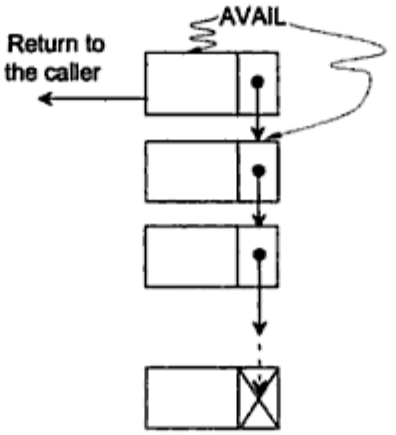


Fig. 3.20 Getting a block from the memory bank.

3. EndWhile
4. ptr1.LINK = PTR
5. PTR.LINK = NULL
6. Stop

The procedure RETURNNODE( ) append a returned block (bearing pointer PTR) at the end of the pool of free storage pointed by AVAIL. Change in pointers can be seen in the Figure 3.21 as dotted line.

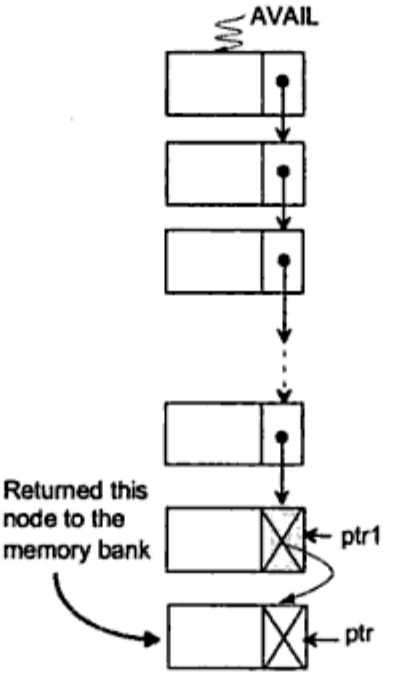


Fig. 3.21 Returning a block to the memory bank.

So far the implementation of fixed block allocation is concerned, this is the most simple strategy. But the main drawback of this strategy is the wastage of space. For example, suppose each memory block is of size 1 K (1024 bytes); now for a request of memory block, say, of size 1.1 K we have to avail 2 blocks (that is 2 K memory space) thus wasting 0.9 K memory space. Making the size of block too small reduces the wastage of space, however, it reduces the overall performance of the scheme.

3.7.2 Variable Block Storage

To overcome the disadvantages of fixed block storage, we can maintain blocks of variable sizes, instead of fixed size blocks. Procedures for GETNODE(NODE) and RETURNNODE(PTR) in this storage representation are stated below:

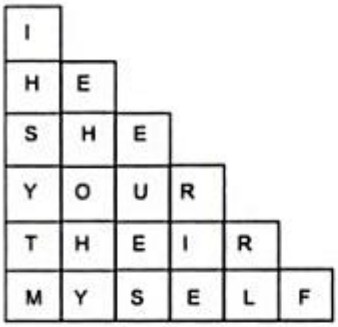
Procedure GETNODE(NODE) //To avail a block from a pool of variable sized blocks
1. If (AVAIL = NULL) then
   1. Print "Memory bank is insufficient"
   2. Exit //Quit the program
2. EndIf
3. ptr = AVAIL
4. While (ptr.LINK ≠ NULL) and (ptr.SIZE < SIZEOF(NODE)) do //Move to the right block
   1. ptr1 = ptr
   2. ptr = ptr.LINK
5. EndWhile
6. If (ptr.LINK = NULL) and (ptr.SIZE < SIZEOF(NODE)) then
   1. Print "Memory request is too large: Unable to serve"
7. Else
   1. ptr1.LINK = ptr.LINK
   2. Return(ptr)
8. EndIf
9. Stop

This procedure assumes that blocks of memory are stored in ascending order of their sizes. Node structure maintains a field to store the size of a block, namely SIZE. SIZEOF(NODE) is a procedure that will return the size of a node (*see* Figure 3.22).

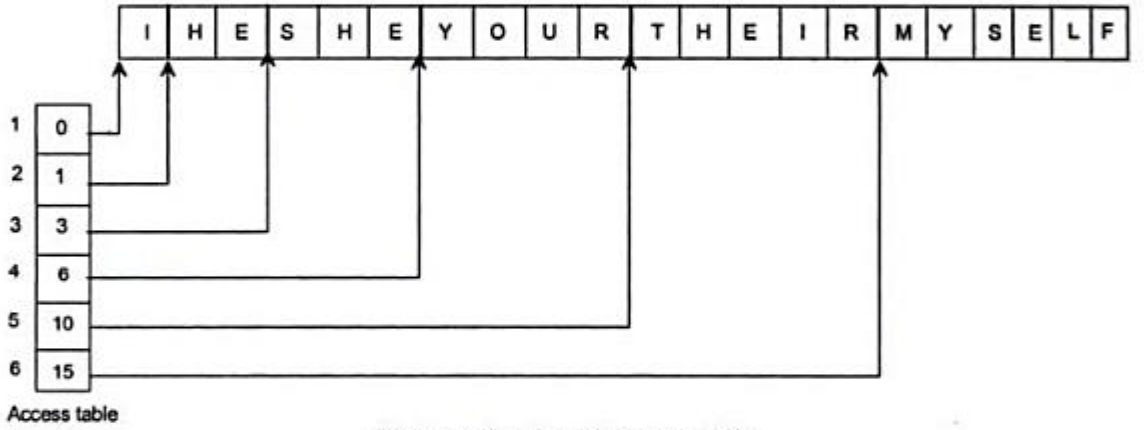
Note that the above procedure will return a block of exactly same size or more than the size that a user program requests.

Next let us describe the procedure RETURNNODE(PTR) to dispose a block into the pool of free storage in the ascending order of the block sizes.

Procedure RETURNNODE(PTR)
1. ptr1 = AVAIL
2. While (ptr1.SIZE < ptr.SIZE) and (ptr1.LINK ≠ NULL)) do //Move to the right position
   1. ptr2 = ptr1
   2. ptr1 = ptr1.LINK
3. EndWhile
4. ptr2.LINK = PTR
5. PTR.LINK = ptr1
6. Stop

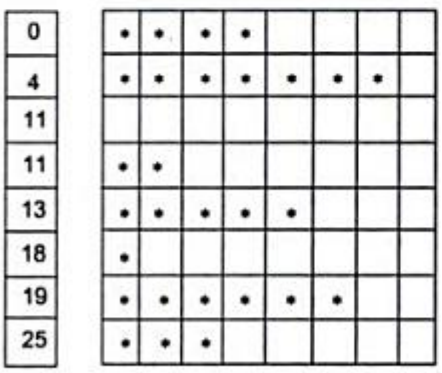


(a) A jagged table and its row-major order

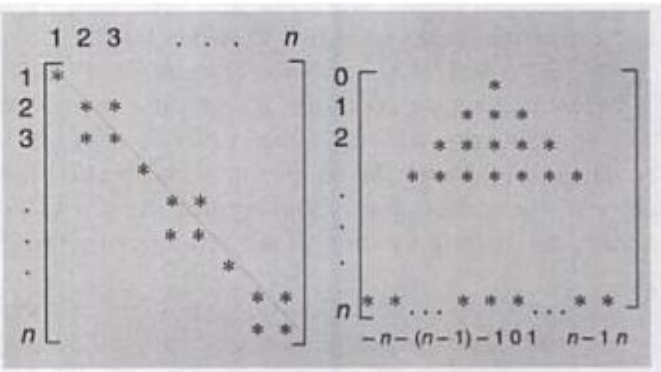


(b) Accessing through access table

Fig. 6.3 Access technique of a jagged table.



(a) Accessing of elements in an asymmetric matrix (jagged table).



(b) Two jagged tables for Assignment 6.1

Fig. 6.4 Jagged tables.

6.3 INVERTED TABLES

It will be judicious if we discuss the concept of inverted tables with the following example.

