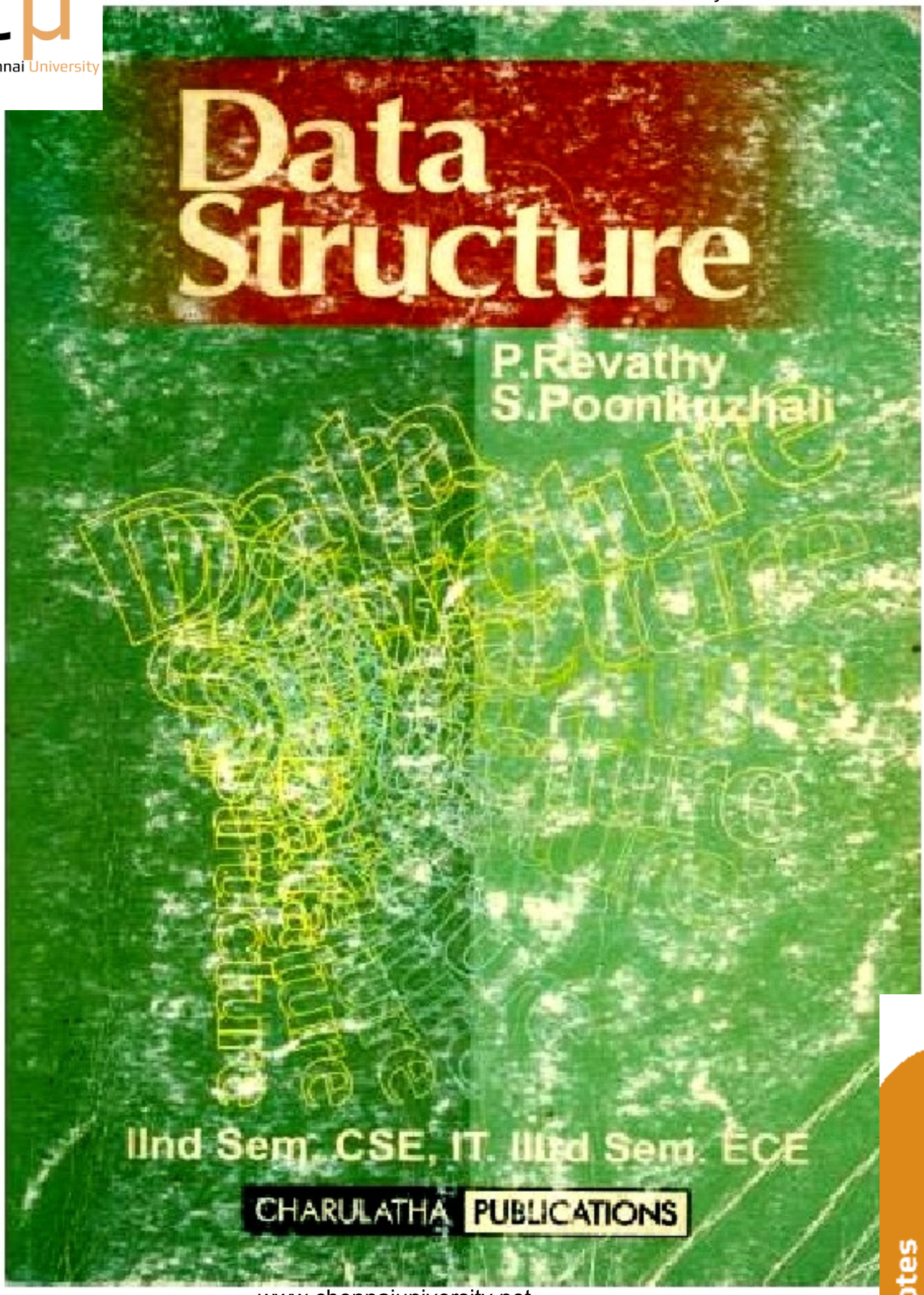


DATA STRUCTURES

Designed as per

ANNA UNIVERSITY Syllabus

For

B.E., IISem. CSE,IT & III Sem. ECE

P. REVATHY

S. POONKUZHALI

Department of CSE

RAJALAKSHIMI ENGINEERING COLLEGE

CHENNAI

CHARULATHA PUBLICATIONS

New No. 24, Thambiah Road
West Mambalam, Chennai - 600 033.
Phone : 24745589, 24746546
Email : charulathapublications@yahoo.com

Acknowledgement

We take this opportunity to express our deep gratitude to our Correspondent ***Thiru. S.Meganathan***, chairperson ***Dr. Thangam Meganathan***, Principal ***Dr K. Sarukesi***, Dean ***Prof. V. N.Srinivasan*** and Head of the Department of Computer Science ***Ms. S. Pramila*** for their inspiration and encouragement in bringing out this book successfully.

Many of our colleagues have read various portions of the manuscript and have given us valuable comments and advice. In particular we would like to thank ***Ms. J. Ida Christy***, ***Ms. N. Sunitha*** and ***Ms. Dhalphena Phemila***.

Of course, we would have nothing to write about without the many people who did their original research that provided the material we enjoy learning and passing on to new generation of students. We thank them for their work.

We would like to extend our thanks to ***Mr. M.R. Bharathi*** of Charulatha Publications.

We would like to give our warmest thanks to ***Mr. N. Mahesh Kumar*** for his expert assistance in helping to prepare the manuscript for typesetting and Charulatha Publications in bringing out this book.

Finally, we would like to thank our family members who had been a constant source of moral support throughout this incredibly project.

Suggestions and comments for further improvement of this book are most welcome. You may please email us at

prevathy@hotmail.com

kuzhal_s@yahoo.co.in

P. Revathy

S. Poonkuzhali

Chennai University

CS1151 - DATA STRUCTURES

AIM

To provide an in-depth knowledge in problem solving techniques and data structures.

OBJECTIVES

- To learn the systematic way of solving problems
- To understand the different methods of organizing large amounts of data
- To learn to program in C
- To efficiently implement the different data structures
- To efficiently implement solutions for specific problems

UNIT I - PROBLEM SOLVING

Problem solving – Top-down Design – Implementation – Verification – Efficiency – Analysis – Sample algorithms.

UNIT II - LISTS, STACKS AND QUEUES

Abstract Data Type (ADT) – The List ADT – The Stack ADT – The Queue ADT

UNIT III - TREES

Preliminaries – Binary Trees – The Search Tree ADT – Binary Search Trees – AVL Trees – Tree Traversals – Hashing – General Idea – Hash Function – Separate Chaining – Open Addressing – Linear Probing – Priority Queues (Heaps) – Model – Simple implementations – Binary Heap

UNIT IV - SORTING

Preliminaries – Insertion Sort – Shellsort – Heapsort – Mergesort – Quicksort – External Sorting

UNIT V - GRAPHS

Definitions – Topological Sort – Shortest-Path Algorithms – Unweighted Shortest Paths – Dijkstra's Algorithm – Minimum Spanning Tree – Prim's Algorithm – Applications of Depth-First Search – Undirected Graphs – Biconnectivity – Introduction to NP-Completeness

TUTORIAL

TEXT BOOKS

1. R. G. Dromey, "How to Solve it by Computer" (Chaps 1-2), Prentice-Hall of India, 2002.
2. M. A. Weiss, "Data Structures and Algorithm Analysis in C", 2nd ed, Pearson Education Asia, 2002. (chaps 3, 4.1-4.4 (except 4.3.6), 4.6, 5.1-5.4.1, 6.1-6.3.3, 7.1-7.7 (except 7.2.2, 7.4.1, 7.5.1, 7.6.1, 7.7.5, 7.7.6), 7.11, 9.1-9.3.2, 9.5-9.5.1, 9.6-9.6.2, 9.7)

REFERENCES

1. Y. Langsam, M. J. Augenstein and A. M. Tenenbaum, "Data Structures using C", Pearson Education Asia, 2004
2. Richard F. Gilberg, Behrouz A. Forouzan, "Data Structures – A Pseudocode Approach with C", Thomson Brooks / COLE, 1998.
3. Aho, J. E. Hopcroft and J. D. Ullman, "Data Structures and Algorithms", Pearson education Asia, 1983.

Notes

TABLE OF CONTENTS

Chennai University

GRAPH

Notes

Chennai University

1 PROBLEM SOLVING

1.1 INTRODUCTION

Computer problem solving is an iteractive process requiring much thought, careful planning, logical precision, persistance and attention to detail. At the same time, it can be challenging, exciting and satisfying experience with personal creativity.

PROGRAMS AND ALGORITHM

A program is a set of explicit and unambiguous instructions expressed in a programming language. It takes the input from the end user and manipulate it according to its instructions and produces an output which represents the 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.

1.2 PROBLEM SOLVING ASPECT

The problem solving is recognized as a creative process which largely defines systematization and mechanization.

The various aspects of problem solving are :

1. Problem definition phase
2. Getting started on a problem
3. The use of specific examples.
4. Simulation among problems.
5. Working backwards from the solution.
6. General problem - solving strategies.

1.2.1 Problem definition phase

The problem definition phase deals with "What must be done rather than how to do it". That is, the user tries to extract a set of precisely defined tasks from the problem statement.

1.2.2 Getting Started on a Problem

There are many ways to solve most problems and also many solutions to most problems, which makes the job of problem solving difficult to recognize quickly which paths are likely to be fruitless or productive. Hence the programmer do not have any idea where to start on the problem, even after the problem definition phase. They became concerned with details of the implementation before they have completely understood or worked out an implementation (i.e.,) independent solution. So it is better not to be concerned about detail in the beginning of the problem solving phase itself.

list be used to formulate the problem ?

5. Will it reduce the amount of computation to find the intermediate results, which is based on the way in which it is arranged.

1.3.3 Construction of loops

To construct any loop, we must take three things in account. They are

- ★ Initial conditions - that apply before the loop begins to execute
- ★ Invariant relation - that apply after each iteration of the loop.
- ★ Termination condition - under which the iterative process must terminate.

1.3.4 Establishing initial conditions for loops

To establish the initial conditions for a loop set the loop variables to the values that assumes to solve the smallest problem associated with the loop. For example, to find the sum of a set of numbers in an array using an iterative construct, the loop variables are *i* the array and loop index, and *s* the variable for accumulating the sum of array elements.

The smallest problem in this case is to find sum of zero numbers which is zero and so the initial values of *i* and *s* must be zero.

1.3.5 Finding the iterative construct

After finding the smallest problem, the next step is to extend it to the next smallest problem. i.e., to get the solution for i = 1, we build on the solution to the problem for i = 0. Similarly from $(i - 1)^{th}$ case, we get the solution for i^{th} case. (where $i \geq 1$).

