DATA STRUCTURE

Siddharth Sir

VANDANA COPIERS
B-47,48, Smriti Nagar, Bhilai
9302186514,9770184741

90/-

S. No.	Date	Title	Page No.	Teacher's Sign / Remarks

classmate

Date ______
Page ______

UNIT-I

INTRODUCTION

17th Jan

Basic Terminologies

→ Data - The term data means a value or set of values e.g. marks of student etc.

→ Data Item - It means a single unit of values e.g. Roll no., name etc.

→ Elementary data items - Such data items which are not divided into sub items are called elementary data items.

→ Group items - Data items that are divided into subitems are group items.

→ Entity - It is something that has certain qualities, characteristics, properties or attributes that may contain some values e.g. student is an entity. The attributes of student may be roll no., name, address etc. The values of these attributes may be 100, Ram, 23/7/10 Bhilai.

→ Entity set - An entity set is a group of or set of similar entities e.g. employees of an organisation, students of a class etc.

→ Information - When the data are processed by applying certain rules, new processed data is called information. The data are not useful for decision making where as information is useful for decision making.

→ Field - It is a single elementary unit of information representing an attribute of an entity e.g. 1, 2, 3, 4 . . etc are represented by a single unit called roll no. field.

→ Record - It is a collection of field values of a given entity e.g. rollno, name, address etc. of a particular student.

→ File - Collection of records of the entities in a given entity set e.g. file containing records of a student of a particular class.

→ Key - It is one or more field(s) in a record that take(s) unique value and can be used to distinguish one record from the others.

Case I - When more than one fields may have unique values. In that case, there exists multiple keys, but at a time we use only one field as a key called primary key. The other(s) key(s) are called as alternate key(s).

Case II - There is no field that has unique values. Then a combination of two or more fields can be used to form a key. Such a key is called composite key.

Case III - There is no possibility of forming a key from within the record then an extra field can be added to the record that can be used as a key.

★ Data Structure

Structure means particular way of data organisation. So data structure refers to the organisation of data in computer memory or the way in which the data is efficiently stored, processed & retrieved, is called data structure.

Data structure simply means a structure that can be used to store a given collection of data in a computer memory.

The organised collection of data is called data structure.

D.S. = organised data + allowed opera?

Data may be organised in many different ways, the logical or mathematical model of a particular data is called data structure.
The choice of particular data model depends on two consideration -
i) It must be rich enough in structure to represent the actual relationship of the data in the real world.
ii) The structure should be simple enough that one can effectively process the data when necessary.

⇒ Classification of Data Structure

Depending on the arrangement of data, data structure may be classified as following -

i) According to its occurrence

a) Linear - In linear data structure the data items are arranged in a linear sequence like array. e.g array, Link list

b) Non-linear - In this D.S., the data items are not in a linear sequence e.g. Tree, Graph.

ii) Acc. to nature of size (memory)

a) Static - Static struct. are one whose size and struct. associated memory location are fixed at compile time. eg. array.

b) Dynamic - Dynamic struct. are one which expand or shrink as required during program execution & their associated memo location can also be changed eg. Link list.

iii) Homogeneous & non-homogeneous DS

a) The homogeneous D.S., all the elements are of same type e.g. array.

b) In Non-homogeneous D.S, the elements are may or may not be of the same type eg. record, structure.

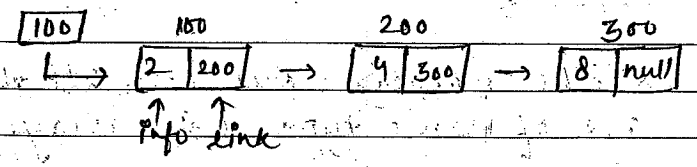
⇒ Types of D.S

1. Array - It is defined as set of finite no. of homogeneous elements or data elements. It means an array can contain

One type of data only. Array elements is stored in continuous memo. location.
eg. int a[5]

2	4	6	8	10
a[0]	a[1]	a[2]	a[3]	a[4]

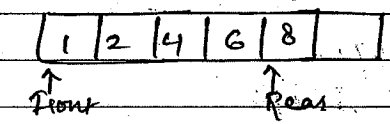
2. Link List - A list can be defined as a collection of variable no. of data items.
The link list are most commonly used D.S. The elements (nodes) of a list must contain atleast 2 fields, one for storing data or information and other for storing address of next element.
eg.



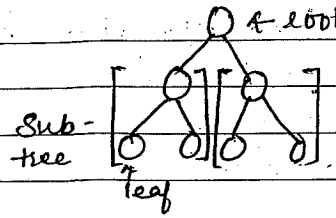
3. Stack - A stack is also an ordered collection of elements like arrays, but it has special feature that deletion & insertion of elements can be done only from one end, called TOP of the stack (TOP). Due to this property it is also called LIFO (Last In First Out) type of data type.
In stack, deletion operation is called POP and insertion operation is called PUSH.
e.g.

TOP 3	6
2	4
1	0

4. Queue - It is FIFO type of data type. In a queue new elements are added to the queue from end called Rear end & the elements are always removed from Front end.
eg.



5. Tree - It can be defined as finite set of data items (nodes). Tree is non-linear type of D.S, in which data items arranged & stored in shortened sequence. Tree represent the hierarchial r/s b/w various elements. In tree
i) There is a special node called root of tree.
ii) The remaining data items are participation which is called sub-tree, each of which is itself a tree.



6. Graph - It is a mathematical non-linear D.S capable of representing many kinds of physical structure. A graph G(V,E) where V is the set of vertices & E is the set of edges. An edge connects a pair of vertices & many have weights such as length, cost etc.

Types of Graph
i) Directed Graph
ii) Undirected Graph
iii) Connected Graph
iv) Not-Connected Graph
v) Simple Graph
vi) Multiple Graph

⇒ Operation performed on D.S

1. Traversing - Accessing (visiting) each record exactly once so that certain items in record may be processed. This accessing or processing is sometimes called visiting the record.

2. Searching - Finding the location of the record with a given key value, finding the location of all records which satisfy one or more condition.

3. Insertion - Adding a new record to the structure.

4. Deletion - Removing a record from structure

5. Sorting - Arranging the record in some logical order.

6. Merging - Combing the records of 2 different sorted files into a single sorted / unsorted file.

Algorithm

An algorithm is a well defined set of computational procedure that takes some value or set of values as I/P and produces some value or set of values as O/P.

An algorithm is finite set of instruction which perform a particular task. In addition, every algorithm must satisfy the following criteria :-

i) I/P - There are some I/P data which are externally supplied to the algorithm.

ii) O/P - There will be atleast some O/P.

iii) Definiteness - Each instruction/steps of the algorithm must be unambiguous.

iv) Finiteness - If we trace out the instruction/steps of an algorithm will terminate after a finite no. of steps.

v) Effectiveness - The steps of algorithm must be sufficient, based that it can easily carried out by a person mechanically using pen & paper.

⇒ Algorithmic notation

i) Comments - [ ]

ii) Variable name - It should always be in capitals.
eg. A, B.

iii) Assignment stmt - := or ←
eg. a := 10 or a ← 10 (Imp.)

iv) I/P - read: var name

v) O/P - write: msg/var name.

vi) Comparing for equality eg. a=b.

vii) Procedure - When we call a func^n it is called procedure.

viii) Control sequence

a) Sequence logic / Sequence flow
Instruction or modules are executed in the obvious sequence.

Step 1: —
step 2: —
:
Step n: —
[END]

b) Selection logic / Conditional flow
i) single alternate

If cond^n, then:
[Module A]
[end of if structure]

ii) Double alternate

If cond^n, then:

[ModuleA]
Else:
[Module B]
[end of if structure]

iii) Multiple alternates

If cond^n(1), then:
[Module A1]
Else if cond^n(2), then:
[Module A2]
:
Else if cond^n(M), then:
[Module AM]
Else:
[Module B]
[end of if structure].

c) Iteration logic / repeative flow.
2 types of loop.

i) Repeat for-loop

Repeat for K=R to S by T
[Module]
[end of loop].

ii) Repeat-- while-loop

Repeat while condition
[Module]
[end of loop].

21st Jan

Algorithm to find Max (DATA, N) of elements of an array DATA.

Algorithm: Find Max (DATA, N)
Given a non empty array DATA with N numerical values, this algorithm finds the location LOC & the value MAX of the largest element of data.

On algo. elements start from 1 and

1. [Initialize] Set K := 1, MAX := DATA[1], LOC := 1

2. repeat Step 3 and 4 while K <= N

3. If MAX < DATA[K], then:
Set MAX := DATA[K] and LOC := K
[End of if structure]

4. Set K := K+1 [update counter variable]
[End of Step2 loop]

5. write: MAX, LOC

6. Exit

Some mathematical notation

1. Properties of Logarithm

i. $\log_b(xy) = \log_b x + \log_b y$
ii. $\log_b(x/y) = \log_b x - \log_b y$
iii. $\log_b x^a = a \log_b x$
iv. $\log_b a = \log_x a / \log_x b$

2. Some Imp. series

3. Floor & Ceiling

i. $\lfloor 5.2 \rfloor = 5$ (floor)
ii. $\lceil 5.2 \rceil = 6$ (ceiling)
iii. $\lfloor 5 \rfloor = \lceil 5 \rceil = 5$
iv. $\lfloor -5.2 \rfloor = -6$
v. $\lceil -5.2 \rceil = -5$

Efficiency of Algorithm

Order of magnitude.

$1, \log n, n, n\log n, n^2, n^3, a^2, n!$
polynomial < exponent < factorial

eg. Calculating order of magnitude.

eg.

Statement	frequency count
main()	0
{	0
int n;	0
int y=2;	1
int z=3;	1
n = y+z;	1
return 0;	1
}	0

Order of mag 4 $\Rightarrow$ 1.

eg.
```
main()                      - 0
{                           - 0
 int i;                     - 0
 int sum=0;                 - 1
 for(i=1; i<=n; i++)        - n+1
   sum=sum+1;               - n
 pf("%d", sum);             - 1
 return 0;                  - 1
}
```

$\therefore 2n+4 \Rightarrow n$

eg.
```
main()                      - 0
{                           - 0
 int i=0;                   - 1
 int sum=0;                 - 1
 while(i>0)                 - 1
 { sum = sum+i;             - 0
   i = i+1;                 - 0
 }                          - 0
 pf("%d", sum);             - 1
}
```

$\Rightarrow 1$

eg.
```
void msum(a,b,c,m,n)        - 0
{ int i,j;                  - 0
  for(i=0; i<m; i++)        - m
    for(j=0; j<n; j++)      - m*n
      c[i][j] = a[i][j] + b[i][j]   - m*n
} pf(..)                    - 1
```

$\Rightarrow 2mn + m + 1 \Rightarrow mn$

eg.
```
for(i=1; i<=n; i=i*2)
```

let $n=16$

$1, 2, 4, 8, 16 \leq n$

$2^i = n$

$\log 2^i = \log n$

$i \log 2 = \log n$

$\boxed{i = \log n}$

★ Our focus should be on the increment stmt. and not on kaha tak chalega loop.

eg.
```
for(i=1; i<=n/2; i++)
```

$= n/2 \Rightarrow n$

eg.
```
for(i=n; i>=0; i=i/2)
```

for $n=16$.

16 8 4 2 1

$\therefore$ Order of magnitude = $\boxed{\log n}$

★ If the increment statement is based on multiplication or division then order of magnitude is logarithmic.
Base of log is not imp.

Q.
```
for(i=1; i<=n/2; i++)       - n/2
  for(j=1; j<=n; j=j*2)     - log n
```

$\Rightarrow \boxed{n \log n}$

* If a time complexity of any prog is calculated then the equivalent order of magnitude taken is the largest term which contributes the major part

eg. $5n^2+6n+12$

Here the term $5n^2$ contributes the major part in time complexity.

→ A major criteria for a good algo. is its efficiency – i.e., how much time and memory are required to solve a particular problem. There is seldom (rare) a single algo. for any problem. So there may be many algo. are ~~refu~~ required to compare this algo. and recognize the best one.

When comparing diff. algo. that solve the same problem, you often find that one algo. is an order of magnitude more efficient than the other

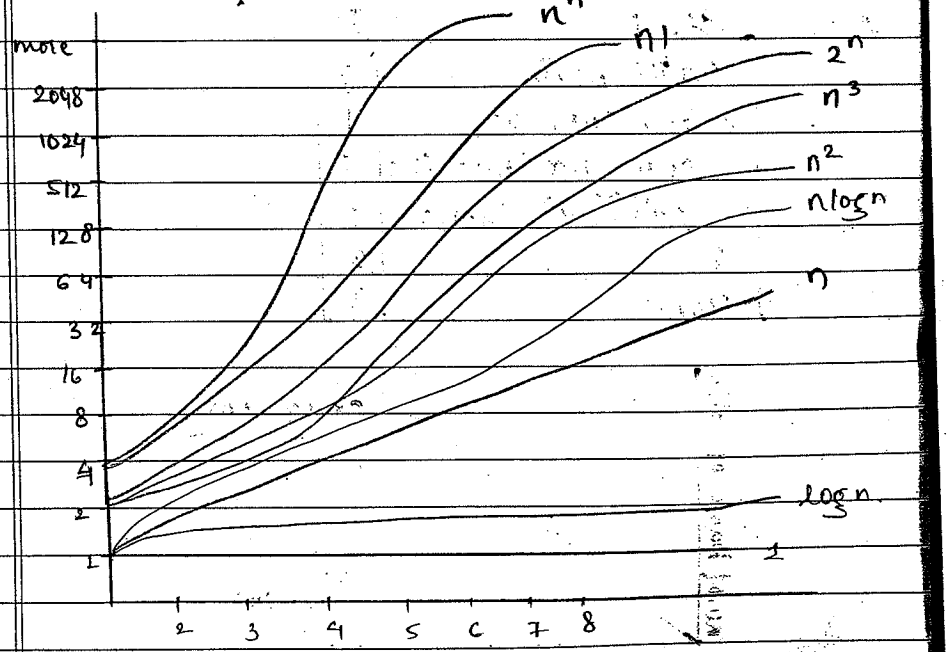
∴ In this case it only makes sense that you be able to recognize and ~~more~~ choose the more efficient algo.

input data.

Algo. are generally ~~recognized~~ analysed. on their time and space requirement.

Some common order of magnitude are.

Order of magnitude	name
1	constant
$\log n$	logarithm
n	linear
$n \log n$	linear logarithm
n^2	Quadratic
n^3	Cubic.
a^n $(a>1)$ or 2^n	Exponent
$n!$	Factorial



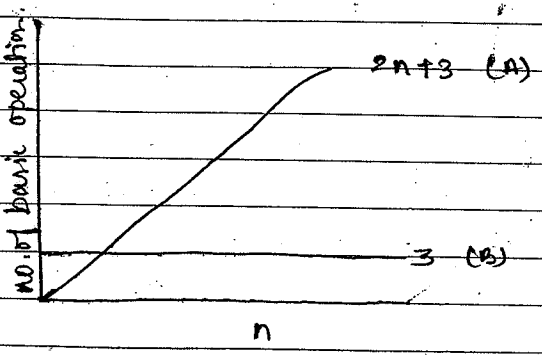
Suppose we have to compare two algo. to computing the sum $1+2+\ldots+n$ for an Integer $n>0$

Algo: A

Statement	f.c.
void main()	0
{ int i;	0
int sum=0;	1
for(i=1; i<=n; i++)	n+1
sum=sum+i;	n.
pf("%d",sum);	1
}	0
Total.	2n+2

Algo: B

Stmt	fc.
void main	0
{ int sum=0;	1
sum = n*(n+1)/2;	1
pf("%d", sum);	1
}	0
Total	3



After analysing the both algo. we found that algo A has Order of mag. of $2n+3$ which is greater than the Order of mag. of B (3).
So, Algo. B is more efficient than A.

Q. Two diff. procedures are written for a given problem, One has a computing time given by 2^n & that for the other is n^3. Specify the range of n for which each would be suitable.

2^n & n^3 for 1 to 9

N	2^n	n^3
0	1	0
1	2	1
2	4	8
3	8	27
4	16	64
5	32	125
6	64	216
7	128	343
8	256	512
9	512	729
10	1024	1000
11	2048	1331
12	4096	1728
13	8192	2197
14	16384	2744
15	32768	3375.

So, from the above analysis, we observed that the computing time given by 2^n is suitable for range 1 to 9 & then n^3 is suitable for above 9.

N	n^2	$2^n/4$
1	1	0.5
2	4	1
3	9	2

Q. Compare the two func^n n^2 & $2^n/4$ for various values of n. Determine when the second becomes larger than the first. n = 1 to 15.

4	16	4 $2^n/4 = 1$ to 8
5	25	8
6	36	16
7	49	32
8	64	64
9	81	128
10	100	256
11	121	512
12	144	1024
13	169	2048
14	196	4096
15	225	8192

From the above, we observed that the second becomes larger than first after N=8

Q. Find which func^n grows faster.
i) $\sqrt{n}$ or $\log n$ n = 1, 2, 5, 10, 15, 20, 30, 40, 50
ii) $n^{\log n}$ or $\log n^n$

Sol) i)

N	$\sqrt{n}$	$\log n$	$n^{\log n}$	$\log n^n$
1	1	0	1	0
2	1.414	0.301	1.232	0.0906
5	2.236	0.698	1.689	0.166
10	3.162	1	10	1
15	3.872	1.176	29.165	11.362
20	4.472	1.301	49.809	193.083
30	5.477	1.477	152.015	120916.1808
40	6.324	1.602	368.634	153868950.7
50	7.071	1.698	770.014	3.23×10^{11}

ii) First func^n i.e $\sqrt{n}$ grows faster.

ii) Second func^n $\log n^n$ grows faster.

Q. List the following func^n from highest to lowest order. If any are of the same order, circle them on your list.

2^n, 2^{n-1}, $\log\log n$, $n^3 + \log n$, $\log n$, $n - n^2 + 5n^3$, n^2, $n!$, n^3, $n \log n$, $(\log n)^2$, $\sqrt{n}$, 6, n, $(3/2)^n$

Answer

$n! > 2^n > n - n^2 + 5n^3 > 2^n 2^{n-1} > n^3 + \log n > n^3 > (3/2)^n > n^2 > n \log n > (\log n)^2 > n > \log n > \sqrt{n} > \log\log n > 6$

Space complexity

Analysis of space complexity of an algo. or program is the amount of memory it needs to run to completion.

Reasons of studying space complexity:
1. If the program is to run on multi

```
{int a=2, b=3, c=4, d;
 d=a+b+c;
}
```
& O(1)

```
{int a[n], i, s=0;
for(i=0; i<n; i++)
 s=s+a[i];
}
```
i.e. n+4 = O(n)

```
{if(n==0)
 return 1;
else
 return n*factorial(n-1)
}
```

```
{int i;
for(i=0; i<n; i++)
 s=s+i;
}
```
i.e 4 = O(1)

user sys., it may be required to specify the amount of memory to be allocated to the program.

We may be interested to know in advance that whether sufficient memo. is available to run the prog.

There may be several possible sol^n with diff. space requirements.

Can be used to estimate the size of the largest problem that a prog can solve.

The space needed by a prog. consists of following components.

→ Instruction space: Space needed to store the executable version of the prog and it is fixed.

→ Data space: Space needed to store all constants, variables. This space is values and has further two components.

a) Space needed by constants and simple variables. This space is fixed.

b) Space needed by fixed sized structural variables, such as arrays and structures.

c) Dynamically allocated space. This space usually varies.

→ Environment stack space: This space is needed to store the info. to resume the suspended (partially completed) func^n. Each time a func^n is invoked the following data is saved on the environment stack:

a) Return address: i.e. from where it has to resume after completion of the called func^n.

b) Values of all lead variables and the values of formal parameters in the func^n being invoked.

The amount of space needed by recursive func^n is called the recursion stack space. For each recursive func^n, this space depends on the space needed by the local variables and the formal parameters. In add^n, this space depends on the max^m depth of the recursion i.e. max^m no. of nested recursive call.

Time Complexity

The time complexity of an algo. or a prog. is the amount of time it needs to run to completion. The exact time will depend on the implementation of the algorithm, prog. language, optimizing the

capabilities of compiler used, the CPU speed, other h/w characteristics/specifications and so on. To measure the time complexity accurately, we have to count all sorts of operations performed in an algo. If we know the time for each one of the primitive ops. performed in a given comp, we can easily compute the time taken by an algo to complete its execution. This time will vary from machine to machine. By analyzing an algo, it is hard to come out with an exact time required.

The no of machine instructions which a prog executes during its running time is called its time complexity. This no depends primarily on the size of prog's I/P. Time taken by a prog is the sum of the compile time and the run time. In time complexity, we consider run time only. The time required by an algo is determined by the no of elementary operations.

The following primitive ops. that are independent from the prog lang are used to calculate the running time:

Assigning a value to a variable
Calling a func^n
Performing an arithmetic op.
Comparing 2 variables.
Indexing into a array of following a ptr reference
Returning from a func^n.

The time complexity also depends on the amount of data inputted to an algo. But we can calculate the order of magnitude for the time req.

eg: add^n of two matrix.

```
void add(int a[][MAX_SIZE], int b[][MAX_SIZE]
         , int c[][MAX_SIZE], int rows, int cols)
{
  int i,j;
  for(i=1;i<=rows;i++)                    rows*(cols+1)
    for(j=1;j<=cols;j++)                  rows*cols.
      c[i][j] = a[i][j] + b[i][j];
}
```

Total $\Rightarrow$ 2row*cols + 2rows + 1

Total time complexity in terms of Big oh Oh (O) = 2rows*cols + 2rows + 1
$= O(rows * cols)$

The general format is -
$f(n)$ = efficiency.

→ Constant
$x = y + z$ $f(n) = 1$

→ Linear loops
for(i=0; i<1000; i++) $f(n) = n$.
application code

for(i=0; i<1000; i+=2) $f(n) = n/2$.
application code.

→ Logarithmic loops
Multiply loop.
for(i=0; i<1000; i=i*2) $f(n) = \log n$.
app. code

Divide loop.
for(i=0; i<1000; i=i/2) $f(n) = \log n$.
app. code

→ Nested loops
~~for(i=1000; i>=0; i=i/2)~~ ~~$f(n) = \log n$.~~
Iterations = Outer loop iterations * Inner loop iterations.

→ Linear logarithmic. 10 log 10.
for(i=0; i<10; i++) $f(n) = n \log n$.
for(j=0; j<10; j*=2)
app. code.

→ Quadratic
for(i=0; i<10; i++) 10*10.
for(j=0; j<10; j++) $f(n) = n^2$
app. code.

→ Dependent Quadratic.
for(i=0; i<10; i++)
for(j=0; j<i; i++) $f(n) = n(n+1)/2 = n^2$
app. code.

Analysis of Algorithm

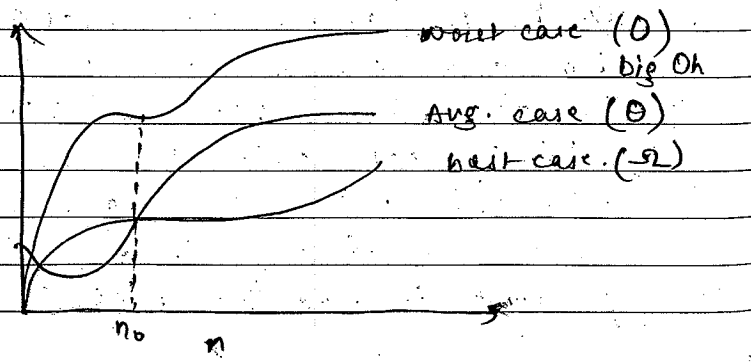
When we analyse an algo. it depends on the I/P data, there are 3 cases:

1. Best case
2. Average case
3. Worst case.

Best case - In this case, the amount of time a prog. might be expected to take on best possible I/P data. OR it is the min. no. of steps that can be executed for the given parameter. Suppose we open a dictionary and luckily we get the meaning of a word which we are looking for. This requires only one step (min. possible) to get the meaning of a word.

Average case - In the average case, the amount of time a prog. might be expected to take on typical (or avg) i/p data. OR.
It is the avg. no. of steps that can be executed for given parameter. If you are searching a word for which neither a min (best case) nor a max (worst case) steps are required is called avg. case. In this case, we definitely get the meaning of the word.

Worst
Best case - In this case, the amount of time a prog. would take on the worst possible i/p configuration. It is the max. no. of steps that can be executed for given parameter. Suppose the case of searching a word in dictionary that word is either not of that word takes max^n possible steps.



Asymptotic notations

The notations we use to describe the asymptotic running time of an algo. are defined in terms of functions whose domains are the set of natural nos. $N = 0, 1, 2 \ldots$ Such notations are convinient for describing the worst case running time fun^n $T(n)$, which is usually defined only on integer i/p sizes. Asymptotic notation is a way of comparing fun^n that ignores constant factors and small I/p sizes.

The main idea of asymptotic analysis is to have a measure of efficiency of algo. that doesn't depend on m/c specific constants, and doesn't require algo. to be implemented & time taken by prog. to be compared. Asymptotic notations are mathematical tools to represent time complexity of algo for asymptotic analysis.

Asymptotic notation : Asymptotic notation is used -

1. To describe the running time of an algo

2. To show the order of growth of func^n
3. To describe algo. efficiency & performance in a meaningful way.
4. Also describe the behaviour of time and space complexity.

There are 5 types

i) O → big oh
ii) Θ → theta
iii) Ω → big omega
iv) o → little oh
v) ω → little omega.

23rd Jan 1. Big Oh notation - The big O notation defines an upper bound of an algo., it bounds a func^n only from above. For eg. consider the case of insertion sort. It takes linear time (Ω) (O(n)), in best case and quadratic time (O(n^2)) in worst case. We can safely say that the time complexity of insertion sort that the time complexity of insertion sort in O(n^2). It provide upper bound asymptotic notation.

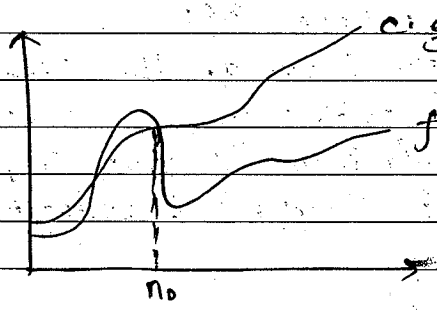
Definition: $O(g(n)) = \{f(n)$: if there exist a +ve constant c & n_0 such that $0 \leq f(n) \leq c \cdot g(n)$, for all $n \geq n_0\}$

$f(n) \leq c \cdot g(n)$ means $f(n)$ is slower than $g(n)$

$$\boxed{f(n) = O(g(n))}$$

OR.

Let $f(n)$ & $g(n)$ are two func^n exist a relation $f(n) = O(g(n))$ if $\lim_{n \to \infty} \frac{(f(n))}{g(n)} = c < \infty$, where c is const.



Property:

1. if f(n) is O(g(n)) and g(n) is O(h(n)) then f(n) is O(h(n))

2. if f1(n) is O(h(n)) & f2(n) is O(h(n)) then f1(n)+f2(n) is O(h(n))

3. if f1(n) is O(h(n)) & f2(n) is O(h(n)) then f1(n)+f2(n) is max (O(h(n)), O(g(n)))

& design the problems in data structure
As we have discussed to develop prog.
of an algo, we should select appropriate
data struct. for that algo.

4. if $f(n)$ is $O(h(n))$ & $f2(n) = O(g(n))$ then $f1(n) \cdot f2(n)$ is $O(g(n)) \cdot h(n)$.
5. If $f(n) = c$ then $f(n)$ is $O(1)$ (Constant)
6. if $f(n) = c * h(n)$ then $f(n)$ is $O(h(n))$.
7. Let $P(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0$, any polynomial func of degree k
then, $p(n) = O(n^k)$, max. polynomial degree.
eg. $27n^3 + 7n^2 + 5n + 10 = O(n^3)$.
8. $n^a = O(n^b)$ only if $a \leq b$ i.e. n^3 is $O(n^3)$, $O(n^4)$, $O(n^5)$ but not $O(n^2)$.
9. $a = O(b)$ where $a \leq b$.
10. All algo. grow at the same rate i.e. while computing the O notation, base of the logarithm is not imp.
$O(\log_a n)$ is $O(\log_b n)$ & $O(\log_b n)$ is $O(\log_a n)$
$a, b > 1$.
11. $\log n$ is $O(n)$
12. $\log^k n$ is $O(n)$.

Advantage

1. It is possible to compare of two algo. with running times.
2. Constants can be ignored. - Units are not imp. $O(7n^2) = O(n^2)$.
3. Lower order terms are ignored. $O(n^3 + 7n^2 + 3) = O(n^3)$
4. Running times of algo. A & B
$T_A(N) = 1000N = O(N)$, $T_B(N) = N^2 = O(N^2)$.
A is asymptotically faster than B.

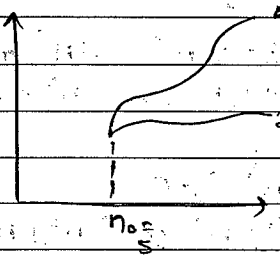
Limitations

1. Big Oh notation has following basic limitation. It contains no effort to improve the prog (common methodology. Big Oh notation does not discuss the way & means to improve the efficiency of the prog, but it helps to analyze & calc. the efficiency (by finding time complexity) of the program.
2. It does not exhibit the potential of the constant. For eg, one algo. is taking $100n^2$ time to execute & other n^3 time. The first algo. is $O(n^2)$, which implies that it will take less time than the other algo. which is $O(n^3)$. However in actual execution the 2nd algo. ✗

Q. $f(n) = 3n + 5$.

Sol. We know that $f(n) \leq c \cdot g(n)$.
$\therefore n \geq 5$

$3n + 5 \leq 3n + 4n$
$\Rightarrow 3n + 5 \leq 4n$
So $\because c = 4$, $g(n) = n$.
$n_0 = 5$



Q. $27n^2 + 16n + 25$.

Sol. We know that $f(n) \leq c \cdot g(n)$
$\therefore n \geq 25$
~~$27n^2 + 16n + n \geq 27n^2 + 16n + 25$~~ $27n^2 + 16n + 25 \leq 27n^2 + 16n + n$
now $n^2 \geq n$

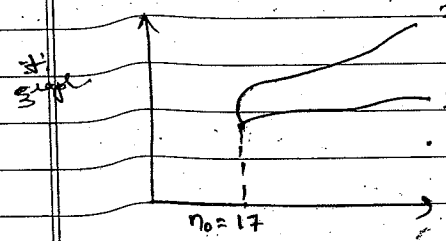
$27n^2 + 16n + n^2 \geq 27n^2$

$27n^2 + 17n \leq 27n^2 + n^2$

$\Rightarrow 27n^2 + 17n \leq 28n^2$

$\therefore C = 28$ & $g(n) = n^2$

& $n_0 = 17$



Q. $5n^3 + n^2 + 3n + 2$

Soln We know that $f(n) \leq C \cdot g(n)$.

So $\therefore n \geq 2$.

$\Rightarrow 5n^3 + n^2 + 3n + n \geq 5n^3 + n^2 + 3n + 2$.

$\Rightarrow 5n^3 + n^2 + 4n \geq 5n^3 + n^2 + 3n + 2$.

now $n^2 \geq n$

$\Rightarrow 5n^3 + n^2 + n^2 \geq 5n^3 + n^2 + 3n$.

$\Rightarrow 5n^3 + 2n^2 \geq 5n^3 + n^2 + 3n$.

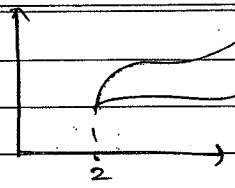
now $n^3 \geq n^2$.

$\Rightarrow 5n^3 + n^3 \geq 5n^3 + 2n^2$

$\Rightarrow 6n^3 \geq 5n^3 + 2n^2$

$\therefore C = 6$ $g(n) = n^3$

$n_0 = 2$.



Q. $f(n) = 3 * 2^n + 4n^2 + 5n + 3$.

Soln

$\Rightarrow 3 * 2^n + 4n^2 +$

$n \geq 3$

$\Rightarrow 3 \cdot 2^n + 4n^2 + 5n + n \geq 3 \cdot 2^n + 4n^2 + 5n + 3$.

$\Rightarrow 3 \cdot 2^n + 4n^2 + 6n \geq 3 \cdot 2^n + 4n^2 + 5n + 3$.

$n^2 \geq n$.

$\Rightarrow 3 \cdot 2^n + 4n^2 + n^2 \geq 3 \cdot 2^n + 4n^2 + 5n$.

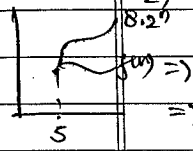
$\Rightarrow 3 \cdot 2^n + 5n^2 \geq 3 \cdot 2^n + 4n^2 + 5n$.

now $2^n \geq n^2$.

$3 \cdot 2^n + 5 \cdot 2^n \geq 3 \cdot 2^n + 5n^2$.

$8 \cdot 2^n \geq 3 \cdot 2^n + 5n^2$.

$\therefore C = 8$, $g(n) = 2^n$, $n_0 = 5$.



↳ Incorrect Bound

Big Oh

According to ratio theorem we know that

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = C < \infty.$$

We know that $f(n) = 7n + 5$

$g(n) = 1$.

$$\lim_{n \to \infty} \frac{7n+5}{1} = \infty \not< \infty.$$

So, relation is invalid.

P.T.

Q. $f(n) = 3n^3 + 4n \neq O(n^2)$.

Sol^n. We from Big Oh ratio theorem

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c < \infty.$$

$$\Rightarrow \lim_{n\to\infty} \frac{3n^3+4n}{n^2}$$

$$\Rightarrow \lim_{n\to\infty} \frac{n(3n^2+4)}{n^2}$$

$$\Rightarrow \lim_{n\to\infty} \frac{3n^2+4}{n}$$

$$\Rightarrow \lim_{n\to\infty} \frac{3n+4}{n}$$

$$\infty \not< \infty.$$

↳ Loose Bound.

eg $2n+3 = O(n^2)$ is it valid or not?

Sol^n. from ratio theorem

$$\lim_{n\to\infty} \frac{2n+3}{n^2}$$

$$\lim_{n\to\infty} \frac{2}{n} + \frac{3}{n^2}$$

$$\frac{2}{\infty} + \frac{3}{\infty}$$

$$= 0 < \infty.$$

This relation is valid but the above bound is not the tighter bound, as we have a smaller func^n (in this case linear). That also satisfies the Big Oh rel^n.

2. Big omega notation (Ω) - It provide lower bound asymptotic notation. Just as Big O notation provides an asymptotic upper bound on a func^n, Ω notation provides an asymptotic lower bound. Ω notation can be useful when we have lower bound on time complexity of an algo. As discussed in the previous post, the best case performance of an algo is generally not useful, the Ω notation is the least used notation.

For a given func^n $g(n)$, we denote by $\Omega(g(n))$, the set of func^n.

Definition : $\Omega(g(n)) = \{ f(n) :$ if there exist a +ve constant c & n$_0$ of such that, $0 \leq c \cdot g(n) \leq f(n)$ for all $n \geq n_0 \}$. means $f(n)$ is faster than $g(n)$.

OR

Let $f(n)$ & $g(n)$ are two func^n such that $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ exists the function

$f(n) \in \Omega(g(n))$.

If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c > 0$, where c is constant.

Property:

1. Let $p(n) = a_n^* n^n + a_{n-1}^* n^{n-1} + a_{n-2}^* n^{n-2} + \dots + a_1^* n^1 + a_0$ be any polynomial func^n. of degree n then, $p(n) = \Omega(n^n)$.

2. $a = \Omega(b)$, $(b \le a)$.

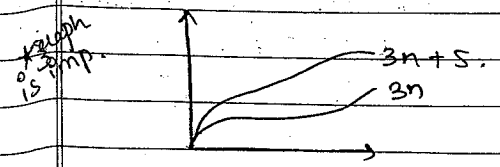
By definition of Ω: $f(n) = \Omega(g(n))$ means $c \cdot g(n) \le f(n)$

Q. $f(n) = 3n+5$ find Ω.

Sol^n $3n \le 3n+5$ for all n.

where $c = 3$ & $g(n) = n$.

$\therefore f(n) = \Omega(n)$.

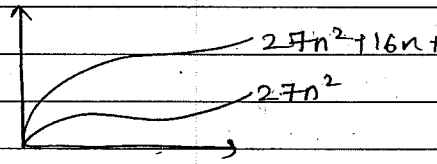


Q. $27n^2+16n+25 = f(n)$ find Ω ?

Sol^n $27n^2 \le 27n^2+16n+25$. for all n.

where $c = 27$ & $g(n) = n^2$

thus $f(n) = \Omega(n^2)$

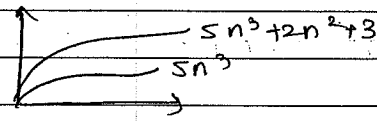


Q. $f(n) = 5n^3+2n^2+3n+2$

Sol^n $5n^3 \le 5n^3+2n^2+3n+2$ for all n.

where $c = 5$ & $g(n) = n^3$

thus $f(n) = \Omega(n^3)$



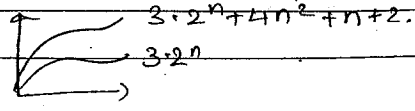
Q. $3*2^n+4n^2+n+2$.

Sol^n $3*2^n \le 3*2^n+4n^2+n+2$ for all n.

where $c = 3$ & $g(n) = 2^n$.

thus

$f(n) = \Omega(2^n)$.



Incorrect bound

eg. $f(n) = 7n+5 = \Omega(n^2)$. Is it valid?

Sol^n Acc. to big Ω ratio theorem

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = C > 0$$

where $f(n) = 7n+5$

$g(n) = n^2$

$$\lim_{n\to\infty} \frac{7n+5}{n^2}$$

$$\lim_{n\to\infty} \frac{7}{n} + \frac{5}{n^2} = 0 \not> 0.$$

∴ It is invalid relation.

Q. $3n^3 + 5n^2 - 2n + 3 = \Omega(n^4)$. Is it valid?

Sol^n Acc to ratio theorem

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = C > 0.$$

$$\lim_{n\to\infty} \frac{3n^3+5n^2-2n+3}{n^4}$$

$$\frac{3}{n} + \frac{5}{n^2} - \frac{2}{n^3} + \frac{3}{n^4} = 0 \not> 0.$$

∴ Relation is invalid.

Loose bound

eg. $7n+5 = \Omega(1)$. Is it valid?

Sol^n $$\lim_{n\to\infty} \frac{f(n)}{g(n)}$$

$$\Rightarrow \lim_{n\to\infty} \frac{7n+5}{1}$$

$$\Rightarrow \infty > 0.$$

∴ The relation is valid & it is loose bound. (The given bound is not the tight lower bound). as we have a large func^n n (linear) that satisfies the big Ω cond^n.

Q. $f(n) = 5n^3 + 3n^2 + 2 = \Omega(n^2)$.

Sol^n $$\lim_{n\to\infty} \frac{5n^3+3n^2+2}{n^2}$$

$$\lim_{n\to\infty} 5n + 3 + \frac{2}{n^2}$$

$$\infty > 0.$$

∴ The relation is valid. & is loose bound.

3. Theta notation (Θ) - The theta notation bounds a func^n from above & below, so it defines exact asymptotic behaviour. A simple way to get theta notation of an exp. is to drop lower order terms & ignore leading constants. For eg. consider the following exp. $3n^3+6n^2+6000 = \Theta(n^3)$. Dropping lower order terms is always fine because there will always be a n_0 after which $\Theta(n^3)$ beats $\Theta(n^2)$ irrespective of the constants involved.

Definition: For a given func^n $g(n)$, we denote $\Theta(g(n))$ is following set of func^{n}s. Theta of $\Theta(g(n)) = \{f(n)$: if there exist a +ve constant c & n_0 such that $0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ where $(n \geq n_0)\}$.

The above definition means, if $f(n)$ is $\Omega(g(n))$, then the value of $f(n)$ is always b/w $c_1 \cdot g(n)$ & $c_2 \cdot g(n)$ for large values of n $(n \geq n_0)$.
The definition of Θ also requires that $f(n)$ must be non-negative for all values of $n > n_0$.

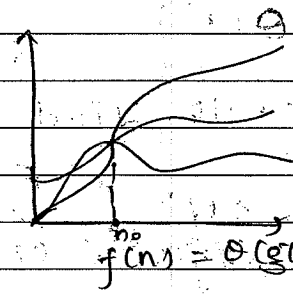
OR.
Let $f(n)$ & $g(n)$ are two func^n such that $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ exist then the func^n $f(n) \in \Theta(g(n))$.
If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ where $0 < c < \infty$.

