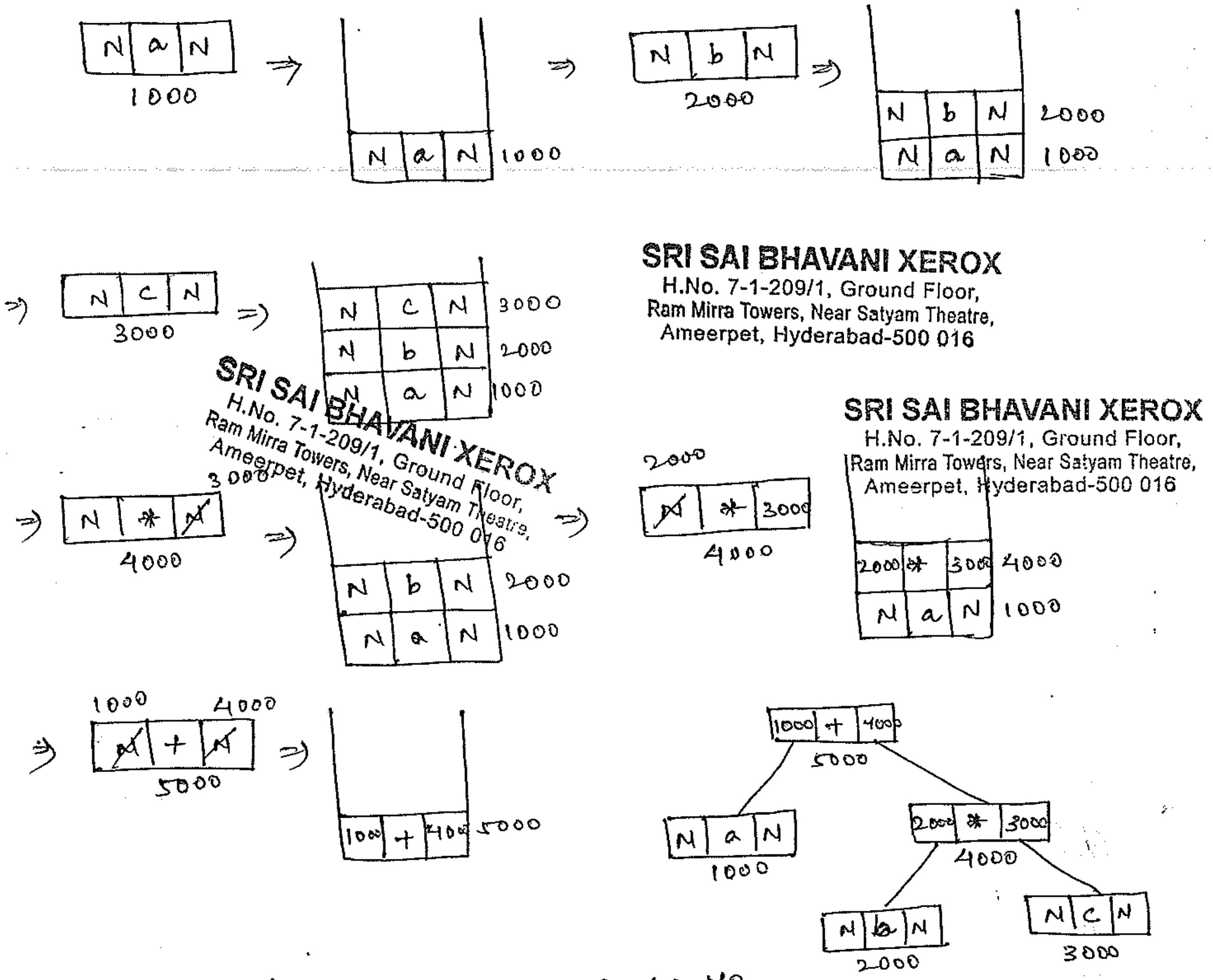
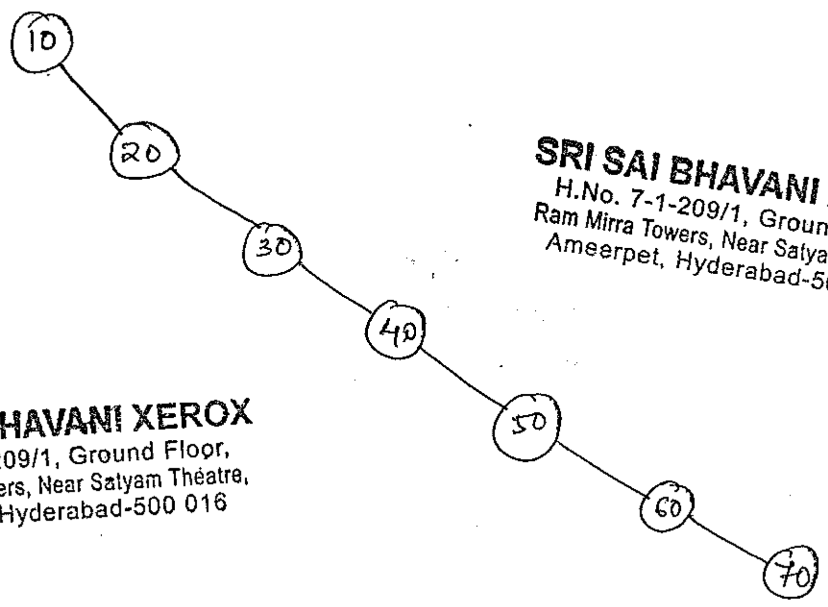


Create BST for 10, 20, 30, 40, 50, 60, 70



→ As in the above BST tree the height of the tree is increases, as the height is increasing the performance of the system will be degraded.

→ To overcome this problem we go for AVL trees.

AVL trees were invented in 1962 by two Russian scientist G.M Adelson Velsky and E.M Landis. (AVL)

Defination :-

An AVL tree is a binary search tree in binary search tree in which the balance factor of every node, which is defined as the difference b/w the heights of the node's left and right sub trees is either 0 or +1 or -1.

* Balance factor = height of left subtree − hight of right sub tree

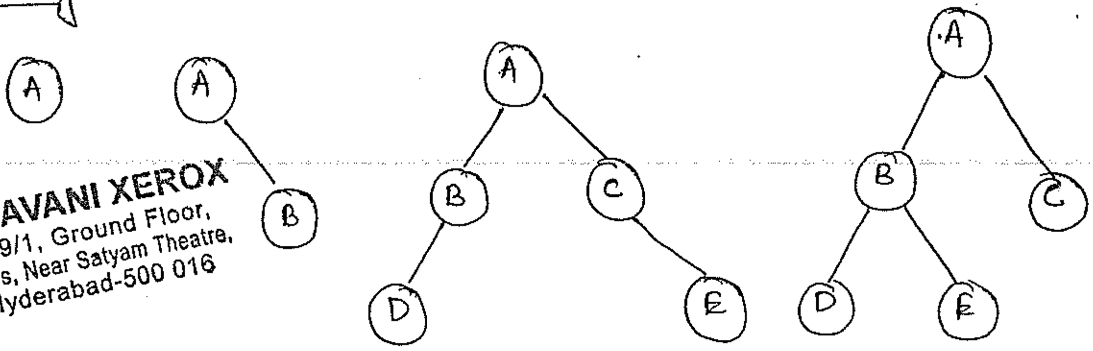
→ Each node in the AVL tree satisfy any one of the following property

(a) A node is called left heavy, if the largest path in its left sub tree is one level larger than the largest path of its right sub tree.

(b) A node is called right heavy, if the ~~ga~~ largest path in its right sub tree is one level larger than the largest path of its left sub tree.

(c) The node is called balanced, if the largest paths in both the right and left sub trees are equal.

left heavy



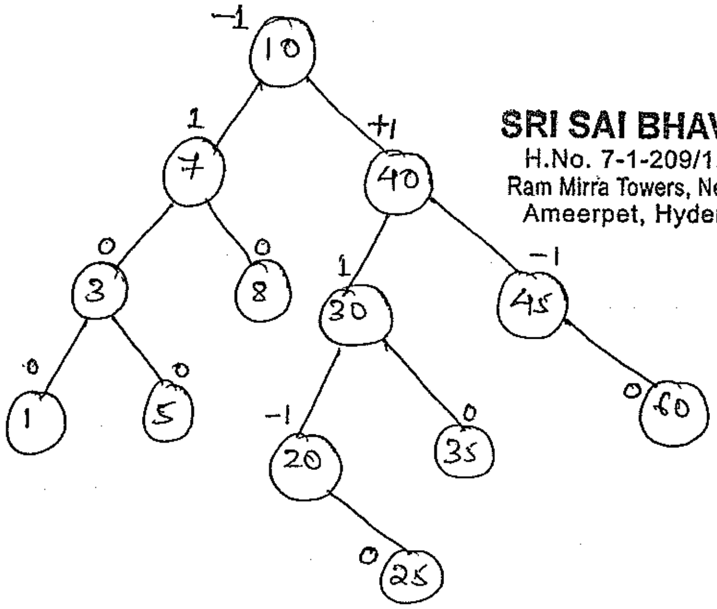
→ The construction of an AVL tree is same as that BST except after the addition of each node a checking has to be done to ensure that the AVL balancing conditions have not been violated.

→ If the new node causes an imbalance in the tree, some rearrangement of the tree nodes must be done.

→ Algorithm for Insirting the node :-

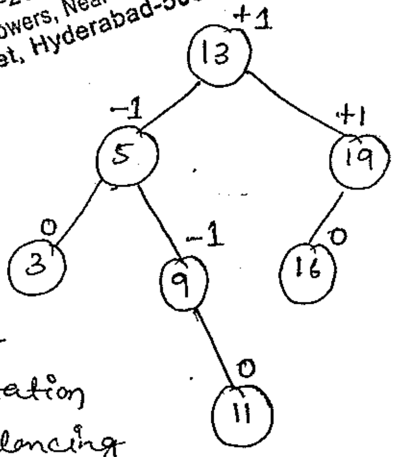
1. Insert the node in the same way as in an ordinary binary tree.
2. Trace the path from the new nodes, back towards the root for checking the height difference of the two sub trees of each node along the way.
3. Consider the node with the imbalance and the two nodes on the layer immediately below.
4. If these ~~tree~~ three nodes lie in a straight line, apply a single rotation to correct the imbalance.
5. If thaese three nodes lie in a dogleg pattern (i.e there is a bend in the path) apply a double rotation to correct the imbalance.
6. Exit.

10 , 7, 40, 3, 8, 30 , 45 , 1 , 5, 20 , 35, 25 , 60



3/01/2017

```
Struct AVL
{
 Struct AVL *LC;
 int balfact;
 int data;
 Struct AVL *RC;
};
```

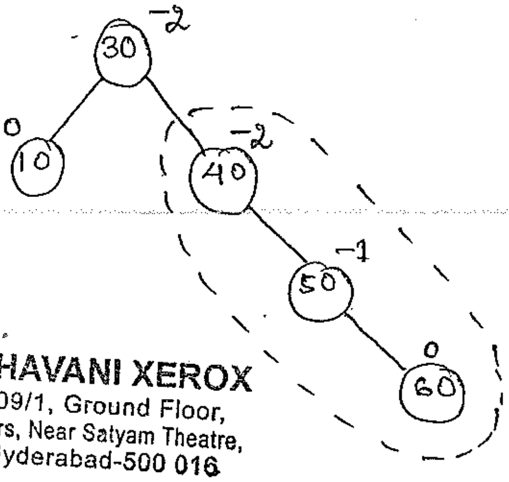


→ In the above tree, if you will do insertion and deletion the nodes may became unbalance by +2 or -2 then by using rotation method we need to perform balancing process for the nodes.

→ The rotations single rotations and double rotations.

→ Single rotations will be takes place stright line unbalance

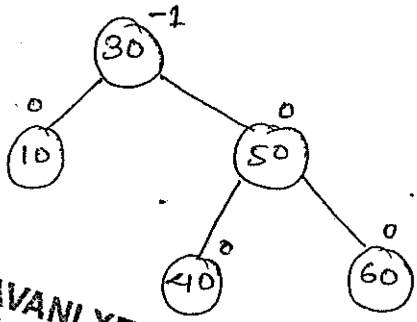
→ Double rotation curved line unbalance.



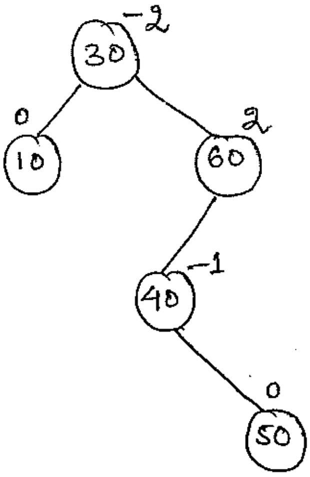
→ The above tree is unbalanced we have to employ the rotation to the nodes below to it which is unbalanced i.e 40, 50 and 60

→ The nodes (3 nodes) lies in a straight line so single rotation is applied to restore the balance.

→ at present it is straight line with -ve unbalance so apply left rotation for unbalanced node.



Double Rotation :-



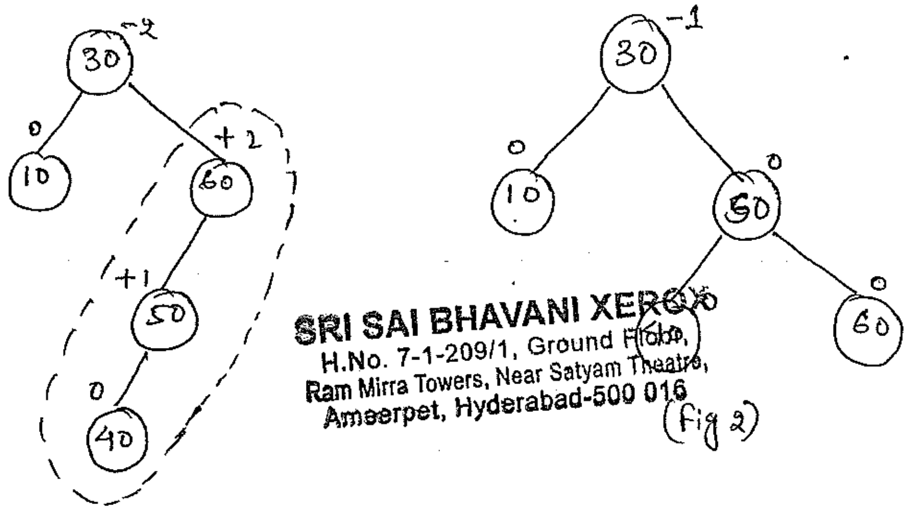
→ while tracing the path, the 1st imbalance is detected at node 60. we restrict our ~~rotatis~~ attention to this node and the two nodes immediatly below it (40 & 50)

→ This three nodes are in a curved line so we need to apply double rotations.

→ A double rotation nothing but consisting of two single rotations which are in opposite directions.

→ The 1st rotation occurs on the two layers below the node when imbalance is found (40 and 50)

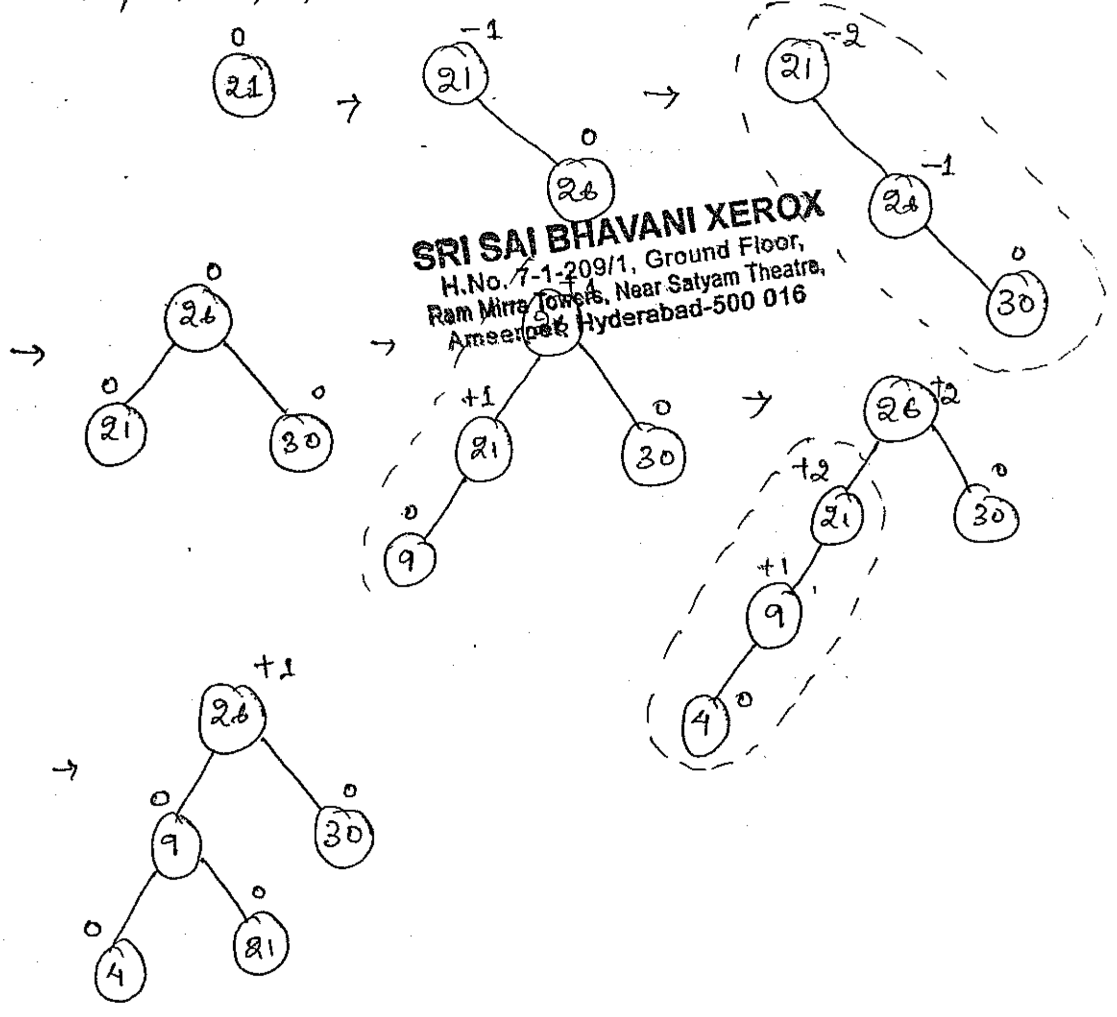
→ rotate the node below 50 by replacing with 40. and now 50 became the child of 50.