Example

```
i : = 0
s : = 0
while i < n do    // 'n' represents number of iterations
begin
        i : = i + 1;
        s : = s + a[i]
end
```

1.3.6 Termination of loops

The loops can be terminated in a number of ways. In general the termination conditions a dictated by the nature of the problem. The simplest condition for terminating a loop occurs wh the number of iterations to be made is known in advance. For example, In Pascal the for-loop c be used as

```
for i : = 1 to n do
  begin
  .....
```

.....

end

This loop terminates unconditionally after n iterations.

The while-loop in Pascal can be used as

```
while (x > 10) do
  begin
    ........
    ........
  end
```

This loop terminates when conditional expression becomes false.

1.4 IMPLEMENTATION OF ALGORITHMS

The implementation of an algorithm that has been properly designed in a top-down fashion should be an almost mechanical process. If an algorithm has been properly designed the path of execution should flow in a straight line from top to bottom which is much easier to understand and debug.

1.4.1 Use of procedures to emphasise modularity

Modularization of program assist for both the development of the implementation and the readability of the main program. This allows to implement a set of independent procedures to perform specific and well defined tasks. In applying modularization in an implementation, the process is not taken too far, to a point at which the implementation again becomes difficult to read because of fragmentation.

1.4.2 Choice of variable names

Choosing appropriate variable and constant names makes programs more meaningful and easier to understand. Each variable should only have one role in a given program.

1.4.3 Documentation of programs

Another useful documenting practice that can be employed is to associate a brief but accurate comment with each procedure. A good programming practice is always to write programs so that they can be executed and used by other people unfamiliar about the program. This means that the program must specify during execution exactly what responses it requires from the user.

1.4.4 Debugging programs

- In implementing an algorithm it is very important to carry out a number of tests to ensure that the program is behaving correctly according to its specifications.
- To make the task of detecting logical errors, a set of statements that will print the information at strategic points in the computation is build into the program.
- The simplest way to implement this debugging tool is to have a Boolean variable (eg. debug) which is set to true when the debugging output for the program is required.

ebugging output can then be parenthesized in the following way:

```
if debug then
  begin
    writeln (....)
    .....
    .....
  end
```

- A good rule to end follow when debugging is not to assume anything.

1.4.5 Program testing

In attempting to test whether or not a program will handle all variations of the problem it was designed to solve that it will cope with the limiting and unusual cases. We might check whether the program solves the smallest problem, whether it handles the case when all data values are the same and so on. Wherever possible, input and output assertions should be accompanied to the programs. We have to build the program that informatively respond to the user when it receives input conditions it was not designed to handle.

A good rule to follow is that fixed numeric constants should only be used in programs for things like the number of months in a year and so on.

1.5 PROGRAM VERIFICATION

The cost of software development is extremely high. Also, it may cause serious changes on sensitive data if the program doesn't work correctly. Therefore, it is essential to verify whether the program works correctly or not.

A better method of program verification is to verify the individual modules as it is completed rather than postponing the verification process to the end.(i.e.,) after integrating the modules.

The various phases in the program verifications are

1. Input and output assertion
2. Computer model for program execution
3. Implications and symbolic executions.
4. Verification of straight-line program segments
5. Verification of program segments with branches
6. Verification of program segments with loops
7. Verification of program segments that employ arrays
8. Proof of termination.

1.5.1 Input and Output assertions

The input and output assertion is the first and foremost step that has to be taken inorder to prov correctness of a program. It provides a formal statement which consists of specifications int

bles.

statements gives two assertions

(a) An input assertion

(b) An output assertion

Input assertion

It specifies any constraints that are placed on the value of the input variables used by the program.

Example

In the division operation let us consider 'd' as the divisor. Hence it is clear that d cannot have the value 0. The input assertion is therefore $d \neq 0$.

Output assertion

It specifies the result symbolically that the program is expected to produce for the input data which satisfies the input assertion.

Example

In calculating the quotient q and the remainders x from the division of 'a' by 'b', then the output assertion is as $(a = q * b + r) \wedge (r < b)$

where '^' represents the logical AND operation.

1.5.2 Compute Model for program execution

A program may have a variety of execution paths leading to successful termination. For a given set of input conditions only one of these paths will be followed. The progress of a computation from specific input conditions through to termination can be thought of as a sequence of transitions from one computation state to another. Each state is defined by the values of all variables at the corresponding point in time.

A state transition model for program execution provides a foundation on which to construct correctness proof of algorithms.

1.5.3 Implications and Symbolic execution

Verifying a program can be formulated as a set of implications which must be shown to be logically true.

The general form of implication is

$$P \supset Q$$

P - Assumption

Q - Conclusion

P	Q	P ⊃ Q
True	True	True
True	False	False
False	True	True
False	False	True

Fig. 1.5(a) Truth Table defining implication

In order to show that these implications or propositions are logical true, it is necessary to use the technique of symbolic execution.

In symbolic execution, all input data values are replaced by symbolic values and all arithmetic operations on numbers are translated into algebraic manipulation of symbolic expression. This enables us to transform the verification procedure into providing that the input assertion with symbolic values substituted for all input variables implies the output assertion with final symbolic values substituted for all variables.

Step	Normal Execution	Symbolic Execution
	Input values : x = 3, y = 1	Input values : $x = \alpha$, $y = \beta$
1.	X : = x - y ⇒ x = 3 - 1 = 2	X : = x - y ⇒ $x = \alpha - \beta$
2.	Y : = x + y ⇒ y = 2 + 1 = 3	Y : = x + y ⇒ $y = (\alpha - \beta) + \beta = \alpha$
3.	X : = y - x ⇒ x = 3 - 2 = 1	X : = y - x ⇒ $x = (((\alpha - \beta) + \beta) - (\alpha - \beta)) = \beta$.

Fig. 1.5(6) Normal and Symbolic execution for exchange mechanism.

Consider the following program segment labelled with input and output assertions :

```
A    readln (x, y);
     {assert ; true)
     x : = x - y;
     y : = x + y;
     x : = y - x;
B    {assert  x = yo ∧ y = xo }
     where xo and yo refer to the initial values of x and y respectively.
```

Fig. 1.5(b) Exchange Mechanism program segment

cation of Straight - line program segments

Let us consider the exchange mechanism program segment.

The verification condition is ;

VC (A - B) : true $\supset \{ x = yo \wedge y = xo \}$ on substitution of initial and final values of all variables, we get :

$$VC\ (A - B) : \text{true} \supset ((\alpha - \beta) + \beta) - (\alpha - \beta))$$

$$= \beta \wedge ((\alpha - \beta) + \beta) = \infty$$

The conclusion part of the verification condition an be simplified to yield $\beta = \beta$ and $\alpha = \alpha$ which is true and so the implication is true.

[Note : VC (A - B) refers to the verification condition over the program segment A to B. Therefore apply the resultant of the program A to the program segment B.

(i.e) Symbolic execution of A is resulted as

$x = \alpha, y = \beta$ [initial values]

$x = ((\alpha - \beta) + \beta) - (\alpha - \beta)$ [Final Value]

$y = (\alpha - \beta) + \beta)$ [Final Value]

program segment B is $\{ x = yo \wedge y = xo \}$.

1.5.5 Verification of program segments with branches

To handle program segments that contain branches, it is necessary to set up and prove verification condition for each branch seperately. Let us consider the following program segment that ensures x is less than or equal to y.

```
        read ln (x, y)
A       {assert P_A : true}
        if x > y then
        begin
        t : = x;
        x : = y;
        y : = t;
        end.
B       {assert P_B : ((x < = y) ∧ (x = xo ∧ y = yo)) ∨ ( x = yo ∧ y = xo);
```

In general, the propositions that must be proved for the basic if construct are $P_A \wedge C_A \supset P_B$

$P_A \wedge \sim C_A \supset P_B$ where C_A is the branch condition.

1.5.6 Verification of Program segments with loops

Problems using program segments with loops can be solved with a loop invariant.

A Loop invariant is a property that captures the progressive computational role of the loop while at the same time remaining true, before and after each loop traversal irrespective of how many times the loop is executed. Let us consider the following single - loop program structure as model.

A {input assertion P_A}

```
[ --
  --
  Straight line program segment
  --
  -- ]
```

B {loop invariant I_B}

while loop - condition do

begin

```
[ --
  --
  Loop-free program segment
  --
  -- ]
```

end

C {Output assertion P_C}

The steps to be employed to verify a loop segment are.

Step 1: We have to show that the loop invariant is initially true. This can be done by setting verification condition,

VC (A - B) for the program segment from A to B.

Step 2 : To show that the loop invariant is still true after the segment of program within the loo been executed. To do this, set up a verification condition as VC (B - B).

loop invariant with initial values of variables set, together with condition for loop execution C_B, implies the truth of the loop invariant with final values of variables.

(i.e) $I_B \wedge C_B \supset I_B$.

Step 3 : The final step is verifying a loop segment, it is necessary to show that, the loop invariant together with the negation of the loop entry condition, implies the assertion that applies on existing form of the loop. The verification condition for this care is VC (B - C). The corresponding proposition will be :

$$I_B \wedge \sim C_B \supset P_C$$

1.5.7 Verfication of program segments that employ arrays

The idea of symbolic execution can be extended to simpler examples that employ arrays by accounting symbolic values of all array elements.

Let us consider the program segment which finds the position of the smallest element in the array.

A {assert $P_A : n \geq 1$}

```
i := 1;
p := 1;
```

B $\{in\,var\,iant\ I_B : (1 \leq i \leq n) \wedge (1 \leq p \leq i) \wedge (a[p] \leq a[1], a[2], ..., a[i])\}$

```
while i < n do
        begin
                i := i + 1;
                if a[i] < a [p] then p! = i
        end
```

C $\{assert\ P_C : (1 \leq p \leq n) \wedge n(a[p] \leq a[1], a[2], ..., a[n])\}$

Assume the initial values of a[1], a[2], ..., a[n] as $\alpha_1, \alpha_2, ...\alpha_n$ and the initial value of n as δ, then the symbolic execution to check the verification condition is given as :

$$VC(A\text{-}B) : \delta \geq 1 \supset (1 \leq 1 \leq \delta) \wedge (1 \leq 1 \leq 1) \wedge \alpha_1 \leq \alpha_1$$

1.5.8 Proof of termination

To prove that a program terminates, it is necessary to show that every loop in the program terminates in a finite number of steps. Consider, for example, the for - loop ;

```
for i = 1 to n do
begin
...
...
end
```

when n is positive and finite, the loop is guaranteed to terminate because, with each iteration, the number of steps to the termination condition being satisfied is reduced by at least one. This reduction can only be carried out a finite number of times and so the loop must terminate.

The proof of termination is much more subtle and elusive for loops which has any one of the following two situations.

* When there is no single variable that is monotonically progressing towards the termination condition.
* An arithmetic combination of two or more variables makes progress towards termination with each iteration.

The problem of proving loop termination can often be approached by associating another expression in addition to the loop invariant, with each loop.

