

SECOND EDITION

Data Structures Using C

A. K. Sharma

ALWAYS LEARNING

PEARSON

Data Structures Using C

This page is intentionally left blank.

Data Structures Using C

Second Edition

A. K. Sharma

Professor and Dean

YMCA University of Science and Technology

PEARSON

Delhi • Chennai

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd.

Licensees of Pearson Education in South Asia

No part of this eBook may be used or reproduced in any manner whatsoever without the publisher's prior written consent.

This eBook may or may not include all assets that were part of the print version. The publisher reserves the right to remove any material in this eBook at any time.

ISBN 9788131792544

eISBN 9789332514225

Head Office: A-8(A), Sector 62, Knowledge Boulevard, 7th Floor, NOIDA 201 309, India

Registered Office: 11 Local Shopping Centre, Panchsheel Park, New Delhi 110 017, India

To
my parents,
wife Suman and daughter Sagun

This page is intentionally left blank.

Contents

<i>Preface to the Second Edition</i>	<i>xiii</i>
<i>Preface</i>	<i>xiv</i>
<i>About the Author</i>	<i>xv</i>
Chapter 1: Overview of C	1
1.1 The History	1
1.2 Characters Used in C	2
1.3 Data Types	2
1.3.1 Integer Data Type (<code>int</code>)	2
1.3.2 Character Data Type (<code>char</code>)	3
1.3.3 The Floating Point (<code>float</code>) Data Type	3
1.4 C Tokens	4
1.4.1 Identifiers	4
1.4.2 Keywords	5
1.4.3 Variables	5
1.4.4 Constants	7
1.5 Structure of a C Program	8
1.5.1 Our First Program	8
1.6 <code>printf()</code> and <code>scanf()</code> Functions	8
1.6.1 How to Display Data Using <code>printf()</code> Function	9
1.6.2 How to Read Data from Keyboard Using <code>scanf()</code>	10
1.7 Comments	10
1.8 Escape Sequence (Backslash Character Constants)	11
1.9 Operators and Expressions	13
1.9.1 Arithmetic Operators	13
1.9.2 Relational and Logical Operators	14
1.9.3 Conditional Operator	16
1.9.4 Order of Evaluation of Expressions	17
1.9.5 Some Special Operators	18
1.9.6 Assignment Operator	18
1.9.7 Bitwise Shift Operators	19
1.10 Flow of Control	20
1.10.1 The Compound Statement	21
1.10.2 Selective Execution (Conditional Statements)	21
1.10.3 Repetitive Execution (Iterative Statements)	25
1.10.4 The <code>exit()</code> Function	27
1.10.5 Nested Loops	28
1.10.6 The Goto Statement (Unconditional Branching)	28

1.11	Input–Output Functions (I/O)	30
1.11.1	Buffered I/O	31
1.11.2	Single Character Functions	32
1.11.3	String-based Functions	33
1.12	Arrays	34
1.13	Structures	34
1.13.1	Defining a Structure in C	35
1.13.2	Referencing Structure Elements	36
1.13.3	Arrays of Structures	36
1.13.4	Initializing Structures	37
1.13.5	Assignment of Complete Structures	37
1.13.6	Nested Structures	38
1.14	User-defined Data Types	39
1.14.1	Enumerated Data Types	40
1.15	Unions	42
1.16	Functions	43
1.16.1	Function Prototypes	44
1.16.2	Calling a Function	45
1.16.3	Parameter Passing in Functions	47
1.16.4	Returning Values from Functions	52
1.16.5	Passing Structures to Functions	52
1.17	Recursion	56
1.17.1	Types of Recursion	60
1.17.2	Tower of Hanoi	65

Chapter 2: Data Structures and Algorithms: An Introduction

72

2.1	Overview	72
2.2	Concept of Data Structures	73
2.2.1	Choice of Right Data Structures	74
2.2.2	Types of Data Structures	76
2.2.3	Basic Terminology Related with Data Structures	77
2.3	Design of a Suitable Algorithm	78
2.3.1	How to Develop an Algorithm?	78
2.3.2	Stepwise Refinement	80
2.3.3	Using Control Structures	81
2.4	Algorithm Analysis	85
2.4.1	Big-Oh Notation	86

Chapter 3: Arrays: Searching and Sorting

93

3.1	Introduction	93
3.2	One-dimensional Arrays	94
3.2.1	Traversal	95
3.2.2	Selection	96
3.2.3	Searching	98

3.2.4	Insertion and Deletion	105
3.2.5	Sorting	109
3.3	Multi-dimensional Arrays	130
3.4	Representation of Arrays in Physical Memory	134
3.4.1	Physical Address Computation of Elements of One-dimensional Arrays	135
3.4.2	Physical Address Computation of Elements of Two-dimensional Arrays	136
3.5	Applications of Arrays	138
3.5.1	Polynomial Representation and Operations	138
3.5.2	Sparse Matrix Representation	141

Chapter 4: Stacks and Queues **151**

4.1	Stacks	151
4.1.1	Stack Operations	152
4.2	Applications of Stacks	156
4.2.1	Arithmetic Expressions	156
4.3	Queues	170
4.3.1	Queue Operations	171
4.3.2	Circular Queue	176
4.3.3	Priority Queue	181
4.3.4	The Deque	185

Chapter 5: Pointers **197**

5.1	Introduction	197
5.1.1	The '&' Operator	197
5.1.2	The '*' Operator	198
5.2	Pointer Variables	198
5.2.1	Dangling Pointers	202
5.3	Pointers and Arrays	203
5.4	Array of Pointers	208
5.5	Pointers and Structures	208
5.6	Dynamic Allocation	210
5.6.1	Self Referential Structures	215

Chapter 6: Linked Lists **227**

6.1	Introduction	227
6.2	Linked Lists	227
6.3	Operations on Linked Lists	231
6.3.1	Creation of a Linked List	231
6.3.2	Travelling a Linked List	236
6.3.3	Searching a Linked List	241
6.3.4	Insertion in a Linked List	243
6.3.5	Deleting a Node from a Linked List	250

6.4	Variations of Linked Lists	253
6.4.1	Circular Linked Lists	254
6.4.2	Doubly Linked List	258
6.5	The Concept of Dummy Nodes	264
6.6	Linked Stacks	266
6.7	Linked Queues	270
6.8	Comparison of Sequential and Linked Storage	274
6.9	Solved Problems	274

Chapter 7: Trees 282

7.1	Introduction	282
7.2	Basic Terminology	284
7.3	Binary Trees	285
7.3.1	Properties of Binary Trees	286
7.4	Representation of a Binary Tree	287
7.4.1	Linear Representation of a Binary Tree	287
7.4.2	Linked Representation of a Binary Tree	289
7.4.3	Traversal of Binary Trees	291
7.5	Types of Binary Trees	298
7.5.1	Expression Tree	298
7.5.2	Binary Search Tree	303
7.5.3	Heap Trees	319
7.5.4	Threaded Binary Trees	340
7.6	Weighted Binary Trees and Huffman Algorithm	352
7.6.1	Huffman Algorithm	354
7.6.2	Huffman Codes	356
7.7	Dynamic Dictionary Coding	360

Chapter 8: Graphs 365

8.1	Introduction	365
8.2	Graph Terminology	366
8.3	Representation of Graphs	368
8.3.1	Array-based Representation of Graphs	368
8.3.2	Linked Representation of a Graph	371
8.3.3	Set Representation of Graphs	373
8.4	Operations of Graphs	373
8.4.1	Insertion Operation	374
8.4.2	Deletion Operation	379
8.4.3	Traversal of a Graph	384
8.4.4	Spanning Trees	396
8.4.5	Shortest Path Problem	401
8.5	Applications of Graphs	408

Chapter 9: Files 412

9.1	Data and Information	412
9.1.1	Data	412
9.1.2	Information	412
9.2	File Concepts	413
9.3	File Organization	415
9.4	Files in C	416
9.5	Files and Streams	416
9.6	Working with Files Using I/O Stream	418
9.6.1	Opening of a File	418
9.6.2	Unformatted File I/O Operations	419
9.6.3	Formatted File I/O Operations	425
9.6.4	Reading or Writing Blocks of Data in Files	426
9.7	Sequential File Organization	430
9.7.1	Creating a Sequential File	430
9.7.2	Reading and Searching a Sequential File	431
9.7.3	Appending a Sequential File	431
9.7.4	Updating a Sequential File	437
9.8	Direct File Organization	442
9.9	Indexed Sequential Organization	445
9.9.1	Searching a Record	445
9.9.2	Addition/Deletion of a Record	446
9.9.3	Storage Devices for Indexed Sequential Files	447
9.9.4	Multilevel Indexed Files	448
9.10	Choice of File Organization	448
9.11	Graded Problems	451

Chapter 10: Advanced Data Structures 459

10.1	AVL Trees	459
10.1.1	Searching an AVL Tree	461
10.1.2	Inserting a Node in an AVL Tree	462
10.2	Sets	468
10.2.1	Representation of Sets	469
10.2.2	Operations on Sets	470
10.2.3	Applications of Sets	476
10.3	Skip Lists	478
10.4	B-Trees	480
10.4.1	Searching a Key in a B-Tree	482
10.4.2	Inserting a Key in a B-Tree	483
10.4.3	Deleting a Key from a B-Tree	484
10.4.4	Advantages of B-Trees	487

10.5	Searching by Hashing	489
10.5.1	Types of Hashing Functions	490
10.5.2	Requirements for Hashing Algorithms	491
10.5.3	Overflow Management (Collision Handling)	491
<i>Appendix A</i>	<i>ASCII Codes (Character Sets)</i>	<i>494</i>
<i>Appendix B</i>	<i>Table of Format Specifiers</i>	<i>495</i>
<i>Appendix C</i>	<i>Escape Sequences</i>	<i>496</i>
<i>Appendix D</i>	<i>Trace of Huffman Algorithm</i>	<i>497</i>
<i>Index</i>	<i>501</i>	

Preface to the Second Edition

I have been encouraged by the excellent response given by the readers to the first edition of the book to work on the second edition. As per the feedback received from the teachers of the subject and the input provided by the team at Pearson Education, the following topics in various chapters of the book have been added:

1. Sparse matrices
2. Recursion
3. Hashing
4. Weighted binary trees
 - a. Huffman algorithm
5. Spanning trees, minimum cost spanning trees
 - a. Kruskal algorithm
 - b. Prims algorithm
6. Shortest path problems
 - a. Warshall's algorithm
 - b. Floyd's algorithm
 - c. Dijkstra's algorithm
7. Indexed file organization

While revising the book, the text has been thoroughly edited and the errors found thereof have been corrected. More examples on important topics have been included.

I hope the readers will like this revised edition of the book and, as before, will provide their much needed feedback and comments for further improvement.

Acknowledgements

I am thankful to Khushboo Jain and Anuradha Pillai for helping me in preparing the solution manual of the book.

A. K. Sharma

Preface

As a student, programmer, and teacher of computer engineering, I find ‘Data Structures’ a core course of computer engineering and particularly central to programming process.

In fact in our day-to-day life, we are confronted with situations such as where I would keep a bunch of keys, a pen, coins, two thousand rupees, a chalk, and five hundred thousand rupees.

I would keep the bunch of keys and coins in the left and right pockets of my pants, respectively. The pen gets clipped to the front pocket of the shirt whereas two thousand rupees would go into my ticket pocket. I would definitely put the five hundred thousand rupees into a safe, i.e., under the lock and key. While teaching, I will keep the chalk in hand. The decision of choosing the places for these items is based on two factors: *ease of accessibility* and *security*.

Similarly, given a problem situation, a mature programmer chooses the most appropriate data structures to organize and store data associated with the problem. The reason being that the intelligent choice of data structures will decide the fate of the software in terms of effectiveness, speed and efficiency—the three most important much-needed features for the success of a commercial venture.

I have taught ‘Data Structures’ for more than a decade and, therefore, the demand to write a book on this subject was there for quite some time by my students and teacher colleagues.

The hallmark of this book is that it would not only help students to understand the concepts governing the data structures but also to develop a talent in them to use the art of discrimination to choose the right data structures for a given problem situation. In order to provide a hands-on experience to budding software engineers, implementations of the operations defined on data structures using ‘C’ have been provided. The book has a balance between the fundamentals and advanced features, supported by solved examples.

This book would not have been possible without the well wishes and contribution of many people in terms of suggestions and useful remarks provided by them during its production. I record my thanks to Dr Ashutosh Dixit, Anuradha Pillai, Sandya Dixit, Dr Komal Bhatia, Rosy Bhatia, Harsh, and Indu Grover.

I am indebted to my teachers and research guides, Professor J. P. Gupta, Professor Padam Kumar, Professor Moinuddin, and Professor D. P. Agarwal, for their encouragement. I am also thankful to my friends, Professor Asok De, Professor Qasim Rafiq, Professor N.S. Gill, Rajiv Kapur and Professor Rajender Sahu, for their continuous support and useful comments.

I am also thankful to various teams at Pearson who made this beautiful book happen.

Finally, I would like to extend special thanks to my parents, wife Suman and daughter Sagun for saying ‘yes’ for this project when both wanted to say ‘no’. I know that I have stolen some of the quality time which I ought to have spent with them.

Some errors might have unwittingly crept in. I shall be grateful if they are brought to my notice. I would also be happy to acknowledge suggestions for further improvement of this book.

A. K. Sharma

About the Author



A. K. Sharma is currently Chairman, Department of Computer Engineering, and Dean of Faculty, Engineering and Technology at YMCA University of Science and Technology, Faridabad. He is also a member of the Board of Studies committee of Maharshi Dayanand University, Rohtak. He has guided ten Ph.D. theses and has published about 215 research papers in national and international journals of repute. He heads a group of researchers actively working on the design of different types of 'Crawlers'.

This page is intentionally left blank.

Overview of C

CHAPTER OUTLINE

- 1.1 The History
- 1.2 Characters Used in C
- 1.3 Data Types
- 1.4 C Tokens
- 1.5 Structure of a C Program
- 1.6 `printf()` and `scanf()` Functions
- 1.7 Comments
- 1.8 Escape Sequence (Backslash Character Constants)
- 1.9 Operators and Expressions
- 1.10 Flow of Control
- 1.11 Input–Output Functions (I/O)
- 1.12 Arrays
- 1.13 Structures
- 1.14 User-defined Data Types
- 1.15 Unions
- 1.16 Functions
- 1.17 Recursion

1.1 THE HISTORY

In 1971 Dennis Ritchi, a system programmer from Bell laboratories, developed a very powerful language called C for writing UNIX, a large and complex operating system. Even the compiler of 'C' was written in C. In fact, it is a high level language that not only supports the necessary data types and data structures needed by a normal programmer but it can also access the computer hardware through specially designed declarations and functions and, therefore, it is often called as a “middle-level” language.

It is popular because of the following characteristics:

- Small size
- Wide use of functions and function calls
- Loose typing
- Bitwise low level programming support
- Structured language
- Wide use of pointers to access data structures and physical memory of the system

Besides the above characteristics, the C programs are small and efficient. A 'C' program can be compiled on variety of computers.

1.2 CHARACTERS USED IN C

The set of characters allowed in C consists of alphabets, digits, and special characters as listed below:

- (i) Letters: Both upper case and lower case letters of English:

A, B, C, X, Y, and Z
a, b, c, x, y, and z

- (ii) Decimal digits:

0, 1, 2, 7, 8, 9

- (iii) Special characters:

! * + \ " < <
(= ! {
%) ; " /
^ _ [; , ?
& _] ' . blank

1.3 DATA TYPES

Every program specifies a set of operations to be done on some data in a particular sequence. However, the data can be of many types such as a numbers, characters, floating points, etc. C supports the following simple data types: Integer data type, character data type, floating point data type.

1.3.1 Integer Data Type (`int`)

An integer is an integral whole number without a decimal point. These numbers are used for counting. Examples of integers are:

923
47
5
15924
-56
2245

C allows four types of representation of integers, i.e., integer, long integer, short integer, and unsigned integer.

- **Integer:** An integer is referred to as `int`. It is stored in one word of the memory.
- **Long integer:** A long integer is referred to as `long int` or simple `long`. It is stored in 32 bits and does not depend upon the word size of the memory.
- **Short integer:** A short integer is referred to as `short int` or simple `short`. It is stored in 16 bits and does not depend upon the size of the memory.
- **Unsigned integers:** C supports two types of unsigned integers: the unsigned `int` and the unsigned `short`.

The unsigned `int` is stored in one word of the memory whereas the unsigned `short` is stored in 16 bits and does not depend upon the word size of the memory.

Examples of invalid integers are:

- (i) 9, 24, 173 illegal-comma used
- (ii) 5.29 illegal-decimal point used
- (iii) 79 248 blank used

1.3.2 Character Data Type (`char`)

It is a non-numeric data type consisting of single alphanumeric character enclosed between a pair of apostrophes, i.e., single quotation marks.

Examples of valid character type are:

```
'A'
'N'
'*'
'7'
```

It may be noted that the character '7' is different from the numeric value 7. In fact, former is of type `char` and later of type `int`. Each character has a numeric code, i.e., the ASCII code. For instance, the ASCII code for the character 'A' is 65 and that of '*' is 42. A table of ASCII codes is given in Appendix A.

A character data type is referred to as **char**. It is stored in one byte of memory. However, the representation varies from computer to computer. For instance, some computers support signed as well as unsigned characters. The signed characters can store integer values from -128 to $+127$ whereas the unsigned characters store ASCII codes from 0 to 255.

1.3.3 The Floating Point (`float`) Data Type

A floating point number is a real number which can be represented in two forms: decimal and exponent forms. Floating point numbers are generally used for measuring quantities.

- (1) **Decimal form:** The floating point number in this form has a decimal point. Even if it is an integral value, it must include the decimal point.

Examples of valid "decimal form" numbers are:

```
973.24
849.
73.0
-82349.24
9.0004
```

- (2) **Exponent form:** The exponent form of floating point number consists of the following parts:

`<integer> . <fraction> e <exponent>`

Examples of valid floating point numbers are:

```
3.45 e 7
0.249 e -6
```

It may be noted that exponent form is a scientific notation wherein the number is broken into two parts: *mantissa* and *exponent*. The mantissa is a floating point number of decimal form. The exponent part starts with a letter 'e' followed by an integer (signed or unsigned).

For example, the number 324.5 can be written as 3.245 times 10^2 . In exponential form, the number is represented as $3.245 e^2$. In fact, the base 10 has been replaced by the character e (or E).

The utility of exponential form is that very small numbers can be easily represented by this notation of floating points.

For example, the number 0.00001297 can be written as $0.1297 e^{-4}$ or as $12.97 e^{-6}$ or as $129.7 e^{-7}$.

C allows the following two data types for floating prints:

- *float*: These types of numbers are stored in 32 bits of memory.
- *double*: The numbers of double data type are stored in 64 bits of memory.

A summary of C basic data types is given in Table 1.1. From this table, it may be observed that character and integer type data can also be declared as unsigned. Such data types are called **unsigned data types**. In this representation, the data is always a positive number with range starting from 0 to a maximum value. Thus, a number twice as big as a signed number can be represented through unsigned data types.

■ **Table 1.1** Basic data types in C

C data type	Size	Lower bound	Upper bound	Represents
Char	1	–	–	Character
Unsigned Char	1	0	255	Character
Short (int)	2	–32768	+32767	Whole number
Unsigned Short int	2	0	65535	Whole number
Long int	4	2,147,438,648	2,141,438,647	Whole number
Float	4	$-3.4 \times 10^{\pm 38}$	$+3.4 \times 10^{\pm 38}$	Real number
Double	8	$-1.7 \times 10^{\pm 308}$	$+1.7 \times 10^{\pm 308}$	Real number

1.4 C TOKENS

A token is a group of characters that logically belong together. In fact, a programmer can write a program by using tokens. C supports the following types of tokens:

- Identifiers
- Keywords
- Constants
- Variables

1.4.1 Identifiers

Symbolic names can be used in C for various data items. For example, if a programmer desires to store a value 27, then he can choose any symbolic name (say, ROLL) and use it as given below:

```
ROLL = 27;
```

Where ROLL is a memory location and the symbol '=' is an assignment operator.

The significance of the above statement is that 'ROLL' is a symbolic name for a memory location where the value 27 is being stored. A symbolic name is generally known as an identifier.

The identifier is a sequence of characters taken from C character set. The number of characters in an identifier is not fixed though most of the C compilers allow 31 characters. The rules for the formation of an identifier are:

- An identifier can consist of *alphabets, digits* and *and/or underscores*.
- It must not start with a *digit*.
- C is case sensitive, i.e., upper case and lower case letters are considered different from each other.
- An identifier can start with an underscore character. Some special C names begin with the underscore.
- Special characters such as blank space, comma, semicolon, colon, period, slash, etc. are not allowed.
- The name of an identifier should be so chosen that its usage and meaning becomes clear. For example, total, salary, roll no, etc. are **self explanatory** identifiers.

Examples of acceptable identifiers are:

```
TOTAL
Sum
Net_sal
P123
a_b_c
total
_sysreg
```

Examples of unacceptable identifiers are:

Bas ic (blank not allowed)

H, rent (special character ` , ' included)

It may be noted here that TOTAL and total are two different identifier names.

1.4.2 Keywords

A keyword is a reserved word of C. This cannot be used as an identifier by the user in his program. The set of C keywords is given in Table 1.2.

■ **Table 1.2** Standard keywords in C

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

1.4.3 Variables

A variable is the most fundamental aspect of any computer language. It is a location in the computer memory which can store data and is given a symbolic name for easy reference. The variables can be used to hold different values at different times during a program run. To understand this concept, let us have a look at the following set of statements:

```
Total = 500.25;                                     ... (i)
Net = Total - 100.00;                                ... (ii)
```

In statement (i), value 500.25 has been stored in a memory location called Total. The variable Total is being used in statement (ii) for the calculation of another variable Net. The point worth noting is that *'the variable Total is used in statement (ii) by its name not by its value'*.

Before a variable is used in a program, it has to be defined. This activity enables the compiler to make available the appropriate amount of space and location in the memory. The definition of a variable consists of its type followed by the name of the variable. For example, a variable called Total of type float can be declared as shown below:

```
float Total;
```

Similarly, the variable net of type int can also be defined as shown below:

```
int Net;
```

Examples of valid variable declarations are:

```
(i) int count;
(ii) int i, j, k;
(iii) char ch, first;
```

- (iv) `float Total, Net;`
- (v) `long int sal;`
- (vi) `double salary.`

Let us now look at a variable declaration from a different perspective. Whenever a variable (say, `int val`) is declared, a memory location called `val` is made available by the compiler as shown in Figure 1.1.

Thus, `val` is the name associated by the compiler to a location in the memory of the computer. Let us assume that, at the time of execution, the physical address of this memory location (called `val`) is 4715 as shown in Figure 1.2.

Now, a point worth noting is that this memory location is viewed by the programmer as a variable called `val` and by the computer system as an address 4715. The programmer can store a value in this location with the help of an assignment operator, i.e., `'='`. For example, if the programmer desires to store a value 100 into the variable `val`, he can do so by the following statement:

```
val = 100;
```

Once the above statement is executed, the memory location called `val` gets the value 100 as shown in Figure 1.3.

C allows the initialization of variables even at the time of declaration as shown below:

```
int val = 100;
```

From Figure 1.3, we can see that besides its type a variable has three entities associated with it, i.e., the name of variable (`val`), its physical address (4715), and its contents (100). The content of a variable is also called its **rvalue** whereas the physical address of the variable is called its **lvalue**. Thus, `lvalue` and `rvalue` of variable `val` are 4715 and 100, respectively. The `lvalue` is of more importance because it is an expression that should appear on the left hand side of assignment operator because it refers to the variable or object.

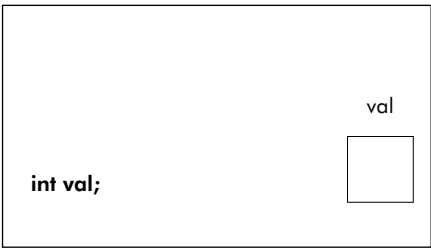


Fig. 1.1 Memory allocated to variable `val`

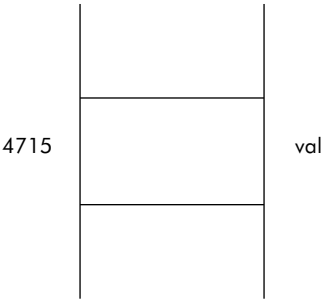


Fig. 1.2 The physical address of `val`

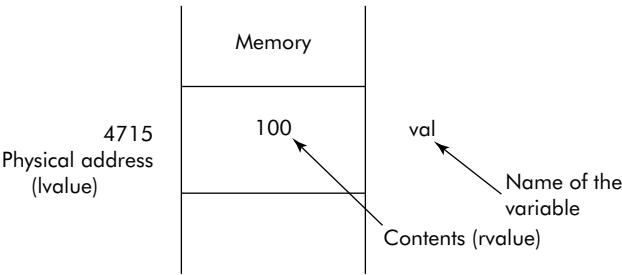


Fig. 1.3 The contents of a variable (`rvalue`)

1.4.4 Constants

A constant is a memory location which can store data in such a manner that its value during execution of a program does not change. Any attempt to change the value of a constant will result in an error message. A constant in C can be of any of the basic data types, i.e., integer constant, floating point constant, and character constant. **const** qualifier is used to declare a constant as shown below:

```
const <type> <name> = <val>;
```

where

const: is a reserved word of C
<type>: is any of the basic data types
<name>: is the identifier name
<val>: is the value to be assigned to the constant.

- (1) **Integer constant:** It is a constant which can be assigned integer values only. For example, if we desire to have a constant called *rate* of type integer containing a fixed value 50, then the following declaration can be used:

```
const int rate = 50;
```

The above declaration means that *rate* is a constant of type integer having a fixed value 50. Consider the following declaration:

```
const int rate;  
rate = 50;
```

The above initialization of constant *rate* is illegal. This is because of the reason that a constant cannot be initialized for a value at a place other than where it is declared. It may be further noted that if a program does not change or mutates a constant or constant object then the program is called as **const correct**.

- (2) **Floating point constant:** It is a constant which can be assigned values of real or floating point type. For example, if it is desired to have a constant called *Pi* containing value 3.1415, then the following declaration can be used.

```
const float Pi = 3.1415;
```

The above declaration means that *Pi* is a constant of type float having a fixed value 3.1415. It may be noted here that by default a floating point constant is of type double.

- (3) **Character constant:** A character constant can contain a single character of information. Thus, data such as 'Y' or 'N' is known as a character constant. Let us assume that it is desired to have a constant called *Akshar* containing the character 'Q'; following declaration can be used to obtain such a constant.

```
const char Akshar = 'Q';
```

The above declaration means that *Akshar* is a constant of type char having a fixed value 'Q'.

A sequence of characters enclosed within quotes is known as a *string literal*. For example, the character sequence "computer" is a string literal. When a string literal is assigned to an identifier declared as a constant, then it is known as a *string constant*. In fact, a string is an array of characters. Arrays are discussed later in the chapter.

1.5 STRUCTURE OF A C PROGRAM

A simple program in C can be written as a module. More complex programs can be broken into sub-modules. In C, all modules or subprograms are referred to as functions.

The structure of a typical C program is given below:

```
main ()
{
    ....;
    ....;
    ....;
}
```

Note:

- (i) C is a case sensitive language, i.e., it distinguishes between upper case and lower case characters. Thus, `main()` is different from `Main()`. In fact, most of the characters used in C are lowercase. Hence, it is safest to type everything in lower case except when a programmer needs to capitalize some text.
- (ii) Every C program has a function called `main` followed by parentheses. It is from here that program execution begins. A function is basically a subprogram and is complete in itself.
- (iii) The task to be performed by a function is enclosed in curly braces called its body, i.e., `{ }`. These braces are equivalent to *begin* and *end* keywords used in some other languages like Pascal. The body of function contains a set of statements and each statement must end with a semicolon.

1.5.1 Our First Program

Let us break the myth that C is a difficult language to start with. Without going into the details, the beginner can quickly start writing the program such as given below:

```
#include <stdio.h>
main()
{
    puts ("This is my first program");
}
```

The above program displays the following text on the screen:

This is my first program.

1.6 printf() AND scanf() FUNCTIONS

C uses `printf()` and `scanf()` functions to write and read from I/O devices, respectively. These functions have been declared in the header file called `stdio.h`.

Let us use `printf()` function to rewrite our first program. The modified program is given below:

```
#include <stdio.h>
main()
{
```

```
    printf ("This is my first program");  
}
```

This program displays the following message on the screen:

This is my first program.

In fact, any text written within the pair of quotes (") is displayed as such by `printf()` function on the screen. However, this function is much more powerful than `puts()` in the sense that it can display all types of data on the screen as explained below.

1.6.1 How to Display Data Using `printf()` Function

An integer, stored in a variable, can be displayed on the screen by including a format specifier (`%d`) within a pair of quotes as shown below:

```
#include <stdio.h>  
main()  
{  
    int age = 25;  
    printf ("%d", age);  
}
```

The above program does the following:

- (1) Declares a variable age of type `int`.
- (2) Initializes age to 25, an integer value.
- (3) Specifies `%d`, a format specifier indicating that an integer is to be displayed by `printf()` and the data is stored in the variable called age.

It may be noted that `printf()` has two arguments separated by a comma, i.e., format specifier written within quotes and the variable age. It may be further noted that both messages and format specifiers can be included within the pair of quotes as shown below:

```
printf (" The age of student = %d", age);
```

The above statement would display the following data on the screen:

The age of student = 25

Thus, the text between the quotes has been displayed as such but for `%d`, the data stored in the variable age has been displayed.

A float can be displayed on the screen by including a format specifier (`%f`) within the pair of quotes as shown below:

```
float rate = 9.5;  
printf ("The rate of provident fund = %f", rate);
```

The above statements would produce the following display:

The rate of provident fund = 9.5

A char can be displayed by including a format specifier (`%c`) within the pair of quotes as shown below:

```
char ch = 'A'  
printf ("The first alphabet is: %c", ch);
```

Obviously, the output of the above set of statements would be:

The first alphabet is: A.

The other specifiers which can be used in `printf()` function are given in Appendix B.

1.6.2 How to Read Data from Keyboard Using `scanf()`

Similar to `printf()` function, `scanf()` function also uses the `%d`, `%f`, `%c`, and other format specifiers to specify the kind of data to be read from the keyboard.

However, it expects the programmer to prefix the address operator `&` to every variable written in the argument list as shown below:

```
int roll;
scanf ("%d", & roll);
```

The above set of statements declares the variable `roll` of type `int`, and reads the value for `roll` from the keyboard. It may be noted that `&`, the address operator, is prefixed to the variable `roll`. The reason for specifying `&` would be discussed later in the book.

Similarly, other format specifiers can be used to input data from the keyboard.

Example 1: Write an interactive program that reads the marks secured by a student for four subjects `Sub1`, `Sub2`, `Sub3`, and `Sub4`, the maximum marks of a subject being 100. The program shall compute the percentage of marks obtained by the student.

Solution: The `scanf()` and `printf()` functions would be used to do the required task. The required program is as follows:

```
#include <stdio.h>
main()
{
    int sub1, sub2, sub3, sub4;
    float percent;
    printf ("Enter the marks for four subjects:");
    scanf ("%d %d %d %d",&sub1, &sub2, &sub3, &sub4);
    percent = (sub1 + sub2 + sub3 + sub4)/ 400.00*100;
    printf ("The percentage = %f", percent);
}
```



For input data 45 56 76 90, the above program computes the percentage and displays the following output:

The percentage = 66.750000

Though the output is correct, but it has displayed unnecessary four trailing zeros. This can be controlled by using field width specifiers discussed later in the chapter.

1.7 COMMENTS

A comment can be added to the program by enclosing the text between the pair `/ * ... * /`, i.e., the pair `/*` indicates the beginning of the comment whereas the pair `*/` marks the end of it. It is also known as multiple line comment. For example, the following line is a comment:

```
/* This is my first program */
```

Everything written within `/*` and `*/` is ignored by the compiler. A comment written in this fashion can overflow to multiple lines as shown below:

```
/* This is an illustration of multiple line comment. The C compiler ignores
these lines. A programmer can use comment lines for documentation of his
programs*/
```

In fact, *a comment is a non-executable statement* in the program.

1.8 ESCAPE SEQUENCE (BACKSLASH CHARACTER CONSTANTS)

C has some special character constants called *backslash* character constants. These are unprintable ASCII characters which can perform special functions in the output statements. A backslash character constant is nothing but a back slash (`\`) character followed by another character.

For example, `"\n"` can be used in a `printf()` statement to send the next output to the beginning of the next line.

Consider the following program:

```
#include <stdio.h>
main()
{
    printf("A backslash character constant");
    printf("prints the output on the next line.");
}
```

The output of this program would be:

A backslash character constant prints the output on the next line.

It may be noted that though two separate `printf()` statements were written in the above program, the output has been displayed on the same line.

In order to display the text in two lines, the `"\n"` character constant should be placed either at the end of the text in the first `printf()` statement or at the beginning of the text in the second `printf()` statement.

Consider the following modified program:

```
#include <stdio.h>
main()
{
    printf("A backslash character constant");
    printf("\n prints the output on the next line.");
}
```

The character constant `"\n"` has been placed at the beginning of the text of the second `printf()` statement and, therefore, the output of the program would be:

**A back slash character constant
prints the output on the next line.**

Some other important backslash constants are listed below:

Backslash character constant	Meaning
<code>\t</code>	tab
<code>\b</code>	backspace
<code>\a</code>	bell (alert)
<code>\n</code>	newline character

Note:

The backslash characters are also known as escape sequences; such characters can be used within apostrophes or within double quotes.

- (1) '**\t**' (**tab**): This character is called a tab character. Whenever it is inserted in an output statement, the display moves over to the next present tab stop. Generally, a tab is of 8 blank spaces on the screen. An example of usage of character '**\t**' is given below:

```
#include <stdio.h>
main()
{
    printf("\n hello \t Comp-Boy");
}
```

The output of the above statement would be on the new line with spacing as shown below:

hello Comp-Boy

- (2) '**\b**' (**Backspace**): This character is also called backspace character. It is equivalent to the backspace key symbol (←) available on the computer or typewriter. It moves one column backward and positions the cursor on the character displayed on that column. An example of usage of character '**\b**' is given below:

```
#include <<stdio.h
main()
{
    printf("\n ASHOKA\b-");
}
```

The output of the above statement would be:

ASHOK-

We can see that the trailing letter 'A' has been overwritten by the character '-' at the end of the string constant <ASHOKA> the reason being that '**\b**' moved the cursor one column backward, i.e., at 'A' and `printf()` printed the character '-' on that column position.

- (3) '**\a**' (**Alert**): This character is also called alert or bell character. Whenever it is inserted in an output statement, it sounds a bell which can be heard by the user sitting on the computer terminal. It can be used in a situation where the programmer wants to catch the attention of the user of this program. An example of usage of '**\a**' character is given below:

```
#include <stdio.h>
main()
{
    printf("\n Error in data \a");
}
```

The output of the above statement would be the following message on the screen and a sounding of a bell on the system speaker.

Error in data

- (4) '**\n**' (**new line**): As discussed earlier, this character is called newline character. Wherever it appears in the output statement, the immediate next output is taken to the beginning of the next new line on the screen. Consider the statement given below:

```
#include <stdio.h>
main()
{
    printf("\n This is \n a test.");
}
```

The output of this statement would be:

```
This is
a test.
```

1.9 OPERATORS AND EXPRESSIONS

An **operator** is a symbol or letter used to specify a certain operation on variables in a program. For example, the symbol '+' is an add operator that adds two data items called operands.

Expressions: An expression is a combination of operands (i.e., constants, variables, numbers) connected by operators and parenthesis. For example, in the expression given below A and B are operands and '+' is an operator.

A + B

C supports many types of operators such as arithmetic, relational, logical, etc. An expression that involves arithmetic operators is known as an arithmetic expression. The computed result of an arithmetic expression is always a numerical value. The expression which involves relational and/or logical operators is called as a boolean expression or logical expression. The computed result of such an expression is a logical value, i.e., either 1 (True) or 0 (False).

The rules of formation of an expression are:

- A signed or unsigned constant or variable is an expression.
- An expression connected by an operator to a variable or a constant is an expression.
- Two expressions connected by an operator is also an expression.
- Two operators should not occur in continuation.

1.9.1 Arithmetic Operators

The valid arithmetic operators supported by C are given in Table 1.3.

1.9.1.1 Unary Arithmetic Operators A unary operator requires only one operand or data item. The unary arithmetic operators supported by C are unary minus ('-'), increment ('++'), and decrement ('--'). As compared to binary operators, the unary operators are right associative in the sense that they evaluate from right to left.

■ **Table 1.3** Arithmetic operators

Symbol	Stands for	Example
+	addition	$x + y$
-	subtraction	$x - y$
*	multiplication	$x * y$
/	division	x/y
%	modulus or remainder	$x\%y$
--	decrement	$--x$ $x--$
++	increment	$x++$

The unary minus operator is written before a numerical value, variable or an expression. Examples of usage of unary minus operator are:

(i) -57 (ii) -2.923 (iii) $++x$ (iv) $--(a*b)$ (v) $8*(-(a+b))$

It may be noted here that the result of application of unary minus on an operand is the negation of its operand.

The operators '++' and '--' are unique to C. These are called increment and decrement operators, respectively. The increment operator '++' adds 1 to its operand. Therefore, we can say that the following expressions are equivalent.

$i = i + 1 \equiv ++i;$

For example, if the initial value of i is 10 then the expression $++i$ will increment the contents of i to 11. Similarly, the decrement operator '--' subtracts 1 from its operand. Therefore, we can say that the following expressions are equivalent:

$j = j - 1 \equiv --j;$

For example, if the initial value of j is 5 then the expression $--j$ will decrement the contents of j to 4.

The increment and decrement operators can be used both as a prefix and as a postfix to a variable as shown below:

$++x$ or $x++$
 $--y$ or $y--$

As long as the increment or decrement operator is not used as part of an expression, the prefix and postfix forms of these operators do not make any difference. For example, $++x$ and $x++$ would produce the same result. However, if such an operator is part of an expression then the prefix and postfix forms would produce entirely different results.

In the prefix form, the operand is incremented or decremented before the operand is used in the program. Whereas in the postfix form, the operand is used first and then incremented or decremented.

1.9.2 Relational and Logical Operators

A relational operator is used to compare two values and the result of such an operation is always logical, i.e., either true or false. The valid relational operators supported by C are given in Table 1.4.

■ **Table 1.4** Relational operators supported by C

Symbol	Stands for	Example
>	greater than	$x > y$
>=	greater than equal to	$x \geq y$
<	less than	$x < y$
<=	less than equal to	$x \leq y$
==	equal to	$x == y$
!=	not equal to	$x != y$

Example 2: Take two variables x and y with initial values 10 and 15, respectively. Demonstrate the usage of relational operators by using x and y as operands.

Solution: The usage of relational operators is illustrated below with the help of a table:

$x = 10, y = 15$



Expression	Result
$x > y$	False
$x + 5 \geq y$	True
$x < y$	True
$x \leq y$	True
$x == y$	False
$x + 5 == y$	True
$x != y$	True

A logical operator is used to connect two relational expressions or logical expressions. The result of such an operation is always logical, i.e., either true or false. The valid logical operators supported by C are given in Table 1.5.

■ **Table 1.5** Logical operators supported by C

Symbol	Stands for	Example
&&	Logical AND	$x \&\& y$
	Logical OR	$x y$
!	Logical NOT	$!x$

Rules of logical operators:

- (1) The output of a logical AND operation is true if both of its operands are true. For all other combinations, the result is false.
- (2) The output of logical OR operation is false if both of its operands are false. For all other combinations, the result is true.
- (3) The logical NOT is a unary operator. It negates the value of the operand.

For initial value of $x = 5$ and $y = 7$, consider the following expression:

$(x < 6) \ \&\& \ (y > 6)$

The operand $x < 6$ is true and the operand $y > 6$ is also true. Thus, the result of above given logical expression is also true. However, the result of following expression is false because one of the operands is false:

$(x < 6) \ \&\& \ (y > 7)$

Similarly, consider the following expression:

$(x < 6) \ || \ (y > 7)$

The operand $x < 6$ is true whereas the operand $y > 7$ is false. Since these operands are connected by logical OR, the result of this expression is true (Rule 2). However, the result of the following expression becomes false because both the operands are false.

$!(x < 6) \ || \ (y > 7)$

Note: The expression on the right hand side of logical operators $\&\&$ and $||$ does not get evaluated in case the left hand side determines the outcome.

Consider the expression given below:

$x \ \&\& \ y$

If x evaluates to false (zero), then the outcome of the above expression is bound to be false irrespective of y evaluating to any logical value. Therefore, there is no need to evaluate the term y in the above expression.

Similarly, in the following expression, if x evaluates to true (non zero) then the outcome is bound to be true. Thus, y will not be evaluated.

$x \ || \ y$

1.9.3 Conditional Operator

C provides a conditional operator ($? :$) which can help the programmers in performing simple conditional operations. It is represented by the symbols ' $?$ ' and ' $:$ '. For example, if one desires to assign the bigger of the two variables x and y to a third variable z the conditional operator is an excellent tool.

The general form of this operation is:

$E1?E2:E3$

where $E1$, $E2$, and $E3$ are expressions.

In the conditional operation, the expression $E1$ is tested, if $E1$ is true then $E2$ is evaluated otherwise the expression $E3$ is evaluated as shown in Figure 1.4.

Consider the following conditional expression:

$z = (x > y)? x : y;$

The expression $x > y$ is evaluated. If x is greater than y , then z is assigned x otherwise z gets y . Examples of valid conditional expressions are:

(i)

$y = (x >= 10)?0:10;$

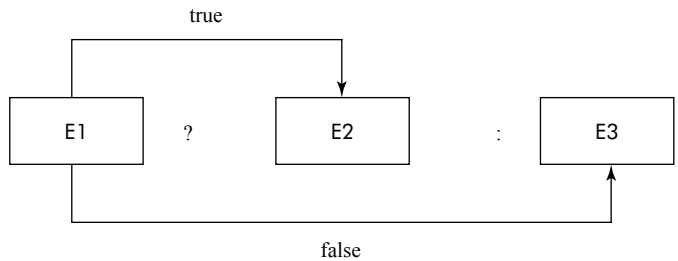


Fig. 1.4 The conditional operator

(ii)

Res = (i < j)? sum + i : sum + j;

(iii)

q = (a == 0)?0:(x/y);

It may be noted here that the conditional operator (? :) is also known as a ternary operator because it operates on three values.

1.9.4 Order of Evaluation of Expressions

A number of logical and relational expressions can be linked together with the help of logical operators as shown below:

(x < y) || (x > 20) && !(z) || ((x < y) && (z > 5))

For a complex expression such as given above, it becomes difficult to make out as to in what order the evaluation of sub-expressions would take place.

In C, the order of evaluation of an expression is carried out according to the operator precedence given in Table 1.6.

Table 1.6 Operator precedence

Operators	Priority	Associativity
- ++ -- !	Highest	Right to left
* / %	<div style="text-align: center;"> ↓ ↓ ↓ ↓ ↓ ↓ </div>	Left to right
+ -		Left to right
< <= > >=		Left to right
== !=		Left to right
&&		Left to right
		Left to right
	Lowest	

It may be noted here that in C, false is represented as zero and true as any non-zero value. Thus, expressions that use relational and logical operators return either 0 (false) or 1 (true).

1.9.5 Some Special Operators

There are many other operators in C. In this section, the two frequently used operators—sizeof and comma operators—have been discussed.

1. **Sizeof operator:** C provides a compile time unary operator called sizeof which, when applied on an operand, returns the number of bytes the operand occupies in the main memory. The operand could be a variable, a constant or a data type. For example, the following expressions:

```
a = sizeof ("sum");  
b = sizeof (char);  
c = sizeof (123L);
```

would return the sizes occupied by variables sum, data type char and constant '123L' on your machine. It may be noted that

- the parentheses used with sizeof are required when the operand is a data type with variables or constants, the parentheses are not necessary.
 - sizeof operator has the same precedence as prefix increment/decrement operators.
2. **Comma operator:** The comma operator is used to string together a number of expressions which are performed in a sequence from left to right.

For example, the following statement

```
a = (x = 5, x + 2)
```

executes in the following order

- (i) value 5 is assigned to variable x.
- (ii) x is incremented by 2.
- (iii) the value of expression $x + 2$ (i.e., 7) is assigned to the variable a.

The following points may be noted regarding comma operator:

- A list of expressions separated by a comma is always evaluated from left to right.
- The final value and type of a list of expressions separated by a comma is always same as the type and value of the rightmost expression in the list.
- The comma operator has the lowest precedence among all C operator.

1.9.6 Assignment Operator

Statements are the smallest executable units of a C program and each statement is terminated with a semicolon. An assignment statement assigns the value of the expression on the right hand side to a variable on the left hand side of the assignment operator (=). Its general form is given below:

```
<variable name> = <expression>
```

The expression on the right hand side could be a constant, a variable or an arithmetic, relational, or logical expression. Some examples of assignment statements are given below:

```
a = 10;  
a = b;  
a = b*c;
```

- (i) The assignment operator is a kind of a store statement, i.e., the value of the expression on the right hand side is stored in the variable appearing on the left side of the assignment operator.

The variable on the left side of the assignment operator is also called **lvalue** and is an accessible address in the memory. Expressions and constants on the right side of the assignment operator are called **rvalues**.

- (ii) The assignment statement overwrites the original value contained in the variable on the left hand side with the new value of the right hand side.
- (iii) Also, the same variable name can appear on both sides of the assignment operator as shown below:

```
count = count + 1;
```

- (iv) Multiple assignments in a single statement can be used, especially when same value is to be assigned to a number of variables.

```
a = b = c = 30;
```

These multiple assignment statements work from right to left and at the end, all variables have the same value. The above statement assigns the value (i.e., 30) to all variables c, b, and a. However, the variables must be of same type.

- (v) A point worth nothing is that C converts the type of value on the right hand side to the data type on the left.

1.9.7 Bitwise Shift Operators

C supports special bitwise shift operators: shift left (<<) and shift right (>>). We know that data at machine level is represented in the binary form and it can be shifted to left or right by these special operators. However, the number of bits to be shifted is specified by an integer written next to the operators as shown below:

```
x << 3;
```

The above statement means that the value contained in x is shifted to left by 3 bits. Thus, the leftmost 3 bits of x will be lost and the resultant vacant 3 rightmost bits will be filled by zeros as shown in Figure 1.5.

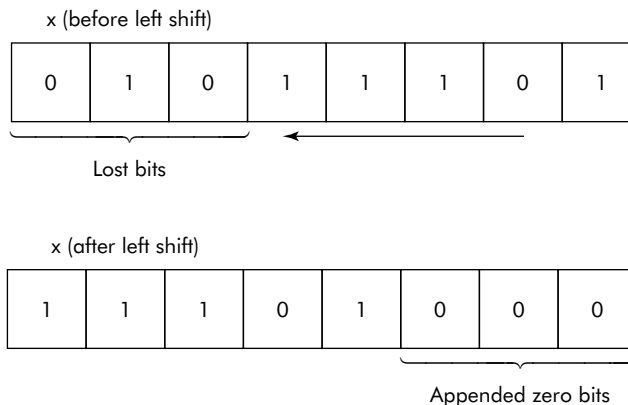


Fig. 1.5 The contents of x before and after left shift

Similarly, the following expression would shift the contents of *y* to right by two bits:

```
y >> 2
```

In this case, the rightmost two bits would be lost and the resultant 2 vacant leftmost bits would be filled by zeros as shown in Figure 1.6.

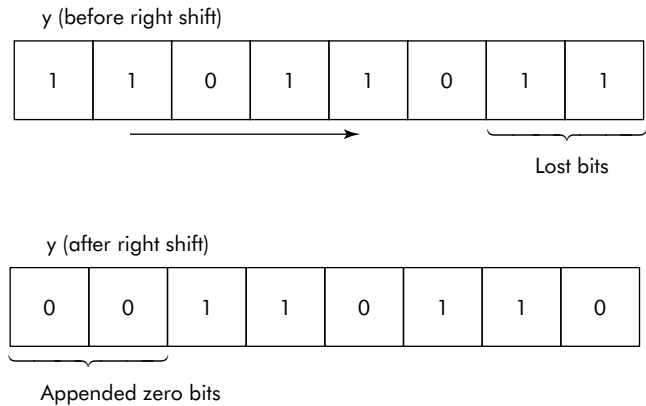


Fig. 1.6 The contents of *y* before and after right shift

1.10 FLOW OF CONTROL

A statement is the smallest executable unit of a C program. It is terminated with a semicolon. It is an instruction given to the computer to perform a particular task like reading input, displaying output or evaluating an expression, etc.

A single statement is also called a simple statement. Some examples of simple statement are given below:

- (i) `int a = 100;`
- (ii) `S = S + a;`
- (iii) `count ++;`

A statement can also be an empty or null statement as shown below:

```
;
```

The high level languages are designed for computers based on Von-Neumann architecture. Since this architecture supports only sequential processing, the normal flow of execution of statements in a high level language program is also sequential, i.e., each statement is executed in the order of its appearance in the program. For example in the following C program segment, the order of execution is sequential from top to bottom.

```
:
x = 10;
y = 20;
z = x + y;
:
```

Order of execution

The first statement to be executed is 'x = 10', and the second statement to be executed is 'y = 20'. The execution of statement 'z = x + y' will take place only after the execution of the statement 'y = 20'. Thus, the processing is strictly sequential. Moreover, every statement is executed once and only once.

Depending upon the requirements of a problem, it is often required to alter the normal sequence of execution in the program. This means that we may desire to *selectively* and/or *repetitively* execute a program segment. A number of C control structures are available for controlling the flow of processing. These structures are discussed in the following sections.

1.10.1 The Compound Statement

A compound statement is a group of statements separated from each other by a semicolon. The group of statements, also called a block of code, is enclosed between a pair of curly braces, i.e., '{ ' and ' } '. The significance of a block is that the sequence of statements enclosed in it is treated as a single unit. For example, the following group of statements is a block.

```
/* a block of code*/
{
    scanf ("%d %d", & a, & b);
    c = a + b;
    printf ("\n %d", c);
}
```

One compound statement can be embedded in another as shown below:

```
{
    :
    {
        :
    }
    :
}
```

In fact, the function of curly braces '{ ' and ' } ' in a C program to create a block, is same as the function of *begin* and *end*, the reserved words in Pascal. C calls these braces as *delimiters*.

1.10.2 Selective Execution (Conditional Statements)

In some cases, it is desired that a selected segment of a program be executed on the basis of a test, i.e., depending upon the state of a particular condition being true or false. In C, 'if statement' is used for selective execution of a program segment.

- (1) **The if statement:** This statement helps us in the selection of one out of two alternative courses of action. The general form of 'if statement' is given below:

```
if (expression)
{
    statement sequence
}
```

where **if:** is a reserved word.

expression: is a boolean expression enclosed within a set of parentheses (These parentheses are necessary even if there is a single variable in the expression).
statement sequence: is either a simple statement or a block. However, it cannot be a declaration.

Examples of acceptable 'if statements' are:

```
(i) if (A > B) A = B;
(ii) if (total < 100) {
        total = total + val;
        count = count + 1;
    }
(iii) if ((Net > 7000) && (l_text == 500)) {
        :
    }
```

- (2) **The if-else statement:** It can be observed from the above examples that the simple if statement does nothing when the expression is false. An **if-else** statement takes care of this aspect. The general form of this construct is given below:

```
if (expression)
{
    statement sequence1
}
else
{
    statement sequence2
}
```

where **if:** is a reserved word.
 expression: is a boolean expression, written within parentheses.
statement sequence1: can be a simple or a compound statement.
 else: is a reserved word.
statement sequence2: can be a simple or a compound statement.

Examples of if-else statements are:

```
(i) if (A > B) C = A;
    else C = B;
(ii) if (x == 100)
    printf ("\n Equal to 100");
    else
    printf ("\n Not Equal to 100");
```

It may be noted here that both the 'if' and 'else' parts are terminated by semicolons.

- (3) **Nested if statements (if-else-if ladder):** The statement sequence of if or else may contain another if statement, i.e., the if-else statements can be nested within one another as shown below:

```
if (exp1)
    if (exp2)
    {
```

```

        :
    }
else
    if (exp3)
    {
        :
    }
else
    {
        :
    }

```

It may be noted here that sometimes the nesting may become complex in the sense that it becomes difficult to decide “which if does the else match”. This is called “dangling else problem”. The C compiler follows the following rule in this regard:

Rule: Each else matches to its nearest unmatched preceding if.

Consider the following nested if:

```
if (x < 50) if (y > 5) Net = x + y; else Net = x - y;
```

In the above statement, the else part matches the second if (i.e., if ($y > 5$)) because it is the nearest preceding if. It is suggested that the nested `if (s)` should be written with proper indentation. The else(s) should be lined up with their matching `if (s)`. Nested `if (s)` written in this fashion are also called if-else-if ladder. For example, the nested if given above should be written as:

```

if (x < 50)
    if (y > 5)
        Net = x + y;
    else
        Net = x - y;

```

However, if one desires to match the else with the first if, then the braces should be used as shown below:

```

if (x < 50) {
    if (y > 5)
        Net = x + y;
}
else
    Net = x - y;

```

The evaluation of if-else-if ladder is carried out from top to bottom. Each conditional expression is tested and if found true, only then its corresponding statement is executed. The remaining ladder is, therefore, by passed. In a situation where none of the nested conditions is found true, the final else part is executed.

- (4) **Switch statement (selection of one of many alternatives):** If it is required in a program to select one of several different courses of action, then the switch statement of C can be used. In fact, it is a multibranch selection statement that makes the control to jump to one of the several statements based on the value of an int variable or expression. The general form of this statement is given as:


```

switch (expression)
{
    case constant 1:
        statement;
        break;
    case constant 2:
        statement;
        break;
    :
    default:
        statement;
}

```

where switch: is a reserved word.

expression: it must evaluate to an integer or character value.

case: is a reserved word.

constant: it must be an int or char compatible value.

statement: a simple or compound statement.

default: is a reserved word and is an optional entry.

break: is a reserved word that stops the execution within the switch and control comes out of the switch construct.

The switch statement works according to the following rules:

- The value of the expression is matched with the random case constants of the switch construct.
- If a match is found then its corresponding statements are executed and when break is encountered, the flow of control jumps out of the switch statement. If break statement is not encountered, then the control continues across other statement. In fact, switch is the only statement in C which is error prone. The reason is that if the control is in a particular case and then it keeps running through all cases in the absence of a proper break statement. This phenomenon is called “**fall-through**”.
- If no match is found and if a default label is present, then the statement corresponding to default is executed.
- The values of the various case constants must be unique.
- There can be only one default statement in a switch statement.

Examples of acceptable switch statements are:

(i)

```

switch (BP)
{
    case 1 : total + = 100;
        break;
    case 2 : total + = 150;
        break;
    case 3 : total + = 250;
}

```

(ii)

```

switch (code)

```

```
{
    case 101 : Rate = 50; break;
    case 102 : Rate = 70; break;
    case 103 : Rate = 100; break;
    default  : Rate = 95;
}
```

In the statement (ii), it can be observed that depending on the value of the code one out of the four instructions is selected and obeyed. For example, if the code evaluates to 103, the third instruction (i.e., Rate = 100) is selected. Similarly if the code evaluates to 101, then the first instruction (i.e., Rate = 50) is selected.

1.10.3 Repetitive Execution (Iterative Statements)

Some problems require a set of statements to be executed a number of times, each time changing the values of one or more variables so that every new execution is different from the previous one. This kind of repetitive execution of a set of statements in a program is known as a loop.

We can categorize loop structures into two categories: **non-deterministic** loops and **deterministic** loops. When the number of times the loop is to be executed is not known, then the loop is called non-deterministic loop otherwise it is called a deterministic loop.

C supports **while**, **do while**, and **for loop** constructs to help repetitive execution of a compound statement in a program. The 'while' and 'do while' loops are non-deterministic loops and the 'for' loop is a deterministic loop.

1. **The while loop:** It is the fundamental conditional repetitive control structure in C. The general form of this construct is given below:

```
while <cond> statement;
```

where **while:** is a reserved word of C.

<cond>: is a boolean expression.

statement: can be a simple or compound statement.

The sequence of operation in a while loop is as follows:

- (1) Test the condition.
- (2) If the condition is true, then execute the statement and repeat step 1.
- (3) If the condition is false, leave the loop and go on with the rest of the program.

It may be noted here that the variables used in the <cond> or boolean expression must be initialized somewhere before the while statement is encountered. It is a **pre-test** loop in the sense that the condition is tested before the body of the loop is executed.

2. **The do-while loop:** It is another conditional repetitive control structure provided by C. The syntax of this construct is given below:

```
do
{
    statement;
}
while <cond>;
```

where **do:** is a reserved word.
 statement: can be a simple or a compound statement.
 while: is a reserved word.
 <cond>: is a boolean expression.

The sequence of operations in a do-while loop is as follows:

- (1) Execute the statement.
- (2) Test the condition.
- (3) If the condition is true, then repeat steps 1 to 2.
- (4) If the condition is false, leave the loop and go on with the rest of the program.

Thus, it performs a **post-test** in the sense that the condition is not tested until the body of the loop is executed at least once.

3. **The for loop:** It is a count controlled loop in the sense that the program knows in advance as to how many times the loop is to be executed. The general form of this construct is given below:

```
for (initialization; expression; increment)
{
    statement
}
```

where **for:** is a reserved word.
 initialization: is usually an assignment expression wherein a loop control variable is initialized.
 expression: is a conditional expression required to determine whether the loop should continue or be terminated
 increment: it modifies the value of the loop control variable by a certain amount.
 statement: can be a simple or a compound statement.

The loop is executed with the loop control variable at initial value, final value and the values in between.

We can increase the power of for-loop with the help of the comma operator. This operator allows the inclusion of more than one expression in place of a single expression in the 'for' statement as shown below:

```
for (exp1a, exp1b; exp2; exp3a, exp3b;) statement;
```

Consider the following program segment:

```

:
for (i = 1, j = 10; i <= 10; i++, j--)
{
    :
}
```

The variable *i* and *j* have been initialized to values 1 and 10, respectively. Please note that these initialization expressions are separated by a 'comma'. However, the required semicolon remains as such. During the execution of the loop, *i* increases from 1 to 10 whereas *j* decreases from 10 to 1 simultaneously. Similarly, the increment and decrement operations have been separated by a 'comma' in the 'for statement'.

Though the power of the loop can be increased by including more than one initialization and increment expression separated with the comma operator but there can be only one test expression which can be simple or complex.

1.10.3.1 The 'break' and 'continue' Statements The **break** statement can be used in any 'C' loop to terminate execution of the loop. We have already seen that it is used to exit from a switch statement. In fact, whenever the break statement is encountered in a loop the control is transferred out of the loop. This is used in a situation where some error is found in the program inside the loop or it becomes unnecessary to continue with the rest of the execution of the loop.

Consider the following program segment:

```
:
while (val != 0)
{
    :
    printf ("\n %d", val);
    if (val < 0) {
        printf ("\n Error in input");
        break;
    }
    :
}
```

Whenever the value contained, in variable `val` becomes negative, the message 'Error in input' would be displayed and because of the break statement the loop will be terminated.

The **continue** statement can also be used in any 'C' loop to bypass the rest of the code segment of the current iteration of the loop. The loop, however, is not terminated. Thus, the execution of the loop resumes with the next iteration. For example, the following loop computes the sum of positive numbers in a list of 50 numbers:

```
:
sum = 0;
for (i = 0; i < 50; i ++ )
{
    printf ("\n %d", val);
    if (val <= 0) continue;
    sum = sum + val;
}
:
```

It may be noted here that continue statement has no relevance as far as the switch statement is concerned. Therefore, it cannot be used in a switch statement.

1.10.4 The `exit()` Function

In the event of encountering a fatal error, the programmer may desire to terminate the program itself. For such a situation, 'C' supports a function called `exit()` which can be invoked by the programmer to exit from the program. This function can be called from anywhere inside the body of the program. For normal termination, the programmer can include an argument '0' while invoking this library function as shown below:

```
#include <stdio.h>
#include <process.h>
void main()
```

```

{
    :
    if (error) exit (0);
    :
}

```

It may be noted here that the file `process.h` has to be included as header file because it contains the function prototype of the library function `exit()`.

1.10.5 Nested Loops

It is possible to nest one loop construct inside the body of another. The inner and outer loops need not be of the same construct as shown below.

```

while <cond>
{
    do
    {
        } while <cond>;
    for (init; exp; inc)
    {
        :
    }
    :
}

```

The rules for the formation of nested loops are:

- (1) An outer for loop and an inner for loop cannot have the same control variable.
- (2) The inner loop must be completely nested inside the body of the outer loop.

1.10.6 The Goto Statement (Unconditional Branching)

The unconditional transfer of control means that the sequence of execution will be broken without performing any test and the control will be transferred to some statement other than the immediate next one. C supports the following statement for this purpose.

```
goto <statement label>
```

where `goto` is a reserved word.

`<statement label>` is an identifier used to label the target statement.

The `goto` statement causes control to be transferred to the statement whose label is specified in the `goto` statement. Consider the following program segment:

```

-
-
goto rpara;
-
-
rpara: -
-
-

```

The goto statement will cause the control to be transferred to a statement whose label is rpara. The normal flow of execution will continue from this statement (i.e., having label rpara) onwards. Consider the following program segment:

```

:
k = 50;
back: I++
:
sum = sum + k*I ;
k--;
goto back;
:

```

It is clear from the above segment that as and when the control reaches the goto statement, it is transferred to the statement with label back.

Statement label A statement label is an identifier which can be placed before any C statement followed by a colon (i.e., :) as shown below:

again: C = A + B;

Thus, **again** is a label attached to the statement C = A + B and the control of execution can be transferred to this statement by the following goto statement.

```
goto again;
```

It may be noted here that the transfer of control out of a control structure is allowed. However, a goto branch into a control structure is not allowed.

Some example programs using control structures are given below.

Example 3: Write a program that reads three numbers and prints the largest of them.

Solution: The if-else ladder would be used. The required program is given below:

```

/* This program displays the largest of three numbers */
#include <stdio.h>
main()
{
    int A, B, C;
    printf ("\n Enter the Numbers A, B, C:");
    scanf ("%d %d %d", &A, &B, &C);
    /* if-else ladder */

    if (A > B)
        if ( A > C)
            printf ("\n A is largest");
        else
            printf ("\n C is largest");
        else
            if (B > C)
                printf ("\n B is largest");
            else

```



```

        printf ("\n C is largest");
        /* end of ladder */
    }

```

Example 4: Write a program that prints all perfect numbers between 2 and 9999, inclusive. A perfect number is a positive integer such that the number is equal to the sum of its proper divisors. A proper divisor is any divisor whose value is less than the number.

Solution: The remainder operator ‘%’ would be used to find whether a number is divisor of another or not. The maximum value of perfect divisor cannot be more than the half the value of the number.

The required program is given below:

```

/* This program finds out all the perfect numbers between 2 and 9999 both
inclusive */
#include <stdio.h>
main()
{
    int num, divisor;
    int sum;
    char ch;

    for (num = 2; num , = 9999; num++)
    {
        sum = 1; /* 1 is proper divisor of all positive integers */

        for (divisor = 2; divisor <= num/2; divisor++)
        {
            if (num % divisor == 0)
                sum = sum + divisor;
        }
        if ( sum == num )
            printf ("\n  %d is a perfect number", num);
    }
}

```



1.11 INPUT-OUTPUT FUNCTIONS (I/O)

Most of the programs read and write data through input/output devices such as screen, keyboard, printer, and disk devices, etc. Obviously, the programmer has to learn how to perform correct I/O operations so that efficient programs can be written. We have already used two C functions `printf()` and `scanf()` in the previous examples. In the following sections, we will discuss some of the following input output functions:

Function	Type	Description
(1) <code>getchar()</code>	stream	reads a character from a stream
(2) <code>putchar()</code>	stream	writes a character to the stream

(3) <code>getc()</code>	stream	reads a character from a stream
(4) <code>putc()</code>	stream	writes a character to a stream
(5) <code>gets()</code>	stream	reads a string from a stream
(6) <code>puts()</code>	stream	writes a string to a stream
(7) <code>cgets()</code>	console	reads a string from system consol
(8) <code>getche()</code>	console	reads a character from system consol
(9) <code>putch()</code>	console	writes a character on system consol
(10) <code>cputs()</code>	console	writes a string on system consol

In order to use the above functions, the programmer must include the '`stdio.h`' header file at the beginning of his program. The C or 'C environment' assumes keyboard as the standard input device and VDU as standard output device. A consol comprises both keyboard and VDU.

By default, all stream I/Os are buffered. At this point, it is better that we understand the term buffered I/O before we proceed to discuss the stream I/O functions.

1.11.1 Buffered I/O

Whenever we read from the keyboard or from a file stored on a device, a block of data is stored in a buffer called input buffer. The input stream, therefore, reads from the buffer with the help of a pointer until the input buffer is empty (see Figure 1.7). As soon as the buffer becomes empty, the next block of data is transferred into the buffer from the I/O device.

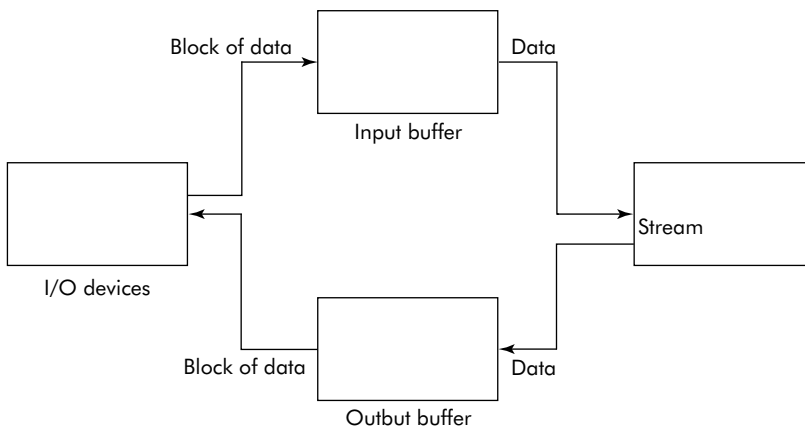


Fig. 1.7 Buffered I/O

Similarly, data written by an output stream pointer is not directly written onto the device but into an output buffer. As soon as the output buffer becomes full, a block of data is written on to the I/O device making output buffer empty.

When the header file `stdio.h` is included in a program, the following standard stream pointers are automatically opened:

Name	Meaning
<code>stdin</code>	standard input
<code>stdout</code>	standard output
<code>stderr</code>	standard error
<code>stdaux</code>	auxiliary storage
<code>stdprn</code>	printer

In the computer, '`stdin`' and '`stdout`' refer to the user's console, i.e., input device as keyboard and output device as VDU (screen). The stream pointers '`stdprn`' and '`stdaux`' refer to the printer and auxiliary storage, respectively. The error messages generated by the stream are sent to standard error (`stderr`).

In order to clear the buffers, a function called '`fflush()`' can be used. For example, input buffer of standard input device can be cleared by invoking the function: '`fflush(stdin)`'.

1.11.2 Single Character Functions

C supports many input and output functions based on character. These functions read or write one character at a time. C supports both stream and consol I/O character-based functions. We will discuss them in the following sections.

- (1) **`getchar()` and `putchar()` Functions:** The `getchar()` function reads a single character from the standard input device. For example, if it is desired to read a character from keyboard in a character variable `ch` then the following statements can be used:

```
:  
char ch;  
ch = getchar();  
:
```

Since the function `getchar()` is a stream I/O function, it is buffered in the sense that the character typed by the user is not passed to the variable `ch` until the user hits the enter or return key, i.e., `\n`. The enter key is itself a character and it also gets stored in the buffer along with the character typed by the user. The entry of 'enter key' character into the buffer creates problems in the normal functioning of `getchar()` function. Therefore, it is suggested that after an input operation from the standard input device (i.e., keyboard) the input buffer should be cleared. This operation is required to avoid interference with subsequent input operations. The function call for this activity is: `fflush(stdin)`. The usage is shown below:

```
:  
char ch;  
ch = getchar();  
fflush(stdin);  
:
```

The function `putchar ()` is used to send a single character to the standard output device. The character to be displayed on the VDU screen is included as an argument to the `putchar ()` function as shown below:

```

:
ch = 'A';
putchar (ch);
:

```

Once the above program segment is executed, the character 'A' will be displayed on the screen.

The `putchar ()` function is also buffered. The function call for clearing the output buffer is `fflush (stdout)`. The output buffer is cleared on its own only when a new line character '\n' is used.

- (2) **getc() andputc() Functions:** The `getc ()` and `putc ()` functions are also character-based functions. They have been basically designed to work with files. However, these functions can also be used to read and write data from standard input and output devices by specifying `stdin` and `stdout` as input and output files, respectively.

For example, the function `getc(stdin)` is equivalent to the function `getchar ()`. Both will get a character from the standard input device (keyboard). The function `getchar ()` by default reads from keyboard whereas the argument `stdin` of function `getc(stdin)` makes it read from a file represented by the standard input device which is nothing but the keyboard.

Similarly, the function `putc(ch, stdout)` is equivalent to the function `putchar (ch)`. Both the functions send the character `ch` to the standard output device, i.e., VDU screen.

- (3) **getche() andputch() Functions:** These two functions are character-based versions of console I/O functions. Therefore, the header file `conio.h` has to be included. In fact, these functions act as an extension to the stream I/O functions. The console I/O functions read from the keyboard and write to the screen directly.

- **getche () :** This function directly reads a character from the console as soon as it is typed without waiting for the enter key (↵) to be pressed. There is another similar function `getch ()` which also reads a character from the keyboard exactly in the same manner but does not show it on the screen. This function is useful in menu selections where the programmer does not want to display the character typed by the user. In fact, the extra 'e' in the `getche ()` function stands for echo, i.e., display the character.

Examples of usage of these functions are:

- (i) `ch = getche();`
- (ii) `ch = getch();`

- **putch () :** This function directly writes a character on the console. It takes character as an argument.

Examples of usage of this function are:

- (i) `putch ('\n');` //It takes the cursor to next line of the screen.
- (ii) `putch (ch);` //It displays the character stored in the variable `ch` on the screen.
- (iii) `putch ('A');` //It displays the character 'A' on the screen.

Note: These functions cannot read special keys from the keyboard such as function keys.

1.11.3 String-based Functions

String-based functions read or write a string of characters at a time. C supports both stream and console I/O string-based functions.

- (1) **gets() and puts() functions:** These functions are string-based I/O functions. The function `gets()` reads a string from the standard input device (keyboard). It is also buffered and, therefore, the return or enter key has to be typed to terminate the input. The `fflush()` function should also be used after each `gets()` to avoid interference. For example, the program segment given below reads a string in a variable `name` from keyboard.

```

:
name = gets() ;
fflush (stdin);
:

```

Similarly, the function `puts()` is used to send a string to the standard output device. The string to be displayed on the VDU screen is included as an argument to the `puts` function as shown below:

```

:
city = "New Delhi";
puts (city);
:

```

Once this program segment is executed, the following message will be displayed on the screen:

```
New Delhi
```

1.12 ARRAYS

A detailed discussion on arrays as data structure is given in Chapter 3.

1.13 STRUCTURES

Arrays are very useful for list and table processing. However, the elements of an array must be of the same data type. In certain situations, we require a construct that can store data items of mixed data types. C supports structures for this purpose. The basic concept of a structure comes from day-to-day life. We observe that certain items are made up of components or sub-items of different types. For example, a date is composed of three parts: day, month, and year as shown in Figure 1.8.

Similarly, the information about a student is composed of many components such as name, age, roll number., and class; each belonging to different types (see Figure 1.9).

The collective information about the student as shown above is called a structure. It is similar to a record construct supported by other programming languages. Similarly, the date is also a structure. The term structure can be precisely defined as '*a group of related data items of arbitrary types*'. Each member of a structure is, in fact, a variable that can be referred to through the name of the structure.

Day	Month	Year
01	02	1998

Fig. 1.8 Composition of 'date'

Name	SACHIN KUMAR
Age	18
Roll	10196
Class	CE41
Student	

Fig. 1.9 Student

1.13.1 Defining a Structure in C

A structure can be defined in C by the keyword `struct` followed by its name and a body enclosed in curly braces. The body of the structure contains the definition of its members and each member must have a name. The declaration ends by a semicolon. The general format of structure declaration in C is given below:

```
struct <name> {
    member 1
    member 2
    :
    member n
};
```

where `struct`: is the keyword.
 `<name>`: is the name of the structure.
 `member 1, 2 ... n`: are the individual member declarations.

Let us now define the following 'C' structure to describe the information about the student given in Figure 1.9.

```
struct student
{
    char Name [20];
    int age;
    int roll;
    char class[5];
};
```

The above declaration means that the student is a structure consisting of data members: Name, age, roll, and class. A variable of this type can be declared as given below:

```
struct student x;
```

Once the above declaration is obeyed, we get a variable `x` of type `student` in the computer memory which can be visualized as shown in Figure 1.10.

The programmer is also allowed to declare one or more structure variables along with the structure declaration as shown below:

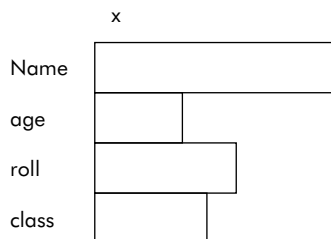


Fig. 1.10

Memory space allocated to variable `x` of type `student`

```

struct student
{
    char name [20];
    int age;
    int roll;
    char class [5];
}
stud1, stud2;

```

In the above declaration, the structure declaration and the variable declaration of the structure type have been clubbed, i.e., the structure declaration is followed by two variable names `stud1` and `stud2` and the semicolon.

1.13.2 Referencing Structure Elements

The structure variable `x` has four members: `name`, `age`, `roll`, and `class`. These members can be designated as: `x.name`, `x.age`, `x.roll`, and `x.class`, respectively. In this notation, an individual member of the structure is designated or qualified by the structure variable name followed by a period and the member name. The period is called a structure member operator or simply a dot operator.

However, the information of a student (Figure 1.10) can be stored in structure variable `x` in either of the following ways:

- (1) Initialize the elements of the structure


```

strcpy (x.name, "SACHIN KUMAR");
x.age = 18;
x.roll = 10196;
strcpy (x.class, "CE41");

```
- (2) The variable `x` can be read in a C program by the I/O statements as shown below:


```

gets(x.name);
fflush(stdin);
scanf("% d %d", & x age., & x.roll);
gets(x.class);

```

1.13.3 Arrays of Structures

An array of structures can be declared just like an ordinary array. However, the structure has to be defined before an array of its type is declared. For example, a 50-element array of type `student` called `stud_list` can be defined as shown below:

```

struct student {
    char name [20];
    int roll;
    int sub1, sub2, sub3, sub4;
    int total;
};
struct student stud_list [50];    /* declaration of an array of structures*/

```

1.13.4 Initializing Structures

When a structure variable is declared, its data members are not initialized and, therefore, contain undefined values. Similar to arrays, the structure variables can also be initialized. For instance, the structure shown in Figure 1.10 can be declared and initialized as shown below:

```
struct student
{
    char name[20];
    int age;
    int roll;
    char class[5];
};

/* Initialize variable of type student*/

struct student stud1 = {"SACHIN KUMAR", 18, 10196, "CE41"};
```

Please notice that the values are enclosed within curly braces. The values are assigned to the members of the structure in the order of their appearance, i.e., first value is assigned to the first member, second to second, and so on.

1.13.5 Assignment of Complete Structures

It has been appreciated that the individual fields of a structure can be treated as simple variables. The main benefit of a structure, however, is that it can be treated as a single entity. For example, if two structure variables `stud1` and `stud2` have been declared as shown below

```
student stud1, stud2;
```

then the following assignment statement is perfectly valid.

```
stud1 = stud2;
```

This statement will perform the necessary internal assignments. The concept of structure assignment can be better understood by the following example program. In this program, a structure variable `stud1` is initialized with certain values and it is assigned to another structure variable `stud2`. The contents of the variable `stud2` are displayed.

```
/* This program illustrates copying of complete structures */
#include <stdio.h>
main()
{
    struct student {
        char name[20];
        int roll;
        int age;
        char class[6];
    };

    /* initialize a variable of type student */
```

```

struct student stud1 = {"Sachin Kumar", 101, 16, "CE_IV"};
struct student stud2;
        /* assign stud1 to stud2 */
stud2 = stud1;
        /* Display contents of stud2 */
printf ("\n The copied contents are....");
printf ("\n Name: %s", stud2.name);
printf ("\n Roll: %d", stud2.roll);
printf ("\n Age : %d" , stud2.age);
printf ("\n class: %s", stud2.class);
}

```

1.13.6 Nested Structures

Nested structures are structures as member of another structure. For instance, the date of birth (DOB) is a structure within the structure of a student as shown in figure below. These types of structures are known as nested structures.

Name	Roll	DOB			Marks
		DD	MM	YY	

The nested structure shown in Figure 16.4 can be declared in a program as shown below:

```

/* Nested structures */
struct date
{
    int dd;
    int mm;
    int yy;
};
struct student
{
    char name[20];
    int roll;
    struct date dob;
    int marks;
};

```

It may be noted here that the member of a nested structure is referenced from the outermost to inner most with the help of dot operators. Let us declare a variable stud of type student:

```

struct student stud;

```

The following statement illustrates the assignment of value 10 to the member `mm` of this nested structure `stud`:

```
stud.dob.mm = 10;
```

The above statement means: store value 10 in `mm` member of a structure `dob` which itself is a member of structure variable called `stud`.

1.14 USER-DEFINED DATA TYPES

In addition to the simple data types such as `integer`, `float`, `char`, etc., C allows the users to declare new data types called user-defined data types. These data types can be declared with the help of a keyword called `typedef` whose general form is given below:

```
typedef <type> <new_type>
```

where `typedef`: is a reserved word.
 `<type>`: is an existing data type.
 `<new_type>`: is the desired user-defined data type.

For example, consider the following declaration:

```
typedef float xyz;
```

In this declaration, an existing data type (`float`) has been redefined as `xyz`. Now, the user can use `xyz` as a synonym for the type `float` as a new type as shown below.

```
xyz x1, x2;
```

In fact, the above declaration is equivalent to the following declaration:

```
float x1, x2;
```

Apparently, it looks as if we have not achieved anything great because the user will have to remember `xyz` as a new type whereas its equivalent `float` type is already available. Nevertheless, the utility of `typedef` lies in a situation where the name of a particular type is very long. The user can also self document his code as shown in the following declaration:

```
typedef float salary;      :  
salary wages-of-month;
```

In the above declaration, the variable `wages-of-month` has been declared of type `salary`, where `salary` itself is of `float` type. Thus, we can see that readability of a program can be enhanced by self documenting the variables and their data types with the help of `typedef`.

For example, the declaration:

```
typedef int age;
```

can be used to increase the clarity of code as shown below :

```
age boy, girl, child;
```

The above statement clearly suggests as to what data the variables `boy`, `girl`, and `child` are going to store. Similarly, `typedef` can also be used to define arrays.

```
typedef float lists [20];  
lists BP, SAL, GSAL;
```


These statements are equivalent to the following:

```
float BP [20], SAL [20], GSAL [20]
```

`typedef` is also extremely useful for shortening the long and inconvenient declarations of structures. For instance, the structure `struct` can be declared as a `typedef` as shown below:

```
typedef struct {
    char name [20];
    int age;
    int roll;
    char class[6];
} student;
```

Now the structure `student` has become the type and a variable of this type can be defined as:

```
student stud1, stud2;
```

The above declaration means that `stud1` and `stud2` are variables of type `student`.

1.14.1 Enumerated Data Types

A user can define a set of integer type identifiers in the form of new data types called enumerated data types. For instance, the user can create a new data type called `color` with the help of `enum` definition as shown below:

```
enum color {
    Blue,
    Yellow,
    White,
    Black
};
```

where `enum` is the keyword that defines an enumerated data type.

The `enum` declaration given above means that a new type, `color`, has been defined. This type contains the constant values: `Blue`, `Yellow`, `White`, and `Black`. Each of these symbols stands for an integer value. By default, the value of the first symbol (i.e., `Blue`) is 0, the value of the second symbol is 1, and so on. Once the type `color` has been defined, then variables of that type can be defined in the following manner:

```
enum color A, B;
```

Now, the variable `A` of type `color` can use the enumerations (`blue`, `yellow`, etc.) in a program.

Thus, the following operations on variable `A` are valid:

```
A = red;
:
if (A == red)
    x = x + 10;
else
    x = x - 5;
```

Enumerated data type can be precisely defined as the ordered set of distinct constant values defined as a data type in a program.

Examples of some valid enumerated data types are:

(i) enum furniture

```
{
    chair,
    table,
    stool,
    desk
};
```

(ii) enum car

```
{
    Maruti,
    Santro,
    Micra,
    Indigo
}
```

It may be noted here that the ordering of elements in enumerated data type is specified by their position in the enum declaration. Thus for furniture type, the following relations hold good:

Expression	Value
Chair<Table	True
Stool<Desk	True
Table==Desk	False
Chair<Desk	True

The enumerated types can also be used in a switch construct. Consider the following program segment.

```
:
/* Enumerated type declaration */
enum furniture
{
    chair,
    table,
    stool,
    desk
};
main()
{
    /* Enumerated variable declaration */
    enum furniture item;
    float cost;
    :
    switch item
```

```

{
    case chair : cost = 250; break;
    case table : cost = 700; break;
    case stool : cost = 120; break;
    case desk : cost = 1500; break;
}
:

```

Similarly, enumerated data types can also be used as a control variable in a for loop as shown below:

```

:
for (item = chair; item <= desk; item++)
{
    :
    :
}

```

The enumerations are also called symbolic constants because an enumeration has a name and an integer value assigned to it which cannot be changed in the program. Therefore, enumerated data types cannot be plainly used in I/O statements. As a result, the statements `scanf`, `printf`, `gets`, `puts`, etc. cannot be directly applied to these data types because the results will not be as desired.

1.15 UNIONS

Suppose in a fete, every visitor is assured to win any one of the items listed below:

- (1) Wall clock
- (2) Toaster
- (3) Electric Iron
- (4) DVD player

Obviously, a visitor to the fete must carry a bag large enough to carry the largest item so that any one of the items won by him can be held in that bag. With the same philosophy in mind, the designers of C introduced a special variable called union which may hold objects of different sizes and types but one at a time. Though a union is a variable and can hold only one item at a time, it looks like a structure. The general format of a union is given below:

```

union <name> {
    member 1
    member 2
    :
    member
};

```

where **union:** is a keyword.
 <name>: is the name of the union.
 member 1, 2, ... n: are the individual member declarations.

1.16 FUNCTIONS

A function is a subprogram that can be defined by the user in his program. It is a complete program in itself in the sense that its structure is similar to `main()` function except that the name `main` is replaced by the name of the function. The general form of a function is given below:

`<type> <name> (arguments)`

- where
- `<type>`: is the type of value to be returned by the function. If no value is returned, then keyword `void` should be used.
 - `<name>`: is a user-defined name of the function. The function can be called from another function by this name.
 - `arguments`: is a list of parameters (parameter is the data that the function may receive when called from another function). This list can be omitted by leaving the parameters empty.

The program segment enclosed within the opening brace and closing brace is known as the function body. For example, the `main` function in C is written as shown below:

```
main()  
{  
}
```

Let us write a function `tri_area()` which calculates the area of a triangle. Two variables `base` and `height` will be used to read the values from the keyboard. The function is given below:

```
float tri_area()  
{  
    float base, height, area;  
    printf ("\n Enter Base and Height");  
    scanf ("%f %f", &base, &height);  
    area = (0.5)*base*height;  
    return (area);  
}
```

It may be noted here that the function `tri_area()` has defined its own variables. These are known as local variables. The advantage of their usage is that once the execution of the function is over these variables are disposed off by the system, freeing the memory for other things.

Let us write another function `clear()` which clears the screen of the VDU

```
/* This function clears the screen by writing blank lines on the screen */  
void clear()  
{  
    int i;  
    for (i = 0; i <= 30; i++)  
        printf ("\n");  
}
```

Similarly, let us write a function `add` which receives two parameters `x` and `y` of type integer from the calling function and returns the sum of `x` and `y`.

```
int add (int x, int y)
{
    int temp;
    temp = x + y;
    return temp;
}
```

In the above function, we have used a local variable `temp` to obtain the sum of `x` and `y`. The value contained in `temp` is returned through `return` statement. In fact, variable `temp` has been used to enhance the readability of the function only, otherwise it is unnecessary. The function can be optimized as shown below:

```
int add (int x, int y)
{
    return (x + y);
}
```

Thus, a function has following four elements:

- (1) A name
- (2) A body
- (3) A return type
- (4) An argument list

The name of the function has to be unique so that the compiler can identify it. The body of the function contains the C statements which define the task performed by the function. The return type is the type of value that the function is able to return to the calling function. The argument or parameter list contains the list of values of variables from outside required by the function to perform the given task. For example, the function given above has a name `add`. It has a body enclosed between the curly braces. It can return a value of type `int`. The argument list is `x` and `y`.

1.16.1 Function Prototypes

Similar to variables, all functions must be declared before they are used in a program. C allows the declaration of a function in the calling program with the help of a function prototype. The general form of a function prototype in the calling program is given below:

`<type> <name> (arguments);`

where `<type>`: is the type of value to be returned by the function.
 `<name>`: is a user-defined name of the function.
 `arguments`: is a list of parameters.
 `;`: the semicolon is necessary.

For example, the prototype declarations for the functions `tri_area()`, `clear()`, and `add()` can be written as shown below:

```
void tri_area();
void clear();
int add(int x, int y);
```

It may be noted that a function prototype ends with a semicolon. In fact, the function prototype is required for those functions which are intended to be called before they are defined. In absence of

a prototype, the compiler will stop at the function call because it cannot do the type checking of the arguments being sent to the function. Therefore, the main purpose of function prototype is to help the compiler in static type checking of the data requirement of the function.

1.16.2 Calling a Function

A function can be called or invoked from another function by using its name. The function name must be followed by a set of actual parameters, enclosed in parentheses and separated by commas. A function call to function `tri_area` from the main program can be written as:

```
tri_area();
```

This statement will transfer the control to function `tri_area`. Since this function does not require any parameter, the argument list is empty. Similarly, in order to call the function 'clear', the following statement will be sufficient:

```
clear();
```

A complete C program involving the two functions `tri_area` and `clear` is given below:

```
#include <stdio.h>
                /* Function ptototypes */
float tri_area();
void clear();

main()
{
    char ch;
    int flag;
    float Area;
    clear();    /* Clear the screen */
    flag = 0;
    while (flag == 0)
    {
        Area = tri_area(); /* call tri_area to compute area */
        printf ("\n Area of Triangle = %5.2f", Area);

        printf ("\n Want to calculate again: Enter Y/N");
        fflush(stdin);
        ch = getchar();
        if (ch == 'N' || ch == 'n')
            flag = 1;
        clear();
    }
}

float tri_area()
{
    float base, height, area;
```

```

        printf ("\n Enter Base and Height");
        scanf ("%f %f", &base, &height);
        area = (0.5)*base*height;
        return (area);
    }

    /* This function clears the screen by writing blank lines on the screen */
    void clear()
    {
        int i;
        for (i = 0; i<= 30; i++)
            printf ("\n");
    }

```

It may be noted here that when this program is executed, the first statement to be executed is the first executable statement of the function `main()` of the program (i.e., `clear()` in this case). A function will be executed only after it is called from some other function or itself.

The variables `base`, `height`, and `area` are local to function `tri_area` and they are not available in the `main()` function. Similarly, the variable `flag` declared in function `main()` is also not available to functions `tri_area()` and `clear()`.

However, the variables declared outside any function can be accessed by all the functions. Such variables are called **global variables**.

Let us consider the following program which declares the variables `flag` and `area` outside the `main` function instead of declaring them inside the functions `main()` and `tri_area()`, respectively.

```

#include <stdio.h>
/* Function prototypes */
void tri_area();
void clear();
/* Global Variables */
int flag;
float area;
main()
{
    char ch;    /* Local Variable */
    clear();    /* Clear the screen */
    flag = 0;
    while (flag == 0)
    {
        tri_area(); /* call tri_area to compute area */
        printf ("\n Area of Triangle = %5.2f", area);
        printf ("\n Want to calculate again: Enter Y/N");
        fflush(stdin);
        ch = getchar();
        if (ch == 'N' || ch == 'n')
            flag = 1;
    }
}

```

```

        clear();
    }
}
/* Function Definitions */
void tri_area()
{
    float base, height;
    printf ("\n Enter Base and Height");
    scanf ("%f %f", &base, &height);
    area = (0.5)*base*height;
}
/* This function clears the screen by writing blank lines on the screen */
void clear()
{
    int i;
    for (i = 0; i<= 30; i++)
        printf ("\n");
}

```

It may be noted in the above program that the variable `area` is available to both the functions `tri_area()` and `main()`. On the other hand, the local variables `base` and `height` are available to function `tri_area` only. They are neither available to `main()` function nor to the function `clear()`. Similarly, variable `i` is also local to function `clear()`.

In our previous programs, we have used our own function `clear()` to clear the screen. C also provides a function called `clrscr()` that clears the screen. From now onwards, we will use this function. However, in order to use `clrscr()` in a program we will have to include `conio.h`, a header file at the beginning of the program.

Note: We should not use global variables in our programs as they make the program or function insecure and error prone.

1.16.3 Parameter Passing in Functions

We have seen that two-way communication between functions can be achieved through global variables. This technique has its own limitations and drawbacks. Let us now see how this communication between the calling function and the called function can be improved and made fruitful.

The two-way communication between the various functions can be achieved through parameters and return statement as done in previous section. The set of parameters defined in a function are called **formal** or **dummy parameters** whereas the set of corresponding parameters sent by the calling function are called **actual parameters**. For instance, the variables `x` and `y` in the argument list of function `add()` are formal parameters. The parameters included in the function call, i.e., `val1` and `val2` of `main()` are actual parameters.

The function that receives parameters can be called in one of the following two ways:

- (1) Call by value.
- (2) Call by reference.

- (1) **Call by value:** For better understanding of the concept of parameter passing, let us consider the function `big()` given below. It receives two values as dummy arguments `x` and `y` and returns the bigger of the two through a return statement.


```
/* This function compares two variables and returns the bigger of the two */
int big (int x, int y)
{
    if (x > y)
        return x;
    else
        return y;
}
```

The function `big()` can be called from the function `main()` by including the variables `a` and `b` as the actual arguments of the function call as shown below.

```
main()
{
    int a = 30;
    int b = 45;
    int c;
    c = big (a, b);
    printf ("\n The bigger = %d ", c);
}
```

The output of the above program would be:

The bigger = 45

Let us now write a program which reads the values of two variables `a` and `b`. The values of `a` and `b` are sent as parameters to a function `exchange()`. The function exchanges the contents of the parameters with the help of a variable `temp`. The changed values of `a` and `b` are printed.

```
#include <stdio.h>
#include <conio.h>
void exchange( int x, int y);
main()
{
    int a, b;
    clrscr();    /* clear screen */
    printf ("\n Enter two values");
    scanf ("%d %d", &a, &b);
    exchange(a,b);
    /* print the exchanged contents */
    printf ("\n The exchanged contents are: %d and %d", a, b);
}
void exchange (int x, int y)
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}
```

The above program seems to be wonderful but it fails down badly because variables `a` and `b` do not get any values back from the function `exchange()`. The reason is that although all values of actual parameters (i.e., `a` and `b`) have been passed on to formal parameters (i.e., `x` and `y`) correctly, the changes done to the values in the formal parameters are not reflected back into the calling function.

Once the above program is executed for input value (say 30 and 45), the following output is produced:

The exchanged contents are: 30 and 45

The reason for this behaviour is that at the time of function call, the values of actual parameters `a` and `b` get copied into the memory locations of the formal parameters `x` and `y` of the function `exchange()`.

It may be noted that the variables `x` and `y` have entirely different memory locations from variables `a` and `b`. Therefore any, changes done to the contents of `x` and `y` are not reflected back to the variables `a` and `b`. Thus, the original data remains unaltered. In fact, this type of behaviour is desirable from a pure function.

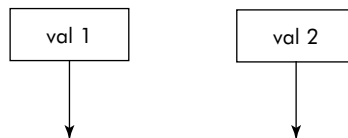
Since in this type of parameter passing, only the values are passed from the calling function to the called function, the technique is called **call by value**.

- (2) **Call by reference:** If the programmer desires that the changes made to the formal parameters be reflected back to their corresponding actual parameters, then he should use call by reference method of parameter passing. This technique passes the addresses or references of the actual parameters to the called function. Let us rewrite the function `exchange()`. In fact, in the called function, the formal arguments receive the addresses of the actual arguments by using pointers to them.

For instance, in order to receive the addresses of actual parameters of `int` type the `exchange()` function can be rewritten as shown below:

```
void exchange (int *val1, int *val2)
{
    int temp;
    temp = *val1;
    *val1 = *val2;
    *val2 = temp;
}
```

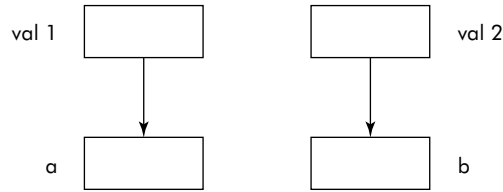
It may be noted that the above function has used two pointers called `val1` and `val2` to receive the addresses of two `int` type actual parameters. However, initially, these are dangling pointers as shown below:



The `exchange()` function would be called from the `main()` function by sending the addresses of the actual arguments as shown below:

```
exchange (& a, & b);
```

After the `exchange()` function is called by the above statement, the addresses of the actual parameters `a` and `b` would be assigned to pointers `val1` and `val2`, respectively as shown below:



Since the function exchanges the contents of variables being pointed by the pointers `val1` and `val2`, the contents of variables `a` and `b` get exchanged which is the goal of this exercise.

The complete program is given below:

```

#include <stdio.h>
#include <conio.h>
void exchange( int *val1, int *val2);
main()
{
    int a, b;
    clrscr();    /* clear screen */
    printf ("\n Enter two values");
    scanf ("%d %d", &a, &b);
    exchange(&a,&b);
    /* print the exchanged contents */
    printf ("\n The exchanged contents are: %d and %d", a, b);
}
void exchange (int *val1, int *val2)
{
    int temp;
    temp = *val1;
    *val1 = *val2;
    *val2 = temp;
}
  
```

After the above program is executed for input values (say 30 and 45), the following output is produced:

The exchanged values are: 45 30

Now, we can see that the program has worked correctly in the sense that changes made inside the function `exchange()` have been available to corresponding actual parameters `a` and `b` in the function `main()`. Thus, the address operator `&` has done the required task.

The main advantage of call by reference method is that more than one value can be received back from the called function.

Note:

- (1) As far as possible, we should use 'pass by value' technique of parameter passing. If some value is to be received back from the function, it should be done through a return statement of the function.

- (2) A function call can use both the methods, i.e., call by value and call by reference simultaneously. For example, the following is a valid function prototype:

```
void calc (int a, float *b, float *c);
```

- (3) When actual parameters are passed from the calling function to the called function by call by value method, the names of the corresponding dummy parameters could be the same or different. However, it is suggested that they may be named different in order to make the program more elegant.

- (3) **Array parameters:** Arrays can also be passed as parameters to functions. This is always done through call by reference method. In C, the name of the array represents the address of the first element of the array and, therefore, during the function call only the name of the array is passed from the calling function. For example, if an array xyz of size 100 is to be passed to a function some () then the function call will be as given below:

```
some (xyz);
```

The formal parameters can be declared in the called function in three ways. The different ways are illustrated with the help of a function some () that receives an array xyz of size 100 of integer type.

Method I:

In this method, the array is declared without subscript, i.e., of an known size as shown below:

```
void some (int xyz [ ])  
{  
}
```

The C compiler automatically computes the size of the array.

Method II:

In this method, an array of same size and type is declared as shown below:

```
void some (int xyz [100])  
{  
}
```

Method III:

In this method, the operator * is applied on the name of the array indicating that it is a pointer.

```
void some (int *xyz)  
{  
}
```

All the three methods are equivalent and the function some () can be called by the following statement:

```
some (xyz);
```

Note:

- (1) The arrays are never passed by value and cannot be used as the return type of a function.
- (2) In C, the name of the array represents the address of the first element or the zeroth element of the array. So when an array is used as an argument to a function, only the address of the array gets passed and not the copy of the entire array. Hence, any changes made to the array inside the function are automatically reflected in the main function. You do not have to use ampersand (&) with array name even though it is a call by reference method. The ampersand (&) is implied in the case of array.

1.16.4 Returning Values from Functions

Earlier in this chapter, we have used functions which either return no value or integer values. The functions that return no value are declared as void. We have been prefixing int to functions which return integer values. This is valid but unnecessary because the default return value of a function in C is of type int. Thus, the following two declarations are equivalent in the sense that we need not prefix int to a function which returns an integer value.

```
int add (int a, int b);
add (int a, int b);
```

On the other hand, whenever a function returns non-integer values, the function must be prefixed with the appropriate type. For example, a function xyz that returns a value of type float can be declared as shown below:

```
float xyz (.....)
{ float a;
  :
  return a
}
```

Similarly, a function ABC that returns a value of type char can be declared as shown below:

```
char ABC ( ..... )
{ char ch;
  :
  return ch;
}
```

1.16.5 Passing Structures to Functions

Similar to variables, the structures can also be passed to other functions as arguments both by value and by reference.

- (1) **Call by value:** In this method of passing the structures to functions, a copy of the actual argument is passed to the function. Therefore, the changes done to the contents of the structures inside the called function are not reflected back to the calling function.

The following program uses a structure variable called stud of type student and passes the structure variable by value to a function print_data(). The function print_data() displays the contents of the various members of the structure variables.

```
/* This program illustrates the passing of the structures by value to a
function */
#include <<stdio.h>>
struct student
{
    char name [20];
    int age;
    int roll;
    char class [5];
};
```

```

/* Prototype of the function */
void print_data (struct student Sob);
main()
{
    struct student stud;
    printf ("\n Enter the student data");
    printf ("\nName:"); fflush(stdin);
    gets (stud.name);
    printf ("\nAge:"); scanf("%d", &stud.age);
    printf ("\nRoll:"); scanf("%d", &stud.roll);
    printf ("\nClass:"); fflush(stdin);
    gets (stud.class);
    print_data(stud); /*The structure is being passed by value*/
}
void print_data (struct student Sob)
{
    printf ("\n The student data..");
    printf ("\nName:"); fflush(stdout);
    puts (Sob.name);
    printf ("Age: %d",Sob.age);
    printf ("\nRoll: %d",Sob.roll);
    printf (",\nClass:"); fflush(stdout);
    puts (Sob.class);
}

```

- (2) **Call by reference:** In this method of passing the structures to functions, the address of the actual structure variable is passed. The address of the structure variable is obtained with the help of address operator '&'. Thus, the changes done to the contents of such a variable inside the function are reflected back to the calling function. The program given below illustrates the usage of call by reference method.

This program reads the data of a student in the function `main()` and passes the structure by reference to the function called `print_data()`.

```

/* This program illustrates the passing of the structures by reference to
a function */
#include <stdio.h>
struct student
{
    char name [20];
    int age;
    int roll;
    char class [5];
};
/* Prototype of the function */
void print_data (struct student &Sob);
main()
{

```

```

    struct student stud;
    printf ("\n Enter the student data");
    printf ("\nName:"); fflush(stdin);
    gets (stud.name);
    printf ("\nAge:"); scanf("%d",&stud.age);
    printf ("\nRoll:"); scanf("%d",&stud.roll);
    printf ("\nClass:"); fflush(stdin);
    gets (stud.class);
    print_data(&stud); /*The structure is being passed by reference*/
}
void print_data (struct student &Sob)
{
    struct student *ptr;
    ptr = sob;
    printf ("\n The student data..");
    printf ("\nName:"); fflush(stdout);
    puts (ptr->name);
    printf ("Age: %d",ptr->age);
    printf ("\nRoll: %d",ptr->roll);
    printf ("\nClass:"); fflush(stdout);
    puts (ptr->class);
}

```

It may be noted here that structures are usually passed by reference in order to prevent the overheads associated with the technique of passing structures by value. The overheads are extra memory space and CPU time used to pass bigger structures.

- (3) **Returning structures from functions:** It is possible to supply a structure as a return value of a function. This can be done by specifying structure as the return type of the function. This feature of C is helpful in a situation where a structure has been passed by value to a function and the changes done to the contents of the structure are needed by the calling function without disturbing the original contents. This concept is illustrated in the example given below:

Example 5: Write a program that reads the data of a students whose structure is given in the figure shown below. The structure is then passed by value to a function called `change-data()` where the name of the student is capitalized. The changed contents of the student data are returned by the function to the function `main()`.

Emp_Code	EmpAdd		Desig
	Apt	Colony	State
	No	FL	

Solution: We will use the function `toupper()` in this program to capitalize the name of the student inside the function `change_data()`. The required program is given below:

```
/* This program illustrates the process of returning a structure from a
function */
#include <stdio.h>
#include <ctype.h>
struct student
{
    char name [20];
    int age;
    int roll;
    char class [5];
};

/* Prototype of the function */
void print_data (struct student *ptr);
struct student change_data (struct student Sob);
main()
{
    struct student stud, newstud;
    printf ("\n Enter the student data");
    printf ("\nName:"); fflush(stdin);
    gets (stud.name);
    printf ("\nAge:"); scanf("%d",&stud.age);
    printf ("\nRoll:"); scanf("%d",&stud.roll);
    printf ("\nClass:"); fflush(stdin);
    gets (stud.class);
    /* call function to change data */
    newstud = change_data(stud);
    printf ("\n The changed data ..");
    print_data(&newstud); /*The structure is being passed by value*/
}

/* This function changes the data of a student */
struct student change_data (struct student Sob)
{
    int i = 0;
    while (Sob.name[i] != '\0')
    {
        /* Capitalize the letters */
        Sob.name[i] = toupper (Sob.name[i]);
        i++;
    }
    return (Sob);
}

void print_data (struct student *ptr)
{
    printf ("\n The student data..");
```



```

    printf ("\nName:"); fflush(stdout);
    puts (ptr->name);
    printf ("Age: %d", ptr->age);
    printf ("\nRoll: %d", ptr->roll);
    printf ("\nClass:"); fflush(stdout);
    puts (ptr->class);
}

```

1.17 RECURSION

We have already discussed *iteration* through various loop structures discussed in Section 1.10.3. Iteration can be defined as an act of performing computation each time by the same method and the result of computation is utilized as the source of data in the next repetition of the loop. For example the factorial of an integer N can be computed by the following iterative loop:

```

fact = 1;
for (i = 1; i <= N; i++)
    fact = fact * i;
printf ("\n The factorial of %d %d", N, fact);

```

We can also use an alternative approach wherein *we reduce the problem into a smaller instance of the same problem*. For example, factorial of N can be reduced to a product of N and the factorial of N – 1 as given below:

$$\text{Fact (N)} = N \times \text{Fact (N - 1)}$$

In the above statement, a function Fact(N) has been defined in terms of Fact(N – 1), where Fact (N – 1) is a smaller problem as compared to Fact(N). This type of definition is known as recursive definition. *Recursion* can be defined as: the ability of a concept being defined within the definition itself. In programming terms, recursion is defined as the ability of a function being called from within the function itself.

Now, by the same definition, the factorial of (N – 1) can be defined as given below :

$$\text{Fact(N - 1)} = (N - 1) \times \text{Fact(N - 2)}$$

We can continue this process till we end up with Fact(0) which is equal to 1, and is also the terminating condition for this reduction process. The terminating condition for a recursive function is also called the *basic solution*. For instance, the basic solution for a list of elements is that if the list contains only one element then this element is largest as well as smallest element. Similarly, if a list contains only one element then it is already sorted.

The process described above can be implemented by using a recursive function Fact() which is defined in terms of itself as shown below:

$$\text{Fact (N)} = \begin{cases} 1 & \text{if } N = 0 \text{ or } N = 1 \\ N * \text{Fact (N - 1)} & \text{otherwise} \end{cases}$$

In C such recursive functions can be implemented by making the function call itself. For example, the recursive function Fact(N) can be defined as follows:

```

int fact (int N)
{
    if (N == 0)
        return (1);
    else
        return (N * fact (N-1));
}

```

It may be further noted that the function Fact is being called by itself but with parameter N being replaced by $N - 1$. A trace of Fact (N) for $N = 4$ is given in Fig.1.11

Example 7: Write a program that uses recursive function fact (N) to compute the factorial of a given number N.

Solution: A complete program to compute factorial of a number using recursion is given below.

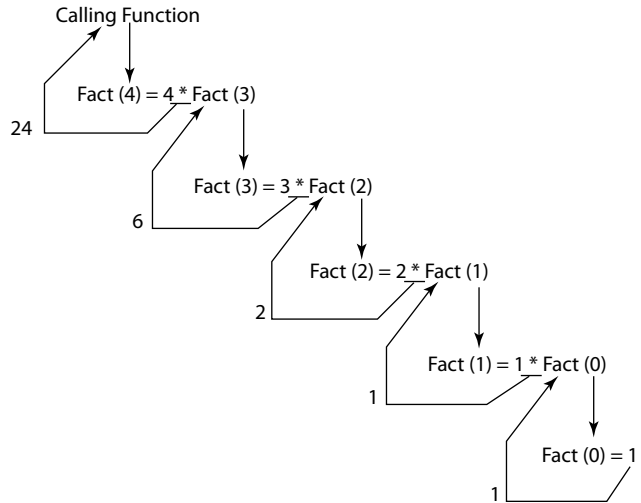


Fig. 1.11 Trace of Fact(4)

```

/* A program to compute factorial of a number using recursion */

```

```

#include<stdio.h>
#include<conio.h>
long int fact(int);
void main()
{
    int num, factorial;
    clrscr();
    printf ("\n This program computes the factorial of a number");
    printf ("\nEnter the number :");
    scanf ("%d",&num);
    factorial = fact(num);
    printf("\nThe factorial of %d is %d", num, factorial);
}

long int fact(int n)
{
    if (n==0)
        return(1);
    else
        return(n*fact(n-1));
}

```

Recursion can be used to write simple, short, and elegant programs. However, a recursive algorithm requires a basic condition that must terminate the recursive calls. The absence of this condition would

result in infinite recursive calls. For example, in function fact the condition ' $N = 0$ ' is the required terminating condition.

Note: Since recursion involves overheads such as CPU time and memory storage, it is suggested that recursion should be used with care.

Example 8: Write a function that computes x^y by using recursion.

Solution: We can use the property that x^y is simply a product of x and x^{y-1} . For example, $6^4 = 6 \times 6^3$. The recursive definition of x^y is given below:

$$\text{Power}(x, y) = \begin{cases} 1 & \text{if } y = 0 \\ X * \text{power}(x, y-1) & \text{otherwise} \end{cases}$$

The required function is given below :

```
// Function to compute x^y
int power(int x,int y)
{
    if (y==0)
        return (1);
    else
        return (x * power(x,y-1));
}
```

Note: The character '^' has been used to indicate power operator.

Example 9: Write a program that computes GCD (greatest common divisor) of given two numbers.

Solution: The greatest common divisor of two numbers can be computed by the algorithm given below:

Algorithm gcd()



Steps

1. Find the larger of the two numbers and store larger in x and smaller in y .
2. Divide the x by y and store the remainder in rem .
3. If rem is equal to zero, then the smaller number (y) is the required GCD and stop.

else store y in x and rem in y and repeat steps 2 and 3

The above given algorithm can be used to develop a recursive function called GCD which takes two arguments x and y and integer type and returns the required greatest common divisor.

```
// This function computes GCD
int gcd(int x,int y)
{
    int rem;
    rem = x%y;
```

```

if(rem==0)
return y;
else
    gcd(y,rem);
}

```

The required program that uses the function gcd(y) is given below :

```

/* This program computes GCD of two numbers with the help of a function
   gcd() .*/
#include<stdio.h>

int gcd(int,int);

void main()
{
    int x,y,ans;
    printf("\nEnter the integers whose gcd is to be found :");
    scanf ("%d %d",&x, &y);
    if (x>y)
        ans = gcd(x,y);
    else
        ans = gcd(y,x);
    printf("\nThe GCD is : %d", ans);
}

// This function computes the GCD
int gcd(int x,int y)
{
    int rem;
    rem = x%y;
    if(rem==0)
        return y;
    else
        gcd(y,rem);
}

```

Example 10: Write a program that generates the first n terms of Fibonacci sequence by recursion. The sequence is 0, 1, 1, 2, 3, 5, 8,

Solution: In a fibonacci series each term (except the first two) can be obtained by the sum of its two immediate predecessors. The recursive definition of this sequence is given below:

$$\text{Fib}(n) = \begin{cases} 0 & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ \text{Fib}(n-1) + \text{Fib}(n-2) & \text{if } n \geq 2 \end{cases}$$



Let us now write a program that uses a function `Fib()` to compute the first `n` terms of the series.

```
/* This program generates Fibonacci series*/

#include<stdio.h>
int fib(int);
void main()
{
    int n;
    int i,term;
    printf( "\nEnter the terms to be generated: " ) ;
    scanf ("%d", &n);
    for(i=1;i<=n;i++)
    {
        term = fib(i);
        printf("%d ", term);
    }
}

// Function to return a fibonacci term

int fib(int n)
{
    if (n==1)
        return 0;
    else
        if (n==2)
            return 1;
    else
        return(fib(n - 1) + fib(n - 2));
}
```

From above, we can summarize the characteristics of recursion as:

- There must exist at least a basic solution called terminating condition. This statement is processed without recursion.
- There must exist a method of reducing a problem into a smaller version of the problem
- A mechanism to throw back the smaller problem back to the recursive function

1.17.1 Types of Recursion

Depending upon the way the recursive function is called, the recursion process can be divided into two categories: direct recursion and mutual recursion. In *direct recursion* the recursive function is called from within itself. In *mutual recursion*, two recursive functions are involved and they mutually call each other as shown in Figure 1.12.

The mutual recursion is also called as *indirect recursion*. It may be noted that in direct recursion the recursive function `myFunc()` is calling itself, whereas in mutual recursion the two recursive functions: `myFunc()` and `yourFunc()` are mutually calling each other.

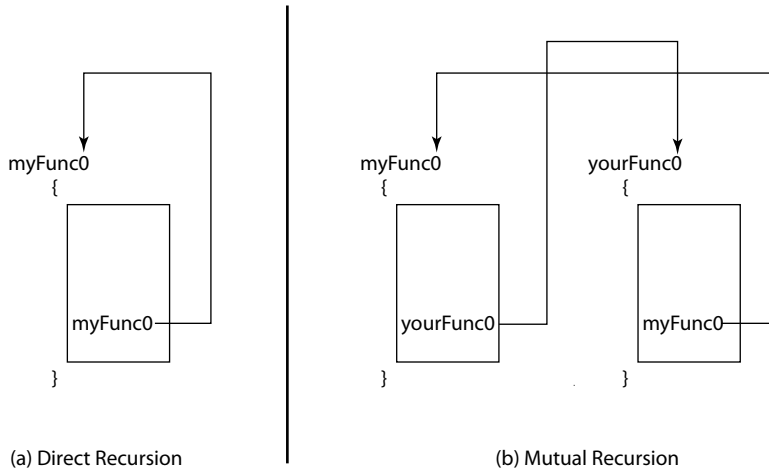


Fig. 1.12 Types of recursion

The direct recursion can be further divided into three categories—linear, binary and tail recursion. Though there are many more terms in vogue such as tree recursion etc, but we will discuss the popular forms of recursion as given below:

A. Linear Recursion

When a recursive function has a simple repetitive structure and calls itself once then it is called as linear recursion. For example, the recursive function called `power()`, given below, is a linear recursive function.

```
int power ( int x, int y)
{
    if ( y == 0 )
        return ( 1 );
    else
        return (x * power ( x, y-1));
}
```

It may be noted that it checks the terminating condition and thereafter performs the single recursive call to itself.

B. Binary Recursion

When a recursive function calls itself twice from within the function then it is called as binary recursion. For example, the `fib()`, given below, calls itself twice i.e. for `fib(n - 1)` and `fib(n - 2)`.

```
// Function to return a fibonacci term
int fib(int n)
{
    if (n==1)
        return 0;
    else
```

```

if (n==2)
    return 1;
else
    return(fib(n-1)+fib(n-2));
}

```

Another example of binary recursion is binary tree recursive operations, done for both left child and right child sub trees.

C. Tail Recursion

When a recursive call in a recursive function is the last call without any pending operation then the recursion is called tail recursion. Thus, the last result of the recursive call is the final result of the function.

Consider the function `fact()`, given below:

```

long int fact(int n)
{
    if (n==0)
        return(1);
    else
        return(n*fact(n-1));
}

```

This function is not tail recursive because the last statement is not recursive call but the multiplication operation: $n * \text{fact}(n - 1)$.

However, the following modified function `fact()` is tail recursive.

```

long fact(int N, int result)
{
    if (N == 1)
        return result;
    else
        return fact(N-1, N * result);    // Tail recursion
}

```

It may be noted that the last statement in the above function is a recursive call without any pending operation. Hence the function is tail recursive. Since the argument 'result' is acting as an accumulator, its initial value must be set to 1. Therefore, for computing factorial of a given number (say 5), the above function must be called with the following statement:

Factorial = fact (5, 1);

Example 11: Write a program that computes the factorial of a given number by using the above given tail recursive function `fact()`.

Solution: The required program is given below:

```

#include <stdio.h>
long fact(int N,int result)
{

```

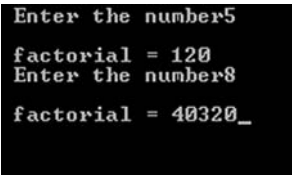


```

        if (N == 1)
            return result;
        else
            return fact(N-1, N * result);
    }
void main()
{
    long factorial;
    int num;
    printf( "\n Enter the number");
    scanf ("%d",&num);
    factorial = fact(num, 1);
    printf ("\n factorial = %u",factorial);
}

```

Sample outputs for the program are given below:



```

Enter the number5
factorial = 120
Enter the number8
factorial = 40320_

```

Example 12: Write a tail recursive function for computing x^y .

Solution: The required function *power* (*x*,*y*) is given below:



```

long power(int x,int y,long result)
{
    if (y == 0)
        return result;
    else
        return power(x, y-1, x * result);           // Tail recursion
}

```

Example 13: Write a program that tests the tail recursive function of Example 12 for given values of *x* and *y*.

Solution: The required program is given below:

```

# include <stdio.h>
long power(int x, int y,long result)
{
    if (y == 0)
        return result;
    else
        return power(x, y-1, x * result);           // Tail recursion
}
void main()
{

```



```

long pow;
int x, y;
printf("\n Enter the x,y");
scanf ("%d %d", &x,&y);
pow = power(x,y,1);
printf("\n x ^ y =%u",pow);
}

```

Sample output of the above program is given below:

```

Enter the x,y 4 3
x ^ y =64
Enter the x,y 5 3
x ^ y =125
Enter the x,y 2 7
x ^ y =128

```

Example 14: Write both normal and tail recursive versions of a function sum(N) that adds the first N integers. For example, sum(6) is computed as following:

$$\text{Sum}(6) = 1 + 2 + 3 + 4 + 5 + 6$$



Solution: The normal recursive function is given below:

```

long sum (int N)
{
    if (N == 1)
        return 1;
    else
        return N + sum (N-1);
}

```

The tail recursive version of function sum() is given below:

```

long sum (int N, int result)
{
    if (N == 1)
        return result;
    else
        return(sum (N-1, N + result));           // Tail recursion
}

```

Example 15: Write a complete program that takes the tail recursive version of the function sum() to compute the first N integers.

Solution: The required program is given below:



```

# include <stdio.h>

long sum (int N, int result)

```

```

{
    if (N == 1)
        return result;
    else
        return( sum (N-1, N + result));
}
void main()
{
    int N, total;
    printf("\n Enter N");
    scanf("%d",&N);
    total = sum(N,1);
    printf("\n sum =%d", total);
}

```

Sample output of the program is given below:

```

Enter N 5
sum =15
Enter N 10
sum =55

```

1.17.2 Tower of Hanoi

The problem of ‘Tower of Hanoi’, based on a legend of Vietnam, was first described by the mathematician François Édouard Anatole Lucas. Later on Henri de Parville also described the same problem based on an ancient Indian legend. It says that the dome of the Kashi Vishwanth Temple at Banaras marks the centre of the world. Under the dome there is ‘Tower of Bramah’.

The statement of the problem is that there are three needles like towers. On one tower there are 64 disks of decreasing sizes. The largest disk being at the bottom of the tower and the smallest at the top as shown in Figure 1.13.

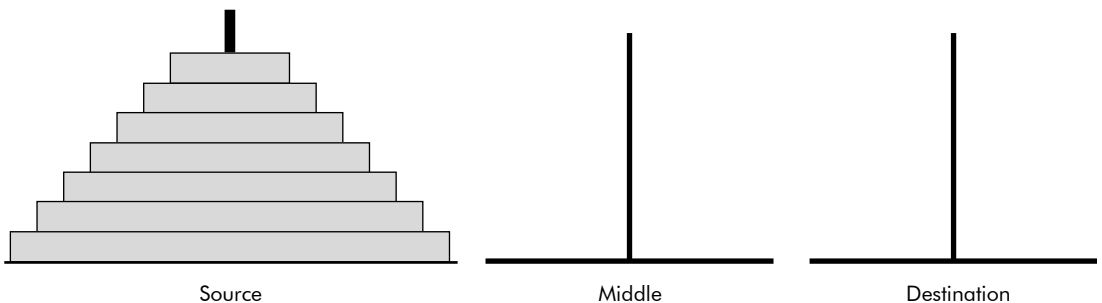


Fig. 1.13 The Tower of Hanoi

A group of monks or priests are moving the disks from source tower to destination tower as per the following rules:

1. At the beginning of the creation, the disks were placed in decreasing size on the source tower
2. The mandate is to move the disks to one of the other tower (say destination) with the help of a third tower called middle.
3. Only one disk can be moved at a time, taken from top of the tower
4. A disk can never be placed on a smaller disk.

The legend predicts that as soon as all the disks are moved to the destination tower, the world will come to an end, i.e., the priests will crumble into dust and the world will vanish.

This problem can be solved with the help of recursion. Assume there are N disks on the source tower. Since on the destination tower, the bottommost disk needs to be the bottom most of the source tower, it is necessary that we move $N - 1$ disks to the middle tower and thereafter move the largest (bottommost) disk to the destination tower as shown in Figure 1.14.

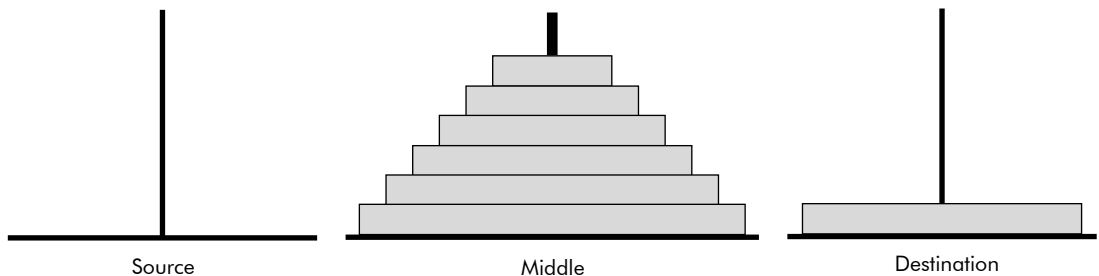


Fig. 1.14 Moving $N - 1$ disks to the middle tower and the bottom most to final tower

Now repeat this exercise with $N - 1$ disks using source tower as the middle tower, and middle tower playing the role of the source tower.

The above given steps can be written using a recursive algorithm as given below:

Algorithm **towOfHan** (N , source, dest, middle)

```
{
  if (  $N == 1$  )
    move a disk from source to dest
  else
    { towOfHan (  $N-1$ , source, middle, dest )
      move a disk from source to dest
      towOfHan (  $N-1$ , middle, dest, source )
    }
}
```

Example 16: Write a program that simulate the moves of Tower of Hanoi for a given number of disks, say N .

