



SURESH
GYAN VIHAR
UNIVERSITY
Accredited by NAAC with 'A+' Grade

Master of Computer Application
(M.C.A.)

Data Structures through C

Semester-I

Author - Dr. Anil Kumar Pal

SURESH GYAN VIHAR UNIVERSITY
Centre for Distance and Online Education
Mahal, Jagatpura, Jaipur-302025

EDITORIAL BOARD (CDOE, SGVU)

Dr (Prof.) T.K. Jain
Director, CDOE, SGVU

Dr. Dev Brat Gupta
*Associate Professor (SILS) & Academic
Head, CDOE, SGVU*

Ms. Hemlalata Dharendra
Assistant Professor, CDOE, SGVU

Ms. Kapila Bishnoi
Assistant Professor, CDOE, SGVU

Dr. Manish Dwivedi
*Associate Professor & Dy, Director,
CDOE, SGVU*

Mr. Manvendra Narayan Mishra
*Assistant Professor (Deptt. of Mathematics)
SGVU*

Mr. Ashphaq Ahmad
Assistant Professor, CDOE, SGVU

Published by:

S. B. Prakashan Pvt. Ltd.

WZ-6, Lajwanti Garden, New Delhi: 110046

Tel.: (011) 28520627 | Ph.: 9205476295

Email: info@sbprakashan.com | Web.: www.sbprakashan.com

© SGVU

All rights reserved.

No part of this book may be reproduced or copied in any form or by any means (graphic, electronic or mechanical, including photocopying, recording, taping, or information retrieval system) or reproduced on any disc, tape, perforated media or other information storage device, etc., without the written permission of the publishers.

Every effort has been made to avoid errors or omissions in the publication. In spite of this, some errors might have crept in. Any mistake, error or discrepancy noted may be brought to our notice and it shall be taken care of in the next edition. It is notified that neither the publishers nor the author or seller will be responsible for any damage or loss of any kind, in any manner, therefrom.

For binding mistakes, misprints or for missing pages, etc., the publishers' liability is limited to replacement within one month of purchase by similar edition. All expenses in this connection are to be borne by the purchaser.

Designed & Graphic by : S. B. Prakashan Pvt. Ltd.

Printed at :

SYLLABUS
MCA (Semester - I)
Data Structure through C (MCA-103)

Learning Outcome:

- While studying the Data Structures course, the student shall be able to:
- Learn the concept of data structures through ADT including List, Stack, and Queues.
- Design and implement various data structure algorithms.
- Understand various techniques for representation of the data in the real world.
- Able to develop application using data structure algorithms

Unit:1

Definition of Algorithm and Data structure- Types of Data structure (linear and non- linear data structure) - Linear data structure: Array, Stack, and Queue - Linked List-doubly linked list and circular list - Recursive and non recursive algorithm.

Unit: 2

Binary Tree – notations, terminology, Representation, Binary tree Traversal and Application - Graph- Notations, Terminology-Representation, Traversal and Application. Performance analysis of an algorithm.-Tabular method.

Unit: 3

Min/Max heaps – Deaps – Leftist Heaps - Binomial Heaps – Fibonacci Heaps - Skew Heaps – Lazy-Binomial Heaps.

Unit: 4

Binary Search Trees – AVL Trees - Red-Black trees – Multi-way Search Trees - B-Trees– Splay Trees – Tries.

Unit:5

Segment Trees – k-d Trees - Point Quad Trees – MX-Quad Trees - R-Trees – TV Trees

Reference books:

1. Á E. Horowitz, S.Sahni and Dinesh Mehta, Fundamentals of Data structures in C++, University Press, 2007.
2. Á E. Horowitz, S. Sahni and S. Rajasekaran, Computer Algorithms/C++, Second Edition, University Press, 2007.
3. Á G. Brassard and P. Bratley, Algorithmics: Theory and Practice, Printice–Hall, 1988.
4. Á V.S. Subramanian, Principles of Multimedia Database systems, Morgan Kaufman,1998

MCA-03: Data Structures through C

Page No:

Block 1	C Programming Language Fundamentals	4
Unit 1	Language Fundamentals	5
Unit 2	Data Types And I/O Functions	14
Unit 3	Control Statements	39
Block 2	Structured Programming with C	55
Unit 4	Arrays and structures	56
Unit 5	Storage classes and Functions	71
Unit 6	File Handling In C	84
Block 3	Data Structures in C	98
Unit 7	Stacks and Queues	99
Unit 8	Linked list	111
Unit 9	Graphs	129
Block 4	Tree, Searching and Sorting	140
Unit 10	Trees, Searching and Sorting techniques	141

Course Introduction

This book provides a complete guide for the implementation of data structures through C concepts. This book is very much helpful for your career development in the statistics field, so C programming with data structure is very much essential in this competitive world. There are large numbers of examples, practical questions, objectives, summaries of important topics, referral books and learning activities available in this book, which are very much useful for universities and job oriented examinations of various reputed companies. It is very useful for BCA, MCA and PGDCA students of Universities, computer institutions and so on.

This book covers four blocks and we discussed many topics with detailed explanation for each block; Block one deals with C programming language, block two deals with structured programming with C, block three deals with the Data structures in C and finally block four deals with Trees, searching and sorting and file organization.

Most of the concepts in the text are illustrated by several examples are important topics in their own right and may be treated as such. We feel that, at the stage of a student's development for which the text is designed, it is more important to cover several examples in great detail than to cover a broad range of topics cursorily. All the programs and algorithms in this text have been tested and debugged. Of course, any errors that remain are the sole responsibility of the authors. We have tried the best to avoid the mistakes and errors, however their presence cannot be ruled out. Your valuable suggestions and corrections are welcomed to improve our quality. This book is dedicated to all of our students and colleagues.

Block 1: Introduction

This block will teach you about the C Programming language fundamentals. This block is divided into three units.

Unit 1: This unit deals with the basic structure of a C program and has been discussed in detail with reference to preprocessor directives.

Unit 2 : This unit deals with the different data types in C and the basic input and output functions in C.

Unit 3 : This unit deals a detailed study on compiling and running a simple C program and also the various looping structures in C. It also includes an introduction to pointers.

Unit-1

Language Fundamentals

Structure

Overview

Learning Objectives

- 1.0 Introduction
- 1.1 C Programming language
- 1.2 Structure of a C Program
- 1.3 preprocessor directives
- 1.4 main() function

Let us sum up

Answer to learning activities

References

Overview

This unit is devoted to the basic introduction to C programming language and has been discussed in detail.

Learning Objectives

At the end of this unit you will have a clear understanding on the basic structure of a C program.

1.0 INTRODUCTION

The vehicle for the computer solution to a problem is a set of explicit and unambiguous instructions expressed in a programming language. This set of instructions is called a *program*. An algorithm corresponds to a solution to a problem that is independent of any programming language. To obtain the

computer solution to a problem once we have the program we usually have to supply the program with input or data. The program then takes this input and manipulates it according to its instructions and eventually produces an output which represents the computer solution to the problem. An *algorithm* consists of a set of explicit and unambiguous finite steps which, when carried out for a given set of initial conditions, produce the corresponding output and terminate in a finite time.

The problem solving aspect consists of the following:

- Problem definition phase – work out what must be done rather than how to do it; must try and extract from the problem statement set of precisely defined tasks.
- Getting started on a problem
- The use of specific examples – pick a specific example of the general problem we wish to solve and try to work out the mechanism that will allow us to solve this particular problem.
- Similarities among problems – bring in as much as past experience as possible to bear on the current problem, in this respect it is important to see if there are any similarities between the current problem and other problems that we have solved or have been solved. Develop an ability to view a problem from a variety of angles.
- Working backwards from the solution – assume that we already have the solution to the problem and then try to work backwards to the starting conditions.
- General problem-solving concepts – using most common principles such as *divide and conquer* strategy;

the original problem is divided into two or more subproblems. which can be solved by the same technique. If it is possible to proceed with this splitting into smaller subproblems we will eventually reach the stage where the subproblems are small enough to be solved without further splitting. The techniques such as *greedy search*, *backtracking* and *branch-and-bound* evaluations are variations of dynamic programming. They all tend to guide a computation in such a way that the minimum amount of effort is expended on exploring solutions that have been established to be suboptimal.

Any algorithm can be written as a program in any of the programming languages. These programming languages are nothing but a set of instructions which when given an input produce the required output.

1.1 C PROGRAMMING LANGUAGE

C is a general purpose language which features economy of expression, modern flow and data structures, and a rich set of operators. C was originally designed for and implemented on the UNIX operating system on the DEC PDP-11, by Dennis Ritchie in 1972. C is not tied to any particular hardware or system, however, and it is easy to write programs that will run without change on any machine that supports C. It is a relatively 'low level' language. It means that it deals with the same sort of objects such as characters, numbers and addresses. These may be combined and moved about with the usual arithmetic and logical operators implemented by actual machines. C is highly portable, that is C written for one computer can be run on another with little or no modification.

The characters that can be used to form words, numbers and expressions. The characters in C are grouped into the following categories:

1. Letters – Uppercase A --- Z, Lowercase a --- z

2. Digits – All decimal digits 0 --- 9

3. Special Characters –

, comma . period

; semicolon : colon

? question mark ‘ apostrophe

“ quotation mark ! exclamation mark

| vertical bar / slash

\ backslash ~ tilde

_ underscore \$ dollar sign

% per cent sign # number sign

& ampersand ^ caret

* asterisk - minus sign

+ plus sign < opening angle bracket or
less than sign

> greater than sign

(left parenthesis) right parenthesis

[left bracket] right bracket

{ left brace } right brace

4. White Spaces –

Blank space, Horizontal Tab, carriage return, New line,
Form feed

Identifiers are names given to various program elements such as variables, function and arrays. They consist of letters and digits in any order but the first character must be a letter. Uppercase and lowercase are allowed, however lowercase is preferred. Underscore can be included in the middle of an identifier.

Valid – x y12 sum_1 _temp TABLE
 Invalid - 4th “x” ord-no error flag

Some compilers recognize 1st 8 characters others accept first 31 characters.

Keywords are reserved words that have standard predefined meanings in C. They can be used for their intended purpose only. They cannot be used as programmer defined identifiers.

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

Some compilers include,

ada far near arm fortran
 pascalentry huge

1.2 C PROGRAM STRUCTURE

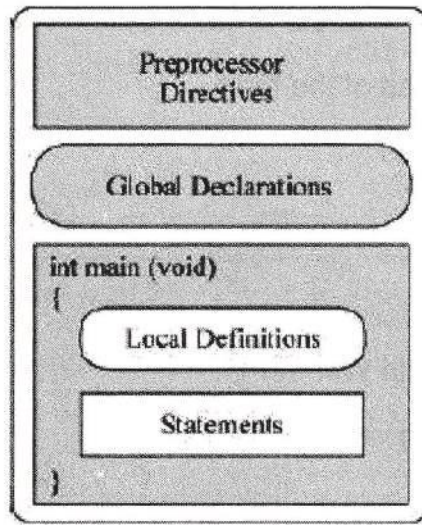


Figure 1.1 Structure of a C Program

1.3 PREPROCESSOR DIRECTIVES

The preprocessor commands include the needed library information for the programs we write. Preprocessor directives start with #

#include copies a file into the source code

#include <systemFilename>

#include "undefinedFilename"

#include <stdio.h>

- stdio.h is the standard input/output header
- .h files are header files
- Header files contain definitions

1.4 MAIN() Function

Programs must have a main() function. The two allowed formats are:

```
int main( void )
```

```
int main(int argc, char *argv[ ])
```

Our first C Program could be of the form,

```
#include <stdio.h>

main()
{
    printf ("Hello World\n");
} /* end of main */
```

The C language has a number of library functions which are not part of the language, but can be included. Some functions return a value, while others a true or false value. Some others carry out the function required but do not return any value. A typical set of library functions will include a large number of functions that are common to most C compilers. It is accessed by writing the function name, followed by a list of arguments about the information being passed to the function. The arguments must be enclosed in parenthesis and separated by commas. The arguments could be constants, variable names, or complex expressions. The parenthesis must be present even if there are no arguments.

Eg: Program reads in a lowercase character and converts it to uppercase and then writes out the uppercase character.

```
#include <stdio.h>

main()
{
    int lower, upper;
```

```
        lower = getchar();  
        upper = toupper (lower);  
        putchar(upper);  
    }
```

Program contains three library functions *getchar*, *toupper* and *putchar*. The first two functions each return a character (*getchar* from the keyboard and *toupper* the converted character). *Putchar* causes the character in uppercase to be displayed. *Getchar* has no arguments and hence empty parenthesis. The preprocessor statement `#include <stdio.h>` causes the contents of the file `stdio.h` to be inserted into the program at the start of the compilation process. The information contained in this file is essential for the proper functioning of the library functions *getchar* and *putchar*.

Let us sum up

We have covered about the basic concepts to programming and outlined the fundamental structure of a C program.

Learning Activities

a) Fill in the blanks:

1. A set of instructions is called a _____.
2. `#include <stdio.h>` is an example for _____.

b) State whether true or false:

- 1) An algorithm need not terminate in finite time.
- 2) The function `main()` is optional in a C program.

Answers to Learning Activities

a) Fill in the blanks:

1. program
2. Preprocessor Directive.

b) State whether true or false:

- 1) False.
- 2) False.

Model Questions

1. Explain in detail the concept of programming.
2. Write notes on the basic structure of a C program.
3. Show the use of library functions in a C program

References

1. Gottfried, B.S., "Schaum's Outline of Theory and Problems of Programming in C", Tata McGraw Hill, Delhi, 1995.
2. Kernighan, B.W. and Ritchi, D.M., The C Programming Prentice Hall of India, Delhi, 1998.

Unit-2

DATA TYPES AND I/O FUNCTIONS

Structure

Overview

Learning objectives

2.0 Data Types in C

2.1 Operators and Expressions

2.2 Input and Output Functions

2.3 Compiling and Running a C Program

Let us sum up

Answer to learning activities

References

Overview

This unit is devoted to the detailed study of the various data types in C. It covers the major input and output functions available in C.

Learning Objectives

At the end of this unit you will have a clear knowledge on the various data types of the several input and output functions in C and their implementation in basic programs.

2.0 DATA TYPES IN C

The basic data types in C are:

- » Integer
- » Character

- » Floating point
- » Double Floating point

Primitive Data Types

Data Type	C-Implementation
void	void
character	char (1 byte)
integer	unsigned short int (1 byte)
	unsigned int (2 or 4 bytes)
	unsigned long int (4 or 8 bytes)
floating point	short int (1 byte)
	int (2 or 4 bytes)
	long int (4 or 8 bytes)
floating point	float (4 bytes)
	double (8 bytes)
	long double (10 bytes)

Figure 2.1: Primitive Data Types in C

These can be augmented using qualifiers such as *short*, *long*, *signed* and *unsigned*.

CONSTANTS:

Numeric constants – are numeric values. Commas and blanks are not allowed. They can be preceded by minus sign. Their values cannot exceed specified minimum and maximum bounds.

Integer Constants are integer valued number. It is a sequence of digits, written in decimal, octal or hexadecimal.

```
Valid    -    0    1    743    9999
Invalid  -    12,245    36.0    10 20 30
          123-45-6789 0900
```

In octal,

Valid - 0 01 0743 07777

Invalid - 743 05280 777.1277

Floating point Constants are decimal numbers that contain either a decimal point or an exponent.

Valid - 1. 0.5 456.89 4000.

5.7E+8 2E7 0.008e-6

Eg : 1.2E-7 or 1.2e-7 0.18e-5 or 18e-7

9 x 10⁵ can be written as 90000.

9e5 9E5 9.0e+5

.9e6 0.9E6 90E4

90.E+4 900e3

Invalid -1 2,000.0 2E56.5 4E5 7

Character Constants are a single character enclosed in apostrophes. Has equivalent ASCII value.

'A' - 65 'Z' - 90

'a' - 97 'z' - 122

'1' - 49 '9' - 57

Escape sequences:

bell \a newline \n

backspace \b form feed \f

vertical tab \v carriage return \r

horizontal tab \t quotation mark \" \' \\\

null \0

String Constants consists of any number of consequent characters enclosed in double quotes.

Symbolic Constant is a name that substitutes for a sequence of characters. The characters may represent a numeric constant, a character or a string constant. When the program is compiled, each occurrence of a symbolic constant is replaced by its corresponding character sequence. These constants are usually defined at the beginning of the program. It is defined by writing,

```
#define      name      text
```

where *name* represents a symbolic name, typically written in uppercase letters and *text* represents the sequence of characters associated with the symbolic name. Note that *text* does not end with a semicolon, as a symbolic constant definition is not a true C statement.

```
#define      TAXRATE    0.65
#define      PI          3.141593
#define      TRUE        1
#define      FALSE       0
#define      SUBJECT     "Operating Systems"
```

VARIABLES:

A variable is an identifier that is used to represent some specified type of information within a designated portion of the program. A variable represents a single data item, that is, a numeric quantity or a character constant. The data item has to be assigned to the variable at some point in the program. The data item can be accessed later in the program simply by referring to the variable.

Eg:

```
int a, b, c;  
char d;  
  
a = 2;  
b = 5;  
c = 9;  
d = 'y';
```

The first two lines are type declarations which specify that variables a, b, c are integer variables and d is a character variable. In the following lines corresponding values are assigned to the variables.

DECLARATIONS:

A declaration associates a group of variables with a specific data type. All variables must be declared before they are used in the executable statements. A declaration consists of a data type followed by one or more variable names, ending with a semicolon. Initial values can be assigned to variables within a type declaration. To do so, the declaration must consist of a data type, followed by a variable name, an equal sign (=) and a constant of the appropriate type. A semicolon must be written at the end.

Eg:

```
int x = 10;  
char sname = 'a';  
float avg = 3.5;  
double sum = 0.45 x 10-5;
```

The value of any expression can be converted to a different data type if desired. To do so, the expression must be preceded by the name of the desired data type, enclosed in parenthesis.

(data type) expression

This type of construction is called as a cast. It is also called as type cast.

Eg:

If i is of type integer (value 4) and j of type float (value 8.5), the expression, $(i + j) \% 4$ is invalid as $(i + j)$ is not integer type. If the expression is changed as, $((\text{int}) (i + j)) \% 4$, it forces the first operand to be an integer and is thus valid.

EXPRESSIONS:

An expression represents a single data-item such as a number or a character. It may contain a single entity, such as a constant, a variable, an array element or a reference to a function. It may also contain some combination of such entities interconnected by one or more operators. Expressions can also represent logical conditions that are either true or false.

Eg:

a + b x = y c = x - z
t == u b <= c ++i

STATEMENTS:

A statement causes the computer to carry out some action. Some of the different classes of statements in C are *expression statements* (causes the value of the expression on the right of the equal sign to be assigned to the variable on the left), *compound statements* (consists of several individual

statements enclosed within a pair of braces { }. The individual statements may be expression statements, compound statements or control statements. Thus this type has the capacity for embedding statements within other statements) and *control statements* (used to create special program features such as logical tests, loops and branches).

2.1 OPERATORS AND EXPRESSIONS

The data items that operators act upon are called operands. Some operators require two operands while some others act upon single operand. Most operators allow the individual operands to be expressions. The operators in C are:

1. Arithmetic
2. Unary
3. Relational
4. Logical
5. Assignment
6. Conditional

Arithmetic Operators: The five operators are addition (+), subtraction (-), multiplication (*), division (/) and remainder after integer division (%), also referred as modulus operator. Operands can be integer, floating point or character (the ASCII values will be taken). The modulus operator and the division operator require the second operand to be non-zero.

Eg.

1. Assume $a = 15$ and $b = 10$, so $a+b = 25$; $a-b = 5$; $a * b = 150$; $a/b = 1$; $a \% b = 5$ (as a and b are integers the decimal values are neglected in division).

2. Assume $a = 15.5$ and $b = 10.2$, so $a+b = 25.7$; $a-b = 5.3$; $a * b = 158.1$; $a/b = 1.52$; $a \% b = 5.2$
3. If a and b are characters, $a+b = 97+98=195$; $b-a = 1$; $a+b-100 = 98$
4. If negative values are used, $a = -2$ and $b = 5$, $a+b = 3$; $a-b = -7$; $a*b = -10$; $a/b = -0.4$;

Unary Operators is a class of operators that act upon single operand to produce a new value. The operators usually precede the operands.

Unary minus operator precedes the positive constant value - -543, -0FF56, -0.7, $-3 * (x + y)$

Increment operator causes its operand to be increased by one – if the variable i is assigned a value 7, $i++$ is equivalent to $i = i + 1$.

Decrement operator causes its operand to be decreased by one - if the variable i is assigned a value 5, $i--$ is equivalent to $i = i - 1$.

The *sizeof* operator returns the size of its operand in bytes. It always precedes its operand.

Eg:

1. `printf ("Integer is %d\n", sizeof(integer));`
2. `char text[] = "California";`
`printf ("Number of characters = %d\n", sizeof text);`
 Number of character = 11

The *cast* operator is also an unary operator.

The **!** operator negates the value of any logical expression (the original expression if true becomes false and

vice versa). The operator is called as logical negation or *logical not* operator.

Relational and Logical Operators: The following operators have the same precedence and fall below the precedence of unary and arithmetic operators.

<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to

the associativity of these operators is left-right.

The equality operators == (equal to) and != (not equal to) fall into separate precedence group below that of the relational operators. These 6 operators are used to form logical expressions which represent conditions that are either true or false. The resulting expressions will be of type integer and so the true value will be 1 and false 0.

The logical operators are && (and) and || (or). These are referred to as *logical AND* and *logical OR*.

Assignment Operators are used to form assignment expressions which assign the value of an expression to an identifier. The format is

identifier (operator) expression

The operators are = += -= *= /= %=

Eg:

expression1 += expression2 is equal to
expression1 = expression1 + expression2

Conditional Operator is used for simple conditions. The conditional expression is written as

expression1 ? *expression2* : *expression3*

When evaluating the conditional expression, *expression1* is evaluated first. If *expression1* is true (value is nonzero), then *expression2* is evaluated and this becomes the value of the conditional expression. If *expression1* is false, then *expression3* is evaluated and this becomes the value of the conditional expression.

The precedence for operators is as follows:

Unary	-, ++, --, !, sizeof, (type)	R→L
Arithmetic	*, /, %, +, -	L→R
Relational	<, <=, >, >=	L→R
Equality	==, !=	L→R
Logical and	&&	L→R
Logical or		L→R
Conditional	? :	R→L
Assignment	= += -= *= /= %=	R→L

2.2 INPUT AND OUTPUT FUNCTIONS

The six major input/output functions available in the standard C library are `getchar`, `putchar`, `scanf`, `printf`, `gets`, `puts`. These permit the transfer of information between the computer and the standard i/o devices.

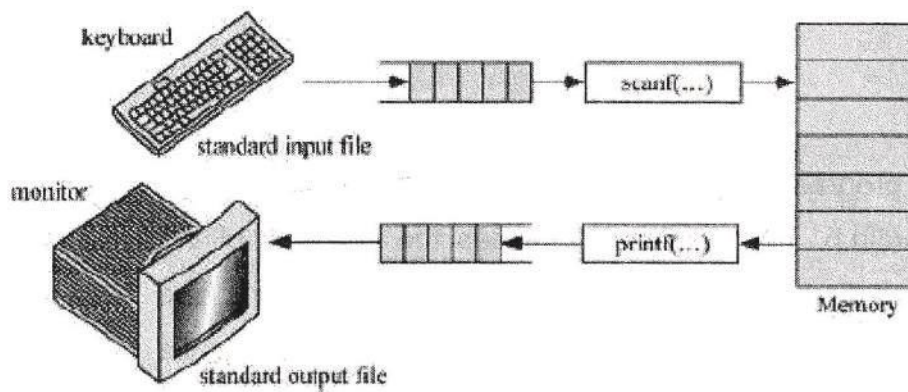


Figure 2.2: I/O functions

Getchar and putchar allow single characters to be transferred into and out of the computer; scanf and printf permit the transfer of single characters, numerical values and strings; gets and puts facilitate the input and output of strings. An input/output function can be accessed from anywhere within a program simply by writing the function name, followed by a list of arguments enclosed by parenthesis. The arguments represent data items that are sent to the function. C includes a collection of header files that provide necessary information (eg. Symbolic constants) in support of the various library functions. These files are entered into the program via an #include statement at the beginning of the program. The header file for standard input/output library functions is called stdio.h.

```
#include <stdio.h>

main()
{
    char c;

    clrscr();

    c = getchar();
```

```
        putchar(c);  
    getch();  
}
```

The program begins with the preprocessor statement `#include <stdio.h>`. This statement causes the contents of the header file `stdio.h` to be included within the program. The header file supplies required information for the library functions used in the program. The next statement in the program heading `main()`, followed by variable declarations. The statement `c = getch()` causes a single character to be entered from the keyboard and assigned to the variable `c`. The reference of `putchar` causes the value of the character variable `c` to be displayed.

getchar() – single characters can be entered into the computer using the C library function `getchar()`. It returns a single character from a standard input device (typically the keyboard). The function does not require any arguments, though a pair of empty parenthesis must follow the word `getchar()`. A reference to the `getchar()` function is written as, *character variable = getchar();* where *character variable* refers to the previously declared character variable. This function can also be used to read multicharacter strings by reading one character at a time within a multipass loop. When an end-of-file condition is encountered when reading a character with the `getchar` function, the value of the symbolic constant `EOF` will automatically be returned. This value will be assigned within the `stdio.h` file. Typically `EOF` will be assigned the value `-1`. Using a `if – else` statement the detection and corrective action for `EOF` condition can be taken care.

putchar() – single characters are displayed using the C library function `putchar()`. The function is complementary to the character input function `getchar()`. The character being transmitted by the function will normally be represented as a character-type variable. It must be expressed as an argument to the function, enclosed in parenthesis, following the word `putchar`. A reference to the `putchar` function is written as, *putchar(character variable)*, where character variable refers to the previously declared character variable. This function can also be used to output multicharacter strings by displaying one character at a time within a multipass loop.

scanf() – the function can be used to enter any combination of numerical values, single characters and strings. The function returns the number of data items that have been successfully entered. The `scanf` function is written as,

scanf (control string, arg1, arg2, argn);

where control string contains certain formatting information and *arg1, arg2, Argn* are arguments that represent the individual data items. The control string comprises of individual groups of characters, with one character group for each input data item. Each character group must begin with a percent sign (%), followed by a conversion character. White spaces can be used in the control string to separate the groups of characters. The conversion characters are:

- 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 whitespace character (the null character \0 is automatically added at the end)
- u data item is a unsigned decimal integer
- x data item is a hexadecimal integer

Each variable must be preceded by an ampersand (&),as they indicate the memory addresses.

```

#include <stdio.h>

main()
{
    char a;
    int x;
    float y;
    double z;

    clrscr();
    scanf ("%c %d %f %f", &c, &x, &y, &z);
    getch();
}

```

The control string is "%c %d %f %f". It contains four character groups. The character group %c indicates that the first argument (item) represent character. The second

character group indicates that the second argument represents an integer. The third character group indicates that the third argument represents a decimal integer value. The fourth character group indicates that the fourth argument represents a double precision decimal integer value. Each argument is preceded by & sign. When entering input, they should be entered with whitespaces in between each argument. If instead of whitespaces, if we want to enter each data item in a separate line we include the new line character. The scanf is modified as,

```
scanf ("\n%c",&c);  
scanf (" \n%d", &x);  
scanf (" \n%f", &y);  
scanf ("\n%f", &z);
```

or

```
scanf ("\n%c \n%d \n%f \n%f", &c, &x, &y, &z);
```

The control string can also include the length of the numeric values entered. If the scanf was modified as,

```
scanf ("%4d", &x);
```

it indicates that the integer number that is entered is of length 3 or 3 digits. If we have a scanf statement as

```
scanf ("%4d %4d %4d", &x1, &x2, &x3);
```

and an input is give as, 1 2 3 the values are assigned as, x1 = 1; x2 = 2; x3 = 3;

If the data entered is 2345 6783 2312 the values assigned are x1=2345; x2=6783; x3=2312.

If the data entered is 12345 23 123 the values assigned are x1=1234; x2=23; x3=123 as the length is 4, in x1 the last digit is ignored.

Other conversion characters include,

- l - signed or unsigned long integer or double precision argument
- h - signed or unsigned short integer
- L - long double

Eg: #include <stdio.h>

```
main()
{
    short ix, iy;
    long lx, ly;
    double dx, dy;
```

```
scanf ("%hd, %ld, %lf", &ix, &lx, &dx);
scanf ("%3ho, %7lx, %15le", &ix, &lx, &dx);
}
```

The first control string indicates first data item to be assigned to a short decimal integer variable, second to a long decimal integer and third to a double-precision value. In the second control string, the first data item is assigned to a short octal integer of maximum field width of 3 characters, second to a long hexadecimal integer of width 5 character and third to a double precision value of max. width 15 characters. For a string data type, the scanf can be as,

```
char
sname[12];
```

```
scanf ("%[\n]", sname);
```

```
printf ("%s", sname);
```

This format accepts input into sname till a RETURN KEY is pressed.

printf() - the function can be used to display any combination of numerical values, single characters and strings.

The scanf function is written as,

```
scanf (control string, arg1, arg2, .... argn);
```

where control string contains certain formatting information and *arg1, arg2, Argn* are arguments that represent the individual data items. The control string comprises of individual groups of characters, with one character group for each output data item. Each character group must begin with a percent sign (%), followed by a conversion character. White spaces can be used in the control string to separate the groups of characters.

The conversion characters are:

- c data item is displayed as a single character
- d data item is displayed as a decimal integer
- e data item is displayed as a floating point value
- f data item is displayed as a floating point value
- g data item is displayed as a floating point value
- h data item is displayed as a short integer
- i data item is displayed as a decimal,
hexadecimal or octal integer
- o data item is displayed as a octal integer
- s data item is displayed as a string followed by a

whitespace character (the null character `\0` is automatically added at the end)

- u data item is displayed as a unsigned decimal integer
- x data item is displayed as a hexadecimal integer

In contrast to the `scanf` function, the arguments do not represent memory addresses and therefore they are not preceded by an ampersand (`&`).

```
#include <stdio.h>

main()
{
    char  sname[10];
        int   x;
        float  y;
        double z;

        clrscr();

scanf ("%s  %d  %f  %f", sname, &x, &y, &z);
printf((" %s  %d  %f  %f", sname, x, y, z);

        getch();
}
```

The first control string indicates first data item is to be assigned to a string variable, second to a decimal integer and third to a floating point value and the fourth to a double-precision value. The conversion character `%e` is different from `%f` in that `%e` displays the output as exponential.

```
#include <stdio.h>

main()
```

```

    { double      x = 5000.0, y = 00.25;
printf ("%f      %f      %f      %f\n\n", x, y, x*y, x/y);
printf ("%e      %e      %e      %e\n\n", x, y, x*y, x/y);
    }

```

the output is given by

```

5000.000000   0.002500   12.500000   2000000.0000
5.000000e003  2.500000e-003   1.250000e001
2.000000e+006

```

A minimum field width can be specified by preceding the conversion character by an unsigned integer. If the number of characters in the corresponding data item is less than the specified field width, then the data item will be preceded by enough leading blanks to fill the specified field. If the number of characters in the data item exceeds the specified field width, however, then additional space will be allocated to the data item, so that the entire data item will be displayed. This is just the opposite of the field width indicator in the scanf function which specifies a maximum field width. The means by which a maximum field width is specified is called as precision. The precision of an unsigned integer is always preceded by a decimal point. If a minimum field width is specified in addition to the precision then the precision specification follows the field width specification.

```

#include <stdio.h>

main()
{
    float      x = 123.456;

```

```

printf ("%7f  %7.3f %7.1f", x, x, x);
printf ("%12e %12.5e%12.3e\n\n", x, x, x);
}

```

the output is,

```

123.456000 123.456      123.5
1.234560e+002      1.23456e+002
1.2345e+002

```

gets() and puts() – facilitate transfer of strings between the computer and the i/o devices. The string ends when the user presses the RETURN KEY. The gets and puts function offer simple alternatives of the use of scanf and printf for reading and displaying strings.

```

#include <stdio.h>

main()
{
char    x[20];

gets(x);

puts(x);

}

```

2.3 COMPILING AND RUNNING A C PROGRAM

The stages of developing your C program are as follows:

CREATING THE PROGRAM:

Create a file containing the complete program, such as the above example. You can use any ordinary editor with which you are familiar to create the file. One such editor is ***textedit***

available on most UNIX systems. The filename must by convention end ``.c" (full stop, lower case c), *e.g. myprog.c* or *progtest.c*. The contents must obey C syntax. For example, they might be as in the above example, starting with the line /* Sample (or a blank line preceding it) and ending with the line } /* end of program */ (or a blank line following it).