Suppose, a telephone company maintains records of all the subscribers of a telephone exchange as shown in Table 6.1. These records can be used to serve several purpose. One of them requires the alphabetical ordering of the name of the subscriber (say, in order to put the name of the subscriber into telephone directory). Second, it requires the lexicographical ordering of the address of subscriber (say, it is required for routine maintenance). Third, it also requires the ascending order of the telephone numbers (say, in order to estimate the cabling charge from the telephone exchange to the location of telephone connection) etc. To serve all these purpose, the telephone company should maintain three sets of records: one in alphabetical order of the NAME, second, the lexicographical ordering of the ADDRESS and third, the ascending order of the phone numbers. But this way of maintaining records leads to the following serious drawbacks:

1. Requirement of extra storage: three times of the actual memory.
2. Difficult in modification of records: if a subscriber change his address then we have to modify this in three storage otherwise consistency in information will be lost.

However, using the concept of inverted tables, we can avoid the multiple sets of records, and we can still retrieve the records by any of the three keys almost as quickly as if the records are fully sorted by that key. Therefore, we should maintain an inverted table. In this case, this table comprise of three columns: NAME, ADDRESS, and PHONE as shown in Table 6.1(b). Each column contains the index numbers of records in the order based on the sorting of the corresponding key. This inverted table, therefore, can be consulted to retrieve information.

Table 6.1 Multi-key Access and Its Inverted Table

(a) Records of a Telephone Exchange

Index	*Name*	*Address*	*Phone*
1	K.R. Narayana	Maker Towers #6	257696
2	A.B. Vajpayee	9 Vivekananda Road	257459
3	L.K. Advani	11 Von Kasturba Marg	257583
4	Mamta Banerjee	342 Patel Avenue	257423
5	Y. Sinha	5 SBI Road	257504
6	D. Kulkarni	369 Faculty Colony	257564
7	T. Krishnamurthy	185 Faculty Colony	257579
8	N. Puranjay	409 Medical Colony	257409
9	Tadi Tabi	Officers Mess #52	257871

(b) Inverted Table

Name	*Address*	*Phone*
2	7	8
6	6	4
1	1	2
3	8	5
4	9	6
8	4	7
7	5	3
9	2	1
5	3	9

6.4 HASH TABLES

There are another type of tables which help us to retrieve information very efficiently. The ideal *hash table* is merely an array of some constant size, the size depends on the application where

Algorithm BUILD_TREE1(*I*, ITEM) //A binary tree currently with node at *I*
Input: ITEM is the data content of the node *I*.
Output: A binary tree with two sub-trees of node *I*.
Data structure: Array representation of tree.

Steps:

```
1. If (I ≠ 0) then                          //If the tree is not empty
   1. A[I] = ITEM                           //Store the content of the node I into the array A
   2. Node I has left sub-tree (Give option = Y/N)?
   3. If (option = Y) then                  //If node I has left sub tree
      1. BUILD_TREE1(2 * I, NEWL)           //Then it is at 2*I with next item as NEWL
   4. Else
      1. BUILD_TREE1(0, NULL)               //Empty sub-tree
   5. EndIf
   6. Node I has right sub-tree (Give option = Y/N)?
   7. If (option = Y)                       //If node I has right sub-tree
      1. BUILD_TREE1 (2*I+1, NEWR)          //Then it is at 2*I+1 with next item as NEWR
   8. Else
      1. BUILD_TREE1(0, NULL)               //Empty sub-tree
   9. EndIf
2. EndIf
3. Stop
```

Algorithm BUILD_TREE2(PTR, ITEM)
Input: ITEM is the content of the node with pointer PTR.
Output: A binary tree with two sub-trees of node PTR.
Data structure: Linked list structure of binary tree.

Steps:

```
1. If (PTR ≠ NULL) then                     //If the tree is not empty
   1. PTR.DATA = ITEM                       //Store the content of node at PTR
   2. Node PTR has left sub-tree (Give option = Y/N)?
   3. If (option = Y) then
      1. lcptr = GETNODE(NODE)              //Allocate memory for the left child
      2. PTR.LC = lcptr                     //Assign it to Left link
      3. BUILD_TREE2(lcptr, NEWL)           //Build left sub-tree with next item as NEWL
   4. Else
      1. lcptr = NULL
      2. PTR.LC = NULL                      //Assign for an empty left sub-tree
      3. BUILD_TREE2(lcptr, NULL)           //Empty sub-tree
   5. EndIf
   6. Node PTR has right sub-tree (give option = Y/N)?
   7. If (option = Y) then
      1. rcptr = GETNODE(NODE)              //Allocate memory for the right child
      2. PTR.RC = rcptr                     //Assign it to Right link
      3. BUILD_TREE2(rcptr, NEWR)           //Build right sub-tree with next item as NEWR
```

```
    8. Else
        1. rcptr = NULL
        2. PTR.RC = NULL                    //Assign for an empty right sub-tree
        3. BUILD_TREE2(rcptr, NULL)
    9. EndIf
2. EndIf
3. Stop
```

7.4 OPERATIONS ON BINARY TREE

Major operations on a binary tree can be listed as:

1. *Insertion.* To include a node into an existing (may be empty) binary tree.
2. *Deletion.* To delete a node from a non-empty binary tree.
3. *Traversal.* To visit all the nodes in a binary tree.
4. *Merge.* To merge two binary trees into a larger one.

Some other special operations are also known for special cases of binary trees. These will be mentioned later on. Now, let us discuss the above mentioned operations for general binary trees.

7.4.1 Insertion

With this operation, a new node can be inserted into any position in a binary tree. Inserting a node as an internal node is generally based on some criteria which is usually in context of a special form of a binary tree. We will discuss here a simple case of insertion as external nodes. Figure 7.18 shows how a node containing data content as *G* can be inserted as a left child of a node having data content *E*. Two algorithms for two different storage of representations (viz., sequential storage and linked storage) are stated below.

Insertion procedure, in fact, is a two-step process: first to search for the existence of a node

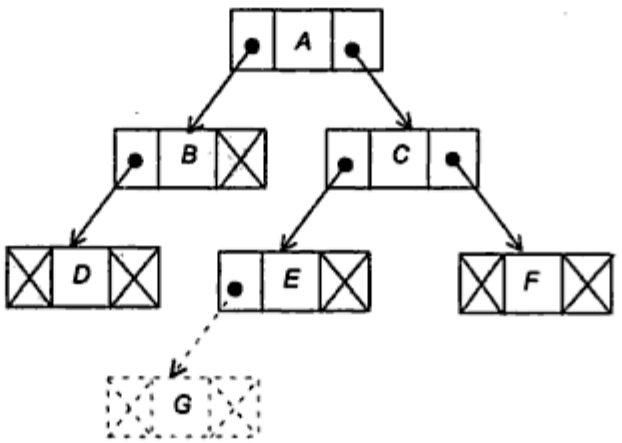


Fig. 7.18 Insertion of a node as an external node into a binary tree.

in the given binary tree after which an insertion to be made, and second is to establish a link for the new node.

Insertion into a sequential representation of a binary tree (as a leaf node)

The following algorithm uses a recursive procedure SEARCH_SEQ to search for a node containing data as KEY.

Algorithm INSERT_SEQ_BIN_TREE(KEY, ITEM)

Input: KEY be the data of a node after which a new node has to be inserted with data as ITEM.

Output: Newly inserted node with data as ITEM as a left or right child of the node with data KEY.

Data structure: Array *A* storing the binary tree.

Steps:

```
1. l = SEARCH_SEQ(1, KEY)                          //Search for the key node in the tree
2. If (l = 0) then
   1. Print "Search is unsuccessful : No insertion"
   2. Exit
3. EndIf                                           //Quit the execution
4. If (A[2 * l] = NULL) or (A[2 * l + 1] = NULL) then  //If the key node has empty link(s)
   1. Read option to read as left (L) or right (R) //child (give option = L/R)
   2. If (option = L) then
      1. If A[2 * l] = NULL then                   //Left link is empty
         1. A[2 * l] = ITEM                        //Store it in the array A
      2. Else                                      //Cannot be inserted as left child
         1. Print "Desired insertion is not possible" //as it already has a left child
         2. Exit                                   //Return to the end of the procedure
      3. EndIf
   3. EndIf
   4. If (option = R) then                         //Move to the right side
      1. If (A[2 * l + 1] = NULL) then             //Right link is empty
         1. A[2 * l + 1] = ITEM                    //Store it in the array A
      2. Else                                      //Cannot be inserted as right child
         1. Print 'Desired operation is not possible' //as it already has a left child
         2. Exit                                   //Return to the end of the procedures
      3. EndIf
   5. EndIf                                        //Key node is having both the child
5. Else                                            //Key node does not have any empty link
   1. Print "ITEM cannot be inserted as a leaf node"
6. EndIf
7. Stop
```

Suppose, T_1 and T_2 are two binary trees. T_2 can be merged with T_1 if all the nodes from T_2, one by one, is inserted into the binary tree T_1 (insertion may be as internal node when it has to maintain certain property or may be as external nodes).