Solution: The above given algorithm towOfHan() has been used to code the required program which is given below:

```
// This program simulates the moves of tower of Hanoi
# include <stdio.h>
# include <conio.h>
```

```
void movDisk(int source, int dest);
void towOfHan(int numDisk, int source, int dest, int mid);

void main ()
{
    int numDisk;
    clrscr();
    printf("\nHow many disks do you want to move?");
    scanf ("%d", &numDisk);
    printf("\n Disk To move %d disks from 1 to 2 using 3 as middle disk:",
    numDisk);
    towOfHan(numDisk, 1, 2, 3);
}

void movDisk(int source, int dest)
{
    printf("\n move a disk from %d to %d", source, dest);
}

void towOfHan(int numDisk, int source, int dest, int mid)
{
    if (numDisk == 1)
        movDisk(source, dest);
    else
    {
        towOfHan(numDisk-1, source, mid, dest);
        movDisk(source, dest);
        towOfHan(numDisk-1, mid, dest, source);
    }
}
```

A sample output of the program is given below:

```
How many disks do you want to move?4
Disk To move 4 disks from 1 to 2 using 3 as middle disk:
move a disk from 1 to 3
move a disk from 1 to 2
move a disk from 3 to 2
move a disk from 1 to 3
move a disk from 2 to 1
move a disk from 2 to 3
move a disk from 1 to 3
move a disk from 1 to 2
move a disk from 3 to 2
move a disk from 3 to 1
move a disk from 2 to 1
move a disk from 3 to 2
move a disk from 1 to 3
move a disk from 1 to 2
move a disk from 3 to 2_
```

TEST YOUR SKILLS

1. What would be the output of following program?

```
#include <stdio.h>
main()
{
    printf("Compu");
    printf("ter");
}
```

Ans. The output would be: Computer

2. What would be the output of following program?

```
#include <stdio.h>
main()
{
    printf("Compu\buter");
}
```

Ans. The output would be: Computer

3. What would be the output of following program?

```
void main()
{
    int k = 30;
    /*
    printf("k = %d", k);
    */
    printf("value of k = %d", k);
}
```

Ans. The output would be: value of k = 30

4. Write four equivalent C statements each of which subtracts 1 from a variable called val.

Ans. The required statements are:

- val = val - 1;
- val--;
- --val;
- val -= 1;

5. Write the following expressions without using relational operators:

- i. if (val == 0)
- ii. if (val != 0)

Ans. Since in C 1 represents true and 0 represents false, the following statements can be used instead of above:

- i. if(!val)
- ii. if(val)

6. Write a single statement that increments the value of a variable val by 1 and then adds it to a variable called num.

Ans. The required statement is: num = num + (++val);

7. What will be the output of the following statements?

```
i. ch = 'A' + 10;
   printf ("%c", ch);
ii. int a = 5;
    printf ("%d", a < 10);
```

Ans. The output would be: (i) K, (ii) 1

8. What would be the output of following program?

```
#include <stdio.h>
main()
{
    int k = 10;
    k = ++k == 11;
    printf ("\n %d", k);
}
```

Ans. The output would be: 1

9. What would be the output of following program?

```
#include <stdio.h>
main()
{
    int i = 8 > 5 ? 100 : 150;
    printf ("\n %d", i);
}
```

Ans. The output would be: 100

10. What would be the output of the following program?

```
enum furniture { Chair,
                Table,
                Bed,
                Almirah
};
#include <stdio.h>
main()
{
    printf ("\n %d %d %d %d", Chair, Table, Bed, Almirah);
}
```

Ans. The output would be: 0 1 2 3

11. What would be the output of the following program?

```
#include <stdio.h>
int first (int, int);
int second (int);
void main()
{
    int x = 10, y = 5, z;
    z = first (x,y);
    printf ("\n z = %d", z);
}
```

```

int first (int x, int y)
{
    return (x + second (y));
}

int second (int y)
{
    return ++y;
}

```

Ans. The output would be: 16

EXERCISES

1. Define the terms: token, keyword, identifier, variable, constant and const correct.
2. What is meant by basic data types? Explain in brief.
3. What would be the appropriate data type for the following?
 - Length of a cloth
 - Age of a student
 - An exclamation mark
4. What is meant by a backslash character? What is the utility of these characters in a program?
5. What would be the output of the following program?

```

#include <stdio.h>
{
    int k;
    printf ("k = %d", k );
}

```

- i. an error ii. unpredictable iii. 34216 iv. 0
6. Write a program that inputs a value in inches and prints in centimeters.
 7. Write a program that converts degree Celsius temperature into its degree Fahrenheit equivalent using the following formula:

$$F = 9/5 C + 32$$
 8. What is the purpose of 'else' clause in an 'if' statement of C?
 9. Write a program that computes a^b where a and b are of real and integer types, respectively.
 10. Describe the function of break and continue statements in C.
 11. Write a program that reverses and sums the digits of an integer number.
 12. Write a program that prints the following output on the screen.

```

A
B B
C C C
D D D D
E E E E E

```

13. Write a program that implements a menu of the following type:

Menu	
Option1	'a'
Option2	'b'
Option3	'c'
Option4	'd'

Enter your choice:

In response to user's selection, it prints the following messages:

Option	Message
1	'One is alone'
2	'Two is a company'
3	'Three is a crowd'
4	'Quitting'

14. Define the terms: arrays, subscript, subscripted variables, and strings.
 15. Write a program that removes duplicates from a list.
 16. Write a program that computes the sum of diagonal elements of a square matrix.
 17. Write a C structure for the record structure given in figure shown below.

Title	Author Name	Publishers		Cost	Edition
		Name	Address		

18. Define the terms: structure and member variables.
 19. Using the structure of following figure, write a program that maintains a list of computer books available in the library.
 20. What is meant by enumerated data types? Explain the concept with the help of suitable examples.
 21. What is meant by a function prototype and why is it needed?
 22. Define recursion and compare it with iteration.
 23. What are the types of recursion?
 24. Explain tail recursion with the help of a suitable example.
 25. What are the basic requirements of a recursive algorithm?
 26. Write a program that uses recursion to add first N natural numbers
 27. Explain the problem of Tower of Hanoi. Write a recursive function that moves N disks from a tower called 'first' to tower 'last' using a tower 'mid'.

Data Structures and Algorithms: An Introduction

2

CHAPTER

CHAPTER OUTLINE

- 2.1 Overview
- 2.2 Concept of Data Structures
- 2.3 Design of a Suitable Algorithm
- 2.4 Algorithm Analysis

2.1 OVERVIEW

A normal desktop computer is based on *Von Neumann architecture*. It is also called a *stored program computer* because the program's instructions and its associated data are stored in the same memory as shown in Figure 2.1.

The program instructions tell the system what operation is to be performed and on what data.

The above model is based on the fact that every practical problem that needs to be solved must have some associated data. Let us consider a situation where a list of numbers is to be sorted in ascending order. This problem obviously has the following two parts:

- (1) The list: say 9, 8, 1, -1, 4, 2, 6, 15, 3
- (2) A procedure to sort the given list of numbers.

It may be noted that *elements of list* and the *procedure* represent the data and program of the problem, respectively.

Now, the efficiency of the program would primarily depend upon the organization of data. When the data is poorly organized, then the program cannot efficiently access it. For instance, the time taken to open the lock of your house depends upon where you have kept the key to it: in your hand, in front pocket, in the ticket pocket of your pant or the inner pocket of your briefcase, etc. There

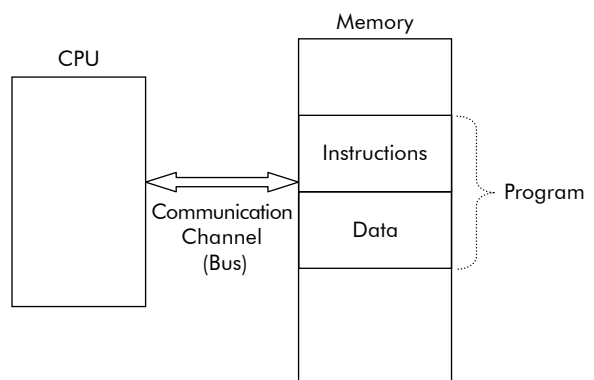


Fig. 2.1 The stored program computer

may be a variety of motives for selecting one or the other of these places to keep the key. Key in the hand or in the front pocket would be quickly accessible as compared to other places whereas the key placed in the inner pocket of briefcase is comparatively much secured. For similar reasons, it is also necessary to choose a suitable representation or structure for data so that a program can efficiently access it.

2.2 CONCEPT OF DATA STRUCTURES

In fact the data, belonging to a problem, must be organized for the following basic reasons:

- **Suitable representation:** Data should be stored in such a manner that it represents the various components of the problem.

For example, the list of numbers can be stored in physical continuous locations. Arrays in 'C' can be used for the required representation as shown in Figure 2.2.

	0	1	2	3	4	5	6	7	8
List	9	8	1	-1	4	2	6	15	3

Fig. 2.2 List stored in physical continuous locations

- **Ease of retrieval:** The data should be stored in such a manner that the program can easily access it. As an array is an *index-value* pair, a data item stored in the array (see Figure 2.2) can be very easily accessed through its index. For example, `List [5]` will provide the data item stored at 5th location, i.e. '2'. From this point of view, the array is a *random-access* structure.
- **Operations allowed:** The operations needed to be performed on the data must be allowed by the representation.

For example, *component by component* operations can be done on the data items stored in the array. Consider the data stored in `List` of Figure 2.2. For the purpose of sorting, the smallest number must come to the head of the list. This shall require the following two operations:

- (1) Visit every element (component) in the list to search the smallest and its position ('-1' at location 3).
- (2) Exchange the smallest with the number stored at the head of the list (exchange '-1' stored at location 3 with '9' stored at location 0). This can be easily done by using a temporary location (say `Temp`) as shown below:

```
Temp    = List[0];
List[0] = List[3];
List[3] = Temp;
```

The new contents of the `List` are as shown in Figure 2.3.

	0	1	2	3	4	5	6	7	8
List	-1	8	1	9	4	2	6	15	3

Fig. 2.3 Contents of the List after exchange operation

The operations 1 and 2, given above, can be repeated on the unsorted part of the List to obtain the required sorted List.

From above discussions, it can be concluded that an array provides a suitable representation for the storage of a list of numbers. Such a representation is also called a *data structure*. **Data structure** is a logical organization of a set of data items that collectively describe an object. It can be manipulated by a program. For example, array is a suitable data structure for a list of data items.

It may be observed that if the array called 'List' represents the *data structure* then the operations carried out in steps 1 and 2 define the *algorithm* needed to solve the problem. The overall performance of a program depends upon the following two factors:

- (1) Choice of right data structures for the given problem.
- (2) Design of a suitable algorithm to work upon the chosen data structures.

2.2.1 Choice of Right Data Structures

An important step of problem solving is to select the data structure wherein the data will be stored. The program or algorithm would then work on this data structure and give the results.

A data structure can either be selected from the *built-in* data structures offered by the language or by designing a new one. Built-in data structures in 'C' are arrays, structures, unions, files, pointers, etc.

Let us consider a situation where it is required to create a merit list of 60 students. These students appeared in a class examination. The data of a student consists of his *name*, *roll number*, *marks*, *percentage*, and *grade*. Out of many possibilities, the following two choices of data structures are worth considering:

Choice I: The various data items related to a student are: name, roll number, marks, percentage, and grade. The total number of students is 60. Therefore, the simplest choice would be to have as many parallel lists as there are data items to be represented. The arrangement is shown Figure 2.4.

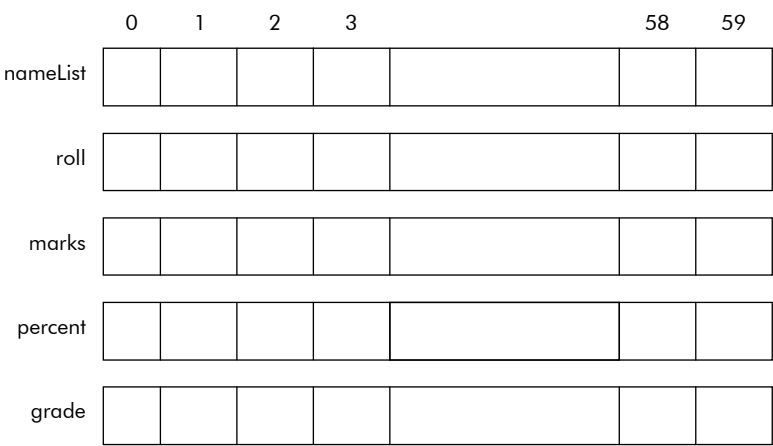


Fig. 2.4 Parallel lists for student data

An equivalent declaration for data structure of Figure 2.4 is given below:

```
char  nameList[60][15];
int    roll[60];
int    marks[60];
float  percent[60];
char   grade[60];
```

Now in the above declaration, every list has to be considered as individual. The generation of merit list would require the list to be sorted. The sorting process would involve a large number of exchange operations. For each list, separate exchange operations would have to be written. For example, the exchange between data of *i*th and *j*th student would require the following operations:

```
char  tempName[15],chTemp;
int  i,j, temp;
float fTemp;

strcpy (tempName, nameList[i]);
strcpy (nameList[i], nameList[j]);
strcpy (nameList[j], tempName);

temp = roll[i];
roll[i] = roll[j];
roll[j] = temp;

temp = marks[i];
marks[i] = marks[j];
marks[j] = temp;

fTemp = percent[i];
percent[i] = percent[j];
percent[j] = fTemp;

chTemp = grade[i];
grade[i] = grade[j];
grade[j] = chTemp;
```

Though the above code may work, but it is not only a clumsy way of representing the data but would require a lot of typing effort also. In fact, as many as *fifteen statements* are required to exchange the data of *i*th student with that of *j*th student.

Choice II: From the built-in data structures, the `struct` can be selected to represent the data of a student as shown below:

```
struct student {
    char  name[15];
    int    roll;
    int    marks;
    float  percent;
    char   grade;
};
```

Similarly, from built-in data structures, an *array* called **studList** can be selected to represent a list. However, in this case each location has to be of type *student* which itself is a structure. The required declaration is given below:

```
struct student studList [60];
```

It may be noted that the above given array of structures is a user-defined data structure constructed from two built-in data structures: *struct* and *array*. Now in this case, exchange between data of *i*th and *j*th student would require the following operations:

```
int i,j;
struct student temp;
temp = studList[i];
studList [i] = studList [j];
studList [j] = temp;
```

Thus, only **three statements** are required to exchange the data of *i*th student with that of *j*th student.

A comparison of the two choices discussed above establishes that Choice II is definitely better than Choice I as far as the selection of data structures for the given problem is concerned. The number of lines of code of Choice II is definitely less than the Choice I. Moreover, the code of Choice II is comparatively easy and elegant.

From the above discussion, it can be concluded that choice of right data structures is of paramount importance from the point of view of representing the data belonging to a problem.

2.2.2 Types of Data Structures

A large number of data structure are available to programmers that can be usefully employed in a program. The data structures can be broadly classified into two categories: *linear* and *non-linear*.

(1) Linear data structures: These data structures represent a sequence of items. The elements follow a linear ordering. For instance, one-dimensional arrays and linked lists are linear data structures. These can be used to model objects like queues, stacks, chains (linked lists), etc., as shown in Figure 2.5.

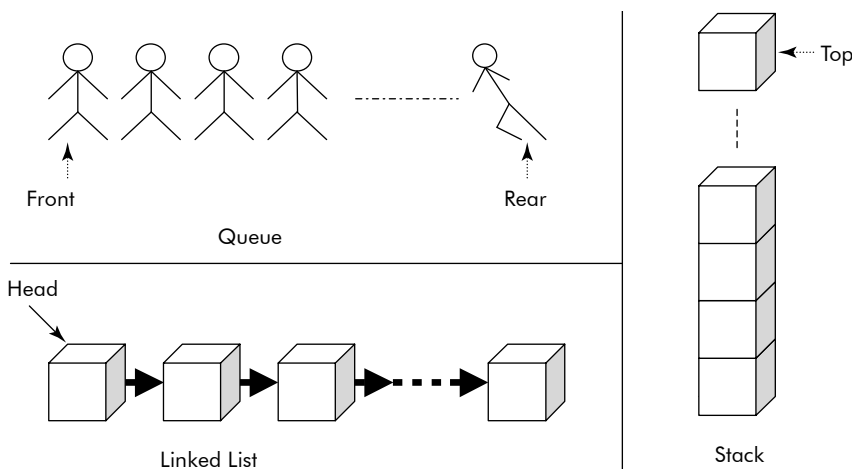


Fig. 2.5 Linear data structures

A linear data structure has following two properties:

- (1) Each element is 'followed by' at most one other element.
- (2) No two elements are 'followed by' the same element.

For example, in the list of Figure 2.3, the 2nd element ('1') is followed by 3rd element ('9') and so on.

(2) **Non-linear data structures:** These data structures represent objects which are not in sequence but are distributed in a plane. For instance, two-dimensional arrays, graphs, trees, etc. are non-linear data structures. These can be used to model objects like tables, networks and hierarchy, as shown in Figure 2.6.

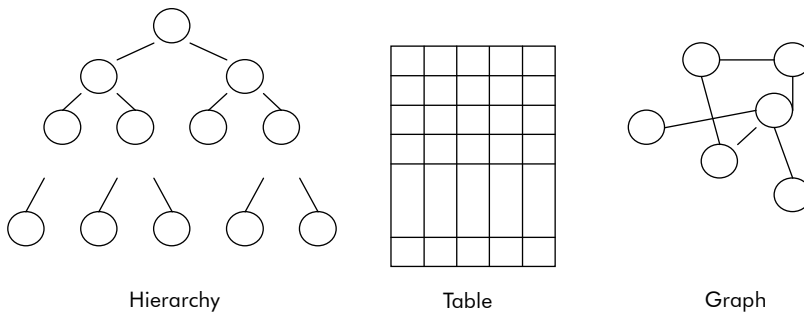


Fig. 2.6 Non-linear data structures

The tree structure does not follow the first property of linear data structures and the graph does not follow any property of the linear data structures. A detailed discussion on this issue is given later in the book.

2.2.3 Basic Terminology Related with Data Structures

The basic terms connected with the data structures are:

- (1) **Data element:** It is the most fundamental level of data. For example, a *Date* is made up of three parts: Day, Month, and Year as shown in Figure 2.7. The Date is called a data object and DD, MM, and YY are called the data elements or data items.

It may be noted that the data object denotes a group and the data elements denote atomic entities.

- (2) **Primitive data types:** It is type of data that a variable may hold. For example, *int*, *float*, *char*, etc. are data types.
- (3) **Data object:** It is a set of elements. For example, the set of fruit 'F' refers to the set $F = \{\text{'mango'}, \text{'orange'}, \text{'grapes'}, \dots\}$. Similarly, a set of counting numbers 'C' refers to the set $C = \{1, 2, 3, 4, \dots\}$.

- (4) **Data structure:** It is a set of related elements.

A set of operations that are allowed on the elements of the data structure is defined.

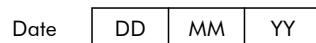


Fig. 2.7 Data elements of data object Date

- (5) **Constant:** It is the memory location that does not change its contents during the execution of a program.
- (6) **Variable:** It is the most fundamental aspect of any computer language and can be defined as a location in the memory wherein a value can be manipulated, i.e., stored, accessed, and modified.

2.3 DESIGN OF A SUITABLE ALGORITHM

After selecting the data structures, an algorithm, based on the data structure, needs to be designed. The set of rules that define how a particular problem can be solved in a finite sequence of steps is called an *algorithm*. An algorithm written in a computer language is called a *program*. An **algorithm** can be defined as a finite sequence of instructions, each of which has a clear meaning and can be executed with a finite amount of effort in finite time.

A computer is simply a machine with no intelligence. In fact, the processor of the computer is a passive device that can only execute instructions. Therefore, the computer must be instructed in unambiguous steps such that it follows them one after the other to perform the given task. Thus, it becomes programmer's duty to precisely define his problem in the form of an algorithm, i.e., unambiguous steps written in simple English. The desirable **features** of an algorithm are:

- Each step of the algorithm should be simple.
- It should be unambiguous in the sense that the logic should be *crisp* and *clear*.
- It should be effective, i.e., it must lead to a unique solution of the problem.
- It must end in a finite number of steps.
- It should be as efficient as possible.

Accordingly, an algorithm has the following characteristics:

- (1) **Input:** This part of the algorithm reads the data of the given problem.
- (2) **Process:** This part of the algorithm does the required computations in simple steps.
- (3) **Finiteness:** The algorithm must come to an end after a finite number of steps.
- (4) **Effectiveness:** Every step of the algorithm must be accurate and precise. It should also be executable within a definite period of time on the target machine.
- (5) **Output:** It must produce the desired output.

Thus, we can say that the algorithm tells the computer when to read data, compute on it, and write the answers.

2.3.1 How to Develop an Algorithm?

In order to develop an algorithm, the problem has to be properly analysed. The programmer has to understand the problem and come out with a method or logic that can solve the problem. The logic must be simple and adaptable on a computer. The following steps can be suggested for developing an algorithm:

- (1) Understand the problem.
- (2) Identify the output of the problem.
- (3) Identify inputs required by the problem and choose the associated data structures.
- (4) Design a logic that will produce the desired output from the given inputs.
- (5) Test the algorithm for different sets of input data.
- (6) Repeat steps 1 to 5 till the algorithm produces the desired results for all types of input and rules.

While solving problems with the help of computers, the following facts should be kept in mind:

“A location in the main memory has the property: *Writing-in* is destructive and *Reading-out* is non-destructive”. In simple words, we can say that if a value is assigned to a memory location, then its

previous contents will be lost. However, if we use the value of a memory location then the contents of the memory do not change, i.e., it remains intact. For example, in the following statement, the value of variables B and C are being used and their sum is stored in variable A

$$A = B + C$$

where, the symbol '=' is an assignment operator.

If previous values of A, B, C were 15, 45, 20, respectively then after execution of the above statement, the new contents of A, B, C would be 65, 45, and 20, respectively.

Consider the following statement:

$$\text{Total} = \text{Total} + 100;$$

The abnormal looking statement is very much normal as far as computer memory is concerned. The statement reveals the following fact:

"Old value of variable 'Total' is being added to an integer 100 and the result is being stored as new value in the variable 'Total', i.e., replacing its old contents with the new contents."

After the execution of the statement with old contents of 'Total' (say 24), the new contents of 'Total' will be 124.

2.3.1.1 Identifying Inputs and Outputs It is very important to identify inputs and outputs of an algorithm. The inputs to an algorithm mean the data to be provided to the target program through input devices such as keyboard, mouse, input file, etc. The output of the algorithm means the final data to be generated for the output devices such as monitor, printer, output file, etc.

The inputs and outputs have to be identified at the beginning of the program development. The most important decision is to differentiate between *major* and *minor* inputs and outputs. The minor inputs are sometimes omitted at the algorithm development stage. The other important issue is to find at which part of the program the inputs would be required and the output generated.

Example 1: Let us reconsider the problem of sorting a given list of numbers. In this case, the required **inputs** are:

- (1) List of numbers
- (2) Size of list
- (3) Type of numbers to be stored in the list
- (4) In which order to be sorted

The **outputs** are:

- (1) The sorted list
- (2) Message for asking the size of the list
- (3) Message for displaying the output list

The messages are the minor outputs.

After identifying the inputs and the outputs, the additional information must be specified before writing the final algorithm.

Input specifications

- In what order and format, the input values will be read?
- What are the lower and upper limits of the input values? For example, the size of the list should not be less than zero and its type has to be necessarily an integer.
- When to know that there is no more input to be read, i.e., what marks the end of the list?

Output specifications

- In what order and format the outputs values will be produced?
- What types of values to be produced with what spacing i.e., decimal spacing, significant digits, etc.
- What main headings and column headings to be printed on the output page.

2.3.2 Stepwise Refinement

We know that if there is even a small logical error in a program, the compiler cannot detect it. The year 2000 problem (Y2K problem) is a glaring example of this type. Thus, an algorithm requires a very careful design, especially pertaining to a large and complex problem.

In order to design an algorithm, the programmer must adopt some methodological approach such as modular/stepwise refinement approach of structured programming techniques. Stepwise refinement uses the “divide and conquer” strategy. In this technique, the problem is divided into a number of steps. Each step is described by a sub-problem which is smaller and simpler as compared to the original program. The sub-problem is again divided into still smaller steps and so on. The refinement continues till we reach to a stage where the steps become fairly simple to implement.

Example 2: Suppose the following *table* is to be printed on the screen.

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
10	20	30	40	50	60	70	80	90	100

The problem at hand is “How to print the *table*”. The first version of the algorithm is given below:

```
Algorithm Print_Table Ver1.
{
    Step 1. Print table
}
```

When we look at the *table*, we find that the table is nothing but a collection of *rows* and we have to print only 10 rows. So, the problem has been reduced to printing a row 10 times. The second refined version of the algorithm is given below:

```
Algorithm Print_Table Ver2.
{
    Step
    1. Row = 1
    2. Print the row
    3. Row = Row + 1
    4. If Row <= 10 repeat steps 2 and 3
    5. Stop
}
```

Now our focus is on “how to print a *row*”. A closer look at the problem suggests that a row is nothing but a collection of *columns* and we have to print only 10 columns for each row. The third refined version of the algorithm is given below:

```
Algorithm Print_Table Ver3.
{
    Step
    1. Row = 1
    2. Col = 1
        2.1 Print the col.
        2.2 Col = Col + 1
        2.3 If Col <= 10 repeat steps 2.1 and 2.2
        2.4 Go to a new line.
    3. Row = Row + 1
    4. If Row <=10 repeat steps 2 and 3
    5. Stop.
}
```

Now our concern is “how to print a *column*”. We find that it is very simple, i.e., we get the value to be printed by multiplying the values contained in the variables *Row* and *Col*. The final version of the refined algorithm is given below:

```
Algorithm Print_Table Ver4.
{
    Step
    1. Row = 1
    2. Col = 1
        2.1 ColVal = Row * Col
        2.2 Print the ColVal.
        2.3 Col = Col + 1
        2.4 If Col <= 10 repeat steps 2.1 to 2.3
        2.5 Go to a new line.
    3. Row = Row + 1
    4. If Row <=10 repeat steps 2 and 3
    5. Stop.
}
```

The programmer stops at this stage where he finds that the step(s) is trivial and can be easily implemented.

2.3.3 Using Control Structures

Every language supports sequence control structures. These structures can be easily used towards writing a program for a given problem.

(1) Sequence structure: The order in which the statements or statement blocks are written within a pair of curly braces ({ , }) in an algorithm are sequentially executed by the computer.

Example 3: Write an algorithm that computes the area of a triangle.

Solution: The solution is trivial. The inputs are base and height. The following formula will be used for the computation of the area of triangle:

$$\text{Area} = \frac{1}{2} * \text{base} * \text{height}$$

where area is the output to be displayed. The simple sequence structure required for this problem is given below:

```
Algorithm AreaTriangle ()
{
    Step
    1. Read base, height;
    2. Area = 0.5 * base * height;
    3. Print Area;
    4. Stop
}
```

It may be noted that in a sequence structure:

- (i) The statements are executed sequentially.
- (ii) The statements are executed one at a time.
- (iii) Each statement is executed only once.
- (iv) No statement is repeated.
- (v) The execution of the last statement marks the end of the sequential structure.

(2) Selection (conditional control) structure: The selection or decision structure is used to decide whether a statement or a block is to be executed or not. The decision is made by testing whether a condition is true or false. The if-else ladder structure can be used for this purpose.

Example 4: Write an algorithm that prints the biggest of given three unique numbers.

Solution:

Input: Three numbers: Num1, Num2, and Num3

Type of numbers: integer.

Output: Display the biggest number and a message in this regard.

```
Algorithm Biggest ()
{
    Step
    1. Read Num1, Num2, Num3;
    2. If (Num1 > Num2)
        2.1 If (Num1 > Num3)
            Print "Num1 is the largest"
        Else
            Print "Num3 is the largest"
    Else
        2.2 If (Num2 > Num3)
            Print "Num2 is the largest"
        Else
            Print "Num3 is the largest"
    3. Stop
}
```

(3) Iteration: The iteration structure is used to repeatedly execute a statement or a block of statements. The loop is repeatedly executed until a certain specified condition is satisfied.

Example 5: Write an algorithm that finds the largest number from a given list of numbers.

Solution:

Input: List of numbers, size of list, prompt for input.

Type of numbers: integer.

Output: Display the largest number and a message in this regard.

The most *basic solution* to this problem is that if there is only one number in the list then the number is the largest. Therefore, the first number of the list would be taken as largest. The largest would then be compared with the rest of the numbers in the list. If a number is found larger than the current largest, then it is stored in the largest. This process is repeated till the end of the list. For iteration, *For loop* can be employed.

The concerned algorithm is given below:

Algorithm Largest

```
{
    Step
    1. Prompt "Enter the size of the list";
    2. Read N;
    3. Prompt "Enter a Number";
    4. Read Num;
    5. Largest = Num;    /* The first number is the largest*/
    6. For (I = 2 to N)
        {
            6.1 Prompt "Enter a Number"
            6.2 Read Num
            6.3 If (Num > Largest) Largest = Num
        }
    7. Prompt "The largest Number ="
    8. Print Largest;
    9. Stop
}
```

It may be noted here that if a loop terminates after a number of iterations, then the loop is called a **finite** loop. On the other hand if a loop does not terminate at all and keeps on executing endlessly, then it is called an **infinite** loop.

(4) Use of accumulators and counters: *Accumulator* is a variable that is generally used to compute sum or product of a series. For example, the sum of the following list of numbers can be computed by initializing a variable (say '*Total*') to zero.

4	8	9	40	32	12	21	1	9
---	---	---	----	----	----	----	---	---

The 1st number of the list is added to *Total* and then the 2nd, 3rd and so on till the end of the list. Finally, the variable *Total* would get the accumulated sum of the list given above.

Example 6: Write an algorithm that reads a list of 50 integer values and prints the total of the list.

Solution: Let us assume that a variable called 'sum' would be used as an accumulator. A variable called Num would be used to read a number from the input list one at a time. For loop will be employed to do the iteration.

The Algorithm is given below:

Algorithm SumList

```
{
    Step
    1. Sum =0;
    2. For (I = 1 to 50)
        {
            2.1 Read Num;
            2.2 Sum = Sum + Num; /* The accumulation of the list */
        }
    3. Prompt "The sum of the list =";
    4. Print Sum;
    5. Stop
}
```

It may be noted that for product of a series, the accumulator is initialized to 1.

Counter is a variable used to count number of operations/actions performed in a loop. It may also be used to count number of members present in a list or a set of items.

The counter variable is generally initialized by 0. On the occurrence of an event/operation, the variable is incremented by 1. At the end of the counting process, the variable contains the required count.

Example 7: Write an algorithm that finds the number of zeros present in a given list of N numbers.

Solution: A counter variable called Num_of_Zeros would be used to count the number of zeros present in a given list of numbers. A variable called Num would be used to read a number from the input list one at a time. For loop will be employed to do the iteration.

The Algorithm is given below:

Algorithm Count_zeros()

```
{
    Step
    1. Num_of_zeros = 0;
    2. For (I = 1 to N)
        {
            2.1 Read Num;
            2.2 If (Num == 0)
                Num_of_zeros = Num_of_zeros +1;
        }
    3. Prompt "The number of zero elements =";
    4. Print Num_of_zeros;
    5. Stop
}
```

Example 8: Write a program that computes the sum of first N multiples of an integer called 'K', i.e.

$$\text{sumSeries} = 1 \times K + 2 \times K + 3 \times K + \dots \dots \dots N \times K$$

Solution: An accumulator called sumSeries would be used to compute the required sum of the series. As the number of steps are known (i.e., N), 'For loop' is the most appropriate for the required computation. The index of the loop will act as the counter.

Input: N, K

Output: sumSeries

The program is given below:

```
/* This program computes the sum of a series */
#include <stdio.h>
void main()
{
    int sumSeries, i;
    int N; /* Number of terms */
    int K; /* The integer multiple*/
    printf("\n Enter Number of terms and Integer Multiple");
    scanf ("%d %d", &N,&K);
    sumSeries = 0; /*Accumulator initialized */
    For (i=1; i<=N; i++)
    {
        sumSeries = sumSeries + i*K;
    }
    printf ("\n The sum of first %d terms = %d", N, sumSeries);
}
```

In fact, it can be concluded that irrespective of the programming paradigms, a general algorithm is designed based on following constructs:

- (1) **Sequence structure:** Statement after statements are written in sequence in a program.
- (2) **Selection:** Based on a decision (i.e., if-else), a section of statements is executed.
- (3) **Iteration:** A group of statements is executed within a loop.
- (4) **Function call:** At a certain point in program, the statement may call a function to perform a particular task.

It is the responsibility of the programmer to use the right construct at right place in the algorithm so that an efficient program is produced. A discussion on analysis of algorithms is given in next section.

2.4 ALGORITHM ANALYSIS

A critical look at the above discussion indicates that the data structures and algorithms are related to each other and have to be studied together. For a given problem, an algorithm is developed to solve it but the algorithm needs to operate on data. Unless the data is stored in a suitable data structure, the design of algorithm remains incomplete. So, both the data structures and algorithm are important for program development.

In fact, an algorithm requires following two *resources*:

- (1) **Memory space:** Space occupied by program code and the associated data structures.
- (2) **CPU time:** Time spent by the algorithm to solve the problem.

Therefore, an algorithm can be analyzed on two accounts: *space* and *time*. We need to develop an algorithm that efficiently makes use of these resources. But then, there are many ways to attack a problem and hence many algorithms at hand to choose from. Obviously, having decided the data structures, *execution time* can be a major criterion for selection of an algorithm.

As a first step, it is understandable that an algorithm would be fast for a small input data. For example, the search for an item in a list of 10 elements would definitely be faster as compared to searching the same item within a list of 1,000 elements. Therefore, the execution time T of an algorithm has to be dependent upon N , the size of input data. Now the question is how to measure the execution time T . Can we exactly compute the execution time? Perhaps not because there are many factors that govern the actual execution of program in a computer system. Some of the important factors are listed below:

- The speed of the computer system
- The structure of the program
- Quality of the compiler that has compiled the program
- The current load on the computer system.

Thus, it is better that we compare the algorithms based on their relative time instead of actual execution time.

The amount of time taken for completion of an algorithm is called **time complexity** of the algorithm. It is a theoretical estimate that measures the *growth rate* of the algorithm $T(n)$ for large number of n , the input size. The time complexity is measured in terms of Big-Oh notation. A brief discussion on this notation is given in next section.

2.4.1 Big-Oh Notation

The **Big-Oh** notation defines that for a large number of input ' n ', the growth rate of an algorithm $T(n)$ is of the order of a function ' g ' of ' n ' as indicated by the following relation:

$$T(n) = O(g(n)).$$

The term '*of the order*' means that $T(n)$ is less than a constant multiple of $g(n)$ for $n \geq n_0$. Therefore, for a constant ' c ', the relation can be defined as given below:

$$T(n) \leq c \cdot g(n) \quad \text{where } c > 0. \quad (2.1)$$

From above relation, we can say that for a large value of n , the function ' g ' provides an upper bound on the growth rate ' T '.

Let us now consider the various constructs used by a programmer to write an algorithm.

(1) Simple statement: Let us assume that a statement takes a unit time to execute, i.e., 1. Thus, $T(n) = 1$ for a simple statement. To satisfy the Big-Oh condition, the above relation can be rewritten as shown below:

$$T(n) \leq 1 \cdot 1 \quad \text{where } c = 1 \text{ and } n_0 = 0.$$

Comparing the above relation with relation 2.1, we get $g(n) = 1$. Therefore, the above relation can be expressed as:

$$T(n) = O(1)$$

(2) Sequence structure: The execution time of sequence structure is equal to the sum of execution time of individual statements present within the sequence structure.

Consider the following algorithm. It consists of a sequence of statements 1 to 4:

```

Algorithm AreaTriangle
{
    Step
    1. Read base, height;
    2. Area = 0.5 * base * height;
    3. Print Area;
    4. Stop
}

```

The above algorithm takes four steps to complete.

Thus, $T(n) = 4$.

To satisfy the relation 2.1, the above relation can be rewritten as:

$$T(n) \leq 4 \cdot 1 \quad \text{where } c = 4 \text{ and } n_0 = 0.$$

Comparing the above relation with relation 2.1, we get $g(n) = 1$. Therefore, the above relation can be expressed as:

$$T(n) = O(1)$$

We can say that the time complexity of above algorithm is $O(1)$, i.e., of the order 1.

(3) The loop structure: The loop structure iteratively executes statements written within its bound. If a loop executes for N iterations, it contains only simple statements.

Consider the algorithm `Count_zeros` of Example 7 which contains a loop. This algorithm takes the following steps for its completion:

```

No. of simple steps           = 4 (Steps 1, 3, 4, 5)
No. of Loops of steps 1 to N   = 1
No. of statements within the loop = 3 (1 comparison, 1 addition, 1 assignment)

```

$$\text{Thus, } T(n) = 3 \cdot N + 4$$

Now for $N \geq 4$, we can say that

$$3 \cdot N + 4 \leq 3 \cdot N + N \leq 4N$$

Therefore, we can say that

$$T(n) \leq 4N \quad \text{where } c = 4, n_0 \geq 4$$

$$T(n) = O(N)$$

Hence, the given algorithm has time complexity of the order of N , i.e., $O(N)$. This indicates that the term 4 has negligible contribution in the expression: $3 \cdot N + 4$.

Note: In case of nested loops, the time complexity depends upon the counters of both outer and inner loops. Consider the nested loops given below. This program segment contains 'S', a sequence of statements within nested loops I and J .

```

For (I = 1; I <= N; I++)
{
    For (J = 1; J <= N; J++)
    {
        S;
    }
}

```


It may be noted that for each iteration of I loop, the J loop executes N times. Therefore, for N iterations of I loop, the J loop would execute $N \times N = N^2$ times. Accordingly, the statement S would also execute N^2 times.

$$\text{Thus, } T(n) = N^2$$

To satisfy the relation 2.1, the above relation can be rewritten as shown below:

$$T(n) \leq 1 \times N^2 \quad \text{where } c = 1 \text{ and } n_0 = 0.$$

Comparing the above relation with relation 2.1, we get $g(n) = N^2$. Therefore, the above relation can be expressed as:

$$T(n) = O(N^2)$$

Hence, the given nested loop has time complexity of the order of N^2 , i.e., $O(N^2)$.

Example 9: Compute the time complexity of the following nested loop:

```
For (I = 1; I <= N; I++)
{
    For (J = 1; J <= I; J++)
    {
        S;
    }
}
```

Solution: In this nested loop structure, for an iteration of I loop, the J loop executes in an increasing order from 1 to N as given below:

$$1 + 2 + 3 + 4 + \dots + (N - 1) + N = N(N + 1)/2 = N^2/2 + N/2.$$

$$\text{Thus, } T(n) = N^2/2 + N/2.$$

Now for $N^2 \geq N/2$, we can say that:

$$N^2/2 + N/2 \leq N^2/2 + N^2 \leq 3 N^2/2.$$

Therefore, we can say that

$$T(n) \leq 3 N^2/2 \quad \text{where, } c = 3/2, n_0 = 1/2$$

$$T(n) = O(N^2)$$

Hence, the given algorithm has time complexity of the order of N^2 , i.e., $O(N^2)$. This indicates that the term $N/2$ has negligible contribution in the expression $N^2/2 + N/2$.

(4) If-then-else structure: In the if-then-else structure, the then and else parts are considered separately. Consider the if-then-else structure given in Figure 2.8.

Let us assume that the 'statement 1' of then part has the time complexity T_1 and that 'statement 2' of else part has the time

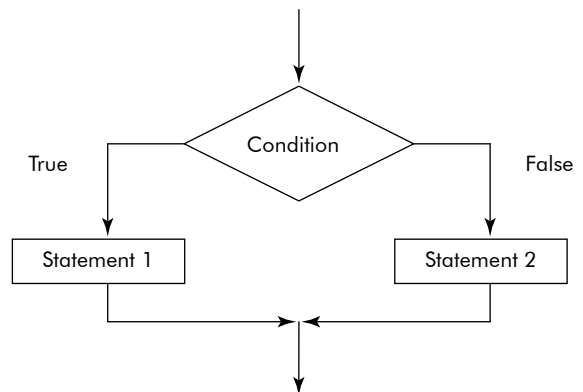


Fig. 2.8 The if-then-else structure

complexity T_2 . The time complexity of if-then-else structure is taken to be the maximum of the two, i.e., $\max(T_1, T_2)$.

Example 10: Consider the following algorithm. Compute its time complexity.

```

If (x > y)
{
    x = x+1;
}
Else
{
    For (i=1; i <= N, i++)
    {
        x = x+i;
    }
}

```

Solution: It may be noted that in the above algorithm, the time complexity of 'then' and 'else' part are $O(1)$ and $O(N)$, respectively. The maximum of the two is $O(N)$. Therefore, the time complexity of the above algorithm is $O(N)$.

Example 11: compute the time complexity of the following relations:

- (1) $T(n) = 2527$
- (2) $T(n) = 8*n + 17$
- (3) $T(n) = 15*N^2 + 6$
- (4) $T(n) = 5*N^3 + N^2 + 2*N$

Solution:

- (1) $T(n) = 2527 \leq 2527*1$, where $c = 2527$ and $n_0 = 0$.
 $= O(1)$

Ans. The time complexity = $O(1)$

- (2) $T(n) = 8*n + 17 \leq 8*n + n \leq 9*n$ for $c = 9$ and $n_0 = 17$
 $= O(N)$

Ans. The time complexity = $O(N)$

- (3) $T(n) = 15*N^2 + 6 \leq 15*N^2 + N$ for $N = 6$
 Now for $N \leq N^2$, we can rewrite the above relation as given below:
 $15*N^2 + N \leq 15*N^2 + N^2 \leq 16*N^2$ for $c = 16$ and $n_0 = 6$
 Thus, $T(n) = O(N^2)$

Ans. The time complexity = $O(N^2)$

- (4) $T(n) = 5*N^3 + N^2 + 2*N$
 For $N^2 \geq 2*N$, we can rewrite the above relation as given below:

$$5*N^3 + N^2 + 2*N \leq 5*N^3 + N^2 + N^2 \leq 5*N^3 + 2*N^2$$

For $N^3 \geq 2*N^2$, we can rewrite the above relation as given below:

$$5*N^3 + 2*N^2 \leq 5*N^3 + N^3 \leq 6*N^3 \text{ for } c = 6 \text{ and } n_0 = 2$$

Thus, $T(n) = O(N^3)$

Ans. The time complexity = $O(N^3)$

Note: All the relations, given in example 10 were polynomials. From the answers, we can say that any polynomial has the time complexity of Big-Oh of its leading term with coefficient of 1.

It is also understandable that the Big-Oh does not provide the exact execution time. Rather, it gives only an asymptotic upper bound of the execution time of an algorithm.

For different types of functions, the growth rates for different values of input size n have been tabulated in Table 2.1.

It can be observed from Table 2.1 that for a given input, the growth rate $O(\lg n)$ is faster than $O(n \lg n)$. For the same input size, the growth rate $O(n^2)$ is the slowest.

Later in the book, for various algorithms, timing estimates will be included using Big-Oh notation.

Table 2.1 Growth rates of functions

n	$\lg n$	$n \lg n$	n^2
1	0	0	1
2	1	2	4
16	4	64	256
256	8	2048	65536
1024	10	10240	1048576
4096	12	49152	16777216

EXERCISES

1. What is a stored program computer?
2. Give the block diagram of Von Neumann architecture of a computer.
3. What is a data structure? What is the need of data structures?
4. List out the areas in which data structures are applied extensively. Why is it important to make a right choice of data structures for problem solving?
5. What are built-in data structures? Give some examples.
6. What are user-defined data structures? Give some examples.
7. Consider a situation where it is required to rank 100 employees of an organization. The data of an employee consists of his name, designation, department, experience and salary. Give the possible choices for the data structures. Draw a comparison between the possible choices of organizing data.
8. What is the difference between data type and data structure?
9. Differentiate between linear and non-linear data structures. Explain with the help of examples.
10. Explain the following basic terminologies associated with the data structures:
 - i. Data element
 - ii. Primitive data types
 - iii. Data object
 - iv. Constant
 - v. Variable
11. What is an algorithm? Differentiate between algorithm and program.
12. List the desirable features of an algorithm.
13. What are the characteristics possessed by an algorithm?
14. What are the steps that can be suggested for developing an algorithm? Consider a suitable scenario and develop an algorithm for solving the problem. Explain each step while developing the algorithm.
15. What are the control structures? Explain the sequence control structures with the help of examples.
16. Write an algorithm that computes area and perimeter of a rectangle.
17. Write an algorithm that computes square of a number by repetitive addition.

18. Write an algorithm that computes GCD and LCM of two given numbers.
19. Explain the selection (conditional control) structures with the help of examples.
20. Write an algorithm that determines whether a number is even or odd.
21. Write an algorithm that computes profit or loss incurred by a salesperson.
22. What is iteration?
23. Write an algorithm that sorts a given list of numbers in ascending order.
24. Write an algorithm that determines whether a number is prime or not.
25. What is the use of accumulators and counters?
26. Write an algorithm that computes average marks of 60 students of a class.
27. Write an algorithm that finds the number of odds and evens present in a given list of N numbers.
28. What are the parameters on the basis of which an algorithm can be analyzed?
29. Define the term 'time complexity'. How can the time complexity of a given algorithm be found?
30. Explain Big-Oh notation with the help of examples.
31. Compute the time complexity of the following relations:
 - i. $T(n) = 410$
 - ii. $T(n) = 4 \cdot n - 12$
 - iii. $T(n) = 13 \cdot N^2 - 7$
 - iv. $T(n) = 3 \cdot N^3 + N^2 + 4 \cdot N$
32. What is the time complexity of the following loop?

```
for (int i = 0; i < n; i++)
{
    statement;
}
```
33. What is the Big-Oh complexity of the following nested loop?
 - i.

```
for (int i = 0; i < n; i++)
{
    for (int j = 0; j < n; j++)
    {
        statement;
    }
}
```
 - ii.

```
for (int i = 0; i < n; i++)
{
    for (int j = i + 1; j < n; j++)
    {
        statement;
    }
}
```
 - iii.

```
for (int i = 0; i < n; i++)
{
    for (int j = n; j > i; j--)
    {
        statement;
    }
}
```

```
iv. for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < m; j++)
        {
            statement1;
        }
        statement2;
    }
```

34. Consider the following algorithm and compute its time complexity.

```
if (a==b)
    statement1;
else if (a > b)
    statement2;
else
    statement3;
```

Arrays: Searching and Sorting

3

CHAPTER

CHAPTER OUTLINE

- 3.1 Introduction
- 3.2 One-dimensional Arrays
- 3.3 Multi-dimensional Arrays
- 3.4 Representation of Arrays in Physical Memory
- 3.5 Applications of Arrays

3.1 INTRODUCTION

An *array* is a data structure with the help of which a programmer can refer to and perform operations on a collection of similar data types such as simple *lists* or *tables* of information. For example, a list of names of ‘N’ number of students of a class can be grouped under a common name (say `studList`). This list can be easily represented by an array called `studList` for ‘N = 45’ students as shown in Figure 3.1

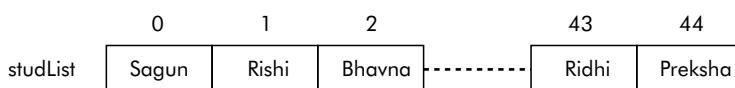


Fig. 3.1 Schematic representation of an array

In fact, the `studList` shown in Figure 3.1 can be looked upon by the following two points of views:

- (1) It is a linear list of 45 names stored in contiguous locations—an abstract view of a list having finite number of homogeneous elements (i.e., 45 names).
- (2) It is a set of 0 to 44 memory locations sharing a common name called `studList`—it is an array data structure in which 45 names have been stored.

It may be noted that all the elements in the array are of same type, i.e., *string of char* in this case. The individual elements within the array can be designated by an index. The individual elements are randomly accessible by integers, called the index.

For instance, the zeroth element (Sagun) in the list can be referred to as `studList [0]` and the 43rd element (Ridhi) as `studList [43]` where 0 and 43 are the indices. An index has to be of type integer.

An index is also called a subscript. Therefore, individual elements of an array are called subscripted variables. For instance, `studList [0]` and `studList [43]` are subscripted variables.

From the above discussion, we can define **an array as a finite ordered collection of items of same type**. It is a set of *index, value* pairs.

An array is a built-in data structure in every programming language. Arrays are designed to have a fixed size. Some languages provide **zero-based** indexing whereas other languages provide **one-based** indexing. 'C' is an example of *zero-based indexing* language because the index of its arrays starts from 0. Pascal is the example of *one-based indexing* because the index of its arrays starts from 1.

An array whose elements are specified by a single subscript is known as *one-dimensional array*. The array whose elements are specified by two or more than two subscripts is called as *multi-dimensional array*.

3.2 ONE-DIMENSIONAL ARRAYS

One-dimensional arrays are suitable for processing lists of items of identical types. They are very useful for problems that require the same operation to be performed on a group of data items. Consider Example 1.

Example 1: The examination committee of the university has decided to give 5 grace marks to every student of a class. Assume that there are 30 students in the class and their marks are stored in a list called 'marksList'. Write a program that adds the grace marks to every entry in the 'marksList'.

Solution: As the problem deals with a list of marks, we can select a one-dimensional array called 'marksList' to represent the list. The grace marks are to be added to all the elements of the list, for which we must use the For-loop. The required program is given below:

```
/* This program adds grace marks to all elements of a list of marks */
#include <stdio.h>
void main()
{
    int marksList[30];
    int i;
    int grace = 5;          /*grace marks */
    /* Read the marksList */
    printf ("\n Enter the list of marks one by one");
    for (i = 0; i < 30; i++)
    {
        scanf ("%d", &marksList[i]);
        marksList[i] = marksList[i] + grace;
    }

    /* Print the Final list */
    printf ("\n The final List is ...");
    printf ("\n S. No.\t Marks");
    for (i = 0; i < 30; i++)
        printf ("\n %d\t%d", i, marksList[i]);
}
```



It may be noted that in the above program, *a component by component processing* has been done on the all elements of the array called marksList.

The following operations are defined on array data structure:

- **Traversal:** An array can be travelled by visiting its elements starting from the zeroth element to the last in the list.
- **Selection:** An array allows selection of an element for a given (random) index.
- **Searching:** An array can be searched for a particular value.
- **Insertion:** An element can be inserted in an array at a particular location.
- **Deletion:** An element can be deleted from a particular location of an array.
- **Sorting:** The elements of an array can be manipulated to obtain a sorted array.

We shall discuss the above-mentioned operations on arrays in the subsequent sections.

3.2.1 Traversal

It is an operation in which each element of a list, stored in an array, is visited. The travel proceeds from the zeroth element to the last element of the list. Processing of the visited element can be done as per the requirement of the problem.

Example 2: Write an algorithm called 'listTravel' that travels a list stored in an array called List of size N. Each visited element is operated upon by an operator called OP.

Solution: The For-loop can be employed to visit and apply the operator OP on every element of the array. The required algorithm is given below:

```
Algorithm listTravel()
{
    Step
    1. For (i = 0; i < N, i++)
        {
            1.1 List[i] = OP(List[i]);
        }
}
```

Example 3: A list of N integer numbers is given. Write a program that travels the list and determines as to how many of the elements of the list are less than zero, zero, and greater than zero.

Solution: In this program, three counters called numNeg, numZero and numPos, would be used to keep track of the number of elements that are less than zero, zero, and greater than zero present in the list. The required program is given below:

```
/* This program determines the number of less than zero, zero, and greater
than zero numbers present in a list */

#include <stdio.h>
void main()
{
    int List [30];
    int N;
    int i, numNeg, numZero, numPos;
    printf ("\n Enter the size of the list");
    scanf ("%d", &N);
```



```

printf ("Enter the elements one by one");
/* Read the List*/
for (i = 0; i < N; i++)
{
    printf ("\n Enter Number:");
    scanf ("%d", &List[i]);
}
numNeg = 0;    /* Initialize counters*/
numZero = 0;
numPos = 0;
/* Travel the List*/
for (i=0; i < N; i++)
{
    if (List[i] < 0)
        numNeg = numNeg + 1;
    else
        if (List[i] == 0)
            numZero = numZero + 1;
        else
            numPos = numPos + 1;
}
}

```

3.2.2 Selection

An array allows selection of an element for a given (random) index. Therefore, the array is also called random access data structure. The selection operation is useful in a situation wherein the user wants query about the contents of a list for a given position.

Example 4: Write a program that stores a merit list in an array called 'Merit'. The index of the array denotes the merit position, i.e., 1st, 2nd, 3rd, etc. whereas the contents of various array locations represent the percentage of marks obtained by the candidates. On user's demand, the program displays the percentage of marks obtained for a given position in the merit list as per the following format:

```

Position: 4
Percentage: 93.25

```

Solution: An array called 'Merit' would be used to store the given merit list. Two variables *pos* and *percentage* would be used to store the merit position and the percentage of a candidate, respectively. A do-while loop would be used to iteratively interact with the user through following menu displayed to the user:

```

Menu:
Query----1
Quit-----2
Enter your choice

```

Note: As per the demand of the program, we will leave the zeroth location as unused and fill the array from index 1.

The required program is given below:

```
/* This program illustrates the random selection operation allowed by
   array data structures */
#include<stdio.h>
#include<conio.h>
void main()
{
    float Merit[30];
    int size;
    int i;
    int pos;
    float percentage;
    int choice;
    printf ("\n Enter the size of the list");
    scanf ("%d", &size);
    printf ("\n Enter the merit list one by one");
    for (i = 1; i <= size; i++)    /* Read Merit */
    {
        printf ("\n Enter data:");
        scanf ("%f", &Merit[i]);
    }
    /* display menu and start session */
    do
    {
        clrscr();
        printf ("\n Menu");
        printf ("\n Query-----1");
        printf ("\n Quit -----2");
        printf ("\n Enter your choice: ");
        scanf ("%d", & choice);
        /* use switch statement
           for determining the choice*/
        switch (choice)
        {
            case 1: printf("\n Enter Position");
                     scanf("%d", &pos);
                     percentage = Merit[pos]; /* the random selection */
                     printf ("\n position = %d", pos);
                     printf ("\n percentage = %4.2f",percentage);
                     break;
            case 2: printf ("\n Quitting");
        }
        printf ("\n press a key to continue...");
        getch();
    }
    while (choice != 2);
}
```

Searching and sorting are very important activities in a data processing environment and they are discussed in detail in the subsequent section.

3.2.3 Searching

There are many situations where we want to find out whether a particular item is present in a list or not. For instance, in a given voter list of a colony a person may search his name to ascertain whether he is a valid voter or not. For similar reasons, passengers look for their names in the railway reservation lists.

Note: System programs extensively search symbols, literals, mnemonics, ‘*compiler and assembler*’ directives, etc.

In fact, search is an operation in which a given list is searched for a particular value. The location of the searched element is informed. *Search* can be precisely defined as an activity of looking for a value or item in a list.

A list can be searched *sequentially* wherein the search for the data item starts from the beginning and continues till the end of the list. This simple method of search is also called **linear search**. It may be noted that for a list of size 1,000, the worst case is 1,000 comparisons.

Let us consider a situation wherein we are interested in searching a list of numbers called ‘numList’ for an element having value equal to the contents of a variable called val. It is desired that the location of the element, if found, be displayed.

The list of numbers can be comfortably stored in an array called numList of type int. To find the element, the list would be travelled in such a manner that each visited element would be compared to the variable ‘val’. If the match is found, then the location of the corresponding position would be stored in a variable called Pos. As the number of elements in the list is known, For-loop would be used to travel the array.

```
Algorithm searchList()
{
    step
    1. read numList
    2. read Val
    3. Pos = -1 ‘initialize Pos to a non-existing position’
    4. for (i = 0; i < N; i++)
        {
            4.1 if (val == numList[i])
                Pos = i;
        }
    5. if (Pos != -1)
        print Pos.
}
```

The ‘C’ implementation of the above algorithm is given below:

```
/* This program searches a given value called Val in a list called numList */
#include<stdio.h>
void main()
{
    int numList[20];
    int N;          /* Size of the list*/
    int Pos, val, i;
```

```

printf ("\n Enter the size of the List");
scanf ("%d", &N);
printf ("\n Enter the elements one by one");
for (i = 0; i < N; i++)
{
    scanf ("%d", &numList[i]);
}
printf ("\n Enter the value to be searched");
scanf ("%d", &val);
/* Search the element and its position in the list*/
Pos = -1;
for (i = 0; i < N; i++)
{
    if (val== numList[i])
    {
        Pos = i;
        break;
    }
    /* The element is found – come out of loop*/
}
if (Pos != -1)
    printf ("\n The element found at %d location", Pos);
else
    printf ("\n Search Failed");
}

```

Example 5: Write a program that finds the second largest element in a given list of N numbers.

Solution: The most basic solution for this problem is that if there are two elements in the list then the smaller of the two will be the second largest. In this case, we would set the second largest number to '-9999', a value not possible in the list.

In given problem, the list will be searched linearly for the required second largest number. Two variables called `firstLarge` and `secLarge` would be employed to store the first and second largest numbers. The following algorithm will be used:

```

Algorithm SecLarge()
{
    step
    1. read num;
    2. firstLarge = num;
    3. secLarge = -9999;
    4. for (i = 2; i <= N; i++)
    {
        4.1 read num;
        4.2 if (firstLarge < num)
            {secLarge = firstLarge;
             firstLarge = num;
            }
    }
}

```



```

        else
            if (secLarge < num)
                secLarge = num;
    }
    4. prompt "Second Large =";
    5. write secLarge;
}

```

The equivalent program is given below:

```

/* This program finds the second largest in a given list of numbers */
#include <stdio.h>
int Num, firstLarge, secLarge;
int N, i;
void main()
{
    printf ("\n Enter the size of the list");
    scanf ("%d", & N);
    printf ("\n Enter the list one by one");
    scanf ("%d", &Num);
    firstLarge = Num;
    secLarge = -9999;
    for (i = 2; i <= N; i++)
    {
        scanf ("%d", &Num);
        if (firstLarge < Num)
        {secLarge = firstLarge;
         firstLarge = Num;
        }
        else
            if (secLarge < Num)
                secLarge = Num;
    }
    printf ("\n Second Large = %d", secLarge);
}

```

The above discussed search through a list, stored in an array, has the following characteristics:

- The search is linear.
- The search starts from the first element and continues in a sequential fashion from element to element till the desired entry is found.
- In the worst case, a total number of N steps need to be taken for a list of size N.

Thus, the linear search is slow and to some extent inefficient. In special circumstances, faster searches can be applied.

For instance, binary search is a faster method as compared to linear search. It mimics the process of searching a name in a directory wherein one opens a page in the middle of the directory and examines the page for the required name. If it is found, the search stops; otherwise, the search is applied either to first half of the directory or to the second half.

3.2.3.1 Binary Search If a list is already sorted, then the search for an entry (say `val`) in the list can be made faster by using ‘*divide and conquer*’ technique. The list is divided into two halves separated by the middle element as shown in Figure 3.2.

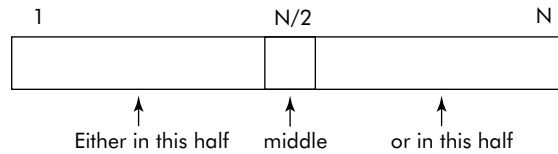


Fig. 3.2 Binary search

The binary search follows the following steps:
Step

- (1) The middle element is tested for the required entry. If found, then its position is reported else the following test is made.
- (2) If `val < middle`, search the left half of the list, else search the right half of the list.
- (3) Repeat step 1 and 2 on the selected half until the entry is found otherwise report failure.

This search is called binary because in each iteration, the given list is divided into two (i.e., binary) parts. Therefore, in next iteration the search becomes limited to half the size of the list to be searched. For instance, in first iteration the size of the list is N which reduces to almost $N/2$ in the second iteration and so on.

Let us consider a sorted list stored in an array called ‘Series’ given in Figure 3.3.

	0	1	2	3	4	5	6	7	8
Series	3	4	5	7	11	13	14	17	21

Fig. 3.3 The ‘Series’ containing nine elements

Suppose we desire to search the *Series* for a value (say 14) and its position in it. Binary search begins by looking at the middle value in the *Series*. The middle index of the array is approximated by averaging the first and last indices and truncating the result, i.e., $(0 + 8)/2 = 4$. Now, the content of the fourth location in *Series* happens to be ‘11’ as shown in Figure 3.4. Since the value we are looking for (i.e., 14) is greater than 11, the middle value, it may be present in the right half (*Series* [5] to *Series* [8]).

	0	1	2	3	4	5	6	7	8
Series	3	4	5	7	11	13	14	17	21

↑
Middle

Fig. 3.4 The middle value in the *Series* (1st step)

Now the middle of the right half is approximated i.e., $(5 + 8)/2 = 6$. We find that the desired element exists at the middle of the right half, i.e., Series [6] = 14 as shown in Figure 3.5.

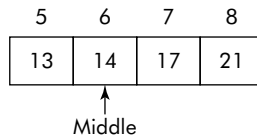


Fig. 3.5 The middle value in the Series (2nd step)

It may be noted that the desired element has been found only in two steps. Thus, it is a much faster method as compared to the linear search. For instance, a list of 1,000 sorted elements would require 10 comparisons to search the entire list. An algorithm for this method is given below.

In this algorithm, we would employ a Boolean variable called `flag`. The `flag` will indicate the presence or absence of the element being searched.

```
Algorithm binSearch ()
{
    Step
    1. First = 0;
    2. Last = N - 1;
    3. Pos = -1;
    4. Flag = false;
    5. While (First <= Last and Flag == false)
    {
        5.1 Middle = (first + last) div 2
        if (Series [middle] == Val)
        {Pos= middle;
        flag = true;
        break from the loop;
        }
        else
        if (Series[middle] < Val) First = middle + 1;
        else Last = middle - 1;
    }
    6. if (flag == true)
        prompt "The value found at";
        write pos;
    else prompt "The value not found";
}
```

It may be noted that 'div' operator has been used to indicate that it is an integer division. The integer division will truncate the results to the nearest integer. If the desired value is found, then the flag is set to true and the *while loop* terminates; otherwise, a stage arrives when first becomes greater than the last, indicating the failure of the search. Thus, the variables `First` and `Last` keep track of the lower and the upper bounds of the array, respectively.

Example 6: Write a program that uses binary search to search a given value called `val` in a list of `N` numbers called `Series`.

Solution: The Algorithm `binSearch()` discussed above is used to write the required program.

```
/* This program uses binary search to find a given value called val in a
   list of N numbers */
#include <stdio.h>
#define true 1
#define false 0
void main()
{
    int First;
    int Last;
    int Middle;
    int Series[20]; /*The list of N sorted numbers*/
    int Val;
    int flag; /*The value to be searched */
    int N, Pos, i;

    printf ("\n Enter the size of the list");
    scanf ("%d", & N);

    printf ("\n Enter the sorted list one by one");

    for (i = 0; i < N; i++)
    {
        scanf ("%d", &Series[i]);
    }
    printf ("\n Enter the number to be searched");
    scanf ("%d", & Val);

    /* BIN SEARCH begins */
    Pos = -1;          /* Non-existing position */
    flag = false;      /* Assume search failure */

    First = 0;
    Last = N - 1;

    while ((First <= Last) && (flag == false))
    {
        Middle = (First + Last)/2;
        if (Series [Middle] == Val)
        {
            Pos = Middle;
            flag = true;
            break;
        }
        else
        {
            if (Series[Middle] < Val)
                First = Middle + 1;
            else

```



```

    Last = Middle - 1;
}
if (flag == true)
    printf ("\n The value found at %d", Pos);
else
    printf ("\n The value not found");
}

```

Binary search through a list, stored in an array, has the following characteristics:

- The list must be sorted, i.e., ordered.
- It is faster as compared to the linear search.
- A list with large number of elements would increase the total execution time, the reason being that the list must be ordered which requires extra effort.

3.2.3.2 Analysis of Binary Search As discussed above, we know that the binary search took two steps to search an element in a list of nine elements. However, in the worst case scenario for a list of 32 elements, we can visualize as follows:

1st step would get a sublist of size 16
 2nd step would get a sublist of size 8
 3rd step would get a sublist of size 4
 4th step would get a sublist of size 2
 5th step would get a sublist of size 1

Let us tabulate (Table 3.1) the number of steps taken by binary search for a given list of size n . From table 3.1, we can infer that for a given input size $n = 2^k$, number of steps = k .

■ **Table 3.1**

Size of list (n)		Number of steps
8	(2^3)	3
32	(2^5)	5
256	(2^8)	8
512	(2^9)	9

Now the relation $n = 2^k$ can be rewritten as $k = \log_2 n$

$$(a^x = n \equiv x = \log_a n)$$

Thus, the time taken by binary search $T(n) = k = \log_2 n$

To satisfy the relation 2.1, the above relation can be rewritten as shown below:

$$T(n) \leq 1 \cdot \log_2 n \quad \text{where } c = 1 \text{ and } n_0 = 0.$$

Comparing the above relation with relation 2.1, we get $g(n) = \log_2 n$. Therefore, the above relation can be expressed as:

$$T(n) = O(\log_2 n)$$

Hence, binary search has time complexity of the order of $\log_2 n$, i.e., $O(\log_2 n)$.

3.2.4 Insertion and Deletion

3.2.4.1 Insertion Insertion is an operation in which a new element is added at a particular place in a list. When the list is represented using an array, the element can be added very easily at the end of the list provided the space is available. However, insertion of an element at a particular location within the list is a difficult operation. For instance, if it is desired to insert an element at i th location of a list then an empty space will have to be created at the i th location of the list. In fact, all the elements from i th location are shifted by one place towards right to accommodate the new element.

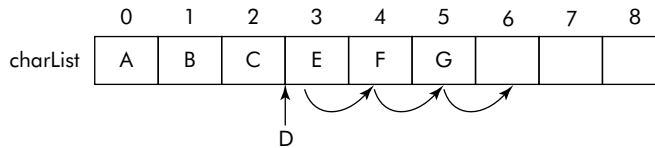


Fig. 3.6 The list charList before insertion

Consider a list of characters, represented using an array called 'charList' as shown in Figure 3.6. Suppose it is desired to insert a character 'D' at its proper place in charList, i.e., at the 3rd location.

Now to accommodate the character 'D', the characters 'G', 'F', and 'E' will have to be shifted one step towards right to locations 6th, 5th, and 4th, respectively as per the operations shown below:

```
charList [6] = charList[5];
charList [5] = charList[4];
charList [4] = charList[3];
```

Once the above copying operation is over, charList [3] becomes available for the storage of incoming character 'D' as shown below:

```
charList [3] = 'D';
```

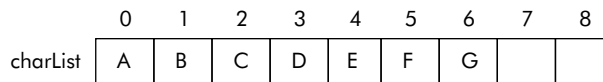


Fig. 3.7 The array charList after insertion

The array charList after the insertion operation is shown in Figure 3.7.

An algorithm for insertion of an element called `val` at i th location in an array called `charList` of size N is given below. Let us assume that the last element in the array is at M th location. For instance, $N = 8$ and $M = 5$ in case of charList shown in Figure 3.6. A variable called `Back` is being used to point to the last empty location:

```
Algorithm insertArray ()
{
    Step
```

```

/* point to the last vacant position */
1. if (M < N) then Back = M + 1;
   else
       STOP;
2. while (Back > I);
   { /* Copy elements to next location in the list */
       2.1 charList [Back] = charList [Back - 1];
       2.2 Back = Back - 1;
   }
3. charList [I] = val; /*Insert the element at ith location */
4. M = M + 1;
}

```

Example 7: Write a program that inserts an element called 'key' at a location called 'loc' in a list of M numbers called 'List'.

Solution: The algorithm insertArray() is being used to implement the required program.

```

/* This program inserts an element called Key into a list of numbers */
#include <stdio.h>
#define N 20
void main()
{
    int List[N];
    int Key;
    int Loc;
    int i, M;
    int back;

    printf ("\n Enter the size of the list (< 20)");
    scanf ("%d", &M);

    printf ("\n Enter the list one by one");
    for (i = 0; i < M; i++)
    {
        scanf ("%d", &List[i]);
    }
    printf ("\n Enter the key and location relative to 0 index");
    scanf ("%d %d", &Key, &Loc);
    /* Insert the 'key' at 'Loc' in 'List' */
    if (Loc > M + 1)
        printf ("\n Insertion not possible");
    else
    {
        back = M + 1;
        /* Shift elements one step right */
        while (back > Loc)
        {

```



```

    List[back] = List [back - 1];
    back--;
}
    /* Insert Key*/
    List[back] = Key;
    M=M + 1;
    /* Display the final list */
    printf ("\n The Final List is .....");
    for (i = 0; i < M; i++)
    {
        printf ("%d ", List[i]);
    }
}
}

```

3.2.4.2 Deletion Deletion is the operation that removes an element from a given location of a list. When the list is represented using an array, the element can be very easily deleted from the end of the list. However, if it is desired to delete an element from an i th location of the list, then all elements from the right of $(i+1)$ th location will have to be shifted one step towards left to preserve contiguous locations in the array.

For instance, if it is desired to remove an element from 4th location of the list given in Figure 3.8, then all elements from right of 4th location would have to be shifted one step towards left.

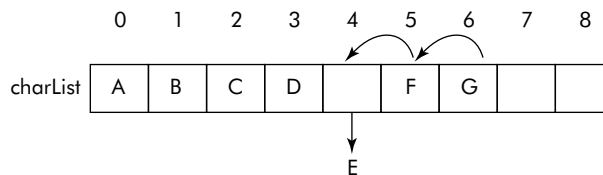


Fig. 3.8 Delete element from 4th location

Now to delete the character 'E', the characters 'F' and 'G' will have to be shifted one step towards left to locations 4th and 5th, respectively as per the operations shown below:

```

charList [4] = charList[5];
charList [5] = charList[6];

```

It may be noted that the contents of `charList [5]` (i.e., 'F') overwrites the contents of `charList [4]` (i.e., 'E'). The array `charList` after the deletion operation is shown in Figure 3.9.

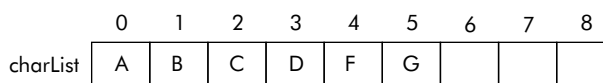


Fig. 3.9 The array `charList` after deletion

An algorithm for deletion of an element from i th location in an array called `charList` of size N is given below. Let us assume that the last element in the array is at M th location. For instance, $N = 8$ and $M = 6$ in case of `charList` shown in Figure 3.8. A variable called `Back` is used to point to the last empty location.

```
Algorithm delElement()
{
    Step
    1. Back = I; /* point to the location from where deletion is desired */
    2. while (Back < M)
    {
        2.1 charList[Back] = charList[Back + 1];    /* Shift elements one
            step left*/
        2.2 Back = Back + 1;
    }
    3. M = M - 1;
    4. Stop
}
```

It may be noted that in step 3, the contents of M have been decremented by 1 to indicate that after deletion the last element in the `charList` would be at $(M - 1)$ th location.

Example 8: Write a program that deletes an element from a location called `loc` in a list of M numbers called `List`.

Solution: The algorithm `delElement()` is used to implement the required program.

```
/*This program deletes an element from list of numbers */
#include <stdio.h>
#define N 20
void main()
{
    int List[N];
    int Loc;
    int i, M;
    int back;
    printf ("\n Enter the size of the list (< 20)");
    scanf ("%d", &M);
    printf ("\n Enter the list one by one");
    for (i = 0; i < M; i++)
    {
        scanf ("%d", &List[i]);
    }
    printf ("\n Enter location from where the deletion is required");
    scanf ("%d", &Loc);
    /* Insert the 'key' at 'Loc' in 'List' */
    if (Loc > M)
        printf ("\n Deletion not possible");
```



```

else
{
    back = Loc;
        /* Shift elements one step left */
    while (back < M)
    {
        List[back] = List [back + 1];
        back++;
    }
    M=M - 1;
        /* Display the final list */
    printf ("\n The Final List is .....");
    for (i = 0; i < M; i++)
    {
        printf ("%d ", List[i]);
    }
}
}

```

It may be noted that arrays are not suitable data structures for problems requiring insertions and deletions in a list. The reason is that we need to shift elements to right or to left for insertion and deletion operations, respectively. The problem aggravates when the size of the list is very large as an equally large number of shift operations would be required to insert and delete the elements. Thus, insertion and deletion are very slow operations as far as arrays are concerned.

3.2.5 Sorting

It is an operation in which all the elements of a list are arranged in a predetermined order. The elements can be arranged in a sequence from smallest to largest such that every element is less than or equal to its next neighbour in the list. Such an arrangement is called *ascending order*. Assuming an array called `List` containing N elements, the ascending order can be defined by the following relation:

$$\text{List}[i] \leq \text{List}[i + 1], \quad 0 < i < N - 1$$

Similarly in descending order, the elements are arranged in a sequence from largest to smallest such that every element is greater than or equal to its next neighbour in the list. The descending order can be defined by the following relation:

$$\text{List}[i] \geq \text{List}[i + 1], \quad 0 < i < N - 1$$

It has been estimated that in a data processing environment, 25 per cent of the time is consumed in sorting of data. Many sorting algorithms have been developed. Some of the most popular sorting algorithms that can be applied to arrays are *in-place* sort algorithms. An *in-place* algorithm is generally a *comparison-based* algorithm that stores the sorted elements of the list in the same array as occupied by the original one. A detailed discussion on sorting algorithms is given in subsequent sections.

3.2.5.1 Selection Sort It is a very simple and natural way of sorting a list. It finds the smallest element in the list and exchanges it with the element present at the head of the list as shown in Figure 3.10.

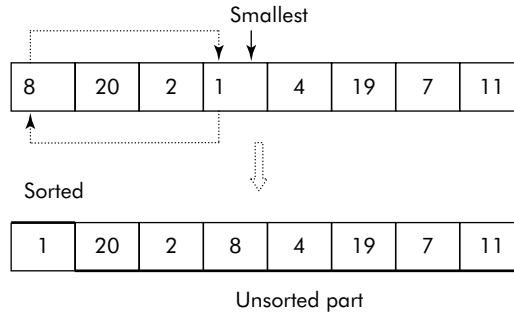


Fig. 3.10 Selection sort (first pass)

It may be noted from Figure 3.10 that initially, whole of the list was unsorted. After the exchange of smallest with the element on the head of the list, the list is divided into two parts: sorted and unsorted.

Now the smallest is searched in the unsorted part of the list, i.e., '2' and exchanged with the element at the head of unsorted part, i.e., '20' as shown in Figure 3.11.

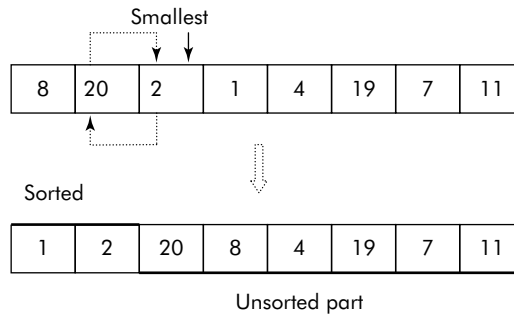


Fig. 3.11 Selection sort (second pass)

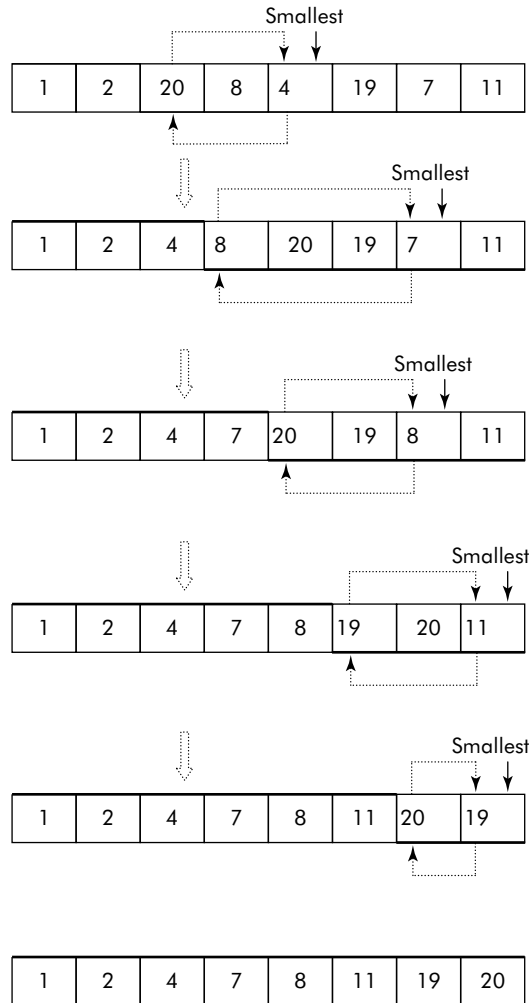
This process of selection and exchange (i.e., a pass) continues in this fashion until all the elements in the list are sorted (see Figure 3.12). Thus, in selection sort, two steps are important—*selection* and *exchange*.

From Figures 3.11 and 3.12, it may be observed that it is a case of nested loops. The outer loop is required for passes over the list and the inner loop for searching smallest element within the unsorted part of the list. In fact, for N number of elements, $N - 1$ passes are made.

An algorithm for selection sort is given below. In this algorithm, the elements of a list stored in an array called `LIST[N]` are sorted in ascending order. Two variables called `Small` and `Pos` are used to locate the smallest element in the unsorted part of the list. `Temp` is the variable used to interchange the selected element with the first element of the unsorted part of the list.

Algorithm `selSort()`

```
{
  Step
  1. For I = 1 to N - 1 /* Outer Loop */
```

**Fig. 3.12** Selection sort

```

{
  1.1 small = List [I];
  1.2 Pos = I;
  1.3 For J = I + 1 to N /* Inner Loop */
  {
    1.3.1 if (List [J] < small)
    {
      small = List[J];
      Pos = J; /* Note the position of the smallest*/
    }
  }
}

```



```

    1.4 Temp = List [I]; /*Exchange smallest with the Head */
    1.5 List [I] = List [Pos];
    1.6 List [Pos] = Temp;
}
2. Print the sorted list
}

```

Example 9: Given is a list of N randomly ordered numbers. Write program that sorts the list in ascending order by using selection sort.

Solution: The required program is given below:

In this program, the elements of a list are stored in an array called `List`. The elements are sorted using above given Algorithm `selSort()`. Two variables `small` and `pos` have been used to locate the smallest element in the unsorted part of the list. `Temp` is a variable used to interchange the selected element with the first element of the unsorted part of the list. With each step, the unsorted part becomes smaller. The process is repeated till all the elements are sorted.

```

/* This program sorts a list by using selection sort */
#include <stdio.h>
main()
{
    int list [10];
    int small, pos, N, i, j, temp;
    printf ("\n Enter the size of the list:");
    scanf ("%d", & N);

    printf ("\n Enter the list: ");
    for (i = 0; i < N; i++)
    {
        printf ("\n Enter Number:");
        scanf ("%d", &list[i]);
    }

    /* Sort the list */

    for (i = 0; i < N - 1; i++)
    {
        small = list[i];
        pos = i;

        /* Find the smallest of the unsorted list */
        for (j = i+1; j < N; j++)
        {
            if (small > list [j])
            {
                small = list [j];
                pos = j;
            }
        }

        /* Exchange the small with the

```



```

        first element of unsorted list */
    temp = list [i];
    list [i] = list [pos];
    list [pos] = temp;
}
printf ("\n The sorted list ...");
for (i = 0; i < N; i++)
    printf ("%d ", list[i]);
}

```

3.2.5.2 Analysis of Selection Sort In selection sort, there are two major operations: comparison and exchange. The average number of exchange operations would be difficult to estimate. Therefore, we will focus on comparison operations.

It may be noted that for every execution of outer loop, the inner loop executes in decreasing order. For example, in a list of (say) eight numbers, the first number would be compared to the remaining seven numbers to find out the smallest. After bringing the smallest to the first position, the second number would be compared to the remaining six, the third number with remaining five, and so on. Thus, the total number of comparisons for a list on N elements would be:

$$(N - 1) + (N - 2) + \dots + 2 + 1 = (N - 1) N/2 = N^2/2 - N/2$$

$$\text{Thus, } T(n) = N^2/2 - N/2.$$

we can say that

$$N^2/2 - N/2 \leq N^2/2.$$

Therefore, we can say that

$$T(n) \leq N^2/2 \quad \text{where } c = \frac{1}{2}, n_0 = 0, g(n) = N^2$$

$$T(n) = O(N^2)$$

Hence, selection sort has time complexity of the order of N^2 , i.e., $O(N^2)$.

3.2.5.3 Bubble Sort It is also a very simple sorting algorithm. It proceeds by looking at the list from left to right. Each adjacent pair of elements is compared. Whenever a pair is found not to be in order, the elements are exchanged. Therefore after the first pass, the largest element bubbles up to the right end of the list. A trace of first pass on a list of numbers is shown in Figure 3.13.

It may be noted that after the pass is over, the largest element in the list (i.e., 20) has bubbled up to the end of the list and six exchanges were made. Now the same process can be repeated for the list for second pass as shown in Figure 3.14.

We observe that after the second pass is over, the list has become sorted and only two exchanges were made. Now a point worth noting is as to how and when it will be decided that the list has become sorted. The simple criteria would be to check whether or not any exchange(s) has been made during the current pass. If 'yes', then the list is not yet sorted otherwise if it is 'no', then it can be decided that the list has just become sorted.

An algorithm for bubble sort is given below. In this algorithm, the elements of a list, stored in an array called `List[N]`, are sorted in an ascending order. The algorithm uses two loops—the outer while

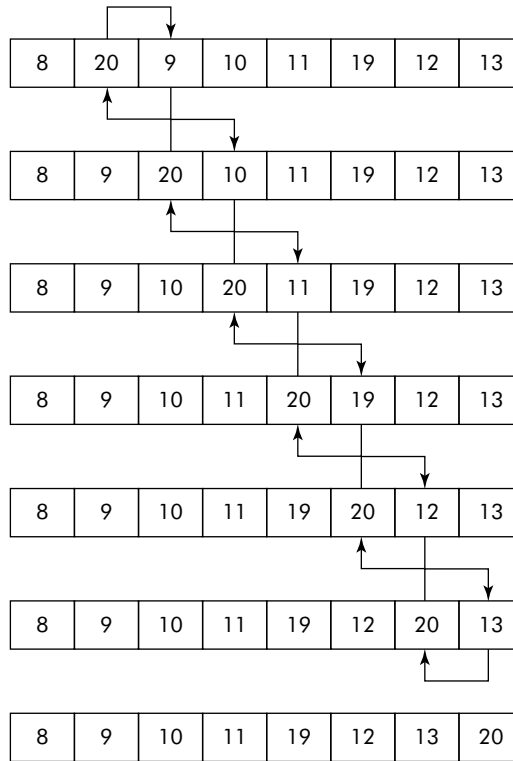


Fig. 3.13 First pass of bubble sort on a list of numbers

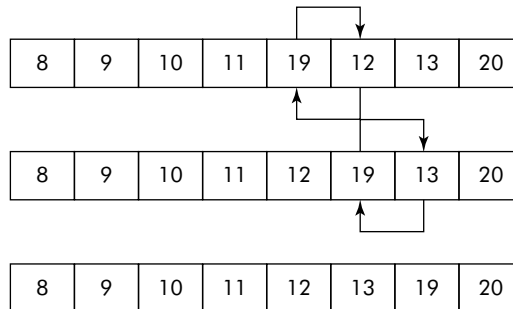


Fig. 3.14 Second pass of the bubble sort

loop and inner For loop. The inner For loop makes a pass on the list. If during the pass, any exchange(s) is made then it is recorded in a variable called `flag`, i.e., `flag` is set to false. The outer while loop keeps track of the `flag`. As soon as the `flag` informs that no exchange(s) took place during the current pass indicating that the list is now sorted, the algorithm stops.

```

Algorithm bubbleSort()
{
    Step
    1. Flag = false;
    2. While (Flag == false)
    {
        2.1 Flag = true;
        2.2 For j = 0 to N - 2
        {
            2.2.1 if (List [J] > List [J + 1])
            {
                temp = List[J];
                List[J] = List [J + 1];
                List [J + 1] = temp;
                Flag=false;
            }
        }
    }
    3. Print the sorted list
    4. Stop
}

```

It may be noted that the algorithm stops as soon as the list becomes sorted. Thus, 'bubble sort' is a very useful algorithm when the list is almost sorted i.e., only a very small percentage of elements are out of order.

Example 10: Given is a list of N randomly ordered numbers. Write a program that sorts the list in ascending order by using bubble sort.

Solution: The required program is given below:

In this program, the elements of a list are stored in an array called `List`. The elements are sorted using above given algorithm `bubbleSort()`.

```

/ *This program sorts a given list of numbers in ascending order, using
bubble sort */
#include <stdio.h>
#define N 20
#define true 1
#define false 0
void main()
{
    int List[N];
    int flag;
    int size;
    int i, j, temp;
    int count; /* counts the number of passes*/
    printf ("\n Enter the size of the list (< 20)");
    scanf ("%d", &size);

```

```

printf ("\n Enter the list one by one");
for (i = 0; i < size; i++)
{
    scanf ("%d", &List[i]);
}

/* Sort the list by bubble sort */
flag = false;
count = 0;
while (flag == false)
{
    flag = true; /* Assume no exchange takes place*/
    count++;
    for (j = 0; j < size-1; j++)
    {
        if (List[j] > List[j+1])
        {
            /* Exchange the contents */
            temp = List[j];
            List[j] = List[j + 1];
            List[j + 1] = temp;
            flag = false; /* Record the exchange operation*/
        }
    }
}

/* Print the sorted list*/
printf ("\n The sorted list is ....");
for (i = 0; i < size; i++)
    printf ("%d ", List[i]);
printf ("\n The number of passes made = %d", count);
}

```

It may be noted that the above program has used a variable called *count* that counts the number of passes made while sorting the list. The test runs conducted on the program have established that the program is very efficient in case of almost sorted list of elements. In fact, it takes only one scan to establish that the supplied list is already sorted.

Note: Bubble sort is also called a **sinking sort** meaning that the elements sink down in the list to their proper position.

3.2.5.4 Analysis of Bubble Sort In bubble sort, again, there are two major operations—comparison and exchange. The average number of exchange operations would be difficult to estimate. Therefore, we will focus on comparison operations.

A closer look reveals that for a list of *N* numbers, bubble sort also has the following number of comparisons, i.e., same as the selection sort.

$$(N - 1) + (N - 2) + \dots + 2 + 1 = (N - 1) N/2 = N^2/2 - N/2$$

Therefore, the time complexity of bubble sort is also $O(N^2)$.

However if the list is already sorted in the ascending order, no exchange operations would be required and it becomes the best case. The algorithm will have only *N* comparisons, i.e., only a linear running time.

3.2.5.5 Insertion Sort This algorithm mimics the process of arranging a pack of playing cards. In the pack of cards, the first two cards are put in correct relative order. The third card is inserted at correct place relative to the first two. The fourth card is inserted at the correct place relative to the first three, and so on.

Given a list of numbers, it divides the list into two part—sorted part and unsorted part. The first element becomes the sorted part and the rest of the list becomes the unsorted part as shown in Figure 3.15. It picks up one element from the front of the unsorted part and inserts it at its proper position in the sorted part of the list. This insertion action is repeated till the unsorted part is exhausted.

It may be noted that the insertion operation requires following steps:

Step

- (1) Scan the sorted part to find the place where the element, from unsorted part, can be inserted. While scanning, shift the elements towards right to create space.
- (2) Insert the element, from unsorted part, into the created space.

The algorithm for the insertion sort is given below. In this algorithm, the elements of a list stored in an array called `List[N]` are sorted in an ascending order. The algorithm uses two loops—the outer For loop and inner while loop. The inner while loop shifts the elements of the sorted part by one step to right so that proper place for incoming element is created. The outer For loop inserts the element from unsorted part into the created place and moves to next element of the unsorted part.

Algorithm `insertSort()`

```
{
  Step
  1. For I = 2 to N /* The first element becomes the sorted part */
  {
    1.1 Temp = List [I]; /* Save the element from unsorted part into temp */
    1.2 J = I - 1;
    1.3 While (Temp <= List [J] AND J >= 0)
    {
      List[J + 1] = List[J]; /* Shift elements towards right */
      J = J - 1;
    }
    1.4 List [J + 1] = Temp;
  }
  2. Print the list
  3. Stop
}
```

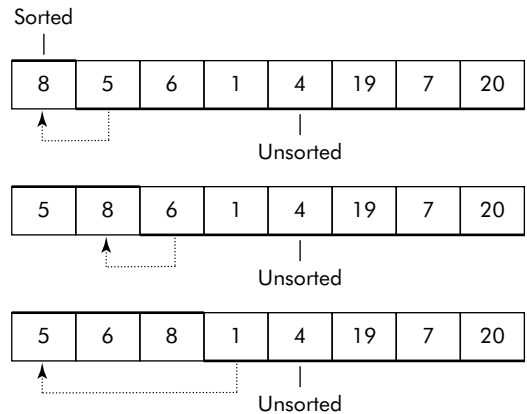


Fig. 3.15 Insertion sort

Example 11: Given is a list of N randomly ordered numbers. Write a program that sorts the list in ascending order by using insertion sort.

Solution: The required program is given below:

```
/*This program sorts a given list of numbers in ascending order using insertion sort */
#include <stdio.h>
#define N 20
void main()
{
    int List[N];
    int size;
    int i, j, temp;
    printf ("\n Enter the size of the list (< 20)");
    scanf ("%d", &size);
    printf ("\n Enter the list one by one");
    for (i = 0; i < size; i++)
    {
        scanf ("%d", &List[i]);
    }
    /* Sort the list by Insertion sort */
    for (i=1; i<size; i++)
    {
        temp = List[i]; /* Pick and save the first element of the unsorted part*/
        j= i - 1;
        while ((temp < List[j])&& (j>=0)) /* Scan for proper place */
        {
            List[j + 1] = List[j];
            j = j - 1;
        }
        List[j+1] = temp; /* Insert the element at the proper place */
    }
    /* Print the sorted list*/
    printf ("\n The sorted list is ....");
    for (i = 0; i < size; i++)
    {
        printf ("%d ", List[i]);
    }
}
```

3.2.5.6 Analysis of Insertion Sort A critical look at the algorithm indicates that in worst case, i.e., when the list is sorted in reverse order, the jth iteration requires (j – 1) comparisons and copy operations. Therefore, the total number of comparison and copy operations for a list of N numbers would be:

$$1 + 2 + 3 + \dots + (N - 2) + (N - 1) = (N - 1) N/2 = N^2/2 - N/2$$

Therefore, the time complexity of insertion sort is also $O(N^2)$.

However if the list is already sorted in the ascending order, no copy operations would be required. It becomes the best case and the algorithm will have only N comparisons, i.e., a linear running time.

3.2.5.7 Merge Sort This method uses following two concepts:

- (1) If a list is empty or it contains only one element, then the list is already sorted. A list that contains only one element is also called **singleton**.
- (2) It uses the old proven technique of 'divide and conquer' to recursively divide the list into sub-lists until it is left with either empty or singleton lists.

In fact, this algorithm divides a given list into two almost equal sub-lists. Each sub-list, thus obtained, is recursively divided into further two sub-lists and so on till singletons or empty lists are left as shown in Figure 3.16.

Since the singletons and empty lists are inherently sorted, the only step left is to merge the singletons into sub-lists containing two elements each (see Figure 3.17) which are further merged into sub-lists containing four elements each and so on. This merging operation is recursively carried out till a final merged list is obtained as shown in Figure 3.17.

Note: The merge operation is a time consuming and slow operation. The working of merge operation is discussed in the next section.

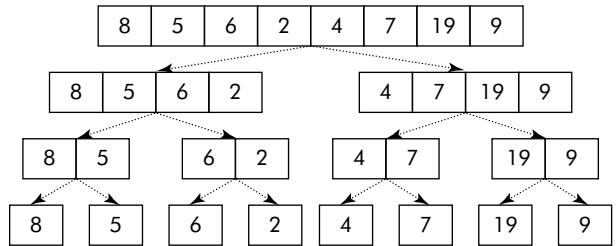


Fig. 3.16 First step of merge sort (divide)

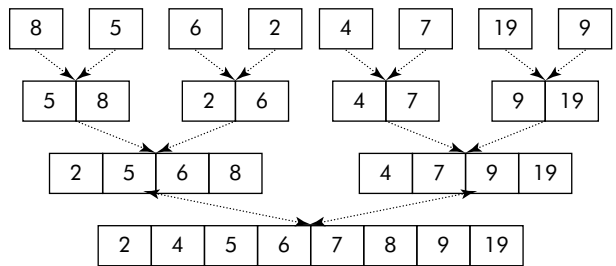


Fig. 3.17 Second step of merge sort (merge)

Merging of lists It is an operation in which two ordered lists are merged into a single ordered list. The merging of two lists PAR1 and PAR2 can be done by examining the elements at the head of the two lists and selecting the smaller of the two. The smaller element is then stored into a third list called `mergeList`. For example, consider the lists PAR1 and PAR2 given in Figure 3.18. Let `Ptr1`, `Ptr2`, and `Ptr3` variables point to the first locations of lists PAR1, PAR2, and PAR3, respectively. The comparison of `PAR1[Ptr1]` and `PAR2[Ptr2]` shows that the element of PAR1 (i.e., '2') is smaller. Thus, this element will be placed in the `mergeList` as per the following operation:

```
mergeList[Ptr3] = PAR1[Ptr1];
Ptr1++;
Ptr3++;
```

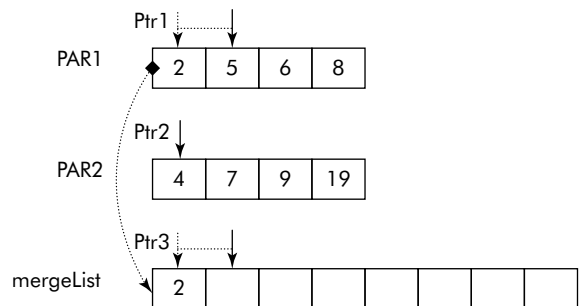


Fig. 3.18 Merging of lists (first step)

Since an element from the list PAR1 has been taken to mergeList, the variable `Ptr1` is accordingly incremented to point to the next location in the list. The variable `Ptr3` is also incremented to point to next vacant location in mergeList.

This process of comparing, storing and shifting is repeated till both the lists are merged and stored in mergeList as shown in Figure 3.19.

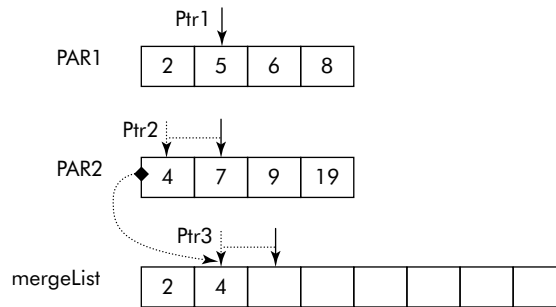


Fig. 3.19 Merging of lists (second step)

It may be noted here that during this merging process, a situation may arise when we run out of elements in one of the lists. We must, therefore, stop the merging process and copy rest of the elements from unfinished list into the final list.

The algorithm for merging of lists is given below. In this algorithm, the two sub-lists are part of the same array `List[N]`. The first sub-list is stored in locations `List[lb]` to `List[mid]` and the second sub-list is stored in locations `List [mid+1]` to `List [ub]` where `lb` and `ub` mean lower and upper bounds of the array, respectively.

Algorithm merge (`List`, `lb`, `mid`, `ub`)

```
{
    Step
    1. ptr1 = lb; /* index of first list */
    2. ptr2 = mid; /* index of second list */
    3. ptr3 = lb; /* index of merged list */
    4. while ((ptr1 < mid) && ptr2 <= ub) /* merge the lists */
    {
        4.1 if (List[ptr1] <= List [ptr2])
            {mergeList [ptr3] = List[ptr1]; /* element from first list is taken */
             ptr1++; /* move to next element in the list*/
             ptr3++;
            }
        4.2 else
            {mergeList [ptr3] = List[ptr2]; /* element from second list is taken*/
             ptr2++; /* move to next element in the list*/
             ptr3++;
            }
    }
}
```

```

    }
    5. while (ptr1 < mid) /* copy remaining first list */
    {
        5.1 mergeList [ptr3] = List[ptr1];
        5.2 ptr1++;
        5.3 ptr3++;
    }
    6. while (ptr2 <= ub) /* copy remaining second list */
    {
        6.1 mergeList [ptr3] = List[ptr2];
        6.2 ptr2++;
        6.3 ptr3++;
    }
    7. for (i = lb; i<ptr3; i++) /* copy merged list back into original
        list */
        7.1 List[i] = mergeList[i];
    8. Stop
}

```

It may be noted that an extra temporary array called `mergeList` is required to store the intermediate merged sub-lists. The contents of the `mergeList` are finally copied back into the original list.

The algorithm for the merge sort is given below. In this algorithm, the elements of a list stored in an array called `List[N]` are sorted in an ascending order. The algorithm has two parts—`mergeSort` and `merge`. The `merge` algorithm, given above, merges two given sorted lists into a third list, which is also sorted. The `mergeSort` algorithm takes a list and stores into an array called `List[N]`. It uses two variables `lb` and `ub` to keep track of lower and upper bounds of list or sub-lists as the case may be. It recursively divides the list into almost equal parts till singletons or empty lists are left. The sub-lists are recursively merged through `merge` algorithm to produce final sorted list.

Algorithm `mergeSort (List, lb, ub)`

```

{
    Step
    1. if (lb < ub)
    {
        1.1 mid = (lb + ub)/2; /* divide the list into two sub-lists */
        1.2 mergeSort (List, lb, mid); /* sort the left sub-list */
        1.3 mergeSort (List, mid +1, ub); /* sort the right sub-list */
        1.4 merge(List, lb,mid+1,ub); /* merge the lists */
    }
    2. Stop
}

```

Example 12: Given is a list of N randomly ordered numbers. Write a program that sorts the list in ascending order by using merge sort.

Solution: The required program uses both the algorithms—`mergeSort()` and `merge()`.

```

/* This program sorts a given list of numbers in ascending order using
merge sort */

```

```

#include <stdio.h>
#include <conio.h>
#define N 20
void mergeSort (int List[], int lb, int ub);
void merge (int List[], int lb, int mid, int ub);
void main()
{
    int List[N];
    int i, size;
    int mid;
    printf ("\n Enter the size of the list (< 20)");
    scanf ("%d", &size);
    printf ("\n Enter the list one by one");
    for (i=0; i< size; i++)
    {
        scanf ("%d", &List[i]);
    }
    /* Sort the list by merge sort */
    mergeSort (List,0,size - 1);
    printf ("\n The sorted list is ....");
    for (i = 0; i< size; i++)
    {
        printf ("%d ", List[i]);
    }
}
void mergeSort (int List[], int lb, int ub)
{
    int mid;
    if (lb < ub)
    {
        mid = (lb + ub)/2;
        mergeSort (List, lb, mid);
        mergeSort (List, mid + 1, ub);
        merge(List, lb, mid + 1,ub);
    }
}
void merge (int List[], int lb, int mid, int ub)
{
    int mergeList[20];
    int ptr1, ptr2, ptr3;
    int i;
    ptr1=lb;
    ptr2=mid;
    ptr3=lb;
    while ((ptr1 <mid) && ptr2 <= ub)
    {

```

```

    if (List[ptr1] <= List [ptr2])
    {mergeList [ptr3] = List[ptr1];
    ptr1++;
    ptr3++;
    }
    else
    {mergeList [ptr3] = List[ptr2];
    ptr2++;
    ptr3++;
    }
}
while (ptr1 < mid)
{mergeList [ptr3] = List[ptr1];
ptr1++;
ptr3++;
}

while (ptr2 <= ub)
{mergeList [ptr3] = List[ptr2];
ptr2++;
ptr3++;
}
for (i = lb; i<ptr3; i++)
{
    List[i] = mergeList[i];
}
}

```

3.2.5.8 Analysis of Merge Sort The merge sort requires the following operations:

- (1) Divide the list into two sub-lists.
- (2) Sort each sub-list.
- (3) Merge the two sub-lists.
- (4) If number of elements = 1, then list is sorted.

For a list of N elements, steps 1 and 2 require two times the number of comparisons needed for sorting a sub-list of size $N/2$. The step 3 requires N steps to merge the two sub-lists.

Therefore, number of comparisons in merge sort can be defined by the following recursive relation:

$$\begin{aligned} \text{NumComp}(N) &= 0 \text{ if } N = 1, \\ &= 2 * \text{NumComp}(N/2) + N \text{ for } N > 1 \end{aligned}$$

Thus, $\text{NumComp}(1) = 0$

$$\text{NumComp}(N) = 2 * \text{NumComp}(N/2) + N \quad (3.1)$$

By similarity, we can define the following:

$$\text{NumComp}(N/2) = 2 * \text{NumComp}(N/4) + N$$

Putting above expressions into Eq. 3.1, we get

$$\text{NumComp}(N) = 4 \text{ NumComp}(N/4) + 2N = 2^2 \text{ NumComp}(N/2^2) + 2N \quad (3.2)$$

By repeating the above pattern, we shall arrive at the following relation:

$$\text{NumComp}(N) = 8 \text{ NumComp}(N/8) + 3N = 2^3 \text{ NumComp}(N/2^3) + 3N \quad (3.3)$$

We can generalize the above relation as:

$$\text{NumComp}(N) = 2^k \text{ NumComp}(N/2^k) + kN \quad (3.4)$$

The merge sort stops when we are left with one element per partition, i.e., when $N = 2^k$.

Now for $N = 2^k$ or $k = \log_2 N$, we can rewrite Eq. 3.4 as:

$$\begin{aligned} \text{NumComp}(N) &= N \text{ NumComp}(N/N) + N \log_2 N \\ &= N \text{ NumComp}(1) + N \log_2 N = N \cdot 0 + \log_2 N \cdot N = N \log_2 N \\ &= N \log_2 N \end{aligned}$$

$$\text{Thus, } T(N) = N \log_2 N \leq C \cdot N \log_2 N + n_0$$

$$T(N) = O(N \log_2 N) \text{ for } c = 1 \text{ and } n_0 = 0$$

Thus, the time complexity of merge sort is of the order of $N \log_2 N$ or simply $n \log n$.

3.2.5.9 Quick Sort This method also uses the technique of ‘divide and conquer’. On the basis of a selected element (pivot) from the list, it partitions the rest of the list into two parts—a sub-list that contains elements less than the pivot and other sub-list containing elements greater than the pivot. The pivot is inserted between the two sub-lists. The algorithm is recursively applied to the sub-lists until the size of each sub-list becomes 1, indicating that the whole list has become sorted.

Consider the list given in Figure 3.20. Let the first element (i.e., 8) be the pivot. Now the rest of the list can be divided into two parts—a sub-list that contains elements less than ‘8’ and the other sub-list that contains elements greater than ‘8’ as shown in Figure 3.20.

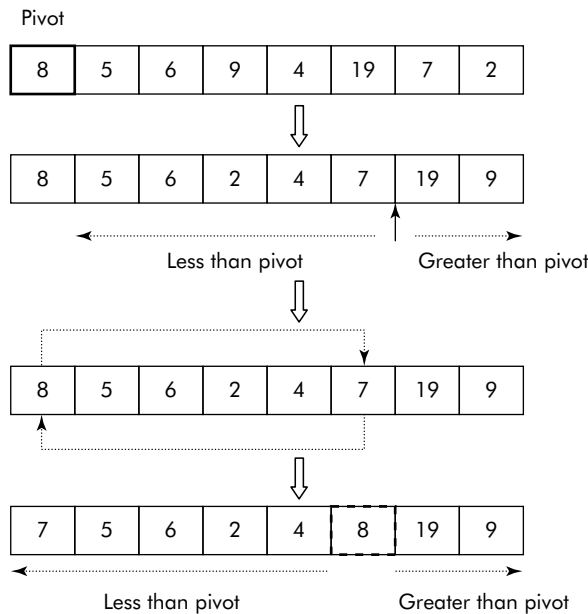


Fig. 3.20 Quick sort

Now this process can be recursively applied on the two sub-lists to completely sort the whole list. For instance, '7' becomes the pivot for left sub-list and '19' becomes pivot for the right sub-list.

Note: Two sub-lists can be safely joined when every element in the first sub-list is smaller than every element in the second sub-list. **Since 'join' is a faster operation as compared to a 'merge' operation, this sort is rightly named as a 'quick sort'.**

The algorithm for the quick sort is given below:

In this algorithm, the elements of a list, stored in an array called `List[N]`, are sorted in an ascending order. The algorithm has two parts—*quickSort* and *partition*. The partition algorithm divides the list into two sub-lists around a pivot. The quickSort algorithm takes a list and stores it into an array called `List[N]`. It uses two variables `lb` and `ub` to keep track of lower and upper bounds of list or sub-lists as the case may be. It employs partition algorithm to sort the sub-lists.

Algorithm `quickSort()`

```
{
  Step
  1. Lb = 0; /*set lower bound */
  2. ub = N - 1; /* set upper bound */
  3. pivot = List [lb];
  4. lb++;
  5. partition (pivot, List, lb, ub);
}
```

Algorithm `partition (pivot, List, lb, ub)`

```
{
  Step
  1. i = lb;
  2. j = ub;
  3. while (i<=j)
  {
      /* travel the list from lb till an element greater than
      the pivot is found */
      3.1 while (List[i] <= pivot) i++;
      /* travel the list from ub till an element smaller than
      the pivot is found */
      3.2 while (List[j] > pivot) j--;
      3.3 if (i <= j) /* exchange the elements */
      {
          temp = List[i];
          List[i] = List[j];
          List[j] = temp;
      }
  }
  4. temp = List[j]; /* place the pivot at mid of the sub-lists */
  5. List[j] = List[lb - 1];
  6. List[lb - 1] = temp;
  7. if (j > lb) quicksort (List, lb, j - 1); /* sort left sub-list */
  8. if (j < ub) quicksort (List, j + 1,ub); /* sort the right sub-list */
}
```

Example 13: Given is a list of N randomly ordered numbers. Write a program that sorts the list in ascending order by using quick sort.

Solution: The required program uses both the algorithms—quickSort() and partition(). In this program, a variable called *Key* has been used that acts as a pivot.

```
/* This program sorts a given list of numbers in ascending order, using
quick sort */
#include <stdio.h>
#include <conio.h>
#define N 20
void partition (int Key, int List[], int lb, int ub);
void quicksort (int List[], int lb, int ub);
void main()
{
    int List[N];
    int i, size, Pos, temp;
    int lb, ub;

    printf ("\n Enter the size of the list (< 20)");
    scanf ("%d", &size);

    printf ("\n Enter the list one by one");
    for (i = 0; i < size; i++)
    {
        scanf ("%d", &List[i]);
    }

    /* Sort the list by quick sort */
    quicksort (List, 0, size - 1);
    /* Print the sorted list*/
    printf ("\n The sorted list is ....");
    for (i = 0; i < size; i++)
    {
        printf ("%d ", List[i]);
    }
}

void quicksort (int List[], int lb, int ub)
{
    int Key; /* The Pivot */
    Key =List [lb]; lb++;
    partition (Key, List, lb, ub);
}

void partition (int Key, int List[], int lb, int ub)
{
    int i, j, temp;
    i = lb;
    j = ub;
```

```

while (i<=j)
{
while (List[i] <= Key) i++;
while (List[j] > Key) j--;
printf("\ni=%d j=%d", i, j);
getch();
if (i <=j)
{
temp = List[i];
List[i] = List[j];
List[j] = temp;
}
}
temp = List[j];
List[j] = List[lb - 1];
List[lb - 1] = temp;
if (j > lb) quicksort (List, lb, j - 1);
if (j < ub) quicksort (List, j + 1, ub);
}

```

3.2.5.10 Analysis of Quick Sort The quick sort requires the following operations:
Step

- (1) If number of elements = 1, then list is sorted and exit.
- (2) Partition the list into two sub-lists around a pivot.
- (3) Place pivot in the middle of two sub-lists.
- (4) Repeat steps 1 to 3.

The best case for quick sort algorithm comes when we split the input as evenly as possible into two sub-lists. Thus in the best case, each sub-list would be of size $n/2$. Anyway, the partitioning operation would require N comparisons to split the list of size N into the required sub-lists.

Now, for a list of size N , the number of comparisons can be defined as follows:

$$\begin{aligned}
 \text{NumComp}(N) &= 0 \text{ when } N = 1 \\
 &= N + \text{NumComp}(\text{sub-list1 of size } N/2) + \text{NumComp}(\text{sub-list2 of size } N/2) \\
 &= N + 2 * \text{NumComp}(N/2)
 \end{aligned}$$

The above relation is same as defined in merge sort; therefore, it can be deduced to the following:

$$T(N) = O(N * \log_2 N)$$

Thus, the best case time complexity of quicksort is of the order $N \log_2 N$ or simply $n \log n$.

3.2.5.11 Shell Sort It is an improvement on insertion sort. Given a list of $\text{List}[N]$, it is divided into k sets of N/k items each. k can assume values from a set $\{1, 3, 5, 19, 41, \dots\}$. Thus, the i th set will have the following elements:

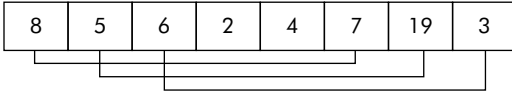
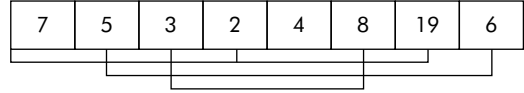
Set $i = \text{List}[i] \text{ List}[i + k] \text{ List}[i + 2k] \dots$

Consider the list given in Figure 3.21.

For $k = 5$, the list is divided into the following sets:

Set 0 = 8, 7

Set 1 = 5, 19

**Fig. 3.21** The sets of a list for $k = 5$ **Fig. 3.22** The list after first pass

Set 2 = 6, 3

Set 3 = 2

Set 4 = 4

Now on each set, the insertion sort is applied resulting in the arrangement shown in Figure 3.22.

For $k = 3$, the list is divided into the following sets:

Set 0 = 7, 2, 19

Set 1 = 5, 4, 6

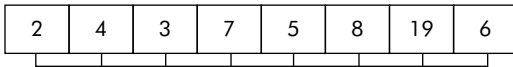
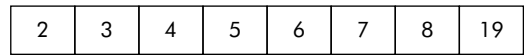
Set 2 = 3, 8

Now on each set, the insertion sort is applied resulting into the arrangement shown in Figure 3.23.

For $k = 1$, the list is represented by the following set:

Set 0 = 2, 4, 3, 7, 5, 8, 19, 6

Application of insertion sort on the above set results in the arrangement shown in Figure 3.24

**Fig. 3.23** The list after second pass**Fig. 3.24** The list after third pass

It may be noted that the list is now sorted after the third pass. Since the variable k takes the diminishing values—5, 3, and 1; the shell sort is also called **diminishing step sort**.

The algorithm for the shell sort is given below. In this algorithm, a list of elements, stored in an array called `List[N]`, are sorted in an ascending order. The algorithm divides the list into sets as per the description given above. The diminishing step values are stored in a list called `dimStep`. A variable 's' is used that moves the insertion operation to the next set. The variable k takes the step size from `dimStep` and moves the index i within the set from one element to another.

Algorithm `shellSort()`

```
{
    Step
    1. initialize the set dimStep to values 1,3,5,...
        /* sort the list by shell sort */
    2. for (step = 0; step < 3; step++)
    {
        2.1 k = dimStep[step]; /* set k to diminishing step */
        2.2 s = 0; /* start from the set of the list */
    }
```

```

        2.3 for (i = s + k; i <size; i += k)
        {
            temp = List[i]; /* save the element from the set */
            j = i - k;
                /* find the place for insertion */
            while ((temp < List[j]) && (j >= 0))
            {
                List[j + k] = List[j];
                j = j - k;
            }
            List[j + k] = temp; /* insert the saved element at its place */
            s++; /* go to next set */
        }
    }
    3. print the sorted list
    4. Stop
}

```

Example 14: Given is a list of N randomly ordered numbers. Write a program that sorts the list in ascending order by using shell sort.

Solution: The required program uses the algorithm `shellSort()`.

```

/* This program sorts a given list of numbers in ascending order using
Shell sort */
#include <stdio.h>
#define N 20
void main()
{
    int List[N];
    int size;
    int i, j, k, p;
    int temp, s, step;
    int dimStep[] = {5,3,1}; /* the diminishing steps */
    printf ("\n Enter the size of the list (< 20)");
    scanf ("%d", &size);
    printf ("\n Enter the list one by one");
    for (i=0; i< size; i++)
    {
        scanf ("%d", &List[i]);
    }

        /* sort the list by shell sort */
    for (step =0; step <3; step++)
    {
        k = dimStep[step]; /* set k to diminishing step */
        s=0; /* start from the set of the list */
        for (i = s + k; i <size; i += k)

```

```

{
    temp = List[i]; /* save the element from the set */
    j=i - k;
    while ((temp <List[j]) && (j >=0)) /* find the place for insertion */
    {
        List[j + k] = List[j];
        j = j - k;
    }
    List[j + k] = temp; /* insert the saved element at its place */
    s++; /* go to next set */
}
}
/* Print the sorted list*/
printf ("\n The sorted list is ....");
for (i = 0; i< size; i++)
{
    printf ("%d ", List[i]);
}
}

```

The analysis of shell sort is beyond the scope of the book.

3.2.5.12 Radix Sort It is a non-comparison-based algorithm suitable for integer values to be sorted. It sorts the elements digit by digit. It starts sorting the elements according to the least significant digit of the elements. This partially sorted list is then sorted on the basis of second least significant bit and so on.

Note: This algorithm is suitable to be implemented through linked lists. Therefore, discussion on radix sort is currently being postponed and it would be dealt with later in the book.

3.3 MULTI-DIMENSIONAL ARRAYS

An array having more than one subscript is called multi-dimensional array. C supports arrays of arbitrary dimensions. However, we will restrict ourselves to two dimensions only. A two-dimensional array has two subscripts. It represents two-dimensional entities such as tables, matrices etc. Therefore, sometimes the two-dimensional arrays are also called *matrix arrays*.

Consider the matrix A given in Figure 3.25. The matrix A has three rows and four columns. It contains 12 elements. Such a matrix can be represented by a two-dimensional array, capable of storing 12 elements in a row-column fashion. Let `MAT_A[3][4]` be the required array with two subscripts. The first subscript denotes rows of the matrix varying from 0 to 2 whereas the second subscript denotes the columns varying from 0 to 3.

$$A = \begin{pmatrix} 5 & 9 & 7 & 4 \\ 2 & 8 & 3 & 5 \\ 6 & 1 & 0 & 12 \end{pmatrix}_{3 \times 4}$$

Fig. 3.25 Matrix 'A'

Now the matrix A of Figure 3.25 can be comfortably stored in the array called `MAT_A[3][4]` as shown in Figure 3.26.

Let us consider a situation where it is desired that a value '5' be added to all elements of the matrix 'A', stored in array called `MAT_A`. This operation can be done by picking the element `MAT_A[0][0]`

We would use the above relation to compute the various elements of the matrix C. The required program is given below:

```

/* This program multiplies two matrices A & B and stores the
resultant matrix in C */
#include <stdio.h>
main()
{
    int  A[10][10], B[10][10], C[10][10];
    int  i, j, k; /* The array indices */
    int  m, n, r; /* Order of matrices */
    printf ("\n Enter the order of the matrices m, r, n:");
    scanf ("%d %d %d", &m,&r,&n);
    printf ("\n Enter The elements of Matrix A:");
    for (i = 0; i < m; i++)
    for (j = 0; j < r; j++)
    {
        printf ("\n Enter an element : ");
        scanf ("%d", &A[i][j]);
    }
    printf ("\n Enter The elements of Matrix B :");
    for (i = 0; i < r; i++)
    for (j = 0; j < n; j++)
    {
        printf ("\n Enter an element : ");
        scanf ("%d", &B[i][j]);
    }
        /* Multiply the matrices */
    for (i = 0; i < m; i++)
    {
        for (j = 0; j < n; j++)
        {
            C[i][j] = 0;
            for (k = 0; k < r; k++)
            {
                C[i][j] = C[i][j] + A[i][k] * B[k][j];
            }
        }
    }
    printf ("\n The resultant matrix is ...\n");
    for (i = 0; i < m; i++)
    {
        for (j = 0; j < n; j++)
        {
            printf ("%d ", C[i][j]);
        }
    }
}

```

```

        printf ("\n");
    }
}

```

Example 16: A *nasty* number is defined as a number which has at least two pairs of integer factors such that the difference of one pair equals to sum of the other pair. For instance, the factors of '6' are 1, 2, 3, and 6.

Now the difference of factor pair (6, 1) is equal to sum of factor pair (2, 3), i.e., $6 - 1 = 2 + 3$. Therefore, '6' is a nasty number.

Choose appropriate data structure and write a program that displays all the nasty numbers present in a list of numbers.

Solution: The following data structures would be used to store the list of numbers and the list of factor-pair:

- (1) A one-dimensional array called `List` to store the list of numbers.
- (2) A two-dimensional array called `pair_of_factors` to store the list of pair-factors of a given number.

For example, the list of pair factors of '24' would be stored as shown below:

1	24
2	12
3	8
4	6



Finally, using a nested loop, the difference of pair factors would be compared with the sum of all pair of factors for equality. The required program is given below:

```

/* This program finds all the nasty numbers from a given list of integer
   numbers */
#include <stdio.h>
#include <math.h>
main()
{
    int LIST [20];
    int pair_of_factors[50][2];
    int i, j, k, size, num, diff, sum, count;

    printf ("\n Enter the size of the list : ");
    scanf ("%d", &size);

    printf ("\n Enter the list : ");
    for (i = 0; i < size; i++)
        scanf ("%d", &LIST[i]);

    for (i = 0; i<size; i++)
    {

```

```

num = LIST[i];
count = 0;
    /* Compute the factors */
pair_of_factors[0][0] = 1;
pair_of_factors[0][1] = num;
for (j = 2 ; j <= sqrt(num); j++)
{
    if (num % j == 0)
    {
        count++;
        pair_of_factors[count][0] = j;
        pair_of_factors[count][1] = num/j;
    }
}

    /* Check for nastiness */
for (j = 0; j <= count; j++)
{
    diff = pair_of_factors[j][1] - pair_of_factors[j][0];
    for (k = 0; k <= count; k++)
    {
        sum = pair_of_factors[k][1] + pair_of_factors[k][0];
        if (diff == sum)
        {
            printf ("\n %d is a Nasty number", num);
            break;
        }
    } /* End of k loop */
} /* End of j loop */
} /* End of i loop */
} /* End of program */

```

3.4 REPRESENTATION OF ARRAYS IN PHYSICAL MEMORY

A programmer looks upon one-dimensional and two-dimensional arrays as ‘lists’ and ‘tables’ (or matrices), respectively. This is a logical view of the programmer about arrays as data structures, capable of storing lists or tables. However, at physical level, the main memory is RAM which is just a linear memory. The relationship between logical and physical views of arrays is shown in Figure 3.28.

Thus, the compiler has to map the various locations of arrays to locations of the physical memory. In fact, for a given array index, the compiler computes its physical load time address of physical memory location. For example, `List[5]` would be loaded at physical address = 105, in the physical memory (see Figure 3.28).

Since physical memory is linear by construction, the address mapping is quite a simple exercise. The compiler needs to know only the starting address of the first element in the physical memory. The rest of the addresses can be generated by a trivial formula. A brief discussion on address computation is given in the subsequent sections.

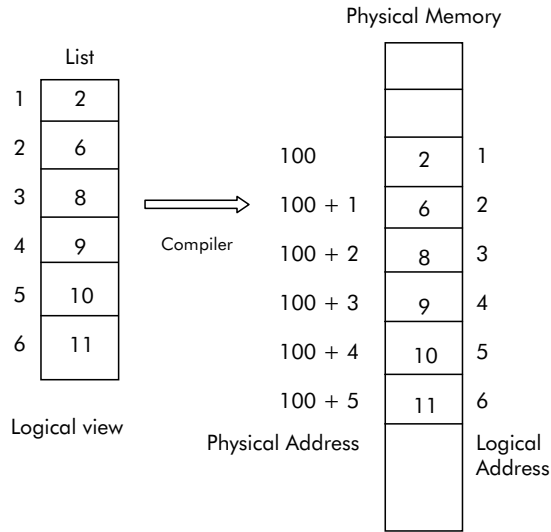


Fig. 3.28 Address mapping by compiler from logical to physical memory

3.4.1 Physical Address Computation of Elements of One-dimensional Arrays

Let us reconsider the array (`LIST[N]`) shown in Figure 3.28. The array called 'LIST' has six elements. It has been stored in physical memory at a starting address '100'. The address map is given below:

The address of 1st element = 100
 2nd element = 100 + 1
 3rd element = 100 + 2
 —
 —
 6th element = 100 + 5

It is very easy to deduce from above address map that the equivalent physical address of an I th element of array = $100 + (I - 1)$. The starting address of array at physical memory (i.e., 100) is called base address (Base) of the array. For example, the base address of array called LIST is '100'.

The formula for address of I th location can be written as given below:

$$\text{Address of List}[I] = \text{Base} + (I - 1) \quad (3.5)$$

Eq. 3.5 is a simple case, i.e., it is assumed that every element of List would occupy only one location of the physical memory. However if an element of an array occupies 'S' number of memory locations, the Eq. 3.5 can be modified as follows:

$$\text{Address of List}[I] = \text{Base} + (I - 1) * S \quad (3.6)$$

where

LIST: is the name of the array.

Base: is starting or base address of array.

I: is the index of the element.

S: is the number of locations occupied by an element of the array.

3.4.2 Physical Address Computation of Elements of Two-dimensional Arrays

Let us reconsider the two-dimensional array called `TAB[2][3]` as shown in Figure 3.29.

The array called 'TAB' has two rows and three columns. It has been stored in physical memory at a starting address '100'. The address map is given below:

The address of 1st row, 1st col = 100

1st row, 2nd col = 100 + 1

1st row, 3rd col = 100 + 2

—

—

2nd row, 3rd col = 100 + 5

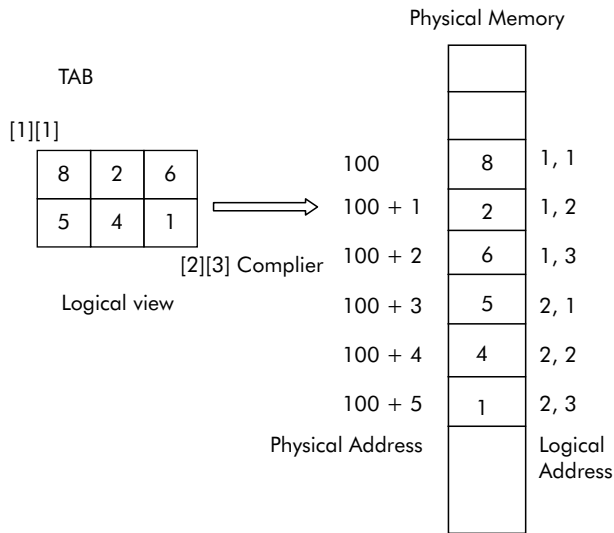


Fig. 3.29 Address mapping by compiler from logical to physical memory.

We know that in two-dimensional array called `TAB`, for every row, there are three columns. Thus, for an element of I th row the starting address becomes: $\text{Base} + (I - 1) * 3$ and if the element in row is at J th column, the complete address becomes:

$$\text{Address of TAB}[I][J] = \text{Base} + (I - 1) * 3 + J - 1 \quad (3.7)$$

To test the above row major order formula, let us calculate the address of element represented by `TAB[2][1]` with $\text{Base} = 100$.

Given: $I = 2, J = 1, \text{Base} = 100$. Putting these values in Eq. 3.7, we get

$$\text{Address} = 100 + (2 - 1) * 3 + 1 - 1 = 100 + 3 + 0 = 103.$$

From Figure 3.29, we find that the address value is correct.

Now we can generalize the formula of Eq. 3.7 for an array of size $[M][N]$ in the following form:

$$\text{Address of } [I][J] \text{ the element} = \text{BASE} + (I - 1) * N + (J - 1) \quad (3.8)$$

The formula given above is a simple case, i.e., it is assumed that every element of TAB would occupy only one location of the physical memory. However if an element of an array occupies 'S' number of memory locations, the row major order formula (3.8) can be modified as given below:

$$\text{Address of TAB [I][J] the element} = \text{BASE} + ((I - 1) * N + (J - 1)) * S \quad (3.9)$$

where

TAB: is the name of the array.

Base: is starting or base address of array.

I: is the row index of the element.

J: is the column index of the element.

N: is number of columns present in a row.

S: is the number of locations occupied by an element of the array.

Similarly, the formula for column major order is as follows:

$$\text{Address of TAB [I][J] the element} = \text{BASE} + ((J - 1) * M + (I - 1)) * S \quad (3.10)$$

where

TAB: is the name of the array.