COMPILATION:

There are many C compilers around. The `cc` being the default Sun compiler. The GNU C compiler `gcc` is popular and available for many platforms. PC users may also be familiar with the Borland `bcc` compiler. There are also equivalent C++ compilers which are usually denoted by `CC` (*note* upper case `CC`). For example Sun provides `CC` and GNU `gcc`. The GNU compiler is also denoted by `g++`. Other (less common) C/C++ compilers exist. All the above compilers operate in essentially the same manner and share many common command line options. The Turbo C version on DOS has an inbuilt compiler that executes the program.

If there are obvious errors in your program (such as mistypings, misspelling one of the key words or omitting a semi-colon), the compiler will detect and report them. There may, of course, still be logical errors that the compiler cannot detect. You may be telling the computer to do the wrong operations. When the compiler has successfully digested your program, the compiled version, or executable, is left in a file called *a.out* or if the compiler option `-o` is used : the file listed after the `-o`.

It is more convenient to use a `-o` and filename in the compilation as in

```
cc -o program program.c
```

which puts the compiled program into the file program (or any file you name following the "-o" argument) **instead** of putting it in the file a.out .

RUNNING THE PROGRAM:

The next stage is to actually run your executable program. To run an executable in UNIX, you simply type the name of the file containing it, in this case ***program*** (or ***a.out***). This executes your program, printing any results to the screen. At this stage there may be run-time errors, such as division by zero, or it may become evident that the program has produced incorrect output.

If so, you must return to edit your program source, and recompile it, and run it again.

THE C COMPILATION MODEL:

We will briefly highlight key features of the C Compilation model here.

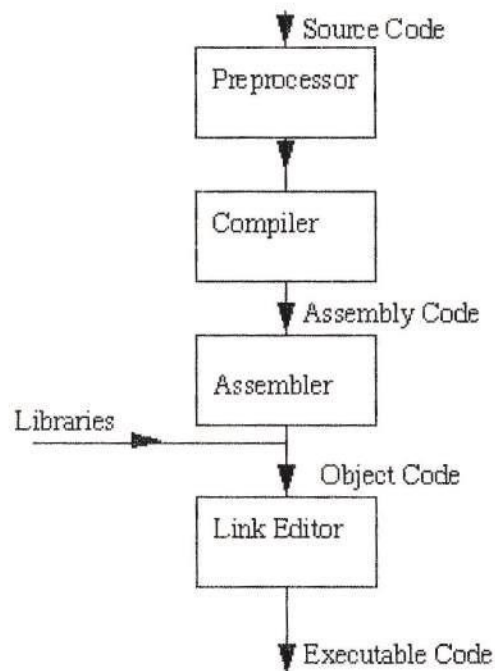


Figure 2.3: The C Compilation Model

THE PREPROCESSOR:

The Preprocessor accepts source code as input and is responsible for

- removing comments
- interpreting special preprocessor directives denoted by #.

For example:

`#include` -- includes contents of a named file. Files usually called **header** files. *e.g*

`#include <math.h>` -- standard library maths file.

`#include <stdio.h>` -- standard library I/O file

`#define` -- defines a symbolic name or constant. Macro substitution.

`#define MAX_ARRAY_SIZE 100`

C COMPILER:

The C compiler translates source to assembly code. The source code is received from the preprocessor.

ASSEMBLER:

The assembler creates object code. On a UNIX system you may see files with a `.o` suffix (`.OBJ` on MSDOS) to indicate object code files.

LINK EDITOR:

If a source file references library functions or functions defined in other source files the **link editor** combines these functions (with `main()`) to create an executable file. External Variable references resolved here also.

USING LIBRARIES:

C is an extremely small language. Many of the functions of other languages are not included in C. **e.g.** No built in I/O, string handling or maths functions. C provides functionality through a rich set function libraries. As a result most C implementations include **standard** libraries of functions for many facilities (I/O **etc.**). For many practical purposes these may be regarded as being part of C. But they may vary from machine to machine. (**cf** Borland C for a PC to UNIX C).

A programmer can also develop his or her own function libraries and also include special purpose third party libraries (**e.g.** NAG, PHIGS). All libraries (except standard I/O) need to be explicitly linked in with the -l and, possibly, -L compiler options described above.

```
#include <time.h>
```

```
char *ctime(time_t *clock)
```

This means that you must have

```
#include <time.h>
```

in your file before you call ctime. And that function ctime takes a pointer to type time_t as an argument, and returns a string (char *). The library gives description for functions.

ctime() converts a long integer, pointed to by clock, to a 26-character string of the form produced by asctime().

Let us sum up

We have covered about the basic data types in C. We have also studied in detail the various input and output features in C.

Learning Activities

a) Fill in the blanks:

1. The four primitive data types are _____, _____, _____ and _____ .

b) State whether true or false:

- 1) gets() and getchar are basic output functions in C.
- 2) Control strings are included to produce formatted output in printf statements.
- 3) The symbol & is used to indicate the logical AND operator in C.

Answers to Learning Activities

a) Fill in the blanks:

1. int, char, float and double.

b) State whether true or false:

- 1) False.
- 2) True.
- 3) False.

Model Questions

1. Explain the primitive data types and their use in C in detail.
2. Write detailed notes on the data input and output statements in C.

References

1. Gottfried, B.S., "Schaum's Outline of Theory and Problems of Programming in C", Tata McGraw Hill, Delhi, 1995.
2. Kerninghan, B.W. and Ritchi, D.M., The C Programming Prentice Hall of India, Delhi, 1998

Unit-3

Control Structures

Structure

Overview

Learning objectives

3.0 IF-ELSE statements

3.1 SWITCH Statements

3.2 Looping structures in C

3.3 Pointers in C

Learning activities

Let us sum up

Answer to learning activities

References

Overview

This unit is devoted to the detailed study of the various Control statements available in C.

Learning Objectives

At the end of this unit you will have a clear knowledge on the ways of representing control over the flow of a C program.

3.0. IF-ELSE STATEMENTS

This is used to decide whether to do something at a special point, or to decide between two courses of action.

The following test decides whether a student has passed an exam with a pass mark of 45

```

if (result >= 45)
    printf("Pass\n");
else
    printf("Fail\n");

```

It is possible to use the if part without the else.

```

if (temperature < 0)
    print("Frozen\n");

```

Each version consists of a test, (this is the bracketed statement following the if). If the test is true then the next statement is obeyed. If it is false then the statement following the else is obeyed if present. After this, the rest of the program continues as normal.

If we wish to have more than one statement following the if or the else, they should be grouped together between curly brackets. Such a grouping is called a compound statement or a block.

```

if (result >= 45)
{
    printf("Passed\n");
    printf("Congratulations\n");
} else
{
    printf("Failed\n");
    printf("Good luck in the resits\n");
}

```

Sometimes we wish to make a multi-way decision based on several conditions. The most general way of doing this is by using the else if variant on the if statement. This works by cascading several comparisons. As soon as one of these gives

a true result, the following statement or block is executed, and no further comparisons are performed. In the following example we are awarding grades depending on the exam result.

```
if (result >= 75)
    printf("Passed: Grade A\n");
else if (result >= 60)
    printf("Passed: Grade B\n");
else if (result >= 45)
    printf("Passed: Grade C\n");
else
    printf("Failed\n");
```

In this example, all comparisons test a single variable called result. In other cases, each test may involve a different variable or some combination of tests. The same pattern can be used with more or fewer else if's, and the final lone else may be left out. It is up to the programmer to devise the correct structure for each programming problem.

3.1. SWITCH STATEMENTS

This is another form of the multi way decision. It is well structured, but can only be used in certain cases where;

- Only one variable is tested, all branches must depend on the value of that variable. The variable must be an integral type. (int, long, short or char).
- Each possible value of the variable can control a single branch. A final, catch all, default branch may optionally be used to trap all unspecified cases.

Hopefully an example will clarify things. This is a function which converts an integer into a vague description. It is useful where we are only concerned in measuring a quantity when it is quite small.

```
estimate(number)  
  
int number;  
  
/* Estimate a number as none, one, two,  
several, many */  
  
{ switch(number) {  
  case 0 :  
    printf("None\n");  
    break;  
  
  case 1 :  
    printf("One\n");  
    break;  
  
  case 2 :  
    printf("Two\n");  
    break;  
  
  case 3 :  
  
  case 4 :  
  
  case 5 :  
    printf("Several\n");  
    break;  
  
  default :  
    printf("Many\n");
```

```

        break;
    }
}

```

Each interesting case is listed with a corresponding action. The break statement prevents any further statements from being executed by leaving the switch. Since case 3 and case 4 have no following break, they continue on allowing the same action for several values of number.

Both if and switch constructs allow the programmer to make a selection from a number of possible actions.

The other main type of control statement is the loop. Loops allow a statement, or block of statements, to be repeated. Computers are very good at repeating simple tasks many times, the loop is C's way of achieving this.

Finding the greatest of 3 given numbers.

```

#include <stdio.h>

main()
{
    int a, b, c;

    printf ("\nEnter the three numbers : ");
    scanf ("%d %d %d", &a, &b, &c);

    if ( (a > b) && (a > c)
        printf ("\n a is the greatest");
    else if (b > c)
        printf ("\n b is the greatest");
    else
        printf ("\n c is the greatest");

    getch();
}

```

Finding the Area of a circle or a rectangle (if ans = 'C' area of circle else area of rectangle)

```
#include <stdio.h>

#define PI 3.14

main()
{
float r, area, volume;

char ans;

printf ("\nEnter radius of the circle : ");

scanf ("%f", &r);

printf ("\nWhat do want to find? Area or Volume.
        Press 'A' or 'V'");

scanf ("%c", &ans);

if (ans == 'A')                {
    area = PI * r * r;

    printf ("\nArea = %f", area);}

else if (ans == 'V')          {
    volume = PI *4/3 * r * r * r;

    printf ("\nVolume = %f",volume); }

}
```

3.2 LOOPING STRUCTURES IN C

The statements so far discussed were executed only once in our program. But in practice we require the repeated execution of certain statements to give us the right results. Thus we need

1. instructions to be executed more than once
2. tests to determine if certain conditions are true or false

Looping is a group of instructions to be executed repeatedly, until some logical condition has been satisfied. The number of repetitions could be known (eg. the details of 10 students) or unknown (a group of instructions that are repeated until the logical condition becomes true).

CONDITIONAL EXECUTION:

Programs require that a logical test be carried out at some particular point, within the program. An action will then be carried out whose exact nature depends upon the outcome of the logical test. We need to form logical expressions that are either true or false. We use the relational operators (<, <=, >, >=) and the equality operators (=, !=) along with operands. In addition to these the logical connectives && (AND), || (OR) and the unary operator (!) are used.

Eg:

```
counter <= 200          (cnt <=200) && (c == '#')
sqrt (b+c) > 0.09
(color == 'R') || (color == 'B')
ans = 0      (ans > 0) && ((ans < 5) || (ans != 0))
balance >= income
! ((pay >= 1000) && (status == 'S'))
c > 't'
letter != 'a'
```

Since the equality operator has higher precedence than logical operator parenthesis can be avoided in some places.

While statement:

```
while <expression>
    statements
```

The included statements will be executed repeatedly as long as the value of the expression is not zero (that is the statement is true). The statements could be simple or compound. *expression* is usually a logical expression that is either true or false (false 0, true is non zero value).

```
main()
{
    int digit = 0;
    while (digit <= 9)    {
        printf ("%d\n", digit);
        digit++;    }
}
```

The loop is run 10 times resulting in 10 consecutive lines of output. The *printf* statement could be rewritten as,

```
printf ("%d\n", digit++);
```

while (character is not end of file signal, output the character just read, get a new character)

```
#include <stdio.h>
#define EOF 0
main()
{
```



```

int c;

c = getchar();

while (c != EOF) {
    putchar (c);
    c = getchar();}
}

```

c is declared to be an *int* and not as a character, so that it can hold the value which the *getchar()* function returns. EOF will be either 0 or -1, and they thus define as

```

#define EOF 0 or
#define EOF -1

#include <stdio.h>

#define EOF 0

main()
{ int c;

while ( (c = getchar()) != EOF )
    putchar (c);           (or)
    printf ("%c", toupper (c));

}

```

Do - while statement:

This is very similar to the while loop except that the test occurs at the end of the loop body. This guarantees that the loop is executed at least once before continuing. Such a setup is frequently used where data is to be read. The test then verifies the data, and loops back to read again if it was unacceptable.

```

do
{   printf("Enter 1 for yes, 0 for no :");
    scanf("%d", &input_value);
} while (input_value != 1 && input_value != 0)

```

For statement:

The for loop works well where the number of iterations of the loop is known before the loop is entered. The head of the loop consists of three parts separated by semicolons.

- » The first is an expression that specifies an initial value for an index. (parameter that controls the looping action – assignment exp.)
- » The second is a test (determines whether or not the loop is continued) the loop is exited when this returns false (logical exp.)
- » The third is a index to be modified at the end of each pass. This is usually an increment of the loop counter (unary or assignment exp.)

Syntax is,

for (exp1; exp2; exp3)

```

#include <stdio.h>

main()
{   int digit;
    for (digit = 0; digit <= 9; digit++)
        printf ("%d\n", digit);
}

```

Any one of the expressions could be omitted,

```

int digit = 0;
    for (; digit <= 9; digit)
        printf ("%d\n", digit++);

```

Below given example is a function which calculates the average of the numbers stored in an array. The function takes the array and the number of elements as arguments.

```

float average(float array[], int count)
{
    float total = 0.0;
    int i;
    for(i = 0; i < count; i++)
        total += array[i];
    return(total / count);
}

```

The for loop ensures that the correct number of array elements are added up before calculating the average. The three statements at the head of a for loop usually do just one thing each, however any of them can be left blank. A blank first or last statement will mean no initialisation or running increment. A blank comparison statement will always be treated as true. This will cause the loop to run indefinitely unless interrupted by some other means. This might be a return or a break statement.

It is also possible to squeeze several statements into the first or third position, separating them with commas. This allows a loop with more than one controlling variable. The example below illustrates the definition of such a loop, with variables hi and lo starting at 100 and 0 respectively and converging.

```

for (hi = 100, lo = 0; hi >= lo; hi--, lo++)

```

The for loop is extremely flexible and allows many types of program behaviour to be specified simply and quickly.

Nested loops:

- » One loop should be completely embedded within the other
- » There can be no overlap
- » Each loop must be controlled by a different index

The break Statement:

We have already met break in the discussion of the switch statement. It is used to exit from a loop or a switch, control passing to the first statement beyond the loop or a switch. With loops, break can be used to force an early exit from the loop, or to implement a loop with a test to exit in the middle of the loop body. A break within a loop should always be protected within an if statement which provides the test to control the exit condition.

The continue Statement:

This is similar to break but is encountered less frequently. It only works within loops where its effect is to force an immediate jump to the loop control statement.

- » In a while loop, jump to the test statement.
- » In a do while loop, jump to the test statement.
- » In a for loop, jump to the test, and perform the iteration.

Like a break, continue should be protected by an if statement. You are unlikely to use it very often.

```
do {
```

```

scanf ("\n%f", &x);
if (x < 0)
    {
        printf ("ERROR – Enter non-negative value");
        continue;    }
} while (x <= 100);

```

The goto Statement:

C has a goto statement which permits unstructured jumps to be made. It is used to alter the normal sequence of program execution by transferring control to some other part of the program.

```
goto label;
```

where label is an identifier used to label the target statement to which control will be transferred. Control may be transferred to any other statement within the program.

```

scanf ("%f", &x);
while (x <= 100)
    {
        ..... if (x < 0) goto errorcheck;
        .....
    }
errorcheck : { printf ("\nERROR – Enter the value for x");
              getch(); }

```

goto statement is used,

- » To branch around statements or a group of statements under certain conditions
- » Jumping to the end of a loop under certain conditions, thus bypassing the remainder of the loop during current pass

- » Jumping completely out of the loop under certain conditions, thus terminating execution of the loop

GOTO statement is not recommended, as

- branching can be done by if-else
- jumping to the end by continue
- jumping out of the loop by break

Usage of goto statement does not encourage structured programming.

The comma operator:

It is used primarily in conjunction with the *for statement*. It permits 2 different expressions to appear in situations where only one expression could ordinarily be used.

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

eg. search for palindromes

3.3 POINTERS IN C

Pointers are data items in memory. It is a variable which contains the address of another variable. Each variable when declared, is allocated a specific area in the computer's memory. The operating system takes care of the address of the variable and does the necessary conversion each time the variable is accessed by the user. The declaration `int a = 10;` assigns a cell called `a` and stores the value 10 to it.

a

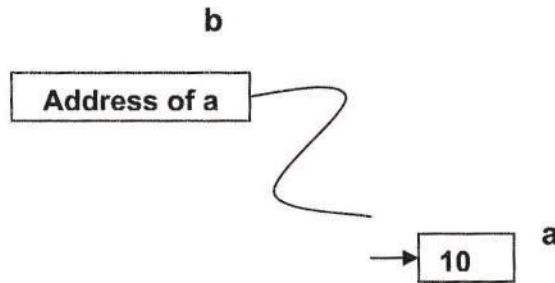
10

```
int a = 10;
```

```
int *b;
```

```
b = &a;
```

The above declaration means that a is an integer with value 10. b contains the address of an integer. The variable b points to an integer and thus is a pointer variable. It points in the direction of the integer a and has the address of a. The '&' sign is called as the address operator and '*' is called as the indirection operator.



Let us sum up

We have covered about the basic branching and control structures in C.

Learning Activities

a) Fill in the blanks:

1. The fundamental branching structures in C are _____ and _____.
2. The FOR loop has to have atleast _____ parameters.

b) State whether true or false:

- 1) The usage of GOTO statements is efficient.
- 2) The working of while statement and do-while are one and the same.

Answers to Learning Activities

a) Fill in the blanks:

- 1) if and switch statements

2) One.

b) State whether true or false:

1) False.

2) False.

Model Questions

1. What are compilers? Show the compilation process of a C program.
2. What are library function? How are they useful in C programs?
3. Describe in detail the IF-ELSE construct in C.
4. What are the types of branching statements available in C?
5. Write detailed notes on the various looping features available in C.

References

1. Kamthane, "Programming with ANSI and Turbo C, Pearson Education, Delhi, 2002.
2. Al Kelley, Iya Pohl.; "A Book on C", Pearson Education, Delhi, 2001.

Block 2: Introduction

This block will teach you about the Structured Programming concepts in C. This block is divided into three units.

Unit 4: This unit deals with the concepts of Arrays, Structures and Unions in C and has been discussed in detail.

Unit 5: This unit deals with the different storage classes in C and the use of functions in C programming.

Unit 6: This unit deals on a detailed study on File handling features in C.

Unit-4

ARRAYS & STRUCTURES

Structure

Overview

Learning Objectives

4.0 Concept of Arrays in C

4.1 Structures

4.2 Unions

4.3 Bit fields

Let us sum up

Answer to learning activities

References

Overview

This unit is devoted to the study of derived data types such as arrays and structures in C and has been discussed in detail.

Learning Objectives

At the end of this unit you will have a clear understanding on the concepts of arrays and structures.

4.0 CONCEPT OF ARRAYS IN C

An array is a set of variables, represented by a single name. The individual variables are called elements and are identified by index numbers. The following example declares an array with ten elements.

```
int x[10];
```

The first element in the array has an index number of zero. Therefore, the above array has ten elements, indexed from zero to nine.

ACCESSING THE ELEMENTS

To access an individual element in the array, the index number follows the variable name in square brackets. The variable can then be treated like any other variable in C. The following example assigns a value to the first element in the array.

```
x[0] = 16;
```

The following example prints the value of the third element in an array.

```
printf("%d\n", x[2]);
```

The following example uses the `scanf` function to read a value from the keyboard into the last element of an array with ten elements.

```
scanf("%d", &x[9]);
```

INITIALISING ARRAY ELEMENTS

Arrays can be initialised like any other variables by assignment. As an array contains more than one value, the individual values are placed in curly braces, and separated with commas. The following example initialises a ten dimensional array with the first ten values of the three times table.

```
int x[10] = {3, 6, 9, 12, 15, 18, 21, 24, 27, 30};
```

This saves assigning the values individually as in the following example.

```
int x[10];  
x[0] = 3;  
x[1] = 6;  
x[2] = 9;  
x[3] = 12;  
x[4] = 15;  
x[5] = 18;  
x[6] = 21;  
x[7] = 24;  
x[8] = 27;  
x[9] = 30;
```

LOOPING THROUGH AN ARRAY:

As the array is indexed sequentially, we can use the for loop to display all the values of an array. The following example displays all the values of an array.

```
#include <stdio.h>  
  
int main()  
{  
    int x[10];  
    int counter;  
    /* Randomise the random number generator */  
    srand((unsigned)time(NULL));  
    /* Assign random values to the variable */  
    for (counter=0; counter<10; counter++)  
        x[counter] = rand();  
}
```

```

    /* Display the contents of the array */
    for (counter=0; counter<10; counter++)
        printf("element %d has the value %d\n", counter,
x[counter]);
    return 0;
}

```

MULTIDIMENSIONAL ARRAYS:

An array can have more than one dimension. By allowing the array to have more than one dimension provides greater flexibility. For example, spreadsheets are built on a two dimensional array; an array for the rows, and an array for the columns. The following example uses a two dimensional array with two rows, each containing five columns.

```

#include <stdio.h>

int main()
{
    /* Declare a 2 x 5 multidimensional array */
    int x[2][5] = { {1, 2, 3, 4, 5},
                    {2, 4, 6, 8, 10} };

    int row, column;

    /* Display the rows */
    for (row=0; row<2; row++)
    {
        /* Display the columns */
        for (column=0; column<5; column++)
            printf("%d\t", x[row][column]);
    }
}

```

```
        putchar('\n');
    }
    return 0;}

```

CHARACTER ARRAYS:

A string is an list (or string) of characters stored contiguously with a marker to indicate the end of the string. Since the characters of a string are stored contiguously, we can easily implement a string by using an array of characters if we keep track of the number of elements stored in the array. However, common operations on strings include breaking them up into parts (called **substrings**), joining them together to create new strings, replacing parts of them with other strings, etc. There must be some way of detecting the size of a current valid string stored in an array of characters.

In C, a string of characters is stored in successive elements of a character array and terminated by the NULL character. For example, the string "Hello" is stored in a character array, `msg[]`, as follows:

```
char msg[SIZE];
    msg[0] = 'H';
    msg[1] = 'e';
    msg[2] = 'l';
    msg[3] = 'l';
    msg[4] = 'o';
    msg[5] = '\0';

```

The NULL character is written using the escape sequence ' `\0`'. The ASCII value of NULL is 0, and NULL is

defined as a macro to be 0 in `stdio.h`; so programs can use the symbol, `NULL`, in expressions if the header file is included. The remaining elements in the array after the `NULL` may have any garbage values. When the string is retrieved, it will be retrieved starting at index 0 and succeeding characters are obtained by incrementing the index until the first `NULL` character is reached signaling the end of the string. Figure 4.1 shows a string as it is stored in memory. Note, string constants, such as "Hello" are automatically terminated by `NULL` by the compiler.

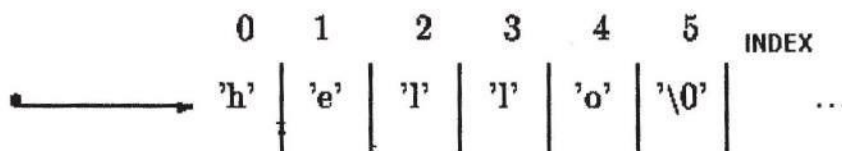


Figure 4.1: String stored in Memory as a Array

4.1 STRUCTURES

A structure is a collection of variables under a single name. These variables can be of different types, and each has a name which is used to select it from the structure. A structure is a convenient way of grouping several pieces of related information together.

A structure can be defined as a new named type, thus extending the number of available types. It can use other structures, arrays or pointers as some of its members, though this can get complicated unless you are careful.

DEFINING A STRUCTURE

A structure type is usually defined near to the start of a file using a `typedef` statement. `typedef` defines and names a new type, allowing its use throughout the program. `typedefs` usually occur just after the `#define` and `#include` statements in a file.

Here is an example structure definition.

```
typedef struct {  
    char name[64];  
    char course[128];  
    int age;  
    int year;  
} student;
```

This defines a new type student variables of type student can be declared as follows.

```
student st_rec;
```

Notice how similar this is to declaring an int or float.

The variable name is st_rec, it has members called name, course, age and year.

ACCESSING MEMBERS OF A STRUCTURE:

Each member of a structure can be used just like a normal variable, but its name will be a bit longer. To return to the examples above, member name of structure st_rec will behave just like a normal array of char, however we refer to it by the name

```
st_rec.name
```

Here the dot is an operator which selects a member from a structure.

Where we have a pointer to a structure we could dereference the pointer and then use dot as a member selector. This method is a little clumsy to type. Since selecting a member from a structure pointer happens frequently, it has its own operator -> which acts as follows. Assume that st_ptr is a pointer

to a structure of type student we would refer to the name member as

```
st_ptr -> name
```

STRUCTURES AS FUNCTION ARGUMENTS:

A structure can be passed as a function argument just like any other variable. This raises a few practical issues. Where we wish to modify the value of members of the structure, we must pass a pointer to that structure. This is just like passing a pointer to an int type argument whose value we wish to change.

If we are only interested in one member of a structure, it is probably simpler to just pass that member. This will make for a simpler function, which is easier to re-use. Of course if we wish to change the value of that member, we should pass a pointer to it.

When a structure is passed as an argument, each member of the structure is copied. This can prove expensive where structures are large or functions are called frequently. Passing and working with pointers to large structures may be more efficient in such cases.

FURTHER USES OF STRUCTURES:

As we have seen, a structure is a good way of storing related data together. It is also a good way of representing certain types of information. Complex numbers in mathematics inhabit a two dimensional plane (stretching in real and imaginary directions). These could easily be represented here by

```
typedef struct {  
    double real;  
    double imag;
```

```
} complex;
```

doubles have been used for each field because their range is greater than floats and because the majority of mathematical library functions deal with doubles by default.

In a similar way, structures could be used to hold the locations of points in multi-dimensional space. Mathematicians and engineers might see a storage efficient implementation for sparse arrays here. Apart from holding data, structures can be used as members of other structures. Arrays of structures are possible, and are a good way of storing lists of data with regular fields, such as databases.

Another possibility is a structure whose fields include pointers to its own type. These can be used to build chains (programmers call these linked lists), trees or other connected structures. These are rather daunting to the new programmer, so we won't deal with them here.

4.2 UNIONS

Unions are the same as structures, except that, where you would have written struct before, now you write union. Everything works the same way, but with one big exception. In a structure, the members are allocated separate consecutive chunks of storage. In a union, every member is allocated the same piece of storage. Sometimes you want a structure to contain different values of different types at different times but to conserve space as much as possible. Using a union, achieves this. Here's an example:

```
#include <stdio.h>

#include <stdlib.h>

main(){
```

```

union {
    float u_f;
    int u_i;
}var;

var.u_f = 23.5;
printf("value is %f\n", var.u_f);
var.u_i = 5;
printf("value is %d\n", var.u_i);
exit(EXIT_SUCCESS);
}

```

If the example had, say, put a float into the union and then extracted it as an int, a strange value would have resulted. The two types are almost certainly not only stored differently, but of different lengths. The int retrieved would probably be the low-order bits of the machine representation of a float, and might easily be made up of part of the mantissa of the float plus a piece of the exponent. The Standard says that if you do this, the behaviour is implementation defined (not undefined). The behaviour is defined by the Standard in one case: if some of the members of a union are structures with a 'common initial sequence' (the first members of each structure have compatible type and in the case of *bitfields* are the same length), and the union currently contains one of them, then the common initial part of each can be used interchangeably.

The most common way of remembering what is in a union is to embed it in a structure, with another member of the structure used to indicate the type of thing currently in the union. Here is how it might be used:

```
#include <stdio.h>

#include <stdlib.h>

/* code for types in union */
#define FLOAT_TYPE  1
#define CHAR_TYPE   2
#define INT_TYPE    3

struct var_type{
    int type_in_union;
    union{
        float  un_float;
        char   un_char;
        int    un_int;
    }vt_un;
}var_type;

void
print_vt(void){
```

```

switch(var_type.type_in_union){
    default:
        printf("Unknown type in union\n");
        break;
    case FLOAT_TYPE:
        printf("%f\n", var_type.vt_un.un_float);
        break;
    case CHAR_TYPE:
        printf("%c\n", var_type.vt_un.un_char);
        break;
    case INT_TYPE:
        printf("%d\n", var_type.vt_un.un_int);
        break;
}
}
main(){
    var_type.type_in_union = FLOAT_TYPE;
    var_type.vt_un.un_float = 3.5;

    print_vt();

    var_type.type_in_union = CHAR_TYPE;
    var_type.vt_un.un_char = 'a';
}

```

```

    print_vt();
    exit(EXIT_SUCCESS);
}

```

That also demonstrates how the dot notation is used to access structures or unions inside other structures or unions.

4.3 BIT FIELDS

They can only be declared inside a structure or a union, and allow you to specify some very small objects of a given number of bits in length. Their usefulness is limited and they aren't seen in many programs. This example should help to make things clear:

```

struct {
    /* field 4 bits wide */
    unsigned field1 :4;
    /*
     * unnamed 3 bit field
     * unnamed fields allow for padding
     */
    unsigned      :3;
    /*
     * one-bit field
     * can only be 0 or -1 in two's complement!
     */
    signed field2 :1;
    /* align next field on a storage unit */
    unsigned      :0;
}

```

```
    unsigned field3 :6;
}full_of_fields;
```

Each field is accessed and manipulated as if it were an ordinary member of a structure. The keywords signed and unsigned mean what you would expect, except that it is interesting to note that a 1-bit signed field on a two's complement machine can only take the values 0 or -1. The declarations are permitted to include the const and volatile qualifiers.

The main use of bitfields is either to allow tight packing of data or to be able to specify the fields within some externally produced data files. C gives no guarantee of the ordering of fields within machine words, so if you do use them for the latter reason, your program will not only be non-portable, it will be compiler-dependent too. The Standard says that fields are packed into 'storage units', which are typically machine words. The packing order, and whether or not a bitfield may cross a storage unit boundary, are implementation defined. To force alignment to a storage unit boundary, a zero width field is used before the one that you want to have aligned.

Bit fields can require a surprising amount of run-time code to manipulate these things and you can end up using more space than they save. Bit fields do not have addresses—you can't have pointers to them or arrays of them.

Let us sum up

We have covered about the concepts arrays and structures in C programs.

Learning Activities

a) Fill in the blanks:

1. An array is a collection of the _____ type of data elements.

b) State whether true or false:

2. Structure elements are accessed using the dot operator.
(True / False)

Answers to Learning Activities

a) Fill in the blanks:

3. Same.

a) State whether true or false:

1. True.

Model Questions

1. Explain in detail the concept of arrays
2. What are structures? How do they differ from arrays?
3. Write a sample program to show the passing of arrays to functions.

References

1. Gottfried, B.S., "Schaum's Outline of Theory and Problems of Programming in C", Tata McGraw Hill, Delhi, 1995.
2. Kernighan, B.W. and Ritchi, D.M., The C Programming Prentice Hall of India, Delhi, 1998.