Property:

1. Let $p(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots a_1 n + a_0$ be any polynomial func^n then $p(n) = \Theta(n^k)$

2. $a = \Theta(b) \Rightarrow a = b$.

Q.



Q. find Θ notation for n^2+n+1.

Sol^n Acc. to def^n $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$.

Ω: $c_1 g(n) \leq f(n)$.
$n^2 \leq n^2+n+1$
$c_1 = 1$ & $g(n) = n^2$.

O: $f(n) \leq c_2 \cdot g(n)$.

$n^2+n+1 \leq n^2+n+n$, $(n \geq 1)$

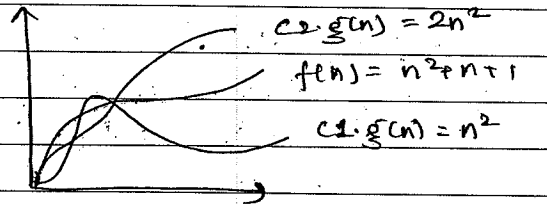
$\leq n^2+2n$

$\leq n^2+n^2$ $(n^2 \geq n)$

$\leq 2n^2$.

where $n_0 = 2$. , $c_2 = 2$.

$\therefore$ $n^2 \leq n^2+n+1 \leq 2n^2$



$\therefore$ $c_1 = 1$, $c_2 = 2$, $g(n) = n^2$

$\therefore$ $f(n) = \Theta(n^2)$

Q. $f(n) = 21n^2+15n+10$ find O?

Sol^n Acc. to Def^n

$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$

Ω : $c_1 \cdot g(n) \leq f(n)$.

$21n^2 \leq 21n^2+15n+10$.

Here $c_1 = 21$ $g(n) = n^2$

O : $f(n) \leq c_2 \cdot g(n)$.

$n \geq 10$.

$10 \leq n$

$21n^2+15n+10 \leq 21n^2+16n$

$n \leq n^2$

$\Rightarrow 21n^2+15n+10 \leq 21n^2+n^2$

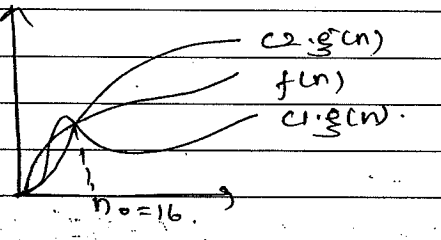
$21n^2+15n+10 \leq 22n^2$

$\therefore$ $c_2 = 22$ & $g(n) = n^2$.

$n_0 = 16$.

$f(n) = \Theta(g(n))$

$f(n) = \Theta(n^2)$.



↳ Incorrect bound

eg. Q. $f(n) = 7n+5 = \Theta(1)$? Check for validity

Sol^n $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$.

Acc to Θ ratio theorem

$\lim_{n \to \infty} \frac{f(n)}{g(n)} = c$. where $0 < c < \infty$.

$\lim_{n \to \infty} \frac{7n+5}{1}$

$\lim_{n \to \infty} \infty = c$.

$\therefore$ $c > 0$ but $c \neq \infty$.

Thus it satisfies the Ω notation but does not satisfies the O notation

Q $27n^2+16n+25 = \Theta(n^3)$. Check?

4. Little o -

i) Little o of $o(g(n)) = \{f(n)$: If there exist a +ve constant c & no such that $f(n) < c \cdot g(n)$, $n \geq n_0$.
OR.

ii) Let $f(n)$ & $g(n)$ be any two fun^n, $f(n)$ belongs to $o(g(n))$ If

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0. \text{ (zero).}$$

conclusion: $a = o(b) \rightarrow a < b$

★ Loose bound of big Oh is the little o..

$$\boxed{f(n) < o(g(n))}$$

Q $3n+5 = o(n^2)$. ? valid or not?

Sol^n Acc to def^n

$$\lim_{n\to\infty} \frac{3n+5}{n^2}$$

$$0 = 0$$

valid

Q. $3n+5 = o(n)$.

Sol^n $$\lim_{n\to\infty} \frac{3n+5}{n}$$

$$\frac{3+5}{n}$$

$$3 \neq 0$$

Not valid.

Q. $4n^3+2n+3 = o(n^3)$.

Sol^n $$\lim_{n\to\infty} \frac{4n^3+2n+3}{n^3}$$

$$4 \neq 0.$$

Not valid.

5. Little ω -

i) $\omega(g(n)) = \{f(n)$: If there exist a +ve constant C & no. such that $C \cdot g(n) < f(n)$, for all $n\}$.

ii) Let $f(n)$ & $g(n)$ be any two fun^n such that $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$.

or $\lim_{n\to\infty} g(n)/f(n) = 0$.

Q. conclusion : $a = \omega(b) \rightarrow a > b$.

Q. Is $2n+1 = \omega(1)$?

Soln. $\lim_{n\to\infty} \frac{2n+1}{1}$

$\infty = \infty$

Which is true.

Q. $2n+1 = \omega(n)$.

Soln $\lim_{n\to\infty} \frac{2n+1}{n}$.

$2 \neq \infty$.

Thus the relation is invalid.

Q. $= 2n+5 = \omega(1)$

Q. $n^2+n+1 = \omega(n^2)$

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

Q. $2^{2n+1} = \omega(2^n)$

valid.

Q. $2n+1 = \omega(n)$

Q. $n^2+n+1 = \omega(n^2)$

Q. $5n^3+10n+2 = \omega(n^5)$

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

Invalid.

Summary

1. $f(n) = O(g(n)) \rightarrow f(n) \leq g(n)$
2. $f(n) = \Omega(g(n)) \rightarrow f(n) \geq g(n)$
3. $f(n) = \Theta(g(n)) \rightarrow f(n) = g(n)$.
4. $f(n) = o(g(n)) \rightarrow f(n) < g(n)$
5. $f(n) = \omega(g(n)) \rightarrow f(n) > g(n)$.

Time Space Trade-off

It refers to a choice b/w algorithmic soln of data processing problem that allows one to decrease the running time of an algo soln by increasing the space to store the data & vice versa.

The best algo. to solve a given prob. is one that requires less space in memory & takes less time to complete its execution. But in practice it is not always possible to achieve both these objectives. As we know there may be more than one approach to solve a particular problem. One approach may take more space but takes less time to complete its execution while the other approach may take less space but takes more time to complete its execution. We may have to sacrifice one at the cost of the other. If space is our constraint, then we have to choose a prog. that requires less space at the cost of more execution time. On the other hand if time is our constraint then we have to choose a prog. that takes less time to complete

its execution at the cost of more space.

Algorithms like Mergesort are exceedingly fast, but require lots of space to do the operation. On the other hand the prog. like Bubble sort is exceedingly ~~down~~ slow, but takes less space. (min. space)

Abstract Data Type (ADT)

An ADT is a data type solely defined in terms of a type and a set of operations on that type. Each operation is defined in terms of its I/p & O/p. without specifying how the data type is implemented. It does not specify how data will be organized in memo. & what algo. will be used for implementing the operations. It is called "abstract" coz it gives an implementation independent view. The process of providing only the essentials and hiding the details is known as abstraction. We can think of ADT as a black box which hides the inner struct. & design of the data type

The array as an ADT -

→ An array object is a set of pairs, <index, value>, such that each index is unique & each index that is defined has a value associated with it (a mapping from indices to values).

→ Operations include setting and retrieving a value for a given index.

An array is a finite sequence of storage cells, for which the following operations are defined -

create (A, N) creates an array A with storage for N items;
A[i] = item stores item in the i^{th} pos' in the array A; &
A[i] returns the value of the item stored in the i^{th} pos' in the array A.

Operations in array ADT

→ create (A, N) creates an array A with storage for N items;
→ bool isEmpty(); - Returns true if array is empty.
→ bool isFull(); - Returns true if array is full.

→ int length(); - Returns the length of the array.

→ void add(int item, int pos) - adds the element at the given pos".

→ void del(int item); - Deletes the given item from the array.

Array

An array is a finite, ordered & collection of homogeneous (similar) data elements. Array is finite coz it contains only limited no. of elements and ordered, as all elements are stored one by one in continuous location of comp. memory. All elements of an array of same type only & hence if it is termed as collection of homogeneous element.

The operations performed →

1. Traversing
2. Search
3. Insertion
4. Deletion
5. Sorting.
6. Merging.

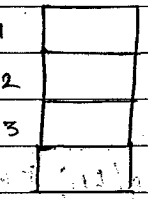
29th Jan

Linear Array → It is a list of finite no. of homogeneous homogeneous data elements i.e. the elements of the array are stored in a continuous memory loc" & all the data elements are of same type.

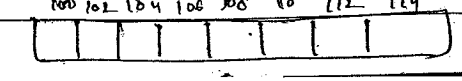
Declaration int a[n];
n = length of array.
length or no. of data elements of the array can be obtained by following formula.

Length = UB - LB + 1
UB = upper bound
LB = Lower bound

Representation of linear array - Let LA be a linear array in the memory of the comp., memory of comp. is simply a sequence of addres location -



where LOC[LA[K]] denotes address of element LA[K] of the array.



Elements of array are store in successive memory cell accordingly. The comp. doesn't need to keep track of every address of every element of LA. But need only address of first element of array. Denoted by Base(LA) called base address.

To calculate address (loc) of any element of LA.

$$\boxed{LOC[LA[K]] = Base(LA) + w(K - LB)}$$

Where w = size of data type in bytes
K = subscript.
LB = lower bound.

Q. Base address of 1st element of 1D array is 2000 & each element of array occupied 4 byte in the memo. then find address of 5th element of 1D array.

Sol: Assume LB = 1

Acc. to formula -

$LOC[LA[K]] = Base(LA) + w(K - LB)$

$= 2000 + 4(5-1)$

$= 2016$

Q. Consider an array x[8] of character and let its base address be 1002. What is the address of x[5]?

Sol: Assume LB = 1

Acc. to formula.

$LOC[x[K]] = base(x) + w(K - LB)$

$= 1002 + 1(5+1)$

$= 1002 + 4$

$= 1006$

Q. Consider the linear array A(5:50), B(-5,10) & C(18). array.

a) Find the no. of elements in each

b) Suppose base(A) = 300 & w = 4 bytes for A. find the address of A[15], A[40] & A[55].

Sol: a) Given A(5:50)

A → LB = 5, UB = 50

B → LB = -5, UB = 10

C → LB = 1, UB = 18.

Now no. of elements in

A = UB - LB + 1 = 50 - 5 + 1 = 46

B = UB - LB + 1 = 10 - (-5) + 1 = 16

C = UB - LB + 1 = 18 - 1 + 1 = 18.

b) Now
base(A) = 300
w = 4 bytes

LOC[A[15]] = 300 + 4(15 - 5)
= 300 + 40
= 340.

LOC[A[40]] = 300 + 4(40 - 5)
= 300 + 140
= 440.

A[55] is not an element of A, ∵ 55 exceeds UB = 50

Operations on linear array -

1 Traversing -

Algorithm: (traversing a linear array)
Here LA is a linear array with lower bound LB and upper bound UB. This algorithm traverses LA applying an operation PROCESS to each element of LA.

1. [Initialize counter] set K := LB.
2. Repeat steps 3 and 4 while K <= UB.
3. [Visit element] apply PROCESS to LA[K].
4. [Increase counter] set K := K + 1.
[End of step 2 loop].
5. Exit.

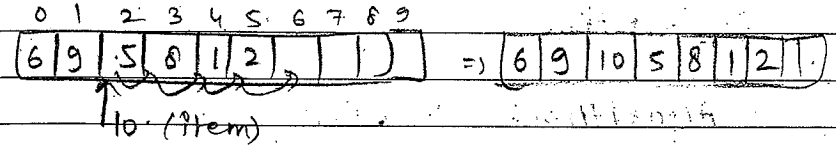
2 Insertion

Algorithm: (Inserting into a linear array)
INSERT (LA, N, K, ITEM).
Here LA is a linear array with N element & K is positive integer such that K <= N. This algorithm inserts an element ITEM into the K^th position in LA.

1. [Initialize counter] set J := N.
2. Repeat steps 3 and 4 while J >= K.
3. [Move J^th element downward] set LA[J+1] := LA[J].
4. [Decrease counter] set J := J - 1.
[End of step 2 loop]
5. [Insert element] set LA[K] := ITEM.
6. [Reset N] set N := N + 1.
7. Exit.

Code:

```
insertion (int a[], int loc, int item)
{ int i;
  for (i = n-1; i >= loc-1; i--)
      a[i+1] = a[i];
  a[loc-1] = item;
  n = n+1;
}
```



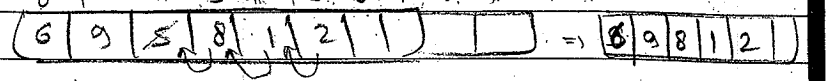
3. Deletion

Algorithm: (Deleting from a linear array)
DELETE (LA, N, K, ITEM)
Here LA is a linear array with N elements & K is a positive integer such that K<=N. This algorithm deletes the K^{th} element from LA.

1. Set ITEM := LA[K]. (keeping the item for record)
2. Repeat for J = K to N-1:
   [Move J+1 element upward]
   Set LA[J] := LA[J+1]
   [End of step2 loop]
3. [Reset N] set N := N-1
4. Exit.

Code:

```
deletion (int a[], int loc)
{ int i, item;
  item = a[loc-1]
  for (i = loc-1; i <= n-1; i++)
      a[i] = a[i+1];
  n = n-1;   pf("Deleted item=%d", item);
}
```



delete loc = 3
i.e a[2]

4. Linear Search

Algorithm: (Linear Search)
LINEAR (DATA, N, ITEM, LOC)
Here DATA is a linear array with N elements, and ITEM is a given item of information. This algorithm finds the location LOC of item in DATA, or sets LOC := 0 if search is unsuccessful.

1. [Insert ITEM at the end of DATA]
   Set DATA[N+1] := ITEM.
2. [Initialize counter] Set LOC := 1.
3. [Search for ITEM.]
   Repeat while DATA[LOC] ≠ ITEM:
   Set LOC := LOC+1

[end of loop]

4. [Successful ?] If LOC = N+1, then:
   Set LOC := 0.

5. Exit.

Ans: Analysis (Complexity) of linear search

Complexity of linear search algo. is measured by the no. f(n) of comparisons required to find item in data. Where data contains N elements.

Best case - This is the case when the key present in the first element, that is in this case, only one comparison will take place. Thus, T(n) the no. of comparisons is

$$T(n) = 1.$$

Avg. case - The key is present at any location in the array. The running time of the avg. case uses the probabilistic notion of expectation. Suppose P_K is the probability that item appears in data[K] and suppose Q is the probability that item does not occur in DATA. Then

$$[P_1 + P_2 + \cdots P_n + Q] = 1.$$

∴ algo uses k comparisons when item appears in data[K]. The avg. no. of comparisons is given by -

$$T(n) = 1 \cdot P_1 + 2 \cdot P_2 + \cdots nP_n + (n+1)Q.$$

Suppose Q is very small & item appears with equal probability in each element then $Q = 0$. & each $P_i = 1/n$ then.

$$T(n) = 1 \cdot 1/n + 2 \cdot 1/n + \cdots n \cdot 1/n + (n+1) \cdot 0$$

$$= 1/n \,[1 + 2 + \cdots + n]$$

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

$$= \frac{n+1}{2}.$$

Worst case: occurs when one must search the key through the entire array DATA., when item does not occur in DATA or occur at last pos. In these case $T(n) = n$ or $n+1$

Case	No. of key comparison	Asymptotic Complexity
1. Best	$T(n) = 1$	$T(n) = O(1)$
2. Average	$T(n) = n+1/2$	$T(n) = \Theta(n)$
3. Worst	$T(n) = n$	$T(n) = O(n)$

Code:

```
#define MAX 10
main()
{ int a[MAX], item, flag=0, i, n;
  clscr();
  pf("Enter the limit b/w 0 & 10");
  scanf("%d", &n);
  pf("Enter array element");
  for(i=0; i<n; i++)
    sf("%d", &a[i]);
  pf("Enter the item to be searched");
    sf("%d", &item);
  for(i=0; i<n; i++)
  { if(a[i] == item)
      { flag=1;
        break;
      }
  }
  if (flag==1)
    pf("item found at %d position", i+1);
  else
    pf("item not found");
  getch();
  return 0;
}

/* for counting same item */
  for(i=0 ... )
  { if(a[i] == item)
      { flag count++;
      }
```

× Count++;
If one item comes more than one time for counting how many no. of times the item

```
  }
  if(count>0)
    pf("item found %d times", count);

}

/* for printing the location of same item pos */
{     int b[MAX], j=0;
  for(i=0; i<n; i++)
  { if(a[i]==item)
    { //flag=1
      count++;
      b[j]=i+1;
      j++;
    }
  }
  if(count>0)
  { for(i=0; i<count; i++)
      pf("\n item found at %d location", b[i]);
  }

}
```

5 Binary Search

Algorithm: (Binary Search)
BINARY (DATA, LB, UB, ITEM, LOC)
Here DATA is a sorted array with lower bounds LB & upper bounds UB, & ITEM is a given item of info. The variables BEG, END & MID denote, respectively, the beginning, end an middle locations of a segment of elements of DATA. This algo. finds the loc^n LOC of ITEM in DATA, or sets LOC := NULL if search is unsuccessful.

1. [Initialize segment variables]
   Set BEG := LB, END := UB, &
   MID = INT ((BEG + END)/2).

2. Repeat steps 3 and 4 while BEG ≤ END and DATA [MID] ≠ ITEM.

3. If ITEM < DATA [MID], then:
   Set END := MID - 1.
   Else:
   Set BEG := MID + 1
   [End of if structure]

4. Set MID := INT ((BEG + END)/2)

[End of step 2 loop]

5. if DATA [MID] = ITEM, then:
   Set LOC := MID.
   Else:
   Set LOC := NULL.
   [End of if structure]

6. Exit

Example
Let DATA be the following sorted 13 elements array
DATA [11, 22, 30, 33, 40, 44, 55, 60, 66, 77, 80, 88, 99].
SUPPOSE we have to search
ITEM = 40.

1. Initially BEG = 1 and END = 13 hence
   MID = Int [(BEG + END)/2]
   = Int [(1+13)/2] = 7.
   And so DATA [MID] = 55.

2. ∵ 40 < 55 END has its value changed by END = MID - 1 = 6.
   hence MID = Int [(1+6)/2] = 3.
   And so DATA [MID] = 30.

3. ∵ 40 > 30 BEG has its value changed by BEG = MID + 1 = 4.

hence MID = int [(4+6)/2] = 5
And so DATA[MID] = 40.
We have found ITEM in location MID = 5.

1. (11), 22, 30, 33, 44, [55], 60, 66, 77, 80, 88, (99)

2. (11), 22, [30], 33, 40, (44), 55, 60, 66, 77, 80, 88, 99

3. 11, 22, 30, (33), [40], (44), 55, 60, 66, 77, 80, 88, 99

[SUCCESSFUL]

Analysis of binary search

The complexity is measured by number T(n) of comparison to locate ITEM in DATA contains n elements. Observe that each comparison reduces the sample size in half.

Let us examine the operations for a specific case, where the no. of elements in the array n is 64.

→ When n = 64 Binary Search is called to reduce size to n = 32.
→ When n = 32 Binary Search is called to reduce size to n = 16.
→ When n = 16 Binary Search is called to reduce size to n = 8
→ When n = 8 —— n —— n = 4
→ When n = 4 —— n —— n = 2
→ When n = 2 —— n —— n = 1.

Thus we see that Binary Search part is called 6 times (6 elements of the array were examined)
for n = 64
Note that $64 = 2^6$
hence we required at most T(n) comparisons to locate ITEM where,

$2^{T(n)} < n$ or equivalent $T(n) = \log_2 n + 1$

That is running time for worst case is approximately equal to $\log_2 n$.

31st Jan

Code:
```
#define MAX 20
main()
{ int a[MAX], beg, end, mid, n, item;
  clrscr();
  pf("Enter the limit ");
  sf("%d", &n);                    in sorted order
  pf("Enter the array element:");
  for(i=0; i<n; i++)
    sf("%d", &a[i]);
  beg = 0;
  end = n-1;
```

pf("Enter item:");
sf("%d", &item);

```
mid = (beg + end)/2;
while(beg <= end && a[mid] != item)
{
  if(item < a[mid])
     end = mid - 1;
  else
     beg = mid + 1;
  mid = (beg + end)/2;
}
if(a[mid] == item).
   pf("Item found at %d location", mid+1);
else
   pf("Not found!!");
getch();
return(0);
}
```

Ques. of Exam

*Q. Search the item '80' from the following list using binary search -
11, 22, 30, 35, 42, 45, 53, 63, 65, 78, 80, 90, 95.

Sol Solve as same as previous e.g. If algo. will asked then write.

Q. Write binary search algo. and trace the search element '91' in following list -
13, 30, 62, 73, 81, 88, 91.
and also write what are the limitation of binary search.

Sol Let DATA be the following sorted 13 elements array
DATA[11, 22, 30, 35, 42, 45, 53, 63, 65, 78, 80, 90, 95]
We have to search - 80.

1. Initially BEG = 1 and END = 13 hence.
MID = ~~BEG~~ int[(BEG + END)/2]
= int[(1+13)/2] = 7
And so DATA[MID] = 53

2. ∵ 80 > 53 BEG has its value changed by BEG = MID + 1 = 7 + 1 = 8.
hence MID = int[(BEG + END)/2]
= int[(8+13)/2] = int[10.5] = 10.
And so DATA[MID] = 78.

3. ∵ 80 > 78 BEG has its value changed by BEG = MID + 1 = 10 + 1 = 11.
hence MID = int[(BEG + END)/2]
= int[(11 + 13)/2] = 12.
And so DATA[MID] = 90.

4. ∵ 80 < 90 END has its value changed by END = MID - 1 = 12 - 1 = 11.
hence MID = int[(BEG + END)/2]
= int[(11 + 11)/2] = 11.
And so DATA[MID] = 80
We have found the item at loc^n MID = 11.

Sol^n 2. Let DATA be the following sorted 7 elements array.

DATA [13, 30, 62, 73, 81, 88, 91]

We have to search 91

1. Initially BEG = 1 & END = 7 hence

MID = int [(BEG + END)/2]

= int [(1+7)/2] = 4

And so DATA[MID] = 73

2. ∵ 91 > 73 BEG has its value changed by BEG = MID + 1 = 4 + 1 = 5.

Hence MID = int [(BEG + END)/2]

= int [(5+7)/2] = 6.

And so DATA[MID] = 88.

3. ∵ 91 > 88 BEG has its value changed by BEG = MID + 1 = 6 + 1 = 7.

hence MID = int [(BEG + END)/2]

= int [(7+7)/2] = 7.

And so DATA[MID] = 91.

We have found the item at loc^n MID = 91.

Limitation of binary search

1. The complexity of binary search is $O(\log_2 n)$. The complexity is same irrespective of the posⁿ of the element, even if it is not present in the array.
2. The algo. assumes that one has direct access to middle element in the list or a sub-list. This means that the list must be stored in some type of array. Unfortunately inserting an element in an array requires element to be moved down the list & deleting an element from an array requires the element to be moved up in the array.
3. The list must be sorted.

Time complexity of an array

	Operation	Time complexity (array)
1.	Read (from any where)	O(1)
2.	Add / Remove at end	O(1)
3.	Add / Remove in the interior	O(n)

4.	Resize	O(n)
5.	Find by position	O(1)
6.	Find by target (value)	O(n)

Multidimensional Array

Many prog. lang. allows multi-D arrays. Here is the general form of a multi-D array declaration:

Type name Type name[size][size1]...[sizeN].

For eg. the following declaration creates a 3D 5.10.4 integer array.

int threedm[5][10][4];

2-D Array

The simplest form of the multi-D array is the 2-D array. A 2-D array is, in essence, a list of 1-D array. To declare a 2-D integer array of size x, y, you would write as -

type name[x][y];

Where type can be any valid C data type and name will be a valid C identifier. A 2D array can be think as a table which will have x no. of rows & y no. of columns. A 2-D array a, which contain 3 rows & 4 columns -

int a[3][4].

	C0	C1	C2	C3
R0	a[0][0]	a[0][1]	a[0][2]	...
R1	a[1][0]	a[1][1]	a[1][2]	...
R2	a[2][0]	a[2][1]	a[2][2]	...

Thus every element in array is identified by an element name of form a[i][j], where a is the name of the array, and i & j are the subscript that uniquely identify each element in a.

→ Representation of 2-D array

Let 'A' be a 2-D mxn array. The

array will be represented in memo. by a block of m×n sequential m/p loc?. In prog. lang. this will be represented by 2 methods –

i) Row major order
ii) Column major order.

i) Row major order

In this elements of matrix are stored in a row by row basis i.e. all the elements in 1st row then in 2nd & so on.

Eg. int a[3][4]

a[0][0] →	01 →	02 →	03]
↳ 10 →	11 →	12 →	12]
↳ 20 →	21 →	22 →	23

Logical representation.

Storage representation of row major order

a[0][0]		← base address.
01		
02		
03		
10		
:		
.		
23		

Address of element in row major order –

Address of element in 2-D array can be calculated by using general formula when elements are arranged in row major order –

Formula – $X[i][j] = \text{Base add.} + \text{size} \times (\text{no. of columns} \times (i - L_1) + (j - L_2))$

$X[i][j] = \text{b.a.} + \text{size} \times [\text{no. of cols} \times (i - L_1) + (j - L_2)]$

b.a = base address of first element.
size (w) = required for storing each elements of array
L_1 = lower bound of rows
L_2 = lower bound of cols.
U_1 = upper bound of rows.
U_2 = upper bound of cols.

Q. Calc. the address of X[6,2] in 2D array x[4...7, -1...3] stored in row major order in main m/p. Assume b.a = 100. and each element requires 2 word of storage.

Sol) Base address B.a = 100.
Word size = 2 bytes.
No. of columns = $U_2 - L_2 + 1 = 3 - (-1) + 1 = 5$.

First row no. = L1 = 4.
First column no. = L2 = 4 - 1.

Using formula

$X[i,j] = B + W[n(i-L1) + (j-L2)]$

$= 100$

$j = 2$, $i = 6$. given.

$X[6,2] = 100 + 2[5(6-4) + (2-(-1))]$

$= 100 + 2[5(2)+3]$

$= 100 + 2(13)$

$= 126.$

4th Feb

Q. Calc. address of $X[2][5]$ in a 2-D array $X[6][6]$ stored in row major order in m/o. Assume b.a. = 100. & each element req. 4 byte. (Assume lower bound $L1$ & $L2 = 0$.)

Sol) Base address b.a = 100.
word size = 4 byte.
Let us assume
$L1 = L2 = 0$.
Using formula

$X[i,j] = B + W[n(i-L1) + (j-L2)]$
given $i = 2$; $j = 5$.

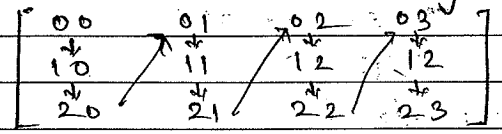
$X[2,5] = 100 + 4[6(2-0) + (5-0)]$

$= 100 + 4[12+5] = 100 + 4[17]$

$= 100 + 68 = 168$

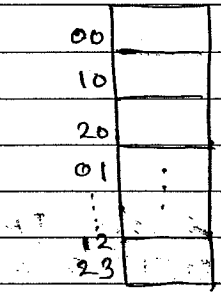
ii) Column major order
In this, elements are stored column by column i.e. all the elements in first column are stored in their order of rows then 2nd column and 3rd column & so on.

Q. Consider the array int a[3][4].



Logical representation.

Storage representation



Address of element of column major order -

$X[i,j] = B.a + W[\text{no. of rows}(j-L2) + (i-L1)]$

Ba = base address
w = word size.

CSVTU
****Q.** Calc. the address of X[0,30] in a 2-D array X[-20...20, 10...35] stored in column major order in the main m/o. Assume the b.a = 500.

Sol^n Base address b.a = 500.
Assume word size w = 1 byte.
No. of rows = U1 - L1 + 1
= 20 - (-20) + 1
= 41.
Acc. to formula

X[i,j] = b.a. + w [rows [j - L2) + (i - L1)]
X[0,30] = 500 + 1 [41 (30 - 10) + (0 - (-20)]
= 500 + [41 X 20 + 20]
= 500 + [820 + 20]
= 500 + [840]
= 1340.

Q. Each element of an array DATA[20][50] requires 4 bytes of storage. B.a of DATA is 2000, determine the loc^n of DATA[10][10]. When array is stored as
a) Row major
b) Column major.

Sol^n Given
B.a = 2000.
w = 4 bytes.
i = 10 = j
no. of cols = 50.
rows = 20.
Assume L1 & L2 = 0.

a) row major.
Acc. to formula
X[i,j] = B.a. + w x [cols x (i - L1) + (j - L2)]
X[10,10] = 2000 + 4 x [50 x (10 - 0) + (10 - 0)]
= 2000 + 4 x [500 + 10]
= 2000 + 4 x [510]
= 2000 + 2040
= 4040

b) Column major.
Acc to formula.
X[i,j] = B.a + w x [rows x (j - L2) + (i - L1)]
X[10,10] = 2000 + 4 x [20 x (10 - 0) + (10 - 0)]
= 2000 + 4 x [200 + 10]
= 2000 + 840
= 2840.

Q. Calc. the address of X[3][2] in 2-D array x[5][4] stored in row major order in the main m/o. Assume the b.a. to be 80 and that each element requires 4 m/o loc^n.
(136)

Sol^n B.a = 80

w = 4 bytes. Assume L_1 & $L_2 = 0$

no. of cols = 4.

Acc to formula

$X[i,j] = b.a + w \times [cols(i-L_1) + (j-L_2)]$

$X[3,2] = 80 + 4[4(3-0) + (2-0)]$

$= 80 + 4[12+2]$

$= 80 + 56$

$= 136$

Q. Calc. the address of $X[4,3]$ in a 2-D array $X[1\cdots5, 1\cdots4]$ stored in row major order in the main m/o. Assume the b.a. to be 1000. and that each element requires 4 words of storage (1056)

Sol? B.a = 1000

w = 4 byte.

$L_1 = 1$, $U_1 = 5$

$L_2 = 1$, $U_2 = 4$

no. of cols = $U_2 - L_2 + 1 = 4 - 1 + 1 = 4$

Acc to formula.

$X[i,j] = b.a + w \times [cols(i-L_1) + (j-L_2)]$

$X[4,3] = 1000 + 4[4(4-1) + (3-1)]$

$= 1000 + 4[12+2]$

$= 1000 + 56 = 1056$.

Application area of Multi-D array

2-D arrays, the most common multi-D arrays, are used to store info. that are normally represent in the table form. 2-D arrays like 1-D arrays, are homogeneous. This means that all of the data in a 2-D arrays is of the same type. E.g. of applications involving 2-D array include -

→ a seating plan for a room (organized by rows & col).

→ a monthly budget (organised by category and month).

→ a grade book where rows might correspond to individual students and columns to student scores.

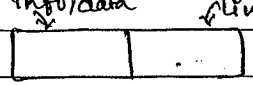
→ Use to implement matrixes in maths.

→ Digital image processing.

LINKED LIST

A linked list or a one way list is a linear collection of data elements, called nodes.; where the linear order is given by means of pointer

Each node is divided into 2 parts-



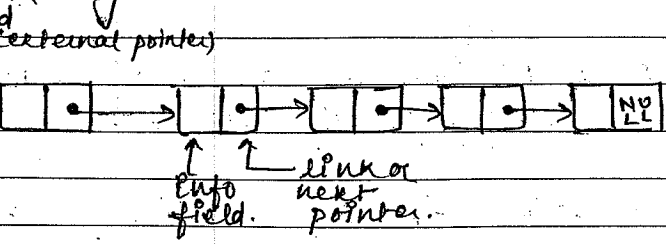
NODE

The 1st part contains the information of element, the 2nd part, called the link field or next pointer field, contains the address of next node in the list.

The entire link list is accessed from an external pointer called head or start pointer, pointing to very first node in the list.

We can access the 1st node through the external pointer, the 2nd node through the next pointer of first node and so on

The pointer of the last node contains a special value called NULL pointer. which is any invalid address.



There are 4 types of link list -

i) Singly link list.
ii) Doubly link list
iii) Circular —"—
iv) Circular double —"—

Advantage

1. Link list are dynamic data structure, it means that they can grow or shrink during the execution of program.
2. Efficient m/o utilization. - m/o is allocated whenever required.
3. Insertion & deletion operation are easier and efficient.
4. Many complex application can be easily carried out with link list.

Disadvantage

1. If no. of fields are more than more m/o is needed.

2. Access to an arbitrary data item is little bit difficult & time consuming.

Link list representation in m/o

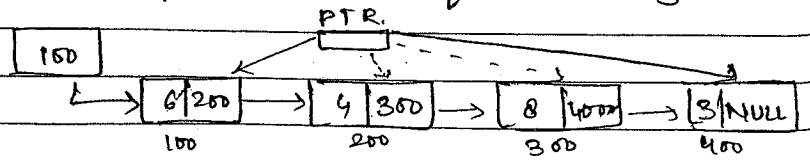
Algorithm

1. Traversing

TRAVERSE (LIST, PTR, START)
Let LIST be a link list in memory. This algorithm traverses list, applying an operation PROCESS to each element of LIST. The variable PTR points to the node currently being processed.

Set
1. PTR := START [Initializes pointer PTR]
2. Repeat steps 3 and 4 while PTR ≠ NULL
3. Apply PROCESS to INFO[PTR]
4. Set PTR := LINK[PTR] [PTR now points to the next node]
   [End of step2 loop]
5. Exit.

Pictorial representation of traversing



5th Feb

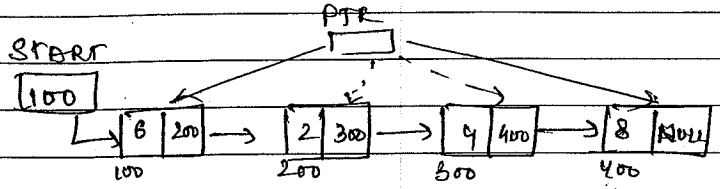
Q. Write an algo. that returns no. of node in the link list.

Q. Write a Procedure that prints the info. of each node of a link list.

Soln PROCEDURE PRINT (INFO, LINK, START)
This procedure prints the info. at each node of the list.

1. Set PTR := START [Initialize pointer PTR]
2. Repeat steps 3 and 4 while PTR ≠ NULL
3. write : INFO [PTR]

4. Set PTR := LINK[PTR] [update pointer PTR]
[End of step 2 loop]
5. Return.



Q. Write a procedure that counts the no. of node in a list.

Sol) PROCEDURE COUNT (INFO, LINK, START)
This procedure counts the no. of nodes in the list.

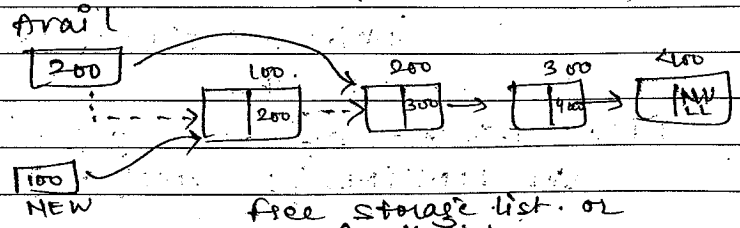
1. Set N:=0 [Initialize counter variable]
2. Set PTR := START [Initialize pointer PTR]
3. Repeat steps 4 and 5 while PTR ≠ NULL
4. Set N := N+1 [Increament counter by 1]
5. Set PTR := LINK[PTR] [update pointer PTR]
[End of step 3 loop]
6. Return.

2. Insertion

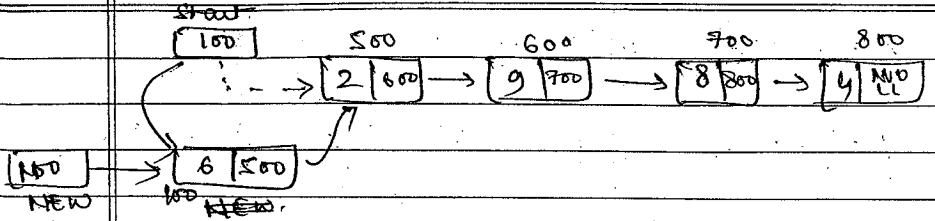
1) Insertion at beginning of a list.

Algorithm: INSFIRST (INFO, LINK, START, ITEM, AVAIL)
This algo. inserts ITEM as the first node in the list.

1. [OVERFLOW?] If AVAIL = NULL, then:
write: Overflow and EXIT
2. [Remove first node from AVAIL list]
3. Set NEW := AVAIL and AVAIL := LINK[AVAIL]
3. Set INFO[NEW] := ITEM [copies new data into NEW node]
4. Set LINK[NEW] := START
[NEW node now points to original first node]
5. Set START := [NEW] [Now START points to the NEW first node]
6. EXIT



free storage list or Avail list.



6th Feb

ii) Insertion after a given node

Algorithm: INSLOC (START, AVAIL, LINK, INFO, ITEM, LOC)

This algorithm inserts ITEM so that ITEM follows the node with location LOC & inserts ITEM as the first item when LOC is NULL

1. [OVERFLOW?] If AVAIL = NULL, then:
   write: Overflow and EXIT

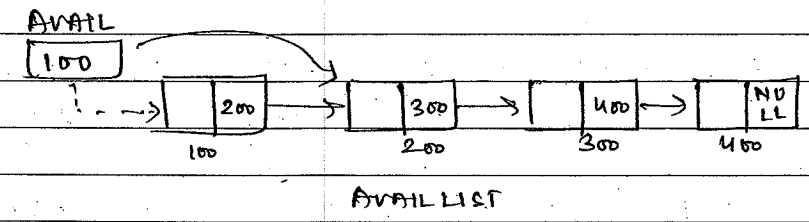
2. [Remove first node from AVAIL list]
   Set NEW := AVAIL and AVAIL := LINK[AVAIL]

3. Set INFO[NEW] := ITEM [copies new data into new node]

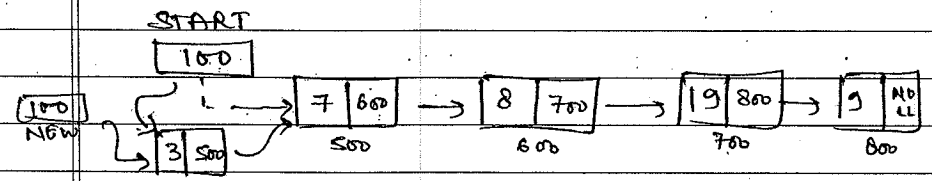
4. If LOC = NULL, then
   Set LINK[NEW] := START and
   START := NEW
   Else: [Insert after node with location LOC]
   Set LINK[NEW] := LINK[LOC] and
   LINK[LOC] := NEW
   [end of If structure]

5. EXIT.

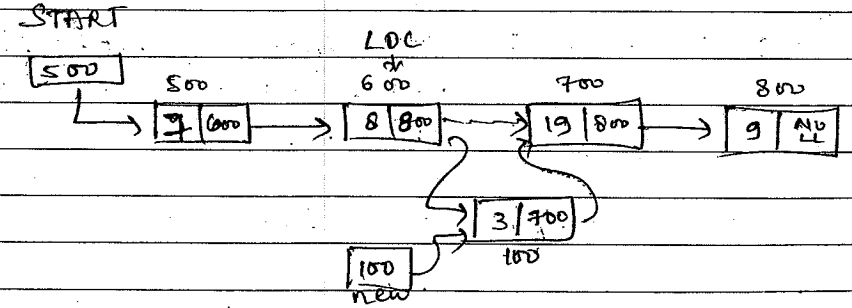
Pictorial Representation



(1) When LOC = NULL



(2) When LOC ≠ NULL



iii) Insertion at the end

Algorithm: INSEND (START, AVAIL, LINK, INFO, ITEM)

This algo. inserts the new node at the end of the list.

1. [OVERFLOW?] If AVAIL = NULL, then: write: Overflow and Exit.

2. [Remove first node from AVAIL list] Set NEW := AVAIL and AVAIL := LINK[AVAIL]

3. Set INFO[NEW] := ITEM [copies new data into new node]

4. Set LINK[NEW] := NULL [new node is the last node]

5. If START == NULL, then: [Empty list] Set START := NEW and EXIT.

Else:

6. Set PTR := START.

7. [Traverse through the list & reach the last node]

Repeat step 8 while LINK[PTR] ≠ NULL.

