S. E. : Analysis of Algorithms

Notes GQ

CONTENTS

Vidyalankar

S.E.[CMPN] : Analysis of Algorithms

Syllabus

Paper : 3 Hrs. **Marks : 100**
Term work : 25

1. **Algorithm Analysis :**
   - Mathematical Background;
   - The Model;
   - The Time Complexity:
     How to Analyze and Measure, Big-Oh and Big-Omega Notations, Best Case, Average Case and Worst Case Analyses.

2. **Sorting Methods :**
   - Efficiency Considerations in Sorting;
   - Different Sorting Methods:
     Bubble Sort, Quicksort, Straight Selection Sort, Binary Tree Sort, Heaps and Heapsort, Heap as Priority Queue, Insertion Sort; Shell Sort; Bucket Sort; Merge Sort; Radix Sort, Time Complexity Calculation, Best Case, Worst Case and Average Case Calculations of the Different Sorting Methods.

3. **Searching Methods :**
   - Efficiency Consideration in Searching:
   - Basic Searching Techniques:
     Sequential Search, Efficiency Considerations for Sequential Search, Searching Ordered Table, Indexed Sequential Search, Binary Search, Interpolation Search, Binary Search Tree, Implementation, Insertions and Deletions, Efficiency Considerations, General Search Trees, Multiway Search Trees, B-Trees, B+- Trees, Tries, AVL Trees.
   - Hashing:
     Hash Functions, Resolving Clashes (Open and Closed Hashing), Hashing in External Storage, Dynamic Hashing.

4. **Graph:**
   - Graph Traversal;
   - Application of Graph Structures:
     Shortest Path Problem, Topological Sorting, Minimum Spanning Tree, Connectivity in a Graph, Euler's and Hamiltonian Graph.

5. **Algorithms :**

   Analysis of all the above Algorithms, Greedy Method, Divide and Conquer Method, Dynamic Programming, Back-Tracking Method.

❑❑❑❑❑❑

The Big–C
If f(n) and g
is big O of g
large positiv

Under these
rapidly than

Example :
Consider the
can also be v

Considering
we cannot si
If we choose
O(n).

Considering
that, for all s
Cn, when n i
Hence f(n) is

If we take C
100n, we wi
C = 3, but an

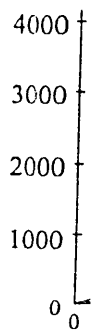
These examp
notation. To
disregarding

If f(n) is a pol
less than r.

Any logarithn
n is $O(n^k)$ for
When we app
some algorith
$O(n)$ n
$O(n^2)$ i
$O(n^3)$ i
$O(2^n)$ i

These five orc
commonly use

The following



Vidyalankar

Ch. 1 : Algorithm Analysis

Marks : 100
Term work : 25

st Case, Average

eaps and Heapsort,
e Sort; Radix Sort,
Calculations of the

arching Ordered
, Binary Search
rations, General
ees.

ing in External

nnectivity in a

uer Method,

The Big–O Notations :

If f(n) and g(n) are functions defined for positive integers, then to write f(n) is O(g(n)). [read f(n) is big O of g(n)] means that there exists a constant C such that $|f(n)| \leq C|g(n)|$ for all sufficiently large positive integers n.

Under these conditions we also say that "f(n) has order at most g(n)" or "f(n) grows no more rapidly than g(n)".

Example :

Consider the function f(n) = 100n. Then f(n) is O(n), since f(n) ≤ Cn for the constant C = 100. It can also be work for any large constant C > 100.

Considering next, the function f(n) = 4n + 200. Since f(n) ≤ 4n is not true for large values of n, we cannot simply show that f(n) is O(n) by taking C = 4. We can choose any larger value for C. If we choose C = 5, then we can find that f(n) ≤ 5n. Whenever n ≥ 200 and therefore f(n) is again O(n).

Considering the next example of polynomials in 'n', choose a function f(n) = $3n^2$ – 100n. Such that, for all small values of n, f(n) is less than n, and for any constant c, f(n) will be greater than Cn, when n is sufficiently large.
Hence f(n) is not O(n).

If we take C = 3, then we can have f(n) ≤ $3n^2$, so f(n) is O(n^2) is true. If we change f(n) to $3n^2$ + 100n, we will still have f(n) is O(n^2). Thus in above case, we would not be able to use C = 3, but any larger value for C would work.

These examples of polynomials generalize to the first and most important rule about the big–O notation. To obtain the order of polynomial function, we extract the term with highest degree, disregarding all constants and all terms with a lower degree.

If f(n) is a polynomial in n with degree r, then f(n) is O(n^r), but f(n) is not O(n^s) for any power 's' less than r.

Any logarithm of n grows more slowly (as n increases) than any positive power of n. Hence log n is O(n^k) for any k > 0, but n^k is never O(log n) for any power k > 0.
When we apply the big–O notation, f(n) will normally be the operation count or running time for some algorithm O(1) means computing time that is bounded by a constant (not dependent on n);

O(n) means that the time is directly proportional to n, and is called **linear time.**
O(n^2) is called **quadratic time.**
O(n^3) is called **cubic time** and
O(2^n) is called **exponential time.**

These five orders, together with log–arithmetic time O(log n) and O(n log n), are the ones most commonly used in analyzing algorithms.

The following figure shows how these seven functions grow with n.

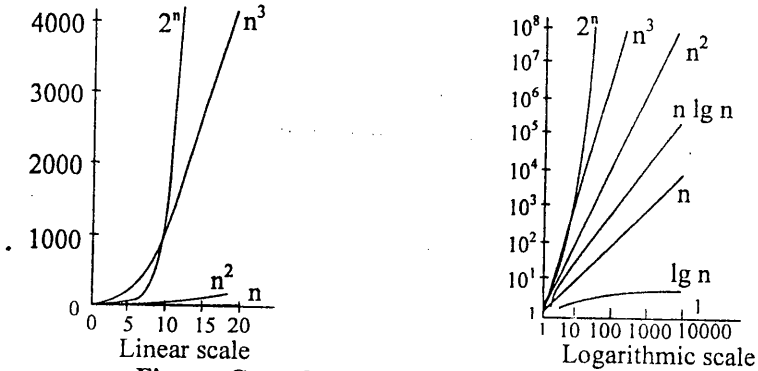


Figure: Growth rates of common functions

Algorithm Analysis :

We can express the conclusions of our algorithm analyses in very simple manner as shown below

(i) On a list of length n, sequential search has running time O(n).
(ii) On an ordered list of length n, binary search has running time O(log n)
(iii) Retrieval from a contiguous list of length 'n' has running time O(1).
(iv) Retrieval from a linked list of length 'n' has running time O(n).

Big–Oh and Big–Omega Notation :

"Big–Oh" notation is used in case of growth rates of functions.

For example, when we say the running time T(n) of some program is $O(n^2)$, it can be also read as "big oh n squared" or just "oh of n squared" i.e. there are positive constants c and no for all $n \geq no$. We have $T(n) = c\,n^2$

Example :

Suppose T(o) = 1, T(1) = 4, and in general $T(n) = (n+1)^2$, then we see that T(n) is $O(n^2)$. Let no = 1 and c = 4 in above case.

i.e. for $n \geq 1$, we have $(n+1)^2 \leq 4n^2$

i.e. $T(n) \leq c\,n^2$ and it is easily proved.

Assume, all running time functions are defined on the non–negative integers, and their values are always negative. We say that T(n) is O(f(n)) if there are constants 'c' and 'no' such that $T(n) \leq c\,f(n)$ whenever $n \geq no$.
A program whose running time is O(f(n)) is said to have growth rate f(n).

Big–Omega Notation :

When we say T(n) is O(f(n)), it is to be known that f (n) is an upper bound on the growth rate of T(n). To specify a lower bound on the growth rate of T(n), we can use the notation T(n) is Ω (g(n)). It can be read as "a big omega of g(n)" or just "omega of g(n)" to mean that there exist a positive constant c such that, $T(n) \geq c\,.\,g(n)$

Example :

To verify that the function $T(n) = n^3 + 2n^2$ is $\Omega(n^3)$. Let c = 1, then $T(n) \geq c\,n^3$ for n = 0,1,. .

For another example, Let T(n) = n for odd $n \geq 1$ and $T(n) = n^2/100$ for even $n \geq 0$. To verify that T(n) is $\Omega(n^2)$, Let c = 1/100 and consider the infinite set of n's : n = 0, 2, 4, 6,.....

MODEL :

A problem can be expressed in terms of a formal model, it is usually beneficial to do so, for once a problem is formalized, we can look for solutions in terms of a precise model and determine whether a program already exists to solve that problem.

Even if there is no existing program, at least we can discover what is known about this model and use the properties of the model to help construct a good solution.

Problems essentially numerical in nature can be modeled by mathematical concepts such as, simultaneous linear equations (e.g. Finding currents in electrical circuits, of finding stresses in frames made of connected beams) or differential equations (e.g. predicting population growth or the rate at which chemicals will react).

Symbol and text processing problems can be modeled by character strings and formal grammars.

Problems of this nature include compilation and information retrieval tasks such as recognizing particular words in lists of titles owned by a library.

Once we have a suitable mathematical model for our problem, we can find a solution in terms of that model.

Initial goal is to find a solution in the form of an algorithm, which is a finite sequence of instructions, each of which has a clear meaning can be performed with a finite amount of effort in a finite length of time.
x : = y + z is an example of an instruction that can be executed in a finite amount of effort.

Instructions can be ex
repetition.

An algorithm termina
algorithm as long as it

Each instruction of a
"finite amount of effo

An instruction can be

Example :

A mathematical mode
roads. To construct t
permitted turns at an
set into as few group
without collisions. W
partition. By finding
light with the smalles

Abstract data type (A
defined on that mode

In an ADT, the opera
other types of operar
can be other than an

There are two proper

ADT's are generaliz
generalizations of pr

The ADT encapsula
on that type can be l

Graded Questions :

1. Define big oh an
2. Write a short not
3. What are time ar
4. Define O, Ω and
5. Compare the ti
   in terms of big –
6. Give examples c
   (i) P-time comp
   (iv)NP-complet
7. State and explai
8. Write a progran
   sort. Also derive
9. To prove that cc
10. Explain Best ca
11. To find the wor
    (i) Insertion
12. Give, using 'big
    function of n.
    (a) procedu
        var
        int
        {
        for i
        f
        }

Instructions can be executed any number of times, provided the instructions themselves indicate repetition.

An algorithm terminate after executing a finite number of instructions. Thus, a program is an algorithm as long as it never enters an infinite loop on any input.

Each instruction of an algorithm must have a "clear meaning" and must be executable with a "finite amount of effort".

An instruction can be carried out in a finite amount of time.

Example :

A mathematical model can be used to help design a traffic light for a complicated intersection of roads. To construct the pattern of lights, we shall create a program that takes as input a set of permitted turns at an integer section (continuing straight on a road is a "turn") and partitions this set into as few groups as possible such that, all turns in a group are simultaneously permissible without collisions. We shall then associate a phase of the traffic light with each group in the partition. By finding a partition with the smallest number of groups, we can construct a traffic light with the smallest number of phases.

Abstract data type (ADT) is consider as a mathematical model with a collection of operations defined on that model.

In an ADT, the operations can take as operands not only instances of the ADT being defined but other types of operands, e.g. integers or instances of another ADT, and the result of an operation can be other than an instance of that ADT.

There are two properties of procedures. (i) generalization (ii) encapsulation

ADT's are generalizations of primitive data types (integer, real and so on), just as procedures are generalizations of primitive operations (+, –, and so on).

The ADT encapsulates a data types in the sense that the definition of the type and all operations on that type can be localized to one section of the program.

Graded Questions :

1. Define big oh and big omega. **[M-03, N-05, 06]**
2. Write a short note on complexity of algorithms and its use.
3. What are time and space complexities of algorithms.
4. Define O, Ω and θ notations. What do they represent.
5. Compare the time complexities of the following algorithms giving their complexities in terms of big – O, Ω, θ. Quick sort, Heap sort, Shell's sort, Insertion sort. **[M-06]**
6. Give examples of the following types of algorithms based on complexity :
   (i) P-time complexity (ii) Constant space algorithm (iii) NP-Hard algorithm
   (iv)NP-complete algorithm (v) Constant time algorithm.
7. State and explain the small and big Asymptotic Notations. **[M-07]**
8. Write a program to sort the given n numbers in ascending order using non recursive merge sort. Also derive its complexity. **[M-07]**
9. To prove that complexity of heap sort is $O(n\log_2 n)$. Write a program for heap sort. **[M-07]**
10. Explain Best case, Average case and Worst case analysis.
11. To find the worst case complexity of : **[D-07]**
    (i) Insertion sort (ii) Merge sort
12. Give, using 'big oh' notation, the worst case running times of the following procedures as a function of n.

(a)
```
procedure matmpy (int n)
    var
    int  i, j, k;
{
    for i : = 1 to n do
        for j : = 1 to n do
        {
            c [i, j] : = 0;
```

```
                for k : = 1 to n do
                    c [i, j] : = c [i, j] + A [i, k] * B [k, j]
            }
        }
```

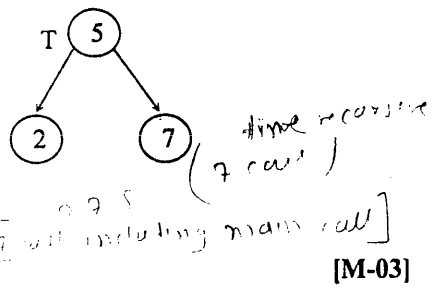
(b)
```
    procedure mystery (int n)
        int  i, j, k;
    {
        for i :  =  1 to n – 1 do
        for j :  =  i + 1 to n do
        for k :  =  1 to j do
        /* some statement requiring O(1) time */
    }
```

(c)
```
    procedure veryodd (int n)
        int  i, j, x, y;
    {
        for i : = 1 to n do
        if odd (i)
        {
            for j : = i to n do
                x : = x + 1;
            for j : = 1 to i do
                y : = y + 1
        }
    }
```

(d)
```
    int function recursive (int n)
    {
        if  n < = 1
            return (1)
        else
            return [recursive (n – 1) + recursive (n – 1)]
    }
```

13. State how recursion is disadvantageous wrt time and space. Consider a tree T as shown in the diagram. State how many times the following procedure is invoked recursively and output of the program. **[M-06]**

```
traversal (tree T)
{
    if (T! = NULL) {
        traversal (T–> left) ;
        traversal (T –> right);
        print (T – > data) ;
    }
}
```



14. Prove that for Quick sort **[M-03]**
    (i) Worst case efficiency is $T(N) = O(N^2)$ and (ii) Best_case efficiency is $T(N) = O(N \log N)$
    (Assume a random pivot (no medium–of–three partitioning) and no cut_off for small array).
15. Explain with suitable example the general rules for running time calculations. **[D-03]**
16. (a) An algorithm takes 0.5 ms for input size 100. How long it will take for input size 500 if the running time is the following. (Assume low order terms are negligible) :– **[D-03]**
    (i) Linear (ii) O (N log N) (iii) Quadratic (iv)Cubic
    (b) Explain Radix Sort.
17. An algorithm runs a given input of ~~sine~~ size N. If N is 4096, the run time is 512 mllisec. If N is (8192) ~~16384~~, the run time is 1024 millisecond. What is the complexity ? What is the big – O notation ? **[D-03]**
18. Write a program to sort the given n numbers using Binary tree sort. **[D-07]**

Bubble Sort

Given : Array 'a

Aim : To sort

Workin :

Bubble ort uses t every va ue of i, j eg. If n = 5, then i fron 0 to i.

```
for
for
```

For ach valu a [j + 1]

He e

```
or (i = n
    for (j =
        if (
        {
```

```
}
```

Let us show v

Le 'a' be arr

0	5
1	–
2	6
3	(
4	–

Hence result

0	–
1	
2	
3	–
4	

Hence resul

0	
1	
2	
3	
4	

Bubble Sort

Given : Array 'a' of 'n' integers [0.....n – 1]

Aim : To sort 'a' in ascending order.

Working :

Bubble sort uses two counter i and j such that i begins with (n – 2) and decrements upto 0. For every value of i, j begins with 0 and increments upto i.

eg. If n = 5, then i ranges from (n–2) down to 0 i.e. 3 down to 0 and for each value of i, j varies from 0 to i.

```
for (i = n – 2 ; i >= 0 ; i– –)
for (j = 0 ; j <= i ; j++)
```

For each value of j we check whether (a [j] > a [j + 1]) and if true, we exchange a [j] with a [j + 1]

Hence

```
for (i = n – 2 ; i >= 0 ; i– –)
    for (j = 0 ; j <= i ; j++)
        if (a [j] > a [j +1 ] )
        {
            t = a [j];
            a[j] = a[ j + 1];
            a [j + 1] = t;
        }
```

Let us show working with diagram.

Let 'a' be array of 5 elements i.e. n = 5

0	5
1	–2
2	6
3	0
4	–3

i	j			
3	0	swap	a [0]	a [1]
3	1			no swap
3	2	swap	a [2]	a [3]
3	3	swap	a [3]	a [4]

Hence resultant array after Pass I is

0	–2
1	5
2	0
3	–3
4	6

i	j		
2	0		No swap
2	1	swap	a[1] ↔ a[2]
2	2	swap	a[2], a[3]

Hence resultant array after pass II is

0	–2
1	0
2	–3
3	5
4	6

i	j	
1	0	No swap
1	1	Swap a[1], a[2]

Hence resultant array after pass III is

0	–2
1	–3
2	0
3	5
4	6

i	j	
0	0	swap a [0], a [1]

Hence the resultant array after pass IV is

0	–3
1	–2
2	0
3	5
4	6

Note : Bubble sort requires (n – 1) passes to sort array of 'n' elements.

Program for Bubble sort

```
# include <stdio.h>
void bubble (int a [ ], int n)
{
    int i , j, t ;
    for (i = n – 2 ; i >= 0 ; i – –)
        for (j = 0 ; j <= i ; j++)
            if (a [j] > a [j + 1] )
            {
                t = a [j];
                a [j] = a [j + 1];
                a [j + 1] = t ;
            }
} /* end Bubble */

void main ( )
{
    int a [100], n, i ,
    printf ("Enter n ");
    scanf ("%d", &n);
    for (i = 0 ; i <= n – 1 ; i ++)
    {
        printf ("Enter element % d", i + 1 );
        scanf ("%d", &a [i] );
    }
    bubble (a, n);
    printf ("sorted array is \ n");
    for ( i = 0 ; i <= n – 1 ; i ++ )
        printf ("%d", a [i] );
        printf ( " \n");
} /*end of main * /
```

Selection Sort

This method also requires (n –1) passes for sorting array of n elements. For eg. to sort array of 5 elements, selection sort needs 4 passes.

In pass number 'i', selection sort finds minimum element between a[i] and a[n – 1] (i.e. from position i to n –1) and then interchange the minimum element with a[i]

Let us show worki

Let 'a' be array o

0	5
1	8
2	0
3	2
4	–1

∴ The array afte

–1	8	
0	1	

In pass 1 : We c interchange this

∴ The array aft

–1	0	
0	1	

Similarly all pas

Pass 0

0	5
1	8
2	0
3	2
4	–1

Algorithm

```
// a is array of n
// i acts as pass
i ← 0 to n – 2
{
    // find mini
    min ← a[i]
    p ← i
    j ← i + 1 t
    {
        if (a [j]
        {
            min
            p ←
        }
    }
    interchange
}
```

Program for s

```
void selection
{
    int i, j, mir
    for (i = 0 ;
    {
```

Let us show working of selection sort using diagrams

Let 'a' be arrary of 5 integers

0	5
1	8
2	0
3	2
4	-1

In pass 0
We check minimum element between a [0] to a [4], which is a[4], (= –1)
We interchange this minimum with a [0]

∴ The array after pass 0 is

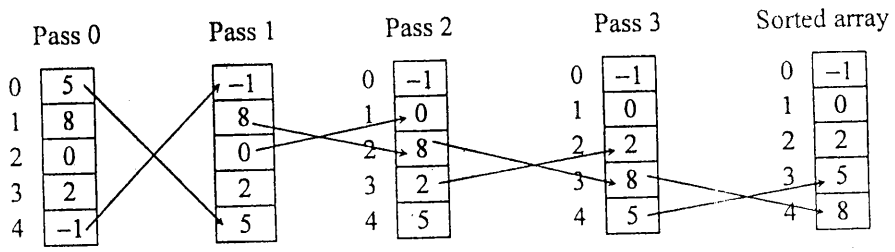
–1	8	0	2	5
0	1	2	3	4

In pass 1 : We check minimum element between a[1] to a[n – 1] which is a[2] (= 0) and we interchange this minimum with a[1]

∴ The array after pass 1 is

–1	0	8	2	5
0	1	2	3	4

Similarly all passes can be shown as follows



Algorithm

```
// a is array of n elements
// i acts as pass number
i ← 0 to n – 2
{
   // find minimum from position i to n –1
   min ← a[i]
   p ← i            // p will give position of minimum
   j ← i + 1  to n –1
   {
      if (a [j] < min )
      {
         min ← a [j]
         p ← j
      }
   }
   interchange (a [i], a [p] )
}
```

Program for selection sort

```
void selection ( int a[ ], int n)
{
    int i, j, min, p, t ;
    for (i = 0 ;  i <= n – 2;  i++)
    {
```

```
        min = a [i] ;
        p = i ;
        for ( j = i + 1; j <= n – 1; j ++)
            if (a [j] < min)
            {
                min = a [j] ;
                p = j ;
            }
            t = a [i] ;
            a [i] = a [p];
            a [p] = t ;
        }
}   // end selection
```

Insertion sort

Insertion sort also requires (n – 1) passes to sort array of n elements.

This method starts with an assumption that the array is divided into sorted partition of length 1 and unsorted partition of length (n – 1)

for eg. if array has 5 integers then sorted and unsorted partitions can be shown as

0	5
1	0
2	–1
3	8
4	2

Sorted partition of 1 element

Unsorted partition of 4 element

In each pass insertion sort picks next element of unsorted partition and inserts it in right place in sorted part. During this process greater elements are pushed down to make a place for element getting inserted.

Let us show the method by diagrams

0	5	0
1	0	5
2	–1	
3	8	
4	2	

In pass 1

We pick a[1] and insert it in the right place in the sorted part. During this process a[0] is pushed down to a[1] Refer the diagram shown.

At the end of pass I the array is

0	0
1	5
2	–1
3	8
4	2

Sorted Partition

similarly all the passes can be shown as follows

Pass I

0	0	0 Sorted
1	5	5 Unsorted
2	–1	
3	8	
4	2	

Pass II

0	0	–1	Sorted
1	5	0	
2	–1	5	Unsorted
3	8		
4	2		

Pass III

–1	
0	Sorted
5	
8	Unsorted
2	

Pass IV

–1	
0	Sorted
5	2
8	5
2	8 Unsorted

Hence the final sorted

–1	0	2
0	1	2

Algorithm for Insertio

```
algorithm insertion ( a
{
   /* aim : to sort arr
   i ← 1 to n – 1
   {
      x ← a[i]   /*
      j ← i      /*
      while (a [j – 1
      {
         a [j] ← a [
         j ← j –1;
      }
      /* insert x at j
      a [j] ← x
   }  /* end of i ←
}  /* finished */
```

Program for Inesrtio

```
void insertion ( int a[
{
   int i, j, x ;
   for (i =1 ; i <= n
   {
      j = i ; x = a [
      while (a [j –
      {
         a [j] = a
         j = j – 1
      }
      a [ j ] = x ;
   } /* end for * /
}
```

Quick Sort (partit

This method work
an algorithm which

eg : Let array be as

50 75 10

Partition process i

We have to fix 50
than 50 , and all el

For fixing 50 at ri
starts with highest

i increases by 1 til
equal to 50.

Hence the final sorted array is

−1	0	2	5	8
0	1	2	3	4

Algorithm for Insertion sort

```
algorithm insertion ( a [ ], n )
{
   /* aim : to sort array a of n elements */
   i ← 1  to  n – 1
   {
      x ← a[i]    / * we will insert x at right */
      j ← i       / * place in sorted partition * /
      while (a [j – 1] > x  and  j > 0)
      {
         a [j] ← a [j – 1];
         j ← j –1;
      }
      /* insert x at j place */
      a [j] ← x
   }  /* end  of i ← 1 to n –1 */
}  /* finished */
```

Program for Inesrtion Sort

```
void insertion ( int a[ ], int n)
{
   int i, j, x ;
   for (i =1 ;  i <= n – 1;  i ++)
   {
      j = i ;  x = a [i] ;
      while (a [j –1] > x  && j > 0)
      {
         a [j] = a [j – 1];
         j = j – 1 ;
      }
      a [ j ] = x ;
   } /* end for * /
}
```

Quick Sort (partition sort)

This method works on Divide & Conquer policy. It partitions the array into 2 partitions using an algorithm which is described below :

eg : Let array be as shown

50	75	10	100	40	200	45	300	15	40

Partition process is as follows :

We have to fix 50 i.e. left most element at such a place that to the left of 50 all elements are less than 50 , and all elements to the right of 50 are more than 50.

For fixing 50 at right place, we use two counters i and j, where i starts with lowest bound and j starts with highest bound.

i increases by 1 till it gets value more than 50 and j decreases by 1 till it gets value less than or equal to 50.

We interchange a [i] and a [j] and repeat this process till (i < j)

50	75	10	100	40	200	45	300	15	400
	i							j	

(i stop at 75 which is more than 50)
(j stops at 15 which is less than or equal to 50)

We interchange a [i] and a [j] as i < j

50	15	10	100	40	200	45	300	75	400
	i							j	

Next i stops at 100 which is more than 50 and j stops at 45 which is less than or equal to 50.

We interchange a[i] and a [j] as i < j

50	15	10	45	40	200	100	300	75	400
			i			j			

Next i stops at 200 which is more than 50

Next j stops at 40 which is less than or equal to 50

We cannot interchange a [i] and a [j]

as i > j

50	15	10	45	40	200	100	300	75	400
				j	i				

Final step : j gives place of partition. Hence 50 will be fixed at j place.

40	15	10	45	| 50 |	200	100	300	75	400

↳ 50 fixed at right place.

Now we will see two functions

Partition and quicksort to sort the array of n elements.

algorithm partition (a [ ] , *l* , r, n) returns integer / * where a is array of n elements */

```
/ * we will partition array a from position ? for */
{
    i ← l
    j ← r
    x ← a [i]      / * we will fix x at such a */
                   / * place that all elements to the */
                   /* left of x will be less than x */
                   /* and all elements to the right of x will be more than x*/

    while (l < r)
    {
        while ( a [i] <= x and i <= r )
            i ← i + 1
        while ( a [j] > x )
            j ← j – 1
        if ( i < j )
        swap a [i] ↔ a [j]
    }.
    /* j is place of partition */

    swap a [l] ↔ a [j]
    return j

} /* end partition */
```

Program for qu

```
# include <stdi
int a[100] ;
int partition (in
{
    int i, j, x, t;
    i = l ; j = r
    while (i < j
    {
        while (
            i ++
        while (
            j –
        if (i < j
        {
            t =
            a [i
            a [j
        }
    } /* end
    /* j gives p
    /* swap a [
    t = a [l];
    a [l] = a [j
    a [j] = t;
    return j;
}

void quick ( i
{
    int p ;
    if (l < r)
    {
        p = pa
        quick
        quick
    } .
} /* end quicl

void main ( )
{
    int i, n ;
    printf ( "
    scanf (" %
    for (i = 0
    {
        printf
        scanf
    }
    quick (0,
    for (i = 0
    print f ("
}
```

Merge sort

Merge sort c
partition cor
After thi

Program for quick sort

```
# include <stdio.h>
int a[100] ;     /* a is global array */
int partition (int a [ ], int l, int r)
{
    int i, j, x, t;
    i = l ; j = r ; x = a [l];
    while (i < j)
    {
        while (a [i] <= x && i <= r)
            i ++ ;
        while (a [j] > x)
            j — ;
        if (i < j)
        {
            t = a [i];
            a [i] = a [j];
            a [j] = t ;
        }
    } /* end while */
    /* j gives place of partition */
    /* swap a [j] with a [l] */
    t = a [l];
    a [l] = a [j];
    a [j] = t;
    return j;     /* return place of partition */
}

void quick ( int l, int r )
{
    int p ;
    if (l < r)
    {
        p = partition (l, r) ;
        quick (l, p – 1 );
        quick ( p + 1, r );
    } .
} /* end quick */

void main ( )
{
    int i, n ;
    printf ( "Enter n")
    scanf (" % d", & n);
    for (i = 0 ; i <= n–1 ; i++)
    {
        printf ("Enter element % d", i + 1 ) ;
        scanf ( "%d", & a [i] ) ;
    }
    quick (0, n – 1) ;
    for (i = 0; i <= n–1 ; i ++ )
    print f ("%d", a [ i ] );
}
```

Merge sort

Merge sort divides the array into 2 equal partitions and continues the process of division till each partition contains only one element.