The expression $\in$, should be chosen to be a function of the variables used in the loop. It should always be non - negative and the value is decreased with each loop iteration.

To verify the loop structure perform the following steps.

Step 1 : Show that the loop invariant I_B, together with condition for loop execution C_B, implies that the value of the expresion $\in$ is greater than zero.

(i.e)

$$TC1(B): I_B \wedge C_B \supset \in > 0$$

The condition $\in \geq 0$ becomes an invariant of the loop.

Step 2 : Show that the loop invariant I_B, together with the condition for loop execution, C_B, implies that the value $\in_0$ of the expression before execution is strictly greater than its value $\in$ after loop execution. (i.e) for a loop B $TC2(B-B): I_B \wedge C_B \supset (\in_0 > \in) \wedge (\in \geq 0)$

1.6 THE EFFICIENCY OF ALGORITHMS

The Efficiency considerations for algorithms are inherently tied in with the design, implemen and analysis of algorithms. The resources used by the algorithms are central processin (CPU) time and internal memory. So the efficiency of the algorithm lies with the economical of these resources.

Some of the suggestions to improve the efficiency of an algorithm in designing are

* Redundant computations
* Referencing array elements
* [illegible] due to late termination

* Early detection of desired output conditions.
* Trading storage for efficiency gains.

1.6.1 REDUNDANT COMPUTATONS

The effects of redundant computations are most serious when they are embedded within a loop that must be executed many times which utilizes unnecessary memory space. The most common mistake using loops is to repeatedly recalculate part of an expression, that remains constant throughout the entire execution phase of the loop. For example

```
S : = 0; K : = 2;
for i = 1 to n do
begin
        S : = k * k * k + i ;
end
```

The unnecessary multiplication can be removed by precomputing a constant K_3 before executing the loop.

```
S : = 0 ; K : = 2 ;
K3 : = k * k * k ;
for i = 1 to n do
begin
        S : = K3 + i ;
end
```

1.6.2 Referencing array elements

Let us consider for example, the two versions of an algorithm for finding the maximum.

Version (1)

```
for i = 2 to n do
    if a [i] > a [p] then
        max : = a [p]
```

version (2)

```
max : = a[1];
for i = 2 to n do
    if a[i] > max then
        max : = a[i];
```

The version (2) implementation would normally preferred because the condition test a[i] > max is more efficient because it uses the variable max which requires only one memory reference instruction, whereas the use of variable a[p] requires two memory references.

1.6.3 Inefficiency due to late termination

Inefficiencies can come into an implementation due to late termination.

Let us consider an example to search an alphabetically ordered list of names for some particular name linearly.

Suppose we were looking for the name 'Ram', then as soon as we had encountered a name that occurred alphabetically later than Ram, (ie) Rekha, we would no need to proceed further.

The inefficient implementation could have the form :

```
while (namesought < > current name and not end of file) do
        get next name from list
```

A more efficient implementation would be :

```
while (namesought > current name and not end of file) do
        get nextname from list.
```

1.6.4 Early detection of desired output conditions

Due to the nature of the input data, the algorithm establishes the desired output condition before the general conditions for termination have been met.

For example, a bubble sort might be used to sort a set of data that is already almost in sorted order. If there have been no exchanges in the current pass, the data must be sorted and so early termination can be applied.

1.6.5 Trading storage for efficiency gains

Storage and efficiency is often used to improve the performance of an algorithm.

One strategy that sometimes used to speed up an algorithm is to implement the algorithm using the least number of loops, which makes the program much harder to read and debug. It is therefore usually better to have one loop for one job.

1.7 THE ANALYSIS OF ALGORITHMS

The analysis of algorithms is made considering both qualitative and quantitative aspects to solution that is economical in the use of computing and human resources which improve performance of an algorithm. A good algorithm usually possess the following qualities capabilities.

1. They are simple but powerful and general solutions.
2. They are user friendly
3. They can be easily updated.
4. They are correct

They are able to be understood on a number of levels.

6. They are economical in the use of computer time, storage and peripherals.
7. They are well documented.
8. They are independent to run on a particular computer.
9. They can be used as subprocedures for other problems.
10. The solution is pleasing and satisfying to its designer.

1.7.1 Computational Complexity

The computational complexity can be measured in terms of space and time required by an algorithm.

Space Complexity

The space complexity of an algorithm is the amount of memory it needs to run the algorithm.

Time Complexity

The time complexity of an algorithm is the amount of time it needs to run the algorithm.

The time taken by the program is the sum of the compile time and run time.

To make a quantitative measure of an algorithm's performance, the computational model must capture the essence of the computation while at the same time it must be divorced from any programming language.

1.7.2 Asymtotic Notation

Asymtotic notations are method used to estimate and represent the efficiency of an algorithm using simple formula. This can be useful for seperating algorithms that leads to different amounts of work for large inputs.

Comparing or classify functions that ignores constant factors and small inputs is called as asymtotic growth rate or asymtotic order or simply the order of functions. Complexity of an algorithm is usually represented in O, o, θ, Ω notations.

Big - oh notation (O)

This is a standard notation that has been developed to represent functions which bound the computing time for algorithms and it is used to define the worst case running time of an algorithm and concerned with very large values of N.

Definition : - $T(N) = O(f(N))$, if there are positive constants C and n_0 such that $T(N) \leq Cf(N)$ when $N \geq n_0$

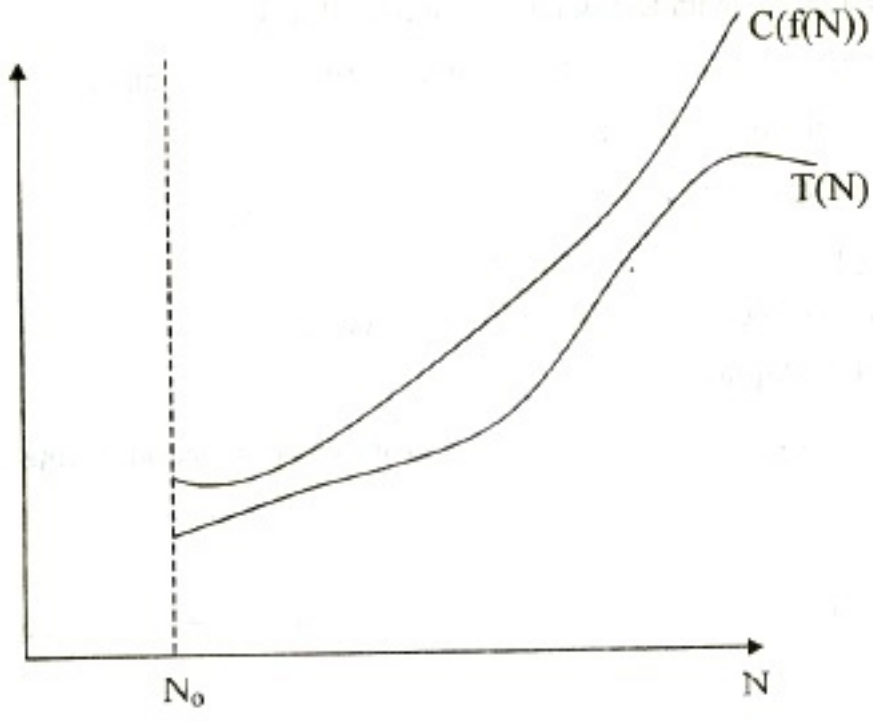


Fig. 1.7(a) Big - oh notation T(N) ∈ O(F(N))

BIG - OMEGA NOTATION (Ω)

This notation is used to describe the best case running time of algorithms and concerned with very large values of N.

Definition : - T(N) = Ω(f(N)), if there are positive constants C and n_o such that T(N) ≥ CF(N when N ≥ n_o

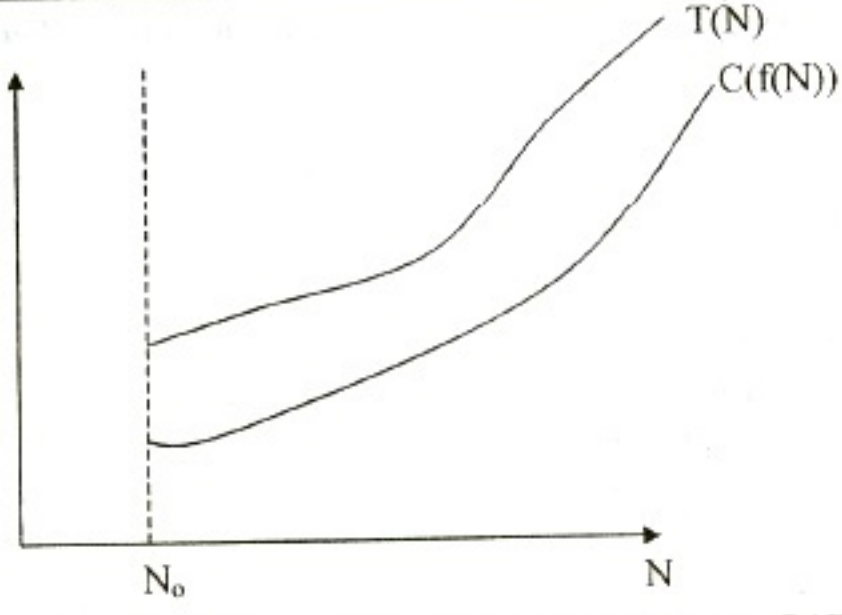


Fig. 1.7(b) BIG - OMEGA NOTATION $T(N) \in \Omega(F(N))$

BIG - THETA NOTATION (θ)

This notation is used to describe the average case running time of algorithms and concern very large values of n.

: - $T(N) = \theta(F(N))$, if there are positive constants C_1, C_2 and n_o such that $T(N) = O(F(N))$ and $T(N) = \Omega(F(N))$ for all $N \geq n_o$.