Another way, when the entire tree T_2 (or T_1) is included as a sub-tree of T_1 (or T_2). For this, obviously we need that in either (or both) tree there must be at least one null sub-tree. We will consider in our subsequent discussion, this second case of merging.

Before, going to perform the merging, we have to test for compatibility. If in both the trees, root node have both the left and right sub-trees then merge will fail; otherwise if T_1 has left sub-tree (or right sub-tree) empty then T_2 will be added as the left sub-tree (or right sub-tree) of the T_1 and vice versa. For example, as in Figure 7.27, T_1 has right sub-tree empty hence T_2 is added as the right sub-tree of T_1. Note that if $T(N)$ denotes the number of nodes n in tree T then

$$T(n1 + n2) = T_1(n1) + T_2(n2)$$

where T is the resultant tree of merging T_1 and T_2.

Following is the algorithm MERGE to define the merge operation.

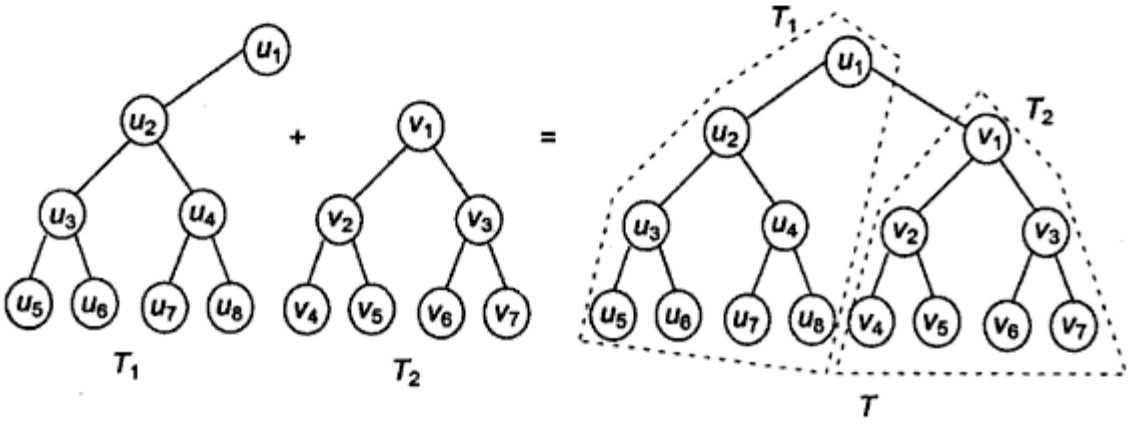


Fig. 7.27 Merging of two binary trees.

Algorithm MERGE($ROOT_1$, $ROOT_2$; ROOT)

Input: Two pointers $ROOT_1$ and $ROOT_2$ are the roots of two binary tree T_1 and T_2 respectively with linked structure.

Output: A binary tree containing all the nodes of T_1 and T_2 having pointer to the root as ROOT.

Data structure: Linked structure of binary tree.

Steps:

1. If ($ROOT_1$ = NULL) then //If T_1 is empty then T_2 is the result
   1. ROOT = $ROOT_2$ //Tree T_2 is the result
   2. Exit
2. Else
   1. If ($ROOT_2$ = NULL) then //If T_2 is empty then T_1 is a result
      1. ROOT = $ROOT_1$
      2. Exit

7.5 TYPES OF BINARY TREES

There are several types of binary trees possible each with its own properties. Few important and frequently used trees are listed as below.

1. Expression tree
2. Binary search tree
3. Heap tree
4. Threaded binary tree
5. Huffman tree
6. Height balanced tree (also known as AVL tree)
7. Decision tree

7.5.1 Expression Tree

An *expression tree* is a binary tree which stores an arithmetic expression. The leaves of an expression tree are operands, such as constants or variable names, and all internal nodes are the operators. Expression tree is always a binary tree because an arithmetic expression contains either binary operators or unary operators (hence an internal nodes has at most two children). Figure 7.29 shows an expression tree for the arithmetic expression

$$(A + B * C) - ((D * E + F)/G)$$

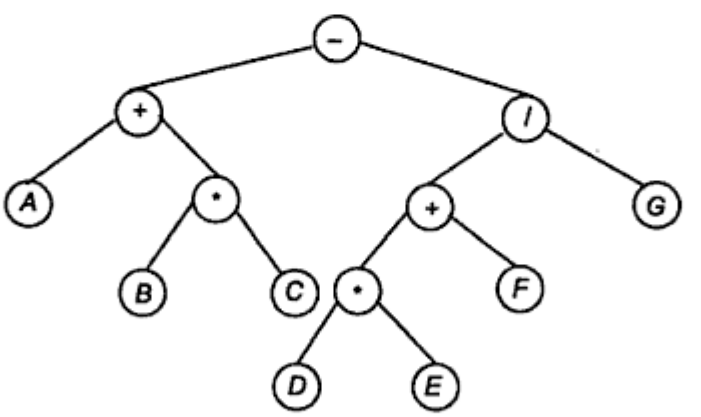


Fig. 7.29 An expression tree.

Construction of an expression tree

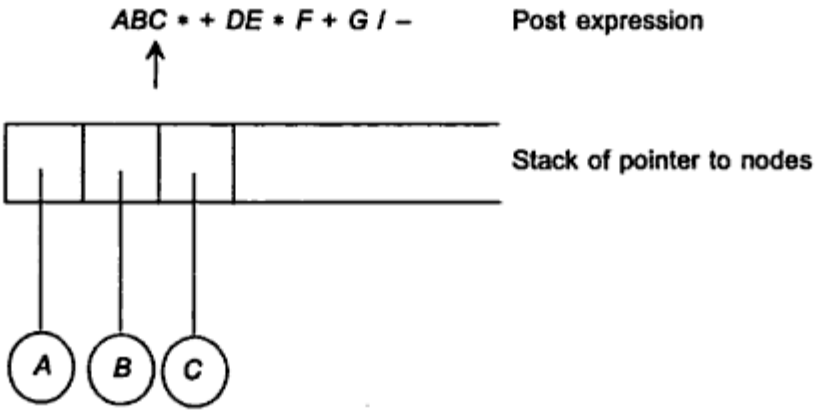
Give an arithmetic expression, our next task is to construct the corresponding expression tree. Generally, we are habituated with arithmetic expressions given in infix notation. But better practice is to construct the expression tree from the postfix notation of the arithmetic expression. Using stack, an expression in infix notation can easily be converted into postfix notation (see Section 4.4.1).

We now give an algorithm to form an expression tree from a postfix notation. First it is illustrated with example then formal description of it will be represented.

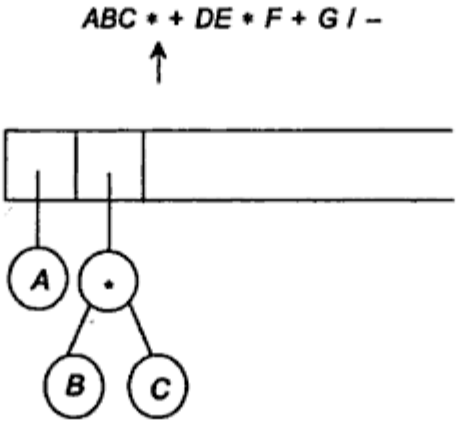
Let us consider a postfix notation of the arithmetic expression as shown in Figure 7.29.

$$ABC * + DE * F + G / -$$

The method is as simple as the evaluation of postfix expression: we have to scan one symbol at a time, if the symbol encountered is an operand we create a one-node tree and push a pointer to it onto a stack. If the symbol is an operator (assume the case of binary operators only), we pop pointers to two trees T_1 and T_2 from the stack (T_1 is popped first) and form a tree whose root is the operator and whose left and right children point to T_2 and T_1 respectively. A pointer to this new tree is then pushed onto the stack. This is illustrated as in Figure 7.30.