After this, the neighbouring partitions are merged and sorted to get final sorted array.

Taking an example of array a of 8 elements

1) 100 50 22 2 88 3 33 5
   0 7
   Find mid point of array
   mid point = (0 + 7) / 2 = 3

2) Two partitions are from a [o] to a [3] and a[4] to a [7]
   100 50 22 2 88 3 33 5
   0 3 4 7
   (left partition) (right partition)

3) Division process is repeated again.

Radix Sort

Program

```
# define m 3
# include <stdio.h>
void radix (int a [ ], int n)
{
    struct node
    {
        int data;
        struct node * next;
    };
    struct node * front [10] , * rear [10];
    struct node * start, * prev , *p;
    int i, j, y, l, k, p1, exp;
    start = NULL;
    for (i = 0; i <= n - 1 ; i ++)
    {
        p = (struct node * ) malloc (sizeof (struct node));
        p → data = a [i];
        p → next = NULL;
        if (start = = NULL)
            start = p;
        else
            prev → next = p;
        prev = p;
    }       /* end of for loop */
    for (i = 1; i<= m ; i ++)
    {
        for (j = 0; j <= 9; j++)
        {
            front [j] = rear [j] = NULL;
        }
        exp = 1;
        for (k = 1; k <= i -1; k ++)
            exp = exp * 10 ;
            p = start ;
            while (p != NULL )
            {
                l = p → data;
                y = (l / exp ) % 10 ;
                if (front [y] = = NULL)
```

```
                }
                r
            }       /*
            p = start
            for (i = (
            {
                a[i] =
                p = l
            }
        }  /* end o

        void main (
        {
            int a [1(
            printf ("
            scanf ("
            for (i =
            {
                prin
                sca
            }
            radix (
            for (i =
                pri
        }  /* end
```

Working of F

901 → 622 →

901 → 622 →
f[1]
r[1]

901 → 622 –
f[1] f[2]
r[1] r[2]

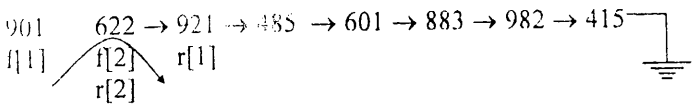
```
            front [y] = p ;
        else
            rear [y] → next = p ;
            rear [y]  = p ;
            p = p → next ;
        }
        start = NULL ;
        for (k = 0 ;  k <= 9 ;  k ++)
        {
            if (front [k] != NULL )
            {
                if (start = = NULL)
                    start = front [k];
                else
                    rear [p1] → next = front [k];
                    p1 = k ;
            }
        }
        rear [p1] → next = NULL;
    }       /* end of for loop */.
    p = start ;
    for (i = 0 ;  i <= n – 1 ;  i ++)
    {
        a[i] = p → data;
        p = p → next;
    }
} /* end of radix */

void main ( )
{
    int a [100], n, i ;
    printf ("Enter n");
    scanf ("%d", &n );
    for (i = 0 ;  i <= n – 1 ;  i ++)
    {
        printf ("Enter element %d", i + 1);
        scanf ("%d", & a [i] );
    }
    radix (a, n );
    for (i = 0 ;  i <= n – 1;  i++)
        printf ("%d", a [i] );
} /* end of main */
```

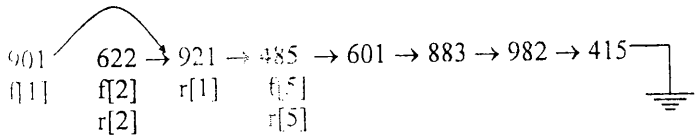
Working of Radix Sort (First Pass)

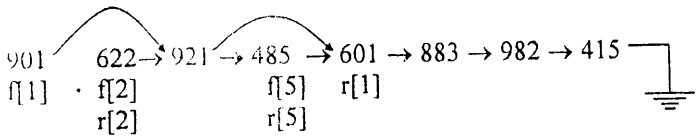
901 → 622 → 921 → 485 → 601 → 883 → 982 → 415

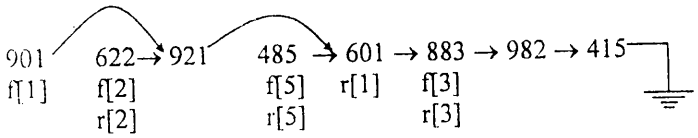
901 → 622 → 921 → 485 → 601 → 883 → 982 → 415
f[1]
r[1]

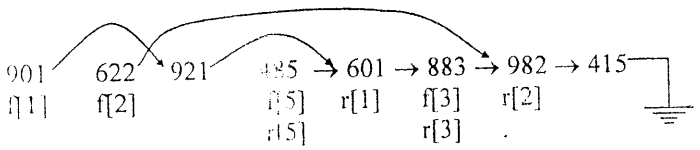
901 → 622 → 921 → 485 → 601 → 883 → 982 → 415
f[1] f[2]
r[1] r[2]

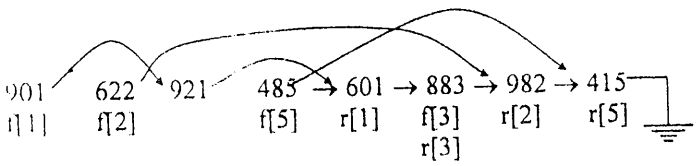


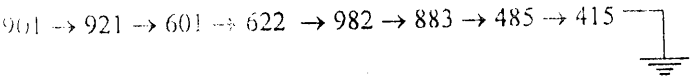




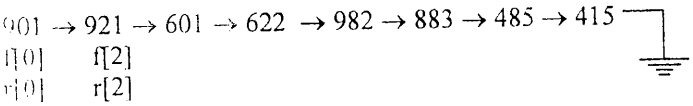


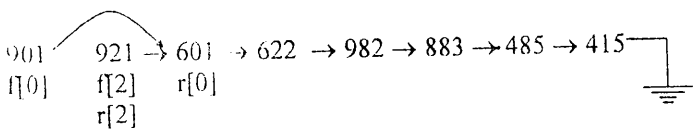


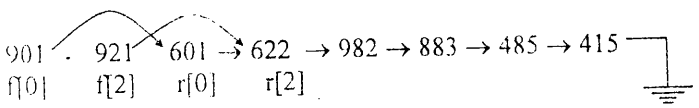


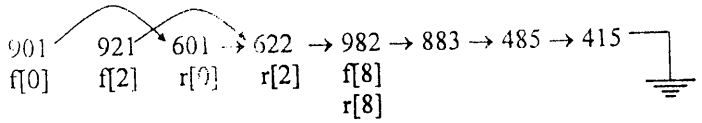


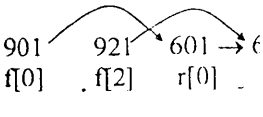
Second Pass

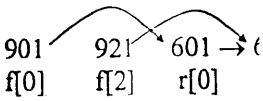


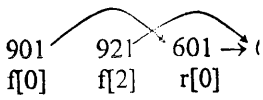




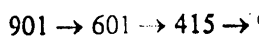


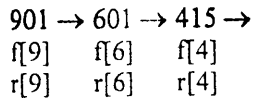


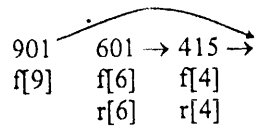


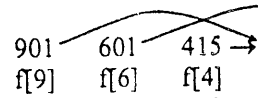


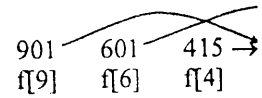
Third Pass

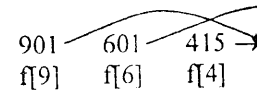


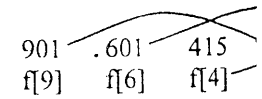












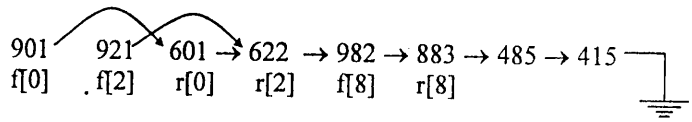
Final link list

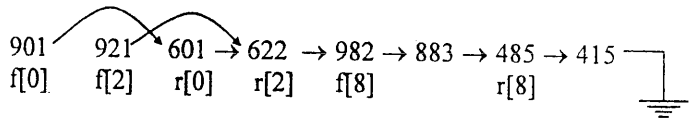
415 → 485 → 601 →

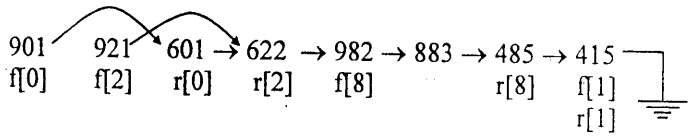
Shell Sort

It is also known as d

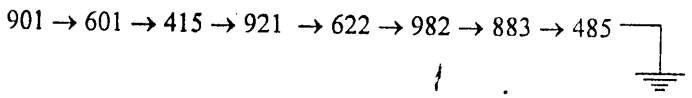
```
Void Shell (int a[ ], i
{
    int h, i, j, x ;
    h = n/2 ;
    while (h ! = 0)
    {
```

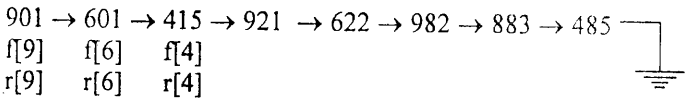


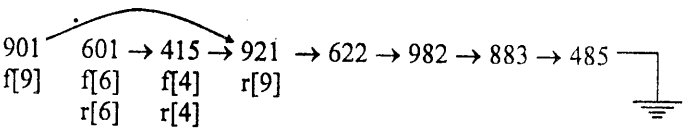


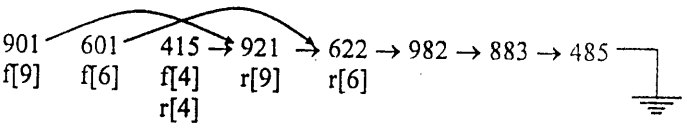


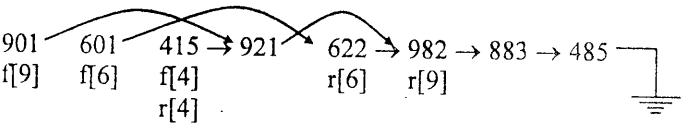
Third Pass

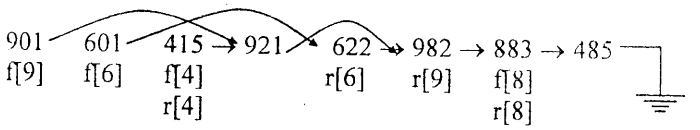


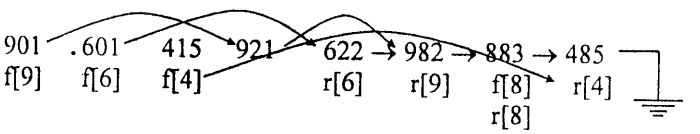




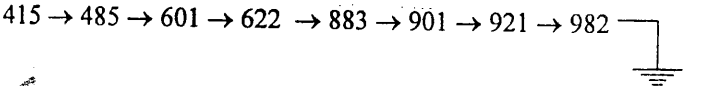








Final link list



Shell Sort

It is also known as diminishing increment sort. It is modification of insertion sort.

```
Void Shell (int a[ ], int n)
{
    int h, i, j, x ;
    h = n/2 ;
    while (h ! = 0)
    {
```

```
        for (i = h ; i <= n – 1 ; i++)
        {
            j = i ; x = a[i];
            while (a[j – h] > x && j > h – 1)
            {
                a[j] = a[j – h] ;
                j = j – h :
            }
            a[j] = x ;
        }       /* end of for loop */
        h = h/2 ;
    }       /* end of while loop */
}       /* end of function */
```

Shell sort works on the technique of comparing the elements of farther distance (h). Insertion sort compares elements which are at distance 1 from each other where as shell sort compares elements which are at distance h from each other.

As we see in the while loop

```
while (a[j – h] > x  && j >  h – 1)
{
    a[j] = a[j – h] ;
    j = j – h ;
}
```

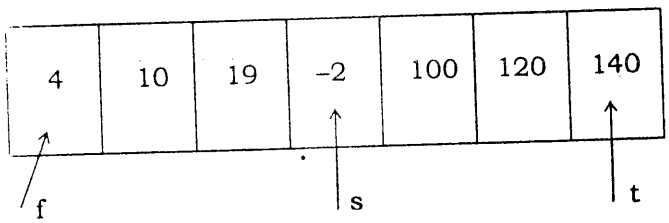
Here element a[j – h] which is h apart from a[j] is compared.
The increment 'h' goes on decreasing due to the expression

```
h = h/2 ;
```

finally h becomes 1 and during this value of h = 1, nothing but insertion sort runs.

Merge Sort Technique

The idea of merge sort comes from the case of merging two subarrays which are already sorted in increasing order. For e.g. :



The array A can be considered as two subarrays ranging from f to s – 1 and s to t. Both these subarrays are already sorted.

These two subarrays can be merge into a single array say 'C' such that C contains all the element of array A in increasing order.

```
#include <stdio.h>

void merge(int left, int right);
void simplemerge(int f, int s, int t);
int a[100];
int n;
```

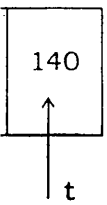
```
void main( )

{

    int i;
    printf("Ente
    scanf("%d"
    for (i = 0; i
    scanf("%d"
    merge(0, n
    printf("Sort
    for (i = 0; i
    printf("%d

}

void merge(int

{

    int s, m;
    s = right - l
    if (s >= 1)

    {

        m = (le
        merge(l
        merge(r
        simpler

    } .
}

void simpleme

{

    int i, j, k, w
    i = f; j = s;
    while (i <=

    {
        if(a[i] <
        temp[+

    else

        temp[+
    }

    /* copy ren

    for (w = i;
    temp[++k]
    for (w = j;
    temp[++k]
```

```
void main( )

{

    int i;
    printf("Enter n ");
    scanf("%d", &n);
    for (i = 0; i <= n - 1; i++)
    scanf("%d", &a[i]);
    merge(0, n - 1);
    printf("Sorted array \n");
    for (i = 0;  i <= n – 1 ;  i++)
    printf("%d ", a[i]);

}

void merge(int left, int right)

{

    int s, m;
    s = right - left;
    if (s >= 1)

    {

        m = (left + right) / 2;
        merge(left, m);
        merge(m + 1, right);
        simplemerge(left, m + 1, right);

    } .
}

void simplemerge(int f, int s, int t)

{

    int i, j, k, w, temp[100];
    i = f; j = s; k = -1;
    while (i <=  s - 1 && j <= t)

    {
        if(a[i] < a[j])
        temp[++k] = a[i++];

    else

        temp[++k] = a[j++];
    }

    /* copy remaining elements in temp array */

    for (w = i; w <= s - 1; w++)
    temp[++k] = a[w];
    for (w = j; w <= t; w++)
    temp[++k] = a[w];
```

her distance (h). Insertion
ıere as shell sort compares

n sort runs.

s which are already sorted



– 1 and s to t. Both these

t C contains all the element

```
/*copy elements of temp back in array a */

for (w = 0; w<= k; w++)
a[f + w] = temp[w];
}  /*end of function simplemerge */
```

We may extend the idea of two subarrays to 'n' subarrays such that each subarrays is sorted in increasing order.

For e.g., consider an array of size n = 100. Which can be divided into 2 subarrays of 25 elements each. These 2 subarrays can again be divided in four subarrays of 12 and 13 elements each. This division can go as till each subarray has only one element. That is for original array having n elements, n subarrays can be formed (each sub array has exactly one element) and then these arrays can be merged using simple merge technique.

As an example consider the array

(10, 9, 8, 7, 6, 5, 4, 3, 2, 1)

Step 1 : 10, 9, 8, 7, 6, 5, 4, 3, 2, 1

mid point is at position 5, so that left and right subarrays consist 5 elements (position 1 to 5) and right subarray consists 5 elements.

Step 2 : Left subarray is divided into 2 equal parts.

10, 9, 8, 7, 6, 5, 4, 3, 2, 1

10 9 8 | 7 6

mid point

10 9 8 ← simple merge will be called

mid point

X X → division is not possible.

At this point when division of subarrays is not possible, the simple merge procedure is called for (10, 9, 8) to sort it as (8, 9, 10).

This process goes on till all subarrays have been sorted.

Analysis of Sorting techniques

1. Insertion Sort :
Worst case : If the point is in the reverse order insertion sort is $o(n^2)$. If the input is in the reverse order test in the loop to push down the heavier element is performed 2 times when i is 2, 3 times when i is 3. Precisely test repeats i times (for every value of i(i ranges from 2 to n).

$$\therefore \sum_{i=2}^{N} i = 2 + 3 + ... + N \approx o(N^2)$$

∴ when inp

n

i = 2 →

n

i = 3 →

In the best case
down heavier ele

2. Quick so
Consider a fi
Assume that
there will be
is divided in
These 2 subl
require n/4 c

After sub di
comparisons

n + 2

= n

= 0

Thus in the b

Worst case o
In the best ca
the middle of

However if o
proper positi
continues n –

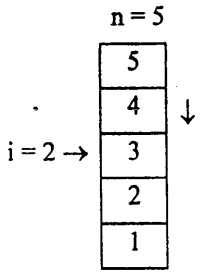
Therefore tot

n + (n

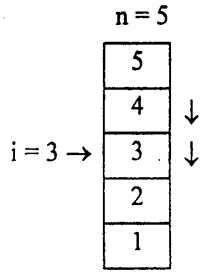
approx.

In worst case

∴ when input is in reverse order insertion sort is $o(N^2)$.



when i = 2, number of comparisons to push down 5 and to place 4 is 2 comparisons.



when i = 3 number of comparisons to push down 4 and 5 and to place 3 is 3 comparisons.

In the best case (i.e. if input is already sorted) the running time is o(N) because the test to push down heavier element always fails.

2. Quick sort :
Consider a file having size n which is power of 2 such that $n = 2^m$ (or $m = \log_2 n$).
Assume that proper position of pivot is always found at the middle of subarray. In this case there will be approximately n comparisons (actually n – 1) on the first half. After which file is divided into 2 subfiles of size n/2 for each of these files n/2 comparisons are required. These 2 subfiles of n/2 size will again decompose to four files of n/4 size each of these require n/4 comparisons.

After sub dividing the file m times gives n files of size 1 each. Thus total number of comparisons is approximately

$$n + 2(n/2) + 4(n/4) + \ldots\ldots + n(n/n)$$

$$= n + n + n + \ldots\ldots + n \text{ (m terms)}$$

$$= 0(nm) \text{ or } 0(n \log_2 n) \text{ because } m = \log_2 n.$$

Thus in the best case (i.e., when pivot is found exactly at the midpoint) the time is $o(n \log^2 n)$.

Worst case of Quick sort :
In the best case we had considered that the pivot value is x[1b] and it finds proper position at the middle of the array.

However if original array is already sorted above condition is false. If for e.g. x[1b] is in its proper position, the original file is split into 2 subfiles of ϕ and n – 1 sizes. This process continues n – 1 times. First of size n, then second of size n – 1, third n – 2 and so on.

Therefore total of

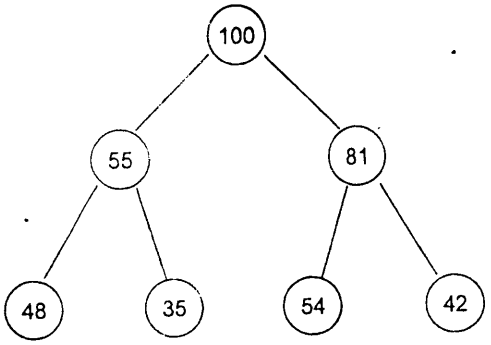
$$n + (n-1) + (n-2) + \ldots\ldots + 2$$

approx. = $o(n^2)$.

In worst case quick sort is $o(N^2)$.

Heap sort analysis :

Heat sort procedure uses a almost complete binary tree to be treated as heap. Such that every parent has a key higher than its children. If n is total number of nodes, the depth of the tree will be $\log_2 n$, for e.g. if n = 7 then $\log_2 n = \log_2 7$ approx. = 3.



$\rightarrow$ n = 7

$\therefore$ depth $\approx \log_2 n \approx 3$.

while adjusting heap during reheapity procedure, if each element in the array gets filtered upto leaf, we will still have at most n log n comparisons.

Thus heat sort is o(n $\log_2 n$) even in the worst case.

Merge Sort Analysis :

For any array of size n (where n is power of 2 i.e. $n = 2^m$ or $m = \log^2 2$) merge sort divides the array in two halves (of size n/2 each) and then the 2 sub–arrays of size n/2 are merged.

The 2 sub arrays of size n/2 are again divided in 4 equal size arrays of n/4 elements and then merged.

This process would go on till n sub–arrays of size 1 each are created. Therefore algorithm time can be calculated as

$$n + 2(n/2) + 4(n/4) + \ldots\ldots + n(n/n)$$

$$= n + n + n + \ldots\ldots\ldots + n \quad \text{(m terms)}$$

$$= n\,m = n \log^n 2$$

Thus merge sort also has running time 0(n $\log^n 2$).

Graded Questions :

1. Explain Bubble sort Algorithm with example.
2. Develop a bubble sort algorithm such that alternate passes go in opposite direction i.e., during first pass the record with largest key will be at the end of the table and during the second pass record with smallest key will be the first record in the table.
3. Explain Quick Sort algorithm, with example calculate its Best, Worst, average case complexity using mathematical derivation. **[N-04]**
4. Write a program to sort given n numbers using 'Quick sort". Derive the complex of Quick sort. **[D-07]**
5. Successive passes Quick sort place first element in proper position. Discuss efficiency of this sorting technique. Modify and write C code for Quick sort to give better performance when list in sorted in required order. **[M-04]**
6. Explain Quick Sort algorithm. Sort following nos. using Quick sort. Show output of each pass : **[M-04]**

   65 70 75 80 85 60 55 50 45
7. Show how the Sorting of the given list is done using insertion sort by every pass **[M-06]**

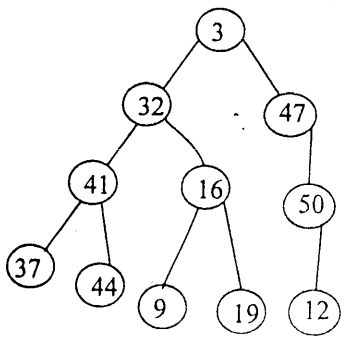
   567 456 455 324 213 665 777 907 990 654

   Give the time and space complexity for every case for insertion sort.
8. Sort the following numbers using insertion sort. Show the output of each pass. **[M-06]**

   23, 78, 45, 08, 32, 56, 18
9. Design an algorithm for sorting an array by diminishing increment.

10. Desi[illegible] and in
    num[illegible]ers.
11. Expl[illegible]in why t
12. Expla[illegible]n with s
13. Explain Binar
14. Deve[illegible]op an al
    unsorted file o
15. Expla[illegible]n the co
16. Defin[illegible] Heap.
17. Assume n = 5
    (i) [illegible]r inserti
    (ii) What will
    s[illegible]quence.
18. Show that the
    size [illegible] array.
19. Expl[illegible]in shell
    complexity us
20. Write short no
21. Deve[illegible]op Radi
    numbers in as
    418, [illegible]23, 238,
22. Compare the
    when : (i) Th
23. Give two arra
    program to ge
24. Explain heap
25. Sort following
    5,
    Show output a
26. Explain merg
    of merge sort
27. What is the c
    Big Oh compl
28. Explain hepif
    32, 45, 8, 23,
29. Prove that, t
    O (N^2).
30. Prove that the
31. Write short nc
32. What is Topol
33. Explain the ba
34. Give Comple
35. Given a set o
    'shell sort' me
36. Explain Interr
37. Show the step

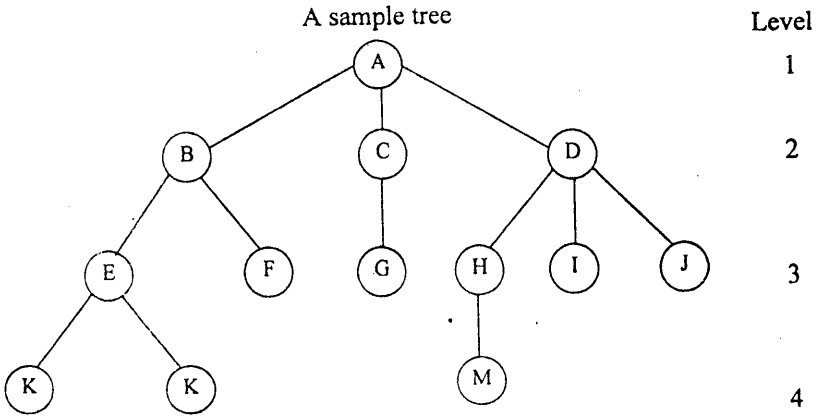
10. Design and implement the selection sort algorithm to sort a given set of randomly ordered n numbers.
11. Explain why the straight selection sort is more efficient than the bubble sort.
12. Explain with suitable example Binary tree sort. [N-06]
13. Explain Binary Tree Algorithm with example.
14. Develop an algorithm using a heap of k elements to find the largest k numbers in a large unsorted file of n numbers.
15. Explain the concept of heap as a Priority Queue.
16. Define Heap. Write a C program to implement Heap sort. [M-04, N-04]
17. Assume n = 5 and input sequence is (8, 5, 9, 3, 2) :
    (i) For insertion sort what will be the intermediate output after each insertion.
    (ii) What will be the output after each iteration if the sort is employed on the same input sequence.
18. Show that the worst case running of insertion sort is a quadratic function of 'n', where 'n' is size of array.
19. Explain shell sort algorithm, with example calculate its best, worst, average case complexity using mathematical derivation. [N-06]
20. Write short note on Merge Sort. [N-04]
21. Develop Radix sort algorithm and by using Radix sort algorithm sort the following set of numbers in ascending order –
    418, 123, 238, 181, 620, 910, 515, 319, 710, show the result of each pass.
22. Compare the complexities of various sorting methods, which methods are more suitable when : (i) The number of elements are small (ii) The number of elements are large.
23. Give two arrays A sorted in ascending order and B sorted in descending order. Write a program to generate an array containing all elements of A as well as B in ascending order.
24. Explain heap sort. Show how heap sort processes the input 31, 41, 59, 26, 53, 58, 97. [M-03]
25. Sort following nos using Heap sort ? [N-04]
    5, 2, 10, 4, 3, 2, 9, 7, 8, 6.
    Show output after each pass.
26. Explain merge sort with suitable example. Also show that the worst case running time of merge sort is O (N log N). [D-03]
27. What is the complexity of Merge sort. Explain it using mathematical derivation (i.e. find Big Oh complexity). [M-04]
28. Explain hepify property of the heap sort. Show how heap sort process the input 78, 56, 32, 45, 8, 23, 19. [D-03]
29. Prove that, the worst case running time of Shell sort, using Shell's increment is $O(N^2)$. [D-03]
30. Prove that the Best case efficiency of Quick sort is : $T(N) = O(N \log N)$
31. Write short note on Topological sorting. [M-03, D-03]
32. What is Topological sort ? Explain it with example. [M-05, N-04, 05]
33. Explain the basic philosophy of 'Quicksort' and develop an algorithm for it. [M-05, N-05]
34. Give Complexity Analysis of 'Quicksort'. [M-05, N-05]
35. Given a set of 'n' numbers, with a program to sort them into non–descending order using 'shell sort' method. Explain the logic clearly taking an example. [M-05]
36. Explain Internal and External sort with example. [N-05]
37. Show the stepwise method of constructing a heep, given the initial binary tree in figure. [N-05]



❑❑❑❑❑❑

Binary Search Method

A tree is a finite set of one or more nodes such that : (i) there is a specially designated node called the root, (ii) the remaining nodes are partitioned into $n \geq 0$ disjoint sets $T_1, \ldots \ldots T_n$ where each of these sets is a tree. $T_1, T_2, \ldots\ldots, T_n$ are called the subtrees of the root.



A node stands for the item of information plus the branches to other items. Consider the above tree. This tree has 13 nodes. The root is A and we will normally draw trees with the root at the top.

Degree : The number of subtrees of node is called its degree. The degree of A is 3, of C is 1 and of F is zero.

Leaf Nodes : Nodes that have degree zero are called leaf or terminal nodes. {K, L, F, G, M, I, J} is the set of leaf nodes in the given tree. Other nodes are known as non terminals.

The level of a node is defined by initially letting the roof be at level one. If a node is at level l, then its children are at level $l + 1$.

The height or depth of a tree is defined to be the maximum level of any node in the tree.

A forest is a set of $n \geq 0$ disjoint trees. The notion of a forest is very close to that of a tree because if we remove the root of a tree we get a forest. Example : from the above tree if we are removing A we get a forest with three trees.

