Contents

Sorting

Bubble ort, Quick Sort, Selection Sort, Tree Sorting: Binary Tree Sort, Heap sort ,Insertion Sorts: Simple insertion, Shell Sort ,Address   calculation Sort, Merge and Radix Sort

Graphs and Trees

Binary trees, operations on binary trees, Applications of binary trees,Binary trees,Binary tree representation,Node representation of binary tree,Internal and external nodes,Implicit array representation of binary trees,choosing a binary tree representation, Binary tree traversal in C,Threaded binary trees.

Graphs:

Definitions , Application of graphs, C representation of graphs, Traversal methods for graphs, depth first traversal, Breadth first traversal.

Reference Books:-

1.Data structures and Algorithm analysis in C, Mark Allen Weiss. 2nd edition, Pearson education Asia, 1997.

2. Data Structures – A pseudocode approach with C, Richard F Giberg and Behrouz A Forouzan, 3rd reprint Thomson course technology, 2005.

3. Data Structures using C and C++ by Yedidyah Langsam and Moshe J. Augenstein and Aaron M Tenenbaum, 2nd edition Pearson Education Asia 2002

Analysis of Algorithm

1.0 INTRODUCTION

A common person's belief is that a computer can do anything. This is far from truth. In reality, computer can perform only certain predefined instructions. The formal representation of this model as a sequence of instructions is called an algorithm, and coded algorithm, in a specific computer language is called a program. Analysis of algorithms has been an area of research in computer science; evolution of very high-speed computers has not diluted the need for the design of time-efficient algorithms.
Complexity theory in computer science is a part of theory of computation dealing with the resources required during computation to solve a given problem. The most common resources are *time* (how many steps (time) does it take to solve a problem) and *space* (how much memory does it take to solve a problem). It may be noted that complexity theory differs from computability theory, which deals with whether a problem can be solved or not through algorithms, regardless of the resources required.
Analysis of Algorithms is a field of computer science whose overall goal is understand the complexity of algorithms. While an extremely large amount of research work is devoted to the worst-case evaluations, the focus in these pages is methods for average-case. One can easily grasp that the focus has shifted from computer to computer programming and then to creation of an algorithm. This is & algorithm design, heart of problem solving.

1.2 MATHEMATICAL BACKGROUND

To analyze an algorithm is to determine the amount of resources (such as time and storage) that are utilized by to execute. Most algorithms are designed to work with inputs of arbitrary length.
Algorithm analysis is an important part of a broader computational complexity -theory, which provides theoretical estimates for the resources needed by any algorithm, which solves a given computational problem. These estimates provide an insight into reasonable directions of search for efficient algorithms.

Definition of Algorithm -

Algorithm should have the following five characteristic features:
1. Input
2. Output
3. Definiteness
4. Effectiveness
5. Termination.

Therefore, an algorithm can be defined as a sequence of definite and effective instructions, which terminates with the production of correct output from the given input.

Complexity classes

All decision problems fall into sets of comparable complexity, called complexity classes.
The complexity class P is the set of decision problems that can be solved by a deterministic machine in polynomial

time. This class corresponds to set of problems, which can be effectively solved, in the worst cases. We will consider algorithms belonging to this class for analysis of time complexity. Not all algorithms in these classes make practical sense as many of them have higher complexity. These are discussed later.
The complexity class NP is a set of decision problems that can be solved by a non- deterministic machine in polynomial time. This class contains many problems like Boolean satisfiability problem, Harniltonian path problem and the Vertex cover problem.

What is Complexity?

Complexity refers to the rate at which the required storage or consumed time grows as a function of the problem size. The absolute growth -depends on the machine used to execute the program, the compiler used to construct the program, and many other factors. We would like to have a way of describing the inherent complexity of a program (or piece of a program), independent of machine/compiler considerations. This means that we must not try to describe the absolute time or storage needed. We must instead concentrate on a "proportionality" approach, expressing the complexity in terms of its relationship to some known function. This type of analysis is known as **asymptotic analysis.** It may be noted that we are dealing with complexity of an algorithm not that of a problem. For example, the simple problem could have high order of time complexity and vice-versa.

Asymptotic Analysis

Asymptotic analysis is based on the idea that as the problem size grows, the. Complexity can be described as a simple proportionality to some known function. This idea is incorporated in the "Big 0", "Omega" and "Theta" notation for asymptotic performance.
The notations like "Little Oh" are similar in spirit to "Big Oh"; but are rarely used in computer science for asymptotic analysis.

Tradeoff between space **and time** complexity

We may sometimes seek **a** tradeoff between space and time complexity. For example, we may have to choose a data structure that requires a lot of storage in order to reduce the computation time. Therefore, the programmer must make a judicious choice from an informed point of view. The programmer must have some verifiable basis based on which a data structure or algorithm can be selected Complexity analysis provides such a basis.
We will learn about various techniques to bind the complexity function. In fact, our aim is not to count the exact number of steps of a program or the exact amount of time required for executing an algorithm. In theoretical analysis of algorithms, it is common to estimate their complexity in asymptotic sense, i.e., to estimate the complexity function for reasonably large length of input 'n'. Big 0 notation, omega notation 2 and theta notation 0 are used for this purpose. In order to measure the performance of **an** algorithm underlying the computer program, our approach would be based on a concept called asymptotic measure of complexity of algorithm. There are notations like big 0, 0, for asymptotic measure of growth functions of algorithms. The most common being big-O notation. The asymptotic analysis of algorithms is often used because time taken to execute an algorithm varies with the input 'n' and other factors which may differ from computer to computer and from run to run. The essences of these asymptotic notations are to bind the growth function of, time complexity with a function for sufficiently large input.

The 0-Notation (Tight Bound)

This notation bounds a function to within constant factors. We *say J(n) = €l(g(n))* if there exist positive constants *n0, c1* and *c2* such that to the right of n0 the value off(n) always lies between *c1g(n)* and *c2g(n),* both inclusive. The *Figure 1.1* gives an idea about function f(n) and *g(n)* where f(n) *&(g(n)).* We will say that the function *g(n)* is asymptotically tight bound *for f(n).*

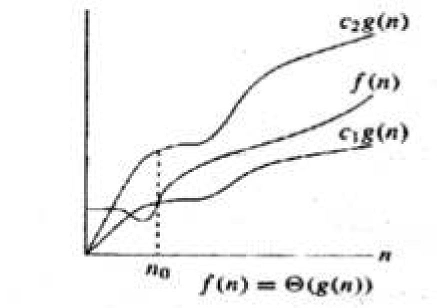


Figure 1.1 : Plot of $f(n) = \Theta(g(n))$

For example, let us show that the function f(n) = – *n2* – *4n* = *0(n2).*
Now, we have to find three positive constants, *c1, c 2* and *n0* such that
c1n2 – *n2* – *4n c2 n2 for all n n0*
z *c1_c2*
3 *n*
By choosing- *n0* = *1* and c2 *1/3* the right hand inequality holds true.
Similarly, by selecting *n0 13 c1 1/39,* the right hand inequality holds true. So, for *c1* = *1/39, c2* = *1/3* and *n0 13,* it follows that *1/3 n2* – *4n & (n2).*
Certainly, there are other choices for *c,,, c2* and *no.* Now we may show that the function f(n) *6n3 (9(n2).*
To prove this, let us assume that *c3* and *n0* exist such that
6n3 cn2 for n n(, But this fails for sufficiently large *n.* Therefore *6n3* **&** *(n2).*

The big **0 notation (Upper Bound)**

This notation gives an upper bound for a function to within a constant factor. *Figure 1.2* shows the plot of *f(n)* = *O(g(n))* based on big **0** notation. We write j(n) = *O(g(n))* if there are positive constants *n0* and *c* such that to the right of n0, the value off(n) always lies on or below *cg(n).*

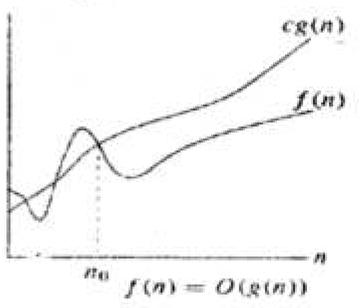


Figure 1.2: Plot of $f(n) = O(g(n))$

Mathematically for a given function *g(n),* we denote a set of functions by *O(g(n))* by the following notation:
O(g(n)) = *([('n).* There exists a positive constant *c* and *n0* such that *0 f(n) cg(n)* for all *n no}*
Clearly, we use 0-notation to define the upper bound on a function by using a constant factor *c.*
We can see from the earlier definition of **&** that (9is a tighter notation than big-O notation.
f(n) an + *c is 0(n)* is also *0(n2),* but *0 (n)* is asymptotically tight whereas *0(n2)* is notation.

Cg (n)

no = O((n)

Whereas in terms of 8 notation; the above function fi'nl is 8 ('nl. As *big-O* notation is upper bound of function, it is often used to describe the worst case running time of algorithms.

The Ω -Notation (Lower Bound)

This notation gives a lower bound for a function to within a constant factor. We write *1(n) = 2(g(n)),* if there are positive constants *n0* and *c* such that to the right of n0, the value *of' J(n)* always lies on or above *cg(n). Figure 1.3* depicts the plot of *f(n) = 2(g(n)).*

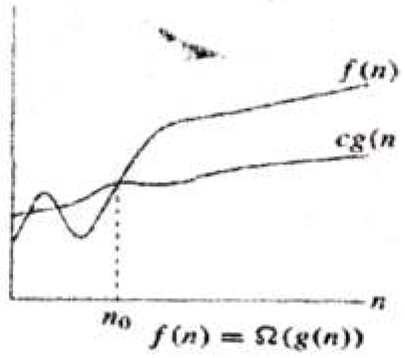


Figure 1.3: Plot of $f(n) = \Omega(g(n))$

Mathematically for a given function *g(n),* we may define *2(g(n))* as the set of functions.
Q(g(n)) = (f(n) : there exists a constant *c* and *n0 0* such that *0 cg(n) f(n)* for all *n n0}*.
Since Q notation describes lower bound, it is used to bound the best case running time of an algorithm.

Asymptotic notation

Let us define a few functions in terms of above asymptotic notation.

Example :f(n) = *3n3 + 2n2 + 4n + 3*
3n3 +2n2+0(n), as4n +3isofQ(n)
=3n3+0(n2), as2n2+0(n) is0(n2)
U (p3)

Example :f(n) *n2 + 3n +* 4 *is 0(n2),* since *n2 + 3n +* 4 < *2n2* for all *n> 10.* By definition of big-0, *3n* + 4 is also *0(n2),* too, but as a convention, we use the tighter bound to the function, i.e., *0(n).*
Here are some rules about big-0 notation:
1. *f(n) = 0f('n))* for any function f In other words, every function is bounded by itself.
2. *aknk* + ***aklnk*** + *ajn* + a0 ***Q(k)*** for allk0 and for all a a1 ak ***ER.*** In other words, every polynomial of degree *k* can be bounded by the function ***nk.***
Smaller order terms can be ignored in big-0 notation.
3. Basis of Logarithm can be ignored in big-U notation i.e. *loga 11* = ***O(logb ii)*** for any bases a, *b.* We generally write *0(log n)* to denote a *logarithm* n to any base.
4. Any logarithmic function can be bounded by a polynomial i.e. *logb n* = O('n')

for any *b* (base of logarithm) and any positive exponent *c> 0.*
5. Any polynomial function can be bounded by an exponential function i.e.
=0 'b").
6. Any exponential function can be bound by the factorial function. For example,
0(n0 for any base a.

13 PROCESS OF ANALYSIS

The objective analysis of an algorithm is to find its efficiency. Efficiency is dependent on the resources that are used by the algorithm. For example,
• CPU utilization (Time complexity)
• Memory utilization (Space complexity)
• Disk usage 0/0)
• Network usage (bandwidth).
There are two important attributes to analyse an algorithm. They are:
Performance: How much time/memory/disk/network bandwidth is actually used when a program is run. This depends on the algorithm, machine, compiler, etc.
Complexity: How do the resource requirements of a program or algorithm scale (the growth of resource requirements as a function of input). In other words, what happens to the performance of an algorithm, as the size of the problem being solved gets larger
•and larger? For example, the time and memory requirements of an algorithm which computes the sum of 1000 numbers is larger than the algorithm which computes the Analysis of sum of 2 numbers.

Algorithms *Time Complexity. The* maximum time required by a *Turing machine* to execute on any input of length a.
Space Complexity. The amount of storage space required by an algorithm v-aries with the size of the problem being solved. The space complexity is normally expressed as an order of magnitude of the size of the problem, e.g., 0(n2) means that if the size of the problem (n) doubles then the working storage (memory) requirement will become four times.
Determination of Time Complexity
The RAM Model
The random access model (RAM) of computation was devised by John von Neumann to study algorithms. Algorithms are studied in computer science because they are independent of machine and language.
We will do all our design and analysis of algorithms based on RAM model of computation:
• Each "simple" -operation (+, .. if. call) takes exactly 1 step.
• Loops and subroutine calls *are not* simple operations, but depend upon the size of the data and the contents of a subroutine.-
• Each memory access takes exactly I step.
The complexity of algorithms using big-0 notation can be defined in the following way for a problem of size n:
• Constant-time method is "order 1": *0(1).* The time required is constant independent of the input size.
• Linear-time method is "order n": *0(n).* The time required is proportional to the input size. If the input size doubles, then, the time to run the algorithm also doubles.
• Quadratic-time method is "order N squared": *0(n2).* The time required is proportional to the square of the input size. If the input size doubles, then, the

time required will increase by four times.
The process of analysis of algorithm (program) involves analyzing each step of the algorithm. It depends on the kinds of statements used in the program.
Consider the following example:
Example 1: Simple sequence of statements
Statement 1;
Statement 2;
Statement k;
The total time can be found out by adding the times for all statements:
Total time (statement 1) ± time(statement 2) + ... + time(statement *k)*

It may be noted that time required by each statement will greatly vary depending on whether each statement is simple (involves only basic operations) or otherwise. Assuming that each of the above statements involve only basic operation, the time for each simple statement is constant and the total time is also constant: *0(1).*
Example 2: fthen-else statements
In this example, assume the statements are simple unless noted otherwise. Th
+
if-then-else statements prt pr<
if (cond) { art
sequence of statements 1
}
else {
sequence of statements 2
}
In this, if-else statement, either sequence 1 will execute, or sequence 2 will execute depending on the boolean condition. The worst-case time in this case is the slower of the two possibilities. For example, if sequence I is *0(N2)* and sequence 2 is *0(1),* then the worst-case time for the whole if-then-else statement would be *0(N2).* TI
Example 3:for loop In
c:
for(i—0i<n;i++){ ol
sequence of statements
} ti
Here, the loop executes n times. So, the sequence of statements also executes *n* times. B
Since we assume the time complexity of the statements are *0(1),* the total time for the it loop is *n * 0(1),* which is *0(n).* Here, the number of statements does not matter as it c will increase the running time by a constant factor and the overall complexity will be 2 same *0(n).*
Ii
Example 4:nested for loop
t
for (I *0;* i < *n; j ++)* { j
for(j=0;j<m;j++){
sequence of statements
) 1
)
Here, we observe that, the outer loop executes *n* times. Every time the outer loop

executes, the inner loop executes *m* times. As a result of this, statements in the inner loop execute a total of *n * m* times. Thus, the time complexity is *0(n * m).* If we modify the conditional variables, where the condition of the inner loop isj < *n* instead off < *m* (i.e., the inner loop also executes *n* times), then the total complexity for the nested loop is *0(n2).*
Example 4: Now, consider a function that calculates partial sum of an integer *n.*

```
int psum(int n)
{
int i, partial_sum;


partial_sum = 0; 1* Line 1 */ Analysis of
for (i = 1; i <= n; i++) { 1* Line 2 */ Algorithms
partial_sum = partial_sum + i*i; 1* Line 3 *1
}
return partial_sum; / Line 4 */
}
```

This function returns the sum from *I* ito *n* of *I* squared, i.e. psum = + *22+ 32* + + *n2.* As we have to determine the running time for each statement in this program, we .have to count the number of statements that are executed in this procedure. The code at line 1 and line 4 are one statement each. The **for ioop** on line 2 are actually *2n+2* statements:

• I = 1; statement : simple assignment, hence one statement.
• i < n; statement is executed once for each value of *i* from 1 to *n+1* (till the condition becomes false). The statement is executed *n* + *1* times.
• i++ is executed once for each execution of the body of the loop. This is executed *n* times.

Thus, the sum is *1+ (n+i)* + *n+1 2n+ 3* times.
In terms of big-O notation defined above, this function is *0(n),* because if we choose *c3,* then we see that *en* > *2n+3.* As we have already noted earlier, big-O notation only provides a upper bound to the function, it is also *0(nlog(n))* and *0(n2),* since > *nlog(n)* > *2n+3.* However, we will choose the smallest function that describes the order of the function and it is *0(n).*
By looking at the definition of Omega notation and Theta notation, it is also clear that it is of *0(n),* and therefore *2(n)* too. Because if we choose *c=i,* then we see that *en <2n+3,* hence *Q(n)* . Since *2n+3* = *0(n),* and *2n+3* = *2('n),* it implies that *2n+3* = *0(n)* , too.
It is again reiterated here that smaller order tenns and constants may be ignored while describing asymptotic notation. For example, iff(n) = *4n+6* instead off(n) 2n +3 in terms of big-O, 2 and *0,* this does not change the order of the function. The function *f(n) 4n+6* = *0(n)* (by choosing *c* appropriately as *5); 4n+6* = 2(n) (by choosing *c* = 1), and therefore *4n+6* = *0(n).* The essence of this analysis is that in these asymptotic notation, we can count a statement as one, and should not worry about their relative execution time which may depend on several hardware and other implementation factors, as long as it is of the order of 1, i.e. *0(1).*
Exact analysis of insertion sort.

Let us consider the following pseudocode to analyse the exact runtime complexity of insertion sort.

Line	Pseudocode	Cost factor	No. of iterations
1	for j=2 to length [A] do	*c1*	$(n-1) + 1$
2	{ key = A[j]	*c2*	$(n-1)$
3	i = j – 1	*c3*	$(n-1)$
4	while (i > 0) and (A[i] > key) do	*c4*	$\sum_{j=2}^{n} T_j$
5	{ *A[i+1] = A[I]*	*c4*	$\sum_{j=2}^{n} T_j - 1$
6	*i = I–1 }*	*c5*	$\sum_{j=2}^{n} T_j - 1$
7	A[*I+1*] = key }	*c6*	*n–1*
	}		

T1 is the time taken to execute the statement during *1th* iteration.
The statement at line 4 will execute *T* number of times.
The statements at lines *5* and 6 will execute 7) *—1* number of times (one step less) each
Line 7 will excute *(n—1)* times

So, total time is the sum of time taken for each line multiplied by their cost factor.

$$T(n) = c1n + c2(n-1) + c3(n-1) + c4\sum_{j=2}^{n} T_j + c5\sum_{j=2}^{n} T_j - 1 + c6\sum_{j=2}^{n} T_j - 1 + c7(n-1)$$

Three cases can emerge depending on the initial configuration of the input list. First, the case is where the list was afready sorted, second case is the case wherein the list is sorted in reverse order and third case is the case where in the list is in random order (unsorted). The best case scenario will emerge when the list fs already sorted.
Worst Case: Worst case running time is an upper bound for running time with any input. It guarantees that, irrespective of the type of input, the algorithm will not take any longer than the worst case time.
Best Case : -It guarantees that under any cirurnstances the running time of algorithms will at least take this much time.
Average case This gives the average running time of algorithm. The running time for any given size of input will be the average number of operations over all problem instances for a given size.
Best Case: If the list is already sorted then *A[i]* <= key at line 4. So, rest of the lines in the inner loop will not execute. Then,
T (n) = *cm* + *c2(n —1)* + *c3(n —1)* + *c4 (n —I)* = *0 (n),* which indicates that the time complexity is linear.
Worst Case: This case arises when the list is sorted in reverse order. So, the boolean condition at line 4 will be true

for execution of line 1.

So, step line 4 is executed $\sum_{j=2}^{n} j = n(n+1)/2 - 1$ times

$$T(n) = c1n + c2(n-1) + c3(n-1) + c4(n(n+1)/2 - 1) + c5(n(n-1)/2) + c6(n(n-1)/2) + c7(n-1)$$

$$= O(n^2).$$

Average case: In most of the cases, the list will be in some random order. That is, it neither sorted in ascending or descending order and the time complexity will lie some where between the best and the worst case.

$T(n)_{best} < T''n)_{Avg.} < T(n)_{worst}$

Figure 1.4 depicts the best, average and worst case run time complexities of algorithms.

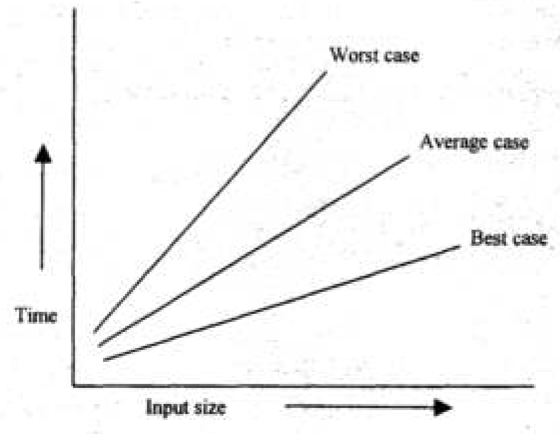


Figure 1.4 : Best, Average and Worst case scenarios

1.4 CALCULATION OF STORAGE and COMPLEXITY

As memory is becoming more and more cheaper, the prominence of runtime complexity is increasing. However, it is very much important to analyse the amount of memory used by a program. If the running time of algorithms is not good then it will take longer to execute. But, if it takes more memory (the space complexity is more) beyond the capacity of the machine then the program will not execute at all. It is therefore more critical than run time complexity. But, the matter of respite is that memory is reutilized during the course of program execution.

We will analyse this for recursive and iterative programs.

For an iterative program, it is usually just a matter of looking at the variable declarations and storage allocation calls, e.g., number of variables, length of an array etc.

The analysis of recursive program with respect to space complexity is more complicated as the

space used at any time is the total space used by all recursive calls active at that time.
Each recursive call takes a constant amount of space and some space for local variables and function arguments, and also some space is allocated for remembering where each call should return to. General recursive calls use linear space. That is, for *n* recursive calls, the space complexity is *0(n).*
Consider the following example: *Binary Recursion* (A *binary-recursive* routine (potentially) calls itself twice).
1. If *n* equals 0 or 1, then return 1
2. Recursively calculatef(n— 1)
3. Recursively calculatef(n—2)
4. Return the sum of the results from steps 2 and 3.
Time Complexity: O(exp *n)*
Space Complexity: O(exp *n)*
Example: Find the greatest common divisor (GCD) of two integers, *m* and *n.* The algorithm for GCD may be defined as follows:
While *m* is greater than zero:
If *n* is greater than *m,* swap *m* and *n.*
Subtract *n* from *m.*
nistheGCD
<u>Code in C</u>

```
int gcd(int m, int n)
/* The precondition are : m>0 and n>0. Let g = gcd(m,n). *1
{while(m >0)
{
if(n >m)
{intt=m;m =n;n =t;}/* swap mandn*/
1* m >= n> 0 */
m -=
}
return

}
```

Analysis of Algorithms
The space-complexity of the above algorithm is a constant. It just requires space for three integers *mn* and t. so, the space complexity is *0(1).*
The time complexity depends on the loop and on the condition whether *m>n* or not.
The real issue is, how much iteration take place? The answer depends on m and *n.*
Best case: If m = *n,* then there is just one iteration. *0(1)*
Worst case: If *n* = 1, then there are m iterations; this is the worst-case
(also equivalently, if *m* = 1 there are n iterations) *0(n).*
The *space complexity* of a computer program is the amount of memory required for its proper execution. The important concept behind space required is that unlike time, space can be reused during the execution of the program. As discussed, there is often a trade-off between the time and space required to run a program.
In formal definition, the space complexity is defined as follows:

ails *Space complexity* of a Turing Machine: The (worst case) maximum length of the tape required to process an input string of length *n.*
In complexity theory, the class *PSPACE* is the set of decision problems that can be for solved by a Turing machine using a polynomial amount of memory, and unlimited time.

1.5 CALCULATION OF TIME COMPLEXITY

Example 1: Consider the following of code:
x = 4y + *3*
z=z+ 1
p—i
As we have seen, x, y, *z* and p are all scalar variables and the running time is constant irrespective of the value of x,y,E and p. Here, we emphasize that each line of code may take different time, to execute, but the bottom line is that they will take constant amount of time. Thus, we will describe run time of each line of code as *0(1).*
Example 2: Binary search
Binary search in a sorted list is carried out by dividing the list into two parts based on the comparison of the key. As the search interval halves each time, the iteration takes place in the search. The search interval will look like following after each iteration
N, N/2, N/4, N/8 8, 4, 2, 1

The number of iterations (number of elements in the series) is not so evident from the above series. But, if we take logs of each element of the series, then
log2 N, log2 N —1, log2 N—2, log2 N—3 . 3, 2, 1, 0
As the sequence decrements by 1 each time the total elements in the above series are *log2 N + 1.*
So, the number of iterations is *log2 N + 1* which is of the order of
0(log2N).
Example 3: Traveling Salesman problem
Given: n connected cities and distances between them
Find: tour of minimum length that visits every city.
Solutions: How many tours are possible?
Because *n!>* 2(n-1)
So *n! = Q (2)* (lower bound)
As of now, there is no algorithm that finds a tour of minimum length as well as covers all the cities in polynomial time. However, there are numerous very good heuristic algorithms.
The complexity Ladder:
• *T(n) 0(1).* This is called constant growth. *T(n)* does not grow at all as a function of *n,* it is a constant. For example, array access has this characteristic. *A[i]* takes the same time independent of the size of the array *A.*
• *T(n) = 0(log2 (n)).* This is called logarithmic growth. *T(n)* grows proportional to the base 2 logarithm of *n.* Actually, the base of logarithm does not matter. For example, binary search has this characteristic.
• *T(n) = 0(n).* This is called linear growth. *T(n)* grows linearly with *n.* For example, looping over all the elements in a one-dimensional array of *n* elements would be of the order of *0(n).*
• *T(n) = 0(n log (n).* This is called **niogn** growth. *T(n)* grows proportional to *n* times the base 2

logarithm of n. Time complexity of Merge Sort has this
characteristic. In fact, no sorting algorithm that uses comparison between elements can be faster than *n* log *n.*
• *T(n) = 0(nk).* This is called polynomial growth. *T(n)* grows proportional to the *k-th* power of *n.* We rarely consider algorithms that run in time *0(k)* where *k* is bigger than 2 . because such algorithms are very slow and not practical. For
example, selection sort is an *0(n2)* algorithm.
• *T(n) = 0(2)* This is called exponential growth. *T(n)* grows exponentially. Exponential growth is the most-danger growth pattern in computer science.
Algorithms that grow this way are basically useless for anything except for very small input size.
Table 1.1 compares various algorithms in terms of their complexities.
Table 1.2 compares the typical running time of algorithms of different orders.
5:

The growth patterns aóv hav hee listed **in order of** inceasj That is. 41) **< O1ogØ <*O(n*** *kigç'n))* ***<O(n2) < O(n***

0(nt)

Table 1.1 elr complexities

Table l3airtson of typical runnmg tiaoalge*hms otdereAtos*rs

Notation	Name	**Example**
0(1)	Constant	**Constant growth. Does not grow as a** function **of n. For example, accessing array for one** element ALE]
O(logn)	Logarithmic	**Einaxy** search
0(n)	**Linear**	**Looping over n elements, of an array ofsizen(norma1ly)**
O(n *log* n)	$orneims a1 1ed "1maritnnic"	**Merge sort**
0(n3)	**Qüa4iis.tk**	**Worst** time vase **for** insertion sort matrix mu1tip1a
*	Polynomi1,	

Array size	Logarithimic: Log N	Linear N		**Quadratic N^2**		onenti* **2**

8 —l2$ *26* 1000	3 --7: 8 10 17	8 128 256 1000 100,000		*64* 16,384 65$36 1 million 10 billiøu		-: *256* *3.4*10 38* *1.15*10^ 77*

UNIT 2 ARRAYS

2.0 INTRODUCTION

This unit introduces a data structure called Arrays. The simplest form of array is a one-dimensional array that may be defined as a finite ordered set of homogeneous elements, which is stored in contiguous memory locations. For example, an array may contain all integers or all characters or any other data type, but may not contain a mix of data types.
The general form for declaring a single dimensional array is:
data type array name (expression);
where data type represents data type of the array. That *is,* integer, char, float etc. array name is the name of array and expression which indicates the number of elements in the array.
For example, consider the following C declaration:
int a[l0OJ;
It declares an array of 100 integers.
The amount of storage required to hold an array is directly related to its type and size. For a single dimension array, the total size in bytes required for the array is computed as shown below.
Memory required (in bytes) = size of (data type) X length of array
The first array index value is referred to as its lower bound and in C it is always 0 and the maximum index value is called its upper bound. The number of elements in the array, called its range is given by upper bound-lower bound.
We store values in the arrays during program execution. Let us now see the process of

initializing an array while declaring it.

```
int a[4] = {34,60,93,2};
int b[ ] = (2,3,4,5);
float c[ ] = (-4,6,81,— 60);
```

We conclude the following facts from these examples:

(i)If the array is initialized at the time of declaration. Then the dimension of the
Structures array is **optional.**
(ii) Till the array **elements** are not given any specific values, they contain garbage values.

