CHAPTER 1

Unit I

Introduction to Algorithms

Syllabus :

Algorithms : Problem Solving, Introduction to algorithm, Characteristics of algorithm, Algorithm design tools: Pseudo-code and flowchart

Complexity of algorithm : Space complexity, Time complexity, Asymptotic notation- Big-O, Theta and Omega, Finding complexity using step count method, Analysis of programming constructs-Linear, Quadratic, Cubic, Logarithmic.

1.1 Introduction to Algorithms

1.1.1 Definition and Characteristics

Q. Define algorithms and its characteristics.
SPPU - Dec. 16, 4 Marks

Q. Define and explain the term Algorithm.
SPPU - Dec. 17, May 18, 1 Mark

Q. State the characteristics of an algorithm.
SPPU - May 19, 2 Marks

The field of computer science revolves around writing of programs for several problems for various domains. A program consists of data structures and algorithms. An algorithm is a set of steps required to solve a problem. These steps are performed on a sample data representing an instance of the problem. Thus an algorithm maps a set of input data (from input domain) to a set of output data (in output domain) through a sequence of operations. An algorithm must have the following properties :

(1) **Input :** Input data, supplied externally (zero or more).

(2) **Output :** Result of the program.

(3) **Finiteness :** In every case, algorithm terminates after a finite number of steps.

(4) **Definiteness :** The steps should be clear and unambiguous.

(5) **Effectiveness :** An algorithm should be written using basic instructions. It should be feasible to convert the algorithm in a computer program.

Take the problem of finding the GCD (Greatest Common Divisor) of two positive integers an example.

Inputs to the algorithm are two positive integers. Output is a positive integer which is GCD of two positive integers given as input.

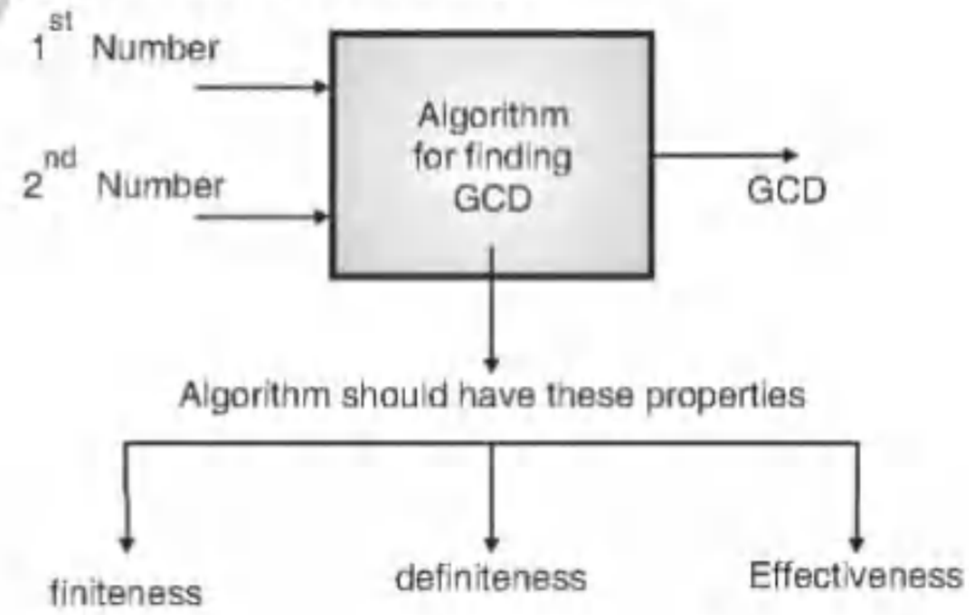


Fig. 1.1.1

The algorithm is described as a series of steps of basic operations. These steps must be performed in a sequence. Each step of the algorithm is labeled.

Step 1 : Read two positive integers and store them in x and y.

Step 2 : Divide x by y. Let the remainder be r and the quotient be q.

Step 3 : If r is zero then goto step 7.

Step 4 : Assign y to x.

Step 5 : Assign r to y.

Step 6 : goto step 2.

Step 7 : Print y (the required GCD)

Step 8 : stop

The steps mentioned in the above algorithm are simple and unambiguous. Anybody, carrying out these steps will clearly know what to do in each step. Hence, the above algorithm satisfies the **definiteness** property of an algorithm.

Table 1.1.1 depicts step-wise execution of the above algorithm on two input numbers 15 and 9.

Table 1.1.1

Steps performed	Value of the variables				
	x	y	r	q	
Step 1	15	9	–	–	Input data supplied externally
Step 2	15	9	6	1	
Step 3	15	9	6	1	
Step 4	9	9	6	1	
Step 5	9	6	6	1	
Step 6	9	6	6	1	
Step 2	9	6	3	1	
Step 3	9	6	3	1	
Step 4	6	6	3	1	
Step 5	6	3	3	1	
Step 6	6	3	3	1	
Step 2	6	3	0	2	
Step 3	6	3	0	2	
Step 7	→ Print the output (GCD) as 3.				

Algorithm terminates after a finite number of steps. It is easy to prove that the algorithm will terminate for any two given integer numbers. Thus the above algorithm satisfies the **finiteness** property of the algorithm.

Every programming language has input statement, output statement, assignment statement, conditional branching statement, unconditional branching statement and hence the above algorithm can easily be converted to a program. Thus the above algorithm satisfies the **effectiveness** property of the algorithm.

An algorithm can be described in many ways. A language like English can be used but it should be free from any ambiguity. Flowcharts, consisting of graphical symbols like :

Table 1.1.2 : Symbols used in flowcharts

Symbol	Purpose
	Terminal Symbol (Start / Stop)
	Assignment Statements, expression etc.
	Read / Print
	Decision making
	Connector
→	Flow indication

Ex. 1.1.1 : Write an algorithm for adding 10 numbers.

Soln. :

Algorithm as a series of steps

Step 1 : Assign 10 to N

Step 2 : Assign 0 to SUM

Step 3 : Assign 1 to I

Step 4 : if (I > N) goto step 9

Step 5 : read X

Step 6 : Assign (SUM + X) to SUM

Step 7 : Assign (I + 1) to I

Step 8 : goto step 4

Step 9 : Print SUM

Step 10 : Stop

Tech Knowledge Publications

Flow chart

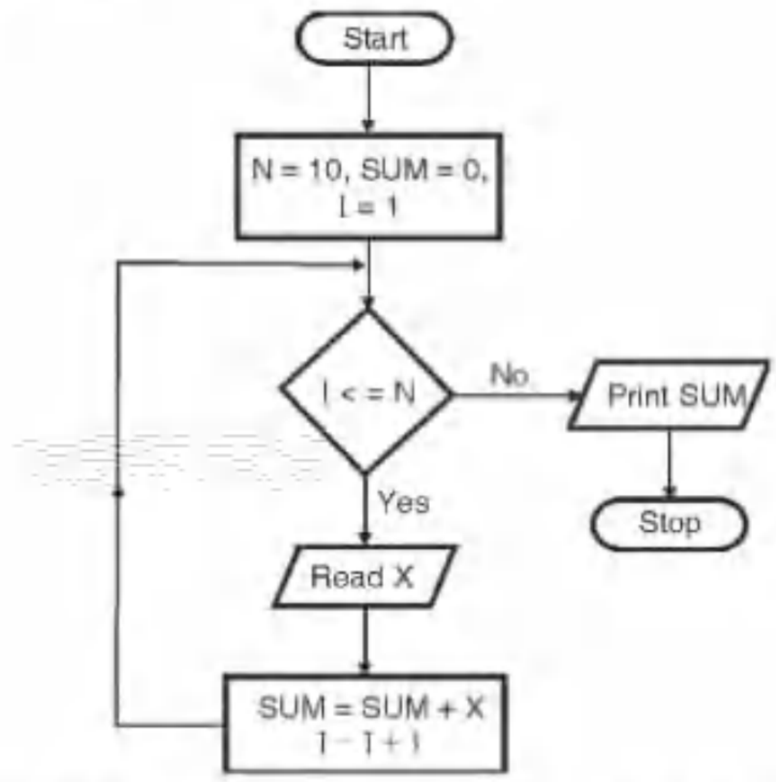


Fig. Ex. 1.1.1 : Flowchart for adding 10 numbers

Ex. 1.1.2 : Give an algorithm to find factorial of a number.

Soln. : Algorithm as a series of steps

Step 1 : read n

Step 2 : Assign 1 to fact

Step 3 : if n < = 0 then goto step 7

Step 4 : Assign n*fact to fact

Step 5 : Assign (n – 1) to n

Step 6 : goto step 3

Step 7 : Print fact

Step 8 : Stop

Flow chart :

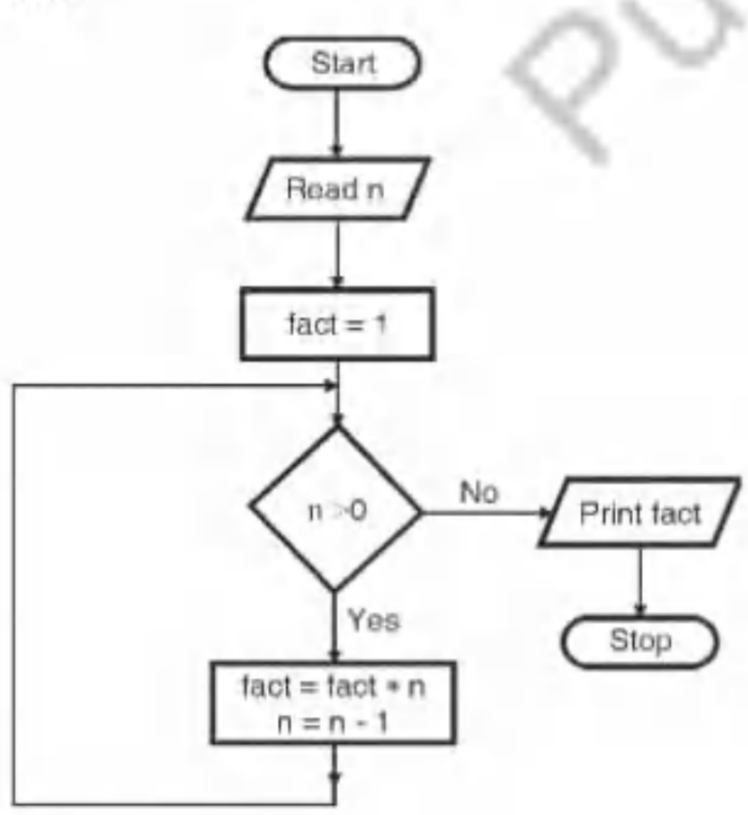


Fig. Ex. 1.1.2

1.1.2 Algorithm Design Tools

The most difficult and important task within programming is the systematic and careful analysis of a whole problem. Therefore, before going to actual programming, a programmer should always go through the following steps :

(i) Design an algorithm representing the process of solution of problem.

(ii) Represent the algorithm through flowchart.

(iii) Code the flowchart i.e. write instruction in a programming language that a specific computer will accept.

(iv) Run the program for the given input.

1.1.2(A) Flowcharting

Flowchart is a graphical representation of an algorithm. It makes use of the basic operations in programming. All symbols are connected among themselves to indicate the flow of information and processing. Flowchart uses a list of symbols. These symbols are shown in the Table 1.1.2.

1.1.2(B) Pseudo-Language

It is often convenient to present algorithm using a Pseudo language. A Pseudo language is a combination of the constructs of a programming language together with informal english statements.

It is often appropriate to formulate an algorithm in terms of the model. Initial versions of the algorithm are often written using general statements. These statements are subsequently refined into more definite instructions.

Ex. 1.1.3 : For the following problems draw the flowchart and also give the C-Pseudo code.

(a) Print all prime factors of given number n.

(b) HCF of two numbers.

(c) Print all prime numbers between two given numbers x and y, where $x \leq y$.

Tech Knowledge Publications

Soln. :

(a) Prime factors of given number n

C-Pseudo code

```
main()
{
    int n, i;
    read n;
    i = 2;
    while(i < = n/2)
    {
        if(n is divisible by i)
            print i;
        else
            i = i + 1;
    }
}
```

Flowchart

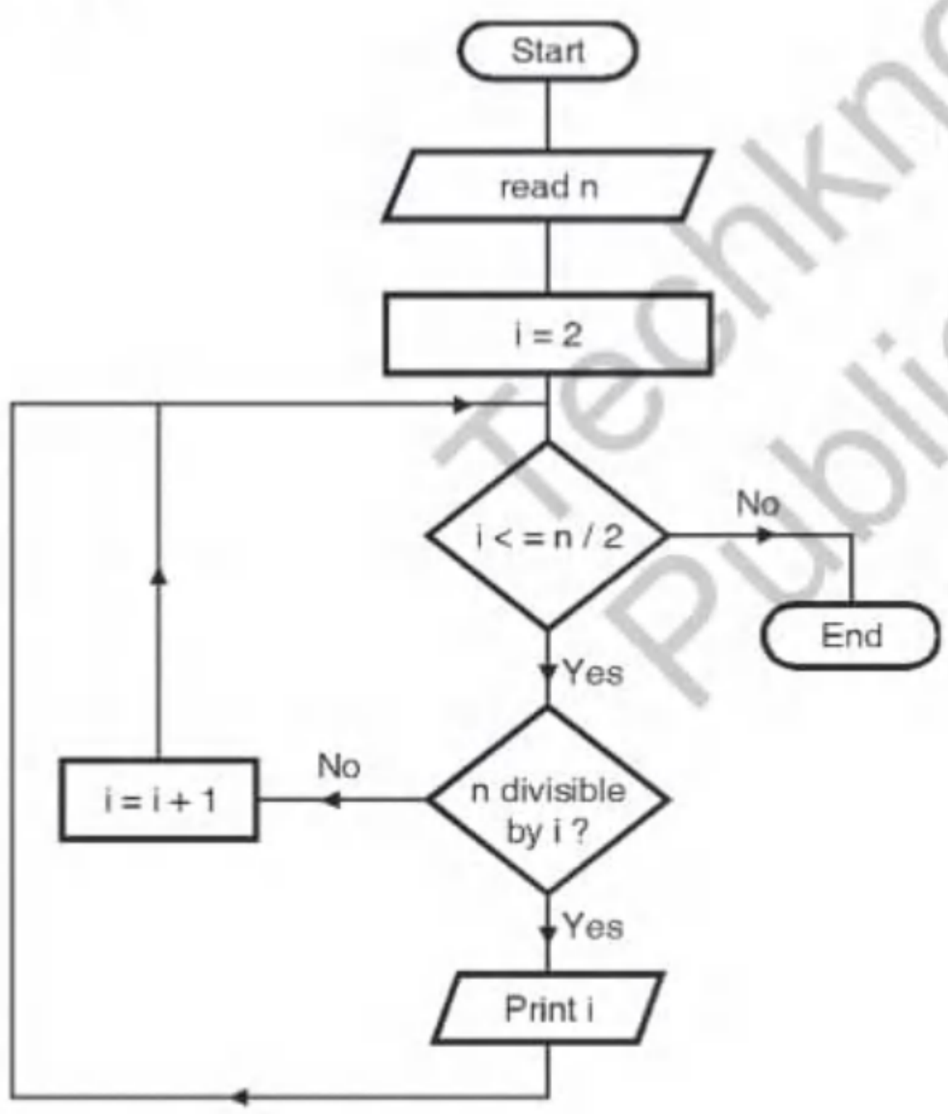


Fig. Ex. 1.1.3(a)

(b) HCF of two numbers

C-Pseudo code

```
void main()
{
    int x, y;
    read x, y;
    if(x < y)
        interchange(x, y);
        while(x is not divisible by y)
            Assign y to x and x% y to y;
    Print y;
}
```

Flowchart

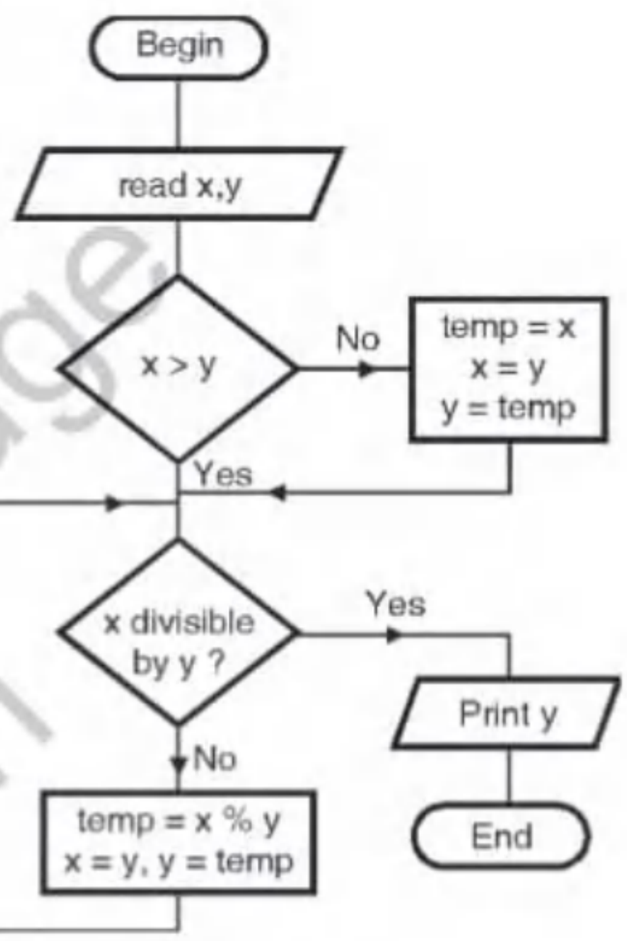


Fig. Ex. 1.1.3(b)

(c) Prime numbers between two given numbers x and y

C- Pseudo code

```
void main()
{
    int x, y, i, j, flag;
    read x, y;
    for(each i between x and y)
    {
        flag = 1;
        for(each j between 2 and i/2)
        {
            if(i is divisible by j)
                flag = 0 ;
        }
        if(flag is equal to 1)
            print i ;
    }
}
```

TechKnowledge Publications

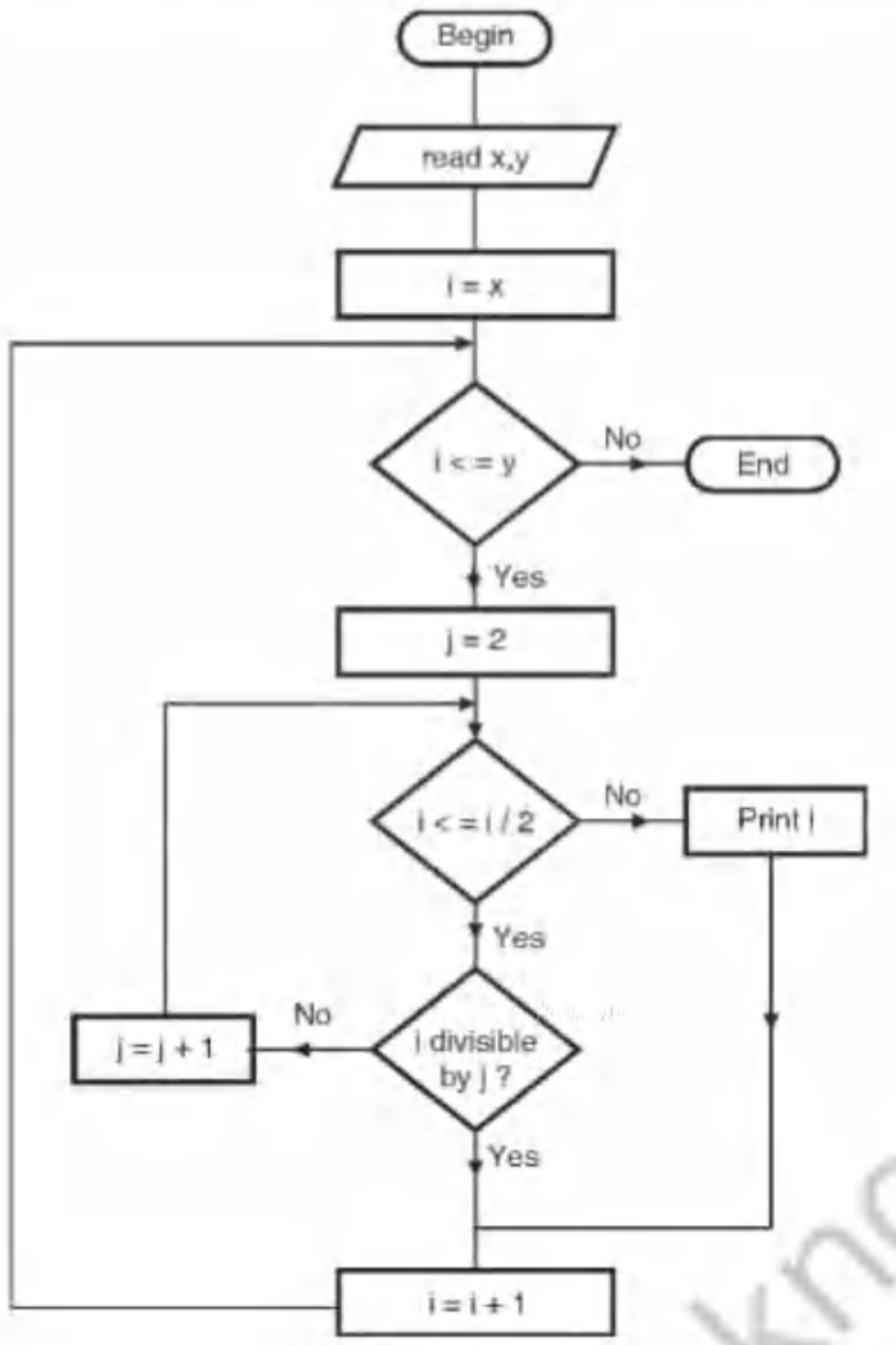


Fig. Ex. 1.1.3(c)

1.1.3 Relation between Data Structure and Algorithm

These is a close connection between the structuring of data and the synthesis of algorithm. Data structure and the algorithm should be thought of as a unit. To understand the above concept, we can consider the example of a list of n pairs of names and phone numbers. $(N_1, P_1), (N_2, P_2), (N_3, P_3)$ ....

If we have to write a program to print the phone number of a person whose name is entered as an input.

(i) If the list is sequenced on name then binary search algorithm will be able to search the name quickly.

(ii) If the list is not sorted then binary search algorithm will not work and the only choice left will be linear search.

In computer science, we study different kind of data objects. For each object, there are number of operations. Objects should be represented in way that makes efficient implementation of operation.

If a linear data is represented using linked organization then modification is less time consuming whereas searching a large list becomes inefficient. In sequential organization, accessing is efficient but modifications are costly.

1.2 Program Development

Program development is not simply writing a program in any programming language and get it working.

- A program is written for a client.
- A number of people work on a software project.
- A software project has to meet user requirements.
- Program has to perform satisfactorily.
- Program should be robust.
- Program should be easy to change to incorporate new features.
- Program should be easy to test and debug.

A program like any other product has to follow a definite sequence of steps. It will not only improve the productivity of a programmer but also improve the quality of the software. From inception of an idea for a software system, until it is implemented and delivered to a customer, the system undergoes gradual development and evolution.

Different phases of creating program

A sample software development cycle comprises of the following phases as shown in Fig. 1.2.1.

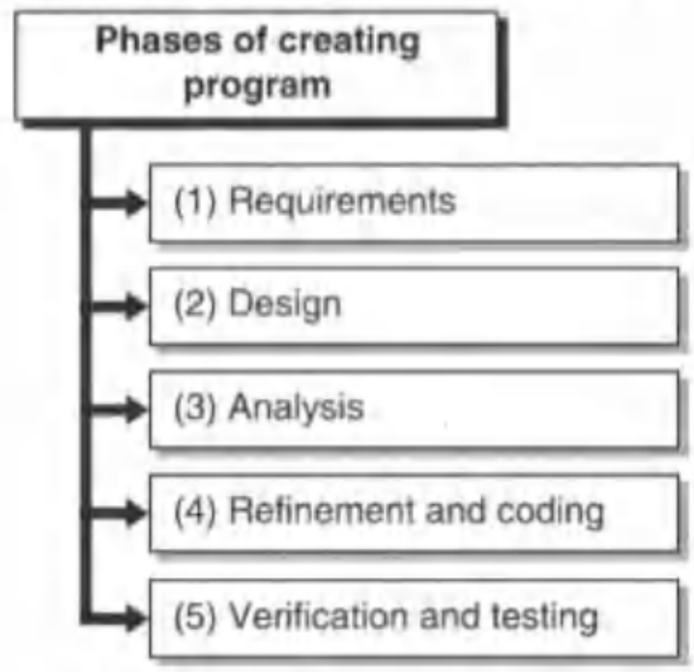


Fig. 1.2.1 : Phases of creating program

(1) Requirements

The purpose of this phase is to identify and document the exact requirements of the program. Requirements of the program written down in terms of end-user requirements.

- Input/output requirement for each case must be written down.
- Facts about the business process must be collected.
- Data must be collected on available files, decision points and how transactions are handled by the current system.

At the end of the requirements phase, programmer has a firm understanding of what is to be done?

(2) Design

Design of a program produces the detail that clearly describes how a software will meet the requirements identified during previous phase.

There may be several data objects (like a list, a polynomial, a matrix etc.). We may have to perform several operations on these objects. These objects may be interrelated. Design of these objects should permit efficient implementation of algorithms for various operation.

(3) Analysis

There may be many ways of meeting user requirements. There could be several algorithms for meeting a requirement. These alternate algorithms should be analyzed and the best performing one should be selected.

(4) Refinement and coding

In this phase, the programmer produces the actual code that will be delivered to the customer as the running program. Each data object must be represented using the programming language. Each algorithm must be coded. A large program should have;

(i) **Modularity :** Program modules should perform independent tasks. Interaction among them should be implemented properly.

(ii) **Abstraction :** Low level details of a program module should remain hidden and should provide an abstract interface.

(iii) **Usability :** A software program is usable if its users find it easy to use. The user interface is an important component of user friendliness.

(iv) **Verifiability :** A software system is verifiable if its properties can be verified easily. It is important to verify correctness or the performance of a software system. Verification can be performed by formal or informal analysis methods or through testing.

(v) **Maintainability :** Software maintainability, commonly refers to the modifications that are made to a program. These modifications can be made for fixing of bugs.

(vi) **Evolvability :** Like other engineering products, software products are modified over time to provide new functions or to change existing functions.

(5) Verification and testing

It is important that program gives correct output for all possible input combination. It may be possible to prove that a program is correct. Proof's about programs are similar to other proofs. Testing is an art of creating a set of sample test patterns. Program is run on the sample data and its output is compared with expected output. If the program fails to respond correctly then debugging is needed to determine what went wrong and how to correct it.

1.3 Algorithm Analysis (Using Step Count)

Q. Define and explain the following terms :
(a) Time Complexity
(b) Space Complexity **SPPU - Dec. 17, 2 Marks**

Q. What is complexity analysis of an algorithm ? **SPPU - May 19, 2 Marks**

Q. What is the complexity of an algorithm? Explain with an example. **SPPU - Dec. 19, 3 Marks**

Time Complexity of an Algorithm

Most often there are many algorithms for solving a problem. On what basis should we choose an algorithm? There are often two contradictory goals.

(i) Algorithm should be easy to understand, write and debug.

(ii) Algorithm should make efficient use of computer resources like CPU, memory etc.

When we write a program to be used a few times, goal (i) is most important. Cost of writing the program will have an upper hand over the cost of running the program, when the program is to be used many times, the cost of running the program and hence the goal (ii) should be given more weightage.

Analysing a program should quantify the requirement of computing resources during execution. Most important of these resources are computer time and memory. Analysis of algorithms focuses on computation of space and time complexity.

Space Complexity of an Algorithm

Space requirement means the space required to store input data either static or dynamic.

Space required on top of the system stack to handle recursion/function Call should also be considered.

Computing time, an algorithm might require for its execution, would normally depend on the size of the input.

```
#include < stdio.h>
void main( )
{
    int, i, n, sum, x;
    sum = 0 ;
    printf("\n Enter no. of data to be added");
    scanf("%d", &n) ;
    for(i = 1; i < = n; i++)
    {
        scanf("%d", &x) ;
        sum = sum + x ;
    }
    printf("\n sum = %d", sum) ;
}
```

Space requirement = Space required to store the variables i, n, sum and x = 2 + 2+ 2 +2 = 8.

[An integer requires 2 bytes of memory space.]

Calculation of computation time

Statement	Frequency	Computation time
sum = 0	1	t_1
printf("\n Enter no. of data to be added");	1	t_2
scanf("%d", &n)	1	t_3
for(i = 1; i < = n; i++)	n + 1	$(n + 1)\, t_4$
scanf ("%d", &x)	n	nt_5
sum = sum + x	n	nt_6
printf("\n sum = %d", sum)	1	t_7

$$\text{Total computation time} = t_1 + t_2 + t_3 + n\,(t_4 + t_5 + t_6) + t_4 + t_7$$

$$T = n\,(t_4 + t_5 + t_6) + (t_1 + t_2 + t_3 + t_4 + t_7)$$

For large n, T can be approximated to

$$T \approx n\,(t_4 + t_5 + t_6) = kn$$

Where, $k = t_4 + t_5 + t_6$, a constant value.

Thus, $T = kn$ or $T \propto n$

(i) $t_1, t_2 \ldots\ldots t_7$ are computer dependent, on a faster computer, the execution time of an instruction will be less.

(ii) for(i = 1; i < = n; i++), will be executed n + 1 times and not n times. For all values of i between 1 and n, the condition i < = n will evaluate to true and when i becomes n+1, the condition i < = n will evaluate to false. Thus the above instruction will be executed n + 1 times.

Ex. 1.3.1 : Determine the frequency count for all statements in the following program segment.

```
1.  i = 1;
2.  while( i < = n)
    {
    }
3.  x = x + 1;
4.  i = i + 1 ;
```

Soln. :

Statement No.	Frequency
1	1
2	n + 1
3	n
4	n

Ex. 1.3.2 : Determine the frequency counts for all statements in the following program segment.

```
1.  for(i = 1; i < = n; i++)
2.  for(j = 1; j < = i; j++)
3.  x = x +1 ;
```

Soln. :

		Frequency		
		Statement No.		
		1	2	3
i = 1	j = 1	1	1	1
	2		1	0
		1	2	1
i = 2	j = 1	1	1	1
	j = 2		1	1
	j = 3		1	0
		1	3	2
i = n	j = 1	1	1	1
	j = 2		1	1
	j = n		1	1
	j = n+1		1	0
		1	n+1	n

Total

Fig. Ex. 1.3.2 : Program Trace

Frequency

Statement No. 1 $= n + 1$

Statement No. 2 $= 2 + 3 + ... + n + (n + 1)$

$= (1 + 2 + 3 + ... + n) + n$

$= n\frac{(n+1)}{2} + n = \frac{1}{2}(n^2 + 3n)$

Statement No. 3 $= 1 + 2 + ... n$

$= n\frac{(n+1)}{2} = \frac{1}{2}(n^2 + n)$

1.3.1 Measuring the Running Time of a Program (Time Complexity)

Running time of a program can be judged on the basis of factors such as :

1. Input to the program.
2. Size of the program.
3. Machine language instruction set.
4. The machine we are executing on.
5. Time required to execute each machine instruction.
6. The time complexity of the algorithm of the program.

1.3.2 Measurement of Growth Rate (Asymptotic Growth Rate)

1.3.2(A) Asymptotic Consideration

When considering time complexities $f_1(n)$ and $f_2(n)$ of two different algorithms for a given problem of size n, we need to consider and compare the behavior of the two functions only for large n. If the relative behavior of two functions for smaller values conflict with the relative behavior for larger values, then we ignore the conflicting behavior for smaller values. For Example consider the following functions.

$f_1(n) = 100n^2$, $f_2(n) = 5n^3$

Representing time complexities of two solutions of a problem.

Techknowledge Publication

n	$f_1(n)$	$f_2(n)$
1	100	5
5	2500	625
10	10000	5000
20	40000	40000

$f_1(n) \geq f_2(n)$ for $n \leq 20$

We would still prefer the solution having $f_1(n)$ as time complexity because,

$f_1(n) \leq f_2(n)$ for all $n \geq 20$

1.3.2(B) Constant Factor in Complexity Measure

Let us consider an algorithm having a timing complexity given by the function $f(n) = 100\ n^2$

[100 : A constant, n : Size of the problem].

The time required for solving a problem, depends not only on the size of the problem but also, on the hardware and software used to execute the solution. The effect of hardware and software on the time required may closely be approximated by a constant.

Suppose, a new computer executes a program two times faster than another computer. Then irrespective of the size of the problem, the new computer solves the problem roughly two times faster than the computer. Thus we conclude that the time requirement for execution of a solution, changes by a constant factor on change in hardware or software.

An important consequence of the above discussion is that if the time taken by one machine in executing a solution is of the order of n^2 (say), then time take by every machine is of the order of n^2. Thus, function different from each other by constant factor, when treated as time complexities, should not be treated as different. i.e. should be treated as complexity wise same.

Following functions have the same time complexities :

$f_1(n) = 5n^2$; $f_2(n) = 100n^2$;

$f_3(n) = 1000n^2$; $f_4(n) = n^2$

Time complexity of $f_1(n)$ = Time complexity of $f_2(n)$ = Time complexity of $f_3(n)$ = Time complexity of $f_4(n)$.

1.3.3 Notation O : (Pronounced as Big-Oh), ($O(n^2)$ is Pronounced as Big-Oh of n^2)

Provides asymptotic upper bound for a given functions. Let f(x) and g(x) be two functions then f(x) is said to be O(g(x)) if there exist two positive integer/real number constants C and K such that;

$f(x) \leq Cg(x)$ for all $x \geq K$

The constant k is due to asymptotic consideration and the constant 'C' for hardware/ software environment.

Ex. 1.3.3 : $f(x) = x^2 + 5x$, $g(x) = x^2$ let us take C = 2

Soln. :

x	$x^2 + 5x$	$2x^2$
1	5	2
2	14	8
5	50	50

$f(x) \leq C\ x^2$

For $x \geq 5$

(asymptotic behavior)

$f(x) \leq Cg(x)$ for all $x \geq K$ where C = 2 and K = 5

Ex. 1.3.4 : For the function defined by $f(x) = 5x^3 + 6x^2 + 1$ show that $f(x) = O(x^3)$

Soln. :

$$f(x) = 5x^3 + 6x^2 + 1 \leq 5x^3 + 6x^3 + 1x^3$$

for all $x \geq 1$ or, $f(x) \leq 12x^3$ for $x \geq 1$

(by replacing each term of x by the highest degree term x^3)

there exist C = 12 and K = 1 such that

$f(x) \leq C\ x^3$ for all $x \geq K$

Hence f(x) is $O(x^3)$.

Note : Time complexity of a polynomial is same as the time complexity of the highest degree term.

Calculating the running time of a program

Ex. 1.3.5 : [A program fragment without a loop] x = x + 1.

Soln. :

The timing complexity is of constant order and it is represented by O(1).

Ex. 1.3.6 :

```
1 : for(i = 1,; i < = n; i++)
    2 : x = x + 1 ;
```

Soln. : Step no. 1 will be executed n + 1 times whereas the step no. 2 will be executed n time. With the assumption that each step requires a constant time, the time required to execute the above program will be of the order of 2n + 1 = O(n).

$$f(n) = 2n + 1$$
$$f(n) \leq Cn \text{ for } C = 3 \text{ and } n \geq 1$$

Conclusion : A loop without nesting has a timing complexity O(n).

Ex. 1.3.7 :

```
for(i = 1; i < = n; i++)
    for(j = 1; j  < = n; j++)
        x = x + 1
```

Soln. : Since, the statement x = x + 1 will be executed n^2 times, the timing complexity of the above program segment is $O(n^2)$.

Ex. 1.3.8 : A large program

```
n = 500 ;                    } O(1) Timing Complexity
x = 1;                       } O(1)
y = 2 ;                      } O(1)
for(i = 1; i < = n; i++ )    }
    x = x + 1 ;              } O(n)
for(j = 1; j < = n; j++)     }
    y = y + 1 ;              } O(n)
for(i = 1; i < = n; i++)     }
for(j = 1; j < = n; j++)     } O(n²)
    x = y + 1;               }
```

Soln. :

Hence, the combined time complexity is,

$O(\max(O(1), O(1), O(1), O(n), O(n), O(n^2))) = O(n^2)$

$$\text{if } f(x) = 5x^3 + 6x^2 + 1$$
$$O(f(x)) = x^3 - \text{proved earlier}$$

Conclusion : if $T_1(n)$ and $T_2(n)$ are the running times of two program fragments P1 and P2 and that $T_1(n)$ is O(f(n)) and $T_2(n)$ is O(g(n)). Then the running time of the complete program P1 followed by P2 is O(max(f(n), g(n)). For finding the timing complexity of a program, consider the program fragment having the highest timing complexity.

Ex. 1.3.9 : Calculate the time complexity for the following program segment.

```
for(i = 1; i < n; i++)
for(j = 0; j < n – i; j++)
if(a[j] > a[j + 1])
{
    temp = a[j] ;          } Statements
    a[j] = a[j+1] ;        } for interchange
    a[j + 1] = temp ;      } of a[j] and a[j +1]
}
```

Soln. : For calculation of timing complexity, it is sufficient to express number of interchanges in terms of n (number of data).

Program trace	Number of interchanges
i = 1, j varies from 0 to n – 2	n – 1
i = 2, j varies from 0 to n – 3	n – 2
.	.
.	.
.	.
.	.
i = n – 1, j varies from n – 1 to n – 1	1

$$\text{Total no. of interchanges} = 1 + 2 + \ldots + (n-2) + (n-1)$$
$$= \frac{n(n-1)}{2} = \frac{1}{2}(n^2 - n)$$
$$= O(n^2)$$

[Consider, the most significant term]

TechKnowledge Publications

Ex. 1.3.10 : Calculate the time complexity for the following program segment

```
int A(int n)
{
    if(n <= 1)
        return(1);
    else
        return(A(n – 1) + A(n – 1));
}
```

Soln. :

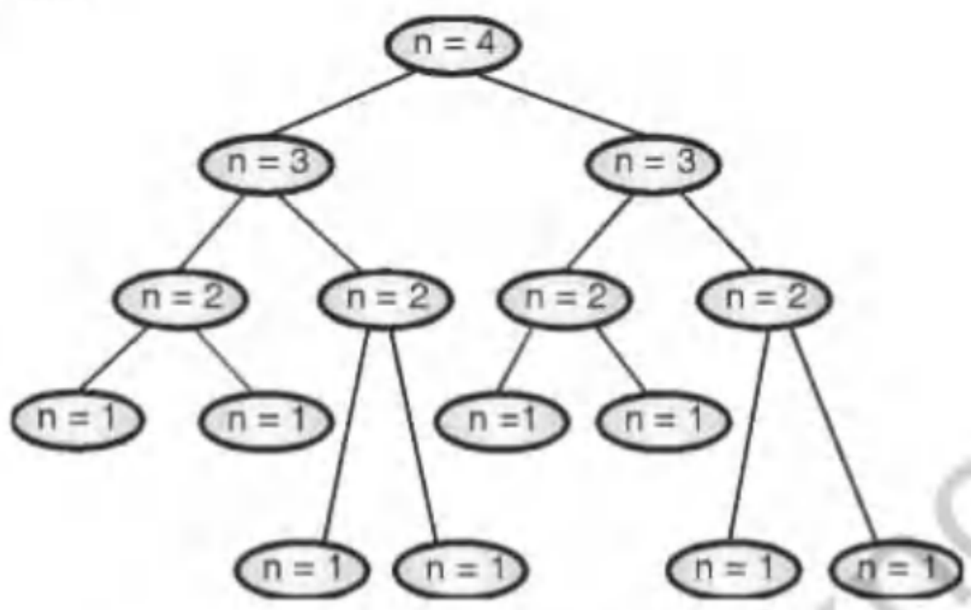


Fig. Ex. 1.3.10 : Recursion tree for n = 4

When the function A is called with the initial value of n = 4, it calls the same function recursively twice with n = 3

$$\text{thus, } f(n) = 2f(n-1) \quad \text{if } n > 1$$

$$f(n) = 1 \quad \text{if } n <= 1 \text{ (Best case)}$$

$$\text{Since } f(n) = 2f(n-1) = 2^2 f(n-2)$$

$$= 2^{n-1} f(n-(n-1))$$

$$= 2^{n-1} f(1) = 2^{n-1} = O(2^n)$$

1.3.4 Best Case, Worst Case and the Average Case Behaviour

Many programs do not produce same timing complexity in every case. Consider the problem of searching an element in an array with 10 elements.

5	1	9	6	2	11	13	6	7	16

Element 5 will be found in one attempt

Element 16 will require 10 comparisons to locate

Best case behaviour = 1 comparison (when the element to be searched is in the beginning)

Worst case behaviour = n comparisons (where n is the number of elements and the element to be searched is at the end of the array)

Average case behaviour = number of comparisons will be n/2 (from probability).

Ex. 1.3.11 : Calculate the worst case time complexity of the following program (Function for binary search) :

```
int binsearch(int a[ ], int i, int j, int x)
{
    int k;
    k = (i + j)/2;
    while(i <= j)
    {
        if(x > a [k])
            i = k + 1;
        else
            if(x < a [k])
                j = k – 1;
            else
            return(k);
    }
    k = (i + j)/2;
    return(– 1);
}
```

OR Comment on time analysis.

SPPU - May 17, May 18, 2 Marks

Soln. :

- Binary search requires that elements are sorted in ascending order.
- i is the starting index (i.e. 0), j is the index of the last element (n – 1).
- k gives the index of the centre element

2	5	9	8	11	13	15	←	element
0	1	2	3	4	5	6	←	index
i			k			j		
↑			↑			↑		
i = 0,			j = 6			K (0 + 6)/2 = 3		

Element to be searched is x.

- Inside the while-loop, if the element x is found at the location k (centre), we return k.
- Inside the while-loop, if the element x is larger than the centre element, we select the second half of the array.
- Inside the while-loop, if the element x is smaller than the centre element; we select the first half of the array.

2	5	9	8	11	13	15
↑ i = 0		First half	↑ Centre element k = 3	Second half		↑ j = 6
Element to be searched, x = 13						
11 ↑ i = 4	13	15 ↑ j = 6		Second half of the element		

- In order to calculate time complexity, binary search for n elements must be expressed in terms of binary search of fewer number (<n) of elements.

$$f(n) = 1 + f(n/2)$$

- If there are 128 elements, then the maximum number of elements between i and j after every iteration will be 64, 32, 16, 8, 4, 2, 1, 0, – 1

$$128 = 2^7$$

- Number of iteration required (say h) = 9 = 7 + 2

$$\therefore \quad 2^{h+2} = n$$

$$\text{or } \log(2^{h+2}) = \log n$$

$$\text{or } h + 2 = \log n$$

$$\therefore \quad h = \log n - 2 = O(\log n)$$

Ex. 1.3.12 : Show that $f(x) = O(x^3)$ if function f(x) is defined as $f(x) = 5x^3 + 6x^2 + 1$. **SPPU - May 17, 3 Marks**

Soln. :

$$\text{If} \quad f(x) = 5x^3 + 6x^2 + 1$$

$$\text{Then} \quad f(x) \le 5x^3 + 6x^3 + x^3$$

$$\le 12x^3$$

$$5x^3 + 6x^2 + 1 = O(x^3) \text{ as}$$

$$f(x) \le C \cdot x^3 \text{ for all } x \ge K \text{ where}$$

$$C = 12 \text{ and } K = 1$$

1.3.5 Different Algorithm Asymptotic Notation

Q. Explain Asymptotic notations-Big O, Omega and Theta with one example of each. **SPPU - Dec. 16, Dec. 17, May 18, 6 Marks**

Q. Explain the notations used in the complexity analysis. **SPPU - May 19, 4 Marks**

Q. Explain the asymptotic notations. Plot the graph for Big O, Omega and Theta with respect to input vs time. **SPPU - Dec. 19, 3 Marks**

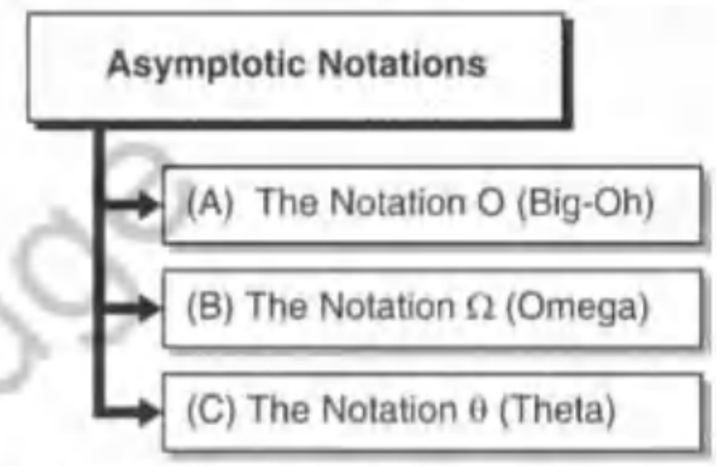


Fig. 1.3.1 : Asymptotic Notation

1.3.5(A) The Notation O (Big-Oh)

The function f(x) is said to be O(g(x)) (pronounced as big-oh) if there exist two positive integer/real number constants C and K such that

$$f(x) \le Cg(x) \text{ for all } x \ge K$$

Example

$$\text{if } f(x) = 2x^3 + 3x^2 + 1$$

$$\text{then } f(x) \le 6x^3 \text{ for all } x \ge 1$$

$$\therefore 2x^3 + 3x^2 + 1 = O(x^3) \text{ as}$$

$f(x) \le C.\ x^3$ for all $x \ge K$ where C = 6 and K = 1

The notation O provides asymptotic upper bound for a given function.

The purpose of the asymptotic growth rate notations and functions denoted by them is to facilitate the recognition of essential character of a complexity function through some simpler function.

1.3.5(B) The Notation Ω (Omega)

This provides asymptotic lower bound for a given function. A function f(x) is said to be Ω(g(x)) (pronounced as big-omega) if there exist two positive integer/real number constants C and K such that

$$f(x) \ge C\,(g(x)) \text{ whenever } x \ge K$$

TechKnowledge Publications

Example

$$\text{if } f(x) = 2x^3 + 3x^2 + 1$$

$$\text{then } f(x) \geq C\,x^3 \text{ for all } x \geq 1,\ C = 1$$

$$\therefore 2x^3 + 3x^2 + 1 = \Omega(x^3)$$

1.3.5(C) The Notation θ (Theta)

- This provides simultaneously both asymptotic upper bound and asymptotic lower bound for a given function.
- A function f(x) is said to be θ (g(x)) (pronounced big theta) if, there exist positive constants C_1, C_2 and K such that $C_2\, g(x) \leq f(x) \leq C_1\, g(x)$ for all $x \geq K$.

Example

For any two functions f(x) and g(x), f(x) = 0(g(x))

If and only if f(x) = O(g(x)) and f(x) = Ω(g(x))

1.3.6 Graphical Interpretation of Different Notations

- The big-Oh notation O provides asymptotic upper bound for a given function.
- The notation Ω, provides asymptotic lower bound for a given function.
- The notation θ, provides both asymptotic upper bound and asymptotic lower bound for a given function.

The graphical interpretation of the three asymptotic notations can be understood with the help of the function, $f(x) = x^3 + 5x + 10$

$$x^3 + 5x + 10 \leq 16\,x^3 \text{ for } x \geq 1$$

Therefore, $x^3 + 5x + 10$ is $O(x^3)$

$$x^3 + 5x + 10 \geq x^3 \text{ for } x \geq 0$$

Therefore, $x^3 + 5x + 10$ is $\Omega(x^3)$

$x^3 \leq x^3 + 5x + 10 \leq 16\,x^3$ for $x \geq 1$

Therefore, $x^3 + 5x + 10$ is $\theta(x^3)$

Table 1.3.1 : Running time of several functions

x	x^3	$x^3 + 5x + 10$	$16\,x^3$
0	0	10	0
1	1	16	16
2	8	28	128
3	27	52	432

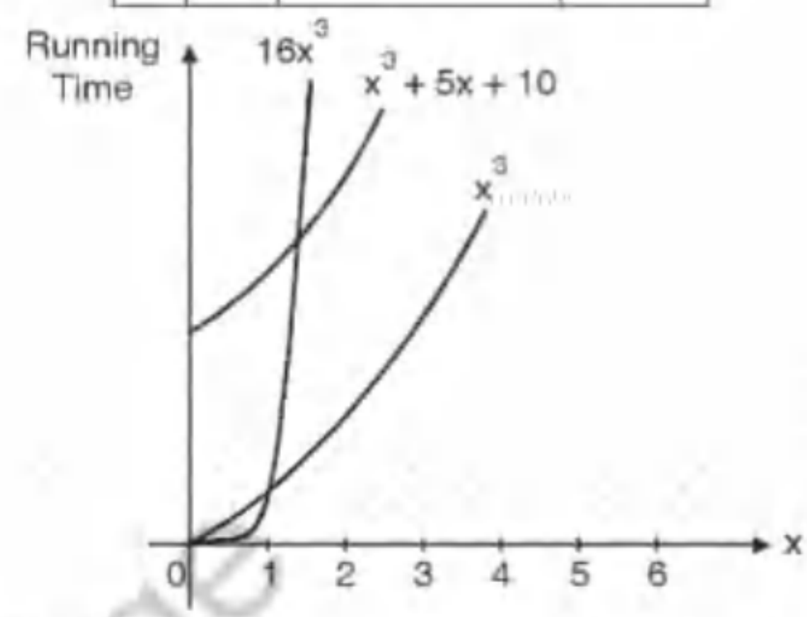


Fig. 1.3.2 : Graphical representation of various functions

x^3 gives the lower bound of $x^3 + 5x + 10$

$16\,x^3$ gives the upper bound of $x^3 + 5x + 10$

1.4 Programming Constructs-linear, Quadratic, Cubic and Logarithmic

Algorithmic complexity, also called running time, refers to the number of steps a program takes as a function of the size of its inputs. Some common running times are :

1. Constant O(1)
2. Linear O(n)
3. Quadratic $O(n^2)$
4. Cubic $O(n^3)$
5. Logarithmic O(log n)

1.4.1 Constant-Time Algorithms [O(1)]

The algorithm takes the same amount of time regardless of its inputs.

Examples

1. Finding sum of two numbers.
2. Finding the first element of a list.

```
int return_first(int a[ ], int n)
{
    return(a[0]);
}
```

1.4.2 Linear-Time Algorithm [O(n)]

The algorithm takes a number of steps directly proportional to the size of the input. In short, if the size of the input doubles, the number of steps doubles.

Examples

1. Finding sum of the elements of a list.

```
int sum(int a[ ], int n)
{
    int i, result = 0;
    for(i = 0; i < n; i++)
        result = result + a[i];
    return(result);
}
```

2. Finding largest number in a list.

```
int largest(int a[ ], int n)
{
    int i, largest;
    largest = a[0];
    for(i = 1; i < n; i++)
        if(largest < a[i])
            largest = a[i];
    return(largest);
}
```

1.4.3 Quadratic-Time Algorithm [O(n^2)]

The algorithm takes a number of steps directly proportional to n^2, where n is the size of the input.

Examples

1. A nested loop.

```
sum = 0;
for(i = 0; i < n; i++)
for(j = 0; j < n; j++)
sum = sum + j;
```

2. Finding sum of the elements of a 2-D array.

```
sum = 0;
for(i = 0, i < n; i++)
for(j = 0; j < n; j++)
sum = sum + a[i] [j];
```

1.4.4 Cubic-Time Algorithm [O(n^3)]

The algorithm takes a number of steps directly proportional to n^3, where n is the size of the input.

Examples

1. A double nested loop.

```
sum = 0;
for(i = 0; i < n; i++)
for(j = 0; j < n; j++)
for(k = 0; k < n; k++)
sum = sum + k;
```

2. Finding sum of the elements of a 3-D array.

```
sum = 0;
for(i = 0; i < n; i++)
for(j = 0; j < n; j++)
for(k = 0; k < n; k++)
sum = sum + a[i] [j] [k];
```

1.4.5 Logarithmic-Time Algorithm [log(n)]

The algorithm takes a number of steps directly proportional to log(n), where n is the size of the input.

Examples

1. Binary search
2. Searching an element in a binary search tree.

Review Questions

Q. 1 What is time complexity of an algorithm? Explain with suitable example. **(6 Marks)**

Q. 2 Discuss the different phases of creating program. **(6 Marks)**

Q. 3 What do you mean by asymptotic notations ? Explain with examples. **(6 Marks)**

Q. 4 Explain Big-oh, omega and theta notation with example. **(6 Marks)**

Q. 5 What is space complexity of an algorithm ? **(3 Marks)**

Q. 6 Explain asymptotic notations with examples. **(5 Marks)**

❑❑❑

Technknowledge Publication

CHAPTER 2

Unit I

Data Structures and Algorithmic Strategies

Syllabus :

Introduction : From Problem to Data Structure (Problem, Logic, Algorithm, and Data Structure). Data Structures : Data, Information, Knowledge, and Data structure, Abstract Data Types (ADT), Data Structure Classification (Linear and Non-linear, Static and Dynamic, Persistent and Ephemeral data structures)

Algorithmic Strategies : Introduction to algorithm design strategies - Divide and Conquer and Greedy strategy.

Exemplar/Case Studies : Multiplication technique by the mathematician Carl Friedrich Gauss and Kasturaba's algorithm for fast multiplication.

2.1 Data

Q. Define and explain the terms Data.
SPPU - May 18, 1 Mark

Data is a collection of numbers, alphabets and symbols combined to represent information. A computer takes raw data as input and after processing of data it produces refined data as output. We might say that computer science is the study of data.

Atomic data are non-decomposible entity. For example, an integer value 523 or a character value 'A' cannot be further divided. If we further divide the value 523 in three digits '5', '2' and '3' then the meaning may be lost.

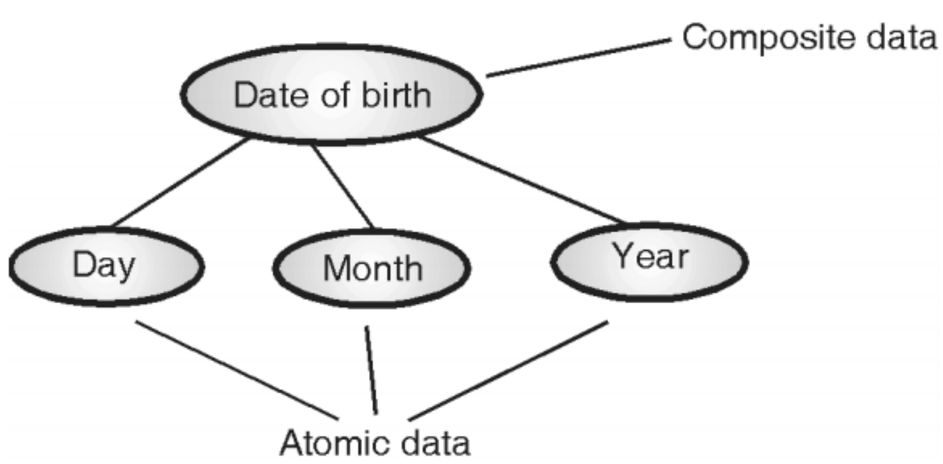


Fig. 2.1.1

Composite data : It is a composition of several atomic data and hence it can be further divided into atomic data.

For example, date of birth (say 15/3/1984) can be separated into three atomic values. First one gives the day of the month, second one gives the month and the last one is the year.

2.1.1 Data Types

A data type is a term which refers to the kind of data that variables may hold in a programming language.

Example : int x ; [x can hold, integer type data]

Every programming language has a method for declaring a set of variable of a particular type. A value stored in a variable cannot be interpreted properly without knowing its type.

A byte of information stored in computer memory may represent an integer value, a character value, a BCD (Binary Coded Decimal) value or a Boolean value. Therefore, it is necessary that the value stored in memory must be treated as of a particular type and interpreted accordingly.

2.1.2 Abstract Data Types (ADT)

Q. Define and explain the term ADT.
SPPU - Dec. 17, May 19, 1 Mark

The concept of abstraction is commonly found in computer science.

Technical Publication

A big program is never written as a monolithic piece of program, instead it is broken down in smaller modules (may be called a function or procedure) and each module is developed independently.

When the program is hierarchical organized as shown in the Fig. 2.1.2, then the "main program" utilizes services at the functions appearing at level 1. Similarly, functions written at level 1 utilize services of functions written at level 2. Main program uses the services of the next level function without knowing their implementation details. Thus a level of **abstraction** is created. When an abstraction is created at any level, our concern is limited to "what it can do" and not "how it is done".

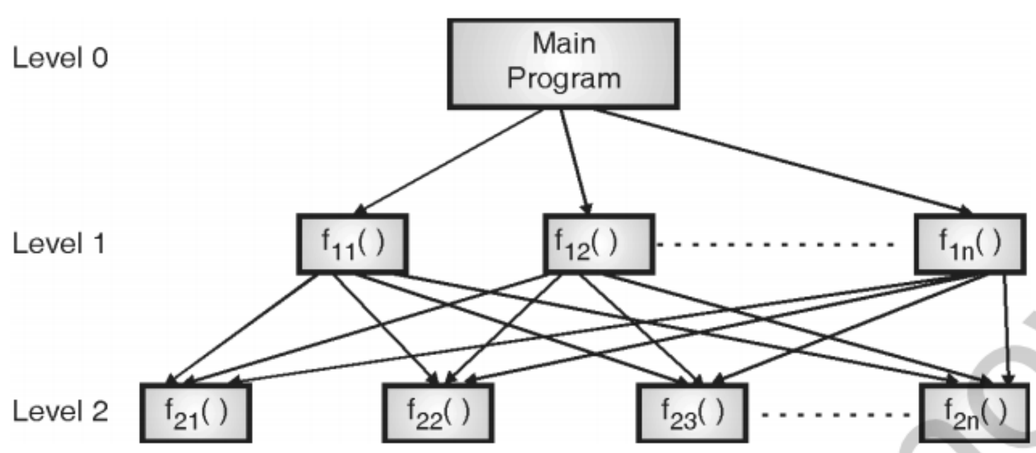


Fig. 2.1.2 : Hierarchical organized program

Abstraction in case of data

Abstraction for primitive types (char, int, float) is provided by the compiler. For example, we use integer type data and also, perform various operations on them without knowing them :

(1) Representation

(2) How various operations are performed on them.

Example

```
int x, y, z;
x = 13;
```

Constant 13 is converted to 2's complement and then stored in x. Representation is handled by the compiler x = y + z ;

Meaning of the operation '+' is defined by the compiler and its implementation details remain hidden from the user.

Implementation details (representation) and how various operations are implemented remain hidden from the user. User is only concerned about, how to use these operations.

Objects such as lists, sets and graphs along with associated operations, can be viewed as abstract data type. Integer, char, real are primitive data types and there are set of operations associated with them.

For the set ADT (Abstract Data Type), we might have operations like union, intersection, size and complement. Once the data type set is defined (representation and associated functions) then the ADT set can be used in any application program.

ADT for an array

The Abstract Data Type "ARRAY"

Arrays are stored in consecutive set of memory locations. An array can be thought of as a set of pair, index and value. For each index which is defined there is a value associated with that index. There are two operations permitted on 'ARRAY' data structure. These two operations are retrieve and store.

ADT ARRAY can be declared as below :

Structure ARRAY(value, index)

declare

CREATE( ) → array

RETRIEVE(array, index) → value

STORE(array, index, value) → array

The function CREATE( ) produces an empty array.

The function RETRIEVE( ) takes as input an array and an index, and either returns the appropriate value or an error.

The function STORE( ) is used to enter new index-value pairs.

The Abstract Data Type "List"

A list is a sequence of 0 or more elements of a given type (element type, could be integer, float etc.). Such a list is often represented as : $a_1, a_2 \ldots\ldots\ldots a_n$

Technknowledge Publications

n = Number of elements in the list

a_i = i^{th} element of the list

- When n = 0, the list is empty having no element.
- Each element other than a_1 has a predecessor
- Each element, other than a_n has a successor.

In order to form an ADT from the mathematical notation of a list, we must represent the list and define a set of operations on objects of type List.

Representation of a List

A List can be represented in 'C' using a structure.

```
typedef struct List
{
    int data[50] ;
    int n ;
}List ;
```

- A List of maximum of 50 integer type elements
- n is the actual number of elements in the List.

A set of representative operations on a List

1. Insert(L, X, P) : Insert x at position P in List L.
2. Locate(X, L) : This function returns the position of element X on List L.
3. Retrieve(L, P) : This function returns the element at position P on list L.
4. Delete(L, P) : Delete the element at position P of the List L.
5. MAKENULL(L) : Creates an empty List L.
6. PrintList(L) : Print the elements of L in the order of occurrence.

2.1.3 Data Object

- Data object refers to a set of variables used in a program.
- It is a place where the data values can be stored, retrieved and manipulated.
- Every data object is associated with data type (type of value that can be stored).
- A complex object (structure type) can have a number of attributes.
- Data object can also be viewed as a runtime instance of a data structure.

2.2 Data Structures

Q. Define and explain the term Data Structure.

SPPU - Dec. 17, May 18, May 19, 1 Mark

- A data structure is merely an instance of an ADT.
- An ADT or data structure is formally defined to be a triplet (D,F,A) where "D" stands for a set of Domains, "F" denotes the set of operations and "A" represents the axioms defining the functions in "F".
- An example of the data structure "Natural Number (NATNO)".

Structure of Natural Number (NATNO)

Operations

1. ZERO( ) → natno
2. ISZERO(natno) → boolean
3. SUCC(natno) →natno
4. ADD(natno, natno) → natno
5. EQUAL (natno, natno) →boolean

Axioms

For all x, y ∈ natno let

6. ISZERO(ZERO) is true
7. ADD(ZERO,Y) is Y
8. EQUAL(x, ZERO), if ISZERO(x) then true else false

D = { natno, boolean}

F = { ZERO, ISZERO, SUCC, ADD, EQUAL }

A = { Line no. 6 to 8 of the structure NATNO }

Technknowledge Publications

2.2.1 Types/Classification of Data Structures

2.2.1(A) Primitive and Non-Primitive

Primitive

The integers, reals, logical data, character data, pointers and reference are primitive data structures. These data types are available in most programming languages as built in type. Data objects of primitive data types can be operated upon by machine level instructions.

Non-Primitive

These data structures are derived from primitive data structures. A set of homogeneous and heterogeneous data elements are stored together.

Examples of Non-primitive data structures are Array, structure, union, linked-list, stack, queue, tree, and graph.

Data Structures Operations

Some at the most commonly used operations that can be performed on data structures are shown in Fig. 2.2.1.

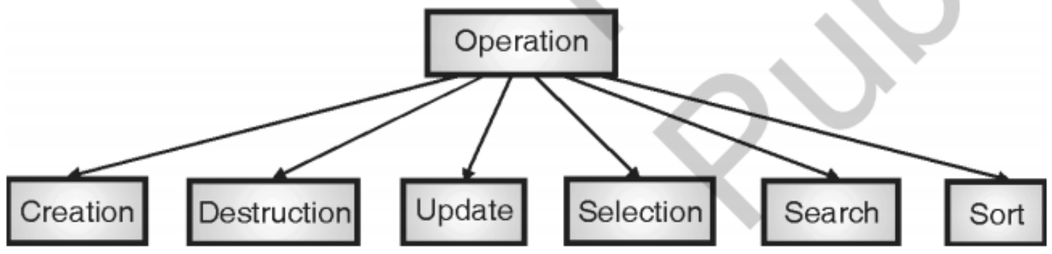


Fig. 2.2.1 : Data structure operations

2.2.1(B) Linear and Non-Linear

Q. Differentiate between linear and non-linear data structure with example. **SPPU - May 17, 3 Marks**

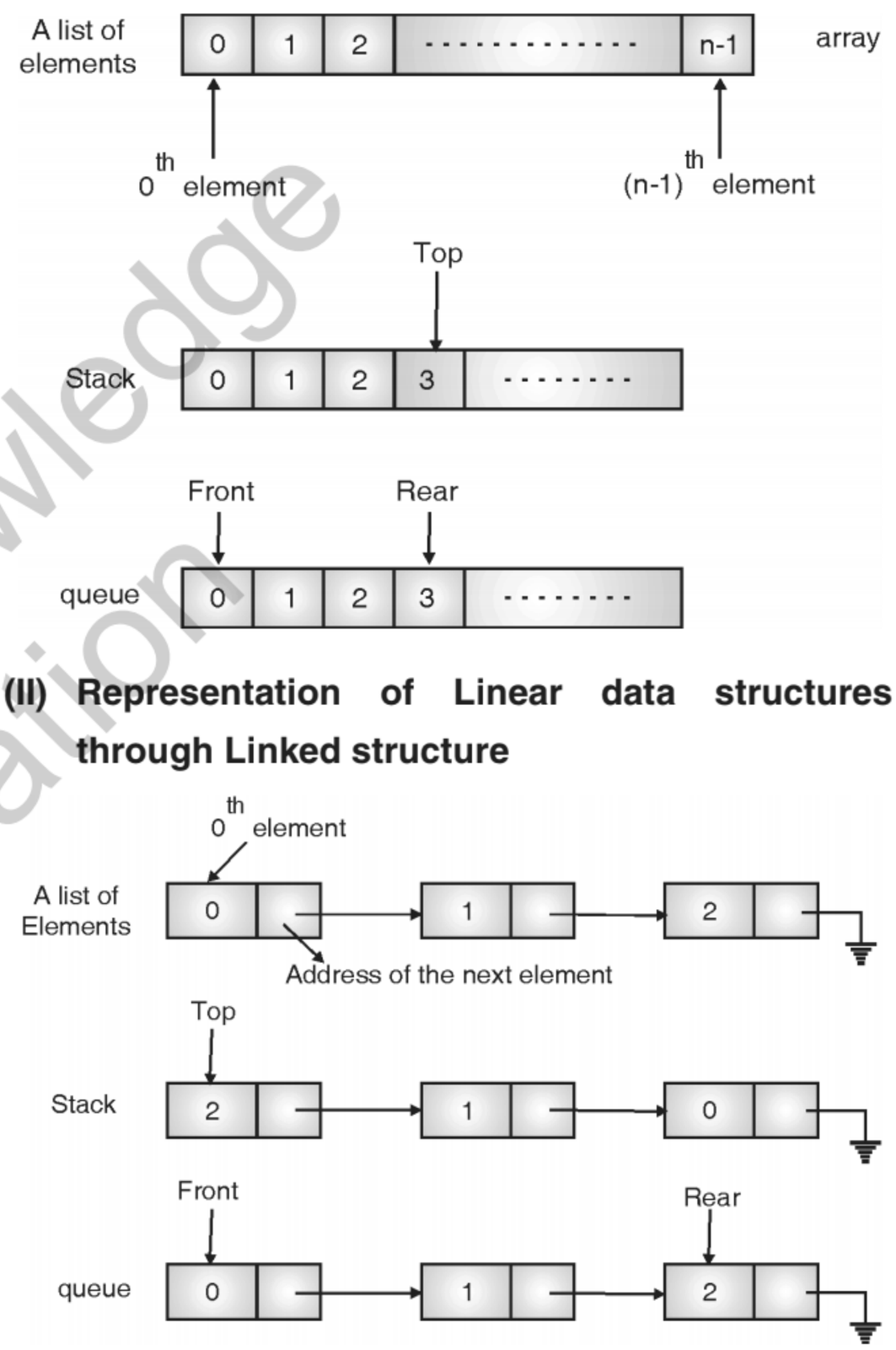
Q. Define and explain the term Linear data structure. **SPPU - Dec. 17, May 18, 1 Mark**

Q. Write a short note on linear and Non-linear data structure with example. **SPPU - Dec. 19, 4 Marks**

Linear

Elements are arranged in a Linear fashion (one dimension). All one to one relation can be handled through Linear data structures. Lists, stacks and queues are examples of linear data structure.

(I) Representation of Linear data structures in an array

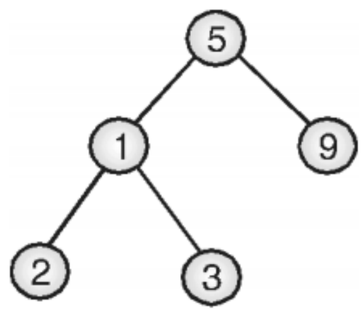


(II) Representation of Linear data structures through Linked structure

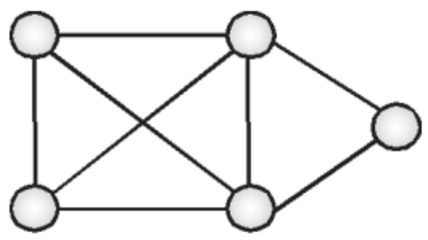
Non-Linear

All one to many, many to one or many to many relations are handled through non-linear data structures. Every data element can have a number of predecessors as well as successors. Tree graphs and tables are examples of non-linear data structures.

TechKnowledge Publications



(a) Tree

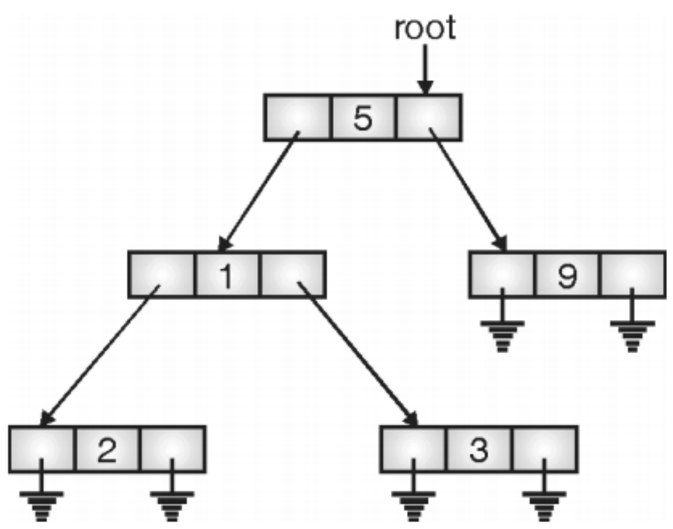


(b) Graph

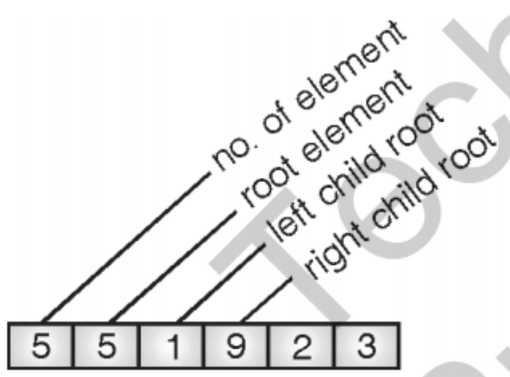
5	9	1
6	4	13
2	5	0
9	6	11

(c) Table

Fig. 2.2.2 : Non-Linear data structures



(a) Representation of the binary tree through linked structure



(b) Representation of the binary tree through an array

Fig. 2.2.3 : Representation of tree of Fig. 2.2.2(a)

2.2.1(C) Static and Dynamic

Q. Explain static and dynamic data structures with examples. **SPPU - Dec. 18, 4 Marks**

Static Memory Allocation

In case of static data structure, memory for objects is allocated at the time of loading of the program. Amount of memory required is determined by the compiler during compilation.

Example : int a[50]

Memory for the array 'a' of 50 elements will be allocated at the time of loading of the program. It may not always be possible to fix the size of the array in advance. Amount of data to be handle is often determined by the user and not by the programmer. Our initial judgment of size, if wrong, may cause failure of program (due to overflow) or wastage of memory space.

- Static data structure causes under utilization of memory (in case of over allocation).
- Static data structure may cause overflow (under allocation).
- No reusability of allocated memory.
- Difficult to guess the exact size of data at the time of writing of program.

Dynamic Memory Allocation

In case of dynamic data structures, the memory space required by variables is calculated and allocated during execution. Dynamic memory is managed in 'C' through a set of library function.

Allocating a block of memory in "C"

```
ptr = (cast_type) malloc(byte_size) ;
```

The "malloc( )" returns a pointer of (cast_type) to an area of memory with size, (byte_size)

Example : x = (int *) malloc(100 * sizeof(int)) ;

- A Linear data structure can be implemented either through static or dynamic data structures. Static data structured is preferred.
- All Linked structures are preferably implemented through dynamic data structures.
- A dynamic data structure provides flexibility in adding, deleting or rearranging data objects at run time.
- Additional space can be allocated at run time.
- Unwanted space can be released at run time.
- It gives re-usability of memory space.

TechKnowledge Publications

2.2.1(D) Persistent and Ephemeral

Persistent

- A data structure is said to be persistent if it can be accessed but cannot be modified.
- Any operation on such data structure creates two versions of data structures. Previous version is saved and all changes are made in the new version. Thus an operation that changes the data will create :
  (1) A copy of old instance of data structure with original values.
  (2) A copy of new instance of data structure with updated values.
- Functional data structures are persistent.

Ephemeral

- If we are able to create data cells and modify their contents, we can create **ephemeral** data structures. These are data structures that change over time. The main advantage of such data structures is their ability to maintain state as a shared resource among many routines. Another advantage in some cases is the ability to write code that is more time-efficient than purely functional code.
- The **disadvantage** of ephemeral data structures are error and complexity. Our program can accidentally and irreversibly change the contents of a data structure.

2.2.1(E) Relationship among Data Object, Data Type, Data Structure and Data Representation

A data type is a term which refers to the kind of data that variables may hold in a programming language. Every programming language has a method for declaring a set of variables of a particular type. A value stored in a variable cannot be interpreted properly without knowing its type. A byte of information stored in computer memory may represent an integer value, a character value, a BCD value or a Boolean value. Therefore, it is necessary that the value stored in memory must be treated as of a particular type and interpreted accordingly.

Data object refers to a set of variables used in a program. Every data object is associated with data type. A complex object can have a number of attributes. Data object can also be viewed as a runtime instance of a data structure. Every type of data is converted into a stream of bits before it is stored in memory. Integers are represented as a signed binary numbers. Floating point numbers are represented in IEEE format.

A data structure is merely an instance of an ADT. In most cases, a data structure is defined using structure and associated functions are written. A proper selection of data structure will determine the efficiency of the associated functions.

Ex. 2.2.1 : Differentiate between static memory and dynamic memory allocation.

Soln. :

Comparison of static and dynamic data structure

Sr. No.	Static Data Structure	Dynamic Data Structure
1.	Memory allocated at the time of loading.	Memory allocated at the time of running.
2.	Static data structure may cause overflow due to under allocation of memory.	No such problem in dynamic data structure.
3.	Under utilization of memory in case of over allocation of memory.	No such problem
4.	Access speed is higher no re-usability of memory.	Access speed is slow re-usable of memory is possible.

Tech Knowledge Publications

2.3 Algorithmic Strategy

Algorithm design is as important as its implementation. A knowledge of design helps in creating a good algorithm. The number of basic design strategies of algorithm is very small. There are many common types of algorithms used to solve problems. They use :

1. Divide and Conquer
2. The Greedy Method
3. Dynamic Programming
4. Backtracking

2.4 Divide and Conquer

Q. Explain divide and conquer strategy with example. **SPPU - May 17, 4 Marks**

Q. Explain the divide and conquer strategy with suitable example. **SPPU - May 18, 4 Marks**

Q. Explain the algorithmic strategy of divide and conquer. Explain its application in binary search. **SPPU - Dec. 18, 4 Marks**

In divide and conquer strategy, we split a problem into subproblems. Sub problems resulting from divide and conquer are of the same type as the original problem. These sub problems are solved recursively. Recursion will eventually terminate when the sub problem to be solved is small enough to be solved without further splitting. Now, there must be a method to combine sub solutions into a solution of the whole.

Thus, divide and conquer algorithms consist of two parts:

1. Divide : smaller problems are solved recursively.
2. The solution to the original problem is then formed form the solutions to the sub problems.

2.4.1 Binary Search

Q. Explain its application in binary search. **SPPU - Dec. 18, 2 Marks**

Consider the problem of determining whether an element x is present in a list. List is sorted in non-decreasing order.

Division of problems into subproblems

- Elements are stored in an array a[ ]
- i and j are indices of the first and the last element respectively. initially, i is 0 and j = n–1,where n is number of elements.
- Element be searched, x is compared with the centre element a[c] c = (i + j)/2).
- Now one of the three things could happen :
  x>a[c], right half of the elements are selected as sub problem. i.e., i = c+1, j = remains unchanged
- x<a[c], left half of the elements are selected as sub problem i.e. i = remains unchanged, j = c–1
- x is equal to a[c], solution is known and further generation of sub problems is not required.
- i will eventually become greater than j, if the element x is not present in the list.

Combining subsolutions

In our algorithm, we generate only one sub problem(left half or right half) at each level of recursion. Solution of sub problem is the solution of the original problem.

The function binsearch( ) has four inputs a[ ], i, j, x.

a[ ] : Array of sorted elements
i : Index of the first element
j : Index of the last element
x : Element to be searched.

At the conclusion of the procedure 1 is returned if x is not present, or the index of the center element is returned if the element is present.

```
int binsearch(int a[ ], int i, int j, int x)
{
    int c;
    if(i > j)
        return(–1);
```

TechKnowledge Publications

```
    c = ( i+ j)/2;
    if(x = = a[c] )
        return(c);
    if(x >a[c])
        return(binsearch(a, c+1, j, x));
    else
        return(binsearch(a, i, c–1, x));
}
```

2.4.2 Examples of Divide and Conquer Strategy

(A) Tower of Hanoi (B) Merge sort

(C) Quick sort (D) Finding of maxima

2.4.2(A) 'Divide and Conquer' Strategy for Tower of Hanoi

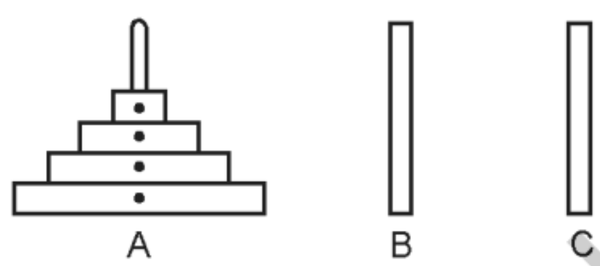


Fig. 2.4.1 : Disks are stacked on peg A in decreasing order of size

A tower of n disks is stacked in decreasing order on peg "A". These disks are to be transferred to peg "B" with the help of peg "C".

Following rules must be followed while transferring disks from one peg to another peg.

(a) Only one disk can be transferred at a time.

(b) At no time a larger disk can be placed on a smaller disk.

Division of n disk problem into sub problems

n disk problem can be converted into two n – 1 disks sub problems. Recursion continues as long as n > 0.

Step 1 : Move n – 1 disks from peg A to peg C.

Step 2 : Move the only disk from peg A to peg B

Step 3 : Move the n – 1 disks from peg C to peg B.

Recursive Definition

$$TOH(n, x, y, z) = \left\{ \begin{array}{c} TOH\ (n-1, x, z, y) \\ \text{move a disk from x to y} \\ TOH\ (n-1, z, y, x) \end{array} \right\} \text{ if } n \geq 1$$

n = Number of disks

x = Source peg

y = Destination peg

z = Empty peg to be used for transfer.

2.4.2(B) Merge Sort

Merge sort is a very popular algorithm for sorting with time complexity O(n log n). Given a sequence of n elements a[0],a[1] …….. a[n–1], these elements are split into two groups a[0], a[1] … $a\left[\frac{n-1}{2}\right]$ and $a\left[\frac{n+1}{2}\right]$ …….. a[n–1]. These two groups are recursively sorted and finally these two sorted lists are merged into a single sorted list.

Division of n number problem into sub problems

Mergesort(a, i, j) is divided into two sub problems:

$$\text{Mergesort}(a, i, \frac{i+j}{2}) \text{ and}$$

$$\text{Mergesort}(a, \frac{i+j}{2}+1, j)$$

Recursion continues as long as i < j.

Solution of the original problem is formed by merging two sorted lists to produce a single sorted list.

In the function merge sort(a, i, j)

a : Array containing elements to be sorted.

i : Index of the first element in a.

j : Index of the last element in a.

'C' function for merge sort

```
void mergesort(int a[ ] , int i , int j)
{
    int c;
    if(i < j )
    {
```

Techknowledge Publications

```
        c = (i + j)/2;
        mergesort(a, i , c);
        mergesort(a, c+1, j);
        merge(a , i, c, j);
    }
}
```

Function 'merge' is explained in chapter on sorting.

2.4.2(C) Quick Sort

In quick sort, an array of n elements is divided into two subarrays through a concept known as partitioning. These sub arrays are not required to be merged later. Elements of the array are re-arranged with the following property:

a[m] ≥ a [i] for i between 0 and m–1

a[j] ≥ a [m] for j between m+1 and n–1.

Point m is known as partition point. Now two sub arrays from a[i].....a[m–1] and a[m+1].....a[j] are recursively sorted using quick sort.

Thus, the concept of partitioning can be used to divide a 'n' number problem into two sub problems. Recursion continues as long as n > 1.

'C' function for quick sort.

```
void quicksort(int a[ ], int i, int j )
{
    int m ;
    if(i < j )
    {
        m = partition(a, i, j );
        quicksort(a, i, m–1);
        quicksort(a, m+1, j);
    }
}
```

Function partition() is explained in chapter on sorting.

2.4.2(D) Finding of Maxima

Problem of finding the largest value in an array with n elements a[0], a[1], ... to a[n–1] can be solved by the divide-and-conquer technique. Above problem can be divided into two sub problems.

max 1 = largest value in array from $a[0].....a\left[\frac{n-1}{2}\right]$

max 2 = largest value in array form $a\left[\frac{n+1}{2}\right].....a[n-1]$

The sub solutions max1 and max2 can be combined to get the largest value in the original array.

if(max1 >max2)

then max1 is the largest number

else

max2 is the largest number.

Recursion terminates, when number of elements n, in the array becomes 1.

'C' function for finding maxima.

```
int maxima(int a[ ], int i, int j )
{
    int max1, max2;
    if(i = = j)
        return(a[i]);
    max1 = maxima(a, i, (i + j)/2);
    max2 = maxima(a, (i + j)/2 + 1, j);
    if(max1>max2)
        return(max1);
    else
        return(max2);
    }
```

2.5 The Greedy Method

Q. Explain the Greedy strategy with suitable example. Comment on its time complexity.

SPPU - Dec. 17, 6 Marks

The greedy method is a very simple technique and it can be applied to a wide variety of problems. Greedy algorithms work in phases in each phase, a decision is made that appears to be good, without regard for future consequences.

Technknowledge Publication

- At each stage, we select an input.
- Input selected is added to the set of optimal solution.
- Selection is made on the basis of some selection procedure.
- Inclusion of next input into the partially constructed optimal solution should result in a feasible solution.

Knapsack Problem

We will first explain the knapsack problem and then see how it can be solved using greedy approach.

We are given n objects and a knapsack(bag).

- Each object is of fixed weight.
- Knapsack has a capacity m(total weight of objects used for filing it).
- There are different types of objects.
- Each type of object gives us a certain profit.

Objective

Fill the knapsack with suitable objects to maximize the profit.

Constraint

Total weight of objects $\leq$ m

Formally, the problem can be stated as;

- Object i has a weight w_i and it gives a profit of P_i.
- A fraction $x_i (1 \geq x_i \geq 0)$ of object i is used to fill the knapsack.

Thus,

Objective is to maximize $\sum_{i=1}^{n} P_i x_i$

With the constraint $\sum_{i=1}^{n} w_i x_i \leq m$.

Greedy approach

(1) Select the object that gives maximum profit per unit weight i.e. Select the object with largest value of P_i/w_i.

(2) Use the object for filling up of knapsack. Now one of the two things can happen.

- Knapsack is filled to its complete capacity.
- Object i is exhausted and some more objects can be added to knapsack.

In Case(1), we finish the algorithms. In case(2), we select the next object with largest P_i/w_i and continue.

Example

Consider the following example of knapsack

There are three types of objects, n = 3

Capacity of knapsack m = 20

Profit due to objects = (25, 24, 15)

(Profit due to object 1 = 25

Profit due to Object 2 = 24

Profit due to Object 3 = 15)

Weights of objects = (18, 15, 10)

Calculating profit per unit weight :

$$i = 1, \frac{25}{18}$$

$$i = 2, \frac{24}{15}$$

$$i = 3, \frac{15}{10}$$

Here $\frac{24}{15} > \frac{15}{10} > \frac{25}{18}$

Step 1 : Select the object 2 for filling the knapsack.

$X_2 = 1$(All objects of type 2 will be used for filling the knapsack.)

Capacity remaining after filling of object of type 2 = 20 – 15 = 5.

Step 2 : Select 1/2 of objects of type 3 to fill the knapsack.

$$\therefore \quad X_3 = 1/2$$

Techknowledge Publications

Step 3 : Since, the knapsack is filled to its complete capacity :

$X_1 = 0$

$\therefore$ Total profit $= 0 \times 25 + 1 \times 24 + 1/2 \times 15$

$= 24 + 7.5 = 31.5$

2.5.1 Job Sequencing with Deadlines

We will first explain the problem of job sequencing with deadlines and then see how it can be solved using the greedy method.

- We are given a set of n jobs.
- With every job i, a deadline d_i is attached
- A job i give a profit of P_i if it is completed by its deadline.
- To complete a job, one has to process the job on the machine for one unit of time.

Objective

Select a subset of jobs to maximize the profit.

Constraint

Selected jobs should be completed by their respective deadlines.

Greedy Approach

In greedy approach, we must find a method for selecting the next job to be included in the set of jobs selected for execution.

- Jobs are considered in non increasing order of their profits.
- Next job to be considered for inclusion is the one that increases profit the most, subject to the constraint of deadline.

For Example :

Let n = 4, $\{p_1, p_2, p_3, p_4\} = \{100, 10, 15, 27)$ and deadlines $\{d_1, d_2, d_3, d_4\} = \{2,1,2,1\}$ Arranging jobs(p_i, d_i) in non-increasing order of their profits :

$(p_1 = 100, d_1 = 2), (p_4 = 27, d_4 = 1),$

$(p_3 = 15, d_3 = 2), (p_2 = 10, d_2 = 1)$

Let J be the set of jobs with optimal solution.

Initially J = ( )

Step 1 : Job 1 is selected

J = (1) with profit of 100

Step 2 : Job 4 is selected

J = (1,4) with profit of 127

Step 3 : Job 3 is selected. It is discarded as (1,2,4) is not a feasible solution.

Step4 : Job 2 is selected. It is discarded as (1,2,4) is not a feasible solution.

Thus J = (1,4) with profit = 127 is the optimal solution. Processing sequence of these jobs = (4,1).

2.5.2 Graph Colouring Problem

Graph colouring problem can be described as given :

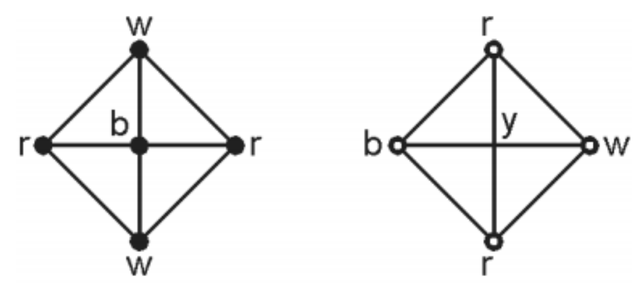
Suppose that, G = (V,E) is Graph.

V = Set of vertices

and E = Set of edges

We have to colour the graph using smallest number of colour. No two adjacent vertices u and v should have the same colour.

Let the set of given colour c = {r, w, b, y} with number of colours n = 4. Fig. 2.5.1 shows proper colouring of graph in two different ways. Fig. 2.5.1 uses three colours. There are many ways to colour this graph properly with three or more colours, but it cannot be done with two or fewer color.



(a) (b)

Fig. 2.5.1

Greedy approach

Greedy approach for colouring of a Graph is explained with the help of an example.

Techknowledge Publications

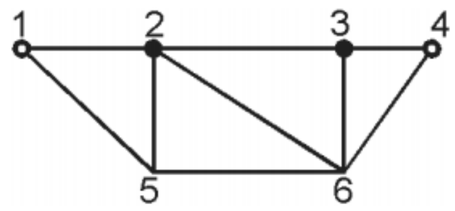


Fig. 2.5.2 : Given graph

Set of colors = {r, b, w, y, p}

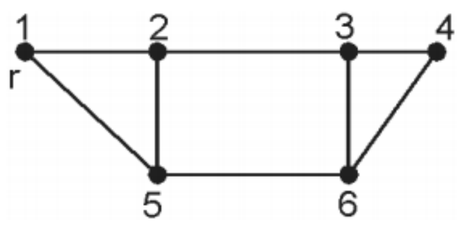
Iteration 1

(a) Pick up vertex that is yet to be coloured

Say, selected vertex = 1

(b) Pick up a colour that has not been used as yet for colouring. Say, selected colour = r

(c) Assign color r to vertex 1.

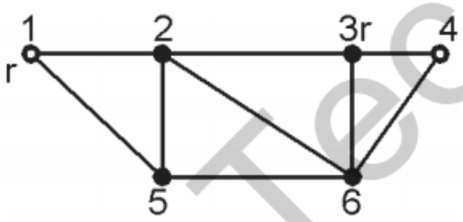


(d) Pick up a vertex u,

u should not have been coloured and should not be adjacent to node 1

Selected vertex = 3

1. Assign same colour to vertex 3



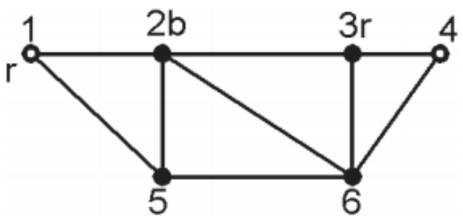
2. Pick up a vertex u,

u should be adjacent to neither 1 nor 3 since there is no such vertex, we repeat the iteration.

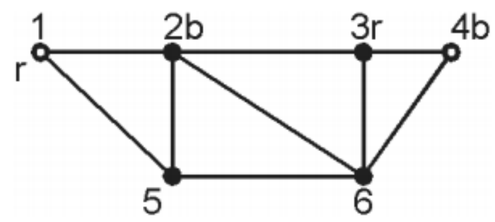
Iteration 2

Next vertex selected for colouring = 2

Next colour selected for colouring = b



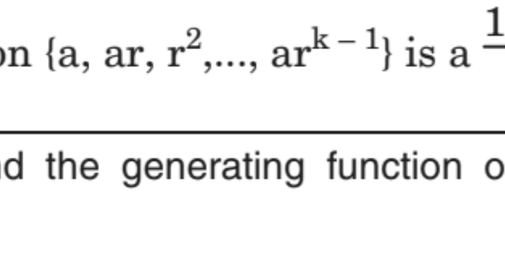
Since the vertex 4 is not adjacent to vertex 2, it can be assigned the same colour.



Iteration 3

Next vertex selected for colouring = 5

Next colours selected for colouring = w

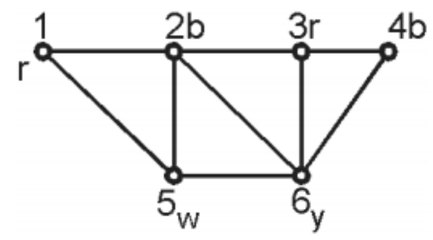


Colour cannot be used to colour any other vertices.

Iteration 4

Next vertex selected for colouring = 6

Next colour selected for colouring = y



Thus, the graph can be coloured using four colours.

2.6 Generating Functions

The generating function for the sequence of real numbers $\{a_0, a_1, a_2, ..., a_n, ...\}$ is given by the power series $a_0 + a_1 x + a_2 x^2 + ... + a_n x^n + ...$

Thus a_n is simply the coefficient of x^n.

For example

1. The generating function of the sequence 1, 2, 3, ... is $f(z) = 1 + 2z + 3z^2 + ...$

2. The generating function for the finite geometric progression $\{a, ar, r^2, ..., ar^{k-1}\}$ is a $\dfrac{1 - r^k z^k}{1 - rz}$

Ex. 2.6.1 : Find the generating function of the sequence 1, 1, 1,....

Soln. : The generating function for the given sequence is given by the power series

$$A(z) = 1 + z + z^2 + z^3 + \ldots \qquad \ldots(1)$$

Technknowledge Publications

Multiplying the power series by z, we get

$$z\,A(z) = z + z^2 + z^3 + \ldots \quad \ldots(2)$$

From the two Equations (1) and (2) on subtraction

$$A(z) - z\,A(z) = 1 \quad \text{or,} \quad A(z) = \frac{1}{1-z}$$

Thus $\frac{1}{1-z}$ is the generating function of the sequence 1, 1, 1,...

Ex. 2.6.2 : Find the generating function for the sequence 0, 1, 2, 3,...

Soln. :

The generating function for the given series is,

$$A(z) = 0 + 1 \cdot z + 2 \cdot z^2 + 3 \cdot z^3 \ldots \quad \ldots(1)$$

Multiplying by z, we get,

$$zA(z) = z^2 + 2z^3 + 3z^4 + \ldots \quad \ldots(2)$$

Taking the difference of two Equations (1) and (2), we get;

$$A(z) - zA(z) = z + z^2 + z^3 + \ldots$$

$$= z(1 + z + z^2 + \ldots) = z \times \frac{1}{1-z}$$

$$\text{or, } (1-z)\,A(z) = \frac{z}{1-z} \quad \text{or} \quad A(z) = \frac{z}{(1-z)^2}$$

Ex. 2.6.3 : Find the generating function for the sequence 1, 2, 3, 4,...

Soln. :

The generating function for the given series is;

$$A(z) = 1 + 2z + 3z^2 + 4z^3 + \ldots \quad \ldots(1)$$

Multiplying by z we get;

$$z \cdot A(z) = z + 2z^2 + 3z^3 + 4z^4 + \ldots \quad \ldots(2)$$

Taking the difference of the two Equation(1) and Equation(2), we get;

$$A(z) - zA(z) = 1 + z + z^2 + z^3 + \ldots = \frac{1}{1-z}$$

$$\text{or,} \quad (1-z)\,A(z) = \frac{1}{1-z}$$

$$\text{or} \quad A(z) = \frac{1}{(1-z)^2}$$

Ex. 2.6.4 : Find the generating function for the sequence 1, a, a^2, a^3, a^4, ...

Soln. : We know that

$$\frac{1}{1-z} = 1 + z + z^2 + \ldots \quad \ldots(1)$$

Replacing z by az in Equation(1)

$$\frac{1}{1-az} = 1 + az + a^2 z^2 + a^3 z^3 + \ldots$$

$\therefore \frac{1}{1-az}$ is the generating function for the series 1, a, a^2, a^3,...

Sr. No.	Series	Generating function
1.	1, 1, 1,... $1 + z + z^2 + z^3 + \ldots$	$\frac{1}{1-z}$
2.	1, 2, 3, 4, ... $1 + 2z + 3z^2 + 4z^3 + \ldots$	$\frac{1}{(1-z)^2}$
3.	0, 1, 2, 3, 4,... $Z + 2z^2 + 3z^3 + \ldots$	$\frac{z}{(1-z)^2}$
4.	1, a, a^2, a^3, ... $1 + az + a^2 z^2 + a^3 z^3 + \ldots$	$\frac{1}{1-az}$

Ex. 2.6.5 : Find discrete numeric function corresponding to the following function : $\frac{1}{5 - 6z + z^2}$

Soln. : Here, we have to find a_n

$$\frac{1}{5-6z+z^2} = \frac{1}{5-5z-z+z^2}$$

$$= \frac{1}{5(1-z) - z(1-z)}$$

$$= \frac{1}{(5-z)(1-z)} = \frac{A}{5-z} + \frac{B}{1-z}$$

We have to calculate the value of constants A and B.

$$\frac{A}{5-z} + \frac{B}{1-z} = \frac{A - Az + 5B - Bz}{(5-z)(1-z)}$$

$$= \frac{(A-5B) - z(A+B)}{(5-z)(1-z)}$$

$$\therefore A + B = 0 \quad \text{and} \quad A - 5B = 1$$

$$\therefore B = -1/6 \quad \text{and} \quad A = \frac{1}{6}$$

Now,

$$\frac{1}{5-6z+z^2} = \frac{1}{6} \times \frac{1}{5-z} - \frac{1}{6} \times \frac{1}{1-z}$$

$$= \frac{1}{6 \times 5} \times \frac{1}{1 - z/5} - \frac{1}{6} \times \frac{1}{1-z}$$

TechKnowledge Publications

The series corresponding to term $\frac{1}{6 \times 5} \times \frac{1}{1 - z/5}$ is given by $\frac{1}{30}\left[1 + \frac{1}{5} z + \frac{1}{5^2} z^2 + ... + \frac{1}{5^n} z^n + ...\right]$

$\therefore$ Co-efficient of z^n is $\frac{1}{30} \times \frac{1}{5^n}$.

The series corresponding to the term $\frac{1}{6} \times \frac{1}{1 - z}$ is given by $\frac{1}{6}$ $[1 + z + z^2 + ... + z^n + ...]$

$\therefore$ Coefficient of z^n is $\frac{1}{6}$.

$\therefore$ Numeric function corresponding to function $\frac{1}{5 - 6z + z^2}$ is given by,

$a_n =$ coefficient of Z^n in 1st term – coefficient of z^n in 2nd term

$$= \frac{1}{30} \times \frac{1}{5^n} - \frac{1}{6} = \frac{1}{6}\left[\frac{1}{5^{n+1}} - 1\right]$$

Ex. 2.6.6 : Find the numeric function for $A(z) = \frac{2}{1 - 4z^2}$

Soln. : Here, we have to find a_n

$$A(z) = \frac{2}{1 - 4z^2} = \frac{2}{1 - (2z)^2}$$

$$= \frac{2}{(1 - 2z)(1 + 2z)}$$

$$= \frac{A}{1 - 2z} + \frac{B}{1 + 2z}$$

After simplification, we get A = 1 and B = 1

$$\therefore A(z) = \frac{1}{1 - 2z} + \frac{1}{1 + 2z} = \frac{1}{1 - 2z} + \frac{1}{1(-2)z}$$

Coefficient of z^n in $\frac{1}{1 - 2z}$ is 2^n

Coefficient of z^n is $\frac{1}{1 - (-2)z}$ is $(-2)^n$.

$\therefore$ Numeric function for $A(z) = \frac{2}{1 - 4z^2}$

$= $ coefficient of z^n in $\frac{1}{1-2z}$ + coefficient of z^n in $\frac{1}{1-(-2)z}$

$$a_n = 2^n + (-2)^n \quad \textbf{or} \quad \begin{bmatrix} a_n = 2^n + 2^n = 2^{n+1} & \text{, n is even} \\ a_n = 2^n - 2^n = 0 & \text{, n is odd} \end{bmatrix}$$

Ex. 2.6.7 : Find the numeric function corresponding to $\frac{z^4}{1 - 2z}$.

Soln. : The series corresponding to $\frac{z^4}{1 - 2z}$ is given by

$z^4 [1 + 2z + 2^2 z^2 + ... + 2^{n-4} z^{n-4} + 2^{n-3} z^{n-3} + ...]$

$= z^4 + 2z^5 + 2^2 z^6 + ... + 2^{n-4} z^n + 2^{n-3} z^{n+1} + ...]$

Coefficient of z^n is 2^{n-4}.

$\therefore a_n = 2^{n-4}$, The required numeric function.

Ex. 2.6.8 : Find the generating function of the numeric function a_r, where $a_r = 3^r + 4^r + 1$, $r \geq 0$.

Soln. : Let, $b_r = 3^r$, $c_r = 4^r$, $d_r = 1$

$\therefore a_r = b_r + c_r + d_r$.

1. Generating function for

$$b_r = 3^r, \ r \geq 0$$

If represents a G.P. $\{1, 3, 3^2, ...3^r...\}$

The generating function for

$$b_r = \frac{1}{1 - 3z}$$

2. Generating function for

$$c_r = 4^r, r \geq 0$$

If represents a G.P. $\{1, 4, 4^2, ...4^r,...\}$

The generating function for

$$c_r = \frac{1}{1 - 4z}$$

3. Generating function for

$$d_r = 1$$

It represents the series {1, 1, 1,..}

The generating function for

$$d_r = \frac{1}{1 - z}$$

$\therefore$ Generating function for a_r

$$= \frac{1}{1 - 3z} + \frac{1}{1 - 4z} + \frac{1}{1 - z}$$

Ex. 2.6.9 : Find the generating function of the numeric function a_r, where $a_r = 5$, $r \geq 0$

Soln. : The generating function for $a_r = 5$, $r \geq 0$ is

$$A(z) = 5 + 5z + 5z^2 + ... + 5z^r + ...$$

$$= 5(1 + z + z^2 + ... + z^r + ...)$$

$$\therefore A(z) = 5 \times \frac{1}{1 - z} = \frac{5}{1 - 2}$$

TechKnowledge Publications

Ex. 2.6.10 : Using generating function, solve the following recurrence relation : $y_{n+2} - 6\,y_{n+1} + 8\,y_n = 0$, $y_0 = 1$, $y_1 = 4$.

Soln. : The recurrence relation can be written as,

$$a_r - 6\,a_{r-1} + 8\,a_{r-2} = 0,\ a_0 = 1,\ a_1 = 4$$

Multiplying the given recurrence relation by z^r, we get

$$a_r\,z^r - 6\,a_{r-1}\,z^r + 8\,a_{r-2}\,z^r = 0$$

Summing for all r's from 2 to ∞, we get;

$$\sum_{r=2}^{\infty} a_r\,z^r - 6\sum_{r=2}^{\infty} a_{r-1}\,z^r + 8\sum_{r=2}^{\infty} a_{r-2}\,z^r = 0 \quad \ldots(1)$$

1. Consider the first term $\sum_{r=2}^{\infty} a_r\,z^r$

$$\sum_{r=2}^{\infty} a_r\,z^r = a_2\,z^2 + a_3\,z^3 + \ldots$$

$$= -(a_0 + a_1\,z) + [(a_0 + a_1\,z) + a_2\,z^2 + a_3\,z^3 + \ldots]$$

$$\ldots[\text{adding and subtracting } a_0 + a_1\,z]$$

$$= A(z) - a_0 - a_1\,z$$

$$\ldots[A[z] = a_0 + a_1\,z + a_2\,z^2 + \ldots]$$

$$= A(z) - 1 - 4z$$

2. Consider the second term $\sum_{r=2}^{\infty} a_{r-1}\,z^r$

$$\sum_{r=2}^{\infty} a_{r-1}\,z^r = a_1\,z^2 + a_2\,z^3 + \ldots.$$

$$= z[a_1\,z + a_2\,z^2 + \ldots]$$

$$= z[A(z) - a_0] = z[A(z) - 1]$$

3. Consider the third term $\sum_{r=2}^{\infty} a_{r-2}\,z^r$

$$\sum_{r=2}^{\infty} a_{r-2}\,z^r = a_0\,z^2 + a_1\,z^3 + \ldots$$

$$= z^2\,[a_0 + a_1\,z + a_2\,z^2 + \ldots] = z^2\,A(z)$$

$$\boxed{A(z) = a_0 + a_1\,z + a_2\,z^2 + \ldots}$$

Putting these values in Equation(1), we get

$$A(z) - 1 - 4z - 6z\,(A(z) - 1) + 8\,z^2\,A(z) = 0$$

or, $A(z)\,[1 - 6z + 8z^2] - 1 - 4z + 6z = 0$

or $A(z)[1 - 6z + 8z^2] = 1 - 2z$

or, $$A(z) = \frac{1 - 2z}{1 - 6z + 8z^2}$$

$$= \frac{(1 - 2z)}{(1 - 2z) + (1 - 4z)} = \frac{1}{1 - 4z}$$

Thus, $a_r = 4^r$

Ex. 2.6.11 : Solve the following recurrence relation $a_r - 3\,a_{r-1} = 2$, $r \geq 1$ with $a_0 = 1$ using the generating function.

Soln. : Multiplying the given recurrence relation by z^r, we get;

$$a_r\,z^r - 3a_{r-1}\,z^r = 2\,z^r,\ r \geq 1$$

Summing for all r's from 1 to ∞, we get

$$\sum_{r=1}^{\infty} a_r\,z^r - 3\sum_{r=1}^{\infty} a_{r-1}\,z^r = 2\sum_{r=1}^{\infty} z^r \quad \ldots(1)$$

1. Consider the first term $\sum_{r=1}^{\infty} a_r\,z^r$

$$\sum_{r=1}^{\infty} a_r\,z^r = a_1\,z + a_2\,z^2 + a_3\,z^3 + \ldots$$

$$= A(z) - a$$

As the generating function A(z) stands for

$$A(z) = a_0 + a_1\,z + a_2\,z^2 + a_3\,z^3 + \ldots$$

2. Consider the second term $\sum_{r=1}^{\infty} a_{r-1}\,z^r$

$$\sum_{r=1}^{\infty} a_{r-1}\,z^r = a_0\,z + a_1\,z^2 + a_2\,z^3 + \ldots = zA(z)$$

3. Consider the third term $\sum_{r=1}^{\infty} z^r$

$$\sum_{r=1}^{\infty} z^r = z + z^2 + z^3 + \ldots$$

$$= z\,[1 + z + z^2 + \ldots] = \frac{z}{1 - z}$$

Techknowledge Publications

Putting these values in Equation(1), we get

$$A(z) - a_0 - 3z\,A(z) = \frac{2z}{1-z}$$

$$\text{or,}\quad (1-3z)\,A(z) = \frac{2z}{1-z} + a_0$$

$$= \frac{2\,z}{1-z} + 1 = \frac{1+z}{1-z}$$

$$\therefore\quad A(z) = \frac{1+z}{(1-z)\,(1-3z)}$$

$$= \frac{2}{1-3\,z} - \frac{1}{1-z}$$

$$\text{Thus,}\quad a_r = 2\cdot 3^r - (1)^r, \quad r \geq 0$$

$$= 2\cdot 3^r - 1, \quad r \geq 0$$

2.7 Multiplication Technique

Carl Friedrich Gauss algorithm

Computationally, multiplying two numbers is much more costly than is adding them. Suppose that multiplying two real numbers cost \$1 and adding them costs a penny.

Suppose we want to multiply two complex numbers, a + ib and c + id.

Now,

$$(a + ib) \times (c + id) = (ac - bd) + i\,(ad + bc)$$

It requires four multiplications and two additions, for a total of \$4.02.

This, as per the Gauss algorithm can be done using, three multiplications and five additions.

$x_1 = a + b$

$x_2 = c + d$

$x_3 = x_1 x_2 = ac + ad + bc + bd$

$x_4 = ac$

$x_5 = bd$

$x_6 = x_4 - x_5 = ac - bd$

$x_7 = x_3 - x_4 - x_5 = bc + ad$

Kasturba's Algorithm

Kasturba's divide and conquer multiplication algorithm takes advantage of the saving proposed by Gauss algorithms.

Review Questions

Q. 1 What is ADT ? Write array as an ADT. **(6 Marks)**

Q. 2 Find the solution to the recurrence relation $a_n = 6\,a_{n-1} - 11\,a_{n-2} + 6\,a_{n-3}$ with initial conditions $a_0 = 2$, $a_1 = 5$ and $a_2 = 15$. **(6 Marks)**

Q. 3 Explain Data type. **(2 Marks)**

Q. 4 Explain the term data structure and its operations. **(6 Marks)**

Q. 5 Differentiate between static memory and dynamic memory allocation. **(4 Marks)**

Q. 6 What is recurrence relation ? Solve the following recurrence relation : $a_r - 7\,a_{r-1} + 10\,a_{r-2} = 0$ given that $a_0 = 0$ and $a_1 = 3$. **(6 Marks)**

Q. 7 Classify data structures and give example of each type. **(6 Marks)**

Q. 8 Explain linear and non-linear data structures. **(6 Marks)**

Q. 9 What is persistent data structure? **(2 Marks)**

Q. 10 Explain static and dynamic data structures with suitable examples. **(3 Marks)**

Techknowledge Publication

Linear Data Structures using Sequential Organization

Syllabus :

Concept of Sequential Organization, Overview of Array, Array as an Abstract Data Type, Operations on Array, Merging of two arrays, Storage Representation and their Address Calculation : Row major and Column Major, Multidimensional Arrays : Two-dimensional arrays, n-dimensional arrays. Concept of Ordered List, Single Variable Polynomial : Representation using arrays, Polynomial as array of structure, Polynomial addition, Polynomial multiplication. Sparse Matrix : Sparse matrix representation using array, Sparse matrix addition, Transpose of sparse matrix- Simple and Fast Transpose, Time and Space tradeoff.

Exemplar/Case Studies : Study use of sparse matrix in Social Networks and Maps. Study how Economists use polynomials to model economic growth patterns, how medical researchers use them to describe the behaviour of Covid-19 virus.

Technknowledge Publication

3.1 Sequential Organization

Q. Define and explain the term Sequential Organization. **SPPU - May 18, 1 Mark**

Sequential Memory Organization

In sequential organization, elements are stored in consecutive memory locations. Array is an example of sequential organization. This form of data structure is simple to use and it is supported by almost all programming languages. It is very simple to understand and time to access an element is constant.

- Simple to use
- Simple to define
- Constant access time
- Mapping by compiler

} Properties of Sequential data structure

An array element can be accessed by a[i], where a is the name of the array and i is the index. Compiler maps a[i] to its physical location in memory.

Address of a[i] is given by starting address of a + i* size of array element in bytes.

This mapping is carried out in constant time, irrespective of which element is being accessed. Sequential data structure suffers from some severe limitations :

- Size of an array is defined at the time of programming.
- Insertion and deletion is time consuming.
- Requires contiguous memory.

First, the size of an array has to be defined when the program is being written and its space is calculated during compilation of the program. This means that the programmer has to take a decision regarding the maximum size of data.

If the actual amount of data stored is less than the maximum size, a good amount of memory will be wasted and on the other hand a larger sample of data can not be handled. To avoid the linear cost of insertion and deletion, we need to ensure that the list is not stored contiguously, since, otherwise entire parts of list will need to be moved.

ADT for Array / Array as an ADT

Array is set of pairs, index and value. For each index, there is a value associated with it. Arrays are often stored in consecutive set of memory locations.

For arrays, there are three operations

1. CREATE( ) → array
2. RETRIEVE(array, index) → value
3. STORE(array, index, value) → array

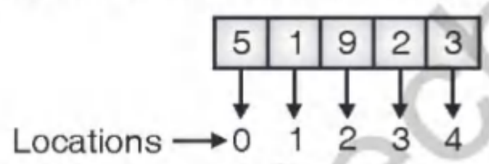
Thus, array ADT can be declared as given

```
array create( );
value retrieve(array, index);
array store(array, index, value);
```

The function create( ) produces a new empty array. The function retrieve takes as input an array and an index and returns the appropriate value. The function store( ) is used to store a new index-value pair.

3.2 Introduction to Arrays

- Array is a collection of elements.
- All elements are of the same type.
- All elements are stored using contiguous memory.



- int a[5], declares an integer array of five elements.
- Elements of an array can be initialised at the time of declaration.

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

It is also possible to initialize individual elements as follows :

```
a[0] = 1;
a[1] = 2;
a[2] = 3;
a[3] = 4;
a[4] = 5;
```

- A particular value is accessed by writing a number called index number or subscript in brackets after the array name.
- For example,

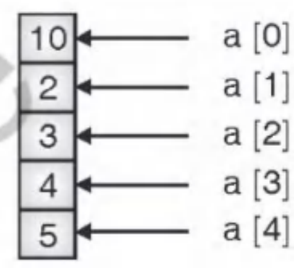
```
a[3]  /*represents the third element of the array.*/
```

- Index starts from 0 and goes upto n – 1, where n is the size of the array.

```
int a[5];  /* an array of 5 elements) */
```

- In an array, element a[i] can be manipulated like a normal variable.

```
a[5] = 0;
a[5] = a[4] + 2;
```



3.3 Representation and Analysis

- Arrays must be declared before they are used. The general form of array declaration is

```
type variable_name[size]
```

The type specifies the type of the element to be stored in the array. Type could be any of the primitive types like int, float, char, long, double etc. or user defined type.

```
int number[10];
```

Declares an array of 10 integer constant with the name number. Any subscript from 0 to 9 is valid.

```
float marks[15];
```

Declares marks to be an array containing 15 real numbers.

Note : C-language does not do any range checking on array subscript. Any reference to the array outside the declared limits may result in unpredictable program results.

```
char name[20];
```

Declares the name as a character array that can hold a maximum of 20 characters.

It may be noted that strings are stored in character array and a string is terminated by a null character ('\0')

Technknowledge Publication

"INDIA"

Each character of the string is treated as an array element and is stored in the memory as given as follows :

'I'	name[0]
'N'	name[1]
'D'	name[2]
'I'	name[3]
'A'	name[4]
'\0'	name[5]

name[i] represents individual element of the string, where as name without any subscript indicates the entire string

Array elements are stored in successive memory location.

Program 3.3.1 : Program to display array elements with their addresses.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a[3] = {5, 4, 3};
    printf("\n a[0], value = %d : address = %u", a[0],
    &a[0]);
    printf("\n a[1], value = %d : address=%u", a[1],
    &a[1]);
    printf("\n a[2], value = %d : address = %u", a[2],
    &a[2]);
    getch();
}
```

Output

```
a[0] , value = 5 : address = 65520
a[1] , value = 4 : address = 65522
a[2] , value = 3 : address = 65524
```

An integer requires 2 bytes of memory locations. Addresses displayed have a difference of 2.

3.4 One-Dimensional Arrays

One-dimensional array is a list of elements or simply a row of elements. In mathematics, we often deal with variables that are simple scripted.

For instance, $\sum_{i=0}^{n} x_i$

x_i, refers to the ith element of x. These variables can be expressed as x[0], x[i], ... x[n – 1]

```
sum = 0;
for (i = 0; i < n; i++)      /* loop for traversing */
sum = sum + x[i];
```

Elements of an array can be scanned from left to right, element by element through the use of a for-loop. Starting index is 0 and it goes upto n – 1 in step of 1.

Program 3.4.1 : Program to read and display the elements of an array alongwith the total.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int i, a[50], sum, n;
    printf("\n Enter no. of elements :");
    scanf("%d", &n);  /* Reading values into Array */
    printf("\n Enter the values :");
    for(i = 0; i<n; i++)
    scanf("%d", &a[i]);    /* computation of total */
    sum = 0;
    for(i = 0; i<n; i++)
    sum = sum+a[i];
    /* printing of all elements of array */
    for(i = 0; i<n; i++)
    printf("\n a[%d] = %d", i, a[i]);
    /* printing of total */
    printf("\n sum = %d", sum);
}
```

Output

```
Enter no. of elements :3
Enter the values :  1    2    3
a[0] = 1
a[1] = 2
a[2] = 3
sum = 6
```

Techknowledge Publication

Array a[50] can store upto maximum of 50 elements. User can store any number of elements by reading and storing a value in the variable n. Variable i is used to scan the array from the location 0 to n – 1.

Sum of the elements of an array can be represented using the conventional mathematical notation.

$\sum_{i=0}^{n-1} x_i$ and same can be converted to an equivalent for loop

```
sum = 0;
for (i = 0; i < n; i ++)
    sum = sum + x[i];
```

3.5 Operations with Arrays

Some of the frequently used operations with arrays are shown in Fig. 3.5.1.

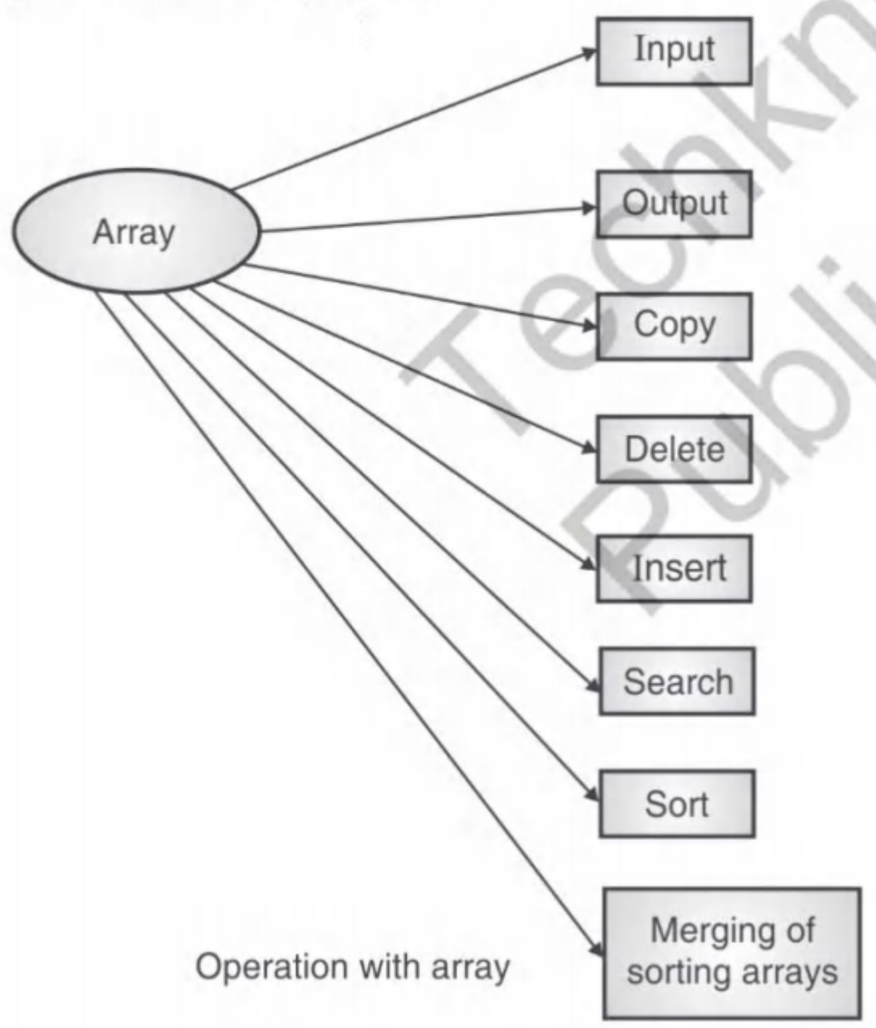
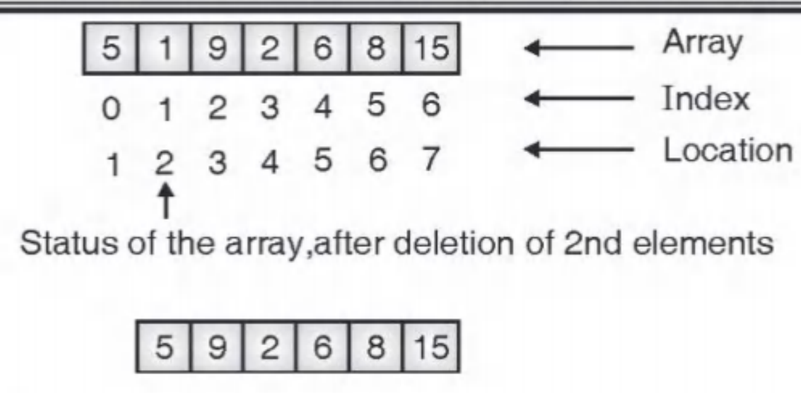


Fig. 3.5.1 : Operations with arrays

3.5.1 Deletion

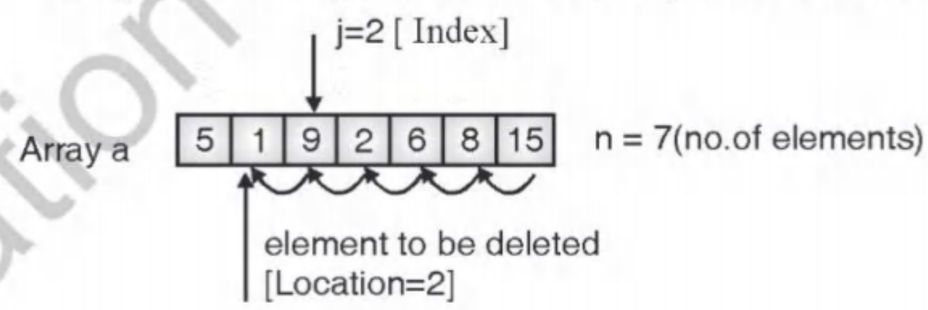
Deletion involves deleting the specified element from the array.



In order to delete the second element, all elements ahead of second location should be moved left by one location.

```
/* delete the element from the location j  */
index of the jth element = j – 1
while(j < n)
{
    a[j – 1] = a[j];
    j++;
}
n = n – 1    // no. of elements is reduced by 1.
```

Loop for shifting of elements left by one location



a[1] = a[2]

a[2] = a[3]

a[3] = a[4]

a[4] = a[5]

a[5] = a[6]

↑

j varies from 2 to 6

Value of n becomes 6 after deletion

Program 3.5.1 : Program for deletion of an element from the specified location.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a[30], n, i, j;
    /*  n : no. of elements stored in array
```

Technknowledge Publications

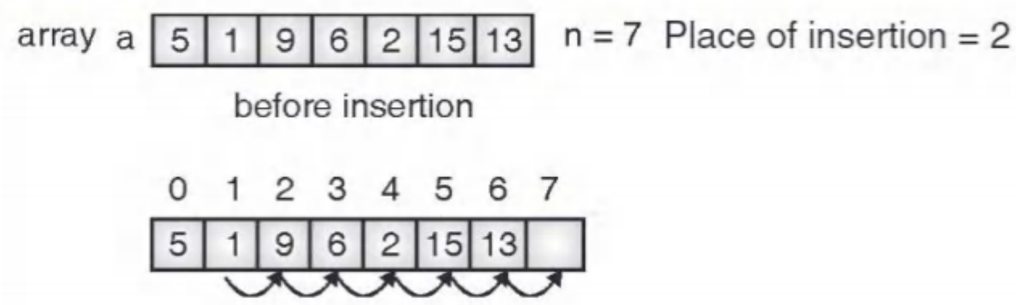
```
    i  : for scanning the array
    j  : location of the element to be deleted
*/
printf("\n Enter number of elements :");
scanf("%d", &n);
/* read n elements in an array  */
printf("\n Enter %d elements :", n);
for(i = 0; i<n; i++)
    scanf("%d", &a[i]);
/* read the location of the element to be deleted */
printf("\n location of the element to be deleted :");
scanf("%d", &j);
/* loop for the deletion  */
while(j<n)
{
    a[j-1] = a[j];
    j++;
}
n--;   /* number of elements reduced by 1 */
/* loop for printing  */
for(i = 0; i<n; i++)
    printf("\n %d", a[i]);
getch();
}
```

Output

```
Enter number of elements :4
Enter 4 elements :23    45    67    12
 location of the element to be deleted :3
 23    45    12
```

3.5.2 Insertion

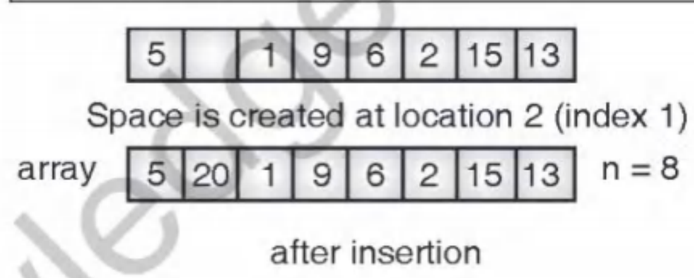
This operation can be used to insert an arbitrary element at specified location in an array.



Creation of space at location 2 index of location 2 is 1 i.e. loc–1

Movement of data to its right starts from last location(index n – 1)

```
for(i = n – 1; i > = loc–1; i – –)
a[i + 1] = a[i];
```



Data to be inserted = 20

Program 3.5.2 : Program for insertion of an element at the specified location.

```
#include<stdio.h>
#include<conio.h>
void main()
{       int a[30], x, n, i, loc;
    /*  x : element to be inserted
        n : no. of elements in the array
        i : for scanning of the array
        loc : place where the new element is to be
    inserted */
    printf("\n Enter no. of elements :");
    scanf("%d", &n);   // read n elements in an array
    for(i = 0; i<n; i++)
        scanf("%d", &a[i]);
    printf("\n Enter the element to be inserted :");
    scanf("%d", &x);
    printf("\n Enter the location);
    scanf("%d", &loc);
    /* create space at the specified location */
    for(i = n–1; i> = loc–1; i– –)
        a[i+1] = a[i];
    n++;   // number of elements is increased by 1.
    a[loc–1] = x;     /* Element inserted  */
```

Technknowledge Publications

```
    /* Printing of result */
    for(i = 0; i<n; i++)
        printf("\n %d", a[i]);
}
```

Output

```
Enter no. of elements :4
12    13    14    15
Enter the element to be inserted :20
Enter the location 2
 12    20    13    14    15
```

Program 3.5.3 : Program to copy elements of an array into another array.

```
#include<stdio.h>
#include<conio.h>
void main()
{   int a[30], b[30], i, n;
/* Element of an array 'a' will be copied into array b
    i -> for scanning of an array
    x -> no. of elements in the array */
    printf("\n Enter no. of elements :");
    scanf("%d", &n);  /* Reading values into Array */
    printf("\n Enter the values :");
    for(i = 0; i<n; i++)
    scanf("%d", &a[i]);
    /* Copying data from array 'a' to array 'b' */
    for(i = 0; i<n; i++)
        b[i] = a[i];
    /* printing of all elements of array */
    printf("the copied array is :");
   for(i = 0; i<n; i++)
        printf("b[%d] = %d", i, b[i]);
    getch();
}
```

Output

```
Enter no. of elements :5
Enter the values :12    22    32    42    52
the copied array is :  b[0] = 12    b[1] = 22
b[2] = 32    b[3] = 42    b[4] = 52
```

3.5.3 Search

The process of finding whether the specified element is an array is known as searching.

Program 3.5.4 : Program to search for an elements in an array.

```
#include<stdio.h>
#include<conio.h>
void main()
{   int a[30], x, n, i;
    /*  a : for storing of data
        x : element to be searched
        n : number of elements in the array
        i : scanning of the array      */
    printf("\n Enter no. of elements :");
    scanf("%d", &n);  //Reading values into Array
    printf("\n Enter the values :");
    for(i = 0; i<n; i++)
        scanf("%d", &a[i]);
    /* read the element to be searched */
    printf("\n Enter the elements to be searched");
    scanf("%d", &x);
    /* search the element */
    i = 0;      /* search starts from the location zero */
    while(i<n && x! = a[i])
        i++;
    //search until the element is not found i.e. x! = a[i]
    /* search until the element could still be found
    i.e. i<n */
    if(i<n)     /* Element is found */
        printf("found at the location = %d", i+1);
    else
        printf("\n not found");
    getch();
}
```

Output

```
Enter no. of elements :3
Enter the values :55    56    57
Enter the elements to be searched 56
found at the location = 2
```

Techknowledge Publications

In the above program, n elements are stored in the array a. Element to be searched x is searched by scanning the array from left to write using the loop.

```
i = 0;
while (i < n && x! = a[i])
    i++;
```

The condition i < n will fail if the element to be searched does not exist in the array. The second condition x! = a[i] will fail when the element x is found in the array. After termination of the loop, one can check the value of i, if the value of i < n then the condition x! = a[i] has failed and the element x has been found in the array.

3.5.4 Merging of Sorted Arrays

Merging of two sorted array a and b is carried out in such way that the resultant array c is sorted after merging.

Array a	[1 5 9 11 13]	n = 5
Array b	[2 4 5 6]	n = 4
Merged array c	[1 2 4 5 5 6 9 11 13]	n = 9

Algorithm

Variable i : To scan the array a,

Variable j : To scan the array b,

Variable k : Number of elements copied into c

Initial conditions :

array a[1 5 9 11 13] array b[2 4 5 6] array c[ ]

↑ ↑ ↑

i = 0 j = 0 k = 0

Since the ith element of the array a is smaller than the jth element of the array b, it is copied into c. i and k are incremented by 1.

a[1 5 9 11 13] array b[2 4 5 6] array c[1]

↑ ↑ ↑

i = 1 j = 0 k = 1

Since b[j] < a[i] { c[k] = b[j], j++, k++ }

a[1 5 9 11 13] array b[2 4 5 6] array c[1 2]

↑ ↑ ↑

i = 1 j = 1 k = 2

Since b[j] < a[i] { c[k] = b[j], j++, k++ }

a[1 5 9 11 13] array b[2 4 5 6] array c[1 2 4]

↑ ↑ ↑

i = 1 j = 2 k = 3

Since a[i] < b[j] { c[k] = a[i], i++, k++ }

a[1 5 9 11 13] array b[2 4 5 6] array c[1 2 4 5]

↑ ↑ ↑

i = 2 j = 2 k = 4

Techknowledge Publications

Since b[j] < a[i] { c[k] = b[j], j++, i++ }

a[1 5 9 11 13] array b[2 4 5 6] array c[1 2 4 5 5]

↑ ↑ ↑

i = 2 j = 3 k = 5

since b[j] < a[i] { c[k] = b[j], j++, k++ }

a[1 5 9 11 13] array b[2 4 5 6] array c[1 2 4 5 5 6]

↑ ↑ ↑

i = 2 j = 4 k = 6

Since, array b[ ] is exhausted, remaining elements of the array a[ ] are copied at the end of array c.

a[1 5 9 11 13] b[2 4 5 6] c[1 2 4 5 5 6 9 11 13]

↑ ↑ ↑

i = 5 j = 4 k = 9

Program 3.5.5 : Program for merging of two array.

OR Write pseudo C code to merge two sorted lists of integers stored in arrays to form a third sorted list. Analyse time complexity of this code.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a[30], b[30], c[30], i, j, k, n1, n2;
    /*  n1 : Number of elements in the array 'a'
        n2 : Number of elements in the array 'b'
    */
    printf("\n Enter no. of elements in 1'st array :");
    scanf("%d", &n1);
    for(i = 0; i<n1; i++)
        scanf("%d", &a[i]);
    printf("\n Enter no. of elements in 2'nd array :");
    scanf("%d", &n2);
    for(i = 0; i<n2; i++)
        scanf("%d", &b[i]);
    i = 0; j = 0; k = 0;  /* merging starts */
    while(i<n1 && j<n2)
    {
        if(a[i]< = b[j])
        {
            c[k++] = a[i++];
        }
        else
        {
            c[k++] = b[j++]
        }
    }
    /* Some elements in array 'a' are still remaining
       where as the array 'b' is exhausted
    */
    while(i<n1)
    {
        c[k++] = a[i++]
    }
    /* Some elements in array b are still remaining
       whereas the array 'a' is exhausted
    */
    while(j<n2)
    {
        c[k++] = b[ j++]
    }
    /* Displaying elements of array 'c' */
      printf("\nMerged array is :");
        for(i = 0; i<n1+n2; i++)
        printf("\n %d", c[i]);
}
```

Techknowledge Publications

Output

```
Enter no. of elements in 1'st array : 5
1 5 9 11 13
Enter no. of elements in 2'nd array : 4
2 4 5 6
Merged array is : 1 2 4 5 5 6 9 11 13
```

Time Complexity

$O(n_1 + n_2)$ where n_1 and n_2 are number of elements in the two list respectively.