List representation of trees : Trees can be represented in list forms also. List representation of the given tree is given below :

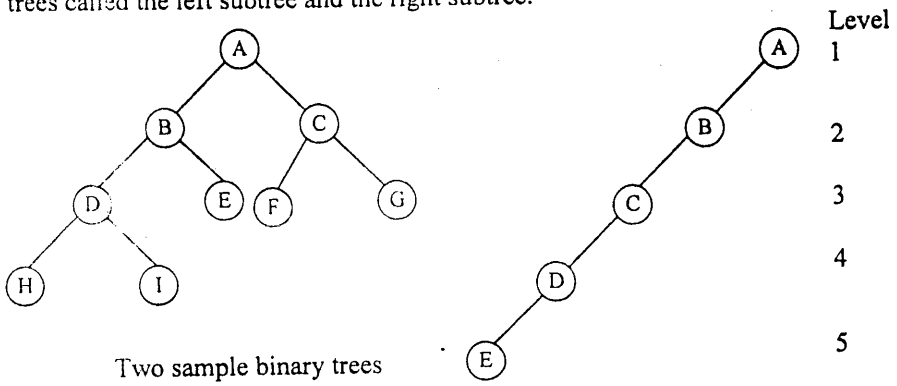
(A (B (E (K, L), F), ((G), D(H (M, I, J)))

The information in the root node comes first followed by a list of the subtrees of that node.

Binary Trees

Definition :

A binary tree is a finite set of nodes which is either empty or consists of a root and two disjoint binary trees called the left subtree and the right subtree.



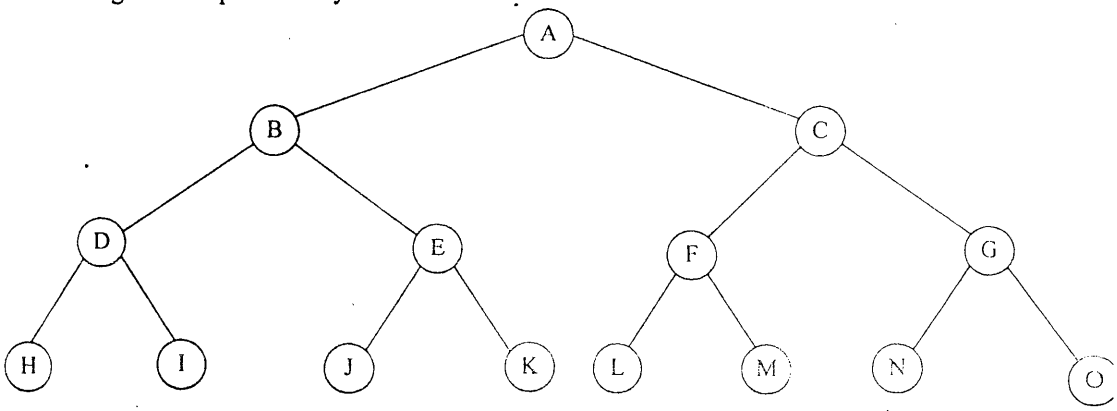
Two sample binary trees

The maximum number of nodes on level i of a binary tree is 2^{i-1}, $i \geq 1$ and
The maximum number of nodes in a binary tree of depth k is $2^k - 1$, $k \geq 1$.

Binary tree representations :
A full binary tree of depth k is a binary tree of depth k having $2^k - 1$ nodes.

Strictly binary tree :
If every non leaf node in a binary tree has non empty left and right subtrees, then the tree is termed as a strictly binary tree.

Complete binary tree :
A complete binary tree of depth d is the strictly binary tree, all of whose leaves are at level d. Following is a complete binary tree.



Binary tree can be represented in several ways in the memory of the computer. Normally we are using sequential (array) representation and linked representation.

Array Representation
A very elegant sequential representation of a binary tree result from sequentially numbering the nodes, starting with nodes on level 1, then those on level 2 and so on. Nodes on any level are numbered from left to right (as shown in fig.a). So, fig.(a) tree can be numbered (as shown in fig. b) follows :

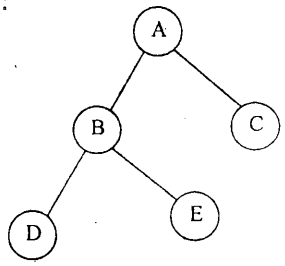


Fig. a

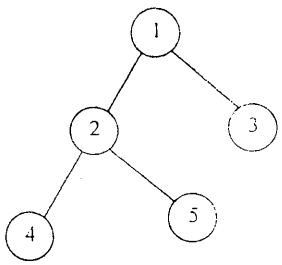


Fig. b

The nodes now may be stored in a one dimensional array tree, with the node numbered i being stored in tree [i]. So, the array representation of the given binary tree is as follows :

tree = | A | B | C | D | E |

Consider the following binary tree (Fig. a). When we are numbering the nodes, we will get the following binary tree (fig. b) below.

Fig. (a)

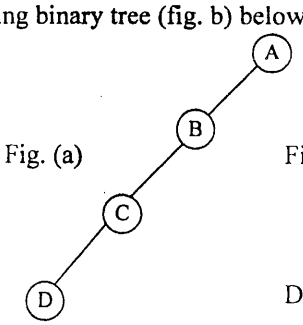
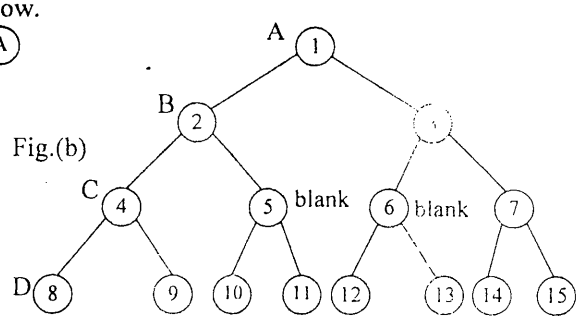


Fig.(b)



So, array representation is :

1	2	3	4	5	6	7	8	9	10	11	
A	B		C				D				

Above representation appears to be good for complete binary trees. It is wasteful for many other binary trees as we have already seen.

Moreover insertion and deletion of nodes from the middle of a tree requires the movement of potentially many nodes to reflect the change in level number of these nodes.

Note :
If a complete binary tree with n nodes is represented sequentially, then for any node with index i, $1 \leq i \leq n$ we have :

1) parent (i) is at i div 2 if $i \neq 1$, when i = 1, then i is the root and has no parent.
2) leftchild (i) is at 2i if $2i \leq n$ if $2i > n$ then i has no left child.
3) Rightchild (i) is at 2i + 1 if $2i + 1 \leq n$. If $2i + 1 \leq n$. If $2i + 1 > n$, then i has no right child.

Linked Representation
In this representation each node will have 3 fields, leftchild, data and rightchild and can be defined in Pascal as :

```
struct tree record
{
struct tree record *leftchild;
char data ;
struct tree record * right child ;
} ;
```

we shall draw such a node using either of the representations below :

leftchild	data	rightchild

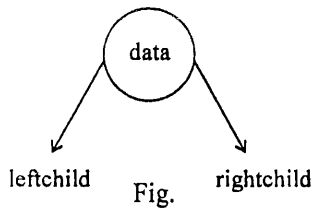


Fig.

Binary tree traversal
Traversing a binary means visiting each node in the binary tree exactly once. A full traversal produces a linear order for the information in a tree. If we let L, D, R stand for moving of traversal. LDR, LRD, DLR, DRL, RDL, and RLD. If we adopt the convention that we traverse left before right then only three traversals remain : LDR, LRD and DLR. To there we assign the names inorder, postorder, and preorder because there is a natural correspondence between these traversals and producing the infix, postfix and prefix forms of an expression.

Following program creates Binary tree and then traverses the tree in Inorder, Preorder and Postorder traversal methods.

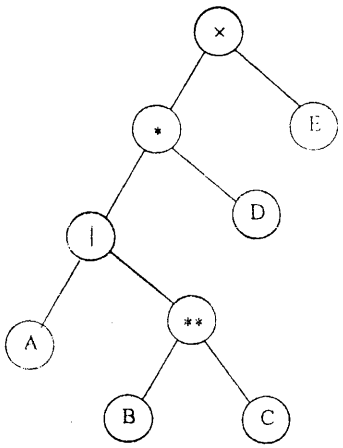
```
# include <stdio.h>
# include <stdlib.h>

struct node
{
    int data;
    struct emp *left, *right;
};

struct node *old, *root, *newrec;

void makeroot ( )
{
    root =  (struct node *) malloc (sizeof (struct node));
    printf ("Enter data ");
    scanf ("%d", &root->data);
```

```
    root->left = r
}

void create (int n
{
    int i;
    struct node *r

    for(i = 2; i <=
    {
        newrec =
        printf ("Er
        scanf ("%
        newrec->l
        prev = nex

        while (nex
        {
            prev =
            if (nex
            next =
            else
                nex
        }
        if (prev->d
            prev->l
        else
            prev->r
    }
}

void inorder (struc
{
    if (p != NULL)
    {
        inorder (p->
        printf ("%d
        inorder (p->
    }
}

void preorder (stru
{
    if (p != NULL)
    {
        printf ("%d
        preorder (p-
        preorder (p-
    }
}

void postorder (stru
{
    if (p != NULL)
    {
        postorder (p
        postorder (p
        printf ("%d
    }
}
```

```
    root->left = root->right = NULL;
}

void create (int n)
{
    int i;
    struct node *next, *prev;

    for (i = 2; i <= n; i++)
    {
        newrec = (struct node *) malloc (sizeof (struct node));
        printf ("Enter data ");
        scanf ("%d", &newrec->data);
        newrec->left = newrec->right = NULL;
        prev = next = root;

        while (next != NULL)
        {
            prev = next;
            if (next->data > newrec->data)
            next = next->left;
            else
                next = next->right;
        }
        if (prev->data > newrec->data)
            prev->left = newrec;
        else
            prev->right = newrec;
    }
}

void inorder (struct node *p)
{
    if (p != NULL)
    {
        inorder (p->left);
        printf ("%d ", p->data);
        inorder (p->right);
    }
}

void preorder (struct node *p)
{
    if (p != NULL)
    {
        printf ("%d ", p->data);
        preorder (p->left);
        preorder (p->right);
    }
}

void postorder (struct node *p)
{
    if (p != NULL)
    {
        postorder (p->left);
        postorder (p->right);
        printf ("%d ", p->data);
    }
}
```

```
void main( )
{
    int n;
    printf ("No of nodes ");
    scanf ("%d", &n);

    makeroot ( );
    create (n);

    printf ("\n");
    inorder (root);
    printf ("\n");
    preorder (root);
    printf ("\n");
    postorder (root);
}
```

Consider the following binary tree :



When we are using inorder algorithm of traverse the given binary tree output will be as follows :

A / B ** C * D + E

The output produced by preorder is

+ * / A * * B C D E

and the output produced by post order is

A B C * * / D * E +

We have written all the above algorithms using recursion. The same thing can be written without recursion also.

Binary Search Tree

Definition : A binary search tree T is a binary tree ; either it is empty or each node in the tree contains a key and

1) All keys in the left subtree of T are less than the key in the root node T .
2) All keys in the right subtree of T are greater than or equal to the key in the root node T.
3) The left and right subtrees are also binary search trees.

* Design and implement an algorithm to search an ordered binary for a given alphabetic key or name.
[ordered binary tree – binary search tree].

Algorithm

1. Establish the search key and the pointer to the root of the tree.
2. Set the not found state and make the current node pointer point to the root and the previous node pointer to nil.

3. While search k
   a) If search ke
      a.1) Sav
      a.2) Adj
      2^1.a) adju
      a^1.1) mak
4. Establish whet
5. Return pointers
   not the search v

Binary Tree Inse

Algorithm develo
The basic steps req
1. Find where new
2. Create a new n
   right pointers w
3. Adjust the prev
Now we can write t

Algorithm
1. Establish the na
2. Find where new
3. Create a new n
   node will have
4. If tree is not init
   a) if tree name
      a.1) adju
      a^1) adju
      a^1) adju
5. Return the upda

B–Tree and B+T

B–Tree : In binary
B–tree is a multiwa
contains at least n
each node in the tre
order 5.

Following diagrams
full (i.e., has 4 key
median of the keys i

Initial B–tree :

Thus inserting 35 in

3. While search key not found and there is still a valid path to follow in the tree do
   a) If search key is not found at the current-node then
      a.1) Save pointer to current node in previous node pointer,
      a.2) Adjust current node pointer to point to the root of left subtree else.
      2^1.a) adjust current node pointer to point to the root of the right subtree else.
      a^1.1) make the not found condition false to signal termination of the search.
4. Establish whether or not search key found.
5. Return pointers to the previous node, the current node, and a variable indicating whether or not the search was successful.

Binary Tree Insertion

Algorithm development

The basic steps requires to insert a new node in a binary search tree.

1. Find where new name is to be placed in tree by searching for it.
2. Create a new node and store in it the name to be inserted. Since it has no subtrees its left and right pointers will be nil.
3. Adjust the previous node's pointer so that it points to the node to be inserted.

Now we can write the algorithm formally as follows :

Algorithm

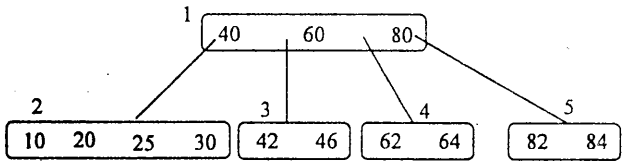
1. Establish the name to be inserted and the pointer to the root of the tree.
2. Find where new name is to be placed in tree using tree search algorithm.
3. Create a new node and store in it the name to be inserted. At the time of insertion this new node will have no subtrees so its left and right pointers must be set to nil.
4. If tree is not initially empty then
   a) if tree name comes alphabetically later than new name then
      a.1) adjust previous node's left pointer so that it points to the new node else.
      a^1) adjust previous node's right pointer so that it points to the new node, else
      a^1) adjust root pointer to point to new node.
5. Return the updated tree and its root pointer.

B–Tree and B+Tree

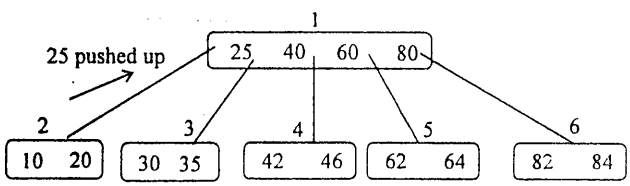
B–Tree : In binary search tree each node contains a single key and points to two subtrees. A B–tree is a multiway search tree. A B–tree of order n is a tree in which each non root node contains at least n div 2 keys. A B–tree of order n is also called n – (n – 1) tree. This means each node in the tree has a maximum of (n – 1) keys and n sons. Thus 4–5 tree is a B–tree of order 5.

Following diagrams show the insertion process in a B–tree of order 5. When a node becomes full (i.e., has 4 keys) and an attempt is made to insert additional key, a split occurs such that median of the keys is sent up to the parent.

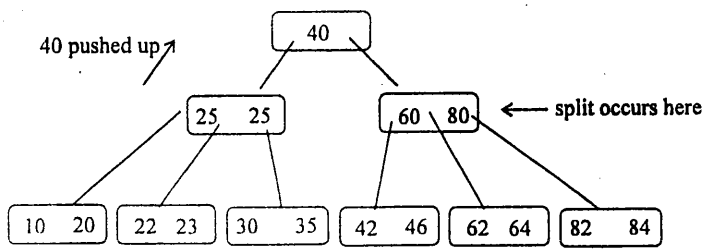
Initial B–tree :



Thus inserting 35 in the tree results splitting of node 2.

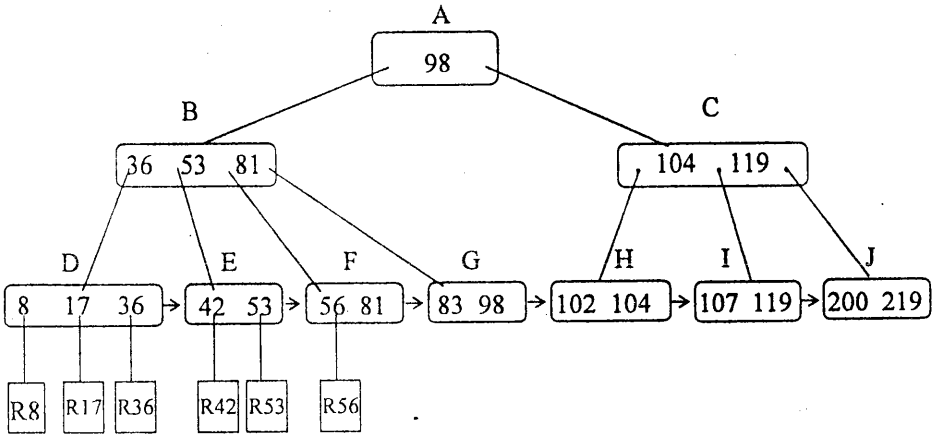


Also inserting 21, 22 will be straight forward. But on insertion of a Key = 23 will again split node 2, which will inturn split node 1 to push 40.



Note : All the leaves are at same level in a B–tree. Hence it is called Balance–tree.

B + tree : A major disadvantage of B–tree is the difficulty of traversing the keys sequentially. A variation to B–tree is B+ tree in which all the keys are maintained in the leafs and keys are replicated in the non–leaf nodes to define paths for locating individual records. The leafs are linked together to provide a sequential path for traversing the keys in the tree.



To locate a record associated with key = 53 (random access), the key is first compared with 98. As key is less than 98, left subtree is visited. 53 is then compared with 36 and than with 53 in node B. Since it is less than or equal to 53 proceed to node E where actual record can be found from there on the tree can also be read sequentially. Since record for 53 is found in node E which has a link to node F with keys 56, 81 and so on.

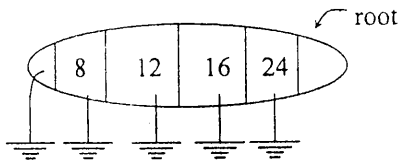
Thus B+ tree can search a key randomly and then traverse the tree sequentially.

Construction of B–Tree having order 4

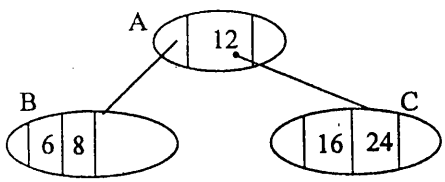
Consider the following key set :

12 8 16 24 6 18 28 100 15 49 68 20 22 80 82 85 88

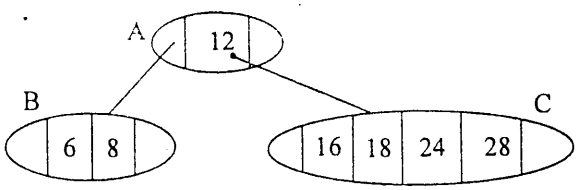
(1)



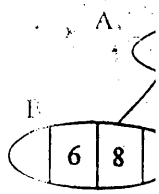
(2) Adding 6 will split the root



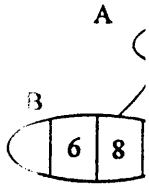
(3) 18, 28 are added in node C.



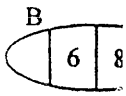
(4) Adding 100 w



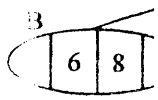
(5) 15 goes in C a



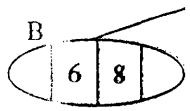
(6) Adding 22 in



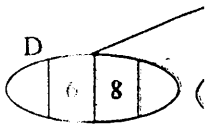
(7) Adding 80 wil



(8) Adding 82 &



(9) 88 will cause



a Key = 23 will again split

split occurs here

82 84

Balance–tree.

ing the keys sequentially. A
d in the leafs and keys are
dual records. The leafs are
the tree.

C

04 119

I J

107 119 200 219

y is first compared with 98.
with 36 and than with 53 in
e actual record can be found
l for 53 is found in node E

quentially.

22 80 82 85 88

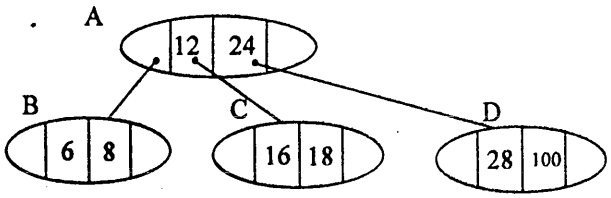
l split the root

2

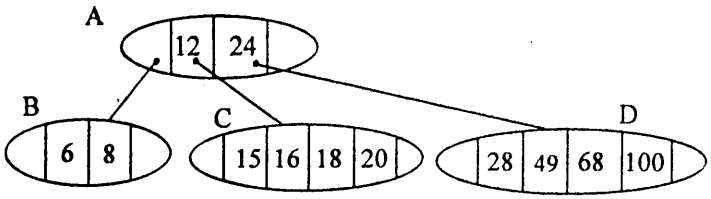
C

16 24

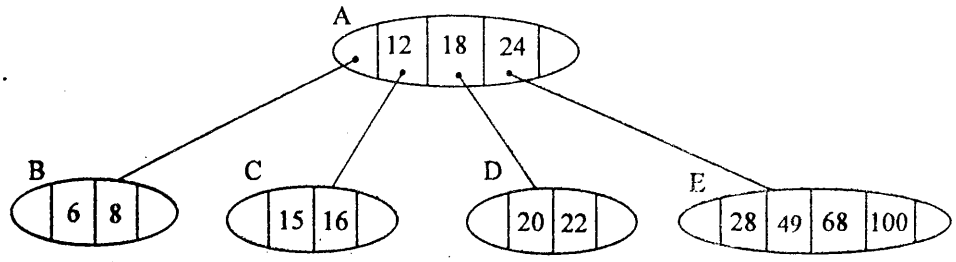
(4) Adding 100 will split C



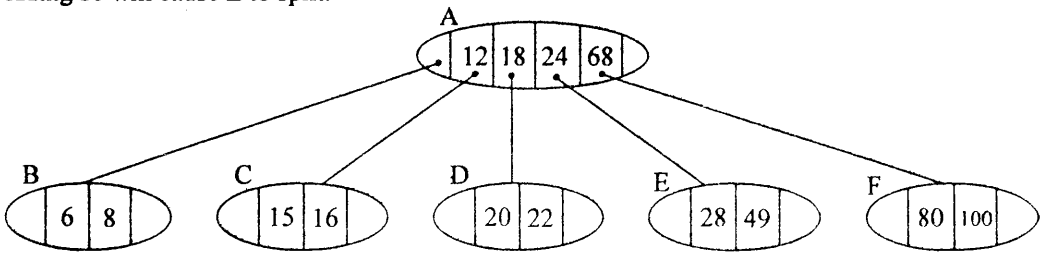
(5) 15 goes in C and 49, 68 goes in D, 20 goes in C.



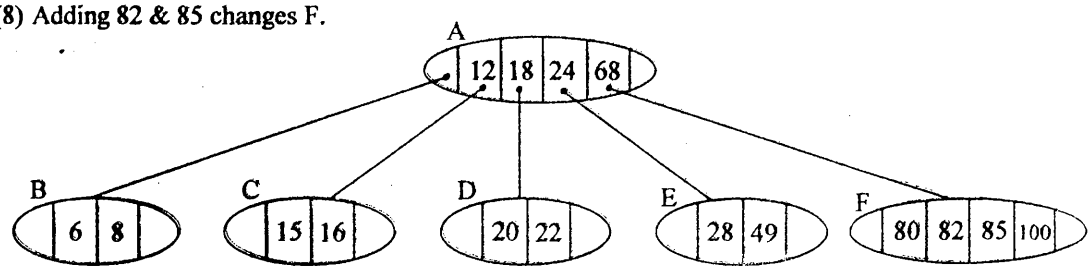
(6) Adding 22 in C causes split at node C.



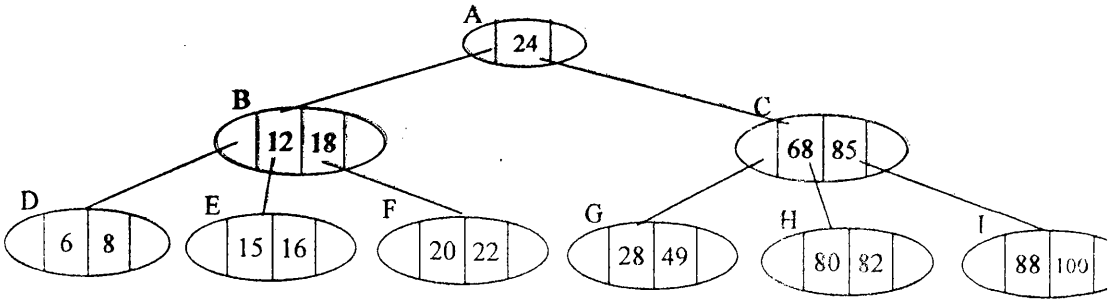
(7) Adding 80 will cause E to split.



(8) Adding 82 & 85 changes F.



(9) 88 will cause F to split which will inturn split the root node A.



Hashing: Hashing is a searching technique in that the address or index of the element to be searched is obtained by computing some arithmetic function. A function that transforms an element into an array index is called a hash function. If h is the hash function and key is the element, then h(key) is called the hash of key and is the index at which the element should be placed.

Collision : Suppose that two elements k_1 and k_2 are such that $h(k_1) = h(k_2)$. Then when element k_1 is entered into the table, it is inserted at position $h(k_1)$. But when k_2 is hashed, an attempt may be made to insert the element into the same position where the element k_1 is stored. Clearly two elements cannot occupy the same position. Such a situation is known as a hash collision or simply collision.

Hashing function : A hashing function f, transforms an identifier x into an array index in the table. The desired properties of such a function are that, it be easily computable ad that it minimize the number of collisions.

Some of the hash functions are given below :

1. **Mid–Square :** This function is computed by squaring the identifier and then using an appropriate number of bits from the middle of the square to obtain the array address. The number of bits to be used to obtain the array index depends on the table size. If r bits are used, the range of values is 2^r, so the size of hash table is chosen to be a power of 2 when this kind of scheme is used.
2. **Division :** Another simple choice for a hash function is obtained by using the mod operator. The element x is divided by some number M and the remainder is used as the hash address for x. i.e. f(x) = x mod M.
   This gives array indexes in the range O to M–1 and so the hash table is atleast of size = M.
3. **Folding :** In this method the element to be hashed is partitioned into several parts, all but the last being of the same length. These parts are then added together to obtain the hash address for that element. There are two ways of carrying out this addition. In the first, all but the last part are shifted so that the least significant bit of each part lines up with the corresponding bit of the last part. The different parts are now added together to get f(x). This method is known as shift folding. The other method of adding the parts is folding at the boundaries. In this method, the element to be hashed is folded at the part boundaries and digits falling into the same position are added together.

P_1	P_2	P_3	P_4	P_5
123	203	241	112	20

Shift folding	folding at the boundaries	
123	123	
203	302	– reverse of 203
241	241	
112	211	– reverse of 112
20	20	
699	897	

4. **Digit analysis :** This method is particularly useful in the case of a static file where all the elements in the table are known in advance. Each element is interpreted as a number using some radix r. The same radix is used for all the elements in the table. Using this radix, the digits of each element are examined. Digits having the most skewed distributions are deleted. Enough digits are deleted so that the number of digits left is small enough to give an array index in the range of the hash table.

Collision handling techniques : (Over flow handling techniques)

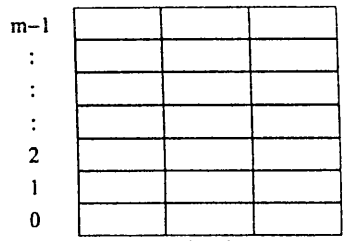
1. **Linear Probing :** In this technique, if the index given by the hash function contains some other element already, the new element will be searched in the next location in the hash table. The technique for finding the next available location is known as linear probing since a linear search will be performed to get the next available location.
2. **Rehashing :** In this technique, we may use a secondary hash function, called a rehash function on the hash key of the element. When the hash position of the item is found to be occupied by a different item during a search, the rehash function is used to locate the item. This rehash function is applied successively until an empty position is found where the item can be inserted.

3. **Ch**ining : T
has**h** to the s
des**i**red key.
4. **Usi**g bucke
Let the size o

The ab**o**e figure
Overflow : An o
full buc**k**et.

Interpolation Se
A techn**i**que for se
If the k**e**ys are u
more ef**f**cient tha
I**n** binary s
argume**n**t key is
betwee**n** these two
mid = low + (h
I**f** the key
average of $\log_2($
search's $\log_2 n$.
I**f** the keys
behavio**r**
I**f** the wors
case inte**r**polation
I**n** practica
uniform**l**y distribut
A variatior
It attem**p**ts to rem
advantag**e** over
establish**i**ng a valu
low + (hi**g**h – low)
min (hig**h** – gap, m
T**h**e next p
interval, **w**here gap
W**h**en the a
from low to mid an
If **t**he argu**m**
doubled, **a**lthough i
T**h**e expect
distributio**n** of keys
O**n** a list
approxim**a**tely 16
134 aver**a**ge com
required only 12.5.
O**n** a unifor
approximately 3.9 –
The worst c**a**
than that for binary
The comput
arithmetic on keys a

3. **Chaining :** The hash table will be a chain, i.e., builds a linked list of all items whose keys hash to the same value. During search, this short linked list is traversed sequentially for the desired key.
4. **Using buckets with move slots :** A bucket is a location which is divided into several slots. Let the size of the bucket is S. Then in one location S elements can be stored.