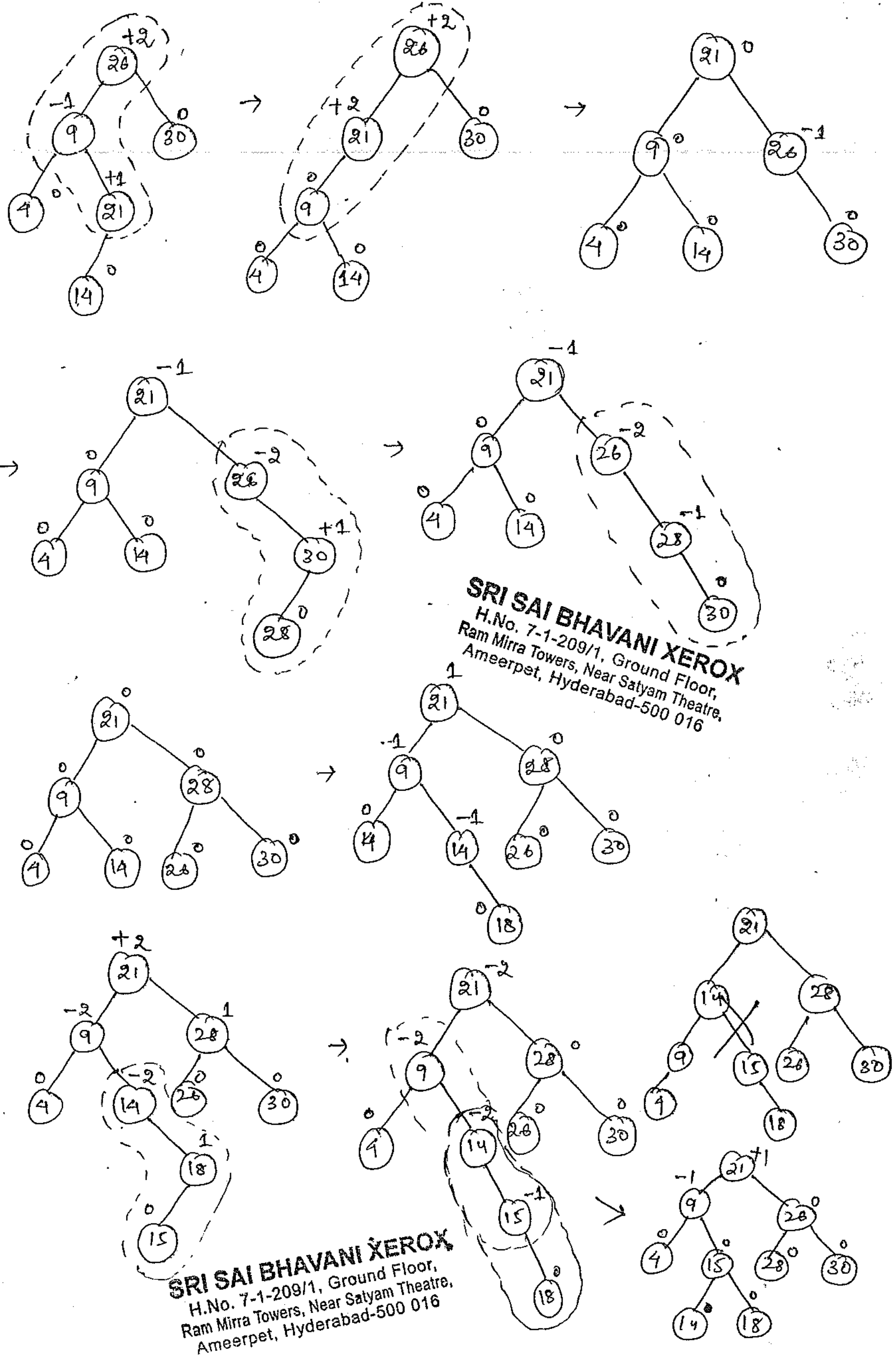


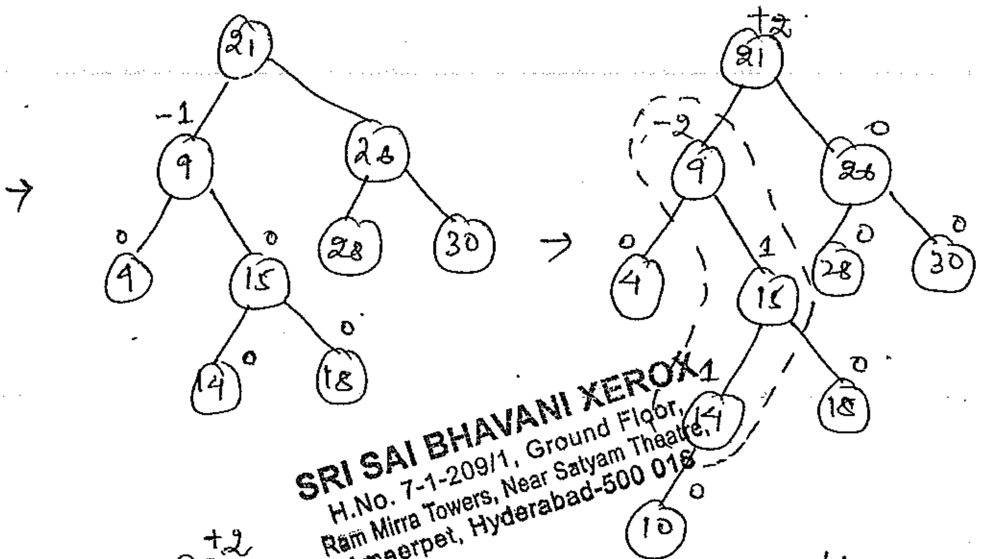
(fig 2)

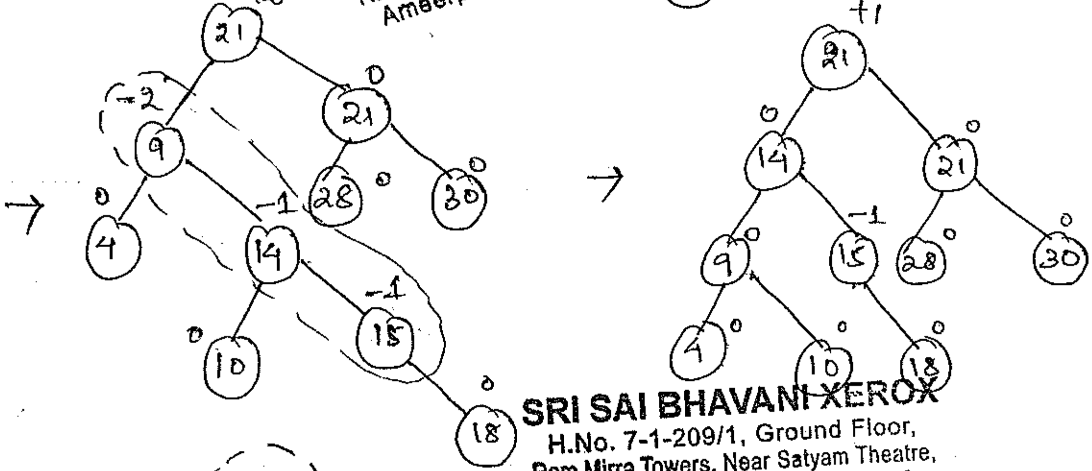
→ Apply the second rotations which invalves the nodes (60, 50, 40). since this ~~8~~ 3 nodes are lies in a stright line apply single rotation to restore the balance by replacing 60 by 50. and placing 60 as a right child of 50. (fig 2)

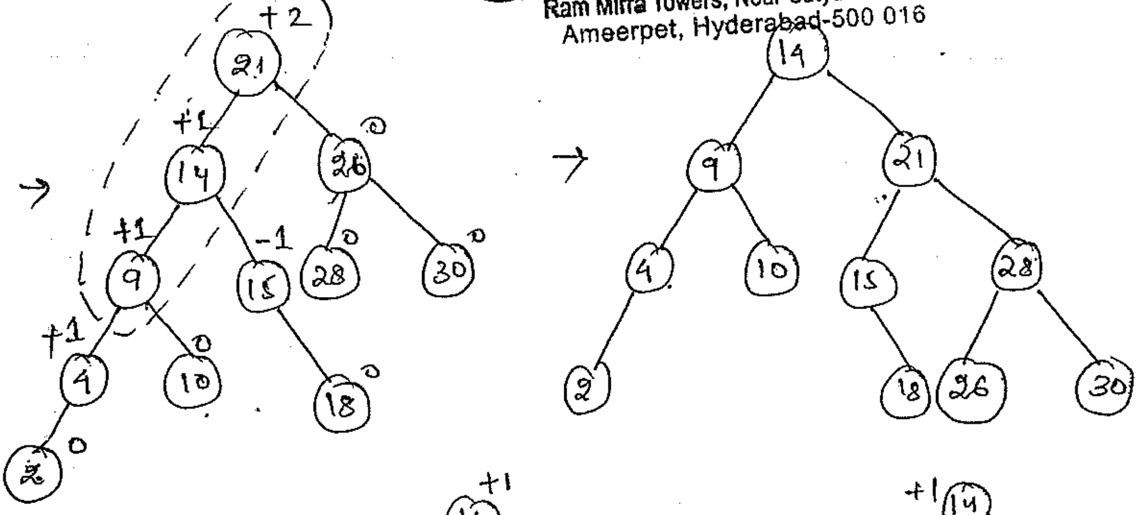
21, 26, 30, 9, 4, 14, 28, 18, 15, 10, 2, 3, 7



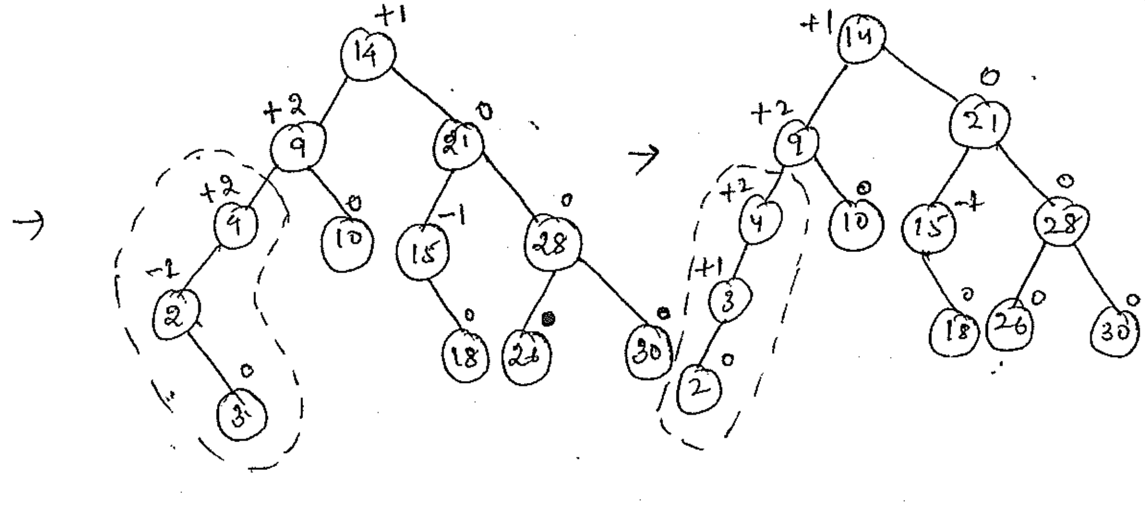
+2 26, -1 9, 0 30, 0 4, +1 21, 0 14
→
+2 26, +2 21, 0 30, 0 9, 0 4, 0 14
→
0 21, 0 9, -1 26, 0 4, 0 14, 0 30
→
-1 21, 0 9, -2 26, 0 4, 0 14, +1 30, 0 28
→
-1 21, 0 9, -2 26, 0 4, 0 14, -1 28, 0 30
→
0 21, 0 9, 0 28, 0 4, 0 14, 0 26, 0 30
→
1 21, -1 9, 0 28, 0 4, -1 14, 0 26, 0 30, 0 18
→
+2 21, -2 9, 1 28, 0 4, -2 14, 0 26, 0 30, 1 18, 0 15
→
-2 21, -2 9, 0 28, 0 4, -2 14, 0 26, 0 30, -1 15, 0 18
21, 14, 28, 9, 15, 26, 30, 4, 18
→
+1 21, -1 9, 0 28, 0 4, 0 15, 0 28, 0 30, 14, 0 18

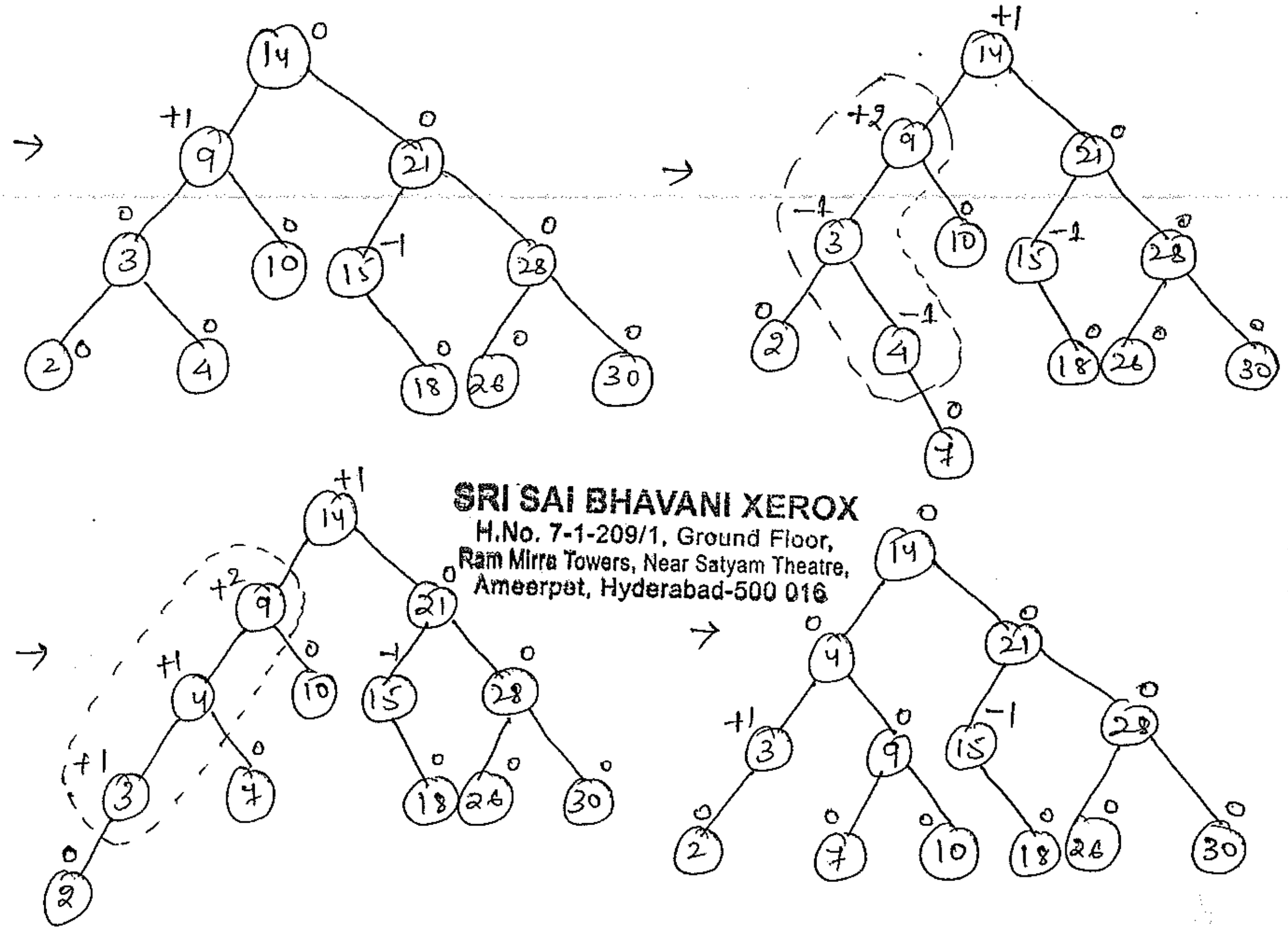

→
21
-1
9
26
0
4
0
15
28
30
14
0
0
18
→
+2
21
-2
9
0
26
0
4
1
15
0
28
0
30
1
14
0
18
0
10


→
+2
21
-2
9
0
21
0
4
-1
14
28
0
30
0
0
10
-1
15
0
18
→
+1
21
0
14
0
21
9
0
-1
15
28
0
0
30
4
0
10
0
18
0


→
+2
21
+1
14
26
0
+1
9
-1
15
28
0
30
0
+1
4
10
0
18
0
2
0
→
14
9
21
4
10
15
28
2
18
26
30

add 7

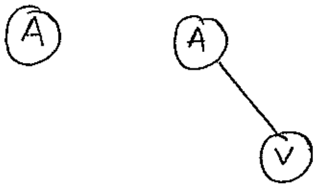

→
+1
14
+2
9
0
21
+2
4
10
0
-1
15
0
28
-1
2
18
0
26
30
0
3
0
→
+1
14
+2
9
0
21
+2
4
10
0
15
-1
0
28
+1
3
18
0
26
0
30
0
2
0

→
14 0
9 +1
21 0
3 0
10 0
15 -1
28 0
2 0
4 0
18 0
26 0
30 0
→
14 +1
9 +2
21 0
3 -1
10 0
15 -1
28 0
2 0
4 -1
18 0
26 0
30 0
7 0
→
14 +1
9 +2
21 0
4 +1
10 0
15 -1
28 0
3 +1
7 0
18 0
26 0
30 0
2 0
→
14 0
4 0
21 0
3 +1
9 0
15 -1
28 0
2 0
7 0
10 0
18 0
26 0
30 0

4/01/2017

4/01/2017

A, V, L, T, R, E, I, S, O, K



Balanced binary tree is a very useful data structure for searching the element with less time. An unbalanced binary tree takes O(n) time to search an element from the tree, in the worst case. But the balanced binary tree takes only O(log n) time complexity in the worst case.