(a) First three symbols are scanned, three nodes are formed and pointers to three nodes are pushed onto the stack.



(b) Operator * is encountered, top two pointers are popped, a tree is formed with * as root and pointer to the root node is pushed onto the stack.

Fig. 7.30 Continued.

- Any node can be accessible from any other node. Threads are usually more to upward whereas links are downward. Thus in a threaded tree, one can move in either direction and nodes are in fact circularly linked. This is not possible in unthreaded counter part because there we can move only in downward direction starting from root.
- Insertions into and deletions from a threaded tree are although time consuming operations (since we have to manipulate both links and threads) but these are very easy to implement.

Various operations on threaded binary trees

Let us investigate the various operations possible on threaded binary trees. We will discuss the operations on inorder threaded binary tree as a representative example.

To find the inorder successor of any node. Given the address of any node in an inorder threaded binary tree we are to find its inorder successors. Finding inorder successor is very straightforward. It can be easily revealed from Figure 7.48(b) that:

- If *N* does not have a right child (that is, if the field RTAG implies a thread) then the thread points to its inorder successor.

Else

- If *N* has a right sub-tree (that is, if RTAG implies a link) then the left-most node in this right sub-tree which does not have any left child is the inorder successor of *N*.

Thus, with these, one can easily verify the following [with reference to Figure 7.48(b)]:

Inorder successor of '*B*' is '*A*'
Inorder successor of '*A*' is '*G*'
Inorder successor of '*D*' is '*F*'
Inorder successor of '*C*' is the HEADER, etc.

Now, we can formulate the procedure INSUCC(*N*) to find the inorder successor of any node *N* in a threaded binary tree. This is given in the following algorithm:

Algorithm INSUCC(PTR)
Input: PTR is the pointer to any node whose inorder successor has to be found.
Output: Pointer to the inorder successor of the node PTR.
Data structure: Linked structure of threaded binary tree.

Steps:

```
1. succ = PTR.RCHILD                    //Get the right child of the node
2. If (PTR.RTAG = FALSE) then           //If the node has right sub-tree
   1. While (succ.LTAG = FALSE)         //Move to the left-most node without left child
      1. succ = succ.LCHILD
   2. EndWhile
3. EndIf
4. Return(succ)                         //Return the pointer to the inorder successor
5. Stop
```

To find the inorder predecessor of any node. Finding the inorder predecessor of a given node is similar to the previous one and can be obtained by simply interchanging the RCHILD by LCHILD and LTAG by RTAG, algorithm INPRED ( ) is as:

tree or we can insert at a desired position. Let us discuss the second case of insertion, that is, insertion of a node into an inorder threaded binary tree.

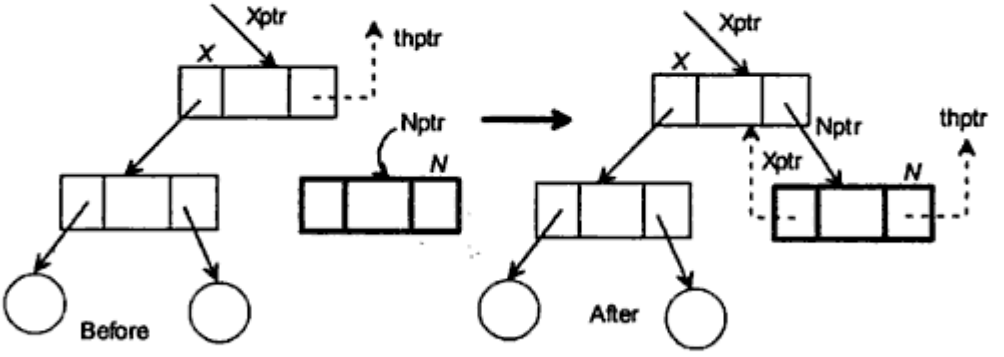
Suppose, there is a node *X* in inorder threaded binary tree whose pointer is given, say, Xptr. We want to insert a node *N* (having its pointer as Nptr) either as left child or as right child. First, we illustrate the pointer modifications required for the purpose. Figure 7.50 illustrates the insertions into an inorder threaded binary tree.

Now, let us discuss the procedure of insertion into an inorder threaded binary tree. Given a node (identified by its data, say *X*), we are to insert another node (let it be *N*) after *X* either as left or as right child. Procedure INSERT_THREAD_IN(...) will first search for the location of *X* in the tree (the tree is recognized by its header pointer HEADER); if found then the node *N* will be inserted as desired. Following is the algorithm INSERT_THREAD_IN.

Algorithm INSERT_THREAD_IN(*X*, *N*)
Inputs: (a) HEADER, the pointer to the header node of the threaded (inorder) binary tree.
(b) *X* is the data of a node after which insertion has to be done.
(c) *N* is the data of a node to be inserted.
Output: If *X* exists in the tree then *N* is inserted after *X*.
Data structure: Linked structure of threaded binary tree.

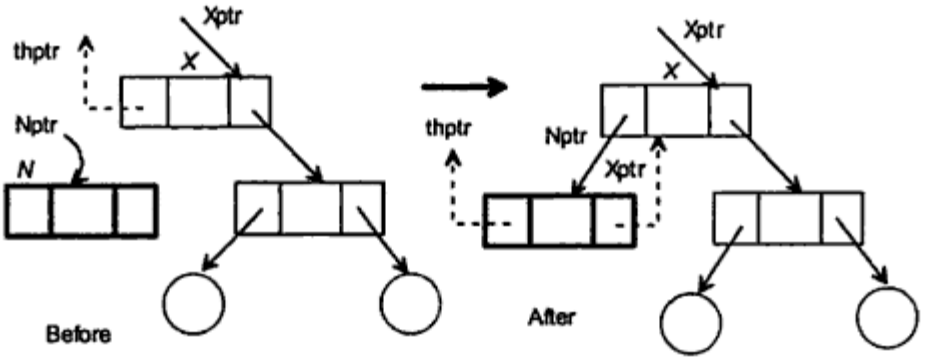
Steps:

```
1. ptr = HEADER.LCHILD                    //Pointer to the root node
2. flag = FALSE                           //Search is not complete
3. While (ptr ≠ HEADER) and (flag = FALSE)
                                          //Continue the searching for X
   1. If (ptr.DATA = X) then
      1. Xptr = ptr                       //Pointer to the node X
      2. flag = TRUE                      //Search is completed
   2. Else
      1. ptr = INSCC(ptr)                 //Go to the next node
   3. EndIf
4. EndWhile
5. If (flag = FALSE) then
   1. Print "Node does not exist: No insertion"
   2. Exit                                //Exit from the insertion
```

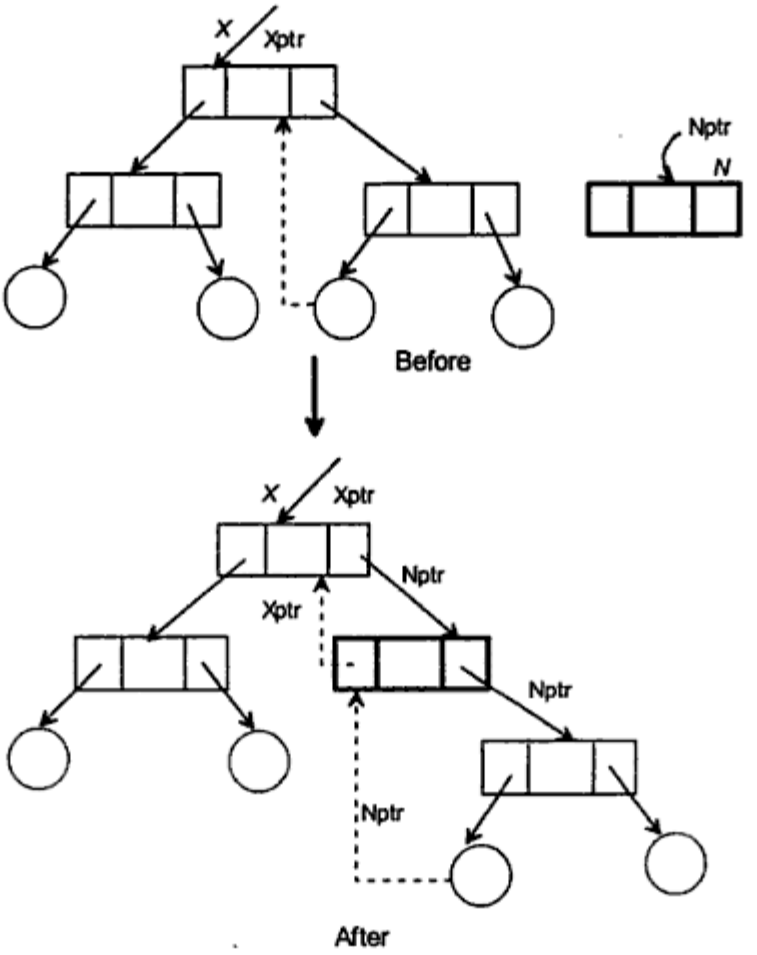


(a) Insert *N* as right child of node *X* when *X* has no right sub-tree

Fig. 7.50 Continued.