The above figure shows a hash table with bucket size is 3

Overflow : An overflow is said to occur when a new element is mapped or hashed by f into a full bucket.

Interpolation Search:

A technique for searching an ordered array is called "interpolation search".

If the keys are uniformly distributed between K(o) and K(n – 1), this method may be even more efficient than binary search.

In binary search, low is set to O and high is set to (n – 1), and throughout the algorithm, the argument key is known to be between K(low) and K(high). Keys are uniformly distributed between these two values, key would be expected to be at approximately position.

mid = low + (high – low) * [(key – k(low)] / [k (high) – k (low)]

If the keys are uniformly distributed through the array, interpolation search requires an average of $\log_2(\log_2 n)$ comparisons and rarely requires much more, compared with binary search's $\log_2 n$.

If the keys are not uniformly distributed, interpolation search can have very poor average behavior.

In the worst case, the value of mid can consistently equal (low + 1) or (high – 1), in which case interpolation search degenerates into sequential search.

In practical situations, keys often tend to cluster around certain values and are not uniformly distributed.

A variation of interpolation search, called robust interpolation search (or fast search). It attempts to remedy the poor practical behavior of interpolation search while extending its advantage over binary search to non uniform key distributions. This is done by establishing a value gap so that mid–low Gap is set to sqrt (high–low + 1). Probe is set to low + (high – low) * [(key – k (low)) / (k (high) – k (low))] and mid is set equal to min (high – gap, max (probe, low + gap)).

The next position used for comparison (mid) is at least gap positions from the ends of interval, where gap is atleast the square root of the interval.

When the argument key is found to be restricted to the smaller root of the two intervals, from low to mid and mid to high, gap is reset to the square root of the new interval size.

If the argument key is found to lie in the larger of the two intervals, the value of gap is doubled, although it is never allowed to be greater than half the interval size.

The expected number of comparisons for robust interpolation search for a random distribution of keys is O(log . log n).

On a list of approximately 40,000 names, binary search requires an average of approximately 16 keys comparisons. In practical situations, interpolation search required 134 average comparisons in an actual experiment, whereas robust interpolation search required only 12.5.

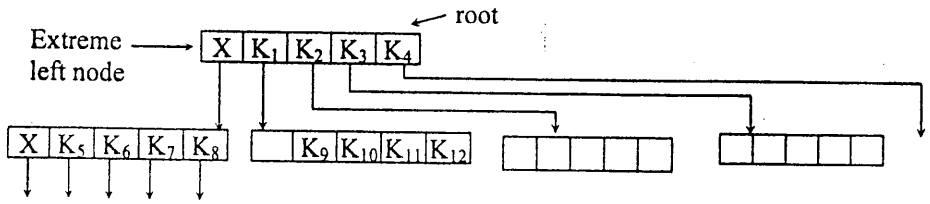
On a uniformly distributed list of approximately 40,000 elements – $\log_2(\log_2 40,000)$ is approximately 3.9 – robust interpolation search required 6.7 average comparisons.

The worst case for robust interpolation search is $(O(\log n)^2)$ comparisons, which is higher than that for binary search, but much better than the O(n) of regular interpolation search.

The computations required by interpolation search are very slow, since they involve arithmetic on keys and complex multiplications and divisions.

General tree

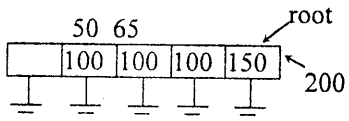
B– tree is called balanced multiway tree. Every node of B–tree can stored n key's and has n+1 pointers. Maximum no. of keys that can be stored in a node of a B–tree is called order of B–tree. e.g. Following is a B–tree of order of 4.



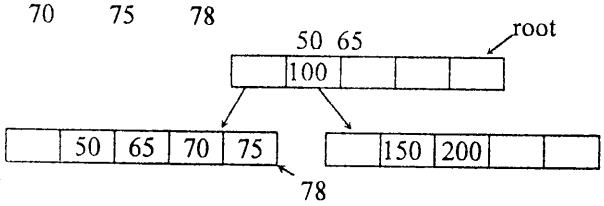
Every key is associated with a pointer which points to child node. There is one extra pointer called Extreme left pointer in every node, which is not associated with any key.
In the diagram values K_5, K_6, K_7, K_8 are less than K_1 values K_9, K_{10}, K_{11}, K_{12} are greater than K_1 but less than K_2 and so on.

Rule imposed in B–tree is as follows

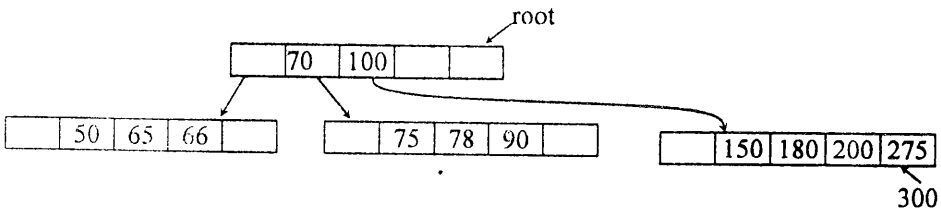
1. Every node of B–tree must contain atleast order divided 2 key's. i.e. Every node must be atleast half filled. Root is exception to this rule.
   e.g. data =100 50 65 150 200 70 75 78. 66 90 180 275 300 320 350 400 410 415 418
   1) add 100 50 65 150 200



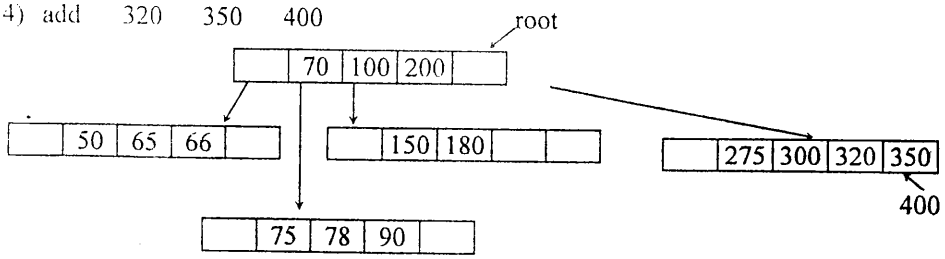
   2) add 70 75 78



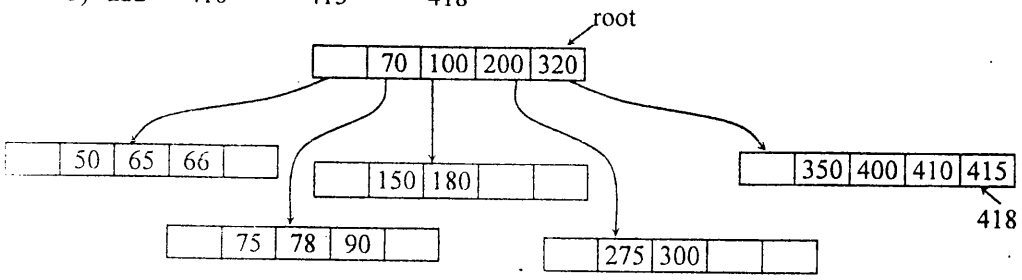
   3) add 66 90 180 275 300

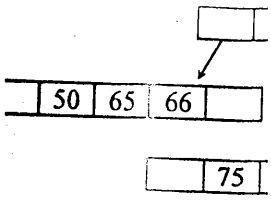


   4) add 320 350 400



   5) add 410 415 418





Split and Propagat
While inserting a n
node is split into tw
of the node is sent
continue if parent nc
will be split the mec
will grow.

e.g. – Consider a noc

The nod is full of t
split will contain 15 a

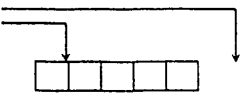
1. Construct B–tree
   data – 1 to 17
   1) ad 1 2

2) add 5 6

3) add 9, 10,

4) add 12

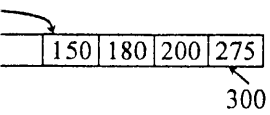
an stored n key's and has n+1
-tree is called order of B–tree.

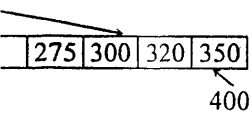


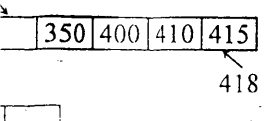
e. There is one extra pointer
vith any key.
$_{10}$, K_{11}, K_{12} are greater than K_1

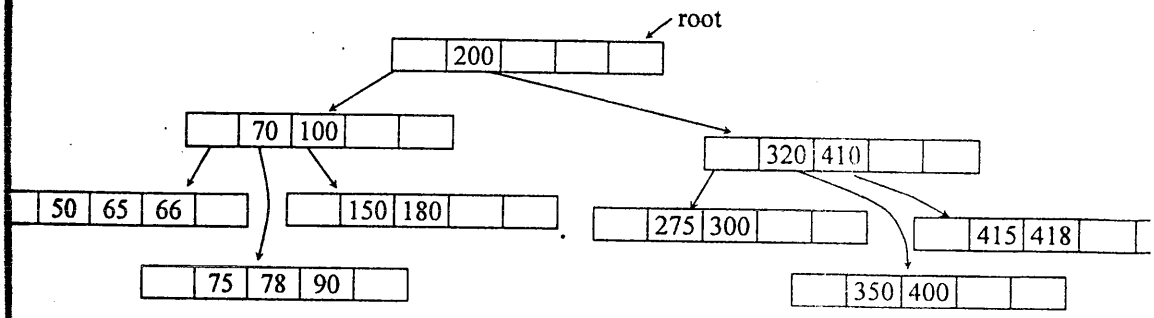
ey's. i.e. Every node must be

) 320 350 400 410 415 418





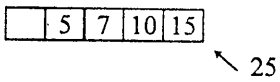




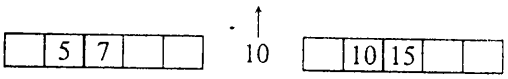
Split and Propagate Process

While inserting a new key in a full node process of split and propagate occurs. The overflown node is split into two different nodes such that every node stores order / 2 key's and the median of the node is sent up (propagate to the parent node. The process of split and propagate may continue if parent node is also full in the worst case if key reaches root which is full then the root will be split the median will be sent up to become new root. This is were the height of the tree will grow.

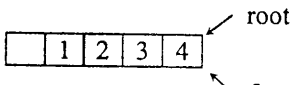
e.g. – Consider a node of order 4 in which 25 is getting inserted.



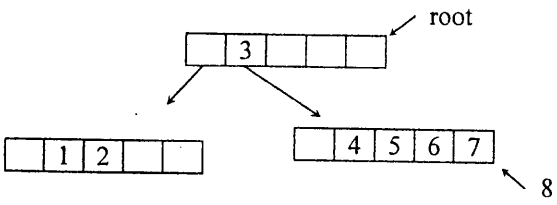
The node is full of therefore it will be split into two nodes left split will contain 5 & 7, Right split will contain 15 and 25 and the median i.e. 10 will be propagated to its parent node



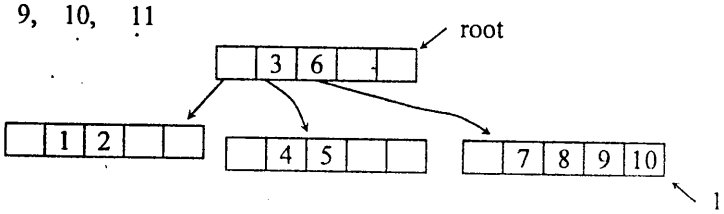
1. Construct B–tree of order 4 for following data
   data – 1 to 17
   1) add 1 2 3 4



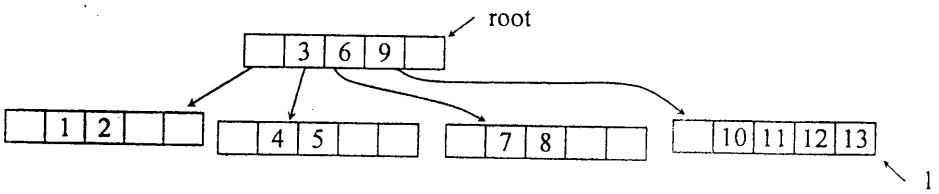
   2) add 5 6 7 8



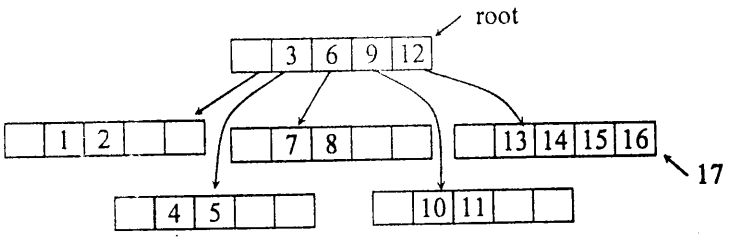
   3) add 9, 10, 11



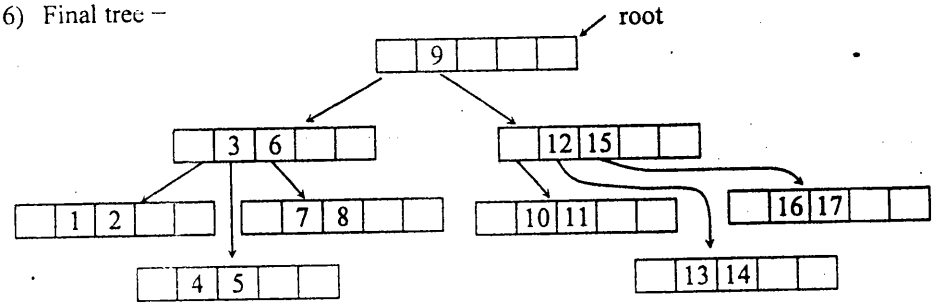
   4) add 12 13 14



5) add 15 16 17



6) Final tree –

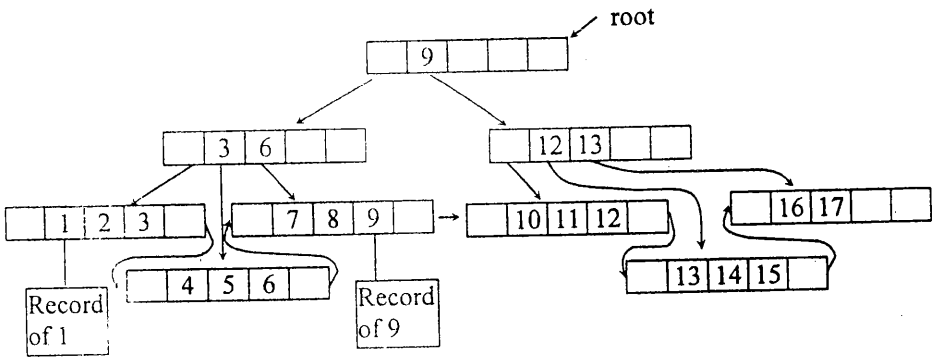


B + trees

Difference between B- tree and B + tree

	B-tree	B^+ tree
1	Every key is stored only once either in leaf node or non leaf node	Value stored in a non-leaf node are repeated again in one of the leaf node. i.e. leaf nodes coil contain all the keys
2	All leaf nodes are at same node but are isolated from each other	All leaves are at same level and are connected by using an additional pointer.
3.	Records are stored inside the B-tree i.e. every key is associated with its Record $\therefore B^-$ tree is larger in size and it cannot fit in RAM.	Records are stored outside B^+ tree and the add. of these records are stored by the pointer of the leaf node. $\therefore B^+$ tree smaller in size and can fit in RAM.
4.	B-tree supports Random traversal only	B^+ tree supports both Random & sequential traversal.

→ Following is a B^+ tree of order 4



Deleting a Key from B tree

Merge and Propogate

After deleting a key from a node, if the node is underflown then that underflown node is merged with neighbouring node and a key is borrowed from a parent node this is merge and propogate. After merging and propagation if the resultant node is overflown then split and propogate process occurs

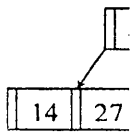
e.g. delete 17, 18

Node C is underfl
sequence 5 7 8 10 1

Delete 20 – Value

To delete 20, take
from leaf node C.

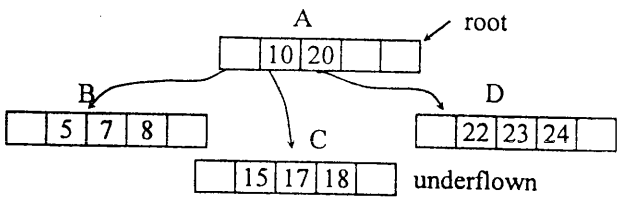
1. Using B-tree of
tree.



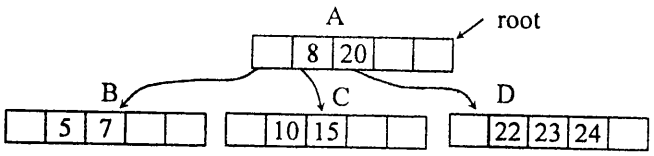
Exchange 60 of ro
propagate.

AVL tree (Adelso
AVL tree is Balanc
$|h_\ell - h_r| \leq 1$
$h\ell$ = height of
hr = height of

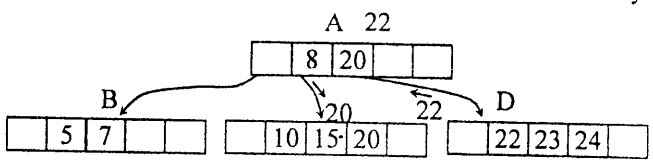
e.g. delete 17, 18



Node C is underflown. So merge node B and C and propagate 10 from parent. Now sorted sequence 5 7 8 10 15 is split into [5 | 7] & [10 | 15] . The median 8 is sent to the pare

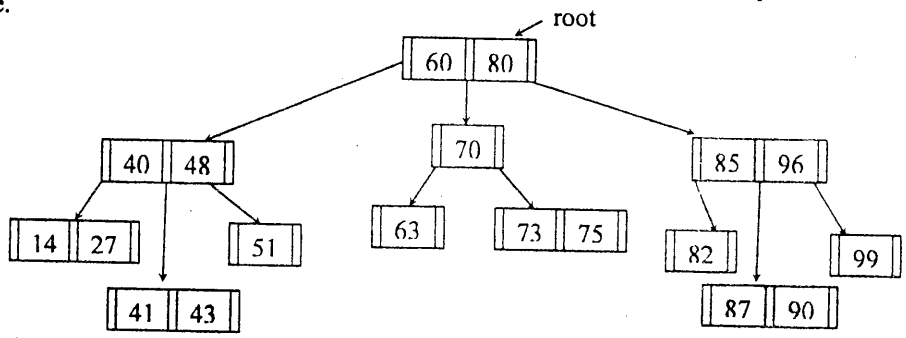


Delete 20 – Value which is present in root node cannot be deleted directly from root

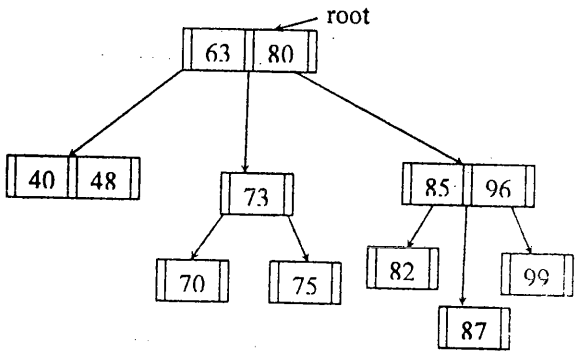


To delete 20, take 22 from node D to the node and rotate 20 the leaf node C. Now delete 20 from leaf node C.

1. Using B-tree of order 3 in the figure below, delete 90 and 60 in each step show the resultant tree.



Exchange 60 of root node with 63 of leaf node and then delete 60. Continue with merge and propagate.



AVL tree (Adelson velskil and landis tree)

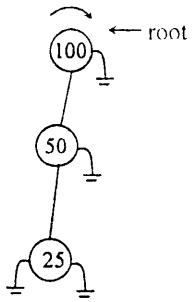
AVL tree is Balanced Binary Search tree. In which every node must satisfy following criteria :–

$|h_\ell - h_r| \leq 1$

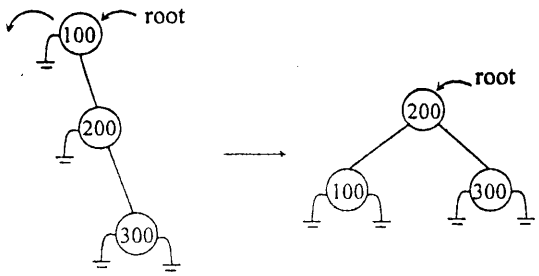
$h\ell$ = height of left subtree

hr = height of Right subtree.

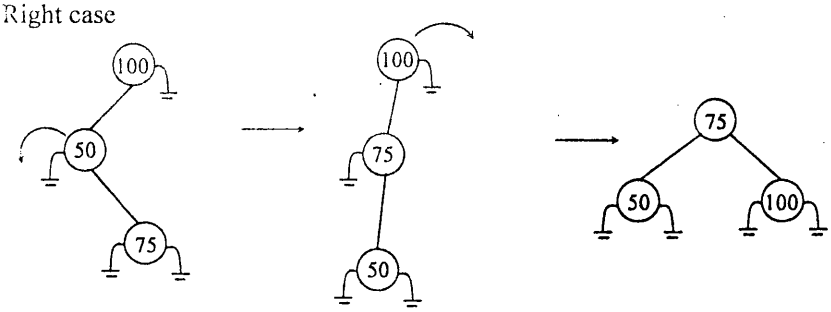
If any node does not obey the criteria then the tree is unbalanced.
AVL gives 4 Rotation Algorithm for 4 different cases, which are useful for Balancing the tree.
Case I – Left – Left Case



100 is not balanced due to left–left case. So rotate 100 to right. This is Right rotation.
Case 2.– Right – Right case



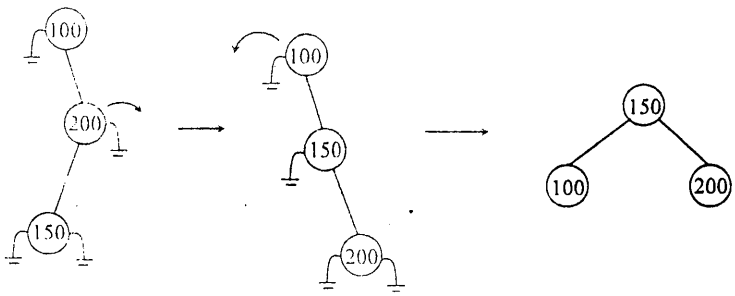
100 is not balanced due to right–right case. So rotate 100 to left. This is left rotation.
Case 3 : – Left – Right case



Left – left case

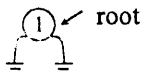
100 is not balanced due to left-right case. So rotate left child of 100. i.e. so this is left rotation.

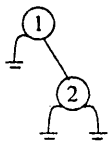
Case 4 : – Right – Left case



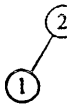
100 is not balanced due to right-left case. So rotate right child of 100. i.e. 150 this is right rotation.

Show the steps of creating AVL tree using following data .

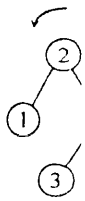
1) 

2) 

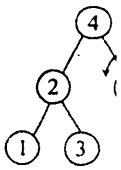
3) 

4) 

Improve Binary s

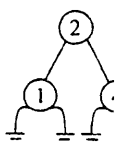
6. 

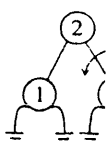
Root 2 is not Bal

7. 

Show the result o

1. 

3. 

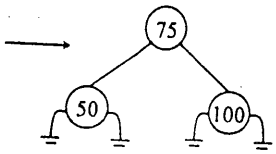
5. 

:d.
re useful for Balancing the tree.

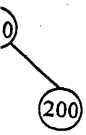
This is Right rotation.

root

This is left rotation.

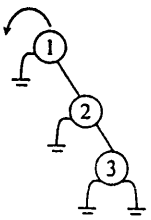


0. i.e. so this is left rotation.

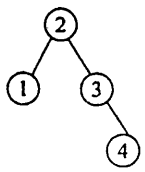


of 100. i.e. 150 this is right

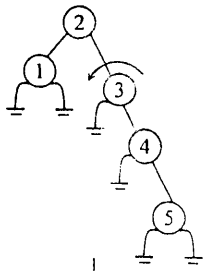
3)

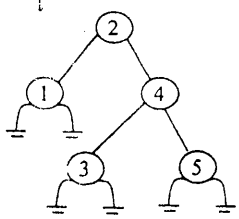


4)



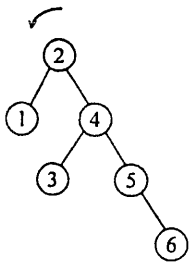
5)

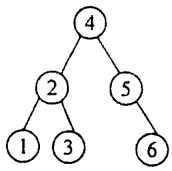




Improve Binary search tree searching become faster

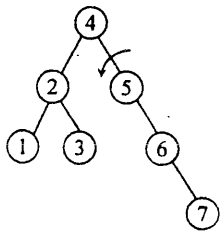
6.

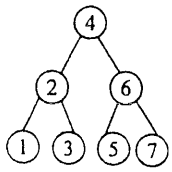




Root 2 is not Balanced due to Right–right, rotate left.

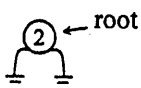
7.



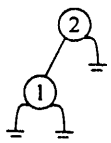


Show the result of inserting 2 1 4 5 9 3 6 7 into an initially empty AVL tree.

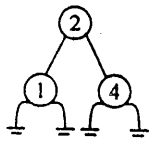
1.



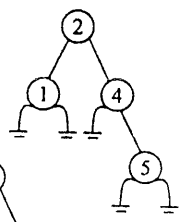
2.



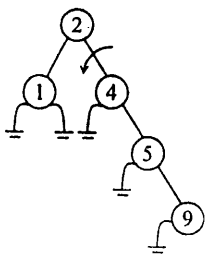
3.

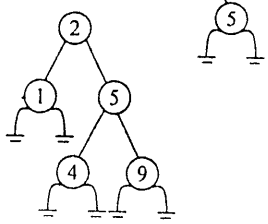


4.

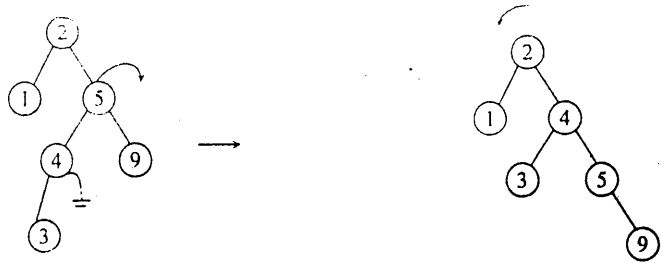


5.

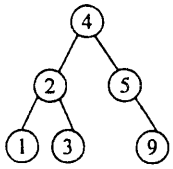




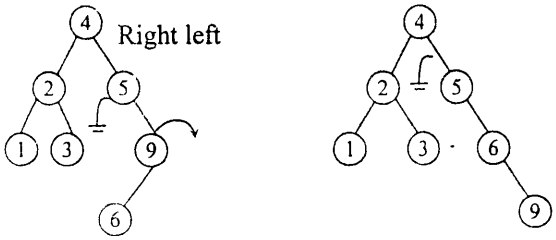
6.



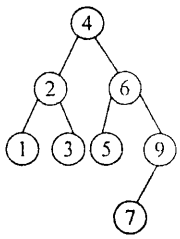
2 is not balance because of 5, so right left



7.



5 is unbalanced rotate to the left



Graded Questions :

1. Give the possible solutions for improving the efficiency of sequential search. **[M-06]**
2. Explain Index Sequential searching method. Give its complexity and application. **[M-06]**
3. Write Short note on Indexed sequential search. **[N-06, D-07]**
4. Develop a program to search an element in a given set of data by binary search technique.
5. Write a program to implement 'Interpolation Search' **[N-06, M-07, D-07]**
6. Suggest the data structure to store the information such that it will be faster to search than binary search. **[M-06]**
7. Explain binary search tree in brief.
8. What are different ways in which a binary tree can be traversed ?
9. Explain how insertion and deletion of a node can take place in binary search tree.
10. Write a C/C++ function to delete a node form binary search tree. **[N-04]**
11. Write a program for binary search. Explain how it is better than linear search.
12. Compare binary search with indexed search.
13. Write short note on B^-tree and B^+ tree. **[N-06]**
14. Explain B+ trees with an example and show how insertions can be done in it. State its applications. **[M-06]**
15. Assume that the following keys are inserted into a B – tree of order 4 : **[N-05]**
    0 1 2 3 4 5 6 7 8 9
    which keys cause page splits to occur ? Which key cause the height of the tree to increase ? Show the growth of this B – tree with neat diagrams.
16. Write note on Digital search trees. **[M-07, D-07]**
17. Show how to implement a tries in external storage. Write a 'C' search and insert routine for a tries.
18. What is Tries ? Explain insertion of element in Tries. **[M-04, 05, N-04, 05, 06]**

19. What are A
20. What are ... it MAR, M...
21. Write a no...
22. Explain A...
23. (a) A bina... below.
    (b) What ... rebalan...
24. A binary t... Draw the t...
25. (a) Define ... general...
    (b) Using t... resultir...
26. Write a f... possible c...
27. Write an ... loop iterat...
28. Draw the ...
29. Write an al...
30. (a) A bina... given t...
    (b) Show t...
31. Suppose ... first spann...
32. Explain D...
33. Explain B...