Base: is starting or base address of array.

I: is the row index of the element.

J: is the column index of the element.

M: is number of rows present in a column.

S: is the number of locations occupied by an element of the array.

Example 17: TAB is a two-dimensional array with five rows and three columns. Each element occupies one memory location. If TAB[1][1] begins at address 500, find the location of TAB [4][2] for row major order of storage.

Solution: Given base = 500, I = 4, J = 2, M = 5, N = 3. Applying formula 3.9, we get:

$$\begin{aligned} \text{Address} &= \text{Base} + ((I - 1) * N + (J - 1)) * S \\ &= 500 + ((4 - 1) * 3 + (2 - 1)) * 1 \\ &= 500 + (9 + 1) \\ &= 510. \end{aligned}$$

Example 18: MAT is a two-dimensional array with ten rows and five columns. Each element is stored in two memory locations. If MAT[1][1] begins at address 200, find the location of MAT [3][4] for row major order of storage.

Solution: Given base = 200, I = 3, J = 4, M = 10, N = 5. Applying formula 3.9, we get:

$$\begin{aligned} \text{Address} &= \text{Base} + ((I - 1) * N + (J - 1)) * S \\ &= 200 + ((3 - 1) * 5 + (4 - 1)) * 2 \\ &= 200 + (10 + 3) * 2 \\ &= 226. \end{aligned}$$

Example 19: An array A[10][20] is stored in the memory with each element requiring two memory locations. If the base address of the array in the memory is 400, determine the location of array element A[8][3] when array is stored as column major order.

Solution: Given base = 400, I = 8, J = 3, M = 10, N = 20. Applying formula 3.10, we get:

$$\begin{aligned} \text{Address} &= \text{Base} + ((J - 1) * M + (I - 1)) * S \\ &= 400 + ((3 - 1) * 10 + (8 - 1)) * 2 \\ &= 400 + (20 + 7) * 2 \\ &= 454. \end{aligned}$$

3.5 APPLICATIONS OF ARRAYS

The arrays are used data structures which are very widely in numerous applications. Some of the popular applications are:

3.5.1 Polynomial Representation and Operations

An expression of the form $f(x) = a_n x^n + a_{n-1} x^{n-1} + a_{n-2} x^{n-2} + \dots + a_0$ is called a polynomial where $a_n \neq 0$.

$a_n x^n, a_{n-1} x^{n-1}, \dots, a_1 x^1, a_0$ are called terms.

$a_n, a_{n-1}, \dots, a_1, a_0$ are called coefficient.

x^n, x^{n-1}, x^{n-2} are called exponents,

Now, a polynomial can be considered as a list comprising coefficients and exponents as shown below:

$$F(x) = \{a_n x^n, a_{n-1} x^{n-1}, \dots, a_1 x^1, a_0, x^0\}$$

For example, the polynomial $6x^5 + 8x^2 + 5$ can be represented as the list shown below:

$$\text{Polynomial} = \{6, 5, 8, 2, 5, 0\}$$

The above list can be very easily implemented using a one-dimensional array, say `Pol[]` as shown in Figure 3.30 where every alternate location contains coefficient and exponent.

	coef	exp	coef	exp	coef	exp
Pol	6	5	8	2	5	0

Fig. 3.30 Array representation of a polynomial

The above representation can be modified to incorporate number of terms present in a polynomial by reserving the zeroth location of the array for this purpose. The modified representation is shown in Figure 3.31.

It may be noted from the above discussion that a general polynomial can be represented in an array with descending order of its degree of terms. Therefore, the operations such as addition, multiplication, and division on polynomials can be easily carried out.


	coef		exp	coef		exp	coef		exp
Pol	3	6	5	8	2	5	0		
									
	No. of terms								

Fig. 3.31 Array representation of a polynomial with zeroth place reserved for number of terms

Example 20: Write a program that reads two polynomials `pol1` and `pol2`, adds them and gives a third polynomial `pol3` such that $\text{pol3} = \text{pol1} + \text{pol2}$.

Solution: A close look at the array representation of polynomials indicates, that two polynomials can be added by merging them using 'algorithm merge()' of Section 3.2.5.7. The only change would be that when the exponents from both the polynomials are same, then the coefficients of both the terms would be added.

The required program is given below:

```
/* This program adds two polynomials pol1 and pol2 to give third polyno-
mial called pol3. It merges the two polynomials with the help of Algorithm
merge() with minor modifications */
#include <stdio.h>
void readPol (int pol[], int terms);
void dispPol (int pol[]);
void main()
{
    int pol1[11];
    int pol2[11];
    int pol3[21];
    int i, j, k;
    int terms;
    int m, n; /* size of polynomials */

    printf ("\n Enter the number of terms in Pol1");
    scanf ("%d", &terms);
    m = terms*2 + 1;

    readPol(pol1,terms);

    printf ("\n Enter the number of terms in Pol2");
    scanf ("%d", &terms);
    n = terms*2 + 1;

    readPol(pol2,terms);

    /* Add the polynomials by merging their terms */
    i = j = k = 2;

    while (i < m && j < n)
    {
        if (pol1[i] > pol2[j])
        {
            pol3[k] = pol1[i];
            pol3[k - 1] = pol1[i - 1];
            i = i + 2;
            k = k + 2;
        }
        else
        if (pol2[j] > pol1[i])
        {
            pol3[k] = pol2[j];
            pol3[k - 1] = pol2[j - 1];
            j = j + 2;
            k = k + 2;
        }
    }
```

```

    else
    {
        pol3[k - 1] = pol1[i - 1] + pol2[j - 1];
        pol3[k] = pol1[i];
        i=i+2;
        j= j+2;
        k= k+2;
    }
}

        /* copy the remaining Pol1 */
while (i < m)
{
    pol3[k] = pol1[i];
    pol3[k - 1] = pol1[i - 1];
    i = i + 2;
    k = k + 2;
}

        /* copy the remaining Pol2 */
while (j < n)
{
    pol3[k] = pol2[j];
    pol3[k - 1] = pol2[j - 1];
    j = j + 2;
    k = k + 2;
}
pol3[0] = (k - 1)/2;

        /* Display the final polynomial */
printf ("\n Final Pol =");
dispPol(pol3);
printf ("\b");
}
void readPol(int pol[], int terms)
{
    int i;
    pol[0] = terms;
    i = 1;
    printf ("\n Enter the terms (coef exp) in decreasing order of degree");

    while (terms > 0)
    {
        scanf ("%d %d", &pol[i], &pol[i+1]);
        i = i + 2;
        terms--;
    }
}

```

```

void dispPol(int pol[])
{
    int size;
    int i;
    size = pol[0]*2 + 1;
    printf ("\n");
    for (i = 1; i < size; i += 2)
    {
        printf ("%d*x^%d+", pol[i], pol[i + 1]);
    }
}

```

We can also choose other data structures to represent polynomials. For instance, a **term** can be represented by 'struct' construct of 'C' as shown below:

```

struct term
{
    int coef;
    int exp;
};

```

and the polynomial called 'pol' of ten terms can be represented by an array of structures of type 'term' as shown below:

```
struct term pol[10];
```

Now the algorithm `merge()` can be applied to add two such polynomials with above chosen data structures. It is left as an exercise for the readers to write that program.

3.5.2 Sparse Matrix Representation

A matrix is called *sparse* when it contains a large number of zero entries. Consider the matrix given in Figure 3.32.

It may be noted that the matrix MatA of order 6×8 has 42 zero entries out of total 48 entries. Thus, it has enough number of zero entries that prompts us to look for an alternate representation that could save the memory space occupied by zero entries. A popular representation of a sparse matrix is given below.

A sparse matrix can be very easily represented by a list of ordered pairs. An ordered pair has the following format

(Row, Col, E)

Where Row: is the row number

Col: is the column number

E: is the non zero entry at the given row and col.

$$\text{MatA} = \begin{pmatrix} 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 7 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 4 \\ 8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}_{6 \times 8}$$

Fig. 3.32 A matrix of order 6×8

Lets us now apply the above representation to MatA to obtain the list of ordered pairs as given below:

```
0, 4, 2
1, 1, 3
1, 6, 5
3, 3, 7
4, 7, 4
5, 0, 8
```

The above representation is also called a *condensed matrix*. It is using comparatively very less space which will become even prominent for large matrices, say of the order of 500×500 or more.

The proposed list of ordered pairs can be modeled as a two dimensional matrix of order of $n + 1 \times 3$. Where n is the number of nonzero elements present in a sparse matrix. However, the 0th ordered pair can be suitably used to store the order of the sparse matrix and the number of nonzero elements present in the matrix as shown in Figure 3.33.

```
int MatA [7][3] = { 6, 8, 6,
                    0, 4, 2
                    1, 1, 3
                    1, 6, 5
                    3, 3, 7
                    4, 6, 4
                    5, 0, 8 };
```

Example 21: Write a program that reads a sparse matrix of order $m \times n$ and stores it into the condensed representation as discussed above i.e. a two dimensional array of size $n+1 \times 3$. Where n is number of non zero elements. Print the contents of the condensed matrix.

Fig. 3.33 The condensed representation

Solution: The required program is given below:

```
/* This program reads a sparse matrix of order m*n and stores it into a
condensed matrix format */
# include <stdio.h>
# include <conio.h>
main()
{
    int matA[7][3];
    int i,j,k;
    int m,n, element;
    clrscr();
    printf ("\n Enter the order (m,n) of the sparse matrix");
    scanf ("%d %d", &m,&n);
    matA[0][0] = m;          /* no. of rows */
    matA[0][1] = n;          /* no. of cols */
    k=1;                     /* point to 1st position of sparse mat */
    printf ("\n Enter the elements of the matrix in row major order");
    for (i=0; i<m; i++)
    for (j=0; j < n; j++)
    {
        scanf ("%d", &element);
        if (element != 0)
```

```

        {
            /* store non zero element into condensed matrix */
            matA[k][0] = i;
            matA[k][1] = j;
            matA[k][2] = element;
            k++;
        }
    }
    matA[0][2] = k-1;          /* record no. of non zero elements */
    printf ("\n The condensed matrix is...");
    for (i=1;i <k; i++)
        printf ("\n %d %d %d", matA[i][0], matA[i][1],matA[i][2]);
}

```

Note: No space has been provided for the sparse matrix. The elements have been read one by one and the nonzero elements have been stored into the condensed matrix.

A sample run of the program is given below:

```

Enter the order <m,n> of the sparse matrix 6 8
Enter the elements of the matrix in row major order
0 0 0 0 2 0 0 0
0 3 0 0 0 0 5 0
0 0 0 0 0 0 0 0
0 0 0 7 0 0 0 0
0 0 0 7 0 0 0 0
0 0 0 0 0 0 4 0
8 0 0 0 0 0 0 0

The condensed matrix is...
0 4 2
1 1 3
1 6 5
3 3 7
4 6 4
5 0 8

```

3.5.2.1 Sparse Matrix Addition Two sparse matrices can be added in the same way as the polynomials were added in previous section. Consider the three sparse matrices Mat1, Mat2, and Mat3 given in Figure 3.34.

where $\text{Mat3} = \text{Mat1} + \text{Mat2}$

$$\begin{array}{ccc}
 \text{Mat 3} & & \text{Mat 1} \quad \text{Mat 2} \\
 \begin{pmatrix} 0 & 4 & 0 & 0 & 2 \\ 0 & 9 & 0 & 0 & 0 \\ 1 & 0 & 5 & 0 & 0 \end{pmatrix} & = & \begin{pmatrix} 0 & 0 & 0 & 0 & 2 \\ 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & 5 & 0 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 4 & 0 & 0 & 0 \\ 0 & 6 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix} \\
 3 \times 5 & & 3 \times 5 \quad 3 \times 5
 \end{array}$$

Fig. 3.34 The addition of matrix

The equivalent condensed form of Mat1, Mat2, Mat3 is given in Figure 3.35.

It may be noted that the final matrix Mat3, is nothing but a merger of two ordered lists Mat1 and Mat2 in a manner similar to polynomials. The 'algorithm merge()' of section 3.2.5.7 can be very easily applied. The only change would be that when the row number and column number from both the condensed matrices are same then the elements of both the matrices would be added. For example, merger of (1 1 3) and (1 1 6) would result into the element (1 1 9).

Mat3		Mat1		Mat2
3 5 5		3 5 3		3 5 3
0 1 4	=	0 4 2	+	0 1 4
0 4 2		1 1 3		1 1 6
1 1 9		2 2 5		2 0 1
2 0 1				
2 2 5				

Fig. 3.35

The equivalent condensed representation

Example 22: Write a program that reads two sparse matrices Mat1 and Mat2 of order $m \times n$, and represents them in condensed form. The condensed matrices are added to give a third condensed matrix called Mat3. It prints Mat3 in row major order.

Solution: We would use following three functions:

readMat(): It reads a sparse matrix and stores it into a condensed form

addsparsMat(): It takes two condensed matrices and adds them into a third condensed matrix

printMat(): It prints a condensed matrix

The required program is given below:

```
/* This program adds two sparse matrices of order m*n */
# include <stdio.h>
# include <conio.h>
void readMat (int matA[][], int m, int n);
void printMat(int matA[][]);
void addsparsMat (int mat1[][], int mat2[][], int mat3[][]);
void main()
{
    int mat1[10][3], mat2[10][3], mat3[20][3];
    int m,n;
    clrscr();
    printf ("\n Enter the order of the sparse mat1 : m, n");
    scanf ("%d %d", &m, &n);
    readMat(mat1,m,n);
    printMat(mat1);
    printf ("\n Enter the order of the sparse mat2 : m, n");
    scanf ("%d %d", &m, &n);
    readMat(mat2,m,n);
    printMat(mat2);
    addsparsMat (mat1,mat2,mat3);
    printf ("\n The final condensed Matrix");
    printMat(mat3);
}
void readMat (int matA[10][3], int m, int n)
{
```

```

int i,j,k;
int element;
matA[0][0] = m;           /* no. of rows */
matA[0][1] = n;           /* no. of cols */
k=1;                      /* point to 1st position of sparse mat */
printf ("\n Enter the elements of the matrix in row major order");
    for (i=0; i<m; i++)
        for (j=0; j < n; j++)
        {
            scanf ("%d", &element);
            if (element != 0)
            {
                matA[k][0] = i;
                matA[k][1] = j;
                matA[k][2] = element;
                k++;
            }
        }
    matA[0][2] = k-1;      /* record no. of non zero elements */
}
void printMat(int matA[10][3])
{
    int i, size;
    size = matA[0][2];
    for (i=0 ; i<= size; i++)
        printf ("\n %d %d %d", matA[i][0], matA[i][1],matA[i][2]);
}
void addsparsMat (int mat1[10][3], int mat2[10][3], int mat3[20][3])
{
    int i,j,k;
    int s1,s2;              /* No. of elements in mat1 and mat2 */
    s1 = mat1[0][2];
    s2 = mat2[0][2];
    i=j=k=0;
    while ( i <= s1 && j <= s2)
    {
        if (mat1[i][0] < mat2[j][0]) /* row of mat1 < mat2 */
        {
            mat3[k][0] = mat1[i][0];
            mat3[k][1] = mat1[i][1];
            mat3[k][2] = mat1[i][2];
            k++; i++;
        }
        else
            if (mat1[i][0] > mat2[j][0]) /* row of mat1 > mat2 */
            {

```

```

    mat3[k][0] = mat2[j][0];
    mat3[k][1] = mat2[j][1];
    mat3[k][2] = mat2[j][2];
    k++; j++;
}
else
{
    /* row of mat1 = mat2 */
    if (mat1[i][1] < mat2[j][1]) /* col of mat1 < mat2 */
    {
        mat3[k][0] = mat1[i][0];
        mat3[k][1] = mat1[i][1];
        mat3[k][2] = mat1[i][2];
        k++; i++;
    }
    else
    if (mat1[i][1] > mat2[j][1]) /* col of mat1 > mat2 */
    {
        mat3[k][0] = mat2[j][0];
        mat3[k][1] = mat2[j][1];
        mat3[k][2] = mat2[j][2];
        k++; j++;
    }
    else
    {
        /* col of mat1 = mat2 */
        mat3[k][0] = mat2[i][0];
        mat3[k][1] = mat2[i][1];
        mat3[k][2] = mat1[i][2] + mat2[j][2];
        k++; i++; j++;
    }
}
}
while (i <= s1) /* copy rest of the elements of mat1 */
{
    mat3[k][0] = mat1[i][0];
    mat3[k][1] = mat1[i][1];
    mat3[k][2] = mat1[i][2];
    k++; i++;
}
while (j <= s2) /* copy rest of the elements of mat2 */
{
    mat3[k][0] = mat2[j][0];
    mat3[k][1] = mat2[j][1];
    mat3[k][2] = mat1[j][2];
    k++; j++;
}
mat3[0][2] = k-1;
}

```

A sample output of the program is given below:

```

Enter the order of the sparse mat1 : m, n 3 5
Enter the elements of the matrix in row major order
0 0 0 0 2
0 3 0 0 0
0 0 5 0 0

3 5 3 condensed Matrix
0 4 2
1 1 3
2 2 5

Enter the order of the sparse mat2 : m, n 3 5
Enter the elements of the matrix in row major order
0 4 0 0 0
0 6 0 0 0
1 0 0 0 0

3 5 3 condensed Matrix
0 1 4
1 1 6
2 0 1

The final condensed Matrix
3 5 5
0 1 4
0 4 2
1 1 9
2 0 1
2 2 5

```

3.5.2.2 Sparse Matrix Transpose The transpose of a matrix of order $m * n$ is defined as a matrix $n * m$ that is formed by interchanging the rows and columns of the matrix. In fact, the transpose of the matrix is obtained by exchanging the element at (i, j) th position in the matrix t with the element at (j, i) th place as shown in Figure 3.36.

It may be noted that the number of rows and number of columns have also been exchanged.

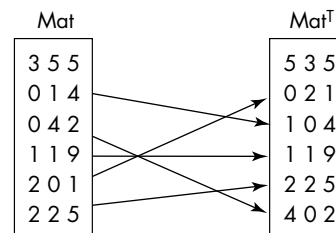


Fig. 3.36 Transpose of a sparse (condensed) matrix

Example 23: Write a program that produces a transpose of a given sparse matrix of order $m*n$.

Solution: The required program is given below

```

/* This program produces a Transpose of a sparse matrix */
# include <stdio.h>
# include <conio.h>
void readMat (int matA[][], int m, int n);
void printMat(int matA[][]);
void transMat (int mat[][], int trans[][]);
void main()
{
    int mat[10][3], trans[10][3];
    int m,n;
    clrscr();

```

```

printf ("\n Enter the order of the sparse mat : m, n");
scanf ("%d %d", &m, &n);
readMat(mat,m,n);
printMat(mat);
transMat (mat,trans);
printf ("\n The transposed condensed Matrix");
printMat(trans);
}
void readMat (int matA[10][3], int m, int n)
{
    int i,j,k;
    int element;
    matA[0][0] = m;          /* no. of rows */
    matA[0][1] = n;          /* no. of cols */
    k=1;                      /* point to 1st position of sparse mat */
    printf ("\n Enter the elements of the matrix in row major order");
        for (i=0; i<m; i++)
            for (j=0; j < n; j++)
                {
                    scanf ("%d", &element);
                    if (element != 0)
                        {
                            matA[k][0] = i;
                            matA[k][1] = j;
                            matA[k][2] = element;
                            k++;
                        }
                }
    matA[0][2] = k-1;        /* record no. of non zero elements */
}
void printMat(int matA[10][3])
{
    int i, size;
    size = matA[0][2];
    for (i=0 ; i<= size; i++)
        printf ("\n %d %d %d", matA[i][0], matA[i][1],matA[i][2]);
}
void transMat (int mat[10][3], int trans[10][3])
{
    int i, j, small, size,pos,temp;
    size = mat[0][2];
    trans[0][0] = mat[0][1];
    trans[0][1] = mat[0][0];
    trans[0][2] = mat[0][2];
    for (i=1; i<=size; i++)
        {

```

```

    trans[i][0] = mat[i][0];
    trans[i][1] = mat[i][1];
    trans[i][2] = mat[i][2];
}
for (i=1; i < size; i++)
{
    small = trans[i][1]; pos =i;
    for (j = i+1; j <=size; j++)
    {
        if (small > trans[j][1])
        {
            small = trans[j][1];
            pos = j;
        }
    }
    else
    if (small == trans[j][1] && trans[i][0] > trans[j][0])
    {
        small = trans[j][1];
        pos = j;
    }
}
temp =trans[i][0];trans[i][0] = trans[pos][0];trans[pos][0] =temp;
temp =trans[i][1];trans[i][1] = trans[pos][1];trans[pos][1] =temp;
temp =trans[i][2];trans[i][2] = trans[pos][2];trans[pos][2] =temp;
temp =trans[i][0];trans[i][0] = trans[i][1]; trans[i][1] = temp;
}
}

```

A sample output is given below:

```

Enter the elements of the matrix in row major order
0 0 0 0 0 8
0 3 0 0 0 0
0 0 0 0 0 0
0 0 0 7 0 0
2 0 0 0 0 0
0 0 0 0 0 0
0 5 0 0 4 0
0 0 0 0 0 0

8 6 6
0 5 8
1 1 3
3 3 7
4 0 2
6 1 5
6 4 4
The transposed condensed Matrix
6 8 6
0 4 2
1 1 3
1 6 5
3 3 7
4 6 4
0 5 8

```

EXERCISES

1. Define the terms: array, subscript, subscripted variables and strings.
2. Write suitable array declaration for the following:
 - 100 items of integer type.
 - A string of 25 characters.
 - A matrix of order 5×4 .
 - Can an array be initialized? If yes, how?
3. What is the purpose of initializing an array?
4. Can array be initialized at the time of declaration?
5. How can a string be stored in an array?
6. Write a program that finds the largest and smallest elements in an array.
7. Write a program that removes duplicate from a list.
8. Write a program that removes an element from an array.
9. Write a program that searches an array SOME of float type for a value less than x . The program should give appropriate message depending upon the outcome of the search.
10. Write a program which searches an ordered array SAL of integer type for a value X .
11. Write a program that gives the sum of positive elements of an array LIST of integer type.
12. Write a program that reads two matrices A and B of order $m \times n$ and compute the following:

$$C = A + B$$

13. What happens to an array if it is assigned with less number of elements at the time of initialization?
14. Write a program that computes the sum of diagonal elements of a square matrix.
15. Write a program that takes a sparse matrix A and finds its transpose A^T and displays it.
16. Let A be an $N \times N$ square matrix array. Write algorithms for the following:
 - Find the number of non-zero elements in A.
 - Find the product of the diagonal elements $(a_{11}, a_{22}, a_{33}, \dots, a_{nn})$.
17. Find out the complexity of binary search. Compare its time complexity with linear search.
18. What is meant by sorting? What are the types of sorting?
19. What is the difference between bubble sort and selection sort?
20. What is meant by quick sort? What are the advantages/disadvantages of quick sort over merge sort?
21. Explain how the selection of pivot plays major role in efficiency of quick sort algorithm and determine the complexity of the quick sort.
22. Trace the steps of insertion sort for the list of numbers: 12, 19, 33, 26, 29, 35, 22. Compute the total number of comparisons made.
23. What is shell sort? Define the worst case analysis of shell sort.
24. What is merge sort? Write algorithm for merge sort and derive its run time complexity.
25. How the two-dimensional arrays are represented in memory? Also, obtain the formula for calculating the address of any element stored in the array, in column major order.
26. Write a program that sorts a given list of numbers using bubble sort.
27. Write a program that sorts a given list of numbers using selection sort.
28. Derive the time complexity of selection sort.
29. Derive the time complexity of insertion sort.
30. Write a program that sorts a given list of numbers using quick sort.
31. Write a program that sorts a given list of numbers using merge sort.

Stacks and Queues

CHAPTER OUTLINE

- 4.1 Stacks
- 4.2 Applications of Stacks
- 4.3 Queues

4.1 STACKS

We are familiar with arrays and the operations performed on them. The various elements can be accessed from any position in a list, stored in an array. Some practical applications require that the additions or deletions be done only at one end. For example, the packets of wheat flour received in a super bazaar are put on top of another in the order of their arrival. Now, a customer can remove only the packet currently lying on the top, i.e., the last packet arrived in the bazaar is the first one to leave. The second customer would get the second last packet and so on. In the mean time, if more packets arrive then they will be added on the current top as shown in Figure 4.1.

It may be noted that in this kind of arrangement, all additions and deletions are made only from one end called top. In general, this activity is called stacking and the structure that allows deletions and additions from one end (top) is called a *stack*. It is also called LIFO (Last In First Out) data structure.

In a restaurant, the dish washing unit maintains the washed plates in a spring-loaded container. The newly washed plate is pushed on the top of the container and the customer may pop a plate from the top only. Therefore, the addition and deletion operations from a stack are accordingly also called Push and Pop, respectively as shown in Figure 4.2.

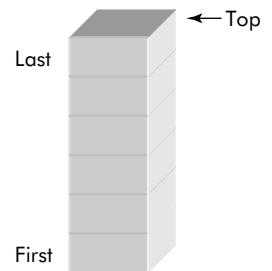


Fig. 4.1 The stack

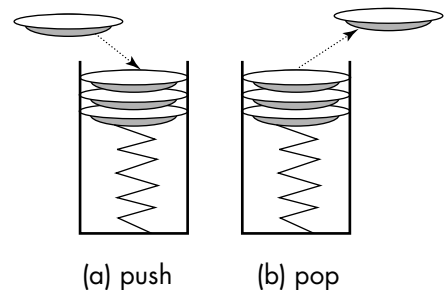


Fig. 4.2 The stack operations

Similarly, the nested calls to functions and procedures are also handled in LIFO fashion in the sense that control from last called procedure is handled first.

A stack can be more precisely defined as a linear collection of items which allows addition and deletion operations only from one end called **top**.

The push operation adds an item onto the **top** of the stack whereas the pop operation removes an element from the current **top** of the stack. The size of the stack depends upon the number of items present in the stack. With the addition and deletion of items, the size of stack accordingly increases or decreases. Therefore, stack is also called a *dynamic data structure* with a capacity to enlarge or shrink.

4.1.1 Stack Operations

As already discussed, there are two main operations (push and pop) associated with a stack. The push operation adds an item on the **top** of the stack whereas the pop operation removes an item from the **top** of the stack. Let us consider the stack shown in Figure 4.3 (a). Currently the **Top**, a variable, is pointing to the item 'D'. Let us now push an item 'E' on the stack. The **Top** is incremented to next location and the item is added at the pointed location as shown in Figure 4.3(b).

It may be noted that the variable called **Top** keeps track of the location where addition can be done. If there is a bound on the size of the stack (say N) then as soon as the **Top** becomes equal to N , the stack is said to be **full**.

Let us now apply pop operation three times in succession and see the results. The following three items will be removed from the stack in LIFO fashion:

'E' 'D' 'C'

The variable **top** is also decremented accordingly after each Pop operation as shown in Figure 4.4.

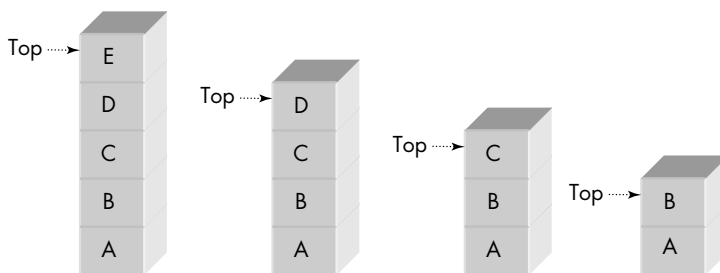


Fig. 4.3 Push operation on stack

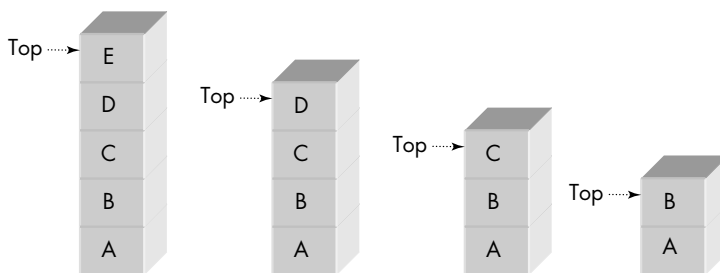


Fig. 4.4 Pop operations on the stack

It may be noted that if more pop operations are done on the stack then a stage will come when there will be no items left on the stack. This condition is called **stack empty**.

The algorithms for push and pop operations are given below. These algorithms use an array called **stack[N]** to represent a stack of N locations. A variable called **Top** keeps track of the **top** of the stack, i.e., the location where additions and deletions are made. Another variable called **item** is used to store the item to be pushed or popped from the stack.

The algorithm for push is given below:

Algorithm Push ()

```
{
    Step
    1. If (Top >= N) then { prompt ("Stack Full"); exit}
    2. Top = Top + 1;
    3. Stack [Top] = item;
    4. Stop
}
```

It may be noted that the item is added to the stack (i.e., push operation) provided the stack is not already full, i.e., at step 1, it is checked whether or not the stack has room for another item.

The algorithm for pop is given below:

Algorithm Pop ()

```
{
    Step
    1. if (Top < 0) then prompt ("Stack Empty");
       else
           Item = Stack [Top];
           Top = Top - 1;
    2. stop
}
```

It may be noted that in the above algorithm, before popping an item from the top, it is checked at step 1 to see if there is at least one item on the stack that can be removed.

The performance of the stack data structure for 'n' elements is:

- The space complexity is $O(n)$.
- The time complexity of each operation is $O(1)$.
- The size of the stack, implemented using array, must be defined in advance, i.e., *a priori*. Moreover, the size cannot be changed.

Example 1: Write a program in 'C' that uses the following menu to simulate the stack operations on items of int type.

Menu – stack operations	
Push an item	1
Pop an item	2
Display stack	3
Quit	4

Enter your choice:

Solution: The above defined algorithms Push () and Pop () would be employed to write the program. The menu would be created using printf () statements and displayed within a do-while loop. In fact, *a do-while loop is the most appropriate loop for display and manipulations of menu items.*


```
        break;
    case 2: result = pop( stack, &top);
        if (result == 9999)
            printf ("\n The stack is empty");
        else
            printf ("\n The popped value = %d", result);
        break;
    case 3: display (stack, top);
        break;
    }
    printf ("\n\n Press any key to continue");
    getch();
}
while (choice != 4);
}

int push (int stack[], int *top, int val, int size)
{
    if (*top >= size)
        return 0;    /* the stack is Full */
    else
    {
        *top= *top + 1;
        stack[*top]= val;
        return 1;
    }
}

int pop (int stack[], int *top)
{
    int val;
    if (*top < 0)
        return 9999;    /* the stack is empty */
    else
    {
        val = stack[*top];
        *top = *top - 1;
        return val;
    }
}

void display (int stack[], int top)
{
    int i;
    printf ("\n The contents of stack are:");
    for (i = top; i >=0; i--)
        printf ("%d ", stack[i]);
}
```

4.2 APPLICATIONS OF STACKS

The stacks are used in numerous applications and some of them are as follows:

- Arithmetic expression evaluation
- Undo operation of a document editor or a similar environment
- Implementation of recursive procedures
- Backtracking
- Keeping track of page-visited history of a web user

4.2.1 Arithmetic Expressions

An arithmetic expression is a combination of variables and/or constants connected by arithmetic operators and parenthesis. The valid arithmetic operators are given in Table 4.1.

The rules for an arithmetic expression are:

- A signed or unsigned variable or constant is an expression.
- An expression connected by an arithmetic operator to a variable or constant is an expression.
- Two expressions connected by an arithmetic operator is also an expression.
- Two arithmetic expressions should not occur in continuation.

Examples of some valid arithmetic expressions are:

- Sum + 10
- Total – 100
- Total/Rate*Interest
- $X^Y - B * C$

Examples of some invalid arithmetic expressions are:

- $X * / Y$ (two operators in continuation)
- $X Y$ (expression without an operator)

The order of evaluation, known as *operator precedence*, is carried out according to the priority of the arithmetic operators given in Table 4.2.

Note:

- The operators of same priority like ($*$, $/$) or ($+$, $-$) are performed from left to right.
- The operators contained within parentheses are evaluated first. When the parentheses are nested, the innermost is evaluated first and so on.
- The exponential operators are evaluated from right to left. For example, in the following expression,

$$A \wedge B \wedge C$$

■ **Table 4.1** Arithmetic operators

Symbol	Stands for	Example
\wedge	Exponentiation	$X \wedge 8$
$/$	Division	X/Y
$*$	Multiplication	$X * Y$
$+$	Addition	$X + Y$
$-$	Subtraction	$X - Y$

■ **Table 4.2** Operator precedence

S. No.	Operator(s)	Priority
1	\wedge , exponentiation	3
2	$/$, $*$, division, multiplication	2
3	$+$, $-$, addition, subtraction	1

The sub-expressions are evaluated from right to left as shown in Figure 4.5.

The arithmetic expression, as discussed above, is called an *infix* expression wherein the operator is placed between two operands. The evaluation takes place according to priorities assigned to the operators (see Table 4.2). However, to overrule the priority of an operator, parenthesis is used. For example, the two *infix* expressions given below are entirely different because in Expression 2, the priority of the division operator '/' has been overruled by the embedded parenthesis.

- (1) $X - Y/Z$
- (2) $(X - Y)/Z$

The order of evaluation of above *infix* expressions is shown in Figure 4.6.

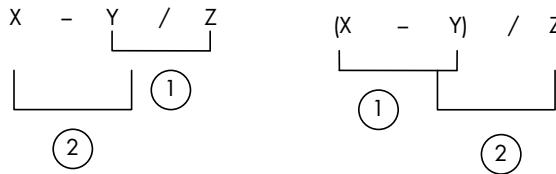


Fig. 4.6 Evaluation order of infix expressions

Thus, parenthesis play an important role in the evaluation of an *infix* expression and the order of evaluation depends upon the placement of parentheses and the operator precedence. Therefore, the *infix* representation of arithmetic expressions becomes inefficient from the compilation point of view. The reason being that to find the sub-expression with highest priority, at each step, repeated scanning of the expression is required from left to right.

A Polish logician J. Lukasiewicz introduced a notation which permits arithmetic expressions without parentheses. The absence of embedded parentheses allows simpler compiler interpretation, translation, and execution of arithmetic expressions. This notation is also called *Polish notation*. It appears in two forms: *prefix* and *postfix* forms.

4.2.1.1 Prefix Expression (Polish Notation) In the *prefix* form, an operator is placed before its operands, i.e., the operators are prefixed to their operands. For example, the *infix* expression $A + B$ is written as '+AB' in *prefix* form.

Examples of some valid *prefix* expressions with their equivalent *infix* expressions are given in Table 4.3.

Table 4.3 Some prefix expressions

Prefix expression	Equivalent infix expression
+AB	$A + B$
*+ABC	$(A + B)*C$
+/A + BC*D - EF	$A/(B + C) + D*(E - F)$

4.2.1.2 Postfix Expression (Reverse Polish Expression) In the postfix expression, the operator is placed after its operands, i.e., the expression uses suffix or postfix operators. For example, the infix expression 'A + B' is written as 'A B +' in postfix form.

Examples of some valid postfix expressions with their equivalent infix expressions are given in Table 4.4.

■ **Table 4.4** Some postfix expressions

Postfix expression	Equivalent infix expression
AB+	A + B
AB + C*	(A + B)*C
ABC +/DEF -*+	A/(B + C) + D*(E - F)

It may be noted that in Polish and reverse Polish notations, there are no parentheses and, therefore, the order of evaluation will be determined by the positions of the operators and related operands in the expression.

A critical look at the postfix expressions indicates that such representation is excellent from the execution point of view. The reason being that while execution, no consideration has to be given to the priority of the operators rather the placement of operators decides the order of execution. Moreover, it is free from parentheses. Same is true for prefix expressions but in this book, only the postfix expressions would be considered.

Thus, there is a need for conversion of infix to postfix expression. The subsequent section discusses two methods to convert the infix expressions to postfix expressions.

4.2.1.3 Conversion of Infix Expression to Postfix Expression An infix expression can be converted into postfix by two methods: parenthesis method and stack method.

(1) Parenthesis method The following steps are used to convert an infix expression to postfix expression by parenthesis method:

- (1) Fully parenthesize the infix expression.
- (2) Replace the right hand parenthesis by its corresponding embedded operator.
- (3) Remove the left hand parenthesis. The resultant expression is in postfix notation.

Consider the infix expression $A/(B + C) + D*(E - F)$. Let us apply the above steps to convert this expression to postfix notation. The operations are shown in Figure 4.7.

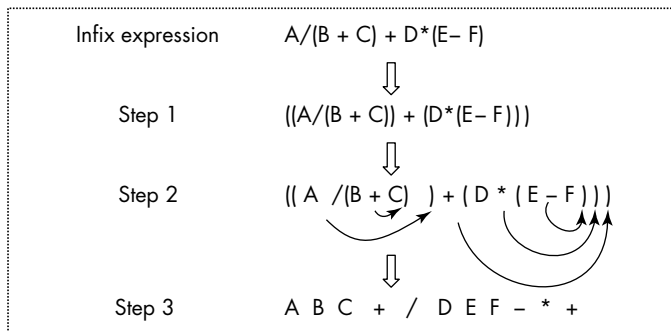


Fig. 4.7 Conversion of infix to postfix expressions

The method discussed above is rather inefficient because it requires following two passes over the expression:

- (1) The first pass to fully parenthesize the expression.
- (2) The second pass to replace the right hand parentheses by their corresponding embedded operators.

It may be noted that the *infix* expression within innermost parentheses should be the first to be converted into *postfix* before expressions in outermost parentheses are picked. This successive elimination of parentheses is done until whole of the expression is converted into *postfix*. This LIFO nature of the 2nd pass suggests that a *stack* can be used for the conversion process.

(2) The stack method In this method, two priorities are assigned to the operators: instack priority and incoming priority as shown in Table 4.5.

■ **Table 4.5** Priorities assigned to operator in the stack method

S. No.	Operator	Instack priority	Incoming priority
1	(, ;	0	6
2)	6	6
3	^	4	5
4	*, /	3	3
5	+, -	2	2

It may be noted that the left hand parenthesis '(' has the highest incoming priority and least instack priority.

Let us assume that an *infix* expression is terminated by the character ';'. We shall call operators and operands of an expression by a general name *element*. The following broader level algorithm `convert()` can be used to convert an *infix* expression to *postfix* expression.

Algorithm `convert()`

```
{
    Step
    1. Scan the expression from left to right.
    2. If the element is an operand, then store the element into the target area.
    3. If the element is an operator, then push it onto a stack with following rules:
        3.1 While the incoming priority is less than or equal to instack priority, pop the operators and store into the target area.
        3.2 If the element is right hand parenthesis ')', then pop all the elements from the stack till left hand parenthesis '(' is popped out. The popped elements are stored in the target area.
    4. If the element is ';', then pop all the elements till the stack is empty and store them into the target area.
    5. Stop
}
```


Example 2: Write a program that uses a stack to convert an infix expression to a postfix expression.

Solution: We would employ the algorithm `convert()`, given above. The following data structures would be used:

The list of allowed binary operators along with their `icp` and `isp` is stored in an array of structures called `opTab[]`. An array called `stack` of type `operator A` is the main data structure. `match()` function is being used to find out as to whether a given element is an operator or an operand depending upon the result (`res`) of `match` operation. The required program is given below:

```
/* This program converts an infix expression to a postfix expression */
```

```
#include <stdio.h>
```

```
#include <conio.h>
```

```
struct operator { /* operator declaration*/
```

```
    char opName;
```

```
    int isp;
```

```
    int icp;
```

```
};
```

```
int push (struct operator stak[], int *top, struct operator val,  
int size);
```

```
struct operator pop (struct operator stak[], int *top);
```

```
void display (struct operator stack[], int top);
```

```
int match(struct operator opTab[8], char element, struct  
operator *op);
```

```
void main()
```

```
{
```

```
    char infix[20];
```

```
    char target[20];
```

```
    struct operator stack[20]; /* the stack declaration */
```

```
    int top = -1; /* initially the stack is empty */
```

```
    int res;
```

```
    char val;
```

```
    int size;
```

```
    int pos;
```

```
    int i;
```

```
    struct operator op, opTemp;
```

```
    struct operator opTab[8] = {'(', 0, 6, /* The operators  
information */
```

```
        ')', 0, 0,
```

```
        '^', 4, 5,
```

```
        '*', 3, 3,
```

```
        '/', 3, 3,
```

```
        '+', 2, 2,
```

```
        '-', 2, 2,
```

```
        ';', 0, -1,
```

```
};
```



```

printf ("\n Enter the size of the infix expression");
scanf ("%d", &size); size--;
printf ("\n Enter the terms of infix expression one by one");
for (i= 0; i <= size; i++)
{fflush(stdin);    /* flush the input buffer */
scanf ("%c", &infix[i]);
}
pos = 0;    /* position in target expression */
for (i=0; i <= size; i++)
{
    res = match(opTab, infix[i], &op); /* find whether operator/operand */
    if (res==0)
    {target[pos] = infix[i];          /* store into target expression */
    pos++;
    }
    else
    {if (top < 0)
    push (stack, &top, op, size);    /* first time Push */
    else
    {opTemp.opName='#';    /* place any value to opTemp */

    if (op.opName == '(')
    { while (opTemp.opName != '(' )
    {
        opTemp = pop (stack, &top);
        if (opTemp.opName != '(' ) /* omit '(' */
        { target [pos] = opTemp.opName;
          pos++;
        }
    }
    }

    else
    {while (stack[top].isp >= op.icp && top >=0)
    {
        opTemp = pop (stack, &top);
        target [pos] = opTemp.opName;
        pos++;
    }
    push(stack, &top, op, size);
    }
    }
    }
}

/* print the postfix expression */

```

```

    printf ("\n The postfix expression is:");
    for (i = 0; i < pos; i++)
    {
        printf ("%c", target[i]);
    }
}

int push (struct operator stack[], int *top, struct operator
val, int size)
{
    if ( *top >= size)
        return 0; /* the stack is full */
    else
    {
        *top= *top + 1;
        stack[*top]= val;
        return 1;
    }
}

struct operator pop (struct operator stack[], int *top)
{struct operator val;
if (*top < 0)
{val.opName = '#';
return val;    /* the stack is empty */
}
else

{
    val = stack[*top];
    *top = *top - 1;
    return val;
}
}

void display (struct operator stack[], int top)
{
    int i;
    printf ("\n The contents of stack are:");
    for (i = top; i >=0; i--)
        printf("%c ", stack[i].opName);
}
int match(struct operator opTab[8], char element, struct operator*op)
{
    int i;
    for (i =0; i <8; i++)

```

```

{
    if (opTab[i].opName == element)
    {
        *op = opTab[i];
        return 1;
    }
}
return 0;
}

```

For following given infix expressions, the above program produces the correct output as shown below:

Infix expression	Postfix expression
(1) $A \wedge B \wedge C + D$;	$A B C \wedge \wedge D +$
(2) $(A - B) * (C/D) + E$;	$A B - C D / * E +$

Example 3: Use a stack to convert the following infix arithmetic expression into a postfix expression. Show the changing status of the stack in tabular form.

$$A + B * C$$

Solution: We would use the algorithm `convert()`. The simulation of this algorithm for the above expression is given in Table 4.6.

The output postfix expression is $ABC* +$

Example 4: Use a stack to convert the following infix arithmetic expression into a postfix expression. Show the changing status of the stack in tabular form.

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

■ **Table 4.6** Conversion of $A + B * C$ into its postfix form

Step	Input element	Action	Stack status	Output (target)
-			empty	
1	A	Write into target	empty	A
2	+	Push	+	A
3	B	Write into target	+	AB
4	*	Push	+ *	AB
5	C	Write into target	+ *	ABC
6	;	Pop, Write into target Pop, Write into target Stop	+ empty	ABC* ABC* +

Solution: We would use the algorithm `convert()`. The simulation of this algorithm for the above expression is given in Table 4.7.

The output postfix expression is $AB - CD / * E +$

■ **Table 4.7** Conversion of $(A - B) * (C/D) + E$ into its postfix form

Step	Input element	Action	Stack status	Output (target)
-			empty	
1	(Push	(
2	A	Write into target	(A
3	-	Push	(-	A
4	B	Write into target	(-	AB
5)	Pop, Write into target Pop	(empty	AB-
6	*	Push	*	AB-
7	(Push	* (AB-
8	C	Write into target	* (AB - C
9	/	Push	* (/	AB - C
10	D	Write into target	* (/	AB - CD
11)	Pop, Write into target Pop	* (*	AB - CD/
12	+	Pop, Write into target Push	Empty +	AB - CD/*
13	E	Write into target	+	AB - CD/* E
14	;	Pop, Write into target	Empty	AB - CD/* E+

Example 5: Use a stack to convert the following infix arithmetic expression into a postfix expression. Show the changing status of the stack in tabular form.

$$X + Y * Z \wedge P - (X/Y + Z)$$

Solution: We would use the algorithm `convert()`. The simulation of this algorithm for the above expression is given in Table 4.8.

The output postfix expression is $XYZP^{\wedge} * + XY/Z + -$

Note: Since a postfix expression has no parentheses, its evaluation becomes a very easy exercise. In fact, a stack can be suitably used for this purpose.

4.2.1.4 Evaluation of Postfix Expressions A stack can be conveniently used for the evaluation of postfix expressions. The expression is read from left to right. Each encountered element is examined. If the element is an operand, then the element is pushed on to the stack. If the element is an operator, then the two operands are popped from the stack and the desired operation is done. The result of the operations is again pushed onto the stack. This process is repeated till the end of the expression (i.e., ';') is encountered. The final result is popped from the stack.

An algorithm for the evaluation of a postfix expression is given below:

Algorithm `evalPostfix()`

```
{
    Step
    1. pick an element from the postfix expression
```

■ **Table 4.8** Conversion of $X + Y * Z \wedge P - (X / Y + Z)$

Step	Input element	Action	Stack status	Output (target)
-			empty	
1	X	Write into target		X
2	+	Push	+	X
3	Y	Write into target	+	X Y
4	*	Push	+ *	X Y
5	Z	Write into target	+ *	X Y Z
6	^	Push	+ * ^	X Y Z
7	P	Write into target	+ * ^	X Y Z P
8	-	Pop, Write into target Pop, Write into target Pop, Write into target Push	+ * + - -	X Y Z P ^ X Y Z P ^ * XYZP ^ * +
9	(Push	- (XYZP ^ * +
10	X	Write into target	- (XYZP ^ * + X
11	/	Push	- (/	XYZP ^ * + X
12	Y	Write into target	- (/	XYZP ^ * + XY
13	+	Pop, Write into target Push	- (- (+	XYZP ^ * + XY /
14	Z	Write into target	- (+	XYZP ^ * + XY / Z
15)	Pop, Write into target Pop	- (-	XYZP ^ * + XY / Z + XYZP ^ * + XY / Z +
16	;	Pop, Write into target Stop	empty	XYZP ^ * + XY / Z + -

```

2. if (element is an operand)
    if ( element != ';' )
        {push element on the stack}
    else
    {
        pop result from the stack;
        print the result;
        exit to step 4;
    }
else
{
    pop operand1 from stack;
    pop operand2 from stack;
    perform the operation;
    push result on the stack;
}

```

```

    }
    3. repeat steps 1 and 2
    4. stop
}

```

Example 6: Write a program that uses a stack to evaluate a postfix expression.

Solution: The algorithm `evalPostfix()` and the following data structures would be used for the required program. The postfix expression will be stored in an array called `postfix[]` of following structure type.

```

struct term    /* structure to store an element */
{
    char element;
    float val;
};
where

```

‘element’ stores an operand or an operator.

‘val’ stores the actual value of the operand or 0 for an operator.

For instance, the expression `ABC*+;` for values `A = 10, B = 15, C = 8` will be stored in `postFix[]` as shown below:

`postFix`

A	B	C	*	–	;
10	15	8	0	0	0

An array called `stack` of type `float` would also be used.

The required program is given below:

```

/* This program uses a stack to evaluate a postfix expression */

#include <stdio.h>
#include <conio.h>

struct term    /* structure to store an element */
{
    char element;
    float val;
};

/* array to store the postfix expression*/

int push (float stack[], int *top, float val, int size);
float pop (float stack[], int *top);

void main()
{
    float stack[20];
    struct term postFix[20];

```



```
int top = -1;
int i;
int size;
float result;
int temp;

float op1, op2; /* the operands */
printf ("\n Enter the number of elements in the postfix expression");
scanf ("%d", & size);

printf ("\n Enter the expression as element value pair");
printf ("\n Enter 0 for an operator including ';'");

for (i = 0; i<size; i++)
{
    printf ("\n Enter element - val pair");
    fflush (stdin);
    scanf ("%c %f", &postFix[i]. element, &postFix[i].val);
}
for (i=0; i< size; i++)
{
    switch (postFix[i].element)
    {
        case '+' : op2 = pop (stack, &top);
                    op1 = pop (stack, &top);
                    result = op1 + op2;
                    temp = push (stack, &top, result, size);
                    if (temp ==0) printf("\n stack full");
                    break;
        case '*' : op2 = pop (stack, &top);
                    op1 = pop (stack, &top);
                    result = op1*op2;
                    temp = push (stack, &top, result, size);
                    if (temp ==0) printf("\n stack full");
                    break;
        case '-' : op2 = pop (stack, &top);
                    op1 = pop (stack, &top);
                    result = op1 - op2;
                    temp= push (stack, &top, result, size);
                    if (temp ==0) printf("\n stack full");
                    break;
        case '/' : op2 = pop (stack, &top);
                    op1 = pop (stack, &top);
                    result = op1/op2;
                    temp = push (stack, &top, result, size);
```



```

        if (temp ==0) printf("\n stack full");
        break;
    case ';' : result = pop (stack, &top);
        printf ("\n The result = %8.2f ", result);
        break;
    default: temp= push (stack, &top, postFix[i].val,size);
        if (temp ==0) printf("\n stack full");
    }
}
}
int push (float stack[], int *top, float val, int size)
{
    if (*top < size)
    {
        *top = *top +1;
        stack [*top] = val;
        return 1;
    }
    else
        return 0;
}
float pop (float stack[], int *top)
{
    float term;
    term = 0;
    if (*top >=0)
    {
        term = stack[*top];
        *top=*top-1;
        return term;
    }
    else
        return term;
}

```

For following given input:

```

A 8
B 6
C 20
* 0
- 0
; 0

```

the program gives the following output:

The Result = -112.00

Example 7: Use a stack to evaluate the following postfix arithmetic expression. Show the changing status of the stack in tabular form.

$$A B C * - \text{ for } A = 8, B = 6, C = 20$$

Solution: We would use the algorithm `evalPostfix()`. The simulation of this algorithm for the above expression is given in Table 4.9.

■ **Table 4.9** Evaluation of $A B C * -$

Step	Input element	Action	Stack status
-			empty
1	A 8	Push	8
2	B 6	Push	8 6
3	C 20	Push	8 6 20
4	* 0	Pop (20) Pop(6) Result = $20 * 6 = 120$ Push	8 6 8 8 120
5	- 0	Pop (120) Pop(8) Result = $8 - 120 = -112$ Push	8 -112
6	; 0	Pop (-112) Print result	empty

Example 8: Use a stack to evaluate the following postfix arithmetic expression. Show the changing status of the stack in tabular form.

$$X Y Z P ^ * + A B / C + - \text{ for } X = 1, Y = 5, Z = 2, P = 3, A = 15, B = 3, C = 8$$

Solution: We would use the algorithm `evalPostfix()`. The simulation of this algorithm for the above expression is given in Table 4.10.

■ **Table 4.10** Evaluation of $X Y Z P ^ * + A B / C + -$

Step	Input element	Action	Stack status
-			empty
1	X 1	Push	1
2	Y 5	Push	1 5
3	Z 2	Push	1 5 2
4	P 3	Push	1 5 2 3
5	^ 0	Pop(3) Pop(2) Result = $2 ^ 3 = 8$ Push	1 5 2 1 5 1 5 8

(continued)

■ **Table 4.10** (continued)

Step	Input element	Action	Stack status
6	* 0	Pop (8) Pop(5) Result = $5 * 8 = 40$ Push	1 5 1 1 40
7	+ 0	Pop (40) Pop(1) Result = $1 + 40$ = 41 Push	1 41
8	A 15	Push	41 15
9	B 3	Push	41 15 3
10	/ 0	Pop (3) Pop(15) Result = $15 / 3 = 5$ Push	41 15 41 41 5
11	C 8	Push	41 5 8
12	+ 0	Pop (8) Pop(5) Result = $5 + 8 = 13$ Push	41 5 41 41 13
13	- 0	Pop (13) Pop(41) Result = $41 - 13$ = 28 Push	41 28
14	; 0	Pop (28) Print result	

4.3 QUEUES

A queue is the most common situation in this world where the first person in the line is the person to be served first; the new comers join at the end. We find that in this kind of arrangement, all additions are made at one end and the deletions made at the other. The end where all additions are made is called the **rear** end. The other end from where the deletions are made is called the **front** end as shown in Figure 4.8.

The queue is also a linear data structure of varying size in the sense that the size of a queue depends upon the number of items currently present in it. With additions and/or deletions, the size of the queue increases or decreases, respectively. Therefore, the queue is also a dynamic data structure with a capacity to enlarge or shrink.

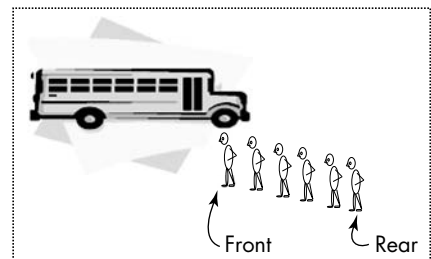


Fig. 4.8 The queue

Consider the queue of processes shown in Figure 4.9. The operating system maintains a queue for scheduled processes that are ready to run. The dispatcher picks the process from the front end of the queue and dispatches it to CPU and a new process joins at the rear end of the queue.

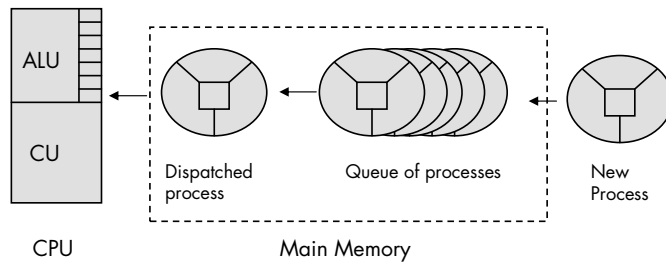


Fig. 4.9 Queue of processes maintained in the main memory

It may be noted that when the new processes join in, the size of the queue increases whereas the size of the queue will reduce when processes finish their job and leave the system.

4.3.1 Queue Operations

A queue, being a linear data structure, can be comfortably mapped on a one-dimensional array. Let us consider a queue maintained in an array called `Queue[N]`, shown in Figure 4.10. A variable called `Front` is pointing to the logical front end of the queue from where the removal or deletion of an element takes place. Currently, `Front` is pointing to an empty location in the queue. Similarly, a variable called `Rear` is pointing to the logical rear end of the queue where the new elements can be added. Currently, `Rear` is pointing to a location containing the item 'G'.

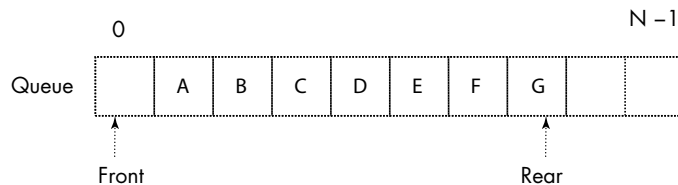


Fig. 4.10 An array representation of a queue

Let us now add an item 'H' into the queue. Before the item is added into the queue, the `Rear` is incremented by one to point to the next vacant location as shown in Figure 4.11. The new item 'H' is added at location currently being pointed by `Rear`.

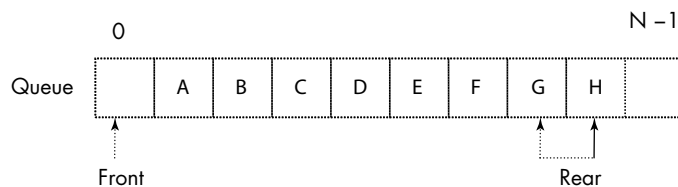


Fig. 4.11 Addition of the item 'H' at the rear end

It may be noted that if more additions are made on the queue, then a stage would come when $\text{Rear} = N - 1$, i.e., last location of the array. Now, no more additions can be performed on the queue. This condition ($\text{Rear} = N - 1$) is called **Queue-Full**.

Let us now delete an item from the queue. Before an item is deleted or removed, the **Front** is incremented by one to point to the next location, i.e., the location containing item 'A'. The item (i.e., 'A') is then deleted as shown in Figure 4.12.

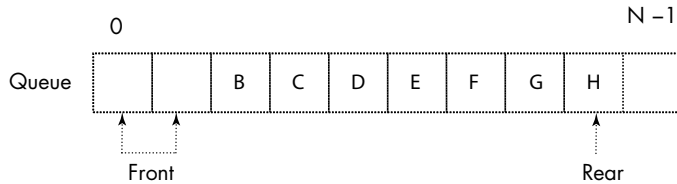


Fig. 4.12 Deletion of the item 'A' from the front end

It may be noted that if more deletions are done on this queue, then a stage would reach when there will be no items on the queue, i.e., $\text{Front} = \text{Rear}$ as shown in Figure 4.13. This condition ($\text{Front} = \text{Rear}$) is called **Queue_empty**.

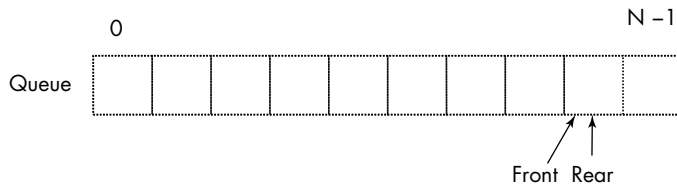


Fig. 4.13 No item in the queue

The algorithms for addition and deletion operations on the queue are given below. These algorithms use an array called `Queue[N]` to represent a queue of size N locations. A variable called **Front** keeps track of the front of the queue, i.e., the location from where deletions are made. Another variable called **Rear** keeps track of the rear end of the queue, i.e., the location where additions are made. Another variable called **item** is used to store the item to be added or deleted from the queue.

The algorithm for addition is given below:

```
Algorithm addQ ( )
{
    Step
    1. if ( $\text{Rear} \geq N$ ) then {prompt ("Queue Full"); exit}
    2.  $\text{Rear} = \text{Rear} + 1$ ;
    3.  $\text{Queue}[\text{Rear}] = \text{item}$ ;
    4. stop
}
```

It may be noted that the item is added to the queue (i.e., addition operation) provided the queue is not already full, i.e., it is checked at step1 whether or not the queue has room for another item.

The algorithm for deletion is given below:

Algorithm `delQ()`

```
{
    Step
    1. if (Front == Rear) then prompt ("Queue Empty");
       else
       Front = Front + 1
       Item = Queue [Front];
    2. stop
}
```

Note that in the above algorithm, before removing an item from the queue, it is checked at step 1 to see if there is at least one item on the queue that can be removed.

The performance of the queue data structure for 'n' elements is given below:

- The space complexity is $O(n)$.
- The time complexity of each operation is $O(1)$.
- The size of the queue, implemented using array, must be defined in advance, i.e., *a priori*. Moreover, the size cannot be changed.

Example 9: Write a program in 'C' that uses the following menu to simulate the queue operations on items of `int` type.

Menu – Queue operations	
Add an item	1
Delete an item	2
Display Queue	3
Quit	4

Enter your choice:

Solution: The above defined algorithms `addQ()` and `delQ()` would be employed to write the program. The menu would be created using `printf()` statements and displayed within a do-while loop. In fact, a do-while loop is the most appropriate loop for display and manipulations of menu items.

Note:

- The `addQ()` function would return 1 if the operation is successful and 0 otherwise, i.e., 0 indicates that the queue was full.
- The `delQ()` function would return the deleted value if the operation is successful and 9999 otherwise, i.e., the unusual value '9999' indicates that the queue was empty.

The required program is given below:

```
/* This program simulates queue operations */
#include <stdio.h>
#include <conio.h>

int addQ (int Queue[], int *Rear, int val, int size);
int delQ (int Queue[], int *Front, int *Rear);
```

```

void display (int Queue[], int Front, int Rear);
void main()
{
    int Queue[20];    /* the queue declaration */
    int Front;
    int Rear;

    int val;
    int size;
    int choice;
    int result;
    Front = Rear=0;    /* initially the queue set to be empty */
    printf("\n Enter the size of the queue");
    scanf ("%d", & size);
    size--;    /* adjusted for index = 0 */
               /* create menu */

    do
    {
        clrscr();
        printf ("\n Menu - Queue Operations");
        printf ("\n Add                1");
        printf ("\n delete            2");
        printf ("\n Display queue    3");
        printf ("\n Quit              4");

        printf ("\n Enter your choice:");
        scanf ("%d", & choice);

        switch (choice)
        {
            case 1: printf ("\n Enter the value to be added");
                     scanf ("%d", & val);
                     result = addQ(Queue, &Rear, val, size);
                     if (result == 0)
                         printf ("\n The queue full");
                     break;
            case 2: result = delQ(Queue, &Front, &Rear);
                     if (result == 9999)
                         printf ("\n The queue is Empty");
                     else
                         printf ("\n The deleted value = %d", result);
                     break;
            case 3: display (Queue, Front, Rear);
                     break;
        }
        printf("\n\n Press any key to continue");
        getch();
    }
}

```

```

    }
    while (choice != 4);
}

int addQ (int Queue[], int *Rear, int val, int size)
{
    if ( *Rear >= size)
        return 0;    /* the queue is Full */
    else
    {
        *Rear= *Rear +1;
        Queue[*Rear] = val;
        return 1;
    }
}

int delQ (int Queue[], int *Front, int *Rear)
{
    int val;
    if (*Front == *Rear)
        return 9999;    /* the queue is empty */
    else
    {
        *Front = *Front + 1;
        val = Queue[*Front];

        return val;
    }
}

void display (int Queue[], int Front, int Rear)
{
    int i;
    printf ("\n The contents of queue are:");
    for (i = Front+1; i <= Rear; i++)
        printf ("%d ", Queue[i]);
}

```

Note:

- (1) The drawback of a linear queue is that a stage may arrive when both Front and Rear are equal and point to the last location of the array as shown in Figure 4.14.

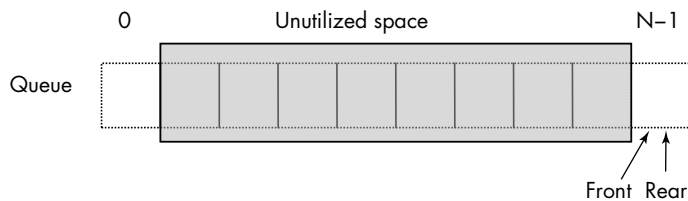


Fig. 4.14 The queue is empty as well as full

Now, the queue is useless because the queue is full ($\text{Rear} \geq N - 1$) as well as empty ($\text{Front} = \text{Rear}$). Thus, no additions and deletions can be performed. This situation can be handled by resetting Front and Rear to 0th location, i.e., $\text{Front} = \text{Rear} = 0$.

- (2) The major drawback of linear queue is that at a given time, all the locations to the left of *Front* are always vacant and unutilized.

4.3.2 Circular Queue

The drawback of the linear queues, as discussed above, can be removed by making the queue spill over the elements from last position of array to the 0th element. This arrangement of elements makes the queue as *circular queue* wherein as soon as Rear or Front exceeds $N - 1$, it is set to 0. Now, there is no end of the queue. The Rear and Front will move around in the queue in the circle and no locations shall be left vacant.

Let us add an item 'C' to the queue of Figure 4.14. The Rear is pointing to location $N-1$. After incrementing Rear (i.e., $\text{Rear} = \text{Rear} + 1 = N - 1 + 1 = N$), we find that it is pointing to a location equal to N , which does not exist. Therefore, Rear is set to location 0, the beginning of the array and the item 'C' is stored at this location pointed by Rear as shown in Figure 4.15.

Now, this arrangement ensures that there will be no space left unutilized in the queue. For instance, let us add one by one the items 'I', 'R', 'C', 'U', 'L', 'A' and 'R' to the queue (see Figure 4.16). We find that the locations left of Front are now being filled, removing the drawback of linear queues.

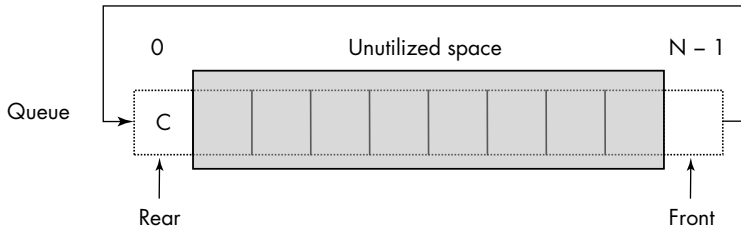


Fig. 4.15 The rear is made to move to the 0th location.

It may be noted that after more additions, a stage will come when Rear becomes equal to Front ($\text{Rear} = \text{Front}$), indicating that the queue is full.

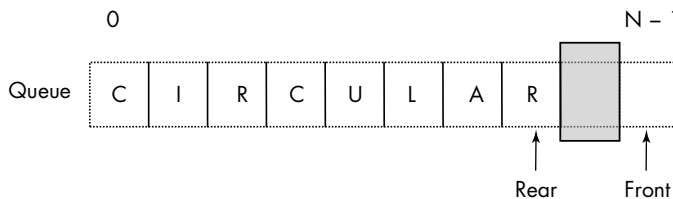


Fig. 4.16 More additions to circular queue

Let us now delete an element from the queue. The Front is pointing to location $N-1$. After incrementing Front (i.e., $\text{Front} = \text{Front} + 1 = N - 1 + 1 = N$), we find that it is pointing to a location equal to N , which does not exist. Therefore, Front is set to location 0, the beginning of the array and the item 'C' is deleted from this location pointed by Front as shown in Figure 4.17.

It may be noted that after more deletions, a stage will come when Front becomes equal to Rear ($\text{Front} = \text{Rear}$), indicating that the queue is empty.

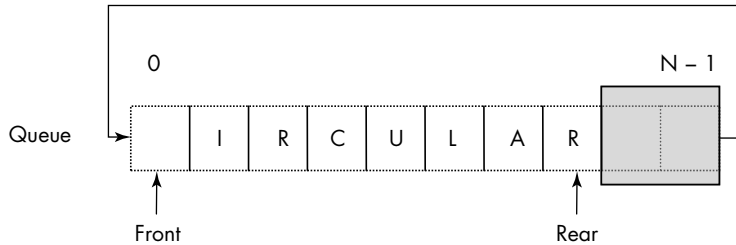


Fig. 4.17 For deletion, Front is set to 0th location

It may be further noted that a very interesting situation has turned up, i.e., for both events—queue 'full' and 'empty'—the condition to be tested is ($\text{Rear} = \text{Front}$). This conflict can be resolved by the following decisions:

- The queue empty condition remains the same, i.e., ' $\text{Front} == \text{Rear}$ '.
- In case of queue full, the condition ' $\text{Rear} + 1 == \text{Front}$ ' would be tested.
- In this arrangement one location, being pointed by Front, shall remain vacant.
- The following formulae would be used for automatic movement of the variables 'Front' and 'Rear' within the circular queue:

$$\text{Rear} = (\text{Rear} + 1) \bmod N$$

$$\text{Front} = (\text{Front} + 1) \bmod N$$

where N is the size of the queue.

The algorithms for addition and deletion operations in a circular queue are given below. These algorithms use an array called `cirQ[N]` to represent a queue of size N locations. A variable called Front keeps track of the front of the queue, i.e., the location from where deletions are made. Another variable called Rear keeps track of the rear end of the queue, i.e., the location where additions are made. Another variable called item is used to store the item to be added or deleted from the queue.

The algorithm for addition is given below:

Algorithm `addQ()`

```
{
    Step
    1. if ( ((Rear + 1) % N) == Front)
        then { prompt ("Queue Full"); exit}
    2. Rear = (Rear + 1) % N;
    3. cirQ [Rear] = item;
    4. stop
}
```

It may be noted that the item is added to the queue (i.e., addition operation) provided the queue is not already full, i.e., at step1 it is checked whether or not the queue has room for another item.

The algorithm for deletion is given below:

Algorithm delQ()

```
{
    Step
    1. if (Front == Rear) then prompt ("Queue Empty");
        else
            Front = (Front + 1) % N
            Item = Queue [Front];
            return Item;
    2. stop
}
```

Note that in the above algorithm, before removing an item from the queue, it is checked at step 1 to see if there is at least one item on the queue that can be removed.

Example 10: Write a program in 'C' that uses the following menu to simulate the circular queue operations on items of int type.

Menu – Circular Queue operations	
Add an item	1
Delete an item	2
Display Queue	3
Quit	4

Enter your choice:

Solution: The above defined algorithms addQ() and delQ() for circular queue would be employed to write the program. The menu would be created using printf () statements and displayed within a do-while loop.

Note:

- The addQ() function would return 1 if the operation is successful and 0 otherwise, i.e., 0 indicates that the queue was full.
- The delQ() function would return the deleted value if the operation is successful and 9999 otherwise, i.e., the unusual value '9999' indicates that the queue was empty.

The required program is given below:

```
/* This program simulates Circular Queue operations */
#include <stdio.h>
#include <conio.h>

int addQ (int cirQ[],int *Front, int *Rear, int val, int size);
int delQ (int cirQ[], int *Front, int *Rear, int size);
void display (int cirQ[], int Front, int Rear, int size);
```

```
void main()
{
    int cirQ[20];    /* the queue declaration */
    int Front;
    int Rear;

    int val;
    int size;
    int choice;
    int result;
    Front = Rear=0;  /* initially the queue set to be empty */
    printf ("\n Enter the size of the queue");
    scanf ("%d", & size);
    /* create menu */
    do
    {
        clrscr();
        printf ("\n Menu – Circular queue operations");
        printf ("\n Add          1");
        printf ("\n Delete       2");
        printf ("\n Display queue  3");
        printf ("\n Quit          4");

        printf ("\n Enter your choice:");
        scanf ("%d", & choice);

        switch (choice)
        {
            case 1: printf ("\n Enter the value to be added");
                     scanf ("%d", & val);
                     result = addQ(cirQ, &Front, &Rear, val, size);
                     if (result == 0)
                         printf ("\n The queue full");
                     break;
            case 2: result = delQ(cirQ, &Front, &Rear, size);
                     if (result == 9999)
                         printf ("\n The queue is Empty");
                     else
                         printf ("\n The deleted value = %d", result);
                     break;
            case 3: display (cirQ, Front, Rear, size);
                     break;
        }
        printf("\n\n Press any key to continue");
        getch();
    }
}
```

```

    while (choice != 4);
}

int addQ (int cirQ[],int * Front, int *Rear, int val, int size)
{
    if ( ((*Rear + 1) % size) == *Front)
        return 0; /* the queue is Full */
    else
    {
        *Rear= (*Rear + 1) % size;
        cirQ[*Rear]= val;
        return 1;
    }
}

int delQ (int cirQ[], int *Front, int *Rear, int size)
{int val;
 if (*Front == *Rear)
 return 9999; /* the queue is empty */
 else
 {
     *Front =( *Front + 1) % size;
     val = cirQ[*Front];

     return val;
 }
}

void display (int cirQ[], int Front, int Rear, int size)
{
    int i;
    printf ("\n The contents of queue are:");
    i=Front;
    while ( i != Rear)
    {
        i = (i + 1) % size;
        printf ("%d ", cirQ[i]);
    }
}

```

It may be noted that the circular queue has been implemented in a one-dimensional array, which is linear by nature. We only view the array to be a circle in the sense that it is imagined that the last location of the array is immediately followed by the 0th location as shown in Figure 4.18.

The shaded area shows the queue elements. When the queue is full, then it satisfies the condition: $(Rear + 1) \% N == Front$. The queue empty condition is indicated by the condition: $Front == Rear$. However, at least one location remains vacant all the time.

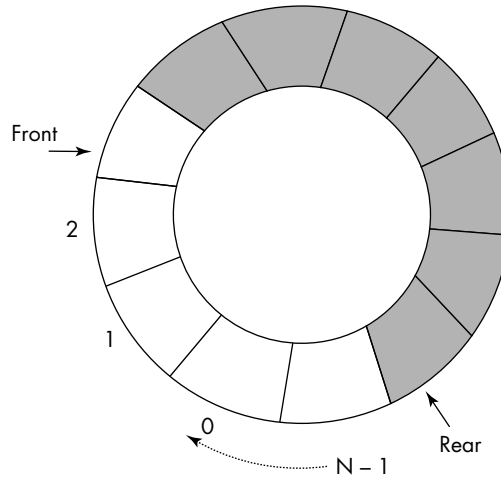


Fig. 4.18 The circular queue

4.3.3 Priority Queue

Sometimes, a queue does not operate on strictly first in first out (FIFO) basis. On the contrary, it stores the elements according to the priority associated with each stored element. In a post office, the airmail has a higher priority than the speed post and speed post in turn has more priority than an ordinary mail. Therefore, the post office makes a queue of mail ordered by their priority, i.e., all airmail letters are placed at the head of the queue followed by speed post letters. The speed post letters are followed by registered parcels and so on. The ordinary mail with least priority lies at the end of the queue.

Let us take an example from operating system. A priority scheduling policy would store the processes as per their priority in a queue. The scheduler will add a new process to the queue according to its priority and the highest priority process would be served first as shown in Figure 4.19.

In fact, the operating system will have to maintain such priority queues at all non-shareable devices such as printers, plotters, etc.

Note: Since every element in the queue is ordered on priority, an entry into the priority queue has to be a pair (e, p) where 'e' is the name of the element and 'p' is its priority. In case of records of items, the 'e' is chosen to be a field or key of the record.

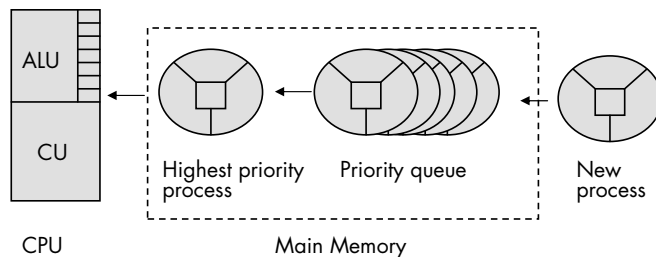


Fig. 4.19 The priority queue maintained by operating system

Consider the job pool given in Table 4.11. The processes opened by the operating system have been assigned priorities.

Now, the scheduler shall create a priority queue that can store a list of pairs (Process No., priority). It has two possible choices to store the pairs as shown in Figure 4.20.

Choice (a): The addition operation simply adds the newly arrived job at the rear end, i.e., in the order of its arrival. This operation is $O(1)$.

The deletion operation requires the process with higher priority to be searched and removed, P7 in this case. This operation is $O(n)$ for n elements in the queue as we have to traverse the queue to find the element.

Table 4.11 Job pool	
Process No.	Priority
P8	2
P5	1
P7	9
P4	6
P2	5

(P8, 2)	(P5, 1)	(P7, 9)	(P4, 6)	(P2, 5)
---------	---------	---------	---------	---------

(a) Jobs stored in the order of their arrival

(P7, 9)	(P4, 6)	(P2, 5)	(P8, 2)	(P5, 1)
---------	---------	---------	---------	---------

(b) Jobs stored in the order of their priority

Fig. 4.20 The choices to store the job pool

Choice (b): The addition operation requires the process with higher priority to be inserted at its proper position in the queue. This operation is $O(n)$ for n elements in the queue as the location has to be found where the insertion can take place.

The deletion operation simply removes the process from the front of the queue because the process with highest priority is already at the front of the queue. This operation is $O(1)$.

Let us now use the choice (b) to implement a priority queue for processes called 'proc' of following structure type

```
struct proc
{
    char process [3];
    int priority;
};
```

The following menu would be used to interact with the users:

Menu – Priority Queue operations	
Add an item	1
Delete an item	2
Display Queue	3
Quit	4

Enter your choice:

A function called `addPq()` inserts an *i*th process '*pi*' with priority '*pr*' at its proper position in the priority queue represented using the array `pQ[N]` where *N* is the size of the queue. A variable '*Rear*' points to rear end of the priority queue, i.e., the least priority process.

A function called `delPq()` removes a process from the current '*Front*' of the priority queue.

The required program is given below:

```
/* This program simulates priority queue operations */

#include <stdio.h>
#include <conio.h>

struct proc{
    char process[3];
    int priority;
};

int addPq (struct proc pQ[], int *Rear, struct proc val, int size);
int delPq (struct proc pQ[], int *Front, int *Rear, struct proc *val);
void display (struct proc pQ[], int Front, int Rear);
void main()
{
    struct proc pQ[20]; /* the priority queue declaration */
    int Front;
    int Rear;

    struct proc val;
    int size;
    int choice;
    int result;
    Front = Rear=0; /* initially the queue set to be empty */
    printf ("\n Enter the size of the queue");
    scanf ("%d", & size);
                                /* create menu */
    do
    {
        clrscr();
        printf ("\n Menu - Priority queue operations");
        printf ("\n Add          1");
        printf ("\n Delete        2");
        printf ("\n Display queue   3");
        printf ("\n Quit           4");

        printf ("\n Enter your choice:");
        scanf ("%d", & choice);

        switch (choice)
        {
```



```

        case 1: printf ("\n Enter the process number and its priority to
                    be added");
                scanf ("%s %d", & val.process, &val.priority);
                result = addPq( pQ, &Rear, val, size);
                if (result == 0)
                {printf ("\n The queue full");}
                break;
        case 2: result = delPq(pQ, &Front,& Rear, &val);
                if (result == 0)
                printf ("\n The queue is empty");
                else
                printf ("\n The deleted proc = %s with priority %d", val.
                    process, val.priority );
                break;
        case 3: display (pQ, Front, Rear);
                break;
    }
    printf("\n\n Press any key to continue");
    getch();
}
while (choice != 4);
}
int addPq (struct proc pQ[], int *Rear, struct proc val, int size)
{
    int i;
    if ((*Rear + 1)>= size)
        return 0; /* the queue is Full */
    else
    {
        *Rear= (*Rear + 1);
        i=*Rear - 1;
        while (val.priority > pQ[i].priority)
        {
            pQ[i+1] =pQ[i];
            i = i - 1;
        }
        i++;
        pQ[i] = val;
    }
    return 1;
}
int delPq (struct proc pQ[], int *Front, int *Rear, struct proc *val)
{
    if (*Front == *Rear)
        return 0; /* the queue is empty */
    else

```

```

{
    *Front = (*Front + 1);
    *val = pQ[*Front];

    return 1;
}
}
void display ( struct proc pQ[], int Front, int Rear)
{
    int i;
    printf ("\n The contents of queue are:");
    i=Front;
    while ( i != Rear)
    {
        i = i + 1;
        printf ("%s , %d ", pQ[i].process, pQ[i].priority);
    }
}

```

It may be noted that in order to keep it simple, the priority queue has been implemented as a linear queue. A sample input is given below:

Enter the process number and its priority to be added P4 7

where 'P4' is the name of the process and '7' is its priority.

4.3.4 The Deque

A deque is a queue that allows additions and deletions from both ends (see Figure 4.21). This data structure is suitable for implementing the 'redo and undo' operations of software such as word processor, paint program, drawing tools, etc.

A one-dimensional array can be suitably used to implement a deque wherein the following four operations need to be designed:

- (1) addRear(): adds an item at the rear end
- (2) delRear(): removes an item from the rear end
- (3) addFront(): adds an item at the front end
- (4) delFront(): removes an item from the front end

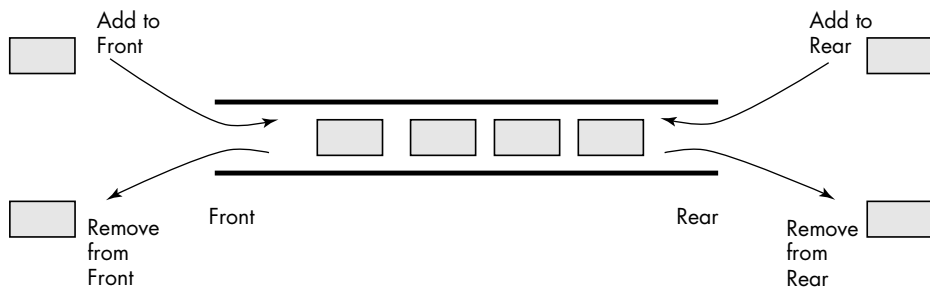


Fig. 4.21 The deque

Some programmers view a deque as two stacks connected back to back as shown in Figure 4.22.

For deque arrangement shown in Figure 4.22, the following four operations need to be designed:

- (1) `pushRear()`: adds an item at the rear end
- (2) `popRear()`: removes an item from the rear end
- (3) `pushFront()`: adds an item at the front end
- (4) `popFront()`: removes an item from the front end

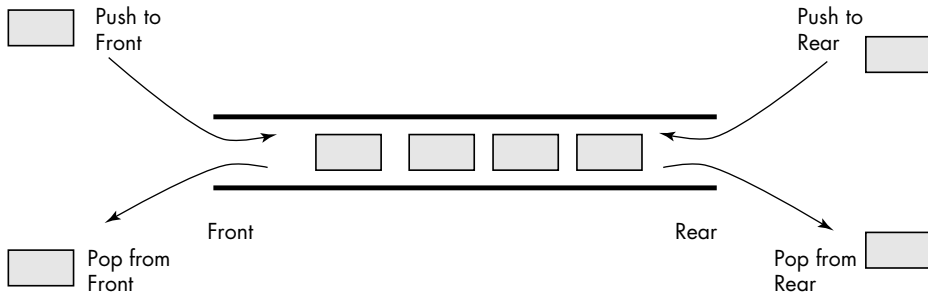


Fig. 4.22 The deque (stack view)

Example 11: Write a program in 'C' that uses the following menu to simulate the deque operations on items of `int` type.

Menu – deque Queue operations	
Add item at Rear	1
Add item at Front	2
Delete item from Rear	3
Delete item from Front	4
Display Queue	5
Quit	6

Enter your choice:

Solution: A one-dimensional array called `dQ[]` would be used to implement the deque wherein the following four functions would be used. The algorithms for these functions are same as used in a linear queue.

- (1) `addRear()`: adds an item at the rear end
- (2) `delRear()`: removes an item from the rear end
- (3) `addFront()`: adds an item at the front end
- (4) `delFront()`: removes an item from the front end

The menu would be created using `printf()` statements and displayed within a `do-while` loop.

The required program is given below:

```
/* This program simulates deQueue operations */

#include <stdio.h>
#include <conio.h>

int addRear (int dQ[], int *Rear, int val, int size);
int addFront (int dQ[], int *Front, int val);

int delRear (int dQ[], int *Rear, int *Front, int *val);
int delFront (int dQ[], int *Front, int *Rear, int *val);

void display (int dQ[], int Front, int Rear);

void main()
{
    int dQ[20]; /* the queue declaration */
    int Front;
    int Rear;

    int val;
    int size;
    int choice;
    int result;
    Front = Rear=0; /* initially the queue set to be empty */
    printf("\n Enter the size of the queue");
    scanf ("%d", & size);
    size--; /* adjusted for index = 0 */
    /* create menu */

    do
    {
        clrscr();
        printf ("\n Menu - dQue Operations");
        printf ("\n Add item at Rear          1");
        printf ("\n Add item at Front          2");
        printf ("\n Delete item from Rear          3");
        printf ("\n Delete item from Front          4");
        printf ("\n Display deque          5");
        printf ("\n Quit          6");

        printf ("\n Enter your choice:");
        scanf ("%d", & choice);

        switch (choice)
        {
            case 1: printf ("\n Enter the value to be added");
                    scanf ("%d", & val);
                    result = addRear(dQ, &Rear, val, size);
```

```

        if (result == 0)
            printf ("\n The queue is full");
        break;
    case 2: printf ("\n Enter the value to be added");
            scanf ("%d", & val);
            result = addFront(dQ, &Front, val);
            if (result == 0)
                printf ("\n The queue is full");
            break;
    case 3: result =delRear(dQ, &Rear,& Front, &val);
            if (result == 0)
                printf ("\n The queue is empty");
            else
                printf ("\n The deleted value = %d", val);

            break;
    case 4: result =delFront(dQ, &Front, &Rear, &val);
            if (result == 0)
                printf ("\n The queue is empty");
            else
                printf ("\n The deleted value = %d", val);

            break;
    case 5: display (dQ, Front, Rear);
            break;
    }
    printf("\n\n Press any key to continue");
    getch();
}
while (choice != 6);
}

int addRear(int dQ[], int *Rear, int val, int size)
{
    if (*Rear >= size)
        return 0;    /* the queue is Full */
    else
    {
        *Rear= *Rear + 1;
        dQ[*Rear]= val;
        return 1;
    }
}

int addFront (int dQ[], int *Front, int val)
{

```

```

        if (*Front <= 0)
            return 0;    /* the queue is full */
        else
        {
            dQ[*Front] = val;
            *Front = *Front - 1;
            return 1;
        }
    }

int delRear (int dQ[], int *Rear, int *Front, int *val)
{
    if ( *Rear==*Front)
        return 0;    /* the queue is empty */
    else
    {
        *val = dQ[*Rear];
        *Rear= *Rear - 1;
        return 1;
    }
}

int delFront (int dQ[], int *Front, int *Rear, int *val)
{
    if (*Front == *Rear)
        return 0;    /* the queue is empty */
    else
    {
        *Front = *Front + 1;
        *val = dQ[*Front];

        return 1;
    }
}

void display (int Queue[], int Front, int Rear)
{
    int i;
    printf ("\n The contents of queue are:");
    for (i = Front+1; i <= Rear; i++)
        printf ("%d ", Queue[i]);
}

```

Example 12: The items can be iteratively removed from both ends of a deque and compared with each other. Therefore, the deque can be easily used to determine whether a given string is a **palindrome** or not. Write a program in 'C' that uses a deque to determine whether a given string is a palindrome or not.

Solution: An array called `String[N]` would be used to store the string. As one location in a queue necessarily remains vacant, an array called `dQ[N+1]` would be used to implement a deque. The functions developed in Example 11 would be used to do the required task, i.e., to determine whether the input string is palindrome or not.



The required program is given below:

```
/* This program uses deQue to determine whether a given string is
a palindrome or not */

#include <stdio.h>
#include <conio.h>

int addRear (char dQ[], int *Rear, char val, int size);
int delRear (char dQ[], int *Rear, int *Front, char *val);
int delFront (char dQ[], int *Front, int *Rear, char *val);
int addFront (char dQ[], int *Front, char val);

void display (char dQ[], int Front, int Rear);
void main()
{
    char String[10];
    char dQ[11]; /* the queue declaration */
    int Front;
    int Rear;

    char val, val1, val2;
    int i, size;
    int flag;
    int result;
    printf ("\n Enter the string");
    scanf ("%s", String);

    i = 0;
    size = 10; /* adjusted for index = 0 */
    Front = Rear = 0; /* initially deque is empty */

    while ( String[i] != '\0') /* Add the string to deque */
    {
        result = addRear( dQ, &Rear, String[i], size);
        if (result == 0)
            printf ("\n The queue full");
        i++;
    }

    size = i - 1;
    flag = 1; /* Assuming that the string is
while ( flag && result)
```

```
{
    result = delRear(dQ, &Rear, &Front, &val1);

    result = delFront(dQ, &Front, &Rear, &val2);

    if (val1 != val2) {flag = 0; break;} /* string is not a palindrome */
}

if (flag==1) printf ("\n The string is palindrome");
else printf ("\n The string is not a palindrome");
}

int addRear(char dQ[], int *Rear, char val, int size)
{
    if ( *Rear >= size)
        return 0; /* the queue is full */
    else
    {
        *Rear= *Rear + 1;
        dQ[*Rear]= val;
        return 1;
    }
}

int addFront (char dQ[], int *Front, char val)
{
    if (*Front <= 0)
        return 0; /* the queue is full */
    else
    {
        dQ[*Front] = val;
        *Front = *Front - 1;
        return 1;
    }
}

int delRear (char dQ[], int *Rear, int *Front, char *val)
{
    if ( *Rear==*Front)
        return 0; /* the queue is empty */
    else
    {
        *val = dQ[*Rear];
        /* If the size of string is odd then do not decrement
        Rear*/
        if ( (*Rear - 1) > *Front) *Rear = *Rear-1;
        return 1;
    }
}
```



```

}

int delFront (char dQ[], int *Front, int *Rear, char *val)
{
    if (*Front == *Rear)
    {
        return 0;
    } /* the queue is empty */
    else
    {
        *Front = *Front + 1;
        *val = dQ[*Front];

        return 1;
    }
}

void display (char Queue[], int Front, int Rear)
{
    int i;
    printf ("\n The contents of queue are:");
    for (i = Front+1; i <= Rear; i++)
        printf ("%c ", Queue[i]);
}

```

Note: The function `addFront()` of deque system is redundant in this application.

The function `delRear()` has been suitably modified for strings having odd number of characters.

Example 13: Two stacks can be implemented using a single array. Write a program in 'C' that allows push and pop operations from the specified stack. It must also check the conditions '*stack empty* and *full*' for a specified stack.

Solution: In a single array, two stacks can be implemented by making them grow from both ends towards the middle as shown in Figure 4.23. It may be noted that 0th element becomes the bottommost element of 1st stack, i.e., *stack1*. Similarly, the $(N - 1)$ th element becomes the bottommost element of the 2nd stack, i.e., *stack2*.

An array called `twoStack[N]` would be used to implement the required task. Two variables called `Top1` and `Top2` would keep track of the top locations of *stack1* and *stack2*, respectively. Initially, `Top1` and `Top2` would be set to locations -1 and N , respectively to suggest that both the stacks are initially empty.

Note: The stack full condition for both the stacks would be: `*top1 + 1 == *top2`

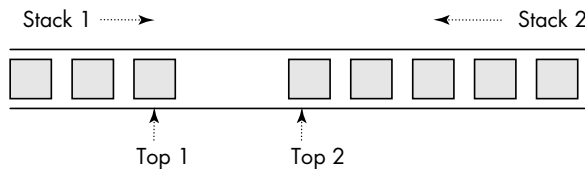


Fig. 4.23

Two stacks that grow towards middle of the array

The required program is given below:

```
/* This program simulates two stacks operations */

#include <stdio.h>
#include <conio.h>

int pushStack1 (int stak[], int *top1,int *top2, int val);
int pushStack2 (int stak[], int *top1,int *top2, int val);
int popStack1 (int stak[], int *top1);
int popStack2 (int stak[], int *top2);
void dispStack1 (int stack[], int top1);
void dispStack2 (int stack[], int top2);
void main()
{
    int twoStack[10];    /* the stack declaration */
    int top1 = -1;    /* initially the stack1 is empty */
    int top2 = 10;    /* initially the stack2 is empty */

    int val;
    int choice;
    int result;

    /* create menu */
    do
    {
        clrscr();
        printf ("\n Menu - twoStack Operations");
        printf ("\n PushStack1          1");
        printf ("\n PushStack2          2");
        printf ("\n PopStack1           3");
        printf ("\n PopStack2          4");
        printf ("\n DispStack1         5");
        printf ("\n DispStack2         6");
        printf ("\n Quit              7");

        printf ("\n Enter your choice:");
        scanf ("%d", & choice);

        switch (choice)
        {
            case 1: printf ("\n Enter the value to be pushed");
                    scanf ("%d", & val);
                    result = pushStack1(twoStack, &top1, &top2, val);
                    if (result == 0)

                        printf ("\n The stack is full");
                    break;
```

```

        case 2: printf ("\n Enter the value to be pushed");
                scanf ("%d", & val);
                result = pushStack2(twoStack, &top1, &top2, val);
                if (result == 0)
                    printf ("\n The stack is full");
                break;
        case 3: result = popStack1( twoStack, &top1);
                if (result == 9999)
                    printf ("\n The stack is empty");
                else
                    printf ("\n  The popped value = %d", result);
                break;
        case 4: result = popStack2( twoStack, &top2);
                if (result == 9999)
                    printf ("\n The stack is empty");
                else
                    printf ("\n  The popped value = %d", result);
                break;
        case 5: dispStack1 (twoStack, top1);
                break;
        case 6: dispStack2 (twoStack, top2);
                break;
    }
    printf("\n\n Press any key to continue");
    getch();
}
while (choice != 7);
}

int pushStack1 (int stack[], int *top1, int *top2, int val)
{
    if (*top1 + 1 == *top2)
        return 0;    /* the stack is full */
    else
    {
        *top1= *top1 + 1;
        stack[*top1]= val;
        return 1;
    }
}

int pushStack2 (int stack[],int *top1, int *top2, int val)
{
    if (*top1 + 1 == *top2)
        return 0;    /* the stack is full */

```

```
        else
        {
            *top2= *top2 - 1;
            stack[*top2]= val;
            return 1;
        }
    }
int popStack1 (int stack[], int *top1)
{int val;
    if (*top1 < 0)
        return 9999;    /* the stack is empty */
    else
    {
        val = stack[*top1];
        *top1 = *top1 - 1;
        return val;
    }
}
int popStack2 (int stack[], int *top2)
{ int val;
    if (*top2 >10)
        return 9999;    /* the stack is empty */
    else
    {
        val = stack[*top2];
        *top2 = *top2 + 1;
        return val;
    }
}
void dispStack1 (int stack[], int top1)
{
    int i;
    printf ("\n The contents of stack1 are:");
    for (i = top1; i >=0; i--)
        printf("%d ", stack[i]);
}
void dispStack2 (int stack[], int top2)
{
    int i;
    printf ("\n The contents of stack2 are:");
    for (i = top2; i <=10; i++)
        printf ("%d ", stack[i]);
}
```

EXERCISES

1. Give static implementation of `stack` by writing `push` and `pop` routine for it.
2. Explain overflow and underflow conditions of a `stack` with examples.
3. How can a `stack` be used in checking the well-formedness of an expression, i.e., balance of the left and right parenthesis of the expression?
4. Write down an algorithm to implement two stacks using only one array. Your `stack` routine should not report an overflow unless every slot in the array is used.
5. Write a non-recursive program to reverse a string using `stack`.
6. Explain `prefix`, `infix`, and `postfix` expressions with examples.
7. Describe a method to convert an `infix` expression in to a `postfix` expression with the help of a suitable example.
8. Translate following `infix` expressions to their equivalent `postfix` expressions:
 - $(x + y - z) / (h + k) * s$
 - $j - k / g^h + (n + m)$
 - $a * (b - c) / d + e * f$
9. Write a program that evaluates an expression represented in `postfix` form.
10. Evaluate the following `postfix` expressions for the provided data:
 - $ab^c * d / e +$ where $a = 5, b = 3, c = d = 2, e = 9$
 - $abcde + * + -$ where $a = 12, b = 4, c = 7, d = 5, e = 2$
 - $ab + cd * + e *$ where $a = 2, b = 6, c = 3, d = 5, e = 9$
11. What are the applications of a `stack`? Support your answer with suitable examples.
12. What is a linear queue? What are the limitations of linear queue?
13. What do you understand by a queue? Write algorithm for inserting and deleting of a data element from the queue.
14. Write the functions `insert()` and `delete()` for a circular queue.
15. Write a menu driven `main()` program that tests the functions developed in Exercise 14.
16. Differentiate between linear queue and circular queue. Which one is better and why?
17. Describe the applications of queues.
18. What are priority queues? Write their applications.
19. What is doubly ended queue? Write a program/algorithm to implement a doubly ended queue (deque).

Pointers

CHAPTER OUTLINE

- 5.1 Introduction
- 5.2 Pointer Variables
- 5.3 Pointers and Arrays
- 5.4 Array of Pointers
- 5.5 Pointers and Structures
- 5.6 Dynamic Allocation

5.1 INTRODUCTION

Pointers are the most powerful feature of 'C'. The beginners of 'C' find pointers hard to understand and manipulate. In an attempt to unveil the mystery behind this aspect, the basics of pointers which generally remain in background are being discussed in the following sections.

5.1.1 The '&' Operator

When a variable *x* is declared in a program, a storage location in the main memory is made available by the compiler. For example, Figure 5.1 shows the declaration of *x* as an integer variable and its storage location in the main memory.

From Figure 5.1, it may be observed that *x* is the name associated by the compiler to a location in the memory of the computer. Let us assume that, at the time of execution, the physical address of this memory location (called *x*) is 2712. Now, a point worth noting is that this memory location is viewed by the programmer as variable *x* and by the operating system as an address 2712. The address of variable *x* can be obtained by '&', an address of operator. This operator when applied to a variable gives the physical memory address of the variable. Thus, `&x` will provide the address 2712. Consider the following program segment:

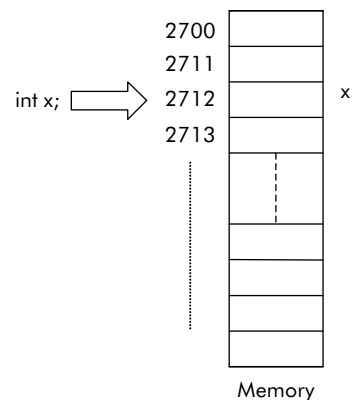


Fig. 5.1 The variable *x* and its equivalent representation in memory

```
x = 15;
printf ("\n Value of x = %d", x);
printf ("\n Address of x = %d", &x);
```

The output of the above program segment would be as shown below:

```
Value of x = 15
Address of x = 2712 (assumed value)
```

The contents of variable x (memory location 2712) are shown in Figure 5.2.

It may be noted here that the value of address of x (i.e., 2712) is assumed and the actual value is dependent on 'machine and execution' time.

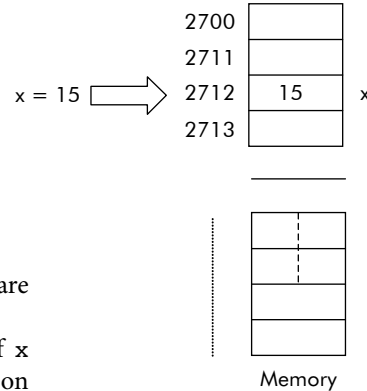


Fig. 5.2 The contents of variable x

5.1.2 The '*' Operator

The '*' is an indirection operator or 'value at address operator'. In simple words, we can say that if address of a variable is known, then the '*' operator provides the contents of the variable. Consider the following program segment:

```
#include <stdio.h>
main()
{
    int x =15;
    printf ("\n Value of x = %d", x);
    printf ("\n Address of x = %u", &x);
    printf ("\n Value at address %d = %d", *(&x));
}
```

The output of the above program segment would be as shown below:

```
Value of x = 15
Address of x = 2712
Value at address 2712 = 15
```

5.2 POINTER VARIABLES

An address of a variable can be stored in a special variable called pointer variable. A pointer variable contains the address of another variable. Let us assume that y is such a variable. Now, the address of variable x can be assigned to y by the statement: `y = &x`. The effect of this statement is shown in Figure 5.3.

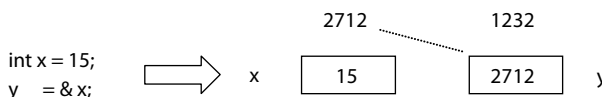


Fig. 5.3 The address of x is being stored in pointer variable y

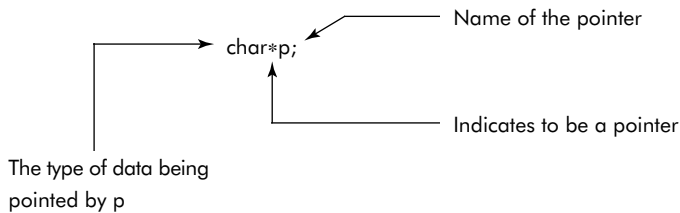
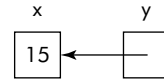
From Figure 5.3, it is clear that *y* is the variable that contains the address (i.e., 2712) of another variable *x*, whereas the address of *y* itself is 1232 (assumed value). In other words, we can say that *y* is a pointer to variable *x*. See Figure 5.4.

A pointer variable *y* can be declared in 'C' as shown below:

```
int *y;
```

The above declaration means that *y* is a pointer to a variable of type `int`. Similarly, consider the following declaration:

Fig. 5.4 *y* is a pointer to *x*



The above declaration means that *p* is a pointer to a variable of type `char`. Consider the following program segment:

```
#include <stdio.h>
main()
{
    int x = 15;
    int *y;
    y = &x;
    printf ("\n Value of x = %d", x);
    printf ("\n Address of x = %u", &x);
    printf ("\n Value of x = %d", *y);
    printf ("\n Address of x = %u", y);
    printf ("\n Address of y = %u", &y);
}
```

The output of the above program segment would be as shown below:

```
Value of x = 15
Address of x = 2712
Value of x = 15
Address of x = 2712
Address of y = 1232
```

Let us, now, consider the following statements:

```
float val = 35.67;
float *pt;
```


The first statement declares `val` to be a variable of type `float` and initializes it by value 35.67. The second statement declares `pt` to be pointer to a variable of type `float`. Please notice that `pt` can point to a variable of type `float` but it is currently not pointing to any variable as shown in Figure 5.5.

The pointer `pt` can point to the variable `val` by assigning the address of `val` to `pt` as shown below:

```
pt = &val;
```

Once the above statement is executed, the pointer `pt` gets the address of `val` and we can say that logically, `pt` points to the variable `val` as shown in Figure 5.6.

However, one must keep in mind that the arrow is for illustration sake. Actually, the pointer `pt` contains the address of `val`. The address is an integer which is not a simple value but a reference to a location in the memory.

Examples of some valid pointer declarations are:

- (i) `int *p;`
- (ii) `char *i, *k;`
- (iii) `float *pt;`
- (iv) `int **ptr;`

The example (iv) means the `ptr` is a pointer to a location which itself is a pointer to a variable of type integer.

It may be noted here that a pointer can also be incremented to point to an immediately next location of its type. For example, if the contents of a pointer `p` of type integer are 5224 then the content of `p++` will be 5226 instead of 5225 (see Figure 5.7). The reason being that an `int` is always of 2 bytes size and, therefore, stored in two memory locations as shown in Figure 5.7.

The amount of storage taken by various types of data is tabulated in Table 5.1.

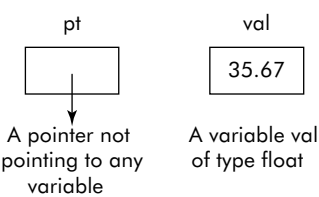


Fig. 5.5 The pointer `pt` is not pointing to any variable

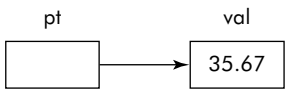


Fig. 5.6 The pointer `pt` pointing to variable `val`

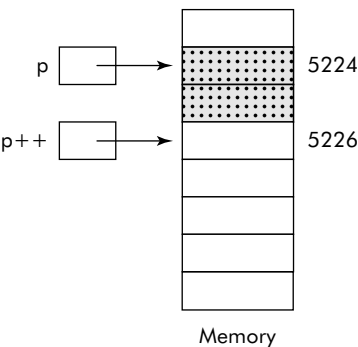


Fig. 5.7 The variable of type `int` occupies two bytes

Thus, a pointer of float type will point to an address of 4 bytes of location and, therefore, an increment to this pointer will increment its contents by 4 locations. It may be further noted that more than one pointer can point to the same location. Consider the following program segment:

```
int x = 37
int *p1, *p2;
```

The following statement:

```
p1 = &x;
```

Makes the pointer p1 point to variable x as shown in Figure 5.8.

The following statement:

```
p2 = p1
```

Makes p2 point to the same location which is being pointed by p1 as shown in Figure 5.9.

The contents of variable x are now reachable by both the pointers p1 and p2. Thus, two or more entities can cooperatively share a single memory structure with the help of pointers. In fact, this technique can be used to provide efficient communication between different parts of a program.

The pointers can be used for a variety of purposes in a program; some of them are as follows:

- To pass address of variables from one function to another.
- To return more than one value from a function to the calling function.
- For the creation of linked structures such as linked lists and trees.

Example 1: Write a program that reads from the keyboard two variables x and y of type integer. It prints the exchanged contents of these variables with the help of pointers without altering the variables.

Solution: We will use two pointers p1 and p2 for variables x and y. A third pointer p3 will be used as a temporary pointer for the exchange of pointers p1 and p2. The scheme is shown in Figure 5.10.

The program is given below:

```
/* This program illustrates the usage of pointers to
   exchange the contents of two variables */

#include <stdio.h>
main()
{
    int x, y;
    int *p1, *p2, *p3; /* pointers to integers */
```

Table 5.1 Amount of storage taken by data types

Data type	Amount of storage
character	1 byte
integer	2 byte
float	4 byte
long	4 byte
double	8 byte

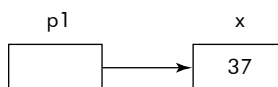


Fig. 5.8 Pointer p1 points to variable x

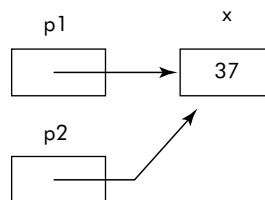


Fig. 5.9 Pointers p1 and p2 pointing to x

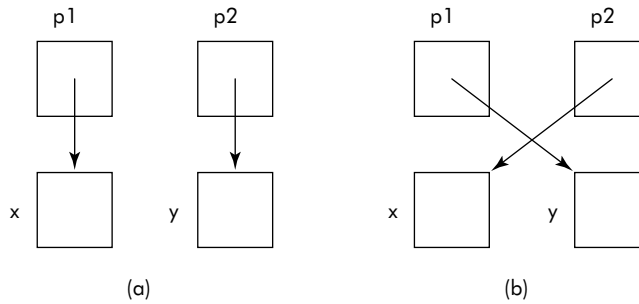


Fig. 5.10 Exchange of contents of variables by exchanging pointers

```
printf ("\n Enter two integer values");
scanf ("%d %d", &x, &y);
/* Assign the addresses x and y to p1 and p2 */
p1 = &x;
p2 = &y;
/* Exchange the pointers */
p3 = p1;
p1 = p2;
p2 = p3;

/* Print the contents through exchanged contents */
printf ("\n The exchanged contents are");
printf (" %d & %d", *p1, *p2);
}
```

Note: The output would show the exchanged values because of exchange of pointers (see Figure 5.10) whereas the contents of x and y have remained unaltered.

5.2.1 Dangling Pointers

A dangling pointer is a pointer which has been allocated but does not point to any entity. Such an un-initialized pointer is dangerous in the sense that a dereference operation on it will result in an unpredictable operation or, may be, a runtime error.

Consider the following code:

```
int * ptr;
*ptr = 50;
```

The first statement allocates the pointer called ptr as shown in Figure 5.11. It may be noted that, currently, ptr is a dangling pointer. The second statement is trying to load a value (i.e., 50) to a location which is pointed by ptr. Obviously, this is a dangerous situation because the results will be

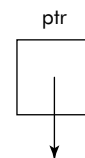


Fig. 5.11 The dangling pointer

unpredictable. The 'C' compiler does not consider it as an error, though some compilers may give a warning.

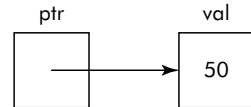
Therefore, following steps must be followed if pointers are being used:

Steps:

- (1) Allocate the pointer.
- (2) Allocate the variable or entity to which the pointer is to be pointed.
- (3) Point the pointer to the entity.

Let us assume that it is desired that the pointer `ptr` should point to a variable `val` of type `int`. The correct code in that case would be as given below:

```
int * ptr;
int val;
ptr = &val;
*ptr = 50;
```



The result of the above program segment is shown in **Fig. 5.12** The correct assignment Figure 5.12.

5.3 POINTERS AND ARRAYS

Pointers and arrays are very closely related to each other. In 'C', the name of an array is a pointer that contains the base address of the array. In fact, it is a pointer to the first element of the array. Consider the array declaration given below:

```
int list[] = {20, 30, 35, 36, 39};
```

This array will be stored in the contiguous locations of the main memory with starting address equal to 1001 (assumed value), as shown in Figure 5.13. Since the array called 'list' is of integer type, each element of this array occupies two bytes. The name of the array contains the starting address of the array, i.e., 1001.

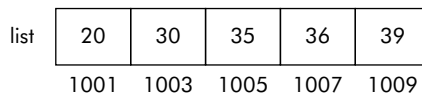


Fig. 5.13 The contiguous storage allocation with starting address = 1001

Consider the following program segment:

```
printf ("\n Address of zeroth element of array list= %d", list);
printf ("\n Value of zeroth element of array list= %d", *list);
```

Once the above program segment is executed, the output would be as shown below:

```
Address of zeroth element of array list = 1001
Value of zeroth element of array list = 20
```

We could have achieved the same output by the following program segment also:

```
printf ("\n Address of zeroth element of array list = %d", &list [0]);
printf ("\n Value of zeroth element of array list = %d", list []);
```

Both the above given approaches are equivalent because of the following equivalence relations:

```
list ≡ &list [0] - both denote address of zeroth element of the array list
*list ≡ list[0] - both denote value of zeroth element of the array list
```

The left side approach is known as pointer method and right side as array indexing method. Let us now write a program that prints out a list by array indexing method.

```
/* Array indexing method */
#include <stdio.h>
main()
{
    static int list [] = {20, 30, 35, 36, 39};
    int i;
    printf ("\n The list is ...");
    for (i = 0; i < 5; i++)
        printf ("\n %d %d ---element", list[i], i);
}
```

The output of this program would be:

This list is...

```
20    0.....element
30    1.....element
35    2.....element
36    3.....element
39    4.....element
```

The above program can also be written by pointer method as shown below:

```
/* Pointer method of processing an array */
#include <stdio.h>
main()
{
    static int list [] = {20, 30, 35, 36, 39};
    int i;
    printf ("\n The list is ...");
    for (i = 0; i < 5; i++)
        printf ("\n %d %d ---element", *(list +i), i);
}
```

It may be noted in the above program that we have used the term `*(list + i)` to print an *i*th element of the array. Let us analyse this term. Since `list` designates the address of zeroth element of the array, we can access its value through value at address operator, i.e., `^`. The following terms are equivalent:

```
-> *list ≡ *(list + 0) ≡ list [0]
-> *(list + 1) ≡ list [1] and so on
```

Thus, we can refer to *i*th element of array list by either of the following ways:

```
*(list + i) or *(i + list) or list [i]
```

So far, we have used the name of an array to get its base address and manipulate it. However, there is a small difference between an ordinary pointer and the name of an array. The difference is that the array name is a pointer constant and its contents cannot be changed. Thus, the following operations on list are illegal because they try to change the contents of a list, a pointer constant.

```
List = NULL;    /* Not allowed */
List = & Val;   /* Not allowed */
list++          /* Not allowed */
list--          /* Not allowed */
```

Consider the following program:

```
#include <stdio.h>
main()
{
    char text[6] = "Helpme";
    char *p = "Helpme";
    printf ("\n %c", text[3]);
    printf ("\n %c", p[3]);
}
```

The output of above program would be:

```
p
p
```

From the above program, it looks as if the following declarations are equivalent as `text[3]` and `p[3]` behave in identical manner.

```
char text [6];
char *p;
```

It may be noted that the above declarations are not at all equivalent though the behaviour may be same. In fact, the array declaration `text [6]` asks from the compiler for six locations of char type whereas the pointer declaration `char *p` asks for a pointer that can point to any variable of following type:

- (i) char – a character
- (ii) string – a string of a characters
- (iii) Null – nowhere

Consider the following declarations:

```
int list = {20, 30, 35, 36, 39};
int *p;    /* pointer variable p */
p = list;  /* assign the starting address of array list to pointer p */
```

Since `p` is a pointer variable and has been assigned the starting address of array list, the following operations become perfectly valid on this pointer:

```
p++, p-- etc.
```

Let us now write a third version of the program that prints out the array. We will use pointer variable `p` in this program.

```
/* Pointer variable method of processing an array */
#include <stdio.h>
main()
{
    static int list [] = {20, 30, 35, 36, 39};
    int *p;
    int i = 0;
    p = list;    /* Assign the starting address of the list */
    printf ("\n The list is ...");
    while (i < 5)
    {
        printf ("\n %d %d ---element", *p, i);
        i++;
        p++;    /* increment pointer */
    }
}
```

The output of the above program would be as shown below:

```
20    0.....element
30    1.....element
35    2.....element
36    3.....element
39    4.....element
```

From our discussion on arrays, we know that the strings are stored and manipulated as array of characters with last character being a null character (i.e., `'\0'`).

For example, the string ENGINEERING can be stored in an array (say `text`) as shown in Figure 5.14.

	0	1	2	3	4	5	6	7	8	9	10	11
Text	E	N	G	I	N	E	E	R	I	N	G	\0

Fig. 5.14 The array called 'text'

We can declare this string as a normal array by the following array declaration:

```
char text [11];
```

Consider the following declaration:

```
char *p;
p = text;
```

In the above set of statements, we have declared a pointer `p` that can point to a character or string of characters. The next statement assigns the starting address of character string `'text'` to the variable pointer `p` (see Figure 5.15).

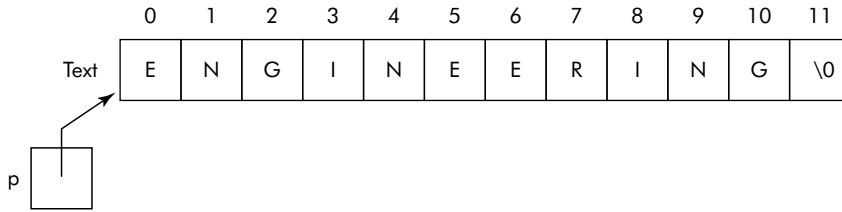


Fig. 5.15 Pointer p points to the array called 'text'

Since `text` is a pointer constant, its contents cannot be changed. On the contrary, `p` is a pointer variable and can be manipulated like any other pointer as shown in the program given below:

```
/* This program illustrates the usage of pointer to a string */
#include <stdio.h>
main()
{
    char text[] = "ENGINEERING"; /* The string */
    char *p; /* The pointer */

    p = text; /* Assign the starting address of string to p */

    printf ("\n The string.."); /* Print the string */
    while (*p != '\0')
    {printf ("%c", *p);
      p++;
    }
}
```

Example 2: What would be the output of the following code?

```
#include <stdio.h>
{
    char *ptr;
    ptr = "Nice";
    printf ("\n% c", *&ptr [1]);
}
```

Solution: The output would be: i.

Example 3: What would be the output of the following code?

```
#include <stdio.h>
main ()
{
    int list [5] = {2, 5, 6, 0, 9};
    3[list] = 4[list] + 6;
    printf ("%d", *(list + 3));
}
```

Solution: The output would be 15.

5.4 ARRAY OF POINTERS

Like any other array, we can also have an array of pointers. Each element of such an array can point to a data item such as variable, array etc. For example, consider the declaration given below:

```
float *x [20];
```

This declaration means that *x* is an array of 20 elements and each element is a pointer to a variable of type float. Let us now construct an array of pointers to strings.

```
char *item [ ]{ Chair,
    Table,
    Stool,
    Desk,
};
```

The above declaration gives an array of pointers called *item*. Each element of *item* points to a string as shown in Figure 5.16.

The advantage of having an array of pointer to strings is that manipulation of strings becomes convenient. For example, the string containing table can be copied into a pointer *ptr* without using *strcpy()* function by the following set of statements:

```
char *ptr; // declare a pointer
           // to a string
ptr = item [1]; // assign the
               // appropriate
               // pointer to ptr
```

It may be noted that once the above set of statements is executed, both the pointers *item*[1] and *ptr* will point to the same string i.e., 'table' as shown in Figure 5.17.

However, the changes made by the pointer *ptr* to the string table will definitely disturb the contents of the string pointed by the pointer *item* [1] and vice versa. Therefore, utmost care should be taken by the programmer while manipulating pointers pointing to the same memory locations.

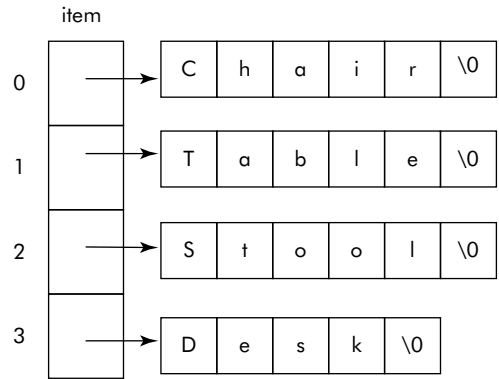


Fig. 5.16 Array of pointers to strings

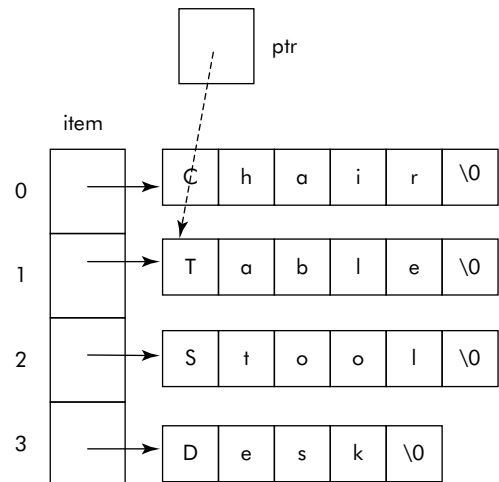


Fig. 5.17 Effect of statement *ptr = item*[1]

5.5 POINTERS AND STRUCTURES

We can have a pointer to a structure, i.e., we can also point a pointer to a structure. A pointer to a structure variable can be declared by prefixing the variable name by a *^*. Consider the following declaration:

```

    struct xyz { /* structure declaration */
        int a;
        float y;
    };
    struct xyz abc; /* declare a variable of type xyz */
    struct xyz *ptr; /* declare a pointer ptr to a variable of type xyz */

```

Once the above declarations are made, we can assign the address of the structure variable `abc` to the structure pointer `ptr` by the following statement:

```
ptr = &abc;
```

Since `ptr` is a pointer to the structure variable `abc`, the members of the structure can also be accessed through a special operator called arrow operator, i.e., ‘`→`’ (minus sign followed by greater than sign). For example, the members `a` and `y` of the structure variable `abc` pointed by `ptr` can be assigned values 30 and 50.9, respectively by the following statement:

```
ptr -> a = 30;
ptr -> y = 50.9;
```

The arrow operator ‘`→`’ is also known as a pointer-to-member operator.

Example 4: Write a program that defines a structure called `item` for the record structure given below. It reads the data of the record through dot operator and prints the record by arrow operator. Use suitable data types for the members of the structure called ‘`item`’.

item	code	quantity	cost
------	------	----------	------

Solution: We will use a pointer `ptr` to point to the structure called ‘`item`’. The required program is given below:

```

/* This program demonstrates the usage of an arrow operator */
#include <stdio.h>
main()
{
    struct item {
        char code[5];
        int Qty;
        float cost;
    };

    struct item item_rec; /* Define a variable of struct type */
    struct item *ptr;      /* Define a pointer of type struct */
                          /* Read data through dot operator */
    printf ("\n Enter the data for an item");
    printf ("\nCode:"); scanf ("%s", &item_rec.code);
    printf ("\nQty:"); scanf ("%d", &item_rec.Qty);
    printf ("\nCost:"); scanf ("%f", &item_rec.cost);
                          /* Assign the address of item_rec */
    ptr = &item_rec;

```

```

/* Print data through arrow operator */
printf ("\n The data for the item...");
printf ("\nCode : %s", ptr -> code);
printf ("\nQty : %d", ptr -> Qty);
printf ("\nCost : %5.2f", ptr -> cost);
}

```

From the above program, we can see that the members of a static structure can be accessed by both dot and arrow operators. However, dot operator is used for simple variable whereas the arrow operator is used for pointer variables.

Example 5: What is the output of the following program?

```

#include <stdio.h>
main()
{
    struct point
    {
        int x, y;
    } polygon[] = {{1,2},{1,4},{2,4},{2,2}};

    struct point *ptr;
    ptr = polygon;

    ptr++;
    ptr -> x++;
    printf ("\n %d", ptr -> x);
}

```

Solution: From the above program, it can be observed that `ptr` is a pointer to an array of structures called `polygon`. The statement `ptr++` moves the pointer from zeroth location (i.e., {1, 2}) to first location (i.e., {1, 4}). Now, the statement `x++` has incremented the field `x` of the structure by one (i.e., 1 has incremented to 2).

The output would be 2.

5.6 DYNAMIC ALLOCATION

We know that in a program when a variable `x` is declared, a storage location is made available to the program as shown in Figure 5.18.

This memory location remains available, even if it is not needed, to the program as long as the program is being executed. Therefore, the variable `x` is known as a **static variable**. Dynamic variables, on the other hand, can be allocated from or returned to the system according to the needs of a running program. However, a dynamic variable does not have a name and can be referenced only through a pointer.

`int x; ≡ x`



Consider the declaration given below:

```
int*p;
```

Fig. 5.18

Memory allocation to variable `x`

The above declaration gives us a dangling pointer *p* that points to nowhere (see Figure 5.19). Now, we can connect this dangling pointer either to a static variable or a dynamic variable. In our previous discussion on pointers, we have always used pointers with static variables. Let us now see how the pointer can be connected to a dynamic variable.

In 'C', dynamic memory can be allocated by a function called `malloc()`. The format of this function is given below:

```
(<type> *) (malloc (<size>))
```

where **<type>**: is the type of variable, which is required at run time.

malloc: is a reserved word indicating that memory allocation is to be done.

<size>: is the size of memory to be allocated. In fact, the size of any variable can be obtained by `sizeof()` function.

Consider the following declaration:

```
int *p;
int Sz;
Sz = sizeof (int);
P = (int*) malloc (Sz);
```

The above set of statements is executed in the following manner;

- (1) The pointer *p* is created. It is dangling, i.e., it is not connected anywhere (see Figure 5.20 (a))
- (2) A variable called *Sz* is declared to store the size of dynamic variable.
- (3) The size of dynamic variable (int in this case) is stored in the variable called *Sz*.
- (4) A dynamic variable of type integer is taken through `malloc()` function and the dangling pointer *p* is connected to the variable (see Figure 5.20 (b)).

It may be noted from Figure 5.20 (b) that a dynamic variable is anonymous in the sense that it has no name and, therefore, it can be referenced only through the pointer *p*. For example, a value (say 50) can be assigned to the dynamic variable by the following statement:

```
*p = 50;
```

The above statement means that it is desired to store a value 50 in a memory location pointed to by the pointer *p*. The effect of this statement is shown in Figure 5.21.