(b) Insert *N* as left child of node *X* when *X* has no right sub-tree



(c) Insert *N* as right child of *X* when *X* has non-empty right sub-tree

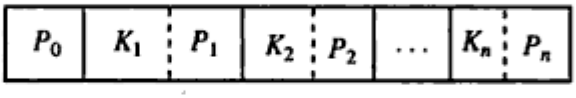
Fig. 7.50 Continued.

Assignment 7.36 For the forest F as shown in Figure 7.85 obtain the preorder and inorder traversals. Find the inorder and preorder traversals on its corresponding binary tree as obtained in Figure 7.86(b). Verify the correspondence among these traversals, if any.

7.7 B TREES

From our previous discussions it is evident that the tree structure is best suitable for maintaining the indices of elements in it. Main purpose of this indexing is to accelerate the search procedure. Binary search tree (BST) uses the concept of tree indexing, where each node contains a key value, pointers to the left sub-tree and right sub-tree. Binary search tree is, in fact, a 2-way search tree and this concept of tree indexing can be generalized for *m-way* ($m \geq 2$) *search tree* ($m = 2$ is the special case for BST) with the following definitions:

1. An m-way search tree T is a tree in which all nodes are of degree $\leq m$.
2. Each node in the tree contains the following attributes:



   where
   $1 \leq n < m$
   K_i $(1 \leq i \leq n)$ are key values in the node
   P_i $(0 \leq i \leq n)$ are pointers to the sub-trees of T.
3. $K_i < K_{i+1}$, $\quad 1 \leq i < n$
4. All the key values in the sub-tree pointed by P_i are less than the key values K_{i+1}, $0 \leq i < n$.
5. All the key values in the sub-tree pointed by P_n is greater than K_n.
6. All the sub-trees pointed by P_i $(0 \leq i \leq n)$ are also m-way search trees.

Figure 7.87 illustrates a 3-way search tree.

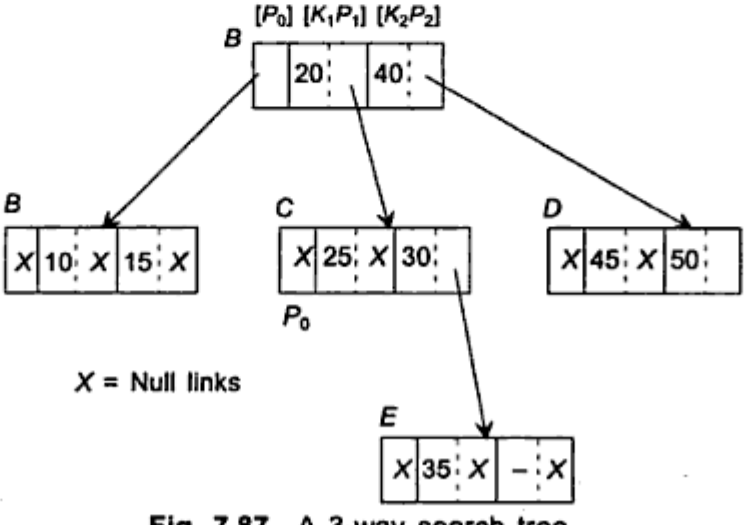


Fig. 7.87 A 3-way search tree.

Lemma 8.4 In a simple digraph, the length of any elementary path is less than or equal to $n - 1$, where n is the number of vertices in the graph. Similarly, the length of any elementary cycle does not exceed n.

Proof The proof is based on the fact that in any elementary path the node appearing in the sequence are distinct. The number of distinct nodes in an elementary path of length l is $l + 1$, since there are only n distinct nodes in the graph, we cannot have an elementary path of length greater than $n - 1$.

For an elementary cycles of length l, the sequence contains l distinct nodes hence the result.

In a graph, all the elementary cycles or path can be determined from the matrix B_n where

$$B_n = A + A^2 + A^3 + \cdots + A^n$$

n being the number of vertices in the graph and A being the adjacency matrix of the graph.

For an example, B_4 of the graph as presented in Figure 8.13, can be obtained as:

$$B_4 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 3 & 4 & 2 & 3 \\ 5 & 5 & 4 & 6 \\ 7 & 7 & 4 & 7 \\ 3 & 2 & 1 & 2 \end{bmatrix} \end{array} \qquad P = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

The element in the i-th row and j-th column of B_4 shows the number of paths having length 4 or less than those exist between v_i to v_j in the graph.

From the matrix B_n, we can obtain the path matrix of the graph as follows: Let $P = [p_{ij}]$ be a path matrix. Then $p_{ij} = 1$ iff there is a non-zero element in the i, j entry of the matrix B_n else $p_{ij} = 0$. Thus, the path matrix of the running example is shown above as P. [*Note:* The path matrix only shows the presence or absence of at least one path between a pair of points and also the presence or absence of a cycle at any node.]

Lemma 8.5

(a) A vertices v_i contains a cycle if the i, i entry in the path matrix P is 1.

(b) A graph is strongly connected if for all $v_i, v_j \in G$, both the i, j entry and j, i entry in the path matrix are 1.

Proof Proof can be easily followed from the previous discussion, and left as an exercise.

As an example, for the graph as given in Figure 8.13, and whose path matrix P is obtained as above, we can conclude that, it is strongly connected and all the vertices have cycle.

Lemma 8.6 Let P be a path matrix of a graph G. If P^T is the transpose of the matrix P, then the i-th row of the matrix $P * P^T$, [which is obtained by the element-wise product (AND) of the elements] gives the strong component containing v_i.

Proof If v_j is reachable from v_i then clearly $p_{ij} = 1$; also if v_i is reachable from v_j, then $p_{ji} = 1$. Therefore the element in the i-th row and j-th column of $P * P^T$ is 1 iff v_i and v_j are mutually reachable. This is true for all j. Hence the result.