3.5.5 Reversing an Array

Program 3.5.6 : Write pseudo 'c' code to reverse the 'n' numbers in one dimensional array.

```
void main( )
{   int a[100], n, i, j;
    read n              // number of elements
    for(i from 0 to n – 1)
        read a[i]       /*  Position i on the first element
                            and  j on the last element */
    i = 0;
    j = n–1;
    while(i < j)
    {
        interchange a[i] and a[j]
        i = i + 1;
        j = j – 1;
    }
    // print the result
    for(i from 0 to n – 1)
        print a[i];
}
```

3.6 Two-Dimensional Arrays

Two dimensional arrays with column and row major implementation

Two-dimensional arrays can be thought of as a table consisting of rows and columns.

int a[3][4], declares an integer array of three rows and four columns. Row index starts from 0 and will go upto 2.

Similarly, column index will start from 0 and will go upto 3.

Table 3.6.1 : Arrangement of elements and their indices

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Program 3.6.1 : Program to show the use of two dimensional array.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a[10][10], m, n, i, j;
    /*  physical size of the array is 10*10
        whereas a specified size of the array
        of the size m × n can be stored in array a,
        m : number of rows
        n : number of columns
    */
    printf("\n Enter the rows and columns :");
    scanf("%d %d", &m, &n);
    /* Reading m*n elements */
    for(i = 0; i<m; i++)    /* m rows */
    for(j = 0; j<n; j++)
    /* inside each row there are n columns */
    {
        printf("\n Enter the value of(%d)(%d)=", i, j);
        scanf("%d", &a[i][j]);
    }
    /* printing of all elements of 2-D array */
    for(i = 0; i<m; i++)
    {
        printf("\n");   // each row starts from newline
        for(j = 0; j<n; j++)
        printf("%d\t", a[i][j]);
    }
    getch();
}
```

Techknowledge Publications

Output

```
Enter the rows and columns :3 4
Enter the value of(0)(0) = 2
Enter the value of(0)(1) = 1
Enter the value of(0)(2) = 0
Enter the value of(0)(3) = 5
Enter the value of(1)(0) = 9
Enter the value of(1)(1) = 3
Enter the value of(1)(2) = 6
Enter the value of(1)(3) = 1
Enter the value of(2)(0) = 4
Enter the value of(2)(1) = 6
Enter the value of(2)(2) = 7
Enter the value of(2)(3) = 8
2    1    0    5
9    3    6    1
4    6    7    8
```

A nested loop is required to scan all elements of a two dimensional array. Outer loop is for rows and the inner loop is for columns.

```
for(i = 0; i < m; i ++)     /* m rows */
for(j = 0; j < n; j ++)     /* n columns */
```

Above nested loop is required for all simple operations like reading, printing, addition of two arrays.

3.6.1 Initializing Two-Dimensional Arrays

Similar to one dimensional arrays, two-dimensional arrays may be initialized by providing a list of initial values.

```
int a[2][3] = {0, 1, 1, 1, 0, 0};
```

Above statement can be equivalently written as

```
int a[2][3] = {{1, 1, 1}, {2 , 2 , 2}};
```

3.6.2 Address Calculation

Q. Derive address calculation formula for one dimensional array with one example. **SPPU - Dec. 16, 6 Marks**

Q. Explain two-dimensional arrays with row and column major implementation. Explain address calculation in both cases with example. **SPPU - Dec. 18, 6 Marks**

Two dimensional arrays with address calculation

An m × n matrix (A[1..m][1..n]) where the row index varies from 1 to m and column index varies from 1 to n can be written as (please note that in c language the index will start from 0 and go upto m – 1).

$$a = \begin{bmatrix} a_{11} & a_{12} & \ldots\ldots & a_{1n} \\ a_{21} & a_{22} & \ldots\ldots & a_{2n} \\ \vdots & & & \\ \vdots & & & \\ \vdots & & & \\ a_{m1} & a_{m2} & \ldots\ldots & a_{mn} \end{bmatrix}$$

a_{ij} denotes the entry in the ith row and the jth column. In computer memory, all elements are stored linearly using contiguous addresses.

Hence, in order to store a two dimensional matrix a, two dimensional address space must be mapped to one dimensional (linear) address space.

In the computer's memory metrices are stored in either **row major form** or **column major form.**

Row major form

Elements are stored row wise i.e. row 1, row 2, ... row m.

Column major form

Elements are stored column wise i.e. column 1, column 2, ... column n.

It is assumed that index goes from 0 to n – 1 for an n-element array.

Address of the ith element = starting address + i × size of each element.

Example

Consider 1-D array as with base address = 1000

A[4] = {10,20,30,40,50};

Address of 2nd element i.e.

$$30 = 1000 + 2 \times 2$$
$$= 1004$$

Mapping of a two dimensional (3 × 4) array

[0][0]	[0][1]	[0][2]	[0][3]	0	1	2	3	0	3	6	9
[1][0]	[1][1]	[1][2]	[1][3]	4	5	6	7	1	4	7	10
[2][0]	[2][1]	[2][2]	[2][3]	8	9	10	11	2	5	8	11
Arrangement of indices				Row major mapping				Column major mapping			

[C uses row-major scheme and FORTRAN uses column major scheme]

To locate the elements a_{ij} in a 2-dimensional array a[L1 ... U1][L2 ... U2] where L1 and U1 are the lower and upper bounds respectively for rows and L2 and U2 are the lower and upper bounds respectively for columns.

$$\begin{array}{lllll} a_{L1\ L2} & a_{L1\ \ L2+1} & a_{L1\ \ L2+2} & \cdots & a_{L1\ \ U2} \\ a_{L1+1\ L2} & a_{L1+1\ \ L2+1} & a_{L1+1\ \ L2+2} & & a_{L1+1\ \ U2} \\ \vdots & & & & \\ a_{U1\ L2} & a_{U1\ \ L2+1} & a_{U1\ \ L2+2} & & a_{U1\ \ U2} \end{array}$$

arrangement of indices for the array a[L1 ... U1][L2 ... U2]

(A) Location of an element a_{ij} (row major form) = B + (i – L1) × (U2 – L2 + 1) × S + (j – L2) × S

B : Base address

U2 : L2 + 1 – Number of columns, i.e. number of elements in each row

S : Number of bytes taken to store each element (an integer requires 2 bytes of memory)

(i – L1) × (U2 – L2 + 1) × S – i^{th} row will start after i – L1 rows, each row has U2 – L2 + 1 elements and the size of each element is S bytes.

Application of above formula for an array declared in C-program.

```
int a[3][4]
```

Let the array is stored in memory at a base location 100. Let each element be stored in memory using S = 2 bytes

Lower bound L1 = 0, Upper bound U1 = 2] row

Lower bound L2 = 0, Upper bound U2 = 3] column

location of [2][1] = 100 + (2 – 0) × (3 – 0 + 1) × 2 + (1 – 0) × 2 = 100 + 16 + 2 = 118

Elements	a_{00}	a_{01}	a_{02}	a_{03}	a_{10}	a_{11}	a_{12}	a_{13}	a_{20}	a_{21}	a_{22}	a_{23}
Locations	100	102	104	106	108	110	112	114	116	118	120	122

Elements stored in contiguous memory locations in row major form.

(B) Location of an element aij (column major form) = B + (j – L2) × (U1 – L1 + 1) × S + (i – L1) × S

location of[2][1] = 100 + (1 – 0) × (2 – 0 + 1) × 2 + (2 – 0) × 2 = 100 + 6 + 4 = 110

Elements	a_{00}	a_{10}	a_{20}	a_{01}	a_{11}	a_{21}	a_{02}	a_{12}	a_{22}	a_{03}	a_{13}	a_{23}
Locations	100	102	104	106	108	110	112	114	116	118	120	122

TechKnowledge Publications

Ex. 3.6.1 : Each element of an array Data [20][50] requires 4 bytes of storage. Base address of Data is 2000. Determine the location of Data [10][10] when the array is stored as : (i) Row major (ii) Column major.

Soln. :

(i) Location of an element a_{ij} (row major form)

$$= B + (i - L1) \times (U2 - L2 + 1) \times S + (j - L2) \times S$$

Here L1 = 0, U1 = 19, L2 = 0, U2 = 49 and B = 2000

required location = 2000+(10–0)×(49–0+1)×4+(10–0)×4

$$= 2000 + 2000 + 40 = 4040$$

(ii) Location of an element a_{ij} (column major form)

$$= B + (j - L2) \times (U1 - L1 + 1) \times S + (i - L1) \times S$$

$$= 2000 + (10) \times (20) \times 4 + 10 \times 4$$

$$= 2000 + 800 + 40 = 2840$$

3.7 Multi-Dimensional Arrays

C allows arrays of 1 or more dimensions. Exact limit is compiler dependent.

General form of a multi-dimensional array is

```
type array_name[S1][S2] ...[Sn];
```

Where S_i is the size of the ith dimension.

Some examples are :

```
int a[3][4][2];
float b[9][6][3];
```

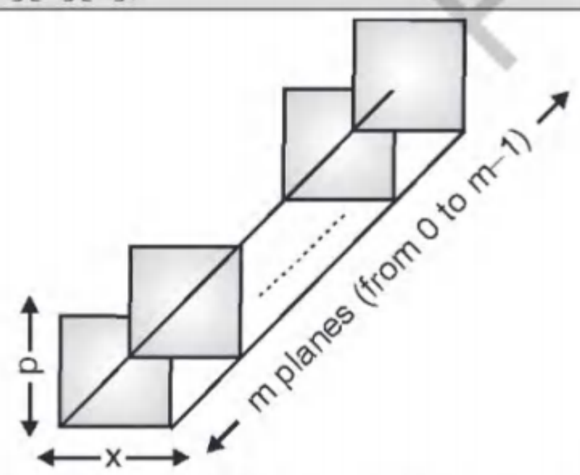


Fig. 3.7.1 : A three dimensional array of the size m×n×p.

m → Number of planes (from plane number 0 to m – 1)

n → Number of rows in each plane (from row number 0 to n – 1)

p → Number of columns in each plane (from column number 0 to p – 1)

Number of elements in each plane = n × P.

Element a[i][j][k], will be found in the ith plane.

Offset of ith plane = i × size of plane = i × n × p

Offset of jth row in a plane = j × number of elements in a row

= j × p

Offset of kth element in a row = k

a[i][j][k] will found at the address = starting address of the array + (i × n × p + j × p + k) × Size of each element

Ex. 3.7.1 : Consider integer array, int arr[5][4] declared in 'c' program. If the base address is 510, find the address of the element arr[3] [2], with row major and column major representation of array.

Soln. :

Base address = 510

Size of each element = 2 bytes

Size of the array = 5 × 4

Address of arr[3] [2] with row major representation : 510 + (4 × 3 + 2) × 2

= 510 + 28 = 538

Address of arr[3] [2] with column major representation : 510 + (2 × 5 + 3) × 2

= 510 + 26 = 536

3.8 Application of Arrays

3.8.1 Addition of Two 2-D Matrices

- If we add two matrices $a_{m \times n}$ and $b_{m \times n}$ producing $c_{m \times n}$ then $(i, j)^{th}$ element of the resultant matrix c[i][j] = a[i][j] + b[i][j].
- Two matrices can be added using the following program segment :

```
for(i = 0; i<m; i++)   } Calculate all elements of the
                       } array c.
for(j = 0; j<n; j++)   } c[i][j] = a[i][j] + b[i][j];
```

- Two matrices to be added must be addition compatible i.e. they must be of the same size.

Technknowledge Publication

Program 3.8.1 : Program for addition of two matrices.

```
#include<stdio.h>
#include<conio.h>
void add(int[ ][10], int[ ][10], int[ ][10], int, int);
void read(int[ ][10], int, int);
void print(int[ ][10], int, int);
void main()
{
    int a[10][10], b[10][10], c[10][10], m, n;
    printf("\n Enter the size of matrix :");
    scanf("%d%d", &m, &n);
    printf("\n Enter the elements of 1'st matrix");
    read(a, m, n);
    printf("\n Enter the elements of 2'nd matrix");
    read(b, m, n);
    add(a, b, c, m, n);
    printf("\n1'st matrix is:");
    print(a, m, n);
    printf("\n2'nd matrix is:");
    print(b, m, n);
    printf("\nresult matrix is:");
    print(c, m, n);
}
void add(int x[ ][10], int y[ ][10], int z[ ][10], int m, int n)
{
    int j, i;
    for(i = 0; i<m; i++)
        for(j = 0; j<n; j++)
            z[i][j] = x[i][j]+y[i][j];
}
void read(int x[ ][10], int m, int n)
{
    int i, j;
    for(i = 0; i<m; i++)
        for(j = 0; j<n; j++)
        {
            scanf("%d", &x[i][j]);
        }
}
void print(int x[ ][10], int m, int n)
{
    int i, j;
    for(i = 0; i<m; i++)
    {   printf("\n");
        for(j = 0; j<n; j++)
            printf("%d\t", x[i][j]);
    }
}
```

Output

```
Enter the size of matrix :3 4
Enter the elements of 1'st matrix
0 0 0 0
1 1 1 1
2 2 2 2
Enter the elements of 2'nd matrix
1 2 3 4
1 2 3 4
1 2 3 4
1'st matrix is:
0    0    0    0
1    1    1    1
2    2    2    2
2'nd matrix is:
1    2    3    4
1    2    3    4
1    2    3    4
result matrix is:
1    2    3    4
2    3    4    5
3    4    5    6
```

Program Segment to Count non-zero Elements of a Matrix

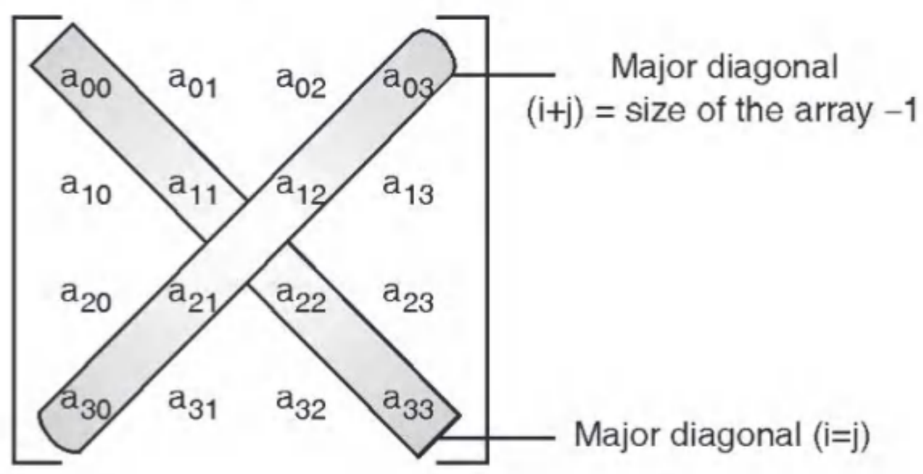
```
count = 0;
for(i = 0; i < m; i++)
    for(j = 0; j < n; j++)
        if(a[i][j] ! = 0)
            count ++;
printf("\n %d", count);
```

A matrix of m × n elements is scanned using the nested loop[row wise and inside the row column wise] and whenever a non-zero value is found, the value of count is incremented by 1.

Techknowledge Publication

Program Segment for finding the Sum of all Elements of: (a) Diagonal, (b) Lower triangle [minus main diagonal] (c) Upper triangle [minus main diagonal]

In order to understand the above; let us consider the case of a 4×4 matrix a[4][4].



Upper triangle

For any element a_{ij}, i < j i.e. row index is less than column index.

Main diagonal

For any element a_{ij}, i = j

(row index of the element = column index)

Lower triangle

For any element a_{ij}, i > j

i.e. row index > column index

Scanning and finding the sum of all elements of lower triangle.

```
sum = 0;
for(i = 1; i < m; i++)
    for(j = 0; j < i; j++)
        sum = sum + a[i][j];
```

- In the outer loop for (i = 1; i < m; i++), i starts from 1 as the first element of the lower triangle is a_{10}.
- Inner loop for (j = 0; j < i; j++) is used to scan the elements of the ith row, j varies from 0 to i – 1 as i > j for lower triangle.

Finding the sum of diagonal elements

```
sum = 0;
for(i = 0; i < m; i++)
    sum = sum + a[i][i];
```

Finding the sum of all elements of the upper triangle

```
sum = 0;
for(i = 0; i < m – 1; i++)
    for(j = i + 1; j < n; j++)
        sum = sum + a[i][j]
```

Algorithm to find product of diagonal elements

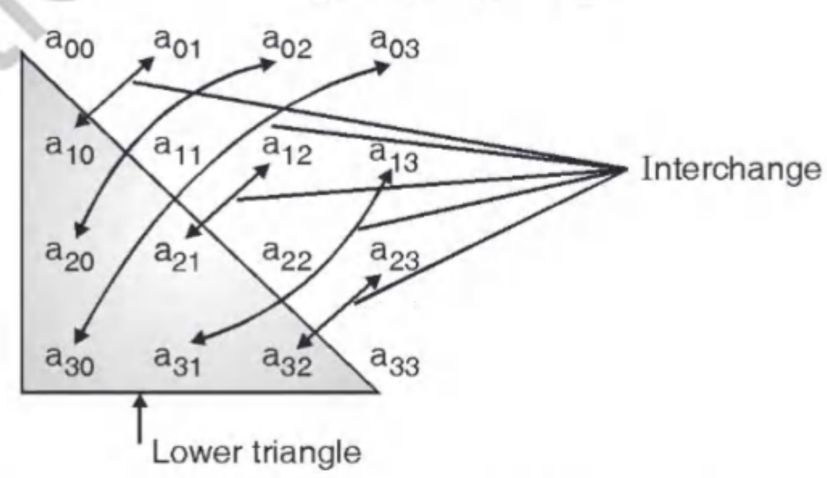
```
PROD = 1;
for (i = 0; i<n; i++)
        PROD = PROD * A[i][i];
printf("\n %d", PROD);
```

3.8.2 Transpose of Square Matrix

Transpose of a square matrix is found by interchanging rows and columns.

$$A = \begin{bmatrix} 5 & 1 & 9 & 6 \\ 2 & 0 & 8 & 4 \\ 9 & 6 & 5 & 2 \\ 1 & 0 & 0 & 1 \end{bmatrix}_{4 \times 4}$$

$$A^T = \begin{bmatrix} 5 & 2 & 9 & 1 \\ 1 & 0 & 6 & 0 \\ 9 & 8 & 5 & 0 \\ 6 & 4 & 2 & 1 \end{bmatrix}$$



Transpose of a square matrix can be found by interchanging all elements a_{ij} of the lower triangular matrix with the corresponding elements a_{ji}.

```
for (i = 1; i<m; i++)
for (j = 0; j<i; j++)
Interchange (a[i][j], a[j][i]);
```

loop to scan elements of the lower triangle

Program 3.8.2 : Program to find the transpose of square matrix.

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

Techknowledge Publication

```
{
    int a[10][10], m, i, j, temp;
    /* Actual size of matrix is m*n
        i, j : for scanning of array
        temp: for interchanging of a[i][j] and a[j][i]
    */
    printf("\n Enter the size of matrix :");
    scanf("%d", &m);

    /* Reading elements of matrix */
    printf("\n Enter the values a:");
    for(i = 0; i<m; i++)
        for(j = 0; j<m; j++)
            scanf("%d", &a[i][j]);

    //To print original square matrix
    printf("\n original square matrix is");
    for(i = 0; i<m; i++)
    {
        printf("\n ");
        for(j = 0; j<m; j++)
        printf("%d\t", a[i][j]);
    }

    /* Finding transpose */
    for(i = 1; i<m; i++)
        for(j = 0; j<i; j++)
        {
            temp = a[i][j];
            a[i][j] = a[j][i];
            a[j][i] = temp;
        }
    /* printing of all elements of final matrix */
    printf("\n Transpose matrix is :");
    for(i = 0; i<m; i++)
    {
        printf("\n ");
        for(j = 0; j<m; j++)
        printf("%d\t", a[i][j]);
    }
    getch();
}
```

Output

```
Enter the size of matrix :4
Enter the values :
5 1 9 6
2 0 8 4
9 6 5 2
1 0 0 1
original square matrix is
5    1    9    6
2    0    8    4
9    6    5    2
1    0    0    1
Transpose matrix is :
5    2    9    1
1    0    6    0
9    8    5    0
6    4    2    1
```

3.8.3 Finding whether a given Square Matrix is Symmetrical

- A square matrix is said to be symmetrical if the matrix $A = A^T$(A^T = transpose of A).
- A square matrix is said to be symmetrical if for all $a_{ij} \in$ lower triangle, $a_{ij} = a_{ji}$.

Program 3.8.3 : Program for checking of symmetry.

```
#include<stdio.h>
#include<conio.h>
int symmetry(int a[ ][10], int m);
void main()
{
    int a[10][10], m, i, j, result;
    printf("\n Enter the size of matrix :");
    scanf("%d", &m);
    /* Reading matrix elements */
    printf("\n Enter the values :");
    for(i = 0; i<m; i++)
        for(j = 0; j<m; j++)
        scanf("%d", &a[i][j]);
        result = symmetry(a, m);
```

Techknowledge Publication

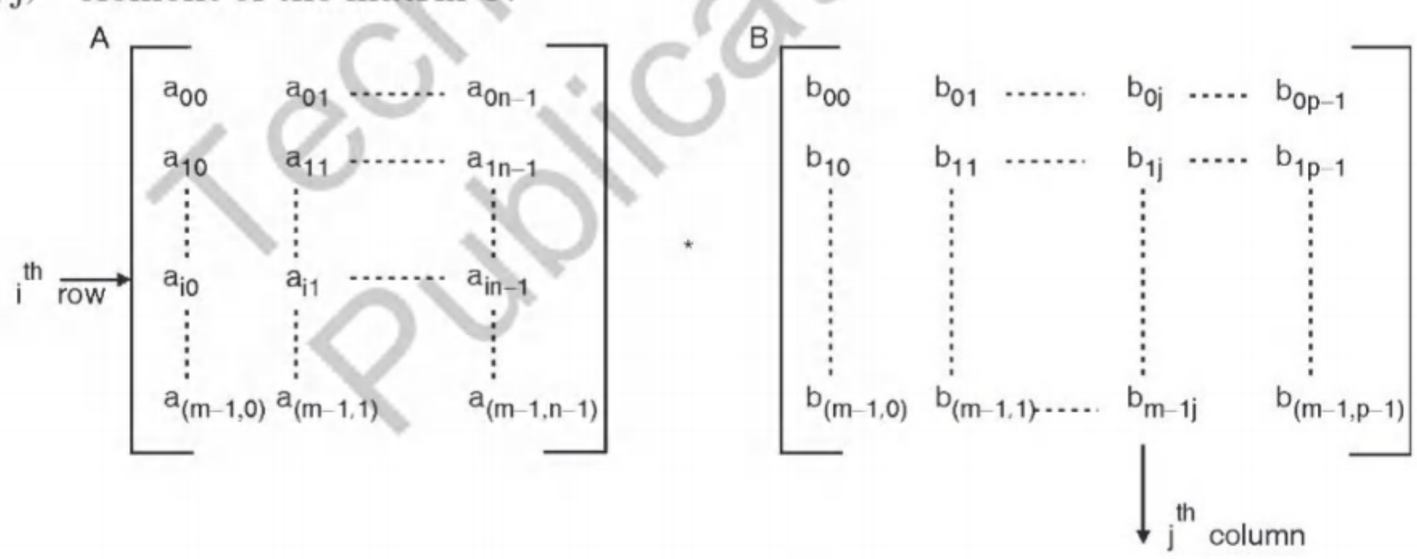
```
        if(result = = 0)
            printf("\n Not symmetrical");
        else
            printf("\n Symmetrical");
        getch();
}
int symmetry(int a[ ][10], int m)
{
    int i, j;
    for(i = 1; i<m; i++)
        for(j = 0; j<i; j++)
            if(a[i][j]! = a[j][i])
                return(0);
    return(1);
}
```

Output

```
Enter the size of matrix :4
Enter the values :
1 2 3 4
2 1 4 3
3 4 1 3
4 3 3 1
Symmetrical
-----------------------------------------
Enter the size of matrix :4
Enter the values :
1 2 3 4
1 2 3 4
1 2 3 4
1 2 3 4
 Not symmtrical
```

3.8.4 Multiplication of Two Matrices $A_{m \times n}$ and $B_{n \times p}$

– In order that two matrices A and B are multiplication compatible, number of columns of matrix A should be equal to the number of rows of the matrix B. The resultant matrix C will have a size of m × p method for finding the $(i, j)^{th}$ element of the matrix C.



C_{ij} = dot product of the ith row of A and the jth column of B

$$\therefore \quad C_{ij} = a_{i0} \times b_{0j} + a_{i1} \times b_{1j} + a_{i2} \times b_{2j} + \ldots + a_{in-1} \times b_{n-1\,j} = \sum_{k=0}^{n-1} a_{ik} \times b_{kj}$$

Equivalent loop for finding C_{ij} in C-language

```
C[i][j]  =  0;
for (k = 0; k < n; k ++)
    C[i][j]  =  C[i][j] + a[i][k] * b[k][j];
```

Once we know how to calculate C_{ij}, we can calculate all elements of the resultant matrix C by varying i from 0 to n – 1 and j from 0 to p – 1.

TechKnowledge Publications

Program 3.8.4 : Program for multiplication of two matrices.

```
#include<stdio.h>
#include<conio.h>
void read(int[ ][10], int, int);
void print(int[ ][10], int, int);
void multiply(int[ ][10], int[ ][10], int[ ][10], int, int,
int);
void main()
{
    int a[10][10], b[10][10], c[10][10], m, n, p;

    /* first matrix of size m*n       (a)
       second matrix of size n*p      (b)
       resultant matrix of size m*p   (c)
    */
    printf("\n Enter the size of 1'st matrix :");
    scanf("%d%d", &m, &n);
    printf("\nEnter no. of columns in the 2'nd matrix :");
    scanf("%d", &p);
    printf("\n Enter the data of 1'st matrix :");
    read(a, m, n);
    printf("\n Enter the data of 2'nd matrix :");
    read(b, n, p);
    multiply(a, b, c, m, n, p);
    printf("\n Result is :");
    print(c, m, p);
}
void read(int a[ ][10], int m, int n)
{
    int i, j;
    for(i = 0; i<m; i++)
        for(j = 0; j<n; j++)
            scanf("%d", &a[i][j]);
}
void print(int a[ ][10], int m, int n)
{
    int i, j;
    for(i = 0; i<m; i++)
    {
        printf("\n ");
            for(j = 0; j<n; j++)
                printf("%d\t", a[i][j]);
    }
}
void multiply(int a[ ][10], int b[ ][10], int c[ ][10], int
m, int n, int p)
{
    int i, j, k;
    for(i = 0; i<m; i++)
        for(j = 0; j<p; j++)
        {
            c[i][j] = 0;
            for(k = 0; k<n; k++)
                c[i][j] = c[i][j]+a[i][k]*b[k][j];
        }
}
```

Output

```
Enter the size of 1'st matrix :2 2
Enter no. of columns in the 2'nd matrix :2
Enter the data of 1'st matrix :
2 2
2 2
Enter the data of 2'nd matrix :
3 3
3 3
Result is :
12      12
12      12
```

3.8.5 Saddle Point

An m × n matrix is said to have a saddle point if some entry a[i][j] is the smallest value in row i and largest value of column j.

Write C function to determine location of saddle point if exists.

```
int saddle(int a[10][10], int m, int n)
{
    for(i = 0; i<m; i++)
    {   //check each row for saddle point
     // find the smallest value of row i
```

TechKnowledge Publications

```
        rowvalue = a[i][0];
        rowloc = 0;
        for(j = 1; j < n; j++)
        if(rowvalue > a[i][j])
        {
            rowvalue = a[i][j];
            rowloc = j;
        }
    //find the largest value in column given by rowloc
        colvalue = a[0][rowloc];
        colloc = 0;
        for(j = 1; j < m; j++)
        if(colvalue < a[j][rowloc])
        {   colvalue = a[j][rowloc]);
            colloc = j;
        }
        if(i = colloc)
        {
         printf("\n saddle point at(%d,%d)",i,rowloc);
         return(i);
        }
    }
        printf("\n saddle point does not exist:");
        return(0);
}
```

3.9 Character String in C-Language

- Strings in C-language are declared as character array. Each character takes one byte of memory.
- String always terminates with a NULL character and it is represented as '\0'. Its ASCII code is 0. The Fig. 3.9.1 shows how the string "INDIA" is stored in the array.

```
Char a[6] = "INDIA"
```

'I'	'N'	'D'	'I'	'A'	'\0'

Fig. 3.9.1 : Storing "INDIA" in an array

- Loop for scanning an array :

```
for (i = 0; a[i] ! = '\0'; i ++)
```

It may be noted that the loop terminates when the '\0' is found.

- Reading a string
  - (a) scanf("%s",a); /* First word of the string will be stored in the character array a[ ] */
  - (b) gets(a); /* gets( ) is another function for reading a string this function can read a string having a number of words */
  - (c) i = 0;

```
while ((a[i] = getchar ( )) ! = '\n ')
    i++ ;
a[i] = '\0';            \\ replace ' \n ' by ' \0'
```

A string is read character by character and stored in the array a[ ] · Loop terminates on reading of end of line character ('\n '). Since the last character of a string is '\0', '\n ' is replaced by '\0'.

- Displaying a string

```
printf ("\n %s", a);
```

- Library functions for strings
  - **(a) strlen(a) :** Gives length of a string
  - **(b) strcat(string1, string2) :** Concatenates two strings, string2 is appended to string1.
  - **(c) strcmp(string1, string2) :** It compares two strings and returns 0 if both the strings are same positive value if string1 > string2 negative value if string1 < string2
  - **(d) strcpy(string1, string2) :** Assigns the contents of string2 to string1.

Program 3.9.1 : Program showing usage of string handling functions.

```
#include<string.h>
#include<stdio.h>
#include<conio.h>
void main()
{
    char s1[30], s2[30];
    int l1, l2;
```

Techknowledge Publications

```
    //reading a string and finding its length
    printf("\n Enter the string :");
    gets(s1);
    l1 = strlen(s1);
    printf("\n length of\t%s = %d", s1, l1);
    //usage of string copying function
    strcpy(s2, s1);
    printf("\n 1st string = %s", s1);
    printf("\n 2nd string = %s", s2);
    //usage of string compare function
    l1 = strcmp(s1, s2);
    if(l1 = = 0)
        printf("\n s1 = s2");
    else
        if(l1>0)
            printf("\n s1>s2");
        else
            printf("\n s1<s2");
    /*string concatenation function*/
    strcat(s1, s2);
    printf("\n final string after concatination=%s",s1);
}
```

Output

```
Enter the string :India
length of India = 5
1st string = India
2nd string = India
s1 = s2
final string after concatination = IndiaIndia
```

Program 3.9.2 : Program to find length of string.

```
#include<stdio.h>
#include<conio.h>
void main()
{   //reading a string and finding its length
    char s[100];
    int l;
    printf("\n Enter the string :");
    gets(s);
    l = 0;
    while(s[l]! = '\0')
        l++;
    printf("\n length of '%s' is = %d", s, l);
    getch();
}
```

Output

```
Enter the string :It is India
length of 'It is India' is = 11
```

C-function for finding length of a string.

```
int length(char a[ ])
{
    int i = 0
    while(a[i]! = '\0')
        i++;
    return(i);
}
```

Program 3.9.3 : Program to copy a string.

```
#include<stdio.h>
#include<conio.h>
void main()
{   char s1[100], s2[100];
    int i;
    //reading a string and finding its length
    printf("\n Enter the string :");
    gets(s1);
    /* array is scanned from left to right and while
    scanning s1, each character is copied to the array
    s2 */
    i = 0;
    while(s1[i]! = '\0')
    {
        s2[i] = s1[i];
        i++;
    }
    //since the '\0' is not copied
    s2[i] = '\0';
    printf("\n result is %s ", s2);
    getch();
}
```

Techknowledge Publication

Output

```
Enter the string :data-structure
result is data-structure
```

3.9.1 String Comparison

Two strings are compared from left to right and the first place of mismatch will terminate the loop for comparison.

1. 1st string s1 = ABCCDC \0 2nd string s2 = ABBCDF \0 ↑ Place of mismatch	s1 > s2, since at the place of mismatch, character 'C' in s1 comes after character 'B' in s2
2. s1 = ABD \0 s2 = ABDFL \0 ↑ Place of mismatch	s2 > s1 since 'F' > '\0'
3. s1 = ABD \0 s2 = ABD \0	s1 = s2 Since both the strings are same

```
loop for comparison
    i = 0;
    while(s1[i] = = s2[i] && s1[i] ! = '\0')
        i++;
```

The first condition s1[i] = = s2[i] takes care that the loop will terminate at the first place of mismatch.

The second condition s1[i]! = '\0' will fail when both the strings are same.

Program 3.9.4 : Program to compare two strings.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    char s1[50], s2[50];
    int i;
    //reading a string and finding its length
    printf("\n Enter two strings :");
    gets(s1);
    gets(s2);
    //loop for comparison
    i = 0;
    while(s1[i] = = s2[i] && s1[i]! = '\0')
    //locate the place of mismatch
        i++;
    if(s1[i] > s2[i])
        printf("\n s1 > s2");
    else
        if(s1[i] < s2[i])
            printf("\n s1 < s2");
            else
                printf("\n s1 = s2");
    getch();
}
```

Output

```
Enter two strings :India
India
s1 = s2
------------------
Enter two strings :maharashtra
maha
s1 > s2
```

3.9.2 Reversing a String

Q. Give pseudo C/C++ code to reverse the string. **SPPU - May 18, 3 Marks**

Q. Write pseudo C/C++ code for reversing a string and state its time complexity. **SPPU - Dec. 18, 4 Marks**

Pseudo C algorithm to reverse a string

Example:

Input string = "INDIA"

Output string = "AIDNI"

– Position i at the beginning and j at the end of the string.

Technknowledge Publication

I N D I A \0

i = 0 j = 4

Interchange i^{th} character with the jth character

– i = i + 1 and j = j – 1 and continue interchanging i^{th} character with the jth character as long as i < j

A N D I I \0

i = 1 j = 3

Loop terminates as the condition i < j fails.

A I D N I \0

i = 2

j = 2

Time Complexity = O(n)

Program 3.9.5 : Program for reversing a string

```
#include<stdio.h>
#include<string.h>
void main()
{
    char s[100], temp;
    int i, j = 0;
    printf("\n Enter the string :");
    gets(s);
    i = 0;
    j = strlen(s)–1;
    while(i<j)
    {
        temp = s[i];
        s[i] = s[j];
        s[j] = temp;
        i++; j– –;
    }
    printf("\n Reverse string is :%s", s);
    getch();
}
```

Output

```
Enter the string : maharashtra
Reverse string is : arthsaraham
```

'C' function for palindrome.

```
int palindrome(char a[ ])
{
    int i = 0, j;
    j = strlen(a) – 1;
    while(i<j)
    {
        if(a[i] ! = a[j])
            return(0);      //Not a palindrome
        i++;
        j– –;
    }
    return(1);              //A palindrome
}
```

3.9.3 String Searching

For Example

MAIN STRING = "INDIA IS A BIG COUNTRY"

SEARCH STRING = "BIG"

Result = Found (location = 12)

String searching is a common problem and it is very useful in text manipulation programs.

Algorithm

```
i = 0; /* index for the main string A[ ]*/
j = 0 /* index for the search string B[ ]*/
while(the search string is not found && search can continue)
{
    search for the 0th character of B[ ] (B[0]) in the main string A[ ]
        while(A[i] ! = B[0])
            i++;
    if(B[0] is found in A[ ])
    then remember the location i
    k = i, j = 0
    Match the remaining characters of B[ ]
```

Techknowledge Publication

```
    while(A[i]==B[j] && A[i]! = '\0' && B[j]!='\0')
    {   i++; j++;       }
    If the entire string of B does not match then
    continue searching from the location k + 1
}
```

If the end of the string A has come then search cannot continue. If the end of the string B has come then the search string has been found.

Program 3.9.6 : Program to find substring.

```
#include<stdio.h>
#include<string.h>
int search(char[ ], char[ ]);
void main()
{
    char a[100], b[40];
    int loc;
    printf("\n Enter the main string :");
    gets(a);
    printf("\n Enter the search string :");
    gets(b);
    loc = search(a, b);
    if(loc = = –1)
        printf("\n not found");
    else
    printf("\n found at location %d", loc+1);
    getch();
}
int search(char a[ ], char b[ ])
{
    int i, j, k;
    i = 0, j = 0;
    while(a[i]! = '\0')
    {
        while(a[i]! = b[0] && a[i]! = '\0')
            i++;
        if(a[i] = = '\0')
            return(–1);        //search can not continue
        k = i;
        while(a[i]==b[j] && a[i]!= '\0' && b[j]!= '\0')
        {   i++; j++;
        }
        if(b[j] = = '\0')
            return(k);
        if(a[i] = = '\0')
            return(–1);
        i = k+1;
        j = 0;
    }
}
```

Output

```
Enter the main string :india
Enter the search string :di
found at location 3
```

3.9.4 String Concatenation

Q. Give pseudo C/ C++ code to concatenate two strings. **SPPU - Dec. 17, 3 Marks**

Second string is copied at the end of the first string.

first string	X	Y	Z	\0
second string	A	B	C	\0

string after concatenation	X	Y	Z	A	B	C	\0

Program 3.9.7 : Program to concatenate two strings.

```
#include<stdio.h>
#include<string.h>
void concatenate(char[ ], char[ ]);
void main()
{
    char s1[50], s2[30];
    printf("\n Enter two strings :");
    gets(s1);
    gets(s2);
    concatenate(s1, s2);
    printf("\n final string is :%s", s1);
    getch();
}
void concatenate(char s1[ ], char s2[ ])
{
    int i, j;
```

TechKnowledge Publications

```
    i = strlen(s1);
    for(j = 0; s2[j]! = '\0'; i++, j++)
        s1[i] = s2[j];
        s1[i] = '\0';
}
```

Output

```
Enter two strings :  maha
                     rashtra
final string is : maharashtra
```

Program 3.9.8 : Program to convert a decimal number from string format to integer format.

Example : Input

'5'	'9'	'3'	'\0'

Output A decimal number 593

ASCII code of '5' = 53

SCII code of '9' = 57

ASCII code of '3' = 51

((53 – 48) × 10 + (57 – 48)) × 10 + (51 – 48) = 593

⇓ ⇓ ⇓

integer 5 integer 9 integer 3

```
#include<conio.h>
#include<stdio.h>
void main()
{
    char s1[30];
    int x, i;
    printf("\n Enter the number :");
    gets(s1);    //number stored in string format
    x = 0;
    for(i = 0; s1[i]! = '\0'; i++)
        x = x*10+s1[i]–48;
    printf("\n number in integer format is :%d", x);
    getch();
}
```

Output

```
Enter the number :543
number in integer format is :543
```

A function substring( ) in C which returns a substring of length n characters, starting from m^th character of string.

```
char * substring(char a[ ], int m, int n)
{
    char *P;
    /* To store the address of a dynamic array of
    character */
    P = (char *) malloc(n + 1);
    for(i = 0; i < n; i++)
        P[i] = a[m + i];
    P[i] = '\0';
    return(P);
}
```

Ex. 3.9.1 : Write pseudo 'C' code to compact a given string to replace sequence of blanks into a single blank.

Soln. :

```
main()
{
    char a[100];
    int i, j, flag ;
    i = 0 ;     //Last character of the compacted string
    j = 0 ;     // For scanning the original string
    flag = 1;  // flag = 0, to skip the additional blanks
    /* read a string */
    read a ;
    while(a[j] is not the end of string )
    {   if(a[i] is a blank)
        {   if( flag = = 1)
            {   flag = 0 ;
                a[i] = a[j] ;
                i ++ ; j++ ;
            }
            else
                j++ ;           // skip blanks
        }
        else
        {   a[i] = a[j] ;
```

Techknowledge Publication

```
            i++ ;
            j++ ;
            flag = 1 ;
        }
    }
}
Print a // display the result
```

Program 3.9.9 : Accept a string and count number of characters, words and special characters in string.

Let us understand the working of the above program with the help of an example :

India, China and USA are big countries.

↑ ↑ ↑ ↑ ↑ ↑ ↑

Beginning of a word as shown with ↑ satisfies the following condition. Either the word begins from location 0 or the Previous character is non-alphanumeric (alphabet or digit).

If the string is stored in array a[ ] and the word begins at location i then :

if(isalnum (a[i]) & (i = = 0 ¦ ¦ ! isalnum (a[i – 1])))

will evaluate to true.

A special character is a character, which is neither a space nor an alphanumeric.

```
#include<stdio.h>
#include<ctype.h>
void main()
{
    char a[80];
    int words, special, characters, i;
    words = special = characters = 0;
    printf("\n Enter the string :");
    gets(a);
    for(i = 0; a[i]! = '\0'; i++)
    {
        characters++;
        if(isalnum(a[i]) &&(i==0||! isalnum(a[i–1])))
            words++;
        if(!isspace(a[i]) && !isalnum(a[i]))
            special++;
    }
    printf("\n Number of characters = %d", characters);
    printf("\n Number of words = %d", words);
    printf("\n Number of special characters = %d", special);
}
```

Output

```
Enter the string :India,China and USA are big countries.
Number of characters = 38
Number of words = 7
Number of special characters = 2
```

3.10 Ordered List

Q. Define and explain the term Ordered List. **SPPU - May 18, 1 Mark**

– It is a linear list of elements arranged in a particular order.

Example

1. List of integer numbers arranged in ascending order (5, 11,20, 30, 45, 90)
2. Days of the week

   (MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY)

– There are a variety of operations that are performed on these lists :

– These operations include :

(1) Finding the length of the list

(2) Retrieving the ith element.

(3) Searching for an element.

(4) Inserting an element at the ith position.

(5) Deleting element from ith position.

(6) Displaying a list.

– A list can be represented in an array.

Tech Knowledge Publications

3.11 Representation of a Polynomial of Degree n in an Array

Q. Explain polynomial representation using arrays with suitable example. **SPPU - May 17, 3 Marks**

Q. Explain polynomial representation using an array with suitable example. **SPPU - May 18, 2 Marks**

Suppose, we have a polynomial :

$$C_{m-1}x^{m-1} + C_{m-2}x^{m-2} + \ldots\ldots + C_0x^0 \qquad \ldots(1)$$

and it is to be represented in an array.

Some of the coefficients, C_i in above polynomial could be zero. We will store terms with non-zero coefficients. Let us suppose that the polynomial has n non-zero terms.

Polynomial in Equation(1), with n terms can be represented by the ordered list of length 2n + 1 :

$$\left(n, \frac{\text{(Power , coefficient)}}{1^{st}\text{ term}}, \frac{\text{(Power , coefficient)}}{2^{nd}\text{ term}}, \ldots\right)$$

For example, $x^3 + 5x^2 + 9$ can be represented as

(3, (3, 1), (2, 5), (0, 9))

Or

0	1	2	3	4	5	6
3	3	1	2	5	0	9

in an array

Ex. 3.11.1 : Represent the following polynomial using an array : $x^4 + 59x + 10$

Soln. :

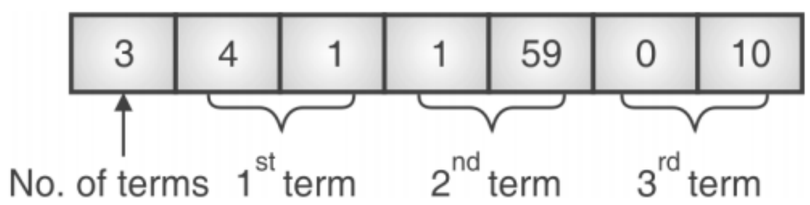
A polynomial in one variable and having n terms can be represented by an ordered list of length 2n + 1.

(n, (power, coefficient), (power, coefficient), ...)

Given polynomial = $x^4 + 59x + 10$

Representation of the polynomial will be as given below : (3, (4, 1), (1, 59), (0, 10))

storing the list in an array :



3.11.1 Pseudo C - Algorithm for Addition of Two Polynomials

```
void polyadd(A, B, C)
{
    /*  A(0 : 2m), B(0 : 2n), C(0 : 2(m + n)) */
    m = A(0);                /* number of terms in A */
    n = B(0);                /* number of terms in B */
    i = j = k = 1;

    while( i < = 2m and j < = 2n)
    {
    if(A(i) is equal to B(j))          //powers are same
    {
        C(k) = A(i);
        C(k + 1) = A(i + 1) + B(j + 1)
                                /* add coefficients */
        i = i + 2; j = j + 2; k = k + 2;
                                /* advance to next term */
    }
    else
        if(A(i) > B(j))
        {
            C(k) = A(i);
            C(k + 1) = A(i + 1);
                                /* store new term */
            i = i + 2; k = k + 2;
                                /* Advance to next term */
        }
        else
        {   C(k) = B(j);
            C(k + 1) = B(j + 1);
                                /* store new term */
            j = j + 2;
            k = k + 2;
        }
    }
    while(i < = 2m)        //copy remaining terms of A
    {
        C(k) = A(i);
        C(k + 1) = A(i + 1);
        i = i + 2;
```

TechKnowledge Publications

```
        k = k + 2;
    }
    while(j < = 2n)   /* copy remaining terms of B*/
    {
        C(k) = B(j);
        C(k + 1) = B(j + 1);
        j = j + 2;
        K = K + 2;
    }
    C(0)  = K/2 ;     /*number of terms in the sum  */
}
```

Time Complexity : O (m + n)

3.11.2 C-Function for Addition of Two Polynomials Represented in an Array

Q. Write pseudo C/C++ code to perform addition of two polynomials using arrays. **SPPU - Dec. 18, 6 Marks**

```
void addp(int a[ ], int b[ ], int c[ ])
{
    int m, n, i, j, k;
    m = a[0];
    n = b[0];
    i = j = k = 1;
    while(i< = 2*m && j< = 2*n)
    {
        if(a[i] > b[j])
        {
            c[k] = a[i];
            c[k+1] = a[i+1];
            k = k+2;
            i = i+2;
        }
        else
        if(a[i] < b[j])
        {
            c[k] = b[j];
            c[k+1] = b[j+1];
            k = k+2;
            j = j+2;
        }
        else
        {
            c[k] = b[j];
            c[k+1] = a[i+1]+b[j+1];
            k = k+2;
            i = i+2;
            j = j+2;
        }
    }
    while(i < = 2*m)
    {
        c[k] = a[i];
        c[k+1] = a[i+j];
        k = k+2; i = i+2i
    }
    while(j< = 2 = *n)
    {
        c[k] = b[j]
        c [k +1] = b[j+1]
        k = k+2; j = j+2;
    }
    c(0) = k/2;
}
```

3.11.3 Representing a Polynomial A(x, y)

If the above polynomial has m non-zero terms then the polynomial can be represented as an ordered list of 3m + 1 terms.

m, Power of x, Power of y, Coefficient	Power of x, Power of y, Coefficient ....
1st term	2nd term

First entry is the number of non-zero terms. Then, for each term there are three entries representing an exponent of x, exponent of y, coefficient triple.

List should be ordered on power of x and inside power of x, it should be ordered on power of y.

Techknowledge Publication

Ex. 3.11.2 : Represent the following polynomial using an array : $3x^3 y^2 + 10xy - 1$

Soln. :

A polynomial in two variables and having n terms can be represented by an ordered list of length 3n + 1.

(n, (power of x, power of y, coefficient), ...)

Given polynomial = $3x^3 y^2 + 10xy - 1$

Representation of the polynomial will be as given as follows : (3, (3, 2, 3), (1, 1, 10), (0, 0, – 1))

storing the list in an array :

3	3	2	3	1	1	10	0	0	–1
No. of terms	1st term			2nd term			3rd term		

Ex. 3.11.3 : Represent the following polynomials using arrays (i) $x^2 + 5xy + y^2 + y - x$ (ii) $5x^2 + 10xy + y^2 + 20$

Soln. : (i) Rewriting the polynomial we get,

$x^2y^0 + 5xy + x^0y^2 + x^0y - xy^0$

Ordering the terms :

We get, $x^2 y^0 + 5xy - x y^0 + x^0 y^2 + x^0 y$

(ii) Array representation :

5	2	0	1	1	1	5	1	0	–1	0	2	1	0	1	1
No.of terms	1st term			2nd term			3rd term			4th term			5th term		

(iii) Re-writing the polynomial

We get, $5x^2 y^0 + 10 xy - x^0 y^2 + 20 x^0 y^0$

4	2	0	5	1	1	10	0	2	1	0	0	20
No.of terms	1st term			2nd term			3rd term			4th term		

Ex. 3.11.4 : Represent the following polynomial using two-dimensional array : (i) $x^2 + xy + 2x^2y$ (ii) $3x^3 + 2y^2x + 5y^3x^3$.

Soln. : (i) $x^2 + xy + 2x^2y$

Power of x ↑	0	1	2	→Power of y
0	0	0	0	
1	0	1	0	
2	1	2	0	

(ii) $2x^3 + 2y^2 x + 5y^3 x^3$

Power of x ↑	0	1	2	3	→ Power of y
0	0	0	0	0	
1	0	0	2	0	
2	0	0	0	0	
3	3	0	0	5	

Ex. 3.11.5 : Write data structure to represent polynomial in two variables. Represent the following polynomials using the declared datastructure in graphical form : (i) $3 + 2x + 9xy^3$ (ii) $x^3 + y^4 + 2x^3y^4$ (iii) $5x^2y^3 + 3x^2 + 4xy + 2$

Soln. :

Data structure to represent a polynomial in two variables. A 2D integer array.

(i) $3 + 2x + 9xy^3$

Power of x ↑	0	1	2	3	→ Power of y
0	3	0	0	0	
1	2	0	0	9	
2	0	0	0	0	
3	0	0	0	0	

(ii) $x^3 + y^4 + 2x^3y^4$

Power of x ↑	0	1	2	3	4	→ Power of y
0	0	0	0	0	1	
1	0	0	0	0	0	
2	0	0	0	0	0	
3	1	0	0	0	2	
4	0	0	0	0	0	

(iii) $5x^2 y^3 + 3x^2 + 4xy + 2$

Power of x ↑	0	1	2	3	→ Power of y
0	2	0	0	0	
1	0	4	0	0	
2	3	0	0	5	
3	0	0	0	0	

Technknowledge Publication

Ex. 3.11.6 : Represent the following polynomials using arrays : (i) $x^3 + 2xy + y^3 - y + x$
(ii) $5x^2 + 10xy + y^2 - 20$.

Soln. :

(i) $x^3 + 2xy + y^3 - y + x$

		0	1	2	3	← Power of y
	0	0	– 1	0	1	
↑	1	1	2	0	0	
Power of x	2	0	0	0	0	
	3	1	0	0	0	

(ii) $5x^2 + 10xy + y^2 - 20$.

		0	1	2	← Power of y
	0	– 20	0	1	
↑	1	0	10	0	
Power of x	2	5	0	0	

3.11.4 Representing a Polynomial A (x,y,z)

If the above polynomial has m non-zero terms then the polynomial can be represented as an ordered list of Y_{m+1} terms.

M, Power x, power of y, power of z, coefficient ....

1st terms

First entry is the number of non-zero terms, Then, for each terms there are four entries representing an exponent of x, exponent of y, exponent of z, coefficient.

List should be ordered on power of x, it should be ordered on power of y and inside power of y it should be ordered on power of z.

Ex. 3.11.7 : Represent the following polynomial in computer memory using the structure defined by you.
$5x^3y^2z + 3x^2y^3z^2 + 6xyz^3 + 10$

Soln. :

4	3	2	1	5	2	3	2	3	1	1	3	6	0	0	0	10
No. of terms	1st term				2nd term				3rd term				4th term			

Ex. 3.11.8 : Using an array, represent the following polynomial p(x, y, z) in computer memory:
$p(x, y, z) = 10\, x^3 y^2 z - 8\, x^3 y^3 z^3 + 5xyz^2 + 10$

Soln. :

A polynomial in three variables and having n terms can be represented by the ordered list of length 4n + 1.

(n, (power of x, Power of y, Power of z, coefficient), ...)

Given polynomial : $10\, x^3y^2z - 8\, x^3 y^3 z^3 + 5xyz^2 + 10$

Can be re-written as : $-8\, x^3 y^3 z^3 + 10\, x^3y^2 z + 5xyz^2 + 10x^0y^0z^0$

Thus, the representation of the polynomial will be as given below :

(4, (3, 3, 3, –8), (3, 2, 1, 10), (1, 1, 2, 5), (0, 0, 0,10))

Storing the list in an array :

4	3	3	3	–8	3	2	1	10	1	1	2	5	0	0	0	10
No. of terms	1st term				2nd term				3rd term				4th term			

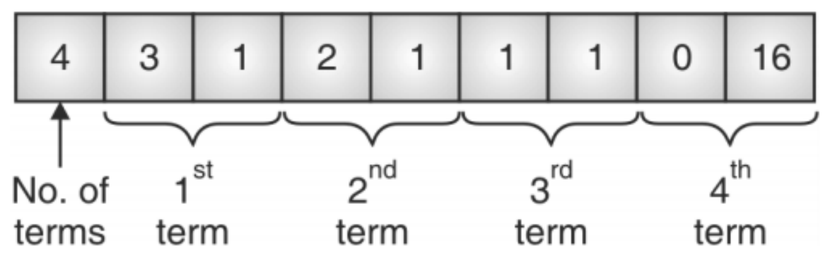
Tech Knowledge Publications

Ex. 3.11.9 : Represent the following polynomial using array : (i) $x^3 + x^2 + x + 16$ (ii) $x^5y^4 + x^3y^3 + x^2 + y^2 + 10$

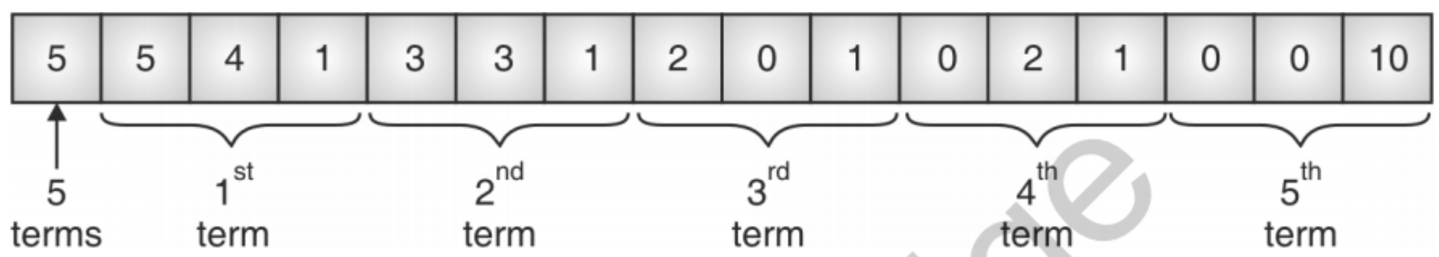
Soln. :

(i) $x^3 + x^2 + x + 16$

The polynomial $x^3 + x^2 + x + 16$ will be represented as :



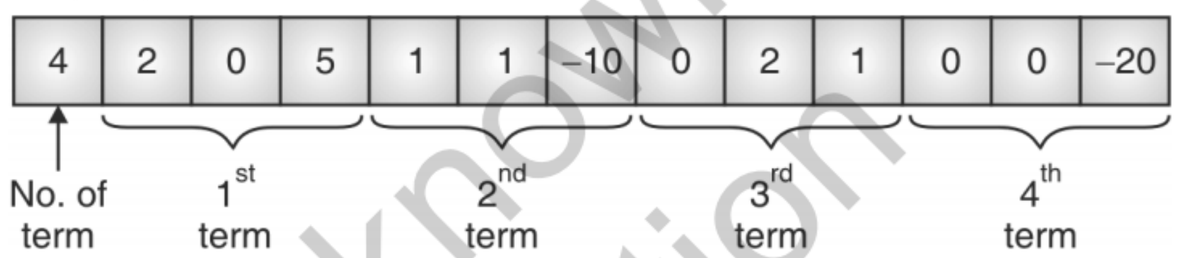
(ii) The polynomial $x^5 y^4 + x^3 y^3 + x^2 + y^2 + 10$ will be represented as :



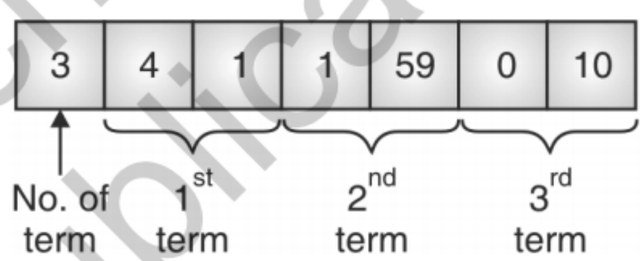
Ex. 3.11.10 : Represent the following polynomials using arrays : (i) 5x ^ 2 – 10xy + y ^ 2 – 20 (ii) x ^ 4 + 59x + 10

Soln. :

(i) The polynomial $5x^2 - 10xy + y^2 - 20$ will be represented as :



(ii) The polynomial $x^4 + 59x + 10$ will be represented as :



3.12 Polynomial as an Ordered List (using Structure)

– A polynomial of degree n can be represented as an ordered list using arrays. A polynomial P(x) of degree n is defined as the following expression.

$$P(x) = a_0 + a_1 x + a_2 x^2 + ... + a_n x^n$$

Where a_i is any real number and n is an integer.

– A term of a polynomial can be thought of as tuple (power, coefficients).

– A polynomial $5 + 9x + 8x^5 + 17x^9$ can be represented in the tuple form as
((0,5) (1,9) (5,8) (9,17))

tuple (0,5) stands for $5x^0$

tuple (1,9) stands for $9x^1$

tuple (5,8) stands for $8x^5$

tuple (9,17) stands for $17x^9$

The list of tuples (terms $a_i x^i$) is sorted on power

– A polynomial may be defined by the following structures

```
typedef struct term
{    int power;
     float coeff;
}term;
typedef struct polynomial
```

Tech Knowledge Publications

```
{
    term a[10]; /* an array of 10 terms */
    int n; /* Actual number of terms in the list */
} polynomial;
```

– There are a variety of operations that are performed on polynomials.

– These operations include

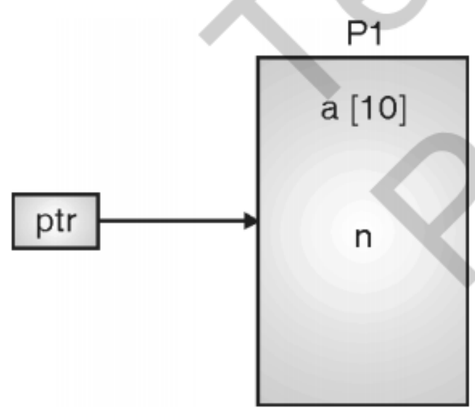
1. Initialize a polynomial
2. Insert a term
3. Read a polynomial
4. Print a polynomial
5. Add two polynomials
6. Multiply two polynomials
7. Evaluate a polynomial

Accessing elements of a polynomial.

Polynomial P1;
/* P1 is a polynomial type variable */

a[10]	P1
n	

P1 is a structure type variable having two fields n and a[10]. n is number of elements in the list and the array a[10] is used to represent a list of terms (power, coefficient).



P1·n → number of terms in polynomial P1

P1·a[i] → i^{th} term of the polynomial

P1·a[i]·power → power field of the ith term

P1·a[i]·coeff → coefficient field of the ith term.

Accessing elements of a polynomial through a pointer.

Polynomial P1, *Ptr;

Ptr = &P1 : Address of P1 is stored in Ptr

Ptr → a[i] : i^{th} term of the polynomial

Ptr → n : Number of terms in polynomial

(Ptr → a[i])·power : Power field of the ith term

(Ptr → a[i])·coeff : Coefficient field of the ith term

3.12.1 Function for Initialisation of a Polynomial

– Initially, number of terms in a polynomial is 0.

– Before using a polynomial type variable, it should be initialized.

```
void init(Polynomial * Ptr)
{
    Ptr → n = 0;
}
```

3.12.2 Inserting a Term

Example

Existing polynomial : $5 + 9x + 10x^3 + 6x^8$

Representation of polynomial

a[(0, 5), (1, 9), (3, 10), (8, 6)]
n = 4

New term to be inserted $3x^2$ ((2, 3))

[(0, 5), (1, 9), (3, 10), (8, 6)]

↑

place of insertion for (2, 3) i.e. $3x^2$

In order to insert the term $3x^2$ at the right place, all terms having higher power, namely (3, 10) and (8, 6) must be moved right by one place.

[(0, 5), (1, 9), (2, 3), (3, 10), (8, 6)]

Polynomial after insertion of $3x^2$

Value of n should also be increased by 1 as number of terms after insertion increases by 1.

'C' function to insert a term in polynomial.

```
void insert(polynomial * ptr, term t)
{
    int i;
    /* Move all higher power term by 1 place·
    Movement should start with the last term */
```

Tech Knowledge Publications

```
    for(i = ptr → n – 1 ; (ptr → a[i])·power > t · power
    && i > = 0; i– –)
    ptr → a[i + 1] = ptr → a[i];
    /* insert the term t */
    ptr → a[i + 1] = t;
    ptr → n = ptr → n + 1;
}
```

3.12.3 Reading a Polynomial

Entire polynomial can be read term by term and stored in a polynomial type variable through the use of insert function.

'C' function to read polynomial.

```
void read(polynomial *Ptr)
{
    int n, i, power;
    float coeff;
    term t;
    init(Ptr);
    printf("\n Enter number of forms :");
    scanf("%d", &n);                /* read n terms */
    for(i = 0; i < n; i++)
    {
        scanf("%d%f", &power, &coeff);
        t·power = power;
        t·coeff = coeff;
        insert(Ptr, t);
    }
}
```

3.12.4 Printing a Polynomial

Example : Given polynomial : $9 + 3x^2 + 10x^4 + 11x^6$

Representation : ((0,9), (2,3), (4,10), (6,11))

Output of the print program :

9 x^0 3 x^2 10 x^4 11 x^6

```
void print(Polynomial *Ptr)
{
    int i
    printf("\n ");
    for(i = 0; i < Ptr → n; i++)
    print(" %5.2 f X∧ % d \t", (Ptr → a[ i ])·Coeff, (Ptr
    → a[i])·power);
}
```

3.12.5 Addition of Two Polynomials

Example

First polynomial	$5 + 9x + 11x^4 + 23x^6$
Second polynomial	$6 + 9x + 3x^2$
Result =	$11 + 18x + 3x^2 + 11x^4 + 23x^6$

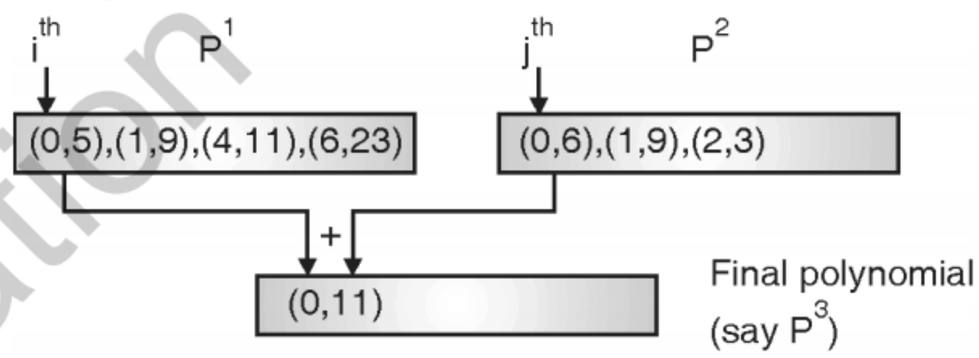
– Step by step, calculation of sum of two polynomials.

First polynomial : $5 + 9x + 11x^4 + 23x^6$ (Say P1)

Representation of P1 ((0, 5), (1, 9), (4, 11), (6, 23))

Second polynomial : $6 + 9x + 3x^2$ (say P2)

Representation of P2 ((0, 6), (1, 9), (2, 3))



Initially i = 0, j = 0;

– If the power of the ith term of P1 is equal to the power of the jth term of P2 then coefficient must be added.

i.e. if (P1→a[i]·power = = P2→a[j]·power) then the term (P1→a[i]·power), (P1→a[i]·coeff + P2→a[j]·coeff) should be inserted in P3.

i and j should be increased by 1.

i.e. i++, j++;

– If the power of the ith term of P1 is less than the power of the jth term of P2 then the ith term of P1 i.e. (P1 → a[i] ·power, P1 → a[i] · coeff) should be inserted in P3.

i should be increased by 1.

i.e. i++

Tech Knowledge Publications

– If the power of the ith term of P1 is greater than the power of the jth term of P2 then the jth term of P2 i.e. (P2→a[j]·power, P2→a[j]·coeff) should be inserted in P3.

j should be increased by 1.

i.e. j++

– If all terms of P1 have been exhausted i.e. i = = P1 → n

then the remaining terms of P2 are added to P3

– If all terms of P2 have been exhausted i.e. j = P2 → n then the remaining terms of P1 are added to P3.

'C' function to add two polynomials.

```
Polynomial add(Polynomial *P1, polynomial *P2)
{
    Polynomial P3;
    term t;
    int i, j;
    i = j = 0;
    init(&P3);
    while(i < P1 → n && j < P2 → n)
    {
        if(P1 → a[i]· power = = P2 → a[j] · power)
        {
            t·power = P1→a[i]·power;
            t·coeff = P1→a[i]·coeff+P2→a[j]· coeff;
            insert(&P3, t);
            i++; j++;
        }
        else
        if(P1 → a[i] · power < P2 → a[j] · power)
        {
            insert(&P3, P1 → a[i]);
            i++;
        }
        else
        {
            insert(&P3, P2 → a[j]);
            j++;
        }
    }
    while(i < P1 → n)
    {
        insert(&P3, P1 → a[i]);
            i++;
    }
    while(j < P2 → n)
    {
        insert(&P3, P2 → a[j]);
        j++;
    }
    return(P3);
}
```

3.12.6 Multiplication of Two Polynomials

Q. Write pseudo C/C++ code to perform polynomial multiplication using arrays.

SPPU - Dec. 16, 6 Marks

First polynomial	P1	$5 + 9x + 3x^4 + 2x^6$
Second polynomial	P2	$6 + 3x + 2x^2$

Step 1 :

Multiplying P1 with the first term of P2

$30 + 54x + 18x^4 + 12x^6$ (say Ptemp)

Ptemp is added to the final polynomial say (P3);

Value of P3 after step 1 = $30 + 54x + 18x^4 + 12x^6$

Step 2 :

Multiplying P1 with the second term of P2

$15x + 27x^2 + 9x^5 + 6x^7$ (say Ptemp)

Ptemp is added to the final polynomial P3

Value of P3 after step2

$= 30 + 69x + 27x^2 + 18x^4 + 9x^5 + 12x^6 + 6x^7$

Step 3 :

Multiplying P1 with the third term of P2.

$10x^2 + 18x^3 + 6x^6 + 4x^8$ (say Ptemp)

Ptemp is added to the final polynomial P3 value of P3 after step 3.

$= 30 + 69x + 37x^2 + 18x^3 + 18x^4 + 9x^5 + 18x^6 + 6x^7 + 4x^8$

Techknowledge Publication

'C' function for multiplication of two polynomials

```
Polynomial multiply(Polynomial *P1, Polynomial *P2)
{
    Polynomial P3, Ptemp;
    term t;
    int i;
    init(&P3)
    for(i = 0; i < P2 → n; i++)
    {
        init(&Ptemp)
        for(j = 0; j < P1 → n; j++)
        {
            t·power = P1→a[j]·power + P2→a[i]·power;
            t·coeff = P1→a[j]·coeff * P2→a[i]·coeff;
            insert(&Ptemp, t);
        }
        P3 = add(&P3, &Ptemp);
    }
    return(P3);
}
```

3.12.7 Evaluation of a Polynomials

It is often required to evaluate a polynomial for a particular value of x. Let P(x) be a polynomial where $P(x) = 2 + 5x^2 + 9x^3$. Value of a term $C_i\ x^i$ can be calculated as : C_i * pow(x, i). Value of each term can be calculated and added together using a 'FOR' loop.

'C' function to evaluate a polynomial.

```
float evaluate(Polynomial *P, float x)
{
    float value;
    int i;
    value = 0.00;
    for(i = 0; i< P → n; i++)
    value=value+(P→a[i]·coeff)*pow(x,P→a[i]·Power);
    return(value);
}
```

Program 3.12.1 : Various operations on polynomials.

```
#include<math.h>
#include<stdio.h>
#include<conio.h>
typedef struct term
{
    int power;
    float coeff;
}term;
typedef struct polynomial
{
    term a[20];
    int n;
}polynomial;
void init(polynomial *ptr);
void read(polynomial *ptr);
void print(polynomial *ptr);
polynomial add(polynomial *p1, polynomial *p2);
polynomial multiply(polynomial *p1, polynomial *p2);
float evaluate(polynomial *p1, float x);
void insert(polynomial *, term);
void main()
{
    polynomial P1, P2, P3;
    int option;
    float x, value;
    do
    {
        printf("\n 1 : create 1'st polynomial");
        printf("\n 2 : create 2'nd polynomial");
        printf("\n 3 : Add polynomials");
        printf("\n 4 : Multiply polynomials");
        printf("\n 5 : Evaluate 1'st polynomial");
        printf("\n 6 : Quit");
        printf("\n Enter your choice :");
        scanf("%d", &option);
        switch(option)
        {
            case 1:read(&P1); break;
            case 2:read(&P2); break;
```

Technknowledge Publication

```
            case 3:P3 = add(&P1, &P2);
                printf("\n 1'st polynomial ->\n ");
                print(&P1);
                printf("\n 2'nd polynomial ->\n ");
                print(&P2);
                printf("\n Sum = ");
                print(&P3);
            break;
            case 4:P3 = multiply(&P1, &P2);
                printf("\n 1'st polynomial ->\n ");
                print(&P1);
                printf("\n 2'nd polynomial ->\n ");
                print(&P2);
                printf("\n Product = ");
                print(&P3);
            break;
            case 5:printf("\n Enter the value of x:");
                scanf("%f", &x);
                value = evaluate(&P1, x);
                printf("\n Value of Polynomial = %6.2f", value);
            break;
        }
    }while(option! = 6);
}
void read(polynomial *Ptr)
{
    int n, i, power;
    float coeff;
    term t;
    init(Ptr);
    printf("\n Enter number of terms :");
    scanf("%d", &n);        /* read n terms */
    for(i = 0; i<n; i++)
    {   printf("\n enter a term(power  coeff.)");
        scanf("%d%f", &power, &coeff);
        t.power = power;
        t.coeff = coeff;
        insert(Ptr, t);
    }
}
void print(polynomial *Ptr)
{
    int i;
    printf("\n ");
    for(i = 0; i<Ptr->n; i++)
        printf("%5.2fX^    %d    \t",    (Ptr->a[i]).coeff,
        (Ptr->a[i]).power);
}
polynomial add(polynomial *P1, polynomial *P2)
{
    polynomial P3;
    term t;
    int i, j;
    i = j = 0;
    init(&P3);
    while(i<P1->n && j<P2->n)
    {
        if(P1->a[i].power = = P2->a[j].power)
        {
            t.power = P1->a[i].power;
            t.coeff = P1->a[i].coeff+P2->a[j].coeff;
            insert(&P3, t);
            i++; j++;
        }
        else
            if(P1->a[i].power < P2->a[j].power)
            {
                insert(&P3, P1->a[i]);
                i++;
            }
            else
            {
                insert(&P3, P2->a[j]);
                j++;
            }
    }
    while(i<P1->n)
    {
        insert(&P3, P1->a[i]);
```

Techknowledge Publications

```
        i++;
    }
    while(j<P2->n)
    {
        insert(&P3, P2->a[j]);
        j++;
    }
    return(P3);
}
polynomial multiply(polynomial *P1, polynomial *P2)
{
    polynomial P3, Ptemp;
    term t;
    int i, j;
    init(&P3);
    for(i = 0; i<P2->n; i++)
    {
        init(&Ptemp);
        for(j = 0; j<P1->n; j++)
        {
            t.power=P1->a[j].power+P2->a[i].power;
            t.coeff = P1->a[j].coeff*P2->a[i].coeff;
             insert(&Ptemp, t);
        }
    P3 = add(&P3, &Ptemp);
    }
    return(P3);
}
float evaluate(polynomial *P, float x)
{
    float value;
    int i, power;
    value = 0.00;
    for(i = 0; i<P->n; i++)
     {
        power = P->a[i].power;
        value = value+(P->a[i].coeff)*pow(x, power);
     }
    return(value);
}
```

```
void init(polynomial *Ptr)
{
    Ptr->n = 0;
}
void insert(polynomial *Ptr, term t)
{
    int i;
    /* Move all higher power terms by 1 place.
       Movement should start with the last term */
    for(i = Ptr -> n–1; (Ptr->a[i]).power > t.power
    && i > = 0; i– –)
    Ptr->a[i+1] = Ptr->a[i];
    /* insert the term t */
    Ptr->a[i+1] = t;
    (Ptr->n)++;
}
```

Output

```
1 : create 1'st polynomial
2 : create 2'nd polynomial
3 : Add polynomials
4 : Multiply polynomials
5 : Evaluate 1'st polynomial
6 : Quit
Enter your choice :1
Enter number of terms :3
enter a term(power  coeff.)2 3
enter a term(power  coeff.)4 6
enter a term(power  coeff.)5 7
1 : create 1'st polynomial
2 : create 2'nd polynomial
3 : Add polynomials
4 : Multiply polynomials
5 : Evaluate 1'st polynomial
6 : Quit
Enter your choice :2
Enter number of terms :2
enter a term(power  coeff.)2 3
enter a term(power  coeff.)6 8
1 : create 1'st polynomial
```

Techknowledge Publication

3.14 Operations on a Sparse Matrix

(1) From a 2D-representation (conventional) to sparse representation (stored as triples).

(2) Sparse representation (stored as triples) to a 2D-representation (conventional).

(3) Transpose (simple transpose, Fast Transpose)

(4) Addition

(5) Multiplication.

(6) Reading a sparse matrix as a list of triples.

(7) Displaying a sparse matrix as a list of triples.

(8) Inserting an element (row, column, values) in a list of triples.

3.14.1 From 2D-Representation to Sparse Representation

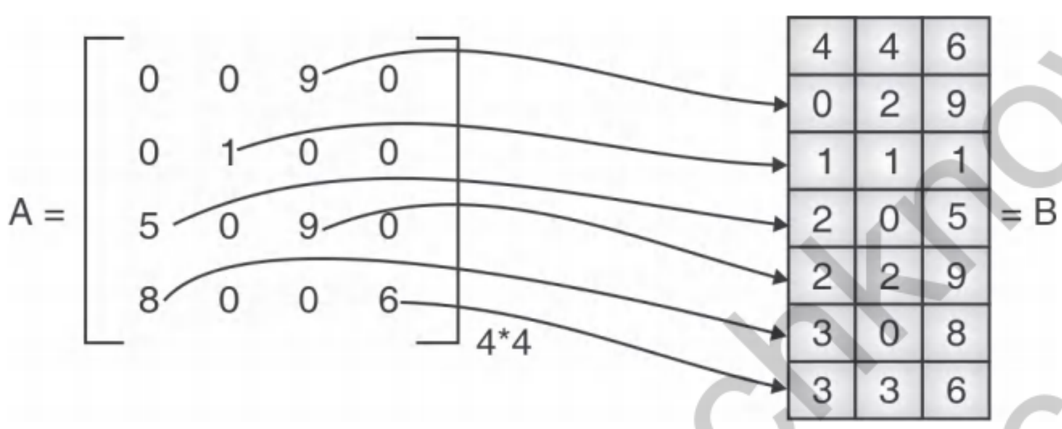


Fig. 3.14.1

– Matrix A is scanned from left to right row wise.

– Whenever a non-zero element is found, it is stored in the array B.

i, j → index for array A

k → index for array B (k = 1, initially)

– Loop for converting a 2D-matrix to its equivalent sparse representation.

```
for(i = 0; i < m; i ++)
    for(j = 0; j < n; j ++)
        if(a[i][j] ! = 0)
        {
            b[K][0] = i;
            b[K][1] = j;
            b[K][2] = a[i][j]
            K = K +1;
        }
```

'C' function for 2D-representation (conventional) to sparse representation (list of 3-triples).

```
void Two_Sparse(int a[10][10], int m, int n, int
b[MAX][3])
{
    // m : Actual number of rows
    // n : Actual number of columns
    int i, j, k;
    k = 1;
    b[0][0] = m;
    b[0][1] = n;
    for(i = 0; i<m; i++)
        for(j = 0; j<n; j++)
        {
            if(a[i][j]! = 0)
            {
                b[k][0] = i;
                b[k][1] = j;
                b[k][2] = a[i][j];
                k++;
            }        // number of non-zero elements
        }
    b[0][2] = k–1;
}
```

3.14.2 From Sparse Representation (List of Triplets) to 2D-representation (Conventional)

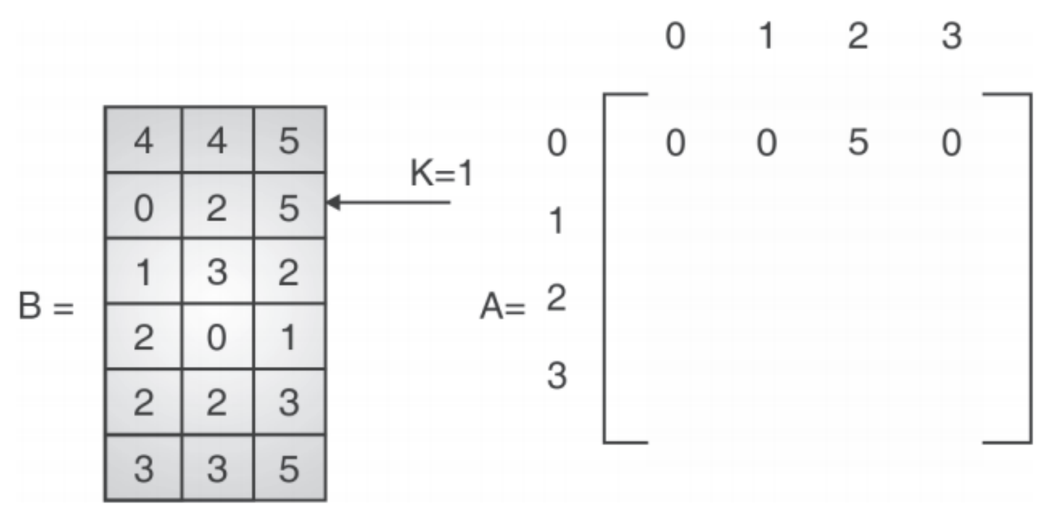


Fig. 3.14.2

First non-zero elements appears at A[0][2] as it is obvious from B[k] with k = 1.

TechKnowledge Publications

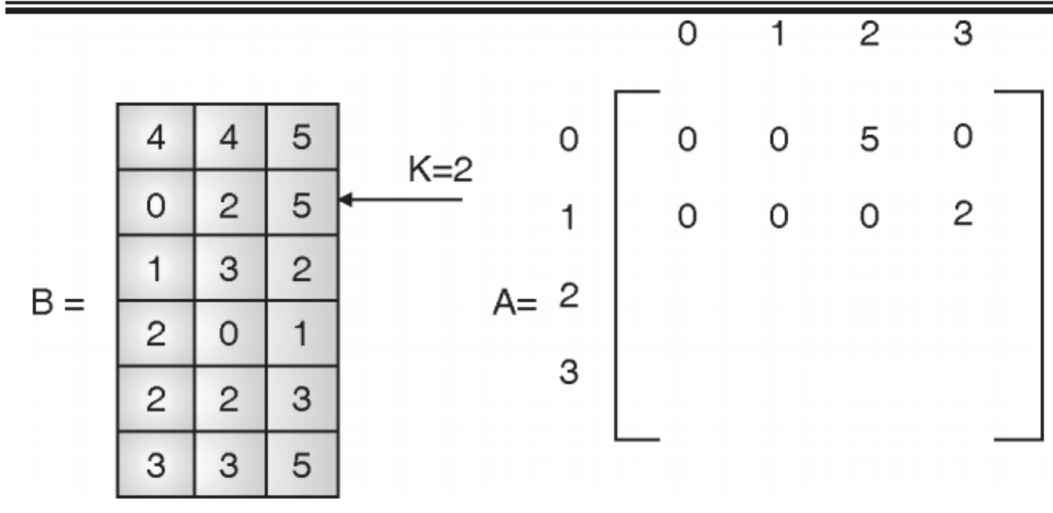


Fig. 3.14.3

Second non-zero element appears at A[1][3] and all elements between A[0][2] and A[1][3] are made zero. Working on the same line, we can get the final 2D matrix as

$$A = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 2 \\ 1 & 0 & 3 & 0 \\ 0 & 0 & 0 & 5 \end{bmatrix} \end{array}$$

Fig. 3.14.4

Algorithm

- k = 1; index for matrix "B" (a matrix of triples) (initially)
- i, j are used to scan the matrix A using the nested loop.

```
for(i = 0; i< m; i ++)
 for(j = 0; j < n; j ++)
```

- Whenever i = b[k][0] and j = b[k][1], a non-zero element must be stored at the location a[i][j]

  i.e. a[i][j] = b[k][2];

  otherwise,

  a[i][j] = 0;

- On finding a non-zero element, k is increased by 1.

Program segment for conversion

```
k = 1;
for(i = 0; i<b[0][0]; i++)
    for(j = 0; j<b[0][1]; j ++)
        {
        if(k > b[0][2] || i ! = b[k][0] || j ! = b[k][1])
            a[i][j] = 0;
        else
        {
            a[i][j] = b[k][2];
            k ++;
        }
    }
}
```

'C' function for sparse representation (list of 3-triples) to 2D representation (conventional).

```
void Sparse_TwoD(int b[MAX][3], int a[10][10])
{
    int i, j, k;
    k = 1;
    for(i = 0; i<b[0][0]; i++)
        for(j = 0; j<b[0][1]; j++)
        {
        if(k>b[0][2] || i! = b[k][0] || j! = b[k][1])
            a[i][j] = 0;
        else
        {
            a[i][j] = b[k][2];
            k++;
        }
    }
}
```

Ex. 3.14.1 : A sample program for reading a sparse matrix (2D form) and converting it to sparse form.

```
#include<stdio.h>
#include<conio.h>
#define MAX 20
void readmatrix(int a[10][10], int m, int n);
void printsparse(int b[MAX][3]);
void TwoD_sparse(int a[10][10], int m, int n, int b[MAX][3]);
void main()
{
    int a[10][10], b[MAX][3], m, n;
    printf("\n Enter the size of matrix(rows, columns)");
    scanf("%d%d", &m, &n);
    readmatrix(a, m, n);
    TwoD_sparse(a, m, n, b);
    printsparse(b);
    getch();
}
void readmatrix(int a[10][10], int m, int n)
```

Tech Knowledge Publications

```
{
    int i, j;
    printf("\n Enter elements of matrix :");
    for(i = 0; i<m; i++)
        for(j = 0; j<n; j++)
            scanf("%d", &a[i][j]);
}
void TwoD_sparse(int a[10][10], int m, int n, int b[MAX][3])
{
    int i, j, k;
    /*  i, j -> index for 2D matrix
       k  -> index for matrix in sparse form
    */
    k = 1;
    b[0][0] = m; b[0][1] = n;
    for(i = 0; i<m; i++)
        for(j = 0; j<n; j++)
            if(a[i][j]! = 0)
            {
                b[k][0] = i;
                b[k][1] = j;
                b[k][2] = a[i][j];
                k++;
            }
    b[0][2] = k–1;
}
void printsparse(int b[MAX][3])
{
    int i, n;
    n = b[0][2];   //Number of 3-triples
    printf("\n rows = %d \t columns = %d", b[0][0], b[0][1]);
    printf("\n ");
    for(i = 0; i< = n; i++)
    printf("%d\t%d\t%d\n ", b[i][0], b[i][1], b[i][2]);
}
```

Output

```
Enter the size of matrix(rows, columns)3 4
Enter elements of matrix :
0 0 5 0
5 0 3 0
0 1 0 2
rows = 3      columns = 4
3     4     5
0     2     5
1     0     5
1     2     3
2     1     1
2     3     2
```

Ex. 3.14.2 : A sample program for reading a sparse matrix in a sparse form and converting it into normal matrix.

```
#include<stdio.h>
#include<conio.h>
#define MAX 20
void printmatrix(int a[10][10], int m, int n);
void readsparse(int b[MAX][3]);
void sparse_twoD(int b[MAX][3], int a[10][10]);
void main()
{
    int a[10][10], b[MAX][3], m, n;
    printf("\n Enter the size of matrix(rows, columns)");
    scanf("%d%d", &m, &n);
    b[0][0] = m;
    b[0][1] = n;
    readsparse(b);
    sparse_twoD(b, a);
    printmatrix(a, m, n);
    getch();
}
void sparse_twoD(int b[MAX][3], int a[10][10])
{   int i, j, k;
    k = 1;
    for(i = 0; i<b[0][0]; i++)
        for(j = 0; j<b[0][1]; j++)
        if(k>b[0][2] || i! = b[k][0] || j! = b[k][1])
            a[i][j] = 0;
        else
        {
            a[i][j] = b[k][2];
            k++;
        }
}
```

Techknowledge Publications

```
void readsparse(int b[MAX][3])
{
    int i, t;
    printf("\n Enter no. of non-zero elements:");
    scanf("%d", &t);
    b[0][2] = t;
    for(i = 1; i< = t; i++)
    {
printf("\nEnter the next triple(row, column, value) :");
    scanf("%d%d%d", &b[i][0], &b[i][1], &b[i][2]);
    }
}
void printmatrix(int a[10][10], int m, int n)
{
    int i, j;
    for(i = 0; i<m; i++)
    {
        printf("\n ");
        for(j = 0; j<n; j++)
            printf("%d\t", a[i][j]);
    }
}
```

Output

```
Enter the size of matrix(rows, columns)3 4
Enter no. of non-zero elements:5
Enter the next triple(row, column, value) :0 2 5
Enter the next triple(row, column, value) :1 0 5
Enter the next triple(row, column, value) :1 2 3
Enter the next triple(row, column, value) :2 1 1
Enter the next triple(row, column, value) :2 3 2
0    0    5    0
5    0    3    0
0    1    0    2
```

3.14.3 Transpose of a Sparse Matrix (Simple/slow)

Transpose of a matrix is obtained by interchanging rows and columns.

In another way, we can say that element in the i, j position gets put in the j, i position.

Transpose of the matrix B1 is obtained as B2 by inserting $(i, j)^{th}$ element of B1 as $(j,i)^{th}$ element in B2.

B1 =

3	4	5
0	2	5
1	0	5
1	2	3
2	1	1
2	3	2

$B2(B_1^{'})$=

4	3	5
2	0	5
0	1	5
2	1	3
1	2	1
3	2	2

Fig. 3.14.5

Algorithm for simple/slow transpose

```
for(i = 1; i < = B1[0][2]; i ++)
{
    B2[i][0] = B1[i][1];
    B2[i][1] = B1[i][0];
    B2[i][2] = B1[i][2];
}

    B2[0][0] = B1[0][1];
    B2[0][1] = B1[0][0];
    B2[0][2] = B1[0][2];
```

- Elements of the matrix B2 are not in the required order.
- Elements should be ordered on row and inside row, they should be ordered on column.
- Thus the above method of finding transpose does not give the desired result.

  A modified algorithm for transpose (Produces an ordered list).
- Conversion of $(i,j)^{th}$ element of B1 to $(i,j)^{th}$ element of B2 must be done column wise.

An Algorithm without the above problem (Simple Transpose)

Step 1 : Copy elements of column 0 from B1 to B2

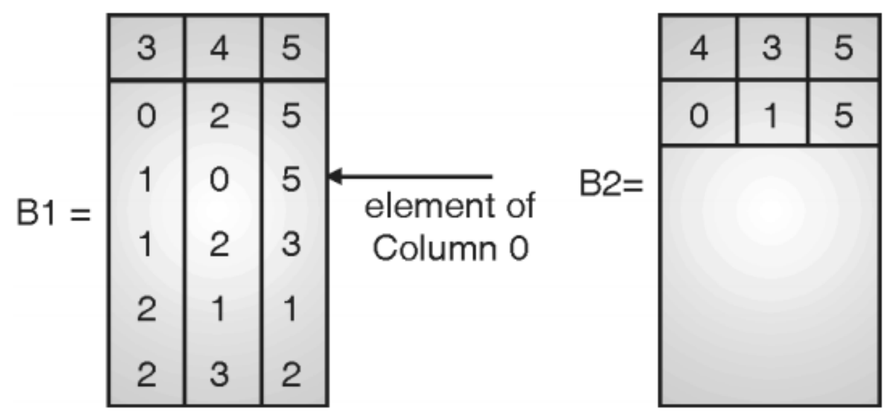


Fig. 3.14.6 : Element (1, 0, 5) is stored as (0, 1, 5)

TechKnowledge Publications

Copy elements of column 1 from B1 to B2.

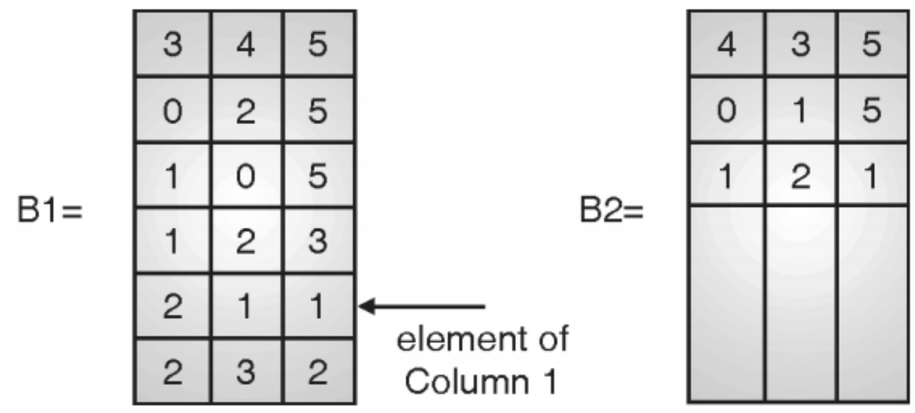


Fig. 3.14.7 : Element (2, 1, 1) is stored as (1, 2, 1)

Copy elements of column 2 from B1 to B2.

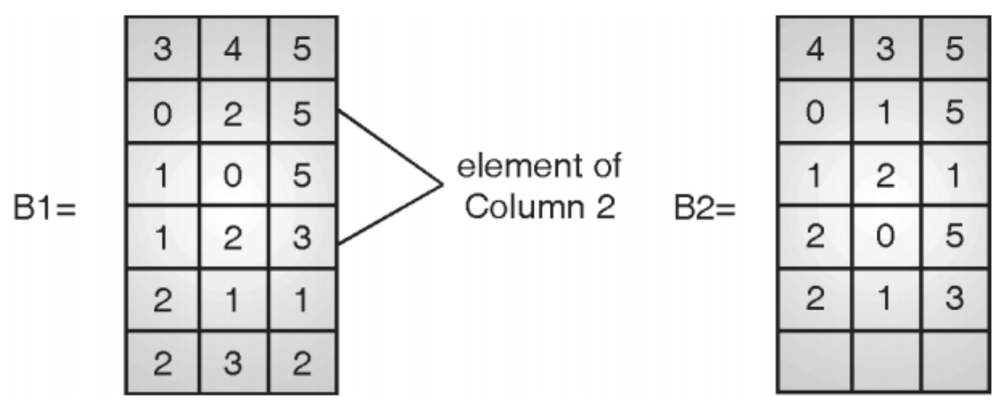


Fig. 3.14.8 : Element (0, 2, 5) is stored as (2, 0, 5)
Element (1, 2, 3) is stored as (2, 1, 3)

Copy elements of column 3 from B1 to B2.

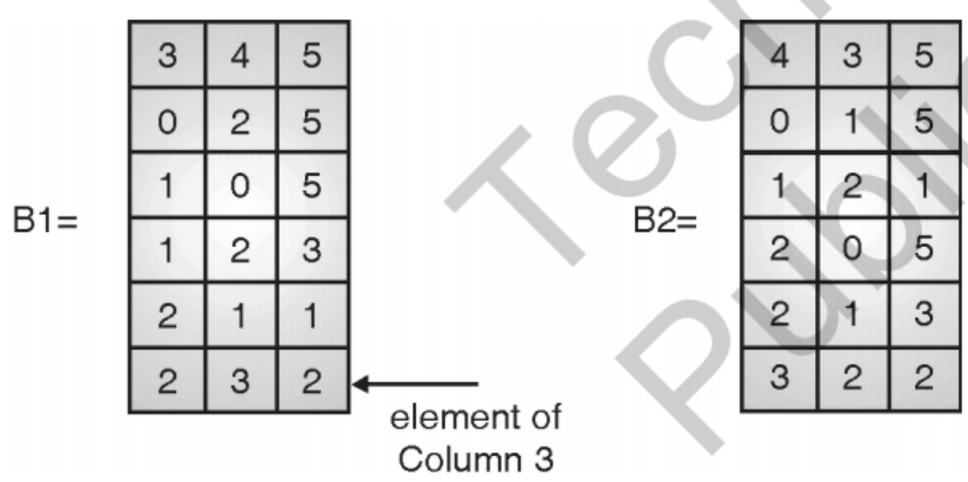


Fig. 3.14.9 : Element (2, 3, 2) is stored as (3, 2, 2)

– Algorithm works properly.

Algorithm

```
for all column j = 0 to j = B1[0][1]–1
{
    for all elements in column j
    {
            insert the element(i, j, val) of B1
            as(j, i, val)in B2.
    }
}
```

3.14.4 Pseudo-C Algorithm for Simple Transpose

Q. Write pseudo c/c++ to perform simple transpose of sparse matrix. **SPPU - Dec. 16, May 19, 4 Marks**

```
transpose(a, b)
{
    /* a is matrix represented in sparse form */
    /* transpose of a is stored in b */
    /* matrix a is of the size m×n with t non-zero terms
       */
    (m, n, t) = (a(0, 0), a(0, 1), a(0, 2));
    (b(0, 0), b(0, 1), b(0, 2)) = (n, m, t);
    j = 1; /* position of next term in b*/
    for(col = 0 to n – 1)    /* transpose by columns */
    {
    for(k = 1 to t)      /* consider all non-zero terms */
    {
        if(a(k, 1) is equal to col)
        {
            (b(j,0),b(j,1),b(j,2)=(a(k,1),a(k,0),a(k,2));
            j = j + 1;
        }
    }
    }
}
```

Program 3.14.1 : A sample program for finding transpose of sparse matrix.

```
#include<stdio.h>
#include<conio.h>
#define MAX 20
void printsparse(int[ ][3]);
void readsparse(int[ ][3]);
void transpose(int[ ][3], int[ ][3]);
void main()
{
    int b1[MAX][3], b2[MAX][3], m, n;
    printf("\n Enter the size of matrix(rows, columns)");
    scanf("%d%d", &m, &n);
    b1[0][0] = m;
```

Techknowledge Publication

```
    b1[0][1] = n;
    readsparse(b1);
    transpose(b1, b2);
    printsparse(b2);
    getch();
}
void readsparse(int b[MAX][3])
{
    int i, t;
    printf("\n Enter no. of non-zero elements:");
    scanf("%d", &t);
    b[0][2] = t;
    for(i = 1; i< = t; i++)
    {
        printf("\n Enter the next triple(row, column,
        value) :");
        scanf("%d%d%d",&b[i][0],&b[i][1],&b[i][2]);
    }
}
void printsparse(int b[MAX][3])
{
    int i, n;
    n = b[0][2];   //Number of 3-triples
    printf("\n rows=%d\tcolumns=%d",b[0][0],b[0][1]);
    printf("\n ");
    for(i = 0; i< = n; i++)
    printf("%d\t%d\t%d\n ", b[i][0], b[i][1], b[i][2]);
}
void transpose(int b1[ ][3], int b2[ ][3])
{
    int i, j, k, n;
    b2[0][0] = b1[0][1];
    b2[0][1] = b1[0][0];
    b2[0][2] = b1[0][2];
    k = 1;
    n = b1[0][2];
    for(i = 0; i<b1[0][1]; i++)
        for(j = 1; j< = n; j++)
    /* if a column number of current triple = = i
    then insert the current triple in b2 */
        if(i = = b1[j][1])
        {
            b2[k][0] = i;
            b2[k][1] = b1[j][0];
            b2[k][2] = b1[j][2];
            k++;
        }
}
```

Output

```
Enter the size of matrix(rows, columns)3 4
Enter no. of non-zero elements:5
Enter the next triple(row, column, value) :0 2 5
Enter the next triple(row, column, value) :1 0 5
Enter the next triple(row, column, value) :1 2 3
Enter the next triple(row, column, value) :2 1 1
Enter the next triple(row, column, value) :2 3 2
rows = 4      columns = 3
4     3     5
0     1     5
1     2     1
2     0     5
2     1     3
3     2     2
```

Program 3.14.2 : Write a program to represent a list of student's records and find topper among them.

```
# include <stdio.h>
# include <conio.h>
typedef struct student
{
    int rollno ;
    char name[20];
    int marks;
} student;
void main()
```

Techknowledge Publications

```
{
    student s[30], topper;
    int n, i;
    printf("\n Enter no. of students:");
    scanf("%d", &n);
    prnitf("\n Enter data:");
    for(i = 0, i < n; i++)
    scanf("%s %d %d", s[i].name, &s[i].rollno, &s[i].marks);
    //find topper
    topper = s[0];
    for(i = 1; i < n; i++)
        if(s[i].marks > topper.marks)
            topper = s[i];
            printf("\n %s \t %d \t %d", topper.name,
            topper.rollno, topper.marks);
}
```

3.14.5 Timing Complexity of the Function Transpose( )

Q. Discuss time complexity. **SPPU - Dec. 16, 2 Marks**

For calculation of timing complexity, it sufficient to calculate the frequency of the nested loop.

```
for (i = 0; i < b1[0][1]; i ++)
  for (j = 1; j < = b1[0][2]; j ++)
  {
  Statement of constant time order
}
```

$$\text{Frequency} = \sum_{i=0}^{n-1} \sum_{j=1}^{t} 1$$

Where, n = Number of columns

and t = Number of non-zero terms.

$$\therefore \quad \text{frequency} = \sum_{i=0}^{n-1} t = nt$$

Timing complexity = O (nt)

It is worse than O(mn) time using arrays perhaps, if t > m.

Space required to store the source and resultant matrices are of the order of number of non-zero tuples.

∴ Space complexity = O(t)

3.15 Fast Transpose

Q. Explain Fast transpose of sparse matrix with suitable example. **SPPU - May 17, 4 Marks**

Q. Explain the need for fast transpose of sparse matrix. **SPPU - Dec. 18, 1 Mark**

Fast Transpose of a Sparse Matrix

- It is another algorithm with much better timing behaviour.
- Number of elements in each column of "B1[ ]" is determined first. This gives number of elements in each row of "B2[ ]".
- From the above information, the starting place of each row in "B2[ ]" can be calculated exactly.

Example :

$$\left.\begin{array}{rcccc} \text{If number of elements in each column of B1} & \rightarrow 2 & 1 & 1 & 2 \\ \text{Column number} & \rightarrow 0 & 1 & 2 & 3 \end{array}\right\} \text{for matrix b1[ ]}$$

After transpose, number of elements in each column of B1[ ] will become number of elements in each row of B2. therefore,

Number of elements in each row of B2 →	2	1	1	2
row number →	0	1	2	3

Tech Knowledge Publications

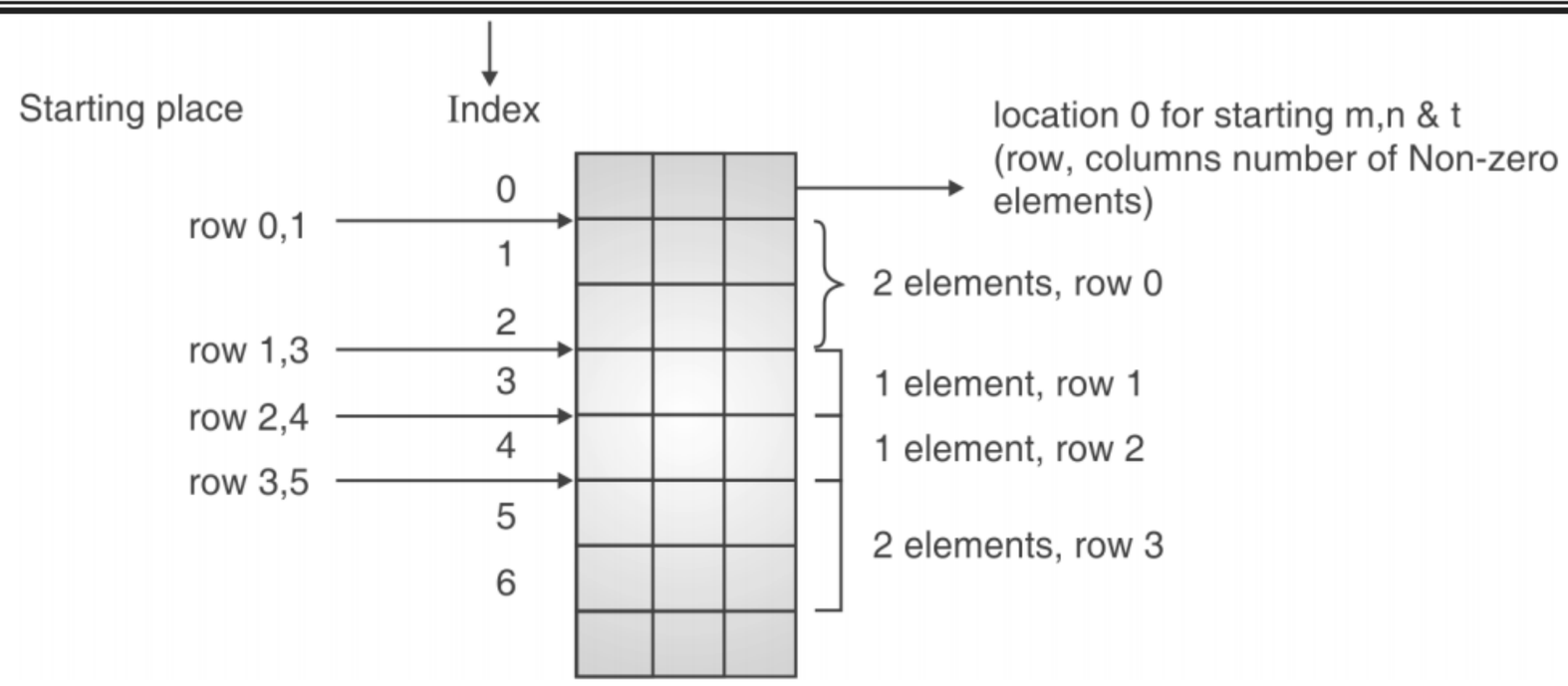


Fig. 3.15.1 : Showing starting place of each row in the transposed matrix "B2[ ]"

Row 0 starts at location 1

Row 1 starts at 1 + cumulative sum of previous elements = 1 + 2 = 3

row 2 starts at 1 + cumulative sum of previous elements = 1 + [2 +1] = 4

row 3 starts at 1 + cumulative sum of previous elements = 1 + [2 + 1 + 1] = 5

– Loop for counting of elements in each column of "B1[ ]"

```
int total[MAX], i, n, t, columnno;
t = b1[0][2];                    /* Number of non-zero elements */
for(i = 1; i < = t; i ++)
{
    columnno = b1[i][1];         /*column Number of ith triple */
    total[columnno] + +;
}
```

– Once, the total number of elements in each column of "B1[ ]" is calculated. Starting row index of each row in "B2[ ]" can be calculated.

```
int index[MAX];
index[0] = 1; /* 0th row starts from location */
for(i = 1; i < n; i ++)
index[i] = index[i – 1] + total[i – 1];
```

– Shown below is the calculation of total[ ] and index[ ] for the given sparse matrix.

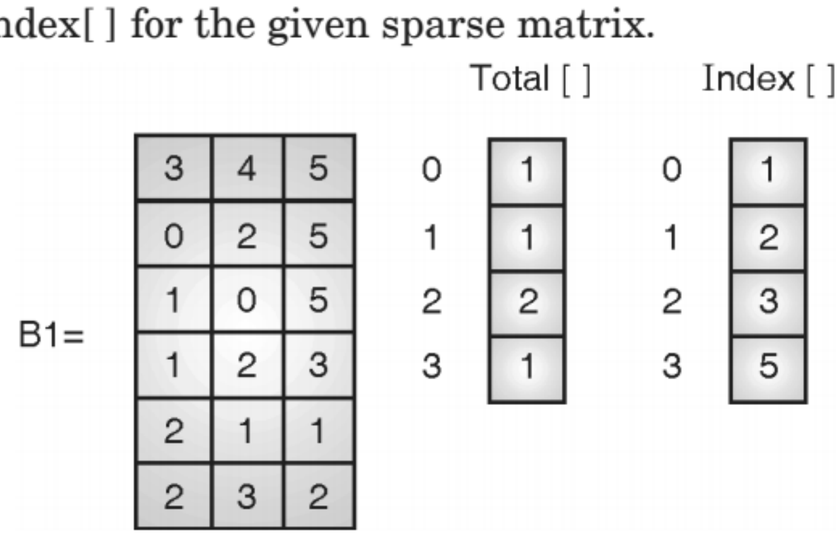


Fig. 3.15.2 : After calculation of total[ ] and index[ ]

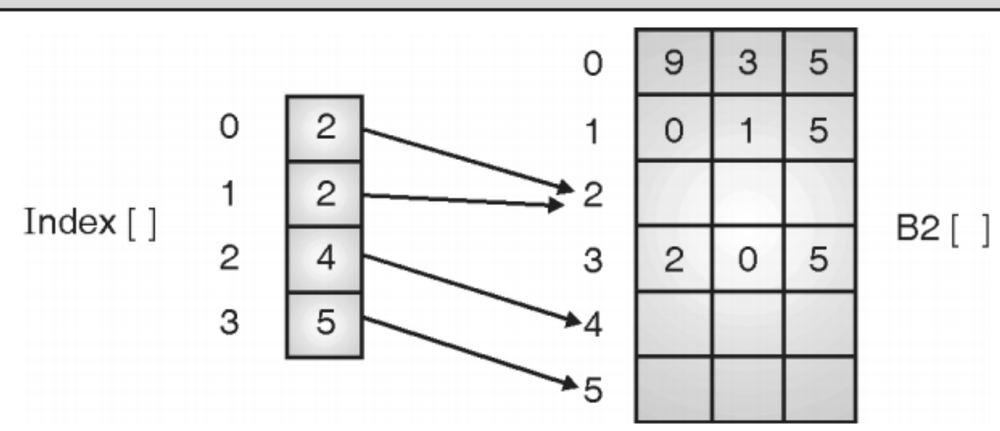


Fig. 3.15.3 : Status of "B2[ ]" and "index[ ]" after insertion of first two triples (0,2,5) and (1,0,5) in "B2[ ]"

– Triplet (0,2,5) of B1 is inserted as triplet (2,0,5) in B2.

– Triplet (2,0,5) is inserted at the location given by index[2] in "B2[ ]". After insertion of (2,0,5) in "B2[ ]", index[2] is increased by 1.

Technknowledge Publication

3.15.1 Pseudo-C Algorithm for Fast Transpose

Q. Write C/C++ pseudocode to perform the simple transpose of sparse matrices. Also, discuss its time complexity. **SPPU - Dec. 19, 6 Marks**

```
Fast_transpose(A, B)
{  /*  A is an array representing a sparse matrix of
       size m×n with t non-zero terms. The transpose
       is stored in B. count( ) and index( ) are used as
       pointers  */
   (m, n, t) = (A(0, 1), A(0, 2), A(0, 3));
   (B, (0, 1), B(0, 2), B(0, 3)) = (n, m, t);
                        /* store dimension of transpose */
   for(i = 0 to n – 1)
       count(i) = 0;              /* initialize to zero */
   for(i = 1 to t)
       count(A(i, 1)) ++;
   //count(k) is the number of elements in row k of B
   index(0) = 1;
   for(i = 1 to n – 1)
   index(i) = index(i – 1) + count(i – 1)
            /* index(i) gives position of row i in B */
                        /* move all element of A to B */
   for(i = 1 to t)
   {
       j = A(i, 1);
       (B(index(j), 0), B(index(j), 1), B(index(j), 2))
        = (A(i, 1), A(i, 0), A(i, 2));
       index(j) ++;
   }
}
```

'C' function for fast transpose.

```
void Fast_transpose(int B1[MAX][3], int B2[MAX][3])
{
    int m, n, t, i, col_num, location;
    int total[MAX], index[MAX];
    m = B1[0][0]; n = B1[0][1]; t = B1[0][2];
    B2[0][0] = n; B2[0][1] = m; B2[0][2] = t;
    for(i = 0; i<n; i++)
        total[i] = 0;
    for(i = 1; i< = t; i++)
    {   col_num = B1[i][1];
        total[col_num]++;
    }
    index[0] = 1;
    for(i = 1; i<n; i++)
        index[i] = index[i–1]+total[i–1];
    for(i = 1; i< = t; i++)
    {
        col_num = B1[i][1];
        location = index[col_num];
        index[col_num]++;
        B2[location][0] = B1[i][1];
        B2[location][1] = B1[i][0];
        B2[location][2] = B1[i][2];
    }
}
```

3.15.2 Calculation of Timing Complexity of "Fast-Transpose( )"

Q. Discuss time complexity of fast transpose. **SPPU - May 17, Dec. 18, 2 Marks**

Q. Write C/C++ pseudocode to perform the simple transpose of sparse matrices. Also, discuss its time complexity. **SPPU - Dec. 19, 6 Marks**

There are four loops in FAST-TRANSPOSE, which we executed n, t, n–1 and t times respectively. Hence the order of magnitude of timing complexity is O(n + t). The computing time of O(n + t) becomes O(nm) when t is of the order of nm. This is the same as when t is of the order of nm. When t is sufficiently small compared to its maximum of nm, "Fast-Transpose" will be faster. Finally, one should note that Fast_Transpose requires more space than simple transpose. There are two additional arrays, namely index[MAX] and total[MAX].

Space required for original and transposed matrix is of the order of t. Space required for index[ ] and total[ ] is of the order of n.

$$\therefore \text{ Space required } = O(t + 2\,n)$$

Technknowledge Publication

3.15.3 Addition of Two Sparse Matrices

Pseudo C code to Add Two Sparse Matrices

(1) Let B1[ ], B2[ ] are two sparse matrices to be added and stored in B3[ ].

(2) Let t1 = B1[0][2] and t2 = B2[0][2] be the number of non-zero elements in B1 and B2.

(3) i,j,k are three variables used as indexes of B1, B2 and B3 respectively.

```
i = j = k = 1
```

(4) Since, the final matrix B3 will have same number of rows and columns as either in B1 or B2.

```
B3[0][0] = B1[0][0]
B3[0][1] = B1[0][1]
```

(5)
```
while(i < = t1 and j < = t2)
{
if(B1[i][0] < B2[j][0])
{
        B3[k][0] = B1[i][0];
        // element from B1 should be added to B3
        B3[k][1] = B1[i][1];
        B3[k][2] = B1[i][2];
        k ++;
        i ++;
        goto last;
}
if(B2[j][0] < B1[i][0])
{       B3[k][0] = B2[j][0];
        B3[k][1] = B2[j][1];
        B3[k][2] = B2[j][2];
        j ++;
        k ++;
        goto last
}
/* if row numbers are same, they should be compared on column numbers */
if(B1[i][1] < B2[j][1])
{
        B3[k][0] = B1[i][0];
        B3[k][1] = B1[i][1];
        B3[k][2] = B1[i][2];
        i ++;
        k ++;
        goto last ;
}
if(B2[j][1] < B1[i][1])
{       B3[k][0] = B2[j][0];
        B3[k][1] = B2[j][1];
        B3[k][2] = B2[j][2];
        j ++;
        k ++;
        goto last;
}
/* otherwise, add the values of both the element */
        B3[k][0] = B1[i][0];
        B3[k][1] = B1[i][1];
        B3[k][2] = B1[i][2] + B2[j][2];
        i ++;
        j ++;
        k ++;
        last :
}
```

(6)
```
while(i < = t1)
{       /* add remaining triples of B1 */
        B3[k][0] = B1[i][0];
        B3[k][1] = B1[i][1];
        B3[k][2] = B1[i][2];
```

Technknowledge Publications

```
            k ++;
            i ++;
    }
(7) while(j < = t2)     /* add remaining triples of B2 */
    {       B3[k][0] = B2[j][0];
            B3[k][1] = B2[j][1];
            B3[k][2] = B2[j][2];
            j ++;
            k ++;
    }
(8) B3[0][2] = k–1;
```

Program 3.15.1 : Program for addition of two sparse matrices.

```
#include<stdio.h>
#include<conio.h>
#define MAX 20
void printsparse(int b[MAX][3]);
void readsparse(int b[MAX][3]);
void addsparse(int b1[MAX][3], int b2[MAX][3], int
b3[MAX][3]);
void main()
{
    int b1[MAX][3], b2[MAX][3], b3[MAX][3];
    readsparse(b1);
    readsparse(b2);
    addsparse(b1, b2, b3);
    printsparse(b3);
    getch();
}
void readsparse(int b[MAX][3])
{
    int i, t, m, n;
    printf("\n Enter no. of rows & columns:");
    scanf("%d%d", &m, &n);
    printf("\n No. of non-zero triples :");
    scanf("%d", &t);
    b[0][0] = m;
    b[0][1] = n;
    b[0][2] = t;
    for(i = 1; i< = t; i++)
    {
    printf("\n Enter the triple(row, column, value) :");
    scanf("%d%d%d", &b[i][0], &b[i][1], &b[i][2]);
    }
}
void addsparse(int b1[MAX][3], int b2[MAX][3], int
b3[MAX][3])
{
    int t1, t2, i, j, k;
    t1 = b1[0][2];
    t2 = b2[0][2];
    i = j = k = 0;
    b3[0][0] = b1[0][0];
    b3[0][1] = b1[0][1];
    while(i< = t1 && j< = t2)
    {
        if(b1[i][0] < b2[j][0])
        {   //row numbers are not equal
            b3[k][0] = b1[i][0];
            b3[k][1] = b1[i][1];
            b3[k][2] = b1[i][2];
            k++;
            i++;
            goto last;
        }
        if(b2[j][0] < b1[i][0])
        {   //row numbers are not equal
            b3[k][0] = b2[j][0];
            b3[k][1] = b2[j][1];
            b3[k][2] = b2[j][2];
            k++;
            j++;
            goto last;
        }
        if(b1[i][1] < b2[j][1])
        {   //row numbers are equal, compare column
            b3[k][0] = b1[i][0];
            b3[k][1] = b1[i][1];
```

Techknowledge Publications

```
            b3[k][2] = b1[i][2];
            k++;
            i++;
            goto last;
        }
        if(b2[j][1] < b1[i][1])
        {   //row numbers are equal, compare column
            b3[k][0] = b2[j][0];
            b3[k][1] = b2[j][1];
            b3[k][2] = b2[j][2];
            k++;
            j++;
            goto last;
        }
        b3[k][0] = b1[i][0];
        //row and column numbers are equal
        b3[k][1] = b1[i][1];
        b3[k][2] = b1[i][2]+b2[j][2];
        k++;
        i++;
        j++;
        last:
    }
    while(i< = t1)
    {   //copy remaining terms from b1
        b3[k][0] = b1[i][0];
        b3[k][1] = b1[i][1];
        b3[k][2] = b1[i][2];
        i++;
        k++;
    }
    while(j< = t2)
    {   //copy remaining terms from b2
        b3[k][0] = b2[j][0];
        b3[k][1] = b1[j][1];
        b3[k][2] = b1[j][2];
        j++;
        k++;
    }
    b3[0][2] = k–1;        //Set number of terms in b3
}
```

```
void printsparse(int b[MAX][3])
{
    int i, t;
    t = b[0][2];
    for(i = 1; i< = t; i++)
    {
    printf("\n %d\t%d\t%d", b[i][0], b[i][1], b[i][2]);
    }
}
```

Output

```
Enter no. of rows & columns:3 4
No. of non-zero triples :5
Enter the triple(row, column, value) : 0 2 5
Enter the triple(row, column, value) : 1 0 5
Enter the triple(row, column, value) : 1 2 3
Enter the triple(row, column, value) : 2 1 1
Enter the triple(row, column, value) : 2 3 2
No. of non-zero triples :5
Enter the triple(row, column, value) : 0 2 1
Enter the triple(row, column, value) : 1 0 1
Enter the triple(row, column, value) : 1 2 1
Enter the triple(row, column, value) : 2 1 1
Enter the triple(row, column, value) : 2 3 1
0     2     6
1     0     6
1     2     4
2     1     2
2     3     3
```

Time complexity of addition of Sparse Matrices

If t_1 and t_2 are number of non-zero elements in first and second matrices respectively, then the time to add = $O(t_1 + t_2)$.

3.15.4 Multiplication of Two Sparse Matrices

'C' function for multiplication of two matrices

```
void multiply(int b1[MAX][3], int b2[MAX][3], int b3[MAX][3])
{
    int b22[MAX][3];
```

Techknowledge Publications

```
int i, j, k, B1rno, B2colno, i1, sum, t1, t2;
t1 = b1[0][2]; t2 = b2[0][2];
if(b1[0][1]! = b2[0][0])
{
    printf("\n can not multiply");
    exit(0);
}
Fast_transpose(b2, b22);
/*  b1 in row major form and b2 in column major
    form, as for multiplication, dot product of a row
    of b1 with column of b2 is taken  */
k = 1;      //index for b3
i = 1;
while(i< = t1)
{
    i1 = i;
    //starting location of the current row in b1
    j = 1;
    while(j< = t2)
    { /* current row of b1 must be multiplied with
        every column of b2 */
        B1rno = b1[i][0];
        B2colno = b22[j][0];
        sum = 0;
    while(i< = t1&& j< = t2 && B1rno = =
    b1[i][0] && B2colno = = b22[j][0])
    {   /*  Multiply the current row of b1 with
            next column of b2 */
        if(b1[i][1] = = b22[j][1])
        {
            sum = sum+b1[i][2]*b22[j][2];
            i++; j++;
        }
        else
            if(b1[i][1]<b22[j][1])
                i++;
            else
                j++;
    }
    if(sum! = 0)
    {
        b3[k][0] = B1rno;
        b3[k][1] = B2colno;
        b3[k][2] = sum;
        k++;
    }
    if(j< = t2)
        i = i1;
        while(B2colno == b22[j][0] && j< = t2)
        /*  Skip remaining terms of the current
            column and goto the next column. */
            j++;
    }
    while(B1rno = = b1[i][0] && i< = t1)
    /* Skip remaining terms of the current row and
    goto the next row. */
    i++;
}
b3[0][0] = b1[0][0];
b3[0][1] = b22[0][0];
b3[0][2] = k–1;
}
```

Program 3.15.2 : Program for multiplication of two sparse matrices.

```
#include<conio.h>
#include<stdio.h>
#define MAX 20

void print_sparse(int b[MAX][3]);
void read_sparse(int b[MAX][3]);
void multiply(int b1[MAX][3], int b2[MAX][3], int
b3[MAX][3]);
void Fast_transpose(int B1[MAX][3], int
B2[MAX][3]);
void main()
{
    int b1[MAX][3], b2[MAX][3], b3[MAX][3];
    read_sparse(b1);
    read_sparse(b2);
    multiply(b1, b2, b3);
    print_sparse(b3);
```

Techknowledge Publications

```
    getch();
}
void multiply(int b1[MAX][3], int b2[MAX][3], int
b3[MAX][3])
{
    int b22[MAX][3];
    int i, j, k, B1rno, B2colno, i1, sum, t1, t2;
    t1 = b1[0][2];
    t2 = b2[0][2];
    if(b1[0][1]! = b2[0][0])
    {
        printf("\n can not multiply");
        exit(0);
    }
    Fast_transpose(b2, b22);
    /*  b1 in row major form and b2 in column major
        form , as for multiplication, dot product of a row
        of b1 with column of b2 is taken  */
    k = 1;      //index for b3
    i = 1;
    while(i< = t1)
    {
        i1 = i;
        j = 1;
        while(j< = t2)
        {
            B1rno = b1[i][0];
            B2colno = b22[j][0];
            sum = 0;
        while(i< = t1&& j< = t2 && B1rno = =
        b1[i][0] && B2colno = = b22[j][0])
        {
            if(b1[i][1] = = b22[j][1])
            {
                sum = sum+b1[i][2]*b22[j][2];
                i++; j++;
            }
            else
                if(b1[i][1]<b22[j][1])
                    i++;
                else
                    j++;
            }
            if(sum! = 0)
            {   b3[k][0] = B1rno;
                b3[k][1] = B2colno;
                b3[k][2] = sum;
                k++;
            }
            if(j< = t2)
                i = i1;
            while(B2colno = = b22[j][0] && j< = t2)
                j++;
        }
        while(B1rno = = b1[i][0] && i< = t1)
            i++;
        }
        b3[0][0] = b1[0][0];
        b3[0][1] = b22[0][0];
        b3[0][2] = k–1;
}
void print_sparse(int b[MAX][3])
{
    int i, n;
    n = b[0][2];   //number of 3-triples
    printf("\n rows = %d \t columns = %d", b[0][0],
    b[0][1]);
    printf("\n ");
    for(i = 0; i< = n; i++)
    printf("%d\t%d\t%d\n ", b[i][0], b[i][1], b[i][2]);
}
void read_sparse(int b[MAX][3])
{
    int i, t;
    printf("\n Enter size of the matrix:");
    scanf("%d%d", &b[0][0], &b[0][1]);
    printf("\n Enter no. of non-zero elements:");
    scanf("%d", &t);

    b[0][2] = t;
    for(i = 1; i< = t; i++)
```

Techknowledge Publication

```
    {
        printf("\n Enter the next triple(row, column,
        value) :");
        scanf("%d%d%d",&b[i][0],&b[i][1], &b[i][2]);
    }
}

void Fast_transpose(int B1[MAX][3], int B2[MAX][3])
{
    int m, n, t, i, col_num, location;
    int total[MAX], index[MAX];
    m = B1[0][0]; n = B1[0][1]; t = B1[0][2];
    B2[0][0] = n; B2[0][1] = m; B2[0][2] = t;
    for(i = 0; i<n; i++)
        total[i] = 0;
    for(i = 1; i< = t; i++)
    {
        col_num = B1[i][1];
        total[col_num]++;
    }
    index[0] = 1;
    for(i = 1; i<n; i++)
        index[i] = index[i–1]+total[i–1];
    for(i = 1; i< = t; i++)
    {
        col_num = B1[i][1];
        location = index[col_num];
        index[col_num]++;
        B2[location][0] = B1[i][1];
        B2[location][1] = B1[i][0];
        B2[location][2] = B1[i][2];
    }
}
```

Output

```
Enter size of the matrix:4 4
Enter no. of non-zero elements:6
Enter the next triple(row, column, value) :0 2 9
Enter the next triple(row, column, value) :1 1 1
Enter the next triple(row, column, value) :2 0 5
Enter the next triple(row, column, value) :2 2 9
Enter the next triple(row, column, value) :3 0 8
Enter the next triple(row, column, value) :3 3 6
Enter size of the matrix:4 4
Enter no. of non-zero elements:6
Enter the next triple(row, column, value) :0 2 9
Enter the next triple(row, column, value) :1 1 1
Enter the next triple(row, column, value) :2 0 5
Enter the next triple(row, column, value) :2 2 9
Enter the next triple(row, column, value) :3 0 8
Enter the next triple(row, column, value) :3 3 6
rows = 4     columns = 4
4    4    8
0    0    45
0    2    81
1    1    1
2    0    45
2    2    126
3    0    48
3    2    72
3    3    36
```

3.15.5 Applications of Sparse Matrices

- Sparse matrix is a matrix that allows special technique to take advantage of the large number of zero elements. Sparse matrices support the following operations on a matrix :
  1. Representation
  2. Transpose
  3. Multiplication.
- A number of simulataneous equations can be represented using a sparse matrix.
- A graph can be represented using a sparse matrix.
- A circuit can be represented using a sparse matrix.
- A network can be represented using a sparse matrix.

Technknowledge Publication

3.16 Representing a Polynomial of Two Variables using Sparse Matrix

Polynomial of two variables can be represented using an ordered list of 3-tuples of the form :

(Power of x, power of y, coefficient)

When these tuples are stored in the list, power of x is increasing. For any power of x, power of y is increasing.

Declaration of the matrix (Polynomial stored as triples)

int poly[MAX] [3]

↑

Maximum of number of rows.

The element poly[0][2] contains number of non-zero terms.

Ex. 3.16.1 : Represent the polynomial $5 + 10x + 5xy + 9x^2y^2 + 2x^3y$. Above polynomial can be represented in the triple form as :

((0,0,5), (1,0,10) (1,1,5) (2,2,9) (3,1,2)) these tuples can be stored in a matrix as given below : Poly[ ][ ]

	–	–	5	← Number of non-zero terms
0th term →	0	0	5	
1st term →	1	0	10	
2nd term →	1	1	5	
3rd term →	2	2	9	
4th term →	3	1	2	

↑ Power of x ↑ Power of y ↑ Coefficient

C-pseudo algorithm for addition of polynomials

```
void padd(poly1, poly2, poly3)
{/* poly1 has t1 terms, poly2 has t2 terms resultant
   polynomial will be stored in poly3
   i = j = k = 1
    i for scanning of polynomial 1, j for polynomial 2
   and k for polynomial 3 */
   t1 = poly1[0][2];
   t2 = poly2[0] [2]
   while(i < = t1 and j< = t2)
   {// terms in both poly1 and poly2
      if(poly[i][0] > poly2[j][0])
      {   // insert jth term of poly2 in poly3
          kth term of poly3 = jth terms of poly2;
          k++;
          j++;
          continue;
      }
      if(poly[i][0] < poly2[j][0])
      {   // insert ith term of poly1 in poly3
          kth term of poly3 = ith term of poly1;
          k++;
          i++;
          continue;
      }
      if(poly1[i][1] < poly2[j][1])
      {   // insert ith term of poly1 in poly3
          kth term of poly3 = ith term of poly1;
          k++;
          i++;
          continue;
      }
      // both terms with identical power
      kth term of poly3 = ith term of poly1+jth terms
      of poly2;
      i++;
      j++;
      k++;
   }
   // insert remaining terms of poly1 in poly3
   while(i< = t1)
   {
      kth term of poly3 = ith term of poly1;
      i++; k++;
   }
   // insert remaining terms of poly2 in poly3
   while(j < = t2)
   {
      kth terms of poly3 = jth term of poly2;
          j++; k++
```

Tech Knowledge Publications

```
    }
    poly3[0][2] = k – 1;  // Number of terms in poly3
}
```

3.17 Case Study - Use of Sparse Matrix in Social Networks and Maps

In a sparse network, we have less number of links compare to maximum possible number of links. Sparse matrix is suitable for representation of social networks and maps most of the real networks are sparse.

In online social networks, edges represent interaction between people.

In a road network, nodes represent interactions and edges roads connecting the interactions.

The adjacency matrix of social networks and maps is very sparse. Such an adjacency matrix can be efficiently represented using a sparse matrix.

3.17.1 Use of Polynomials to Model Economic Growth Pattern

Stastics are crucial in determining the economic growth of the country in knowing the performance and effectivenss of economy. For example monetary policy as the name implies is one of the major economic stabilization weapons involving measures designated to regulate and control the volume, cost, availability and direction of money and credit in the economy to achieve some specific economic policy objective.

We can find a model in a form of linear regression in which the relationship between the independent variable X and dependent variable Y can be represented as an nth order polynomial.

$$Y_i = \beta_0 + \beta_1 X_i + \beta_2 X_i^2 + \in$$

3.17.2 Polynomial to Describe the Behaviour of Covid-19 Virus

The 2019 novel coronavirus is also known COVID-19. The fatality rate for this virus has been estimated to be 4.5 % to 14.8 % depending upon the age group.

Elderly people with underlying diseases like diabetes and hypertension are considered at the highest risk. Symptoms for this disease can take 2-14 days to appear and can range from fever, cough, shortness of breath to pneumonia, kidney failure and even death. A well tested vaccine is not available and hence a key part in managing the pandemic is to decrease the epidemic peak, also known as flattening the epidemic curve. We can use a polynomial to create a mathematical model to describe the behavior of COVID-19 virus.

1. Finding the rate of spread of the disease.
2. Developing a polynomial model to evaluate the spread of disease.
3. Prediction of COVID-19 outbreak using the model.

The polynomial regression can be used to create a model. A polynomial regression with degree 1 is a linear regression. Choosing the value of a degree is a challenging task.