Multi-Way Trees:-

→ This multiway trees will takes place by large no of data are existed.

→ Basically we use in all databases.

B-Trees:-

Q. where we use B-Trees:-

→ In the databases when we retriving the data from a large amount we use B.Trees.

→ suppose when a query is defined, that query doesnot falls on the database engine. for example

select * from emp where sal > 3000 ;

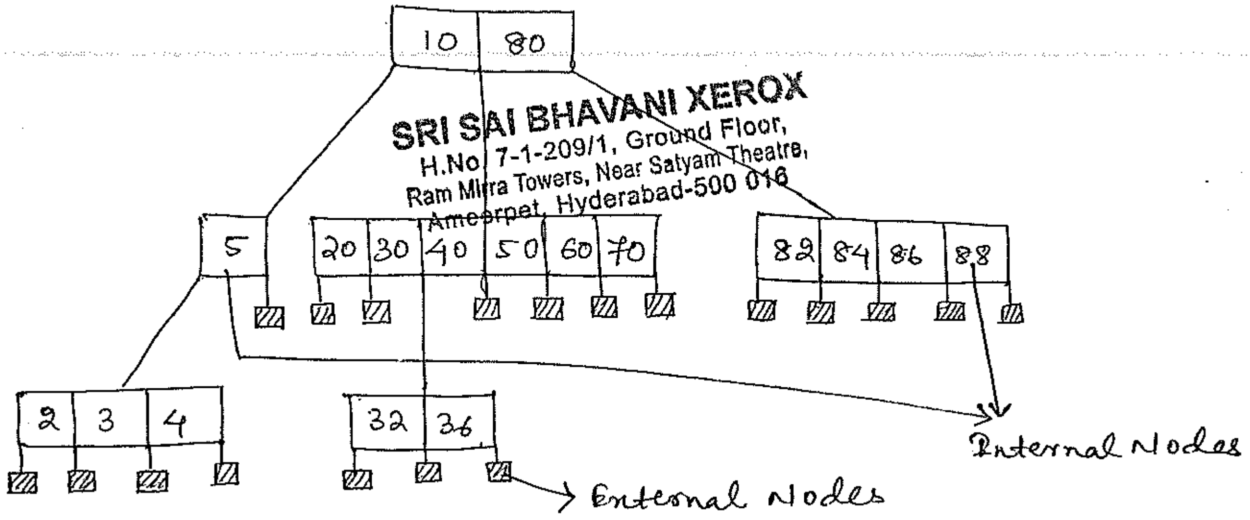
→ first internally create index on salary column then automatically B.Tree is created.

→ Index must be create, then only B-Tree is created.

→ The query doesnot affect on a database record, it affect on B-Tree.

→ Every multi-way trees has to satisfy the m-way proper--ties.

7-way tree example



→ An m-way search tree may be empty. If it is not empty it is a tree that satisfy the below property:-

① → In the corresponding extended search tree (obtained by replacing zero pointers with external nodes), each internal node has up to m children and between 1 and m-1 elements (External nodes contain no elements and have no children)

→ As per the above point we accepted 7-way tree minimum = 1, max = 6 internal nodes has to exist

② → Every node with P elements has exactly P+1 children every node will be present one extra child.

③ → Consider any node with P elements. Let $K_1, \ldots K_p$ be the keys of these elements. The elements are ordered so that $K_1 < K_2 < \ldots < K_p$. Let $C_0, C_1, \ldots C_p$ be the P+1 children of the node. The elements in the subtree with root C_0 have keys smaller than K_1, those in the subtree with root C_p have keys larger than K_p and those in the subtree with root C_i have keys larger then K_i but smaller than K_{i+1}, $1 <= i < p$.

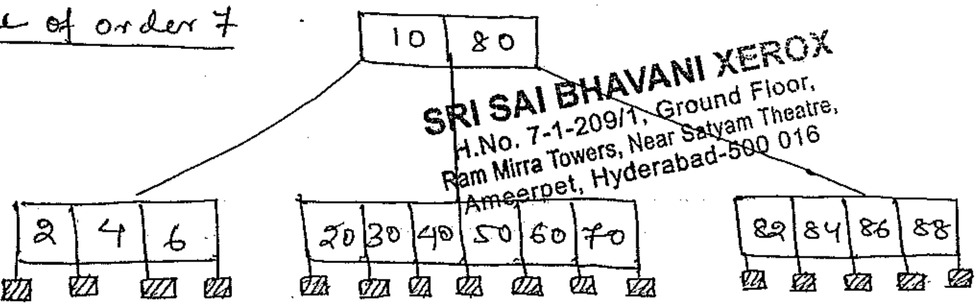
what ever the data is present all the data has to present from decreasing to increasing order i.e in ascending order.

→ All the above 3 points has to satisfy by the B-Tree.

Defination :-
A B-tree of order m is an m-way search tree. If the B-tree is not empty, the corresponding extended tree satisfies the following properties

1. The root has at least two children
2. All internal nodes other than the root have at least [m/2] children.
3. All external nodes are at same level.

B-Tree of order 7



(The B-Tree of order 7 appears in above fig. All nodes are at level 3} the root has three children and all remaining internal nodes have at least four children. Additionally, it is a 7-way Search tree)

5/01/2017

Analysis And Design

Q. what is an algorithm?

→ An algorithm is nothing but a sequence of statements for solving a problem.

example :- preparing an omlet.

→ why analysis of algorithms?

To go from city A to city B there can be many ways to carry out by flight, by bus, by train, and also by bycycle.

→ Depending on the availability and convience we choose any one of them, similarly in computer science multiple algorithms are available for solving the same problem.

Example :- Sorting problem has many algorithms like insertion, selection and many more algorithms.

→ Analysis helps us to determining which of them efficiency in terms of time and space consumed.

Goal of Analysis of algorithms :-

→ The goal of analysis of algorithms is to compare algorithm mainly interms of running time but also interms of other factors, memory, developers affect etc.

what is running time analysis?

It is a process of determining how processing time increases as the size of the problem increases. Input size is the no of elements in the input depending on the problem type, the input may be different types. The following of the common types → size of the array

→ Polynomial degree

→ No of elements in the matrix etc.

How to compare algorithms ?

To compare the algorithms some of the few objectives

1. Execution time :- Not a good measure as the execution time are specific to a particular computer

2. No of statements executed :- Not a good measure since the no of statemente changes witha the programming language as well as style of the program.

Example :-

Biggest of 4

Algorithm - 1

```
if (a>b)
  if (a>c)
    if (a>d)
Return a
else
 Return d
End if //
Else
 if (c>d)
 Return c
 Else
 Return d
  End if
  End if
  Else
  If (b>c)
    If (b>d)
    Return b
 Else
  Return d
  End if
  Else
   If (c>d)
   Return c
            Else
            Return d
            End if
            End if
            End if
```

Algorithm - 2

```
Big = a ;
if ( b> big)
Big = b ;
End if
If (c > big)
 Big = c ;
 End if
If (d > big)
 Big = d ;
 End if
 Return big
```

→ As in the above two algorithms no of comparisons are same, so that the time span for both will be the same.

→ so the ideal solution is the no of comparisions is the major factor for any program.

→ Let us assume that ~~the~~ we express the running time of an algorithm as a function of input n i.e f(n) and Compare, the different functions corresponding to running time. This type of comparision is indepedent of ~~mission~~ machine time, programming time etc.

What is the rate of growth?

→ The rate at which the running time increases as a function of input called rate of growth.

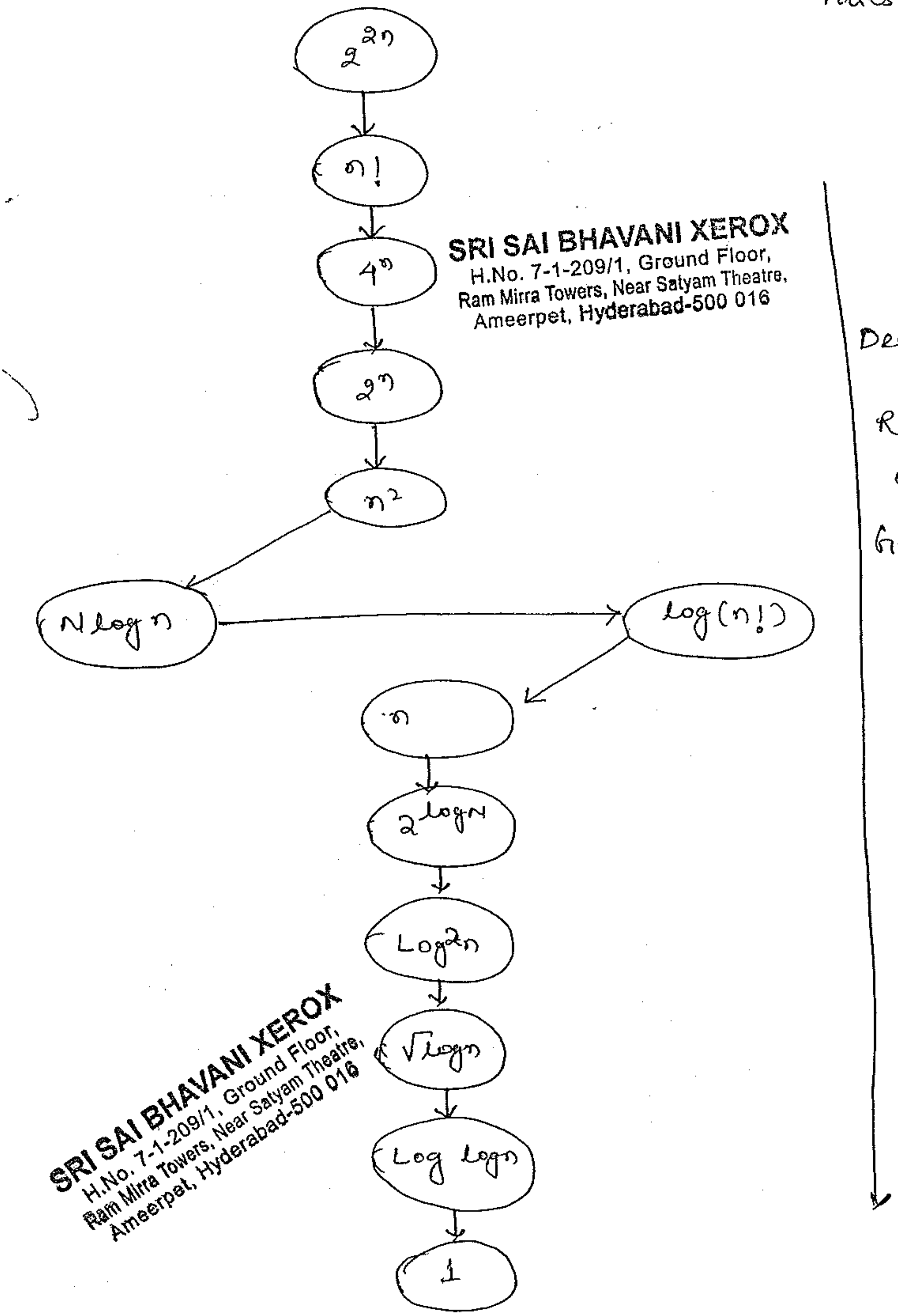
→ Let us assume that you went to a shop to buy a car & a cycle. If anybody ask you what u re buying? generally we say buying a car. This is because cost of car is too high as compared to cost of cycle.

$$\text{Total cost} = \text{cost of car} + \text{cost of cycle}$$

for the above session cost of a car of a cycle in terms of functions.

Commonly used Rate of Growth :-

Below diagram shows the relationship between the rates of growth.



6·7·2017

Time Complexity	Name	Example
1	Constant	Adding an element to the front of a linked list
Logn	Logarithmic	finding an element to the in a sorted array.
N	Linear	finding an element in an unsorted array
Nlogn	Linear Logarithmic	sorting n items by divide and -conquer (Merge sort.
N^2	Quadratic	shortest path between two nodes in a graph.
N^3	Cubic	Matrix Multiplication
2^n	Exponential	The Towers of Hanoi problems

Types of Analysis :-

→ To analyse the given algorithm we need to know on what inputs the algorithm takes less time performing and what inputs takes long time we have to analyze.

→ To analyze the algorithms we need some kind of syntax and forms the base for asympotic analysis / Notations

→ There are three types of analysis ⇒ worst case,
⇒ Best case
⇒ Average case.

1. O (big oh) → Worst case → Upper bound
2. Ω (big omega) → Best case → lower bound
3. Θ or ∂ (big theta) → Average case → Upper and lower bound.

→ Before solving a problem we have informal solution nothing but algorithm.

→ for a program we can define one or two or more algorithms, then we make analysis which one is better. in the sence of less time, and less memory. to compare such orders of growth we have above notations.

1. Big oh (O) :-

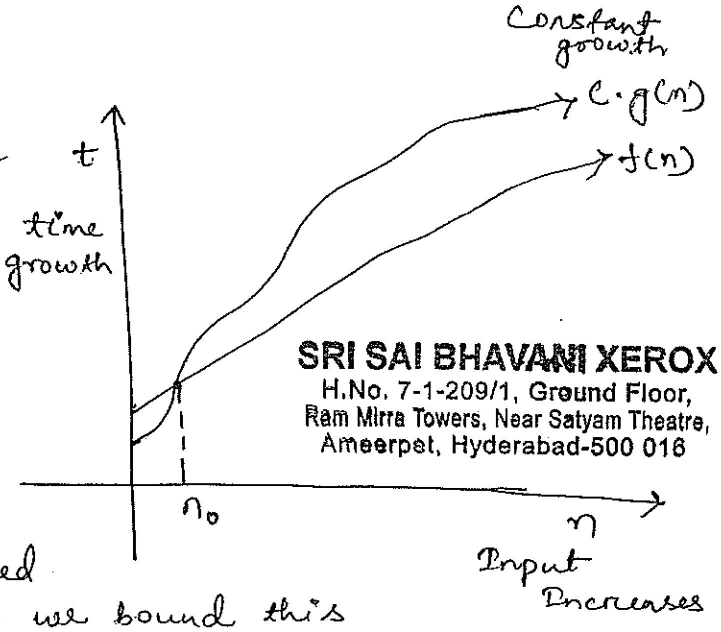
n=n₀

→ As input size is increasing time increases as f(n)

→ In keeping larger input problem time increases as f(n) and n is the size of input.

→ After some limit $n = n_0$

→ for a given function we need to findout f(n). At present we bound this with some other function c.g(n)



→ Value of c.g(n) always greater than f(n) which means

$$\boxed{f(n) \leq c\cdot g(n) \text{ for all } n \geq n_0}$$

where c and n are real numbers and $c > 0, n_0 \geq 1$

→ If this condition is satisfied with this equation then we can say that $\boxed{f(n) = O g(n)}$ g → growth.

→ which statement we can prove $f(n) < g(n)$?

