INDEX

Name: Kopadia Dhruv J.

Std: Div.: Roll No.:

School:

No.	Subject	Date	Page No.	Remark

1.

* Data structure is a particular way of organising a data in a computer. So it can be use efficiently.

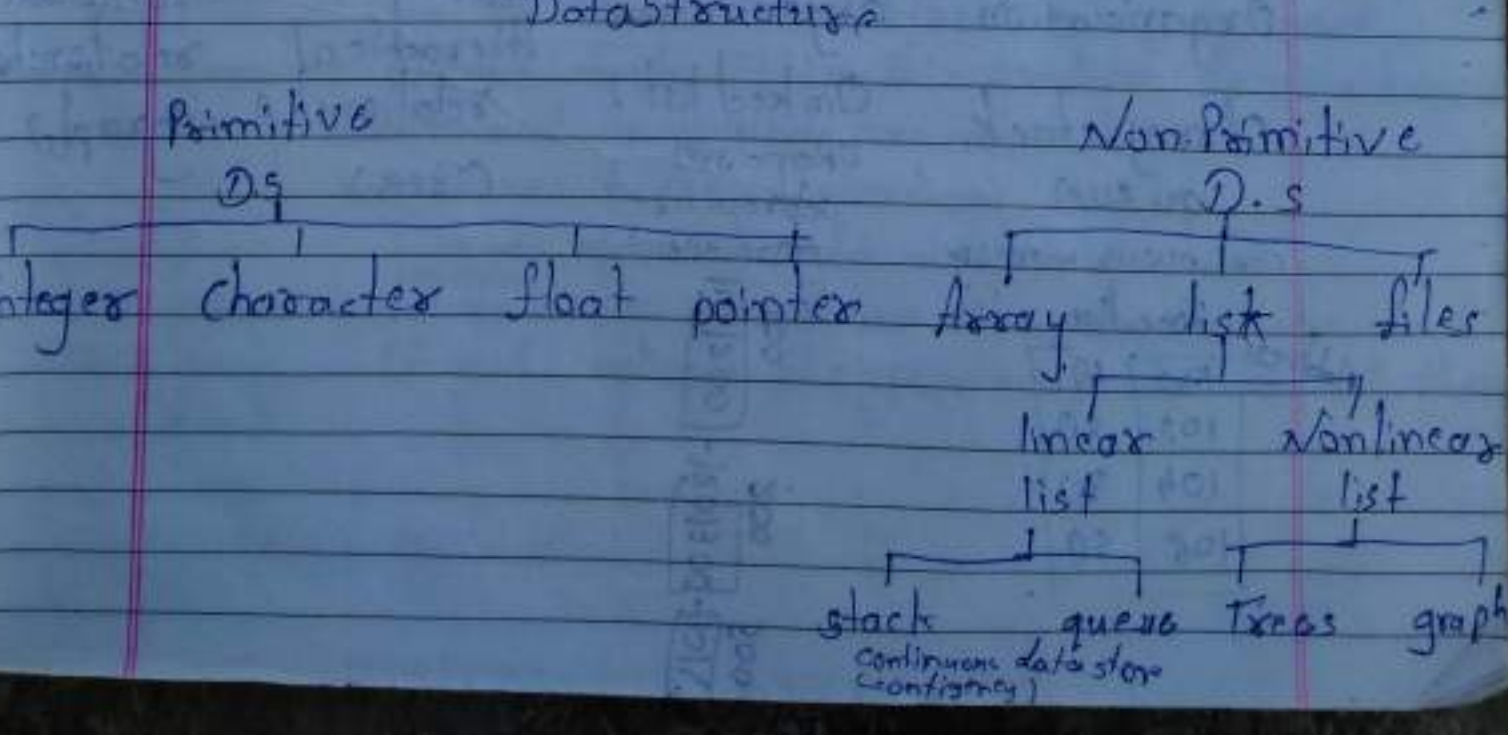
* There are two types of data structure:

- primitive datastructure
- non-primitive "

* Primitive Datastructure - P.D are provided by programming language as a basic building blocks. These are also known as buildtin type or basic type

* Non-primitive Datastructure - The datastructure which is derived from primitive data-structure or primary data structure are know N.P.D

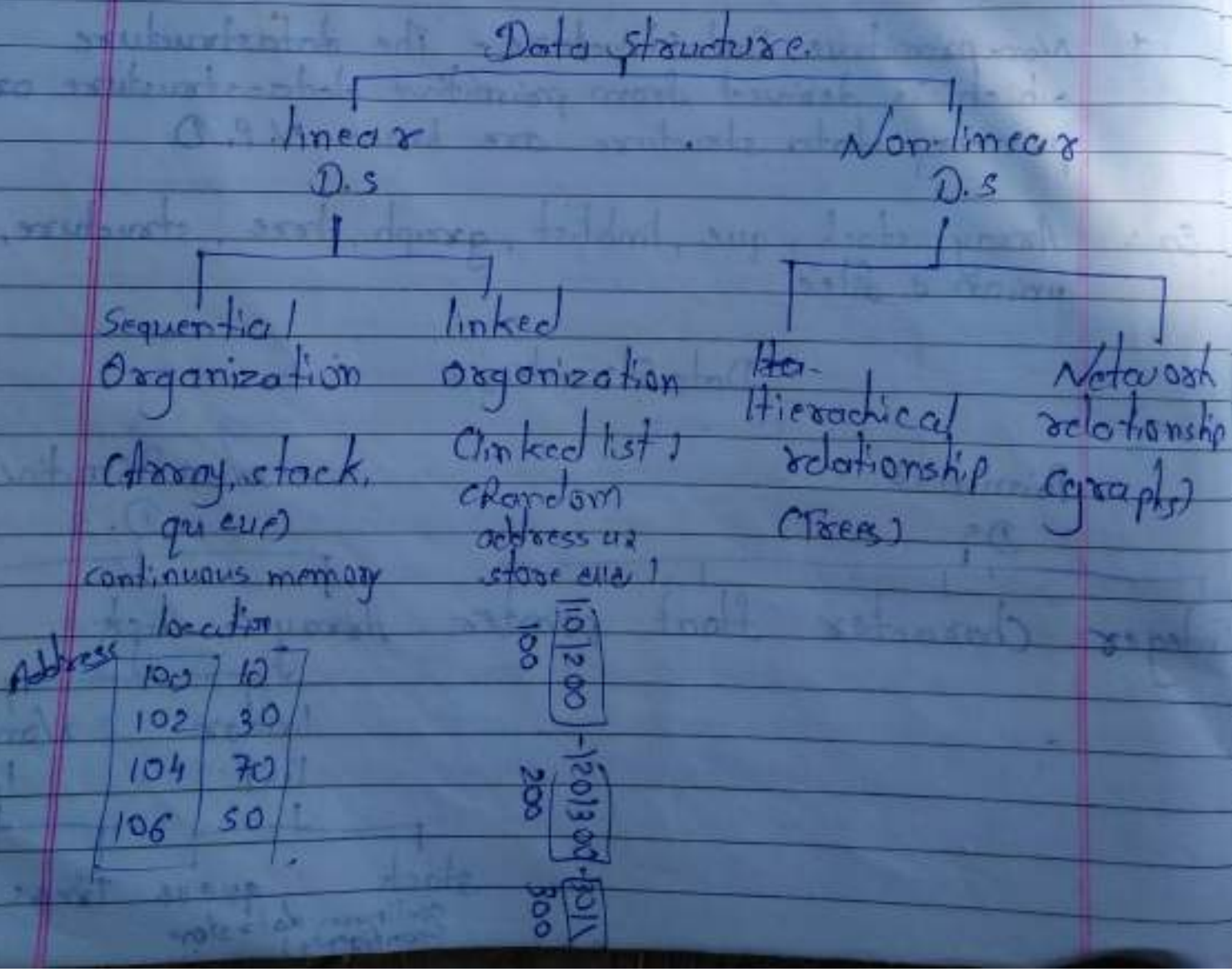
Ex:- Array, stack, que, linklist, graph, tree, structure, union & files



- Non-primitive D.s are also known as Abstract D.s or Abstract Data type.

Notes - * All datastructure allow us to perform different operation on data. We select this D.s based on which type of operation is required

* There are (two) 2 types of D.s:
- linear D.s
- Non linear D.s



* Linear D.S :

In L.D.S the elements are stored in sequential order it means if we know the address of first data item when we can retry the secondone and so on

* Sequential Organization : In S.O element occupied continuous or consigutive memory location

* linked list : It In this D.S components are exist in a certain sequence but they are not necessarily stored in consigutive memory location
- It uses a pointer which shows the relationship between the element with the help of links

- Stack follows a concept of LIFO (last in first out).
- stack can be represented as a linear array.
- A Every stack has a variable TOP associated with it
- TOP is use to store the address of TOP most element of the stack. It is this position from where the element will be added or deleted.
- Stack has three basic operations
(i) push - insert element into stack
(ii) pop (remove) - delete " from the top of the stack
(iii) peep - It will indicate the topmost element of the stack

Overflow condition : TOP ≥ max TOP ≥ max-1
underflow " : TOP = 0 or TOP = -1

* Disadvantages:
- slow access to the other element
- memory wastage.