Review Questions

Q. 1 Explain sequential memory organization using suitable data structure. **(6 Marks)**

Q. 2 Derive formula to calculate address of any location in two-dimensional arrays. **(4 Marks)**

Q. 3 Explain the two dimensional arrays in details with column and row major implementation and address calculation in both the cases. **(6 Marks)**

Q. 4 Write an algorithm for multiplication of sparse matrix. Comment on time and space complexity for both. **(10 Marks)**

Techknowledge Publication

Q. 5 Write pseudo C algorithm to reverse a string. **(3 Marks)**

Q. 6 Write an algorithm to add two sorted polynomials in single variable. Analyze its time complexity. **(7 Marks)**

Q. 7 Write address calculation for elements of one dimensional array. **(2 Marks)**

Q. 8 Represent the sparse matrix using suitable data structure and write a pseudo C code to find transpose of a spare matrix using slow transpose. **(7 Marks)**

Q. 9 Write a pseudo C algorithm for addition of two sparse matrices. Analyze its time complexity. **(7 Marks)**

Q. 10 Define polynomial. **(2 Marks)**

Q. 11 Explain simple and fast transpose of a sparse matrix with example. **(6 Marks)**

Q. 12 What is sparse matrix ? What are its applications ? **(5 Marks)**

Q. 13 Write pseudo C/C++ algorithm to find transpose of sparse matrix using fast transpose algorithm. **(4 Marks)**

❑❑❑

Techknowledge Publication

Model Question Paper (In Sem.)

Fundamentals of Data Structures (210242)

Semester III - Computer Engineering (Savitribai Phule Pune University)

Time : 1 Hour **Maximum Marks : 30**

Strictly as per the New Credit System Syllabus (2019 Course)
Savitribai Phule Pune University w.e.f. academic year 2020-2021

Instructions to the candidates :

1) ***Answer Q.1 or Q.2, Q.3 or Q.4, Q.5 or Q.6, Q.7 or Q.8, Q.9 or Q.10***
2) ***Neat diagrams must be drawn wherever necessary.***
3) ***Figure to the right indicate full marks.***
4) ***Make suitable assumptions, if necessary.***

Technical Knowledge Publication

Q. 1 Define and explain the following terms : **(5 Marks)**

(a) Data structure

(b) Complexity of an algorithm

(c) ADT

(d) Linear and non-Linear Data Structure

(e) Asymptotic notation

OR

Q. 2 **(a)** Explain the Asymptotic notation. Plot the graph for Big O, Omega and Theta notation with respect to input vs time. **(3 Marks)**

(b) Write a short note on persistent and ephemeral data structures. **(2 Marks)**

Q. 3 Write a C/C++ program to count the frequency of each data in an array. **(5 Marks)**

OR

Q. 4 Write a C/C++ program to check whether a given integer number is a palindrome. **(5 Marks)**

Q. 5 What is divide and conquer strategy ? Explain and give an example. **(5 Marks)**

OR

Q. 6 What is greedy strategy ? Explain and give all example. **(5 Marks)**

Q. 7 Explain relationship among data, data structure and algorithm. **(5 Marks)**

OR

Q. 8 Calculate timing complexity of the given raw segment.

```
int A(int n)
{
    if(n < = 1)
        return(1) ;
    else
        return(A(n – 1) + A (n – 1)) ;
}
```

(5 Marks)

Q. 9 **(a)** Write C/C++ pseudocode to perform the simple transpose of sparse matrices. Also, discuss its time complexity? **(5 Marks)**

(b) Each element of an array Data [20] [50] requires 4 bytes of storage. Base address of data is 2000. Determine the location of Data [10] [10] when the array is stored as : (i) Row major (ii) Column major **(5 Marks)**

(c) Explain the algorithm for addition of two polynomials. **(5 Marks)**

OR

Q. 10 **(a)** Write C/C++ function for multiplication of two polynomials. Also, discuss its time complexity. **(5 Marks)**

(b) Explain the algorithm for fast transpose of a sparse matrix. Also, discuss its time complexity. **(5 Marks)**

(c) Represent the following polynomials using array :

(i) $x^3 + 5x^2 + x + 16$ (ii) $x^4 + 59x + 10$ **(5 Marks)**

❑❑❑

Technical Knowledge Publication

CHAPTER 4

Unit III

Searching

Syllabus :

Searching : Search Techniques-Sequential Search/Linear Search, Variant of Sequential Search- Sentinel Search, Binary Search, Fibonacci Search, and Indexed Sequential Search.

Exemplar/Case Studies : Use of Fibonacci search in non-uniform access memory storage and in Optimization of Unimodal Functions.

4.1 Searching

Searching is a technique of finding an element in a given list of elements. List of elements could be represented as follows :

(a) Array (b) Linked List

(c) Binary tree (d) B-tree

(e) Heap

Elements could also be stored in file. Searching technique should be able to locate the element to be searched as quickly as possible. Many a time, it is necessary to search a list of records to identify a particular record. Usually, each record is uniquely identified by its key field and searching is carried out on the basis of key field. If search results in locating the desired record then the search is said to be successful. Otherwise, the search operation is said to be unsuccessful. The complexity of any searching algorithm depends on number of comparisons required to find the element. Performance of searching algorithm can be found by counting the number of comparisons in order to find the given element.

Some of the standard searching techniques, includes :

1. Sequential search
2. Binary search
3. Fibonacci search

4.2 Sequential - Linear Search

Q. Explain sequential search with appropriate example and compare their time complexity and space complexity. **SPPU - Dec. 17, May 18, 4 Marks**

In sequential search elements are examined sequentially starting from the first element. The process of searching terminates when the list is exhausted or a comparison results in success.

Algorithm for searching an element 'key' in an array 'a[ ]' having n elements :

The search algorithm starts comparison between the first element of a[ ] and "key". As long as a comparison does not result in success, the algorithm proceeds to compare the next element of "a[ ]" with "key". The process terminates when the list is exhausted or the element is found.

'C' function for sequential search

```
int sequential(int a[ ], int key, int n)
{
    int i = 0;
    while(i < n)
    {
        if(a[i] = = key)
            return(i);
```

Technknowledge Publication

```
        i ++;
    }
    return(– 1);
}
```

The function returns the index of the element in 'a[ ]' for successful search. A value – 1 is returned if the element is not found.

Analysis of sequential search algorithm

Number of comparisons required for a successful search is not fixed. It depends on the location (place) being occupied by the key element. An element at i^{th} location can be searched after i-comparisons. Number of comparisons required is probabilistic in nature. We could best calculate the average number of comparisons required for searching an element.

Let P_i is the probability that the element to be searched will be found at i^{th} position. i number of comparisons will be required to search the ith element.

∴ Expected number of comparisons for a successful search.

$$C = 1.P_1 + 2.P_2 + \ldots + n.P_n$$

Since the element could be found at any location with equal probability.

$$P_1 = P_2 \ldots = P_n = 1/n$$

$$\therefore \quad C = \frac{1}{n} + \frac{2}{n} + \ldots + \frac{n}{n} = \frac{1}{n}(1 + 2 + \ldots + n)$$

$$= \frac{n(n+1)}{2n} = \frac{n+1}{2}$$

Time complexity of sequential search = O(n).

Program 4.2.1 : A sample program for sequential search.

```
#include<stdio.h >
#include<conio.h >
int sequential_search(int a[ ], int key, int n);
void main()
{   int a[50], n, key, i;
    printf("\n No. of elements :");
    scanf("%d", &n);
    printf("\n Enter %d numbers :", n);
    for(i = 0; i<n; i ++)
        scanf("%d", &a[i]);
    printf("element to be searched :");
    scanf("%d", &key);
    i = sequential_search(a, key, n);
    if(i = = –1)
        printf("\n  Not found");
    else
        printf("\n Found at location = %d", i + 1);
    getch();
}
int sequential_search(int a[ ], int key, int n)
{   int i;
    i = 0;
    while(i<n && key ! = a[i])
        i ++;
    /* search terminates here */
    if(i<n)
        return(i);
    return(–1);
}
```

Output

```
No. of elements : 4
Enter 4 numbers : 66    77  88    99
element to be searched : 88
Found at location = 3
```

4.2.1 Sequential Search on a Sorted Array

Expected number of comparisons required for unsuccessful search can be reduced if the array is sorted.

Array a[ ]	1	5	9	14	23	25

Element to be searched = 10 ↑ Search should terminate here

Assuming elements are sorted in ascending order, the search should terminate as soon as an element with a value equal to or greater than "key" (element to be searched) is found. After termination of search, if the value of the element is equal to "key" then search is successful.

Tech Knowledge Publication

If the value of the element is greater than "key" then search ends unsuccessfully.

'C' function for sequential search in a sorted array.

```
int search_seq_sorted(int a[ ], int key, int n)
{   int i;
    i = 0;
    while(i < n && key > a[i])
        i ++;                /* search terminates here */
    if(i < n && key = = a[i])
        return(i);
    return(–1);
}
```

4.3 Binary Search

Q. Write an algorithm for binary search. Derive recurrence relation and find out time complexity of the search. **SPPU - May 17, 7 Marks**

Q. Explain binary search with appropriate example and compare their time complexity and space complexity. **SPPU - Dec. 17, May 18, 3 Marks**

Q. Write an algorithm for searching an element using binary search. Discuss the time complexity of algorithm in best case and worst case. **SPPU - May 19, 7 Marks**

Linear search has a time complexity O(n). Such algorithms are not suitable for searching when number of elements is large. Binary search exhibits much better timing behaviour in case of large volume of data with timing complexity : $O(\log_2 n)$

Number of comparisons for n = 2^{20} (1 million)

Sequential search(in worst case) = 2^{20} comparisons.

Binary search(in worst case)=$\log_2 2^{20}$= 20 comparisons

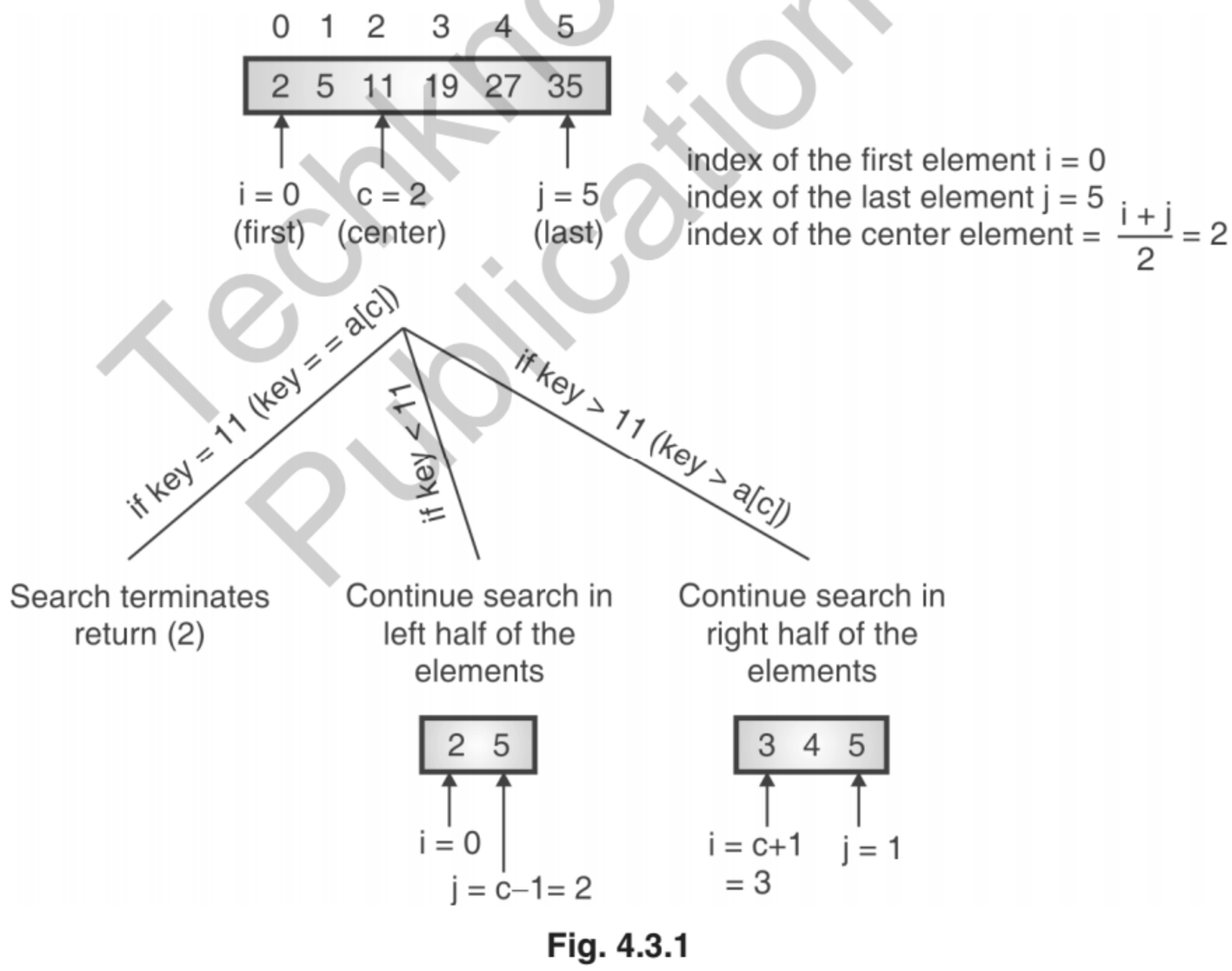


Fig. 4.3.1

Linear search (sequential search) may need 1 million comparisons for searching an element in an array having 1 million elements. Binary search will require, just 20 comparisons for the same task. Binary search uses a much better method of searching. Binary search is applicable only when the given array is sorted.

This method makes a comparison between the "key" (element to be searched) and the middle element of the array.

- Since elements are sorted, comparisons may result in either a match or comparison could be continued with either left half of elements or right half of the elements.
- Left half of elements could be selected by simply making j = c–1
- Right half of element could be selected by simply making i = c + 1
- Process of selecting either the left half or the right half continues until the element is found or element is not there.

Ex. 4.3.1 : [Searching the element 29 in the given array]

Index	0	1	2	3	4	5	6	7	8
Elements	5	9	11	15	25	29	30	35	40

i = 0 $\quad c = \frac{i+j}{2} = \frac{0+8}{2} = 4 \quad$ j = 8

Soln. : Element to be searched, key = 29

Step 1 : Since key > a[c] (29 > 25) right half is selected.

index	5	6	7	8
elements	29	30	35	40

i = 5 c = 6 j = 8

Step 2 : Since key < a[c] (29 < 30) left half is selected.

Index 5

element 29

i = 5 c = 5 j = 5

element is found and search is successful.

Ex. 4.3.2 : [searching the element 6 in the given array]

Index	0	1	2	3	4	5	6	7	8
Elements	5	9	11	15	25	29	30	35	40

Soln. :

Initially	i = 0	j = 8	c = 4	[Since key <a[c]; 6 < 25, j = c–1]
Next iteration	i = 0	j = 3	c = 1	[Since key <a[c]; 6 < 9, j = c–1]
Next iteration	i = 0	j = 0	c = 0	[Since key <a[c]; 6 < 5, j = c–1]
Next iteration	i = 1	j = 0	c = 0	Element not found.

i > j indicates, element is not found

'C' Function for binary search (Non-recursive)

```
int bin_search(int a[ ], int i, int j, int key)
{
    int c;
    c = (i + j)/2;
    while(a[c]! = key && i< = j)
    {
    /* search as long as element is not found and it
    could be there */
    if(key > a[c])
        i = c + 1; // select right half
    else
        j = c – 1; //select left half
        c = (i + j)/2;
    }
    if(i< = j)
        return(c);
    return(–1);
}
```

Analysis of binary search algorithm worst case behaviour (Unsuccessful search)

Let us assume that the array contains n elements.

∴The maximum number of elements after 1 comparison

$$= n/2$$

The maximum number of elements after 2 comparisons

$$= n/2^2$$

The maximum number of elements after h comparisons

$$= \frac{n}{2^h}$$

For the lowest value of h[maximum number of elements left = 1]

$$\therefore \frac{n}{2^h} = 1 \text{ or } 2^h = n$$

$$\text{or} \quad h = \text{Log}_2 n = O(\text{Log}_2 n)$$

Thus, the worst case behaviours of binary search algorithm has a timing complexity = $O(\text{Log}_2 n)$

A searching algorithm requires maximum number of comparisons in unsuccessful search.

Techknowledge Publication

Average case behaviour [Successful search]

Let us try to understand the average case behaviour with the help of an array having 15 elements.

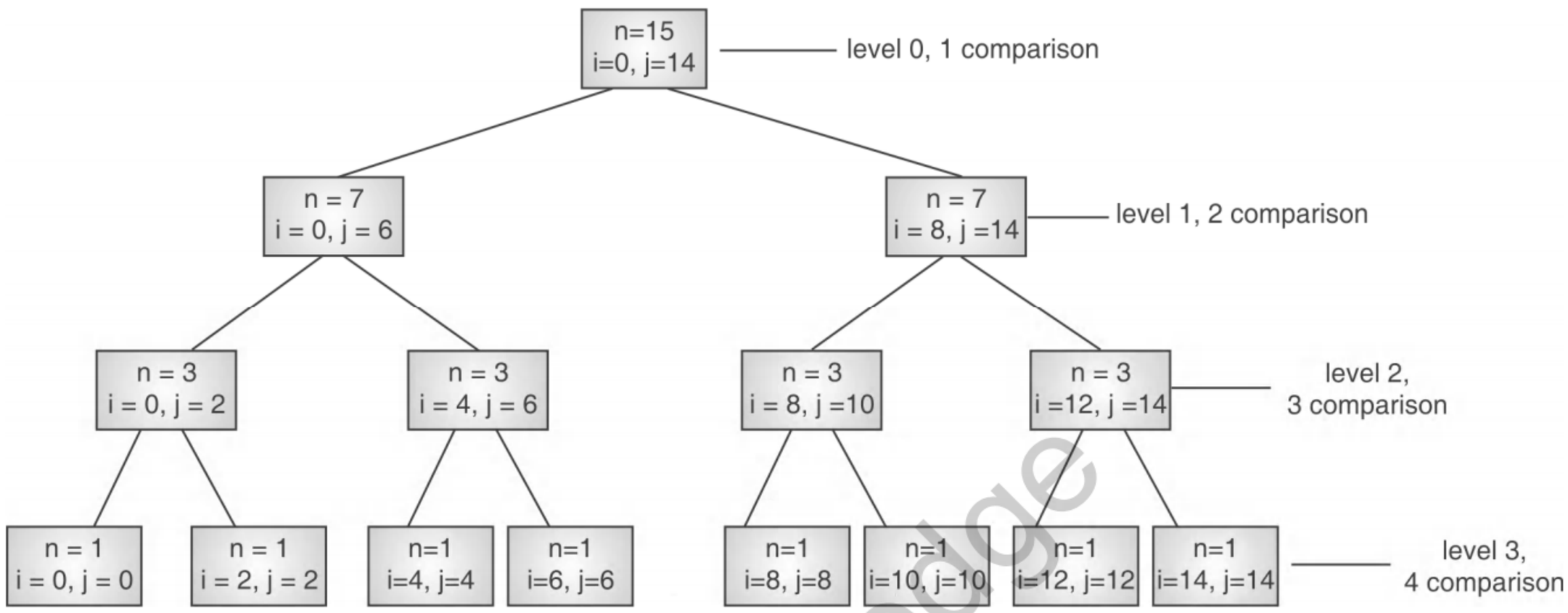


Fig. 4.3.2

- Binary search starts with 15 elements and if the element is found at the centre, search terminates with a success with 1 comparison i.e. success at level 0.
- At level 1, search starts with either left half of the elements or the right half of the elements. Probability of successful search at level 1 is twice that of a successful search at level 0. Level zero has one event (one node) and level 1 has 2 events (2 nodes) of success.
- Relation between last level (h = 3) and number of elements (n = 15) is given by $2^{3+1} = 15 + 1$

Or $\quad 2^{h+1} = n + 1$

Or $\quad h + 1 = \log_2 (n + 1)$

Or $\quad h = [\log_2 (n + 1)] - 1$

∴ Total number of comparison to search every element of array is given by :

$$C = 1.2^0 + 2.2^1 + \ldots + (h + 1)\, 2^h \quad \ldots(1)$$

$$2C = 1.2^{1+} 2.2^2 + \ldots + h2^{h+} (h + 1)\, 2^{h+1} \quad \ldots(2)$$

Subtracting Equation (2) from Equation(1)

$$-C = 1.2^{0+} 2^1 + 2^{2+} \ldots 2^h - (h + 1)\, 2^{h+1}$$

or $\quad C = (h + 1)\, 2^{h+1} - [2^{0+} 2^{1+} 2^{2+} \ldots + 2^h]$

$$= (h + 1)2^{h+1} - (-1 + 2^{h+1}) = h.\, 2^{h+1+} 1$$

Therefore, average number of comparisons

$$= \frac{h.\, 2^{h+1} + 1}{n} = \frac{h.\, 2^{h+1} + 1}{2^{h+1} - 1}$$

$$\approx \text{ h for large n} = O \log_2 (n)$$

Thus, expected number of comparisons for successful as well as unsuccessful search $O(\log_2 n)$

Analysis of binary search (Best case)

Algorithm will need just 1 comparison if the element is found at the center.

Timing complexity = O(1)

Recursive function for binary search

```
int bin_search(int a[ ], int i, int j, int key)
{
    int c;
    if(i< = j)
    {
        c = (i + j)/2;
        if(key = = a[c])
            return(c);
        / * Change it to <
        if the numbers are in descending order */
        if(key > a[c])
            return(bin_search(a, c + 1, j, key));
```

TechKnowledge Publications

```
        return(bin_search(a, i, c–1, key));
    }
    return(–1);
}
```

Program 4.3.1 : Program to implement binary search algorithm.

```
#include<stdio.h >
#include<conio.h >
void bubble_sort(int [ ], int);
int bin_search(int [ ], int, int);
void main()
{
    int a[30], n, i, key, result;
    printf("\n Enter number of elements :");
    scanf("%d", &n);
    printf("\n Enter the array elements :");
    for(i = 0; i<n; i ++)
    scanf("%d", &a[i]);
    printf("\n Enter the element to be searched :");
    scanf("%d", &key);
    bubble_sort(a, n);
    result = bin_search(a, key, n);
    if(result = = – 1)
        printf("\n Element not found :");
    else
        printf("\n Element  is found at location %d",
        result + 1);
    getch();
}
void bubble_sort(int a[ ], int n)
{
    int i, j, temp;
    for(i = 1; i<n; i ++)
        for(j = 0; j<n–i; j ++)
            if(a[j] > a[j + 1])
            {
                temp = a[j];
                a[j] = a[j + 1];
                a[j + 1] = temp;
            }
}
int bin_search(int a[ ], int key, int n)
{   int i, j, c;
    i = 0;
    j = n–1;
    c = (i + j)/2;
    while(a[c]! = key && i< = j)
    {
        if(key > a[c])
            i = c + 1;
        else
            j = c–1;
        c = (i + j)/2;
    }
    if(i< = j)
        return(c);
    return(–1);
}
```

Output

```
Enter number of elements : 6
Enter the array elements :12 34 54 23 11 90
Enter the element to be searched : 11
Element  is found at location 1
```

Ex. 4.3.3 : Apply binary search on the following numbers stored in array from A [0] to A [10]. 9, 17, 23, 38, 45, 50, 57, 76, 79, 90, 100 to search numbers – 10 to 100.

Soln. : Searching 10

Index	0	1	2	3	4	5	6	7	8	9	10
Element	9	17	23	38	45	50	57	76	79	90	100
Position	↑ i					↑ k					↑ j

i	j	k
0	10	5

Techknowledge Publication

Step 1 : Since 10 < A [5], j = k – 1 = 4

Index	0	1	2	3	4
Element	9	17	23	38	45
Position	↑ i		↑ k		↑ j

i	j	k
0	4	2

Step 2 : Since 10 < A [2], j = k – 1 = 1

Index	0	1
Element	9	17
Position	↑↑ i k	↑ j

i	j	k
0	1	0

Step 3 : Since 10 > A [0], j = k + 1 = 1

Index	1
Element	17
Position	↑↑↑ i j k

i	j	k
1	1	1

Step 4 : Since 10 < A [1], j = k – 1 = 0

As i becomes less than j, element 10 is not in the array A [ ]

Searching 100

Index	0	1	2	3	4	5	6	7	8	9	10
Element	9	17	23	38	45	50	57	76	79	90	100
Position	↑ i					↑ k					↑ j

i	j	k
0	10	5

Step 1 : Since 100 > A [5], i = k + 1 = 6

Index	6	7	8	9	10
Element	57	76	79	90	100
Position	↑ i		↑ k		↑ j

i	j	k
6	10	8

Step 2 : Since 100 > A [8], i = k + 1 = 9

Index	9	10
Element	90	100
Position	↑↑ i k	↑ j

i	j	k
9	10	9

Step 3 : Since 100 > A [9], i = k + 1 = 10

Index	10
Element	100
Position	↑↑↑ i j k

i	j	k
10	10	10

Since, the element to be searched is found at A[10], search terminates with a success.

Tech Knowledge Publications

4.3.1 Comparison between Linear and Binary Search

Q. Explain linear search with example. State its time complexity and compare it with binary search.

SPPU - Dec. 18, 7 Marks

Sr. No.	Linear search	Binary search
1.	Simple algorithm	Relatively Complex algorithm.
2.	On an average $\frac{n}{2}$ comparisons are needed	It requires log(n) comparisons.
3.	Data need not be sorted.	Data is required to be sorted.
4.	Time complexity O(n)	Time complexity O(log n)
5.	Iterative algorithm	Recursive algorithm

4.4 Fibonacci Search

Q. Write an algorithm for Fibonacci search.

SPPU - Dec. 16, 4 Marks

Fibonacci sequence is given by :

0 1 1 2 3 5 8 13 21 34 .....

which is defined as :

$F_0 = 0; \quad F_1 = 1; \quad F_i = F_{i-1} + F_{i-2}, \quad i \geq 2$

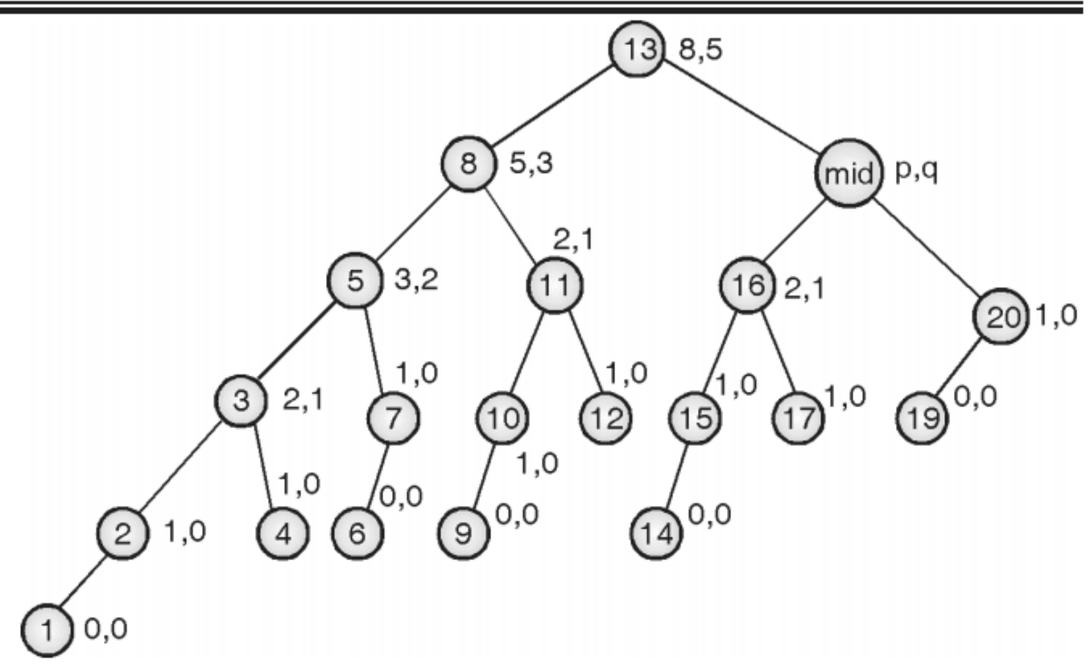
In binary search, during subsequent iterations the list is divided around the center. In Fibonacci search, the list splits according to the Fibonacci sequence.

- Fibonacci search requires that elements be ordered in non-decreasing order.
- Fibonacci search in an array with n elements starts by locating F_K,

  $F_K \geq n$ and $F_{K-1} < n$
- We have to maintain three values during iteration. Let us say that these values are mid, p, q.

Initial values of

$$\text{mid} = n - F_{K-2} + 1$$
$$p = F_{K-2}$$
$$q = F_{K-3} \qquad \text{[Starting index = 1]}$$



Initial Values : $F_K = 21$, $F_{K-1} = 13$, $F_{K-2} = 8$,

Mid = $n - F_{K-2} + 1 = 20 - 8 + 1 = 13$

Fig. 4.4.1 : Fibonacci search with n = 20

Key to be searched (say key) is compared with F_{mid} with following outcomes :

(i) if (Key < K_{mid})

if (q is 0 then search terminates with a failure)

else

[Search continues with new values of mid, p, q]

$$\text{mid} = \text{mid} - q \qquad \{q \text{ is same as } F_{K-3}\}$$
$$p = F_{K-3}$$
$$q = F_{K-4}$$

(ii) if (Key is equal to K_{mid})

Search terminates with a success.

(iii) if (Key > K_{mid})

if (P is 1 then search terminates with a failure)

else

[Search continues with new values of mid, p, q]

$$\text{mid} = \text{mid} + q$$
$$p = F_{K-4}$$
$$q = F_{K-5}$$

Decision tree for n = 20 is given in Fig. 4.4.1 for understanding of the algorithm. Values of mid, p, q are also mentioned against each node.

Search continues, either by selecting the array right of the mid or left of the mid.

In each case, three values are maintained.

Tech Knowledge Publications

Left of the mid is selected

new value of mid = mid – q

new value of p = F_{K-2}

new value of q = F_{K-3}

Right of the mid is selected

new value of mid = mid + q

new value of p = F_{K-3}

new value of q = F_{K-4}

4.4.1 Binary Search versus Fibonacci search

(1) Both are based on the principle of divide and conquer.

(2) Fibonacci search involves only addition and subtraction. Binary search requires division as well.

(3) The average performance of Fibonacci search is better than that of binary search on computers for which division takes sufficiently more time than addition / subtraction.

4.4.2 Selection of Searching Algorithm

(1) Linear search should be used when data is not sorted. A linear search may require maximum of N (number of elements) comparisons. Linear search is found to be very costly for large database.

(2) Fibonacci or binary search should be used when data is sorted.

(3) The average performance of Fibonacci search is better than that of binary search on computers for which division takes sufficiently more time than addition / subtraction.

Program 4.4.1 : C Program for Fibonacci search. Program uses a function FIBSEARCH() for fibonacci search. Function requires three parameters.

(1) Array of elements sorted in non-descending order.

(2) Number of elements in the array.

(3) Key to be searched.

Function returns the index of element, if search terminates in a success, – 1 otherwise.

```
#include <stdio.h >
#include <conio.h >
int fib(int n);          // return a Fibonacci Number
int fibsearch(int a[ ], int key, int mid , int p , int q);
                         //Fibonacci search function
void main()
{   int a[30], key, mid, p, q, k, n, i, result;
    printf("\n Enter No. of elements : ");
    scanf("%d" , &n);
    printf("\nEnter a sorted list of %d elements : ", n);
    for(i = 0; i<n; i ++)
    scanf("%d", &a[i]);
    printf("\nEnter the the element to be searched : ");
    scanf("%d", &key);
    /* locate the kth term of Fibonacci series, such that
    fib(k) > n */
    for(k = 1; fib(k) < = n; k ++)
    p = fib(k–2);
    q = fib(k–3);
    mid = n–p + 1;
    result = fibsearch(a, key, mid, p, q);
    if(result = = –1)
        printf("\n  Not found : ");
    else
        printf("\n  Found at location = %d", result);
    getch();
}
int fib(int n)
{
    if(n = = 0) return(0);
    if(n = = 1)  return(1);
    return(fib(n–1) + fib(n–2));
}
int fibsearch(int a[ ], int key, int mid , int p , int q)
{
    int temp;
    if(key = = a[mid–1])
    return(mid);
    if(key > a[mid–1])
    {   if(p = = 1)
        return(–1);
```

TechKnowledge Publications

```
        mid = mid + q;              // new mid
        p = p–q;                    //p = fib(k–4)
        q = q–p;                    //q = fib(k–5)
        return(fibsearch(a, key, mid, p, q));
    }
    else                            //key < a[mid]
    {
        if(q = = 0)
        return(–1);
        mid = mid–q;                // new mid
        temp = p–q;
        p = q;                      //p = fib(k–3)
        q = temp;                   //q = fib(k–4)
        return(fibsearch(a, key, mid, p, q));
    }
}
```

Output

```
Enter number of elements : 9
Enter a sorted list of 9 elements : 2 6 15 18 25 30 36
40 49
Enter the element to be searched : 40
Found at location = 7
```

Ex. 4.4.1 : Show the working of Fibonacci search with an example.

Soln. : Numbers : 3 5 6 11 15 17 18 20 21 25

$$n = 10$$

$$\text{Key to be searched} = 20$$

Step 1 : Finding F_K such that $F_K \geq n$.

First few numbers of Fibonacci series are given below :

0 1 1 2 3 5 8 <u>13</u> 21

It is clear that $F_7 > n$

$\{F_7 = 13 \text{ and } n = 10\}$

$$\therefore \quad K = 7$$

Step 2 : Initial values of mid, p and q are given by :

$$\text{Mid} = n - F_{K-2} + 1$$

$$= n - F_5 + 1 = 10 - 5 + 1 = 6$$

$$p = F_{K-2} = F_5 = 5$$

$$q = F_{K-3} = F_4 = 3$$

Step 3 : Iterations for Fibonacci search.

(a) Since the key (Value = 20) > a [mid– 1]

i.e. 20 > a [5] or 20 > 17

new values of mid, p and q will be :

$$\text{mid} = \text{mid} + q = 6 + 3 = 9$$

$$p = 2 \quad [F_{K-4}]$$

$$q = 1 \quad [F_{K-5}]$$

(b) Since the key (Value = 20) < a [mid – 1] {a [mid] = a(8)}

new values of,

$$\text{mid} = \text{mid} - q = 9 - 1 = 8$$

$$p = 1$$

$$q = 0$$

(c) Since, the key (Value = 20) is equal to a[7], algorithm terminates with a success.

Element is found at location = 8.

[location = 1 + index]

4.4.3 Timing Complexity of Fibonacci Search

Q. Find out time complexity of Fibonacci Search.

SPPU - Dec. 16, 3 Marks

K^{th} term of a Fibonacci series satisfies the following equation.

$$F_K = F_{K-1} + F_{K-2}$$

Characteristic equation for the recurrence relation is $X^2 - X - 1 = 0$.

$$\text{or} \quad X = \frac{1 \pm \sqrt{5}}{2}$$

Two roots of the equation are :

$$\alpha_1 = \frac{1 + \sqrt{5}}{2} \text{ and } \alpha_2 = \frac{1 - \sqrt{5}}{2}$$

K^{th} term of the Fibonacci series is given by

$$F_K = A\,\alpha_1^K + B\,\alpha_2^K$$

Techknowledge Publication

Since, $F_1 = F_2 = 1$, $A = \frac{1}{\sqrt{5}}$, $B = \frac{1}{\sqrt{5}}$

$$\therefore \quad F_K = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^K + \left(\frac{1-\sqrt{5}}{2}\right)^K\right]$$

For large K, the term $\left(\frac{1-\sqrt{5}}{2}\right)^K$ tends to 0.

$\therefore$ F_K is upper bounded by $\left(\frac{1-\sqrt{5}}{2}\right)^K$

$\therefore$ $F_K \le \alpha_1^K$ or, $\log F_K \le K \log \alpha_1$

Since, the number of comparisons required are of the order of K, where the number of elements satisfies the inequality,

$$F_K \ge n$$

$\therefore$ Timing Complexity = O (log (n))

4.5 Index Sequential Search

4.5.1 Indexing

Files are typically stored on magnetic disks. Records are stored in files. In database application, we have to store several files like :

(1) File of records of students

(2) Marks of students

(3) Fee details of students

Each record in a file s identified by a key field. Record of a student can be located by his roll number. Thus roll number is the key field.

To search for a record on disk, a record must be read in memory buffer. If the record does not contain the desired key then the next record must be read. The process of reading the next record continues until the desired record is located or all records have been read.

This can be very time consuming for a large file. The goal is to locate the record with minimum number of record accesses.

Indexing can speed up retrieval of records. In indexing, we use an additional auxiliary access structures called indexes.

Index structure provides an addition access path to quickly retrieve a record form the main file.

An index is an orderd file whose records are of fixed length with two fields :

(a) Key field (b) Record no.

Table 4.5.1

(a) Index file

Roll no.	Record no.
75	1
101	0
125	2
150	4
160	3

(b) Data file (Student file)

	Name	Roll no.	Age	———
0	Mohan	101	24	———
1	Sohan	75	23	———
2	Pradeep	125	25	———
3	Amit	160	24	———
4	Deepak	150	26	———
	|	|	|	|
	|	|	|	|
	|	|	|	|
	|	|	|	|
	|	|	|	|

Searching a record in an indexed file :

- Since an index is an ordered file, a key can be searched using binary search.
- If search ends in a success, the index entry will give the required record no.
- Record no. can be used to access the record in data file.

Ex. 4.5.1 : It is required to know the name and age of the student with roll no. = 150.

Soln. :

Step 1 : Using binary search, locate a record with roll no. = 150 in index file.

Step 2 : Get the record no. of the required student from the index file.

Record no. = 4 for roll no. = 150

Step 3 : Read record no. 4 from the data file.

Techknowledge Publication

4.5.2 Sequential File

Table 4.5.2 : Data file (File is ordered on roll no.)

Name	Roll no.	Age
Sohan	75	23
Mohan	101	24
Pradeep	125	25
Deepak	150	26
Amit	160	24
|	|	|
|	|	|
|	|	|

We can physically order the records of a file on disk based on the value of one of their fields called the Ordering field. Data file as shown in the Table 4.5.2 is ordered on roll no.

Advantages of a sequential file

(a) Reading the records in order of ordering key values is extremely efficient.

(b) Searching a record on an ordering key field results in faster access as binary search technique can be used.

Disadvantages of sequential file

Inserting and deleting records are expensive as records must remain physically ordered. To insert a record, we must find its correct position in the file and then make space in the file to insert the record in that position. similarly, to delete a record, we must find the record to be deleted and then physically move all records ahead of it, backward by 1 place.

4.5.3 Indexed Sequential File

Records of an indexed sequential file are sorted on ordering key value. Another file called a Sparse index is also created. As shown in the Fig. 4.5.1, the index file contains a very few entries. A set of records can be accessed through a single entry in index file.

Since, a hard disk in accessed block wise, we can have one entry in index file for each block in data file.

Each index entry in this case will be the value of the primary key field for the first record in a block and a pointer to that block.

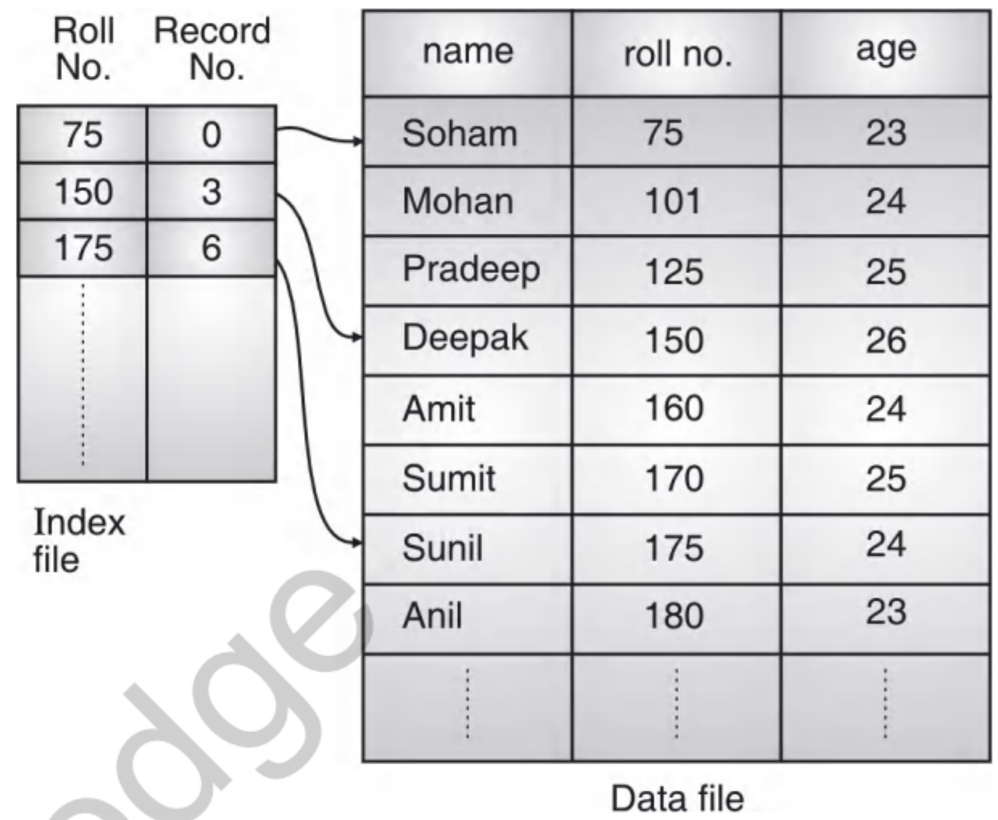


Fig. 4.5.1

Searching a record

To retrieve a record with a given key x, we first search the index file for a pair (x, b). b is the block no. of a block having a record with key value x. In reality, we may not get an entry (x, b) in the index file. We should search for the largest y such that $y \leq x$ and there is a pair (y, b) in the index file. Then key x appears in block b if it is present in the main file.

Several strategies can be used to search the index file.

(1) Linear search (2) Binary search

Linear search is suitable for small index file. A better method of searching is binary search.

After the pair (y, b) is located in the index file, data file block b is retrieved. Now, sequential search can be carried out on the data block to retrieve the record with key equal of x.

Advantages and disadvantages of index sequential search

The index file of a indexed sequential file needs substantially fewer blocks than does the data file as :

(a) There are fewer index entries than there are records in the data files.

Tech Knowledge Publications

(b) Each index entry is smaller in size than a data record. More index entries can fit in one block.

A binary search on the index file requires fewer block accesses than a binary search on the data file.

A major problem with indexed sequential file is insertion and deletion of records. With index file, the problem is compounded, if we attempt to insert a record in its correct position in data file, we have to not only move records to make space for the new record but also change index entries.

4.6 Sentinel Search

- An extra record inserted at the end of an array is called a sentinel. The speed of the sequential search can be improved by storing the key being searched at the end of the array.
- This will eliminate the extra comparison inside the loop for number of elements in the array.

The example is given below

```
int sentinel search(int a[ ], int n , int key)
{   int i;
    a[n] = key;
    i = 0;
    while(a[i] ! = key)
        i ++;
    return(i);
}
```

In the above function, we do not compare i with n as the algorithm will always terminate with success.

4.7 Case Study - Use of Fibonacci Search

The case study is being given for two cases :

1. Fibonacci search in non-uniform access memory storage
2. Fibonacci search in optimization of unimodal function.

Fibonacci search technique uses divide and conquer algorithm. The technique narrows down search range with the aid of Fibonacci numbers. The Fibonacci search is preferred over binary search, in some cases, as it slightly reduces the average time needed to access a storage location by examining locations having lower dispersion.

Non-uniform Access Memory Storage

In non-uniform access memory, memory access time is not uniform. For example : In case of hard disk, it is much faster to access the next element in the same cylinder. An element in the same cylinder can be accessed without further movement of the read/write head. Here, the access time will be shorter.

Similarly, to move the reading arm across less cylinders is faster than moving it across more cylinders. Since, the Fibonacci search inherently searches elements having lower dispersion; it will perform better in case of non-uniform access memory storage.

Optimization of Unimodal Function

A common function f(x) is a unimodal function if for some value m, it is monotonically increasing for $x \le m$ and monotonically decreasing for $x \ge m$. f(m) is the maximum value of the function and there are no other local maxima. Fibonacci search can be used for the maximum (minimum) of a unimodal function in an interval.

Review Questions

Q. 1 Explain linear search and binary search. **(4 Marks)**

Q. 2 Write an algorithm for linear search technique. **(6 Marks)**

Q. 3 Write an algorithm for searching an element in a list of integers using binary search. Discuss the time complexity of algorithm in best case and worst case. **(6 Marks)**

Q. 4 Discuss time complexity of linear search. **(2 Marks)**

❑❑❑

Techknowledge Publication

CHAPTER 5

Unit III

Sorting

Syllabus :

Sorting : Types of Sorting-Internal and External Sorting, General Sort Concepts-Sort Order, Stability, Efficiency, and Number of Passes, Comparison Based Sorting Methods-Bubble Sort, Insertion Sort, Selection Sort, Quick Sort, Shell Sort, Non-comparison Based Sorting Methods-Radix Sort, Counting Sort, and Bucket Sort, Comparison of All Sorting Methods and their complexities.

Exemplar/Case Studies : Timesort as a hybrid stable sorting algorithm

5.1 Sorting

Sorting is a process of ordering a list of elements in either ascending or descending order. Sorting can be divided into two categories :

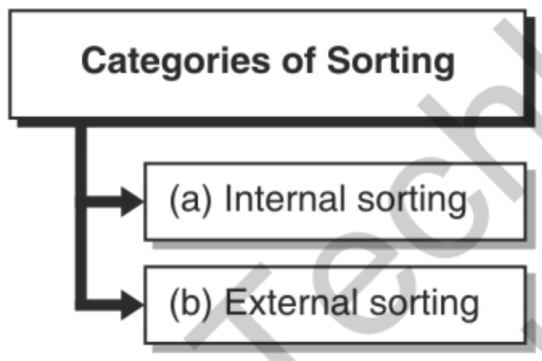


Fig. 5.1.1 : Categories of Sorting

(a) Internal sorting

Internal sorting takes place in the main memory of the computer. Internal sorting can take advantage of the random access nature of the main memory. Elements to be sorted are stored in an integer array. These elements can be sorted using one of the various algorithms discussed in this chapter.

(b) External sorting

External sorting is carried on secondary storage. External sorting becomes a necessity if the number of elements to be sorted is too large to fit in main memory. External sorting algorithms should always take into account that movement of data between secondary storage and main memory is best done by moving a block of contiguous elements.

- Simple sorting algorithms like bubble sort, insertion sort take $O(n^2)$ time to sort n elements.
- More complicated algorithms like quick sort, merge sort, heap sort take O(n log n) time to sort n elements.
- Sorting algorithms like bubble sort, insertion sort, merge sort are stable. Whereas quick sort and heap sort are unstable. A sorting algorithm is said to be **stable** if after sorting, identical elements appear in the same sequence as in the original unsorted list.

5.1.1 Sort Stability

Q. Write a short note on stability of sorting. Compare heap sort and quick sort with one example and discuss time complexity.

SPPU - Dec. 16, May 17, 7 Marks

Q. Write pseudo C/C++ code for radix sort.

SPPU - May 19, 6 Marks

Let us try to understand the concept of sort stability with the help of an example.

Technical Knowledge Publication

Table 5.1.1 : Unsorted list

Name	Subject	Marks
Mohan	Phy	65
Sohan	Che	70
Mohan	Che	68
Amit	Che	74
Sohan	Phy	75

Table 5.1.2 : Sorted list (stable)

Name	Subject	Marks
Amit	Che	74
Mohan	Phy	65
Mohan	Che	68
Sohan	Che	70
Sohan	Phy	75

Table 5.1.3 : Sorted list (not stable)

Name	Subject	Marks
Amit	Che	74
Mohan	Che	68
Mohan	Phy	65
Sohan	Che	70
Sohan	Phy	75

- Table 5.1.1 gives a list of unsorted records. These records are to be sorted on name.
- Two records appear with the name 'Mohan'. Similarly there are two records with name 'Sohan'. Any sorting method will place the records with non-distinct keys in consecutive locations. If the original ordering among the records with non-distinct keys is preserved in sorted file then such a sorting method is known as stable.
- Sorted list of Table 5.1.2 has been produced using a stable sorting method. Relative ordering of two records of 'Mohan' and two records of 'Sohan' is preserved in sorted list.
- Sorted list of Table 5.1.3 has been produced using a sorting method which is not stable. Relative ordering of two records of 'Mohan' is not preserved in sorted list.

Some examples of stable sorting algorithms are :

(1) Bubble sort (2) Selection sort

(3) Insertion sort (4) Merge sort

Some examples of unstable sorting algorithms are :

(1) Quick sort (2) Heap sort

(3) Shell sort

5.1.2 Sort Efficiency

- In order to find the amount of time required to sort a list of n elements by a particular method, we must find the number of comparisons required to sort. Some methods are data sensitive and for such methods finding of exact number of comparisons become difficult. Usually, data sensitive algorithms are analyzed for various cases.

  (1) Best case (2) Worst case

  (3) Average case

- Average case of an algorithm is calculated with an assumption that data distribution is random. Most of the primitive sorting algorithms like bubble sort; selection sort and insertion sort are not suited for sorting a large file. These algorithms have a timing requirement of $O(n^2)$.
- On the contrary, these sorting algorithms require hardly any additional memory space for sorting. A list with few records should preferably be sorted using these algorithms. Advance sorting algorithms like quick sort, merge sort and heap sort have a timing requirement of O(n log n) but they too have some additional problems as :

  (1) Complex algorithm

  (2) Complex data structure

Technknowledge Publication

(3) Unstable sorting algorithm

(4) Additional memory requirement

(5) Data sensitivity

– These algorithms should be used when the list to be sorted contains considerably large number of records.

5.1.3 Passes

– Most of the sorting algorithms work in passes. In every pass, a number is placed at position where it will appear in the sorted list. Inside a pass, numbers are compared and exchanged as required by the algorithm.

– For example, if a list of a n elements is to be sorted using bubble sort then sorting of n elements will require n – 1 passes. If the elements are stored in an array from a[0] to a[n – 1] then at the end of pass 1, largest element will be in a[n – 1] at the end of pass 2, second largest element will be in a[n – 2]. At the end of pass n – 1, the smallest element will be in a[0].

In the loop of bubble sort, given below.

```
for (i = 1; i < n; i++)
    for (j = 0; j < n – i; j++)
    {
    }
```

Outer loop on i is for passes. Inner loop is for comparison and exchange (if required) of two adjacent elements.

5.1.4 Importance of Sorting and Searching

Sorting and searching are fundamental operations which are frequently used in many applications. A typical example where searching technique is applied is telephone directory.

We may consider the telephone directory as a file, each record having three fields : name, address and phone number. The key is usually the person's name. There are two techniques for searching :

(1) Linear search

(2) Binary search

Linear search may need maximum of n–1 comparisons. Binary search is a much better searching method with (log n) comparisons. Binary search key requires the file to be sorted on search key. Thus searching is a fundamental operation and it works faster with sorted data. In most of the data processing activities like.

(1) Salary processing

(2) Inventory control

(3) Financial accounting

(4) Share processing

(5) Transaction processing in a bank

(6) Sales purchase

Sorting plays a very important role. Master file and transaction file are kept sorted on binary key. This makes merging and searching algorithms to be more efficient. Similarly, in database applications, most of the tables are sequenced for efficient execution of range based queries.

5.2 Insertion Sort

An element can always be placed at a right place in sorted list of element.

Example : List of elements (sorted) 5 9 10 15 20

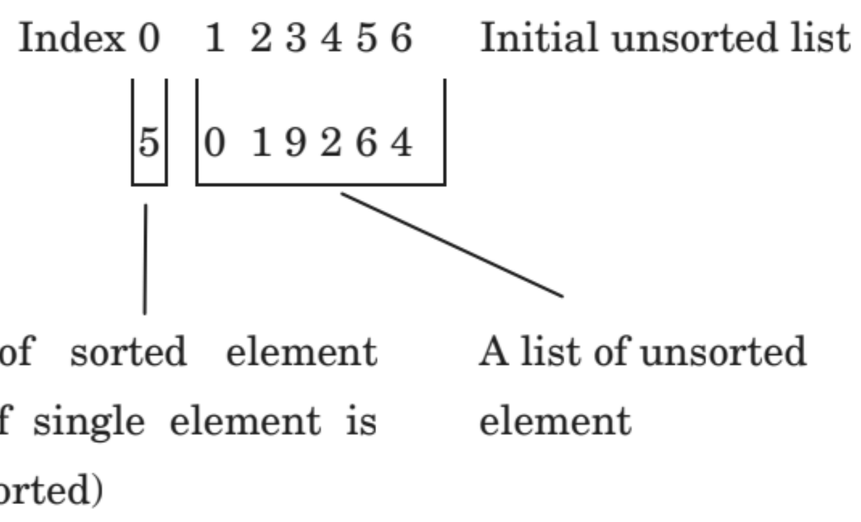
Element to be placed = 6

It the element 6 is to be inserted in a sorted list of elements (5, 9, 10, 15, 20), its rightful place will be between 5 and 9. Elements with value > 9 should be moved right by one place. Thus, creating a space for the incoming element.

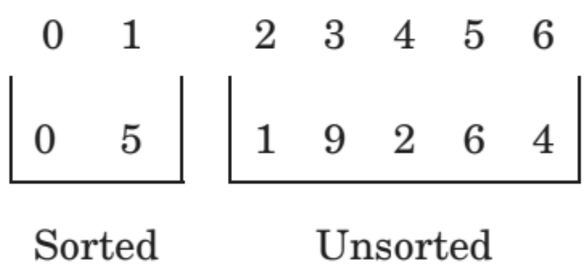
5	9	10	15	20		→	Moved right by 1 place
5	6	9	10	15	20	→	Element 6 is inserted between 5 and 9

Tech Knowledge Publications

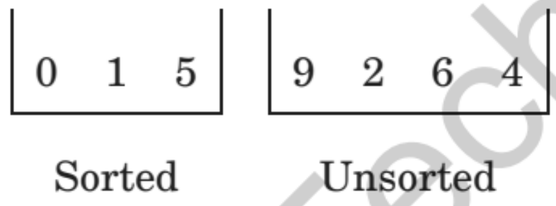
Insertion sort is based on the principle of inserting the element at its correct place in a previously sorted list. I can be varied from 1 to n – 1 to sort the entire array.



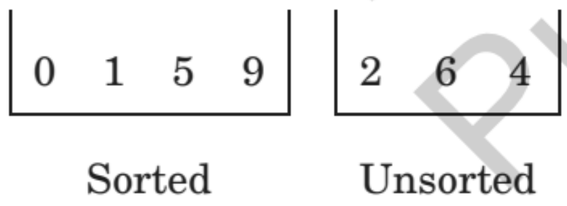
1st iteration (place element at location '1' i.e. a[1], at its correct place)



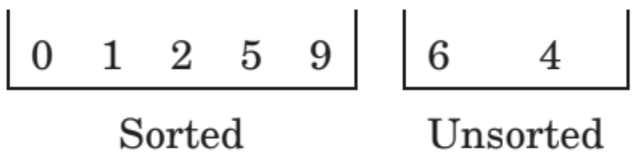
2nd iteration (place a[2] at its correct place)



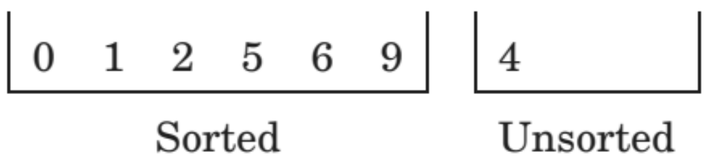
3rd iteration (place a[3] at its correct place)



4th iteration (Place a[4] at its correct place)



5th iteration (Place a[5] at its correct place)

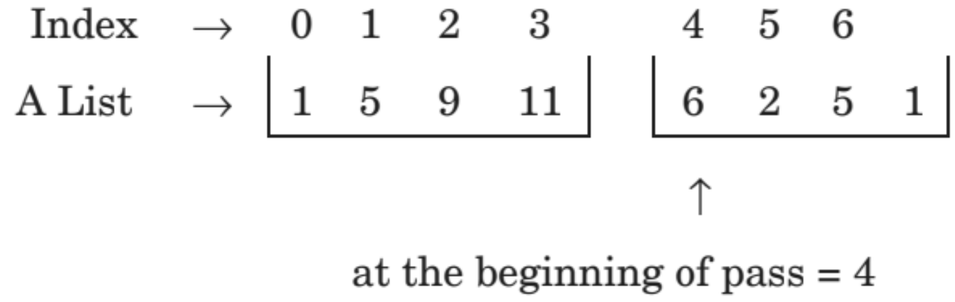


6th iteration (Place a[6] at its correct place)

0 1 2 4 5 6 9

Fig. 5.2.1 : Sorting of elements using insertion sort

Insertion sort requires n–1 passes to sort an array having n elements. Before the ith pass starts, elements at position 0 through i–1 are already sorted.



Element at position i = 4 can be inserted at its correct place after he following operations.

```
temp = a[4]
a[4] = a[3]
a[3] = a[2]
a[2] = temp
```

All elements larger than 6 are moved right by 1 place. i^{th} element is saved in the variable temp to protect its value before it is overwritten due to data movement.

i^{th} element can also be inserted at its correct place with the help of the following program segment.

```
temp = a[i] ;
for( j = i – 1 ; j> = 0 && temp < a[ j] ; j – –)
a[ j + 1] = a[j];              /*move input */
a[ j + 1] = temp;
```

'C' function for insertion sort

```
void insertion_sort(int a[ ], int n)
{
    int i, j, temp;
    for(i = 1; i<n; i++)  /* passes 1 through n – 1 */
    {
        temp = a[i];
        for( j = i – 1; j > = 0 && a[ j] > temp; j – –)
            a[j + 1] = a[ j];
            a[j + 1] = temp;
    }
}
```

Technknowledge Publication

Analysis of insertion sort

For loop of lines (3 – 4) will be executed $\sum_{i-1}^{n-1} i$ times under its worst case behaviour. If the number to be sorted are initially in descending order then the for loop of lines (3 – 4) will make i iterations during pass i.

20 10 8 6 4 2 1 – **initially**

Elements							Pass	Positions moved
10	20	8	6	4	2	1	1	1
8	10	20	6	4	2	1	2	2
6	8	10	20	4	2	1	3	3
4	6	8	10	20	2	1	4	4
2	4	6	8	10	20	1	5	5
1	2	4	6	8	10	20	6	6

Thus, the total number of movement of data (if the list is initially in descending order) for sorting using insertion sort = $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$.

Worst case behaving $O(n^2)$

If the input list presented for sorting is presorted the running time will be O(n), because the test a[j] > temp in the inner loop (lines 3 – 4) will fail immediately. Thus only one comparison is made in each pass.

Total number of comparisons = n – 1

Best case behaviour O(n)

Insertion sort can exploit the partially sorted nature of data. Insertion sort is highly efficient if the array is already in almost sorted order.

Program 5.2.1 : A sample program for insertion sort (for an array of integers).

```
#include<conio.h>
#include<stdio.h>
void insertion_sort(int[ ], int);
void main()
{
    int a[50], n, i;
    printf("\n Enter no. of elements :");
    scanf("%d", &n);
    printf("\n Enter array elements :");
    for(i = 0; i < n; i++)
        scanf("%d", &a[i]);
    insertion_sort(a, n);
    printf("\n Sorted array is :");
    for(i = 0; i<n; i++)
        printf("%d", a[i]);
    getch();
}
void insertion_sort(int a[ ], int n)
{
    int i, j, temp;
    for(i = 1; i < n; i++)
    {
        temp = a[i];
        for( j = i–1; j > = 0 && a[ j]>temp; j– –)
            a[ j+1] = a[j];
            a[ j+1] = temp;
    }
}
```

Output

```
Enter no. of elements :  5
Enter array elements :  57  89  64  56  77  333
Sorted array is      :  57  77  89  333  64
56
```

Ex. 5.2.1 : Show all passes to sort the values in descending order using insertion sort.

56, 12, 84, 56, 28, 0, –13, 47, 94, 31, 12, – 2

Tech Knowledge Publication

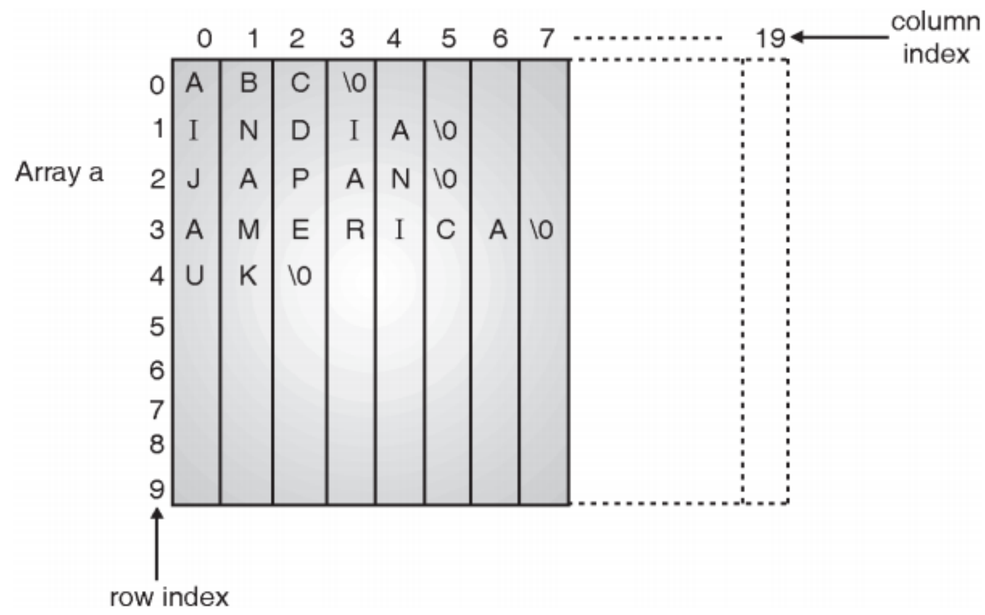
Soln. :

Initial	56	12	84	56	28	0	–13	47	94	31	12	–2
After pass 1	56	12	84	56	28	0	–13	47	94	31	12	–2
	Sorted											
After pass 2	84	56	12	56	28	0	–13	47	94	31	12	–2
	Sorted											
After pass 3	84	56	56	12	28	0	–13	47	94	31	12	–2
	Sorted											
After pass 4	84	56	56	28	12	0	–13	47	94	31	12	–2
	Sorted											
After pass 5	84	56	56	28	12	0	–13	47	94	31	12	–2
	Sorted											
After pass 6	84	56	56	28	12	0	–13	47	94	31	12	–2
	Sorted											
After pass 7	84	56	56	47	28	12	0	–13	94	31	12	–2
	Sorted											
After pass 8	94	84	56	56	47	28	12	0	–13	31	12	–2
	Sorted											
After pass 9	94	84	56	56	47	31	28	12	0	–13	12	–2
	Sorted											
After pass 10	94	84	56	56	47	31	28	12	12	0	–13	–2
	Sorted											
After pass 11	94	84	56	56	47	31	28	12	12	0	–13	–2
	Sorted											

Technical Knowledge Publications

5.2.1 Sorting an Array of Strings using Insertion Sort

A list of strings can be represented using a two dimensional array.



For example : A list { “ABC”, “INDIA”, “JAPAN”, “AMERICA”, “UK” } of strings can be represented as given follows :

char a[10][20];

a[0] is the string “ABC”

a[1] is the string “INDIA”

a[2] is the string “JAPAN”

a[3] is the string “AMERICA”

a[4] is the string “UK”

– Two numbers a[j] and temp are compared using the relational operation “a[j] > temp”. Whereas the two string a[j] and temp should be compared using the library function strcmp(a[j], temp).

– Function strcpy(a[j+1], a[j]) should be used to move the string from location j to j+1 in the array.

Program 5.2.2 : Program showing sorting of strings using insertion sort.

```
#include<conio.h>
#include<stdio.h>
#include<string.h>
void insertion_sort(char[ ][20], int);
void main( )
{   char a[10][20];
    int i, n;
    printf("\n Enter no. of strings :");
    scanf("%d", &n);
    printf("\n Enter strings :");
    for(i = 0; i<n; i++)
        gets(a[i]);
        insertion_sort(a, n);
        printf("\n Sorted strings :\n ");
    for(i = 0; i<n; i++)
        printf("%s", a[i]);
    getch();
}
void insertion_sort(char a[ ][20], int n)
{   int i, j;
    char temp[20];
    for(i = 0; i<n; i++)
    {
     strcpy(temp, a[i]);
     for(j = i–1; j>= 0 && strcmp(a[j], temp)>0; j– –
)
     strcpy(a[j+1], a[j]);
     strcpy(a[j+1], temp);
    }
}
```

Output

```
Enter no. of strings :  5
Enter strings     : xyz    abc  ccc aaa
Sorted strings    : aaa    abc  ccc xyz
```

5.2.2 Sorting an Array of Records on the given Key using Insertion Sort

A company maintains record of its employees. Each record has the following format :

ENO	NAME	OCCUPATION	LOCATION

It is required to sort employee records on "ENO".

Program 5.2.3 : Program to sort employee records on "ENO".

```
#include<stdio.h>
#include<conio.h>
struct employee
{
    int ENO;
    char name[30];
    char occup[15];
    char loca[20];
};
void main()
{
    struct employee temp;
    int i, j, n;
    static struct employee employees[ ]
        = {{15, "MOHAN", "PROG", "DELHI"},
        {5, "RAM", "ANAL", "PUNE"},
        {12, "JOHN", "MANA", "MUMBAI"},
        {3, "SOHAN", "PROG", "NASIK"}  };
    n = 4;
    for(i = 1; i<n; i++)
    {
        temp = employees[i];
        for(j = i – 1; j> = 0 && employees[j].ENO >
        temp.ENO; j – –)
        employees[j+1] = employees[j];
        employees[j+1] = temp;
    }
    for(i = 0; i<n; i++)
    {
            printf("\n %d \t %7s \t %5s \t %7s",
            employees[i].ENO,     employees[i].name,
            employees[i].occup, employees[i].loca);
        getch();
    }
}
```

Techknowledge Publication

Output

```
3     SOHAN    PROG    NASIK
5     RAM      ANAL    PUNE
12    JOHN     MANA    MUMBAI
15    MOHAN    PROG    DELHI
```

Ex. 5.2.2 : Here are five integers 1, 7, 3, 2, 0. Sort them using insertion sort.

Soln. :

Pass (i)	Comparisons (j)	List to Sort	Remarks
		1 7 3 2 0	original list
i = 1	j = 0	1 7 3 2 0	7 > 1, therefore inner loop terminates
i = 2	j = 1	1 7 3 2 0	7 > 3, move 7 right
	j = 0	1 7 7 2 0	3 > 1, inner loop terminates
		1 3 7 2 0	insert 3
i = 3	j = 2 j = 1 j = 0	1 3 7 7 0 1 3 3 7 0 1 3 3 7 0 1 2 3 7 0	7 > 2, move 7 right 3 > 2, move 2 right 2 > 1, inner loop terminates insert 2
i = 4	j = 3 j = 2 j = 1 j = 0 j = – 1	1 2 3 7 7 1 2 3 3 7 1 2 2 3 7 1 1 2 3 7 1 1 2 3 7 0 1 2 3 7	7 > 0 3 > 0 2 > 0 1 > 0 j = – 1, inner loop terminates Insert 0

Hence the sorted list = {0, 1, 2, 3, 7}

Ex. 5.2.3 : Show that the average case running time of insertion sort is $O(n^2)$.

Soln. :

Number of comparisons (average) required to locate the point of insertion of the ith element.

$$= \frac{1}{i}[1 + 2 + 3 + \ldots + i] = \frac{1}{i} \times \frac{i\,[i + 1]}{2} = \frac{i + 1}{2}$$

Since, in the outer loop i varies from 1 to n – 1.

∴ Total number of comparisons

$$= \sum_{i=1}^{n-1} \frac{i + 1}{2} = \frac{1}{2} \sum_{i=1}^{n-1} (i + 1) = \frac{1}{2}\left[(n - 1) + \frac{n\,(n - 1)}{2}\right]$$

$$= \frac{1}{2}\left[\frac{2\,(n - 1) + n\,(n - 1)}{2}\right] = \frac{1}{4}\,[(n + 2)\,(n - 1)]$$

$$= \frac{1}{4}[n^2 + n - 2] = O\,(n^2)$$

Ex. 5.2.4 : Sort the following nos. using insertion sort. Show all passes : 50, 10, 78, 40, 30, 02, 04, 15.

Soln. :

Initial list	50	10	78	40	30	02	04	15
After pass 1	10	50	78	40	30	02	04	15
	Sorted							
After pass 2	10	50	78	40	30	02	04	15
	Sorted							
After pass 3	10	40	50	78	30	02	04	15
	Sorted							
After pass 4	10	30	40	50	78	02	04	15
	Sorted							
After pass 5	02	10	30	40	50	78	04	15
	Sorted							
After pass 6	02	04	10	30	40	50	78	15
	Sorted							
After pass 7	02	04	10	15	30	40	50	78
	Sorted							

Ex. 5.2.5 : Show the output of each pass using insertion sort to arrange the following numbers in ascending order : 150, 350, 100, 250, 200, 50, 300.

Soln. :

Initial list	150	350	100	250	200	50	300
After pass 1	150	350	100	250	200	50	300
	Sorted						
After pass 2	100	150	350	250	200	50	300
	Sorted						
After pass 3	100	150	250	350	200	50	300
	Sorted						
After pass 4	100	150	200	250	350	50	300
	Sorted						
After pass 5	50	100	150	200	250	350	300
	Sorted						
After pass 6	50	100	150	200	250	300	350
	Sorted						

Ex. 5.2.6 : Sort the following numbers using insertion sort 55, 85, 45, 11, 34, 05, 89, 99, 67
Discuss its time complexity and space complexity.
SPPU - May 18, 6 Marks

Soln. :

Initial list	55	85	45	11	34	05	89	99	67
After pass 1	55	85	45	11	34	05	89	99	67
	Sorted								
After pass 2	45	55	85	11	34	05	89	99	67
	Sorted								
After pass 3	11	45	55	85	34	05	89	99	67
	Sorted								
After pass 4	11	34	45	55	85	05	89	99	67
	Sorted								
After pass 5	05	11	34	45	55	85	89	99	67
	Sorted								
After pass 6	05	11	34	45	55	85	89	99	67
	Sorted								
After pass 7	05	11	34	45	55	85	89	99	67
	Sorted								
After pass 8	05	11	34	45	55	67	85	89	99
	Sorted								

$$\text{Space Complexity} = O(n)$$

Ex. 5.2.7 : Explain insertion sort algorithm and sort the given list using insertion sort : 7, 4, 10, 6, 3, 12, 1, 8, 2, 15, 9, 5.
SPPU - May 19, 6 Marks

Soln. :

Initial list	7	4	10	6	3	12	1	8	2	15	9	5
Pass 1	4	7	10	6	3	12	1	8	2	15	9	5
	Sorted											
Pass 2	4	7	10	6	3	12	1	8	2	15	9	5
	Sorted											
Pass 3	4	6	7	10	3	12	1	8	2	15	9	5
	Sorted											
Pass 4	3	4	6	7	10	12	1	8	2	15	9	5
	Sorted											
Pass 5	3	4	6	7	10	12	1	8	2	15	9	5
	Sorted											
Pass 6	1	3	4	6	7	10	12	8	2	15	9	5
	Sorted											
Pass 7	1	3	4	6	7	8	10	12	2	15	9	5
	Sorted											
Pass 8	1	2	3	4	6	7	8	10	12	15	9	5
	Sorted											
Pass 9	1	2	3	4	6	7	8	10	12	15	9	5
	Sorted											
Pass 10	1	2	3	4	6	7	8	9	10	12	15	5
	Sorted											
Pass 11	1	2	3	4	5	6	7	8	9	10	12	15
	Sorted											

Tech Knowledge Publications

5.3 Bubble Sort

Bubble sort is one of the simplest and the most popular sorting method. The basic idea behind bubble sort is as a bubble rises up in water, the smallest element goes to the beginning. This method is based on successive selecting the smallest element through exchange of adjacent element.

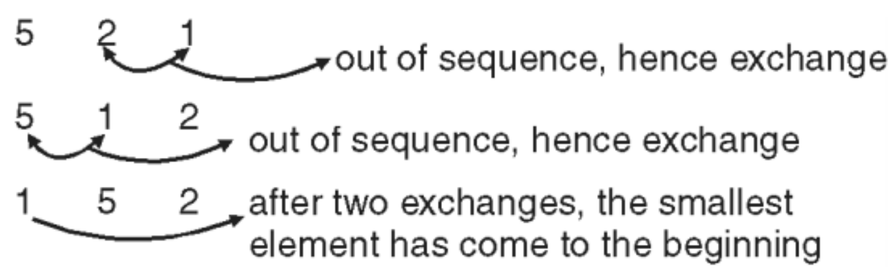


Fig. 5.3.1 : First pass of the bubble sort

Let n be the number of elements in an array a[ ]. The first pass begins with the comparison of a[n – 1] and a[n – 2]. If a[n – 2] is larger than a[n – 1], the two elements are exchanged. The smaller elements, now at a[n – 2] is compared with a[n – 3] and if necessary the elements are exchanged to place the smaller one in a[n – 3]. Comparison progresses backward and after the last comparison of a[1] and a[0] and possible exchange the smallest element will be placed at a[0].

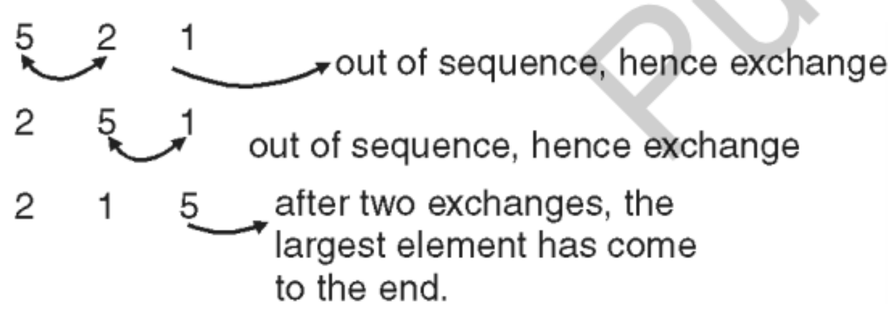


Fig. 5.3.2 : First pass of the bubble sort with comparison in the forward direction.

As a variation, the first pass could begin comparison with a[0] and a[1]. If a[0] is larger than a[1], the two elements are exchanged. Larger one will be placed in a[1] after the first comparison. Comparison can work forward and after the last comparison of a[n – 2] and a[n – 1], the larger element will be placed in a[n –1].

The **second pass** is an exact replica of the first pass except that this time, the pass ends with the comparison and possible exchange of a[n – 3] and a[n – 2]. After the end of second pass, the second largest element will be placed at a[n – 2].

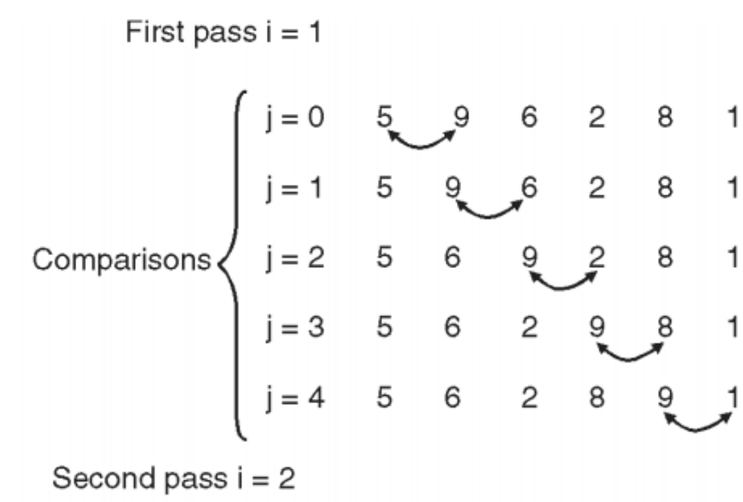
Bubble sort requires a total of n – 1 passes. If n – 1 elements are arranged (one element each pass) in ascending order form a[1] to a[n – 1], smallest element will finally left at a[0].

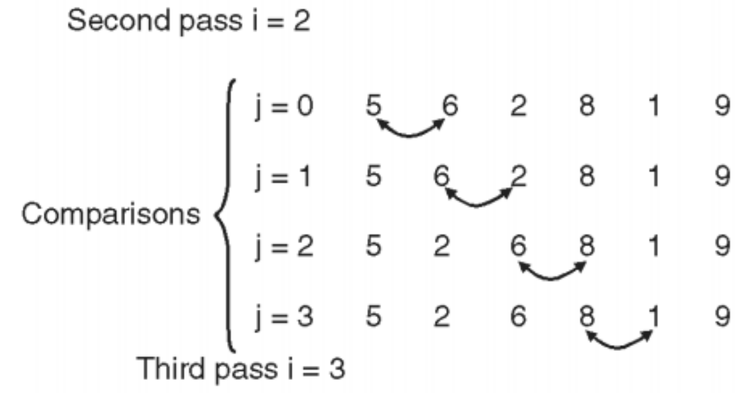
Pseudo code for bubble sort

Loop for sorting an array a[ ] having n elements

```
for(i = 1; i < n; i ++)
    for(j = 0; j < n – i; j ++)
        if(a[j] > a[j + 1])
            exchange a[j] and a[j + 1]
Outer loop is for passes and the inner loop is for
comparisons.
In pass 1(i = 1),
        there will be n–1 comparisons (j = 0 to n–2)
In pass 2(i = 2),
        there will be n–2 comparisons (j = 0 to n–3)
:
:
In pass n – 1 (i = n–1),
        there will be 1 comparison (j = 0 to 0)
```

Original array with n = 6, Elements : 5 9 6 2 8 1





TechKnowledge Publications

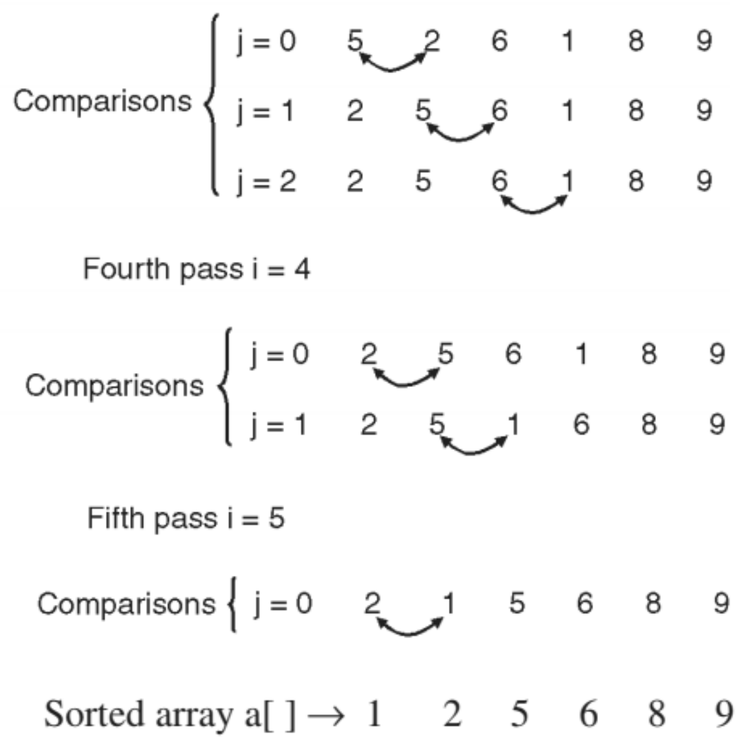


Fig. 5.3.3 : Illustration of bubble sort

Program 5.3.1 : Program for sorting an integer array using bubble sort.

```
#include<conio.h>
#include<stdio.h>
void bubble_sort(int[ ], int);
void main()
{
    int a[30], n, i;
    printf("\n Enter no. of elements :");
    scanf("%d", &n);
    printf("\n Enter array elements :");
    for(i = 0; i<n; i++)
        scanf("%d", &a[i]);
    bubble_sort(a, n);
        printf("\n Sorted array is :");
    for(i = 0; i<n; i++)
        printf("%d", a[i]);
    getch();
}
void bubble_sort(int a[ ], int n)
{
    int i, j, temp;
    for(i = 1; i<n; i++)
        for(j = 0; j<n–i; j++)
            if(a[j]>a[j+1])
            {
              temp = a[j];
              a[j] = a[j+1];
              a[j+1] = temp;
            }
}
```

Output

```
Enter no. of elements : 5
Enter array elements :57 89   64  56  77   333
Sorted array is :     56  57  64  77  89  333
```

Improved version of bubble sort

- In our algorithm for bubble sort, sorting algorithm requires n – 1 passes. The method may even be terminated earlier if no exchange is found necessary in an earlier pass. Absence of any exchange in a pass ensures that the elements are already sorted and there is no point in continuing further.
- Absence of any exchange in inner loop of bubble sort can be detected through a variable flag.
- Before entering the inner loop, flag is set to 0(flag = 0)
- If an exchange is required then flag is set to 1 in inner loop.
- On exiting the inner loop, if the value of the flag is found to be 0, algorithm terminates.

'C' function for improved bubble sort.

```
void imp_bubble sort(int a[ ], int n)
{
    int i, j, temp, flag;
    flag = 1;
    for(i = 1; i < n && flag = = 1; i + +)
    {
        flag = 0
        for(j = 0; j < n – i; j + +)
        if(a[j] > a[j + 1])
        {
            flag = 1;
            temp = a[j] ;
            a[j] = a[j + 1];
```

Techknowledge Publication

```
            a[j + 1] = temp;
        }
    }
}
```

Analysis of bubble sort

Let us consider the loop for bubble sort :

Line 1 for(i = 1; i < n; i + 1)

Line 2 for(j = 0; j < n – i; j ++)

Line 3 if(a[j] > a[j + 1])

Line 4 exchange a[j] and a[j + 1];

For a fixed value of i, the loop of lines(2 – 4) runs n – i times.

Hence total number of comparisons

$$= \sum_{i=1}^{n-1} (n-i) = (n-1) + (n-2) + \ldots\ 3 + 2 + 1$$

$$= \frac{n(n-1)}{2} = \frac{1}{2}(n^2 - n) = \mathbf{O(n^2)}$$

Ex. 5.3.1 : Show output of each pass using bubble sort to arrange the following nos. in ascending order. Write pseudo C code for bubble sort : 10, 9, 8, 7, 6, 5, 4, 3, 2, 1.

Soln. :

Pass No.	Data at the end of the pass
1.	9, 8, 7, 6, 5, 4, 3, 2, 1, 10
2.	8, 7, 6, 5, 4, 3, 2, 1, 9, 10
3.	7, 6, 5, 4, 3, 2, 1, 8, 9, 10
4.	6, 5, 4, 3, 2, 1, 7, 8, 9, 10
5.	5, 4, 3, 2, 1, 6, 7, 8, 9, 10
6.	4, 3, 2, 1, 5, 6, 7, 8, 9, 10
7.	3, 2, 1, 4, 5, 6, 7, 8, 9, 10
8.	2, 1, 3, 4, 5, 6, 7, 8, 9, 10
9.	1, 2, 3, 4, 5, 6, 7, 8, 9, 10

For pseudo C code, refer to **Section 5.3**.

Ex. 5.3.2 : Write pseudo C code to sort a list of integers using bubble sort. Show output of each pass for the following list : 10, 5, 4, 18, 17, 1, 2.

Soln. :

Pass – I, i = 1 :

j = 0	10	5	4	18	17	1	2
j = 1	5	10	4	18	17	1	2
j = 2	5	4	10	18	17	1	2
j = 3	5	4	10	18	17	1	2
j = 4	5	4	10	17	18	1	2
j = 5	5	4	10	17	1	18	2
	5	4	10	17	1	2	18

Pass – II, i = 2

j = 0	5	4	10	17	1	2	18
j = 1	4	5	10	17	1	2	18
j = 2	4	5	10	17	1	2	18
j = 3	4	5	10	17	1	2	18
j = 4	4	5	10	1	17	2	18
	4	5	10	1	2	17	18

Pass – III, i = 3

j = 0	4	5	10	1	2	17	18
j = 1	4	5	10	1	2	17	18
j = 2	4	5	10	1	2	17	18
j = 3	4	5	1	10	2	17	18
	4	5	1	2	10	17	18

Pass – IV, i = 4

j = 0	4	5	1	2	10	17	18
j = 1	4	5	1	2	10	17	18
j = 2	4	1	5	2	10	17	18
	4	1	2	5	10	17	18

Pass – V, i =5

j = 0	4	1	2	5	10	17	18
j = 1	1	4	2	5	10	17	18
	1	2	4	5	10	17	18

Technoledge Publications

Pass – VI, i = 4

j = 0	1	2	4	5	10	17	18
	1	2	4	5	10	17	18

Ex. 5.3.3 : Sort the following list in ascending order using bubble sort. Show all passes. Analyze time complexity. 9, 7, – 2, 4, 5, 3, – 6, 2, 1, 8

Soln. :

Pass 1 :

9	7	–2	4	5	3	–6	2	1	8
7	9	–2	4	5	3	–6	2	1	8
7	–2	9	4	5	3	–6	2	1	8
7	–2	4	9	5	3	–6	2	1	8
7	–2	4	5	9	3	–6	2	1	8
7	–2	4	5	3	9	–6	2	1	8
7	–2	4	5	3	–6	9	2	1	8
7	–2	4	5	3	–6	2	9	1	8
7	–2	4	5	3	–6	2	1	9	8
7	–2	4	5	3	–6	2	1	8	9

Pass 2 :

7	–2	4	5	3	–6	2	1	8	9
–2	7	4	5	3	–6	2	1	8	9
–2	4	7	5	3	–6	2	1	8	9
–2	4	5	7	3	–6	2	1	8	9
–2	4	5	3	7	–6	2	1	8	9
–2	4	5	3	–6	7	2	1	8	9
–2	4	5	3	–6	2	7	1	8	9
–2	4	5	3	–6	2	1	7	8	9
–2	4	5	3	–6	2	1	7	8	9

Pass 3 :

–2	4	5	3	–6	2	1	7	8	9
–2	4	5	3	–6	2	1	7	8	9
–2	4	5	3	–6	2	1	7	8	9
–2	4	3	5	–6	2	1	7	8	9
–2	4	3	5	–6	2	1	7	8	9
–2	4	3	–6	5	2	1	7	8	9
–2	4	3	–6	2	5	1	7	8	9
–2	4	3	–6	2	1	5	7	8	9
–2	4	3	–6	2	1	5	7	8	9

Pass 4 :

–2	4	3	–6	2	1	5	7	8	9
–2	4	3	–6	2	1	5	7	8	9
–2	3	4	–6	2	1	5	7	8	9
–2	3	–6	4	2	1	5	7	8	9
–2	3	–6	2	4	1	5	7	8	9
–2	3	–6	2	1	4	5	7	8	9
–2	3	–6	2	1	4	5	7	8	9

Pass 5 :

–2	3	–6	2	1	4	5	7	8	9
–2	3	–6	2	1	4	5	7	8	9
–2	–6	3	2	1	4	5	7	8	9
–2	–6	2	3	1	4	5	7	8	9
–2	–6	2	1	3	4	5	7	8	9
–2	–6	2	1	3	4	5	7	8	9

Pass 6 :

–2	–6	2	1	3	4	5	7	8	9
–6	–2	2	1	3	4	5	7	8	9
–6	–2	2	1	3	4	5	7	8	9
–6	–2	1	2	3	4	5	7	8	9

TechKnowledge Publications

Further passes will have no effect and the data will remain as it is.

Ex. 5.3.4 : Show the output of each pass using bubble sort to arrange the following numbers in ascending order.

90, 87, 76, 65, 43, 32, 19, 7, 0, – 17.

Soln. :

Pass-I

90 87 76 65 43 32 19 7 0 –17
87 90 76 65 43 32 19 7 0 –17
87 76 90 65 43 32 19 7 0 –17
87 76 65 90 43 32 19 7 0 –17
87 76 65 43 90 32 19 7 0 –17
⋮
87 76 65 43 32 19 7 0 –17 90

Pass-II

87 76 65 43 32 19 7 0 –17 90
76 87 65 43 32 19 7 0 –17 90
⋮
76 65 43 32 19 7 0 –17 87 90

Pass-III

76 65 43 32 19 7 0 –17 87 90
65 76 43 32 19 7 0 –17 87 90
⋮
65 43 32 19 7 0 –17 76 87 90

Pass-IV

65 43 32 19 7 0 –17 76 87 90
43 65 32 19 7 0 –17 76 87 90
⋮
43 32 19 7 0 –17 65 76 87 90

Pass-V

43 32 19 7 0 –17 65 76 87 90
32 43 19 7 0 –17 65 76 87 90
⋮
32 19 7 0 –17 43 65 76 87 90

Pass-VI

32 19 7 0 –17 43 65 76 87 90
19 32 7 0 –17 43 65 76 87 90
⋮
19 7 0 –17 32 43 65 76 87 90

Pass-VII

19 7 0 –17 32 43 65 76 87 90
7 19 0 –17 32 43 65 76 87 90
⋮
7 0 –17 19 32 43 65 76 87 90

Pass-VIII

7 0 –17 19 32 43 65 76 87 90
0 7 –17 19 32 43 65 76 87 90
⋮
0 –17 7 19 32 43 65 76 87 90

Pass-IX

0 –17 7 19 32 43 65 76 87 90
–17 0 7 19 32 43 65 76 87 90

Ex. 5.3.5 : Write a C/C++ program that counts the number of times a given int/data occurs in an array.

SPPU - Dec. 19, 6 Marks

Soln. :

Algorithm :

1. Sort the given data using bubble sort
2. Print the frequency of each element

Program :

```
#include <conio.h>
#include <stdio.h>
void bubble_sort(int [ ], int);
void main()
{
    int a[30], n, i, j, x;
    printf("\n enter number of elements :");
    scanf("%d", & n);
```

TechKnowledge Publications

```
    printf("\n enter array elements:");
    for(i = 0; i<n; i++)
    scanf("%d", &a[i]);
    bubble_sort(a, n);
    i = 0;
    while(i<n)
    {
        j = 0;          //frequency
        x = a [i];
        while (x == a[i] && i<n)
        {
            i++;
            j++;
        }
    printf("\n %d \t %d", x, j);
    }
}
void bubble_sort(int a[ ], int n]
{
    int i, j, temp;
    for(i = 1; i<n; i++)
    for(j = 0; j<n–i; j++)
        if(a[j] > a [j + 1])
        {
            temp = a[j];
            a[j] = a[j + 1];
            a[j + 1] = temp;
        }
}
```

5.4 Selection Sort

Selection sort is a very simple sorting method. In the ith pass, we select the element with lowest value among a[i], a[i + 1] … , a[n – 1] and we swap it with a[i].

As a result, after i passes (pass number 0 to i – 1) first i elements will be in sorted order. Selection sort can be described by

```
for(i = 0; i < n – 1; i ++)
    select the smallest element among
        a[i], …. , a[n –1] and swap it with a[i];
```

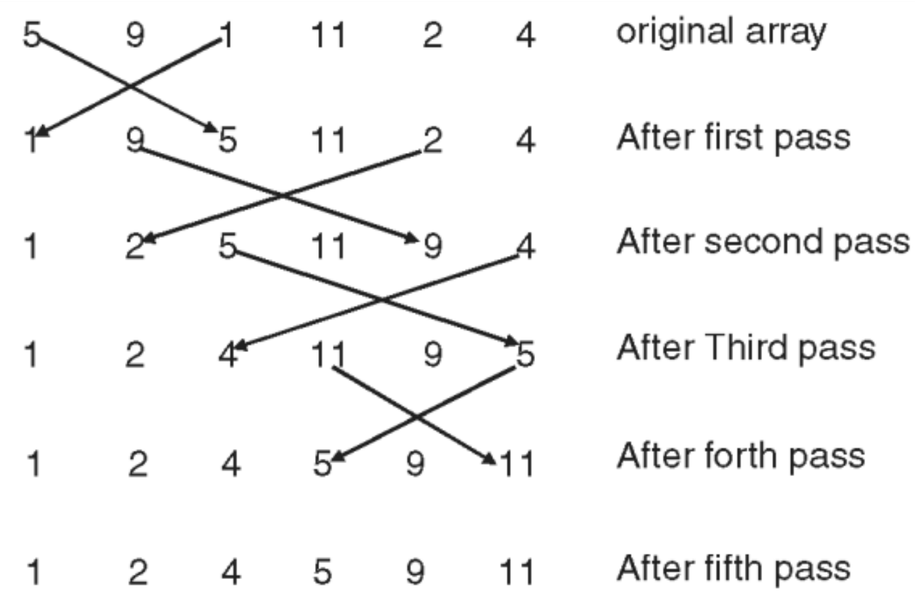


Fig. 5.4.1 : Illustration of selection sort

'C' function for selection sort.

```
void selectionsort(int a[ ], int n)
{   int i, j, k, temp;
    /*  i – outer loop
        j → inner loop
        k →index of the smallest element
        temp → for swapping */
    for(i = 0; i < n – 1; i ++)
    {
        k = i ;
        /* ith element is assumed to be the smallest */
        for(j = i + 1; j < n; j ++)
        if(a[j] < a[k])
            k = j;
        if(k ! = i)
        {
            temp = a[i];
            a[i] = a[k];
            a[k] = temp;
        }
    }
}
```

Analysis of selection sort

Selection sort is not data sensitive. In i^{th} pass, n – i comparisons will be needed to select the smallest element.

Thus, the number of comparisons needed to sort an array having n elements.

$$= (n-1) + (n-2) + \ldots + 2 + 1$$

$$= \frac{n(n-1)}{2} = \frac{1}{2}(n^2 - n) = O(n^2)$$

Techknowledge Publications

Ex. 5.4.1 : Compare the three sorting algorithms :

(a) Bubble sort (b) Selection sort

(c) Insertion sort

Soln. : A sorting algorithm can be judged on the basis of following parameters :

1. **Simplicity of algorithm :** All the three sorting algorithms are equally simple to write.
2. **Timing complexity :** Bubble sort and selection sort are not data sensitive. Both of them have a timing requirement of $O(n^2)$.

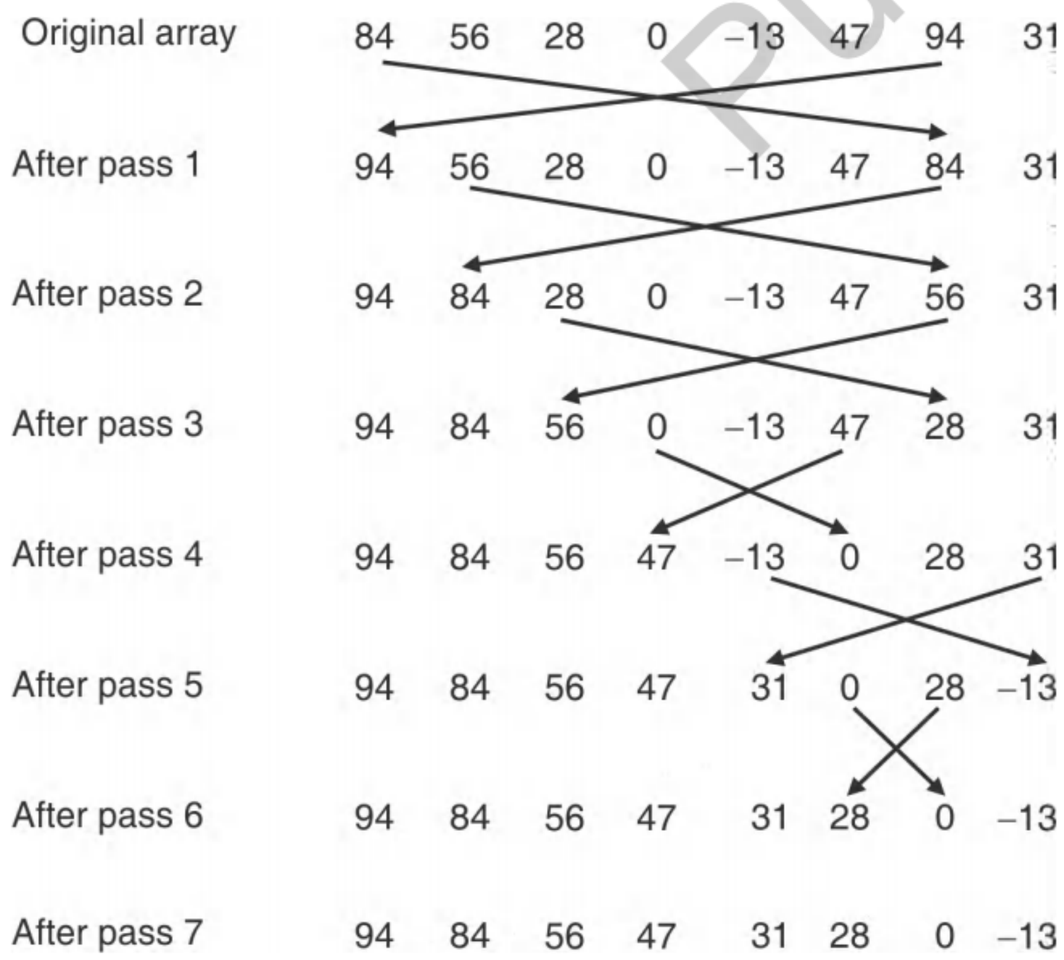
   Insertion sort is data sensitive. It works much faster if the data is partially sorted.

   Best case behaviour (when input data is sorted) = O(n).

   Worst case behaviour (when input data is in descending order) = $O(n^2)$.
3. **Sort stability :** All the three sorting algorithms are stable.
4. **Storage requirement :** No additional storage is required. All of them are in-place sorting algorithms.
5. All of them can be used for internal as well as external sorting.

Ex. 5.4.2 : Show all the passes to sort the values in descending order : 84, 56, 28, 0, –13, 47, 94, 31.

Soln. :



Ex. 5.4.3 : Compare the selection and insertion sort with respect to (i) Sort stability (ii) Efficiency (iii) Passes.

Soln. :

(i) **Sort stability :** Both selection and insertion sort are stable algorithm. Both the algorithm are based on linear scanning of the array and comparison of adjacent elements. Hence, the identical numbers will maintain their relative sequence even in the sorted array.

(ii) **Efficiency :** Loop for selection sort is given below :

```
for(i = 0; i < n–1; i++)
select the smallest element
   among a[i + 1] ...., a[n–1] and swap it with a[i];
```

In the ith pass, n–i comparisons will be needed to select the smallest element. n–1 passes are required to sort the array.

Thus, the number of comparisons needed to sort an array having n elements.

$$= (n-1) + (n-2) + \ldots + 2 + 1$$

$$= \frac{n(n-1)}{2} = \frac{1}{2}(n^2 - 2)$$

Which can be considered to be of the order of n^2 for large n.

Loop for **insertion** sort is given below :

```
for(i = 1; i<n; i++)
{
   temp = a[j];
   for(j = i – 1; j>=0 && a[j] > temp; j– –)
   a[j + 1] = a [j];
   a[j + 1] = temp;
}
```

Inner loop is data sensitive. If the input list presented for sorting is presorted then the test a[j]> temp in the inner loop will fail immediately. Thus, only one comparison will be made in each pass.

Thus the total number of comparisons (Best case)

$$= n - 1 \approx n \text{ for large } n$$

Techknowledge Publications

It the number to be sorted are initially in descending order then the inner loop will make i iterations in i^{th} pass.

∴ Total number of comparisons.

$$= 1 + 2 + 3 \ldots + (n-1)$$

$$= \frac{n(n-1)}{2} = \frac{1}{2}(n^2 - n) \approx n^2 \text{ for large n.}$$

Insertion sort is data sensitive will the best case behaviour of the order of n^2.

(iii) Passes

Both selection and insertion sort requires n – 1 passes.

Ex. 5.4.4 : Consider the following numbers sort them using "Selection sort". Show the output after each pass. 50, 20, 70, 40, 30.

Soln. :

Original data :	50	20	70	40	30
After pass 1 :	20	50	70	40	30
After pass 2 :	20	30	70	40	50
After pass 3 :	20	30	40	70	50
After pass 4 :	20	30	40	50	70

∴ Sorted Array : 20, 30, 40, 50, 70

Ex. 5.4.5 : Sort the following and show the status after every pass using selection sort : 34, 9, 78, 65, 12 –8

Soln. :

Original array :	34	9	78	65	12	–5
After pass 1 :	–5	9	78	65	12	34
After pass 2 :	–5	9	78	65	12	34
After pass 3 :	–5	9	12	65	78	34
After pass 4 :	–5	9	12	34	78	65
After pass 5 :	–5	9	12	34	65	78

Ex. 5.4.6 : Sort the following list using selection sort. Show output of each pass and write time complexity. Data : 10, 6, 13, 7, 5, 51, 27, 2, 3,15, –3, 4.

Soln. :

Pass No.	Data											
Initially	10	6	13	7	5	51	27	2	3	15	–3	4
1	–3	6	13	7	5	51	27	2	3	15	10	4
2	–3	2	13	7	5	51	27	6	3	15	10	4
3	–3	2	3	7	5	51	27	6	13	15	10	4
4	–3	2	3	4	5	51	27	6	13	15	10	7
5	–3	2	3	4	5	51	27	6	13	15	10	7
6	–3	2	3	4	5	6	27	51	13	15	10	7
7	–3	2	3	4	5	6	7	51	13	15	10	27
8	–3	2	3	4	5	6	7	10	13	15	51	27
9	–3	2	3	4	5	6	7	10	13	15	51	27
10	–3	2	3	4	5	6	7	10	13	15	51	27
11	–3	2	3	4	5	6	7	10	13	15	27	51

Time complexity : $O(n^2)$

Ex. 5.4.7 : Sort the following and show the status of every pass using selection sort : 34, 9, 78, 65, 12, – 5

Soln. :

Original array :	34	9	78	65	12	–5
After pass 1 :	–5	9	78	65	12	34
After pass 2 :	–5	9	78	65	12	34
After pass 3 :	–5	9	12	65	78	34
After pass 4 :	–5	9	12	34	78	65
After pass 5 :	–5	9	12	34	65	78

∴ Sorted Array is –5, 9, 12, 34, 65, 78

5.5 Quick Sort

Q. Explain quick sort. **SPPU - May 17, 2 Marks**

Q. Explain the algorithm of Quick sort with suitable example. **SPPU - Dec. 17, Dec. 19, 4 Marks**

Tech Knowledge Publications

Quick sort is the fastest internal sorting algorithm with the time complexity = O(n log n). The basic algorithm to sort an array a[ ] of n elements can be described recursively as follows :

1. If n < = 1, then return
2. Pick any element V in a[ ]. This is called the pivot.

   Rearrange elements of the array by moving all elements $x_i > V$ right of V and all elements $x_i \leq V$ left of V. If the place of the V after re-arrangement is j, all elements with value less than V, appear in a[0], a[1] … a[j – 1] and all those with value greater than V appear in a[j + 1] … a[n – 1].
3. Apply quick sort recursively to a[0] ….. a[j – 1] and to a[j + 1] …. a[n – 1]

Entire array will thus be sorted by as selecting an element V.

(a) Partitioning the array around V.

(b) Recursively, sorting the left partition.

(c) Recursively sorting the right partition.

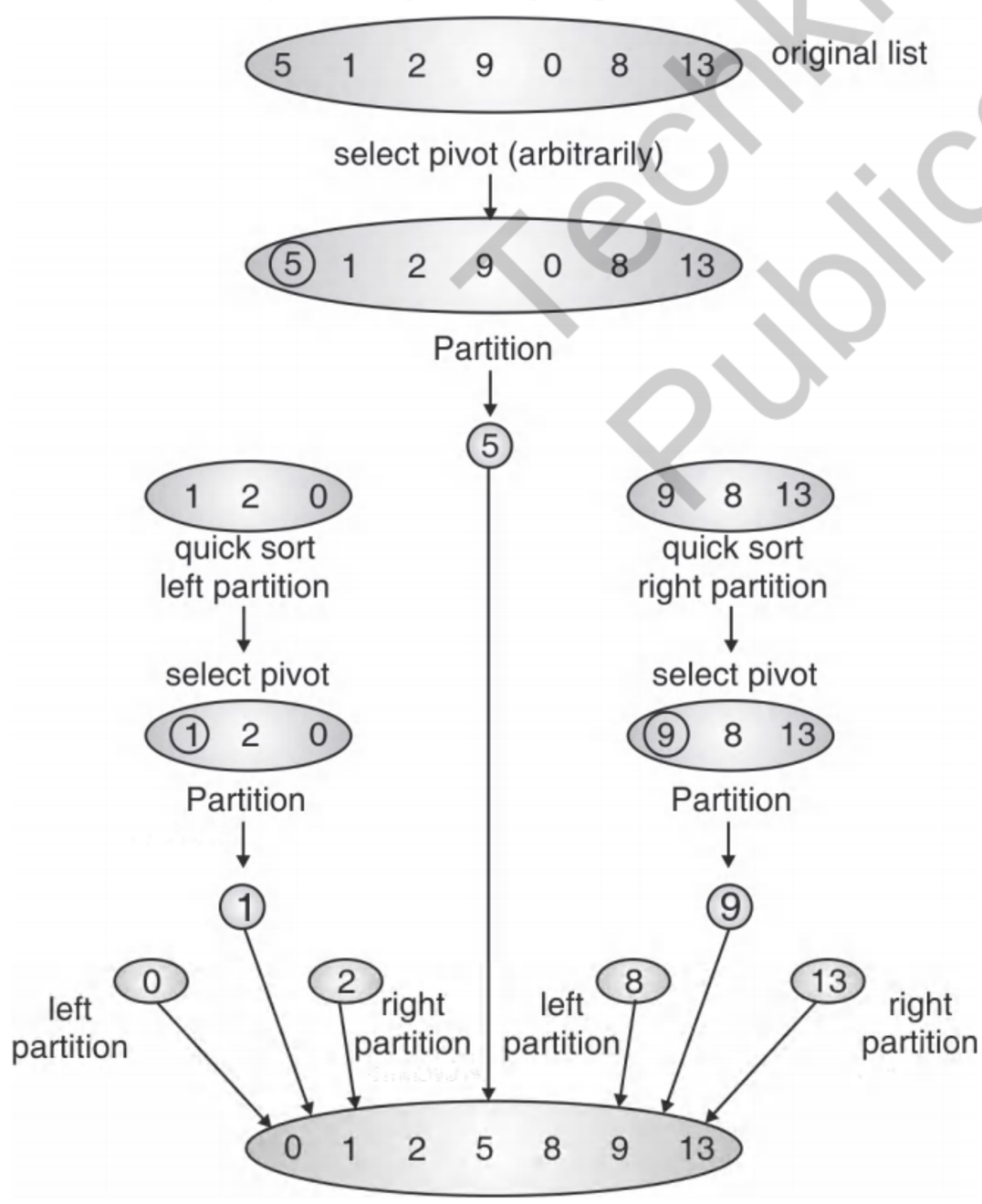


Fig. 5.5.1 : Illustration of quick sort

5.5.1 Picking a Pivot

We will choose the popular choice of the first element as the pivot. There are several strategies for pivoting. These strategies will be discussed later in the chapter.

Choose the first element as the pivot

5.5.2 Partitioning

Let the elements of the array a[*l*], a[*l* + 1] …. a[n – 1], a[n] are to be partitioned around the pivot V = a[*l*]. Initial value of *l* will be 0 and that of n will be n – 1. As recursion proceeds, these values will change. Index of the pivot (say K) will satisfy the following conditions

$$K \geq l \qquad\qquad K \leq n$$

Two cursors (index variable) are initialized with i = *l* + 1 and j = n – 1.

i moves towards right searching for an element greater than V.

j moves towards left searching for an element smaller than V.

When these elements are found they are interchanged. Again i moves towards right and j towards left and exchange is made whenever necessary. The process ends when i is equal to j or i > j.

j gives the index of the pivot element after termination of the above process.

Above procedure is explained with the help of the following example.

Let the array of number be,

0	1	2	3	4	5	6	7	8	9	10	11	12
30	35	10	15	20	34	5	18	6	11	13	26	38

Initially *l* = 0, j = 12, V = a[*l*] {i.e. V = 30}

i is set to *l* + 1 and j is set to n – 1 as shown below.

Technknowledge Publication

0	1	2	3	4	5	6	7	8	9	10	11	12
30	35	10	15	20	34	5	18	6	11	13	26	38

i = 1 j = 12

Now i moves right, while a[i] < 30. Hence i does not move further to right as a[1] > 30.

Then, j moves left, while a[j] > 30.

0	1	2	3	4	5	6	7	8	9	10	11	12
30	35	10	15	20	34	5	18	6	11	13	26	38

i=1 interchange j=11

At this point a[i] and a[j] are interchanged and the movement of i and j resumes.

0	1	2	3	4	5	6	7	8	9	10	11	12
30	26	10	15	20	34	5	18	6	11	13	35	38

i=5 j=10

i moves right to a[5] as 34 > 30. j moves left to a[10] as 13 < 30. Once again a[i] and a[j] are swapped.

0	1	2	3	4	5	6	7	8	9	10	11	12
30	26	10	15	20	13	5	18	6	11	34	35	38

j=9 i=10

Now, i moves to a[10] as 34 > 30 and j moves to a[9] as 11 < 30.

Since, i > j, the process ends. j gives the location of the pivot element. Pivot element a[*l* with *l* = 0] is interchanged with a[i] to partition the array.

0	1	2	3	4	5	6	7	8	9	10	11	12
11	26	10	15	20	13	5	18	6	30	34	35	38

Left partition Right partition

Pivot

Please note that all element a[i] with $l \le i < j$ are less than 30 and all elements a[j] with $u \ge i > j$ are greater than 30.

'C' function for the above partitioning algorithm.

```
int partition(int a[ ], int l, int u)
{
    int v, i, j, temp;
    v = a[l];
    i = l;
    j = u+1;
    do
    {
        do
        i++;
        while(a[i]<v && i< = u);
        do
            j– –;
        while(v<a[j]);
        if(i<j)
        {
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
        }
    }while(i<j);
    a[l] = a[j];
    a[j] = v;
    return(j);
}
```

'C' function for sorting an array of element using quick sort.

```
void quick_sort(int a[ ], int l, int u)
{
    int j;
    if(l<u)
    {
        j = partition(a, l, u);
        quick_sort(a, l, j–1);
        quick_sort(a, j+1, u);
    }
}
```

Tech Knowledge Publication

Program 5.5.1 : Write a program in 'C' to implement quick sort.

```
#include<conio.h>
#include<stdio.h>
void quick_sort(int[ ], int, int);
int partition(int[ ], int, int);
void main()
{
    int a[30], n, i;
    printf("\n Enter no. of elements :");
    scanf("%d", &n);
    printf("\n Enter array elements :");
    for(i = 0; i<n; i++)
        scanf("%d", &a[i]);
        quick_sort(a, 0, n–1);
        printf("\n Sorted array is :");
    for(i = 0; i<n; i++)
        printf("%d", a[i]);
    getch();
}
void quick_sort(int a[ ], int l, int u)
{
    int j;
    if(l<u)
    {
        j = partition(a, l, u);
        quick_sort(a, l, j–1);
        quick_sort(a, j+1, u);
    }
}
int partition(int a[ ], int l, int u)
{
    int v, i, j, temp;
    v = a[l];
    i = l;
    j = u+1;
    do
    {
        do
            i++;
        while(a[i]<v && i< = u);
        do
            j– –;
        while(v<a[j]);
        if(i<j)
        {
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
        }
    }while(i<j);
    a[l] = a[j];
    a[j] = v;
    return(j);
}
```

Output

```
Enter no. of elements    : 4
Enter array elements     : 23   1    55    33
Sorted array is          : 1    23   33    55
```

Techknowledge Publication

Ex. 5.5.1 : Here are sixteen integers : 22, 36, 6, 7, 9, 26, 45, 75, 13, 31,62, 27, 76, 33, 16, 62, 49. Sort them using quick sort.

Soln. :

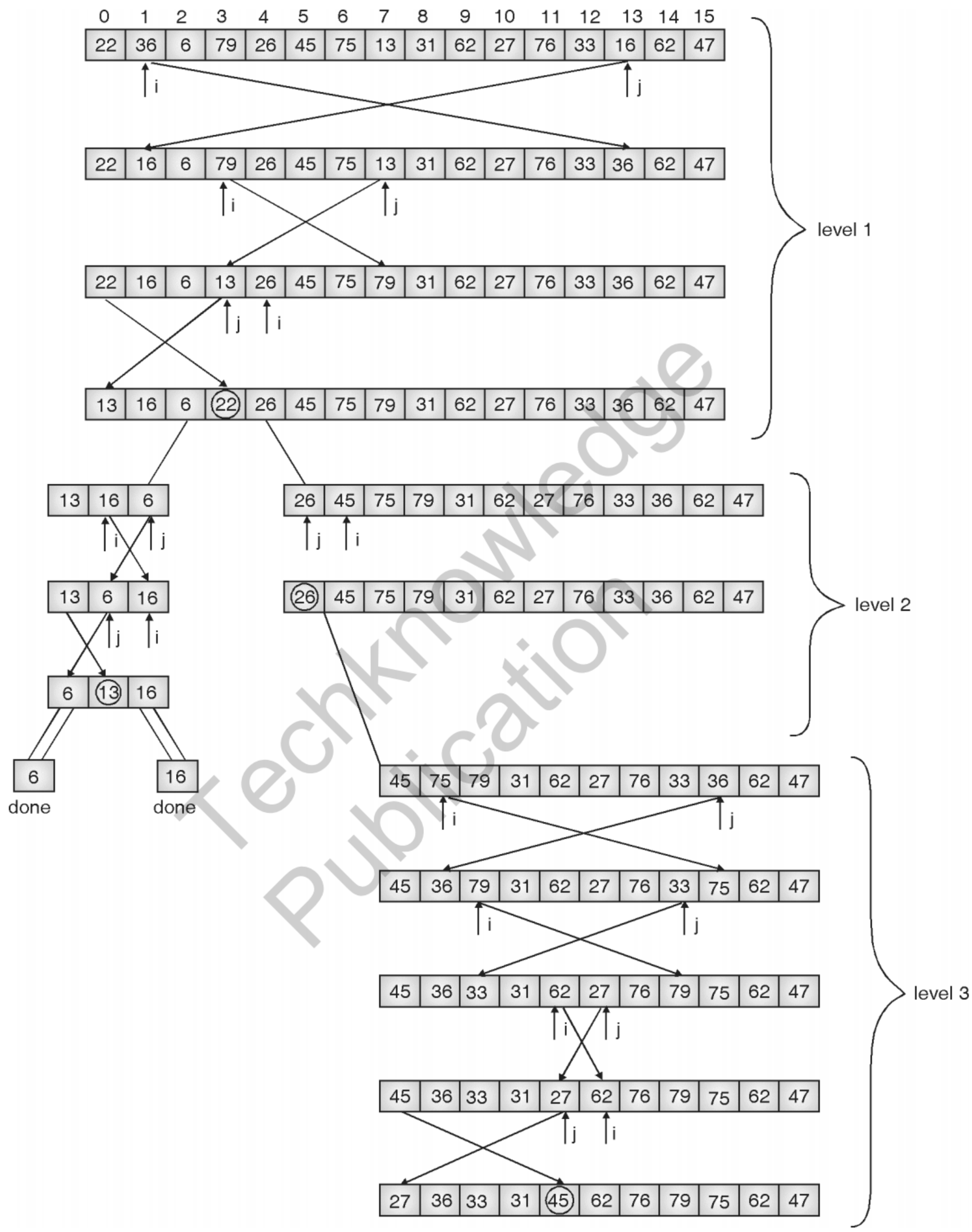


Fig. Ex. 5.5.1 contd..

Techknowledge Publications

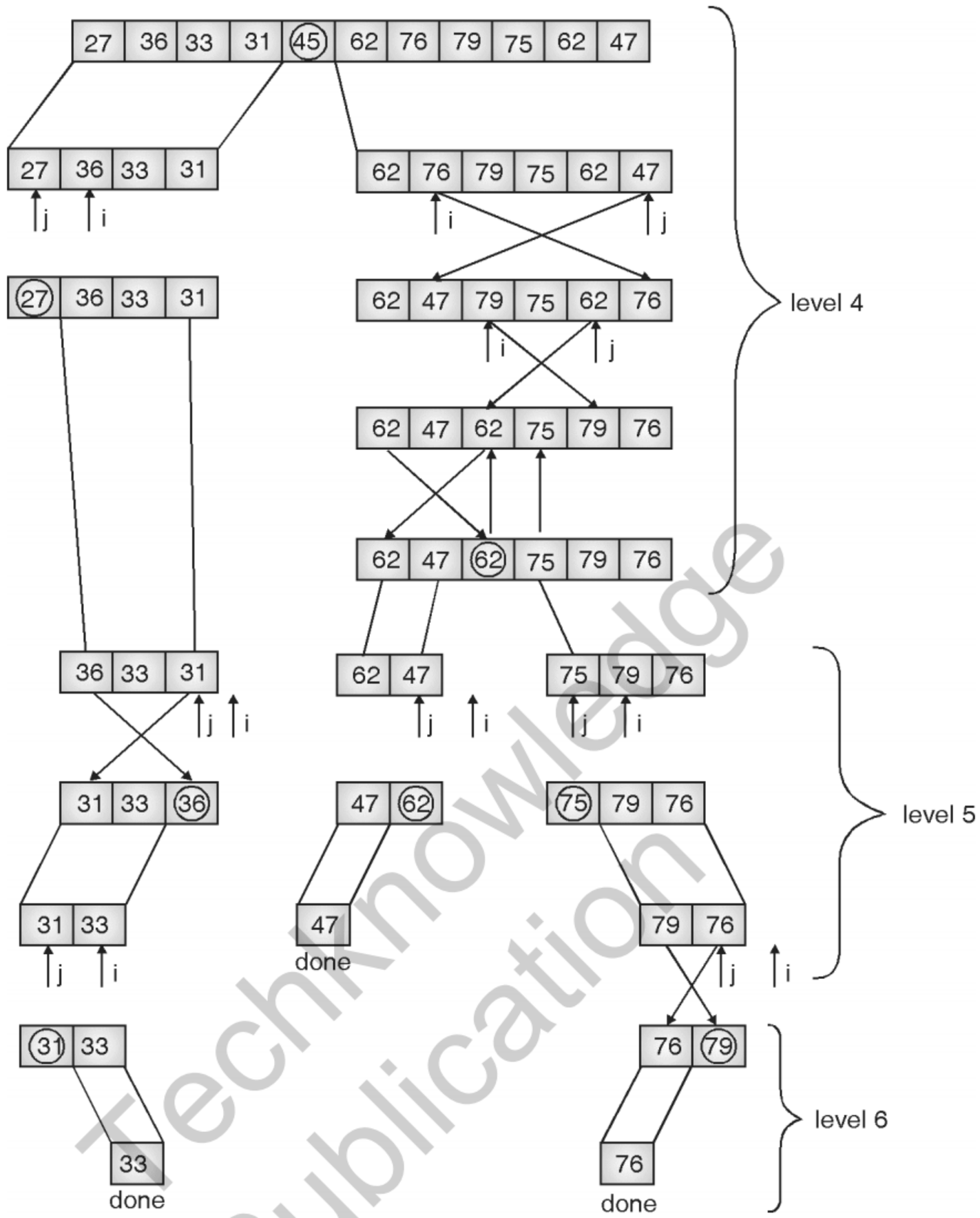


Fig. Ex. 5.5.1

Note : Partition point at each level is encircled.

Ex. 5.5.2 : Consider the following set of numbers, sort them using quick sort. Clearly indicate the Pivot element and the partition at each step : 20, 54, 48, 37, 12, 92, 86, 07.

Soln. :

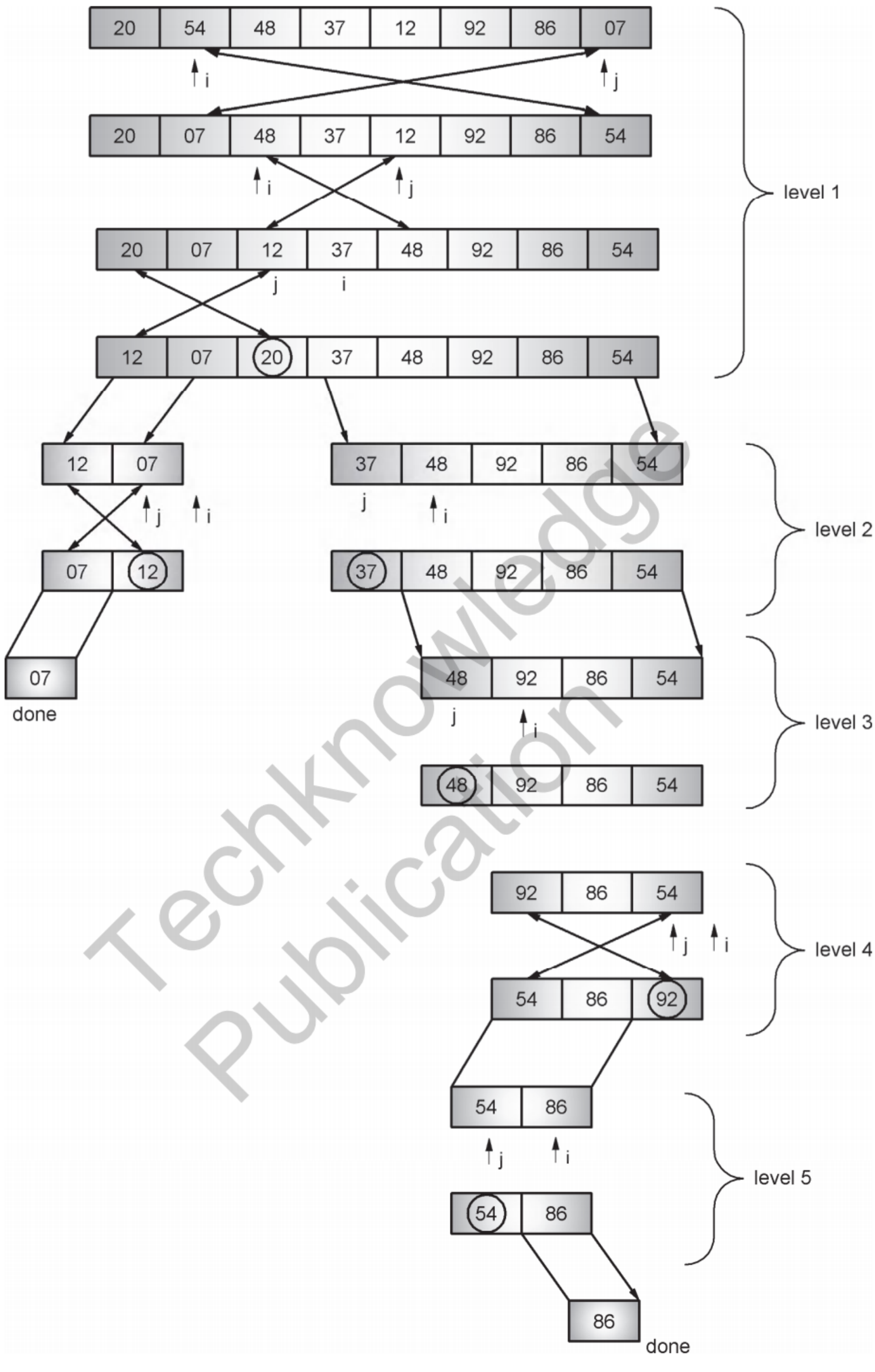


Fig. Ex. 5.5.2

Note : Partition point at each level is encircled.

Tech Knowledge Publications

Ex. 5.5.3 : Sort the following data using quick sort clearly indicate the pivot element and partition at each step. List the data after every pass. 56, 12, 84, 56, 28, 0, – 13, 47, 94, 31, 12, – 2.

Soln. :

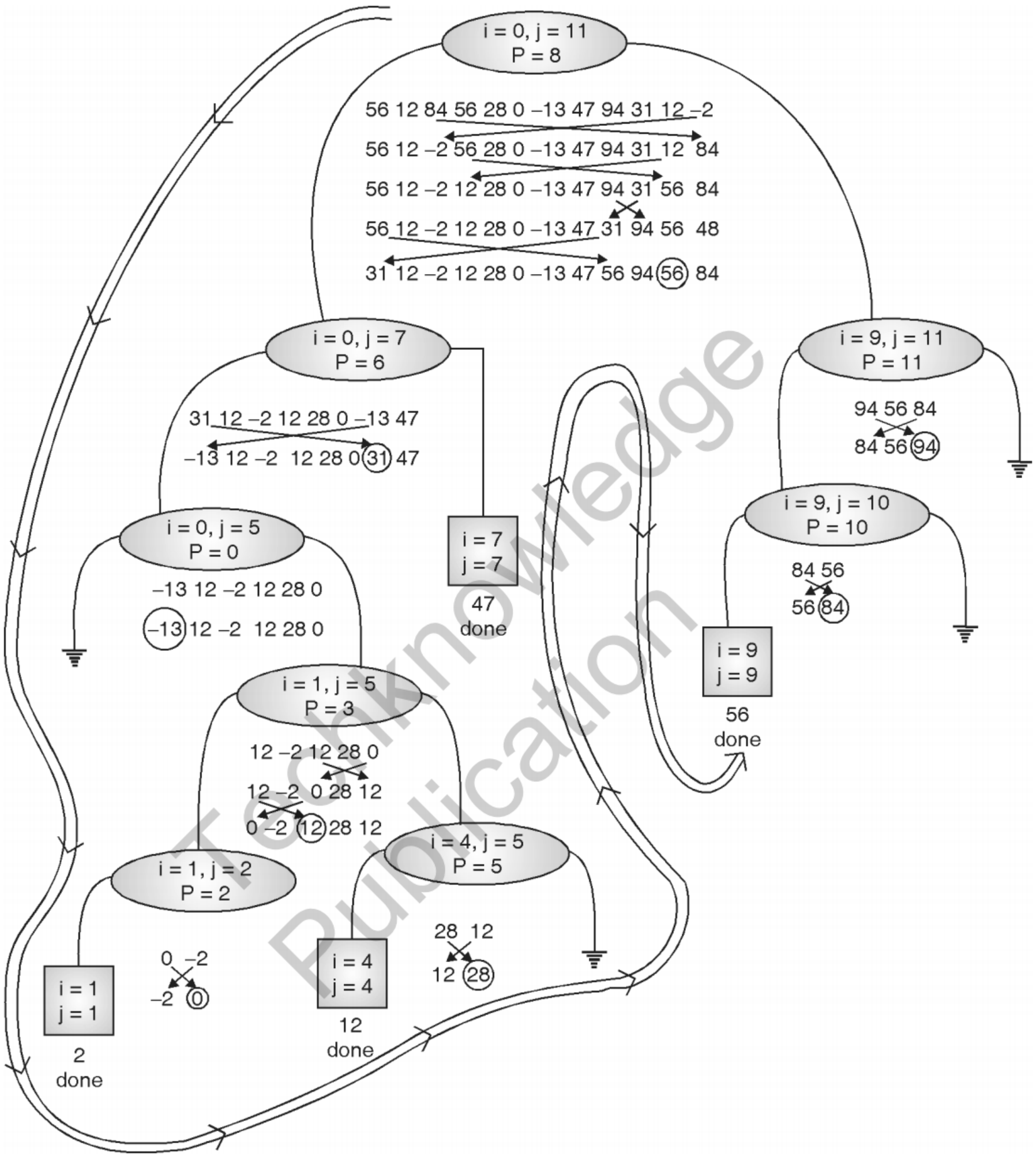


Fig. Ex. 5.5.3

i = Index of the first data

j = Index of the last data

P = Partition point (partition element is encircled)

Data at the end of every pass :

To write the data at the end of every pass, follow the path are given as follows :

Initial data :	56	12	84	56	28	0	– 13	47	94	31	12	– 2
After pass 1 :	31	12	– 2	12	28	0	– 13	47	56	94	56	84
After pass 2 :	– 13	12	– 2	12	28	0	31	47	56	94	56	84
After pass 3 :	– 13	12	– 2	12	28	0	31	47	56	94	56	84
After pass 4 :	– 13	0	– 2	12	28	12	31	47	56	94	56	84
After pass 5 :	– 13	– 2	0	12	28	12	31	47	56	94	56	84
After pass 6 :	– 13	– 2	0	12	12	28	31	47	56	94	56	84
After pass 7 :	– 13	– 2	0	12	12	28	31	47	56	84	56	94
After pass 8 :	– 13	– 2	0	12	12	28	31	47	56	56	84	94

Ex. 5.5.4 : Consider the following set of numbers. Sort them using quick sort. Clearly indicate the pivot element and partition at each step. List the data after every pass – 20, 54, 48, 37, 12, 92, 86, 07.