2.2 ARRAYS AND POINTERS

C compiler does not check the bounds of arrays. It is your job to do the necessary work for checking boundaries wherever needed.
One of the most common arrays is a string, which is simply an array of characters terminated by a null character. The value of the null character is zero. A string constant is a one-dimensional array of characters terminated by a null character (O).
For example. Consider the following:

char message [] **{'e', 'x', 'a',** 'm', **'p', 'l','e',"O'}:**
Also, consider the following string which is stored in an array:
sentence\n"
Figure 2.1 shows the way a character array is stored in memory. Each character in the array occupies one byte of memory and the last character is always "0'. Note that '\O' and '0' are not the same. The elements of the character array are stored in contiguous memory locations.

S	E	N	T	E	N	C	E	\0

Figurec **2.1: String** in memory

C concedes a fact that the user would use strings very often and hence provides a short cut for

initialization of strings.
For example, the string used above can also be initialized as
char name [I = ‘sentence\n”;
Note that, in this declaration ‘\O’ is not necessary. C inserts the null character automatically.
Multidimensional arrays are defined in the same manner as one-dimensional arrays, except that a separate pair of square brackets is required for each subscript. Thus a two-dimensional array will require two pairs of square brackets, a three-dimensional array will require three pairs of square brackets and so on.

The format of declaration of a multidimensional array in C is given below:

Data type array name [expr I) [expr 2].... [expr n];
where data type is the type of array such as int, char etc., array name is the nane of array and expr 1, expr 2. ....expr n are positive valued integer expressions.
The schematic of a two-dimensional array of size 3 x 5 is shown as

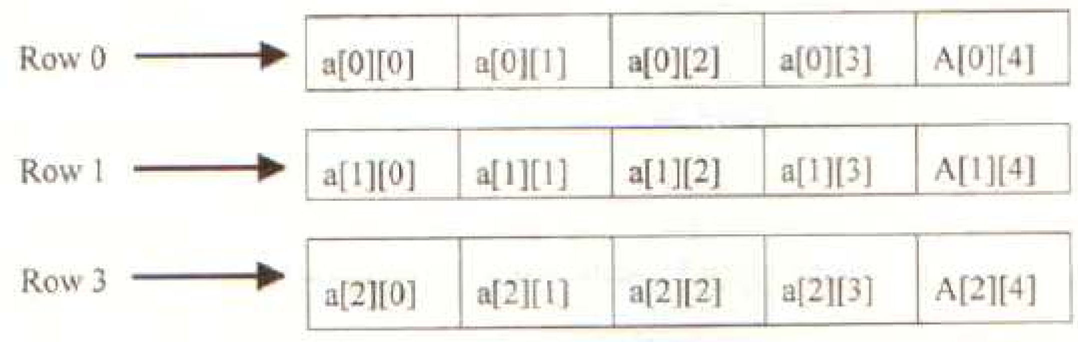


Figure 2.2: Schematic of a Two-Dimensional Array

In the case of a two-dimensional array, the following formula yields the number of bytes of memory needed to hold it:
bytes = size of 1q index x size of 2nd index x size of (base type)
The pointers and arrays are closely related. As you know, an array name without an index is a pointer to the first element in the array.

Consider the following array:

char p[10];

p and &p[O1 are identical because the address of the first element of an array is the same as the address of the array. So, an array name without an index generates a pointer. Conversely a pointer can be indexed as if it were declared to be an array.

For example. consider the following program fragment:

int xa[l0]; x a;
x[5] = 100; (x+5) 100;

Both assignment statements place the value 100 in the sixth element of a. Furthermore the (0.4) element of a two-dimensional array may be referenced in the following two ways: either by array indexing a[0][4]. or by the pointer
*((int)a+4).
In general, for any two-dimensional array a[jJ[k] is equivalent to:
((base type)a + (j * rowlength)k)

2.3 SPARSE MATRICES

Matrices with good number of zero entries are called sparse matrices.

Consider the following matrices of Figure 2.3.

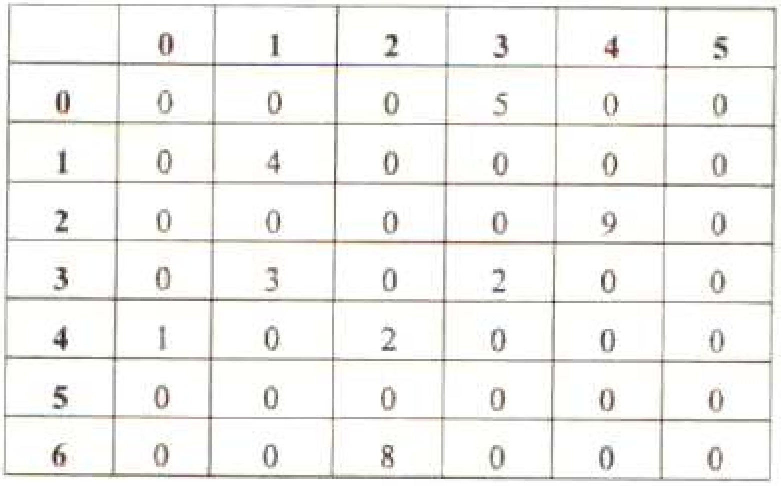
$$\begin{pmatrix} 4 & & & & \\ 3 & -5 & & & \\ 1 & 0 & 6 & & \\ -7 & 8 & -1 & 3 & \\ 5 & -2 & 0 & 2 & -8 \end{pmatrix} \qquad \begin{pmatrix} 5 & -3 & & & & & \\ 1 & 4 & 3 & & & & \\ & 9 & -3 & 6 & & & \\ & & 2 & 4 & -7 & & \\ & & & 3 & -1 & 0 & \\ & & & & 6 & -5 & 8 \\ & & & & & 3 & -1 \end{pmatrix}$$

(a) (b)

Figure 2.3: (a) Triangular Matrix (b) Tridiagonal Matrix

A triangular matrix is a square matrix in which all the elements either above or below the main diagonal are zero. Triangular matrices are sparse matrices. A tridiagonal matrix is a square matrix in which all the elements except for the main diagonal, diagonals on the immediate upper and lower side are zeros. Tri-diagonal matrices are also sparse matrices.
Let us consider a sparse matrix from storage point of view. Suppose that the entire sparse matrix is stored. Then, a considerable amount of memory, which stores the matrix, consists of zeroes. This is n.othing but wastage of memory. In real life applications, such wastage may count to megabytes. So, an efficient method of storing sparse matrices has to be looked into.

Figure 2.4 shows a sparse matrix of order 7 x 6.



	0	1	2	3	4	5
0	0	0	0	5	0	0
1	0	4	0	0	0	0
2	0	0	0	0	9	0
3	0	3	0	2	0	0
4	1	0	2	0	0	0
5	0	0	0	0	0	0
6	0	0	8	0	0	0

Figure 2.4: Representation of a sparse matrix of order 7 × 6

A common way of representing non-zero elements of a sparse matrix is the 3-tuple form. The first row of sparse matrix always specifies the number of rows, number of columns and number of non-zero elements in the matrix. The number? Represents the total number of rows sparse matrix. Similarly, the number 6 represents the total number of columns in the matrix. The number 8 represents the total number of non-zero elements in the matrix. Each non zero element is stored from the second row, with the I" and 2 elements of the row, indicating the row number and column number respectively in which the element is present in the original matrix. The 3flt element in this row stores the actual value of the non-zero element. For example, the 3- tuple representation of the matrix of Figure 2.4 is shown in Figure

7	7	9
0,	3,	5
1,	1,	4
2,	4,	9
3,	1,	3
3,	3,	2
4,	0,	1
4,	2,	2
6,	2,	8

Figure 2.5: 3-Tuple representation **of Figure 2.4**

The following program 1.1 accepts a matrix as input, which is sparse and prints the corresponding 3-tuple representations.

Program 1.1

```
/* The program accepts a matrix as Input and prints the 3-tuple representation
of lt*1
# include<stdioh>
void main ()
{
int a[5][5],rowscolumns,ij;
print f"enter the order of the matrix. The order should be less than 5 x
scan f("%d %d"&rows,&columns);
prfntf("Enter the elements of the matrix:n");
for(i=O;i<rows;i++)
for(j0j<columnsj++)
(scanf("°,4d",&a[il[j]);
)
printf("The 3-tuple representation of the matrix is:\n");
for(I=0;i<rows;i++)
for(j=0;j<columns;j++)
{
if (a[i][j] !=O)
{
printf("%d%d %d", (i+l),(j+1 ),a[i][j]);
)
}
)
```

Output:

```
enter the order of the matrix. The order should be less than 5 x 5:
Enter the elements of the matrix:
1 2 3
0 1 0
0 0 4
The 3-tuple representation of the matrix is:
1  1  1
1  2  2
1  3 3
2  2 1
3  3 4
```

The program initially prompted for the order of the input matrix with a warning that the order should not be greater than 5 x 5. After accepting the order, it prompts for the elements of the matrix. After accepting the matrix, it checks each element of the matrix for a non-zero. If the element is non-zero. Then it prints the row number and column number of that element along with its value.

2.4 POLYNOMIALS

Polynomials like 5x4 + 2x' + 71 + lOx — 8 can be represented using arrays. Arithmetic operations like addition and multiplication of polynomials are common and most often, we need to write a program to implement these operations.
The simplest way to represent a polynomial of degree 'n' is to store the coefficient of (n+1) terms of the polynomial in an array. To achieve this, each element of the array should consist of two 'values, namely, coefficient and exponent. While maintaining the polynomial, it is assumed that the exponent of each successive term is less than that of the previous term. Once we build an array to represent a polynomial, we can use such an array to perform common polynomial operations like addition and multiplication.
Program 1.2 accepts two polynomials as input and adds them.

Program 1.2

```
/* The program accepts to polynomials as input and prints the resultant polynomial due to the
addition of input polynomials*/
#include<stdio.h>
void main()
{
  int poly 1 [6][2)poly2[6J[2],term I ,term2,match,proceed.ij;
printI'Enter the number of terms in the first polynomial. They should be less than 6:\n");
scan t("%d",&term I);
printf("Enter the number of terms in the second polynomial. They should be
less than 6:\n");
scan t "%d"&term2);
printf("Enter the coefficient and exponent of each term of the first
polynomial:\n");
for(i=O;i<term I ;i++)
{scanf("%d,°/od",&poly I [iJ[OJ,&poly I 1i][ 1]);
}
printf'Enter the coefficient and exponent of each term of the second
polynomial :\n");
for(i —O ;i <term2 :i++)
{scanf("%d %d".&poly2[i][OI,&poly2[i][ II)
}
printf1"The resultant polynomial due to the addition of the input two
polynomials:\n");
for(i=O;i<terml ;i++)
{
maich=O;
for(j =OJ<term2j++)
{ if(match-=O)
if(poly I [i][ I jpoly2[j[ I])
{ print1"%d °od\n",(poly 1 [iJ[O]+po1y2[j[O]), poly I [i][ I));
match1
}
}
for(i=O;i<term I ;i+4)
{ proceedl;
tr(j=Oj<term2j++)
ifproceed=1)
itpoIy1[i}( I]!poIy2U1[IJ)
proceed= 1;
else
procecd0;
}
if(proceed==1)
printft"°/od %dn",poly 1 [i][Ojpo1y1 [i][ 1]);
}
for(iO;i<term2;i++)
{ proceedl;
for j=Oj<term I j++)
{ itproceed-=I)
itpoly2[ij[ I ]!polyl IjJ[I])
proceed= 1;
else
proceed—0;
if (proceed I)
printf"°/od °/od"poly2[i][OJ.poly2[i][ 1]),
}
```

Output:

```
Enter the number of terms in the first polynomial.They should be less than 6 : 5.
Enter the number of terms in the second polynomial.They should be less than 6 : 4.
Enter the coefficient and exponent of each term of the first polynomial:
12
24
36
29
```

```
Enter the coelticient and exponent of each term of the second polynomial:
52
69
36
57
The resultant polynomial due to the addition of the input two polynomials:
62
66
107
24
18
69
```

The program initially prompted for the number of terms of the two polynomials. Then, it prompted for the entry of the terms of the two polynomials one after another. Initially, it adds the coefficients of the corresponding terms of both the polynomials whose exponents are the same. Then, it prints the terms of the first polynomial that does not have corresponding terms in the second polynomial with the same exponent. Finally, it prints the terms of the second polynomial that does not have corresponding terms in the first polynomial.

2.5 REPRESENTATION OF ARRAYS

It is not uncommon to find a large number of programs, which process the elements of an array in sequence. But, does it mean that the elements of an array are also stored in sequence in memory. The answer depends on the operating system under which the program is running. However, the elements of an array are stored in sequence to the extent possible. if they are being stored in sequence, then how are they sequenced. Is it that the elements are stored row wise or column wise? Again. it depends on the operating system. The former is called row major order and the later is called column major order.

2.5.1 Row Major Representation

The first method of representing a two-dimensional array in memory is the row major representation. Under this representation, the first row of the array occupies the first set of the memory location reserved for the array, the second row occupies the next set,, and so forth. The schematic of row major representation of an Array is shown in Figure 2.6. Let us consider the following two-dimensional array:

$$\begin{array}{cccc} a & b & c & d \\ e & f & g & h \\ i & j & k & l \end{array}$$

To make its equivalent row major representation, we perform the following process:

Move the elements of the second row starting from the first element to the memory location adjacent to the last element of the first row. When this step is applied to all the rows except for the first row, you have a single row of elements. This is the Row major representation.

By application of above mentioned process, we get {a, b, c, d, e, f, g, h, i,j, k, I }

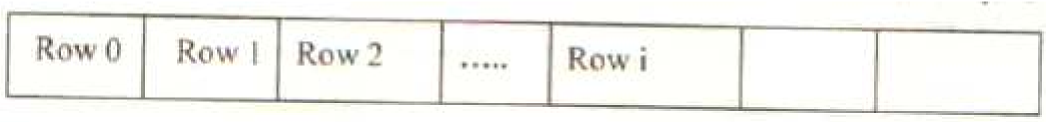


Figure 2.6: Schematic of a Row major representation of an Array

2.5.2 Column Major Representation

The second method of representing a two-dimensional array in memory is the column major representation. Under this representation, the first column of the array occupies the first set of the memory locations reserved for the array. The second column occupies the next set and sc forth. The schematic of a column major representation is shown in Figure 2.7.
Consider the following two-dimensional array:

$$\begin{array}{cccc} a & b & c & d \\ e & f & g & h \\ i & j & k & l \end{array}$$

To make its equivalent column major representation, we perform the following process:
Transpose the elements of the array. Then, the representation will be same as that of the row major representation.
By application of abovc mentioned process, we get {a, e, I, b, f,j, c, g, k, d, h, I)

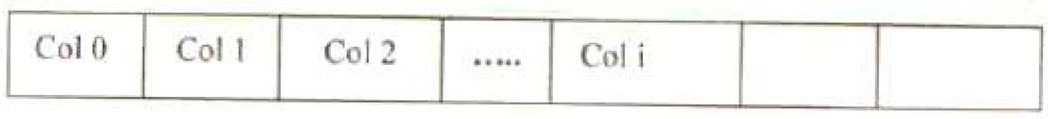


Figure 2.7: Schematic of a Column major representation of an Array

2.6 APPLICATIONS

Arrays are simple, but reliable to use in more situations than you can count. Arrays are used in those problems when the number of items to be solved is fixed. They are easy to traverse, search and sort. It is very easy to manipulate an array rather than other subsequent data structures. Arrays are used in those situations where in the size of array can be established before hand. Also, they are used in situations where the insertions and deletions are minimal or not present. Insertion and deletion operations wiN lead to wastage of memory or will increase the time complexity of the program due to the resuming of elements.

UNIT 4 STACKS

4.0 INTRODUCTION

One of the most useful concepts in computer science is stack. In this unit, we shall examine this simple data structure and see why it plays such a prominent role in the area of programming. There are certain situations when we can insert or remove an item only at the beginning or the end of the list.

A stack is a linear structure in which items may be inserted or removed only at one• end called the *top of the stack.* A stack may be seen in our daily life, for example, *Figure 4.1* depicts a stack of dishes. We can observe that any dish may

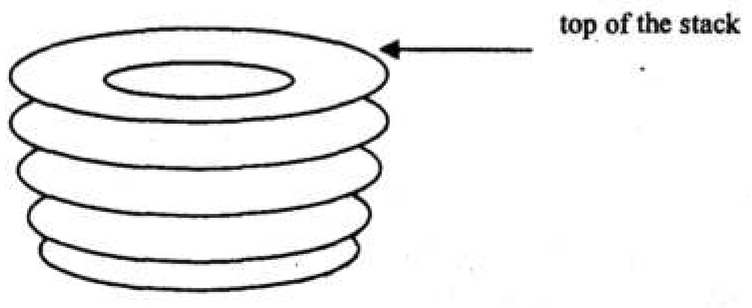


Figure 4.1: A stack of dishes

be added or removed only from the top of the stack. It concludes that the item added last will be the item removed first. Therefore, stacks are also called LIFO (Last In First Out) or FILO (First In Last Out) lists. We also call these lists as "piles" or "push-down list".

Generally, two operations are associated with the stacks named Push & Pop.

• *Push* is an operation used to insert an element at the top.

• *Pop* is an operation used to delete an element from the top

Example 4.1

Now we see the effects of push and pop operations on to an empty stack. *Figure 4.2(a)* shows (i) an empty stack; (ii) a list of the elements to be inserted on to stack; and (iii) a variable top which helps us keep track of the location at which insertion or removal of the item would occur.

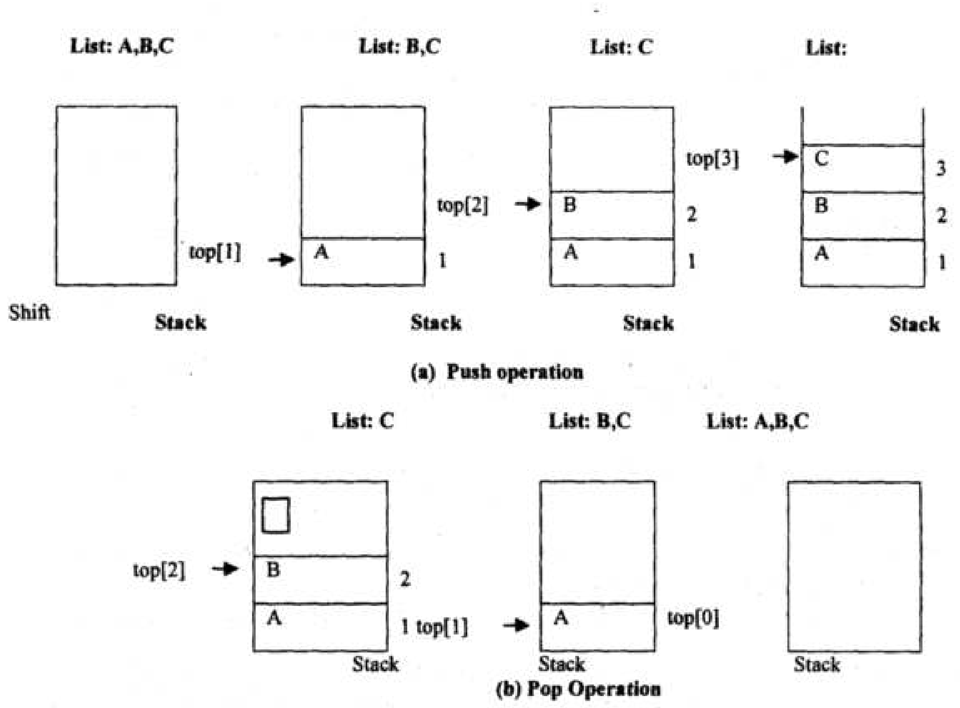


Figure 4.2: Demonstration of (a) Push operation, (b) Pop operation

Initially in *Figure 4.2(a),* top contains 0, implies that the stack is empty. The list contains three elements, A, B &C. In *Figure 4.2(b),* we remove an element A from the list of elements, push it on to stack. The value of top becomes I, pointing to the location of the stack at which A is stored.
Similarly, we remove the elements B & C from the list one by one and push them on to the stack. Accordingly, the value of the top is incremented. *Figure 4.2(a)* explains the pushing of B and C on to stack. The **top** now contains value 3 and pointing to the location of the last inserted element C.
On the other hand, *Figure 4.2(b)* explains the working of pop operation. Since, only the top element can be removed from the stack, in *Figure 4.2(b),* we remove the top element C (we have no other choice). C goes to the list of elements and the value of the top is decremented by 1. The top now contains value 2, pointing to B (the top element of the stack). Similarly, in *Figure 4.2(b),* we remove the elements B and A from the stack one by one and add them to the list of elements. The value of top is decremented accordingly.
There is no upper limit on the number of items that may be kept in a stack. However, if a stack contains a single item and the stack is popped, the resulting stack is called empty stack. The pop operation cannot be applied to such stacks as there is no element to pop, whereas the push operation can be applied to any stack.

4.2 ABSTRACT DATA TYPE-STACK

Conceptually, the stack abstract data type mimics the information kept in a pile on a desk. Informally, we first consider materials on a desk, where we may keep separate stacks for bills that need paying, magazines that we plan to read, and notes we have taken. We can perform several operations that involve a stack:

• start a new stack;
• place new information on the top of a stack;
• take the top item off of the-stack;
• read the item on the top; and
• determine whether a stack is empty. (There may be nothing at the spot where the stack should be).

When discussing these operations, it is conventional to call the addition of an item to the top of the stack as a push operation and the deletion of an item from the top as a **pop operation.** (These terms are derived from the working of a spring-loaded rack containing a stack of cafeteria trays. Such a rack is loaded by pushing the trays down on to the springs as each diner removes a tray, the lessened weight on the springs causes the stack to pop up slightly).

4.3 IMPLEMENTATiON OF STACK

Before programming a problem solution that uses a stack, we must decide how to represent a stack using the data structures that exist in our programming language. Stacks may be represented in the computer in various ways, usually by means of a one-way list or a linear array. Each approach has its advantages and disadvantages. A stack is generally implemented with two basic operations – push and pop. **Push** means to insert an item on to stack, The push algorithm is illustrated in *Figure 4.3(a).* Here, tos is a pointer which denotes the position of top most item in the stack. Stack is represented by the array **arr** and **MAXSTACK** represents the maximum possible number of elements in the stack. The pop algorithm is illustrated in *Figure 4.3(b).*

```
Step 1: [Check for stack overflow]
if tos >“MAXSTACK
print “Stack overflow” and exit
Step 2: [Increment the pointer value by one]
tos=tos+ I
Step 3: [Insert the item]
arr[tosj’value
Step 4: Exit
```

Figure 4.3(a): Algorithm to push an item onto the stack

The pop operation removes the topmost item from the stack. After removal of top most value tos is decremented by 1.

```
Step 1: [Check whether the stack is empty]
iftos=O
print "Stack underfiow" and exit
Step 2: [Remove the top most item]
value"arr[tos]
tos=tos- 1
Step 3: [Return the item of the stack]
return(value)
```

Figure 4.3(b): Algorithm to pop an element from the stack

4.3.1 Implementation of Stack Using Arrays

A Stack contains an ordered list of elements and an array is also used to store ordered list of elements. Hence, it would be very easy to manage a stack using an array. However, the problem with an array is that we are required to declare the size of the array before using it in a program. Therefore, the size of stack would be fixed. Though an array and a stack are totally different data structures, an array can be used to store the elements of a stack. We can declare the array with a maximum size large enough to manage a stack. Program 4.1 implements a stack using an array.

```
#include<stdio.h>
 int choice, stack[l 0], top, element;
void menuO;
void pushO;
void pope;
void showelementsO;
void main()
{ choiceelement=1;
topo;
menuO;
}
void menu()
{
printf("Enter one of the following options:\n");
printfi"PUSH l\n POP 2\n SHOW ELEMENTS 3\n EXIT 4\n"); scanf("%d", &choice);
if(choicel)
{
pushO; menuO;
}
if (choice=2)
{
popO;menuO;
}
if (choice==3)
```

```
{
showelementsO; menus;
}




} Stacks
void push()
{
if(top<9)
{
printf(t'Enter the element to be pushed to stack:\n");
scanf("%d", &element);
stack[top]=element;
H-top;
}
else
{
printf("Stack is full\n");
}
return;
}
void pop()
{
if(top>0)
{
--top;
element = stack[topj;
printf('Popped element:%d\n", element);
}
else
{
printfl"Stack is empty\n");
}
return;
}
void showelements()
{
if (top<=0)
printf("Stack is empty\n");
else
for(int i'O; i<top; ++i)
printf("%d\n", stack[i]);
}
```

Program 4.1: Implementation of stack using arrays

Explanation
The size of the stack was declared as 10. So, stack cannot hold more than 10 elements. The main operations that can be performed on a stack are push and pop. How ever, in a program, we need to provide two more options . namely, *showelements* and *exit. showelements* will display the elements of the stack. In case, the user is not interested to perform any operation on the stack and would like to get out of the program, then s/he will select *exit* option. It will log the user out of the program. *choice* is a variable which will enable the user to select the option from the push, pop, showelements and

exit operations. *top* points to the index of the free location in the stack to where the and Trees next element can be pushed. *element* is the variable which accepts the integer that has to be pushed to the stack or will hold the top element of the stack that has to be popped from the staók. The array *stack* can hold at most 10 elements. *push* and *pop* will perform the operations of pushing the element to the stack and popping the element from the stack respectively.
4.3.2 Implementation of Stack Using Linked Lists
In the last subsection, we have implemented a stack using an array. When a stack is implemented using arrays, it suffers from the basic limitatioti of an array – that is, its size cannot be increased or decreased once it is declared. As a result, one ends up reserving either too much space or too less space for an array and in turn for a stack. This problem can be overcome if we implement a stack using a linked list In the case of a linked stack, we shall push and pop nodes from one end of a linked list. The stack, as linked list is represented as a singly connected list. Each node in the linked list contains the data and a pointer that gives location of the next node in the list. Program 4.2 implements a stack using linked lists,

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
/ Definition of the structure node *7
typedef struct node
{
int data;
struct node *next;
/ Definition of push function *1
void push(node **tos,int item)
{
node *temp;
temp=(node*)malloc(sizeof(node)); / create a new node dynamically *7
if(temp=NULL) /* If sufficient amount of memory is /
{ / not available, the function malloc will *7
printf("\nError: Insufficient Memory Space"); 7* return NULL to temp / getch();
return;
}
else 7* otherwise/
{
temp->dataitem; 1* put the item in the data portion of node*/
temp>next=*tos; /*insert this node at the front of the stack *7
*tosemp; 7* managed by linked list*/
} /'end of function push*/
1* Definition of pop function */
```

```
int pop(node **tos)




Staiks
node *temp;
temp*tos;
mt item;
jf*tosNULL)
retum(NULL);
else
{
*tos(*tos>.>nt; /*To pop an element from stack*/
item=temp->data / remove the front node of the */
free(temp); 7* stack managed by L.L*/
return (item);
}
} /*j of function pop"!
7* Definition of display function */
void display(node *tos)
{
node *temptos;
if(tempNULL) /* Check whether the stack is empty*/
{
printf(\nStack is empty");
return;
}
else
{
while(temp!=NULL)
{
printf("\n/od",temp->data); /* display all the values of the stack*/
temptemp->next; 1* from the front node to the last node*/
}
}
) /*end of function display*/
7* Definition of main function */
void main()
{
mt item, ch;
char choice'y';
node *pNULL;
'do
{
```

```
clrscrO;
printf("\t\t\t\t* * * * *MENU* *
printf("\n\t\t\tl. To PUSH an element");
printf("\n\t\t\t2. To POP an element");
printf"\n\t\t\t3. To DISPLAY the elements of stack");
printf"\n\t\t\t4. Exit");
printf("\n\nn\t\t\tEnter your choice:-");
scanf("%d",&ch);
switch(ch)
{
```