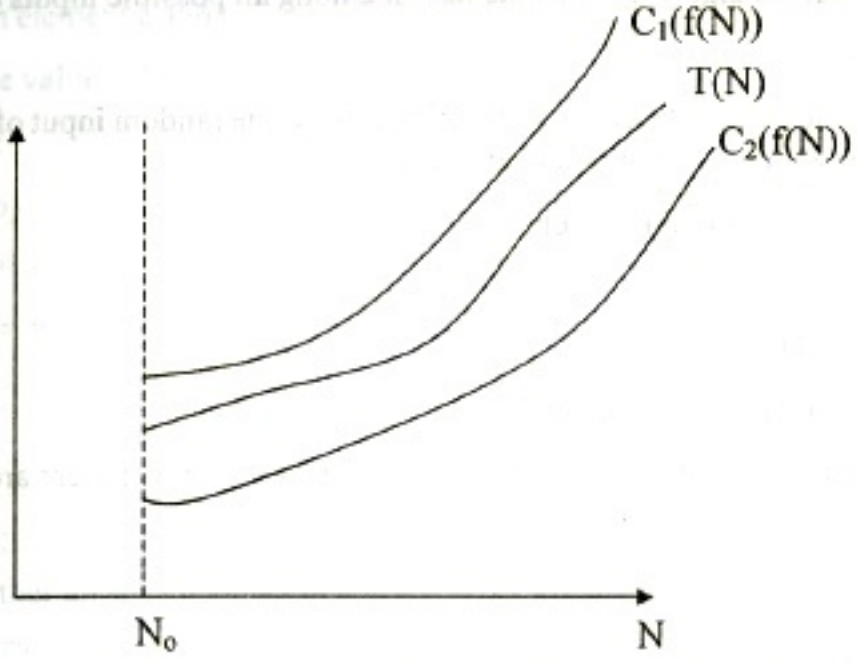


Fig. 1.7 (c) BIG - THETA NOTATION $T(N) \in \theta(F(N))$

Little - Oh Notation (o)

This notation is used to describe the worstcase analysis of algorithms and concerned with small values of n.

Definition : $T(N) = o(F(N))$ *if* $T(N) = O(F(N))$ *and* $T(N) \neq \theta(T(N))$

Basic Asymptotic Efficiency Classes

Computing Time	Name
0(1)	constant
0(log n)	Logarithmic function
0(n)	Linear
$0(n^2)$	quadratic
$0(n^3)$	cubic
$0(2^n)$	exponential
0(nlogn)	n - log - n Logarithmic
n!	factorial

1.7.3 WORST - CASE, BEST - CASE AND AVERAGE - CASE EFFICIENCIES

Worst - Case - Efficiency

The worst - case efficiency of an algorithm is its efficiency for the worst - case input of size n, which is an input of size n for which the algorithm runs the longest among all possible inputs of that size.

Best - Case Efficiency

The best - case efficiency of an algorithm is its efficiency for the best case input of size n, which is an input of size n for which the algorithm runs the fastest among all possible inputs of that size.

Average - Case Efficiency

The average - case efficiency of an algorithm is its efficiency for the random input of size n, which makes some assumptions about possible inputs of size n.

For example, let us consider sequential search

ALGORITHM

SequentialSearch (A[0...n-1],K)

// Input : An array A[0..n-1] and a search key k.

// Output : Returns the index of the first element of A that matches R or -1 if there are no matching elements.

$i \rightarrow 0$

while i < n and A[i] # k do

i $\rightarrow$ i + 1

if i < n return i

else return - 1

Here, the best - case efficiency is 0(1) where the first element is itself the search element and the worst - case efficiency is 0(n) where the last element is the search element or the search element may not be found in the given array.

1.8 SAMPLE ALGORITHMS

PROBLEM 1 (a) :

Give two variables a and b, exchange the values assigned to them without using temporary variable.

INPUT : Given element a and b

OUTPUT : The value of b is stored in variable a and the value of a is stored in variable b.

ALGORITHM :

```
Swap (a, b)
    a = a + b;
    b = a - b;
    a = a - b;
return a, b
```

Chennai University

PROBLEM 1 (b)

Give two variables a and b, exchange the values assigned to them using temporary variable.

INPUT : Given element a and b

OUTPUT : The values of a and b is swapped

ALGORITHM:

```
Swap (a, b)
        t = a;
        a = b;
        b = t;
return a, b
```

PROBLEM 2 :

To check whether all the elements in a given array are distinct.

INPUT : An array A[0...n-1]

OUTPUT : Returns "true" if all the elements in A are distinct and "false" otherwise.

ALGORITHM:

```
UniqueElements (A[0...n-1])
        for i = 0 to n-1 do
            for j = i + 1 to n-1 do
                    if A[i] = = A[j]
                    return false
        return true
```

PROBLEM 3 :

Make a count of the number of students on a particular subject who passed the examination scoring 50 and above.

INPUT : Number of Students (n)

OUTPUT : Total count of the students who passed in that subject.

ALGORITHM:

```
Passcount (n)
        count = 0;
            i = 0;
            while (i < n) do
                    read (m)
```

```
            if m > = 50 then
                    count = count + 1;
            i = i + 1;
            endwhile
        return count
```

PROBLEM 4(a) :

To find the sum of a given set of numbers.

INPUT : An array A[0.. n - 1]

OUTPUT : returns sum of the elements in the given array.

ALGORITHM:

```
Sum1(A [0 .. n-1]
        sum = 0;
            for i = 0 to n -1 do
                    sum = sum + A[i]
return sum
```

PROBLEM 4(b) :

To find the sum of the squares of a given set of numbers.

INPUT : An array A[0.. n - 1]

OUTPUT : Returns the sum of squares of a given set of numbers.

ALGORITHM:

```
Sumsquare (A[0.. n - 1])
        sum = 0;
            for i = 0 to n - 1 do
                    sum = sum + A[i] * A[i];
return sum
```

PROBLEM 5(a) :

To compute factorial of a given number without recursion.

INPUT : The element n

OUTPUT : Returns factorial of the given element n.

ALGORITHM:

```
fact (n)
        f = 1;
```

```
for (i = 1; i <= n; i ++)
    f = f * i;
return f
```

PROBLEM 5(b) :

To compute factorial of a given number using recursion.

INPUT : A positive integer n

OUTPUT : Returns the factorial of the given element.

ALGORITHM:

```
fact (n)
    if (n == 0)
        return 1
    else
        return fact (n - 1) * n
```

PROBLEM 6(a) :

Generate the Fibonacci series for first n terms.

INPUT : A positive integer n

OUTPUT : Prints Fibonacci series for n terms.

ALGORITHM:

```
Fibo (n)
    F1 = -1;    F2 = 1;
    Fib = 0;
        for i = 1 to n do
            Fib = F1 + F2 ;
            write Fib
            F1 = F2;
            F2 = Fib;
```

PROBLEM 6(b) :

Generate n^{th} Fibonacci term using recursion.

INPUT : A positive integer n

OUTPUT : The n^{th} Fibonacci number.

ALGORITHM:

```
Fibo (n)
    if n ≤ 1
        return n
    else
        return Fibo(n-1) + Fibo(n-2)
```

PROBLEM 7(a) :-

Design an algorithm to evaluate the function sin(x) as defined by the infinite series expansion.

$$Sin(x) = \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + ...$$

INPUT : - Get the values for x and n

OUTPUT : - Sine function for the given terms is calculated.

ALGORITHM:-

```
Sineseries (x, n) // Get x in radians
S = 0;
term = x;
i = 1;
while i <= n do
    S = S + term;
    term = (term * x * x * (-1)) / ((i + 1) * (i + 2));
    i = i + 2
write S;
```

PROBLEM 7(b) :-

The exponential growth constant e is characterized by the expression

$$e = \frac{1}{0!} - \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + ...$$

Device an algorithm to compute e to n terms.

INPUT : - Get the values of n (no. of terms)

OUTPUT : - Sum of the exponential series for 'n' terms is calculated.

M:-

```
EXPOSERIES (n)
        i = 1;
        e = 1;
        f = 1;
        while i <= n do
            e = e + 1/f ;
            f = f * i ;
            i = i + 1;
        write e ;
```

PROBLEM 8 :-

To sort the given set of numbers.

INPUT : - An array A[0..n - 1]of orderable element.

OUTPUT : - Array A[0..n - 1] sorted in ascending order.

ALGORITHM:-

```
Sort (A[0.. n - 1]
    for i = 0 to n - 2 do
        for j = 0 to n - 2 - i do
            if A[j + 1] < A[j]
                swap A[j] and A[j + 1]
```

PROBLEM 9 :-

To search a given element in the array using binary search non recursively.

INPUT : - An array A[0..n - 1]sorted in ascending order and a search key k.

OUTPUT : - An index of the array's element that is equal to K or -1 if there is no such element.

ALGORITHM:-

```
Binary Search (A[0 .. n - 1], K)
    l = 0;
    r = n - 1;
    while l ≤ r do
        m = (l + r) / 2
        if K = A[M]
```

```
                return m
            else if K < A[M]
                r = m - 1
            else
                l = m + 1
        return - 1
```

PROBLEM 10 :-

Design an algorithm that accepts a positive integer and reverse the order of its digits.

INPUT : - A positive integer n.

OUTPUT : - A positive integer n with reversed order of its digits.

ALGORITHM :-

```
Reverse (n)
    s = 0;
    while n > 0 do
            r = n % 10 ;
            s = r + s * 10;
            n = n/10;
    write s
```

PROBLEM 11 :-

To compute the multiplication of two square matrices of order n.

INPUT : - Two n x n matrices A and B.

OUTPUT : - Matrix C = AB

ALGORITHM :-

```
MatrixMult (A[0..n - 1, 0.. n - 1], B[0..n - 1, 0..n -1])
    for i = 0 to n - 1 do
            for j = 0 to n - 1 do
                C[i, j] = 0;
                for k = 0 to n - 1 do
                    C[i, j] = C[i, j] + A[i, k] * B[k, j];
return C
```

Chennai University

12(a) :-

To convert the given decimal number into its binary form.

INPUT : - A positive integer n.

OUTPUT : - A binary representation for the given decimal number n.

ALGORITHM:-

```
Decitobin (n)
    C = 0;
    while n > 0 do
        r = n% 2;
        a [i] = r;
        n = n/2 ;
        C = C + 1;
    endwhile
        for i = C - 1 to 0
        write a[i];
```

PROBLEM 12(b) :-

To convert a binary number into a decimal number.

INPUT : - A positive integer in binary form.

OUTPUT : - decimal form of the given number is obtained.

ALGORITHM:-

```
Bintodeci (n)
    q = n;
    s = 0;
    i = 0;
    while q > 0 do
        r = q % 10 ;
        s = s + r * pow (2, i);
        i = i + 1;
    endwhile
    write s ;
```

Questions

Part - A

1. Define an algorithm
2. Write an algorithm to swap two numbers.
3. Define the top - down design strategy.
4. What is meant by problem solving?
5. Mention some of the problem solving strategies.
6. What is divide and conquer strategy?
7. What are the steps involved in problem solving?
8. Write an algorithm to find the factorial of a given number.
9. List the types of control structures.
10. Define the worst case and average case complexities of an algorithm.
11. How do you debug a program?
12. What is program testing?
13. What is program verification?
14. Write down the various phases in the program verification.
15. Define space complexity and time complexity.
16. What are the qualities and capabilities of good algorithm.
17. How can you improve an efficiency of an algorithm in the design phase?
18. What do you mean by proof of termination?
19. What is O - notation?
20. Write an algorithm to find the sum of a set of numbers.