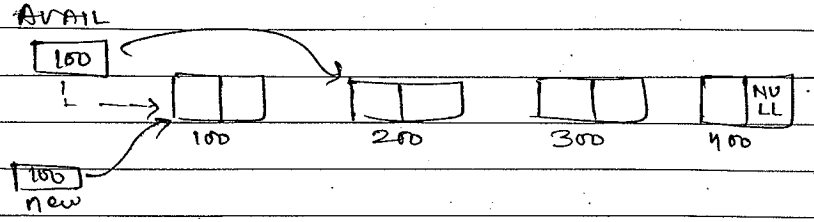
8. Set PTR := LINK[PTR] [update PTR]

[End of step 7 loop]

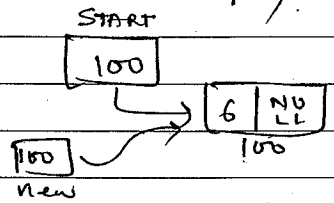
9. Set LINK[PTR] := NEW [insert new node at the end]

[End of If structure]

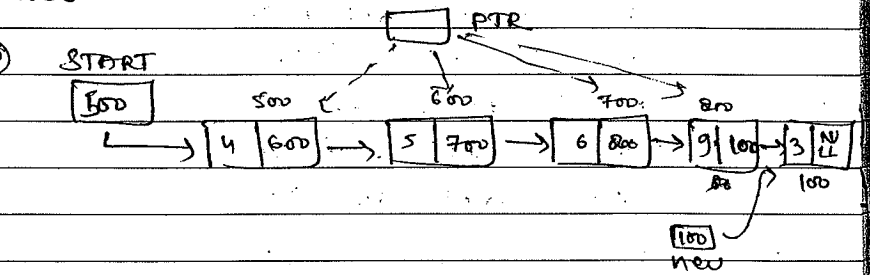
10. Exit



① List Empty.



②



CODE:

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

struct node
{ int info;
  struct node *link;
} *start;
start = NULL;
void create();
void traverse();
void insert_atbeg();
void insert_atend();
void insert_loc();
void del_beg();
void del_end();
void del_loc();



main()
{
  int ch;
  clrscr();
  create();
  while(1)
  { pf("\n MENU\n");
    pf("\n 1. Traverse\n");
    pf("\n 2. Insert at beg.\n");
    pf("\n 3. Insert at end\n");
    pf("\n 4. Insert after loc\n");
    pf("\n 5. Exit\n");
    pf("\n Enter ur choice");
    sf("%d", &ch);
    switch(ch)
    { case 1: traverse();
              break;
      case 2: insert_atbeg();
              break;
      case 3: insert_atend();
              break;
      case 4: insert_loc();
              break;
      case5: exit(0);
      default: pf("oops wrong choice!");
    }
  }
getch();
return(0);
}

void create()
{
  struct node *tmp, *ptr;
  int item;
  char ch='y';
  while(ch=='y'||ch=='y')
  { tmp = (struct node*) malloc(sizeof(struct node));
```

```
if (tmp == NULL)
  { pf("Overflow");
    return;
  }
pf("Enter the item value");
sf("%d", &item);
tmp->info = item;
tmp->link = NULL;
if (start == NULL)
   start = tmp;
else
  { ptr = start;
    while(ptr->link != NULL)
      ptr = ptr->link;
    ptr->link = tmp;
  }
pf("\n Do you want to add more
      node? ");
fflush(stdin);
sf("%c", &ch);
}
}

void traverse()
{ struct node *ptr;
  if (start == NULL)
    pf("\n list is empty");
else
  { ptr = start;
    while (ptr != NULL)
      { pf("\t%d", ptr->info);
        ptr = ptr->link;
      }
  }
}

void insert_at_beg()
{
  struct node *tmp;
  int item;
  tmp = (struct node *) malloc(sizeof(struct node));
  if (tmp == NULL)
    { pf("Overflow"); getch();
      return;
    }
  pf("\n Enter the item value:");
  sf("%d", &item);
  tmp->info = item;
  tmp->link = start;
  start = tmp;
}

void insert_at_end()
{
  struct node *tmp, *ptr;
  int item;
```

* for clearing buffer

```
tmp = (struct node*) malloc(size of (struct node));
if (tmp == NULL)
  { pf("Overflow");
    getch();
    return;
  }
pf("Enter item value:");
sf("%d", &item);
tmp -> info = item;
tmp -> link = NULL;
if (start == NULL)        //if empty list.
   start = tmp;
else                      //if non-empty list
  {
    ptr = start;
    while (ptr -> link != NULL)
       ptr = ptr -> link;
    ptr -> link = tmp;
  }
}

void insert_loc()
{
 struct node *tmp, *ptr;
 int item, pos, i;
 tmp = (struct node *) malloc (sizeof (struct node));
  if (tmp == NULL)
    { pf("Overflow"); getch();
      return;
    }
pf("Enter item value:");
sf("%d", &item);
pf("\n Enter the position after which
      u want to enter:");
sf("%d", &pos);
tmp -> info = item;  ptr = start;
for (i=0; i<pos; i++)
  {  if (ptr == NULL)
       {pf("Enter correct loc:");
          return;
       }
     ptr = ptr -> link;
  }
tmp -> link = ptr -> link;
ptr -> link = tmp;
}
```

8th Feb

3. Deletion

Q. Write an algo. for deletion of -
i) first node of list
ii) Last node of the list.
iii) A node following a given node.

Sol' i) First node of the list.

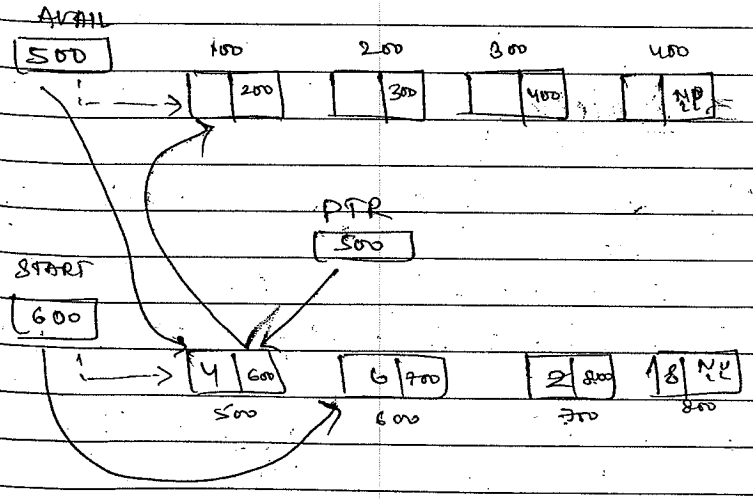
Algorithm: DEL_FIRST(INFO, LINK, AVAIL, START)

Let LIST be a link list in m/o. This algo. deletes the first node from a link list. If there is only single node in the list, the algo. removes that node & make the list empty, by initializing START = NULL.

1. [UNDERFLOW?] If START = NULL, then:
   write: Underflow and EXIT.

2. [Return node to AVAIL list]
   Set PTR := START and START := LINK[PTR]
   Set LINK[PTR] := AVAIL and AVAIL := PTR

3. Exit.



ii) Last node of the list

Algorithm: DEL_LAST (INFO, LINK, AVAIL, START)

Let LIST be a link list in m/o. This algo. deletes the last node of the list.

1. [UNDERFLOW?] If START = NULL, then:
   write: Underflow and Exit.

2. [Single node?] If LINK[START] := NULL, then:
   Set PTR := START and START := NULL,
   LINK[PTR] := AVAIL and AVAIL := PTR and exit.
   [end of if structure]

3. Else

3. [Initialize pointer variable]
   Set SAVE := START and PTR := LINK[START] [or SAVE]

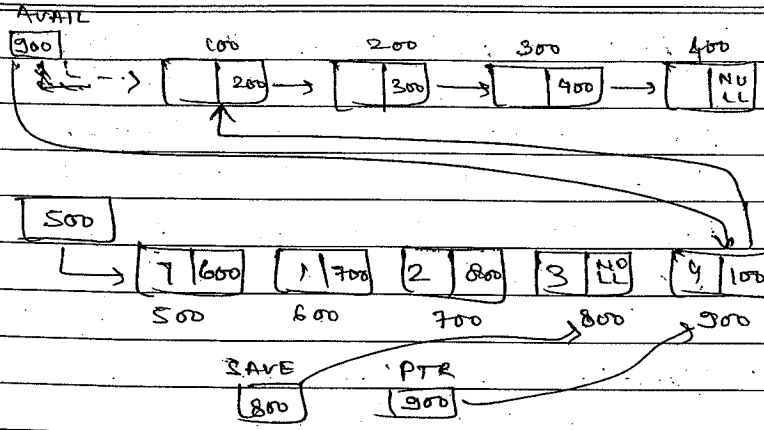
4. Repeat step 5 while LINK[PTR] ≠ NULL

5. SAVE := PTR and PTR := LINK[PTR]
   [update pointer PTR]
   [end of step 4 loop]

6. LINK[SAVE] := NULL

7. [Return node to AVAIL LIST]
   Set LINK[PTR] := AVAIL and AVAIL := PTR

8. EXIT.



iii) A node following a given node.

Algorithm: DELETE-AFTER (INFO, LINK, AVAIL, START, LOC, LOCP)

Let LIST be a link list in memory. This algo. deletes the node N with location LOC. LOCP is the loc" of the node which preceedes N or, when N is the first node LOCP = NULL

1. If LOCP = NULL, then:

   Set START := LINK[START] [Delete node N]

   Else:

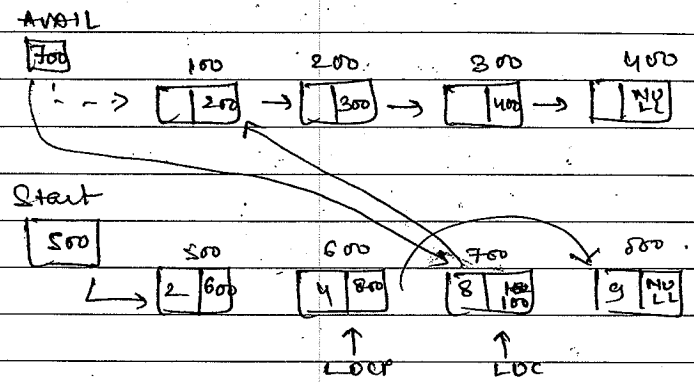
   Set LINK[LOCP] := LINK[LOC] [Delete node N]

   [end of if structure]

2. [Return node to AVAIL LIST]:

   Set LINK[LOC] := AVAIL & AVAIL := LOC.

3. EXIT.



CODE:

```
void del_beg()
{ struct node *tmp;
  if(start==NULL)
  { pf("\n List is empty (underflow)");
    getch();
    return;
  }
  tmp = start;
  start = start->link;
  pf("\n Deleted node is %d", tmp->info);
  free(tmp);
}
```

```
void del_end()
{ struct node *tmp, *prev, *ptr;
  if (start==NULL)
    { pf("underflow");
      getch();
      return;
    }
  if(start->link==NULL)
    { tmp=start;
      start=NULL;
    }
  else
    { prev=start;
      ptr=start->link;
      while( ptr->link!=NULL)
        { prev=ptr;
          ptr=ptr->link;
        }
      tmp=ptr;
      prev->link=NULL;
    } free(tmp);

void del_loc()
{ struct node *tmp, *prev, *ptr;
  int pos, i;
  if (start==NULL)
    { pf("underflow");
      getch(); return;
    }
  pf("\n Enter the position after
      which you want to delete:");
  sf("%d", &pos); if(pos<1) { pf("Invalid pos"
                              return;}
  prev=start;
  ptr=start->link;
  for(i=1; i<=pos; i++)
    { prev=ptr;
      ptr=ptr->link;
      if(ptr==NULL)
        {pf("Invalid pos");
         return;
        }
    }
  prev->link=ptr->link;
  tmp=ptr;   pf("\n deleted node %d"
                tmp->info);
  free(tmp);
}
```

4th Feb

Q. Briefly enumerate the various operations that can be performed on link list. Write a func^n that removes all duplicate elements from a link list.

```
void dup()
{ struct node *ptr, *ptr1, *prev, *tmp;
  int data;
  if (start==NULL)
    { pf("list is empty");
      return;
```

```
ptr = start;
while (ptr != NULL)
{ data = ptr->info;
  prev = ptr;
  ptr1 = ptr->link;
  while(ptr1 != NULL)
  { if (data == ptr1->info)
    { tmp = ptr1;
      prev->link = ptr1->link;
      ptr1 = ptr1->link;
      free(tmp);
    }
    else
    { prev = ptr1;
      ptr1 = ptr1->link;
    }
  }
  ptr = ptr->link;
}
}
```

4. Sorting

```
void sort()
{ struct node *p1, *p2;
  int temp;
  if (start == NULL)
  { pf("List is empty");
    return;
  }
  for (p1 = start; p1->link != NULL; p1 = p1->link)
  { for(p2 = p1->link; p2 != NULL; p2 = p2->link)
    { if (p1->info > p2->info)
      { tmp = p1->info;
        p1->info = p2->info;
        p2->info = temp;
      }
    }
  }
}
```

5. Reverse

```
void reverse()
{ struct node *p1, *p2, *p3;
  if (start == NULL)
  { pf("List is empty");
    return;
  }
  p1 = start;
  p2 = p1->link;
  p3 = p2->link; p1->link = NULL; p2->link = p1;
  while (p3 != NULL)
  { p1 = p2;
    p2 = p3;
    p3 = p3->link;
    p2->link = p1;
  }
  start = p2;
```

12th feb

6. Searching

i) When list is unsorted.

Algorithm: SEARCH (LINK, INFO, START, ITEM, LOC).

Let LIST be a link list in m/o. This algo finds the loc^n LOC of the node where item first appears in list or sets LOC = NULL (If item not found).

1. [Initialize pointer PTR] PTR := START.
2. Repeat step 3 while PTR ≠ NULL.
3. If ITEM = INFO[PTR], then:
   Set LOC = PTR and Exit.

   Else
   PTR := LINK[PTR] [Update PTR]

   [end of If struct]
   [End of step 2 loop]
4. [Search unsuccessful] Set LOC := NULL.
5. Exit.

ii) When list is sorted

Algorithm: SEARCH (LINK, INFO, START, ITEM, LOC).

Let LIST be a sorted list in m/o. This algo finds the loc^n LOC of the node where item first appears in list or sets LOC = NULL.

1. [Initialize pointer PTR] Set PTR := START.
2. Repeat step 3 while PTR ≠ NULL.
3. If ITEM ≠ INFO(PTR), then:
   Set PTR := LINK[PTR] [Update PTR]

   Else if: ITEM = INFO[PTR], then: [Search successful]
   Set LOC := PTR and Exit.

   Else: Set LOC := NULL and exit. [Search Unsuccessful]
   [end of If struct]
   [End of step 2 loop]
4. [Search unsuccessful] Set LOC := NULL.
5. Exit.

7 Merging

Algorithm : MERGE [START1, START2, LINK, INFO, START3]

Let LIST1 and LIST2 be sorted list in m/o. START1 & START2 be two ~~start~~ starts pointing to first nodes of two lists. and both are merged together & into a sorted 3rd link list.

1. [Initialize pointers] Set PTR1 := START1 & PTR2 := START2.

2. If PTR1 = NULL, then:
set START3 := PTR2 & exit.

If PTR2 = NULL, then:
set START3 := PTR1 & exit

3. If INFO[PTR1] < INFO[PTR2]
set START3 := PTR1 and PTR1 := LINK[PTR1]

Else:
Set START3 := PTR2 and PTR2 := LINK[PTR2]
[end of If struct].

4. Set PTR3 := START3 [Initialize pointer]

5. Repeat while PTR1 ≠ NULL & PTR2 ≠ NULL
If INFO[PTR1] < INFO[PTR2], then:
LINK[PTR3] := PTR1. and PTR1 := LINK[PTR1]

Else:
LINK[PTR3] := PTR2. and PTR2 := LINK[PTR2]
[end of If struct.]
PTR3 := LINK[PTR3]
[end of steps loop]

6. If PTR1 ≠ NULL, then:
LINK[PTR3] := PTR1.

7. If PTR2 ≠ NULL, then:
LINK[PTR3] := PTR2

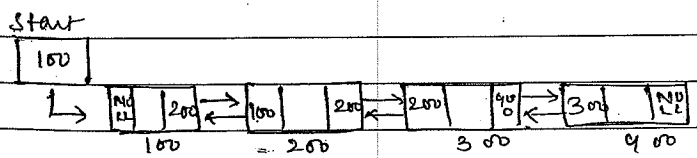
8. Exit.

13th feb

Doubly Link list

A pointer field forward, which contains the location of next node in the list.

A pointer field back, which contains the location of preceeding node in the list.



Memory representation

Operations on doubly link list

1. Traversing

Algorithm: TRAVERSE (INFO, FORW, BACK, ~~AVAIL~~ START)
Let LIST be a doubly link list in m/o. This algo. traverses the list.

1. [Empty?] If START = NULL, then:
   write: "List is empty" & EXIT.
2. [Initialize pointer variable] Set PTR := START.
3. Repeat Steps 4 & 5 while PTR ≠ NULL
4. Apply PROCESS to INFO[PTR]
5. [Update PTR] Set PTR := FORW[PTR]
   [End of step 3 loop]
6. EXIT.

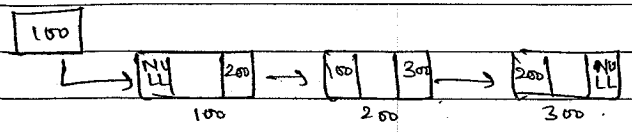
2. Insertion

i) Insert at first pos"

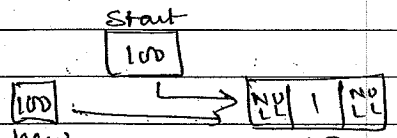
Algorith: INSERT_FIRST_DLL(INFO, FORW, BACK, START, AVAIL, ITEM).
Let LIST be a DLL in m/o. This algo. inserts new node at first pos".

1. [Overflow?] If AVAIL = NULL, then:
   write: "Overflow" & exit.
   [end of If struct].
2. [Remove first node from AVAIL LIST]
   Set NEW := AVAIL & AVAIL := FORW[AVAIL]
3. [Copy DATA to a new node] INFO[NEW] := ITEM
4. [List is empty?] If START = NULL, then:
   Set START := NEW, FORW[START] := NULL
   & BACK[NEW] := NULL.
   OR START
   Else:
   Set BACK[START] := NEW, FORW[NEW] := START
   BACK[NEW] := NULL & START := NEW.
   [end of If struct.]
5. Exit

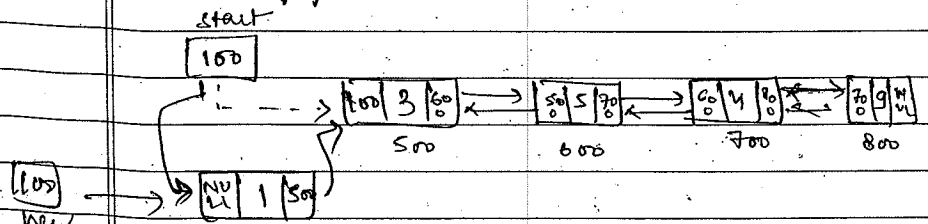
AVAIL



① Empty list.



② Non-empty.



ii) Insert at last

Algorithm: INS_LAST_DLL (INFO, FORW, BACK, START, AVAIL, ITEM)

Let LIST be a DLL in m/o. This algo inserts a new node at last.

1. [Overflow?] If AVAIL = NULL, then:
   Write: "Overflow" & exit.
   [end of if struct].

2. [Remove first node from AVAIL LIST]
   Set NEW := AVAIL & AVAIL := FORW[AVAIL]

3. [Copy DATA to new node] INFO[NEW] := ITEM

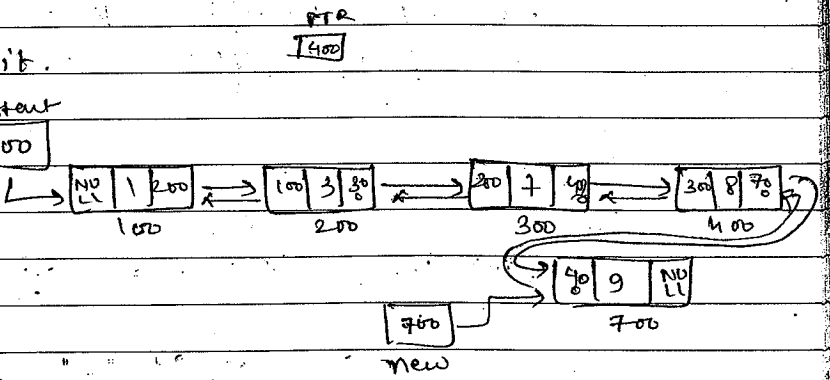
4. [Empty list?] If START = NULL, then:
   Set START := NEW, FORW[START] := NULL
   & BACK[NEW] := NULL & exit.
   [end of If struct].

5. [Initialize pointer] Set PTR := START.

6. Repeat while FORW[PTR] ≠ NULL
   PTR := FORW[PTR]
   [end of Step 6 loop].

7. [Insert new node at last pos]
   Set FORW[NEW] := NULL,
   BACK[NEW] := PTR & FORW[PTR] := NEW

8. Exit.



ii) Insert at location

Algorithm: INS_LOC_DLL(INFO, FORW, BACK, START, AVAIL, ITEM, LOCA, LOCB)

Let LIST be a DLL in m/o. This algo. inserts a new node b/w LOCA & LOCB.

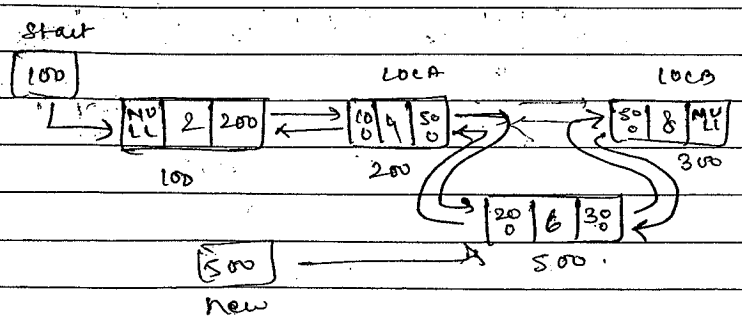
1. [OVERFLOW?] If AVAIL = NULL, then:
   write: "Overflow" & exit.
   [end of If struct].

2. [Remove first node from AVAIL LIST]
   Set NEW := AVAIL & AVAIL := FORW[AVAIL]

3. [Copy DATA to new node] INFO[NEW] := ITEM.

4. [Insert b/w LOCA & LOCB]
   Set BACK[NEW] := LOCA, FORW[NEW] := LOCB.
   FORW[LOCA] := NEW & BACK[LOCB] := NEW

5. EXIT.



3. Deletion

i) Delete the first node.

Algorithm: DEL_FST_DLL(FORW, INFO, BACK, AVAIL, START)

Let LIST be a DLL in m/o. This algo. deletes the first node

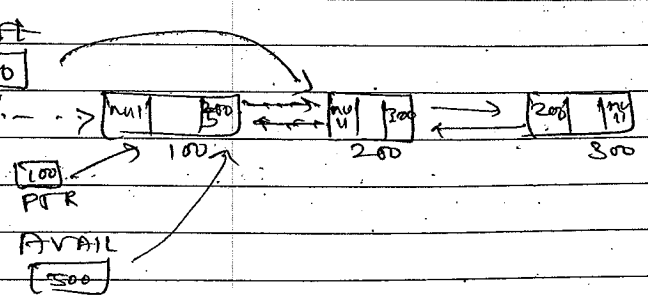
1. [Underflow?] If START = NULL, then:
   write: "Underflow" & exit.

2. [Single node] If FORW[START] = NULL, then:
   Set PTR := START & START := NULL & Exit.

3. Else:
   Set PTR := START, START := FORW[PTR],
   BACK[START] := NULL.

3. [Return deleted node to AVAIL LIST]
   Set FORW[PTR] := AVAIL & AVAIL := PTR

4. EXIT.



ii) Delete the last node

Algorithm: DEL_LST_DLL(FORW, BACK, INFO, AVAIL, START)
Let LIST be a DLL in m/o. This algorithm deletes the last node.

1. [Underflow?] If START = NULL, then:
   write: "Underflow" & exit.

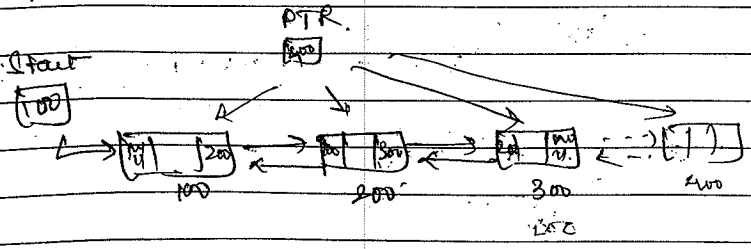
2. [Single Node?] If FORW[START] = NULL, then:
   Set PTR := START & START := NULL & exit.
   Else.

3. [Initialize pointer] Set PTR := START.
   Repeat while FORW[PTR] ≠ NULL
   Set PTR := FORW[PTR] [update PTR]
   [end of step 3 loop]

4. Set FORW[BACK[PTR]] := NULL
   [deletes last node]

5. [Return deleted node to AVAIL LIST]
   Set FORW[PTR] := AVAIL & AVAIL := PTR

6. EXIT.



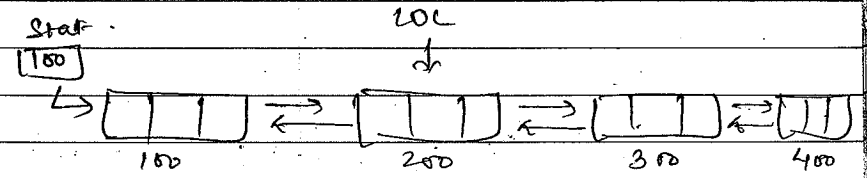
iii) Delete the given node

Algorithm: DEL_LOC_DLL(FORW, INFO, BACK, AVAIL, START, LOC)
Let LIST be a DLL in m/o.
Suppose we are given the locⁿ LOC of the node 'N' in the LIST & we want to delete 'N' from the list.

1. [Delete node] Set FORW[BACK[LOC]] := FORW[LOC] & BACK[FORW[LOC]] := BACK[LOC].

2. [Return deleted node to AVAIL LIST]
   Set FORW[AVAIL] := NULL & AVAIL := LOC.

3. EXIT.



iv) Delete after a given node

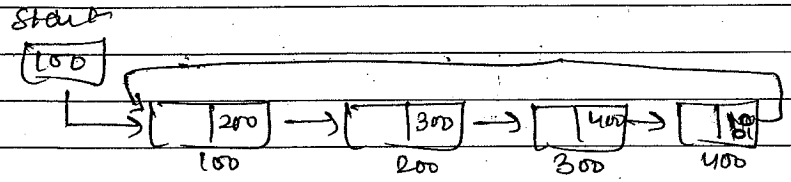
Algorithm: DEL_AFTERLOC(FORW, INFO, BACK, AVAIL, START, LOC)
Let LIST be a DLL in m/o.
Suppose we are given the locⁿ LOC of the node 'N'. & we have to delete the next node of LOC.

1. [Delete Node] Set FORW[LOC] := FORW[FORW[LOC]]
and BACK[FORW[FORW[LOC]]] := LOC

2. [Return deleted node to AVAIL LIST]
Set FORW[PTR] := AVAIL & AVAIL := PTR

Circular Link List

It is a link list in which link of last node contains address of the first node.



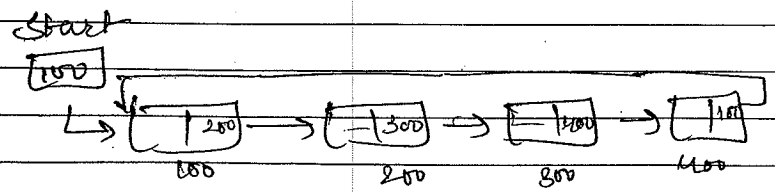
Memory representation

Operations

1 Traversing

Algorithm: TRAVERSE-CLL(INFO, LINK, START)
Let LIST be a CLL in m/o. This algo. traverses the list.

1. [Empty list?] If START = NULL, then:
   write: "List is empty" & exit.
   [end of if struct]
2. [Initialize PTR] Set PTR := START
3. Repeat steps 4&5 while LINK[PTR] ≠ START
4. Apply PROCESS to INFO[PTR]
5. [Update PTR] PTR := LINK[PTR]
6. [Process last node] INFO[PTR]
7. Exit.



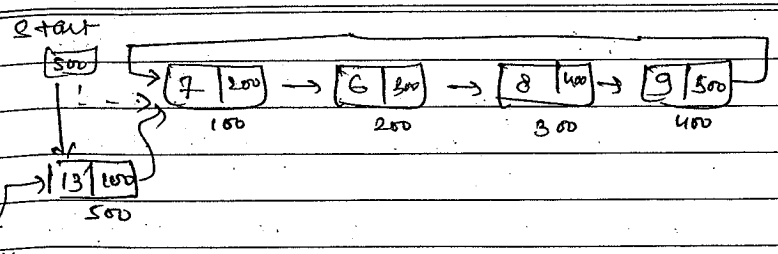
2 Insertion

i) Insertion at first position.

Algorith: INS-AT-CLL(INFO, LINK, START, AVAIL, ITEM)
Let LIST be a CLL in m/o. This algo. inserts a first node in the list.

1. [Overflow?] If AVAIL = NULL, then:
   write: "Overflow" & exit.
2. [Remove first node from AVAIL LIST]
   Set NEW := AVAIL & AVAIL := LINK[AVAIL]
3. [Copy DATA in new node] Set INFO[NEW] := ITEM
4. [Empty list?] If START = NULL, then:
   Set START := NEW & LINK[NEW] := NEW.

   Else: [Initialize pointer]
   Set PTR := START.
   Repeat while LINK[PTR] ≠ START.
   PTR := LINK[PTR] [update PTR]
   [end of loop]
   [end of If struct].
5. [Insert new node at first pos"]
   Set LINK[NEW] := START, START := NEW,
   & LINK[PTR] := START
6. Exit.



ii) Insert at last node

Algorithm: INS_LAST_CLL (INFO, LINK, START, AVAIL, ITEM).

Let LIST be a CLL in m/o. This algo. inserts a the node at last.

1. [Overflow?] If AVAIL = NULL, then:
   write: "Overflow" & exit.

2. [Remove first node from AVAIL LIST]
   Set NEW := AVAIL & AVAIL := LINK[AVAIL]

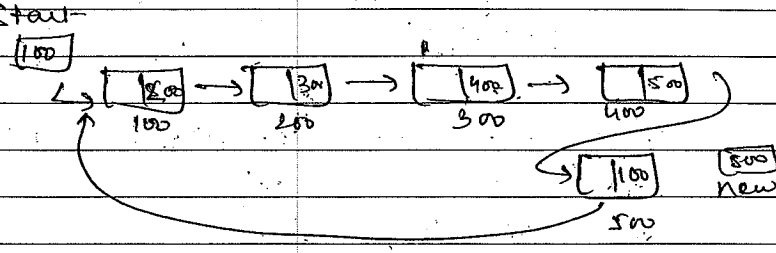
3. [Copy DATA in new node] Set
   Set INFO[NEW] := ITEM.

4. [Empty List?] If START = NULL, then:
   Set START := NEW & LINK[NEW] := NEW

   Else: [Initialize pointer]
   Set PTR := START
   Repeat while LINK[PTR] ≠ START
   PTR := LINK[PTR] [Update PTR]
   [end of loop struct]
   [end of if struct].

5. [Insert new node at last]
   Set LINK[PTR] := NEW &
   LINK[NEW] := START.

6. Exit.



iii) Insert new node after location of given data.

Algorithm: INS_LOC_CLL (INFO, LINK, START, AVAIL, ITEM, LOC)

Let LIST be a CLL in m/o. This algo. inserts the new node after a given location

1. [Overflow?] If AVAIL = NULL, then:
   write: "Overflow" & exit.

2. [Remove first node from AVAIL LIST]
   Set NEW := AVAIL & AVAIL := LINK[AVAIL]

3. [Copy DATA in new node]
Set DNFO[NEW] := ITEM.

4. [Initialize pointer] Set PTR := START.

5. Repeat step 6 & 7 while LINK[PTR] ≠ START.

6. If INFO[PTR] = DATA, then:
Set LINK[NEW] := PTR, LINK[PTR],
LINK[PTR] := NEW. & Exit.

7. [update PTR] set PTR := LINK[PTR]
[end of if struct]
[end of step 5 loop]

8. If INFO[PTR] = DATA, then:
LINK[PTR] := NEW & LINK[NEW] := START
[end of If struct].

9. Exit. Else:
Write; "Data not found" &
[end of If struct]

9. Exit.

3. Deletion

i) Deletion of first node

Algorithm: DEL-FIRST-CLL(INFO, LINK, START, AVAIL, LAST).

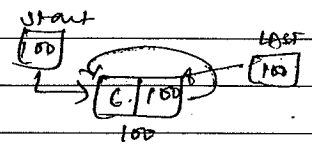
Let LIST be a CLL in m/o. This algo. delete first node where START pointer points the first node and last pointer points the last node

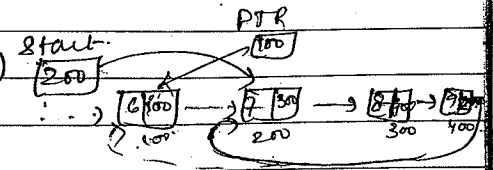
1. [Underflow?] If START = NULL, then:
write "Underflow" and exit.
[end of if struct].

2. [Single node?] If START = LINK[START], then:
Set PTR := START, START := NULL & LAST := NULL
Else:
Set PTR := START, START := LINK[PTR] &
LINK[LAST] := START.

3. [Return deleted node to AVAIL LIST]
Set LINK[PTR] := AVAIL & AVAIL := PTR.

4. Exit.

i)


ii)


iii) Delete the last node

Algorithm: DEL_LST_CLL (INFO, LINK, START, AVAIL, LAST)

Let LIST be a C.link.list in memory. This algo. deletes last node where START pointer points to the first node and LAST pts. points to last node.

1. [Underflow?] If START = NULL, then: write "Underflow" & exit.

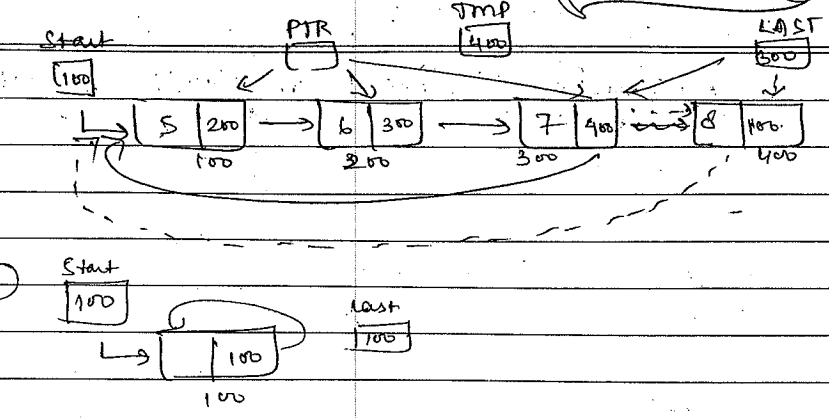
2. [Single node?] If START = LINK[START], then: Set PTR := START, START := NULL & LAST := NULL.

   Else: Set PTR := START [Initialize pointer]
   Repeat while LINK[PTR] ≠ START [val]
   Set PTR := LINK[PTR] [update pointer]
   [end of loop]
   [end of if struct]

3. [Delete last node] Set tmp := LAST,
   Set LINK[PTR] := START & LAST := PTR

4. [Return deleted node to AVAIL LIST]
   Set LINK[tmp] := AVAIL & AVAIL := tmp.

5. Exit.



iii) Delete after a given node

Algorithm: DEL_AFT_CLL (INFO, LINK, START, LAST, AVAIL, LOC, LOCP)

Let the LIST be a CLL in m/o. This algo. deletes a node after a given location. LOC points the node which is to be deleted and LOCP points the node previous to LOCP.

1. [Delete first node?] If LOCP = NULL, then:
   Set START := LINK[LOC] & LINK[LAST] := START

   Else:
   Set LINK[LOCP] := LINK[LOC]
   [end of if struct]

2. [Return deleted node to AVAIL LIST]
   Set LINK[LOC] := AVAIL & AVAIL := LOC.

Analysis of Time complexity of LINK LIST

Operation of L.L.	Time complexity
1. Adding an element at beginning.	O(1)
2. Adding an element at internal posn	O(n)
3. Deleting an element from beginning	O(1)
4. Deleting an element at internal posn	O(n)
5. Retrieving the nth element	O(n)

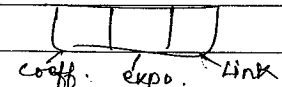
Application of L.L

1. Representation and manipulation of polynomial

The imp. application of L.L. is to represent the polynomial and their representation.

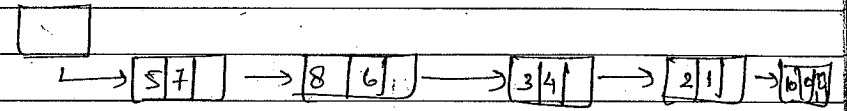
The main advantage of link list for polynomial is that it can accomodate no. of n's in growing size.

Structure of a node



Representation of polynomial

eg. $P(n) = 5n^7 + 8n^6 + 3n^4 + 2n + 10$.



$P = 5 + 2x + 4x^2 + 5x^5 + 6x^7$

$Q = 4 + 5x + 7x^4 + 9x^6$

$R = 9 + 5x + 7x^2 + 4x^4 + 5x^5 + 9x^6 + 6x^7$

Algorithm: ADD-POLY(P, Q, R, START1, START2, LINK, AVAIL, START3, COEF, EXP).

1. [Overflow ?] If AVAIL = NULL, then write "Overflow" & exit.

2. [Initialize pointer] Set PPTR := START1 and QPTR := START2.

3. Repeat steps 4 to 7 while PPTR ≠ NULL & QPTR ≠ NULL.

4. [Remove first node from AVAIL LIST].
Set NEW := AVAIL & AVAIL := LINK[AVAIL]

5. CASE1: [Exponents are equal?] If EXP[PPTR] = EXP[QPTR], then:
Set EXP[NEW] := EXP[PPTR],
COEF[NEW] := COEF[PPTR] + COEF[QPTR].
& LINK[NEW] := NULL.
If START3 = NULL, then:
Set START3 := NEW
Else:
[Initialize pointer] Set RPTR := START3
Repeat while LINK[RPTR] ≠ NULL
[update pointer] RPTR := LINK[RPTR]
[end of loop]
LINK[RPTR] := NEW
[end of if structure].
[update pointer] PPTR := LINK[PPTR]
& QPTR := LINK[QPTR].
[end of CASE 1].

6. CASE2: If EXP[PPTR] < EXP[QPTR], then:
Set EXP[NEW] := EXP[PPTR],
COEF[NEW] := COEF[PPTR],
LINK[NEW] := NULL.
If START3 = NULL, then:
Set START3 := NEW
Else:
[Initialize pointer] Set RPTR := START3.
Repeat while LINK[RPTR] ≠ NULL.

[Update ptr] RPTR := LINK[RPTR]
[end of loop]
Set LINK[RPTR] := NEW.
[end of if struct]
[update ptr] Set PPTR := LINK[PPTR]
[end of CASE2]

7. CASE3: If EXP[PPTR] ≠ EXP[QPTR], then:
Set EXP[NEW] := EXP[QPTR],
COEF[NEW] := COEF[QPTR],
LINK[NEW] := NULL.
~~ELSE:~~
If START3 = NULL, then:
Set START3 := NEW.
Else:
[Initialize ptr] Set RPTR := START3
Repeat while LINK[RPTR] ≠ NULL
[Update] RPTR := LINK[RPTR]
[end of loop]
Set LINK[RPTR] := NEW
[end of if struct]
[update ptr] Set QPTR := LINK[QPTR]
[end of CASE3]

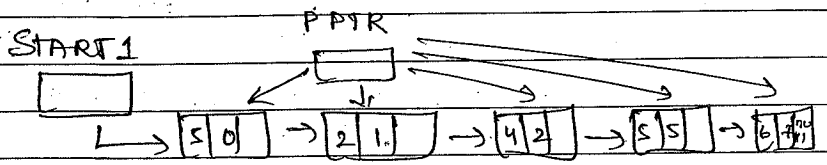
[end of step 3 loop]

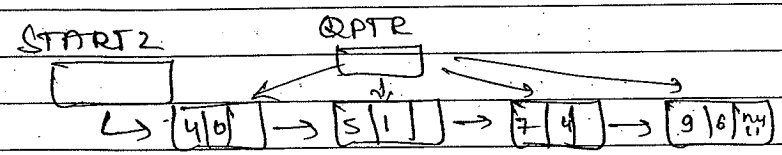
8. Repeat while PPTR ≠ NULL
[Remove first node from AVAIL LIST] Set NEW := AVAIL &
AVAIL := LINK[AVAIL].
Set EXP[NEW] := EXP[PPTR],&
COEF[NEW] := COEF[PPTR] &
LINK[NEW] := NULL.
If START3 = NULL, then:
Set START3 := NEW.
Else:
[Initialize ptr] Set RPTR := START3
Repeat while LINK[RPTR] ≠ NULL
[update] RPTR := LINK[RPTR]
[end of loop]
Set LINK[RPTR] := NEW
[end of if struct]
[update] Set PPTR := LINK[PPTR]
[end of step 8 loop]

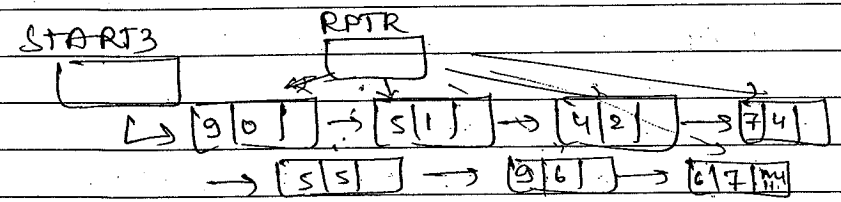
9. Repeat while QPTR ≠ NULL
[Remove first node from AVAIL]
Set NEW := AVAIL & AVAIL := LINK[AVAIL]
Set EXP[NEW] := EXP[QPTR],
COEF[NEW] := COEF[QPTR] &
LINK[NEW] := NULL.
If START3 = NULL, then:
Set START3 := NEW
Else:
[Initialize ptr] Set RPTR := START3
Repeat while LINK[RPTR] ≠ NULL
[update] RPTR := LINK[RPTR]
[end of loop]
Set LINK[RPTR] := NEW

[end of If struct]
[update] Set QPTR := LINK[QPTR]
[end of step 9 loop]

10 Exit.







UNIT-II

STACK

A stack is a list elements in which an element may be inserted or deleted only at one end i.e.; called TOP of stack.
This means, in particular, that elements are removed from stack in reverse order of that in which they are inserted into stack.
Stack follows the LIFO technique.

In stack only 3 operations are possible-

i) PUSH :- It is a term used to insert an element into a stack.

ii) POP :- It is a term used to delete an element from a stack.

iii) Traversing :- It is a term used to traverse all element from a stack.

Implementation of Stack.

We can implement stack by two ways:
i) Array
ii) Link list.

MAXSTK → max. no. of elements that can be held by the Stack

i) Using an array

a) PUSH

Algorithm: PUSH (STACK, TOP, MAXSTK, ITEM)
"This algo. Pushes the ITEM on to the STACK at TOP pos", initially TOP = 0 & Maxm of stack is MAXSTK.

1. [STACK already filled?] If TOP = MAXSTK, then: write "Overflow" & exit.
2. Set TOP := TOP + 1 [Increase TOP by 1].
3. Set STACK[TOP] := ITEM. [PUSH the Item on to the stack]
4. Exit.

ii) POP

Algorithm: POP (STACK, TOP, ITEM)
"This algo. deletes the top element of STACK & assign it to variable ITEM.

1. [Empty STACK?] If TOP = 0, then: write: "Underflow" & exit.
2. Set ITEM := STACK[TOP] [assign the value of Top element to ITEM]
3. Set TOP := TOP - 1. [Decrease TOP by 1]
4. Exit.

iii) Traversing

Algorithm: TRAVERSE (STACK, TOP)
"This algo. traverses the STACK by assigning variable 'I' by value of TOP.

1. [empty STACK?] If TOP = 0, then: write: "Stack is empty" & exit.
2. Set I = TOP [Initialize counter variable]
3. Repeat step 4 & 5 while I > 0
4. Apply PROCESS to STACK[I].
5. [Decrease counter variable] Set I := I - 1. [Decrease by 1.]
   [end of step 3 loop]
6. Exit.

CODE:

```
#include<stdio.h>
#include<conio.h>
#define MAX 10
 int TOP = -1;

 main()
 { int item, ch;
```

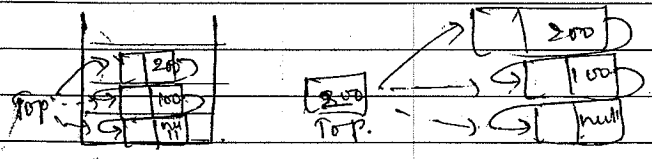
```
void push(int stack[], int item)
 {
  if (TOP == MAX-1)
    { pf("stack is full");
      return;
    }
  TOP = TOP+1;
  STACK[TOP] = item;
 }

void pop(int stack[])
 { int item;
  if (TOP == -1)
    { pf("stack is empty");
      return;
    }
  TOP Item = stack[TOP];
   TOP = TOP-1;
 }

void traverse (int stack[])
 { int i;
  if (TOP == -1)
    { pf("stack empty");
      return;
    }
  for(i=0; i<=TOP; i++)
    pf("%d", stack[i]);
}
```

Implementation of stack using Link list-

1. PUSH

Algo: PUSH (LINK, INFO, AVAIL, TOP, ITEM)
This algo pushes an item into the L.L.

1. [Overflow?] If AVAIL = NULL, then:
   write "Overflow" & exit.
2. [Remove first node from AVAIL LIST]
   Set NEW := AVAIL & AVAIL := LINK[AVAIL]
3. [copies the ITEM into new node]
   Set INFO[NEW] := ITEM
4. [New node points to the original top node in the stack] Set LINK[NEW] := TOP.
5. Set TOP := NEW [Reset TOP to point new node]
6. Exit.



2. POP

Algo: POP (LINK, INFO, AVAIL, TOP, ITEM)
This algo deletes the TOP element of ~~the~~ Linked stack & assign it to ITEM

1. [Underflow?] If TOP = NULL, then:
   write: "Underflow" & exit.
2. [copies the top element into ITEM]
   Set ITEM := ~~STACK~~ INFO[TOP].
3. Set TEMP := TOP & TOP := LINK[TOP]
4. [Return deleted node to AVAIL list]
   Set LINK[TEMP] := AVAIL & AVAIL := TEMP.
5. Exit.

3. Traverse

Algo: TRAVERSE (LINK, INFO, TOP)
This algo traverses the linked stack.

1. If TOP = NULL [Stack empty?]
   write "Empty stack" & exit.
2. Set PTR := TOP [Initialise pointer PTR]
3. Repeat steps 4 & 5 while PTR ≠ NULL
4. Apply PROCESS to INFO[PTR]
5. Set PTR := LINK[PTR]
   [end of ~~if~~ step 3 loop]
6. Exit.

200

Next
200

26th Feb

Priority ∧ or ↑	Prefix	Infix	Postfix
* / % (L→R)	+ab	a+b	ab+
+ - (L→R)	-+abc	a+b-c	ab+c-
	+a*bc	a+b*c	abc*+
	-+a*bcd	a+b*c-d	abc*+d-
	+*ab/cd	a*b+c/d	ab*cd/+
	/*-ab+cd-ef	(a-b)*(c+d)/(e-f)	ab-cd+*ef-/
	+-+*ab%/cefg*hi	a*b+c/e%f-g+(h*i)	ab*ce/f%+g-hi*+
	+a**-bc*%/dfg+hij	a+(b-c)*((d/f)%g*(h+i))*j	abc-df/g%hi+**j*+

* If it is of 2M or 4M then by hand otherwise do it by stack.

Q. ((a+b)/d) + (e-f)

Post:
⇒ ([ab+]/d) + [ef-]
⇒ [ab+d/] + [ef-]
⇒ ab+d/ef-+.

Pre
⇒ ([+ab]/d) + [-ef]
⇒ [/+abd] + [-ef]
⇒ +/+abd-ef.

Q. Translate, by inspection & hand, each infix exp. into its equivalent postfix exp.

i) (a-b) * (d/e)
ii) (a+b↑d) / (e-f) + g
iii) a * (b+d) / e-f * (g+h/k)

Sol:
i) [ab-] * [de/]
ab-de/*

ii) (a+[bd↑]) / [ef-] + g
[abd↑+] / [ef-] + g
[abd↑+ef-]/ + g
[abd↑+ef-/g+]

iii) a * [bd+] / e-f * (g+[hk/])
a * [bd+] / e-f * [ghk/+]
[abd+*] / e-f * [ghk/+]
[abd+*e/] - [fghk/+*]
abd+*e/fghk/+*-

Q. Translate, by inspection and hand, each infix exp. into its equivalent prefix exp.

Sol:
i) [-ab] * [/de]
*-ab/de

Infix → Postfix ← POLISH
Infix → Prefix ← Inverse POLISH

ii) $(a+[\uparrow bd]) / [-ef] + g$
$[+a\uparrow bd] / [-ef] + g$
$[/+a\uparrow bd\ -ef] + g$
$+/+a\uparrow bd-efg$

iii) $a*[+bd] / e-f*(g+[/hk])$
$a*[+bd] / e-f*[+g/hk]$
$[*a+bd] / e-f*[+g/hk]$
$[*a+bd] / e-[*f+g/hk]$
$[/*a+bde] - [*f+g/hk]$
$-/*a+bde*f+g/hk$

Using Stack (Infix to Postfix)

Algorithm: POLISH (Q, P)
Suppose Q is an arithmetic exp. written in Infix notation. This algorithm finds the equivalent Postfix exp. P

1. PUSH "(" on to stack ")" to the end of Q
2. SCAN Q from Left to right and repeat steps 3 to 6 for each element of Q until the stack is empty.
3. If operand is encountered, add it to P.
4. If a "(" is encountered, PUSH it onto stack
5. If an operator "⊗" is encounted, then:
   a) Repeatedly POP from stack and add to P, each operator (on the top of stack) which has the same precedence as or higher precedence than ⊗
   b) Add ⊗ to stack.
6. If a ")" is encountered, then:
   a) Repeatedly POP from stack & add to P, each operator (on the top of stack), until a left parenthesis is encountered.
   b) Remove the "(" [Do not add the left parenthesis i.e "(" to P]
   [end of if struct]
   [end of step 2 loop]
7. Exit.

ab for operator encounter ... to higher priority wale to expression upar me, or same precedence wala nikla h tab, the upar wale ko hi expression me.

Que: Q. = a+b*(c-d)/e+f%g)

SS	Stack	Exp. P.
a	(	a
+	(+	a
b	(+	ab
*	(+*	ab
(	(+*(	ab
c	(+*(	abc
−	(+*(−	abc
d	(+*(−	abcd
)	(+*	abcd−
/	(+/	abcd−*
e	(+/	abcd−*e
+	(+	abcd−*e/+
f	(+	abcd−*e/+f
%	(+%	abcd−*e/+f
g	(+%	abcd−*e/+fg
)	−	abcd−*e/+fg%+

Agar upar k operator k priority zyadah to usko shift karenge expression me

Q. Q = A + (B*C − (D/E↑F)*G)*H

Q = (A+B)*(C*D/E%F)*(G/H)

Infix to Prefix using stack

Algorithm : INVERSEPOLISH

This algo. converts the Infix into Prefix.

1. Reverse the i/p string.
2. Examine the next element in the I/P string from left to right.
3. If it is operand, add it to O/P string.
4. If it is ")", PUSH it on stack
5. If it is an operator, then:
   a) If stack is empty, PUSH operator on stack.
   b) If top of the stack is ")", PUSH operator on stack.
   c) If it has same or higher priority then the top of stack, PUSH operator on stack.
   d) Else POP the operator from the stack and add it to O/P string

6. If it is ")", POP operator from stack and add them to O/P string until a "(" is encountered, POP and discard "(".

7. If ~~there~~ is more I/P, then: ~~go to step 2.~~

8. If there is no more I/P, then: unstack the remaining operator & add them to O/P string.

9. Reverse the O/P string.

10. Exit.

~~Q = a + b * (c - d) / e~~

Q = (A+B) * (C * D / E % F) * (G / H).

=) H / G (*) F % E / D * C (*) B + A (

SS.	Stack	Exp. O/P.
)	)	
H	)	H
/	) /	H
G	) /	H G
(	—	H G /
*	*	H G /
)	*)	H G /
F	*)	H G / F
%	*) %	H G / F
E	*) %	H G / F E
/	*) % /	H G / F E
D	*) % /	H G / F E D
*	*) % / *	H G / F E D
C	*) % / *	H G / F E D C
(	*	H G / F E D C * / %
*	* *	H G / F E D C * / %
)	* *)	H G / F E D C * / %
B	* *)	H G / F E D C * / % B
+	* *) +	H G / F E D C * / % B
A	* *) +	H G / F E D C * / % B A
(	* *	H G / F E D C * / % B A +
—	—	H G / F E D C * / % B A + * *

O/P = * * + A B % / * C D E F / G H

1st March. Evaluation of postfix exp.

This algo. finds the value of an arithmetic exp. P, written in postfix notation.

1. Add a ")" at the end of P.

2. Scan P from left to right & repeat step 3 & 4 for each element of P until the sentencial (" ")" is encountered.

3. If an operand is encountered, then:

PUSH in an stack.

4. If an operator & is encountered, then:

INDEX

Sr. No.	Date	Title	Page No.

Dr. Siddhartha Chaubey
Prof (CSE)
SSTC, Bhilai

DATE | | PAGE

Queue

A queue is a linear list of elements in which deletion of elements can take place only at one end called the "FRONT" and insertion can take place only at other end called the "REAR".

Queue is also called FIFO list. Since the first element in a queue will be the first element out of the queue. In other words, the order in which elements enter in a queue is the order in which they leave.

Queue are implemented in 2 ways-
i) array
ii) link list.

1) Using Array.

1. Insertion

Algorithm: QINSERT (FRONT, REAR, N, ITEM, QUEUE)

This algo. inserts an element into a queue, where queue is a linear array. N is the max^m size of array. FRONT and REAR are the variable which points first element and REAR points to last element.

1. [Queue is already filled?] If REAR = N, then:
write "OVERFLOW" & exit.

2. [Queue is empty?] If FRONT = 0, then:
Set FRONT := REAR := 1
Else
Set REAR := REAR + 1.
[end of If structure]

3. Set QUEUE[REAR] := ITEM [Copy the ITEM]

4. Exit.

2. Deletion

Algo: QDELETE (FRONT, REAR, N, ITEM, QUEUE)

This algo deletes an ITEM from a QUEUE and assign it to variable ITEM, where QUEUE is a linear array. N is max^m size of array. FRONT point the first element and REAR points last element.

1. [Empty QUEUE?] If FRONT = 0 or FRONT = REAR + 1, then:
write "underflow" & exit.

2. Set ITEM := QUEUE[FRONT] [assign front element to ITEM].

3. Set FRONT := FRONT + 1 [update FRONT]

4. Exit.

3. Traverse

Algo: QTRAVERSE (QUEUE, FRONT, REAR)

This algo traverses the queue where I is a counter variable. Initialised by the value of FRONT and traverses upto the REAR.

1. [Queue is empty?] If FRONT = 0, then:
write "Empty Queue" & exit.

2. Set I := FRONT [Initialize counter variable by FRONT]

3. Repeat steps 4 & 5 while I <= REAR.

4. Apply PROCESS to QUEUE[I]

5. Set I := I + 1 [update counter var I by 1]
[end of step 3 loop]

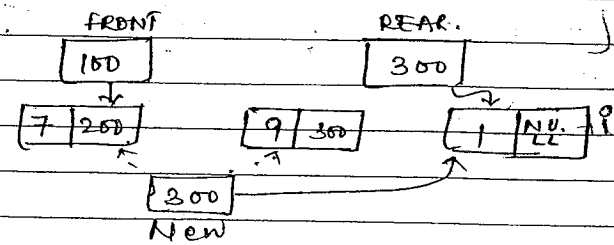
6. Exit

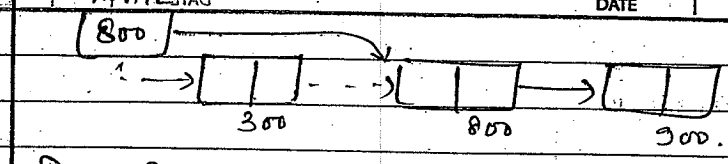
ii) Link representation of Queue

1. Insert

Algo: QINSERT (INFO, LINK, AVAIL, ITEM, FRONT, REAR).

1. [OVERFLOW?] If AVAIL = NULL, then:
   write "Overflow" & exit.
2. [Remove first node from AVAIL LIST]
   Set NEW := AVAIL & AVAIL := LINK[AVAIL]
3. Set INFO[NEW] := ITEM & LINK[NEW] := NULL.
   [Copy ITEM into NEW node]
4. If FRONT = NULL, then:
   Set FRONT := REAR := NEW [If Queue is empty, then ITEM is the first element in QUEUE]
   Else:
   Set LINK[REAR] := NEW & REAR := NEW
   [Now REAR points to the new node appended to the end of the list linked queue]
   [end of If structure]
5. Exit.



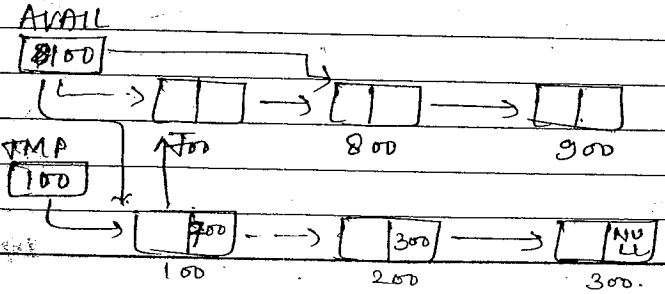


2. Deletion

Algo: LINK_Q_DELETE (INFO, LINK, AVAIL, ITEM, FRONT, REAR)

This algo. deletes the FRONT element of the linked queue and store it in item.

1. [Underflow?] If FRONT = NULL, then:
   write "Overflow" & exit.
2. Set TMP := FRONT [If link list is not empty, remember FRONT in temp. var]
3. Set ITEM := INFO[FRONT]
4. Set FRONT := LINK[FRONT] [Reset FRONT to point to the next element in the queue].
5. [Return deleted node TMP to the AVAIL LIST]
   Set LINK[TMP] := AVAIL & AVAIL := TMP.
6. EXIT.

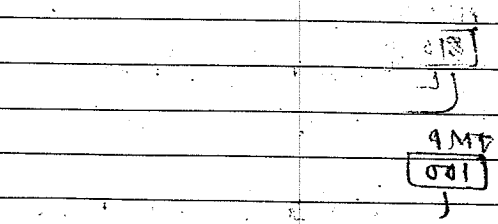


3. Traverse

Algo: LINK-Q-TRAVERSE(INFO, LINK, FRONT).

This algo. traverses the queue & apply PROCESS to the queue.

1. [Empty Queue?] If FRONT = NULL, then: write "Empty Queue" & exit.
2. Set PTR := FRONT [Initialize pointer]
3. Repeat steps 4 & 5 while PTR ≠ NULL
4. Apply PROCESS TO INFO[PTR].
5. Set PTR := LINK[PTR] [update pointer]
   [end of step 3 loop]
6. Exit.



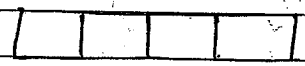
4th March

Circular Queue

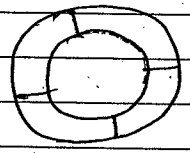
In a simple queue, when the rear pointer reaches at the end, insertion will be denied even if space is available at the front. One way to remove this disadvantage or drawback is by using circular queue.

Circular queue is same as ordinary queue but logically it implies that a[1] comes after a[n]

Physical representation.



Logical representation.



1. Insertion

Algo: CIR_Q_INSERT(FRONT, REAR, ITEM, QUEUE, N)

This algo. inserts an element ITEM into a circular queue.

[Is Queue is already filled?] If FRONT = REAR + 1 OR REAR = N

FRONT = REAR + 1
write "Overflow" & Exit.

2. [Find new value of REAR]
If FRONT = 0, then:
Set FRONT := REAR := 1
[Q is initially empty]

Else if: REAR = N, then:
Set REAR = 1

Else
Set REAR = REAR + 1

[end of if structure]

3. QUEUE[REAR] := ITEM [copy to ITEM]

4. EXIT.

2. Deletion

Algo: CIR-Q-DELETE (FRONT, REAR, ITEM, N QUEUE)

This algo. deletes an ITEM from a C. queue & assign it to ITEM.

1. [Empty queue?] If FRONT = 0, then:
write "underflow" & exit.

2. Set ITEM := QUEUE[FRONT] [copy ITEM]

3. [Find new value of FRONT]
If FRONT = REAR, then:
Set FRONT := REAR := 0. [Queue has only one element]

Else if: FRONT = N
Set FRONT := 1

Else:
Set FRONT := FRONT + 1

3. Exit.

3. Traverse

Algo: CLR_Q-TRAVERSE (FRONT, REAR, QUEUE, N)

This algo. TRAVERSE the C. queue, where I is a counter variable initialized by FRONT.

1. [Empty Queue?] If FRONT = 0, then write "Empty Queue" & exit.

2. Case I: If FRONT <= REAR, then:
Set I := FRONT [Initialize I]
a) Repeat b) & c) while I <= REAR
b) Apply PROCESS to QUEUE[I]
c) [Update I] I := I + 1
[end of loop]
[end of Case I]

3. Case II : If FRONT > REAR, then:
Set I := FRONT [Initialize I]
a) Repeat b) & c) while I <= N
b) Apply PROCESS to QUEUE[I]
c) [update I] I := I+1
[end of loop]
[end of case II]
Set I := 1 [Initialize I by 1]
a) Repeat b) & c) while I <= REAR
b) Apply PROCESS to QUEUE[I]
c) [update I] I := I+1
[end of loop]
[end of case II]

4. EXIT.

Double Ended Queue (De-Queue)

A De-Queue is a linear list in which element can be added or removed at either end but not from the middle.

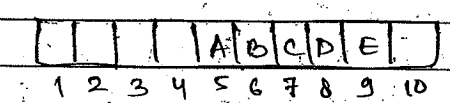
There are various ways of representing a De-Queue but mostly De-Queue is maintained by cir. array.

De-queue are represented by 2 pointers LEFT & RIGHT, which points to two ends of De-queue. Therefore

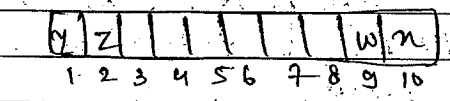
There are 2 variations of De-queue.
i) I/P restricted Dequeue.
ii) O/P ——— n ———.

i) I/P restricted De-queue - It is a De-que which allows insertion at only one end of the list but allow deletion at both ends of list

ii) O/P restricted De-queue - It is a D-Q. which allows deletion at only one end of the list but allows insertion at both end.



LEFT=5 RIGHT=9



LEFT=9 RIGHT=2.

Priority Queue

It is a collection of elements such that each element has been assigned a priority and such that, the order in which the element are deleted & process comes from the following rules.

i) An element higher priority is processed before any element of lower priority.
ii) Two elements with same priority are processed according to the order in which

they were added to the queue.

Representation of priority queue.
There are various ways on maintaining a p.queue in m/o.

i) One way list (Link list)
ii) Multiple queues (array).

i) One way list of P.Q.

To maintain a P.Q in m/o, by means of a one way list is -
a) Each node in the list will contain 3 items of information, an information field INFO, a priority no. PRN, & a link LINK. a node
b) A node X precedes Y in the list
→ When X has higher priority than Y
→ When both have same priority but X was added to the list before Y.

1. Deletion

Algo: DEL-P-Q (ITEM, INFO, START)
This algo. deletes and process the first element in a P.Q. which appears in m/o as a One way list.

1. Set ITEM := INFO[START] [This saves the data of the first node]
2. Delete first node from the list.
3. PROCESS ITEM.
4. Exit.

2. Insertion

Algo: INS-P-Q (ITEM, N)
This algo. adds an ITEM with priority no. N to a P.Q. which is maintained in m/o as a one way list.

1. Traverse the one way list until finding a node X whose priority no. exceeds N. Insert ITEM in front of node X.
2. If no such node is found, insert an ITEM as the last element of the list.
3. Exit.

5th March

ii) Multiple Queue

Another way to maintain a priority queue is to use a separate queue for each level of priority. Each such queue will appear in its own circular array & must have its own pair of pointer front and rear.

If each queue is allocated, the same amount of space, the 2-D array can be used instead of linear array.

	FRONT
1	2
2	1
3	0
4	5
5	4

	REAR
1	2
2	3
3	0
4	1
5	4

Priority	1	2	3	4	5	6
1	A					
2	B	C	K			
3						
4	F				D	E
5				G		

in Priority queue

Algo. for deleting & inserting element that is maintained in m/o by a 2-D array QUEUE

i) Deletion

Algo:
This algo. deletes and processes the first element in P.Q. maintained by a 2-D. array QUEUE

1. [Find the first non-empty QUEUE]
Find the the smallest K such that FRONT[K] ≠ NULL

2. Delete and process the FRONT element in row K of QUEUE.
3. EXIT.

ii) Insertion

Algo:
This algo. adds an ITEM with priority no. N to a P.Q maintained by a 2-D array QUEUE.

1. Insert ITEM at the rear element in row M of QUEUE.
2. EXIT.

DATE 26/3/14 PAGE

HASHING

We have seen different searching techniques, where each time is basically dependent on the no. of elements. Sequential, binary and all the search trees are totally dependent on no. of elements and so many key comparisons are also involved.

Now our need is to search the element in constant time and less key comparison should be involved.

Suppose all the elements are in array size N. Let us take all the keys are unique and in the range 0 to N-1. Now we are storing the records based on the key where the array index & key are same. Now we can access the record in constant time and there are no key comparison are involved.

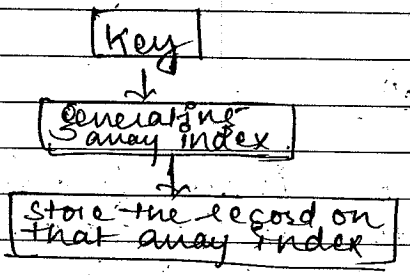
Eg: Let us take 5 records where keys are 9, 4, 6, 7, 2, it will be stored in array as

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

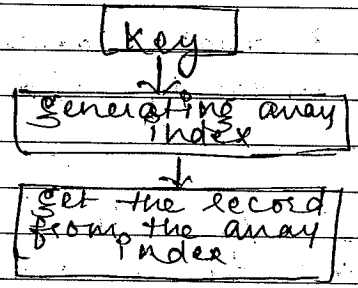
Here we can see the record which has key value 2 can be directly accessed through array index a[2]

Now the idea that comes in picture is hashing where we will convert the key into array index and put the records in array. & in the same way for searching the record, convert the key into array index & get the record from array.

For storing

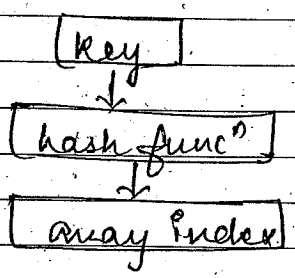


For searching/retrieving



The generation of array index uses hash func^n, which converts the keys into array index and the array which supports hashing for storing &

searching record is called hash table.



So we can say each key is mapped on a particular array index through hash func^n.

Hash func^n

Hash func^n is a func^n which, when applied to the key produce an integer which can be used as an address in a hash table.

The intend is that elements will be relatively randomly and uniformly distributed.

Perfect hash func^n is a func^n which, when applied to all the members of set of items to be stored in a hash table, produces a unique set of integers within some suitable range. Such func^n produce no collision

Good hash func^n minimizes collision by spreading the elements uniformly throughout the array.

The two principle criteria used in selecting a hash func^n H : K → L are as follows -

i) The func^n H should be very easy and quick to compute.

ii) The func^n H should, as far as possible uniformly distribute the hash addresses throughout the set L so that there are min. no. of collisions.

There are basically 3 types of hash func^n -

1. Mid Square Method

★ PHOTOCOPY //

28th March

iii) Double hashing

In this the increament factor is not constant as in linear & quadiatic probing, but it depends on the key. The increament factor is another hash func^n and hence the name double hashing.

The formula for double hashing can be written as -

$$H(K,i) = (h(K) + ih'(K)) \% m$$

m is size of table.

Where the 2nd hash func^n H' is used for resolving a collision. Suppose a record R with key 'k' has the hash address

$H(K) = h$ (not equal) m

$H'(K) = h' \neq m$

then we linearly search the loc^n with address $h, h+h', h+2h', h+3h', \ldots$

for eg consider the inserting the keys 46, 28, 21, 35, 57, 39, 19, 50 into a hash table of size m=11 using double hashing

Consider that the hash func^n are -

$h(K) = K \% m$

$h'(K) = 7 - (\text{Key} \% 7)$

Hash table

0	1	2	3	4	5	6	7	8	9	10
		46				28	28	57	35	21

1. Insert 46:

$H(46,0) = (46 \% 11 + 0(7 - (46 \% 7))) \% 11$

$= 2$

T[2] is empty

2. Insert 28:

$H(28,0) = (28 \% 11 + 0(7 - (28 \% 7))) \% 11$

$= 6$

T[6] is empty.

3. Insert ~~21~~ 21

$H(21,0) = (21 \% 11 + 0(7 - (21 \% 7))) \% 11$