```
case 1:
printf("\n Enter an element which you want to push "); scanf("%d",&item);
push(&p,item);
break;
case 2:
item,op(&p);
ifi:item!=NULL);
printf("\n Detected item is%d",item);
break;
case 3:
printf"\nThe elements of stack are");
display(p);
break;
case 4:
exit(0);
} /*SW closed */
printf("\n\n\t\t Do you want to run it again yin");
scanf("%c",&choice);
} while(choice'y');
}
/* end of function main*/
```

Program 4.2: Implementation of Stack **using Linked Lists**

Similarly, as we did in the implementation of stack using arrays, to know the working of this program, we executed it thrice and pushed 3 elements (10, 20, 30). Then we call the function display in the next run to see the elements in the stack.

Explanation

Initially, we defined a structure called *node.* Each node contains two portions, data and a pointer that keeps the address of the next node in the list. The *Push* function will insert a node at the front of the linked list, whereas *pop* function will delete the node from the front of the linked list. There is no need to declare the size of the stack in advance as we have done in the program where in we implemented the stack using arrays since we create nodes dynamically as well as delete them dynamically. The function *display* will print the elements of the stack.

N

12

4.4 ALGORITHMIC IMPLEMENTATION OF MULTIPLE STACKS

So far, now we have been concerned only with the representation of a single stack. What happens when a

data representation is needed for several stacks? Let us see an array X whose dimension is m. For convenience, we shall assume that the indexes of the array commence from 1 and end at m. If we have only 2 stacks to implement in the same array X, then the solution is simple.
Suppose A and B are two stacks. We can define an array stack A with n1 elements and an array stack B with n2 elements. Overflow may occur when either stack A contains more than n1 elements or stack B contains more than n2 elements.
Suppose, instead of that, we define a single array stack with n = n1 + n2 elements for stack A and B together. See the *Figure* 4.4 below. Let the stack A "grow" to the right, and stack B "grow" to the left. In this case, overflow will occur only when A and B together have more than n = n1 + n2 elements. It does not matter how many elements ii1dividually are there in each stack.

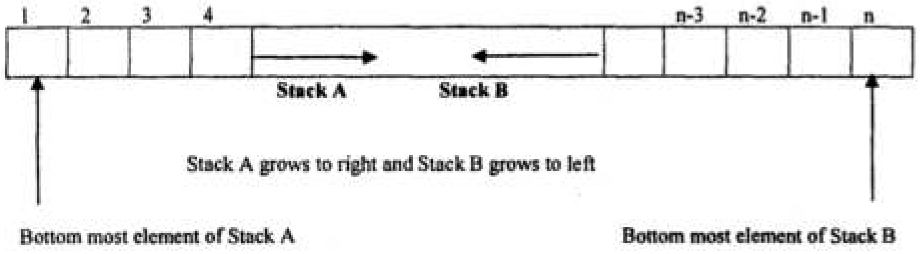

1
2
3
4
n-3
n-2
n-1
n
Stack A
Stack B
Stack A grows to right and Stack B grows to left
Bottom most element of Stack A
Bottom most element of Stack B

Figure 4.4: Implementation of multiple stacks using arrays

But, in the case of more than 2 stacks, we cannot represent these in the same way because a one-dimensional array has only two fixed points X(1) and X(m) and each stack requires a fixed point for its bottom most element. When more than two stacks, say n, are to be represented sequentially, we can initially divide the available memory X(l:m) into n segments. If thesizes of the stacks are known, then, we can allocate the segments to them in proportion to the expected sizes of the various stacks. If the sizes of the stacks are not known, then, X(i :m) may be dIvided into equal segments. For each stack i, we shall use BM (I) to represent a position one less-than the position in X for the bottom most element of that stack. TM(i), I I nwillpoint to the topmost element of stack i. We shall use the boundary condition BM (i) TM (i) iff the ii" stack is empty (refer to *Figure 4.5).* If we grow the i stack in lower memory

indexes than the i+ 1 · stack, then, with roughly equal initial segments we have

BM (i) = TM (i) = $\lfloor m/n \rfloor (i-1)$, $1 \le i \le n$, as the initial values of BM (i) and TM (i).

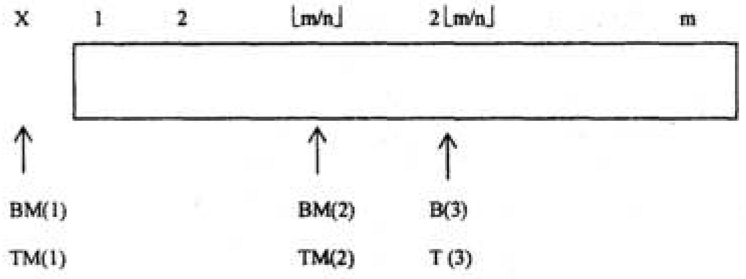


Figure 4.5: Initial configuration for n stacks in X(1:m)

All stacks are empty and memory is divided into roughly equal segments.
Figure 4.6 depicts an algorithm to add an element to the ith stack. *Figure 4.7* depicts an algorithm to delete an element from the **ith** stack.
ADE(i,e)

```
Step1: ifTM (i)=BM (1+1)
Print "Stack is full" and exit
Step2: [Increment the pointer value by one]
TM(i)←TM(i)+1
X←TM (i))E← e
Step3: Exit
```

Figure 4.6: Algorithm to add **an element to i stack**
//delete the topmost elements of stack i.

```
DELETE(i,e)
Step 1: if TM (i)=BM (i)
Print "Stack is empty" and exit
Step2: [remove the topmost item]
e- X(TM (i))
TM (i)-TM(i)-1
Step3: Exit
```

Figure 4.7: Algorithm to delete an **element from i stack**

4.5 APPLICATIONS

Stacks are frequently used in evaluation of arithmetic expressions. An arithmetic expression consists of operands and operators. Polish notations are evaluated by stacks. Conversions of different notations *(Prefix,* Postfix, Infix) into one another are performed using stacks. Stacks are widely used inside computer when recursive functions are called. The computer evaluates an arithmetic expression written in infix notation in two steps. First, it converts the infix expression to postfix expression and then it evaluates the postfix expression. In each step, stack is used to accomplish the task.

UNIT 5 QUEUES

5.0 INTRODUCTION

Queue is a linear data structure used in various applications of computer science. Like people stand in a queue to get a particular service, various processes will wait in a queue for their turn to avail a service. In computer science, it is also called a FIFO (first in first out) list. In this chapter, we will study about various types of queues.

5.2 ABSTRACT DATA TYPE-QUEUE

An important aspect of Abstract Data Types is that they describe the properties of a data structure without specifying the details of its implementation. The properties can be implemented independent of any implementation.in any programming language.
Queue is a collection of elements, or items, for which the following operations are defined:
createQueue(Q) : creates an empty queue **Q;**
isEmpty(Q): is a boolean type predicate that returns "true" if **Q** exists and Is empty, and returns "false" otherwise;
addQueue(Q,item) adds the given item to the queue **Q;**
and deleteQueue **(Q,** item) : delete an item from the queue **Q;**
next(Q) removes the least recently added item that remains in the queue **Q,** and returns it as the value of the function;

isEmpty (createQueue(Q)): is alwaysztrue, and **Queues**
deleteQueue(createQueue(Q)).: error

The primitive isEmpty(Q) is required toknow whetherthe queue is empty or not, because calling next on an empty queue should cause.an error. Like stack, the situation may be such when the queue is "full" in the case of a finite queue. But we avoid defining this here as it would depend on the actual length of the Queue defined in a specific problem.
The word "queue" is like the queue of customers at a counter for any service, in which customers are dealt with in the order in which they arrive i.e. first in first out (FIFO) order. In most cases, the first customer in the queue is-the first to be served.
As pointed out earlier, Abstract Data Types describe the properties of a structure without specifying an implementation in any way. Thus,. An algorithm, which works with a "queue" data structure, will work wherever it is implemented. Different implementations are usually of different efficiencies..

5.3 IMPLEMENTATION OF QUEUE

A physical analogy for a queue is a line at a booking counter. At a booking counter, customers go to the *rear* (end) of the line and customers are attendedto various services from the *front* of the line. Unlike stack, CustOmers are added at the rear end and deleted from the front end in a queue (FIFO).
An example of the queue in cornfiuter science is pfint jobs scheduled for printers. These jobs are maintained in a queue. The job fired forthe printer first gets printed first. Same is the scenario fôrjob scheduling in the CPU of computer.
Like a stack, a queue also (usually) holds data elementsof the same type. We usually graphically display a queue horizontally. *Figure 5.1* depicts a queue of 5 characters.

Figure 5.2: Queue of figure **5.1** after addition **of** new element

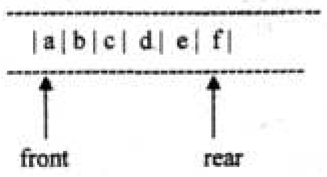


Figure 5.1: A queue of characters

The rule followed in a queue is that elements are added at the *rear* and come off of the *front* of the queue. After the addition of an element to the above queue, the position of rear pointer changes as shown below. Now the *rear* is pointing to the new element 'g' added at the rear of the queue(refer to *Figure 5.2*).

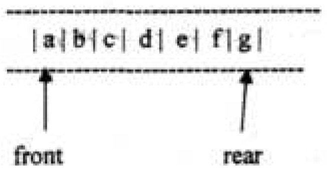


Figure 5.2: Queue of figure 5.1 after addition of new element

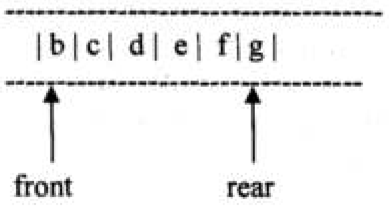


Figure 5.3: Queue of figure 5.2 after deletion of an element

Figure 5.3: Queue of figure 5.2 after deletion of an element

A lgorithm for addition of an element to the queue

Step 1: Create a new element to be added
Step 2: If the queue is empty, then go to step 3, else perform step 4
Step 3: Make the front and rear point this element
Step 4: Add the element at the end of the queue and shift the rear pointer to the newly added element.

Algorithm for deletion of an element from the queue

Step I: Check for Queue empty condition. If empty, then go to step 2, else go to step 3
Step 2: Message "Queue Empty"
Step 3: Delete the element from the front of the queue. If it is the last element in the queue, then perform *step* a else *step b*
a) make front and rear point to null
b) shift the front pointer ahead to point to the next element in the queue

5.3 1 Array **implementation of a queue**

As the stack is a list of elements, the queue is also a list of elements. The stack and the queue differ only in the position where the elements can be added or deleted. Like other liner data structures, queues can also be implemented using arrays. Program *5.1* lists the implementation of a queue using arrays.

```
#include "stdio.h"
#define QUEUE_LENGTH 50 struct queue
{  int element[QUEUE_LENGTH];  int front, rear, choice,x,y;

}

struct queue q;

mainO

{

int choice,x;
printf ("enter 1 for add and 2 to remove element front the queue")
printf("Enter your choice")
scanf("%d",&choice);
switch (choice)
{
case 1
printf ("Enter element to be added :");
scanf("%d",&x);
add(&q,x);
break;

case 2:
deleteO;
break;

}

}
add(y)
{
++q.rear;
if(q.rear < QUEUE_LENGTH)
q.element[q.rear] = y;

else
printf("Queue overflow")

}

delete()

{

if q.front> q.rear printf("Queue empty");
else{

x = q.element[q.front]; q.front++;

}

retrun x;

}
```

Program 5.1: Array implementation of a Queue

5.3.2 Linked List Implementation of a queue

The basic element of a linked list is a “record” structure of at least two fields. The object that holds the data and refers to the next element in the list is called a node (refer to *Figure 5.4).*

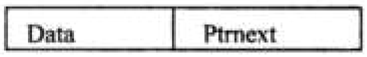


Figure 5.4: Structure of a node

The *data* component may contain data of any type. *Ptrnext* is a reference to the next element in the queue structure. *Figure 5.5* depicts the linked list representation of a queue.

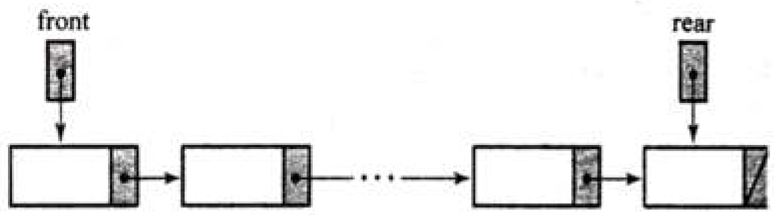


Figure 5.5: A linked list representation of a Queue

Program *5.2* gives the program segment for the addition of an element to the queue.
Program *5.3* **gives the** program **segment for the deletion of an element from the** queue.

```
add(int value)
{
struct queue Knew;
new (struct queue*)malloc(sizeof(queue));
new->value = value;
new->next = NULL;
if (front = NULL)
{
queueptr = new;
front = rear queueptr
)
else
{
rear->next = new;
rearnew;
}
}
```

Program 5.2: Program segment for addition of an element to the queue

```
delete()
{
 int delvalue =0;
if (front == NULL)

 printf(”Queue Empty”);
{
delvalue = front->value;
if (front->next =NULL)
{
free(front);
queueptr=front=rear=NULL;
}
else
{
front=front->next;
free(queueptr);
queueptr=front;
}
}
}
```

r **Program 5.3: Program segment for** deletion **of an** element from the queue

4 IMPLEMENTATION OF MULTIPLE QUEUES **Queues**

So far, we have seen the representation of a single queue, but many practical applications in computer science require several queues. Multiqueue is a data structure wheremuitipIequeues are maintained. This type of data structures arc used for process scheduling. We may use one dimensional array or multidimensional array to represent a multiple queue.

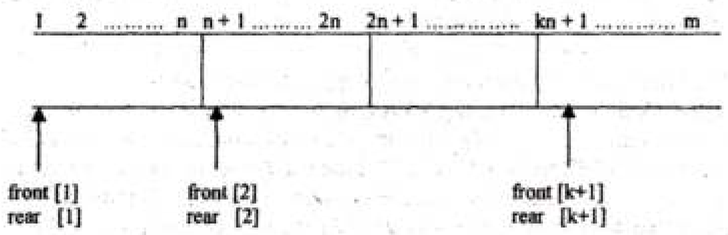

1 2 .... n n + 1 .... 2n 2n + 1 .... kn + 1 .... m
front [1]
rear [1]
front [2]
rear [2]
front [k+1]
rear [k+1]

Figure 5.6: Multiple queues in an array

A multiqueue implementation using a single dimensional array withm elements is depicted in *Figure 5.6.* Each queue has *n* elements which are mapped to a liner array of m elements.

ArrayImplcmentation of a multiqueue

Program *5.4* gives the **program** Segment using arrays for the addition of an element to a queue in the multiqueue.

```
addmq(i,x) / Add x to queue i *1
{
int i,x;
++rear[iJ;
if( rear[i] ==front[i+lJ)
printf'Queue is full");
else
{
rear[i] rear[i]+ 1;
mqueue[rearf ii] = x;
)
}
```

Program *5.4:* **Program segment for the addition of an element to the** queue

Program *5.5* gives the program segment for the deletion of an element from the queue.

```
delmq(i) / Delete an element from queue i *1
{
int i,x;
if( front[il = rear[i])
printf'Queue is empty");
{
x = mqueue[front[iJl;
front[i] = front[i}-l;
return x;
}
}
```

Program 5.5: Program segment for the deletion of an element from the queue

5.5 IMPLEMENTATIONF CIRCULAR QUEUES

One of the major problems with the linear queue is the lack of proper utilisation of space. Suppose that the queue can store 100 elements and the entire queue is full. So, it means that the queue is holding 100 elements. In case, some of the elements at the front are deleted, the element at the last position in the queue continues to be at the same position and there is no efficient way to find out that the queue is not full. In this way, space utilisation in the case of linear queues is not efficient. This problem is arising due to the representation of the queue.
The alternative representation is to depict the queue as circular. In case, we are representing the queue using arrays, then, a queue with *n* elements starts from index *0* and ends at *n-1.So,* clearly, the first element in the queue will be at index 0 and the last element will be at n-i when all the positions between index 0 and n-l(both inclusive) are filled. Under such circumstances, front will point to 0 and rear will point to n-i. However,, when a new element is to be added and if the rear is pointing to ri-I, then, it needs to be checked if the position at index 0 is free. If yes, then the element can be added to that position and rear can be adjusted accordingly. In this way, the utilisation of space is increased in the case of a circular queue.
In a circular queue, front will point to one position less to the first element anti-clock wise. So, if the first element is at position 4 in the array, then the front will point to position 3. When the circular queue is created,, then both front and rear point t& index 1. Also, we can conclude that the circular queue is empty in case both front and rear point to the same index. *Figure 5.7* depicts a circular queue.

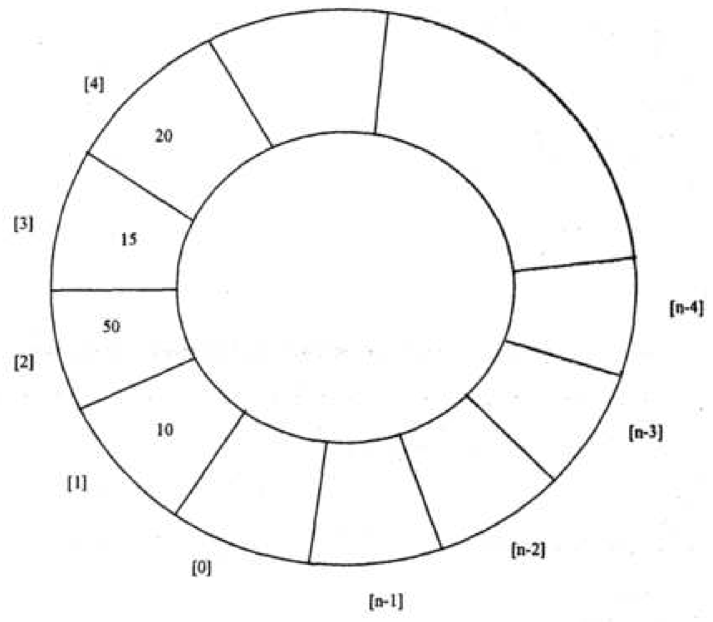


Figure 5.7 : A circular queue (Front = 0, Rear = 4)

Algorithm for Addition of an element to the circular queue: Queues

Step-i: If "rear" of the queue is pointing to the last position then go to step-2 or else *Step-3*

Step-2: make the "rear" value as 0

Step-3: increment the "rear" value by one

Step-4: a. if the *"front"* points where "rear" is pointing and the qieue holds a not NULL value for it, then its a "queue overflow" state, so quit; else go to step-b

b. add the new value for the queue position pointed by the "rear"

Algorithm for deletion of **an element from the circular queue:**

Step-1: If the queue is empty then say "queue is empty" and quit; else continue

Step-2: Delete the "front" element

Step-3: If the "front" is pointing to the last position of the queue then go to step-4 else go to step-5

to Step-4: Make the "front" point to the first position in the queue and quit

Step-5: Increment the "front" position by one

5.5.1 Array implementation of a circular queue

A circular queue can be implemented using arrays or linked lists. Program *5.6* gives the array implementation of a circular queue.

```
#include <stdio.h>
void add(int);
void deleteelement(void);
int max= 10; /*the maximum limit for queue has been set*/
static int queue[10);
int front0, rear-1; /*queue is initially empty*/
void main()
{
int choice x;
printf ("enter 1 for addition and 2 to remove element front the queue and 3 for exit");
printf("Enter your choice");
scanf("%d",&choice);
switch (choice)
{
case 1
prlntf ('Enter the element to be added :");
scanf("%d",&x);
add(x);
break;
case 2:
deleteelement();
break;
)
void add(int y)
{
if(rear==max-l)
rear=0;
else
```

```
rear=rear + 1;
if (front== rear && queue[front] != NULL)
printf("Queue Overflow");
else
queue[rear]=y;
}


void deleteelement( )
{
 int deleted_front =0;
if (front = NULL)
printf("Error Queue empty");
else
{
deleted_front = queue[front];
queue[frout NULL;
if(front==maxl)
'front 0;
else
front = front + 1;
}
}
```

Program *56:* Array implementation of **a circular queue**

5.5.2 Linked list implementation of a circular **queue**

Link list representation of a circular queue is more efficient as **it** uses space more **efficiently, of course with** the **extra cost of storing the** pointers. Program **5.7 gives the linked list representation of a circular queue.**

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

{

int value;
int *next

}
typedefstruct cq cqptr
cqptr p, front, rear
main()
{
int choice,x;
/* Initialise the circular queue *1
```

```
cqptr = front = rear = NULL;
printf ("Enter I for addition and 2 to delete element from the queue")
printf("Enter your choice")
scanf("%d",&choice);
switch (choice)
{
case 1:
printf ("Enter the element to be added :");
scanf("%d",&x);
add(&q,x);
break;
case 2:
delete();
break;
}
}


/**********Add element*****/
add(int value)
{
struct cq new;
new =(struct cq*)malloc(sizeof(queue));
new->value = value
new->next = NULL;
if (front==NULL)
{

cqptr=new;
front rear = queueptr;
}
else
{
rear->next = new;
rear=new;
}
}

/* *************** delete element *********/
delete()
{
```

```
int delvalue =0;
if (front ==NULL)
{ printf("Queue is empty");
delvalue = front->value;
if (front->next==NULL)
{
free(front);
queueptr = front rear = NUIJ.;
}
 else
{
front=front->next;
free(queueptr);
queueptr = front;
)
)
```

Program 5.7 ; Linked list Implementation of a Circular ueu

5.6 IMPLEMENTATION 01? DEQUEUE

Dequeue (a double ended queue) is an abstract data type similar to queue, where addition and deletion of elements are allowed at both the ends. Like a linear queue and a circular queue, a dequeue can also be implemented using arrays or linked lists.

5.6.1 Array implementation of a dequeue

If Dequeue is Implemented using arrays, then it will suffer with the same problems than a liner queue had suffered. Program 5.8 gives the array implementation of a dequeue.

```
#include<stdio.h>
#define QUEUE_LENGTH 10;
int dq[QUEUE_LENGTH];
int front, rear, choice,x,y;
main()
(
int choice,x:
front= rear=-1; /* 1nitlialise the front and rear to null i.e empty queue *1
printf ("enter 1 for addition and 2 to remove element from the front of the queue"); printf("enter 3 for addition and 4 to remove element from the rear of the queue"); printf("Enter yow choice");
scanf("%d",&choiee);
switch (choice)
```

```
case 1:
printf("Enter element to be added :");
scanf("%d",&x);
add_front(x);
break;


case 2:
delete_frontO;
break;
case 3:
printf ("Enter the element to be added :");
scanf(”%d ",&x);
add rear(x);
break;
case 4:
delete_rear();
break;
}
}
/*************** Add at the front
add_front(int y)
{
if (front ==0)
{
printf(”Element can not be added at the front44);
return;
else
{
front=front- 1;
dq[front] =y;
if (front ==-1) front= 0,;
}
*************
                    Delete from the front
delete front()
{
if (front==-1)
printf(”Queue empty”);
else
return dq[front]
if (front ==rear)
front =rear= -1
else
front= front + 1;
}
/*************** Add at the rear *******/
```

```
add_rear(int y)
if (front == QUEUE_LENGTH -1)
{
printf(”Element can not be added at.the rear “)
return;
else
{
rear= rear+l;
dq[rear]= y;
if(rear==-1)
rear=0;
}
}
/*************** Delete at the rear******/
delete_rear()
{
if rear ==-l
printf (“deletion is not possible from rear”);
else
{
if (front==rear)
front = rear=-l
else
{ rear=rear— 1;
return dq[rear];
}
}
}
```

Program 5.8: Array implementation of a Dequeue

542 Linked list implementation of a dequene

double ended queues are implemented with doubly linked lists.
A doubly link list can traverse in both the directions as it has two pointers namely left nd right. The right pointer points to the next node on the right where as the left pointer points tot1 previous node on the left. Program 5.9 gives the linked list Irnp1ementtion of a Dequeue.

```
# include ‘stdiQ.h’
#dcfinc NULL 0
struct dq {

int info;
int*left;
int *right;
| };
typdf tmct dq ctqptri
dqptr p, tp;
dqptr head;
 dqptr tail;
rnain()
{
int choice, I, x;
dqptr n;.
dqptr getnode ( );
printfl("\n Enter 1: Start 2 : Add at Front 3 Add atRear 4; Deletat Front 5:
Delete at Back");
while (1)
(
printf("\n 1: Start 2:AddatPront3:AddatBack4:DeleteatFront5:Delete at Back 6 : exit");
scanf("%d" &choice);
switch (choice)
{
case  1;
create list( );
break;
case 2:
eq_front( )
break;
case 3:
eq_back( );
break;
ease 4:
dq_front( );
break;
ease 5:
dq_back( );
break;
case 6:
exit(6);
}
}

}

creãte_list( )
{

int I, x;
dqptr t;
p = getnode( );
tp = p
p->left=getnode( );
```

```
p->info= 10;
p_right= getnode( );
return;
}
dqptr getnode( )
{
p =(dqptr) malloc(sizeof(struct dq));
return p;
}
dqernpty(dq q)
{
return q->head = NULL;
```

UNIT 3 LISTS

3.0 INTRODUCTION.

In the previous unit, we have discussed arrays. Arrays are data structures of fixed size. Insertion and deletion involves reshuffling of array elements. Thus, array manipulation is time-consuming and inefficient. In this unit, we will see abstract data type-lists, array implementation of lists and linked list implementation. Doubly and Circular linked lists and their applications. In linked lists. items can be added or removed easily to the end or beginning or even in the middle.

3.2 ABSTRACT DATA TYPE-LIST

Abstract Data Type (ADT) is a useful tool for specifying the logical properties of data type. An ADT is a collection of values and a set of operations on those values. Mathematically speaking, “a TYPE is a set,, and elements of set are Values of that type”.
ADT List
A list of elements of type T is a finite sequence of elements of type T together with the operations of create, update, delete, testing for empty, testing for full, finding the size, traversing the elements.
In defining Abstract Data Type, we are not concerned with space or time efficiency as well as about implementation details. The elements of a list may be integers, characters, real numbers and combination of multiple data types.
Consider a real world problem, where we have a company and we want to store the details of employees. To store this, we need a data type, which can store the type details containing names

of employee date of joining, etc. The list of employees may increase depending on the recruitment and may decrease on retirements or
termination of employees. To make it very simple and for understanding purposes, we are taking the name of employee field and ignoring the date of joining etc. The operations we have to perform on this list of employees are creation, insertion, deletion, visiting, etc. We define employee list as

```
typedef struct
{
char name[20];
S.
S..
) emp_list;
```

Operations on emp_list can be defined as

```
Create.emplist (emp list • emp_list)
{
Here, we will be writing create function by taking help of ‘C’ programming
language. .1
}
```

The list has been created and name is a valid entry in emplist. and position p
specifies the position in the list where name has to inserted

```
insert emplist (emp_list • emp_list, char name, int position )
{
‘ Here, we will be writing insert function by taking help of ‘C’ programming language. ‘/
)
delete emplist (emp_list • emp_list, char ‘name)
(
I Here, we will be writing delete function by taking help of’C’ programming
language. ‘1
visit_emplist (emp_list • emp_list)
{
I’ Here, we will be writing visit function by taking help of’C’ programming language.. 1
I,
```

The list can be implemented in two ways: the contiguous (Array) implementation and the linked (pointer) implementation. In contiguous implementation, the entries in the list are stored next to each other within an array. The linked list implementation uses pointers and dynamic memory allocation. We will be discussing array and linked list implementation in our next section.

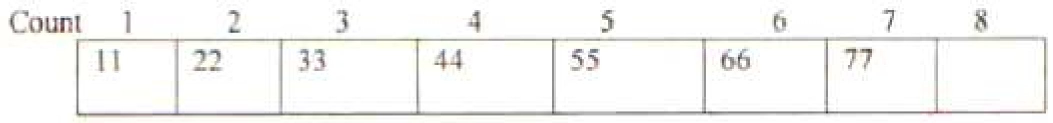
3.3 ARRAY IMPLEMENTATION OF LISTS

In the array implementation of lists, we will use array to hold the entries and a
separate counter to keep track of the number of positions are occupied. A structure will be declared which consists of Array and counter.

```
typedef struct
{
int count;
int entry[l00];
```

} list;

For simplicity, we have taken list entry as integer. Of course, we can also take list entry as structure of employee record or student record, etc.



Count	1	2	3	4	5	6	7	8
	11	22	33	44	55	66	77	

Insertion

In the array implementation of lists, elements are stored in continuous locations. To add an element to the list at the end, we can add it without any problem. But, suppose if we want to insert the element at the beginning or middle of the list, then we have to rewrite all the elements after the position where the element has to be inserted. We have to shift (n)th element to (n÷l)th position, where 'n' is number of elements in the list. The (n—I)th element to (n)h position and this will continue until the (r) th element to (r + 1) position, where 'r' is the position of insertion. For doing this, the count will be incremented.
From the above example, if we want to add element '35' after element 33'. We have to shift 77 to 8th position, 66 to 7th position, so on, 44 to 5d position.