* Applications:
(ii) Reverse a string (word) eg:
(iii) conversion of expression infix to prefix ta
infix to postfix att
postfix to infix
prefix to infix
(iii) parsing

* Queue :-

- Queue is a FIFO data structure in which the element that was inserted first is the first one to be taken out.
- The element in a queue are add one end called rear & remove from the other end called front

* Two operation perform:
Enqueue - insert, adding
dequeue - deleting, remove

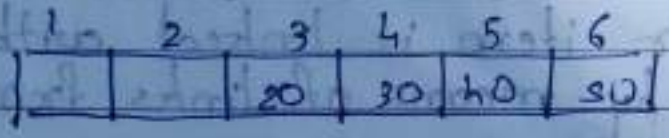
* Three Types of queue:
(i) Simple queue rear ≥ max (iii) double ended queue
(ii) Circular queue f == 1
r ≥ max f = r = = 0 (empty)

* Application:
- printer
- people checkout from supermarket
- callcentre phone system
- CPU task scheduling
- Interrupts are handle in the same order as they come

* Disadvantages:-
- Slow access to the other item

Circular queue:

overflow condition :- $f == 1$ && $r >= max$
underflow condition :- $f = r = 0$

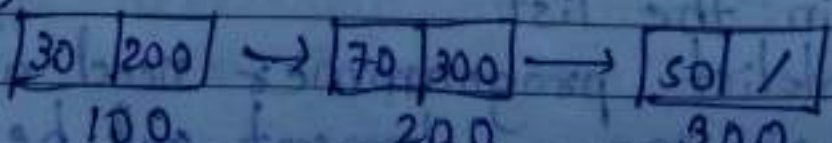


* linkedlist:

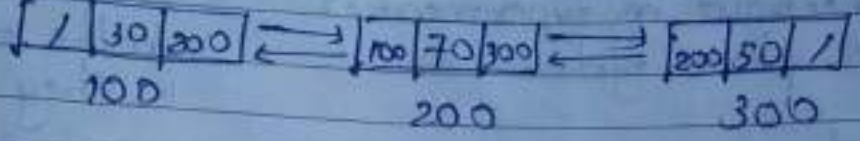
⇒ 3 types of linkedlist:
- Singly linkedlist
- doubly "
- circular "

There are 2 part:
- nodes part
- Address "

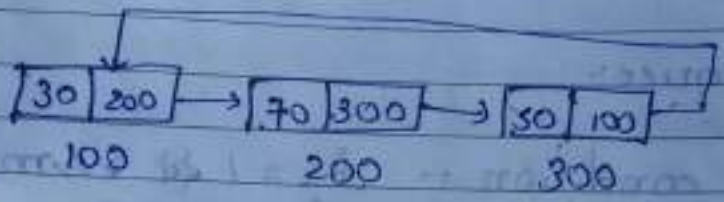
(1) Singly linked list:



(ii) doubly linked list:



(iii) Circular linked list:-



- A link list is a collection of item in which each particular item is linked with other & order is given by means of links from one item to other.
- Each item is denoted by node. which consist of two field

value (element) link (address of other element)

Advantages:

- It is a dynamic data structure in which the element can be added or deleted from any where in the list.
- In linklist programmer need not worry about how many element will be stored in

linked list.

- In L.L each element is allocated memory as it is added to the linked list.

Application:

- It is use to implement stack, queue, tree, graph.

* Disadvantages:

- slow search operation & require more memory.
- reverse traversing is difficult.

Trees:

- It is a non-Linear Datastructure in which data item called as node are arranged in some sequence.

Advantages:-

- Item can be located very quickly in tree.
- Insertion is easy.

Disadvantages:

- Deletion algorithm is complex

Application:

- telephone derectory assistant information store in tree so name & no. can be quickly find out.

* Graph:

- Graph is basically a collection of vertices & edges that connect this vertices
- It is a generalization of tree structure where nested of having purely parent to child relation any kind of complex relation bet vertices can be represented
- Graph doesn't have any root node. Every node in the graph connected with other node
- Tree has a constrain that node can have many children but only one parent. But graph doesn't have any restriction

* Applications:

- node can be represent city & edges can represent the road CIT can be use in GPS)
- A graph can also use to represent a comp. network where nodes are workstation & edges are n/w connection CIT can be used in work station)

* Disadvantages:

Algorithams are slow & very complex

* Performance Analysis :- select best algorithm

- Performance analysis of an algorithm is the process of calculating a space required by the algorithm & time required by the algorithm
- Performance analysis of an algorithm is performed by using the following measure.

(i) space space required to complete the task of ~~that~~ an algorithm
(ii) time required " " " " " the algorithm

* Space complexity:

- Tota amount of computer memory required by an algorithm to complete it's execution is called as S.C of that algorithm.
- for any algorithm memory requires for following purposes

(i) memory required to store program instruction ~~return statement~~
(ii) " " " " constant value &
" " " " variable value.
c=1

Example:

```
int square(int a)    a require 2 bytes
{
    return a*a;      return " 2 bytes
}                              4 bytes
```

constant space complexity

int a=10
a=10010 } byte=4

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

operation

depend input values

* Time complexity: (T.C)

The T.C of that algorithm is the total amount of type time required by an algorithm to complete its execution.

- To calculate the 'T.C' we check only how our program is behaving for the different input value to perform all the operation like arithmatic, logical, relational, conditional, Assignment operation, return values etc.

- If any program require fix amount of time for all i/p values then it's time complexity is said to be constant time complexity.

- If the amount of time required by an algorithm is increased with the increase of i/p values then that time complexity said to be 'Linear T.C'

best case = ω
avera

Asympatic notation of algorithm is a mathamatical representation of complexity.

(i) Big-oh(O) — normally represent. worst case
(ii) Big-omega (Ω) best case
(iii) Big-theta (Θ) Average case

* Big-oh :

- Big-oh notation always indicates maxm time required by algorithm which describe the worst case of an algorithm time complexity.

$f(n) = O(g(n))$

actual time — max^m time

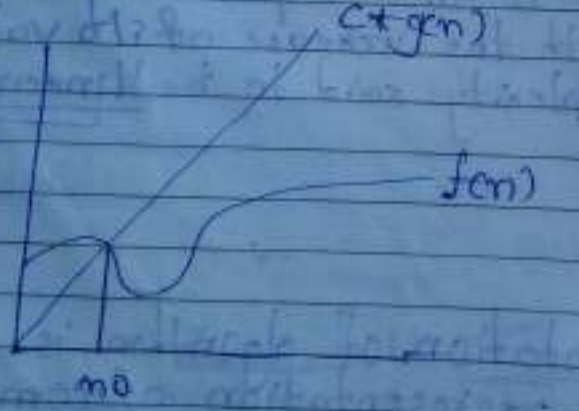
$f(n) \leq c * g(n)$ where

} worst case

$f(n)$ = algorithm runtime $n \Rightarrow n_0$

$g(n)$ = arbitary T.C you are trying to relate to your algo.

c = some real constant



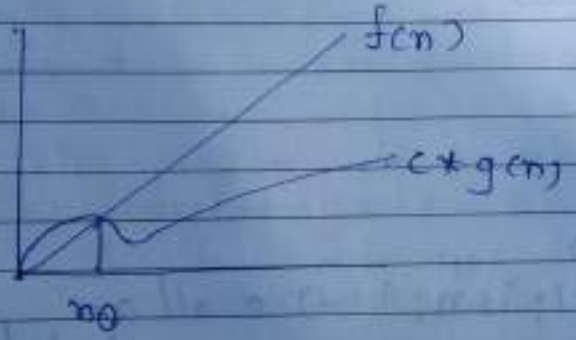
$f(n) = \Omega(g(n))$

actual time min type
$f(n) \geq c * g(n)$

ex. $f(n)$ = algo. return $n \geq n_0$

$g(n)$ = arbitary time complexity you are trying to relate to your algo.

c = some real ~~or~~ constant



It's always indicate minm time required by algorith which describle the ~~average~~ best case of an algo. time complexity.

Big-theta

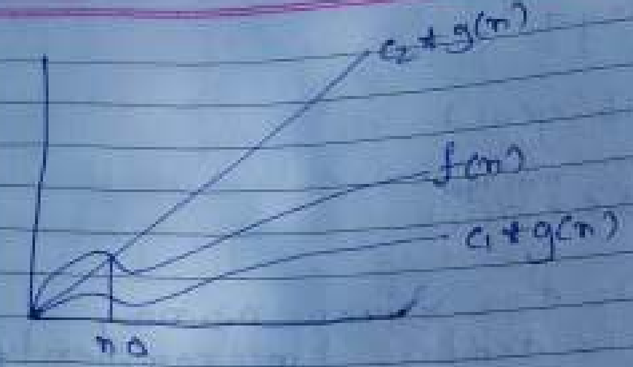
$f(n) = \Theta(g(n))$

worst case best case
$0 \leq c_1 * g(n) \leq f(n) \leq c_2 * g(n)$

$f(n)$ = algo. runtime $n \geq n_0$.

$g(n)$ = arbitary time complexity you are trying to relate to your algo

c_1, c = some real constant.



2.

Time complexity:

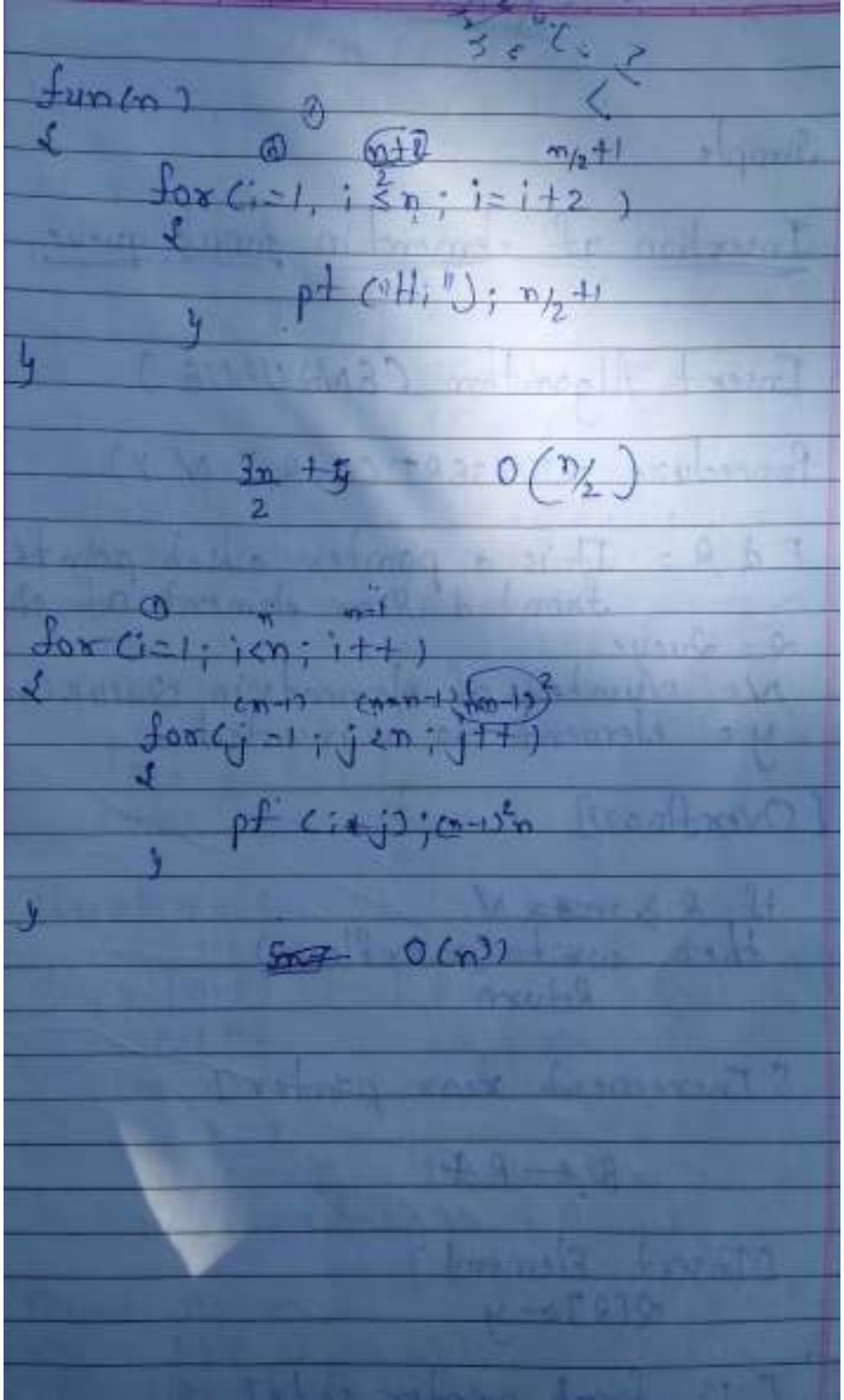
Example:

3

```
fun(n)
{
    for(i=1; i<=n; i++)   • all are calculated
    {
        pf("Hello"); n
    }
}
```

= 1+n+1+n+n [3n+2] — time

O(n) — Below of n time and task complete and



```
fun(n)
{
    for(i=1; i<=n; i=i+2)
    {
        pf("Hi");
    }
}
```

$\frac{3n}{2} + 5 \quad O(n/2)$

```
for(i=1; i<n; i++)
{
    for(j=1; j<n; j++)
    {
        pf(i*j);
    }
}
```

$O(n^2)$

Queue

Data structure

(1.) Simple

Insertion of element in simple queue:

Insert Algorithm (ENQUEUE)

Procedure QINSERT (Q, R, F, N, Y)

F & R = It is a pointer which points to front & Rear element of Queue.

Q = Queue

N = Number of elements in Queue. (Size of Que

y = element to be inserted

(i) [Overflow?]

if R ≥ ~~max~~ N
then write (Overflow)
Return

(ii) [Increment rear pointer]

R ← R+1

(iii) [Insert Element]

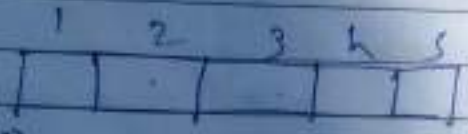
Q[R] ← y

(iv) [Is front pointer set?]

if F = 0

then F ← 1
Return

(i) N = 5



F=0 R=0

(ii) Y = 10

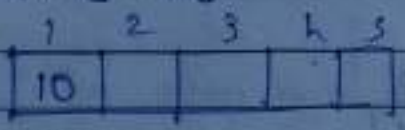
x >, 5

0 >, 5 false

⇒ x = x + 1

x = 1

⇒ a[1] = 10 ⇒ P = 0 ⇒ F = 1



F=1 x=1

(iii) Insert Y = 5

F=1 x=2

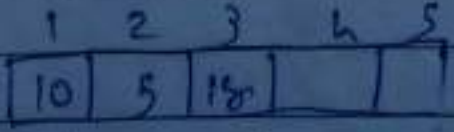
⇒ x >= N ⇒ 1 >= 5 false

⇒ x = 2

⇒ a[2] = 5

⇒ f = 1 ⇒ false

(iii) Insert Y = 18



F=1 x=3

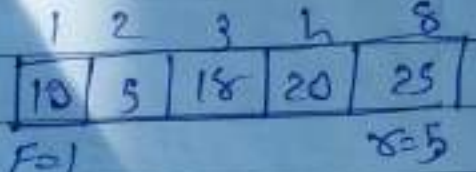
3 ≥ 5 false

⇒ r = r + 1 = 4

⇒ Q[4] = 20

⇒ F = 1 false

(v) Insert Y = 25



r ≥ N

⇒ 4 ≥ 5 false

⇒ r = 5

⇒ Q[5] = 25

⇒ f = 1 ⇒ false

(vi) r ≥ N

5 ≥ 5 ⇒ True

Overflow

DELETE Algorithm (Dequeue)

Procedure DELETE (Q, F, R)

F & R = It is a pointer which points to front & Rear elements of Queue

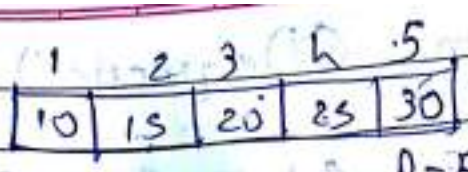
Q = Queue

y = temporary variable

[Underflow ?]

if F=0 (F=-1) {starts with 0}
then write ('UNDERFLOW')
RETURN(0)

[Delete element]
y ← Q[F]



Deletion :-

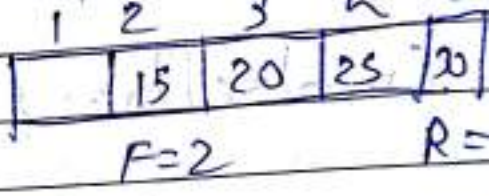
(i)

(i) $F = 1$

$1 \neq 0$

(ii) $y = Q[1]$

$y = 10$



(iii) $F = R \Rightarrow 1 \neq 5$

$F = 2 \quad (F+1)$

$y = 10$

(ii)

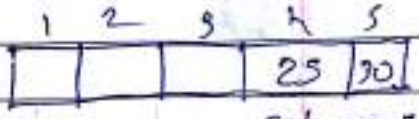
(i) $F = 2$

(iii) (i) $F = 3$

$3 \neq 0$

(ii) $Y = \alpha[3]$

$Y = 20$



F=4 R=5

(iii) $F = R \Rightarrow 3 \neq 5$

$F = 4$ (F++)

(iv) $Y = 20$

(iv) (i) $F = 4 \Rightarrow 4 \neq 0$

(ii) $Y = \alpha[4] = 25$

F=5 R=5

(iii) $4 \neq 5$

$F = 5$ (F++)

(iv) $Y = 25$

(v) (i) $F = 5 \Rightarrow 5 \neq 0$

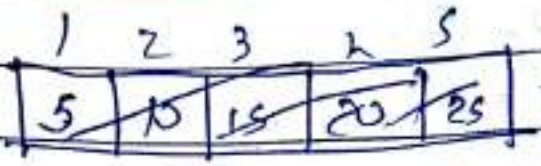
(ii) $Y = \alpha[F] = \alpha[5]$

$Y = 30$

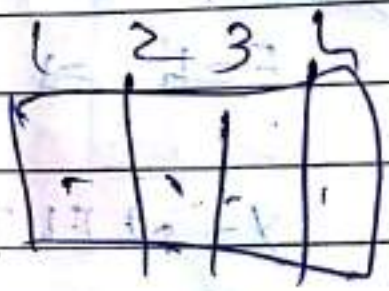
F=0 R=0

(iii) $5 = 5$

$F = 0$ $R = 0$



Consider the simple queue given below with f
four memory cell which has front & rear p
zero.
draw the simple queue structure and repres
the position of front & rear object for ea
of the following operation



Insert A, B, C
Delete A, B
Insert D, E

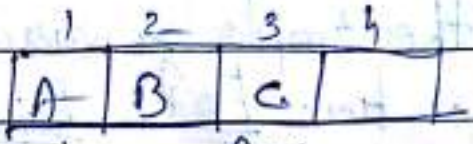
R=F=0

2>, 4.

x = 3

a[3] = C

F ≠ 0



F ≠ 0

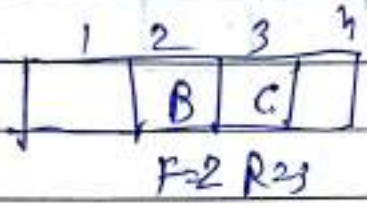
⇒ ~~a[F]~~ y = a[F]

y = a[1]

y = A

~~F = 0~~ F ≠ R

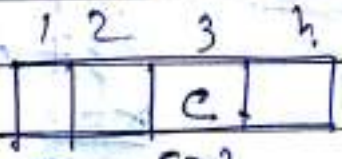
⇒ F = 2



y = a[2]

y = B

F ≠ R

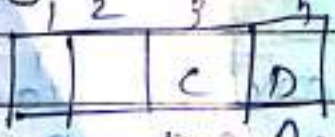


~~x~~ 3 > 4

x = 4

y = a[4]

y = D



4 > 4 condition true

condition overflow.

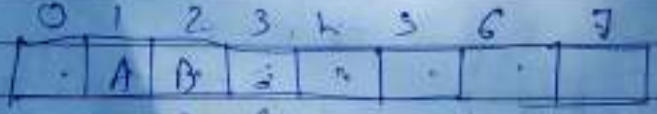
Q= If queue starting from 0 then
Overflow condition $r \geq max-1$
$F = -1$

VALENTINE
Page No.:
Date: / /

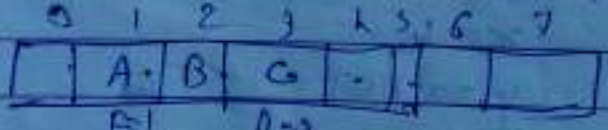
* Consider the simple queue given below & memory cell. which has F=1 R=2 draw the simple queue structure & represent the position of front & Rear point for each of the following operation

(i) Insert C & D
(ii) Delete one alphabet
(iii) Insert E & F
(iv) Delete one alphabet
(v) Insert G & H