Unit-5

STORAGE CLASSES AND FUNCTIONS

Structure

Overview

Learning objectives

5.0 Storage Classes in C

5.1 Functions

5.2 Implementation of Pointers in functions

5.3 Passing Arrays to Functions

5.4 Passing Functions to Functions

5.5 Passing Pointers to Functions

Let us sum up

Answer to learning activities

References

Overview

This unit is devoted to the detailed study of the various storage classes supported by C. It covers the major usage of function calls in C.

Learning Objectives

At the end of this unit you will have a clear knowledge on the various storage classes in C and the usage of function class to provide modular programming in C.

5.0 STORAGE CLASSES IN C

Storage classes refers to the permanence of a variable and its scope within the program. It is thus the portion of the program over which the variable is recognized.

Register - This is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word) and cant have the unary '&' operator applied to it (as it does not have a memory location).

```
{  
    register int Miles;  
}
```

Register should only be used for variables that require quick access - such as counters. It should also be noted that defining 'register' goes not mean that the variable will be stored in a register. It means that it MIGHT be stored in a register - depending on hardware and implementation restrictions.

Automatic – This variable is declared within a function. It is local to that function. The scope of the variable is confined to that function. Another variable of the same name can be defined in a different function as each of them are independent. The word *auto* could be added before the variable name but it is optional. When we exit from the function the value is lost.

External – This variable is not confined to any function. The scope extends from the point of definition through the remainder of the program. It is recognized globally. It can be accessed by any function. The value set in one function can be used in another. Syntax - *extern <variable name>*

Static – These variables are defined within individual functions. Their functionality is the same as automatic variables. The value assigned to them is retained till the life of the program is over. The syntax is given as, *static <variable name>*
<value> *static int a = 10;*

5.1 FUNCTIONS

Function is a self- contained program segment that carries out some specific, well-defined task. Every C program consists of one or more functions. `main()` is also a function. Program execution will always begin by carrying out the instructions in `main()`. A function

- Will carry out its intended action whenever it is accessed
- Will process information passed to it from the calling portion of the program and return a single value. Information will be passed to the function via special identifiers called arguments or parameters.

A function has three principal components.

- The first line – contains the type specification of the value returned by the function, followed by the function name, and optionally a set of arguments separated by commas and enclosed in parenthesis. Empty parenthesis pair must follow the function name if the definition does not include any argument.

*data-type name (formal arg1, formal arg2,
.....formal argn)*

where *data-type* represents the data type of the value that is returned. *Name* is the function name. The ***formal arguments*** allow information to be transferred from the

calling portion of the function. The corresponding arguments in the function reference are called **actual arguments** as they define the information actually being transferred.

→ Argument declarations – all formal arguments must be declared at this point in the function. Each formal argument must have the same data-type as its corresponding actual argument.

→ Action to be taken by the function – compound statements referred as the body of the function.

The return statement causes control to be returned to the point from which the function was accessed.

return expression;

The use of user-defined functions allow a large program to be broken down into a number of smaller, self-contained components, each of which has some unique, identifiable purpose.

→ Certain instructions could be accessed repeatedly from several different places within the program. The repeated instructions are placed within a single function, which can then be accessed whenever it is needed.

→ A different set of data can be transferred to the function each time it is accessed.

→ Program is easier to write and debug.

```
#include <stdio.h>
```

```
#define PI 3.14
```

```
void areaperi (float r);
```

```
main()
```

```

    { float r;
      clrscr();
      printf ("\nEnter radius of the circle : ");
      scanf ("%f", &r);
      areaperi (r);
      getch();
    }

void areaperi (float r)
{
float area, volume;

area = PI * r * r;

printf ("\nArea = %f", area);

volume = PI *4/3 * r * r * r;

printf ("\nVolume = %f", volume);

return;

```

5.2 IMPLEMENTATION OF POINTERS IN FUNCTIONS

Passing pointers to functions can be done in two ways,

CALL BY VALUE:

In call-by-value, the argument expression is evaluated, and the resulting value is bound to the corresponding variable in the function (usually by capture-avoiding substitution or by copying the value into a new memory region). If the function or procedure is able to assign values to its parameters, only the local copy is assigned -- that is, anything passed into a function call is unchanged in the caller's scope when the function returns.

```
#include <stdio.h>
```

```

void modify (int a);

main()
{
    int a;

        printf ("\nEnter value for a : ");

    scanf ("%d", &a);

    modify (a );

    printf ("\nValue of a in Main is : %d ", a);

    getch();

}

void modify (int a)
{
    a = a * 100;

    printf ("\nValue of a in function is : %d", a);

    return;

}

```

The value of a is that a was modified has to be returned to the main to view the altered value.

CALL BY REFERENCE:

In *call-by-reference* evaluation, a function is passed an implicit reference to its argument rather than the argument value itself. If the function is able to modify such a parameter, then any changes it makes will be visible to the caller as well. If the argument expression is an L-value, its address is used. Otherwise, a temporary object is constructed by the caller and a reference to this object is passed; the object is then discarded when the function returns.

```

#include <stdio.h>

void modify (int *a)
{
    *a = *a * 100;
}

main()
{
    int a;

    printf ("\nEnter value for a : ");

    scanf ("%d", &a);

    modify (&a );

    printf ("\nValue of a in Main is : %d ", a);

    getch();
}

void modify (int *a)
{
    *a = *a * 100;

    printf ("\nValue of a in function is : %d", *a);

    return;
}

```

5.3 PASSING ARRAYS TO FUNCTIONS

Let us illustrate the utility of passing arrays to functions by means of an example that reads an array and prints a list of scores using functional modules. The function, `read_intaray()`, reads scores and stores them, returning the number of scores read. The function, `print_intaray()`, prints the contents of the array. The refined algorithm for `main()` can be written as:

print title, etc.

```
n = read_intaray(exam_scores, MAX);
```

```
print_intaray(exam_scores, n);
```

Here we have passed an array, `exam_scores`, and a constant, `MAX` (specifying the maximum size of the proposed list), to `read_intarray()` and expect it to return the number of scores placed in the array. Similarly, when we print the array using `print_intarray`, we give it the array to be printed and a count of elements it contains. We know that in order for a *called* function to access objects in the *calling* function (such as to store elements in an array) we must use *indirect access*, i.e. pointers. So, `read_intarray()` must indirectly access the array, `exam_scores`, in `main()`. One unique feature of C is that array access is *always* indirect; thus making it particularly easy for a called function to indirectly access elements of an array and store or retrieve values.

A sample program for this is given by,

```
#include <stdio.h>

#define MAX 10

int read_intarray(int scores[], int lim);

void print_intarray (int scores[], int lim);

main()
{
    int n, exam_scores[MAX];

    printf ("*** LIST OF EXAM SCORES ***\n\n");

    n = read_intarray(exam_scores, MAX);

    print_intarray (exam_scores, n);
}
```



```

/* Function to read scores */
int read_intarray(int scores[], int lim)
{
    int n, count = 0;

    printf ("Type scores, EOF to quit\n");

    while ((count < lim) && (scanf("%d", &n) != EOF))    {
        scores [count] = n;
        count++;
    }

    return count;
}

```

```

/*Function to print scores */
void print_intarray (int scores[], int lim)
{
    int i;

    printf ("\n *** EXAM SCORES ***\n\n");

    for (i = 0; i < lim; i++)
        printf ("%d\n", scores[i]);
}

```

5.4 PASSING FUNCTIONS TO FUNCTIONS

In computer programs, functions form the powerful building blocks that allow developers to break code down into simple, more easily managed steps, as well as let programmers break programs into reusable parts. As a nod to the wonderful function, in this article I'll demonstrate how to craft new template-based functions at runtime and explain how to build

functions that can be configured at runtime using function parameters. A practical use of nameless functions is that of building functions to pass as arguments to other functions.

An example code is given below:

```
int square(int x)
{
    return x * x;
}
```

```
int main()
```

```
{
```

```
    int my_test_data[6] = {1, 2, 3, 4, 5, 6};
```

```
    int list_of_squares[6],
```

```
    list;
```

```
    /* Note that this would only work if C had a function
```

```
    * called map which worked with integer functions.
```

```
    * * doesn't, so this is largely theoretical. Also remember
```

```
    * that C doesn't know the size of the arrays, so you have
```

```
to
```

```
    * pass it explicitly. Also, list_of_squares has to be passed
```

```
    * as a parameter in C.
```

```
    */
```

```
    map(square, my_test_data, list_of_squares, 6);
```

```
    printf("");
```

```

for(i=0; i<6; i++)
{
    printf("%d ", list_of_squares[i]);
}
printf("\n");
}

```

5.5 PASSING POINTERS TO FUNCTIONS

A useful technique is the ability to have pointers to functions. Their declaration is easy: write the declaration as it would be for the function, say

```
int func(int a, float b);
```

and simply put brackets around the name and a * in front of it that declares the pointer. Because of precedence, if you don't parenthesize the name, you declare a function returning a pointer:

```
/* function returning pointer to int */
```

```
int *func(int a, float b);
```

```
/* pointer to function returning int */
```

```
int (*func)(int a, float b);
```

Once you've got the pointer, you can assign the address of the right sort of function just by using its name like an array. A function name is turned into an address when it's used in an expression. You can call the function using one of two forms:

```
(*func)(1,2);
```

```
/* or */
```

```
func(1,2);
```

Here's a simple example.

```
#include <stdio.h>
#include <stdlib.h>
void func(int);
main(){
    void (*fp)(int);
    fp = func;
    (*fp)(1);
    fp(2);
    exit(EXIT_SUCCESS);
}
void func(int arg){
    printf("%d\n", arg);
}
```

If you like writing finite state machines, you might like to know that you can have an array of pointers to functions, with declaration and use like this:

```
void (*fparr[])(int, float) = {
    /* initializers */
};
/* then call one */
fparr[5](1, 3.4);
```

Let us sum up

We have covered about the basic data types in C. We have also studied in detail the various input and output features in C.

Learning Activities

a) Fill in the blanks:

- 1) The automatic storage class is similar to that of _____.
- 2) The two types of parameters are _____ and _____.

b) State whether true or false:

1. In C array index by default starts with 0. (True / False).
2. A function call has four principal components. (True/False).

Answers to Learning Activities

a) Fill in the blanks:

- 1) Static class.
- 2) Formal, actual.

b) State whether true or false:

1. True.
2. False.

Model Questions

1. Explain about the various storage classes in C
2. Write detailed notes on call by value and call by reference.

References

1. Gottfried, B.S., "Schaum's Outline of Theory and Problems of Programming in C". Tata McGraw Hill, Delhi, 1995.
2. Kerninghan, B.W. and Ritchi, D.M., The C Programming Prentice Hall of India, Delhi, 1998.

Unit-6

FILE HANDLING IN C

Structure

Overview

Learning objectives

6.0 Text Files

6.1 Binary Files

Let us sum up

Answer to learning activities

References

Overview

This unit is devoted to the detailed study of the various File Handling features available in C.

Learning Objectives

At the end of this unit you will have a clear knowledge on the ways of representing file structures in C program.

6.0 TEXT FILES

Text files in C are straightforward and easy to understand. All text file functions and types in C come from the **stdio** library.

When you need text I/O in a C program, and you need only one source for input information and one sink for output information, you can rely on **stdin** (standard in) and **stdout** (standard out). You can then use input and output redirection at the command line to move different information streams through the program. There are six different I/O commands in <stdio.h> that you can use with stdin and stdout:

- **printf** - prints formatted output to stdout
- **scanf** - reads formatted input from stdin
- **puts** - prints a string to stdout
- **gets** - reads a string from stdin
- **putc** - prints a character to stdout
- **getc, getchar** - reads a character from stdin

The advantage of stdin and stdout is that they are easy to use. Likewise, the ability to redirect I/O is very powerful. For example, maybe you want to create a program that reads from stdin and counts the number of characters:

```
#include <stdio.h>

#include <string.h>

void main()

{

    char s[1000];

    int count=0;

    while (gets(s))

        count += strlen(s);

    printf("%d\n",count);

}
```

Enter this code and run it. It waits for input from stdin, so type a few lines. When you are done, press CTRL-D to signal end-of-file (eof). The `gets` function reads a line until it detects eof, then returns a 0 so that the while loop ends. When you press CTRL-D, you see a count of the number of characters in

stdin (the screen) (Use `man gets` or your compiler's documentation to learn more about the `gets` function.)

Now, suppose you want to count the characters in a file. If you've compiled the program to an executable named `xxx`, you can type the following:

```
xxx < filename
```

Instead of accepting input from the keyboard, the contents of the file named `filename` will be used instead. You can achieve the same result using pipes:

```
cat < filename | xxx
```

You can also redirect the output to a file:

```
xxx < filename > out
```

This command places the character count produced by the program in a text file named `out`.

Sometimes, you need to use a text file directly. For example, you might need to open a specific file and read from or write to it. You might want to manage several streams of input or output or create a program like a text editor that can save and recall data or configuration files on command. In that case, use the text file functions in `stdio`:

- `fopen` - opens a text file
- `fclose` - closes a text file
- `feof` - detects end-of-file marker in a file
- `fprintf` - prints formatted output to a file
- `fscanf` - reads formatted input from a file
- `fputs` - prints a string to a file
- **`fgets`** - reads a string from a file

- **fputc** - prints a character to a file
- **fgetc** - reads a character from a file

TEXT FILES: OPENING

You use **fopen** to open a file. It opens a file for a specified mode (the three most common are r, w, and a, for read, write, and append). It then returns a file pointer that you use to access the file. For example, suppose you want to open a file and write the numbers 1 to 10 in it. You could use the following code:

```
#include <stdio.h>

#define MAX 10

int main()
{
    FILE *f;
    int x;
    f=fopen("out","w");
    if (!f)
        return 1;
    for(x=1; x<=MAX; x++)
        fprintf(f,"%d\n",x);
    fclose(f);
    return 0;
}
```

The **fopen** statement here opens a file named **out** with the w mode. This is a destructive write mode, which means that if **out** does not exist it is created, but if it does exist it is destroyed and

a new file is created in its place. The `fopen` command returns a pointer to the file, which is stored in the variable `f`. This variable is used to refer to the file. If the file cannot be opened for some reason, `f` will contain `NULL`.

MAIN FUNCTION RETURN VALUES:

This program is the first program in this series that returns an error value from the main program. If the `fopen` command fails, `f` will contain a `NULL` value (a zero). We test for that error with the `if` statement. The `if` statement looks at the `True/False` value of the variable `f`. Remember that in C, 0 is `False` and anything else is `true`. So if there were an error opening the file, `f` would contain zero, which is `False`. The `!` is the NOT operator. It inverts a Boolean value. So the `if` statement could have been written like this:

```
if (f == 0)
```

That is equivalent. However, `if (!f)` is more common.

If there is a file error, we return a 1 from the main function. In UNIX, you can actually test for this value on the command line. See the shell documentation for details.

The `fprintf` statement should look very familiar: It is just like `printf` but uses the file pointer as its first parameter. The `fclose` statement closes the file when you are done.

TEXT FILES: READING

To read a file, open it with `r` mode. In general, it is not a good idea to use `fscanf` for reading: Unless the file is perfectly formatted, `fscanf` will not handle it correctly. Instead, use `fgets` to read in each line and then parse out the pieces you need.

The following code demonstrates the process of reading a file and dumping its contents to the screen:

```

#include <stdio.h>

int main()
{
    FILE *f;
    char s[1000];

    f=fopen("infile","r");
    if (!f)
        return 1;
    while (fgets(s,1000,f)!=NULL)
        printf("%s",s);
    fclose(f);
    return 0;
}

```

The `fgets` statement returns a `NULL` value at the end-of-file marker. It reads a line (up to 1,000 characters in this case) and then prints it to `stdout`. Notice that the `printf` statement does not include `\n` in the format string, because `fgets` adds `\n` to the end of each line it reads. Thus, you can tell if a line is not complete in the event that it overflows the maximum line length specified in the second parameter to `fgets`.

C Errors to Avoid - Do not accidentally type `close` instead of `fclose`. The `close` function exists, so the compiler accepts it. It will even appear to work if the program only opens or closes a few files. However, if the program opens and closes a file in a loop, it will eventually run out of available file handles and/or

memory space and crash, because **close** is not closing the files correctly.

6.1 BINARY FILES

Binary files are very similar to arrays of structures, except the structures are in a disk file rather than in an array in memory. Because the structures in a binary file are on disk, you can create very large collections of them (limited only by your available disk space). They are also permanent and always available. The only disadvantage is the slowness that comes from disk access time.

Binary files have two features that distinguish them from text files:

- » You can jump instantly to any structure in the file, which provides random access as in an array.
- » You can change the contents of a structure anywhere in the file at any time.

Binary files also usually have faster read and write times than text files, because a binary image of the record is stored directly from memory to disk (or vice versa). In a text file, everything has to be converted back and forth to text, and this takes time.

C supports the file-of-structures concept very cleanly. Once you open the file you can read a structure, write a structure, or seek to any structure in the file. This file concept supports the concept of a **file pointer**. When the file is opened, the pointer points to record 0 (the first record in the file). Any **read operation** reads the currently pointed-to structure and moves the pointer down one structure. Any **write operation** writes to

the currently pointed-to structure and moves the pointer down one structure. **Seek** moves the pointer to the requested record.

Keep in mind that C thinks of everything in the disk file as blocks of bytes read from disk into memory or read from memory onto disk. C uses a file pointer, but it can point to any byte location in the file. You therefore have to keep track of things.

The following program illustrates these concepts:

```
#include <stdio.h>

/* random record description - could be anything */
struct rec
{
    int x,y,z;
};

/* writes and then reads 10 arbitrary records
   from the file "junk". */
int main()
{
    int i,j;
    FILE *f;
    struct rec r;

    /* create the file of 10 records */
    f=fopen("junk","w");
    if (!f)
```

```

    return 1;
for (i=1;i<=10; i++)
{
    r.x=i;
    fwrite(&r,sizeof(struct rec),1,f);
}
fclose(f);

```

```

/* read the 10 records */
f=fopen("junk","r");
if (!f)
    return 1;
for (i=1;i<=10; i++)
{
    fread(&r,sizeof(struct rec),1,f);
    printf("%d\n",r.x);
}
fclose(f);
printf("\n");

```

```

/* use fseek to read the 10 records
in reverse order */
f=fopen("junk","r");
if (!f)

```

```

    return 1;
for (i=9; i>=0; i--)
{
    fseek(f,sizeof(struct rec)*i,SEEK_SET);
    fread(&r,sizeof(struct rec),1,i);
    printf("%d\n",r.x);
}
fclose(f);
printf("\n");

/* use fseek to read every other record */
f=fopen("junk","r");
if (!f)
    return 1;
fseek(f,0,SEEK_SET);
for (i=0;i<5; i++)
{
    fread(&r,sizeof(struct rec),1,f);
    printf("%d\n",r.x);
    fseek(f,sizeof(struct rec),SEEK_CUR);
}
fclose(f);
printf("\n");

```

```

/* use fseek to read 4th record,
   change it, and write it back */
f=fopen("junk","r");
if (!f)
    return 1;
fseek(f,sizeof(struct rec)*3,SEEK_SET);
fread(&r,sizeof(struct rec),1,f);
r.x=100;
fseek(f,sizeof(struct rec)*3,SEEK_SET);
fwrite(&r,sizeof(struct rec),1,f);
fclose(f);
printf("\n");

/* read the 10 records to insure
   4th record was changed */
f=fopen("junk","r");
if (!f)
    return 1;
for (i=1;i<=10;i++)
{
    fread(&r,sizeof(struct rec),1,f);
    printf("%d\n",r.x);
}
fclose(f);

```



```
    return 0;
}
```

In this program, a structure description `rec` has been used, but you can use any structure description you want. You can see that `fopen` and `fclose` work exactly as they did for text files.

The new functions here are `fread`, `fwrite` and `fseek`. The `fread` function takes four parameters:

- A memory address
- The number of bytes to read per block
- The number of blocks to read
- The file variable

Thus, the line `fread(&r,sizeof(struct rec),1,f);` says to read 12 bytes (the size of `rec`) from the file `f` (from the current location of the file pointer) into memory address `&r`. One block of 12 bytes is requested. It would be just as easy to read 100 blocks from disk into an array in memory by changing 1 to 100.

The `fwrite` function works the same way, but moves the block of bytes from memory to the file. The `fseek` function moves the file pointer to a byte in the file. Generally, you move the pointer in `sizeof(struct rec)` increments to keep the pointer aligned to a record. You can use three options when you call

- `SEEK_SET`
- `SEEK_CUR`
- `SEEK_END`

`SEEK_SET` moves the pointer `x` bytes down from the beginning of the file (from byte 0 in the file). `SEEK_CUR` moves

the pointer `x` bytes down from the current pointer position. **SEEK_END** moves the pointer from the end of the file (so you must use negative offsets with this option).

Several different options appear in the code above. In particular, note the section where the file is opened with `r+` mode. This opens the file for reading and writing, which allows records to be changed. The code seeks to a record, reads it, and changes a field; it then seeks back because the read displaced the pointer, and writes the change back.

Let us sum up

We have covered about the File Handling features in C in detail.

Learning Activities

a) Fill in the blanks:

- 1) The two types of file handled by C are _____ and _____.
- 2) The `fseek`, `fread` and `fwrite` functions take _____ parameters.

b) State whether true or false:

1. File descriptor is a pointer data type. (True / False)
2. Structures can be passed as records to files. (True/False).

Answers to Learning Activities

a) Fill in the blanks:

- 1) text, binary
- 2) Four.

b) State whether true or false:

1. True.
2. True.

Model Questions

1. What are Files? List the different types of files associated with C.
2. Write a detailed note on Text files
3. Describe in detail Binary Files. available in C.

References

1. Kamthane, "Programming with ANSI and Turbo C, Pearson Education, Delhi, 2002.
2. Al Kelley, Iya Pohl, "A Book on C", Pearson Education, Delhi, 2001.

Block 3: Introduction

This block will teach you about Data Structures mainly about the various data structure types supported by the C language. This block is divided into four units.

Unit 7: This unit deals with the stack and queue data structures and has been discussed in detail.

Unit 8 : This unit deals with the linked list data structure and its various types.

Unit 9 : This unit deals a detailed study on Graphs.

Unit-7

Stacks and Queues

Structure

Overview

Learning Objectives

7.0 Introduction to Data Structures

7.1 Stacks

7.2 Implementation of stacks using Arrays in C

7.3 Queues

7.4 Implementation of stacks using Arrays in C

7.5 Application of Queues

Let us sum up

Answer to learning activities

References

Overview

This unit is devoted to stack and queue data structures and has been discussed in detail.

Learning Objectives

At the end of this unit you will have a clear understanding on the basic concepts of data structures and its implementation as a stack and a queue.

7.0 INTRODUCTION TO DATA STRUCTURES

A data *type* is a well-defined collection of data with a well-defined set of operations on it. A data *structure* is an actual implementation of a particular abstract data type. In computer

science, a data structure is a way of storing data in a computer so that it can be used efficiently. Often a carefully chosen data structure will allow a more efficient algorithm to be used. The choice of the data structure often begins from the choice of an abstract data structure. A well-designed data structure allows a variety of critical operations to be performed, using as few resources, both execution time and memory space, as possible. Data structures are implemented using the data types, references and operations on them provided by a programming language.

Different kinds of data structures are suited to different kinds of applications, and some are highly specialized to certain tasks. For example, B-trees are particularly well-suited for implementation of databases, while routing tables rely on networks of machines to function. In the design of many types of programs, the choice of data structures is a primary design consideration, as experience in building large systems has shown that the difficulty of implementation and the quality and performance of the final result depends heavily on choosing the best data structure. After the data structures are chosen, the algorithms to be used often become relatively obvious. Sometimes things work in the opposite direction - data structures are chosen because certain key tasks have algorithms that work best with particular data structures. In either case, the choice of appropriate data structures is crucial.

The fundamental building blocks of most data structures are arrays, records, discriminated unions, and references. For example, the nullable reference, a reference which can be null, is a combination of references and discriminated unions, and the simplest linked data structure, the linked list, is built from records and nullable references.

There is some debate about whether data structures represent implementations or interfaces. How they are seen may be a matter of perspective. A data structure can be viewed as an interface between two functions or as an implementation of methods to access storage that is organized according to the associated data type.

7.1 STACKS

A stack is a list in which all insertions and deletions are made at one end, called the top. The last element to be inserted into the stack will be the first to be removed. Thus stacks are sometimes referred to as *Last In First Out* (LIFO) lists.

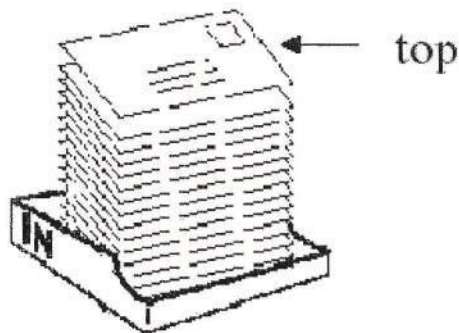


Figure 7.1: A Stack structure

The following operations can be applied to a stack:

InitStack(Stack): creates an empty stack

Push (Item): pushes an item on the stack

Pop(Stack): removes the first item from the stack

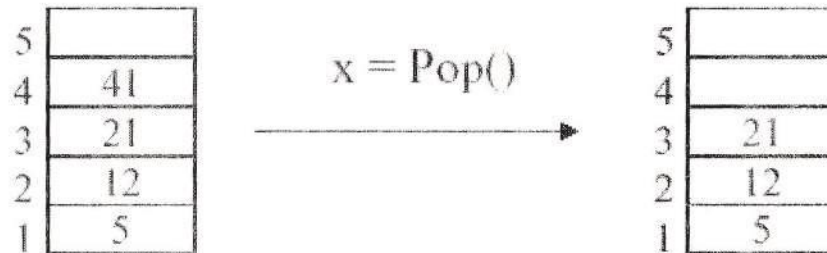
Top(Stack): returns the first item from the stack without removing it

isEmpty(Stack): returns true if the stack is empty

PUSH



POP



7.2 IMPLEMENTATION USING ARRAYS

We have here the implementation of a Stack data structure using C.

```
int StackArray[50]; // StackArray can contain
// up to 50 numbers
int top=-1; // index of the top element of the stack
// -1 used to indicate an empty stack
```



```

void Push(int elem)
{
top++;
StackArray[top] = elem;
}

int Pop()
{
int elem = StackArray[top];
top--;
return elem;
}

```

Abstract Data Type

An Abstract Data Type is some sort of data together with a set of functions (interface) that operate on the data. Access is only allowed through that interface. The implementation details are 'hidden' from the user.

The Stack-ADT can be given by

```

#define STACKSIZE 50

struct Stack
{
int item[STACKSIZE];
int top;
};

void InitStack(Stack &st);
void Push(Stack &st, int elem);

```

```
int Pop (Stack &st);  
int Top (Stack st);  
bool isEmpty(Stack st);
```

Stack specification Using the Stack ADT

```
#include "stack.h"  
  
void main()  
{  
  
Stack st1, st2; // declare 2 stack variables  
InitStack(st1); // initialise them  
InitStack(st2);  
  
Push(st1, 13); // push 13 onto the first stack  
Push(st2, 32); // push 32 onto the second stack  
  
int i = Pop(st2); // now popping st2 into i  
int j = Top(st1); // returns the top of st1 to j  
// without removing element  
};
```

Application of Stacks:

Evaluation of arithmetic expressions:

Usually, arithmetic expressions are written in *infix* notation, e.g.

$$A+B*C$$

An expression can as well be written in *postfix* notation (also called *reverse polish notation*):

A+B becomes AB+

A*C becomes AC*

A+B*C becomes ABC*+

(A+B)*C becomes AB+C*

Evaluating expressions

Given an expression in postfix notation. Using a stack they can be evaluated as follows:

- Scan the expression from left to right
- When a value (operand) is encountered, push it on the stack
- When an operator is encountered, the first and second element from the stack are popped and the operator is applied
- The result is pushed on the stack

2 14 + 5 *

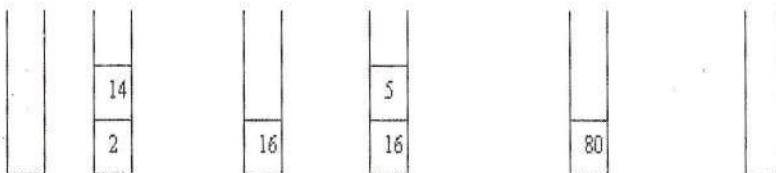
push 2
push 14

pop 14
pop 2
push 2 + 14 = 16

push 5

pop 5
pop 16
push 16 * 5 = 80

pop answer: 80



7.3 QUEUES

A Queue is an ordered collection of items from which items may be deleted at one end (called the *front* of the queue) and into which items may be inserted at the other end (the *rear* of the queue).

The following operations can be applied to a queue:

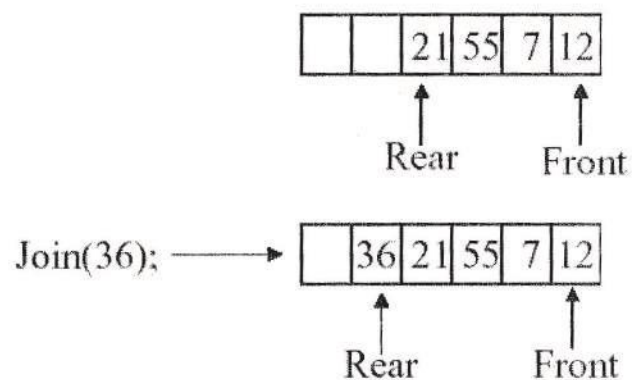
InitQueue(Queue): creates an empty queue

Join (Item): inserts an item to the rear of the queue

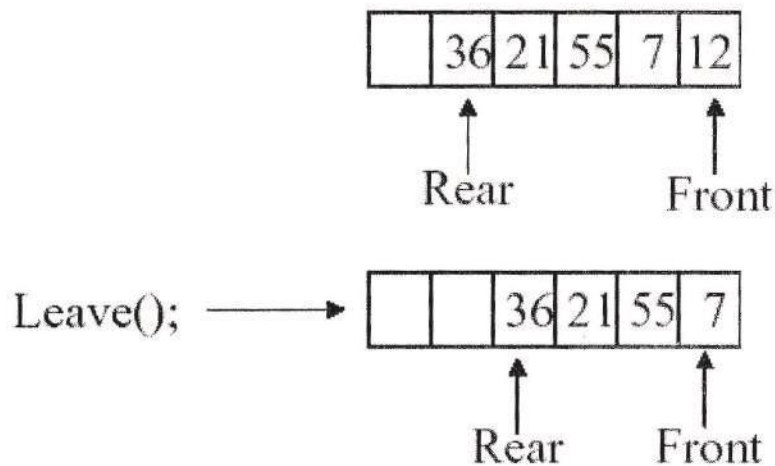
Leave(Queue): removes an item from the front of the queue

isEmpty(Queue): returns true is the queue is empty

FIFO-lists

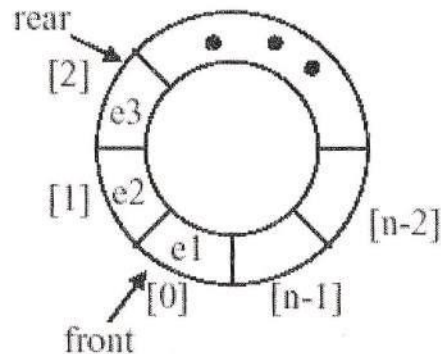


Elements can only be added to the rear of the queue and removed from the front of the queue.