Let us $f(n) = 3n+2$ $g(n) = n$

Example for Big oh (O) Notation :-

1. Given $f(n) = 3n+2$, prove that $f(n) \in O(n)$

solution :- first we have to compare c and n_0

By defination,

$$\underbrace{3n+2}_{f(n)} <= \underbrace{c \cdot n}_{g(n)} \quad \text{for } n >= n_0$$

Dividing both sides by n

$$3 + 2/n <= c \quad \text{for } n >= n_0$$

Setting $n_0 = 1$, we get $c >= 5$

now we can say that $3n+2 <= 5n$, for $n >= 1$

Now $3*1 + 2 <= 5*1$, for $n = 1$

$3*2 + 2 <= 5*2$, for $n = 2$ hence the proof

2. Show that $3n^2 + 4n - 2 \in O(n^2)$

solution :- we need to find c and no such that

$$3n^2 + 4n - 2 <= c * n^2 \quad \text{for all } n >= n_0$$

divide both side by n^2, we get

$$3 + 4/n - 2/n^2 <= c$$

if we choose $n_0 = 1$, then we need ~~to~~ a value of c such that

$$3 + 4 - 2 <= c$$

we can set c equal to 5. Now we have,

$$3n^2 + 4n - 2 <= 5n^2 \quad \text{for all } n >= 1, \text{ and } c = 5$$

now, $3*1 + 4*1 - 2 <= 5*1$ for $n = 1$

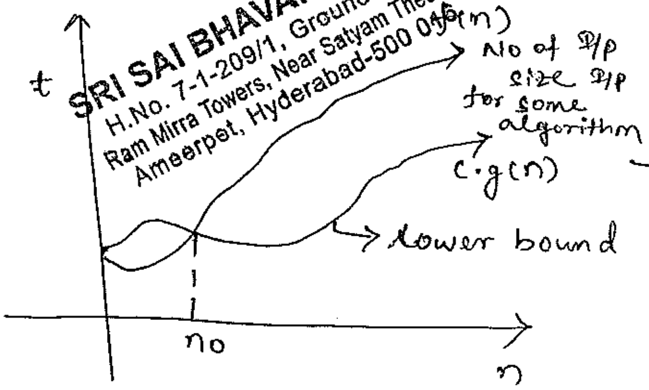
$3*4 + 4*2 - 2 <= 5*4$ for $n = 2$ Hence the proof

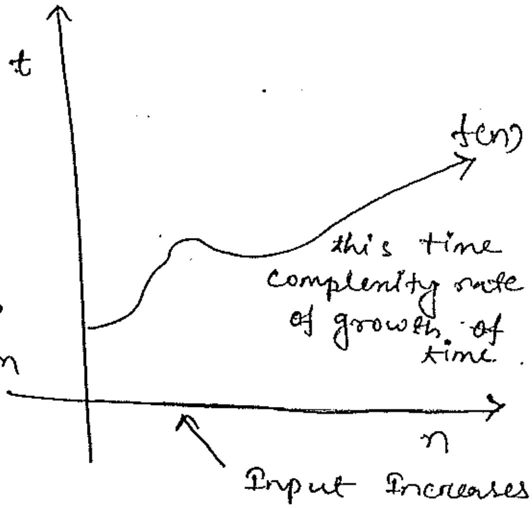
$g(n) = n^2, n^3, n^n, 2^n$

Least upper bound and higher upper bound.

2. Big Omega (Ω) Notation :-

→ This is abr initial input of f(n) if the growth has been accepted then growth will be like below fig. given.





$$\boxed{f(n) \geq c \cdot g(n) \text{ for all } n \geq n_0}$$

Where always $n_0 > 1$

Example for Big omega (Ω) Notation :-

1. Given $f(n) = 5n^2$, prove that $f(n) \in \Omega(n)$.

Solution :- As per the defination

$$5n^2 >= c \cdot n$$
$$c \cdot n <= 5n^2$$

Divide both side by n

we get $c <= 5n$

setting $n_0 = 1$, results $c <= 5$, therefore we can set c=1

$$5n^2 >= 1 * n$$

now, $5 * 1 >= 1 * 1$ for $n=1$

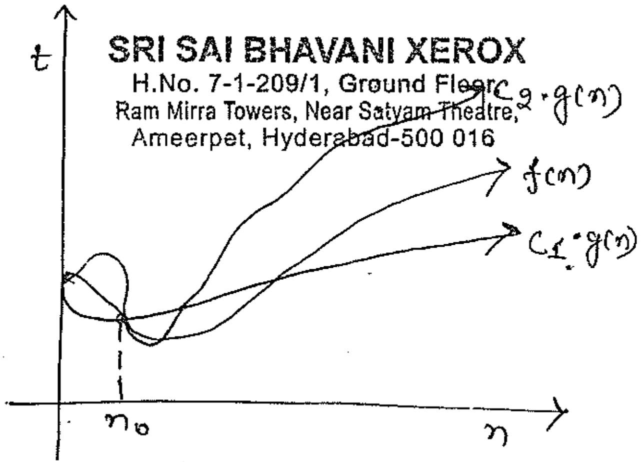
$5 * 4 >= 1 * 2$ for $n=2$ (proofed)

→ If f(n) is lower bound of n (lower bounded by n) we go for Logn, Loglog n (lower bounds)

3. Big theta (Θ) :-

→ If f is a function growing like above we should find a function both upper and lower bounds which changes C

→ If f(n) is bounded by $C_1 \cdot g(n)$ X $C_2 \cdot g(n)$ then we can say f(n) is Θ(n)



→ The constants c_1 and c_2 would be different and

$\boxed{C_2\, g(n) \le f(n) \le C_1\, g(n)}$ for all $n \ge n_0$

n	logn	nlogn	n^2	n^3	2^n
1	0.0	0.0	1.0	1.0	2.0
2	1.0	2.0	4.0	8.0	4.0
5	2.3	11.5	25.0	125.0	32.0
10	3.3	33.2	100.0	1000.0	1024.0

⇒ fastest to lowest

$O(1) \rightarrow O(\log_2 n) \rightarrow O(n) \rightarrow O(n^2) \rightarrow O(n^2) \rightarrow O(2^n) \rightarrow O(3^n)$

O(1) → const

O(n) → Linear

$O(n^2)$ → quadratic

$O(n^3)$ → cubic

$O(n^n)$ → exponential

O(logn) is faster than O(n)

O(nlogn) is faster than $O(n^2)$ but not as good as O(n)

8/01/2017

→ The time analysis i.e complexities we can observe two methods of algorithms, iterative desciplene and recursive desciplene,

→ The iterative method is normally using by loops, the iterative method it is not possible we go for recursive methodalogy.

```
①  A( )
   {
    int i;
     for(i=1 to n)
        Pf("Kiran");
   }
```

In this for loop total n times executed and printf also n times displayed. so the total time complexity is O(n).

```
②  A( )
   {
     int i, j;
       for(i=1 to n)
          for(j=1 to n)
              Pf ("Kiran");
   }
```

Outer loop n times and every time of outer loop runs inner loop runs n times. Totally $O(n^2)$

```
③  A( )
   {
    int i=1, s=1;
      while (s<=n)
        {
         i++;
         s=s+i;
         Pf("Kiran");
        }
   }
```

Now see how many times the while loop is executed

i=1, s=1, s<=n the loop is rotating upto n.

where i is incrementing in terms of 1.

where s is incrementing in terms of Value of i.

→ which means i is incrementing in leaear where s depends on i (incrementing). Let us how to findout i & s are incrementing.

initially s = 1
i = 1

s	1	3	6	10	15	21	.... >n
i	1	2	3	4	5	6	....

→ when this entire loop is going to stop whenever S value reaches a point of greater than n.

→ Let us how many times the loop is executed :-

let us K, by the time we reach K iterations, this while loop stops after K iterations.

After K iteration the value of $S > n$. Therefore what will be the value of S after K iteration.

$S = 1$, $S = 3$, for $i = 1$, $i = 2$, if you observe like this then sum of 1st n natural numbers, the value is nothing but 'S'. value 3.

When $i = 3$, S value is sum of first 3 natural numbers
when $i = 4$, S value is sum of first 4 natural numbers
which means, when reaches to K the S value will be sum of 1st n natural numbers.

i.e S value will be $\frac{K(K+1)}{2}$

Now the loop has to stop $\frac{K(K+1)}{2} > n$

$\frac{K^2+K}{2} > n$

$\cancel{\frac{K^2}{2} + \frac{K}{2} > n}$

$\Rightarrow K^2+K > 2 \times n$

$\Rightarrow K^2 + (\frac{1}{2})^2 + 2 \cdot K \cdot \frac{1}{2} > 2n + \frac{1}{4}$

$(K+\frac{1}{2})^2 > 2n + \frac{1}{4}$

$(K+\frac{1}{2}) > \sqrt{2n+\frac{1}{4}}$

Here the time Complexity is $O(\sqrt{n})$

(4)

```
A()
{ int i=1;
   for(i=1; i^2 <=n; i++)
      printf("Kiran");
}
```

→ This is also sqare root of $n \rightarrow (\sqrt{n})$, $i^2 <= n$ we can write as $i <= \sqrt{n}$, the statement executed $\sqrt{n}$ times (there is no break statement)

```
⑤  A()
   { int  i=1;
     for(i=1; i<=n; i++)
        {
          for ( j=1 ; j<=i; j++)
             {
               for(k=1; K<=100 ; k++)
                  {
                    pf("kiran");
                  }
             }
        }
   }
```

→ we have i, j and K ~~othe~~ outer for loop runs i to n inner for loop j = 1 to i .

No of times the inner for loop is executed depends on i.

→ Inner most for loop running for 1 to 100 . for loop is independent of i & j. so it is constant. so every time it will execute j loop , it is executed K loop 100 times so what is the value of i and for each for loop how many times it is executed . Initially i = 1 , then j how many times K loop is also executed .

i = 1
j = 1 since 1 to i
K loop executes 100 times in total .

i = 1
j = 1 ~~since~~ to 2 , j = 2 , times executed
K = 2 * 100 = 200 times

i = 3
j = 1 to 3 , j = 3 times executed .
K = 3 * 100 = 300 times

i = n
j = n times
K = n * 100 times

$100 + 2 * 100 + 3 * 300 + \dots + n * 100$

$100(1+2+3+\dots+n)$

$100(n(n+1))/2 = O(n^2)$

so time complexity is $O(n^2)$.

(6)
```
A()
{
  int i,j,k,n;
   for (i=1 ; i<=n ;i++)
    {
      for (j=1; j<=i^2; j++)
        {
          for (k=1; k<=n/2; k++)
              {
                 Pf("Kiran");
              }
        }
    }
}
```

→ outer loop n times

→ inner for loop j value depends on i.

if $i=1$, j loop executes 1 time

if $i=2$, j loop executes 4 times

which means j value depends on i, whenever j value depends on i, we have to unrole, it means for every value of i how many times it is executed we have to check j.

The inner most K loop executed K is not depedency on i or j it is indepedent loop.

$i=1$
$j=1$
$K=\frac{n}{2} * 1$

$i=2$
$j=4$ (for every execution of j k is executing)
$K = n/2 * 4$

$i=3$
$j=9$ (for every execution of j k is executing)
$k=\frac{n}{2} * 9$

$i=n$
$j=n^2$
$K = n/2 * n^2 = \frac{n^3}{2}$

Total time

$\frac{n}{2} * 1 + \frac{n}{2} * 4 + \frac{n}{2} * 9 + \cdots + \frac{n}{2} * n^2$

$\frac{n}{2} (1+4+9+\cdots+n^2)$

~~$\frac{n}{2} ((n(n+1)(2n+1))/6$~~ $\frac{n}{2} (n(n+1)(2n+1)) 6$

i.e sum of squares of n natural numbers, we can also write as $O(n^4)$

Time complexity is $O(n^4)$

$f(n) = n^k + n^{k-1} + n^{k-2} + \cdots O(n^k)$

⑦ A()

```
{
  for(i=1; i<n; i=i*2)
     Pf("kiran");
}
```

→ i=1 to n

→ i is incrementing not in steps of 1, it is incrementing double times, which means, incrementing not linear, which means multiple by 2, double the times. then how many times it is executing

i	1	2	4....n
	2^0	2^1	$2^2 \ldots 2^k$

→ Let us take k iterations, at 2^k, it stops then how many times it is executed means $2^k = n$, it is executing k times, which means 0 to k−1, then what is the value of k, is log n times.

$K = \log n$

Time Complexity is $O(\log n)$

→ whenever i is running 1 to n then i is incrementing interms of 2, then the time complexity is $O(\log_2 n)$

→ Log n bas 2
instead of 2 we have 3, then $i = i * 3$, then $\log_3 n$

9/01/2017

(8)
```
A()
 {
  int i, j, k;
  for(i=n/2; i<=n; i++)
   {
    for (j=1; j<=n/2; k=k*2)
      {
        printf ("kiran");
      }
   }
 }
```

```
for (j=1; j<=n/2; j++)
 {
  for(k=1; k<=n; k=k*2)
   {
    Pf("kiran");
   }
 }
}
```

$i = n/2$
$j = 1$
$k =$

$i = \frac{n}{2}$ this 'i' going from n/2 times
second n/2 times
$k = k*2$ already seen $\log_2 n$
Every loop running indepedently.
so directly take and multiply.
$n/2 * n/2 * \log_2 n$

Total Time complenity is → $O(n^2 \log_2 n)$

(9)
```
A()
{
 int i, j, K ;
 for ( i = n/2 ; i<=n ; i++)
   for(j=1 ; j<=n ; j = 2*j )
     for(K=1 ; K <=n ; K=K*2)
        Pf("Kiran");
 }
```

$i = n/2$, i going for $n/2$ times

$j=2*j$ second is $\log_2 n$

$K = K*2$ is $\log_2 n$

all loop works indepedently

$\frac{n}{2} * \log_2 n * \log_2 n$

$\frac{n}{2} (\log_2 n)^2$

$= \frac{n}{2} * (\log_2 n)^2$

$O(n(\log_2 n)^2) \rightarrow$ Time complexity.

(10)
```
A()
{
 for(i=1; i<=n ; i++)
    for(j=1;j<=n ; j=j+i)
        Pf("Kiransirds");
}
```

$i = 1 \rightarrow n$

$j \rightarrow n$ $j = j+i$

$i = 1$	$i = 2$	$i = 3$
$j = 1$ to n	$j=1$ to n	$j=1$ to n
n times	$n/2$ times	$n/3$ times

$i = K$	$i = n$
$j = 1$ to n	$j = 1$ to n
n/K	n/n

$\Rightarrow n(1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n})$

$\Rightarrow n(\log n)$

Time complexity $= O(n \log n)$

SORTING

Q. what is sorting ?

Arranging the data in alphabatical order A to Z and Z to A

Q. why sorting necessary ?

Sometimes sorting significantly reduced the complexity of the problem we can use sorting as technique to reduce search complexity.

Q. Classification of sorting algorithms ?

sorting algorithms are generally categorised based on the following parameters

1. By number of comparisions :-

In this method of sorting algorithms are classified based on the number of comparisions. for comparisions based sorting algorithms best case behaviour is O(nlogn) and worst case behaviours is O(n²) comparision based sorting algorithms evaluate the elements of the list by key comparisions Operation and need at least O(logn) comparision for most inputs.

2. Non - comparisions :-

Non comparision (linear) sorting algorithms like counting sort, bucket sort and radix sort etc. Linear sorting algorithms. impose few restrictions on the inputs to improve the complexity.

→ The following table illustrates analysis of all comparisions based algorithm and their stableness property

Sorting Tec	Best Case	Avg case	Worst case	stable
Bubble sort	θ(n)	θ(n²)	θ(n²)	Yes
Insertion sort	θ(n)	θ(n²)	θ(n²)	Yes
selection sort	θ(n²)	θ(n²)	θ(n²)	No
Shell sort	θ(n²)	θ(n²)	θ(n²)	No
Merge sort	θ(nlogn)	θ(nlogn)	θ(nlogn)	Yes
Quick sort	θ(nlogn)	θ(nlogn)	θ(n²)	NO
Heap sort	θ(n)	θ(nlogn)	θ(nlogn)	NO

10/01/2017

Non Comparison based sorting Techniques :-

The following table illustrates analysis of all the non Comparision based sorting algorithms and their stableness property.

Name	Best case	Avg case	Worst case	Stable
Counting sort	$\Theta(n+m)$	$\Theta(n+m)$	$\Theta(n+m)$	Yes
Radix sort	$\Theta(d(n+m))$	$\Theta(d(n+m))$	$\Theta(d(n+m))$	Yes
Bucket sort	$\Theta(d(n+m))$	$\Theta(d(n+m))$	$\Theta(d(n+m))$	Yes

where

n → number of elements in the array
m → Range of input elements is from $1 \dots m$
d → maximum of degits of any number in the input array

Stable Sorting :-

A sorting technique is stable if the records with equal keys retain their original relative order

(i) By number of swap :-

In this method of sorting algorithms are categorised by number of swaps (also called inversion)

(ii) By Memory usage :-

Some sorting algorithms are in place and they need $O(1)$ or $O(\log n)$ memory to create auxillary locations for sorting the data temporily.

(iii) By ~~Recursion :-~~ stability :-

Sorting algorithms is stable if for all indices i and j such that the key $A[i]$ equals key $A[j]$, if record $R[i]$ precedes the record $R[j]$ in the original file, record $R[i]$ precedes $R[j]$ in the sorted list. few sorting algorithms maintain the relative positions even after sorting.

(iv) By recursion :- sorting algorithms are either recursions [quick

(iv) By Adaptability :-

few sorting algorithms complexity changes based on pre-sortedness [quick-sort] pre-sortedness of the input affects the running ~~data~~ time. Algorithms that takes this into account are known to be adaptive.

Other classification :-

1. Internal sorting
2. External sorting.

Applications following are the some of the applications of sorting :-

a) Displaying of search results while using search engine in order of relevance.

b) To perform Binary search

c) selection algorithm

d) Data compression algorithms like Burrows-wheeler transformation etc.

e) Graph algorithms like kruskal etc.

f) Geometry algorithms like grahms scan, closest pair etc

g) suffix array construction etc.

Bubble Sort

→ Explain and then The algorithms gets its name from the way the smaller elements "bubble".

→ To the top of the list. Generally insertion sort has better performance than bubble sort some researchers suggest that we should not teach bubble sort because of simplicity and complexity.