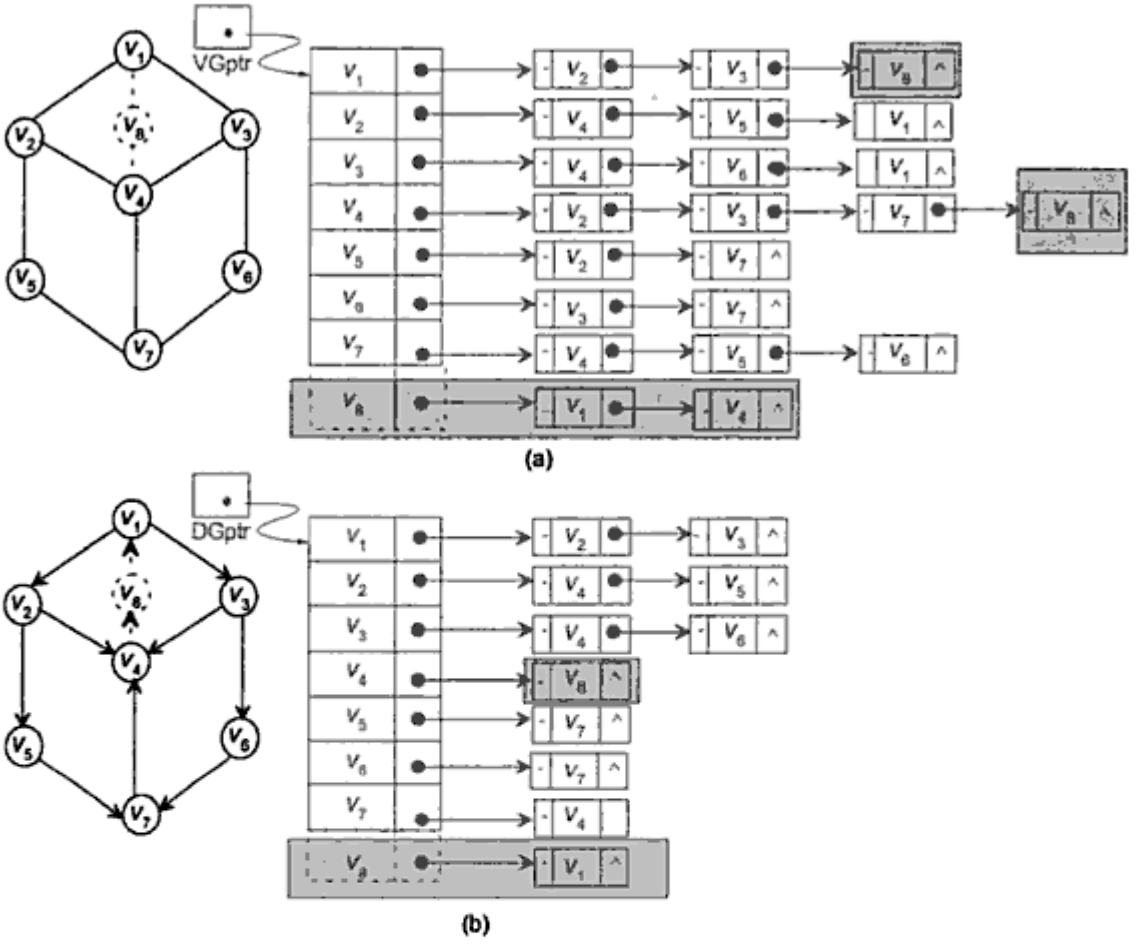


Fig. 8.17 Insertion of a vertex into (a) an undirected graph and (b) a digraph.

In case of insertion of a vertex into an undirected graph, if v_x is inserted and v_i be its adjacent vertex then v_i has to be incorporated in the adjacency list of v_x as well as v_x has to be incorporated in the adjacency list of v_i.

On the other hand, if it is a digraph and if there is a path from v_x to v_i, then we add a node for v_i into the adjacency list of v_x; if there is an edge from v_i to v_x, add a node for v_x in the adjacency list of v_i.

Following Algorithms INSERT_VERTEX_LL_UG is to insert a vertex into a graph and INSERT_VERTEX_LL_DG is to insert a vertex into a digraph.

Algorithm INSERT_VERTEX_LL_UG(V_x, X)

Input: V_x, the new vertex that has to be inserted with
$X = [v_{x1}, v_{x2}, \ldots, v_{xl}]$, l number of adjacent vertices to the vertex V_x.
Let N = Number of vertices currently present in the graph.

Output: A graph with new vertex v_x and its adjacent edges v_{xi}, $i = 1, 2, \ldots, l$, if adjacent vertices exist.

Data structure: Linked structure of undirected graph and UGPtr is the pointer to it.

Steps:

1. $N = N + 1$, $V_x = N$ //Number of vertices is increased by 1 and label //of the new vertex is N

/*To add the adjacency list of the new vertex, V_x in the graph*/

2. For $i = 1$ to l do
   1. Let $j = X[i]$ //j is the label of i-th adjacent vertex
   2. If ($j \geq N$) then //This label is not in graph
      1. Print "No vertex labelled $X[i]$ exist: Edge from V_x to $X[i]$ is not established".
   3. Else
      1. INSERT_SL_END (UGptr[N], $X[i]$) //Insert $X[i]$ into the list of vertices
      2. INSERT_SL_END (UGptr[j],V_x) //To establish edges from $X[i]$ to V_x
   4. EndIf
3. EndFor
4. Stop

Now, let us describe the algorithm INSERT_LL_DG to insert a vertex into a digraph.

Algorithm INSERT_VERTEX_LL_DG(V_x, X, Y)

Input: V_x, the new vertex that has to be inserted.

$X = [v_{x1}, v_{x2}, ..., v_{xm}]$, the list of adjacent vertices that has edges from V_x to v_{xi}, $i = 1, 2, ..., m$.

$Y = [v_{y1}, v_{y2}, ..., v_{yn}]$, the list of adjacent vertex that has edges from v_{yi} ($i = 1, 2, ..., n$) to V_x.

Let N be the number of vertices currently present in the graph.

Output: A graph with new vertex V_x and directed edges from V_x to v_{xi}, $i = 1, 2, ...$, and from v_{yi} ($i = 1, 2, ..., n$) to V_x, if such v_{xi} and v_{yi} exist.

Data structure: Linked structure of undirected graph and DGptr is the pointer to it.

Steps:

1. $N = N + 1$, $V_x = N$ //New vertex V_x is counted as next numbered in the graph

/*To set the edge from V_x to v_{xi}, $i = 1, 2, ..., m$

2. For $i = 1$ to m do
   1. Let $j = X[i]$ //j is the label of i-th adjacent vertex of V_x
   2. If $j \geq N$ then //j does not exist in the graph
      1. Print "No vertex labelled $X[i]$ does not exist: Edge from V_x to $X[i]$ is not established".
   3. Else //Set the edge
      1. INSERT_SL_END(DGptr[N], $X[i]$)
   4. EndIf
3. EndFor

/*To set edge from v_{yi}, $i = 1, 2, ..., n$ to V_x*/

4. For $i = 1$ to n do
   1. Let $j = Y[i]$ //j is the label of i-th adjacent vertex
   2. If $j \geq N$ then //j does not exist in the graph
      1. Print "No vertex labelled $Y[i]$ does not exist: Edge from $Y[i]$ to V_x is not established"
   3. Else //Set the edge
      1. INSERT_SL_END(DGptr[j],V_x)
   4. EndIf
5. EndFor
6. Stop

The shortest path from the array PATH can be obtained by backward movement. For example, for the vertex 5, its immediate predecessor is 3 (at PATH[5]), immediate predecessor of vertex 3 is 2 (at PATH[3]), immediate predecessor of vertex 2 is 1 (at PATH[2]), and vertex 1 is the source vertex. Thus, the shortest path for vertex 5 from the source vertex 1 is 1–2–3–5, and length of this shortest path is 5.

We have discussed about the single source shortest paths problem and its solution due to Dijkstra's algorithm. We have illustrated the algorithm with a weighted undirected graph. The algorithm is fairly applicable to solve the problem even if the graph is directed weighted, directed non-weighted, and non-weighted undirected, that is irrespective of the type of the graph. In case of non-weighted graph, weight of each edge should be taken as 1.

Assignment 8.15

1. Show how Dijkstra's algorithm works on each of the graph as shown in Figure 8.32. Source vertices are denoted by thick circles.

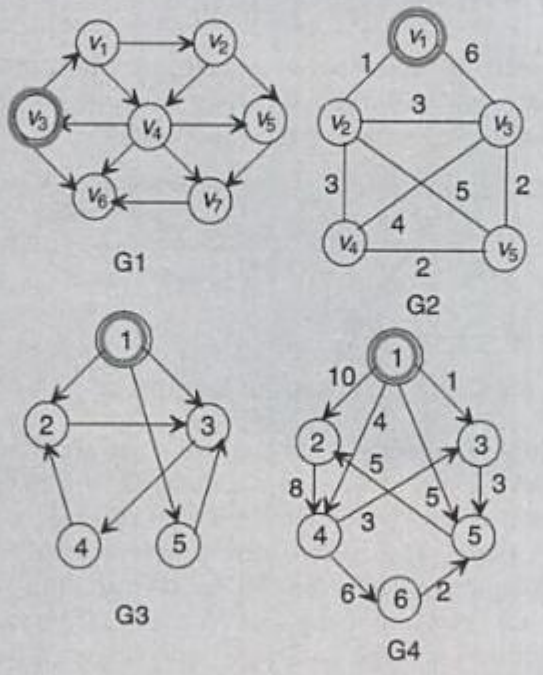


Fig. 8.32

2. There is another way to solve the single source shortest path problem. This approach is very similar to the breadth first search (BFS); the modification that is required is to use a priority queue instead of ordinary queue as used in BFS method. The priority of a vertex in the queue will be determined by the shortest path length of its predecessor plus the arc length (weight) from the predecessor to the vertex. Write an algorithm based on the above mentioned approach.
3. Figure 8.33 gives a road map connecting various places in a city and the cost of the tickets. Find the most economical route from place *X* to *Y*.