7.4 IMPLEMENTATION OF QUEUES USING ARRAYS IN C

Removing an element from the queue is an expensive operation because all remaining elements have to be moved by one position. A more efficient implementation is obtained if we consider the array as being 'circular':



Joining the Queue

Initially, the queue is empty, i.e. $\text{front} == \text{rear}$. If we add an element to

the queue we

- 1) check if the queue is not full
- 2) store the element at the position indicated by rear
- 3) increase rear by one, wrap around if necessary (in this case rear always points to the last item in the queue -- the rear item)

Adding one element

```
if (rear == QSIZE - 1)
```

```
    rear = 0;
```

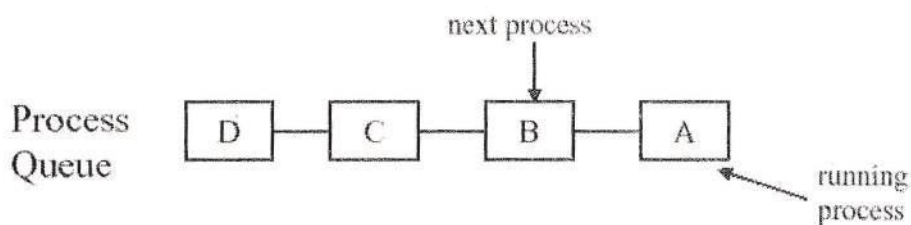
```
else
```

rear = rear+1;

add an element to the queue

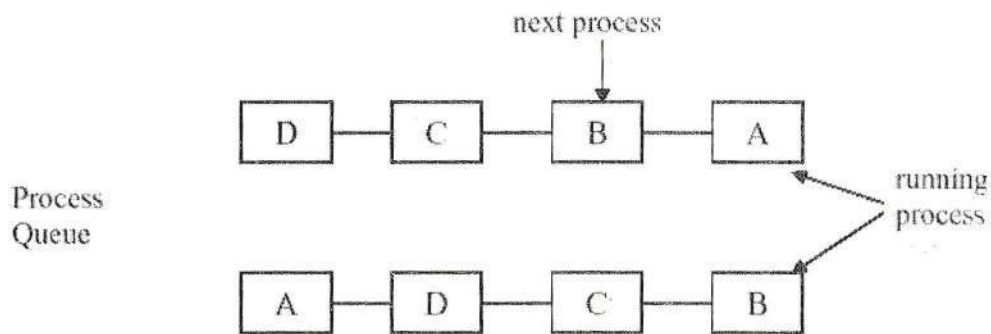
7.5APPLICATION OF QUEUES

In a multitasking operating system, the CPU time is shared between multiple processes. At a given time, only one process is running, all the others are 'sleeping'. The CPU time is administered by the scheduler. The scheduler keeps all current processes in a queue with the active process at the front of the queue.



Round-Robin Scheduling:

Every process is granted a specific amount of CPU time, its 'quantum'. If the process is still running after its quantum run out, it is suspended and put towards the end of the queue.



The Queue-ADT:

```
#define QSIZE 50  
  
struct Queue
```

```

{
int items[QSIZE];
int rear;
int front;
};
queue.h
void InitQueue(Queue &q);
void Join(Queue &q, int elem);
int Leave(Queue &q);
bool isEmpty(Queue q);

The Queue-ADT
#define QSIZE 50
struct Queue
{
int items[QSIZE];
int rear;
int front;
};
queue.h
void InitQueue(Queue &q);
void Join(Queue &q, int elem);
int Leave(Queue &q);
bool isEmpty(Queue q);

```

Let us sum up

We have covered about the basic data structures namely stacks and queues and shown their implementation using arrays in C.

Learning Activities

a) Fill in the blanks:

- 1) The fundamental building blocks of most data structures are _____ and _____.
- 2) An application of queues is in _____.

b) State whether true or false:

1. Stacks are used as a FIFO list. (True / False)
2. Queues are used to evaluate arithmetic expressions. (True/False).

Answers to Learning Activities

a) Fill in the blanks:

- 1) arrays, records
- 2) Job scheduling.

b) State whether true or false:

1. False.
2. False.

Model Questions

1. Explain in detail about stack data structure.
2. Write notes on the applications of queues.
3. Show the stack implementation of the expression $3 * 4 + 5$.

References

1. An Introduction to Data Structures with Applications by Tremblay, J.P. and Sorenson, P.G.

2. Data Structures using C and C++ by Y.Langesam, M.J. Augenstein and A.M. Tenenbaum.

Unit-8

LINKED LIST

Structure

Overview

Learning objectives

8.0 Linked List

8.1 Singly Linked List

8.2 Two-Way Linked List

8.3 Circular Linked List

8.4 Implementation of Linked Lists Structure in C

Let us sum up

Answer to learning activities

References

OVERVIEW

This unit is devoted to the detailed study of the Linked List Data Structure. It covers the three major types of linked lists.

Learning Objectives

At the end of this unit you will have a clear knowledge on the various types of linked lists and its implementation.

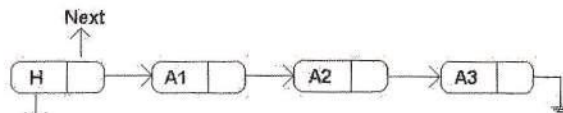
8.0 LINKED LISTS

A linked list is an algorithm for storing a list of items. It is made of any number of pieces of memory (nodes) and each node contains whatever data you are storing along with a pointer (a link) to another node. By locating the node referenced by that pointer and then doing the same with the pointer in that new node and so on, you can traverse the entire list.

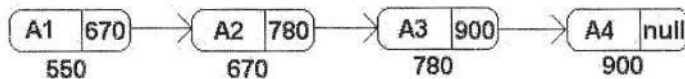
Because a linked list stores a list of items, it has some similarities to an array. But the two are implemented quite differently. An array is a single piece of memory while a linked list contains as many pieces of memory as there are items in the list. Obviously, if your links get messed up, you not only lose part of the list, but you will lose any reference to those items no longer included in the list (unless you store another pointer to those items somewhere).

Some advantages that a linked list has over an array are that you can quickly insert and delete items in a linked list. Inserting and deleting items in an array requires you to either make room for new items or fill the "hole" left by deleting an item. With a linked list, you simply rearrange those pointers that are affected by the change. Linked lists also allow you to have different-sized nodes in the list. Some disadvantages to linked lists include that they are quite difficult to sort. Also, you cannot immediately locate, say, the hundredth element in a linked list the way you can in an array. Instead, you must traverse the list until you've found the hundredth element.

General Representation

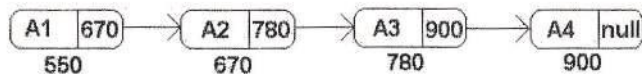


In the above figure there are 4 nodes. The data of the nodes are **A1** , **A2** , **A3** & **A4**.The next part of **A1** points the address of **A2** .The next part of **A2** points the address of **A3** .The next part of **A3** points the address of **A4** .The next part of **A4** contains **null** value. That means it contains no address.



In the above figure, the address specifies where the nodes are stored in memory. The node A1 is in the address 550.The next part of A1 contains the 670 which is the address of next node A2 .The node A2 is in the address 670.The next part of A2 contains the 780 which is the address of next node A3 .The node A3 is in the address 780.The next part of A3 contains the 900 which is the address of next node A4 .The next part of A4 contains null. Because A4 is the last node of the linked list. Using this technique the nodes are virtually connected.

8.1 SINGLY LINKED LIST



Singly linked list is the collection of nodes. Each node contains two parts. One is data field another one is address of the next node. Using this address field we can find the next node. So there is no limitation for storing the data. The nodes in the list are stored anywhere of the memory. The last node of

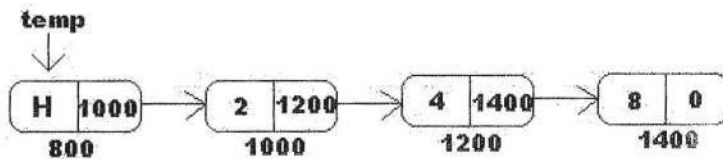
the list contains null value. In the program the value zero is used to represent the null value. The head node is used to represent where the list is start. The head node contains no data. There are five possible operations in a link list.

INSERTION

There are two possible ways to insert the new node in the list.

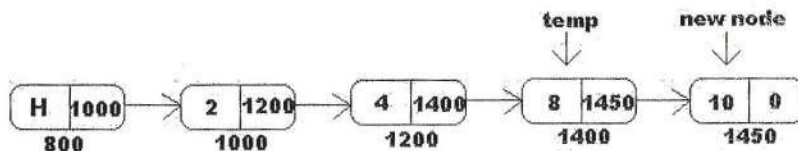
- INSERT THE NODE AS LAST NODE
- INSERT THE NODE BETWEEN TWO EXISTING NODES

Consider the following list.



Initially set the pointer **temp** in the head node. We want to insert the **new node** as the last node of the list. The position is **4** (excluding **Head** node.) Always, before perform the operation insert or delete move the **temp** to the previous position of insertion or deletion position.

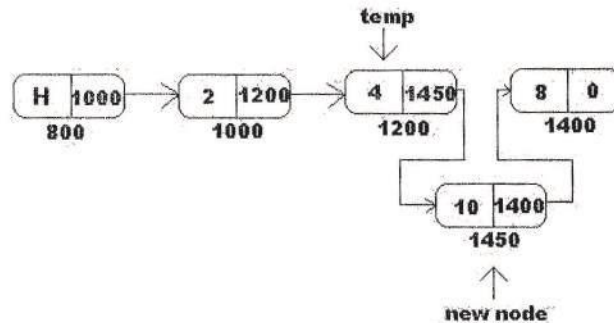
The **new node** is in the memory address **1450**.The **data** stored in the **new node** is **10**. After insert the **new node** the list has changed as follow as.



How can we link the **new node** with list?

1. Move the pointer **temp** to the previous position (3).
2. Store the address of **new node** (1450) in the address field of previous node **temp**.
3. Store **null** value on the address field of **new node** .

Now , the **new node** is connected with list.



Another one possible insertion in the list is, insert the **new node** between two existing nodes. This is same as the above example. We want to insert the **new node** at position **3**.The position is between two nodes. Like the above, move the pointer **temp** to the previous position of insertion position. So, we should move **temp** to position **2**. After insert the list has changed as follow as. Remember, the nodes in the list are anywhere in the memory. Using pointers we can link the nodes.

The steps involved in the insertion between two nodes are..

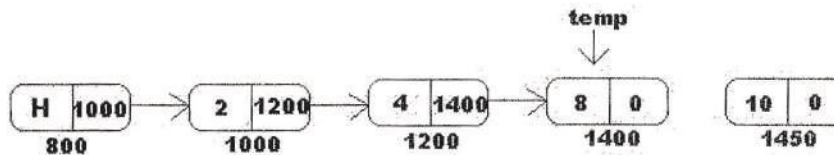
1. Move the pointer **temp** to the previous position (2).
2. Store the address of the next node (in position 4 address (1400)) of **new node** in the address field of **new node**.
3. Store the address of **new node** (1450) in the address field of previous node **temp**.

DELETION:

Like, Insertion there two possible deletions in the list.

- DELETE THE LAST NODE

- DELETE THE NODE BETWEEN TWO EXISTING

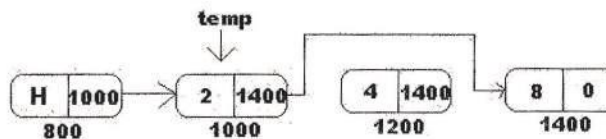


- NODES

In the first case, like insertion move the pointer **temp** to the previous node of deletion node. In the above figure, there are four nodes except head node. The last node is in the position **4**. To remove that node we can do the following steps.

1. Move the pointer **temp** to the previous position of last node.
2. Store **null** in the address field of **temp**.

Now, the last node has removed from the list. Another one possible deletion is, remove the node that is placed between two nodes. This is same as the above example. We want to delete the node at position **2**. The position is between two nodes. Like the above, move the pointer **temp** to the previous position of deletion position. So, we should move **temp** to position **1**. After delete, the list is changed as follow as.



The steps involved in the deletion operation are..

1. Move the pointer **temp** to the previous position (1).

2. Store the address of the node (1400) which is placed next of the deletion node in the address field of **temp** node. Now, the node has been removed.

The other operations on linked lists include view search the element and search the position of an element.

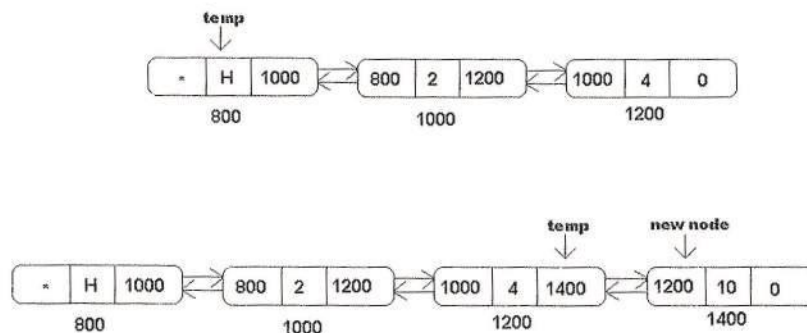
8.2 TWO WAY LINKED LIST

In the Two-way or doubly link list there are three parts in each node. There are prev, next, data. The prev contains the address of the previous node. The next contains the address of the next node. The data contains the value. The head node is used to represent where the list starts. The head node contains no data. The last node of the list contains null value.

There are two possible ways to insert the new node in the list.

- INSERT THE NODE AS LAST NODE
- INSERT THE NODE BETWEEN TWO EXISTING NODES

Consider the following list.

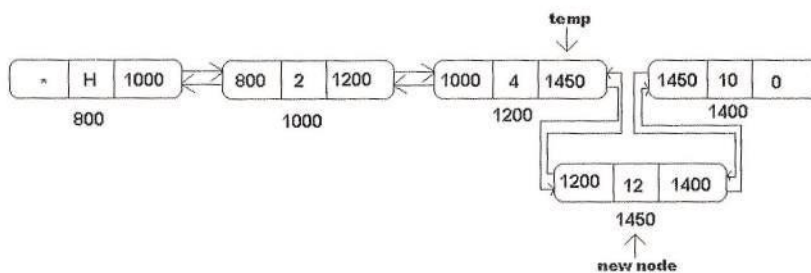


Initially set the pointer **temp** in the head node. We want to insert the **new node** as the last node of the list. The position is **3** (excluding **Head** node.) Always, before perform the operation insert or delete move the **temp** to the previous

position of insertion or deletion position. The **new node** is in the memory address **1400**. The **data** stored in the **new node** is **10**. After insert the **new node** the list has changed as follow as. How can we link the **new node** with list?

1. Move the pointer **temp** to the previous position (2).
2. Store the address of **new node** (1400) in the address field of previous node **temp**.
3. Store **null** value on the address field of **new node** .

Now, the **new node** is connected with list. Another one possible insertion in the list is, insert the **new node** between two existing nodes. This is same as the above example. We want to insert the **new node** at position **3**. The position is between two nodes. Like the above, move the pointer **temp** to the previous position of insertion position. So, we should move **temp** to position **2**. After insert the list has changed as follow as.



Remember, the nodes in the list are anywhere in the memory. Using pointers we can link the nodes. The steps involved in the insertion between two nodes are.

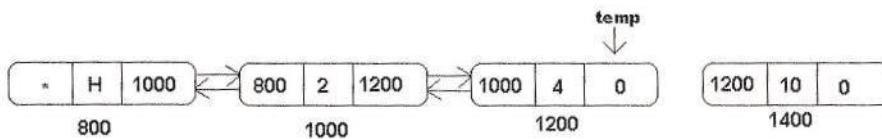
1. Move the pointer **temp** to the previous position (2).
2. Store the address of the next node (in position 4 address (1400)) of **new node** in the address field of **new node**.

3. Store the address of **new node** (1450) in the address field of previous node **temp**.

Like, Insertion there two possible deletions in the list.

- DELETE THE LAST NODE
- DELETE THE NODE BETWEEN TWO EXISTING NODES

In the first case, like insertion move the pointer **temp** to the previous node of deletion node.



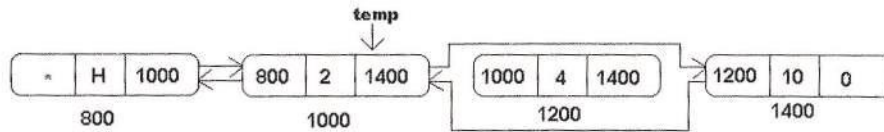
In the above figure, there are four nodes except head node. The last node is in the position **3**. To remove that node we can do the following steps.

1. Move the pointer **temp** to the previous position of last node.
2. Store **null** in the address field of **temp**.

Now, the last node has removed from the list. Another one possible deletion is, remove the node that is placed between two nodes. This is same as the above example. We want to delete the node at position **2**. The position is between two nodes. Like the above, move the pointer **temp** to the previous position of deletion position. So, we should move **temp** to position **1**. After delete, the list is changed as follows.

The steps involved in the deletion operation are..

1. Move the pointer **temp** to the previous position (1).



2. Store the address of the node (1400) which is placed next of the deletion node in the address field of **temp** node.

Now, the node has been removed. To show all the elements in the list, we should do the following steps.

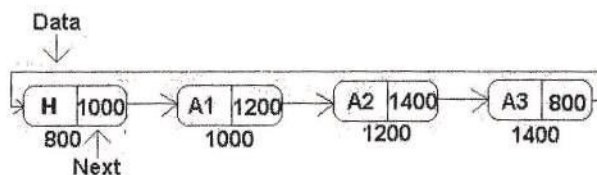
1. Move the pointer **temp** to the first node of the list.
2. Everytime after display the element move the pointer **temp** to the next position.
3. Repeat the process until the position is not **null**.

Search the position means, find the element which is stored in the particular position.

Initially the pointer **temp** points the first node. Move the pointer **temp** and check the position.

If the position is presented then display the element which is stored in that position. Search the no that is presented in the list are not. Initially the pointer **temp** points the first node. Move the pointer **temp** and check the number. If the number is presented then display the element and position.

8.3 CIRCULAR LINKED LIST

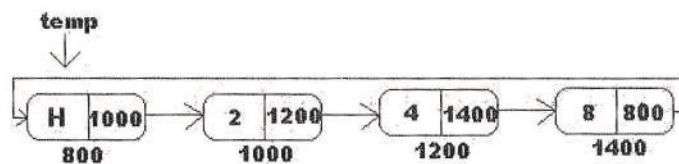


In the circular linked list is the collection of nodes. Each node contains two parts. One is data field another one is address of the next node. Using this address field we can find the next node. So there is no limitation for storing the data. The nodes in the list are stored anywhere of the memory. The last node of the list contains the starting address of the list. So the list is called as circular list. In the program the value zero is used to represent the null value. The head node is used to represent where the list is start. The head node contains no data.

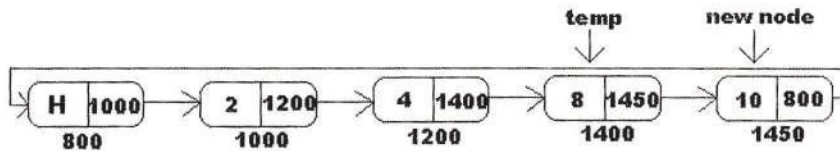
There are two possible ways to insert the new node in the list.

- INSERT THE NODE AS LAST NODE
- INSERT THE NODE BETWEEN TWO EXISTING NODES

Consider the following list.



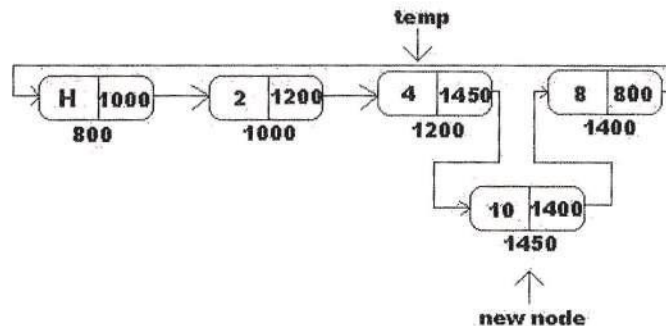
Initially set the pointer **temp** in the head node. We want to insert the **new node** as the last node of the list. The position is **4** (excluding **Head** node.) Always, before perform the operation insert or delete move the **temp** to the previous position of insertion or deletion position. The **new node** is in the memory address **1450**. The **data** stored in the **new node** is **10**. After insert the **new node** the list has changed as follow as.



How can we link the **new node** with list?

1. Move the pointer **temp** to the previous position (3).
2. Store the address of **new node** (1450) in the address field of previous node **temp**.
3. Store **the starting address** value on the address field of **new node**.

Now, the **new node** is connected with list. Another one possible insertion in the list is, insert the **new node** between two existing nodes. This is same as the above example. We want to insert the **new node** at position 3. The position is between two nodes. Like the above, move the pointer **temp** to the previous position of insertion position. So, we should move **temp** to position 2. After insert the list has changed as follow as.



Remember, the nodes in the list are anywhere in the memory. Using pointers we can link the nodes.

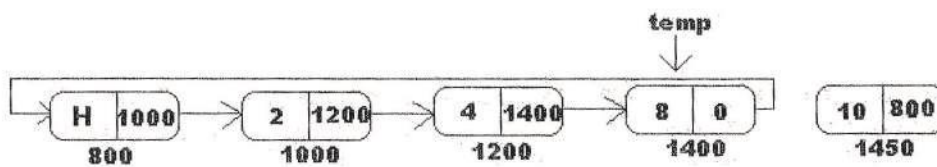
The steps involved in the insertion between two nodes are.

1. Move the pointer **temp** to the previous position (2).

2. Store the address of the next node (in position 4 address (1400)) of **new node** in the address field of **new node**.
3. Store the address of **new node** (1450) in the address field of previous node **temp**.

Like , Insertion there two possible deletions in the list.

- DELETE THE LAST NODE
- DELETE THE NODE BETWEEN TWO EXISTING NODES

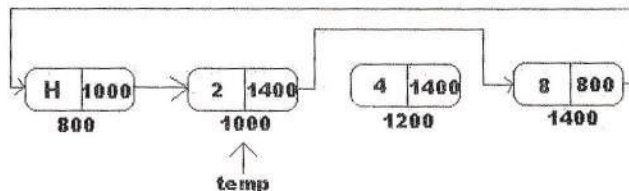


In the first case, like insertion move the pointer **temp** to the previous node of deletion node.

In the above figure, there are four nodes except head node. The last node is in the position 4. To remove that node we can do the following steps.

1. Move the pointer **temp** to the previous position of last node.
2. Store **the starting address** in the address field of **temp**.

Now, the last node has removed from the list.



Another one possible deletion is, remove the node that is placed between two nodes. This is same as the above example. We want to delete the node in position 2. The position

is between two nodes. Like the above, move the pointer **temp** to the previous position of deletion position. So, we should move **temp** to position 1. After delete, the list is changed as follows.

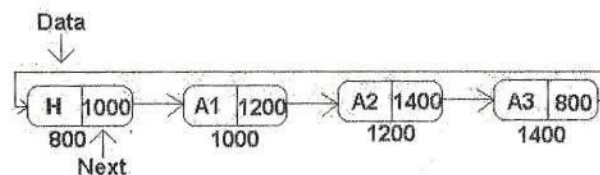
The steps involved in the deletion operation are..

1. Move the pointer **temp** to the previous position (1).
2. Store the address of the node (1400) which is placed next of the deletion node in the address field of **temp** node.

Now, the node has been removed. To show all the elements in the list, we should do the following steps.

1. Move the pointer **temp** to the first node of the list.
2. Everytime after display the element move the pointer **temp** to the next position.
3. Repeat the process until the position is not **the starting address**.

Search the position means, find the element which is stored in the particular position. Initially the pointer **temp** points the first node. Move the pointer **temp** and check the position. If the position is presented then display the element which is stored in that position. Search the no that is presented in the list are not. Initially the pointer **temp** points the first node. Move the pointer **temp** and check the number. If the number is presented then display the element and position.



8.4 Implementation of Linked Lists Structure in C

The following C programs illustrate the use of structures in C to implement linked lists.

Program 1

```
struct employee {
char LastName [80];
char FirstName [80];
float salary;    };
```

```
struct employee *ptr1;
struct employee *ptr2;
struct employee *ptr3;
```

```
main ()
```

```
{
```

```
ptr1 = malloc (sizeof(struct
employee));
```

```
strcpy(ptr1 -> LastName ,
"Smith");
```

```
strcpy(ptr1 -> FirstName ,
"Margaret");
```

```
ptr1 -> salary = 70000;
```

```
ptr2 = malloc (sizeof(struct
employee));
```

```
strcpy(ptr2 -> FirstName , "Robert");
```

```
strcpy(ptr2 -> LastName , ptr1 -> LastName);
```

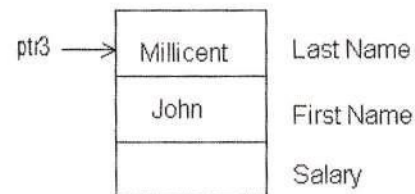
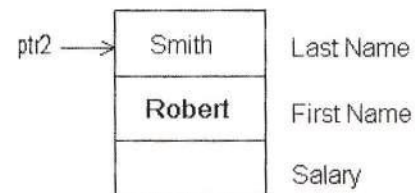
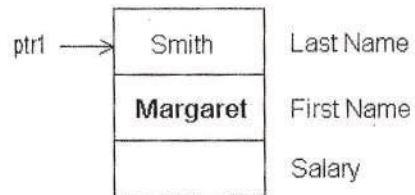
```
ptr2 -> salary = 60000;
```

```
ptr3 = malloc (sizeof(struct employee));
```

```
strcpy(ptr3 -> FirstName , "John");
```

```
strcpy(ptr3 -> LastName , "Millicent");
```

```
ptr3 -> salary = 10000;
```



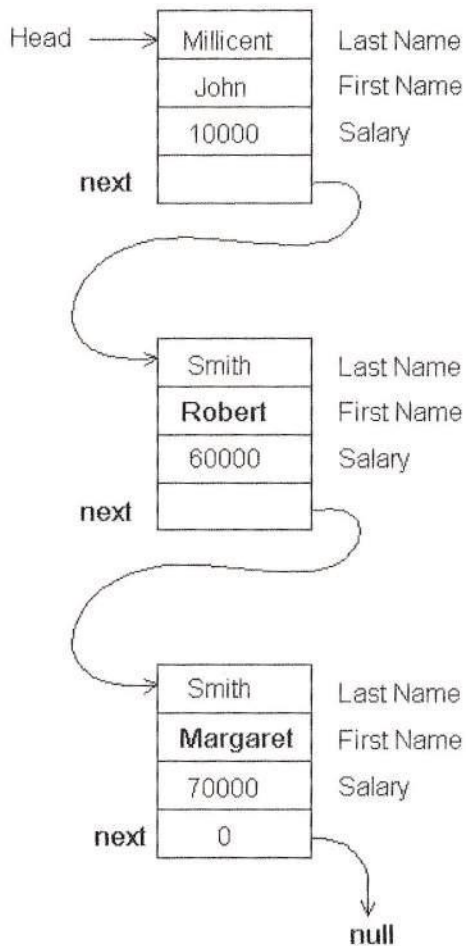
```

printf("%s, %s makes $%f\n",ptr1 ->LastName,ptr1 -
>FirstName,ptr1 ->salary);

printf("%s, %s makes $%f\n",ptr2 ->LastName,ptr2 -
>FirstName,ptr2 ->salary);

printf("%s, %s makes $%f\n",ptr3 ->LastName,ptr3 -
>FirstName,ptr3 ->salary);
}

```



Program 2

```

struct employee
{
char LastName [80];
char FirstName [80];
float salary;
struct employee
*next;
};

```

```

struct employee
*head,*temp;

```

```

main ()
{
head = 0;

```

```

temp = malloc
(sizeof(struct
employee));

```

```

strcpy(temp -> LastName , "Smith");
strcpy(temp -> FirstName , "Margaret");
temp -> salary = 70000;
head = temp;

```



```
temp -> next = 0;
temp = malloc (sizeof(struct employee));
strcpy(temp -> FirstName , "Robert");
strcpy(temp -> LastName , "Smith");
temp -> salary = 60000;
temp -> next = head;
head = temp;
```

```
temp = malloc (sizeof(struct employee));
strcpy(temp -> FirstName , "John");
strcpy(temp -> LastName , "Millicent");
temp -> salary = 10000;
temp -> next = head;
head = temp;
```

```
// print out all elements
temp = head;
while (temp != 0 )
{
printf("%s, %s makes $%f\n",temp -
>LastName,temp ->FirstName,temp ->salary);
temp = temp -> next;
}
}
```

Let us sum up

We have covered about the basic concepts on linked lists and its structure. We have also seen the various types in linked lists with relevant detailed examples.

Learning Activities

a) Fill in the blanks:

3. A singly linked list has two parts namely _____ and _____.

b) State whether true or false:

1. Linked list are more advantageous than arrays. (True / False).
2. Circular linked list have links at both the ends of the node. (True/False).
3. Inserting or Deleting an element at the given position cannot be done using singly linked list.

Answers to Learning Activities

a) Fill in the blanks:

4. Data, pointer.

a) State whether true or false:

1. True.
2. True.
3. False.

Model Questions

1. Explain the insertion of 35 in the given singly linked list – 12, 45, 67, 79, 134.
2. Write detailed notes on Circular linked list.

References

1. Algorithms and Data Structures – Programmes by Nicklaus Wirth.
2. Data Structures using C and C++ by Y.Langesam, M.J. Augenstein and A.M. Tenenbaum

Unit-9

GRAPHS

Structure

Overview

Learning objectives

9.0 GRAPHS

9.1 Adjacency Matrix

9.2 Implementation of a directed Graphs in C

Let us sum up

Answer to learning activities

References

OVERVIEW

This unit is devoted to the detailed study of the Graph data structure, which an advanced data structure.

Learning Objectives

At the end of this unit you will have a clear knowledge on the ways of representing graphs and its implementation in C.

9.0 GRAPHS

It is possible to place all components of the data types we have considered thus far in a linear order. That is, we can identify the first component, the second, etc. It is not always possible to this.

For example, consider the relationship of father and sons. In this case the nodes cannot be put into a linear order. A data type to represent this would need to be a non-linear data structure. What is needed is the ability to represent the nodes and the connections between nodes. A collection of nodes and the connections between them is called a graph.

Basically all non-linear structures are graphs. We can generally explain about a simple limited type of graph called a tree. Tree is an important data structure that represent a hierarchy.

Trees/hierarchies are very common in our life:

- Tree of species (is-a)
- Component tree (part-of)
- Family tree (parent-child)

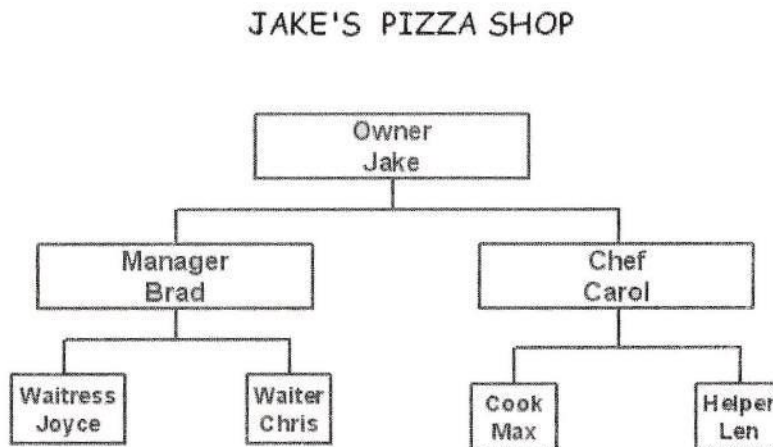


Figure 9.1: graphs example

9.1. ADJACENCY MATRIX

One of the methods of representing Graphs is by creating an equivalent Adjacency matrix for the same. The Adjacency Matrix is constructed as follows.

Given a graph, for each edge (u, v) , set $A[u][v] = 1$; otherwise the entry is 0. If the edge has a weight associated with it, set $A[u][v]$ to the weight. Space requirement is $\Theta(|V|^2)$; It is alright if the graph is *dense*, i.e., $|E| = \Theta(|V|^2)$.