It may be observed that *p* is simply a pointer or an address and `*p` gives the access to the memory location or the variable where it is pointing to. It is very important to learn the difference between the pointer and its associated dynamic variable. Let us consider the following program segment:

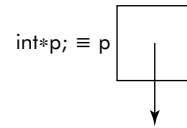


Fig. 5.19 The dangling pointer *p*

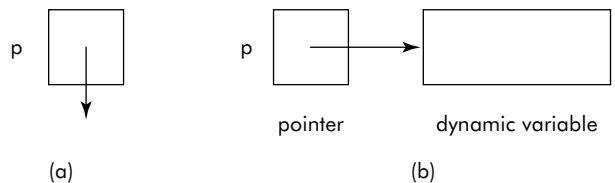


Fig. 5.20 The effect of `malloc()` function



Fig. 5.21 The effect of assignment of `*p = 50`

```

int *Aptr, *Bptr;                /* declare pointers */
Aptr = (int *) malloc (sizeof int); /* allocate dynamic memory */
Bptr = (int *) malloc (sizeof (int));
*Aptr = 15;
*Bptr = 70;

```

The outcome of the above program segment would be as shown in Figure 5.22.

Let us now see the effect of the following assignment statement on the memory allocated to the pointers `Aptr` and `Bptr`.

```
Aptr = Bptr;
```

The above assignment makes `Aptr` point to the same location or variable which is already being pointed by `Bptr` as shown in Figure 5.23. Thus, both `Aptr` and `Bptr` are now pointing to the same memory location.

A close observation would reveal that the dynamic variable containing value 15 is now lost in the sense that it is neither available to `Aptr` any more and nor it is available to the system. The reason is that the system gave it to the pointer `Aptr` and now `Aptr` is pointing to some other location. This dynamic variable, being anonymous, (i.e., without name) is no more accessible. Even if we want to reverse the process, we cannot simply do it. This phenomenon of losing dynamic variables is known as **memory bleeding**. Thus, utmost care should be taken while manipulating the dynamic variables. It is better to return a dynamic variable whenever it is not required in the program.

A dynamic variable can be returned to the system by function called `free()`. The general form of usage of this operator is given below:

```
free (p_var);
```

where `free`: is a reserved word

`p_var`: is the pointer to the dynamic variable to be returned to the system.

Let us consider pointers shown in Figure 5.23. If it is desired to return the dynamic variable pointed by pointer `Bptr`, we can do so by the following statement:

```
free (Bptr);
```

Once the above statement is executed, the dynamic variable is returned back to the system and we are left with two dangling pointers `Bptr` and `Aptr` as shown in Figure 5.24. The reason for this is obvious because both were pointing to the same location. However, the dynamic variable containing value 15 is anonymous and not available anymore and, hence, a total waste of memory.

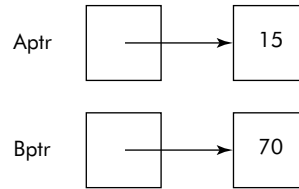


Fig. 5.22 Value assignment through pointers

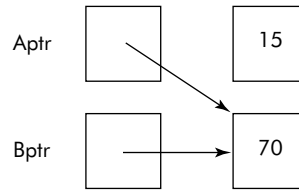


Fig. 5.23 Both pointers pointing to the same location

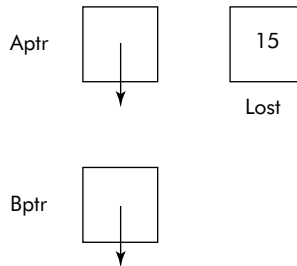


Fig. 5.24 The two dangling pointers and a lost dynamic memory variable

Example 6: Write a program that dynamically allocates an integer. It initializes the integer with a value, increments it, and print the incremented value.

Solution: The required program is given below:

```
#include <stdio.h>
#include <alloc.h>
main()
{
    int *p;          /* A pointer to an int */
    int Sz;

    /* Compute the size of the int */
    Sz = sizeof (int);
    /* Allocate a dynamic int pointed by p */
    p = (int *) malloc (Sz);
    printf ("\n Enter a value :");
    scanf ("%d", p);
    *p = *p + 1;

    /* Print the incremented value */
    printf ("\n The Value = %d", *p);
    free(p);
}
```



Example 7: Write a program that dynamically allocates a structure whose structure diagram is given below. It reads the various members of the structure and prints them.

Student	Name	Age	Roll
---------	------	-----	------

Solution: We will use the following steps to write the required program:

- (1) Declare a structure called student.
- (2) Declare a pointer ptr to the structure student.
- (3) Compute the size of the structure.
- (4) Ask for dynamic memory of type student pointed by the pointer ptr.
- (5) Read the data of the various elements using arrow operator.
- (6) Print the data.
- (7) Return the dynamic memory pointed by ptr.

The required program is given below:

```
#include <stdio.h>
#include <alloc.h>
main()
{
    struct student {    /* Declare the structure */
        char name[15];
        int age;
        int roll;
    };
    struct student *ptr;
    int Sz;

        /* Compute the size of struct */
    Sz = sizeof(struct student);
        /* Ask for dynamic memory of type student*/
    ptr = (struct student *) malloc(Sz);

    printf ("\n Enter the data of the student");

    printf ("\nName:"); /*fflush(stdin);*/ gets(ptr -> name);
    printf ("\nAge:"); scanf ("%d", &ptr -> age);
    printf ("\nRoll:"); scanf ("%d", &ptr -> roll);

    printf ("\n The data is...");

    printf ("\nName :"); puts(ptr -> name);
    printf ("\nAge :%d", ptr -> age);
    printf ("\nRoll :%d", ptr -> roll);
    free (ptr);
}
```

Note: An array can also be dynamically allocated as demonstrated in the following example.

Example 8: Write a program that dynamically allocates an array of integers. A list of integers is read from the keyboard and stored in the array. The program determines the smallest in the list and prints its location in the list.

Solution: The solution to this problem is trivial and the required program is given below:

```
/* This program illustrates the usage of dynamically allocated array */
#include <stdio.h>
main()
{
    int i, min, pos;
    int *list;
    int Sz, N;
    printf ("\n Enter the size of the list:");
    scanf ("%d", &N);
```



```

    Sz = sizeof(int) * N ;    /* Compute the size of the list */

    /* Allocate dynamic array size N */
    list = (int*) malloc (Sz);
    printf ("\n Enter the list");
                                /* Read the list */
    for (i = 0; i < N; i++)
    {
        printf ("\n Enter Number:");
        scanf ("%d", (list + i));
    }

    pos = 0;    /* Assume that the zeroth element is min */
    min = *(list + 0);
                /* Find the minimum */
    for (i = 1; i < N; i++)
    {
        if (min > *(list + i))
        {min = *(list + i);
         pos = i;
        }
    } /* Print the minimum and its location */
    printf ("\n The minimum is = %d at position = %d", min, pos);
    free (list);
}

```

From the above example, it can be observed that the size of a dynamically allocated array can be specified even at the run time and the required amount of memory is allocated. This is in sharp contrast to static arrays for whom the size have to be declared at the compile time.

5.6.1 Self Referential Structures

When a member of a structure is declared as a pointer to the structure itself then the structure is called a self referential structure. Consider the following declaration:

```

struct chain {
    int val;
    struct chain *p;
};

```

The structure '*chain*' consists of two members: *val* and *p*. The member *val* is a variable of type *int* whereas the member *p* is a pointer to a structure of type *chain*. Thus, the structure *chain* has a member that can point to a structure of type *chain* or may be itself. This type of self referencing structure can be viewed as shown in Figure 5.25.

Since pointer *p* can point to a structure variable of type *chain*, we can connect two such structure

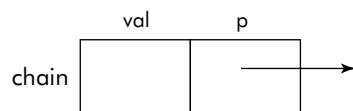


Fig. 5.25 Self referential structure chain

variables A and B to obtain a linked structure as shown in Figure 5.26.

The linked structure given in Figure 5.26 can be obtained by the following steps:

- (1) Declare structure chain.
- (2) Declare variable A and B of type chain.
- (3) Assign the address of structure B to member p of structure A.

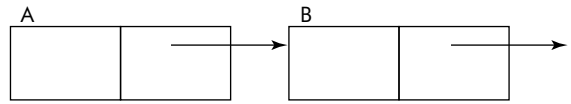


Fig. 5.26 Linked structure

These above steps have been coded in the program segment given below:

```
struct chain {    /* Declare structure chain */
    int val;
    chain *p;
};
struct chain A, B; /* Declare structure variables A and B */
A.p = &B;          /* Connect A to B */
```

From Figure 5.26 and the above program segment, we observe that the pointer p of structure variable B is dangling, i.e., it is pointing to nowhere. Such pointer can be assigned to NULL, a constant indicating that there is no valid address in this pointer. The following statement will do the desired operation:

```
B.p = NULL;
```

The data elements in this linked structure can be assigned by the following statements:

```
A.val = 50;
B.val = 60;
```

The linked structure now looks like as shown in Figure 5.27.

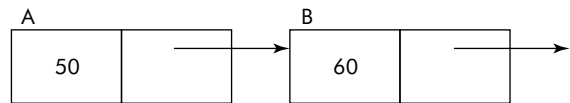


Fig. 5.27 Value assignment to data elements

We can see that the members of structure B can be reached by two methods:

- (1) From its variable name B through dot operator.
- (2) From the pointer p of variable A because it is also pointing to the structure B. However, in this case the arrow operator is needed.

Consider the statements given below:

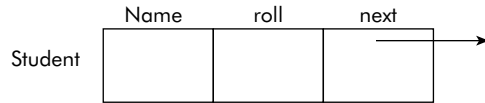
```
printf ("\n the contents of member val of B = %d", B.val);
printf ("\n the contents of member val of B = %d", A.p -> val);
```

Once the above statements are executed, the output would be:

```
The contents of member val of B = 60
The contents of member val of B = 60
```

The linked structures have great potential and can be used in numerous programming situations such as lists, trees, etc.

Example 9: Write a program that uses self referential structures to create a linked list of nodes of following structure. While creating the linked list, the student data is read. The list is also travelled to print the data.



Solution: We will use the following self referential structure for the purpose of creating a node of the linked list.

```
struct stud{
    char name [15];
    int roll;
    struct stud next;
};
```

The required linked list would be created by using three pointers: *first*, *far* and *back*. The following algorithm would be employed to create the list:

Step

- (1) Take a new node in pointer called *first*.
- (2) Read *first*→name and *first*→roll.
- (3) Point *back* pointer to the same node being pointed by *first*, i.e., *back* = *first*.
- (4) Bring a new node in the pointer called *far*.
- (5) Read *far*→name and *far*→roll.
- (6) Connect next of *back* to *far*, i.e., *back* → *next* = *far*.
- (7) Take *back* to *far*, i.e., *back* = *far*.
- (8) Repeat steps 4 to 7 till whole of the list is constructed.
- (9) Point next of *far* to NULL, i.e., *far* → *next* = NULL.

(10) Stop.

The required program is given below:

```
#include <stdio.h>
#include <alloc.h>
main()
{
    struct student {
        char name [15];
        int roll;
        struct student *next;
    };
    struct student *First, *Far, *back;
    int N, i;
    int Sz;
    Sz = sizeof (struct student);
```

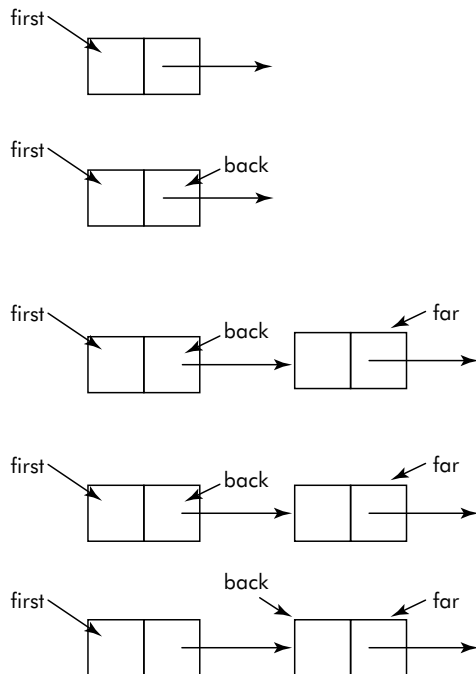


Fig. 5.28 Creation of a linked list

```

printf ("\n Enter the number of students in the class");
scanf ("%d", &N);
        /* Take first node */
First = (struct student *) malloc(Sz);
        /* Read the data of the first student */

printf ("\n Enter the data");
printf ("\n Name : "); fflush(stdin); gets(First -> name);
printf ("\n Roll : "); scanf("%d", &First -> roll);
        /* point back where First points */
back = First;

for (i =2; i <=N; i++)
{
        /* Bring a new node in Far */
        Far = (struct student *) malloc (Sz);
        /* Read Data of next student */
        printf ("\n Name: "); fflush(stdin); gets(Far -> name);
        printf ("\n Roll: "); scanf("%d", &Far -> roll);
        /* Connect Back to Far */
        back -> next = Far;
        /* point back where Far points */
        back = Far;
}
        /* Repeat the process */
Far -> next = NULL;

        /* Print the created linked list */
Far = First;    /* Point to the first node */

printf ("\n The Data is ....");
while (Far != NULL)
{
        printf ("\n Name: %s , Roll : %d", Far -> name, Far -> roll);
        /* Point to the next node */
        Far = Far -> next;
}
}

```

Example 10: Modify the program developed in Example 1 such that

- (1) assume that the number of students in the list is not known.
- (2) the list of students is split into the two sublists pointed by two pointers: `front_half` and `back_half`. The `front_half` points to the front half of the list and the `back_half` to the back half. If the number of students is odd, the extra student should go in the front list.

Solution: The list of students would be created in the same fashion as done in Example 1. The following steps would be followed to split the list in the desired manner:

- (1) Travel the list to count the number of students.
- (2) Compute the middle of the list.

- (3) Point the pointer called `front_half` to `first` and travel the list again starting from the `first` and reach to the `middle`.
- (4) Point a pointer called `back_half` to the node that succeeds the `middle` of the list.
- (5) Attach `NULL` to the next pointer of the `middle` node.
- (6) Print the two lists, i.e., pointed by `front_half` and `back_half`.

The required program is given below:

```
#include <stdio.h>
#include <alloc.h>
main()
{
    struct student {
        char name [15];
        int roll;
        struct student *next;
    };
    struct student *First, *Far, *back;
    struct student *front_half, *back_half;
    int N, i, count, middle;
    int Sz;
    Sz = sizeof (struct student);
    printf ("\n Enter the number of students in the class");
    scanf ("%d", &N);
        /* Take first node */
    First = (struct student *) malloc (Sz);
        /* Read the data of the first student */

    printf ("\n Enter the data");
    printf ("\n Name: "); fflush(stdin); gets(First -> name);
    printf ("\n Roll: "); scanf("%d", &First -> roll);
    First -> next =NULL;
        /* point back where First points */
    back = First;

    for (i = 2; i <=N; i++)
    {
        /* Bring a new node in Far */
        Far = (struct student *) malloc (Sz);
        /* Read data of next student */
        printf ("\n Name: "); fflush(stdin); gets(Far -> name);
        printf ("\n Roll: "); scanf("%d", &Far -> roll);
        /* Connect Back to Far */
        back -> next = Far;
        /* Point back where Far points */
        back = Far;
    }
        /* Repeat the process */
```

```

Far -> next = NULL;
    /* Count the number of nodes */

Far = First; /* Point to the first node */
count = 0;
while (Far != NULL)
{
    count ++;
    Far = Far -> next;
}
    /* Split the list */
if ( count == 1 )
{
    printf ("\n The list cannot be split");
    front_half = First;
}
else
{
    /* Compute the middle */
    if ( (count % 2) == 0) middle = count/2;
    else middle = count/2 + 1;
    /* Travel the list and split it */
    front_half = First;
    Far = First;

    for (i =1; i <= middle; i++)
    {
        back = Far;
        Far = Far -> next;
    }
    back_half = Far;
    back -> next = NULL;
}

    /* Print the two lists */
Far = front_half;
printf ("\n The Front Half...:");
for (i = 1; i <= 2; i++)
{
    while (Far != NULL)
    {
        printf("Name: %s, Roll: %d ", Far -> name, Far -> roll);
        Far = Far -> next;
    }
    if (count == 1 || i == 2) break;
    printf ("\n The Back Half...: ");
    Far = back_half;
}
}

```

FREQUENTLY ASKED QUESTIONS

1. What is meant by a pointer?

Ans. It is a variable which can only store the address of another variable.

2. Are the expressions `(*p)++` and `++*p` equivalent?

Ans. Yes

3. What happens if a programmer assigns a new address to a dynamic variable before he either assigns its address to another pointer or deletes the variable?

Ans. The dynamic variable will be lost, i.e., it becomes inaccessible to program as well as the operating system.

4. What is meant by memory bleeding?

Ans. The lost dynamic variables lead to memory bleeding.

5. Pointers always contain integers. Comment.

Ans. We know that a pointer contains an address of other variables. Since an address can only be a whole number or integer, the pointers contain integers only.

6. What does the term `**j` mean?

Ans. The term `**j` means that `j` is a pointer to a pointer.

7. Are the expressions `*p++` and `++*p` equivalent?

Ans. No. The expression `*p++` returns the contents of the variable pointed by `p` and then increments the pointer. On the other hand, the expression `++*p` returns the incremented contents of the variable pointed by `p`.

8. Differentiate between an uninitialized pointer and a `NULL` pointer.

Ans. An un initialized pointer is a dangling pointer which may contain any erroneous address. A `NULL` pointer does not point to any address location.

TEST YOUR SKILLS

1. Give the output of the following program:

```
#include <stdio.h>
main()
{
    int i, j;
    char *str = "CALIFORNIA";
    for (i = 0; str[i]; i++)
    {
        for (j = 0; j <= i; j++)
            printf ("%c", str[j]);
        printf ("\n");
    }
}
```

Ans. The output would be

```
C
CA
CAL
CALI
CALIF
CALIFO
CALIFOR
CALIFORN
CALIFORNI
CALIFORNIA
```

2. What would be the output of the following?

```
#include <stdio.h>
main()
{
    int a;
    *a = 50;
    printf ("%d", a);
}
```

Ans. The output would be 50.

3. What would be the output of the following code?

```
#include <stdio.h>
main()
{int i = 50;
int *j = &i;
printf ("\n %d", ++ *(j));
}
```

Ans. The output would be 51.

4. What would be the output of the following?

```
#include <stdio.h>
main()
{
    char *ptr = "abcd";
    char ch = ++ *ptr ++;
    printf ("\n %c", ch);
}
```

Ans. The output would be b.

5. What would be the output of the following code? Assume that the array starts at location 5714 in the memory.

```
#include <stdio.h>
main()
{
    int tab [3][4] = {5, 6, 7, 8,
```

```

        1, 2, 3, 4,
        9, 10, 0, 11};
printf ("\n %d  %d", *tab[0] + 1, *(tab[0] + 1));
printf ("\n %d", *(* (tab + 0) + 1));
}

```

Ans. The output would be

6 6

6

6. What would be the output of the following code?

```

#include <stdio.h>
main()
{
    struct val {
        int Net;
    };
    struct val x;
    struct val *p;

    p = &x;
    p -> Net = 50;
    printf ("\n %d", ++ (x.Net));
}

```

Ans. The output would be 50.

7. What would be the output of the following code?

```

#include <stdio.h>
main()
{
    struct val {
        int Net;
    };
    struct val x;
    struct val *p;

    p = &x;
    p -> Net = 50;
    printf ("\n %d", (x.Net)++);
}

```

Ans. The output would be 21.

EXERCISES

1. Explain the `&` and `*` operators in detail with suitable examples.
2. Find the errors in the following program segments:

(a)

```
int val = 10
int * p;
p = val;
```

(b)

```
char list [10];
char p;
list = p;
```

3. (i) What will be the output of the following program:

```
#include <stdio.h>
main()
{
    int val = 10;
    int *p, **k;
    p = &val;
    k = &p;
    printf ("\n %d %d %d %d", p, *p, *k, **k);
}
```

(ii)

```
#include <stdio.h>
main()
{
    char ch;
    char *p;
    ch = 'A';
    p = &ch;
    printf ("\n %c %c", ch, (*p)++);
}
```

4. What will be the output of the following:

```
#include <stdio.h>
main()
{
    int list[5], i;
    *list = 5;
    for (i = 1; i < 5; i++)
        *(list + i) = *(list + i - 1)*i;
    printf ("\n");
    for (i = 0; i < 5; i++)
        printf ("%d ", *(list + i)) ;
}
```

5. Find the errors in the following program segment:

```
struct xyz {
    char *A;
    char *B;
    int val;
};
xyz s;
A = "First text";
B = "Second text";
```

6. What will be the output of the following program segment?

```
#include <stdio.h>
main()
{
    struct Myrec {
        char ch;
        struct Myrec *link;
    };
    struct Myrec x, y, z, *p;
    x.ch = 'x';
    y.ch = 'y';
    z.ch = 'z';
    x.link = &z;
    y.link = &x;
    z.link = NULL;
    p = &y;
    while (p!= NULL)
    {
        printf ("%c ", p->ch);
        p = p -> link;
    }
}
```

7. What will be the output of the following program?

```
#include <stdio.h>
main()
{
    float cost [] = {35.2, 37.2, 35.42, 36.3};
    float *ptr[4];
    int i;
    for (i = 0; i < 4; i++)
        *(ptr + i) = cost + i;
    for (i = 0; i < 4; i++)
        printf ("%f ", (*(ptr + i)));
}
```

Assume the base address of the array cost to be 2025.

8. What will be the output of the following program:

```
#include <stdio.h>
main()
{
    char item[]="COMPUTER";
    int i;
    for (i = 7; i >= 0 ; i--)
        printf ("%c", *(item + i));
}
```

Linked Lists

CHAPTER OUTLINE

- 6.1 Introduction
- 6.2 Linked Lists
- 6.3 Operations on Linked Lists
- 6.4 Variations of Linked Lists
- 6.5 The Concept of Dummy Nodes
- 6.6 Linked Stacks
- 6.7 Linked Queues
- 6.8 Comparison of Sequential and Linked Storage
- 6.9 Solved Problems

6.1 INTRODUCTION

An array is a very simple and extremely useful data structure. A large number of applications based on stacks and queues can be easily implemented with the help of arrays. Lists and tables have natural similarity with one- and two-dimensional arrays. Though the allocation of space is sequential in an array, the access to its elements is random. For a given index the element can be accessed in a time, independent of its location in the array.

However, there are many problems associated with this data structure. Some of the important problems are as follows:

- (1) An array is a static data structure and, therefore, its size should be known in advance before its usage.
- (2) If a list of data items is to be stored, then an array of approximate size is used, i.e., the array has to be large enough to hold the maximum amount of data one could logically expect. This is totally a guess work and can result in overflow or wastage of main memory.
- (3) What if the number of elements in the list is not known in advance?
- (4) Insertion or deletion operations in an array, except from the end, are a costly exercise. These operations require large number of shifting of elements in one direction or the other. This a very time consuming exercise.

The above problems can be dealt with the help of *linked structures*, discussed in the subsequent sections.

6.2 LINKED LISTS

Linked structures are a completely different way to represent a list of items. Each element of the list is made to point to the next element in the list as shown in Figure 6.1(a).

An element in the list is called a node. The node is a *self-referential* structure, having two parts: *Data* and *Next* as shown in Figure 6.1(b). The *Data* part contains the information about the element and the *Next* part contains a pointer to the next element or node in the list. For example, in Figure 6.1(a), a list consisting four names (“Preksha”, “Sagun”, “Bhawna”, and “Ridhi”) is represented using linked structures. Such a list is called *linked list* or a linear linked list or a singly linked list.

Linked list is a series of nodes. Each node has two parts: data and next. The data part contains the information about the node and the next part is a pointer that points to next node. The next of last node points to NULL.

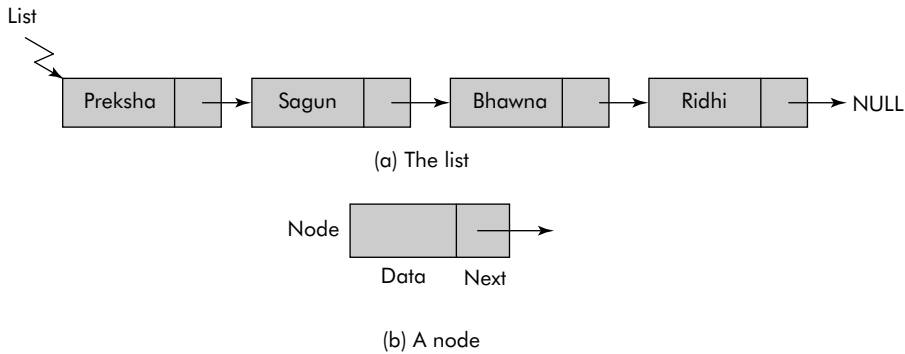


Fig. 6.1 Linked representation of a list

It may be noted that the list is pointed by a pointer called ‘List’. Currently List is pointing to a node containing the name ‘Preksha’, the next part of which is pointing to ‘Sagun’, the next of which is pointing to ‘Bhawna’, and so on.

By convention, a node X means that the node is being pointed by a pointer X as shown in Figure 6.2. In an algorithm, the data part of X is written as DATA(X) and the next part is written as NEXT(X).

Let us now insert a new node containing data called ‘Samridhi’ between ‘Preksha’ and ‘Sagun’. This insertion operation can be carried out by the following steps:

Step

1. Get a node called ptr.
2. DATA(ptr) = ‘Samridhi’
3. NEXT(ptr) = NEXT(List)
4. NEXT(List) = ptr

The third step means that the Next pointer of ptr should point to the same location that is currently being pointed by Next of List.

The effect of above steps (1–4) is shown in Figure 6.3.

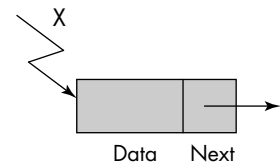


Fig. 6.2 A node called X

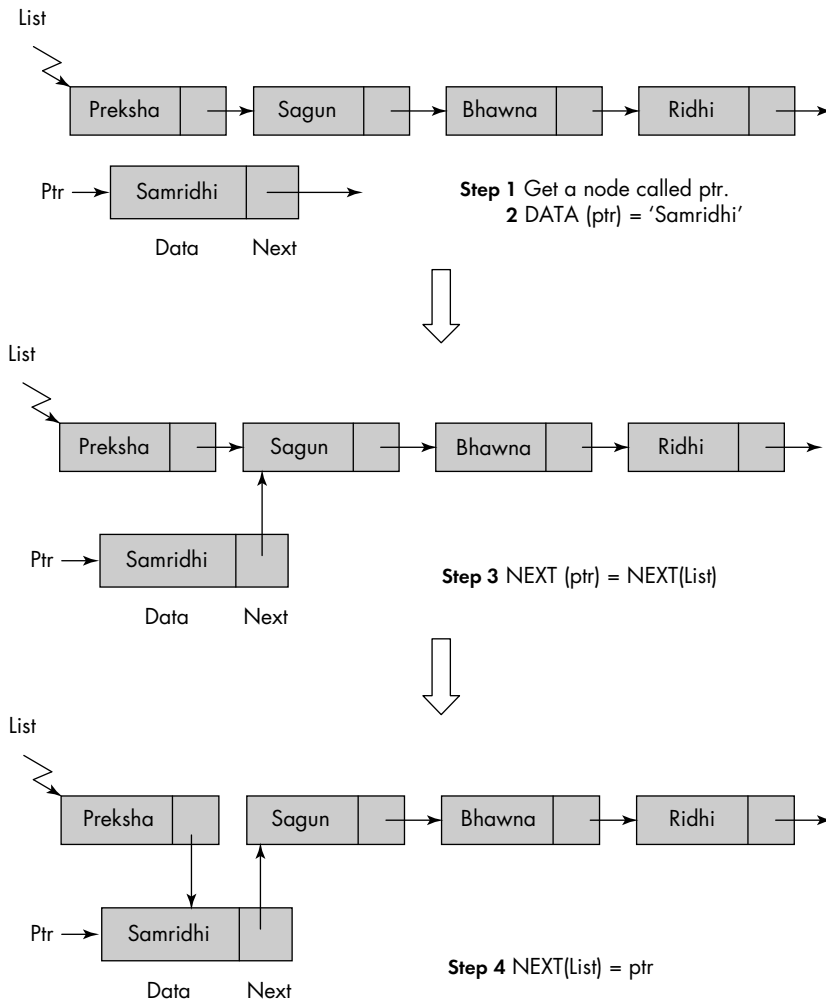


Fig. 6.3 Insertion of a node in the linked list

It may be noted that the insertion operation is a very simple and easy operation in a linked list because, unlike arrays, no copying or shifting operations are required.

Similarly, it is easy to delete an element from a linked list. To delete Bhawna from the list given in Figure 6.3, we need only to redirect the next pointer of Sagun to point to Ridhi as shown in Figure 6.4. This can be done by the following the following steps:

Step

1. Point a pointer called ptr1 to the node containing the name 'Bhawna'.
2. Point a pointer called ptr2 to the node containing the name 'Sagun'.
3. Next (ptr2) = Next(ptr1)
4. Return ptr1

The effect of the above code segment is shown in Figure 6.4.

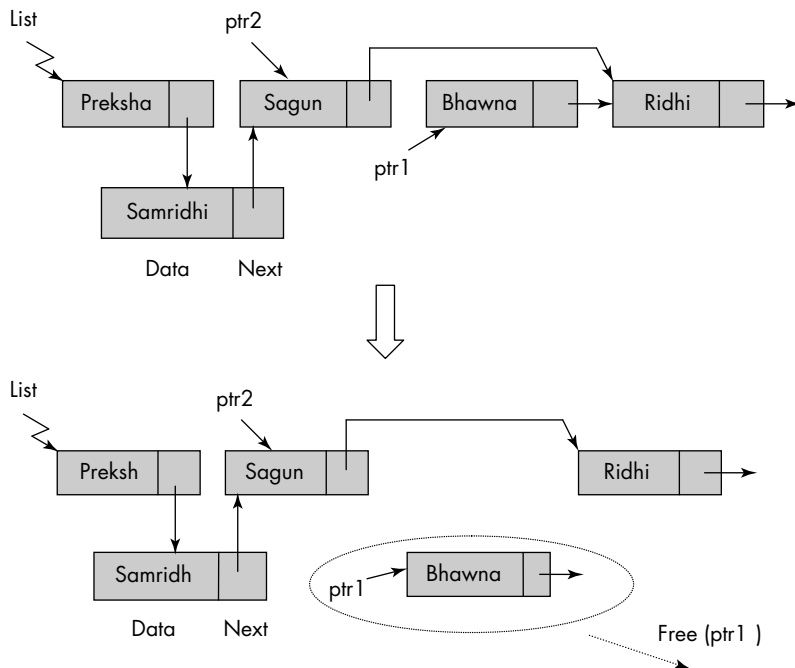


Fig. 6.4 Deletion of a node

From the above discussion, it is clear that unlike arrays, there is no longer a direct correspondence between the logical order of elements present in a list and the way the nodes are arranged physically in the linked list (see Figure 6.5).

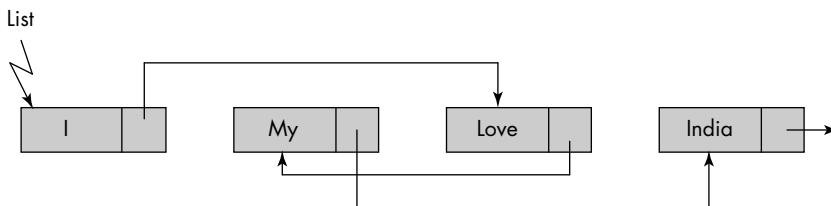


Fig. 6.5 Logical versus physical order in a linked list

The physical order of the nodes shown in Figure 6.5 is “I My Love India” whereas the logical order is “I Love My India”. It is because of this feature that a linked list offers a very easy way of insertions and deletions in a list without the physical shifting of elements in the list. While manipulating a linked list, the programmer keeps in mind only the logical order of the elements in the list. Physically, how the nodes are stored in the memory is neither known to nor the concern of the programmer.

6.3 OPERATIONS ON LINKED LISTS

The linked list is a dynamic data structure. The following operations are associated with linked lists:

- (1) Creation of a linked list
- (2) Traversing a linked list
- (3) Searching in a linked list
- (4) Inserting a node in a linked list
- (5) Deleting a node from a linked list

6.3.1 Creation of a Linked List

From the above discussion, we know that a node of a linked list is essentially a structure because it contains data of different types. In addition to the information part, it contains a pointer that can point to a node, i.e., to itself or to some other node. Such structures are called self-referential structures.

6.3.1.1 Self-referential Structures When a member of a structure is declared as a pointer to the structure itself, then the structure is called self-referential structure. Consider the following declaration:

```
struct chain {    int val;
struct chain *p;
};
```

The structure called 'chain' consists of two members: val and p. The member val is a variable of type int whereas the member p is a pointer to a structure of type chain. Thus, the structure chain has a member that can point to a structure of type chain or may be itself. This type of self-referencing structure can be viewed as shown in Figure 6.6.

Since pointer p can point to a structure variable of type chain, we can connect two such structure variables, A and B, to obtain a linked structure as shown in Figure 6.7.

The linked structure given in Figure 6.7 can be obtained by the following steps:

- (1) Declare structure chain.
- (2) Declare variables A and B of type chain.
- (3) p(A) = B

These steps have been coded in the program segment given below:

```
struct chain {    /* declare structure chain */
int val;
chain *p;
};

struct chain A, B; /* declare structure variables A and B */
A.p = &B;          /* connect A to B*/
```

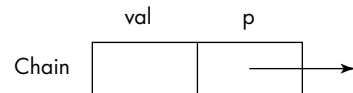


Fig. 6.6 Self-referential structure chain

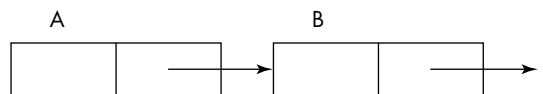


Fig. 6.7 Linked structure

From Figure 6.7 and the above program segment, we observe that the pointer `p` of structure variable `B` is dangling, i.e., it is pointing to nowhere. Such pointer can be assigned to `NULL`, a constant indicating that there is no valid address in this pointer. The following statement will do the desired operation:

```
B.p = NULL;
```

The data elements in this linked structure can be assigned by the following statements:

```
A.val = 50;
```

```
B.val = 60;
```

The linked structure now looks like as shown in Figure 6.8.

We can see that the members of structure `B` can be reached by the following two methods:

- (1) From its variable name `B` through dot operator.
- (2) From the pointer `p` of variable `A` because it is also pointing to the structure `B`. However, in this case the arrow operator (`->`) is needed to access the field called `val` as shown below.

Consider the statements given below:

```
printf ("\n the contents of member val of B = %d", B.val);
printf ("\n the contents of member val of B = %d", A -> p.val);
```

Once the above statements are executed, the output would be:

```
The contents of member val of B = 60
```

```
The contents of member val of B = 60
```

The linked structures have great potential and can be used in numerous programming situations such as lists, trees, etc. The major advantage of linked structures is that the pointers can be used to allocate dynamic memory. Consider the following declaration:

```
struct chain *ptr;
```

The above statement declares a pointer called `ptr`, which can point to a structure of type `chain` [see Figure 6.9 (a)]. Let us use `ptr` to take one such structure through dynamic allocation with the help of `malloc()` function of 'C'. From Chapter 5, we know that the `malloc()` function needs the size

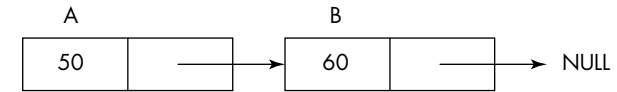
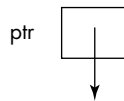


Fig. 6.8 Value assignment to data elements



(a) The dangling pointer



(b) The dynamic allocation of a structure of type `chain` pointed by the pointer `ptr`

Fig. 6.9 Dynamic allocation of a structure

of the memory that is desired by the programmer. Therefore, in the first step, given below, the size of the structure has been obtained and in the second step the memory of that size has been asked through `malloc()` function.

```
Size = sizeof (struct chain);
ptr = (struct student *) malloc (Size);
```

The effect of the above statements is shown in Figure 6.9 (b). It may be noted that the node pointed by `ptr` has no name. In fact all dynamic memory allocations are anonymous.

Let us now use self-referential structures to create a linked list of nodes of structure type given in Figure 6.10. While creating the linked list, the student data is read. The list is also travelled to print the data.

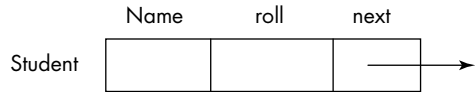


Fig. 6.10 The self-referential structure 'student'

We will use the following self-referential structure for the purpose of creating a node of the linked list:

```
struct stud {
    char name [15];
    int roll;
    struct stud*next;
};
```

The required linked list would be created by using three pointers—`first`, `far`, and `back`. The following algorithm would be employed to create the list:

Algorithm `createLinkedList()`

```
{
    Step
    1. Take a new node in pointer called first.
    2. Read first -> name and first -> roll.
    3. Point back pointer to the same node being pointed by first, i.e.,
       back = first.
    4. Bring a new node in the pointer called far.
    5. Read far -> name and far -> roll.
    6. Connect next of back to far, i.e., back -> next = far.
    7. Take back to far, i.e., back = far.
    8. Repeat steps 4 to 7 till whole of the list is constructed.
    9. Point next of far to NULL, i.e., far -> next = NULL.
    10. Stop.
}
```

The effect of the above steps is shown in Figure 6.11.

The required program is given below:

```
#include <stdio.h>
#include <alloc.h>
struct student {
    char name [15];
```

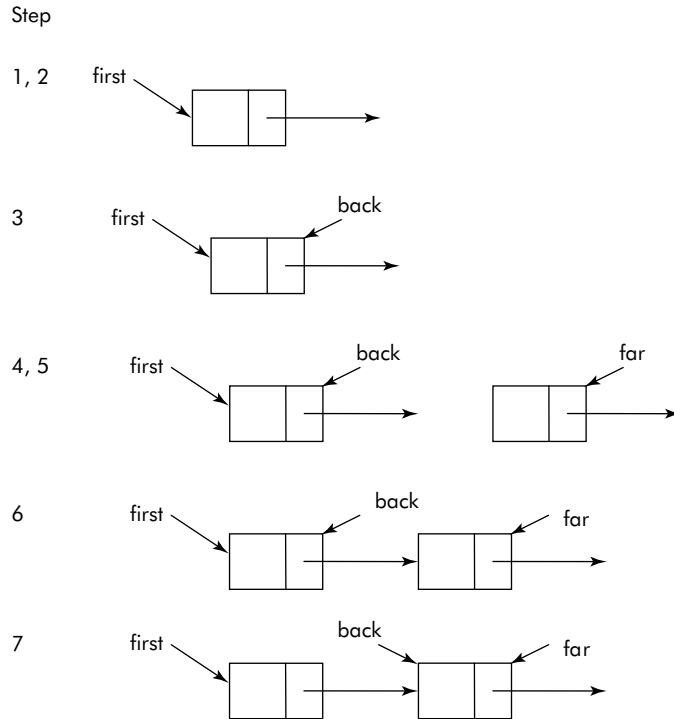


Fig. 6.11 Creation of a linked list

```
int roll;
struct student *next;
};

struct student *create(); /* Function that creates the linked list */
void dispList(struct student * First); /* Function to display linked list */

void main()
{
    struct student *First;
    First = create();
    dispList(First);
}

struct student *create()
{
    struct student * First,*Far, *back;
    int N,i;
    int Sz;
    Sz = sizeof (struct student);
    printf("\n Enter the number of students in the class");
```

```

scanf("%d", &N);
if ( N == 0) return NULL; /* List is empty */
                          /* Take first node */
First = (struct student *) malloc (Sz);
                          /* Read the data of the first student */

printf ("\n Enter the data");
printf ("\n Name: "); fflush(stdin); gets(First->name);
printf ("\n Roll: "); scanf("%d", &First->roll);
First->next = NULL;
                          /* point back where First points */
back = First;

for (i =2; i <=N; i++)
{
    /* Bring a new node in Far */
    Far = (struct student *) malloc (Sz);
    /* Read Data of next student */
    printf ("\n Name: "); fflush(stdin); gets(Far->name);
    printf ("\n Roll: "); scanf("%d", &Far->roll);
    /* Connect Back to Far */

    back->next = Far;
    /* point back where Far points */

    back = Far;
}
/* Repeat the process */
Far->next = NULL;

return First;           /* return the pointer to the linked list */
}

/* Print the created linked list */
void dispList(struct student *First)
{ struct student *Far;
Far = First;           /* Point to the first node */

printf ("\n The List is ....");
while (Far != NULL)
{
    printf (" %s - %d, ", Far->name, Far->roll);
    /* Point to the next node */

    Far = Far->next;
}
}

```

It may be noted that the next pointer of the last node of a linked list always points to NULL to indicate the end of the list. The pointer First points to the beginning of the list. The address of an in between node is not known and, therefore, whenever it is desired to access a particular node, the travel to that node has to start from the beginning.

6.3.2 Travelling a Linked List

Many a times, it is required to travel whole or a part of the list. For example, the operations such as counting of nodes, processing of information of some nodes, or printing of the list, etc. requires traversal of the list.

Consider the linked list shown in Figure 6.12. It is pointed by First. Let us travel this list for the purpose of counting the number of nodes in the list.

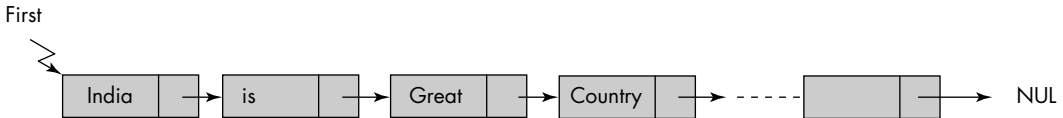


Fig. 6.12 The list pointed by First

Let us use a pointer `ptr` to travel the list. A variable called `count` would be used to count the number of nodes in the list. The travel stops when a NULL is encountered. The following algorithm called `travelList()` travels the given list pointed by First.

Algorithm `travelList()`

```

{
  Step
  1. if First == NULL then { print "List Empty"; Stop}
  2. ptr = First;    /* ptr points to the node being pointed by First */
  3. count = 0;
  4. while (ptr != Null)
  {
    4.1 count = count + 1;
    4.2 ptr = NEXT(ptr);
  }
  5. print "The number of nodes =", count;
  6. End
}
  
```

In the above algorithm, step 4.2, given below, is worth noting:

```
ptr = NEXT(ptr)
```

The above statement says that let `ptr` point to the same location which is currently being pointed by `NEXT` of `ptr`. The effect of this statement on the list of Figure 6.12 is shown in Figure 6.13.

It may be noted that the step 4.2 takes the `ptr` to next node. Since this statement is within the scope of the while loop, `ptr` travels from the First to the NULL pointer, i.e., the last node of the list.

Example 1: Write a program that travels a linked list consisting of nodes of following struct type. While traveling, it counts the number of nodes in the list. Finally, the count is printed.

```

struct student {
  char name [15];
  int roll;.
  struct student *next;
};
  
```



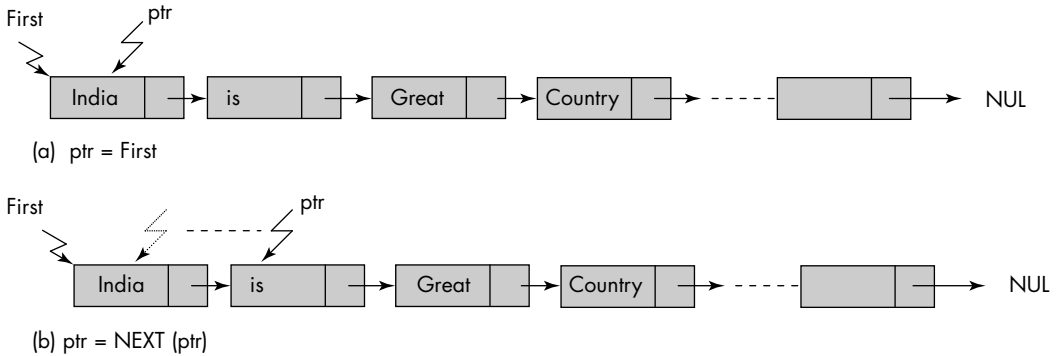


Fig. 6.13 Significance of the operation $\text{ptr} = \text{NEXT}(\text{ptr})$

Solution: A linked list of nodes of student type would be created by using a function called `create()`. The nodes would be added to the list until the user enters a roll number less than or equal to 0, i.e., (`roll <= 0`). A function `countNode()` based on the algorithm `travelList()` would be used to travel and count the nodes in the list.

The required program is given below:

```
/* This program travels a linked list and counts the number of nodes */
#include <stdio.h>
#include <alloc.h>
struct student {
    char name [15];
    int roll;
    struct student *next;
};
struct student *create();
int countNode(struct student *First);

void main()
{
    int count;
    struct student *First;
    First = create();
    count = countNode(First);
    printf("\n The number of nodes = %d", count);
}

/* This function creates a linked list */
struct student *create()
{
    struct student *First, *Far, *back;
    int i;
    int Sz;
```

```

Sz = sizeof (struct student);

        /* Take first node */
First = (struct student *) malloc (Sz);
        /* Read the data of the first student */

printf ("\n Enter the data of the first student");
printf ("\n Name: "); fflush(stdin); gets(First->name);
printf ("\n Roll: "); scanf("%d", &First->roll);
First->next = NULL;

if (First->roll <=0){
    printf ("\n Nodes = 0");
    First = NULL;
    exit(1);} /* Empty list*/

        /* point back where First points */
back = Far = First;

while (Far->roll >0)
{
    /* Bring a new node in Far */
    Far = (struct student *) malloc (Sz);
        /* Read Data of next student */
    printf ("\n Name: "); fflush(stdin); gets(Far->name);
    printf ("\n Roll: "); scanf("%d", &Far->roll);
    if (Far->roll <=0) break; /* End of the list */
        /* Connect Back to Far */
    back->next = Far;
        /* point back where Far points */
    back = Far;
} /* repeat the process */
back->next = NULL;
return First;
}

        /* Travel and count the number of nodes */
int countNode(struct student *First)
{
    struct student *ptr;
    int count;
    count =0;
    ptr = First; /* Point to the first node */
    while (ptr != NULL)
    { count = count + 1;
        /* Point to the next node */
        ptr = ptr->next;
    }
    return count;
}

```

Example 2: Write a program that travels a linked list consisting of nodes of following struct type. Assume that the number of students in the list is not known.

```
struct student {
    char name [15];
    int roll;
    struct student *next;
};
```

While travelling the list of students, it is split into two sub-lists pointed by two pointers: *front_half* and *back_half*. The *front_half* points to the front half of the list and the *back_half* to the back half of the list. If the number of students is odd, the extra student should go into the front list.

Solution: The list of students would be created in the same fashion as done in Example 1. The number of nodes would be counted during the creation of the linked list itself. Assume that the linked list is being pointed by First. The following steps would be followed to split the list in the desired manner:

- (1) Compute the middle of the list.
- (2) Point *front_half* to First.
- (3) Point another pointer Far to First. Let Far travel the list starting from the first and reach to the middle.
- (3) Point *back_half* to the node that succeeds the middle of the list.
- (4) Attach NULL to the next pointer of the middle node, now pointed by Far.
- (5) Print the two lists, i.e., pointed by *front_half* and *back_half*.

The required program is given below:

```
/* This program travels a linked list and splits it into two halves */

#include <stdio.h>
#include <alloc.h>
struct student {
    char name [15];
    int roll;
    struct student *next;
};

struct student * create (int *count);
struct student * split (struct student *First, int *count);
void dispList(struct student *First);
void main()
{ struct student *First,*front_half, *back_half;
  int count;
  First = create(&count);
  if (count == 1)
  {
      printf ("\n The list cannot be split");
      front_half = First;
  }
  else
  {
```



```

    front_half = First;
    back_half = split (First, &count);
}
printf ("\n The First Half...");
dispList(front_half);
printf ("\n The Second Half...");
dispList(back_half);
}

struct student *create (int *count)
{
    struct student *First, *Far, *back;
    int Sz;
    Sz = sizeof (struct student);
        /* Take first node */
    First = (struct student *) malloc (Sz);
        /* Read the data of the first student */
    printf ("\n Enter the data of the first student");
    printf ("\n Name: "); fflush(stdin); gets(First->name);
    printf ("\n Roll: "); scanf("%d", &First->roll);
    First->next=NULL;
    if (First->roll <=0){
        printf ("\n Nodes = 0");
        First = NULL;
        exit(1);} /* Empty list*/
    *count =1;
        /* point back where First points */
    back = Far = First;
    while (Far->roll >0)
    {
        /* Bring a new node in Far */
        Far = (struct student *) malloc (Sz);
        /* Read Data of next student */
        printf ("\n Name: "); fflush(stdin); gets(Far->name);
        printf ("\n Roll: "); scanf("%d", &Far->roll);
        if (Far->roll <=0) break; /* End of the list */
        *count = *count + 1;
        /* Connect back to Far */
        back->next = Far;
        /* point back where Far points */
        back = Far;
    } /* repeat the process */
    back->next = NULL;
    return First;
}

struct student *split (struct student *First, int *count)
{

```

```

    struct student *Far,*back;
    int middle,i;
    Far = First;    /* Point to the first node */
                    /* Split the list */

    {
        /* compute the middle */
        if ( (*count % 2) == 0) middle = *count/2;
        else middle = *count / 2 + 1;
        /* Travel the list and split it */
        for (i =1; i <= middle; i++)
        {
            back =Far;
            Far = Far->next;
        }
        back->next = NULL;
        return Far;
    }
}

/* Print the list */
void dispList(struct student *First)
{
    struct student *Far;
    Far = First;
    while (Far != NULL)
    {
        printf(" %s, %d ", Far->name, Far->roll);
        Far = Far->next;
    }
}

```

6.3.3 Searching a Linked List

Search is an operation in which an item is searched in the linked list. In fact, the list is travelled from the first node to the last node. If a visited node contains the item, then the search is successful and the travel immediately stops. This means that in the worst case, whole of the list would be travelled. An algorithm for searching an item 'X' in the list is given below.

In this algorithm a linked list, pointed by First, is travelled. While travelling, the data part of each visited node is compared with the item 'X'. If the item is found, then the search stops otherwise the process continues till the end of the list (i.e., NULL) is encountered. A pointer ptr is used to visit the various nodes of the list.

```

Algorithm searchList()
{
Step
    1. if First == NULL {
        print "List Empty"; exit();}

```

```

2. Read the item X
3. ptr = First;
4. flag = 0;
5. while (ptr != NULL)
    {
        5.1 if ( X== DATA (ptr) then { flag = 1; break; }
        5.2. ptr = NEXT(ptr);
    }
6. if (flag ==1) then print "Item found" else print "Item not found"
7. Stop
}

```

Example 3: Modify program of Example 1 such that the program searches the record of a student whose roll number is given by the user.

Solution: Instead of writing the code for creation and display of linked lists again and again, now onwards we would use the functions `create()` and `dispList()` developed in earlier programs. A new function based on algorithm `searchList()` would be written and used to search the record of a student.

The required program is given below:

```

/* This program searches an item in a linked list */

#include <stdio.h>
#include <alloc.h>
#include <process.h>

struct student {
    char name [15];
    int roll;
    struct student *next;
};

struct student *create();
int searchList(struct student *First, int Roll);

void main()
{
    struct student *First;
    int Roll, result;
    First =create();

    printf ("\n Enter the Roll of the student to be searched");
    scanf("%d", &Roll);
    result = searchList(First, Roll);
    if (result == 1)
        printf ("\n The Student found");
    else
        printf ("\n The Student not found");
}

```



```

struct student *create()    /* Any of the create() function designed in
above examples can be used */
{
}

int searchList(struct student *First, int Roll)
{ struct student *Far;
  int flag;
  Far = First;    /* Point to the first node */
  flag = 0;
  while (Far != NULL)
  {
    if (Roll == Far->roll)
    {
      flag=1;
      break;
    }
    /* Point to the next node */
    Far = Far->next;
  }
  if (flag == 1) return 1;
  else return 0;
}

```

6.3.4 Insertion in a Linked List

We have already appreciated that linked lists are most suitable and efficient data structures for insertion and deletion operations. The insertion of a node in a linked list involves the following three steps:

- (1) Take a node. Store data into the node.
- (2) Search the location in the list where the node is to be inserted.
- (3) Insert the node.

The location where the node is to be inserted in a linked list can either be at the beginning or at any other arbitrary position in the list. An algorithm that inserts a node at the beginning of the list is given below:

In this algorithm, a node pointed by `ptr` is inserted in a linked list whose first node is pointed by the pointer `First`.

```

Algorithm insertAtHead()
{
  Step
  1. Take a node in ptr.
  2. Read DATA(ptr).
  3. If (First == NULL) then
  {
    3.1 First = ptr;
    3.2 NEXT (First) = NULL;
  }
}

```

```

else
{
    3.1 NEXT(Ptr) = First;
    3.2 First = ptr;
}
4. Stop
}

```

The list structure before and after the insertion operation is shown in Figure 6.14.

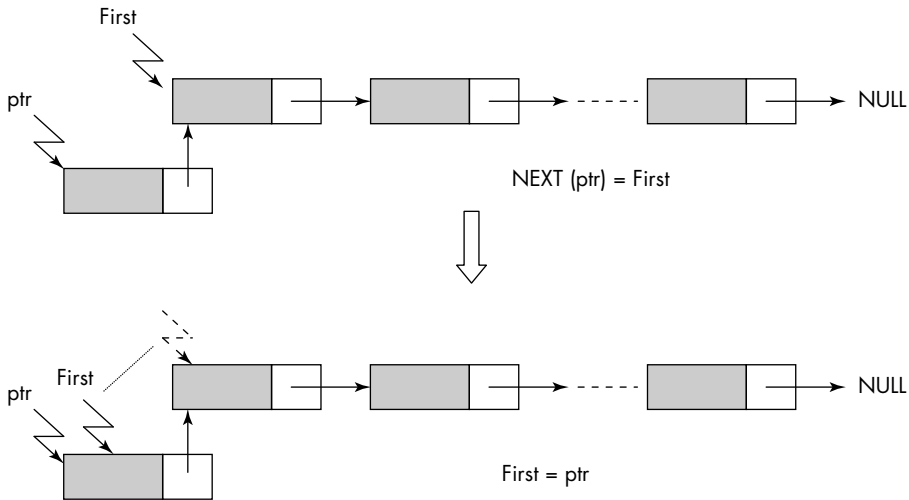


Fig. 6.14 Insertion at the beginning of a list

It may be noted that after the insertion operation, both `First` and `ptr` are pointing to first node of the list.

If it is desired to insert a node at arbitrary position in a linked list, then it can either be inserted after a node or before a node in the list. An algorithm for insertion of a node after a selected node in a linked list is below.

Let `First` be the pointer to the linked list. In this algorithm, a node is inserted after a node with data part equal to 'val'. A pointer '`ptr`' travels the list in such a way that each visited node is checked for data part equal to `val`. If such a node is found then the new node, pointed by a pointer called '`nptr`', is inserted.

Algorithm `insertAfter()`

```

{
    Step
    1. ptr = First
    2. while (ptr != NULL)
    {
        2.1 if (DATA (ptr) = val)
        { take a node in nptr;
          Read DATA(nptr);

```

```

    Next (nptr) = Next (ptr);
    Next (ptr) = nptr;
    break;
}
2.2 ptr = Next(ptr);
}
3. Stop
}

```

The list structure before and after the insertion operation is shown in Figure 6.15.

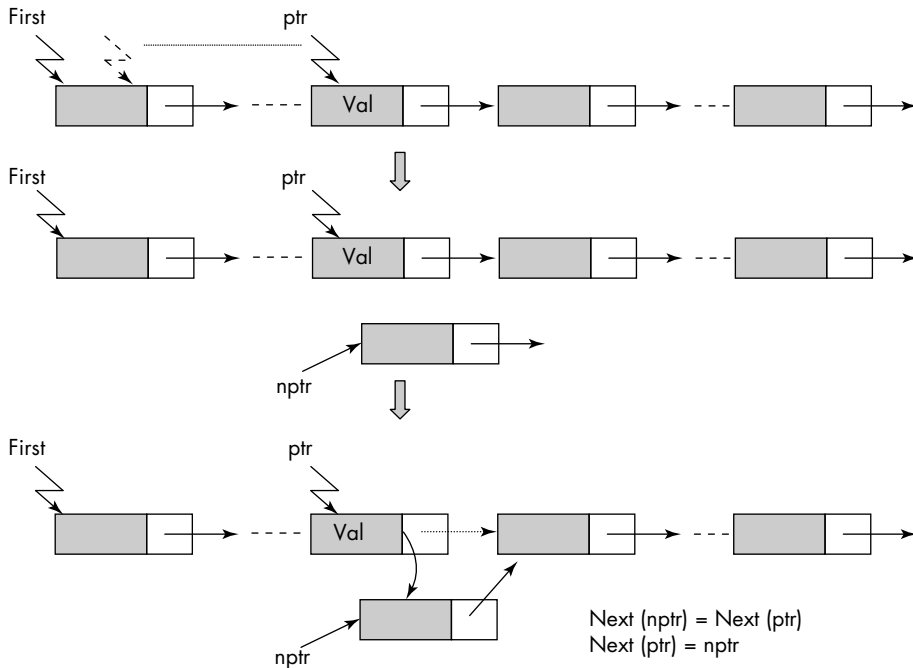


Fig. 6.15 Insertion after a selected node in the list

Example 4: Modify Example 3 such that after creation of a linked list of students, the program asks from the user to enter data for the student which is desired to be inserted into the list after a given node.

Solution: The required program is given below. It takes care of both the cases, i.e., it inserts the student as per his roll number in the list, which can be at the beginning or at any other location in the list.

```

/* This program inserts an item in a linked list */
#include <stdio.h>
#include <alloc.h>
#include <process.h>
struct student {
    char name [15];

```

```

    int roll;
    struct student *next;
};

struct student *create();
struct student * insertList(struct student *First, int Roll, int *flag);
void dispList(struct student *First);
void main()
{
    struct student *First;
    int Roll, result;
    First = create();
    if (First != NULL)    /* Insert in between */
    {
        printf ("\n Enter the roll no. of the student");
        printf ("\n after which the insertion is to be made :");
        scanf("%d", &Roll);
    }
    First = insertList(First, Roll, &result);
    if (result == 1)
    {printf ("\n The list....");
        dispList(First);
    }
    else
        printf ("\n The place not found");
}

struct student * create ()    /* Here any of the create() function designed
in above examples can be used */
{
}

struct student* insertList(struct student *First, int Roll, int *flag)
{
    struct student *ptr,*Far;
    int Sz;    /* Insert at the Head of the list */
    if ((First == NULL) || (First->roll > Roll))
    {
        ptr = (struct student *) malloc (Sz);
        /* Read Data of next student */
        printf ("\n Name: "); fflush(stdin); gets(ptr->name);
        printf ("\n Roll: "); scanf("%d", &ptr->roll);
        ptr->next =First;
        First = ptr;
        *flag = 1;    /* indicates that insertion was made*/
    }
    else
    { Far = First;    /* Point to the first node */
        *flag = 0;

```

```

while (Far != NULL)
{
    if (Roll == Far->roll)
    {
        *flag=1;
        break;          /* The location of the insertion found*/
    }
    /* Point to the next node */
    Far = Far->next;
}
if (*flag==1)
{
    ptr = (struct student *) malloc (Sz);
    /* Read Data of student for insertion */
    printf("\n Enter the data of student");
    printf ("\n Name: "); fflush(stdin); gets(ptr->name);
    printf ("\n Roll: "); scanf("%d", &ptr->roll);

    ptr->next = Far->next; /* Insert node */
    Far->next = ptr;
}
}
ptr = First;
return ptr;
}
void dispList(struct student *First) /* Here any of the dispList()
function designed in above examples can be used */
{
}

```

Note: It is easier to insert a node after a selected node in a linked list as only one pointer is needed to select the node. For example, in the above program, the *Far* pointer has been used to select the node after which the insertion is made.

However, if insertion is needed to be made before a selected node then it is cumbersome to insert node with only one pointer. The reason being that when the node is found before which the insertion is desired, there is no way to reach to its previous node and make the insertion. The easier and elegant solution is to use two pointers (*Far* and *back*) which move in tandem, i.e., the pointer *Far* is followed by *back*. When *Far* reaches the desired location, *back* points to its previous location as shown in Figure 6.16.

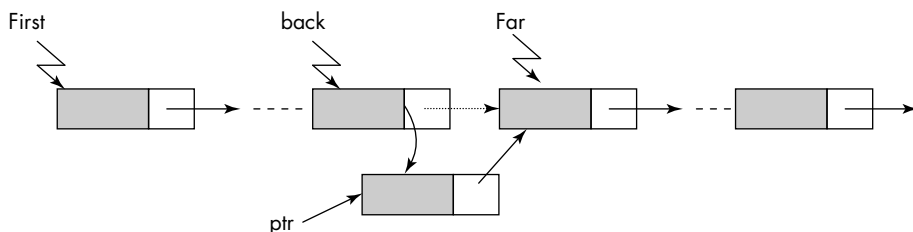


Fig. 6.16 Insertion before a given node in a linked list

An algorithm for insertion of `data` before a selected node in a linked list pointed by `First` is given below:

```

Algorithm insertBefore()
{
    Far = First;
    If (DATA(Far) == 'Val')    /* Check if the first node is the desired node */
    {
        take a new node in ptr;
        Read DATA(ptr);
        Next(ptr) = Far;
        First = ptr;
        Stop;
    }
    while (Far != NULL )
    {
        back = Far;
        Far = Next(Far);
        If (DATA(Far) == 'Val')    /* Check if the node is the desired node */
        {
            take a new node in ptr;
            Read DATA(ptr);
            Next(ptr) = Far;
            Next(back) = ptr;
            break;
        }
    }
}

```

Example 5: A sorted linked list of students in the order of their roll numbers is given. Write a program that asks from the user to enter `data` for a missing student which is desired to be inserted into the list at its proper place.

Solution: The sorted linked list of students in the order of their roll numbers would be created by using function `create()`, developed in the previous programs. The `data` of the student which is to be inserted is read and its proper location found, i.e., before the node whose roll number is greater than the roll number of this incoming student. The algorithm `insertBefore()` would be used for the desired insertion operation.

The required program is given below:

```

/* This program inserts a node in a linked list */
#include <stdio.h>
#include <alloc.h>
#include <process.h>
struct student {
    char name [15];
    int roll;
    struct student *next;
}

```

```

};
struct student *create();
struct student * insertList(struct student *First, struct student newStud,
int *flag);
void dispList(struct student *First);
void main()
{
    struct student *First, stud;
    int Roll, result;
    First = create();
    printf ("\n Enter the data of the student which is to be inserted");

    scanf("%s %d", &stud.name, &stud.roll);
    First = insertList(First, stud, &result);
    if (result == 1)
    {printf ("\n The list....");
        dispList(First);
    }
    else
        printf ("\n The place not found");
}
struct student * create ()    /* Here any of the create() function designed
in above examples can be used */
{
}

struct student* insertList(struct student *First, struct student newStud,
int *flag)
{
    struct student *ptr,*Far,*back;
    int Sz;          /* Insert at the Head of the list */
    if ((First == NULL) || (First->roll > newStud.roll))
    {
        Far = (struct student *) malloc (Sz);
                /* Read Data of next student */
        strcpy(Far->name, newStud.name);
        Far->roll = newStud.roll;
        Far->next =First;
        First = Far;
        *flag =1;    /* indicates that insertion was made*/
    }
    else
    { ptr =back = First;    /* Point to the first node */
        *flag =0;
        while (ptr != NULL)
        {
            ptr = ptr->next;

```

```

        if (newStud.roll > back->roll && newStud.roll < ptr->roll)
        {
            *flag=1;
            break; /* The location of the insertion found*/
        }
        back = ptr;
    }
    if (*flag==1)
    {
        Far = (struct student *) malloc (Sz);
                /* Read Data of student for insertion */
        strcpy(Far->name, newStud.name);
        Far->roll = newStud.roll;

        Far->next = ptr; /* Insert node */
        back->next = Far;
    }
}
ptr = First;
return ptr;
}
void dispList(struct student *First) /* Here any of the dispList() function
designed in above examples can be used */
{
}

```

6.3.5 Deleting a Node from a Linked List

A delete operation involves the following two steps:

- (1) Search the list for the node which is to be deleted.
- (2) Delete the node.

The first step requires that two pointers in tandem be used to search the node needs to be deleted. After the search, one pointer points to the selected node and the other points to immediate previous node.

An algorithm for deletion of a node from a list pointed by `First` is given below. It uses two pointers, `ptr` and `back`, that travel the list in such a way that each visited node is checked. If it is the node to be deleted, then `ptr` points to this selected node and the pointer `back` points to its immediate previous node as shown in Figure 6.17.

Algorithm `delNode()`

```

{
    1. if (DATA (First) = 'VAL' /* Check if the starting node is the desired
one */
    {
        1.1 ptr = First;
        1.2 First = Next (First);
        1.3 Free(ptr);
    }
}

```

```

1.4 Stop;
}
2. Back = First;
3. ptr = Next (First)
4. while (ptr != NULL)
{
  4.1 if (DATA(ptr) = 'VAL')
  {
    Next(Back) = Next (ptr);    /* The node is deleted */
    Free(ptr);
    break;
  }
  4.2 back = ptr;
  4.3 ptr = Next(ptr);
}
}
5. Stop
}

```

The list structure before and after the deletion operation is shown in Figure 6.17.

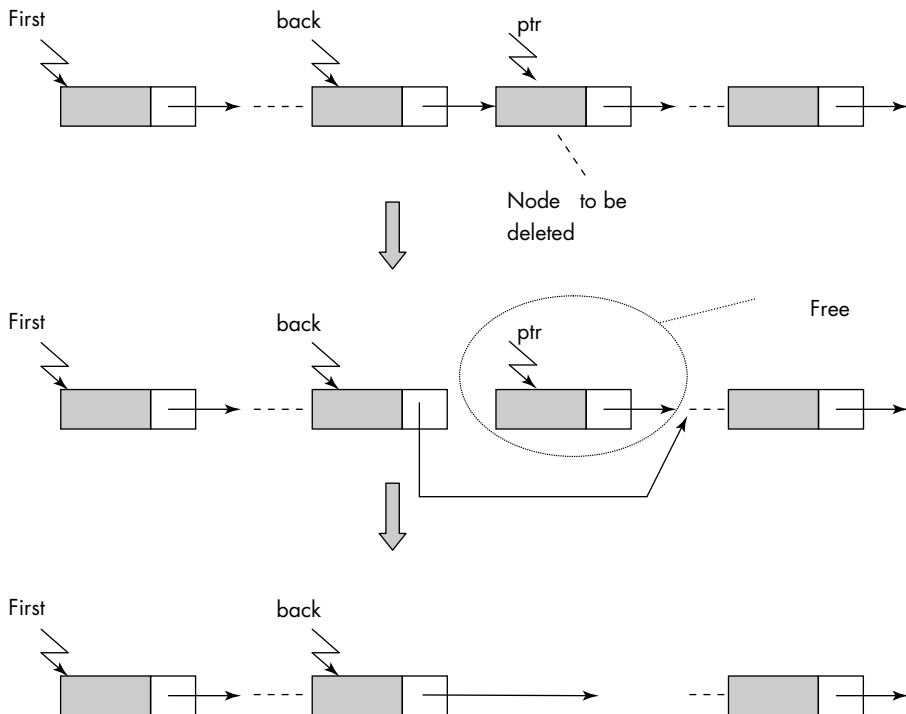


Fig. 6.17 The deletion operation

Example 6: A linked list of students in the order of their roll numbers is given. Write a program that asks from the user to enter roll number of the student which is desired to be deleted from the list.

Solution: The linked list of students in the order of their roll numbers would be created by using function `create()`, developed in the previous programs. The roll number of the student which is to be deleted is read and its proper location found. The algorithm `delNode()` would be used for the desired deletion operation.

The required program is given below:

```
/* This program deletes a node from a linked list */

#include <stdio.h>
#include <alloc.h>
#include <process.h>
struct student {
    char name [15];
    int roll;
    struct student *next;
};
struct student *create();
struct student *delNode(struct student *First, int roll, int *flag);
void dispList(struct student *First);
void main()
{
    struct student *First;
    int Roll, result;
    First = create();
    printf ("\n Enter the Roll of the student which is to be deleted");

    scanf(" %d", &Roll);
    First = delNode(First, Roll, &result);
    if (result == 1)
    {printf ("\n The list....");
     dispList(First);
    }
    else
        printf ("\n The place not found");
}

struct student *create ()    /* Here any of the create() function designed
in above examples can be used */
{
}

struct student* delNode(struct student *First, int Roll, int *flag)
{
    struct student *ptr,*Far,*back;
    int Sz;
    if (First == NULL) { *flag=0; return First; }
```

```

        /* delete from the Head of the list */
if (First->roll == Roll)
{
    ptr = First;
    First = First->next;
    free(ptr);
    *flag = 1;          /* indicates that deletion was made*/
}
else
{
    ptr = back = First;    /* Point to the first node */
    *flag = 0;
    while (ptr != NULL)
    {
        ptr = ptr->next;
        if (Roll == ptr->roll)
        {
            *flag = 1;
            break;        /* The location of the deletion found*/
        }
        back = ptr;
    }
    if (*flag == 1)
    {
        back->next = ptr->next;
        free(ptr);
    }
}
return First;
}

void dispList(struct student *First)    /* Here any of the dispList() function
designed in above examples can be used */
{
}

```

6.4 VARIATIONS OF LINKED LISTS

There are many limitations of a linear linked list. Some of them are as follows:

- (1) The movement through the linked list is possible only in one direction, i.e., in the direction of the links.
- (2) As discussed above, a pointer is needed to be pointed to a node before the node that needs to be deleted.
- (3) Insertion or appending of a node at the end of the linked list can only be done after travelling whole of the list.

In order to overcome the limitations, many variations of linear or singly linked lists have been suggested in the literature. Some of the popular variations of linear linked lists are given in subsequent sections.

6.4.1 Circular Linked Lists

A circular linked list is a linked list in which the last node points to the node at the beginning of the list as shown in Figure 6.18. Thus, it forms a circle, i.e., there is no NULL pointer. In fact, every node has a successor.

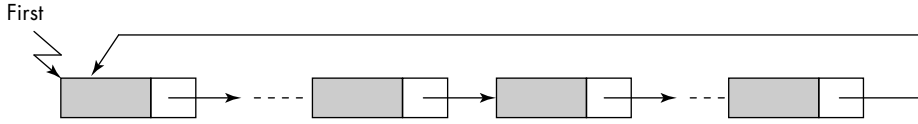


Fig. 6.18 Circular linked list

The main advantage of the circular linked list is that from any node, in the list, one can reach to any other node. This was not possible in a linear linked list, i.e., there was no way of reaching to nodes preceding the current node.

A modified algorithm for creation of circular linked list is given below:

Algorithm createCirLinkList()

```
{
  Step
  1. Take a new node in pointer called first.
  2. Read Data (First).
  3. Point back pointer to the same node being pointed by first, i.e.,
     back = first.
  4. Bring a new node in the pointer called far.
  5. Read Data (far).
  6. Connect next of back to far, i.e., back -> next = far.
  7. Take back to far, i.e., back = far.
  8. Repeat steps 4 to 7 till whole of the list is constructed.
  9. Point next of far to First, i.e., far -> next = First.
  10. Stop.
}
```

Example 7: Write a function create() that creates a circular linked list of N students of following structure type:

```
struct student {
  char name [15];
  int roll;
  struct student *next;
};
```

Solution: We have used the algorithm createCirLinkList() to write the required function which is given below:

```
struct student *create()
{
  struct student * First, *Far, *back;
```

```

int N,i;
int Sz;
Sz = sizeof (struct student);
printf("\n Enter the number of students in the class");
scanf("%d", &N);
if ( N==0) return NULL;    /* List is empty */
    /* Take first node */
First = (struct student *) malloc (Sz);
    /* Read the data of the first student */

printf ("\n Enter the data");
printf ("\n Name: "); fflush(stdin); gets(First->name);
printf ("\n Roll: "); scanf("%d", &First->roll);
First->next = NULL;
    /* point back where First points */
back = First;
for (i =2; i <=N; i++)
{
    /* Bring a new node in Far */
    Far = (struct student *) malloc (Sz);
    /* Read Data of next student */
    printf ("\n Name: "); fflush(stdin); gets(Far->name);
    printf ("\n Roll: "); scanf("%d", &Far->roll);
    /* Connect Back to Far */
    back->next = Far;
    /* point back where Far points */
    back = Far;
}
    /* Repeat the process */
Far->next = First;    /* Point the last pointer to the first node */
return First; /* return the pointer to the linked list */
}

```

Example 8: Write a program that travels a circular linked list consisting of nodes of following struct type. While travelling, it counts the number of nodes in the list. Finally, the count is printed.

```

struct student {
    char name [15];
    int roll;
    struct student *next;
};

```



Solution: A circular linked list of nodes of student type would be created by using the function called `create()` developed in Example 7. The list would be travelled from the first node till we reach back to the first node. While travelling, the visited nodes would be counted. The required program is given below:

```

/* This program travels a circular linked list and counts the number of
nodes */

#include <stdio.h>
#include <alloc.h>

```



```

struct student {
    char name [15];
    int roll;
    struct student *next;
};

struct student *create();
int countNode(struct student * First);
void main()
{
    int count;
    struct student *First;
    First = create();
    count = countNode(First);
    printf("\n The number of nodes = %d", count);
}

struct student *create()    /* Here the create() function of Example 7 can
be used */
{
    /* Travel and count the number of nodes */
    int countNode(struct student * First)
    {
        struct student *ptr;
        int count;
        count =0;
        ptr = First;    /* Point to the first node */

        do
        { count = count +1;
          /* Point to the next node */
          ptr = ptr->next;
        }
        while (ptr != First);    /* The travel stops at the first node */
        return count;
    }
}

```

Note: The above representation of circular linked list (Figure 6.18) has a drawback as far as the insertion operation is concerned. Even if the node is to be added at the head of the list, the complete list will have to be travelled so that the last pointer is made to point to the new node. This operation ensures that the list remains circular after the insertion. It may be noted that this extra travel was not required in the case of linear linked list.

The above drawback of circular linked lists can be avoided by shifting the First pointer to the last node in the list as shown in Figure 6.19.

It may be noted that by having this representation, both ends of the list can be accessed through the First. The First points to one end and the Next (First) points to another.

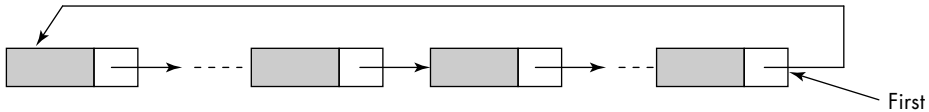


Fig. 6.19 Another representation of a circular linked list

A node can be added at the head of the list by the following algorithm:

Algorithm insertHead()

```
{
  Step
  1. Take a new node in ptr.
  2. Read DATA(ptr).
  3. Next(ptr) = Next (First).
  4. Next(First) = ptr.
}
```

The effect of the above algorithm is shown in Figure 6.20.

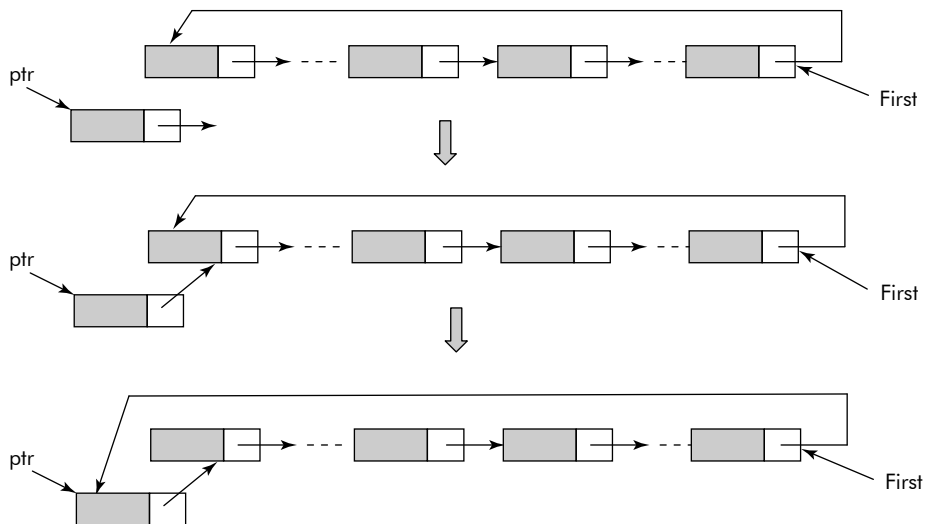


Fig. 6.20 Insertion of a node at the head of a circular linked list without travel

A node can be added at the tail (i.e., last) of the list by the following algorithm:

Algorithm insertTail()

```
{ Step
  1. Take a new node in ptr.
  2. Read DATA(ptr).
  3. Next(ptr) = Next (First).
```

```
4. Next(First) = ptr.  
5. First = ptr.  
}
```

The effect of the above algorithm is shown in Figure 6.21.

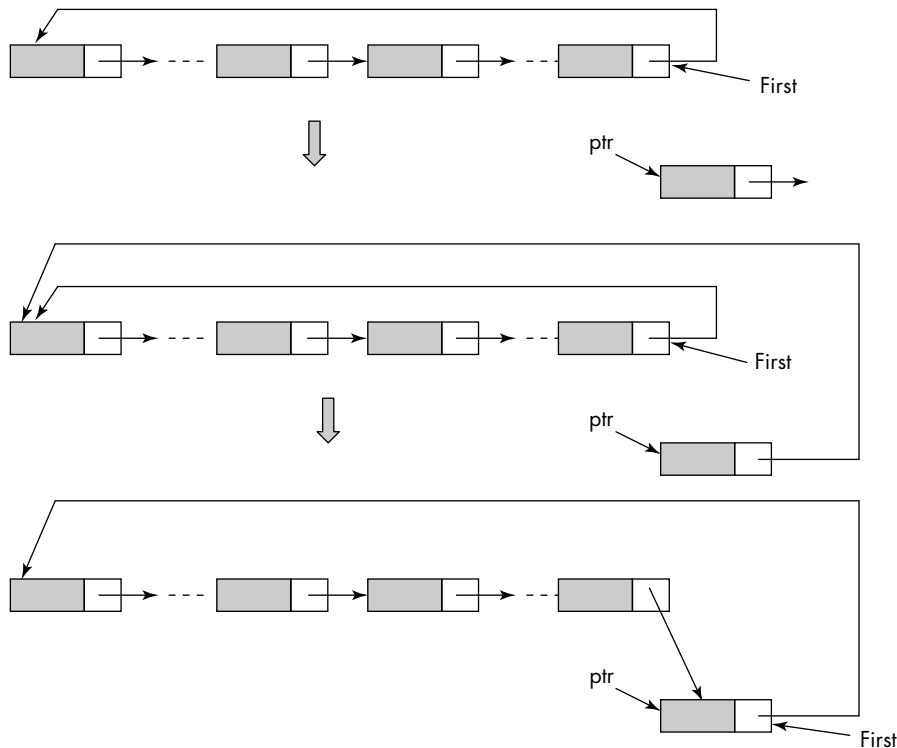


Fig. 6.21 Insertion of a node at the tail of a circular linked list without travel

It may be noted that in both the cases of insertion (at the head and tail), no travel of the circular linked list was made. Therefore, the new representation has removed the drawback of the ordinary circular linked list. In fact, it offers the solution in the order of $O(1)$.

6.4.2 Doubly Linked List

Another variation of linked list is a doubly linked list. It is a list in which each node contains two links: leftLink and rightLink. The leftLink of a node points to its preceding node and the rightLink points to the next node in the list (see Figure 6.22).

From Figure 6.22, it can be noticed that a node in the doubly linked list has the following three fields:



Fig. 6.22 Node of a doubly linked list

- (1) Data : for the storage of information.
- (2) leftLink : pointer to the preceding node.
- (3) rightLink : pointer to the next node.

The arrangement of nodes in a doubly linked list is shown in Figure 6.23.

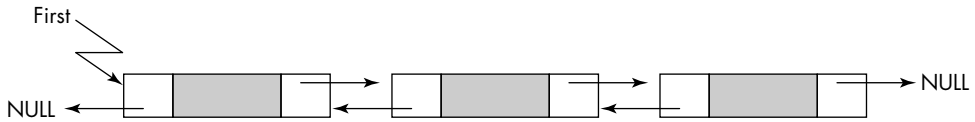


Fig. 6.23 A doubly linked list

The algorithm for creation of doubly linked list is given below:

Algorithm createDList()

```
{
    Step
    1. Take a new node in pointer called First.
    2. Point leftLink of first to NULL, i.e., first->leftLink = NULL.
    3. Read Data(First).
    4. Point back pointer to the node being pointed by First, i.e., back = First.
    5. Bring a new node in the pointer called Far.
    6. Read Data(Far).
    7. Connect rightLink of back to Far, i.e., back->rightLink = Far.
    8. Connect leftLink of Far to Back, i.e., Far->leftLink = Back.
    9. Take back to Far, i.e., back = Far.
    10. Repeat steps 5 to 9 till whole of the list is constructed.
    11. Point rightLink of far to NULL, i.e., Far-> rightLink = NULL.
    12. Stop.
}
```

Example 9: Write a program that uses a function create() to create a doubly linked list of N students of following structure type.

```
struct student {
    char name [15];
    int roll;
    struct student *leftList, *rightList;
};
```

It travels the list in both directions, i.e., forward and backward. The data part of visited nodes is displayed.

Solution: We have used the algorithm createDList() to write the required program. Two functions—dispForward() and dispBackward()—are also written to travel the list in both the directions. The required program is given below:

```
/* This program creates a doubly linked list of students */
#include <stdio.h>
#include <alloc.h>
```

```

struct student {
    char name [15];
    int roll;
    struct student *leftList,*rightList;
};

struct student *create();
void dispForward(struct student *First);
void dispBackward(struct student *First);
void main()
{
    struct student *First;
    First = create();
    dispForward(First);
    dispBackward(First);
}

struct student *create()
{
    struct student * First,*Far, *back;
    int N,i;
    int Sz;
    Sz = sizeof (struct student);
    printf("\n Enter the number of students in the class");
    scanf("%d", &N);
    if ( N==0) return NULL; /* List is empty */
    /* Take first node */
    First = (struct student *) malloc (Sz);
    /* Read the data of the first student */
    printf ("\n Enter the data");
    printf ("\n Name: "); fflush(stdin); gets(First->name);
    printf ("\n Roll: "); scanf("%d", &First->roll);
    First->rightList = NULL;
    First->leftList = NULL;
    /* point back where First points */
    back = First;
    for (i =2; i <=N; i++)
    {
        /* Bring a new node in Far */
        Far = (struct student *) malloc (Sz);
        /* Read Data of next student */
        printf ("\n Name: "); fflush(stdin); gets(Far->name);
        printf ("\n Roll: "); scanf("%d", &Far->roll);
        /* Connect Back to Far */
        back->rightList = Far;
        /* point back where Far points */
        Far->leftList = back;
    }
}

```

```

        back = Far;
    }
    /* Repeat the process */
    Far->rightList = NULL;
    return First;
}

/* Print the created linked in forward direction */

void dispForward(struct student *First)
{ struct student *Far;
  Far = First; /* Point to the first node */
  printf ("\n The Forward List is ....");
  while (Far != NULL)
  {
    printf (" %s - %d, ",Far->name, Far->roll);
    /* Point to the next node */
    Far = Far->rightList;
  }
}

/* Print the created linked in backward direction */

void dispBackward(struct student * First)
{ struct student *Far;
  Far = First; /* Point to the first node */
  while (Far->rightList != NULL)
  { /* Travel to the last node */
    Far = Far->rightList;
  }
  printf ("\n The Back word List is ....");
  while (Far != NULL) /* Travel the list in backward direction*/
  {
    printf (" %s - %d, ",Far->name, Far->roll);
    /* Point to the next node */
    Far = Far->leftList;
  }
}

```

Consider the doubly linked list given in Figure 6.24.

The following relations hold good for the pointer ptr:

- ptr \equiv ptr \rightarrow leftLink \rightarrow rightLink
- ptr \equiv ptr \rightarrow rightLink \rightarrow leftLink

The first relation indicates that the rightLink of leftLink of ptr is pointing to the same node being pointed by ptr. Similarly, the second relation indicates that the leftLink of rightLink of ptr is pointing to the same node being pointed by ptr.

The above relations help us to insert node in a doubly linked list. The discussion on this aspect is given in Section 6.4.2.1.

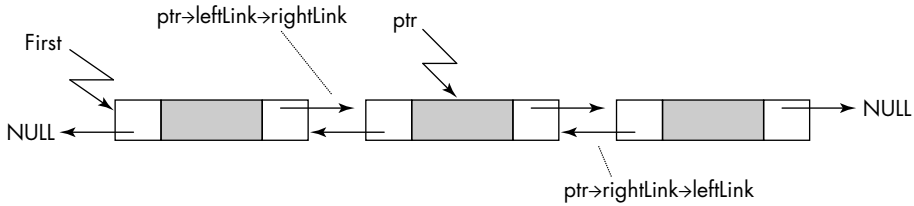


Fig. 6.24 The left of right and right of left is the same node

6.4.2.1 Insertion in Doubly Linked List Insertion of a node x before a node pointed by ptr can be done by the following program segment:

- (1) $x \rightarrow \text{rightLink} = ptr \rightarrow \text{leftLink} \rightarrow \text{rightLink};$
- (2) $ptr \rightarrow \text{leftLink} \rightarrow \text{rightLink} = x;$
- (3) $x \rightarrow \text{leftLink} = ptr \rightarrow \text{leftLink};$
- (4) $ptr \rightarrow \text{leftLink} = x;$

The effect of the above program segment is shown in Figure 6.25. The above four operations have been labelled as 1–4 in Figure 6.25.

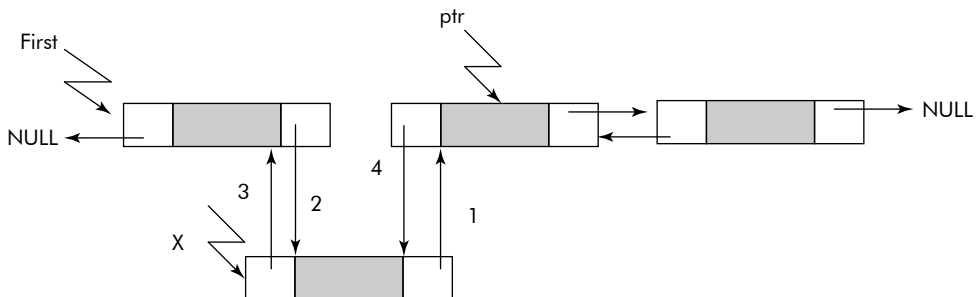


Fig. 6.25 Insertion of node X before a node pointed by ptr

Similarly, insertion of a node x after a node pointed by ptr can be done by the following program segment:

- (1) $x \rightarrow \text{leftLink} = ptr \rightarrow \text{rightLink} \rightarrow \text{leftLink};$
- (2) $ptr \rightarrow \text{rightLink} \rightarrow \text{leftLink} = x;$
- (3) $x \rightarrow \text{rightLink} = ptr \rightarrow \text{rightLink};$
- (4) $ptr \rightarrow \text{rightLink} = x;$

The effect of the above program segment is shown in Figure 6.26. The above four operations have been labeled as 1–4 in Figure 6.26.

6.4.2.2 Deletion in Doubly Linked List Deletion of a node pointed by ptr can be done by the following program segment:

- (1) $ptr \rightarrow \text{left} \rightarrow \text{right} = ptr \rightarrow \text{right};$
- (2) $ptr \rightarrow \text{right} \rightarrow \text{left} = ptr \rightarrow \text{left};$
- (3) $\text{free}(ptr);$

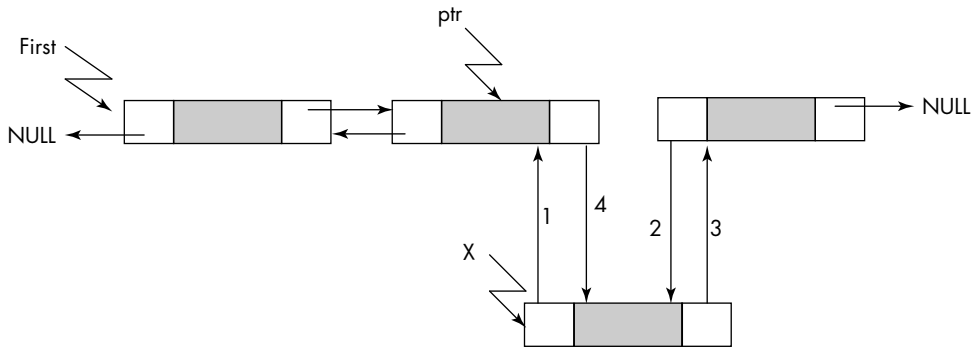


Fig. 6.26 Insertion of node X after a node pointed by ptr

The effect of the above program segment is shown in Figure 6.27. The above three operations have been labeled as 1–3 in Figure 6.27.

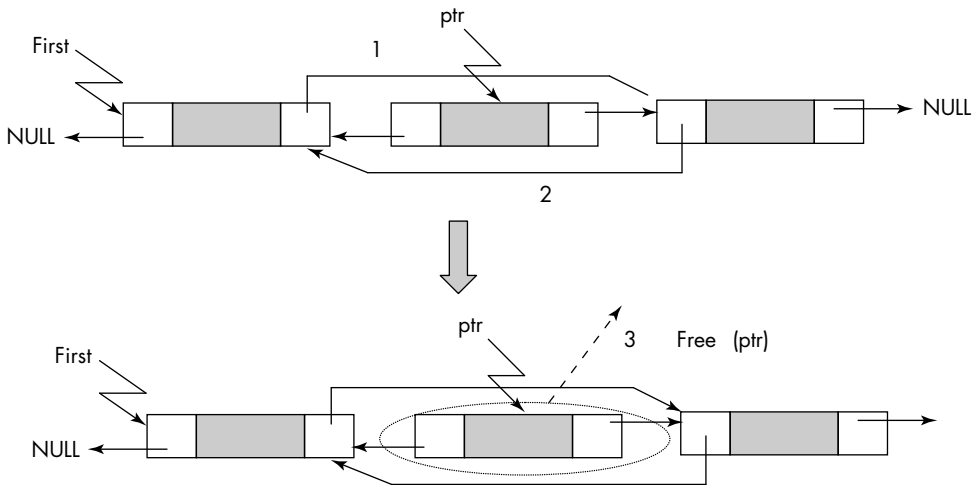


Fig. 6.27 Deletion of a node pointed by ptr

Note: During insertion or deletion of a node, the `leftLink` and `rightLink` pointers are appropriately fixed so that they point to the correct node in the list.

From Figures 6.23 to 6.27, it may be observed that the doubly linked list has the following advantages over the singly linked list:

- (1) With the help of `rightLink`, the list can be traversed in forward direction.
- (2) With the help of `leftLink`, the list can be traversed in backward direction.
- (3) Insertion after or before a node is easy as compared to singly linked list.
- (4) Deletion of a node is also very easy as compared to singly linked list.

It is left as an exercise for the readers to implement the insertion and deletion operations using 'C'.

6.5 THE CONCEPT OF DUMMY NODES

The linked list and its variants, discussed above, have a major drawback. Every algorithm has to take care of special cases. For instance, besides a normal operation, an algorithm has to make sure that it works for the following exceptional or boundary cases:

- (1) the empty list
- (2) on the head element of the list
- (3) on the last element of the list

The above situation occurs when a node in a linked list has no predecessor or a successor. For example, the first node in a linked list has no predecessor and, therefore, insertion or deletion at the end requires a special case. Similarly, an empty circular linked list will not be circular. Same is the case with doubly linked list.

The above anomalous situation can be resolved with the help of a *dummy node*. The dummy node is a *head* or *front* node which is purposefully left blank. The First pointer always points to the dummy node as shown in Figure 6.28.

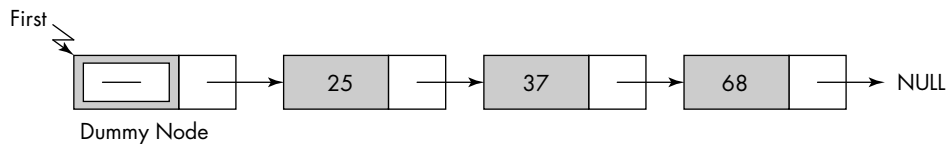


Fig. 6.28 Linked list with a dummy node

It may be noted that the list shown in Figure 6.28 represents a list of integers (25, 37, 68 ...). The dummy node does not contain any value. It may be further noted that first node of the list (containing 25) is also having dummy node as its predecessor and, therefore, insertion at the head of the node can be done in a normal fashion as done in algorithm `insertAfter()` of section 6.3.4.

Similarly, dummy nodes can also be used for circular and doubly linked lists as shown in Figure 6.29.

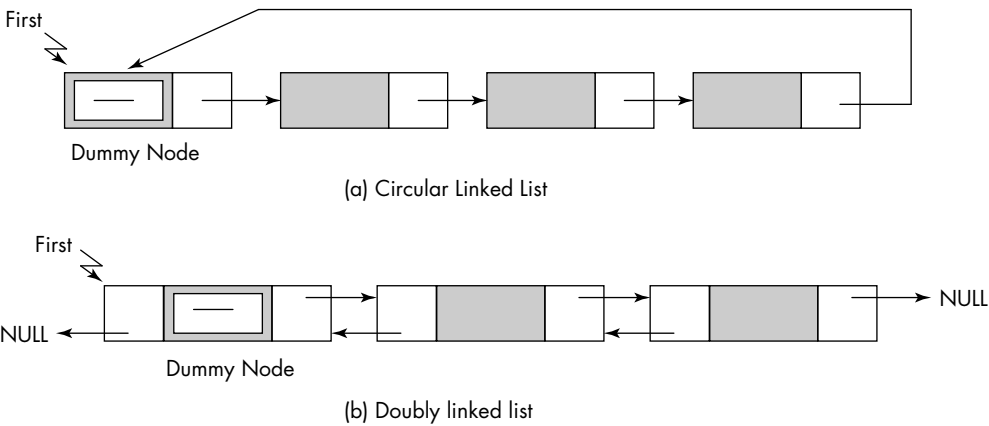


Fig. 6.29 Circular and doubly linked lists with dummy nodes

Note: With the inclusion of a dummy node, an algorithm does not have to check for special or exceptional cases such as existence of empty list or insertion/deletion at the head of the list and the like. Thus, the algorithm becomes easy and applicable to all situations.

Some programmers employ the dummy node to store useful information such as number of elements in the list or the name of the group or set to which the elements belong to. For example, the list shown in Figure 6.28 can be modified so that the dummy node contains information about number of elements present in the list (3 in this case). The modified list is given in Figure 6.30.

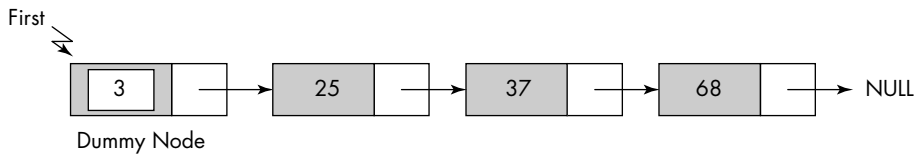


Fig. 6.30 Dummy node contains important information

Many programmers use a dummy node called a *trailer node* at the tail end so that each node in the list has a successor.

Another benefit of dummy node is that an empty circular linked list is also circular in nature. Same is the case with doubly linked list as shown in Figure 6.31.

Empty list without dummy node Empty list with dummy node

Singly linear linked list		
Singly circular linked list		
Doubly linked list		

Fig. 6.31 Comparison of empty lists with and without dummy nodes

It may be noted that with the help of dummy nodes, the basic nature of linked lists remains intact.

6.6 LINKED STACKS

We know that a stack is a LIFO (last in first out) data structure. Earlier in the book, the stack was implemented using an array. Though the array implementation was efficient but a considerable amount of storage space was wasted in the form of unutilized locations of the array. For example, let us assume that at a given time, there are five elements in a stack. If the stack is implemented using 20 locations, then 15 locations out of 20 are being wasted.

A solution to this problem is dynamic storage allocation, i.e., implement stack using linked lists. Since a stack involves frequent additions and deletions at one end called 'Top', the linked list is definitely a better choice from the point of implementation.

Obviously, the structure of the node in a linked stack will also be self-referential. The arrangement of nodes is shown in Figure 6.32. It may be noted that its one end is being pointed by a pointer called Top and the other end is pointing to NULL.

An algorithm for pushing an element on to the stack is given below. In this algorithm, a node pointed by ptr is pushed onto the stack.

```
Algorithm Push ( )
{
    Step
    1. Take a new node in ptr;
    2. Read DATA(ptr);
    3. Next (ptr) = Top;
    4. Top = ptr;
}
```

The structure of linked stack after a push operation is shown in Figure 6.33.

An algorithm for popping an element from the stack is given below. In this algorithm, a node from the Top of the stack is removed. The removed node is pointed by ptr.

```
Algorithm POP ( )
{
    Step
    1. if (Top = NULL) then prompt 'stack empty'; Stop
    2. ptr = Top
    3. Top = Next (Top);
    4. return DATA(ptr)
    5. Free ptr
}
```

The structure of linked stack after a POP operation is shown in Figure 6.34.

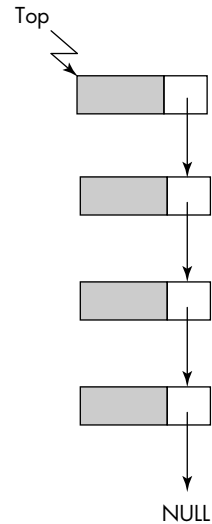


Fig. 6.32 A linked stack

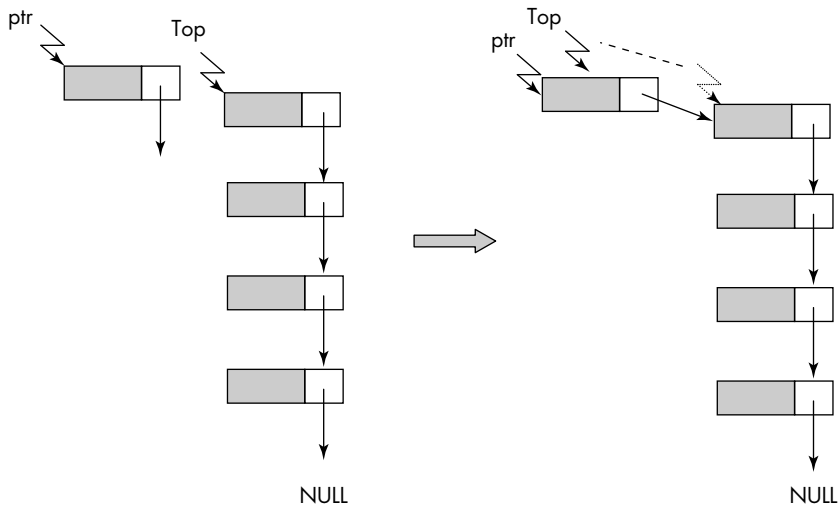


Fig. 6.33 Linked stack after a PUSH operation

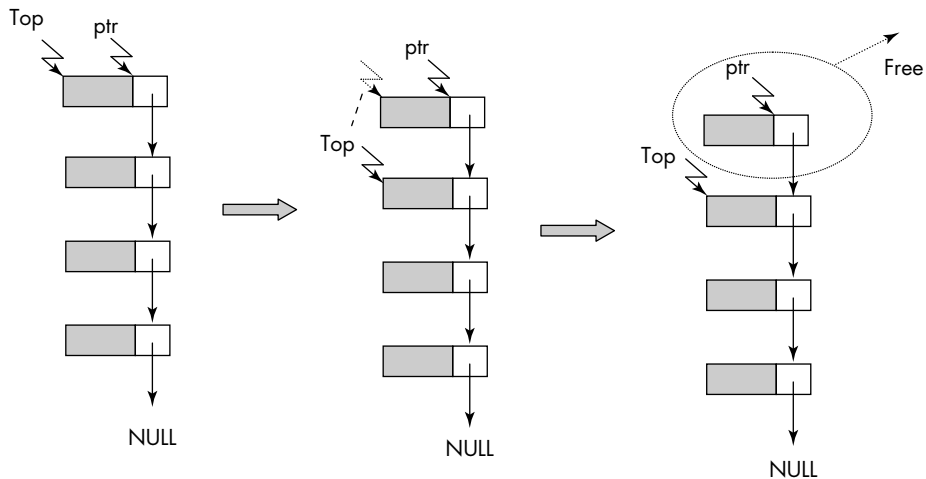


Fig. 6.34 Linked stack after a POP operation

Example 10: Write a menu driven program that simulates a linked stack for integer data items.

Solution: The algorithms `Push()` and `POP()` developed in this section would be used to write the program that simulates a linked stack for integer data items. Each node of the linked stack would be of following structure type:

```

struct node
{
    int item;
    struct node * next;
};

```

The required program is given below:

```

/* This program simulates a linked stack for integer data items */

#include <stdio.h>
#include <alloc.h>
#include <conio.h>

struct node
{
    int item;
    struct node *next;
};

struct node *push (struct node *Top, int item);
struct node *pop (struct node *Top, int * item);
void dispStack(struct node *Top);
void main ()
{
    int item, choice;
    struct node *Top;
    Top = NULL;
    do
    { clrscr();
    printf ("\n Menu ");
    printf ("\n Push          1");
    printf ("\n Pop           2");
    printf ("\n Display        3");
    printf ("\n Quit           4");
    printf ("\n\ Enter Choice: ");
    scanf ("%d", &choice);
    switch (choice)
    {
        case 1: printf ("\n Enter the integer item to be pushed:");
                scanf ("%d", &item);
                Top = push (Top, item);
                break;
        case 2: if (Top == NULL)
                { printf ("\n Stack empty");
                  break;
                }
                Top = pop (Top, &item);
    }
    }
}

```

```
        printf ("\n The popped item = %d", item);
        break;
    case 3: dispStack(Top);
        break;
}
printf ("\n press any key"); getch();
}
while (choice != 4);
}

struct node * push (struct node *Top, int item)
{
    struct node *ptr;
    int size;
    size = sizeof (struct node);
    ptr = (struct node *) malloc (size);
    ptr->item = item;
    ptr->next = Top;
    Top = ptr;
    return Top;
}

struct node *pop (struct node *Top, int *item)
{
    struct node *ptr;
    if (Top == NULL) return NULL;
    ptr = Top;
    Top = Top->next;
    *item = ptr->item;
    free (ptr);
    return Top;
}

void dispStack(struct node *Top)
{
    if (Top == NULL)
        printf ("\n Stack Empty");
    else
    {
        printf ("\n The Stack is: ");
        while (Top != NULL)
        {
            printf (" %d ", Top->item);
            Top =Top->next;
        }
    }
}
```

6.7 LINKED QUEUES

We know that a queue is a FIFO (first in first out) data structure. Since this data structure also involves deletions and additions at **front** end and **rear** end respectively, a linked list is definitely a better choice from the point of implementation.

A linked queue will have the same node structure as in the case of a stack as shown in Figure 6.35. Two pointers called Front and Rear would keep track of the front and rear end of the queue.

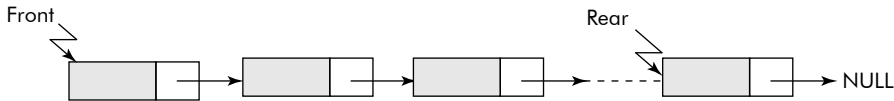


Fig. 6.35 A linked queue

An algorithm for adding an element on to the queue is given below. In this algorithm, a node pointed by ptr is added into the queue.

```
Algorithm AddQueue ( )
{
    Step
    1. Take a new node in ptr;
    2. Read DATA(ptr);
    3. Next (ptr) = Next (Rear);
    4. Next(Rear) = ptr;
    5. Rear = ptr;
}
```

The structure of linked queue after addition operation is shown in Figure 6.36.

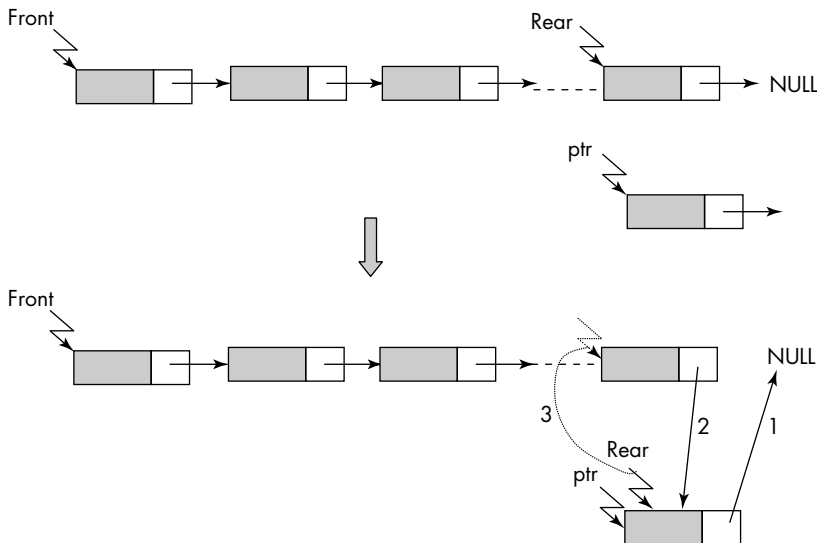


Fig. 6.36 The linked queue after an addition operation

An algorithm for deleting an element from the linked queue is given below. In this algorithm, a node from the front of the queue is removed.

Algorithm delQueue()

```
{
  Step
  1. if (Front == Rear) then prompt 'Queue empty'; Stop
  2. ptr = Front;
  3. Front = Next (Front);
  4. Free ptr
  5. return DATA(Front)
}
```

The structure of linked queue after a deletion operation is shown in Figure 6.37.

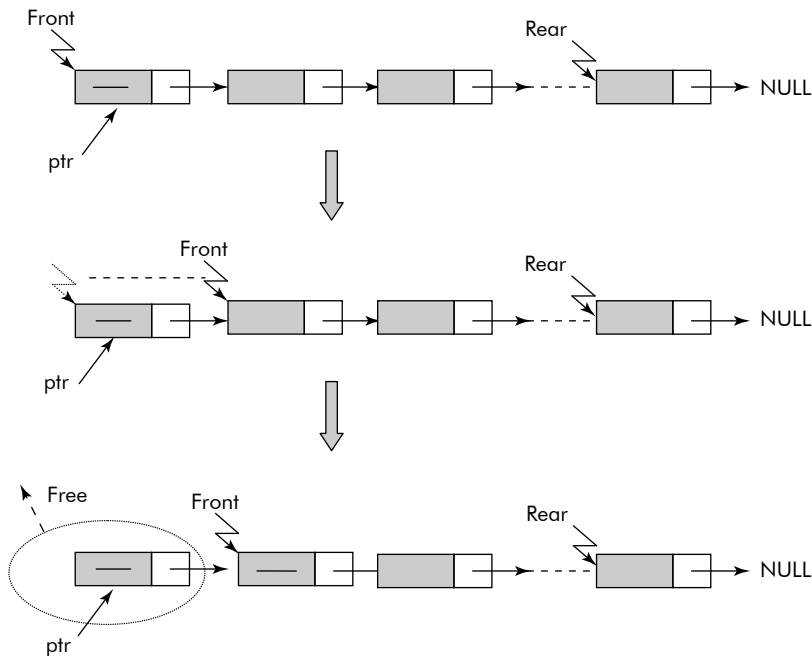


Fig. 6.37 The linked queue after a deletion operation

It may be noted that the Front always points to a blank node, i.e., which does not contain any data. Thus, when the queue is empty then both Front and Rear point to a blank node.

Example 11: Write a menu-driven program that simulates a linked queue for integer data items.

Solution: The algorithms AddQueue() and delQueue() developed in this section would be used to write the program that simulates a linked queue for integer data items. Each node of the linked queue would be of following structure type:

```
struct node
{
  int item;
```



```
    struct node * next;
};
```

The required program is given below:

```
/* This program simulates a linked queue for integer data items */
#include <stdio.h>
#include <alloc.h>
#include <conio.h>

struct node
{
    int item;
    struct node *next;
};

struct node *addQ (struct node *Rear,struct node *Front, int item);
struct node *delQ (struct node *Rear,struct node *Front, int *item);
void dispQ(struct node *Rear,struct node *Front);

void main ()
{
    int item, choice, size;
    struct node *Front, *Rear;
    size = sizeof (struct node);
    Front = (struct node *) malloc (size);
    Front->item = -9999;      /* dummy node */
    Front->next = NULL;
    Rear = Front;           /* Initialize Queue to empty */
    do
    { clrscr();
    printf ("\n Menu ");
    printf ("\n Add          1");
    printf ("\n Delete        2");
    printf ("\n Display        3");
    printf ("\n Quit          4");
    printf ("\n\n Enter Choice : ");
    scanf ("%d", &choice);
    switch (choice)
    {
        case 1: printf ("\n Enter the integer item to be added:");
                scanf ("%d", &item);
                Rear = addQ (Rear, Front, item);
                break;
        case 2: if (Front == Rear)
                { printf ("\n Queue empty");
                  break;
                }
                Front =delQ (Rear, Front, &item);
```

```
        printf ("\n The deleted item = %d", item);
        break;
    case 3: dispQ(Rear, Front);
        break;
}
printf ("\n press any key"); getch();
}
while (choice != 4);
}

struct node * addQ (struct node *Rear, struct node *Front, int item)
{
    int size;
    struct node *ptr;
    size = sizeof (struct node);
    ptr = (struct node *) malloc (size);
    ptr->item = item;
    ptr->next = Rear->next;
    Rear->next = ptr;
    Rear = ptr;
    return ptr;
}

struct node * delQ (struct node *Rear, struct node *Front, int * item)
{
    struct node *ptr;
    ptr = Front;
    Front = Front->next;
    *item = Front->item;
    Front->item = -9999;    /* The dummy node */
    free (ptr);
    return Front;
}

void dispQ(struct node *Rear, struct node *Front)
{
    if (Front == Rear)
        printf ("\n The queue empty");
    else
    { printf ("\n The queue is... ");
        do
        {
            Front = Front->next;
            printf(" %d ", Front->item);
        }
        while (Front != Rear);
    }
}
```

6.8 COMPARISON OF SEQUENTIAL AND LINKED STORAGE

A list can be stored in a one-dimensional array and in a linked list as well. The choice of data structure depends upon the problem at hand and the type of operations to be performed on the data. A summarized comparison of sequential and linked storage is given below:

- Unlike arrays, the linked list is a dynamic data structure. The linked list implementation adds the elements to the list by getting the memory allocated dynamically. Therefore, the number of elements present in the linked list is limited only by the memory available to the operating system. We can say that the limitation of arrays has been effectively handled by the linked list implementation.
- Linked lists store the elements in non-contiguous locations. The nodes are independent of each other.
- Linked lists are useful for storing elements in the order of insertion and subsequent traversal in that order.
- However, the linked lists occupy more space than the arrays because each node contains an extra pointer besides the information.
- Compared to arrays, linked lists are slow in many ways. The dynamic memory is allocated through a system call to operating system making the processing very slow. Moreover, every element is referred to through an extra reference made through a pointer thereby making the processing of linked list further slow.
- Access to an element in a linked list is purely sequential whereas an element in array can be accessed through its index. Thus, arrays are faster in accessing an element.
- Nevertheless, a linked list offers a very easy way of insertions and deletions in a list without the physical shifting of elements in the list.
- Without moving the nodes, the elements of the list can be reordered by simply changing the pointers.

6.9 SOLVED PROBLEMS

Problem1: Write an algorithm which travels a linked list pointed by `List`. It travels the list in such a way that it reverses the links of the visited nodes, i.e., at the end the list becomes reversed.

Solution: An algorithm called `revLink()` is given below. In this algorithm, a linked list pointed by the pointer `List` is travelled with the help of three pointers `back`, `Ahead`, and `ptr`. During the travel, the direction of link of each visited node is reversed, i.e., it points to its immediate preceding node. At the end, the pointer `List` is made to point to the last node as shown in Figure 6.38.

Algorithm `revLink()`

```
{
    Step
    1. if (List == NULL) return List;
    2. ptr = List;
    3. back = List;
    4. Ahead = next(ptr);
    5. next (List) = NULL;
    6. while (Ahead != NULL) repeat steps 7 to 10
    7. ptr = Ahead;
```

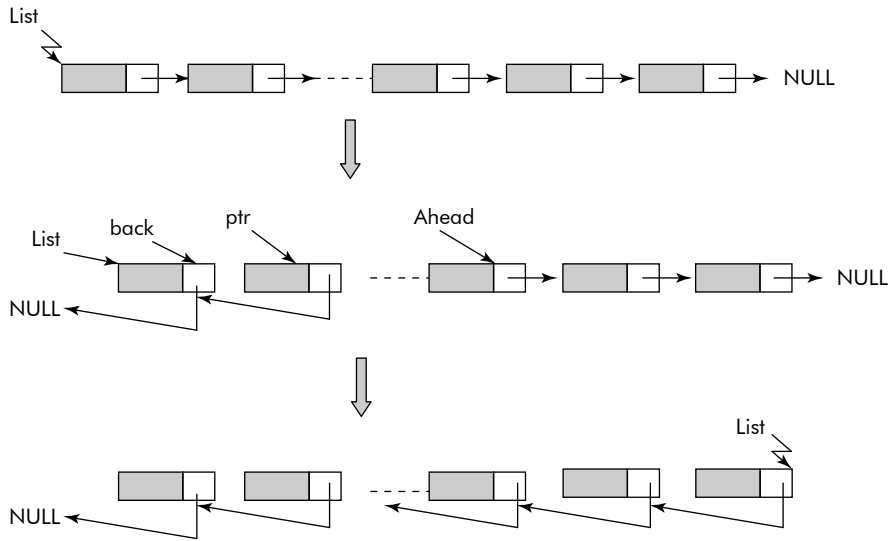


Fig. 6.38 Reversal of links of a linked list

```

8. Ahead = next(Ahead)
9. next (ptr) = back;    /* reverse the link */
10. back = ptr;
11. List = ptr;
12. Stop;
}

```

Problem 2: Write a program which travels a linked list pointed by List. It travels the list in such a way that it reverses the links of the visited nodes, i.e., at the end the list becomes reversed and its contents are displayed.

Solution: The algorithm revLink() developed in Problem 1 is used to write the given program that reverses a linked list of students pointed by a pointer called List. The required function is given below:

```

/* This function reverses the links of a linked list of students */
#include <stdio.h>
#include <alloc.h>
struct student {
    char name [15];
    int roll;
    struct student *next;
};

struct student *create();
struct student *revLink(struct student *List);
void dispList(struct student *First);

```

```

void main()
{
    struct student *First;
    First = create();
    First = revLink(First);
    dispList(First);
}

struct student *create()
{
    struct student *First, *Far, *back;
    int N, i;
    int Sz;
    Sz = sizeof (struct student);
    printf("\n Enter the number of students in the class");
    scanf("%d", &N);
    if ( N==0) return NULL; /* List is empty */
                                /* Take first node */
    First = (struct student *) malloc (Sz);
                                /* Read the data of the first student */
    printf ("\n Enter the data");
    printf ("\n Name: "); fflush(stdin); gets(First->name);
    printf ("\n Roll: "); scanf("%d", &First->roll);
    First->next = NULL;
                                /* point back where First points */
    back = First;
    for (i =2; i <=N; i++)
    {
        /* Bring a new node in Far */
        Far = (struct student *) malloc (Sz);
                                /* Read Data of next student */
        printf ("\n Name: "); fflush(stdin); gets(Far->name);
        printf ("\n Roll: "); scanf("%d", &Far->roll);
                                /* Connect Back to Far */
        back->next = Far;
                                /* point back where Far points */
        back = Far;
    }
                                /* repeat the process */
    Far->next = NULL;
    return First;
}

struct student *revLink( struct student *List) /* The function that reverses
the list */
{
    struct student *back, *ptr, *ahead;
    if (List == NULL)
    {

```

```

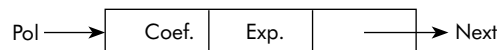
        printf ("\n The list is empty");
        return NULL;
    }
    ptr = List;
    back = List;
    ahead = ptr->next;
    List->next = NULL;
    while (ahead != NULL)
    {
        ptr = ahead;
        ahead = ahead->next;
        ptr->next = back;    /* Connect the pointer to preceding node*/
        back = ptr;
    }
    List = ptr;
    return List;
}

/* Print the created linked list */
void dispList(struct student *First)
{ struct student *Far;
  Far = First;    /* Point to the first node */
  printf ("\n The List is ....");
  while (Far != NULL)
  {
      printf (" %s - %d, ", Far->name, Far->roll);
      /* Point to the next node */
      Far = Far->next;
  }
}

```

Problem 3: Write a program that adds two polynomials Pol1 and Pol2, represented as linked lists of terms, and gives a third polynomial Pol3 such that $\text{Pol3} = \text{Pol1} + \text{Pol2}$.

Solution: We will use the following structure for the term of a polynomial:



As done in Chapter 3, the polynomials will be added by merging the two linked lists. The required program is given below:

```

/* This program adds two polynomials represented as linked lists
pointed by two pointers pol1 and pol2 and gives a third polynomial
pointed by a third pointer pol3 */

# include <stdio.h>
struct term
{

```

```

    int coef;
    int exp;
    struct term *next;
};

struct term * create_pol();

void main()
{
    struct term *pol1,*pol2,*pol3, *ptr1,*ptr2,*ptr3, *back, *bring, *ptr;
    int size;
    size = sizeof (struct term);
    printf ("\n Polynomial 1");
    pol1 = create_pol();
    printf ("\n Polynomial 2");
    pol2 = create_pol();
    ptr1=pol1;
    ptr2=pol2;
    ptr3 =pol3=NULL;
    while (ptr1 != NULL && ptr2 != NULL) /* Add Polynomials by merging */
    {
        if ( ptr1->exp > ptr2->exp)
        {
            if (pol3 == NULL)
            {
                pol3 = (struct term *) malloc(size);
                pol3->coef=ptr1->coef;
                pol3->exp=ptr1->exp;
                ptr3=pol3;
                ptr1= ptr1->next;
                ptr3->next = NULL;
            }
            else
            {
                bring =(struct term *) malloc(size);
                bring->coef=ptr1->coef;
                bring->exp=ptr1->exp;
                ptr3->next =bring;
                ptr3=bring;
                ptr1=ptr1->next;
                ptr3->next =NULL;
            }
        }
        else
        {
            if ( ptr1->exp < ptr2->exp)
            {
                if (pol3 == NULL)
                {

```

```

        pol3 = (struct term *) malloc(size);
        pol3->coef=ptr2->coef;
        pol3->exp=ptr2->exp;
        ptr3=pol3;
        ptr2= ptr2->next;
        ptr3->next = NULL;
    }
else
    {
        bring =(struct term *) malloc(size);
        bring->coef=ptr2->coef;
        bring->exp=ptr2->exp;
        ptr3->next =bring;
        ptr3=bring;
        ptr2=ptr2->next;
        ptr3->next =NULL;
    }
}
else
    {
        if (pol3 == NULL)
        {
            pol3 = (struct term *) malloc(size);
            pol3->coef=ptr1->coef+ptr2->coef;
            pol3->exp=ptr2->exp;
            ptr3=pol3;
            ptr2= ptr2->next;
            ptr1=ptr1->next;
            ptr3->next = NULL;
        }
        else
        {
            bring =(struct term *) malloc(size);
            bring->coef=ptr1->coef+ptr2->coef;
            bring->exp=ptr2->exp;
            ptr3->next =bring;
            ptr3=bring;
            ptr2=ptr2->next;
            ptr1=ptr1->next;
            ptr3->next =NULL;
        }
    }
}
}
/* while */
/* Append rest of the Polynomial */
if (ptr2->next ==NULL) ptr3->next = ptr1;

```



```

        if (ptr1->next ==NULL) ptr3->next = ptr2;
            /* Print the Final Polynomial */
            ptr = pol3;
            printf ("\n Final Polynomial ...");
            while (ptr != NULL)
            {
                printf ("%d ^ %d ", ptr->coef, ptr->exp);
                if (ptr->exp !=0) printf ("+");
                ptr = ptr->next;
            }
        }

        /* Read the terms of a polynomoial and create the linked list */
        struct term * create_pol()
        {
            int size;
            struct term *first, *back, *bring;
            size = sizeof(struct term);
            first = (struct term *) malloc(size);
            printf ("\n input Polynomial term by term. Last term with exp =0");
            printf ("\n Enter a term - coef exp");
            scanf ("%d %d", &first->coef, &first->exp);
            first->next = NULL;
            back = first;
            while (back->exp != 0)
            {
                bring = (struct term *) malloc(size);
                printf ("\n Enter a term - coef exp");
                scanf ("%d %d", &bring->coef, &bring->exp);
                back->next = bring;
                back=bring;
            }
            back->next = NULL;
            return first;
        }
    }

```

EXERCISES

1. What is a linear linked list?
2. What is the need for a linked list? On what account, linked lists are better than arrays?
3. Write a program that takes a linked list pointed by List and traverses it in such a manner that after the travel the links of visited nodes become reversed.
4. Write a program to create a linked list and remove all the duplicate elements of the list.
5. What is a circular linked list? What are its advantages over linear linked list? Write a program/algorithm to insert a node at desired position in a circular linked list.

6. Write a program that takes two ordered linked lists as input and merges them into single ordered linked list.
7. What is a doubly linked list? Write program/algorithm for showing the following operations on a doubly linked list:
 - Create
 - Insert
 - Delete
8. What are the advantages of doubly linked list over singly linked list?
9. Write the program that inserts a node in a linked list after a given node.
10. Write the program that inserts a node in a linked list before a given node.
11. Write a program that deletes a given node from a linked list.
12. Write a function that deletes an element from a doubly linked list.
13. Implement a stack (LIFO) using singly linked list.
14. Write a program to implement a queue (FIFO) using singly linked list.
15. How do you detect a loop in a singly linked list? Write a 'C' program for the same.
16. How do you find the middle of a linked list without counting the nodes? Write a 'C' program for the same.
17. Write a 'C' program that takes two lists List1 and List2 and compare them. The program should return the following:
 - 1 : if List1 is smaller than List2
 - 0 : if List1 is equal to List2
 - 1 : if List1 is greater than List2
18. Write a function that returns the nth node from the end of a linked list.
19. Write a program to create a new linear linked list by selecting alternate element of a given linear linked list.
20. Write an algorithm to search an element from a given linear linked list.
21. Explain the merits and demerits of static and dynamic memory allocation techniques.
22. Define a header linked list and explain its utility.