Part - B

1. Explain Top - down design strategy in detail.
2. Write down the algorithm for binary search.
3. The exponential growth constant e is characterized by the expression

   $e = \frac{1}{0!} - \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots\ldots$. Device an algorithm to compute e to n terms.
4. Explain in detail the types of analysis that can be performed on an algorithm.
5. Write an algorithm to perform matrix multiplication and give the analysis for it.
6. Design an algorithm to compute the sum of the squares of n numbers.
7. Design an algorithm that accepts a positive integer and reverses the order of digits.
8. Write an algorithm to convert a decimal number to its corresponding octal form.

ABSTRACT DATA TYPE

In programming each program is breakdown into modules, so that no routine should ever exceed a page. Each module is a logical unit and does a specific job modules which inturn will call another module.

Modularity has several advantages

1. Modules can be compiled separately which makes debugging process easier.
2. Several modules can be implemented and executed simultaneously.
3. Modules can be easily enhanced.

Abstract Data type is an extension of modular design.

An abstract data type is a set of operations such as Union, Intersection, Complement, Find etc.,

The basic idea of implementing ADT is that the operations are written once in program and can be called by any part of the program.

2.1 THE LIST ADT

List is an ordered set of elements.

The general form of the list is

$A_1, A_2, A_3, \ldots\ldots, A_N$

A_1 - First element of the list

A_N - Last element of the list

N - Size of the list

If the element at position i is A_i then its successor is A_{i+1} and its predecessor is A_{i-1}.

Various operations performed on List

1. Insert (X, 5) - Insert the element X after the position 5.
2. Delete (X) - The element X is deleted
3. Find (X) - Returns the position of X.
4. Next (i) - Returns the position of its successor element i+1.
5. Previous (i) - Returns the position of its predecessor i-1.
6. Print list - Contents of the list is displayed.
7. Makeempty - Makes the list empty.

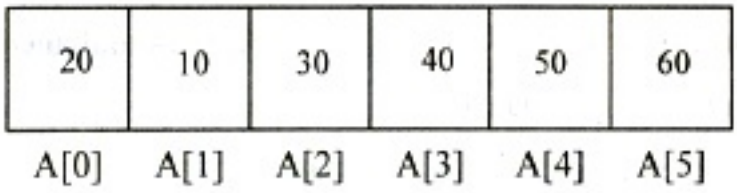
2.1 .1 Implementation of List ADT

1. Array Implementation
2. Linked List Implementation
3. Cursor Implementation.

Implementation of List

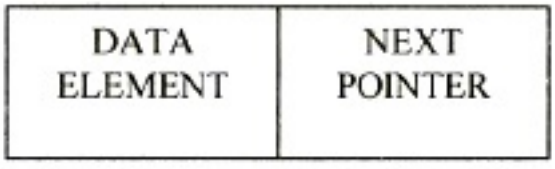
is a collection of specific number of data stored in a consecutive memory locations.

* Insertion and Deletion operation are expensive as it requires more data movement
* Find and Printlist operations takes constant time.
* Even if the array is dynamically allocated, an estimate of the maximum size of the list is required which considerably wastes the memory space.



Linked List Implementation

Linked list consists of series of nodes. Each node contains the element and a pointer to its successor node. The pointer of the last node points to NULL.



Insertion and deletion operations are easily performed using linked list.

Types of Linked List

1. Singly Linked List
2. Doubly Linked List
3. Circular Linked List.

2.1.2 Singly Linked List

A singly linked list is a linked list in which each node contains only one link field pointing to the next node in the list.

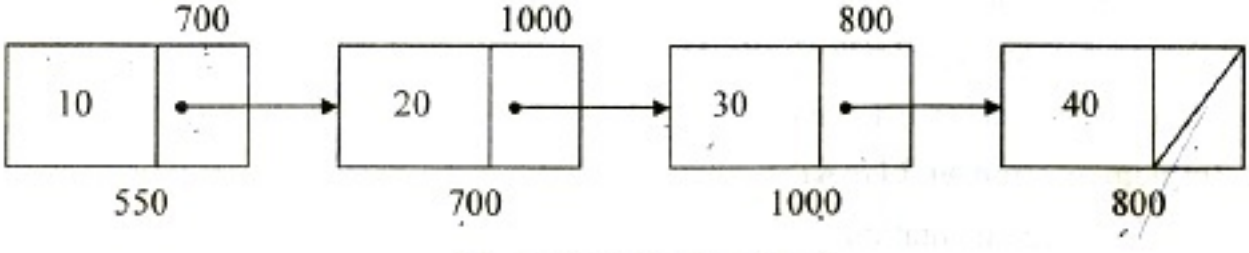


Fig. 2.1 LINKED LIST

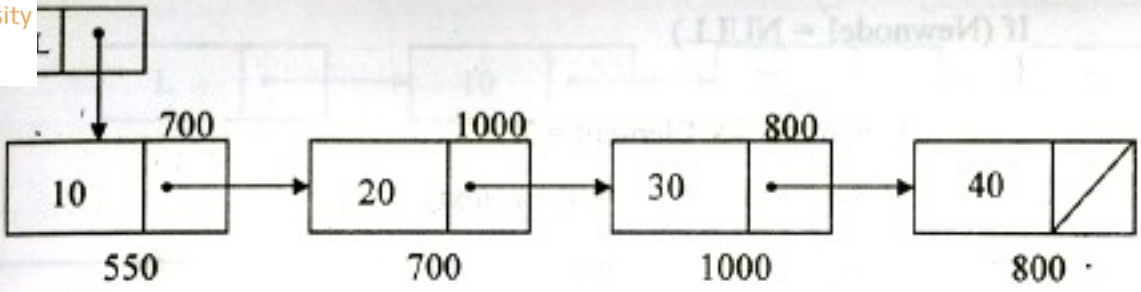


Fig. 2.1 LINKED LIST WITH A HEADER

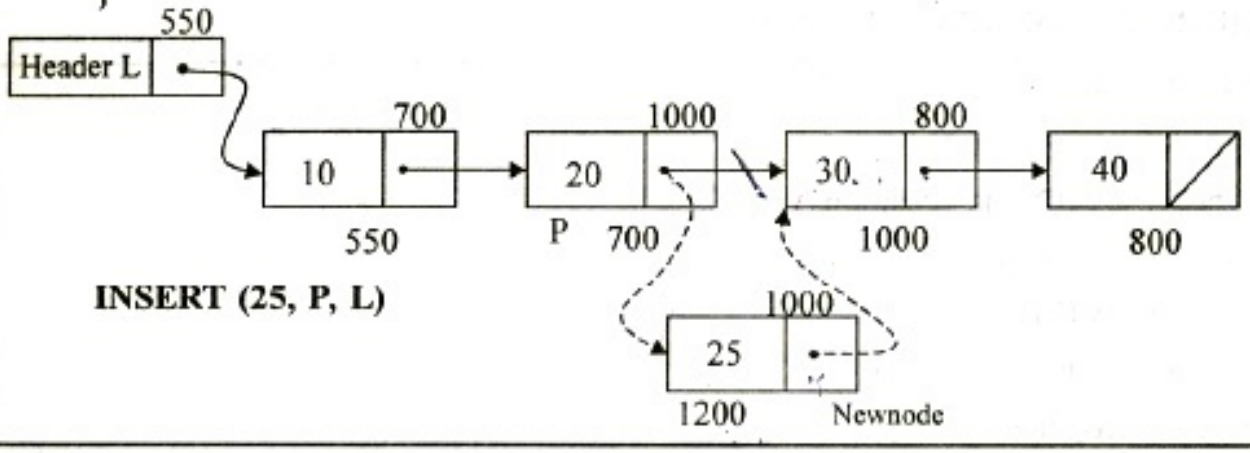
DECLARATION FOR LINKED LIST

```
Struct node ;
typedef struct Node *List ;
typedef struct Node *Position ;
int IsLast (List L) ;
int IsEmpty (List L) ;
position Find(int X, List L) ;
void Delete(int X, List L) ;
position FindPrevious(int X, List L) ;
position FindNext(int X, List L) ;
void Insert(int X, List L, Position P) ;
void DeleteList(List L) ;
Struct Node
{
        int element ;
        position Next ;
};
```

ROUTINE TO INSERT AN ELEMENT IN THE LIST

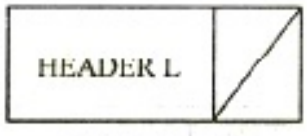
```
void Insert (int X, List L, Position P)
        /* Insert after the position P*/
{

position Newnode;
Newnode = malloc (size of (Struct Node));
```

```
If (Newnode! = NULL)
{
        Newnode → Element = X;
        Newnode → Next = P →Next;
        P →Next = Newnode;
}
}
```



ROUTINE TO CHECK WHETHER THE LIST IS EMPTY

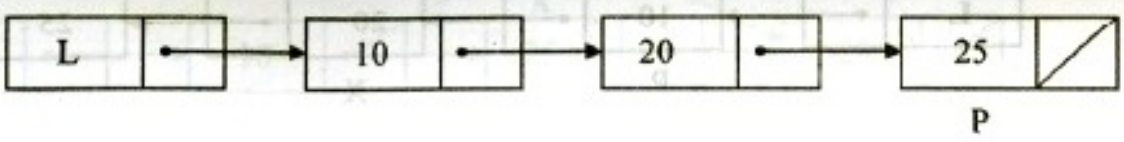
```
int IsEmpty (List L) /*Returns 1 if L is empty */
{
        if (L →Next = = NULL)
          return (1);
}
```



Empty List

ROUTINE TO CHECK WHETHER THE CURRENT POSITION IS LAST

```
int IsLast (position P, List L) /* Returns 1 is P is the last position in L */
{
        if (P →Next = = NULL)
                return (1);
```

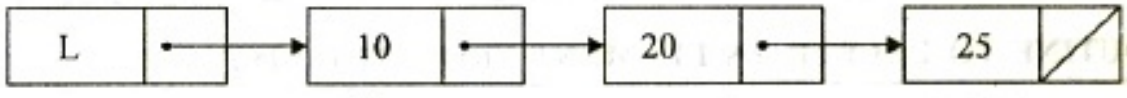


}

FIND ROUTINE

```
position Find (int X, List L)
{
        /*Returns the position of X in L; NULL if X is not found */
        position P;
        P = L → Next;
        while (P! = NULL && P → Element ! = X)
                P = P → Next;
        return P;
}
```

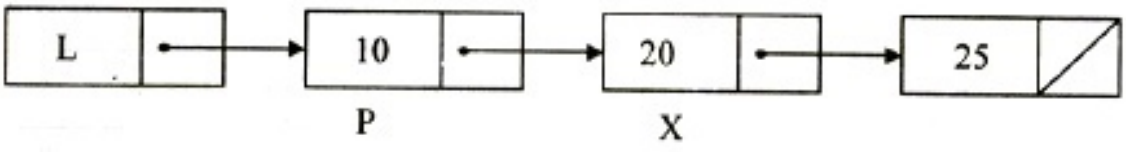
FIND (10)