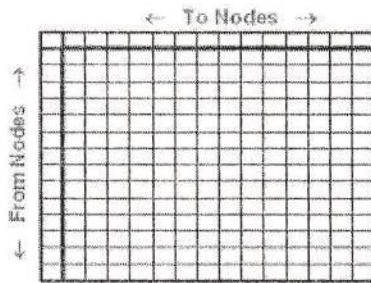
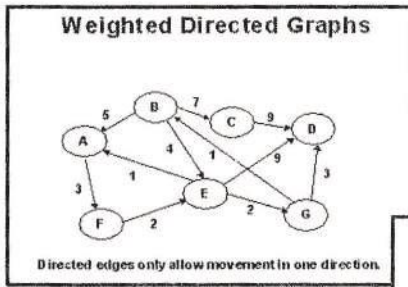


Figure 9.2: adjacency matrix

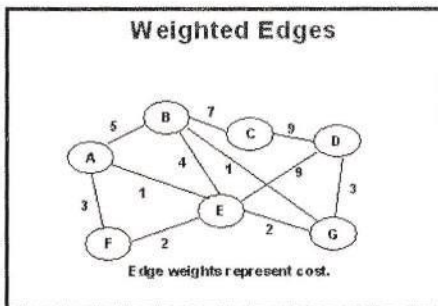
Initially the matrix is empty. Each edge adds an entry in the matrix. A directed graph uses the entire matrix. An undirected graph will have 2 entries per edge. Unweighted graphs insert 1 in the place value and weighted graphs the corresponding weights.

The following figures show some examples for constructing adjacency matrix.



Directed

		TO						
		A	B	C	D	E	F	G
FROM	A	-	3	.
	B	5	-	7	.	4	.	.
	C	.	.	-	9	.	.	.
	D	.	.	.	-	.	.	.
	E	1	.	.	9	-	.	2
	F	2	-	.
	G	.	1	.	3	.	.	-



Undirected

	A	B	C	D	E	F	G
A	-	5	.	.	1	3	.
B	5	-	7	.	4	.	1
C	.	7	-	9	.	.	.
D	.	.	9	-	9	.	3
E	1	4	.	9	-	2	2
F	3	.	.	.	2	-	.
G	.	1	.	3	2	.	-

Figure 9.3: Directed and undirected graph

9.2. IMPLEMENTATION OF DIRECTED GRAPHS IN C

The most fundamental graph problem is to traverse every edge and vertex in a graph in a systematic way. Indeed, most of the basic algorithms you will need for bookkeeping

operations on graphs will be applications of graph traversal.

These include:

- Printing or validating the contents of each edge and/or vertex.
- Copying a graph, or converting between alternate representations.
- Counting the number of edges and/or vertices.
- Identifying the connected components of the graph.
- Finding paths between two vertices, or cycles if they exist.

Since any maze can be represented by a graph, where each junction is a vertex and each hallway an edge, any traversal algorithm must be powerful enough to get us out of an arbitrary maze. For *efficiency*, we must make sure we don't get lost in the maze and visit the same place repeatedly. By being careful, we can arrange to visit each edge exactly twice. For *correctness*, we must do the traversal in a systematic way to ensure that we don't miss anything. To guarantee that we get out of the maze, we must make sure our search takes us through every edge and vertex in the graph.

The key idea behind graph traversal is to mark each vertex when we first visit it and keep track of what we have not yet completely explored. Although bread crumbs or unraveled threads are used to mark visited places in fairy-tale mazes, we will rely on Boolean flags or enumerated types. Each vertex will always be in one of the following three states:

- *undiscovered* - the vertex in its initial, virgin state.
- *discovered* - the vertex after we have encountered it, but before we have checked out all its incident edges.

- *completely-explored* - the vertex after we have visited all its incident edges.

Obviously, a vertex cannot be *completely-explored* before we discover it, so over the course of the traversal the state of each vertex progresses from *undiscovered* to *discovered* to *completely-explored*.

We must also maintain a structure containing all the vertices that we have discovered but not yet completely explored. Initially, only a single start vertex is considered to have been discovered. To completely explore a vertex, we must evaluate each edge going out of it. If an edge goes to an undiscovered vertex, we mark it *discovered* and add it to the list of work to do. If an edge goes to a *completely-explored* vertex, we will ignore it, since further contemplation will tell us nothing new about the graph. We can also ignore any edge going to a *discovered* but not *completely-explored* vertex, since the destination must already reside on the list of vertices to completely explore.

Regardless of which order we use to fetch the next vertex to explore, each undirected edge will be considered exactly twice, once when each of its endpoints is explored. Directed edges will be considered only once, when exploring the source vertex. Every edge and vertex in the connected component must eventually be visited. Why? Suppose the traversal didn't visit everything, meaning that there exists a vertex u that remains unvisited whose neighbor v was visited. This neighbor v will eventually be explored, and we will certainly visit u when we do so. Thus we must find everything that is there to be found.

The order in which we explore the vertices depends upon the container data structure used to store the *discovered* but not *completely-explored* vertices. There are two important possibilities:

- *Queue* - by storing the vertices in a first in, first out (FIFO) queue, we explore the oldest unexplored vertices first. Thus our explorations radiate out slowly from the starting vertex, defining a so-called *breadth-first search*.
- *Stack* - by storing the vertices in a last in, first out (LIFO) stack, we explore the vertices by lurching along a path, visiting a new neighbor if one is available, and backing up only when we are surrounded by previously discovered vertices. Thus our explorations quickly wander away from our starting point, defining a so-called *depth-first search*.

9.3. GRAPH TRAVERSALS

There are many different applications of graphs. As a result, there are many different algorithms for manipulating them. However, many of the different graph algorithms have in common the characteristic that they systematically visit all the vertices in the graph. That is, the algorithm walks through the graph data structure and performs some computation at each vertex in the graph. This process of walking through the graph is called a *graph traversal*.

While there are many different possible ways in which to systematically visit all the vertices of a graph, certain traversal methods occur frequently enough that they are given names of their own. This section presents two of them—depth-first traversal and breadth-first traversal.

Breadth-First Traversal (BFS)

The breadth-first traversal of a tree visits the nodes in the order of their depth in the tree. Breadth-first tree traversal first visits all the nodes at depth zero (i.e., the root), then all the nodes at depth one, and so on.

Since a graph has no root, when we do a breadth-first traversal, we must specify the vertex at which to start the traversal. Furthermore, we can define the depth of a given vertex to be the length of the shortest path from the starting vertex to the given vertex. Thus, breadth-first traversal first visits the starting vertex, then all the vertices adjacent to the starting vertex, and then all the vertices adjacent to those, and so on.

First, the starting vertex is enqueued. Then, the following steps are repeated until the queue is empty:

1. Remove the vertex at the head of the queue and call it v .
2. Visit v .
3. Follow each edge emanating from v to find the adjacent vertex and call it u . If u has not already been put into the queue, enqueue it.

Notice that a vertex can be put into the queue at most once. Therefore, the algorithm must somehow keep track of the vertices that have been enqueued.

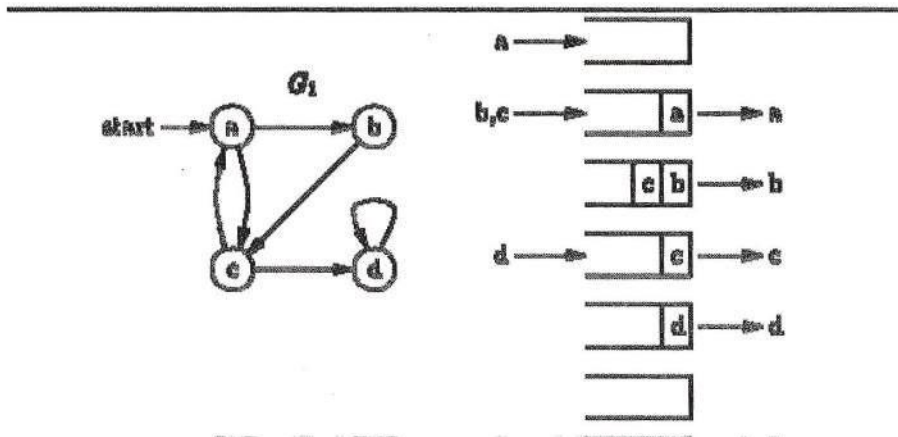


Figure 9.4: Breadth-first traversal.

Figure 9.4 illustrates the breadth-first traversal of the directed graph G_1 starting from vertex a . The algorithm begins by inserting the starting vertex, a , into the empty queue. Next, the head of the queue (vertex a) is dequeued and visited, and the vertices adjacent to it (vertices b and c) are enqueued. When, b is dequeued and visited we find that there is only adjacent vertex, c , and that vertex is already in the queue. Next vertex c is dequeued and visited. Vertex c is adjacent to a and d . Since a has already been enqueued (and subsequently dequeued) only vertex d is put into the queue. Finally, vertex d is dequeued and visited. Therefore, the breadth-first traversal of G_1 starting from a visits the vertices in the sequence

a, b, c, d

DEPTH-FIRST TRAVERSAL (DFS)

A depth-first traversal of a tree always starts at the root of the tree. Since a graph has no root, when we do a depth-first traversal, we must specify the vertex at which to begin.

A depth-first traversal of a tree visits a node and then recursively visits the subtrees of that node. Similarly, depth-first traversal of a graph visits a vertex and then recursively visits all the vertices adjacent to that node. The catch is that the graph may contain cycles, but the traversal must visit every vertex at most once. The solution to the problem is to keep track of the nodes that have been visited, so that the traversal does not suffer the fate of infinite recursion.

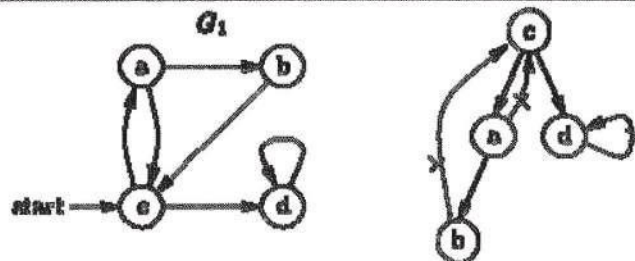


Figure 9.5: Depth-first traversal

For example, Figure illustrates the depth-first traversal of the directed graph G_1 starting from vertex c . The depth-first traversal visits the nodes in the order

c, a, b, d

A depth-first traversal only follows edges that lead to unvisited vertices. As shown in Figure if we omit the edges that are not followed, the remaining edges form a tree. Clearly, the depth-first traversal of this tree is equivalent to the depth-first traversal of the graph.

Let us sum up

We have covered about the basic data structures namely stacks and queues and shown their implementation using arrays in C.

Learning Activities

a) Fill in the blanks:

1. A graph is a _____ type of data structure.
2. Main components of a graph are _____ and _____.
3. Two types of search operations _____ and _____.

b) State whether true or false:

- 1) Queues or Stacks can be used to implement search graphs.
(True/False).

Answers to Learning Activities

a) Fill in the blanks:

- 1) non-linear.
- 2) nodes, edges.
- 3) breadth-first, depth-first.

b) State whether true or false:

1. True.

Model Questions

1. What are Graphs? Show the adjacency matrix representation of a graph.
2. Show the implementation of a directed graph in C.

References

1. An Introduction to Data Structures with Applications by Tremblay, J.P. and Sorenson, P.G.
2. Computer Algorithms, Introduction to Design and Analysis by Sara Baase and Allen Van Gelder

Block 4: Introduction

In this block, we will learn about the concept of trees, and searching and sorting techniques. This block is divided into one unit are as follows

Unit 10: it deals with the Trees, Searching, Sorting and File organizations.

Unit-10

Trees, searching and sorting

Structure

Overview

Learning Objectives

10.1. Trees

10.1.1. Binary Trees

10.1.2. Binary Tree Representations

10.1.3. Tree Traversals

10.1.4. AVL Trees

10.1.4. B-Trees

10.1.5. Applications of Trees

10.2. Searching Techniques

10.2.1. Linear Search

10.2.2. Binary Search

10.3. Sorting Methods

10.3.1. Selection Sort

10.3.2. Insertion Sort

10.3.3. Quick Sort

10.3.4. Heap Sort

10.3.5. Two-way Merge Sort

10.4. File Organizations

10.4.1. Sequential Organization

10.4.2. Indexed Sequential Organization

10.4.3. Direct Organization

Let us sum up

Answers to Learning Activities

References

OVERVIEW

The block begins with a detailed description of trees from basic concepts to their representations and manipulations. Some of the common sorting techniques are illustrated. The methods of searching large amounts of data to find one particular piece of information are considered. Finally various file organizations are described.

LEARNING OBJECTIVES

After completing this block 4 you should be able to:

- Know the structure of various trees and their operations
- Understand the basic mechanisms of searching and sorting
- Write C programs for the above methods.

10.1. TREES

A tree structure means that the data is organized so that items of information are related by branches. Familiar examples of trees are *genealogies* and *organization charts*. Trees are used to help analyze electrical circuits and to represent the structure of mathematical formulas.

In computer applications, one of the most familiar uses of tree structures is to organize file systems. The files are kept in *directories (folders)* that are defined recursively as sequences of directories and files.

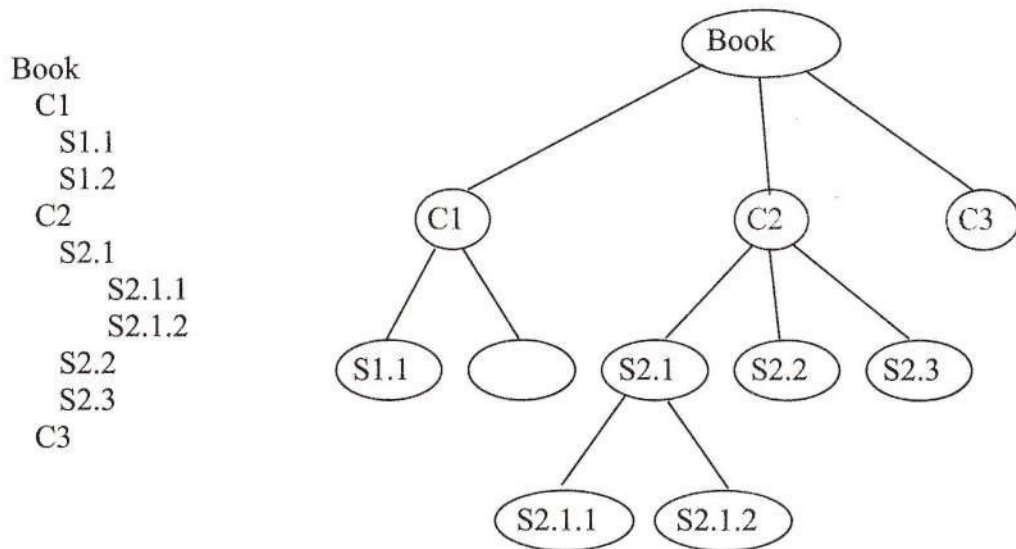


Diagram 10.1

Trees also used to organize information in database systems and to represent the syntactic structure of source programs in compilers.

A *tree* imposes a hierarchical structure on a collection of items called nodes. Diagram 4.1 shows the table of contents of a book and the tree representation. There are many different types of trees which are called as:

- Binary trees and M-ary trees
- Ordered trees
- Rooted trees
- Free trees

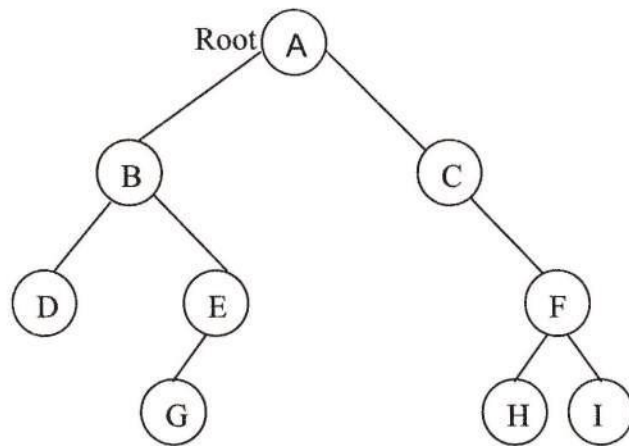


Diagram 10.2

10.1.1. BINARY TREES

A *binary tree* is a finite set of elements that is either empty or partitioned into three disjoint subsets. The first subset contains a single element called the *root* of the tree. The other two subsets are themselves binary trees, called the *left* and *right subtrees* of the original tree. A left or right subtree can be empty.

Each element of a binary tree is called a *node* of the tree. A *node* is often depicted as a letter, a string, or a number with a circle around it. A tree with no nodes is called *null tree*.

If A is the root of a binary tree and B is the root of its left or right subtree, then A is said to be the *father* of B and B is said to be the *left* or *right son* of A. A node that has no sons is called a *leaf*.

The father-son relationship is depicted by a line. Trees are normally drawn top-down with the father above the son. Diagram 10.2 shows sample binary trees. Diagram 10.3 illustrates some structures that are not binary trees.

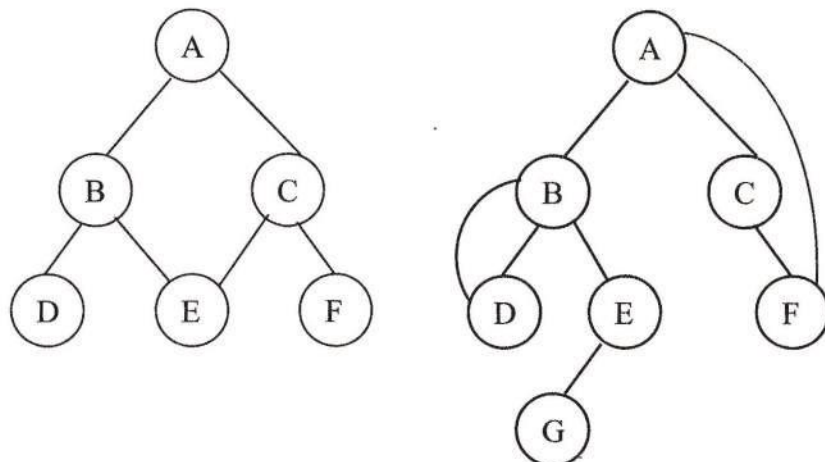


Diagram 10.3

If n_1, n_2, \dots, n_k is a sequence of nodes in a tree such that n_i is the father of n_{i+1} for $1 \leq i < k$, then this sequence is called a *path* from node n_1 to n_k . The length of a path is one less than the number of nodes in the path. For example A-B-E-G is a path of length 3 in Diagram 10.2. There is a path of length of zero from every node to itself.

Node n_1 is an *ancestor* of node n_2 and n_2 is a *descendant* of n_1 if n_1 is either the father of some ancestor of n_2 ; that is if there is a path from node n_1 to node n_2 . A node n_2 is a left descendant of node n_1 if n_2 is either the left son of n_1 or a descendant of the left son of n_1 . A right descendant may be similarly defined. Two nodes are brothers if they are left and right sons of the same father.

The *height* of a node in a tree is the length of a longest path from the node to a leaf. The height of a tree is the height of

the root. The height of a binary tree with N nonleaf nodes is at least $\log_2 N$ and at most $N - 1$.

The *depth* of the node is the length of the unique path from the root to that node. The depth of a binary tree is the maximum level of any leaf in the tree. Thus equals the length of the longest path from the root to any leaf. Thus the depth of the tree of Diagram 10.2 is 4.

Going from the leaves to the root is called “climbing” the tree, and going from the root to the leaves is called “descending” the tree.

If every nonleaf node in a binary tree has nonempty left and right subtrees, the tree is termed a *strictly binary tree*. A strictly binary tree with n leaves always contains $2n - 1$ nodes.

The *level* of a node in a binary tree is defined as follows: the root of the tree has level 0, and the level of any other node in the tree is one higher than the level of its father. For example, in the binary tree of Diagram 10.2, node B is at level 1 and node G is at level 3.

A complete binary tree of depth d is the strictly binary tree of all of whose leaves are at level d . Diagram 10.4 illustrates the complete binary tree of depth 3.

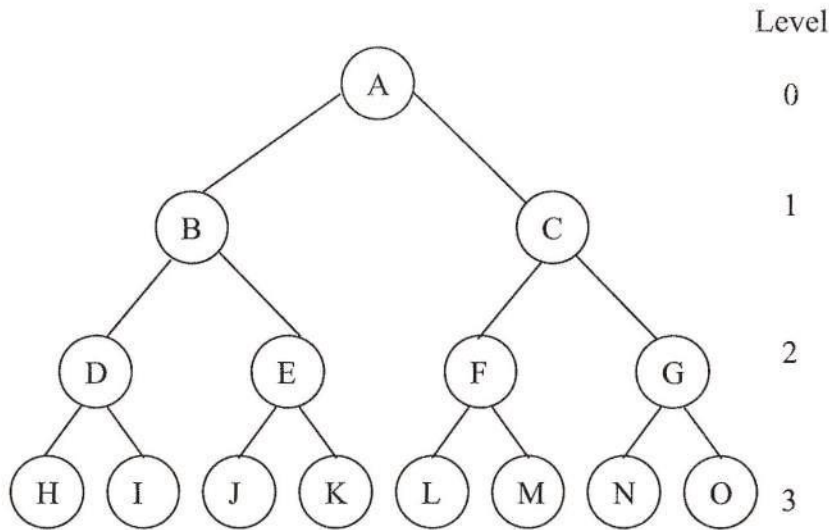


Diagram 10.4

If a binary tree contains m nodes at level l , it contains at most $2m$ nodes at level $l + 1$. Since a binary tree can contain at most one node at level 0 (the root), it can contain at most 2^l nodes at each level l between 0 and d . The binary tree of depth d contains exactly 2^d nodes at level d . The total number of nodes in a complete binary tree of depth d , t_n , equals the sum of the number of nodes at each level between 0 and d . Thus

$$t_n = 2^0 + 2^1 + 2^2 + \dots + 2^d = \sum_{j=0}^d 2^j$$

By induction, it can be shown that this sum equals $2^{d+1} - 1$. Since all leaves in such a tree are at level d , the tree contains 2^d leaves and, therefore, $2^d - 1$ nonleaf nodes.

Similarly, if the number of nodes, t_n , in a complete binary tree is known, depth d can be computed from the above equation. Thus d equals $\log_2(t_n + 1) - 1$. A binary tree of depth d is an almost complete binary tree if:

1. Each leaf in the tree is either at level d or at level $d - 1$

2. For any node n in the tree with a right descendant at level d , all the left descendants of n that are leaves are also at level d .

The nodes of an almost complete binary tree can be numbered so that the root is assigned the number 1, a left son is assigned twice the number assigned its father, and a right son is assigned one more than twice the number its father. Diagram 10.5 illustrates this numbering technique.

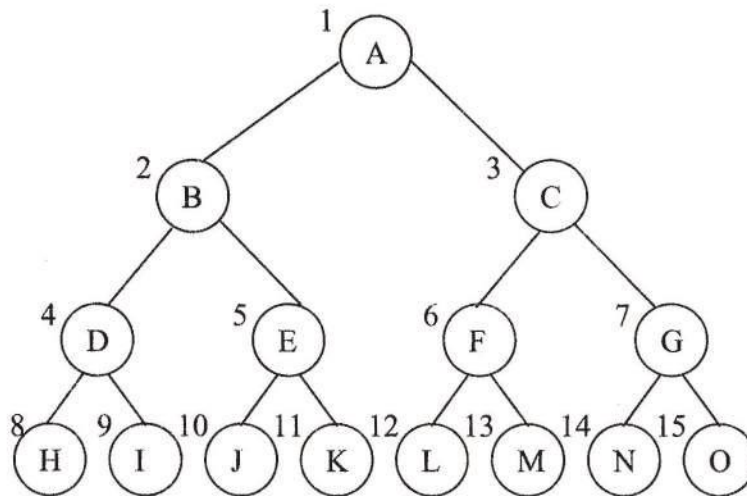


Diagram 10.5

An almost complete binary tree with n leaves has $2n - 1$ nodes, as does a strictly binary tree with n leaves. An almost complete binary tree with n leaves that is not strictly binary has $2n$ nodes.

An almost complete binary tree of depth d is the intermediate between the complete binary tree of depth $d - 1$, that contains $2^d - 1$ nodes, and the complete binary tree of depth d , which contains $2^{d+1} - 1$ nodes.

Operations on Binary Trees:

There are a number of primitive operations that can be applied to a binary tree. If p is a pointer to a node n of a binary tree, then the functions

- $info(p)$ – returns the contents of n
- $left(p)$ – returns pointer to the left son of n
- $right(p)$ – returns pointer to the right son of n
- $father(p)$ – returns pointer to the father of n
- $brother(p)$ – returns pointer to the brother of n

These functions return the null pointer if n has no left son, right son, father, or brother. The logical functions $isleft(p)$ and $isright(p)$ return the value of true if n is a left or right son, respectively, of some other node in the tree, and false otherwise. The $isleft$ function may be implemented as follows:

```
q = father(p);
if (q == null)
    return (false);           /* p points to the root */
if (left(q) == p)
    return (true);
return(false);
```

or, even simpler, as $father(p) \&\& p == left(father(p))$. $isright$ may be implemented in a similar manner, or by calling $isleft.brother(p)$ may be implemented using $isleft$ or $isright$ as follows:

```
if (father(p) == null)
    return(null);
if (isleft(p))
    return (right(father(p)));
return(left(father(p)));
```


In constructing a binary tree, the operations `maketree`, `setleft`, and `setright` are useful.

- `maketree(x)` – creates a new binary tree consisting of a single node with information field `x` and returns a pointer to that node.
- `setleft(p, x)` – accepts a pointer `p` to a binary tree node with no left son. It creates a new left son of `node(p)` with information field `x`.
- `setright(p, x)` – analogous to `setleft` except that it creates a right son of `node(p)`.

10.1.2. BINARY TREE REPRESENTATIONS

The representations of binary trees are array representation and linked representation.

Node Representation of Binary Trees:

Tree nodes may be implemented as array elements or as allocations of a dynamic variable. Each node contains `info`, `left`, `right`, and `father` fields which points to the node's left son, right son, and father respectively. Using the array implementation, the binary tree can be declared as

```
#define NUMNODES 500
struct nodetype {
    int info;
    int left;
    int right;
    int father;
};
struct nodetype node[numnodes];
```

Under this representation, the operations `info(p)`, `left(p)`, `right(p)`, and `father(p)` are implemented by references to

node[p].info, node[p].left, node[p].right, and node[p].father, respectively. The operations *isleft(p)*, *isright(p)*, and *brother(p)* can be implemented using the operations *left(p)*, *right(p)*, and *father(p)*.

Once the array of nodes is declared, an available list can be created by executing the following statements:

```
int avail, i;
{
    avail = 1;
    for (i=0; i<NUMNODES; i++)
        node[i].left = i + 1;
    node[NUMNODES-1].left = 0;
}
```

In the linked array representation of a binary tree each node in a tree is taken from the available pool when needed and returned to the available pool when no longer in use. The available list is not a binary tree but a linear list whose nodes are linked together by the left field.

Alternatively, a node may be defined by

```
struct nodetype {
    int info;
    struct nodetype *left;
    struct nodetype *right;
    struct nodetype *father;
};
typedef struct nodetype *NODEPTR;
```

The operations *info(p)*, *left(p)*, *right(p)*, and *father(p)* would be implemented by references to *p->info*, *p->left*, *p->right*, and *p->father*, respectively. An explicit available list is not needed in this representation.

The routines *getnode* and *freenode* simply allocate and free nodes using the routines *malloc* and *free*. This representation is called the *dynamic node* representation of a binary tree.

Both the linked array representation and the dynamic node representation are implementations of an abstract linked (node) representation in which explicit pointers link together the nodes of a binary tree.

The *maketree* function may be written as follows:

```
NODEPTR maketree(x)
int x;
{
    NODEPTR p;
    p = getnode ( );
    p->info = x;
    p->left = NULL;
    p->right = NULL;
    return(p);
}
```

The routine *setleft*(p, x) sets a node with contents x as the left son of node(p):

```
setleft (p, x)
NODEPTR p;
int x;
{
    if (p == NULL)
        printf ( " void insertion \n");
    else if ( p->left != NULL)
        printf ( " invalid insertion \n");
    else
        p->left = maketree (x);
}
```

It is not always necessary to use father, left, and right fields. If a tree is always traversed in downward fashion (from the root to the leaves), the father operation is never used; so a father field is unnecessary. Similarly, if a tree is always traversed in upward fashion (from the leaves to the root), left and right fields are not needed.

Internal and External Nodes:

In the linked representation of binary trees, left and right pointers are needed only in nonleaf nodes. Sometimes two separate sets of nodes are used for nonleaves and leaves. Nonleaf nodes contain info, left, and right fields and are allocated as dynamic records or as an array of records.

Leaf nodes do not contain a left or right field and are kept as a single info array that is allocated sequentially as needed. Alternatively, they can be allocated as dynamic variables containing only an info value.

In this case leaf nodes and nonleaf nodes are distinguished, leaves are called *external nodes* or *terminal nodes* and nonleaves are called *internal nodes* or *non-terminal nodes*.

A son pointer within an internal node must be identified as pointing to an internal or external node. This can be done in C in two ways. One technique is to declare two different node types and pointer types and to use a union for internal nodes, with each alternative containing one of the two pointer types.

The other technique is to retain a single type of pointer and a single type of node, where the node is a union that does or does not contain left and right pointer fields.

Implicit Array Representation of Binary Tree:

The n nodes of an almost complete binary tree can be numbered from 1 to n , so that the number assigned a left son is twice the number assigned its father, and the number assigned a right son is 1 more than twice the number assigned its father.

In C, arrays start at position 0; therefore instead of numbering the tree nodes from 1 to n , they are numbered from 0 to $n-1$. For any node with position p , $0 \leq p \leq n-1$,

- (i) $\text{leftson}(p)$ is at $2p + 1$ if $2p+1 \leq n-1$. if $2p+1 > n-1$, then node p has no left son.
- (ii) $\text{rightson}(p)$ is at $2p + 2$ if $2p+2 \leq n-1$. if $2p+2 > n-1$, then node p has no right son.
- (iii) $\text{father}(p)$ is at position $(p - 1)/2$ if $p \neq 0$. When $p = 0$, node p is the root and has no father.

Thus the operation $\text{left}(p)$ is implemented by $2 * p + 1$ and $\text{right}(p)$ by $2 * p + 2$. Given a left son at position p , its right brother is at position $p + 1$ and, given a right son at position p , its left brother is at position $p - 1$. $\text{father}(p)$ is implemented by $(p - 1) / 2$.

p points to a left son if p is odd. Thus, the test for whether node p is a left son is to check whether $p \% 2$ is not equal to 0. Diagram 4.6 illustrates an array that represents an almost complete binary tree.

This representation can clearly be used for all binary trees though in most cases there will be a lot of unutilized space. Diagram 4.7 illustrates the array representation for the non-almost-complete binary trees.