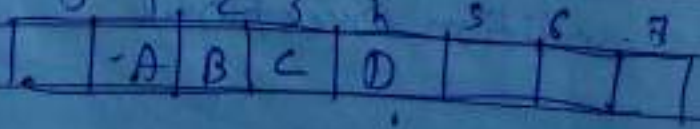
(i)



step-1 $r \geq 7$
$2 \geq 7$ condition false
$r = 3$
$Q[3] = C$
$F = 1$
$\Rightarrow 1 \neq 0$ condition false



step-2 $r \geq 7$
$3 \geq 7$ condition false
$r = 4$ $Q[4] = D$
$F = 1$
$1 \neq 0$ condition false



step=3

F=1

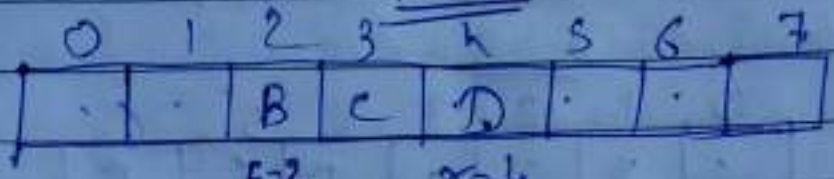
1≠-1

y=a[1]

y=A

1≠4 condition false

F=2



step=4

x≥7

4≥7 condition false

x=5

a[5]=E

F=2

2≠-1 condition false

step=5

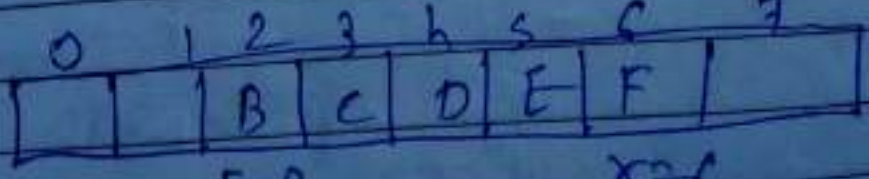
5≥7 condition false

x=6

a[6]=F

F=2

2≠5 condition false



step-6

f=2

2≠-1

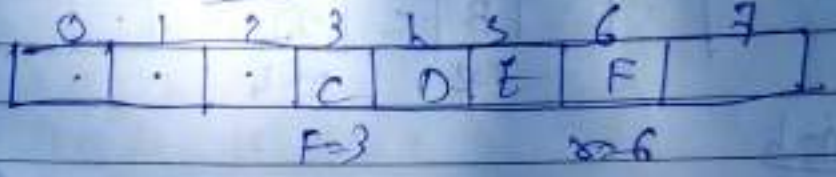
~~f=2~~

~~a[2]=B~~

a[2]=B

2≠5 condition false.

f=3



step-7

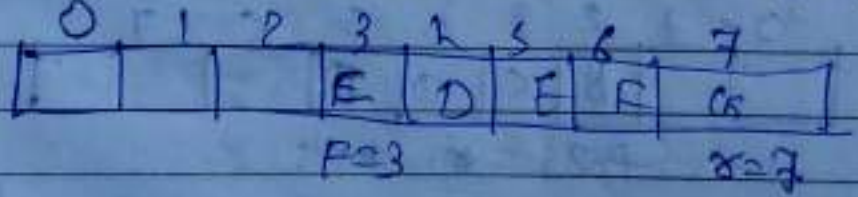
6≥7 condition false

r=7

a[7]=G

f=3

3≠-1 condition false



step-8

7≥7 condition false

Queue Overflow

N = number of element in queue
Y = element to be inserted

Algorithm :

[Reset rear pointer ?]
if R=N
then R ← 1
else R ← R+1

[Overflow ?]
if F=R
then write ('Overflow')
Return

∴ R = 1

(ii) F = 0

which is ~~not~~ F ≠ R.

F = 1

R = 1

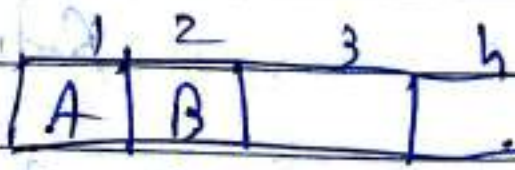
(iii) a[1] = A

(iv) F = 0 then F = 1

(i) R = 1

1 ≠ 4 then

R = 2



(ii) F = 1

i.e. F ≠ R

F = 1 R = 2

(iii) a[2] = B

(iii) (i) R = 2

2 ≠ 4 then

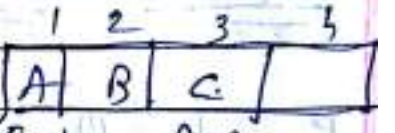
R = 3

(ii) F = 1

i.e F ≠ R

(iii) a[3] = c

(iv) F = 1 → 1 ≠ 0



(iv) (i) R = 3

3 ≠ 4 then

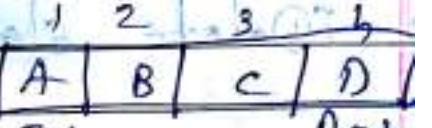
R = 4

(ii) F = 1

i.e F ≠ R

(iii) a[4] = D

(iv) F = 1 → 1 ≠ 0



(v) (i) R = 4

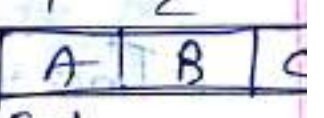
i.e R = N

then R = 1

(ii) F = 1

∴ F = R

∴ stack is overflow



(Q) DELETE (F, R, Q, N)

[Underflow ?]

if f = 0
then write (UNDERflow)
Return (0)

[Delete element]

y ← Q[F]

[Queue empty ?]

if F = R
then F ← R ← 0
Return (y)

[Increment front pointer]

if F = = N
F ← 1
else
F ← F + 1
return (y)

(ii) $y = Q[1]$

$y = A$

1	2	3	4
	B	C	D

F=2 R=4

(iii) $F \neq R$

$1 \neq 4$

then $F \neq N$

$1 \neq 4$

the F=2

Delete A

(i) ~~4 = 4~~ $R = N$

$4 = 4$

$R = 1$

1	2	3
E	B	C

R=1 F=2

(ii) $R \neq F$

$1 \neq 2$

(iii) $Q[1] = E$

(iv) $F = 2$ $2 \neq 0$ condition false

Insert C

a) F=2 ⇒ 2≠0
condition false

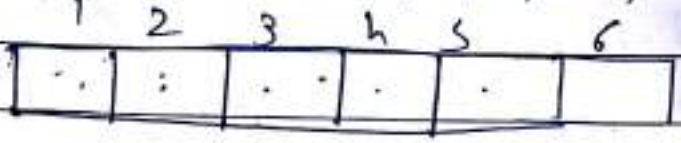
c) y=a[2]
y=B

c) F≠R
2≠1
then

Ex: 1 Consider the CQ given below with 6 me
cell which has front & rear = 0 Draw th
circular queue structure & represent the
position of front & real pointers for each
the following operation

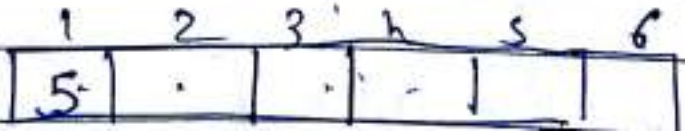
(A) Insert 5, 10, 15, 20, 25
(B) Delete 5, 10
(C) Insert 30, 35, 40
(D) Delete 15, 20, 25
(E) Insert 45, 50, 55
(F) Delete 30, 35, 40, 45, 50, 55

⇒



F=0
R=0

(1) (i) R=0, N=6
0 ≠ 6
condition false
R=1
(ii) 0 ≠ 1
condition false
(iii) Q[1] = 5
(iv) F=0 ⇒ F=1



(ii) ⇒ R=1, N=6

1 ≠ N

condition false

R=2.

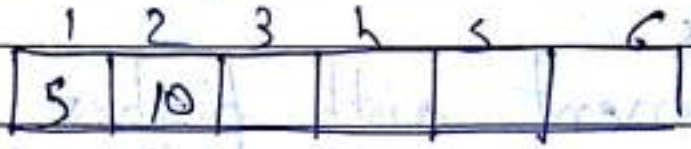
⇒ R=2, F=1

2 ≠ 1

condition false

⇒ A[2] = 10

⇒ 1 ≠ 0 false



F=1 R=2

(iii)

* Priority Queue:

- It means Priority Queue is an abstract data in which each element is assigned a priority.
- The priority of the element will be used to determine the order in which this element will be proceed
- The general rule of processing the element in P.Q is
  (i) the element with higher priority is proceed before an element with lower priority
  (ii) two element with same priority are process on a first come first serve basis
  (iii) ~~It is~~
  (iv) It is a queue in which we are able to insert or remove item from any position based on some priority is often refer to as a priority queue
  (v) P.Q are widly used in O.S to execute the highest priority first processed

Implementation of priority queue

There are 2 ways to implement priority queue
(i) sorted list: element stored in sort order
(ii) Unsorted list:

insertion O(n)
deletion O(1)

deletion O(n)
insertion O(1)

sorted list: (ii) unsorted li

Insertion O(n) deletion O(n)

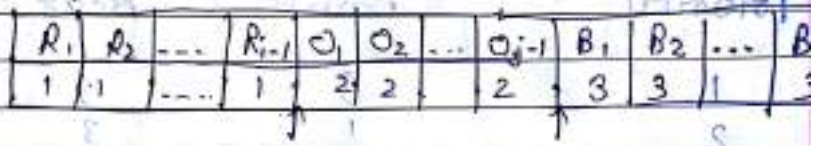
deletion O(1) Insertion O(1)

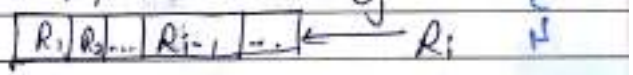
This can be implemented using linklist-

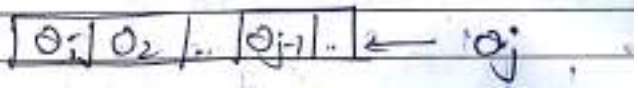
list

ing

link list :-



Priority 1 

Priority 2 

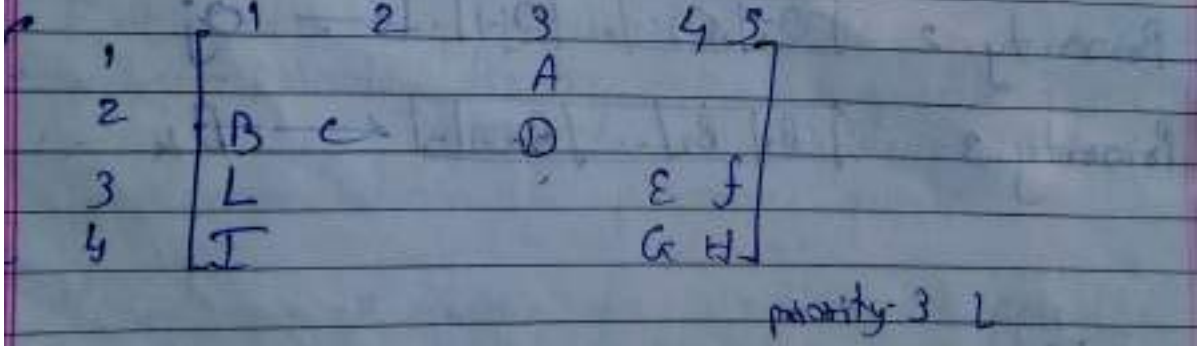
Priority 3 

a single sequential storage structure is
for the priority queue the insertion of n
element must be placed in the middle of
This can require a movment of several el

It is better to split the PQ into diff^n d
each having its own storage structure

Array representation of priority queue (

Priority	front	Rear
1	3	3
2	1	3
3	4	5
4	4	1



To insert a new element with priority

Delete:

To delete an element we have to find the first non empty queue and then process the front element of first a non empty queue

* DEQUES (double ended queue)

- A dequeue is a linear list of element into which element may be inserted & deleted from both ends
- It is also known as head tail linklist. because
- element can be deleted & inserted from both end.
- No element can be added & deleted from the middle.
- In computer memory dqueue is implemented as CQArray or doubly linklist.