→ The only ~~when~~ advantage over other implementations is that it can detect wheather the input list already sorted or not.

```
Void Bubblesort (int A[], int n)
 {
  for (int pass = n-1 ; pass >= 0; pass --)
    {
     for (int i=0; i<=pass -1; i++)
      {
        if (A[i] > A[i+1])
          {
           int temp = A[i];
           A[i] = A[i+1];
           A[i+1] = temp;
          }
      }
    }
 }
```

(n-1)(n-2)
$n^2 - 2n - n +$
$n^2 - 3n$
n(n-3)

→ The above logic is taking place on order of $O(n^2)$.

→ Algorithm takes $O(n^2)$ (even in best case) we can improve it by using one extra flag, No more swaps indicate the completion of sorting. If the list already sorted we can use this flag to skip the remaining passes.

```
Void Bubblesort Improved (int A[], int n)
 {
  int pass, i, temp, swapped = 1;
   for (pass = n-1; pass >= 0 && swapped; pass--)
    {
     swapped = 0;
     for (int i=0; i<=pass-1; i++)
       {
        if (A[i] >A[i+1])
         {
          int temp = A[i];
          A[i] = A[i+1];
          A[i+1] = temp;
          swapped = 1;
         }
       }
    }
 }
```

2 12 23 34 45

→ The modified version improve the best case to $O(n)$

11/1/2017

Insertion Sort :-

Insertion sort is a simple and efficient comparision sort. In this algorithm each iteration removes an element from the input data and inserts it into the correct position in the list being sorted. The choice of the element being removed from the input is random and this process repeated until the input elements have gone through

Advantage:-

→ Simple implementation

→ Efficient for small data

→ Adaptive :- if the input list is presorted [may be completely] then insertion sort takes O(n+d) where d is the number of inversions.

→ Stable :- Maintanes relative order of input data if the keys are same.

→ In-place :- It requires only Constant amount O(1) of additional memory space.

Examples :- Library books keeping, playing cards placing etc.

Analysis of Insertion sort :-

Suppose we have the elements 9, 6, 5, 0, 8, 2, 7, 1, 3 we can findout total permutations for 'n' number, it takes O(n!). Time taken is very fastly as the rate of growth if you observe the previous table.

9	6	5	0	8	2	7	1	3
j=1	2	3	4	5	6	7	8	9

→ Key = 6 (element) now slowly J is moving to n, and whenever J is moving to n and i going to compareing all the elements and comparing all the elements and Comparision going to whenever we find the current location and now in worst case how many comparisions and movements are required.

→ If j = 2, which means examing the second element then what happens if examine the 1st element, then howmany comparisions are required if j = 2, no of comparisions in worst case.

$j = 2 \rightarrow$ 1 comp + 1 Movement = 2 in WC $(2*1)$

$j = 3 \rightarrow$ 2 comp + 2 movement = 4 WC $(2*2)$

$j = 4 \rightarrow$ 3 comp + 2 mov = 6 $(2*3)$

$j = 5 \rightarrow$ 4 comp + 4 mov = 8 $(2*4)$

if we observe this pattern

$j = n - \underset{\text{comp}}{\underset{\uparrow}{(n-1)}} + \underset{\text{movements}}{\underset{\uparrow}{(n-1)}}$ i.e $2(n-1)$

so what is total time taken for worst complexity is for $j = 2,\ j = 3,\ j = n$

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

$2(1+2+\cdots n-1)$ this is nothing but first sum of 10 - 1 natural Nos this is nothing but

$(2(n-1)(n))/2 = O(n^2)$. Nothing but order n^2 in worst case.

Best Case:- 9 8 7 6 5 4 3 2 1 0

I/P is already sorted list which means 1 2 3 4 5 6 7 8 9 in which one comparisions for j = 2 one comparision. we can find actual position No moments.

J = 3 No of comp is 1 No of movement is nothing.

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

1 comparision and Zero movements

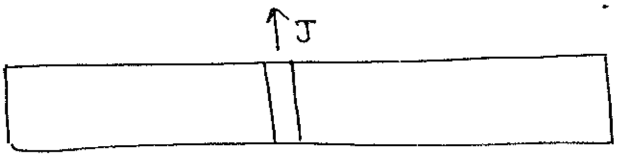
→ for all this $n-1$ elements i needs $n-1$ comparision

Best case $\underline{\Omega(n-1) = \Omega(n)}$

→ Insertion sort is "INPLACE" algorithm. Inplace means "STABLE" i.e we are not taking any extra space to sort the algorithm.

→ In an Insertion sort time complexity depends on the no of comparisions and no of movements.

→ How can we decrement comparisions and movements. If you observe in insertion sort always J sorted



→ whenever we exam and inorder to findout the position of J in this sorted list actually doing in linear order so it takes of $O(n)$ time.

→ If you apply binary method of checking the total no of time Comparisions are required to find the actual position for the Jth element, ~~may~~ might fall down $O(\log n)$.

→ But the movements will be the same.

→ Instead of linear search one element one element may be reduce the time complexity to $O(\log n)$ this improve the Time complexity ~~to~~ it doesnot the reason if we find small we need to move all elements for moving it take $O(n)$.

→ for every searching and placing it takes $O(n)$ time I have to do it nearly $n-1$ times so it is $O(n^2)$

→ Decrease no of movements since it is an array it happens but taking LINKED List which linked list is better Doubly LL movement can reduce directly can insert with constant time time. But I cannot implement Binary search on it I need to apply Linear search on it.

→ To find out location I need to compare from 1 2 5 9 so even though i used DLL.
No of comparisions are same movements can reduce. Therefore any way for searching O(n) so we can't improve the insertion sort time complexity reason movements are more.

18/01/2017.

Merge Sort:

Whenever more number of data is present better to prefer Quick, Merge etc. type of sorting.

→ Merge sort is more better rather than insertion sort, Sometimes it is better than quick sort also.

```
MERGE(A, P, q, r)
{
  n1 = q-P+1;
  n2 = r-q;
 Let L[1....n+1] and R[1 to n2+1] be new arrays
       for(i=1 to n1)
         L[i] = A[P+i+1]
 for(j=1 to n2)
   R[j] = A[q+j]
       L[n1+1] = ∞
       R[n2+1] = ∞
   i = 1, j = 1
     for(K=P to r)
```

```
if(L[i] <= R[j])
    A[k] = L[i]
     i = i+1
else
    A[k] = R[j]
     j = j+1
```

→ Merge sort is good interms of time complexity, the major point in merge sort is merging.

Merging :-

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

→ Imagin an array divide the array into two parts i.e two lists, 1 part 1 to 4 and other part 5 to 8.

→ The ~~part~~ 2 lists are already sorted

→ Again they have to sort which containing the elements

→ How does this algorithm works?

→ p to q sorted and q+1 to r is sorted and i want to merge them into a single list sorted.

p			q	q+1			r
1	2	3	4	5	6	7	8
1	5	7	8	2	4	6	9

→ Take an array like 'L' which will have exactly the number of elements from p to q, size should be size+1 (+1 for ∞)

→ Then copy all the element to the ~~two~~ list

L

1	5	7	8	∞

R

2	4	6	9	∞

→ To copy the list, it takes O(n) time complexity.

→ Every end of the array write down the ∞ (practically) infinite not possible but place an infinit it indecates a large number.

→ Then copy the two list into the [illegible] array by taking some pointer variables like [illegible].

→ compare left side arra[illegible] element and right side array of 1st element w[illegible] is pointing by i and j copy into 3rd array which is pointing by K and every time of loop rotations i, j, k incremented.

→ so the merge procedure take the entire list like

L

1	5	7	8	∞

i

R

2	4	6	9	∞

j

1	2	4	5	6	7	8	9

K

→ let us code explanation MERGE (A, P, q, r)
A is an array, remaining pointers.

→ Now count no of elements in the 1st list i.e

$n_1 = q - P + 1$

$n_2 = r - q$

→ I created two arrays left array (L) and right array (R) and +1 for ∞.

→ The 1st for loop copy the 1st list into the array, the 2nd for loop copy the second list in the array.

→ so the total time complexity, 1st copy the elements into two arrays into 1 array, it takes O(n).

→ so how many coping and comparision require for this , ~~f~~for the total no of elements in an array and 'n' terms.

→ To write one element into an array, i have to perform 1 comparision, therefore, if there are n elements then we have to perform n comparisions and also every element after compare it has to written down

→ Therefore there has to be n comparisions, n coping has to takesplace. so $(n+n) = O(n)$.

→ In this time complexity and space complexity are $O(n)$

→ merge sort is out of place sorting.

→ Because all the elements are separately coping.

what is the need of ~~two~~ to add ∞ ?

p			q	q+1			r
10	20	30	40	1	5	6	9

10	20	30	40	∞

1	5	6	9	∞

→ one list is copied in this and other list has to copy we need to compare with ∞.

→ 10 is compared with ∞, ∞ insured if one list is over, other list has to compare, so due to ∞, one list is copied other list automatically copies, so no need to check others.

→ code has to change if ∞ not available. so any where merging two list if extra space is given then only time complexity is $O(n)$. If not given, then time complexity might be increases.

so some other sorting method has to applying so merging need i/p is given 'n' numbers and we need 'n' extra space. in-order to copy all the elements from given list to next list we have O(n) times.

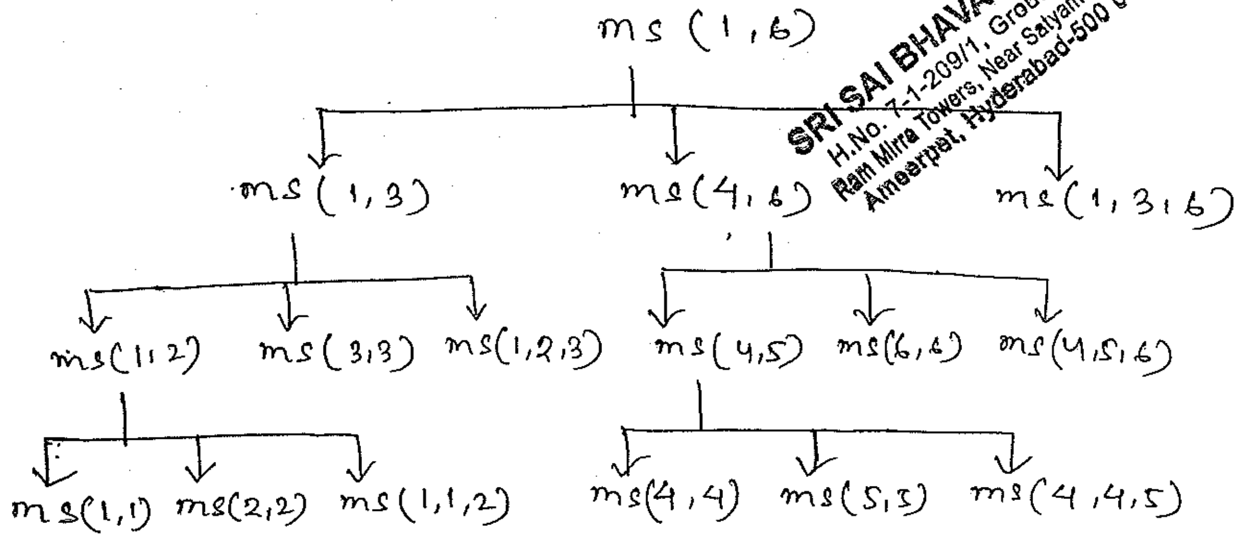
```
merge_sort (A, P, r)
  {  if P < r
        q = ⌊(P+r)/2⌋
     merge_sort (A, P, q);
     merge - sort (A, q+1, r);
     merge (A, P, q, r);
  }
```

→ If false on the category, devide and conquer.

→ In order to sort a size of n, divide the array into two halfs.

→ merge_sort is different than merging.

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



→ merge_sort can be stopped when p and q equals

19/01/2016

Program of sorting using merge sort without recursion

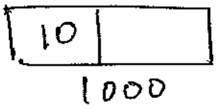
```
#include <stdio.h>
#define MAX 30
main()
{
  int arr[MAX] ,temp[MAX] , i, j, K ,n, size ,l1, h1, l2, h2;
 printf ("Enter the number of elements :");
 scanf ("%d", &n);
 for ( i=0; i<n ; i++)
   {
    printf ("Enter element %d :" , i+1);
    scanf ("%d", &arr[i]);
   }
 printf ("Unsorted list is :");
 for (i=0; i<n; i++)
    printf ("%d", arr[i]);
/* l1 lower bound of first pair and so on */
  for (size = 1 ; size <n; size = size * 2)
   {
    l1 = 0;
    K =0;   /* index for temp array */
    while ( l1 + size < n)
     {
      h1 = l1 + size -1 ;
      l2 = h1 + 1 ;
      h2 = l2 + size -1;
     if (h2 >=n)   /* h2 exceeds the limit of arr */
        h2 = n-1 ;
/* Merge the two pairs with lower limit l1 and l2 */
      i = l1;
      j = l2;
while (i<=h1 && j<=h2)
 {  if (arr[i] <= arr[j])
          temp [K++] = arr[i++];
      else
          temp [K++] = arr[j++];
  }
 while (i<=h1)
    temp [K++] = arr[i++];
  while (j<=h2)
    temp[K++] = arr[j++];
 /* Merging completed */
   l1 = h2+1 ; /* Take the next two pairs for merging*/

 } /* End of while*/
for (i=l1; K<n; i++)  /* any pair left */
 temp [K++] = arr[i];
for ( i=0; i<n; i++)
   arr[i] = temp[i];
printf ("\n [illegible] \n Elements are :", size);
 for ([illegible]; i++)
   printf(" %d ", arr[i]);
 } /* End of for loop*/
printf (" sorted list is : \n");
 for (i=0; i<n; i++)
   printf ("%d", arr[i]);
   printf ("\n");
 } /* End of main() */
```

program for sorting using merge sort with recursion

```
#include<stdio.h>
#define MAX 20
int array[MAX];
main()
 { int i, n;
   printf ("Enter the number of elements :");
   scanf ("%d ", &n);
   for (i=0; i<n; i++)
    { printf ("Enter the element %d :" i+1);
      scanf ("%d ", &array[i]);
    }
   printf ("Unsorted list is :\n");
   for(i=0; i<n; i++)
      printf (" %d ", array [i]);
   merge_sort (0, n-1);
  printf ("\n sorted list is : \n")
   for ( i=0 ; i<n ; i++)
      printf ("%d", array [i]);
     printf ("\n");
  } /* End of main() */
 merge_sort (int low, int high)
  { int mid ;
     if (low! = high)
      { mid =(low+high)/2 ;
        merge_ sort (low, mid );
        merge - sort (mid +1 , high);
        merge (low, mid , high);
      }
   } /* end of merge_sort */
```

```
merge (int low , int mid , int high)
  {
    int temp[MAX] ;
    int i=low;
     int j= mid +1 ;
     int K = low;
   while ( (i<=mid && (j <=high))
     {
        if (array[i] <= array [j])
             temp [k++] = array [i++] ;
        else
             temp [ k++] = array [j++] ;
     } /* End of while */
   while (i<=mid)
        temp [k++] = array [i++];
   while (j<=high)
        temp[k++] = array [j++] ;
     for (i=low , i<=high ; i++)
          array [i] = temp[i];
     } /* end of merge() */
```



last = ~~NULL~~ 1000

tmp = 1000

```
Void preorder(tree * r)
  { int top=0;
  tree * S[20] , *ptr= r;
    S[0] = NULL;
  while (ptr != NULL)
  { pf ("%d \t", ptr-> info);
    if (ptr-> right != NULL)
      S[++top] = ptr->right;
    if( ptr-> left != NULL)
      ptr = ptr-> left;
    else
      ptr = S[top--];
  } }
```

```
Void inorder (tree *r)
{ int top =0;
  node * S[20] , *ptr=r;
   S[0] = NULL;
  while(ptr != NULL)
   { S[++top] = ptr;
     ptr = ptr -> left;
   }
  ptr= S[top--];
  while( ptr != NULL)
   {
   pf( "%d\t", ptr-> info);
   if (ptr -> right != NULL)
   { ptr = ptr -> right;
   while (ptr != NULL)
       {
        S [~~++~~top] = ptr;
        ptr = ptr->left;
       }
   } ptr = S[top--];
  }
}
```

```
Void postorder( tree * r)
 { tree * S[100];
   int top = -1;
  tree * temp = r;
  while (temp != NULL)
   { top++;
   S~~tack~~[top] = temp;
   temp = temp->left;
  }
a:  temp = S[top];
   top--;
  if (temp ->
```

n = 10

i++

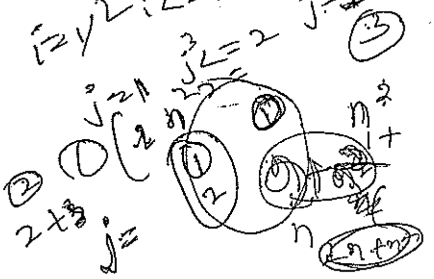
i=1, 2 i <= 2

j= ~~1~~ 2

j <= 2

10

3



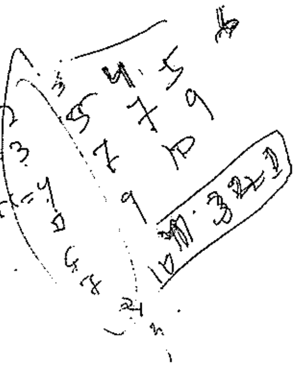
~~3+1~~

11 1+1 = 2

12 2+1 = 3

13 3+1 = 4

14



```
Void printnthfromlast ( struct node * head , int n)
 { int len = 0 , i ;
   struct node * temp = head ;
   while ( temp != NULL)
     { temp = temp -> next ;
       len ++ ;
     }
    if ( len < n )
      return ;
    temp = head ;
    for ( i = 1 ; i < len - n + 1 ; i ++ )
       temp = temp -> next ;
       P.f ( "%d" , temp -> data )
       return ;
 }
```