Before Insertion

Count	1	2	3	4	5	6	7	
	11	22	33	44	55	66	77	

Step 1

Count	1	2	3	4	5	6	7	8
	11	22	33	44	55	66	77	77

Step 2

Count	1	2	3	4	5	6	7	8
	11	22	33	44	55	66	66	77

Step 3

Count	1	2	3	4	5	6	7	8
	11	22	33	44	55	55	66	77

Step 4

Count	1	2	3	4	5	6	7	8
	11	22	33	44	44	55	66	77

Step 5

Count	1	2	3	4	5	6	7	8
	11	22	33	35	44	55	66	77

Program 3.1 will demonstrate the insertion of an element at desired position

```
/ Inserting an element into contiguous list (Linear Array) at specified position /
Into contigous list,C 1
# include<stdioh>
/ definition of linear list 'I
typedef struct
{
int data[10]; int count;
}list;
/'prototypes of functions 1
void insert(list , int, int);
void create(list *);
void traverse(list *);


Definition of the insert function /


void insert list start, int position. int element)
{
```

```
int temp = start->count;
while( temp > position)
{
start->data[temp±I] = start->data[temp];
temp --;
)
start->data[position element;
start->count+-+;
}
! definition of create function to READ data values into the list ‘I
void create(list start)
int i=0, testl;
while(test)
{
push(stdin);
printf(”\n input value value for: %d:(zero to come out) “, i);
scanf(”%d”, &start->data[i]);
if(start->data[i]== 0)
test = = 0;
else
}
start->counti;
)
/ OUTPUT FUNCTION TO PRINT ON THE CONSOLE 1
void traverse(list start)
{
int i;
for(i =0; I< start->count;i++)
{
printf(’\n Value at the position: %d,%d,i, start->data[i]);
}
}
/*main function ‘I
void main()
{
int position, element;
list i;
create(& I);
printf(”’n Entered list as follows:\n”);
fflush(stdin);
traverse(& I);
fflush(stdin);
printf(”’.n input the position where you want to add a new data item:”); scanf(”°/od”. &position);
ff1ush( stdin):
printf”\n input the value for the position:”);
scanf(”°/od”, &element);
```

```
insert(&l, position, element);
traverse(&l);
}
```

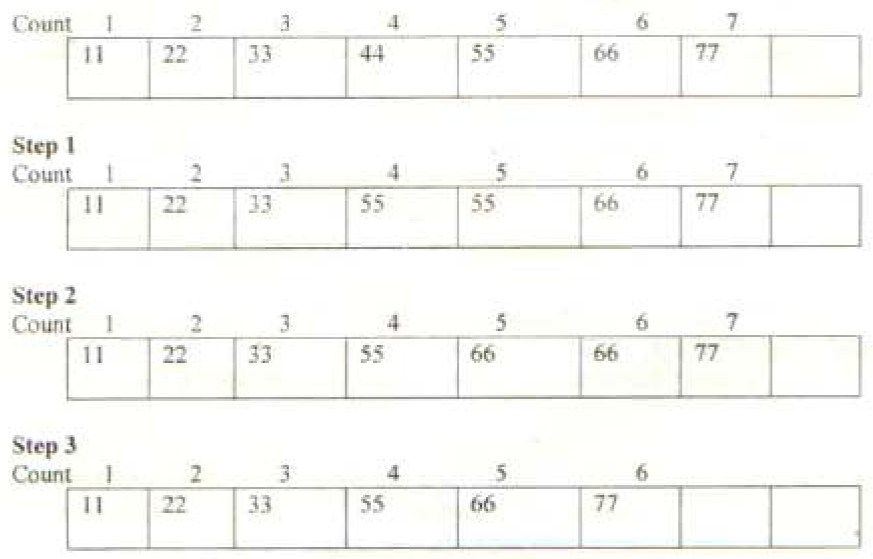
Program 3.1: Insertion of an cknicn into a linear array.

Deletion

To delete an element in the list at the end, we can delete it without any problem. But. suppose if we want to delete the element at the beginning or middle of the list, then, we have to rewrite all the elements after the position where the element that has to be deleted exists. We have to shift (r+ J)th element to fth position, where br is position of deleted element in the list, the (r + 2f element to (r + if position, and this will continue until the (n)e1ement to (n--I)th position, where n is the number of elements in the list. And then the count is decremented.

From the above example, if we want to delete an element '44' from list. We have to shifi 55 to 4th position, 66 to 5th position, 77 to 6th position.

Before deletion



Program 3.2 will demonstrate deletion of an element from the linear array

```
/* declaration of delete_list function */
void delete_list(list , int);
/ definition of delete_list functionf
/' the position of the element is given by the user and the element is deleted from the list 'I
void delete list(list 'start. mt position)
{

int temp = position;
 printf("\n information which we have to delete: %d",l->data[position]); while( temp <= start-
>count-1)
{
start->data[temp] start->data[temp+ 1];
temp ++;
}
start->count start-.>count-- 1
}
‘ main function /
void main()
{

printf(’\n input the position of clement you want to delete:”); scanf(”%d”,&position);
fflush(stdin);
delete_list( &1, position);
traverse(&I);
}
```

Program 32: Deletion of an element from the linear arra

3.4 LINKED LISTS - IMPLEMENTATION

The Linked list is a chain of structures in which each structure consists of data as well as pointer, which stores the address (link) of the next logical structure in the list.
A linked list is a data structure used to maintain a dynamic series of data. Think of a linked list as a line of bogies of train where each bogie is connected on to the next bogie. If you know where the first bogie is, you can follow its link to the next one. By following links, you can find any bogie of the train. When you get to a bogie that isn’t holding (linked) on to another bogie, you know you are at the end.
Linked lists work in the same way, except programmers usually refer to nodes instead of bogies. A single node is defined in the same way as any other user defined type or object, except that it also contains a pointer to a variable of the same type as itself.
We will be seeing how the linked list is stored in the memory of the computer. In the following Figure 3.1, we can see that start is a pointer which is pointing to the node which contains data as madan and the node madan is pointing to the node mohan and the last node babu is not pointing to any node. 1000,1050.1200 are memory addresses

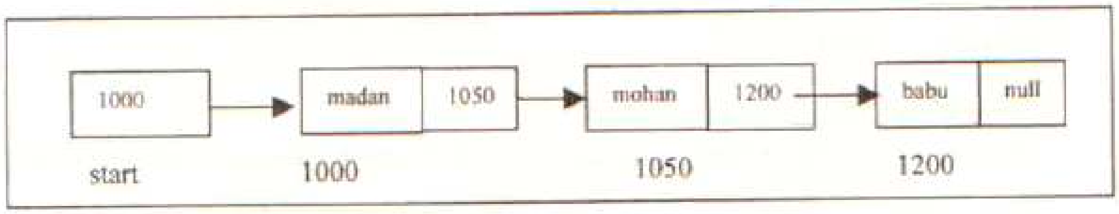


Figure 3.1: A Singly linked list

Consid
er the following definition:

```
typedef struct node
{
int data;
struct node ‘next;
} list;
```

Once you have a definition for a list node, you can create a list simply by declaring a pointer to the first element, called the “head’. A pointer is generally used instead of a regular variable. List can be defined as

list ‘head;

It is as simple as that! You now have a linked list data structure. It isn’t altogether useful at the moment. You can see if the list is empty. We will be seeing how to declare and define list-using pointers in the following program 3.3.

```
#include <stdio.h>

typedef struct node
{
   int data;
   struct node *next;
} list;

int main()
{
   list *head = NULL; /* initialize list head to NULL */
   if (head == NULL)
   {
      printf("The list is empty!\n");
   }
}
```

Program 3.3: Creation of a linked list

In the next example (Program 3.4), we shall look to the process of addition of new nodes to the list with the function create_list().

```
#include<stdio.h>
#include<stdlib.h>
#define NULL 0

struct linked_list
{
         int data;
         struct linked_list *next;
};
typedef struct linked_list list;

 void main()
 {
          list  *head;
          void create(list *);
          int count(list *);
          void traverse(list *);
          head=(list *)malloc(sizeof(list));
          create(head);
```

```
        printf(" \n traversing the list \n");
        traverse(head);
        printf("\n number of elements in the list  %d \n", count(head));
}

void create(list *start)
{
        printf("inputthe element -1111 for coming oout of the loop\n");
        scanf("%d", &start->data);
        if(start->data == -1111)
                start->next=NULL;
        else
        {
                start->next=(list*)malloc(sizeof(list));
                create(start->next);
        }
}

void traverse(list *start)
{
        if(start->next!=NULL)
        {
                printf("%d --> ", start->data);
                traverse(start->next);
        }
}

int count(list *start)

{
        if(start->next == NULL)
                return 0;
        else
                return (1+count(start->next));
}
```

<u>ALGOR1TIIM </u>(Insertion of element into a linked list)

Step 1 Begin
Step 2 if the list is empty or a new element comes before the start (head) element, then insert the new element as start element.
Step 3 else. if the new element comes after the last element, then insert the new element as the end element.
Step 4 else, insert the new element in the list by using the find function, find function returns the address of the found element to the insert_list
function.
Step 5 **End.**

Figure 3.2 depicts the scenario of a linked list of two elements and a new element which has to be inserted between **them.** *Figure* 3.3 depicts the scenario of a linked list after insertion of a new element into the linked list of *Figure* 3.2.

Before insertion

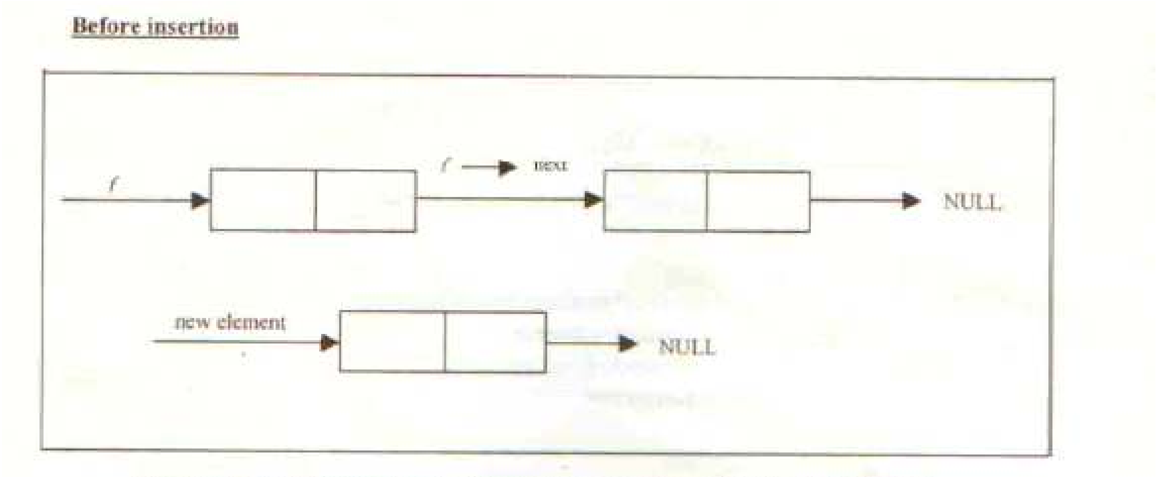


Figure 3.2: A linked list of two elements and an element that is to be inserted

After insertion

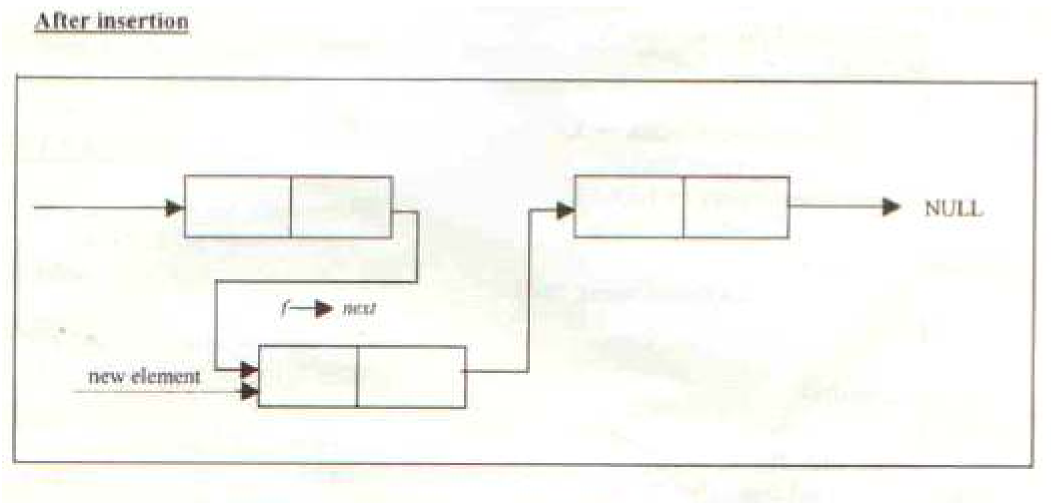


Figure 3.3: Insertion of a new element into linked list

Program 3.5 depicts the code for the insertion of an element into a linked list by searching for the position of insertion with the help of a find function.

```
INSERT FUNCTIO1b
‘prototypes of insert and find functions 1
list * insert_list(list)
list • find(list ‘, int);
/definition of insert function 1
list • insert_list(list start)

{

list n, f;
int key, element;
print f(”enter value of new element”); scanf(”%d”, &element);
printf”eneter value of key element”); scanfr’(%d” ,&key);
if(start->data =key)
n=(list*)mallo(sizeof(list));
n->data=element;
n->next = start;
start=n;

{
f=find(start, key);
if (f= NULL)
printf(”\n key is not found \n”);
else
{
n=(list )malloc(sizeof(list));
n->data=element;
n->next=f->next;
f->nextn;
}
return(start);
)
/*definition of find function */
list • find(list start, int key)
{
if(start->next->data =key)
return(start);
if(start->next->next =NULL)
return(NULL);
else
find(start->next, key):
void main()
{
list head;
void creatc(list );
int count(list 9;
void traverse(list 9;
head=(list 4)malloc(sizeof(list));
create(head);
printft" \n traversing the created list \n");
traverse( head);
printf'\n number of elements in the list %d \n", count(head)); headinsert_list(head);
printf" \n traversing the list after insert '.n");
traverse(head);
}
```

Program 3.5: InsertIon of an element into a linked list at a specific position

```
ALGORIThM (Deletion of an element from the linked list)
Step I Begin
Step 2 if the list is empty, then element cannot be deleted
Step 3 else, if element to be deleted is first node, then make the stan (head) to point to the second element.
Step 4 else, delete the element from the list by calling find function and returning the found address of the element.
Step S End
```

Figure 3.4 depicts the process of deletion of an element from a linked list.

After Deletion

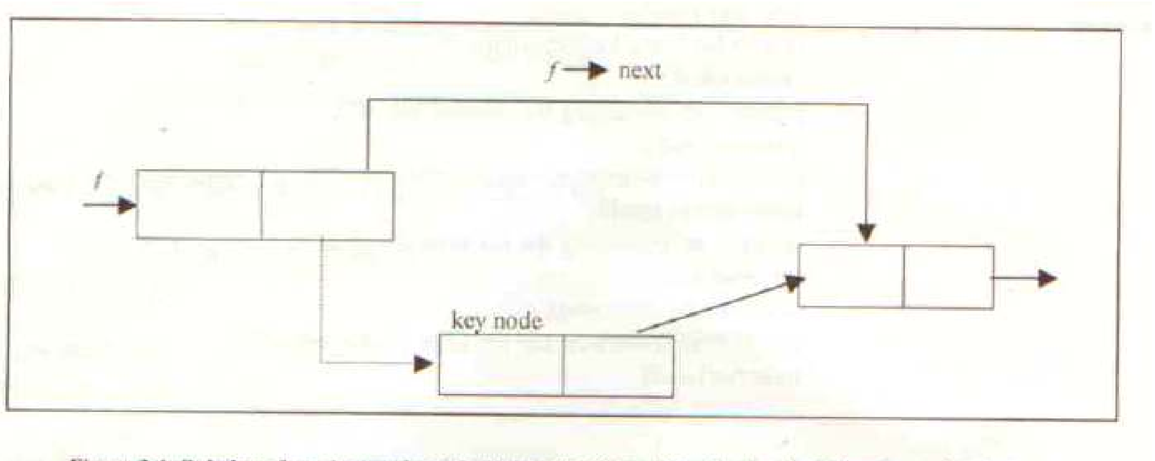


Figure 3.4: Deletion of an element from the linked list (Dotted line depicts the link prior to deletion)

Flure 3.4: Deletion of an element front the linked list plotted line depicts the link prior to deletion)

Program 3.6 depicts the deletion of an element from the linked list. It includes a function, which specifically searches for the element to be deleted.

DELETE LIST FUNCTION

```
        /*  prototype of delete_function */
        list *delete_list(list *);
        list *find(list *, int);

        /*definition of delete_list */
        list *delete_list(list *start)
        {
                int key; list * f, * temp;
                printf("\n enter the value of element to be deleted \n");
                scanf("%d", &key);
                if(start->data == key)
                {
                        temp=start->next;
                        free(start);
                        start=temp;
                }
                else
                {
                        f = find(start,key);
                        if(f==NULL)
                                printf("\n key not fund");
                        else
                        {
                                temp = f->next->next;
                                free(f->next);
                                f->next=temp;
                        }
                }
                return(start);

    void main()
    {

            list  *head;
            void create(list *);
        int count(list *);
        void traverse(list *);
        head=(list *)malloc(sizeof(list));
        create(head);
        printf(" \n traversing the created list \n");
        traverse(head);
        printf("\n number of elements in the list  %d \n", count(head));
        head=insert(head);
        printf(" \n traversing the list after insert \n");
        traverse(head);
        head=delete_list(head);
        printf(" \n traversing the list after delete_list \n");
        traverse(head);
}
```

3.5 DOUBLY LINKED LISTS-IMPLEMENTATION

In a singly linked list, each element contains a pointer to the next element. We have seen this before. In single linked list, traversing is possible only in one direction. Sometimes, we have to traverse the list in both directions to improve performance of algorithms. To enable this, we require links in both the directions, that is, the element should have pointers to the right element as well as to its left element. This type of list is called doubly linked list.

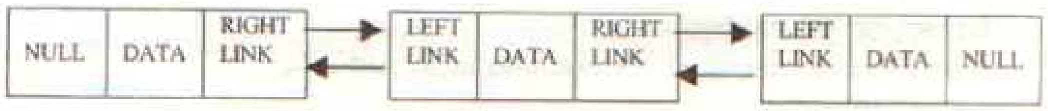


Figure 3.5: A Doubly Linked List

Doubly linked list (Figure 35) is defined as a collection of elements, each element consisting of three fields:
• pointer to left element,
• data field, and
• pointer to right element.

Left link of the leftmost element is set to NULL which means that there is no left element to that. And, right link of the rightmost element is set to NULL which means that there is no right element to that.

ALGORITHM (Creation)

```
Step 1  begin
Step 2  define a structure ELEMENT with fields
                    Data
              Left pointer
             Right pointer
Step 3declare a pointer by name head and by using (malloci)) memory allocation function
allocate space for one element and store the address in head pointer
Head = (ELEMENT ) malloc(sIzeof(ELEMENT))

Step 4 read the value for head->data
head->Ieft = NULL
head->right (ELEMENT ) malloc(sizc of(ELEMEN11)
Step 5 repeat step3 to create required number of elements
Step 6 end
```

Program 3.7 depicts the creation of a Doubly linked list.

```
/ CREATION OF A DOUBLY LINKED List
i DBLINK.C 1
# include <stdio.h>
# include <malloc.h>
struct di_list
{
int data;
struct di_list right:
struct di_list left;
typedef struct di_list dust;
void di_create (dust );
void traverse (dust 9;
I Function creates a simple doubly linked list 'I
void dI_create(dlist start)
{
printf("\n Input the values of the element -111110 come out: "); scanf"%d", &start->data);
igstart->data !=-1 Ill)
{
start->right (dust ')malloc(sizeof(dlist));
start->right->lefi = stait
start->right->right NULL:
dl_create(start->right);
)
else
start->right = NULL;
}
/ Display the list /
void traverse (dust start)
{
printf"\n traversing the list using right pointer\n");
do{
```

```
printf” %d = “, start->data);
start start->right;
} while (start->right); I. Show value of last start only one time /
printf”’n traversing the list using left pointern”);
start=s tart-> left;
do
printf” %f “, start->data);
start = start->left;
) while(start->right);
}
void main()
{
dust ‘head;
head (dust 9 malloc(sizeof(dlist));
head-> left=N U LL;
head->right=N U LL;
dlcreate(head);
prinif(”n Created doubly linked list is as follows”);
travcrse( head);
}
```

Program 3.7: Creation of a Doubly Linked LIst

OUTPUT

```
Input the values of the element .1 111 to conic out:
1
Input the values of the element -1111 to come out:
input the values of the clement -1111 to come out:
Tnput the values of the element -1111 to come out:
—liii
Created doubly linked list is as follows
traversing the list using right pointer
1=2=3=
traversing the list using left pointer
32=l=
```

3.6 CIRCULARLY LINKED LISTS

1M ILEMENTATION

A linked list in which the last element points to the first element is called CIRCULAR linked list.
The chains do not indicate first or last element: last element does not contain the NULL pointer.
The external pointer provides a reference to starting element.
The possible operations on a circular linked list are:

• Insertion,

• Deletion, and
• Traversing
Figure 3.6 depicts a Circular linked list.

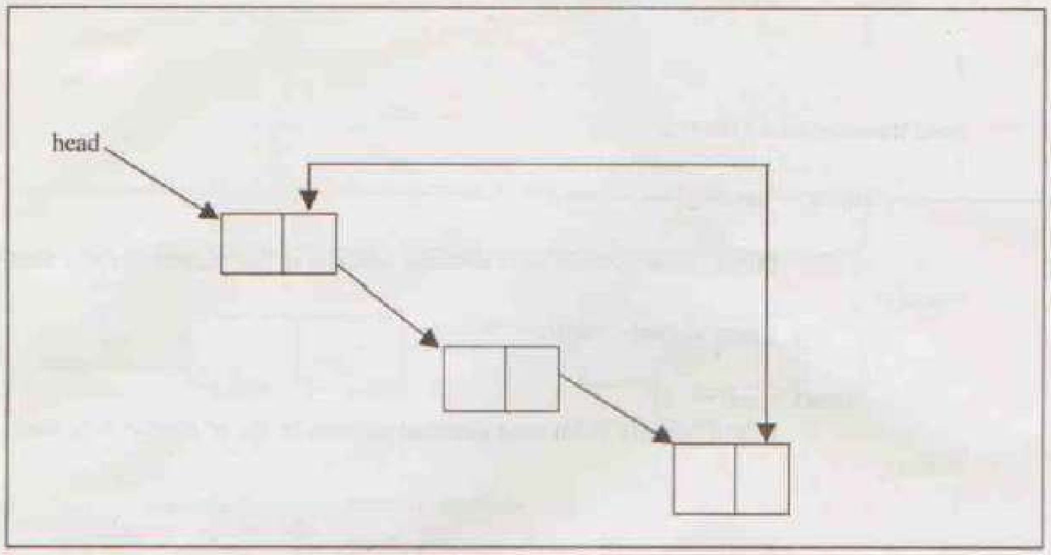


Figure 3.6 : A Circular Linked List

Figure .3.6 : A Circular linked Lht
Program 3.8 depicts the creation of a Circular linked list

```
#include<stdio.h>
#include<st4lib.h>
#define NULL 0
struct linked list
{
int data;
struct linked list next;
typedefstruct linked list clist;
clist hcad, s;
void main()
{
void create clist(clist 9;
int count(clisi 9;
void travcrse(clist 9;
head(clist ‘)malloc(sizeof(clist));
shead;
create clist( head);
printf(” n traversing the created clist and the starting address is %u \n”, head);
traverse(head);
printft(”\n number of elements in the clist O,/j \n”, count(head));
)
void create clist(clist start)
{
printf(”input the element -1111 for coming out of the loop\n’); scan f(O/odH, &start->data);
ifstart->data=-llll)
start-.>next=s;
else
I
start->next=(clist )malloc(sizeof(cl 1st));

create clist(start>next);

)
void traverse(clist start)
{
if(start->next!=s)
{
printf(”data is %d \t next element address is %u\n”, start->data, start >next); traverse(start-
>next);
}
if(start.>next = s)
printf(”data is %d \t next element address is %u\n’1start->data, start >next); int count(clist* start)
{
if(start->next s)
return 0;
else
return( 1 +count(start.>next));
```

Program 3.8: Creation of a Circular linked list

```
ALGORITHM (Insertion of an element into a Circular Linked List)
Step I Begin
Step 2 if the list is empty or new element comes before the start (head) element, then insert the
new element as start element.
Step 3 else, if the new element comes after the last element, then insert the new element at the
end element and adjust the pointer of last element
to the start element.
Step 4 else, insert the new element in the list by using the find function, find function returns the
address of the found element to the insert_list
function.
Step 5 End.
```

If new item is to be inserted after an existing element, then, call the find function recursively to trace the ‘key’ element. The new element is inserted before the ‘key’ element by using above algorithm.

Figure 3.7 depicts the Circular linked list with a new element that is to be inserted.

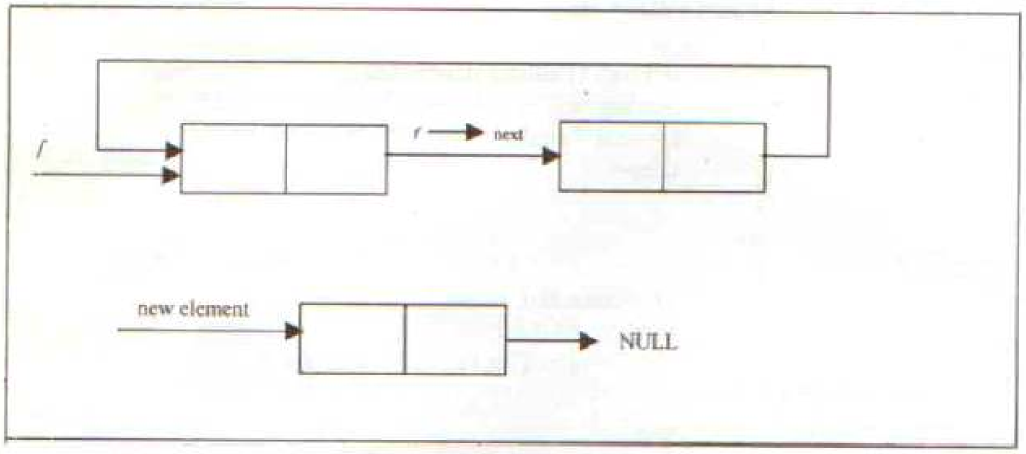


Figure 3.7: A Circular Linked List and a node that is to be inserted into it.

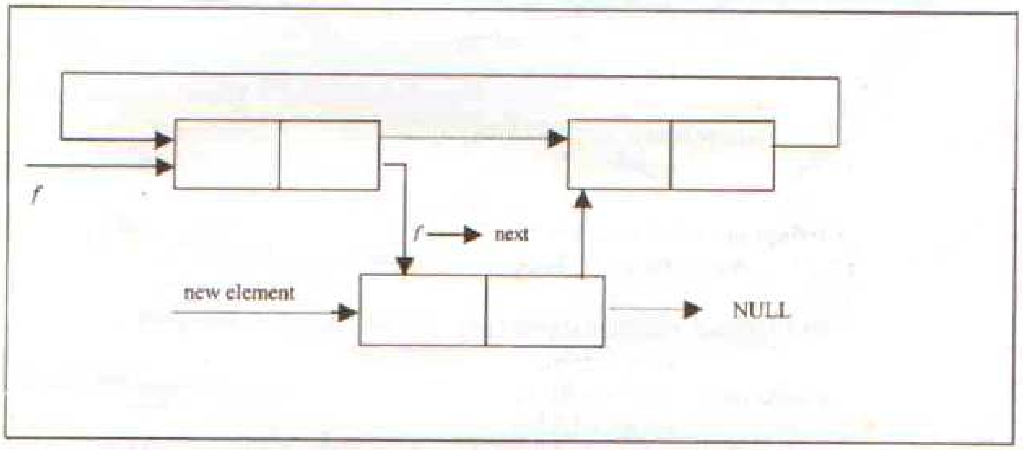


Figure 3.8 depicts a Circular linked list with the new element inserted between first and second nodes of Figure 3.7.

Program 3.9 depicts the code for insertion of a node into a Circular linked list.

```
#include<stdio.h>
#include<stdlib.h>
#detine NULL 0
struct linked list
int data;
struct linked list next;
typedefstruct linked_list clist;
clist head, s;
```

```
I. prototype of find and insert
clist • find(clist , int);
clist • insert clist(clist 9;
/definition of insert_clist function 1
clist insert_clist(clist start) {

clist n. *nl;
int hey, x,
printf(’enter value of new
scanfl:”%d”, &x);
printf(”eneter value of key element”);
scanfr%d”,&key);
iRstart->data =key)
{
n=(clist )malloc(sizeof(clist));
n->data=x;
n->next = start;
start=n;
)
else
{
ni find(start, key);
ifnl = NULL)
printf”\n key is not found\n”);
else
{
n=(clist )malloc(sizeof(clist));
n->data=x;
n->nextn I ->next;
n I ->nextn;
}
}
return(slart);
)
/definition of find function ‘F
clist find(clist start. Tnt key)
{
if(start->next->data key)
retum(srart);
ifstart->next->ncxt = NULL)
return(NU LL);
else
Ii nd( start->next, key);
}
void main()
{
```

```
void create_ciist(clist );
mt count(clist );
void traverse(clist );
head(clist • )malloc(sizeof(clist));
shead;
create_cl ist( head);
printf(" \n traversing the created clist and the starting adaress is %u \n", head);
traverse(head);
printf("\n number of elements in the clist °/od \n", count(head)); headinsertclist(head);
printf("\n traversing the clist after insert_clist and starting address is %u n"head);
traverse(head);


Leg'

    void create clist(clist 'start)
    {
    printf("inputthe element -Il I for coming OOUt 01 the loop\n"); scanf("%d", &start->data);
    ifstart->data -liii)
    start->nex I
    else
    start-next(cIist')maBoc(si zecl 1st));
    create ci ist(starl->next);
    }
    }
    void traverse(clist 'start)
    i1(start->next!s)
    •1
    printf("data is %d •t next element address is %u\n", start->data. start >next); traverse(start->
    next);
    )
    itstart->next = s)
    printf("data is %d 't next element address is %u\n",start->data, start >next): }
    mt count(clist start)
    {
    if(start->next = s)
    return 0;
    else
    return( I +couflt(stajl->next));
    }
```

Program 3.9 Insertion of a node into a Circular Linked List

Figure 3.9 depicts a Circular linked list from which an element was deleted.