Hashing

34. What is H... with exam...
35. Explain h... technique.
36. Explain H...
37. Explain in...
38. Explain fo...
    (i) Mid-sq...
39. Explain dy...
40. What is H...

19. What are AVL trees ? Explain with suitable example four cases that require rebalancing. [N-06, D-07

20. What are AVL trees ? Consider following set of symbols and construct AVL trees usir it MAR, MAY, JUNE, JULY, JAN, FEB, AUG, SEPT, NOV, OCT, APRIL, DEC.

[N-04, M-0

21. Write a note on Multiway search tree.

22. Explain AVL RL and LR rotation with suitable example. [M-06]

23. (a) A binary tree has 10 nodes. The inorder and preorder traversal of the tree are show below. Draw the tree: Preorder JCBADEFIGH
Inorder ABCDFJGIH [M-03]

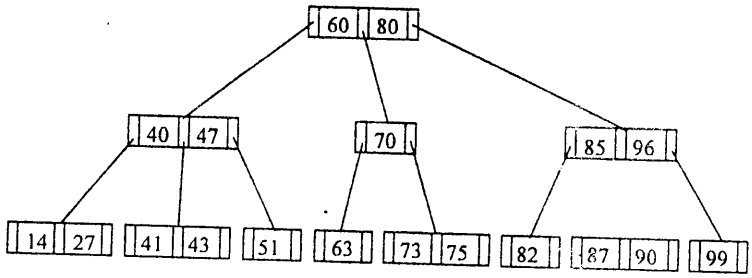
(b) What are AVL trees ? Explain with suitable example the four cases that requir rebalancing. [M-03, 04]

24. A.binary tree has eight nodes. The inorder and postorder traversal of the tree is given below Draw the tree.

Postorder : F E C H G D B A
Inorder : F C E A B N D G [M-07]

25. (a) Define General trees. Explain with suitable example the steps required to represent general tree with a binary tree format. [M-03]

(b) Using the B–tree of order 3 shown in figure below, delete 90, 60. In each steps show the resulting B–tree. [M-03]



26. Write a function to implement node delete operation of Binary search tree. Check all possible conditions. [M-07, D-07]

27. Write an algorithm for Binary Search and find 88 in the data given below. At each loop iteration, including the last show the content of first, last and mid.
8, 13, 17, 26, 44, 56, 88, 97. [M-03]

28. Draw the B tree of order 3 created by inserting the following data arriving in sequence.
92, 24, 6, 7, 11, 8, 22, 4, 5, 16, 19, 20, 78. [M-07]

29. Write an algorithm for sequential search. [D-03]

30. (a) A binary tree has seven nodes. The preorder and postorder traversal of the tree is given below. Draw the tree :– Preorder :– GFDABEC
Postorder :– ABDCEFG [D-03]

(b) Show the result of the inserting 2, 1, 4, 5, 9, 3, 6, 7 into an initially empty AVL tree.

31. Suppose we perform a Breath first search of an undirected graph and build a Breath first spanning tree. Show that all edges in the tree are either tree edge or cross edge. [D-03]

32. Explain DFS algorithm. Explain it with suitable example. [M-04, M-05]

33. Explain Breadth First search algorithm, explain it by example. [N-04, N-05]

Hashing

34. What is Hashing ? What are collisions ? Explain at least 3 collision-handling techniques with example. [M-06, N-06]

35. Explain hash addressing. Which are hashing techniques ? Implement hashing using any technique. [D-07]

36. Explain Hashing. Give methods to resolve clashes. [M-04]

37. Explain in brief the hashing in external storage.

38. Explain following hash functions in brief :
(i) Mid-square (ii) Division (iii) Folding (iv) Digit analysis

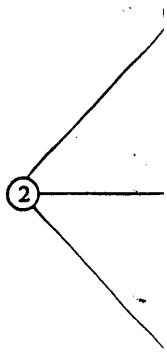
39. Explain dynamic hashing and extendible hashing with suitable example. [M-03]

40. What is Hashing ? What are Collisions ? What are the different ways to handle collisions ? [M-04, N-04]

41. Compare hashing with chaining and hashing without chaining. Give an example to show the difference between the two for larger amount of values and amount of values of numbers to be sorted.
42. Using modulo division method and linear probing, store the keys shown below, array with 19 elements. How many collisions occurred ? What is the density of the list after all keys have been inserted ? **[M-06, 07]**
    224562 137456, 214562, 140145, 214576,
    162145, 144467, 199645, 234534, 199645,
43. Explain direct hashing with suitable example. **[D-03]**
44. What are AVL trees ? Consider following set of symbols and construct AVL tree using them.
    MAR, MAY, JUNE, JULY, JAN, FEB, AUG, SEPT, NOV, OCT, APRIL, DEC. **[M-05]**
45. State various requirements of Hashing function.
    What are collisions ? Explain various collision handling techniques with example. **[M-05]**
46. Write an algorithm for Binary search method. Prove that it's efficiency Is $O(\log_2 n)$. **[M-06]**
47. Discuss the Binary search technique taking an examples. **[M–05]**
48. Explain Various Collision Handling techniques **[N-06]**

❑❑❑❑❑❑

A graph G, consi
pairs of vertices;
and edges of gra
graph the pair o
(V_2, V_1) represen
$< V_2, V_1 >$ and <



The graphs G_1 ar
$V(G_1) = \{1,$
$V(G_2) = \{1,$
$V(G_3) = \{1,$

The maximum n

graph with exact

while G_2 and G
maximum numb

If (V_1, V_2) is an
(V_1, V_2) is incide

A subgraph of G
A path from vert
such that $(V_p, V$
number of edges

A simple path is
such as (1, 2), (2
3 in G. The first

A cycle is a simj
G_1. 1, 2, 1 is a c

In an undirected
from V_1 to V_2.
V_i and V_j in V(C

ning. Give an example to show the
nd amount of values of numbers to

re the keys shown below, array
t is the density of the list after all
[M-06, 07]

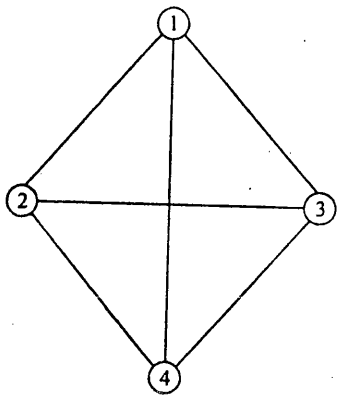
[D-03]
nd construct AVL tree using them.
, OCT, APRIL, DEC. [M-05]

nniques with example. [M-05]
s efficiency Is $O(\log_2 n)$. [M-06]
[M–05]
[N-06]

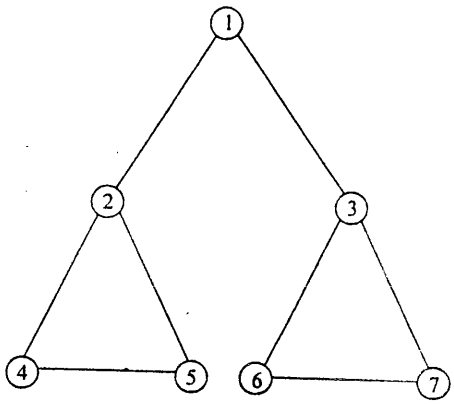
Vidyalankar

Ch.4 : Graph

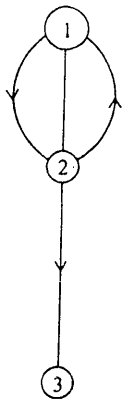
A graph G, consists of two sets V and E. V is a finite non–empty set of vertices, E is a set of pairs of vertices; these pairs are called edges. V(G) and E(G) will represent the sets of vertices and edges of graph G. We will also write G = (V, E) to represent a graph. In an undirected graph the pair of vertices representing any edge is unordered. Thus, the pairs (V_1, V_2) and (V_2, V_1) represent the same edge. In a directed graph each edge is represented by a direct pair $< V_2, V_1 >$ and $<V_1, V_2>$ represent two different edges.



G_1



G_2



The graphs G_1 and G_2 are undirected G_3 is a directed graph or digraph.

$V(G_1) = \{1, 2, 3, 4\}$
$V(G_2) = \{1, 2, 3, 4, 5, 6, 7\}$
$V(G_3) = \{1, 2, 3\}$

$E(G_1) = \{(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)\}$
$E(G_2) = \{(1, 2), (1, 3), (2, 4), (2, 5), (3, 6), (3, 7)\}$
$E(G_3) = \{<1, 2>, <2, 1>, <2, 3>\}$

The maximum number of edges in an undirected graph of n vertices is $\frac{n(n-1)}{2}$. An n vertices graph with exactly $\frac{n(n-1)}{2}$ edges is said to be complete. G_1 is the complete graph on 4 vertices while G_2 and G_3 are not complete graphs. In the case of directed graph on n vertices the maximum number of edges in n (n – 1).

If (V_1, V_2) is an edge in a graph, then we shall say the vertices V_1 and V_2 are adjacent and edge (V_1, V_2) is incident on vertices V_1 and V_2.

A subgraph of G is a graph G ′ such that $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$.
A path from vertex V_p to vertex V_q in graph G is a sequence of vertices $V_p, V_{i_1}, V_{i_2}, \ldots V_{i_n}, V_q$, such that $(V_p, V_{i_1}), (V_{i_1}, V_{i_2}), \ldots\ldots, (V_{i_n}, V_q)$ are edges in E(G). The length of a path is the number of edges in it.

A simple path is a path in which all vertices except possibly the first and last are distinct. A path such as (1, 2), (2, 4), (4, 3) we write as 1, 2, 4, 3. Paths 1, 2, 4, 3 and 1, 2, 4, 2 are both of length 3 in G. The first is a simple path while the second is not.

A cycle is a simple path in which the first and last vertices are the same. 1, 2, 3, 1 is a cycle in G_1. 1, 2, 1 is a cycle in G_3.

In an undirected graph, G, tow vertices V_1 and V_2 are said to be connected if there is a path in G from V_1 to V_2. An undirected graph is said to be connected if for every pair of distinct vertices V_i and V_j in V(G) there is a path from V_i to V_j in G.

Graph Representations :

Graph can be represented in several forms. We shall study only the two most commonly used forms viz., adjacency matrix representation and adjacency list representation.

I. Adjacency Matrix

Let G = (V, E) be a graph with n vertices, n ≥ 1. The adjacency matrix of G is a 2 dimensional n × n array, say A, with the property that A [i, j] = 1 if and only if the edge (V_i, V_j) is E(G). A[i, j] = 0 if there is no such edge in G. The adjacency matrix for the graph G_1 is given below.

$$\text{Adj} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

The adjacency matrix is also known as bit matrix. In the adjacency matrix 1 can be considered as true and 0 can be considered as false. So this matrix can be considered as a Boolean matrix.
If Adj [i, j] = true or 1 then we can say that there exists a path from i to j of length.

1. Find out Adj 2 by multiplying Adj with Adj (use Boolean matrix multiplication rules.)
   If Adj_2 [i, j] = true or 1 then we can say that there exists a path from i to j of length.
2. Similarly if Adj_3 [i, j] = true or 1 then there exists a path from i to j of length 3 with Adj_3 is product of Adj_2 and Adj.

In general if Adj_n [i, j] = true or 1 then there exists a path from i to j of length n, where Adj_n is the product of Adj_{n-1} and Adj.

Transitive Closure

Let Adj be the adjacency matrix of a given graph.
Then
Path = Adj or Adj_2 or Adj_3 or Adj_4
is known as the transitive closure of the matrix Adj, where Adj_2 is the Boolean produce of Adj with Adj, Adj_3 is the Boolean product of Adj_2 and Adj and so on.

If Path [i, j] = 1 or true then we can say that there is a path from i to j of length 1 or 2 or 3 ...or n.

Boolean Matrix Multiplication

Boolean Matrix : It is a 2 dimensional array which contains only two types of values either true or false.

There are two differences between numerical matrix multiplication and Boolean matrix multiplications.

1. Numerical multiplication will be replaced by conjunction (i.e., * is replaced by and) in Boolean matrix multiplication.
2. Numerical addition will be replaced by disjunction (i.e., + is replaced by or) in Boolean matrix multiplication).

Procedure for finding transitive closure

```
void transclose (int adj [ ] [maxnodes] , path [ ] [maxnodes ] )
{
    int i, j, k ;
        int newprod [maxnodes]  [maxnodes] ,   adjprod [maxnodes] [maxnodes] ;
    adjprod = adj ;
    path = adj ;
    for (i = ϕ ; i < = maxnodes – 2 ; i++)
    {
        prod (adjprod, adj, newprod) ;
        {finding the Boolean matrix products of adjprod, adj }
        for (j = ϕ; j < = maxnodes – 1; j++)
```

```
                for (
                    path
                adjprod
        }
}

C ro ing for
void  rod (int a
{
    in  val ;
    in  i, j, k ;
    fo  (i = ϕ, i
        for (j =
        {
            val =
            for (
            val =
            c [ i
        }
}
```

Assumptions
Maxnodes – nu
the ma n progra

Wars all's alg
void t nsclose
{
    int , j, k ;
    pat = adjm
    for k = ϕ, k
        for (i = ϕ
            if (pa
            for (
        path [i]
}

To analyse the e
n is the number
loop which is re

Wars all's alg
The a ove meth
anothe algorith
algori m to find
Warsh ll's algor

II. A jacency I
Using djacency
matrix (diagona
graph. When gr
can be reduced t
made ossible th
are re esented.

In this represent
list for each vert
vertex i. Each n

```
        for (k = 1 ; k <= maxnodes – 1; k ++)
            path [ i ] [ k ] = path [ i ] [ k ] || newprod [j, k] ;
        adjprod = newprod ;
    }
}
```

C routing for Boolean Matrix Multiplication

```
void prod (int a [ ] [maxnodes], b [ ] [maxnodes] , c [ ] [maxnodes] )
{
    int val ;
    int i, j, k ;
    for (i = ϕ, i <= maxnodes – 1; i + + )
        for (j = ϕ ; j <= maxnodes – 1 ; j + + )
        {
            val = ϕ ;
            for (k = ϕ; k <= maxnodes – 1; k + +)
            val = val || (a [ i ] [ k ] && b [ k ] [ j ] ) ;
            c [ i ] [ j ] = val ;
        }
}
```

Assumptions

Maxnodes – number of nodes in the graph ; adjmatrix is Boolean type matrix will be declared in the main program.

Warshall's algorithm

```
void transclose (int adjmat [ ] [maxnodes], int path [ ] [maxnodes] )
{
    int i, j, k ;
    path = adjmat ,
    for (k = ϕ, k <= maxnodes – 1, k + + )
        for (i = ϕ ; i <= maxnodes – 1; i + +)
            if (path [ i ] [ k ] = = 1)
            for ( j = ϕ ; j <= maxnodes – 1 ; j + + )
        path [ i ] [ j ] = path [ i ] [ j ] || path [ k ] [ j ] ;
}
```

To analyse the efficiency of this procedure, note that finding the Boolean product is $O(n^3)$, where n is the number of graph nodes. In transclose, this process (the call to prod) is embedded in a loop which is repeated n – 1 times, so the entire transitive closure procedure is $O(n^4)$.

Warshall's algorithm

The above method to find out the transitive closure is quite inefficient. Warshall has devised another algorithm to find out the transitive closure which is $O(n^3)$. This is known as Warshall's algorithm to find the transitive closure.

Warshall's algorithm's Pascal implementation is given below :

II. Adjacency List Representation

Using adjacency matrices all algorithms will require atleast $O(n^2)$ time as $n^2 - n$ entries of the matrix (diagonal entries are zero) have to be examined where n is the number of nodes of the graph. When graphs are sparse, i.e., most of the terms in the adjacency matrix are zero, the time can be reduced to O(e + n) where e is the number of edges in the graph. Such a speed up can be made possible through the use of linked lists in which only the edges that are present in the graph are represented. This type of representation is known as adjacency List representation.

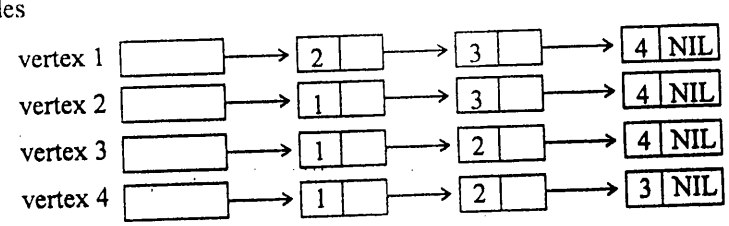
In this representation n rows of the adjacency matrix are represented as n liked lists. There is one list for each vertex in the graph. The nodes in a list i represent the vertices that are adjecent from vertex i. Each node has at least two fields : vertex and link.

Each list has a headnode. The headnodes are sequential providing easy random access to the adjacency list for any particular vertex. The declarations in Pascal for the adjacency list representation is given below :

```
Type nextnode  = ↑ node ;
          node = Record
                   vertex  : integer ;
                   link : nextnode ;
                 end ;
headnodes : array [1 . maxnodes' of nextnode ;
```

Adjacency list representation for graph G_1 is given below :

headnodes



Write a Pascal function to find out whether there is a path of fixed length between two nodes of a graph.

Graph Traversal

Given an undirected graph G = (V, E) and a vertex V in V(G) we are interested in visiting all vertices in G that are reachable from V (i.e., all vertices connected to V). We shall look at two ways of doing this : Depth first search and Breadth first search.

1. Depth first search

Depth first search of an undirected graph proceeds as follows :
The start vertex V is visited. Next an unvisited vertex w adjacent to v is selected and a depth first search from w initiated. When a vertex u is reached such that all its adjacent vertices have been visited, we back up to the last vertex visited which has an unvisited vertex w adjacent to it and initiate a depth first search from w. The search terminates when no unvisited vertex can be reached from any of the visited ones.

Algorithm is given below

```
Procedure dfs (V : integer) ;
{Given an undirected graph G = (V, E) with n vertices and an array visited [ n ] initially
set to false, this algorithm visits all vertices reachable from V. Visited in a global array}

Var   W : integer ;
begin
      Visited [V] = true ;
      for each vertex w adjacent to V do
      if not visited [W] then dfs (W) ;
      end ;
```

In case the graph G is represented by its adjacency lists then the vertices w adjacent to v can be determined by following a chain of links. Since the algorithm dfs would examine each node in the adjacency lists at most once and there are 2e list nodes, the time to complete the search is O(e). If G is represented by its adjacency matrix, then the time to determine all vertices adjacent to v is O(n). Since at most n vertices are visited, the total time is $O(n^2)$.

A depth first traversal traverses a single path of the graph as far as it can go (that is, until visits a node with no successors or a node all of whose successors have already been visited). There may be several depth first traversals. The traversal depends very much on how the graph is represented and how the nodes are numbered.

2. Breadth first search

Starting at vertex v and marking it as visited breadth first search next all unvisited vertices adjacent to v. Then unvisited vertices adjacent to there vertices are visited and so on.

Algorithm follows :

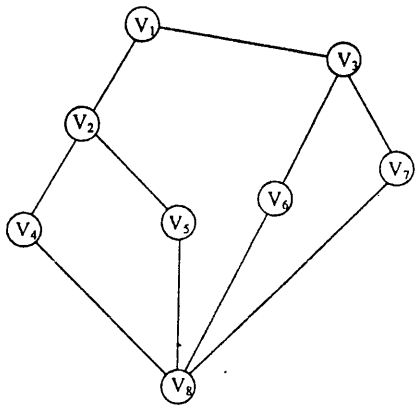
```
Procedure bfs (v : integer) ;
{A breadth first search of G is carried out beginning at vertex v. All vertices visited are marked
as visited [ i ] = true. The graph G and array visited are global and visited is initialized to false.
Initializequeue, addqueue, emptyqueue and deletequeue are procedures/functions to handle queue
operation }
Var     w : integer ;
        q : queue ;
begin
        visited [ v] = true ;
        initialize queue ( q ) ; {q is a queue }
        addqueue (q, v) ; {add vertex v to queue}
        while not empty queue (q) do
            begin
        deletequeue (q, v) ; {remove from queue vertex v}
        for all vertices w adjacent to v do
        if not visited [ w ] then
begin
        adqueue (q, w) ;
        visited [ w] = true ;
            end ;
        end ;
end ;
```

Each vertex visited gets into the queue exactly once, the while loop is iterated at most n times. If an adjacency matrix is used, then the loop takes O(n) time for each vertex visited. The total time is therefore $O(n^2)$. In case of adjacency lists are used the loop has a total cost of $d_1 + \ldots + d_n = O(e)$ where d_i = degree (V_i).

Note :

Degree : The degree of a vertex is the number of edges incident to that vertex. The degree of vertex (1 ( ) graph G_1 is 3. In case of a directed graph we define the in–degree of a vertex V to be the number of edges for which V is the head. The out–degree is defined to be the number of edges for which V is the tail.

Consider the graph given below :

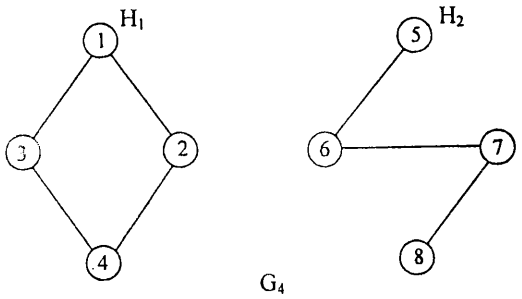


If you are initiating a depth first search starting the node V_1, then the vertices of the graph will be visited in the order $V_1, V_2, V_4, V_8, V_5, V_6, V_3, V_7$.

If a breadth first search is initiated from vertex V_1 then the vertices of the graph are visited in the order $V_1, V_2, V_3, V_4, V_5, V_6, V_7, V_8$.

Connected Components

One of the applications of graph traversal is to find all connected components of a given graph.
Consider the following graph

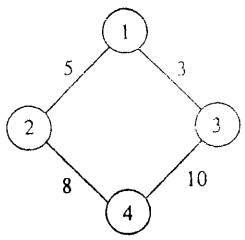


G_4

The above graph has two connected components viz H_1 and H_2.
Using graph traversal algorithm one can determine whether a graph is connected or not. After bfs or dfs calls, if there is any unvisited vertex in the graph, then we can decide that the graph is not connected.

An algorithm to find the connected components of a given graph is given below :
Procedure comp (g : undirected graph) ;
{determine the connected components of g, g has n ≥ 1 Vertices. Visited is now a local array }

Weighted graphs :



Consider the above graph. A number is associated with each arch or edge of the graph. Such a graph in which a number is associated with each are is called a weighted graph or a network. These weights can be cost of construction, the length of the link etc.

Greedy Method

The greedy method is one of the most straight forward algorithm design technique ; and it can be applied to a wide variety of problems.

The greedy method suggests that one can devise an algorithm which works in stages, considering one input at a time. At each stage, a decision is made regarding whether or not a particular input is in an optimal solution. This is done by considering the inputs in an order determined by some selection procedure. If the inclusion of next input into the partially constructed optimal solution will result in an infeasible solution then this input is not added to the partial solution.

(Most problems have n input and require us to obtain a subset that satisfies some constraints is called a feasible solution. We are required to find a feasible solution that optimizes (minimizes or maximizes) a given objective function. A feasible solution that does this is called an optimal solution.)

Knapsack Problem

We will see how the greedy strategy can be applied to solve the knapsack problem.
Problem definition : We are given n objects and a knapsack. Object i has a weight wi and the knapsack has a capacity M. If a fraction x_i , $0 \le x_i \le 1$, of object i is placed into the knapsack then a profit of $P_i X_i$ is earned.

The objective is to obtain a filling of the knapsack that maximizes the total profit earned.
So, the problem may be formally stated as

maximize

subject to

and $0 \le x_i \le$

The profits and we

A feasible solution
solution is a feasib

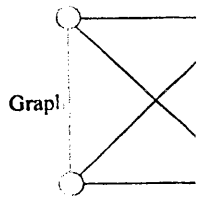
Algorithm

All the objects sho
i and wi is the wei
Procedure knapsac
{P and W are two
objects ordered so
M is the knapsack
Begin For i = 1 to
x [
cu
i =
While (i <
begin X(i)
i
cu
enc
if i <= n t

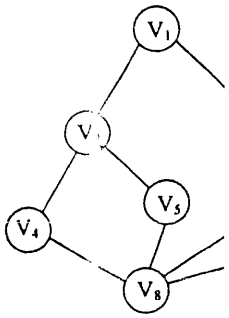
Spanning tree

Any tree conside
known as a spani
depth first spanni
breadth first span

The following fig



The following fig



maximize $\sum_{i=1}^{n} P_i X_i$ .....(1)

subject to $\sum_{i=1}^{n} n\ W_i X_i\ m$ .....(2)

and $0 \le x_i \le 1, 1 \le i \le n.$ .....(3)

The profits and weights are positive numbers.

A feasible solution (or filling) is any set $(x_1, ..., x_n)$ satisfying (2) and (3) above. An optimal solution is a feasible solution for which (1) is maximum.

Algorithm

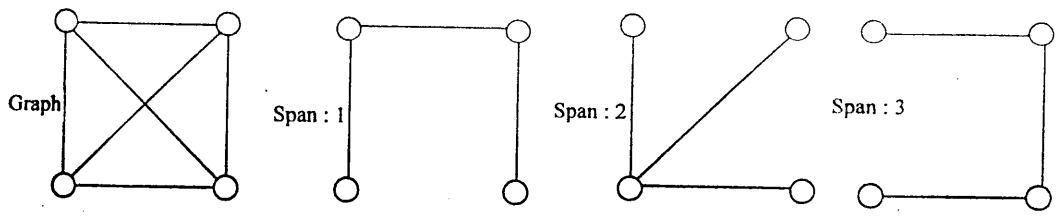
All the objects should be sorted into non increasing order of pi/wi where pi is the profit on object i and wi is the weight of object i.

```
Procedure knapsack (P, W : array [1 – n] of real) ;
{P and W are two array of size 1 ... n and contain the profits and weights respectively of the n
objects ordered so that (i) W (i) ≥ P (i + 1) / W(i + 1).
M is the knapsack size and x is an array of size and which is the solution array}.
Begin For i = 1 to N Do
        x [ i ] = 0;  {initialize solution to zero}
        cu = m ;   {remaining knapsack capacity}
        i = 1;
  While (i <= n) and (W [ i ]  <= cu ) Do
  begin X(i) = 1
        i   = i + 1
        cu  =  cu – W [ i] ;
        end ;
  if i <= n then X [ i ] = cu / W [ i ].
```

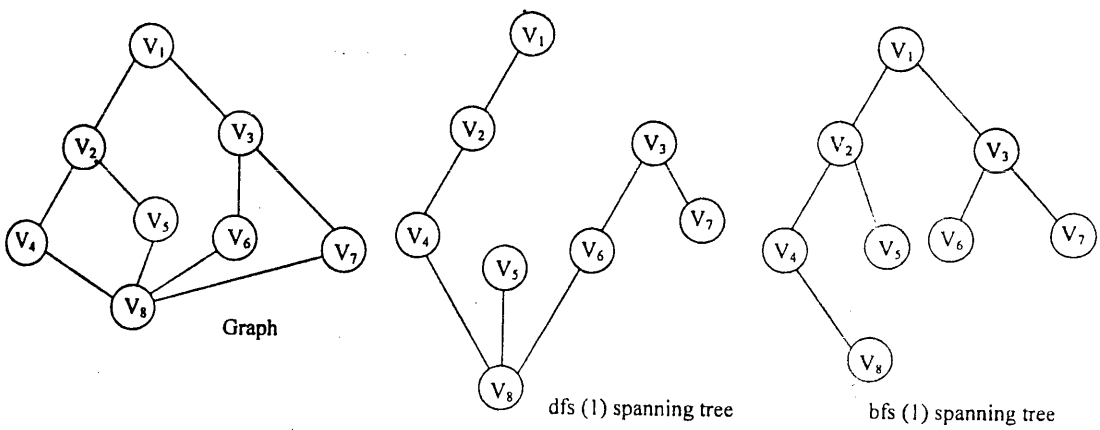
Spanning tree

Any tree considering solely of edges of a graph and including all the vertices of the graph is known as a spanning tree. The spanning tree resulting from a depth first search is known as a depth first spanning tree and the spanning tree resulting from a breadth first search is know as a breadth first spanning tree.