Trees

CHAPTER 7

CHAPTER OUTLINE

- 7.1 Introduction
- 7.2 Basic Terminology
- 7.3 Binary Trees
- 7.4 Representation of a Binary Tree
- 7.5 Types of Binary Trees
- 7.6 Weighted Binary Trees and Huffman Algorithm
- 7.7 Dynamic Dictionary Coding

7.1 INTRODUCTION

There are situations where the nature of data is specific and cannot be represented by linear or sequential data structures. For example, as shown in Figure 7.1 hierarchical data such as ‘types of computers’ is non-linear by nature.

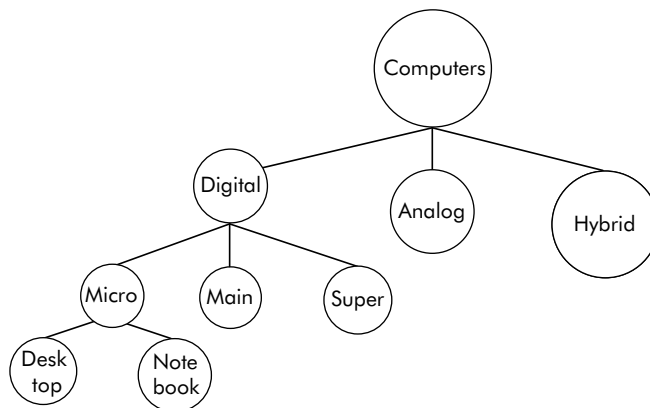


Fig. 7.1 Hierarchical data

Situations such as roles performed by an entity, options available for performing a task, parent–child relationships demand non-linear representation. A tourist can go from Mumbai to Goa by any of the options shown in Figure 7.2.

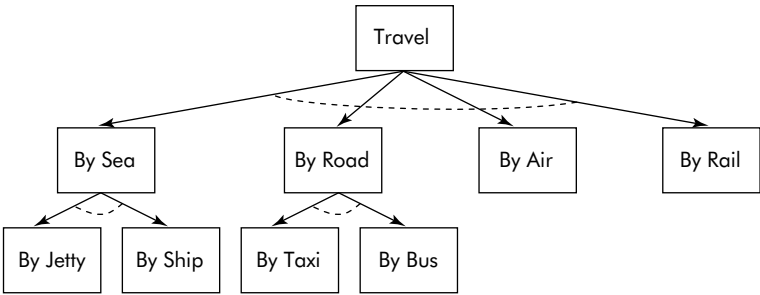


Fig. 7.2 Options available for travelling from Mumbai to Goa

Similarly, the information about folders and files is kept by an operating system as a set of hierarchical nodes as shown in Figure 7.3.

Even the phrase structure of a sentence of a language is represented in a similar fashion (see Figure 7.4).

A critical look at the data representations, shown in figures 7.1–7.4, indicates that the structures are simply upside down trees where the entity at the highest level represents the root and the other entities at lower level are themselves roots for subtrees. For example in Figure 7.2, the entity ‘Travel’ is the **root** of the tree and the options ‘By Sea’ and ‘By Road’, ‘By Air’ and ‘By Rail’ are called **child nodes** of ‘Travel’, the root node. It may be noted that the entities ‘By Road’, ‘By Sea’ are themselves roots of subtrees. The nodes which do not have child nodes are called leaf nodes. For example, the nodes such as ‘By Jetty’, ‘By Ship’, ‘By Air’, etc. are leaf nodes.

A tree can be precisely defined as a set of 1 or more nodes arranged in a hierarchical fashion. The node at the highest level is called a root node. The other nodes are grouped into T1; T2...Tk subsets. Each subset itself is a tree. A tree with zero nodes is called an empty tree.

It may be noted that the definition of a tree is a recursive definition, i.e., the tree has been defined in its own terms. The concept can be better understood by the illustration given in Figure 7.5.

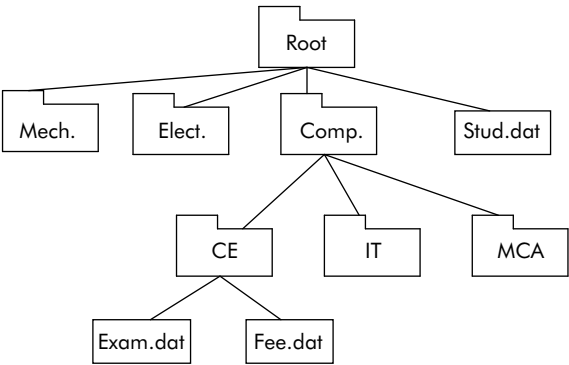


Fig. 7.3 The representation of files and folders

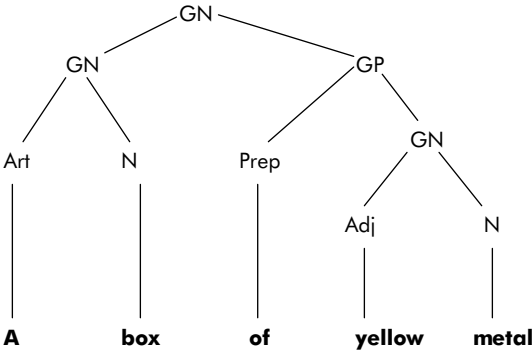


Fig. 7.4 The phrase structure of a sentence

From Figure 7.5, the following components of a tree can be identified:

Root	: A
Child nodes of A	: B, C, D
Sub-Trees of A	: T1, T2, T3
Root of T1	: B
Root of T2	: C
Root of T3	: D
Leaf nodes	: E, G, H, I, J, K, L

Before proceeding to discuss more on trees, let us have a look at basic terminology used in context of trees as given in Section 7.2.

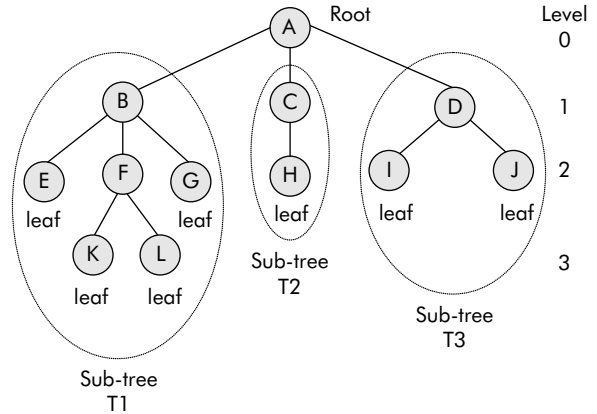


Fig. 7.5 The tree and its related terms

7.2 BASIC TERMINOLOGY

Parent: An immediate predecessor of a node is called its parent.

Example: D is parent of I and J.

Child: The successor of a node is called its child.

Example: I and J are child nodes of node D.

Sibling: Children of the same parent are called siblings.

Example: B, C, and D are siblings. E, F, and G are also siblings.

Leaf: The node which has no children is called a leaf or terminal node.

Example: I, J, K, L, etc. are leaf nodes.

Internal node: Nodes other than root and leaf nodes are called *internal* nodes or *non-terminal* nodes.

Example: B, C, D and F are internal nodes.

Root: The node which has no parent.

Example: A is root of the tree given in Figure 7.5.

Edge: A line from a parent to its child is called an edge.

Example: A-B, A-C, F-K are edges.

Path: It is a list of unique successive nodes connected by edges.

Example: A-B-F-L and B-F-K are paths.

Depth: The depth of a node is the length of its path from root node. The depth of root node is taken as zero.

Example: The depths of nodes G and L are 2 and 3, respectively.

Height: The number of nodes present in the longest path of the tree from root to a leaf node is called the height of the tree. This will contain maximum number of nodes. In fact, it can also be computed as one more than the maximum level of the tree.

Example: One of the longest paths in the tree shown in Figure 7.3 is A-B-F-K. Therefore, the height of the tree is 4.

Degree: Degree of a node is defined as the number of children present in a node. Degree of a tree is defined equal to the degree of a node with maximum children.

Example: Degree of nodes C and D are 1 and 2, respectively. Degree of tree is 3 as there are two nodes A and B having maximum degree equal to 3.

It may be noted that the node of a tree can have any number of child nodes, i.e., branches. In the same tree, any other node may not have any child at all. Now the problem is how to represent such a general tree where the number of branches varies from zero to any possible number. Or is it possible to put a restriction on number of children and still manage this important non-linear data structure? The answer is 'Yes'. We can restrict the number of child nodes to less than or equal to 2. Such a tree is called **binary tree**. In fact, a general tree can be represented as a binary tree and this solves most of the problems. A detailed discussion on binary trees is given in next section.

7.3 BINARY TREES

Binary tree is a special tree. The maximum degree of this tree is 2. Precisely, in a binary tree, each node can have no more than two children. A non-empty binary tree consists of the following:

- A node called the root node
- A left sub-tree
- A right sub-tree

Both the sub-trees are themselves binary trees as shown in Figure 7.6.

Kindly note that contrary to a general tree, an empty binary tree can be empty, i.e., may have no node at all. Examples of valid binary trees are given in Figure 7.7.

Since the binary trees shown in Figure 7.7 are not connected by a root, it is called a collection of *disjoint trees* or a *forest*. The binary tree shown in Figure 7.7 (d) is peculiar tree in which there is no right child. Such a tree is called **skewed to the left** tree. Similarly, there can be a tree **skewed to the right**, i.e., having no left child.

There can be following two special cases of binary tree:

- (1) Full binary tree
- (2) Complete binary tree

A **full binary tree** contains the maximum allowed number of nodes at all levels. This means that each node has exactly zero or two children. The leaf nodes at the last level will have zero children and the other non-leaf nodes will have exactly two children as shown in Figure 7.8.

A **complete binary tree** is a full tree except at the last level where all nodes must appear as far left as possible. A complete binary tree is shown in Figure 7.9. The nodes have been numbered and there is no missing number till the last node, i.e., node number 12.

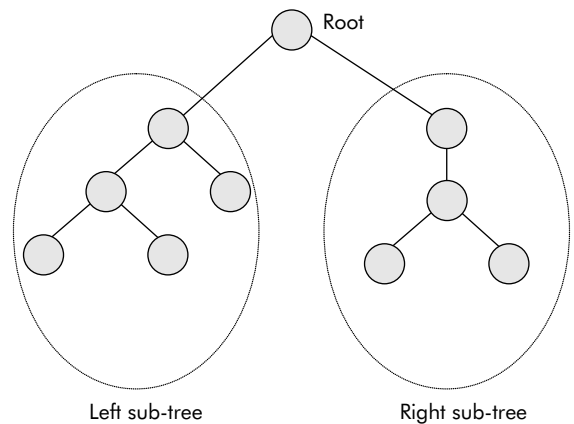


Fig. 7.6 A binary tree

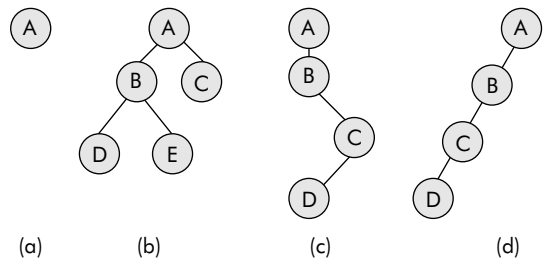


Fig. 7.7 Examples of valid binary trees

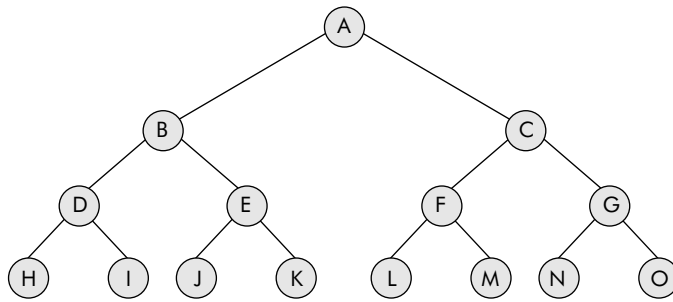


Fig. 7.8 A full binary tree

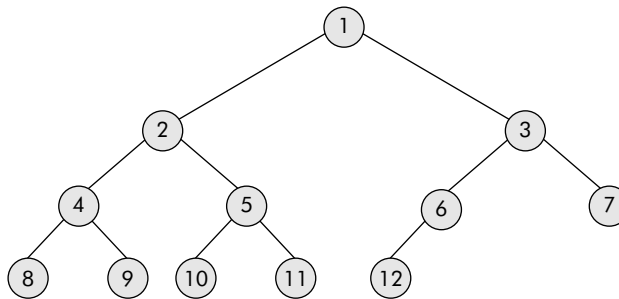


Fig. 7.9 Complete binary tree

Consider the binary tree given in Figure 7.10 which violates the condition of the complete binary tree as the node number 11 is missing. Therefore, it is not a complete binary tree because there is a missing intermediate node before the last node, i.e., node number 12.

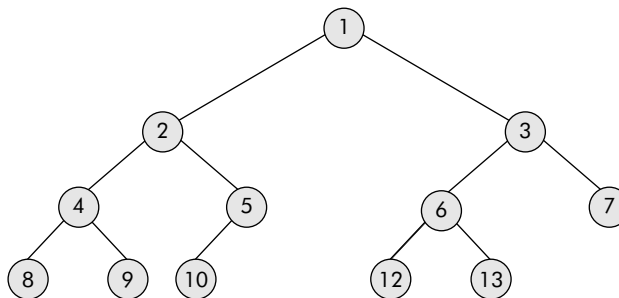


Fig. 7.10 An incomplete binary tree

7.3.1 Properties of Binary Trees

The binary tree is a special tree with degree 2. Therefore, it has some specific properties which must be considered by the students before implementing binary trees; these are:

Consider the binary tree given in Figure 7.10.

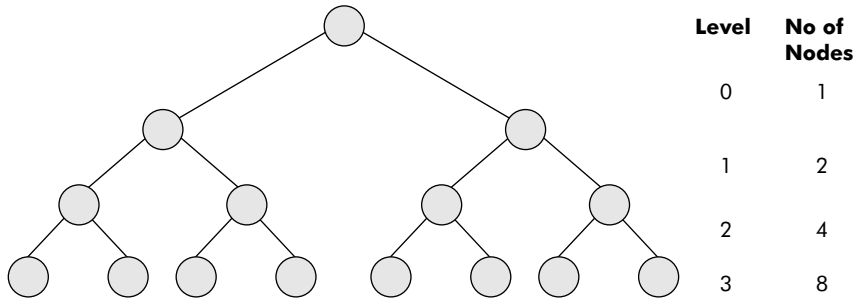


Fig. 7.10 A full binary tree

- (1) The maximum number of nodes at level 0 = 1
 The maximum number of nodes at level 1 = 2
 The maximum number of nodes at level 2 = 4
 The maximum number of nodes at level 3 = 8
 By induction, it can be deduced that **maximum nodes in a binary tree at level $l = 2^l$**
- (2) The maximum number of nodes possible in a binary tree of depth 0 = 1
 The maximum number of nodes possible in a binary tree of depth 1 = 3
 The maximum number of nodes possible in a binary tree of depth 2 = 7
 The maximum number of nodes possible in a binary tree of depth 3 = 15
 By induction, it can be deduced that **maximum nodes in a binary tree at depth $k = 2^k - 1$**
- (3) It may be noted that the **number of leaf nodes is 1 more than the number of non-leaf nodes** in a non-empty binary tree. For example, a binary tree of depth 2 has 4 leaf nodes and 3 non-leaf nodes. Similarly, for a binary tree of depth 3, number of leaf nodes is 8 and non-leaf nodes is 7.
- (4) Similarly, it can be observed that the **number of nodes is 1 more than the number of edges in a non-empty binary tree**, i.e., $n = e + 1$, where n is number of nodes and e is number of edges.
- (5) A skewed tree contains minimum number of nodes in a binary tree. Consider the skewed tree given in Figure 7.7 (d). The depth of this tree is 3 and the number of nodes in this tree is equal to 4, i.e., 1 more than the depth. Therefore, it can be observed that the number of nodes in a skewed binary tree of a depth k is equal to $k + 1$. In other words, it can be said that **minimum number of nodes possible in a binary tree of depth k is equal to $k + 1$** .

7.4 REPRESENTATION OF A BINARY TREE

The binary tree can be represented in two ways—linear representation and linked representation. A detailed discussion on both the representations is given in subsequent sections.

7.4.1 Linear Representation of a Binary Tree

A binary tree can be represented in a one-dimensional array. Since an array is a linear data structure, the representation is accordingly called linear representation of a binary tree. In fact, a binary tree of height h can be stored in an array of size $2^h - 1$. The representation is quite simple and follows the following rules:

- (1) Store a node at i th location of the array.
- (2) Store its left child at $(2*i)$ th location.
- (3) Store its right child at $(2*i + 1)$ th location.
- (4) The root of the node is always stored at first location of the array.

Consider the tree given in Figure 7.8. Its linear representation using an array called `linTree` is given in Figure 7.11.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
linTree =	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O

Fig. 7.11 Linear representation using `linTree` (of the tree of Figure 7.8)

In an array of size N , for the given node i , the following relationships are valid in linear representation:

- (1) `leftChild(i) = 2*i` { When $2*i > N$ then there is no left child }
- (2) `rightChild(i) = 2*i + 1` { When $2*i + 1 > N$ then there is no right child }
- (3) `parent(i) = [i/2]` { node 0 has no parent – it is a root node }

The following examples can be verified from the tree shown in Figure 7.8:

- The parent of node at $i = 5$ (i.e., E) would be at $[5/2] = 2$, i.e., B.
- The left child of node at $i = 4$ (i.e., D) would be at $(2*4) = 8$, i.e., H.
- The right child of node $i = 10$ (i.e., J) would be at $(2*10) + 1 = 21$ which is > 15 , indicating that it has no right child.

It may be noted that this representation is memory efficient only when the binary tree is full whereas in case of skewed trees, this representation is very poor in terms of wastage of memory locations. For instance, consider the skewed tree given in Figure 7.7 (d). Its linear representation using an array called `skewTree` is given in Figure 7.12.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
skewTree =	A	B		C				D							

Fig. 7.12 Linear representation of the skewed tree of Figure 7.7(d) using `skewTree`

It may be noted that out of 15 locations, only 4 locations are being used and 11 are unused resulting in almost 300 per cent wastage of storage space.

The disadvantages of linear representation of binary tree are as follows:

- (1) In a skewed or scanty tree, a lot of memory space is wasted.
- (2) When binary tree is full then space utilization is complete but insertion and deletion of nodes becomes a cumbersome process.
- (3) In case of a full binary tree, insertion of a new node is not possible because an array is a static allocation and its size cannot be increased at run time.

7.4.2 Linked Representation of a Binary Tree

A critical look at the organization of a non-empty binary tree suggests that it is a collection of nodes. Each node comprises the following:

- It contains some information.
- It has an edge to a left child node.
- It has an edge to a right child node.

The above components of a node can be comfortably represented by a self-referential structure consisting of DATA part, a pointer to its left child (say `leftChild`), and a pointer to its right child (say `rightChild`) as shown in Figure 7.13.

Now the node shown in Figure 7.13 will become the building block for construction of a binary tree as shown in Figure 7.14.

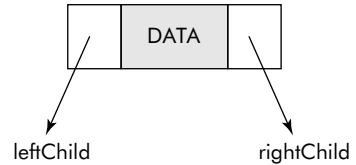


Fig. 7.13 A node of a binary tree

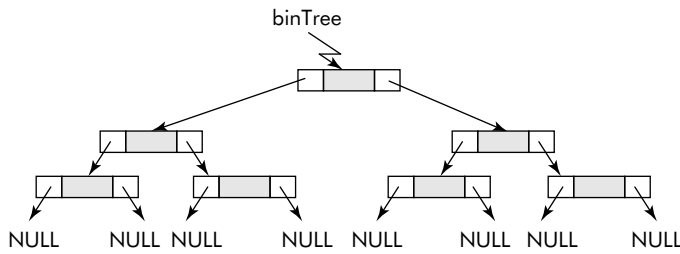


Fig. 7.14 A binary tree represented through linked structures

It may be noted that the root node is pointed by a pointer called `binTree`. Both `leftChild` and `rightChild` of leaf nodes are pointing to `NULL`. Let us now try to create binary tree.

Note: Most of the books create only binary search trees which are very specific trees and easy to build. In this book a novel algorithm is provided that creates a generalized binary tree with the help of a stack. The generalized binary tree would be constructed and implemented in 'C' so that the reader gets the benefit of learning the technique for building binary trees.

7.4.2.1 Creation of a Generalized Binary Tree

Let us try to create the tree shown in Figure 7.15.

A close look at the tree shown in Figure 7.15 gives a clue that any binary tree can be represented as a list of nodes. The list can be prepared by following steps:

- (1) Write the root.
- (2) Write its sub-trees within a set of parenthesis, separated by a comma.

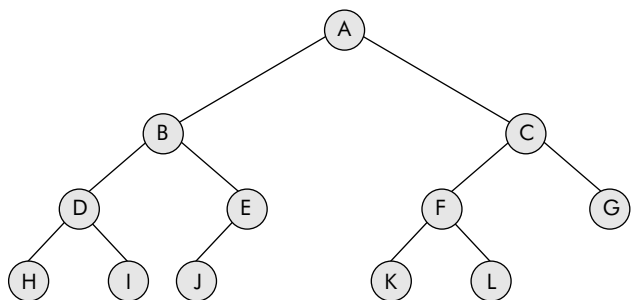


Fig. 7.15 The binary tree to be created

- (3) Repeat steps 1 and 2 for sub-trees till you reach leaf nodes. A missing child in a non-leaf node is represented as \$.

By applying the above steps on binary tree of Figure 7.15, we get the following list:

```

A ( B, C)
  ↓
A ( B (D, E), C (F,G))
  ↓
A ( B (D (H, I), E (J, $)), C (F (K,L),G))

```

So the final list is: A (B (D (H, I), E (J, \$)), C (F (K,L),G))

Now the above list can be used to create its originating binary tree with the help of a stack. It is stored in a list called `treeList` as given below:

```

treeList[] = {'A', '(', 'B', '(', 'D', '(', 'H', 'I', ')', 'E', '(', 'J', '$', ')', ')', ')', 'C', '(', 'F', '(', 'K', 'L', ')', 'G', ')', ')', '#'};

```

The algorithm that uses the `treeList` and a stack to create the required binary tree is given below:

Algorithm `createBinTree()`

```

{
  Step
  1. Take an element from treeList;
  2. If (element != '(' && element != ')') && element != '$' && element != '#' )
  {
    Take a new node in ptr;
    DATA(ptr) = element;
    leftChild (ptr) = NULL;
    rightChild (ptr) = NULL;
    Push (ptr);
  }
  3. if (element = ')')
  {
    Pop (LChild);
    Pop (RChild);
    Pop(Parent);
    leftChild(Parent) = LChild;
    rightChild(Parent) = RChild;
    Push (Parent);
  }
  4. if (element = '(' ) do nothing;
  5. if (element = '$' ) Push (NULL);
  6. repeat steps 1 to 5 till element != #;
  7. Pop (binTree);
  8. stop.
}

```

A simulation of the above algorithm for the following treeList is given in Figure 7.16.

```
treeList[] = { 'A', '(', 'B', '(', 'D', '(', 'H', 'I', ')', 'E', '(', 'J', '$', ')',
               ')', 'C', '(', 'F', '(', 'K', 'L', ')', 'G', ')', ')', '#';
```

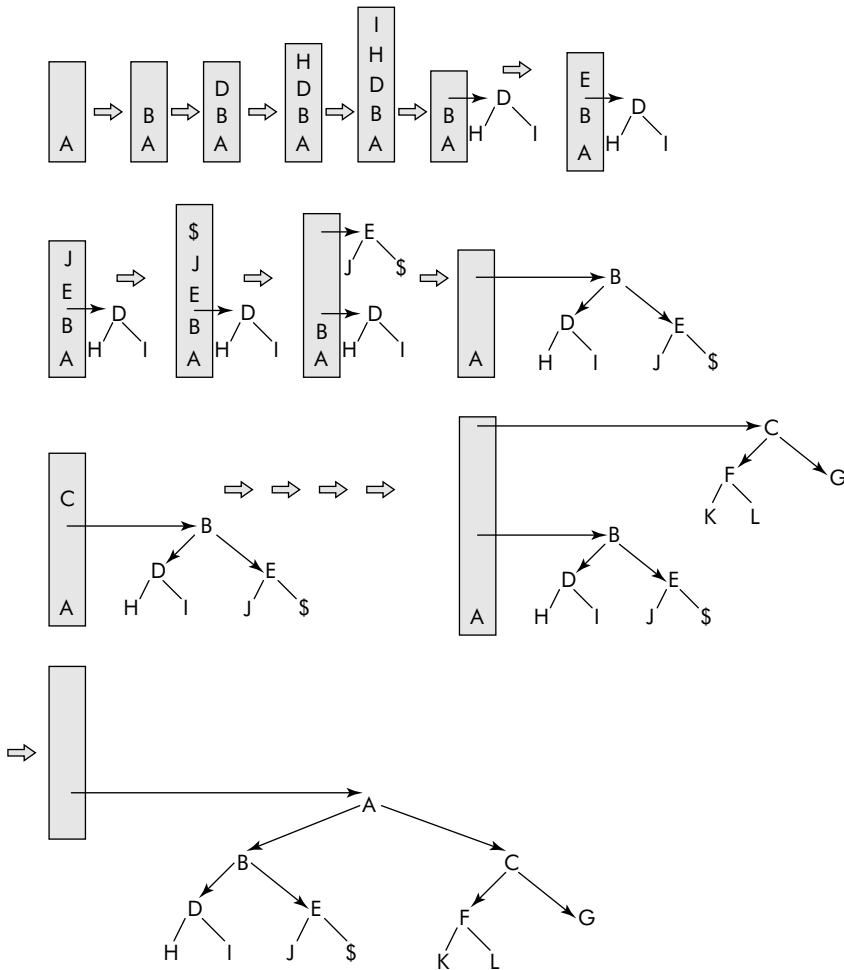


Fig. 7.16 The creation of binary tree using the algorithm createBinTree()

It may be noted that the algorithm developed by us has created exactly the same tree which is shown in Figure 7.15.

7.4.3 Traversal of Binary Trees

A binary tree is traversed for many purposes such as for searching a particular node, for processing some or all nodes of the tree and the like. A system programmer may implement a symbol table as binary search tree and process it in a search/insert fashion.

The following operations are possible on a node of a binary tree:

- (1) Process the visited node – V.
- (2) Visit its left child – L.
- (3) Visit its right child – R.

The travel of the tree can be done in any combination of the above given operations. If we restrict ourselves with the condition that left child node would be visited before right child node, then the following three combinations are possible:

- (1) L-V-R : Inorder travel
 - travel the left sub-tree
 - process the visited node
 - travel the right sub-tree
- (2) V-L-R : Preorder travel
 - process the visited node
 - travel the left sub-tree
 - travel the right sub-tree
- (3) L-R-V : Postorder travel
 - travel the left sub-tree
 - travel the right sub-tree
 - process the visited node

It may be noted that the travel strategies given above have been named as preorder, inorder, and postorder according to the placement of operation: 'V' – 'process the visited node'. Inorder (L-V-R) travel has the operation 'V' in between the other two operations. In preorder (V-L-R), the operation 'V' precedes the other two operations. Similarly, the postorder traversal has the operation 'V' after the other two operations.

7.4.3.1 Inorder Travel (L-V-R) This order of travel, i.e., L-V-R requires that while travelling the binary tree, the left sub-tree of a node be travelled first before the data of the visited node is processed. Thereafter, its right sub-tree is travelled.

Consider the binary tree given in Figure 7.17. Its root node is pointed by a pointer called Tree.

The travel starts from root node A and its data is not processed as it has a left child which needs to be travelled first indicated by the operation L. Therefore, we move to B. As B has no left child, its data is processed indicated by the operation V. Now the right child of B needs to be travelled which is NULL, indicated by the operation R. Thus, the travel of B is complete and the travel of left sub-tree of A is also over. As per the strategy, the data of A is processed (V) and the travel moves to its right sub-tree, i.e., to node C (R). The process is repeated for this sub-tree with root node as C. The final trace of the travel is given below:

B A F D G C E

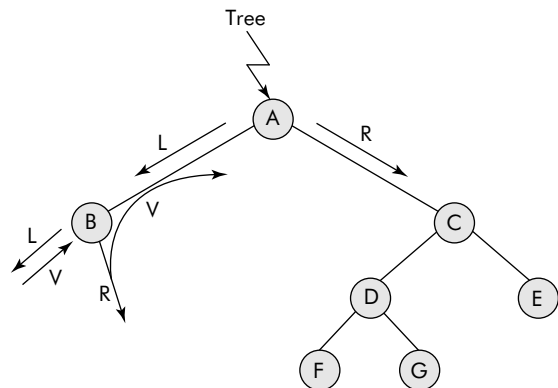


Fig. 7.17 Partial travel of L-V-R order

An algorithm for inorder travel (L-V-R) is given below. It is provided with the pointer called Tree that points to the root of the binary tree.

```
Algorithm inorderTravel(Tree)
{
    if (Tree == NULL) return
    else
    {
        inorderTravel (leftChild (Tree));
        process DATA (Tree);
        inorderTravel (rightChild (Tree));
    }
}
```

Example1: Write a program that creates a generalized binary tree and travels it using an inorder strategy. While travelling, it should print the data of the visited node.

Solution: We would use the algorithms createBinTree() and inorderTravel(Tree) to write the program.

The required program is given below:



```
/* This program creates a binary tree and travels it using inorder fashion */

#include <stdio.h>
#include <alloc.h>
#include <conio.h>

struct tnode
{
    char ch;
    struct tnode *leftChild, *rightChild;
};

struct node
{
    struct tnode *item;
    struct node *next;
};

struct node *push (struct node *Top, struct node *ptr);
struct node *pop (struct node *Top, struct node **ptr);
void dispStack(struct node *Top);
void treeTravel (struct tnode *binTree);
void main ()
{ int i;

    /* char treeList[] = { '1', '(', '2', '(', '$', '4',
        '),' , '3', '(', '5', '6', '),' , '),' , '#' }; */
    char treeList[] = { 'A', '(', 'B', '(', 'D', '(', 'H', 'I', '),' , 'E', '(',
        'J', '$', '),' , '),' , 'C', '(', 'F', '(', 'K', 'L', '),' , 'G', '),' , '),' , '#' };
}
```

```

struct node * Top, *ptr, *LChild, *RChild, *Parent;
struct tnode *binTree;
int size, tsize;
size = sizeof (struct node);
Top =NULL;
i=0;
while (treeList[i] != '#')
{
    switch (treeList[i])
    {
        case ')':Top = pop (Top, &RChild);
            Top = pop (Top, &LChild);
            Top = pop (Top, &Parent);
            Parent->item->leftChild = LChild->item;
            Parent->item->rightChild =RChild->item;
            Top = push (Top, Parent);
            break;
        case '(':break;
        default:
            ptr = (struct node *) malloc (size);
            if (treeList[i] == '$')
                ptr->item = NULL;
            else
            {
                ptr->item->ch = treeList[i];
                ptr->item->leftChild =NULL;
                ptr->item->rightChild =NULL;
                ptr->next = NULL;
            }
            Top = push (Top, ptr);
            break;
        }
        i++;
    }
    Top = pop (Top, &ptr);
    binTree = ptr->item;
    printf("\n The Tree is..");
    treeTravel(binTree);
}

struct node *push (struct node *Top, struct node *ptr)
{
    ptr->next = Top;
    Top = ptr;
    return Top;
}

```

```

struct node *pop (struct node *Top, struct node **ptr)
{
    if (Top == NULL) return NULL;
    *ptr = Top;
    Top = Top->next;
    return Top;
}

void dispStack(struct node *Top)
{
    if (Top == NULL)
        printf ("\n Stack Empty");
    else
    {
        printf ("\n The Stack is: ");
        while (Top != NULL)
        {
            printf (" %c ", Top->item->ch);
            Top =Top->next;
        }
    }
}

void treeTravel (struct tnode *binTree)
{
    if (binTree == NULL) return;
    treeTravel (binTree->leftChild);
    printf ("%c ", binTree->ch);
    treeTravel (binTree->rightChild);
}

```

It may be noted that a linked stack has been used for the implementation. The program has been tested for the binary tree given in Figure 7.15 represented using the following tree list:

```

treeList[] = {'A', '(', 'B', '(', 'D', '(', 'H', 'I', ')', 'E', '(', 'J', '$', ' ', ' ', ' ', ' ', 'C', '(', 'F', '(', 'K', 'L', ')', 'G', ')', ')', ')', '#'};

```

The output of the program is given below:

H D I B J E A K F L C G

The above output has been tested to be true.

The program was also tested on the following binary tree (shown as a comment in this program):

```

char treeList[] = {'1', '(', '2', '(', '$', '4', ')', '3', '(', '5', '6', ')', ')', '#'}

```

The output of the program is given below:

2 4 1 5 3 6

It is left as an exercise for the students to generate the binary tree from the tree list and then verify the output.

7.4.3.2 Preorder Travel (V-L-R) This order of travel, i.e., V-L-R requires that while travelling the binary tree, the data of the visited node be processed first before the left sub-tree of a node is travelled. Thereafter, its right sub-tree is travelled.

Consider the binary tree given in Figure 7.18. Its root node is pointed by a pointer called Tree.

The travel starts from root node A and its data is processed (V). As it has a left child, we move to B (L) and its data is processed (V). As B has no left child, the right child of B needs to be travelled which is also NULL. Thus, the travel of B is complete and the processing of node A and travel of left sub-tree of A is also over. As per the strategy, the travel moves to its right sub-tree, i.e., to node C (R). The process is repeated for this sub-tree with root node as C. The final trace of the travel is given below:

A B C D F G E

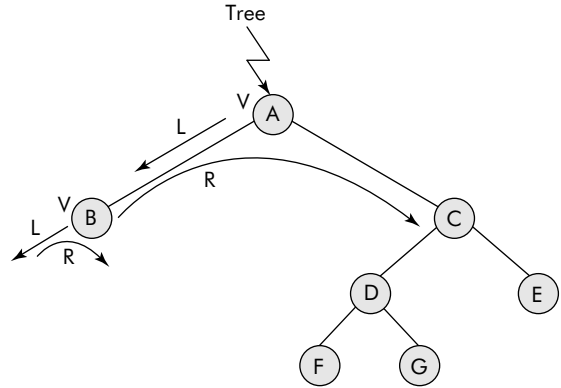


Fig. 7.18 Partial travel of V-L-R order

An algorithm for preorder travel (V-L-R) is given below. It is provided with a pointer called Tree that points to the root of the binary tree.

```
Algorithm preorderTravel (Tree)
{
    if (Tree == NULL) return
    else
    { process DATA (Tree);
      preorderTravel (leftChild (Tree));
      preorderTravel (rightChild (Tree));
    }
}
```

Example2: Write a program that creates a generalized binary tree and travels it using preorder strategy. While travelling, it should print the data of the visited node.

Solution: We would use the algorithms createBinTree() and preorderTravel(Tree) to write the program. However, the code will be similar to Example 1. Therefore, only the function preorderTravel() is provided.

The required program is given below:

```
void preOrderTravel (struct tnode *binTree)
{
    if (binTree == NULL) return;
    printf ("%c ", binTree->ch);
    preOrderTravel (binTree->leftChild);
    preOrderTravel (binTree->rightChild);
}
```

The function has been tested for the binary tree given in Figure 7.15 represented using the following tree list:

```
treeList[] = { 'A', '(', 'B', '(', 'D', '(', 'H', 'I', ')', 'E', '(', 'J', '$',
               ')', ')', 'C', '(', 'F', '(', 'K', 'L', ')', 'G', ')', ')', '#';
```

The output of the program is given below:

A B D H I E J C F K L G

The above output has been tested to be true.

The program was also tested on the following binary tree:

```
char treeList[] = { '1', '(', '2', '(', '$', '4', ')', '3', '(', '5', '6', ')', ')', '#';
```

The output of the program is given below:

1 2 4 3 5 6

It is left as an exercise for the students to generate the binary tree from the `treeList` and then verify the output.

7.4.3.3 Postorder Travel (L-R-V) This order of travel, i.e., L-R-V requires that while travelling the binary tree, the data of the visited node be processed only after both the left and right sub-trees of a node have been travelled.

Consider the binary tree given in Figure 7.19. Its root node is pointed by a pointer called `Tree`.

The travel starts from root node A and its data is not processed as it has a left child which needs to be travelled first indicated by the operation L. Therefore, we move to B. As B has no left child, the right child of B needs to be travelled which is NULL indicated by the operation R. Now, data of node B is processed indicated by the operation V. Thus, the travel of B is complete and the travel of left sub-tree of A is also over. As per the strategy, the travel moves to its right sub-tree, i.e., to node C (R). The process is repeated for this sub-tree with root node as C. The final trace of the travel is given below:

B F G D E C A

An algorithm for postorder travel (L-R-V) is given below. It is provided with the pointer called `Tree` that points to the root of the binary tree.

```
Algorithm postOrderTravel(Tree)
{
    if (Tree == NULL) return
    else
    { postOrderTravel (leftChild (Tree));
```

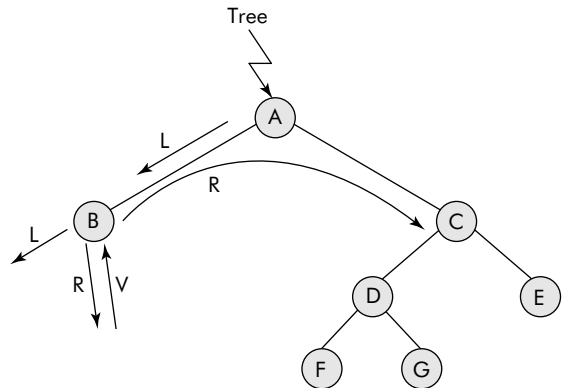


Fig. 7.19 Partial travel of L-R-V order

```

    postOrderTravel (rightChild (Tree));
    process DATA (Tree);
}
}

```

Example3: Write a program that creates a generalized binary tree and travels it using preorder strategy. While travelling, it should print the data of the visited node.

Solution: We would use the algorithms `createBinTree()` and `postOrderTravel(Tree)` to write the program. However, the code will be similar to Example 1. Therefore, only the function `postOrderTravel()` is provided.

The required function is given below:

```

void postOrderTravel (struct tnode *binTree)
{
    if (binTree == NULL) return;
    postOrderTravel (binTree->leftChild);
    postOrderTravel (binTree->rightChild);
    printf ("%c ", binTree->ch);
}

```

The function has been tested for the binary tree given in Figure 7.15 represented using the following tree list:

```

treeList[] = {'A', '(', 'B', '(', 'D', '(', 'H', 'I', ')', ')', 'E', '(', 'J', '$',
              ')', ')', ')', 'C', '(', 'F', '(', 'K', 'L', ')', ')', 'G', ')', ')', '#'};

```

The output of the program is given below:

H I D J E B K L F G C A

The above output has been tested to be true.

The program was also tested on the following binary tree:

```

char treeList[] = {'1', '(', '2', '(', '$', '4', ')', '3', '(', '5', '6', ')', ')', '#'}

```

The output of the program is given below:

4 2 5 6 3 1

It is left as an exercise for the students to generate the binary tree from the treeList and then verify the output.

7.5 TYPES OF BINARY TREES

There are many types of binary trees. Some of the popular binary trees are discussed in subsequent sections.

7.5.1 Expression Tree

An expression tree is basically a binary tree. It is used to express an arithmetic expression. The reason is simple because both binary tree and arithmetic expression are binary in nature. In binary tree, each node has two children, i.e., left and right child nodes whereas in an arithmetic expression, each arithmetic

operator has two operands. In a binary tree, a single node is also considered as a binary tree. Similarly, a variable or constant is also the smallest expression.

Consider the preorder arithmetic expression given below:

$$* + A * B C - D E$$

The above arithmetic expression can be expressed as an expression tree given in Figure 7.20.

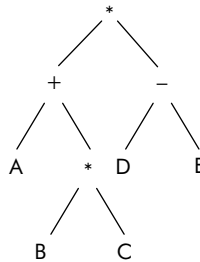


Fig. 7.20 An expression tree of $(* + A * B C - D E)$

The utility of an expression tree is that the various tree travel algorithms can be used to convert one form of arithmetic expression to another i.e. prefix to infix or infix to post fix or vice-versa.

For example, the inorder travel of the tree given in Figure 7.20 shall produce the following infix expression.

$$A + B * C * D - E$$

The postorder travel of the tree given in Figure 7.20 shall produce the following postfix expression:

$$A B C * + D E - *$$

Similarly, the preorder travel of the tree given in Figure 7.20 shall produce the following prefix expression:

$$* + A * B C - D E$$

From the above discussion, we can appreciate the rationale behind the naming of the travel strategies VLR, LVR, and LRV as preorder, inorder, and postorder, respectively.

Example 4: Write a program that travels the expression tree of Figure 7.21 in an order chosen by the user from a menu. The data of visited nodes are printed. Design an appropriate menu for this program.

Solution: The expression tree is being expressed as a `treeList` given below:

```
treeList = - (+ (P, / (Q, A)), B) #
```

The above list would be used to write the required menu driven program. The various binary tree travel functions,

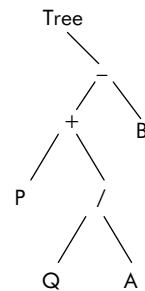


Fig. 7.21 An expression tree

developed above, would be used to travel the expression tree as per the user choice entered on the following menu displayed to the user:

Menu	
Inorder	1
Preorder	2
Postorder	3
Quit	4

Enter your choice:

The required program is given below:

```

/* This program travels an expression tree */
#include <stdio.h>
#include <alloc.h>
#include <conio.h>

struct tnode
{
    char ch;
    struct tnode *leftChild, *rightChild;
};

struct node
{
    struct tnode *item;
    struct node * next;
};

struct node *push (struct node *Top, struct node *ptr);
struct node *pop (struct node *Top, struct node **ptr);
void dispStack(struct node *Top);
void preOrderTravel (struct tnode *binTree);
void inOrderTravel (struct tnode *binTree);
void postOrderTravel (struct tnode *binTree);
void main ()
{
    int i, choice;
    char treeList[] = {'-', '(', '+', '(', 'P', '/', '(', 'Q', 'A', ')', ')', ')',
                      'B', ')', ')', '#'};

    struct node * Top, *ptr, *LChild, *RChild, *Parent;
    struct tnode *binTree;
    int size, tsize;
    size = sizeof (struct node);
    Top =NULL;
    i=0;
    while (treeList[i] != '#')
    {

```

```
switch (treeList[i])
{
case ')': Top = pop (Top, &RChild);
    Top = pop (Top, &LChild);
    Top = pop (Top, &Parent);
    Parent->item->leftChild = LChild->item;
    Parent->item->rightChild = RChild->item;
    Top = push (Top, Parent);
    break;
case '(': break;
default:
    ptr = (struct node *) malloc (size);
    if (treeList[i] == '$')
        ptr->item = NULL;
    else
    {
        ptr->item->ch = treeList[i];
        ptr->item->leftChild = NULL;
        ptr->item->rightChild = NULL;
        ptr->next = NULL;
    }
    Top = push (Top, ptr);
    break;
}
i++;
}
Top = pop (Top, &ptr);
/* display Menu */
do {
    clrscr();
    printf ("\n Menu      Tree Travel");
    printf ("\n\n Inorder      1");
    printf ("\n\n Preorder      2");
    printf ("\n\n Postorder     3");
    printf ("\n\n Quit        4");
    printf ("\n\n Enter your choice:");
    scanf ("%d", &choice);
    binTree = ptr->item;
    switch (choice)
    {
case 1: printf ("\n The Tree is..");
        inOrderTravel(binTree);
        break;
case 2: printf ("\n The Tree is..");
        preOrderTravel(binTree);
        break;
```

```

        case 3: printf("\n The Tree is..");
                postOrderTravel(binTree);
                break;
        };
        printf("\n Enter any key to continue");
        getch();
    }
    while (choice != 4);
}

struct node * push (struct node *Top, struct node *ptr)
{
    ptr->next = Top;
    Top = ptr;
    return Top;
}

struct node * pop (struct node *Top, struct node **ptr)
{
    if (Top == NULL) return NULL;
    *ptr = Top;
    Top = Top->next;
    return Top;
}

void dispStack(struct node *Top)
{
    if (Top == NULL)
        printf ("\n Stack Empty");
    else
    {
        printf ("\n The Stack is: ");
        while (Top != NULL)
        {
            printf (" %c ", Top->item->ch);
            Top =Top->next;
        }
    }
}

void preOrderTravel (struct tnode *binTree)
{
    if (binTree == NULL) return;
    printf ("%c ", binTree->ch);
    preOrderTravel (binTree->leftChild);
    preOrderTravel (binTree->rightChild);
}

```

```

void inOrderTravel (struct tnode *binTree)
{
    if (binTree == NULL) return;
    inOrderTravel (binTree->leftChild);
    printf ("%c ", binTree->ch);
    inOrderTravel (binTree->rightChild);
}

void postOrderTravel (struct tnode *binTree)
{
    if (binTree == NULL) return;
    postOrderTravel (binTree->leftChild);
    postOrderTravel (binTree->rightChild);
    printf ("%c ", binTree->ch);
}

```

The output of the program for choice 1 (inorder travel) is given below:

The Tree is ...: P + Q / A - B

The output of the program for choice 2 (preorder travel) is given below:

The Tree is ...: - + P / Q A - B

The output of the program for choice 3 (postorder travel) is given below:

The Tree is ...: P Q A / + B -

It is left as an exercise for the readers to verify the results by manually converting the expression to the required form.

7.5.2 Binary Search Tree

As the name suggests, a binary search tree (BST) is a special binary tree that is used for efficient searching of data. In fact a BST stores data in such a way that binary search algorithm can be applied.

The BST is organized as per the following extra conditions on a binary tree:

- The data to be stored must have unique key values.
- The key values of the nodes of left sub-tree are always less than the key value of the root node.
- The key values of the nodes of right sub-tree are always more than the key value of the root node.
- The left and right sub-trees are themselves BSTs.

In simple words we can say that the key contained in the left child is less than the key of its parent node and the key contained in the right child is more than the key of its parent node as shown in Figure 7.22.

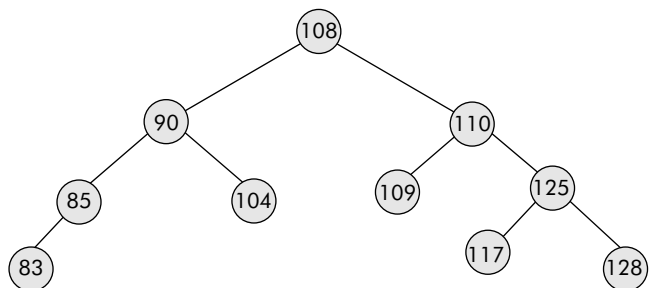


Fig. 7.22 A binary search tree (BST)

Kindly note that the inorder travel of the binary search tree would produce the following node sequence:

Tree is ... 83 85 90 104 108 109 110 117 125 128

The above list is sorted in ascending order indicating that a **binary search tree is another way to represent and produce a sorted list**. Therefore, a binary search algorithm can be easily applied to a BST and, hence, the name, i.e., binary **search** tree.

7.5.2.1 Creation of a Binary Search Tree A binary search tree is created by successively reading values for the nodes. The first node becomes the root node. The next node's value is compared with the value of the root node. If it is less, the node is attached to root's left sub-tree otherwise to right sub-tree, and so on. When the sub-tree is NULL, then the node is attached there itself. An algorithm for creation of binary search tree is given below.

Note: This algorithm gets a pointer called binTree. The creation of BST stops when an invalid data is inputted. It uses algorithm attach() to attach a node at an appropriate place in BST.

```
binTree = NULL;    /* initially the tree is empty */
Algorithm createBST(binTree)
{
    while (1)
    {
        take a node in ptr
        read DATA (ptr);
        if (DATA (ptr) is not valid) break;
        leftChild (ptr) = NULL;
        rightChild (ptr) = NULL;
        attach (binTree, ptr);
    }
}
return binTree;
Algorithm attach (Tree, node)
{
    if (Tree == NULL)
        Tree = node;
    else
    {
        if (DATA (node) < DATA(Tree))
            attach (Tree->leftChild);
        else
            attach (Tree-> rightChild);
    }
    return Tree ;
}
```

Example 5: Write a program that creates a BST wherein each node of the tree contains non-negative integer. The process of creation stops as soon as a negative integer is inputted by the user. Travel the tree using inorder travel to verify the created BST.

Solution: The algorithm `createBST()` and its associated algorithm `attach()` would be used to write the program. The required program is given below:

```
/* This program creates a BST and verifies it by travelling it using
inorder travel */

#include <stdio.h>
#include <alloc.h>
#include <conio.h>

struct binNode
{
    int val;
    struct binNode *leftChild, *rightChild;
};

struct binNode *createBinTree();
struct binNode *attach(struct binNode *tree, struct binNode *node);
void inOrderTravel (struct binNode *binTree);
void main()
{
    struct binNode *binTree;
    binTree = createBinTree();
    printf ("\n The tree is...");
    inOrderTravel(binTree);
}

struct binNode *createBinTree()
{int val, size;
struct binNode *treePtr, *newNode;
treePtr = NULL;
printf ("\n Enter the values of the nodes terminated by a negative value");
val = 0;
size = sizeof (struct binNode);
while (val >=0)
{
    printf ("\n Enter val");
    scanf ("%d", &val);
    if (val >=0)
    {
        newNode = (struct binNode *) malloc (size);
        newNode->val =val;
        newNode->leftChild =NULL;
        newNode->rightChild=NULL;
        treePtr = attach (treePtr, newNode);
    }
}
return treePtr;
}
```



```

struct binNode *attach(struct binNode *tree, struct binNode *node)
{
    if (tree == NULL )
        tree = node;
    else
    {
        if (node->val < tree-> val)
            tree->leftChild = attach (tree->leftChild, node);
        else
            tree->rightChild= attach (tree->rightChild, node);
    }
    return tree;
}

void inOrderTravel (struct binNode *binTree)
{
    if (binTree == NULL) return;
    inOrderTravel (binTree->leftChild);
    printf ("%d-", binTree->val);
    inOrderTravel (binTree->rightChild);
}

```

The above program was given the following input:

56 76 43 11 90 65 22

And the following output was obtained when this BST was travelled using inorder travel:

11 22 43 56 65 76 90

It may be noted that the output is a sorted list verifying that the created BST is correct. Similarly, the program was given the following input:

45 12 34 67 78 56

and the following output was obtained:

12 34 45 56 67 78

The output is a sorted list and it verifies that the created BST is correct.

Note: The drawback of a binary search tree is that if the data supplied to the creation process are in a sequence or order then the BST would be a skewed tree. The reason being that every succeeding number being greater than its preceding number would become a right child and would result in a skewed right binary tree. For example, the following input would produce a skewed tree shown in Figure 7.23.

25 35 43 67 89 120

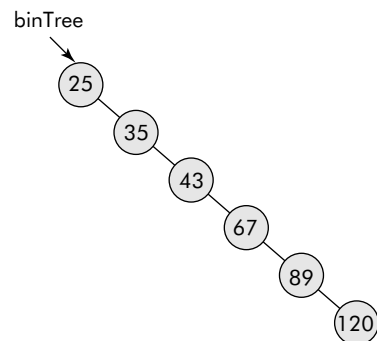


Fig. 7.23 The skewed binary search tree

7.5.2.2 Searching in a Binary Search Tree The most desirable and important application of a BST is that it is used for searching an entry in a set of values. This method is faster as compared to arrays and linked lists. In an array, the list of items needs to be sorted before a binary search algorithm could be applied for searching an item into the list. On the contrary, the data automatically becomes sorted during the creation of a BST. The item to be searched is compared with the root. If it matches with the data of the root, then success is returned otherwise left or right sub-tree is searched depending upon the item being less or more than the root.

An algorithm for searching a BST is given below; it searches Val in a BST pointed by a pointer called 'Tree'.

```
Algorithm searchBST(Tree, Val)
{
    if (Tree = NULL)
        return failure;
    if (DATA (Tree) == val)
        return success;
    else
        if (Val < DATA (Tree))
            search (Tree->leftChild, Val);
        else
            search (Tree->rightChild, Val);
}
```

Example 6: Write a program that searches a given key in a binary search tree through a function called searchBST(). The function returns 1 or 0 depending upon the search being successful or a failure. The program prompts appropriate messages like "Search successful" or "Search unsuccessful".

Solution: The above algorithm searchBST() is used. The required program is given below:

```
#include <stdio.h>
#include <alloc.h>
#include <conio.h>

struct binNode
{
    int val;
    struct binNode *leftChild, *rightChild;
};

struct binNode *createBinTree();
struct binNode *attach(struct binNode *tree, struct binNode *node);
int searchBST (struct binNode *binTree, int val);

void main()
{ int result, val;
  struct binNode *binTree;
  binTree = createBinTree ();
  printf ("\n Enter the value to be searched");
  scanf ("%d", &val);
  result = searchBST(binTree, val);
```

```

    if (result == 1)
    printf ("\n Search Successful");
    else
    printf ("\n Search Un-Successful");
}

struct binNode  *createBinTree()
{
} /* this function createBinTree() is same as developed in above examples */

int searchBST (struct binNode *binTree, int val)
{ int flag;
  if (binTree == NULL)
  return 0;
  if (val == binTree->val)
  return 1;
  else
  if (val < binTree-> val)
  flag=searchBST(binTree->leftChild, val);
  else
  flag=searchBST(binTree->rightChild, val);
  return flag;
}

```

The above program has been tested.

7.5.2.3 Insertion into a Binary Search Tree The insertion in a BST is basically a search operation. The item to be inserted is searched within the BST. If it is found, then the insertion operation fails otherwise when a NULL pointer is encountered the item is attached there itself. Consider the BST given in Figure 7.24.

It may be noted that the number 49 is inserted into the BST shown in Figure 7.24. It is searched in the BST which comes to a dead end at the node containing 48. Since the right child of 48 is NULL, the number 49 is attached there itself.

Note: In fact, a system programmer maintains a symbol table as a BST and the symbols are inserted into the BST in search-insert fashion. The main advantage of this approach is that duplicate entries into the table are automatically caught. The search engine also stores the information about downloaded web pages in search-insert fashion so that possible duplicate documents from mirrored sites could be caught.

An algorithm for insertion of an item into a BST is given below:

```

Algorithm insertBST (binTree, item)
{
    if (binTree == NULL)
    {
        Take node in ptr;
        DATA (ptr) = item;
        leftChild(ptr) = NULL;
        rightChild(ptr)= NULL;
    }
}

```

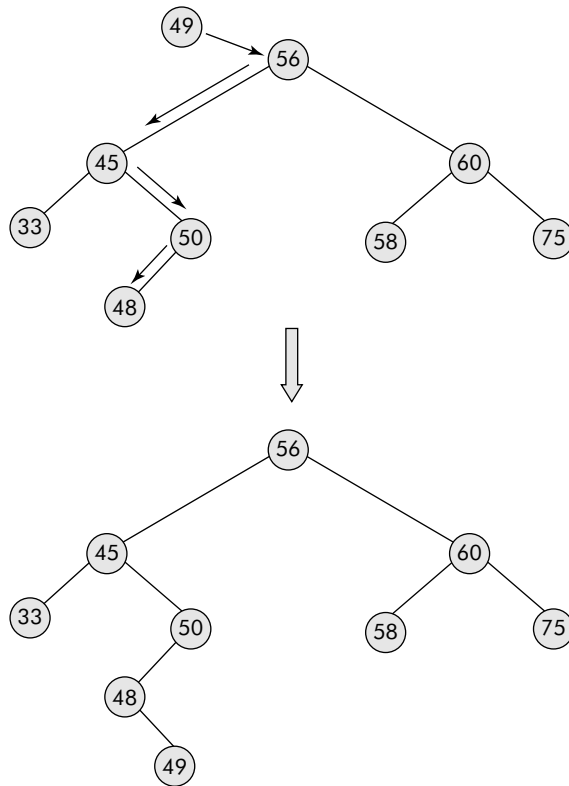


Fig. 7.24 Insertion operation in BST

```

    binTree = ptr;
    return success;
}
else
    if (DATA (binTree) == item)
        return failure;
else
    { Take node in ptr;
      DATA (ptr) = item;
      leftChild (ptr) = NULL;
      rightChild(ptr)= NULL;
      if (DATA (binTree) > item)
          insertBST (binTree->leftChild, ptr);
      else
          insertBST (binTree->rightChild, ptr);
    }
}

```

Example 7: Write a program that inserts a given value in a binary search tree through a function called `insertBST()`. The function returns 1 or 0 depending upon the insertion being success or a failure. The program prompts appropriate message like “Insertion successful” or “Duplicate value”. Verify the insertion by travelling the tree in inorder and displaying its contents containing the inserted node.

Solution: The above algorithm `insertBST()` is used. The required program is given below:

```
/* This program inserts a value in a BST */

#include <stdio.h>
#include <alloc.h>
#include <conio.h>

struct binNode
{
    int val;
    struct binNode *leftChild, *rightChild;
};

struct binNode *createBinTree();
struct binNode *attach(struct binNode *tree, struct binNode *node);
struct binNode *insertBST (struct binNode *binTree, int val, int *flag);
void inOrderTravel (struct binNode *Tree);
void main()
{ int result, val;
  struct binNode *binTree;
  binTree = createBinTree ();
  printf ("\n enter the value to be inserted");
  scanf ("%d", &val);
  binTree = insertBST(binTree, val, &result);
  if (result == 1)
  { printf ("\n Insertion Successful, The Tree is..");
    inOrderTravel (binTree);
  }
  else
    printf ("\n duplicate value");
}

struct binNode *createBinTree()
/* this function createBinTree() is same as developed in above examples */
{
}

struct binNode *attach(struct binNode *tree, struct binNode *node)
{
    if (tree == NULL )
        tree = node;
    else
    {
```

```

        if (node->val < tree-> val)
            tree->leftChild = attach (tree->leftChild, node);
        else
            tree->rightChild= attach (tree->rightChild, node);
    }
    return tree;
}

struct binNode * insertBST (struct binNode *binTree, int val, int
*flag)
{ int size;
struct binNode *ptr;
size = sizeof(struct binNode);
if (binTree == NULL)
{ ptr = (struct binNode *) malloc (size);
ptr->val = val;
ptr->leftChild = NULL;
ptr->rightChild =NULL;
binTree = ptr;
*flag=1;
return binTree;
}
if (val == binTree->val)
{
*flag= 0;
return binTree;
}
else
if (val < binTree-> val)
binTree->leftChild=insertBST(binTree->leftChild, val, flag);
else
binTree->rightChild=insertBST(binTree->rightChild, val, flag);
return binTree;
}

void inOrderTravel (struct binNode *Tree)
{
    if (Tree != NULL)
    {
        inOrderTravel (Tree->leftChild);
        printf ("%d - ", Tree->val);
        inOrderTravel (Tree->rightChild);
    }
}

```

The above program was tested for the following input:

Nodes of the BST: **45 12 67 20 11 56**

Value to be inserted: **11**

The output: “Duplicate value” was obtained indicating that the value is already present in the BST. Similarly, for the following input:

Nodes of BST: 65 78 12 34 89 77 22

Value to be inserted: 44

The output: Insertion successful, the Tree is .. 12 22 34 44 65 77 78 89, indicating that the insertion at proper place in BST has taken place.

7.5.2.4 Deletion of a Node from a Binary Search Tree Deletion of a node from a BST is not as simple as insertion of a node. The reason is that the node to be deleted can be at any of the following positions in the BST:

- (1) The node is a leaf node.
- (2) The node has only one child.
- (3) The node is an internal node, i.e., having both the children.

Case 1: This case can be very easily handled by setting the pointer to the node from its parent equal to NULL and freeing the node as shown in Figure 7.25.

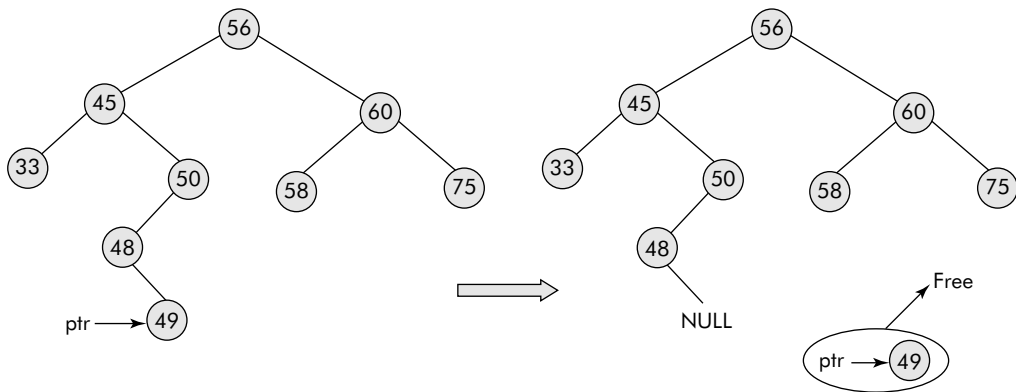


Fig. 7.25 Deletion of a leaf node

The leaf node containing 49 has been deleted. Kindly note that its parent's pointer has been set to NULL and the node itself has been freed.

Case 2: This case can also be handled very easily by setting the pointer to the node from its parent equal to its child node. The node may be freed later on as shown in Figure 7.26.

The node, containing 48, having only one child (i.e., 49) has been deleted. Kindly note that its parent's pointer (i.e., 50) has been set to its child (i.e., 49) and the node itself has been freed.

Case 3: From discussions on BST, it is evident that when a BST is travelled by inorder, the displayed contents of the visited nodes are always in increasing order, i.e., the successor is always greater than the predecessor. Another point to be noted is that inorder successor of an internal node (having both children) will never have a left child and it is always present in the right sub-tree of the internal node.

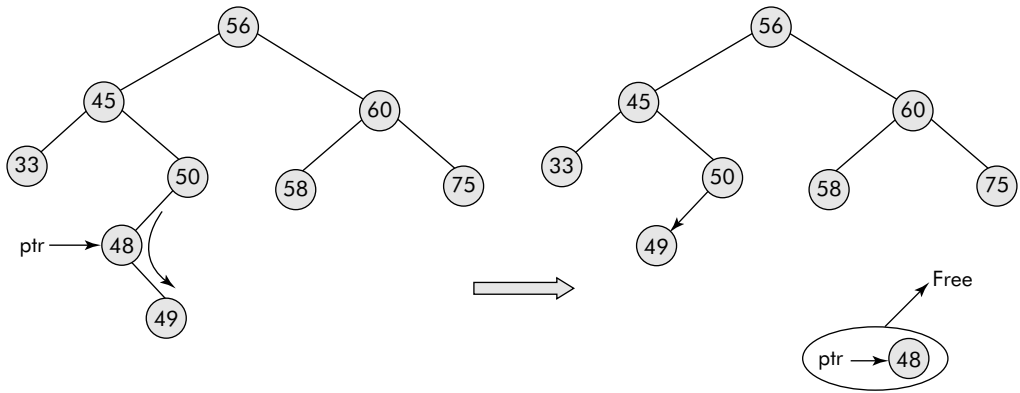


Fig. 7.26 Deletion of a node having only one child

Consider the BST given in Figure 7.26. The inorder successor of internal node 45 is 48, and that of 56 is 58. Similarly, in Figure 7.22, the inorder successors of internal nodes 90, 108, and 110 are 104, 109, and 117, respectively and each one of the successor node has no left child. It may be further noted that the successor node is always present in the right sub-tree of the internal node.

The successor of a node can be reached by first moving to its right child and then keep going to left till a NULL is found as shown in Figure 7.27.

It may be further noted that if an internal node `ptr` is to be deleted then the contents of inorder successor of `ptr` should replace the contents of `ptr` and the successor be deleted by the methods suggested for Case 1 or Case 2, discussed above. The mechanism is shown in Figure 7.28.

An algorithm to find the inorder successor of a node is given below. It uses a pointer called `succPtr` that travels from `ptr` to its successor and returns it.

```

Algorithm findSucc(ptr)
{
    succPtr = rightChild (ptr);
    while (leftChild (succPtr != NULL))
        succPtr = leftChild (succPtr);
    return succPtr;
}

```

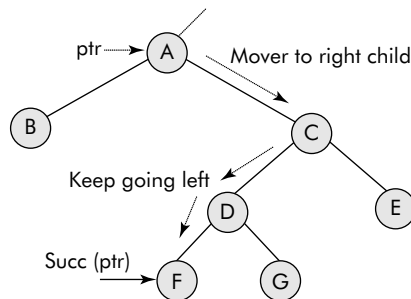


Fig. 7.27 Finding inorder successor of a node

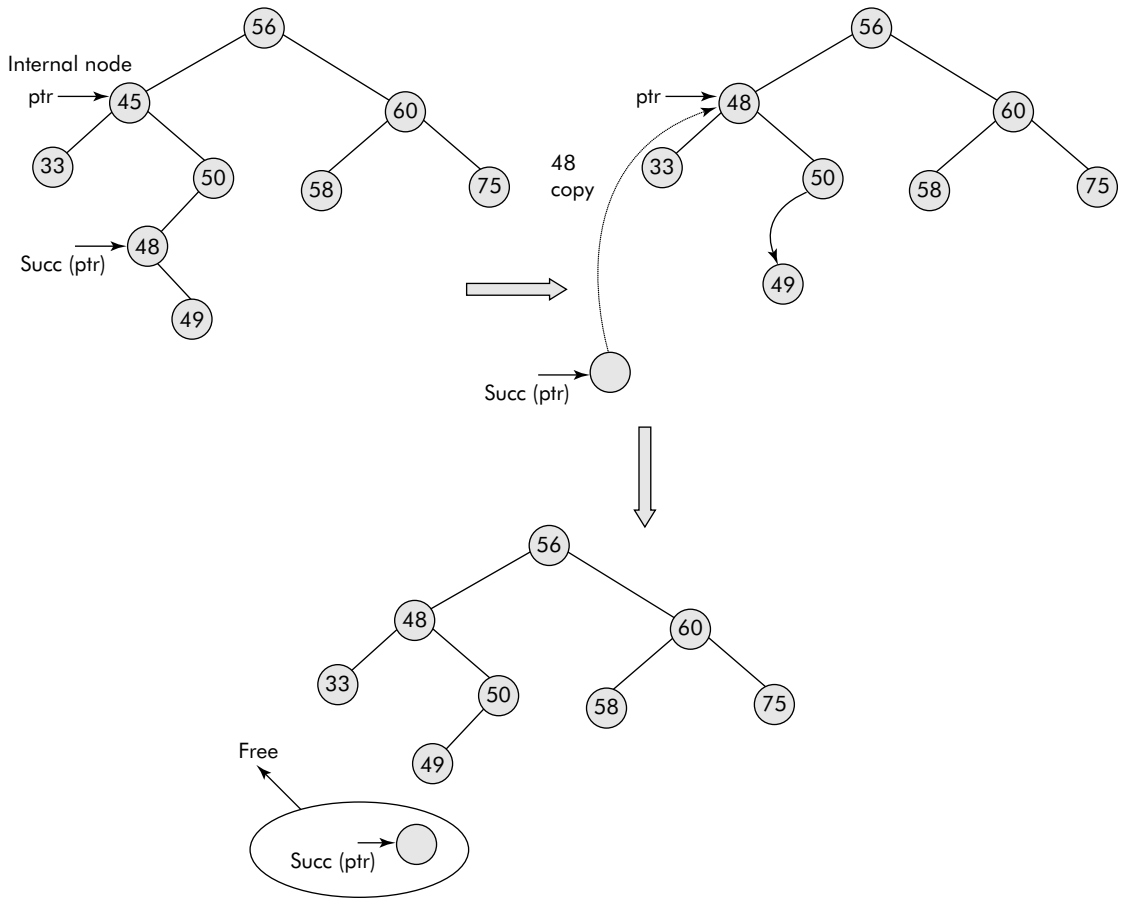


Fig. 7.28 Deletion of an internal node in BST

The algorithm that deletes a node from a BST is given below. It takes care of all the three cases given above.

```

Algorithm del NodeBST(Tree, Val)
{
    ptr = search (Tree, Val);      /* Finds the node to be deleted */
    parent = search (Tree, Val);   /* Finds the parent of the node */
    if (ptr == NULL) report error and return;

    if (leftChild (ptr) == NULL && rightChild(ptr) == NULL)    /* Case 1*/
    {
        if (leftChild (Parent) == ptr)
            leftChild (Parent ) = NULL;
        else
            if (rightChild (Parent) == ptr)

```

```

        rightChild (Parent ) = NULL;
        free (ptr);
    }
if (leftChild (ptr ) != NULL && rightChild(ptr) == NULL)    /*Case 2 */
{
    if (leftChild (Parent) == ptr)
        leftChild (Parent ) = leftChild (ptr);
    else
        if (rightChild (Parent) == ptr)
            rightChild (Parent ) = leftChild (ptr);
        free (ptr);
}
else
if (leftChild (ptr ) == NULL && rightChild(ptr) != NULL)    /*Case 2 */
{
    if (leftChild (Parent) == ptr)
        leftChild (Parent ) = rightChild (ptr);
    else
        if (rightChild (Parent) == ptr)
            rightChild (Parent ) = rightChild (ptr);
        free (ptr);
}
if (leftChild (ptr ) != NULL && rightChild(ptr) != NULL)    /*Case 3 */
{
    succPtr = findSucc (ptr);
    DATA (ptr) = DATA (succPtr);
    delNodeBST (Tree, succPtr);
}

```

Example 8: Write a program that deletes a given node from binary search tree. The program prompts appropriate message like “Deletion successful”. Verify the deletion for all the three cases by travelling the tree in inorder and displaying its contents.

Solution: The above algorithms `delNodeBST()` and `findSucc()` are used. The required program is given below:

```

/* This program deletes a node from a BST */

#include <stdio.h>
#include <alloc.h>
#include <conio.h>

struct binNode
{
    int val;
    struct binNode *leftChild, *rightChild;
};

```

```

struct binNode * createBinTree();
struct binNode *attach(struct binNode *tree, struct binNode *node);
struct binNode *searchBST (struct binNode *binTree, int val, int *flag);
struct binNode *findParent (struct binNode *binTree, struct binNode *ptr);
struct binNode *delNodeBST (struct binNode *binTree, int val, int *flag);
struct binNode *findSucc (struct binNode *ptr);
void inOrderTravel (struct binNode *Tree);
void main()
{ int result, val;
  struct binNode *binTree;
  binTree = createBinTree ();
  printf ("\n Enter the value to be deleted");
  scanf ("%d", &val);
  binTree = delNodeBST(binTree, val, &result);
  if (result == 1)
  {printf ("\n Deletion successful, The Tree is..");
   inOrderTravel (binTree);
  }
  else
  printf ("\n Node not present in Tree");
}

struct binNode * createBinTree() /* this function createBinTree() is
same as developed in above examples */
{
}

struct binNode *attach(struct binNode *tree, struct binNode *node)
{ /* this function attach() is same as developed in above examples */
}

struct binNode * delNodeBST (struct binNode *binTree, int val, int *flag)
{ int size, nval;
  struct binNode *ptr, *parent,*succPtr;
  if (binTree == NULL)
  { *flag =0;
    return binTree;
  }
  ptr = searchBST (binTree, val, flag);
  if (*flag == 1)
  parent = findParent (binTree, ptr);
  else
  return binTree;
  if (ptr->leftChild == NULL && ptr->rightChild == NULL) /* Case 1*/
  {
    if (parent->leftChild == ptr)
      parent->leftChild = NULL;

```

```
    else
    if (parent->rightChild == ptr)
        parent->rightChild = NULL;
    free (ptr);
}

if (ptr->leftChild != NULL && ptr->rightChild == NULL)
{
    if (parent->leftChild == ptr)
        parent->leftChild = ptr->leftChild;
    else
        if (parent->rightChild == ptr)
            parent->rightChild = ptr->leftChild;
    free (ptr);
    return binTree;
}
else
if (ptr->leftChild == NULL && ptr->rightChild != NULL)
{
    if (parent->leftChild == ptr)
        parent->leftChild = ptr->rightChild;
    else
        if (parent->rightChild == ptr)
            parent->rightChild = ptr->rightChild;
    free (ptr);
    return binTree;
}

if (ptr->leftChild != NULL && ptr->rightChild != NULL)
{
    succPtr = findSucc (ptr);
    nval = succPtr->val;
    delNodeBST (binTree, succPtr->val, flag);
    ptr->val = nval;
}
return binTree;
}

void inOrderTravel (struct binNode *Tree)
{
    if (Tree != NULL)
    {
        inOrderTravel (Tree->leftChild);
        printf ("%d - ", Tree->val);
        inOrderTravel (Tree->rightChild);
    }
}
```

```

struct binNode * findSucc(struct binNode *ptr)
{ struct binNode *succPtr;
  getch();
  succPtr = ptr->rightChild;
  while (succPtr->leftChild != NULL)
    succPtr = succPtr->leftChild ;
  return succPtr;
}

struct binNode *searchBST (struct binNode *binTree, int val, int *flag)
{
  if (binTree == NULL)
    { *flag = 0;
    return binTree;
  }
  else
  {
    if (binTree->val == val)
    {
      *flag = 1;
      return binTree;
    }
    else
    { if (val < binTree->val)
      return searchBST(binTree->leftChild, val, flag);
    else
      return searchBST(binTree->rightChild, val, flag);
    }
  }
}

struct binNode *findParent (struct binNode *binTree, struct binNode *ptr)
{ struct binNode *pt;
  if (binTree == NULL)
    return binTree;
  else
  {
    if (binTree->leftChild == ptr || binTree->rightChild == ptr )
      { pt = binTree; return pt; }
    else
    { if (ptr->val < binTree-> val)
      pt= findParent(binTree->leftChild, ptr);
    else
      pt= findParent(binTree->rightChild, ptr);
    }
  }
  return pt;
}

```

The above program was tested for the following input:

Case 1:

Nodes of the BST: **38 27 40 11 39 30 45 28**

Value to be deleted: 11, a leaf node

The output: “Deletion successful”-The Tree is.. **27 28 30 38 39 40 45** was obtained.

Case 2:

Similarly, for the following input:

Nodes of BST: **38 27 40 11 39 30 45 28**

Value to be deleted: 30, a node having no right child.

The output: Deletion successful, the Tree is.. **11 27 28 38 39 40 45**, indicating that the non-leaf node 30 having only one child has been deleted.

Case 3:

Similarly, for the following input:

Nodes of BST: **38 27 40 11 39 30 45 28**

Value to be deleted: 40, an internal node having both the children.

The output: Deletion successful, the Tree is.. **11 27 28 30 38 39 45**, indicating that the internal node 40 having no child has been deleted.

7.5.3 Heap Trees

A heap tree is a complete binary tree. This data structure is widely used for building priority queues. Sorting is another application in which heap trees are employed; the sorting method is known as heap sort. As the name suggests, a heap is a collection of partially ordered objects represented as a binary tree. The data at each node in heap tree is less than or equal to the data contained in its left child and the right child as shown in Figure 7.29. This property of each node having data part less than or equal to its children is called *heap-order property*.

As we go from leaf nodes to the root (see Figure 7.29), data of nodes starts decreasing and, therefore, the root contains the minimum data value of the tree. This type of heap tree is called a ‘**min heap**’.

On the contrary, if a node contains data greater than or equal to the data contained in its left and right child, then the heap is called as ‘**max heap**’. The reason being that as we move from leaf nodes to the root, the data values increase and, therefore, in a max heap the data in the root is the maximum data value in the tree as shown in Figure 7.30.

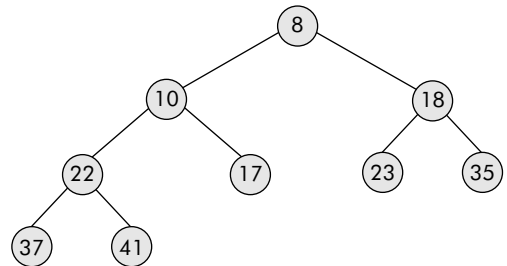


Fig. 7.29 A min-heap tree

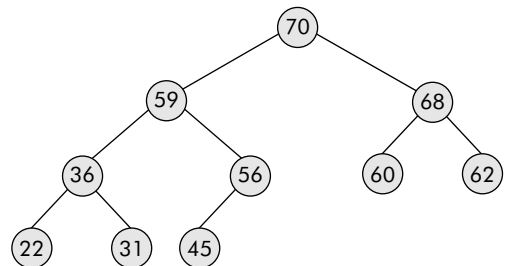


Fig. 7.30 A max-heap tree

A complete binary tree is called a heap tree (say max heap) if it has the following properties:

- (1) Each node in the heap has data value more than or equal to its left and right child nodes.
- (2) The left and right child nodes themselves are heaps.

Thus, every successor of a node has data value less than or equal to the data value of the node. The height of a heap with N nodes = $\lfloor \log_2 N \rfloor$.

7.5.3.1 Representation of a Heap Tree Since a heap tree is a complete binary tree, it can be comfortably and efficiently stored in an array. For example, the heap of Figure 7.30 can be represented in an array called 'heap' as shown in Figure 7.31. The zeroth position contains total number of elements present in the heap. For example, the zeroth location of heap contains 10 indicating that there are 10 elements in the heap shown in Figure 7.31.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
heap =	10	70	59	68	36	56	60	62	22	31	45				

Fig. 7.31 Linear representation of a heap tree

It may be noted that this representation is simpler than the linked representation as there are no links. Moreover from any node of the tree, we can move to its parent or to its children, i.e., in both forward and backward directions. As discussed in Section 7.4.1, the following relationships hold good:

In an array of size N , for the given node i , the following relationships are valid in linear representation:

- (1) $\text{leftChild}(i) = 2*i$ {When $2*i > N$ then there is no left child}
- (2) $\text{rightChild}(i) = 2*i + 1$ {When $2*i + 1 > N$ then there is no right child}
- (3) $\text{parent}(i) = \lfloor i/2 \rfloor$ {node 0 has no parent – it is a root node}

The operations that are defined on a heap tree are:

- (1) Insertion of a node into a heap tree
- (2) Deletion of a node from a heap tree

A discussion on these operations is given in the subsequent sections.

7.5.3.2 Insertion of a Node into a Heap Tree The insertion operation is an important operation on a heap tree. The node to be inserted is placed on the last available location in the heap, i.e., in the array. Its data is compared with its parent and if it is more than its parent, then the data of both the parent and the child are exchanged. Now the parent becomes a child and it is compared with its own parent and so on; the process stops only when the data part of child is found less than its parent or we reach to the root of the heap. Let us try to insert '69' in the heap given in Figure 7.31. The operation is illustrated in Figure 7.32.

An algorithm that inserts an element called item into a heap represented in an array of size N called `maxHeap` is given below. The number of elements currently present in the heap is stored in the zeroth location of `maxHeap`.

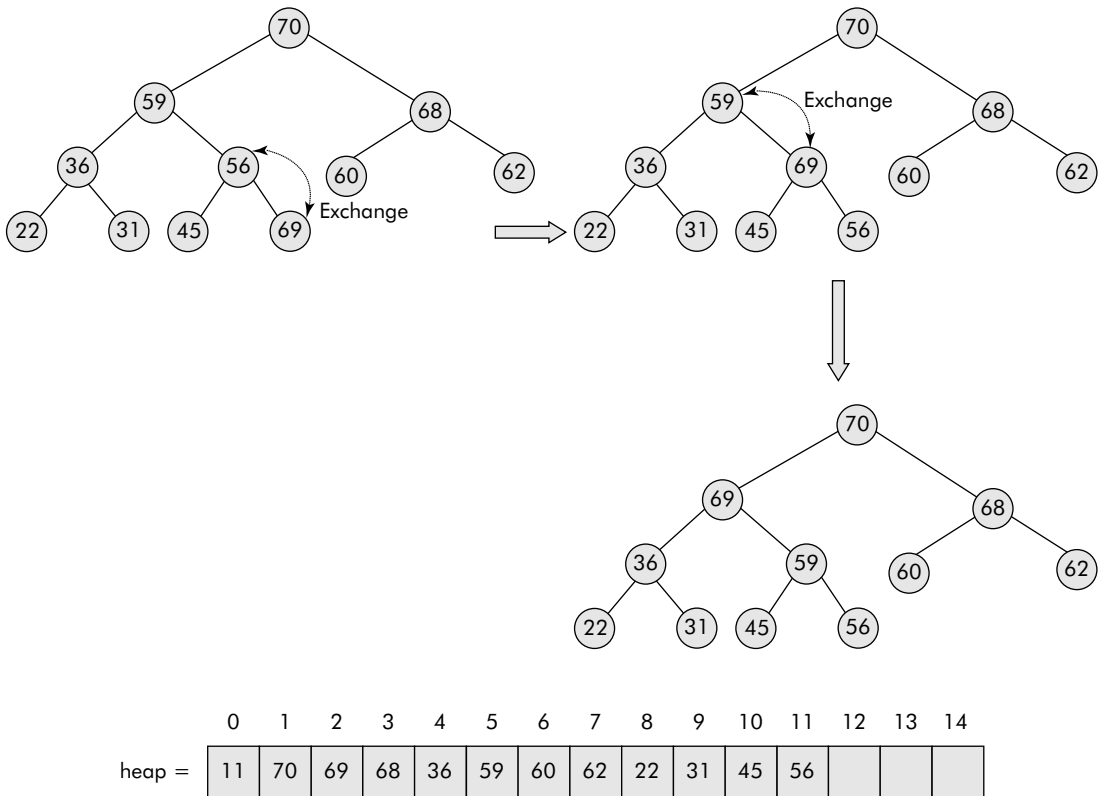


Fig. 7.32 Insertion of a node into a heap tree

```

Algorithm insertHeap(item)
{
    if (heap [0] == 0)
    { heap[0] = 1;
      heap[1] = item;
    }
    else
    {
        last = heap[0] +1;
        if (last >size)
            prompt message "Heap Full";
        else
        {
            heap[last] = item;
            heap[0] = heap [0] +1;
        }
    }
}

```

```

I = last;
Flag = 1;
while (Flag)
{
    J = abs (I/2);    /* find parent's index */
    if (J >= 1)
    {
        if (heap[J] < heap [I])
        {
            temp = heap[J];
            heap [J]= heap [I];
            heap [I] = temp;    /* Parent becomes child */
            I =J;
        }
        else
            Flag = 0;
    }
    else
        Flag=0;
}
}
}
}
}

```

Note: The insertion operation is important because the heap itself is created by iteratively inserting the various elements starting from an empty heap.

Example 9: Write a program that constructs a heap by iteratively inserting elements input by a user into an empty heap.

Solution: The algorithm `insertHeap()` would be used. The required program is given below:

```

/* This program creates a heap by iteratively inserting elements into a
heap Tree */

#include <stdio.h>
#include <conio.h>
void insertHeap(int heap[], int item, int size);
void dispHeap (int heap[]);
void main()
{
    int item, i, size;
    int maxHeap [20];
    printf ("\n Enter the size(<20) of heap");
    scanf ("%d", &size);
    printf ("\n Enter the elements of heap one by one");
    for (i=1; i<=size; i++)

```

```
{
    printf ("\n Element:");
    scanf ("%d", &item);
    insertHeap(maxHeap, item, size);
}
printf ("\n The heap is..");
dispHeap(maxHeap);
}

void insertHeap(int heap[],int item, int size)
{ int last, I, J, temp, Flag;
if (heap [0] == 0)
{ heap[0] = 1;
heap[1] = item;
}
else
{
    last = heap[0] + 1;
    if (last >size)
    {
        printf ("\n Heap Full");
    }
    else
    {
        heap[last] = item;
        heap[0]++;
        I = last;
        Flag = 1;
        while (Flag)
        {
            J = (int) (I/2);
            if (J >= 1)
            {
                /* find parent */
                if (heap[J] < heap [I])
                {
                    temp = heap[J];
                    heap [J]= heap [I];
                    heap [I] = temp;
                    I =J;
                }
            }
            else
                Flag = 0;
        }
        else
            Flag=0;
    }
}
```

```

    }
}
}

void dispHeap (int heap[])
{
    int i;
    for (i = 1; i <= heap[0]; i++)
        printf ("%d ", heap[i]);
}

```

The above program was tested for the input: **22 15 56 11 7 90 44**

The output obtained is: **90 15 56 11 7 22 44**, which is found to be correct.

In the next run, 89 was added to the list, i.e., 22 15 56 11 7 90 44 **89**.

We find that the item '89' has been inserted at proper place as indicated by the output obtained thereof: **90 89 56 15 7 22 44 11**

It is left as an exercise for the reader to verify above results.

7.5.3.3 Deletion of a Node from a Heap Tree The deletion operation on heap tree is relevant when its root is deleted. Deletion of any other element has insignificant applications. The root is deleted by the following steps:

Step

- (1) Save the root in temp.
- (2) Bring the right most leaf of the heap into root.
- (3) Move root down the heap till the heap is ordered.
- (4) Return temp containing the deleted node.

The operation, given in step2 above is also called reheap operation. Consider the heap given in Figure 7.33.

Let us delete its root (i.e., 70) by bringing the rightmost leaf (i.e., 56) into the root as shown in Figure 7.34. The root has been saved in temp.

An algorithm that deletes the maximum element (i.e., root) from a heap represented in an array of size N called `maxHeap` is given below. The number of elements currently present in the heap is stored in the zeroth location of `maxHeap`.

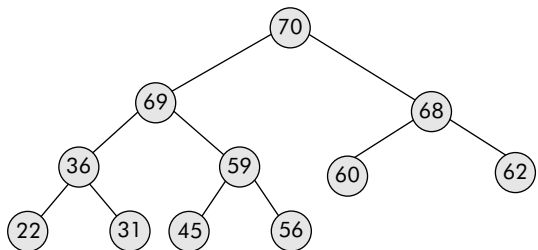


Fig. 7.33 A heap tree

Algorithm `delMaxHeap(heap)`

```

{
    if (heap[0] == 1) return heap[1];    /* There is only one element in the heap */
    last = heap[0];
    tempVal = heap[1];    /* Save the item to be deleted and returned */
    heap[1] = heap[last];    /* Bring the last leaf node to root */
}

```

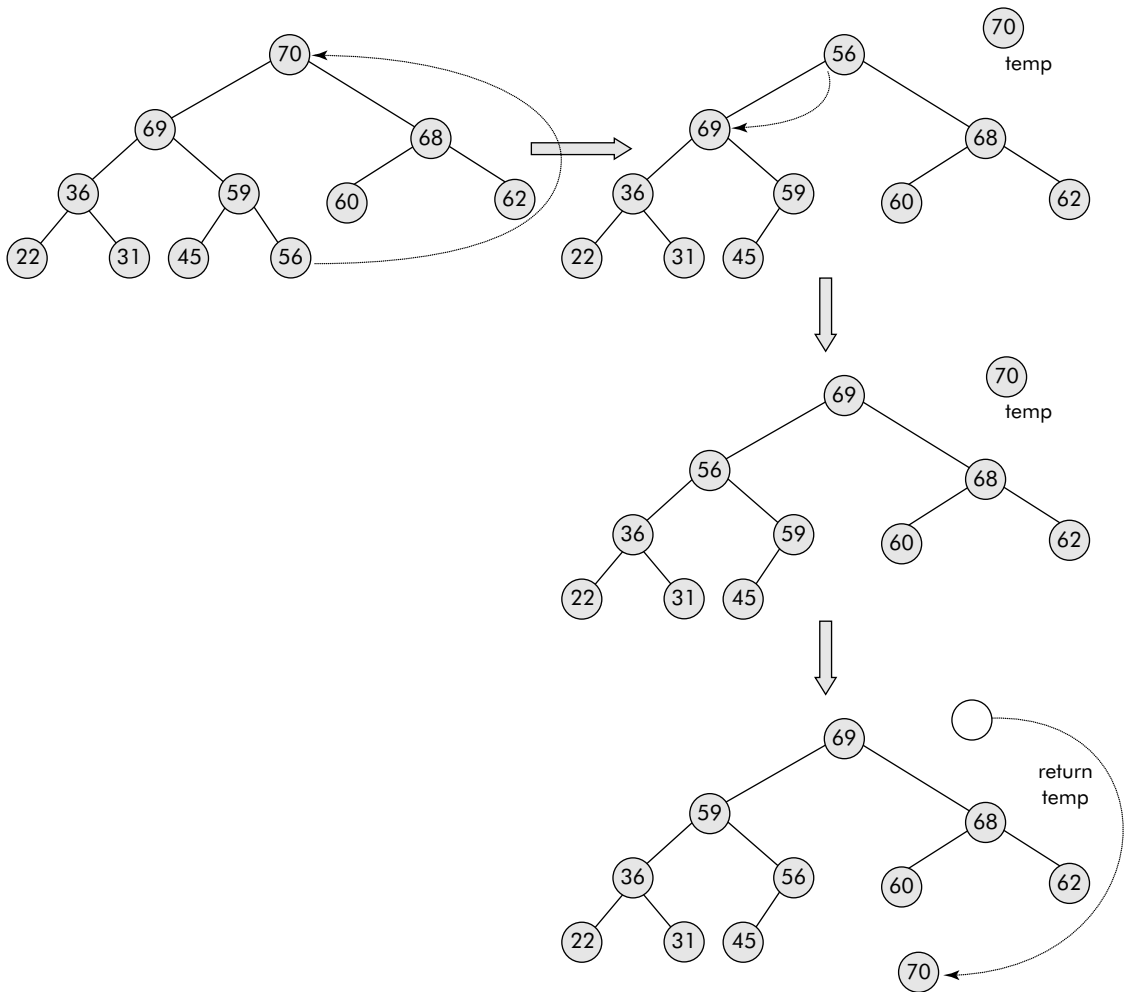


Fig. 7.34 Deletion of root from a heap

```

heap[0]--; /* Reduce the number of elements by 1 */
K = last - 1;
I=1; /* Point to root */
Flag = 1;
while (Flag)
{
    J = I*2;
    if (J <= K)
    { /* find bigger child and store its position in pos */
        if (heap[J] > heap[J+1])
            pos = J;
    }
    if (heap[I] > heap[pos])
        swap(heap[I], heap[pos]);
    I = pos;
}
return temp;

```

```

else
    pos = J+1;
if (heap [pos] > heap[I])    /* perform reheap */
{
    temp = heap[pos];
    heap [pos]= heap [I];
    heap [I] = temp;
    I =pos;    /* parent becomes the child */
}
else
    Flag = 0;
}
else
    Flag=0;
}
return tempVal;
}

```

Example 10: Write a program that deletes and displays the root of a heap. The remaining elements of the heap are put to reheap operation.

Solution: The algorithm `delMaxHeap()` would be used. The required program is given below:

```

/* This program deletes the maximum element from a heap Tree*/

#include <stdio.h>
#include <conio.h>
int delMaxHeap(int heap[]);
void insertHeap(int heap[],int item, int size);
void dispHeap(int heap[]);
void main()
{
    int item, i, size;
    int maxHeap [20];
    printf ("\n Enter the size(<20) of heap");
    scanf ("%d", &size);
    printf ("\n Enter the elements of heap one by one");
    for (i=1; i<=size; i++)
    {
        printf ("\n Element :");
        scanf ("%d", &item);
        insertHeap(maxHeap, item, size);
    }
    printf ("\n The heap is..");
    dispHeap(maxHeap);
    item = delMaxHeap(maxHeap);
    printf ("\n The deleted item is : %d", item);
    printf ("\n The heap after deletion is..");
}

```

```
    dispHeap(maxHeap);
}

void insertHeap(int heap[],int item, int size)
{ int last, I, J, temp, Flag;
  if (heap [0] == 0)
  { heap[0] = 1;
    heap[1] = item;
  }
  else
  {
    last = heap[0] +1;
    if (last >size)
    {
      printf ("\n Heap Full");
    }
    else
    {
      heap[last] = item;
      heap[0]++;
      I = last;
      Flag = 1;
      while (Flag)
      {
        J = (int) (I/2);
        if (J >= 1)
        { /* find parent */
          if (heap[J] < heap [I])
          {
            temp = heap[J];
            heap [J]= heap [I];
            heap [I] = temp;
            I =J;
          }
        }
        else
          Flag = 0;
      }
      else
        Flag=0;
    }
  }
}

void dispHeap (int heap[])
{
  int i;
```



```

    for (i = 1; i <= heap[0]; i++)
        printf ("%d ", heap[i]);
}
int delMaxHeap(int heap[])
{
    int temp, I, J, K, val, last, Flag, pos;
    if (heap[0] == 1) return heap[1];
    last = heap[0];
    val = heap[1];
    heap[1] = heap[last];
    heap[0]--;
    K = last - 1;
    I = 1;
    Flag = 1;
    while (Flag)
    {
        J = I * 2;
        if (J <= K)
        {
            /* find child */
            if (heap[J] > heap[J+1])
                pos = J;
            else
                pos = J+1;
            if (heap[pos] > heap[I])
            {
                temp = heap[pos];
                heap[pos] = heap[I];
                heap[I] = temp;
                I = pos;
            }
            else
                Flag = 0;
        }
        else
            Flag = 0;
    }
    return val;
}

```

The above program was tested for input: **34 11 56 78 23 89 12 5**
and the following output was obtained:

The heap is: **89 56 78 11 23 34 12 5**

The deleted item is: **89**

The heap after deletion is: **78 56 34 11 23 5 12**

It is left as an exercise for the students to verify the results by manually making the heaps before and after deletion of the root.

7.5.3.4 Heap Sort From discussion on heap trees, it is evident that the root of a maxHeap contains the largest value. When the root of the max heap is deleted, we get the largest element of the heap. The remaining elements are put to reheap operation and we get a new heap. If the two operations delete and reheap operation is repeated till the heap becomes empty and the deleted elements stored in the order of their removal, then we get a sorted list of elements in descending order as shown in Figure 7.35.

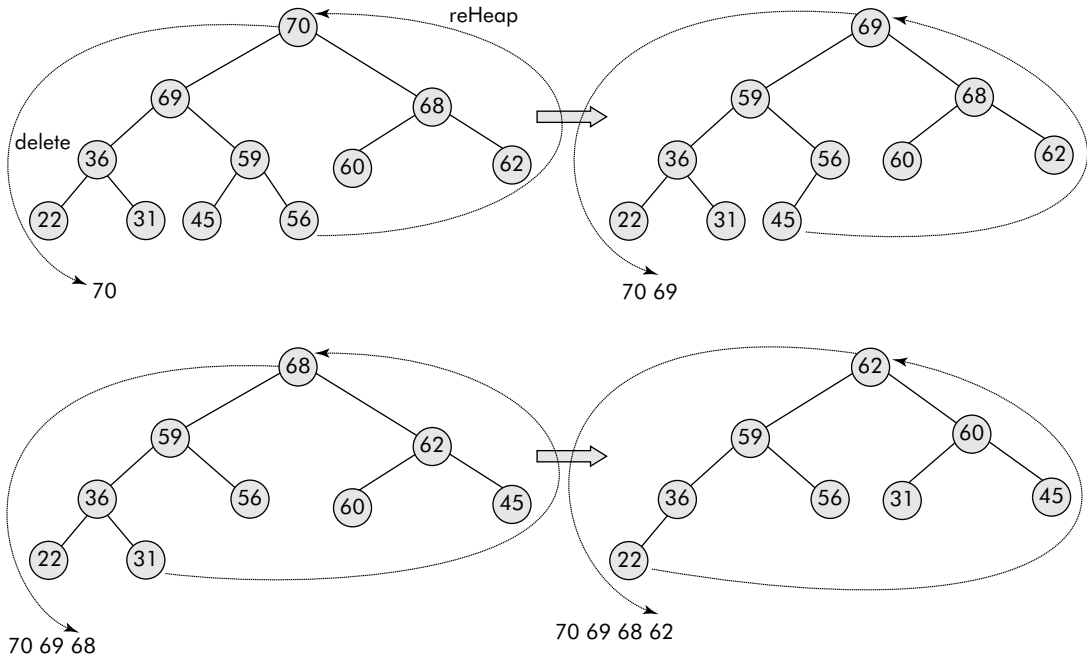


Fig. 7.35 Repetitive deletion from a heap produces sorted list

Example 11: Write a program that sorts a given list of numbers by heap sort.

Solution: We would modify the program of Example 10 wherein the element deleted from heap would be stored in a separate array called `sortList`. The deletion would be carried out iteratively till the heap becomes empty. Finally, `sortList` would be displayed. The required program is given below:

```
/* This program sorts a given list of numbers using a heap tree*/

#include <stdio.h>
#include <conio.h>
int delMaxHeap(int heap[]);
void insertHeap(int heap[],int item, int size);
void dispHeap (int heap[]);
void main()
```

```

{
    int item, i, size, last;
    int maxHeap [20];
    int sortList [20];
    printf ("\n Enter the size(<20) of heap");
    scanf ("%d", &size);
    printf ("\n Enter the elements of heap one by one");
    for (i=1; i<=size; i++)
    {
        printf ("\n Element :");
        scanf ("%d", &item);
        insertHeap(maxHeap, item, size);
    }
    printf ("\n The heap is..");
    dispHeap(maxHeap);
    getch();
    last = maxHeap[0];
    for (i =0; i < last; i++)
    {
        item = delMaxHeap(maxHeap);
        sortList [i] =item;
    }
    printf ("\n The sorted list...");
    for (i = 0; i < last; i++)
        printf ("%d ", sortList[i]);
}

```

```

void insertHeap(int heap[],int item, int size)
{ int last, I, J, temp, Flag;
  if (heap [0] == 0)
  { heap[0] = 1;
    heap[1] = item;
  }
  else
  {
    last = heap[0] +1;
    if (last >size)
    {
        printf ("\n Heap Full");
    }
    else
    {
        heap[last] = item;
        heap[0]++;
        I = last;
        Flag = 1;
    }
  }
}

```

```

while (Flag)
{
    J = (int) (I/2);
    if (J >= 1)
    { /* find parent */
        if (heap[J] < heap [I])
        {
            temp = heap[J];
            heap [J]= heap [I];
            heap [I] = temp;
            I =J;
        }
        else
            Flag = 0;
    }
    else
        Flag=0;
}
}
}

void dispHeap (int heap[])
{
    int i;
    for (i = 1; i <= heap[0]; i++)
        printf ("%d ", heap[i]);
}

int delMaxHeap(int heap[])
{
    int temp, I, J, K, val, last, Flag, pos;
    if (heap[0] == 1) return heap[1];
    last = heap[0];
    val = heap[1];
    heap[1]=heap[last];
    heap[0]--;
    K = last - 1;
    I=1;
    Flag = 1;
    while (Flag)
    {
        J = I*2;
        if (J <= K)
        { /* find child */
            if (heap[J] > heap[J+1])
                pos = J;

```

```

    else
        pos = J+1;
    if (heap [pos] > heap[I])
    {
        temp = heap[pos];
        heap [pos]= heap [I];
        heap [I] = temp;
        I = pos;
    }
    else
        Flag = 0;
}
else
    Flag=0;
}
return val;
}

```

The above program was tested for the input: **87 45 32 11 34 78 10 31**

The output obtained is given below:

The heap is: **87 45 78 31 34 32 10 11**

The sorted list is ... **87 78 45 34 32 31 11 10**

Thus, the program has performed the required sorting operation using heap sort.

Note: The above program is effective and gives correct output but it is inefficient from the storage point of view. The reason being that it is using an extra array called `sortList` to store the sorted list.

The above drawback can be removed by designing an **in-place heap sort** algorithm that uses the same array to store the sorted list in which the heap also resides. In fact, the simple modification could be that let root and the last leaf node be exchanged. Without disturbing the last node, reheap the remaining heap. Repeat the exchange and reheap operation iteratively till only one element is left in the heap. Now, the heap contains the elements in ascending order. The mechanism is illustrated in Figure 7.36.

Algorithm `IpHeapSort()`

```

{
    if (heap[0] > 1)
    {
        do
        {
            last = heap[0];
            temp = heap[1];    /* exchange root and last leaf node*/
            heap[1] = heap[last];
            heap[last] = temp;
            heap[0]--;
            last = heap[0];    /* last points to new last leaf node */
            K = last - 1;
        }
    }
}

```

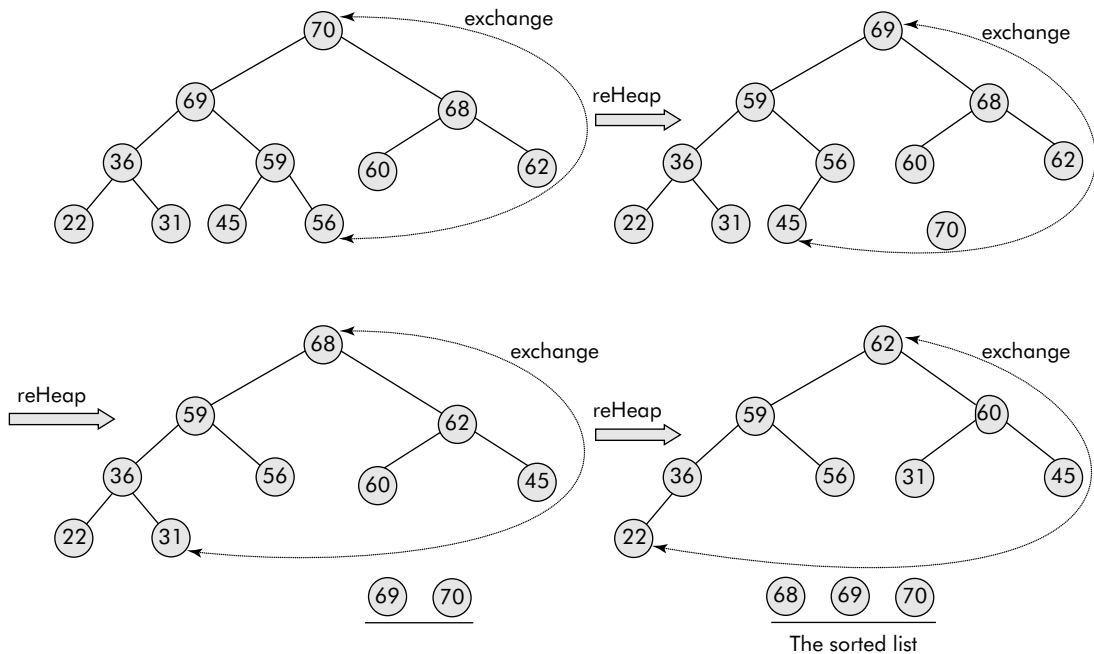


Fig. 7.36 The in-place heap sort

```

I=1;
Flag = 1;
while (Flag)
{
    J = I*2;
    if (J <= K)
    { /* find child which is greater than parent*/
        if (heap[J] > heap[J+1])
            pos = J;
        else
            pos = J+1;
        if (heap [pos] > heap[I])
        {
            temp = heap[pos]; /* exchange parent and child */
            heap [pos]= heap [I];
            heap [I] = temp;
            I =pos;
        }
    }
    else
        Flag = 0;
}

```

```

        else
            Flag=0;
    }
}
while (last >1);
}
}

```

It may be noted that the element at the root is being exchanged with the last leaf node and the rest of the heap is being put to reheap operation. This is iteratively being done in the do-while loop till there is only one element left in the heap.

Example 12: Write a program that in-place sorts a given list of numbers by heap sort.

Solution: We would employ the algorithm `IpHeapSort()` wherein the element from the root is exchanged with the last leaf node and rest of the heap is put to reheap operation. The exchange and reheap is carried out iteratively till the heap contains only 1 element.

The required program is given below:

```

/* This program in place sorts a given list of numbers using heap tree*/
#include <stdio.h>
#include <conio.h>
void sortMaxHeap(int heap[]);
void insertHeap(int heap[],int item, int size);
void dispHeap (int heap[]);
void main()
{
    int item, i, size, last;
    int maxHeap [20];
    printf ("\n Enter the size(<20) of heap");
    scanf ("%d", &size);
    printf ("\n Enter the elements of heap one by one");
    for (i=1; i<=size; i++)
    {
        printf ("\n Element :");
        scanf ("%d", &item);
        insertHeap(maxHeap, item, size);
    }
    printf ("\n The heap is..");
    dispHeap(maxHeap);
    getch();
    sortMaxHeap(maxHeap);
    printf ("\n The sorted list...");
    for (i = 1; i <= size; i++)
        printf ("%d ", maxHeap[i]);
}

void insertHeap(int heap[],int item, int size)
{ int last, I, J, temp, Flag;

```



```
if (heap [0] == 0)
{ heap[0] = 1;
  heap[1] = item;
}
else
{
    last = heap[0] +1;
    if (last >size)
    {
        printf ("\n Heap Full");
    }
else
{
    heap[last] = item;
    heap[0]++;
    I = last;
    Flag = 1;
    while (Flag)
    {
        J = (int) (I/2);
        if (J >= 1)
        { /* find parent */
            if (heap[J] < heap [I])
            {
                temp = heap[J];
                heap [J]= heap [I];
                heap [I] = temp;
                I =J;
            }
            else
                Flag = 0;
        }
        else
            Flag=0;
    }
}
}
}

void dispHeap (int heap[])
{
    int i;
    for (i = 1; i <= heap[0]; i++)
        printf ("%d ", heap[i]);
}
```



```

void sortMaxHeap(int heap[])
{
    int temp, I, J, K, val, last, Flag, pos;
    if (heap[0] > 1)
    {
        do
        {
            last = heap[0];
            temp = heap[1]; /*Exchange root with the last node */
            heap[1] = heap[last];
            heap[last] = temp;
            heap[0]--;
            last = heap[0];
            K = last -1;
            I=1;
            Flag = 1;
            while (Flag)
            {
                J = I*2;
                if (J <= K)
                {
                    /* find child */
                    if (heap[J] > heap[J+1])
                        pos = J;
                    else
                        pos = J+1;
                    if (heap [pos] > heap[I])
                    {
                        temp = heap[pos];
                        heap [pos]= heap [I];
                        heap [I] = temp;
                        I =pos;
                    }
                }
                else
                    Flag = 0;
            }
            else
                Flag=0;
        }
        while (last >1);
    }
}

```

7.5.3.5 Merging of Two Heaps Merging of heap trees is an interesting application wherein two heap trees are merged to produce a tree which is also a heap. The merge operation is carried out by the following steps:

Given two heap trees: Heap1 and Heap2

Step

- (1) Delete a node from Heap2.
- (2) Insert the node into Heap1.
- (3) Repeat steps 1 and 2 till Heap 2 is not empty.

Consider the merge operation shown in Figure 7.37.

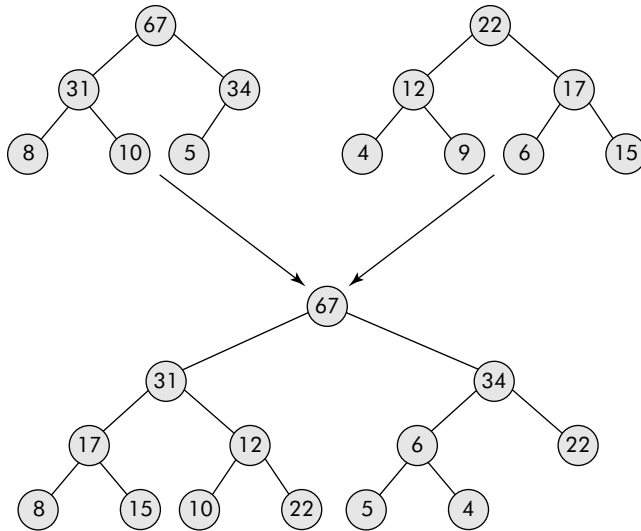


Fig. 7.37 Merging of two heap trees

Example 13: Write a program that merges two heap trees to produce a third tree which is also a heap.

Solution: The required program is given below:

```
/* This program merges two heap trees */

#include <stdio.h>
#include <conio.h>
int delMaxHeap(int heap[]);
void insertHeap(int heap[], int item, int size);
void dispHeap (int heap[]);
void main()
{
    int item,i,size1, size2;
    int maxHeap1 [20],maxHeap2[20];
    printf ("\n Enter the size(<20) of heap1");
    scanf ("%d", &size1);
    printf ("\n Enter the elements of heap one by one");
    for (i=1; i<=size1; i++)
```

```

{
    printf ("\n Element :");
    scanf ("%d", &item);
    insertHeap(maxHeap1, item, size1);
}
printf ("\n The heap1 is..");
dispHeap(maxHeap1);
printf ("\n Enter the size(<20) of heap2");
scanf ("%d", &size2);
printf ("\n Enter the elements of heap one by one");
for (i=1; i<=size2; i++)
{
    printf ("\n Element :");
    scanf ("%d", &item);
    insertHeap(maxHeap2, item, size2);
}
printf ("\n The heap2 is..");
dispHeap(maxHeap2);
for (i = 1; i <= size2; i++)
{
    item = delMaxHeap(maxHeap2);
    size1=size1+1;
    insertHeap (maxHeap1, item,size1);
}
printf ("\n The heaps after merge is..");
dispHeap(maxHeap1);
}

void insertHeap(int heap[], int item, int size)
{ int last, I, J, temp, Flag;
if (heap [0] == 0)
{ heap[0] = 1;
heap[1] = item;
}
else
{
    last = heap[0] +1;
    if (last >size)
    {
        printf ("\n Heap Full");
    }
    else
    {
        heap[last] = item;
        heap[0]++;
    }
}
}

```

```
I = last;
Flag = 1;
while (Flag)
{
    J = (int) (I/2);
    if (J >= 1)
    {
        /* find parent */
        if (heap[J] < heap [I])
        {
            temp = heap[J];
            heap [J]= heap [I];
            heap [I] = temp;
            I =J;
        }
        else
            Flag = 0;
    }
    else
        Flag=0;
}
}
}

void dispHeap (int heap[])
{
    int i;
    for (i = 1; i <= heap[0]; i++)
        printf ("%d ", heap[i]);
}

int delMaxHeap(int heap[])
{
    int temp, I, J, K, val, last, Flag, pos;
    if (heap[0] == 1) return heap[1];
    last = heap[0];
    val = heap[1];
    heap[1]=heap[last];
    heap[0]--;
    K = last -1;
    I=1;
    Flag = 1;
    while (Flag)
    {
        J = I*2;
        if (J <= K)
        {
            /* find child */
```

```

        if (heap[J] > heap[J+1])
            pos = J;
        else
            pos = J+1;
        if (heap [pos] > heap[I])
        {
            temp = heap[pos];
            heap [pos]= heap [I];
            heap [I] = temp;
            I =pos;
        }
        else
            Flag = 0;
    }
    else
        Flag=0;
}
return val;
}

```

The above program was tested on the heap trees given in Figure 7.37. The input provided is given below:

Heap1: 67 31 34 8 10 5

Heap2: 22 12 17 4 9 615

The output obtained is given below:

The heap after merge is ...67 31 34 17 12 6 22 8 15 10 9 5 4

It is left for the reader to verify the output from Figure 7.37.

Note: Both MaxHeap and MinHeap trees can be merged. The type of output merged tree will depend upon the type of Heap1. If Heap1 is maxHeap, then the heap obtained after merge operation would also be of type maxHeap. Similarly, if Heap1 is minHeap, then the heap obtained after merge operation would also be of type minHeap.

7.5.4 Threaded Binary Trees

From the discussion on binary trees, it is evident that creation of a tree is a one time process. But, a tree is travelled numerous times consuming a large part of the overall time taken by an algorithm. Moreover, the travel algorithms are recursive in nature amounting to large usage of storage. Therefore, there is a scope for reducing the travels or the number of steps required to travel a tree. For example, during inorder travel, we move from a node to its successor and from the successor to its successor, and so on.

The question is can we move directly to an inorder successor of a node in a tree. The answer is yes if we can keep pointers from the nodes to their successors and move through the tree using these pointers.

We know that in a binary tree of N nodes, there are ' $N + 1$ ' NULL pointers. Therefore, the unused NULL pointers of the binary tree can be employed for the purpose of non-recursive travelling within a binary tree. Let us modify the structure of the node of binary tree as per the following rules:

- If a node is having an empty left child (i.e., a NULL pointer), then replace it by a special pointer that points to its predecessor node in the tree.
- If a node is having an empty right child (i.e., a NULL pointer), then replace it by a special pointer that points to its successor node in the tree.
- The special pointer is called a thread.

Consider the binary tree given in Figure 7.38 wherein a binary tree has been assigned the threads. It may be noted that the normal pointers have been shown with thick lines and the threads with dotted lines.

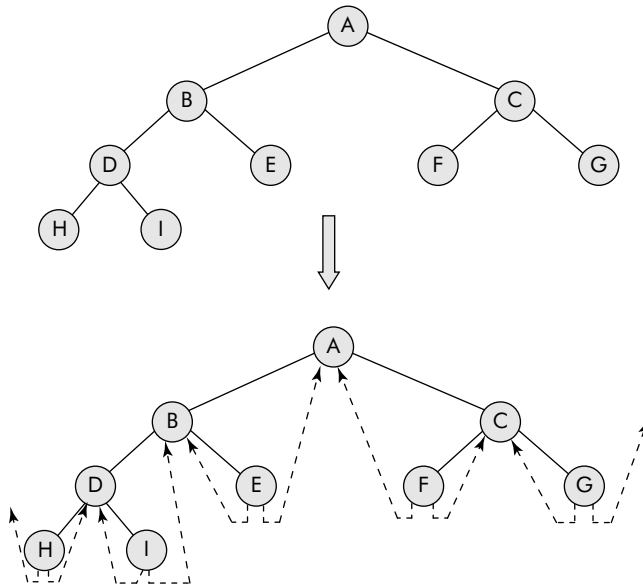


Fig. 7.38 The vacant NULL pointers have been replaced by threads

The resultant binary tree of Figure 7.38 is called a **threaded binary tree (TBT)**. It may be noted that the characteristic of this special tree is that a left thread of a node points to its inorder predecessor and the right thread to its inorder successor. For example, the left thread of 'E' is pointing to 'B', its predecessor and the right thread of 'E' is pointing to 'A', its successor.

A **threaded binary tree** is a binary tree in which a node uses its empty left child pointer to point to its inorder predecessor and the empty right child pointer to its inorder successor.

Now, the problem is that how a program would differentiate between a normal pointer and a thread. A simple solution is to keep variables called LTag and RTag, having value 1 or 0 indicating a normal pointer or a thread, respectively. Accordingly, a node of a threaded binary tree would have the structure as given in Figure 7.39.



Fig. 7.39 A node of a threaded binary tree

Let us now use the structure of Figure 7.39 to represent the threaded binary tree of Figure 7.38. The arrangement is shown in Figure 7.40.

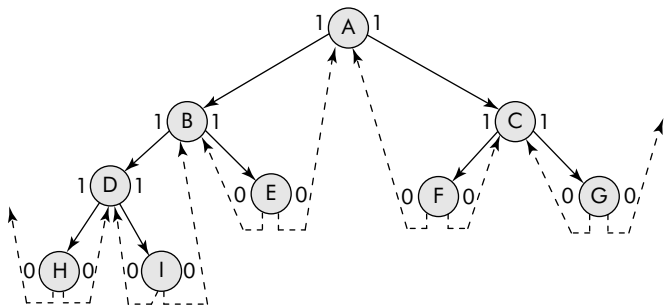


Fig. 7.40 The threaded binary tree with node structure having the Tag bits

A left thread pointing to NULL indicates that there is no more inorder predecessor. Similarly, a NULL right thread indicates that there is no more inorder successor. For example, the left thread of 'H' is pointing to NULL and the right thread of 'G' is pointing to NULL. This anomalous situation can be handled by keeping a head node having a left child and a right thread as shown in Figure 7.41. This dummy node does not contain any value. In fact, it is an empty threaded binary tree because both its left child and right thread are pointing to itself.

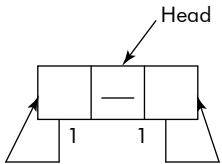


Fig. 7.41 A head node indicating an empty threaded binary tree

Now the dangling threads of a binary tree can be made to point to the head node and the left child of the head node to the root of the binary threaded tree as shown in Figure 7.42.

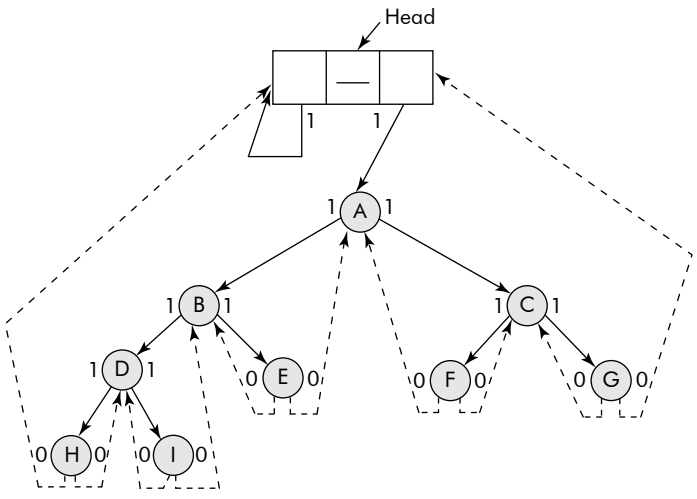


Fig. 7.42 A binary threaded tree with a head node

Now with the arrangement shown in Figure 7.42, it can be appreciated that an inorder successor of a node pointed by ptr can be very easily found by the following set of rules:

- If RTag of ptr is 0, then the rightChild is pointing to its successor.
- If RTag of ptr is 1 (i.e., thread), then the left most node of its right sub-tree is its successor.

An algorithm that returns inorder successor of a node is given below:

```
Algorithm findSucc(ptr)
{
    if (RTag (ptr) == 0)
        return rightChild (ptr);
    else
    {
        ptr = rightChild (ptr)
        while (LTag (ptr) =1)
            ptr = leftChild(ptr);
    }
    return ptr;
}
```

Similarly, an inorder predecessor of a node pointed by ptr can be very easily found by the following set of rules:

- If LTag of ptr is 0, then the leftChild is pointing to its predecessor.
- If LTag of ptr is 1 (i.e., thread), then the right most node of its left sub-tree is its predecessor.

An algorithm that returns inorder predecessor of a node is given below:

```
Algorithm findPred(ptr)
{
    if (LTag (ptr) == 0)
        return leftChild (ptr);
    else
    {
        ptr = leftChild (ptr)
        while (RTag (ptr) =1)
            ptr = rightChild(ptr);
    }
    return ptr;
}
```

Now we can write an algorithm for non-recursive inorder travel of a threaded binary tree (TBT), which is given below:

```
Algorithm inOrderTBT(Tree)
{
    Tree = Head;
    while (1)
    {
        tree = findSucc(tree);
        if (tree = Head) return;
        else
    }
```



```

    print DATA (Tree);
}
}

```

The following operations can be defined on a threaded binary tree.

- Insertion of a node into a TBT.
- Deletion of a node from a TBT.

As the threaded binary tree has been designed to honour inorder travel, the insertion and deletion have to be carried out in such a way that after the operation the tree remains the inorder threaded binary tree. A brief discussion on both the operations is given in the subsequent sections.

7.5.4.1 Insertion into a Threaded Binary Tree A node 'X', having Data part equal to 'T', can be inserted into an inorder threaded binary tree after node 'Y' in either of the following four ways:

- Case 1: When X is inserted as left child of Y and Y has an empty left child.
- Case 2: When X is inserted as right child of Y and Y has an empty right child.
- Case 3: When X is inserted as left child of Y and Y has a non-empty left child
- Case 4: When X is inserted as right child of Y and Y has a non-empty right child.

CASE 1:

When Y (Say Data part equal to 'F') has an empty left child, then it must be a thread. The insertion can be made by making the left thread of X (Data part equal to 'T') to point where the left thread of Y is currently pointing to. Thereafter, X becomes the left child of Y for which LTag of Y is reset to 1. The right child of X is set as thread pointing to the node Y, indicating that Y is the successor of X. The insertion operation is shown in Figure 7.43.

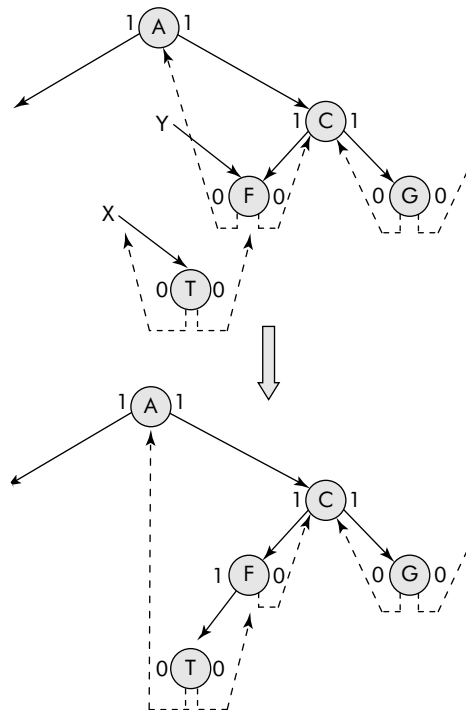


Fig. 7.43 Insertion of a node X after Y having no left child

It may be noted that the node containing 'T' has become the left child of node containing 'F'. After the insertion, the tree is still a threaded binary tree. The predecessor of 'T' is 'A' and the successor of 'T' is 'F'.

An algorithm for insertion of node as left thread is given below:

Algorithm insertAtLThread(X, Y)

```

{
  if (LTag(Y) == 0)
  {
    LTag (X) = 0;
    leftChild (X) = leftChild (Y);
    LTag (Y) = 1;
    leftChild (Y) = X;
    RTag (X) = 0;
    rightChild (X) = Y;
  }
}

```

CASE 2:

When Y (Data part equal to 'F') has an empty right child, then it must be a thread. The insertion can be made by making the right thread of X (Data part equal to 'T') to point where the right thread of Y is currently pointing to. Thereafter, X becomes the right child of Y for which RTag of Y is reset to 1. The left child of X is set as a thread pointing to the node Y, indicating that Y is the predecessor of X. The insertion operation is given in Figure 7.44.