ALGORITHM (Deletion of an element from a Circular Linked List)

```
Step 1 Begin
Step 2 if the list is empty, then element cannot be deleted.
```

Step 3 else, if element to be deleted is first node, then make the start (head) to point to the second element.
Step 4 else, delete the element from the list by calling find function and returning the found address of the element.
Step S End.

Program 310 depicts the code for the deletion of an element from the Circular linked
.

```
#include<stdio.h>
include<stdio.h>
#define NULL 0
struct linked list
int data;
struct linked list next;
typedefstruct linked list clist,
clist head, s;
/ prototype of find and delete functionf
clist delete clist(clist 9;
clist • tind(clist , hit);
/detinition of delete_clist Sf
clist ‘delete_clist(clist stan)
{
int key; clist • temp;
printf”n enter the value of element to be deleted \n”);
scan fl”%d”, &key);
itlstart->data key)
{
Iemp=start->next;
free( start);
start=temp;
}
else
{
f find(starikey);
if(f==NULL)
printfl”\n key not fund”);
else
{
temp = f->next->ncxt;
free(f->next);
f->nexttemp;
}
}
return(start);
)
```

```
/*definit ion of find function*/
clist • find(clist stan, irn key)
{
i1(start->next->data key)
return(start);
in:start->next->next = NULL)
return(N ULL);
else
find(start->next, key);
)
void main()
{
void create clist(clist 9:
mt count(clist 9;
void traverse(clist 9;
head=(clist ‘9rnalloc(sizeof(clist));
s=head;
created ist( head);
printft” ‘n traversing the created clist and the starting address is %u ‘n”, head);
traverse(head);
printt”\n number of elements in the clist %d n”, count(head)):
head=deletecl ist( head);
printf(” n traversing the clist after dekte_clistand starting address is %u n” head);
traverse( head);
}
void create_dllst(clist *start)
{
printfinputthe element -1111 for coming oout of the loop\nH); scanf(0%d”, &sart->data);
if(start->data -liii)
start.>nexts;
else
{
start->nexl=(clist )mal loc(sizeof(cI 1st)):
create_cl ist(s(art->next);
}
}
void traversetcl,st start)
{
I fstart->next !=s)
{
prinlfC’data is °‘od \t next element address is %u\n”, start->data,
siart-’ next);
traverse( start->next);
I
if(start->next s)
printf(”data is %d ‘t next element address is %u\n”,start->data,
start->next);
}
mt count(clist start)
{
iftstart->next s)
return 0;
else
return( I +count(start->nex);
I
```

Progrim 3.10: Deletion of iu element rrom (he circulir linked liii

3.7 APPLICATIONS

Lists are used to maintain POLYNOMIALS in the memory. For example, we have a function f(x)= 7x5 +9k" — 6x' + 3x2.

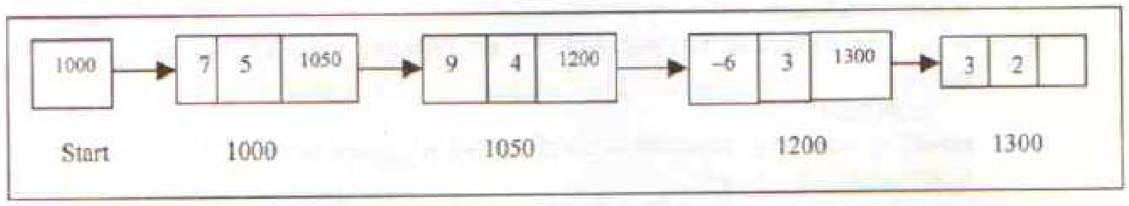


Figure 3.10: Representation of a Polynomial using a singly linked list

1000,1050,12001300 Polynomial contains two components, coefficient and an exponent, and 'x' is a formal parameter. The polynomial is a sum of terms, each of which consists of coefficient and an exponent. In computer, we implement the polynomial as list of structures consisting of coefficients and exponents.

Program 3.11 accepts a Polynomial as input. It uses linked list to represent the Polynomial. it also prints the input polynomial along with the number of nodes in it.

```
/ *Representation of Polynomial using Linked List */ include <stdio.h>
include <mal loch>
struct link
{
char sign;
int coef;
int expo;
struct link next;
typedef struct link poly;
void insertion(poly);
void create_poly(poly);
void display(poly);
/' Function create a ploynomial list 1
void create_poly(poly,start)
{
char ch;
static int i;
printf'("\n input choice n for break: ");
ch getchar();
if(ch!= 'n')
{
printf("\n Input the sign: %d: ",i+ 1);
scan f( "%c", &start->sign);
printf"\n Input the coefficient value: %d: ', 1+1);
scanf("%d", &start->coef);
printf"\n Input the exponent value: %4: ",1+1);
scanf( "%", &start->expo);
push(stdin);
start->next = (poly 9 malloc(sizeof(poly));
create_poly(start->next);

else
start-> next= NULL;
)
1 Display the polynomial /
void display(poly start)
{
if(start->next ! NULL)
{
printf" O/", start->sign);
printfl" %d", start->coet);
printf("X'%d", start->expo);
display(start->next);
}
}
I' counting the number of nodes */
int count_poly(poly 'start)
{
if(start->next== NULL)
return 0;
else
return( i+countpoly(start->next));
)
I, Function main 1
void main()
poly head = (poly ') malloc(sizeoftpoly));
create_poly( head);
printf("\n Total nodes = %d sn", countpoly(head));
display(head); }
```

Program*m 3.11: Representation of Polynomial using Linked list

STRINGS

A string is an array of characters. They can contain any ANSCII characters and are useful in many operations. A characters occupies a single byte. Therefore a string of length N characters requires N bytes of memory. Since strings do not use bounding indexes it is important to mark their end. Whenever enter key is pressed by the user the compiler treats it as the end of string. It puts a special character '\0' (NULL) at the end and uses it as the end of the string marker there onwards.

When the function scanf() is used for reading the string, it puts a '\0' character when it receives space. Hence if a string must contain a space in it we should use the function gets().

3.4. STRING FUNCTIONS

Let us first consider the functions, Which are required for general string operations. The string functions are available in the header file "string.h". We can also write these ourselves to understand their working. We can write these functions using

1. Array of Characters

2. Pointers.

3.4.1. String Length

The length of the string is the number of characters in the string, which includes spaces, and all ASCII characters. As the array index starts at zero, we can say the position occupied by '\0' indicates the length of that string. Let us write these functions in two different ways mentioned earlier.

3.4.1.1. Using Arrays

```
int strlen1(char s[])

{

    int i=0;

        while(s[i] != '\0')

          i++;

     return(i);

 }
```

Here we increment the position till we reach the end of the string. The counter contains the size of the string.

3.4.1.2. Using Pointers

```
 int strlen1(char *s)
{
 char *p;
 p=s;
while(*s != '\0')
s++;
return(s-p);
};
```

The function is called in the same manner as earlier but in the function we accept the start address in s. This address is copied to p. The variable s is incremented till we get end of string. The difference in the last and first address will be the length of the string.

3. STRING COPY : COPY S2 TO S1.

In this function we have to copy the contents of one string into another string.

3.5.1. Using Array

```
 void strcopy(char s1[], char s2[])
{
 int i=0;
while(s2[i] != '\0')
s1[i]=s2[i++];
s1[i]='\0'
}
```

Till i th character is not '\0' copy the character s and put a '\0' as the end of the new string.

3.5.2.Using Pointers

```
  void strcpy(char *s1, char *s2)
```

```
{
  while(*s2)
      {
        *s1 = *s2;
           s1++;
            s2++;
       }
*s1 = *s2;
}
```

3.6.STRING COMPARE

3.6.1.Using Arrays

```
 void strcomp(char s1[], char s2[])
  {
    int i = 0;
    while(s1[i] != '\0' && s2[i] != '\0')
    {
      if(s1[i] != s2[i])
          break;
        else
             i++ ;
      }
return(s1[i] – s2[i]);
}
```

The function returns zero, if the two strings are equal. When the first string is less compared to second, it returns a negative value, otherwise a positive value.

The reader can write the same function using the pointers.

3.7. CONCATENATION OF S2 TO THE END OF S1

At the end of string one add the string two. Go till the end of the first string. From the next position copy the characters from the second string as long as there are characters in the second string and at the end close it with a '\0' character. This is left as an exercise for the student.

UNIT 9 SEARCHING

Searching -is the process of looking for something: Finding one piece of data that has been stored within a whole group of data. It is often the most time-consuming part of many computer programs. There are a variety of methods, or algorithms, used to search for a data item, depending on how much data there is to look through, what kind of data it is, what type of structure the data is stored in, and even where the data is stored - inside computer memory or on some external medium.
Till now, we have studied a variety of data structures, their types, their use and so on. In this unit, we will concentrate on some techniques to search a particular data or piece of information from a large amount of data. There- are basically two types of searching techniques, Linear or Sequential Search and Binary Search.
Searching is very common task in day-to-day life, where we are involved some or other time, in searching either for some needful at home or office or market, or searching a word in dictionary . In this unit, we see that if the things are organized in some manner, then search becomes efficient and fast.
All the above facts apply to our computer programs also. Suppose we have a telephone directory stored in- the memory in an array which contains Name and Numbers. Now, what happens if we have to find a number? The answer is search that number in the array according to name (given). If the names were organized in same order, searching would have been fast.
So, basically a search algorithm is an algorithm which accepts an argument ‘a ‘ and tries- to find the corresponding data where the match of ‘a’ occurs in a file or in a

9.2 LINEAR SEARCH

Linear search is not the most efficient way to search for an item in a collection of items. However, it is very simple to implement. Moreover, if the array elements are arranged in random order, it is the only reasonable way to search. In addition, efficiency becomes important only in large arrays; if the array is small, there aren’t many elements to search and the amount of time it takes is not even noticed by the user. Thus, for many situations, linear search is a perfectly valid approach.
Before studying Linear Search, let us define some terms related to search.
Afile is a collection of records and a record is in turn a collection of fields. A field,

which is used to differentiate among various records, is known as a 'key'.
For example, the telephone directory that we discussed in previous section can be considered as a file, where each record contains two fields: name of the person and phone number of the person.
Now, it depends on the application whose field will be the 'key'. It can be the name of person (usual case) and it can also be phone number. We will locate any particular record by matching the input argument 'a' with the key value.
The simplest of all the searching techniques is Linear or Sequential Search. As the name suggests, all the records in a file are searched sequentially, one by one, for the matching of key value, until a match occurs.
The Linear Search is applicable to a table which it should be organised in an array. Let us assume that a file contains 'n' records and a record has 'a' fields but only one key The values of key are organised in an array say 'm'. As the file has 'n' records, the size of array will be 'n' and value at position R(i) will be the key of record at position i. Also, let us assume that 'el' is the value for which search has to be made or it is the search argument.
Now, let us write a simple algorithm for Linear Search.
Algorithm
Here, m represents the unordered array of elements
n represents number of elements-in the array and
el represents the value to be searched in the list

```
Sep 1: [Initialize]
k=O
flag =1
Step 2: Repeat step 3 for k=0,1,2… n-i
Step 3: if(m[k]=el)
then
flag=0
print "Search is successful" and element is found at location (k+ 1)
stop
endif
Step 4: if (flag=i) then
print "Search is unsuccessful"
endif
 Step 5: stop
```

Program 9.1 gives the program for Linear Search.

```
/*program for Linear Search/
/*Heer Files*/
#include<stdio.h>
#include<conio.h>
/*Global Variables*/
```

```
int search;
 int flag;
1* Function Declarationsf
jut input (int *, jut, jut);
void linear search (int , int, int);
void display (int , jut);
1* Functions /
void linear_search(int m[], jut n, int el)
{
int jut k;
flag 1;
for(k=O; k<n; k++)
{
if(m=[k]=el])
{
printf(”\n Search is Successful\n”);
printf(”\n Element: %i Found at location : %i”, element, k+ 1); flag 0;
}
}
if(flag==l)
printf(4’\n Search is unsuccessful”);
}
void display(int m[ 1’ int n)
{
jut i;
for(i=0; i< 20; i++)
{
printf(”%d”, m[i];
}
}
int input(int m[ J, jut u, int el)
{
int i;
u = 20;
e130;
printf(”Number of elements in the list: %d”,.n);
for(i=0;i<20;i++)
{
m[i]rand( )=%100;
}
printf(”\n Element to be searched :%d”, el);
search = el;
return u;
}
/ Main Function*/


void main()
{
int n, el, m[200];
number = input(m, n,el);
el= search;
printf(”\n Entered list as follows: ‘n”);
display(m, n);
linear_search(m, n, el);
printf(”\n In the following list\n”);
display(m, n);
}
```

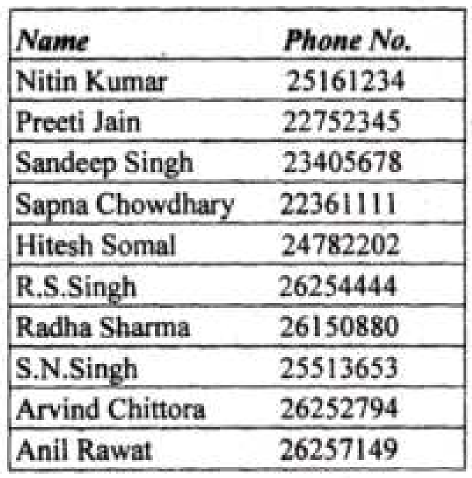
Program 9.1: Linear Search

Program 9.1 examines each of the key values in the array ‘m’, one by one and stops when a match occurs or the total array is searched.

Example:

A telephone directory with n = 10 records and Name field as key. Let us assume that the names are stored in array ‘m’ i.e. m(0) to m(9) and the search has to be made for name “Radha Sharma”, i.e. element “Radha Sharma”.

Telephone Directory



Name	*Phone No.*
Nitin Kumar	25161234
Preeti Jain	22752345
Sandeep Singh	23405678
Sapna Chowdhary	22361111
Hitesh Somal	24782202
R.S.Singh	26254444
Radha Sharma	26150880
S.N.Singh	25513653
Arvind Chittora	26252794
Anil Rawat	26257149

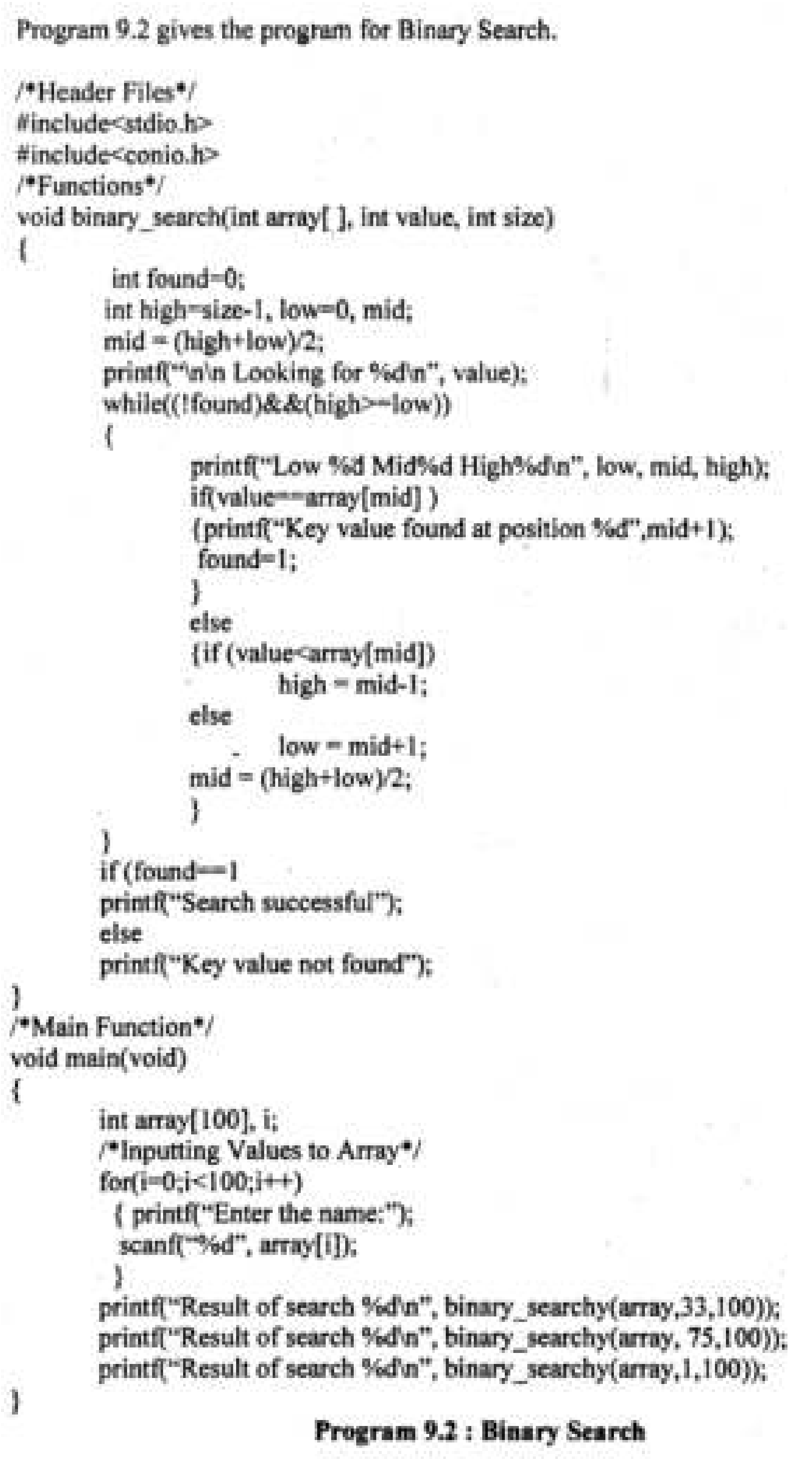
The above algorithm will search for element “Radha Sharma” and will stop at 6th index of array and the required phone number is “26150880”, which is stored at

position 7 i.e. 6+1.
Efficiency of Linear Search
How many number of comparisons are there in this search in searching for a given element?
The number of comparisons depends upon where the record with the argument key appears in the array. If record is at the first place, number of comparisons is ‘1’, if record is at last position ‘n’ comparisons are made.
If it is equally likely for that the record can appear at any position in the array, then, a successful search will take (n+ 1)/2 comparisons and an unsuccessful search will take ‘n’ comparisons.
In any case, the order of the above algorithm is 0(n).
V

9.3 BINARY SEARCH

An unsorted array is searched by linear search that scans the array elements one by one until the desired element is found.
The reason for sorting an array is that we search the array “quickly”. Now, if the array is sorted, we can employ binary search, which brilliantly halves the size of the search space each time it examines one array element.
An array-based binary search selects the middle element in the array and compares its value to that of the key value. Because, the array is sorted, if the key value is less than the middle value then the key must be in the first half of the array. Likewise, if the value of the key item is greater than that of the middle value in the array, then it is known that the key lies in the second half of the array. In either case, we can, in effect, “throw out” one half of the search space or array with only one comparison.
Now, knowing that the key must be in one half of the array or the other, the binary search examines the mid value of the half in which the key must reside. The algorithm thus narrows the search area by half at each step until it has either found the key data or the search fails.
As the name suggests, binary means two, so it divides an array into two halves for searching. This search is applicable only to an ordered table (in either ascending or in descending order).
Let us write an algorithm for Binary Search and then we will discuss it. The array consists of elements stored in ascending order.
Algorithm

```
Step 1: Declare an array 'k' of size 'n' i.e. k(n) is an array which stores all the keys of a file containing 'n'
records
Step 2: i<-O
Step 3: low<-O, high<-n-l
Step 4: while (low < high)do
mid = (low + high)/2
if (key=k[mid]) then
write "record is at position", mid+I I/as the array
starts from the 0th position
else
if(key < k[mid]) then
high = mid - 1
else
low = mid + I
endif
endif
endwhile
Step 5: Write "Sorry, key value not found"
Step 6: Stop
```



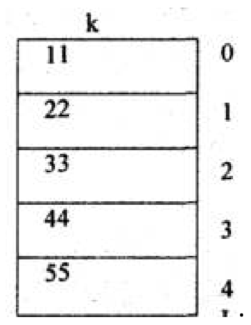
Program 9.2 gives the program for Binary Search.

```
/*Header Files*/
#include<stdio.h>
#include<conio.h>
/*Functions*/
void binary_search(int array[ ], int value, int size)
{
            int found=0;
           int high=size-1, low=0, mid;
           mid = (high+low)/2;
           printf("\n\n Looking for %d\n", value);
           while((!found)&&(high>=low))
           {
                       printf("Low %d Mid%d High%d\n", low, mid, high);
                       if(value==array[mid] )
                       {printf("Key value found at position %d",mid+1);
                        found=1;
                       }
                       else
                       {if (value<array[mid])
                                   high = mid-1;
                       else
                                   low = mid+1;
                       mid = (high+low)/2;
                       }
           }
           if (found==1
           printf("Search successful");
           else
           printf("Key value not found");
}
/*Main Function*/
void main(void)
{
           int array[100], i;
           /*Inputting Values to Array*/
           for(i=0;i<100;i++)
             { printf("Enter the name:");
              scanf("%d", array[i]);
             }
           printf("Result of search %d\n", binary_searchy(array,33,100));
           printf("Result of search %d\n", binary_searchy(array, 75,100));
           printf("Result of search %d\n", binary_searchy(array,1,100));
}
```

Program 9.2 : Binary Search

Example: Let us consider a file of 5 records, i.e., n 5

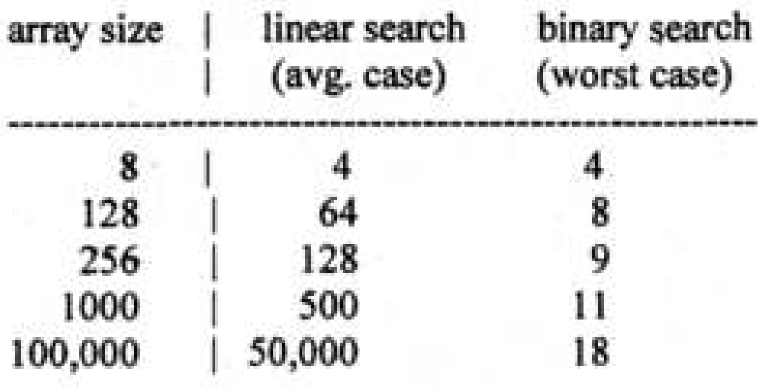
And k is a sorted array of the keys of those 5 records



Let key 55, low 0, high 4
Iteration I: mid (0+4)/2 2
k(mid) = k (2) = 33
Now key> k (mid)
Solow’mid+13
Iteration 2: low = 3, high =4 (low <= high)
Mid = 3+4 /2 3.5 3 (integer value)
Here key> k (mid)
Solow3+l 4
Iteration 3: low = 4, high 4 (low< high) V
Mid = (4+4)/2 =4
Here key = k(mid)
So, the record is at mid+l position, i.e., 5
Efficiency of Binary Search -

Each comparison in the binary search reduces the number of possible candidates where the key value can be found by a factor of 2 as the array is divided in two halves in each iteration. Thus, the maximum number of key comparisons is approximately log n. So, the order of binary search is 0 (log n).
Comparative Study of Linear and Binary Search
Binary search is lots faster than linear search. Here are some comparisons:

NUMBER OF ARRAY ELEMENTS EXAMINED



array size	linear search (avg. case)	binary search (worst case)
8	4	4
128	64	8
256	128	9
1000	500	11
100,000	50,000	18

A binary search on an array is 0 (tog2 n) because at each test, you can "throw out" one half of the search space or array whereas a linear search on an array is 0(n).
It is noteworthy that, for very small arrays a linear search can prove faster than a binary search. However, as the size of the array to be searched increases, the binary search is the clear winner in terms of number of comparisons and therefore overall speed.
Still, the binary search has some drawbacks. First, it requires that the data to be searched be in sorted order. If there is even one element out of order in the data being searched, it can throw off the entire process. When presented with a set of unsorted data, the efficient programmer must decide whether to sort the data and apply a binary search or simply apply the less-efficient linear search. Is the cost of sorting the data is worth the increase in search speed gained with the binary search? If you are searching only once, then it is probably to better do a linear search in most cases.

9.4 APPLICATIONS

The searching techniques are applicable to a number of places in today's world, may it be Internet, search engines, on line enquiry, text pattern matching, finding a record from database, etc.
The most important application of searching is to track a particular record from a large file, efficiently and faster.
Let us discuss some of the applications of Searching in the world of computers.
1. Spell Checker
This application is generally used in Word Processors. It is based on a program for checking spelling, which it checks and searches sequentially. That is, it uses the concept of Linear Search. The program looks up a word in a list of words from a dictionary. Any word that is found in the list is assumed to be spelled correctly. Any word that isn't found is assumed to be spelled wrong.

2. Search Engines
Search engines use software robots to survey the Web and build their databases. Web documents are retrieved and indexed using keywords. When you enter a query at a search engine website, your input is checked against the search engine's keyword indices. The best matches are then returned to you as hits. For checking, it uses any of the Search algorithms.
Search Engines use software programs known as robots, spiders or crawlers. A robot is a piece of software that automatically follows hyperlinks from one document to the next around the Web. When a robot discovers a new site, it sends information back to its main site to be indexed. Because Web documents are one of the least static forms of publishing (i.e., they change a lot), robots also update previously catalogued sites. How quickly and comprehensively they carry out these tasks vary from one search engine to the next.
3. String Pattern matching

Document processing is rapidly becoming one of the dominant functions of computers. Computers are used to edit, search and transport documents over the Internet, and to display documents on printers and computer screens. Web 'surfing' and Web searching are becoming significant and important computer applications, and many of the key computations in all of this document processing involves character strings and string pattern matching. For example, the Internet document formats HTML and XML are primarily text formats, with added tags for multimedia content. Making sense of the many terabytes of information on the Internet requires a considerable amount of text processing. This is accomplished using trie data structure, which is a tree-based structure that allows for faster searching in a collection of strings.

9.5 SUMMARY

Searching is the process of looking for something. Searching a list consisting of 100000 elements is not the same as searching a list consisting of 10 elements. We discussed two searching techniques in this unit namely Linear Search and Binary Search. Linear Search will directly search for the key value in the given list. Binary search will directly search for the key value in the given sorted list. So, the major difference is the way the given list is presented. Binary search is efficient in most of the cases. Though, it had the overhead that the list should be sorted before search can start, it is very well compensated through the time (which is very less when compared to linear search) it takes to search. There are a large number of applications of Searching out of whom a few were discussed in this unit.

UNIT 10 Sorting

10.0 INTRODUCTION

Retrieval of information is made easier when it is stored in some predefined order. Sorting is, therefore, a very important computer application activity. Many sorting algorithms are available. Different environments require different sorting methods. Sorting algorithms can be characterized in the following two ways:

1. Simple algorithms, which require the order of n7 (written as O (n2)) cornparisons to sort n items.

2. Sophisticated algorithms that require the O(nlog2n) comparisons to sort n items.

The difference lies in the fact that the first method moves data only over small distances in the process of sorting, whereas the second method moves data over large distances, so that items settle into the proper order sooner, thus resulting in fewer comparisons. Performance of a sorting algorithm can also depend on the degree of order already present in the data

There are two basic categories of sorting methods: Internal Sorting and External Sorting. Internal sorting is applied when the entire collection of data to be sorted is small enough so that the sorting can take place within the main memory. The time required to read or write is not considered to be significant in evaluating the performance of internal sorting methods. External sorting methods are applied to larger collection of data which reside on secondary devices. Read and write access times are a major concern in determining sorting performances of such methods.
In this unit, we will study some methods of internal sorting. The next unit will discuss methods of external sorting.

10.2.1 Insertion Sort

This is a naturally occurring sorting method exemplified by a card player arranging the cards dealt to him. He picks up the cards as they are dealt and inserts them into the required position. Thus at every step, we insert an item into its proper place in an already ordered list.
We will illustrate insertion sort with an example (refer to Figure 10.1) before presenting the formal

algorithm.