There are two types of DEQUES:

(1) input restricted dqueue
(2) output restricted dqueue

double Ended Queue

both insertion & deletion are possible.

(i) Input restricted

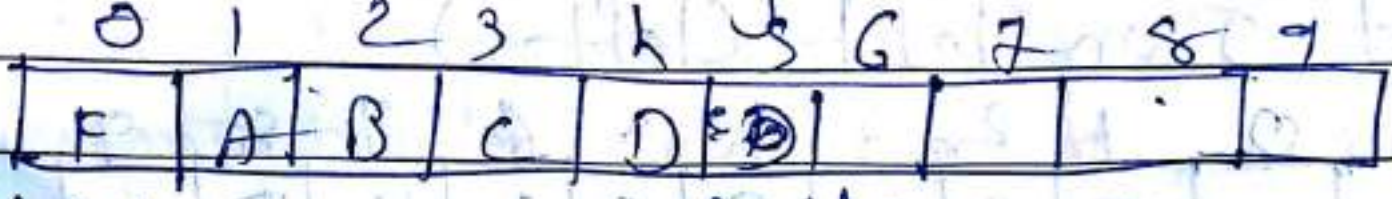
(ii) Output restricted

* Consider a DQ given below with 10 memory cell which has left = 1 & Right = 5. Draw the DQ structure & represent the position of left and right pointer for each of the following oper

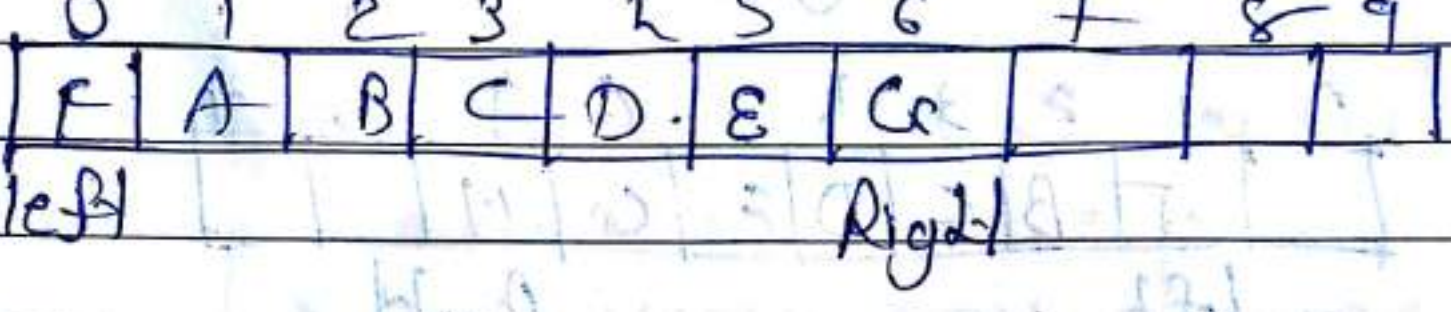
1. Insert F on the left
2. Insert G & h on the right
3. Delete two alphabets from left. nearest will delete
4. Insert I on the right
5. " J & on " left.
6. Delete two alphabet from the Right

left Right
1 6

Insert f on left to incremented by 1.

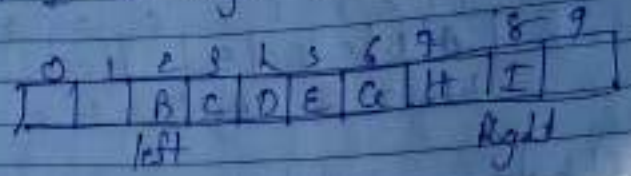


Insert G on right incremented by 1.

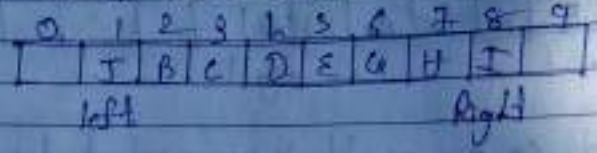


Insert H on right incremented by 1.

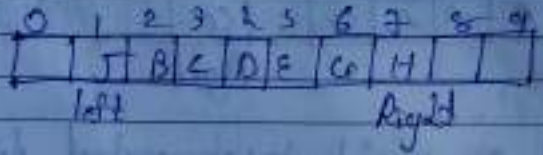
4. Insert I on right ++



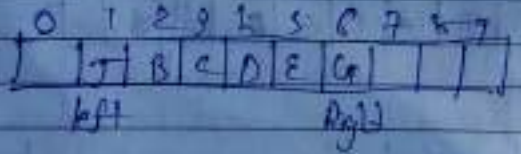
5. Insert I on left (left--)



6. Delete I from Right (Right--)



Delete H from Right (Right--)



⇒ If there is only one element into the DQ and if deletion operation is preform set left & right to zero (if queue is start from 0) else left-d

* Link list:

What is link list:

- A list is a collection of elements in which each element is linked with other and order is given by link from 1 element to another.
- Each element is denoted by node.
- First part contain the value of the node & second contain the address of next node.

* Advantages of array:

- Array is very simple to use & easy to create.
- No memory management needed.
- In array we can access any element with the help of index directly.
- Array is quick to look.

* Disadvantages of array:

- It is a static in nature. memory allocated at compile time.
- Insertion & deletion of element is more complica in comparison to linklist.

* Advantages of link list:

- It is a dynamic data structure.
- The size of the linklist can grow or shrink in size during execution of the program.

In linklist programmer need not a
how many element will be stored in the link
In a link list each element is allocated me
as it is added to the list.

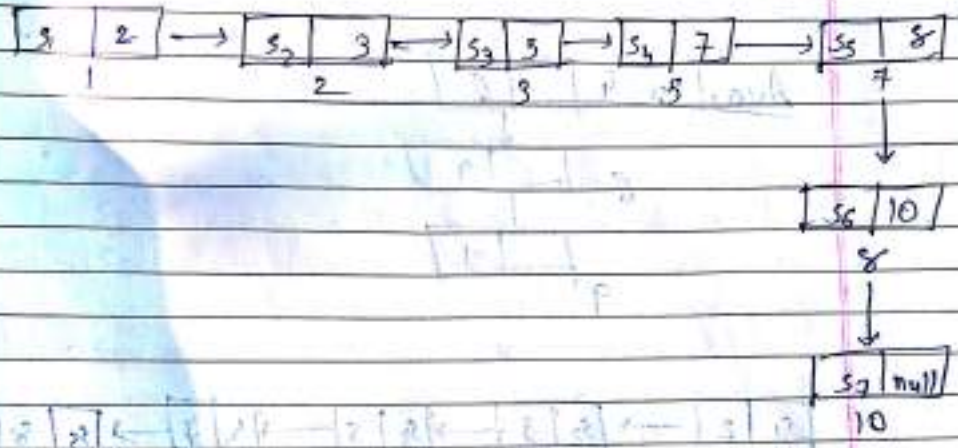
Disadvantages of L.L:

lots of overhead (lots of malloc call pair
assignment) (time consuming more
We have to traverse the entire list to go
end node
It consume extra space to contain the ad
of the next node.

Application of L.L:

To implement stack, queue, graph, tree are
implemented using L.L
Consider the history section of web browse
it creates the linklist of web pages visited
So when you press the back button the pre
node data is fetch

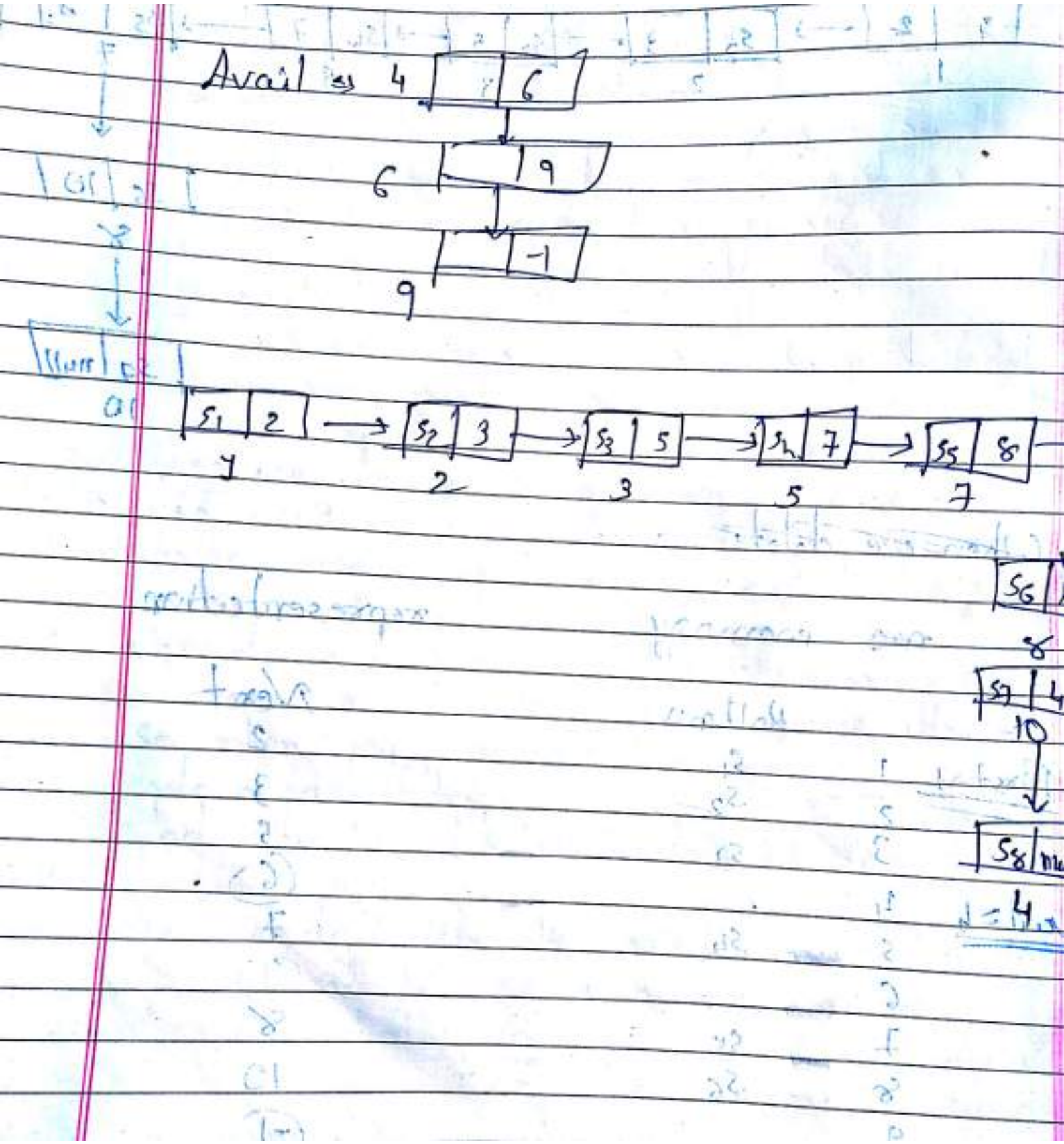
* Representation of linklist in memory



1. ~~When we delete~~

~~no~~ memory representation

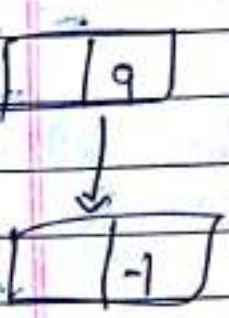
		Roll no	Next
First=1	1	S_1	2
	2	S_2	3
	3	S_3	5
Avail=4	4		6
	5	S_4	7
	6		9
	7	S_5	8
	8	S_6	10
	9		-1
	10	S_7	-1