Soln. :

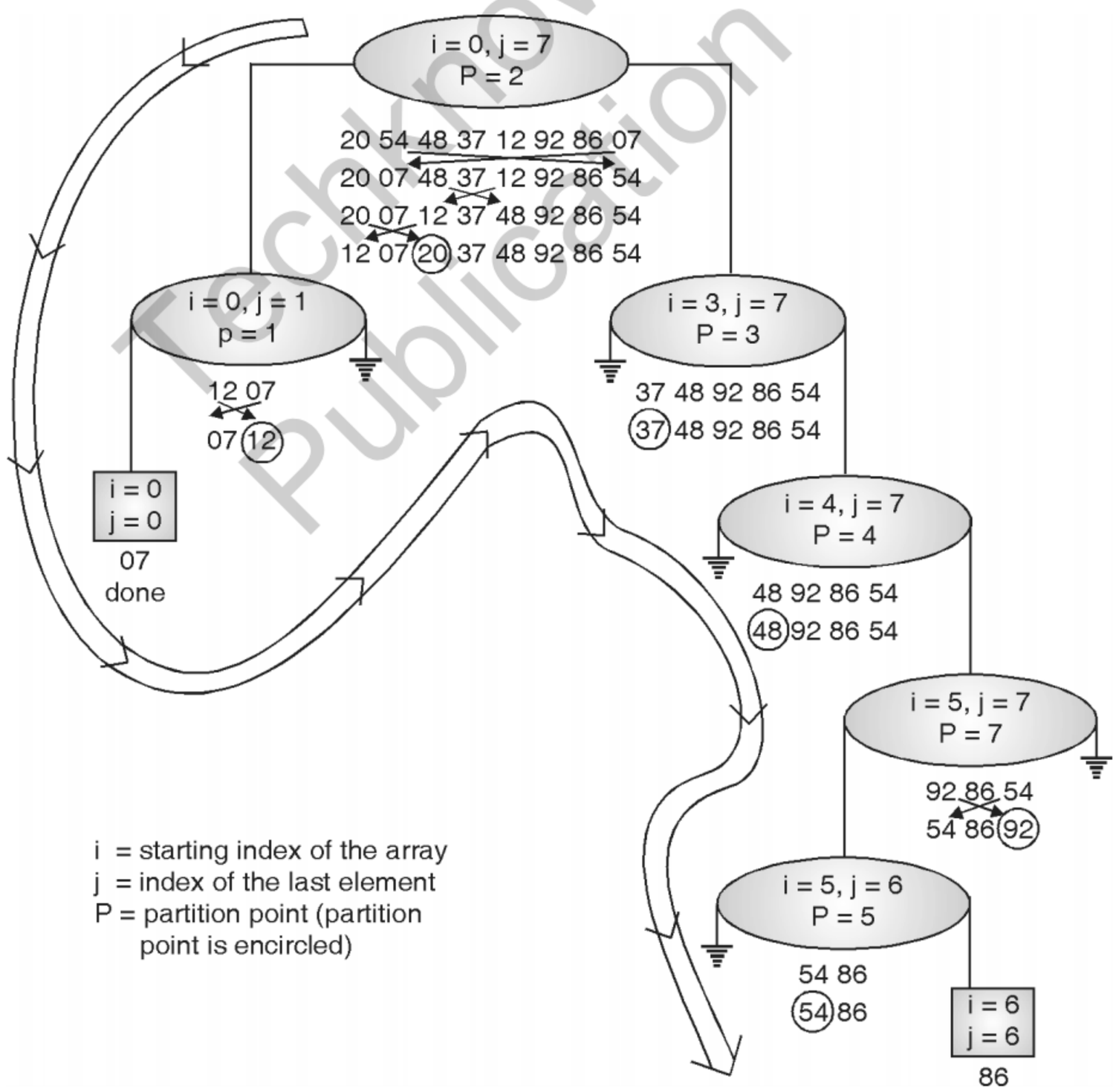


Fig. Ex. 5.5.4

Tech Knowledge Publications

i = Starting index of the array.

j = Index of the last element.

P = Partition point (partition point is encircled)

Data at the end of every pass :

To write the data at the end of every pass, follow the path are given as follows :

Initial data :	20	54	48	37	12	92	86	07
After pass 1 :	12	07	(20)	37	48	92	86	54
After pass 2 :	[07]	(12)	(20)	37	48	92	86	54
After pass 3 :	[07]	(12)	(20)	(37)	48	92	86	54
After pass 4 :	[07]	(12)	(20)	(37)	(48)	92	86	54
After pass 5 :	[07]	(12)	(20)	(37)	(48)	54	86	(92)
After pass 6 :	[07]	(12)	(20)	(37)	(48)	(54)	[86]	(92)

Ex. 5.5.5 : Sort the following list of numbers using Quick sort 25, 29, 36, 32, 38, 44, 54, 40. Show the result stepwise.

Soln. :

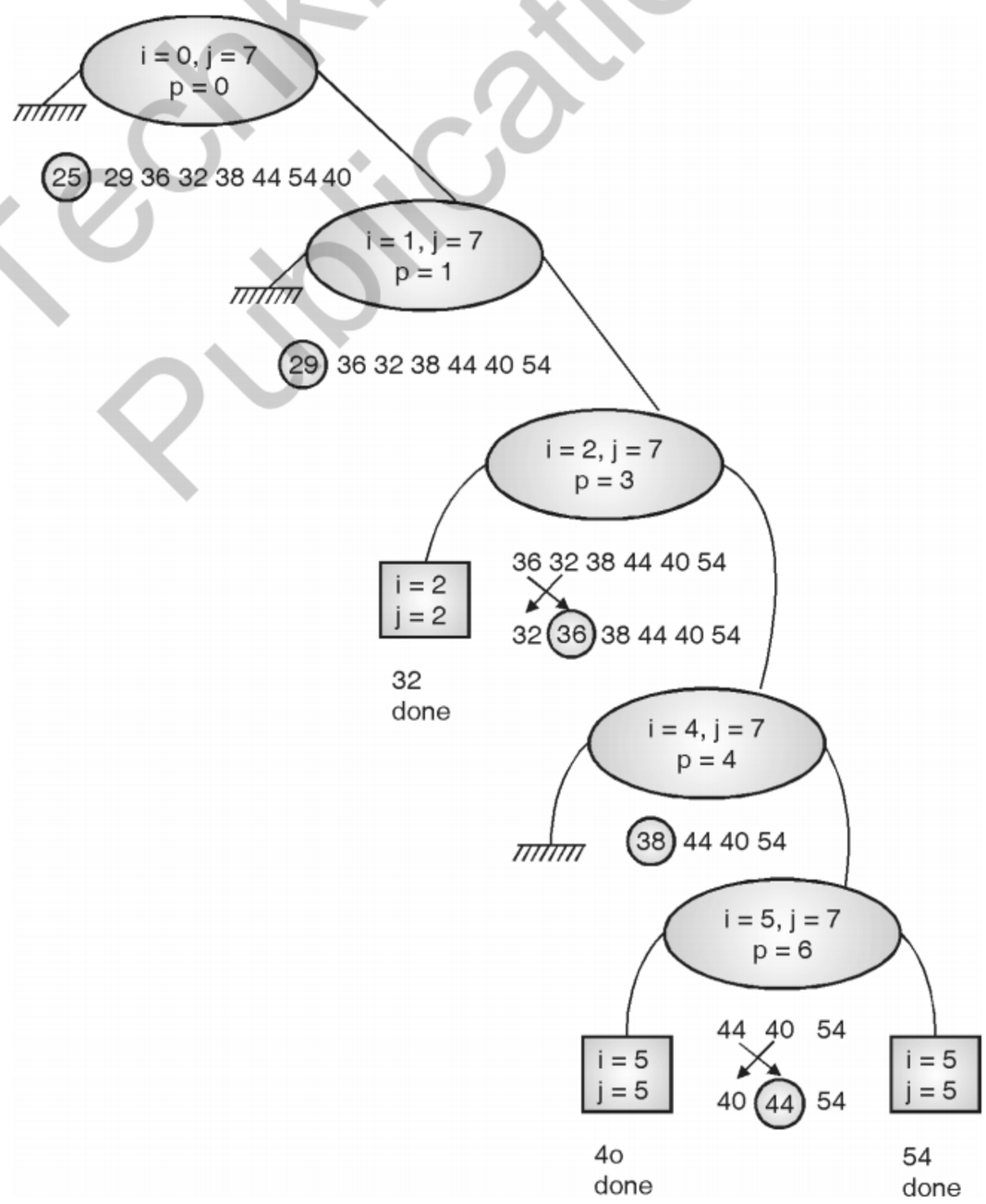


Fig. Ex. 5.5.5

TechKnowledge Publications

Data after each pass

Pass	Data							
1	25	29	36	32	38	44	40	54
2	25	29	36	32	38	44	40	54
3	25	29	32	36	38	44	40	54
4	25	29	32	36	38	44	40	54
5	25	29	32	36	38	40	44	54

Ex. 5.5.6 : By using Quick sort, sort the following numbers in non-decreasing order. Show the contents by array after each pass. 45, 12, 5, 78, 19, 2, 56, 1, 62.

Soln. :

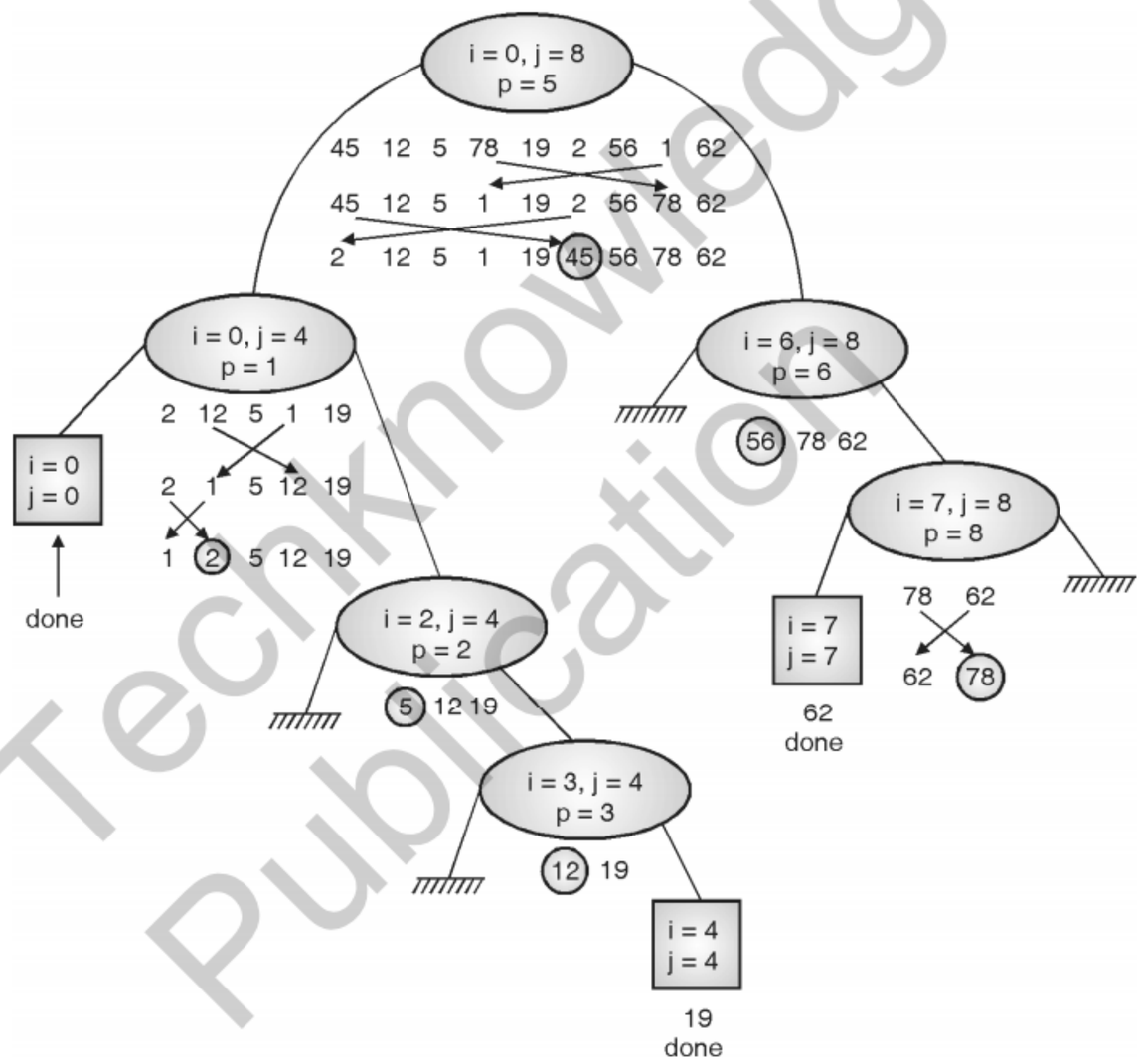


Fig. Ex. 5.5.6

Data after each pass

Pass	Data								
1	2	12	5	1	19	45	56	78	62
2	1	2	5	12	19	45	56	78	62
3	1	2	5	12	19	45	56	78	62
4	1	2	5	12	19	45	56	78	62
5	1	2	5	12	19	45	56	78	62
6	1	2	5	12	19	45	56	62	78

Techknowledge Publication

Ex. 5.5.7 : Write pseudo 'c' code to sort 'n' numbers stored in an array in non-increasing order using Quick sort. State its time and space complexity. Apply your algorithm to sort the following numbers in non-increasing order. Display the contents of the array in each pass.
23, 9, 100, 17, 45, 57, 79, 45.

Soln. :

```
void quick_sort(int a [ ], int l, int u)
{   int j ;
    if(l < u)
    {
        j = partition(a, l, u) ;
        quick_sort(a, l, j – 1) ;
        quick_sort(a, j + 1, u) ;
    }
}
int partition(int a [ ], int l, int u)
{   int v, i, j, temp ;
    i = u ;
    j = l – 1 ;
    do
    {
        do
        i – – ;
        while(a [i] > v && i > = u) ;
        do
            j + + ;
        while(v > a [j]) ;
        if(i > j)
        {
            temp = a [i] ;
            a [i] = a [j] ;
            a [j] = temp ;
        }
    }while(i > j) ;
    a [u] = a [j] ;
    a [j] = v ;
    return(j) ;
}
```

Time complexity = O (n log n)

Space complexity

(i) Space required to store n numbers, which is of the order of n.

(ii) Space required on top of the stack to handle recursion. This could vary from of 'the order of log n' to 'the order of n'.

(iii) Space required to store other local variables, which is of constant order.

Sorting of data in non-increasing order.

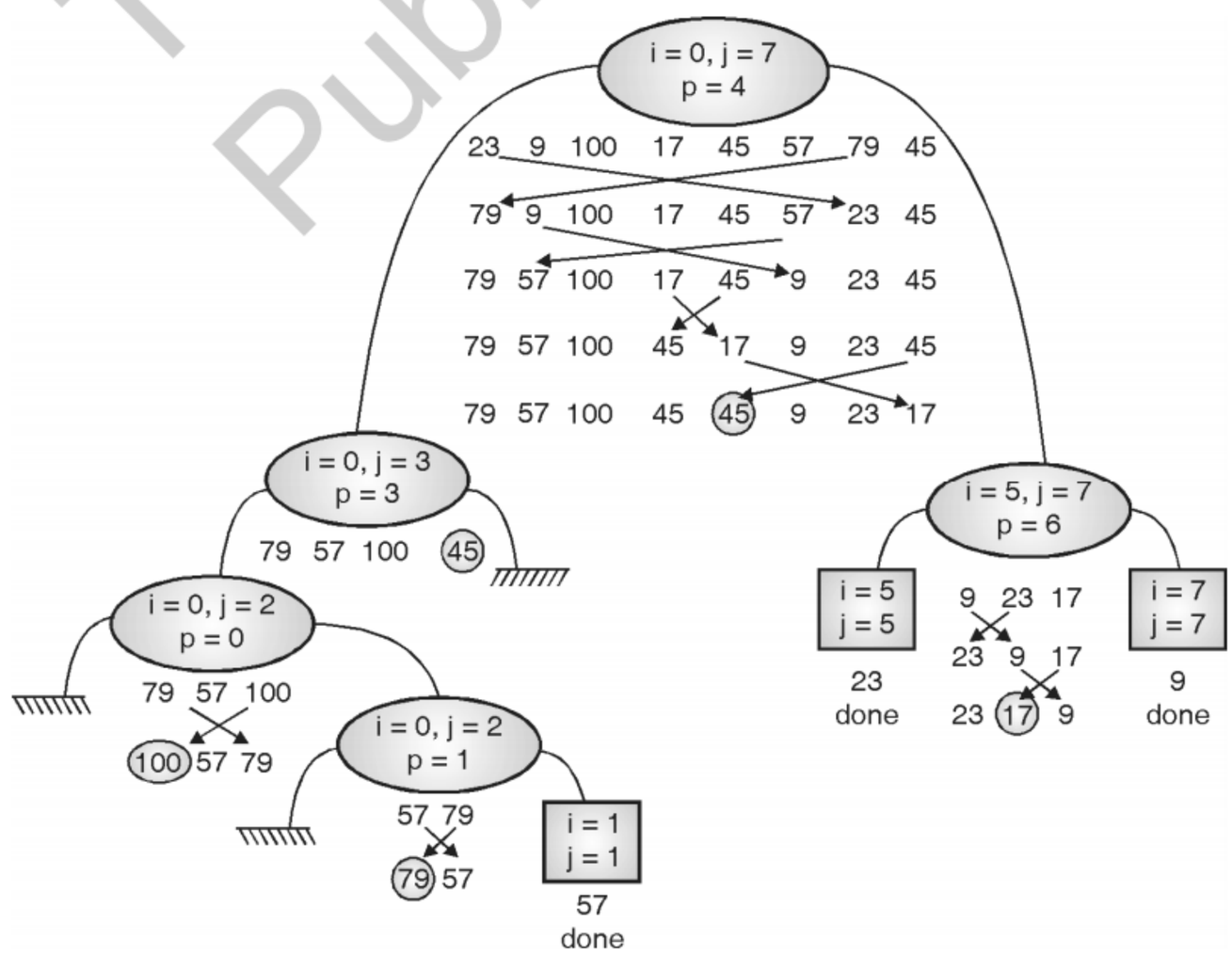


Fig. Ex. 5.5.7

Data after every pass

Pass	Data							
1	79	57	100	45	45	9	23	17
2	79	57	100	45	45	9	23	17
3	100	57	79	45	45	9	23	17
4	100	79	57	45	45	9	23	17
5	100	79	57	45	45	23	17	9

Ex. 5.5.8 : Sort using quick sort the following and show all passes : 34, 9, 78, 65, 12, – 8.

Soln. :

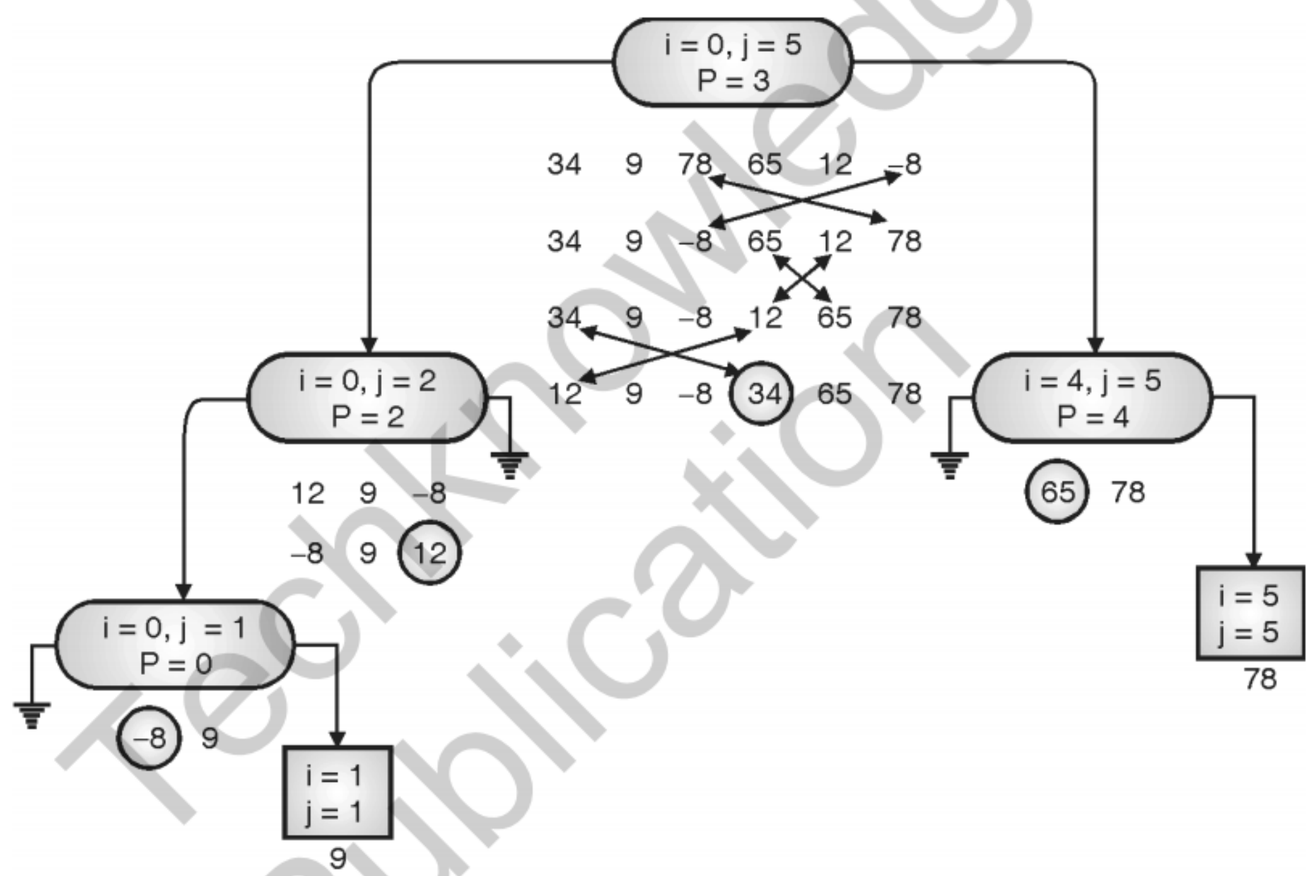


Fig. Ex. 5.5.8

Data after various passes

Initial Data	34	9	78	65	12	– 8
After Pass 1	12	9	– 8	34	65	78
After pass 2	– 8	9	12	34	65	78
After pass 3	– 8	9	12	34	65	78
After pass 4	– 8	9	12	34	65	78

Ex. 5.5.9 : Sort the following data to ascending order using Quick sort. Show all passes with pivot : 17, 8, –9, 2, 0, –5, 7, 20, 11, 15.

Soln. :

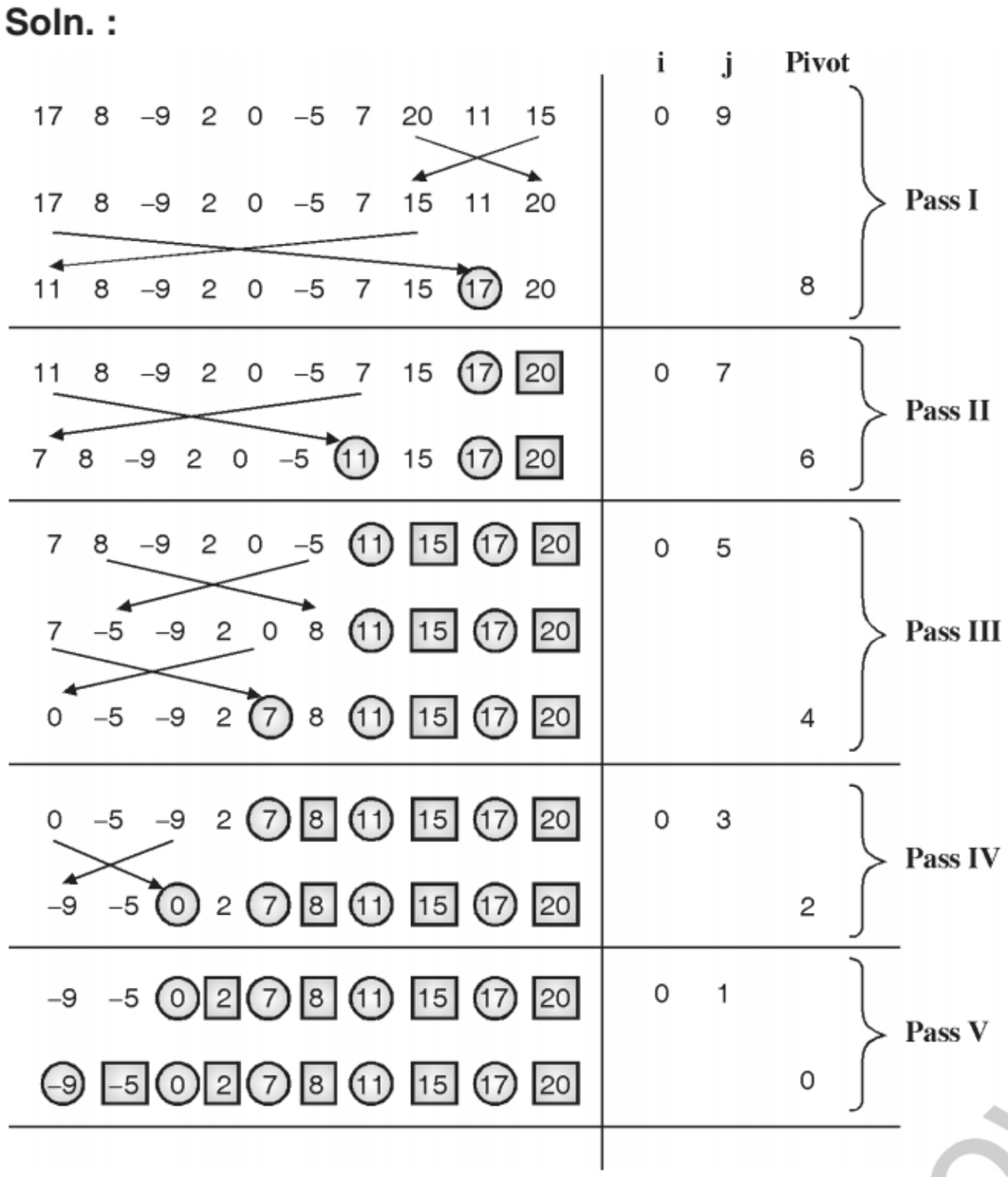


Fig. Ex. 5.5.9

Ex. 5.5.10 : Explain Quick Sort and sort the given list using quick sort : 15, 08, 20, –4, 16, 02, 01, 12, 21, –2.

SPPU - Dec. 16, 6 Marks

Soln. : Sorting of given numbers using quick sort

Initial data : 15, 08, 20, –4, 16, 02, 01, 12, 21, –2

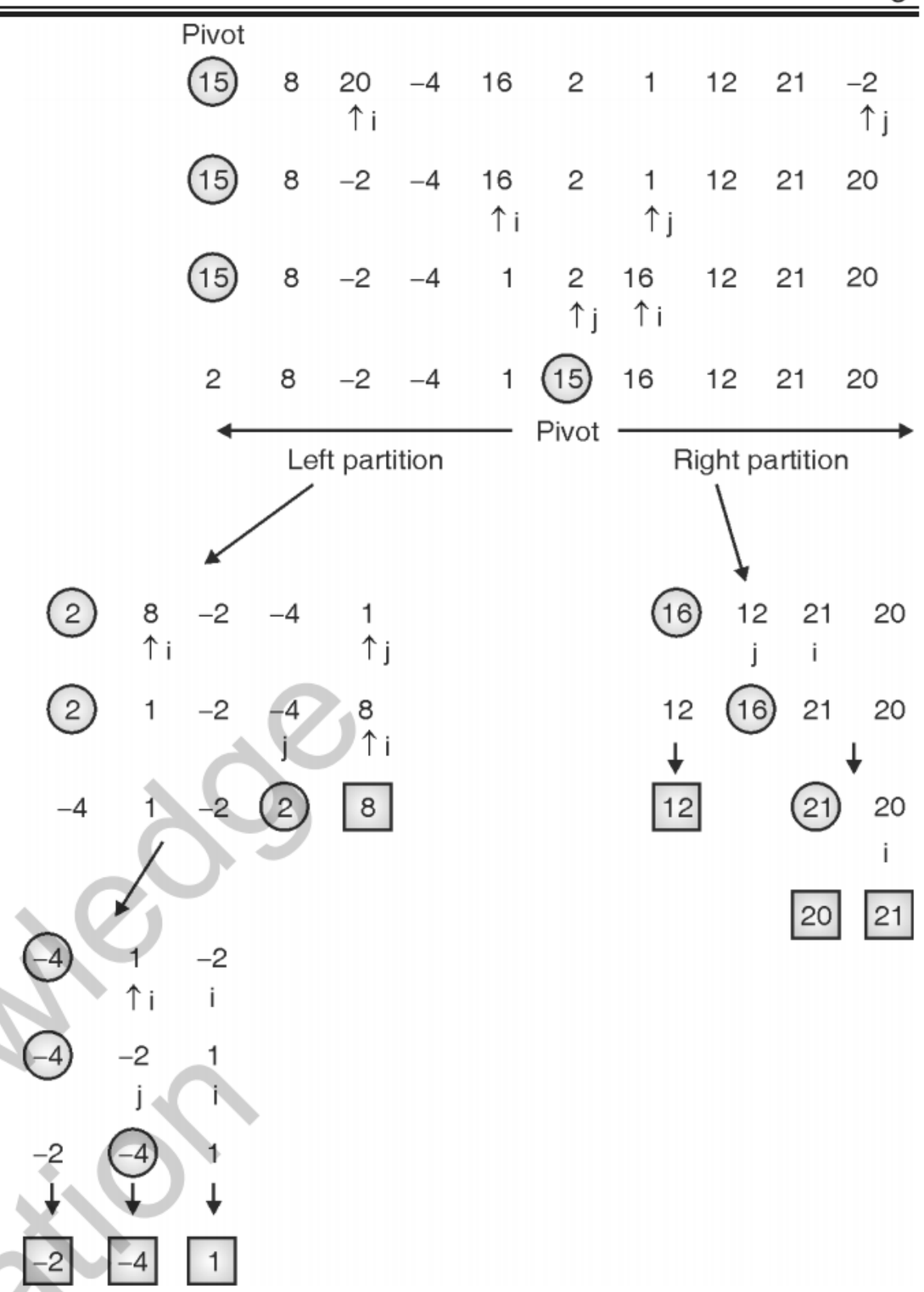


Fig. Ex. 5.5.10

Sorted Numbers are – 2, – 4, 1, 2, 8, 12, 15, 16, 20, 21

Ex. 5.5.11 : Sort the given list using quick sort : 39, 09, 81, 45, 90, 27, 72, 18.

SPPU - May 17, 4 Marks

Soln. :

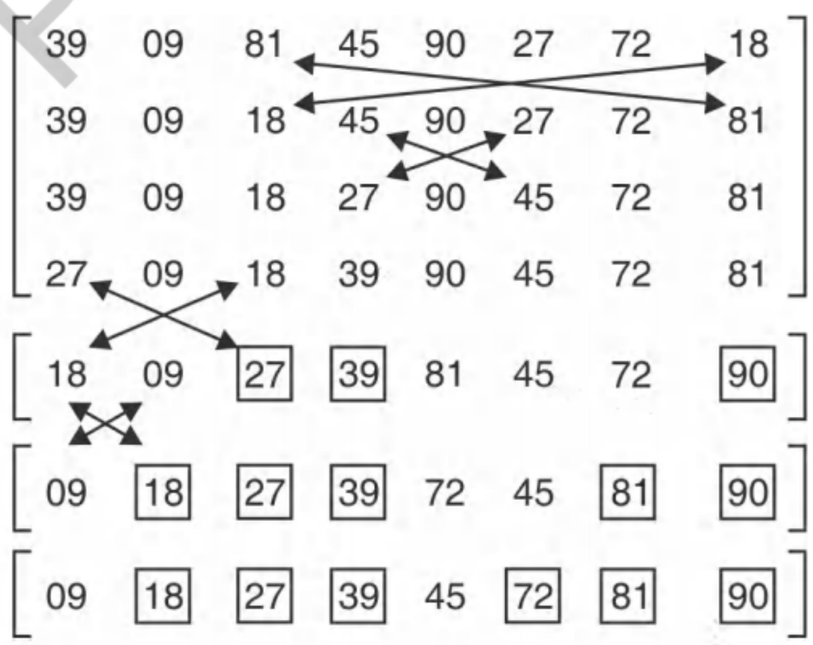


Fig. Ex. 5.5.11

Sorted list : 09, 18, 27, 39, 45, 72, 81, 90

TechKnowledge Publications

Ex. 5.5.12 : Sort the following numbers using quick sort : 25, 82, 17, 23, 38, 7, 64, 86, 21. **SPPU - Dec. 18, 5 Marks**

Soln. :

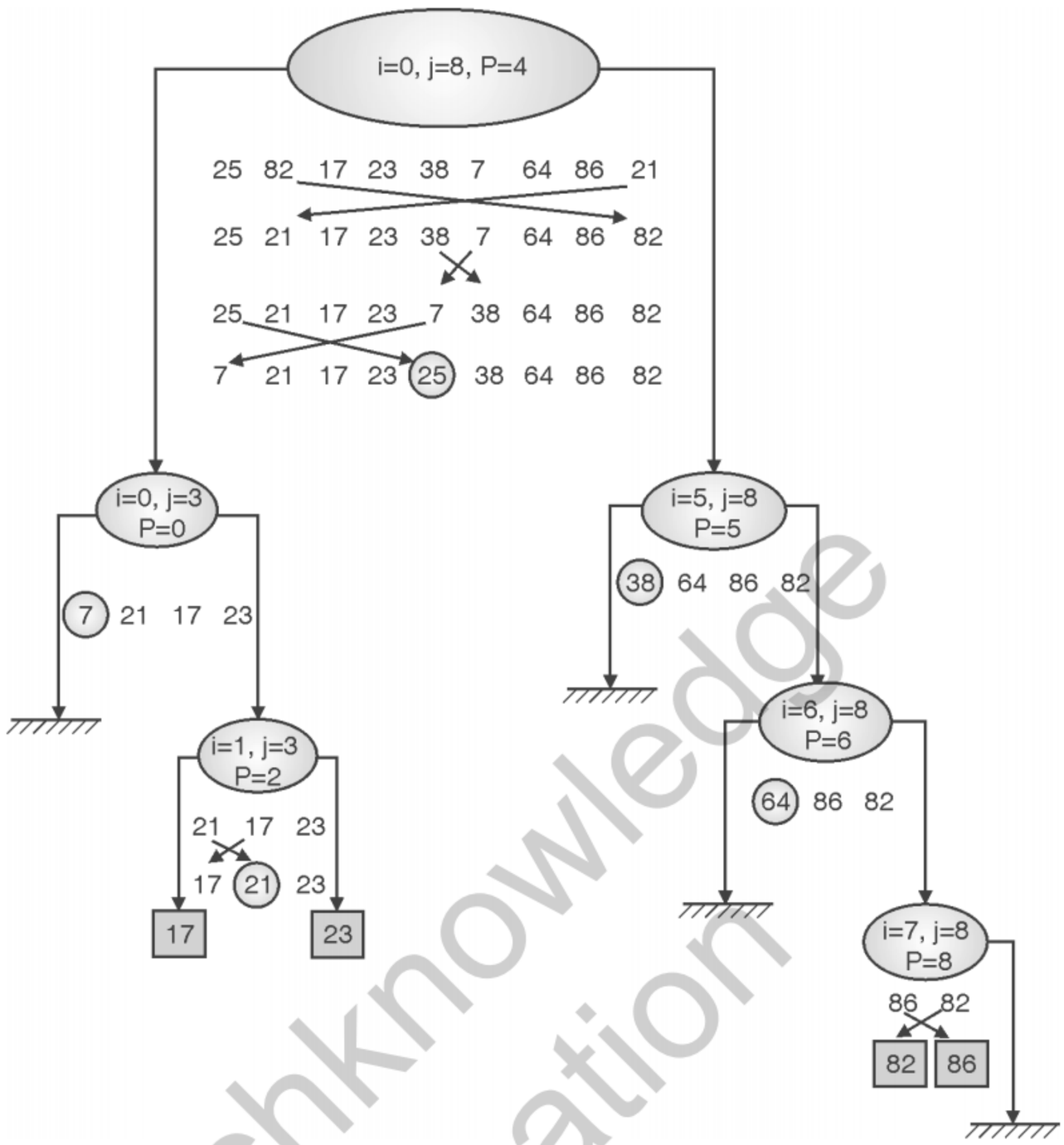


Fig. Ex. 5.5.12

Sorted List : 7, 17, 21, 23, 25, 38, 64, 80, 86

Space Complexity : O(n)

5.5.3 Running Time of Quick Sort

Q. Discuss quick sort time complexity and space complexity. **SPPU - Dec. 17, Dec. 18, Dec. 19, 2 Marks**

Quick sort takes O(n log n) time on the average to sort n elements and $O(n^2)$ time in worst case. Quick sort is a recursive algorithm. Its timing behaviour can be expressed using a recurrence formula. Solution of the recurrence formula will give its timing complexity. Since, the recursion terminates when N = 0 or 1.

We can take,

$$T(0) = 1 \qquad T(1) = 1$$

Running time of quick sort = running time of the left partition + running time of the right partition + time spent in partitioning the elements.

Since, partitioning requires scanning of the array linearly. It can be taken as CN, where C is a constant.

Running Time of the left partition = T(j), left partition contains j elements.

Running Time of the right partition = T(N – j – 1)

$$\therefore T(N) = T(j) + T(N - j - 1) + CN$$

Space complexity : Quick sort is an inplace algorithm but it does require additional space for recursion.

$\therefore$ Space complexity = O(n) + O(log n)

Techknowledge Publications

5.5.3(A) Worst-Case Analysis for Quick Sort to Sort List of Numbers in Ascending Order

In the worst case, either j = 0 or n – j – 1 is equal to 0. j will always be 0 if the data presented for sorting is already sorted in ascending order. n – j – 1 will always be 0 if the data presented for sorting is already sorted in descending order.

Let us take the case of j = 0

$$T(N) = T(0) + T(N-1) + CN \text{ for } N > 1 \text{ neglecting } T(0) \text{ being insignificant}$$

$$\begin{aligned} T(N) &= T(N-1) + CN \\ &= T(N-2) + C(N-1) + CN \\ &= T(N-3) + C(N-2) + C(N-1) + CN \\ &\vdots \\ &= T(1) + C \cdot 2 + C \cdot 3 + \ldots \\ &\quad + C(N-2) + C(N-1) + CN \\ &= T(1) + C \cdot \sum_{i=2}^{N} i \\ &= O(N^2) \end{aligned}$$

5.5.3(B) Best-Case Analysis

Running time will be minimum if the pivot is always found in the middle. Left partition and the right partition will have equal number of elements.

$$\begin{aligned} \therefore T(N) &= 2T(N/2) + CN = 2^2\, T(N/2^2) + 2CN \\ &= 2^3\, T(N/2^3) + 3CN \end{aligned}$$

Let us assume that $N/2^h = 1$

$$\begin{aligned} \therefore N &= 2^h \text{ or } \log_2 N = h \\ \therefore T(N) &= 2^h\, T(N/2^h) + h \cdot C \cdot N \\ &= 2^h\, T(1) + C\, h\, N \\ &= N\, T(1) + C \cdot N \cdot \log_2 N \\ &= O(N \log_2 N) \end{aligned}$$

5.5.3(C) Average-Case Analysis

Let us consider the recurrence relation,

$$T(N) = T(j) + T(N-j-1) + CN$$

Since, the variable j could take any value from 0 to O(n – 1) with equal probability,

The recurrence relation for average case can be written as,

$$T(N) = \frac{1}{N}\left[\sum_{j=0}^{N-1} T(j) + \sum_{j=0}^{N-1} T(N-j-1)\right] + CN$$

As j varies from 0 to N – 1, N – j – 1 varies from N – 1 to 0.

$$\text{Hence } \sum_{j=0}^{N-1} T(N-j-1) = \sum_{j=0}^{N-1} T(j)$$

$$\therefore T(N) = \frac{2}{N} \sum_{j=0}^{N-1} T(j) + CN \qquad \ldots(1)$$

$$\text{Or} \quad NT(N) = 2 \sum_{j=0}^{N-1} T(j) + CN^2 \qquad \ldots(2)$$

With N written as N – 1, equation (2) becomes

$$(N-1)\, T(N-1) = 2 \sum_{j=0}^{N-2} T(j) + C(N-1)^2 \qquad \ldots(3)$$

If we subtract Equation (3) from Equation (2), we get,

$$NT(N) - (N-1)\, T(N-1) = 2T(N-1) + 2CN - C$$

Dropping C being insignificant and arranging terms.

$$NT(N) = (N+1)\, T(N-1) + 2CN$$

Dividing throughout by N(N + 1)

We get,

$$\begin{aligned} \frac{T(N)}{N+1} &= \frac{T(N-1)}{N} + \frac{2C}{N+1} \\ &= \frac{T(N-2)}{N-1} + \frac{2C}{N} + \frac{2C}{N+1} \end{aligned}$$

Techknowledge Publications

$$= \frac{T(N-3)}{N-2} + \frac{2C}{N-1} + \frac{2C}{N} + \frac{2C}{N+1}$$

$$\vdots$$

$$= \frac{T(1)}{2} + \frac{2C}{3} + \ldots \frac{2C}{N} + \frac{2C}{N+1}$$

$$= \frac{T(1)}{2} + 2C \sum_{i=3}^{N+1} \frac{1}{i}$$

$$2C \sum_{i=3}^{N+1} \frac{1}{i} \approx \int \frac{1}{N} = \log N$$

$$\therefore \frac{T(N)}{N+1} = \frac{T(1)}{2} + \log N$$

$$\therefore T(N) = \frac{T(1)}{2}(N+1) + (N+1)\log N$$

$$= O(N \log N)$$

Note : Quick sort gives its worst behaviour when data presented for sorting is already sorted in either ascending or descending order.

Ex. 5.5.13 : Write a non recursive algorithm of quick sort recursive algorithm.

Soln. :

The recursive algorithm for quick sort is given below.

```
void quick_sort(int a[ ], int l, int u)
{
    int j;
    if(l<u)
    {
        j = partition(a, l, u); /* 1 */
        quick_sort(a, l, j–1); /* 2 */
        quick_sort(a, j+1, u); /* 3 */
    }
}
```

Above algorithm of quick_sort( ) is similar to preorder traversal of a binary tree.

Line number 1 of quick_sort( ) can be treated as visit( )

Line number 2 of quick_sort( ) can be treated as traversing of left sub tree.

Line number 3 of quicksort( ) can be treated as traversing of right sub tree.

Left child of node (a, *l*, u) is (a, *l*, j – 1) and the right child of node (a, *l*, u) is (a, j +1, u).

A stack can be used to remember the current node as we traverse down the tree using the left child. Stack helps in traversing right sub tree of all nodes, visited so far.

'C' function for non-recursive quick sort.

```
void quick_sort(int a[ ], int l, int u)
{
    stack s;
    int i, j;
    init(&s);
    while(i<u)
    {
        j = partition(a, l, u);
        push(&s, j+1);
        push(&s, u);
        u = j–1; /* goto */
    }
    while(!empty(&s))
    {
        l = pop(&s); /* goto right */
        u = pop(&s);
        while(l<u)
        {
            j = partition(a, l, u);
            push(&s, j+1);
            push(&s, u);
            u = j–1;
        }
    }
}
```

Program 5.5.2 : Program for non-recursive quick sort.

```
#include<stdio.h>
# define MAX 20
#include<conio.h>
typedef struct stack
{
    int data[50];
    int top;
```

Techknowledge Publication

```
}stack;
int empty(stack *);
void init(stack *);
void push(stack *, int);
int pop(stack *);
int partition(int[ ], int, int);
void quick_sort(int[ ], int, int);
void main()
{
    int a[30], n, i;
    printf("\n No. of elements :");
    scanf("%d", &n);
    printf("Enter elements :");
    for(i = 0; i<n; i++)
        scanf("%d", &a[i]);
    quick_sort(a, 0, n–1);
    printf("\n Sorted data ->\n ");
    for(i = 0; i<n; i++)
        printf("%d", a[i]);
    getch();
}
void quick_sort(int a[ ], int l, int u)
{
    stack s;
    int i, j;
    init(&s);
    while(i<u)
    {
        j = partition(a, l, u);
        push(&s, j+1);
        push(&s, u);
        u = j–1;
    }
    while(!empty(&s))
    {
        l = pop(&s);
        u = pop(&s);
        while(l<u)
        {
            j = partition(a, l, u);
            push(&s, j+1);
            push(&s, u);
            u = j–1;
        }
    }
}
void init(stack *s)
{
    s->top = –1;
}
int empty(stack *s)
{
    if(s->top = = –1)
        return(1);
    return(0);
}
void push(stack *s, int x)
{
    s->top = s->top +1;
    s->data[s->top] = x;
}
int pop(stack *s)
{
    int x;
    x = s->data[s->top];
    s->top = s->top –1;
    return(x);
}
int partition(int a[ ], int l, int u)
{
    int v, i, j, temp;
    v = a[l];
    i = l;
    j = u+1;
    do
    {
    do
        i++;
    while(a[i]<v && i< = u);
    do
```

Techknowledge Publication

```
        j– –;
    while(v<a[ j]);
    if(i<j)
    {
        temp = a[i];
        a[i] = a[ j];
        a[ j] = temp;
    }
    }while(i<j);
    a[l] = a[ j];
    a[ j] = v;
    return( j);
}
```

Output

```
Enter elements : 234    11   555   54
Sorted data    ->   11   54   234   555
```

5.5.4 Role of Pivot in Efficiency of Quick Sort

- A choice of pivot always influences the efficiency of quick sort algorithm. The popular choice of pivot as the first element is acceptable when input data is random.
- If the input data is in either ascending or descending order, pivot will provide a poor partition. Quick sort will require quadratic time for sorting.
- Quick sort will give its best behaviour when partition is found near center of the input data. In such a case, quick sort will sort the data in (n log n) time.
- Median of the array is good choice for partition. Unfortunately, it is hard to calculate and would slow down quick sort.
- A good estimate can be obtained by picking three elements randomly and using the median of these three as pivot.
- As an alternative, input data can be randomised and subsequently, first element can be taken as pivot.

5.6 Two-Way Merge Sort

Q. Discuss merge sort time and space complexity.
SPPU - May 18, 2 Marks

Q. Explain merge sort algorithm using divide and conquer strategy with an example.
SPPU - May 19, 5 Marks

- Merge sort runs in O(N log N) running time. It is a very efficient sorting algorithm with near optimal number of comparisons. It is based described using a recursive algorithm. Basic operation in merge sort is that of merging of two sorted lists into one sorted list. Merging operation has a linear time complexity.
- Recursive algorithm used for merge sort comes under the category of divide and conquers technique. An array of n elements is split around its centre. Producing two smaller arrays. After these two arrays are sorted independently, they can be merged to produce the final sorted array. The process of splitting and merging can be carried recursively till there is only one element in the array. An array with 1 element is always sorted. Let us try to understand the method through an example.

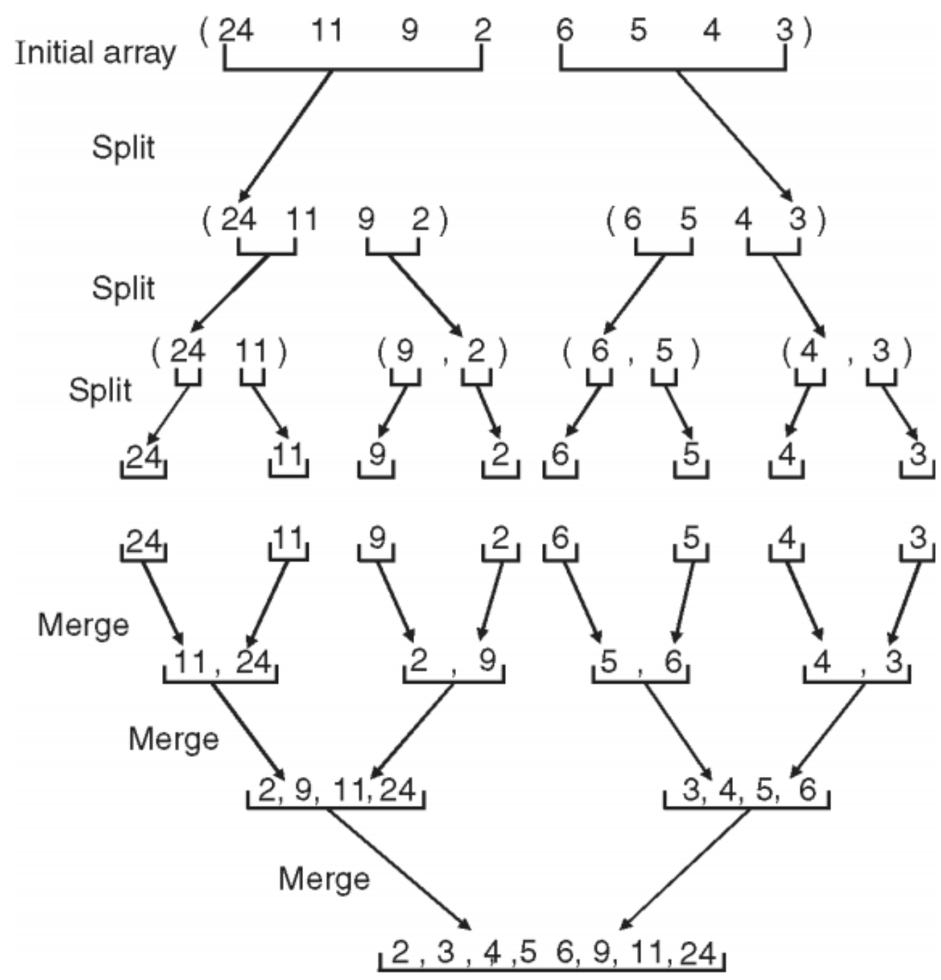


Fig. 5.6.1

Techknowledge Publications

- Given array has 8 elements. Index of the first element is i = 0, index of the last element j = 7.
- In order to divide the above list around the middle element, the index of the centre element $k = \frac{i+j}{2} = \frac{0+7}{2} = 3$.
- Merge sort is applied recursively to left half of the list from i = 0 to j = 3.
- After sorting of the left half of the list. Right half of the list is sorted from i = 4 to j = 7 recursively using merge sort.
- After both the lists are sorted, left list from i = 0 to j = 3 and right list from i = 4 to j = 7, these two lists are merged to produce a single sorted array.

'C' function for merge sort.

```
void mergesort(int a[ ], int i, int j)
{
    int k;
    if(i < j)
    {
        k = (i + j)/2;
        mergesort(a, i, k);
        mergesort(a, k+1, j);
        merge(a, i, k, j);
    }
}
```

Recursive function merge sort can be seen as postorder traversal of the binary tree :

Where traversal of the left subtree is given by - mergesort(a, i, k);

And traversal of the right subtree is given by - mergesort(a, k + 1 , j);

Visiting a node is given by merge(a, i, k, j).

If the root node starts with an array (24, 11, 9, 2, 6, 5, 4, 3) having eight elements. Value of i and j for the root node will be i = 0 and j = 7 respectively. After split, value of i and j for the left child will be i = 0 and j = 3 and for the right child it will be i = 4 and j = 7. Recursion will go on expanding until the list can no longer be divided.

Techknowledge Publication

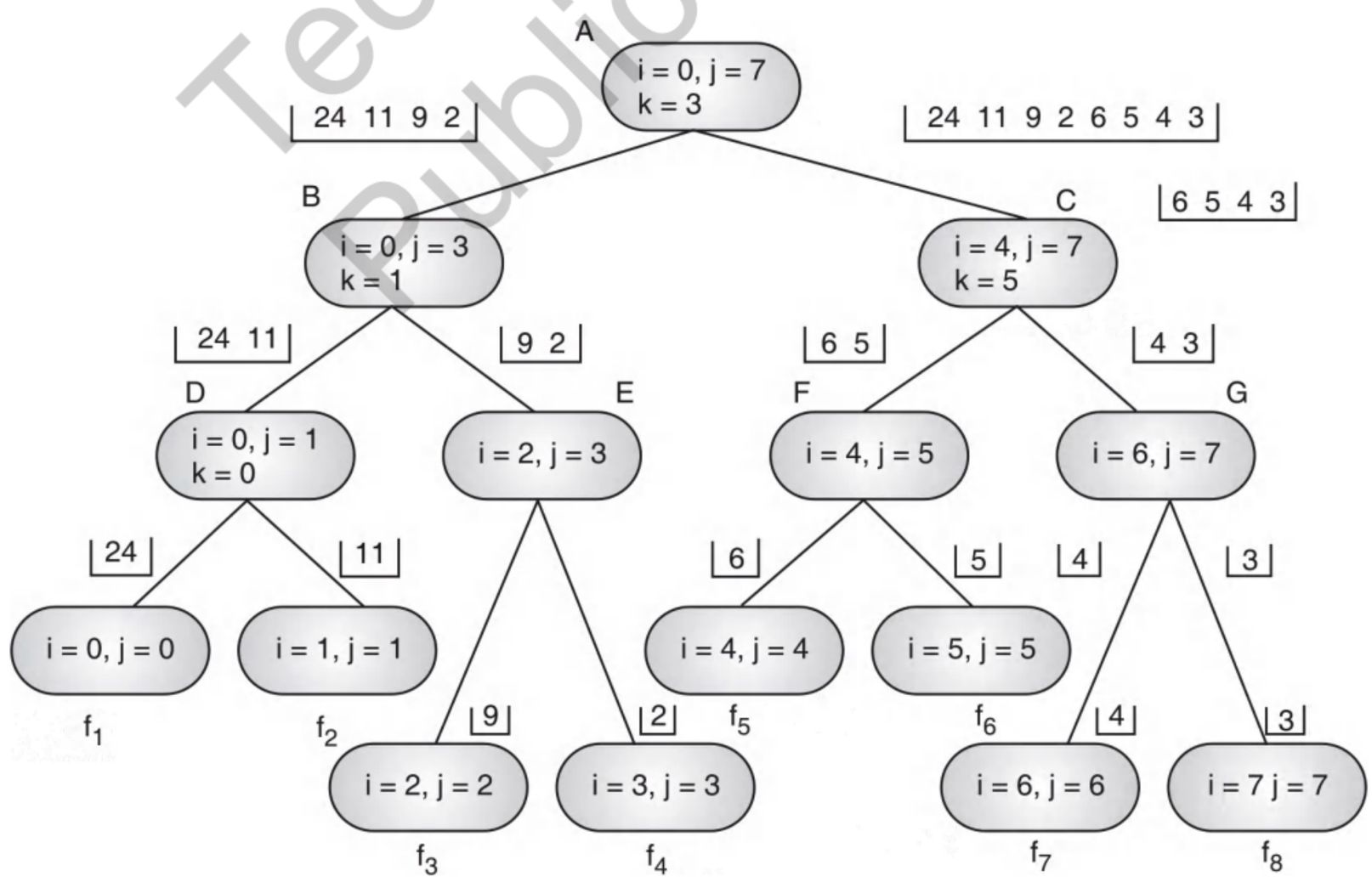


Fig. 5.6.2

Post order traversal on the recursion tree will list the nodes in the following sequence : D E B F G C A

Visit D[Merge (24) (11) giving (11, 24)]

Visit E[Merge (9) (2) giving (2, 9)]

Visit B[Merge (11, 24)(2, 9) giving (2, 9, 11, 24)]

Visit F[Merge (6) (5) giving (5, 6)]

Visit G[Merge (4) (3) giving (3, 4)]

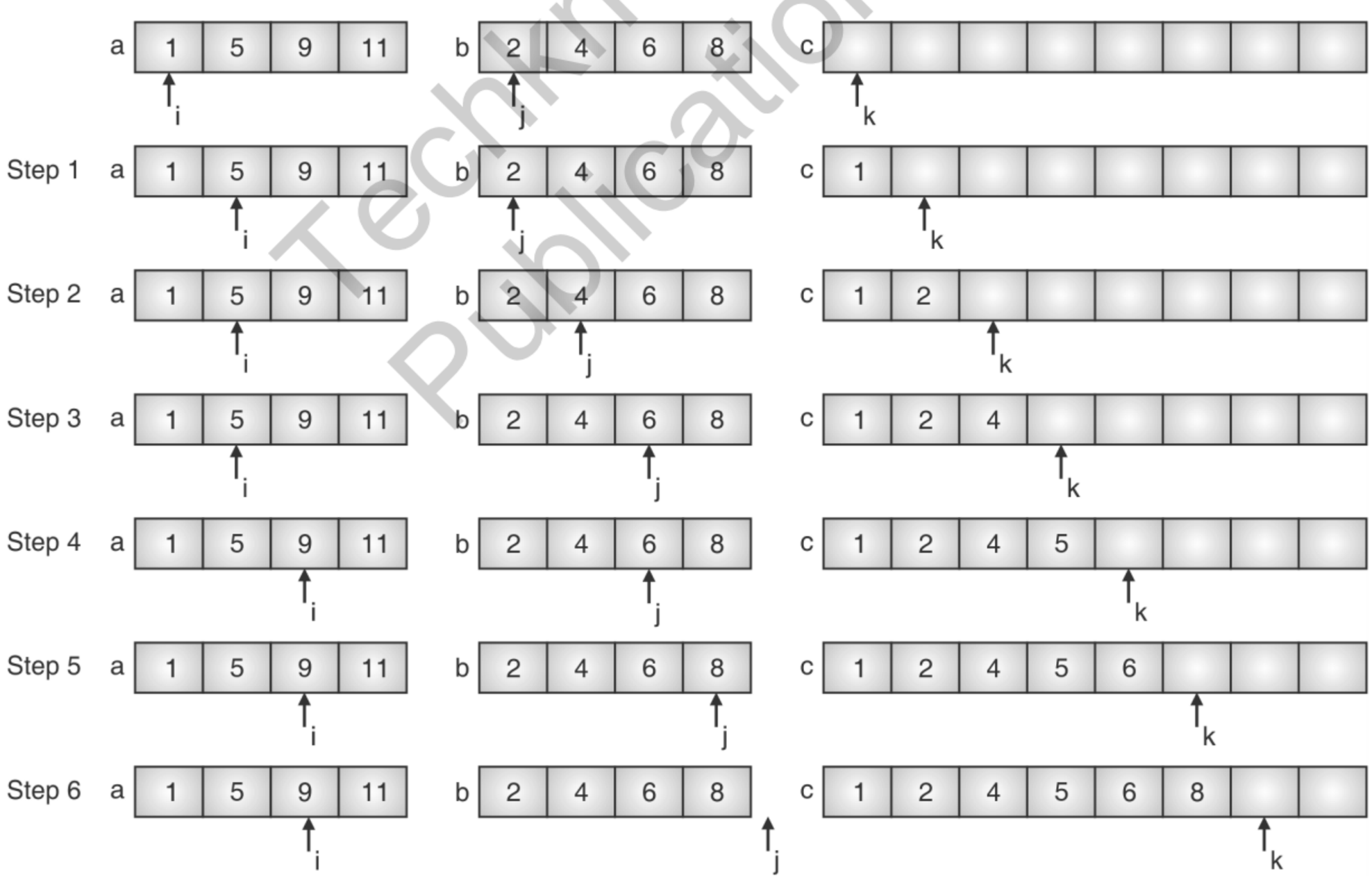
Visit C[Merge (5, 6) (3, 4) giving (3, 4, 5, 6)]

Visit A[Merge (2, 9, 11, 24), (3, 4, 5, 6) giving (2, 3, 4, 5, 6, 9, 11, 24)]

5.6.1 Merging

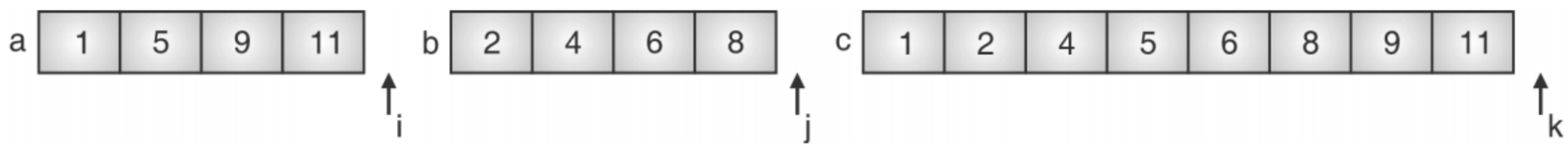
The fundamental operation in merge sort algorithm is merging two sorted arrays. The merging algorithm takes two sorted arrays a[ ] and b[ ] as input and the third array c[ ] as output. Three variables i, j and k are initially set to the beginning of the three arrays a[ ], b[ ] and c[ ]. The smaller of a[i] and b[j] is copied to c[k] and appropriate variables (i, k) or (j, k) are advanced. When either of the input array a[ ] or b[ ] is exhausted, the remainder of the other array is copied to c.

Initial conditions



TechKnowledge Publications

Array b[ ] is exhausted, the remainder of the array a[ ] are added to c[ ].



Ex. 5.6.1 : Sort the following list of numbers using merge sort. Show result stepwise : 50, 10, – 10, 40, 15, 25, 20, 35, 30.

Soln. :

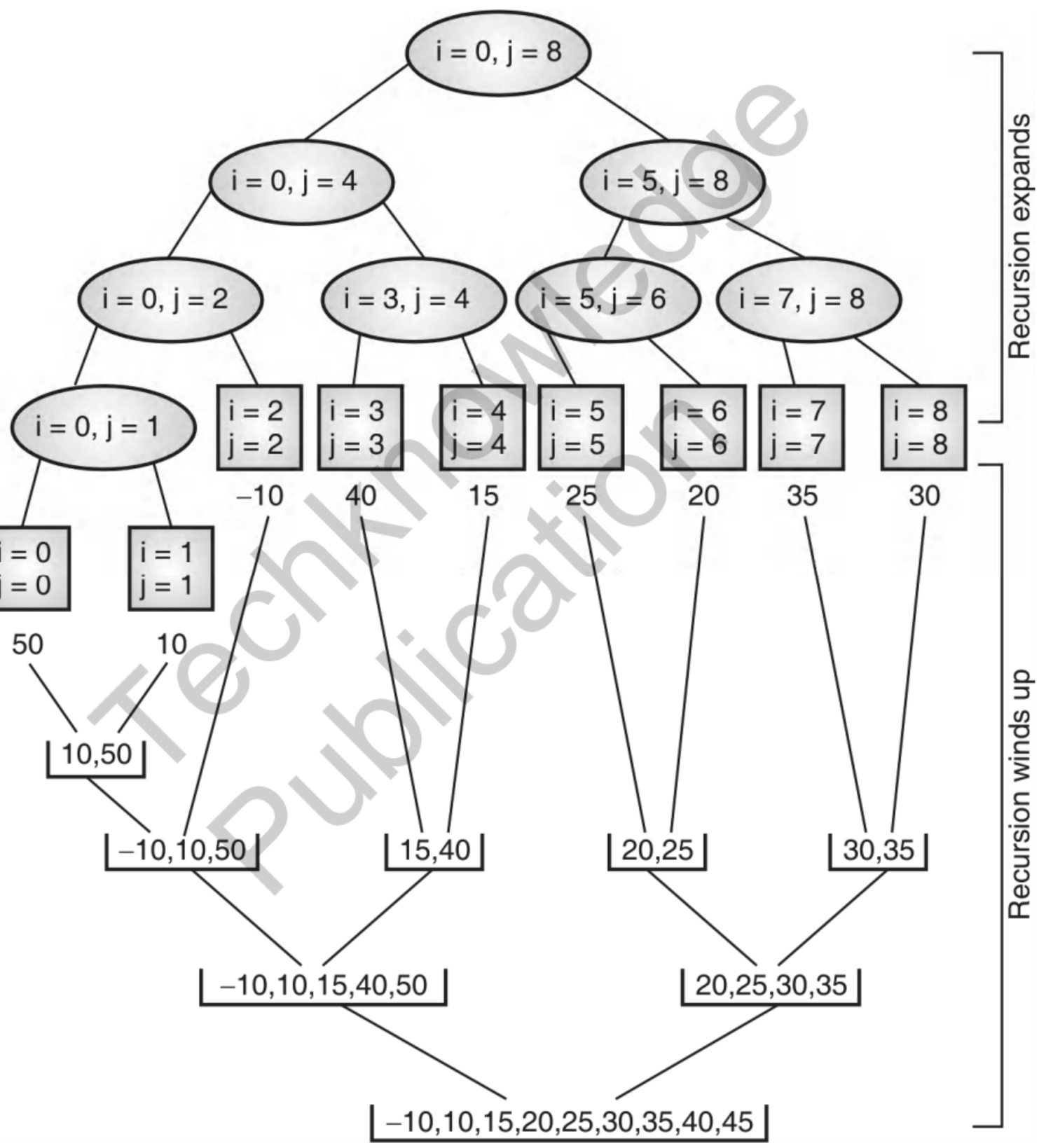


Fig. Ex. 5.6.1

Ex. 5.6.2 : Sort the following numbers using iterative merge sort in ascending order.

56, 12, 84, 56, 28, 0, – 13, 47, 94, 31, 12, – 2

Soln. :

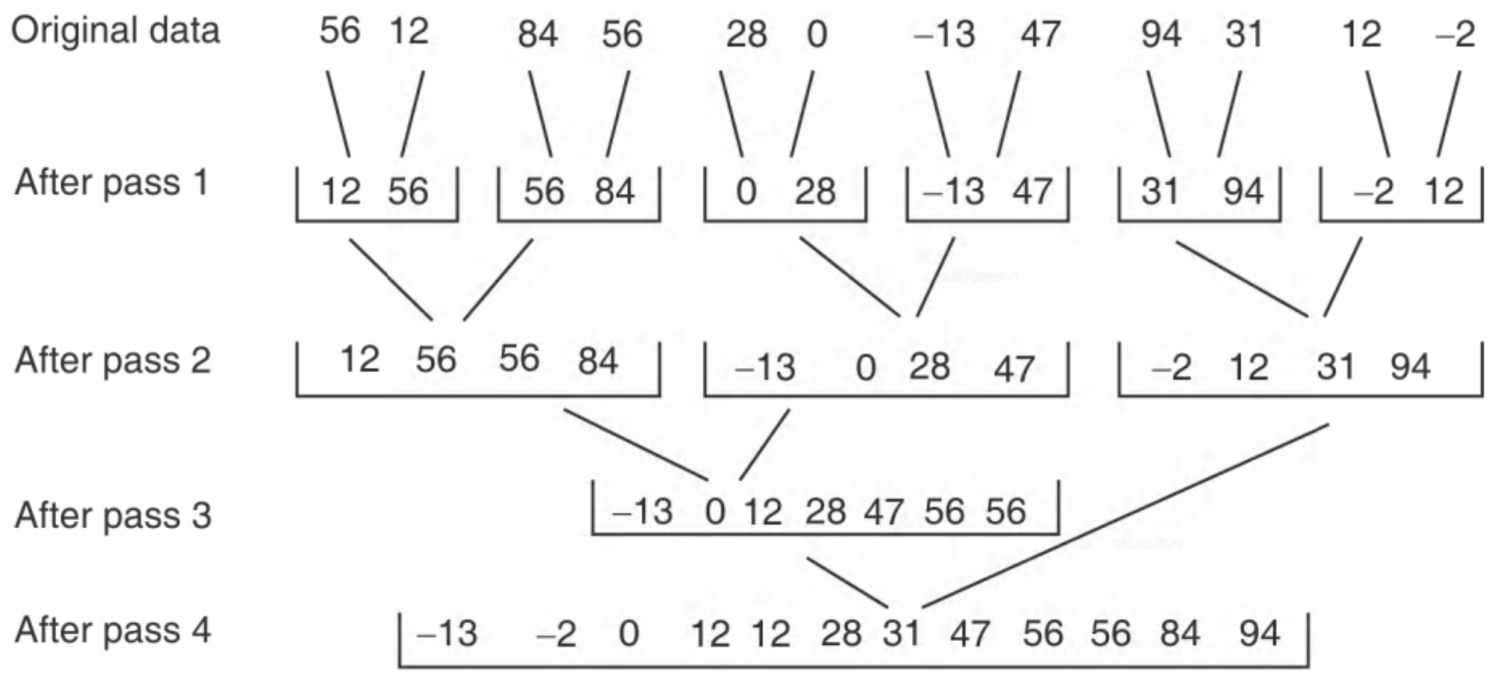


Fig. Ex. 5.6.2

Ex. 5.6.3 : Consider the following set of numbers, sort them using iterative merge sort. Show all passes :
20, 24, 48, 37, 12, 92, 86, 07.

Soln. :

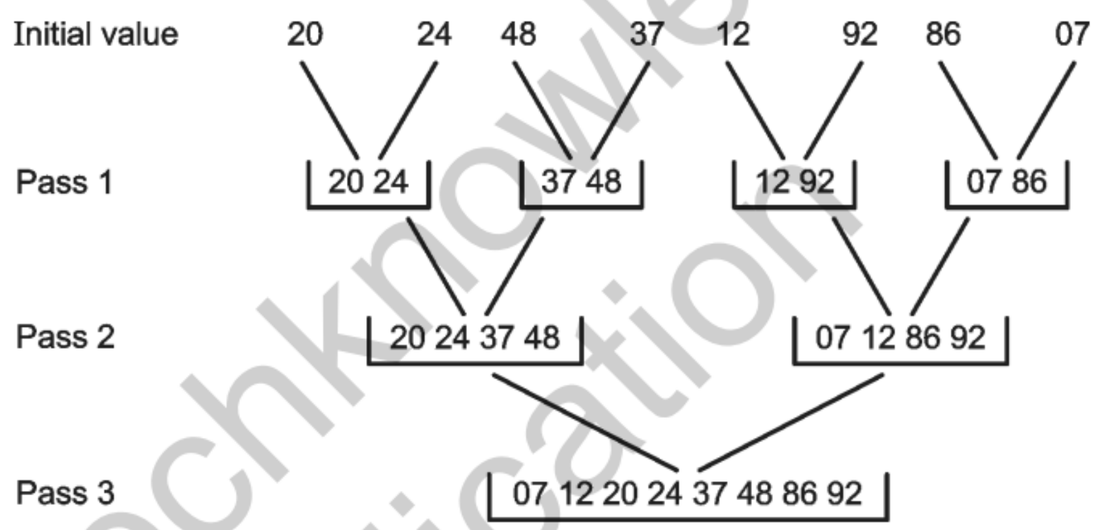


Fig. Ex. 5.6.3

Ex. 5.6.4 : Compare the selection and merge sort with respect to (i) Sort stability (ii) Efficiency (iii) Passes.

Soln. :

(i) Both selection and merge sort is stable algorithm. Both the algorithms are based on left to right scanning of the array and hence the identical numbers will maintain their relative sequence even in the sorted array.

(ii) Loop for selection sort is given below :

for (i = 0; i<n–1; i++)

select the smallest element

along a[i], ….. a [n – 1] and swap it with a [i];

In the ith pass, n – i comparisons will be needed to select the smallest element. n – 1 passes are required to sort the array.

Thus, the number of comparison needed to sort an array having n elements.

$$= (n-1) + (n-2) + \ldots + 2 + 1$$

$$= \frac{n(n-1)}{2} = \frac{1}{2}(n^2 - n)$$

Which can be considered to be of the order of n^2 for large n.

Recursive definition of merge sort is given by,

$$\text{Sort}(a,i,j) = \text{Sort}\left(a, \frac{i+j}{2}, j\right) \text{ if } (i < j)$$

$$\text{Sort}\left(a, \frac{i+j}{2} + 1, j\right)$$

$$\text{Merge}\left(a, i, \frac{i+j}{2} + 1, j\right)$$

Sorting an array of n elements has been expressed recursively into two sub problems of n/2 elements. Merging of the two arrays has a linear time complexity.

Hence,
$$T(n) = 2T(n/2) + n = 2^2T(n/4) + 2n$$
$$= 2^2\,T(n/2^2) + 2n$$
$$= 2^hT(n/2^h) + hn$$

Let us find the smallest h satisfying

$$2^h \geq n \quad \text{or} \quad h \geq \log n$$

For simplicity, let us take the case of h = log n equation (1) can be re-written as,

$$T(n) = nT(1) + h \log n$$

For large n, T(n) can be approximated to n log n.

Thus, selection sort has a timing complexity of quadratic order. Merge sort has a timing complexity of the order of n log n.

(iii) Selection sort requires n – 1 passes, whereas merge sort required log n passes.

Ex. 5.6.5 : Write output of each pass of merge sort for the following list : 26, 5, 77, 1, 61, 11, 59, 15, 8, 19 .

Soln. :

Output is being shown for iterative merge sort.

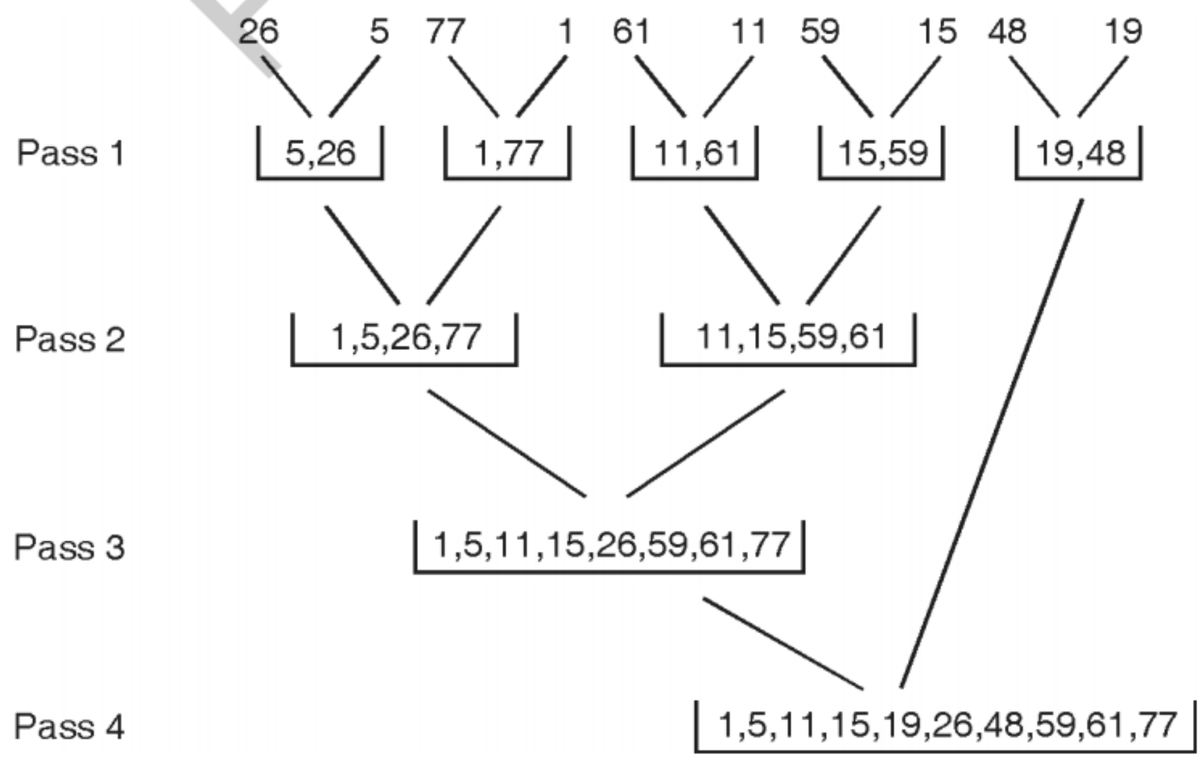


Fig. Ex. 5.6.5

TechKnowledge Publications

Ex. 5.6.6 : Sort the following data using merge sort. Show all passes : 25, 3, 55, 2, 60, 10, 50, 14, 36, 18.

Soln. :

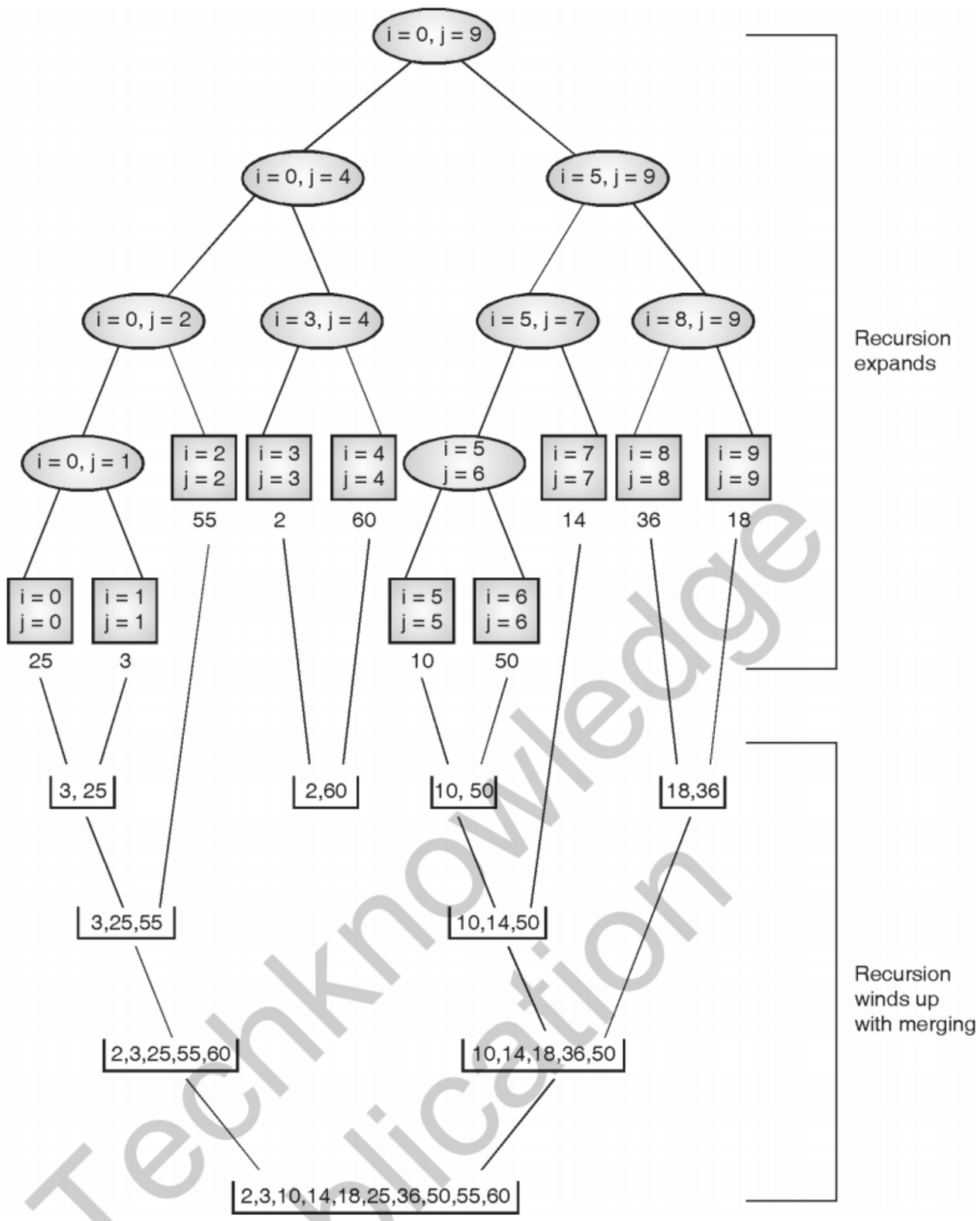


Fig. Ex. 5.6.6

Ex. 5.6.7 : Show all passes to sort the values using Merge Sort in ascending order : 21, 5, 7, 11, 35, 76, 0, 9, 27, 45.

Soln. :

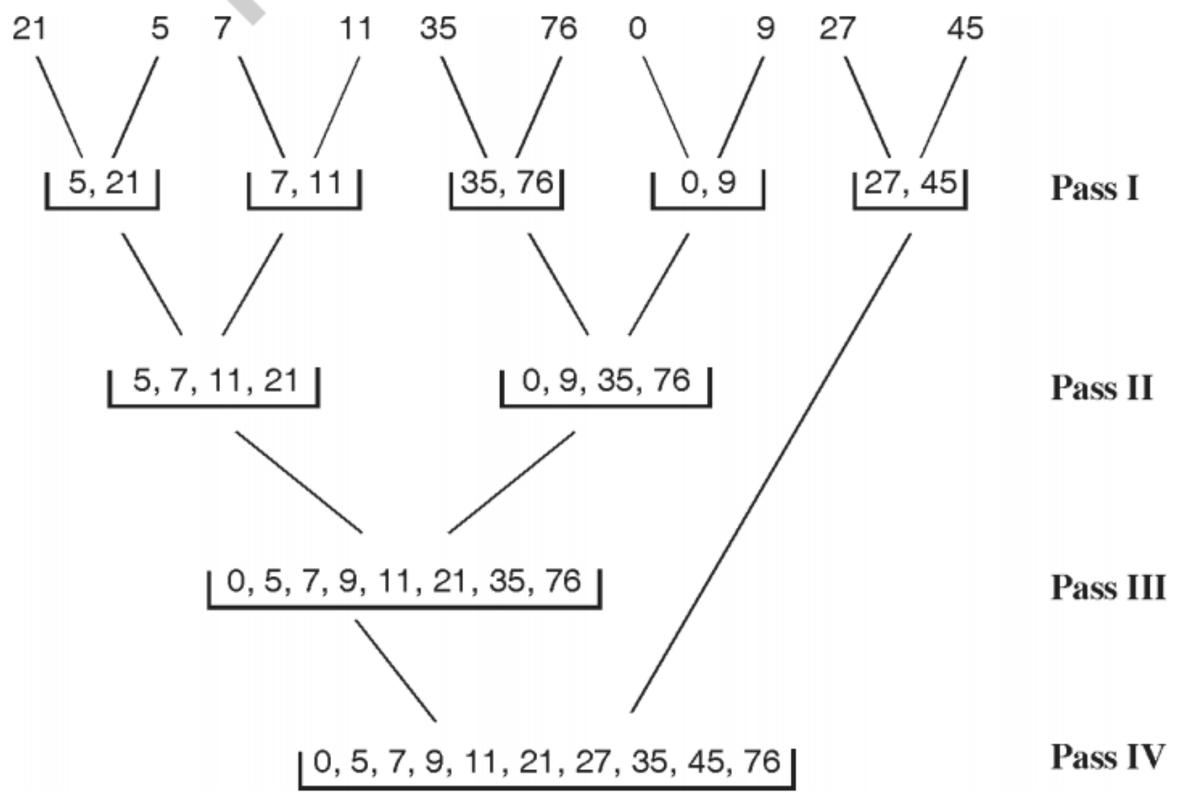


Fig. Ex. 5.6.7

Techknowledge Publications

Ex. 5.6.8 : Sort the following numbers using Merge sort : 55, 85, 45, 11, 34, 05, 89, 99, 67. **SPPU - Dec. 17, 4 Marks**

Soln. :

Sorting using merge sort

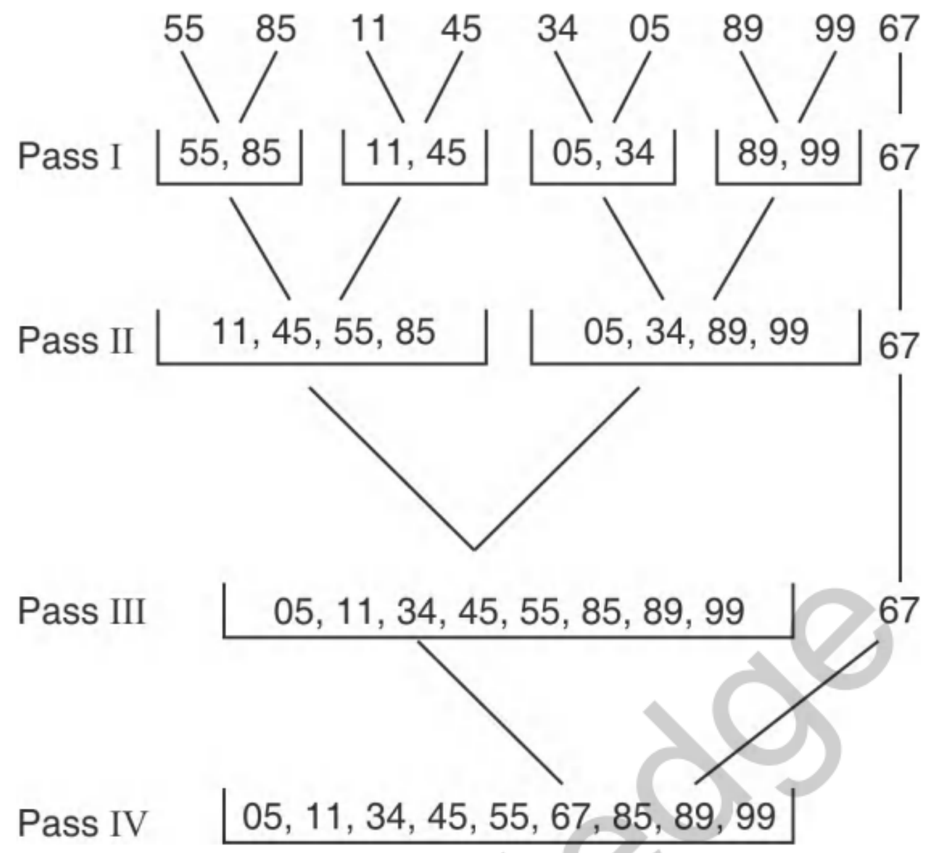


Fig. Ex. 5.6.8

Ex. 5.6.9 : Explain Merge sort using the following example : 18, 13, 12, 22, 15, 24, 10, 16, 19, 14, 30.

SPPU - May 18, 4 Marks

Soln. :

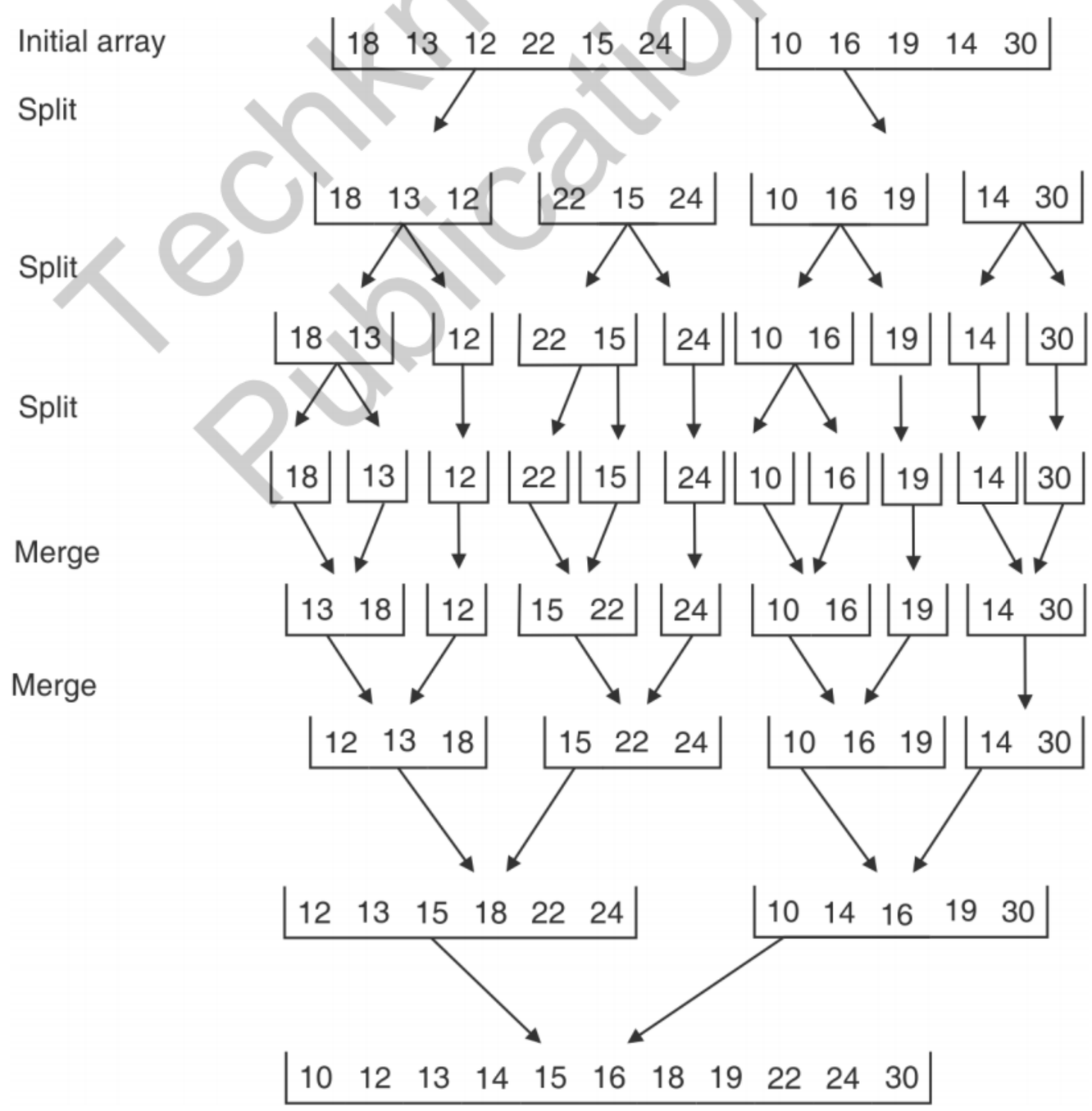


Fig. Ex. 5.6.9

TechKnowledge Publications

'C' function for merging of two sorted arrays.

```
void merge(int a[ ], int l, int m, int u)
{
    int c[MAX];
    int i, j, k;
    /* First array is supposed to be
       from location l to m of array a[ ] */
    /* Second array is supposed to be from m+1
       to u of array a[ ] */
    /* Array c[ ] is used for merging */
    /* after merging array c[ ] is copied back to a[ ] */
    i = l;
    j = m+1;
    k = 0;
    while(i< = m && j< = u)
    {
        if(a[i] < a[ j])
        {
            c[k] = a[i];
            k++; i++;
        }
        else
        {
            c[k] = a[ j];
            k++; j++;
        }
    }
    while(i< = m)
    {
        c[k] = a[i];
        i++; k++;
    }
    while( j< = u)
    {
        c[k] = a[ j];
        k++; j++;
    }
    for(i = l, j = 0; i< = u; i++, j++)
        a[i] = c[ j];
}
```

Program 5.6.1 : Program for implementation of merge sort.

```
#include<stdio.h>
#include<conio.h>
#define MAX 30
void merge_sort(int[ ], int, int);
void merge(int[ ], int, int, int);
void main()
{
    int a[30], n, i;
    printf("\n No. of elements :");
    scanf("%d", &n);
    printf("Enter array elements :");
    for(i = 0; i<n; i++)
        scanf("%d", &a[i] );
    merge_sort(a, 0, n–1);
    printf("\n Sorted data ->\n ");
    for(i = 0; i<n; i++)
        printf("%d", a[i]);
    getch();
}
void merge_sort(int a[ ], int i, int j)
{
    int k;
    if(i<j)
    {
        k = (i+j)/2;
        merge_sort(a, i, k);
        merge_sort(a, k+1, j);
        merge(a, i, k, j);
    }
}
void merge(int a[ ], int l, int m, int u)
{
    int c[MAX];
    int i, j, k;
    /* First array is supposed to be
       from location l to m of array a[ ] */
    /* Second array is supposed to be from m+1
       to u of array a[ ] */
    /* Array c[ ] is used for merging */
```

Techknowledge Publication

```
/* after merging array c[ ] is copied back to a[ ] */
i = l;
j = m+1;
k = 0;
while(i< = m && j< = u)
{
    if(a[i] < a[ j])
    {
        c[k] = a[i];
        k++; i++;
    }
    else
{
        c[k] = a[ j];
        k++; j++;
    }
}
while(i< = m)
{
    c[k] = a[i];
    i++; k++;
}
while( j< = u)
{
    c[k] = a[j];
    k++; j++;
}
for(i = l, j = 0; i< = u; i++, j++)
    a[i] = c[ j];
}
```

Output

```
No. of elements      : 4
Enter array elements : 32    56    22    1
Sorted data     ->   1   22    32        56
```

5.6.2 Analysis of Merge Sort

Q. Discuss merge sort time complexity and space complexity. **SPPU - Dec. 17, May 19, 2 Marks**

Since, merge sort is a recursive program, we must write a recurrence relation for the running time.

Time to merge sort N numbers is equal to the time to merge sort left N/2 element, plus time to merge sort right N/2 element, plus time to merge two linear lists of N/2 elements. Merging requires one pass of linear scanning of the two input arrays (each having N/2 elements). Hence the time to merge is linear.

T (1) = 1, since recursion terminates when N = 1. Time to merge sort an array having 1 element is constant.

$$T(N) = 2T(N/2) + N$$

Or
$$\frac{T(N)}{N} = \frac{T(N/2)}{N/2} + 1 \qquad \ldots(1)$$

Equation (1) can be written as (expanding T (N / 2)),

$$= \frac{T(N/4)}{N/4} + 1 + 1$$

$$= \frac{T(N/2^2)}{N/2^2} + 2$$

Let us assume, $N = 2^h$ i.e. $N / 2^h = 1$

Equation (1) can be written as,

$$\frac{T(N)}{N} = \frac{T(N/2^h)}{N/2^h} + h$$

$$= \frac{T(1)}{1} + h = T(1) + h \qquad \ldots(2)$$

Since, $N = 2^h$

Taking log on both sides, we have $\log N = \log_2^{2^h} = h$

Writing h as log N in Equation (2)

$$\frac{T(N)}{N} = T(1) + \log N \quad \text{or}$$

$$T(N) = NT(1) + N \log N$$

$$= \mathbf{O(N \log N)}$$

Although the running time of merge sort is $O(N \log_2 N)$, it is seldom used for internal sorting. Merging of two sorted arrays require additional memory. Two sorted arrays are merged through a temporary array and the contents of the temporary array are copied back requiring additional space and time. Merge sort is found to be very effective in external sorting. It requires O(n) extra space, including O(log n) space for the recursive call on the stack.

TechKnowledge Publications

5.6.3 Non-Recursive Merge Sort

Non-recursive merge sort uses bottom up approach.

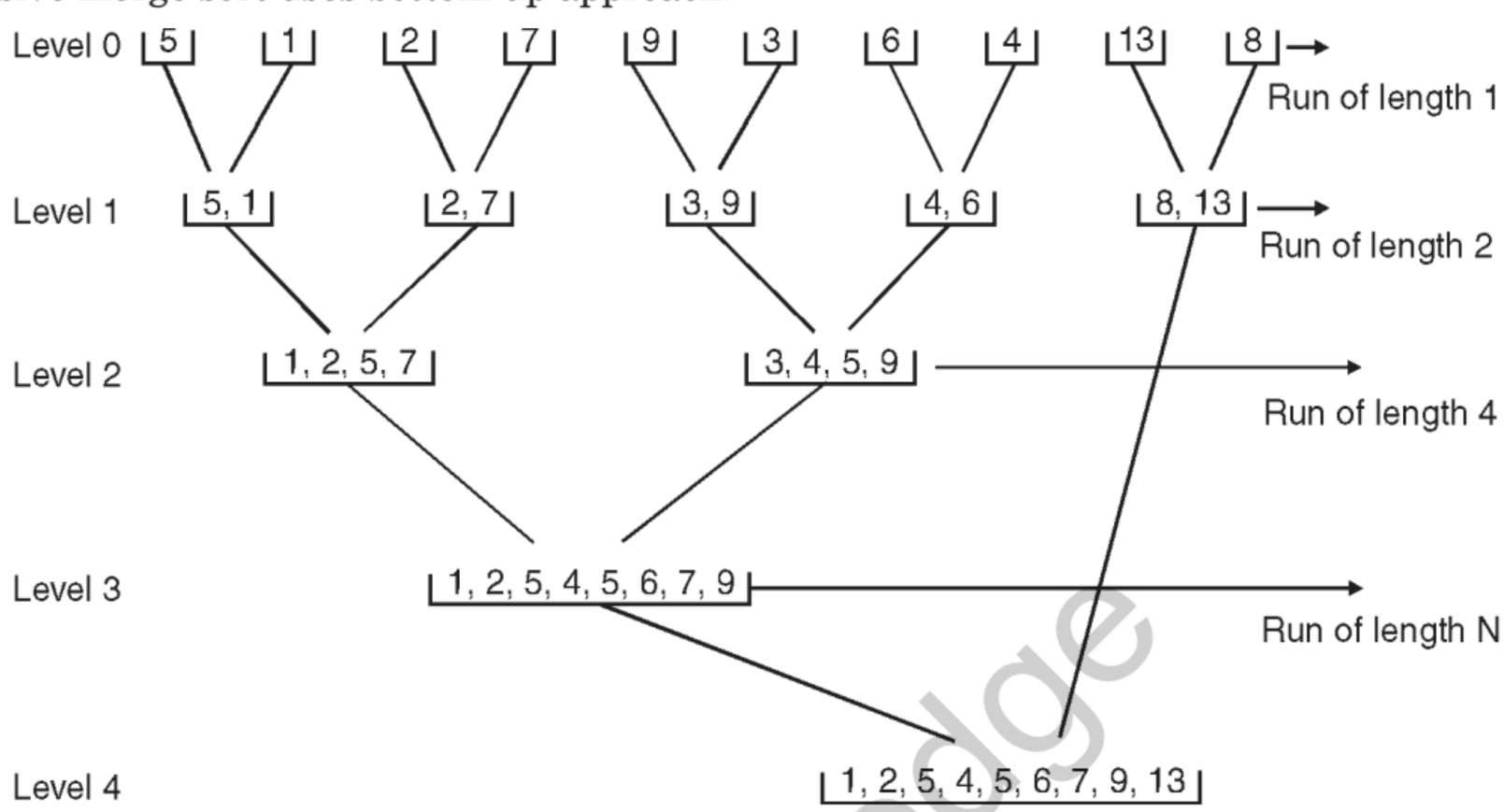


Fig. 5.6.3

When the initial array a[ ] of N elements is divided into N groups (each group having 1 element), elements inside a group are sorted (a group with 1 element is always sorted). This is also called that initially, array is organized into runs of length 1.

The first pass of the algorithm merges n pair of runs of length = 1 at level 0 to create runs of length 2 at level 1.

The second pass merges a pair of runs of length = 2 at level 1 to create runs of length 4 at level 2. In every successive pass, number of runs is reduced by half and the length of runs is doubled.

If P passes are required for sorting, then $2^P >= N$.

'C' function for non-recursive merge sort

```
void non_recursive_mergesort(int a[ ], int N)
{
    int run_size, no_of_runs, i, j;
    run_size = 1;    //one element at a time at level 0
    while(run_size<N)
    {
        no_of_runs = N/run_size;
        if(no_of_runs*run_size < N)
        no_of_runs++;  //last run of smaller size
        i = 0; j = run_size;
        /* i = begining of the first run
        j = begining of the next run */
        while(no_of_runs > 1)
        {
            if(i+2*run_size > N)  //last run of smaller size
                merge(a, i, j–1, N–1);
            else
                merge(a, i, j–1, j+run_size–1);
            i = i+run_size;
            j = j+run_size;
            no_of_runs = no_of_runs–2;
        }
        run_size = run_size * 2;
    }
}
```

5.7 Counting Sort

Q. Write a pseudo C/C++ code to sort the data using bucket sort in ascending order.

SPPU - May 18, 7 Marks

TechKnowledge Publications

If we have n integers in the range 0 to R–1, we can use this information to obtain a fast sort known as bucket sort.

- We taken an array frequency[ ] of size R.
- Array frequency[ ] is initialised to zero.
- Array frequency[ ] has R cells, which are initially empty.
- When a number a_i is read, increment frequency $[a_i]$ by 1.
- After all numbers are read, print the sorted list from the array frequency[ ].

Program 5.7.1 : Program for bucket sort. It is assumed that all numbers are positive. Numbers are in the range 0 to 999.

```
#include<stdio.h>
void main()
{
    int frequency[1000], n, i, x;
    printf("\n Enter no. of elements :");
    scanf("%d", &n);
    /* Initialise the array frequency */
    for(i = 0; i<1000; i++)
        frequency[i] = 0;
    printf("\n Enter the data to be sorted :");
    for(i = 0; i<n; i++)
    {
        scanf("%d", &x);
        frequency[x]++;
    }
    /* print the result */
    printf("\n result is :");
    for(i = 0; i< = n; i++)
        if(frequency[i]>0)
            while(frequency[i]>0)
            {
                printf("%d\t", i);
                frequency[i]– –;
            }
}
```

Output

```
Enter no. of elements : 5
Enter the data to be sorted : 5 4 3 2 1
result is : 1    2    3    4    5
```

Ex. 5.7.1 : Initialize a one-dimensional array of size 50 with alphabetic data. Sort the array without physically disturbing the data in the array.

Soln. :

It is simple example of bucket sort since the ASCII of characters are in the range of 0 to 127, we can use this information to obtain a fast sort known as bucket sort.

We take an array frequency[ ] of size 128.

Array frequency is initialized to zero.

When a character a_i is read, increment frequency$[a_i]$ by 1.

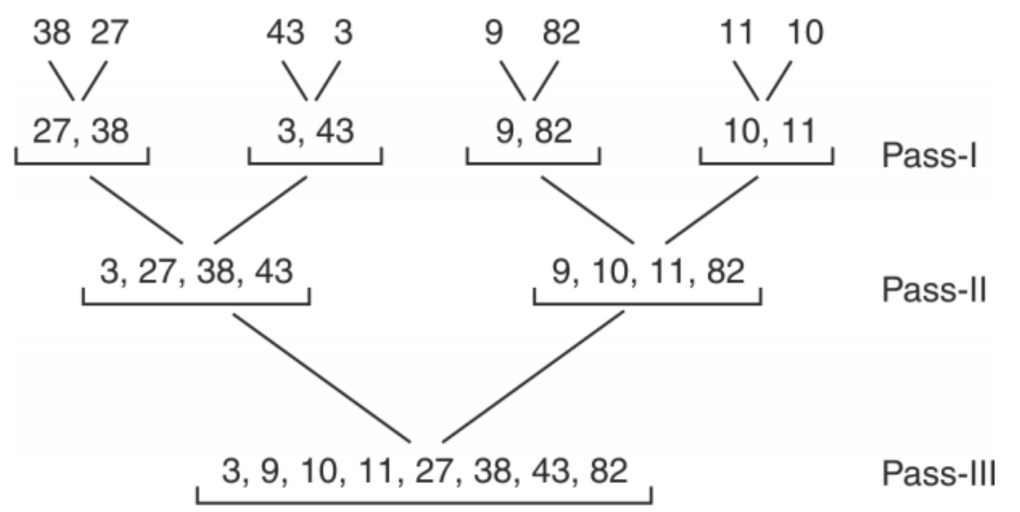
After all characters are read, print the sorted array using the array frequency[ ].

```
# include <stdio.h>
void main()
{
    int frequency[128], i;
    char a[50];
    printf("\n enter 50 alphabets; ");
    gets(a);
    /* initialize the array frequency */
    for(i = 0; i<128; i++)
        frequency [i] = 0;
    /* find the frequency of each alphabet */
        for(i = 0; a[i]! = '\0'; i++)
        frequency [a[i]++; ]
    /* Print the result */
    for(i = 0; i < 128; i++)
    while(frequency [i]>0)
    {   printf("%c", i);
        frequency[i]– – ;
    }
}
```

Techknowledge Publication

Ex. 5.7.2 : Sort the following list using merge sort :
38, 27, 43, 3, 9, 82, 11, 10

Soln. :



Ex. 5.7.3 : Sort the following numbers in ascending order using radix sort :
14, 1, 66, 74, 22, 36, 41, 59, 64, 54

Soln. : Buckets after 1st pass

0	1	2	3	4	5	6	7	8	9
	41 1	22		54 64 74 14	36 66				59

Merged list = 1 41 22 14 74 64 54 66 36 59

Buckets after 2nd pass

0	1	2	3	4	5	6	7	8	9
1	14	22	36	41	59 54	66 64	74		

Merged list = 14 22 36 41 54 59 64 66 74

5.8 Radix Sort

Radix sort is generalization of bucket sort. To sort decimal numbers, where the radix or base is 10, we need 10 buckets. These buckets are numbered 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Sorting is done in passes.

Number of passes required to sort using shell sort is equal to number of digits in the largest number in the list.

Range	Passes
0 to 99	2 passes
0 to 999	3 passes
0 to 9999	4 passes

- In the first pass, numbers are sorted on least significant digit. Numbers with the same least significant digit are stored in the same bucket.
- In the 2nd pass, numbers are sorted on the second least significant digit. At the end of every pass, numbers in buckets are merged to produce a common list.
- Number of passes depends on the range of numbers being sorted.
- The following example shows the action of radix sort.

Initial numbers :

10 5 99 105 55 100 135 141 137 200 199

Buckets after 1st pass

Bucket no.	0	1	2	3	4	5	6	7	8	9
	200 100 10	141				135 55 105 5		137		199 99

Merged list = 10 100 200 141 5 105 55 135 137 99 199

TechKnowledge Publications

Buckets after 2nd pass

0	1	2	3	4	5	6	7	8	9
100									
5									
200			137						199
100	10		135	141	55				99

Merged list = 100 200 5 105 10 135 137 141 55 99 199

Buckets after third pass

0	1	2	3	4	5	6	7	8	9
	199								
	141								
99	137								
55	135								
10	105								
5	100	200							

Merged list = 5 10 55 99 100 105 135 137 141 199 200

TechKnowledge Publications

5.8.1 Algorithm for Radix Sort

```
/* Numbers to be sorted are in array a[ ] and there
are N numbers */
large = find the largest number in a[ ]
passes = Number of digits in large
Div = 1
/* divisor for extracting ith least significant digit */
for(i = 1; i < = passes; i++)
{
    Initialize all buckets b0 to b9
    for each number x from a[0] to a[N – 1]
    {   bucket_no =(x/div)%10
        insert x in bucket with bucket_number
    }
Copy elements of buckets back in array a[ ]
DIV = DIV * 10
}
exit
```

5.8.2 C Function for Radix Sort

Input to functions are :

n = number of elements in the array a[ ]

a[ ] = Array holding elements.

Buckets are represented using a two dimensional array buckets [10] [10]. Each bucket can hold upto ten values. Another array count[10] gives number of elements in the corresponding bucket. If a bucket [1] has five elements then the value of count [1] will be five.

Whenever, an element is inserted in i^{th} bucket, count [i] should be incremented by 1.

```
void radix(int a[ ], int n)
{
   int buckets[10][10], count[10];
   int passes, large, div, bucketno, i, j, k;
   /* find the largest number in a[ ]  */
```

```
    large = a[0];
    for(i = 1; i < n; i++)
        if(a[ i] > large)
            large = a[i];
    /* Find number of digits in large and store the count
        in passes */
    passes = 0;
    while(large > 0)
    {   passes++
        large = large /10;
    }
    /* perform radix sort */
    div = 1;

    for(i = 1; i < = passes; i++)
    {           /*initialize bucket */
        for( j = 0; j < = 9; j++)
            count [ j] = 0;

        /* insert elements in respective buckets */

        for(j = 0; j < n; j++)
        {   bucketno =(a[ j]/div)%10;
            buckets[bucketno][count[bucketno]] = a[j];
            count [bucketno]++;
        }
        /* Merge elements of buckets and store them in
            a[0] ...a[n – 1] */
        j = 0;
        for(bucketno = 0; bucketno < = 9;
bucketno++)
        for(k=0; k < count[bucketno]; k++)
            a[j++] = buckets[bucketno] [k];
            div = div * 10;
    }
}
```

Program 5.8.1 : Program to sort elements of an array using radix Sort.

```
#include <stdio.h>
#include <conio.h>
void radix(int a[ ], int n);
void main( )
{
    int a[30], n, i;
    printf("\n enter no. of elements");
    printf("\n enter data :");
        scanf("%d", &a[n]);
    for(i = 0; i < n; i++)
        scanf("%d", &a[i]);
        radix(a, n);
    printf("\n sorted data : ");
    for(i = 0; i < n; i++)
        printf("%d\t", a[i]);
}
void radix(int a[ ], int n)
{
    int buckets[10][10], count[10];
    int passes, large, div, bucketno, i, j, k;
    /* Find the largest number in a [ ]  */
    large = a[0];
    for(i = 1; i < n; i++)
        if(a[ i] > large)
            large = a[i];
    /* Find number of digits in large and store the count
    in passes */
    passes = 0;

    while(large > 0)
    {   passes++
        large = large /10;
    }
    /* perform radix sort */
    div = 1;

    for(i = 1; i < = passes; i++)
    { /*initialize bucket */
        for( j = 0; j < = 9; j++)
            count [j] = 0;
        /* insert elements in respective buckets */
        for( j = 0; j < n; j++)
        {
```

Techknowledge Publication

```
        bucketno = (a[j]/div)%10;
        buckets[bucketno][count[bucketno]] = a[j];
        count [bucketno]++;
    }
    /* Merge elements of buckets and store them in
       a[0] ...a[n – 1] */
    j = 0;

  for(bucketno = 0; bucketno < = 9; bucketno++)
      for(k=0; k < count[bucketno]; k++)
          a[j++] = buckets[bucketno] [k];
          div = div * 10;
}
```

Output

```
enter no. of elements      : 11
enter data : 10 5 99 105 55 100 135 141 137 200 199
sorted data : 5 10 55 99 100 105 135 137 141 199
200
```

5.8.3 Analysis of Radix Sort

If the number to be sorted are k digits long then inside every pass :

Distribution of elements among buckets take O(n) time.

Merging of elements take O(n) time

Number of passes = k

∴ Overall complexity is O(kn)

if k is regarded as a constant

then

O(kn) = O(n).

Thus, timing complexity of Radix Sort = O(n)

Space requirement : There is additional memory requirement for buckets and storing number of elements in each bucket.

Radix sort is the most preferred algorithm for sorting of strings.

Ex. 5.8.1 : Sort the following elements in ascending order using radix sort. Show all passes :
121, 235, 55, 973, 327, 179.

Soln. : Numbers are being sorted using radix sort. Radix sort is generalization of bucket sort.

Buckets after 1st pass

Bucket Number	0	1	2	3	4	5	6	7	8	9
		121		973		55 235		327	178	

Fig. Ex. 5.8.1(a)

Merged list : 121, 973, 235, 55, 327, 178

Buckets after 2nd pass

Bucket Number	0	1	2	3	4	5	6	7	8	9
			327 121	235		55		178 973		

Fig. Ex. 5.8.1(b)

Merged list : 121, 327, 235, 55, 973, 178

Buckets after 3rd pass :

Bucket Number	0	1	2	3	4	5	6	7	8	9
	55	178 121	235	327						973

Fig. Ex. 5.8.1(c)

Merged list : 55, 121, 178, 235, 327, 973

Ex. 5.8.2 : Sort the following numbers using radix sort. Show all the passes to sort the values using bucket sort in ascending order.
56 12 84 56 28 0 – 13 47 94 31 12 – 2

Soln. : Above example is being sorted using the generalized bucket sort (i.e. Radix sort).

Since, the Radix sort cannot handle negative numbers, it is necessary to subtract the smallest number from every number to remove negative numbers from the array.

Smallest number = – 13

Subtracting – 13 from every number, we get,

69 25 97 69 41 13 0 60 107 44 25 11

Techknowledge Publication

Buckets after 1st pass (sorting on least significant digit) :

0	1	2	3	4	5	6	7	8	9
60	11				25		107		69
0	41		13	44	25		97		69

Merged list : 0 60 41 11 13 44 25 25 97 107 69 69

Buckets after 2nd pass :

0	1	2	3	4	5	6	7	8	9
						69			
107	13	25		44		69			
0	11	25		41		60			97

Merged list

0 107 11 13 25 25 41 44 60 69 69 97

0	9		9
97			
69			
69			
60			
44		– – – – – – –	
41			
25			
25			
11			
0	104		

Merged list : 0 11 13 25 25 41 44 60 69 69 97 107

Adding the value – 13 (it was subtracted earlier)

We get,

– 13 – 2 0 12 12 28 31 47 56 56 84 94

5.9 Bucket Sort

- Here, elements are first divided into several groups called buckets.
- The elements inside each bucket are sorted using any of the suitable sorting algorithms.

Techknowledge Publications

– Sorting of elements inside a bucket can also be done by recursively calling the same algorithm. Finally, the elements of the buckets are combined to produce the sorted array.

Example

1. Given list of elements

   = 5, 3, 11, 9, 4, 23, 17, 3, 8, 15, 20, 21

2. Create an array of size 5, where each elements of the array is a bucket.

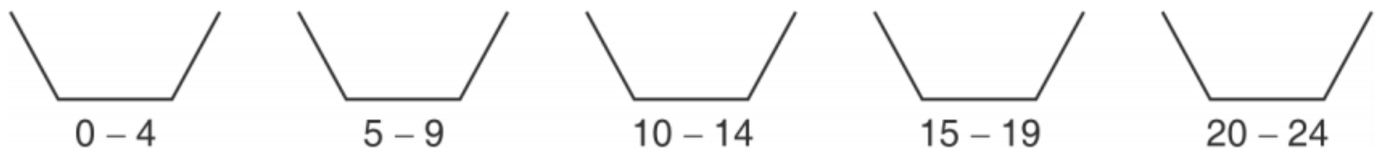


Fig. 5.9.1

3. Elements are inserted into the buckets according to the range of the bucket.

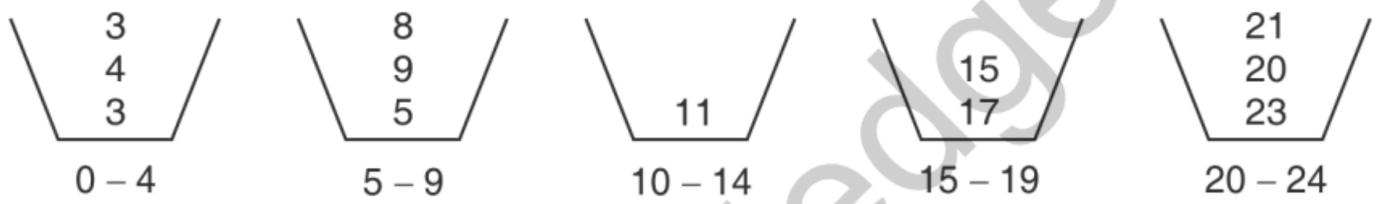


Fig. 5.9.2

4. The elements of each bucket are sorted using any of the algorithms.

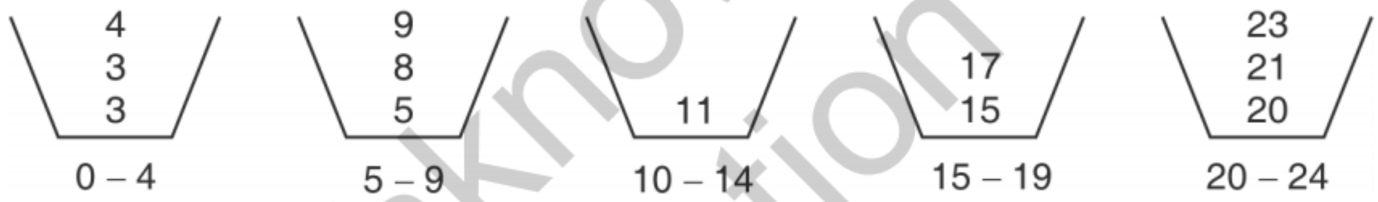


Fig. 5.9.3

5. The elements of the buckets are gathered to produce the sorted list.

   3, 3, 4, 5, 8, 9, 11, 15, 17, 20, 21, 23

Algorithm

```
BucketSort()
{
    Create an array of buckets, where each bucket can hold a range of values.
    for all the buckets
        initialize each bucket with 0 values
    for each element in the list
        insert the element in the suitable bucket
    for all the buckets
        Sort elements in each bucket
    Gather elements from each bucket
    end BucketSort
}
```

TechKnowledge Publications

Program 5.9.1 : Program to sort elements of an array using bucket sort.

```
#include <stdio.h>
#include <conio.h>
void BucketSort(int a[ ], int n);
void InsertSort (int a[ ], int n);
void main( )
{
    int a[30], n, i;
    printf("\n enter no. of elements :");
    scanf("%d", &n);
    printf("\n enter data :");
    for(i = 0; i < n ; i++)
        scanf("%d", &a[i]);
    BucketSort(a, n);
    printf("\n sorted data :\");
    for(i = 0; i < n; i++)
    printf("%d \t", a[i]);
}
void BucketSort(int a[ ], int n)
{
    int bucket[10] [10], count[10], range[10];
    int large, small, i, step, j, k;
    large = small = a[0];
    /* find smallest and the largest element to find the
    range */
    for(i = 1; i < n ; i + +)
    {
        if (a[i] > large)
        large = a[i];
        if (a[i] < small)
        small = a[i];
    }
    step = (large – small)/10 + 1 ;
    for(i = 0; i < n; i + +)
    range[i] = i * step – 1;
    //storing data in buckets after initialization
    for(i = 0; i < n; i + +)
    count[i] = 0;
    for(i = 0; i < n; i ++)
    {   //locate the bucket
        j = 0;
        while(a[i] < = range[j])
        j ++;
        bucket[j] [count[j]] = a [j];
        count[j] = count[j] + 1;
    }
    //sort the individual buckets
    for(i = 0; i < n; i + +)
    InsertSort(bucket[i], count[i]);
    //gather data
    j = 0;
    for(i = 0; i < n; i + +)
    {
        k = 0;
        while(k < count[i])
        {
            a[i] = bucket[i] [k];
            k ++; j ++;
        }
    }
}
    void InsertSort(int a[ ], int n);
    {
        int i, j, temp;
        for(i = 1; i < n; i + +)
        {
            temp = a[i];
           for(j = i – 1; j >= 0 && a[j] > temp ; j – –)
            a[j + 1] = a[j],
            a[j + 1] = temp;
        }
    }
```

TechKnowledge Publications

5.10 Shell Sort

This method makes repeated use of insertion sort. If an array of n elements is to be sorted using shell sort then sorting requires that a number d_i called increment should be chosen before every pass. d_i should be less than n. d_i should diminish with every pass and for the last pass d_i should be 1.

Initial value of increments d_i can be taken as n/2 and in subsequent passes d_i should be halved.

i.e.

$$d_1 = n/2$$
$$d_2 = d_1/2$$
$$d_3 = d_2/2$$
$$d_{i+1} = d_i/2$$

If an array a[0] … a[7], has 8 elements then the initial value of increment $d_1 = 8/2 = 4$ with increment as 4, original array will be divided into four sub-lists :

1st sub list : (a [0], a[0 + 4])

2nd sub list : (a [1], a[1 + 4])

3rd sub list : (a [2], a[2 + 4])

4th sub list : (a [3], a[3 + 4])

Each of these lists is sorted independently using insertion sort.

In the second pass, the value of increment d_2 is taken as,

$d_2 = d_1/2 = 4/2 = 2$

Array will be divided into two sub lists :

1st sub list : (a[0], a[0 + 2], a[0 + 4], a[0 + 6])

2nd sub list : (a[1], a[1 + 2], a[1 + 4], a[1 + 6])

Each of these lists is sorted independently using insertion sort.

In the third pass the value of increment d_3 is taken as,

$d_3 = d_2/2 = 2/2 = 1$

Since, the value of increment is 1, entire array with elements (a[0], a[1], a[2] … a[6], a[7])

Will be sorted using insertion sort.

5.10.1 C Function for Shell Sort

Ex. 5.10.1 : The function shell sort, as given below, sometimes called diminishing increment sort, sorts an array a[n] of integers by sorting n/2 pairs (a[i], a[n / 2 + i]) for $0 \le i < n/2$ in the first pass, n/4 four-tuples (a[i], a[n/4 + i], a[n/2 + i], a[3 n / 4 + i]) for $0 \le i < n / 4$ in the second pass, n/8 eight-tuples in the third pass and so on. In each pas the sorting is done using insertion sort in which we stop sorting once we encounter two elements in the proper order.

Soln. :

```
void shellsort(int a[ ], int n)
{
    int i, j, step;
    int temp;
    for(step = n/2; step > 0; step = step/2)
    for(i = step; i < n; i++)
    {
        temp = a[i];
        for(j = i; j > = step; j = j – step)
            if(temp < a[j – step])
                a[j] = a[j – step];
            else
                break;
        a[j] = temp;
    }
}
```

Program 5.10.1 : Sorting of data using shell sort

```
#include <stdio.h>
#include <conio.h>
void shellsort(int a[ ], int n);
void main( )
{
    int a[30], i, n;
    printf("\n enter no. of elements :");
    scanf("%d", &n);
```

TechKnowledge Publications

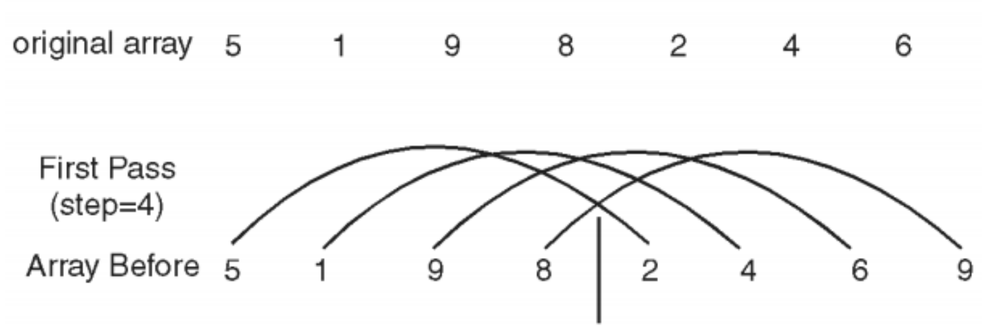
```
    printf("\n enter data :");
    for(i = 0; i < n ; i++)
        scanf("%d", &a[i]);
        shellsort(a, n);
        printf("\n sorted data :\n ");
    for(i = 0; i < n; i++)
        printf("%d/t", a[i]);
}
void shellsort(int a[ ], int n)
{
    int i, j, step;
    int temp;
    for(step = n/2; step > 0; step = step/2)
    for(i = step; i < n; i++)
    {
        temp = a[i];
        for(j = i; j > = step; j = j – step)
            if(temp < a[j – step])
                a[j] = a[j – step];
            else
                break;
        a[j] = temp;
    }
}
```

Output

```
enter no. of elements : 8
enter data : 1 5 2 4 6 11 3 7
sorted data : 1 2 3 4 5 6 7 11
```

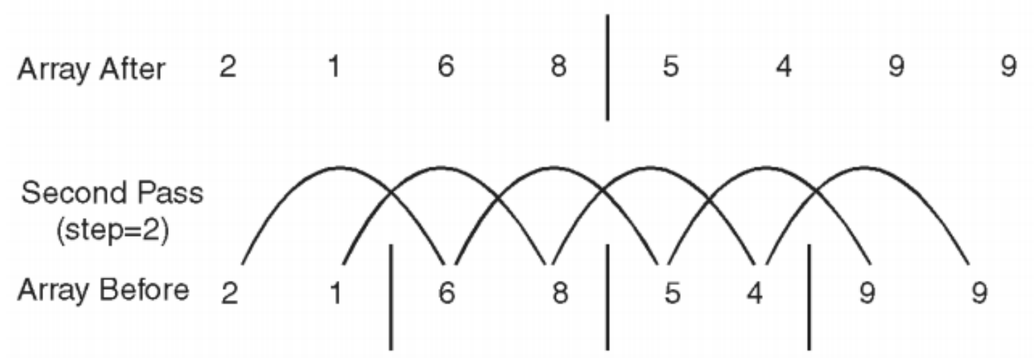
Ex. 5.10.2 : Sort the following sequence of integers (5 1 9 8 2 4 6 9) using shell sort.

Soln. :



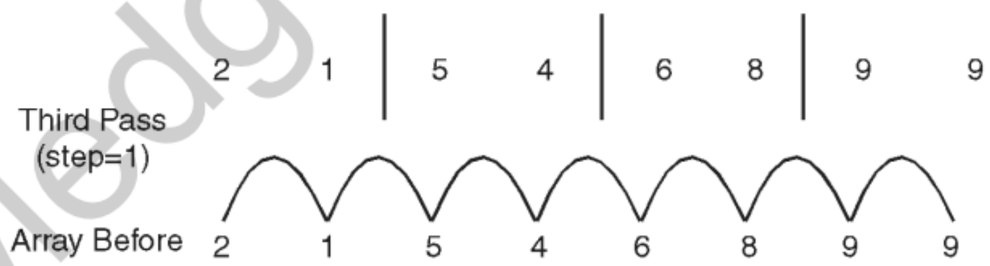
There are four groups, each group has 2 elements.

Elements of a group are sorted using insertion sort.



There are 2 groups, each group has 4 elements.

Elements of a group are sorted using insertion sort.



There is only one group, group has 8 elements.

Elements of the group are sorted using insertion sort.

Array after 1 2 4 5 6 8 9 9

Ex. 5.10.3 : Explain Shell sort and sort the given list using shell sort 08, 03, ,02, 11, 05, 14, 00, 02, 09, 04, 20 **SPPU - Dec. 16, 6 Marks**

Soln. :

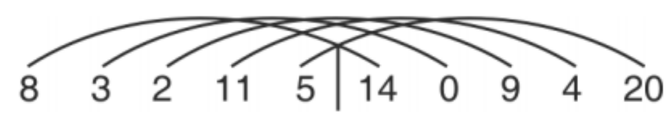
Sorting of given numbers using Shell sort

08, 03, 02, 11, 5, 14, 00, 09, 04, 20

Original array : 8 3 2 11 5 14 0 9 4 20

First pass : (Step 5)

Array before :



There are five groups each has 2 elements which are sorted independently using insertion sort.

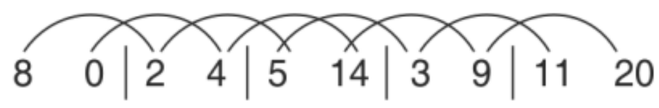
Array after :

8 0 2 4 5 14 3 9 11 20

Techknowledge Publication

Second pass : (Step 2)

Array before :



There are two groups each has 5 elements sorted using insertion sort.

Array after :

2 0 3 4 5 9 8 14 11 20

Third pass : (Step 1)

Array before :

There is only one groups with 10 elements, sorted using insertion sort.

Array after :

0 2 3 4 5 8 9 11 14 20

5.11 External versus Internal Sorting

- Internal sorting requires that input data should be stored in main memory.
- If the input data is too large to fit into memory, internal sorting cannot be used.
- External sorting algorithms take advantage of random access nature of primary memory.
- For example :
  a) Shell sort algorithm compares a[j] and a [j – step]
  b) Partition algorithm in quick sort compares a[i] and a[j].
- If the input is stored on the sequential device than comparison of two random elements will be very costly. Hence, shell sort or quick sort should not be used for external sorting.
- Some of the sorting algorithms are based on comparisons of adjacent elements. These algorithms can be used for external sorting. Merge sort is one of such algorithms with O(n log n) time complexity.

Examples of internal sorting :

(1) Quick sort

(2) Shell sort

(3) Radix sort

These algorithms have already been discussed in detail.

Examples of external sorting :

(1) Merge sort (2) Bubble sort

(3) Selection sort (4) Insertion sort

We will discuss merge sort for external sorting;

Suppose, input data is stored on tape T1 and there are additional three tapes which can be used for merging. If memory can hold maximum of 3 records then status of tapes after every run is shown in the example below.

Initial condition

T1	5	11	3	9	6	55	42	7	13	25	21	24	15
T2													
T3													
T4													

Data on tapes after 1st run

T1							
T2							
T3	3	5	11	7	13	42	15
T4	6	9	55	21	24	25	

Data on tapes after 2nd run

T1	3	5	6	9	11	55	15
T2	7	13	21	24	25	42	
T3							
T4							

Techknowledge Publication

Data on tapes after 3rd run

T1												
T2												
T3	3	5	6	7	9	11	13	21	24	25	42	55
T4	15											

Data on tapes after 4th run

T1	3	5	6	7	9	11	13	15	21	24	25	42	55
T2													
T3													
T4													

Ex. 5.11.1 : "External sorting calls for internal sorting as well". Justify.

Soln. :

Every sorting algorithm is based on passes. Inside a pass, records are compared on key values and the records can be shuffled based on the outcome of comparison. In case of external sorting, records to be compared are read into memory variables and they are re-written at appropriate places.

Pseudo code for bubble sort for external sorting is given to explain the above concept.

```
n ← number of records in the file ;
rec1, rec2, : record type ;
for(i = 1 ; i < n ; i ++)
    for(j = 0 ; j < n – i ; j ++)
    {
        rec1 ← read the jth record ;
        rec2 ← read the( j + 1)th record ;
        if(rec1 · key > rec2 · key)
        {
            write rec2 at jth place ;
            write rec1 at( j + 1) th place ;
        }
    }
```

5.12 Comparison of Sorting Algorithms

Q. Compare bubble sort, insertion sort and selection sort with one example and discuss time complexity.

SPPU - May 17, 3 Marks

Sorting algorithm	Efficiency	Passes	Sort stability
Bubble sort	$O(n^2)$	n – 1	Stable
Selection sort	$O(n^2)$	n – 1	Stable
Insertion sort			
Best case	O(n)	n – 1	Stable
Worst case	$O(n^2)$	n – 1	
Quick sort			
Best case	O(n log n)	log n	Unstable
Worst case	$O(n^2)$		
Merge sort	O(n log n)	log n	Stable
Radix sort / Counting sort	O(n)	No. of digits in the n – 1 largest number	Stable

TechKnowledge Publications

5.13 Best-case, Worst-case and Average-case Analysis of Sorting Algorithm

	Best-case	Worst-case	Average-case
Bubble sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion sort	O(n)	$O(n^2)$	$O(n^2)$
Quick sort	O(n log n)	$O(n^2)$	O(n log n)
Merge sort	O(n log n)	O(n log n)	O(n log n)
Radix sort	O(n)	O(n)	O(n)

If the sorting algorithm is not data sensitive, its best case, worst case and average case timing behaviour will be same.

Bubble sort, selection sort, merge sort and radix sort algorithms are not data sensitive. Quick sort and shell algorithms are data sensitive.

5.14 Case Study - Timesort is a Hybrid Stable

Sorting Algorithm

It is not a standalone algorithm. It is a combination of several algorithms, which are utilized to give a better efficiency. The basic mechanism of the algorithm is explained below :

1. The array containing N elements is split into sub-arrays.
2. Each sub-array is sorted using insertion sort.
3. These sub-arrays are merged into a single-sorted array using the technique used in merge sort.

The efficiency of time sort algorithm is given below :

Best case : O(n)

Average case : O(n log n)

Worst case : O(n log n)

This algorithm came into existence in the year 2002. It has a standard sorting algorithm in python, open JDK 7 and Android JDK 1.5 even since. The algorithm can exploit the partially sorted nature of data using the insertion sort.