Example: Sort the following list using the insertion sort method:

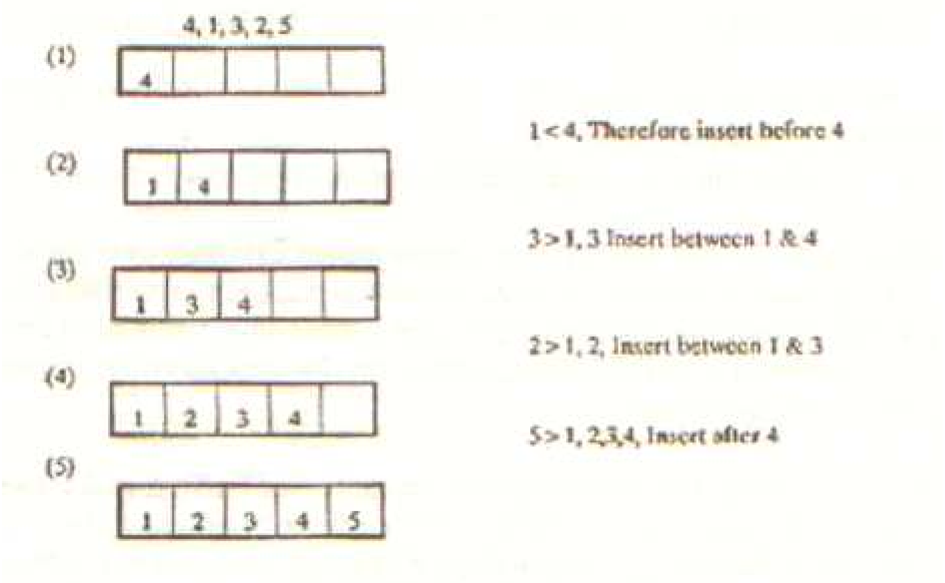

4, 1, 3, 2, 5
(1)
(2)
(3)
(4)
(5)
1 < 4, Therefore insert before 4
3 > 1, 3 Insert between 1 & 4
2 > 1, 2, Insert between 1 & 3
5 > 1, 2,3,4, Insert after 4

Figure 10.1: Insertion sort

Thus to find the correct position search the list till an item just greater than the target is found. Shift all the items from this point one down the list. Insert the target in the vacated slot. Repeat this process for all the elements in the list. This results in sorted list.

10.2.2 Bubble Sort

In this sorting algorithm, multiple swappings take place in one pass. Smaller elements move or 'bubble' up to the top of the list, hence the name given to the algorithm.

In this method, adjacent members of the list to be sorted are compared.If the item on top is greater than the item immediately below it, then they are swapped. This process is carried on till the list is sorted.

The detailed algorithm follows:

Algorithm: BUBBLE SORT

```
1. Begin
2. Read the n elements
3. for i=1 to n
   for j=n downto i+1
       if a[j] <=a[j-1]
4. swap (a[j],a[j-1])
```

5. 4.End //of Bubble Sort

6.

7. Total number of comparisions in bubble sort

8. =(N-1)+(N-2)…+2+1

9. =(N-1)*N / 2 =O(N2)

10.

11. 10.1 gives the program segment for Quick sort. It uses recursion.

```
 Quick sort (A,m,n)
int A[],m,n
{
int i,j, k;
if m<n
{
```

12.

```
i=m
j=n+l;
k=A[m];
do
do
while (A[i] <k);
do
```

13.

```
--j;
              while (A[j]> k);
if(i<j)
{
temp = A[i];
A[i] A[j];
A[j]temp;
}
while (i<j);
temp = A[m];
 A[j]= temp;
Quicksort(A,m,j- 1);
Quicksort(A,j+ I ,n);
}
```

14.

Program 10.1: Quick Sort

The Quick sort algorithm uses the O(N Log2N) comparisons on average. The performance can be improved by keeping in mind the following points.

I. Switch to a faster sorting scheme like insertion sort when the sublist size becomes comparatively small.

2. Use a better dividing element in the implementations.
It is also possible to write the non-recursive Quick sort algorithm.

10.2.4 2-Way Merge Sort

Merge sort is also one of the 'divide and conquer' class of algorithms. The basic idea in this is to divide the list into a number of sub lists, sort each of these sub lists and merge them to get a single sorted list. The illustrative implementation of 2 way merge sort sees the input initially as n lists of size 1. These are merged to get n/2 lists of size 2. These n/2 lists are merged pair wise and so on till a single list is obtained. This can be better understood by the following example. This is also called concatenate sort. Figure 10.2 depicts 2-way merge sort.
Mergesort is the best method for sorting linked lists in random order. The total computing time is of the 0(n log2n).

5/2 = 2 i.e. the parent is R. Its children are at positions 2 x 5 & (2 x 5) + I, i.e.
$0 + 11 respectively i.e. E & I are its children.

A Heap is a complete binary tree, in which each node satisfies the

heap condition. represented as an array.

We will now study the operations possible on a heap and see how these can be combined to generate a sorting algorithm.

The operations on a heap work in 2 steps.

I. The required node is inserted/deleted/or replaced.

2. The above operation may cause violation of the heap condition so the heap is traversed and modified to rectify any such violations.
Example: Consider the insertion of a node R in the heap 1.
1. Initially R is added as the right child of J and given the number 13.
2. But, R > I. So, the heap condition is violated.
3. Move R up to position 6 and move i down to position 13.
4. R> P. Therefore, the heap condition is still violated.
5.SwapRandP.
4. The heap condition is now satisfied by all nodes to get the heap of Figure 10.5.

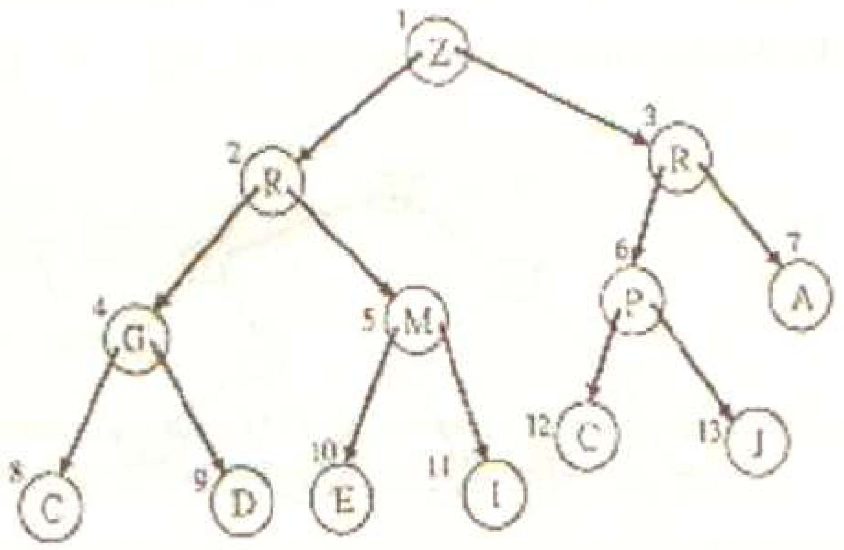


Figure 10.5: A Heap

This algorithm is guaranteed to sort n elements in (n log2n) time.

We will first see two methods of heap construction and then removal in order from the heap to sort the list.

I. Top down heap construction

• Insert items into an initially empty heap. Satisfying the heap condition at all steps.

2. Bottom up heap construction

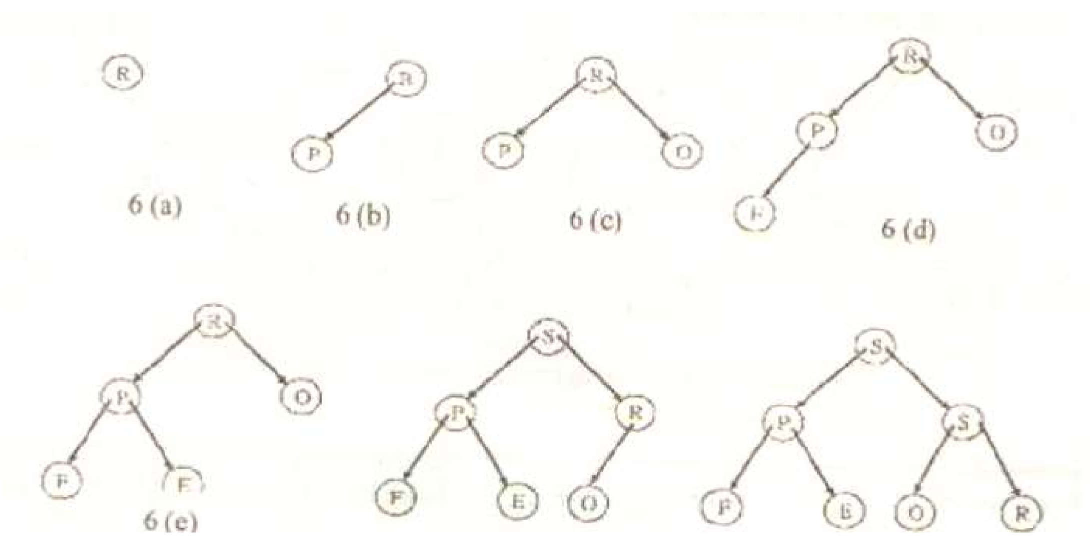
• Build a heap with the items in the order presented.

• From the right most node modify to satisfy the heap condition. We will exemplify this with an example.

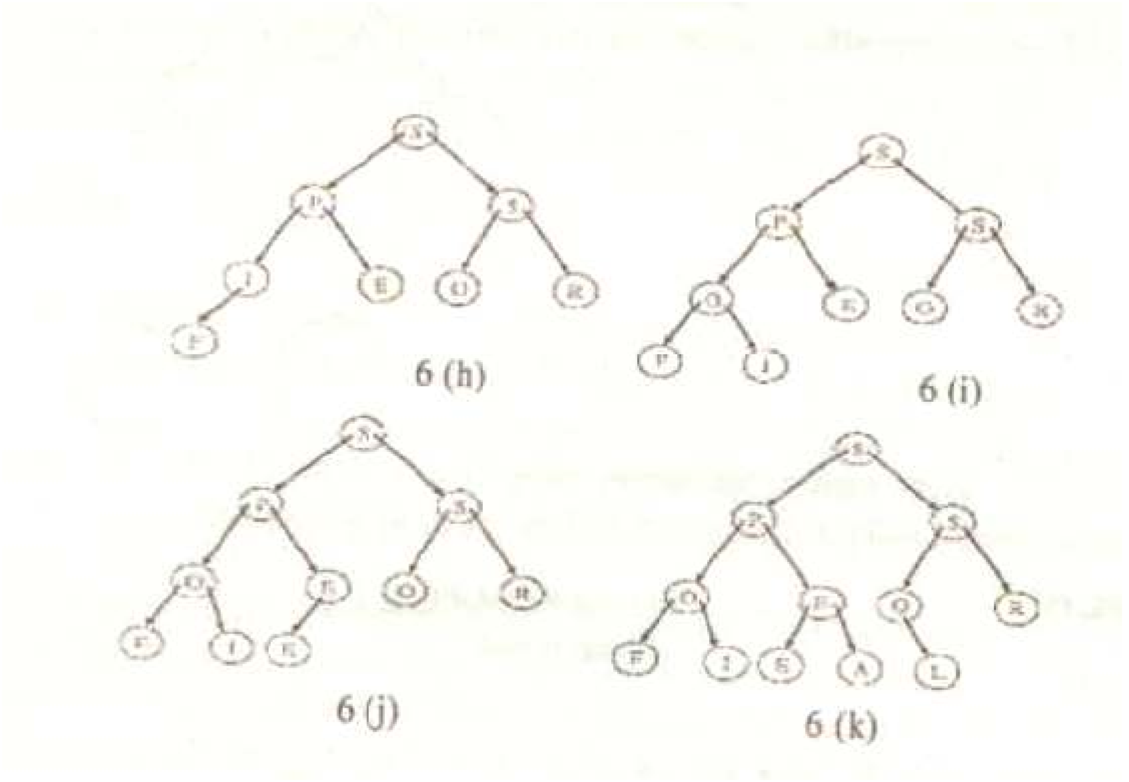
Example: Build a heap of the following using top down approach for heap construction.

PROFESSIONAL

Figure 10.6 shows different steps of the top down construction of the heap.



15. Example: The input file is (2,3,81.64.4,25,36,16,9, 49). When the file is interpreted as a binary tree, it results in Figure 10.7. Figure 10.8 depict the heap.



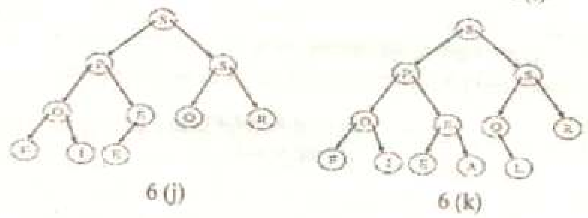

6 (j) 6 (k)

Figure 10.6: Heap Sort (Top down Construction)

Example: The input file is (2,3,81,64,4,25,36,16,9, 49). When the file is interpreted as a binary tree, it results in *Figure 10.7*. *Figure 10.8* depicts the heap.

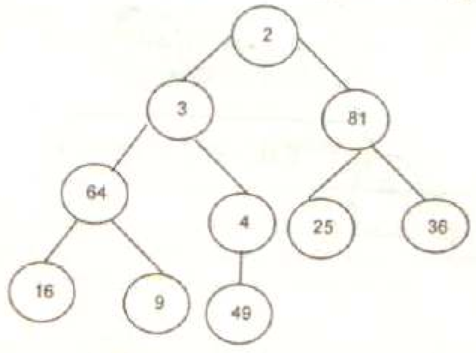


Figure 10.7: A Binary tree

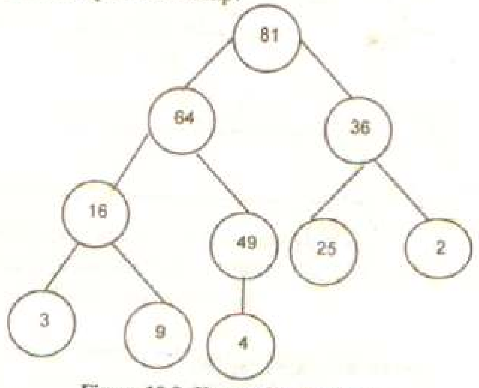
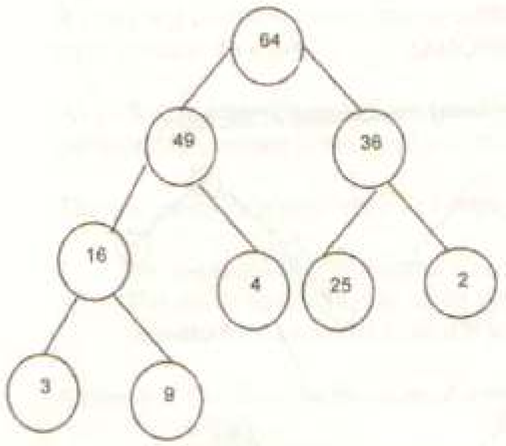
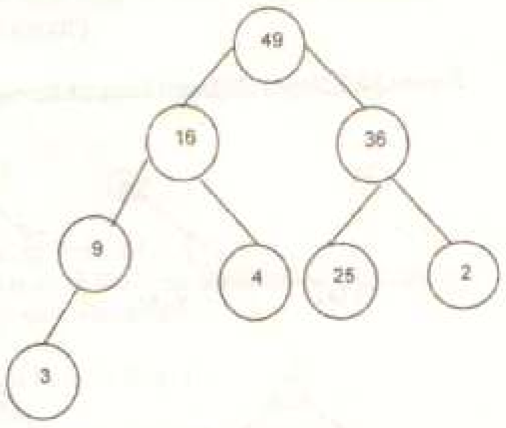


Figure 10.8: Heap of figure 10.7

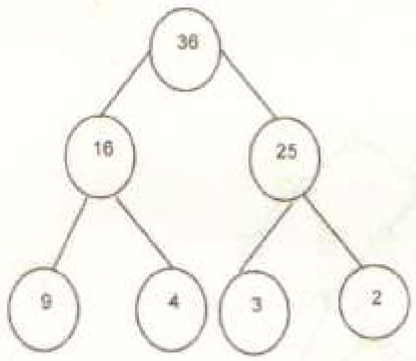
Figure 10.7: A Binary nec Figure **10.8: Heap of** fIgure **10.7**



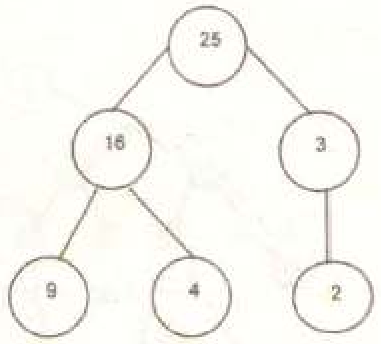
Sorted: 81
Heap size: 9



Sorted: 81,64
Heap size: 8



Sorted: 81,64,49
Heap size:7



Sorted:81,64,49,36
Heap size:6

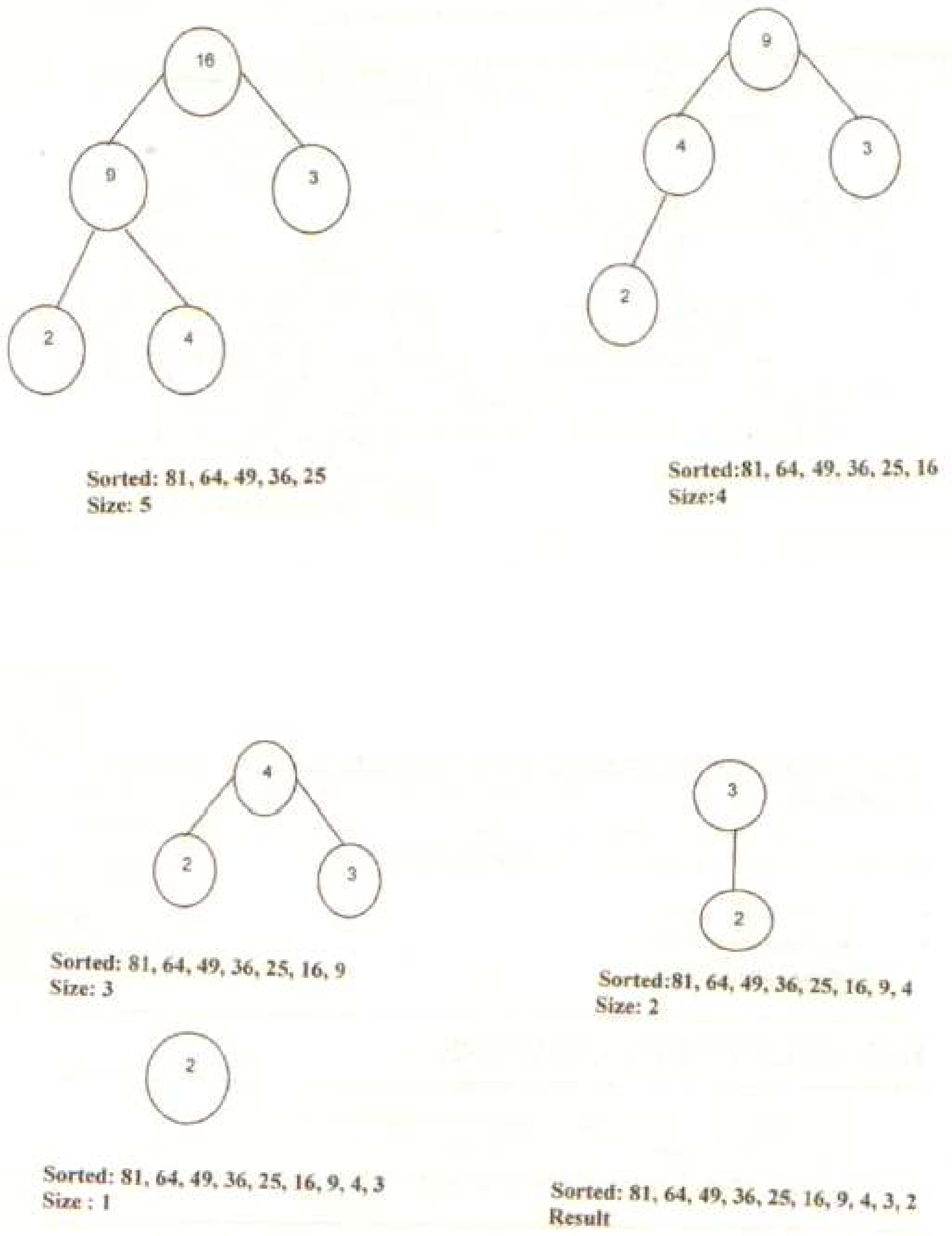

16
9
3
2
4
Sorted: 81, 64, 49, 36, 25
Size: 5
9
4
3
2
Sorted:81, 64, 49, 36, 25, 16
Size:4
4
2
3
Sorted: 81, 64, 49, 36, 25, 16, 9
Size: 3
3
2
Sorted:81, 64, 49, 36, 25, 16, 9, 4
Size: 2
2
Sorted: 81, 64, 49, 36, 25, 16, 9, 4, 3
Size : 1
Sorted: 81, 64, 49, 36, 25, 16, 9, 4, 3, 2
Result

Figure 10.9 : Various steps of figure 10.8 for a sorted file

10.3 SORTING ON SEVERAL KEYS

So far, we have been considering sorting based on single keys. But, in real life applications, we may want to sort the data on several keys. The simplest example is that of sorting a deck of cards. The first key for sorting is the suit-clubs, spades, diamonds and hearts. Then, within each suit, sorting the cards in ascending order from Ace, twos to king. This is thus a case of sorting on 2 keys.

Now, this can be done in 2 ways.

I • Sort the 52 cards into 4 piles according to the suit.

• Sort each of the 4 piles according to face value of the cards.

2 • Sort the 52 cards into 13 piles according to face value.

• Stack these piles in order and then sort into 4 piles based on suit.

The first method is called the MS1) (Most Significant Digit) sort and the second method is called the LSD (Least Significant Digit) sort. Digit stands for a key. Though they are called sorting methods, MSD and LSD sorts only decide the order of sorting. The actual sorting could be done by any of the sorting methods discussed in this unit.

Graphs

8.0 INTRODUCTION

In this unit, we will discuss a data structure called Graph. In fact, graph is a general tree with no parent-child relationship. Graphs have many applications in computer science and other fields of science. In general, graphs represent a relatively less restrictive relationship between the data items. We shall discuss about both undirected graphs and directed graphs. The unit also includes information on different algorithms which are based on graphs.

8.2 DEFINITIONS

A graph G may be defined as a finite set V of vertices and a set E of edges (pair of connected vertices). The notation used is as follows:
Graph G = (V, E)
Consider the graph of Figure 8.1.
The set of vertices for the graph is V — {1, 2, 3, 4, 5}. Graphs
The set of edges for the graph is 13 = {(1,2), (1,5), (1,3), (5,4), (4,3), (2,3) }.
The elements of 13 are always a pair of elements.

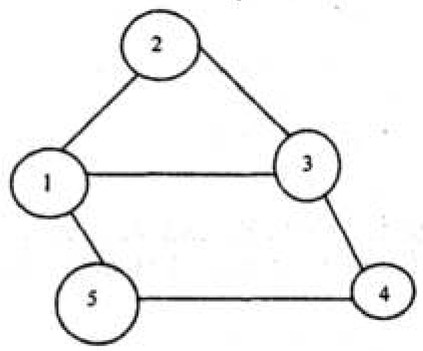


Figure 8.1: A graph

It may be noted that unlike nodes of a tree, graph has a very limited relationship between the nodes (vertices). There is no direct relationship between the vertices I and 4 although they are connected through 3.
Directed graph and Undirected graph: If every edge (a,b) in a graph is marked by a direction from a to b, then we call it a Directed graph (digraph). On the other hand, if directions are not marked on the edges, then the graph is called an Undirected graph. In a Directed graph, the edges (1,5) and (5,1) represent two different edges whereas in an Undirected graph, (1,5) and (5,1) represent the same edge. Graphs are used in

various types of modeling. For example, graphs can be used to represent connecting
roads between cities.
Graph terminologies:
Adjacent vertices: Two vertices a and b are said to be adjacent if there is an edge
connecting a and b. For example, in Figure 8.1, vertices 5 and 4 are adjacent.
Path: A path is defined as a sequence of distinct vertices, in which each vertex is
adjacent to the next. For example, the path from 1 to 4 can be defined as a sequence of
adjacent vertices (1,5), (5,4).
.path, p, of length, k, through a graph is a sequence of connected vertices:
p =
Cycle : A graph contains cycles if there is a path of non-zero length through the graph,
p <VO,V1,...,Vk> such that V0 = Vk.
Edge weight: It is the cost associated with edge.
Loop: It is an edge of the form (v,v).
Path length : It is the number of edges on the path.
Simple path : It is the set of all distinct vertices on a path (except possibly first and
last).

Spanning Trees: Aspanningfreefagraph,G, isasetofV1-l edges thaI connect all vertices ôfthe graph.
There are different representations of a graph. They are:
• Adjacency list representation
• Adjacency matrix representation.
Adjacency list representation
An Adjacency list representation of a Graph 0 = {V, E} consists of an array of adjacency lists denoted by
ad] of V list. For each vertex ucV, adj[u] consists of all vertices adjacent to u in the graph G.
Consider the graph of Figure 8.2.

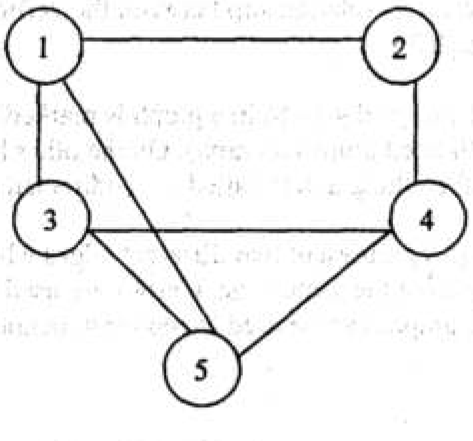


Figure 8.2: A Graph

The following is the adjacency list representation of graph of Figure 8.2:

adj [1] = {2, 3, 5}
adj [2] = {1, 4}
adj [3] = {1, 4, 5}
adj [4] = {2, 3, 5}
adj [5] = {1, 3, 4}

An adjacency matrix representation of a Graph G=(V, E) is a matrix $A(a_{ij})$ such that

$$a_{ij} = \begin{cases} 1 \text{ if edge (i, j) belongs to E} \\ 0 \text{ otherwise} \end{cases}$$

The adjacency matrix for the graph of *Figure 8.2* is given below:

$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ 1 & 0 & 1 & 1 & 0 & 1 \\ 2 & 1 & 0 & 0 & 1 & 1 \\ 3 & 1 & 0 & 0 & 1 & 1 \\ 4 & 0 & 1 & 1 & 0 & 1 \\ 5 & 1 & 0 & 1 & 1 & 0 \end{array}$$

Observe that the matrix is sym etricaongthemaindiagonal. If we definethe adjacency matrix as A and th transpose as AT,then foran undirectedgraph"i as above, A = AT.
Graph connectivity:
A connected graph is a graph in which path exists between every pair of vertices.
A strongly connected graph is a directed graph in which every pair of distinct vertices are connected with each other.
A weakly connected graph is a directed graph whose underlying graph is connected, but not strongly connected.
A complete graph is a graph in which there exists edge between every pair of vertices.

8.3 SHORTEST PATH ALGORITHMS

A driver takes shortest possible route to reach destination. The problem that we will discuss here is similar to this kind of finding shortest route in a graph. The graphs are weighted directed graphs. The weight could be time, cost, losses other than distance designated by numerical values.
Single source shortest path problem : To find a shortest path from a single source to every vertex of the Graph.

Consider a graph G = (V,.E). We wish to.find out the shortest path from a single source vertex scV, to every vertex vcV. The single source shortest path algorithm (Dijkstra's Algorithm) is based on assumption that no edges have negative weights.
The procedure followed to find shortest path are based on a concept called relaxation. This method repeatedly decreases the upper bound of actual shortest path of each vertex from the source till it equals the shortest-path weight. Please note that shortest path between two vertices contains other shortest path within it.

8.3.1 Dijkstra's Algorithm

Djikstra's algorithm (named after its discover, Dutch computer scientist E.W. Dijkstra) solves the problem of finding the shortest path from a point in a graph (the source) to a destination with non-negative weight edge.
It turns out that one can find the shortest paths from a given source to all vertices (points) in a graph in the same time. Hence, this problem is sometimes called the single-source shortest paths problem.
Dijkstra's algorithm is a greedy algorithm, which finds shortest path between all pairs of vertices in the graph. Before describing the algorithms formally, let us study the method through au example.