3. If found, let it be i. Include it as POSET[k] = i, $k = k + 1$. Set INCLUDE[i] = TRUE as i-th vertex is included in topological order. Delete the i-th column in the adjacency matrix so that entries of the i-th vertex will not be considered later on.
4. Repeat step 2-3 until POSET array is not full.

We will adopt the logical deletion of the i-th row and the i-column by making all the entries in them as –1. Now, we are in a position to describe the algorithm TOPOLOGICAL_SORT, as stated below:

Algorithm TOPOLOGICAL_SORT
Input: Gptr, the pointer to a graph. Let N be the number of vertices in the graph.
Output: POSET, the topological order of the vertices.
Data structure: INCLUDED[1 ... N], array of Boolean values.

Steps:

```
   /*Initialization*/
1. For i = 1 to N do
   1. POSET[i] = NULL
   2. INCLUDED[i] = FALSE
2. EndFor
3. k = 1                                  //Location of the next vertex in topological order
4. flag = TRUE                            //To control the repetition
5. While (flag) do                        //Repeat the steps
   1. For i = 1 to N do                   //Search for the left-most column
      1. If (not INCLUDED[i]) then        //If the i-th vertex is not already included
            /*Test for zero indegree of i*/
         1. zeroIndegree = TRUE
         2. For j = 1 to N do
            1. If (Gptr[i][j] > 0) Then
               1. zeroIndegree = FALSE
               2. Break                   //Quit the j-loop
            2. EndIf
         3. EndFor
            /*Set when indegree[i] = 0*/
         4. If (zeroIndegree) then
            1. INCLUDED[i] = TRUE         //Set the included field
            2. POSET[k] = i               //Include the vertex i into order
            3. k = k + 1
               /*Delete the i-th row and i-th column*/
            4. For j = 1 to N do
               1. Gptr[i][j] = -1         //Delete the i-th row
               2. Gptr[j][i] = -1         //Delete the i-th column
            5. EndFor
            6. Break                      //Quit the i-loop
         5. EndIf
      2. EndIf
   2. EndFor                              //i-loop
```

```
    /*If the vertex with zero indegree is no more found*/
  3. If (i = N) and (zeroIndegree = FALSE) then
     1. flag = FALSE                    //To stop the iteration
     2. Print "Graph is not acyclic"
     3. Return (NULL)
  4. EndIf
6. EndWhile
7. If (k = N) then                      //When the POSET array is full
   1. Return (POSET)
8. EndIf
9. Stop
```

It can be seen that the above described algorithm is applicable on both weighted and non-weighted directed acyclic graph.

Another easier implementation of the above algorithm is possible when the graph is represented with linked list. The implementation of that algorithm is left as an exercise for the reader.

Assignment 8.16 Following is a problem that highlights a simple application of topological sorting in practical field of applications.

Suppose, in the course curriculum of B.Tech. degree in computer science, the various courses with their prerequisite(s) are known and are listed as below:

Course	*Prerequisite(s)*
C31	C15, C25
C32	C21, C14
C33	C11, C24
C34	C22, C13
C35	C25
C21	C12
C22	C13
C23	C12, C13, C14
C24	C11
C25	C15
C11	Nil
C12	Nil
C13	Nil
C14	Nil
C15	Nil

Draw a graph pertaining to the above mentioned information. From the graph, obtain a topological sorting depicting which course can be registered after which course.

8.5.3 Minimum Spanning Trees

The next problem in the graph theory we are going to consider is *minimum spanning* tree

Algorithm UNION_LIST_SET(S_i, S_j; S)
Input: S_i and S_j are the headers of two single linked lists representing two distinct sets.
Output: S, the union of two sets S_i and S_j.
Data structure: Linked list representation of set.

Steps:

```
   /*To get a header node for S and initialize it*/
1. S = GETNODE(NODE)                          //Get a pointer to a node
2. S.LINK = NULL, S.DATA = NULL               //Initialization of the new node
   /*To copy the entire list of Si into S*/
3. ptri = Si.LINK
4. While (ptri ≠ NULL) do
   1. data = ptri.DATA                        //Copy from Si
   2. INSERT_SL_FRONT(S, data)                //Include it into the resultant set S
   3. ptri = ptri.LINK                        //Move to the next node in Si
5. EndWhile
6. ptrj = Sj.LINK                             //Pointer to the first node in Sj
   /*For each element in Sj, add it to S if it is not in Si*/
7. While (ptrj ≠ NULL) do
   1. ptri = Si.LINK                          //Search for the present element in Si
   2. While (ptri.DATA ≠ ptrj.DATA) do
      1. ptri = ptri.LINK
   3. EndWhile
   4. If (ptri = NULL) then                   //When the element is not in Si
      1. INSERT_SL_FRONT(S, ptrj.DATA)
   5. EndIf
   6. ptrj = ptrj.LINK                        //Move to the next element in Sj
8. EndWhile
9. Return(S)
10. Stop
```

In the above mentioned algorithm, we have assumed the method INSERT_SL_FRONT(S, KEY) to insert a node containing the data value KEY at the front of the linked list S. This procedure can be obtained from Chapter 3.

Intersection

Like the union operation, the intersection operation can be defined by searching a list for each element in the other list. Suppose, S_i and S_j are the two sets in the form of linked list. We are interested in finding $S \equiv S_i \cap S_j$, the intersection of S_i and S_j. Initially, S is empty. For each element in S_i, we have to search whether that element is in S_j or not. If that element is in S_j, we are to insert a node corresponding to that element in S.

In Figure 9.14, the intersection of two sets is illustrated. The detail of the procedure is described in the algorithm INTERSECTION_LIST_SET.

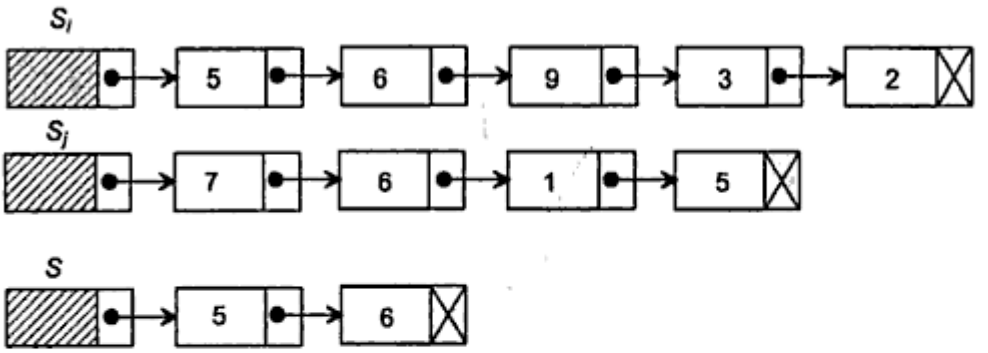


Fig. 9.14 Intersection of two sets in linked list form.

Algorithm INTERSECTION_LIST_SET(S_i, S_j; S)
Input: S_i and S_j are the headers of two single linked lists representing two sets.
Output: S, the intersection of two sets S_i and S_j.
Data structure: Linked list representation of set.

Steps:

/*To get a header node for S and initialize it*/
1. S = GETNODE(NODE)
2. S.LINK = NULL, S.DATA = NULL

/*Search the list S_j for each element in S_i*/
3. ptr$i$ = S_i.LINK
4. While (ptr$i$ ≠ NULL) do
   1. ptr$j$ = S_j.LINK
   2. While (ptr$j$.DATA ≠ ptr$i$.DATA) and (ptr$j$ ≠ NULL) do
      1. ptr$j$ = ptr$j$.LINK
   3. EndWhile
   4. If (ptr$j$ ≠ NULL) then //When the element is found in S_j
      1. INSERT_SL_FRONT(S, ptr$j$.DATA)
   5. EndIf
   6. ptr$i$ = S_i.LINK
5. EndWhile
6. Return(S)
7. Stop

As in the algorithm UNION_LIST_SET, we have used the procedure INSERT_SL_FRONT in this algorithm.