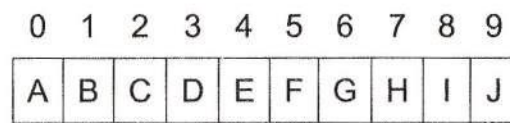
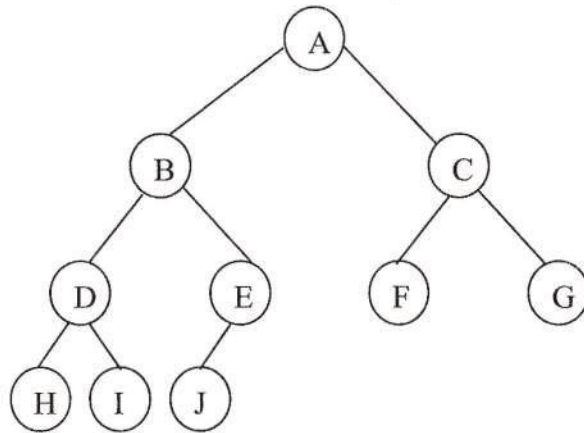


Diagram 10.6

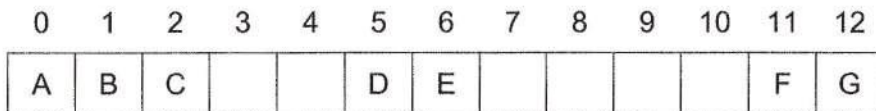
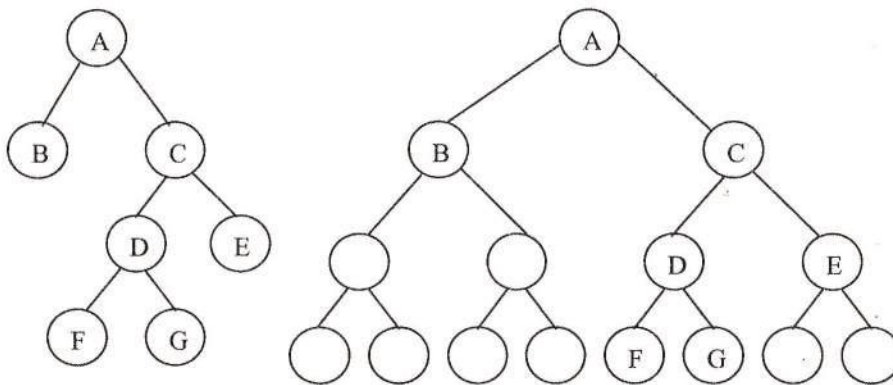


Diagram 10.7

The implicit representation is also called the *sequential representation*.

So an array element is allocated whether or not it serves to contain a node of a tree. Therefore, unused array elements

must be flagged as null tree nodes. This can be achieved in two ways:

- Setting a special value to `info[p]` if node `p` is null. This special value should be invalid as the information content of a legitimate tree node.
- Alternatively, a logical flag field, used may be added to each node. Each node then contains two fields: `info` and `used`. The entire structure is contained in an array `node`. `used(p)`, implemented as `node[p].used`, is `TRUE` if node `p` is not a null node and `FALSE` if it is a null node. `info[p]` is implemented by `node[p].info`.

The binary tree can be represented as an array of father of each node in the tree. As an example, the tree of Diagram 4.6 is represented as follows:

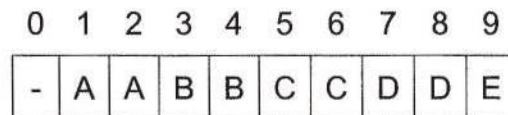


Diagram 10.8

The root node does not have a father, so the father value is entered as `NULL`.

Choosing a Binary Tree Representation:

The sequential representation is simpler, although it is necessary to ensure that all pointers are within array bounds. This sequential representation clearly saves storage space for trees known to be almost complete, since it eliminates the fields `left`, `right`, and `father` and does not even require a `used` field.

It is also efficient for trees that are only a few nodes short of being almost complete although a `used` field might then be required. However, the sequential representation can only be

used in a context in which only a single tree is required, or where the number of trees needed and each of their maximum sizes is fixed in advance.

By contrast, the linked representation requires left, right, and father fields but allows much more flexible use of the collection of nodes. In the linked representation, a particular node may be placed at any location in any tree, whereas in the sequential representation a node can be utilized only if it is needed at a specific location in a specific tree.

In addition, the total number of trees and nodes is limited only by a amount of available memory. Thus the linked representation is preferable in the general, dynamic situation of many trees of unpredictable shape.

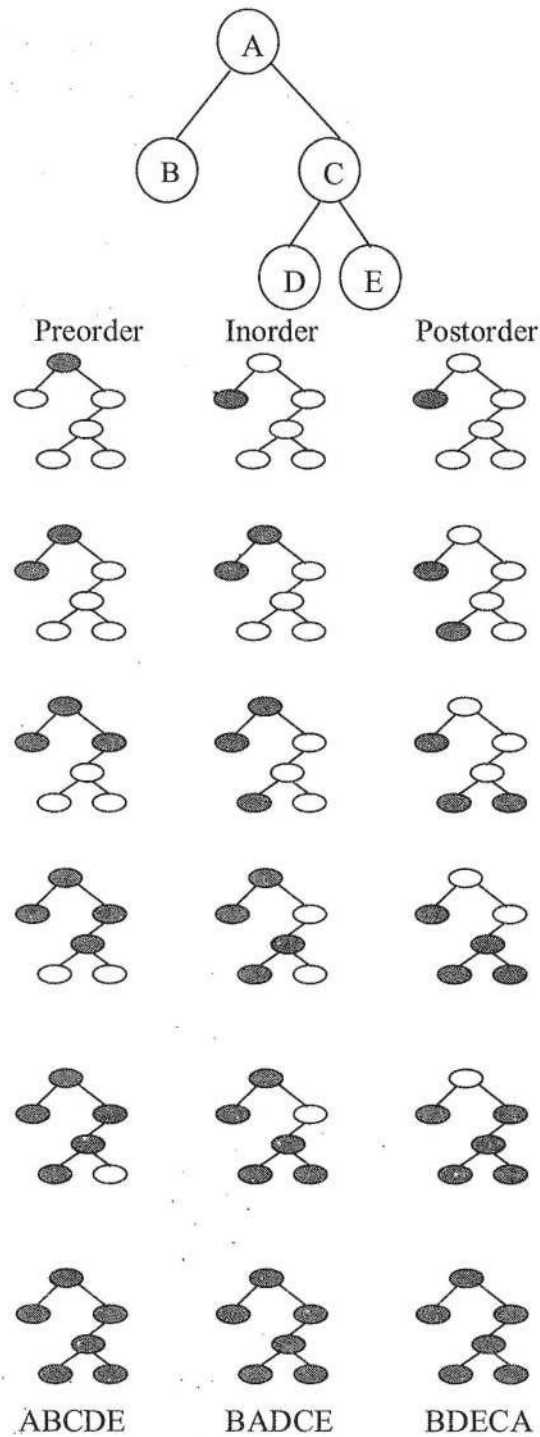
10.1.3. TREE TRAVERSALS

One of the most common operations performed on tree structures is traversal. This is a procedure by which each node is processed exactly once in a systematic manner. The meaning of “processed” depends on the nature of the application. There are three main ways of traversing a binary tree:

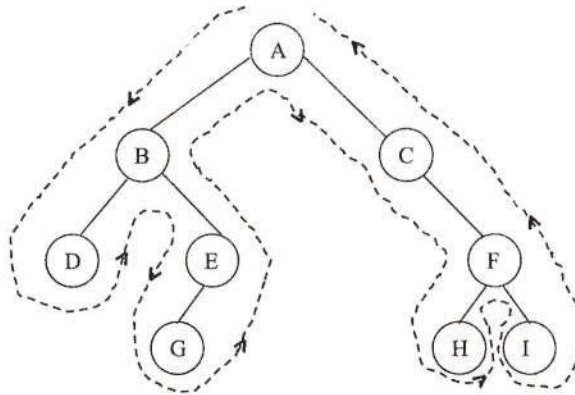
- *Preorder (or depth-first order)*, where the node is first visited, then the left and right subtrees are visited.
- *Inorder (or symmetric order)*, where the left subtree is first visited, then the node is visited, then the right subtree is visited.
- *Postorder*, where the left and right subtrees are visited, the node is visited.

In each of these methods, nothing needs to be done to traverse an empty binary tree. Diagram 4.9 illustrates a binary

tree and its three traversals. The easiest way to implement each order is by using recursion.



A useful trick for producing the three node orderings is the following: If a tree is walked around the outside of the tree as shown in Diagram 4.10, starting at the root, moving counterclockwise, and staying as close to the tree gives useful trick for producing the three node orderings.



Preorder: ABDEGCFHI
 Inorder: DBGEAHFIC
 Postorder: DGE BHIFCA

Diagram 10.10

For preorder, a node is listed when it is passed for the first time. For postorder, a node is listed when it passed for the last time that is when it moves to its parent. For inorder, a leaf is listed for the first time pass, and an internal node for the second time passed.

Binary Tree Traversals in C:

The three C routines *pretrav*, *intrav*, and *posttrav* print the contents of a binary tree in preorder, inorder, and postorder, respectively. The parameter to each routine is a pointer to the root node of a binary tree. The dynamic node representation is used.

```

void pretrav(NODEPTR tree)
{
    if (tree != NULL) {
        printf("%d\n", tree->info);          /* Visit the root */
        pretrav(tree->left);                 /* traverse the left subtree */
        pretrav(tree->right);                /*traverse the right subtree */
    }
}

```

```

void intrav(NODEPTR tree)
{
    if (tree != NULL) {
        intrav(tree->left);                 /* traverse the left subtree */
        printf("%d\n", tree->info);        /* Visit the root */
        intrav(tree->right);                /*traverse the right subtree */
    }
}

```

```

void posttrav(NODEPTR tree)
{
    if (tree != NULL) {
        posttrav(tree->left);              /* traverse the left subtree */
        posttrav(tree->right);             /*traverse right subtree */
        printf("%d\n", tree->info);        /* Visit the root */
    }
}

```

The following is a nonrecursive routine to traverse a binary tree in inorder.

```

#define MAXSTACK 100

void intrav2(NODEPTR tree)
{
    struct stack {
        int top;
        NODEPTR item[MAXSTACK];
    } s;
    NODEPTR p;

```

```

s.top = -1;
p = tree;
do {
    /* travel down left branches as far as possible */
    /*      saving pointers to nodes passed      */
    while (p != NULL)    {
        push (s, p);
        p = p->left ;
    }
    /* check if finished */
    if (!empty(s)) {
        /* at this point the left subtree is empty */
        p = pop(s);
        printf ("%d\n", p->info); /* visit the root */
        p = p->right; /* traverse the right subtree */
    }
}
}

```

The recursive *intrav* generally executes much more quickly than the nonrecursive *intrav2*. The primary cause of the inefficiency of *intrav2* as written is the calls to push, pop, and empty.

If the words “left” and “right” are interchanged in the preceding definitions, three new traversal orders, which are called the converse preorder, converse inorder, and converse postorder, respectively. The converse traversal orders for the example tree of Diagram 4.9 are

ACBED (*converse preorder*)

ECDAB (*converse inorder*)

EDCBA (*converse postorder*)

Expression Trees:

A strictly binary tree can be used to represent an expression containing operands and binary operators. The root of the strictly binary tree contains an operator that is to be applied to the results of evaluating the expressions represented by the left and right subtrees.

1. Every leaf is labeled by an operand and consists of that operand alone.
2. Every nonleaf (internal) node is labeled by an operator. Suppose n is labeled by a binary operator θ , and the expression E_1 is represented by the left child and E_2 by the right child. Then n represents the expression $E_1 \theta E_2$.

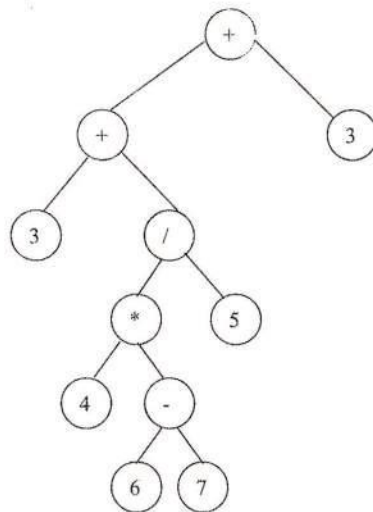


Diagram 10.11 Expression $3 + 4 * (6 - 7) / 5 + 3$

Diagram 10.11 illustrates the above expression and its tree representation. The preorder of the listing of the labels gives the *prefix* form of an expression, where the operator precedes its left and right operands. Similarly, a postorder listing of the labels of an expression tree gives the *postfix* (or *Polish*) representation of an expression. A postfix expression is written

by placing the operator after its two operands. No parentheses are necessary in the prefix and postfix expressions.

The inorder traversal of an expression tree gives the *infix* expression itself, but with no parentheses. As a binary expression tree does not contain parentheses, the ordering of the operations is implied by the structure of the tree.

Threaded Binary Trees:

The wasted NULL links of binary trees can be replaced by *threads*. A binary tree is threaded according to a particular traversal order. A NULL pointer in the left or right field of a node with empty left or right subtrees the inorder predecessor or successor can be assigned.

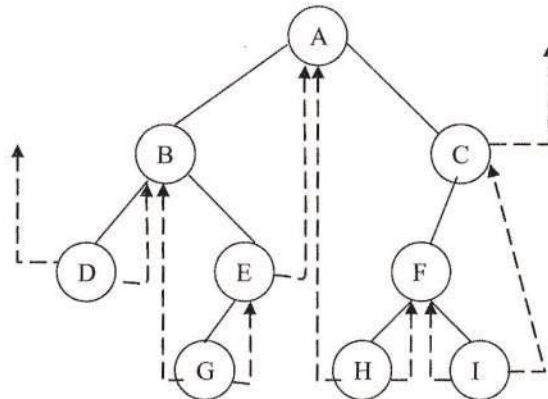


Diagram 10.12

Diagram 10.12 shows the binary trees with threads replacing NULL pointers in nodes with empty left or right subtrees. The threads are drawn with dotted lines to differentiate them from tree pointers. The leftmost and rightmost node in the tree still has a NULL pointer, since it has no inorder predecessor or successor respectively.

In the *right in-threaded binary trees* (Diagram 10.13) the right NULL pointers are replaced by the inorder successor. To

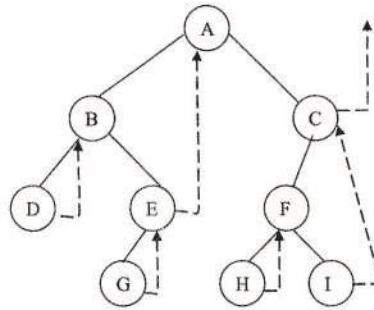


Diagram 10.13

implement a right in-threaded binary tree under the dynamic node implementation of a binary tree, a separate boolean flag, *rthread*, is included within each node to indicate whether or not its right pointer is a thread. The *rthread* field of the rightmost node of a tree is also set to TRUE. Thus a node is defined as follows:

```

struct nodetype {
    int info;
    struct nodetype *left;
    struct nodetype *right;
    int rthread;
};
typedef struct nodetype * NODEPTR;

```

The following is a routine to implement inorder traversal of a right in-threaded binary tree.

```

void intrav3 (NODEPTR tree)
{
    NODEPTR p, q;
    p = tree;
    do {
        q = NULL;
        while ( p != NULL) {           /* traverse left branch */
            q = p;
            p = p->left;
        }
        if (q!=NULL) {

```

```

        printf ("%d\n", q->info);
        p = q->right;
        while (q->rthread && p != NULL) {
            printf("%d\n", p->info);
            q = p;
            p = p->right;
        }
    }
} while (q!=NULL);
}

```

In a right in-threaded binary tree the inorder successor of any node can be found efficiently. The routines *maketree*, *setleft*, and *setright* are as follows.

```

NODEPTR maketree(int x)
{
    NODEPTR p;
    p = getnode();
    p->info = x;
    p->left = NULL;
    p->right = NULL;
    p->rthread = TRUE;
    return(p);
}

void setleft(NODEPTR p, int x)
{
    NODEPTR q;

    if (p == NULL)
        error ("void insertion");
    else if (p->left != NULL)
        error ("invalid insertion");
    else {
        q = getnode( );
        q->info = x;
        p->left = q;
        q->left = NULL;
    }
}

```



```

        /* the inorder successor of node(q) is node(p) */
        q->right = p;
        q->rthread = TRUE;
    }
}

void setright(NODEPTR p, int x)
{
    NODEPTR q, r;

    if (p == NULL)
        error ("void insertion");
    else if (!p->rthread)
        error ("invalid insertion");
    else {
        q = getnode( );
        q->info = x;
        /* save the inorder successor of node(p) */
        r = p->right;
        p->right = q;
        p->rthread = FALSE;
        q->left = NULL;
        /* The inorder successor of node(q) is */
        /* the previous successor of node(p) */
        q->right = r;
        q->rthread = TRUE;
    }
}

```

4.1.4 AVL TREES

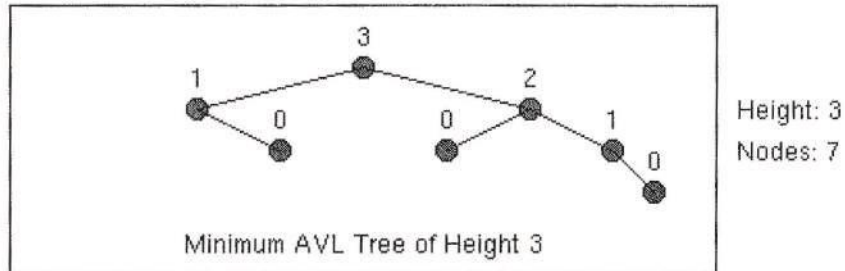
An AVL tree is a self-balancing binary search tree, and the first such data structure to be invented. In an AVL tree the heights of the two child subtrees of any node differ by at most one, therefore it is also called height-balanced. Lookup, insertion, and deletion all take $O(\log n)$ time in both the average and worst cases. Additions and deletions may require the tree to be rebalanced by one or more tree rotations.

The AVL tree is named after its two inventors, G.M. Adelson-Velsky and E.M. Landis, who published it in their 1962 paper "An algorithm for the organization of information."

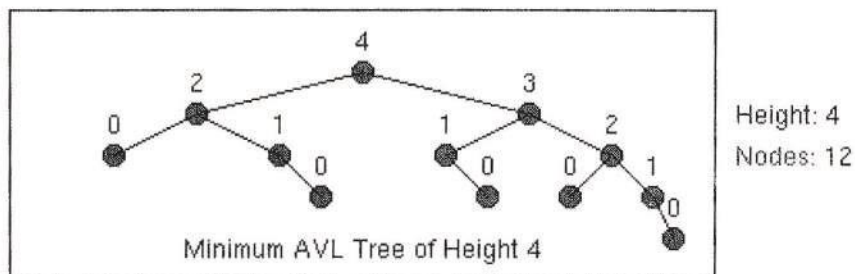
The **balance factor** of a node is the height of its right subtree minus the height of its left subtree. A node with balance factor 1, 0, or -1 is considered balanced. A node with any other balance factor is considered unbalanced and requires rebalancing the tree. The balance factor is either stored directly at each node or computed from the heights of the subtrees.

AVL trees are often compared with red-black trees because they support the same set of operations and because red-black trees also take $O(\log n)$ time for the basic operations. AVL trees perform better than red-black trees for lookup-intensive applications.

AVL (Adelson-Velskii and Landis) Tree Examples



A minimum AVL tree of height 3



A minimum AVL tree of height 4

The basic operations of an AVL tree generally involve carrying out the same actions as would be carried out on an unbalanced binary search tree, but preceded or followed by one or more operations called tree rotations, which help to restore the height balance of the subtrees.

Insertion

Insertion into an AVL tree may be carried out by inserting the given value into the tree as if it were an unbalanced binary search tree, and then retracing one's steps toward the root updating the balance factor of the nodes.

If the balance factor becomes -1, 0, or 1 then the tree is still in AVL form, and no rotations are necessary. If the balance factor becomes 2 or -2 then the tree rooted at this node is unbalanced, and a tree rotation is needed. At most a single or double rotation will be needed to balance the tree. Only the nodes traversed from the insertion point to the root of the tree need be checked, and rotations are a constant time operation, and because the height is limited to $O(\log(n))$, the execution time for an insertion is $O(\log(n))$.

Deletion

If the node is a leaf, remove it. If the node is not a leaf, replace it with either the largest in its left subtree or the smallest in its right subtree, and remove that node. Thus the node that is removed has at most one leaf. After deletion retrace the path back up the tree to the root, adjusting the balance factors as needed.

The retracing can stop if the balance factor becomes -1 or 1 indicating that the height of that subtree has remained unchanged. If the balance factor becomes 0 then the height of the subtree has decreased by one and the retracing needs to continue. If the balance factor becomes -2 or 2 then the subtree

is unbalanced and needs to be rotated to fix it. If the rotation leaves the subtree's balance factor at 0 then the retracing towards the root must continue since the height of this subtree has decreased by one. This is in contrast to an insertion where a rotation resulting in a balance factor of 0 indicated that the subtree's height has remained unchanged.

The time required is $O(h)$ for lookup plus $O(h)$ rotations on the way back to the root; so the operation can be completed in $O(\log n)$ time.

Lookup

Lookup in an AVL tree is performed exactly as in an unbalanced binary search tree, except because of the height-balancing of the tree, a lookup takes $O(\log n)$ time. No special provisions need to be taken, and the tree's structure is not modified by lookups. (This is in contrast to splay tree lookups, which do modify their tree's structure.)

If each node additionally records the size of its subtree (including itself and its descendants), then the nodes can be retrieved by index in $O(\log n)$ time as well. Once a node has been found in a balanced tree, the *next* or *previous* node can be obtained in amortized constant time. (In a few cases, about $2 \cdot \log(n)$ links will need to be traversed. In most cases, only a single link need be traversed. On the average, about two links need to be traversed.)

4.1.4. B-TREE

A *multiway search tree* of order n is a general tree in which each node has n or fewer subtrees and contains one fewer key than it has subtrees. The nodes that have the maximum number of subtrees are called *full nodes*. If all the leaves are at the same level then the tree is said to be *balanced*.

Formally, a *B-tree* of order n is a balanced order- n multiway search tree with the following properties:

1. The root is either a leaf or has at least two sons.
2. Each nonroot node contains at least $(n-1)/2$ keys.
3. Each path from the root to a leaf has the same length.

A B-tree of order n is also called an n - $(n - 1)$ tree or an $(n - 1)$ - n tree. Thus, a 4-5 tree is a B-tree of order 5. Diagram 10.14 shows a B-tree of order 5, in which it is assumed that at most three records fit in a leaf block.

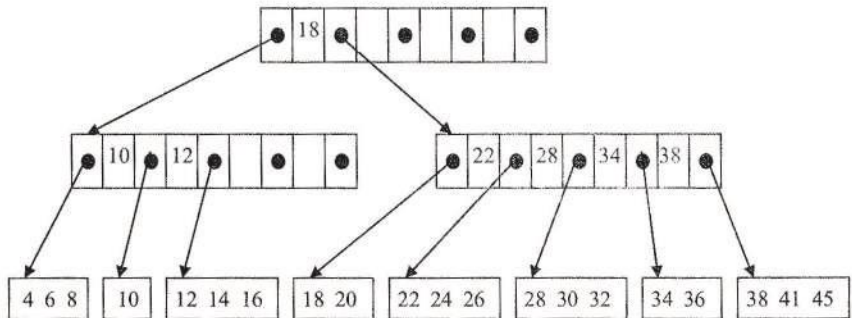


Diagram 10.14

A B-tree can be viewed as a hierarchical index in which each node occupies a block in external storage. The root of the B-tree is the first level index. Each non-leaf node in the B-tree is of the form

$$(p_0, k_1, p_1, k_2, p_2, \dots, k_n, p_n)$$

where, p_i is a pointer to the i^{th} child of the node, $0 \leq i \leq n$, and k_i is a key, $1 \leq i \leq n$. The keys within a node are in sorted order so $k_1 < k_2 < \dots < k_n$.

All keys in the subtree pointed to by p_0 are less than k_1 . For $1 \leq i < n$, all keys in the subtree pointed to by p_i have values greater than or equal to k_i and less than k_{i+1} . All keys in the subtree pointed to by p_n are greater than or equal to k_n .

Retrieval

To retrieve a record r with key value x , we trace the path from the root to the leaf that contains r , if it exists in the file. This path is traced by successively fetching internal nodes $(p_0, k_1, p_1, k_2, p_2, \dots, k_n, p_n)$ and finding the position of x relative to the keys k_1, k_2, \dots, k_n .

If $k_i \leq x < k_{i+1}$, then fetching the node pointed to by p_i and this process is repeated. If $x < k_1$, p_0 is used to fetch the next node; if $x \geq k_n$, p_n is used. When this process comes to a leaf, the record with key value x is searched.

Insertion

To insert a record r with key value x into a B-tree, the leaf L is located at which r should belong. If there is space for r in L , then r is inserted into L in the proper sorted order.

If there is no space for r in L , a new block L' is created and half of the records are moved from L to L' , inserting r into its proper place in L or L' . This process is performed to its father F if it has already n pointers. Inserting the record with key value 23 into a B-tree shown in Diagram 10.14 produces the B-tree in Diagram 10.15.

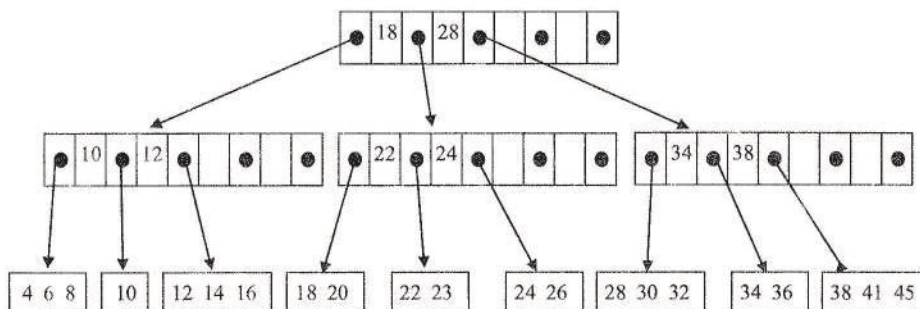


Diagram 10.15

Deletion

To delete a record r with key value x , the leaf L containing r is found and it is removed from L , if it exists. The key values of L 's father F are updated to reflect the change in L .

If L becomes empty after deletion, the keys and pointers in F are adjusted to reflect the removal of L . If the number of sons of F is less than $n/2$, the node F' immediately to the left (or the right) of F at the same level is examined.

If F' has at least $n/2 + 1$ sons, the keys and pointers in F and F' are evenly distributed between F and F' . Then the keys of F and F' are modified in the father of F . If necessary, this change is recursively rippled to as many ancestors of F .

If F' has exactly $n/2$ sons, then F and F' are combined into a single node. The key and pointer to F' are to be removed from the father of F .

Removing record 10 from the B-tree in Diagram 10.15 results into B-tree of Diagram 10.16.

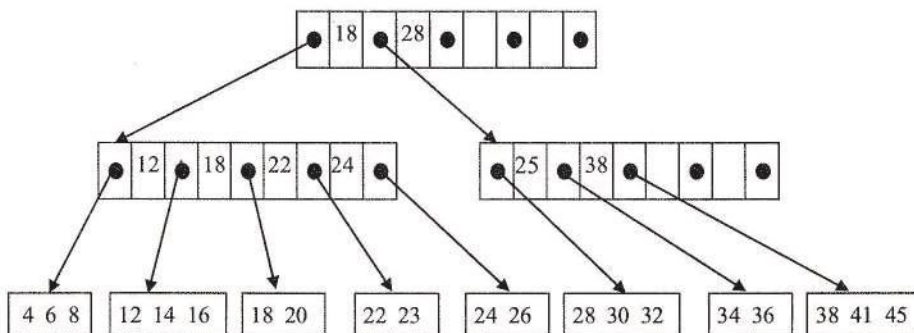


Diagram 10.16

4.1.5. APPLICATIONS OF TREES

A binary tree can be used to find all duplicates in a list of numbers. The first number in the list is placed as the root of the tree with empty left and right subtrees. Each successive number

in the list is then compared to the number in the root. If it matches, then it is the duplicate. If it is smaller, the left subtree is examined; if it is larger, the right subtree is examined.

If the subtree is empty, the number is not a duplicate and is placed into a new node at that position in the tree. If the subtree is nonempty, the number is compared with the contents of the root of the subtree and the entire process is repeated with the subtree. An algorithm for doing this follows.

```
/* read the first number and insert it */
/* into a single-node binary tree */

scanf("%d", &number);
tree = maketree(number);
while (there are numbers left in the input) {
scanf("%d", &number);
p = q = tree;
while (number != info(p) && q != NULL) {
    p = q;
    if in (number < info(p))
        q = left(p);
    else
        q = right(p);
}
if (number == info(p) )
    printf("%d %s \n", number, "is a duplicate");
/* insert number to the right or left of p */
else if (number < info(p))
    setleft(p, number);
else
    setright(p, number);
}
```

Diagram 10.17 illustrates the tree constructed from the input 14, 15, 4, 9, 7, 18, 3, 5, 16, 4, 20, 17, 9, 14, 5.

Given a list of numbers in an input file, it can be printed in ascending order. As the numbers are read, they can be inserted into a binary tree such as the one of Diagram 10.17. However, unlike the previous algorithm used to find duplicates, duplicate values are also placed in the tree.

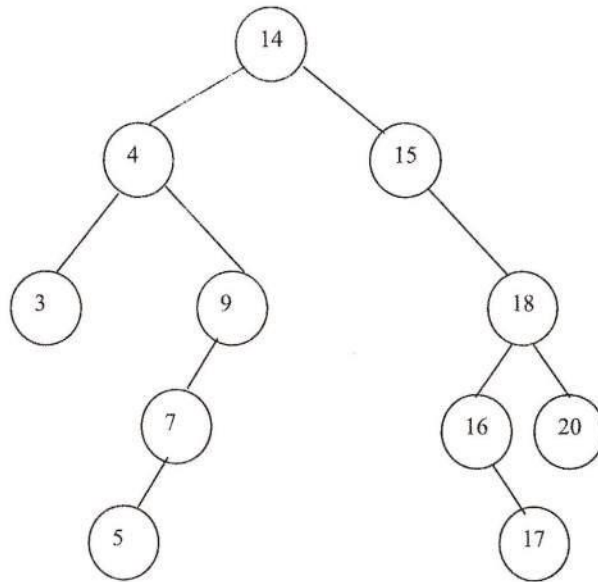


Diagram 10.17

When a number is compared with the contents of a node in the tree, a left branch is taken if the number is smaller than the contents of the node and a right branch if it is greater or equal to the contents of the node. Thus if the input list is 14, 15, 4, 9, 7, 18, 3, 5, 16, 4, 20, 17, 9, 14, 5 the binary tree of Diagram 10.18 is produced.

Such a binary tree has the property that all elements in the left subtree of a node n are less than the contents of n , and all elements in the right subtree of n are greater than or equal to the contents of n . A binary tree that has this property is called a binary search tree. If a binary search tree is traversed in inorder (left, root, right) and the contents of each node are printed as the node is visited, the numbers are printed in ascending order.

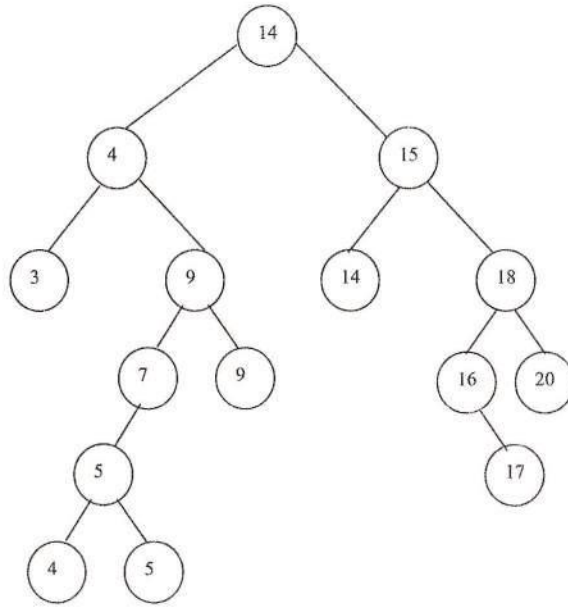


Diagram 10.18

Learning Activity 10.1

- ◆ Write your answer in the space given below.
 - ◆ Check your answer with the one given at the end of the unit.
1. Answer the following questions about the tree of Diagram 10.19.