}

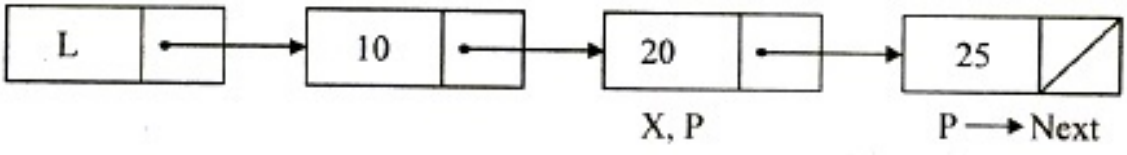
FIND PREVIOUS ROUTINE

```
position FindPrevious (int X, List L)
{
        /* Returns the position of the predecessor */
        position P;
        P = L;
        while (P → Next ! = Null && P → Next → Element ! = X)
                P = P → Next;
        return P;
}
```



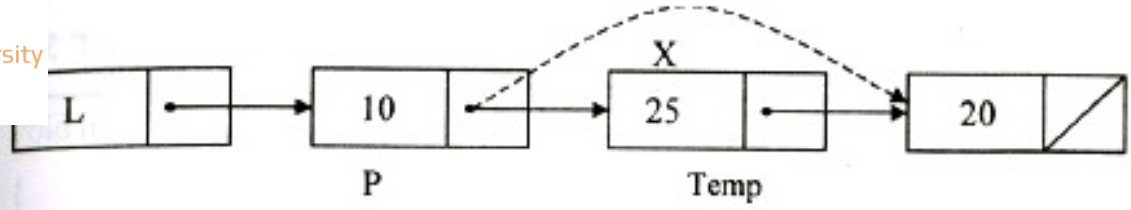
FINDNEXT ROUTINE

```
position FindNext (int X, List L)
{
        /*Returns the position of its successor */
        P = L → Next;
        while (P → Next! = NULL  && P → Element ! = X)
                P = P → Next;
        return P → Next;
}
```

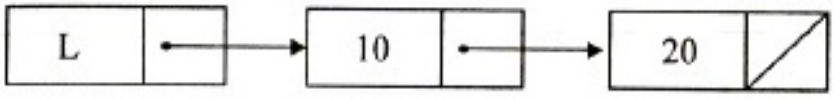


ROUTINE TO DELETE AN ELEMENT FROM THE LIST

```
void Delete(int X, List L)
{
        /* Delete the first occurence of X from the List */
        position P, Temp;
        P = Findprevious (X,L);
        If (!IsLast(P,L))
        {
                Temp = P → Next;
                P → Next = Temp → Next;
                Free (Temp);
        }
}
```



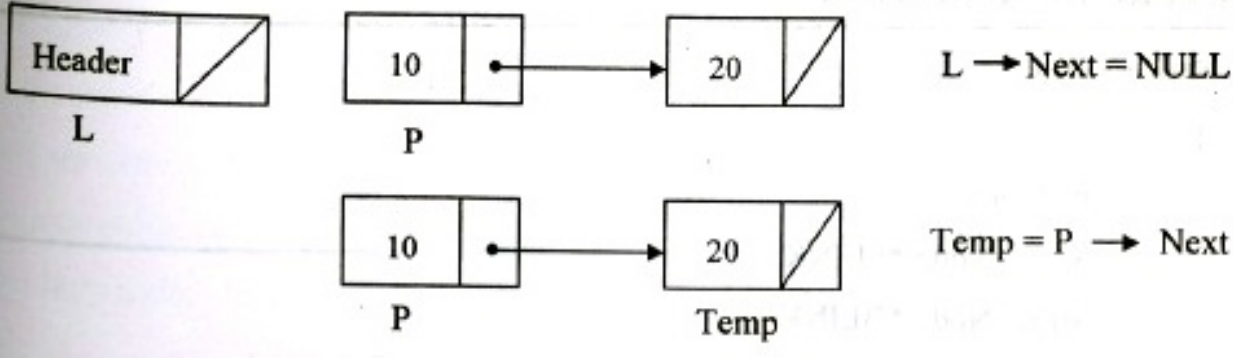
BEFORE DELETION

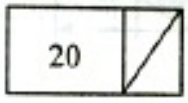


AFTER DELETION

ROUTINE TO DELETE THE LIST

```
void DeleteList (List L)
{
        position P, Temp;
        P = L → Next;
        L → Next = NULL;
        while (P! = NULL)
        {
                Temp = P → Next
                free (P);
                P = Temp;
        }
}
```





P

free (P)

P = Temp

Temp, P = NULL

2.1.3 Doubly Linked List

A Doubly linked list is a linked list in which each node has three fields namely data field, forward link (FLINK) and Backward Link (BLINK). FLINK points to the successor node in the list whereas BLINK points to the predecessor node.

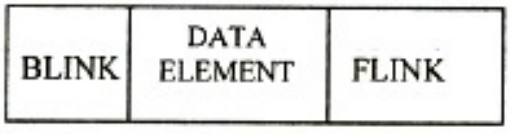


Fig. 2.1.3 (a) NODE IN DOUBLY LINKED LIST

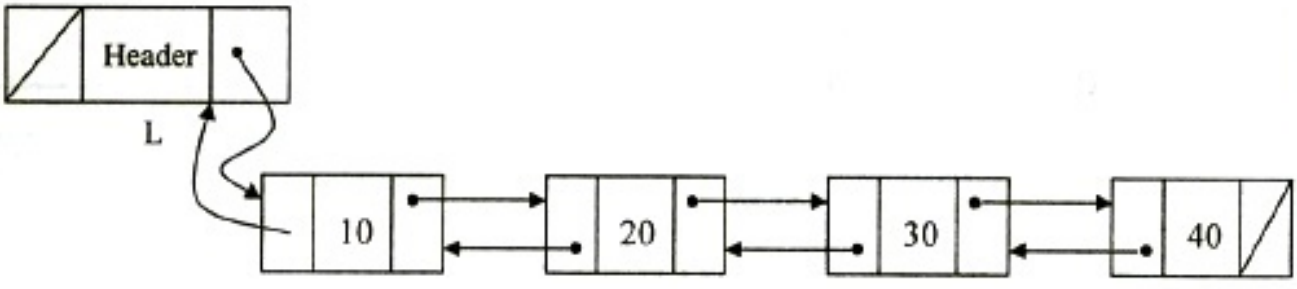


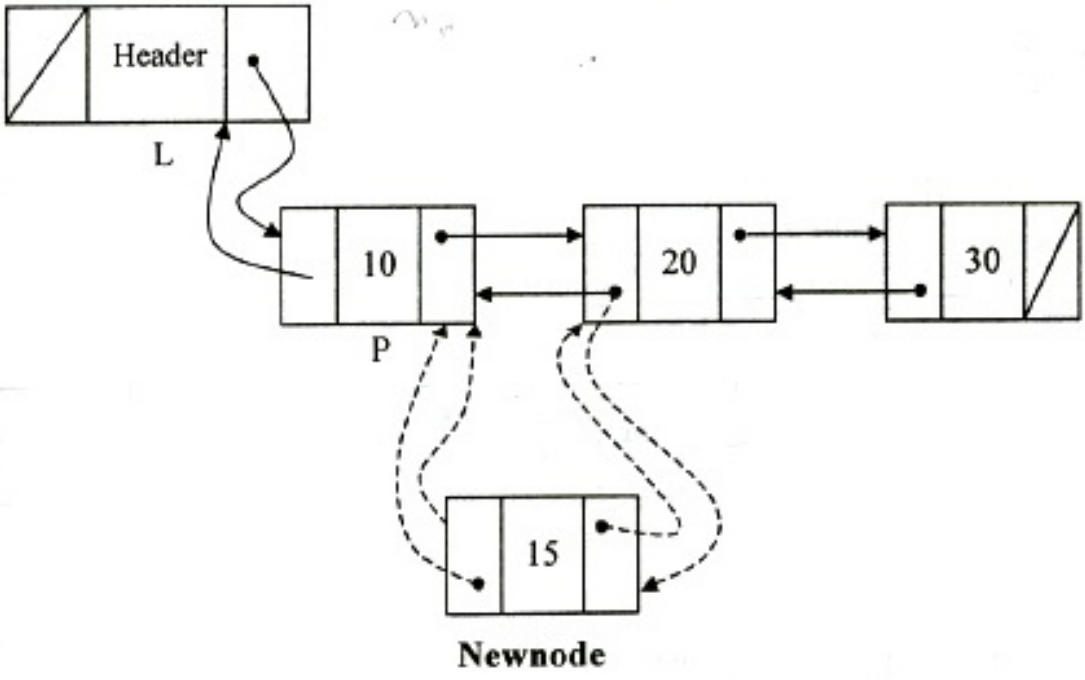
Fig. 2.1.3 (b) DOUBLY LINKED LIST

STRUCTURE DECLARATION : -

```
Struct Node
{
        int Element;
        Struct Node *FLINK;
        Struct Node *BLINK
};
```

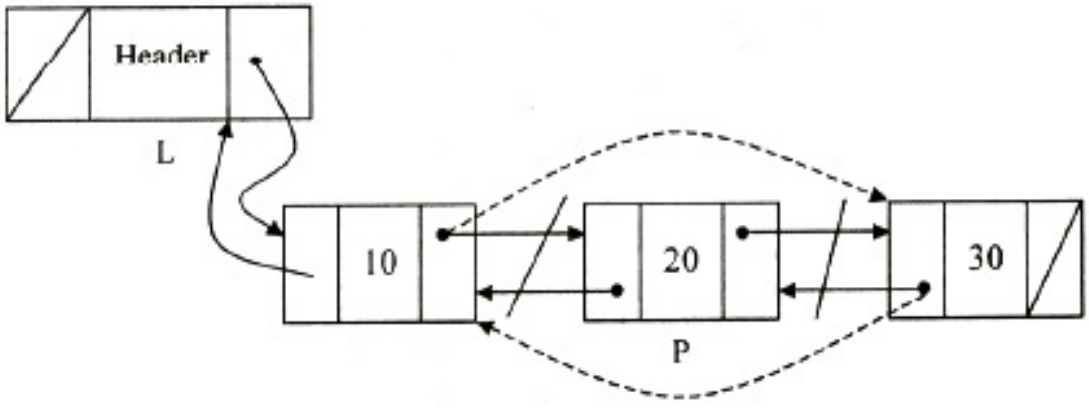
TO INSERT AN ELEMENT IN A DOUBLY LINKED LIST

```
Insert (int X, list L, position P)
{
    Struct   Node * Newnode;
    Newnode = malloc (size of (Struct Node));
    If (Newnode ! = NULL)
    {
        Newnode → Element = X;
        Newnode → Flink = P → Flink;
        P → Flink → Blink = Newnode;
        P → Flink = Newnode ;
        Newnode → Blink = P;
    }
}
```



TINE TO DELETE AN ELEMENT

```
void Delete (int X, List L)
{
        position P;
        P = Find (X, L);
        If ( IsLast (P, L))
        {
                Temp = P;
                P → Blink →Flink = NULL;
                free (Temp);
        }
        else
        {
                Temp = P;
                P → Blink →Flink = P →Flink;
                P →Flink →Blink = P →Blink;
                free (Temp);
        }
}
```



Advantage

- Deletion operation is easier.
- Finding the predecessor & Successor of a node is easier.