The following figure shows a complete graph and three of its spanning trees.



The following figure shows a graph and its dfs (1) spanning tree and dfs (1) spanning tree.



Minimum Cost spanning tree

Give a connected weighted graph G, it is often desired to create a spanning tree T for G such that the sum of the weights of the tree edges in T is as small as possible. Such a tree is called a minimum cost spanning tree and represent the cheapest way of connecting all the modes in G.

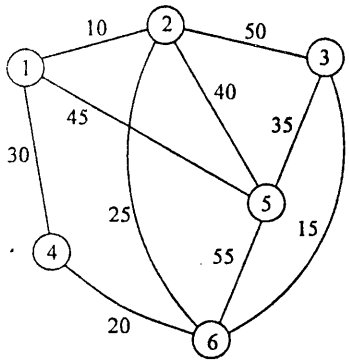
A greedy method to obtain a minimum cost spanning tree would build this tree edge by edge. The next edge to include is chosen according to some optimization criterion. The simplest such criterion would be to choose an edge that results in a minimum increase in the sum of the costs of edges so for included. Prim's algorithm to find out the minimum cost spanning tree.

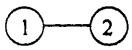
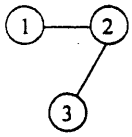
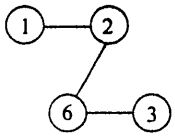
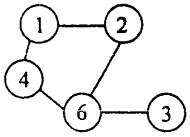
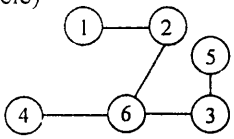
As I mentioned earlier in a greedy method a minimum cost spanning tree would build edge by edge. The next edge to include is chosen according to some optimization criterion. The criterion would be to choose an edge that results in a minimum increase in the sum of the costs of the edges so far included. These are two ways to interpret this criterion.

1. The set of edges so far selected form a tree. Thus, if A is the set of edges selected so far, then A forms a tree. The next edge (u, v) to be included in a is A minimum cost edge not in A with the property the A ∪ { (u, v)} is also a tree. This selection criterion will result in a minimum cost spanning tree. This method is known as Prim's algorithm.
2. In the second interpretation the edges of the graph are considered in non decreasing order of cost. The set T of edges so far selected for the spanning tree be such that it is possible to complete T into a tree. Thus T many not be a tree at all stages in the algorithm. T can be completed into a tree if and only if there are no cycles in T. This method is known as Kruskal's algorithm.

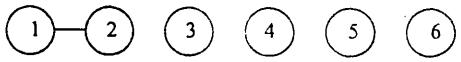
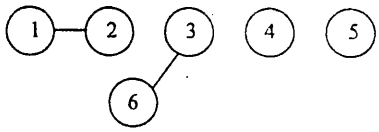
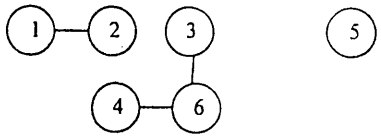
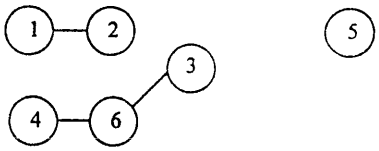
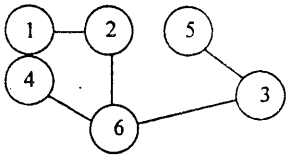
Consider the following graph

The following is the stages in Prim's algorithm to find the minimum cost spanning tree of the given graph.



Edge	Cost	Spanning tree
(1, 2)	10	
(2, 6)	25	
(2, 6)	15	
(6, 4)	20	
(1, 4)	reject (cycle)	
(3, 5)	35	

The following is the stages in Kruskal's algorithm to find the minimum cost spanning tree of the same graph.

Edge	Cost	Spanning tree
		
(1, 2)	10	
(3, 6)	15	
(4, 6)	20	
(2, 6)	25	
(1, 4)	30	(reject) (creating cycle)
(3, 5)	35	

Kruskal's algorithm to find the minimum cost spanning tree :
E set of edges in the graph which is sorted in ascending order.

Algorithm

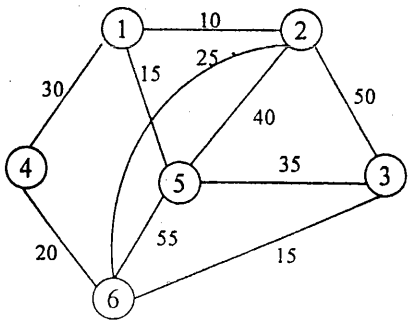
T ← ϕ {empty set}
While T contains fewer then n – 1 edges and E ≠ ϕ do
begin
chose an edge (v, w) from E of lowest cost delete (v, w) from E
if (v, w) does not create a cycle in T then add (v, w) to T.
else discard (v, w)
end ;

Shortest Path Problem

In a weighted graph, it is frequently desired to find the shortest path between two nods s and t. The shortest path is defined as a path from s to t such that the sum of the weights of the edges on the path is minimized.

The following is the algorithm due to Dijkstra, to find the shortest distance between two nodes of a graph.

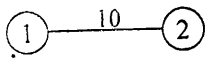
Minimum Spanning Tree Using Prim's Algorithm



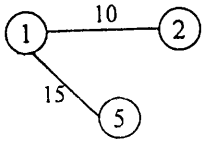
E = {(1, 2, 10), (1, 5, 15), (3, 6, 15), (4, 6, 20), (2, 6, 25), (1, 4, 30), (3, 5, 35), (2, 5, 40), (2, 3, 50), (5, 6, 55)}

Adding successive edges in the spanning tree is as follows :

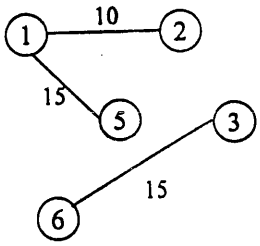
(1) (1, 2, 10)



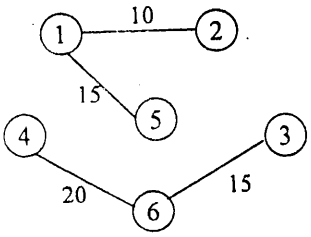
(2) (1, 5, 15)



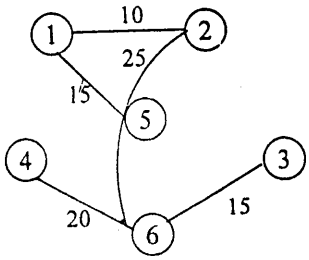
(3) (3, 6, 15)



(4) (4, 6, 20)

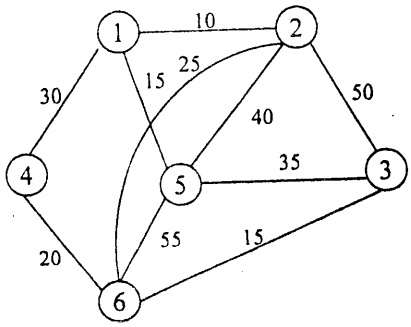


(5) (2, 6, 25)



Next edges (1, 4, 30), (3, 5, 35), (2, 5, 40), (2, 3, 50), (5, 6, 55) are discarded from the tree because they result in cycle.

Minimum Spanning Tree, Using Prim's Algorithm :



(1) Process vertex 1

d					
0	α	α	α	α	α

p					
0	0	0	0	0	0

x [1]

d and p change to :

d					
0	10	α	30	15	α

p					
0	1	0	1	1	α

x [1]

(2) Process vertex 2

d and p will change to

d					
0	10	50	30	15	25

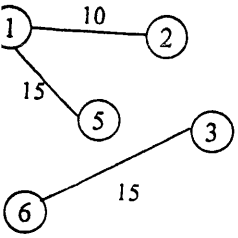
p					
0	1	2	1	1	2

x [1, 2]

20), (2, 6, 25), (1, 4, 30), (3, 5, 35),

s :

(3) (3, 6, 15)



, (5, 6, 55) are discarded from

x
[1]

x
[1]

x
1, 2]

(3) Process vertex 5
d and p will change to :

d					
0	10	35	30	15	25

p					
0	1	5	1	1	2

x
[1, 2, 5]

(4) Process vertex 6 :

d					
0	10	15	20	15	25

p					
0	1	6	6	1	2

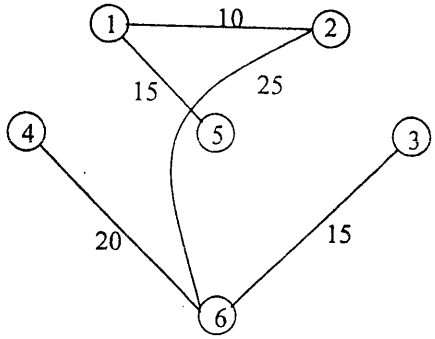
x
[1, 2, 5, 6]

(5) Process vertex 3 (d and p does not change)

d					
0	10	15	20	15	25

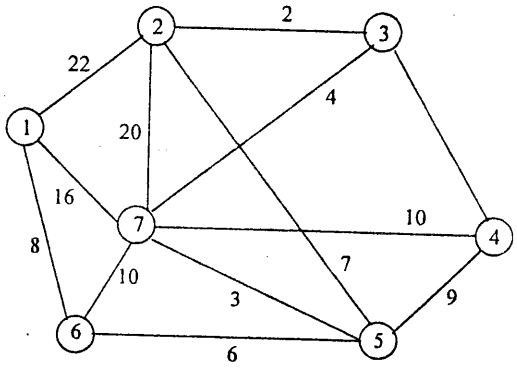
p					
0	1	6	6	1	2

x
[1, 2, 5, 6, 3]



Minimum Span Tree

Shortest distance between vertex 1 to vertex 4 using Dijkstra's Algorithm.



(1) Process vertex 1
d and p array are as follows :

d						
0	α	α	α	α	α	α

p							
0	0	0	0	0	0	0

x
[1]

dc
ϕ

d and p will change as follows :

d						
.0	22	α	α	α	8	16

p							
0	1	0	0	0	1	1

x
[1]

dc
ϕ

(2) Process vertex 6. dc = d[6] = 8.
d and p will change as :

d						
0	22	α	α	14	8	16

p							
0	1	0	0	6	1	1

x
[1, 6]

(3) Process vertex 5 dc = d[5] = 14.

d						
0	21	α	23	14	8	16

p						
0	5	0	5	6	1	1

x = [1, 6, 5]

(4) Process vertex 7 dc = d[7] = 16

d						
0	21	20	23	14	8	16

p						
0	5	7	5	6	1	1

x = [1, 5, 6, 7]

(5) Process vertex 3 dc = d[3] = 20
d and p does not change.

d						
0	21	20	23	14	8	16

p						
0	5	7	5	6	1	1

x = [1, 3, 5, 6, 7]

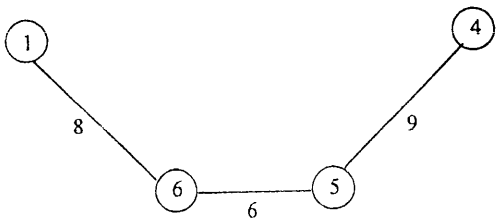
(6) Process vertex 2 dc = d[2] = 21
d and p does not change.

d						
0	21	20	23	14	8	16

p						
0	5	7	5	6	1	1

x = [1, 2, 3, 5, 6, 7]

Therefore shortest distance between vertex 1 and 4 = d[4] = 23.
The shortest distance is given by P array.
P[4] = 5. Therefore vertex 4 is connected to 5.
P[5] = 6 Therefore vertex 5 is connected to 6.
P[6] = 1 where 1 is source vertex.



8 Queens Problem

This is a classic combinatorial problem to place eight queens on an 8 × 8 chessboard so that no two "attack", that is so that no two of them are on the same row, column or diagonal.

For solving this problem, using a design strategy known as backtracking.
In order to use the back trace method, the desired solution must be expressible as an n–triple $(x_1, x_2, \ldots, x_n)$.

In 8 queen's problem, the result or solution can be expressed as $(x_1, x_2, \ldots, x_8)$ where is the position of the I queen, x_2 is the position of the II queen and so on. Queens are numbered from 1 to 8 and without losing generality we are assuming that Queen I will be placed in row 1, queen 2 will be placed in 2nd row and so on.
So x1 will be any value from 1 to 8 since there are 8 columns in one row.
x2 will be any value from 1 to 8 and so on.

To solve this problem, first we are finding a place for the first queen. Let it be 1. i.e., x[1] = 1.

Then try to find the place of 2nd queen. Since queen 1 is already placed in 1st column the second queen can not be placed in 1st or 2nd columns. Since if we are placing queen 2 in column 1, then queen 1 and queen 2 will be in the same column. That is not allowed. Similarly if we are placing queens in 2nd column then queen 1 and queen 2 will be in a diagonal. That is also not allowed. So we can place 2nd queen in 3rd column. So X[2] = 3. If we are not able to find a proper location for a queen then we will backtrack and try to adjust the position of the previous queen. If it is not possible, we will be backtrack once again.

Algorithm

To find a proper position of queen we are using one procedure. PLACE

```
int x[20];

int place(int k)
{
int i=1,flag = 1;

while (i<=k-1 &
  if (x[i] == x[k
    flag = 0;
  else
    i++;

return flag;
}

void main()
{
 int k,i;
 k=1; x[k] = 0;
 while (k > 0)
 {
  x[k]++;
  while (x[k] <
   x[k]++;

  if (x[k] <= 8
  {
    if (k == 8)
    {
    for (i=1;i<
     printf("%
    printf("\n"
    }
    else
    {
    k++;
    x[k] = 0;
    }
  }
   else
    k--; /* ba
}

}
```

Note : ABS (
using this we
from 1 to K -

X [i] – – colu
X[k] – – colu
i – – – – row
k – – – – rov

The above fi
k^th row of th
Using the pr

x

1 | 1 [1, 6, 5]

x

1 | 1 [1, 5, 6, 7]

x

| 1 [1, 3, 5, 6, 7]

x

| 1 [1, 2, 3, 5, 6, 7]

23.

n 8 × 8 chessboard so that no
umn or diagonal.

king.

expressible as an n–triple (x_1,

(x_1, x_2, …, x_8) where is the
Queens are numbered from 1
be placed in row 1, queen 2

ow.

Let it be 1. i.e., x[1] = 1.

d in 1st column the second
queen 2 in column 1, then
wed. Similarly if we are
diagonal. That is also not
we are not able to find a
e position of the previous

```
int x[20];


int place(int k)
{
int i=1,flag = 1;

while (i<=k-1 && flag)
  if (x[i] == x[k] || abs(x[i]-x[k]) == abs(i-k))
     flag = 0;
  else
    i++;

return flag;
}


void main()
{
 int k,i;
 k=1; x[k] = 0;
 while (k > 0)
 {
  x[k]++;
  while (x[k] <= 8 && ! place(k))
   x[k]++;

  if (x[k] <= 8)
  {
    if (k == 8)
    {
    for (i=1;i<=8;i++)
     printf("%d  ",x[i]);
    printf("\n");
    }
    else
    {
    k++;
    x[k] = 0;
    }
  }
  else
    k--; /* backtrack */
}

}
```

Note : ABS (X [i] – X[K]) = ABS (i – k)
using this we are checking whether the queen K is coming in diagonal with any of the queens from 1 to K – 1.

X [i] – – column position of i^{th} queen.
X[k] – – column position of k^{th} queen.
i – – – – row position of i^{th} queen.
k – – – – row position of k^{th} queen.

The above function PLACE will return true if it can find out a proper place for k^{th} queen in the k^{th} row of the 8 × 8 chess board. Otherwise it will return false.
Using the procedure PLACE now we will write a procedure for solving 8 queen problem.

Graph coloring problem

let a be a graph and m be a given positive integer. We want to discover it the nodes of G can be colored in such a way that no two adjacent nodes have the same color get only m colors are use of this is termed as the m colorability decision problem.

For solving this problem we are using back tracking technique. Here we can express the solution in n triple form. i.e., $(x_1, x_2, \ldots, x_n)$.

Let G be a graph of n nodes. Then the solution of the above problem can be written as $(x_1, x_2, \ldots, x_n)$ where x_1 is the color assigned to node 1, x_2 is the color assigned to node 2 and so on.

Following algorithm finds a solution for a graph having V vertices and E egdes. Given M different colors the algorithm finds whether the graph is m–colorable or not. It also prints all possibilities in which the graph can be colored using M different colors.

Algorithm for graph color

{ Consider a graph G(V,E) having V vertices and E edges such that G[v1 , v2] = 1 if there is an edge connecting v1 and v2 else G[v1, v2] = 0}
{M different colors are given and we want to color the graph in such a way that the adjacent vertices should not have same color}

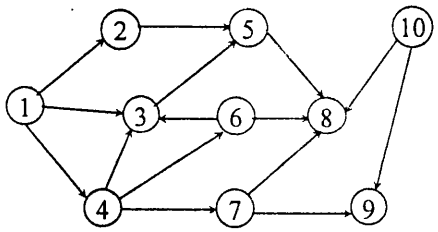
{ X is array of V elements such that x[k] stores color of Kth vertex}
{ notcolored(k) is routine which checks whether vertex k is properly colored }
{ notcolored(k) return true if vertex k is not colored properly else it return false}

```
k = 1 ; x[k] = 0
while (k > 0)
{
    x[k] = x[k] + 1;
    while (x[k] <= m and notcolored(k) )
        x[k] = x[k] + 1;

    if (x[k] <= m)
    {
        if (k= = V)
            display the array x { X has the solution}
        else
        {
            k = k + 1;
            x[k] = 0;
        }
    }
    else
        k = k – 1; { backtrack}
}
```

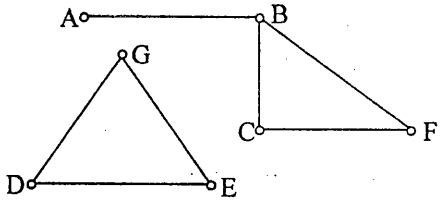
Graded Questions :

1. Define the following terms with respect to spanning trees. Give example. **[M-06]**
   (a) Tree edge (b) cross edge (c) forward edge (d) backward edge.
2. Explain Topological Sorting. **[M-06, 07, D-07]**
3. Explain different methods of representing a graph in computer system. **[N-05]**
4. What are the different ways available to represent graphs in memory ? What are the applications of Graph? **[M-04, M-05]**
5. State applications of graph theory. **[M-06]**
6. Define graph and write a note on application of graphs. **[N-05]**
7. Design and develop shortest path algorithm.
8. Write an algorithm for breadth first traversal of graph. Explain with suitable example. **[M-06]**
9. W ite a prog
10. E plain the
    (a Minimur
    (b Depth Fi
    (c Breath Fi
11. W te a funct
12. G e algori
    B dth Firs
    Fi t Searc
    sp ning t
    Co sider the
13. E lain DFS
14. W t is grap
    G e the algo
15. Ex lain Link
16. E lain direc
17. D ne graph
    g h.
18. W t is trans
    g hs.
19. U g Dijkstr
20. What is a spar

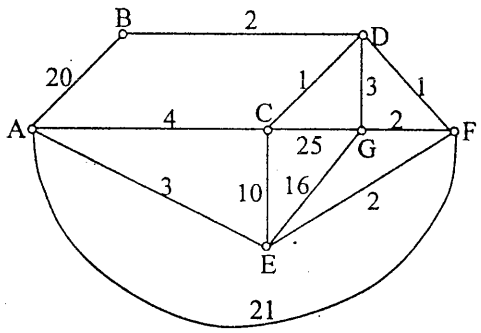
9. Write a program to give Array representation of a graph. [N-06]
10. Explain the following terms w.r.t. graphs :
    (a) Minimum Lost Spanning Tree.
    (b) Depth First Search
    (c) Breath First Search
11. Write a functions to implement DFS and BFS graph searching methods. [D-07]
12. Give algorithm / pseudo code for Breadth First Search and Use Breadth First Search technique to give a spanning tree for the following. Consider the node 1 as source. [M-06]



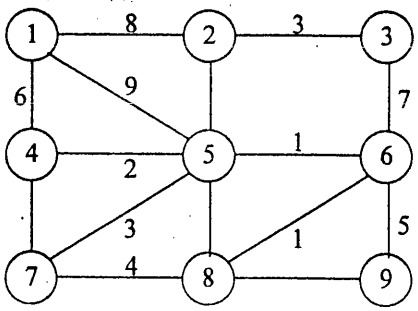
13. Explain DFS traversal of graph with example. What are different applications of graph ? [N-06]
14. What is graph? What are the different applications of graph? Explain DFS traversal of graph. Give the algorithm of DKS. [M-07]
15. Explain Linked representation of graph in brief.
16. Explain directed acyclic graph in brief.
17. Define graph. What is a directed graph ? Give a data structure for implementing directed graph.
18. What is transitive closure ? Using adjacency, find the transitive closure for the following graphs.



19. Using Dijkstra's algorithm, find the shortest path in the following graph from vertex A to G.



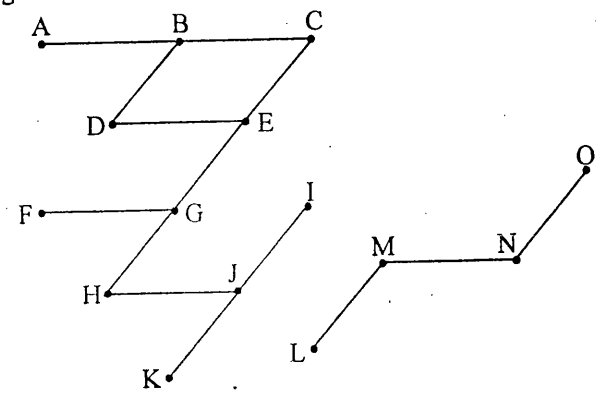
20. What is a spanning tree ? Find the minimum cost spanning tree for the following :



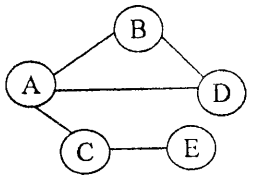
21. What is minimum spanning tree ? What are its application ? Explain prim's algorithm with example. **[M-04,N-04, N-06]**

22. State 10 popular tree applications and Draw various trees types. **[M-06]**

23. For the graph in figure generate the transitive closure.

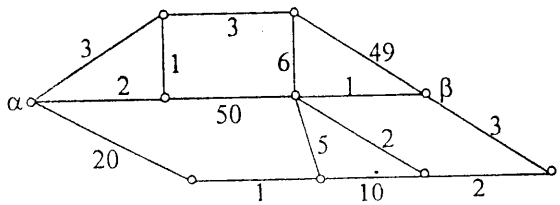


24. For a graph G = (V, E). Write a program to find partition of V such that no two vertices in a subject are adjacent to each other equation – **[M-04]**

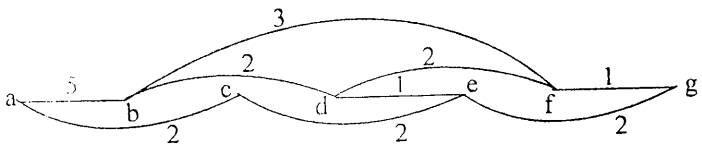


Output for given graph in
{ {A, E }, { B, C } , { D} }

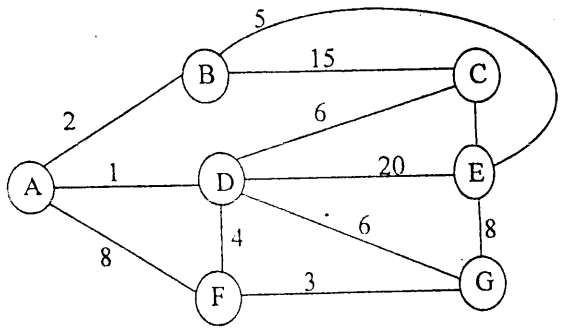
25. (a) Find the shortest path between α and β.



(b) Find minimum cost spanning tree using Kruskals algorithm for the following :



26. Using DIJKSTRA'S shortest path algorithm find shortest path from A to all vertices.



27. Write short
28. Design and
29. Write non–r
30. (a) Explain starting

(b) Write a
(c) Give Di

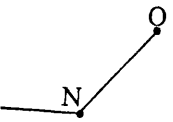
31. Give Prim' spanning tr

32. To find the

33. Write shor
34. Does eith Justify yo suitable al

35. Give an every ed example.

? Explain prim's algorithm with
[M-04,N-04, N-06]
es. [M-06]



V such that no two vertices
[M-04]

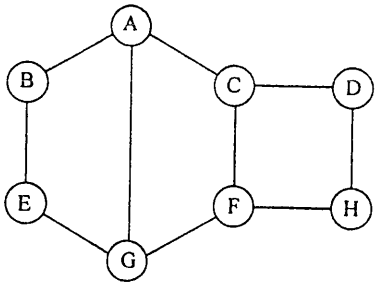
r given graph in
, { B, C } , { D} }

he following :

> g

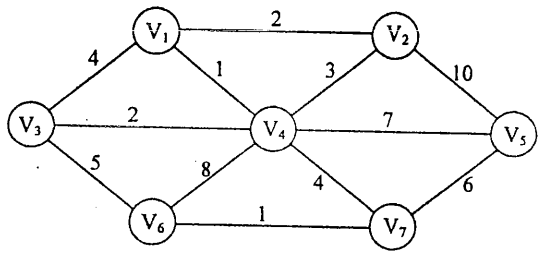
A to all vertices.