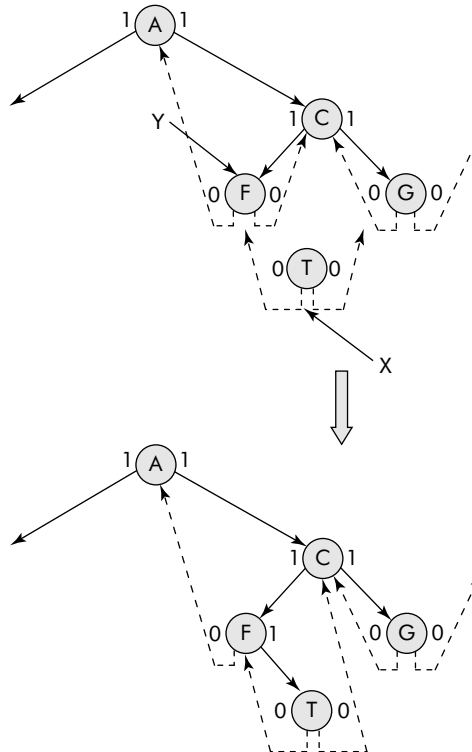


Fig. 7.44 Insertion of a node X after Y having no right child

It may be noted that the node containing 'T' has become the right child of node containing 'F'. After the insertion, the tree is still a threaded binary tree. The predecessor of 'T' is 'F' and the successor of 'T' is 'C'.

An algorithm for insertion of node as right thread is given below:

```
Algorithm inserAtRThread(X,Y)
{
    if (RTag(Y) == 0)
    {
        RTag (X) =0;
        rightChild (X) = rightChild (Y);
        RTag (Y) =1;
        rightChild (Y) = X;
        LTag (X) = 0;
        leftChild (X) = Y;
    }
}
```

CASE 3:

When Y (Data part equal to 'C') has non-empty left child, the insertion can be made by making the left child of X (Data part equal to 'T') point where the left child of Y is currently pointing to. Thereafter, X becomes the left child of Y for which Lchild of Y is pointed to X. The right child of X is set as thread pointing to the node Y, indicating that Y is the successor of X.

Now the most important step is to find the inorder predecessor of Y (pointed by ptr) and make it inorder predecessor of X by making ptr's right thread to point to X. The insertion operation is shown in Figure 7.45.

It may be noted that the node containing 'T' has become the left child of node containing 'C'. After the insertion, the tree is still a threaded binary tree. The predecessor of 'T' is 'F' and the successor of 'T' is 'C'.

An algorithm for insertion of node as left child is given below:

```
Algorithm insertAtLchild(X,Y)
{
    RTag (X) =0;    /* Connect the right thread of X to Y */
    rightChild (X) = Y
    LTAG (X) = 1;    /* Connect the leftChild of X to the node pointed by
    left child o Y */
    leftChild (X) = leftChild (Y)
                    /* Point the left child of Y to X */
    leftChild (Y) = X;
    ptr = inOrderPred (Y)    /* Find the inorder predecessor of Y */
    rightChild (ptr) =X;    /* Connect the right thread of (ptr) predecessor
    to X */
}
```

CASE 4:

When Y (Data part equal to 'C') has non-empty right child, the insertion can be made by making the right child of X (Data part equal to 'T') point where the right child of Y is currently pointing to. Thereafter, X becomes the right child of Y by pointing the right child of Y to X. The left thread of X is set as a thread pointing to the node Y, indicating that Y is the predecessor of X.

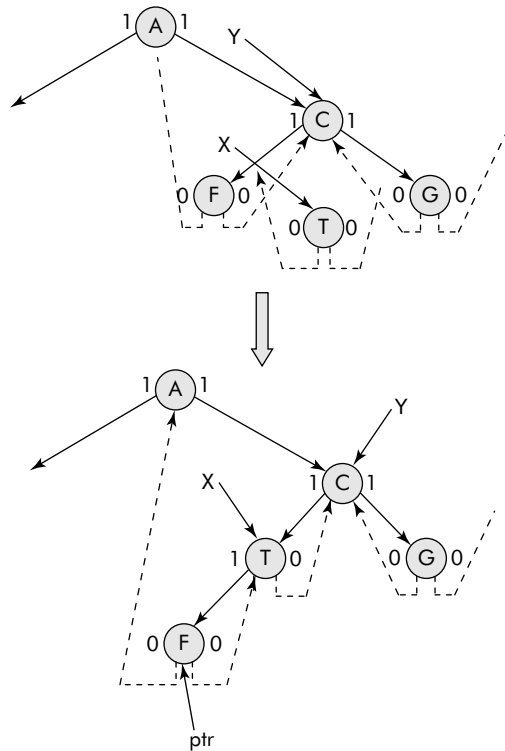


Fig. 7.45 Insertion of a node at non empty left child of a given node

Now, the most important step is to find the inorder successor of Y (pointed by *ptr*) and make it inorder successor of X by making *ptr*'s left thread point to X. The insertion operation is shown in Figure 7.46.

It may be noted that the node containing 'T' has become the right child of node containing 'C'. After the insertion, the tree is still a threaded binary tree. The predecessor of 'T' is 'C' and the successor of 'T' is 'G'.

An algorithm for insertion of node as right child is given below:

Algorithm insertAtRchild(X, Y)

```
{
    LTag (X) = 0;    /* Connect the left thread of X to Y */
    leftChild (X) = Y
    RTag (X) = 1;    /* Connect the leftChild of X to the node pointed by
                     left child o Y */
    rightChild (X) = rightChild (Y)
                     /* Point the left child of Y to X */
    rightChild (Y) = X;
    ptr = inOrderSucc (Y) /* Find the inorder successor of Y */
    leftChild (ptr) =X; /* Connect the left thread of ptr (successor) to X */
}
```

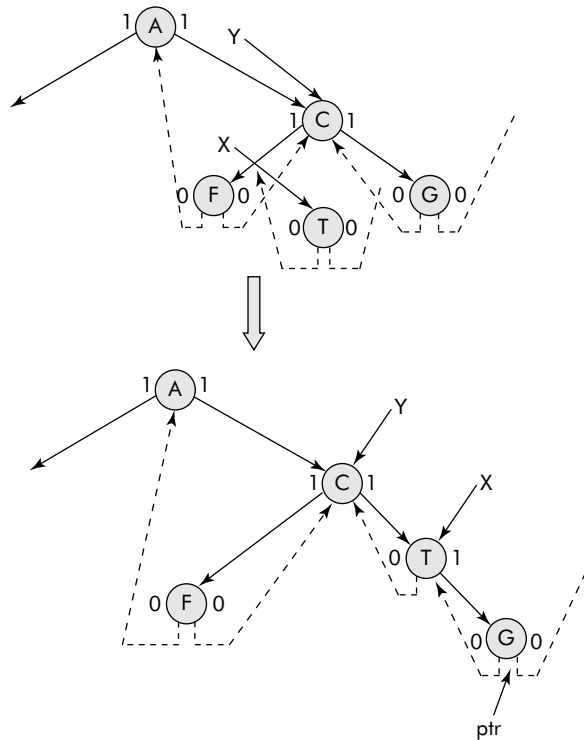


Fig. 7.46 Insertion of a node at non-empty right child of a given node

A brief discussion on deletion of a node from a threaded binary tree is given in Section 7.5.4.2.

7.5.4.2 Deletion from a Threaded Binary Tree A node 'X', having Data part equal to 'T', can be deleted from an inorder threaded binary tree in any of the following five ways:

- Case 1: When X is a left leaf node.
- Case 2: When X is a right leaf node
- Case 3: When X is only having a right sub-tree.
- Case 4: When X is only having a left sub-tree.
- Case 5: When X is having both sub-trees.

Before an attempt is made to delete a node from any tree, the parent of the node must be reached first so that the dangling pointers produced after deletion can be properly handled. Therefore, the following algorithm is given that finds the parent 'Y' of a node 'X'.

```
Y = Tree;
```

```
Algorithm findParent (X,Y)
```

```
{
  If (Y = NULL) return;
  if ((Y -> leftChild == X || Y -> (rightChild == X)) return Y
```

```

    findParent (X, Y -> leftChild);
    findParent (X, Y -> rightChild);
}

```

The above algorithm is recursive. A non-recursive algorithm that uses the threads to reach to the parent of a node X is given below. It uses the strategy wherein it reaches the successor of the rightmost node. If the left child of the successor points to X, then the successor is parent. Otherwise, move to the left child and then keep going to right till a node is found whose right child is equal to X.

```

Algorithm findParent (X)
{
    ptr = X;
    while (RTag(ptr) == 1)    /* Find the node which has a right thread */
        ptr = rightChild (ptr);
    succ = rightChild (ptr);    /* Move to the successor node */
    if (succ -> leftChild == X)
        Y = succ;    /* Parent found, point it by Y*/
    else
        { suc = leftChild (succ);
        while (rightChild (succ) != X)
            succ = rightChild (succ);
        Y = succ;
        }
    return Y;
}

```

Now the algorithm findParent() can be conveniently used to delete a node from a threaded binary tree. The various cases of deletion are discussed below:

CASE 1:

When X is a left leaf node, then its parent Y is found (see Figure 7.47). The left child of Y is made to point where X's left thread is currently pointing to. The LTag of Y is set to 0, indicating that it has become a thread now.

An algorithm for deletion of a node X which is a left leaf node is given below:

```

Algorithm delLeftLeaf (X,Y)
{
    Y = findParent (X);
    if (leftChild (Y) == X)
    {
        leftChild (Y) = leftChild (X);    /* Make left child of parent as thread and let
                                           it point where X's left thread is pointing */
        LTag (Y) = 0;
    }
    return X;
}

```

It may be noted from Figure 7.47 that node X has been deleted and the left child of its parent Y has been suitably handled.

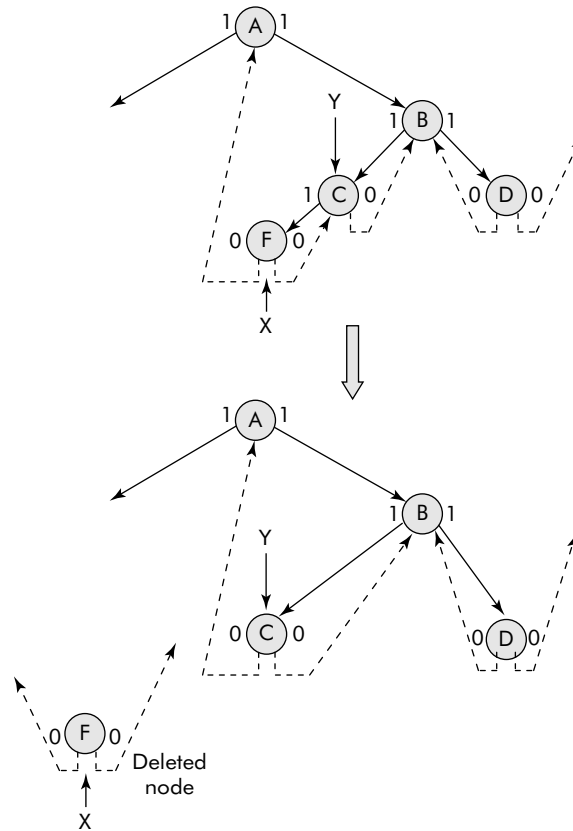


Fig. 7.47 Deletion of a left leaf node

CASE 2:

When X is a right leaf node, then its parent Y is found. The right child of Y is made to point where X's right thread is currently pointing to. The RTag of Y is set to 0, indicating that it has become a thread now.

An algorithm for deletion of a node X which is a right leaf node is given below:

Algorithm delRightLeaf (X,Y)

```
{
    Y = findParent (X);
    if (rightChild (Y) == X)
    {
        rightChild (Y) = rightChild (X);    /* Make right child of parent as thread
                                              and let it point where X's left thread is
                                              pointing */

        RTag (Y) = 0;
    }
    return X;
}
```

This case is similar to CASE 1 and, therefore, illustration for this case is not provided.

CASE 3:

When X is having a right sub-tree, its parent Y is found. A pointer called 'grandson' is made to point to the right child of X (see fig 7.48). The right child of Y is pointed to grandson. The successor of X is found. The left thread of successor is pointed to Y, indicating that the successor of X has now become the successor of parent as shown in Figure 7.48.

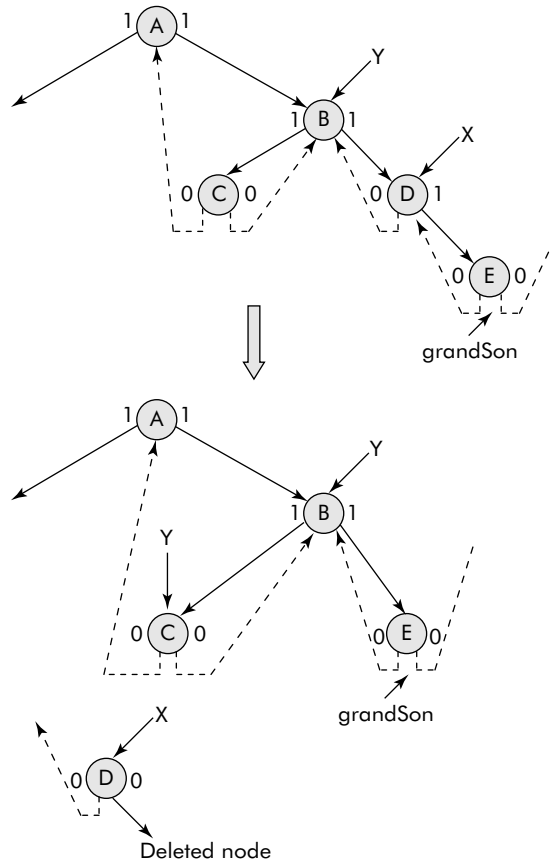


Fig. 7.48 Deletion of a node having only right sub-tree

It may be noted that in Figure 7.48, the grandson is also the successor of X.

An algorithm for deletion of X when X is having a right sub-tree is given below:

Algorithm delNonLeafR (X, Y)

```
{
    Y = findParent (X);
    if (RTag (X) == 1)
    {
        grandson = rightChild (X);
```



```

    rightChild (Y) = grandson;
    succ = inOrderSucc(X);
    leftChild (succ) = Y;
}
return X
}

```

CASE 4:

When X is having only left sub-tree, its parent Y is found. A pointer called 'grandson' is made to point to the left child of X. The left child of Y is pointed to grandson. The predecessor of X is found. The right thread of predecessor is pointed to Y, indicating that the predecessor of X has now become the predecessor of parent.

An algorithm for deletion of X when X is having a left sub-tree is given below:

Algorithm delNonLeafL (X, Y)

```

{
    Y = findParent (X);
    if (LTag (X) == 1)
    {
        grandson = leftChild (X);
        leftChild (Y) = grandson;
        pred = inOrderPred(X);
        rightChild (pred) = Y;
    }
    return X
}

```

This case is similar to CASE 3 and, therefore, illustration for this case is not provided.

CASE 5:

When X is having both sub-trees, then X is not deleted but its successor is found. The data of successor is copied in X and the successor is deleted using either CASE 1 or CASE 2, whichever is applicable.

An algorithm for deletion of X when X is having both sub-trees is given below:

Algorithm delBoth (X)

```

{
    succ = findSucc (X);
    DATA (X) = DATA (succ);
    Delete succ using Case1 or Case2
}

```

Note: The applications of threaded binary trees are limited in the sense that the above discussions pertain to inorder threaded binary trees only. For postorder, the above algorithms need to be rewritten. The implementation is rather complex.

7.6 WEIGHTED BINARY TREES AND HUFFMAN ALGORITHM

Information is an important resource but transportation of right information to a rightful receiver is equally important. For example, a specialist doctor in Delhi must know the complete online data of a medical tourist (say from Afghanistan) from the patient's hospital in Kabul.

A search engine collects information from various Web sites with the help of downloaders called *crawlers*. The information, if large, needs to be compressed before sending it over network so as to save bandwidth traffic. The huge data collected by the search engine needs to be encoded prior to storing it in its local repository.

Similarly, Fax machines must also compress and encoded data before throwing it on the telephone line.

All these applications require efficient encoding techniques. In this section, Huffman coding technique is discussed with a view to introduce the reader to the area of *data compression*, a potential area of research nowadays.

The following terms of binary trees are important for discussions related to the topic of this section.

2-Tree or Extended Binary Tree: It is binary tree with the property that each node has either 0 children or 2 children. The tree given in Figure 7.49 is an Extended Binary Tree (EBT)

Internal Node: The node with two children is called an internal node. For example, in Figure 7.49, the nodes B, C, D, E, and F are internal nodes.

External Node: A node with no children

is called an external node. This is also popularly known as a *leaf node*. For example, the nodes G, H, I, J, K, L, and M of Figure 7.49 are external or leaf nodes.

Path Length: The number of edges traversed to reach a node is called the path length of the node. For example the path length of Node D is 2 and that of Node K is 3. Of course, the path length of root node A is 0.

As per convention, it may be noted that internal and external nodes have been represented by circle and rectangle, respectively. This representation of a binary tree is also called as Extended Binary Tree (EBT). Thus, in EBT, an internal node is represented as a circle and the external node as a rectangle.

External Path Length: Let L_{EXT} be the external path length of a binary tree. It is defined as sum of path lengths of all external nodes. The external path length of the tree given in Figure 7.49 is given below:

$$L_{EXT} = L_H + L_I + L_J + L_K + L_L + L_M + L_G = 3 + 3 + 3 + 3 + 3 + 3 + 2 = 20$$

Where L_i : is the path length of external node i

This can be formally defined as:

$$L_{EXT} = \sum_{i=1}^N L_i$$

Where L_i : is the path length of external node i

Internal Path length: Let L_{INT} be the internal path length of a binary tree. It is defined as sum of path lengths of all internal nodes. The internal path length of the tree given in Figure 7.49 is given below:

$$L_{INT} = L_A + L_B + L_C + L_D + L_E + L_F = 0 + 1 + 1 + 2 + 2 + 2 = 8$$

Where L_i : is the path length of internal node i

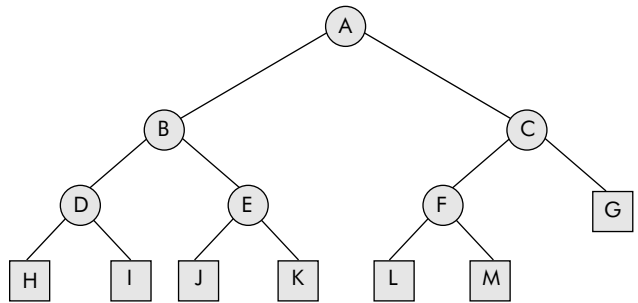


Fig. 7.49 An extended binary tree

This can be formally defined as:

$$L_{INT} = \sum_{i=1}^N L_i$$

Where L_i : is the path length of internal node i

From induction , we can say that : $L_{EXT} = L_{INT} + 2 * N$, where N is number of internal nodes.

In EBT of Figure 7.49, $N = 6$, therefore, $L_{EXT} = L_{INT} + 2 * N = 8 + 2 * 6 = 20$.

Weighted Binary Tree: So far we have considered the cost of an edge between two nodes as one unit. In real world the edges may differ in terms of length, weight, degree of comfort, profit, utility etc. However, in order to keep things simpler, let us keep the cost between each edge as one but assign a weight W_i to an i th external node as shown in Figure 7.50.

Since the weighted binary tree is also an extended binary tree, therefore it is also called as a weighted extended binary tree.

Let W_i be the weight of an i th external node with path length L_i then the weighted external path length of weighted binary tree L_w is defined as:

$$L_w = \sum_{i=1}^N W_i * L_i$$

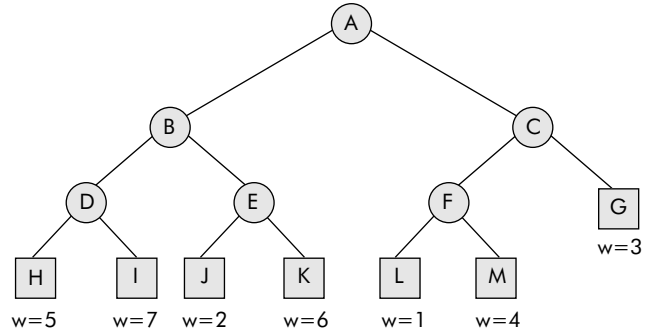


Fig. 7.50 A weighted binary tree

Where N is the number of external nodes.

From above given relation, the weighted path length of the extended binary tree of Figure 7.50 is computed as:

$$L_w = 5 * 3 + 7 * 3 + 2 * 3 + 6 * 3 + 1 * 3 + 4 * 3 + 3 * 2 = 81$$

The purpose of above discussion was to create a weighted binary tree with minimum weighted path length. It has many applications such as: data compression techniques for sending messages over communication networks, storing of collection of documents on secondary storage, etc.

D. Huffman provided a solution to this problem. The Huffman algorithm is discussed in next section.

7.6.1 Huffman Algorithm

For a given list of weights $w_1, w_2, w_3, \dots, w_n$, find a weighted binary tree T of minimum-weighted path length out of all the possible weighted binary trees. Huffman gave a bottom up approach for this problem. The simple steps for solving this problem are given below:

1. Sort the given list of weights into increasing order, place them on a priority queue called List.
2. Pick two minimum weights from the List as external nodes and form a sub-tree
3. Compute the weighted path list of the sub-tree obtained in Step 2. Insert this weight in the List
4. Repeat steps 2–3 till only one weight is left on the List.

Consider the external weights of the tree given in Figure 7.50. The external nodes with their weights are:

G	H	I	J	K	L	M
3	5	7	2	6	1	4

The sorted list is given below:

L	J	G	M	H	K	I
1	2	3	4	5	6	7

The trace of above given steps is provided in Figure 7.51:

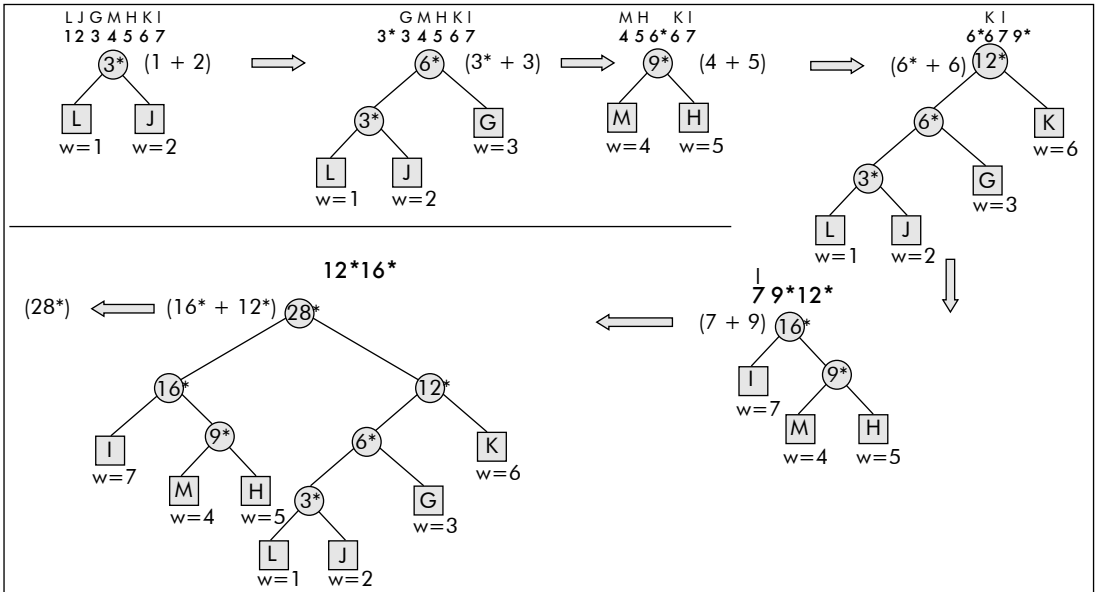


Fig. 7.51 Construction of a Huffman Tree

A copy of the final Huffman tree obtained in Figure 7.51 is given in Figure 7.52

Now the weighted path length of this tree can be computed by adding the weights of all internal nodes as given below:

$$L_w = 28 + 16 + 9 + 12 + 6 + 3 = 74$$

It is left as an exercise to the reader to create another weighted binary tree from the list of weights and verify that the Huffman tree obtained above has the minimum weighted path length.

It may be noted that when duplicate values are available in the list of weights then depending upon the order of their positions

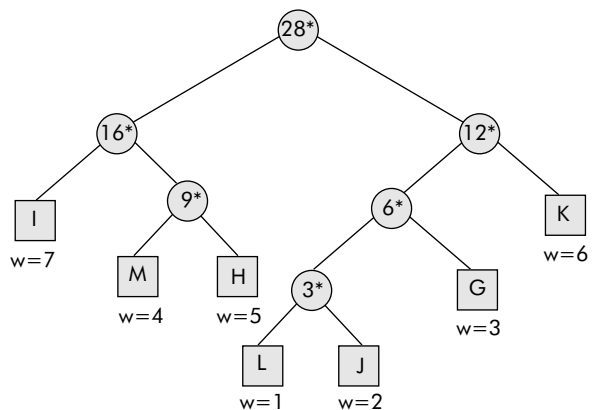


Fig. 7.52 The final Huffman Tree

in the list, we may have many different Huffman trees. For instance, in the example given above in second step if we swap the values of 3* and 3 and in third step, swap the values of 6* and 6, we would get a different Huffman tree as compared to the Huffman tree obtained above (see Figure 7.53). However, both the Huffman trees will have the same weighted path lengths, i.e. 74.

Thus, construction of Huffman tree by the Huffman algorithm may not always result in a unique tree.

As discussed above, the Huffman tree can be usefully applied to data compression techniques for storing files and sending messages. Infact, Huffman not only gave the algorithm for construction of Huffman tree but also offered a coding technique for data compression applications. The technique is discussed in next section.

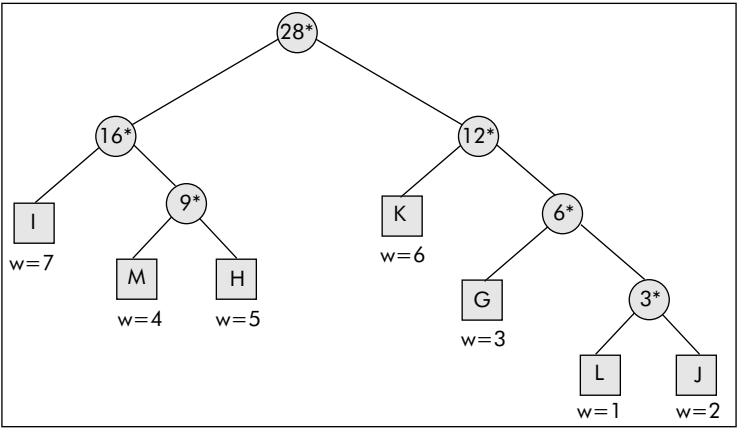


Fig. 7.53 An alternate Huffman Tree

7.6.2 Huffman Codes

Normally, the alphabets are stored using standard codes such as ASCII, UNICODE, EBCDIC etc. These are fixed length codes. The ASCII coding scheme uses 8 bits for a character whereas the UNICODE uses 16 bits. These codes are useful only when the text, to be stored, contains all the alphabets with equal number of their instances in the text, otherwise it becomes a costly method in terms of storage. For example, the string ‘madam’ would use 40 bits of storage for using ASCII code and 80 bits for UNICODE.

However, a closer look reveals that the string consists of 2 instances of alphabet ‘m’, 2 instances of alphabet ‘a’, and 1 instance of alphabet ‘d’. Let us now assign them the following binary codes:

Alphabet	Binary code
d	00
a	01
m	10

Using the above codes, we can compress the string ‘madam’ into a much shorter code of 10 bits as shown below:

1001000110

Thus, we have saved a significant amount of storage space i.e. 10 bits as compared to 40 or 80 bits. But the coding scheme used by us is still a fixed length coding scheme.

Huffman provided an excellent variable length coding scheme which effectively compresses the data by assigning short codes to most frequently occurring characters. The less frequent characters get longer codes. Thus, the technique saves the storage space in terms of 25–90 per cent of the original size

of the text. For a given piece of text, the following steps are used for developing the codes for various alphabets contained in the text:

1. Count the frequency of occurrence of each alphabet in the text.
2. Based on the frequency, assign weights to the alphabets.
3. Create a Huffman tree using the weights assigned to the alphabets. Thus, the alphabets become the external nodes of the Huffman tree.
4. Starting from root, label the left edge as 0 and right as 1.
5. Traverse the sequence of edges from root to an external node i.e. an alphabet. The assembly of labels of the edges from root to the alphabet becomes the code for the alphabet.

The Huffman codes applied to Huffman tree of Figure 7.52 are given in Figure 7.54

The codes assigned to various leaf nodes are listed in the table given below:

Character	Weight	Code
I	7	00
K	6	11
H	5	011
M	4	010
G	3	101
J	2	1001
L	1	1000

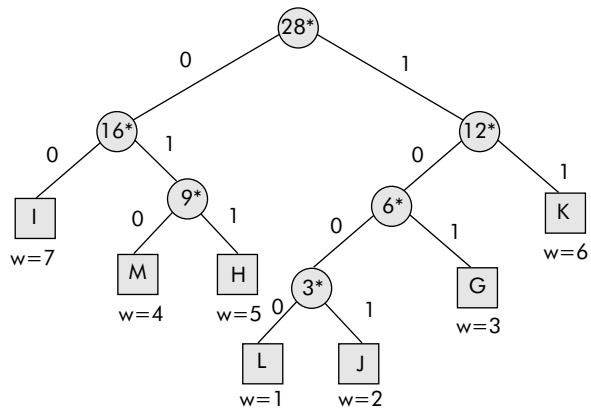


Fig. 7.54 The assignment Huffman codes

It may be noted that the heaviest nodes: I and K have got shorter codes than the lighter nodes.

Example 14: Develop Huffman code for this piece of text ‘Nandini’

Solution: The trace of steps followed for assigning the Huffman codes to the various characters of the given text are given below:

1. The frequency count of each character appearing in the above text is given in the following table:

Character	Frequency
N	1
a	1
d	1
n	2
i	2

2. Let the frequency be the weight of each character.
3. The sorted list of the characters in the order of their weights is given below:

N a d n i
1 1 1 2 2

4. From the weights develop the Huffman Tree. The trace of Huffman algorithm is given Figure 7. 55.

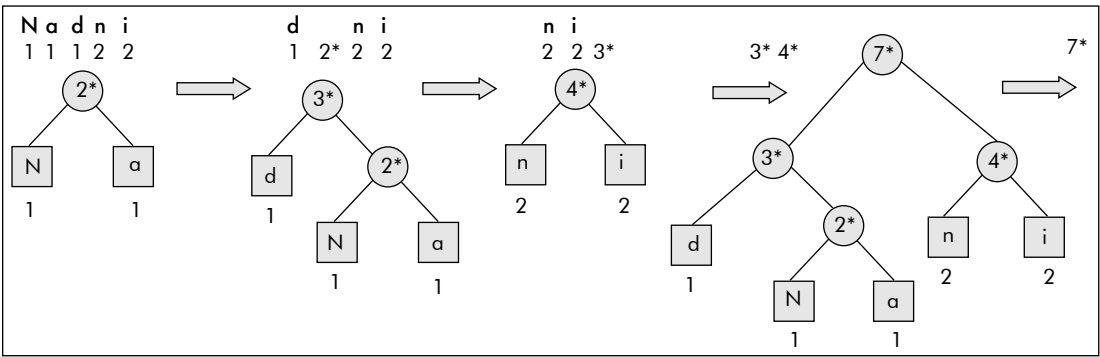


Fig. 7.55 Development of the Huffman tree

5. Assign the Huffman codes as shown in Figure 7.56.

The codes assigned to various leaf nodes are listed in the table given below:

Character	Weight	Code
d	1	00
N	1	010
a	1	011
n	2	10
i	2	11

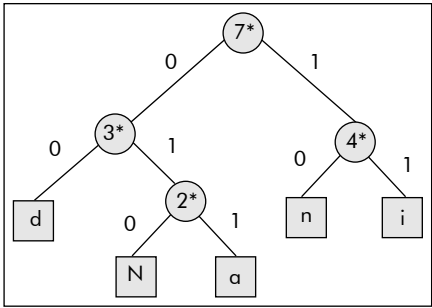


Fig. 7.56 Assignment of Huffman codes

Now, the code assigned to the text 'Nandini' is: **0100111000111011 = 16 bits**

In ASCII coding scheme, we would have used $8 \times 7 = 56$ bits. Thus, we have saved 40 bits (56 – 16) of storage space in this small piece of text comprising of only 7 characters.

Example 15: Develop Huffman code for this piece of text given below:

“when you are on the left you are on the right. when you are on the right, you are on the wrong”

Also compute the amount storage space in terms of bits saved by using Huffman coding.

Solution: The text contains 93 characters. We will represent the space with the character: B. The trace of steps followed for assigning the Huffman codes to the various characters of the given text are given below:

Step 1: The frequency count of each character appearing in the above text is given in the following table:

Character	Frequency	Character	Frequency	Character	Frequency
w	3	u	4	i	2
h	8	a	4	g	3
e	11	r	7	.	2
n	7	t	7	,	2
y	4	l	1	Б (space)	18
o	9	f	1		

- Let the frequency be the weight of each character.
- The sorted list of the characters in the order of their weights is given below:
l f l . , w g y u a n r t h o e Б
1 1 2 2 2 3 3 4 4 4 7 7 7 8 9 11 18
- From the weights, develop the Huffman tree with the help of Huffman algorithm. The trace is given in Appendix D.
- Assign the Huffman codes as shown in Figure 7.57.

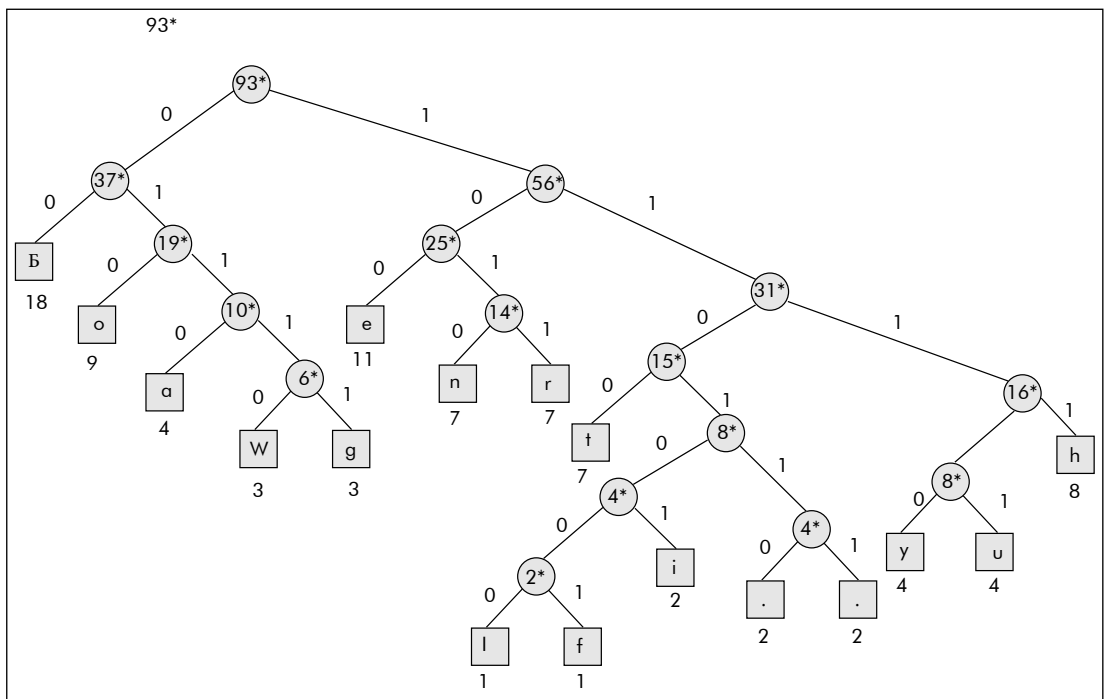


Fig. 7.57 Assignment of Huffman codes

The codes assigned to various leaf nodes are listed in the table given below:

Character	Frequency	Code	Character	Frequency	Code
Б (space)	18	00	l	1	1101000
o	9	010	f	1	1101000
a	4	0110	i	2	110101
w	3	01110	.	2	110110
g	3	01111	,	2	110111
e	11	100	y	4	11100
n	7	1010	u	4	11101
r	7	1011	h	8	1111
t	7	1100			

As expected, the most frequently occurring character called space (Б) has got the shortest code i.e. '00'. Similarly, the character 'e' has been assigned a shorter code i.e. 100. Obviously, the least frequently occurring characters like l & f have been assigned the longest codes 1101000 and 1101001 respectively.

Let us now compute the total space occupied by the text using Huffman coding. The computed bits per character are given in table below:

Character	Frequency	Code	No of Bits	Character	Frequency	Code	No of Bits
Б (space)	18	00	36(18*2)	l	1	1101000	7
O	2	010	27	f	1	1101000	7
A	4	0110	16	i	2	110101	12
W	3	01110	15	.	2	110110	12
G	3	01111	15	,	2	110111	12
E	11	100	33	y	4	11100	20
N	7	1010	28	u	4	11101	20
R	7	1011	28	h	8	1111	32
T	7	1100	28				

Total bits = 348

It may be noted that in ASCII coding scheme, we would have used $8 \times 93 = 744$ bits. Thus, we have saved 396 bits ($744 - 348$) of storage space in this piece of text comprising of only 93 characters.

Therefore percentage of space saved = $396 / 744 * 100 = 53.2\%$

7.7 DYNAMIC DICTIONARY CODING

Dynamic dictionary coding is a common data compression technique. In this method, the input text is scanned from the first word in such a manner that:

- The words are inserted into an empty dictionary in a search/insert fashion, i.e., the word is searched in the dictionary, and if not found it is inserted in the dictionary and also written on the output.
- If the word is found in the dictionary then its position from the dictionary is written on the output instead of the word.

Example: Input Text: “When you are on the left—you are on the right and when you are on the right—you are on the wrong”.

The dynamic dictionary obtained from above input text is given below:

The dynamic dictionary

Word	Position
When	1
you	2
are	3
on	4
the	5
left	6
right	7
and	8

Output text: When you are on the left 2 3 4 5 right and 1 2 3 4 5 7 2 3 4 5 wrong.

Example 16. Write a program that reads text into an array of strings and compresses the data using dynamic dictionary coding technique.

Solution: An array of strings to store 50 words of 20 characters each would be used to represent the text. The required program is given below:

```
/* This program implements Dynamic Dictionary Decoding */
# include <stdio.h>
main()
{
    char text[50][20];
    char Dictionary[30][20];
    int i,j, flag, wordCount, DictCount;
    i=-1;
    printf ("\n Enter the text terminated by a ### : \n") ;
    do
    {
        i++;
        scanf ("%s", text[i]);
    }
    while (strcmp (text [i], "###"));
    wordCount = --i;
    strcpy (Dictionary[0], text[0]);
    DictCount =0;          /* The Dictionary gets the first word */
    printf ("\n The Text is : %s ",text[0]); /* print the first word */
    for (i = 1;i <= wordCount; i++)
    {
        flag= 0;
```



```

for ( j = 0; j <= DictCount; j++)
{
    if (!strcmp (text[i], Dictionary[j]))
    {
        printf ("%d ", j+1);
        flag = 1 ;
        break;
    }
}
/* End of j Loop */
if (flag == 0)
{
    DictCount ++;
    strcpy (Dictionary[DictCount], text[i]);
    printf ("%s ", text[i]);
}
}
/* End i Loop */
}

```

The sample output of the program is given below:

```

Enter the text terminated by a ### :
when you are on the left you are on the right and when you are on the right you
are on the wrong ###
The Text is : when you are on the left 2 3 4 5 right and 1 2 3 4 5 7 2 3 4 5 wr
ong _

```

EXERCISES

1. What is the need of tree data structure? Explain with the help of examples. List out the areas in which this data structure can be applied extensively.
2. What is a tree? Discuss why definition of tree is recursive. Why it is said to be non-linear?
3. Define the following terms:

(i) Root	(iii) Leaf nodes
(ii) Empty tree	(iv) Sub-tree
4. Discuss the following terms with suitable examples:

(i) Parent	(vi) Path
(ii) Child	(vii) Depth
(iii) Sibling	(viii) Height
(iv) Internal node	(ix) Degree
(v) Edge	
5. Explain binary tree with the help of examples. Discuss the properties of binary tree that need to be considered.

6. Discuss the concept of full binary tree and complete binary tree. Differentiate between the two types with the help of examples.
7. Discuss various methods of representation of a binary tree along with their advantages and disadvantages.
8. Present an algorithm for creation of a binary tree. Consider a suitable binary tree to be created and discuss how the algorithm can be applied to create the required tree.
9. What do you mean by tree traversal? What kinds of operations are possible on a node of a binary tree? Give an algorithm for inorder traversal of a binary tree. Taking an example, discuss how the binary tree can be traversed using inorder traversal.
10. What do you mean by preorder traversal of a tree? Discuss an algorithm for preorder traversal of a binary tree along with an example.
11. Give an algorithm for postorder traversal of a binary tree. Taking an example, discuss how the binary tree can be traversed using postorder traversal.
12. What are the various types of binary trees?
13. What is an expression tree? Consider the following preorder arithmetic expressions:

(i) $+ * - A B C D$	(iv) $/ * A + B C D$
(ii) $+ A + B C$	(v) $+ * A B / C D$
(iii) $/ - A B C$	(vi) $* A + B / C D$

Draw expression tree for the above arithmetic expressions. Give the inorder, preorder and postorder traversal of the expression trees obtained.
14. What is a binary search tree? What are the conditions applied on a binary tree to obtain a binary search tree? Present an algorithm to construct a binary search tree and discuss it using a suitable example.
15. Discuss various operations that can be applied on a BST. Present an algorithm that searches a key in a BST. Take a suitable binary tree and search for a particular key in the tree by applying the discussed algorithm.
16. Discuss how the insert operation can be performed in a binary tree with the help of an algorithm. Consider a binary tree and insert a particular value in the binary tree.
17. Discuss how the delete operation can be performed in a binary tree. Also, present an algorithm for the same. Consider a binary tree and delete a particular value in the binary tree.
18. Discuss the process of deletion of a node in a binary tree for the following cases by taking suitable examples:
 - (i) The node is a leaf node.
 - (ii) The node has only one child.
 - (iii) The node is an internal node, i.e., has both the children.
19. What are heap trees? Discuss representation of a heap tree. Discuss a minheap tree and a maxheap tree with the help of examples. What are the operations that can be performed on a heap tree?
20. Discuss insertion and deletion of a node into/from a heap tree using examples. Present an algorithm for both the operations.

21. Consider the BST shown in Figure 7.58:
Perform the following operations on the above BST:

- (i) Search a given key, say 7, in the BST.
- (ii) Insert a given key, say 10, in the BST.
- (iii) Insert a given key, say 52, in the BST.
- (iv) Delete a given key, say 11, in the BST.
- (v) Delete a given key, say 10, in the BST.
- (vi) Delete a given key, say 1, in the BST.

Display appropriate messages.

22. Consider the following list of numbers input by the user:

12, 5, 18, 90, 43, 36, 81, 10, 15, 8, 23

- (i) Construct a BST for the above list of non-negative integers.
- (ii) Search a given key, say 43, in the BST created.
- (iii) Search a given key, say 93, in the BST created.
- (iv) Insert a given key, say 48, in the BST created.
- (v) Insert a given key, say 23, in the BST created.
- (vi) Delete 23 from the above BST created.
- (vii) Delete 5 from the above BST created.
- (viii) Delete 12 from the above BST created.

Display appropriate messages.

23. What do you mean by sorting? Discuss heap sort in detail with the help of an example.

24. Consider the maxheap shown in Figure 7.59. Perform the desired operations on the heap to obtain a sorted list of elements. Display the sorted list obtained.

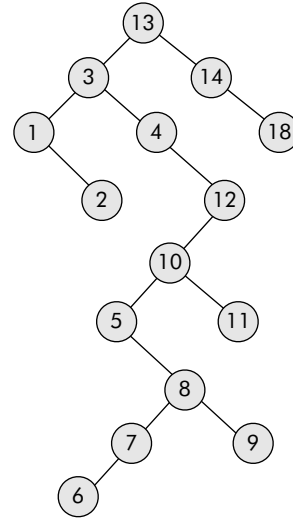


Fig. 7.58

A binary search tree of Q.21

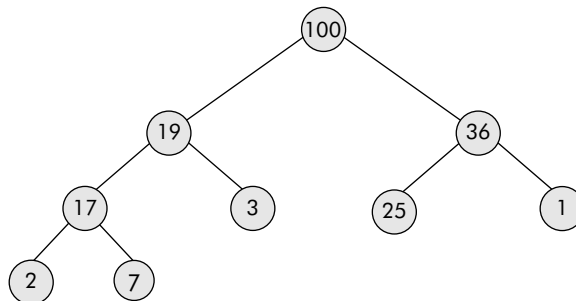


Fig. 7.59

Maxheap of Q.24

Graphs

8 CHAPTER

CHAPTER OUTLINE

- 8.1 Introduction
- 8.2 Graph Terminology
- 8.3 Representation of Graphs
- 8.4 Operations of Graphs
- 8.5 Applications of Graphs

8.1 INTRODUCTION

I want to meet Amitabh Bachchan. Do I know somebody who knows him? Or is there a path of connected people that can lead me to this great man? If yes, then how to represent the people and the path? The problem becomes challenging if the people involved are spread across various cities. For instance, my friend Mukesh from Delhi knows Surender at Indore who in turn knows Wadegoanker residing in Pune. Wadegoanker is a family friend of the great Sachin at Mumbai. The friendship of Sachin and Amitabh is known world over. There is an alternative given by Dr. Asok De. He says that his friend Basu from Kolkata is the first cousin of Aishwarya Rai, the daughter in law of Amitabh.

The various cities involved in this set of related people are connected by roads as shown Figure 8.1.

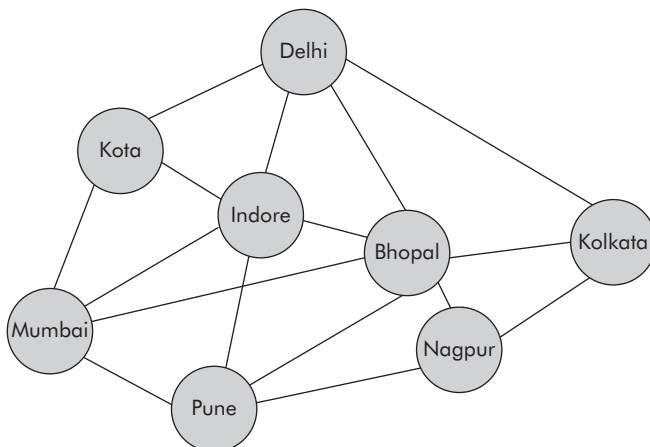


Fig. 8.1 Cities connected with roads

Now the problem is how to represent the data given in Figure 8.1. Arrays are linear by nature and every element in an array must have a unique successor and predecessor except the first and last elements. The arrangement shown in Figure 8.1 is not linear and, therefore, cannot be represented by an array. Though a tree is a non-linear data structure but it has a specific property that a node cannot have more than one parent. The arrangement of Figure 8.1 is not following any hierarchy and, therefore, there is no parent-child relationship.

A close look at the arrangement suggests that it is neither linear nor hierarchical but a network of roads connecting the participating cities. This type of structure is known as a *graph*. In fact, a graph is a collection of nodes and edges. Nodes are connected by edges. Each node contains data. For example, in Figure 8.1, the cities have been shown as nodes and the roads as edges between them. Delhi, Bhopal, Pune, Kolkata, etc. are nodes and the roads connecting them are edges. There are numerous examples where the graph can be used as a data structure. Some of the popular applications are as follows:

- **Model of www:** The model of world wide web (www) can be represented by a collection of graphs (directed) wherein nodes denote the documents, papers, articles, etc. and the edges represent the outgoing hyperlinks between them.
- **Railway system:** The cities and towns of a country are connected through railway lines. Similarly, road atlas can also be represented using graphs.
- **Airlines:** The cities are connected through airlines.
- **Resource allocation graph:** In order to detect and avoid deadlocks, the operating system maintains a resource allocation graph for processes that are active in the system.
- **Electric circuits:** The components are represented as nodes and the wires/connections as edges.

Graph: A graph has two sets: V and E , i.e., $G = \langle V, E \rangle$, where V is the set of *vertices* or *nodes* and E is the set of *edges* or *arcs*. Each element of set E is a pair of elements taken from set V . Consider Figure 8.1 and identify the following:

$V = \{\text{Delhi, Bhopal, Calcutta, Indore...}\}$

$E = \{(\text{Delhi, Kota}), (\text{Bhopal, Nagpur}), (\text{Pune, Mumbai})...\}$

Before proceeding to discuss more on graphs, let us have a look at the basic terminology of graphs discussed in the subsequent section.

8.2 GRAPH TERMINOLOGY

In this book, the following terms related to graphs are used:

Directed graph: A directed graph is a graph $G = \langle V, E \rangle$ with the property that its edges have directions. An edge $E: (v_i, v_j)$ means that there is an arrow whose head is pointing to v_j and the tail to v_i . A directed graph is shown in Figure 8.2(a). For example, (B, A) , (A, D) , (C, D) , etc. are directed edges. A directed graph is also called a *digraph*.

Undirected graph: An undirected graph is a graph $G = \langle V, E \rangle$ with the property that its edges have no directions, i.e., the edges are two ways as shown in Figure 8.2(b). For example, (B, A) , (A, B) , etc. are two way edges. A graph always refers to an undirected graph.

Weighted graph: A weighted graph is a graph $G = \langle V, E \rangle$ with the property that its edges have a number or label associated with it. The number is called the weight of the edge. A weighted graph is shown in Figure 8.2(c). For example, $(B, A) 10$ may mean that the distance of B from A is 10m. A weighted graph can be an undirected graph or a directed graph.

Path: It is a sequence of nodes in which all the intermediate pair nodes between the first and the last nodes are joined by edges as shown in Figure 8.2(d). The path from A to F is shown with the help of dark edges. The *path length* is equal to the number of edges present in a path. The path length of $A-D-E-F$ is 3 as it contains three edges.

However in the case of weighted graphs, the path length is computed as the sum of labels of the intermediate edges. For example, the path length of path A-D-C-E of Figure 8.2(c) is $(11 + 12 + 9)$, i.e., 32.

It may be noted that if no node occurs more than once in a path then the path is called a *simple path*. For example in Figure 8.2(b), A-B-C is a simple path. However, the path A-D-E-C-D is not a simple path.

Cycle: A path that starts and ends on the same node is called a cycle. For example, the path D-E-C-D in Figure 8.2(b) is a cycle.

Connected graph: If there exists a path between every pair of distinct nodes, then the undirected graph is called a connected graph. For example, the graphs shown in Figure 8.3(b), (c) and (d) are connected graphs. However, the graph shown in Figure 8.3(a) is not a connected graph.

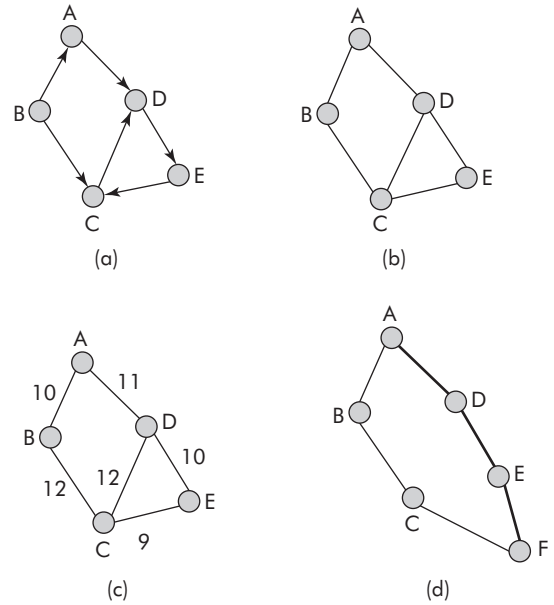


Fig. 8.2 Various kinds of graphs

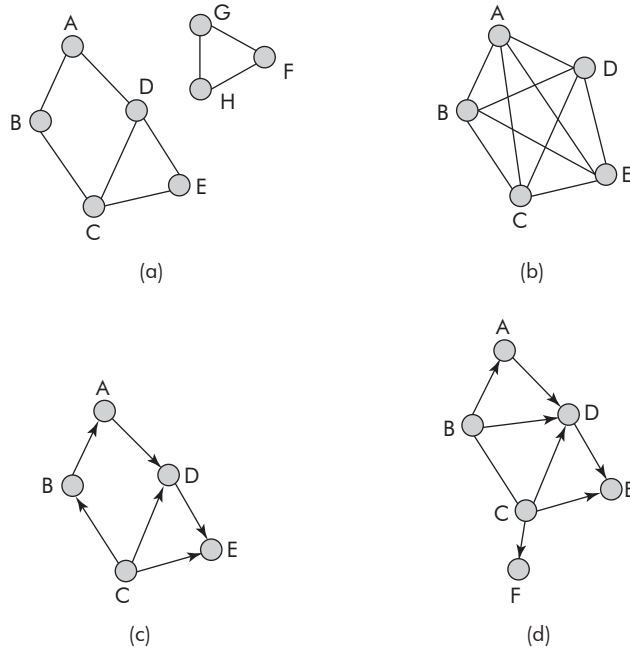


Fig. 8.3 More graphs

Adjacent node: A node Y is called adjacent to X if there exists an edge from X to Y. Y is called successor of X and X predecessor of Y. The set of nodes adjacent to X are called neighbours of X.

Complete graph: A complete graph is a graph $G = \langle V, E \rangle$ with the property that each node of the graph is adjacent to all other nodes of the graph, as shown in Figure 8.3(b).

Acyclic graph: A graph without a cycle is called an acyclic graph as shown in Figure 8.3(c). In fact, a tree is an excellent example of an acyclic graph.

Self loop: If the starting and ending nodes of an edge are same, then the edge is called a self loop, i.e., edge (E, E) is a self loop as shown in Figure 8.4(a).

Parallel edges: If there are multiple edges between the same pair of nodes in a graph, then the edges are called parallel edges. For example, there are two parallel edges between nodes B and C of Figure 8.4(b).

Multigraphs: A graph containing self loop or parallel edges or both is called a multigraph. The graphs shown in Figure 8.4 are multigraphs.

Simple graph: A simple graph is a graph which is free from self loops and parallel edges.

Degree of a node or vertex: The number of edges connected to a node is called its degree. The maximum degree of a node in a graph is called the degree of the graph. For example, the degree of node A in Figure 8.3(a) is 2 and that of Delhi in Figure 8.1 is 4. The degree of graph in Figure 8.1 is 6.

However, in case of *directed graphs*, there are two degrees of a node—*indegree* and *outdegree*. The *indegree* is defined as the number of edges incident upon the node and *outdegree* as the number of edges radiating out of the node.

Consider Figure 8.3(d), the *indegree* and *outdegree* of C is 0 and 4, respectively. Similarly, the *indegree* and *outdegree* of D is 3 and 1, respectively.

Pendent node: A pendent node in a graph is a node whose *indegree* is 1 and *outdegree* is 0. The node F in Figure 8.3(d) is a pendent node. In fact, in a tree all leaf nodes are necessarily pendent nodes.

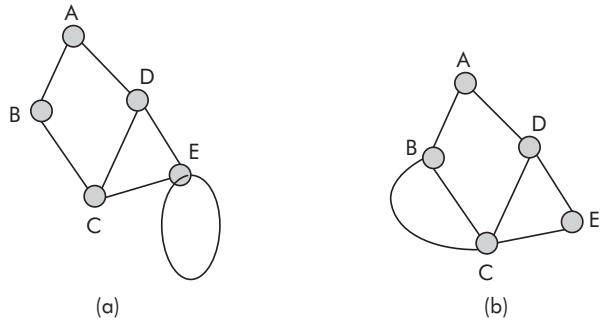


Fig. 8.4 Multigraphs

8.3 REPRESENTATION OF GRAPHS

A graph can be represented in many ways and the most popular methods are given below:

- Array-based representation
- Linked representation
- Set representation

These representations are discussed in detail in the sections that follow.

8.3.1 Array-based Representation of Graphs

A graph can be comfortably represented using an adjacency matrix implemented through a two-dimensional array. **Adjacency matrix** is discussed in detail below.

A graph of order N, i.e., having N nodes or vertices can be comfortably modelled as a square matrix $N \times N$ wherein the rows and the column numbers represent the vertices as shown in Figure 8.5.

A cell at the cross-section of vertices can have only Boolean values, 0 or 1. An entry (v_i, v_j) is 1 only when there exists an edge between v_i and v_j , otherwise a 0 indicates an absence of edge between the two.

The adjacency matrices for graphs of Figures 8.3(a–d) are given in Figure 8.6. From Figure 8.6(a), it may be observed that a row provides list of successors of a node and the column provides list of predecessors. The successors of B are A and C (see Figure 8.6(a)). An empty column under B suggests that there are no predecessors of B.

It may be noted that in the case of weighted graph (Figure 8.6 (c)), the edge weight has been stored instead of 1 to mark the presence of an edge. Some important properties of adjacency matrices are as follows:

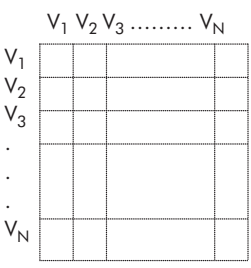


Fig. 8.5 A matrix representing a graph

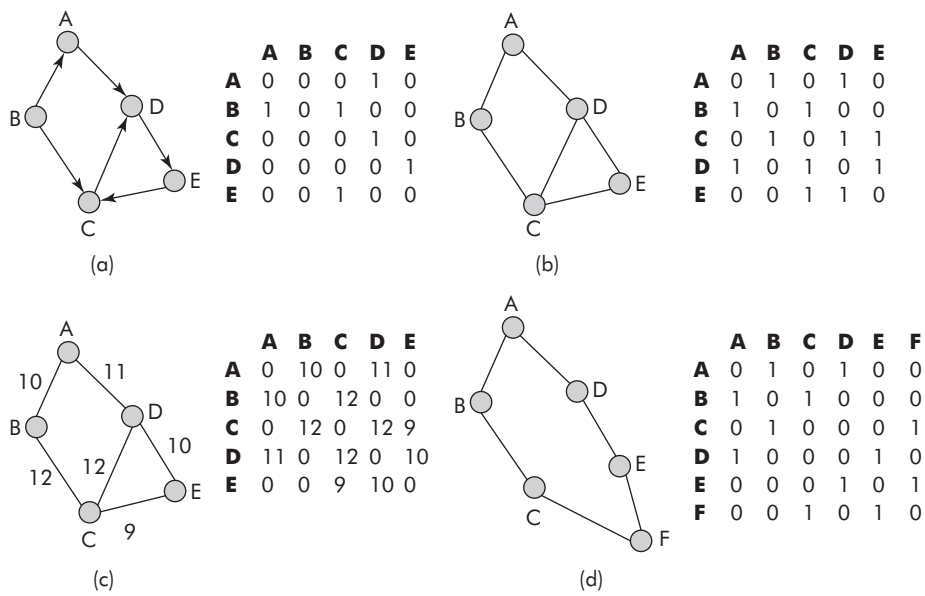


Fig. 8.6 The equivalent adjacency matrices for graphs

Property 1: It may further be noted that the adjacency matrix is *symmetric* for an undirected graph. This can be easily proved that an edge (v_i, v_j) of an undirected graph gets an entry into an adjacency matrix A at two locations: A_{ij} and A_{ji} . Thus in the matrix A for all i, j , $A_{ij} = A_{ji}$, which is the property of a symmetric matrix. Therefore, the following relation also holds good:

$$A = A^T$$

where A^T is the transpose of adjacency matrix A.

From Figure 8.6(a), it can be confirmed that in case of directed graphs, the adjacency matrix is not symmetric and therefore the following relation holds true.

$$A \neq A^T$$

However, for simple graphs, the diagonal elements are always 0.

Property 2: For a directed graph G with adjacency matrix A , the diagonal of matrix $A \times A^T$ gives the out degrees of the corresponding vertices.

Consider the graph and its adjacency matrix A given in Figure 8.7.

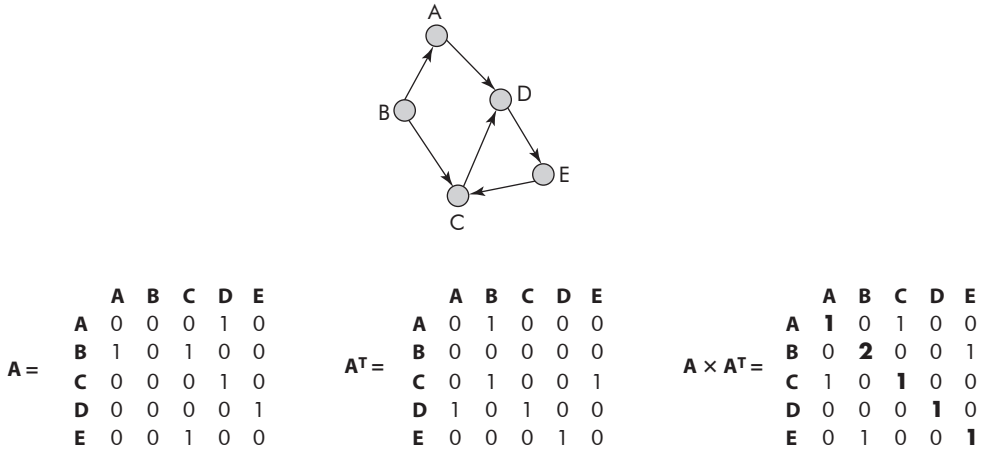


Fig. 8.7 The out degree of vertices denoted by diagonal elements of $A \times A^T$

It may be noted that the diagonal elements of $A \times A^T$ are $\{1, 2, 1, 1, 1\}$, which are in fact the out-degrees of vertices A, B, C, D and E, respectively.

Similarly, for a directed graph G with adjacency matrix A , the diagonal of matrix $A^T \times A$ gives the in degrees of the corresponding vertices. Consider the graph and its adjacency matrix A given in Figure 8.8.

It may be noted that the diagonal elements of $A^T \times A$ are $\{1, 0, 2, 2, 1\}$, which are in fact the in degrees of vertices A, B, C, D and E, respectively.

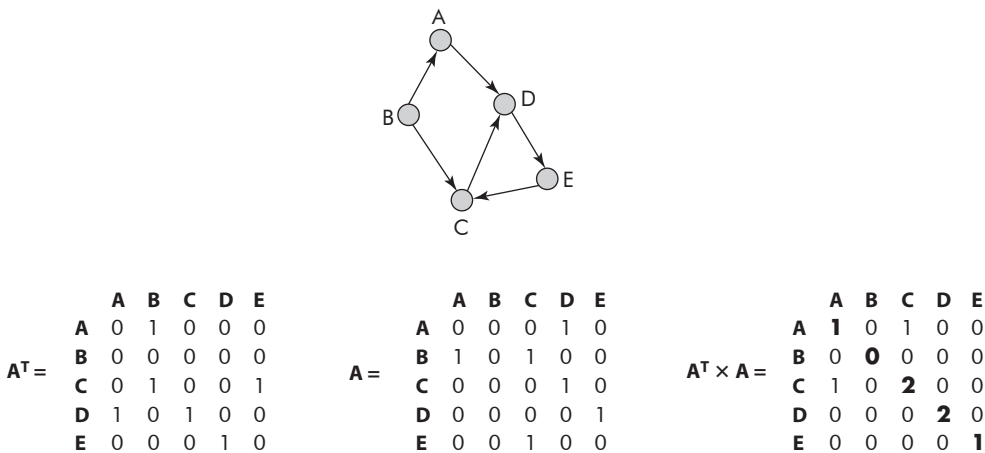


Fig. 8.8 The in degree of vertices denoted by diagonal elements of $A^T \times A$

Property 3: For a directed graph G with adjacency matrix A , the element A_{ij}^k of matrix A^k (k th power of A) gives the number of paths of length k from vertex v_i to v_j . This means that an element A_{ij} of A^2 will denote the number of paths of length 2 from vertex v_i to v_j . Consider Figure 8.9. A^2 gives the path length of 2 from a vertex v_i to v_j . For example, there is one path of length 2 from C to A (i.e., $C-B-A$). Similarly, A^3 gives the path length of 3 from a vertex v_i to v_j . For example, there is one path of length 3 from E to D (i.e., $E-C-B-D$).

The major drawback of adjacency matrix is that it requires $N \times N$ space whereas most of the entries in the matrix are 0, i.e., the matrix is sparse.

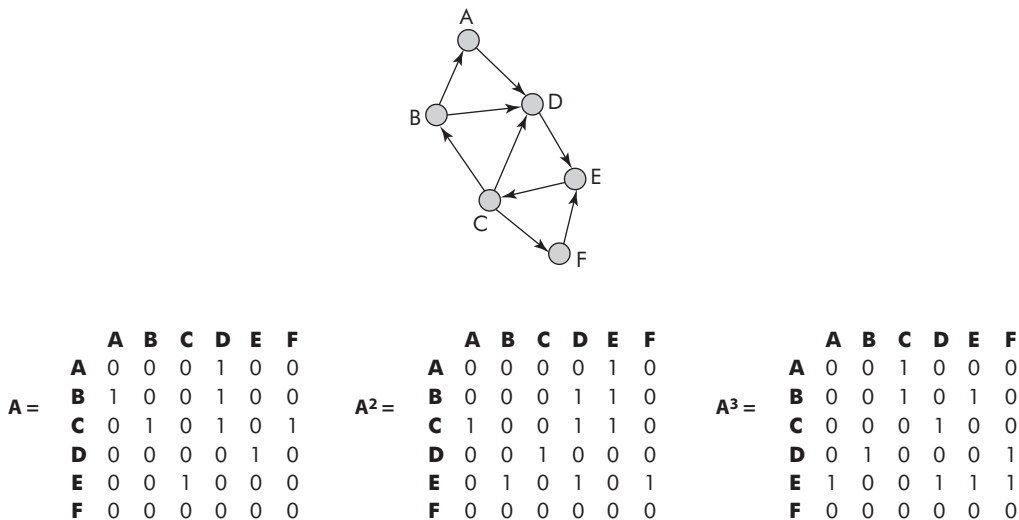


Fig. 8.9 Number of paths from a vertex to another vertex

8.3.2 Linked Representation of a Graph

In order to save the space, an array of linked lists is used to represent the adjacent nodes and the data structure is called adjacency list. For example, the adjacency list for a directed graph is shown in Figure 8.10. Similarly, the adjacency list for an undirected graph is shown in Figure 8.11.

The above representation involves a mix of array and linked lists, i.e., it is an array of linked lists. For pure linked representation, the array part can be replaced by another linked list of head nodes as shown in Figure 8.12. The head node contains three parts: data, pointer to adjacent node and a pointer to next head node as shown in Figure 8.13.

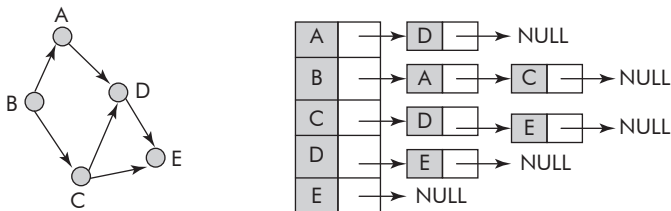


Fig. 8.10 Adjacency list for a directed graph

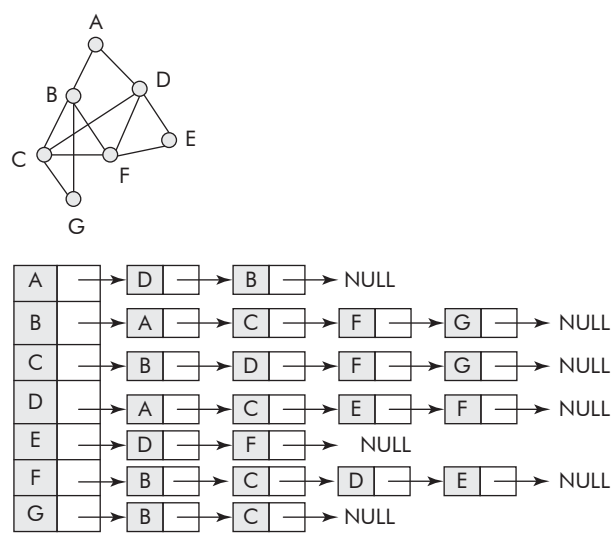


Fig. 8.11 Adjacency list for an undirected graph

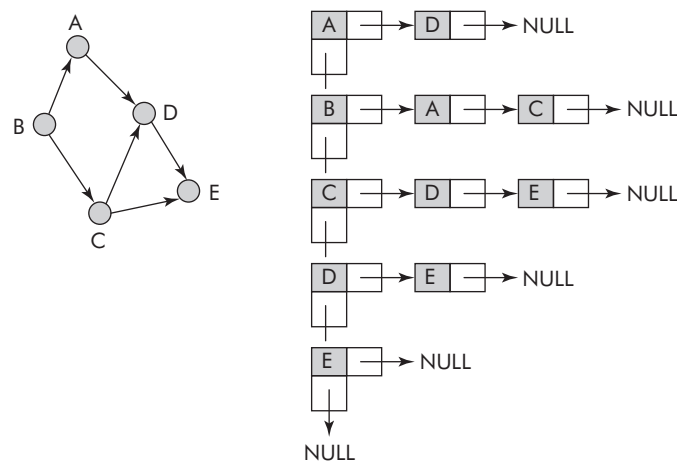


Fig. 8.12 The linked representation of an adjacency list of a directed graph

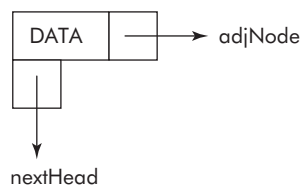


Fig. 8.13 The structure of a head node

Note: If in a graph G , there are N vertices and M edges, the adjacency matrix has space complexity $O(N^2)$ and that of adjacency list is $O(M)$.

8.3.3 Set Representation of Graphs

This representation follows the definition of the graph itself in the sense that it maintains two sets: V - 'Set of Vertices' and E - 'Set of Edges'. The set representation for various graphs is given in Figure 8.14.

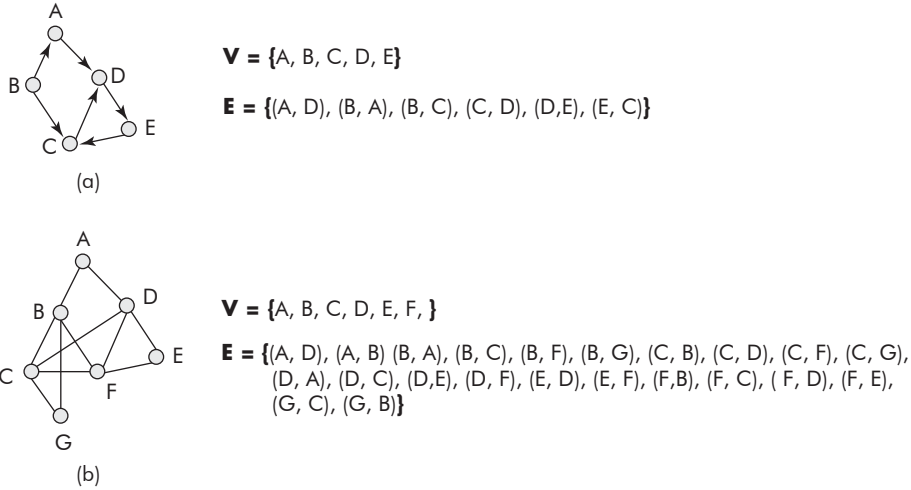


Fig. 8.14 The set representation of graphs

In case of weighted graphs, the weight is also included with the edge information making it a 3-tuple, i.e., (w, v_i, v_j) where w is the weight of edge between vertices v_i and v_j . The set representation for a weighted graph is given in Figure 8.15.

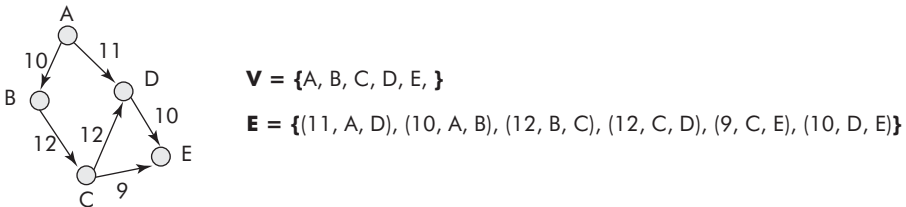


Fig. 8.15 The set representation of a weighted graph

This set representation is most efficient as far as storage is concerned but is not suitable for operations that can be defined on graphs. A discussion on operations on graphs is given in the section 8.4.

8.4 OPERATIONS OF GRAPHS

The following operations are defined on graphs:

- (1) **Insertion:** There are two major components of a graph—vertex and edge. Therefore, a node or an edge or both can be inserted into an existing graph.

- (2) **Deletion:** Similarly, a node or an edge or both can be deleted from an existing graph.
- (3) **Traversal:** A graph may be traversed for many purposes—to search a path, to search a goal, to establish a shortest path between two given nodes, etc.

8.4.1 Insertion Operation

The insertion of a vertex and its associated edge with other vertices in an adjacency matrix involves the following operations:

- (1) Add a row for the new vertex.
- (2) Add a column for the new vertex.
- (3) Make appropriate entries into the rows and columns of the adjacency matrix for the new edges.

Consider the graph given in Figure 8.16.

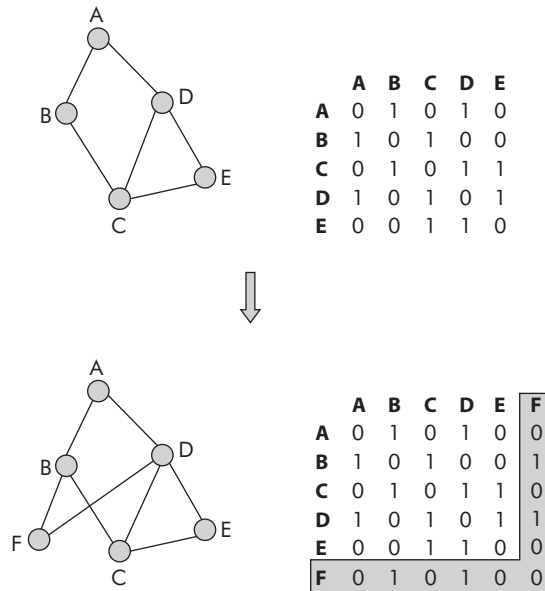


Fig. 8.16 Insertion of a vertex into an undirected graph

It may be noted that a vertex called F has been added to the graph. It has two edges (F, B) and (F, D). The adjacency matrix has been accordingly modified to have a new row and a new column for the incoming vertex. Appropriate entries for the edges have been made in the adjacency matrix, i.e., the cells (F, B), (F, D), (B, F) and (D, F) have been set equal to 1.

An algorithm for insertion of a vertex is given below:

```
/* This algorithm uses a two-dimensional matrix adjMat[][] to store the
adjacency matrix. A new vertex called verTex is added to the matrix by adding
an additional empty row and an additional empty column. */
```

```
Algorithm addVertex ()
```

```
{
    lastRow = lastRow + 1;
```

```

    lastCol = lastCol + 1;
    adjMat[lastRow][0] = verTex;
    adjMat[0][lastCol] = verTex;
    Set all elements of last row = '0';
    Set all elements of last Col = '0';
}

```

An algorithm for insertion of an edge into an undirected graph is given below:

```

/* This algorithm uses a two-dimensional matrix adjMat[][] to store the adjacency matrix. A new edge (v1, v2) is added to the matrix by adding its entry (i.e., '1') in the row and column corresponding to v1 and v2, respectively. */

```

Algorithm addEdge()

```

{
    Find row corresponding to v1, i.e., rowV1;
    Find col corresponding to v2, i.e., colV2;
    adjMat[rowV1][colV2] = '1';          /* make symmetric entries */
    adjMat[colV2][rowV1] = '1';
}

```

Example 1: Write a program that implements an adjacency matrix wherein it adds a vertex and its associated edges. The final adjacency matrix is displayed.

Solution: Two functions called addVertex() and addEdge() would be used for insertion of a new vertex and its associated edges, respectively. Another function called dispAdMat() would display the adjacency matrix.

The required program is given below:

```

/* This program implements an adjacency matrix */
#include <stdio.h>
#include <conio.h>

void addVertex (char adjMat[7][7], int numV, char verTex);
void addEdge (char adjMat[7][7], char v1, char v2, int numV);
void dispAdMat(char adjMat[7][7], int numV);

void main()
{
    /* Adjacency matrix of Figure 8.17 */
    char adjMat[7][7] = { '-', 'A', 'B', 'C', 'D', 'E', ' ',
        'A', '0', '1', '0', '1', '0', ' ',
        'B', '1', '0', '1', '0', '0', ' ',
        'C', '0', '1', '0', '1', '1', ' ',
        'D', '1', '0', '1', '0', '1', ' ',
        'E', '0', '0', '1', '1', '0', ' ',
        ' ', ' ', ' ', ' ', ' ', ' ', ' ' };

    int numVertex = 5;
    char newVertex, v1, v2;
    char choice;
    dispAdMat (adjMat, numVertex);
}

```



```

printf ("\n Enter the vertex to be added");
fflush (stdin);
newVertex = getchar();
numVertex++;
addVertex (adjMat, numVertex, newVertex);
do
{
    fflush(stdin);
    printf ("\n Enter Edge: v1 - v2");
    scanf ("%c %c", &v1, &v2);
    addEdge (adjMat, v1,v2,numVertex);
    dispAdMat (adjMat, numVertex);
    fflush (stdin);
    printf ("\n do you want to add another edge Y/N");
    choice = getchar();
}
while ((choice != 'N') && (choice != 'n'));
}

void addVertex (char adjMat[7][7], int numV, char verTex)
{ int i;
  adjMat[numV][0] = verTex;
  adjMat[0][numV] = verTex;
  for (i = 1; i<= numV; i++)
  {
    adjMat[numV][i] = '0';
    adjMat[i][numV] = '0';
  }
}

void addEdge (char adjMat[7][7], char v1, char v2, int numV)
{ int i, j, k;
  i = 0;
  for (j = 1; j <= numV; j++)
  {
    if (adjMat[i][j] == v1)
    {
      for (k = 0; k <= numV; k++)
      {
        if (adjMat [k][0] == v2)
        {
          adjMat[k][j] = '1';
          adjMat[j][k] = '1'; break; /* making symmetric entries */
        }
      }
    }
  }
}
}
}
}

```

```

void dispAdMat(char adjMat[7][7], int numV)
{
    int i,j;
    printf ("\nThe adj Mat is--\n");
    for (i=0; i<= numV; i++)
    {
        for (j=0; j<=numV; j++)
        {
            printf ("%c ", adjMat[i][j]);
        }
        printf ("\n");
    }
    printf ("\n\n Enter any key to continue");
    getch();
}

```

The above program has been tested for data given in Figure 8.16. The screenshots of the result are given in Figure 8.17.

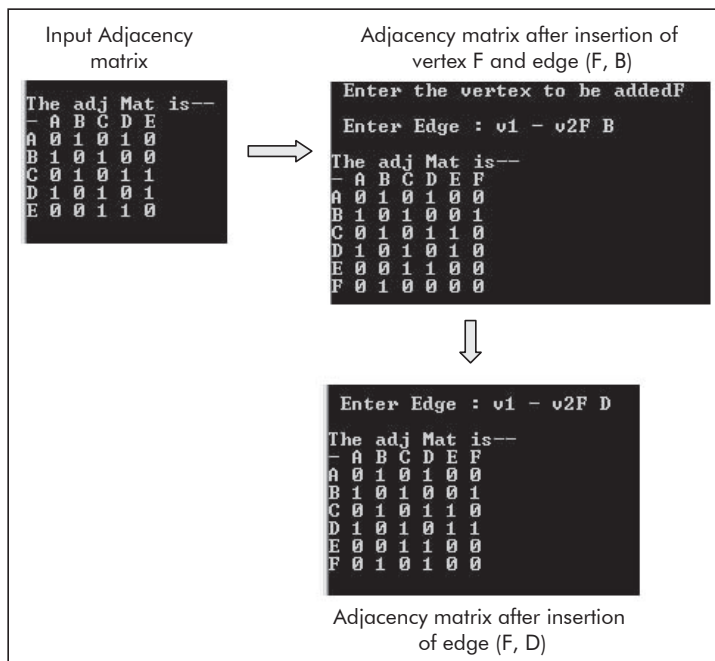


Fig. 8.17 The adjacency matrix before and after insertion of vertex F

Note: In case of a directed graph, only one entry of the directed edge in the adjacency matrix needs to be made. An algorithm for insertion of a vertex in a directed graph is given below:

```
/* This algorithm uses a two-dimensional matrix adjMat[][] to store the
adjacency matrix. A new vertex called verTex is added to the matrix by
adding an additional empty row and an additional empty column. */
```

Algorithm addVertex ()

```
{
    lastRow = lastRow + 1;
    lastCol = lastCol + 1;
    adjMat[lastRow][0] = verTex;
    adjMat[0][lastCol] = verTex;
    Set all elements of last row = '0';
    Set all elements of last Col = '0';
}
```

An algorithm for insertion of an edge into a directed graph is given below:

```
/* This algorithm uses a two-dimensional matrix adjMat[][] to store the
adjacency matrix. A new edge (v1, v2) is added to the matrix by adding its
entry (i.e., '1') in the row corresponding to v1. */
```

Algorithm addEdge ()

```
{
    Find row corresponding to v1, i.e., rowV1;
    Find col corresponding to v2, i.e., colV2;
    adjMat[rowV1][colV2] = '1';
}
```

Accordingly, the function addEdge() needs to be modified so that it makes the adjacency matrix asymmetric. The modified function is given below:

```
/* The modified function for addition of edges in directed graphs */
```

```
void addEdge (char adjMat[7][7], char v1, char v2, int numV)
```

```
{ int i,j, k;
  j=0;
  for (i=1; i<=numV; i++)
  {
    if (adjMat[i][j] == v1)
    {
      for (k=1; k<= numV; k++)
      {
        if (adjMat [0][k] ==v2)
        {
          adjMat[i][k] = '1'; break;
        }
      }
    }
  }
}
```

8.4.2 Deletion Operation

The deletion of a vertex and its associated edges from an adjacency matrix, for directed and undirected graph, involves the following operations:

- (1) Deleting the row corresponding to the vertex.
- (2) Deleting the col corresponding to the vertex.

The algorithm is straight forward and given below:

Algorithm delVertex (verTex)

```
{
    find the row corresponding to verTex and set all its elements = '0';
    find the col corresponding to verTex and set all its elements = '0';
}
```

The deletion of an edge from a graph requires different treatment for undirected and directed graphs. Both the cases of deletion of edges are given below:

An algorithm for deletion of an edge from an undirected graph is given below:

/ This algorithm uses a two-dimensional matrix adjMat[][] to store the adjacency matrix. A new edge (v1, v2) is deleted from the matrix by deleting its entry (i.e., '0') from the row and column corresponding to v1 and v2, respectively. */*

```
Algorithm delEdge ()          /* undirected graph */
{
    Find row corresponding to v1, i.e., rowV1;
    Find col corresponding to v2, i.e., colV2;
    adjMat[rowV1][ColV2] = '0';    /* make symmetric entries */
    adjMat[colV2][rowV1] = '0';
}
```

An algorithm for deletion of an edge from a directed graph is given below:

/ The algorithm uses a two-dimensional matrix adjMat[][] to store the adjacency matrix. An edge (v1, v2) is deleted from the matrix by deleting its entry (i.e., '0') from the row corresponding to v1 */*

```
Algorithm delEdge()          /* directed graph */
{
    Find row corresponding to v1, i.e., rowV1;
    Find col corresponding to v2, i.e., colV2;
    adjMat[rowV1][ColV2] = '0';
}
```

Example 2: Write a program that deletes a vertex and its associated edges from an adjacency matrix. The final adjacency matrix is displayed.

Solution: A function called delVertex() would be used for deletion of a vertex and its associated edges from an adjacency matrix. Another function called dispAdMat() would display the adjacency matrix.

The required program is given below:

```

/* This program deletes a vertex from an adjacency list */

#include <stdio.h>
#include <conio.h>

void delVertex (char adjMat[7][7], int numV, char verTex);
void dispAdMat (char adjMat[7][7], int numV);

void main()
{
    char adjMat[7][7] = { '-', 'A', 'B', 'C', 'D', 'E', ' ',
                          'A', '0', '1', '0', '1', '0', ' ',
                          'B', '1', '0', '1', '0', '0', ' ',
                          'C', '0', '1', '0', '1', '1', ' ',
                          'D', '1', '0', '1', '0', '1', ' ',
                          'E', '0', '0', '1', '1', '0', ' ',
                          ' ', ' ', ' ', ' ', ' ', ' ', ' '};

    int numVertex = 5;
    char Vertex;
    dispAdMat (adjMat, numVertex);
    printf ("\n Enter the vertex to be deleted");
    fflush (stdin);
    Vertex=getchar();
    delVertex (adjMat, numVertex, Vertex);
    dispAdMat (adjMat, numVertex);
}

void delVertex (char adjMat[7][7], int numV, char verTex)
{
    int i, j, k;
    j = 0;
    for (i = 1; i <= numV; i++)
    {
        if (adjMat[i][j] == verTex)
        {
            adjMat[i][0] = '-';
            for (k = 1; k <= numV; k++)
            {
                adjMat[i][k] = '0';
            }
        }
    }
}

i = 0;
for (j = 1; j <= numV; j++)
{
    if (adjMat[i][j] == verTex)
    {
        adjMat[0][j] = '-';
        for (k = 1; k <= numV; k++)
    
```

```

        {
            adjMat[k][j] = '0';
        }
    }
}
}

void dispAdMat(char adjMat[7][7], int numV)
{
    int i, j;
    printf ("\nThe adj Mat is--\n");
    for (i = 0; i <= numV; i++)
    {
        for (j = 0; j <= numV; j++)
        {
            printf ("%c ", adjMat[i][j]);
        }
        printf ("\n");
    }
    printf ("\n\n Enter any key to continue");
    getch();
}

```

The screenshots of the output of the program are shown in Figure 8.18.

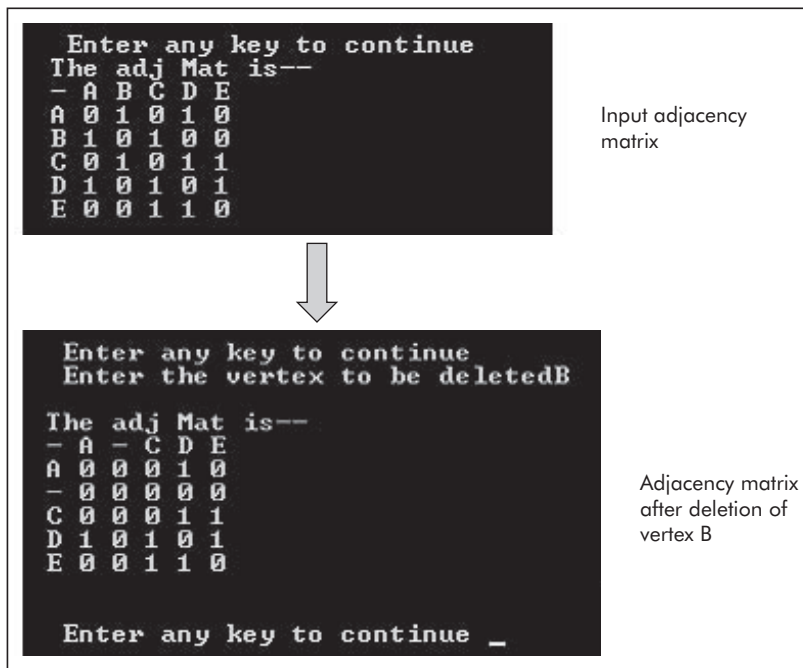


Fig. 8.18 The adjacency matrix before and after deletion of vertex B

Example 3: Write a program that deletes an edge from an adjacency matrix of an undirected graph. The final adjacency matrix is displayed.

Solution: A function called `delEdge()` would be used for deletion of an edge from an adjacency matrix. Another function called `dispAdMat()` would display the adjacency matrix.

The required program is given below:

```
/* This program deletes an edge of an undirected graph from an adjacency
matrix */

#include <stdio.h>
#include <conio.h>
void delEdge (char adjMat[7][7], char v1, char v2, int numV);
void dispAdMat(char adjMat[7][7], int numV);

void main()
{
    char adjMat[7][7] = { '-', 'A', 'B', 'C', 'D', 'E', ' ',
        'A', '0', '1', '0', '1', '0', ' ',
        'B', '1', '0', '1', '0', '0', ' ',
        'C', '0', '1', '0', '1', '1', ' ',
        'D', '1', '0', '1', '0', '1', ' ',
        'E', '0', '0', '1', '1', '0', ' ',
        ' ', ' ', ' ', ' ', ' ', ' ', ' '};

    int numVertex = 5;
    char v1, v2;
    dispAdMat (adjMat, numVertex);
    printf ("\n Enter the edge to be deleted");
    fflush (stdin);
    printf ("\n Enter Edge : v1 - v2");
    scanf ("%c %c", &v1, &v2);
    delEdge (adjMat, v1,v2,numVertex);
    dispAdMat (adjMat, numVertex);
}

void delEdge (char adjMat[7][7], char v1, char v2, int numV)
{ int i, j, k;
  i = 0;
  for (j = 1; j <= numV; j++)
  {
      if (adjMat[i][j] == v1)
      {
          for (k = 0; k <= numV; k++)
          {
              if (adjMat [k][0] == v2)
              {
                  adjMat[k][j] = '0';
                  adjMat[j][k] = '0'; break; /* making symmetric entries */
              }
          }
      }
  }
}
```

```

    }
  }
}

void dispAdMat(char adjMat[7][7], int numV)
{
  int i,j;
  printf ("\nThe adj Mat is--\n");
  for (i=0; i<= numV; i++)
  {
    for (j=0; j<=numV; j++)
    {
      printf ("%c ", adjMat[i][j]);
    }
    printf ("\n");
  }
  printf ("\n\n Enter any key to continue");
  getch();
}

```

The screenshots of the output of the program are given in Figure 8.19.

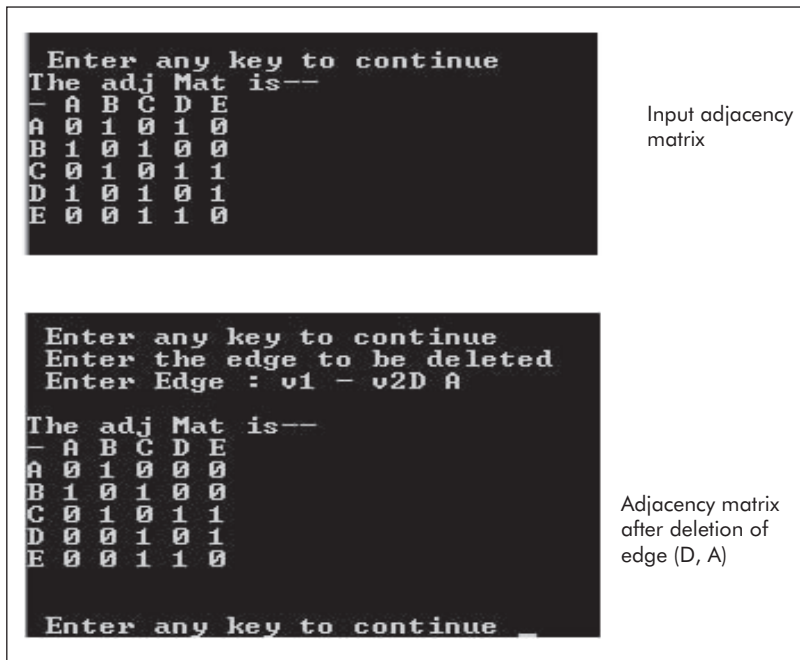


Fig. 8.19 The adjacency matrix before and after deletion of edge (D, A)

The function `delEdge ()` needs to be modified for directed graphs so that it does not make the adjacency matrix as symmetric. The modified function is given below:

```
/* The modified function for deletion of edges of directed graphs */
void delEdge (char adjMat[7][7], char v1, char v2, int numV)
{ int i, j, k;
  j = 0;
  for (i = 1; i <= numV; i++)
  {
    if (adjMat[i][j] == v1)
    {
      for (k = 1; k <= numV; k++)
      {
        if (adjMat [i][k] == v2)
        {
          adjMat[i][k] = '0';break;
        }
      }
    }
  }
}
```

8.4.3 Traversal of a Graph

Travelling a graph means that one visits the vertices of a graph at least once. The purpose of the travel depends upon the information stored in the graph. For example on www, one may be interested to search a particular document on the Internet while on a resource allocation graph the operating system may search for deadlocked processes.

In fact, search has been used as an instrument to find solutions. The epic 'Ramayana' uses the search of goddess 'Sita' to find and settle the problem of 'Demons' of that yuga including 'Ravana'. Nowadays, the search of graphs and trees is very widely used in artificial intelligence.

A graph can be traversed in many ways but there are two popular methods which are very widely used for searching graphs; they are: Depth First Search (DFS) and Breadth First Search (BFS). A detailed discussion on both is given in the subsequent sections.

8.4.3.1 Depth First Search (DFS) In this method, the travel starts from a vertex then carries on to its successors, i.e., follows its outgoing edges. Within successors, it travels their successor and so on. Thus, this travel is same as inorder travel of a tree and, therefore, the search goes deeper into the search space till no successor is found. Once the search space of a vertex is exhausted, the travel for next vertex starts. This carries on till all the vertices have been visited. The only problem is that the travel may end up in a cycle. For example, in a graph, there is a possibility that the successor of a successor may be the vertex itself and the situation may end up in an endless travel, i.e., a cycle.

In order to avoid cycles, we may maintain two queues: *notVisited* and *Visited*. The vertices that have already been visited would be placed on *Visited*. When the successors of a vertex are generated, they are placed on *notVisited* only when they are not already present on both *Visited* and *notVisited* queues thereby reducing the possibility of a cycle.

Consider the graph shown in Figure 8.20. The trace of DFS travel of the given graph is given in Figures 8.20 and 8.21, through entries into the two queues *Visited* and *notVisited*.

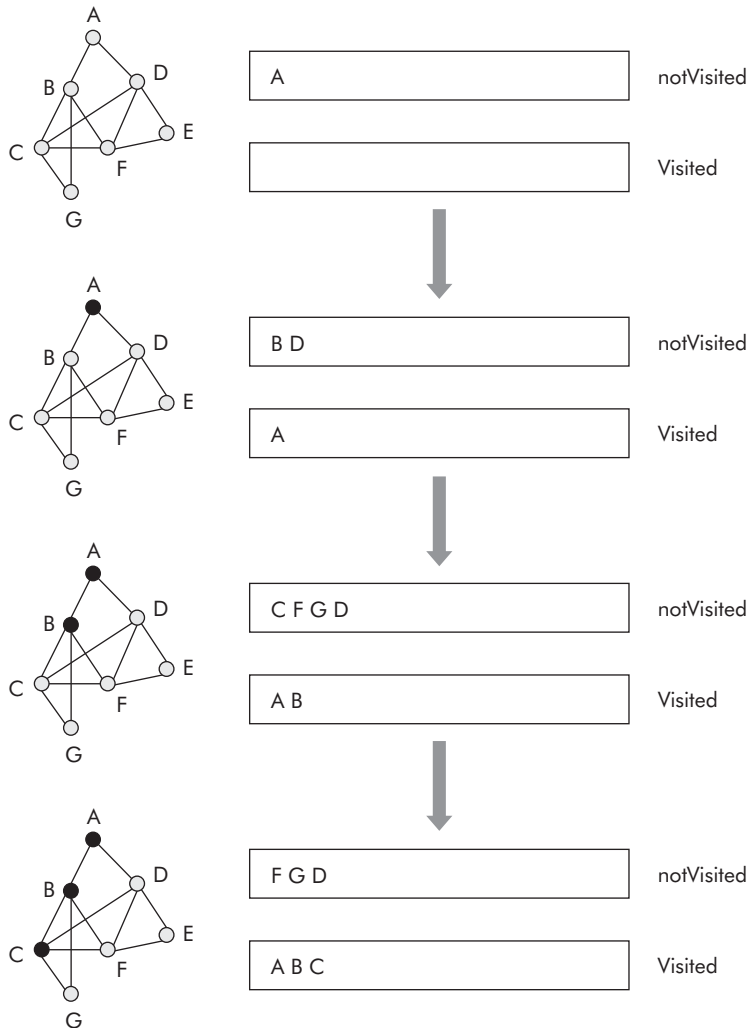


Fig. 8.20 The trace of DFS travel of a graph

The algorithm for DFS travel is given below:

Algorithm DFS (firstVertex)

```
{
  add firstVertex on notVisited;
  place NULL on Visited;                               /* initially Visited is empty */
  while (notVisited != NULL)
  {
    remove vertex from notVisited;
    generate successors of vertex ;
  }
}
```

```
for each successor of vertex do
{
    if (successor is not present on Visited AND successor is not present
    on notVisited) Then add vertex on front of notVisited;
}
if (vertex is not present on Visited) Then add vertex on Visited;
}
```

The final visited queue contains the nodes visited during DFS travel, i.e., A B C F E G D.

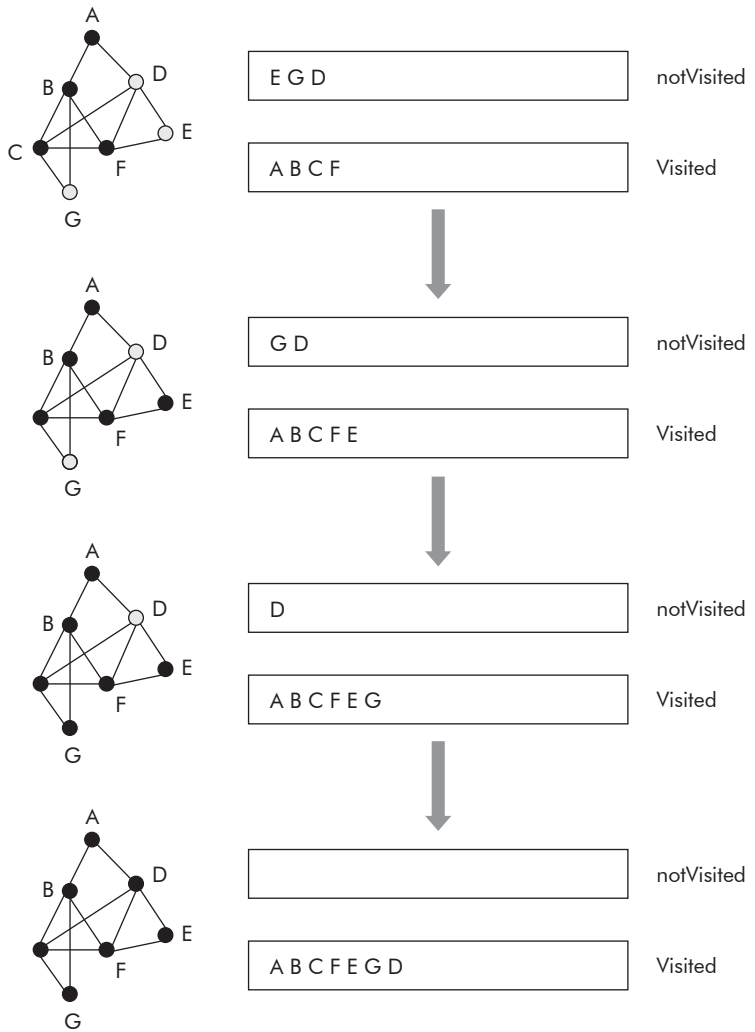


Fig. 8.21 The trace of DFS travel of a graph

Example 4: Write a program that travels a given graph using DFS strategy.

Solution: We would use the following functions:

- (1) **void addNotVisited (char Q[], int *first, char vertex); :**
This function adds a vertex on notVisited queue in front of the queue.
- (2) **void addVisited (char Q[], int *last, char vertex);**
This function adds a vertex on Visited queue.
- (3) **char removeNode (char Q[], int *first);**
This function removes a vertex from notVisited queue.
- (4) **int findPos (char adjMat[8][8], char vertex);**
This function finds the position of a vertex in adjacency matrix, i.e., adjMat.
- (5) **void dispAdMat(char adjMat[8][8], int numV);**
This function displays the contents of adjacency matrix, i.e., adjMat.
- (6) **int ifPresent (char Q[], int last, char vertex);**
This function checks whether a vertex is present on a queue or not.
- (7) **void dispVisited (char Q[], int last);**
This function displays the contents of Visited queue.

We would use the adjacency matrix of graph shown in Figures 8.20 and 8.21. The required program is given below:

```
/* This program travels a graph using DFS strategy */
#include <stdio.h>
#include <conio.h>

void addNotVisited (char Q[], int *first, char vertex);
void addVisited (char Q[], int *last, char vertex);
char removeNode (char Q[], int *first);
int findPos (char adjMat[8][8], char vertex);
void dispAdMat(char adjMat[8][8], int numV);
int ifPresent (char Q[], int last, char vertex);
void dispVisited (char Q[], int last);
void main()
{ int i,j, first, last, pos;
char adjMat[8][8] = {   '-', 'A', 'B', 'C', 'D', 'E', 'F', 'G',
    'A', '0', '1', '0', '1', '0', '0', '0',
    'B', '1', '0', '1', '0', '0', '1', '1',
    'C', '0', '1', '0', '1', '0', '1', '1',
    'D', '1', '0', '1', '0', '1', '1', '0',
    'E', '0', '0', '0', '1', '0', '1', '0',
    'F', '0', '1', '1', '1', '1', '0', '0',
    'G', '0', '1', '1', '0', '0', '0', '0'};

int numVertex = 7;
char visited [7], notVisited [7];
```

```

char vertex, v1,v2;
clrscr();
dispAdMat (adjMat, numVertex);
first = 7;
last =-1;
addNotVisited (notVisited , &first, adjMat[1][0]);
while (first < 7)
{
    vertex = removeNode (notVisited, &first);
    if (! ifPresent(visited, last, vertex))
        addVisited( visited,&last, vertex);

    pos =findPos (adjMat, vertex);
    for (i = 7; i>= 1; i--)
    {
        if (adjMat[pos][i] == '1')
        {
            if (! ifPresent(notVisited, 7, adjMat[0][i]) &&
                (! ifPresent(visited, 7, adjMat[0][i]) ))
                addNotVisited( notVisited,&first, adjMat[0][i]);
        }
    }
}
printf ("\n Final visited nodes...");
dispVisited (visited, last);
}

void dispAdMat(char adjMat[8][8], int numV)
{
    int i,j;
    printf ("\nThe adj Mat is--\n");
    for (i=0; i<= numV; i++)
    {
        for (j=0; j<=numV; j++)
        {
            printf ("%c ", adjMat[i][j]);
        }
        printf ("\n");
    }
    printf ("\n\n Enter any key to continue ");
    getch();
}

/* This function checks whether a vertex is present on a Q or not */
int ifPresent (char Q[], int last, char vertex)
{

```

```
int i, result=0;
for (i=0; i<= last; i++)
{
    if (Q[i] == vertex)
    { result= 1;
      break;}
}
return result;
}

void addVisited (char Q[], int *last, char vertex)
{
    *last = *last + 1;
    Q[*last] = vertex;
}

void addNotVisited (char Q[], int *first, char vertex)
{
    *first = *first - 1;
    Q[*first] = vertex;
}

void dispVisited (char Q[], int last)
{
    int i;
    printf ("\n The visited nodes are ...");
    for (i =0; i <= last; i++)
        printf (" %c ", Q[i]);
}

char removeNode (char Q[], int *first)
{
    char ch;
    ch = Q[*first];
    Q[*first] = '#';
    *first = *first + 1;
    return ch;
}

int findPos (char adjMat[8][8], char vertex)
{
    int i;
    for (i = 0; i <= 7; i++)
    {
        if (adjMat[i][0]==vertex)
            return i;
    }
    return 0;
}
```

The output of the program is shown in Figure 8.22. The final list of visited nodes matches with the results obtained in Figures 8.20 and 8.21, i.e., the visited nodes are A B C F G D E.

```

The adj Mat is--
- A B C D E F G
A 0 1 0 1 0 0 0
B 1 0 1 0 0 1 1
C 0 1 0 1 0 1 1
D 1 0 1 0 1 1 0
E 0 0 0 1 0 1 0
F 0 1 1 1 1 0 0
G 0 1 1 0 0 0 0

Enter any key to continue
Final visited nodes...
The visited nodes are ... A B C F E G D

```

Fig. 8.22 The output of DFS travel

8.4.3.2 Breadth First Search (BFS) In this method, the travel starts from a vertex then carries on to its adjacent vertices, i.e., follows the next vertex on the same level. Within a level, it travels all its siblings and then moves to the next level. Once the search space of a level is exhausted, the travel for the next level starts. This carries on till all the vertices have been visited. The only problem is that the travel may end up in a cycle. For example, in a graph, there is a possibility that the adjacent of an adjacent may be the vertex itself and the situation may end up in an endless travel, i.e., a cycle.

In order to avoid cycles, we may maintain two queues—*notVisited* and *Visited*. The vertices that have already been visited would be placed on *Visited*. When the successor of a vertex are generated, they are placed on *notVisited* only when they are not already present on both *Visited* and *notVisited* queues, thereby reducing the possibility of a cycle.

Consider the graph given in Figure 8.23. The trace of BFS travel of the given graph is given in Figures 8.23 and 8.24, through entries into the two queues—*Visited* and *notVisited*.

The *Visited* queue contains the nodes visited during BFS travel, i.e., A B D C F G E.

The algorithm for BFS travel is given below:

Algorithm BFS (firstVertex)

```

{
    add firstVertex on notVisited;
    place NULL on Visited; /* initially Visited is empty */
    while (notVisited != NULL)
    {
        remove vertex from notVisited;
        generate adjacents of vertex;
        for each adjacent of vertex do
        {
            if (adjacent is not present on Visited AND adjacent is not present
                on notVisited) Then add adjacent at Rear on notVisited;
        }
    }
}

```

```

    }
    if (vertex is not present on Visited) Then add vertex on Visited;
  }
}

```

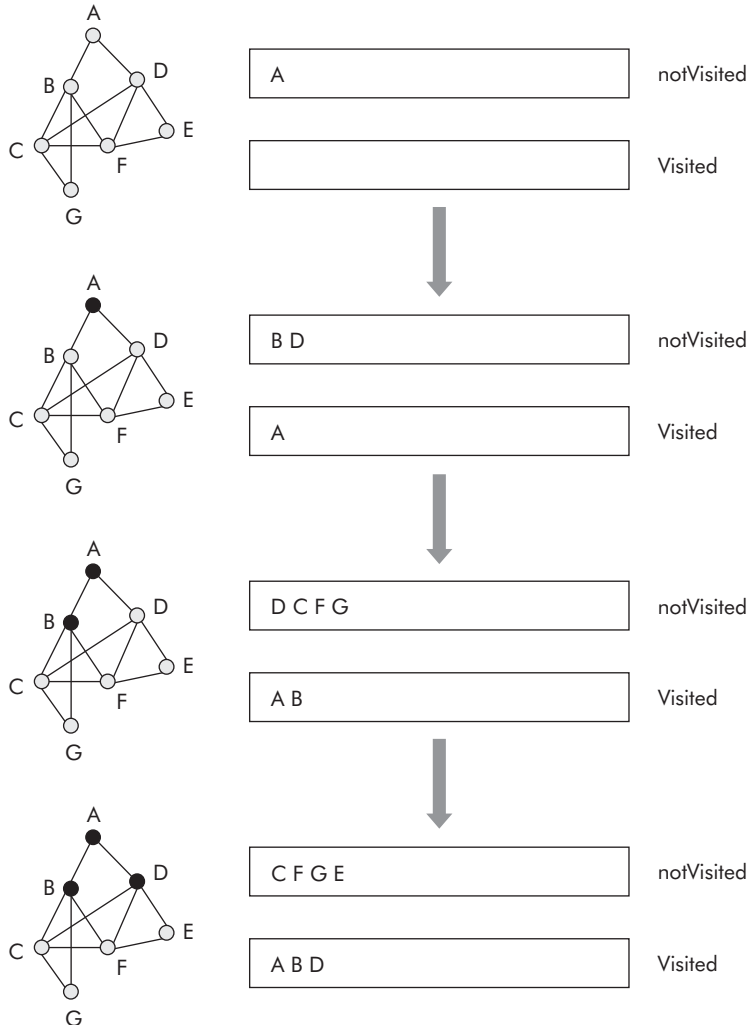


Fig. 8.23 The trace of BFS travel of a graph

Example 5: Write a program that travels a given graph using breadth first search strategy.

Solution: We would use the following functions:

(1) **void addNotVisited (char Q[], int *first, char vertex); :**

This function adds a vertex on notVisited queue at Rear of the queue.


```
(2) void addVisited (char Q[], int *last, char vertex);  
(3) char removeNode (char Q[], int *first);  
(4) int findPos (char adjMat[8][8], char vertex);  
(5) void dispAdjMat(char adjMat[8][8], int numV);  
(6) int ifPresent (char Q[], int last, char vertex);  
(7) void dispVisited (char Q[], int last);
```

The purpose of the above functions has been explained in Example 4. The function addNotVisited() has been slightly modified.

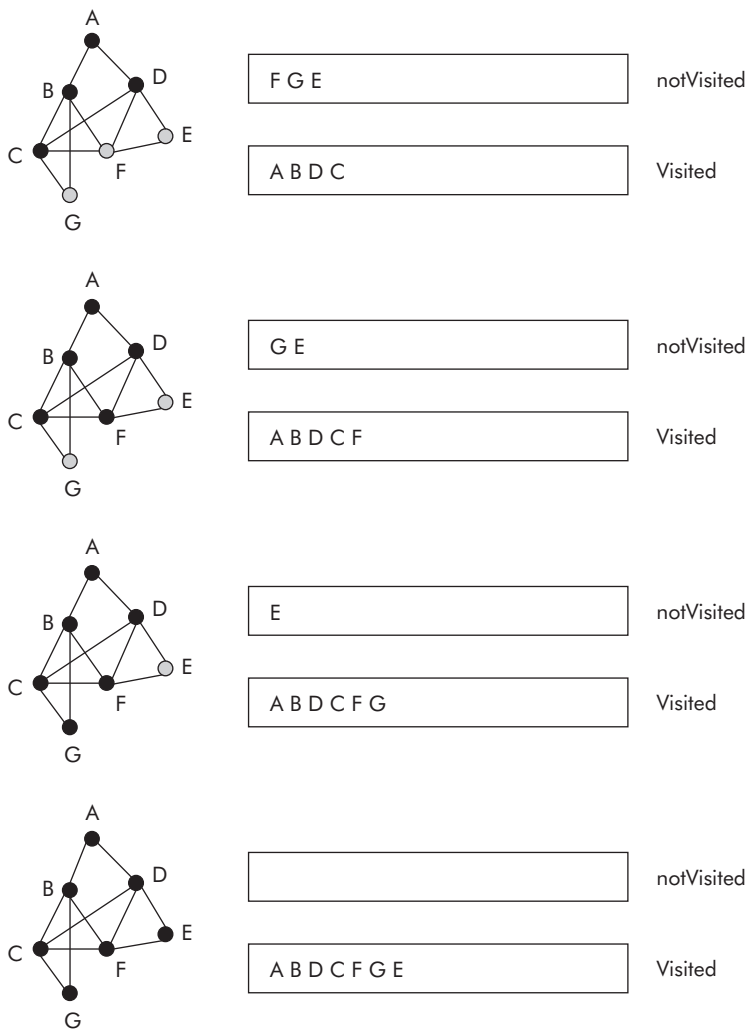


Fig. 8.24 The trace of BFS travel of a graph

The adjacency matrix of graph shown in Figures 8.20 and 8.21 is used. The program of Example 4 has been suitably modified which is given below:

```

/* This program travels a graph using BFS strategy */

#include <stdio.h>
#include <conio.h>

void addNotVisited (char Q[], int *first, char vertex);
void addVisited (char Q[], int *last, char vertex);
char removeNode (char Q[], int *first);
int findPos (char adjMat[8][8], char vertex);
void dispAdMat(char adjMat[8][8], int numV);
int ifPresent (char Q[], int last, char vertex);
void dispVisited (char Q[], int last);
void main()
{ int i,j, first, last, pos, Rear;
  char adjMat[8][8] = {   '-', 'A', 'B', 'C', 'D', 'E', 'F', 'G',
    'A', '0', '1', '0', '1', '0', '0', '0',
    'B', '1', '0', '1', '0', '0', '1', '1',
    'C', '0', '1', '0', '1', '0', '1', '1',
    'D', '1', '0', '1', '0', '1', '1', '0',
    'E', '0', '0', '0', '1', '0', '1', '0',
    'F', '0', '1', '1', '1', '1', '0', '0',
    'G', '0', '1', '1', '0', '0', '0', '0'};

  int numVertex = 7;
  char visited [7], notVisited [7];

  char vertex, v1, v2;
  clrscr();
  dispAdMat (adjMat, numVertex);
  first = -1;
  Rear = -1;
  last =-1;
  addNotVisited (notVisited , &Rear, adjMat[1][0]);
  while (first <= Rear )
  {
    vertex = removeNode (notVisited, &first);
    if (! ifPresent(visited, last, vertex))
      addVisited( visited,&last, vertex);

    pos =findPos (adjMat, vertex);
    for (i = 1; i<= 7; i++)
    {
      if (adjMat[pos][i] == '1')
      {
        if (! ifPresent(notVisited, 7, adjMat[0][i]) &&
          (! ifPresent(visited, 7, adjMat[0][i]) ))

```

```

        {
            addNotVisited( notVisited,&Rear, adjMat[0][i]);
        }
    }
}

printf ("\n Final visited nodes...");
dispVisited (visited, last);
}

void dispAdMat(char adjMat[8][8], int numV)
{
    int i,j;
    printf ("\nThe adj Mat is--\n");
    for (i=0; i<= numV; i++)
    {
        for (j=0; j<=numV; j++)
        {
            printf ("%c ", adjMat[i][j]);
        }
        printf ("\n");
    }
    printf ("\n\n Enter any key to continue ");
    getch();
}

/* This function checks whether a vertex is present on a Q or not */
int ifPresent (char Q[], int last, char vertex)
{
    int i, result = 0;
    for (i = 0; i <= last; i++)
    {
        if (Q[i] == vertex)
        { result = 1;
          break;}
    }
    return result;
}

void addVisited (char Q[], int *last, char vertex)
{
    *last = *last + 1;
    Q[*last] = vertex;
}

void addNotVisited (char Q[], int *Rear, char vertex)
{
    *Rear = *Rear + 1;
    Q[*Rear] = vertex;
}

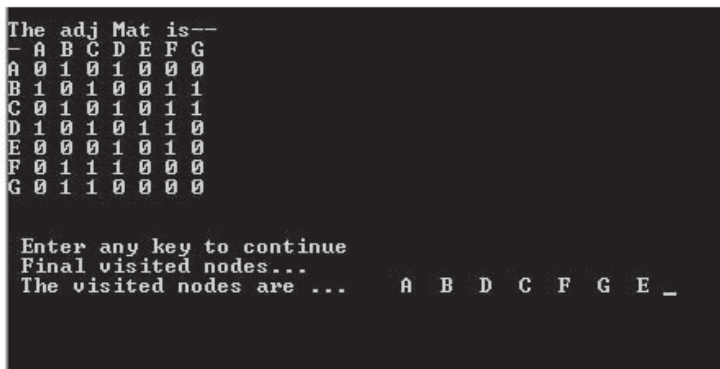
```

```

}
void dispVisited (char Q[], int last)
{
    int i;
    printf ("\n The visited nodes are ...");
    for (i =0; i <= last; i++)
    {
        printf (" %c ", Q[i]);
    }
}
char removeNode (char Q[], int *first)
{
    char ch;
    ch = Q[*first];
    Q[*first] = '#';
    *first = *first + 1;
    return ch;
}
int findPos (char adjMat[8][8], char vertex)
{
    int i;
    for (i = 0; i <= 7; i++)
    {
        if (adjMat[i][0]==vertex)
            return i;
    }
    return 0;
}

```

The output of the program is shown in Figure 8.25. The final list of visited nodes matches with the results obtained in Figures 8.23 and 8.24, i.e., the visited nodes are A B D C F G E.



```

The adj Mat is--
- A B C D E F G
A 0 1 0 1 0 0 0
B 1 0 1 0 0 1 1
C 0 1 0 1 0 1 1
D 1 0 1 0 1 1 0
E 0 0 0 1 0 1 0
F 0 1 1 1 0 0 0
G 0 1 1 0 0 0 0

Enter any key to continue
Final visited nodes...
The visited nodes are ...  A B D C F G E _

```

Fig. 8.25 The output of BFS travel of Example 5

Note: Implementation of operations on graphs using adjacency matrix is simple as compared to adjacency list. In fact, adjacency list would require the extra overhead of maintaining pointers. Moreover, the linked lists connected to vertices are independent and it would be difficult to establish cross-relationship between the vertices contained on different lists, which is otherwise possible by following a column in adjacency matrix.

8.4.4 Spanning Trees

A connected undirected graph has the following two properties:

- There exists a path from every node to every other node.
- The edges have no associated directions.

For example, the graph shown in Figure 8.26 is a connected undirected graph.

We know that a tree is a special graph which does not contain a cycle. Thus, we can remove some edges from the graph of Figure 8.26 such that it is still connected but has no cycles. The various sub-graphs or trees so obtained are shown in Figure 8.27.

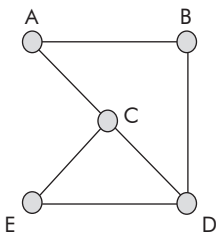


Fig. 8.26 A connected undirected graph

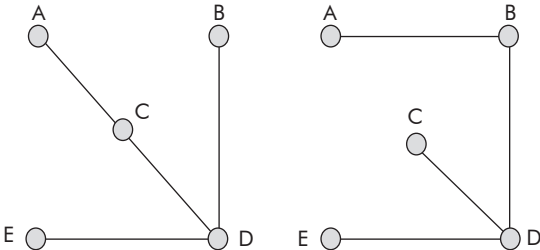


Fig. 8.27 The acyclic sub-graphs or trees

Each tree shown in Figure 8.27 is called a spanning tree of the graph given in Figure 8.26. Precisely, we can define a **spanning tree of a graph G** as a **connected sub-graph, containing all vertices of G, with no cycles**. It may be noted that there will be exactly $n-1$ edges in a spanning tree of a graph G having n vertices. Since a spanning tree has very less number of edges, it is also called as a **sparse graph**.

A graph can be traveled in two ways: Breadth First Search (BFS) and Depth First Search (DFS). In these travels, the cycles are avoided. Therefore the travels result in two trees: BFS-spanning tree and DFS-spanning trees. The DFS and BFS spanning trees corresponding to graph of Figure 8.26 are given in Figure 8.28

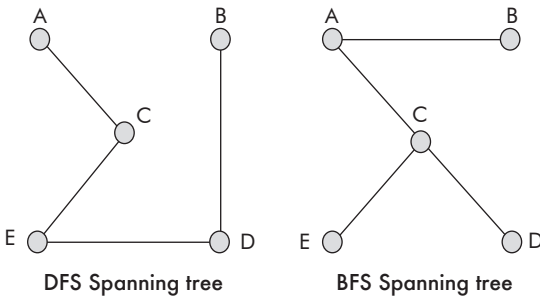


Fig. 8.28 DFS and BFS spanning trees

8.4.4.1 Minimum Cost Spanning Trees From previous section, we know that multiple spanning trees can be drawn from a connected undirected graph. If it is weighted graph then the cost of each spanning tree can be computed by adding the weights of all the edges of the spanning tree. If the weights of the edges are unique then there will be one spanning tree whose cost is minimum of all. The spanning tree with minimum cost is called as minimum spanning tree or Minimum-cost Spanning Tree (MST).

Having introduced the spanning trees, lets us now see where they can be applied. An excellent situation that comes to mind is a housing colony wherein the welfare association is interested to lay down the electric cables, telephone lines and water pipelines. In order to optimize, it would be advisable to connect all the houses with minimum number of connecting lines. A critical look at the situation indicates that a spanning tree would be an ideal tool to model the connectivity layout. In case the houses are not symmetrical or are not uniformly distanced then an MST can be used to model the connectivity layout. Computer network cabling layout is another example where an MST can be usefully applied.

There are many algorithms available for finding an MST. However, following two algorithms are very popular.

- Kruskal's algorithm
- Prim's algorithm

A detailed discussion on these algorithms is given below:

Kruskal's algorithm: In this algorithm, the edges of graph of n nodes are placed in a list in ascending order of their weights. An empty spanning tree T is taken. The smallest edge is removed from the list and added to the tree only if it does not create a cycle. The process is repeated till the number edges is less than or equal to $n-1$ or the list of edges becomes empty. In case the list becomes empty, then it is inferred that no spanning tree is possible.