Review Questions

Q. 1 Write pseudo C code for bubble sort. **(10 Marks)**

Q. 2 Write and explain worst case input for quick sort to sort list of numbers in ascending order. State worst case time complexity. **(8 Marks)**

Q. 3 Explain the following terms : **(6 Marks)**

(i) Internal sorting

(ii) External sorting

Q. 4 Explain the importance of sorting technique in computer science field. What is sort stability? **(4 Marks)**

Q. 5 What is the importance of pivot element in the quick sort method ? **(2 Marks)**

Q. 6 Comment on time and space complexity in best, average and worst cases for insertion sort. **(6 Marks)**

Q. 7 Define the following terms with respective sorting.

(i) Sort stability (ii) Efficiency (iii) Passes **(4 Marks)**

Technical Knowledge Publication

CHAPTER 6

Unit IV

Linked List

Syllabus :

Introduction to Static and Dynamic Memory Allocation, Linked List : Introduction, of Linked Lists, Realization of linked list using dynamic memory management, operations, Linked List as ADT, Types of Linked List : singly linked, linear and Circular Linked Lists, Doubly Linked List, Doubly Circular Linked List, Primitive Operations on Linked List-Create, Traverse, Search, Insert, Delete, Sort, Concatenate. Polynomial Manipulations-Polynomial addition. Generalized Linked List (GLL) concept, Representation of Polynomial using GLL.

Exemplar/Case Studies : Garbage Collection

6.1 Representation and Implementation of Singly Linked Lists

6.1.1 Comparison between Array and Linked Lists

Q. Compare sequential and linked organisation of data.

SPPU - Dec. 18, 2 Marks

Array data structure is simple to use and it is supported by almost all programming languages. It is very simple to understand and time to access any element from an array is constant.

- Simple to use
- Simple to define
- Constant access time
- Mapping by compiler

} Properties of array data structure

An array element can be accessed by a[i], where 'a' is the name of the array and 'i' is the index.

Compiler maps a[i] to its physical location in memory. Address of a[i] is given by starting address of a + i ∗ size of array element in bytes.

This mapping is carried out in constant time, irrespective of which element is accessed. Array data structure suffers from some severe limitations :

- Size of an array is defined at the time of programming.
- Insertion and deletion is time consuming.
- Requires contiguous memory.

First, the size of an array has to be defined when the program is being written and its space is calculated during compilation of the program. This means that the programmer has to take a decision regarding the maximum size of data. If the actual amount of data stored is less than the maximum size, a good amount of memory will be wasted and on the other hand a larger sample of data cannot be handled. To avoid the linear cost of insertion and deletion, we need to ensure that the list is not stored contiguously, since otherwise entire parts of list will need to be moved.

Advantages of linked lists

(i) Linked list is an example of dynamic data structure. They can grow and shrink during execution of the program.

(ii) Representation of linear data structure. (Inline data like polynomial, stack and queue can easily be represented using linked list).

Technknowledge Publication

(iii) Efficient memory utilization. Memory is not pre-allocated like static data structure. Memory is allocated as per the need. Memory is deallocated when it is no longer needed.

(iv) Insertion and deletions are easier and efficient. Insertion and deletion of a given data can be carried out in constant time.

6.1.2 Dynamic Memory Management

In case of dynamic data structures, the memory space required by variables is calculated and allocated during execution. C gives us two choices for reserving memory locations for a variable :

1. Static 2. Dynamic

A conceptual view of memory is shown in Fig. 6.1.1.

- main( ) must be in memory all times.
- Each called function must be in memory while it is active.

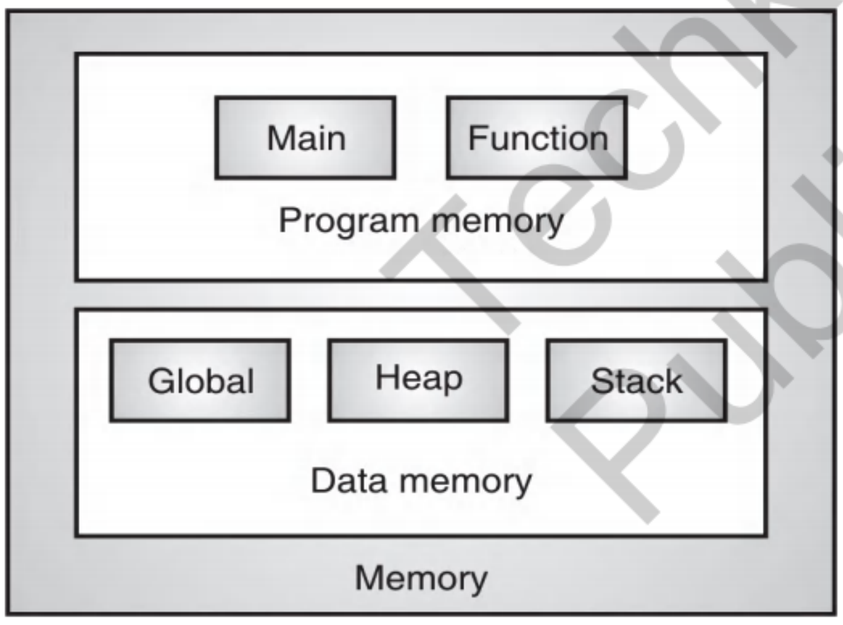


Fig. 6.1.1 : Conceptual view of memory

- To support recursive calling of a function multiple copies of its local variables are stored on top of the stack. This memory is also known as stack memory.
- Heap memory is unused memory and memory from it is allocated during program execution with the help of memory allocation functions.
- Memory for global data is allocated at the time of loading the program.

6.1.3 Memory Management Function during Runtime

These functions are shown in the Fig. 6.1.2.

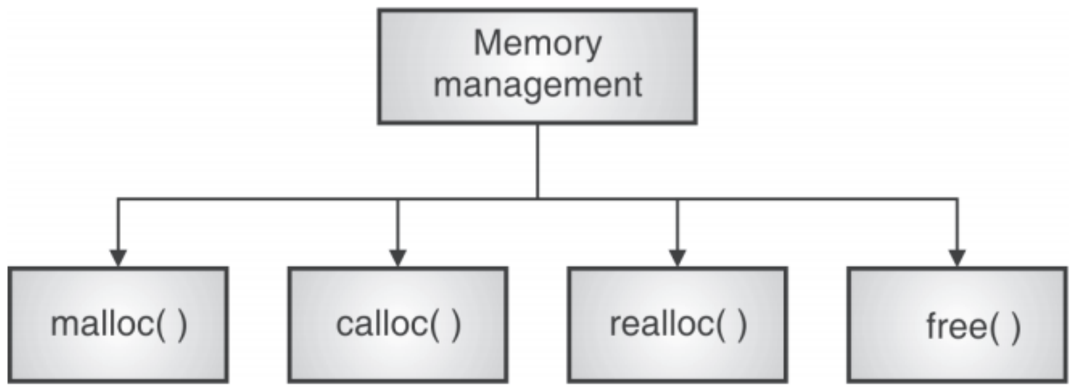


Fig. 6.1.2 : Memory management functions

1. Block Memory Allocation (malloc( ))

The malloc( ) function allocates a block of memory that contains the number of bytes specified in its parameter. The usage of malloc( ) function is shown below :

```
Ptr = (cast _ type) malloc(byte _ size);
```

The malloc( ) function returns a pointer (of cast _ type) to an area of memory with size, byte-size.

2. Contiguous Memory Allocation (calloc( ))

This function can be used to allocate a contiguous block of memory, primarily for arrays. It differs from malloc( ) only to the extent that it sets memory to null characters. The usage of calloc( ) is shown below :

```
Ptr = (cast_type)
(element_count, size_of_each_element);
```

The calloc( ) function returns a pointer to the allocated memory.

3. Reallocation of Memory (realloc( ))

This function changes the size of the previously allocated block of memory. This is done by either deleting or extending the memory at the end of the block. If memory cannot be extended, realloc( ) allocates a completely new block of memory. The usage of realloc( ) function is shown below :

```
Ptr = realloc(ptr, newsize);
```

Tech Knowledge Publications

4. Releasing Memory (free( ))

When the allocated memory is no longer needed, it should be returned back by using the function free( ). The usage of the function is shown below :

```
free(ptr);
```

6.1.4 Representation

Q. Define Linked List. **SPPU - May 19, 1 Mark**

The linked list consists of a series of structures. They are not required to be stored in adjacent memory locations. Each structure consists of a data field and address field. Address field contains the address of its successors. Fig. 6.1.3 shows the actual representation of the structure.

Fig. 6.1.3 : Representation of the structure

A variable of the above structure type is conventionally known as a **node**. Fig. 6.1.4 gives a representation of a linked list of three nodes.

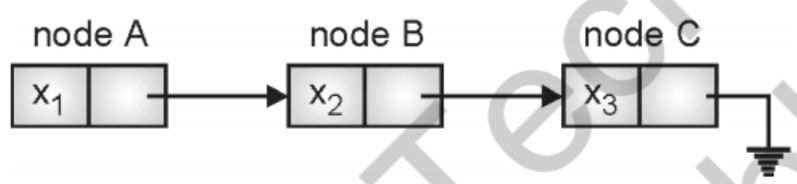


Fig. 6.1.4 : Linked list

A list consisting of three data x_1, x_2, x_3 is represented using a linked list. Node A stores the data x_1 and the address of the successor (next) node B. Node B stores the data x_2 and the address of its successor node C. Node C contains the data x_3 and its address field is grounded (NULL pointer), indicating it does not have a successor.

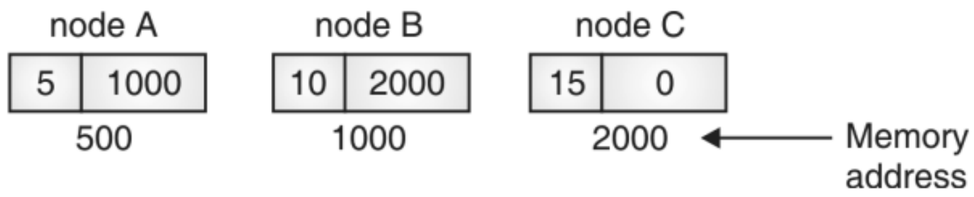


Fig. 6.1.5 : Memory representation of a linked list

Fig. 6.1.5 gives a memory representation of the linked list shown in Fig. 6.1.4.

Nodes A, B and C happen to reside at memory locations 500, 1000 and 2000 respectively. $x_1 = 5$, $x_2 = 10$ and $x_3 = 15$. Node A resides at the memory location 500, its data field contains a value 5 and its address field contains 1000, which is the address of its successor node. Address field of node C contains 0 as it has no successor.

6.1.5 Implementation

Structures in "C" can be used to define a node. Address of the successor node can be stored in a pointer type variable.

```
typedef struct node
{
    int data;
    struct node *next;
}node;
```

It is a **self referential structure** in "C". It is assumed that the type of data to be stored in each node is predefined to be of integer type. Next field of the structure is a pointer type variable, used for storing address of its successor node. Nodes are manipulated during run-time. A programming language must provide following facilities for run time manipulation of nodes.

- Acquiring memory for storage during run-time.
- Freeing memory during execution of the program, once the need for the acquired memory is over.

In "C" programming language, memory can be acquired through the use of standard library functions malloc( ) and calloc( ). Memory acquired during runtime can be freed through the use of library function free( ).

```
/* 1 */    node *P;
/* 2 */    P = (node *) malloc(sizeof(node));
/* 3 */    P → data = 5;
/* 4 */    P → next = NULL
```

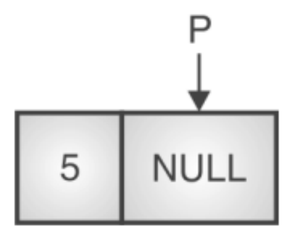


Fig. 6.1.6 : Memory for a node has been allocated during run-time. Its address is stored in the pointer P

TechKnowledge Publications

In the program segment above, the line 1 declares a variable P with the storage class node ∗. It can store the address of a node that has been created dynamically. sizeof (node) in line 2 is storage requirement in number of bytes to store a node. Malloc (sizeof(node)) returns the address of the allocated memory block and it is assigned to variable P. The address returned by malloc (sizeof(node)) is type casted using type casting operator(node ∗) before assigning it to the pointer P.

P → data = 5, in line 3 stores a value of 5 in the data field of node whose address is stored in pointer P. P → next = NULL stores a value of NULL in the next field of the node whose address is stored in the pointer P.

A linked list with the address of the head node

As an array is referenced by its starting address, a linked list is known by the address of its head node. Address of the head node is stored in a pointer variable (node ∗ head) head.

All manipulations on the linked list can be performed through the address of the starting node. Through the variable "head", first node can be accessed and through the address of the second node stored in the next field of the first node, second node can be accessed and so on.

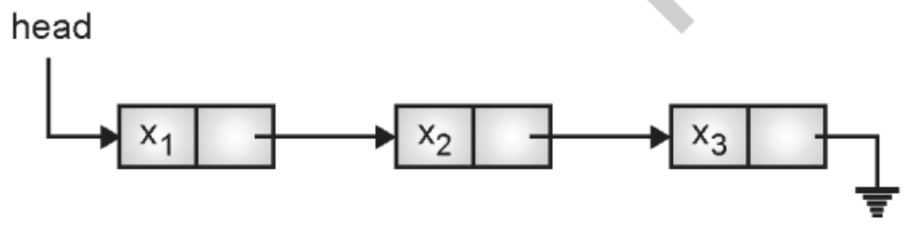


Fig. 6.1.7 : A linked list with the address of the head node

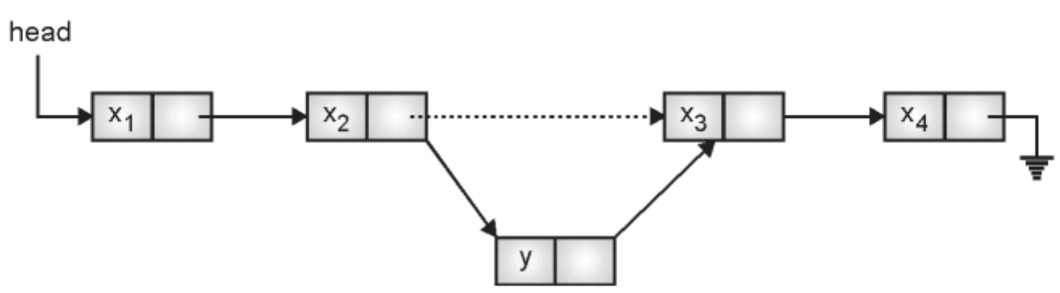


Fig. 6.1.8 : Insertion into a linked list

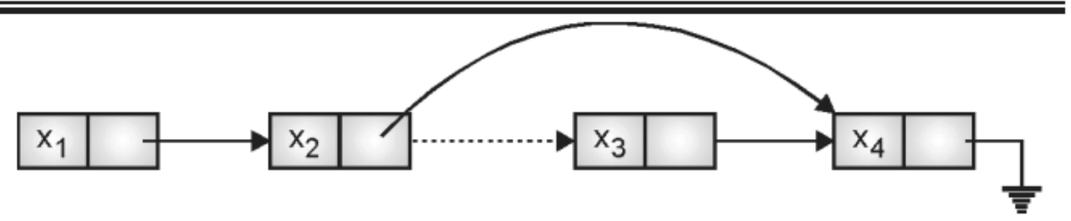


Fig. 6.1.9 : Deletion from a linked list

Insertion into a linked list requires obtaining a new node and then changing values of two pointers. The general idea is shown in Fig. 6.1.8. The dashed line represents the old pointer. It is changed to point to new node. Deletion of a node can be performed in one pointer change. Fig. 6.1.9 shows the result of deleting the node containing x_3. The next field of the node containing x_2 is changed to point to the node containing x_4. Node containing x_3 is freed using the library function free( ) subsequently.

A linked list with header node

A linked list with header node is shown below :

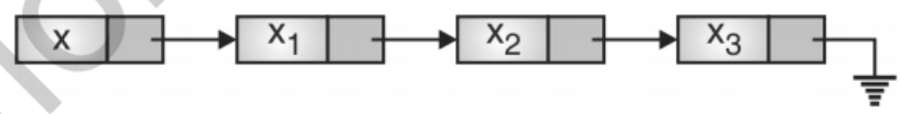


Fig. 6.1.10

Header node is always at the beginning with header node, algorithms for insertion and deletion become easy as otherwise, two cases must be considered :

1. Insertion in empty list, deletion of two only (last) element.
2. Insertion if the list is not empty, deletion if the size > 1.

6.1.6 Types of Linked List

6.1.6(A) Singly Linked List

In this type of linked list two successive nodes of the linked list are linked with each other in sequential linear manner. Movement in forward direction is possible.

TechKnowledge Publications

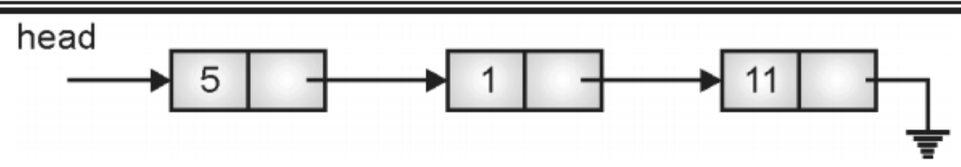


Fig. 6.1.11 : Singly linked list

6.1.6(B) Doubly Linked List

In this type of linked list each node holds two-pointer fields. In doubly linked list addresses of next as well as preceding elements are linked with current node.

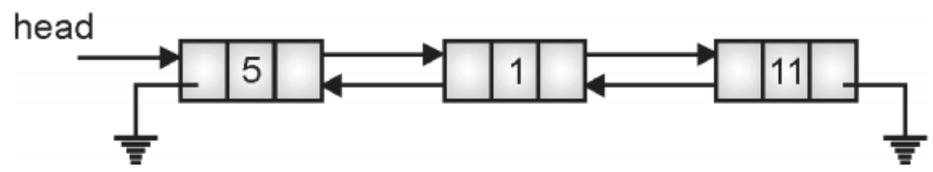


Fig. 6.1.12 : Doubly linked list

6.1.6(C) A Circular Linked List

In a circular list the first and the last elements are adjacent. A linked list can be made circular by storing the address of the first node in the next field of the last node.

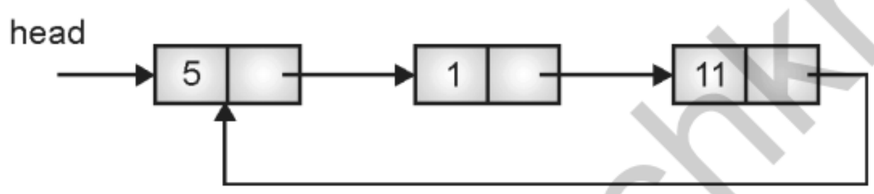


Fig. 6.1.13 : Circular linked list

Ex. 6.1.1 : Compare linked list with arrays with reference to the following aspects :

(i) Accessing any element randomly

(ii) Insertion and deletion of an element

(iii) Utilization of computer memory

Soln. :

Sr. No.	Aspects/ parameters	Array	Linked list
1.	Accessing any element randomly	Fast	Slow
2.	Insertion and deletion of an element	Slow	Fast
3.	Utilization of computer memory	No reusability	Reusability

6.2 Basic Linked List Operations

We can treat a linked list as an abstract data type and perform following basic operations.

1. Creating a linked list.
2. Traversing the linked list.
3. Printing the link list.
4. Counting the nodes in the linked list.
5. Searching an item in the linked list.
6. Inserting an item.
7. Deleting an item.
8. Concatenating two lists.
9. Inversion.
10. Sorting of elements stored in a linked list.
11. Merging of two sorted linked list.
12. Separating a linked list in two linked lists.

Ex. 6.2.1 : Explain linked list as an ADT.

OR Write pseudo C/ C++ code to represent Singly linked list as an ADT. **SPPU - Dec. 17, 6 Marks**

Soln. :

Data type for linked list as an ADT (data is assumed to be of integer type)

```
{
    int data;
    struct node * next;
} node;
```

Operations on linked list

(i) Create (ii) Delete (iii) Insert

(iv) Make null (v) Search (vi) Print

(vii) Sort

Prototype of functions used for various operations

(i) node ∗ create( );

(ii) node ∗ delete(node ∗ head, int location);

(iii) node ∗ insert(node ∗ head, int data, int location)

(iv) node ∗ make*** (node ∗ head);

Techknowledge Publications

(v) node * search(node * head, int data);

(vi) void print(node * head);

(vii) void sort(node * head);

6.2.1 Creating a Linked List

Program 6.2.1 : Program to create linked list.

```
#include<conio.h>
#include<stdio.h>
typedef struct node
{
    int data;
    struct node *next;
}node;
void main()
{
    node *HEAD, *P;
    int n, x, i;
    //number of items to be inserted
    printf("\n number of items :");
    scanf("%d", &n);
    //get the 1'st node with its address in HEAD
    HEAD = (node*)malloc(sizeof(node));
    //read the data in 1'st node
    scanf("%d", &(HEAD->data));
    HEAD->next = NULL;
    /* HEAD points to the 1st node, while P points to
       the last node */
    P = HEAD;
    /*  In case of single node 1st and last node are
       same insert the remaining nodes */
    for(i = 1; i<n; i++)
    {
        P->next = (node*)malloc(sizeof(node));
        //new node is inserted as the next node after P
        P = P->next;
        P->next = NULL;
        scanf("%d", &(P->data));
    }
}
```

– Let us trace the working of the above program manually.

– Assume n = 3 (3 nodes to be created) with inputs as 5, 1, 9.

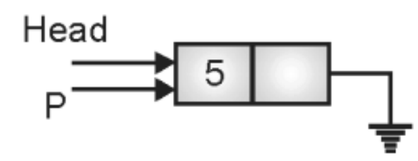


Fig. 6.2.1 : The linked list after insertion of the first item

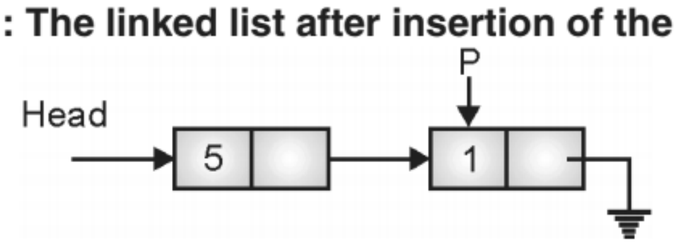


Fig. 6.2.2 : The linked list after first iteration of the "for loop"

The address of the newly acquired node is stored in the next field of the node pointed by P. Subsequently, P is moved to the next node.

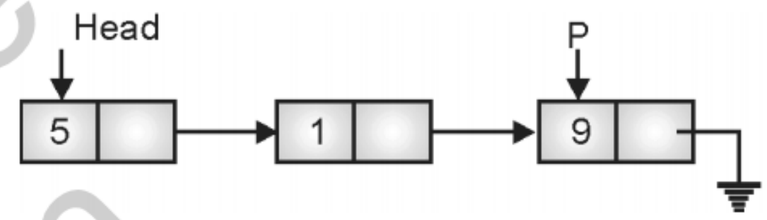


Fig. 6.2.3 : The status of the linked list after insertion of 3 elements

Program 6.2.2 : Program to create linked list through 'create' function.

```
#include<conio.h>
#include<stdio.h>
typedef struct node
{
    int data;
    struct node *next;
}node;
node * create(int);
void main()
{
    node *HEAD;
    int n;
    HEAD = NULL;        //link list is empty
    printf("\n no. of items :");
    scanf("%d", &n);
    HEAD = create(n);
    //create function returns the address of first node
```

Tech Knowledge Publication

```
}
node * create(int n)
{
    node *head, *P;
    int i;
    head = (node*)malloc(sizeof(node));
    head->next = NULL;
    scanf("%d", &(head->data));
    P = head;   //insert the remaining nodes
    for(i = 1; i<n; i++)
    {
        P->next = (node*)malloc(sizeof(node));
        //new node is inserted as the next node after P
        P = P->next;
        scanf("%d", &(P->data));
        P->next = NULL;
    }
    return(head);
}
```

6.2.2 Traversing a Linked List

Traversal of a linked list always starts from the first node. In order to traverse a linked list in the forward direction, a pointer type variable is assigned the address of the first node.

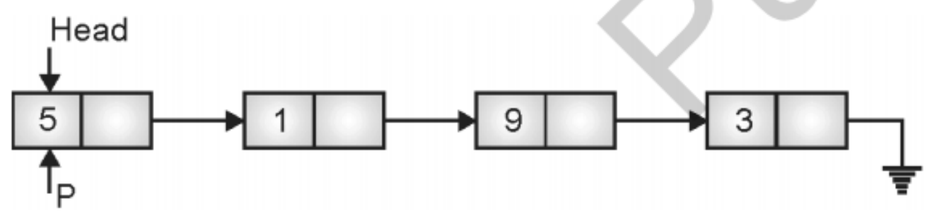


Fig. 6.2.4 : Traversal starts from the first node

```
P = HEAD;
```

Entire list can be traversed through the following program segment.

```
P = HEAD;
while(P! = NULL)
    P = P → next;
/*  Next field of the node pointed by P contains the
    address of the next node */
```

6.2.3 Counting Number of Nodes in a Linked List through Count Function

```
int count(node *P)
{
    int i;
    i = 0;
    while(P ! = NULL)
    {
        i = i + 1;
        P = P → next;
    }
    return(i);
}
```

Program 6.2.3 : Program to create linked list interactively and print the list and total number of items in the list.

OR Write a C/C++ program to create singly linked list of integers and display it forward.

```
#include<conio.h>
#include<stdio.h>
typedef struct node
{
    int data;
    struct node *next;
}node;
node * create(int);
void print(node *);
int count(node *);
void main( )
{
    node *HEAD;
    int n, number;
    printf("\n Number of items :");
    scanf("%d", &n);
    HEAD = create(n);
    //create function returns the address of first node
    print(HEAD);
    number = count(HEAD);
    printf("\n Number of nodes = %d", number);
}
node * create(int n)
```

Techknowledge Publications

```
{
    node *head, *P;
    int i;
    head = (node*)malloc(sizeof(node));
    head->next = NULL;
    scanf("%d", &(head->data));
    P = head;
    //create subsequent nodes
    for(i = 1; i<n; i++)
    {
        P->next = (node*)malloc(sizeof(node));
        //new node is inserted as the next node after P
        P = P->next;
        scanf("%d", &(P->data));
        P->next = NULL;
    }
    return(head);
}
}
void print(node *P)
{
    while(P! = NULL)
    {
        printf("<- %d ->", P->data);
        P = P->next;
    }
}
int count(node *P)
{
    int i = 0;
    while(P! = NULL)
    {
        P = P->next;
        i++;
    }
    return(i);
}
```

Output

```
Number of items :4
12
13
14
15
<- 12 -> <- 13 -> <- 14 -> <- 15 ->
Number of nodes = 4
```

6.2.4 Printing a List through Print Function

```
void print(node *P)
{
   while(P! = NULL)
    {
        printf("\n%d", P → data);
            P = P → next;
    }
}
```

Above function traverses a linked list and while traversing it prints the integer data stored in the node.

6.2.5 Inserting an Item

Inserting a new item, say x, has three situations

1. Insertion at the front of the list.
2. Insertion in the middle of the list.
3. Insertion at the end of the list.

Algorithm for placing the new item at the beginning of a linked list

1. Obtain space for new node.
2. Assign data to the data field of the new node.
3. Set the next field of the new node to the beginning of the linked list.
4. Change the reference pointer of the linked list to point to the new node.

For inserting the new data after a node N1

1. Obtain space for new node.
2. Assign value to its data field.
3. Search for the node N1.
4. Set the next field of the new node to point to N1 → next.
5. Set the next field of N1 to point to the new node.

Techknowledge Publication

'C' function to insert a data in a linked list after a given data.

```
node *insert(node *head, int x, int key)
{ /* data x is to be inserted after the key,
     if key is –1then x is to be inserted as a front
     node */
    node *P, *q;
    /* obtain space for the new node */
    P = (node *) malloc(sizeof(node));
    /* store x in the new node */
    P → data = x;
    if(key == – 1)
    {
        /* insert the node at the front of the list */
        P → next = head;
        head = P;
    }
    else
    {
        /* search for the key in the linked list */
        q = head;
        while(key! = q → data && q! = NULL)
            q = q → next;
        if(q ! = NULL)
        {
        /* if the key is found */
        P → next = q → next;
        q → next = P;
    }
  }
  return(head);
}
```

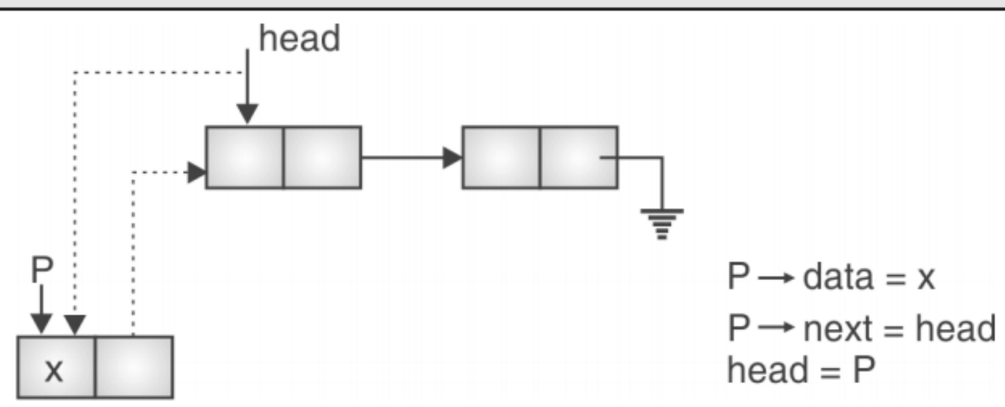


Fig. 6.2.5 : Insertion at the front

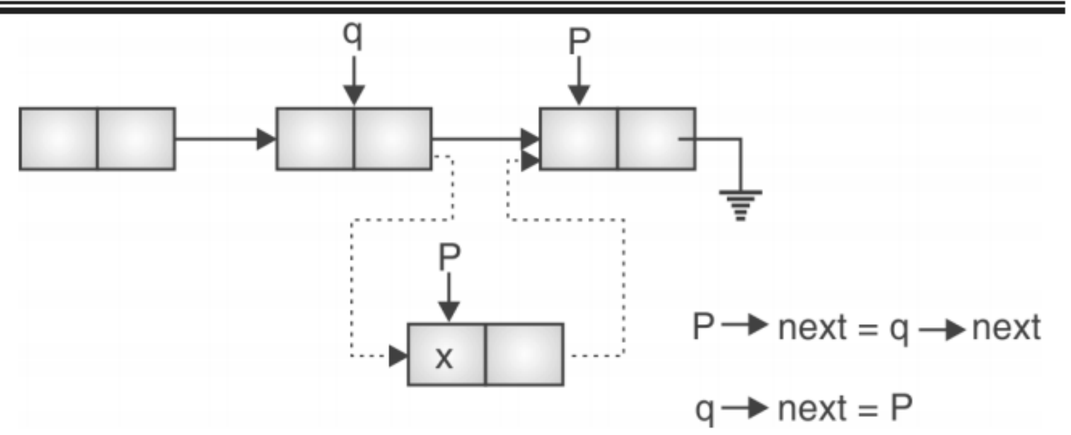


Fig. 6.2.6 : Insertion in between

6.2.5(A) Inserting an Item at the End of a Linked List

Algorithm

[Insert an item 'x' in a linked list, referenced by the pointer 'head']

1. Acquire memory for new node

   i.e. p = (node*) malloc(sizeof (node));

2. Assign value to the data filed and make its 'next' field 'NULL'

   i.e. P → data = x;

   P → next = NULL;

3. If 'head' is 'NULL'

   then

   [head = p ;

   goto step 6]

4. Position a pointer q on the last node by traversing the linked list form the first node and until it reaches the last node.

   i.e. q = head

   while (q → next ! = NULL)

   q = q → next ;

5. Store the address of the newly acquired node, pointed by P, in the next field of node pointed by q.

   i.e. q → next = p;

6. Stop.

'C' function for inserting an element 'x' at the end of linked list, referenced by head.

```
node *insert_end(node *head, int x)
{
```

Technknowledge Publication

```
    node *p, *q;
    p = (node *)malloc(sizeof(node));
    p->data = x;
    p->next = NULL;
    if(head == NULL)
        return(p);
    q = head;
    while(q->next! = NULL)
        q = q->next;
    q->next = p;
    return(head);
}
```

6.2.5(B) Inserting a Data 'x' at a given Location 'LOC' in a Linked List, Referenced by 'head'

Q. Write a pseudo C/C++ code to insert node into a singly linked list. **SPPU - May 18, 3 Marks**

Algorithm

1. Acquire memory for new node with its address in pointer p.

   i.e. p = (node *) malloc(sizeof(node));

2. Assign value to data field and make its 'next' field 'NULL'.

   i.e. p → data = x;

   p → next = NULL;

3. if (LOC == 1)

   then

   [insert the node, pointed by P, at the beginning of the linked list. This can be done by following steps.]

   (a) Store head in the next field of the node pointed by p

   i.e. p → next = head;

   (b) Move head to the newly connected node.

   i.e. head = p;

   goto step 7

4. [If LOC > 1]

   Position a pointer q on $(LOC - 1)^{th}$ node .

   i.e. q = head;

   for (i = 1; i < (LOC – 1); i++)

   q = q → next;

5. if q is NULL

   [then report " overflow" and terminate the algorithm.

   goto step 7]

6. Insert the node pointed by p, after the node pointed by q.

   i.e. p → next = q → next;

   q → next = p;

7. Stop.

'C' function for inserting an element x at the location 'LOC' in a linked list, referenced by head.

```
node *insert_LOC(node *head, int x, int LOC)
{
    node *p, *q;
    int i;
    p = (node *)malloc(sizeof(node));
    p->data = x;
    p->next = NULL;
    if(LOC == 1)
    {
        p->next = head;
        return(p);
    }
    q = head;
    for(i = 1; i<LOC–1; i++)
        if(q! = NULL)
            q = q->next;
        else
        {
            printf("\n Overflow");
            return(head);
        }
    p->next = q->next;
```

Technknowledge Publication

```
    q->next = p;
    return(head);
}
```

Program 6.2.4 : Write a C program to create singly linked list of integers, display the same and count Number of even elements in the list (use functions).

```
#include < stdio.h>
#include <conio.h>
#include <stdlib.h>
typedef struct node
{
    int data;
    struct node *next;
} node;
node *create();
void print(node *head);
int count_even(node *head);
void main()
{
    node *head;
    int result;
    head = create;
    print(head);
    result = count_even(head);
    printf("\n No. of even data = %d", result);
    getch();
}
node *create()
{
    node *head, *p;
    int i, n;
    head = NULL;
    printf("\n Enter number of data:");
    scanf("%d", &n);
    printf("\n Enter the data:");
    for(i = 0; i<n; i++)
    {
        if(head == NULL)
            p = head = (node*)malloc(sizeof(node));
        else
        {
            p->next = (node*)malloc(sizeof(node));
            p = p->next;
        }
        p->next = NULL;
        scanf("%d", &(p->data));
    }
    return(head);
}
void print(node *head)
{
    node *p;
    printf("\n \n ");
    for(p = head; p! = NULL; p = p->next)
    printf("%d  ", p->data);
}
int count_even(node * head)
{
    int s = 0;
    for(; head! = NULL; head = head→next)
        if(head→data%2 == 0)
            s++;
    return(s);
}
```

Ex. 6.2.2 : Explain role of stack in recursion and implement the following using a recursive functions.

(i) Display singly linked list reverse

(ii) Count no. of nodes in singly linked list.

Soln. : Nested calls of functions are managed through stack. When a function is invoked, memory is allocated on stack for,

- Local variables
- Return address

On termination of execution of the function, memory is de-allocated. Details regarding the called function are stored in a specific structure known as Activation record. An activation record consists of :

1. Storage space for all local variables
2. Definition of function
3. Return address
4. A pointer to activation record

Techknowledge Publication

Activation records are stored on the stack. When a function is called, its activation record is pushed into the stack and control is transferred to the called Program. When the function execution completes, the control returns to the caller function. It obtains the return address from the return address field of the activation record.

```
void display_reverse(node *h)
{
    if(h ! = NULL)
    {
        display_reverse(h→ next);
        printf("%d", h→data);
    }
}
int count(node *h)
{
    if(h == NULL)
        return(0);
    return(1 + count(h→next));
}
```

6.2.5(C) Inserting an Element in a Priority Linked List

Inserting an element 'x' in a sorted list of integers represented using a linked list. List should remain sorted after insertion.

OR

Insert an element 'x' in a priority linked list. Elements are ordered on increasing priority.

Algorithm

[Existing linked list is referenced by the pointer 'head'. New data to be inserted is 'x'.]

1. Acquire memory for new node.

   i.e. p = (node *) malloc(sizeof(node));

2. Assign value to its data field and make its next field 'NULL'.

   i.e. p → data = x;

   p → next = NULL;

3. If head is NULL or x is less than the data stored in the first node (i.e. node pointed by 'head')

   then

   [insert the node P at the beginning of the linked list and terminate the algorithm]

   if (head == NULL || x < head → data)

   {

   p → next = head;

   head = p;

   goto step 5

   }

4. [If head is not NULL and x > head → data]

   Locate the point of insertion, as per the priority of x.

   i.e.

   q = head;

   while(q→next! = NULL && x>q→next→data)

   q = q → next;

   x should be inserted after the node pointed by q.

   i.e. p → next = q → next;

   q → next = p

5. Stop.

'C' function to insert an element in priority linked list.

```
node *insert_priority(node *head, int x)
{
    node *p, *q;
    p = (node *)malloc(sizeof(node));
    p->data = x;
    p->next = NULL;
    if(head == NULL || x<head->data)
    {
        p->next = head;
        head = p;
        return(head);
    }
    q = head;
```

Tech Knowledge Publications

```
    while(q->next! = NULL && x>q->next->data)
        q = q->next;
    p->next = q->next;
    q->next = p;
    return(head);
}
```

6.2.6 Deleting an Item

Q. Write a pseudo C/C++ code to delete intermediate node from singly linked list.

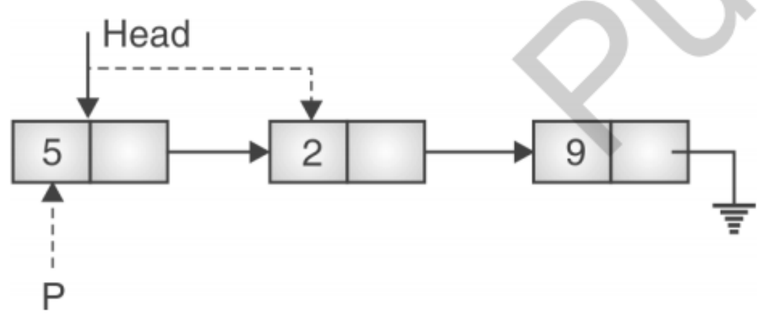
SPPU - Dec. 17, 3 Marks

Deleting a node from the list is even easier than insertion, as only one pointer value needs to be changed. Here again we have three situations.

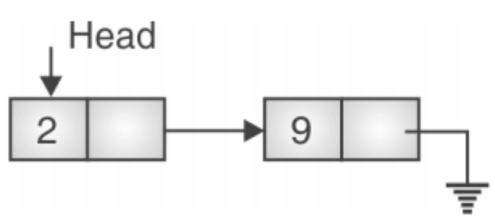
1. Deleting the first item.
2. Deleting the last item.
3. Deleting from the middle of the list.

Algorithm for deleting the first item.

1. Store the address of the first node in a pointer variable, say P.
2. Move the head to the next node.
3. Free the node whose address is stored in the pointer variable P.



(a) Before deletion



(b) After deletion

Fig. 6.2.7 : Deleting the first item

Algorithm for deleting a node from the middle of the linked list

1. Store the address of the preceding node in a pointer variable P. Node to be deleted is marked as key node.
2. Store the address of the key node in a pointer variable q, so that it can be freed subsequently.
3. Make the successor of the key node as the successor of the node pointed by P.
4. Free the node whose address is stored in the pointer variable q.

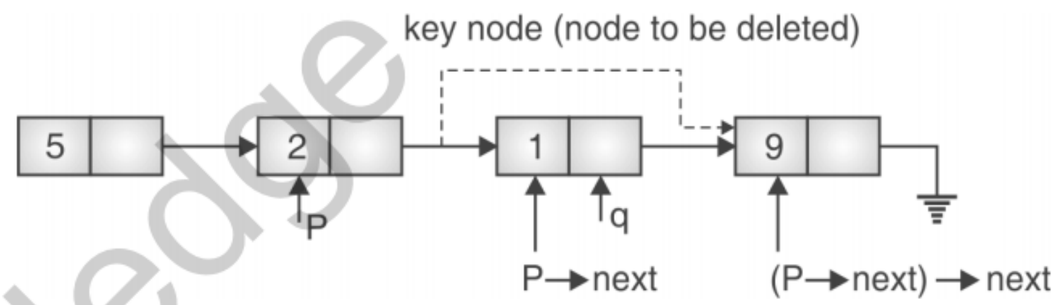


Fig. 6.2.8 : Deletion of middle element

'C' function to delete a data from a linked list.

```
node * delete(node *head, int x)
/* x to be deleted from a linked list */
{
  node *P, *q;

      if(x == head → data)
      {
          /* deleting the first item */
          P = head;
          head = head → next;
          free(P);
      }
      else
      {
       while(x!=(P→next)→data && P→next! = NULL)
          P = P → next;
          if(P → next ! = NULL)      /* if x exists */
          {
              q = P → next;
              P → next = (P → next) → next;
              free(q);
               /* release space of the key node */
          }
```

TechKnowledge Publications

```
        }
        return(head);
}
```

6.2.6(A) Deletion of the Last Node of a Linked List

Algorithm

[Deleting last node of a linked list, referenced by the pointer head]

In order to delete the last node, we must position a pointer q on the last but one node. Address of the node to be deleted is stored in pointer p, So that the memory allocated to it can be freed.

1. If the first node itself is the last node then

   [make the linked list empty]

   i.e. if (head → next == NULL)

   {

   free (head);

   head = NULL; goto step 4

   }

2. [otherwise]

   position a pointer q on last but one node

   i.e. q = head;

   while (q → next → next ! = NULL)

   q = q → next;

3. Delete the last node

   i.e. p = q → next ;

   free (p);

   q → next = NULL;

4. Stop.

'C' function to delete last node of a linked list.

```
node *delete_last(node *head)
{
    node *p, *q;
    if(head->next == NULL)
    {
        free(head);
        head = NULL;
        return(head);
    }
    q = head;
    while(q->next->next! = NULL)
        q = q->next;
    p = q->next;
    free(p);
    q->next = NULL;
    return(head);
}
```

6.2.6(B) Deletion of a Node at Location 'LOC' from a Linked List

Algorithm to delete a node from singly linked list

[Linked list is referenced by the pointer 'head']

1. if (LOC == 1)

   then

   [the node to be deleted is the first node This can be done by following steps]

   (a) Store head address of the first node in the pointer p

   i.e. P = head;

   (b) Move head to the next node

   i.e. head = head → next

   (c) Free the memory allocated to the node to be deleted

   i.e. free(p);

   goto step 5

2. [otherwise]

   position a pointer q on $(LOC - 1)^{th}$ node.

   i.e. q = head;

   for (i = 1; i<LOC–1; i++)

   q = q →next;

3. if 'q' is 'NULL'

   then

   [report underflow and terminate the algorithm goto step5]

4. Delete the desired node

   i.e. p = q → next;

Techknowledge Publications

q → next = p → next;

free(p);

5. Stop.

'C' function to delete a node at location 'LOC' from linked list.

```
node *delete_LOC(node *head, int LOC)
{
    node *p, *q;
    int i;
    if(LOC == 1)
    {
        p = head;
        head = head->next;
        free(p);
        return(head);
    }
    q = head;
    for(i = 1; i<LOC–1; i++)
        q = q->next;
    if(q == NULL)
    {
        printf("\n Underflow");
        return(head);
    }
    p = q->next;
    q->next = p->next;
    free(p);
    return(head);
}
```

6.2.6(C) Delete a Linked List, Referenced by the Pointer Head

Algorithm

1. if head == NULL

   then

   [goto step 3]

2. [otherwise] Delete the first node.

   i.e. P = head;

   head = head → next;

   free(p);

   goto step 1.

3. Stop.

'C' function to delete a linked list, referenced by the pointer head.

```
node *delete_list(node *head)
{
    node *p;
    while(head! = NULL)
    {
        p = head;
        head = head->next;
        free(p);
    }
    return(head);
}
```

Ex. 6.2.3 : Write an algorithm for traversing a singly linked list that deletes all the nodes having negative key/value. **SPPU - Dec. 19, 6 Marks**

Soln. : Algorithm is being given in the form of C-function.

```
node*delete_negative (node*head)
{
    node *p, *q;
    while (head →data <0 && head ! = NULL)
    {
        p = head;
        head = head→next;
        free(p);
    }
    p = head→next;
    while(p→data < 0 && p ! = NULL)
    {
        q = p;
        p = p→next;
        free(q);
    }
    return(head) ;
}
```

Techknowledge Publication

6.2.7 Concatenation of Two Linked Lists

Algorithm for concatenation

Let us assume that the two linked lists are referenced by head1 and head2 respectively.

(1) If the first linked list is empty then return head2.

(2) If the second linked list is empty then return head1.

(3) Store the address of the starting node of the first linked list in a pointer variable, say P.

(4) Move the P to the last node of the linked list through simple linked list traversal technique.

(5) Store the address of the first node of the second linked list in the next field of the node pointed by P and return head1.

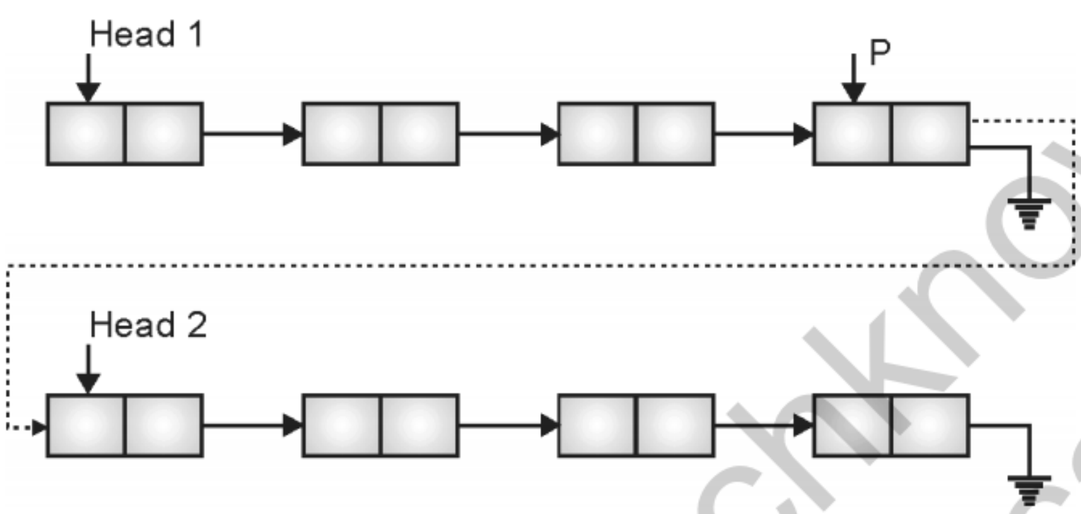


Fig. 6.2.9

'C' function to concatenate two linked lists.

```
node * concatenate(node *head1, node *head2)
{
    node *P;
    if(head1 == NULL)
    // If the first linked list is empty
        return(head2);
    if(head2 == NULL)
    // If the second linked list is empty
        return(head1);
        P = head1;
    //Place P on the first node of the first linked list
    while(P → next ! = NULL)
    // Move P to the last node
        P = P → next;
        P→next = head2;

    /* Address of the first node of the second linked list
    stored in the last node of the first linked list */
    return(head1);
}
```

6.2.8 Inversion of Linked List

Q. Write an algorithm to perform reverse operations on singly linked list: **SPPU - Dec. 16, 3 Marks**

Q. Write a pseudo C/C++ Code to reverse singly linked list. **SPPU - May 17, Dec. 19, 4 Marks**

A linked list can be reversed by changing the direction of the pointer, iteratively.

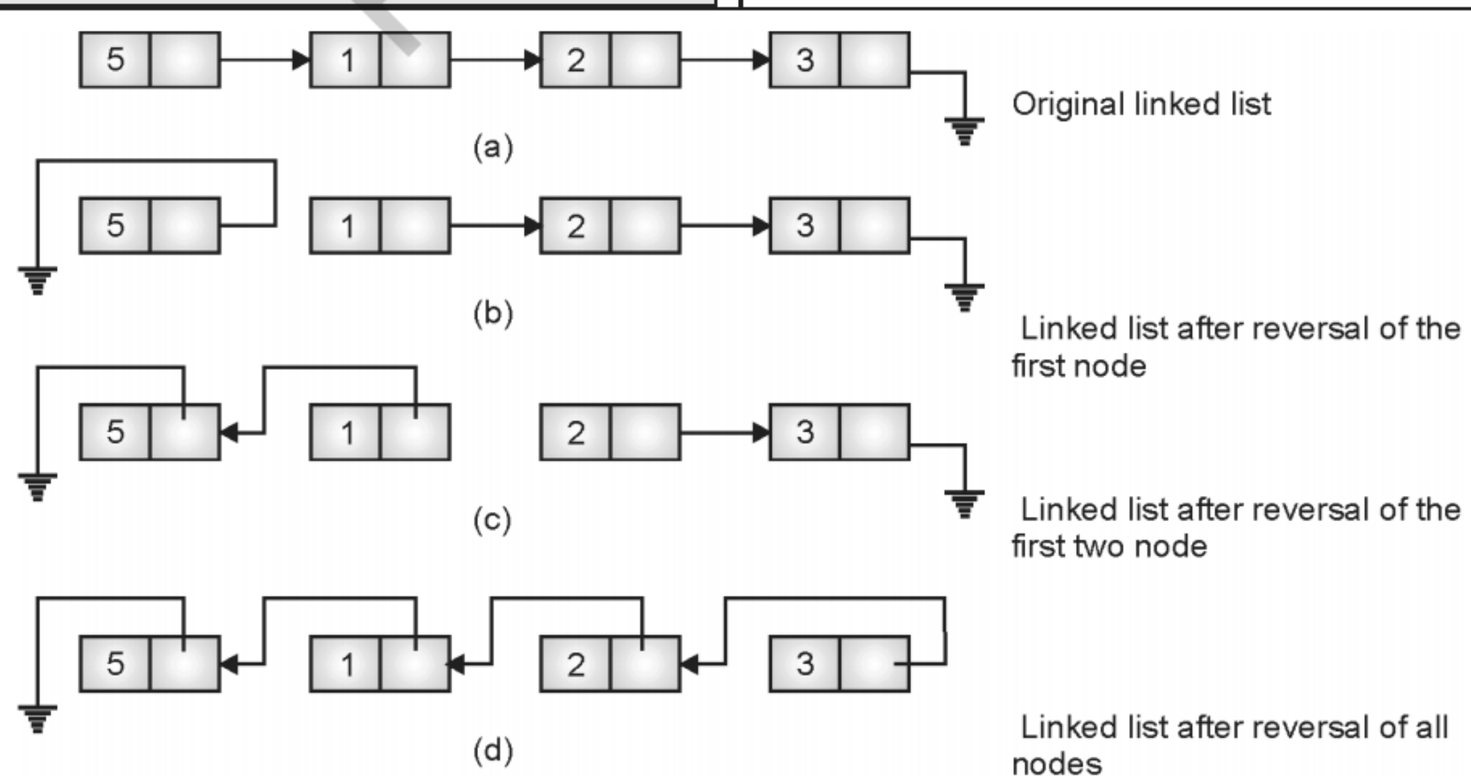


Fig. 6.2.10

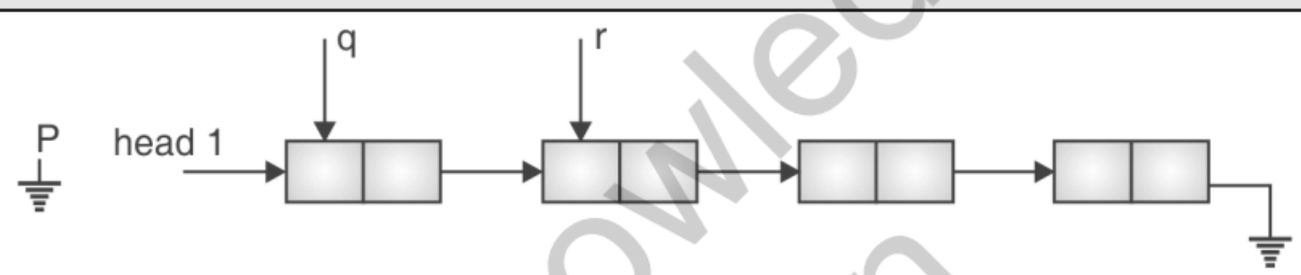
Tech Knowledge Publications

Algorithm for reversing the linked list

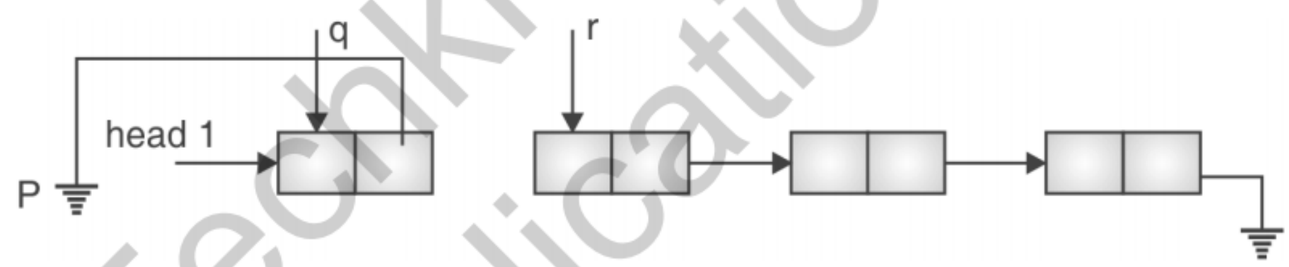
Let us take three pointer variables P, q and r.

P references the linked list reversed so far. q points to the linked list to be reversed. r points to the next node of q.

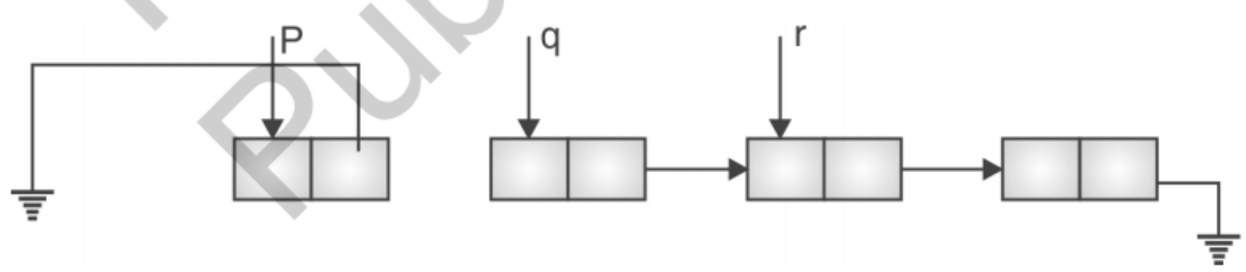
```
P = NULL; q = head;      r = q → next;
while(all nodes have not been reversed)
{
    reverse the node pointed by q.
    q → next = P;
    move P, q, r forward by a node.
    P = q;
    q = r
    r = r → next;
}
```



(a) Initial value of p, q, r



(b) After reversing the first node



(c) After moving p, q, r forward

Fig. 6.2.11

'C' function for inversion

```
node * invert(node *head)
/* invert a linked list pointed by head */
{
    node *P , *q , *r;
    P = NULL; q = head; r = q → next;
    /* initial values of P, q and r */
    while(q ! = NULL)
    {   /* until all nodes have been reversed */
        q → next = P;
        /* move P, q, r forward by a node */
        P = q;
        q = r;
        if(r ! = NULL)
            r = r → next;
    }
  return(P);
}
```

TechKnowledge Publications

6.2.9 Searching a Data 'x' in a Linked List, Referenced by the Pointer Head

Algorithm

In order to search an element in a linked list, we start traversing the linked list from the first node. Traversal ends with a success if the element found. If the element is not found then search ends in a failure.

1. P = head;
2. if the data stored in node pointed by P is x
   then
   [end the search with success.
   i.e. return (1)
   ]
3. Continue searching
   i.e. P = P → next.
6. if end of linked list
   then
   [end with failure
   i.e. return(0);
   ]
5. goto step 2
6. Stop.

Search functions

(I) 'C' function for searching an element x in a linked list referenced by the pointer head. Function returns 1 if search ends in a success, otherwise, the function returns 0.

```
int search1(node *head, int x)
{
    node *p;
    p = head;
    while(p! = NULL)
    {
        if(P->data == x)
            return(1);
        p = p->next;
    }
    return(0);
}
```

(II) 'C' function for searching an element x in a linked list referenced by the pointer head. Function returns address of the node if search ends in a success, otherwise, the function returns 'NULL'.

```
node *search2(node *head, int x)
{
    node * P;
    P = head;
    while(P ! = NULL)
    {
        if(P->data == x)
            return(P);
        P = P -> next;
    }
    return(NULL);
}
```

(III) 'C' function for searching an element x in a linked list referenced by the pointer head. Function returns location of the node if search ends in a success, otherwise, the function returns '–1'.

```
int search3(node *head, int x)
{
    int i = 1;
    node *p;
    p = head;
    while(p! = NULL)
    {
        if(p->data == x)
            return(i);
        i++;
        p = p->next;
    }
    return(–1);
}
```

Techknowledge Publications

6.2.10 Searching an Element x in a Sorted Linked List

Algorithm

[Linked list is referenced by the pointer head. Elements of the linked list are ordered in ascending order]

In order to search an element in a linked list of sorted elements, we start traversing the linked list from the first node, using a pointer P.

Search ends with success if the element is found in the node pointed by P.

Search ends in a failure under any of the given condition.

(a) P has become 'NULL'

(b) P → data is greater than x.

1. P = head
2. if P →data is equal to x

   then

   [report success and

   end the traversal

   ]
3. if P is Null or P → data > x

   then

   [report failure and

   end the traversal

   ]
4. Continue searching

   i.e. P = P → next;

   goto step 2.
5. Stop.

'C' function to search an element in sorted list.

```
int search_sorted(node *head, int x)
{
    node *p;
    p = head;
    while(p! = NULL)
    {
        if(p->data == x)
            return(1);
        if(p->data>x)
            return(0);
        p = p->next;
    }
    return(0);
}
```

6.2.11 New Linear Linked List by Selecting Alternate Element

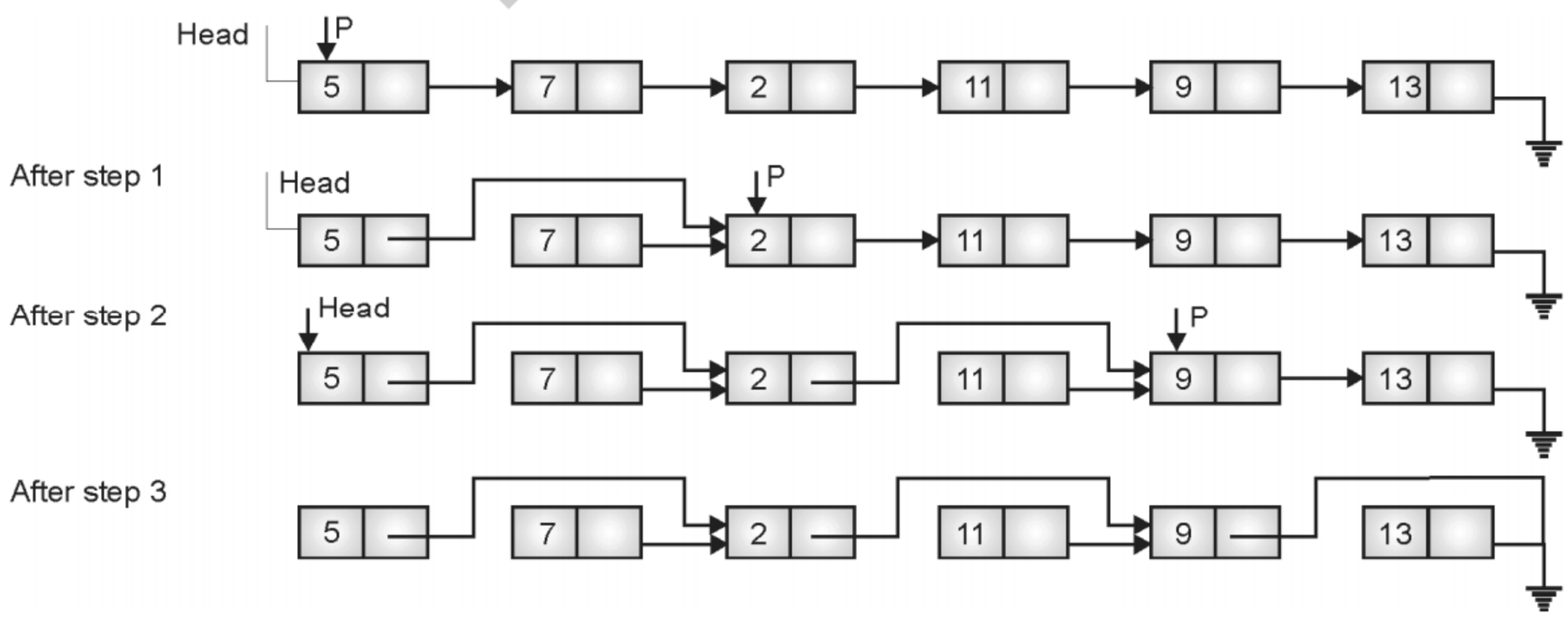


Fig. 6.2.12

Techknowledge Publication

Algorithm

[Linked list is referenced by the pointer head]

1. P = head
2. if (P → next is equal to NULL)

   then

   goto step 5
3. P → next = P → next → next

   P = P → next;
4. goto step 2
5. Stop.

```
void create_alternate(node *head)
{
    node *p;
    p = head;
    while(p! = NULL && p->next! = NULL)
    {
        p->next = p->next->next;
        p = p->next;
    }
}
```

6.2.12 Handling of Records through Linked List

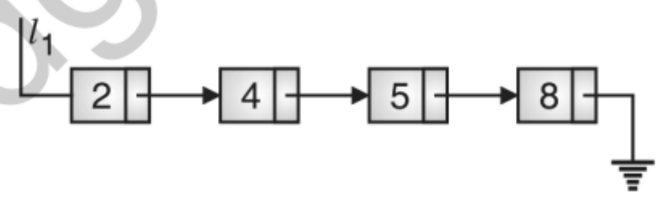
Assume a singly linked list where each node contains student details like name, rollno and percentage of marks. Following is a COUNT( ) function to traverse the linked list and count how many students have obtained more than 60% marks.

```
/* Structure used to represent a node */
typedef struct node
{
    char name [30];
    int rollno;
    float percent;
    struct node *next;
}node;
int COUNT(node *head)
{
    int n = 0;
    while(head! = NULL)
    {
        if(head -> percent > 60.00)
            n++;
        head = head -> next;
    }
    return(n);
}
```

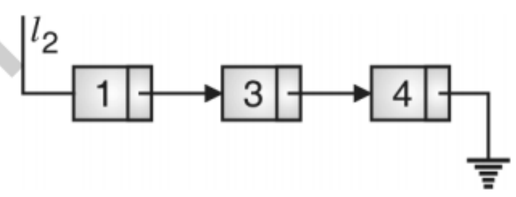
6.2.13 Merging of Sorted Linked Lists

Merging of two sorted linked l_1 and l_2 with resultant list as l is being explained with help of an example.

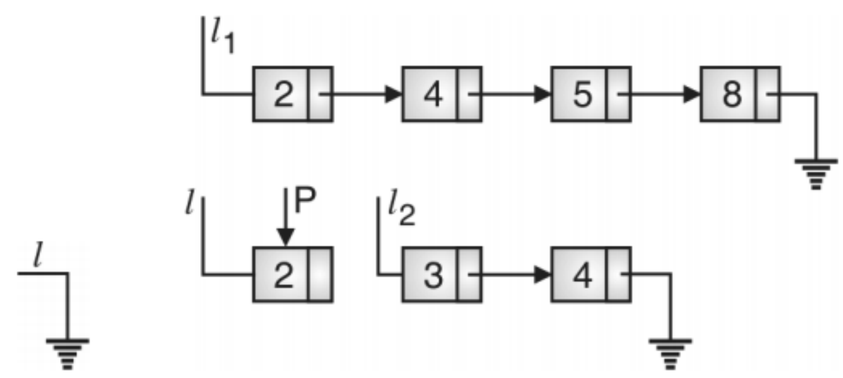
First list l_1 (given) :



Second list l_2 (given) :

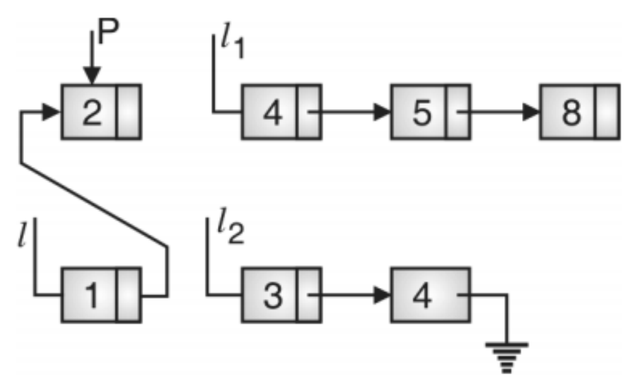


Final list l (initially NULL) :



Merging

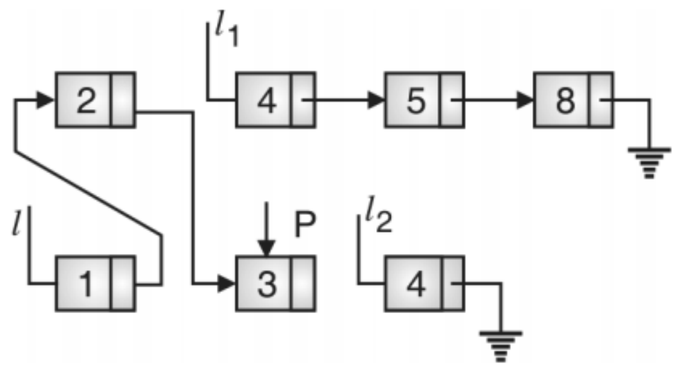
Step 1 : Smaller of the two elements pointed by l_1 and l_2 is copied to l. A pointer P is positioned on the last mode of l.



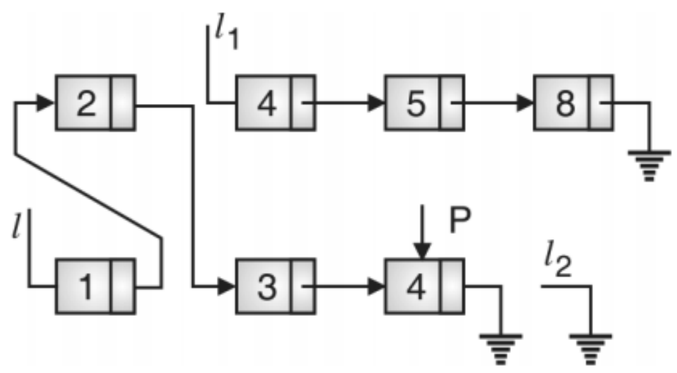
Step 2 : Smaller of the two elements pointed by l_1 and l_2 is appended to l. P is positioned on the last node of l.

Techknowledge Publication

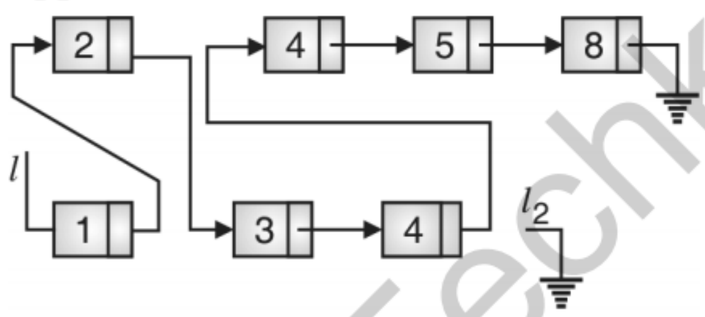
Step 3 : Smaller of the two elements pointed by l_1 and l_2 is appended to l. P is positioned on the last node of l.



Step 4 : Smaller of the two elements pointed by l_1 and l_2 is appended to l. P. is positioned on the last node.



Step 5 : Since, l_2 has become NULL, the list l_1 is appended at the end of l.



'C' function for merging of two sorted lists pointed by *l*1 and *l*2

```
node *  merge(node *  l1, node * l2)
{
    node *l = NULL, *P ;
    if(l1 == NULL)        //first list is empty
        return(l2);
    if(l2 == NULL)        //second list is empty
        return(l1) ;
    // copy the first element
    if(l1 → data < l2 → data)
    {
        l = P = l1 ;
        l1 = l1 → next ;
    }
    else
    {
        l = P = l2 ;
        l2 = l2 → next ;
    }
    // Append remaining data to produce l.
    while(l1 ! = NULL && l2 ! = NULL)
    {
    if(l1 → data < l2 → data)
    {
        P → next = l1 ;
        l1 = l1 → next;
        P = P → next ;
    }
    else
    {
        P → next = l2 :
        l2 = l2 → next ;
        P = P → next ;
    }
    }
    /* One of the lists l1 or l2 will have some elements
       */
    if(l1 ! = NULL)
        P → next = l1 ;
    else
        P → next = l2 ;
    return(l) ;
}
```

6.2.14 Splitting a Linked List at the Middle and Merge with Second Half as First Half

'C' program to create a single linked list and split it at the middle and make the second half as the first and vice versa. Display the final list.

```
#include<stdio.h>
#include<conio.h>
typedef struct node
{
    int data;
    struct node *next;
```

TechKnowledge Publications

```
}node;
void main()
{
    node *head = NULL, *p, *q;
    int n, i, x;
    printf("\n Enter no. of nodes :");
    scanf("%d", &n);
    /*  create a linked list of n nodes */
    for(i = 0; i<n; i++)
    {
        printf("\n Enter the next element :");
        scanf("%d", &x);
        p = (node*)malloc(sizeof(node));
        p->data = x;
        p->next = NULL;
        if(head == NULL)
            head = q = p;
        else
        {
            q->next = p;
            q = p;
        }
    }
    /* locate the centre of linked list */
    p = q = head;
    while(q->next! = NULL)
    {
        p = p->next;
        q = q->next;
        if(q->next! = NULL)
            q = q->next;
    }
    q->next = head;
    head = p->next;
    p->next = NULL;
    /* Display the final list */
    for(p = head; p! = NULL; p->next)
    {
        printf("\n %d", p->data);
        p = p->next;
    }
}
```

Output

```
Enter no. of nodes :3
Enter the next element :12
Enter the next element :23
Enter the next element :34
34
12
23
```

6.2.15 Removing Duplicate Elements from a Linked List

```
void delete_duplicates(node *head)
{
    int x;
    node *p;
    while(head! = NULL)
    {
        x = head → data;
        /* Delete all subsequent nodes with x in its data
        field */
        p = head;
        while(p → next ! = NULL)
        {
            if(p → next → data == x) //delete
                p →next = p → next→next;
            else
                p = p→next;
        }
        head = head →next;
        // find the duplicates of the next element
}
```

Ex. 6.2.4 : Write recursive functions for :

(i) Display SLL forward

(ii) Display SLL reverse

Soln. :

(i) Display SLL forward

```
void display_forward(node*h)
{
    if(h! = NULL)
    {
```

Techknowledge Publications

```
        printf("%d", h → data);
        display_forward(h →next);
    }
}
```

(ii) Display SLL reverse

```
void display_reverse(node*h)
{
    if(h! = NULL)
    {
        display_reverse(h→next);
        printf("%d", h →data);
    }
}
```

Ex. 6.2.5 : Suppose a linked list consists of numerical values. Write a function for finding the maximum element of the list and the product of all the numbers in the List.

Soln. :

```
void example(node *h)
{
    int product = 1, max;
    max = h → data;
    while(h ! = NULL)
    {
        product = product *h–>data;
        if(h –> data > max)
            max = h –> data;
        h = h –> next;
    }
        printf("\n product = %d", product);
        printf("\n largest number = %d", max);
}
```

6.3 Circular Linked List

Q. Write pseudo C/C++ code to represent circular linked list as an ADT. **SPPU - May 19, 6 Marks**

In a circular linked list, last node is connected back to the first node. In some applications, it is convenient to use circular linked list.

A queue data structure can be implemented using a circular linked list, with a single pointer "rear" as the front node can be accessed through the rear node.

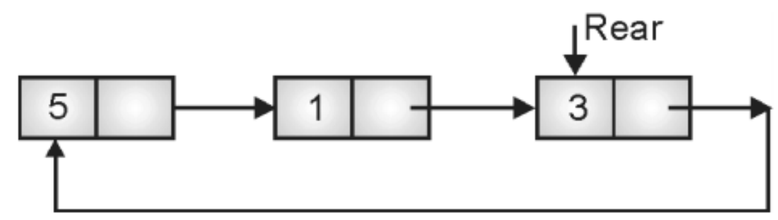


Fig. 6.3.1

Insertion of a node at the start or end of a circular linked list identified by a pointer to the last node of the linked list takes a constant amount of time. It is irrespective of the length of the linked list.

Algorithm for traversing a circular list is slightly different than the same algorithm for a singly connected linked list because "NULL" is not encountered.

'C' function for inserting a number at the rear of a circular linked list.

```
node * insert_rear(node * rear, int x)
{
    node *P;
    P = (node*) malloc(sizeof(node));
        /* acquire memory for the current data */
    P → data = x;
    if(rear == NULL)
    {
        /* inserting in an empty linked list */
        rear = P;
        /* node is connected back to the same node */
        P → next = P;
        return(rear);
    }
    else
    {
        P → next = rear → next;
        rear → next = P;
        /* node P is made a part of the circle */
        rear = P
        return(rear);
    }
}
```

Techknowledge Publication

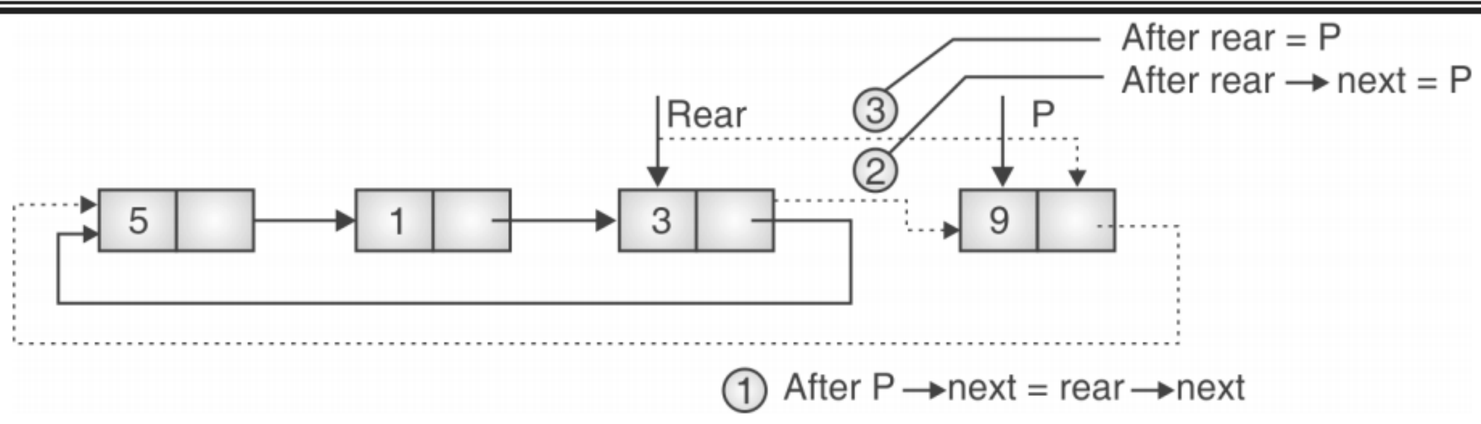


Fig. 6.3.2 : Insertion of a node at rear in circular linked list

'C' function for inserting a number at the front of the circular linked list.

```
node * insert_front(node * rear, int x)
{
    node *P;
    P = (node *) malloc(sizeof(node));        /* acquire memory */
    P → data = x;
    if(rear == NULL)
    {   rear = P;
        P → next = P;
        return(rear);
    }
    else
    {
        P → next = rear → next;
        rear → next = P;
        return(rear);                         /* rear is not moved after insertion */
    }
}
```

'C' function for traversing a circular linked list.

```
void print(node * rear)
{
    node *P;
    if(rear ! = NULL)
    {
        P = rear → next;
        /* start traversing from the front */
        do
        {
            printf("\n %d", P → data);
            P = P → next;
        } while(P ! = rear → next);
    }
}
```

In a circular linked list, starting and the terminating case for traversal are same. In such a case, do-while is the most suitable construct for traversing.

Ex. 6.3.1 : Write a 'C' function to insert and delete a Nth element in circular linked list ? Give the pictorial representation of the same.

Soln. :

(i) Insertion

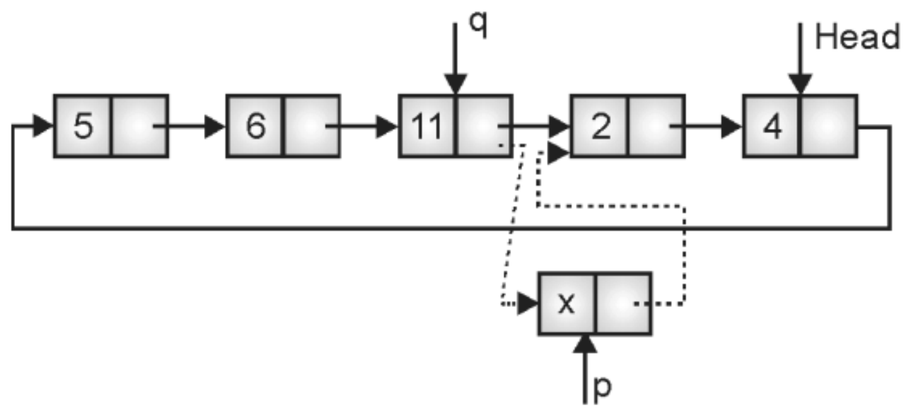


Fig. Ex. 6.3.1(a) : Insertion

q points to $(N-1)^{th}$ element. Element to be inserted is pointed by p.

Techknowledge Publications

'C' function for insertion.

```
node * insert_cir_loc(node *head, int x, int loc)
{
    node *p, *q;
    int i;
    p = (node *)malloc(sizeof(node));
    p->data = x;
    p->next = NULL;
    if(loc == 1)
        if(head == NULL)
        {
            p->next = p;
            return(p);
        }
        else
        {
            p->next = head->next;
            head->next = p;
            return(head);
        }
        q = head->next;
        for(i = 1; i<loc–1; i++)
        if(q! = head)
            q = q->next;
        else
        {
            printf("\n Overflow");
            return(head);
        }
        p->next = q->next;
        q->next = p;
    return(head);
}
```

(ii) Deletion

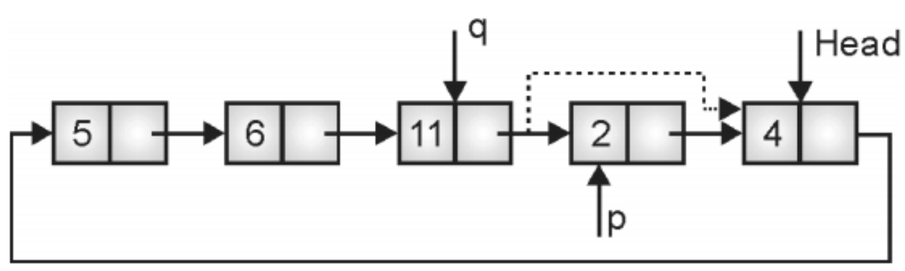


Fig. Ex. 6.3.1(b) : Deletion

q points to $(N–1)^{th}$ element. Element to be deleted is pointed by p

'C' function for deletion.

```
node *delete_cir_loc(node *head, int loc)
{
    node *p, *q;
    int i;
    if(loc == 1)
    {
        if(head->next == head)
        {
            free(p);
            return(NULL);
        }
        else
        {
            p = head->next;
            head->next = p->next;
            free(p);
            return(head);
        }
    }
    q = head->next;
    for(i = 1; i<loc–1; i++)
        if(q! = head)
            q = q->next;
        else
        {
            printf("\n Underflow");
            return(head);
        }
        p = q->next;
        q->next = p->next;
        free(p);
    return(head);
}
```

Ex. 6.3.2 : Write a program for Circular SLL of integers with the following options :

(i) Create list

(ii) Display list

(iii) Delete a specified value

(iv) Insert a value after a given item.

Techknowledge Publications

Soln. :

```
# include <stdio.h>
# include <conio.h>
    typedef struct node
    {
        int data;
        struct node * next;
    }node;
    node * create()
    {
        node * head, * p;
        int i, n;
        printf("\n Enter no.  of data:");
        scanf("%d", &n);
        printf(\n Enter data:");
        head = (node*) malloc(sizeof(node));
        head→next = NULL;
        scanf("%d", & head→data);
        p = head;
        for(i = 1; i < n; i++)
        {
            p→next = (node*) malloc(sizeof(node));
            p = p→next;
            p→next = NULL;
            scanf("%d", &p→data);
        }
            return(head);
    }
    void display(node * h)
    {
        for( ; h! = NULL; h = h→next)
            printf("%d\t", h→data);
    }
    node * delete_specified(node *h, int x)
    {
        node *p, *q;
        if(x == h→data)
        {
            p = head;
            h = h→next;
            free(p);
        }
        else
        {
        while(x!=p→next→data&&p→next!=NULL)
        p = p→next;
        if(p→next! = NULL)
        {
            q = p→next;
            p→next = p→next→next;

            free(q);
            }
        }
        return(h);
    }

    void insert_after(node *h, int x, int y)
    {   // insert x after y
        node *p, *q;
        q = (node*)malloc(sizeof(node));
        q→data = x;
        q→next = NULL;
        p = h;
        while(p→data ! = x && p! = NULL)
        p = p→next;
        if(p! = NULL)
        {
            q→next = p→next;
            p→next = q;
        }
    }
    void main()
    {
        node* head = NULL;
        int op, x, y;
        do
        {
            printf("\n 1) Create\n 2) Display\n
            3) Delete\n 4) Insert\n 5) Quit");
            printf("\n Enter your choice :");
```

Techknowledge Publication

```
        scanf("%d", &op);
    switch(op)
    {
    case 1 : head = create(); break;
    case 2 : display(head); break;
    case 3 : printf("\n Enter the data:");
             head = delete_specified(head, x);
             break;
    case 4 : printf("\n Enter specified data and
             data to be inserted:");
             scanf("%d%d", &x, &y);
             insert_after(y, x); break;
    }
    } while(op! = 5);
}
```

Ex. 6.3.3 : Write pseudo C function to display the singly circular link list storing character data in the reverse manner.

Soln. :

```
void display_reverse(node *h)
{
    node *p, *q;
    p = h;
  /*pointer p moves in reverse with the help of q */
    do
    {
        printf("%d\t", p→data);
        q = h;
        while(q→next! = p)
            q = q→next;
        p = q;
    } while(p! = h);
}
```

6.3.1 Applications of Circular Linked List

Circular linked list can be used for storing a list of elements. It allows insertion in constant time at both ends of the list. In general, it can be used for following applications :

1. Stack
2. Queue
3. Dequeue
4. Priority queue
5. Polynomial
6. List
7. Sparse matrix.

6.4 Doubly Linked List

In a singly linked list, we can easily move in the direction of the link. Finding a node, preceding any node is a time consuming process. The only way to find the node which precedes a node is to start back at the beginning of the list. If we have a problem where moving in either direction is often necessary, then it is useful to have doubly linked lists. Each node has two link fields, one linking in the forward direction and one in the backward direction.

A node in a doubly linked list has at least 3 fields, say "data", "next" and "previous". A doubly linked list is shown in Fig. 6.4.1.

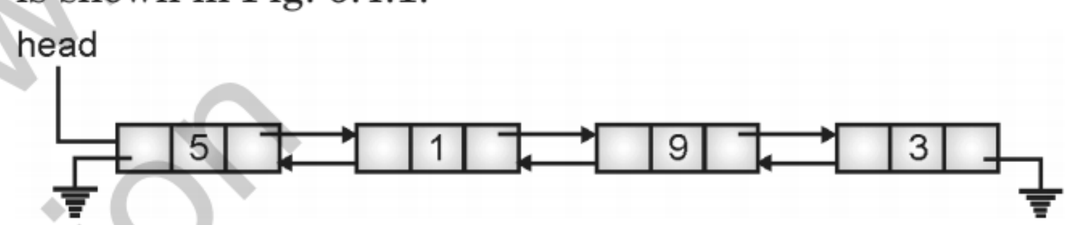


Fig. 6.4.1 : A doubly linked list

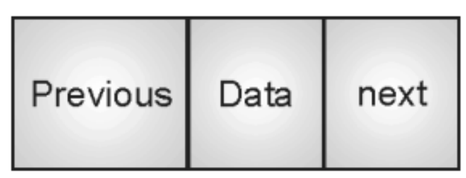


Fig. 6.4.2 : Structure of the node

A node of a doubly linked list can be defined using the following structure.

```
typedef struct dnode
{
    int data;
    struct dnode *next, *prev;
} dnode;
```

6.4.1 Creation of a Doubly Linked List

Fig. 6.4.3 shows stepwise insertion of 3 elements, namely 5, 2 and 6 Pointer P is used to help insertion at the rear of the linked list.

Techknowledge Publication

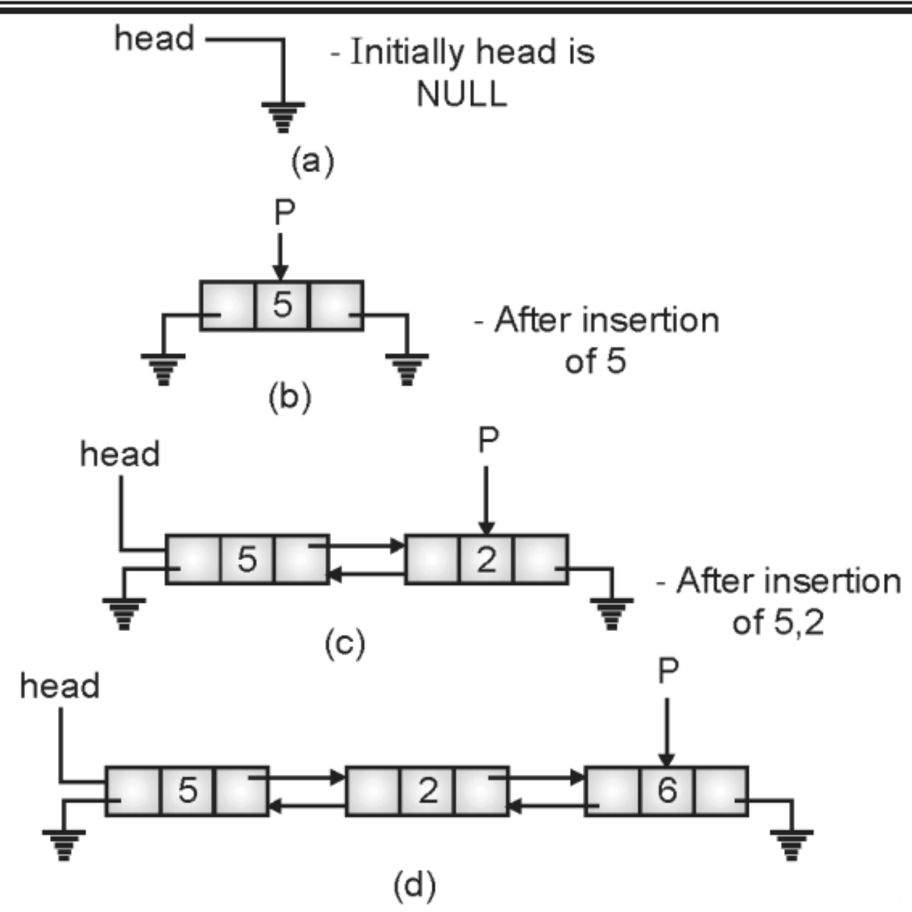


Fig. 6.4.3

Algorithm for insertion of x at the rear

dnode *P, *q;

Step 1 : Acquire memory for the new data

q = (dnode*) malloc(sizeof(dnode));

Step 2 : Store x in the newly acquired node

q → data = x;

Step 3 : Make previous and next field as NULL;

q → prev = q → next = NULL;

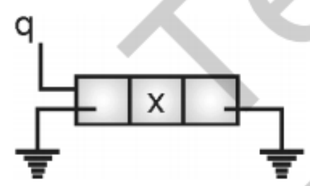


Fig. 6.4.4 : Status of node after step1 to step3

Case I : Inserting in an empty list (head = NULL)

```
if(head == NULL)
{
    head = P = q;
}
```

Case II : Inserting in an non-empty list

```
else
{
    P → next = q;
    q → prev = P;
    P = q;
}
```

Note : A linked list can be created through a repeated application of (1) reading a new value of n and then inserting at rear.

Program 6.4.1 : A sample program for creation of a doubly linked list and printing its elements in forward and reverse direction.

OR Write pseudo c/c++ code to represent doubly linked list as an ADT. **SPPU - Dec. 16, 6 Marks**

```
#include<conio.h>
#include<stdlib.h>
#include<stdio.h>
typedef struct dnode
{
    int data;
    struct dnode *next, *prev;
}dnode;
dnode * create();
void print_forward(dnode *);
void print_reverse(dnode *);

void main()
{
    dnode *head;
    head = NULL;   // initially the list is empty
    head = create();
    printf("\n Elements in forward direction :");
    print_forward(head);
    printf("\n Elements in reverse direction :");
    print_reverse(head);
}

dnode *create()
{
    dnode *h, *P, *q;
    int i, n, x;
    h = NULL;
    printf("\n Enter no. of elements :");
    scanf("%d", &n);
    for(i = 0; i<n; i++)
    {
```

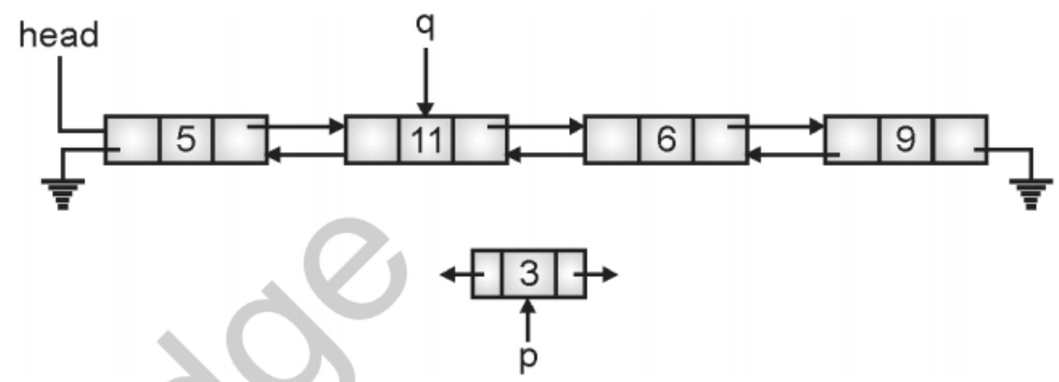
Tech Knowledge Publication

```
        printf("\n Enter next data: ");
        scanf("%d", &x);
        q = (dnode*)malloc(sizeof(dnode));
        q->data = x;
        q->prev = q->next = NULL;
        if(h == NULL)
            P = h = q;
        else
            P->next = q;
            q->prev = P;
            P = q;
    }
    return(h);
}

void print_forward(dnode *h)
{
    while(h! = NULL)
    {
        printf("<- %d ->", h->data);
        h = h->next;
    }
}
void print_reverse(dnode *h)
{
    while(h->next! = NULL)
        h = h->next;
    while(h! = NULL)
    {
        printf("<- %d ->", h->data);
        h = h->prev;
    }
}
```

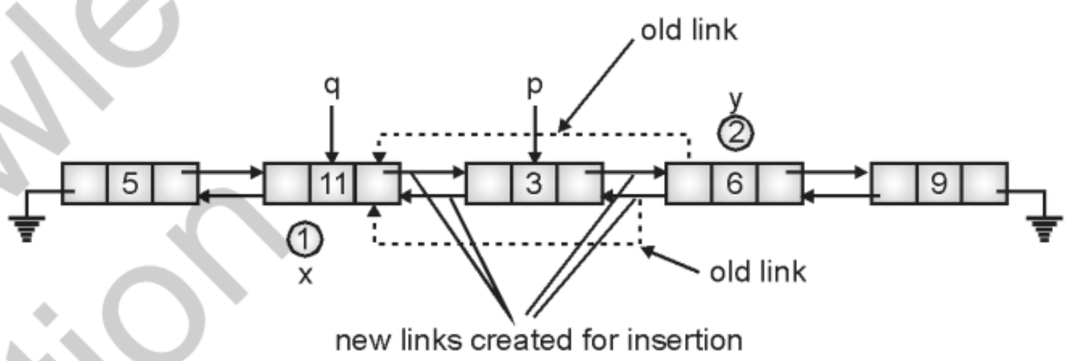
Output

```
Enter no. of elements :4
Enter next data:1
Enter next data:2
Enter next data:3
Enter next data:4
Elements in forward direction :
<-1-> <-2-><-3-><-4->
Elements in reverse direction :
<-4-> <-3-><-2-><-1->
```

Insertion and deletion are two basic operations on such lists. Consider that a new node pointed to by "P" is to be **inserted** after the node pointed to by q in a doubly linked list as shown in the Fig. 6.4.5.



(a) Insertion of a node in a doubly linked list



(b) A doubly linked list after the insertion of the node
Fig. 6.4.5

Links of two nodes are altered during insertion. In the Fig. 6.4.5(b), a new node pointed to by P is inserted between the two nodes x and y. Following links must be changed for proper insertion of node pointed by P between x and y.

(1) Right link of x

(2) Left link of y

(3) Left and right links of node pointed to by P

Instructions for making above changes

```
P → next = q → next;     //right link of P set to y
P → prev = q;            // left link of P set to x
q → next = P;            // right link of x set to P
(P → next) → prev = P;   // left link of y set to P
```

Techknowledge Publications

'C' function for inserting a node pointed to by P after a node pointed to by q.

```
void insert(dnode *P, dnode *q)
{
    P → next = q → next;
    P  → prev = q;
    q → next = P;
    if(P → next ! = NULL)
    /* insertion at the end */
    P → next → prev = P;
}
```

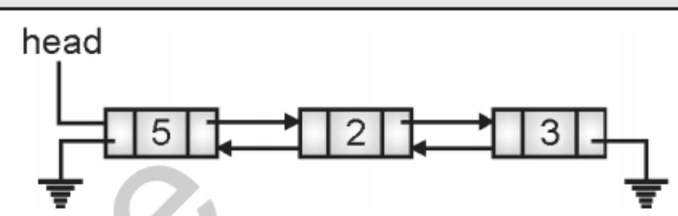
If the right pointer of x is NULL then the insertion is being performed at the end. Hence, there is no question of modifying the left pointer of y.

'C' function for inserting a node pointed to by p before a node pointed to by q.

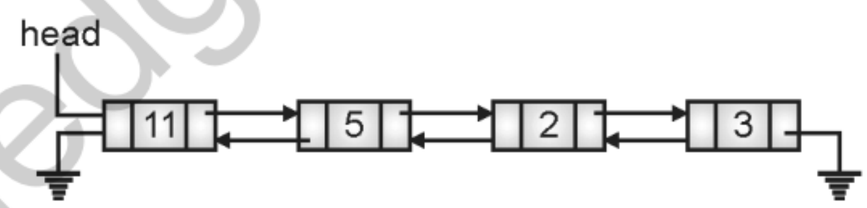
```
dnode * insert(dnode * head, dnode *P, dnode * q)
{
    if(q → prev == NULL)
    { // insert at the beginning
        P→ next = q;
        P→ prev = NULL;
        q→ prev = p;
        return(p);
    }
    else
    {   P → prev = q → prev;
        P → next = q;
        P → prev → next = p;
        q → prev = p;
        return(head);
    }
}
```

'C' function for inserting a value x, at the beginning of doubly linked list.

```
dnode * insert2(dnode * head, int x)
{
    dnode *P;
    P = (dnode *) malloc(sizeof(dnode));
    /* get memory for the new node */
    P → data = x;
    P → prev = NULL;
    P → next = head;
    if(head ! = NULL)
    /* not an empty list */
    head → prev = P;
    return(P);
}
```



(a) Original linked list



(b) Linked list after insertions of 11 at the front

Fig. 6.4.6

'C' function for inserting a value x, at the end of a doubly linked list.

```
donde * insert3(dnode *head, int x)
{
    dnode *P, *q;
    P = (dnode *) malloc(sizeof(dnode));
    P → data = x;
    P → prev = P → next = NULL;
    if(head == NULL)            /* empty list */
        return(P);
                                /* goto the last node */
    q = head;
    while(q → next ! = NULL)
        q = q → next;
    P → prev = q
    q → next = P;
    return(head);
}
```

TechKnowledge Publications

6.4.2 Deletion of a Node

Q. Write an algorithm to delete intermediate node from doubly linked list. **SPPU - May 17, Dec. 19, 3 Marks**

Q. Write pseudo C/C++ code to delete a node from a doubly linked list. **SPPU - Dec. 18, 6 Marks**

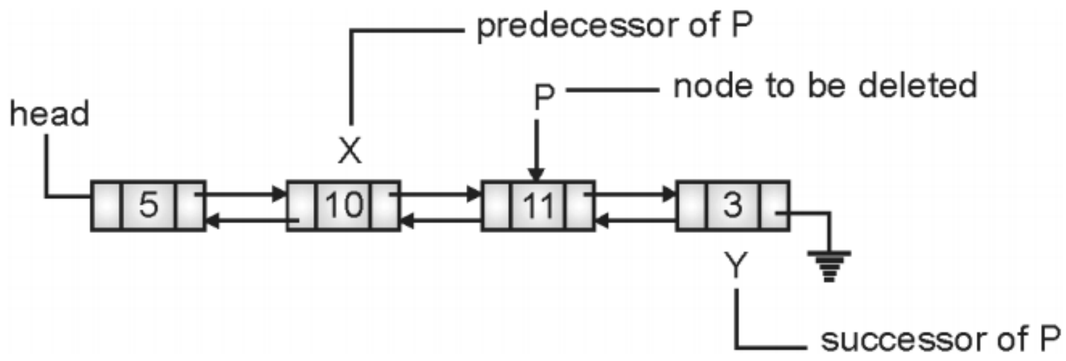


Fig. 6.4.7(a) : Node pointed to by P is to be deleted

When a node, pointed to by P is to be deleted then its predecessor node x and it's successors node y (as shown in Fig. 6.4.7(a)) will be affected.

Right link at x should be set to y.
Left link at y should be set to x.

Release the memory allocated to the node pointed to by P.

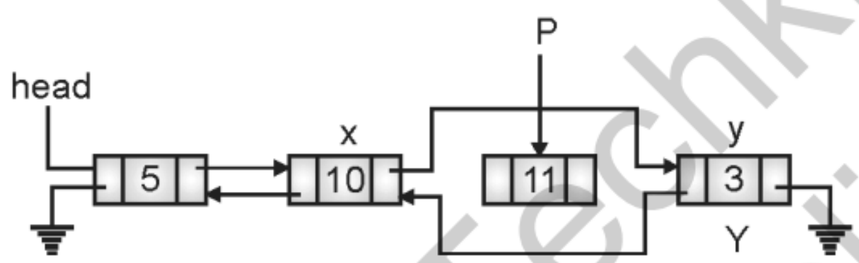


Fig. 6.4.7(b) : Links to be modified as shown above

C-instructions for deletions

```
P → prev → next = P → next;  // right at x set to y
P → next → prev = P → prev;  //left link at y set to x
free(P);
```

'C' function for deletion of a node pointed by p in a doubly linked list.

```
dnode * delete_double(dnode *head, dnode *P)
{
    if(P == head)    /* deleting the head node */
    {
        P → next → prev = NULL;
        head = P → next;
        free(P);
        return(head);
    }
    P → prev → next = P → next;
    if(P → next ! = NULL)      /* not the last node */
        P  → next  → prev = P  → prev;
        free(P);
    return(head);
}
```

- Special care should be taken to delete the first or the last node.
- If the node to be deleted is the first node then head should be advanced to the next node.
- If the node to be deleted is the last node then there is no node to its right.

Program 6.4.2 : Give node structure to represent a list names using DLL and write C functions for the following

(i) Display list forward

(ii) Display list reverse

(iii) Display names starting with letter S or s.

```
//Node structure
typedef struct node
{
        char data[30];
        struct node *next, *prev;
} node;
void display_forward(node *p)
{
    while(p! = NULL)
     {
        printf("\n %s", p→data);
        p = p→next;
     }
}
void display_reverse(node *p)
{
    if(p! = NULL)
    {
        display_reverse(p→next);
        printf("\n %s", p→data);
    }
}
```

TechKnowledge Publications

```
void display start_s(node *p)
{
    while(p! = NULL)
    {
        if(p→data[0] == 'S' ¦¦ p→data[0] = 's')
            printf("\n %s", p→data);
        p = p→next;
    }
}
```

Ex. 6.4.1 : Compare singly linked list and doubly linked list.

Soln. : Comparison of SLL and DLL

Sr. No.	Parameters	SLL	DLL
1.	Memory requirement	SLL takes less memory	Take more memory as there are two times, for would are backward.
2.	Speed of insertion and deletion.	Takes less time.	Takes more time as backward links are required to modified.
3.	Scanning	Only forward	Both forward and backward.
4.	Double ended ensure	Difficult to implement double ended queue.	Easy to implement double ended queue.

6.5 Doubly Linked Circular List

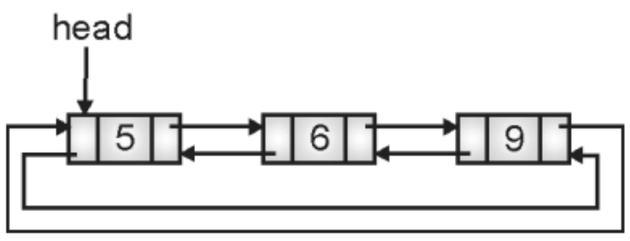


Fig. 6.5.1 : A doubly linked circular list

- In a doubly linked circular list, the previous pointer of the first node points to the last node.
- Next pointer of the last node points to the first node.
- In a doubly linked circular list, insertion at the front as well as at the rear can be done in constant time order (O(1)).
- If the list is not circular then insertion at the front can be done in constant time order (O(1)) but the insertion at the rear requires traversing of entire linked list to locate the last node. Time complexity of insertion at the end is of the order of (O(n)).

Fig. 6.5.2 showing stepwise insertion of 3 elements namely 5, 2 and 6 in a doubly linked circular list.

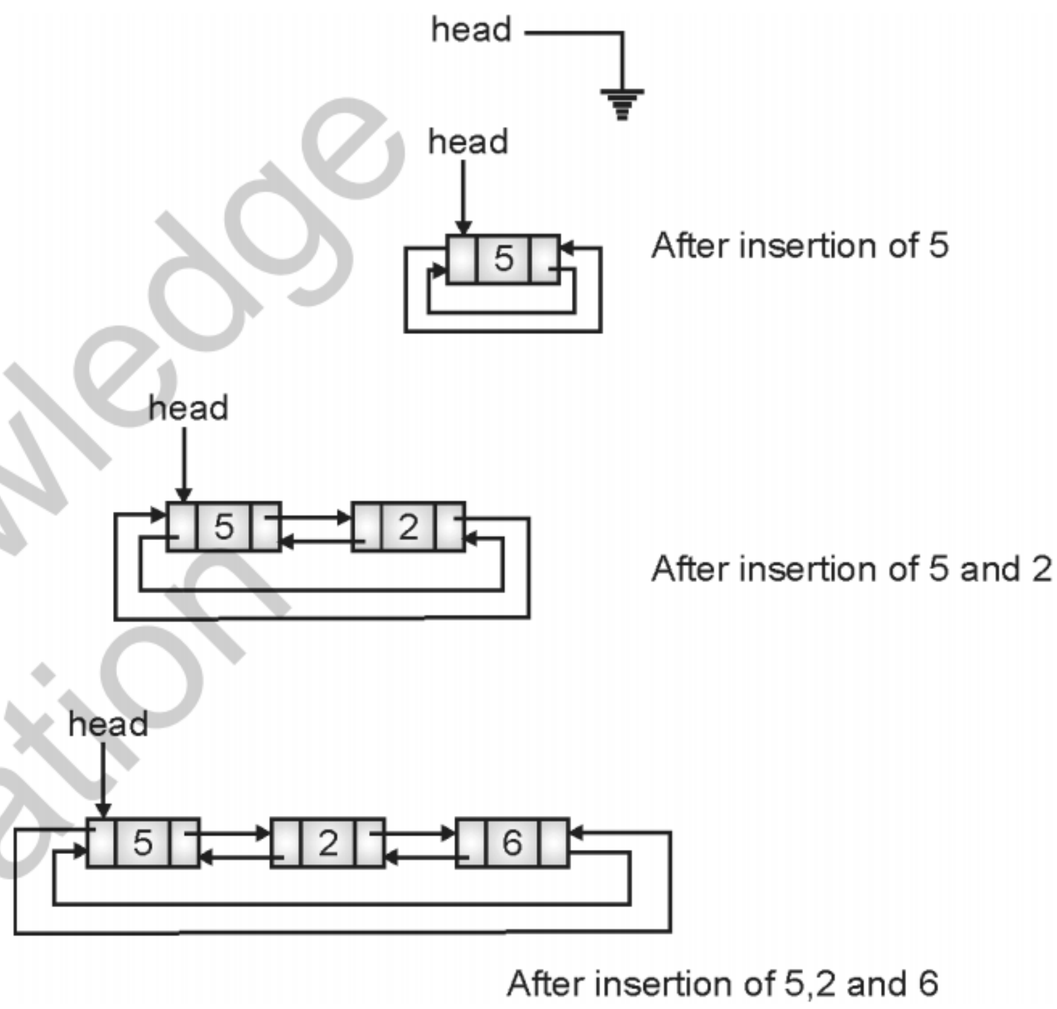


Fig. 6.5.2

Program 6.5.1 : A sample program for insertion of 5 elements in a doubly circular linked list and subsequently printing of elements stored in the list.

```
#include<conio.h>
#include<stdlib.h>
#include<stdio.h>
typedef struct dnode
{
    int data;
    struct dnode *next, *prev;
}dnode;
dnode * insert(dnode *head, int x);
void print(dnode *);
void main()
{
```

Tech Knowledge Publications

```
    dnode *head;
    int i, x;
    head = NULL;    // initially the list is empty
    printf("\n Enter 5 elements :");
    for(i = 1; i< = 5; i++)
    {
        scanf("%d", &x);
        head = insert(head, x);
    }
    print(head);
    getch();
}
dnode *insert(dnode *head, int x)
{
    dnode *P;
    P = (dnode*)malloc(sizeof(dnode));
    P->data = x;
    P->prev = P->next = NULL;
    if(head == NULL)
    {   //inserting in an empty link list
        P->prev = P->next = P;
        return(P);
    }
    P->prev = head->prev;
    P->next = head;
    head->prev->next = P;
    head->prev = P;
    return(head);
}
void print(dnode *head)
{
    dnode *P;
    P = head;
    do
    {
        printf("<- %d ->", P->data);
        P = P->next;
    }while(P! = head);
}
```

Output

```
Enter 5 elements :  5     4    3    2    1
<- 5 -><- 4 -><- 3 -><- 2 -><- 1 ->
```

Ex. 6.5.1 : Explain importance of header node in a linked list.

Soln. : Header node is an additional node, added at the beginning of a linked list. It does not contain any valid data. A list (5, 3, 2, 9) is shown below using a linked list with header node.

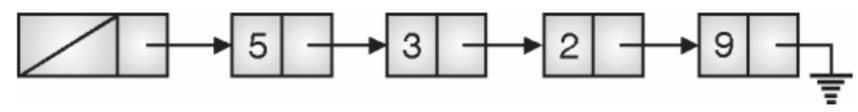


Fig. Ex. 6.5.1

Advantages

(1) The address of the first node does not change and hence the address of the header node can always be passed by value.

(2) In all cases, the insertion at the beginning requires the same steps. We do not have to consider whether the linked list is empty or not.

(3) In all cases, the deletion of the first data requires the same steps. We do not have to consider whether the list will become empty or not after deletion of the first data.

Disadvantage

(1) Additional memory is needed for the header node.

Ex. 6.5.2 : Write C code to (i) delete a node pointed by pointer *temp* in a circular DLL. (ii) insert a new node pointed by pointer *newp* after a node pointed by pointer *temp* in circular DLL.

Soln. :

(i)

temp → prev → next = temp → next;

temp → next → prev = temp → prev;

free (temp);

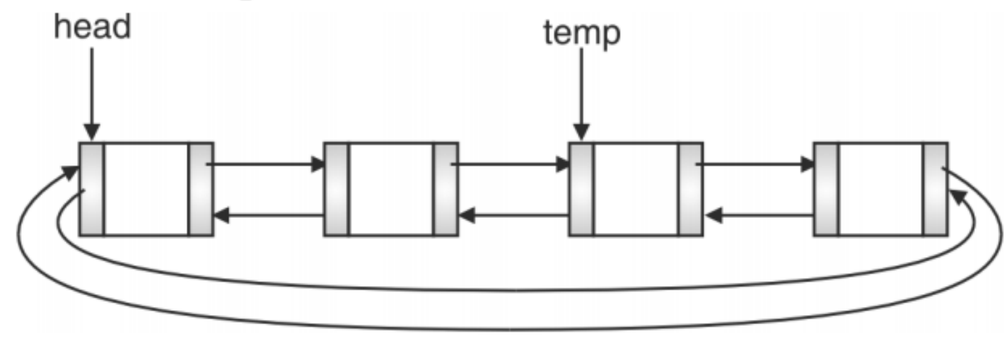


Fig. Ex. 6.5.2

(ii)

newp → next = temp → next;

newp → prev = temp;

temp → next = newp;

newp → next → prev = newp;

Techknowledge Publication

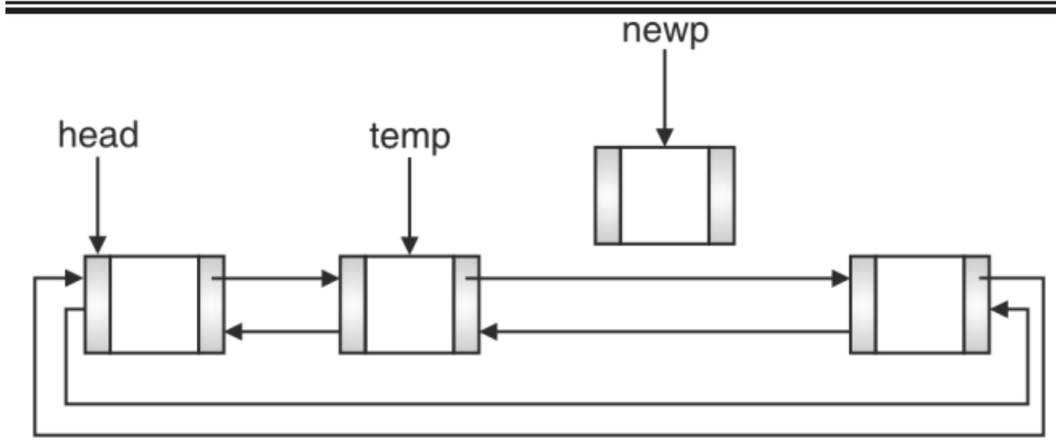


Fig. Ex. 6.5.2(a)

6.6 Applications of Linked Lists

6.6.1 Polynomials as Linked Lists

Q. Explain polynomial representation using linked list with an example. **SPPU - May 19, 3 Marks**

Representation of polynomial using linked list

A polynomial P(x) of degree n is defined by the following expression.

$$P(x) = a_0 + a_1x^1 + a_2x^2 + \ldots\ldots\ldots\ldots + a_nx^n$$

Where a_n is any real number and n is an integer.

A polynomial can be thought of as the ordered list of non-zero terms. Each term is a 2-tuple containing power and coefficient.

Example of polynomial as linked list

Thus, the polynomial $5 + 6x^2 - 9x^4$ is represented as ((0,5), (2,6), (4, – 9)). Tuple (0, 5) stands for $5x^0$; the second tuple (2, 6) indicates $6x^2$ and the last tuple denotes $-9x^4$. The linked representation of the said polynomial is shown below.

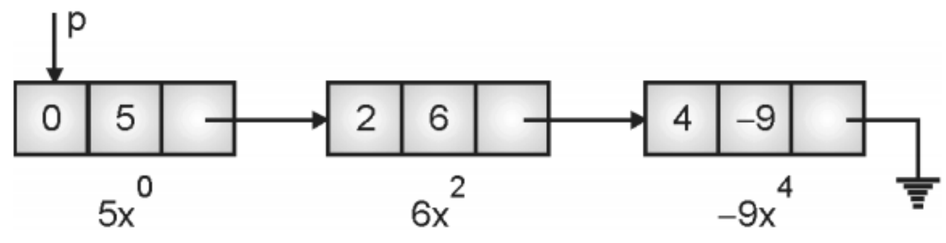


Fig. 6.6.1

Each term of the polynomial can be defined by the following structure.

```
typedef struct pnode
{
    float coeff;
    int pow;
    struct pnode * next;
} pnode;
```

We can treat a polynomial an abstract data type and perform following basic operations

(1) Creating a polynomial

(2) Printing a polynomial

(3) Addition of two polynomials

(4) Multiplication of two polynomials

(5) Evaluation of a polynomial.

Program 6.6.1 : A program to create and print a polynomial.

```
#include<conio.h>
#include<stdio.h>
#include<stdlib.h>
typedef struct pnode
{
    int pow;
    float coeff;
    struct pnode *next;
}pnode;
pnode create(int);    //create polynomials of n terms
void print(pnode *);
/* print the polynomial referenced by address of first
node*/
void main()
{
    pnode *HEAD;
    int n;
    printf("\n Enter no. of terms :");
    scanf("%d", &n);
    HEAD = create(n);
    print(HEAD);
}
```

Tech Knowledge Publication

'C' function to create a polynomial.

Creation of a linked list representing a polynomial is similar to that of a normal linked list. Each node of the linked list contains two data field, namely power and coefficient. Since, polynomial is an ordered list, data for each term must be entered in ascending order on power.

```
pnode *create(int n)
{
    pnode *head, *p;
    int i;
    // create the first node
    head = (pnode *)malloc(sizeof(pnode));
    //acquire memory for the first node
    p = head;
    head->next = NULL;
    printf("\n Enter power and coefficient : ");
    scanf("%d %f", &(p->pow), &(p->coeff));
    for(i = 1; i<n; i++)
    {
      //create subsequent nodes
      p->next = (pnode*)malloc(sizeof(pnode));
      p = p->next;
      p->next = NULL;
      printf("\n Enter power and coefficient :");
      scanf("%d%f", &(p->pow), &(p->coeff));
    }
    return(head);
}
```

'C' function to print a polynomial.

Printing of a polynomial is similar to printing a linked list.

```
void print(pnode *p)
{
    printf("\n ");
    while(p! = NULL)
    {
        printf(" %5.2fx^ %d \t", p->coeff, p->pow);
        p = p->next;   //goto the next node
    }
}
```

6.6.2 Addition of Two Polynomials

Q. Write pseudo C/C++ code for polynomial addition using singly linked list. **SPPU - May 18, 6 Marks**

In order to add two polynomials denoted by "P" and "q", both the linked lists are scanned term by term. Whenever the powers of the current terms of "P" and "q" are same, their coefficients are added to obtain the coefficient of the term of the new polynomial.

The exponent part of this term is the same as that of the current term of either of "P" or "q".

Subsequently, both pointers pointing to the current nodes are advanced to point to the next node. If the power part of one polynomial, say P, is less than a new node is created which is a copy of the term of "P" and this new node is added to the resultant list.

When one of the lists of "P" or "q" is exhausted then the remaining nodes of the other list is simply copied to the resultant list.

First polynomial	$5 + 9x + 10x^2 + 13x^5 + 15x^7$	(p_1)
Second polynomial	$6 + 13x^2 + 9x^3$	(p_2)
Resultant polynomial	$11 + 9x + 23x^2 + 9x^3 + 13x^5 + 15x^7$	$(p_1 + p_2)$

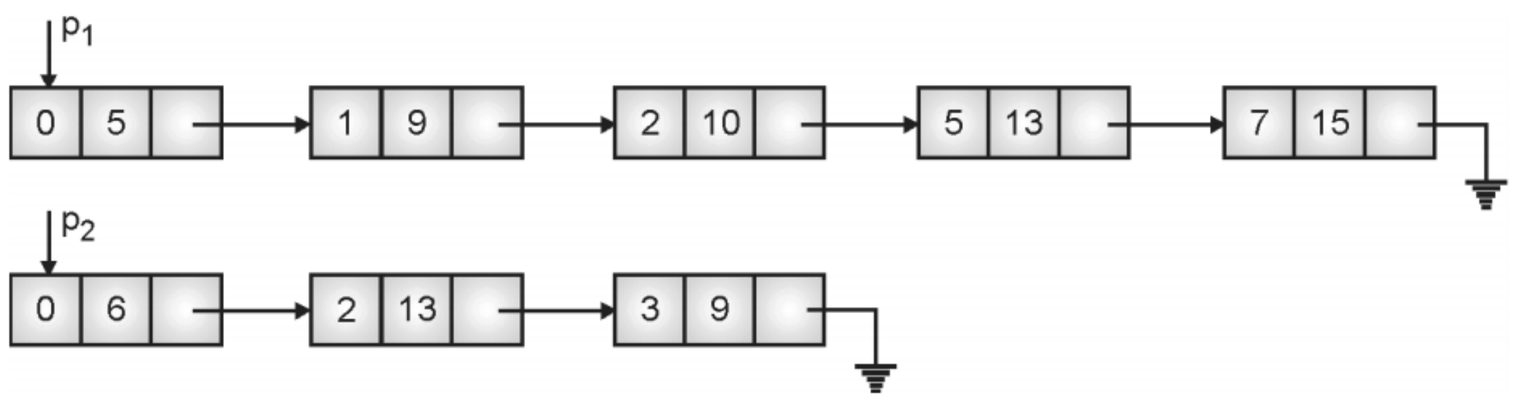


Fig. 6.6.2 : Representation of two polynomial using linked list

Tech Knowledge Publications

Since, the power of the terms pointed by p_1 and p_2 are same, coefficients are added and the new term is inserted into the resultant polynomial p_3. Pointer r_3 helps in inserting, subsequent terms at the rear end of the resultant polynomial.

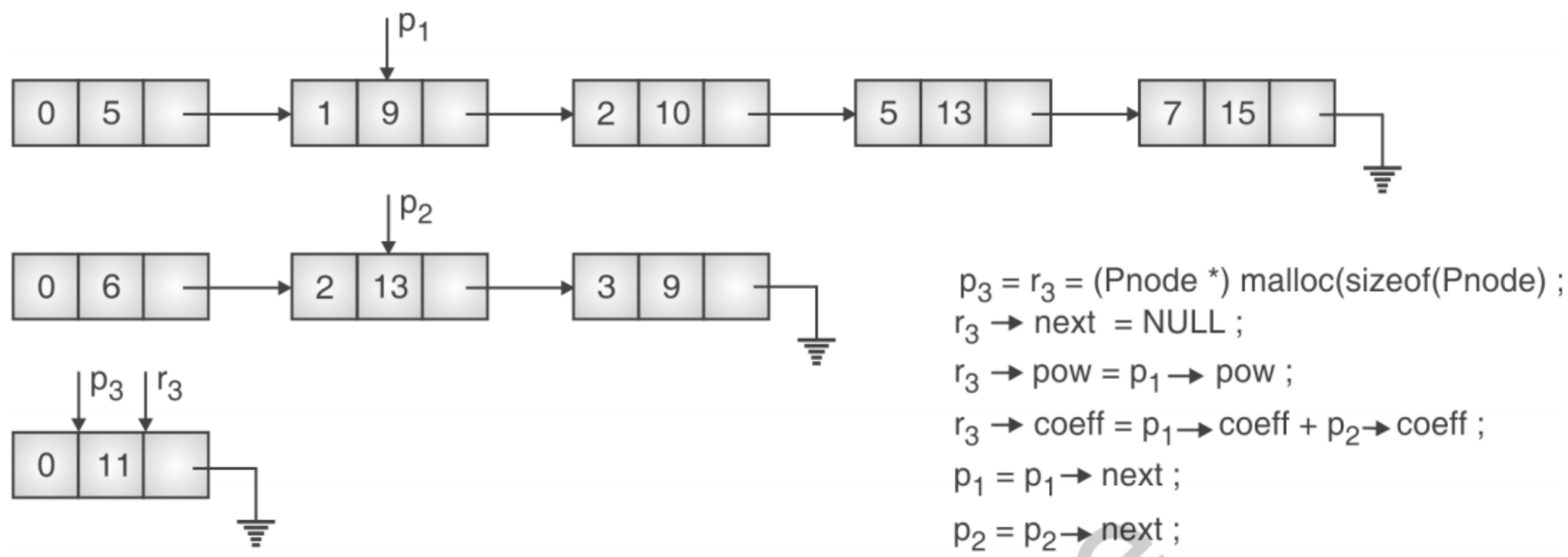


Fig. 6.6.2(a)

Since, the power of the term pointed by p_1 is less than the power of the term pointed by p_2, term pointed by p_1 is added to p_3. Term is added as a next node of r_3. p_1 and r_3 are advanced by a node.

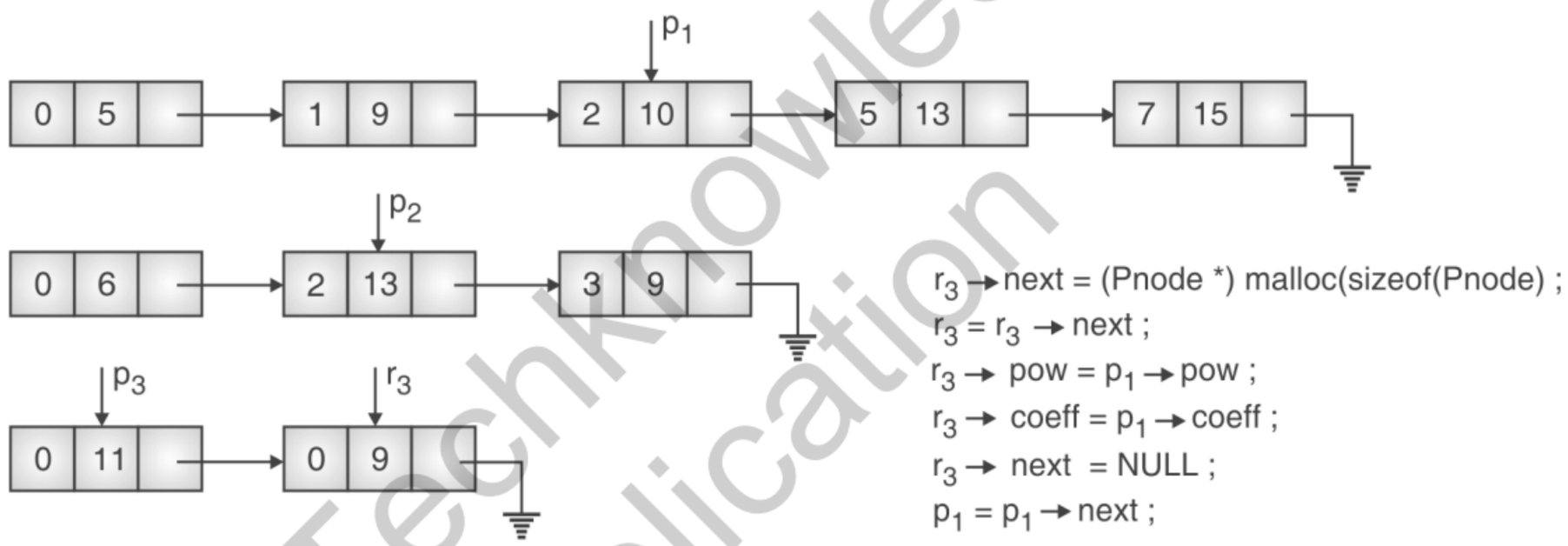


Fig. 6.6.2(b)

Coefficients of the terms pointed by p_1 and p_2 are added and the new term is inserted into p_3.

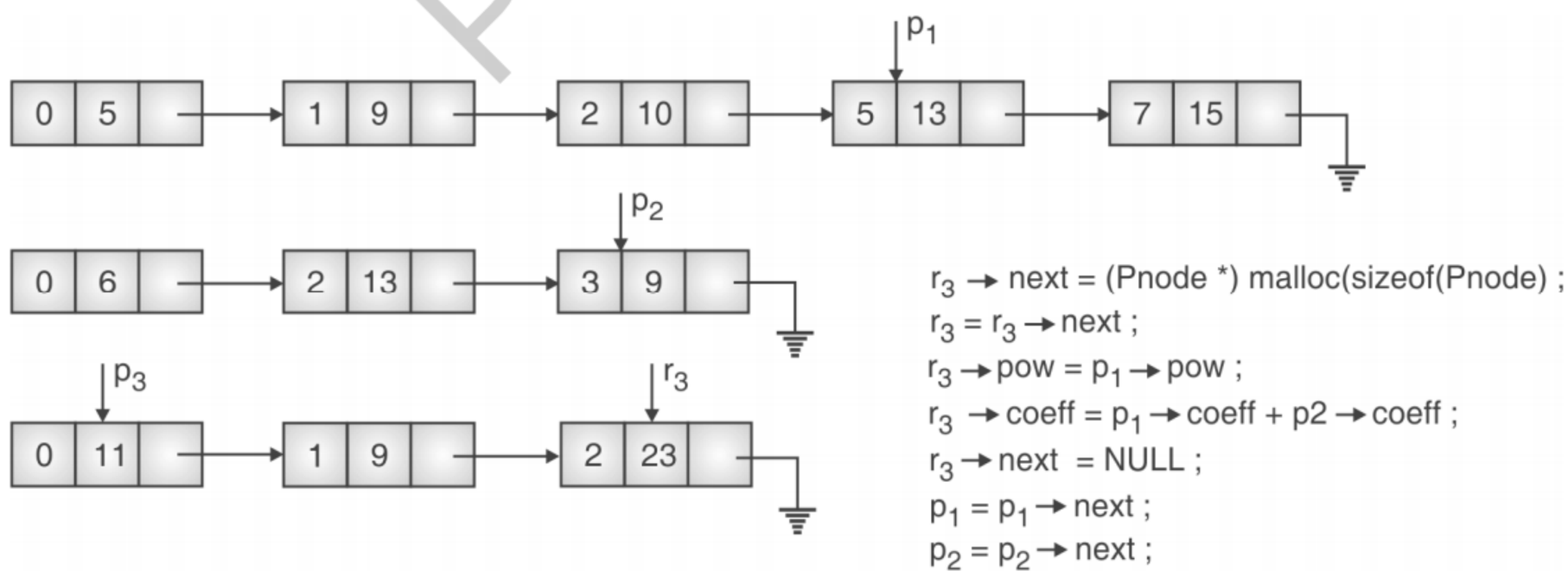


Fig. 6.6.2(c)

Since the power of the term pointed by p_2 is less than the power of the term pointed by p_1, term pointed by p_2 is inserted into p_3.

Technknowledge Publications

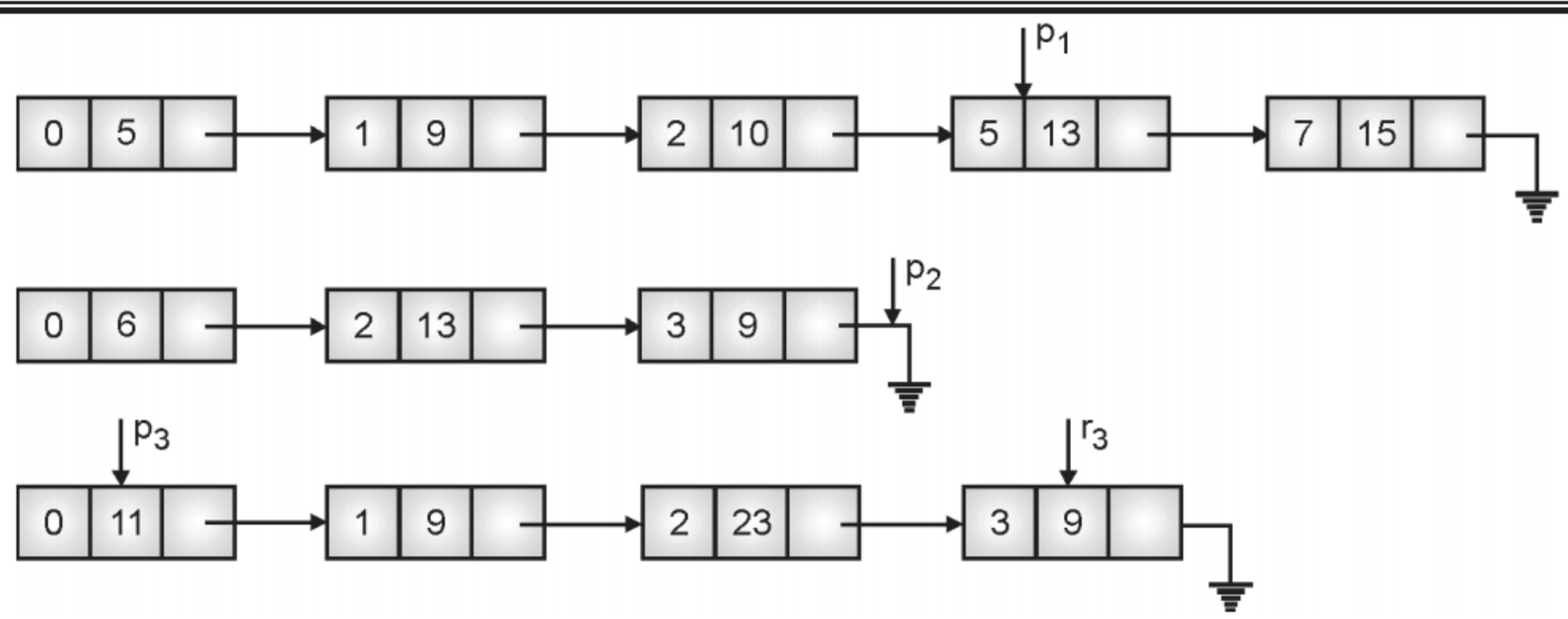


Fig. 6.6.2(d)

Since no more terms are left in p_2, all the terms of p_1 are inserted at the end of p_3.