$n = 4$

$i = 2$

$n = 1$

$i = 1$, $j = 4$

150

$n^2 + 2ij + 1 =$

$1 + 2 + 1 = 4$

$n^2 + (i - j) = 4$

$= n^2 + 0 = 1$

100

$\frac{n}{2}$

1

$i = 2$

$j = 4$

$i = 8$

4

2 $j =$

$j <= 2^2 \; 4$

1 2 3 4

$n = 2$

$i = 1$, $j <= 1$ to i^2

$1 <= 1$

$k = 1$; $K <= \frac{2}{2}$; $k{+}{+}$

1 2

Kiran

n

$i = 1$

$n = (n+1)n$

$\frac{n}{2}(1 + 2 * \frac{n}{2}$

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

$\frac{n}{2}$

$1 + 2^2 + 3^2$ n^3

$1 + 4 + 9$

14 $\frac{n}{2}(n$

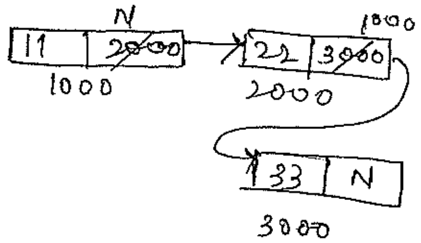
addatbeg ( )
{
~~struct temp~~,
struct 1

Void reverse ( )
{
struct

```
struct node
  {
   int info;
   struct node * next;
  } * head;
Void reverse ( )
  {
   struct node *tmp, *tmp1, *var;
    tmp = head;
    var = NULL;
    while ( tmp != NULL )
     { tmp1 = var;
       var = tmp;
       tmp = tmp -> next;
       var -> next = tmp1;
     }
     head = var;
  }
```

n=2
n=3 → 1
n=4 → 2 2
n=2 2
i=1
5=3 i<3
i=1×2 = 2
2×2 = 4
n-2

head = 1000



tmp = ~~1000~~ 2000
Var = ~~N~~ ~~1000~~ 2000
tmp1 = ~~N~~ 1000

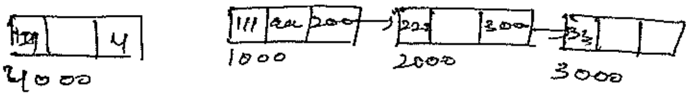
i = 1   j = 1

pf("enter the position").
sf("%d", &n);
swap(~~*head~~, n);

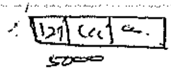
n=5   n = 3
i<5   j=1
2<5   1<=1
4<8   i=2
8 j++
2<=1

100
100   1000
1000

start = 1000
tmp = Start .



tmp = 1000



q = tmp
= 1000 .

q -> link = 1000

```
struct emp
 {
   int eno;
   char *ename;
```

Ctrl + F7
addwatch

```
addatbeg( )
{
  struct emp *tmp, *q;
  ~~tmp = start~~; ~~tmp~~ q = start;
  tmp = (struct emp *) malloc( sizeof (struct emp)) ;
  printf ("Enter the eno :");
  scanf (" %d", &tmp -> eno);
  printf( "Enter the ename:");
  scanf ("%s", &tmp -> ename);
  ~~tm~~ q -> link = tmp;
     start = tmp ;
}
add after ( pos)
 {
   struct emp *tmp ,
      for(i=0 ; i< pos-1 ;
         {
```

```
Void reverse( )
}
struct node *t1,*t2, *var,
    t1 = head;
    var = NULL
    while ( t1 != NULL)
    {
     ~~tmp ->~~ var
     t2 = var ;
     var = t1;
     t1 = t1 -> next;
     var -> next = t2;
    }
    head = var
```

$n = 2$
$i = n/2 = \boxed{1}$

i	1	2
n	~~n~~ 1	

$j=1 ; j <= \frac{n}{2} ; k = k * \frac{n}{2} = 2$ — 1

$j=1 ; j <= n/2 ; j++)$ $n = 2$ — 4 m.

n	2	4
lo	1	2

$n = 4$ $\boxed{j=1}$ 2

$j=1 ; j <= 4/2 ;$

1
2
4
8
16

$i=1$ $i^2 <= n$ $i++$

$\boxed{n}$ $\boxed{1 <= 4}$ $4 <= 2$ ~~2~~ 3

k

k_1 $\boxed{1}$

① www.edeserve.co.in

② www.geeksforgeeks.org ⇒ <u>Books</u>

Book Name for ds { Data structures and Algorithms Made easy

Narasimha Karumanchi

③ www.allitebooks.com

n, res, ele

case 3 4 : addatbeg( );
break;

case 5 : add after ( )
6.

aruna.somula@gmail.com

bhab.
i = 6
bharnakhiratkar9@gmail.com

j = 5

i = 5 k = ~~4~~

5 i <= 10 i++

6

$\frac{n}{\Sigma}(1+2^2+3^2+4^2+\cdots n^2)$

$\frac{n}{\Sigma}(n^8_9$

5

S = 1 i = 1

n = 7

is = 1 n = 5

S = 1 n = 5

S <= 5

13 2

1 ≤ 3 4

i = ~~2~~ 3

s = 1 + 2 = 3

3 + 3 = 6

Kim..

m.

i = ~~1~~ ~~2~~ 3

s = ~~1~~ + 2 = ~~3~~

Kiran

S <= 7

S ≤

i = ~~1~~ 2

1 + 2 = 3

i = ~~2~~ 3

S = 2 + 1 = 3

3 + 3 = 6

n = 4

k

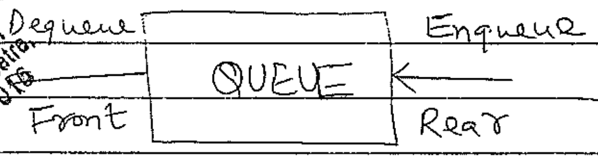
$\boxed{1+2}$

$\boxed{3+3}$ 3

i = ~~1~~ ~~2~~ 3

s = ~~3~~ + 3 = 6

K Ri



Queue model

A queue is a one type of ds where insertion of elements at one end and deletation from the other hand.

The ends from where the elements are inserted is Called rear end.

The end from where the elements are deleted is Called front end.

queue is also Called as first in first out (FIFO) data structure.

Applications

Direct Application:

① Operating System schedule jobs (with equal priority) in the order of arraval (eg: a print queue), simulation of real-world queues such as ticket-Counter or any other FCFS scenario require a queue.

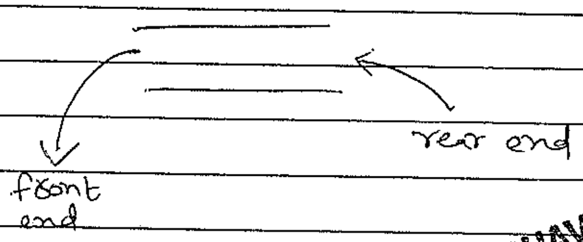
② Multiprogramming
③ Asynchronous data transfer (file IO, pipes, sockets)
④ Waiting times of customers at Call centers.
⑤ Determining no. of cashiers to have a Super market.
⑥ Anti virus checking on FCFS drive (File)

Page No
Date . \ \

* The different types of queues, ordinary queue, circular queue, Deque and priority queue.

Queues can be implimented either by using arrays or linked List.

for Suppose an ordinary queue is implementing by arrays, initially the front and rare position points to -1, indicating there is no elements in the queue.

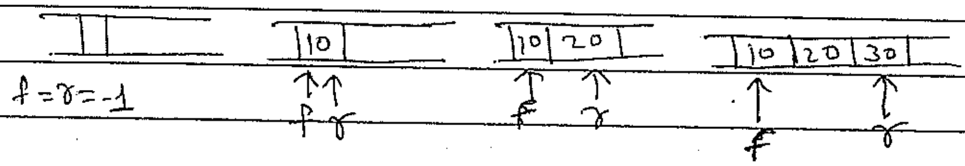


f = r = -1
MAX = 4

Whenever Insertion takes place rear will be incremented.
whenever deletion takes place front will be incremented.
When rear reaches to the max position it will be overflow.

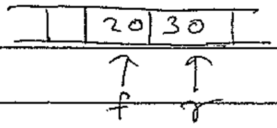
insert 10, 20 & 30

if (r == max - 1)

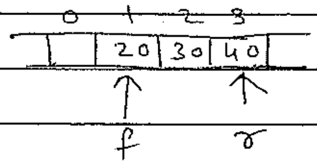


if we perform the delete operation, we need to check front position i.e front == -1. if this condition satisfy, it is underflow.

* implement delete operation.



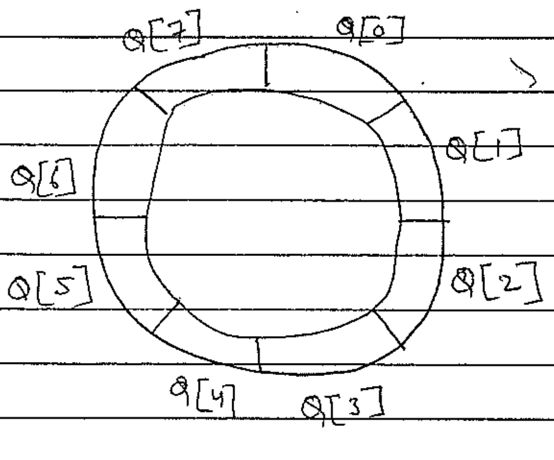
at this position two memory blocks are free.
insert 40 in the queue by checking the condition



⇒ insert 50 in the queue. it will check the condition as the condition not satisfied, it will be over flow. but one of the memory block is free but still there is a wastage.

onse again insertion will not ~~be~~ takes place until the front end rare reaches to the same position.

Circular queue:



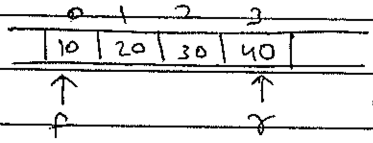
* The circular queue will overcome the drawback of ordinary queue i.e wastage of memory blocks.

* in this also initially the front & rare positions points to -1.

* At the time of insertion it check the conditions.

insert 4 elements in the queue

f=-1
r=-1



if ((f==0) && (r==max-1)) || (f = r+1)

ॐ

24/01/16

if (font == (rear+1) % Max)

```
# include<stdio.h>
#define MAX 5

int cqueue_arr[MAX];
 int front = -1;
 int rear = -1;

main()
{
     int choice;
while(1)
{
    pf(" 1. insert \n");
    pf (" 2. Delete \n");
    pf (" 3. Display \n");
    pf(" 4. Quit \n");
    pf(" Enter choice :");
    sf (" %d", & choice);

    switch( choice)
    {
      case 1:
              insert() ;
              break;

      case 2:
              del();
              break;

      case 3:
              display() ;
              break;

      case 4:
              exit(1);

     default :
              pf(" wrong choice \n") ;
        } // end of switch
      }// end of while
    }// End of main()
```

```
insert()
{
  int added_item;
  if((front == 0 && rear==MAX-1) ||(front== rear+1))
  {
     pf(" Queue overflow \n");
     return;
  }
  if( front == -1) // if queue is empty
   {
      front = 0;
      rear = 0;
   }
  else
     if (rear == MAX-1) //rear is at last position of queue

         rear = 0;
    else
         rear = rear+1;

pf(" input the element for insertion in queue:");
 sf("%d", &added_item);
  Cqueue_arr[rear] = added_item ;
} // end of insert()

del()
{
  if( front == -1)
   {
     pf(" Queue underflow \n");
     return;
   }
```

```
pf(" Element deleted from queue is: %d \n", ~~cqueue~~
                              cqueue_arr[front]);
if (front == rear) //queue has only one element
{
    front = -1;
    rear = -1;
}
else
    if (front == MAX-1)
        front = 0;
    else
        front = front + 1;
} //end of del()

display()
{
   int front_pos = front, rear_pos = rear;
   if (front == -1)
   {
      pf(" Queue is Empty");
      return;
   }
   pf(" Queue elements");
   if ( front_pos <= rear_pos)
      while( front_pos <= rear_pos)
      {
         pf(" %d", cqueue_arr[front_pos]);
         front_pos++;
      }
   else
      {
         while( front_pos <= MAX-1)
```

Page No.
Date .   \   \

```
{
   pf("%d", cqueue_arr[front_pos]);
   front_ pos++;
 }


front_pos =0;
while( front_pos <=rear_pos)
 {
     pf ("%d", cqueue_arr['front_pos]);
     front _ pos ++;
  }
} // end of else
  pf("\n");
} // end of display()
```

priority Queues

The priority queue is a special type of data structure in which items can be inserted or deleted based on the priority.

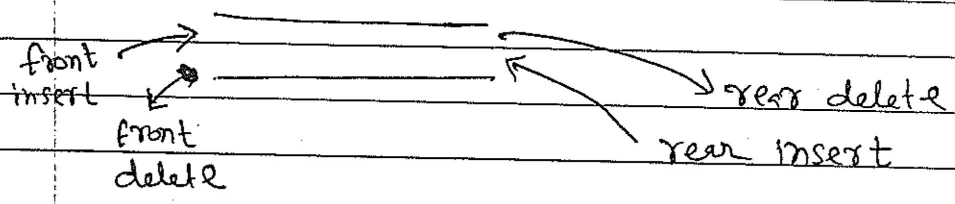
Always one element with heighest priority is processed before processing any of the lower priority elements.

if the elements in the queue are of Same priority, then the element which is inserted into the queue is processed.

priority queues are used in job scheduling algorithms in the design of operating system where the jobs with ~~higher~~ higher priorities have to be processed first.

```
display()
{
   struct node *ptr;
    ptr = front;
    if ( front == NULL)
        pf ("Queue is empty");
    else
     {
        pf (" Queue is : \n");
        pf (" priority item \n");
       while( ptr != NULL)
        {
           pf ("%5d %5d \n", ptr->priority, ptr->info);
           ptr = ptr->link;
        }
   }  // end of else
}  // end of display()
```

Dequeue



in this dequeue extra two operations will be takes place like front insertion & rear deletation.

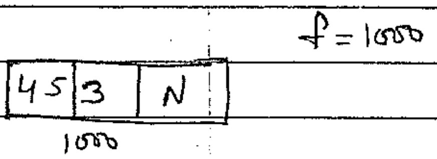
This will be implemented in different ways like input restricted & o/p restrected.

in the i/p restrected if the insertion takes place at 1st time, there is no chance of insertion.
in the o/p restricted if we make a rear

```
//Queue is empty or item to be ~~add~~ added has priority more than
 1st item */

if ( front == NULL || item_priority < front-> priority)
{
      tmp->link = front;
       front = tmp;
}
   else
   {
       q = front;
while( q->link != NULL && q->link ->priority <= item_priority)
           q = q-> link;
             tmp -> link = q-> link;
              q -> link = tmp;
       } //end of else
} // End of insert()

 del()
 {
       struct node *tmp;
       if ( front == NULL)
           pf( "Queue Underflow");
       else
         {
             tmp = front;
              pf (" Deleted item is %d", tmp-> info);
              front = front->link;
              free ( tmp);
          }
  } // end of del ()
```



```
switch(choice)
{
     Case 1:
              insert();
              break;

     Case 2:
              del();
              break;

     Case 3:
              display();
              break;

     Case 4:
              exit(1);
     default:
          pf(" wrong choice !! \n");
     } // end of switch
   } // end of while
} // end of main()

insert()
{
   Struct node *tmp, *q;
   int added_item, item_priority;
  tmp =(Struct node *) malloc (sizeof(struct node));
  pf(" input the item value to be added in the queue: ");
   Sf("%d", &added_item);
   pf(" Enter priority: ");
   Sf("%d", &item_priority);
   tmp->info = added_item;
   tmp->priority = item_priority;
```

The priority queues are classified into groups,

Ascending priority queue:
while deleting an element from the queue, only the smallest element is removed first.

Descending priority queue:
while deleting an element from the queue, only the largest element is deleted first.