27. Write short notes on Kruskal's Algorithm. [N-06]
28. Design and develop Warshall's algorithm.
29. Write non–recursive C program for Breadth First traversal of a graph. [M-04]
30. (a) Explain DFS algorithm. Give the breath first traversal of the graph shown below, starting from vertex A. [M-03]

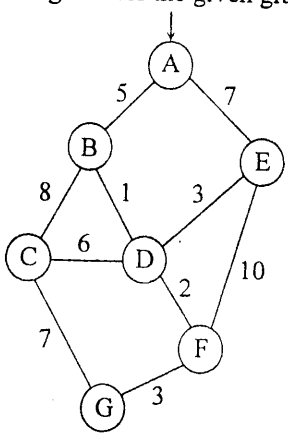


(b) Write a 'C' program for depth first traversal of graph. [N-04]
(c) Give Dijkstra's algorithm for shortest path. Explain the same with suitable example. [M-03, D-03]

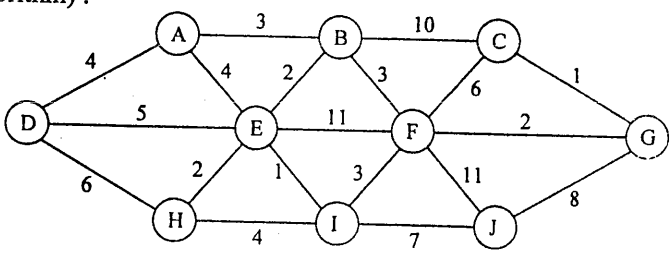
31. Give Prim's algorithm for minimum spanning tree. Use the same to find a minimum spanning tree for graph below : [M-03]



32. To find the minimum spanning tree for the given graph using PRIMS's algorithm. [D-07]

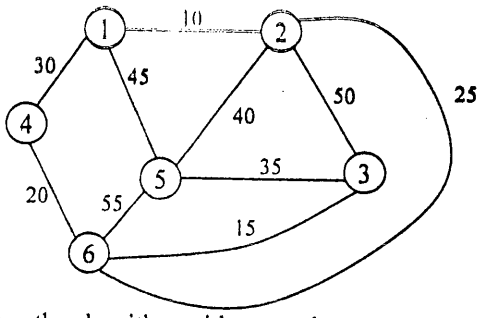


33. Write short note on Euler circuits. [M-03]
34. Does either Prim's or Kruskal's algorithm work if there are negative edge weights ? Justify your answer. Find the minimum spanning tree for the graph shown below (use suitable algorithm) : [D-03]



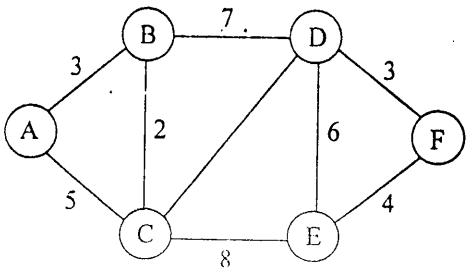
35. Give an algorithm to find in an undirected (connected) graph, a path that goes through every edge exactly once in each direction. Explain the same algorithm with suitable example. [D-03]

36. Explain any two applications of minimum spanning trees and for the following graph find minimum spanning tree using Kruskal's algorithm. **[M-05]**



37. Explain all pairs shortest paths algorithm with example. **[M-05, N-05]**

38. To find shortest path from node A to F. Also write a program for shortest path algorithm. **[M-07]**



39. Explain Prim's algorithm with example. How it is different than Kruskal;s algorithm ? **[N-05]**

40. Write Short note on Graph Traversal techniques. **[N-06]**

❑❑❑❑❑❑

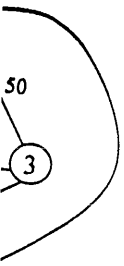
Progr m for dep

```
# incl e <stdio.h
int g[1 ] [10], vis
/*  cr  egraph ( )
void d  (int k)
{
    int  ;
    vis ed [k] = 1
    prin f ("%d vi
    for  i = 1 ; i <
    {
        f (g [k] [i
        if (visi
            dfs
    }
}  /*  end of dfs
void main ( )
{
    int i ;
    creategraph ( )
    for (i = 1; i <
        if (visited
            dfs (i)
}  /*  end main-(
```

Progra m for Pri

```
# include <stdio.h
int g[20] [20], vis
int v, e;

void creategraph(
{
    int i, j, a, b, w
    printf("Enter
    scanf("%d", &
    printf("Enter
    scanf("%d", &

    for (i = 1; i <
    for (j = 1; j <
    g[i] [j] = 0;

   for(i = 1; i < =
   {
      p[i] = visite
      d[i] = 3276
   }
   for( i = 1; i <
   {
   printf("Enter
   scanf("%d %
   g[a] [b] = g[b
   }
```

trees and for the following graph find [M-05]



gram for shortest path algorithm. [M-05, N-05] [M-07]

F

Kruskal;s algorithm ? [N-05] [N-06]

Program for depth first search

```
# include <stdio.h>
int g[10] [10], visited [10], v, e ;
/*  creategraph ( ) function should be the same as written for Prim's algorithm  */
void dfs (int k)
{
    int i ;
    visited [k] = 1 ;
    printf ("%d visited \n", k) ;
    for (i = 1 ;  i <= v ;  i++)
    {
        if (g [k] [i] = = 1)  /* k and i are adjacent  */
            if (visited [i] = = 0)    /*  i is not visited  */
                dfs (i) ;     /*  call dfs for vertex i     */
    }
}  /*  end of dfs  */
void main ( )
{
    int i ;
    creategraph ( ) ;
    for (i = 1;  i <= v ;  i++)
        if (visited [i] = = 0)
            dfs (i) ;
}  /*  end main ( )    */
```

Program for Prim's algorithm

```
# include <stdio.h>
int g[20] [20], visited[20], d[20], p[20];
int v, e;

void creategraph( )
{
    int i, j, a, b, w;
    printf("Enter number of vertices");
    scanf("%d", &v);
    printf("Enter number of edges");
    scanf("%d", &e);

    for (i = 1; i < = v; i++)
    for (j = 1; j < = v; j++)
    g[i] [j] = 0;

    for(i = 1; i < = v; i++)
    {
        p[i] = visited[i] = 0;
        d[i] = 32767;
    }
    for( i = 1; i < = e; i++)
    {
    printf("Enter edge a, b and weight w");
    scanf("%d  %d %d", &a, &b, &w);
    g[a] [b] = g[b] [a] = w;
    }
```

```
}
void prim( )
{
   int current, totalvisited, mincost, i;

   current = 1; d[current] = 0;
   totalvisited = 1;
   visited[current] = 1;

   while (totalvisited != v)
   {
   for ( i = 1; i < = v; i++)
    {
        if (g[current] [i] ! = 0)
        if (visited[i] = = 0)
        if (d[i] > g[current] [i]
        {
       d[i] = g[current] [i];
       p[i] = current;
         }
         }
    min = 32767;
    for( i = 1; i < = v; i++)
    { .
   if  (visited[i] = = 0)
   if  (d[i] < min )
   {
      min = d[i];
      current = i;
   }
     }
    visited [current] = 1;
    totalvisited ++;
  } /* end of the while */

  mincost = 0;
  for( i = 1; i < = v; i++)
  mincost  + = d[i];
  printf("Minimum cost = %d \n", mincost);
  printf("Minimum span tree is \n");

  for( i = 1; i < = v; i++)
  printf("vertex  %d is connected to %d\n", i, p[i]);
} /* end of prim */

void main ( )
{
   creategraph( );
   prim ( );
}
```

Program for dijkstra's algorithm

```
# include <stdio.h>
int g[20] [20], visited[20], d[20], p[20];
int v,e;

void creategraph( )
{
```

```
   /* same as in prim
}

void dijkstra( )
{
    int source, dest
    printf("Enter so
    scanf("%d  %d
    current = sourc
    d[source] = 0;
    dc = d[current]

    while (visited[
    {
    for (i = 1; i < =
    {
        if ( g[curre
        if (visited
        if (d[i] > g
        {
       d[i] = g[cu
       p[i] = curr
        }
         } /* end
     min = 32767;
     for( i = 1; i <
     { .
        if ( visited
        if ( d[i] <
        {
        min = d[i
        current =
         }
     } / *  end of

     dc = d[curre
     visited[curre

    } / *  end of v

     printf("Shor
     printf("%d \
     printf("path
     l = dest;
     do
       {
      x = p[l];
      printf ("V
      l = x;
       }
     while ( l ! =
   }  /* end of di

   Program for
   int m , n ,x[10
   mcolor(k)
   int k;
   {
   int i ;
```

```
  /* same as in prims program */
}

void dijkstra( )
{
    int source, dest, current, i, dc, l, x;
    printf("Enter source & destination vertex");
    scanf("%d  %d", &source, &dest);
    current = source;
    d[source] = 0; visited[source] = 1;
    dc = d[current];

    while (visited[dest] ! = 1 )
    {
   for (i = 1; i < = v; i++)
   {
        if ( g[current] [i] ! = 0)
        if (visited [i] = = 0)
        if (d[i] > g[current] [i]  + dc)
        {
       d[i] = g[current] [i] + dc;
       p[i] = current;
        }
         } /*  end of for  */
   min = 32767;
   for( i = 1; i < = v; i++)
   {
       if ( visited[i] = = 0)
       if ( d[i] < min)
       {
       min = d[i];
       current = i;
        }
   } / *  end of for  */

   dc = d[current]
   visited[current] = 1;

  } / *  end of while  */

   printf("Shortest distance from %d  to %d \n", source, dest);
   printf("%d \n",  d[dest]);
   printf("path = \n");
   l = dest;
   do
     {
    x = p[l];
    printf ("Vertex  %d connected to %d \n", l, x);
    l = x;
      }
   while ( l ! = source);
}  /* end of dijkstra  * /
```

Program for Graph Colouring

```
int m , n ,x[10],graph[10][10];
mcolor(k)
int k;
{
int i ;
```

```
        while(1)
        {
         callnextvalue(k);
         if( x[k] == 0 ) break ;
         if( k == n )
           {
           for( i = 1 ; i <= n ; i++ )
            printf("%4d",x[i]);
            printf("\n");
           }
         else
          mcolor(k+1);
        }
       }

      callnextvalue(k)
      {
       int j ;
       while(1)
        {
          x[k] = (x[k] + 1)  %  (m+1);
          if( x[k] == 0 ) return;
          for( j = 1;j<=n;j++)
          if( graph[k][j]  &&  x[k] == x[j] )
            break;

       if( j == n + 1 ) return;
       }
      }

      main()
      {
       int i ;
       int e ;
       int a , b ;
       printf("Enter the vertices\n");
       scanf("%d",&n);
       printf("Enter no of edges\n");
       scanf("%d",&e);
       for(i=1;i<=e;i++)
       {
       printf("Enter edge\n");
       scanf("%d%d",&a,&b);
       graph[a][b] = 1 ;
       graph[b][a] = 1 ;
       }
      for(i=1;i<=n;i++)
         x[i] = 0;
       printf("Enter no of colors\n");
       scanf("%d",&m);
      mcolor(1);
}
```

/* Program to convert infix to postfix*/

```
#include <stdio.h>
int icp(char x)
{
switch (x)
{
case '*' :
case '/' : return 4;
        break;
case '+' :
case '-' : return 3;
        break;
}
}

int isp(char x)
{
switch (x)
{
case '*' :
case '/' : return 4;
        break;

case '+' :
case '-' : return 3;
        break;
case '(' : return 2;
        break;
case '~' : return  1
        break;
}
}

void main()
{
 char expr[15], p
 int i, j = 0, top=-
 printf("Enter a ir
 scanf("%s", expr

 /*push a symbol
 top++;
 stack[top] = '~';
 for (i=0 ; expr[i]
 {
  ch = expr[i];
  if (ch == '*' || c
  {
   while (icp(ch)
   {
   p[j] = stack[to
   top--;
   j++;
   }
   /*push ch on s
   top++;
   stack[top] = c
  }
  else
  if (ch == '(')
  {
   top++;
   stack[top] = c
  }
  else
  if (ch == ')')
```

```
 case '*' :
 case '/' : return 4;
        break;
 case '+' :
 case '-' : return 3;
        break;
 }
 }

 int isp(char x)
 {
 switch (x)
 {
 case '*' :
 case '/' : return 4;
        break;

 case '+' :
 case '-' : return 3;
        break;
 case '(' : return 2;
        break;
 case '~' : return  1;
        break;
 }
 }

 void main()
 {
  char expr[15], p[15] , stack[15], ch;
  int i, j = 0, top=-1;
  printf("Enter a infix expression ");
  scanf("%s", expr);

  /*push a symbol ~ on stack */
  top++;
  stack[top] = '~';
  for (i=0 ; expr[i] != '\0' ; i++)
  {
   ch = expr[i];
   if (ch == '*' || ch == '/' || ch == '+' || ch == '-')
   {
    while (icp(ch) <= isp(stack[top]))
    {
    p[j] = stack[top];
    top--;
    j++;
    }
    /*push ch on stack */
    top++;
    stack[top] = ch;
  }
  else
  if (ch == '(')
  {
   top++;
   stack[top] = ch;
  }
  else
  if (ch == ')')
```

```
    {
     while (stack[top] != '(')
     {
     p[j] = stack[top];
     top--;
     j++;
     }
     /*remove ( from stack */
     top--;
    }
    else
    {
    /* ch is an operand */
     p[j] =ch;
    j++;
    }
 } /*end for */
 /*pop remaining operators from stack */
 while (stack[top] != '~')
 {
 p[j] = stack[top];
 j++;
 top--;
 }
      .
p[j] = '\0';
printf("postfix expression is %s \n" , p);
}
```

/*program to evalute a postfix expression */
/*the expression has one digit operands and possible operators are *,+,/,- */

```
#include <stdio.h>
void main()
{
    float stack[10],op1,op2,r,finalresult;
    char expr[15], ch;
    int i, top=-1;

    printf("Enter a postfix expression ");
    scanf("%s", expr);

    for(i=0 ; expr[i] != '\0' ; i++)
    {
      ch = expr[i]; /*ch is next token of expr */
      /* check  if ch is an operator */
      if (ch == '*' || ch=='/' || ch == '+' || ch == '-')
      {
       op1 = stack[top];
       top--;
       op2 = stack[top];
       top--;

      switch (ch)
      {
      case '*' :r=op2*op1;
          break;
      case '/' :r=op2/op1;
          break;
```

```
          case '+' :r=op2
              break;
          case '-' :r=op2
              break;
          }
          /*push result
          top++;
          stack[top] = r;
         }
         else
         {
          /*ch is an oper
          /* push the nu
          top++;
          stack[top] = ch
          }
      } /*end for */

      finalresult = stack
      printf("result is %
}
```

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

struct node
{
 char data;
 struct node *left,*ri
};

void inorder(struct n
{
 if (p!=NULL)
 {
  inorder(p->left);
  printf("%c" , p->
  inorder(p->right)
 }
}

void preorder(struc
{
 if (p!=NULL)
 {
  printf("%c" , p->
  preorder(p->left)
  preorder(p->righ
 }
}

void main()
{
 char expr[15] , ch
 int i,top=-1 ;
 struct node *new
 struct node *stack
```

```
        case '+' :r=op2+op1;
            break;
        case '-' :r=op2-op1;
            break;
        }
        /*push result on stack */
        top++;
        stack[top] = r;
       }
       else
       {
       /*ch is an operand */
       /* push the numeric value of ch on the stack */
       top++;
       stack[top] = ch - 48;
       }
    } /*end for */

    finalresult = stack[top];
    printf("result is %f\n" , finalresult);
}
```

/*program to convert postfix to infix expression */

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

struct node
{
 char data;
 struct node *left,*right;
};

void inorder(struct node *p)
{
 if (p!=NULL)
 {
  inorder(p->left);
  printf("%c" , p->data);
  inorder(p->right);
 }
}

void preorder(struct node *p)
{
 if (p!=NULL)
 {
  printf("%c" , p->data);
  preorder(p->left);
  preorder(p->right);
 }
}

void main()
{
 char expr[15] , ch;
 int i,top=-1 ;
 struct node *newrec , *op1, *op2 ,*root;
 struct node *stack[10];
```

```
 printf("Enter a postfix expression ");
 scanf("%s", expr);
 for(i=0 ; expr[i] != '\0' ; i++)
 {
  ch = expr[i];
  /* check  if ch is an operator */
  if (ch == '*' || ch=='/' || ch == '+' || ch == '-')
  {
    op1 = stack[top];
    top--;
    op2 = stack[top];
    top--;
    newrec = (struct node *) malloc(sizeof(struct node));
    newrec->data = ch;
    newrec->left = op2;
    newrec->right = op1;
    /*push newrec on stack */
    top++;
    stack[top] = newrec;
  }
  else
  {
   /*ch is an operand */
    newrec = (struct node *) malloc(sizeof(struct node));
    newrec->data = ch;
    newrec->left = NULL;
    newrec->right = NULL;
    /*push newrec on stack */
    top++;
    stack[top] = newrec;
  }
 } /*end for */

 root = stack[top];

 /* call inorder traversal method to print infix expression */
 printf("infix expression is \n");
 inorder(root);
 printf("\n");
 /* call preorder traversal method to print prefix expression */
 printf("prefix expression is \n");
 preorder(root);
 printf("\n");
} /*end of main */
```

/*program to convert prefix to infix expression */

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
struct node
{
 char data;
 struct node *left,*right;
};

void inorder(struct node *p)
{
 if (p!=NULL)
 {
  inorder( ->left);
  printf(" c" , p->d
  inorder( ->right);
 }
}

void post rder(struct
{
 if (p!=N JLL)
 {
  postor ler(p->left)
  postorder(p->righ
  printf("%c" , p->
 }
}

void main()
{
 char expr[15] , ch;
 int i,top=-1, len ;
 struct node *newr
 struct node *stack
 printf("Enter a pre
 scanf("%s", expr);
 len = strlen(expr);
 for(i=len-1 ; i >=
 {
  ch = expr[i];
  /* check  if ch is
  if (ch == '*' || ch
  {
    op1 = stack[to
    top--;
    op2 = stack[to
    top--;
    newrec = (stru
    newrec->data
    newrec->left =
    newrec->right
    /*push newrec
    top++;
    stack[top] = n
  }
  else
  {
   /*ch is an oper
    newrec = (str
    newrec->data
    newrec->left
    newrec->righ
    /*push newre
    top++;
    stack[top] =
  }
 } /*end for */

 root = stack[top

 /* call inorder t
```

```
  {
   inorder(p->left);
   printf("%c" , p->data);
   inorder(p->right);
  }
 }

 void postorder(struct node *p)
 {
  if (p!=NULL)
  {
   postorder(p->left);
   postorder(p->right);
   printf("%c" , p->data);
  }
 }

 void main()
 {
  char expr[15] , ch;
  int i,top=-1, len ;
  struct node *newrec , *op1, *op2 ,*root;
  struct node *stack[10];
  printf("Enter a prefix expression ");
  scanf("%s", expr);
  len = strlen(expr);
  for(i=len-1 ; i >= 0 ; i--)
  {
   ch = expr[i];
   /* check  if ch is an operator */
   if (ch == '*' || ch=='/' || ch == '+' || ch == '-')
   {
    op1 = stack[top];
    top--;
    op2 = stack[top];
    top--;
    newrec = (struct node *) malloc(sizeof(struct node));
    newrec->data = ch;
    newrec->left = op1;
    newrec->right = op2;
    /*push newrec on stack */
    top++;
    stack[top] = newrec;
   }
   else
   {
    /*ch is an operand */
    newrec = (struct node *) malloc(sizeof(struct node));
    newrec->data = ch;
    newrec->left = NULL;
    newrec->right = NULL;
    /*push newrec on stack */
    top++;
    stack[top] = newrec;
   }
  } /*end for */

  root = stack[top];

  /* call inorder traversal method to print infix expression */
```

```
 printf("infix expression is \n");
 inorder(root);
 printf("\n");
 /* call preorder traversal method to print prefix expression */
 printf("postfix expression is \n");
 postorder(root);
 printf("\n");
} /*end of main */
```

Program for Radix Sort

```
#include <stdio.h>

/*this program will sort array of 3 digit numbers */

void radix(int a[],int n , int d)
{
 /*d are the number of digits in each number */
 int  bucket[50][10] , count[10], digit , i , exp =1;
 int row , col , m;

 for (m = 1; m<=d ; m++)
 {
  /*initialize count array */
  for (i=0 ; i<=9 ; i++)
    count[i] = -1;

  /*insert array elements into buckets */
  for (i=0 ; i<=n-1 ; i++)
  {
   digit = (a[i] / exp) % 10;
   count[digit]++;
   row = count[digit];
   col = digit;
   bucket[row][col] = a[i];
  }

  /*copy all buckets back into the array a */
  i = 0;
  for (col=0 ; col<=9 ; col++)
  {
   if (count[col] != -1)
   {
   for (row = 0 ; row <= count[col] ; row++)
   {
    a[i] = bucket[row][col];
    i++;
   }
   }
  }
  exp = exp * 10;

 } /*go back to major for loop */
} /*end radix sort */


void main()
{
 int a[50] , n ,i;
 printf("enter n ");
```

```
scanf("%d" , &n);

for (i=0 ; i<=n-1;
{
 printf("enter ele
 scanf("%d" , &a
}
radix(a,n,3);
/*now print sorte
for (i=0;i<=n-1;i-
 printf("%d " , a
printf("\n");
}
```

Program for Ind

```
#include <stdio.h>
int a[100] , index[
void createindex(
{
 int i;
 r = -1;
 for (i=0;i<=n-1;
 {
  r++;
  index[r][0] = i
  index[r][1] = a
 }
}

void indexsearch
{
 int x,address , i
 printf("enter va
 scanf("%d" , &
 if (x<a[0] || x>a
 {
  printf("value
  return ;
 }

 i = 0;
 while(index[i]
  i++;

 i--;

 address = inde
 while(a[addre
  address++;

 address--;

 if (a[address]
  printf("value
 else
  printf("valu
}

void main()
```

```
    scanf("%d" , &n);

    for (i=0 ; i<=n-1; i++)
    {
     printf("enter element %d " , i+1);
     scanf("%d" , &a[i]);
    }
    radix(a,n,3);
    /*now print sorted array */
    for (i=0;i<=n-1;i++)
     printf("%d  " , a[i]);
    printf("\n");
  }
```

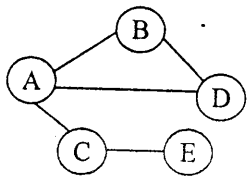
Program for Indexed Sequential Search

```
#include <stdio.h>
int a[100] , index[35][2] , r , n;
void createindex()
{
 int i;
 r = -1;
 for (i=0;i<=n-1;i+=3)
 {
  r++;
  index[r][0] = i;
  index[r][1] = a[i];
 }
}

void indexsearch()
{
 int x,address , i;
 printf("enter value to search ");
 scanf("%d" , &x);
 if (x<a[0] || x>a[n-1])
 {
  printf("value not found \n");
  return ;
 }

 i = 0;
 while(index[i][0] <= x && i<=r)
  i++;

 i--;

 address = index[i][1];
 while(a[address] <= x && address<=n-1)
  address++;

 address--;

 if (a[address] == x)
  printf("value found\n");
 else
  printf("value not found \n");
}

void main()
```

```
{
int x;

printf("Enter number of elements ");
scanf("%d", &n);
printf("enter sorted array\n");

for (x = 0 ; x<=n-1; x++)
{
 printf("Enter element %d " , x+1);
 scanf("%d", &a[x]);
}

createindex();
indexsearch();
}
```

Program for Partition a Graph

Q. For a graph G = (V, E). Write a program to find partition of V such that no two vertices in a subject are adjacent to each other equation – **[M-04]**



Output for given graph in
{ {A, E }, { B, C } , { D} }

Solution :

```
#include <stdio.h>
int g[20][20] , visited[20] , v ,e;
void creategraph()
{
 int i,j,a,b;
 printf("enter number of vertices ");
 scanf("%d" , &v);
 printf("enter number of edges ");
 scanf("%d" , &e);
 for (i=1;i<=v;i++)
 for (j=1;j<=v;j++)
 g[i][j] = 0;

 for (i=1;i<=e;i++)
 {
  printf("enter edge information (a,b) ");
  scanf("%d %d" , &a,&b);
  g[a][b] = g[b][a] = 1;
 }
} /*end creategraph */

void partition()
{
 int current , c ,i;
 current = 1;
 c = 0;
 while (c!= v)
 {
  printf("%d " , current);
  visited[current]=
  c++;
  for (i=1;i<=v;i++
  {
   if (g[current][i]
   {
   printf("%d " , i)
   visited[i] = 1;
   c++;
   }
  }

 printf("\n");
  while(visited[cu
   current++;
 }
} /*end partition *

void main()
{
 creategraph();
 partition();
}
```

Program for Bucl

```
#include <stdio.h>

/*this program wil

void bucket(int a[]
{

 /*d are the numbe

 int bucket[50][1(
 int row , col , m;

  /*initialize coun
  for (i=0 ; i<=9 ;
   count[i] = -1;

  /*insert array el
  for (i=0 ; i<=n-1
  {
   digit = (a[i]) %
   count[digit]++
   row = count[di
   col = digit;
   bucket[row][c
  }

  /*copy all buck
  i = 0;
  for (col=0 ; col
  {
   if (count[col]
   {
   for (row = 0 ;
   {
```

```
    visited[current]=1;
    c++;
    for (i=1;i<=v;i++)
    {
     if (g[current][i] == 0  && visited[i] != 1)
     {
     printf("%d " , i);
     visited[i] = 1;
     c++;
     }
    }

   printf("\n");
    while(visited[current] == 1  && current <= v)
     current++;
  }
 } /*end partition */

 void main()
 {
  creategraph();
  partition();
 }
```

Program for Bucket Sort

```
#include <stdio.h>

/*this program will sort array of 1 digit numbers */

void bucket(int a[],int n)
{

 /*d are the number of digits in each number */

 int  bucket[50][10] , count[10], digit , i;
 int row , col , m;

  /*initialize count array */
  for (i=0 ; i<=9 ; i++)
   count[i] = -1;

  /*insert array elements into buckets */
  for (i=0 ; i<=n-1 ; i++)
  {
   digit = (a[i])  % 10;
   count[digit]++;
   row = count[digit];
   col = digit;
   bucket[row][col] = a[i];
  }

 /*copy all buckets back into the array a */
 i = 0;
 for (col=0 ; col<=9 ; col++)
 {
  if (count[col] != -1)
  {
  for (row = 0 ; row <= count[col] ; row++)
  {
```

such that no two vertices
[M-04]

ven graph in
3, C } , { D} }

```
        a[i] = bucket[row][col];
        i++;
      }
      }
    }
} /*end bucket sort */


void main()
{

 int a[50] , n ,i;
 printf("enter n ");
 scanf("%d" , &n);

 for (i=0 ; i<=n-1; i++)
 {
  printf("enter element %d " , i+1);
  scanf("%d" , &a[i]);
 }

 bucket(a,n);

 /*now print sorted array */
 for (i=0;i<=n-1;i++)
  printf("%d  " , a[i]);

 printf("\n");
}
```

Program for Breadth first search

```
#include <stdio.h>
int g[20][20] , visited[20] , v ,e;
struct queue
{ int a[20], front , rear;
}q;
void creategraph()
{
 int i,j,a,b;
 printf("enter number of vertices ");
 scanf("%d" , &v);
 printf("enter number of edges ");
 scanf("%d" , &e);
 for (i=1;i<=v;i++)
 for (j=1;j<=v;j++)
 g[i][j] = 0;
 for (i=1;i<=e;i++)
 { printf("enter edge information (a,b) ");
  scanf("%d %d" , &a,&b);
  g[a][b] = g[b][a] = 1;
 }
} /*end creategraph */
void insert(int current , struct queue *q)
{
 q->rear++;
 q->a[q->rear] = current;
 if (q->front == -1)
  q->front = 0;
}
```

```
int delqueue(struct
{ int x;
 x = q->a[q->front]
 if (q->front == q->
  q->front = q->re
 else
  q->front++;
 return x;
}
int found(int curre
{ int i;
 for (i=0 ; i<=q->r
  if (q->a[i] == cu
   return 1;
 return 0;
}
int empty(struct q
{
 return (q->front
}
void bfs(int curre
{
 int i;
 insert(current,&
 while (!empty(&
 { current = delq
  printf("%d " ,
  visited[current
  for (i=1;i<=v;i
  {
   if (g[current]

   {
    insert(i,&q)
   }
  }
 }
} /*end bfs */
void main()
{  q.front = q.r
 creategraph();
 bfs(1); }
```

```
int delqueue(struct queue *q)
{ int x;
  x = q->a[q->front];
  if (q->front == q->rear)
    q->front = q->rear = -1;
  else
    q->front++;
  return x;
}
int found(int current , struct queue *q)
{ int i;
  for (i=0 ; i<=q->rear ; i++)
    if (q->a[i] == current)
      return 1;
  return 0;
}
int empty(struct queue *q)
{
  return (q->front == -1);
}
void bfs(int current)
{
  int i;
  insert(current,&q);
  while (!empty(&q))
  { current = delqueue(&q);
    printf("%d " , current);
    visited[current]=1;
    for (i=1;i<=v;i++)
    {
      if (g[current][i] == 1  && visited[i] != 1 && !found(i,&q))