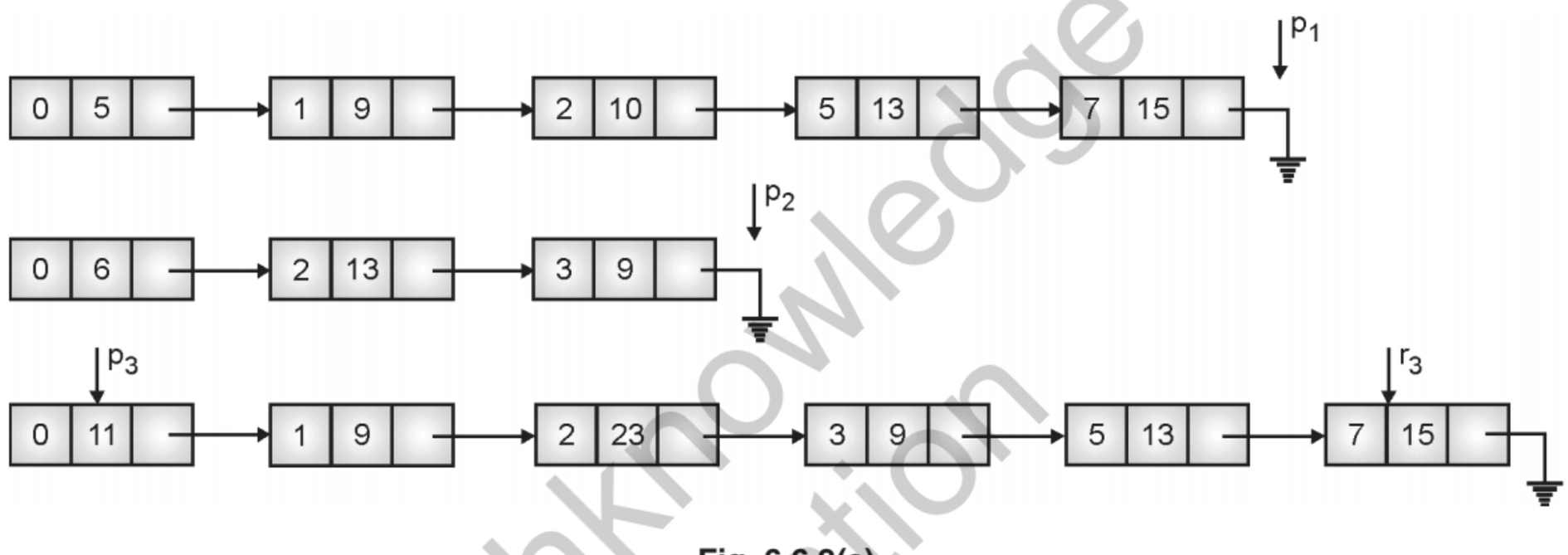


Fig. 6.6.2(e)

```
while(p1 ! = NULL)
{
    r3 -> next = (pnode*) malloc(sizeof(pnode));
    r3 = r3 -> next;
    r3 -> pow = p1 -> pow;
    r3 -> coeff = p1 -> coeff;
    r3 -> next = NULL
    p1 = p1 -> next;
}
```

'C' function for addition of two polynomials.

```
pnode *addpoly(pnode *p1, pnode *p2)
{
    pnode *p3, *r3;
    p3 = NULL;
    while(p1! = NULL && p2! = NULL)
    {
        if(p3 == NULL)
        {
            p3 = r3 = (pnode*)malloc(sizeof(pnode));
            r3->next = NULL;
        }
        else
        {
            r3->next=(pnode*)malloc(sizeof(pnode));
            r3 = r3->next;
            r3->next = NULL;
        }
        if(p1->pow<p2->pow)
        {
            r3->pow = p1->pow;
            r3->coeff = p1->coeff;
            p1 = p1->next;
        }
        else
            if(p2->pow<p1->pow)
            {
                r3->pow = p2->pow;
                r3->coeff = p2->coeff;
```

TechKnowledge Publications

```
            p2 = p2->next;
        }
        else
        {
            r3->pow = p1->pow;
            r3->coeff = p1->coeff + P2->coeff;
            p1 = p1->next;
            p2 = p2->next;
        }
    }
    //insert the remaining terms of P1 into P3
    while(p1 ! = NULL)
    {
        if(p3 == NULL)
        {
            p3 = r3 = (pnode*)malloc(sizeof(pnode));
            r3->next = NULL;
        }
        else
        {
            r3->next = (pnode*)malloc(sizeof(pnode)) ;
            r3 = r3->next;
            r3->next = NULL;
        }
        r3->pow = p1->pow;
        r3->coeff = p1->coeff;
        p1 = p1->next;
    }
    // insert the remaining terms of p2 into p3
    while(p2! = NULL)
    {
        if(p3 == NULL)
        {
            p3 = r3 = (pnode*)malloc(sizeof(pnode));
            r3->next = NULL;
        }
        else
        {
            r3->next=(pnode*)malloc(sizeof(pnode));
            r3 = r3->next;
            r3->next = NULL;
        }
        r3->pow = p2->pow;
        r3->coeff = p2->coeff;
        p2 = p2->next;
    }
    return(p3);
}
```

Timing complexity for addition of two polynomial assuming n terms in first polynomial and m terms in order is O (m + n)

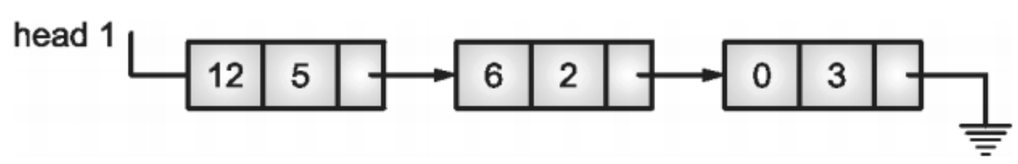
Ex. 6.6.1 : Consider the following polynomials represented using linked lists.

$C1 = 5x^{12} + 2x^6 + 3$, $C2 = 6x^{12} - 5x^8 + 12x^5$

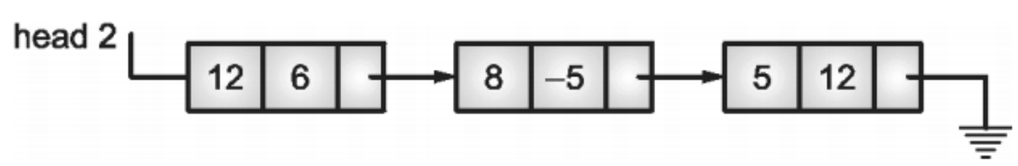
Show the addition process of above polynomials diagrammatically.

Soln. :

Representation of C1



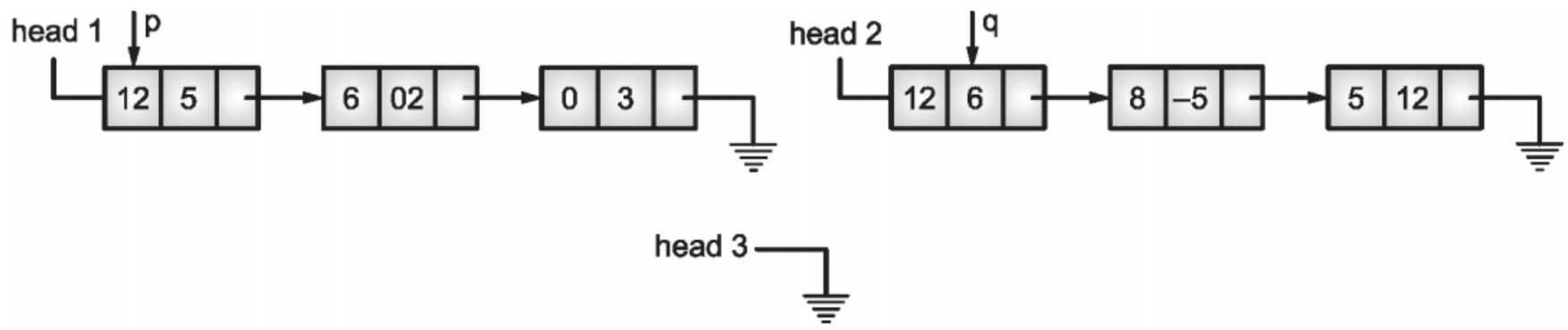
Representation of C2



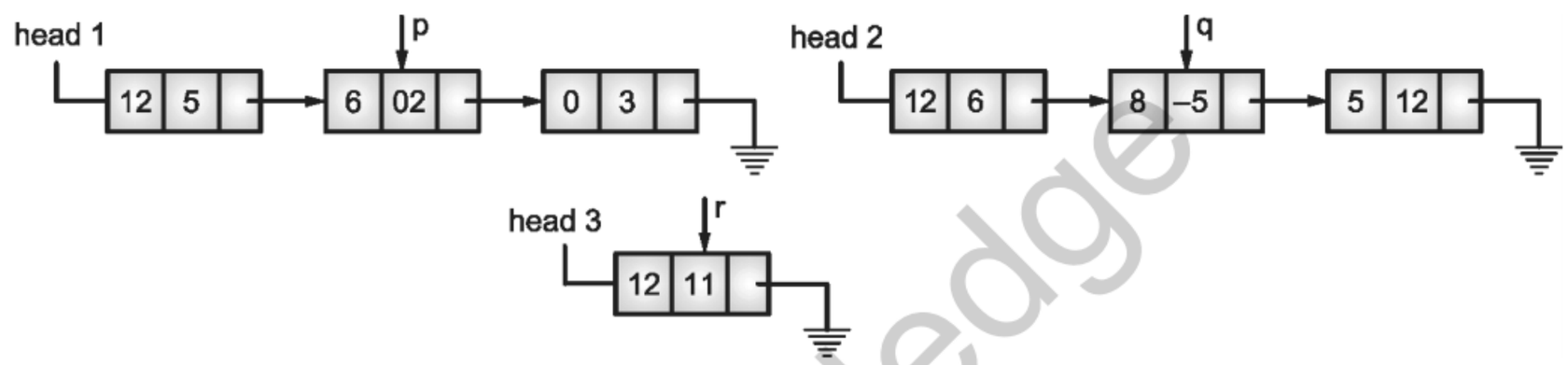
Techknowledge Publication

Addition :

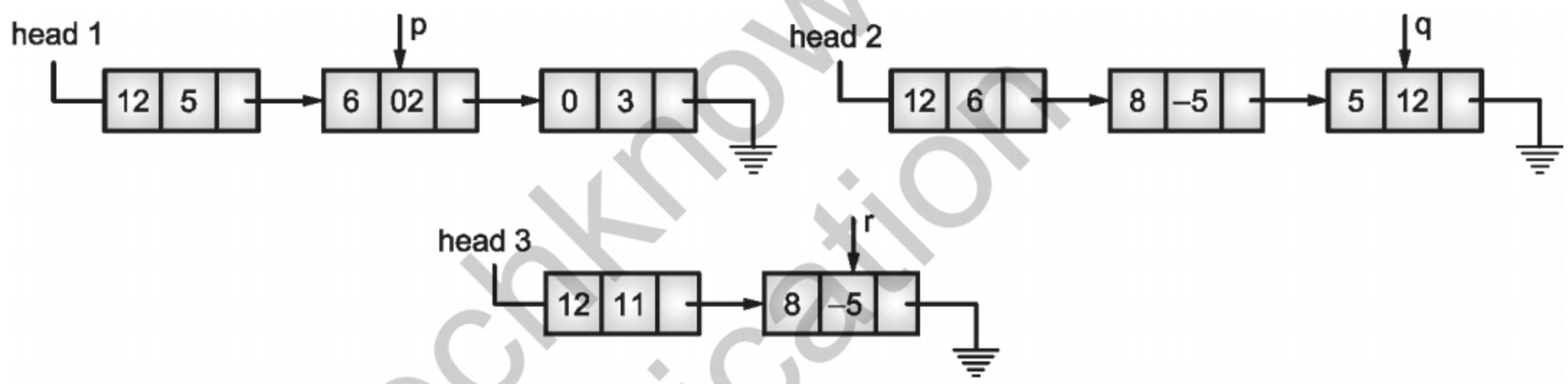
Final result is stored in a linked list reference by head 3



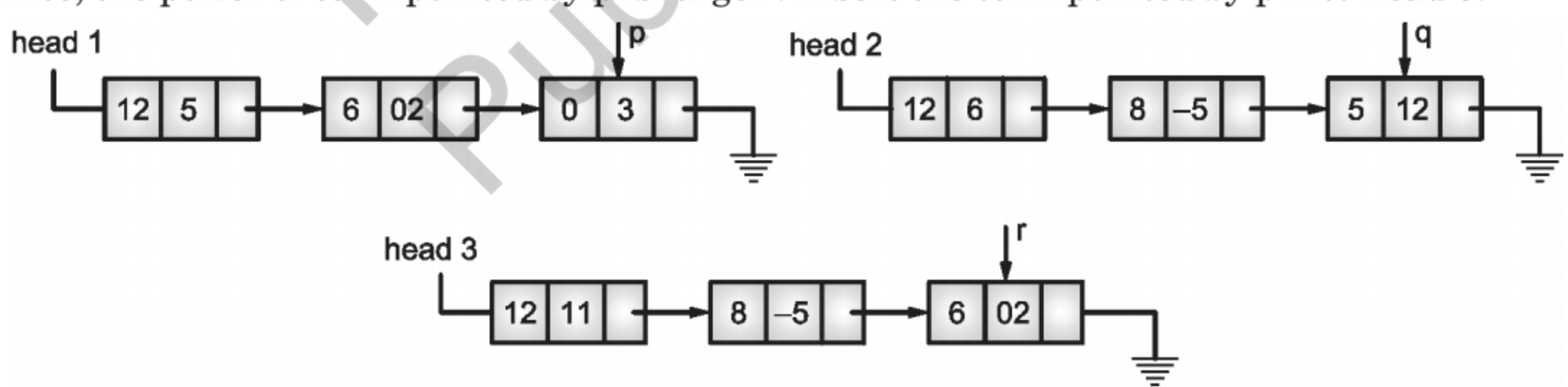
Step 1 : Since, the power of terms pointed by p and q are same : Add the terms and insert the resultant term in head 3.



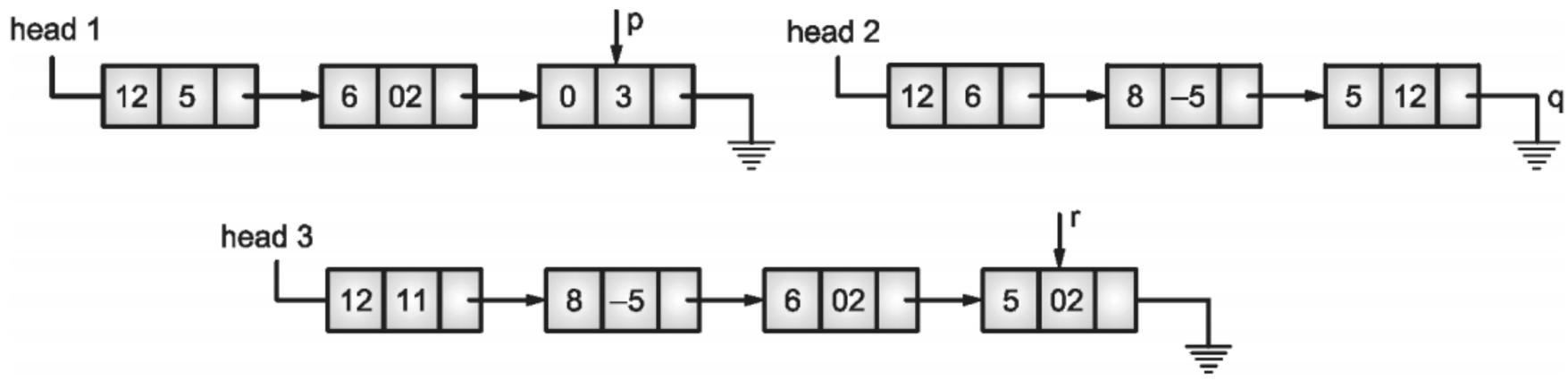
Step 2 : Since, the power of term pointed by q is larger : Insert the term pointed by q into head 3.



Step 3 : Since, the power of term pointed by p is larger : Insert the term pointed by p into head 3.

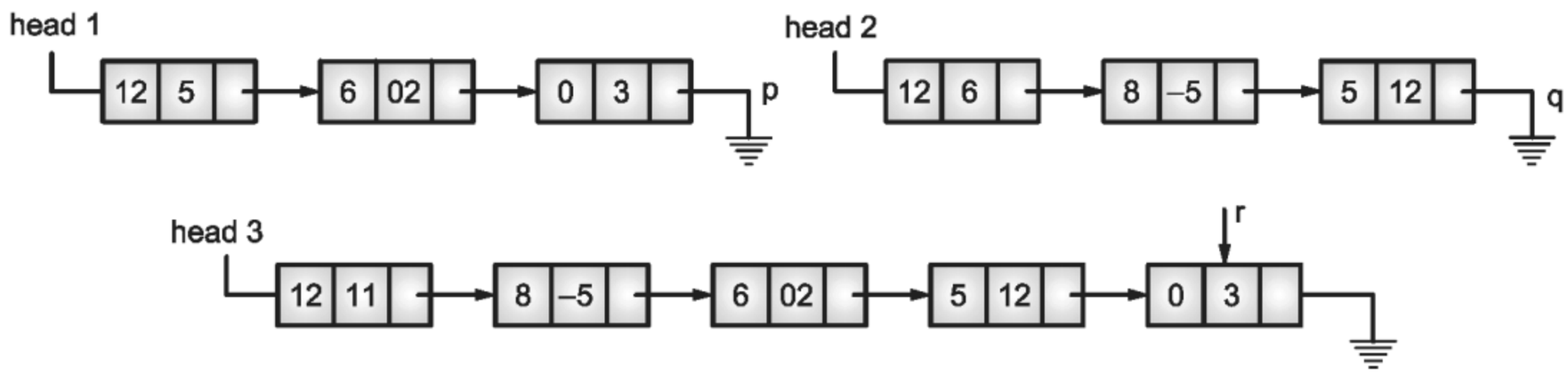


Step 4 : Since, the power of term pointed by q is larger : Insert the term pointed by q into head 3.



Techknowledge Publications

Step 5 : Since, the pointer q has become NULL : remaining terms of head 1 are inserted into head 3.



Final polynomial = $11x^{12} + (-5)x^8 + 02x^6 + 12x^5 + 3$

Multiplication of two polynomials

$5 + 2x + 9x^3 + 3x^5$	P1
$3 + x^2 + 2x^4$	P2
$15 + 6x + 27x^3 + 9x^5$	P3[P1 is multiplied with 3]
$5x^2 + 2x^3 + 9x^5 + 3x^7$	P4[P1 is multiplied with n^2]
$10x^4 + 4x^5 + 18x^7 + 6x^9$	P5[P1 is multiplied with $2n^4$]

P3, P4 and P5 are added to produce the resultant polynomial.

Algorithm for multiplication of two polynomials P1 and P2.

```
R3 and temp are of polynomial type.
R3 ← NULL
Temp ← NULL
while(P2! = NULL)
{
    multiply the current term of P2 with the polynomial
    P1 with its result in temp
    temp ←(polynomial P1) × (term pointed by P2) add
    the polynomial "temp" to the final polynomial "R3"
    R3 ← R3 + temp;
    P2 ←(P2 → next)      //skip the current term of P2
}
```

'C' function for multiplication of two polynomials.

```
pnode * multiply(pnode *P1, pnode *P2)
{
    pnode * temp, *r3;
    r3 = NULL;
    while(P2! = NULL)
    {
        temp = multiterm(P1, P2);
        /* function multiterm multiplies the polynomial
        P1 with the current term of P2 */
        r3 = addpoly(r3, temp); ]
        P2 = P2 → next;
    }
    return(r3);
}
pnode * multiterm(pnode * P1, pnode *x)
{
    pnode * new, * r1;
    new = NULL;
    while(P1 ! = NULL)
    {
        if(new == NULL)
        {
            r1=new=(pnode *) malloc(sizeof(pnode));
            r1 → next = NULL;
        }
        else
        {
            r1→next=(pnode*) malloc(sizeof(pnode));
            r1 = r1 → next;
            r1 → next = NULL;
        }
        r1 → coeff = P1 → coeff * x → coeff;
        r1 → pow = P1 → pow + x → pow;
        P1 = P1 → next;
    }
    return(new);
}
```

Techknowledge Publications

'C' function for evaluation of a polynomial.

```
float evaluate(pnode * P1, float val)
{
    float x;
    x = 0.00;
    while(P1 ! = NULL)
    {
        x = x + P1 → coeff * pow(val, P1 → pow);
        P1 = P1 → next;
    }
    return(x);
}
```

6.7 Concept of Skip List

A list can be represented using a linked list. The time required to search an element is proportional to the number of nodes that have to be examined. If the list of elements is sorted and it is stored in an array, we can quickly search an element using binary search. Binary search required **log n** comparisons to search an element. In a simple linked list, sorted, behaviour of data cannot be exploited to quickly search an element.

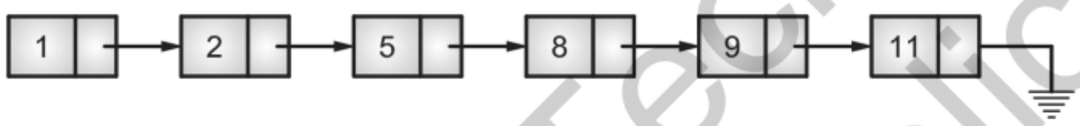


Fig. 6.7.1 : A simple linked list

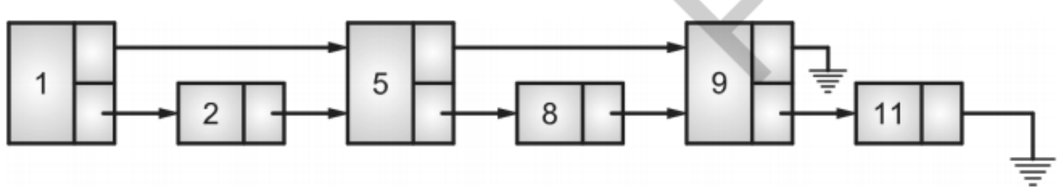


Fig. 6.7.2 : Linked list with pointers to two cells ahead

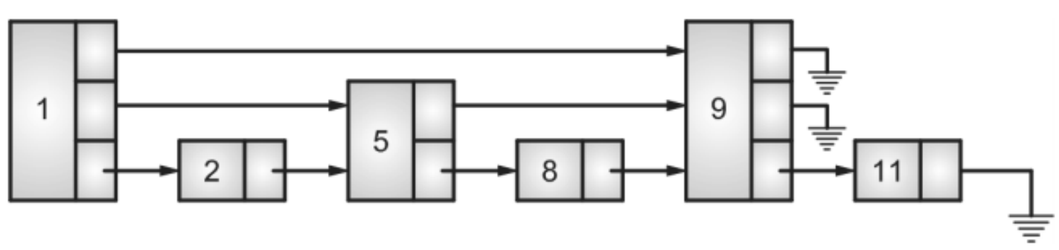


Fig. 6.7.3 : Linked list with pointers to four cells ahead

Fig. 6.7.1 shows a linked list in which alternate node has an additional pointer to the node two ahead of it in the list.

Because of this, at most N/2 + 1 nodes are examined to search an element.

Above idea can be extended as shown in the Fig. 6.7.2. Here, every fourth node has a pointer to the node four ahead. Because of this, at most N/4 + 2 nodes are examined to search an element.

In general, above concept can be extended and every 2^{ith} node can have a pointer to the node 2^i ahead of it. This will require examining of log N node on an average to search an element.

6.8 Generalized Linked List

Q. Explain Generalized linked list with example.
SPPU - Dec. 17, May 18, 3 Marks

Q. Explain the concept of Generalized linked list.
SPPU - May 19, 2 Marks

6.8.1 Introduction

A linear list is defined as a finite sequence of elements (n> = 0) a_1, a_2, ... a_n, which is denoted by A = (a_1, a_2, ... a_n). Elements of the linear list are atomic in nature. Elements of the linear list can be addressed by its index.

- A = ( ) : An empty or null list.
- B = (a, b, c) : A list of three elements with B(1) = a, B(2) = b and B(3) = C.

In a **Generalized** list, each element a_i is either an atom or a list itself.

- C = (a, (b, c), d) : A list of three elements.

  First element 'a' is atomic.

  Second element (b, c) is list.

  Third element 'd' is atomic.

Thus, a generalized list, A is finite sequence of n > = 0 elements a_1, a_2, ... a_n where each a_i is either an atom or a list.

- a_1 is said to be the **head of A.**
- (a_2 ... a_n) is called the **tail of A.**

Tech Knowledge Publication

6.8.2 Representation of Generalized Lists

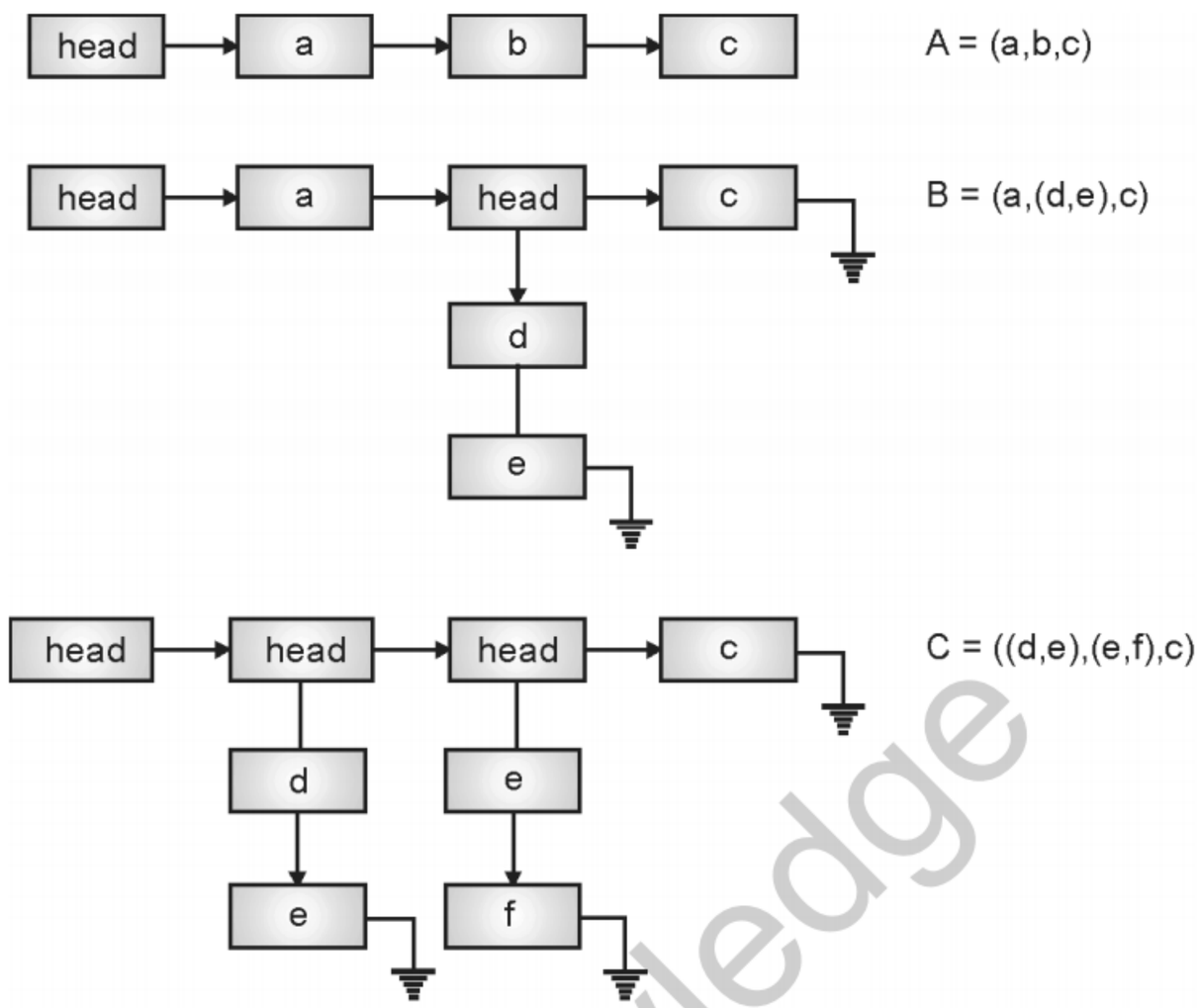
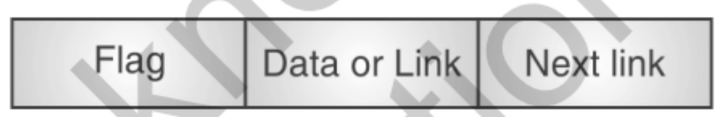


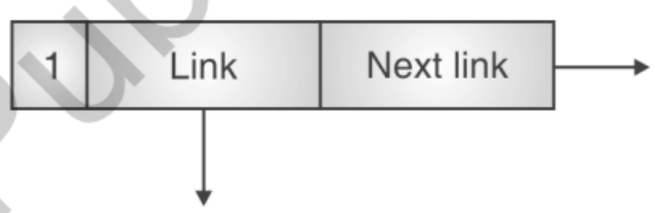
Fig. 6.8.1 : Linked representation of the lists A, B and C

– Each node of a linked list can either be a head node or a data node (for atoms).



– When flag = 0, the node is a data node.

– When flag = 1, the node is a head node.



The field flag, identified whether the node is a data node or a head node.

Ex. 6.8.1 : Give representation of the generalized list. A = (1, 2, (3, (4, 5)), 6)

Soln. :

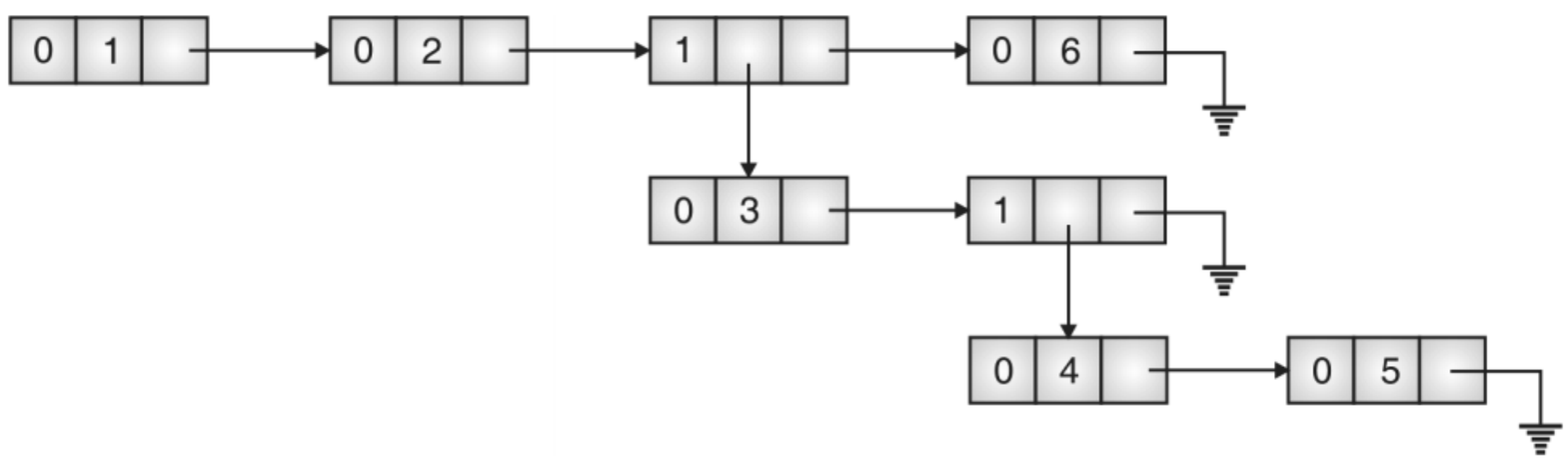


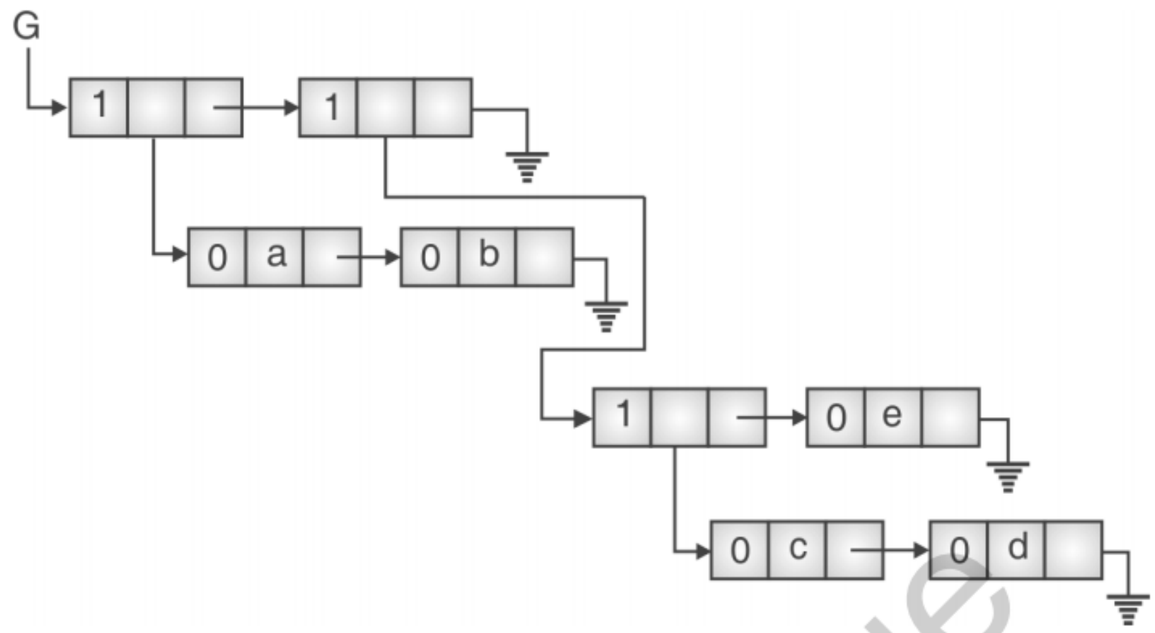
Fig. Ex. 6.8.1 : Linked representation of the list A

It is assumed that if flag is set to 0 then the entity is atomic; if flag is set to 1 then the entity is a list.

Ex. 6.8.2 : Represent the following GLL : (i) G = ((a, b), ((c, d), e)) (ii) (p, q, (r, s, (t, u, v), w), x, u)

Soln. :

(i) Representation of ((a, b), ((c, d), e))



(ii) Representation of (p, 2, (r, s, (t, u, v), w), x, u)

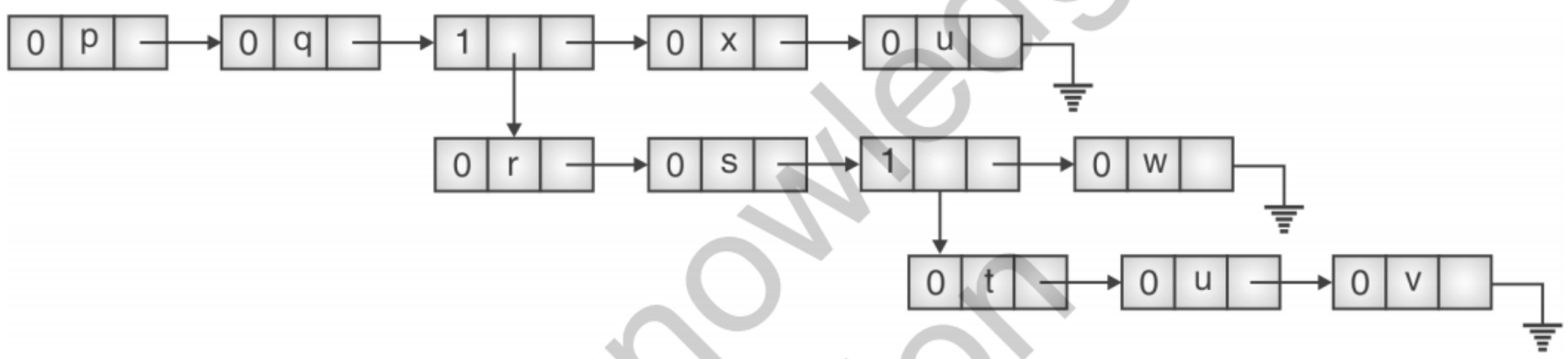
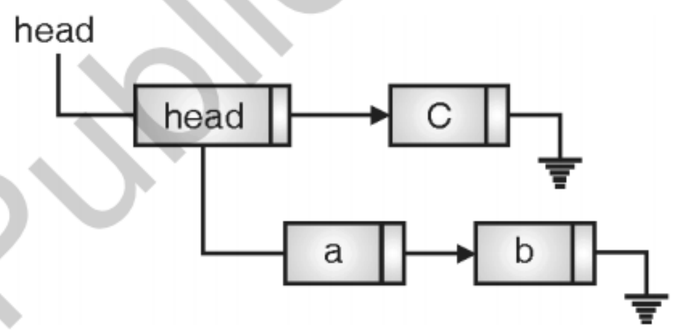


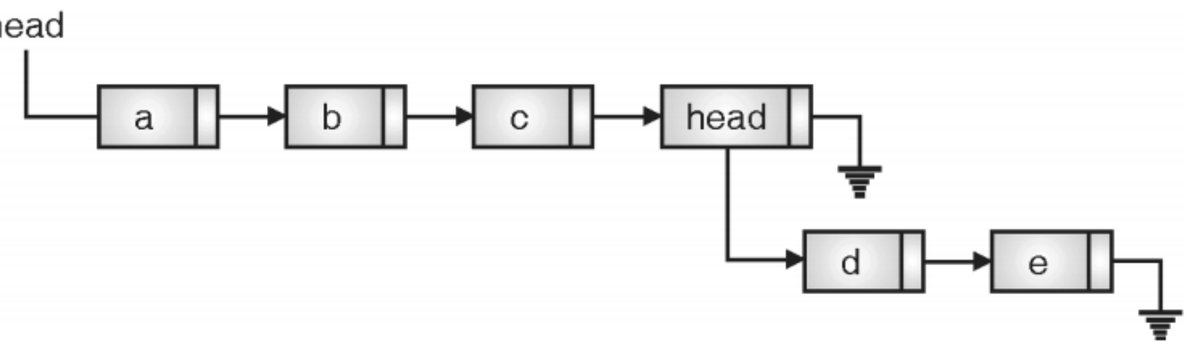
Fig. Ex. 6.8.2

Ex. 6.8.3 : Represent the following lists using generalized linked list :(i) ((a, b), c) (ii) (a, b, c, (d, e))

Soln. : (i) ((a, b), c)



(ii) (a, b, c, (d, e))



Ex. 6.8.4 : Write data structure to represent the following generalized lists using linked list and represent them
(i) (a, (b, c), d) (ii) (p, q, (r, s, (t, u), v))

Soln. : Structure of the node

```
typedef_struct node
{
    int flag;
    struct node *next;
    union
    {
        char data;
        struct node *down;
    } shared;
}node;
```

– When flag = 0, the node is data node.

– When flag = 1, the node is a head node.

(i) Representation of (a, (b, c), d)

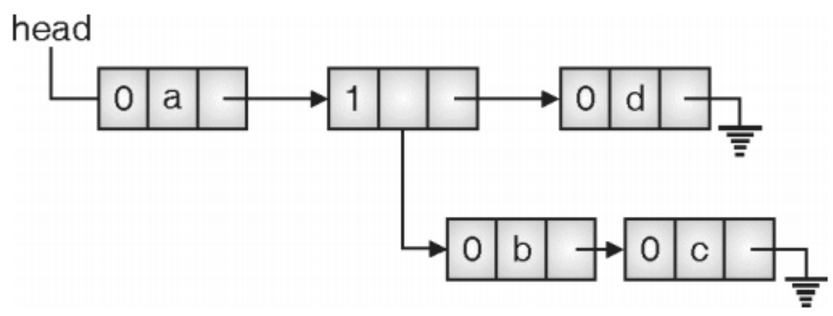


Fig. Ex. 6.8.4

(ii) Representation of (p, q, (r, s, (t, u), v))

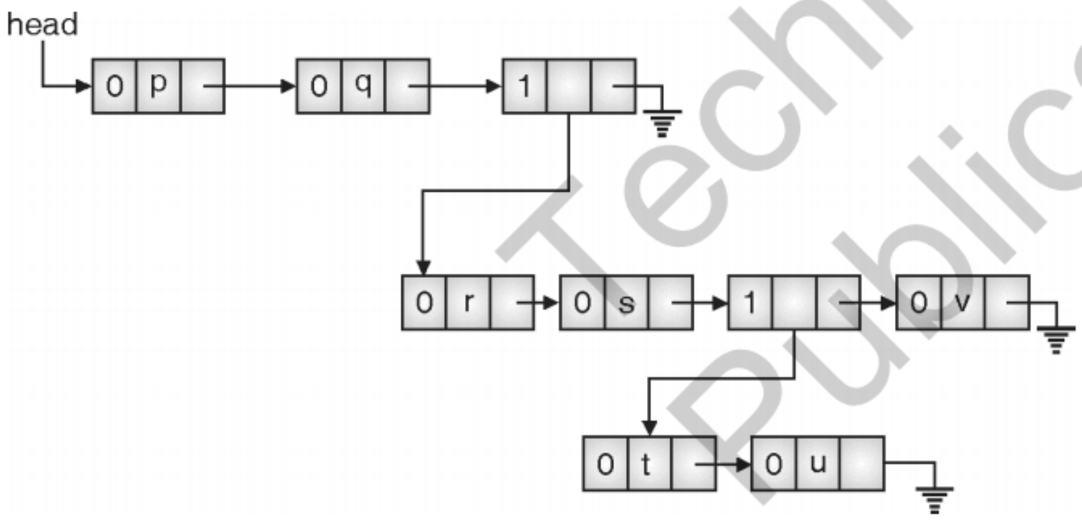


Fig. Ex. 6.8.4(a)

Ex. 6.8.5 : Represent the following GLL : ((a, b), (c, d), e)

Soln. : ((a, b), (c, d), e)

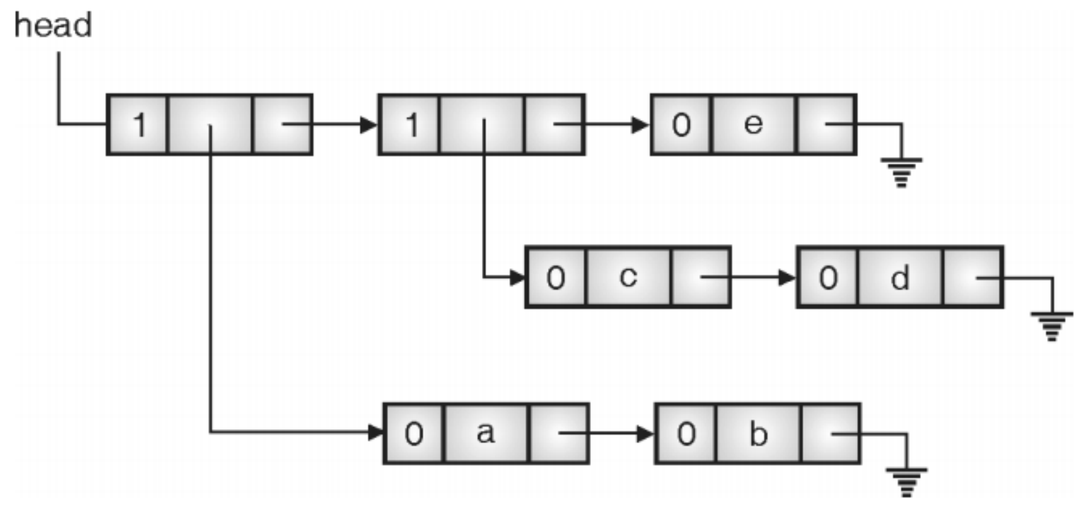


Fig. Ex. 6.8.5

Ex. 6.8.6 : Write data structure to represent a node of a generalized linked list. Represent the following lists :

(i) (a, b, c, (d, e, f), g, h)

(ii) (p, (q, r), s, (t, u), v).

Soln. : Structure of node

```
typedef struct node
{
    int flag;
    struct node * next;
    union
    {
        char data;
        struct node * down;
    } shared;
} node;
```

– When flag = 0, the node is data node

– When flag = 1, the node is head node.

(i) (a, b, c (d, e, f), g, h)

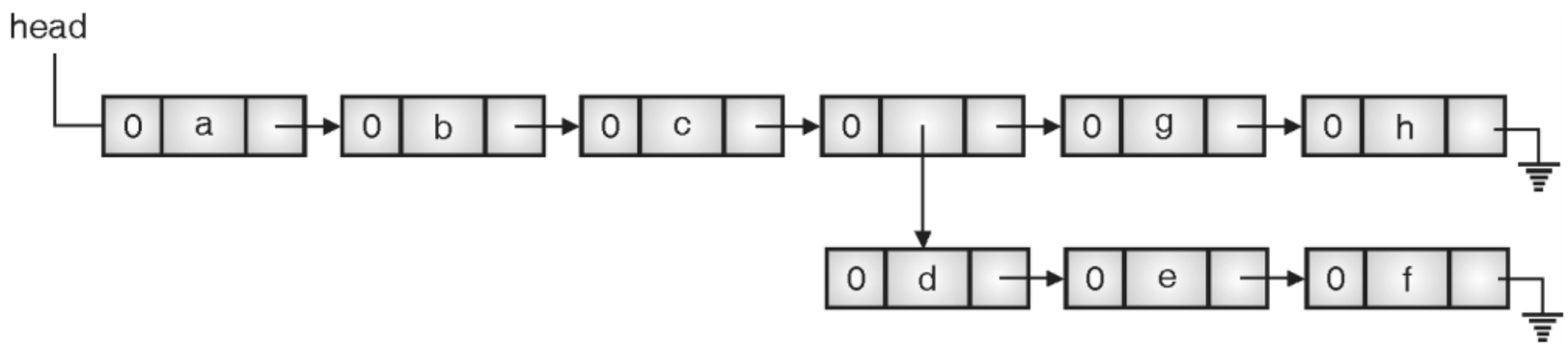


Fig. Ex. 6.8.6

Techknowledge Publication

(ii) (p, (q, r), s, (t, u), v)

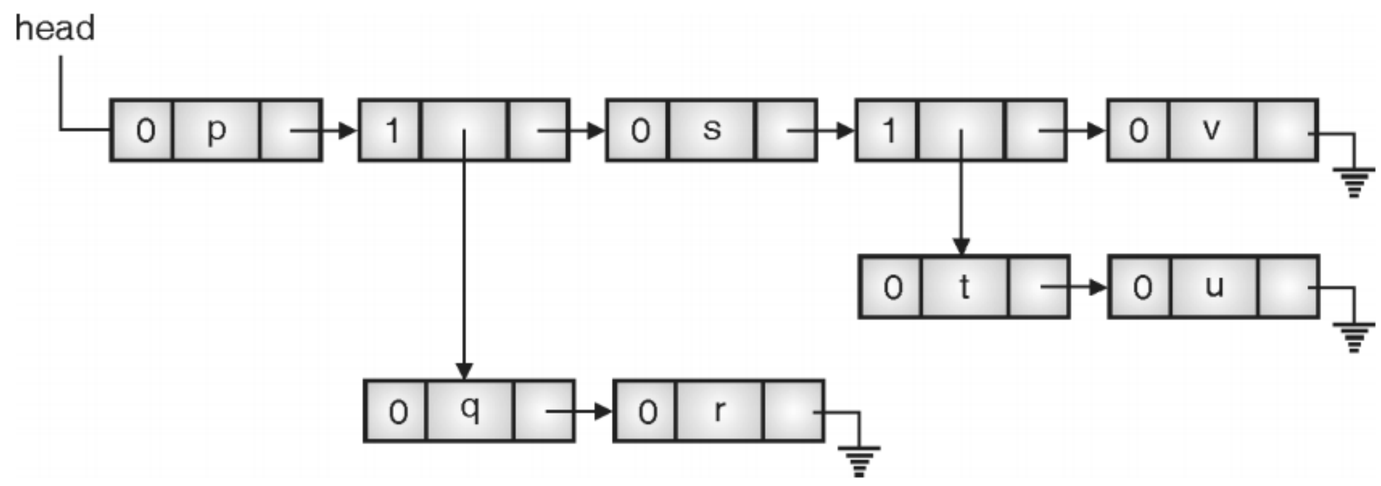


Fig. Ex. 6.8.6(a)

Ex. 6.8.7 : Represent following GLL. (1) ((a,b) , c, (d,e)) (2) (p,q, (r,(t,u,s),v,w))

Soln. :

(1) ((a,b) , c, (d,e))

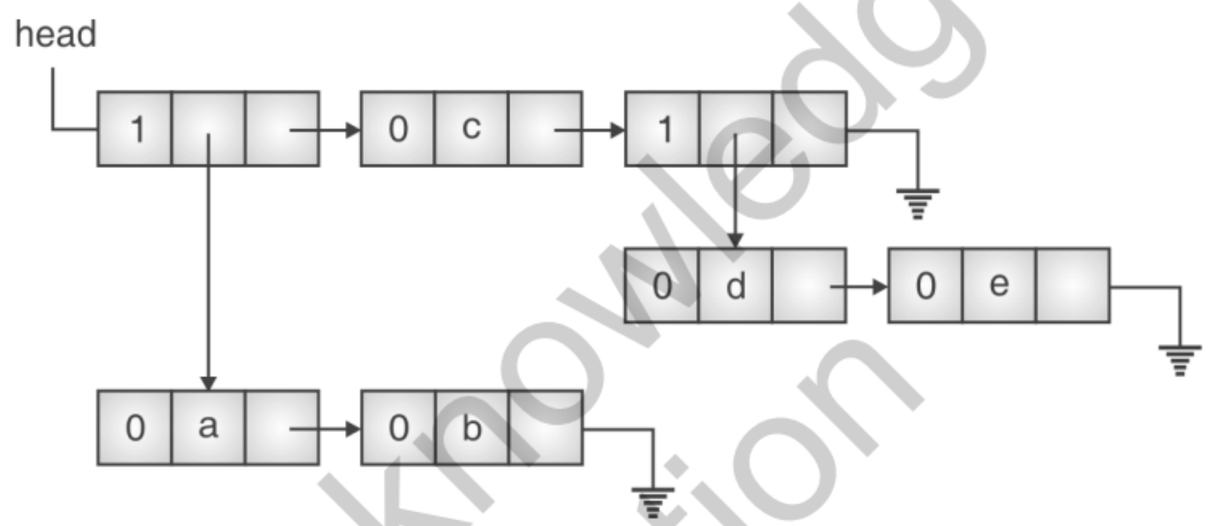


Fig. Ex. 6.8.7

(2) (p,q,(r,(t,u,s),v,w))

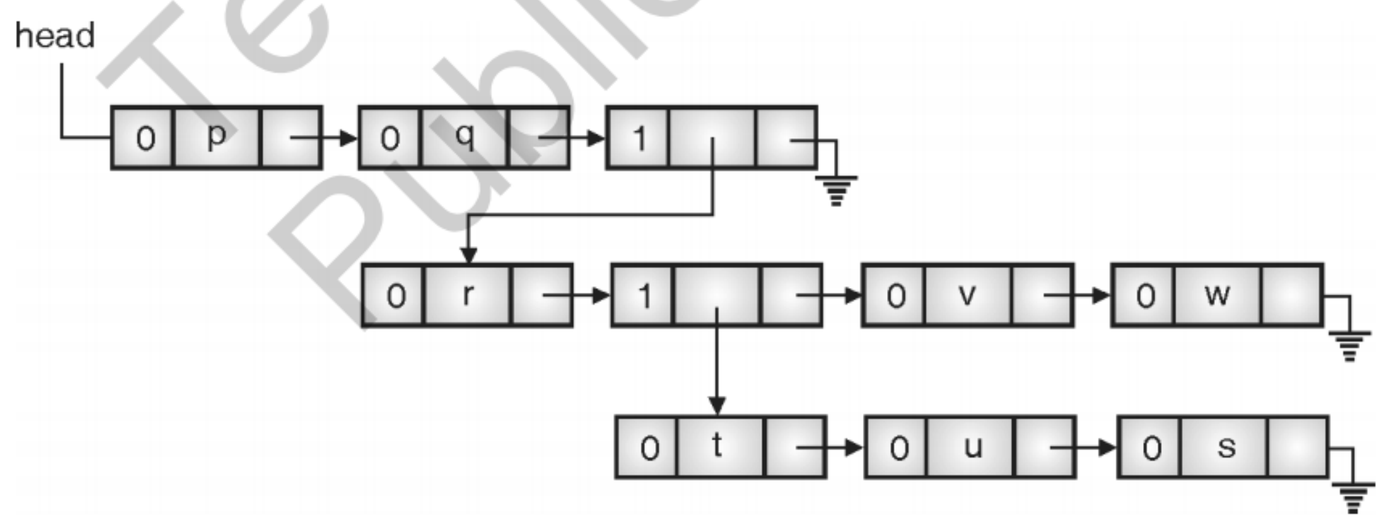


Fig. Ex. 6.8.7(a)

Ex. 6.8.8 : Represent following polynomial using GLL . (L, (M, (N, (O, P)), Q), R, (S, T), (A, (B, C)))

Soln. : Step 1 :

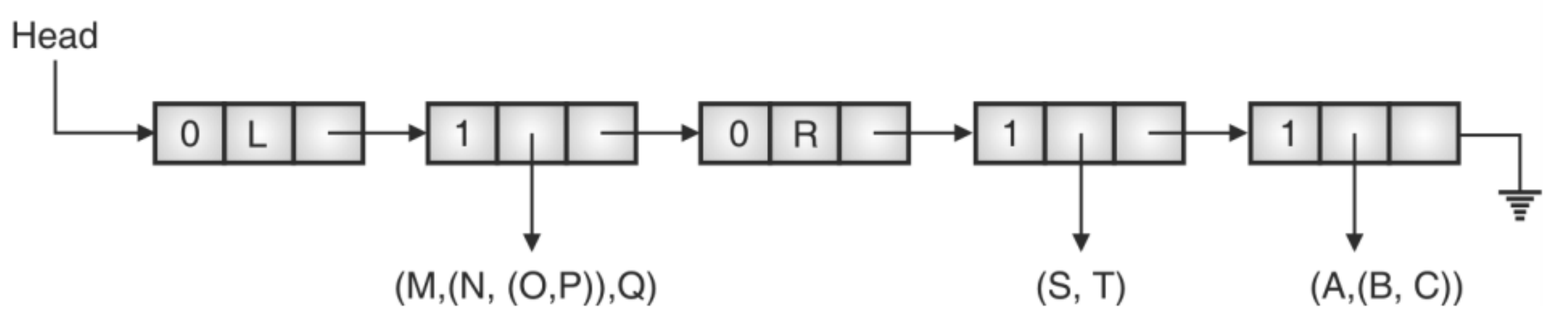


Fig. Ex. 6.8.8

Step 2 :

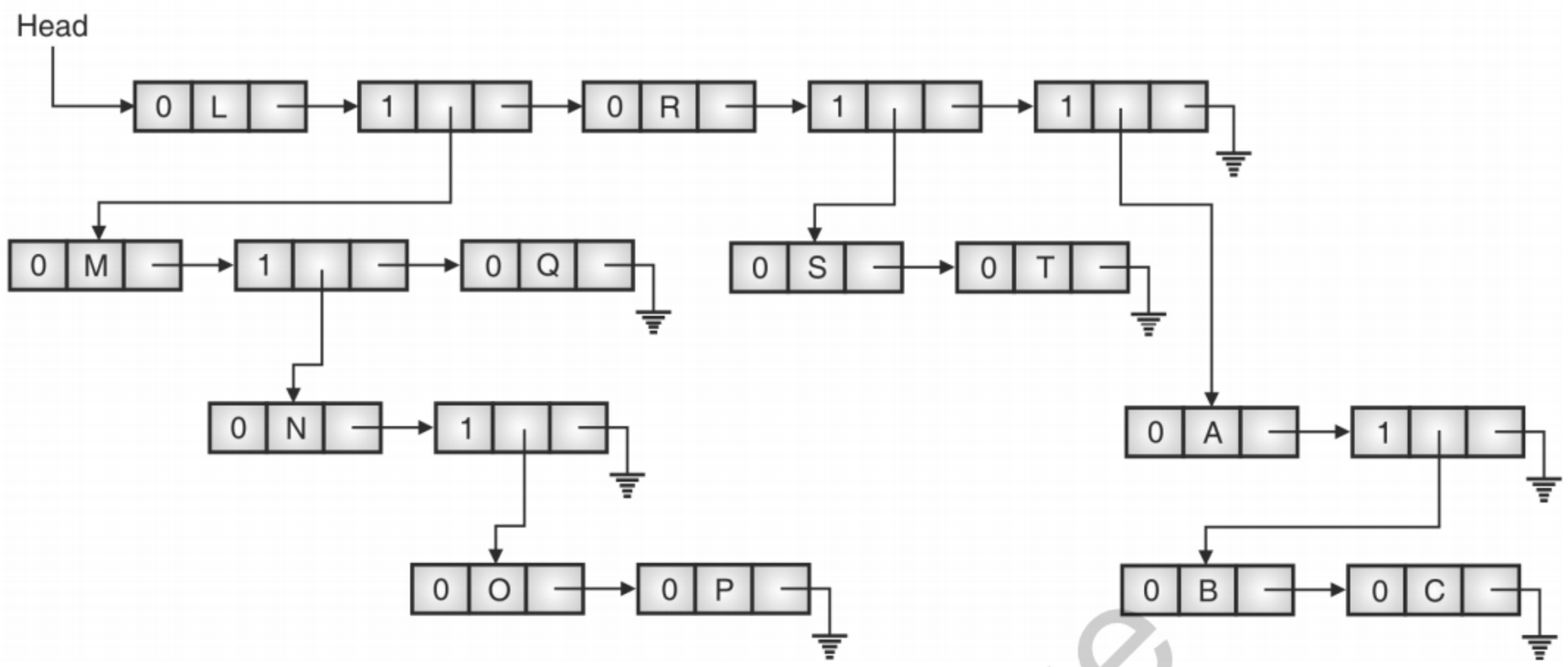


Fig. Ex. 6.8.8(a)

Ex. 6.8.9 : Represent the following polynomial by using generalized linked list: (a, b, (c, d (e, g), h), (f))

SPPU - May 17, 3 Marks

Soln. :

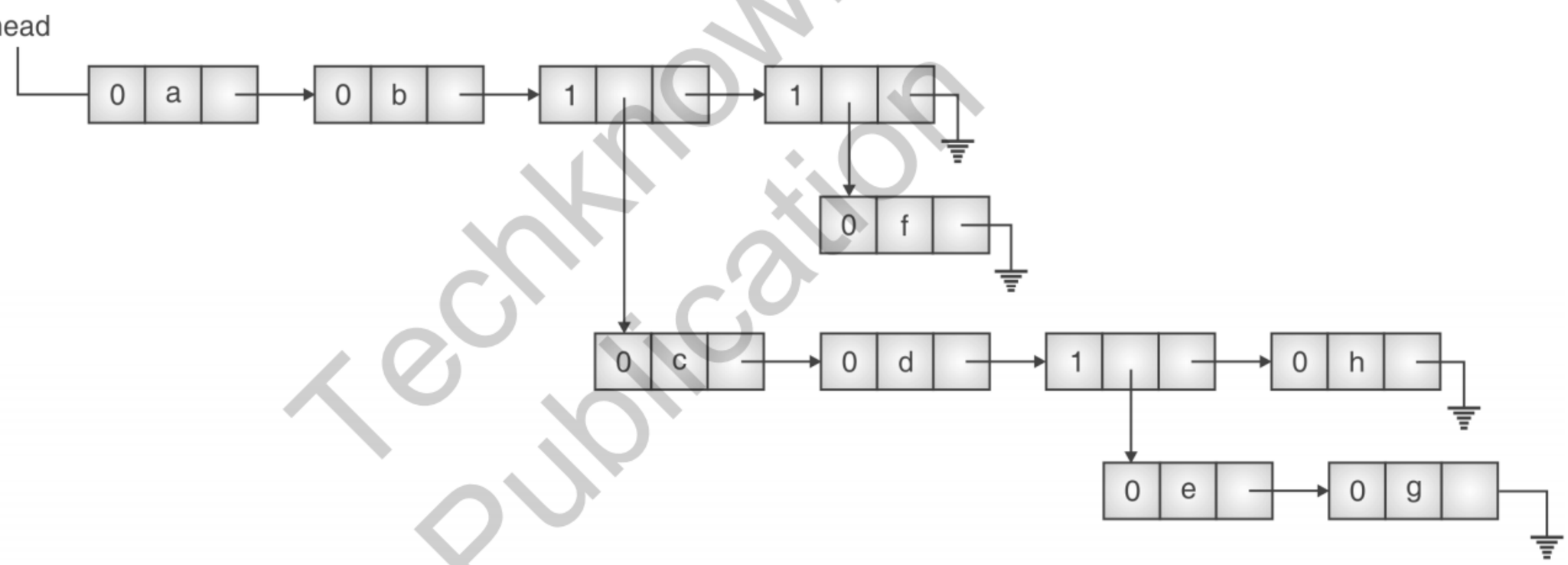


Fig. Ex. 6.8.9

Ex. 6.8.10 : Represent the following using GLL : (p, q, (r, s (u, v), w), (x, y))

SPPU - Dec. 18, 3 Marks

Soln. :

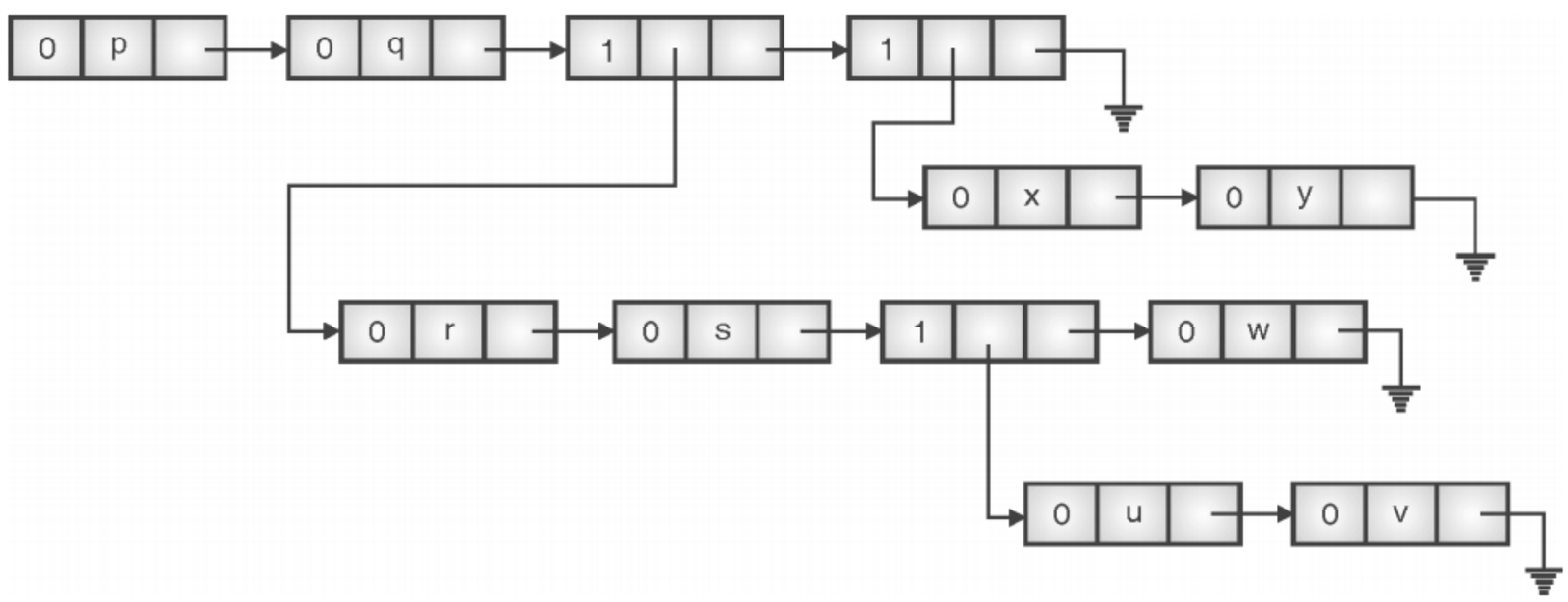


Fig. Ex. 6.8.10

Techknowledge Publication

6.9 Representation of Polynomial using Generalized List

A polynomial can be represented using a generalized list.

A polynomial in a single variable can be represented using a simple list.

$$p(x) = 5x^6 + 9x^2 + 3x + 7$$

Each term of a polynomial can be represented using a 2-tuple, (power, coefficient).

Thus $5x^6 + 9x^2 + 3x + 7$ can be represented as a list of tuples :

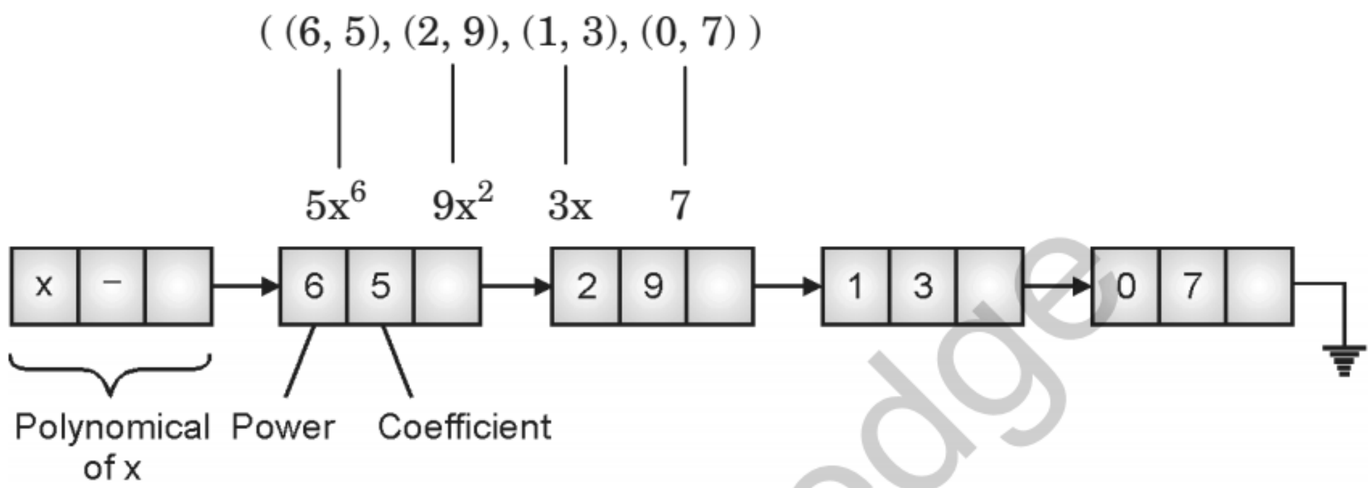


Fig. 6.9.1 : A single variable polynomial

A node consists of three fields :

1. Variable name or power
2. Coefficient
3. Address of the next node

A polynomial in two variables

Let us consider a polynomial as given below : $y^3 (5x^2 + 9x) + y(5x + 6) + (9x^3 + 7)$

In the above polynomial, coefficient of y^3 is another polynomial, $5x^2 + 9x$.

Similarly coefficient of y is $(5x + 6)$ and coefficient of y^0 is $(9x^3 + 7)$.

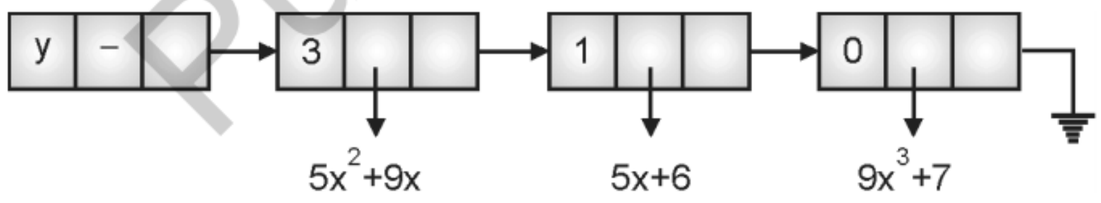
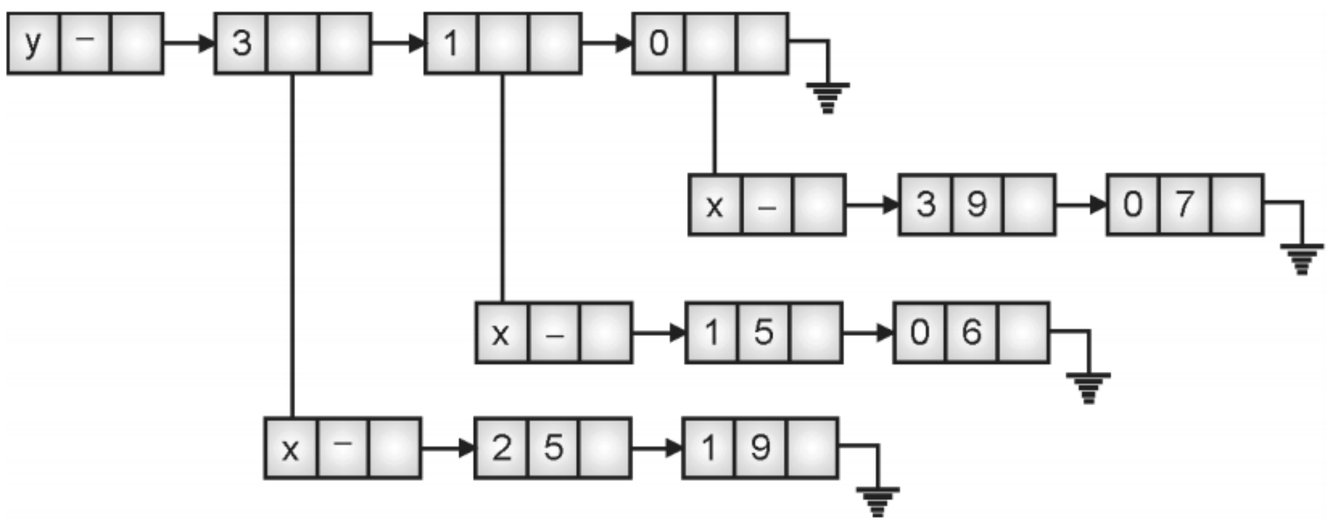


Fig. 6.9.2 : A linked representation of polynomial in two variables. Coefficient field of the node contains address of a polynomial

Further, representing the polynomial in x as a linked list, we get.



TechKnowledge Publications

Structure of the node

```
typedef struct polynode
{
    union
    {
        int power;
        char variable;
    } first_field;
    int flag;
    union
    {
        struct polynode *dnext;
        float coefficient;
    } second_field;
    struct polynode *next
} polynode;
```

Flag = 0, center field contains coefficient

Flag = 1, center field contains address of a polynomial.

Power and variable, share the same memory area and hence they have been declared as a union. Similarly coefficient and dnext share the same memory area and hence they have been declared as union.

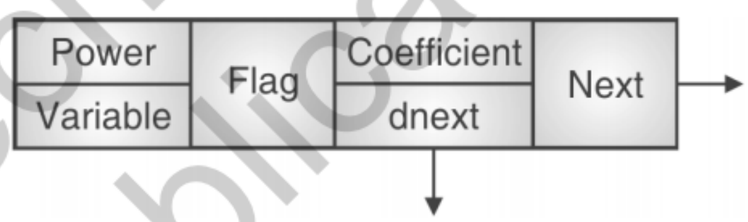


Fig. 6.9.3 : Pictorial view of a node

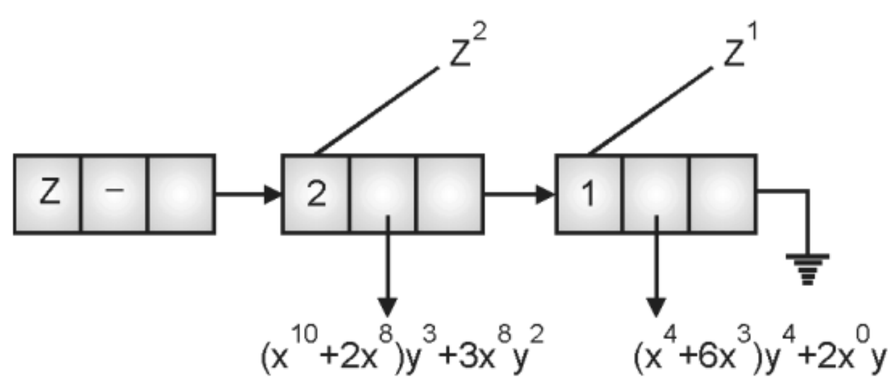
Ex. 6.9.1 : Represent the following polynomial using GLL.

$X^{10}y^3z^2 + 2x^8y^3z^2 + 3x^8y^2z^2 + x^4y^4z + 6x^3y^4z + 2yz$

Soln. : Rewriting the polynomial we get,

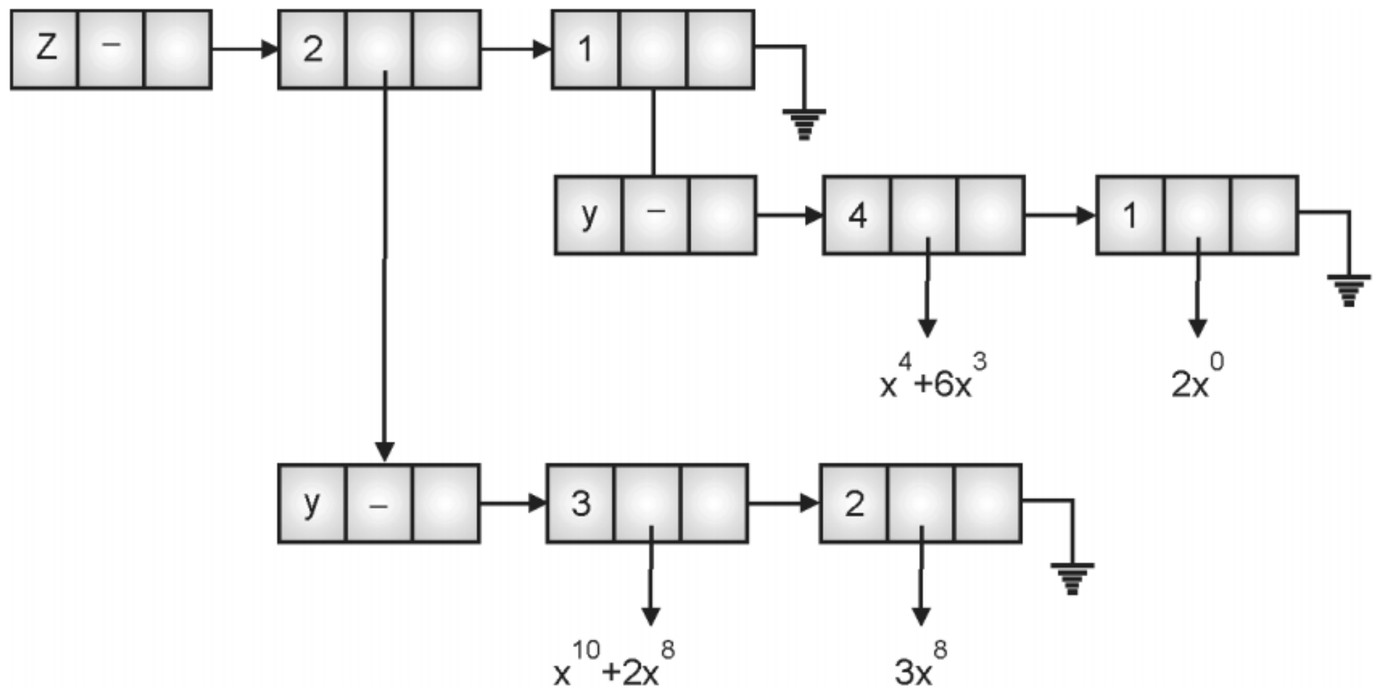
$$((x^{10} + 2x^8)y^3 + 3x^8y^2) z^2 + ((x^4 + 6x^3)y^4 + 2y)z$$

Step 1 :

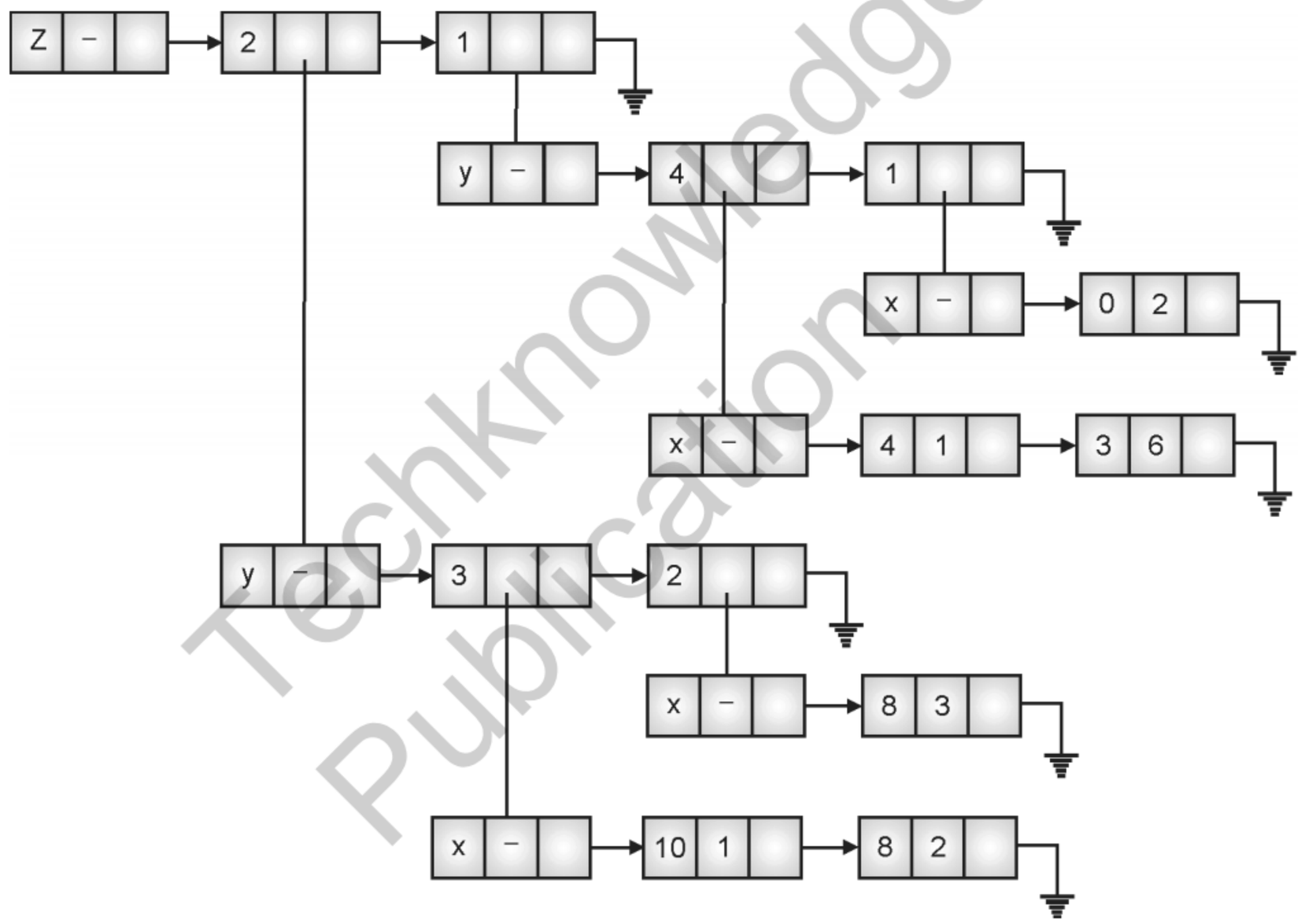


TechKnowledge Publications

Step 2



Step 3



TechKnowledge Publications

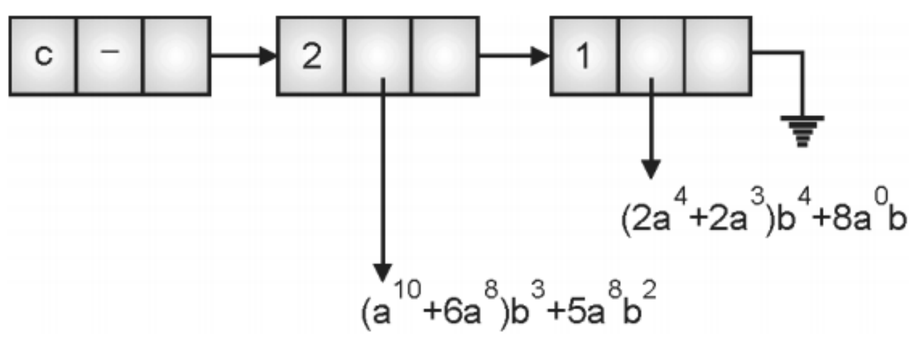
Ex. 6.9.2 : Represent the following polynomial using GLL.

$$P(a,b,c) = a^{10}b^3c^2 + 6a^8b^3c^2 + 5a^8b^2c^2 + 2a^4b^4c + 2a^3b^4c + 8bc$$

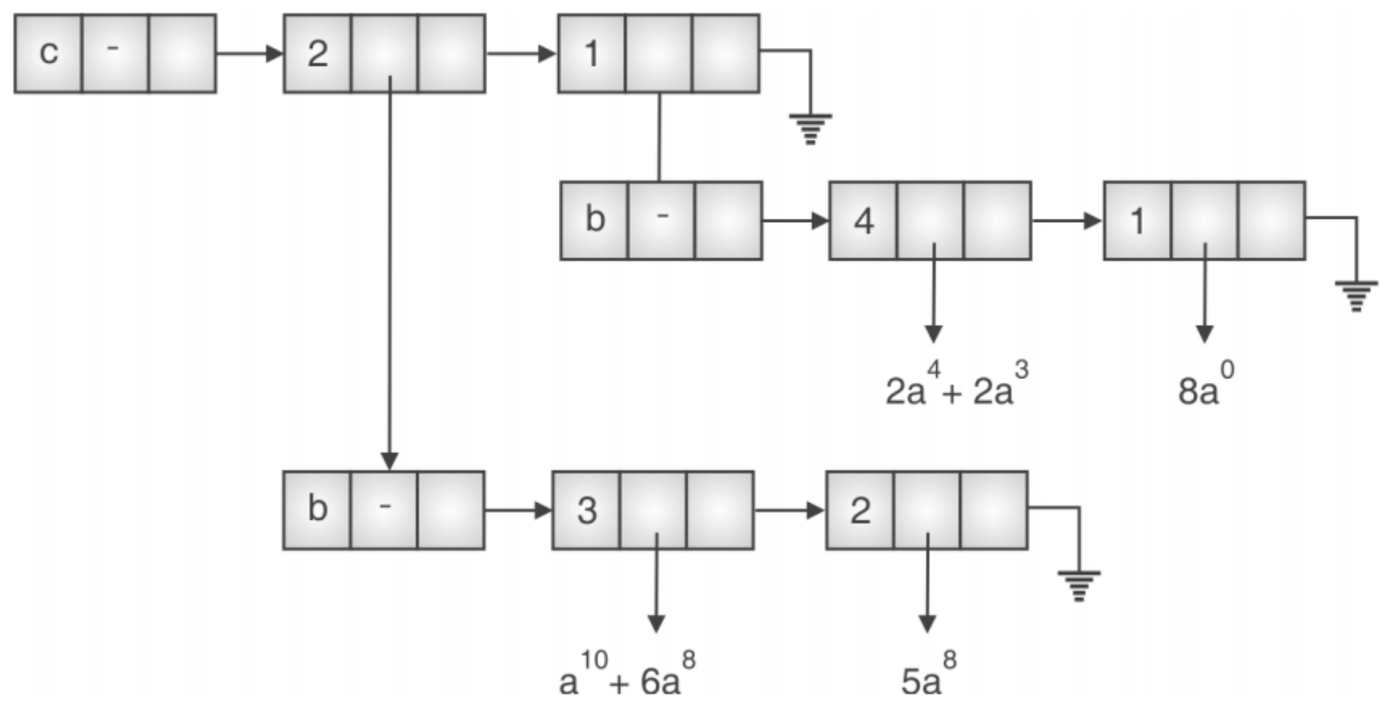
Soln. : Rewriting the polynomial we get :

$$(a^{10}b^3 + 6a^8b^3 + 5a^8b^2)c^2 + (2a^4b^4 + 2a^3b^4 + 8b)c = ((a^{10} + 6a^8)\, b^3 + 5a^8b^2)c^2 + ((2a^4 + 2a^3)\, b^4 + 8a^0b)c$$

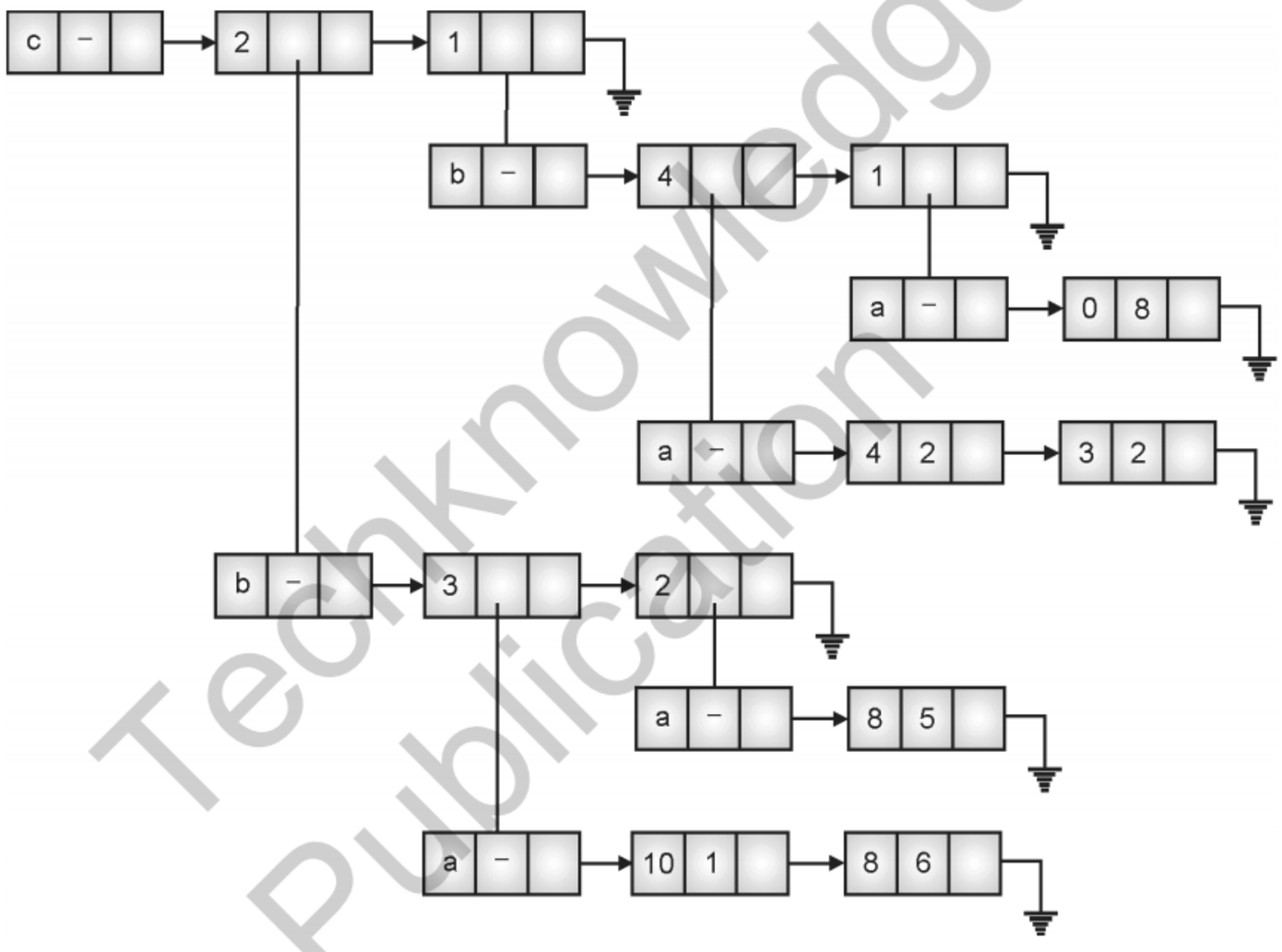
Step 1



Step 2



Step 3



TechKnowledge Publication

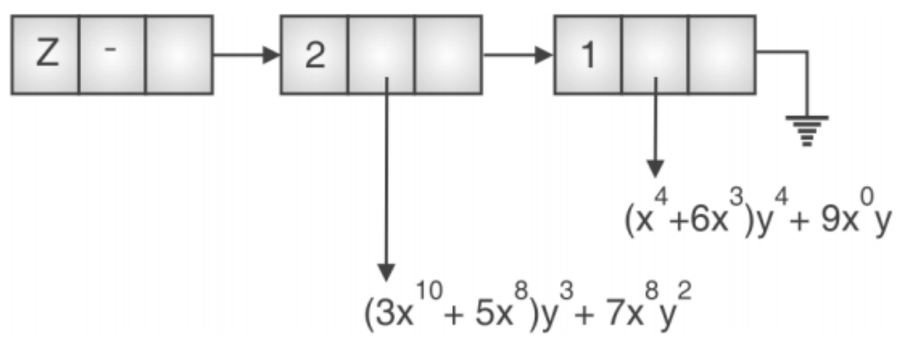
Ex. 6.9.3 : Represent the following polynomial using GLL.

$$3x^{10}y^3z^2 + 5x^8y^3z^2 + 7x^8y^2z^2 + x^4y^4z + 6x^3y^4z + 9yz$$

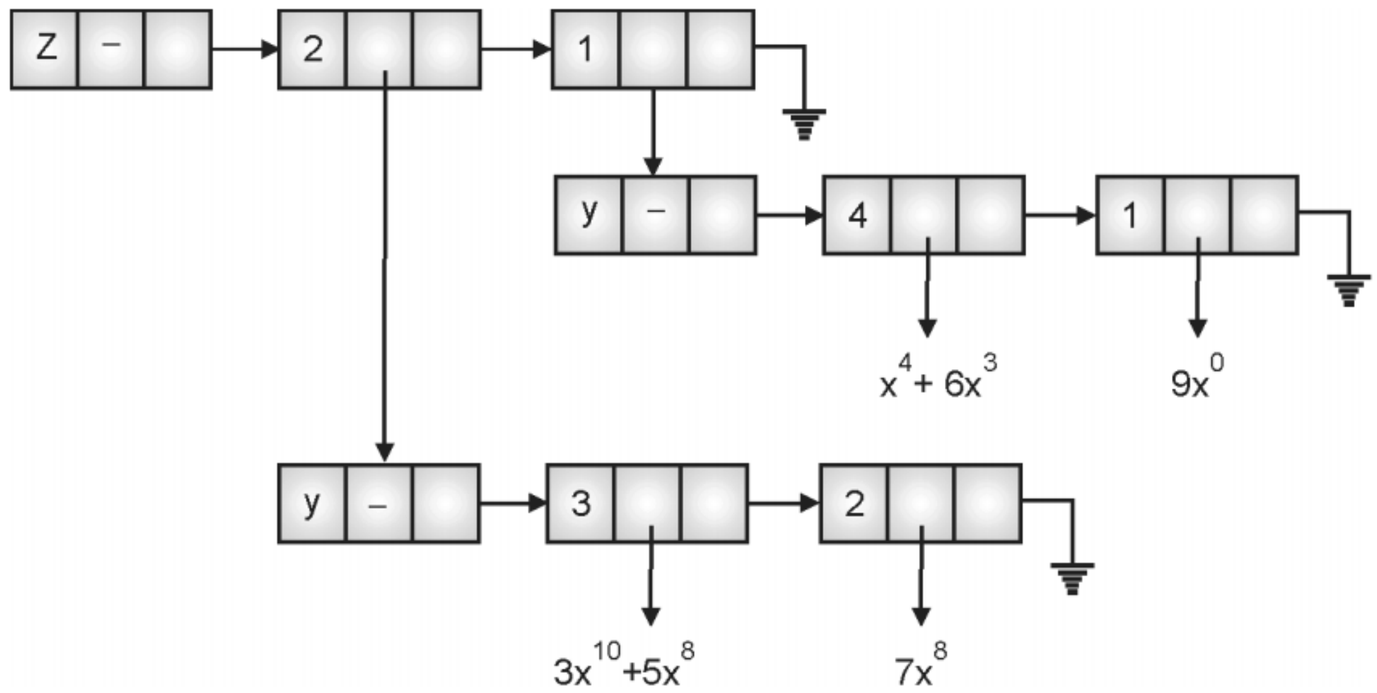
Soln. : Rewriting the polynomial we get :

$$(3x^{10}y^3 + 5x^8y^3 + 7x^8y^2)z^2 + (x^4y^4 + 6x^3y^4 + 9x^0y)z = ((3x^{10} + 5x^8)y^3 + 7x^8y^2)z^2 + ((x^4 + 6x^3)y^4 + 9x^0y)z$$

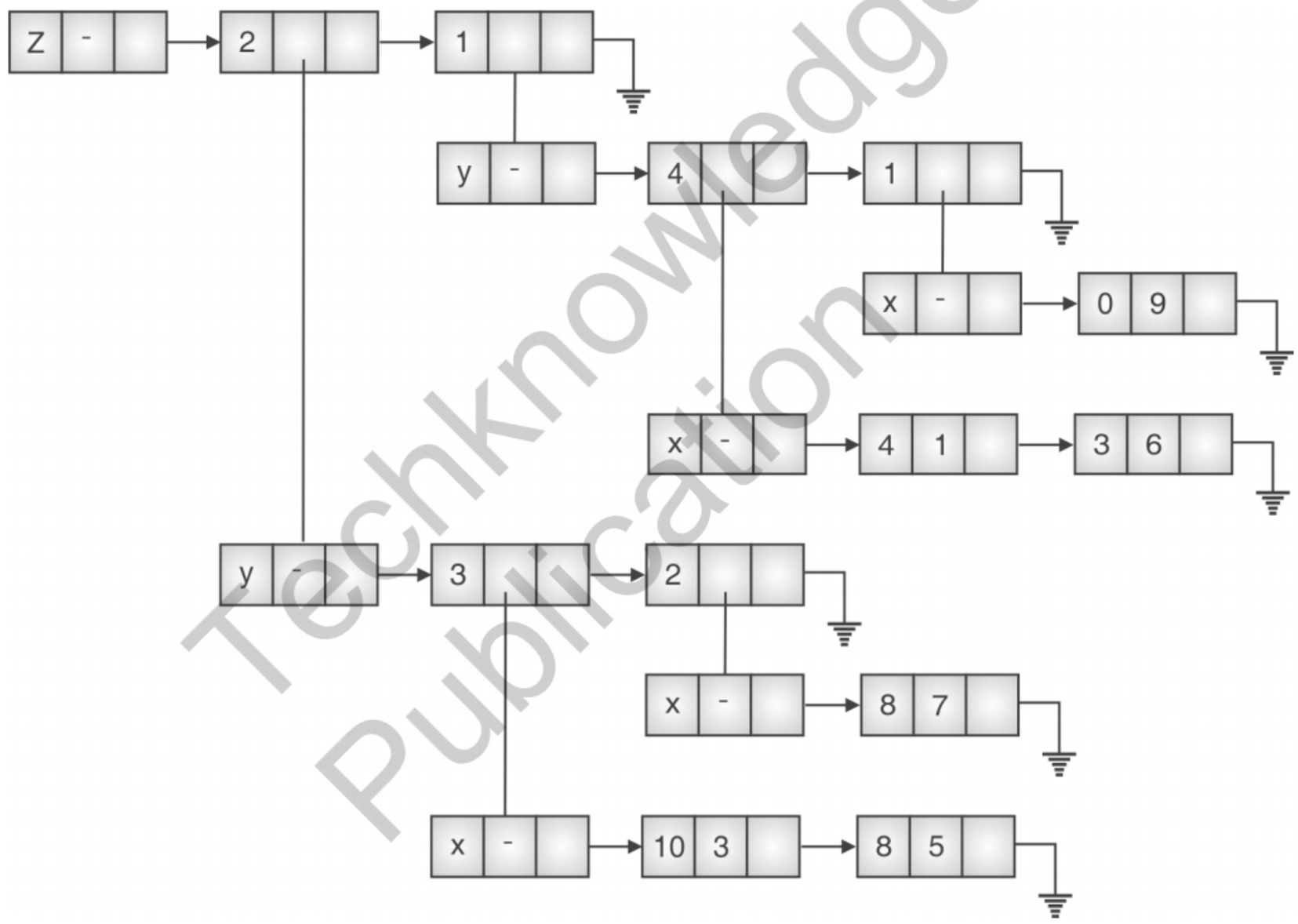
Step 1



Step 2



Step 3

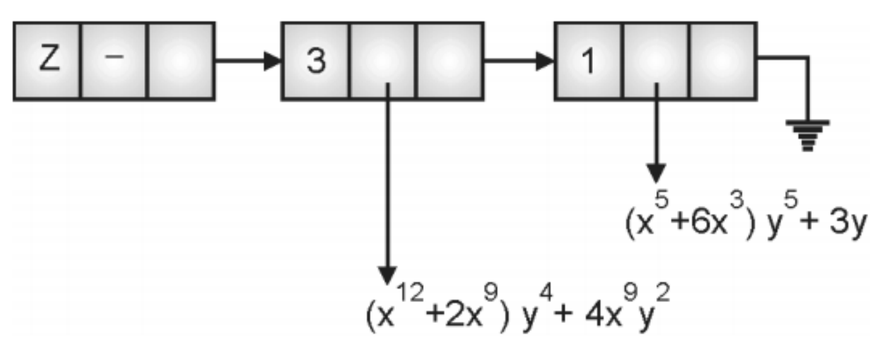


Ex. 6.9.4 : Represent the following polynomial using GLL.

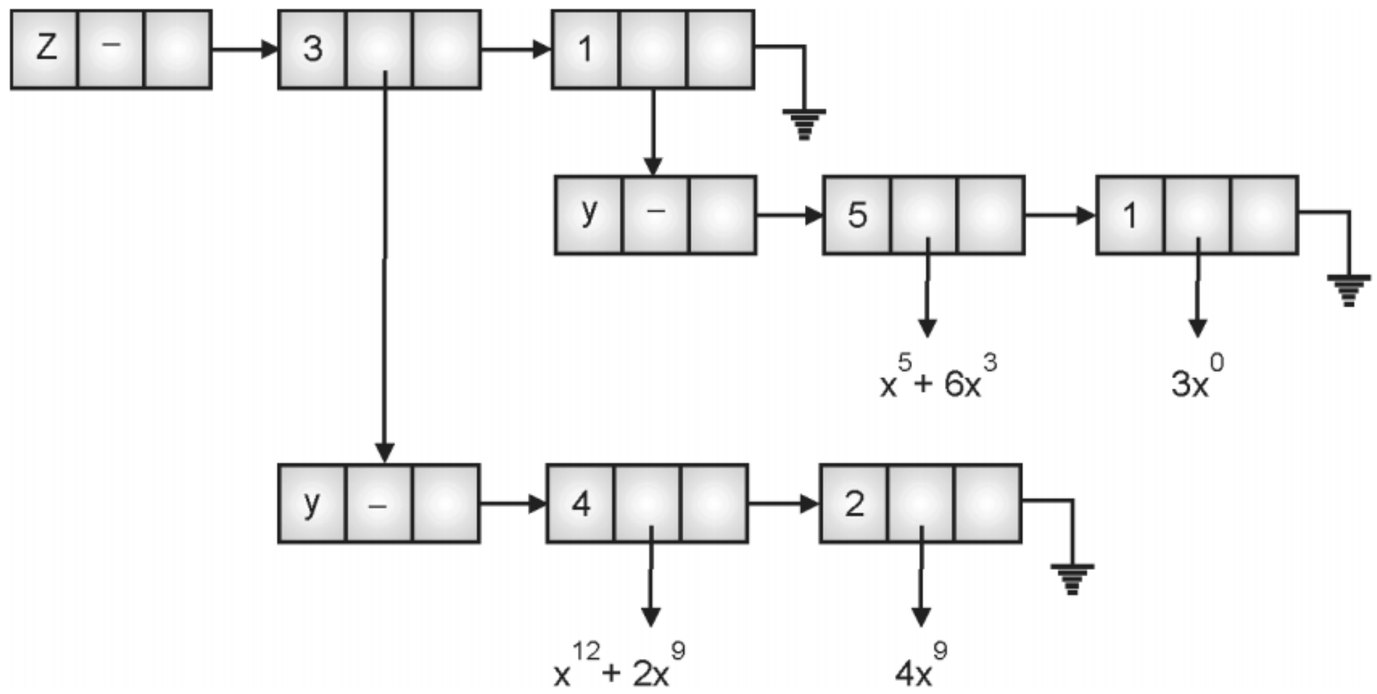
$$((x^{12} + 2x^9)y^4 + 4x^9y^2)z^3 + ((x^5 6x^3)y^5 + 3y)z$$

Soln. :

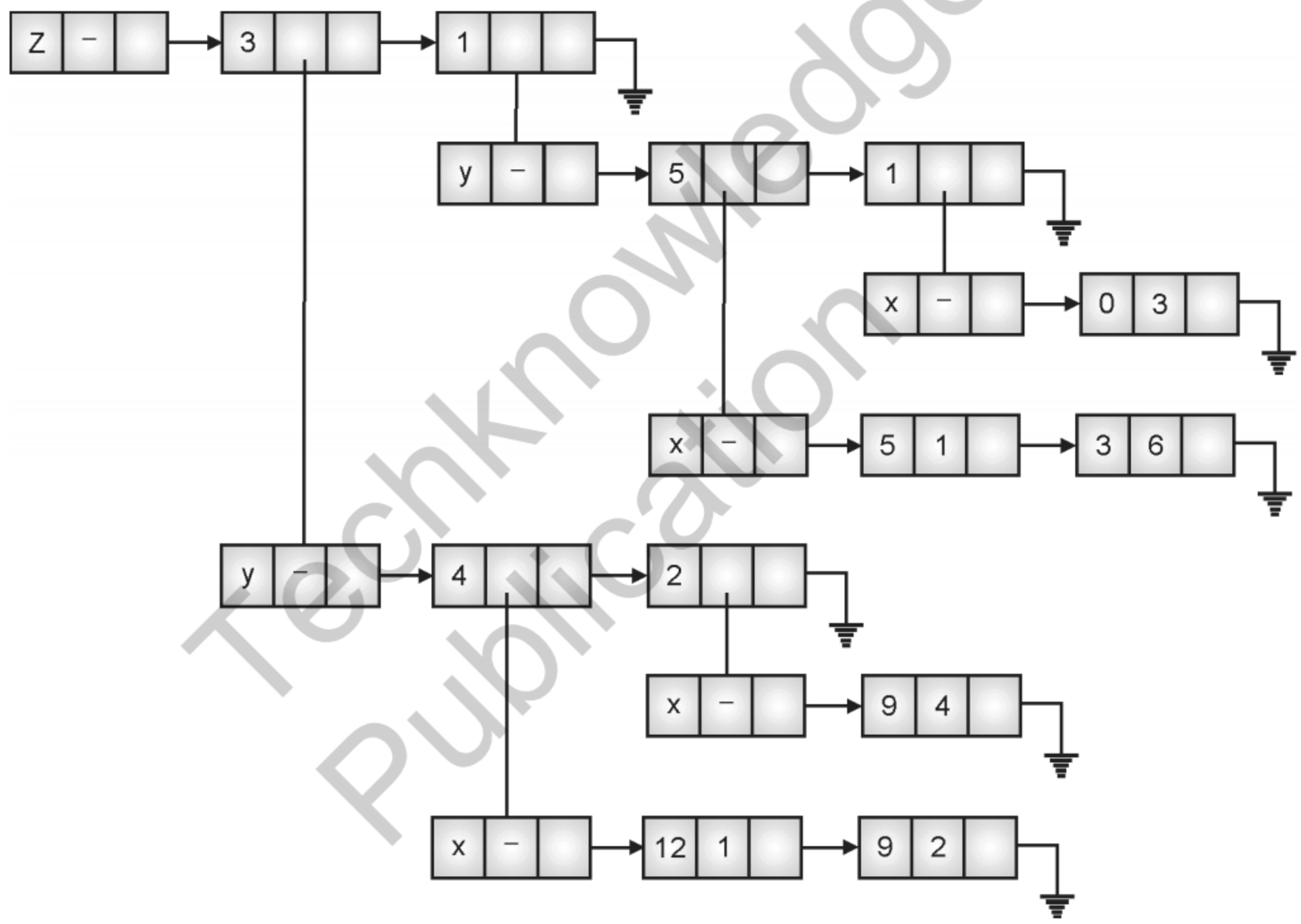
Step 1



Step 2



Step 3



Technknowledge Publications

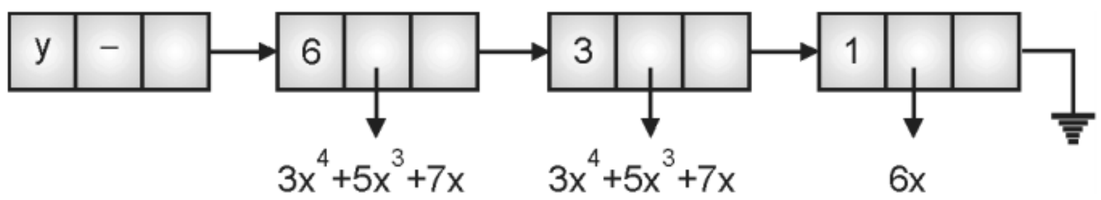
Ex. 6.9.5 : Represent the following polynomial using GLL.

$3x^4y^3 + 5x^3y^3 + 7xy^3 + 3x^4y^6 + 5x^3y^6 + 7xy^6 + 6xy$

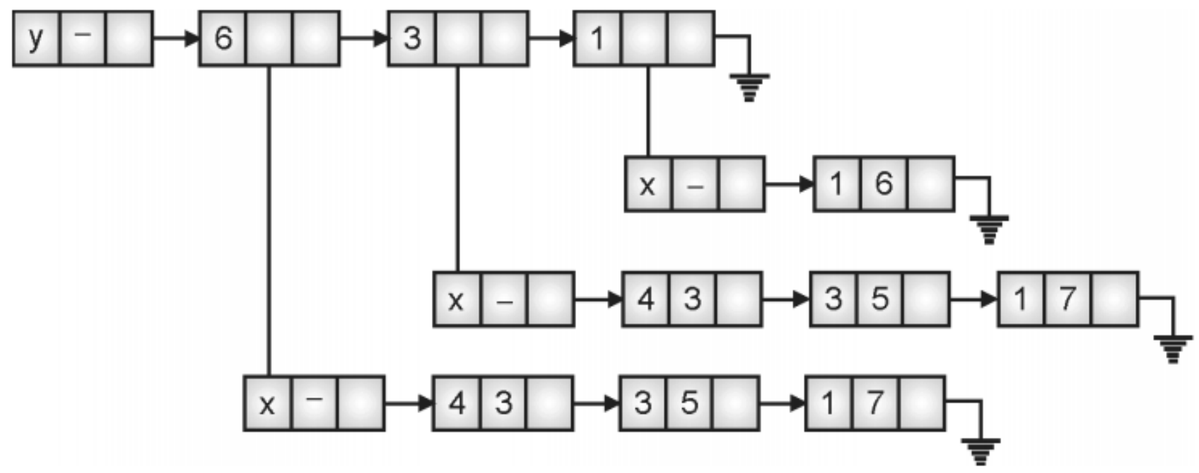
Soln. : Re-writing the polynomial, we get :

$(3x^4 + 5x^3 + 7x)y^6 + (3x^4 + 5x^3 + 7x)y^3 + 6xy$

Step 1 :



Step 2 :



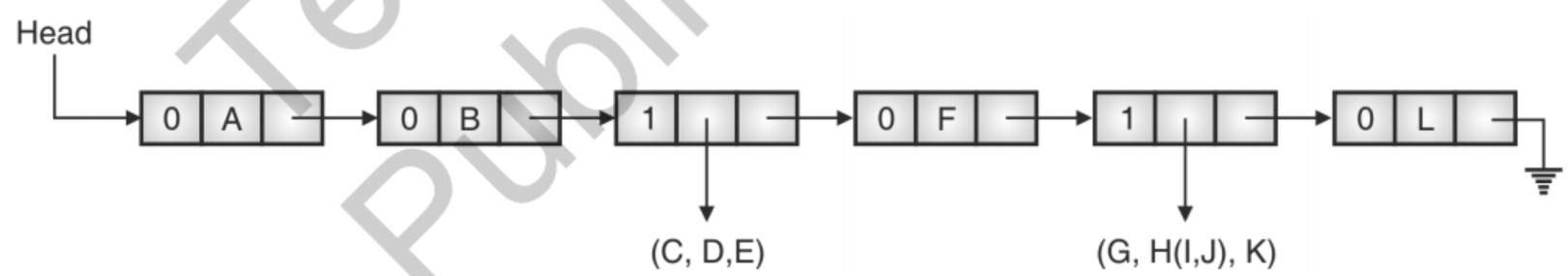
Ex. 6.9.6 : Write node structure and represent following list using generalized linked list.
(A, B, (C, D, E), F, (G, H, (I, J), K), L)

Soln. : Node structure

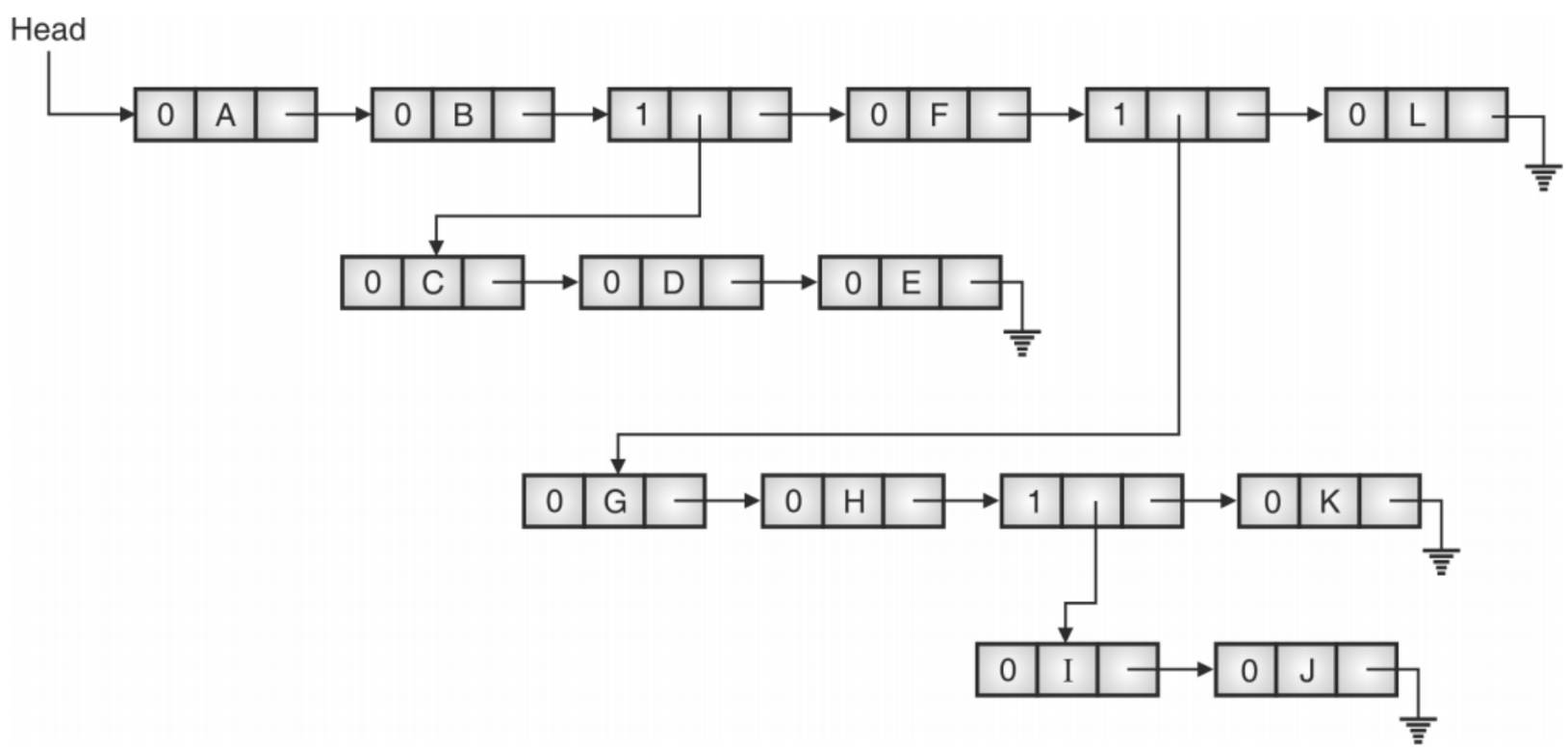
```
typedef struct node
{
    int flag;
    struct node * next;
    union
    {
        char data;
        struct node * down;
    } shared;
} node;
```

Representation of (A, B, (C, D, E), F, (G, H, (I, J), K), L)

Step 1 :



Step 2 :



TechKnowledge Publications

6.10 Linked List in Array

Many languages do not support pointers. If linked lists are required and pointers are not available then linked list can be created in an array.

When linked lists are implemented through pointer then,

(1) Data is stored in a node. Each node contains data and a pointer to the next node.

(2) Dynamic memory is managed through malloc( ) and free( ).

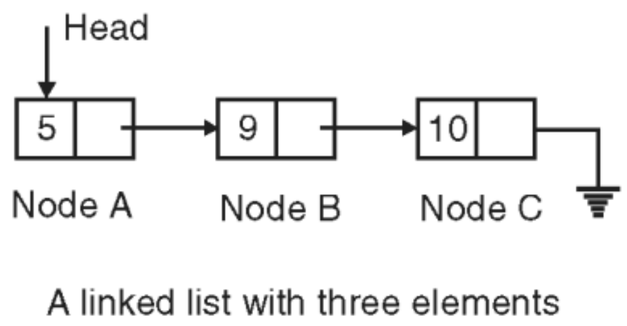


Fig. 6.10.1: A doubly linked circular list

Any implementation of linked list in an array should be able to simulate the above two features.

Condition 1 can be satisfied through an array of nodes. Index can be used in place of an address. It can also be handled through an array of the type "int nodes[2][MAX]".

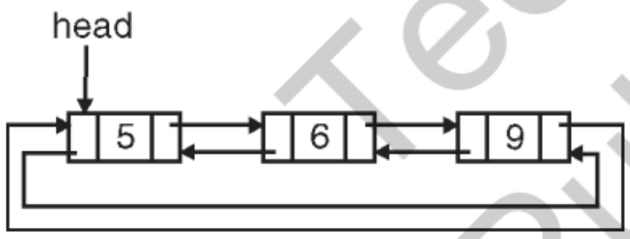


Fig. 6.10.2 : A linked list with three elements

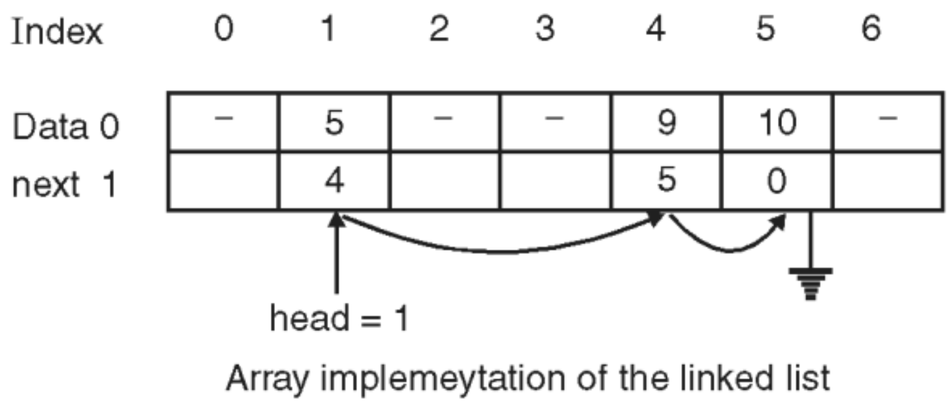


Fig. 6.10.3 : Array implementation of the linked list of Fig. 6.10.2

- Fig. 6.10.3 gives an implementation of the linked list of Fig. 6.10.2
- Position 1, 4 and 5 in the array are selected arbitrarily to store modes A, B and C respectively of the linked list.
- Next field of the node A in array contain 4, which is the index of node B in array.
- Next field of the node B in array contains 5, which is the index of node C in array.

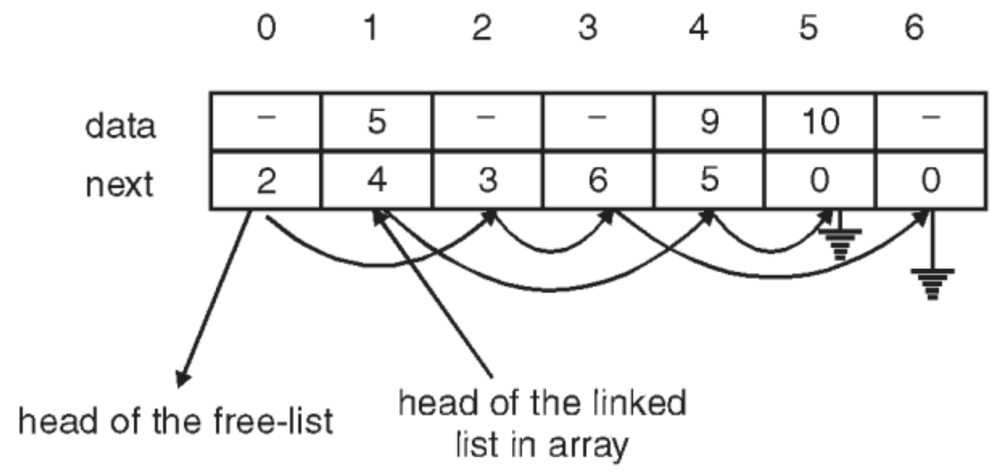


Fig. 6.10.4 : A free list of cells 2, 3 and 6 a linked list with cells 1, 4 and 5

- Next field of the node C in array contains 0, indicating it is the last node.
- Condition 2 can be implemented by maintaining a list of free cells in the array. These cells can be connected to each other with the index of the first free cell in cell 0.

6.11 Memory Allocation

One of the following strategies can be used for memory allocation.

1. First-fit 2. Best-fit
3. Worst-fit 4. Next-fit

6.11.1 First-Fit Allocation

In the first-fit method, the free list is traversed sequentially to find the first free block whose size is larger than or equal to the amount requested. Once the block is found,

- It is removed from the free list and free list is suitably manipulated. Every block is of fixed size.
- The block is split into two portions. The first block remains in the free list and the second is allocated. Block size is not fixed.

TechKnowledge Publications

Ex. 6.11.1 : On a system using first-fit allocation, status of the memory is as follows after initial allocation.

U = used	U	A	U	A	U	A	U	A
A = Available	20k	10k	30k	30k	20k	10k	40k	20k

At what starting address will each of the additional request 20k and 10k be allocated.

Soln. :

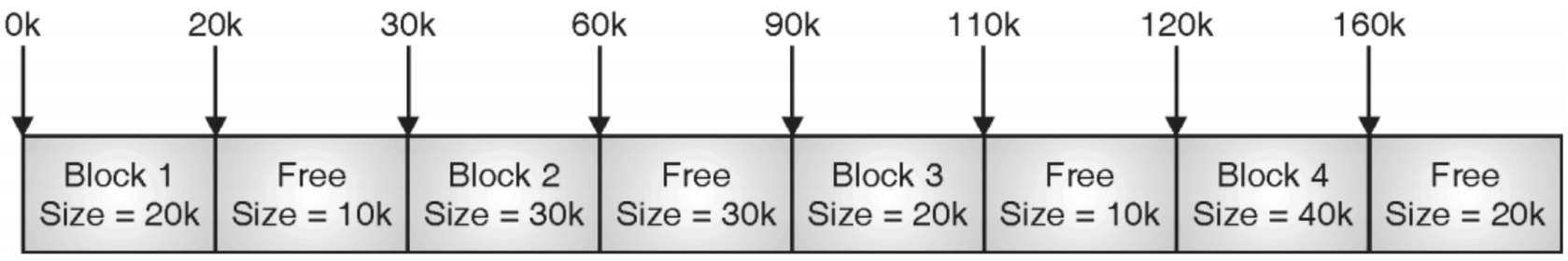


Fig. Ex. 6.11.1

A request of 20k will locate the first free block of size ≥ 20k starting at the address 60k. This block will be split into two part, second part will be allocated. Block will be allocated at the address 70k.

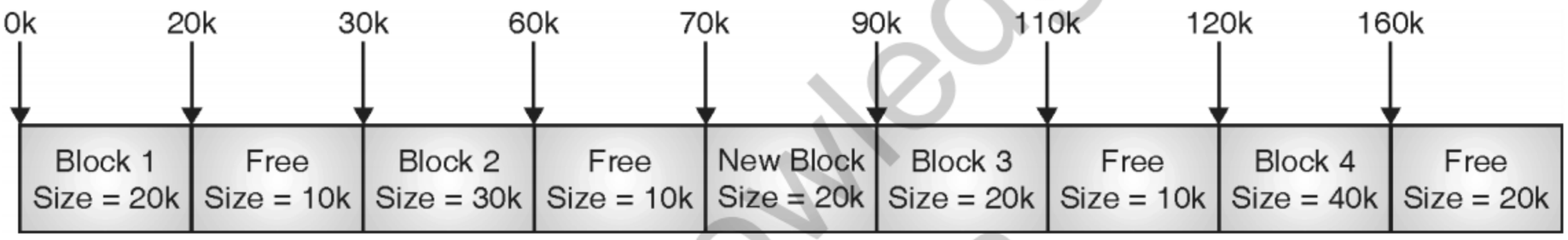


Fig. Ex. 6.11.1(a)

An addition request for 10k will locate the first free block of size ≥ 10k starting at address 20k. The entire block will be allocated.

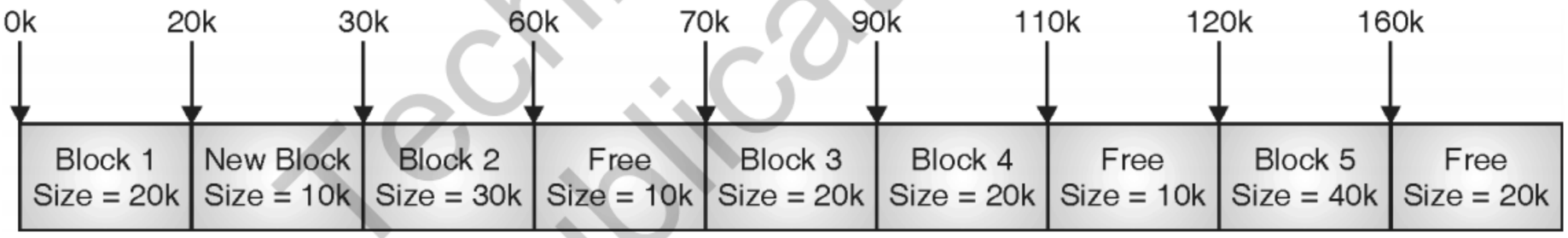


Fig. Ex. 6.11.1(b)

6.11.1(A) Algorithm for First-Fit

```
P = RL(head);
/* P is being used to traverse the free list */
    do
    {
        if(SIZE(P) ≥ n)
        /*  n is the size of the requested block. A block of required size is found */
        {
           diff = SIZE(P) – n;
            if(diff < ∈)  /* allocated the whole block */
             {
                  RL(LL(P)) = RL(P);
                  TAG(P) = TAG(P + SIZE(P) – 1) = 1
```

TechKnowledge Publications

```
            /* set tags */
            return(P);
        }
        else
        {
            SIZE(P) = diff;
            UL(P + diff – 1) = P;
            TAG(P + diff – 1) = 0;
            /* upper portion is unused */
            P = P + diff;
            /* set at the start of the allocated block */
            SIZE(P) = n;
            TAG(P) = TAG(P + n – 1) = 1;
            /* set tags for allocated blocks */
            return(P);
        }
    }
    P = RL(P);
} while(P! = RL(head));
return(NULL);
```

6.11.2 Best-Fit Allocation

In the best-fit method, we obtain the smallest free block whose size is greater than or equal to the requested size.

Ex. 6.11.2 : On a system using best-fit allocation, status of the memory is as follows after initial allocation.

U = used	U	A	U	A	U	A	U	A
A = Available	20k	10k	30k	30k	20k	10k	40k	20k

At what starting address will each of the addition request 20k and 10k will be allocated.

Soln. :

A request of 20k will located the smallest free block whose size is greater than or equal to 20k at the address 160k. It is shown in Fig. Ex. 6.11.2.

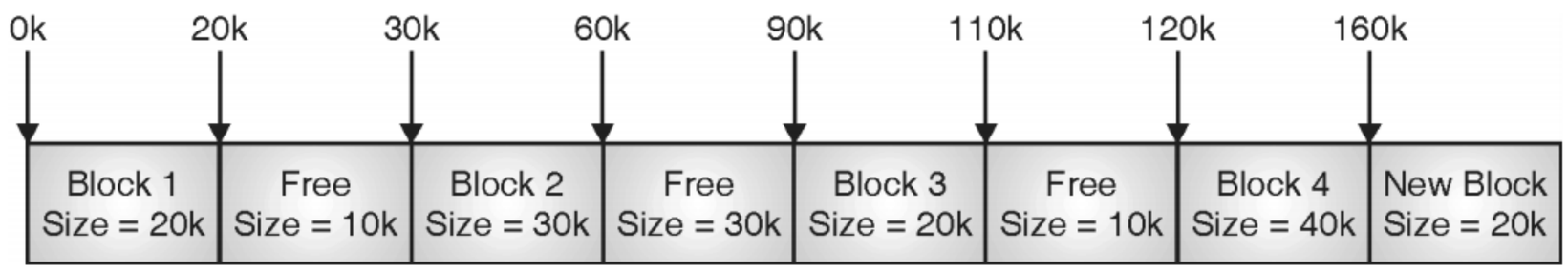


Fig. Ex. 6.11.2

An additional request for 10k will locate the smallest free block whose size is greater than or equal to 10k at the address 20k. It is show in Fig. Ex. 6.11.2(a).

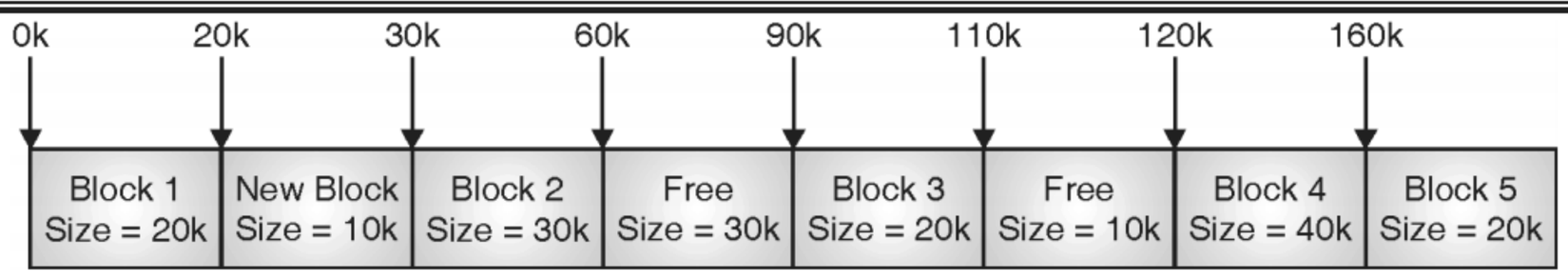


Fig. Ex. 6.11.2(a)

6.11.2(A) Algorithm for Best-Fit

```
q = RL(head);
/*q is being used to traverse the free list */
P = NULL;  /* P will point to the Best-fit block */
rsize = memory_size + 1;
 /*rsize will be size of the block pointed by P */
do
{   if(SIZE(q) > = n && SIZE(q) < rsize)
   /* n is the size of the requested block */
     {
          P = q;
          rsize = SIZE(q);
     }
     q = RL(q);
}
while(q! = RL(head)); /* traverse the entire list */
if(P == NULL)
     return(P);
else
{   diff = SIZE(P) – n;
     if(diff < ∈)
     {    /* allocate the whole block */
          RL(LL(P)) = LL(P);
            TAG(P) = TAG(P+SIZE(P)–1) = 1;
          /* set tags */
            return(P);
     }
     else
     {    SIZE(P) = diff;
          UL(P + diff – 1) = P;
          /* set the uplink */
          TAG(P + diff –1) = 0;
          /* upper portion is unused */
          P = P + diff;
          /*set at the start of the allocated block */
          SIZE(P) = n;
          TAG(P) = TAG(P + n – 1) = 1;
          /* set tags for allocated blocks */
          return(p);
     }
}
```

6.11.3 Worst Fit

This method is opposite of best-fit method. We obtain the largest free block whose size is greater than or equal to the requested size.