$= 10$

T[2] is not empty.

4. Insert 35

$H(35,1) = (35 \% 11 + 1(7 - (35 \% 7))) \% 11$

$= 9$

T[7] is empty.

5. Insert 57

$H(57,0) = (57 \% 11 + 0(7 - (57 \% 7))) \% 11$

$=$

2. Separate chaining

→ In this method, l.l are maintained for elements that have same hash addresses

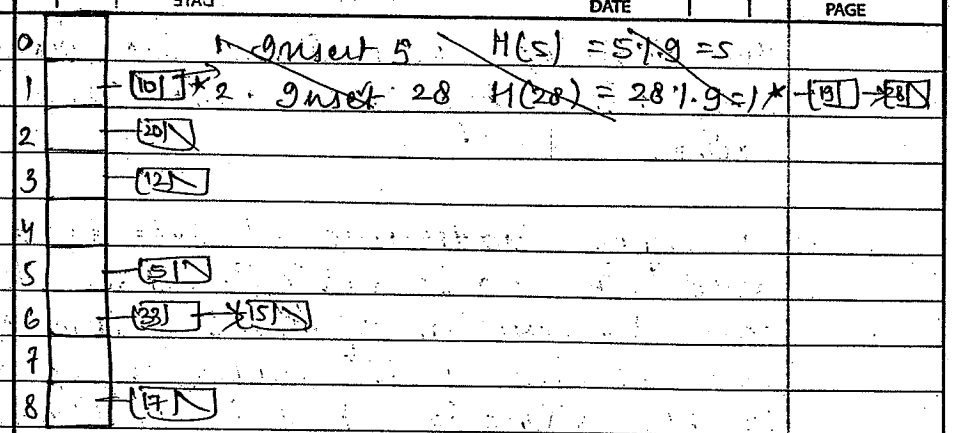
→ Here the hash table does not contain actual keys and records but it is just an array of pointers, where each pointer points to a linked list.

→ All elements having same hash addres i' will be stored in a separate linked list & the starting address of the linked list will be stored in the index 'i' of the hash table.

→ So, array index 'i' of the hash table contains a pointer to the list of all elements that share hash address.

→ This linked list are refered to as chain hence, the method is named as separate chaining.

eg Let us consider the insertion of elements 5, 28, 19, 15, 20, 33, 12, 17, 10 into a chained hash table.
Let us suppose hash table has 9 slots and the hash func is $H(K) = K \pmod 9$.



1. Insert 5 → H(5) = 5 % 9 = 5
2. 28 → H(28) = 28 % 9 = 1
3. 19 → H(19) = 19 % 9 = 1
4. 15 → H(15) = 15 % 9 = 6
5. 20 → H(20) = 20 % 9 = 2
6. 33 → H(33) = 33 % 9 = 6
7. 12 → H(12) = 12 % 9 = 3
8. 17 → H(17) = 17 % 9 = 8
9. 10 → H(10) = 10 % 9 = 1

* The hash table is made of array of pointers.

Difference b/w open addressing and separate chaining

1. In open addressing, accessing any record involves comparison with key which have diff hash values which increases the no of probes

In chaining comparisons are done only with keys that have same hash value.

2. In open addressing all records are stored in hash table, itself, so there can be problem of hash table overflow and to avoid this, enough space has to be allocated at the compilation time.

In separate chaining, there will be no problem hash table overflow because linkage dynamically allocated so, there is no limitation on the no. of records that can be inserted.

3. Separate chaining is best suited for applications where the no. of records is not known in advance.

4. In open addressing, it is best if some locations are always empty.

In separate chaining there is no wastage of space cos the space for records is allocated when they arrive.

5. Implementation of insertion & deletion is simple is separate chaining but complex in open addressing.

6. In separate chaining, the load factor denotes the avg no. of elements in each list & it can be greater than one.

In open addressing load factor is always less than one.

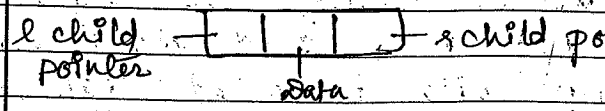
Load factor

UNIT - III

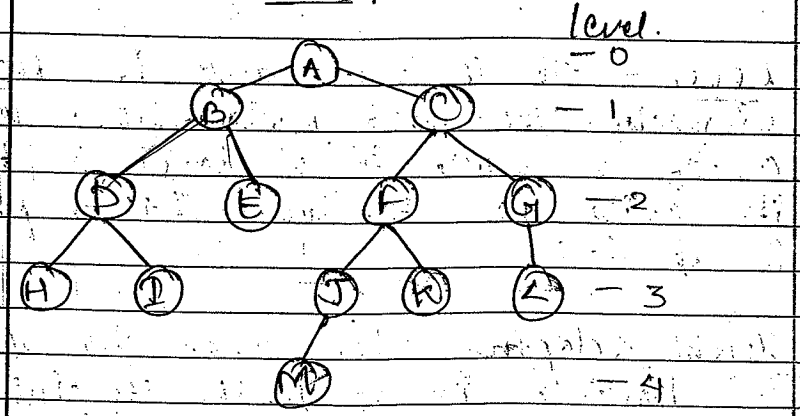
TREE

1. Basic Terminologies

1. **Node** - This is the main component of any tree structure. A node of a tree stores the actual data and links to the other node.



Node



2. **Parent** - The parent of a node is the immediate predecessor of a node
eg. B is the parent of E.

3. **Child** - If the immediate predecessor is the parent of the node then all immediate successor of a node are called child.
eg. B & C are child of A.

4. Branch / edge (link) - This is a pointer to a node in a tree.

5. Root - This is a specially designated node which has no parent.
eg. A is the root.

6. Leaf - (external node) - The node which is at the end and does not have any child is called leaf node.
eg. H, I, E, M, K, L are leaf.

7. Level - It is the rank in the hierarchy. The root node have level 0. If a node is at level l then its child is at level l+1 & parent is at l-1.

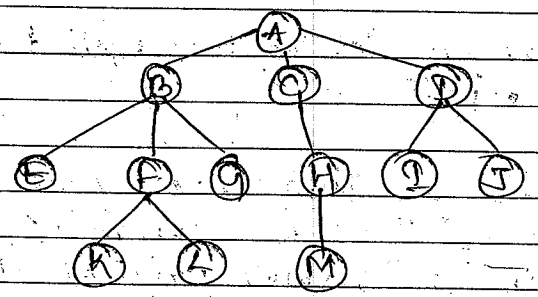
8. Height & depth - The max no. of nodes i.e possible in a path starting from the root node to a leaf is called the height of a tree.
Generally height = level + 1.
eg here h=5.

9. Degree - The max no. of children i.e ~~is possible~~ possible for a node is known as degree of a node.
eg. Degree of B=2 C=2 J=1.
G=1 A=2 M=0 so on...
D=2 I=0 E=0 ...

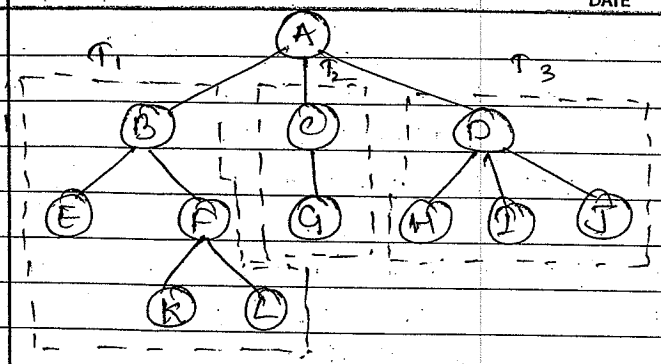
Degree of a tree is = max^m degree of any node of the tree.
here max degree is 2.
then degree of tree = 2.

10. Sibling - The node which have the same parent are called sibling.
eg. B & C are siblings.

Q. Observe the following tree & find
1. height 2. level of H, C, K 3. Degree of tree
4. longest path 5. Parent of M & Sibling of I
7. Child of B.



Tree - A tree is a finite set of one or ~~node~~ more node such that -
i) There is a specially designated node called root.
ii) The remaining nodes are partitioned into n (n>0) disjoint sets $T_1, T_2, \ldots, T_n$, where each T_i ($i = 1 \ldots n$) is a tree; $T_1, T_2, \ldots T_n$ are called ~~are~~ subtree of the root.



Sample tree T.

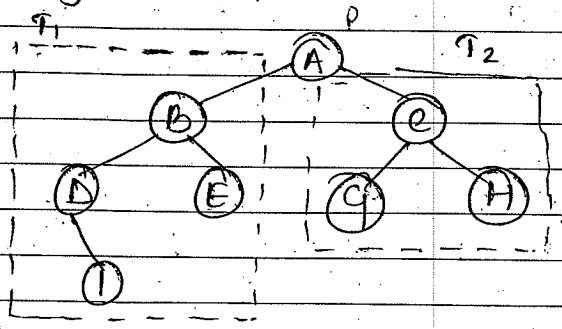
30th March

(2) Binary Tree T

A Binary tree is defined as a finite set of elements called nodes such that,

i) T is empty (called the null tree or empty tree)

ii) T contains a distinguished node R, called the root of T & the remaining nodes of T form a ordered pair of disjoint binary tree T_1 & T_2.



max^m	no. of nodes at any level.
2^0	
2^1	
2^2	
2^3	

21

bell

In binary tree the degree can be zero but not ~~not~~ exceed the degree of two.

1. Properties

1. In any binary tree, the max^m no. of nodes on the level 'l' is 2^l, $l \geq 0$

2. The max^m no. of nodes possible in a binary tree of height 'h' is $2^{h+1}-1$

3. The min^m no. of nodes possible in a binary tree of height 'h' is 'h'.

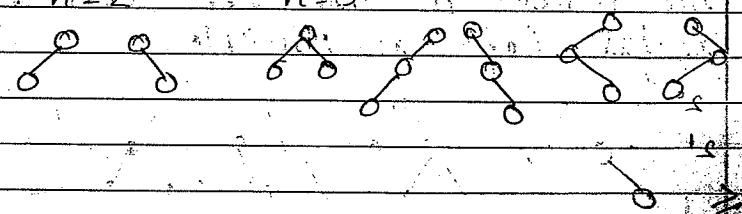
4. For any non-empty binary tree, if 'n' is the no. of nodes and 'e' is the no. of edges then $n = e+1$

5. For any non empty binary tree T, if n_0 is the no. of leaf nodes and n_2 is the no. of internal nodes (degree $=2$), then $n_0 = n_2 + 1$

6. The height of a complete binary with 'n' no. of nodes is $\lceil \log_2(n+1) \rceil$

7. The total no. of binary trees possible with 'n' nodes is $\frac{1}{n+1} {}^{2n}C_n$

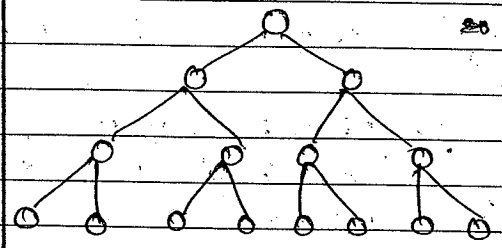
eg: $n=2$ $n=3$



Full Binary Tree

A binary tree is a full binary tree if it contains the max^m possible no. of nodes at all levels.

Every level can contain max 2^l no of nodes where l = level

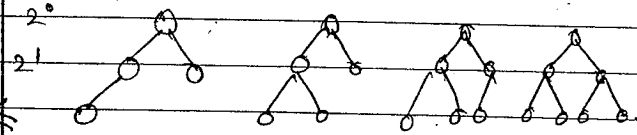


The above tree is full binary tree coz all level contain 2^l nodes

The tree can contain max^m $2^h - 1$ or $2^d - 1$ node, h = height d = depth.

(1) Complete binary Tree

A binary tree is said to be a complete binary tree if all its levels, accept possibly the last level, have the max^m no. of possible nodes and all the nodes in the last level appear as far left as possible.



The depth 'd_n' of the complete binary tree 'T_n' with n nodes is given by

$d_n = \lfloor \log_2 n + 1 \rfloor$

(2) Representation of Binary Tree

A binary tree must represent a hierarchical r/s b/w a parent node and (atmost 2) child nodes.

There are 2 common methods for the representation of binary tree.-

i) By means of array.
ii) ~~n~~ l.l.

i) Linear / Array representation

This type of representation is static in the sense that a block of m/o for an array is allocated before storing the actual tree in it, and once the m/o is allocated, the size of the tree is restricted.

Following rules can be used to decide the loc^n of any node of a tree in the array.

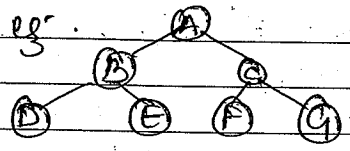
a) The root node is at loc^n 1.
b) For any node with index i, ~~one less~~ $1 \le i \le n$ (for some n)
→ Parent (i) = $\lfloor i/2 \rfloor$ for the nodes.

when $i = 1$, there is no parent

→ L·child $(i) = 2*i$, if $2*i > n$ then i has no l·child.

→ R·child $(i) = 2*i + 1$, if $2*i+1 > n$ then i has no R·child.

eg.



A	B	C	D	E	F	G
1	2	3	4	5	6	7

ii) L·k representation / Dynamic

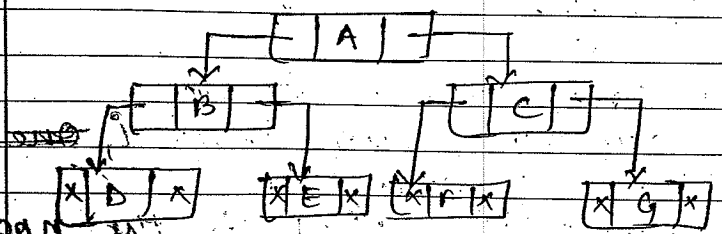
Consider a binary tree T. T will be maintained in m/b by means of a linked representation which uses 3 parallel arrays, INFO, LEFT & RIGHT and a pointer variable ROOT.

First of all each node N of T correspond to a loc^n K such that -

→ INFO[K] contains the data at the node N

→ LEFT[K] contains the loc^n of left child of node N.

→ RIGHT[K] contains the loc^n of right child of node N.



ROOT = 5

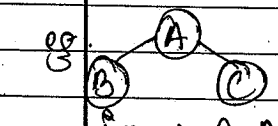
AVAIL = 6

	DATA	LEFT	RIGHT
1	G	0	0
2	D	0	0
3	C	10	1
4		0	
5	A	7	3
6		8	
7	B	2	9
8		9	
9	E	0	0
10	F	0	0

i) Traversing :-

1. Inorder : L V R
2. Pre order : V L R
3. Post order : L R V

eg.

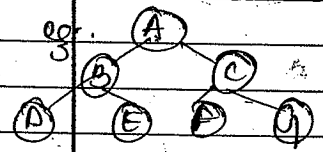


In → B A C

Pre → A B C

Post → B C A

eg.

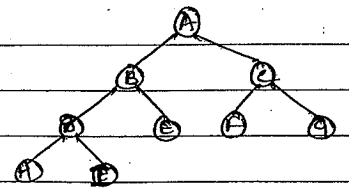


In → D B E A F C G

Pre → A B D E C F G

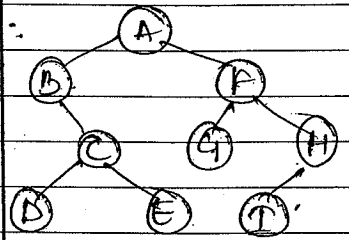
Post → D E B F G C A

eg



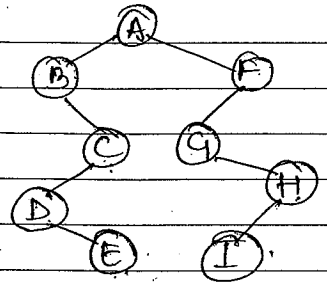
in → ~~H E D~~ ~~H D E~~ H D I B E A F C G.
Pre → A B D H I E C F G
Post → H I D E B F G C A

eg:



in → ~~D C E B A G F I H~~. B D C E A G F I H
pre → A B C D E F G H I.
post → D E C B G I H F A

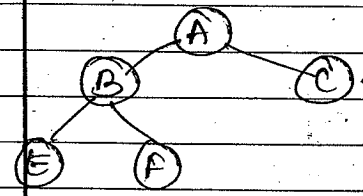
eg.



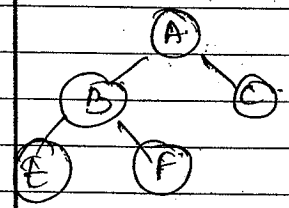
in → ~~B D E~~ B D E C A G I H F ~~A~~
pre → A B C D E F G H I
post → ~~A~~ E D C B I H G F A

eg 2nd forming tree with the help of given order.

eg · Pre → ~~A B E F C~~ L → R scan.
In → E B F A C

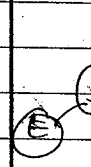


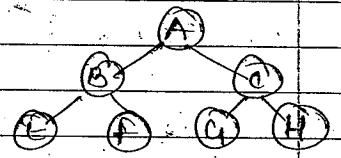
eg · ~~Post → E F B C D~~ L ← R ~~→~~ scan
in → E B F A C



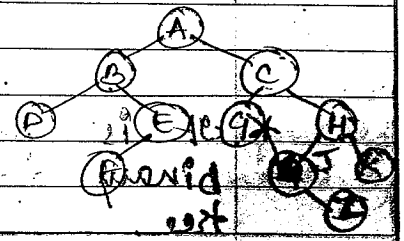
eg · Pre → ~~A B E F C D G H~~
in → E B F A G C H

Post → ~~E F B G H C A~~
in → E B F A G C H

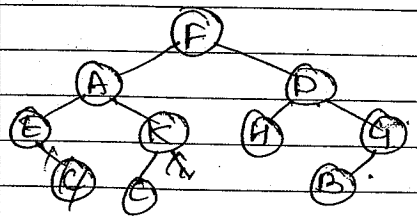




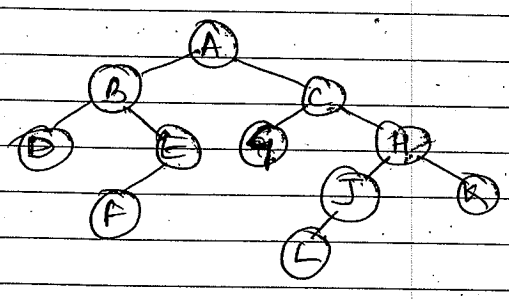
eg · pre → ~~A B D E F G~~ H J K K
in → D B F E A G C L J H K.



eg. Pre → FAEKCDHGB
In → EACKFHDBG

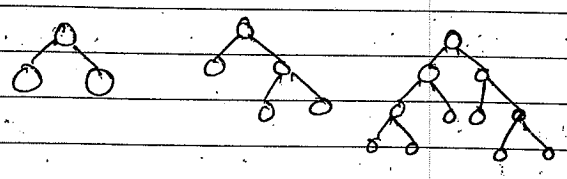


eg. post → CKEAHBGDF
In → EACKFHDBG



Strictly Binary Tree.
A binary tree is a strictly binary tree if each node in the tree is either a leaf node or has exactly two children i.e. there is no node with one children

eg.



It is not necessary that a strictly binary tree will be complete binary tree.

The given eg are strictly binary tree coz they each node have either 0 or two children

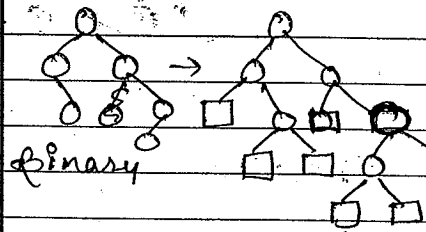
Property

→ A strictly binary tree with 'n' non-leaf nodes has n+1 leaf nodes.
→ A strictly binary tree with 'n' leaf nodes always has 2n-1 nodes.

(8) Extended Binary Tree
If in a binary tree, each empty sub tree (null link) is replaced by a special node then the resulting tree is extended binary. So we can convert a binary tree to an extended binary tree by adding special node to leaf nodes & nodes that have only one child.

The special nodes added to the tree are called external nodes and the original nodes of the tree are internal nodes.

eg



* External node = Internal node + 1

External nodes are shown by squares & internal nodes by circle.

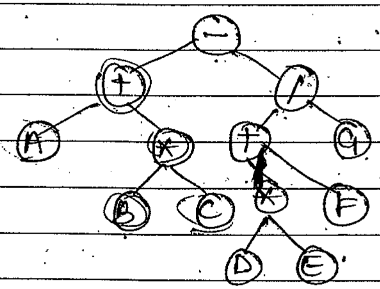
*IRV B-Tree

Expression Tree

An exp. tree is a binary tree which stores an arithmetic exp. The leaf of an exp. tree are operands, such as constants or variable names, and all internal nodes are the operators.

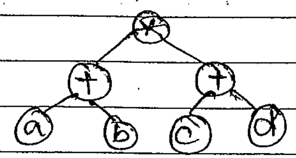
An exp-tree is always a binary coz arithmetic exp. contains either binary operators or unary operator.

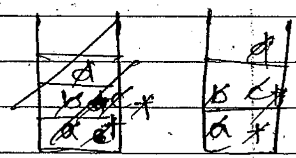
eg. (A+B*C) = ((D*E+F) / G)



eg (a+b) * (C+d)

Post → ab+cd+*

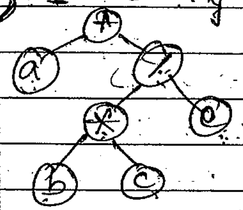


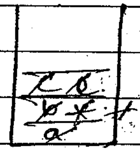


Steps for forming exp. tree.

Postfix — infix — prefix.

eg abc*e/+



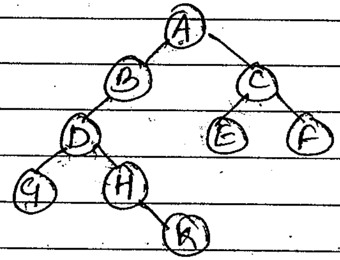


Infix (inorder of tree) → a+b*c/e

Prefix (preorder —n) → +a/*bce

Imp: Tree traversing using stack

i. Preorder using stack.



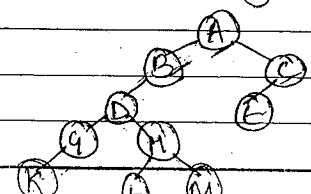
★ Algorithm from book

4	
3	A K
2	C F
1	NULL

Pre – ABDGHKCEF Null

Soln ABDGHKCEF.

ii. Inorder using stack



Soln – KG DLHMBAEC.

★ Algorithm from book.

6	K
5	G L
4	H M
3	B E
2	A C
1	NULL

* Correction

6) (b) go to step 2.

PTR = A B D G K null

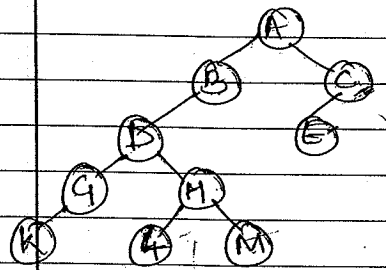
= K G B H M null

= K G M null

= M H A C E null

= E C null

iii. Postorder using stack



Soln. K G L M H D B E C A.

* Algorithm from book

9	
8	K K
7	G M M
6	H H
5	B
4	B E
3	H C
2	A
1	NULL

PTR = A B D G K null

= K G H L null

= L M null

= M H D B C E null

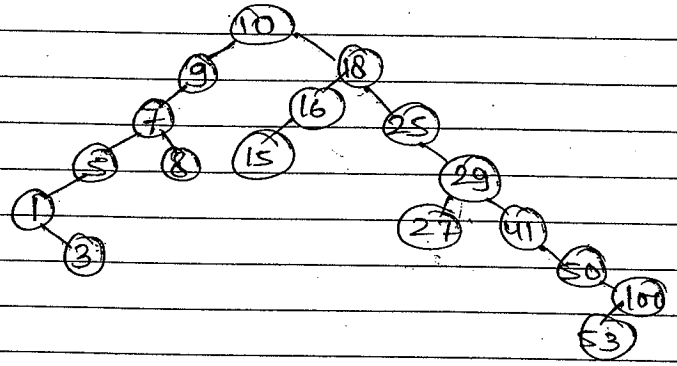
= G C A null.

Binary Search Tree

Suppose 'T' is a binary tree, then T is called a binary search tree if each node 'N' of 'T' has the following properties:-

i) The value at N is greater than every value in the left subtree of N.
ii) And is less than every value in the right sub tree of N.

eg. 10, 18, 9, 7, 25, 8, 29, 16, 1, 3, 27, 8, 15, 41, 50, 100, 53.



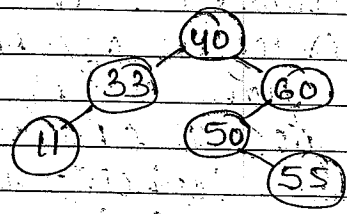
Searching & Inserting in binary Search Tree

Suppose an 'ITEM' of info. is given. The following algo. finds the loc^n of ITEM in the BST 'T' or inserts ITEM as a new node in its appropriate place in the tree.

a) Compare ITEM with the root node 'N' of the tree.
   i) If ITEM < N, proceed to the L.Child of N
   ii) If ITEM > N, proceed to the R.Child

b) Repeat step (a) until one of the following occurs.
   i) We meet a node N such that ITEM = N. In this case the search is successful.
   ii) We meet an empty subtree, which indicates that the search is unsuccessful & we insert ITEM in place of empty subtree.

Q. Draw a binary tree for the given series 40, 60, 50, 33, 55, 11.



* write each step.

Deletion

Suppose 'T' is a BST & let ITEM of info is given. The following algo. deletes the ITEM from T.

i) The deletion algo. first finds the loc^n

of node N which contains ITEM & also the loc^n of parent node P(N). The way 'N' is deleted from the T depends primarily on the no. of children of node 'N'.

There are 3 cases -

(I) N has no children. Then N is deleted from T by simply replacing the loc^n of N in the parent node P(N) by the null pointer.

(II) N has exactly one child. Then N is deleted from T by simply replacing the loc^n of N in P(N) by the loc^n of the only child of N.

(III) N has two children. Let S(N) denote the inorder succession of N. Then N is deleted from T by simply first deleting S(N) from T (by using case I & II) and then replacing node N in T by the node S(N).

AVL Tree (Balance Tree)

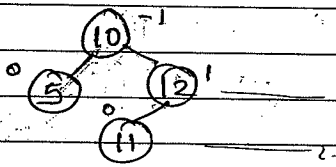
AVL - Adelson - Velskii and Landis.

An empty binary tree is an AVL Tree. A non-empty binary tree 'T' is an AVL tree if and only if given T^L & T^R to be the left and right subtrees of T & $h(T^L)$ & $h(T^R)$ to be the heights of subtrees T^L & T^R resp, T^L & T^R are AVL trees and $|h(T^L) - h(T^R)| \leq 1$ $(-1, 0, 1)$

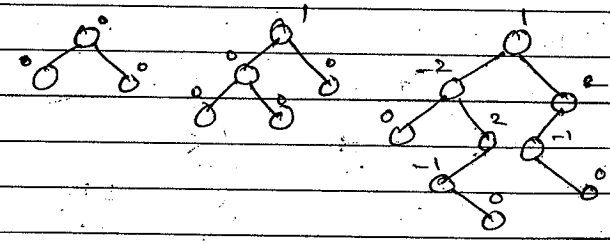
$h(T^L) - h(T^R)$ is known as balance factor. and for an AVL tree the balance factor of a node can be either 0, 1 or -1.

An AVL Search Tree is a BST which is an AVL tree.

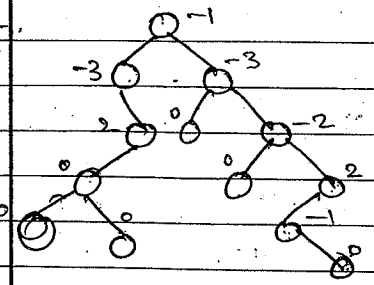
eg:



eg:-



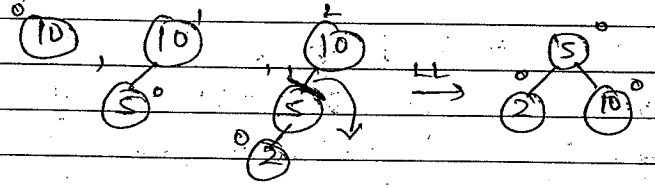
eg.



Rotations:

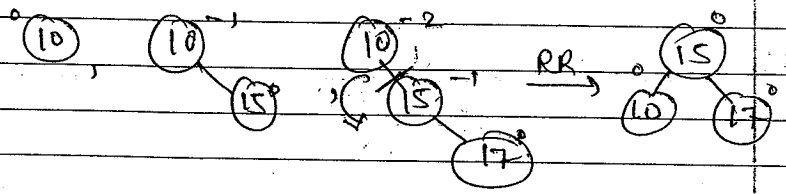
1. LL (Left left)

eg 10, 5, 2.



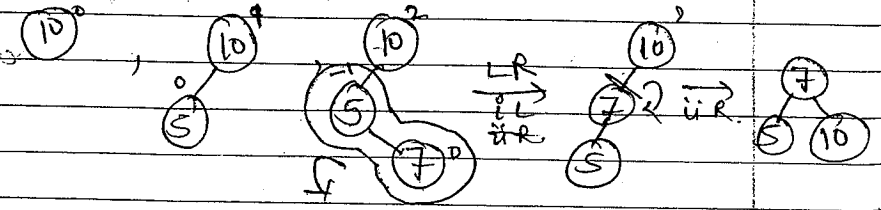
2. RR:

eg. 10, 15, 17.



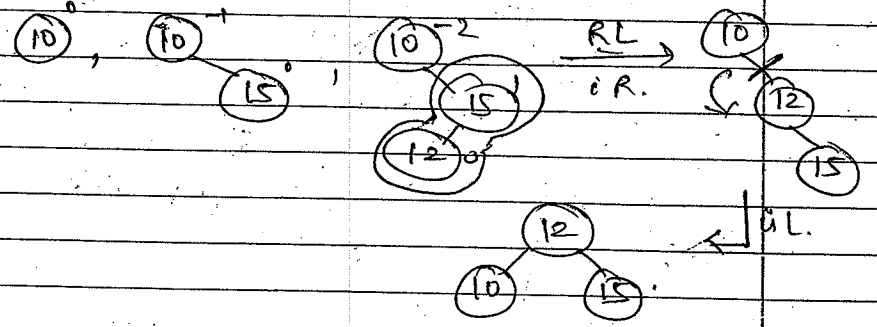
3. LR

eg. 10, 5, 7.

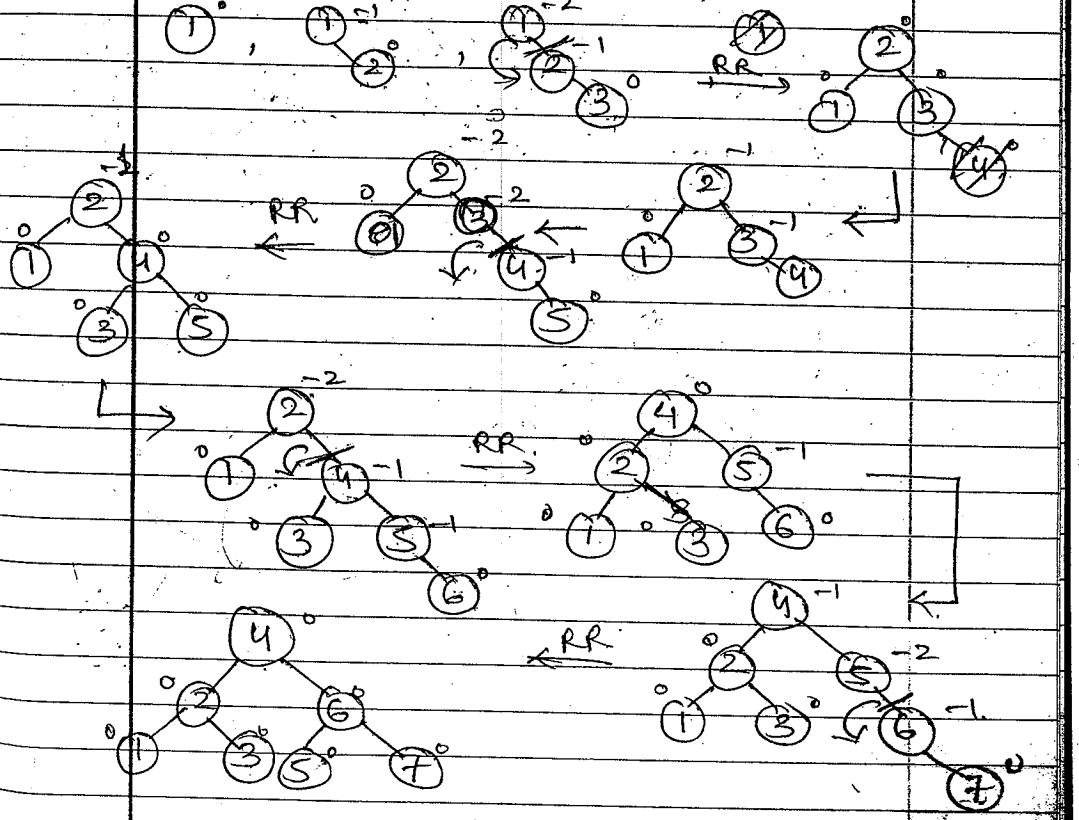


4. RL

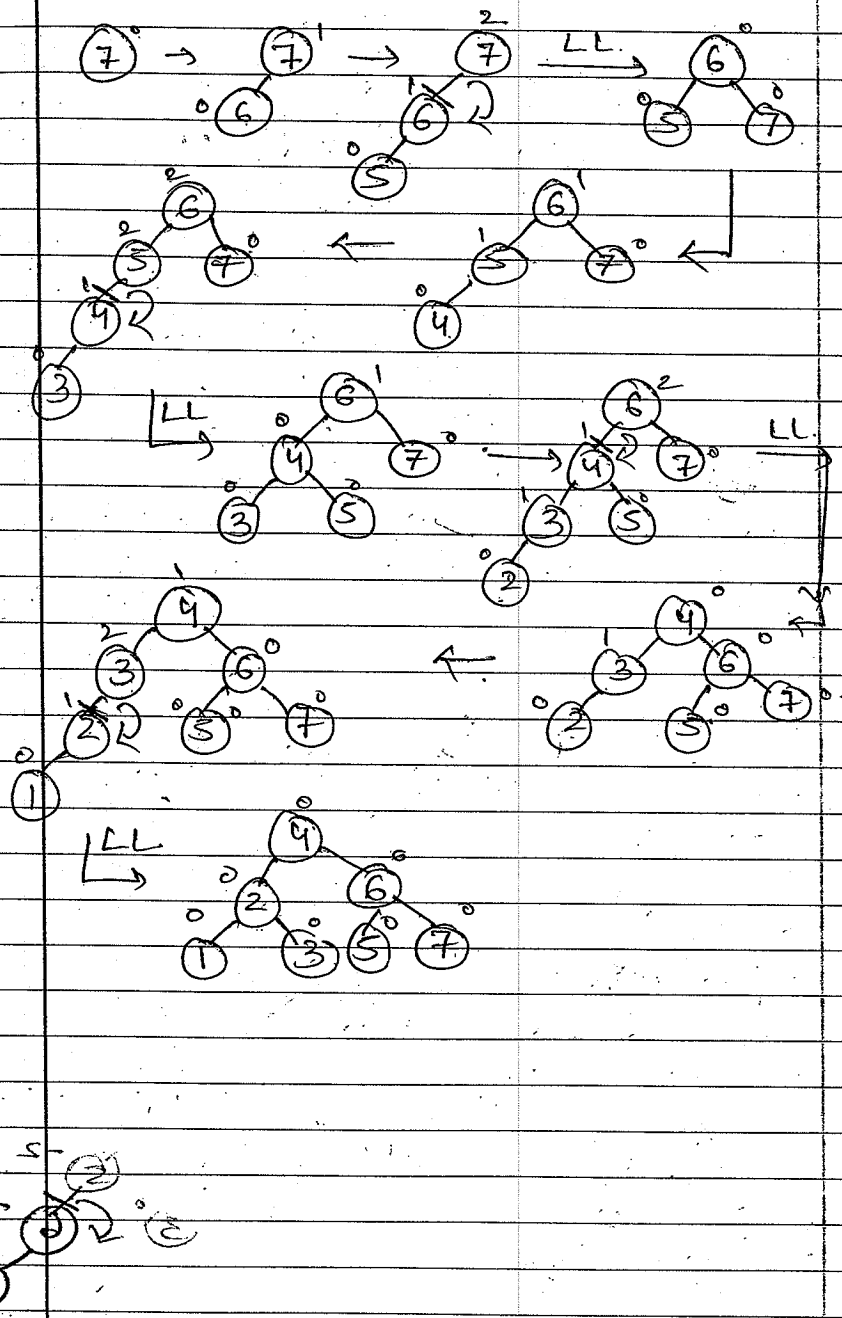
eg. 10, 15, 12.



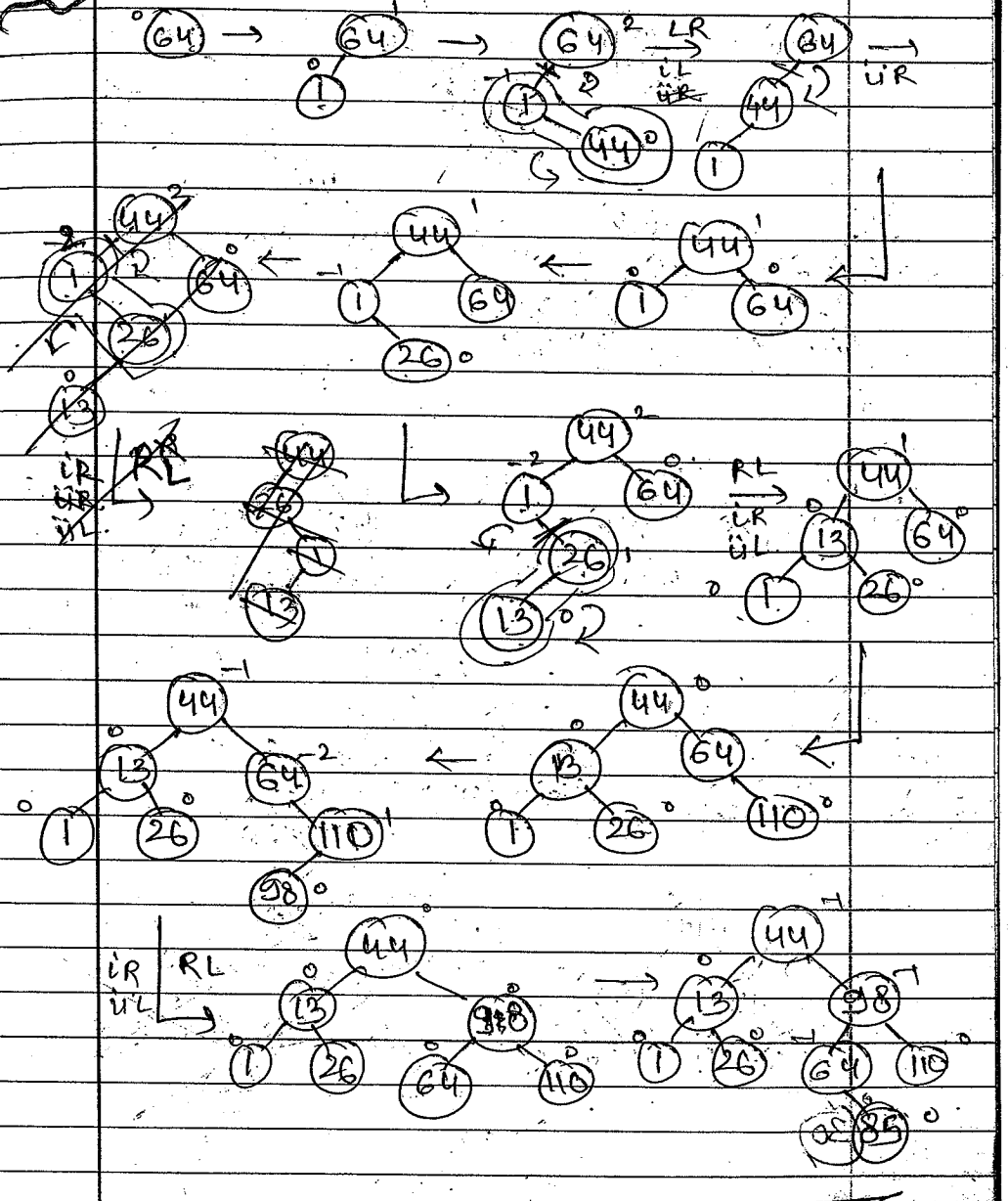
Q. 1, 2, 3, 4, 5, 6, 7.



Q. 7, 6, 5, 4, 3, 2, 1.

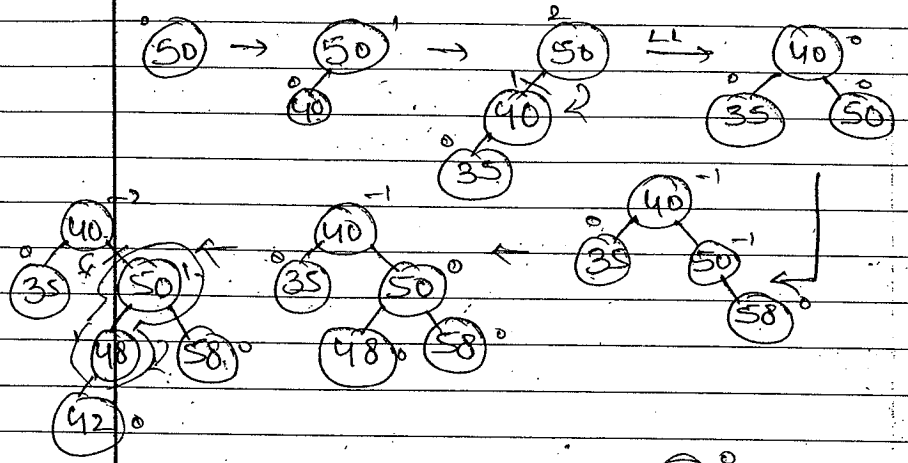


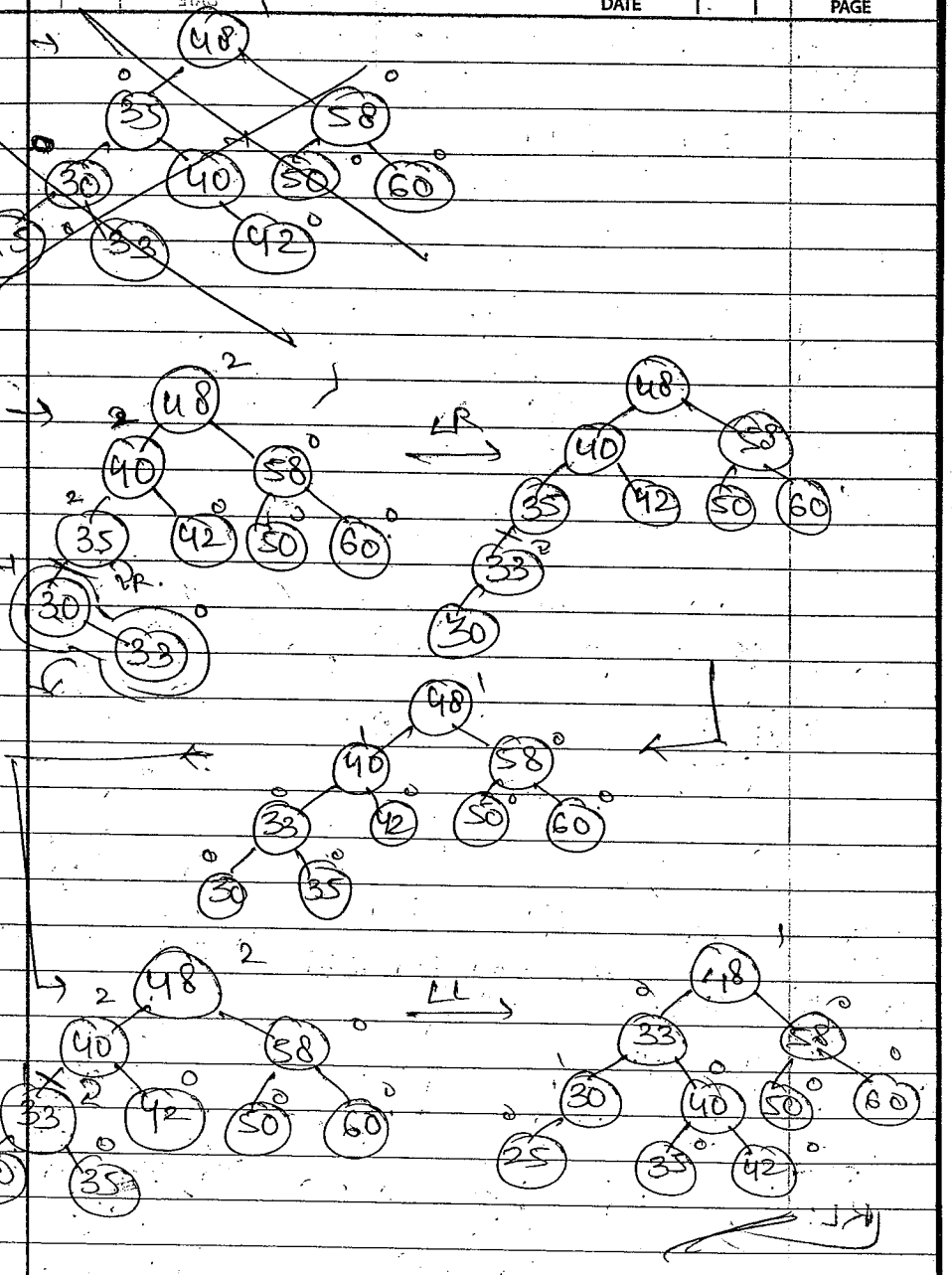
Q. 64, 1, 44, 26, 13, 110, 98, 85.



DATE | | PAGE

Q. 50, 40, 35, 58, 48, 42, 60, 30, 33, 25



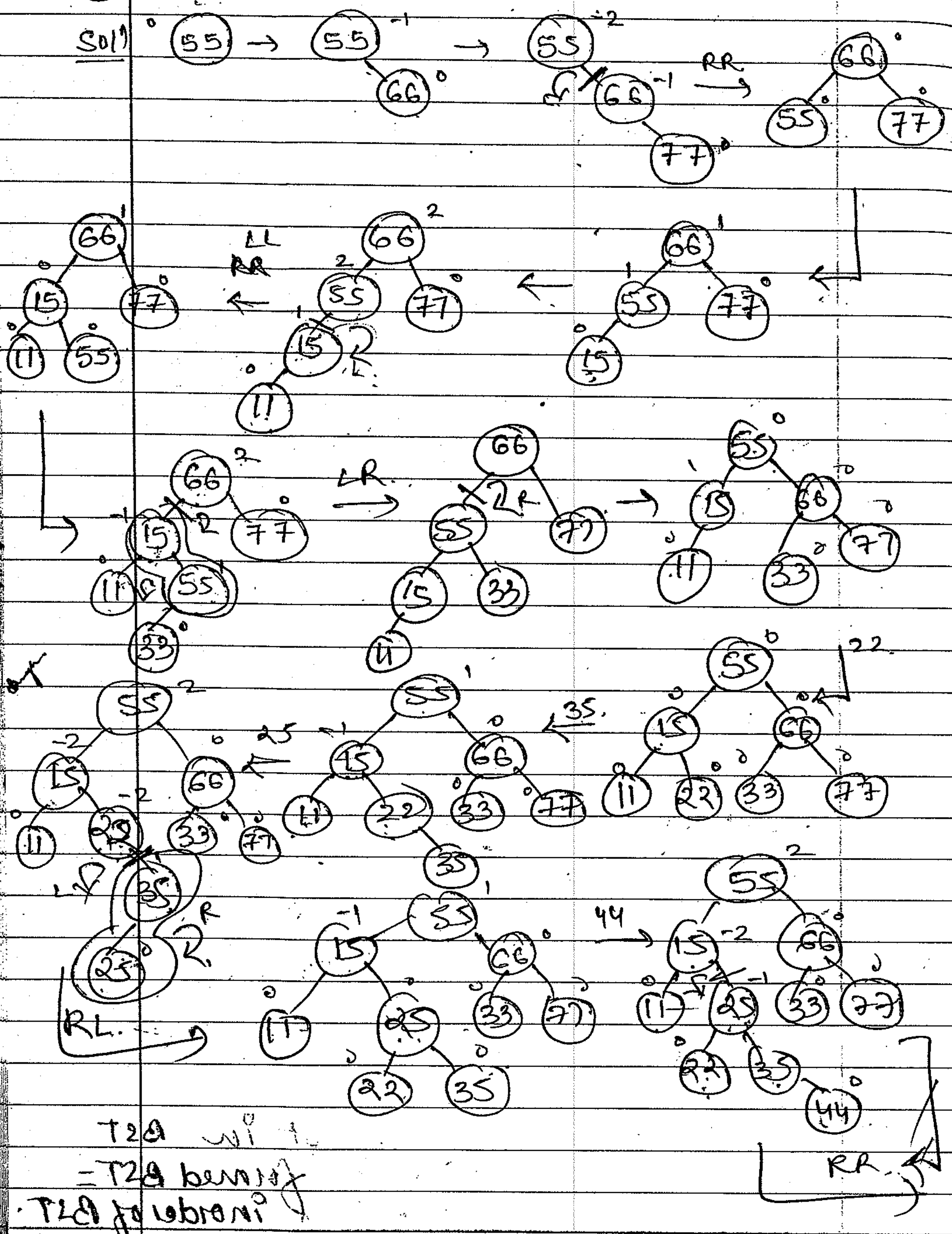


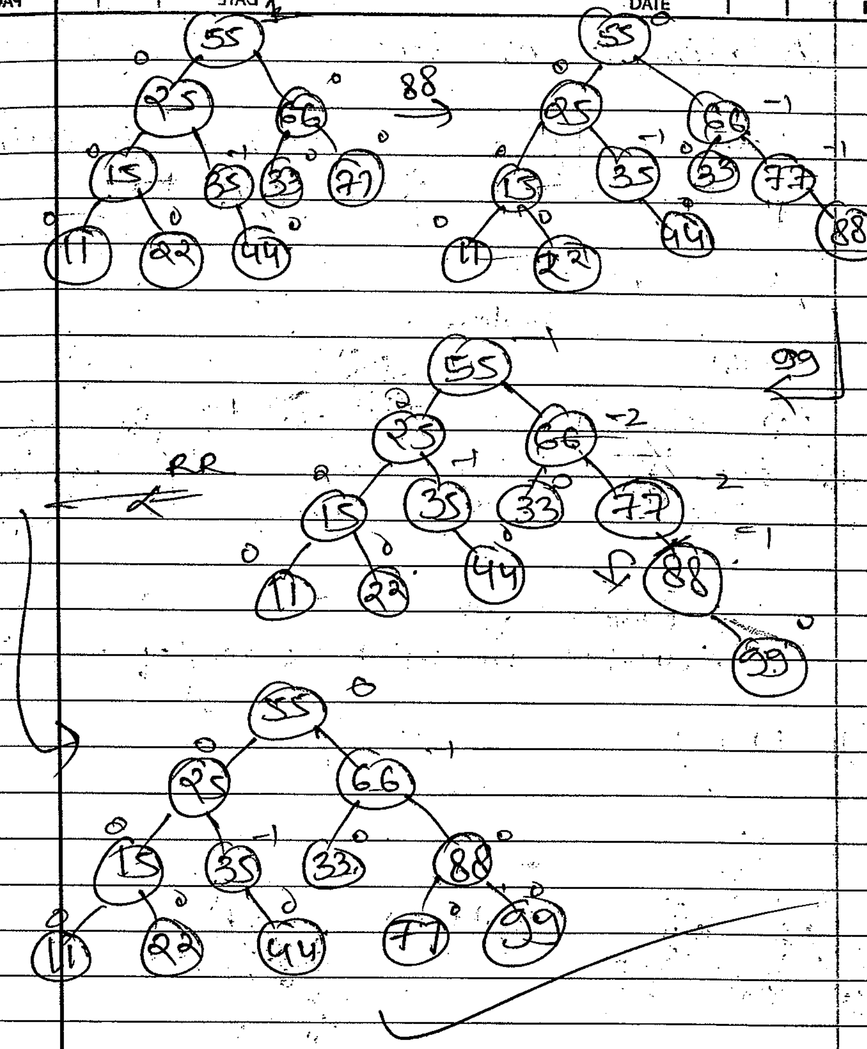
* If there is any doubt in BST then the inorder of formed BST = previous inorder of BST.

DATE PAGE

Q. 55, 66, 77, 15, 11, 33, 22, 35, 25, 44, 88, 99.

Sol^n





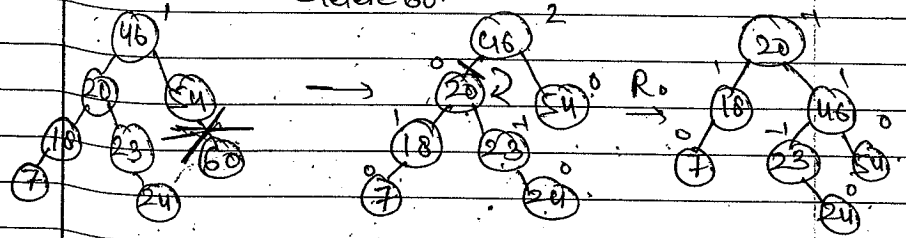
2nd Apr

DATE | | PAGE

Deletion in AVL tree

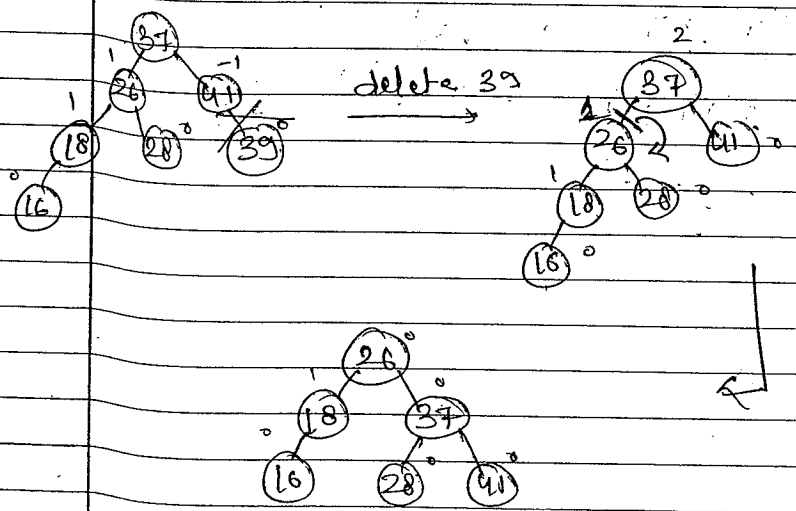
These are following rotation in deletion of any node from AVL tree

i) R0 rot^n



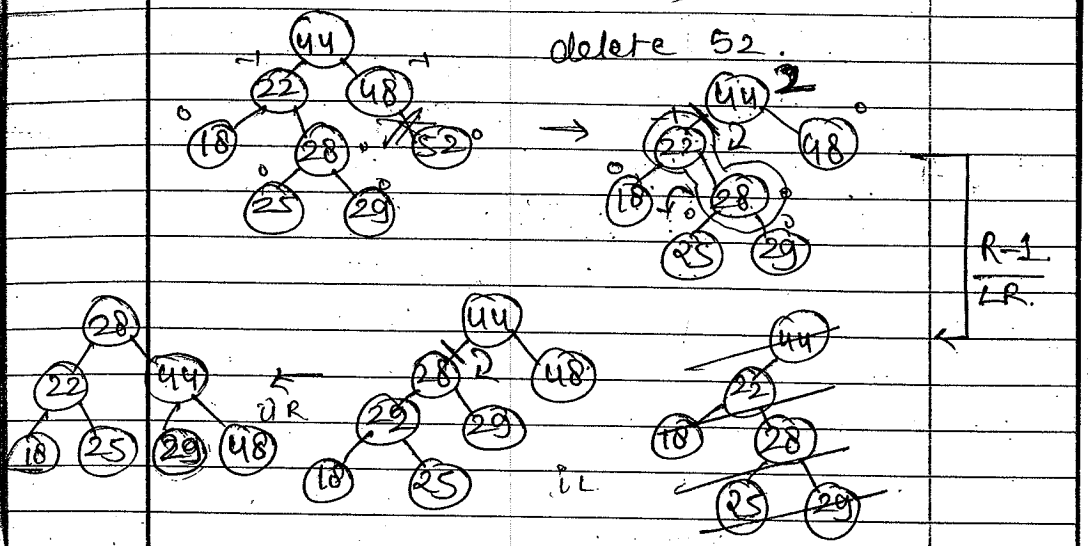
→ Rotate right
* deleted on right, 0 on left.

ii) R1 rot^n



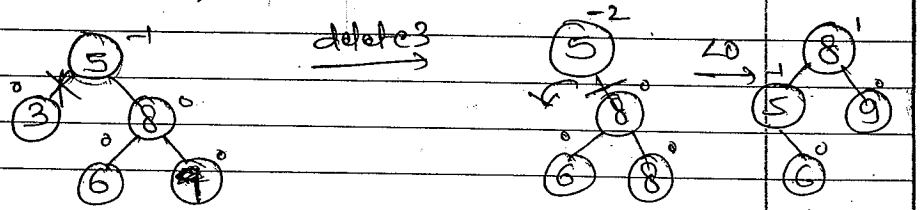
→ Rotate right
* right delete., 1 on left.

iii) R-1 rot^n (Double rot^n) (LR)



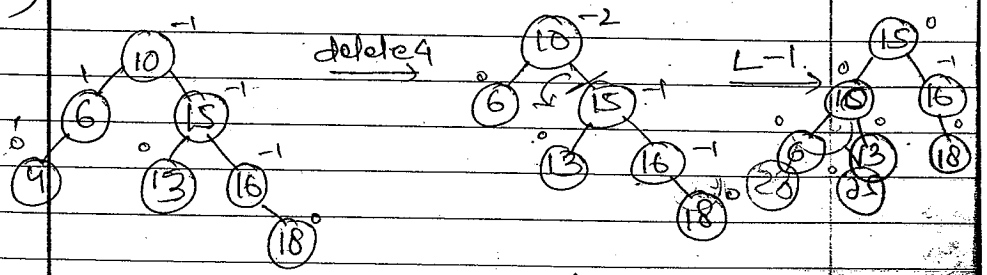
→ Rotate LR

iv) L0 rot^n (Left Rotate)



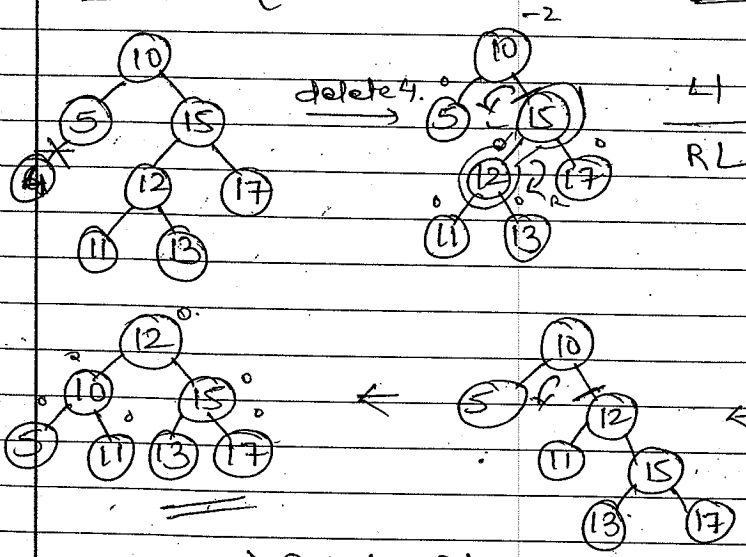
→ Rotate left.

v) L-1 rot^n

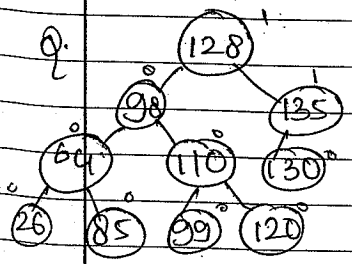


→ Rotate left.

vi) LL got (Double Rotation) RL

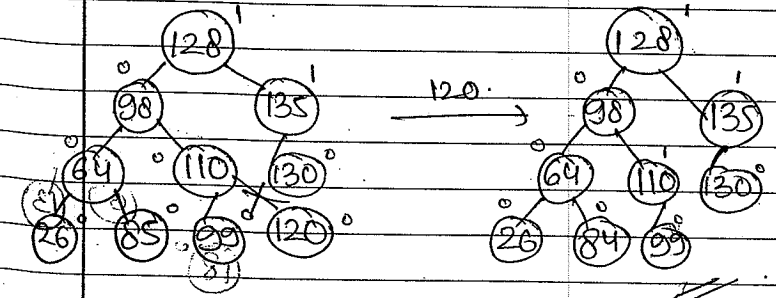


Q.



Delete 120, 64, 130.

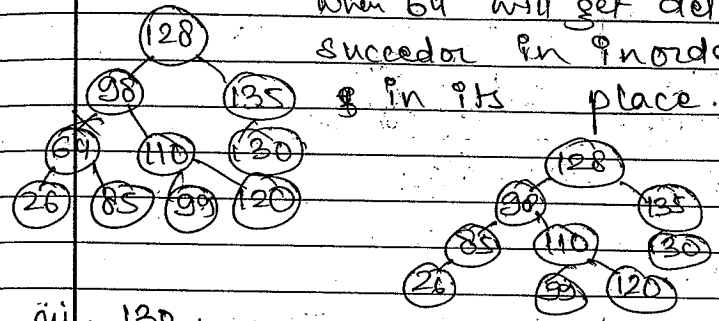
i. 120.



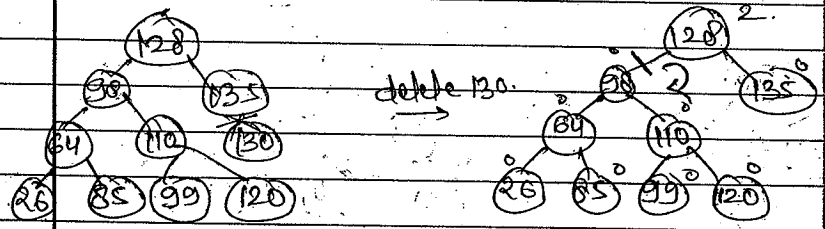
ii. 64.

Inorder.
26, 64, 85, 98, 99, 110, 120, 128, 130, 135.

When 64 will get deleted its succedor in inorder will be 85 in its place. i.e. 85.

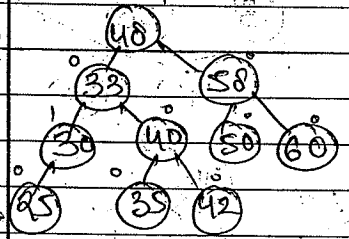


iii. 130.



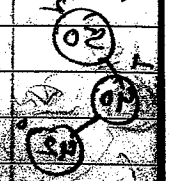
✗ Agar exam me aya to continuously delete karna hai.

Q.

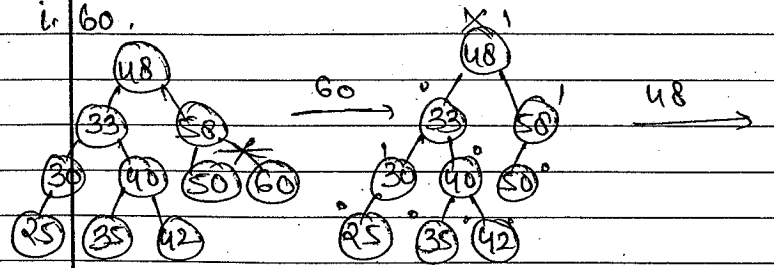


Delete

60, 40, 25, 30, 35, 33, 58.



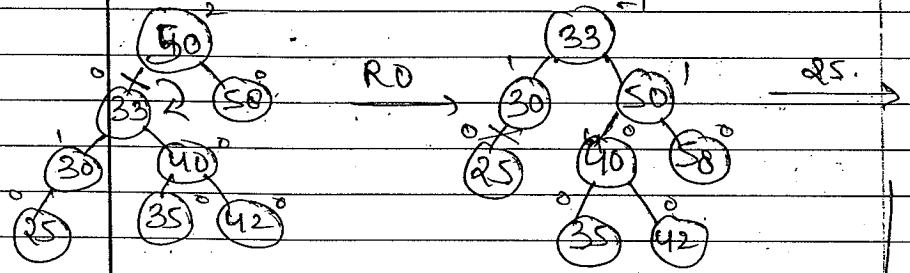
i. 60.



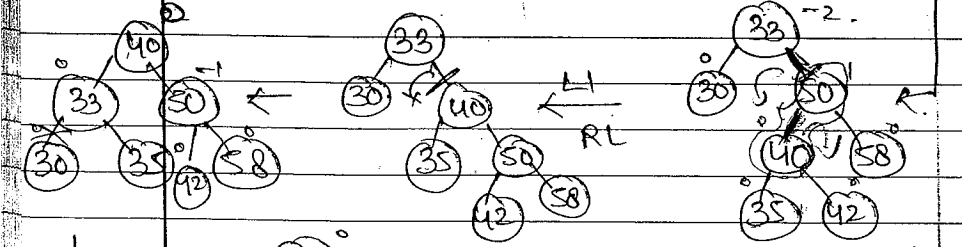
48

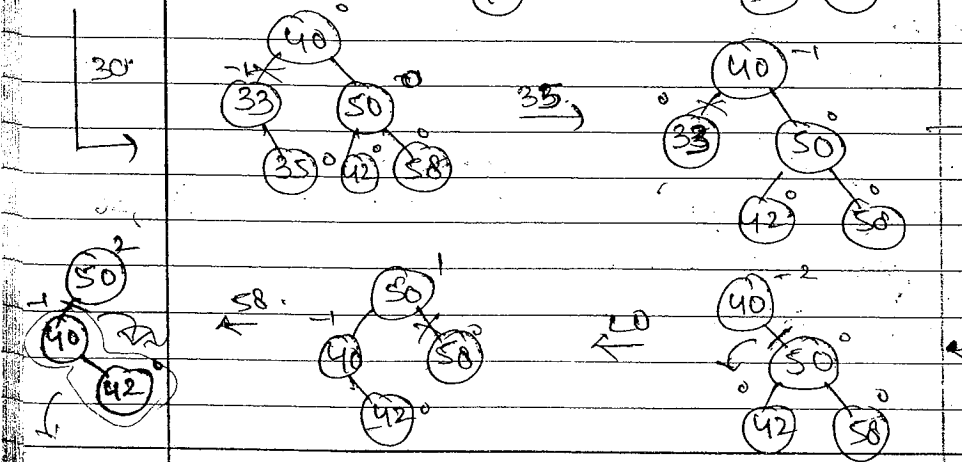
In order.

25, 30, 33, 35, 40, 42, 48, 50, 58.

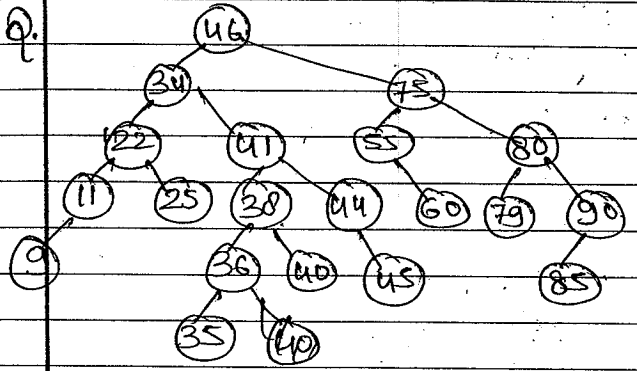


25.





Q.

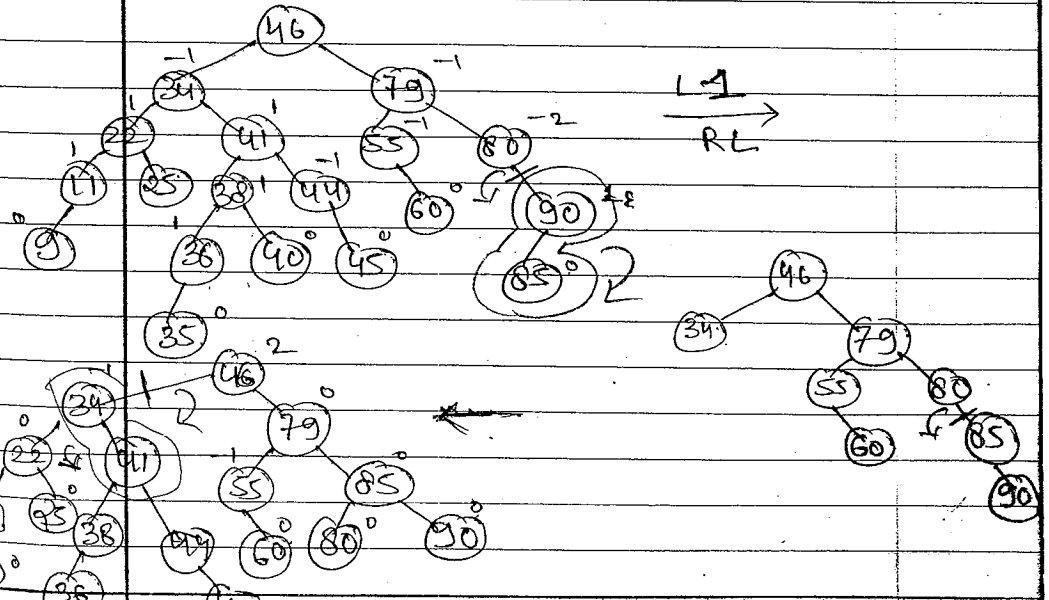


Delete 75.

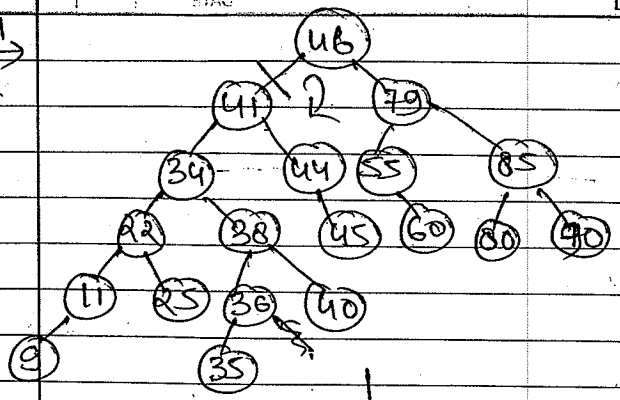
Soln

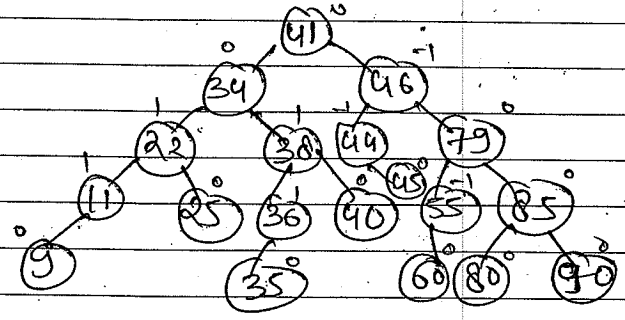
In order.

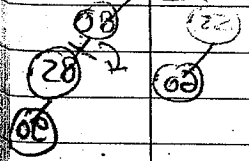
9, 11, 22, 25, 34, 35, 36, ~~40~~, 38, 40, 41, 44, 45, 46, 55, 60, 75, 79, 80, 85, 90.



R-1
LR







M-way Search Tree

In Internal searching - (linear, binary, binary search tree, AVL search tree etc). we have assumed that data to be searched is present in primary storage area but in external searching in which data is to retrieved from sec. storage like disk files.

We know, that the access time in the case of sec. storage is much more than that of primary storage. So while doing ext. searching, we should try to reduce the no. of access.

In multi-way search tree, a node can hold more than one value and can have more than 2 children. Due to large branching factor of multi-way search tree, their height is reduced, so the no. of nodes traversed to reach a particular node also decreases.

Definition: A M-way search tree of order 'M' is a search tree in which any node can have at most 'M' children. The properties of non-empty M-way search tree of order 'M' are -

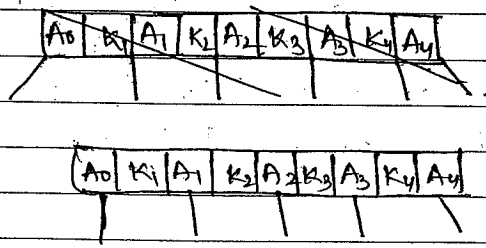
i) Each node can hold max M-1 keys & can have max N children.

ii) A node with 'N' children has N-1 key values i.e. the no. of key values is one less than the no. of children. Some of the children can be null.

iii) The keys in a node are in ascending order.

iv) Keys in a non-leaf node will divide the left and right sub trees, where value of left subtree keys will be less and value of right sub tree keys will be more than that particular key.

Eg. Consider the 5-way search tree.



a) The above node has the capacity to hold 4 keys and 5 children.

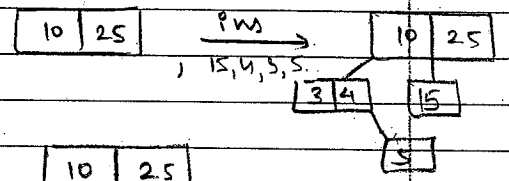
- Keys (K_1, K_2, K_3, K_4)
- Pointers to children (A_0, A_1, A_2, A_3, A_4).

b) $K_1 < K_2 < K_3 < K_4$