Consider the graph given in Figure 8.29

The sorted list of edges in increasing order of their weight is given below:

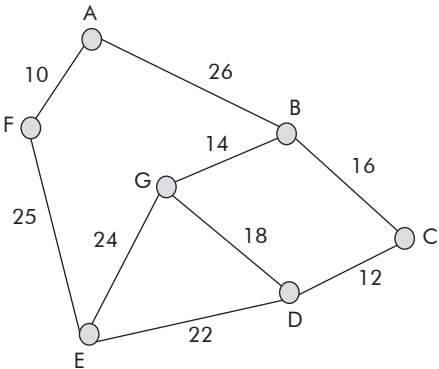


Fig. 8.29 A connected undirected graph

Edge	Weight
A-F	10
C-D	12
B-G	14
B-C	16
D-G	18
D-E	22
E-G	24
E-F	25
A-B	26

Now add the edges starting from shortest edge in sequence from A-F to A-B. The step by step creation of the spanning tree is given in Figure 8.30 (Steps 1-6)

Having obtained an MST, we are now in a position to appreciate the algorithm proposed by Kruskal as given below:

- Input: T: An empty spanning tree
E: Set of edges
N: Number of nodes in a connected undirected graph

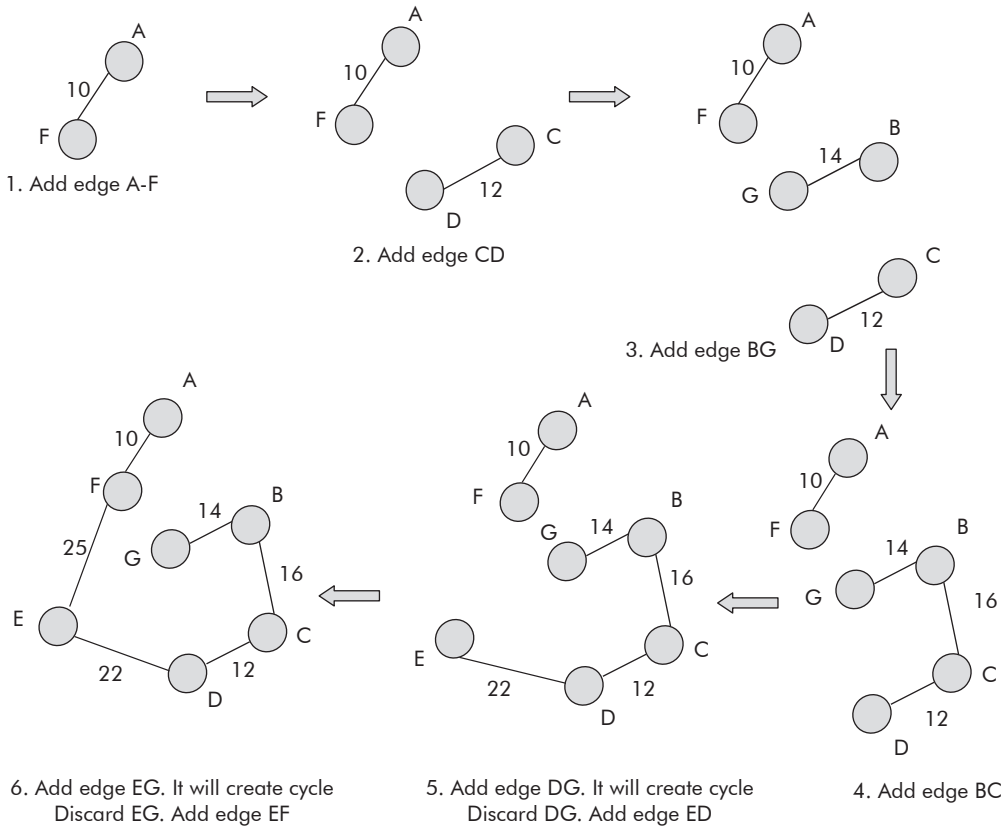


Fig. 8.30 Creation of spanning tree

Kruskal's algorithm `MST()`

```
{
Step 1. sort the edges contained in E in ascending order of their weights
2. count = 0
3. pick an edge of lowest weight from E. Delete it from E
4. add the edge into T if it does not create a cycle else discard
   it and repeat step 3
5. count = count +1
6. if (count < N-1) && E not empty repeat steps 2-5
7. if T contains less than n-1 edges and E is empty report no spanning
   tree is possible. Stop.
8. return T.
}
```

Example 6: Find the minimum spanning tree for the graph given in Figure 8.31 using Kruskal's algorithm.

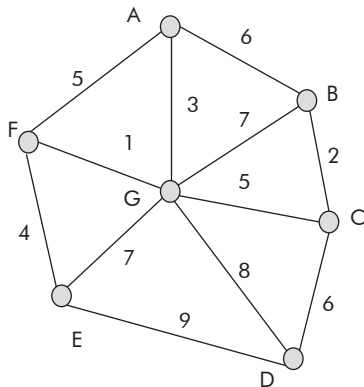


Fig. 8.31 A connected undirected graph

Solution: The sorted list of edges in increasing order of their weight is given below:

Edge	Weight
F-G	1
B-C	2
A-G	3
E-F	4
A-F	5
C-G	5
A-B	6
C-D	6
B-G	7
E-G	7
D-G	8
D-E	9

Now add the edges starting from shortest edge in sequence from F-G to D-E. The step by step creation of the spanning tree is given in Figure 8.32 (Steps 1-6)

Prim's algorithm

In this algorithm the spanning tree is grown in stages. A vertex is chosen at random and included as the first vertex of the tree. Thereafter a vertex from remaining vertices is so chosen that it has a smallest edge to a vertex present on the tree. The selected vertex is added to the tree. This process of addition of vertices is repeated till all vertices are included in the tree. Thus at given moment, there are two set of vertices: T—a set of vertices already included in the tree, and E—another set of remaining vertices. The algorithm for this procedure is given below:

Input: T: An empty spanning tree
 E: Empty set of edges
 N: Number of nodes in a connected undirected graph

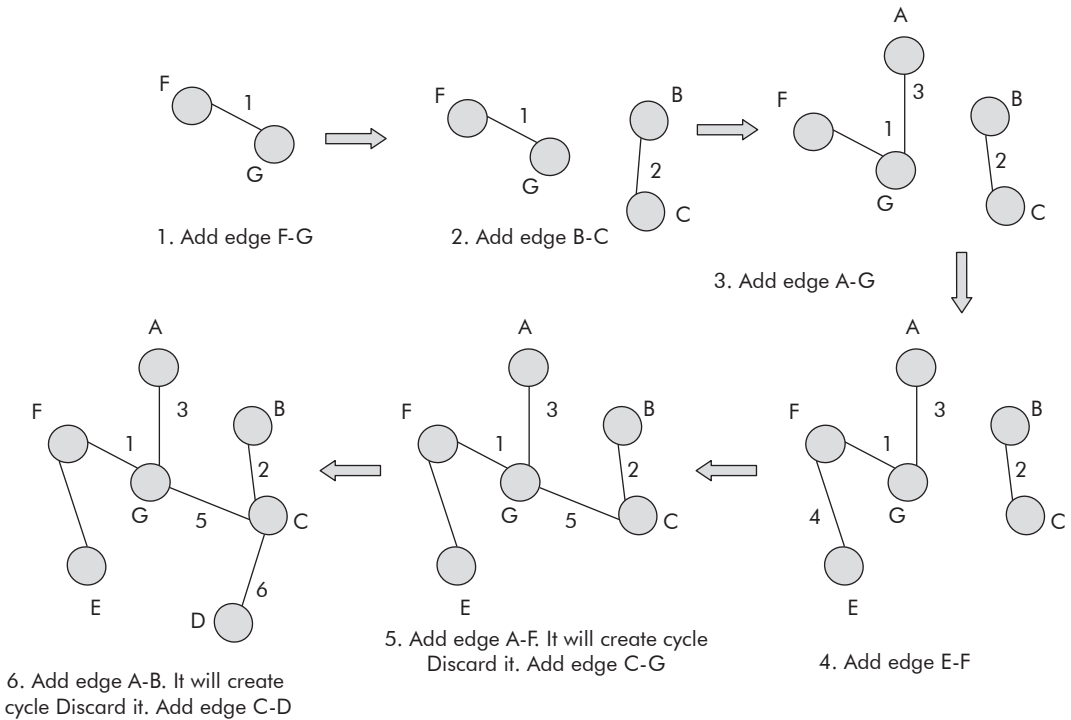


Fig. 8.32 Creation of spanning tree

Algorithm Prim()

```
{
    Step 1. Take random node v add it to T. Add adjacent nodes of v to E
    2. while ( number of nodes in T < N)
    { 2.1 if E is empty then report no spanning tree is possible.
        Stop.
        2.3 for a node x of T choose node y such that the edge (x,y) has
            lowest cost
        2.4 if node y is in T then discard the edge (x,y) and repeat step 2
        2.5 add the node y and the edge (x,y) to T.
        2.6 delete (x,y) from E.
        2.7 add the adjacent nodes of y to E.
    }
    3. return T
}
```

It may be noted that the implementation of this algorithm would require the adjacency matrix representation of a graph whose MST is to be generated.

Let us consider the graph of Figure 8.31. The adjacency matrix of the graph is given in Figure 8.33.

Let us now construct the MST from the adjacency matrix given in Figure 8.33. Let us pick A as the starting vertex. The nearest neighbour to A is found by searching the smallest entry (other than 0) in its row. Since 3 is the smallest entry, G becomes its nearest neighbour. Add G to the sub graph as edge AG

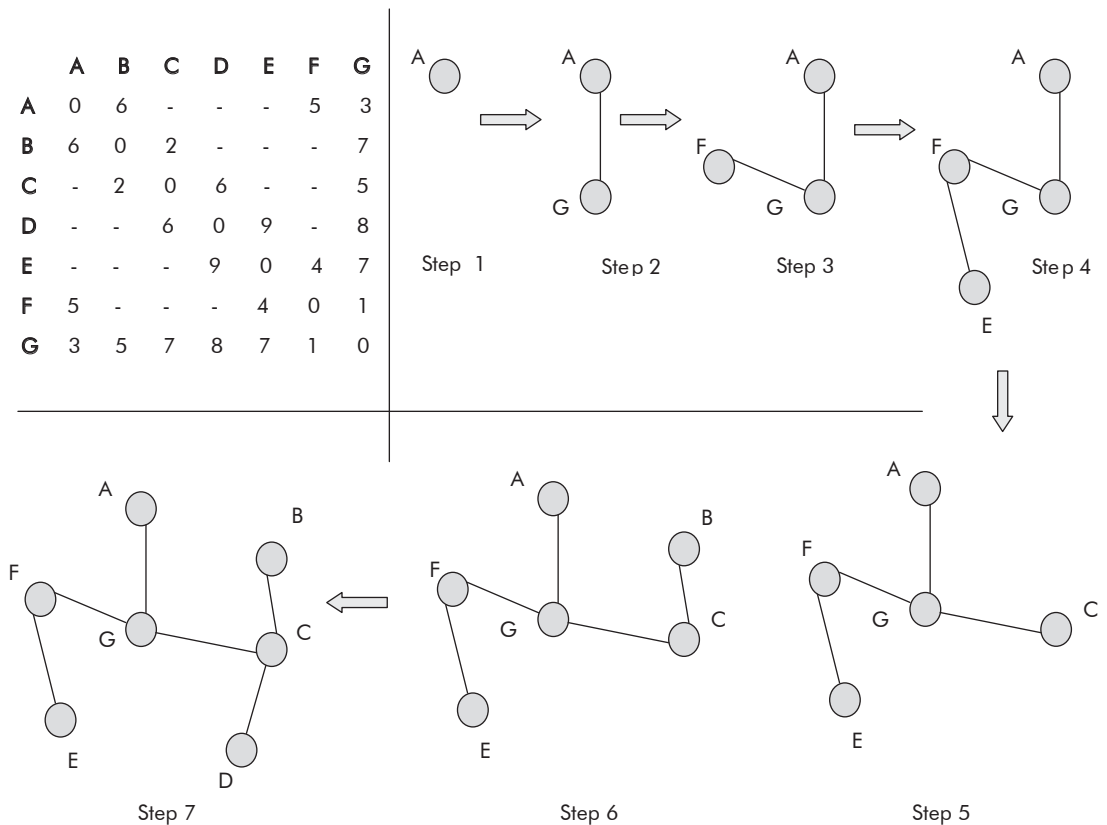


Fig. 8.33 The adjacency matrix and the different steps of obtaining MST

(see Figure 8.33). The nearest neighbour to sub graph A–G is F as it has the smallest entry i.e. 1. Since by adding F, no cycle is formed then it is added to the sub-graph. Now closest to A–G–F is the vertex E. It is added to the subgraph as it does form a cycle. Similarly edges G–C, C–B, and C–D are added to obtain the complete MST. The step by step creation of MST is shown in Figure 8.33.

8.4.5 Shortest Path Problem

The communication and transport networks are represented using graphs. Most of the time one is interested to traverse a network from a source to destination by a shortest path. For example, how to travel from Delhi to Puttaparthi in Andhra Pradesh? What are the possible routes and which one is the shortest? Similarly, through which route to send a data packet from a node across a computer network to a destination node? Many algorithms have been proposed in this regard but the most important algorithms are Warshall's algorithm, Floyd's algorithm, and Dijkstra's algorithm.

8.4.5.1 Warshall's Algorithm This algorithm computes a path matrix from an adjacency matrix A. The path matrix has the following property:

$$\text{Path}[i][j] = \begin{cases} 1 & \text{if a path from vertex } v_i \text{ to } v_j \text{ exists} \\ 0 & \text{otherwise} \end{cases}$$

Note: The path can be of length one or more

The path matrix is also called the *transitive closure* of A i.e. if path [i][k] and Path [k][j] exist then path [i][j] also exists.

We have already introduced this concept in Section 8.3.1 vide Property 3 wherein it was stated that “For a given adjacency matrix A, the element A_{ij}^k of matrix A^k (kth power of A) gives the number of paths of length k from vertex v_i to v_j ”. Therefore, the transitive closure of an adjacency matrix A is given by the following expression:

$$\text{Path} = A + A^1 + A^2 + \dots + A^n$$

The above method is compute intensive as it requires a series of matrix multiplication operations to obtain a path matrix.

Example 7: Compute the transitive closure of the graph given in Figure 8.34.

Solution: The step wise computation of the transitive closure of the graph is given in Figure 8.35.

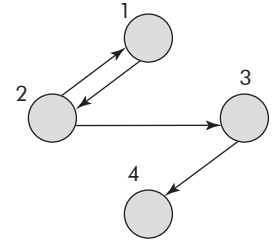


Fig. 8.34

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$A^2 = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$A^3 = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$A^4 = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\Rightarrow \text{Path} = A + A^2 + A^3 + A^4$$

$$\Rightarrow \text{Path} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\Rightarrow \text{Path} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Fig. 8.35 Transitive closure of a graph

	1	2	3	4
1	0	1	0	0
2	1	0	1	0
3	0	0	0	1
4	0	0	0	0

Adjacency matrix

Warshall gave a simpler method to obtain a path or reachability matrix of a directed graph (diagraph) with the help of only AND or OR Boolean operators. For example, it determines whether a path from vertex v_i to v_j through v_k exists or not by the following formula

$$\text{Path}[i][j] = \text{Path}[i][j] \vee (\text{Path}[i][k] \wedge \text{Path}[k][j])$$

Thus, Warshall's algorithm also computes the transitive closure of an adjacency matrix of a diagraph.

The algorithm is given below:

Input: Adjacency matrix $A[][]$ of order $n \times n$

Path Matrix $\text{Path}[][]$ of order $n \times n$, initially empty

```

Algorithm Warshall()
{
    /* copy A to Path */
    for (i = 1; i <= n; i++)
    {
        for (j = 1; j <= n; j++)
        {
            Path[i][j] = A[i][j];
        }
    }

    /* Find path from  $v_i$  to  $v_j$  through  $v_k$  */
    for (k = 1; k <= n; k++)
    {
        for (i = 1; i <= n; i++)
        {
            for (j = 1; j <= n; j++)
            {
                Path[i][j] = Path[i][j] || (Path[i][k] && Path[k][j]);
            }
        }
    }
    return Path;
}
    
```

Example 8: Apply Warshall's on the graph given in Figure 8.34 to obtain its path matrix

Solution: The step by step of the trace of Warshall's algorithm is given in Figure 8.36.

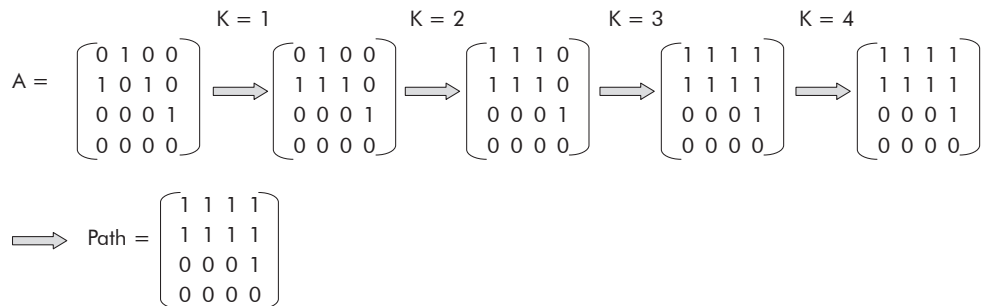


Fig. 8.36

8.4.5.2 Floyd's Algorithm The Floyd's algorithm is almost same as Warshall's algorithm. The Warshall's algorithm establishes as to whether a path between vertices v_i and v_j exists or not? The Floyd's algorithm finds a shortest path between every pair of vertices of a weighted graph. It uses the following data structures:

- (1) A cost matrix, Cost [][] as defined below:

$$\text{Cost}[i][j] = \begin{cases} \text{Cost} & \text{if there is an edge between } v_i \text{ and } v_j \\ \alpha & \text{if there is no edge between } v_i \text{ and } v_j \\ 0 & \text{if } i = j \end{cases}$$

- (2) A distance matrix Dist[][] wherein an entry $D[i][j]$ is the distance of the shortest path from vertex v_i to v_j .

In fact, for given vertices v_i and v_j , it finds out a path from v_i to v_j through v_k . If v_i - v_k - v_j path is shorter than the direct path from v_i to v_j then v_i - v_k - v_j is selected. It uses the following formula (similar to Warshall):

$$\text{Dist}[i][j] = \min (\text{Dist}[i][j], (\text{Dist}[i][k] + \text{Dist}[k][j]))$$

The algorithm is given below:

```
Algorithm Floyd ()
{
    for (i = 1; i <= n; i++)
        for (j = 1; j <= n; j++)
            Dist[i][j] = Cost[i][j]; /* copy cost matrix to distance matrix */
    for (k = 1; k <= n; k++)
    {
        for (i = 1; i <= n; i++)
        {
            for (j = 1; j <= n; j++)
                Dist[i][j] = min ( Dist[i][j] , ( Dist[i][k] + Dist[k][j] );
        }
    }
}
```

Example 9: For the weighted graph given in Figure 8.37, find all pairs shortest paths using Floyd's algorithm.

Solution: The cost matrix and the step by step of the trace of Floyd's algorithm is given in Figure 8.38.

8.4.5.3 Dijkstra's Algorithm This algorithm finds the shortest paths from a certain vertex to all vertices in the network. It is also known as *single source shortest path problem*. For a given weighted graph $G = (V, E)$, one of the vertices v_0 is designated as a source. The source vertex v_0 is placed on a set S with its shortest path taken as zero. Where the set S contains those vertices whose shortest paths are known. Now iteratively, a vertex (say v_i) from remaining vertices is added to the set S such that its path is shortest from the source. In fact this shortest path from v_0 to v_i passes only through the vertices present in set S . The process stops as soon as all the vertices are added to the set S .

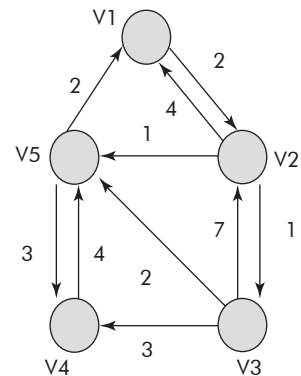


Fig. 8.37

						K = 1					K = 2									
						1	2	3	4	5										
Cost	1	0	2	a	a	a						0	2	3	a	3				
	2	4	0	1	a	1						4	0	1	a	1				
	3	a	7	0	3	2						11	7	0	3	2				
	4	a	a	a	0	4						a	a	a	0	4				
	5	2	a	a	3	0						2	4	5	3	0				
K = 5						K = 4					K = 3									
						0	2	3	6	3						0	2	3	6	3
						3	0	1	4	1						4	0	1	4	1
						4	6	0	3	2						11	7	0	3	2
						6	8	9	0	4						a	a	a	0	4
						2	4	5	3	0						2	4	5	3	0

Fig. 8.38

The dijkstra algorithm uses following data structures:

- (1) **A vector called Visited[]**. It contains the vertices which have been visited. The vertices are numbered from 1 to N. Where N is number of vertices in the graph. Initially Visited is initialized as zeros. As soon as a vertex i is visited (i.e. its shortest distance is found) Visited[i] is set to 1.
- (2) **A two dimensional matrix Cost[][]**. An entry Cost[i][j] contains the cost of an edge (i, j). If there is no edge between vertices i and j then cost[i][j] contains a very large quantity (say 9999).
- (3) **A vector dist[]**. An entry dist[i] contains the shortest distance of ith vertex from the starting vertex v1.
- (4) A vector S contains those vertices whose shortest paths are now known.

The algorithm is given below:

Algorithm dijkstra ()

```

{
    S[1] =1; Visited [1] = 1; dist[1] =0;    /* Initialize. Place v1 on S */
    for ( i = 2; i <= N; i++; )
    {
        Visited [i] = 0;
        dist [i] = cost [1][i]
    }
    for (i = 2; i <= N; i++;)
        /* Find vertex with minimum cost and its number stored in pos */
    {
        min = 9999;    pos = -1;

```

```
for ( j = 2; j <= N; j++)
{
    if (visited [j] = 0 && dist[j] < min )
    {
        pos = j;
        min = dist[j];
    }
}

/* Add the vertex (i.e. pos)
visited [pos] = 1; S[i] = pos;

/* for each vertex adjacent to pos, update distances */
for ( k = 2; k <=N; k++)
{
    if (visited [k] == 0)
        if ( dist (pos) + cost [pos][k] < dist [k])
            dist [k] = dist (pos) + cost [pos][k];
}
```

Consider the graph given in Figure 8.39.
The corresponding cost matrix and the stepwise construction of shortest cost of all vertices with respect to vertex V1 is given in Figure 8.40.

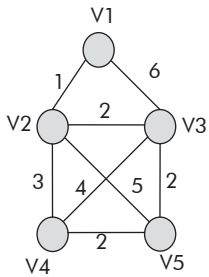


Fig. 8.39 An undirected weighted graph

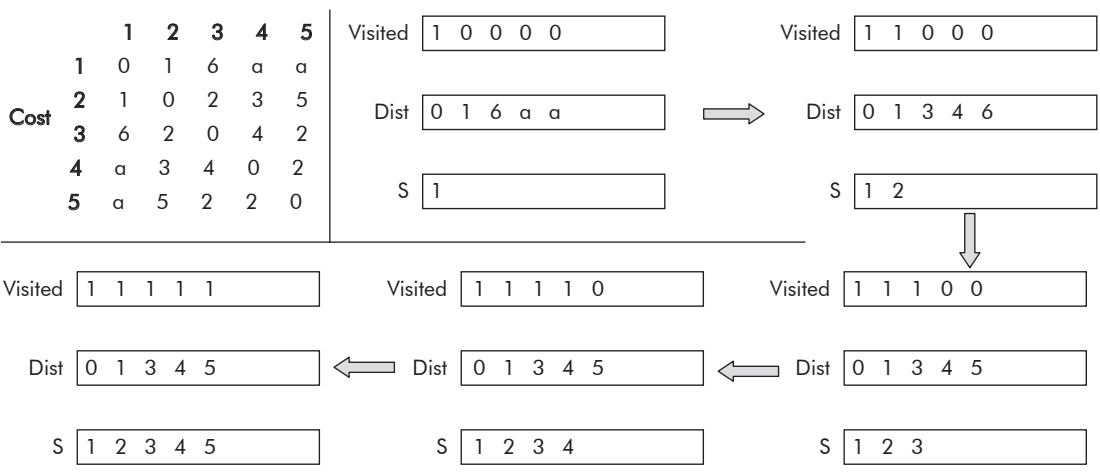


Fig. 8.40 Cost matrix and different steps of Dijkstra's algorithm

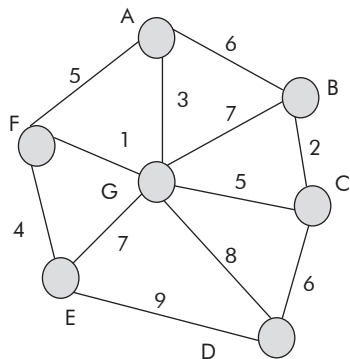


Fig. 8.41

Example 10: For the graph given in Figure 8.41, find the shortest path from vertex A to all the remaining verices.

Solution: Let us number the vertices A–G as 1–7. The corresponding cost matrix and the stepwise construction of shortest cost of all vertices with respect to vertex 1 (i.e. A) is given in Figure 8.42.

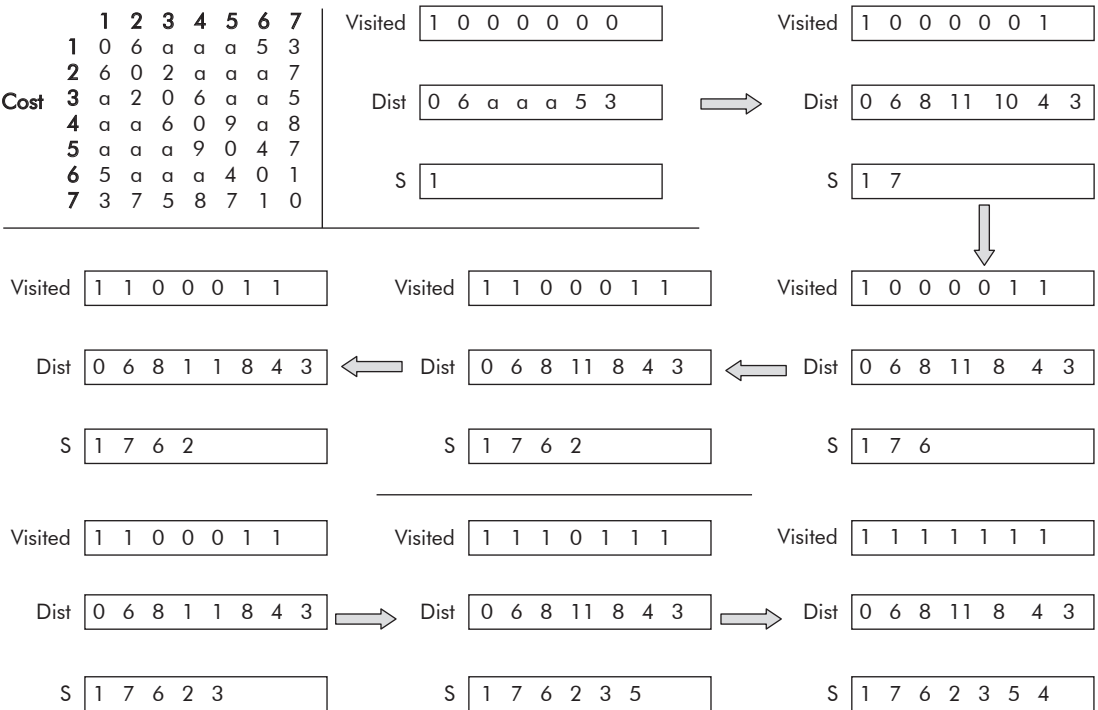


Fig. 8.42 Cost matrix and different steps of Dijkstra's algorithm

8.5 APPLICATIONS OF GRAPHS

There are numerous applications of graphs. Some of the important applications are as follows:

- (1) **Model of www:** The model of world wide web (www) can be represented by a graph (directed) wherein nodes denote the documents, papers, articles, etc. and the edges represent the outgoing hyperlinks between them as shown in Figure 8.43.

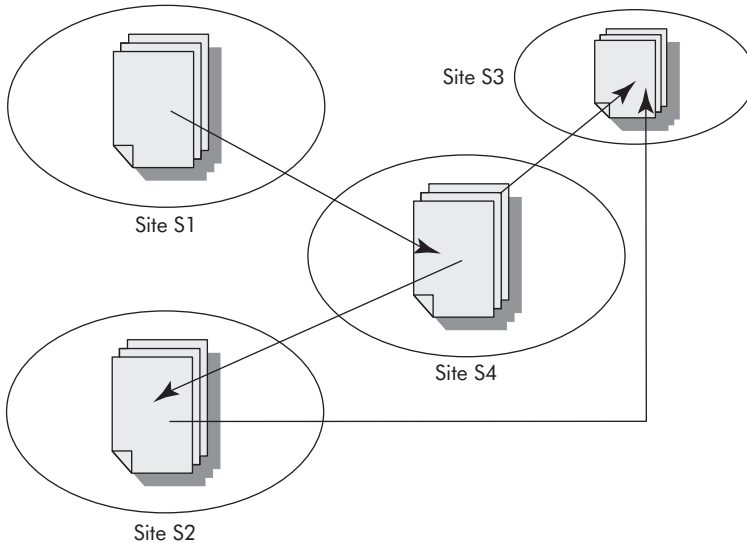


Fig. 8.43 The graphic representation of www

It may be noted that a page on site S1 has a link to page on site S4. The pages on S4 refer to pages on S2 and S3, and so on. This hyperlink structure is the basis of the web structure similar to the web created by a spider and, hence, the name world wide web.

- (2) **Resource allocation graph:** In order to detect and avoid deadlocks, the operating system maintains a resource allocation graph for processes that are active in the system. In this graph, the processes are represented as rectangles and resources as circles. Number of small circles within a resource node indicates the number of instances of that resource available to the system. An edge from a process P to a resource R, denoted by (P, R), is called a **request edge**. An edge from a resource R to a process P, denoted by (R, P), is called an **allocation edge** as shown in Figure 8.44.

It may be noted from the resource allocation graph of Figure 8.27 that the process P1 has been allocated one instance of resource R1 and is requesting for an instance of resource R2. Similarly, P2 is holding an instance of both R1 and R2. P3 is holding an instance of R2 and is asking for an instance of R1. The edge (R1, P1) is an allocation edge whereas the edge (P1, R2) is a request edge.

An operating system maintains the resource allocation graph of this kind and monitors it from time to time for detection and avoidance of deadlock situations in the system.

- (3) **Colouring of maps:** Colouring of maps is an interesting problem wherein it is desired that a map has to be coloured in such a fashion that no two adjacent countries or regions have the

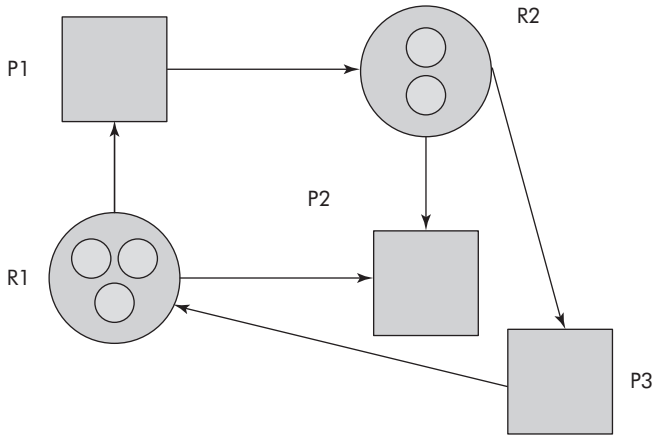


Fig. 8.44 A resource allocation graph

same colour. The constraint is to use minimum number of colours. A map (see Figure 8.45) can be represented as a graph wherein a node represents a region and an edge between two regions denote that the two regions are adjacent.

- (4) **Scene graphs:** The contents of a visual scene are also managed by using graph data structure. Virtual reality modelling language (VRML) supports scene graph programming model. This model is used by MPEG-4 to represent multimedia scene composition.

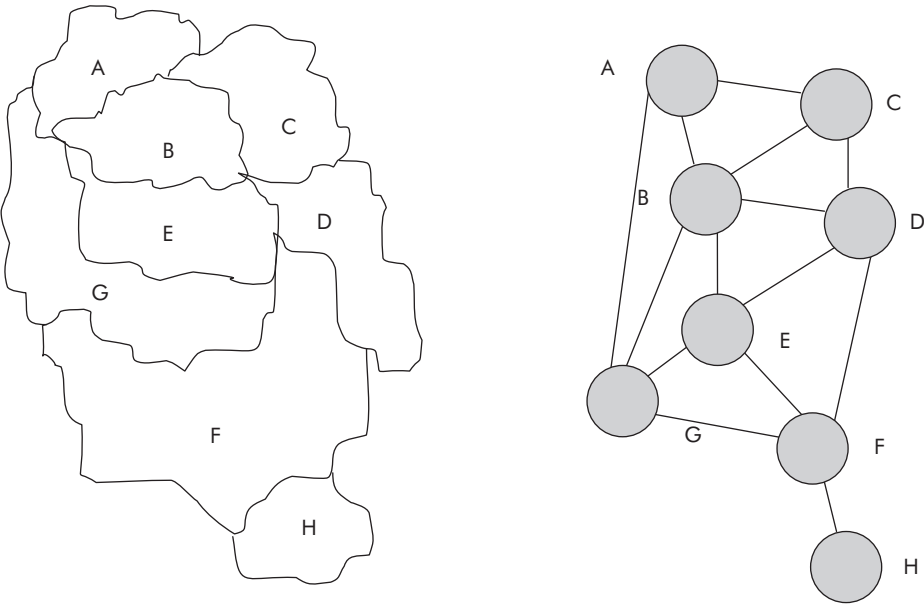


Fig. 8.45 The graphic representation of a map

A scene graph comprises nodes and edges wherein a node represents objects of the scene and the edges define the relationship between the objects. The root node called parent is the entry point. Every other node in the scene graph has a parent leading to an acyclic graph or a tree like structure as shown in Figure 8.46.

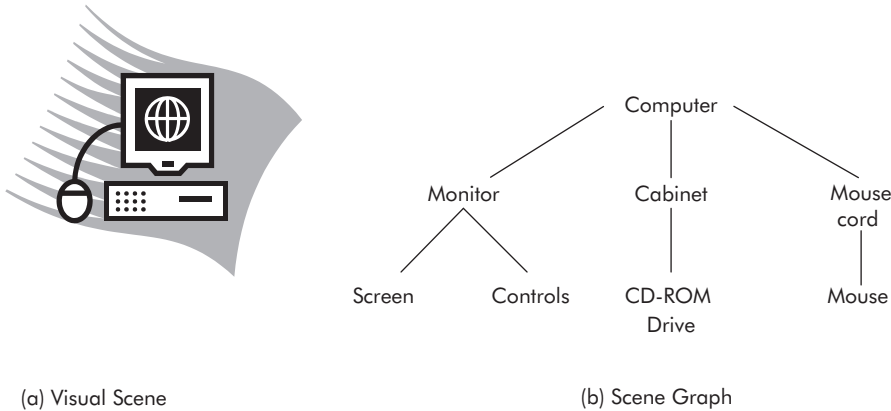


Fig. 8.46 A scene graph for a visual scene

For more reading on scene graphs, the reader may refer to the paper “Understanding Scene Graphs” by Aeron E. Walsh, chairman of Mantis development at Boston College.

Besides above, the other popular applications of graphs cited in books are:

- Shortest path problem
- Spanning trees
- Binary decision graph
- Topological sorting of a graph

EXERCISES

1. Explain the following graph theory terms:
 - a. Node
 - b. Edge
 - c. Directed graph
 - d. Undirected graph
 - e. Connected graph
 - f. Disconnected graph
2. What is a graph and how is it useful?
3. Draw a directed graph with five vertices and seven edges. Exactly one of the edges should be a loop, and should not have any multiple edges.
4. Draw an undirected graph with five edges and four vertices. The vertices should be called v1, v2, v3 and v4 and there must be a path of length three from v1 to v4.

5. Draw the directed graph that corresponds to the following adjacency matrix:

	V0	V1	V2	V3
V0	1	0	1	0
V1	1	0	0	0
V2	0	0	0	1
V3	1	0	1	0

Also, write down the adjacency list corresponding to the graph.

6. Write a 'C' function to compute the indegree and outdegree of a vertex of a directed graph when the graph is represented by an adjacency list.
7. Write a program in 'C' to implement graph using adjacency matrix and perform the following operations:
 - a. Depth First Search
 - b. Breadth First Search
8. Consider a graph representing an airline's network: each node represents an airport, and each edge represents the existence of a direct flight between two airports. Suppose that you are required to write a program that uses this graph to find a flight route from airport p to airport q. Would you use breadth first search or depth first search? Explain your answer.
9. Write the adjacency matrix of the graph shown in figure below. Does the graph have a cycle? If it has, indicate all the cycles.

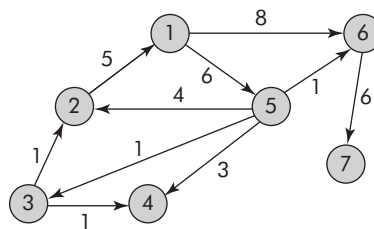


Fig. 8.47

10. Write a function that inserts an edge into an undirected graph represented using an adjacency matrix.
11. Write a function that inserts an edge into a directed graph represented using an adjacency matrix.
12. Write a function that deletes an edge from a directed graph represented using an adjacency matrix.
13. Write a function that deletes an edge from an undirected graph represented using an adjacency matrix.
14. Find the minimum spanning tree for the graph given in Figure 8.39.
15. Write the Warshall's algorithm.
16. Write the Dijkstra's algorithm.

Files

CHAPTER 9

CHAPTER OUTLINE

- 9.1 Data and Information
- 9.2 File Concepts
- 9.3 File Organization
- 9.4 Files in C
- 9.5 Files and Streams
- 9.6 Working with Files Using I/O Stream
- 9.7 Sequential File Organization
- 9.8 Direct File Organization
- 9.9 Indexed Sequential Organization
- 9.10 Choice of File Organization
- 9.11 Graded Problems

9.1 DATA AND INFORMATION

The ancient scratches on the rocks, writings on Ashoka pillars, the scriptures, and the caves of Ajanta and Ellora—all indicate that the human beings have a basic tendency to record information. The oldest records were made on clay tablets as long ago as 3700 BC. Most of the Indian scriptures were written on the thin sheets of bark of Bhojpatra, a tree found in the upper Himalayas. In second century AD, paper was developed in China. This event brought a revolution and the paper became a writing medium in whole of the world.

In an enterprise, whether it is a small shop at the corner of the street, a school or a large industry, it can be observed that each and every person uses lot of paper and a large amount of data and information exchange takes place. The workers deal with actual physical work and produce data whereas the managers deal with the information written on papers.

Let us now delve upon the terms data and information, especially in the context of files.

9.1.1 Data

The word data is a plural of *datum*, which means fact. Thus, data is a collection of facts and figures. It can be represented by symbols. For example, in a business house, data can be the name of an employee, his salary, number of hours worked, etc. Similarly in an educational institute, the data can be the marks of a student, roll numbers, percentage, etc.

9.1.2 Information

It is the data arranged in an ordered and useful form. Thus, the data can be processed so that it becomes meaningful and understandable (see Figure 9.1).

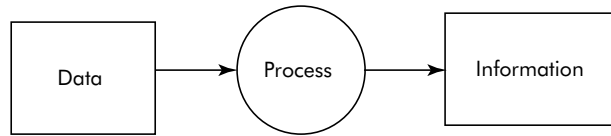


Fig. 9.1 Processing of data into information

For example, in Figure 9.2(a), the collection of characters and digits is meaningless by itself because it could refer to total marks and names of students, the house numbers and names of house owners, etc. Once the data is arranged in an order as shown in Figure 9.2(b), anybody can come to know that the data refers to names of persons and their corresponding ages. Therefore, this meaningful data can be called information.

20 RAM 18
SHAM 40
RAFIQ SINGH 60
JOHN 18
SAGUN 10

(a) Data

Name	Age
SAGUN	10
SHAM	18
JOHN	18
RAM	20
RAFIQ	40
SINGH	60

(b) Information

Fig. 9.2 Data and information

9.2 FILE CONCEPTS

Let us consider the information recorded in Figure 9.3. Each entry into the table shown in this figure is a set of three data items: Roll, Name, and Marks. The first entry states that the roll number and total marks of a student named Ajay are 001 and 50, respectively. Similarly, the second entry relates to a student named RAM whose roll number is 002 and total marks are 67.

Each entry of the table is also known as a **record** of a student. The record can be defined as a set of related data items of a particular entity such as a person, machine, place, or operation. An individual data item within a record is known as a **field**.

Thus, the record of a student is made up of following three fields:

- (1) Roll
- (2) Name
- (3) Marks

Let us assume that a particular class has 30 students. Now, the entire information of the class consisting of 30 records is referred to as a **file**. A file, therefore, consists of a number of records and each record consists of a number of items known as fields. The creation and maintenance of files is one of the

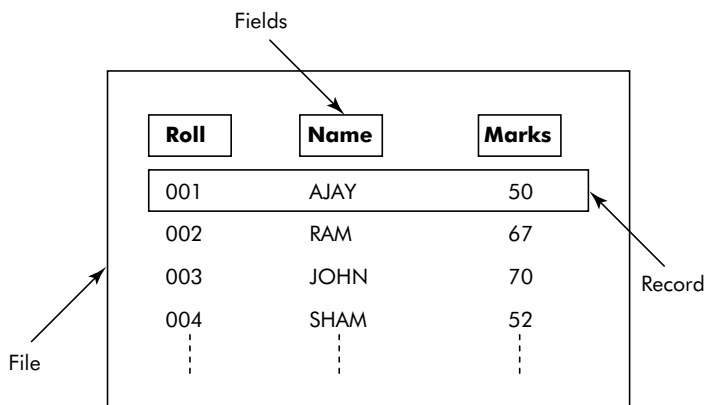


Fig. 9.3 File of records

most important activities in an organization. Depending upon its purpose, a file can be called by a specific name or type. The various types of files are tabulated in Table 9.1.

Table 9.1 Types of files

Type	Purpose	Examples
Master file	Collection of records about an important activity. It may reflect the history of events or information which can be used as a source of reference. The master file is always kept up-to-date. These are fairly permanent files.	Student fee file, Personnel file, Inventory file, Stock file.
Transaction file	A temporary file with two purposes: (1) collection of data about events as they occur and (2) updation of master files. These files are temporary in nature as these are deleted after master file is updated.	Fee receipt file, Purchase file, Sales invoice file.
Reference file	This file contains information which is required for reference and changes periodically, i.e., price list, students' attendance, share rates, etc.	Student-attendance file, Share rates file.
Report file	Data extracted from master files in order to prepare a report.	Student detainee file, Scholarship file, Taxes file.
Sort file	A working file of records to be ordered in a particular sequence.	Merit list file.
Back up file	A copy of a master or a transaction file made to ensure that a duplicate is available, if anything happens to the original.	Stud-back up file, Data back up file.
Archival files	Copies made for long-term storage of data maintained for future reference. These files are generally stored off-line, i.e., away from the computer centre.	Security file, History file.

The arrangement of records in a file can be different for different situations. The file of records shown in Figure 9.3 can be arranged according to names instead of roll numbers, i.e., in alphabetical order. The rearranged file is shown in Figure 9.4.

It may be noted here that in the file shown in Figure 9.3, the records were arranged according to Roll field whereas in the file shown in Figure 9.4, the records are arranged according to Name field. However, a record in both the files can be identified by the field called Roll. Basically, a field which is used to identify a particular record within a file is called as a **key field**. It is the key field according to which, generally, all records within a file are arranged.

The records in a file can be arranged in the following three ways:

- (1) Ascending/Descending order: The records in the file can be arranged according to ascending or descending order of a key field. For example, the records in the file of Figure 9.3 are in ascending order of its Roll field.
- (2) Alphabetical order: If a field is of string or character type, then this arrangement is used. For example, the records in the file of Figure 9.4 are in alphabetical order.
- (3) Chronological order: In this type of order, the records are stored in the order of their occurrence, i.e., arranged according to dates or events. If the key field is a date, i.e., date of birth, date of joining, etc., then this type of arrangement is used. For example, the records of a file shown in Figure 9.5 are arranged according to date of birth (DOB).

Roll	Name	Marks
001	AJAY	50
003	JOHN	70
002	RAM	67
004	SHAM	52
⋮	⋮	⋮

Fig. 9.4 File in alphabetical order

Roll	Name	DOB
004	SHAM	09:02:79
003	JOHN	11:02:79
001	AJAY	15:03:79
002	RAM	23:11:79

Fig. 9.5 File in chronological order

9.3 FILE ORGANIZATION

In a non-computerized company, the data and information are held on files kept in a paper filing system. If the company had a computer, the files would be held on computer storage. The former system is known as manual filing system and the latter is known as electronic data processing (EDP) filing system. The characteristics of a manual filing system are as follows:

- Records are stored as documents in files using papers.
- Records are arranged using a system of indices.
- The files are handled and maintained by people.

The characteristics of EDP filing system are as follows:

- Used in computer environment.
- Files are stored on computer storage.
- Records are sorted and arranged according to a key field.
- The files are handled and maintained by specific programs written in a high level language such as PASCAL, C, software packages, etc.

The manual filing system is suitable for small organizations where fast processing and storage of data is not required. In the EDP filing system, large amount of data can be managed efficiently. The data storage and retrieval becomes very fast. It may be noted that in the manual filing system, the files are generally arranged in a meaningful sequence. The records are located manually by the person in charge of the files. However when the files are stored on the computer, the files have to be kept in such a way that the records can be located, processed, and selected easily by the computer program. The handling of files depends on both input and output requirements, and the amount of data to be stored. How best can the files be arranged for ease of access? This necessity has led to the development of a number of file organization techniques as listed below:

- (i) Sequential file organization
- (ii) Direct access file organization
- (iii) Indexed sequential file organization

9.4 FILES IN C

All the programs that we have written so far have extensively used input and output statements. The `scanf` statement is used to input data from keyboard and `printf` statement to output or display data on visual display unit (VDU) screen. Thus, whenever some data is required in a program, it tries to read from the keyboard. The keyboard is an extremely slow device. For small amount of data, this method of input works well. But what happens when huge data is needed by a program? For example, a program which generates the merit list of joint entrance examination (JEE) may require data in the tune of 10,000 to 20,000 records. It is not possible to sit down and type such a large amount of data in one go. Another interesting situation is: what happens when the data or results produced by one program are required subsequently by another program? On the next day, the results may even be required by the program that produced them. Therefore, we need to have a mechanism such as files by virtue of which the program can read data from or write data on magnetic storage medium.

When the data of a file is stored in the form of readable and printable characters, then the file is known as a text file. On the other hand, if a file contains non-readable characters in binary code then the file is called a binary file. For instance, the program files created by an editor are stored as text files whereas the executable files generated by a compiler are stored as binary files. An introduction to files supported by 'C' is given in the following sections.

9.5 FILES AND STREAMS

In 'C', a stream is a data flow from a source to a sink. The sources and sinks can be any of the input/output devices or files. For input and output, there are two different streams called input stream and output stream as shown in Figure 9.6.

We have already used the following formatted stream I/O functions in our previous programs:

Stream	Description
<code>scanf</code>	standard input stream
<code>printf</code>	standard output stream

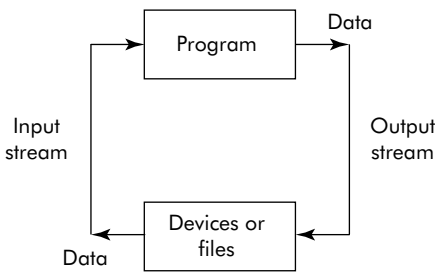


Fig. 9.6 Stream I/O

The standard source and sink are keyboard and monitor screen, respectively. These unformatted streams are initialized whenever the header file `<stdio.h>` is included in a program.

In fact, C supports a very large number of I/O functions capable of performing a wide variety of tasks. The I/O system of C is categorized into following three categories:

- (1) The stream I/O system
- (2) The console I/O system
- (3) The low-level I/O system

Some of the important stream I/O functions are listed in Table 9.2.

■ **Table 9.2** Some important stream I/O functions in C

Function	Purpose
<code>fclose</code>	close a stream
<code>feof</code>	test for end of file
<code>flush</code>	flush a stream
<code>fgetc</code>	read a character from a stream
<code>fgetchar</code>	read a character from a stream
<code>fgets</code>	read a string from a stream
<code>fopen</code>	open a stream
<code>fprintf</code>	send formatted data to a stream
<code>fputc</code>	send a character to a stream
<code>fputs</code>	send a string to a stream
<code>fread</code>	read a block of data from a stream
<code>fscanf</code>	read formatted data from a stream
<code>fseek</code>	reposition a stream pointer
<code>fwrite</code>	write a block of data to a stream
<code>fputs</code>	send a string to a stream
<code>getw</code>	read an integer from a stream
<code>putw</code>	send an integer to a stream

Some of the important console I/O functions are listed in Table 9.3.

■ **Table 9.3** Some important console I/O functions in C

Function	Purpose
<code>cgets</code>	read a string from the console
<code>clrscr</code>	clear text window
<code>cprintf</code>	write formatted data to the console
<code>getch</code>	read a character from the console
<code>getche</code>	read a character from the console and echo it
<code>gotoxy</code>	move the cursor to a specified location
<code>putch</code>	write a character to the console

Some of the low-level I/O functions are listed in Table 9.4.

■ **Table 9.4** Some important low-level I/O functions in C

Function	Purpose
chmod	Change the mode a file
close	Close the file
eof	Test for end of file
file length	Determine the length of a file in byte
lseek	Move or read data of a file pointer
open	Open a file
read	Read data from a file
setmode	Set mode of a file
write	Write data to a file

9.6 WORKING WITH FILES USING I/O STREAM

An I/O stream is a sequence of characters flowing from a source to a sink. When the data is read by a program from a device, the device becomes the source which places the data on the stream (see Figure 9.6). Obviously, the program becomes the sink that receives the data from the stream. While writing the data, the program and the device reverse the roles.

A stream is accessed by a stream pointer as shown in Figure 9.7. The stream pointer points to the beginning of a stream.

A file is opened as a stream with the help of a pointer of type FILE. Having opened the file, it is accessed through a pointer in a fashion very similar to accessing an array of strings.

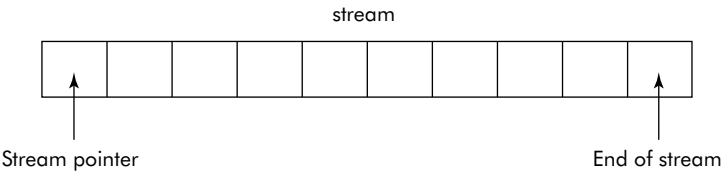


Fig. 9.7 Working of a stream

9.6.1 Opening of a File

A file can be opened in stream I/O by using function called `fopen ()`. It takes two arguments: file name and type of operation (see Table 9.5). The `fopen ()` function returns a pointer of type FILE.

Thus, a file (say "Myfile") can be opened for reading by using the following steps:

STEP 1: Declare a pointer (say `ptr`) of type FILE i.e., **FILE*ptr;**

STEP 2: Open the file using a stream I/O function called `fopen ()`.

```
ptr = fopen ("Myfile", "r");
```

The above statement requests the system to open a file called “myfile” in read mode and assigns its pointer to `ptr`.

A file can be closed by a function called `fclose()` which takes one argument of type `FILE`. Thus, the file (“myfile”) can be closed by the following statement:

```
fclose (ptr);
```

■ **Table 9.5** File opening modes

Mode	Purpose
“r”	Open a file for reading. If file does not exist, NULL is returned.
“w”	Open a file for writing. If the file does not exist, a new file is created. If the file exists, the new content overwrites the previous contents.
“r+”	Open the file for both reading and writing. If the file does not exist, NULL is returned.
“w+”	Open the file for both reading and writing. If the file exists, the previous contents are overwritten by the new one.
“a”	Open a file for appending. If the file exists, the new data is written at the end of the file else the new file is created.
“a+”	Open the file for reading and appending. If the file does not exist, a new file is created.

9.6.2 Unformatted File I/O Operations

An opened file can be read or written by the following set of unformatted I/O functions:

```
fgetc(), get(), fgetchar()
getw(), fputc(), putc()
fputchar(), fputs(), putw()
```

A brief description of each function has already been provided in Table 9.2.

Let us consider a situation wherein we desire to create a file called “message.dat” containing the following text:

**“When one apple fell, Newton was disturbed;
but when he found that all apples fell;
it was gravitation and he was satisfied”.**

The file “message.dat” would be opened with the help of `fopen()` function and the function `fputs()` would be used to write a line in the file. The `fputs()` function requires two arguments as shown below:

```
fputs (<string>, <file-pointer>)
```

where `<string>` is string of characters to be written in the file; and
`<file-pointer>` is the pointer to the file in which the text is to be written.

The required program is given below:

```
#include <stdio.h>
main()
```

```

{
    FILE *ptr;
    ptr = fopen ("message.dat", "w");

    if (!ptr)
    {
        printf ("\n The file cannot be opened");
        exit(1);
    }
    fputs("When an apple fell, Newton was disturbed\n", ptr);
    fputs("but when he found that all apples fell, \n", ptr);
    fputs("it was gravitation and he was satisfied", ptr);
    fclose(ptr);
}

```

The above program is simple wherein the first statement of function `main()` declares a pointer to a `FILE` and in the second statement a file called "message.dat" is opened in write mode. The next compound statement checks whether the operation was successful or not. Rest of the statements write the text, line by line in the file. The last statement closes the file.

A string from a file can be read by a function called `fgets()`. This function requires three arguments as shown below:

```
fgets (<string>, n, <file-pointer>)
```

where `<string>` is the character string, i.e., an array of `char` in which a group of characters from the file would be read.

`n` is the number of characters to be read from the file. The function reads a string of characters until the first new line ("`\n`") character is encountered or until the number of characters read is equal to `n - 1`.

`<file-pointer>` is the pointer to the file from which the text is to be read.

The end of a file can be detected by a function called `feof(<file.pointer>)`. This function can be used in a situation where a program needs to read whole of the file. For example, the following program uses `feof()` and `fgets()` functions to read the "message.dat" file till the end of the file is encountered. The strings read from the file are displayed on the screen.

```

#include<stdio.h>
main()
{
    char text[80];
    FILE*ptr;
    ptr = fopen("message.dat", "r");
    if(!ptr)
    {
        printf("\n the file cannot be opened");
        exit(1);
    }
    while(!feof(ptr))
    {
        fgets(text, 80, ptr);
        printf("\ %s", text);
    }
}

```

```

    }
    fclose(ptr);
}

```

Similarly, the functions `fgetc()` and `fputc()` functions can be used to read from or write a character to a file. The function `fputc()` takes two arguments, i.e., a character variable, and the file pointer whereas the function `fgetc()` takes only one argument, i.e., the file pointer.

For example, `ch=fgetc(ptr)` reads a character in `ch` from a file pointed by the pointer called `ptr`. `fputc(ch, ptr)` writes a character stored in `ch` to a file pointed by the pointer called `ptr`.

Example 1: Write a program that copies the file called “message.dat” to another file called “new.dat”.

Solution: The above program can be written by using the following steps.

Steps

- (1) Open “message.dat” for reading.
- (2) Open “new.dat” for writing.
- (3) Read a character from “message.dat”. If the character is “eof”, then go to step 5 else step 4.
- (4) Write the character in “new.dat”. Go to step 3.
- (5) Close “message.dat” and “new.dat”.

The required program is given below:

```

#include <stdio.h>
#include <conio.h>
main()
{
    FILE *ptr1, *ptr2;
    char ch;
    ptr1 = fopen("message.dat", "r");
    if (!ptr1)
    {
        printf ("\n The file %s cannot be opened", "message.dat");
        exit(1);
    }

    ptr2 = fopen("new.dat", "w");
    if (!ptr2)
    {
        printf ("\n The file %s cannot be opened", "new.dat");
        exit(1);
    }
    clrscr();
    while (!feof(ptr1))
    {
        ch = fgetc (ptr1);
        fputc( ch, ptr2);
    }
    fclose (ptr1);
    fclose (ptr2);
}

```

The efficacy of the above program can be verified by opening the “new.dat” either in notepad or in the Turbo-C editor.

It may be noted that till now we have used programs which are rigid in nature, i.e., the programs work on particular files. For instance, the above program works on “message.dat” and “new.dat” files and will not work with other files. This situation can be easily handled by taking the name of the file from the user at run time. Consider the program segment given below:

```
.
.
.
char filename [20];
FILE *ptr;
printf("\n Enter the name of the file to open for reading");
scanf("%s", filename);
ptr = fopen(filename, "r");
.
.
.
```

The above program asks the user for the name of the file to be opened giving freedom to the user to opt for any of the available files or to provide altogether a new name for a file.

Example 2: Write a program that asks from the user for the file to be opened and then displays the contents of the file.

Solution: The required program is given below:

```
#include <stdio.h>
#include <conio.h>
main()
{
    char filename[20], ch;
    FILE *ptr;
    printf("\n Enter the name of the file to be opened for reading:");
    scanf("%s", filename);
    ptr = fopen(filename, "r");
    if (!ptr)
    {
        printf ("\n The file %s cannot be opened", filename);
        exit(1);
    }
    clrscr();
    while ((ch=fgetc(ptr)) != EOF)
        printf ("%c", ch);
    fclose (ptr);
}
```

Sample **Input:** Enter the name of the file to be opened for reading: new.dat

Output: When an apple fell, Newton was disturbed;
but when he found that all apples fell;
it was gravitation and he was satisfied.

It may be noted that in the above program, instead of `feof()`, the keyword `EOF` has been used to detect the end of file. This is another method of doing the same thing. In fact, it is especially useful where a file has to be read character by character as was the case in the above program.

Example 3: Write a program that reads a file of numbers and sorts the numbers in ascending order. The sorted numbers are stored in a new file.

Solution: Assume that the input file (say “datafile.dat” contains the numbers which need to be sorted. The `getw()` function would be used to read the numbers from the file into an array of integers called `List`. The sorted list of numbers would be finally stored in an output file (say “sorted file.dat”).

The required program is given below:

```
/* This program reads a file of numbers, sorts them and
writes the sorted list into a new file */

void sort (int List[100], int N);
#include <stdio.h>
main()
{
    FILE *infile, *outfile;
    int List[100];
    int i, j, Num;
    char input_file[20], output_file[20];
    printf("\n Enter the name of Input file:");
    scanf("%s", input_file);
    printf("\n Enter the name of output file:");
    scanf("%s", output_file);
    infile = fopen (input_file, "r");
    if (! infile)
    {
        printf ("\n The file %s cannot be opened", input_file);
        exit(1);
    }
    outfile = fopen (output_file, "w");
    if (! outfile)
    {
        printf ("\n The file %s cannot be opened", output_file);
        exit(1);
    }
    /* read the input file */
    i = -1;
    while (!feof(infile))
    { i++;
      Num = getw(infile);
      /* Do not include EOF */
      if(!feof(infile)) List[i] = Num;
    }
    /* Sort the list */
    sort(List, i);
}
```



```

/* write into the output file */
for (j = 0; j <= i; j++)
    putw(List[j], outfile);
fclose(infile);
fclose(outfile);
}
void sort (int List[100], int N)
{
    int i, j, small, pos, temp;
    for (i = 0; i < N - 1; i++)
    {
        small = List[i];
        pos = i;
        for (j = i + 1; j < N; j++)
        {
            if (List [j] < small)
            {
                small = List[j];
                pos = j;
            }
        }
        temp = List[i];
        List[i] =List[pos];
        List[pos] = temp;
    }
}

```

Note: The files created in the above program cannot be verified by opening them in normal text editor like notepad. The reason is that the functions `getw()` and `putw()` read/write the numbers in a file with a format different from text format.

Therefore, a file consisting of numbers can be read by `getw()` function. In fact, following program can be used to create the `datafile.dat`.

```

#include <stdio.h>
main()
{
    FILE *ptr;
    int val, i;
    ptr = fopen("datafile.dat", "w");
    do
    {
        printf ("\n Enter a value");
        scanf ("%d", &val);
        if (val != -9999) putw(val, ptr);
    }
    while (val != -9999);
    fclose(ptr);
}

```

The numbers are written in the file as long as system does not encounter -9999, a number used to identify the end of the list.

Similarly, the following program can be used to see the contents of a file (say "sortedfile.dat")

```
#include <stdio.h>
main()
{
    FILE *ptr;
    int val, i;
    ptr = fopen("sortedfile.dat", "r");
    printf ("\n");
    while (!feof (ptr))
    {
        val =getw(ptr);
        if (!feof(ptr)) printf ("%d ", val); /*Do not print EOF*/
    }
    fclose(ptr);
}
```

It may be further noted that both the programs take care that the EOF (a number) does not get included into the list of numbers, being manipulated by them.

9.6.3 Formatted File I/O Operations

C supports `fscanf()` and `fprintf()` functions to read or to write from files. The general format of these functions is given below:

```
fscanf (<file pointer>, <format string>, <argument list>);
fprintf (<file pointer>, <format string>, <argument list>);
```

where <file pointer> is the pointer to the file in which I/O operations are to be done.

<format string> is the list of format specifiers, i.e., %d, %f, %s, etc.

<argument list> is the list of arguments.

Example 4: Write a program that takes a list of numbers from the user through standard input device, i.e., keyboard. The number -9999 marks the end of the list. The list is then written on a user defined file by using `fprintf()` function.

Solution: The required program is given below:

```
#include <stdio.h>
main()
{
    FILE *ptr;
    int val, i;
    char outfile[20];
    printf("\n Enter the name of the file :");
    scanf("%s", outfile);
    ptr = fopen(outfile, "w");
    do
    {
```

```

    printf ("\n Enter a value");
    scanf ("%d", &val);
    if (val != -9999) fprintf(ptr, "%d ", val);
}
while (val != -9999);
fclose(ptr);
}

```

Example 5: Use the file created in Example 4. Read the list of numbers contained in the file using `fscanf()` function. Print the largest number in the list.

Solution: The numbers from the file would be read one by one and compared for getting the largest of them. After EOF, the largest would be displayed.

The required program is given below:

```

#include stdio.h
main()
{
    FILE *ptr;
    int val, i, large;
    char infile[20];
    printf("\n Enter the name of the file:");
    scanf("%s", infile);
    ptr = fopen(infile, "r");
    printf("\n The list is ...");
    large = -9999;
    while (1)
    {
        fscanf(ptr,"%d", &val);
        if (feof(ptr)) break;
        printf ("%d ", val);
        if (large < val) large = val;
    }
    printf("\n The largest is: %d", large);
    fclose(ptr);
}

```

Sample output:

```

Enter the name of the file: Mydata.dat
The list is 12 34 56 78 32 41 21 13
The largest is: 78.

```

9.6.4 Reading or Writing Blocks of Data in Files

A block of data such as an array or a structure can be written in a file by using `fwrite()` function. Similarly, the block of data can be read from the file using `fread()` function.

The general formats of `fread()` and `fwrite()` functions are given below:

```

fread(<address of object>, <size of object>, <number of items>, <file pointer>);
fwrite(<address of object>, <size of object>, <number of items>, <file pointer>);

```

where `<address of object>` is the address or pointer to the object from which the data is to be read/written on the file and vice versa.

`<size of object>` is computed by the function `sizeof()` and included in the argument list
`<number of items>` is the number of data items to be read or written. Generally, it is set as 1.
`<file pointer>` is the pointer to the file from which data is to be read or written.

Let us now try to write the following structure into a file (say `test.dat`):

```
struct stud {
    char name [20];
    int roll;
};
```

The following statements can be used to do the required task:

```
FILE *ptr;
struct stud {
    char name [20];
    int roll;
};
struct stud ob = {"Ram", 101};
ptr = fopen ("test.dat", "w");
fwrite (& ob, sizeof (struct stud), 1, ptr);
fclose (ptr);
```

A similar code can be written to read the structure from a file.

A complete program that writes a structure into a file and then reads and displays it, is given below:

```
#include <stdio.h>
main()
{ FILE *ptr;
    struct stud {
        char name[20];
        int roll;
    };
    struct stud obl;
    struct stud ob = {"Ram", 101};
    ptr = fopen("test.dat", "w");
    fwrite (&ob, sizeof (struct stud), 1, ptr);
    fclose(ptr);
    ptr = fopen("test.dat", "r");
    fread (&obl, sizeof (struct stud), 1, ptr);
    fclose(ptr);
    printf ("\n %s", obl.name);
    printf ("\n %d", obl.roll);
}
```

Example 6: Write a program that creates a file called "Marks.dat" and writes the records of all the students studying in a class.

Solution: The required program is given below:

```
#include <stdio.h>
struct student {
    char name[20];
    int roll;
    int marks;
};
main()
{ FILE * ptr;
  struct student studob;
  int t = sizeof (struct student);
  ptr = fopen ("marks.dat", "w");
  printf ("\n Enter the student data terminated by roll = -9999");
  while (1)
  {
    printf ("\nName:"); scanf("%s", studob.name);
    printf ("\nRoll:"); scanf("%d", &studob.roll);
    if (studob.roll == -9999) break;
    printf ("\nmarks:"); scanf("%d", &studob.marks);
    fwrite(&studob, t, 1, ptr);
  }
  fclose (ptr);
}
```

Example 7: A file called “Marks.dat” contains the records of students having the following structure:

```
struct student
{
    char name [20];
    int roll;
    int marks;
};
```

Write a program that reads marks and creates the following two files:

- (1) “Pass.dat” should contain the roll numbers of students scoring more than or equal to pass-marks stored in a variable called pass_marks.
- (2) “FAIL.dat” should contain the roll numbers of students scoring below the pass_marks.

Solution: The required program is given below:

```
#include <stdio.h>
struct student {
    char name[20];
    int roll;
    int marks;
};
main()
{ FILE * ptr, *p, *f;
  struct student studob;
```

```

int pass_marks;
int t = sizeof (struct student);
ptr = fopen ("marks.dat", "r");
p = fopen("pass.dat", "w");
f = fopen ("fail.dat", "w");
printf ("\n Enter the passing parks :");
scanf ("%d", &pass_marks);
while (!feof(ptr))
{
    fread(&studob,t,1,ptr);
    if (feof(ptr)) break;
    if( studob.marks >= pass_marks)
        fprintf (p, "%d ", studob.roll);
    else
        fprintf (f,"%d ", studob.roll);
}
fclose (ptr);
fclose (p);
fclose(f);
}

```

Example 8: Write a program that opens the “pass.dat” and “fail.dat” files created in the program in Example 7 and displays two separate lists of roll numbers who have passed and failed in the examination.

Solution: The required program is given below:

```

#include <stdio.h>
main()
{ int roll;
  FILE *p, *f;
  p = fopen("pass.dat", "r");
  if (! p)
  {
      printf ("\n The file %s cannot be opened", "pass.dat");
      exit(1);
  }
  f = fopen ("fail.dat", "r");
  if (! f)
  {
      printf ("\n The file %s cannot be opened", "fail.dat");
      exit(1);
  }
  printf ("\n The list of pass students :");
  while (! feof(p))
  {
      fscanf(p,"%d", &roll);
      if (!feof(p)) printf ("%d ", roll);
      roll = getch();
  }
}

```

```

}
printf ("\n The list of fail students :");
while (!feof(f))
{
    fscanf(f,"%d", &roll);
    if (!feof(f)) printf ("%d ", roll);
}
fclose (p);
fclose(f);
}
}

```

9.7 SEQUENTIAL FILE ORGANIZATION

This file organization is the simplest way to store and retrieve records of a file. In this file, the records are stored in a meaningful order according to a key field. The first record in the order is placed at the beginning of the file. The second record is stored right after the first, the third after the second, and so on. However, the records may be ordered in ascending or descending order by the key field which can be numeric (such as student roll number) or alphabetic (such as student name). It may be noted that the order never changes afterwards. The arrangement of records in a sequential file is shown in Figure 9.8.

Since all records in the file are stored sequentially by position, the records can be identified by the key field only.

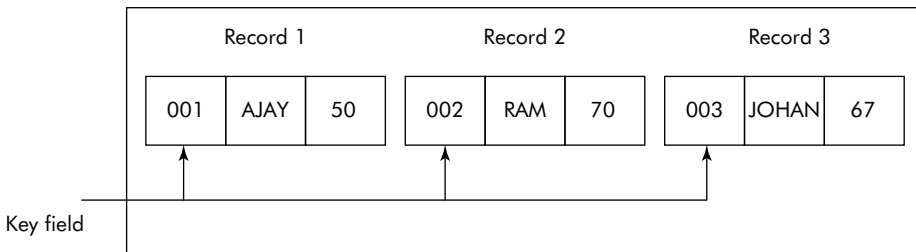


Fig. 9.8 Arrangement of records in a file

9.7.1 Creating a Sequential File

The sequential file can be created on a storage device (magnetic tape or disk) by the following three steps:

- (1) A name for the file is chosen. The system is requested to open a file on the storage device with the chosen file name.
- (2) The records for the file are input sequentially one by one into the computer which in turn stores them in the order of their arrival on to the file.
- (3) After all the records are transferred into the file, the system is requested to close the file.

A flow chart for this process is given in Figure 9.9. The above mentioned steps are carried out with the help of a program written in a high level language. All the high level languages, such as BASIC, FORTRAN, PASCAL and C, support sequential files.

9.7.2 Reading and Searching a Sequential File

Suppose student data of an engineering college is maintained in a file (say “stud.dat”)

Let us assume that on a particular day, the principal of the school requires some information from the file called `stud.dat`. Now, the obvious step is to read the file to get the required information. The read operation for a sequential file is explained below.

To read a sequential file, the system always starts at the beginning of the file and the records are read one by one till the required record is reached or end of the file is encountered. The end of the file (EOF) indicates that all the records in the file have been read. For instance, if the desired record happens to be 50th one in a file, the system starts at the first record and reads ahead one record at a time until the 50th is reached.

The reading of a file involves the following operations:

- (1) Open the file.
- (2) Read a record.
- (3) Check if the record is the desired one. If yes, then display information and perform step 6.
- (4) Check for the end of the file (EOF). If yes, then perform step 6.
- (5) Go to step 2.
- (6) Close the file.

A flow chart for the reading of file is given in Figure 9.10.

The step 3, i.e., check if the record is the desired one is carried out by matching the key field of the record read from the file with the key field of the desired record. For example, if the principal desires the information about a student whose roll number is 125 then the ROLL field of each record read from the file will be compared with the value 125. The process will be repeated till a record with ROLL equal to 125 is encountered.

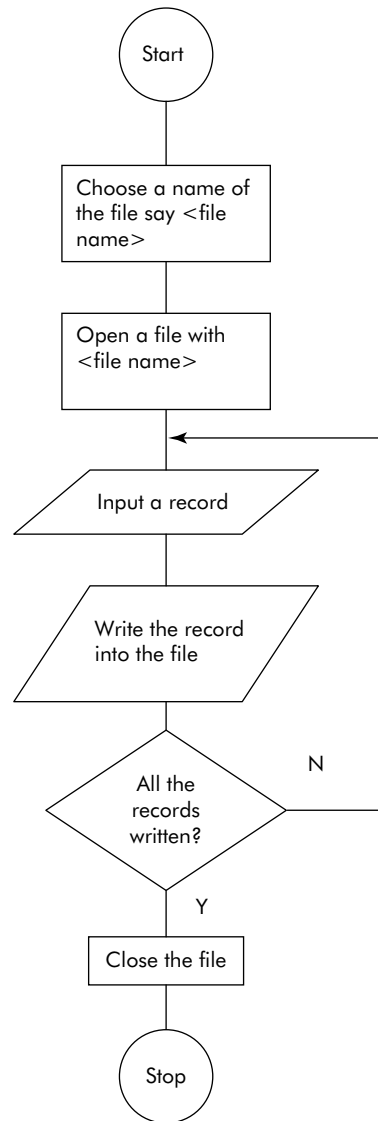
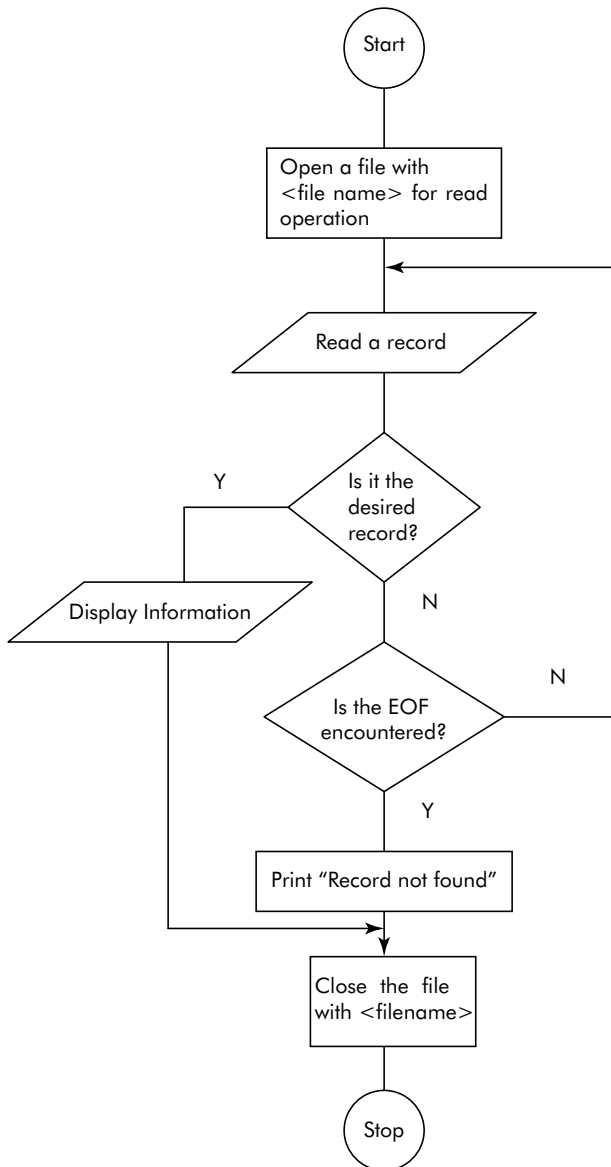
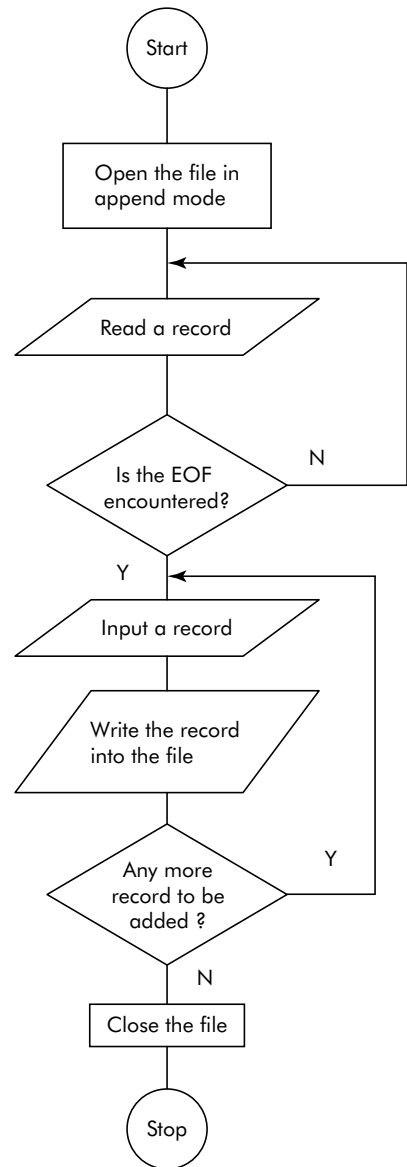


Fig. 9.9 Creation of a sequential file

9.7.3 Appending a Sequential File

The append operation is done with a view of adding more records to an existing file. In this operation, the file is opened and is read record by record, till EOF is encountered. The new records which are to be added to the file are then written into the file. Once all the records have been added, the file is closed. Thus, the following operations are done for the purpose of appending a file:

- (1) Open the file for append operation.
- (2) Read the file till EOF is encountered.

**Fig. 9.10** Reading sequential file**Fig. 9.11** Appending a sequential file

- (3) Read the record to be added.
- (4) Write the record on the file.
- (5) Close the file.

A flow chart for appending of a file is given in Figure 9.11.

Example 9: Write a program that creates a sequential file of records of students who are players of different games. The structure of student record is given below:

```
struct student
{
char name [20]
int roll;
char class [5];
char game [15];
};
```

Solution: The required program that creates the file (say “Play_Master.dat”) is given below:

```
/* This program creates a sequential file */

#include <stdio.h>
struct student {
    char name[20];
    int roll;
    char class[5];
    char game[15];
};
main()
{
    FILE *ptr;
    char myfile[15];
    int t = sizeof(struct student);
    struct student studrec;
    printf("\n Enter the name of the file:");
    scanf("%s", myfile);
    ptr = fopen (myfile, "w");
    printf("\n Enter the records of the students one by one");
    studrec.roll = 0;
    while (studrec.roll != -9999)
    {
        printf("\nName:"); fflush(stdin); gets(studrec.name);
        printf("\nRoll:"); scanf("%d", &studrec.roll);
        printf("\nClass:"); fflush(stdin); gets(studrec.class);
        printf("\nGame:"); fflush(stdin); gets(studrec.game);
        if (studrec.roll != -9999)
            fwrite(&studrec, t, 1, ptr);
    }
    fclose (ptr);
}
```

Example 10: Write a program that uses “Play_Master.dat” created in Example 9. For a given game, it displays the data of all the students who play that game.

Solution: The required program is as follows:

```

/* This program searches a sequential file */
#include <stdio.h>
struct student {
    char name[20];
    int roll;
    char class[5];
    char game[15];
};
main()
{
    FILE *ptr;
    char game[15], myfile[15];
    int t = sizeof(struct student);
    struct student studrec;
    printf("\n Enter the name of the file to be opened");
    scanf("%s", myfile);
    ptr = fopen (myfile, "r");
    printf("\n Enter the name of the game:");
    fflush(stdin);
    gets(game);
    fread(&studrec,t,1,ptr);
    while (!feof(ptr))
    {
        if (! strcmp(game, studrec.game))
        { printf("\nName =%s", studrec.name);
          printf("\nRoll =%d", studrec.roll);
          printf("\nclass =%s", studrec.class);
          printf("\nGame =%s", studrec.game);
        }
        fread(&studrec,t,1,ptr);
    }
    fclose(ptr);
}

```

Example 11: Write a program that opens the “Play_Master.dat” file and appends records in the file till the roll = -9999 is encountered.

Solution: The required program is given below:

```

/* This program appends records in a sequential file */
#include <stdio.h>
struct student {
    char name[20];
    int roll;
    char class[5];
    char game[15];
};

```

```

main()
{
    FILE *ptr;
    char myfile[15];
    int t = sizeof(struct student);
    struct student studrec;
    printf("\n Enter the name of the file:");
    scanf("%s", myfile);
    ptr = fopen (myfile, "r+");
    printf("\n Enter the records to be appended ");
    while (!feof (ptr))
        fread(&studrec,t,1,ptr);
    studrec.roll=0;
    while (studrec.roll !=-9999)
    {
        printf("\nName:"); fflush(stdin); gets(studrec.name);
        printf("\nRoll:"); scanf("%d", &studrec.roll);
        printf("\nClass:"); fflush(stdin); gets(studrec.class);
        printf("\nGame:"); fflush(stdin); gets(studrec.game);
        if (studrec.roll != -9999)
            fwrite(&studrec, t, 1, ptr);
    }
    fclose (ptr);
}

```

It may be noted that in the above program, the file was opened in “r+” mode. In fact, the same job can be done by opening the file in “a”, i.e., append mode. The program that uses this mode is given below:

```

/* This program appends records in a sequential file */
#include <stdio.h>
struct student {
    char name[20];
    int roll;
    char class[5];
    char game[15];
};
main()
{
    FILE *ptr;
    char myfile[15];
    int t = sizeof(struct student);
    struct student studrec;
    printf("\n Enter the name of the file:");
    scanf("%s", myfile);
    ptr = fopen (myfile, "a");
    printf("\n Enter the records to be appended");
}

```

```

studrec.roll = 0;
while (studrec.roll != -9999)
{
    printf("\nName:"); fflush(stdin); gets(studrec.name);
    printf("\nRoll:"); scanf("%d", &studrec.roll);
    printf("\nClass:"); fflush(stdin); gets(studrec.class);
    printf("\nGame:"); fflush(stdin); gets(studrec.game);
    if (studrec.roll != -9999)
        fwrite(&studrec, t, 1, ptr);
}
fclose (ptr);
}

```

Example 12: Modify the program written in Example 11 so that it displays the contents of a file before and after the records are appended to the file.

Solution: A function called show() would be used to display the contents of the file. The required program is given below:

```

/* This program displays the contents before and after it appends records
in a sequential file */

```

```

#include <stdio.h>
void show(FILE *p, char x[15]);
struct student {
    char name[20];
    int roll;
    char class[5];
    char game[15];
};
main()
{
    FILE *ptr;
    char myfile[15];
    int t = sizeof(struct student);
    struct student studrec;
    printf("\n Enter the name of the file:");
    scanf("%s", myfile);
    printf("\n The Records before append");
    show(ptr, myfile);
    ptr = fopen (myfile, "a");
    printf("\n Enter the records to be appended ");
    studrec.roll=0;
    while (studrec.roll !=-9999)
    {
        printf("\nName:"); fflush(stdin); gets(studrec.name);
        printf("\nRoll:"); scanf("%d", &studrec.roll);
        printf("\nClass:"); fflush(stdin); gets(studrec.class);
    }
}

```

```

    printf("\nGame:"); fflush(stdin); gets(studrec.game);
    if (studrec.roll != -9999)
        fwrite(&studrec, t, 1, ptr);
}
fclose (ptr);
printf("\n The Records after append");
show(ptr, myfile);
}
void show(FILE *p, char myfile[15])
{
    struct student studrec;
    p=fopen(myfile, "r");
    fread(&studrec, sizeof(studrec),1,p);
    while (!feof(p))
    {
        printf ("\nName:"); fflush(stdout); puts(studrec.name);
        printf("\nRoll:"); printf("%d", studrec.roll);
        printf("\nClass:"); fflush(stdout); puts(studrec.class);
        printf("\nGame:"); fflush(stdout); puts(studrec.game);
        fread(&studrec, sizeof(studrec),1,p);
    }
    fclose(p);
}
}

```

9.7.4 Updating a Sequential File

The updation of a file means that the file is made up-to-date with the latest changes relating to it. The original file is known as a master file or old master file and the temporary file which contains the changes or transactions is known as transaction file. Examples of transactions are making purchases, payment for purchases, total sales in the day, etc.

In order to update a sequential file, the transactions are sorted according to the same key used for the old master file, i.e., both the files are sorted in the same order on the same key. Thus, the master file and the transaction file become **co-sequential**. A new file called new master file is opened for writing. The old master file and the transaction file are merged according to the following logic:

“The key of the record from the transaction file is compared with key of the record from the old master file. If identical, the transaction record is written on the new master file. Otherwise, the master record is written on the new master file”.

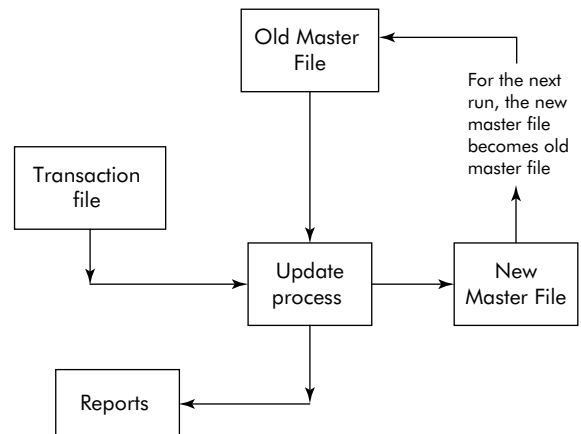


Fig. 9.12 Transaction processing

Example 13: A file called old master of a bank contains the records of saving fund A/C of its customers in ascending order of account numbers. The format of the record is given below:

```
Name
Acnt_No
Balance_Amt
```

A file called transaction file contains the details about transactions done in A/C on a particular day. It also contains the records in the ascending order of A/C numbers. Each record of that file has the following format:

```
Acnt_No
Trans_type
Trans_amt
```

The field Trans-amt contains the amount withdrawn or deposited to the account depending upon the value of Trans-type being 0 or 1, respectively. Write a program that reads the two files and does the transaction processing giving a third file called new master.

Solution: In this program, the following menu would be displayed so that the possible actions can be conveniently carried out by the user:

```
Menu
Create old master file      0
Create transaction file    1
Generate new master file   2
Show contents of a file    3
Quit                       4
```

The required program is given below:

```
/* This program maintains the saving A/C of a bank */

#include <stdio.h>
#include <conio.h>
struct Acnt_Rec {
    char name [20];
    int Acnt_No;
    float Balance_Amt;
};
struct Trans_Rec {
    int Acnt_No;
    int Trans_type;
    float Trans_Amt;
};
void creat_T(char Trans[15]);
void creat_O(char O_master[15]);
void creat_N(char O_master[15],char Trans[15], char N_master[15]);
void showfile(char fname[15],int f_type);
main()
{
```

```

struct Acnt_Rec Acntob;
struct Trans_Rec Transob;
char O_master[15], N_master[15], Trans[15], fname[15];
int choice, f_type;
/* Show Menu */
do
{ clrscr();
  printf("\nMenu");
  printf ("\n");
  printf ("\n Create Old master file      0");
  printf ("\n Create Transaction file    1");
  printf ("\n Create New master file      2");
  printf ("\n Show Contents of a file     3");
  printf ("\n Quit                          4");
  printf ("\n Enter your choice:"); scanf ("%d", &choice);
  switch (choice)
  {
    case 0: clrscr();
      printf ("\n Enter the name of the file");
      fflush(stdin); gets(O_master); creat_O(O_master);
      break;
    case 1: clrscr();
      printf ("\n Enter the name of the file");
      fflush(stdin); gets(Trans); creat_T(Trans);
      break;
    case 2: clrscr();
      printf ("\n Enter the name of Old master file");
      fflush(stdin); gets(O_master);
      printf ("\n Enter the name of New master file");
      fflush(stdin); gets(N_master);
      printf ("\n Enter the name of Transaction file");
      fflush(stdin); gets(Trans);
      creat_N(O_master,Trans, N_master);
      break;
    case 3: printf ("\nEnter the name of the file to be displayed");
      fflush(stdin); gets(fname);
      printf ("\n Enter 0/1 for Master/Transaction");
      scanf ("%d", &f_type); showfile(fname,f_type);
      break;
  }
}
while (choice != 4);
}

void creat_O(char O_master[15])
{
  FILE *ptr;

```



```

    struct Acnt_Rec Aob;
    int t = sizeof(Aob);
    ptr = fopen (O_master,"w");
    printf("\n Enter the records one by one terminated by A/c = -9999");
    Aob.Acnt_No=0;

    while (Aob.Acnt_No != -9999)
    {
        printf ("\nName:"); fflush(stdin); gets(Aob.name);
        printf ("\nA/C No:"); scanf("\n%d", &Aob.Acnt_No);
        printf ("\nBalance Amount:"); scanf("\n%f", &Aob.Balance_Amt);
        if (Aob.Acnt_No != -9999) fwrite (&Aob,t,1,ptr);
    }
    fclose(ptr);
}

void creat_T(char Trans[15])
{ FILE *ptr;
  struct Trans_Rec Aob;
  int t = sizeof(Aob);
  ptr = fopen (Trans,"w");
  printf ("\n Enter the records one by one terminated by A/c = -9999");
  Aob.Acnt_No=0;

  while (Aob.Acnt_No != -9999)
  {
      printf ("\n A/C No:"); scanf("\n%d", &Aob.Acnt_No);
      printf ("\n Transaction Type (0/1):"); scanf("\n%d", &Aob.Trans_
      type);
      printf ("\n Transaction Amount:"); scanf("\n%f", &Aob.Trans_Amt);
      if (Aob.Acnt_No != -9999) fwrite (&Aob,t,1,ptr);
  }
  fclose(ptr);
}

void creat_N(char O_master[15],char Tran[15], char N_master[15])
{
    FILE *Old, *New, *Trans;
    struct Acnt_Rec Oob;
    struct Trans_Rec Tob;
    int t1, t2;
    t1 = sizeof (struct Acnt_Rec);
    t2 = sizeof(struct Trans_Rec);
    /* open files */
    Old = fopen(O_master, "r");
    Trans = fopen(Tran, "r");
    New = fopen(N_master, "w");

    /* perform transaction processing */

```

```

fread(&Oob,t1,1,Old);
fread(&Tob,t2,1,Trans);
while (!feof(Old) && !feof(Trans))
{
    if (Oob.Acnt_No < Tob.Acnt_No)
    {fwrite (&Oob,t1,1,New);
    fread(&Oob,t1,1,Old);
    }
    else
    {
        while ((Oob.Acnt_No == Tob.Acnt_No ) && !feof(Trans))
        {
            if (Tob.Trans_type == 0)
                Oob.Balance_Amt = Oob.Balance_Amt - Tob.Trans_Amt;
            else
                Oob.Balance_Amt =Oob.Balance_Amt + Tob.Trans_Amt;
            fread(&Tob,t2,1,Trans);
        }
        fwrite (&Oob,t1,1,New);
        fread(&Oob,t1,1,Old);
    }
}

/*Copy rest of the file*/
while (!feof(Old))
{
    fwrite (&Oob,t1,1,New);
    fread(&Oob,t1,1,Old);
}

/*Close files */
fclose(Old);
fclose(New);
fclose(Trans);
}

void showfile(char fname[15],int f_type)
{ FILE *ptr;
  char ch;
  struct Acnt_Rec Aob;
  struct Trans_Rec Tob;
  int t1 = sizeof (Aob);
  int t2 = sizeof(Tob);
  if (f_type == 0)
  {
      ptr = fopen (fname, "r");
      fread (&Aob,t1,1,ptr);
      while(!feof(ptr))
      {
          printf ("\nName:"); fflush(stdout); puts(Aob.name);

```

```
printf ("\nA/C No:"); printf("\n%d", Aob.Acnt_No);
printf ("\nBalance Amount:"); printf("\n%f", Aob.Balance_Amt);
fread (&Aob,t1,1,ptr);
ch = getch();
}
}
else
{ ptr = fopen (fname, "r");
fread (&Tob,t2,1,ptr);
while(!feof(ptr))
{
printf ("\n A/C No:"); printf("\n%d", Tob.Acnt_No);
printf ("\n Transaction Type:"); printf("\n%d", Tob.Trans_type);
printf ("\n Transaction Amount:"); printf("\n%f", Tob.Trans_Amt);
fread (&Tob,t2,1,ptr);
ch = getch();
}
}
fclose(ptr);
}
```

9.8 DIRECT FILE ORGANIZATION

The direct files allow records to be read or written on to a file without proceeding sequentially from the beginning. In this organization, the records are not stored in sequence. To read or write, an address is first calculated by applying a mathematical function to the key field of the record. The record is then stored at the generated address. The mathematical function is called a Hash function.

However, a poor form of direct access can be obtained in C files by using the `fseek()` function. With the help of this function, the stream file pointer can be positioned at a particular record. It takes three arguments as shown below:

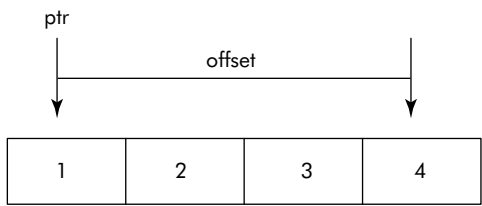
```
fseek (<file pointer>, <offset>, <start-position>)
```

where `<file pointer>` is the pointer to the file.

`<start position>` is the start position of the file. It can have either of the following values:

Value	Meaning
0	The starting position is the beginning of the file.
1	Use the current file pointer as the start- ing position.
2	The starting position is the end of the file.

`<Offset>` is the offset, i.e., how much the pointer should advance from the starting position.



From Figure 9.13, it may be noted that file pointer (`ptr`) is positioned at the

Fig. 9.13 The working of `fseek(ptr, 4, 0)` function.

beginning of the file because the start position is equal to 0 (zero). The offset (i.e., 4) would move the file pointer to the fourth record.

Similarly, the `fseek(ptr, 9, 2)` suggests that from the end of the file move back by 9 records.

Thus, it is a **relative file organization** wherein the records are accessed directly in relation with the starting point, specified within the file. The offset is usually computed by the following formula:

$$\text{offset} = \text{record number} \times \text{size of a record}$$

Example 14: Write a program that creates a file of room data of a hotel having following structure. The rooms are numbered starting from 1.

```
struct room
{
    int room_no;
    char room_type;
    char status;
};
```

where room type may take value S/D, i.e., single bed or double bed. Similarly, status may take value A/N, i.e., available or not available. Use `fseek()` function to display the information about a particular room of the hotel.

Solution: The required program is given below:

```
/* This program uses fseek() function to search
records in a file */
#include <stdio.h>
struct room {
    int room_no;
    char room_type;
    char room_status;
};
main()
{ FILE *ptr;
  char filename[20];
  int t = sizeof(struct room);
  struct room rob;
  int no_of_rooms, i, choice;
  printf ("\n Enter the name of the file");
  fflush(stdin); gets(filename);
  printf ("\n Enter the number of rooms");
  scanf("%d", &no_of_rooms);
  /* create the file */
  ptr = fopen(filename, "w");
  printf ("\n Enter the room data one by one");
  for (i = 1; i <= no_of_rooms; i++)
  { rob.room_no = i;
    printf ("\n Room No. %d:", i);
    printf ("\n Room_Type (S/D):");
    fflush(stdin); scanf ("%c", &rob.room_type);
    rob.room_type = toupper(rob.room_type);
```

```

printf("\n Room_Status (A/N):");
fflush(stdin); scanf ("%c", &rob.room_status);
rob.room_status = toupper (rob.room_status);
fwrite (&rob,t,1,ptr);
}
fclose(ptr);
ptr = fopen(filename, "r");
/*search a record */
do
{
    printf("\n Menu");
    printf("\n");
    printf("\n Display info      1");
    printf("\n Quit                2");
    printf("\n Enter your choice");
    scanf("%d", &choice);
    if (choice == 1)
    {
        printf("\n Enter the room no.");
        scanf("%d", &i);
        fseek(ptr, (i - 1) * t, 0);
        fread(&rob, t, 1, ptr);
        clrscr();
        printf("\n Room No.= %d", rob.room_no);
        printf("\n Room Type = %c", rob.room_type);
        printf("\n Room Status= %c", rob.room_status);
    }
    else
        break;
}
while (1);
fclose (ptr);
}

```

The **characteristics of a relative file** are given below :

- A relative file has fixed length records.
- All records in the file are written and accessed with the help of a record number.
- Unless the record number for a record within the file is known, one cannot gain access to the information that the record contains.
- The file is open for both writing and reading operations at once.

The **advantages of direct files** are given below :

- Any record can be directly accessed.
- File processing and updation activities can be performed without the creation of new master file.
- Speed of record processing in case of large files is very fast as compared to sequential files.
- The files can be kept up-to-date because on-line updation is possible.

- Concurrent processing of several files is possible. For example, when a sale is recorded in a company, the account receivable can be posted on one file and simultaneously the inventory withdrawal can also be written on another file.

9.9 INDEXED SEQUENTIAL ORGANIZATION

A very large number of applications require a mixture of sequential and direct processing. For example, in an inventory control system, it is desired that at regular intervals the transactions be processed sequentially in a file and if needed, make direct accesses inquiries into the same file. File organization using the Indexed Sequential Access Method (ISAM) provide the essential benefits of both the sequential and random methods.

The indexed organization is similar to a telephone directory wherein one can retrieve a telephone number by using an index provided at the beginning. In this organization, the records of the file are stored sequentially in blocks on the disk. Each block can store a specified amount or set of records. The records in this file can be accessed through an index. An index is a separate file from the original file. Each entry into the index contains only two items of data: the value of the keyfield of last record in the block and its corresponding starting block address (see Figure 9.14).

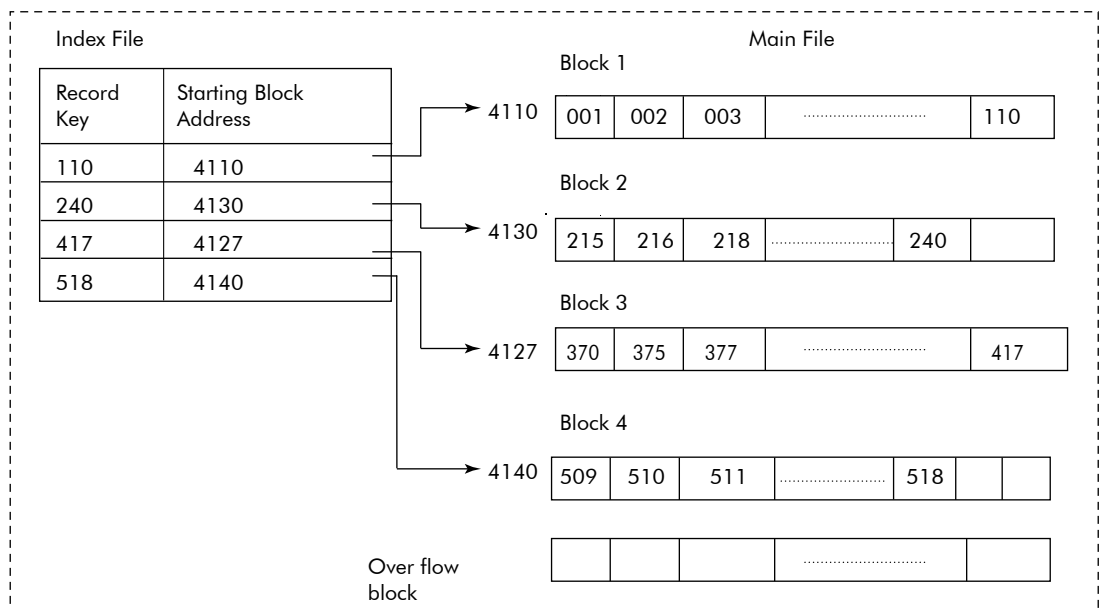


Fig. 9.14 Indexed sequential organization

9.9.1 Searching a Record

To search a specified record in an indexed sequential file, the index is first searched to find the block in which the key of the required record is present. When index entry is found, the address of the corresponding block containing the record on the disk is noted and then whole of the block is brought in the

main memory. Once the block is brought inside the main memory, the record is searched sequentially in the block.

It may be noted from Figure 9.14 that each entry in the index contains the highest key value for any record in the block and the starting address of the block. Now if we want to search a record with key value 376, then the index is searched first. The first entry (110) of the index is compared with the search key (376). Since 110 is the highest key value in its corresponding block and it is less than the search key (376), the comparison with next entry is done. The comparison continues till it reaches to third entry and it is found that the key of the entry (417) is larger than the search key (376). Thus it is established that the required record should be on its corresponding block. The block's starting address is 4127. Now the access is made to disk address 4127 and the block is brought into the main memory. The search key is compared with key of the first record of the block (370). Then it moves on to the second record and the process is repeated till it reaches to the third record (key = 376). The keys match establishing the fact that the required record has been found. Now, it can perform the desired operation on the record such as display the contents, print certain values, etc.

9.9.2 Addition/Deletion of a Record

When records are added to an already existing indexed sequential file, the system inserts them at proper places so as to retain the sequential nature of each block. For example, assume that records with keys 217 and 371 are to be added into the file shown in Figure 9.14. The record with key 217 is searched into the file and its exact location where the record is to be inserted i.e. in the block starting with address 4130 and after the record with key value 216. Figure 9.15 shows that the record with key 217 has been inserted after record with key 216 and the rest of the records have been moved down to create space for the incoming record.

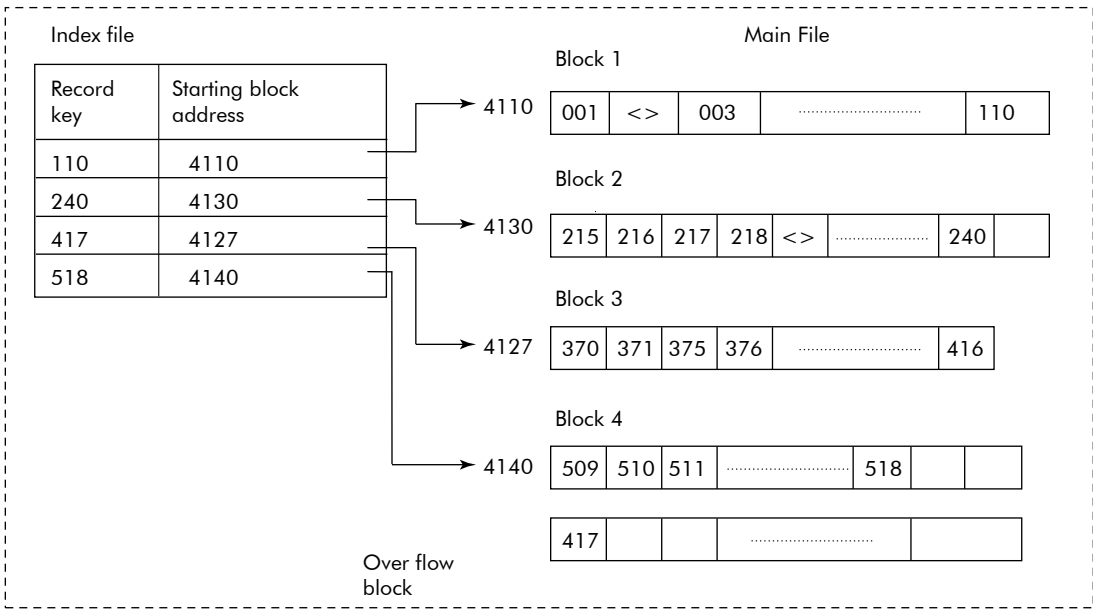


Fig. 9.15 Operations on indexed sequential file

The addition of record (371) will push the record (417) out of the Block 3. The reason being that the block is already full, and to accommodate the incoming record (371) the record from the extreme end has to leave the block. The overflowed record is then stored in a separate block known as overflow block. The deletion of a record from the file is logical deletion, i.e., the record is not physically or actually deleted from the file but it is marked with special characters to indicate that the record is no longer accessible. For example, if the records with key value 002 and 219 are to be deleted then these records are searched in the file. If they are found then the records are marked with special characters (<> in our case) (see Figure 9.15). The presence of '<>' in a record indicates that the record has been deleted.

The modifications in a record involves the following steps

- (1) Search the record
- (2) Read the record
- (3) Change the record
- (4) Rewrite the changed record in its original location.

The ISAM files are very flexible in the sense that they can be processed either sequentially or randomly, depending on the requirements of the user.

9.9.3 Storage Devices for Indexed Sequential Files

The magnetic disks are the most suitable storage media for Indexed sequential files. The ISAM files are mapped on to the cylinders of the disk. We know that the index file consists of three parts: index, main file (data part), and overflow area. A cylinder on the disk is accordingly divided into three parts: index area, prime area, and overflow area as shown in Figure 9.16

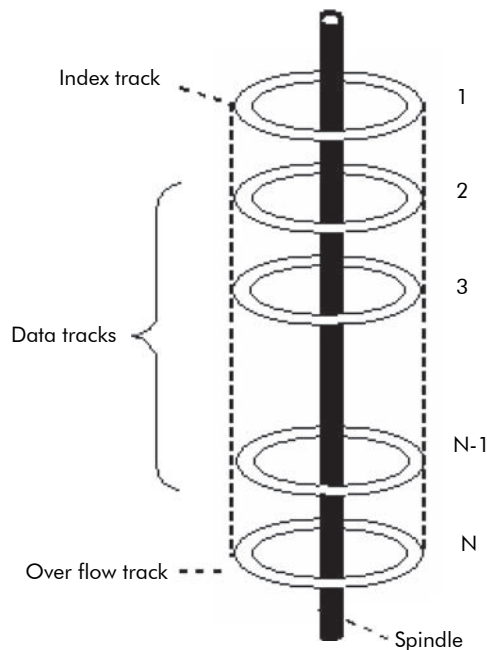


Fig. 9.16

ISAM on a cylinder of a magnetic disk

- **Index area:** The first track in each cylinder is reserved for an index. This index describes the storage of records on the various tracks of the cylinder.
- **Prime area (Data tracks):** The various records of the file are written on this area on a cylinder always starting from the 2nd track. The records are recorded sequentially along the various tracks. The highest key on the track and the track address are entered into the index.
- **Overflow area:** The area is created to accommodate the overflowed records in the file. The overflow area is generally the last track on the same cylinder. This area is unoccupied at the time of creation of file.

9.9.4 Multilevel Indexed Files

If the file is so large that it cannot be accommodated on one cylinder then the records of the file are stored on many cylinders. To handle this situation, a separate index known as cylinder index is maintained to indicate how records are distributed over a number of cylinders. Therefore, we have a hierarchy of indices in the form of multilevel index as shown in Figure 9.17. The cylinder index has entries consisting of two parts: the highest key value on the cylinder and the cylinder number.

To search a record in a multi level index file, the cylinder index is first searched to find the cylinder on which the record is stored. Once the cylinder is known then the track index is searched to find the block on which the record is residing. Rest of the process is same as in the case of single level indexed file.

The advantages of indexed files are as follows:

- Provides good flexibility for users who need both direct accesses and sequential progressing with the same file.
- It makes rapid access possible over the indexes.
- Interactive processing is possible.

The disadvantages of indexed files are as follows:

- ISAM files generate excessive overflows, and vacant storage locations are created.
- Extra storage space is required for the index especially when multilevel indexes are maintained.
- The overflow areas sometimes overflow to other cylinders and thus cause much read/write head movement making the access very slow. Therefore, the file has to be reorganized and indexes rearranged accordingly. This is a time consuming process.

9.10 CHOICE OF FILE ORGANIZATION

The choice of a file organization depends on the nature of a given application. How a data file is used, determines which particular file organization method is appropriate for it. The following factors play an important part in the selection of a file organization.

1. Activity ratio: A record is said to be active if it requires processing in a file. The portion containing records in a file that requires processing is called as activity of a file. The ratio of number of active records and the total number of records in the file is known as activity ratio

$$\text{Activity ratio} = \frac{\text{Number of active records}}{\text{Total number of records in the file}}$$

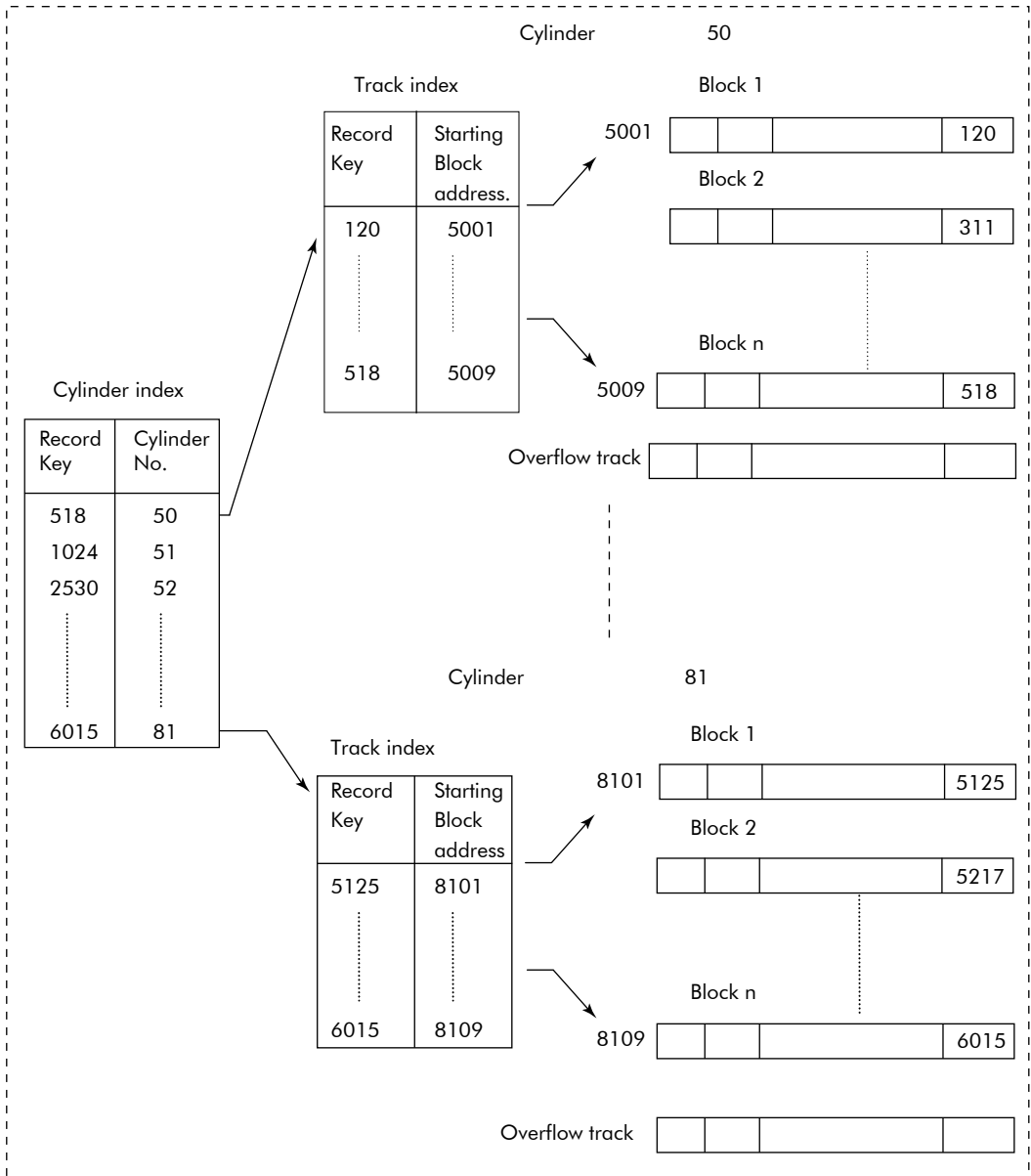


Fig. 9.17 Multilevel indexed file

A sequential file can be used if the activity ratio is high, i.e., more than 60–70 per cent. On the other hand, if the activity ratio is very low, then the direct file can be chosen. The indexed sequential files are less efficient than the direct files for low activity. A comparison between the various files organizations is shown in Figure 9.18.

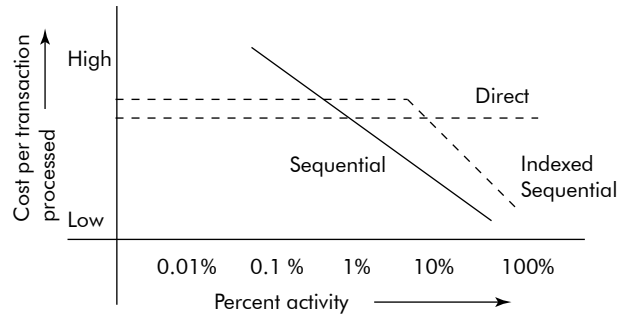


Fig. 9.18 Impact of file activity ration on a file type

2. Response time: The response time can be defined as the amount of time elapsed between a demand for processing and its completion. The response time of a sequential file is always very long because it involves both the waiting time i.e. until a batch of transaction is run and the processing time. The direct and indexed sequential files have a faster response time.

The direct files allow quick retrieval of records. Generally faster response time is comparatively more expensive. Since the direct and indexed files are more up to date, the added cost per transaction is justified.

3. Volatility: It is the measure of percentage of records added to or deleted from a file during its use over a period of time. Since each file organization requires different methods to add or delete records, the volatility becomes a very important factor as far as file processing and its organization is concerned.

In a direct file, the process for adding and deleting records is simple and time saving compared to a sequential file. The indexed sequential files are also very volatile. The direct or indexed files have a problem of reorganization and overflow overheads.

4. Size: The amount of processing required for a file is directly proportional to its size. The size of a file is determined in terms of total number of records in the file. The size of a record is measured in terms of total number of characters in the record.

Small files do not require any painstaking design. On the other hand large ones require careful design. A small sequential file may be stored on a direct access device to avoid waiting time, i.e., mounting and dismounting of tapes is avoided. Large sequential files are to be stored on a tape.* As far as direct files are concerned, the size has no relevance

5. Integration: Generally, a business organization maintains integrated file system that allows online updation of interdependent files. Random and indexed organization allow all the files affected by a particular transaction to be updated online. Sequential files are not suitable for such system of files.

6. Security: A back up is taken during every run of a sequential file. Therefore a sequential file stored is more secured. In fact, sequential files processing provide automatic data backup. The data stored in direct and indexed files are comparatively not secured. Special techniques such as frequent mirroring of disks, are used to insure file integrity. On the other hand the magnetic tapes are more prone to physical damage.

* Nevertheless, the size of available hard disks is so large(more than 500GB) that tapes have become obsolete and redundant.

9.11 GRADED PROBLEMS

Problem 1: A macro processor is a text substitution program where macro consists of a macro name and its definition as shown below:

Macro name	Macro definition (replacement text)
YMCA	YMCA institute of Engineering

The macro processor replaces all occurrences of macro name in a document by its corresponding definition. Write a program that takes a text in which the macros are defined at the beginning by the statements such as given below. The definitions end by the keyword “MEND”.

```
#define CE Computer Engineering
#define PC Personal computer
MEND
<text of the document>
```

The program implements the macro processor, i.e., reads the macros and their definitions and does the text substitution for each macro call.

Solution: The following data structures would be employed to do the required task:

- (1) A structure to store a macro name and its definition.

```
struct macro
{
    char Name [10];
    char Mdef [50];
};
```

- (2) An array of structures of type macro to store all macro definitions.

```
struct macro Mlist [15]
```

The program would require the following functions:

Function	Description
1. get-taken()	: to read a word from the input text document.
2. build-Mtable()	: to build a table of macro names and definitions using the array of structure, i.e., Mlist [].
3. writeout()	: write the output to the output text file.

A complete program that does the required task is given below:

```
/* This program implements a macro processor, a text substitution
program */
#include <stdio.h>
struct macro /* to store a macro */
{
    char Mname[10];
    char Mdef[50];
};
FILE *infile, *outfile;
/* A function to build Macro table */
void build_Mtable(struct macro list[15], FILE *ptr);
/* function to read a word from input text file */
```

```

void get_token (FILE *infile, char token[20], char *break_point , int
*flag);
char inputfile[15], outputfile[15];
/* function to write a word to output text file */
void writeout(FILE *outfile, char token[20],char ch, struct macro
Mlist[15]);
main()
{
    char token[20];
    char break_point;
    int flag, j;
    char ch;
    struct macro Mlist[15] ;
    printf ("\n Enter the name of input file:");
    fflush(stdin); gets(inputfile);
    printf ("\n Enter the name of output file:");
    fflush(stdin); gets(outputfile);
    infile = fopen(inputfile, "r");
    outfile = fopen(outputfile, "w");
    build_Mtable(Mlist, infile);
    flag = 0;
    clrscr();
    /* flag == 3 indicates EOF encountered */
    while (!feof(infile) && flag != 3)
    { flag=0;
      token[0] = '\0';
      ch= '';
      get_token(infile, token, &ch, &flag);
      writeout(outfile, token, ch, Mlist);
    }
    fclose(infile);
    fclose(outfile);
} /* end of main */

void get_token(FILE * infile, char token[30], char *break_pt, int *flag)
{ int i;
  char ch;
  i = -1;
  while (!(*flag))
  { ch = fgetc(infile); if (ch == EOF) { *flag = 3; return; }
    switch(ch)
    {
        /* break points for words or tokens */
        case '\n':
        case ' ' :
        case ',' : /* flag == 1 indicates that a break point occurred */
        case '.' :

```

```

        case ';' : *break_pt = ch; *flag = 1; token[++i] = '\0';
            break;
        default: token[++i] = ch;
    }
}
} /* end of get_token */

void build_Mtable(struct macro list[15], FILE *ptr)
{ char token[20], ch1, temp[2];
  int i;
  token[0]='\0';
  i = -1;
  while (strcmp(token,"MEND"))
  { int flag = 0;
    char ch = '';
    get_token(ptr,token, &ch, &flag);
    if (!strcmp(token, "define"))
    { i++;
      flag=0;
      ch='';
      get_token(ptr,token, &ch, &flag);
      strcpy (list[i].Mname, token);
      list[i].Mdef[0]='\0';
      while (ch != '\n')
      { flag=0;
        ch = '';
        get_token(ptr,token, &ch, &flag);
        strcat(list[i].Mdef , token);
        if (ch != '\n')
        { temp[0] = ch;
          temp [1] = '\0';
          strcat(list[i].Mdef , temp);
        }
      }
    }
    i++;          /* Marks the end of the macro list*/
    strcpy (list[i].Mname,"END");
  } /* end of build_Mtable */

void writeout(FILE *outfile,char token[20], char ch, struct macro
Mlist[15])
{
  int i = 0;
  while (strcmp(Mlist[i].Mname, "END"))
  {

```

```

/* replace macro by its definition */
if (!strcmp(Mlist[i].Mname, token ))
{ fputs(Mlist[i].Mdef, outfile);
  fputc(ch, outfile);
  return;
};
i++;
}
fputs(token, outfile);
fputc(ch, outfile);
}

```

Sample Input:

```

#define ce Computer Engineering
#define ymca YMCA Institute of Engineering
#define aks A. K. Sharma
#define fbd Faridabad
#define MDU Maharishi Dayanand University, Rohtak
MEND

```

ymca is an Institute situated in fbd. It is a Government of Haryana institute. It is affiliated to MDU. The institute is rated as the best engineering college in the state of Haryana. The ce department is the largest department of the institute. aks works in ce department as a Professor.

Sample Output:

YMCA Institute of Engineering is an Institute situated in Faridabad. It is a Government of Haryana institute. It is affiliated to Maharishi Dayanand University, Rohtak. The institute is rated as the best engineering college in the state of Haryana. The Computer Engineering department is the largest department of the institute. A.K. Sharma works in Computer Engineering department as a Professor.

Note: The program does not perform syntax checking, i.e., if the terms “define” and “MEND” are misspelled, the program will go hay wire.

Problem 2: Statements in BASIC language are written in such a manner that each statement starts with an integer number as shown below:

```

10 REM This is a sample statement
20 input A, B
30 goto 150
:
:
:
100 If (x y) goto 20

```

Write a program that reads a file of a BASIC program. For a given offset, it adds the offset to all the statement numbers. The arguments of the goto statements should also be modified to keep the program consistent. In fact, this type of activity is done by a loader, i.e., when it loads a program into the main memory for execution, it adds the offset of the address of the main memory to all address modifiable parts of the program such as to goto or jump statements and the like.

Solution: The `get_token()` and `writeout()` functions of previous program would be used in this program to read a token and to write the tokens to a given file. The following functions of `stdlib.h` would be used to convert a string to integer and vice versa.

- (1) `<int> atoi (string)` : This function takes a string of characters and converts it into an equivalent integer.
- (2) `void itoa (int, string, <radix>)` : This function converts an integer to an equivalent string. Radix is the radix of the integer such as 2, 10 depending upon the number being binary or decimal.

The required program is given below:

```
/* This program implements a relative loader */

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
/* function to read a word from input text file */
void get_token (FILE *infile, char token[20], char *break_point , int
*flag);
char inputfile[15], outputfile[15];
/* function to write a word to output text file */
void writeout(FILE *outfile,char token[20],char ch);
main()
{
    FILE *infile, *outfile;
    char token[20];
    char last_token[20]="";
    char break_point;
    int flag, j, temp;
    int offset;
    char ch, chl;
    printf ("\n Enter the name of input file:");
    fflush(stdin); gets(inputfile);
    printf ("\n Enter the name of output file:");
    fflush(stdin); gets(outputfile);
    printf ("\n Enter the value of offset (int):");
    scanf("%d", &offset);
    infile = fopen(inputfile, "r");
    outfile = fopen(outputfile, "w");
    flag = 0;
    clrscr();
    chl = '\n';
    /* flag == 3 indicates EOF encountered */
    while (!feof(infile) && flag != 3)
    { flag=0;
```



```

    token[0] = '\0';
    ch = ' ';
    get_token(infile, token, &ch, &flag);
    /* Check if it is a number appearing at the
    beginning of a line or as argument of
    goto statement */
    if ((isdigit (token [0]) && chl == '\n' ) || (isdigit (token [0]) &&
    (!strcmp(last_token, "goto")))) )
    { temp = atoi(token);
      temp = temp + offset;
      itoa(temp, token, 10);
    }
    writeout(outfile, token, ch);
    chl = ch;
    strcpy(last_token , token);
  }
  fclose(infile);
  fclose(outfile);
} /* end of main */

void get_token(FILE * infile, char token[30], char *break_pt, int *flag)
{int i;
 char ch;
 i = -1;
 while (!(*flag))
 { ch = fgetc(infile); if (ch == EOF) { *flag = 3; return; }
 switch(ch)
 {
     /* break points for words or tokens */
     case '\n':
     case ' ':
     case ',': /* flag == 1 indicates that a break point occurred */
     case '.':
     case ';': *break_pt = ch; *flag = 1; token[++i] = '\0';
       break;
     default : token[++i] = ch;
   }
 }
} /* end of get_token */

void writeout(FILE *outfile, char token[20], char ch)
{
    fputs(token, outfile);
    fputc(ch, outfile);
}

```

Sample input:

```

10 Rem this is a loader
20 Input A, B
30 If (A 20) goto 20
40 C 5 A 1 5 * 20;
50 if (C 50) goto 100
60 C 5 C 1 B
70 go to 50
100 stop

```

Value of offset 5 50

Sample output:

```

60 Rem this is a loader
70 Input A, B
80 If (A 20) goto 70
90 C 5 A 1 5 * 20;
100 if (C 50) goto 150
110 C 5 C 1 B
120 go to 50
150 stop

```

EXERCISES

1. What are the different methods of opening a file? Explain in brief.
2. Differentiate between `fwrite()` and `fput()` functions.
3. How can the end of a file be detected? Explain in brief.
4. Write a program which counts the number of record in a given file.
5. Give suitable declarations for the following files:
 - a. A file containing components where each component consists of the following information:

Name:	20 characters
Code:	integer type
NET:	float type
 - b. A text file called Book.
7. Write a program which creates an exact copy of a given file called Ofile in the form of a new file called Nfile. Choose appropriate component type for this problem.
8. Write a program which counts the number of times a given alphabet stored in variable `ch` appears in a text file.
9. Write a program which searches a given file for a record equal to an input value of emp-code. The component record structure of the file is as given below:

Emp-code	<div style="border: 1px solid black; width: 150px; height: 25px;"></div>
Total	<div style="border: 1px solid black; width: 100px; height: 25px;"></div>

Choose appropriate data types wherever necessary.

10. Write a program which reads a text file called document and generates a table of frequency count of alphabets appearing in the text. The output should be displayed in the following form:
Frequency count

Alphabet	Count
A	
B	
C	
.	
.	
.	
Z	

11. A student record is defined as given in Example 5. Write a program which reads a file of such records (say class-file) and prints the result in the following format:

S. No.	Condition	No. of Students
1	Marks < 50%	xx
2	Marks > 50% and < 60%	xx
3	Marks >= 60% and < 75%	xx
4	Marks >= 75% and < 90%	xx
5	Marks >= 90	xx

(Assume Max marks in each subject = 100).

12. What is EOF? Explain its utility.
13. Modify Example 11 such that it generates a list of students who secure more than 80 per cent marks.
14. Write an interactive menu driven 'C' program to create a text file and then display the file. Create another text file by converting each line of the newly created text file into a lowercase string. Display the newly created file. In case number of lines exceeds 22, file should be displayed one screen at a time.
15. Write an interactive menu driven 'C' program to create a text file and then display the file. Create another text file by reversing each line of the newly created text file. Display the newly created file. In case number of lines exceeds 22, file should be displayed one screen at a time.
16. Explain relative files in detail.
17. Explain indexed sequential organization.
18. Write an explanatory note on multilevel indexed files.
19. Define file activity ratio, response time, and volatility.
20. How records are edited or deleted from an indexed file?

Advanced Data Structures

CHAPTER OUTLINE

- 10.1 AVL Trees
- 10.2 Sets
- 10.3 Skip Lists
- 10.4 B-Trees
- 10.5 Search by Hashing

10.1 AVL TREES

During the discussion on binary search trees (BST) in Chapter 7, it was appreciated that BST is an excellent tool for fast search operations. It was also highlighted that if the data input to BST is not properly planned, then there is every likelihood that the BST may end up in a skewed binary tree or at least a lopsided binary tree wherein one subtree has more height than the other subtree. Consider the data given below:

Cat Family = {Lion, Tiger, Puma, Cheetah, Jaguar, Leopard, Cat, Panther, Cougar, Lynx, Ocelot}

This data can be provided in any combination and the resultant BSTs would be different from each other depending upon the pattern of input. Some of the possible BSTs produced from the above data are shown in Figure 10.1.