memory Representation

1	S_1	2
2	S_2	3
3	S_3	5
4	S_8	(-1)
5	S_4	7
6		(9)
7	S_5	8
8	S_6	10
9		(0-1)
10	S_7	4

Avail = 6



When we delete a node from a linklist then operating system change status of the memory occupied by that node from memory available.
Compiler maintain a list of all the free memory cells.
The list of available space is called as free pool.

linklist has a pointer variable which store the address of the first node of the linklist.
likewise for the free pool (which is a linklist of all the free memory cell)

- Now suppose Now when new student [illegible] has be added when a memory add. is pointed by avail will be taken & use to store the information
- After the insertion next available free space address will be stored in avail

* If we delete a node space occupied by that node must be given back to the free pool so memory can be reuse by other program.
- The operating system does this task of adding free memory to the free pool.
- This task perform by O.S when it finds the CPU either or the programe are in a short of memory space
- The O.S scan through all the memory cell & mark those cell that are being use by some other program
- It collects all the cells which are not being use and add their address to free pool.
- This process is called as garbage collection.

Algorithm

a) Simply linklist

* W.A.A for insert a node at beginning.

Insert at beginning :-

FIRST = INSERT(X, FIRST);

X = new element which is to be inserted
first = a pointer to the first element of linklist
in which node contain info and link part.
avail = pointer to the top element of the
availability of stack.
new = temporary variable.

Procedure :-

[Underflow?]

if AVAIL = NULL
then write ('AVAILABILITY STACK UNDERFLO
Return (first))

[Obtain address of next free space]

New ← Avail

[Remove free node from availability stack]

Avail ← LINK(Avail)

4. [Initialize fields of new node & its link to the list]

INFO(NEW) ← x
LINK(NEW) ← first

5. [Return address of new node]

Return(NEW)

Avail 200 | | 250

250 | | 300

300 | | 350

350 | | 400

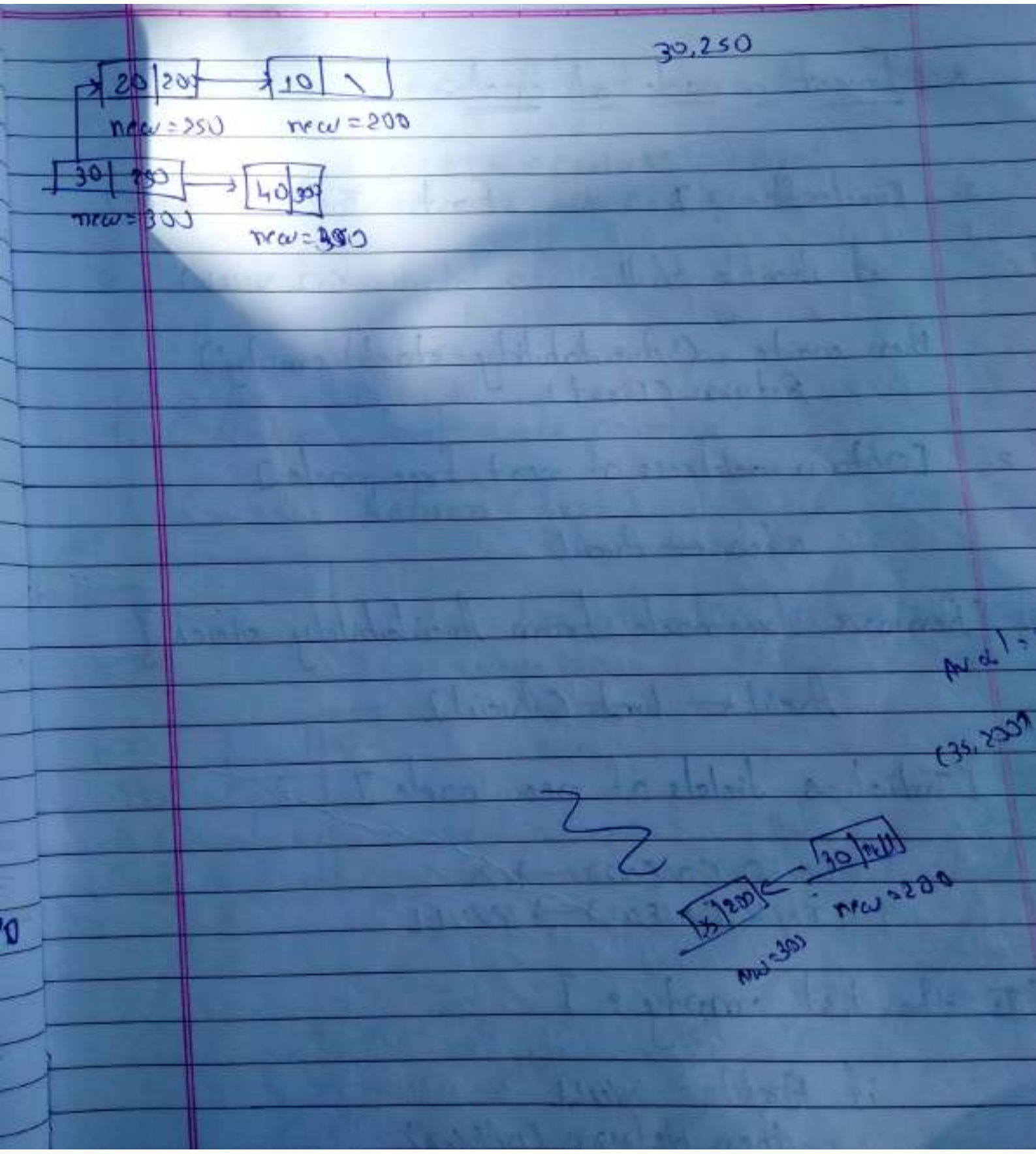
400 | |

temp = first
while(temp != null)
{
printf("%d", *temp);
temp ← LINK(temp);
}

50, 40, 30, 20, 10

Insert 10, 20, 30, 40, 50, 60

| 50 | 350 | → | 40 | 300 | → | 30 | 250 | → | 20 | 200 | → | 10 | null |



* Insert a node at end :

1. [Underflow ?]
   if Avail = Null
   then write ('Availability stack empty').
   Return (First)

2. [Obtain address of next free node]
   New ← Avail

3. [Remove free node from Availability stack]
   Avail ← Link (Avail)

4. [Initialize fields of new node]
   INFO (NEW) → X
   LINK (NEW) → NULL

5. [Is the list empty ?]
   if First = NULL
   then Return (NEW)

6. [Initiate search for last node]
   save ← First

First = Insert (X, First)
(30, NULL)

7. [Search for end of list]

Repeat while LINK(SAVE) ≠ NULL
SAVE ← LINK(SAVE)

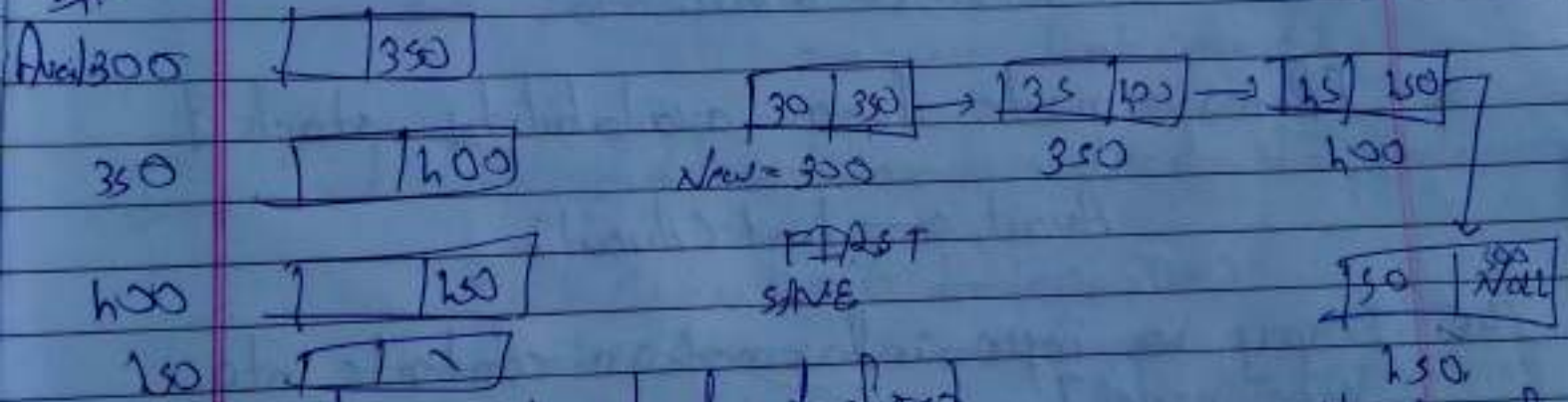
8. [set link field of last node to new]

link(save) ← new

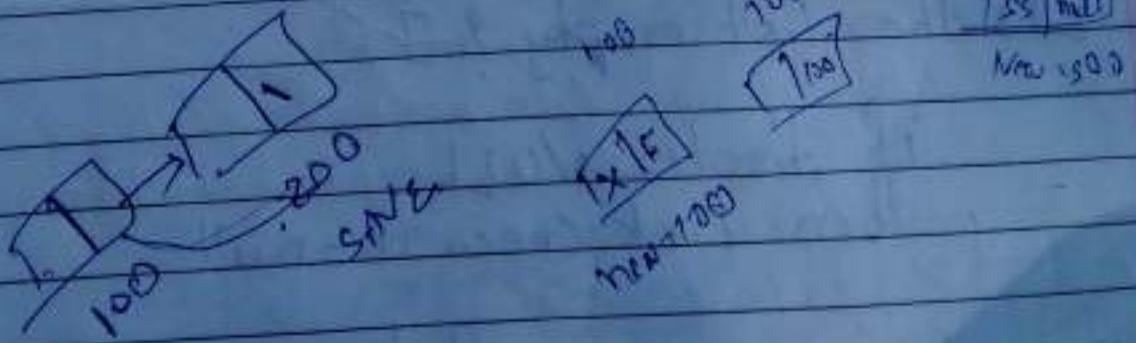
9. (Return first node pointer)