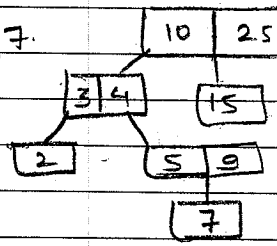
c) The key K_1 is greater than all the keys in sub tree pointed to by pointer A_0 and less that all the keys in sub-tree pointed to by pointer A_1. Similarly this get holds true for other keys also.

Q. Find 3-way search tree of 10, 25, 15, 4, 3, 5, 9, 2, 7, 11, 12, 50, 26, 23, 22, 13, 24.

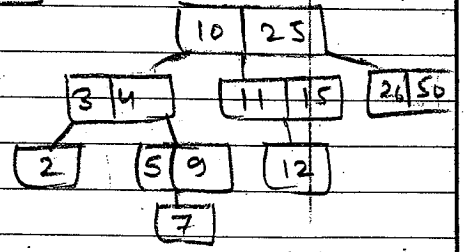
Sol) Insert 10, 25



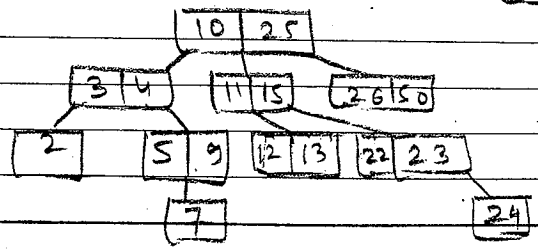
Insert 9, 2, 7.



Insert 11, 12, 50 26.

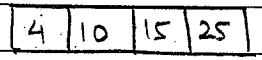


23, 22, 13, 24.

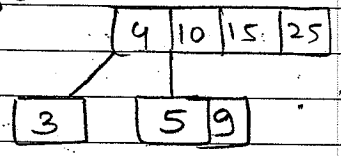


Q 5-way.
10, 25, 15, 4, 3, 5, 9, 2, 7, 11, 12, 50, 26, 23, 22, 13, 24.

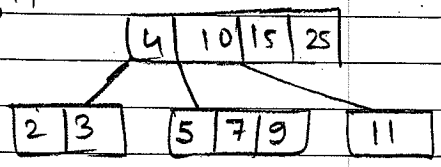
Insert 10, 25, 15, 4



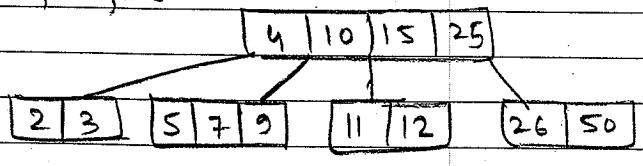
3, 5, 9



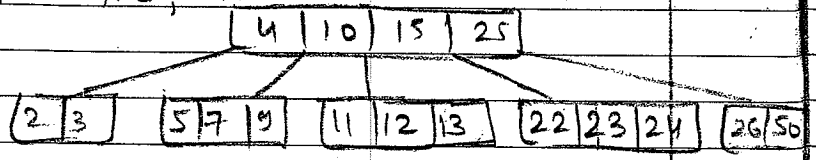
2, 7, 11



12, 50, 26



23, 22, 13, 24.



Q. Find 5-way search tree of 65, 71, 70, 66, 75, 68, 72, 77, 74, 69, 83, 73, 82, 88, 67, 76, 78, 84, 85, 80.

B-Tree

M-way search Tree has the advantage of minimizing file accesses due to their restricted height. However it is necessary that the height of the tree be kept as low as possible and ∴ there arises the need to maintain balanced m-way search tree.

Such a balanced m-way search tree is defind as B-Tree.

Definition: A B-Tree of order 'm', if non-empty, is an m-way search tree in which-

i) The root has atleast 2 child nodes & atmost m-child nodes.

ii) The Internal nodes except the root have atleast $\lceil \frac{m}{2} \rceil$ child nodes and atmost m-child nodes.

iii) The no. of keys in each internal node is one less than the no. of child nodes. And these keys

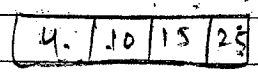
partition the keys in the subtrees of the node in a manner similar to m-way search tree.

iv) All leaf nodes are on the same level.

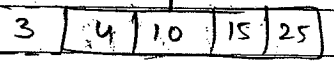
Eg: 5 order B-tree.

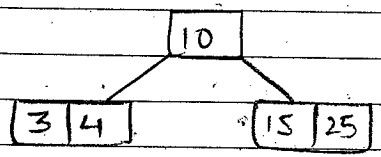
10, 25, 15, 4, 3, 5, 9, 2, 7, 11, 12, 50, 26, 23, 22, 13, 24, 6, 8

→ 10, 25, 15, 4

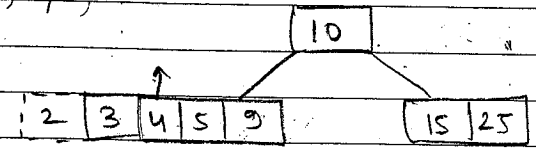


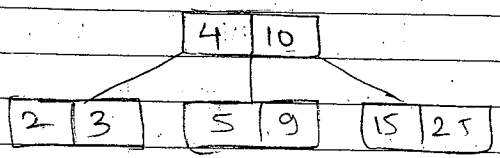
→ 3



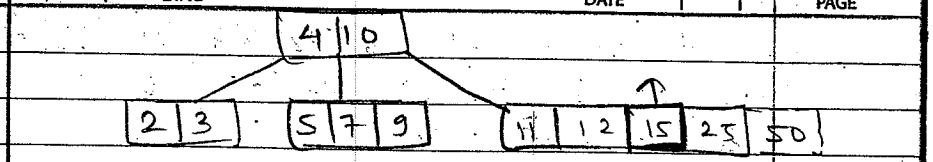


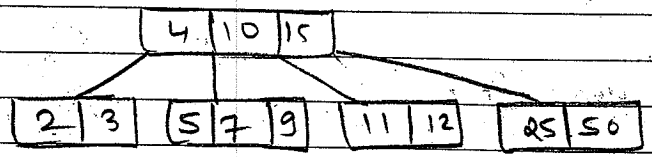
→ 5, 9, 2,



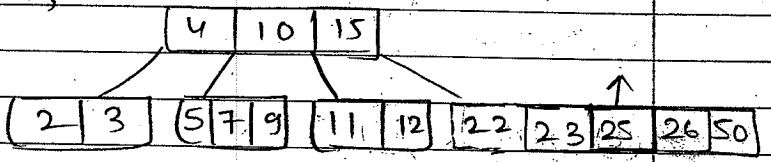


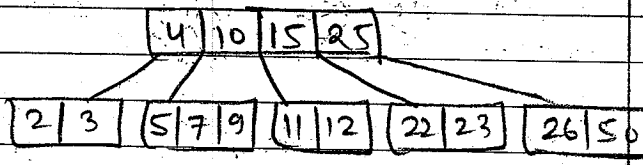
→ 7, 11, 12, 50



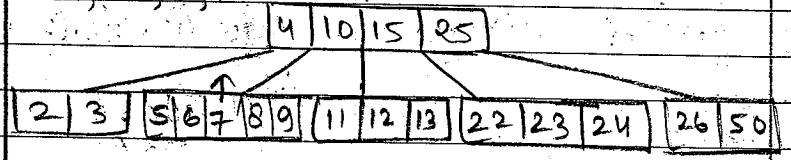


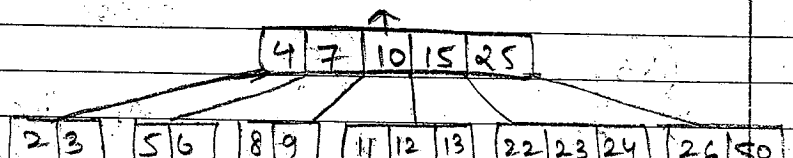
→ 26, 23, 22

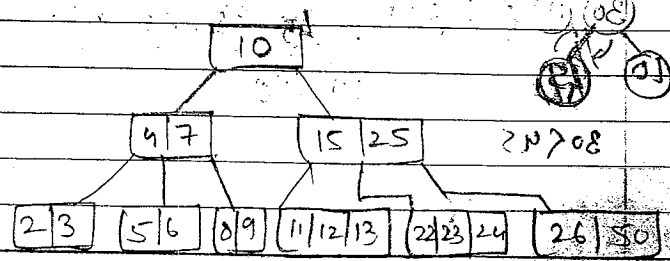




→ 13, 24, 6, 8







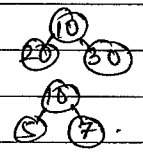
Q 65, 71, 70, 66, 75, 68, 72, 77, 74, 69, 83, 73, 82, 88, 67, 76, 78, 84, 85, 80

By 3 & 5 order.

Heap Sort

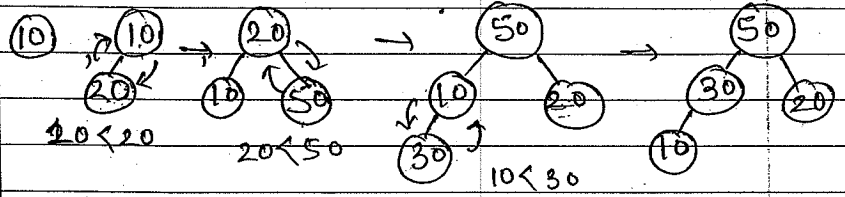
→ min

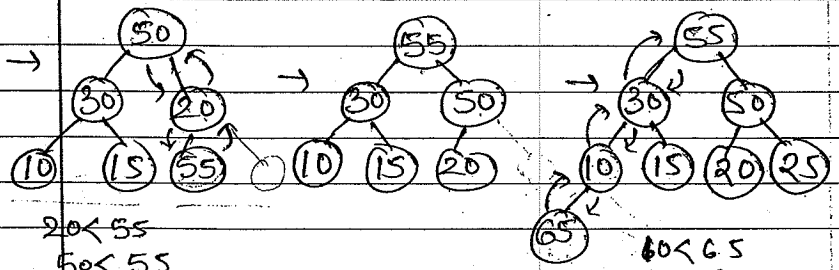
→ max

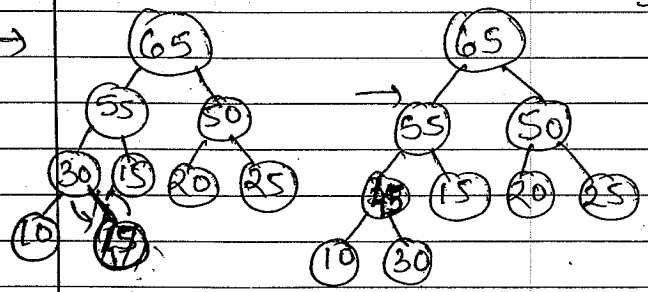


10, 20, 50, 30, 15, 55, 25, 65, 45

Building max heap.

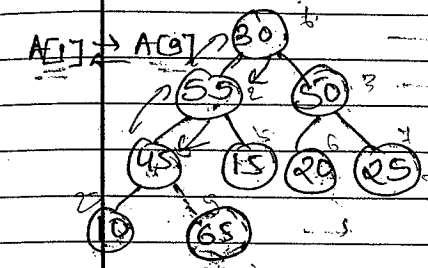




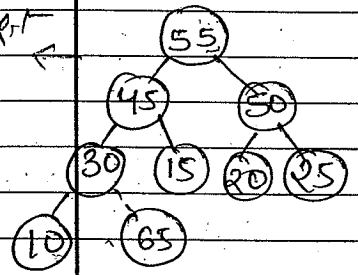


Applying heap sort

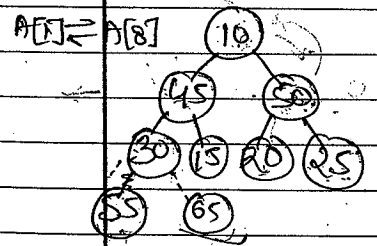
i) Interchange first and last node

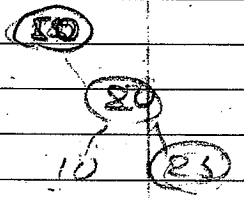


ii) Apply max heap prop. on A[1]

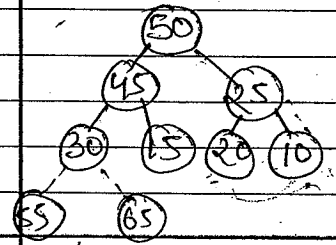


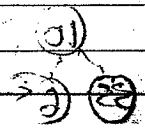
iii) Again Interchange first and last.





iv) Apply max heap prop. on A[1]

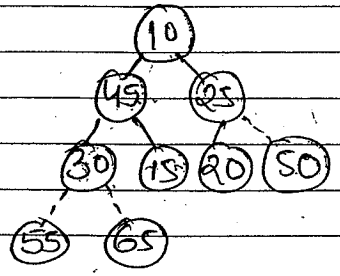




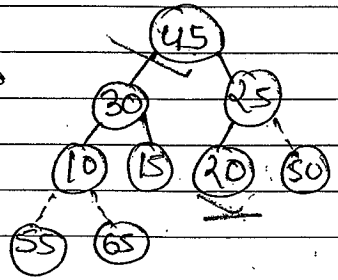
DATE | | PAGE

v) Again interchange first & last.

1 ⇄ 7.

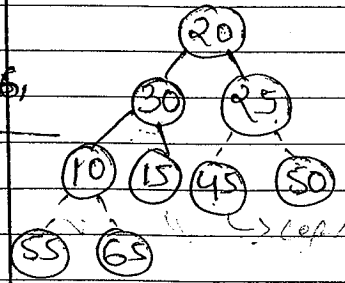


vi) max heap prop. on A[1]

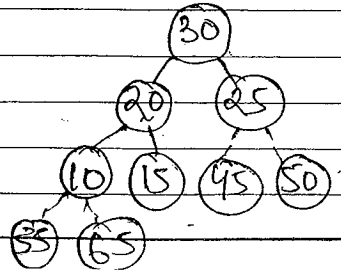


vii) Interchange

1 ⇄ 6,

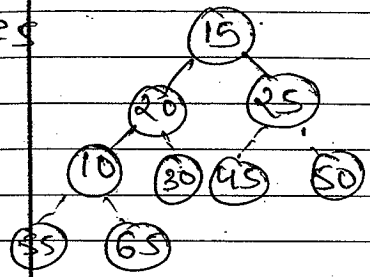


viii) max heap prop.

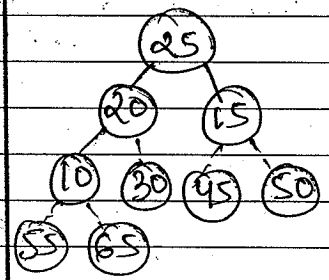


ix) Interchange

1 ⇄ 5

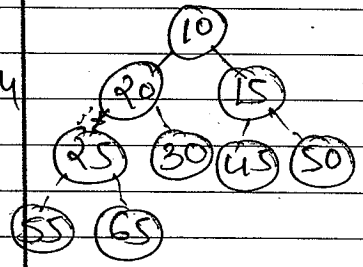


x) Max heap prop.

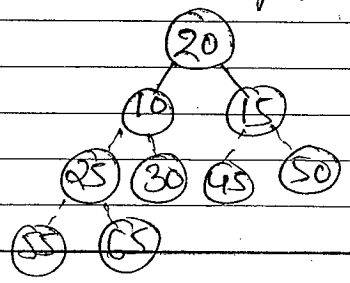


xi) Interchange.

1 ⇄ 4

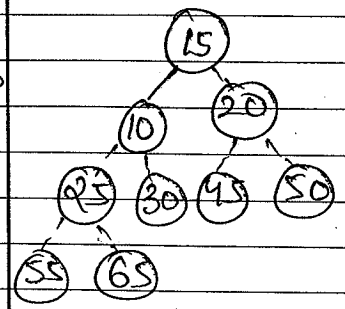


xii) Max heap prop.

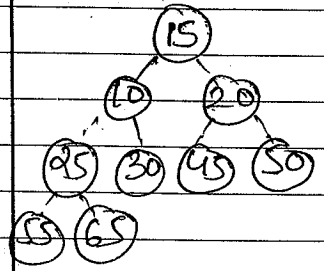


xiii) Interchange

1 ⇄ 3

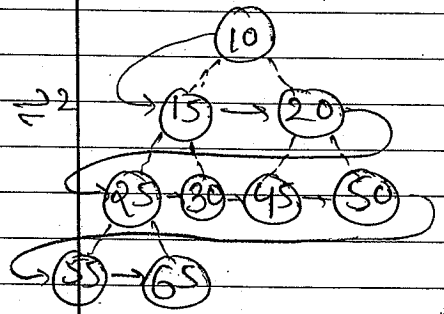


xiv) max heap prop. (no need to apply).



xv) Interchange.

1 ⇄ 2



Q 25, 35, 18, 9, 46, 70, 48, 23, 78, 12, 95.

Threaded Binary Tree

In the l.l representation of a binary tree 'T' where half of the entries in the pointer fields LEFT & RIGHT will contain NULL entries. This space may be more efficiently used by replacing the null entries by special pointers, called threads, which point to the nodes higher in the tree. Such trees are called threaded trees.

In comp. m/c, an extra field called tag or flag is used to distinguish a thread from a normal pointer.

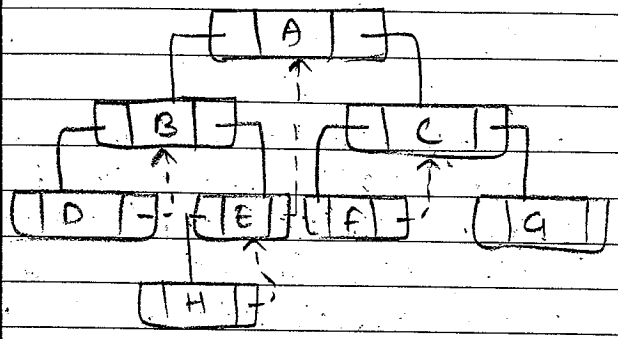
Note: ① Binary tree with 'n' nodes, where $n \geq 0$ has exactly n+1 null links

There are many ways to represent a threaded binary tree.

i) Right Threaded binary tree - The right NULL pointer of each node can be replaced by a thread to the successor of that node under inorder traversal called a RIGHT thread.

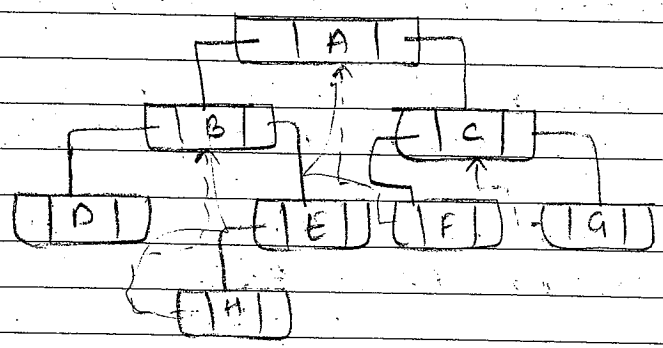
& the tree will be called Right threaded tree.

eg



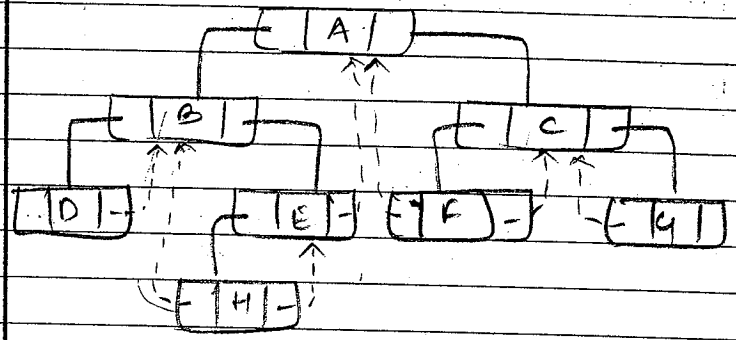
In order - DBHEAFCG.

ii) Left Threaded Binary Tree - The left NULL ptr. of each can be replaced by a thread to the predecessor of that node under inorder traversal called a left Threaded and tree will be called a L.T.T.



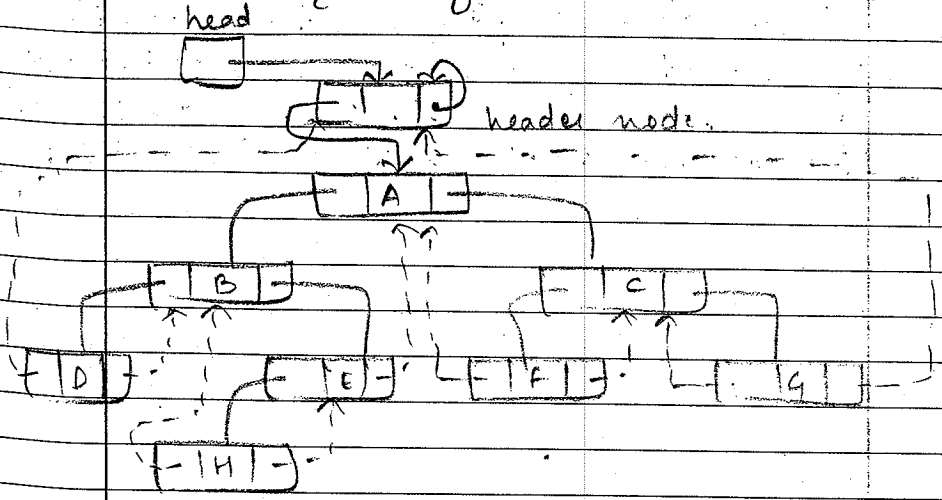
In order DBHEAFCG.

Fully Threaded B.T - Both left & right NULL ptr. can be used to point to predecessor and successor of that node, resp. under Inorder traversal such a tree is called F.T.T



Fully threaded B.T with header node. In full threaded B.T, the first node in Inorder traversal has no predecessor and last node has no successor. So the left ptr of the left most node which is the first node in Inorder traversal and the right ptr of the right most node which is the last node in Inorder traversal contains NULL. So we can take a dummy node or extra node called the header node. When this extra node is used, the tree ptr. variable which we call START/HEAD, will point to the header node & the left ptr of the header node will point-

to the root of T.



Traversing in T.B.T

Inorder traversal

In Inorder traversal the left most of the tree is traversed first of all. So, first we traverse the left most node of the tree & then we find the inorder successor of each node and traverse it. As, we know that right most node of the tree is the last node in inorder traversal & its right pointer is a thread pointing to the header node, hence we will stop when we reach header node.

Preorder traversal

In this traversal we will start traversing from the left child of the header node. If the node has a left child then that left child will be traversed, otherwise if the node has a right child then that right child will be traversed. If the node has neither left nor right child then with the help of right threads we will reach that inorder successor of the node which has a right subtree. Now, this subtree will be traversed as preorder. We will continue our process until we will reach the header node.

B+ tree

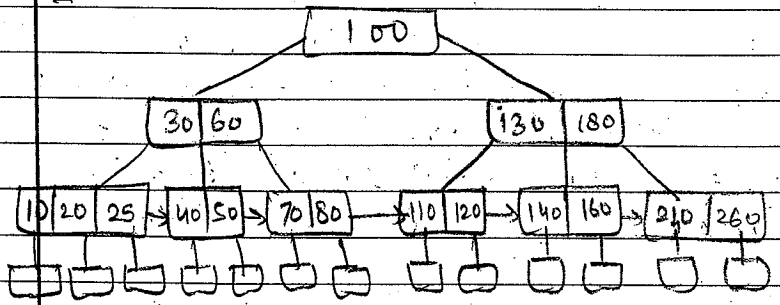
In B-Tree we can access record only randomly. We cannot traverse the record sequentially. After finding particular record, we cannot get the previous or next record coz it does not provide sequential traversing.

B+ tree has the both prop:, random access as well as sequential access.

In B+ tree all the keys which are in non-leaf node will be in leaf

node also and each leaf node will point to the next leaf node, so we can traverse sequentially also. Here keys in leaf node will point to the full record attached with that particular tree.

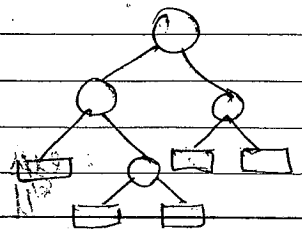
Eg:- Let us take a B+ Tree of order 5.



Huffman Algo

In extended Binary tree the path length of any node is the no. of nodes traversed from root to that node.

Consider the following extended tree or 2-Tree, the external path length is $L_E = 2+3+3+2+2 = 12$



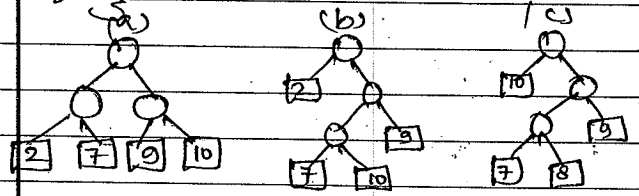
Similarly total path length for internal nodes are $L_i = 0+1+1+2 = 4$

Suppose every external node has some weight 'w', then the weighted path length for the ext. node will be

$P = w_1l_1 + w_2l_2 + \cdots + w_nl_n$

where w_i & l_i are weight & path length of an external node.

Suppose we create diff. trees which have same weights on ext. nodes then it is not necessary that they have same weighted path length.



$P1 = 2\times2 + 7\times2 + 9\times2 + 10\times2 = 56$

$P2 = 2\times1 + 7\times3 + 10\times3 + 9\times2 = 71$ &

$P3 = 10\times1 + 7\times3 + 8\times3 + 9\times2 = 73.$

We can see that 3 diff. trees have diff. path lengths even same tree of (b) & (c) have diff. path lengths.

The general prob. that the 'huffman' is solve.

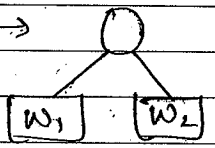
Suppose a list of 'n' weights is

Given $w_1, w_2 \ldots w_n$.
Among all the 2-Trees with 'n' external node & with the given 'n' weights, find a tree T with a min. weighted path.

Algo.

1. Suppose, there are 'n' weights $w_1, w_2 \ldots w_n$.

2. Take 2 min. weights among the 'n' given weights. Suppose w_1 & w_2 first two min. weights then sub tree will be -



3. Now the remaining weights will be $w_1+w_2, w_3, w_4 \ldots w_n$.

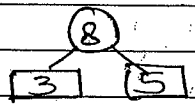
4. Create all sub tree at the last weight

Q Suppose A, B, C, D, E, F, G are 7 elements with weights as follows

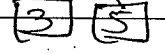
Item	A	B	C	D	E	F	G
wt	15	10	5	3	7	12	25

Create an extended binary tree by huffman algo.

Sol ① Taking 2 nodes ~~when~~ with min. wt. i.e. 3 & 5.

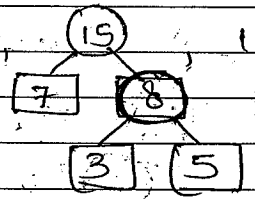


Now elements in the list are ~~15, 10,~~ & 15, 10, 8, 7, 12, 25



② Taking 2 min weights nodes i.e, 7 & 8

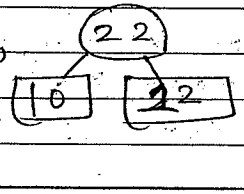
15, 10, 15, 12, 25



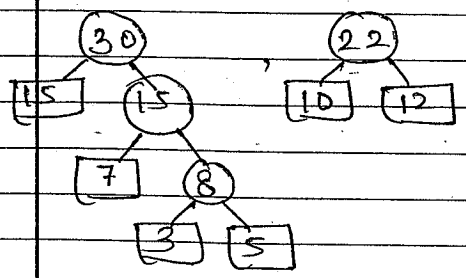
Now the elements in list are-

③ Taking 2 min weights 10 & 12.

15, 22, 15,

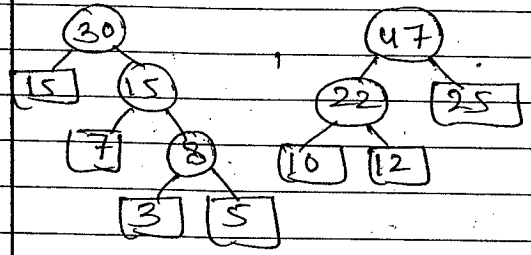


IV. Taking 2 min weight 15 & 15.



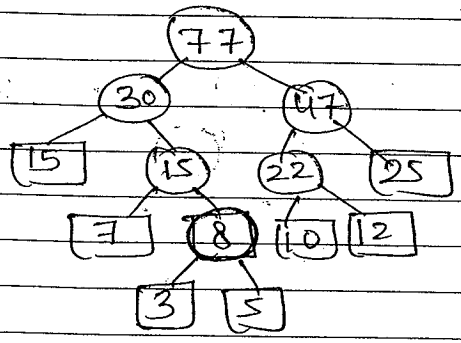
, 22 , 25.

V. Taking 22 & 25.



,

VI. Taking 30 & 47.



Application

This algo is used to perform the encoding and decoding of a long msg consisting of a set of symbols.

5th Apr

UNIT-IV

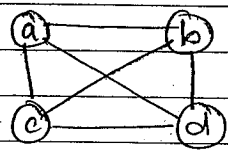
GRAPH

A graph 'G' = (V, E) is a collection of sets 'V' & 'E' where 'V' is the collection of vertices & E is collection of edges.

A graph can be of 2 types -
i) Directed
ii) Undirected.

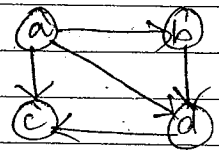
ii) Undirected - A graph which has un-ordered pair of vertices is called un-directed graph. If there is an edge b/w vertices 'u' & 'v' then it can be represented as either (u, v) or (v, u)

eg.



where V(G) = {a, b, c, d}
E(G) = {(a, b) (a, c) (a, b) (b, c) (c, d)}

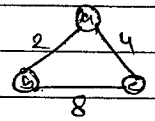
i) Direct-ed - A Di-graph is a graph which has ordered pair of vertices <u, v>.
In this type of graph, a direction is associated with each edge i.e. <u, v> & <v, u> represents diff. edges.



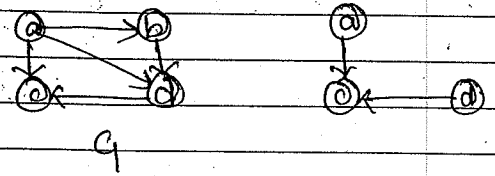
V(G) = {a, b, c, d}
E(G) = {<a, b> <b, d> <a, c> <c, d> <a, b>}
2i
log

Terminology -

1. **Weighted graph** - If the edges have been assigned with some non-ve values as weight. The wt. of the edge may represent cost or length etc. associated with the edge.



2. **Sub graph** - A graph 'H' is said to be a sub graph of another graph 'G' if and only if the vertex set of H is subset of G & edge set of H $\subseteq$ of G.



3. **Adjacency** - is a rel^n b/w 2 vertices of a graph. Vertex 'v' is adjacent to another vertex 'u' if there is an edge from u to v.

In an U.D.G, if we have an edge (u,v), it means that there is an edge from u to v & also an edge from v to u. So, the adjacency rel^n is symm. for - U.D.G. i.e., If (u,v) is an edge then u is adjacent to v & vice versa.

In a digraph, if $\langle u,v \rangle$ is an edge then v is adjacent to u but u is not adjacent to v.

4. **Incidence** - is a rel^n b/w a vertex and an edge of a graph. In an U.D.G the edge (u,v) is incident on vertices u & v.

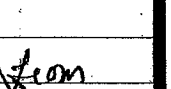
In digraph, the edge $\langle u,v \rangle$ is incident from vertex u & is incident to 'v'.

5. **Paths** - A path from vertex u_1 to u_n is a sequence of vertices $u_1, u_2, u_3 \ldots u_n$ such that u_2 is adjacent to u_1, u_3 is adjacent u_2 ...

6. **Length of path** - The length of a path is the total no. of edges included in the path.

Note: For a path with n vertices the length = n - 1

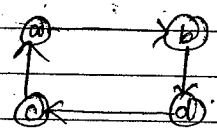
7. **Reachable** - If there is a path P from vertex u to vertex v, then v is reachable from u



is said to be reachable from vertex u via path 'P'.

8. Simple path - It is a path in which all the vertices are distinct.

9. Cycle - In a digraph, a pat $u_1, u_2, u_3 \cdots u_{n-1}, u_n$ is called a cycle if it has atleast 2 vertices and the first & last vertices are same i.e. $u_1 = u_n$.



10. Simple Cycle - A cycle $u_1, u_2 \cdots u_{n-1}, u_n$ is simple if the vertices $u_2, u_3, \cdots u_{n-2}, u_{n-1}$ are distinct.

11. Cyclic graph - A graph that has one or more cycle is called cyclic graph.

12. Acyclic graph - A graph that has no cycle is called acyclic graph.

13. Degree - In an U.D.G, the degree of a vertex is the no. of edges incident on it.

deg(a) = 2
deg(b) = 1
deg(c) = 1.

In digraph, each vertex has an Indegree and an Outdegree.

(b) = 1
(c) = 1
(d) = 2

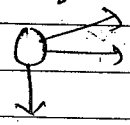
(b) = 2
(c) = 2
(d) = 0.

14. Indegree - of a vertex 'v' is the no. of edges entering on the vertex 'v' or no. of edges incident to vertex v.

15. Outdegree - of a vertex 'v' is the no. of edges leaving the vertex 'v' or no. of edges which are incident from vertex 'v'.

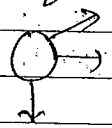
16. Source - A vertex which has no incoming edges but has outgoing edges is called the source.

The indegree of a source is 0



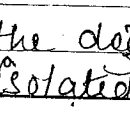
17. Sink - A vertex which has no outgoing edges, but has incoming edges, is called a sink.

The outdegree of a sink is 0.

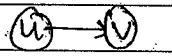


18. Pendent - A vertex in a digraph is said to be pendent if its indegree is 1 & outdegree is 0.

19. Isolated - If the degree is 0 then it is called an isolated vertex.



20. Successor & Predecessor - In a digraph if a di. vertex v is adjacent to u then v is said to be the successor of u & u is said to be the predecessor of v.



21. Max^m no. of edges in graph - In n' is total no. of vertices in graph then an U.D.G can have max^m $n(n-1)$ edges & an Digraph can have $n(n-1)$ edges.

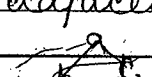
22. Loop - A edge is called loop or self-edge if it starts and ends on same vertex.

23. Multiple edges - If there is more than one edge b/w a pair of vertices then the edges are known as multiple edges. or parallel edges.

24. Multigraph - A graph which contains loop or multiple edges.

25. Simple graph - A graph which does not have loop or multiple edges.

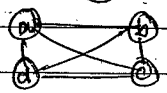
26. Regular - A graph is regular if every vertex is adjacent to same no. of vertices.



27. Connected - In a graph (U.D.G) 'G', two vertices v_i & v_j are said to be connected if there is a path in G from v_i to v_j

28. Strongly connected - A graph G' is said to be strongly connected if every pair of distinct vertices v_i, v_j in G', there is a directed path from v_i to v_j & also from v_j to v_i.

29. Complete graph - In a graph (U.D.G) 'G', if there is a connection b/w all the vertices through edges.



Representation of a graph

There are 2 standard ways of maintaining a graph G in the m/o of a comp.

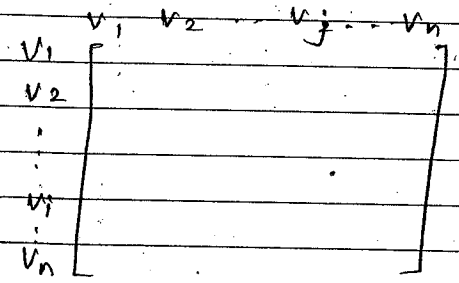
i. Sequential
ii. L.L.

i) Sequential - representation of G is represented by means of its adjacency matrix.

Adjacency matrix - is the matrix, which keeps the info. of adjacent node.

In other words we can say that this matrix keeps the info, whether this vertex is adjacent to any other vertex or not.

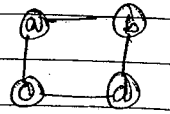
General representation of adjacency matrix is -



The entries in this matrix are placed according to the following rule.

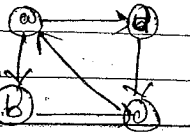
$$a_{ij} = \begin{cases} 1, & \text{if there is an edge } v_i \text{ to } v_j \\ 0, & \text{otherwise} \end{cases}$$

The adjacency matrix representation of U.D.G.



	a	b	c	d
a	0	1	1	0
b	1	0	0	1
c	1	0	0	1
d	0	1	1	0

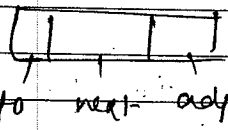
of D.G.



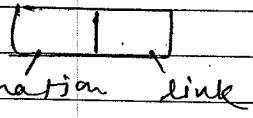
	a	b	c	d
a	0	1	0	1
b	0	0	1	0
c	1	0	0	0
d	0	0	1	0

ii) L.L - Let 'G' be a graph with 'n' vertices. In linked representation, 2 node structures are used.

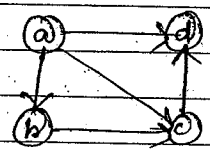
a) for vertex

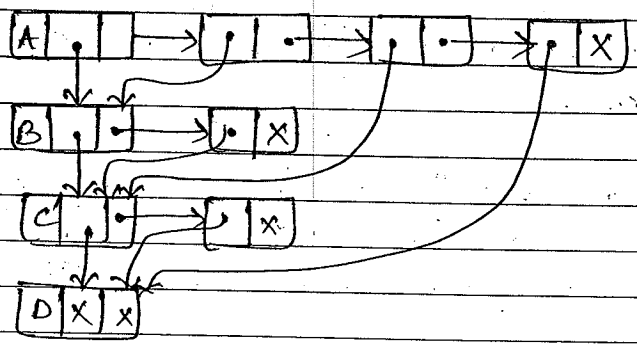


b) for edge



e.g.





Traversing of a Graph

i) BFS (based on queue)

The general idea behind BFS, beginning at a starting node A is as follows. First we examine the starting node A, then we examine all the neighbour

of A. Then we examine all the neighbours of neighbours of A.

Algorithm

This algo. executes a B.F.S. on a graph G. beginning at a starting node A.

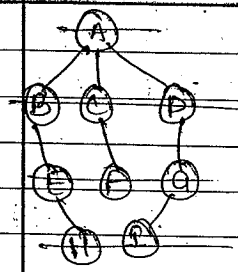
1. Initialize all nodes to the ready state (STATUS = 1).

2. Put the starting node A in queue & change its STATUS to the waiting state (STATUS = 2)

3. Repeat 4 & 5 until QUEUE is empty.

4. Remove the front node N of QUEUE. Process N & change the status of N to the processed state. STATUS = 3

5. Add to the REAR of QUEUE all the neighbours of N that are in the ready state (STATUS = 1). & change their status to the waiting state. (STATUS = 2).
[end of step 3 loop]



A B C D E F G H I

ii) DFS (based on stack)

The general idea behind a DFS beginning at a starting node A is as follows. First we examine the starting node A. Then we examine each node N along a path P which begins at A; i.e., we process a neighbour of A, then a neighbour of a neighbour of A & so on.

After coming to a "dead end", i.e., to the end of the path P, we back track on P until we can continue along another path P' and so on.

Algorithm

This algo. executes a DFS on a graph G beginning at the starting node A.

1. Initialize all nodes to the ready state (status = 1)

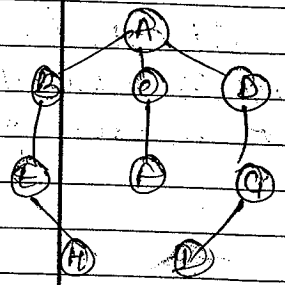
2. PUSH the starting node A on to the STACK & change its status to the waiting state (STATUS=2)

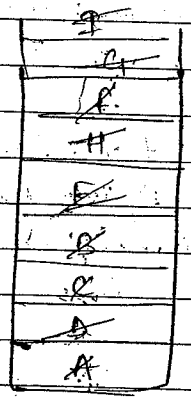
3. Repeat 4 & 5 until STACK is empty.

4. POP the top node N of stack, PROCESS N & change its STATUS to the processed state (STATUS=3)

5. PUSH on to STACK all the neighbours of N that are still in the ready state (status=1), and change their status to waiting state (STATUS=2).
[end of step 3 loop]

6. Exit.





A B E H R F D
G D.

Shortest Path Algorithm

Dijkstra Algo - is an algo. for solving the single source shortest path problem on an edge - weighted graph in which all the weights are non-ve. It finds the shortest path from some initial vertex say v_x to all other vertices one by one.

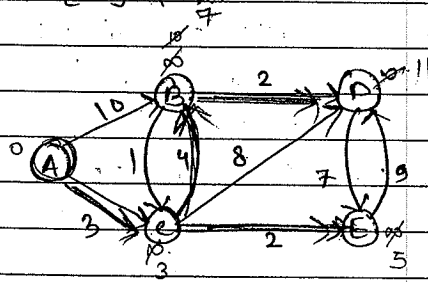
Algorithm:

```
1  INITIALIZE-SINGLE-SOURCE(G, s)
2.  S ← ϕ
3.  Q ← V[G]
4.  while Q ≠ ϕ
5.        do  u ← EXTRACT-MIN(Q)
6.            S ← S ∪ {u}
7.            for each vertex v ∈ Adj[u]
8.                do RELAX(u, v, w)
```

```
INITIALIZE-SINGLE-SOURCE(G, s)
1. for each vertex v ∈ V[G].
2.       do d[v] ← ∞
3.          π[v] ← NIL.
4.  d[s] ← 0.
```

RELAX(u,v,w)

1. If $d[v] > d[u] + w(u,v)$
2. then $d[v] \leftarrow d[u] + w(u,v)$
3. $\pi[v] \leftarrow u$



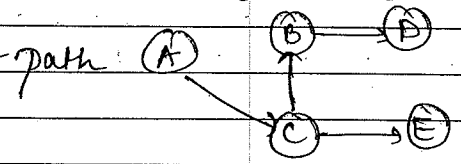
Rough qsol^n

	A	B	C	D	E
d[v]	∞ 0	∞ 10 7	∞ 3	∞ 11	∞ 5
π[v]	NIL	~~NIL~~ C	~~NIL~~ A	~~NIL~~ B	~~NIL~~ C

Q = {A, B, C, D, E}

S = {∅} S = {A, C, E, B, D}

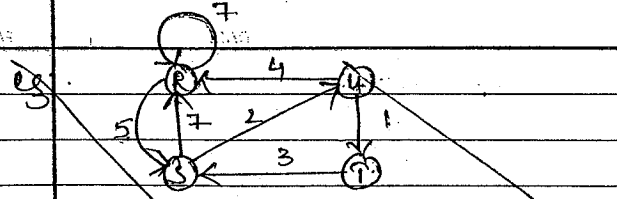
∴ Shortest path



Warshaller's Algorithm (All pair shortest path).

A weighted graph 'G' with 'm' nodes is maintained in m/o by its weight matrix W. This algorithm finds a matrix Q such that Q[I,J] is the length of a shortest path from node N_i to node N_j. INFINITY is a ~~way~~ very large no & MIN is the min. value func^n.

1. Repeat for I, J = 1, 2, ... M. [Initializes Q]
   If W[I,J] = 0, then set Q[I,J] := INFINITY.
   Else:
   set Q[I,J] := W[I,J]
   [end of loop]
2. Repeat steps 3 & 4 for K = 1, 2 .. M [Update Q]
3. Repeat step 4 for I = 1, 2 .. M
4. Repeat for J = 1, 2 .. M
   set Q[I,J] := MIN(Q[I,J], Q[I,K] + Q[K,J])
   [end of step 3 loop]
   [end of step 2 loop]
5. Exit.



$$\Rightarrow W = \begin{array}{c} \\ R \\ S \\ T \\ U \end{array} \begin{array}{c} \begin{array}{cccc} R & S & T & U \end{array} \\ \begin{bmatrix} 7 & 5 & 0 & 0 \\ 7 & 0 & 0 & 2 \\ 0 & 3 & 0 & 0 \\ 4 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

Now.

$$Q_0 = \begin{array}{c} \\ R \\ S \\ T \\ U \end{array} \begin{array}{c} \begin{array}{cccc} R & S & T & U \end{array} \\ \begin{bmatrix} 7 & 5 & \infty & \infty \\ 7 & \infty & \infty & 2 \\ \infty & 3 & \infty & \infty \\ 4 & \infty & 1 & \infty \end{bmatrix} \end{array} \quad \begin{array}{c} \\ R \\ S \\ T \\ U \end{array} \begin{array}{c} \begin{array}{cccc} R & S & T & U \end{array} \\ \begin{bmatrix} RR & RS & - & - \\ SR & - & - & SU \\ - & TS & - & - \\ UR & - & UT & - \end{bmatrix} \end{array}$$

$$Q_1 = \begin{array}{c} \\ R \\ S \\ T \\ U \end{array} \begin{array}{c} \begin{array}{cccc} R & S & T & U \end{array} \\ \begin{bmatrix} 7 & 5 & \infty & \infty \\ 7 & (12) & \infty & 2 \\ \infty & 3 & \infty & \infty \\ 4 & (9) & 1 & \infty \end{bmatrix} \end{array} \quad \begin{array}{c} \\ R \\ S \\ T \\ U \end{array} \begin{array}{c} \begin{array}{cccc} R & S & T & U \end{array} \\ \begin{bmatrix} RR & RS & - & - \\ SR & SRS & - & SU \\ - & TS & - & - \\ UR & URS & UT & - \end{bmatrix} \end{array}$$

→ R in middle RS, RR+RS.

Q. [RS] = min [5, 7+5] = 5.

[RT] = min [∞, 7+∞] = ∞.

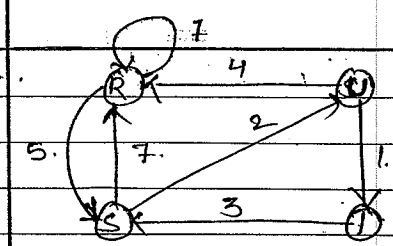
[RU] = min [RT, RR+RU] = min[∞, 7+∞]
= ∞.

ST = Min(ST, SR+RT) = (∞, 7+∞) = ∞

SU = (SU, SR+RU) = (2, 7+∞) = 2

1

1



$$\Rightarrow W = \begin{array}{c} \\ R \\ S \\ T \\ U \end{array} \begin{array}{c} \begin{array}{cccc} R & S & T & U \end{array} \\ \begin{bmatrix} 7 & 5 & 0 & 0 \\ 7 & 0 & 0 & 2 \\ 0 & 3 & 0 & 0 \\ 4 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

$$Q_0 = \begin{array}{c} \\ R \\ S \\ T \\ U \end{array} \begin{array}{c} \begin{array}{cccc} R & S & T & U \end{array} \\ \begin{bmatrix} 7 & 5 & \infty & \infty \\ 7 & \infty & \infty & 2 \\ \infty & 3 & \infty & \infty \\ 4 & \infty & 1 & \infty \end{bmatrix} \end{array} \quad \begin{bmatrix} RR & RS & - & - \\ SR & - & - & SU \\ - & TS & - & - \\ UR & - & UT & - \end{bmatrix}$$

→ R in middle

Q. [RS] = min [RS, RR+RS] = [5, 7+5] = 5.

Q. [RT] = min [RT, RR+TR] = [∞, 7+∞] = ∞.

[RU] = [RU, RR+UR] = [∞, 7+∞] = ∞.

[SR] = [SR, SR+RR] = [7, 7+7] = 7.

[SS] = [SS, SR+RS] = [∞, 7+5] = 12.

[ST] = [ST, SR+RT] = [∞, 7+∞] = ∞.

[SU] = [SU, SR+RU] = [2, 7+∞] = 2.

[TR] = [TR, TR+RR] = [∞, ∞+7] = ∞.

[TS] = [TS, TR+RS] = [3, ∞+5] = 3

[TT] = [TT, TR+RT] = [∞, ∞+∞] = ∞.

[TU] = [TU, TR+RU] = [∞, ∞+∞] = ∞.

[UR] = [UR, UR+RR] = [4, 4+7] = 4.

[US] = [US, UR+RS] = [∞, 4+5] = 9

[UT] = [UT, UR+RT] = [1, 4+∞] = 1.

[UU] = [UU, UR+RU] = [∞, 4+∞] = ∞.

R S T U

$Q_1 =$

	R	S	T	U
R	7	5	∞	∞
S	7	(12)	∞	2
T	∞	3	∞	∞
U	4	(9)	1	∞

RR	RS	–	–
SR	SRS	–	SU
–	TS	–	–
UR	URS	UT	–

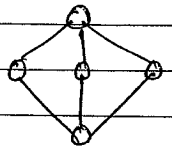
DATE | | PAGE

DATE | | PAGE

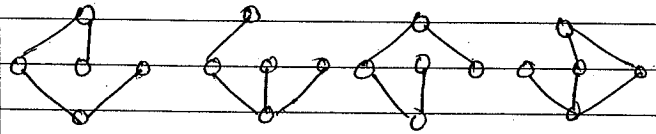
Spanning Tree

A tree T is said to be a spanning tree of the connected graph G if T is a sub-graph of G & T contain all vertices of G without forming a cycle or ckt.

eg: Consider the following connected graph.



Spanning trees are -



Minimum Cost spanning tree

If a weighted graph is considered, then the weight of the spanning tree T of a graph G can be calculated by summing all the individual weight in the spanning tree T. But we have seen that for any graph G there can be many spanning tree. This is also in case of weighted graph for which we can get different spanning tree having different weights.

A min. cost spanning tree is a spanning tree of min. weights.

MST - Min. Spanning Tree.

i) Kruskal's Algorithm - It is one of the eg. of greedy algo. In which edges are selected at greed with min. weight & added to M.S.T till no ckt. is formed. It is used to find min. cost.

Steps for solving Kruskal's algo -

1. Arrange all the edges in increasing order of their weight.
2. Add to min. spanning tree, the edges, if it does not form a ckt.
3. Continue, till all edges are visited.

Algorithm

```
MST-KRUSKAL(G, w)
1. A ← ∅
2. for each vertex v ∈ V[G]
3.     do MAKE-SET(v)
4. Sort the edges of E into non-decreasing order by wt. 'w'.
5. for each edge (u,v) ∈ E, taken in non-decreasing order by wt.
6.     do if FIND-SET(u) ≠ FIND-SET(v)
7.         then A ← A ∪ {(u,v)}
8.             UNION(u,v)
9. Return A
```