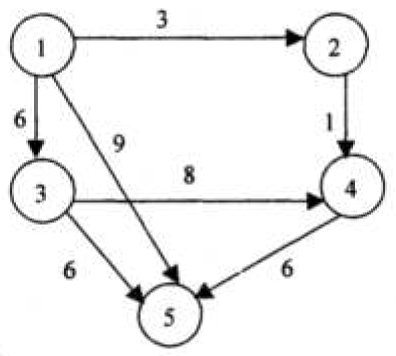


Figure 8.5: A Directed Graph with no negative edge(s)

Dijkstra's algorithm keeps two sets of vertices:

S is the set of vertices whose shortest paths from the source have already been determined

Q = V-S is the set of remaining vertices.

The other data structures needed are:

d array of best estimates of shortest path to each vertex from the source
p1 an array of predecessors for each vertex, predecessor is an array of vertices to which shortest path has already been determined.

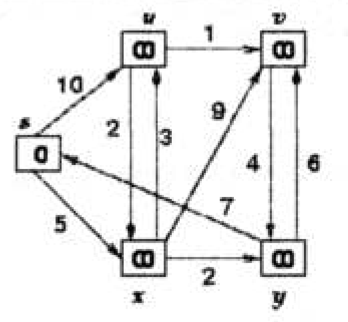
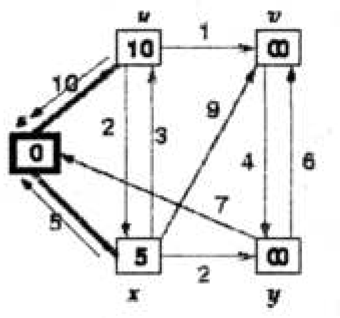
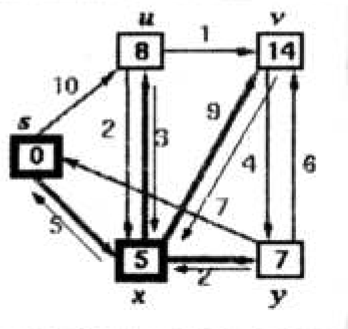
The basic operation of Dijkstra's algorithm is edge relaxation. If there is an edge from u to v, then the shortest known path from s to u can be extended to a path from s to v by adding edge (u,v) at the end.

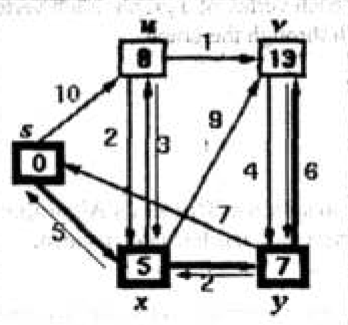
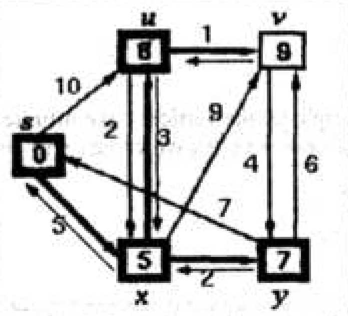
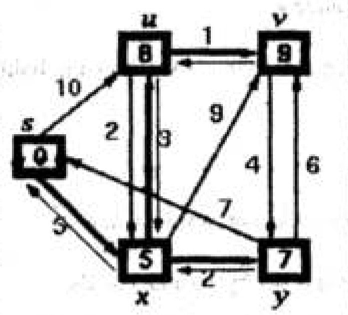
This path will have length d[u]+w(u,v). If this is less than d[vj, we can replace the current value of d[v] with the new value.

The predecessor list is an array of indices, one for each vertex of a graph. Each vertex entry contains the index of its predecessor in a path through the graph.

Operation of Algorithm

The following sequence of diagrams illustrates the operation of Dijkstra's Algorithm. The bold vertices indicate the vertex to which shortest path has be determined

	Initialize the graph, all the vertices have infinite costs except the source vertex which has zero cost
	From all the adjacent vertices, choose the closest vertex to the source s. As we initialized d[s] to 0, it's s. (shown in bold circle) Add it to S Relax all vertices adjacent to s, i.e u and x Update vertices u and x by 10 and 5 as the distance from s.
	Choose the nearest vertex, x. Relax all vertices adjacent to x Update predecessors for u, v and y. Predecessor of x = s Predecessor of v = x ,s Predecessor of y = x ,s add x to S

 u v 8 13 1 10 s 0 2 3 9 4 6 7 5 5 7 2 x y	Now y is the closest vertex. Add it to S. Relax v and adjust its predecessor.
 u v 8 9 1 10 s 0 2 3 9 4 6 7 5 5 7 2 x y	u is now closest, add it to S and adjust its adjacent vertex, v.
 u v 8 9 1 10 s 0 2 3 9 4 6 7 5 5 7 2 x y	Finally, add v to S. The predecessor list now defines the shortest path from each node to s.

```
Dijkstra's algorithm
* Initialize d and pi *
for each vertex v in V( g)
g.d[v] := infinity
g.pi[v] := nil
g.d[s] :=0;
* S to empty *
S:={0}
Q:V(g)
* While (V.S) is not null
while not Empty(Q)
I. Sort the vertices in V-S according to the current best estimate of their distance frdm the source
u : Extract-Mm (Q);
2. Add vertex u, the closest vertex in V-S, to S. Add Node (S, u);
```

```
3. Relax allthe vertices still .ifl V-S comiected to u
relax( Node U, Node v, double w[]fl)
if d[v} > dfuj + w[u]v} then
d[vJ := d[u] + w[u][v]
pifv]:=u
```

In summary, this algorithm starts by assigning a weight of Infinity to all vertices, and then' selecting a source and assigning a weight of zero to it. Vertices are a4ded to the set for which shortest paths are known.. When. a vertex is selected, the weights of its adjacent vertices' are relaxed. Once all vertices are relaxed, their predecessor's vertices are updated (p1). The cycle of selection, weight relaxation and predecessor update is repeated until the shortest path to all vertices has been found.

Complexity of Algorithm

The simplest implementation of the Dijkstra's algorithm stores vertices of set Q in an ordinary linked list or array, and operation Extract-Min(Q) is simply a linear search through all vertices iii Q. In this case, the running time is ((nZ).

8.3.2 Graphs with Negative edge costs

We have seen that the above Dijkstra's single source shortest-path algorithm works for graphs with non-negative edges (like road networks). The following two scenarios can emerge out of negative cost edges iii a graph:

- Negative edge with non- negative weight cycle reachable from the source.
- Negative edge with non-negative weight cycle reachable from source.

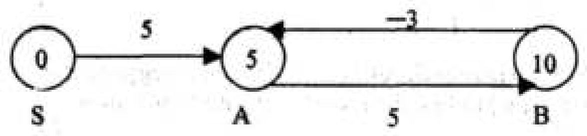


Figure 8.6 : A Graph with negative edge and non-negative weight cycle

The net weight of the cycle is 2(non-negative)(refer to *Figure 8.6*).

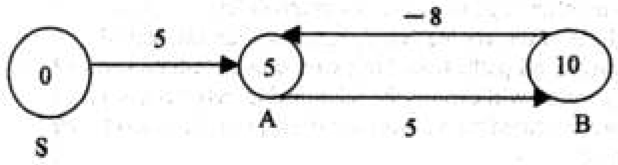


Figure 8.7: A graph with negative **edge and negative weight cycle**

The net weight of the cycle is —3(negative) (refer to Figure 8.7). The shortest path from A to B is not well defined as the shortest path to this vertex are infinite, i.e., by
traveling each cycle we can decrease the cost of the shortest path by 3, like (S, A, B) is path (S, A, B, A, B) is a path with less cost and so on.
Dijkstra’s Algorithm works only for directed graphs with non-negative weights (cost).

<u>8.3 Acyclic Graphs</u>

A path in a directed graph is said to form a cycle is there exists a path (A,B,C P) such that A = P. A graph is called Acyclic if there is no cycle in the graph.
8.3.4 All Pairs Shortest Paths Algorithm
In the last section, we discussed about shortest path algorithm, which starts with a single source and finds shortest path to all vertices in the graph. In this section, we shall discuss the problem of finding shortest path between all pairs of vertices in a graph. This problem is helpful in finding distance between all pairs of cities in a road atlas. All pairs shortest paths problem is mother of all shortest paths problems.
In this algorithm, we will represent the graph by adjacency matrix.
The weight of an edge C in an adjacency matrix representation of a directed graph is represented as follows
. Weight of the directed edge from i to j i.e. (i,j) if I j and (ij) belongs to
= co if i j and(i, j)does not belong to E
Given a directed graph G = (V. E), where each edge (v, w) has a non-negative cost C(v , w), for all pairs of vertices (v, w) to find the lowest cost path from v to w.

The All pairs shortest paths problem can be considered as a generalization of single- source-shortest-path problem, by using Dijkstra’s algorithm by varying the source node among all the nodes in the graph. Jf negative edge(s) is allowed, then we can’t use Dijkstra’s algorithm.
In this section we shall use a recursive solution to all pair shortest paths problem known as Floyd-Warshall algorithm, which runs in 0(n3) time.
This algorithm is based on the following principle. For graph G let V = { 1, 2, 3,.. .,n}.Let us consider a sub set of the vertices (1, 2, 3 ,k. For any pair of vertices that belong to V. consider all paths from ito j whose intermediate vertices are from { 1, 2, 3 k}. This algorithm will exploit the relationship between path p and shortest path from ito j whose intermediate vertices are from { 1, 2, 3 k-i } with the following two possibilities:
I. If k is not an intermediate vertex in the path p, then all the intermediate vertices of the path pare in {1, 2, 3 ,k-l}. Thus, shortest path from i toj with intermediate vertices in (1, 2, 3 ,k-l } is also the shortest path from i toj with vertices in (1,2,3, ..., k}.
2. If k is an intermediate vertex of the path p, we break down the path p into path p1 from vertex ito k and path p2 from vertex k to j. So, path p1 is the shortest
path from ito k with intermediate vertices in (1, 2, 3, .. .,k-1}.
During iteration process we find the shortest path from ito j using only vertices (1, 2,
3, ..., k-i) and in the next step, we find the cost of using the kth vertex
intermediate step. If this results in lower cost, then we store it.
After n iterations (all possible iterations), we find the lowest cost path from ito j using all vertices (if necessary).
Note the following:
Initialize the matrix
C[i][j] = co if(i,j) does not belong toE for graph 0 = (V, E)
Initially, D[i][j] = Cf i}[j]
We also define a path matrix P wherePfij[jJ holds intermediate vertex k on the least cost path from ito j that Iead to the shortest path from ito j.
Algorithm (All Pairs Shortest Paths)

```
N = number of rows of the graph
D[i[j] = C[i][j]
Fork from Iton
Do for i Ito n
Do for j = ito n
D[i[jJ= minimum( , dIk + dk’) )
Enddo
Enddo
Enddo
```

where d°’’ minimum path from Ito j using k-i intermediate vertices where dIk = minimum path from ito k using k-i intermediate vertices where dkJ = minimum path from k to j using k-i intermediate vertices
Program 8.1 gives the program segment for the All pairs shortest paths algorithm.

```
AllPairsShortestPaths(int N, Matrix C, Matrix P, Matrix D)
{
```

```
inti,j,k
if i =j

then
for(i’’0; i<N;i++)
{
for (j= 0; j<N; j++)
{
D[iJ[j]=C[i]lj];
P[i][j]=1;
}
D[i][j] = 0;
}
for (k=0; k<N; k++)
{

Graph Algorithms and f(, (Q; i<N; i++)
Searching Techniques {
for(j=O;J<N;J±+)
{
if(D[i][lc]+ D[k]fj] <Pliltil)
D[i][j] = Di][kJ + D[k]fj];
P[i]fj] = k;
}
}
}
}
/*********** End ***********/
```

8.4 MINIMUM COST SPANNING TREES

A spanning tree of a graph is just a subgraph that contains all the vertices and is a tree (with no cycle) . A graph may have many spanning trees

. Figure 8.8: A Graph Fgure 8.9 : Spanning trees of the Graph of Figure 8.9

Consider the graph of Figre 8.8. It’s spanning trees are shown in FigutE 8.9.

Now, if the graph is a weighted graph (length associated with each edge). The weight of the tree is just the sum of weights of its edges. Obviously, different spmnng trees have diffefem weights or lengths. Our objective is to find the Check Your Progress 1

1) A graph with no cycle is called raph.
2) Adjacency matrix of an undirected graph is __________ on main diagonal.
3) Represent the following graphs(Figure 8.3 and Figure 8.4) by adjacency matrix: length (weight) spanning tree.

Suppose, we have a group of islands that we wish to link with bridges so that it is 3raphs
• possible to travel from one island to any other in the group. The set of bridges which will enable one to travel from any islaid to any other at minimum capital cost to..the government is the minimum cost spaiing tree.
8.4.1 Kruskal’s Algorithm
Krushkal’s algorithm uses the concept otforest of trees. InitiI4 the forest consists of n single node trees (and no edges). At each step, we add one (the cheapest one) edge so that it links two trees together. If it forms a cycle, it would simply mean that it liflks two nodes that were already connected. So, we reject it.
The steps in Kruskal’s Algorithm are as follows:
I. The forest is constructed from the graph G - with each node as a separate tree in the forest.
2. The edges ate placedin a priority queue.
3. Dountil we have added n-I edges to the graph,
I. Extiact the cheapest edgefiom the queue.
2. If it ibrms a cycle, then añk already exists between the concerned nodes. Hence reject it.
3. Else add it to the forest. Adding it to the forest will join two trees together.
The forest of trees is a partition of the original set of nodes; !ñitiallyall the trees have exactly one node in them. As the algorithm progresses, we form a union of two of e trees (sub-sets), until eventually the partitiàñ has only one sub-set contäirting all Ik nodes.
Let us seethe sequence of operatiohs to findIhe MinimurnCost Spanning Trée(MST) in agraçih using KruskaPs algorithm, consider the graphöf Figure 8.10., Figure 8.11 shows the construction of MST of graph of FIgure 8.10.

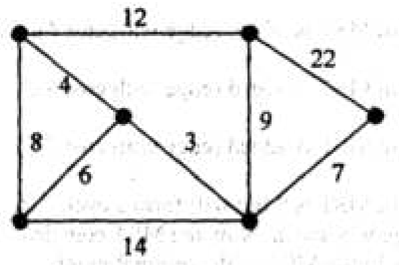


Figure 8.10 : A Graph

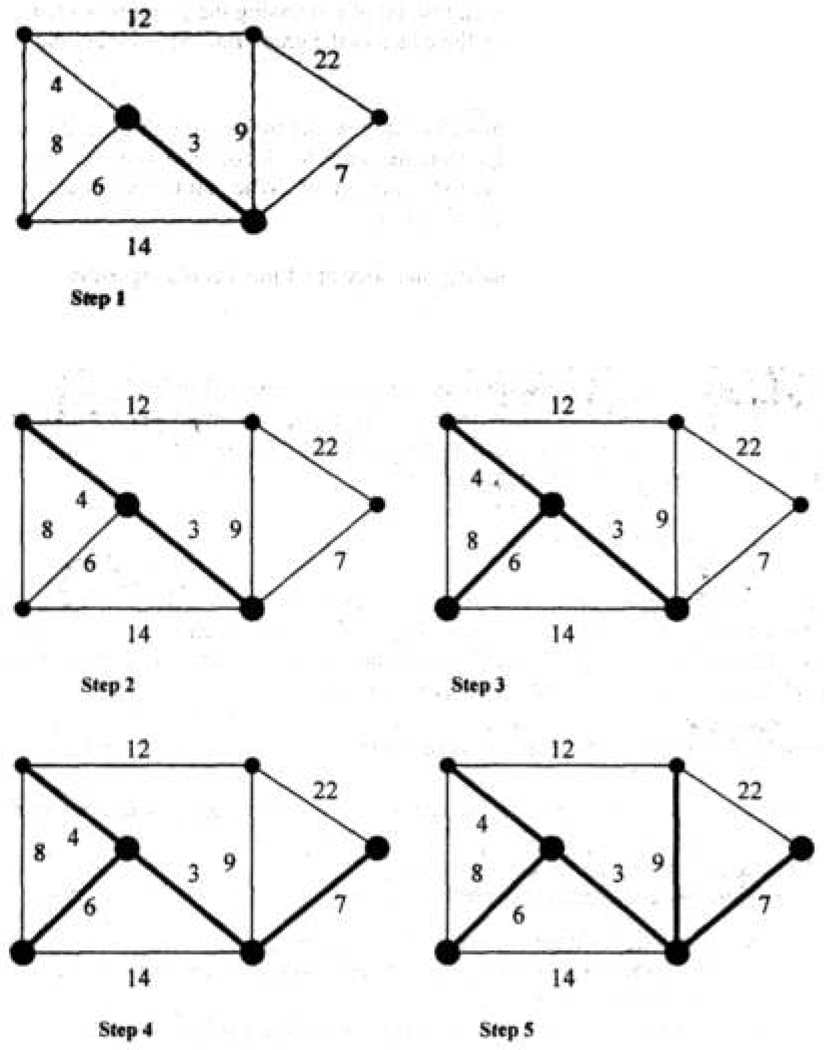


Figure 8.11 : Construction of Minimum Cost Spanning Tree for the Graph of Figure 8.10 by application of Kruskal's algorithm

The following are various steps in the construction of MST for the graph of Figure 8.10 using Kruskal's algorithm.

Step.1,: The lowest cost edge is selected from the graph which is not in MST
(initially MST is empty). The lowest cost edge is 3 which is added, t the MST (shown in bold edges)
Step 2: The next lowest cost edge which is not in MST is added (edge with cost 4). Step 3 : The next

lowest cost edge which is not in MST is added (edge with cost 6).
Step 4 The neXt lowest cost edge which is not in MST is added (edge with cost 7).
Step 5 : The next lowest cost edge which is not in MST is 8 but will form a cycle. So, it is discarded. The next lowest cost edge 9 is added. Now the MST contains all the vertices of the graph. This results in the MST of the original graph.

8.4.2 Prim's Algorithm
Prim's algorithm uses the concept of sets. Instead of processing the graph by sorted order of edges, this algorithm processes the edges in the graph randomly by building up disjoint sets.
It uses two disjoint sets A and A. Prim's algorithm works by iterating thftugh the nodes and then finding the shortest edge from the set A to that of set A (i.e. out side A), followed by the addition of the node to the new graph. When all the nodes are processed, we have a minimum cost spanning tree.
Rather than building a sub-graph by adding one edge at a time, Prim's algorithm
. builds a tree one vertex at a time. -

The steps in Prim's algorithm are as follows:

Let G be the graph with n vertices for which minimum cost spanning tree is to be generated.

Let T be the minimum spanning tree. Let T be a single vertex x.
while (T has fewer than n vertices)

{

find the smallest edge connecting T to G-T add it to T

}
Consider the graph of Figure 8.10. Figure 8.12 shows the various steps involved in
the construction of Minimum Cost Spanning Tree of graph of Figure 8.10.

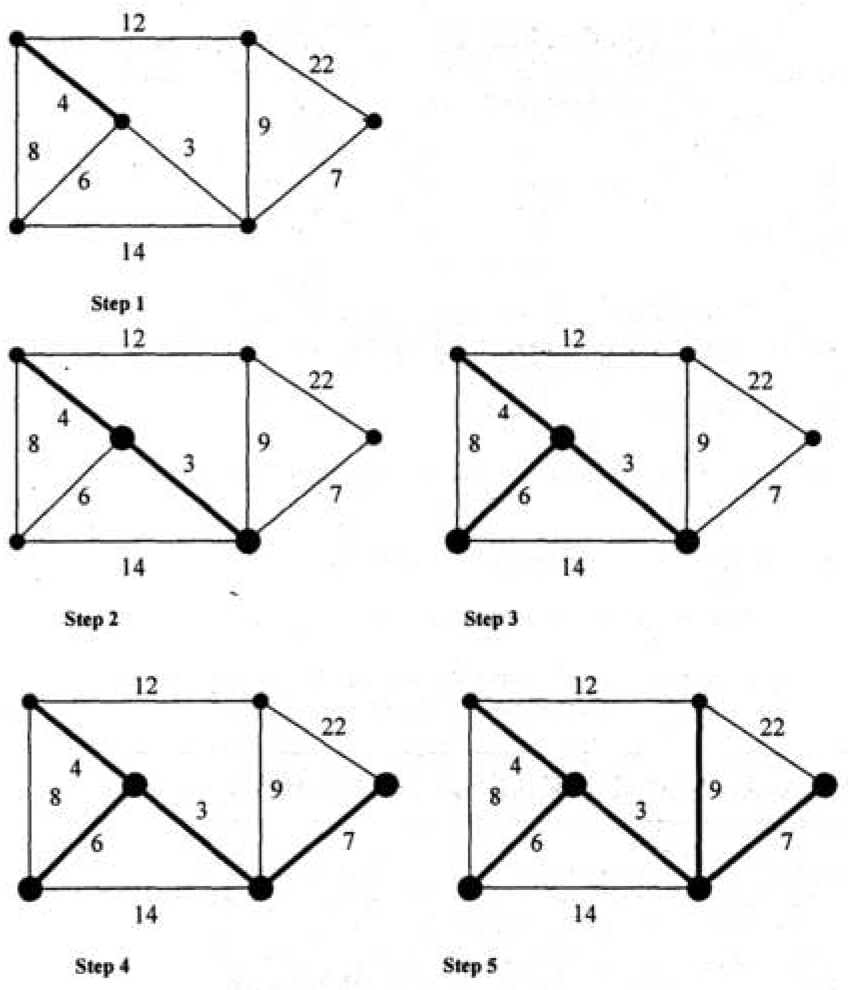


Figure 8.12 : Construction of Minimum Cost Spanning Tree for the Graph of Figure 8.10 by application of Prim's algorithm

The following are various steps in the construction of MST for the graph of Figure 8.10 using Prim's algorithm.

Step 1: We start with a single vertex (node). Now the set A contains this single node and set A contains rest of the nodes. Add the edge with the lowest cost from A to A. The edge with cost 4 is added.

Step 2: Lowest cost path from shaded portion of the graph to the rest of the graph (edge with cost 3) is selected and added to MST.

Step 3: Lowest cost path from shaded portion of the graph to the rest of the graph (edge with cost 6) is selected and added to MST.

Step 4: Lowest cost path from shaded portion of the graph to the rest of the graph (edge with cost 73) is selected and added to MST.

Step 5: The next lowest cost edge to the set not in MST is 8 but forms a cycle. So, it is discarded. The next lowest cost edge 9 is added. Now the MST contains all the vertices of the graph. This results in the MST of the original graph.

Comparison of Kruskal's algorithm and Prim's algorithm

	Kruskal's algorithm	Prim's algorithm
Principle •	Based on generic minimum cost spanning tree algorithms	A special case of generic minimum cost spanning tree algorithm. Operates like Dijkstra's algorithm for finding shortest path in a graph.
Operation	Operates on a single set of edges in the graph	Operates on two disjoint sets of edges in the graph
Running time	O(E log E) where E is the number of edges in the graph	O(E log V), which is asymptotically same as Kruskal's algorithm

For the above comparison, it may be observed that for dense graphs having more number of edges for a given number of vertices, Prim's algorithm is more efficient.

8.4.3 Applications

The minimum cost spanning tree has wide applications in different fields. It represents many complicated real world problems like:

1. Minimum distance for traveling all cities at most one (traveling salesman problem)

in electronic circuit design, to connect n pins by using n-l wires, using least wire.
3. Spanning tree also finds their application in obtaining independent set of circuit equations for an electrical network.

8.5 BREADTH FIRST SEARCH (BFS)

When BPS is applied, the vertices of the graph are divided into two categories. The vertices, which are visited as part of the search and those vertices, which are not visited as part of the search. The strategy adopted in breadth first search is to start search at a vertex(source). Once you started at source, the number of vertices that are yisited as part of the search is I and all the remaining vertices need to be visited. Then, search the vertices which are adjacent to the visited vertex from left to order. In this way, all the vertices of the graph are searched.
consider the digraph of Figure & 13. Suppose that the search started from S. Now, the vertices (from left to right) adjacent to S which are not yisited as part of the search are , ç, A. Hence, B,C an4 A are visited after S as part of the BFS. Then, F is the unvisited vertex adjacent to B. Hence, the visit to B, C and A is followed by F. The unvisited vertex adjacent of C is D. So, the visit to F is followed by D. There are no unvisited vertices adjacent to A. Finally, the unvisited vertex E adjacent to D is Graphs
visited.
Hence, the sequence of vertices visited as part of BFS is S, B, C, A, F, D and E.

8.6 DEPTH FIRST SEARCH (DFS)

The strategy adopted in depth first search is to search deeper whenever possible. This algorithm repeatedly searches deeper by visiting unvisited vertices and whenever an unvisited vertex is not found, it backtracks to previous vertex to find out whether there are still unvisited vertices.
As seen, the search defined above is inherently recursive. We can find a very simple recursive procedure to visit the vertices in a depth first search. The DES is more or less similar to pre-order tree traversal. The process can be described as below:
Start from any vertex (source) in the graph and mark It visited. Find vertex that is adjacent to the source and not previously visited using adjacency matrix and mark it visited. Repeat this process for all vertices that is not visited, if a vertex is found visited in this process, then return to the previous step and start the same procedure from ther'.

If returning back to spurce is not possible, then DES from the originally selected source is complete and start DFS using any unvisited vertex.

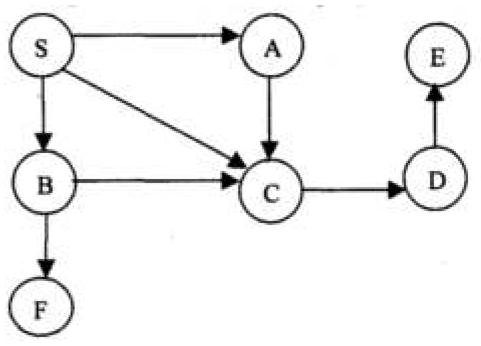


Figure 8.13 : A Digraph

Consider the digraph of Figure 8.13. Start with S and mark it visited. Then visit the next vertex A, then C and then D and at last E. Now there are no adjacent vertices of E to be visited next. So, now, backtrack to previous vertex D as it also has o unvisited vertex. Now backtrack to C, then A, at last to S. Now S has an unvisited vertex B. Start DFS with B as a root node and then visit F. Now all the nodes of the graph are visited.
Figure 8.14 shows a DFS tree with a sequence of yisits. The first number indicates the time at which the vertex is visited first and the second number indicates the time at which the vertex is visited during back tracking.

UNIT 6 TREES

6.0 INTRODUCTION

Have you ever thought how does the operating system manage our files? Why do we have a hierarchical file system? How do files get saved and deleted under hierarchical directories? Well, we have answers to all these questions in this section through a hierarchical data structure called Trees! Although most general form of a tree can be defined as an acyclic graph, we will consider in this section only rooted tree as general tree does not have a parent-child relationship.

Tree is a data structure which allows you to associate a parent-child relationship between various pieces of data and thus allows us to arrange our records, data and files in a hierarchical fashion. Consider a Tree representing your family structure. Let us say that we start with your grand parent; then come to your parent and finally, you and your brothers and sisters. In this unit, we will go through the basic tree structures first (general trees), and then go into the specific and more popular tree called binary- trees.

6.2 ABSTRACT DATA TYPE-TREE

Definition: A set of data values and associated operations that are precisely specified independent of any particular implementation.
Since the data values and operations are defined with mathematical precision, rather than as an implementation in a computer language, we may reason about effects of the

Operations, relationship to other abstract data types, whether a programming language implements the particular data type, **etc.**

Consider the following abstract data type:
Structure Tree
type Tree nil fork (Element , Tree, Tree)
Operations:
null : Tree -> Boolean
leaf: Tree -> Boolean
fork : (Element Tree , Tree) **-> Tree**
left : Tree -> Tree *I/It* **depicts the properties of tree that left of a tree is also a** tree.
right: Tree -> Tree
contents: **Tree -> Element**
height (nil) 0
height (fork(e,TT')) = I +max(height(T), heigbt(T'))
weight (nil) =0
weight (fork(e,T,T)) H-weigbt(T)+weight(T')

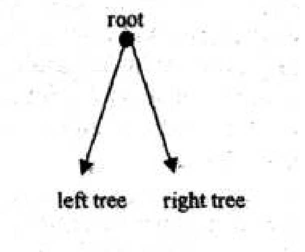


Figure 6.1: A binary tree

Rules:
null(nil) = true //nil **is an empty tree**
null(fork(e, **T,** *V))=* false */1* **•e: element, T** andT are two sub **tree**
leaf(fork(e, aiI, nil)) true
leaf(fork(e, T, T)) false if not rzull(T) or not null(V)
leatnil) error
left(fork(e, T, 1')) = T
left(nil) = error
right(fork(e, T, 1")) = r
right(nil) error

contents(fork(e, **T T'))** **e**
contents(nil) error.
Look at the definition of Tree **(ADT). A way to** think **of a** *binary* tree **is that it is either empty (nil) or** contains an element and two sub trees which are themselves **binary trees** (Refetto *F(gure6.1).*
Fork operation joins two **aub tree with aparent node** and

produces another Binary tree. It may be noted that a tree consisting of a single leaf Is **Tre** defined to be of height 1.
Definition : A tree is a connected, acyclic graph (Refer to *Figure* 6.2).
It is so connected that any node in the graph can be reached from any other node by exactly one path. It does not contain any cycles (circuits, or closed paths), which would imply the eisteace of more than one path between two nodes. This is the most general kind of tree, and **my be** converted into the more familiar form by designating a node as the root. We can represent a tree as a construction consisting of nodes, and edges which represent a relationship between twQ nodes. In *Figure 6.3,* we will consider most common **tree called rooted tree. A** rooted tress has a single root node which has no parents.