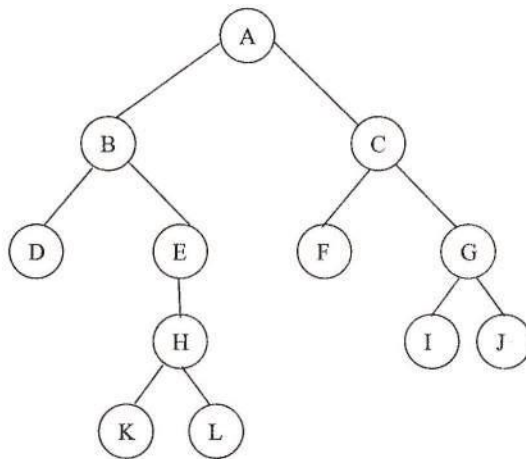
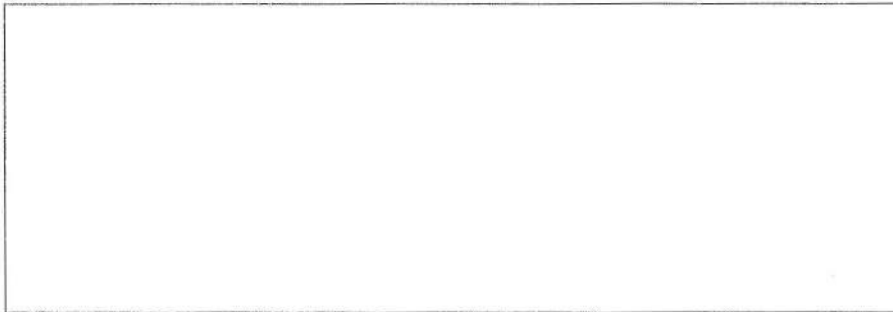


Diagram 10.19

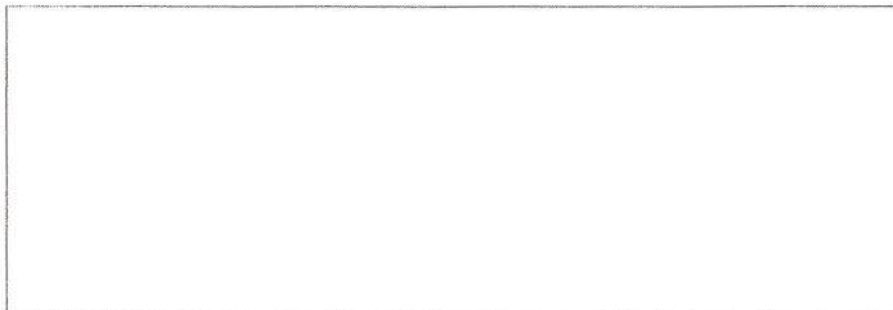
- a) Which nodes are leaves?
- b) Which node is the root?

- c) What is the father of node C?
- d) Which nodes are sons of C?
- e) Which nodes are ancestors of E?
- f) Which nodes are descendants of E?
- g) What are the right brothers of D and E?
- h) Which nodes are to the left and to the right of G?
- i) What is the depth of node C?
- j) What is the height of node C?

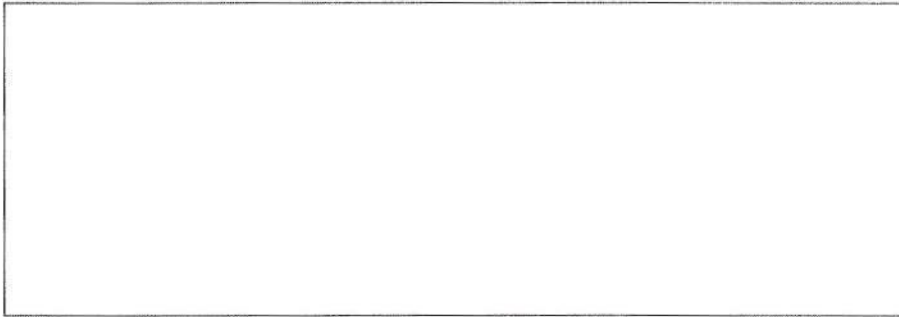


2. List the nodes of Diagram 10.19 (consider H as the left son of E) in

- a) Preorder,
- b) Inorder, and
- c) Postorder



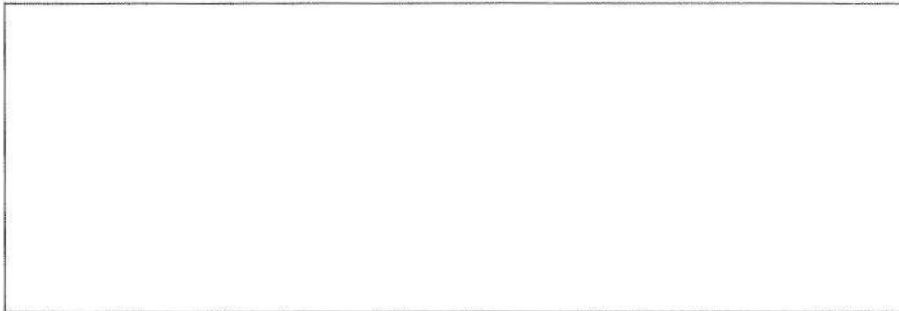
3. Prove that a binary tree with N internal nodes has $N+1$ external nodes.



4. Draw tree representations for the following expressions:

a) $A * (B + C * (D + E))$

b) $A * (B + C) * D + E$



10.2 SEARCHING TECHNIQUES

A table or a file is a group of elements, each of which is called a *record*. A *key* is associated with each record which is used to differentiate among different records. If the key is contained in the record is called an *internal key* or an *embedded key*. *External keys* are keys stored in a separate table that includes pointers to the records.

A *primary key* is a unique set of key and if the key is not unique, the key is called as *secondary key* if there are two or more records with the same key.

A search algorithm is an algorithm that accepts an argument x and tries to find a record whose key is x . The algorithm may return the entire record or, it may return a pointer to that record. A successful search is called as *retrieval*. A table of records in which a key is used for retrieval is called a *search table* or *dictionary*.

It is possible that the search for a given argument in a table is unsuccessful if there is no such record in the table; then the algorithm may return a special “null record” or null pointer. In some times if a search is unsuccessful the new record may be added to the table. An algorithm that does this is called a *search and insertion algorithm*.

The table or file may be organized as an array of records, a linked list, a tree, or even a graph. Searches in which the entire table is constantly in main memory are called *internal* searches, whereas those in which most of the table is kept in auxiliary storage are called *external* searches.

Dictionary as an Abstract Data Type

A search table or a dictionary can be presented as an abstract data type.

```
typedef KEYTYPE . . .      /* a type of key */
typedef RECTYPE . . .     /* a type of record */
RECTYPE nullrec = . . .   /* a “null” record */

KEYTYPE keyfunct(r)
RECTYPE r;
{ . . .
};
```

The abstract data type table may then be declared as a set of records.

Algorithmic Notation:

A table (keys plus records) organized as an array might be declared in C by

```
#define TABLESIZE 1000
typedef KEYTYPE . . .
typedef RECTYPE . . .

struct {
    KEYTYPE k;
    RECTYPE r;
} table [TABLESIZE];
```

or it might be declared as two separate arrays:

```
KEYTYPE k[TABLESIZE];
RECTYPE r[TABLESIZE];
```

In the first case the *i*th key would be referenced as `table[i].k`; in the second case, it will be `k[i]`.

Similarly, for a table organized as a list, either the array representation of a list or the dynamic representation of a list could be used. In the former case the key of the record pointed to by a pointer *p* would be referenced as `node[p].k`; in the latter case, as `p -> k`.

10.2.1. LINEAR SEARCH

The simplest technique for searching an unsorted table for a particular record is to scan each entry in the table in a sequential manner until the desired record is found. This method is called as *linear (or sequential) search*.

This search is applicable to a table organized either as an array or as a linked list. Let *k* be an array of *n* keys, `k[0]`

through $k[n-1]$, and r an array of records, $r[0]$ through $r[n - 1]$, such that $k[i]$ is the key of $r[i]$. An algorithm for such a search procedure is as follows:

```
for (i = 0; i < n; i++)
    if (key == k[i])
        return (i);
return (-1);
```

This function returns the index number of the matching entry if there is one; otherwise, it returns -1.

Storing the table as a linked list has the advantage that the size of the table can be increased dynamically as needed. If the table is organized as a linear linked list pointed by *table* and linked by a pointer field *next*. The sequential insertion search for a linked list may be written as follows:

```
q = null;
for (p = table; p != null && k(p) != key; p = next(p))
    q = p;
if (p != null)          /* this means that k(p) == key */
    return (p);
/* insert a new node */
s = getnode( );
k(s) = key;
r(s) = rec;
next(s) = null;
if (q == null)
    table = s;
else
    next (q) = s;
return(s);
```

Efficiency of Sequential Searching:

The number of comparisons made by a sequential search in a table of size n can given as: if the record is the first

one in the table, only one comparison is performed; if the record is the last one in the table, n comparisons are necessary. On the average, a successful search will take $(n + 1)/2$ comparisons and an unsuccessful search will take n comparisons. In any case, the number of comparisons is $O(n)$.

Searching an Ordered Table:

If the table is stored in ascending or descending order of the record keys, the searching method can be improved. In this case only $n/2$ comparisons are needed. This is because the missing of a key can be easily known as a larger key is encountered.

Indexed Sequential Search:

To improve the efficiency of the search in a sorted table an auxiliary table, called an *index*, is set aside in addition to the sorted table. This method is called the *indexed sequential search*.

Each element in the index consists of a key *kindex* and a pointer to the record in the file that corresponds to *kindex*. The elements in the index, as well as the elements in the file, must be sorted on the key (Diagram 10.20). The algorithm used for searching an indexed sequential file is as follows:

```
for (i = 0; i < indxsize && kindex(i) <= key; i++)
    ;
lowlim = (i == 0) ? 0 : pindex(i - 1);
hilim = (i == indxsize) ? n-1 : pindex(i) - 1;
for (j = lowlim; j <= hilim && k(j) != key; j++)
    ;

return((j > hilim) ? -1 : j);
```

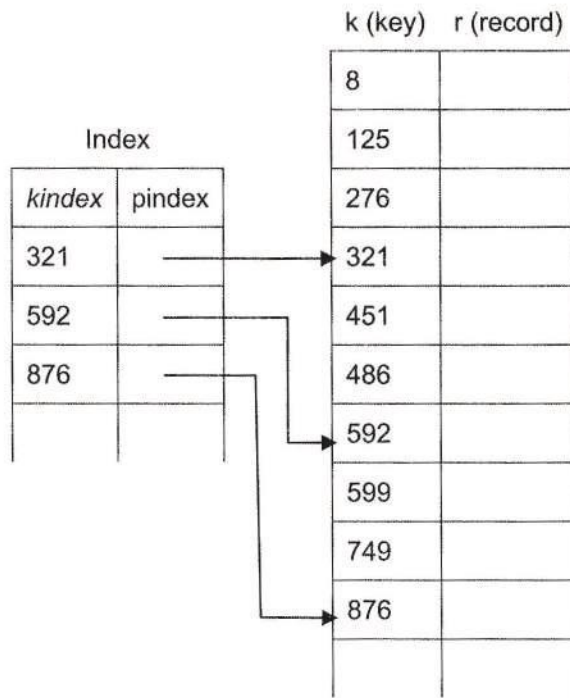



Diagram 10.20

10.2.2. BINARY SEARCH

The most efficient method of searching a sorted sequential table without the use of indices or tables is the binary search. The argument is compared with the key of the middle element of the table. If they are equal, the search ends successfully; otherwise, either the upper or lower half of the table must be searched in a similar manner. Because of the division of the table to be searched into two equal parts, this search is called *binary search*.

A recursive C routine for binary search is written as follows:

```
int binsrch(int a[], int x, int low, int high)
{
    int mid;
    If (low > high)
        return (-1);
    mid = (low + high) / 2;
    return (x == a[mid] ? mid : x < a[mid] ?
```

```

        binsrch(a, x, low, mid-1) :
        binsrch(a, x, mid+1, high);
    }

```

The function above accepts an array *a* and an element *x* as input and returns the index *i* in array *a* such that *a*[*i*] equals *x*, or -1 if no such *i* exists.

The nonrecursive version of the binary search algorithm is presented as follows:

```

low = 0;
high = n - 1;
while(low <= high) {
    mid = (low + high) / 2;
    if (key < k[mid])
        high = mid - 1;
    else if (key > k[mid])
        low = mid + 1;
    else
        return (mid);          /*successful*/
}
return (-1);
}

```

A trace of this algorithm for the sample table 75, 151, 203, 275, 318, 489, 524, 591, 647, and 727 is given for *x* = 275, 727, and 340 in Table 10.1.

Each comparison in the binary search reduces the number of possible candidates by a factor of 2. Thus the maximum number of key comparisons is never more than $\log_2 n$. Thus the running time of binary search is $O(\log n)$.

Search for 275				Search for 727				Search for 340			
Iteration	L	H	M	Iteration	L	H	M	Iteration	L	H	M
1	1	10	5	1	1	10	5	1	1	10	5
2	1	4	2	2	6	10	8	2	6	10	8
3	3	4	3	3	9	10	9	3	6	7	6
4	4	4	4	4	10	10	10	4	7	7	7
								5	7	6	

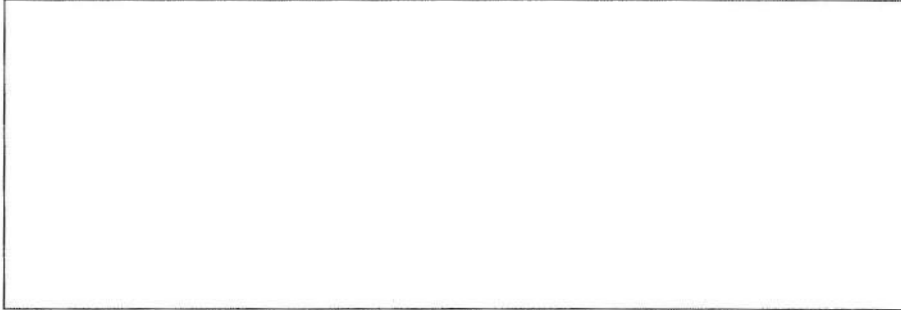
Table 10.1 Binary-search trace

Learning Activity 10.2

- ◆ Write your answer in the space given below.
 - ◆ Check your answer with the one given at the end of the unit.
1. Modify the sequential search algorithm to add a record *rec* with key *key* to the table if key is not already there.

2. Write an algorithm for sequential search in a sorted table.

3. Give the binary trace for $x = 90$ and 500 in the following array of elements. 23, 76, 90, 112, 371, 405, 450, 537, 620, 777



4.3 SORTING TECHNIQUES

A *file* of size of n is a sequence of n items $r[0], r[1], \dots, r[n - 1]$. Each item in the file is called a *record*. A *key* $k[i]$ is associated with each record $r[i]$. The key is usually a subfield of the entire record. The file is said to be sorted on the key if $i < j$ implies that $k[i]$ precedes $k[j]$ in some ordering on the keys.

Depending on the data type of the key, records can be sorted numerically or, more generally, alphanumerically. In numerical sorting, the records are arranged in ascending or descending order according to the numerical value of the key.

In the example of the telephone book, the file consists of all the entries in the book. Each entry is a record. The key upon which the file is sorted is the name field of the record. Each record also contains fields for an address and a telephone number.

A sort can be classified as being *internal* if the records that it is sorting are in main memory or *external* if some of the records that it is sorting are in auxiliary storage. The main difference between the two is that an internal sort can access any item easily whereas an external sort must access items sequentially or at least in large blocks.

It is possible for two records in a file to have the same key. A sorting method is said to be *stable* if it preserves the relative order of duplicate keys in the file.

The general criteria for judging a sorting algorithm are

- How fast can it sort information in an average case?
- How fast are its best and worst cases?
- Does it exhibit natural or unnatural behavior?
- Does it rearrange elements with equal keys?

Bubble Sort:

Let x is an array of integers of which the first n are to be sorted so that $x[i] \leq x[j]$ for $0 \leq i < j < n$. The basic idea underlying the bubble sort is to pass through the file sequentially. In each pass an element is compared with its predecessor ($x[i]$ with $x[i-1]$) and the two elements are interchanged if they are not in proper order. The elements are like bubbles in a tank of water – each seeks its own level. A simple form of the bubble sort is:

```
void bubble (int x[ ], int n)
{
    int i, j, t;

    for (i = 1; i < n; i++)
        for (j = n-1; j >= i; j--) {
            if (x[j-1] > x[j]) {
                /* exchange elements */
                t = x[j-1];
                x[j-1] = x[j];
                x[j] = t;
            }
        }
}
```

Considering the file 25 57 48 37 12 92 86 33. The following comparisons are made on the first pass.

x[7] with x[6]	(33 with 86)	Interchange
x[6] with x[5]	(33 with 92)	Interchange
x[5] with x[4]	(33 with 12)	No Interchange
x[4] with x[3]	(12 with 37)	Interchange
x[3] with x[2]	(12 with 48)	Interchange
x[2] with x[1]	(12 with 57)	Interchange
x[1] with x[0]	(33 with 26)	Interchange

Thus, after the first after the first pass, the file is in the order 12 25 57 48 37 33 92 86. The complete set of iterations is the following:

Iteration 0	25	57	48	37	12	92	86	33
Iteration 1	12	25	57	48	37	33	92	86
Iteration 2	12	25	33	57	48	37	86	92
Iteration 3	12	25	33	37	57	48	86	92
Iteration 4	12	25	33	37	48	57	86	92
Iteration 5	12	25	33	37	48	57	86	92
Iteration 6	12	25	33	37	48	57	86	92
Iteration 7	12	25	33	37	48	57	86	92

Diagram 10.21

There are $n-1$ passes and $n-1$ comparisons on each pass. Thus the total number of comparisons is $(n - 1) * (n - 1)$, which is $O(n^2)$.

10.3.1. SELECTION SORT

A selection sort selects the element with the lowest value and exchanges it with the first element. Then, from the remaining $n - 1$ elements, the element with the smallest key is

found and exchanged with the second element, and so forth. The exchanges continue to the last two elements. The code that follows shows the basic selection sort.

```
void selectionsort(int x[], int n)
{
    int i, minidx, j, min;

    for (i = 0; i < n - 1; i++) {
        min = x[0];
        minidx = 0;
        for (j = i + 1; j < n; j++)
            if (x[j] < min) {
                min = x[j];
                minidx = j;
            }
        x[minidx] = x[i];
        x[i] = min;
    }
}
```

The first pass makes $n - 1$ comparisons, the second pass makes $n - 2$, and so on. Therefore, there is a total of

$$(n - 1) + (n - 2) + (n - 3) + \dots + 1 = n * (n - 1) / 2$$

comparisons, which is $O(n^2)$. There is little additional storage required. Therefore, the sort may be categorized as $O(n^2)$.

For example, if the selection method were used on the following array 25 57 48 37 12 92 86 33 each pass would look like this:

Iteration 0	25	57	48	37	12	92	86	33
Iteration 1	12	57	48	37	25	92	86	33
Iteration 2	12	25	48	37	57	92	86	33
Iteration 3	12	25	33	37	57	92	86	48
Iteration 4	12	25	33	37	57	92	86	48

Iteration 5	12	25	33	37	48	92	86	57
Iteration 6	12	25	33	37	48	57	86	92
Iteration 7	12	25	33	37	48	57	86	92

Diagram 10.22

10.3.2. INSERTION SORT

The insertion sort initially sorts the first two members of the array. Next, the algorithm inserts the third member into its sorted position in relation to the first two numbers. Then it inserts the fourth element into the list of three elements. The process continues until all the elements have been sorted. The code for a version of the insertion sort is shown next.

```

void insertionsort(int x[], int n)
{
    int i, k, y;

    for(k = 1; k < n; k++) {
        /* insert x[k] into the sorted file */
        y = x[k];
        /* move down 1 position all elements greater than y */
        for (i = k - 1; i >= 0; && y < x[i]; i--)
            x[i + 1] = x[i];
        /* insert y at proper position */
        x[i + 1] = y;
    }
}

```

The number of comparisons occur during an insertion sort depends upon the ordering of the file. If the file is sorted, only one comparison is made on each pass, so that the sort is $O(n)$. If the file is initially sorted in the reverse order, the sort is $O(n^2)$, since the total number of comparisons is

$$(n - 1) + (n - 2) + \dots + 3 + 2 + 1 = n * (n - 1)/2$$

The complete set of iterations for the insertion sort for the file with elements 25 57 48 37 12 92 86 33 is shown in Diagram 10.23.

Iteration 0	25	57	48	37	12	92	86	33
Iteration 1	25	57	48	37	12	92	86	33
Iteration 2	25	57	48	37	12	92	86	33
Iteration 3	25	48	57	37	12	92	86	33
Iteration 4	25	37	48	57	12	92	86	33
Iteration 5	12	25	37	48	57	92	86	33
Iteration 6	12	25	37	48	57	92	86	33
Iteration 7	12	25	37	48	57	86	92	33
Iteration 8	12	25	33	37	48	57	86	92

Diagram 10.23

4.3.3. QUICKSORT

Quicksort is a divide-and-conquer method for sorting. This sort is also called as *partition exchange sort*. Let x be an array, and n the number of elements in the array to be sorted. An element a called as *pivot* is selected from a specific position within the array (for example, $a = x[0]$). Suppose that the elements of x are partitioned so that a is placed into position j and the following conditions hold:

1. Each of the elements in positions 0 through $j - 1$ is less than or equal to a .
2. Each of the elements in positions $j + 1$ through $n - 1$ is greater than or equal to a .

If these two conditions hold for a particular a and j , a is the j th smallest element of x , so that a remains in position j

when the array is completely sorted. If this process is repeated with the subarrays $x[0]$ through $x[j - 1]$ and $x[j + 1]$ through $x[n - 1]$ and any subarrays created by the process in successive iterations, the final result is a sorted file.

The outline of an algorithm *quick*(lb, ub) to sort all the elements in an array x between positions lb and ub as follows:

```
if (lb >= ub)
    return;
partition(x, lb, ub, j);
quick(x, lb, j - 1);
quick(x, j + 1, ub);
```

The object of *partition* is to allow a specific element to find its proper position with respect to the others in the subarray. One way to effect a partition efficiently is the following: Let $a = x[lb]$ be the element whose final position is sought. Two pointers, up and down, are initialized to the upper and lower bounds of the subarray, respectively.

At any point during execution, each element in a position above up is greater than or equal to a , and each element in a position below down is less than or equal to a . the two pointers up and down are moved towards each other in the following fashion.

1. Repeatedly increase the pointer down by one position until $x[down] > a$.
2. Repeatedly decrease the pointer up by one position until $x[up] <= a$.
3. If $up > down$, interchange $x[down]$ with $x[up]$.

The process is repeated until the condition in step 3 fails ($up \leq down$), at which point $x[up]$ is interchanged with $x[lb]$, whose final position was sought, and j is set to up .

The algorithm can be implemented by the following procedure.

```
void partition (int x[], int lb, int ub, int *pj)
{
    int a, down, temp, up;

    a = x[lb];
    up = ub;
    down = lb;
    while (down < up)
        while (x[down] <= a && down < ub)
            down ++;          /* move up the array */
        while (x[up] > a)     /* move down the array */
            up --;
        if (down < up) {
            /* interchange x[down] and x[up] */
            temp = x[down];
            x[down] = x[up];
            x[up] = temp;
        }
    }
    x[lb] = x[up];
    x[up] = a;
    *pj = up;
}
```

if a file of size n is a power of 2, such that $n = 2^m$, and $m = \log_2 n$. If the position of pivot is the middle of the array, then there will be n comparisons on the first pass, after which the file is split into two subfiles each of size $n/2$, approximately. For each of these two files there are $n/2$ comparisons, and a total of four files of each size $n/4$ are formed. Thus the total number of comparisons for the entire sort is

$$n + 2*(n/2) + 4*(n/4) + 8*(n/8) + \dots + n*(n/n)$$

or

$$n + n + n + n + \dots + n \text{ (m terms)} = n * m$$

comparisons. There are m terms because the file is subdivided m times. Thus the total number of comparisons is $O(n*m)$ or $O(n \log n)$.

If an initial array is given as 12 25 57 48 37 33 92 86 the steps are performed to obtain the sorted list.

25 57 48 37 12 92 86 33

12	25	57	48	37	92	86	33
----	----	----	----	----	----	----	----

12	25	57	48	37	92	86	33
----	----	----	----	----	----	----	----

12	25	48	37	33	57	92	86
----	----	----	----	----	----	----	----

12	25	37	33	48	57	92	86
----	----	----	----	----	----	----	----

12	25	33	37	48	57	92	86
----	----	----	----	----	----	----	----

12	25	33	37	48	57	86	92
----	----	----	----	----	----	----	----

12	25	33	37	48	57	86	92
----	----	----	----	----	----	----	----

Diagram 10.24

10.3.4. HEAPSORT

The heapsort is an in-place (does not require any additional storage space) sort that requires only $O(n \log n)$ operations regardless of the order of the input.

A *descending heap* (also called *max heap* or *descending partially ordered tree*) of size n is defined as an almost complete binary tree of n nodes such that the content of each node is less than or equal to the content its father. If the sequential representation of an almost complete binary tree is used, this condition reduces to inequality

$$\text{info}[j] \leq \text{info}[(j-1)/2] \text{ for } 0 \leq ((j-1)/2) < j \leq n-1$$

For this condition the root of the tree contains the largest element in the heap. Also any path from the root to a leaf is an unordered list in descending order.

It is also possible to define an *ascending heap* (or a *min heap*) as an almost complete binary tree such that the content of each node is greater than or equal to the content of its father. In this structure, the root contains the smallest element of the heap, and any path from the root to a leaf is an ascending order.

Heapsort Procedure:

A heapsort procedure is outlined in C as follows:

```
void heapsort (int x[ ], int n)
{
    int i, elt, s, f, ivalue;

    /* preprocessing phase; create initial heap */
    for (i = 1; i < n; i++) {
        elt = x[i];
        s = i;
        f = (s-1)/2;
```

```

while (s > 0 && x[f] < elt) {
    x[s] = x[f];
    s = f;
    f = (s-1)/2;
}
x[s] = elt;
}
/* selection phase; repeatedly remove x[0], insert it */
/*   in its proper position and adjust the heap   */
for (i = n-1; i > 0; i--) {
    ivalue = x[i];
    x[i] = x[0];
    f = 0;
    if (i == 1)
        s = -1;
    else
        s = 1;
    if (i > 2 && x[2] > x[1])
        s = 2;
    while (s >= 0 && ivalue < x[s]) {
        x[f] = x[s];
        f = s;
        s = 2*f+1;
        if (s+1 <= i-1 && x[s] < x[s+1])
            s = s+1;
        if (s > i-1)
            s = -1;
    }
    x[f] = ivalue;
}
}

```

A complete binary tree with n nodes has $\log(n + 1)$ levels. Thus if each element in the array were a leaf, requiring it to be filtered through the entire tree both while creating and adjusting the heap, the sort would still be $O(n \log n)$.

Diagram 4.25 illustrates the creation of a heap of size 8 from the original file 25 57 48 37 12 92 86 33. Diagram 4.26

illustrates the adjustment of the heap as $x[0]$ is repeatedly selected and placed into its proper position in the array and the heap is readjusted, until all the elements are processed.

In the average case the heapsort is not efficient as the quicksort. However, heapsort is far superior to quicksort in the worst case. In fact, heapsort remains $O(n \log n)$ in the worst case.