      {
       insert(i,&q);
      }
    }
  }
} /*end bfs */
void main()
{   q.front = q.rear = -1;
  creategraph();
  bfs(1); }
```

A Greedy Algorithm: Shortest Paths:

Consider a directed graph G in which every edges has a non negative weight attached, and our problem is to find a path from one vertex 'v' to another 'w' such that the sum of weights on the path is as small as possible such a path called 'shortest path'. The weights may represent costs, time or some quantity other than distance.

Consider example of airline routes, with each vertex representing a city and the weight on each edge the cost of flying from one city to the second. The problem is to find a routine from city 'v' to city 'w' such that total cost is a minimum.

Consider the directed graph as shown below.

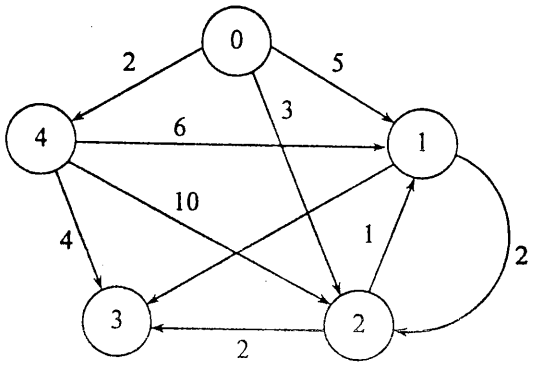
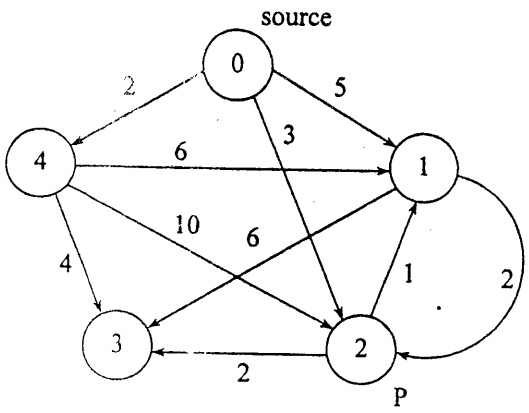


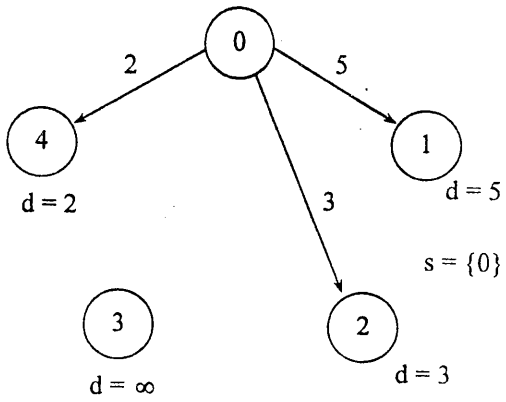
Fig. A directed graph with weights.

The shortest path from vertex 0 to vertex 1 goes via vertex 2 and has a total cost of 4, compared to the cost of 5 for the edge directly from 0 to 1 and the cost of 8 for the path via vertex 4.

From starting node (vertex) called the source and finding the shortest path to every other vertex for simplicity, we take the source to be vertex 0, and our problem then consists of finding the smallest path from vertex 0 to every other vertex in the graph. The basic requirement is that the weights are all non–negative.



(a)



(b)

If the inclusion of next input into the partially constructed optimal solution will result in an in feasible solution then this input is not added to the partial solution.

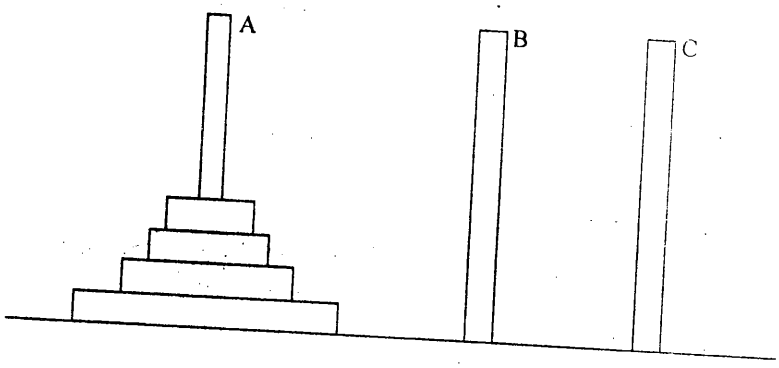
(Most problems have 'n' input and require us to obtain subset that satisfies some constraints is called a feasible solution. We are required to find a feasible solution that optimizes (minimizes or maximizes) a given objective function. A feasible solution that does this is called an optimal solution)

Divide and conquer algorithm :

Divide and conquer is the most important and widely applicable technique for designing efficient algorithms. It consists of breaking a problem of size n into smaller problems in such a way that from solutions to smaller problems we can easily construct a solution to entire problem.

Considering the problem of "tower of Hanoi" Puzzle. It consists of three pegs A, B and C. Initially peg A has some number of disks on it, starting with the largest one at the bottom and successively smaller ones on top.

The initial positions in tower of Hanoi puzzle are as follows:



The object of the puzzle is to move the disks one at a time peg to peg, never placing a larger disks on top of a smaller one, eventually ending with all disks on peg B.

The puzzle can be solved by the following simple algorithm:

Imagine the pegs arranged in a triangle on odd–numbered moves, move the small disk one peg clockwise on even–numbered moves, make the only legal move not involving the smallest disk.

The above algorithm is concise and correct, but it is hard to understand, why it works, and hard to invent on the spur of the moment. So we go for divide and conquer approach is as follows:

Consider the problem of moving the 'n' smallest disks from A to B can be thought of as consisting of two sub problems of size n–1.

First move the 'n–1' smallest disks from peg A to peg C, exposing the n^{th} smallest disk on peg A.

Move the disk from A to B. Then move the 'n–1' smallest disks from C to B.

Moving the 'n–1' smallest disk is accomplished by a recursive application of the method. As the n disks involved in the moves are smaller than any other disks.

Actual moment of individual disks is not obvious, and hand simulation is hard because at the stacking of recursive calls. This algorithm is conceptually simple to understand and more efficient one.

The technique of divide and conquer has widespread applicability. It is used in algorithm design as well as in designing circuits.

If the inclusion of next input into the partially constructed optimal solution will result in an in feasible solution then this input is not added to the partial solution.

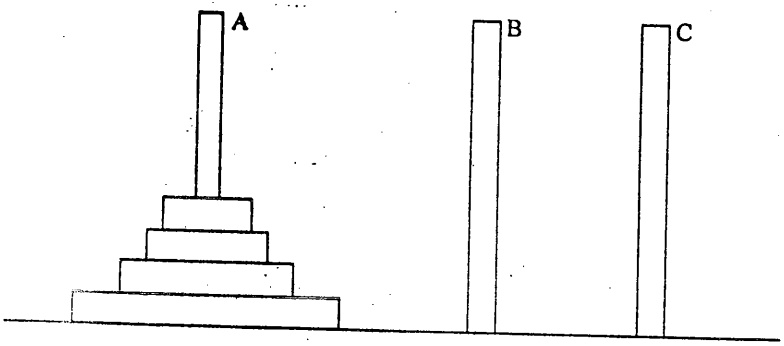
(Most problems have 'n' input and require us to obtain subset that satisfies some constraints is called a feasible solution. We are required to find a feasible solution that optimizes (minimizes or maximizes) a given objective function. A feasible solution that does this is called an optimal solution)

Divide and conquer algorithm :

Divide and conquer is the most important and widely applicable technique for designing efficient algorithms. It consists of breaking a problem of size n into smaller problems in such a way that from solutions to smaller problems we can easily construct a solution to entire problem.

Considering the problem of "tower of Hanoi" Puzzle. It consists of three pegs A, B and C. Initially peg A has some number of disks on it, starting with the largest one at the bottom and successively smaller ones on top.

The initial positions in tower of Hanoi puzzle are as follows:



The object of the puzzle is to move the disks one at a time peg to peg, never placing a larger disks on top of a smaller one, eventually ending with all disks on peg B.

The puzzle can be solved by the following simple algorithm:

Imagine the pegs arranged in a triangle on odd–numbered moves, move the small disk one peg clockwise on even–numbered moves, make the only legal move not involving the smallest disk.

The above algorithm is concise and correct, but it is hard to understand, why it works, and hard to invent on the spur of the moment. So we go for divide and conquer approach is as follows:

Consider the problem of moving the 'n' smallest disks from A to B can be thought of as consisting of two sub problems of size n–1.

First move the 'n–1' smallest disks from peg A to peg C, exposing the n^{th} smallest disk on peg A.

Move the disk from A to B. Then move the 'n–1' smallest disks from C to B.

Moving the 'n–1' smallest disk is accomplished by a recursive application of the method. As the n disks involved in the moves are smaller than any other disks.

Actual moment of individual disks is not obvious, and hand simulation is hard because at the stacking of recursive calls. This algorithm is conceptually simple to understand and more efficient one.

The technique of divide and conquer has widespread applicability. It is used in algorithm design as well as in designing circuits.

Constructing Tennis Tournaments:

Considering the design of a round robin tennis tournament schedule for $n = 2^k$ players.

Each player must play every other player and each player must play one match per day for n–1 days are the conditions for tournament and we need to find the minimum number of days needed to complete the tournament.

The tournament schedule is thus an 'n' row by 'n–1' column table, whose entry in row I and column j is the player I must contend with on the j^{th} day.

The divide–and–conquer approach constructs a schedule for one–half of the players. This schedule is designed by recurssive application of the algorithm by finding a schedule for one–half of these players and so on.

Suppose there are eight players. The schedule for player 1 through 4 fills the upper left corner (4 rows × 3 columns) of the scheduled being constructed. The lower left corner (4 rows × 3 columns) of the schedule must pit the high numbered players against one another. This subschedule is obtained by adding 4 to each entry in the upper left.

Now the problem gets simplified, only thing remain is to have lower numbered players play high–numbered player. This is easily accomplished by having player 1 through 4 play 5 through 8 respectively on day 4 cyclically permuting 5 through 8 on subsequent days.

The figure of a round–robin tournament for eight players is as shown below:

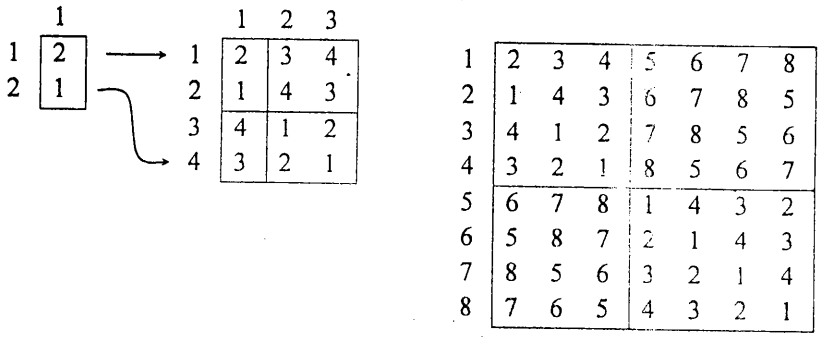


Fig. A round–robin tournament for eight players

The reader should now be able to generalize the ideas of the above shown figure to provide a schedule for 2^k players for any K.

** (Divide and conquer algorithm can also be used in QUICK SORT and MERGE SORT)

The problem of Multiplying Long Integers : -

Considering the problem of multiplying two n–bit integers X and Y. Algorithm for multiplication of n–bit (n–digit) integer usually involves computing 'n' practical products of size 'n and thus is an $O(n^2)$ algorithm. Divide–and–conquer approach to integer multiplication would break each of X and Y into two integers of n/2 bits as shown below :

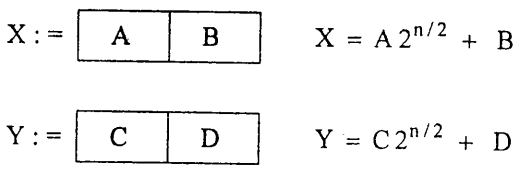


Fig. Breaking n–bit integers into n/2 – bit pieces.

The product of X and Y can be written as

$$XY = AC\,2^n + (AD + BC)\,2^{n/2} + BD \quad \rightarrow \quad (1)$$

If XY is evaluated in straightforward way, then we have to perform 4 multiplication of (n/2)–bit integers (AC, AD, BC and BD), three addition of integers with at most 2n bits and two shifts (multiplication by 2^n and $2^{n/2}$).

These additions and shifts take O(n) steps. The recurrence for T(n) can be written as follows, the total number of bit operations needed to multiply n–bit integers according to equation (1) is

$$T(1) = 1$$
$$T(n) = 4T(n/2) + cn \quad \rightarrow \quad (2)$$

From equation (2) we get an asymptotic improvement if we decrease the number of subproblems.

Consider, the following formula for multiplying X by Y.

$$XY = AC\,2^n + [(A-B)(D-C) + AC + BD]\,2^{n/2} + BD \quad \rightarrow \quad (3)$$

Formula (3) is more complicated than formula (1). It requires only three multiplications of (n/2)–bit integers, AC, BD and (A–B) (D–C), six additions or subtractions, and two shifts.

The multiplications take O(n) steps, the time T(n) to multiply n–bit integers by (3) is given by

$$T(1) = 1$$
$$T(n) = 3T(n/2) + cn$$

Solution is $T(n) = O(n^{\log_2 3}) = O(n^{1.59})$

The complete algorithm, including the details implied by the fact that equation (3) required multiplication of negative, as well as positive, (n/2)–bit integers is as given below:

Divide and conquer integer multiplication algorithm :

```
Function mult(x, y, n : integer) : integer;
   {  x and y are signed integers ≤ 2^n.
      n is a power of 2. The function returns xy  }
Var
      s : integer ;  { holds the sign  xy  }
      m1, m2, m3 : integer ;     { hold the three products  }
      A, B, C, D  :  integer ;   { hold left and right halves of x & y }
begin
      s : = sign(x) * sign(y);
      x : = abs(x) ;
      y : = abs(y) ,             { make x and y positive }
      if (n = 1) then
         if (x = 1) and (y = 1) then
            return (s)
         else
            return (0)
         else
            begin
               A  : =  left(n/2) bits of x ;
               B  : =  right n/2 bits of x ;
               C  : =  left n/2 bits of y ;
               D  : =  right n/2 bits of y ;
               m1 : =  mult (A, C, n/2) ;
               m2 : =  mult (A–B, D–C, n/2) ;
               m3 : = mult (B, D, n/2)
               return (s*(m1 * 2^n + (m1 + m2 + m3 ) * 2^(n/2) + m3 ))
            end
end {mult}
```

Above divide–and conquer algorithm is asymptotically faster and takes $O(n^{1.59})$ steps. It is generally used for large problems (actually above 500 bits).

Dynamic Programming :

There is no way to divide a problem into a smaller number of subprograms whose solution can be combined to solve the original problem. There are only a polynomial number of suboproblems, and thus we must be solving some subproblem many times.

From implementation point of view it is simpler to create a table of the solutions to all the subproblems. We fill in the table without regard to whether or not a particular subproblem is actually in the overall solution. The filling–in of a table of subproblems to get a solution to a given problem has been termed "dynamic programming".

The form of a dynamic programming algorithm may vary, but there is the common theme of a table to fill and an order in which the entries are to be filled.

Dynamic programming technique shall be illustrated by two examples.
i) Calculating odds on a match like the World Series.
ii) Triangulation problem.

1) World Series Odds :

Consider two terms, A and B, are playing a match to see who is the first to win 'n' games for some particular n. The World Series is such a match, with n = 4 we may suppose that A and B are equally competent, so each has 50% chance of winning any particular game.

Let P(i, j) be the probability that if A needs i games to win, and B needs j games, that A will eventually win the match.

Example : i) i = 2, j = 3 and then P(2, 3) gives value as 11/16.

$$P(i, j) = 1 \quad \text{if } i = 0 \text{ and } j > 0$$
$$= 0 \quad \text{if } i > 0 \text{ and } j = 0$$
$$= (P(i-1, j) + P(i, j-1))/2 \quad \text{if } i > 0 \ \& \ j > 0 \rightarrow \quad (1)$$

It takes maximum time $O(2^{i+j})$.

Let T(n) be the maximum time taken by a call to P(i, j), where i + j = n then from above equation (1) gives

$$T(1) = c$$
$$T(n) = 2T(n-1) + d$$

Where 'c' and 'd' are some constants.

Table of odds is used for computing p(i, j) is as shown below :–

	1/2	21/32	13/16	15/16	1	4
	11/32	1/2	11/16	7/8	1	3 ↑
	3/16	5/16	1/2	3/4	1	2 j
	1/16	1/8	1/4	1/2	1	1
	0	0	0	0		0
	4	3	2	1	0	

← i

Fig. Table of odds

Odds Calculation :

The analysis of function odds is easy.

```
function  odds (i, j : integer) : real ;
  Var
        s, k : integer ;
    begin
      for s : = 1 to i + j  do
      begin
            { compute diagonal of entries whose indices sum to s }
            P[ 0, s] : = 1.0 ;
            P[s, 0 ] : = 0.0 ;
          for k : = 1 to s–1  do
            P[k, s--k] : = (p[k–1, s–k]  + P[k,  s–k–1])/2.0
    end ;
        return ( P[i, j] )
    end ; {odds }
```

The outerloop takes time $O\left(\sum_{s=1}^{n} s\right)$ or $O(n^2)$, where i + j = n. Thus dynamic programming takes $O(n^2)$ time, compared with $O(2^n / \sqrt{n})$ for the straightforward approach. Since $2^n / \sqrt{n}$ grows widely faster than n^2, so dynamic programming is prefer for recursive approach under essentially any circumstances.

2) <u>The Triangulation Problem :</u>

The problem of triangulating a polygon is also a example of dynamic programming. Here the vertices of a polygon and a distance measure between each pair of vertices are given.

The distance may be the ordinary (Eucledian) distance in the plane, or it may be an arbitrary cost function given by a table. The problem is to select a set of chords (lines between non–adjacent vertices) such that no two chords cross each other, and the entire polygon is divided into triangles. The total length (distance between endpoints) of the chords selected must be minimum. Such a set of chords called <u>'minimal triangulation'</u>.

Example :
Considering the following figure of 'a heptagon and a triangulation'

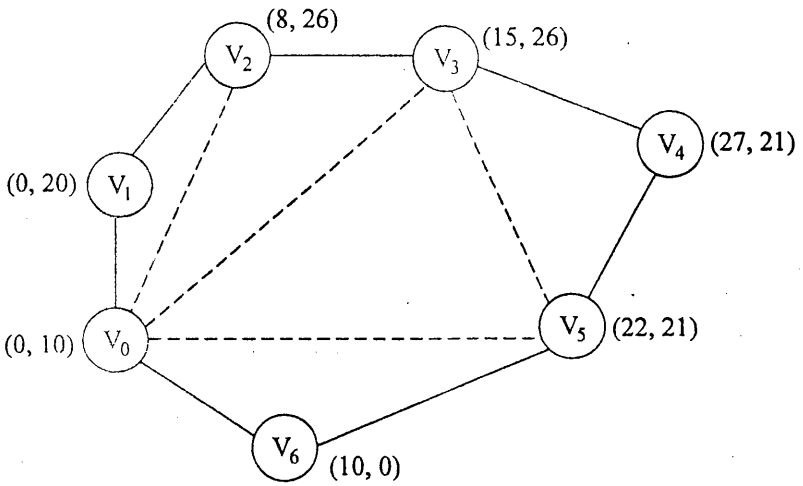


Fig. A heptagon and a triangulation.

The above figure shows a seven–sided polygon and the (x, y) co–ordinates of the vertices. The distance function is the ordinary Euclidean distance. A triangulation, which happen not to be

minimal, is shown by dashed lines. Its cost is the sum of the lengths of the chords (V_0, V_2), $(V_0, V_3), (V_0, V_5)$ & $(V_3, V_5) = \sqrt{8^2+16^2} + \sqrt{15^2+16^2} + \sqrt{22^2+2^2} + \sqrt{7^2+14^2} = 77.56$.

Backtracking :

Sometimes we are facing the task of finding an optimal solution to a problem, there is no applicable theory to help us to find the optimum, except by resorting to exhaustive search. So new systematic, exhaustive technique is used known as 'backtracking'.

In this technique a partial solution is derived at each step and validity of partial solution is checked and if incorrect we backtrack and repair the solution.

Examples: Chess, 8 queen problem, puzzle, tic–tac–toe problem.
Consider the puzzle of how to place eight queens on a chessboard, so that no queen can attack another. One of the rule for chess is that, a queen can take another piece that lie on the same row, same column, or the same diagonal as that of queen.

The 8 queen problem is a classic combinatorial problem to place eight queens on an 8 × 8 chessboard that no two queens are in direct positioning of attack i.e. no two queens out of eight are on the same row, same column or diagonal. For solving above problem, we are using a design in strategy known as backtracking.

The chessboard has eight rows and eight columns.

In this problem of 8–queens, the solution can be expressed as $(x_1, x_2, \ldots, x_8)$, where x_1 is the position of Ist queen, x_2 is the position at IInd queen and so on.

Queens are numbered from 1 to 8 and without losing generally. We are assuming that Ist Queen will be placed in 1st row, Queen II will be placed in 2nd row and so on.

So x_1 will be any value from 1 to 8, since there are eight columns in one row even x_2 will be having any value from 1 to 8 and so on.

To solve the above problem, first we are finding a place for the first queen. Let it be 1. i.e. x [1] = 1. Then we try to find the place of 2nd queen. Since queen I is directly placed in 1st column the IInd Queen cannot be placed in 1st or 2nd column, otherwise there will be a direct attack between Queen I and Queen II.

Since we are placing Queen I in first column, so we have to place Queen II in column 3. Therefore x[2] = 3.

If we are not able to find a proper location for a queen then we will backtrack and try to adjust the position of the previous queen. If it is not possible we will be backtrack once again.

There are many solutions to above problem, out of which two solutions are as shown below.

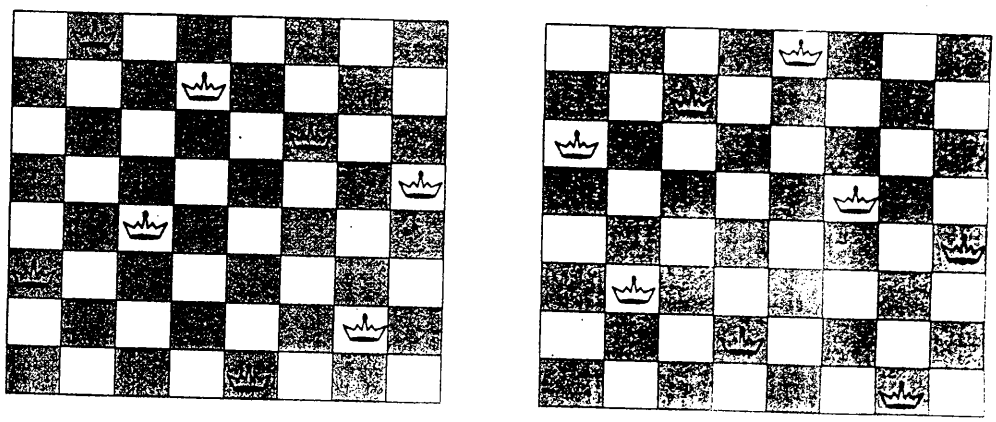


Fig. Two configurations showing eight non–attacking queens

Consider Simple example for four queens.

Consider 4 × 4 chessboard and we have to place 4 Queens on it, such that there will no direct attack between them. i.e No two queens are in same row or same column or diagonal.

We are placing each queen in different rows. Since there are 4 Queens and 4 rows, so we are placing each Queen in different rows.

In 4 Queen problem, the solution can be expressed as (x_1, x_2, x_3, x_4) where x_1 is the position of Ist Queen, x_2 will be the position of IInd Queen and so on.

x_1 will be having any value from 1 to 4 as there are 4 columns in a row.
x_2 will be having any value from 1 to 4 & so on.

We are first placing the Ist Queen in 1st column. So the IInd Queen is going to place in 3rd or 4th column, such that there should not be direct attack between Queen I and Queen II.
The position is as shown below:

	1011	1012	1013	1014
row 1	♛	?	?	?
row 2	X	X	♛	?
row 3	X	X	X	X
row 4				

(a) Dead end

'?' in above figure implies that one more position is possible for the queen to be place.

In above shown figure we are not getting the solution for placing 4 Queens on 4 × 4 boards so we will backtrack and try to adjust the position of IInd Queen.

The figure can be shown as follows:

♛	?	?	?
X	X	X	♛
X	♛	X	X
X	X	X	X

(b) Dead end

Even in above figure (b) we are not able to find a proper locations for Queen so we will backtrack. So for Queen III and Queen II, no other possibility of position. So we change the position of Ist Queen.

X	♛	X	X
X	X	X	♛
♛	X	X	X
X	X	♛	X

(c) solution

		♛	
♛			
			♛
	♛		

(d) solution

The above figure (c) gives the perfect solution of 4 Queens on 4 × 4 board and no Queen is direct attacking the other Queen.

One more solution is possible which is as shown in figure (d).

Alpha Beta – Pruning :

Considering the following game tree with values assigned at the leaves.

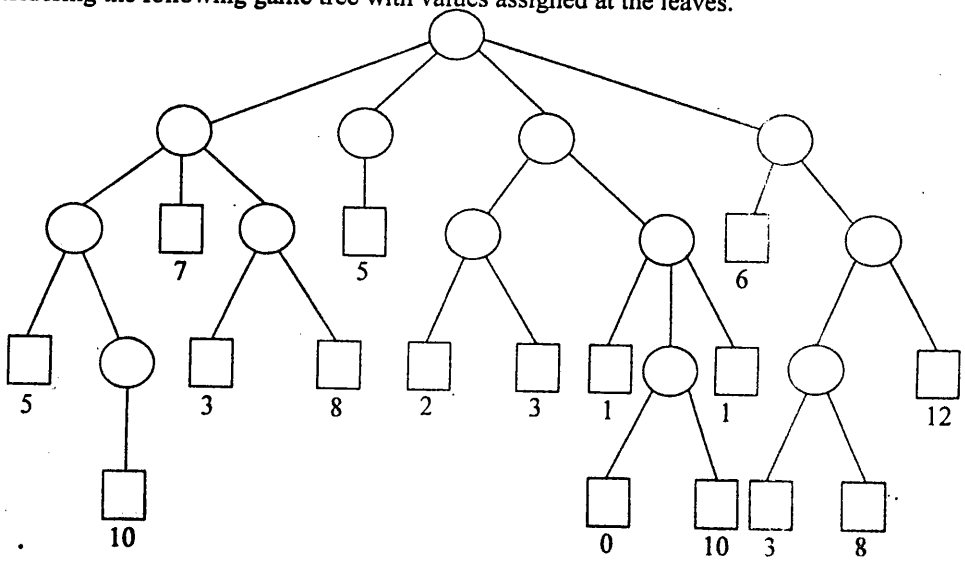


Fig. A game tree with values assigned at the leaves.

In above shown game tree, it is not necessary to obtain the values for all the vertices while doing the minimax process, for there are some parts of the tree in which the best move certainly cannot appear.

Let us suppose that we work our way through the tree starting at the lower left, and filling in the value for a parent vertex as soon as we have the values for all its children.

After we have done all the vertices in the two main branches on the left, we find values of 7 and 5, and therefore the maximum value will be atleast 7.

When we go to next vertex on level 1 and its left child, we find that the value of this child is 3. At this stage, we are taking minima, so the value to be assigned to the parent on level 1 cannot possibly be more than 3, since 3 is less than 7, the first player will take the leftmost branch, and we can exclude the other branch. The vertices that, in this way, need never be evaluated, as shown with dotted lines, and are shown in following figure.

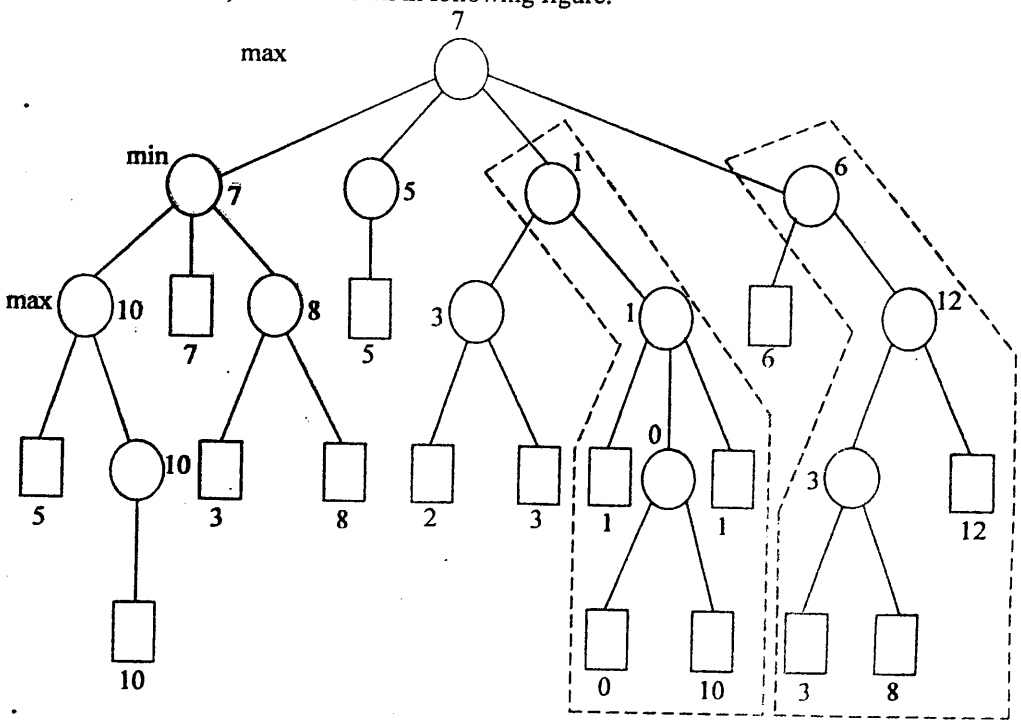


Fig. Minimax evaluation of a game tree

The process of eliminating vertices in this way is called alpha–beta pruning.

The Greek letters α(alpha) and β(beta) are generally used to denote the cut off points found.

Modify the function LookAhead so that it uses alpha–beta pruning to reduce the number of branches investigated.

Divide and Conquer Algorithm

It is a technique of solving a problem in which the algorithm divides the data set into smaller pieces. Solution is found on smaller pieces first and then the smaller solutions are combined to get final solution.

For eg. Merge sort divides the array continuosly into two parts till every partition is small enough to be sorted (1 element). The smaller sorted partitions are merged(combined) together to give a sorted array.

Draw the diagram of Merge sort process.

Greedy Algorithm

This type of algorithm has some criteria based on which it chooses the next part of solution. We say the algorithm tries to greedily satify this criteria. For e.g.

1. Kruskal's algorithm for finding minimum cost spanning tree selects edges in increasing order of weights. Thus it selects that edge first which has least weight. so algorithm is greedy for next edge with least weight.
2. Prim's algorithm for finding minimum cost spanning tree is greedy for next vertex at shortest distance.

Give an example of Kruskal's algorithm as solved in the book.

Dynamic Programming

This type of algorithm stores the result of current pass in some buffer. This result stored is used by next passes to continue finding the solution. The buffer can be any data structure like array or matrix or even a link list.

Examples of Dynamic Programming

1. Following is a program which prints 20 Fibonacii numbers.

```
void main()
{
int a=1,b=1,c,x;
printf("%d %d ",a,b);

for (x=1;x<=18;x++)
{
c=a+b;
printf("%d " , c);
a=b;
b=c;
}
}
```

In the above code a and b are the buffers used as a and b stores previous 2 Fibonacii numbers.

2. Generation of Pascal's triangle can be done using concept of dynamic programming.

```
          1
        1   1
      1 2 1
    1 3 3 1
  1 4 6 4 1
  V V V V
1 5 10 10 5 1
```

Each line can be temperorily stored in buffer like array. Then to generate next line we can add successive values of array to generate next line.