In a more formal way, we can define a tree T as a finite set of one or more nodes such that there Is one designated node r called the root of T, and the remaining nodes in (T – { r)) are partitioned into n> 0 disjoint subsets T1, T2, ... T each of which is a tree, and whose roots ri , r2, ... rk , respectively, are children of r. The general tree is a generic tree that has one root node, and every node in the tree can have an unlimited number of child nodes. One popular use of this kind of tree is a Family Tree. **A tree** Is an instance of a more general category called graph.
• A tree conIts **of nodes** connected by edges.
• A root Is a node without parent.
• Ueaves are nodes with no children.
• The root is at level I. The child nodes of root are at level 2. The child nodes of nodes at level 2 are at level 3 and so **on.**
• The depth (height) of a inary tree is equal to the number of levels in it.
• Branching fhctor defines the maximum number of children **to** any node. **So, a** branching factor of 2 means a binary tree.

ge

Figure 6.3 Tree as $ coniccte4 acyclic grapb

Root
Internal node – **node**

L.evcl
Lçvel 2
Level S

FIgure 6. A rooted tree

operations, relationship to other abstract data types, **whether a programming language implements the particular data type, etc.**
Consider the following abstract data type:
Structure Tree
type Tree nil | fork (Element ,Tree , Tree)
Operations:
null : Tree -> Boolean
leaf: Tree -> **Boolean**
fork : (Element **Tree, Tree) -> Tree**
left : Tree-> **Tree *I/It* depicts the properties of tree that left oh tree is** also **a tree.**
right: Tree -> Tree
contents: Tree -> Element
height *(nil)* =0
height (fork(e,T,P)) = l+max(height(T), heigJit(T))
weight (nil) =0 |
weight (fork(e,T,T')) **= I +weight(T)+weight(T')**
root
left tree right tree

Figure 6.i A binary tree

Rules:

null(nil) = true *II* **nil is an** empty tree

nu1iØk(e, T, **V))** false *II* •e : element, T and T are two sub tree

1eaf(forke, all, nil)) true

leaf(fork(e, T, 1”)) = **false if not nuIl(T) or not null(T’)**

leaf(nil) = **error**

lefl(fork(e, T, T)) T

left(nil) = error

right(fork(e, T, T’)) = **T**

right(nil) error

contents(fork(e, **T,, T)) e’**

contents(nil) = error

Look at the **definition of Tree (ADT). A way to think of a** *binary* tree **is that it is either** empty **(nil)** or contains an element and two sub trees which are themselves binary frees (Refer to *Figure* ***6.1).*** Fork operation joins two sub tree with aparent node and

produces another Binary tree, It m.y be noted that a tree consisting of a single leaf Is Trees defined to be of height 1.

Definition : A tree is a connected, acyclic graph (Refer to *Figure 6.2).*

It is so connected that any node in the graph can be reached from any other node by exactly one path. It does not contain any cycles (circuits, or closed paths), which would imply the existence of more than one path between two nodes. This Is the most general kind of tree, and may bç converted into the **more** familiar form by designating a node as the root. We can represent a tree a construction consisting of nodes, and edges which represent a relationship between two nodes. In *Figure 6.3,* we will consider most common tree called **rooted tree. A rooted** tress has a single root node which has no parents.

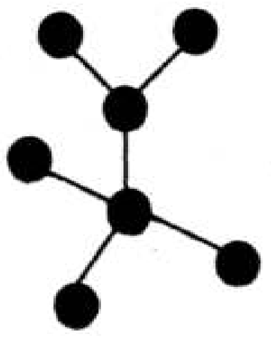


Figure 6.2 : Tree as a connected acyclic graph

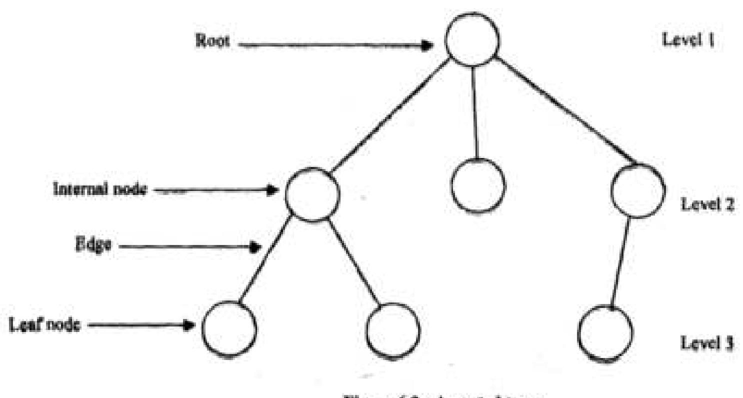


Figure 6.3 : A rooted tree

In a more formal way, we can define a tree T as a finite set of one or more nodes such that there is one designated node r called the root of T, and the remaining nodes in **(T** —(r)) are partitioned into n> 0 disjoint subsets T1, T2, ... T each of which is a tree, and whose roots ri , r2, ..., rk , respectively, are children of r. The general tree is a generic tree that has one root node, and every node in the tree can have an unlimited number of child nodes. One popular use of this kind of tree is a Family Tree. A tree is an instance of a more general category called graph.

• A **tree** consists of nodes connected by edges.
• A root Is a node without parent.
• Leaves are nodes with no children.
• The root is at level 1. The child nodes of root are at level 2. The child nodes of nodes at level 2 are **at level 3** and so on.
• The depth (height) **of a** Binary tre is equal to the number of levels in it.
• Branching factor defines the maximum number of children toy node. So, a branching factor of 2 means a binary tree.

• Breadth defines the number of nodes at a level.
a The depth of a node M in a tree is the length of the path from the root of the tree
to M
• A node In a Binary tree has at most 2 children.
The following are the properties of a Tree.
Full Tree A e with all the leaves at the same level, and all the non-leaves having the same degree
• Level h of a full tree has d nodes.
• The first levels of a fulltree hnve l+d+d+d +d4+ +dh (d
- 1) nodes where d is the degree of nodes.
• The iwmber of edges = the number of nodes — 1 (Why? Because, an edge re'esents the relationship between a child and a parent, and every node has a
• parent except the rt.
• A tree of height *h* and degree *d* has at most *d* **h** - 1 elements.

Complete Trees

Tree js a dynamic data structures. Trees can expand and contract as the program executes and are implemented through pointers. A tree deallocates memory when an element is deleted.
Non-linear data structures: Linear data structures have properties of ordering relationship (can the elements/nodes of tree be sorted?). There is no first node or last node. There is no ordering relationship among elements of tree. Ites of tree can be partilly ordered Into a hierarchy vIa parent-child relationship. soot node is at the top of the hierarchy and leafs are at the bottom layer of the hierarchy. Hence, trees pan be termed as hierarchical data structures,

6.3 IMPLEMENTATION OF TREE

The most common way to add nodes to a general tree Is to first find the desired parent of the node you want to insert, then add the node to the parent's child list. The most common implementations insert the nodes one at a time, but since egch node can be considered a tree on its own, other implementations build up an entire sub-tree before adding it to a larger tree. As the nodes are added and deleted dynamIcally from a tree, tree are often implemented by link lists. However, it is simpler to write algorithms for a data representation where the numbers of nodes are fixed. *Figure* 6.4 depicts the structure of the node of a general k-ary tree.

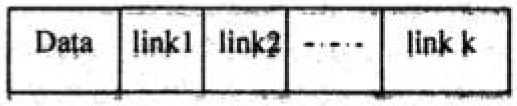


Figure 6.4 : Node structure of a general k-ary tree

-

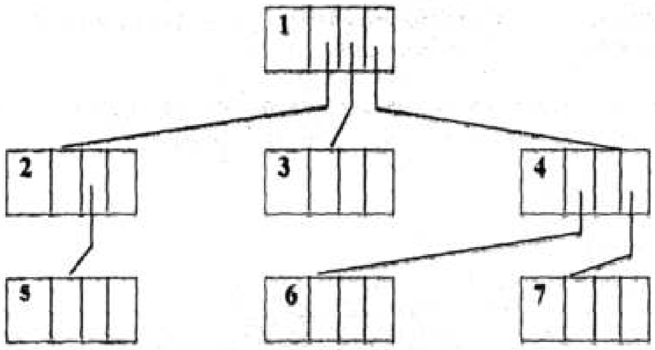


Figure 6.5: A linked list representation of tree (3-ary tree)

-

- *Figure 6.5* depicts a tree with one data element and three pointers. The number of pointers required to implement a general tree depend of the maximum degree of nodes in the tree.

6.4 TREE TRAVERSALS

There are three types of tree traversals, namely, Preorder, Postorder and Inorder. *Preorder traversal:* Each node is visited before its children are visited; the root i visited first.

Algorithm for **pre-**order traversal;

1. visit root node
2. traverse left sub-tree in preorder
3. traverse right sub-tree In preorder

Example ofpre order traversal: Reading of a book, as we do not read next chapter unless we complete all sections of previous chapter and all it's sections (refer to *Figure 6.6).*

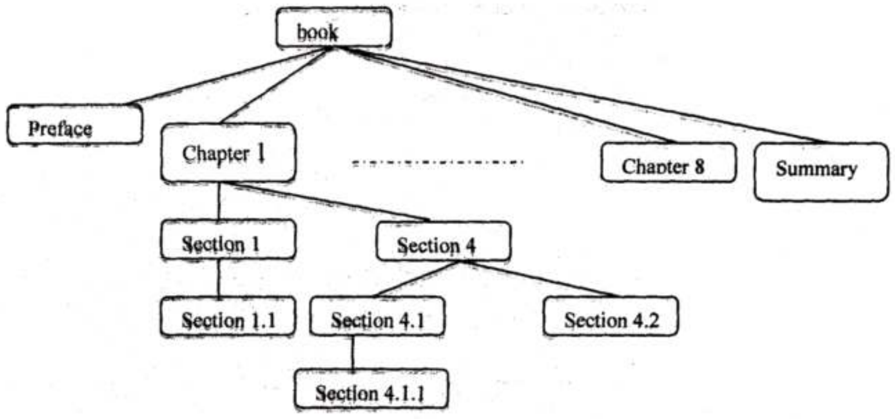


Figure 6.6 : Reading a book : A preorder tree traversal

As each node is traversed only once, the time complexity of preorder traversal is
T(n) 0(n), where n is number of nodes in the tree.
Postorder traversal: The children of a node are visited before the node itself; the root
is visited last; Every node is visited after its descendents are visited. Algorithm **for postorder traversah**
1. traverse left sub-tree in post order
2. traverse right sub-tree in post order
3. visit root node,
Finding the space occupied by files and directories in a file system requires a
postorder traversal as the space occupied by directory requires calculation of space
required by all files in the directory (children in tree structure) (refer **to** *Figure 6.7)*

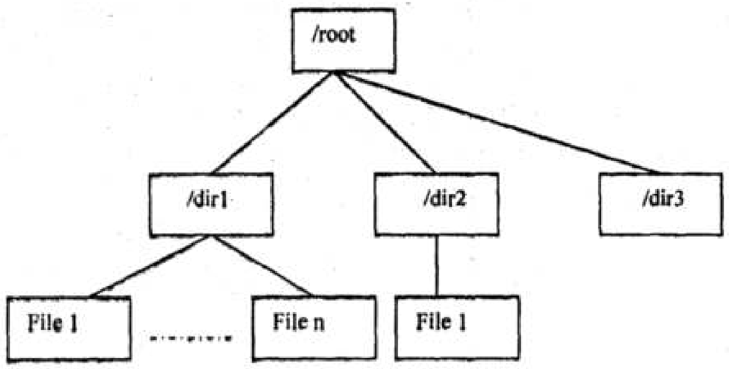


Figure 6.7 : Calculation of space occupied by a file system : A post order traversal

As each node is traversed only once, the time complexity of post order traversal is
T(n) = 0(n), where n Is number of nodes in the tree.
Inorder traversal: The left sub tree is visited, then the node and then right sub-tree.
Algorithm for inorder traversal:
1. traverse left sub-tree
2. visIt node
3. traverse right sub-tree

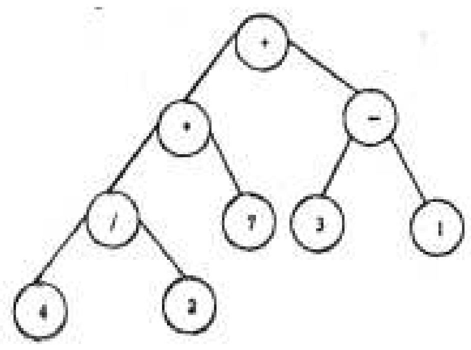


Figure 6.8 : An expression tree : An inorder traversal

Inordtr traversal can be best described by an expression tree, where the operators are **Trrs** at paEent node and operands are at leaf nodes.
Let ps consider the above expressiort tree (refer to *Figure 6.8).* The preorder, postorder and inorder traversal are given below:
preorder Traversal: + * / 4 2 7 - 3 1
postorder traversal: 4 2 / 7 * 3 1 - +
inorder traversal : - *((((4 /* 2) * *7)* + (3 - 1))
There is another tree traversal (of course, not very common) is called level order, where all the nodes of the same level are travelled first starting from the root (refer to *Figure 6.9).*

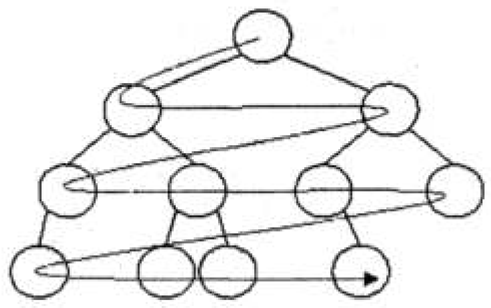


Figure 6.9: Tree Traversal: Level Order

6.5 BINARY TREES

A binary tree is a special tree where each non-leaf node can have atmost two child nodes. Most important types of trees which are used to model yes/no, on/off, higher/lower, i.e., binary decisions are binary trees.
Recursive Definition: A binary tree is either empty or a node that has left and right sub-trees that are binary trees. Empty trees are represented as boxes (but we will almost always omit the boxes).
In a formal way, we can define a binary tree as a finite set of nodes which is either empty or partitioned in to sets of T0, T1, **Tr,** where T0 is the root and T1 and Tr are left and right binary trees, respectively.

Properties of a binary free

- If a binary tree contains it nodes, then it contains exactly *n* – 1 edges;
- A Binary tree of height *h* has 2” modes or less,
- If we have a binary tree containing *n* nodes, then the height of the tree is at most *n* and at least ceiling log2(n + 1).
- If a binary tree has n nodes at a level I then, it has at most 2n nodes at a level 1+1
- The total number of nodes in a binary tree with depth d (root has depth zero) is N 20+21+22+

Full Binary Trees: A binary tree of height h which had 2” —l elements is called a Full Binary Treed

(exmplete Binary Trees: A binary tree whereby if the height is d, and all levels, except possibly level d, are completely full. If the bottom level is incomplete, then it has all nodes to the left side. That is the tree has been filled in the level order from left to right.

66 IMPLEMENTATION OF A BINARY TREE

Like general tree, binary trees are implemented through linked lists. A typical node in a Binary tree has a structure as follows (refer to *Figure 6.10):*

struct NODE

{

struct NODE *leftchjld;

jut nodevalue; *1** this can be of any data type **1* struct NODE *rightthild;

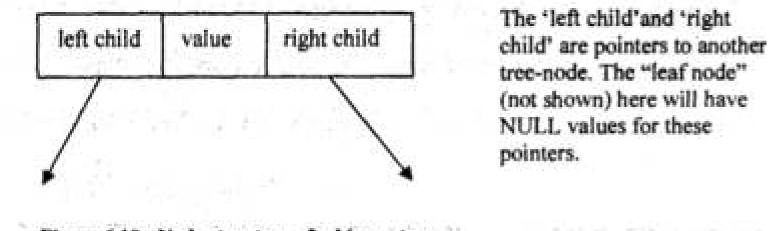

left child
value
right child
The 'left child'and 'right child' are pointers to another tree-node. The "leaf node" (not shown) here will have NULL values for these pointers.

Figure 6.10 : Node structure of a binary tree

The binary tree creation follows a very simple principle. For the new efeent to be added, compare it with the current element in the tree. If its value is less than the current element in the tree, then move towards the left side of that element or else to its right. If there is no sub tree on the left, then make your new alement as the left child of that current element or else compare it with the existing left child and follow the same rule. Exactly, the same has to done for the case when your new element is greater than the current element in the tree but this time with the right child. Though this logic is followed for the creation of a Binary tree, this logic is often suitable to search for a key value in the binary tree.

Algorithm for the implementation of a Binary tree:

Step-I: If value of new element <current element, then **go to** step-2 or else step -3 Step-2: If the current element does not have a left sub-tree, then make your new element the Jdfthe teiemi., else theexisting left child **TPCCS**
as your curfentelement and go testep-i ..
Step-3: If the current element does not have a right sub-tree, then make your new
element the right child of the current element; else make the existing right

child as your current element and go to step-1
Program 6.1 depicts the segment of code for the creation of a binary tree.

```
struct NODE
{
struct NODE * left;
int value;
struct NODE *right;
create_tree( struct NODE *curr, struct NODE * new)
{
if(new->value <= curr->value)
{
if(curr..>Ieft != NULL)
create_tree(curr->left, new);
else
curr->left = new;
}
else
{
if(curr->right != NULL)
create_tree(curr->right, new);
else
curr->right = new;
}
}
```

Program 6.1 Binary tree creation

Array-based representation of a Binary Tree

Consider a complete binary tree T having n nodes where each node contains an item (value). Label the nodes of the complete binary tree T from top to bottom and from left to right 0, 1, ... n-i. Associate with T the array A where the **ith** entry of A is the item in the node labelled i of T, i = **0, 1,** ... n-i. Figure 6.11 depicts the array representation of a Binary tree of Figure 6.16.

Given the index i of a node, we can easily and efficiently compute the index of its parent and left and right children:

Index of Parent: (i – 1)/2, Index of Left Child: 2i + 1, Index of Right Chil& 2i + 2.

Node #	**Item**	**Left child**	**Rightchild**
0	**A**	**I**	2
1	**B**	**3**	**4**
2	**C**	**-1**	**-I**
3	**D**	***5***	***6***
4	**E**	7	**S**
5	***0***	-1	**-1**
6	**I-I**	**-1**	**-1**

7	I	-1	-1
8	J	-1	-1
9	*?*	*?*	*?*

Figure 6.11 Array Representation of a Binary Tree

First column represents index of node, second column consist of the item stored in the **and Trees** node and third and fourth columns **indicate** the positions of left and right children
(—1 indicates that there is no child to that particular node.)

7 BINARY TREE TRAVERSALS

We have already discused about three tree traversal methods in the previous section on general tree. The same three different ways to do the traversal _preorder, inorder and postorder are applicable to binary tree also,
Let us discuss the inorder binary tree traversal for following binary tree (refer to
Figire 612):
We start from the root i.e. * We are supposed to visit its left sub-tree then visit the node itself and its right sub-tree. Here, root has a left sub-tree rooted at +. So, we move to + and check for its left sub-tree (we are suppose repaeat this for every node). Again, + has a left sub-tree rooted at 4. So, we have to check for 4's 'eft sub-tree now, but 4 doesn't have any left sub-tree and thus we will visit node 4 first (print in our case) and check for its right sub-tree. As 4 doesn't have any right stab-tree, we'll go back and visit node +; and check for the right sub-tree of +. It has a right sub-tree rooted at *5* and so we move to *5*. Well, *5* doesn't have any left or right sub-free. So, we just visit *5* (print 5)and track back to +. As we have already visited + so we track back to * . As we are yet to visit the node itself and so we visit * l,efore checking for the right sub-tree of *, which is 3. As 3 does not have any left or right sub-trees, we visit
3.
So, the inorder traversal results in 4 + *5* * 3

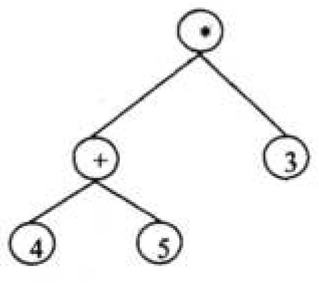


Figure 6.12 : A binary tree

Algorithm: Inorder

```
Step-i: For the current node, check whether it has a left child. If it has, then go to step-2 or
else go to step-3
Step-2: Repeat step-I for this left child
Step-3: Visit (i.e. printing the node in our case) the curreht node
Step-4: For the current node check whether it has a right child. If it has, then go to step-S
Step-5: Repeat step-i for this right child
```

The preoreder and postorder traversals are similar to that of a general binary tree. The general thing we have seen in all these tree traversals is that the traversal mechanism is inherently recursive in nature.

6.7.1 Recursive Implementation of Binary Tree Traversals

There are three cla.ssic ways of recursively traversing a binary tree. **In** each of these, the left and right sub-trees are visited recursivoly and the distinguishing feãtuie is **when the element in the root is visited or processed.**

Program 6.2, Program 6.3 and Program 6.4 depict the inorder, preorder and postorder traversals of a Binary tree.

```
struct NODE
{
struct NODE * left; Trees
 int value; /* can be of any type */
struct NODE *right;
inorder(struct NODE *curr)
{
if(curr->left != NULL) inorder(curr->left);
printf"%d", curr->value);
if(curr->right ! NULL) inorder(curr->right);
```

Program 6.2 : Inorder traversal of a binary tree

```
struct NODE
{
struct NODE * left;
mt value; /* can be of any type *1
struct NODE *right;
preorder(struct NODE *curr)
{
printf("%d", curii->value);
if(curr->Ieft ! NULL) preorder(curr->left);
if(curr->right ! NULL) preorder(curr->right);
)
```

Program 6.3 : Preorder traversal of a binary tree

```
ght struct NODE
struct NODE *left;
int value; /* can be of any type */
struct NODE *right;
postorder(struct NODE * cuff)
{
ifcurr->left ! NULL) postorder(curr->left);
ifcurr->right != NULL) postorder(curr->right);
printf"%d", curr->value);
}
```

Program 6.4 : Postorder traversal of a binary tree

In a preorder traversal, the root is visited first (pre) and then the left and right subtrees are traversed. **In a** postorder traversal, the left sub-tree is visited first, followed by right sub-tree which is then followed by root. In an inorder traversal, the left subtree is visited first, followed by root, followed by right sub-tree.

6.7.2 Non-recursive implementation of binary tree traversals

As we have seen, as the traversal mechanisms were inherently recursive, the implementation was also simple through a recursive procedure. However, in the case of a non-recursive method for traversal, it has to be an iterative procedure; meaning, all the steps for the traversal of a node have to be under a loop so that the same can be applied to all the nodes in the tree.

Algorithm : Non-recursive preorder binary tree traversal

```
Stack S
push root onto S
repeat until S is empty
{
v= pop S
if v is not NULL
visit v
push v's right child onto S
push v's left child onto S
}
```

Program 6.5 depicts the program segment for the implementation of non-recursive preorder traversal.

```
/ preorder traversal of a binary tree, implemented using a stack */
void preorder(binary_tree_type tree)
{
stack_type *stack;
stack create_stackO;
push(tree, stack); /* push the first element of the tree to the stack *1 while (!empty(stack))
{
tree = pop(stack);
visit(tree);
push(tree->riglit, stack); /* push right child to the stack */ L
push(tree->left, stack); /* push left child to the stack *1
}
}
```

Program 6.5: Non-recursive implementation of preorder traversal

In the worst case, for preorder traversal, the stack will grow to size n/2, where n is number of nodes in the tree. Another method of traversing binary tree non-recursively which does not use stack requires pointers to the parent node (called threaded binary tree).

A threaded binary tree is a binary tree in which every node that does not have a right child has a THREAD (a third link) to its INORDER successor. By doing this threading we avoid the recursive method of traversing a tree and use of stack, which makes use of a lot of memory and time.

A node structure of threaded binary is:

The node structure for a threaded binary tree varies a bit and its like this –

```
struct NODE
{
struct NODE leftchild; Trees
int node value;
struct NODE *rightchild;
struct NODE *thread; /* third pointer to it's inorder successor *1
}
```

6.8 APPLICATIONS

Trees are used enormously in computer programming. These can be used for improving database search times (binary search trees, 2-3 trees, AVL trees, red-black trees), Game programming (minimax trees, decision trees, pathfinding trees), 3D graphics programming (quacltrees, octrees), Arithmetic Scripting languages (arithmetic precedence trees), Data compression (Huffman trees), and file systems (Btrees, sparse indexed trees, tries). Figure 6.13 depicts a tic-tac-toe game tree showing various stages of game.

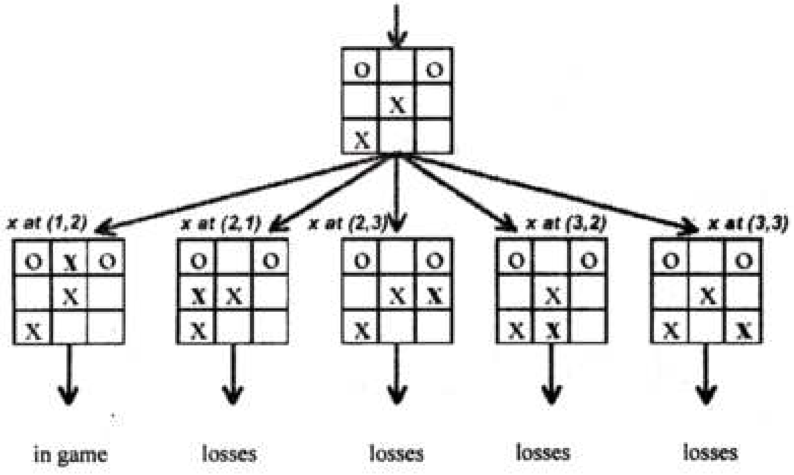


Figure 6.13 : A tic-tac-toe game tree showing various stages of game

In all of the above scenario except the first one, the player (playing with X) ultimately looses in subsequent moves.
The General tree (also known as Linked Trees) is a generic tree that has one root node, and every node in the tree can have an unlimited number of child nodes. One popular use of this kind of tree is in Family Tree programs. In game programming, many games use these types of trees for decision-making processes as shown above for tic-tac-toe. A computer program might need to make a decision based on an event that happened.
But this is just a simple tree for demonstration. A more complex Al decision tree would definitely have a lot more options. The interesting thing about using a tree for decision-making is that the options are cut down for every level of the tree as we go down, greatly simplifying the subsequent moves and improving the speed at which the Al program makes a decision.
The big problem with tree based level progressions, however, is that sometimes the tree can get too large and complex as the number of moves (level in a tree) increases. Imagine a game offering just two choices for every move to the next level at the end of each level in a ten level game. This would require a tree of 1023 nodes to be created.

Binary trees are used for searching keys. Such trees are called Binary Search trees(refer to *Figure 6.14).*
A Binary Search Tree (BST) is a binary tree with the following properties:
1. The key of a node is always greater than the keys of the nodes in its left sub-tree
2. The key of a node is always smaller than the keys of the nodes in its right sub-tree
root

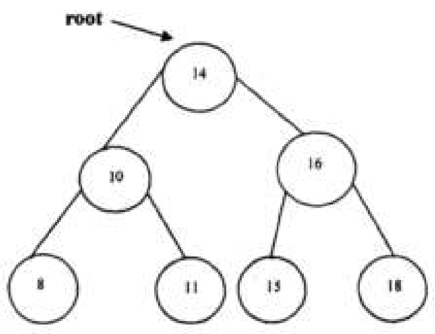


Figure 6.14 : A binary search tree (BST)

It may be seen that when nodes of a BST are traversed by inorder traversal, the keys appear in sorted order:

```
inorder(root)
{
inorder(root.left)
print(root.key)
inorder(root.right)
}
```

Binary Trees are also used for evaluating expressions.
A binary tree can be used to represent and evaluate arithmetic expressions.
1. If a node is a leaf, then the element in it specifies the value.
2. If it is not a leaf, then evaluate the children and combine them according to the operation specified by the element.
Figure 6.13 depicts a tree which is used to evaluate expressions.

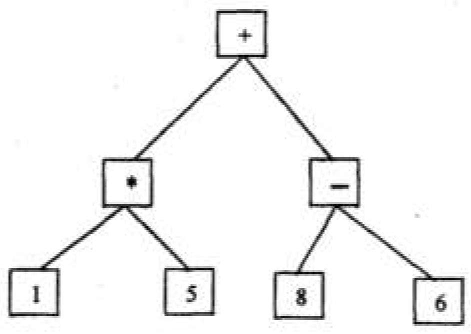


Figure 6.15 : Expression tree for 1 * 5 + 8 - 6