6.11.4 Next Fit

In case of next fit searching in the free list starts from the next of the previously allocated node. It reduces the search time.

6.12 Garbage Collection

Garbage collection is opposite of memory allocation. Garbage is memory occupied by objects that are no longer in use by the program. A garbage collector collects garbage and returns it to the general pool of free memory. The basic principles of garbage collection are :

1. Find data objects in a program that is not to be accessed in the future.
2. Reclaim the resources used by these objects.

Review Questions

Q. 1 Give node structure to represent a list names using DLL and write C functions for the following :

(i) Display list forward

(ii) Display list reverse

(iii) Display names starting with letter S or s. **(8 Marks)**

Tech Knowledge Publications

Q. 2 Write a pseudo C/C++ code for addition of two polynomials and comment on time complexity of them assuming n terms in first polynomial and m terms in other. **(10 Marks)**

Q. 3 Compare array and link list. **(6 Marks)**

Q. 4 Write pseudo C code to delete a node from doubly link list. **(6 Marks)**

Q. 5 Write pseudo C code to add a node in singly circular link list with integer data. **(6 Marks)**

Q. 6 Explain the concept of polynomial representation using linked list with suitable example. **(6 Marks)**

Q. 7 Write a C function to reverse a linear singly linked list by changing link pointers. **(6 Marks)**

Q. 8 Write advantages of linked memory organization. **(3 Marks)**

Q. 9 What is doubly linked list? Write C code to

(i) delete a node pointed by pointer temp in a circular DLL.

(ii) insert a new node pointed by pointer newp after a node pointed by pointer temp in circular DLL. **(5 Marks)**

Q. 10 Explain following :

(i) Garbage collection

(ii) Memory allocation **(6 Marks)**

Q. 11 Write a C/C++ program to create singly inked list of integers and display it forward. **(6 Marks)**

Q. 12 What is generalized list ? Give node structure to represent multivariable polynomial using GLL. **(4 Marks)**

Q. 13 Write a C code for reversing the singly linked list without using additional data structure. **(6 Marks)**

Q. 14 Write a pseudo code to merge two sorted linked lists into the third. **(7 Marks)**

Q. 15 What are the advantages of linked list over array? Describe different types of linked list. **(7 Marks)**

Q. 16 Write a C function to perform the following operation on SLL : **(6 Marks)**

(i) Insert element at any position

Q. 17 Write C function to delete any node in case of DLL. **(6 Marks)**

Q. 18 Write a pseudo code to delete a node from singly linked list. **(7 Marks)**

Q. 19 Write a pseudo code to insert a node at start and at end in DLL. **(7 Marks)**

Technknowledge Publication

❑❑❑

CHAPTER 7

Unit V

Stacks

Syllabus :

Basic concept, stack Abstract Data Type, Representation of Stacks Using Sequential Organization, stack operations, Multiple Stacks, Applications of Stack- Expression Evaluation and Conversion, Polish notation and expression conversion, Need for prefix and postfix expressions, Postfix expression evaluation, Linked Stack and Operations. Recursion- concept, variants of recursion- direct, indirect, tail and tree, Backtracking algorithmic strategy, use of stack in backtracking.

Exemplar/Case Studies : Android- multiple tasks/multiple activities and back-stack, Tower of Hanoi, n Queens problem.

Tech Knowledge Publication

7.1 Introduction

Q. Write pseudo C/C++ code to implement stack as an ADT. **SPPU - Dec. 16, 6 Marks**

Q. What is stack ? Write an ADT for stack. **SPPU - Dec. 17, 6 Marks**

Q. Define Stack. **SPPU - May 19, 1 Mark**

Q. Write pseudo C/C++ code to represent a stack as an ADT. **SPPU - Dec. 19, 4 Marks**

– Stack is a LIFO (Last In First Out) structure. It is an ordered list of the same type of elements. A stack is a linear list where all insertions and deletions are permitted only at one end of the list. When elements are added to stack it grows at one end. Similarly, when elements are deleted from a stack, it shrinks at the same end.

– Fig. 7.1.1 shows expansion and shrinking of a stack. Initially stack is empty.

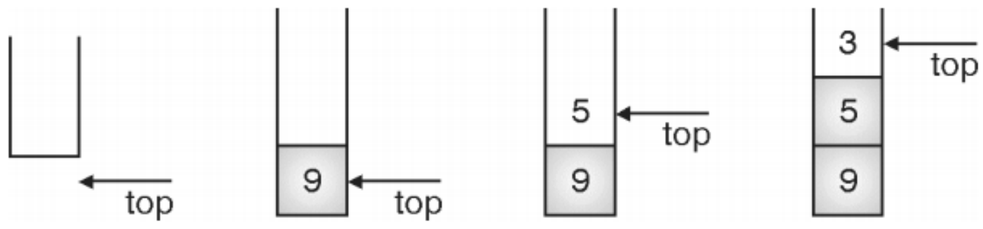


Fig. 7.1.1 : Insertion of 9,5,3 in a stack

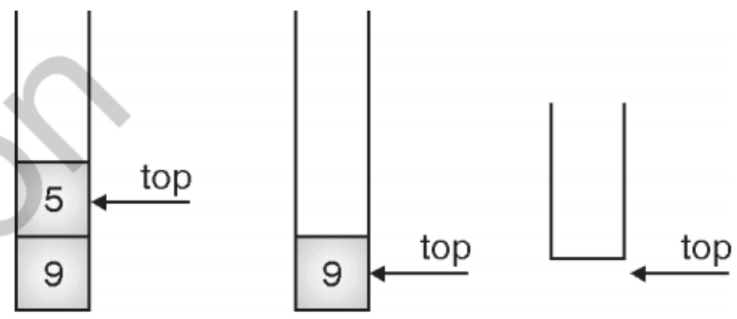


Fig. 7.1.2 : Deletion of 2 elements from the stack

– A variable top, points to the top element of the list.

Stack as an ADT :

Stack is a LIFO structure. Stack can be represented using an array. A one-dimensional array can be used to hold elements of a stack. Another variable "top" is used to keep track of the index of the top most element.

Formally, a stack may be defined as follows :

```
typedef struct stack
{
    int data[SIZE];
    int top;
} stack;
/* SIZE is a constant, maximum number of elements
that can be stored */
```

Associated with the object stack there are several operations that are necessary :

(1) void initialize(stack *p) : It initializes a stack as an empty stack. Initial value of top is set to –1.

(2) int empty(stack *p) : Function checks whether the stack is empty. It returns 1 or 0 depending on whether the stack is empty or not.

(3) int full(stack *p) : The function checks whether the stack is full. A stack is said to be full when the associated array is completely filled up. Whenever the stack is full, top points to the last element (i.e. data[SIZE – 1]) of the array. The function returns 1 or 0 depending on whether the stack is full or not.

(4) int pop(stack *p) : The function deletes topmost element from the stack and also returns it to the calling program. Deletion from an empty stack will cause underflow.

(5) void push(stack *p, int x) : The function inserts the element x onto the stack pointed by P. Insertion will cause an overflow if the stack is full.

7.2 Operations on Stacks

1. **Initialize( ) :** Make a stack empty
2. **Empty( ) :** To determine if a stack is empty or not
3. **Full( ) :** To determine if a stack is full or not
4. **Push( ) :** if a stack is not full then push a new element at the top of the stack (similar to insert in a list)
5. **Pop( ) :** If a stack is not empty, then pop the element from its top(similar to delete( ) from a list)

7.3 Array Representation

A one-dimensional array can be used to hold elements of a stack. Another variable "top" is used to keep track of the index of the top most element.

Formally, a stack may be defined as follows:

```
typedef struct stack
{   int data[MAX] ;
    int top ;
}stack ; /*MAX is a constant, maximum number of elements that can be stored */
```

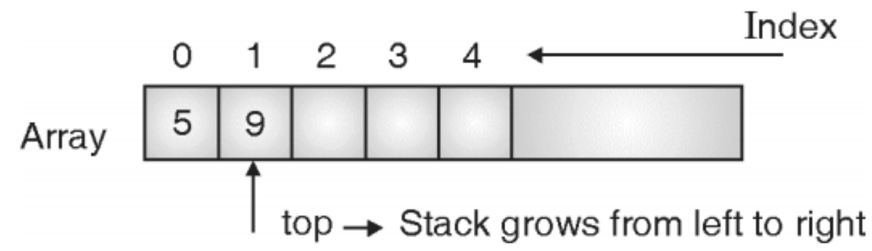


Fig. 7.3.1

- Initially, top is set to –1
- A stack with top as –1 is an empty stack.
- When the value of top becomes MAX – 1 after a series of insertions, it is full.
- After "push" operation top = top + 1 (Stack Growing)
- After "pop" operation top = top – 1 (Stack shrinking)

7.3.1 'C' Functions for Primitive Operations on a Stack

```
{
    s → top = –1 ;
}
```

Initially, stack is empty.

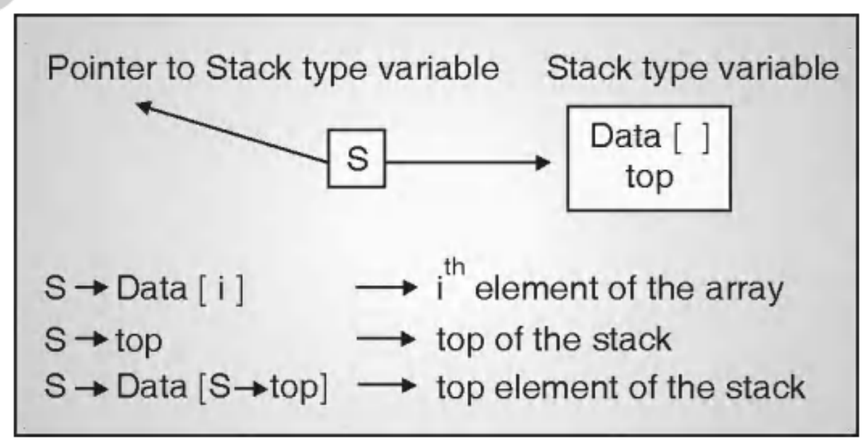


Fig. 7.3.2

Accessing elements of a stack using a pointer.

```
int empty(stack * s)
{   if(s → top == – 1)
        return(1) ;    // stack is empty
    return(0) ;        // stack empty condition is false
}
```

Above function checks whether, a stack is empty [containing 0 elements]. Function returns a value 1(true) if the stack is empty. Function returns a value 0(false) if the stack is not empty (there are some elements).

Tech Knowledge Publications

```
int full(stack *s)
{   if( S→top == MAX –1)
        return(1);
    return(0);
}
```

A stack is full, if no more elements can be added.

Above function checks whether a stack is full (no more elements can be added). Function returns a value 1 (true) if the stack is full. Function returns a value 0 (false) if the stack is not full (more elements could be added).

Fig. 7.3.3

```
void push(stack *s, int x)
{
    s → top = s → top + 1 ;
    s → data[s → top] = x;
}
```

Note : main program should ensure that the stack is not full before making a call to push( ) function. It may cause an overflow, otherwise.

```
int pop(stack *s)
{
    int x ;
    x = s → data[s → top] ;
    s → top = s →top – 1 ;
    return(x);
}
```

Note : main program should ensure that the stack is not empty before making a call to pop( ) function. It may cause an underflow, otherwise.

The function push( ), increases the value of the top by 1 and then stores the element at the top location in "data" array.

The function pop( ) returns the top element from the stack. It also, reduces the value of top by 1 (one element is deleted from the stack).

7.3.2 Program Showing Stack Operations

Convert a decimal number to binary using a stack

Example : Input No. = 21 (Decimal)

21%2 = 1	1
10%2 = 0	0 1
5%2 = 1	1 0 1
2%2 = 0	0 1 0 1
1%2 = 1	1 0 1 0 1

$(21)_{10} = (1\ 0\ 1\ 0\ 1)_2$

A decimal number can be converted to binary form through repeated division of the number by 2. Remainder at each step is saved in a stack. Finally, the bits stored in stack can be printed one by one through POP( ) operation.

Program 7.3.1 : Program for conversion of decimal number to binary form.

```
#include<stdio.h>
#include<conio.h>
#define MAX 20
typedef struct stack
{
    int data[MAX];
    int top;
}stack;
void init(stack *);
int empty(stack *);
```

Techknowledge Publication

```
int full(stack *);
int pop(stack *);
void push(stack *, int);
void main()
{
    stack s;
    int x;
    init(&s);
    printf("\n Enter decimal number:");
    scanf("%d", &x);
    while((x! = 0))
    {
        if(!full(&s))
        {
            push(&s, x%2);
            x = x/2;
        }
        else
        {
            printf("\n stack overflow");
            exit(0);
        }
    }
    printf("\n ");
    while(!empty(&s))
    {
        x = pop(&s);
        printf("%d", x);
    }
}
void init(stack *s)
{
    s->top = – 1;
}
int empty(stack *s)
{
    if(s->top == –1)
        return(1);
    return(0);
}
```

```
int full(stack *s)
{
    if(s->top == MAX – 1)
        return(1);
    return(0);
}
void push(stack *s, int x)
{
    s->top = s->top + 1;
    s->data[s->top] = x;
}
int pop(stack *s)
{
    int x;
    x = s->data[s->top];
    s->top = s->top – 1;
    return(x);
}
```

Output

```
Enter decimal number:12
1100
```

Program 7.3.2 : Program for reversing a string using a stack

```
# define MAX 30
typedef struct stack
{
    char data [MAX];
    int top;
} stack;
void init(stack *p)
{
    p → top = –1;
}
int empty(stack *p)
{
    if(p → top  == –1)
    return(1);
    return(0);
}
int full(stack *p)
{
```

Tech Knowledge Publications

```
    if(p → top == MAX – 1)
        return(1);
    return(0);
}
void push(stack *p, char x)
{
    if(! Full(p))
    {
        p → top = p → top + 1;
        p → data [p → top] = x;
    }
    else
        printf("\n stack is full");
}
char pop(stack *p)
{
    char x;
    if(! empty(p) )
    {
        x = p → data [p → top];
        p → top = p → top – 1;
        return(x);
    }
    printf("\n stack is empty:");
    return('\0');
}
void main()
{
    stack s;
    char x;
    init(&s);
    printf("\n enter a string:");
    while((x = getchar( )) ! = '\n ')
    if(! full(&s))
        push(&s, x);
    else
    {
        printf("\n stack is full…");
        exit(0);
    }
    printf("\n result = ");
    while(! empty(&s))
    {
        x = pop(&s);
        printf("%c", x);
    }
}
```

Output

```
enter a string : India
result = aidnI
```

7.3.3 Operations on Stack Considering Overflow and Underflow : [Array Implementation]

'C' functions for operations on stack

```
void push()
{
    if(p->top == N –1)
        printf("\n overflow !! cannot be inserted");
    else
    {
        p->top = p->top + 1;
        p->data[p->top] = x;
    }
}
int pop(stack *p)
{
    int x;
    if(p->top == –1)
        printf("\n Underflow !!! cannot be deleted");
    else
    {
        x = p->data[p->top];
        p->top = p->top –1;
    }
    return(x);
}
void print(stack *p)
{
    int i;
    for(i = p->top; i> = 0; i– –)
        printf("\n %d", p->data[i]);
}
```

Techknowledge Publication

7.4 Linked Representation of a Stack

Stack can be representation efficiently through Linked lists. When a stack is represented using a Linked list, it is never full as long as system has memory for dynamic allocation of space for a node. A stack can be represented through a singly connected Linked list.

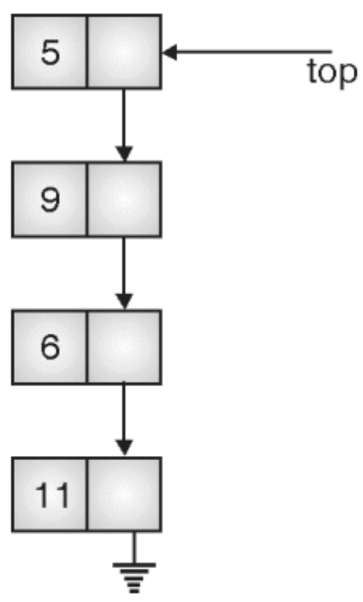


Fig. 7.4.1 : Linked list representation of a stack

Structure of a node

```
typedef struct node
{   int data ;
    struct node *next ;
} node ;
```

– Top is a pointer type variable, pointing to the top node

Declaration of top

```
node *top ;
```

– Memory for a new node can be allocated through malloc( )

```
(node *) malloc(sizeof(node))
```

sizeof(node) gives number of bytes required to store a node.

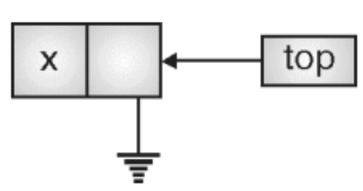


Fig. 7.4.2

Function malloc( ) returns the address of the memory block allocated to store a node. Address returned by malloc( ) is type casted through (node*) so that the address can be treated as an address of a node.

– Initially top is NULL (Empty Stack) ;

– Inserting an element x in an empty stack

```
top = (node*) malloc(sizeof(node)) ;
top → data = x ;
top  → next = NULL ;
```

– Subsequent elements can be inserted in the stack through the following operations.

```
node * P ;
// A pointer for acquiring memory for new node
P = (node * ) malloc(sizeof(node)) ;
```

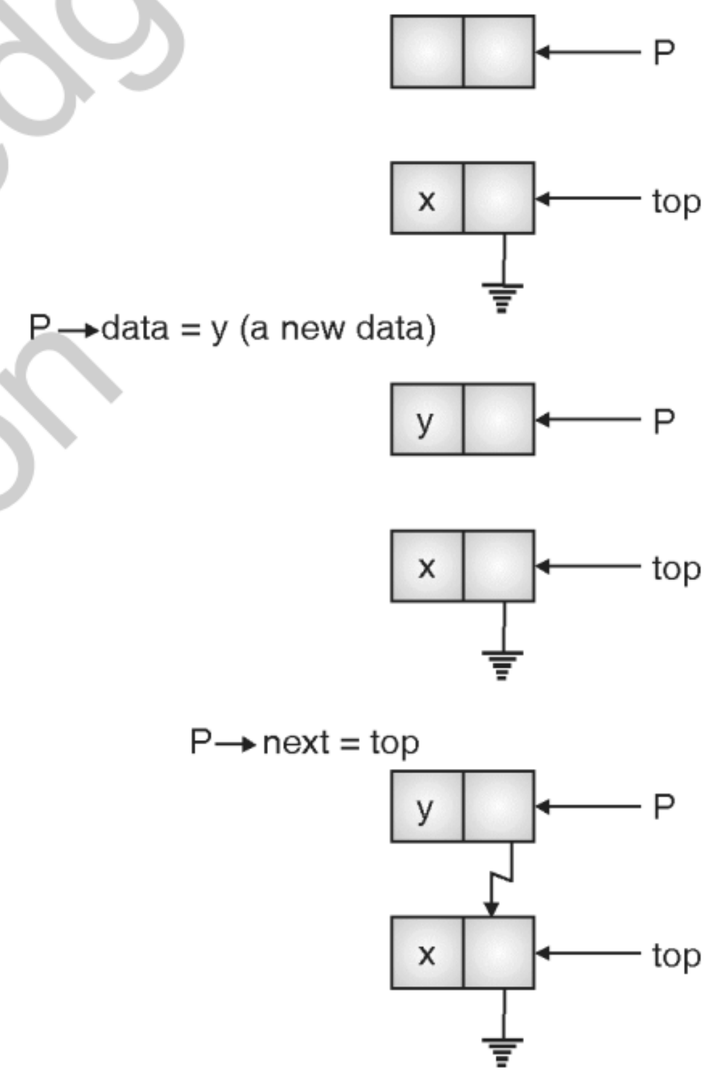


Fig. 7.4.3

– Address of the top node (address is stored in the pointer type variable top) is copied in the next field of node whose address is stored in P

```
top = P ;
```

Address of the new node is copied in the pointer top.

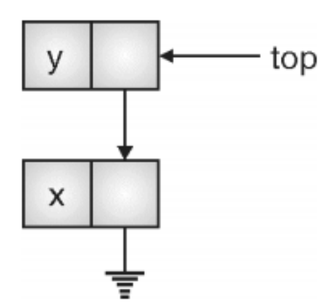


Fig. 7.4.4

Techknowledge Publication

– An element from the stack can be deleted through the following operations.

```
P = top
/* Save the address of the top node in a pointer P.
   So that the memory occupied by the node can
   be freed*/
top = top → next ;          //top node is deleted
free( P ) ;                 // memory freed
```

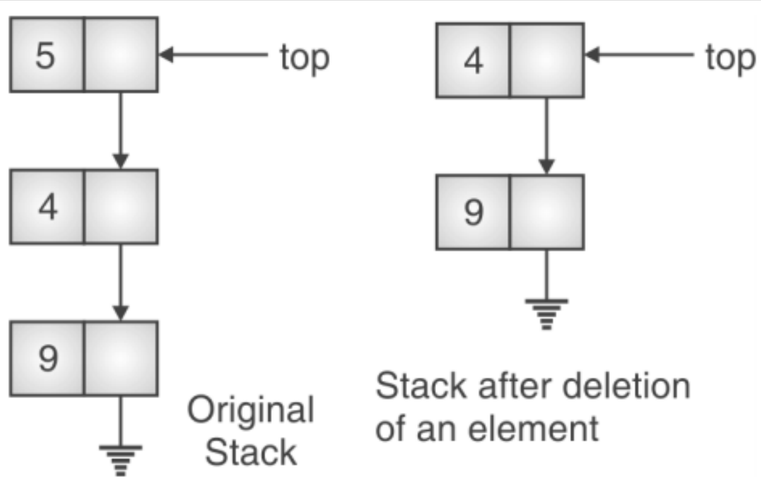


Fig. 7.4.5

7.4.1 Functions for Stack Operations

Q. Give pseudo C/C++ code to implement the following operations on linked stack.

(i) Create

(ii) Push data **SPPU - May 18, 4 Marks**

(a) init Function

```
void init(stack **T) ;
/*calling the function from the main program */
void main(c)
{
    stack * top ;
    init(& top) ;
}
```

Since, the function init( ) modifies the value of top, address of the pointer type variable 'top' must be passed to the function. If the content of a variable (defined in calling program) is to be modified by the called function then the calling program must pass its address. Pointer variable "top" is passed by address and the corresponding parameter in function init is declared as stack **T.

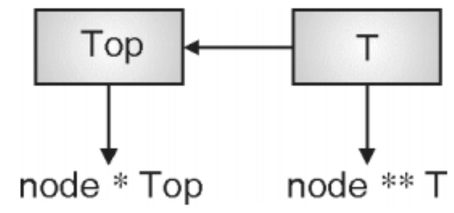


Fig. 7.4.6

Contents of the variable top can be accessed through T and it is simply *T.

(b) push Function

```
void push(stack ** T, int x)
```

insert a data in a stack, referenced by *T(Top)

(c) pop Function

```
int pop(stack ** T)
```

Delete the top element of a stack referenced by *T(Top) and return it to the calling program.

(d) empty Function

```
int empty(stack * Top)
```

Check, whether the stack is empty. Top is passed by value as the function empty( ) will not change the variable Top.

'C' functions for stack operations.

```
void init(stack **T)
{
    * T = NULL ; /* make the stack empty */
}
void push(stack ** T, int x)
{
    stack * P ;
    /* set a new node */
    P = (stack *) malloc(sizeof(stack)) ;
    P → data = x ;
    /* attach the node at the front  */
    P → next = *T ;
    * T = P ;
}
int pop(stack **T)
{
    int x ;
    stack * P ;
    P = * T ;
```

Techknowledge Publication

```
    *T = P → next ;
    x = P→ data ;
    free(P) ;
    return(x) ;
}
int empty(stack *Top)
{   if(Top ==  NULL)
    return(1) ;
    return(0) ;
}
```

A program showing usage of stack

Reverse a string using a stack represented using a linked list.

Program 7.4.1 : Program to reverse a string represented using a stack.

```
#include<stdio.h>
#include<conio.h>
typedef struct stack
{ char data;
  struct stack *next;
} stack;
void init(stack **);
int empty(stack *);
char pop(stack **);
void push(stack **,char);
void main()
{
  stack *TOP;
  char x;
  init(&TOP);
  printf("\n Enter the string :");
  while((x = getchar())! = '\n ')
      push(&TOP,x);
  printf("\n ");
  while(!empty(TOP))
  {
      x = pop(&TOP);
      printf("%c",x);
  }
}
void init(stack **T)
{
  *T = NULL;
}
int empty(stack *TOP)
{
  if(TOP == NULL)
      return(1);
  return(0);
}
void push(stack **T,char x)
{
  stack *P;
  P = (stack *)malloc(sizeof(stack));
  P–>data = x;
  P–>next = *T;
  *T = P;
}
char pop(stack **T)
{
  char x;
  stack * P;
  P = *T;
  *T = P–>next;
  x = P–>data;
  free(P);
  return(x);
}
```

Output

```
Enter the string :structure
erutcurts
```

7.5 Application of Stack

– Stack data structure is very useful. Few of its usages are given below :

1. Expression conversion
   (a) Infix to postfix (b) Infix to prefix
   (c) Postfix to infix (d) Prefix to infix
2. Expression evaluation
3. Parsing
4. Simulation of recursion
5. Function call

Techknowledge Publications

7.5.1 Expression Representation

Q. Write an algorithm for postfix with suitable example.
SPPU - May 17, 3 Marks

There are three popular methods for representation of an expression.

(a) infix x + y operator between operands

(b) prefix + x y operator before operands (Polish Notation)

(c) postfix x y + operator after operands (Reverse Polish Notation)

Example :

Infix x + y * z
Prefix + x * y z
Postfix x y z * +

(a) Evaluation of an infix expression

Infix expressions are evaluated left to right but operator precedence must be taken into account. To evaluate x + y * z, y and z will be multiplied first and then it will be added to x.

Note : Infix expressions are not used for representation of an expression inside computer, due to additional complexity of handling of precedence.

(b) Evaluation of a prefix expression

To understand the evaluation of prefix expression, let us consider an example

+ 5 * 3 2

Find an operator from right to left and perform the operation.

+ 5 * 3 2

first operator

First operator is * and therefore, 3 * 2 is performed.

Expression becomes : + 56

First operator is + and therefore, 5 and 6 are added.

Expression becomes : 11

(c) Evaluation of postfix (reverse polish) expression

Let us consider an example

5 3 2 * +

– Find the first from left to right and perform the operation.

– First operator is * and therefore, 3 and 2 are multiplied.

Expression becomes : 5 6 +

– First operator is + and therefore, 5 and 6 are added

Expression becomes : 11

Note : Prefix and postfix expressions are free from any precedence. They are more suited for mechanization. Computer uses postfix form for representation of an expression.

7.5.2 Evaluation of a Postfix Expression using a Stack

Q. Write an algorithm for postfix evaluation.
SPPU - May 17, 3 Marks

Q. Explain evaluation of postfix expression using stack with suitable example. **SPPU - May 18, 6 Marks**

Q. Explain the algorithm for evaluation of a postfix expression with an example.
SPPU - Dec. 18, 3 Marks

Given expression **stack**

6 5 3 + 9 * +

Initially stack is empty.

– First token is an operand, push 6 on the stack

5 3 + 9 * +

6

– Next token is an operand, push 5 on the stack

3 + 9 * +

5
6

– Next token is and operand, push 3 on the stack

+ 9 * +

3
5
6

– Next token is an operator, pop two operands 3 and 5, add them and push the result on the stack

Tech Knowledge Publication

9 * +

8
6

– Next token is an operand, push 9 on the stack

* +

9
8
6

– Next token is an operator, pop two operands 9 and 8, multiply them and push the result on the stack

+

72
6

– Next token is an operator, pop two operands 72 and 6, add them and push the result back on the stack

NULL | 78 | ← Final Result

Algorithm for evaluation of an expression

```
x : tokentype
    operand1, operand2 : operand_type ;
    operator : operator_ type ;
s : stack
    initialize(s) ;
    x = nexttoken( ) ;
    /* read the next token */
    while(x)
    {
    if(x is an operator)
    {
        operand2 = pop(s);
        operand1 = pop(s);
        push(s, evaluate(x, operand1, operand2));
    }
    else
        push(s, x) ;
        x = nexttoken( ) ;
    }   /* stack contains the evaluated value of the
            expression */
```

Program 7.5.1 : Program for evaluation of a postfix expression

```
/* Assumption-- primary operators '–, +, *, /, %'
   operand -- a single digit */
#include<stdio.h>
#include<conio.h>
#define MAX 20
typedef struct stack
{
    int data[MAX];
    int top;
}stack;
void init(stack *);
int empty(stack *);
int full(stack *);
int pop(stack *);
void push(stack *, int);
int evaluate(char x, int op1, int op2);
void main()
{
    stack s;
    char x;
    int op1, op2, val;
    init(&s);
    printf("\n Enter the expression(i.e59+3*)\n single
    digit operand and operators only:");
    while((x = getchar())! = '\n ')
    {
        if(isdigit(x))
            push(&s, x–48);
        else
        {
            op2 = pop(&s);
            op1 = pop(&s);
            val = evaluate(x, op1, op2);
            push(&s, val);
        }
    }
    val = pop(&s);
    printf("\n value of expression = %d", val);
}
int evaluate(char x, int op1, int op2)
{
```

Techknowledge Publications

```
    if(x == '+')
        return(op1 + op2);
    if(x == '–')
        return(op1 – op2);
    if(x == '*')
        return(op1 * op2);
    if(x == '/')
        return(op1 / op2);
    if(x == '%')
        return(op1 % op2);
}
void init(stack *s)
{
    s->top = –1;
}
int empty(stack *s)
{
    if(s->top == –1)
        return(1);
    return(0);
}
int full(stack *s)
{
    if(s->top == MAX –1)
        return(1);
    return(0);
}
void push(stack *s, int x)
{
    s->top = s->top + 1;
    s->data[s->top] = x;
}
int pop(stack *s)
{
    int x;
    x = s->data[s->top];
    s->top = s->top –1;
    return(x);
}
```

Output

```
Enter the expression(i.e59+3*)
single digit operand and operators only:653+9*+
value of expression = 78
```

Ex. 7.5.1 : Evaluate the following postfix expression and show stack after every step in tabular form.

Given : A = 5, B = 6, C = 2, D = 12, E = 4

ABC + *DE\–

Soln. :

Input	stack
ABC + * DE/–	
BC + *DE/–	5
C + * DE/–	6 5
+ * DE/–	2 6 5
* DE/-	8 5
DE/–	40
E/–	12 40
/–	4 12 40
–	3 40
End	37

value of the expression = 37

Techknowledge Publication

Ex. 7.5.2 : Find the value of expression 5 6 2 + * 12 4 / –

Show the contents of stack and variables operand 1, operand 2 and value.

Soln. :

Input	Stack	Operand 1	Operand 2	Value
5 6 2 + * 12 4 / –	Empty	–	–	–
6 2 + * 12 4 / –	5	–	–	–
2 + * 12 4 / –	6 5	–	–	–
+ * 12 4 / –	2 6 5	–	–	–
* 12 4 / –	8 5	6	2	8
12 4 / –	40	5	8	40
4 / –	12 40	–	–	–
/ –	4 12 40	–	–	–
–	3 40	12	4	3
End	37	40	3	37

TechKnowledge Publications

7.5.3 Evaluation of a Prefix (Polish) Expression

- A prefix expression is evaluated by scanning it from right to left.
- An operand is pushed on top of the stack.
- In case of an operator, two operands are popped from the stack, evaluated and pushed back onto the stack.
- Final result is found on top of the stack.

Ex. 7.5.3 : Evaluate the following prefix expression using stack for A = 16, B = 2 , C = 3, D = 10 and E = 4. Show step by step the contents of stack.

Prefix Expression = – + / A∧ BC * DE * AC

Soln. :

Input	Stack	
– + / A ∧ BC * DE * AC		
– + / A ∧ BC * DE * A	3	Push
– + / A ∧ BC * DE *	16 3	Push
– + / A ∧ BC * DE	48	Evaluate
– + / A ∧ BC * D	4 48	Push
– + / A ∧ BC *	10 4 48	Push
– + / A ∧ BC	40 48	Evaluate
– + / A ∧ B	3 40 48	Push
– + / A ∧	2 3 40 48	Push
– + / A	8 40 48	Evaluate
– + /	16 8 40 48	Push
– +	2 40 48	Evaluate
–	42 48	Evaluate
Null	–6	Evaluate

Value of expression = – 6

Algorithm for evaluation of a prefix expression

```
st : stack ;
initialize st ;
Prefix ← read the prefix expression ;
for(i ← index of the last character of prefix,
    down to 0 )
{
    x ← prefix[i] ;
    if( x is an operand )
    {   Push(st, char_to_int(x)) ;
    }
    else
    {   operand1 ← pop(st) ;
        operand2 ← pop(st) ;
        val ← evaluate( X, operand1 , operand2)
        push(st, val) ;
    }
}
val = pop(st) ;
print val ;
```

Program 7.5.2 : Evaluation of a prefix expression.

```
/* program for Evaluation of prefix expression
   operators supported '+, -, *, /, %, ∧, (, )
   operands supported -- all single character operands
*/
#include<stdio.h>
#include<conio.h>
#include<ctype.h>
```

Techknowledge Publication

```
#define MAX 50

typedef struct stack
{
    int data[MAX];
    int top;
}stack;

void init(stack *);
int empty(stack *);
int full(stack *);
int pop(stack *);
void push(stack *, int );
int top(stack *); //value of the top element
void eval_prefix(char prefix[ ]);
int evaluate(char x, int op1, int op2);

void main()
 {
    char prefix[30];
    clrscr();
    printf("\n Enter a prefix expression : ");
    gets(prefix);
    eval_prefix(prefix);
    getch();
 }
void eval_prefix(char prefix[ ])
{
    stack s;
    char x;
    int op1, op2, val, i;
    init(&s);
    for(i = strlen(prefix)–1; i> = 0; i– –)
     {
        x = prefix[i];
        if(isalpha(x))
        {    printf("\n Enter the value of %c : ", x);
             scanf("%d", &val);
             push(&s, val);
        }
        else
        {    //pop two operands and evaluate
             op1 = pop(&s);
             op2 = pop(&s);
             val = evaluate(x, op1, op2);
             push(&s, val);
        }
    }
    val = pop(&s);
    printf("\n value of expression = %d", val);
}
int evaluate(char x, int op1, int op2)
{
    if(x == ' + ')  return(op1 + op2);
    if(x == '–')  return(op1 – op2);
    if(x == '*')  return(op1 * op2);
    if(x == '/')  return(op1 / op2);
    if(x == '%')  return(op1 % op2);
}
void init(stack *s)
{   s->top = –1;
}
int empty(stack *s)
{
    if(s->top == –1)
    return(1);
    return(0);
}
int full(stack *s)
{   if(s->top == MAX –1)
    return(1);
    return(0);
}
void push(stack *s, int x)
{   s->top = s->top + 1;
    s->data[s->top] = x;
}
int pop(stack *s)
{   int x;
    x = s->data[s->top];
```

Techknowledge Publication

```
    s->top = s->top –1;
    return(x);
}
int top(stack * p)
{   return(p->data[p->top]);
}
```

7.5.3(A) Conversion of an Expression from Infix to Postfix

Q. Write an algorithm to convert infix expression to postfix expression **SPPU - May 17, 6 Marks**

– Example of an infix expression :
(A + B ∧ C) * D + E ∧ 5, where A, B, C, D and E are integer constants and B ∧ C means B^C

Manual method

Step 1 : In manual evaluation of the above expression, B ∧ C must be calculated first. Hence, we convert B ∧ C to its equivalent postfix equivalent i.e. BC ∧

(A + B ∧ C) * D + E ∧ 5 : Original expression

(A + BC∧) * D + E ∧ 5 : BC∧ should treated as a single integer number (single token) and A + BC∧ should be the next operation to be performed

ABC∧ + * D + E ∧ 5 : ABC∧ + * D is the next operation to be performed

ABC∧ + D* + E ∧ 5 : E ∧ 5 is the next operation to be performed

ABC∧ + D* + E5∧ : + is the last operation to be performed

ABC∧ + D* E5∧ + : Final expression.

Algorithmic approach

A + B * C

– In the above example, evaluation of * precedes evaluation of + as * has higher precedence over +.

Note : Perform an operation if the current operator has equal or higher precedence over the succeeding operator.

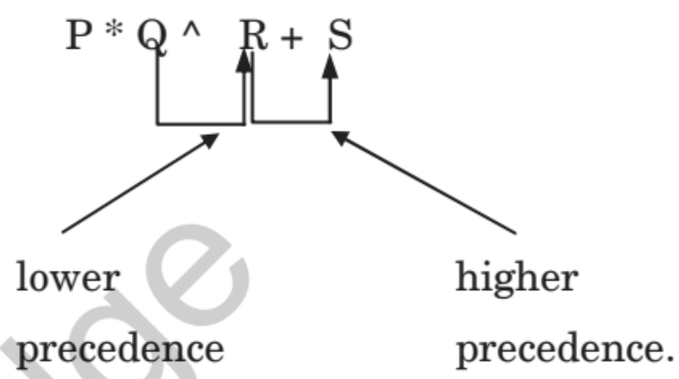
– Thus an operator coming on the right of the current operator will determine if the current operation should be performed.

P + Q + R

– Same precedence and hence P + Q can be performed

P * Q + R

– Higher precedence and hence P * Q can be performed



Sequence in which operations will be performed ^, *, +

– All operators must be saved on top of the stack until we get an equal or higher precedence operator

Conversion of P * Q ∧ R + S

Expression	Stack	Output	Remark
P*Q ∧ R + S	NULL	–	–
*Q∧R + S	NULL	P	Operand must be printed
Q∧R + S	*	P	* operation will be performed if the next operator is of lower or equal precedence.
∧R + S	*	PQ	Operand must be printed
R + S	* ∧	PQ	* Cannot be performed as ∧ has higher precedence
+ S	* ∧	PQR	Operand must be printed
+ S	NULL	PQR∧*	All higher precedence operators (compare to +) are

Technknowledge Publications

Expression	Stack	Output	Remark
			popped and printed and finally the current operator is pushed on top of the stack.
S	+	PQR∧*	–
NULL	+	PQR∧* S	Operand must be printed
Null	Null	PQR∧* S +	Finally, all operators are popped and printed.

Algorithm for conversion of an expression from infix to postfix

```
s : stack
    while(more tokens)
    {   x ← next token ;
        if(x is an operand)
            print x
        else
        while(precedence(x)<=precedence(top(s))
        {   print(pop(s))
            push(s, x)
        }
    }
    while(! empty(s))
    print(pop(s)) ;
```

Ex. 7.5.4 : Convert the following expression from infix to postfix using a stack.

a && b ¦¦ c ¦¦ ! (e > f)

Soln. :

Expression	Stack	Output	Comment
a &&b ¦¦ c ¦¦ ! (e>f)	NULL	–	Initial condition
&&b ¦¦ c ¦¦ ! (e >f)	NULL	a	Print a
b ¦¦ c ¦¦ ! (e > f)	&&	a	Push &&
¦¦ c ¦¦ ! (e>f)	&&	ab	Print b
¦¦ c ¦¦ ! (e>f)	NULL	ab &&	pop and print higher precedence operators
c ¦¦ ! (e>f)	¦¦	ab &&	push the current operator
¦¦ ! (e>f)	¦¦	ab&& c	Print c
¦¦ ! (e>f)	NULL	ab&&c ¦¦	Pop and print equal or higher precedence operators
!(e > f)	¦¦	ab&&c ¦¦	Push the current operator
! (e>f)	¦¦	ab&&c ¦¦	Pop and print equal or higher precedence operators
(e > f)	¦¦ !	ab&&c ¦¦	Push the current operator
e > f	¦¦ ! (	ab&&c ¦¦	'(' should always be pushed.
> f)	¦¦ ! (	ab&&c ¦¦ e	Print e
> f)	¦¦ ! (	ab&&c ¦¦ e	Pop and print equal or higher precedence operators. Remember '(' has lowest precedence
f)	¦¦ ! (>	ab&&c ¦¦ e	Push the current operator

Tech Knowledge Publications

Expression	Stack	Output	Comment
)	¦ ¦ ! (>	ab&&c ¦ ¦ e f	Print f
NULL	¦ ¦ !	ab&&c ¦ ¦ e f >	when the next input is`)' then all operators until '(' should be popped and printed
NULL	NULL	ab&&c ¦ ¦ e f >! ¦ ¦	Pop and print all operations.

Ex. 7.5.5 : Convert the following arithmetic expression into postfix and show stack status after every step in tabular forms :
A + (B*C – (D/E – F) * G) * H

Soln. :

Stack	Input	Output
Empty	A + (B * C– (D/E – F) * G) * H	–
Empty	+ (B * C– (D/E – F) * G) * H	A
+	(B * C– (D/E – F) * G) * H	A
+ (	B * C– (D/E – F) * G) * H	A
+ (	* C– (D/E – F) * G) * H	AB
+ (*	C– (D/E – F) * G) * H	AB
+ (*	– (D/E – F) * G) * H	ABC
+ (–	(D/E – F) * G) * H	ABC*
+ (– (	D/E – F) * G) * H	ABC*
+ (– (	/E – F) * G) * H	ABC*D
+ (– (/	E – F) * G) * H	ABC*D
+ (– (/	– F) * G) * H	ABC*DE
+ (– (–	F) * G) * H	ABC*DE/
+ (– (–	) * G) * H	ABC*DE/F
+ (–	* G) * H	ABC*DE/F–
+ (– *	G) * H	ABC*DE/F–
+ (– *	) * H	ABC*DE/F–G
+	* H	ABC*DE/F–G*–
+*	H	ABC*DE/F–G*–
+ *	End	ABC*DE/F–G*–H
Empty	End	ABC*DE/F–G*–H*+

TechKnowledge Publications

Ex. 7.5.6 : Convert the following arithmetic expression into postfix and show stack status after every step at each step :

(i) (A + B) * D + E / (F + A * D) + C (ii) A * (B + C) / D – G

Soln. : (i) (A + B) * D + E / (F + A * D) + C

Stack	Input	Output
Empty	(A + B) * D + E / (F + A * D) + C	–
(	A + B) * D + E / (F + A * D) + C	–
(	+ B) * D + E / (F + A * D) + C	A
(+	B) * D + E / (F + A * D) + C	A
(+	) * D + E / (F + A * D) + C	AB
Empty	* D + E / (F + A * D) + C	AB+
*	D + E / (F + A * D) + C	AB+
*	+ E / (F + A * D) + C	AB+D
+	E / (F + A * D) + C	AB+D*
+	/ (F + A * D) + C	AB+D*E
+ /	(F + A * D) + C	AB+D*E
+ / (	F + A * D) + C	AB+D*E
+ / (	+ A * D) + C	AB+D*EF
+ / (+	A * D) + C	AB+D*EF
+ / (+	* D) + C	AB+D*EFA
+ / (+ *	D) + C	AB+D*EFA
+ / (+ *	) + C	AB+D*EFAD
+ /	+ C	AB +D*EFAD*+
+	C	AB +D*EFAD*+ / +
+	End	AB +D*EFAD*+/+C
Empty	End	AB +D*EFAD*+/+C+

(ii) A * (B + C) / D – G

Stack	Input	Output
Empty	A * (B + C) / D – G	–
Empty	* (B + C) / D – G	A
*	(B + C) / D – G	A
*(	B + C) / D – G	A
*(	+ C) / D – G	AB
*(+	C) / D – G	AB
*(+	) / D – G	ABC
*	/ D – G	ABC+
/	D – G	ABC+*

Tech Knowledge Publications

Stack	Input	Output
/	– G	ABC+*D
–	G	ABC+*D/
–	End	ABC+*D/G
Empty	End	ABC+*D/G–

Ex. 7.5.7 : Explain the stepwise conversion using stack for the given infix expression to the postfix expression : A * (B + C) * D **SPPU - Dec. 17, 2 Marks**

Soln. :

Sr. No.	Operation (Stack)	Infix expressions	Postfix expression
1.	Empty	A*(B + C)*D	–
2.	Empty	*(B + C)*D	A
3.	*	(B + C)*D	A
4.	*(	B + C)*D	A
5.	*(	+C*D	AB
6.	*(+	C)*D	AB
7.	*(+	)*D	ABC
8.	*	*D	ABC+
9.	*	D	ABC++
10.	*	–	ABC+*D
11.	Empty	–	ABC+*D*

Ex. 7.5.8 : Explain the stepwise conversion using stack for the given infix expression to the postfix expression : A * B + C * D. **SPPU - May 18, 2 Marks**

Soln. :

Sr. No.	Stack	Infix expression	Postfix expression
1.	Empty	A * B + C * D	–
2.	Empty	* B + C * D	A
3.	*	B + C * D	A
4.	*	+ C * D	AB
5.	+	C * D	AB *
6.	+	*D	AB * C
7.	+ *	D	AB * C
8.	+ *	–	AB * CD
9.	Empty	–	AB * CD * +

∴ Postfix expression : AB * CD * +

Technknowledge Publications

Ex. 7.5.9 : Convert the following expression from infix to postfix using a stack. [a + (b–c)] *[(d–e) / (f–g + h)].

Soln. :

Expression	Stack	Output
[a + (b–c)] * [(d–e)/ (f–g +h)]	NULL	–
a + (b–c)] * [(d–e)/ (f–g +h)]	[	–
+ (b–c)] * [(d–e)/ (f–g +h)]	[	a
(b–c)] * [(d–e)/ (f–g +h)]	[+	a
b–c)] * [(d–e)/ (f–g + h)]	[+ (	a
–c)] * [(d–e)/ (f–g +h)]	[+ (	ab
c)] * [(d–e)/ (f–g +h)]	[+ (–	ab
)] * [(d–e)/ (f–g +h)]	[+ (–	abc
] * [(d–e)/ (f–g +h)]	[+	abc –
* [(d–e)/ (f–g +h)]	NULL	abc – +
[(d–e)/ (f–g +h)]	*	abc – +
(d–e)/ (f–g +h)]	*[	abc – +
d–e)/ (f–g +h)]	*[(	abc – +
–e)/ (f–g +h)]	*[(	abc – + d
e)/ (f–g +h)]	* [(–	abc – + d
)/ (f–g +h)]	* [(–	abc – + de
/ (f–g +h)]	*[	abc – + de –
(f–g +h)]	*[/	abc – + de –
f–g +h)]	* [/ (	abc – + de –
–g +h)]	* [/ (	abc – + de – f
g +h)]	* [/ (–	abc – + de – f
+h)]	* [/ (–	abc – + de – fg
h)]	* [/ (+	abc – + de – fg –
)]	* [/ (+	abc – + de – fg –h
]	*[/	abc – + de – fg –h +
NULL	*	abc – + de – fg –h + /
NULL	NULL	abc – + de – fg –h + / *

Techknowledge Publication

Ex. 7.5.10 : Convert following infix expression to an equivalent postfix expression : A + B * C – D/F

Soln. :

Expression	Stack	Output	Comment
A + B * C – D/F	NULL	–	Initial condition
+ B * C – D/F	NULL	A	Print
B*C – D/F	+	A	Pop equal and higher precedence operators from the stack and then push the current operator.
* C – D/F	+	AB	Print
C – D /F	+ *	AB	Pop equal and higher precedence operators from the stack and then push the current operator
– D/F	+ *	ABC	Print
D/F	–	ABC * +	Pop equal and higher precedence operators from the stack and then push the current operator
/F	–	ABC * + D	Print
F	–/	ABC * + D	Pop equal and higher precedence operators from the stack and then push the current operator
–	–/	ABC* + DF	Print
–	NULL	ABC * + DF/–	Pop all operators, one by one from the stack and print them.

Postfix equivalent = ABC * + DF /–

Ex. 7.5.11 : Translate the following infix expression in its equivalent postfix expression.

(a + b) ld↑ ((e – f) + g)

Soln. :

Input	Stack	Output
(a + b) | d↑ ((e – f) + g)	NULL	–
(a + b) | d↑ ((e – f) + g)	(	–
+ b) | d↑ ((e – f) + g)	(	a
b) | d↑ ((e – f) + g)	(+	a
) | d↑ ((e – f) + g)	(+	ab
| d↑ ((e – f) + g)	NULL	ab+
d↑ ((e – f) + g)	|	ab+
↑ ((e – f) + g)	|	ab+d
((e – f) + g)	|↑	ab+d
(e – f) + g)	|↑(	ab+d

Tech Knowledge Publications

Input	Stack	Output
e – f) + g)	|↑((	ab+d
– f) + g)	|↑((	ab+de
f) + g)	|↑((–	ab+de
) + g)	|↑((–	ab+def
+ g)	|↑(	ab+def –
g)	|↑(+	ab+def –
)	|↑(+	ab+def –g
–	|↑	ab+def –g+
–	NULL	ab+def –g+↑|

Ex. 7.5.12 : Explain step by step conversion using stack for given infix expression to postfix expression.
((a / (b – c + d)) * (e – a) * c **SPPU - Dec. 16, 6 Marks**

Soln. :

Sr. No.	Stack	Infix Expression	Postfix Expression
1.	Empty	(a/ (b – c + d)) * (e – a) * c	–
2.	(	a/(b – c + d)) * (e – a) * c	–
3.	(	/(b – c + d)) * (e – a) * c	a
4.	(/	(b – c + d)) * (e – a) * c	a
5.	(/(	b – c + d)) * (e – a) * c	a
6	(/(	– c + d)) * (e – a) * c	ab
7.	(/(–	c + d)) * (e – a) * c	ab
8.	(/(–	+ d)) * (e – a) * c	abc
9.	(/(+	d)) * (e – a) * c	abc –
10.	(/(+	)) * (e – a) * c	abc – d
11.	(/	)* (e – a) * c	abc – d +
12.	empty	* (e – a) * c	abc – d+/
13.	*	(e – a) * c	abc – d +/
14.	*(	e – a) * c	abc – d +/e
15.	*(	– a) * c	abc – d +/e
16.	*(–	a) * c	abc – d +/e
17.	*(–	) * c	abc – d +/ea
18.	*	* c	abc – d +/ea–
19.	*	c	abc – d +/ea –*
20.	*	–	abc – d +/ea –*c
21.	–	–	abc – d +/ea – *c*

Techknowledge Publication

Ex. 7.5.13 : Explain process of conversion of an infix expression to post fix expression using stack : A * (B – C)/E ^ F + G. **SPPU - May 19, 6 Marks**

Soln. :

Sr. No.	Stack	Infix expression	Postfix expressions
1	Empty	A * (B – C)/E ∧ F + G	–
2	Empty	* (B – C)/E ∧ F + G	A
3	*	(B – C)/E ∧ F + G	A
4	* (	B – C)/E ∧ F + G	A
5	* (	– C) /E ∧ F + G	AB
6	* (–	C) /E ∧ F + G	AB
7	* (–	) /E ∧ F + G	ABC
8	*	/E ∧ F + G	ABC –
9	/	E ∧ F + G	ABC – *
10	/	∧ F + G	ABC – * E
11	/ ∧	F + G	ABC – * E
12	/ ∧	+ G	ABC – * E F
13	+	G	ABC – * E F ∧ /
14	+	–	ABC – * E F ∧ / G
15	Empty	–	ABC – * E F ∧ / G +

Ex. 7.5.14 : Convert following infix expression to postfix expression : ((A + B) – C* (D / E)) + F. Use stack and show step by step conversion. **SPPU - Dec. 19, 6 Marks**

Soln. :

Sr. No.	Stack	Infix expression	Postfix (Output)
1.	Empty	((A + B) – C*(D/E)) + F	–
2.	(	(A + B) – C*(D/E)) + F	–
3.	((	A + B) – C*(D/E)) + F	–
4.	((	+ B) – C* (D/E)) + F	A
5.	((+	B) – C* (D/E)) + F	A
6.	((+	) – C* (D/E)) + F	AB
7.	(	– C* (D/E)) + F	AB +

Tech Knowledge Publications

Sr. No.	Stack	Infix expression	Postfix (Output)
8.	(–	C* (D/E)) + F	AB +
9.	(–	* (D/E)) + F	AB + C
10.	(–*	(D/E)) + F	AB + C
11.	(–*(	D/E)) + F	AB + C
12.	(–*(	/E)) + F	AB + CD
13.	(–*(/	E)) + F	AB + CD
14.	(–*(/	)) + F	AB + CDE
15.	(–*	)+ F	AB + CDE/
16.	Empty	+ F	AB + CDE/*–
17.	+	F	AB + CDE/* –
18.	+	End	AB + CDE/*– F
19.	Empty	End	AB + CDE/* – F +

Result = AB + CDE/* – F+

Program 7.5.3 : Program for conversion of infix to its postfix form operators supported '+,-,*,/,%,∧,(,) operands supported -- all single character operands

```
/* Program for conversion of  infix into its postfix form
   operators supported '+, -, *, /, %, ∧, (, )
   operands supported -- all single character operands
 */
#include<stdio.h>
#include<conio.h>
#include<ctype.h>
#define MAX 50
typedef struct stack
{
   int data[MAX];
   int top;
}stack;
int precedence(char);
void init(stack *);
int empty(stack *);
int full(stack *);
int pop(stack *);
void push(stack *, int );
int top(stack *);              //value of the top element
void infix_to_postfix(char infix[ ], char postfix[ ]);

void main()
 {
   char infix[30], postfix[30];
   clrscr();
   printf("\n Enter an infix expression : ");
   gets(infix);
   infix_to_postfix(infix, postfix);
   printf("\n Postfix : %s  ", postfix);
   getch();
 }
void infix_to_postfix(char infix[ ], char postfix[ ])
{
   stack s;
   char x;
   int i, j;    //i: index for infix[ ], j: index for postfix
   char token;
   init(&s);
```

TechKnowledge Publications

```
    j = 0;
    for(i = 0; infix[i]! = '\0'; i++)
    {
        token = infix[i];
        if(isalnum(token))
            postfix[j++] = token;
        else
            if(token == '(')
                push(&s, '(');
            else
                if(token == ')')
                  while((x = pop(&s))! = '(')
                      postfix[j++] = x;
                else
                {
while(precedence(token)< = precedence(top(&s))
&& !empty(&s))
                    {
                        x = pop(&s);
                        postfix[j++] = x;
                    }
                    push(&s, token);
                }
    }
    while(!empty(&s))
    {
        x = pop(&s);
        postfix[j++] = x;
    }
postfix[j] = '\0';
}
int precedence(char x)
{
    if(x == '(')
        return(0);
    if(x == '+' || x == '–')
        return(1);
    if(x == '*' || x == '/' || x == '%')
        return(2);
    return(3);
}

void init(stack *s)
{
    s->top = – 1;
}
int empty(stack *s)
{
    if(s->top == – 1)
        return(1);
    return(0);
}
int full(stack *s)
{
    if(s->top == MAX – 1)
        return(1);
    return(0);
}

void push(stack *s, int x)
{
    s->top = s–>top + 1;
    s->data[s–>top] = x;
}

int pop(stack *s)
{
    int x;
    x = s->data[s–>top];
    s->top = s–>top – 1;
    return(x);
}

int top(stack * p)
{   return(p–>data[p–>top]);
}
```

Output

```
Enter infix expression:a*(b+c)/d+g
abc+*d/g+
```

Techknowledge Publication

7.5.4 Conversion of an Expression from Infix to Prefix

Example of an infix expression :

(A + B ∧ C) * D + E ∧ 5 where A, B, C, D and E are integer constant and B ∧ C stands for B^C.

Manual method :

In manual evaluation of the above expression, B^C must be calculated first. Hence, we convert B^C to its equivalent pretfix equivalent i.e. ^BC.

(A + B^C) * D + E^5 : original expression

(A + ^BC) * D + E^5 : ^BC should be treated as a single operand.

+ A^BC * D + E^5 : A + ^BC is the next operation.

* + A^BCD + E^5 : +A^BC*D is the next operation.

* + A^ BCD + ^E5 : E^5 is the next operation.

+ * + A^BCD^E5 : Addition is the last operation.

Algorithmic approach :

An infix expression can be converted into prefix form with the help of following steps :

1. Reverse the infix expression
2. Make every '(' (opening bracket) as ')' (closing bracket) and every ')' as '('
3. Convert the modified expression to postfix form. Use the algorithm given in the section 7.5.3
4. Reverse the postfix expression

Above algorithm can be explained by carrying out the steps from 1 to 4 on the example :

(A + B ∧ C) * D + E ∧ 5

Step 1 : Reverse the infix expression.

5 ∧ E + D *) C ∧ B + A(

Step 2 : Make every '(' as ')' and every ')' and '('

5∧ E + D * (C ∧ B + A)

Step 3 : Convert the expression to postfix form :

Expression (input)	Stack	Output	Comment
5∧ E + D * (C∧ B + A)	Empty	–	Initially
∧ E + D * (C∧ B + A)	Empty	5	Print
E + D * (C∧ B + A)	∧	5	Push
+ D * (C∧ B + A)	∧	5 E	Push
D * (C∧ B + A)	+	5 E ∧	Pop and Push
* (C∧ B + A)	+	5 E ∧ D	Print
(C∧ B + A)	+*	5 E ∧ D	Push
C∧ B + A)	+*(	5 E ∧ D	Push
∧ B + A)	+*(	5 E ∧ D C	Print
B + A)	+*(∧	5 E ∧ D C	Push
+ A)	+*(∧	5 E ∧ D C B	Print
A)	+*(+	5 E ∧ D C B∧	Pop and Push
)	+*(+	5 E ∧ D C B ∧ A	Print
End	+*	5 E ∧ D C B ∧ A +	Pop until '('
End	Empty	5 E ∧ D C B ∧ A + * +	Pop everything

Step 4 : Reverse the expression + * + A ^ B C D ^ E 5

Tech Knowledge Publications

Pseudo code for conversion of an expression from infix into prefix

```
{
    S : stack of characters ;
    infix ← read the infix expression;
    infix ← reverse the infix expression ;
    infix ← reverse the direction of brackets ;
    prefix ← A NULL string
    i = 0 ;
    while( infix [i] is not equal to '\0)
    {
        x ← infix [i] ;
        if( x is an operand )
            add the character x at the end of prefix ;
        else
            if( x is '(' )
                push(S, '(')
            else
                if( x is ')' )
                {
                    while(( x ← pop(S)) ! = '(' )
                    prefix ← prefix + pop(s) ;
                }
                else
                {
while(precedence(x)<precedence(top(S))
                    {
                        prefix ← prefix + pop(S) ;
                    }
                        push(S, X) ;
                }
        i ← i + 1 ;
    }
        while( ! empty(S) )
            prefix ← prefix + pop(s) ;
        prefix ← reverse the prefix ;
        print prefix ;
    }
```

Program 7.5.4 : To convert infix expression to prefix expression.

```
/* Program for conversion of infix into its prefix form
   operators supported '+, -, *, /, %, ∧, (, )
   operands supported -- all single character operands
*/
#include<stdio.h>
#include<conio.h>
#include<ctype.h>
#define MAX 50
typedef struct stack
{
    int data[MAX];
    int top;
}stack;
int precedence(char);
void init(stack *);
int empty(stack *);
int full(stack *);
int pop(stack *);
void push(stack *, int );
int top(stack *);              //value of the top element
void infix_to_prefix(char infix[ ], char prefix[ ]);
void infix_to_postfix(char infix[ ], char postfix[ ]);

void main()
{
    char infix[30], prefix[30];
    clrscr();
    printf("\n Enter an infix expression : ");
    gets(infix);
    infix_to_prefix(infix, prefix);
    printf("\n prefix: %s  ", prefix);
    getch();
}
void infix_to_prefix(char infix[ ], char prefix[ ])
 {
    int i, j;
    char temp, in1[30];
    // reverse the infix expression  & store it in in1[ ]
    for(i = strlen(infix)-1, j = 0; i> = 0; i– –, j+ +)
    in1[j] = infix[i];
    in1[j] = '\0';
```

Technknowledge Publications

```
    // reverse the direction of brackets
    for(i = 0; in1[i]! = '\0'; i++)
    {
        if(in1[i] == '(')
            in1[i] = ')';
        else
            if(in1[i] == ')')
            in1[i] = '(';
    }
    // convert from infix to postfix
    infix_to_postfix(in1, prefix);
    //reverse the final expression
    for(i = 0, j = strlen(prefix) –1; i<j; i++, j– –)
    {
        temp = prefix[i];
        prefix[i] = prefix[j];
        prefix[j] = temp;
    }
}
void infix_to_postfix(char infix[ ], char postfix[ ])
{
    stack s;
    char x;
    int i, j;   //i : index for infix[ ], j : index for postfix
    char token;
    init(&s);
    j = 0;
    for(i = 0; infix[i]! = '\0'; i++)
    {
        token = infix[i];
        if(isalnum(token))
            postfix[j++] = token;
        else
            if(token == '(')
                push(&s, '(');
            else
                if(token == ')')
                  while((x = pop(&s))! = '(')
                      postfix[j++] = x;
                else
                {
                while(precedence(token)< precedence
                    (top(&s)) && !empty(&s))
                    {
                        x = pop(&s);
                        postfix[j++] = x;
                    }
                    push(&s, token);
                }
        }
    while(!empty(&s))
    {
        x = pop(&s);
        postfix[j++] = x;
    }
postfix[j] = '\0';
}

int precedence(char x)
{
    if(x == '(')
        return(0);
    if(x == '+' || x == '–')
        return(1);
    if(x == '*' || x == '/' || x == '%')
        return(2);
    return(3);
}
void init(stack *s)
{
    s->top = –1;
}
int empty(stack *s)
{
    if(s->top == –1)
        return(1);
    return(0);
}
int full(stack *s)
{
```

Techknowledge Publications

```
    if(s->top == MAX–1)
        return(1);
    return(0);
}

void push(stack *s, int x)
{
    s->top = s->top + 1;
    s->data[s->top] = x;
}

int pop(stack *s)
{
    int x;
    x = s->data[s->top];
    s->top = s->top–1;
    return(x);
}

int top(stack * p)
{
    return(p->data[p–>top]);
}
```

7.5.5 Conversion of Expression from Postfix into Infix

Manual method

Given expression : abc – + de – fg – h + / *

bc – should be converted to b – c

and should be treated as a single token.

a [b – c] + de – fg – h + / *

[a + b – c] de – fg – h + / *

[a + b – c] [d – e] fg – h + / *

[a + b – c] [d – e] [f – g] h + / *

[a + b – c] [d – e] [f – g + h] / *

[a + b – c] [(d – e) / (f – g + h)] *

(a + b – c) * (d – e) / (f – g + h)

– Expression is scanned from left to right.

– Whenever an operator is found, operator is put in between the previous two operands.

Conversion through a stack :

Expression	Stack (a stack of strings)
abc – + de –fg –h + / *	NULL
bc – + de – fg –h + /*	"a"
c – + de –fg – h + /	"b" "a"
– + de – fg – h + / *	"c" "b" "a"
+ de – fg –h + / *	"b –c" "a"
de – fg – h + / *	"a + b – c"
e – fg – h + / *	"d" "a + b – c"
– fg – h + / *	"e" "d" "a + b – c"
fg – h + / *	"d – e" "a + b – c"
g – h + / *	"f" "d – e" "a + b – c"
– h + / *	"g" "f" "d – e" "a + b – c"
h + / *	"f – g" "d – e" "a + b – c"
+ / *	"h" "f – g" "d – e" "a + b – c"

Techknowledge Publication

/ *	"f – g + h" "d – e" "a + b – c"
*	"(d – e) / (f – g + h)" a + b – c
NULL	(a + b– c) * (d – e) / (f – g + h)

Algorithm for conversion from postfix into infix

```
s1, s2 : string  ;
st : stack of  character strings ;
initialize st
x = nexttoken( ) ;
while(more tokens)
{
    if(x is an operand)
        push(st, x) ;
    else
    {   s2 = pop(st) ;
        s1 = pop(st) ;
        push(st, "(" + s1+x+s2+")" ) ;
        /* + → concatenation operator * /
    }
    x = nexttoken( ) ;
    // stack contains the final expression in infix from
}
```

Program 7.5.5 : To convert postfix into infix

```
/*Conversion of an expression from postfix to infix*/
#include<ctype.h>
#include<stdio.h>
#include<conio.h>
#define MAX 20
#include<string.h>
void convert_infix(char x);
char stack[MAX][MAX];
void postfix_to_infix(char postfix[ ], char prefix[ ]);
int top;
// array stack[ ][ ] is being used  as a stack of strings
void main()
{
    char postfix[30], infix[30];
    clrscr();
    printf("\n Enter a postfix expression :");
    gets(postfix);
    postfix_to_infix(postfix, infix);
    printf("\n Infix : %s", infix);
    getch();
}
void postfix_to_infix(char postfix[ ], char infix[ ])
{   char x, st1[20];
    int i;
    top = –1;
    for(i = 0; postfix[i]! = '\0'; i++)
    //scan postfix expression From left to right
    {
        x = postfix[i];
        if(isalnum(x))
        {
            /* convert token to string form */
            st1[0] = x;
            st1[1] = '\0';
            // push the operand on the stack s2
            top = top + 1;
            strcpy(stack[top], st1);
        }
        else // if operator, convert to infix
        convert_infix(x);
    }
    //Result is on top of the stack
    strcpy(infix, stack[top]);
}
void convert_infix(char x)
{
    char st1[30], st2[10];
```

Technknowledge Publication

```
    st1[0] = '(';
    st1[1] = '\0';
    strcat(st1, stack[top – 1]);
    st2[0] = x;
    st2[1] = '\0';
    strcat(st1, st2);
    strcat(st1, stack[top]);
    st2[0] = ')';
    st2[1] = '\0';
    strcat(st1, st2);
    top = top – 1;
    strcpy(stack[top], st1);
}
```

7.5.6 Conversion of Expression from Postfix into Prefix

- Expression is scanned from left to right.
- Whenever an operator is found, it is put before the previous two operands

Example : abc – + de – fg –h + / *

a [– bc] + de – fg – h + / *

[+ a – bc] de – fg – h + /*

[+ a – bc] [– de] fg – h + / *

[+ a – bc] [– de] [– fg] h + / *

[+ a – bc] [– de] [+ – f g h] / *

[+ a – bc] [/ –de + – fgh] *

[* + a –bc / – de + – fgh]

Algorithm for conversion from postfix to prefix

```
S1 , S2 : string
st : stack of character strings ;
initialize st
x = nexttoken( ) ;

while( more tokens )
{
    if(x is an operand )
        push(st, x) ;
    else
    {
        S2 = pop(st) ;
        S1 = pop(st) ;
        push(st, x + S1 + S2) ;
        /* + is a concatenation operator */
    }
    x = nexttoken( ) ;
}
// stack contains the final expression in prefix form
```

Program 7.5.6 : To convert postfix to prefix

```
// Conversion of an expression from postfix to prefix

#include<ctype.h>
#include<stdio.h>
#include<conio.h>
#define MAX 20
#include<string.h>

void convert_prefix(char x);
char stack[MAX][MAX];
void postfix_to_prefix(char postfix[ ], char prefix[ ]);
int top;

//array stack[ ][ ] is being used as a stack of strings

void main()
{
    char postfix[30], prefix[30];
    clrscr();
    printf("\n Enter a postfix expression :");
    gets(postfix);
    postfix_to_prefix(postfix, prefix);
    printf("\n Prefix :    %s", prefix);
    getch();
}
```

Techknowledge Publication

```
void postfix_to_prefix(char postfix[ ], char prefix[ ])
{
    char x, st1[20];
    int i;
    top = –1;
    for(i = 0; postfix[i]! = '\0'; i++)
    //scan the infix expression from left to right.
    {
        x = postfix[i];
        if(isalnum(x))
        {
            /* convert character x  to string format */
            st1[0] = x;
            st1[1] = '\0';
            // push the operand on the stack s2
            top = top + 1;
            strcpy(stack[top], st1);
        }
        else  // if operator , convert to prefix
        {
            convert_prefix(x);
        }
    }
    //Result is on top of the stack
    strcpy(prefix, stack[top]);
}

void convert_prefix(char x)
{
    char st1[30];
    st1[0] = x;
    st1[1] = '\0';
    strcat(st1, stack[top – 1]);
    strcat(st1, stack[top]);
    top = top –1;
    strcpy(stack[top], st1);
}
```

Ex. 7.5.15 : Convert the following postfix expression to infix and prefix using stack. Show the contents of stack at every stage :

Given expression : abcde∧∧* –

Soln. :

(i) Postfix to infix

Expression	Stack	Action
a b c d e ∧ ∧ * –	Empty	initial
b c d e ∧ ∧ * –	a	push
c d e ∧ ∧ * –	b a	push
d e ∧ ∧ * –	c b a	push
e ∧ ∧ * –	d c b a	push
∧ ∧ * –	e d c b a	push
∧ * –	(d^e) c b a	Convert
* –	c∧ (d∧e) b a	Convert
–	b*c∧ (d ∧e) a	convert

Techknowledge Publication

Expression	Stack	Action
End	a–b * c∧ (d ∧e)	Convert

Postfix expression : a – b * C ∧ (d ∧ e)

(ii) Postfix to prefix

Expression	Stack	Action
a b c d e ∧ ∧ * –	Empty	initial
b c d e ∧ ∧ * –	a	push
c d e ∧ ∧ * –	b a	push
d e ∧ ∧ * –	c b a	push
e ∧ ∧ * –	d c b a	push
∧ ∧ * –	e d c b a	push
∧ * –	∧de c b a	Convert
* –	∧c∧de b a	Convert
–	*b∧c∧de a	convert
End	– a*b ∧ c∧de	Convert

Prefix expression : – a * b∧ c ∧ d e

7.5.7 Conversion of Expression from Prefix to Infix

- Expression is scanned from right to left :
- Whenever an operator is found, it is put in between the two operands.

Example : * + a – bc / – d e + – fgh

* + a – bc / – d e + f – g h

* + a – bc / – d e f – g + h

* + a – bc / d – e f – g + h

* + a – bc (d – e) / (f – g + h)

* + a b – c (d – e) / (f – g + h)

* a + b – c (d – e) / (f – g + h)

(a + b – c) * (d – e) / (f – g + h)

Conversion with the help of a stack

- Prefix expression is scanned from right to left.
- When all operand is found, it is pushed onto the stack.
- When an operator is found, two sub-expressions are popped from the stack. These two sub-expressions and the operator being scanned are merged to create an infix sub-expression and this expression is pushed onto the stack.

Example : * + a – bc / – d e + – fgh

Prefix expression	Stack
* + a – bc / – d e + – f g h	
* + a – bc / – d e + – f g	h
* + a – bc / – d e + – f	g h

TechKnowledge Publications

Prefix expression	Stack
* + a – bc / – d e + –	f g h
* + a – bc / – d e +	f – g h
* + a – bc / – d e	f – g + h
* + a – bc / – d	e f – g + h
* + a – bc / –	d e f – g + h
* + a – bc /	d – e f – g + h
* + a – bc	(d–e) / (f – g + h)
* + a – b	c (d–e) / (f – g + h)
* + a –	b c (d–e) / (f – g + h)
* + a	b – c (d–e) / (f – g + h)
* +	a b – c (d–e) / (f – g + h)
*	a + b – c (d–e) / (f – g + h)
End	(a + b – c) * (d–e)/(f – g + h)

Algorithm for conversion from prefix into infix

```
S1, S2 : String ;
St : Stack of strings ;
initialize St ;
Prefix ← read the prefix expression ;
for( i ← index of last character of prefix, down to 0 )
{   X = prefix[i] ;
    if( X is an operand)
        Push(St, X);
    else
    {
        S1 = Pop(St) ;
        S2 = Pop(St) ;
        Push(St, "(" + S1 + X + S2 + ")" ) ;
        // + is a concatenation operator
    }
}   // Stack contains the final expression in infix form
```

Program 7.5.7 : Convert a prefix expression into infix form

```
/*Conversion of an expression from prefix to infix*/
#include<ctype.h>
#include<stdio.h>
#include<conio.h>
#define MAX 20
#include<string.h>
void convert_infix(char x);
char stack[MAX][MAX];
void prefix_to_infix(char prefix[ ], char infix[ ]);
int top;
/* Array 'stack[ ][ ]' is being used as a stack of strings
   */
void main()
{   char infix[30], prefix[30];
    clrscr();
    printf("\n Enter a prefix expression :");
    gets(prefix);
    prefix_to_infix(prefix, infix);
    printf("\n Infix :     %s", infix);
    getch();
}
```

Technknowledge Publications

```
void prefix_to_infix(char prefix[ ], char infix[ ])
{   char x, st1[20];
    int i;
    top = –1;
    for(i = strlen(prefix)–1; i> = 0; i – –)
    //scan the prefix string from right to left
    {
        x = prefix[i];
        if(isalnum(x))
        {
            /* convert token to string form */
            st1[0] = x;
            st1[1] = '\0';
            // push the operand on the stack s2
            top = top + 1;
            strcpy(stack[top], st1);
    }
        else  // if operator convert to infix
            convert_infix(x);
    }
    //Result is on top of the stack
    strcpy(infix, stack[top]);
}
void convert_infix(char x)
{   char st1[30], st2[10];
    st1[0] = '(';
    st1[1] = '\0';
    strcat(st1, stack[top]);
    st2[0] = x;
    st2[1] = '\0';
    strcat(st1, st2);
    strcat(st1, stack[top – 1]);
    st2[0] = ')';
    st2[1] = '\0';
    strcat(st1, st2);
    top = top – 1;
    strcpy(stack[top], st1);
}
```

7.5.8 Conversion of Expression from Prefix into Postfix

- Expression is scanned from right to left.
- Whenever an operator is found, it is put after the two operands.

Example : * + a – bc / – d e + – f g h

* + a – bc / – d e +	f g – h
* + a – bc / – d e	f g – h +
* + a – bc /	d e –	f g – h +
* + a – bc	d e – f g – h + /
* + a – bc	d e – f g – h + /
* a b c – +	d e – f g – h + /
a b c – + d e – f g – h + / *	

Conversion with the help of a stack

- Prefix expression is scanned from right to left. When an operand is found, it is pushed onto the stack
- When an operator is found, two sub-expressions are popped from the stack. These two sub-expressions and the operator being scanned are merged to create a postfix sub-expression and this expression is pushed onto the stack.

Prefix expression	Stack
* + a – bc / – de + – fgh	
* + a – bc / – de + – fg	h
* + a – bc / – de + – f	g h
* + a – bc / – de + –	f g h

Techknowledge Publication

Prefix expression	Stack
* + a – bc / – de +	fg – h
* + a – bc / – de	fg – h+
* + a – bc / – d	e fg – h+
* + a – bc / –	d e fg – h+
* + a – bc /	de – fg – h+
* + a – bc	de – fg – h+/
* + a – b	c de – fg – h+/
* + a –	b c de – fg – h+ /
* + a	bc– de – fg – h+ /
* +	a bc – de – fg – h + /
*	abc – + de – fg – h + /
End	abc – + de – fg – h + / *

Algorithm for Conversion from prefix to postfix

```
S1, S2 : string ;
St : stack of strings ;
initialize St,
prefix ← read the prefix expression ;
for(i ← index of the last character of prefix,
down to 0)
{
    X = prefix[i] ;
    if( X is an operand )
        push(St, X ) ;
    else
    {
        S1 = Pop(St) ;
        S2 = pop(St) ;
        push(St, S1 + S2 + X ) ;
        // + is a concatenation operator
    }
}
//stack contains the final expressions in postfix form
```

Program 7.5.8 : Convert a prefix expression into postfix form

```
/* Conversion of an expression from
prefix into postfix */
#include<ctype.h>
#include<stdio.h>
#include<conio.h>
#define MAX 20
#include<string.h>

void convert_postfix(char x);
char stack[MAX][MAX];
void prefix_to_postfix(char prefix[ ], char postfix[ ]);
int top;
/* Array 'stack[ ][ ]' is being used as a stack of strings
*/

void main()
{
    char postfix[30], infix[30], prefix[30];
    clrscr();
    printf("\n Enter a prefix expression :");
    gets(prefix);
    prefix_to_postfix(prefix, postfix);
```

Technknowledge Publication

```
    printf("\n Postfix :    %s", postfix);
    getch();
}
void prefix_to_postfix(char prefix[ ], char postfix[ ])
{
    char x, st1[20];
    int i;
    top = –1;
    for(i = strlen(prefix) –1; i> = 0; i – –)
    {   //scan the prefix string from right to left.
        x = prefix[i];
        if(isalnum(x))
         {
            /* convert token to string form */
            st1[0] = x;
            st1[1] = '\0';
            // push the operand on the stack s2
            top = top + 1;
            strcpy(stack[top], st1);
        }
        else  // if operator, convert to postfix
        {
            convert_postfix(x);
        }
       }
   //Result is on top of the stack
    strcpy(postfix, stack[top]);
}

void convert_postfix(char x)
{
    char st1[30], st2[30];
    st2[0] = x;
    st2[1] = '\0';
    strcpy(st1, stack[top]);
    strcat(st1, stack[top – 1]);
    strcat(st1, st2);
    top = top – 1;
    strcpy(stack[top], st1);
}
```

7.5.9 Fully Parenthesizing an Infix Expression

In a fully parenthesized expression, every operator has a corresponding opening and closing parenthesis. A fully parenthesized expression is free from precedence.

An infix expression can be fully parenthesized by using the approach used in conversion of an infix expression into postfix form. When, we have to fully parenthesize an expression :

We maintain two stacks

(a) Operator-stack : To handle operator precedence

(b) Output-stack : To store the parenthesized sub-expressions.

Algorithm for fully parenthesizing an expression

```
S1 :    Stack of characters /* for storing operators */
S2 :    Stack of strings /* output-stack */
St1, St2, St3 : string
while(more tokens)
{
    X ← nexttokens() ;
    if( x is an operand )
        push( S2, X );
    else
    {
        if( x is an opening bracket )
            push(S1, '(') ;
        else
            if( x is a closing bracket )
            while(( x = pop(s1) ) ! = '(' )
            {
                St2 = pop(S2)
                St1 = pop(S2)
                St3 = '(' + St1 + x + St2 + ')';
                Push(S2, St3) ;
                // + is a concatenation operator
            }
            else
            {   // x is an operator
        while(procedence(x)< = procedence(top(S1))
```

Techknowledge Publication

```
        {
            y = pop(S1) ;
            St2 = pop(S2) ;
            St1 = pop(S2) ;
            St3 = '(' + St1 + y + St2 + ')' ;
            push(S2, St3) ;
        }
        push(S1, x) ;
        }
        }
```

```
        }
        while(! empty(S1) )
        {
            x = pop(S1) ;
            St2 = pop(S2) ;
            St1 = pop(S1) ;
            St3 = '(' + St1 + X + St2 + ')'
            push(S2, St3) ;
        }
        // stack S2 contains the final result.
```

Ex. 7.5.16 : Show all steps of conversion from infix to its corresponding fully parenthesized form for the following expression.

a ∧ b / c * d + e – f

Stack of operators	Input	Stack of output
	a ∧ b / c * d + e – f	
	∧ b / c * d + e – f	a
∧	b / c * d + e – f	a
∧	/ c * d + e – f	b a
/	c * d + e – f	(a ∧ b)
/	* d + e – f	c (a ∧ b)
*	d + e – f	(a ∧ b) / c)
*	+ e – f	d ((a ∧ b) / c)
+	e – f	(((a ∧ b) / c) * d)
+	– f	e (((a ∧b) / c) * d)
–	f	((((a ∧ b) / c) * d) + e)
–	End	f ((((a ∧ b) / c) * d) + e)
	End	(((((a ∧ b) / c) * d) + e) – f)

Techknowledge Publications

Program 7.5.9 : Fully parenthesising an infix expression

```
/* Program for conversion of an infix expression to its
   fully parenthesizes form
   operators supported '+, –, *, /, %, ∧, (, )
   operands supported --all single character operands
*/
#include<stdio.h>
#include<conio.h>
#include<ctype.h>
#define MAX 50
typedef struct stack
{
    int data[MAX];
    int top;
}stack;
int stack1[10][30]; // a stack of strings
int top1;
int precedence(char);
void init(stack *);
int empty(stack *);
int full(stack *);
char  pop(stack *);
void push(stack *, char );
char  top(stack *); //value of the top element
void parenthesize(char infix[ ], char final[ ]);
void putbrackets(char x)
{
    int l;
    char st1[40];
    st1[0] = '(';
    st1[1] = '\0';
    strcat(st1, stack1[top1 – 1]);
    l = strlen(st1);
    st1[l] = x;
    st1[l+1] = '\0';
    strcat(st1, stack1[top1]);
    l = strlen(st1);
    st1[l] = ')';
    st1[l+1] = '\0';
    top1 = top1 – 1;
    strcpy(stack1[top1], st1);
}
void main()
{
    char infix[30], final[30];
    clrscr();
    printf("\n Enter an infix expression : ");
    gets(infix);
    parenthesize(infix, final);
    printf("\n Result  : %s  ", final);
    getch();
}
void parenthesize(char infix[ ], char final[ ])
{
    stack s;
    char x;
    int i;
    char token;
    init(&s);
    top1 = –1;
    for(i = 0; infix[i]! = '\0'; i++)
    {
        token = infix[i];
        if(isalnum(token))
        {
            top1++;
            stack1[top1][0] = token;
            stack1[top1][1] = '\0';
        }
        else
            if(token == '(')
                push(&s, '(');
            else
                if(token == ')')
                 while((x = pop(&s))! = '(')
```

Techknowledge Publication

```
                putbrackets(x);
            else
            {
while(precedence(token)< = precedence(top(&s))
&& !empty(&s))
            {
                x = pop(&s);
                putbrackets(x);
            }
                push(&s, token);
            }
    }
    while(!empty(&s))
    {
        x = pop(&s);
        putbrackets(x);
    }
strcpy(final, stack1[top1]);
}
int precedence(char x)
{
    if(x == '(')
        return(0);
    if(x == '+' || x == '-')
        return(1);
    if(x == '*' || x == '/' || x == '%')
        return(2);
    return(3);
}
void init(stack *s)
{
    s->top = –1;
}
int empty(stack *s)
{
    if(s–>top == –1)
        return(1);
    return(0);
}

int full(stack *s)
{
    if(s–>top == MAX–1)
        return(1);
    return(0);
}

void push(stack *s, char x)
{
    s–>top = s–>top + 1;
    s–>data[s–>top] = x;
}
char pop(stack *s)
{
    int x;
    x = s–>data[s–>top];
    s–>top = s->top – 1;
    return(x);
}
char top(stack * p)
{
    return(p–>data[p–>top]);
}
```

7.5.10 Conversion of a Fully Parenthesized Infix Expression into Postfix

A fully parenthesized infix expression can easily be converted into postfix form by moving every operator to replace its corresponding closing bracket.

Ex. 7.5.17 : Convert the following expression to postfix form:

$$(((a \wedge b) * c) + (d * (e / f)))$$

Soln. : Each operator is moved to replace its corresponding right parenthesis. It is shown below :

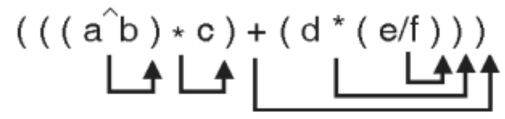


Fig. Ex. 7.5.17

Postfix equivalence : $a\ b \wedge c * d\ e\ f / * +$

Techknowledge Publications

Algorithm

- A fully parenthesized expression can be converted into a postfix form with the help of a stack.
- Expression is scanned from left to right.
- Every operand is sent to the output.
- Every operator is pushed onto the stack.
- If the character being scanned is a closing bracket then an operator is popped from the stack and is sent to output.

Input	Stack	Output
(((a ∧ b) * c) + (d * e/f)))		–
((a ∧ b) * c) + (d * e/f)))		–
(a ∧ b) * c) + (d * e/f)))		–
a ∧ b) * c) + (d * e/f)))		–
∧ b) * c) + (d * e/f)))		a
b) * c) + (d * e/f)))	∧	a
) * c) + (d * e/f)))	∧	a
* c) + (d * e/f)))		ab
c) + (d * e/f)))	*	ab∧
) + (d * e/f)))	*	ab∧
+ (d * e/f)))		ab∧c*
(d * e/f)))	+	ab∧c *
d * e/f)))	+	ab∧c *
* e/f)))	+	ab∧c *d
e/f)))	* +	ab∧c *d
/f)))	* +	ab^c * de
f)))	/ * +	ab∧c * de
)))	/ * +	ab∧c * def
))	* +	ab∧c * def /
)	+	ab∧c * def / *
End		ab∧c * def / *+

Program 7.5.10 : Conversation of a fully parenthesized expression into postfix form.

```
/*      program for conversion of a fully parenthesized
        expression to postfix form operators supported
        '+, -, *, /, %, ∧, (, )
        operands supported -- all single character
        operands
*/
#include<stdio.h>
#include<conio.h>
#include<ctype.h>
#define MAX 50
```

TechKnowledge Publications

```
typedef struct stack
{
    int data[MAX];
    int top;
}stack;

void init(stack *);
int empty(stack *);
int full(stack *);
char  pop(stack *);
void push(stack *, char );
char  top(stack *); //value of the top element
void infix_to_postfix(char infix[ ], char postfix[ ]);
void main()
{
    char infix[30], postfix[30];
    clrscr();
    printf("\n  Enter  a  fully  parenthesized  infix
    expression:");
    gets(infix);
    infix_to_postfix(infix, postfix);
    printf("\n Result  : %s  ", postfix);
    getch();
}
void infix_to_postfix(char infix[ ], char postfix[ ])
{
    stack s;
    char x;
    int i, j;
    char token;
    init(&s);
    j = 0;
    for(i = 0; infix[i]! = '\0'; i++)
    {
        token = infix[i];
        if(isalnum(token))
        postfix[j++] = token;
        else
        if(token == ')')
            postfix[j++] = pop(&s);
        else
            if(token ! = '(')
                push(&s, token);
    }
    postfix[j] = '\0';
}
void init(stack *s)
{
    s->top = –1;
}
int empty(stack *s)
{
    if(s->top == –1)
    return(1);
    return(0);
}

int full(stack *s)
{
    if(s->top == MAX–1)
        return(1);
    return(0);
}
void push(stack *s, char x)
{
    s->top = s->top + 1;
    s->data[s->top] = x;
}

char pop(stack *s)
{
    int x;
    x = s–>data[s->top];
    s–>top = s->top –1;
    return(x);
}
char top(stack * p)
{
    return(p->data[p->top]);
}
```

Techknowledge Publications

7.5.11 Conversion of a Fully Parenthesized Infix Expression into Prefix Form

A fully parenthesized infix expression can easily be converted into prefix form by moving every operator to replace its corresponding opening bracket.

Ex. 7.5.18 : Convert the following expression to prefix form

(((a ∧ b) * c) + (d * (e / f)))

Soln. : Each operator is moved to replace its corresponding left parenthesis. It is shown below.

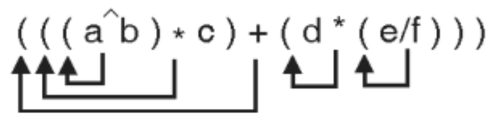


Fig. Ex. 7.5.18

Prefix equivalence : + * ∧ abc * d / e f

Algorithm

- A fully parenthesized expression can be converted into a prefix form with the help of two stacks
  - a) Stack to remember operators
  - b) Stack to reverse the output.
- Expression is scanned right to left.
- Every operand is sent to the output-stack
- Every operator is pushed onto the operator-stack
- If the character being scanned is an opening bracket then an operator is popped and sent to the output stack.
- Finally, the output-stack is emptied to get the prefix expression.

Input	Operator stack	Output stack
(((a ∧ b) * c) + d)		
(((a ∧ b) * c) + d		
(((a ∧ b) * c) +		d
(((a ∧ b) * c	+	d
(((a ∧ b) *	+	c d
(((a ∧ b)	* +	c d
(((a ∧ b	* +	c d
(((a ∧	* +	b c d
(((a	∧ * +	b c d
(((	∧ * +	a b c d
((	* +	∧ a b c d
(	+	* ∧ a b c d
End		+ * ∧ a b c d

Prefix expression : + * ∧ abcd

Technknowledge Publication

Program 7.5.11 : Conversion of a fully parenthesized expression into prefix form.

```
/*      Program for conversion of a fully parenthesized
        expression  into prefix form operators supported
        '+, -, *, /, %, ∧, (, )
        operands supported -- all single character
        operands
*/

#include<stdio.h>
#include<conio.h>
#include<ctype.h>
#define MAX 50

typedef struct stack
{
    int data[MAX];
    int top;
}stack;

void init(stack *);
int empty(stack *);
int full(stack *);
char  pop(stack *);
void push(stack *, char );
char  top(stack *);          //value of the top element
void infix_to_prefix(char infix[ ], char prefix[ ]);

void main()
{
    char infix[30], prefix[30];
    clrscr();
    printf("\n Enter a fully parenthesized infix expression:");
    gets(infix);
    infix_to_prefix(infix, prefix);
    printf("\n Result  : %s  ", prefix);
    getch();
}

void infix_to_prefix(char infix[ ], char prefix[ ])
{
    stack s, s1;
    char x;
    int i, j;
    char token;
    init(&s); init(&s1);
    j = 0;

    for(i = strlen(infix)–1; i> = 0; i– –)
    {
        token = infix[i];
        if(isalnum(token))
        push(&s1, token);
        else
        if(token == '(')
        {
            x = pop(&s);
              push(&s1, x);
        }
        else
            if(token ! = ')')
                push(&s, token);
    }
    while(!empty(&s1))
        prefix[j++] = pop(&s1);
        prefix[j] = '\0';
}

void init(stack *s)
{
    s->top = –1;
}

int empty(stack *s)
{
    if(s->top == –1)
    return(1);
    return(0);
}

int full(stack *s)
```

Techknowledge Publication

```
{
    if(s->top == MAX-1)
    return(1);
    return(0);
}

void push(stack *s, char x)
{
    s->top = s->top + 1;
    s->data[s->top] = x;
}

char pop(stack *s)
{
    int x;
    x = s->data[s->top];
    s->top = s->top - 1;
    return(x);
}

char top(stack * p)
{
    return(p->data[p->top]);
}
```

Ex. 7.5.19 : Convert the following expression into postfix expression.

(i) ((a + b) – (c * d)) / (e / (f ∧ 2))

(ii) (a + b) * d + e / (f + a * d) + c

Soln. :

(i) Given expression :

((a + b) – (c ∗ d)) /(e/ (f ∧ 2))

Fully parenthesizing the expression and moving the operator to its corresponding right parenthesis.

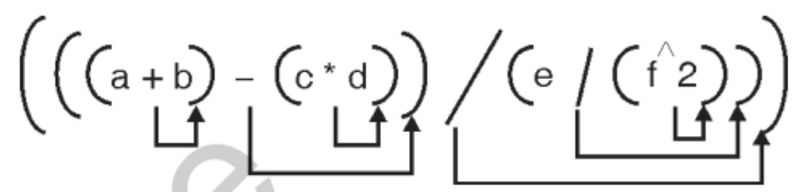


Fig. Ex. 7.5.19(a)

Postfix expression = ab + cd ∗ – e f 2 ∧ //

(ii) (a + b) ∗ d + e/ (f + a + d) + c

Fully parenthesizing the expression and moving the operator to its corresponding right parenthesis.

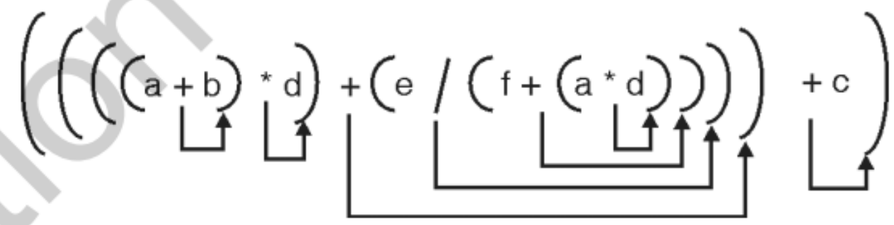


Fig. Ex. 7.5.19(b)

Postfix expression = ab + d ∗ e f a d ∗ + / + c +

Ex. 7.5.20 : Convert the following infix expression into postfix expression by showing contents of stack for every iteration.

(((A / (B ∧ C)) + (D & E)) – (A & C))

Evaluate the postfix expression obtained by you using stack for A = 27, B = 3, C = 2, D = 3, E = 17.

Soln. : A fully parenthesized expression can be converted into a postfix form with the help of a stack.

Expression is scanned from left to right.

Every operand is sent to two output.

Every operator is pushed onto the stack.

If the character being scanned is a closing bracket then an operator is popped from the stack and is sent to output.

Input	Stack	Output
(((A / (B ∧ C)) + (D & E)) – (A & C))	Empty	–
((A / (B ∧ C)) + (D & E)) – (A & C))	Empty	–
(A / (B ∧ C)) + (D & E)) – (A & C))	Empty	–

TechKnowledge Publications

Input	Stack	Output
A/ (B ∧ C)) + (D & E)) – (A & C))	Empty	–
/ (B ∧ C)) + (D & E)) – (A & C))	Empty	A
(B ∧ C)) + (D & E)) – (A & C))	/	A
B ∧ C)) + (D & E)) – (A & C))	/	A
∧ C)) + (D & E)) – (A & C))	/	AB
C)) + (D & E)) – (A & C))	/∧	AB
)) + (D & E)) – (A & C))	/∧	ABC
) + (D & E)) – (A & C))	/	ABC∧
+ (D & E)) – (A & C))	Empty	ABC∧/
(D & E)) – (A & C))	+	ABC∧/
D & E)) – (A & C))	+	ABC∧/
& E)) – (A & C))	+	ABC∧/ D
E)) – (A & C))	+&	ABC∧/ D
)) – (A & C))	+&	ABC∧/ DE
) – (A & C))	+	ABC∧/ DE&
– (A & C))	Empty	ABC∧/ DE& +
(A & C))	–	ABC∧/ DE& +
A & C))	–	ABC∧/ DE& +
& C))	–	ABC∧/ DE& + A
C))	–&	ABC∧/ DE& + A
))	–&	ABC∧/ DE& + AC
)	–	ABC∧/ DE& + AC &
End	Empty	ABC∧/ DE& + AC & –

Postfix expression : ABC∧/ DE& + AC & –

Techknowledge Publication

Evaluation of the expression

Expression : ABC∧/ DE& + AC & –

A = 27, B = 3, C = 2, D = 3, E = 17

Expression	Stack	
ABC^/ DE& + AC & –		
BC∧/ DE& + AC & –	27	
C∧/ DE& + AC & –	3 27	
∧/ DE& + AC & –	2 3 27	
/ DE& + AC & –	9 27	$3^2 = 9$
DE& + AC & –	3	27/9 = 3
E& + AC & –	3 3	
& + AC & –	17 3 3	
+ AC & –	1 3	3 = 000011 17 = 010001 & 000001
AC & –	4	3 + 1 = 4
C & –	27 4	
& –	2 27 4	
–	2 4	27 = 11011 2 = 00010 & 00010
End	2	4 – 2 = 2

Output = 2

Ex. 7.5.21 : Convert the following prefix expression into postfix expression. * + a – bc /–de + – fgh.

SPPU - May 17, 3 Marks

Soln. :

* + a – bc / – de + – fgh

* + a [bc –] / [de –] + [fg –] h

* [abc – +] / [de –] [fg – h +]

* [abc – +] [de – fg – h + /]

abc – + de – fg – h + / *

7.5.12 Well Formedness of Parenthesis

An expression is said to be well formed if every opening bracket has a corresponding closing bracket and there is no extra bracket.

Examples of well formed Expressions :

(() ()) ((())) ((()) ()) ( )

7.5.12(A) Algorithm to Read in aParenthesized Infix Expression and Check Well-formedness of Parenthesis

(i) Check whether parentheses match

s : stack of characters

x : character type

Step 1 : x ← read the next token

Step 2 : if(x == '(')

Push(s,x);

Step 3 : if(x == ')')

if(top element of the stack is '(')

pop(s);

else

Print "mismatch"

Step 4 : if(more tokens)

Techknowledge Publications

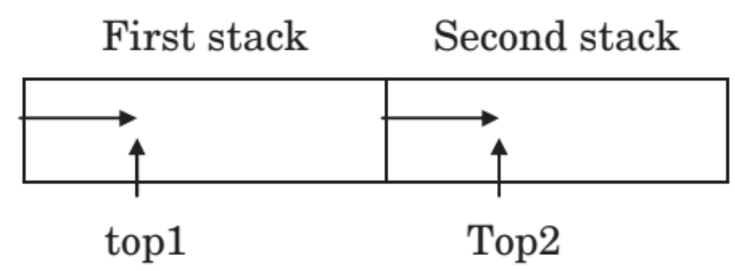
```
        goto step1
Step 5 : if(stack is not empty)
            Print "mismatch"
        else
            Print "match"
Step 6 : stop
```

Program 7.5.12 : Program for checking well formedness of parenthesis.

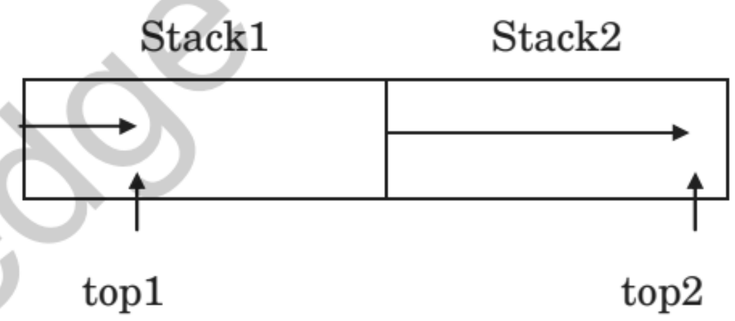
```
void main()
{
    char x;  stack s;
    init(&s);
    printf("\n enter an parenthesized expression:");

    while((x = getchar( ))! = '\n ')
    {
        switch(x)
        {
            case '(' : push(&s, '('); break;
            case ')' : if(! empty(&s))
                            pop(&s);
                       else
                       {
                            printf("\n mismatch ...");
                            exit(0);
                       }
                       break;
            default : break;
        }

    if(! Empty(&s))
        printf("\n mismatch:");
    else
        printf("\n well formed:");
    }
}
```

7.6 Representation of Two Stacks in an Array

First representation :

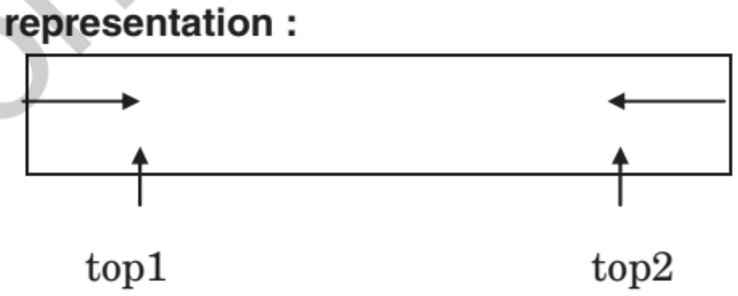


Array is divided into two equal parts. First part is reserved for stack 1 and the second part is reserved for stack 2. In such a division if a stack becomes full, it cannot utilize the empty space of the other stack.



Stack 2 is full and it cannot utilize the empty space of stack 1.

Second representation :



First stack starts from the beginning whereas the second stack starts from the end. These two stacks grow in opposite direction. Empty space between two stacks can be utilized by either of the stacks. This form of representation minimizes the changes of overflow and hence it is superior over the first representation.

Stacks can be manipulated using the following function

(1) Initialize( ) (2) Push( )

(3) POP( ) (4) Empty( )

(5) Full( )

'C' function for implementation of stack

```
# define MAX 50
typedef struct Dstack
{
```

Techknowledge Publication

```
    int top1;
    int top2;
    int data[MAX] ;
}Dstack ;

void initialize(Dstack *P)
{
    P→top1 = –1;
    P→top2 = MAX ;
}

int empty(Dstack *p, int stackno)
{
    if( stackno == 1 && P →top1 == –1)
        return(1);
    if(stackno == 2 && P→top2 == MAX)
        return(1) ;
    return(0) ;
}

int full(Dstack *p)
{
    if(p →top1 + 1 == p →top2)
        return(1);
    return(0);
}

void push(Dstack *P, int stackno, int x)
{
  if(stackno == 1)
  {         P → top1 = P → top1 + 1;
        P → data[P → top1] = x;
    }
    else
    {   P → top2 = P → top2 – 1;
        P → data[P → top2] = x ;
    }
}

int POP(Dstack *P, int stackno)
{   int x ;
    if(stackno == 1)
    {   x = P→data[P→top1] ;
        P→top1 = P→ top1 –1 ;
        return(x) ;
    }
        x = P→ data[P→top2] ;
        P→ top2 = P→top2 +1 ;
    return(x) ;
}
```

7.7 Representation of Multiple Stacks in an Array

Several stacks can be stored in a single array. Suppose A[ ] is an array of size MAX and N number of stacks are to be stored in the array. MAX is a predefined constant.

We can divide the available memory A[0…MAX–1] into N segments. Each stack can be allocated one of these segments. For each stack I, as shown in the Fig. 7.7.1. T[i] and B[i] represent the indices of topmost and bottommost element of the stack i.

Initial values of T[i] and B[i] is calculated as given below

```
for(i = 0 ; i < N ; i ++)
{
    T[i] = B[i] = (MAX/N) * i ;
}
```

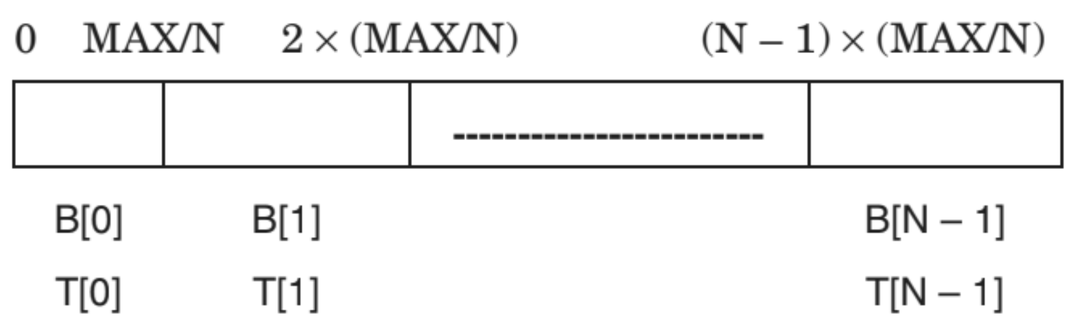


Fig. 7.7.1. : N-stacks in an array of size of MAX Initial values of B[i] and T[i]

push( ) and pop( ) can be performed on any of the stacks. Both push( ) and pop( ) operation have an additional parameter, stack number.

Tech Knowledge Publications

'C' function for push and pop.

```
void push(int stack_no,int x)
{
   int top;
   int i; i = stack_no;
   if(T[i] == B[i+1])
       printf("\n Overflow");
   else
   {
       T[i] = T[i]+1;
       top = T[i];
       A[top] = x;
   }
}
int pop(int stack_no)
{
   int top,x;
   int i;
   i = stack_no.;
   if(T[i] == B[i])
   {
       printf("\n Underflow");
       return(–1);
   }
   else
   {
       top = T[i];
       x = A[top];
       T[i] = T[i]–1;
   }
   return(x);
}
int empty(int stack_no)
{
   int i;
   i = stack_no;
   if (T[i] == B[i]
       return (1);
   return(0);
}
int full(int stack_no)
{
   int i;
   i = stack_no;
   if (T[i] == B[i + 1])
       return (1)
   return (0);
}
```

'Stack Full' condition for stack i does not imply that there is no free space in the array A[ ] . In fact, there could be lot of unused space between stack j and stack j + 1. We could design an algorithm that will permit push( ) operation in the stack i to use unused space of array A.

Algorithm for creating space between stack i and stack i + 1.

(a) Determine the least j, i < j < n such that there is free space between stack j and stack j + 1. Now move all stacks starting from stack number i + 1 to j, right by one place. This will create a space between stack i and stack i + 1.

(b) If there is no such j in (a) then find the largest j such that 0 ≤ j < i and there is space between stack j and stack j + 1 ; Now move all stack starting from stack number j + 1 to stack no i, left by one place. This will create a free space between stack i and stack i + 1.

(c) If there is no such j in (b), then all MAX spaces in A[ ] have been utilized and there is no free space.

7.8 Recursion

7.8.1 Introduction

Q. What is recursion. **SPPU - Dec. 17, 2 Marks**

Q. Define Recursion **SPPU - May 19, 1 Mark**

Q. What is recursion? Explain use of the stack for "Undo operation in a word processor (MS office/open office)". **SPPU - Dec. 19, 3 Marks**

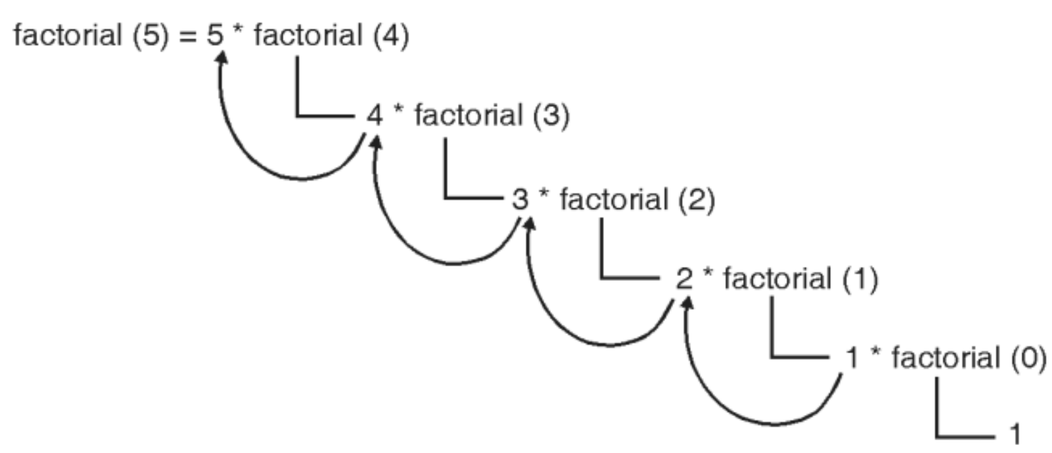
Recursion is a fundamental concept in mathematics. When a function is defined in terms of itself then it is called a recursive function. Consider the definition of factorial of a positive integer n.

Tech Knowledge Publications

$$\text{Factorial}(n) = \begin{cases} 1 & \text{if } (n=0) \\ n * \text{factorial}(n-1), & \text{otherwise} \end{cases}$$

Function "factorial ( )" is defined in terms of itself for n > 0. Value of the function at n = 0 is 1 and it is called the base. Recursion terminates on reaching the base.

Example :



Recursion expands when n > 0 and it starts winding up on hitting the base (n = 0).

Undo operation

Stack can be used to remember the details of modification operations. These details can be used to reverse all modification operations during the current session of edit.

7.9 Converting a Recursive Function to an Equivalent C-Function

7.9.1 Finding Factorial of an Integer Number

Recursive Definition of Factorial of an Integer

$$f(n) = 1 \quad \text{if } (n == 0)$$
$$n * f(n-1), \quad \text{otherwise}$$

'C' function for finding factorial

```
int factorial(int n)
{
    if(n == 0)
        return(1);

    return(n * factorial(n – 1));
}
```

Sequence of calls to be made has been shown in the Fig. 7.9.1 to give a better insight into recursion.

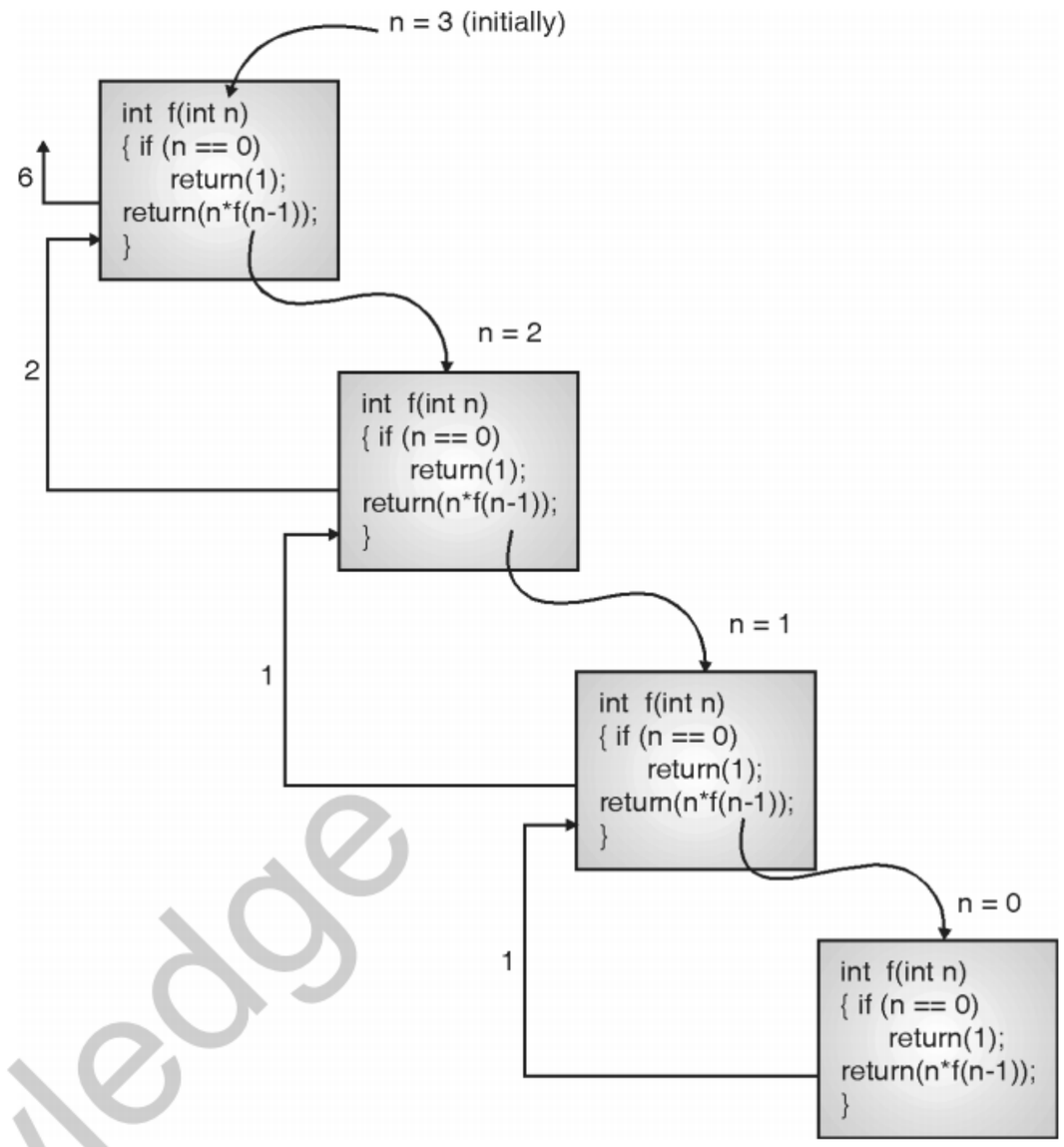


Fig. 7.9.1

Sequence of calls could also be shown through a recursion tree.

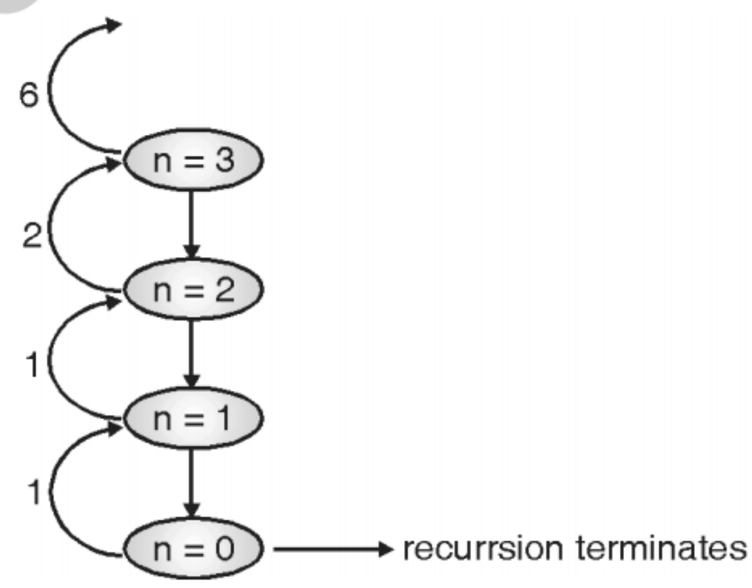


Fig. 7.9.2 : Recursion tree for finding factorial (3)

Finding n^{th} term of Fibonacci Sequence

Recursive definition

$$T_n = T_{n-1} + T_{n-2} \quad \text{if } n > 1$$
$$T_1 = 1$$
$$T_0 = 0$$

Tech Knowledge Publications

'C' function for finding n^{th} term of a Fibonacci sequence.

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

In the above example $f_{ib}(n) = fib(n-1) + fib(n-2)$ for $n > 1$ is a case of binary recursion. The first term on the right fib (n – 1) is known as left recursion and the second term on the right fib(n – 2) is known as right recursion.

Fibonacci series

0	1	1	2	3
|	|	|	|	|
T_0	T_1	T_2	T_3	T_4

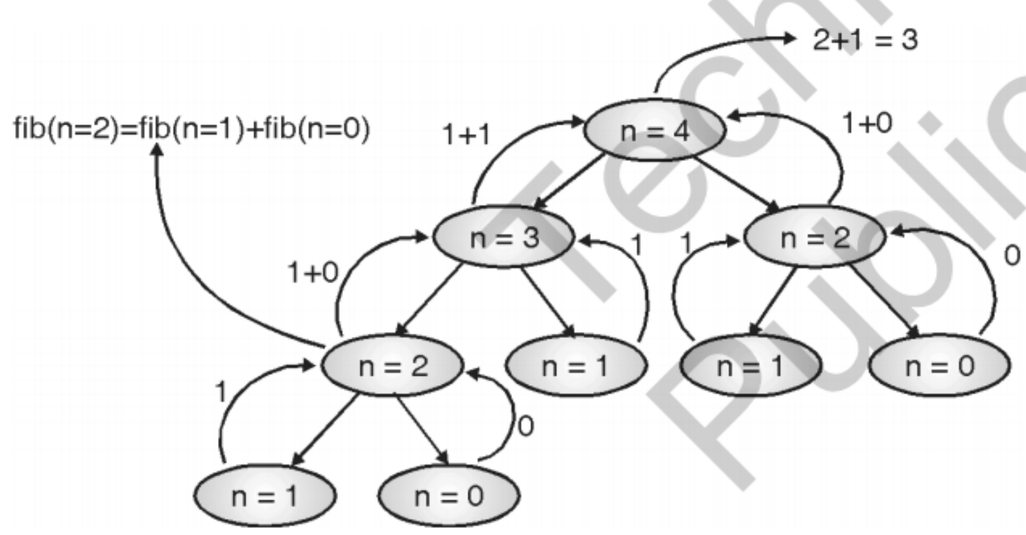


Fig. 7.9.3 : A recursion tree, showing calculation of T_4

Program 7.9.1 : Program for finding factorial of a number through the use of recursive function.

```
#include<stdio.h>
#include<conio.h>
int fact(int);

void main()
{
    int x, n;
    printf("\n Enter the value of n :");
    scanf("%d", &n);
    x = fact(n);
    printf("\n %d", x);
    getch();
}

int fact(int n)
{
    if(n == 0)
        return(1);

    return(n*fact(n–1));
}
```

Output

```
Enter the value of n :5
120
```

Program 7.9.2 : Write a non-recursive algorithm to find factorial of a positive number.

Step 1 : Fact = 1 //for storing the result
Step 2 : i = 1 //for looping
Step 3 : Read n //for finding $\lfloor n$
Step 4 : if(i > n) then goto Step 8
Step 5 : Fact = Fact * i
Step 6 : i = i + 1
Step 7 : goto Step 4
Step 8 : Print Fact
Step 9 : Stop
Step 10 : End.

Program 7.9.3 : Write recursive function to add digits of a given positive integer.

```
int adddigit(int x)
{
    int sum = 0;
    if(x == 0)
        return(0);
    sum = sum + x %10;
    x = x/10;
    return(sum + adddigit(x));
}
```

Techknowledge Publication

7.10 Examples of Recursion

7.10.1 Finding Sum of the Elements Stored in an Array

sum (a, n) = a [n – 1] + sum (a, n – 1) ... if n > 1

= a [0] ..... if n = 1

a → Array

n → Number of elements

sum (a [5, 9, 2, 6], 4) = 6 + sum (a [5, 9, 2], 3)

– Definition of finding sum of array elements is declared in terms of sum itself.

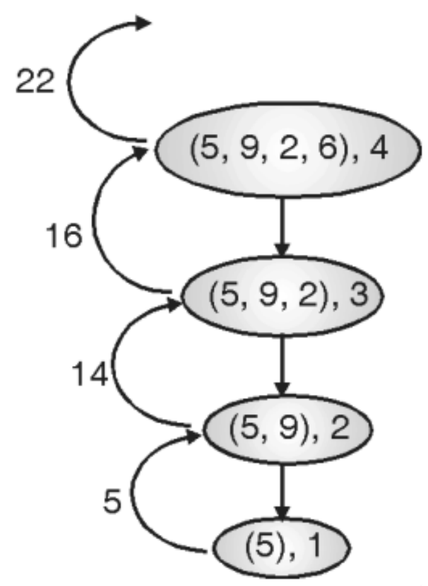


Fig. 7.10.1 : Recursion tree showing calculation of sum of 4 elements 5, 9, 2, 6

– On the right hand side of the recursive definition sum (a, n) = a (n – 1) + sum (a, n – 1), the value of n reduces by 1. Thus it has moved closer to the point of termination (n = 1) and the recursive will definitely terminate.

– Every recursion should have at least one case for termination. Recursion in the above example terminates when n reaches 1. Sum of the elements of an array having one element will be the element itself.

'C' function for finding sum of the elements of an array.

```
int sum( int a[ ], int n)
{
    if(n == 1)
        return( a [0]);
    else
        return(a [n – 1] + sum(a, n – 1));
}
```

7.10.2 Finding Length of a String

length(s) = 0 if s[0] = '\0'

1 + length(s + 1) otherwise

Example : length("xyz")

= 1 + length("yz")

= 1 + 1 + length("y")

= 2 + 1 + 1 + length("") = 3

'C' function for finding length of a string.

```
int length(char s[ ])
{
    if(s[0] == '\0' )
        return(0);
    return(1 + length(s + 1));
}
```

7.10.3 Reversing a String

Reverse (s, i, j) = (interchange (s[i], s[j])

reverse (s, i + 1, j – 1))

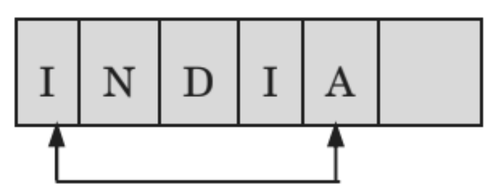
if i < j

s = Character array

i = index of the first character

j = index of the last character

Example



interchange the characters 'I' and 'A' and then

reverse the string "NDI".

'C' function for reversing a string.

```
void reverse(char *s, int i, int j)
{
    char temp;
    if(i < j)
    {
        temp = s [i];
```

TechKnowledge Publication

```
        s [i] = s [j];
        s [j] = temp;
        reverse(s, i + 1, j – 1);
    }
}
```

7.10.4 Searching a Number in an Array

a → array

i → index of the starting element

j → index of the last element

key → element to be searched

search (a, i, j, key) = – 1 ..... if i > j

i..... if (a[i] == x)

Search (a, i + 1, j, key) otherwise

A return value of –1 indicates that the element is not found otherwise it gives the index of the location of occurrence of the search element.

'C' function for searching an element

```
int search(int * a, int i, int j, int key)
{
    if(i > j)
        return(–1);

    if(a [i] == key)
        return(i);

    return(search(a, i + 1, j, key));
}
```

7.10.5 Finding Largest Element in an Array

largest (a, i, j) = a [i] ..... if (i = j)

max (a [i], largest (a, i + 1, j)) ..... otherwise

'C' function for finding the largest element of an array

```
int largest(int * a, int i, int j)
{
    if(i == j)
        return(a[i]);
    if(a [i] > largest(a, i + 1, j))
        return(a [i]);

    return(largest(a, i +1, j));
}
```

7.10.6 Binary Search

binsearch (a, i, j, key) = – 1 .... if (i > j)

$\frac{i+j}{2}$ ..... Key = a $\left[\frac{i+j}{2}\right]$

binsearch (a, i, $\frac{i+j}{2}$ – 1, key)... Key < a $\left[\frac{i+j}{2}\right]$

binsearch (a, $\frac{i+j}{2}$ + 1, j, key) ... Key > a $\left[\frac{i+j}{2}\right]$

'C' function for binary search

```
int binsearch(int *a, int i, int j, int key)
{
    int k;
    k = (i + j)/2;

    if(i > j)
        return(– 1);

    if(key < a [k])
        return(binsearch(a, i, k – 1, key);

    return(binsearch(a, k + 1, j, key);
}
```

7.10.7 Tower of Hanoi Problem

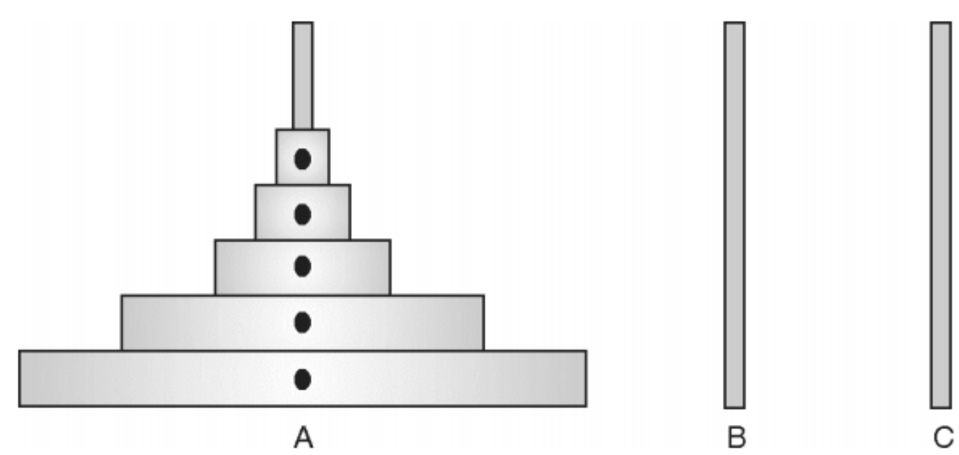


Fig. 7.10.2 : Disks are stacked on peg A in decreasing order of size

Techknowledge Publication

A tower of n disks is stacked in decreasing order on peg "A". These disks are to be transferred to peg "B" with the help of peg "C". Following rules must be followed while transferring disks from one peg to another peg.

(a) Only one disk can be transferred at a time.

(b) At no time a larger disk can be placed on a smaller disk.

In order to define the problem of Tower of Hanoi recursively, we must express its solution in terms of a problem of n – 1 disks.

n-disks problem

Move n disks from peg A to peg B using peg C.

Step 1 : Move n – 1 disks from peg A to peg C.

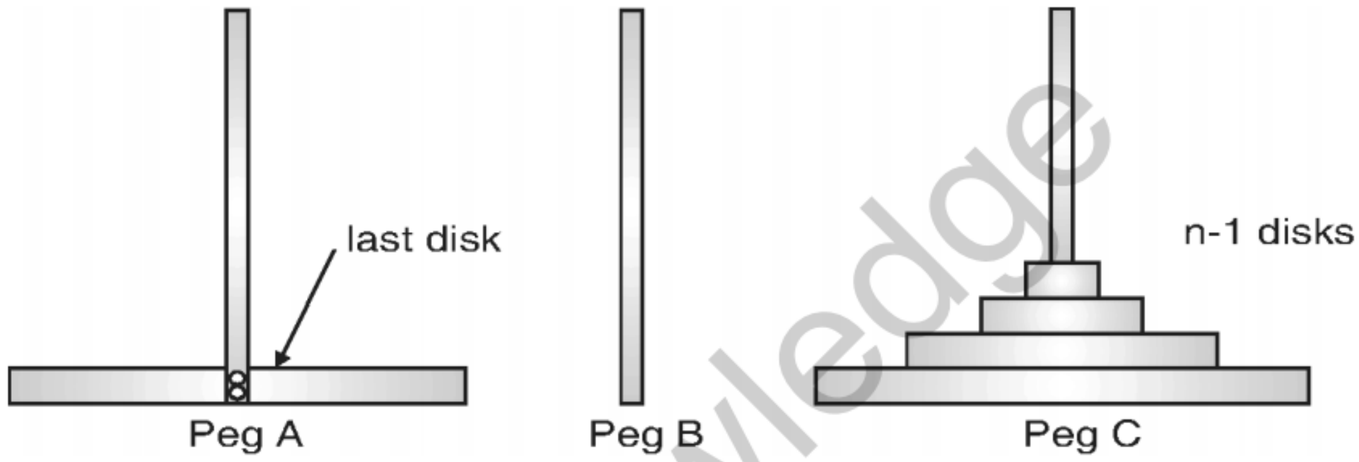


Fig. 7.10.2(a)

Step 2 : Move the only disk from peg A to peg B.

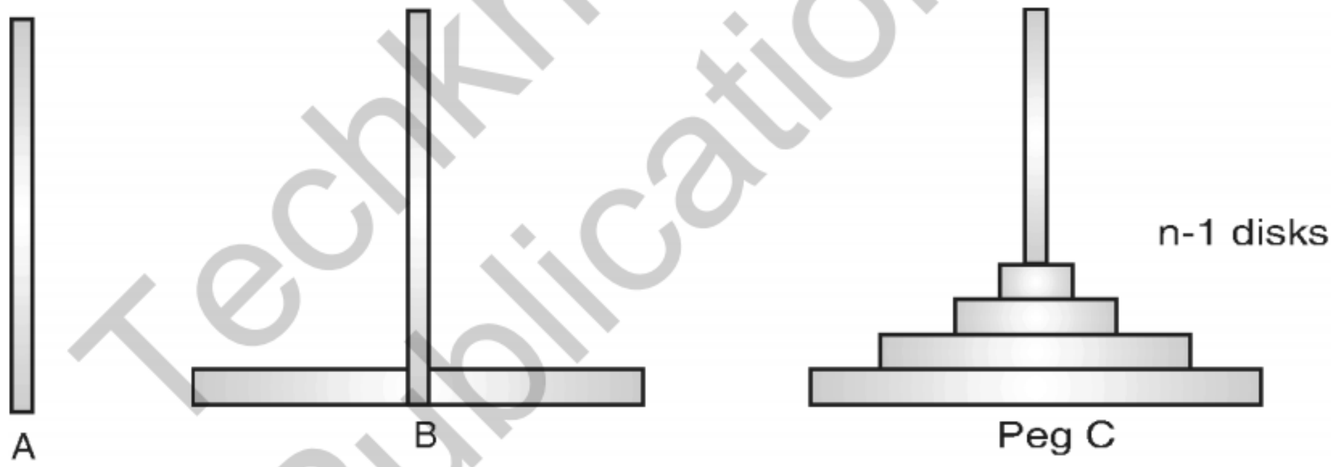


Fig. 7.10.2(b)

Step 3 : Move the n – 1 disks from peg C to peg B.

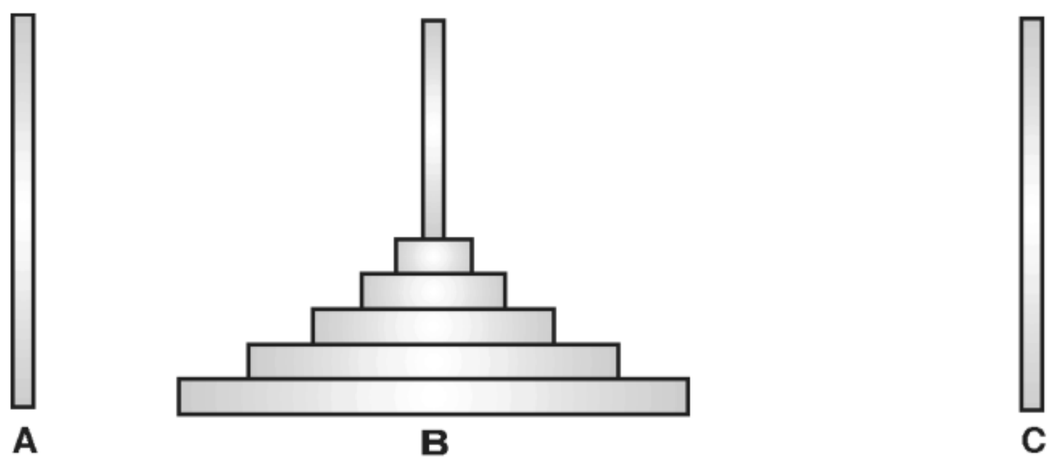


Fig. 7.10.2(c)

Recursive definition

$$\text{TOH}(n, x, y, z) = \left\{ \begin{array}{l} \text{TOH}(n-1, x, z, y) \\ \text{move a disk from x to y} \\ \text{TOH}(n-1, z, y, x) \end{array} \right\} \text{ if } n \geq 1$$

Techknowledge Publications

n = Number of disks

x = Source peg

y = Destination peg

x = Empty peg to be used for transfer

TOH (n, x, y, z) = transfer n plates from peg x to peg y using peg z.

TOH (n – 1, x, z, y) = transfer n – 1 plates from peg x to peg z using peg y.

TOH (n – 1, z, y, x) = transfer n – 1 plates from z to y using peg x.