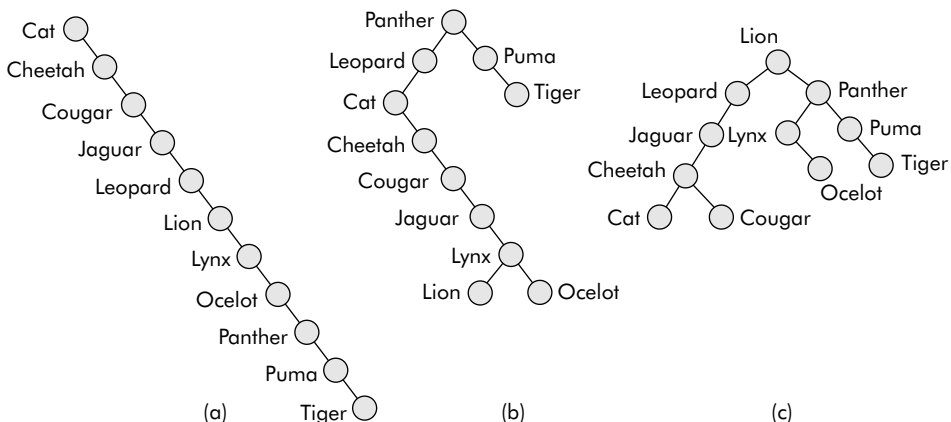


Fig. 10.1 BSTs produced from same data provided in different combinations

It may be noted that the BST of Figure 10.1 (a) is a skewed binary tree and the search within such a data structure would amount to a linear search. The BST of Figure 10.1 (b) is heavily lopsided towards left whereas the BST of Figure 10.1 (c) is somewhat balanced and may lead to a comparatively faster search. If the data is better planned, then definitely an almost balanced BST can be obtained. But the planning can only be done when the data is already with us and is suitably arranged before creating the BST out of it. However, in many applications such as 'symbol processing' in compilers, the data is dynamic and unpredictable and, therefore, creation of a balanced BST is a remote possibility. Hence, the need to develop a technique that maintains a balance between the heights of left and right subtrees of a binary tree has always been felt; the major aim being that whatever may be the order of insertion of nodes, the balance between the subtrees is maintained.

A *height balanced binary tree* was developed by Adelson-Velenskii and Landis, Russian researchers, in 1962 which is popularly known as an 'AVL Tree'. The AVL Tree assumes the following basic subdefinitions:

- (1) The height of a tree is defined as the length of the longest path from the root node of the tree to one of its leaf nodes.
- (2) The balance factor (BF) is defined as given below:

$$BF = \text{height of left subtree } (H_L) - \text{height of right subtree } (H_R)$$

With the above subdefinitions, the AVL Tree is defined as a balanced binary search tree if all its nodes have balance factor (BF) equal to -1 , 0 , or 1 .

In simple words we can say that in AVL Tree, the height of two subtrees of a node differs by at most one. Consider the trees given in Figure 10.2.

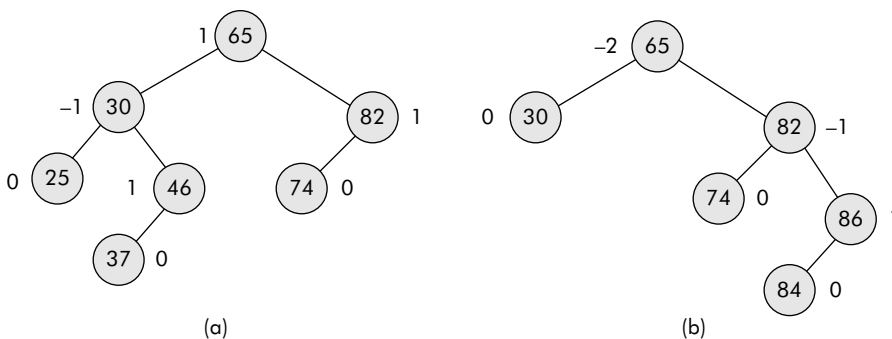


Fig. 10.2 The binary search trees with BF labelled at each node

It may be noted that in Figure 10.2 (a), all nodes have BF within the range, i.e. -1 , 0 , or 1 . Therefore, it is an AVL Tree. However, the root of graph in Figure 10.2 (b) has BF equal to -2 , which is a violation of the rule and hence the tree is not AVL.

Note: A complete binary search tree is always height balanced but a height balanced tree may or may not be a complete binary tree.

The height of a binary tree can be computed by the following simple steps:

- (1) If a child is NULL then its height is 0 .
- (2) The height of a tree is $= 1 + \max(\text{height of left subtree, height of right subtree})$

Based on the above steps, the following algorithm has been developed to compute the height of a general binary tree:

```

Algorithm compHeight (Tree)
{
    if (Tree == NULL)
        {height = 0; return height}
    hl = compHeight (leftChild (Tree));
    hr = compHeight (rightChild (Tree));
    if (hl >= hr) height = hl + 1;
    else
        height = hr + 1;
    return height;
}

```

The following operations can be performed on an AVL Tree:

- (1) Searching an AVL Tree.
- (2) Inserting a node in an AVL Tree.
- (3) Deleting a node from an AVL Tree.

10.1.1 Searching an AVL Tree

An AVL Tree can be searched for a key K by visiting nodes starting from root. The visited node is checked for the presence of K. If found, then the search stops otherwise if value of K is less than the key value of visited node, then the left subtree is searched else the right subtree is searched. The algorithm for AVL search is given below:

```

Algorithm searchAVL (Tree, K)
{
    if (DATA (Tree) == K)
        {prompt " Search successful" ; Stop;}
    if (DATA (Tree) < K)
        searchAVL (leftChild (Tree), K);
    else
        searchAVL (rightChild (Tree), K);
}

```

The search for key (K = 37) will follow the path as given in Figure 10.3.

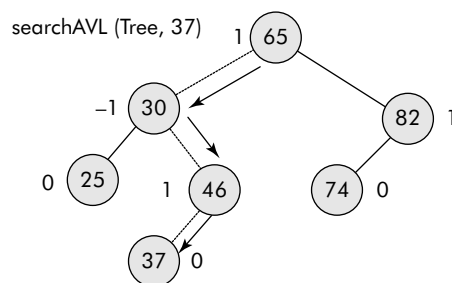


Fig. 10.3 Search for a key within an AVL tree

10.1.2 Inserting a Node in an AVL Tree

Whenever a node is inserted into a subtree which is full and whose height is already 1 more than its sibling, then the subtree becomes unbalanced. This imbalance in a subtree ultimately leads to imbalance of whole of the tree as shown in Figure 10.4. An insertion of node '50' has caused an imbalance in the tree. The nearest ancestor whose BF has become ± 2 is called a pivot. The node 30 is the pivot in the unbalanced tree of Figure 10.4.

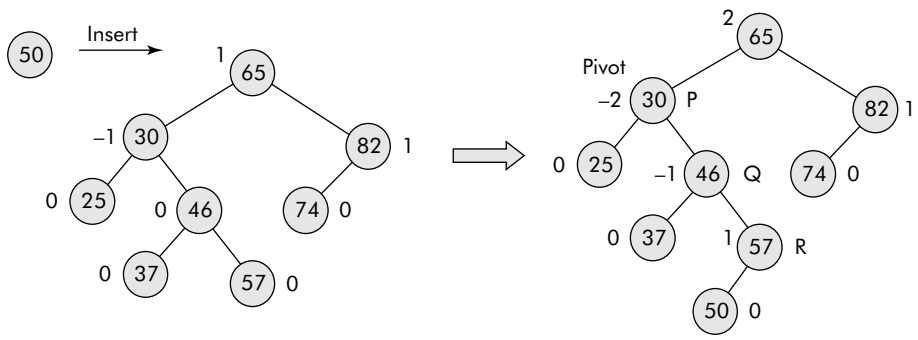


Fig. 10.4 An insertion into a full subtree

The imbalance can be removed by carefully rearranging the nodes of the tree. For instance, the zone of imbalance in the tree is the left subtree with root node 30 (say Node P). Within this subtree, the imbalance is in right subtree with root node 46 (say Node Q). The closest node to the inserted node is 57. Call this node as R. A point worth noting is that P, Q, and R are numbers and can be rearranged as per binary search rule, i.e., the smallest becomes the left child, the largest as the right child and the middle one becomes the root. This rearrangement can eliminate the imbalance in the section as shown in Figure 10.5.

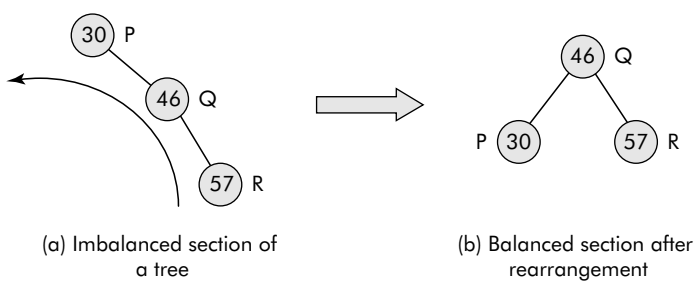


Fig. 10.5 The rearrangement of nodes

It may be noted that imbalanced section of the tree has been rotated to left to balance the subtree. The rotation of a subtree in an AVL Tree is called an AVL rotation. Depending upon, in which side of the pivot a node is inserted, there are four possible types of AVL rotations as discussed in following sections.

10.1.2.1 Insertion of a Node in the Left Subtree of the Left Child of the Pivot In this case, when a node in an AVL Tree is inserted in the left subtree of the left child (say LL) of the pivot then the imbalance occurs as shown in Figure 10.6 (a).

It may be noted that in this case, after the rotation, the pivot P has become the right child, Q has become the root. Q_R , the right child of Q has become the left child of P. The Q_L , the left child of Q remains intact.

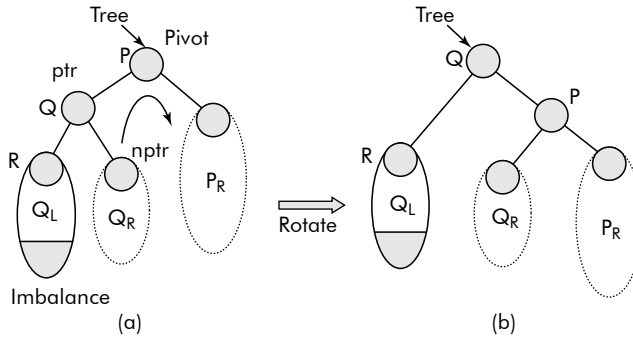


Fig. 10.6 AVL rotation towards right

The algorithm for this rotation is given below:

/* The Tree points to the Pivot */

Algorithm rotateLL (Tree)

```

{
    ptr = leftChild (Tree); /* point ptr to Q */
    nptr = rightChild (ptr); /* point nptr to right child of Q */
    rightChild (ptr) = Tree; /* attach P as right child of Q */
    leftChild (Tree) = nptr; /* attach right child of Q as left child of P */
    Tree = ptr; /* Q becomes the root */
    return Tree;
}

```

Consider the AVL Tree given in Figure 10.7 (a). The tree has become imbalanced after the insertion of 12 [see Figure 10.7 (b)].

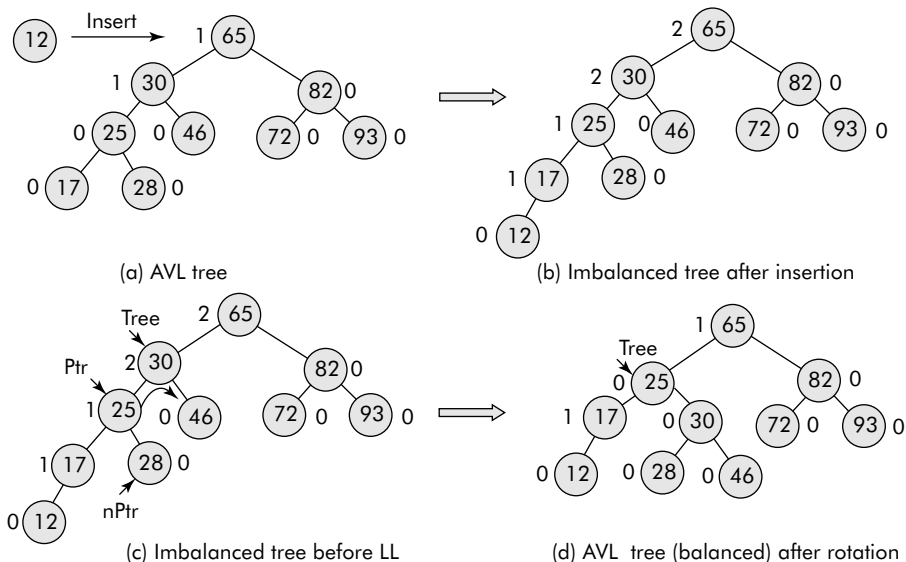


Fig. 10.7 The AVL rotation (LL)

Figure 10.7 (c) shows the application of pointers on the nodes of the tree as per the AVL rotation. Figure 10.7 (d) shows the final AVL Tree (height balanced) after the AVL rotation, i.e., after the rearrangement of nodes.

10.1.2.2 Insertion of a Node in the Right Subtree of the Right Child of the Pivot In this case, when a node in an AVL Tree is inserted in the right subtree of the right child (say RR) of the pivot then the imbalance occurs as shown in Figure 10.8 (a).

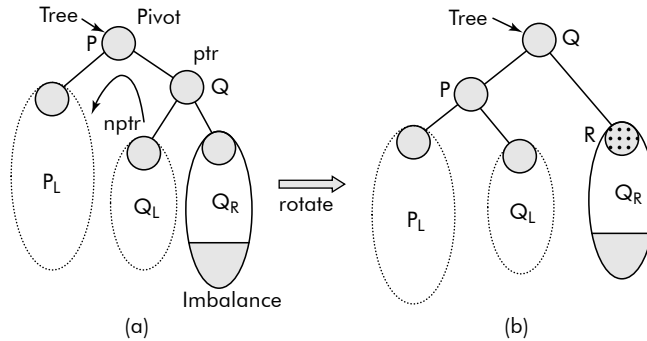


Fig. 10.8 AVL rotation towards left

It may be noted that in this case, after the rotation, the pivot P has become the left child, Q has become the root. Q_L , the left child of Q (i.e., Q_L) has become the right child of P. The Q_R , the right child of Q remains intact.

The algorithm for this rotation is given below:

/ The Tree points to the Pivot */*

Algorithm rotateRR (Tree)

```
{
    ptr = rightChild (Tree);    /* point ptr to Q */
    nptr = leftChild (ptr);    /* point nptr to left child of Q */
    leftChild (ptr) = Tree;    /* attach P as left child of Q */
    rightChild (Tree) = nptr;  /* attach left child of Q as right child of P */
    Tree = ptr;                /* Q becomes the root */
    return Tree;
}
```

Consider the AVL Tree given in Figure 10.9 (a). The tree has become imbalanced after the insertion of 100 (see Figure 10.9 (b)).

Figure 10.9 (c) shows the application of pointers on the nodes of the tree as per the AVL rotation. Figure 10.9 (d) shows the final AVL Tree (height balanced) after the AVL rotation, i.e., after the rearrangement of nodes.

10.1.2.3 Insertion of a Node in the Right Subtree of the Left Child of the Pivot In this case, when a node in an AVL Tree is inserted in the right subtree of the left child (say LR) of the pivot then the imbalance occurs as shown in Figure 10.10 (a).

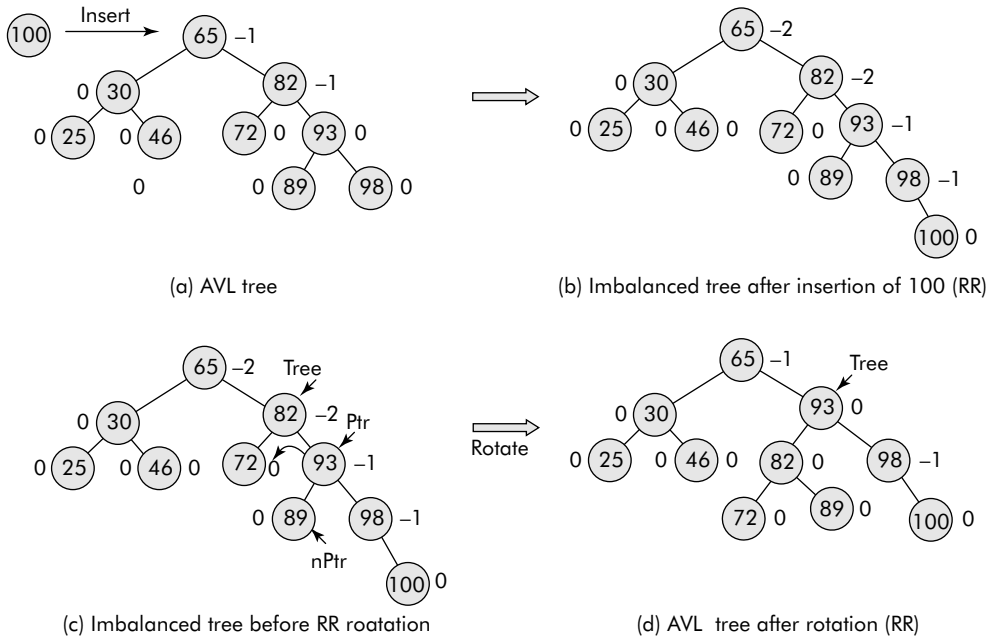


Fig. 10.9 The AVL rotation (RR)

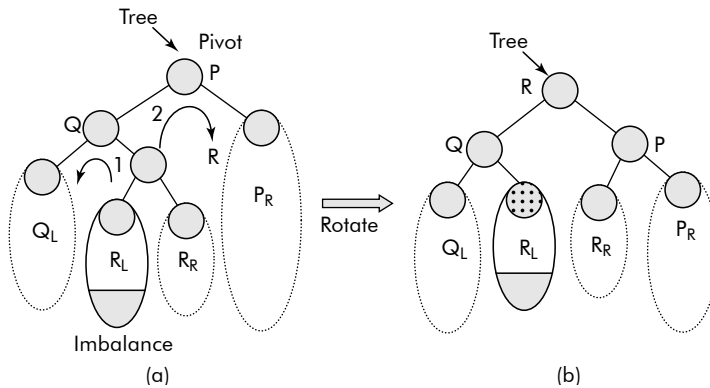


Fig. 10.10 The double rotation Left-Right (LR)

This case requires two rotations: rotation 1 and rotation 2.

Rotation 1: In this rotation, the left child of R becomes the right child of Q and R becomes the left child of P. Q becomes the left child of R. The right child of R remains intact.

Rotation 2: In this rotation, the right child of R becomes the left child of P. R becomes the root. P becomes the right child of R.

The final balance tree after the double rotation is shown in Figure 10.10 (b).

The algorithm for this double rotation is given below:

```
/* The Tree points to the Pivot */
```

```
Algorithm rotate LR (Tree)
```

```
{
    ptr = leftChild (Tree);    /* point ptr to Q */
    rotateRR (ptr);           /* perform the first rotation */
    ptr = leftChild (Tree);    /* point ptr to R */
    rotateLL (ptr);
}
```

The trace of the above algorithm is shown in Figure 10.11 wherein the AVL Tree has become imbalanced because of insertion of node 50 in the right subtree of left child of pivot [see Figures 10.11 (a) and (b)].

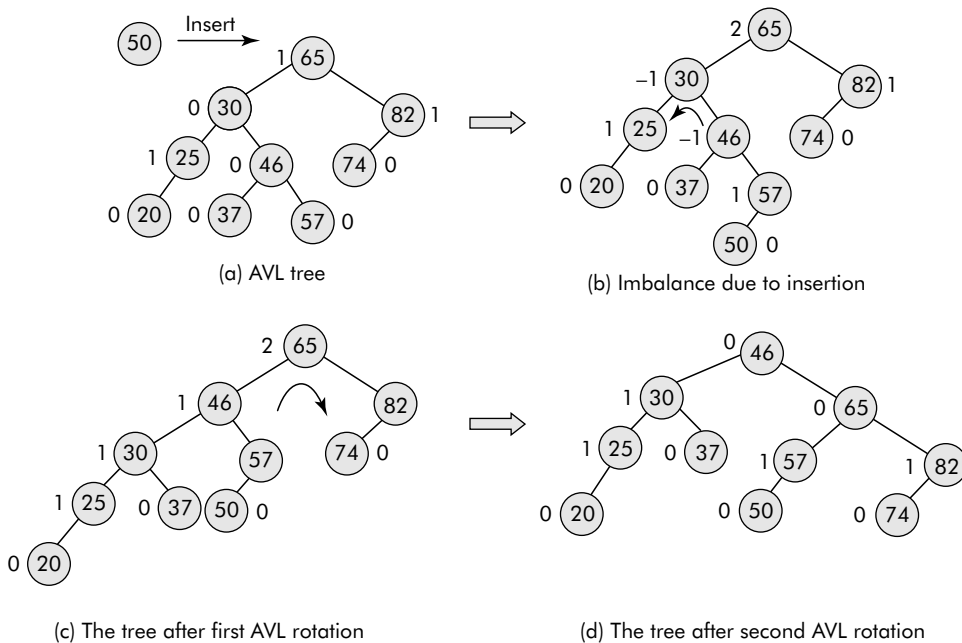


Fig. 10.11 The trace of double rotation on an imbalanced AVL tree

After the double rotation, the tree has again become height balanced as shown in Figures 10.11 (c) and (d).

10.1.2.4 Insertion of a Node in the Left Subtree of the Right Child of the Pivot In this case, when a node in an AVL Tree is inserted in the left subtree of the right child (say LR) of the pivot then the imbalance occurs as shown in Figure 10.12 (a).

This case also requires two rotations: rotation 1 and rotation 2.

Rotation 1: In this rotation, the right child of R becomes the left child of Q and R becomes the right child of P. Q becomes the right child of R. The left child of R remains intact.

Rotation 2: In this rotation, the left child of R becomes the right child of P. R becomes the root. P becomes the left child of R.

The final balance tree after the double rotation is shown in Figure 10.12 (b).

Following is the algorithm for this double rotation:

```
/* The Tree points to the Pivot */
```

```
Algorithm rotate RL (Tree)
```

```
{
    ptr = rightChild (Tree);    /* point ptr to Q */
    rotateLL (ptr);             /* perform the first rotation */
    ptr = rightChild (Tree);    /* point ptr to R */
    rotateRR (ptr);
}
```

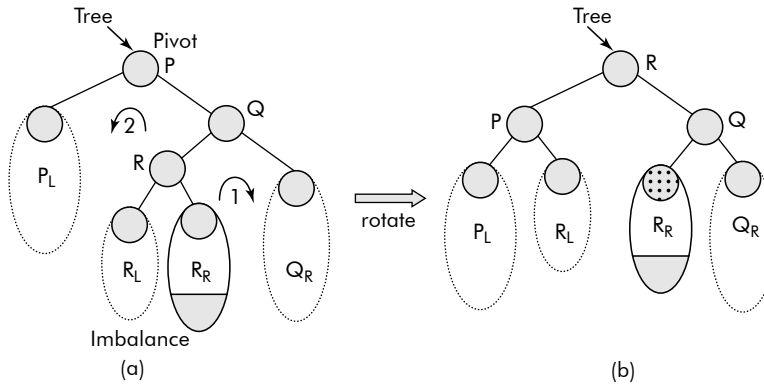


Fig. 10.12 The double rotation Right-Left (RL)

The trace of the above algorithm is shown in Fig. 10.13 wherein the AVL tree has become imbalanced because of insertion of node 92 the left subtree of right child of pivot (See Fig. 10.13(a), (b)). After the double rotation, the tree has again become height balanced as shown in Fig. 10.13(c), (d).

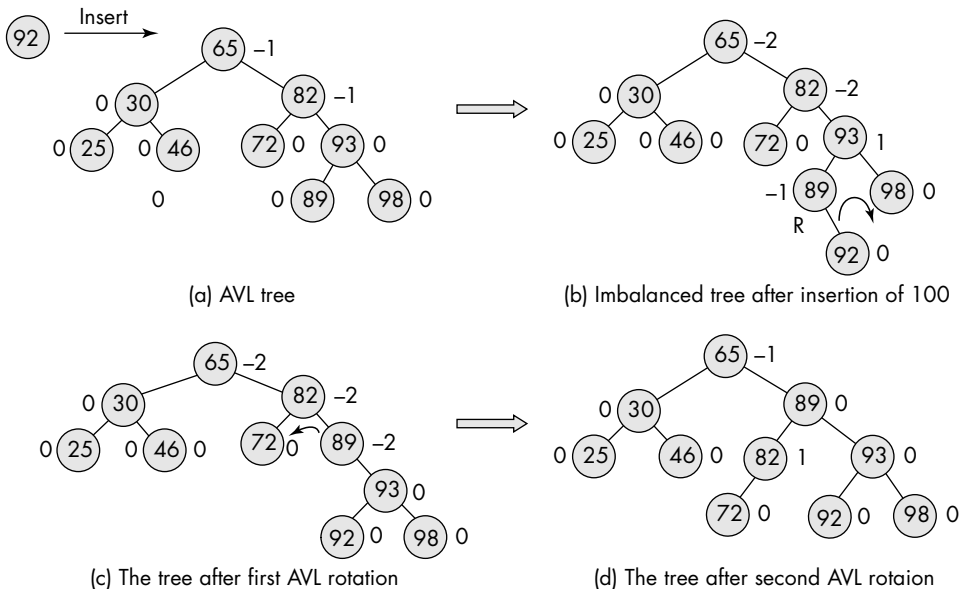


Fig. 10.13 The trace of double rotation on an imbalanced AVL tree

10.2 SETS

A set is an unordered collection of homogeneous elements. Each element occurs at most once with the following associated properties:

- All elements belong to the Universe. The Universe is defined as “all potential elements of set”.
- An element is either a member of the set or not.
- The elements are unordered.

Some examples of sets are given below.

Cat_Family = {Lion, Tiger, Puma, Cheetah, Jaguar, Leopard, Cat, Panther, Cougar, Lynx, Ocelot}

Fruits = {mango, orange, banana, grapes, apple, guava, peach}

LAN = {Comp, Mech, Elect, Civil, Accts, Estb, Mngmt, Hostel}

Note: The LAN is a set of nodes of a local area network of an educational institute spread over various departments like computer engineering, mechanical engineering, electrical engineering, etc.

The basic terminology associated with sets is given below:

- **Null set:** If a set does not contain any element, then the set is called empty or null set. It is represented by Φ .
- **Subset:** If all elements of a set S_1 are contained in set S_2 , then S_1 is called a subset of S_2 . It is denoted by $S_1 \subseteq S_2$.

Example: The set of *big_Cat* = {Lion, Tiger, Panther, Cheetah, Jaguar} is a subset of the set of *Cat_Family*.

- **Union of sets:** The union of two sets S_1 and S_2 is obtained by collecting all members of either set without duplicate entries. This is represented by $S_1 \cup S_2$.

Example: If $S_1 = \{1, 3, 4\}$, $S_2 = \{4, 5, 6, 7, 8\}$,
then $S_1 \cup S_2 = \{1, 3, 4, 5, 6, 7, 8\}$.

- **Intersection of sets:** The intersection of two sets S_1 and S_2 is obtained by collecting common elements of S_1 and S_2 . This is represented by $S_1 \cap S_2$.

Example: If $S_1 = \{1, 3, 4\}$, $S_2 = \{4, 5, 6, 7, 8\}$,
then $S_1 \cap S_2 = \{4\}$.

- **Disjoint sets:** If two sets S_1 and S_2 do not have any common elements, then the sets are called disjoint sets, i.e., $S_1 \cap S_2 = \Phi$.

Example: $S_1 = \{1, 3\}$, $S_2 = \{4, 5, 6, 7, 8\}$ are disjoint sets.

- **Cardinality:** Cardinality of a set is defined as the number of unique elements present in the set. The cardinality of a set S is represented as $|S|$.

Example: If $S_1 = \{1, 3, 4\}$, $S_2 = \{4, 5, 6, 7, 8\}$,
then $|S_1| = 3$ and $|S_2| = 5$.

- **Equality of sets:** If S_1 is subset of S_2 and S_2 is subset of S_1 , then S_1 and S_2 are equal sets. This means that all elements of S_1 are present in S_2 and all elements of S_2 are present in S_1 . The equality is represented as $S_1 \subseteq S_2$.

Example: If $S_1 = \{1, 3, 4\}$, $S_2 = \{3, 1, 4\}$,
then $S_1 \equiv S_2$.

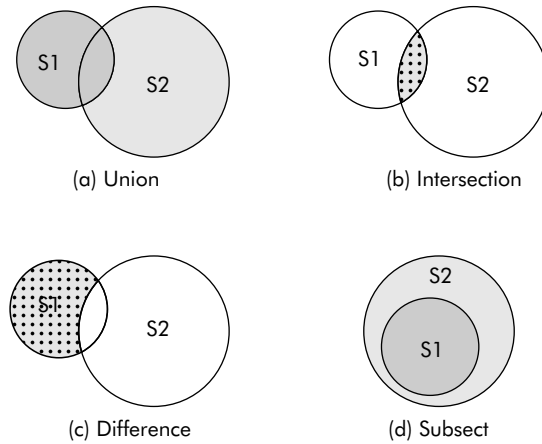


Fig. 10.14 Various set operations

- **Difference of sets:** Given two sets $S1$ and $S2$, the difference $S1 - S2$ is defined as the set having all the elements which are in $S1$ but not in $S2$.

Example: If $S1 = \{1, 3, 4\}$, $S2 = \{3, 2, 5\}$,
then $S1 - S2 = \{1, 4\}$.

The graphic representation of various operations related to sets is shown in Figure 10.14.

- **Partition:** It is a collection of disjoint sets belonging to same Universe. Thus, the union of all the disjoint sets of a partition would comprise the Universe.

10.2.1 Representation of Sets

A set can be represented by the following methods:

- List representation
- Hash table representation.
- Bit vector representation
- Tree representation

List and hash table representations are more popular methods as compared to other methods. A brief discussion on list and hash table representations is given in following sections.

10.2.1.1 List Representation A set can be very comfortably represented using a linear linked list. For example, the set of fruits can be represented as shown in Figure 10.15.

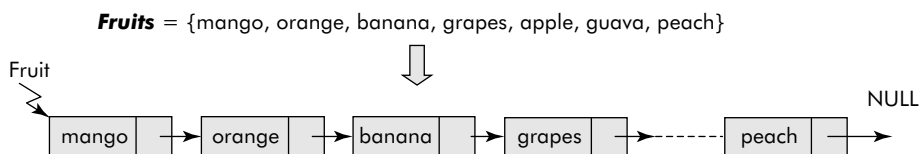


Fig. 10.15 The list representation of a set

Since the set is an unordered collection of elements, linked list is the most suitable representation for the sets because elements can be dynamically added or deleted from a set.

10.2.1.2 Hash Table Representation Hash table representation of a set is also a version of linked representation. A hash table consists of storage locations called buckets. The size of a bucket is arbitrary, i.e., it can hold any number of elements as it stores them into a linked list. Consider the following set:

Tokens = {126, 235, 100, 317, 68, 129, 39, 423, 561, 222, 986}

Let us store the above set in a hash table of five buckets with hash function as given below:

$$H(K) = K \bmod 5$$

Where, K is an element;

H(K) gives the bucket number in which the element K is to be stored.

The arrangement is shown in Figure 10.16.

Tokens = {126, 235, 100, 317, 68, 129, 39, 423, 561, 222, 986}

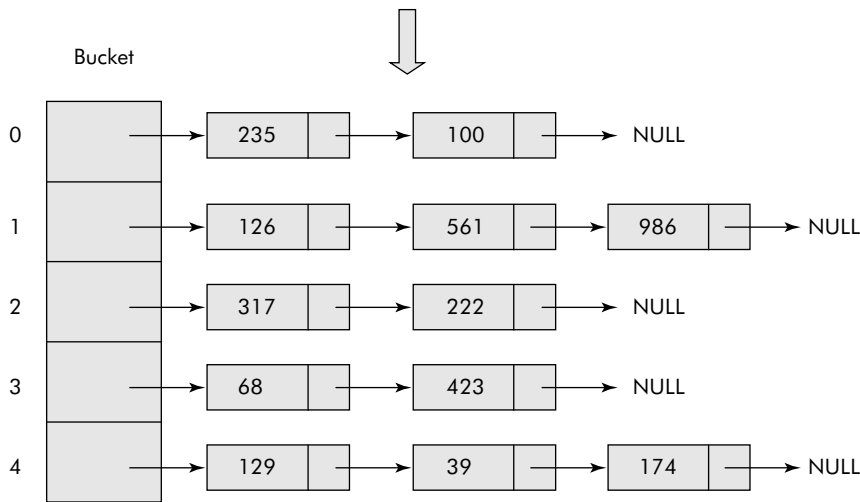


Fig. 10.16 Hash table representation of a set

It may be noted that each bucket is having a linked list consisting of nodes containing a group of elements from the set.

10.2.2 Operations on Sets

The following operations can be defined on sets:

- (1) Union
- (2) Intersection
- (3) Difference
- (4) Equality

A brief discussion on each operation is given in the sections that follow.

10.2.2.1 Union of Sets As per the definition, given two sets S1 and S2, the union of two sets is obtained by collecting all the members of either set without duplicate entries. Let us assume that the sets have been represented using lists as shown in Figure 10.17. The union set S3 of S1 and S2 has been obtained by the following steps:

- (1) Copy the list S1 to S3.
- (2) For each element of S2, check if it is member of S1 or not and if it is not present in S1 then attach it at the end of S3.

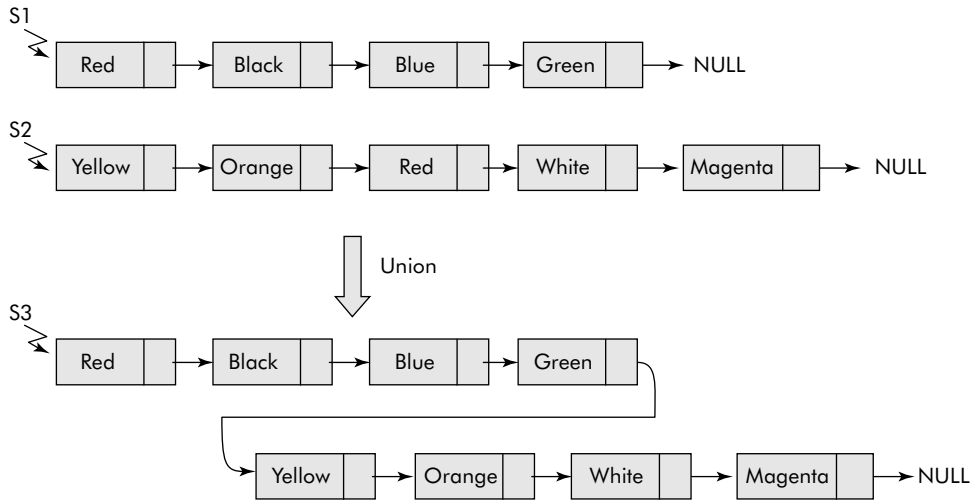


Fig. 10.17 Union of sets

The algorithm for union of two sets S1 and S2 is given below:

```

/* This algorithm uses two functions: copy() and ifMember(). The copy ()
   function is used to copy a set to another empty set (say S1 to S3). The
   ifMember function is employed to test the membership of an element in
   a given set */

```

Algorithm unionOfSets (S1, S2)

```

{
    S3 = Null;
    S3 = copy (S1, S3, last);    /* copy Set S1 to S3 */
    ptr = S2;
    while (ptr != Null)
    {
        Res = ifMember (ptr, S1);    /* Check membership of ptr in S1 */
        if (Res == 0)
        {    /* if the node of S2 is not member of S1 */

```



```

        nptr = new node;
        DATA (nptr) = DATA (ptr)
        NEXT (nptr) = NULL;
        NEXT (last) = nptr;
        last = nptr;
    }
    ptr = Next (ptr);
}
return S3;
}

/* This function is used to copy a set to another empty set, i.e., Set1
to Set3 */

```

Algorithm copy (S1, S3)

```

{
    ptr = S1;
    back = S3;
    ahead = Null;
    while (ptr != Null)
    { ahead = new node;
      DATA (ahead) = DATA (ptr);
      NEXT (ahead) = Null;
      if (back == Null) /* first entry into S3 i.e. S3 is also Null */
      {
          back = ahead;
          S3 = ahead;
      }
      else
      {
          NEXT (back) = ahead;
          back = ahead;
      }
      ptr = NEXT (ptr);
    }
    return S3;
}

/* This function is employed to test the membership of an element pointed
by ptr, in a given set S */

```

Algorithm ifMember (ptr, S)

```

{
    Flag = 0;
    nptr = S;
    while (nptr != Null)
    {

```

```

    if (DATA (ptr) == DATA (nptr))
    {
        Flag = 1;
        break;
    }
    nptr = NEXT (nptr);
}
return Flag;
}

```

10.2.2.2 Intersection of Sets As per the definition, given two sets S1 and S2, the intersection of two sets is obtained by collecting all the common members of both the sets. Let's assume that the sets have been represented using lists as shown in Figure 10.18. The intersection set S3 of S1 and S2 has been obtained by the following steps:

- (1) Initialize S3 to Null;
- (2) last = S3;
- (3) For each element of S2, check if it is member of S1 or not and if it is present in S1 then attach it at the end of S3.

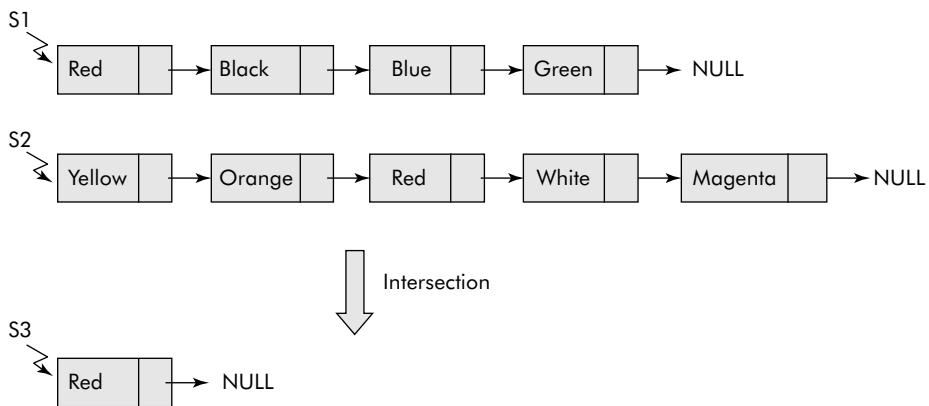


Fig. 10.18 Intersection of sets

The algorithm for intersection of two sets S1 and S2 is given below:

```

/* This algorithm uses the function ifMember() which tests the membership
   of an element in a given set */

```

Algorithm intersectionOfSets (S1, S2)

```

{
    S3 = Null;
    last = S3;
    ptr = S2;
    while (ptr != Null)
    {
        Res = ifMember (ptr, S1);    /* Check membership of ptr in S1 */
    }
}

```

```

if (Res == 1)
{
    /* if the node of S2 is member of S1 */
    nptr = new node;
    DATA (nptr) = DATA (ptr);
    NEXT (nptr) = NULL;
    If (last == Null) /* S3 is empty and it is first entry */
    {
        last = nptr;
        S3 = last;
    }
    else
    {
        NEXT (last) = nptr;
        last = nptr;
    }
}
ptr = Next (ptr);
}
return S3;
}

```

Note: The `ifMember()` function is already defined in the previous section.

10.2.2.3 Difference of Sets As per the definition, given two sets $S1$ and $S2$, the difference $S1 - S2$ is obtained by collecting all the elements which are in $S1$ but not in $S2$. Let us assume that the sets have been represented using lists as shown in Figure 10.19. The difference set $S3$, i.e., $S1 - S2$ has been obtained by the following steps:

- (1) Initialize $S3$ to `Null`;
- (2) `last = Null`;
- (3) For each element of $S1$, check if it is a member of $S2$ or not and if it is not present in $S2$ then attach it at the end of $S3$.

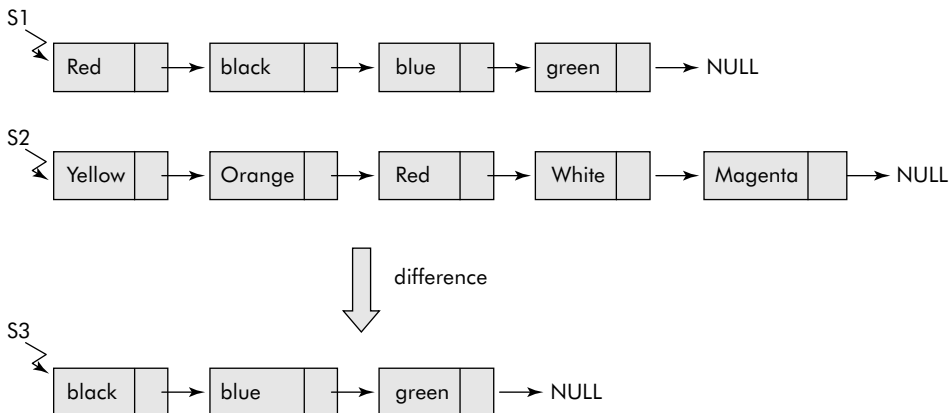


Fig. 10.19 Difference of sets

The algorithm for difference of two sets S1 and S2 is given below:

```
/* This algorithm uses the function ifMember() which tests the membership
   of an element in a given set */
```

Algorithm differenceOfSets (S1, S2)

```
{
    S3 = Null;
    last = S3;
    ptr = S1;
    while (ptr != Null)
    {
        Res = ifMember (ptr, S2);    /* Check membership of ptr in S2 */
        if (Res == 0)
        { /* if the node of S1 is not member of S2 */
            nptr = new node;
            DATA (nptr) = DATA (ptr)
            NEXT (nptr) = NULL;
            If (last == Null)    /* S3 is empty and it is first entry */
            {
                last = nptr;
                S3 = last;
            }
            else
            {
                NEXT (last) = nptr;
                last = nptr;
            }
        }
        ptr = Next (ptr);
    }
    return S3;
}
```

Note: The ifMember () function is already defined in the previous section.

10.2.2.4 Equality of Sets As per the definition, given two sets S1 and S2, if S1 is subset of S2 and S2 is subset of S1 then S1 and S2 are equal sets. Let us assume that the sets have been represented using lists. The equality $S1 \equiv S2$ has been obtained by the following steps:

- (1) For each element of S1, check if it is a member of S2 or not and if it is not present in S2 then report failure.
- (2) For each element of S2, check if it is a member of S1 or not and if it is not present in S1 then report failure.
- (3) If no failure is reported in step1 and step 2, then infer that $S1 \equiv S2$.

The algorithm for equality of two sets S1 and S2 is given in the following:

```
/* This algorithm uses the function ifMember() which tests the membership
   of an element in a given set */
```

```

Algorithm equalSets (S1, S2)
{
    ptr = S1;
    flag = 1;
    while (ptr != Null)
    {
        Res = ifMember (ptr, S2);    /* Check membership of ptr in S2 */
        if (Res == 0)
        {flag = 0;
        break;
        }
        ptr = NEXT (ptr);
    }
    if (flag == 0) {prompt "not equal"; exit ();}
    ptr = S2;
    while (ptr != Null)
    {
        Res = ifMember (ptr, S1);    /* Check membership of ptr in S1 */
        if (Res == 0)
        {flag = 0;
        break;
        }
        ptr = NEXT (ptr);
    }
    if (flag == 0) {prompt "not equal"; exit ();}
    else
        prompt "Sets Equal";
}

```

Note: The `ifMember()` function is already defined in the previous section.

10.2.3 Applications of Sets

There are many applications of sets and a few of them are given below:

- (1) **Web linked pages:** For each item displayed to user, Google offers a link called “*Similar*” pages (see Figure 10.20), which is in fact the list of URLs related to that item. This list can be maintained as a set of related pages by using Union operation.

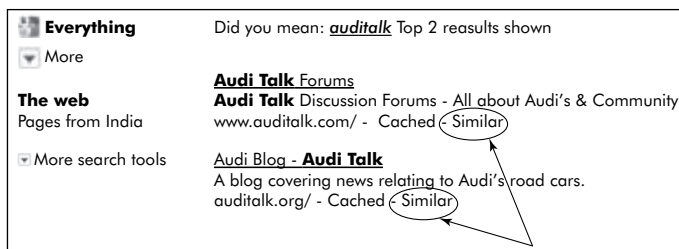


Fig. 10.20 The link to similar URLs

Consider the web pages visited by a user as shown in Figure 10.21.

The arrows between the various pages indicate the link between them. Initially, all pages are stored as singletons in the order of their visit by the users as given below:

$\{P_4\}, \{P_{11}\}, \{P_7\}, \{P_6\}, \dots, \{P_5\}$

Afterwards, the related pages are grouped together by union of the participating singletons as shown below:

- $\{P_4\} \cup \{P_7\} \cup \{P_{11}\} \cup \{P_9\}$
- $\{P_6\} \cup \{P_{12}\} \cup \{P_4\}$
- $\{P_5\}$

We get the following three related partitions, i.e., the disjoint sets:

- P_4, P_7, P_{11}, P_9
- P_6, P_{12}, P_4
- P_5

From the above representation, it can be found

as to whether two elements belong to the same set or not. For example, P_6 and P_{12} belong to the same set. A click on page P_{11} would produce a list of links to P_4, P_7 and P_9 , grouped as “Similar” pages to P_{11} .

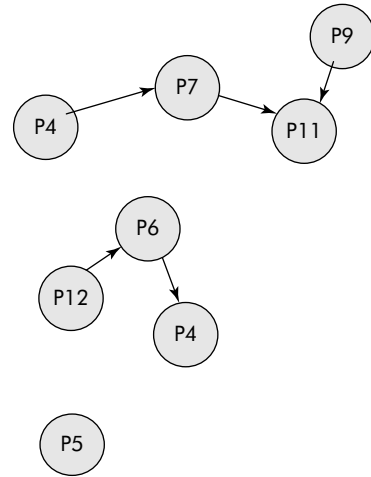


Fig. 10.21

The web pages visited by a user

(2) **Colouring black and white photographs:** This is an interesting application in which an old black and white photograph is converted into a coloured one. The method is given below:

- (i) The black and white photograph is digitized, i.e., it is divided into pixels of varying shades of grey, i.e., ranging from white to black.
- (ii) Pixels belonging to the same part or component are called equivalent pixels. For example, pixels belonging to eyeballs are equivalent. Similarly, pixels belonging to the shirt of a person are equivalent.
- (iii) The equivalence is decided by comparing the grey levels of adjacent pixels.
- (iv) In fact, 4 pixels are taken at a time to find the equivalent pixels among the adjacent pixels. This activity is also called 4-pixel square scan.
- (v) The scanning starts from left to right and top to bottom.
- (vi) Each pixel is assigned a colour.
- (vii) The equivalent pixels are given the same colour.
- (viii) If two portions with different coloured pixels meet each other, then it is considered as part of the same component and then both the portions are coloured with the same colour chosen from either of the colours as shown in Figure 10.22.

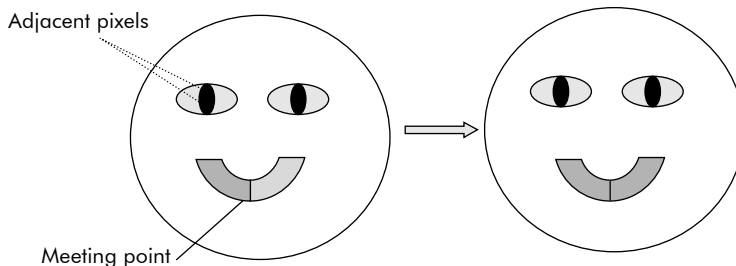


Fig. 10.22

The colouring of black and white photographs

It may be noted that the equivalence relation between the pixels is obtained through set operations where an equivalence relation R over a set S can be seen as a partitioning of S into disjoint sets.

- (3) **Spelling checker:** A spelling checker for a document editor is another interesting application of sets, especially the hash table representation. A normal dictionary is maintained using hash table representation of sets. A simple-most dictionary (**Dict.**) will have 26 buckets because there are 26 alphabets in English language. Similarly, the words of a document (**Doc**) would also be represented in the same fashion as shown in Figure 10.23. A difference operation **Doc** - **Dict** would produce all those words which are in document but not in dictionary, i.e., the misspelled words.

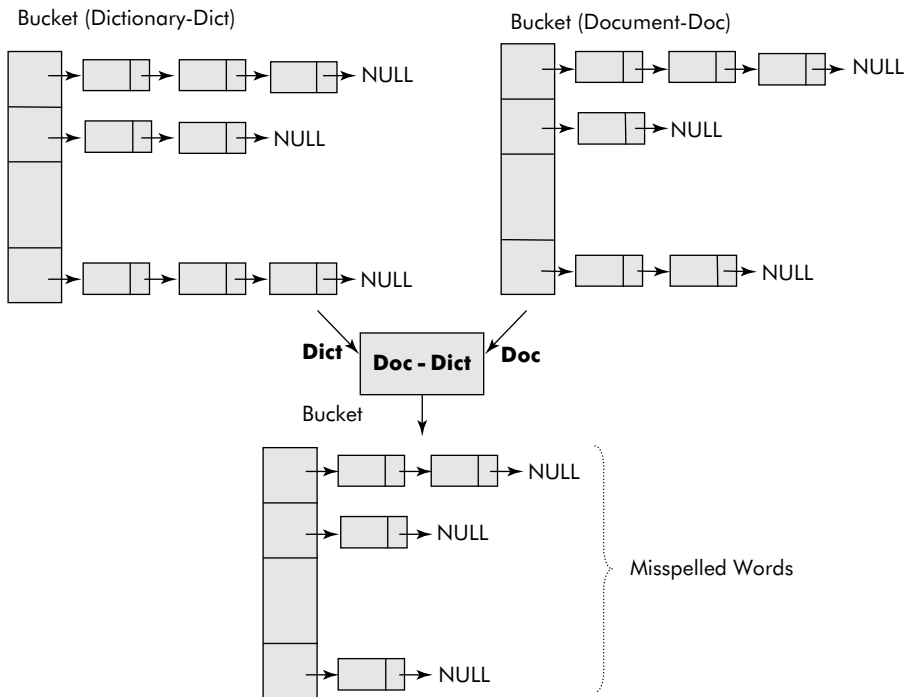


Fig. 10.23 The working of spelling checker

10.3 SKIP LISTS

Search is a very widely used operation on lists. A one-dimensional array is most suitable data structure for a list that does not involve insertion and deletion operations. If the list is unsorted, linear search is used and for a sorted list, binary search is efficient and takes $O(\log n)$ operations.

For random input, AVL Trees can be used to search, insert, and delete numbers with time complexity as $O(\log n)$. However, the balancing of AVL Trees requires rotations in terms of extra $O(\log n)$ operations. Binary search trees tend to be skewed for random input.

However, when insertion and deletion operations are important and the search is also mandatory then a list can be represented using a sorted linked list. A search operation on a sorted linked list takes $O(n)$ operations.

Now, the question is how to make efficient search operation on sorted linked lists? In 1990, Bill Pugh proposed an enhancement on linked lists and the new data structure was termed as *skip list*.

A skip list is basically a sorted linked list in ascending order. Now, extra links are added to selected nodes of the linked list so that some nodes can be skipped while searching within the linked list so that the overall search time is reduced. Consider the normal linked list shown in Figure 10.24. Let us call this list as level 0.

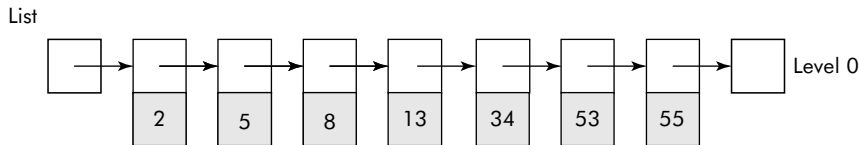


Fig. 10.24 A sorted linked list

It may be noted that the time complexity of various operations such as *search*, *insertion*, *deletion*, etc. on a sorted linked list is $O(N)$.

Let us now add extra links in such a manner that every alternative node is skipped. The arrangement is shown in Figure 10.25. The chain of extra links is called level 1. The head node is said to have height equal to 1.

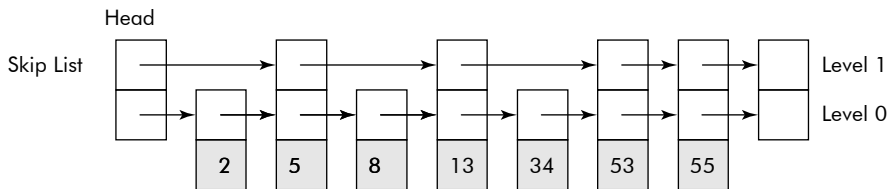


Fig. 10.25 A skip list with level 1

Now, we carry on to add another level of links, i.e., level 2 wherein every alternate node of level 1 is skipped as shown in Figure 10.26. The height of head node has become 2.

The levels of links are added till we reach a stage where the link from head node has reached almost to the middle element of the linked list as has happened in Figure 10.26.

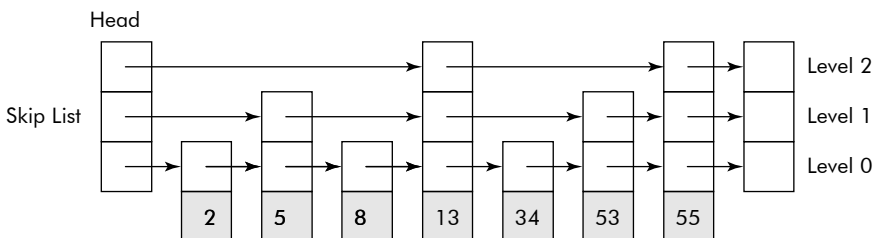


Fig. 10.26 A skip list with level 2

It may be noted that in level 1, links point to every second element in the list. In level 2, the links point to every fourth element in the list. Thus, it can be deduced that a link in the i th level will point to 2^i th element in the list. **A point worth noting is that the last node of the skip list is pointed by links of all levels.**

Now the search within the skip list shall take $O(\log n)$ time as it will follow binary search pattern. For example, the search for '53' will follow the route shown in Figure 10.27.

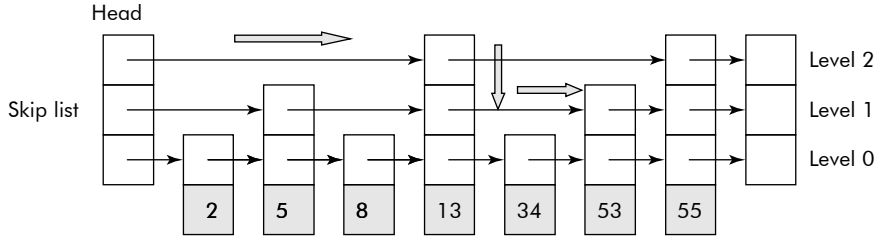


Fig. 10.27 The path followed for searching '53' in the list

The algorithm that searches a value VAL in the skip list is given below:

Algorithm skipListSearch ()

```
{
    curPtr = Head;
    For (I = N; I > 1; I--)
    {
        while (DATA (curPtr[i] < VAL)
            CurPtr[I] = NEXT (CurPtr [I]);
        If (DATA (curPtr[I])) == VAL) ; return success;
    }
    return failure;
}
```

It may be noted that insertion and deletion operations in the list will disturb the balance of the skip list and many levels of pointer adjustment would be required to redistribute the levels of pointers and heights of head and intermediate nodes.

10.4 B-TREES

A binary search tree (BST) is an extremely useful data structure as far as search operations on static data items is concerned. The reason is that static data can be so arranged that the BST generated out of the data is almost height balanced. However, for very large data stored in files, this data structure becomes unsuitable in terms of number of disk accesses required to read the data from the disk. The remedy is that each node of BST should store more than one record to accommodate the complete data of the file. Thus, the structure of BST needs to be modified. In fact, there is a need to extend the concept of BST so that huge amount of data for search operations could be easily handled.

We can look upon a BST as a two-way search tree in which each node (called root) has two subtrees—left subtree and right subtree, with following properties:

- (1) The key values of the nodes of the left subtree are always less than the key value of the root node.
- (2) The key values of the nodes of the right subtree are always more than the key value of the root node.

The concept of two-way search tree can be extended to create an m -way search tree having node structure shown in Figure 10.28.

The m -way tree has following properties:

- (1) Each node has any number of children from 2 to M , i.e., all nodes have degree $\leq M$, where $M \geq 2$.
- (2) Each node has keys (K_1 to K_n) and pointers to its children (P_0 to P_n), i.e., number of keys is one less than the number of pointers. The keys are ordered, i.e., $K_i < K_{i+1}$ for $1 \leq i < n$.
- (3) The subtree pointed by a pointer P_i has key values less than the key value of K_{i+1} for $1 \leq i < n$.
- (4) The subtree pointed by a pointer P_n has key values greater than the key value of K_n .
- (5) All subtrees pointed by pointers P_i are m -way trees.

It may be noted that all keys in the subtree to the left of a key are predecessors of the key and that on the right are successors of the key as shown in Figure 10.29. The m -way tree shown in the figure is of the order 3, i.e., $m = 3$.

A close look at the m -way tree suggests that it is a multilevel index structure. In order to have efficient search within an m -way tree, it should be height balanced. In fact, **a height balanced m -way tree is called as B-tree**. A precise definition of a B-tree is given below.

A B-tree of order m , has the following properties:

- (1) The root has at least two children. If the tree contains only a root, i.e., it is a leaf node then it has no children.
- (2) An interior node has between $\lfloor m/2 \rfloor$ and m children.
- (3) Each leaf node must contain at least $\lfloor m/2 \rfloor - 1$ children.
- (4) All paths from the root to leaves have the same length, i.e., all leaves are at the same level making it height balanced.

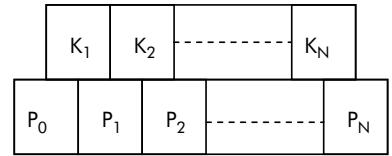


Fig. 10.28 M-way tree

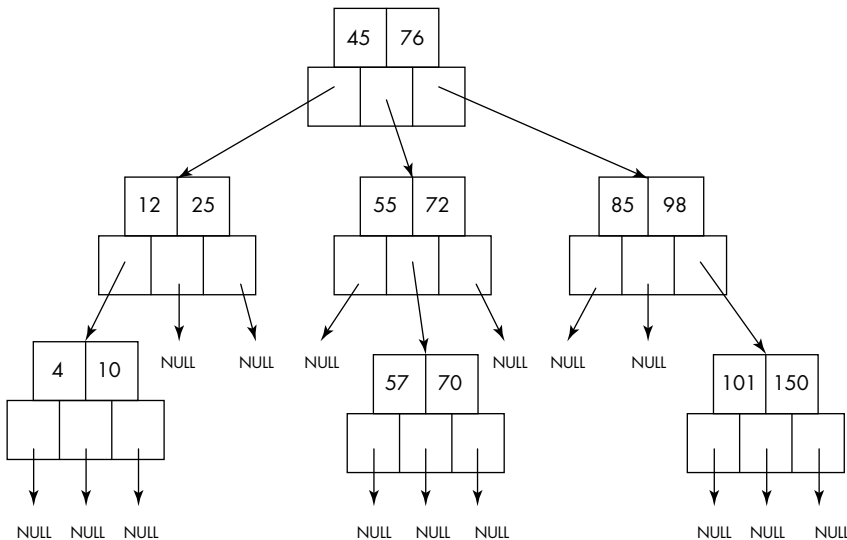


Fig. 10.29 An m -way search tree

Consider the B-tree given in Figure 10.30. It is of order 5 because all the internal nodes have at least $\lfloor 5/2 \rfloor = 3$ children and two keys. In fact, the maximum number of children and keys are 5 and 4, respectively. As per rule number 3, each leaf node must contain at least $\lfloor 5/2 \rfloor - 1 = 2$ keys.

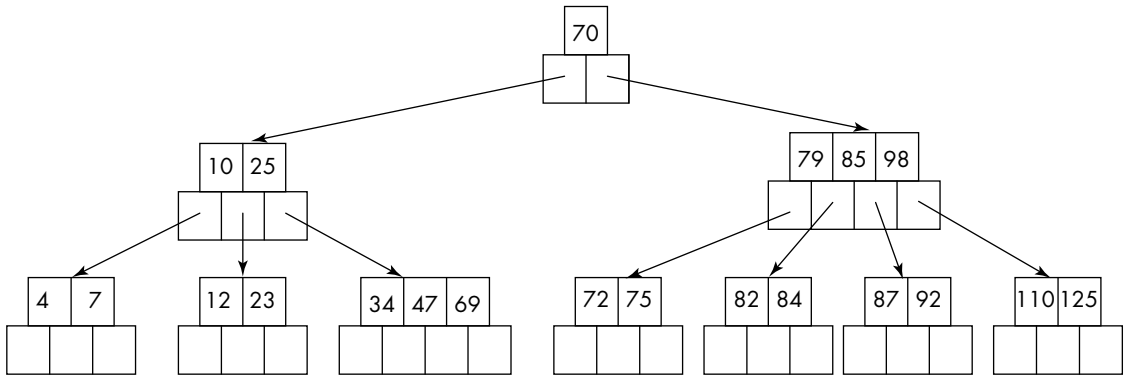


Fig. 10.30 A B-tree of size 5

It may be noted that the node of a B-tree, by its nature, can accommodate more than one key. We may design the size of a node to be as large as a block on the disk (say a sector). As a block is read at a time from the disk, compared to a normal BST, less number of disk accesses would be required to read/write data from the disk.

The following operations can be carried out on a B-tree:

- (1) Searching a B-tree
- (2) Inserting a new key in B-tree
- (3) Deleting a key from a B-tree

10.4.1 Searching a Key in a B-Tree

A key K can be searched in a B-tree in a fashion similar to a BST. The search within a node, including root, follows the following algorithm:

```

Algorithm searchBTree()
{
    Step
    1. Compare  $K$  with  $K_i$  for  $1 \leq i \leq n$ .
    2. If  $K_i == K$ , then report search successful and return.
    3. Find the location  $i$  such that  $K_i \leq K < K_{i+1}$ .
    4. Move to the child node pointed by  $P_i$ .
    5. If  $P_i$  is equal to NULL, then report failure and return else repeat steps 1 to 4.
}

```

The search for $K = 82$ will follow the path as shown in Figure 10.31.

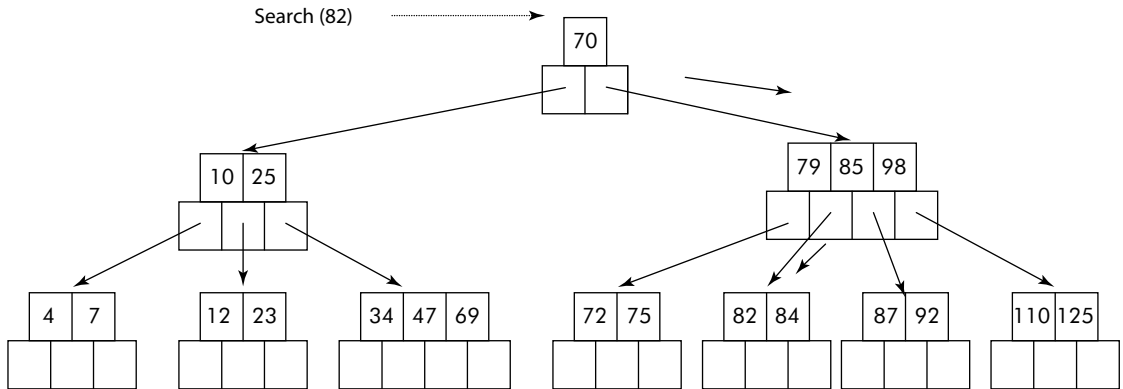


Fig. 10.31 Search within a B-tree

10.4.2 Inserting a Key in a B-Tree

The insertion in a B-tree involves three primary steps: search – insert – (optional) split a node. The search operation finds the proper node wherein the key is to be inserted. The key is inserted in the node if it is already not full, else the node is split on the median (middle) key and the median number is inserted in the parent node. An algorithm that inserts a key in a B-tree is as follows:

Algorithm insertKey(Key)

```
{
  Step
  1. Search the key in the B-tree.
  2. If it is found then prompt "Duplicate Key".
  3. If not found, then mark the leaf. /* Unsuccessful search always
    ends at a leaf */
  4. If the leaf has empty location for the key, then insert it at the
    proper place.
  5. If the leaf is 'full', then split the leaf into two halves at the
    median key.
  6. Move the median key from the leaf to its parent node and insert
    it there.
  7. Insert the key into the appropriate half.
  8. Adjust the pointers of the parent so that they point to the
    correct half.
  9. STOP
}
```

Let us insert the following list of numbers into an empty B-tree:

3, 14, 7, 1, 8, 5, 11, 17, 13, 6, 23, 12, 20, 26

The trace of the above algorithm on the list of numbers is given in Figures 10.32 and 10.33.

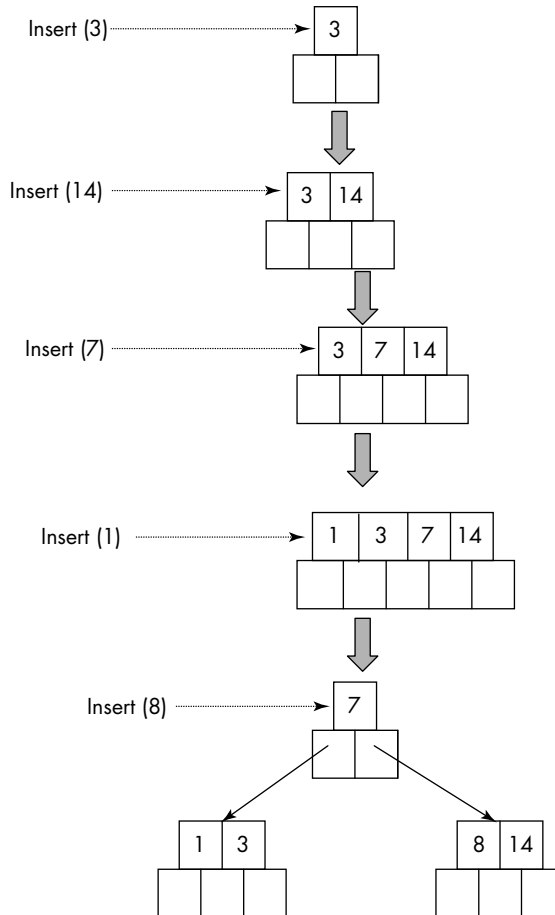


Fig. 10.32 Insertion of keys in a B-tree

10.4.3 Deleting a Key from a B-Tree

Deletion of a key from a node of B-tree has to be done with lot of care because it might violate the basic properties of the B-tree itself. For example, one must consider the following cases:

- (1) After the deletion of a key in a leaf node, if the degree of the node decreases below the minimum degree of the tree then the siblings must be combined to increase the degree to the acceptable level.

Consider the deletion of $K = 8$ from the B-tree given in Figure 10.34.

Since the key (8) is in the leaf of the B-tree and its deletion does not violate the degree of that leaf, the key is deleted and the other elements in the node are appropriately shifted to maintain the order of the keys.

- (2) If the node is internal and has children, the children must be rearranged so that the basic properties of the B-tree are conformed to. In fact, we find its successor from the leaf node. If key is K and its successor is S , then S is deleted from the leaf node. Finally, S replaces the K .

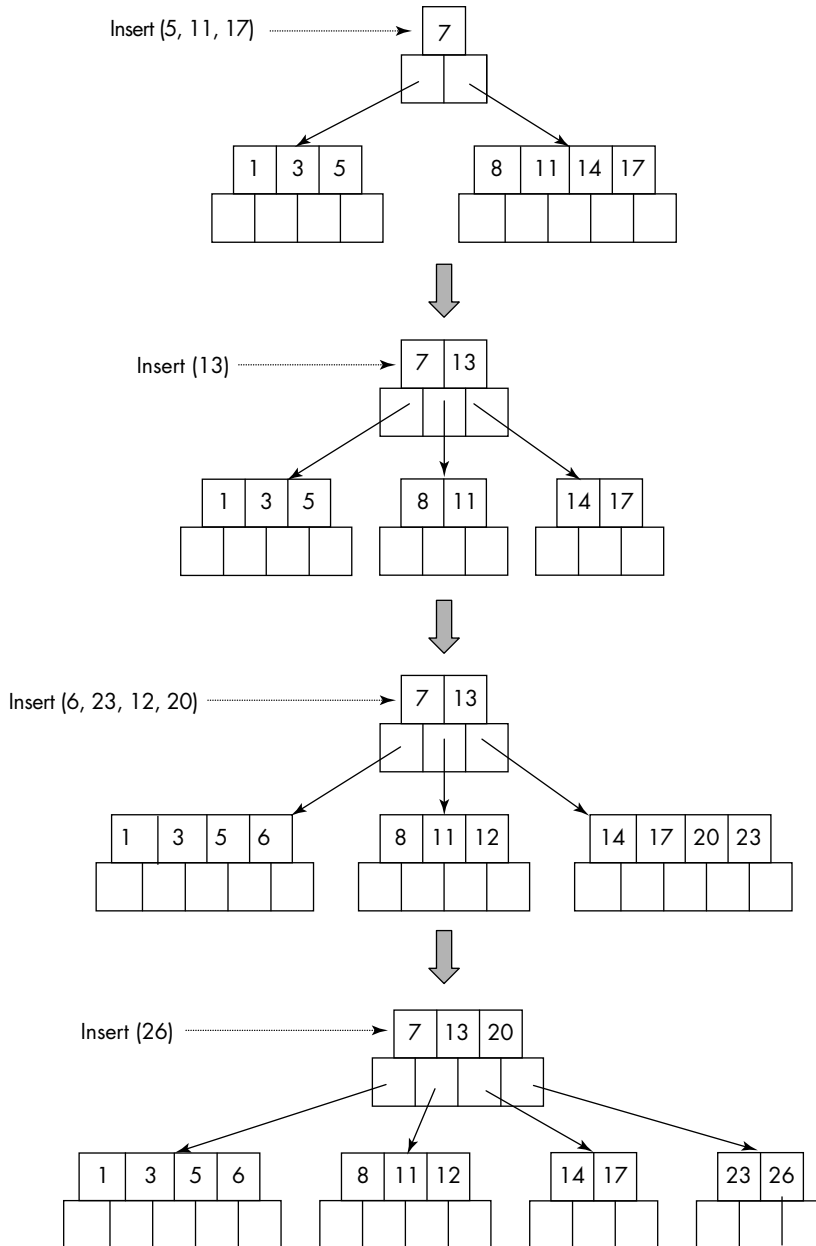


Fig. 10.33 Insertion of keys in a B-tree

Consider the deletion of 20 from the B-tree of Figure 10.35.

It may be noted that 23, the successor of 20, has been deleted from the child of node containing 20. The successor (23) has replaced the key (20) as the final step.

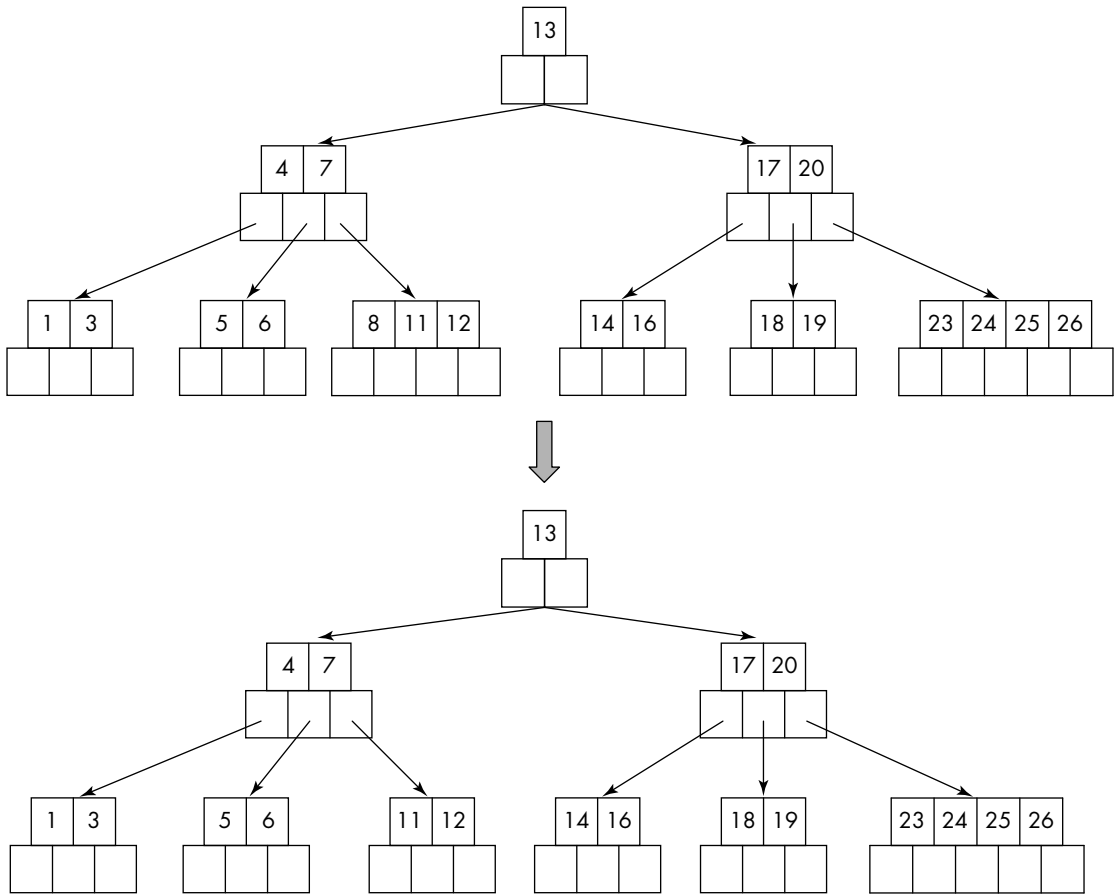


Fig. 10.34 Deletion of a key from a leaf of the B-tree

Consider the deletion of 19 from the B-tree of Figure 10.36.

It may be noted that although the node containing 19 is a leaf node, the deletion violates the property that there should be at least two keys in the leaf node of a B-tree of order 5. Therefore, if an immediate sibling (left or right) has an extra key, then a key is borrowed from the parent (23) that is stored in place of the deleted key (19). The extra key (24) from the sibling is shifted to the parent as shown in Figure 10.36.

Let us finally consider the case of deletion of key (5). The deletion of 5 leaves no extra key in the leaf and its siblings (left or right) have also no extra keys to donate. Therefore, the parent key (4) is moved down and the siblings are combined as shown in Figure 10.37(a).

The moving down of 4 has caused a new problem that the parent node now contains only one key 7 (see Figure 10.37), which is another violation. As its sibling is also not having an extra key, the siblings need to be combined and bring 13 from the parent node. The final tree after adjustment is shown in Figure 10.37(b). However, the height of the tree has reduced by 1.

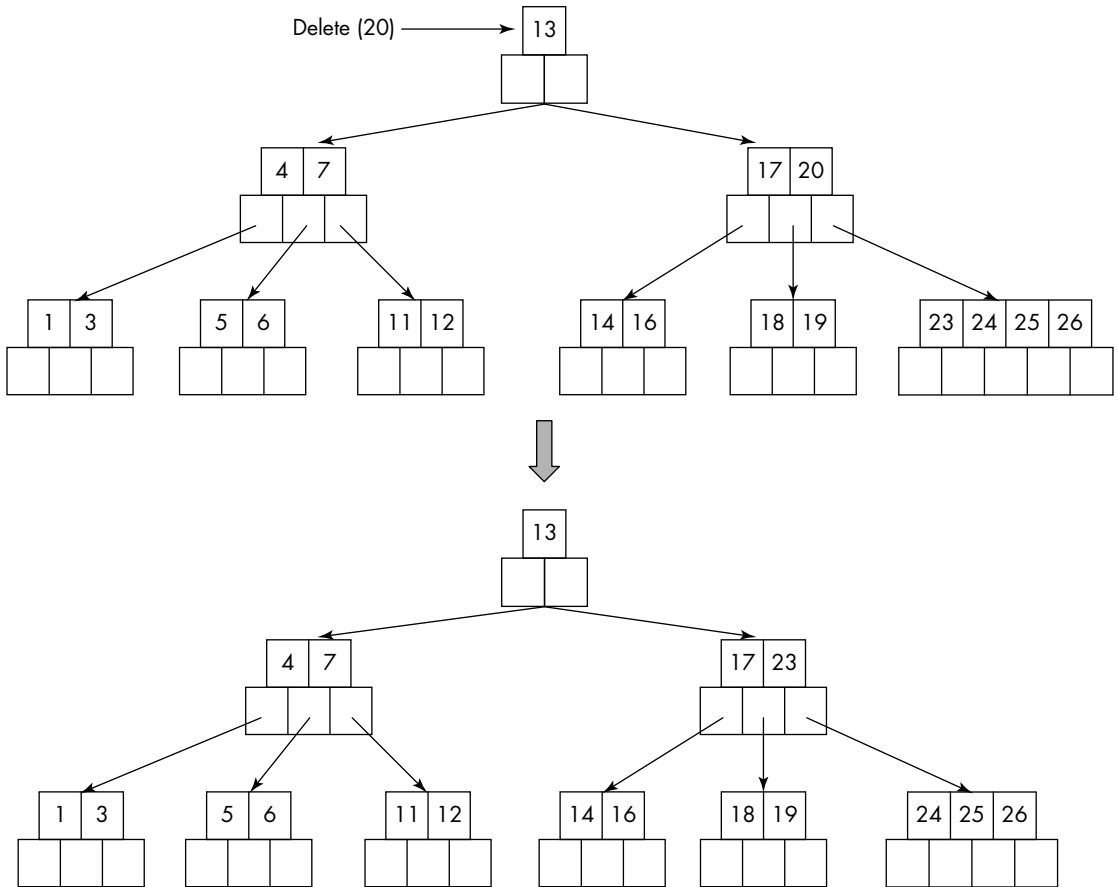


Fig. 10.35 Deletion of a key from an internal node of the B-tree

Note: From the above discussion, it is clear that the insertion, and especially deletion, is a complicated operation in B-trees. Therefore, the size of leaf nodes is kept as large as 256 or more so that insertion and deletion do not demand rearrangement of nodes.

10.4.4 Advantages of B-Trees

- (1) As the size of a node of a B-tree is kept as large as 256 or more, the height of the tree becomes very small. Such a node supports children in the range of (128 to 256). Therefore, insertion and deletion operations will generally remain within the limit without causing over- or underflow of the node.
- (2) As the height of a B-tree is very small, the key being searched is normally found at a shallow depth.
- (3) If the root and next level of nodes are kept in the main memory (RAM), more than a million records of file can be indexed. Consequently, fewer disk accesses would be required to read the required record from the disk.

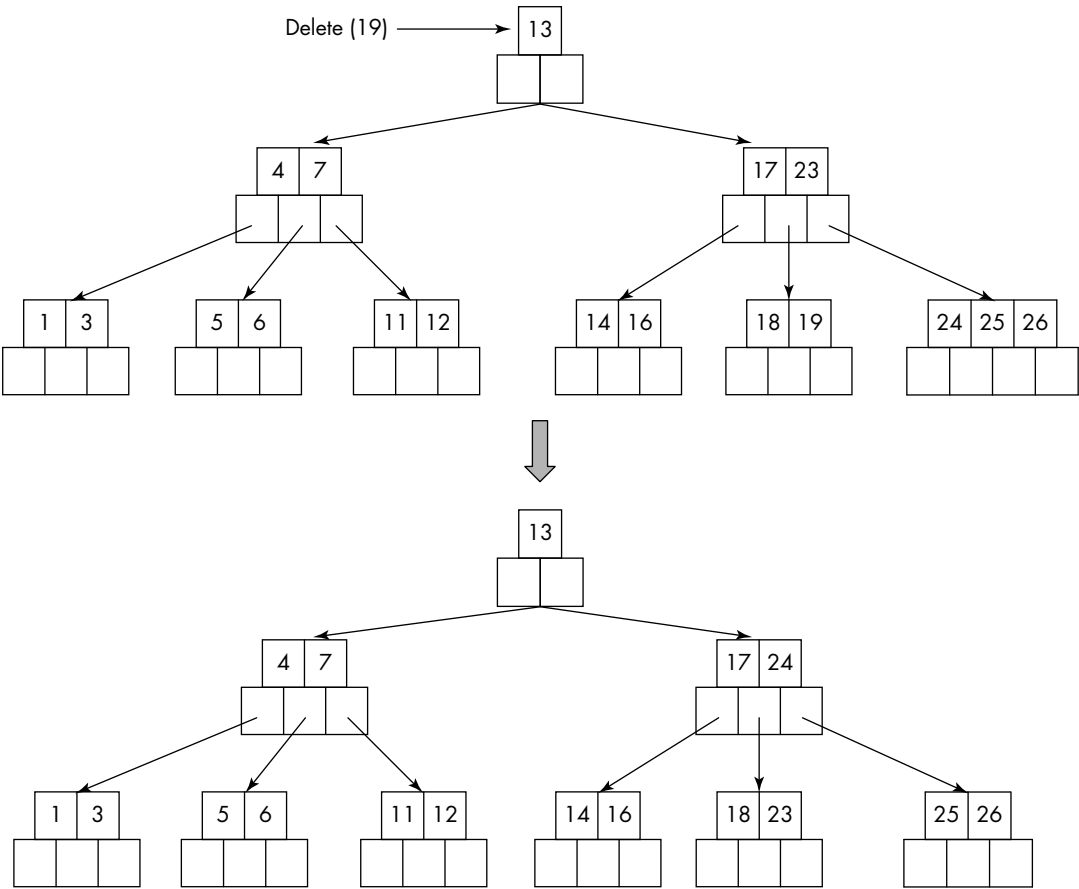


Fig. 10.36 Deletion of a key from a leaf causing violation

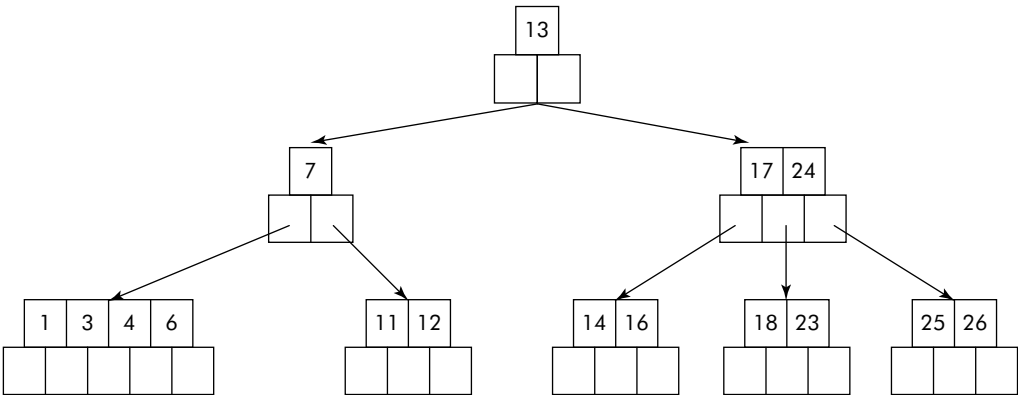


Fig. 10.37 (a) B-tree after deletion of key, $k = 5$

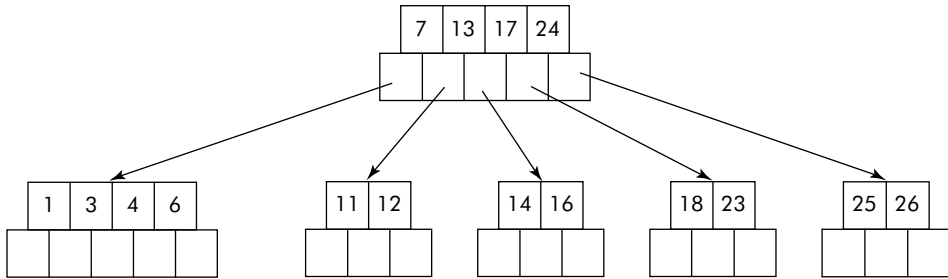


Fig. 10.37 (b) B-tree after double violation

10.5 SEARCHING BY HASHING

In system programming, *search* is a very widely used operation. An assembler or a compiler very frequently searches the presence of a token in a table such as *symbol table*, *opcode table*, *literal table* etc. Earlier in the book, following techniques have been discussed for search operations:

- Linear search
- Binary search
- Binary search trees

Averagely, the linear search requires $O(n)$ comparisons to search an element in a list. On the other hand, the binary search takes $O(\log n)$ comparisons to search an element in the list but it requires the list should already be sorted.

Some applications such as direct files, require the search to be of order $O(1)$ in the best case i.e. the time of search should be independent of the location of the element or key in the storage area. In fact, in this situation a hash function is used to obtain a unique address for a given key or a token. The term 'hashing' means *key transformation*. The address for a key is derived by applying an arithmetic function called 'hash' function on the key itself. The address so generated is used to store the record of the key as shown in Fig. 10.38.

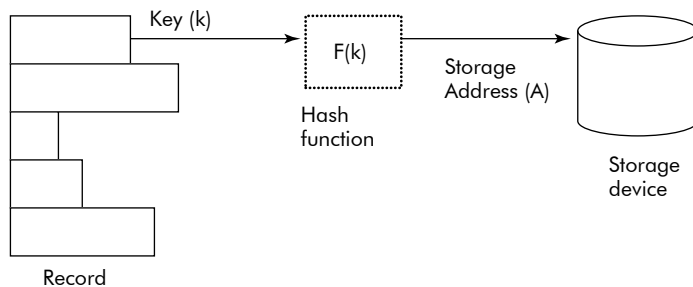


Fig. 10.38 Direct address generation

In the following section, various hashing functions are discussed.

10.5.1 Types of Hashing Functions

Many hashing functions are available in the literature. Most of the efficient hash functions are complex. For instance, MD5 and SHA-1 are complex but very efficient hash functions used for cryptographic applications. However, for searching records or tokens in files or tables, even simple hash functions give good results. Some of the common Hashing functions are described below.

1. Midsquare method: The value of the key K is squared and afterwards a suitable number of digits from the middle of K^2 is chosen to be the address of the record on the disk. Let us assume that 4th and 5th digit from the right of K^2 will be selected as the hash address as shown below :

K	:	5314	6218	9351
K^2	:	28238596	38663524	87441201
Hash Address	:	38	63	41

It may be noted that the records with $K = 5314$, 6218 , and 9351 , would be stored at addresses 38, 63, and 41 respectively.

2. Division method: In this method the key K is divided by a prime number where a prime number is a number that is evenly divisible (without a remainder) only by one or by the number itself. After the division, the quotient is discarded and the remainder is taken as the address. Let us consider the examples given below:

- | | |
|--|---|
| <p>(i) Let key $K = 189235$
 → Hash address</p> <p>(ii) Let key $K = 5314$
 → Hash address</p> | <p>Prime number (p) = 41 (say)
 = $K \bmod p$
 = $189235 \bmod 41$
 = 20 (remainder)</p> <p>Prime number (p) = 41
 = $K \bmod p$
 = $5314 \bmod 41$
 = 25 (remainder)</p> |
|--|---|

3. Folding method: In this method, the key is split into pieces and a suitable arithmetic operation is done on the pieces. The operations can be add, subtract, divide etc. Let us consider the following examples:

- (i) Let key $K = 189235$
 Let us split it into two parts 189 and 235
 By adding the two parts we get,
- | |
|------------------|
| 189 |
| <u>235</u> |
| Hash address 424 |
- (ii) Let key $K = 123529164$
 Splitting the key into pieces and adding them, we get
 Let us split it into three parts 123, 529 and 164
 By adding the two parts we get,
- | |
|------------------|
| 123 |
| 529 |
| <u>164</u> |
| Hash address 816 |

It may be observed from the examples given above, that there are chances that the records with different key values may hash to the same address. For example, in folding technique, the keys 123529164 and 529164123 will generate the same address, i.e., 816. Such mapping of keys to the same address is known as a collision and the keys are called as synonyms.

Now, to manage the collisions, the overflowed keys must be stored in some other storage space called overflow area. The procedure is described below:

- When a record is to be stored, a suitable hashing function is applied on the key of the record and an address is generated. The storage area is accessed, and, if it is unused, the record is stored there. If there is already a record stored, the new record is written in the overflow area.
- When the record is to be retrieved, the same process is repeated. The record is checked at the generated address. If it is not the desired one, the system looks for the record in the overflow area and retrieves the record from there.

Thus, synonyms cause loss of time in searching records, as they are not at the expected address. It is therefore essential to devise hash algorithms which generate minimum number of collisions. The important features required in a hash algorithm are given in the following section.

10.5.2 Requirements for Hashing Algorithms

The important features required in a Hash algorithm or functions are :

- **Repeatable:** A capability to generate a unique address where a record can be stored and retrieved afterwards.
- **Even distribution:** The records of a file should be evenly distributed throughout the allocated storage space.
- **Minimum collisions:** It should generate unique addresses for different keys so that number of collisions can be minimized.

Nevertheless, the collisions can be managed by using suitable methods as discussed in next section:

10.5.3 Overflow Management (Collision Handling)

The following overflow management methods can be used to manage the collisions.

1. Chaining: The overflowed records are stored in a chain of collisions maintained as linked lists as shown in Figure 10.39

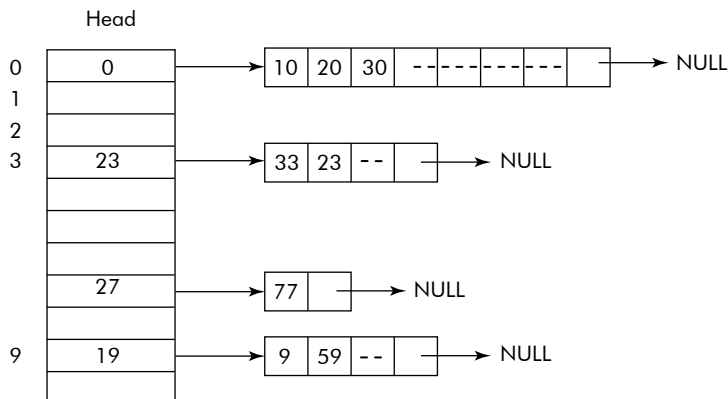


Fig. 10.39 Chaining

Assuming that $(K \bmod 10)$ was taken as the hash function. Therefore, the keys with values 0, 10, 20, 30 etc. will map on the same address i.e. 0 and their associated records are stored in a linked list of nodes as shown in figure. Similarly 19, 9, and 59 will also map on the same address i.e. 9 and their associated records stored in a linked list of nodes.

It may be noted that when a key K is searched in the above arrangement and if it is not found in the head node then it is searched in the attached linked list. Since the linked list is a pure sequential data structure, the search becomes sequential. A poor hash function will generate many collisions leading to creation of long linked lists. Consequently the search will become very slow. The remedy to above drawback of chaining is rehashing.

2. Rehashing: When the address generated by a hash function $F1$ for a key K_j collides with another key K_i then another hash function $F2$ is applied on the address to obtain a new address $A2$. The collided Key K_j is then stored at the new address $A2$ as shown in Fig. 10. 40

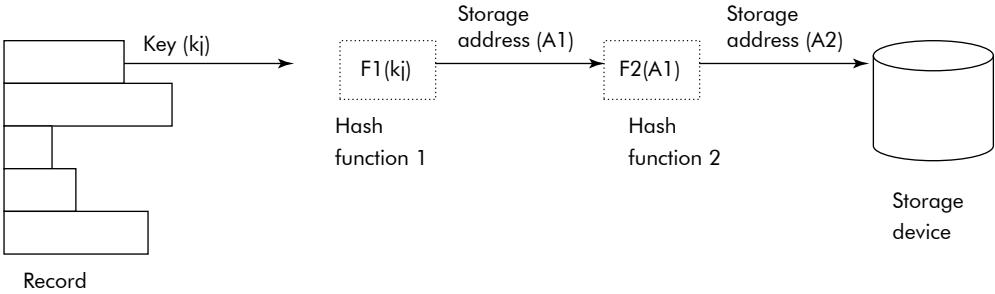


Fig. 10.40 Rehashing

It may be noted that if the space referred to by $A2$ is also occupied, then the process of rehashing is again repeated. In fact rehashing is repeatedly applied on the intermediate address until a free location is found where the record could be stored.

EXERCISES

1. When does a binary search tree (BST) become a skewed tree?
2. What is an AVL Tree? What are its advantages over a BST?
3. Write an explanatory note on AVL rotations.
4. For the AVL Tree shown in Figure 10.41, label each node in the AVL Tree with the correct balance factor. Use 1 for left higher, 0 for equal height, and -1 for right higher subtrees.

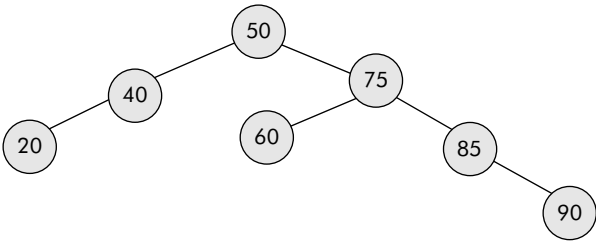


Fig. 10.41 An AVL tree

5. For the AVL Tree shown in Fig. 10.41, draw the final AVL Tree that would result from performing the actions indicated from (a) to (e) listed below:
 - a. Insert the key 10.
 - b. Insert the key 95.
 - c. Insert the key 80, and then insert the key 77.
 - d. Insert the key 80, and then insert the key 83.
 - e. Insert the key 45.

Note: Always start with the given tree, the questions are not accumulative. Also, show both the data value and balance factor for each node.
6. Define set data structure. How are they different from arrays? Give examples to show the applications of the set data structures.
7. Write an algorithm that computes the height of a tree.
8. Write an algorithm that searches a key K in an AVL Tree.
9. What are the various cases of insertion of a key K in an AVL Tree?
10. Define the terms: null set, subset, disjoint set, cardinality of a set, and partition.
11. What are the various methods of representing sets?
12. Write an algorithm that takes two sets S1 and S2 and produces a third set S3 such that $S3 = S1 \cup S2$.
13. Write an algorithm that takes two sets S1 and S2 and produces a third set S3 such that $S3 = S1 \cap S2$.
14. Write an algorithm that takes two sets S1 and S2 and produces a third set S3 such that $S3 = S1 - S2$.
15. How are web pages managed by a search engine?
16. Give the steps used for colouring a black and white photograph.
17. Write a short note on spell checker as an application of sets.
18. You are given a set of persons P and their friendship relation R, that is, $(a, b) \in R$ if a is a friend of b. You must find a way to introduce person x to person y through a chain of friends. Model this problem with a graph and describe a strategy to solve the problem.
19. The *subset-sum* problem is defined as follows:

Input: a set of numbers $A = \{a1, a2, \dots, aN\}$ and a number x;

Output: 1 if there is a subset of numbers in A that add up to x.

Write down the algorithm to solve the *subset-sum* problem.
20. What is a skip list? What is the necessity of adding extra links?
21. Write an algorithm that searches a key K in a skip list.
22. What is an m-way tree.
23. Define a B-tree. What are its properties?
24. Write an algorithm for searching a key K, in a B-tree.
25. Write an algorithm for inserting a key K, in a B-tree.
26. Write an explanatory note on deletion operation in a B-tree.
27. What are the advantages of a B-tree.

ASCII Codes (Character Sets)

A APPENDIX

A.1 ASCII (American Standard Code for Information Interchange) Character Set

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0																
16																
32	␣	!	"	#	\$	%	&	;	()	*	+	,	-	.	/
48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
96	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112	p	q	r	s	t	u	v	w	x	y	z	{		}	~	

Codes 00–31 and 127 are nonprintable control characters.

A.2 EBCDIC (Extended Primary Coded Decimal Interchange Code) Character Set

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0																
16																
32																
48																
64	␣										␣	.	<	(+	
80	&										!	\$	*)	;	¬
96	—	/										,	%	_	>	?
112											:	#	@	'	=	"
128		a	b	c	d	e	f	g	h	i						
144		j	k	l	m	n	o	p	q	r						
160			s	t	u	v	w	x	y	z						
176																
192		A	B	C	D	E	F	G	H	I						
208		J	K	L	M	N	O	P	Q	R						
224			S	T	U	V	W	X	Y	Z						
240	0	1	2	3	4	5	6	7	8	9						

Table of Format Specifiers

Format Specified	Meaning
c	data item is a single character
d	data item is a decimal integer
e	data item is a floating point value
f	data item is a floating point value
g	data item is a floating point value
h*	data item is a short integer
i	data item is a decimal, hexadecimal or octal integer
o	data item is a octal integer
s	data item is a string followed by a white space
u	data item is a is an unsigned decimal integer
x	data item is a hexadecimal integer
[...]*	data item is a string which may include while space characters

*Only applicable for data input i.e for scanf() function.

Escape Sequences

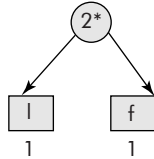
Character	Escape Sequence	ASCII Value
bell (alert)	\a	$\phi\phi7$
backspace	\b	$\phi\phi8$
horizontal tab	\t	$\phi\phi9$
newline (line feed)	\n	$\phi1\phi$
vertical tab	\v	$\phi11$
form feed	\f	$\phi12$
carriage return	\r	$\phi13$
quotation mark (")	\"	$\phi34$
apostrophe (')	\'	$\phi39$
question mark (?)	\?	$\phi63$
backslash (\)	\\	$\phi92$
null	\0	$\phi\phi\phi$
octal number	\000	(0 represents an octal digit)

Trace of Huffman Algorithm

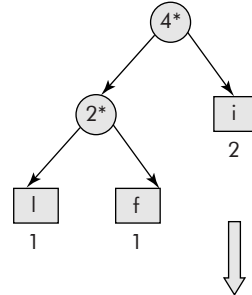
D

APPENDIX

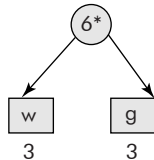
l f i . , w g y u a n r t h o e B
1 1 2 2 2 3 3 4 4 4 4 7 7 7 8 9 11 18



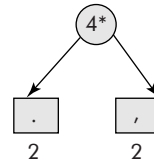
i . , w g y u a n r t h o e B
2* 2 2 2 3 3 4 4 4 4 7 7 7 8 9 11 18



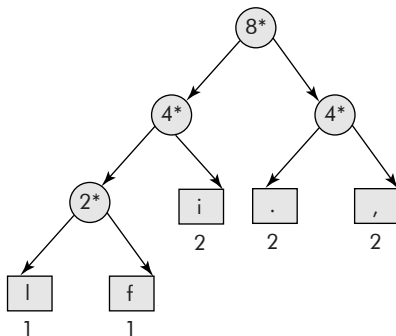
w g y u a n r t h o e B
3 3 4* 4* 4 4 4 7 7 7 8 9 11 18



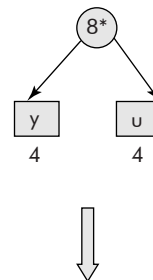
. , w g y u a n r t h o e B
2 2 3 3 4* 4 4 4 7 7 7 8 9 11 18

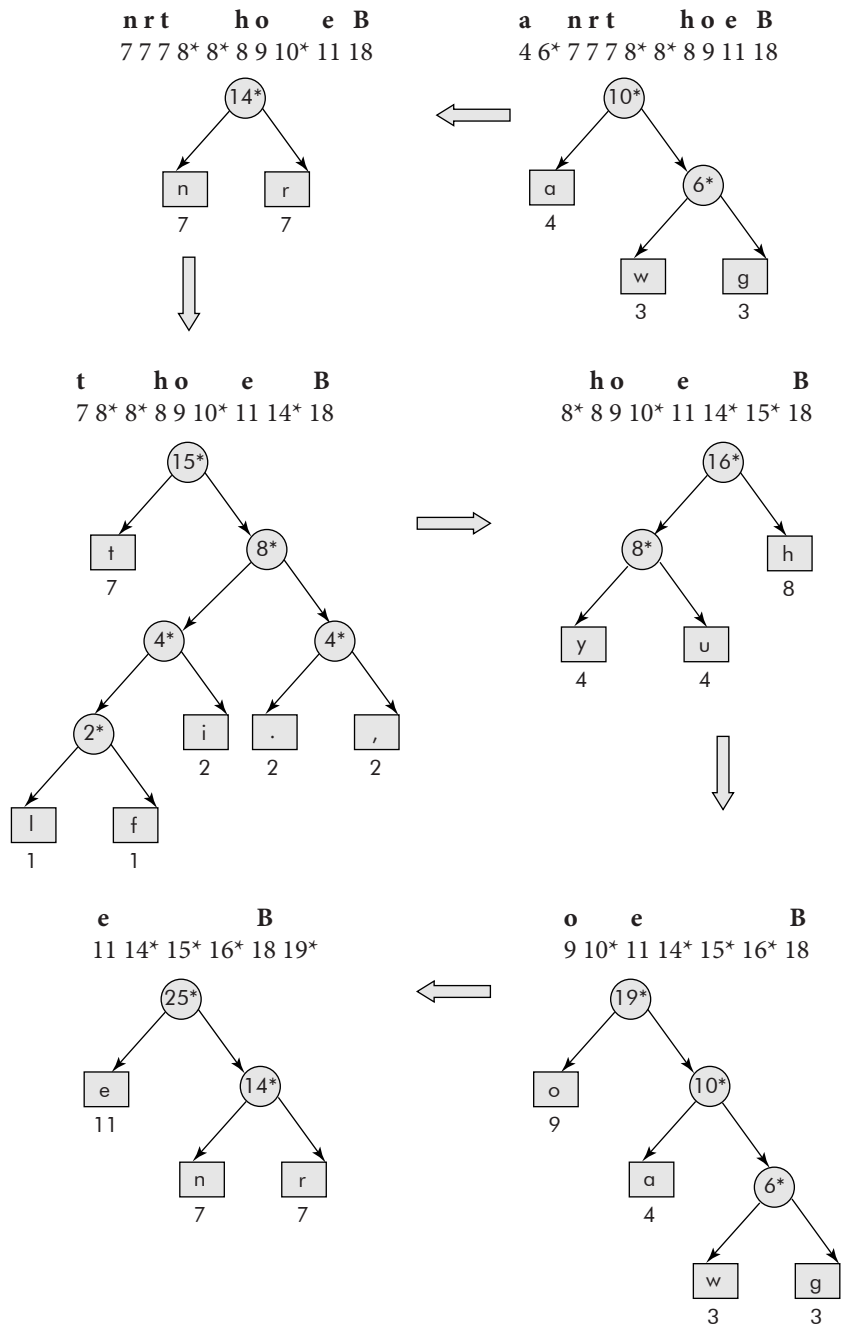


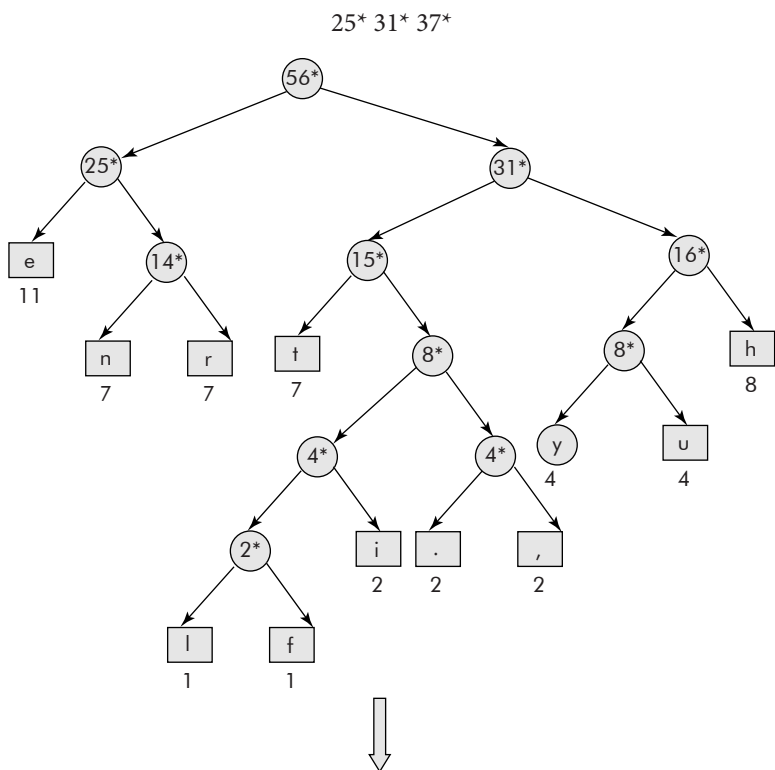
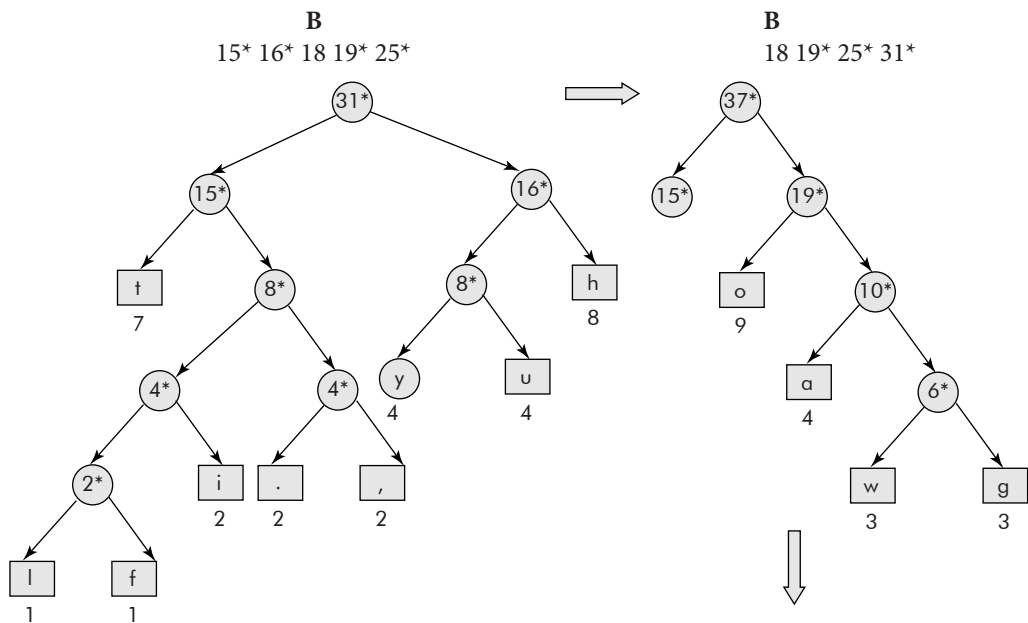
y u a n r t h o e B
4* 4* 4 4 4 6* 7 7 7 8 9 11 18



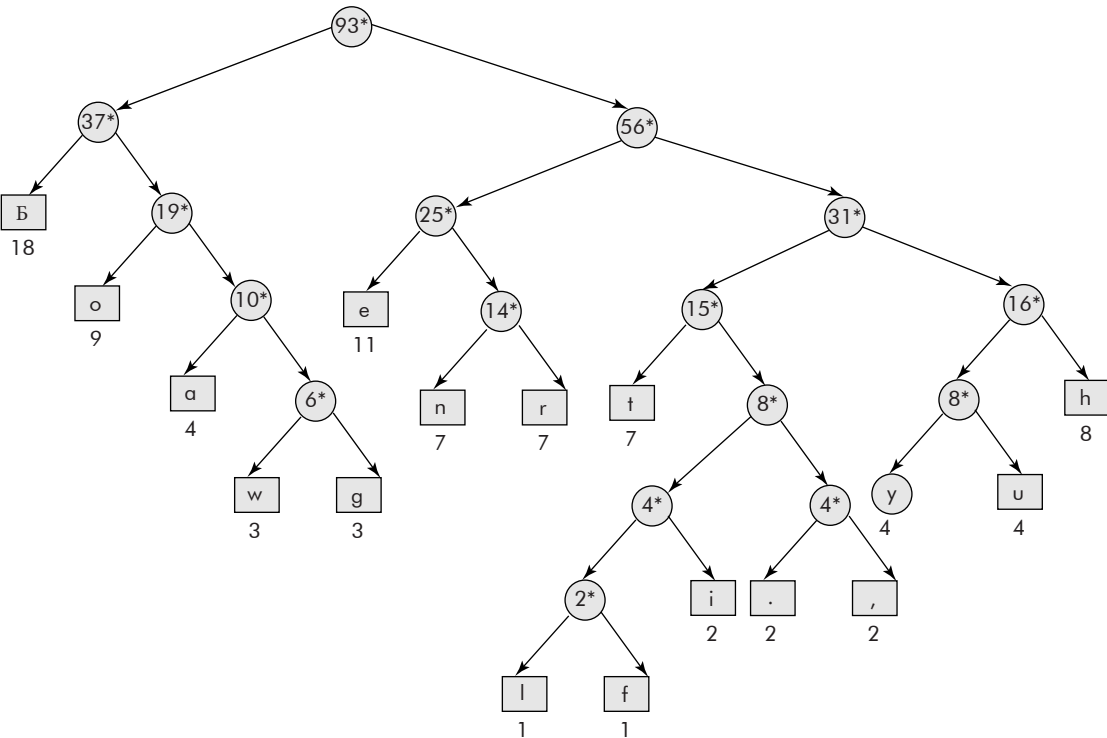
y u a n r t h o e B
4 4 4 6* 7 7 7 8* 8 9 11 18







37* 56*



Index

& operator, 197–198
“*” operator, 198

A

accumulators, 83
activity ratio, 448–449
acyclic graph, 368
adjacency matrix, 368
adjacent node, 368
advanced data structures, 459–492
algorithm, 78–90
 analysis of, 85–86
 Big-Oh notation, 86–90
 design of, 78
 stepwise refinement, 80–81
 using control structures, 81–85
 ways to develop, 78–80
archival file, 414
arithmetic operators, 13–14
 unary arithmetic operators, 13–14
array of pointers, 208
arrays, 34, 93–149
 applications of, 138–149
 multi-dimensional arrays, 130–134
 one-dimensional array, 94–130
 pointers and, 203–207
 representation in physical memory,
 134–138
 two-dimensional arrays, 136–137
assignment operator, 18–19
AVL trees, 459–467
 inserting a node in, 462–467
 searching of, 461

B

back up file, 414
backlash character constants. *See*
 escape sequence
BFS. *See* breadth first search (BFS)
Big-Oh notation, 86–90
binary recursion, 61–65
binary search tree (B-trees), 303–319,
 480–489
 advantages of, 487–489
 creation of, 304–306
 deleting a key in, 484–487
 deletion of a node from, 312–319
 inserting a key in, 483–484
 insertion into, 308–312

 searching a key in, 482–483
 searching in, 307–308
binary trees, 285–298
 binary search tree 303–319
 complete binary tree, 285
 expression tree, 298–303
 full binary tree, 285
 heap trees, 319–340
 linked representation of, 289–291
 threaded binary trees, 340–352
 traversal of, 291–298
 weighted binary trees, 352–360
bitwise shift operator, 19–20
breadth first search (BFS), 390–396
break statement, 27
B-trees. *See* binary search tree
 (B-trees)
 chaining, 491–492
 rehashing, 492
bubble sort, 113–116
 analysis of, 113–116
buffered I/O, 31–32

C

C, 1–70
 characters used in, 2
 comments, 10–11
 console I/O functions in, 417
 data types, 2–4
 defining a structure, 35–36
 escape sequence, 11–13
 flow of control, 20–30
 history of, 1
 low-level I/O functions in, 418
 operators and expressions, 13–20
 stream I/O functions in, 417
 structure of, 8
 tokens, 4–7
C tokens, 4–7
 constants, 7
 identifiers, 4–5
 keywords, 5
 variables, 5–6
call by reference, 49–51, 53–55
call by value, 47–49, 52–53
character constant, 7
character data type (`char`), 3
child, 284
circular linked list, 254–258

circular queue, 176–181
collision handling. *See* overflow
 management
colouring of maps, 408–409
column major order, 131
comma operator, 18
comments, 10–11
comparison-based algorithm, 109
complete binary tree, 285
complete graph, 368
compound statement, 21
condensed matrix, 142
conditional operators, 16–17
conditional statements, 21–25
 if statement, 21–22
 if-else statement, 22
 nested if statements, 22–23
 switch statement, 23–25
connected graph, 367
`const` correct, 7
`const` qualifier, 7
constants, 7
 character constant, 7
 floating point constant, 7
 integer constant, 7
continue statement, 27
control structures, 81–85
counters, 83
cycle, 367

D

dangling pointers, 202–203
data, 412
data element, 7777
data object, 77
data structures, 73–78
 concept of, 73–78
 terminology related with, 77–78
 types of, 76–77
data tracks, 448
data types, 2–4
 character data type (`char`), 3
 floating data type (`float`), 3–4
 integer data type (`int`), 2
 user-defined data types, 39–42
degree, 284–285
deletion operation, graph, 379–384
deletion, 107–109
Dennis, Ritchi, 1

depth first search (DFS), 384–390
 depth, 284
 deque, 185–195
 deterministic loops, 25
 DFS. *See* depth first search (DFS)
 difference of sets, 474–475
 Dijkstra's algorithm, 404–407
 diminishing step sort, 128
 direct file organization, 442–445
 direct file, 444–445
 advantages of, 444–445
 directed graph, 366
 division method, 490
 doubly linked list, 258–263
 do-while loop, 25–26
 dummy nodes, 264–266
 concept of, 264–266
 dynamic allocation, 210–220
 dynamic data structure, 152
 dynamic dictionary coding, 360–362

E

edge, 284
 enumerated data types, 40–42
 equality of sets, 475–476
 escape sequence, 11–13
`exit ()` function, 27–28
 expression tree, 298–303
 expressions, 17
 order of evaluation of, 17
 extended binary tree, 353
 external node, 353
 external path length, 353

F

file, 413
 concepts of, 413–415
 in C, 416
 opening of, 418–419
 stream and, 416–418
 types of, 414
 working with files using stream
 I/O, 418–430
 file opening modes, 419
 file organization, 415–416
 finite loop, 83
 floating data type (`float`), 3–4
 floating point constant, 7
 flow of control, 20–30
 compound statement, 21
 conditional statements, 21–25
 `exit ()` function, 27–28
 `goto` statement, 28–30
 nested loops, 28
 Floyd's algorithm, 404
 folding method, 490–491

for loop, 26
 formatted file I/O operations,
 425–426
 full binary tree, 285
 functions, 43–56
 calling, 45–47
 parameter passing in, 47–51
 passing structures to, 52–56
 prototypes, 44–45
 returning values from, 52

G

`getc ()` function, 33
`getchar ()` function, 32–33
`getche ()` function, 33
`gets ()` function, 34
`goto` statement, 28–30
 graph
 applications of, 408–410
 array-based representation of,
 368–371
 linked representation of, 371–373
 operations of, 373–408
 set representation of, 373

H

hash table representation, 470
 hashing, 489–492
 searching by, 489–492
 hashing algorithms, 491
 requirements of, 491
 hashing functions, 490–491
 types of, 490–491
 heap trees, 319–340
 deletion of a node from, 324–328
 heap sort, 329–336
 insertion of a node into, 320–324
 merging of two, 336–340
 representation of, 320
 height, 284
 height branched binary tree, 460
 Huffman algorithm, 352–360
 Huffman codes, 356–360

I

I/O operations
 formatted file I/O operations,
 425–426
 reading or writing blocks of data in
 files, 426–430
 unformatted file I/O operations,
 419–425
 I/O stream, 418–430
 identifiers, 4–5
 if statement, 21–22
 if-else statement, 22

index area, 448
 indexed files, 448
 advantages of, 448
 disadvantages of, 448
 indexed sequential organization,
 445–448]
 addition/deletion of record,
 446–447
 multilevel indexed files, 448
 searching a record, 445–446
 storage devices for, 447–448
 infinite loop, 83
 infix method, 158–159
 information, 412–413
 inorder traversal, 292–295
 input/output functions (I/O), 30–34
 buffered I/O, 31–32
 single character functions, 32–33
 string-based functions, 33–34
 insertion, 105–107
 insertion operation, graph, 374–378
 insertion sort, 117–119
 analysis of, 117–119
 integer, 2
 integer constant, 7
 integer data type (`int`), 2
 integration, 450
 internal node, 284, 353
 internal path length, 353–354
 intersection of sets, 473–474
 iterative statements, 25–27
 break statement, 27
 continue statement, 27
 do-while loop, 25–26
 for loop, 26
 while loop, 25

K

key field, 415
 key transformation, 489
 keywords, 5
 Kruskal's algorithm, 397–399

L

linear data structures, 76–77
 linear linked list, 228
 linear recursion, 61
 linked list, 227–280
 circular linked list, 254–258
 deleting a node from, 250–253
 doubly linked list, 258–263
 insertion in, 243–250
 operations on, 231
 searching, 241–243
 travelling, 236–241
 variations of, 253–263

linked queues, 270–273
 linked stacks, 267–269
 linked storage, 274
 list representation, 469
 long integer, 2
 loop, 368

M

master file, 414
 matrix arrays, 130
 memory bleeding, 212
 merge sort, 119–124
 analysis of, 119–124
 midsquare method, 490
 minimum cost spanning trees,
 396–401
 model of www, 408
 multi-dimensional arrays,
 130–134
 multigraphs, 368
 multilevel indexed files, 448

N

nested if statements, 22–23
 nested loops, 28
 nested structures, 38–39
 non-deterministic loops, 25
 non-linear data structures, 76–77

O

one-dimensional array, 94–130
 deletion, 107–109
 insertion, 105–107
 physical address computation of
 elements of, 135
 searching, 98–104
 selection, 96–98
 sorting, 109–113
 traversal, 95–96
 operators, 13–20
 arithmetic operators, 13–14
 assignment operator, 18–19
 bitwise shift operator, 19–20
 conditional operators, 16–17
 relational and logical operators,
 14–16
 special operators, 18
 overflow area, 448
 overflow management, 491–492
 chaining, 491–492
 rehashing, 492

P

parallel edges, 368
 parent, 284
 path, 284, 366–367

pointer variables, 198–203
 dangling pointers, 202–203
 pointers, 197–223
 & operator, 197–198
 ‘*’ operator, 198
 arrays and, 203–207
 pointer variables, 198–203
 structures and, 208–210
 polish notation. *See* prefix expression
 postfix expression, 158
 evaluation of, 164–170
 postorder transversal, 297–298
 post-test loop, 26
 prefix expression, 157
 preorder transversal, 296–297
 pre-test loop, 25
 Prim’s algorithm, 399–401
 primitive data types, 83
 printf () functions, 9–10
 display data using, 9–10
 priority queue, 181–185
 putc () function, 33
 putchar () function, 33
 putchar () functions, 32–33
 puts () function, 34

Q

queues, 170–195
 circular queue, 176–181
 deque, 185–195
 linked queues, 270–273
 operations, 171–176
 priority queue, 181–185
 quick sort, 124–127
 analysis of, 124–127

R

radix sort, 130
 recursion, 56–68
 types of, 60–65
 reference file, 414
 relational and logical operators,
 14–16
 relative file, 444
 characteristics of, 444
 repetitive execution. *See* iterative
 statements
 report file, 414
 resource allocation graph, 408
 response time, 450
 reverse polish notation. *See* postfix
 expression
 roll field, 415
 root, 284
 row major order, 131
 rvalue, 6

S

scanf () functions, 9–10
 read data from keyboard using, 10
 scene graphs, 409–410
 searching, 98–104
 selection, 96–98
 selection sort, 109–113
 analysis of, 113
 selective execution. *See* conditional
 statements
 self referential structures, 215–220,
 231–236
 sequential file organization, 430–442
 appending a sequential file,
 431–437
 creating a sequential file, 430
 reading and searching a sequential
 file, 431
 updating a sequential file, 437–442
 sequential storage, 274
 sets, 468–478
 application of, 476–478
 difference of sets, 474–475
 equality of sets, 475–476
 intersection of sets, 473–474
 operation on, 470–476
 representation of, 469
 union of sets, 471–473
 shell sort, 127–130
 analysis of, 127–130
 short integer, 2
 shortest path problem, 401–407
 sibling, 284
 simple graph, 368
 single character functions, 32–33
 singleton, 119
 sinking sort, 116
 sizeof operator, 18
 skip list, 478–480
 sort file, 414
 sorting, 109–113
 bubble sort, 113–116
 insertion sort, 117–119
 merge sort, 119–124
 quick sort, 124–127
 selection sort, 109–113
 shell sort, 127–130
 spanning trees, 396–401
 sparse graph, 396
 sparse matrix addition, 143–147
 sparse matrix representation,
 141–149
 sparse matrix addition, 143–147
 sparse matrix transpose, 147–149
 sparse matrix transpose, 147–149
 special operators, 18

- comma operator, 18
- sizeof operator, 18
- spelling checker, 478
- stack method, 159
- stacks, 151–170
 - applications of, 155–170
 - linked stacks, 266–269
 - operations, 152–155
- statement label, 29
- stored program computer, 72
- stream, 416–418
- stream I/O, 418–430
- string-based functions, 33–34
- structures, 34–39
 - array of, 36
 - assignment of, 37–38
 - initializing, 36
 - nested structures, 38–39
 - pointers and, 208–210
- switch statement, 23–25

T

- threaded binary trees, 340–352
 - deletion from, 348–352
 - insertion into, 344–347
- time complexity, 86
- Tower of Hanoi, 65–67
- transaction file, 414
- transversal of graph, 384–396
- traversal, 95–96
- trees, 282–362
 - terminology related to, 284–285
- two-dimensional arrays, 136–137
 - physical address computation of elements of, 136–137

U

- unary arithmetic operators, 13–14
- unconditional branching. *See* goto statement
- undirected graph, 366
- unformatted file I/O operations, 419–425

- union of sets, 471–473
- unions, 42
- unsigned integer, 2
- user-defined data types, 39–42
 - enumerated data types, 40–42

V

- variables, 5–6
- vertex, 368
- volatility, 450
- Von Neumann architecture, 72

W

- Warshall's algorithm, 401–403
- web linked pages, 476–477
- weighted binary trees, 352–360
 - Huffman algorithm and, 352–360
- weighted graph, 366
- while loop, 25
- world wide web, 408