Difference

Suppose, two sets S_i and S_j in the form of linked lists are given and we are to find $S_i - S_j$, the difference of S_j from S_i. The difference is illustrated in Figure 9.15. A simple way to find the difference is to find $S_i \cap S_j$ first. Then we have to delete all the elements from S_i which is in $S_i \cap S_j$.

then next, we test whether an element in a set, is also present in the other set. The algorithm EQUALITY_LIST_SET is stated as below to realize it.

Algorithm EQUALITY_LIST_SET(S_i, S_j)
Input: S_i and S_j are the headers of the two sets in linked list representation.
Output: Returns TRUE if two sets S_i and S_j are equal else FALSE.
Data structure: Linked list representation of set.

Steps:

/*Count the number of elements in S_i and S_j*/
1. li = 0, lj = 0
2. ptr = S_i.LINK. //To count $|S_i|$
3. While (ptr ≠ NULL) do
   1. li = li + 1
   2. ptr = ptr.LINK
4. EndWhile
5. ptr = S_j.LINK //To count $|S_j|$
6. While (ptr ≠ NULL) do
   1. lj = lj + 1
   2. ptr = ptr.LINK
7. EndWhile
8. If (li ≠ lj) then
   1. flag = FALSE
   2. Exit //Quit the program
9. EndIf

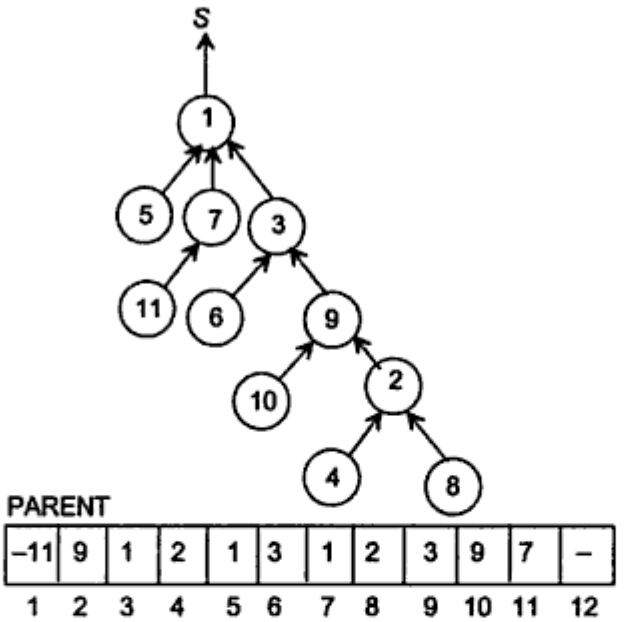
/*Compare the elements in S_i and S_j*/
10. ptr$i$ = S_i.LINK, flag = TRUE
11. While (ptril ≠ NULL) and (flag = TRUE) do
    1. ptr$j$ = S_j.LINK
    2. While (ptr$j$.DATA ≠ ptr$i$.DATA) and (ptr$j$ ≠ NULL) do
       1. ptr$j$ = ptr$j$.LINK
    3. EndWhile
    4. ptr$i$ = ptr$i$.LINK //Move to the next element in S_i
    5. If (ptr$j$ = NULL) then
       1. flag = FALSE
    6. EndIf
12. EndWhile
13. Return(flag)
14. Stop

Assignment 9.3

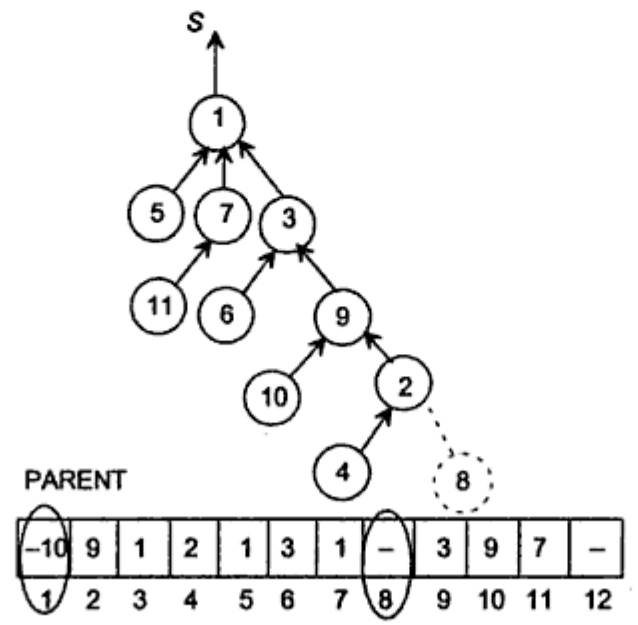
1. A set S_i will be termed as a subset of another set S_j iff for all $x \in S_i$, $x \in S_j$. Show how this criteria can be employed to decide whether a set is a subset of another set by rewriting the algorithm EQUALITY_LIST_SET.
2. Describe a procedure to decide whether two linked lists (single) representing two bags are equal or not.

Case 2: Exclusion of an element which is not a leaf node.

- Let the element to be excluded is *i*.
- Traverse the PARENT array to select an element *j* such that it is a successor of *i* and is a leaf node.



(a) Tree representation of a set *S*



(b) The set *S* after the exclusion of 8

Fig. 9.17 Continued.

```
    6. EndIf
    7. PARENT[X] = NULL                    //X is excluded
       /*Traverse the PARENT array to replace all the occurrence of X by item*/
    8. For i =1 to N do
       1. If (PARENT[i] = X) then
          1. PARENT[i] = item
       2. EndIf
    9. EndFor
 9. EndIf                                  //Case 2
10. Return(PARENT)                         //The reduced tree form of the set without X.
11. Stop
```

Assignment 9.6 Trace the algorithm EXCLUSION to exclude the following elements from the set whose tree representation is shown in Figure 9.17(a).

12
8
9
1

9.4 APPLICATIONS OF SETS

So far we have discussed different techniques of representation of sets and implementation of operations on them. We know, different application requires a particular representation of sets. In this section, we will discuss the use of sets in few applications.

1. Hash table representation of sets in spelling check.
2. Bit array representation of sets in information system.
3. Tree representation of sets in client-server environment.

9.4.1 Spelling Checker

As an application of hash table representation of sets, let us consider a very simple spelling checker system. Our spelling checker system is divided into three steps:

Step 1. The spelling checker system maintains a dictionary which is initially in a file. In this step, read the dictionary from the file into the hash table form of the set, say, DICTIONARY. Each element in the set is valid word according to the dictionary.

Step 2. The document to be spell checked is read word by word and the list of input word is gathered into another set in the form of hash table, say, DOCUMENT. Each element in this set is correct or incorrect word according to the DICTIONARY.

Step 3. An intersection operation on the two sets DICTIONARY, and DOCUMENT can be performed to obtain all the misspelled words in the documents, this is another hash table representation of set, and let it be MISSPELLED.

Classic Data Structures

D. Samanta

This text is designed for an introductory undergraduate course in data structures for computer science and engineering students. Written in a very accessible style, the book is also appropriate for students of polytechnics who will immensely benefit from its clear and concise analytic explanations presented in simple language. The book describes different types of classic data structures such as arrays, linked lists, stacks, queues, tables, trees, graphs and sets, providing a deep understanding of the essential concepts. To make the book both versatile and complete, the readers are exposed to the full range of design concepts featuring all the important aspects of each type of data structure. Among other distinguishing features, the book:

- Shows various ways of representing a data structure
- Examines different operations to manage a data structure
- Illustrates several applications of a data structure.

In this title, the algorithms are presented in English-like constructs instead of programming codes for ease of comprehension by students, without being bogged down in the details of a particular programming language. The algorithms described are nonetheless suitable for generic implementation, particularly with C++ and Java.

The book includes numerous exercises in the form of section-wise assignments and end-of-chapter problems to enable readers to build understanding of the concepts and acquire problem-solving skills.

D. SAMANTA, Ph.D is Assistant Professor, School of Information Technology, Indian Institute of Technology Kharagpur. He is the author of *Object-Oriented Programming with C++ and Java* (Prentice-Hall of India, 2000).

ISBN 81-203-1874-9

9 788120 318748

Rs. 225.00