Return(first)

#



SAVE is always located at first.
First is fixed i.e. first node are inserted that is for



If Avail = Null
then write ('Availability stack empty')
Return (first)

2. [Obtain the address of next free node]
New ← Avail

3. [Remove node from availability stack]
Avail ← link(Avail)

if INFO(NEW) ≤ INFO(FIRST)
then LINK(NEW) ← first
Return (New)

[Initialize temporary variable]

SAVE ← first

[search for predecessor of new node]

Repeat while Link(save) ≠ null

and Info(Link(save)) ≤ Info(new)
save ← Link(save)

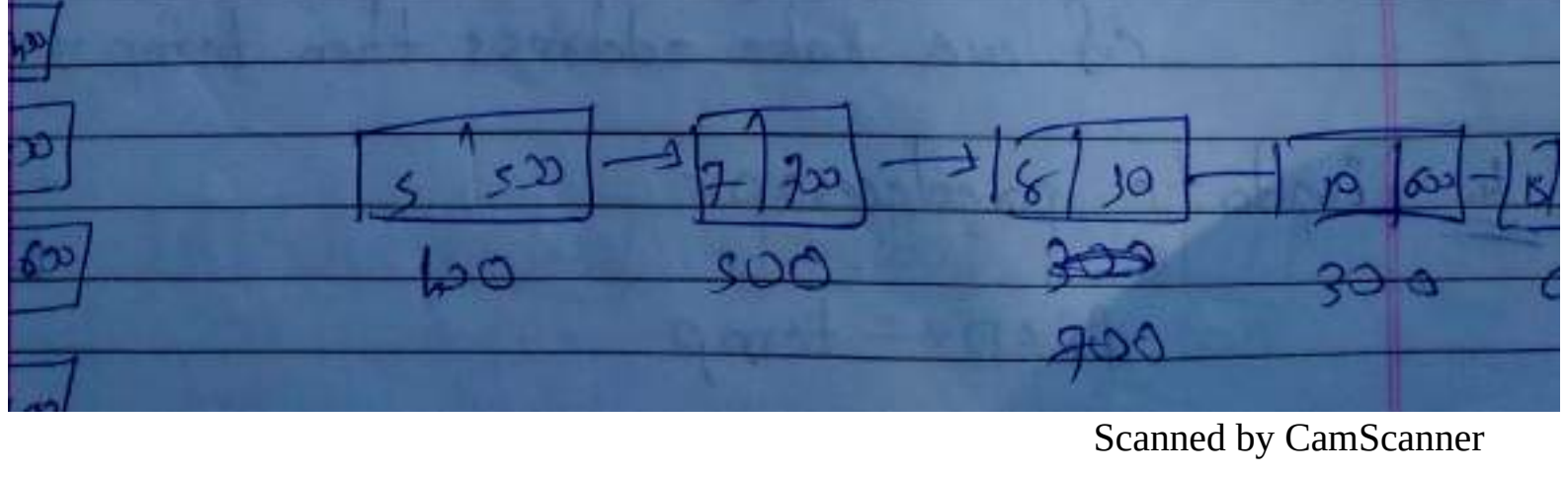
[set link fields of new node & its predecessor]

Link(New) ← link(save)
link(save) ← New

Return first node pointer)

Return (first)

20, 40, 50, 80, IN
20, 80, 50, 40, IN



* Delete an element from the Singly

Delete (x, first)

Pred which keeps the track of the prede
of temp

1. [Empty list ?]

if first = Null

then write ('Underflow')

Return

2. [Initialize search for x]

Temp ← first

3. [find x]

Repeat through step-5 while Info (te
and link (temp) ≠ Null
if we take address then temp ≠

4. [Update predecessor

PRED ← temp

[move to next node]

Temp ← link (temp)

[if we take x as address a
step T₁ ← temp)

[Delete x]

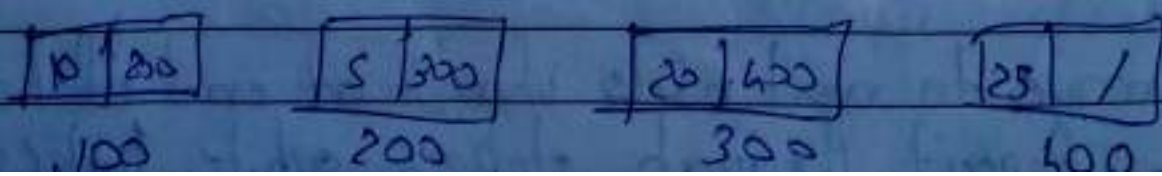
if x = Info (first) (if x as address
x = first)
then first ← link (first)

else
link (PRED) ← link (Temp)

[Return node to availability area]
link (x) ← Avail
Avail ← T₁ [if x is address Avail ←
Return

[End of list?]
if Info (temp) ≠ x [if x
as addr
then write (NODE (temp ≠ x)
NOT FOUND)
Return

(S, po)



[Empty list?]

if FIRST == NULL
then Return (NULL)

[Copy first node]

if Avail == Null
then write ('Availability stack empty')
Return 0;
else

Link = ptr
first = Begin
Info = field

5. [Update predecessor & save pointer]

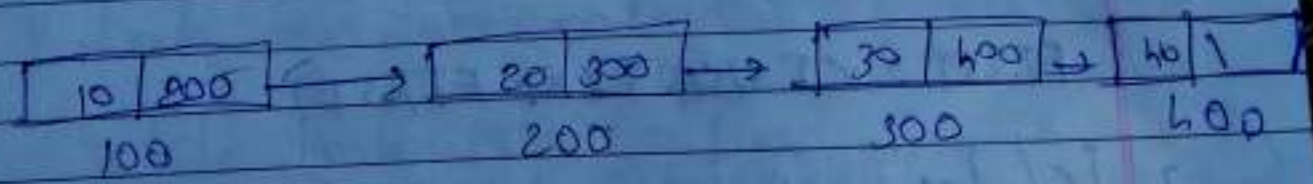
PRED ← new
save ← link(save)

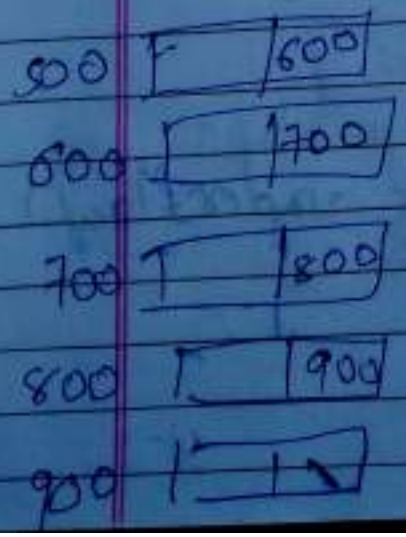
6. [Copy node]

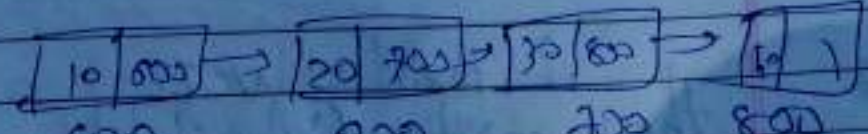
if Avail == null
then write ('Availability stack empty')
Return 0
else
[create node] new ← Avail
Avail ← link(Avail)
field(new) ← Info(save)
PTR(PRED) ← New

7. [Set link of last node and return]

PTR(NEW) ← NULL
Return (BEGIN)

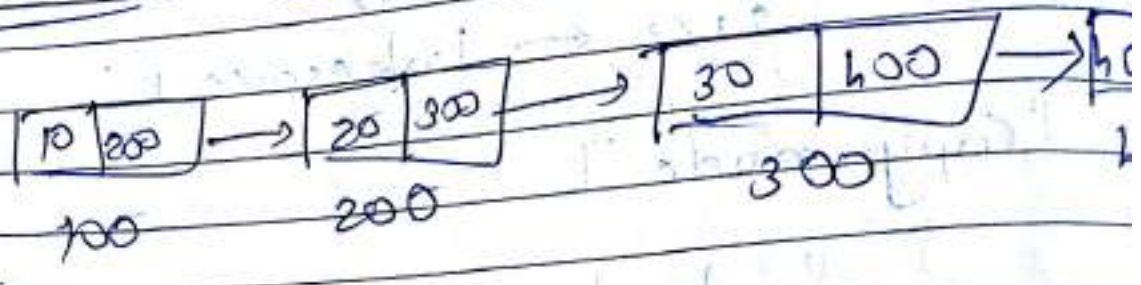






Begin

Circular link list:-



* Advantages:

⇒ In CLL linklist every node is accessible from given node. It means all node can be reach. chaining through the list.

* Disadvantages:

The only down side of CLL is the com of iteration.

* Insert a node at front of the circular (Head)

FIRST = CINSERT(X, FIRST)

1. [Underflow?]

if Avail = NULL
then write ('Availability stack underflo
Return (FIRST)

list always clamps.