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

Struct node
{
    int priority;
    int info;
     Struct node *link;
 }* front = NULL;

main()
 {
    int choice;
    while(1)
    {
        pf(" 1. insert \n");
        pf(" 2. delete \n");
        pf(" 3. Display \n");
        pf(" 4. Quit \n");
         pf(" enter your choice:");
         sf(" %d", & choice);
```

Page No.
Date.

deletation 1 time we have to perform remaining deletion from the front.

FB: KIRAN SIR's C, C++, Data Structure Students

ॐ

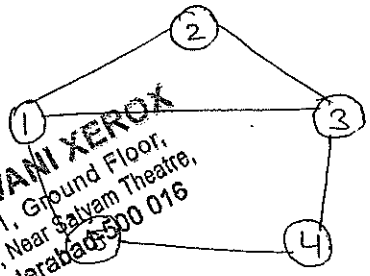
Graphs

A graph G consist of two things :
i.e G=(V,E).

1. A set V of elements called node (or points or vertices)

2. A set E of edges such that each edge e in E is identified with a unique (unordered) pair [u,v] of nodes in V, denoted by e=[u,v].

Ex:

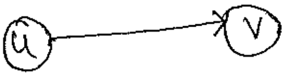


$V=\{1,2,3,4,5\}$

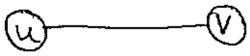
$E=\{(1,2),(1,5),(1,3),(5,4),$
$(4,3),(2,3)\}$

Terminologies in graphs

Directed Edge
- Ordered pair of Vertices (u,v)
- first vertex u is the origin
- 2nd vertex v is the destination
- Ex: one way road traffic



Undirected Graph:
- unordered ~~parr~~ pair of vertices (u,v)
- Ex: Railway line

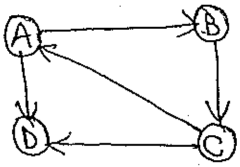


~~Undirected Edge~~

Directed Graph:

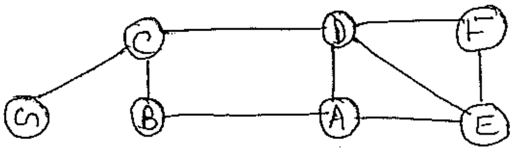
Directed graph is a graph which consist of directed edges.

- all edges are directed
- Ex: route network



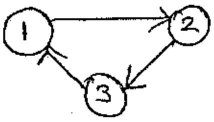
Undirected Graph:

- all the edges are undirected
- Ex: flight network

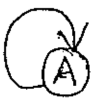


when an edge connects two vertices, the vertices are said to be adjanent to each other and that the edge is incident on both vertices.

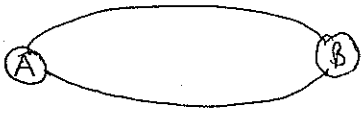
* a graph with no cycles is called a tree. A tree is an acyclic connected graph



* a self loop is an edge that connect a vertex to itself



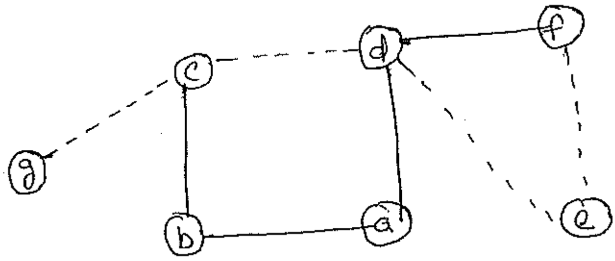
*Two edges are parallel if they connect the same pair of vertices



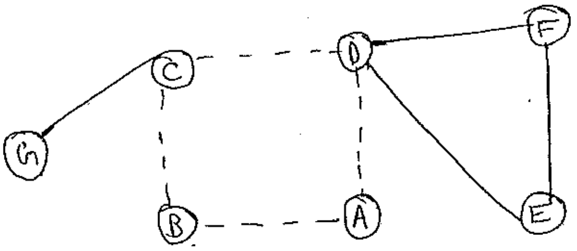
* Degree of the vertex is the no. of edges incident on it.

* A subgraph is a ~~Subgraph 2~~ subset of a graphs edge(with associated vertices) that forms a graph.

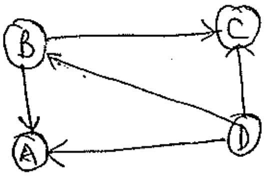
* A path in a graph is a sequence of adjancent vertices. simple path is a path with no repeated vertices. in the graph below dotted line represent a path from G to E

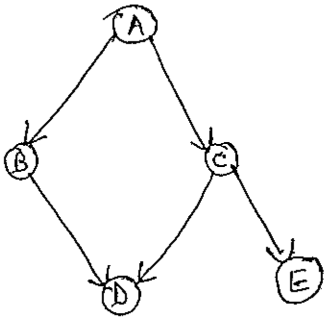


* a cycle is a graph where 1st and last vertices are same. A simple is a cycle with no repeated vertices or edges (except the 1st and last vertices)



* A Directed acyclic graph(DAG) is a directed graph with no cycles





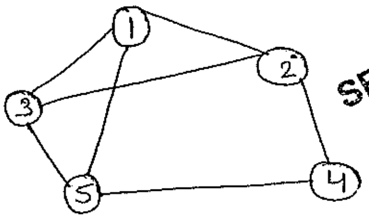
Representation of Graph :

Graphs can be represented as

Adjacent list representation.
Adjacent matrix representation.

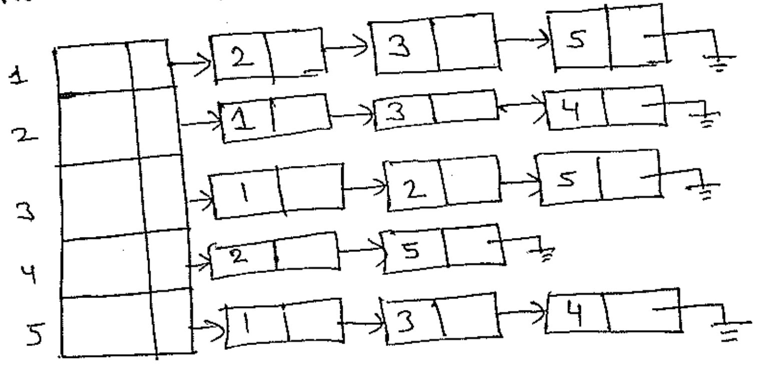
Adjacent List representation:

An adjacent list representation of a graph G = { V, E } consists of an array of adjacency list denoted by adj of v list. i.e. adj[v].



The adjacency list representation of the above graph is,

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



We are representing Linked structures.

Disadvantage it takes O(n) time to determine whether there is an arc from vertex i to vertex j.

Adjacent matrix representation:

An ordinary matrix representation of a graph G = (V,E) is a matrix A (a ij)

such that,

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

i.e., for above example the adjacency matrix is

	1	2	3	4	5	6
1	0	1	1	0	1	
2	1	0	1	1	0	
3	1	1	0	0	1	
4	0	1	0	0	1	
5	1	0	1	1		

advantage:

simple to implement.

Disadvantage:

takes $O(n^2)$ space to represent a graph

time $O(n^2)$

Graph traversal Techniques:

There are two types of technique of visiting every node in a graph.

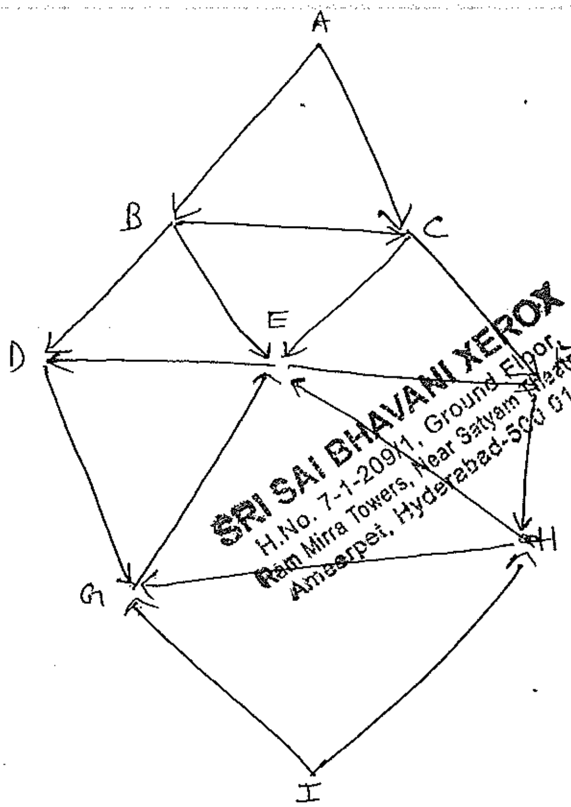
The two technique

① BFS → Breadth-first search

② DFS → Depth-first search.

DFS & BFS are two graph traversal algorithms, both start at Some vertex V and then visit all vertices reachable from V.

if there are vertices that remain unvisited, that is, if there are vertices that are not reachable from V, then the only way they can be visited is by starting new traversal by selecting new vertex.



Given input graph G(V,E) and a source vertex S from where the ~~we~~ checking starts.

* The BFS checks systematically travers the edges of G to explore every vertex i.e reachable from S. then we exam all the vertices neighbour to the source vertex then we travel neighbour vertex and so on.

a ~~quirey~~ queue is used to keep track of the progress of ~~the of~~ traversing the neighbours.

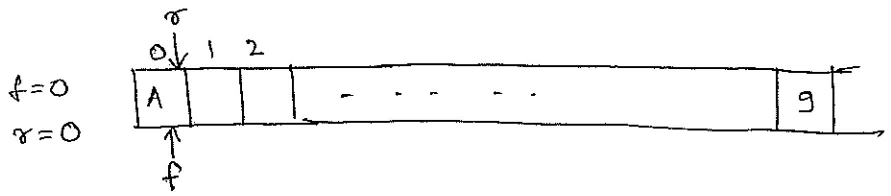
* The linked list (edges & List) representation of the graph starting from the source vertex A will takes place.

Algo:

Step 1: initially push A to the queue

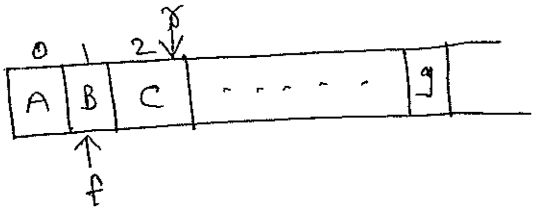
Initially

Vertex	Adjacency List
A	B, C
B	C, D, E
C	E, F
D	G
E	D, F
F	H.
G	E,
H	E, G
I	G, H

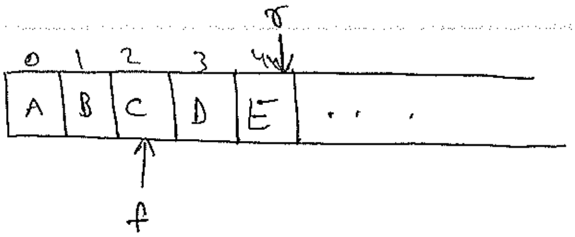


Step 2:

pop the front element A from the queue by incrementing front = front + 1 and display it.
Then push the neighbour vertices of A to the queue by incrementing rear = rear + 1, if it is not in the queue



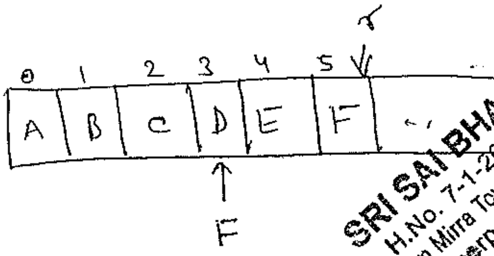
Step 3:
pop the front element from B ~~of H~~ from the queue and disploy it. then add the neighbour ends of B to the queue if it is not in the queue.



one of the neighbouring element of C of B is present in the queue so C is not added to the queue.

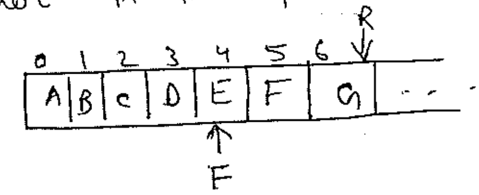
step4:

Remove the front element C and display it. and add the neighbour vertices of C if it is not present in queue.



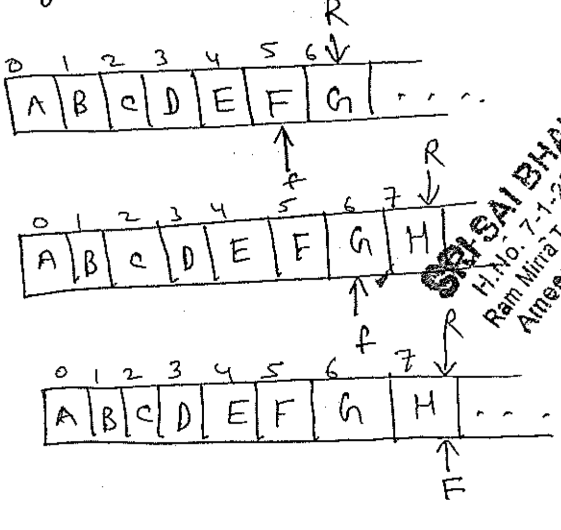
step5:

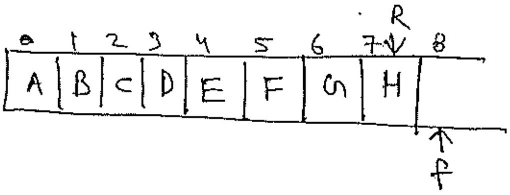
Remove the front element D and put the neighbours of D if it is not in the queue.



step 6:

again this process repeated until front >rear, i.e remove the front element E of the queue and add the neighbouring vertex if it is not present in the queue.





so, A,B,C,D,E,F,G,H is the BFS traversal of the graph.

Algorithm

1. Input the vertices of the graph and its edges G= (V,E)
2. Input the Source vertex and assign it to the variable S.
3. Add or push the source vertex and assign it to the variable S.
4. Repeat the steps 5 & 6 until the queue is empty (i.e., front > rear)
5. pop the front element of the queue and display it as visited.
6. push the vertices, which is neighbor to just, popped element, if it is neighbor to just, popped element, if it is not in the queue and displayed (i.e., not visited.
7. exit.

DFS technique

The depth first search (DFS), as its name suggest, is to Search deeper in the graph, whenever possible. Given an input graph G=(V,E) and a source vertex S, from where the searching starts. First we visit the Searching node.

Then we travel through each node along a path, which begins at S. That is we visit a neighbour vertex of S and so on.

The implementation of BFS is almost same except a stack is used instead of the queue.

Consider the graph and its linked list representation.

suppose the source vertex is I.

Step 1:

initially push I on to the stack

Stack: I

Display:

Step 2:

pop and display the top element, and then push all the neighbors of the poped element (i.e., I) onto the stack, if it is not visited (or displayed or not in the stack.)

Stack: G, H

Display: I

Step 3:

pop and display the top element and then push all the neighbours of popped the element (i.e., H) onto top of the stack, if it is not visited.

Stack: G, E

Display: I, H

The popped element H has two neighbours E and G. G is already visited, means G is either in the stack or displayed. Here G is in the stack. So only E is pushed onto the top of the stack.

Step 4:

pop and display the top element of the stack. push all the neighbors of the popped element on to the stack, if it is not visited.

Stack : G, D, F
Display : I, H, E

Step 5:
pop and display the top element of the stack, push all the neighbors of the popped element onto the stack, if it is not visited.

Stack : G, D
Display : I, H, E, F

The popped element (or vertex) F has neighbor(s) H, which is already visited.
Then H is displayed, and will not be pushed again on to the stack.

Step 6: The process is repeated as follows.

Stack : G
Display : I, H, E, F, D

STACK : // Now the stack is empty
Display : I, H, E, F, D, G.

So, I, H, E, F, D, G is the DFS traversal of graph from the source vertex I.

Algorithm

1. input the vertices and edges of the graph G=(V,E).
2. Input the source vertex and assign it to the variable S.
3. push the source vertex to the stack.
4. Repeat step 5 & 6 until the stack is empty.
5. pop the ~~top~~ element of the stack and display it.
6. push the vertices which is neighbor to just popped element, if it is not in the queue and displayed (i.e; not visited).
7. Exit.

Quick sort:

11, 2, 9, 13, 57, 25, 17, 1, 90, 3

This quick sort is like divide & Concure process. in this we have to take one key element. The key element can be 1st element, center element or last element.

Suppose you taken the key element is 11.

1st way to split the array in two parts.

The key left side less than the pivot.

Right side greater than the pivot.

$x[i]$

Key (pivot) → 11, 2, 9, 13, 57, 25, 17, 1, 90, 3

↑ i (at 11) ↑ j (at 3)

we have to move from left side i until > than the pivot element occurs.

1st x_{key} Compared with x_i (it self).

as the condition not satisfied (not greater), so ~~move~~ increment the index of i untile > than the key element.

11, 2, 9, 13, 57, 25, 17, 1, 90, 3

↑ i (at 13) ↑ j (at 3)

then stop incrementing of i and start decrementing j until less than the pivot element.

as $x[j]$ is less than the pivot element so stop decrementing of j.

as the i index less than j, exchange $x[i]$ and $x[j]$

11, 2, 9, 3, 57, 25, 17, 1, 90, 13

↑ i (at 3) ↑ j (at 13)

increment i until i > pivot.

```
11, 2, 9, 3, 57, 25, 17, 1, 90, 13
          ↑                     ↑
          i                     j
```

stop incrementing of i & decrement j.

decrement j until j < pivot

```
11, 2, 9, 3, 57, 25, 17, 1, 90, 13
             ↑            ↑
             i            j
```

i index < j (i<j)

so swap x[i] & x [j]

```
11, 2, 9, 3, 1, 25, 17, 57, 90, 13
             ↑          ↑
             i          j
```

proceed incriment of i, stop decrement as x[i] > x[pivot].

```
11, 2, 9, 3, 1, 25, 17, 57, 90, 13
                ↑       ↑
                i       j
```

stop decrement of j. after decrementing

```
11, 2, 9, 3, 1, 25, 17, 57, 90, 13
             ↑  ↑
             j  i
```

as i index > j, so exchange x [j] with x[pivot].