```
MAKE_SET(x)
P[x] ← x
RANK[x] ← 0.
```

```
FIND_SET(x)
1. if x ≠ P[x]
       then P[x] ← FIND_SET(P[x])
   return P[x]
```

UNION(x,y)

1. LINK(FIND-SET(x), FIND-SET(y))

LINK(x,y)

1. if rank[x] > rank[y]
2. then P[y] ← x
3. else P[x] ← y
4. if rank[x] = rank[y]
5. then rank[y] ← rank[y] + 1

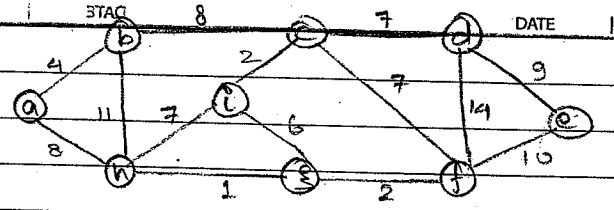
ii) Prim's Algorithm

Algorithm

MST-PRIMS(G, w, r)

1. for each u ∈ V[G]
2. do key[u] ← ∞
3. π[u] ← NIL
4. key[r] ← 0
5. Q ← V[G]
6. while Q ≠ ∅
7. do u ← EXTRACT-MIN(Q)
8. for each v ∈ Adj[u]
9. do if v ∈ Q & w(u,v) < key[v]
10. then π[v] ← u
11. key[v] ← w[u,v]

Q.

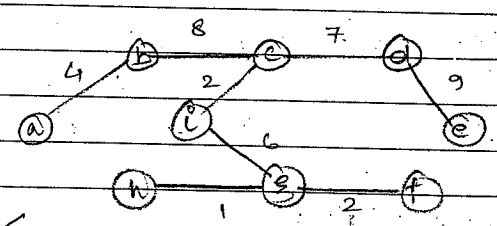


eg of Prim's.

→ MST-PRIMS(G, w, r) By Prim's.

	a	b	c	d	e	f	g	h	i
key	∞ 0	∞ 4	∞ 8	∞ 7	∞ 10 9	∞ 2	∞ 1	∞ 6	∞ 2
π	NIL	NIL a	NIL b	NIL c	NIL d	NIL g	NIL i	NIL g	NIL c

Q = {a, b, c, d, e, f, g, h, i}



Total = 39

→ // By Kruskal's.

	a	b	c	d	e	f	g	h	i
P[u]	a b	b h	c i	d h	e h	f h	g h	h i	i h
rank[x]	0	0 1	0	0	0	0	0	0	0 2

by Make-Set Algo.

A = { }

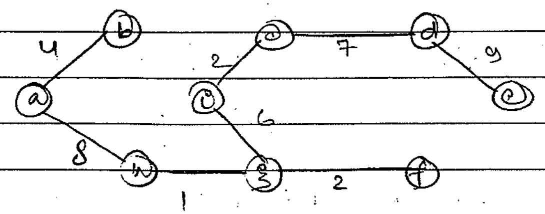
Non-decreasing order = {g,h}, {c,i}, {f,g}, {a,b}, {i,g}, {c,f}, {c,d}, {i,h}, {a,h}, {b,c}, {d,e}, {f,e}, {b,h}, {d,f}

→ A = { {g,h}, {c,i}, {f,g}, {a,b}, {i,g}, {c,d}, {a,h}, {d,e} }

We will select the edge whose find set is equal

Find set

{g, h}	{c, i}	{f, g}	{a, b}	{i, g}	{c, f}	{c, d}	{i, h}
g ≠ h	c ≠ i	f ≠ h	a ≠ b	i ≠ h	h = h ✗	h ≠ d	h = h ✗
{a, h}	{b, c}	{d, e}	{f, e}	{b, h} ✗	{d, f}		
b ≠ h	✗ h = h	h ≠ e	✗ h = h	h = h	✗ h = h		



Total = 39.

(U-5) **Selection Sort (A, N)**

This algo. sorts the array A.

1. Repeat step 2 & 3 for K := 1, 2, 3 . . N-1
2. Call MIN (A, K, N, LOC)
3. [Interchange A[K] and A[LOC]]
   Set TMP := A[K], A[K] := A[LOC],
   & A[LOC] := TMP.
   [end of loop]
4. Exit.

Procedure MIN (A, K, N, LOC)

An array A is in m/o. This procedure finds the locn LOC of the smallest element among A[K], A[K+1], . . . A[N]

1. Set MIN := A[K] & LOC := K
   [Initialize]
2. Repeat for J := K+1, K+2, ... N
   If MIN > A[J], then:
   Set MIN := A[J] & LOC := J
   [end of loop].
3. Return.

Q. Sort the following list in ascending array using selection sort.
{77, 33, 44, 11, 88, 22, 66, 55}

Sol^n K is the no. of passes and LOC is the locn of smallest element.

Min. dhundte hai ... min value nikalte, fir usko ek tak ... (kar) sabse check karte

Pass	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
K=1 LOC=4	(77)	33	44	(11)	88	22	66	55
K=2 LOC=6	11	(33)	44	77	88	(22)	66	55
K=3 LOC=6	11	22	(44)	77	88	(33)	66	55
K=4 LOC=6	11	22	33	(77)	88	(44)	66	55
K=5 LOC=8	11	22	33	44	(88)	77	66	(55)
K=6 LOC=7	11	22	33	44	55	(77)	(66)	88
K=7 LOC=7	11	22	33	44	55	66	(77)	88
SORTED	11	22	33	44	55	66	77	88

Analysis of Selection Sort

When the no. T(n) of comparisons in the selection sort algo. is independent of original order of elements. Observe that MIN(A,K,N,LOC) requires N-K comparisons i.e. there are N-1 comparison during pass 1 to find the smallest element, there are N-2 comparison during pass two, to find 2nd smallest element and so on, then $T(n) = (n-1)+(n-2)+(n-3)+\cdots+1 = \frac{n(n-1)}{2}$.

Case	no. of comparison	complexity
Best case	$\frac{n(n-1)}{2}$	$O(n^2)$
Avg.	$\frac{n(n-1)}{2}$	$O(n^2)$
Worst	$\frac{n(n-1)}{2}$	$O(n^2)$

Bubble Sort (DATA, N)

Here, DATA is an array with N elements. This algo. sorts the elements in DATA.

1. Repeat step 2 & 3 for K=1 to N-1
2. Set PTR := 1 [Initialize pass pointer PTR]
3. Repeat while PTR <= N-K [execute,]
   a) If DATA[PTR] > DATA[PTR+1], then:
      Interchange DATA[PTR] & DATA[PTR+1]
   [end of if struct]
   b) Set PTR := PTR+1
   [end of inner loop]
[end of outer loop]
4. Exit.

eg. Pass 1

(32)	(51)	27	85	66	23	13	57	
32	(51)	(27)	85	66	23	13	57	
32	27	(51)	(85)	66	23	13	57	
32	27	51	(85)	(66)	23	13	57	
32	27	51	66	(85)	(23)	13	57	
32	27	51	66	23	85	(13)	(57)	
32	27	51	66	23	13	85	57	

Pass 2

(32)	(27)	51	66	23	13	57	85
27	(32)	(51)	66	23	13	57	85
27	32	(51)	(66)	23	13	57	85
27	32	51	(66)	(23)	13	57	85
27	32	51	23	(66)	(13)	57	85
27	32	51	23	13	66	57	85
27	32	51	23	13	57	66	85
27	32	51	23	13	57	66	85

Pass 3: (27) (32) 51 23 13 57 66 85
27 (32) (51) 23 13 57 66 85
27 32 (51) (23) 13 57 66 85
27 32 23 (51) (13) 57 66 85
27 32 23 13 (51) (57) 66 85
27 32 23 13 51 57 66 85

Pass 4: (27) (32) 23 13 51 57 66 85
27 (32) (23) 13 51 57 66 85
27 23 (32) (13) 51 57 66 85
27 23 13 (32) (51) 57 66 85
27 23 13 32 51 57 66 85

Pass 5: (27) (23) 13 32 51 57 66 85
23 (27) (13) 32 51 57 66 85
23 13 (27) (32) 51 57 66 85
23 13 27 32 51 57 66 85

Pass 6: (23) (13) 27 32 51 57 66 85
13 (23) (27) 32 51 57 66 85
13 23 27 32 51 57 66 85

Pass 7: (13) (23) 27 32 51 57 66 85
13 23 27 32 51 57 66 85

Sorted: 13 23 27 32 51 57 66 85

Analysis

The complexity of bubble sort can be computed by calculation of no. $T(n)$ of comparisons. There are $(N-1)$ comparison during the 1st pass which places largest element in last posn. There are $(N-2)$ in 2nd step which places 2nd largest element in the 2nd last posn & so on. Thus the worst-case time complexity $T(n) = (n-1)+(n-2)+\cdots+1 = \frac{n(n-1)}{2} = O(n^2)$.

Insertion Sort (A, N)

This algo sorts the array A with N elements.

1. Set A[0] := −∞ [Initialize A[0]]
2. Repeat step 3 to 5 for K = 2, 3, ..., N
3. Set TMP := A[K] & PTR := K−1.
4. Repeat while TMP < A[PTR]
   a) Set A[PTR+1] := A[PTR]
   [moves the element 4wd]
   b) Set PTR := PTR − 1
   [end of loop]

5 Set A[PTR+1] := TMP [Insert element in proper place]
[end of step 2 loop].

6. Exit.

Q Sort following list in ascending order using Insertion sort.
77, 33, 44, 11, 88, 22, 66, 55.

(Sol) ← Right to left ok.

Passes	A[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
K=1	-∞	(77)	33	44	11	88	22	66	55
K=2	-∞	~~77~~	(33)	44	11	88	22	66	55
	T=33 -∞	~~77~~	77						
K=3	-∞	33	~~77~~	(44)	11	88	22	66	55
	T=44 -∞	33	~~77~~	77					
K=4	-∞	33	44	~~77~~	(11)	88	22	66	55
	T=11 -∞	33	44	~~77~~	77				
		33	44	~~44~~	77				
		33	33	44	77				
K=5	-∞	11	33	44	77	(88)	22	66	55
K=6	-∞	11	33	44	77	88	(22)	66	55
	T=22 -∞	11	33	44	77	88	88		
		11	33	44	77	77	88		
		11	33	44	44	77	88		
		11	33	33	44	77	88		
K=7	-∞	11	22	33	44	77	88	(66)	55

* T = 66

	-∞	11	22	33	44	77	88	88
	-∞	11	22	33	44	77	77	88
K=8	-∞	11	22	33	44	66	77	88 (55)

T = 55

	-∞	11	22	33	44	66	77	88 88
	-∞	11	22	33	44	66	77	77 88
	-∞	11	22	33	44	66	66	77 88
	-∞	11	22	33	44	55	66	77 88

Sorted - -∞ 11 22 33 44 55 66 77 88.

Analysis

Best case - The I/P list is always in sorted order in best case the no. of comparison in each iteration is 1. The total no. of iteration required is = N-1, where N = size of array. ∴ T(n) = 1+1+1+... + N-1 upto N-1 iteration = N-1. = O(N).

1+1+1+..

Worst case - The I/P array is sorted but in reverse order. In this case ith iteration requires i no. of comparison. Hence, considering the total n-1 iteration. The T(n) = 1+2+3+ ..... + n-1 = n(n-1)/2 = O(n²)

1+2+3+4

Avg case - Array is in random order, in avg case

there will be approx. $\frac{K-1}{2}$ comparisons in the inner loop. So in the avg case $T(n) = \frac{1}{2} + \frac{2}{2} + \frac{3}{2} + \cdots + \frac{n-1}{2} = \frac{n(n-1)}{4}$

$= O(n^2)$

Radix sort

1. An array A, where N is the no. of elements, this algo. sorts the array A in ascending order., b no. of auxiliary arrays each of size n is used.

1. For I := 1 to c do [c denotes the most significant pos]
2. for J := 1 to n do [for all elements in the array A].
3. X := EXTRACT (A[J], I) [get the i^{th} component in the j^{th} element]
4. Enque (Q_X, A[J])
5. [end of inner for loop]. [combine all elements from all auxiliary arrays to A (assume A is empty)]
6. for K = 0 to (P b − 1) do
7. while Q_K is not empty, do
8. y = Dequene (Q_K)
9. Insert (A, y).
10. [end of while] [end of for] [end of for]
11. Exit.

A: | 136 | 487 | 358 | 469 | 570 | 247 | 598 | 639 | 205 | 609 |
Dip List

Distribution of elements into 10 auxiliary arrays

13th Apr. Shaastriya Chowk Meerut

11:30 a.m.

A	570	205	136	487	247	358	598	469	639	609

Combining all elements from auxillary arrays to A

Q_0	205	609
Q_1		
Q_2		
Q_3	136	639
Q_4	247	
Q_5	358	
Q_6	469	
Q_7	570	
Q_8	487	
Q_9	598	

A	205	609	136	639	247	358	469	570	487	598

Q_0		
Q_1	136	
Q_2	205	247
Q_3	358	
Q_4	469	487
Q_5	570	598
Q_6	609	639
Q_7		
Q_8		
Q_9		

A	136	205	247	358	469	487	570	598	609	639

The radix sort is based on the values of the actual digits in the positional representation of the nos. being sorted.

eg. The no. 235 in decimal notation is written with a 2 in 100th posn, 3 in 10th & 5 in 1st posn. The larger of two such integers of equal length can be determined as follows -

Start at the most significant digit & advance thru the least significant digit as long as the corresponding digits in the two nos. match. The no. with the larger digit in the 1st posn, in which the digits of two nos. do not match is larger of the 2 nos. Of course if all the digits of both nos. match, the nos are equal.

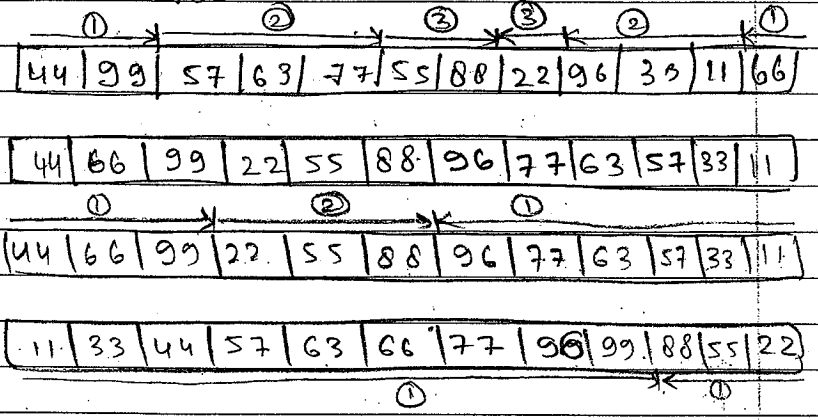
Two-way Merge-Sort

An alternate method to the divide & conquer based merge sort technique is the 2-way merge sort.

The 2-way merge sort is based on the principle "Burn the candle at both ends"

In a manner similar to the scanning procedure we have considered in quick sort.

In 2-way merge sorting, we examine the i/p list from both the ends, left & right and moving towards the middle



Garbage Collection

Suppose some m/o space becomes available coz a node is deleted from a list or an entire list is deleted from a prog. Clearly, we want the space to be available for future use. One way to bring this in use is to immediately re-insert the space into the free storage list. However this method may be too time consuming for the O.S of the comp.

For this purpose O.S of comp may periodically collect all the deleted space on to the free storage list, this is called garbage collection.

It usually takes place in 2 steps i.e. The comp. runs thru all lists, tagging those cells which are currently in use & then the comp. runs thru the m/o, collecting all untagged space on to the free storage list.

Overflow & Underflow

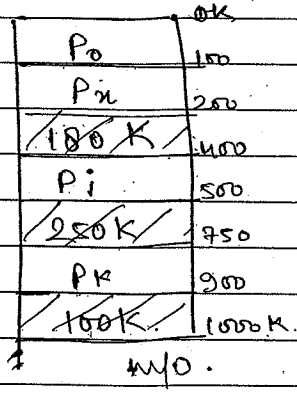
Sometimes new data are to be inserted into a D.S. but there is no available space i.e, the free storage list is empty, This is called overflow.

Similarly, the term underflow refers to the situation where one wants to delete the data from a D.S i.e. empty.

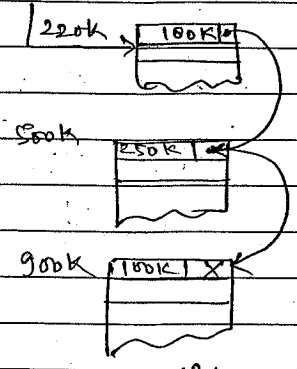
Compaction

The associated problem with dynamic storage allocation sys is that the areas of free m/o are scattered with the actively used partitions through out the m/o.

eg. Consider the sys whose m/o map is shown



Avail.



12th Apl When all requests for storage are of fixed size then it is just enough to link all unused block together into an available space list to serve the requests.

However, when storage requests are of varying sizes, it might appear quite sufficient.

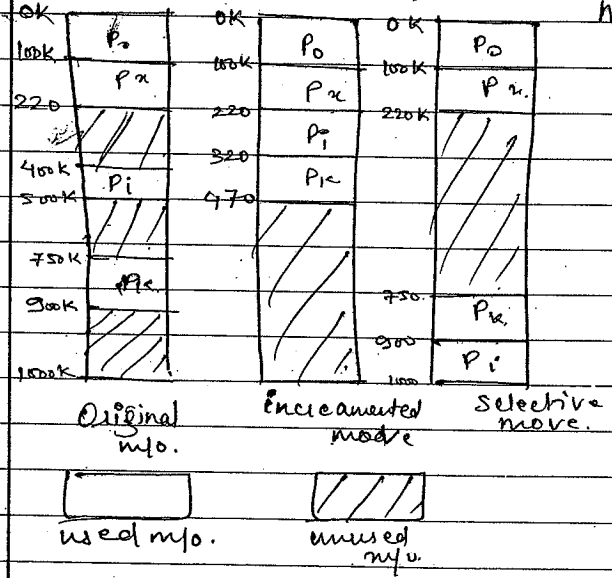
This problem appears due to the external fragmentation and it is commonly encountered in dynamic m/o mgmt sys. with variable size. When the m/o becomes seriously fragmented, the only out may be to relocate some or all portions into one of the m/o and thus combine the m/o into one large free area.

This technique for reclaiming storage is called compaction or defragmentation.

2 strategies for compaction -

i) Incremental compaction - In this strategy all the free blocks are moved into one end of the m/o to make a large hole

ii) Selective compaction - In this compaction, it searches for a min. no. of free blocks, the movement of which yields a larger free hole, this hole may not be at the end, it may be anywhere.



Right me bada - u
left me bada - u

DATE | | PAGE

Quick Sort (Only que not algo)

Q. 44, 33, 11, 55, 90, 40, 60, 99, 22, 88, 66

pivot

44, 33, 11, 55, 90, 40, 60, 99, 22, 88, 66

22, 33, 11, 55, 90, 40, 60, 99, 44, 88, 66

22, 33, 11, 44, 90, 40, 60, 99, 55, 88, 66

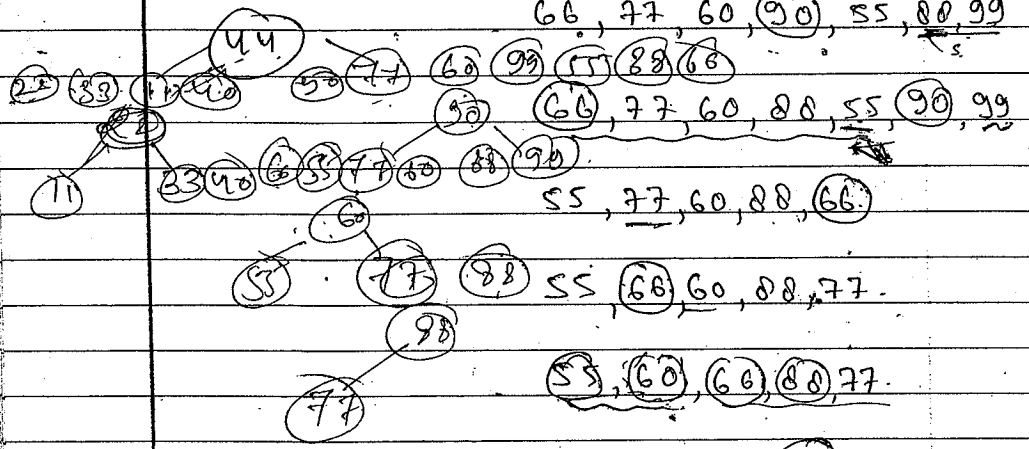
22, 33, 11, 40, 90, 44, 60, 99, 55, 88, 66

22, 33, 11, 40, 44, 90, 77, 60, 99, 55, 88, 66

11, 33, 22, 40 — 90, 77, 60, 99, 55, 88, 66

11, 22, 33, 40 — 66, 77, 60, 99, 55, 88, 90

66, 77, 60, 90, 55, 88, 99



55, 77, 60, 88, 66

55, 66, 60, 88, 77

55, 60, 66, 88, 77

55, 60 — 88, 77

77, 88

11, 22, 33, 40, 44, 55, 60, 66, 77, 88, 90, 99

DATE | | PAGE

Q. Use quick sort algo to sort the following list

Same 44, 33, 11, 55, 77, 90, 40, 60, 99, 22, 88, 66.

Soln The above given array is a list of 12 elements. Now apply quick sort algo to the list.

1. Selecting 44 as the pivot value.
(44), 33, 11, 55, 77, 90, 40, 60, 99, (22), 88, 66

Scanning the list from R to L, comparing each no. with 44 & stopping at the first no. less than 44, & interchange the two nos.

22, 33, 11, (55), 77, 90, 40, 60, 99, (44), 88, 66

2. Beginning with 22, scan the list in oppo. direction from L to R comparing each no. with 44 & stoping at first no. greater than 44 & interchange both.

22, 33, 11, (44), 77, 90, (40), 60, 99, 55, 88, 66

3. Beginning with 55 scan from R to L comparing each no. & stoping at the no less than 44, interchange both.

22, 33, 11, 40, (77), 90, (44), 60, 99, 55, 80, 66.

4. Beginning with 40, scan the list from LTR comparing each no. & stopping when 1st no. greater than 44 is found, interchange both.

22, 33, 11, 40, 44, 90, 77, 60, 99, 55, 80, 66.

5. Repeating the steps until all the nos. less than 44 are to the left & all nos. greater than 44 are to the right.

22, 33, 11, 44, (44), 90, 77, 60, 99, 55, 80, 66.

6. The above reduction step is repeated with each sub list containing two or more elements, then the list is sorted.

Recursion pg. 6.18.

Suppose P is a procedure containing either a call stmt to itself or a call stmt to a second procedure that may eventually result in a call stmt back to the original procedure P. Then P is called recursion procedure.

The program will not continue to run infinite. A recursive procedure must have the following 2 properties -

→ Their must be a certain criteria called base criteria for which procedure does not calls itself.

→ Each time the procedure does call itself directly or indirectly, it must be closure to the base criteria.

```
display (5);

void display (int x)
{ if (x==0)
    return;
  else
    pf("%d", x);
    display (x-1);
}
```

5,4,3,2,1

o/p

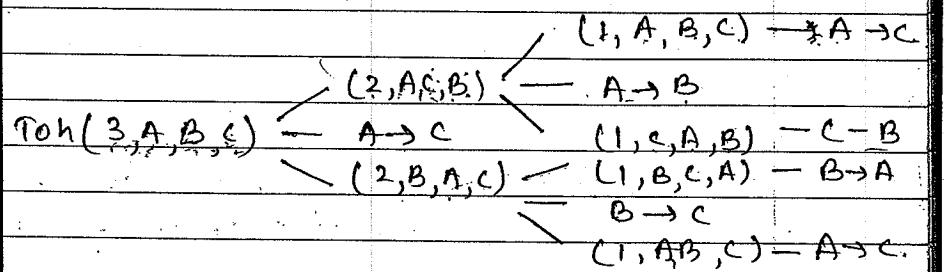
5 4 3 2 1

```
}
if change
:
else                                    O/P
display(x-1);                           1 2 3 4 5
pf("%d",x);
}.


void main()
{                                       O/P.
sum(5);                                 0
}                                       0
void sum(int x)                         1
{                                       2
static int s;                           3
if (x>0)                                4.
{
s = s+x;
x--;
sum(x);
}
pf("%d",x);
}.
```

DATE | | PAGE

Tower of Hanoi

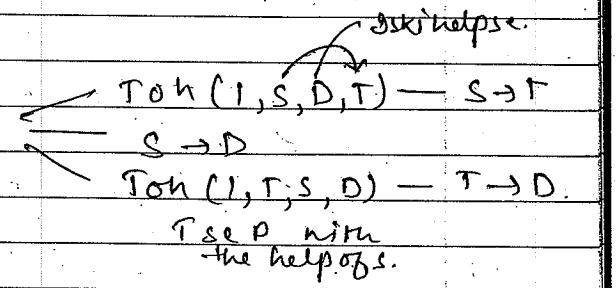
Kahan se kahan with the help of kiske.
A → C with the help of B.
B is in center.
(A, B, C).
Iski help se



for 1 disk.

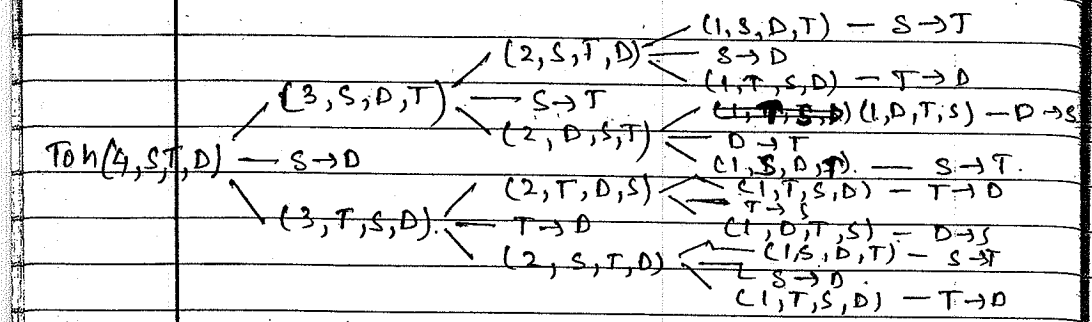
Toh (1, S, T, D) — S → D.

for 2 disk



pto

for 4 disks.



Algorithm

This algorithm gives a recursive sol^n to the tower of Hanoi problem for N disks.

1. If N=1, then:
   a) Write : SRC → DEST
   b) Exit.
   [end of if struct].
2. [Move N-1 disks from peg source to peg temp].
   Call TOWER (N-1, SRC, DEST, TEMP)
3. Write : SRC → DEST.
4. [Move N-1 disks from peg temp to peg dest]
   Call TOWER (N-1, TEMP, SRC, DEST)
5. Exit.

Tail recursion

A recursive call is

Tail recursion

A recursive call is tail recursive if it is the last stmt to be executed inside the func^n.

When the last thing a func^n (or procedure) does is to call itself. Such a func^n is called Tail recursive.

A func^n may make several recursive calls but a call is only tail recursive if the caller returns immediately after it.

eg. void display(int n).

```
{ if (n==0)
    return;
  pf("%d", n);}
  display(n-1); //tail recursion.
}
```

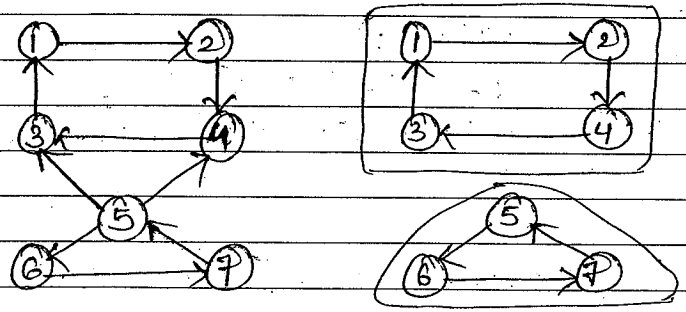
Translation of recursive procedure into non-recursive procedure.

pg. 6.27.

Connected Component

A graph G=(V,E) where V is a set of vertices of size N and E is set of edges of size M. The connected components of G are the sets of vertices such that all vertices in each set are mutually connected (reachable by some path) and no two vertices in different sets are connected.

Finding connected components is used in many diverse fields such as computer vision where pixels in a 2 or 3-D image are grouped in a region representing objects or phases of objects; spin models in Physics, VLSI circuit design, communication networks etc.




```
void connected (void)
{
    /* determine the connected components
       of a graph */
    int i;
    for (i=0; i<n; i++)
        if(!visited[i])
        {
            dfs
            pf("\n");
        }
}
```

Activity networks

There are 2-types of activity networks

i. AOV – Activity on vertex
ii. AOE – Activity on edge.

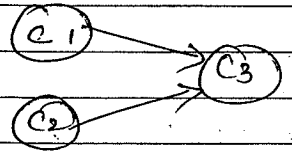
i. AOV

A directed graph G in which the vertices represents tasks or activities and the edges represent precedence relation b/w tasks is an activity on vertex network or AOV network.

eg: There are lists of courses needed for a CS branch in a university. Some of these courses may be taken independently of others. Other courses have pre-requisites and can be taken only if all the pre-requisites have already been taken.

The DS course cannot be started untill certain programming and math.

courses have been completed. Thus, pre-requisites define precedence relation b/w courses. The r/s defined may be more clearly represented using a directed graph in which the vertices represent courses and the directed edges represent pre-requisites.

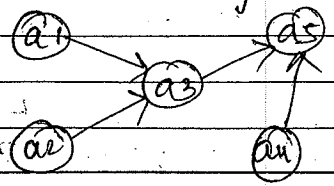


- AOV Network

Course no.	Course name	Pre-requisite
C1	C Programming	NONE
C2	Discrete Maths	NONE
C3	Data Structure	C1, C2.

ii) AOE

An activity network closely related to the AOV in the activity on edge or AOE network. The task to be performed on a project are represented by edges leaving a vertex cannot be started until the event at that vertex occured. An event occurs only when all activities entering it have been completed.



a_3 can be started only when a_1 and a_2 are completed and similarly a_5 is started when a_3 & a_4 are completed.

There are total 5 events where a_1, a_2 interpreted as a start activity and activities a_3 & a_5 cannot be started until events a_1, a_2 & a_4 cannot be completed.

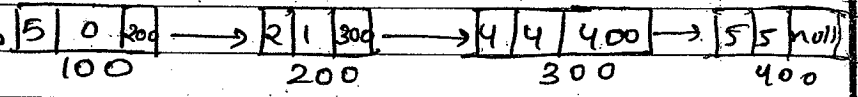
$P = 5 + 2x + 4x^4 + 5x^5$

$Q = 4 + 5x + 7x^2$

$R = 9 + 7x + 7x^2 + 4x^4 + 5x^5$

START 1

100



Min[old path, new path]

$\min(T_i)$

$w(i,j) =$

	R	S	T	U
R	7	5	0	0
S	7	0	0	2
T	0	3	0	0
U	4	0	4	0

$Q_0 =$

	R	S	T	U	
R	7	5	∞	∞	RR RS — —
S	7	∞	∞	2	SR — — SU
T	∞	3	∞	∞	— TS — —
U	4	∞	4	∞	UR — UT —

$Q_1 =$

	R	S	T	U	
R	7	5	∞	∞	RR RS — —
S	7	(12)	∞	2	SR SRS — SU
T	∞	3	∞	∞	TR Ts — —
U	4	(9)	1	∞	UR URS UT —

$Q_2 =$

	R	S	T	U	
R	7	5	∞	(7)	RR RS — RSU
S	7	12	∞	2	SR SS — SU
T	(10)	3	∞	(5)	TSR TS — TSU
U	4	9	(7)	(11)	UR US UST USU

$Q_3 =$

	R	S	T	U	
R	7	5	∞	7	RR RS — RU
S	7	12	∞	2	SR SS — SU
T	10	3	∞	5	TR TS — TU
U	4	(4)	1	(6)	UTR UTR UT UTU

$Q_4 =$

	R	S	T	U	
R	7	5	(8)	7	RR RS RUT RU
S	(6)	(6)	(3)	2	SUR SUS SUT SU
T	(9)	3	(6)	5	TUR TS TUT TU
U	4	4	1	6	UR US UT UU

26/3/19

"Hashing" (Unit-5)

→ We have seen different searching techniques where search time is basically dependent on the no. of elements. Sequential, binary search & all the search trees are totally dependent on no. of elements & so many key comparisons are involved.

Now our need is to search the element in constant time & less key comparisons should be involved. Suppose all the element are in array [N]. Let us take all the keys are unique & in the range 0-(N-1). Now we are storing the records in array based on the key where array index & keys are same. Now, we can access the records in const. time & there no key comparisons are involved. Eg→ Let us take 5 records where keys are -

9 4 6 7 2

It will be stored in array as -

arr

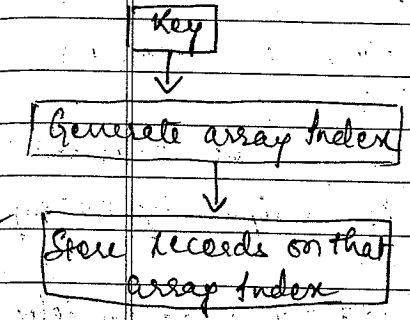
		2		4		6	7		9
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

Here, we can see the record which has value 2 can be directly accessed through array index arr[2].

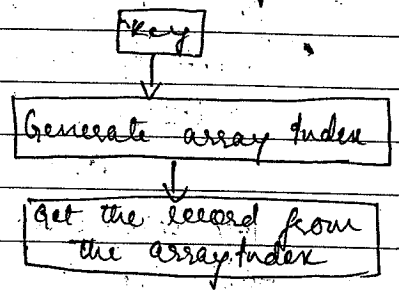
Now, the idea that comes in picture is Hashing where we will convert the key into array index & put the records in array & in the same way for searching the record, convert the key into array index & get the records from the array.

This can be described as -

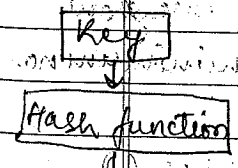
for storing record



for searching record



The generation of array index uses Hash(), which converts the keys into array index & the array which supports hashing for storing record & searching record called Hash table.



So we can say each key is mapped on a particular index through hash(). If each key is mapped on a unique hash table address then this situation is ideal situation, But there may be possibility that our hash() is generating same hash table address for different key, This situation is called collision.

→ Hash Functions - It when applied to the key, produce an integer which can be used as an address in a hash table. The intent is that elements will be relatively randomly & uniformly distributed. Perfect hash() is a function which when applied to all the members of set of items to be stored in a hash table, produces a unique set of integers within some suitable range. Such function produces no collisions.

: Good hash() minimizes the collisions by spreading the elements uniformly throughout the array.

The 2 principle criteria used in selecting a hash() $H : K \to L$ are as follows -

-1- The function H should be very easy & quick to compute.

-2- The function H should, as far as possible, uniformly distribute the hash addresses throughout the set 'L' so that there are a minimum no. of collisions.

→ There are basically 3 types of hash().

-1- Mid-Square Method - In this, the key K is squared & some digits or bits from the middle of this square are taken as address. Generally the selection of digits depends on the size of the table. It is important that same digits should be selected from the square of all the keys.

→ The Hash() 'H' is defined by $H(K) = L$, where L is obtained by deleting digits from both ends of K^2. Eg →

Suppose we have a table of size 1000 so we need 3 digit address.	Key:	1337	1273	1391
	Square of key:	1787569	1620529	1934881
	Address:	875	205	348

-2- Folding Method - The key K is partitioned into a no. of parts, $K_1, \ldots, K_r$, where each part except possibly the last has the same no. of digits as the required address. Then the parts are added together ignoring the last carry i.e. $H(K) = K_1 + K_2 + \ldots + K_r$

Eg → Suppose we have a table of size 1000 & we have to find a address for a 12 digit key. Since the address should be of 3 digits we will break the key in parts containing 3 digits

Eg → 738239456527

738 | 239 | 456 | 527 (dividing into 3 digits)

(Adding all) 738 + 239 + 456 + 527 = 1960. After adding & ignoring the final carry 1, the hash address for the key is "960".

→ The above technique is called shift folding.

→ Shift folding can be modified to get another folding technique called boundary folding. In this method, the key is assumed to be written on a paper which is folded at the boundaries of the parts of the key, so all even parts are reversed before addition.

Eg. 738 932 456 725
(reversed) (reversed)

After adding the key is "2851".
Now ignoring the final carry the hash address for the key is "851".

5- Division Method (Modulo division) - In this method, the key is divided by the table size & the remainder is taken as the address for hash table.

→ Choose a no. 'm' larger than the no. 'n' of keys in 'k'. (The no. m is usually chosen to be a prime no. or a no. without small divisors, since this frequently minimizes the no. of collisions). The hash (:) fn is defined by

H(K) = K (mod m) or H(K) = K (mod m) + 1

→ Here K (mod m) denotes the remainder when K is divided by m.

Collision Resolution Technique

→ Suppose we want to add a new record R with key 'k' to our file F, but suppose the memory location address H(k) is already occupied. This situation is called collision i.e. a collision occurs when more than one keys map to same hash value in the hash table. There are 2 techniques to resolve collisions -

(i) Collision resolution by open addressing (closed hashing)
In this the key which causes the collision is placed inside the hash table itself by at a location other than its hash address. Initially a key value is mapped to a particular address in the hash table.
→ If that address is already occupied then we will try to insert the key at some other empty location inside the table. The array is assumed to be closed & hence this method is named as closed hashing. To search for an empty location inside the table, there are 3 techniques -

(a) Linear probing - If address given by hash) is already occupied, then the key will be inserted in the next empty position in the hash table.
→ If the address given by hash table is 'A' & it is not empty, then we will try to insert the key at next location i.e. at address 'A+1'. If address (A+1) is also occupied then we will try to insert at the

next address (A+2) & we will keep on trying successive location, till we find an empty location where the key can be inserted. While probing the array for empty positions we assume that the array is closed or circular i.e. if array size is 'N' then after (N-1)th position, search will resume from 0th position of the array

Eg ⇒ Consider inserting the key – 29, 46, 18, 36, 43, 21, 24, into a hash table of size (m = 11) using linear probing consider the primary hash() is H(K) = K (mod m).

Initial state of hash table – T ⇒

0	1	2	3	4	5	6	7	8	9	10

1- Insert 29 - We know the linear probing hash() is H(K, i) = (h(K) + i) mod TSIZE

H(29, 0) = (29 % 11 + 0) % 11 = 7

2- Insert 46 ⇒

H(46, 0) = (46 % 11 + 0) % 11 = 2 ⇒ T[2] is empty

3- Insert 18 ⇒

T[7] is not free so, next probe sequence

H(18, 1) = (46 % 11 + 1) % 11 = 8 ⇒ T[8]

4- Insert 36 ⇒

H(36, 0) = (36 % 11 + 0) % 11 = 3 ⇒ T[3]

5- Insert 43 ⇒

H(43, 0) = (43 % 11 + 0) % 11 = 10 ⇒ T[10]

6- Insert 21 ⇒

H(21, 0) = (21 % 11 + 0) % 11 = 0 ⇒ T[0]

T →

0	1	2	3	4	5	6	7	8	9	10
21	54	46	36	24			29	18		43

7- Insert 24

H(24, 2) = (24 % 11 + 2) % 11 = 4 ⇒ T[4].

8- Insert 54

H(54, 1) = (54 % 11 + 1) % 11 = 1 ⇒ T[1]

Disadvantage - is primary clustering. When about ½ of the table is full, there is tendency of cluster formation i.e. groups of records stored next to each other are created. In the above eg. a cluster of indices 10, 0, 1, 2, 3, 4 is formed. If a key is mapped to any of these indices then it will be stored at index 5 & the cluster will become bigger. Suppose a key is mapped to index 10, then it will be stored at 5, far away from its home address. The no. of probes for inserting or searching this key will be 7.

(ii) Quadratic probing - In linear probing, colliding keys are stored near the initial collision point, resulting in formation of clusters.

In quadratic probing this problem is solved by storing the colliding keys away from the initial collision point. The formula for quadratic probing can be written as -

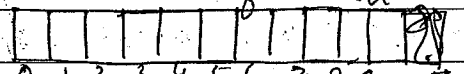
H(K, i) = (h(K) + i²) mod T size.

The value of 'i' varies from 0 to [T]size - 1 & H is the hash() . Here also, the array is assumed to be closed. The search for empty location will be in the sequence Eg -

Q: Consider inserting the key - 46, 28, 21, 35, 57, 39, 19, 50, into a hash table of size (m = 11) using quad. hashing where primary hash() is h(K) = K(mod m)

Sol: For quad. probing we have h(K) = K mod m 4.

Initial state of hash table is -



-1- Insert 46.

H(46,0) = (46 % 11 + 0²) % 11 = 2 → T[2] is empty

-2- Insert 28

H(28,0) = (28 % 11 + 0²) % 11 = 6 = T[6] is free

-3- Insert 21

H(21,0) = (21 % 11 + 0²) % 11 = 10 = T[10] is empty

-4- Insert 35

H(35,0) = (35 % 11 + 0²) % 11 = 2 T[2] is not empty

H(35,1) = (35 % 11 + 1²) % 11 = 3 T[3] is empty.

-5- Insert 57

H(57,0) = (57 % 11 + 0²) % 11 = 2 T[2] is not empty

H(57,1) = (57 % 11 + 1²) % 11 = 3 T[3] is not empty

H(57,2) = (57 % 11 + 4) % 11 = 6 T[6] is empty

-6- Insert 39

H(39,0) = (39 % 11 + 0²) % 11 = 6 T[6] is not empty.

H(39,1) = (39 % 11 + 1²) % 11 = 7 T[7] is empty

-7- Insert 19

H(19,0) = (19 % 11 + 0²) % 11 = 7 T[7] is not empty.

H(19,1) = (19 % 11 + 1²) % 11 = 8 T[8] is empty.

-8- Insert 50.

H(50,0) = (50 % 11 + 0²) % 11 = 6, T[6] is not empty.

H(50,1) = (50 % 11 + 1²) % 11 = 7, T[7] is not empty.

H(50,2) = (50 % 11 + 2²) % 11 = 10, T[10] is not empty.

H(50,3) = (50 % 11 + 3²) % 11 = 4, T[4] is empty.

Final Table →

0	1	2	3	4	5	6	7	8	9	10
57		46	35	50		28	39	19		21

Disadvantage - 1- This does not have the problem of primary clustering as in linear probing, but it gives another type of clustering problem known as 2° clustering

⇒ Keys that have same hash address will probe the same sequence of location leading to 2° clustering.

Eg⇒ In the above table, keys 46, 35 & 57 are all mapped to location 2 by the hash() 'H', & so the location that will be probed in each case are same.

46 → 2, 3, 6, 0, 7... / 35 → 2, 3, 6, 0, 7... / 57 → 2, 3, 6, 0, 7...

-2- Another limitation of this probe is that it can't access all the positions of the table. An insert operation may fail despite of empty locations in the hash tab

⇒ This problem can be eliminated by taking size of hash table to be a prime no.

(iii) Double hashing – In this, the increment factor is not constant as in linear or quadratic probing, but it depends on the key. The increment factor is another hash function & hence the name double hashing. The formula for double hashing can be written as –

$H(K, i) = (h(K) + i\,h'(K)) \bmod \text{Table size}$

Here, the 2nd hash function H' is used for resolving a collision. Suppose a record 'R' with key 'K' has the hash address $H(K) = h$ $H'(K) = h' \neq m$. Then we linearly search the locations with addresses $h, h+h', h+2h', h+3h', \ldots$

Eg ⇒ Consider inserting the keys 46, 28, 21, 35, 57, 39, 19, 50 into a hash table of size ($m = 11$) using double hashing. Consider that the hash() are $h(K) = K \% m$

$h'(K) = 7 - (\text{key} \% 7)$

Initially table is

0	1	2	3	4	5	6	7	8	9	10

– 1 – Insert 46

$H(46, 0) = (46 \% 11 + 0\,(7 - (46 \% 7))) \% 11 = 2$ → T[2] is empty

– 2 – Insert 28

$H(28, 0) = (28 \% 11 + 0\,(7 - (28 \% 7))) \% 11 = 6$

– 3 – Insert 21

$H(21, 0) = (21 \% 11 + 0\,(7 - (21 \% 7))) \% 11 = 10$ = T[10] is empty

– 4 – Insert 35

$H(35, 0) = ((35 \% 11 + 0\,(7 - (21 \% 7))) \% 11 = 2$

$H(35, 1) = ((35 \% 11 + 1\,(7 - (35 \% 7))) \% 11 = 10$

Final

0	1	2	3	4	5	6	7	8	9	10
		46			35	28				21

$H(35, 2) = (35 \% 11 + 2\,(7 - 35 \% 7)) \% 11 = 0$

– 5 – Insert 57

$H(57, 0) = (57 \% 11 + 0\,(7 - 57 \% 7)) \% 11 = 2$

$H(57, 1) = (57 \% 11 + 1\,(7 - 57 \% 7)) \% 11 = 9$

– 6 – Insert 39

$H(39, 0) = (39 \% 11 + 0\,(7 - 39 \% 7)) \% 11 = 6$

$H(39, 1) = (39 \% 11 + 1\,(7 - 39 \% 7)) \% 11 = 10$

$H(39, 2) = (39 \% 11 + 2\,(7 - 39 \% 7)) \% 11 = 3$

– 7 – Insert 19

$H(19, 0) = (19 \% 11 + 0\,(7 - (19 \% 7)) \% 11 = 8$

– 8 – Insert 50

$H(50, 0) = (50 \% 11 + 0\,(7 - 50 \% 7)) \% 11 = 6$

$H(50, 1) = (50 \% 11 + 1\,(7 - 50 \% 7)) \% 11 = 2$

Separate Chaining –

→ In this method, linked list are maintained for elements that have same hash address.

→ Here the hash table doesnot contain actual keys & records but it is just an array of pointers, where each pointer points to a linked list.

→ All elements having same hash address 'i' will be stored in a separate linked list & the starting address of that linked list will be stored in the index 'i' of the hash table.

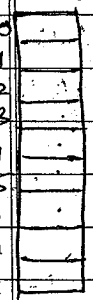
→ So, array index 'i' of the hash table contains a pointer P to the list of all elements that share the hash address 'i'.

→ These linked lists are referred to as chains &

hence, the method is named as separate chaining

Eg => Let us consider the insertion of elements 5, 28, 19, 15, 20, 33, 12, 17, 10 into a chained hash table. Let us suppose the hash table has 9 slots & the hash() is $H(k) = k \pmod 9$.

Soln



→ Initial state of table.

-1- Insert 5

$H(5) = 5\%9 = 5$ / T[5] is empty.

-2- Insert 28

$H(28) = 28\%9 = 1$ /.

-3- Insert 19

$H(19) = 19\%9 = 1$

-4- Insert 15

$H(15) = 15\%9 = 6$

-5- Insert 20

$H(20) = 20\%9 = 2$

-6- Insert 33

$H(33) = 33\%9 = 6$

-7- Insert 12

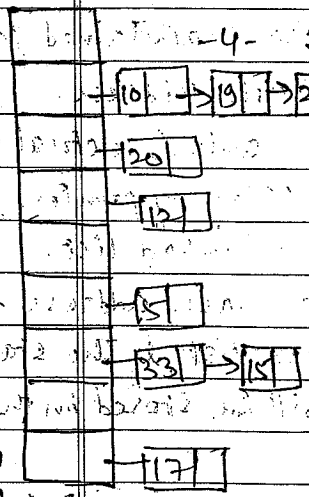
$H(12) = 12\%9 = 3$

-8- Insert 17

$H(17) = 17\%9 = 8$

-9- Insert 10

$H(10) = 10\%9 = 1$



Final state of hash table;

Diff. bet. open addressing & separate chaining

-1- In open addressing, accessing any record involves comparisons with keys which have different hash values which increased the no. of probes. In chaining comparisons are done only with keys that have same hash values.

-2- In open addressing, all records are stored in the hash table itself, so there can be problem of hash table overflow & to avoid this, enough space has to be allocated at the compilation time. In separate chaining there will be no problem of hash table overflow because linked lists are dynamically allocated so, there is no limitation on the no. of records that can be inserted.

=> Separate chaining is best suited for applications where the no. of records is not known in advance.

-3- In open addressing, it is best if some locations are always empty. In chaining, there is no wastage of space because the space for records is allocated when they arrive.

-4- Implementation of insertion & deletion is simple in separate chaining but complex in open addressing.

-5- In separate chaining the load factor denotes the average no. of elements in each list & it can be greater than 1, but in open addressing load factor is always less than 1.

* load factor.