Disadvantage

- More Memory Space is required since it has two pointers.

ar Linked List

ıked list the pointer of the last node points to the first node. Circular linked list can be implemented as Singly linked list and Doubly linked list with or without headers.

Singly Linked Circular List

A singly linked circular list is a linked list in which the last node of the list points to the first node.

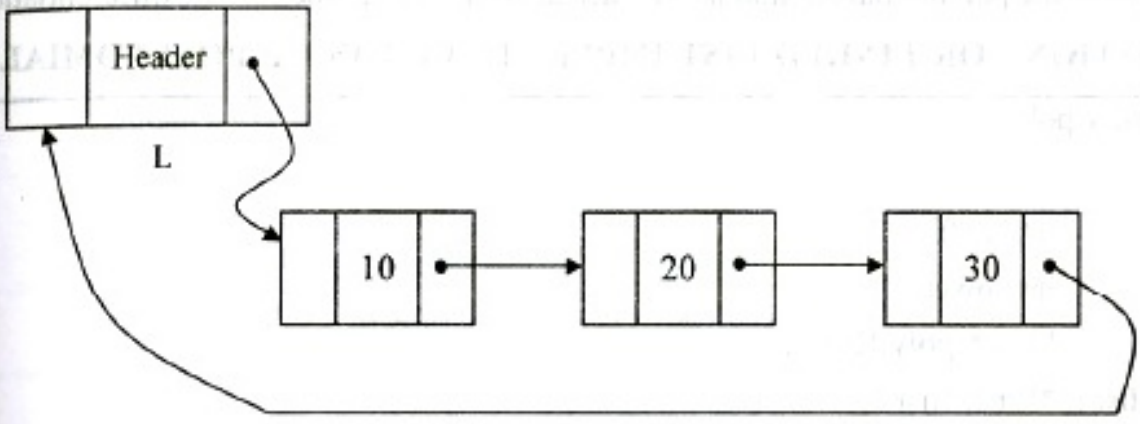


Fig. 2.1.4 Singly Linked Circular List With Header

Doubly Linked Circular List

A doubly linked circular list is a Doubly linked list in which the forward link of the last node points to the first node and backward link of the first node points to the last node of the list.

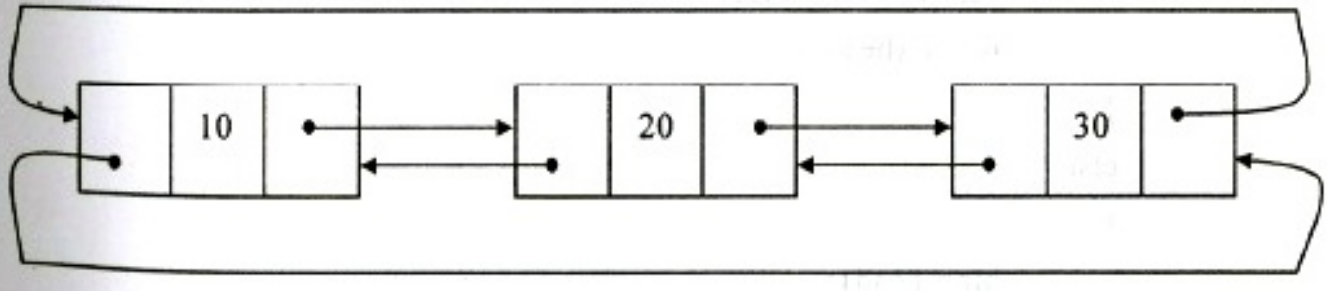


Fig. 2.1.4 (b) Doubly Linked Circular List With Header

Advantages of Circular Linked List

- It allows to traverse the list starting at any point.
- It allows quick access to the first and last records.
- Circularly doubly linked list allows to traverse the list in either direction.

5 Applications of Linked List

1. Polynomial ADT
2. Radix Sort
3. Multilist

Polynomial ADT

We can perform the polynomial manipulations such as addition, subtraction and differentiation etc

DECLARATION FOR LINKED LIST IMPLEMENTATION OF POLYNOMIAL ADT

```
Struct poly
{
        int coeff ;
        int power;
        Struct  poly *Next;
}*list 1, *list 2, *list 3;
```

CREATION OF THE POLYNOMIAL

```
poly create (poly *head1, poly *newnode1)
{
        poly *ptr;
        if (head1= =NULL)
        {
                head1 = newnode1;
                return (head1);
        }
        else
        {
                ptr = head1;
                while (ptr → next ! = NULL)
                ptr = ptr → next;
                ptr → next = newnode1;
        }
        return (head1);
}
```

add ( )

```
poly *ptr1, *ptr2, *newnode;
ptr1 = list1;
ptr2 = list2;
while (ptr1! = NULL && ptr2! = NULL)
{
        newnode = malloc (sizeof (Struct poly));
        if (ptr1 → power = = ptr2 → power)
        {
                newnode → coeff = ptr1 → coeff + ptr2 → coeff;
                newnode → power = ptr1 → power;
                newnode → next = NULL;
                list 3 = create (list3, newnode);
                ptr1 = ptr1 → next;
                ptr2 = ptr2 → next;
        }
        else
        {
                if (ptr1 → power > ptr2 → power)
                {
                        newnode → coeff = ptr1 → coeff;
                        newnode → power = ptr1 → power;
                        newnode → next = NULL;
                        list3 = create (list3, newnode);
                        ptr1 = ptr1 → next;
                }
```

```
                else
                {
                        newnode → coeff = ptr2 → coeff;
                        newnode → power = ptr2 → power;
                        newnode → next = NULL;
                        list3 = create (list3, newnode);
                        ptr2 = ptr2 → next;
                }
        }
    }
```

SUBTRACTION OF TWO POLYNOMIAL

```
    void sub ( )
    {
        poly *ptr1, *ptr2, *newnode;
        ptr1 = list1 ;
        ptr2 = list 2;
        while (ptr1! = NULL && ptr2! = NULL)
        {
            newnode = malloc (sizeof (Struct poly));
            if (ptr1 → power = = ptr2 → power)
            {
                newnode → coeff = (ptr1 → coeff) - (ptr2 → coeff);
                newnode → power = ptr1 → power;
                newnode → next = NULL;
                list3 = create (list 3, newnode);
                ptr1 = ptr1 → next;
                ptr2 = ptr2 → next;
            }
            else
            {
```

```
                if (ptr1 → power > ptr2 → power)
                {
                        newnode → coeff = ptr1 → coeff;
                        newnode → power = ptr1 → power;
                        newnode → next = NULL;
                        list 3 = create (list 3, newnode);
                        ptr1 = ptr1 → next;
                }
                else
                {
                        newnode → coeff = - (ptr2 → coeff);
                        newnode → power = ptr2 → power;
                        newnode → next = NULL;
                        list 3 = create (list 3, newnode);
                        ptr2 = ptr2 → next;
                }
            }
        }
    }
```

POLYNOMIAL DIFFERENTIATION

```
void diff ( )
{
        poly *ptr1, *newnode,
        ptr1 = list 1;
        while (ptr1 ! = NULL)
        {
                newnode = malloc (sizeof (Struct poly));
                newnode → coeff = ptr1 → coeff *ptr1 → power;
                newnode → power = ptr1 → power - 1;
                newnode → next = NULL;
                list 3 = create (list 3, newnode);
                ptr1 = ptr1 → next;
        }
}
```

Chennai University

ix Sort : - (Or) Card Sort

x Sort is the generalised form of Bucket sort. It can be performed using buckets from 0 to 9

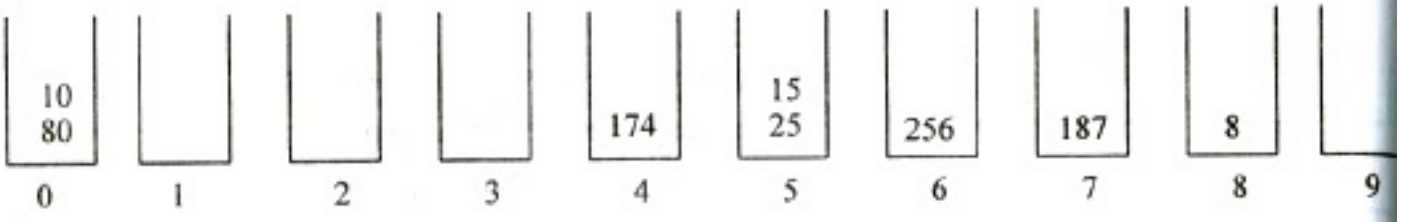
In First Pass, all the elements are sorted according to the least significant bit.

In second pass, the numbers are arranged according to the next least significant bit and so on thi process is repeated until it reaches the most significant bits of all numbers.

The numbers of passes in a Radix Sort depends upon the number of digits in the numbers given.