Program 7.10.1 : Program for Tower of Hanoi.

```
#include<stdio.h>
#include<conio.h>
void TOH(int n, char x, char y, char z);
void main()
{
    int n;
    printf("\n Enter number of plates:");
    scanf("%d", &n);
    TOH(n, 'A', 'B', 'C');
    getch();
}
void TOH(int n, char x, char y, char z)
{
    if(n>0)
    {
        TOH(n–1, x, z, y);
        printf("\n %c -> %c", x, y);
        TOH(n–1, z, y, x);
    }
}
```

Output

```
Enter number of plates:3
A -> B
A -> C
B -> C
A -> B
C -> A
C -> B
A -> B
```

TechKnowledge Publication

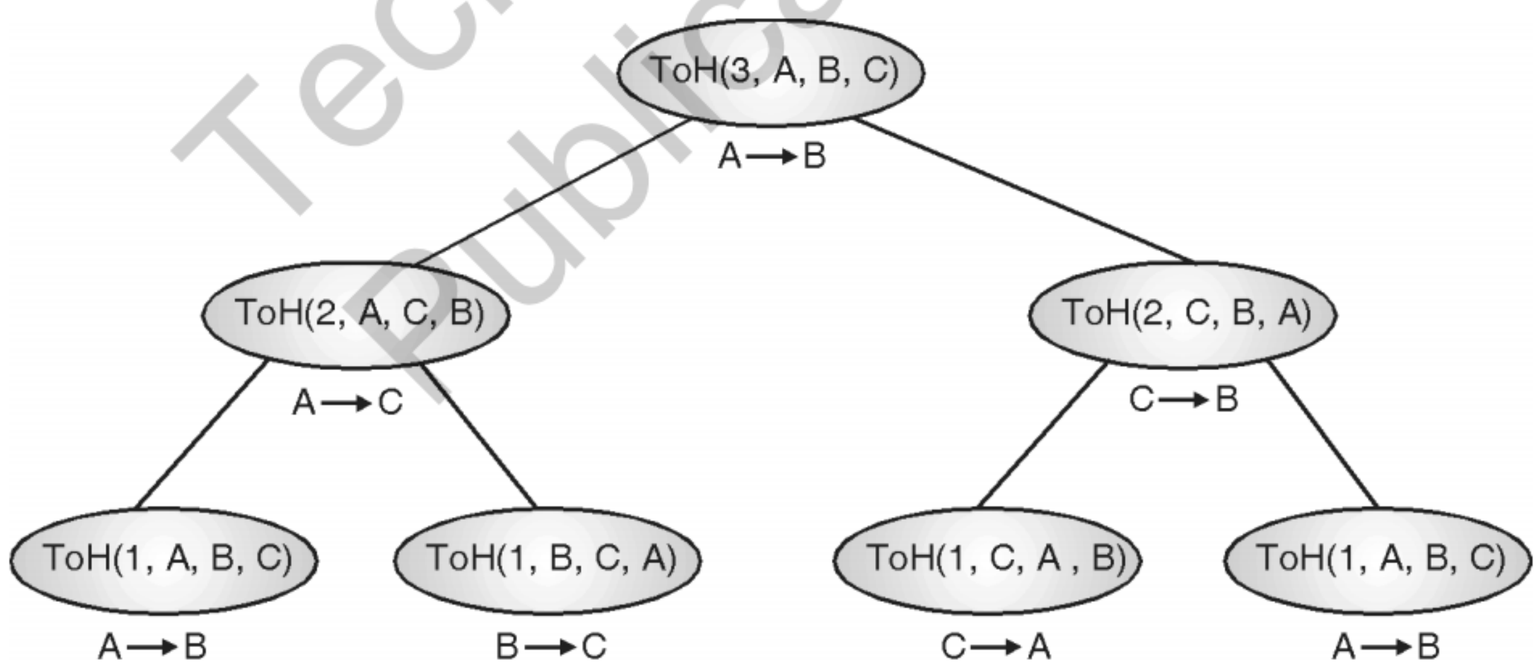


Fig. 7.10.3 : Recursion tree for the function TOH ( ) for n = 3 plates

Sequence of moves

(1) A → B (2) A → C (3) B → C

(4) A → B (5) C → A (6) C → B

(7) A → B

Sequence of moves can easily be written by making an inorder traversal on recursion tree. Concept of inorder traversal is explained in the chapter on "tree".

7.11 Backtracking

Many a time we have to find an optimal solution to a problem, yet there may be no applicable theory to help us find the optimum. We resort to exhaustive search. Such an exhaustive search technique is known as backtracking.

A practical example of backtracking algorithm is the problem of arranging furniture in a new house. There are many possibilities but only few will be accepted by the owner. Starting with an empty house, each piece of furniture is placed in some part of the room. If all the furniture is placed and the owner is satisfied then the process terminates. When we reach a point where the arrangement is not desirable then we undo the last step and try some other combination. This may force another undo.

7.12 Variants of Recursion

There are several types of recursion :

1. Direct Recursion

When a function calls itself, it is known as direct recursion.

Example :

$$\text{Factorial}(n) = \begin{cases} 1, & \text{if } n = 0 \\ n * \text{Factorial}(n-1), & \text{if } n > 0 \end{cases}$$

2. Indirect Recursion

If function A calls function B, function B, function C and function C calls function A.

3. Tail Recursion

Tail recursion refers to a recursive call at the last line. Tail recursion can be eliminated by changing the recursive call to a goto preceded by a set of assignments per function call. This simulates the recursive call because nothing needs to be saved after the recursive call finishes. We can just goto the top of the function with the values that would have been used in a recursive call.

Recursive function TOH with tail recursion

```
void TOH(int n, char x, char y, char z)
{
    if(n > 0)
    {
        TOH(n – 1, x, z, y);
        printf("\ n% c → % c", x, y);
        TOH(n – 1, z, y, x);          /*  tail recursion  */
    }
}
```

Without tail recursion

```
void TOH(int n, char x, char y, char z)
{
    char temp;
    label0 :
        if(n > 0)
        {
            TOH(n – 1, x, z, y);
            printf("\n % c → % c", x, y);
            temp = x;
            x = z;
            z = temp;
            n = n – 1;
            goto label0;
        }
}
```

4. Tree Recursion

When a function is called at several places in its body then it becomes an example of tree recursion.

Example : The nth term of a Fibonacci sequence is given by

$$T_n = T_{n-1} + T_{n-2} \quad \text{if } n > 1$$
$$T_1 = 1$$
$$T_0 = 0$$

Techknowledge Publication

A recursion tree for T_4 is given below

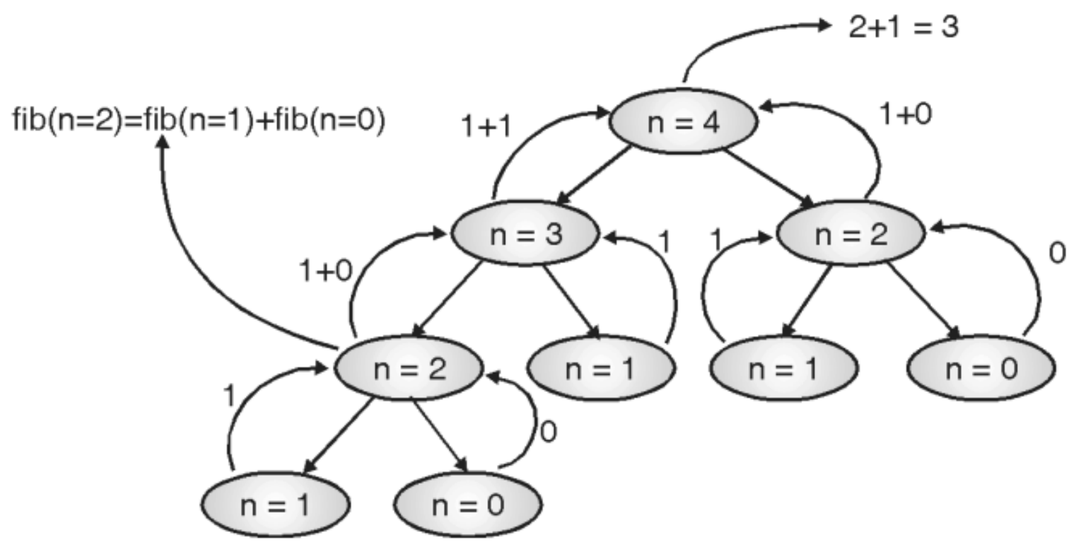


Fig. 7.12.1 : A recursion tree, showing calculation of T_4

7.13 Backtracking Algorithmic Strategy

Q. What is backtracking ? **SPPU - Dec. 18, 2 Marks**

In many problems, there may not be any known algorithm for finding optimal solution. In such cases, we have to resort to exhaustive search to locate the optimal solution. Backtracking is an exhaustive search technique.

7.13.1 Use of Stack in Backtracking

Q. Explain use of stack for recursion. **SPPU - Dec. 17, Dec. 18, 2 Marks**

Backtracking is a form of recursion. Any recursive algorithm can be rewritten as a stack algorithm. In fact, this how your recursive algorithms are translated into machine or assembly language. The concept of removing recursion is explained below.

7.13.2 Removal of Recursion

- Any recursive function can be converted to non-recursive function through use of a stack.
- A recursive call is similar to a call to another function.
- Any call to a function requires that the function has storage area where it can store its local variables and actual parameters.
- Return address must be saved before a call is made to a function.
- Storage area for local variables, actual parameters and the return address can be provided through a stack.
- In case of a recursive call, the value of local variables, parameters and the return address must be saved on the stack.
- While returning from a nested call, the previous outer call must be recalled with resetting all the local variables and operation must resume from where it was suspended.

Rules for converting a recursive algorithm to non-recursive one

(a) Initialization

1. Declare stack – It will hold local variables, parameters, return address etc.
2. The first statement after the stack initialization must have a label.

(b) Steps required to replace a recursive call

3. Push all local variables and parameters into the stack.
4. Push an integer "i" into the stack. It gives the return address.
5. Set the value of formal parameters.
6. Transfer the control to the beginning of the function (i.e. first label immediately after initialization of stack) using goto.
7. There should always be a label statement immediately following the recursive call. This label is the return address.

(c) Steps required at the end of recursive function

8. If the stack is empty, then the recursion is finished.
9. Otherwise, pop the stack to restore the values of all local variables and parameters called by value.
10. Pop the return address.

7.14 Case Study : The n-queens Problem (8 Queen as a Case)

Q. Explain use of backtracking in n–Queen's problem. **SPPU - May 19, 4 Marks**

Technknowledge Publications

Ex. 7.14.1 : The problem is to place eight queens on an 8×8 chessboard. No two should attack each other.

a) No two should be on the same row

b) No two should be on the same column

c) No two should be on the same diagonal

	0	1	2	3	4	5	6	7
0				Q				
1						Q		
2								Q
3		Q						
4							Q	
5	Q							
6			Q					
7					Q			

Fig. Ex. 7.14.1 : One solution to the 8-queens problem

Soln. :

Let us assume that :

(a) Rows of the chess board are numbered from 0 to 7

(b) Columns of the chess board are numbered from 0 to 7

(c) Queens are numbered from 0 to 7

Since, each row can have exactly one queen, we can assume that queen i is placed in row i of the chessboard.

A solution of the chessboard is given by 8 tuple $(x_0, x_4, \ldots. x_7)$ where x_i is the column on which queen i is placed.

Solution of board of Fig. Ex. 7.14.1(a) can be represented as (3, 5, 7, 1, 6, 0, 2, 4)

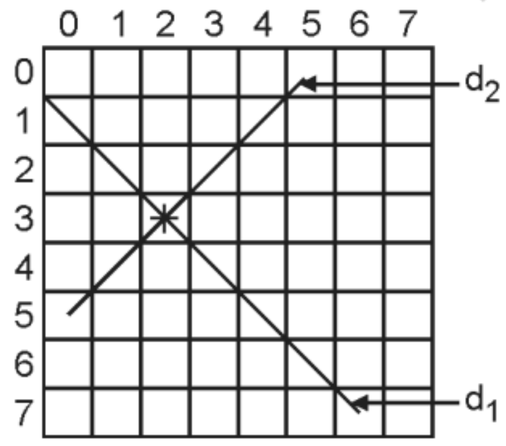


Fig. Ex. 7.14.1(a) : A chess board

Relation between two cells on the same diagonal

Let us consider a cell with index (3,2).

Cells on diagonal d_1 are given by : (1, 0), (2,1), (4,3), (5,4), (6,5), (7,6).

Cells on diagonal d_2 are given by : (5, 0), (4,1), (2,3), (1,4), (0,5).

Two elements (i, j) and (k, l) on diagonal d_1 will have following relation :

$$i - j = k - l$$

$$\text{Or, } j - l = i - k \qquad \ldots(1)$$

Two elements (i, j) and (k, l) on diagonal d_2 will have following relation:

$$i + j = k + l$$

$$\text{Or, } j - l = k - i \qquad \ldots(2)$$

From (1) and (2) we can infer that two queens lie on the same diagonal if and only if

$$|j - l| = |i - k|$$

Function **queens( )** prints all possible placements of n queens on an n×n chessboard so that they are non-attacking. Function queens( ) has two parameters :

(1) **k :** row in which the current queen is to be placed.

(2) **n :** number of queens.

Initial values of k and n are 0 and 7 respectively.

```
void queens(int k, int n)
{
    int i, j ,
    /*find a non-attacking cell in kth row*/
    for(i = 0; i<n, i++)
    if(place(k, i ) == 0)
    {
        x[k] = i ,
        if(k == n)   /*if all queens have been placed*/
        {
            for(j = 0; j<n; j++)
            printf("%d \t", x[i]);
            printf("\n ");
        }
    }
        else
            queens(k+1, n);
}
```

TechKnowledge Publications

x[ ] is global array and initialized with –1. Function place(k,i) checks whether the queen can be placed in k^{th} row and i^{th} column. If the queen can be placed, the function returns a value 1 otherwise a value 0.

```
int place(int k, int i)
{
    int j;
    /* check the position with respect to queens
       already placed */
    for(j = 0; j<k; j++)
        if(x[j] ! = –1)
            if((x[j] == i) ||(abs(x[j]–i) == abs(j–k)))
        return(0);
    return(1);
}
```

7.15 Case Study : Android-Multiple Tasks / Multiple Activities and Back Stack

A task is a collection of activities that users interact with when performing a certain job. These activities are arranged in a stack in the order in which each activity is opened. This stack is known as back stack.

When a current activity starts another activity, the new activity is pushed on the top of the stack. The previous activities remain in the stack but they are stopped.

- When an activity stops, the system retains the current state of its user interface.
- With the help of back button, the current activity can be popped from the stack. This will also resume the previous activity.

Review Questions

Q. 1 What is recursion ? Write and explain recursive function to find factorial of a number. **(6 Marks)**

Q. 2 Write recursive function to add digits of a given positive integer. **(4 Marks)**

Q. 3 Write a pseudo code to find out length of string without using library function. **(4 Marks)**

Q. 4 Write a pseudo code for the following stack operations :

(i) push operation ;

(ii) pop operation **(4 Marks)**

❑❑❑

Technknowledge Publication

CHAPTER 8

Unit VI

Queues

Syllabus :

Basic concept, Queue as Abstract Data Type, Representation of Queue using Sequential organization, Queue Operations, Circular Queue and its advantages, Multi-queues, Linked Queue and Operations. Deque-Basic concept, types (Input restricted and Output restricted), Priority Queue- Basic concept, types(Ascending and Descending).

Exemplar/Case Studies : Priority queue in bandwidth management

8.1 Array Implementation of Queues

Q. Define the term Linear queue with example.

SPPU - Dec. 17, May 18, Dec. 19, 2 Marks

Q. Explain linear queue and circular queue with suitable example.

SPPU - Dec. 17, May 18, Dec. 19, 3 Marks

8.1.1 Definition

It is a special kind of list, where items are inserted at one end (the rear) and deleted from the other end (front). Queue is a FIFO (First In First Out) list.

We come across the term queue in our day to day life. We see a queue at a railway reservation counter, or a movie theatre ticket counter. Before getting the service, one has to wait in the queue. After receiving the service, one leaves the queue. Service is provided at one end (the front) and people join at the other end (rear).

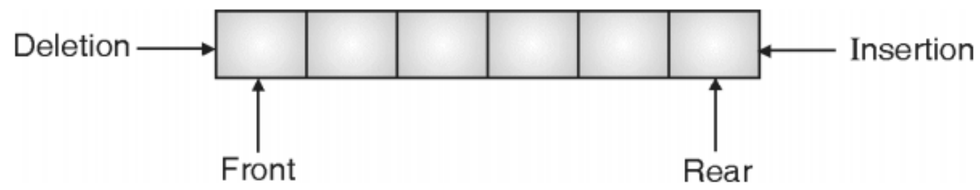


Fig. 8.1.1 : Insertion of elements is done at the rear end and deletion from the front end

8.1.2 Application of Queues

Various features of operating system are implemented using a queue.

(a) Scheduling of processes (Round Robin Algorithm).

(b) Spooling (to maintain a queue of jobs to be printed).

A queue of client processes waiting to receive the service from the server process.

– Various application software using non-linear data structure tree or graph requires a queue for breadth first traversal.

– Simulation of a real life problem with the purpose of understanding its behaviour. The probable waiting time of a person at a railway reservation counter can be found through the technique of computer simulation if the following concerned factors are known :

(a) Arrival rate

(b) Service time

(c) Number of service counters

8.1.3 Array Representation and Implementation of Queues

An array representation of queue requires three entities.

(a) An array to hold queue elements.

(b) A variable to hold the index of the front element.

(c) A variable to hold the index of the rear element.

Technical Publication

A queue datatype may be defined formally as follows :

```
# define MAX 30
typedef struct queue
{
    int data[MAX];
    int front, rear;
} queue;
```

– During initialization of a queue, its front and rear are set to –1.

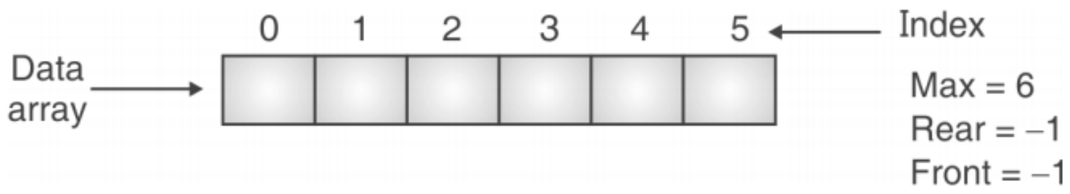


Fig. 8.1.2 : An empty queue after initialization

– Fig. 8.1.3 shows the status of a queue after insertion of the element '5'.

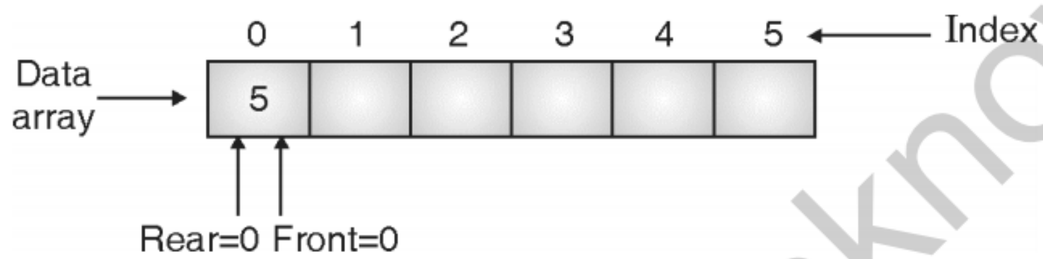


Fig. 8.1.3 : A queue after insertion of the first element '5'

– On subsequent insertions, front remains at the same place, where rear advances.

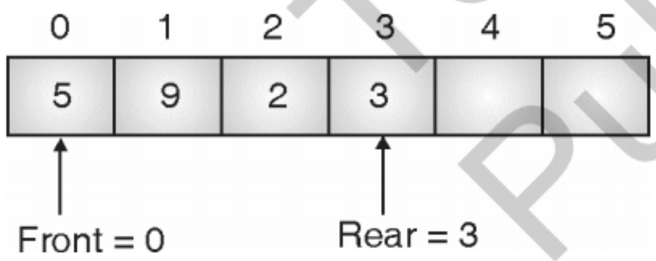


Fig. 8.1.4 : A queue after insertion of four elements

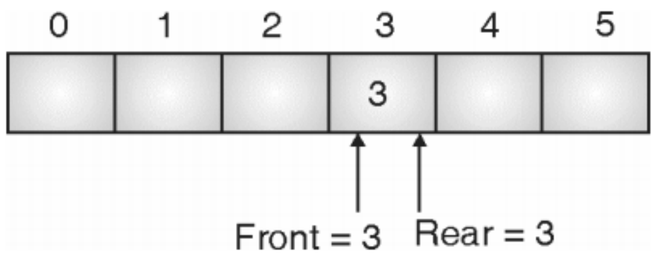


Fig. 8.1.5 : A queue after 3 successive deletions

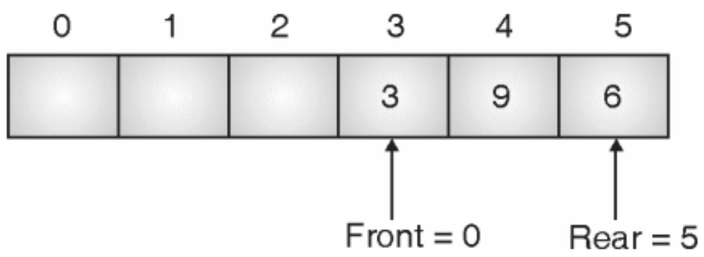


Fig. 8.1.6 : A queue after 2 successive insertions

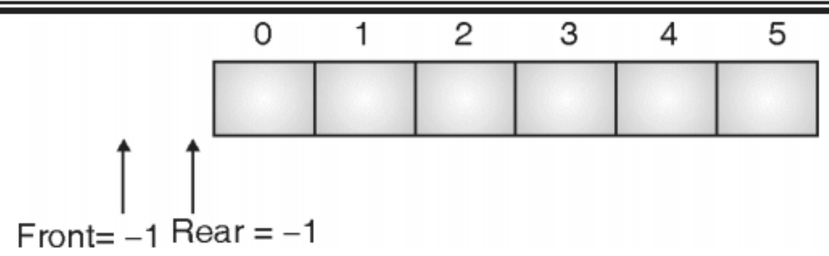


Fig. 8.1.7 : A queue after 3 successive deletions

Following points can be observed :

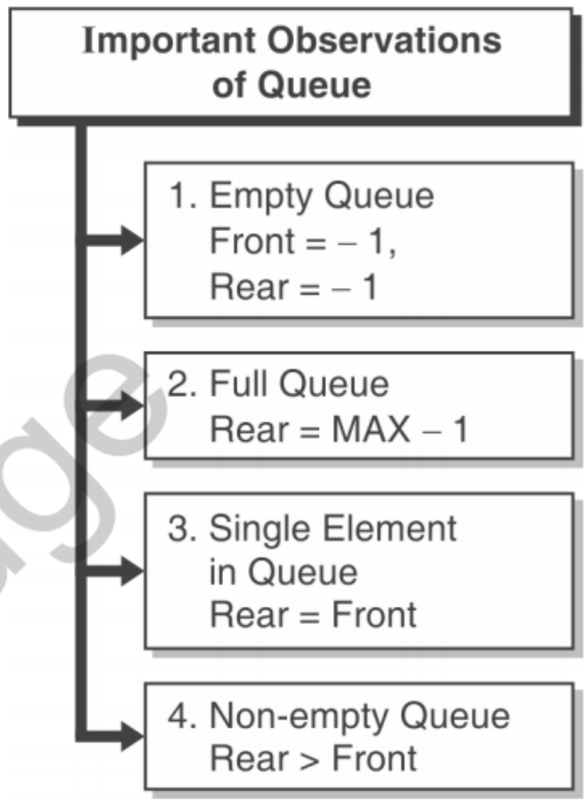


Fig. 8.1.8

(1) If the queue is empty then front = –1 and rear = –1.

(2) If the queue is full then rear = MAX – 1.

(where MAX is the size of the array used for storing of queue elements).

(3) If rear = front then the queue contains just one element.

(4) If rear > front then queue is non-empty.

Problem with the above representation of the queue :

Note : Problem of overflow-Refer to the Fig. 8.1.6, the queue has become full as rear = MAX –1. There are three vacant spaces (location 0, 1, 2) but these spaces can not be utilized.

Problem of overflow can be handled by moving the queue elements to their left by number of vacant spaces. This operation could be very time consuming for a large queue.

Tech Knowledge Publications

8.2 Operations on Queue

8.2.1 Operations on Queue Implemented using Array

Q. Write pseudo C/C++ code to perform insert and delete operation on linear queue.
SPPU - Dec. 18, 6 Marks

A set of useful operations on a queue includes :

(1) **initialize( ) :** Initializes a queue by setting the value of rear and front to –1.

(2) **enqueue( ) :** Inserts an element at the rear end of the queue.

(3) **dequeue( ) :** Deletes the front element and returns the same.

(4) **empty( ) :** It returns true(1) if the queue is empty and returns false(0) if the queue is not empty.

(5) **full( ) :** It return true(1) if the queue is full and returns false(0) if the queue is not full.

(6) **print( ) :** Printing of queue elements.

Realization of queue operations through C-function

(a) Defining the maximum size of the queue.

```
#define MAX 50
```

(b) Data structure declaration

```
typedef struct Q
{
    int R, F;
    int data[MAX];
} Q;
```

"R" and "F" store the index of the rear and the front element respectively. "data[MAX]" is used to store the queue elements.

(c) Declaring a queue type variable.

Q q_1, q_2 ;

q_1 and q_2 are queue type variables

(d) Declaring a queue type pointer variable.

Q q_1, *P;

P = &q_1

A pointer variable "P" can be used to store the address of a queue type variable.

(I) 'C' function to initialize( ).

```
void initialize(Q *P)
{
    P → R = – 1;
    P → F = – 1;
}
```

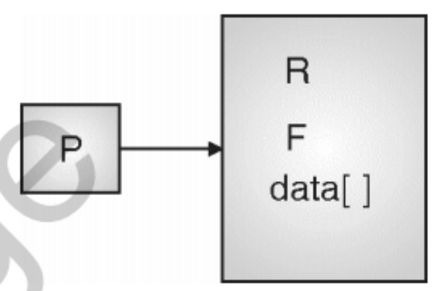


Fig. 8.2.1 : Elements of the structure 'Q' can be accessed through the pointer P (P → R, P → F, P → data[ ])

In the function "void initialize(Q *P)" a "Q" type variable is passed by address. Members of the structure type variables can be accessed through a pointer as shown in the Fig. 8.2.1. Initially the queue is empty and hence the value of the two variables "R (rear)" and "F (front)" is set to – 1.

(II) 'C' function to check whether queue is empty or not.

```
int empty(Q *P)
{
    if(P → R = = – 1)
        return(1);
    return(0);
}
```

Whenever the queue is empty, P → R (i.e. rear field (R) of the queue type variable whose address is stored in P) or P → F will be – 1.

(III) 'C' function to check whether queue is full or not.

```
int full(Q *P)
{
    if(P → R = = MAX – 1)
        return(1);
    return(0);
}
```

TechKnowledge Publications

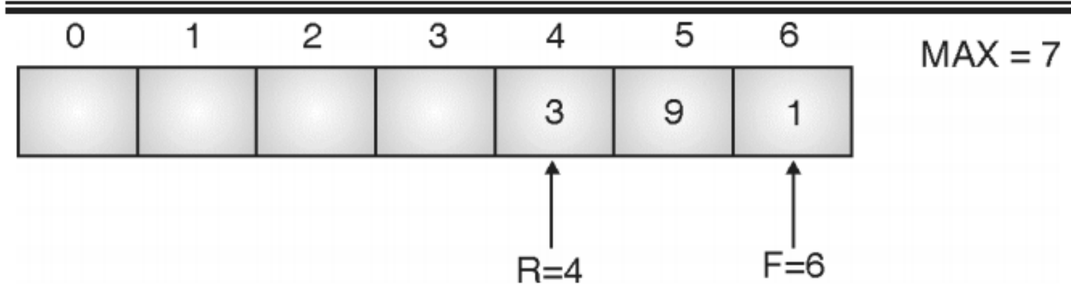


Fig. 8.2.2 : The queue is full as there is no space left for further insertion

(IV) enqueue( )

Algorithm for insertion in a queue

(1) Inserting in an empty queue.

R = F = 0;

data[R] = x /* x is the element to be inserted */

/* Both R and F will point to the only element of the queue */

(2) Inserting in a non-empty queue.

R = R + 1;

data[R] = x;

/* rear is advanced by 1 and the element x is stored at the rear end of the queue */.

Note : Insertion should be carried out after it is checked that the queue is not full.

'C' function for insertion in a queue using array.

```
void enque(Q *P, int x)
{
    if(P → R = = – 1)          /* empty queue */
    {   P → R = P → F = 0;
        P → data[ P→ R] = x;
    }
    else
    {   P → R = P → R + 1;
        P → data[P → R] = x;
    }
}
```

(V) dequeue( )

Algorithm for deletion from a queue :

(1) Deletion of the last element or only element(R = F)

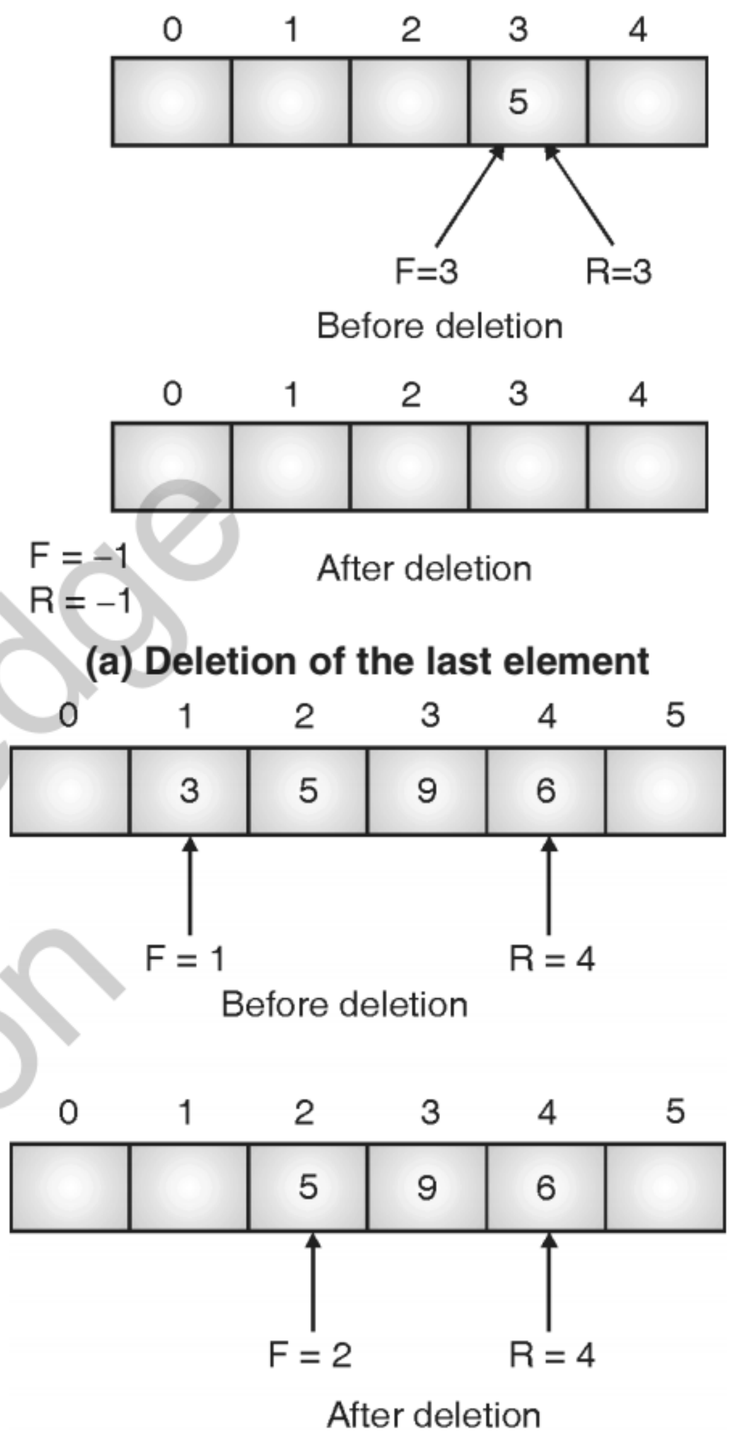
(a) x = data[F] (b) R = F = – 1;

(c) return(x)

(2) Deletion of the element when the queue length > 1

(a) x = data[F] (b) F = F + 1

(c) return(x);



(b) Deletion of an element from the queue when the queue length > 1

Fig. 8.2.3

'C' function for deletion in a queue.

```
int dequeue(Q *P)
{   int x;
    x = P->data[P->F];
    if(P->R = = P->F)
    {   P->R = –1;
        P->F = –1;
    }
    else
        P->F = P->F+1;
    return(x);
}
```

Technknowledge Publication

Note : Deletion of an element from the queue should be carried out after checking that it is not empty.

Program 8.2.1 : Show the status of the queue after every operation : (a) Insert 5 elements (b) Delete 2 elements

```
#include<conio.h>
#include<stdio.h>
#define MAX 10
typedef struct Q
{
    int R, F;
    int data[MAX];
} Q;
void initialise(Q *P);
int empty(Q *P);
int full(Q *P);
void enqueue(Q *P, int x);
int dequeue(Q *P);
void print(Q *P);
void main()
{
    Q q;
    int x, i;
    initialise(&q);
    printf("\n Enter 5 elements :");
    for(i = 1; i< = 5; i++)
    {
        scanf("%d", &x);
        if(!full(&q))
            enqueue(&q, x);
        else
        {
            printf("\n Queue is full .....exiting");
            exit(0);
        }
    }
    print(&q);
    for(i = 1; i< = 2; i++)
    {
        if(!empty(&q))
            x = dequeue(&q);
        else
        {
            printf("\n cannot delete...Queue is empty");
            exit(0);
        }
    }
    print(&q);
}
void initialise(Q *P)
{
    P->R = –1;
    P->F = –1;
}
int empty(Q *P)
{
    if(P->R = = –1)
        return(1);
    return(0);
}
int full(Q *P)
{
    if(P->R = = MAX–1)
        return(1);
    return(0);
}
void enqueue(Q *P, int x)
{
    if(P->R = = –1)
    {
        P->R = P->F = 0;
        P->data[P->R] = x;
    }
    else
    {
        P->R = P->R+1;
        P->data[P->R] = x;
    }
}
int dequeue(Q *P)
{
    int x;
    x = P->data[P->F];
    if(P->R = = P->F)
    {
        P->R = –1;
        P->F = –1;
```

Techknowledge Publications

```
    }
    else
        P->F = P->F+1;
    return(x);
}
void print(Q *P)
{
    int i;
    if(!empty(P))
    {
        printf("\n ");
        for(i = P->F; i< = P->R; i++)
            printf("%d\t", P->data[i]);
    }
}
```

Output

```
Enter 5 elements :  5     4     3    2    1
5     4     3     2     1
3     2     1
```

8.2.2 Queue as an ADT

Data type for queue in an array

```
#define MAX 30
/* A queue with maximum of 30 elements */
typedef struct queue
{   int data [MAX];
    int rear, front;
}   queue;
```

Operations on a queue

(i) initialize( ) : Make the queue empty

(ii) empty( ) : Determine if queue is empty.

(iii) full( ) : Determine if queue is full.

(iv) enqueue( ) : Insert an element at the rear end of the queue.

(v) dequeue( ) : Delete the front element.

(vi) print( ) : Print elements of the queue.

Prototype of functions used for various operations on queue

(i) void initialise(queue *p);

(ii) int empty(queue *p);

Function returns 1 or 0, depending on whether the queue pointed by p is empty or not.

(iii) **int full(queue *p) :** Function returns 1 or 0, depending on whether the queue is full or not.

(iv) void enqueue(queue *p, int x);

(v) int dequeue(queue *p) ;

function returns the front element.

(vi) void print(queue *p);

(vii) **enqueue( ) :** Operation will cause an overflow if the queue is full.

(viii) **dequeue( ) :** Operation will cause an underflow if the queue is empty( ).

Ex. 8.2.1 : Consider the following queue of characters, implemented as array of six memory locations :

Front = 2, Rear = 3

Queue : –, A, D, –, –, –

Where '–' denotes empty cell. Describe the queue as the following operations take place

(i) Add 'S'

(ii) Add 'J'

(iii) Delete two letters

(iv) Shift towards left to bring all free spaces to the right side

(v) Insert M, H, I and delete on letter.

Soln. : Initial queue :

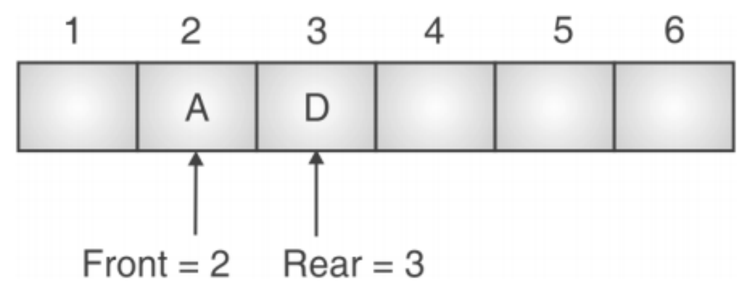


Fig. Ex. 8.2.1(a)

(i) Add 'S'

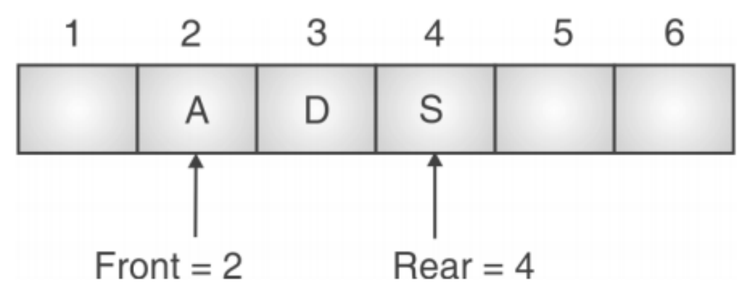


Fig. Ex. 8.2.1(b)

(ii) Add 'J'

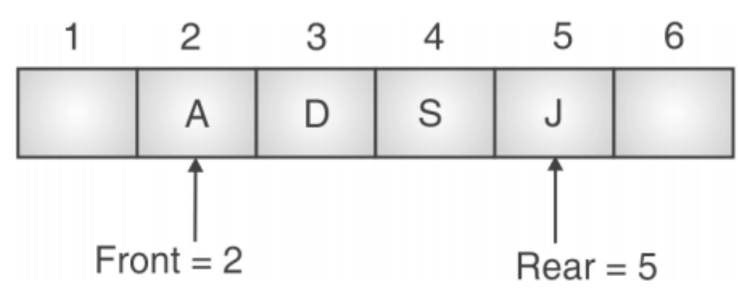


Fig. Ex. 8.2.1(c)

Technknowledge Publication

(iii) Delete two letters

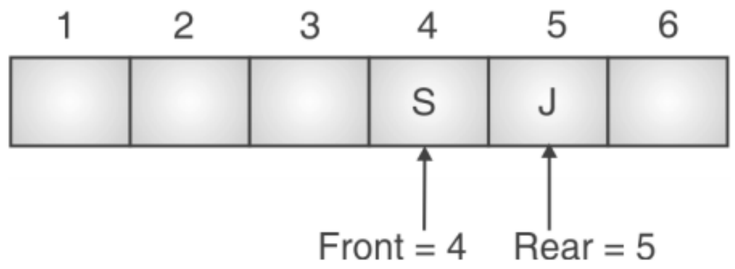


Fig. Ex. 8.2.1(d)

(iv) Shift towards left to bring all free spaces to the right.

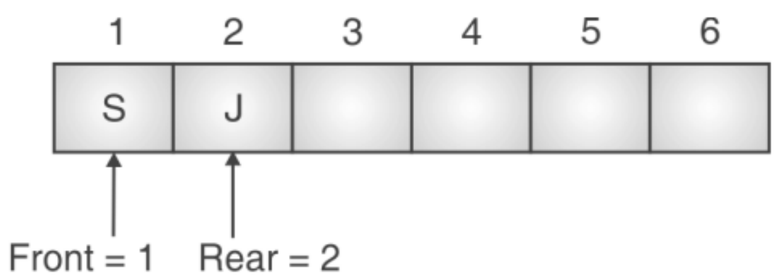


Fig. Ex. 8.2.1(e)

(v) Insert M, H, I and delete one letter.

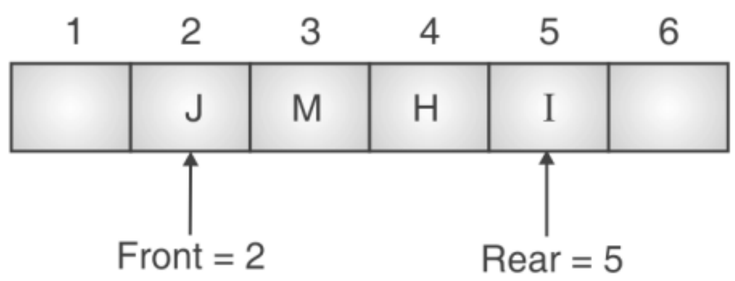


Fig. Ex. 8.2.1(f)

Ex. 8.2.2 : Write pseudo 'C' code to implement linear queue using an array to perform the following operations :

(i) Addition of an element to queue

(ii) Deletion of an element from queue

(iii) Display front element

(iv) Display rear element

Soln. :

(i) Addition of an element to queue

```
void enqueue(Q, item, n, rear, front)
{
    /* Insert item into the queue represented in
       Q(0:n–1) */
    if(rear = = n)
        print "queue is full";
    else
    {
        if(rear = = –1)          // queue is empty
            rear = front = 0;
        else
            rear = rear + 1;
        Q(rear) = item;
    }
}
```

(ii) Deletion of an element from the queue

```
item dequeue(Q, n, rear, front)
//Delete an item from the queue represented in Q(0: n – 1)
{
    if(front = = –1)
        print "underflow";
    else
    {   item = Q(front);
        if(front = = rear)
        //delete the last element
            rear = front = –1;
        else
            front = front –1;
        return(item);
    }
}
```

(iii) Display front element

```
void Display_front(Q, front)
{  /* Display the front element of the queue
      represented in Q(0: n–1) */
    if(front = = –1)  //queue is empty
        print "queue is empty"
    else
        print Q(front);
}
```

(iv) Display rear element

```
void Display_rear(Q, rear)
 {
    /* Display the rear element of the queue
       represented in Q(0:n–1) */
    if(rear = = –1) "queue is empty"
        print "queue is empty"
    else
        print Q(rear);
}
```

Technknowledge Publication

Ex. 8.2.3 : A linear queue using array has a size of 3. Perform the following on this queue and show the sequence of steps with necessary diagrams indicating values of front, rear and contents of queue :

(i) insert 10 (ii) insert 20
(iii) insert 30 (iv) delete an element
(v) insert 40 (vi) delete an element
(vii) delete an element (viii) delete an element
(ix) insert 50

Soln. :

Operation	Queue
Initial	0 1 2 Rear = –1, Front = –1
Insert 10	0 1 2 10 Rear = 0, Front = 0
Insert 20	0 1 2 10 20 Front = 0 Rear = 1
Insert 30	0 1 2 10 20 30 Front = 0 Rear = 2
Delete an element	0 1 2 20 30 Front = 1 Rear = 2
Delete an element	0 1 2 30 Front = 2, Rear = 2
Delete an element	0 1 2 Rear = –1, Front = –1

Operation	Queue
Insert 50	0 1 2 50 Front = 0, Rear = 0

8.2.3 Operations on Queue Implemented using Linked Structure

Q. Write pseudo C/C++ code to implement linked queue. **SPPU - May 18, 7 Marks**

Q. Write pseudo C/C++ code to implement a simple queue using linked list. **SPPU - May 19, 6 Marks**

A set of useful operations on a queue includes :

(1) initialize( ) : Initializes a queue by setting the value or rear and front pointer to "NULL".

(2) enqueue( ) : Inserts an element at the rear end of the queue.

(3) dequeue( ) : Deletes the front element and returns the same.

(4) empty( ) : It returns true(1) if the queue is empty and returns false(0) if the queue is not empty.

(5) Print( ) : It prints the queue elements from front to rear.

(a) Data structure declaration.

```
typedef struct node
{
    int data;
    struct node *next;
}node;
typedef struct Q
{
    node *R;
    node *F;
} Q;
```

(b) Declaring a queue type variable

$Q\ q_1, q_2$;

Element q_1 of the type Q(queue)

Technknowledge Publication

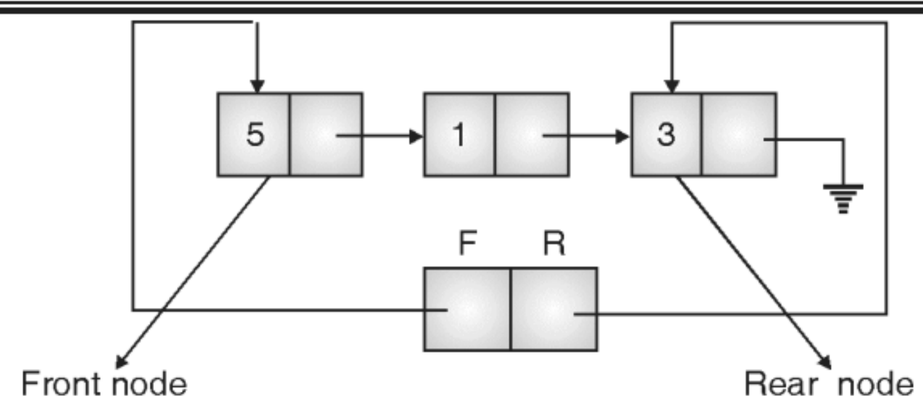


Fig. 8.2.4 : Memory representation of a queue

Data field of the front node is $(q_1.F) \rightarrow$ data

Next field of the front node is $(q_1.F) \rightarrow$ next

Data field of the rear node is $(q_1.R) \rightarrow$ data

Next field of the rear node is $(q_1.R) \rightarrow$ next;

(c) Declaring a queue type pointer

Q q_1, *P;

A queue type variable
P = & q_1;

A queue type pointer
/* P points to q_1 */

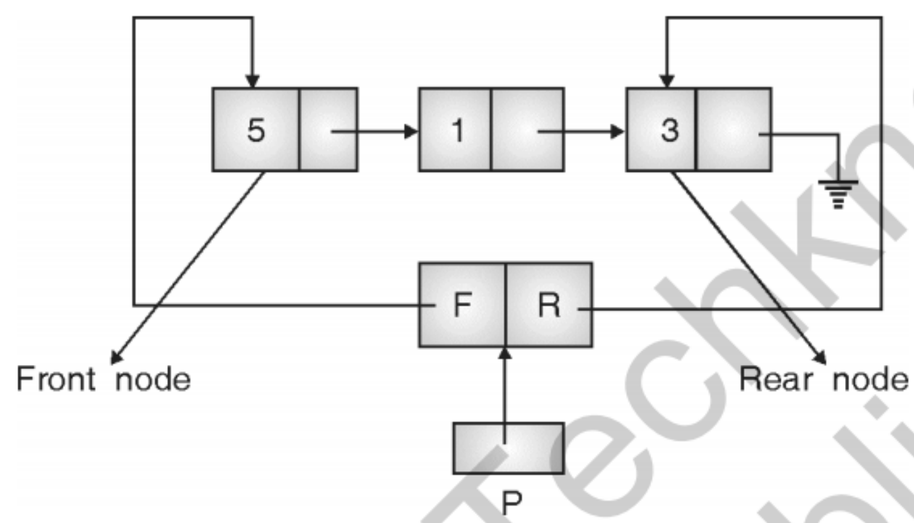


Fig. 8.2.5 : Memory representation of a queue with its address in P

Data field of the front node is $(P \rightarrow F) \rightarrow$ data;

Next field of the front node is $(P \rightarrow F) \rightarrow$ next;

Data field of the rear node is $(P \rightarrow R) \rightarrow$ data;

Next field of the rear node is $(P \rightarrow R) \rightarrow$ next;

'C' function to initialize queue.

```
void initialize(Q *P)
{
    P → R = NULL;
    P → F = NULL;
}
```

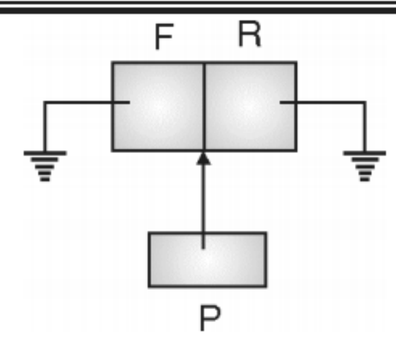


Fig. 8.2.6 : Initial state of a queue. queue is addressed through the pointer P

'C' function to check whether queue is empty or not.

```
int empty(Q *P)
{
    if(P → R = = NULL)          /* empty queue */
        return(1);
    return(0);
}
```

enqueue( ) function

enqueue function inserts a value (say x) at the rear end of the queue.

Steps required for insertion of an element x in a queue

- Memory is acquired for the new node.
- Value x is stored in the new node.
- New node is inserted at the rear end.
- Special care should be taken for insertion into an empty queue. Both rear and front pointers will point to the only element of the queue.

```
Node *P; Q q;
P=(node *) malloc(sizeof(node)); // acquire memory
P → data = x;                    // store x in the node
P → next = NULL;
if(empty(&q))
{       /* inserting element  in an empty queue */
        q.R = q.F = P;
}
else
{
        (q.R) → next = P;
        q.R = P;
}
```

Tech Knowledge Publications

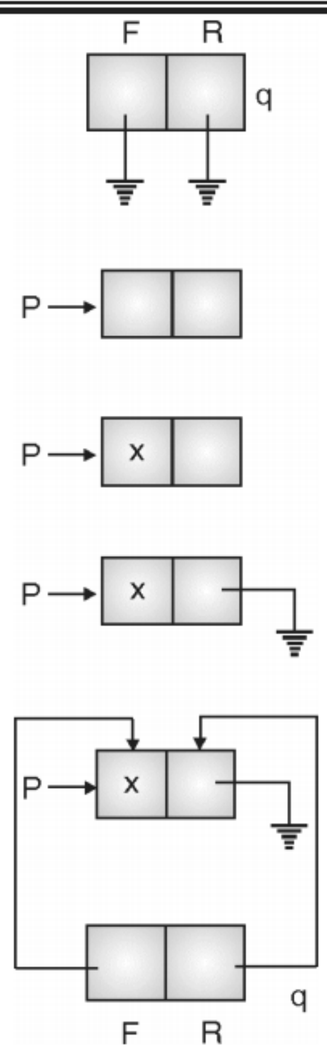
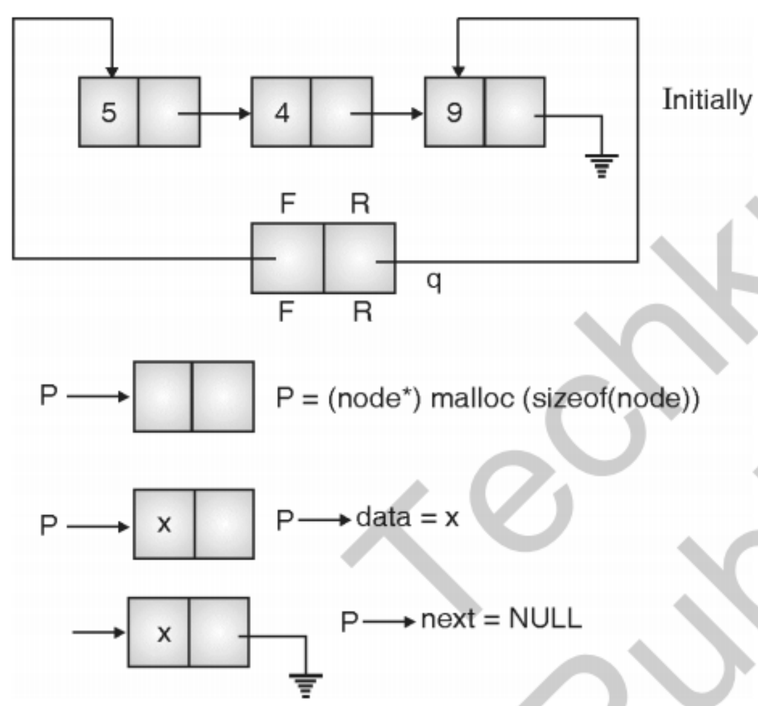


Fig. 8.2.7 : Insertion in an empty queue (stepwise)



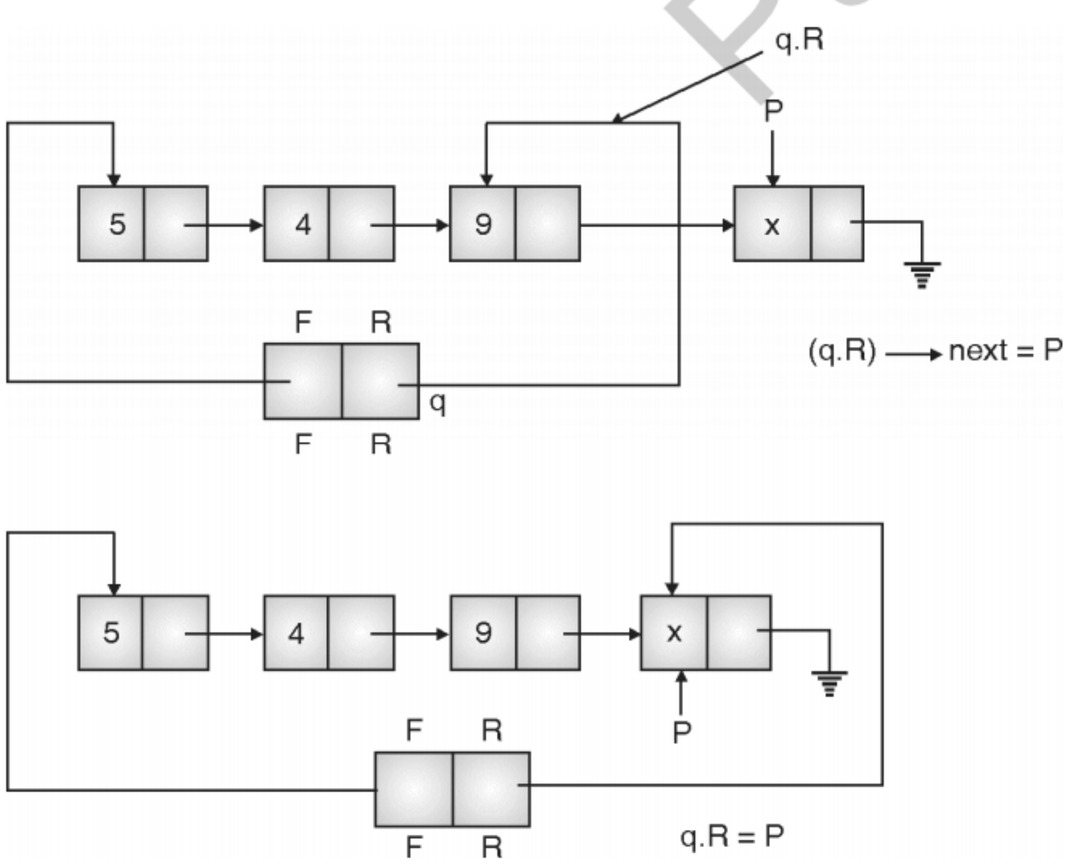


Fig. 8.2.8 : Insertion into a non-empty queue (stepwise)

Queue type variable "q" should be passed by address for insertion( ) operation value of q (rear field) changes after insertion.

'C' function for insertion.

```
void insert(Q *qP, int x)
{
    node *P;
    P = (node *) malloc(sizeof(node));
    P → data = x;
    P → next = NULL;
    if(empty(qP))
    {
        qP → R = qP → F = P;
    }
    else
    {
        qP → R → next = P;
        qP → R = p;
    }
}
```

dequeue( ) Function

dequeue( ) function deletes the front node of the queue and returns the value stored in the node, to the calling program. Front pointer of the queue is advanced to point to the next node. Special care should be taken while deleting the last node. As it will make the queue empty after deletion. Memory used by the deleted node should be released.

Steps required for deletion of a node from the queue :

```
Node *P ; Q  q;
P = q.F; /* store the address of the front node in P */
x = P → data
if(q.R = = q.F)          /* deleting the last node */
{
    initialize(&q);      /* make the queue empty */
    free(P);
    return(x);
```

TechKnowledge Publications

```
}
else
{   q.F = P → next;    /* Advance the front pointer */
    free(P);
    return(x);
}
```

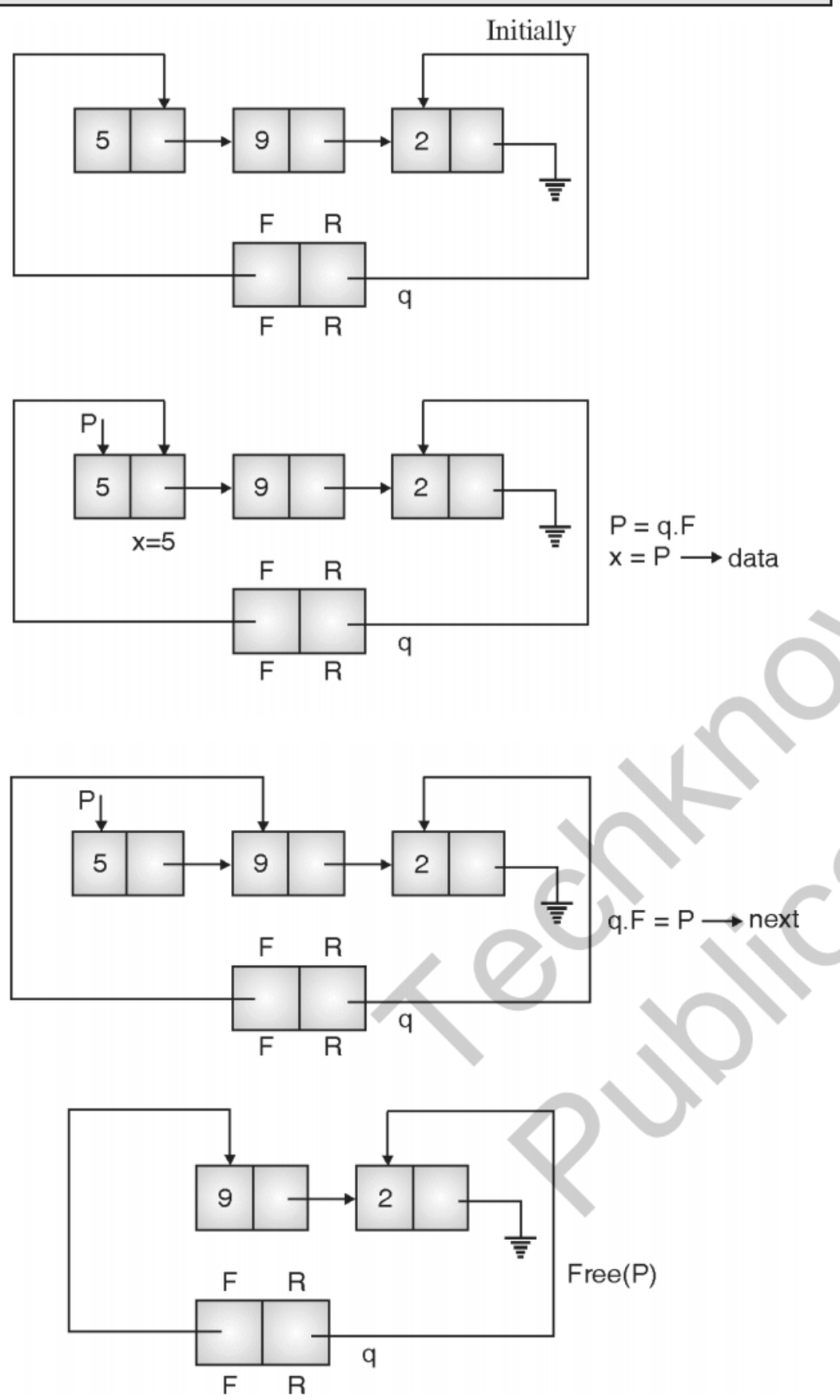


Fig. 8.2.9 : Deletion of the front node from a queue (stepwise)

'C' function for deletion

```
int dequeue(Q *qP)
{
    int x;
    node *P;
    P = qP → F;
    x = P → data;
    if(qP → R = = qP → F)   /* last element */
        initialize(qP);
    else
        qP → F = P → next;
    free(P);
    return(x);
}
```

'C' function for printing queue.

Elements of the queue can be printing by traversing the underlying linked list starting from the front node.

```
void print(Q *qP)
{
    node *P;
    P = qP → F; /* start from front */
    while(P! = NULL)
    {
        printf("\n %d", P -> data);
        P = P → next;
    }
}
```

Program 8.2.2 : A sample program showing various operations on a queue represented using a linked list. (a) Insert five values in the queue. (b) Print elements of the queue. (c) Delete 2 elements from the queue. (d) Print elements of the queue. (e) Delete the remaining elements of the queue.

OR Write pseudo C/C++ code to implement circular queue using linked list. **SPPU - May 17, 7 Marks**

```
#include<conio.h>
#include<stdio.h>
#define MAX 10
typedef struct node
{
    int data;
    struct node *next;
}node;

typedef struct Q
{
    node  *R, *F;
}Q;
```

Technical Knowledge Publications

```
void initialise(Q *);
int empty(Q *);
int full(Q *);
void enqueue(Q *, int);
int dequeue(Q *);
void print(Q *);
void main()
{
    Q q;
    int x, i;
    initialise(&q);
    printf("\n Enter 5 elements :");
    for(i = 1; i< = 5; i++)
    {
        scanf("%d", &x);
        enqueue(&q, x);
    }
    printf("\n displaying queue :");
    print(&q);
    x = dequeue(&q);
    x = dequeue(&q);
    printf("\n after deletion of two elements :");
    print(&q);
    //delete remaining elements
    while(!empty(&q))
        x = dequeue(&q);
    getch();
}
void initialise(Q *qP)
{
    qP->R = NULL;
    qP->F = NULL;
}
void enqueue(Q *qP,int x)
{
    node *P;
    P=(node*)malloc(sizeof(node));
    P->data=x;
    P->next=NULL;
    if(empty(qP))
    {
        qP->R=P;
        qP->F=P;
    }
    else
    {
        (qP->R)->next=P;
        qP->R=P;
    }
}
int dequeue(Q *qP)
{
    int x;
    node *P;
    P=qP->F;
    x=P->data;
    if(qP->F==qP->R)  //deleting the last element
        initialise(qP);
    else
        qP->F=P->next;
    free(P);
    return(x);
}
void print(Q *qP)
{
    int i;
    node *P;
    P = qP->F;
    while(P! = NULL)
    {
        printf("\n %d", P->data);
        P = P->next;
    }
}
int empty(Q *qp)
{
    if(qp->R = = NULL)
    return 1;
        return 0;
}
```

Techknowledge Publication

Output

```
Enter 5 elements :  4   5  34   21  90
displaying queue :
4
5
34
21
90
after deletion of two elements :
34
21
90
```

8.3 Circular Queues

Q. What is circular queue? Explain the advantages of circular queue over the linear queue.
SPPU - Dec. 16, 6 Marks

Q. Define the circular queue with example.
SPPU - Dec. 18, 2 Marks

8.3.1 Queue using a Circular Array

Q. What is circular Queue? Explain the advantages of Circular Queue over Linear Queue.
SPPU - May 17, 6 Marks

Q. Give the advantages of circular queue over linear queue. **SPPU - Dec. 17, May 18, 3 Marks**

Q. Define the term circular queue with example.
SPPU - May 18, 2 Marks

Q. Write pseudo C/C++ code to implement a circular queue using arrays.
SPPU - May 19, Dec. 19, 6 Marks

Q. Explain the advantages of Circular Queue over Linear Queue. **SPPU - Dec. 19, 3 Marks**

(Advantage of circular queue)

There is one potential problem with implementation of queue using a simple array.

The queue may appear to be full although there may be some space in the queue.

– After insertion of five elements in the array (a queue) as shown in Fig. 8.3.1,

$$\left.\begin{array}{l}\text{rear} = 4\\ \text{front} = 0\end{array}\right\}\ \text{queue is full}$$

– After two successive deletions

$$\left.\begin{array}{l}\text{rear} = 4\\ \text{front} = 2\end{array}\right\}\ \text{queue is full}$$

– Queue in the Fig. 8.3.1 is full as there is no empty space ahead of rear.

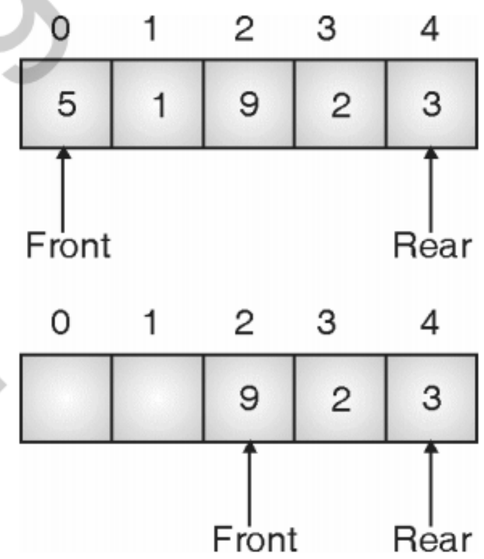


Fig. 8.3.1 : Queue is full, although locations 0 and 1 are vacant

The simple solution is that whenever rear gets to the end of the array, it is wrapped around to the beginning. Now, the array can be thought of as a circle. The first position follows the last element.

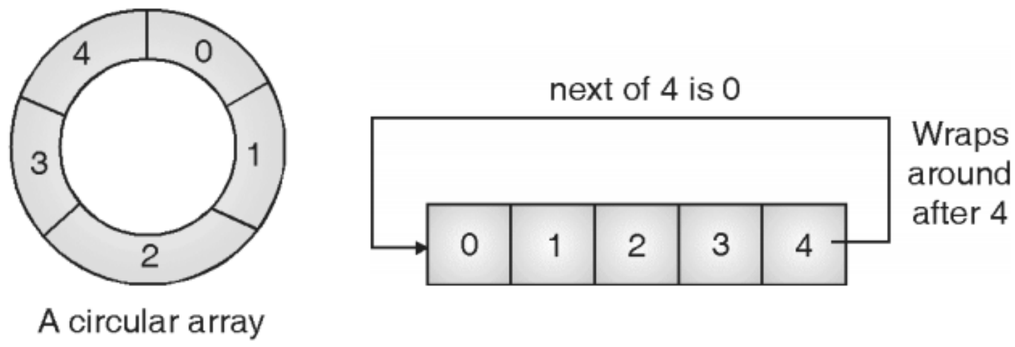


Fig. 8.3.2 : A circular array

In a circular array, the queue is found somewhere around the circle in consecutive positions.

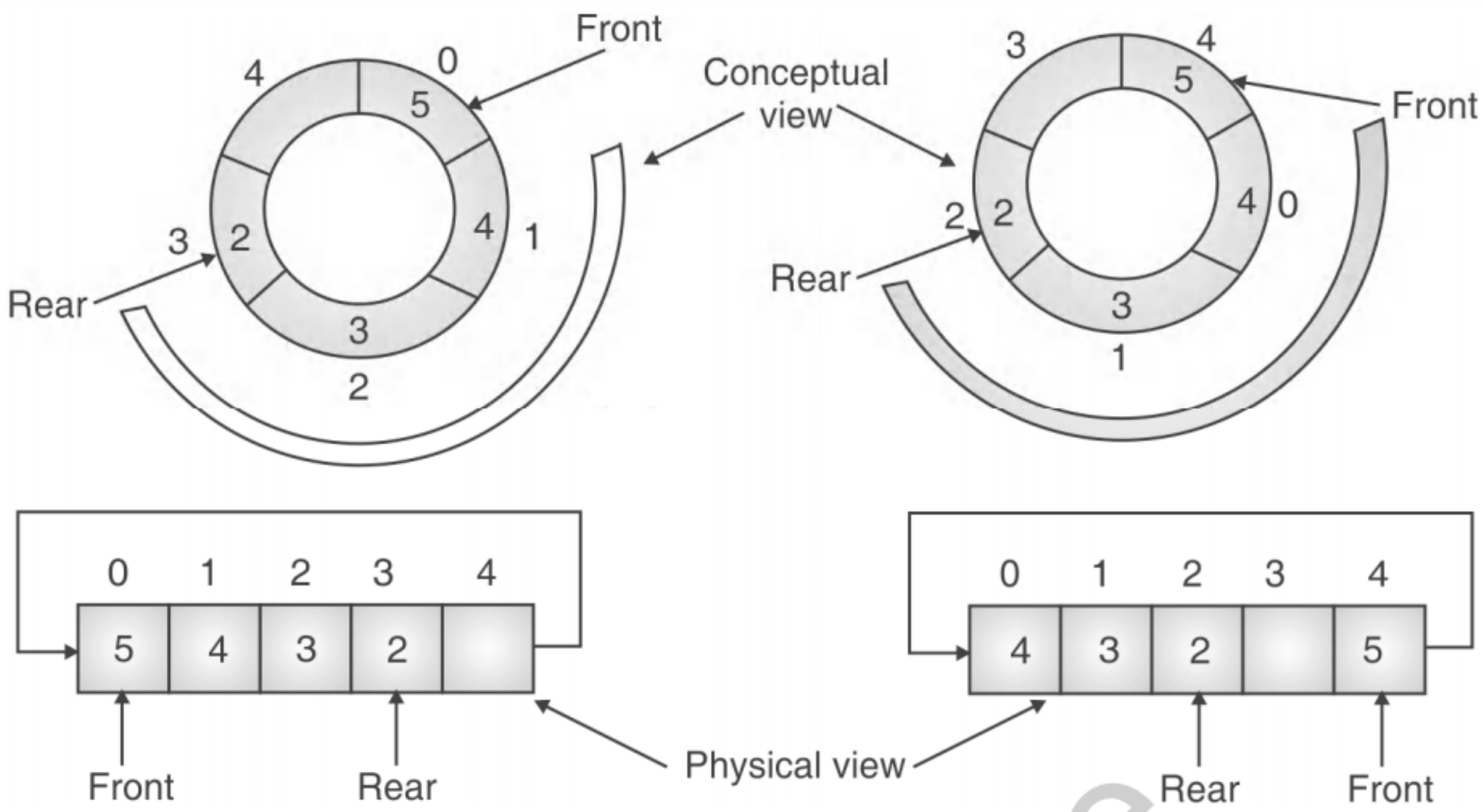


Fig. 8.3.3 : Queue at various places in an array

Advantages of circular queue over linear queue

Sr. No.	Linear queue	Circular queue
1.	Problem of overflow even when there may be space.	No overflow as long as there is space in the queue.
2.	Shifting of data may be needed to handle the issue of false overflow.	Shifting of data is not required.

8.3.1(A) Implementation of a Circular Movement inside a Linear Array

- To give a circular movement inside an array, whenever we go past the last element of the array, it should come back to the beginning of the array.
- Thus, when we move around an array of size five, following sequence for locations should be generated.

0 1 2 3 4 0 1 2 3 4 .....

i= (i + 1)%5 → Array of size 5

i = (i + 1)%MAX → Array of size MAX expression used for circular movement

i	(i + 1)%5		
0	(0 + 1)%5	=	1
1	(1 + 1)%5	=	2
2	(2 + 1)%5	=	3
3	(3 + 1)%5	=	4
4	(4 + 1)%5	=	0 ← i wraps around to 0

```
i = 0
while(1)
i =(i + 1)%5;
```

← Above program segment will continue generating a circular sequence 0 1 2 3 4 0 1 2 3 .....

Data type for queue in a circular array.

```
# define MAX 30
/* A queue with maximum of 30 elements */
typedef struct queue
{
    int data[MAX];
    int front, rear;
}queue;
```

Various operation on the queue in a circular array

'C' function to initialize the queue by setting values of rear and front as –1.

```
void initialize(queue *P)
{
```

Tech Knowledge Publications

```
    P → rear = – 1;
    P → front = –1;
}
int empty(queue *P)
{
    if(P → rear = = – 1)
        return(1);
    return(0);
}
int full(queue *P)
{
    if((P → rear + 1)% MAX = = P → front)
        return(1);
    return(0);
}
```

Fig. 8.3.4 shows, queue is full.

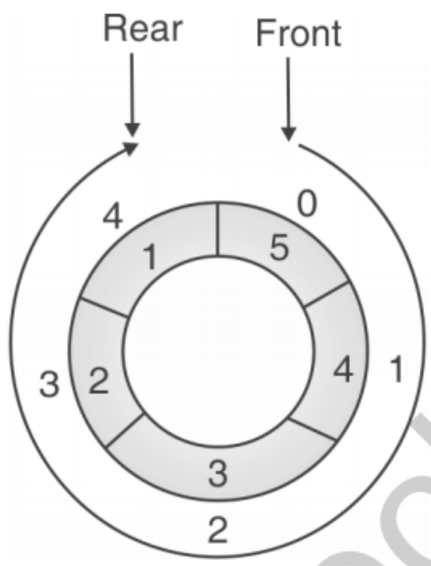


Fig. 8.3.4

When the queue becomes full, there will be no space ahead of rear in the circular array. Hence, front will be found, immediately ahead of rear in a queue that is full.

i.e. (p → rear + 1) % MAX == P → front

```
void enqueue(queue *P, int x)
{
    if(empty(P))                    /* empty queue */
    {
    P → rear = P → front = = 0;
    /* rear and front will point to the same element */
    P → data[P → rear] = x;
    }
    else
    {
        P → rear = (P → rear + 1)% MAX;
        /* Advance P in the circular array */
        P → data[P → rear] = x;
    }
}
int dequeue(queue *P)
{
        int x;
        x = P → data[P → front];
        if(P → rear = = P → front)
        /* deleted the last element */
            initialize(P);
        else
            P → front = (P → front + 1)% MAX;
        /* front advances to the next position in the
        circular array */
        return(x);
}
void print(queue *P)
{
    int i;
    i = P → front;
    while(i ! = P → rear)
    {
        printf("\n %d", P → data[i]);
        i = (i + 1) %MAX;
    }
    printf("\n %d", P → data[P → rear]);
}
```

Program 8.3.1 : Program showing various operations on a circular queue.

OR Write pseudo C/C++ to implement circular queue using arrays. **SPPU - Dec. 16, Dec. 17, 7 Marks**

OR Write pseudo C/C++ code to represent QUEUE as an ADT **SPPU - May 17, 7 Marks**

```
#include <stdio.h>
#include <conio.h>
#define MAX 30
typedef struct queue
{
    int data[MAX];
    int rear, front;
} queue;
```

Techknowledge Publication

```
void initialize(queue *p);
int empty(queue *p);
int full(queue *p);
void enqueue(queue *p, int x);
int dequeue(queue *p);
void print(queue *p);

void main( )
{
    int x, op, n;
    queue q;
    initialize(&q);
    do
    {
        printf("\n 1)create \n 2)insert \n 3)delete
        \n 4)print \n 5)Quit");
        printf("\n enter your choice :");
        scanf("%d", &op);
        switch(op)
        {
    case 1 : printf("\n enter no. of elements :");
            scanf("%d", &n);
            initialize(&q);
            printf("\n enter the data :");
            for(i = 0; i < n; i++)
            {
                scanf("%d", &x);
                if(full(&q))
                {
                    printf("\n queue is full ...");
                    exit(0);
                }
                enqueue(&q, x);
            }
                break;
    case 2 : printf("\n enter the element to be
            inserted :");
            scanf("%d", &x);
            if(full(&q))
            {
                printf("\n queue is full ...");
                exit(0);
            }
            enqueue(&q, x);
                break;
    case 3 : if(empty(&q))
            {
                printf("\n queue is empty ...");
                exit(0);
            }
                x = dequeue(&q);
                printf("\n element = %d", x);
                break;
    case 4 : print(&q);
                break;
        default : break;
        }
    }while(op! = 5);
}
void initialize(queue *P)
{
    P → rear = – 1;
    P → front = –1;
}
int empty(queue *P)
{
    if(P → rear = = – 1)
        return(1);
    return(0);
}
int full(queue *P)
{
    if((P → rear + 1)% MAX = = P → front)
        return(1);
    return(0);
}
void enqueue(queue *P, int x)
{
    if(empty(P))                    /* empty queue */
    {
```

Techknowledge Publication

```
    P → rear = P → front = = 0;
    /* rear and front will point to the same element */
    P → data[P → rear] = x;
    }
    else
    {
        P → rear = (P → rear + 1)% MAX;
        /* Advance P in the circular array */
        P → data[P → rear] = x;
    }
}
int dequeue(queue *P)
{
    int x;
    x = P → data[P → front];
    if(P → rear = = P → front)
        /* deleted the last element */
        initialize(P);
    else
        P → front = (P → front + 1)% MAX;
        /* Front advances to the next position in the
        circular array */
    return(x);
}
void print(queue *P)
{
    int i;
    i = P → front;
    while(i ! = P → rear)
    {
        printf("\n %d", P → data[i]);
        i = (i + 1) %MAX;
    }
    printf("\n %d", P → data[P → rear]);
}
```

Ex. 8.3.1 : Consider a circular queue of size 5 having initial status as :

Front	Rear	Circular queue				
		0	1	2	3	4
1	2		P	Q		

Show the value of front, rear and the contents of circular queue after every step in tabular form for the following operations :

(1) R is added

(2) Delete 2 letters

(3) S, T, U are added

(4) Three letters are deleted

(5) V is added.

Soln. :

Operation	Front	Rear	Circular queue				
			0	1	2	3	4
Initial	1	2		P	Q		
R is added	1	3		P	Q	R	
Delete 2 letters	3	3				R	
S,T,U are added	3	1	T	U		R	S
Three letters are deleted	1	1		U			
V is added	1	2		U	V		

8.3.2 Queue using a Circular Linked List

Q. Explain circular queue using linked list. Write pseudo C code for enqueue operation.

SPPU - Dec. 18, 7 Marks

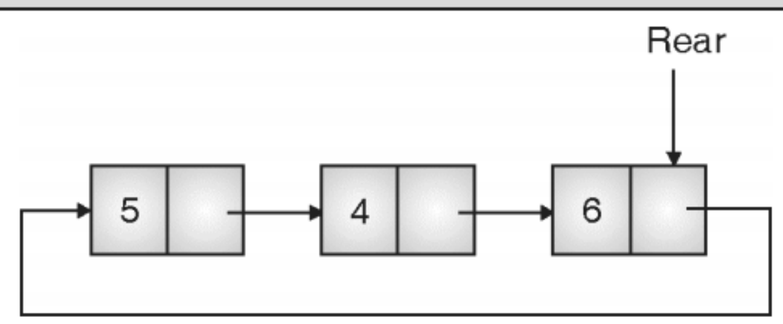


Fig. 8.3.5

In a circular linked list, rear node points back to the front node. Address of the front node can be found through the rear node.

Address of the front node = rear → next

Tech Knowledge Publications

When a queue is implemented through a singly circular linked list, only one pointer i.e. address of the rear node is required to be maintained.

In contrast, when a queue is maintained using a linked list, two pointers must be maintained :

(a) Address of the front node (for deletion)

(b) Address of the rear node (for insertion)

Special care should be taken, while inserting an element in an empty queue. Similarly, after deletion of the last element, the queue should become empty.

Steps for insertion of element x in queue represented using a circular linked list :

(a) Acquire memory for the node

```
P = (node *) malloc(sizeof(node))
```

(b) Store the data x in the node

```
P → data = x;
```

If the queue is empty

(c) newly acquired node should be connected back to itself.

P → next = P;

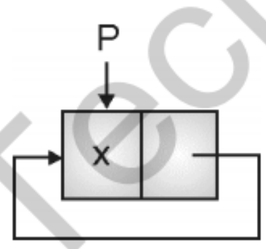


Fig. 8.3.6

(d) front should point to the only element

front = P

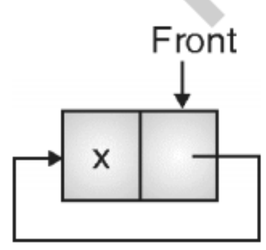


Fig. 8.3.7

If the queue is not empty

(c) P → next = rear → next

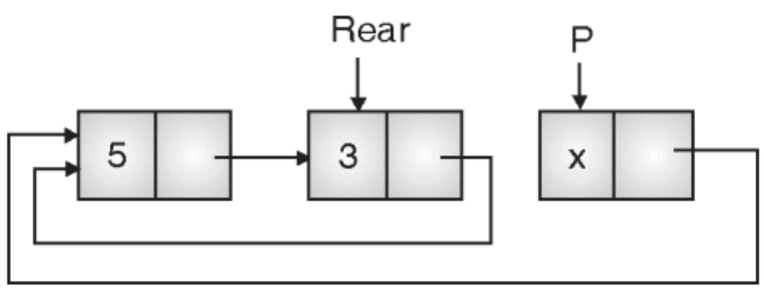


Fig. 8.3.8

(d) rear → next = P

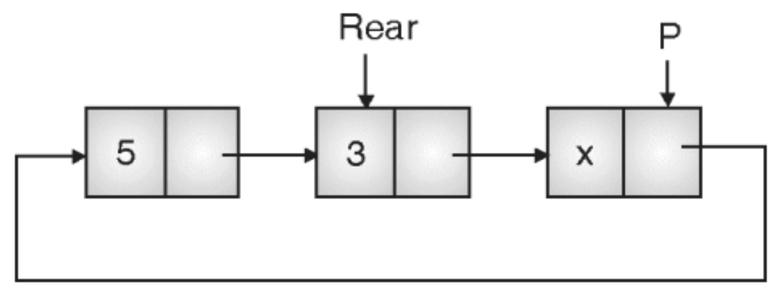


Fig. 8.3.9

(e) rear = P

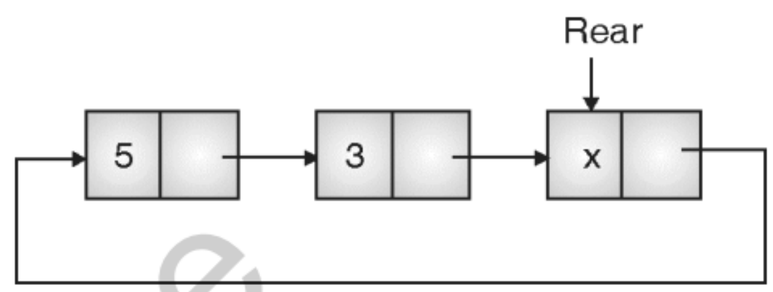


Fig. 8.3.10

'C' function for insertion of an element in a queue represented using a circular linked list.

```
void enqueue(node **R, int x)
//queue is referenced by address of rear node.
{
    node *P;
    P = (node *) malloc(sizeof(node));
    P → data = x;
    if(*R = = NULL)
    {
        P → next = P; *R = P;
    }
    else
    {   P → next = (*R) → next;
        (*R) → next = P;
        *R = P;
    }
}
```

Steps for deletion of an element from a queue represented using a circular linked list.

(a) P = rear → next;

P Point to the node to be deleted

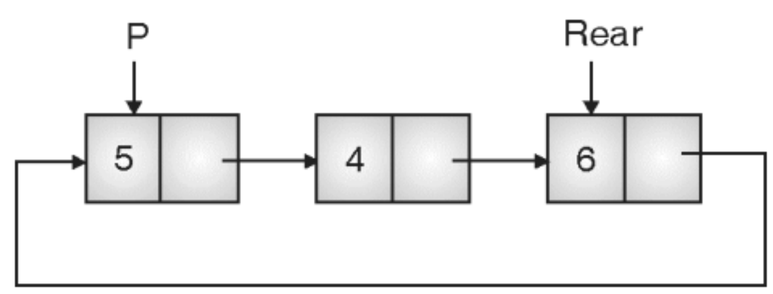


Fig. 8.3.11

Technknowledge Publication

(b) x = P → data;

if (last node is being deleted)

{

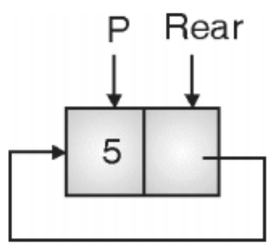


Fig. 8.3.12

(c) release the memory of the node being deleted.

free(P);

(d) Make the queue empty

rear = NULL;

(e) Return the value stored in the front node

return(x);

}

else

{

(c) Remove the front node from the queue

rear → next = P → next

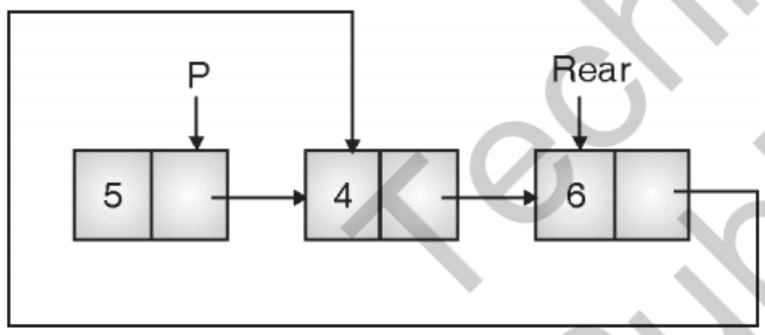


Fig. 8.3.13

(d) release the memory of the node being deleted

free(P)

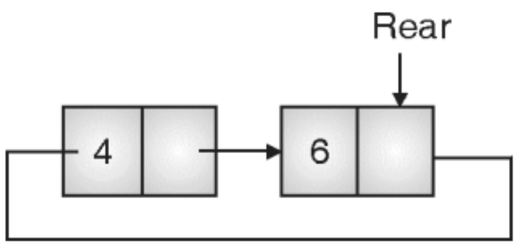


Fig. 8.3.14

(e) return the value stored in the front node

return(x)

}

'C' function for deletion of the front node from a queue represented using a circular linked list.

```
int dequeue(node **R)
//pointer rear is passed by address
{   node *P;
    int x;
    P = (*R) → next;
    /*  *R is same as rear as R contains the address of
        rear. */
    x = P → data;
    if(P → next = = P)
    /* deleting the last node */
    {
        *R = NULL;
        free(P);
        return(x);
    }
    (*R) → next = P → next;
    free(P);
    return(x);
}
```

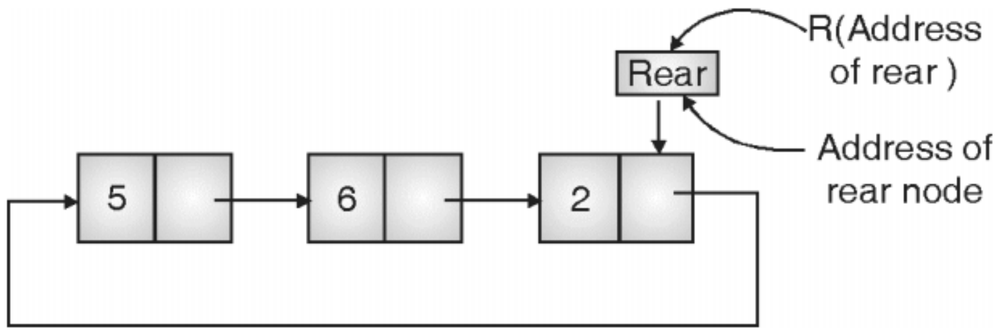


Fig. 8.3.15

In the above function, rear is passed by address as after deletion, rear may change. Receiving variable in function int dequeue(node **R) is declared as node **R. R contains the address of rear and hence, *R can be used in place of 'rear'.

'C' function for printing elements of a queue represented using a circular linked list.

```
void Print(node * rear)
{
```

Technknowledge Publication

```
 node P;
 P = rear → next;
/* start printing from the front */
do
{
    printf("\n %d", P → data);
    P = P → next;
  } while(P! = rear → next);
}
```

In case of a circular linked list, the starting case and the termination case for the loop used for traversal of the linked list are identical.

(1) We start printing from the front node.

(2) We terminate printing on reaching the front node.

Such cases are best handled through do-while loops.

Program 8.3.2 : Program for showing various operations on a queue represented using circular linked list.

```
#include<stdio.h>
#include<conio.h>
typedef struct node
{
  int data;
  struct node *next;
}node;
void init(node **R);
void enqueue(node **R,int x);
int dequeue(node **R);
int empty(node *rear);
void print(node *rear);
void main()
{
  int x,option;
  int n = 0,i;
  node *rear;
  init(&rear);
  clrscr();
  do
  {
    printf("\n 1. Insert\n 2. Delete\n 3. Print\n 4. Quit");
    printf("\n your option:    ");
    scanf("%d",&option);
    switch(option)
    {
      case 1 :
      printf("\n Number of Elements to be inserted");
      scanf("%d",&n);
      for(i=0; i<n; i++)
      {
          scanf("\n %d",&x);
          enqueue(&rear,x);
      }
      break;
      case 2 :
      if(! empty(rear))
      {
        x = dequeue(&rear);
        printf("\n Element deleted = %d",x);
      }
      else
      printf("\n Underflow..... Cannot deleted");
        break;
      case 3 :   print(rear);
        break;
    }
  }while(option != 4);
 getch();
}
void init(node **R)
{
    *R = NULL;
}
void enqueue(node **R,int x)
{
  node *p;
  p = (node *)malloc(sizeof(node));
  p->data = x;
```

Techknowledge Publications

```
    if(empty(*R))
    {
        p->next = p;
        *R = p;
    }
    else
    {
        p->next = (*R)->next;
        (*R)->next = p;
        (*R) = p;
    }
}
int dequeue(node **R)
{
    int x;
    node *p;
    p = (*R)->next;
    p->data = x;
    if(p->next == p)
    {
            *R = NULL;
            free(p);
            return(x);
    }
    (*R)->next = p->next;
    free(p);
    return(x);
}
void print(node *rear)
{
    node *p;
    if(!empty(rear))
    {
            p = rear->next;
    }
    p = p->next;
    do
    {
            printf("\n %d",p->data);
            p = p->next;
    }while(p != rear->next);
}
int empty(node *P)
{
    if(P->next== –1)
        return(1);
    return(0);
}
```

Output

```
1. Insert
2. Delete
3. Print
4. Quit
your option:    1
Element to be inserted  4
        12 23 34 45
1. Insert
2. Delete
3. Print
4. Quit
your option:    3
        12 23 34 45
1. Insert
2. Delete
3. Print
4. Quit
your option:    2
Element deleted = 4
1. Insert
2. Delete
3. Print
4. Quit
your option:    3
        23 34 45
1. Insert
2. Delete
3. Print
4. Quit
your option:    4
```

Techknowledge Publication

8.4 Dequeues

Q. Write pseudo C/C++ code to represent dequeue and perform the following operations :

(1) Create Dequeue

(2) Insert

(3) Delete

(4) Display **SPPU - Dec. 16, 7 Marks**

Q. Define the term dequeue with example.

SPPU - Dec. 17, Dec. 18, Dec. 19, 2 Marks

Q. Explain Dequeue with the insert and delete operations performed on it.

SPPU - May 19, 7 Marks

The word **dequeue** is a short form of **double ended queue.** It is general representation of both stack and queue and can be used as stack and queue. In a dequeue, insertion as well deletion can be carried out either at the rear end or the front end. In practice, it becomes necessary to fix the type of operation to be performed on front and rear end. Dequeue can be classified into two types :

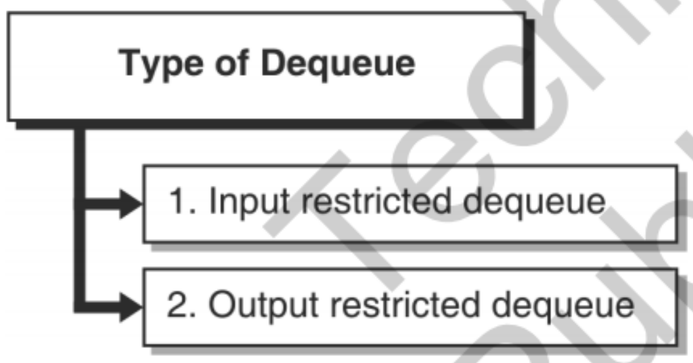


Fig. 8.4.1: Type of Dequeue

1. Input restricted dequeue

The following operations are possible in an input restricted dequeue :

(i) Insertion of an element at the rear and

(ii) Deletion of an element from front end

(iii) Deletion of an element from rear end

2. Output restricted dequeue

The following operations are possible in an output restricted dequeue.

(i) Deletion of an element from front end

(ii) Insertion of an element at the rear end

(iii) Insertion of an element at the front end

There are various methods to implement a dequeue.

(a) Using a circular array

(b) Using a singly linked list.

(c) Using a singly circular linked list.

(d) Using a doubly linked list.

(e) Using a doubly circular linked list.

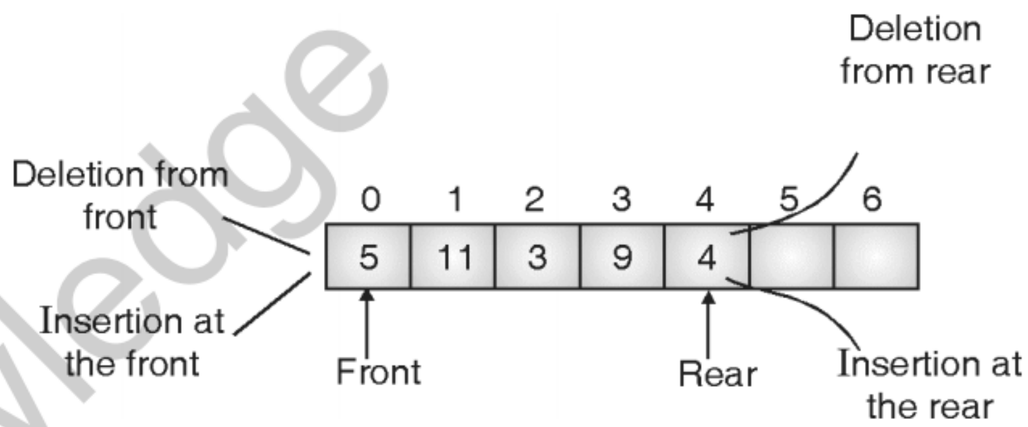


Fig. 8.4.2 : A dequeue in a circular array

Operations associated with dequeue :

(a) empty( ) : Whether the queue is empty ?

(b) full( ) : Whether the queue is full ?

(c) initialize( ) : Make the queue empty

(d) enqueueR( ) : Add item at the rear end of the queue.

(e) enqueueF( ) : Add item at the front end of the queue.

(f) dequeueR( ) : Delete item from the rear end of the queue.

(g) dequeueF( ) : Delete item from the front end of the queue.

Timing complexity of various dequeue operations :

enqueueR( ) – 0(1) – constant time.

enqueueF( ) – 0(1) – constant time.

dequeueR( ) – 0(1) – constant time.

dequeueF( ) – 0(1) – constant time.

Advantage of dequeue : The dequeue is a general representation of both stack and queue it can be used both as stack or a queue.

Technknowledge Publication

Dequeue as an ADT

(I) Data type for dequeue in an array

```
#define MAX 30
/* A queue with maximum of 30 elements */
typedef struct DQ
{
    int data [MAX];
    int rear, front;
} DQ;
```

(II) Operations on a dequeue

(i) **initialize( ) :** Make the queue empty.

(ii) **empty( ) :** Determine if queue is empty.

(iii) **full( ) :** Determine if queue is full.

(iv) **enqueueF( ) :** Insert an element at the front end of the queue.

(v) **enqueueR( )** : Insert an element at the rear end of the queue.

(vi) **dequeueR( ) :** Delete the rear element.

(vii) **dequeueF( ) :** Delete the front element.

(viii) **print( ) :** Print elements of the queue.

(III) Prototype of functions used for various operations on queue

(i) void intialize(DQ *p);

(ii) int empty(DQ *p);

Function returns 1 or 0, depending on whether the queue pointed by p is empty or not.

(iii) int full(DQ *p);

Function returns 1 or 0, depending on whether the queue pointed by p is full or not.

(iv) void enqueueF(DQ *p, int x);

(v) void enqueueR(DQ *p, int x);

(vi) int deleteR(DQ *p);

(vii) int deleteF(DQ *p);

(viii) void print(DQ *p);

(ix) enqueueR( ) and enqueueF( ) will cause an overflow if the queue is full.

(x) dequeueR( ) and dequeueF( ) will cause an underflow if the queue is empty.

8.4.1 Implementation of Dequeue using a Circular Array

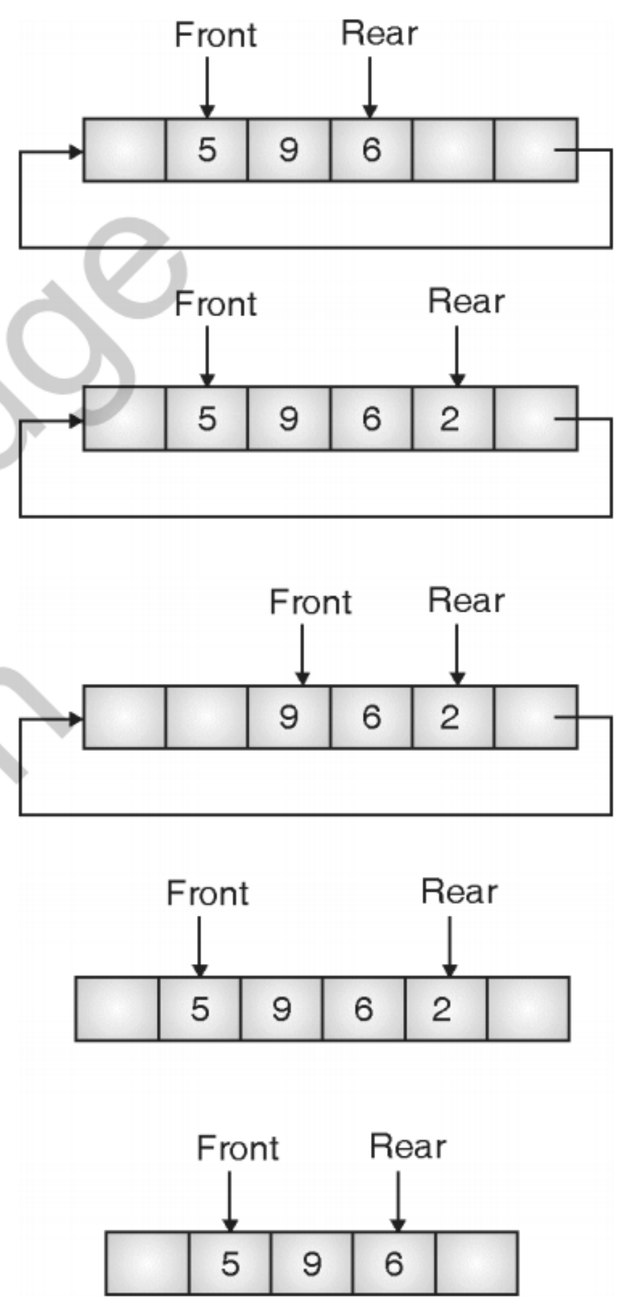


Fig. 8.4.3 : Various operations on dequeue and movement of front and rear in clock-wise or anti-clockwise direction

Expression for clock-wise movement

rear = (rear + 1) % MAX

where MAX is the size of the array.

front = (front + 1) % MAX

Expression for anti-clockwise movement :

rear = (rear – 1 + MAX) % MAX

front = (front – 1 + MAX) % MAX

Tech Knowledge Publications

'C' functions for various operations on a dequeue represented using a circular array.

A dequeue can be represented using the following data type.

```
typedef struct dequeue
 {
    int data[MAX];
    int front, rear;
 } dequeue;
```

(1) Initialize the queue by setting values of rear and front as – 1.

```
void initialize(dequeue *P)
{
    P → rear = –1;
    P → front = –1;
}
```

(2) Test, whether the dequeue is empty ?

```
int empty(dequeue *P)
{
   if(P → rear == –1)
       return(1);
   return(0);
}
```

(3) Test, whether the dequeue is full ?

```
int full(dequeue *P)
{
   if((P → rear + 1) % MAX ==P → front)
       return(1);
   return(0);
}
```

(4) Add item at the rear end of the dequeue.

```
void enqueueR(dequeue *P, int x)
{
    if(empty(P))
    {
        P → rear = 0;
        P → front = 0;
        P → data[0] = x;
      }
    else
    {
        P → rear = (P → rear + 1) % MAX;
        P → data[p → rear] = x;
    }
}
```

(5) Add item at the front end of the dequeue on addition of item at the front end, front will move in anti-clockwise direction.

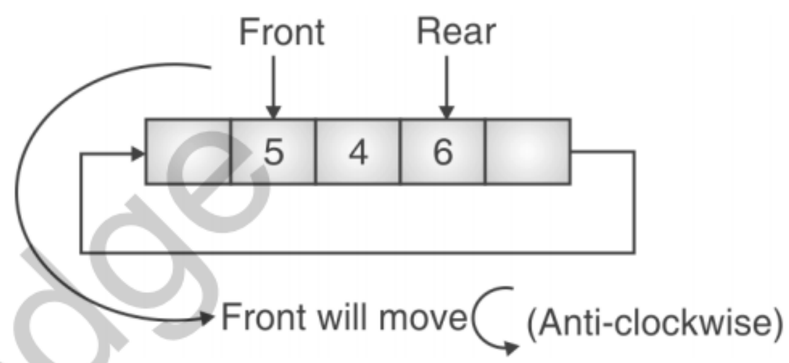


Fig. 8.4.4

```
void enqueueF(dequeue *P, int x)
 {
   if(empty(P))
   {
       P → rear = 0;
       P → front = 0;
       P → data[0] = x;
   }
   else
   {
       P → front = (P → front – 1 + MAX) % MAX;
       P → data[P → front] = x;
   }
}
```

(6) Delete an item from the front end of the dequeue :

```
int dequeueF(dequeue *P)
{
    int x;
    x = P → data[P → front];
    if(P → rear = = P → front)
    /* delete the last element */
        initialize(P);
    else
        P → front = (P → front + 1) % MAX;
    return(x);
}
```

Techknowledge Publications

(7) Delete an item from the rear end of the dequeue :

```
int dequeueR(dequeue *P)
{
    int x;
    x = P → data[P → rear];
    if(P → rear = = P → front)
        initialize(P);
    else
        P → rear = (P → rear – 1 + MAX) % MAX;
    return(x);
}
```

Program 8.4.1 : A program showing various operations on a dequeue represented using a circular queue.

```
#include <stdio.h>
#include <conio.h>
#define MAX 30
typedef struct dequeue
{   int data [MAX];
    int rear, front;
} dequeue;
void initialize(dequeue *p);
int empty(dequeue *p);
int full(dequeue *p);
void enqueueR(dequeue *p, int x);
void enqueueF(dequeue *p, int x);
int dequeueF(dequeue *p);
int dequeueR(dequeue *p);
int print(dequeue *p);
void main( )
{
    int x, op, n;
    dequeue q;
    initialize(&q);
    do
    {
    printf("\n1)create    \n2)insert(rear)\n3)insert(front)
    \n4)Delete(rear)\n5)Delete(front)");
    printf("\n6)print \n7)Quit");
    printf("\n enter your choice :");
    scanf("%d", &op);
    switch(op)
    {
    case 1 :
        printf("\n enter no. of elements :");
        scanf("%d", &n);
        initialize(&q);
        printf("\n enter the data :");
        for(i = 0; i< n; i++)
        {
            scanf("%d", &x);
            if(full(&q))
            {
                printf("\n queue is full …");
                exit(0);
            }
            enqueueR(&q, x);
        }
        break;
    case 2 :
        printf("\n enter element to be inserted:");
        scanf("%d", &x);
        if(full(&q))
        {
            printf("\n queue is full …");
            exit(0);
        }
        enqueR(&q, x);
        break;
    case 3 :
        printf("\n enter the element to be inserted :");
        scanf("%d", &x);
        if(full(&q))
        {
            printf("\n queue is full …");
            exit(0);
        }
            enqueueF(&q, x);
            break;
    case 4 : if(empty(&q))
            {
```

Techknowledge Publication

```
            printf("\n queue is empty ...");
            exit(0);
        }
            x = deleteR(&q);
            printf("\n element = %d", x);
            break;
    case 5 : if(empty(&q))
        {
            printf("\n queue is empty ...");
            exit(0);
        }
        x = deleteF(&q);
        printf("\n element = %d"  , x);
        break;
    case 6 : printf(&q); break;
    default : break;
    }while(op! = 7);
}
void initialize(dequeue *P)
{
    P → rear = –1;
    P → front = –1;
}
int empty(dequeue *P)
{
    if(P → rear = = –1)
        return(1);
    return(0);
}
int full(dequeue *P)
{
    if((P → rear + 1) % MAX = = P → front)
        return(1);
    return(0);
}
void enqueueR(dequeue *P, int x)
{
    if(empty(P))
    {
        P → rear = 0;
        P → front = 0;
        P → data[0] = x;
    }
    else
    {
        P → rear = (P → rear + 1) % MAX;
        P → data[p → rear] = x;
    }
}
void enqueueF(dequeue *P, int x)
{
    if(empty(P))
    {
        P → rear = 0;
        P → front = 0;
        P → data[0] = x;
    }
    else
    {
        P → front = (P → front – 1 + MAX) % MAX;
        P → data[P → front] = x;
    }
}
int dequeueF(dequeue *P)
{
    int x;
    x = P → data[P → front];
    if(P → rear = = P → front)
    /* delete the last element */
        initialize(P);
    else
        P → front = (P → front + 1) % MAX;
    return(x);
}
int dequeueR(dequeue *P)
{
    int x;
    x = P → data[P → rear];
    if(P → rear = = P → front)
        initialize(P);
```

Techknowledge Publication

```
    else
        P → rear = (P → rear – 1 + MAX) % MAX;
        return(x);
}
void print(dequeue *p)
{
    int i;
    i = p → front;
    while(i! = p → rear)
    {
        printf("\n %d", p → data[i]);
        i = (i + 1)% MAX;
    }
    printf("\n %d", p → data [p → rear]);
}
```

8.5 Priority Queue

Q. Explain array implementation of priority queue with all basic operations.
SPPU - May 17, 6 Marks

Q. Define the term priority queue with example.
SPPU - Dec. 17, May 18, Dec. 19, 2 Marks

Q. Explain priority queue.
SPPU - Dec. 17, Dec. 18, Dec. 19, 2 Marks

Q. What is Priority queue ? Describe the operations on priority queue and explain its applications.
SPPU - May 19, 7 Marks

Priority queue is an ordered list of homogeneous elements. In a normal queue, service is provided on the basis of First-in-first-out. In a priority queue service is not provided on the basis of "first-come-first-served" but rather than each element has a priority based on the urgency of need.

- An element with higher priority is processed before other elements with lower priority.
- Elements with the same priority are processed on "first-come-first served" basis.

An example of priority queue is a hospital waiting room. A patient having a more fatal problem will be admitted before other patients.

Other applications of priority queues is found in long term scheduling of jobs processed in a computer. In practice, short processes are given a priority over long processes as it improves the average response of the system.

There are two types of priority queues : Ascending priority queue and a descending priority queue.

If the priority queue is ascending, the smallest priority element is deleted first. And if it is descending, the largest priority element is deleted first.

8.5.1 Implementation of Priority Queues

Q. Write pseudo C/C++ code to implement priority queue operations.
SPPU - May 18, 7 Marks

Q. Write ADT for priority queue and state its applications.
SPPU - Dec. 18, 5 Marks

Q. Give pseudo C/C++ code for the array implementation of a priority queue.
SPPU - Dec. 19, 4 Marks

Priority queue can be implemented using a circular array :

As the service must be provided to an element having highest priority, there could be a choice between :

(a) List is always maintained sorted on priority of elements with the highest priority element at the front. Here, deletion is trivial but insertion is complicated as the element must be inserted at the correct place depending on its priority.

(b) List is maintained in the "FIFO" form but the service is provided by selecting the element with highest priority. Deletion is difficult as the entire queue must be traversed to locate the element with highest priority. Here, insertion is trivial (at rear end).

Tech Knowledge Publications

Implementation of a Priority Queue using a circular array (Priority queue as an ADT)

Data type for priority queue in a circular array

```
# define MAX 30
/* A queue with maximum of 30 elements */
typedef struct pqueue
{
    int data[MAX];
     int front, rear;
} pqueue;
```

Operations on a priority queue

(i) initialize( ) : Make the queue empty.

(ii) empty( ) : Determine if the queue is empty.

(iii) full( ) : Determine if the queue is full.

(iv) enqueue( ) : Insert an element as per its priority.

(v) dequeue( ) : Delete the front element (front element will have the highest priority)

(vi) print( ) : Print elements of the queue.

Prototype of functions used for various operations on the queue

(i) void initialize(pqueue *p);

(ii) int empty(pqueue *p);

(iii) int full(pqueue *p);

(iv) void enqueue(pqueue *p, int x);

(v) int dequeue(pqueue *p);

(vi) void print(pqueue *p);

(vii) enqueue( ) operation will cause an overflow if the queue is full.

(viii) Dequeue( ) operation will cause an underflow if the queue is empty.

C-implementation of functions

```
void initialize(pqueue *p)
{   p → rear = – 1;
    p → front = – 1;
}
```

```
/* A value of rear or front as –1, indicates that the
   queue is empty.*/
int empty(pqueue *p)
{
    if(P->rear = = – 1)
        return(1);    /* queue is empty */
    return(0);        /* queue is not empty */
}
int full(pqueue *p)
{
    /* if front is next to rear in the circular array then
    the queue is full */
    if(p → rear + 1)% MAX = = p → front)
        return(1); /* queue is full */
    return(0);
}
void enqueue(pqueue *p, int x)
{
    int i;
    if(full(p))
        printf("\n overflow …");
    else
    {   /* inserting in an empty queue */
        if(empty(p))
        {   p → rear = p → front = 0;
            p → data [0] = x;
        }
        else
        {
/* move all lower priority data right by one place */
        i = p → rear ;
    while(x > p → data [i])
    {   p → data [(i + 1)%MAX] = p → data [i];
        /* position i on the previous element */
        i = (i – 1 + MAX) % MAX;
/* anti-clockwise  movement inside the queue */
        if((i + 1)% MAX = = p → front)
            break;
    /* if all elements have been moved */
    }
```

Techknowledge Publication

```
        /* insert x */
        i = (i + 1)% MAX;
        p → data [i] = x;
        /* re-adjust rear */
      p → rear = (p → rear + 1) % MAX;
    }
  }
}
int dequeue(pqueue *p)
{
    int x;
    if(empty(p))
        printf("\n underflow …");
    else
    {
        x = p → data [p → front];
        if(p → rear = = p → front)
        /* delete last element */
            initialize(p);
        else
            p → front = (p → front + 1) % MAX;
    }
    return(x);
}
void print(pqueue *p)
{
    int i, x;
    i = p → front;
    while( i! = p → rear)
    {
        x = p data [i];
        printf("\n %d", x);
        i = (i + 1) % MAX;
    }
    /* print the last data */
        x = p → data [i];
        printf("\n %d", x);
}
```

Program 8.5.1 : Program showing various operations on a priority queue.

OR Give pseudo C/C++ code for array implementation of priority queue. **SPPU - Dec. 17, 5 Marks**

```
#include <stdio.h>
#include <conio.h>
#define MAX 30
typedef struct pqueue
{
    int data [MAX];
    int rear, front;
} pqueue;
void initialize(pqueue *p);
int empty(pqueue *p);
int full(pqueue *p);
void enqueue(pqueue *p, int x);
int dequeue(pqueue *p);
void print(pqueue *p);
void main( )
{
    int x, op, n;
    pqueue q;
    initialize(&q);
do
{
    printf("\n1)create\n 2)insert\n 3)Delete\n 4)print\n
    5)Quit");
    printf("\n enter your choice:");
    scanf("%d", &op);
    switch(op)
    {
    case 1 :
        printf("\n enter no. of elements :");
        scanf("%d", &n);
        initialize(&q);
        printf("enter the data:");
        for(i = 0; i < n; i++)
        {
            scanf("%d" &x);
            if(full(&q))
            {
                printf("\n queue is full …");
                exit(0);
```

Techknowledge Publication

```
            }
            enqueue(&q, x);
        }
        break;
    case 2 :
        printf("\n enter the element to be inserted");
        scanf("%d", &x);
        if(full(&q))
        {
            printf("\n queue is full ...");
            exit(0);
        }
        enqueue(&q, x);
        break;
    case 3 :
        if(empty(&q))
         {
                printf("\n queue is empty ...");
                exit(0);
         }
            x = dequeue(&q);
            printf("\n element = %d", x);
            break;
    case 4 :    print(&q);
            break;
    default :   break;
    }
  } while(op! = 5);
}
void initialize(pqueue *p)
{
    p → rear = – 1;
    p → front = – 1;
}
/* A value of rear or front as –1, indicate that the
    queue is empty. */
int empty(pqueue *p)
{
    if(P->rear = = – 1)
        return(1);    /* queue is empty */
    return(0);    /* queue is not empty */
}
int full(pqueue *p)
{
    /* if front is next rear in the circular array then the
        queue is full */
    if(p → rear + 1)% MAX = = p → front)
        return(1); /* queue is full */
    return(0);
}
void enqueue(pqueue *p, int x)
{   int i;
    if(full(p))
    printf("\n overflow ...");
    else
    {   /* inserting in an empty queue */
        if(empty(p))
        {
            p → rear = p → front = 0
            p → data [0] = x;
        }
        else
        {
/* move all lower priority data right by one place */
        i = p → rear ;
        while(x > p → data [i])
        {   p → data [(i + 1)%MAX] = p → data [i];
            /* position i on the previous element */
            i = (i – 1 + MAX) % MAX;
    /* anticlock wise movement inside the queue */
        if((i + 1)% MAX = = p → front)
          break; /* if all elements have been moved */
        }
        /* insert x */
        i = (i + 1)% MAX;
        p → data [i] = x;
        /* re-adjust rear */
        p → rear = (p → rear + 1) % MAX;
    }
  }
}
```

Techknowledge Publications

```
int dequeue(pqueue *p)
{
    int x;
    if(empty(p))
        printf("\n underflow ...");
    else
    {
        x = p → data [p → front];
        if(p → rear = = p → front)
        /* delete last element */
            initialize(p);
        else
            p → front = (p → front + 1) % MAX;
    }
        return(x);
}
void print(pqueue *p)
{
    int i, x;
     i = p → front;
    while( i! = p → rear)
    {
        x = p data[i];
        printf("\n %d", x);
        i = (i + 1) % MAX;
    }
    /* print the last data */
    x = p → data[i];
    printf("\n %d", x);
}
```

8.6 Representation of Multiple Queues in an Array

Q. Define the Multi-queue with example.

SPPU - Dec. 18, 2 Marks

Several queues can be stored in a single array. Suppose A[ ] is an array of size MAX and N number of queues are to be stored in the array. MAX is a predefined constant.

We can divide the available memory A[0 ... MAX–1] into N segments. Each queue can be allocated one of these segments. For each Queue i, as shown in the Fig. 8.6.1, R[i] and F[i] represent the indices of rear and front of queue i.

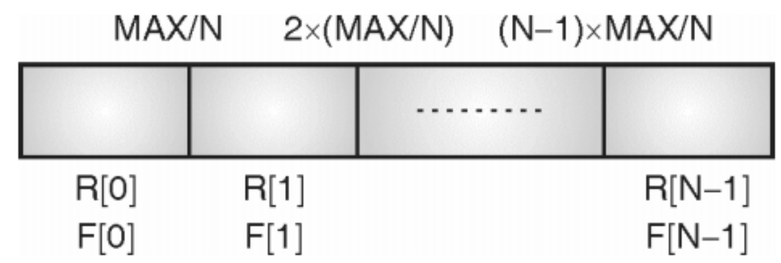


Fig. 8.6.1

Initial values of R[i] and F[i] is calculated as given below.

```
for(i = 0; i < N; i++)
R[i] = F[i] = (MAX/N)*i;
```

Insert( ) and Delete( ) can be performed on any of the queues. Both insert( ) and Delete( ) have an additional parameter, queue number.

'C' function for insert and delete.

```
void insert(int queue_no, int x)
{
    int i, rear;
    i = queue_no;
    if(R[i] = = F[i+1])
        printf("\n Overflow");
    else
    {
        R[i] = R[i]+1;
        rear = R[i];
        A[rear] = x;
    }
}
int delete(int queue_no)
{
    int i, front, x;
    i = queue_no;
    if(R[i] = = F[i])
    {
        printf("\n Underflow");
        return(–1);
    }
    else
```

Techknowledge Publications

```
    {
        front = F[i];
        x = A[front];
        F[i] = F[i]+1;
        return(x);
    }
}
```

'Queue full' condition for queue i does not imply that there is no free space in the array A[ ]. In fact, there could be a lot of unused space between queue j and queue j + 1. We could design an algorithm that will permit insert( ) operation in the queue i to use unused space of array A.

Algorithm for creating space between queue i and queue i + 1.

(a) Determine the least j, i < j < n such that there is free space between queue j and queue j + 1. Now move all queues starting from queue number i + 1 to j, right by one place. This will create a space between queue i and queue i + 1.

(b) If there is no such j in (a) then find the largest j such that 0 ≤ j < i and there is space between queue j and queue j + 1. Now, move all queues starting from queue number j + 1 to number i, left by one place. This will create a free space between queue i and queue i + 1.

8.7 Applications of Queue

Q. Explain applications of priority queue in detail.

SPPU - Dec. 16, 6 Marks

Queue is a very useful data structure. Various features of operating system are implemented using a queue.

(a) Scheduling of processes (Round Robin Algorithm)

(b) Spooling (to maintain a queue of jobs to be printed)

(c) A queue of client processes waiting to receive the service from the server process.

(d) Various application software using non-linear data structure tree or graph requires a queue for breadth first traversal.

(e) Simulation of a real life problem with the purpose of understanding its behaviour. The probable waiting time of a person at a railway reservation counter can be found through the technique of computer simulation if the following concerned factors are known :

(1) Arrival rate

(2) Service time

(3) Number of service counters.

We will discuss some of the **applications** in detail :

(1) Josephus problem

(2) Job scheduling

(3) Queue simulation.

8.7.1 Josephus Problem

This is an interesting programming problem known in the literature as Josephus problem. The problem is given as :

- Suppose there are n children standing in a queue.
- Students are numbered from 1 to n in the clockwise direction.
- Students choose a lucky number say m.

They start counting in clock-wise direction from the child designated as 1. The counting proceeds until the mth child is identified. m^{th} child is eliminated from the queue. Counting for the next round begins from the child next to the eliminated one and proceeds until the mth child is identified. This child is then eliminated and the process continues. After few rounds of counting only one child is left and this child is declared as winner.

Tech Knowledge Publications

Implementation

It is required to write a program to identify a winner of the game. Input to the function are two parameters.

n : number of children

m : lucky number

function returns on integer identifying the winner.

```
int winner(int m, int n)
{   queue q;
    int i;
    init(&q);
    /*  create a queue of n integers, numbered from
        1 to n.   Number i stands for ith child*/
    for(i = 1; i< = n; i++)
        enqueue(&q, i);
    for(j = 1; j<n; j++)
    {/* n–1 iterations to eliminate n–1  children*/
        for(i = 1; i<m; i++)  /* skip m–1 children*/
        {
            x = dequeue(&q);
            enqueue(&q, x);
        }
        x = dequeue(&q); /* remove mth children*/
        }
        x = dequeue(&q);        /*last child winner*/
    return(x);
}
```

8.7.2 Job Scheduling

Programs (jobs) entering a computer for execution are scheduled using some strategies. The purpose is to improve :

(i) CPU utilization (ii) Response time

(iii) System utilization

The jobs entering the system are stored in the queue. These jobs can be selected for execution as per some pre-conceived strategy.

We will discuss one of the scheduling methods.

Round Robin technique

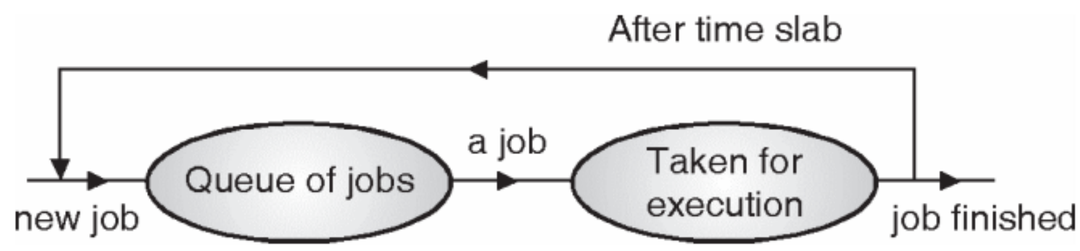


Fig. 8.7.1

Suppose there are n jobs waiting for execution. In Round Robin technique, CPU gives a fixed quantum of time to each job.

A job is selected from the queue of jobs.

It is executed for a fixed quantum of time. If the job is not completed by this time then it is removed and pushed back in queue of jobs.

Let us try to understand it with the help of an example.

The Table 8.7.1 shows the jobs and the required run time.

Table 8.7.1

Job no.	Run time
1	2 units
2	1 unit
3	2 units

Quantum of time = 1 unit

Initial status of queue

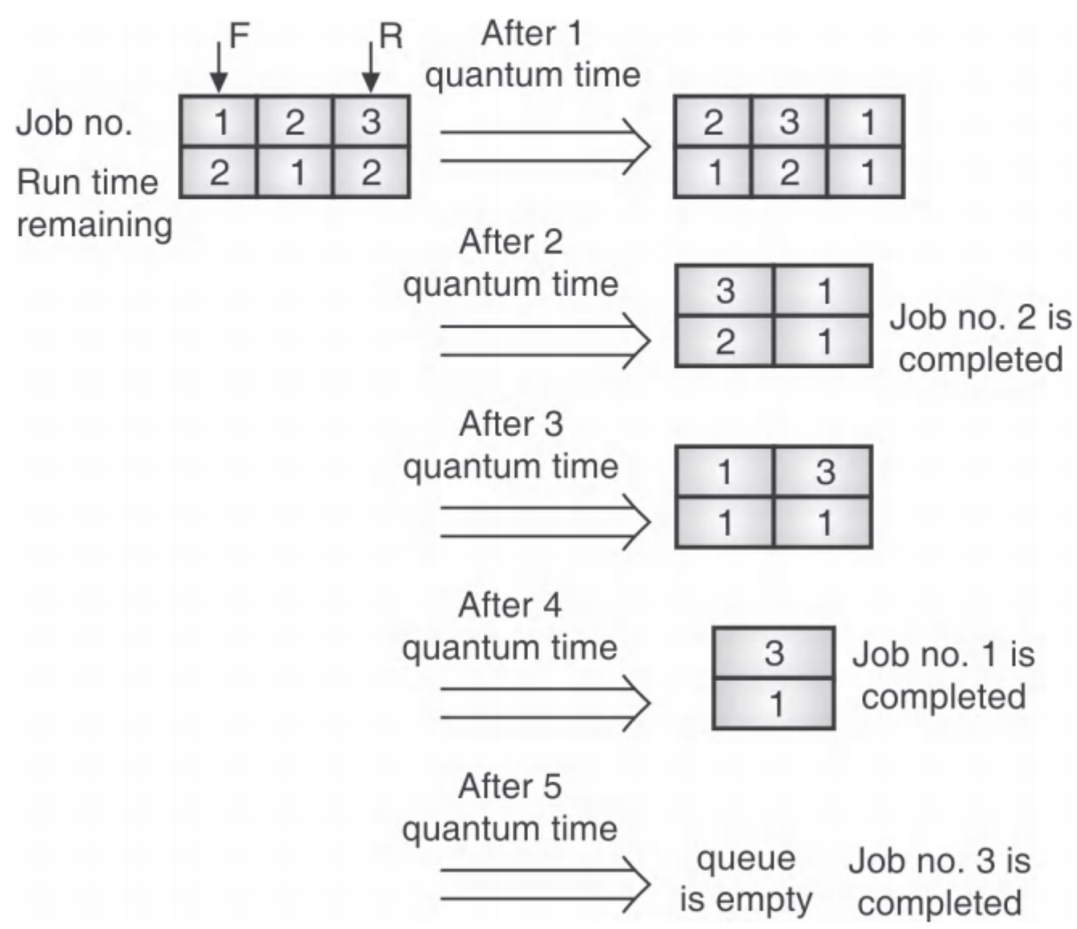


Fig. 8.7.2

Techknowledge Publication

```
void RR(Queue q)
{   /*  q is a queue of jobs */
    /*  Each job has two fields  1 . jobno & 2. execution
        time */
    job t;
    int quantum = 1;
    int time = 0;
    while(! empty(q))
    {
        t = dequeue(q);
        time = time + quantum;
        t.execution_time – = quantum;
        if(t.execution_time > 0)
            enqueue(q, t);
        else
            print t.jobno, time;
    }
}
```

8.7.3 Queue Simulation

Queuing system deals with computing probabilistically, how long users expect to wait on a line, how long the line gets. Waiting time and length of queue depends on :

(1) Arrival rate of users

(2) Service time.

These parameters are given as probabilistic distribution function. We might be interested in finding how long on average a customer has to wait.

Computer can be used to simulate the behavior of a queue. A bank can utilize this information to determine how many cash counters are needed to ensure reasonable service time.

A typical queue simulation involves processing of events :

(1) Customer arriving

(2) Customer departing

On arrival, a customer joins the queue and on departure he leaves the queue.

We can use the probabilistic function to generate a stream of ordered pairs. Each pair consists of :

(1) Arrival time

(2) Service time.

These pair are enqueued. A simulated clock can be used to give the concept of time. At each tick of the clock, all pending events are processed.

If the event is arrival, we check to see if a cash counter is free. If there is none, we place the arrival on the queue.

The waiting line of customers can be implemented as a queue. A cash counter can be implemented as a shared resource. Whenever the cash counter becomes free, a customer from the queue can be assigned to the shared resource (cash counter).

8.8 Case Study - Priority Queue in Bandwidth Management

Weighted fair queueing offers dynamic fair queueing that divides bandwidth across queues of traffic based on weights. This offers a satisfactory response time to critical applications.

In case of priority queuing, packets belonging to one priority class of traffic are sent before all lower priority traffic to ensure timely delivery of high priority packets in case of congestion.

Congestion is a normal phenomena in bandwidth management. Congestion management features controls congestion by determining the order in which packets are sent as per their priority.

When no congestion exists, packets are sent as they arrive. But when the network is congested, the packets are put in priority queues before they are serviced.

Tech Knowledge Publications

Review Questions

Q. 1 Write a C function insert and delete to implement queue using array. **(6 Marks)**

Q. 2 When a queue is implemented using array, if rear becomes maximum size of queue, we cannot insert element even when there is space available. Suggest a solution to this problem. Justify your answer. **(6 Marks)**

Q. 3 What is the disadvantage of linear queue ? Suggest suitable method to overcome. **(6 Marks)**

Q. 4 Write a short note on circular queue. **(6 Marks)**

❑❑❑

Techknowledge Publication

Model Question Paper (End Sem.)

Fundamental of Data structures (210242)

Semester III - Computer Engineering (Savitribai Phule Pune University)

Time : $2\frac{1}{2}$ Hours **Maximum Marks : 70**

Strictly as per the New Credit System Syllabus (2019 Course)
Savitribai Phule Pune University w.e.f. academic year 2020-2021

Technical Knowledge Publication

Instructions to the candidates :

1) *Answer Q.1 or Q.2, Q.3 or Q.4, Q.5 or Q.6, Q.7 or Q.8,*
2) *Neat diagrams must be drawn wherever necessary.*
3) *Figure to the right indicate full marks.*
4) *Make suitable assumptions, if necessary.*

Q. 1 **(a)** Explain quick sort and sort the given list using quick sort : **(6 Marks)**

15, 8, 7, 17, 6, 3, 51, 7.

(b) Explain fibonacci search with an example. And, also give its function in C/C++. **(7 Marks)**

(c) Explain insertion sort and give its timing complexity. **(5 Marks)**

OR

Q. 2 **(a)** Explain merge sort and sort the given list using merge sort :

55, 85, 13, 9, 21, 17, 56, 3, – 2 **(6 Marks)**

(b) Explain various sorting algorithms with respect to their timing complexity. **(4 Marks)**

(c) Give C/C++ code for binary search and show its functioning for searching 9 in the given list of data :

11, 3, 5, 15, 6, 9, 1, 0, – 5, 37. **(7 Marks)**

Q. 3 **(a)** Write an algorithm to sort elements of singly linked list. **(6 Marks)**

(b) Write C/C++ code to reverse the singly linked list. **(6 Marks)**

(c) Represent the following polynomial using GLL.

$((x^{12} + 2x^9)\, y^4 + 4x^9\, y^2)z^3 + ((x^5\, 6x^3)y^5 + 3y)z$ **(5 Marks)**

OR

Q. 4 **(a)** Write C/C++ code to delete alternate elements from a linked list. **(6 Marks)**

(b) Write C/C++ code to delete the last element from a doubly linked list. **(5 Marks)**

(c) Write C/C++ code to add two polynomials represented using linked list. **(6 Marks)**

Q. 5 **(a)** Explain stack as an ADT using an array. **(5 Marks)**

(b) Convert following infix expression to postfix expression.

((A + B) – C* (D ↑ E) + F)

Use stack and show step by step conversion. **(6 Marks)**

(c) Write a C/C++ program to convert an infix expression into prefix forms. **(7 Marks)**

OR

Q. 6 **(a)** Explain, how multiple stacks can be stored in an array. **(6 Marks)**

(b) Explain Tower of Hanoi problem and give its C/C++ code. **(5 Marks)**

(c) Convert following infix expression to prefix forms :

((A + B) – C* (D ↑ E) + F). **(6 Marks)**

Q. 7 **(a)** Write a pseudo C/C++ code to implement a circular queue using arrays. **(6 Marks)**

(b) Explain linear queue and circular queue with suitable example. Give advantages of circular queue over linear queue. **(6 Marks)**

(c) Define the following terms with examples.

(i) Linear queue

(ii) Dequeue

(iii) Priority queue **(5 Marks)**

OR

Q. 8 **(a)** Explain linear queue and circular queue with suitable example. Give advantages of circular queue over linear queue. **(6 Marks)**

(b) Implement priorirty queue using arrays. **(11 Marks)**

Techknowledge Publication

❑❑❑

Note

Technical Knowledge Publication

Note

Technical Knowledge Publication