```
1, 2, 9, 3, 11, 25, 17, 57, 90, 13
```

* at this position the piovt element left will be less element and right will be greater elements.

* once again make a quick sort for the pivot left element as well as pivot right element.

This process implimented with the help of recursion.

Analysis of Quick sorts:

Running time of partition:

Let $n = high - (low + 1)$,

we observe that during the execution of partition, we always have $j > i-1$, because all keys below i are always smaller than or equal to pivot and all the keys above j are always larger than or equal to pivot.

This implies that the running time of partition is $O(n)$.

Running time of quick sort: "worst-case".

in worst case the array is always partitioned into sub arrays of sizes 1 and n-1.

$$T(n) = \begin{cases} O(1) & \text{if } n=1 \\ T(n-1)+O(n) & \text{if } n>1 \end{cases}$$

This recurrence can be shown as a tree

↑ n-1
n-2
n-3
-
-
- ↓

n-1
n-2
n-3
.
.
.

$O(n^2)$

There fore the worst case running of Quick sort is $O(n^2)$.

Best case Analysis:

The best case arises when the array is always split in the midd

then,

$$T(n) = T([n/2]) + T([n/2]) + o(n) .$$

$$= 2T(n/2) + d(n)$$

$$= 2(2T(n/4) + d(n/2)) + d(n)$$

$$= 4T(n/4) + 2d(n)$$

$$= 4(2T(n/8) + d(n/4)) + 2.d.n$$

$$= ----$$

$$= ----.$$

$$= 2^k . T(n/2^k) + k.d.n.$$

$$= 2^k . T(n/2^k) + k.d.n$$

$$= n.T(1) + d.n.k \qquad \text{since } 2^k = n$$

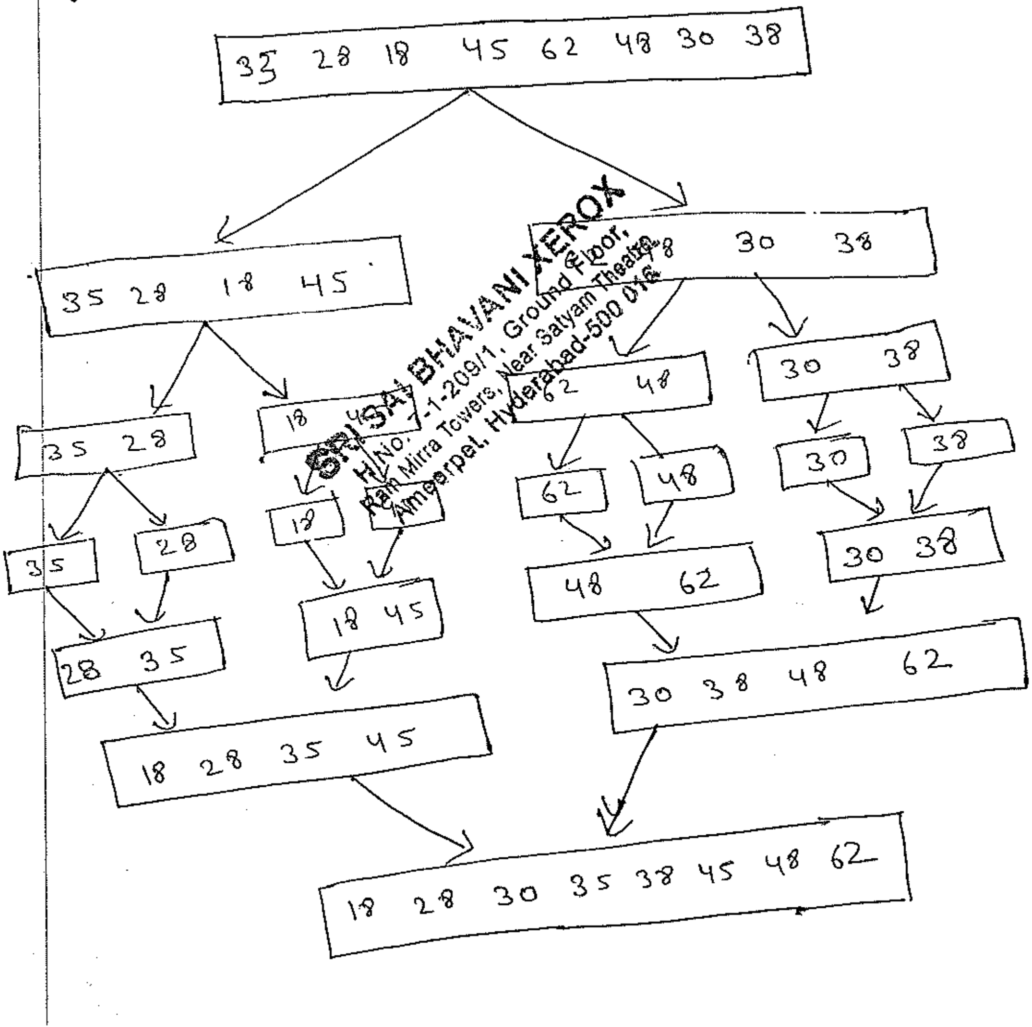
$$= n.1 + d.n.\log(n) \qquad \text{because } n = 2^k$$

$$T(n) = O(n \log n)$$

```
#include <Stdio.h>
#include< conio.h>
int  Split ( int *, int, int);
Void main()
{
  int arr[10] = {11,2,9,13,57,25,17,1,90,3);
   int i;
    Void   quicksort( int * , int , int);
             clrscr();
              pf (" Quick sort. \n");
         pf(" ~~*~~\n Array before sorting: \n");
            for(i=0, i<=9 ; i++)
                 pf ("%d\t", arr[i]);
```

```
quicksort( arr , 0, 9 ;
    Pf ( " \n Array  after sorting:\n");
      for( i =0 ; i< =9 ; i++)
            Pf (" %d \t", arr[i]);
         getch();
   }
void  quicksort (int a[], int lower, int upper)
  {    int  i ;
       if ( upper >lower)
         {
            i = split ( a, lower, upper) ;
            quicksort ( a, lower , i-1) ;
              quicksort ( a , i+1 , upper);
            }
       }
  int split(int a[] , int lower, int upper)
     {    int i , p, q, t;
          p = lower + 1;
           q = upper ;
           i = a[lower];
        while ( q >= p)
          {    while( a[p] < i) // L to R
                   p++;
         while ( a[q] >i) // R to L
                     q--;
               if( q> p) // swapping q & p elements
               {  t = a[p];
                  a[p] = a[q];
                }  a[q] = t;
}
 t = a[lower]; // swapping with pivot element
  a[lower] = a[9];
   a[9] = t;
   return 9;
}
```

Merge Sort



Explanation:

Consider List of element in it is larger element. we create different arrays based in the radix base in the digit in a no. (in a decimal number system 0 to 9)

* ~~[illegible]~~ Different arrays each containing different slots.

* First step we use the least significant digit to decide which bucket has to go in for number.

* & we have only one digit in bucket. consider the least significant.

* This is similar to Hashing where we have using digit lower most digit number decide which bucket goes builds hashing.

* write down no. from 1st bucket to 2nd bucket 30 21 32 . . . 29

* In subsequent iterations instead contains LSD. we consider the next digit ZERO's there are in goes ZERO slot.

~~Secco~~

* second digit zero go for slot '0'.

* combine together and place in order 4, 5, 9 . . .

* They are present maximum no. of digits for all the no. only two so radix sort granty is that once two iteration are performed are the numbers present in the order.

So, Radix sort achieve the sorting much more efficiently. it is not dendendent on no. of elements rather it depends on no. of digits, maximum no of digits in an element however u keep in mind raxid sort requres lot additional memory all the buckets each bucket have multiple elements are require to perform this radix sort operations other than the radix sort efficient tech.. compare to bubble and selection.

```
main( )
{
  int  a[]={9,47,21,32,5,13,27,4,54,76);
   int  arr[10] [3] , K, dig, i;

   dig=1;
   for(K=0; K<=1; K++)
    {
        initq(arr);
        radix(a,arr,10,dig);
         combine(a,arr);
         dig *=10;
     }
     for(i=0; i<10; i++)
          pf(" %3d", a[i]);

      }
      initq( int arr[10] [3])
       {     int i,j;
              for(i=0; i<10; i++)
               {   for(j=0; j<3; j++)
                      arr[i][j] = 0;
                }
           }
      radix(int a[], int arr[] [3], int n, int dig)
       {
          int i,j,key;
           for(i=0; i<n; i++)
            {   key = (a[i]/dig) % 10;
                for(j=0; j<3; j++)
                 {   if(arr[key][j]==0)
                      {    arr[key][j]=a[i];
                            break;
                       }
                  }
              }
        }
```

```
Combine( int a[], int arr[][3])
 {
      int  i, j, x=0;
       for ( i=0; i<10; i++)
        {
            for(j=0; j<3; j++)
             {  if (arr[i] [j]  !=0)
                {
                   a[x] = arr[i] [j];
                   x++;
                }
             }
        }
 }
```

Searching & Hashing

Searching is a process of checking or finding an element from the list of elements.

There are two type of checking process

1. Linear Search or sequential.
2. Binary search.

Suppose that the search item is in the list. Then the no. of key comparision depends on where in the list the search element is located.

if the search item is in the first element of L, we make only one key comparision. This is the best Case. On the other hand, if the search item is the last element in the list, the algo makes n comparision This is the worst Case.

The best & the worst cases are not likely to occure every we apply the sequential search on L.

So it would be more helpful if we could determine the behaviour of the algo. i.e we need to determine the average no, of key comparisions the sequential search algo. makes in the sucessful case.

To determine the average no. of Comparision in the sucessful Case of the sequential search algo.

① consider all possible Cases.

② Find the no. of Comparisions for each case.

③ Add the no. of Comparision and divide by the no. of cases.

if the search item, called the ~~largest~~ target, is the 1st element in the list. one Comparision is required. if the ~~ta~~ target is second element in the list, k Comparision are required. we assume that the target Can be any element in the list. i.e all list elements are equally likely to be the target. Suppose that there are n elements in the list. The following expression gives the average no. of Comparisions:

$$\frac{1+2+\cdots+n}{n}$$

it is known that

$$1+2+\cdots+n = n(n+1)/2$$

Therefore, the following expression gives the averageno. of Comparision made by the sequential search in the sucessful Case.

$$\frac{1+2+\cdots+n}{N} = \frac{1}{n} \quad \frac{n(n+1)}{2} \quad \frac{(n+1)}{2}$$

This expression shows that, on average the sequential search searches half the list. it, thus follows that if the list ~~is~~ size is 1 000 000 on average, the sequential search makes 500000 Comparisions. As a result, the sequential search is not efficient for large lists. O(n).

* To make it better in performance, go for binary checking.
* in binary checking the list must be in sorted order.

* As you can see, the sequential search is not efficient for large lists because on average, the sequential search searches half the list. we therefore describe another search algo. called binary search which is very fast.

However a binary search can be performed only on ordered lists. we there fore assume that the list is ordered. The binary search algo use the divide and conqure tech. to search the list.

First the search item is compared with the middle element of the list. If the search item is found the search terminates. If the search item is less than the middle element of the list, we restrict the search to the 1st half of the list, otherwise we search the second half of the list.

0	1	2	3	4	5	6	7	8	9	10	11
4	8	19	25	34	39	45	48	65	75	89	95

```
int first = 0;
 int last = length-1;
  int midl;
  int found = 0;
  while ( first <= last && !found)
   {   mid = ( first + last) / 2;
       if ( list[mid] = item)
         found = 1;
         else if(list[mid] >item)
         last = mid - 1;
          else
              first = mid + 1;
     }
```

```
if (found)
    return mid;
else
    return -1;
}
```

* find out an element 89.

Iteration	first	last	mid	List[mid]	No. of Comparision
1	0	11	5	39	2
2	6	11	8	66	2
	9	11	10	89	1 (found is true)

the item is found at 10th position and the total no. of Comparision is 5.

Q find out an item 34

Iteration	first	last	mid	Last[mid]	No. of Comparision
1	0	11	5	39	2
2	0	4	2	19	2
3	3	4	3	25	2
4	4	4	2 34	34	1

* find 22

Iteration	First	Last	mid	Last[mid]	No. of Comp.
1	0	11	5	39	2
2	0	4	2	19	2
3	3	4	3	25	2
4	3	2			(Condition not satisfy)

PERFORMANCE OF BINARY SEARCH:

Suppose that L is a Sorted list of size 1024 and we want
to determine if an item X is in L.
from the binary search algo. it follows that every iteration of
the while Loop cuts the size of the while loop will have at
most 11 iterations to determine whether X is in L.
Because every iteration of the while Loop makes two item
(key) comparision. that is X is compared twice with the element
of L, the binary search will make almost 22 comparisions
to determine whether x is in L. On the other hand, recall
that a sequential search on average will make 512
Comparisions to determine whether X is in L.

To better understand how fast binary search is compared
with sequential search,

Suppose that
L is the size 1048576. Because $1048576 = 2^{20}$, it follows
that the while loop in a binary search will have at most
21 iterations to determine whether an element is in L.

every iteration of the while loop makes two key (that is, item) Comparisions. There bore to determine, whether an element is in L, a binary search makes at most 42 item Comparisions.

Note that $40 = 2*20 = 2 * 2\log_2 2^{20} = 2 * \log_2 (1048576)$

in general, suppose that L is a Sorted list of size n. More over, suppose that n is a power of 2, that is $n = 2^m$ bor som non negative integer m. After each iteration of the for loop about half the element are left to search, that is the search sublists for the next iteration is half the size of current sublist.

Algo	Sucessful Search	Unsucessful Search
		$n = O(n)$
Sequential Search	$(n+1)/2 = O(n)$	
Binary search	$2\log_2 n - 3 = O(\log_2 n)$	$2\log_2(n+1) = O(\log_2 n)$

* in this two techniques also, we are not finding the element in a single attempt. so in order to find in single attempt we go bor hashing techniques.

ॐ

01/02/16

The searching time of the each searching technique, that were discussed ~~the~~ in the privious section, depends on the comparison i.e., in Comparisions required for an array Awith n elements. To increase the efficiency, i.e., to reduce the searching time, we need to aviod unnecessary Comparisions.

Hashing is a technique where we can compute the location of the desired record in order to retrive it in a single access (or Comparision).

Let there is a table of n employee records and each employee record is defined by a unique employee code, which is a key to the record and employee name. if the key (or employee code) is used as the array index, then the record can be accessed by the key directly. if L is the memory location where each record is related with the key. if we can locate the memory address of a record from the key then the desired record can be retrived in a single access. For notational and coding convenience, we assume that the keys in K and the address in L are (decimal) integers.

So the location is selected by applying a function which is called hash function or hashing function from the key K. Unfortunately Such a function H may not yield different values (or index or many address); it is possible that two different keys K1 and K2 will yield ~~hat~~ the same hash address. This ~~sis~~ situation is called as Hash Collision.

Hashing functions are of different types:

① division method
② mid square method
③ folding method

// Division Method

Table is an array of database file where the employee details are stored. Choose a no. m, which is larger than the no. of keys K i.e. m is greater than the total no. of records in the TABLE.

The no. m is usually choosen to be prime no. to minimize the collision.

The Hash function H is defined by H(K) = K (mod m)

where H(K) is the hash address (or index of the array) and here K (mod m) means the remainder when K is divided by m.

For example:

Let a company has 90 employees and 00, 01, 02, ..., 99 be the two digit 100 memory address (or index or hash address) to store the records. we have employee code as the key.

Choose m in such a way that it is greater than 90. suppose

m = 93

Then for the followcing employee code (or key K)

H(K) = H(2103) = 2103 (mod 93) = 57

H(K) = H(6147) = 6147 (mod 93) = 9

H(k) = H(3750) = 3750 (mod 93) = 30

Hash address	Employee Code (keys)	Employee Name and Details
0		
1		
2		

9	6147	[illegible]

30	3750	Swapna

57	2103	Bunty
90		

* Mid Square method:

More interesting hashing function in this first we convert the hashing function into convert the no. A to Z to use identifiers for digits or for nos. use A to Z and 27 to 28 we use identifiers 0 to 9 respectively identifier A=1 ... Z, 0=27, 1=28, 9=36.

We have to find octal equivalent of each this identifier (character) and then we square the octal equivalent the result of this squaring option is also stored in octal no. System. What is the octal no. system octal no. system is one in which the no. of octal (0 to 7) no.'s are stored as base of 8

Consider an example:

X	X^2	$(X')^2$
A	01	1
B	02	4
---	---	---
Y	31	1701
Z	32	2000
0	33	2101
1	34	2204
A1	134	20420
A 2	135	20711
CAT	030124	125620

What we define for more no. of employees.

Folding method:

Folding

→ Distribute digits in Multiple partition.
→ Excluding last make all partition equal.
→ Add partition values to get the hash values.

Folding another technique in this method we divide the digits within the no. in partition so if your 5 digits no. in three partition 1 of 2 digits and another → remainder of two digits and last one one digit and we add this digits together and print has a hash value.

No. of digits	partition
if the digit is 1	1
2	2
3	1,2 (two partition)
4	1,1,2 (remainder of two digits)
5	2,2,1
6	1,1,1,1,2

Ofcourse for large no. of digits we have multiple possibility are possible of equal size you just decide which way to use. It is not necessary to divide a six digit no. in two partition of 3 digits. each you can also divide 3 partition of two digits each or 6 partition of 1 digit each.

But more the no. of digits in a partition more the randomiza-tion you will in most cases so calculate based on partition data.

7	3,3,1
8	3,3,2
9	2,2,2,2,1

K	2103	7148	12345
K1 K2 K3	21,03	71,48	12,34, 5
H(K)= K1+K2+K3	H(2103) 21+03=24	H(7148) 71+48=119	H(12345) = 12+34+5 = 51

$K_2, K_4, \ldots H(7148)$

K	2103	7148	12345
$K_1, k_2\, k_3$	21, 03	71, 48	12, 34, 5
Reversing $K_2\ K_4$	21 30	71, 84	12, 43, 5
$H(k) =$ $k_1+k_2+k_3$	$H(2103)$ $21+30=51$	$H(7148)$ $71+84=\overset{\text{ommit}}{\boxed{1}}55$	$H(12345) =$ $12+43+5=60$