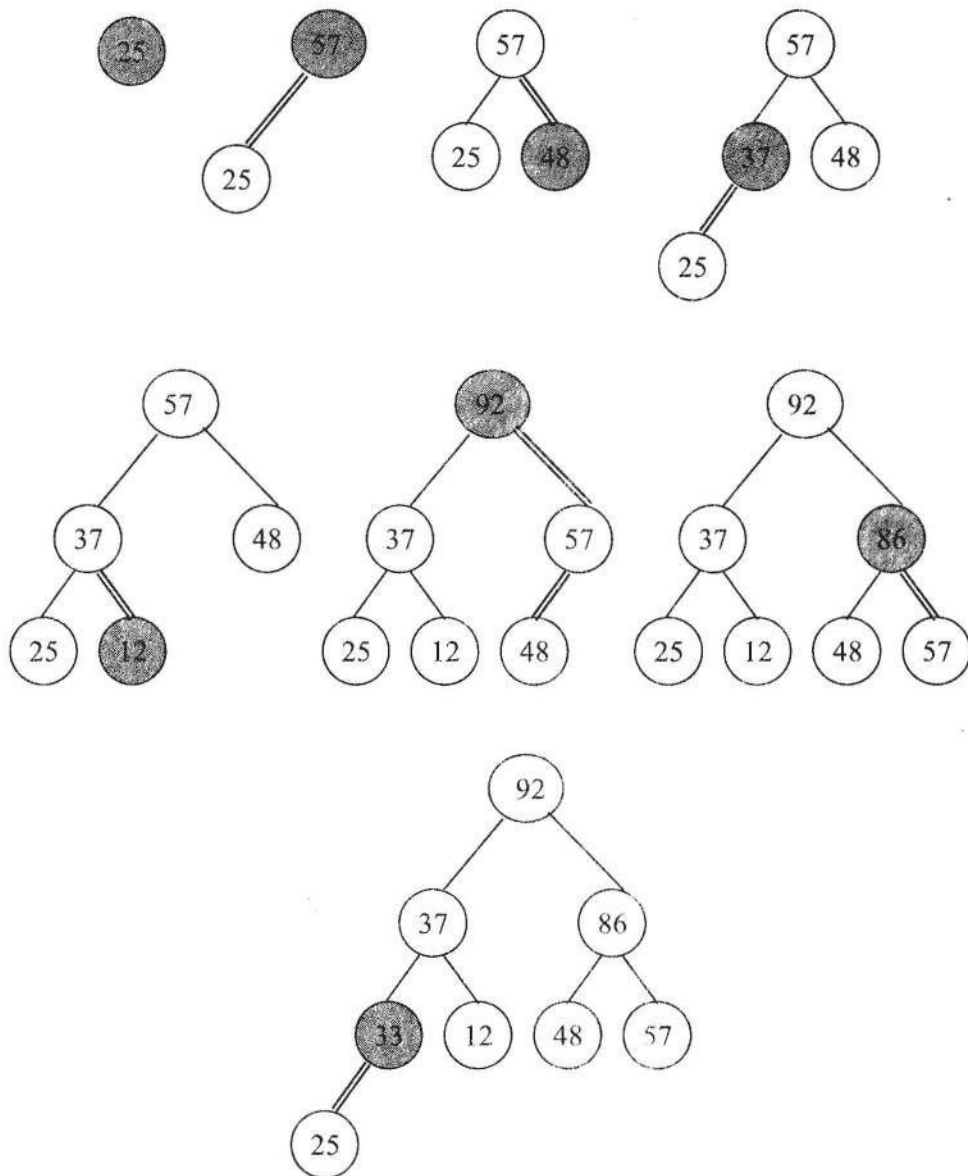


Diagram 10.25

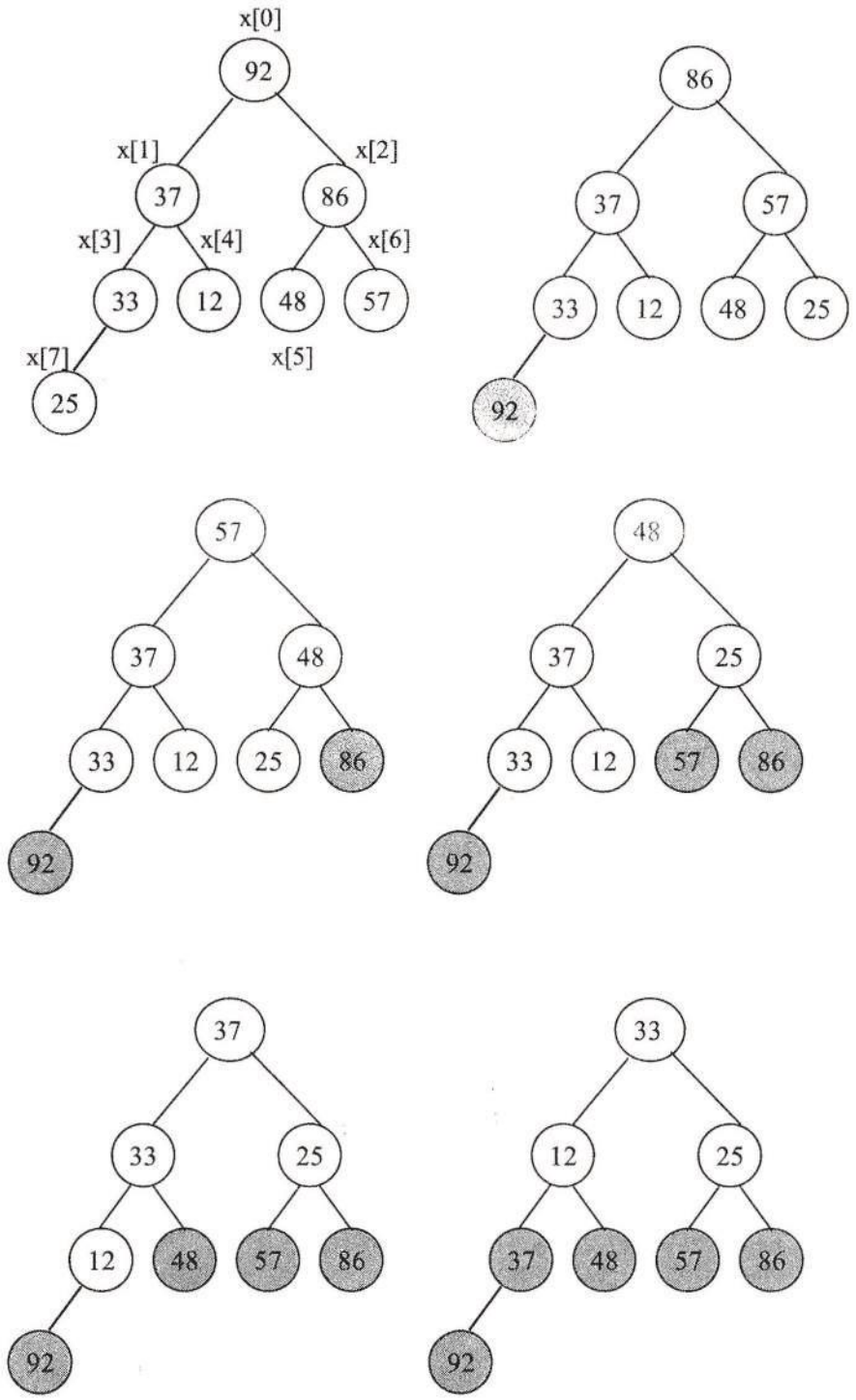


Diagram 10.26

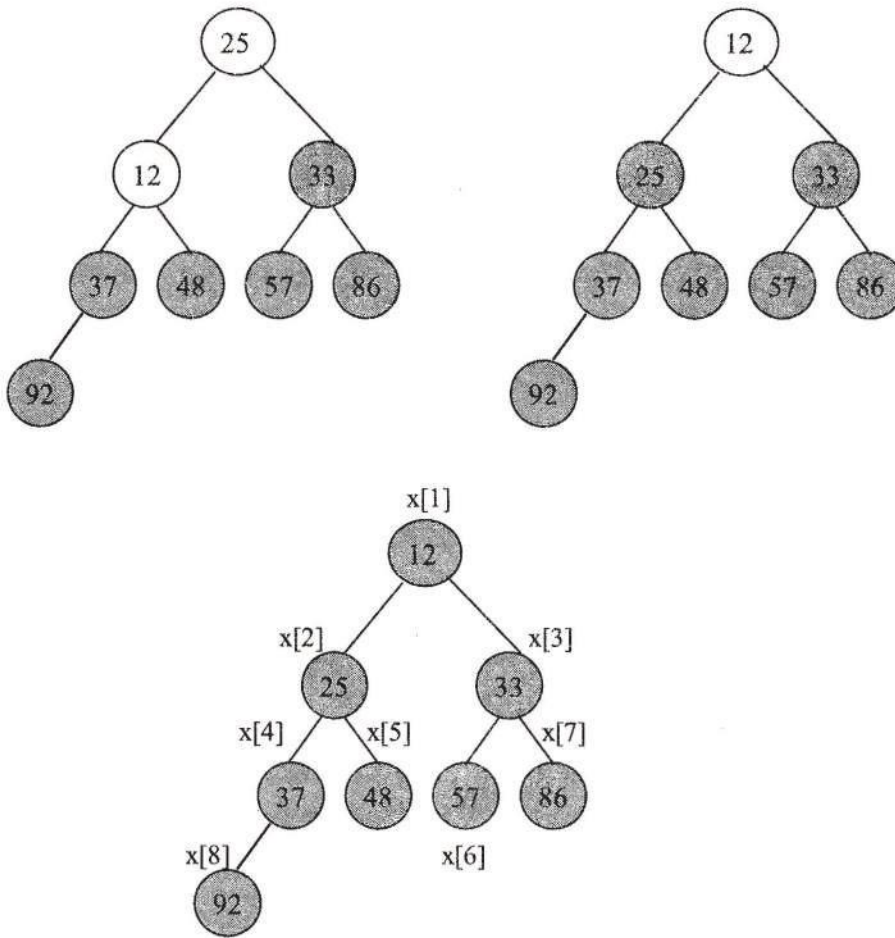


Diagram 10.26 (cont.)

10.3.5. TWO-WAY MERGE SORT

Merging is the process of combining two or more sorted files into a third sorted file.

Divide the file into n subfiles of size 1 and merge adjacent pairs of files. Then there will be $n/2$ files of size 2. Repeat this process until there is only one file remaining of size n . Diagram 4.27 illustrates this process on a sample file. Each individual file is contained boxes.

A routine to implement the merge sort is described as follows: an auxiliary array aux of size n is required to hold the results of merging two subarrays of x. The variable size contains the size of the subarrays being merged. Since at any time the two files being merged are both subarrays of x, lower and upper bounds are required to indicate the subfiles of x being merged.

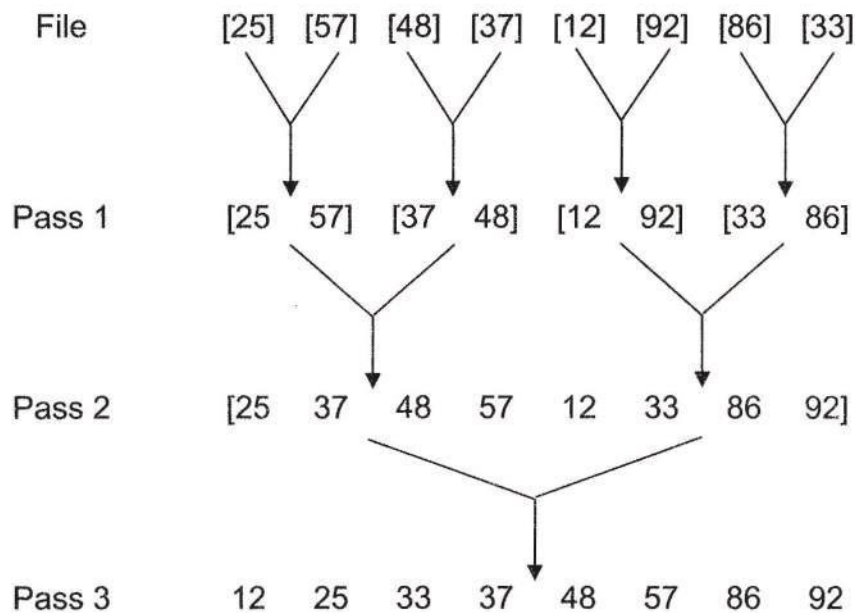


Diagram 10. 27

l1 and u1 represent the lower and upper bounds of the first file, and l2 and u2 represent the lower and upper bounds of the second file, respectively. i and j are used to reference elements of the source files being merged, and k indexes the destination file aux. The routine follows:

```
#define NUMELTS . . .

void mergesort (int x[ ], int n)
{
    int aux[NUMELTS], i, j, k, l1, l2, size, u1, u2;

    size = 1; /* merge files of size 1*/
    while (size < n) {
```

```

l1 = 0;
k = 0;
while (l1+size < n) {
    l2 = l1+size;
    u1 = l2-1;
    u2 = (l2+size-1 < n) ? l2+size-1 : n-1;
    /* proceed through the two subfiles */
    for (i = l1, j = l2; i <= u1 && j <=u2; k++)
        /* enter smaller into the array aux */
        if (x[i] <= x[j])
            aux[k] = x[i++];
        else
            aux[k] = x[j++];
    /* one of the files exhausted */
    for (; i <= u1; k++)
        aux[k] = x[i++];
    for (; j <= u2; k++)
        aux[k] = x[j++];
    l1 = u2+1;
}
for (i = l1; k < n; i++)
    aux[k++] = x[i];
for (i = 0; i < n; i++)
    x[i] = aux[i];
size *= 2;
}
}

```

There are obviously no more than $\log_2 n$ passes in mergesort, each involving n or fewer comparisons. Thus, mergesort requires no more than $n \cdot \log_2 n$ comparisons. But mergesort requires $O(n)$ additional space for the auxiliary array.

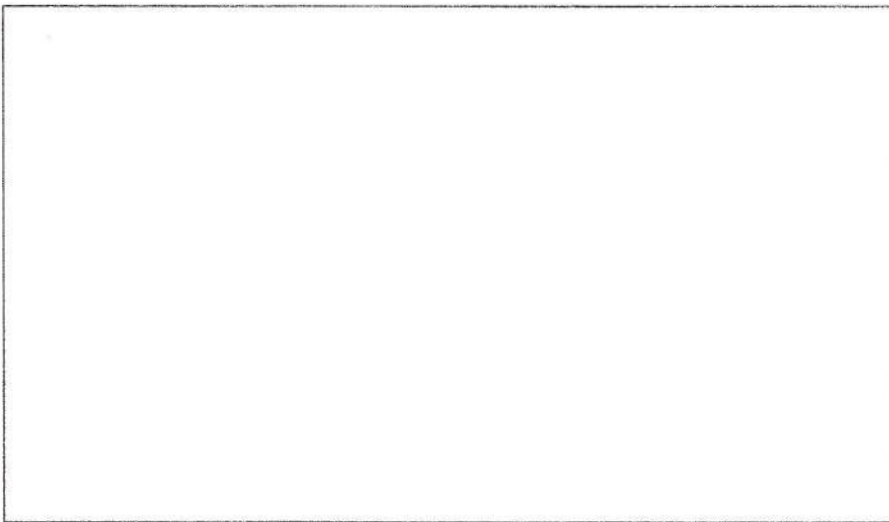
Comparison Of Sorting Algorithms:

Algorithm	Average	Worst case	Space usage
Bubble sort	$n^2 / 4$	$n^2 / 2$	In place
Selection sort	$n^2 / 4$	$n^2 / 4$	In place
Insertion sort	$n^2 / 4$	$n^2 / 4$	In place
Quicksort	$O(n \log_2 n)$	$n^2 / 2$	Extra $\log_2 n$ entries
Heapsort	$O(n \log_2 n)$	$O(n \log_2 n)$	In place
Merge sort	$O(n \log_2 n)$	$O(n \log_2 n)$	Extra n entries

Table 10.2

Learning Activity 10.3

- ◆ Write your answer in the space given below.
 - ◆ Check your answer with the one given at the end of the unit.
1. Sort the array of 10 elements 42, 23, 74, 11, 65, 58, 94, 36, 99, 87 using the sorting methods
- a) Bubble sort
 - b) Insertion sort
 - c) Selection sort
 - d) Quicksort
 - e) Heapsort
 - f) Merge sort



10.4. FILE ORGANIZATIONS

Most operating systems provide a set of basic file organizations that are popular with the users of the system. The three most common types of file organizations are

- sequential
- indexed sequential
- direct

10.4.1. SEQUENTIAL ORGANIZATION

In a sequential file, records are stored one after the other on a storage device. The sequential allocation is simple, and flexible enough to handle large volumes of data, a sequential file has been the most popular basic file structure used in the data-processing industry.

All types of external storage devices support a sequential file organization. Some devices, by their physical nature, can only support sequential files.

The records are maintained in the logical sequence of their primary key values. Accessing a particular record requires the accessing of all previous records in the file. Search for a given record in sequential file requires, on average, access to half the records in the file.

Each time a read or write operation is executed for a particular storage device, a block of logical records is transferred. The apparent difference in a program's read and write statements and read and write commands issued for a particular device is resolved by using a buffer between external storage and the data area of a program.

A buffer is a section of main memory which is equal in size to the maximum size of a block of logical records used by a program. The data-management routines of the operating system use buffers for the "blocking" and "deblocking" of records.

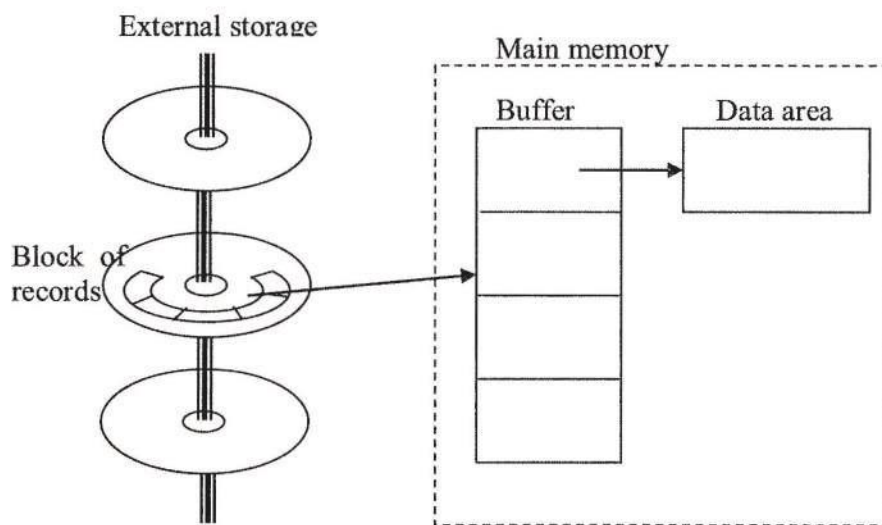


Diagram 10.28

Blocking and deblocking can be illustrated as follows: when the first read statement is executed on a sequential file, a block of records is moved from external storage to a buffer. The

first record in the block is then transferred to the program's data area. For each subsequent execution of a read statement, the next successive record in the buffer is transferred to the data area.

Only after every record in the buffer has been moved to the data area, in response to read statements, does the next read statement cause another block to be transferred to the buffer from external storage. The new records in the buffer are moved to the data area, as described previously, and this entire process is repeated for each block that is read.

In a similar fashion, write statements cause transfer of program data to the buffer. When the buffer becomes full, then the block is written on the external storage device immediately after the preceding block of records.

The buffering technique described above is called single buffering. Multiple buffering makes use of a queue of buffers which are normally controlled by the operating system.

The important points concerning the sequential processing of sequential files can be summarized as follows:

1. Sequential processing is most advantageous if a large number of transactions can be batched to form a single "run" on the file.
2. A new file should be created if there are any additions and a significant number of deletions requested.
3. Quick response time should not be expected for a transaction or a batch or transactions.
4. The requirement that the records in a sequential file be ordered by a particular key is not essential if the file is

being scanned to perform the same operation on every record.

10.4.2. INDEXED SEQUENTIAL ORGANIZATION

To improve the quick response time of a sequential file, a type of indexing technique can be added. An index is a set of <key, address> pairs. Indexes created from a sequential set of primary keys are referred to as indexed sequential. The term index file describes the indexes and data file to refer to the data records.

The index provides for random access to records, while the sequential nature of file provides easy access to the subsequent records as well as sequential processing. An additional feature of this file system is the overflow area.

Sequential index	Block at address	No of comparisons	Record key	No of comparisons	
K_s	A_1	1	K_1	1	(2)
			K_2	2	(3)
			\vdots		
			K_s	s	(s+1)
K_{2s}	A_2	2	K_{s+1}	1	(3)
			K_{s+2}	2	(4)
			\vdots		
			K_{s1+s2}		
			\vdots		
K_n	A_m	m	\vdots		
			K_n	s	(s+m)

Diagram 10.29

If the index file contains only the address part of <key, address> pair, it is called an *implicit index* else it is called an *explicit index*. In a limit indexing or partial indexing scheme, a single entry per track is maintained in the index. A number of memory locations are grouped together and can be identified by a single address.

Considering a set of sorted keys $\langle K_1, K_2, \dots, K_n \rangle$, with $K_1 < K_2 < \dots < K_n$, divided into m groups of sizes $\langle s_1, s_2, \dots, s_n \rangle$ with the sorted order of the keys within each group. Each group is identified by the key with the largest value in that group and called the *sequential index key*. Diagram 10.29 illustrates this.

It is also possible to create a hierarchy of indexes with the lowest level index pointing to records, while the higher level indexes point to the indexes below them (Diagram 10.30).

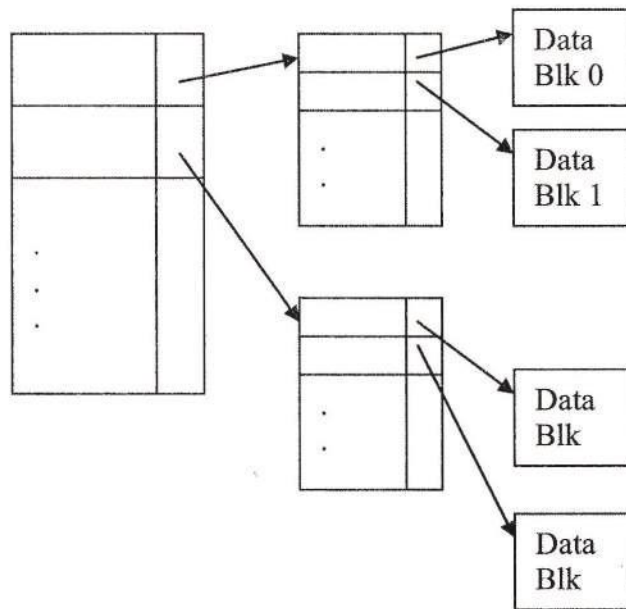


Diagram 10.30

An indexed sequential file is made up of the following components:

1. A primary data storage area contains the records written by the users programs.
2. Overflow area(s) permits the addition of records to the files.
3. A hierarchy of indices. In a random enquiry update, the physical address is obtained using indices.

When using a disk device to store index-sequential files, the data is stored on cylinders, each of which is made up of a number of tracks. Each track index entry is made up of the following items:

1. The address of the prime data track to which the entry refers.
2. The highest key of a record in the prime data track.
3. The highest key of a logical record in that data track, including records in the overflow areas.
4. The address of a record with the lowest key in the overflow area associated with that track.

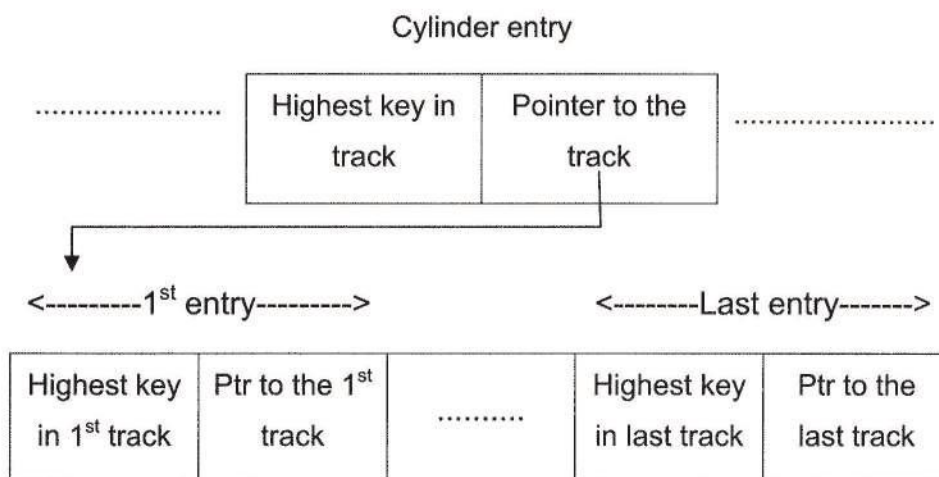


Diagram 10.31

The important properties relating to indexed sequential files are:

1. Indexed sequential files provide reasonably fast access to records using either sequential or direct processing.
2. For relatively static files, the independent overflow area can be eliminated.
3. For highly volatile files, the access time for a record becomes excessive as overflow areas become filled.
4. The housekeeping details for indexed sequential files are generally provided in most systems.

4.4.3. DIRECT ORGANIZATION

In direct file organizations, the key value is mapped directly to the storage location by hashing. The hash function h maps the key value k to the value $h(k)$, which is used as an address.

The hash function must map the key values uniformly and it should not map many different key values to a single address. There are innumerable ways of converting a key to a numeric value. The important properties related to direct files are:

1. Direct access to records in a direct file is rapid.
2. The space utilization for a direct file is poor compared to other file organizations.
3. The performance attained using a direct file is very dependent upon the key-to-address transformation algorithm adopted.

- Records can be accessed serially but not sequentially unless separate ordered list of keys is maintained.

Learning Activity 10.4

- ◆ Write your answer in the space given below.
- ◆ Check your answer with the one given at the end of the unit.

- Compare the three file organizations.

Let us sum up

The various tree structures and their representations are discussed. The algorithms for the operations on trees are implemented in C. A number of searching and sorting algorithms are coded in C, and they are analyzed based on the running time and additional space is made. The main file organizations sequential, index sequential and direct are explained with their advantages and disadvantages.

ANSWERS TO LEARNING ACTIVITIES

Learning Activity 10.1

- Leaves – D, K, L, F, I, J
 - Root – A
 - Father of node C – A
 - Sons of C – F, G
 - Ancestors of E – A, B

- f) Descendants of E – H, K, L
- g) Right brothers of D and E – E, None
- h) Nodes to the left and to the right of G – F and none
- i) Depth of node C – 2
- j) Height of node C – 1

2. The three tree traversals of the given binary tree is

Preorder: ABDEHKL CFGIJ

Inorder: DKLHEBFIJGCA

Postorder: DBKHLEAFCIGJ

3. This can be proved by induction. A binary tree with no internal nodes has one external node, so the property holds for $N = 0$. For $N > 0$, any binary tree with N internal nodes has k internal nodes in its left subtree and $N - 1 - k$ internal nodes its right subtree for some k between 0 and $N-1$, since the root is an internal node. By the inductive hypothesis, the left subtree has $k + 1$ external nodes and the right subtree has $N - k$ external nodes, for a total of $N + 1$.

4. a) $A * (B + C * (D + E))$

b) $A * (B + C) * D + E$

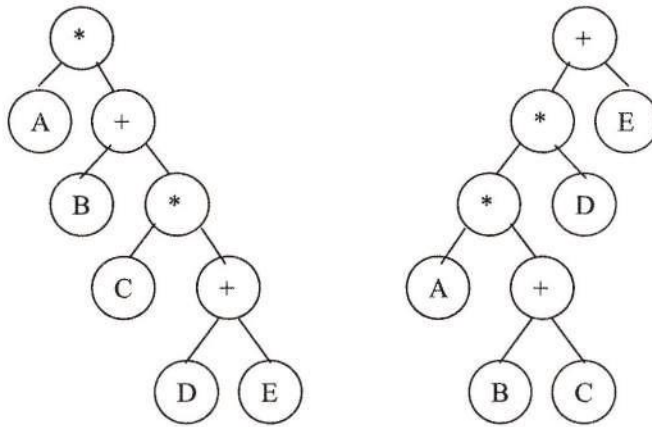


Diagram 4.32

Learning Activity 10.2

1. The sequential search algorithm to add a record *rec* with key *key* to the table if key is not already there as follows:

```

for (i = 0; i < n; i++)
    if (key == k[i])
        return (i);
k(n) = key;
r(n) = rec;
n++;
return (n-1);

```

2. An algorithm for sequential search in a sorted table is

```

for ( i = 0; i < n && key <= k(i); i++ )
    if (key == k(i))
        return (i);
return (-1);

```

3. The binary trace for $x = 90$ and 500 in the following array of elements. 23, 76, 90, 112, 371, 405, 450, 537, 620, 777

Search for 90				Search for 500			
Iteration	L	H	M	Iteration	L	H	M
1	1	10	5	1	1	10	5

2	1	4	2	2	6	10	8
3	3	4	3	3	6	7	6
				4	7	7	7
				5	8	7	

Learning Activity 10.3

1. Sort the array of 10 elements 42, 23, 74, 11, 65, 58, 94, 36, 99, 87 using the sorting methods

a) Bubble sort

0		42	23	74	11	65	58	94	36	99	87
1		11	42	23	74	36	65	58	94	87	99
2		11	23	42	36	74	58	65	87	94	99
3		11	23	36	42	58	74	65	87	94	99
4		11	23	36	42	58	65	74	87	94	99

b) Selection sort

0		42	23	74	11	65	58	94	36	99	87
1		11	23	74	42	65	58	94	36	99	87
2		11	23	74	42	65	58	94	36	99	87
3		11	23	36	42	65	58	94	74	99	87
4		11	23	36	42	65	58	94	74	99	87
5		11	23	36	42	58	65	94	74	99	87
6		11	23	36	42	58	65	74	94	99	87
7		11	23	36	42	58	65	74	87	94	99
8		11	23	36	42	58	65	74	87	94	99

c) Insertion sort

0	42	23	74	11	65	58	94	36	99	87
1	42	23	74	11	65	58	94	36	99	87
2	23	42	74	11	65	58	94	36	99	87
3	23	42	74	11	65	58	94	36	99	87
4	11	23	42	74	65	58	94	36	99	87
5	11	23	42	65	74	58	94	36	99	87
6	11	23	42	58	65	74	94	36	99	87
7	11	23	42	58	65	74	94	36	99	87
8	11	23	36	42	58	65	74	94	99	87
9	11	23	36	42	58	65	74	94	99	87
10	11	23	36	42	58	65	74	87	94	99

d) Quicksort

0	42	23	74	11	65	58	94	36	99	87
1	11	23	36	42	65	58	94	74	99	87
2	11	23	36	42	65	58	94	74	99	87
3	11	23	36	42	65	58	94	74	99	87
4	11	23	36	42	58	65	94	74	99	87
5	11	23	36	42	58	65	94	74	99	87
6	11	23	36	42	58	65	87	74	94	99

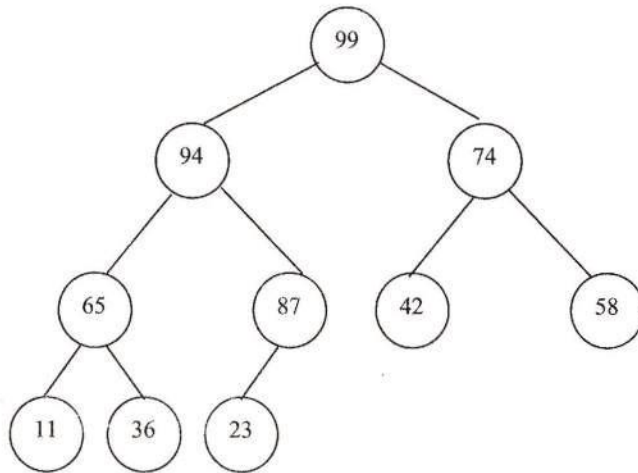
7 11 23 36 42 58 65 74 87 94 99

8 11 23 36 42 58 65 74 87 94 99

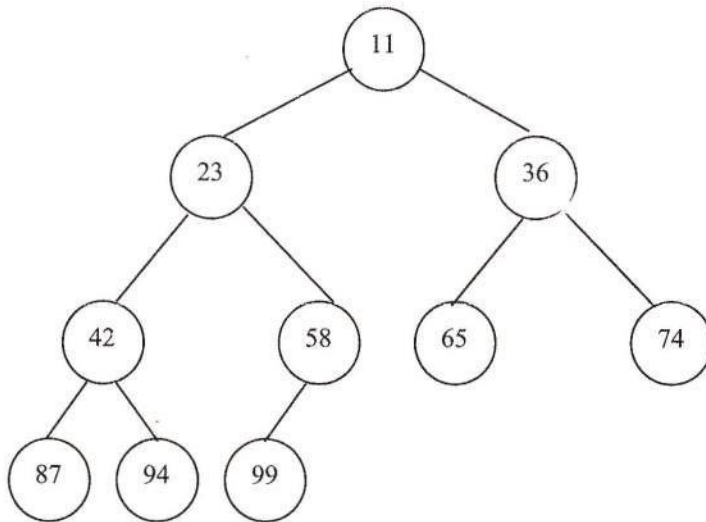
9 11 23 36 42 58 65 74 87 94 99

e) Heapsort

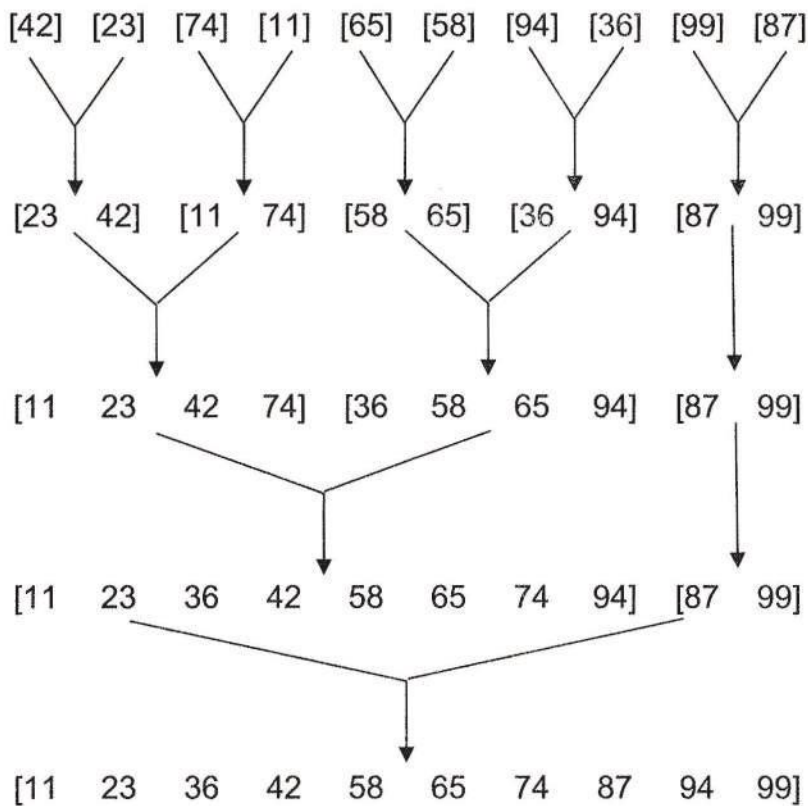
The heap created with all 10 elements:



After sorting:



f) Merge sort



Learning Activity 10.4

1. Comparison of three file organizations:

	Sequential	Indexed sequential	Direct
Sequential access	Suitable	Suitable	Not suitable
Random access	Not suitable	Possible	Suitable
Record insertion	Creation of new file	Requires overflow area	Possible
	Sequential	Indexed sequential	Direct

Deletion	Creation of new file	Marking	Possible
Update	Creation of new file	Possible	Possible
Overhead	None	Index file	Buckets

Model Questions

1. Explain in detail about sequential files.
2. Differentiate Direct and Indexed files.
3. Show the implementation of Collision Management in Direct files.

References

1. An Introduction to Data Structures with Applications by Tremblay, J.P. and Sorenson, P.G.
2. Nicklaus Wirth, "Algorithms and Data Structures – Programmes" Prentice Hall of India Pvt. Ltd., New Delhi, 2002.
3. Y.Langesam, M.J. Augenstein and A.M. Tenenbaum "Data Structures using C and C++" II edition, Pearson Education, New Delhi, 2002.