PASS 1 :

INPUT : 25, 256, 80, 10, 8, 15, 174, 187

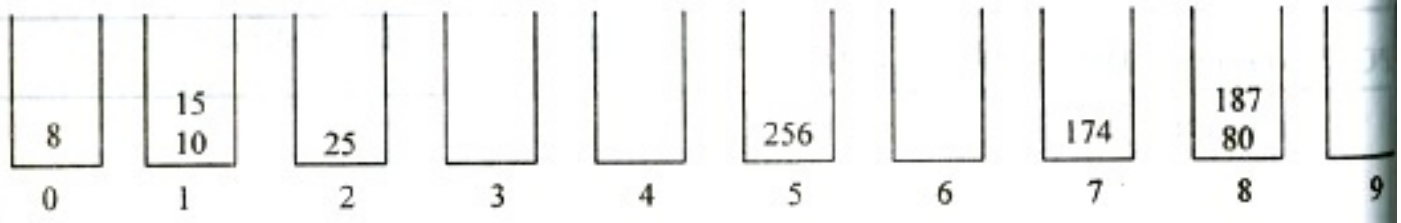


Buckets

After Pass 1 : 80, 10, 174, 25, 15, 256, 187, 8

PASS 2 :

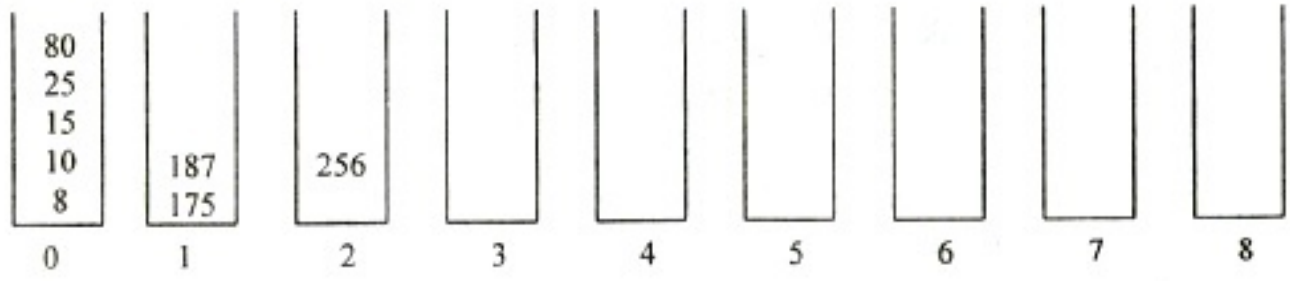
INPUT : 80, 10, 174, 25, 15, 256, 187, 8



After Pass 2 : 8, 10, 15, 25, 256, 174, 80, 187

PASS 3 :

INPUT : 8, 10, 15, 25, 256, 174, 80, 187



After pass 3 : 8, 10, 15, 25, 80, 175, 187, 256

Notes

umber of digits in the given list is 3. Therefore the number of passes required to sort ements is 3.

Multi Lists

More complicated application of linked list is multilist. It is useful to maintain student registration, Employee involved in different projects etc., Multilist saves space but takes more time to implement.

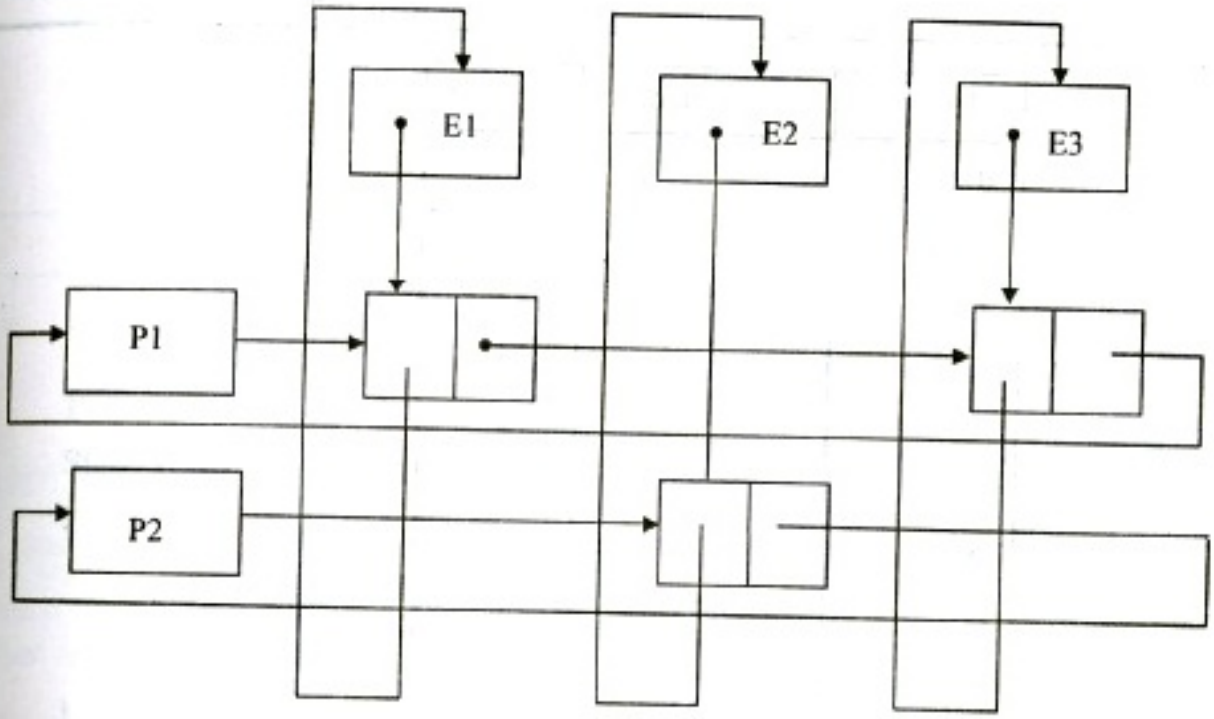


Fig. 2.1.5 Multilist Implementation For Employee Project Allocation

An employee can involve in any number of projects and each project can be implemented by any number of employees.

Employee E_1 is working on project P_1, E_2 is working on project P_2 & E_3 is working on project P_1. Project P_1 is implemented by the Employees E_1 & E_3. Project P_2 is implemented by the Employee E_2.

2.1.6 Cursor Implementation of Linked Lists

Cursor implementation is very useful where linked list concept has to be implemented without using pointers.

Comparison on Pointer Implementation and Curson Implementation of Linked List.

Pointer Implementation	Cursor Implementation
1. Data are stored in a collection of structures. Each structure contains a data and next pointer.	Data are stored in a global array of structures. Here array index is considered as an address.

number of digits in the given list is 3. Therefore the number of passes required to sort elements is 3.

Multi Lists

More complicated application of linked list is multilist. It is useful to maintain student registration, Employee involved in different projects etc., Multilist saves space but takes more time to implement.

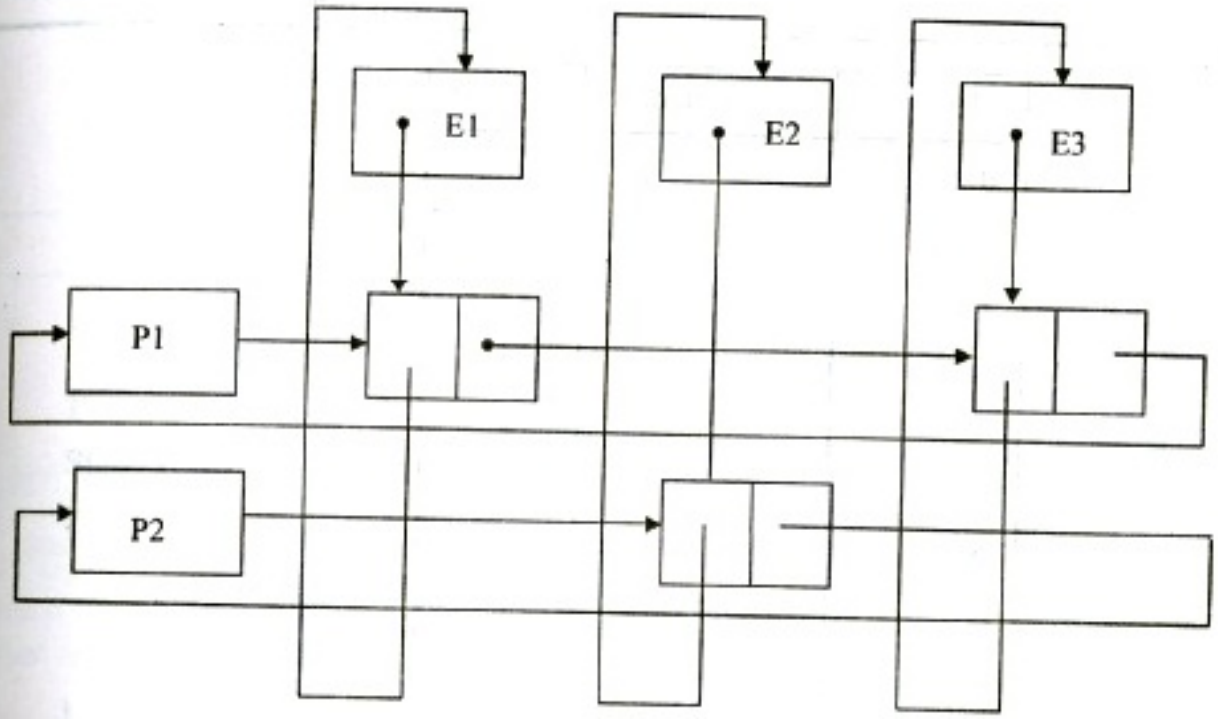


Fig. 2.1.5 Multilist Implementation For Employee Project Allocation

An employee can involve in any number of projects and each project can be implemented by any number of employees.

Employee E_1 is working on project P_1, E_2 is working on project P_2 & E_3 is working on project P_1. Project P_1 is implemented by the Employees E_1 & E_3. Project P_2 is implemented by the Employee E_2.

2.1.6 Cursor Implementation of Linked Lists

Cursor implementation is very useful where linked list concept has to be implemented without using pointers.

Comparison on Pointer Implementation and Curson Implementation of Linked List.

Pointer Implementation	Cursor Implementation
1. Data are stored in a collection of structures. Each structure contains a data and next pointer.	Data are stored in a global array of structures. Here array index is considered as an address.

```
    position Next;
};
Struct Node Cursorspace [space size];
```

ROUTINE FOR CURSOR ALLOC & CURSOR FREE

```
Static position CursorAlloc (void)
{
        position P;
        P = CursorSpace [0].Next;
        CursorSpace [0].Next = CursorSpace [P].Next;
        return P;
}
Static void CursorFree (position P)
{
        CursorSpace [P].Next = CursorSpace [0].Next;
        CursorSpace [0].Next = P;
}
```

ROUTINE TO CHECK WHETHER THE LIST IS EMPTY

```
int IsEmpty (List L)
{       /* Returns 1 if the list is Empty */.
        if (Cursorspace [0].  Next = = 0)
        return (1);
}
```

ROUTINE FOR ISLAST

```
int IsLast (Position P, List L)
{       /* Returns 1 if the p is in last position */
        if (CursorSpace [P].Next = = 0)
        return (1);
}
```

TINE TO FIND AN ELEMENT

```
position Find (int X, List L)
{
        position P;
        P = CursorSpace [L].Next;
        while (P && CursorSpace [P].Element ! = X)
                P = CursorSpace [P].Next;
        return P;
}
```

ROUTINE TO DELETE AN ELEMENT

```
void Delete (int X, List L)
{
        position P, temp;
        P = Findprevious (X, L);
        if (! IsLast (P, L))
        {
                temp = CursorSpace [P].Next;
                CursorSpace [P].Next = CursorSpace [temp].Next;
                CursorFree (temp);
        }
}
```

ROUTINE FOR INSERTION

```
void Insert (int X, List L, position P)
{
        position newnode;
        newnode = CursorAlloc ( );
        if (newnode ! = 0)
        CursorSpace [newnode].Element = X;
        CursorSpace [newnode]. Next = CursorSpace [P].Next;
        CursorSpace [P].Next = newnode;
}
```