2. NEW ← AVAIL

3. [Remove free node from availability stack]

Avail ← Link(Avail)

4. [Initialize field of new node]

Info(new) ← x

5. [check for the empty list]

if first = NULL
then
first ← new
LINK(new) ← first
return new

6. [set PTR]

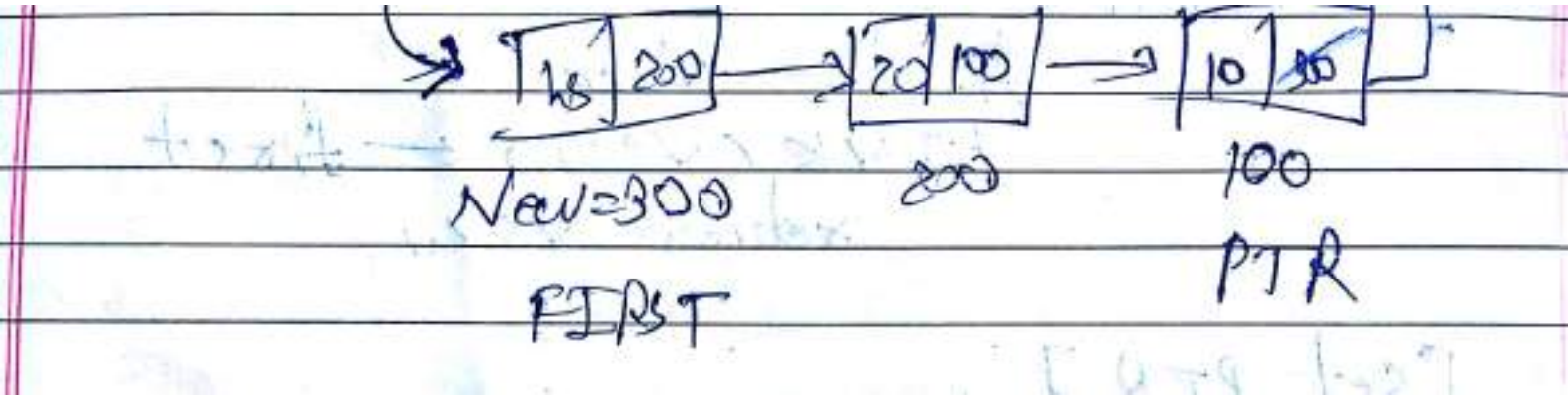
PTR ← FIRST

7. [check for last node]

while Link(PTR) != First
PTR ← Link(PTR)

8. [Add new node at front]

Link(new) ← first
link(PTR) ← new
first ← new
return first



[check for last node]

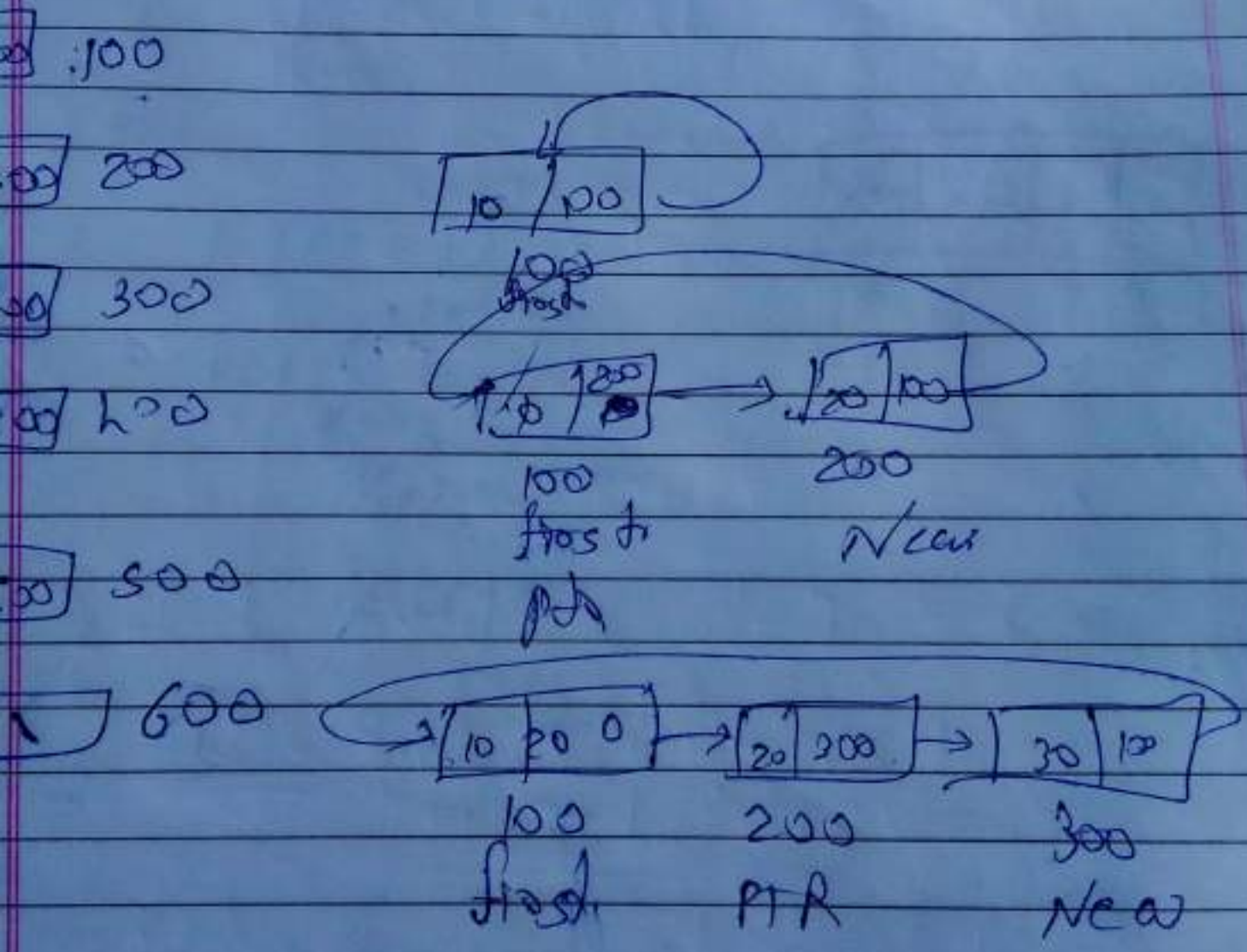
while Link (PTR) != first

PTR ← link (PTR)

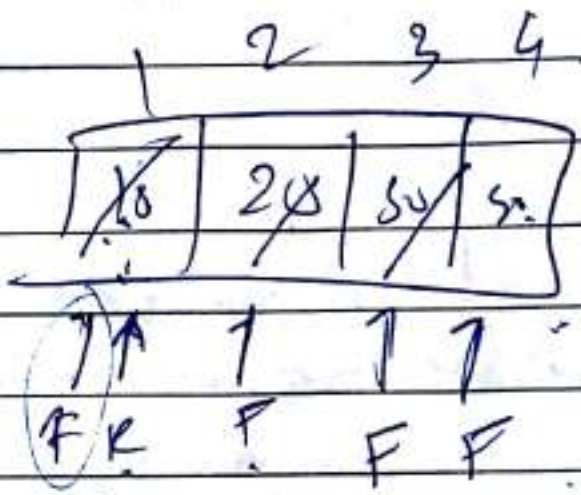
[Add new node at end]

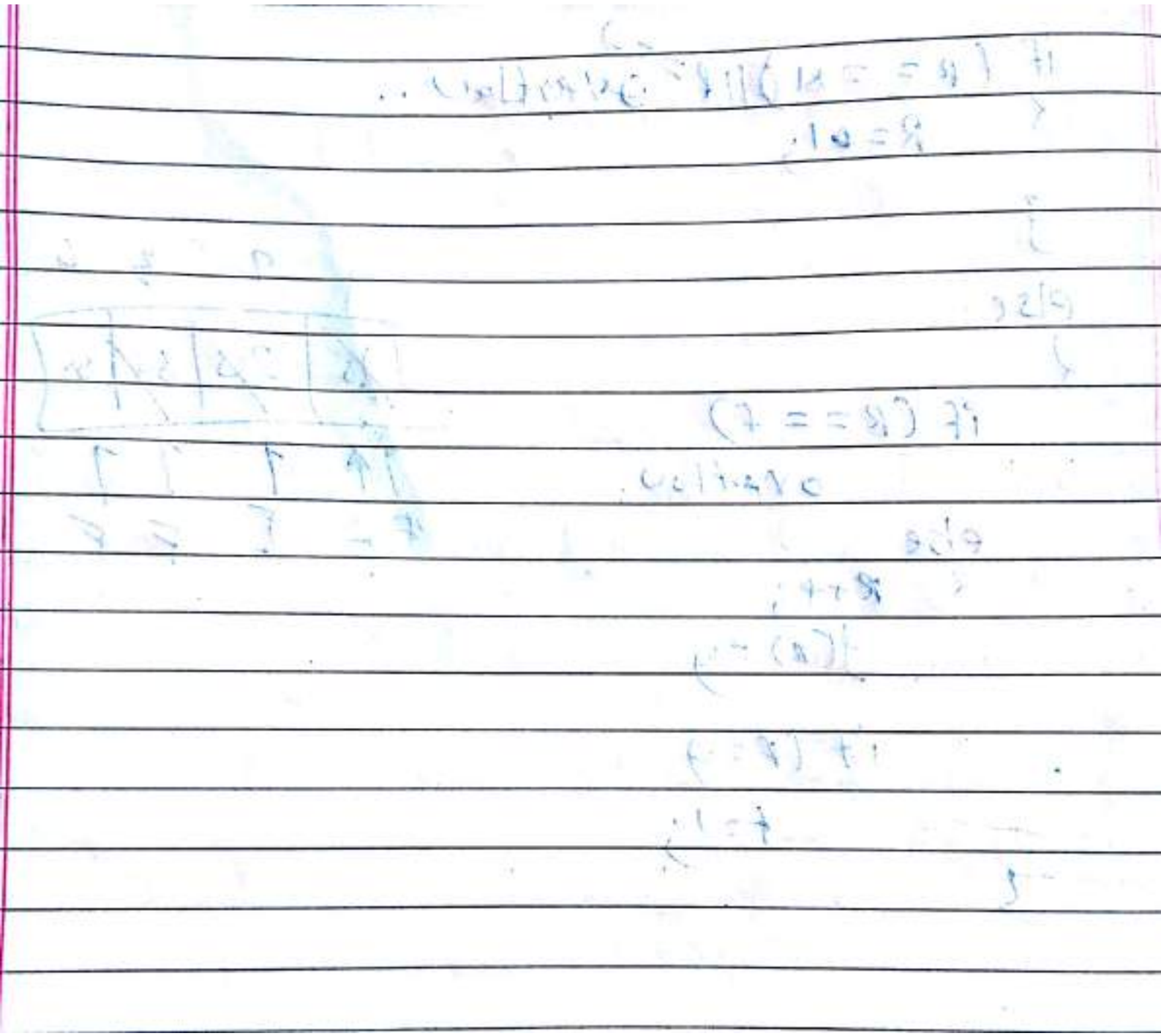
Link (PTR) ← NEW

Return (first)

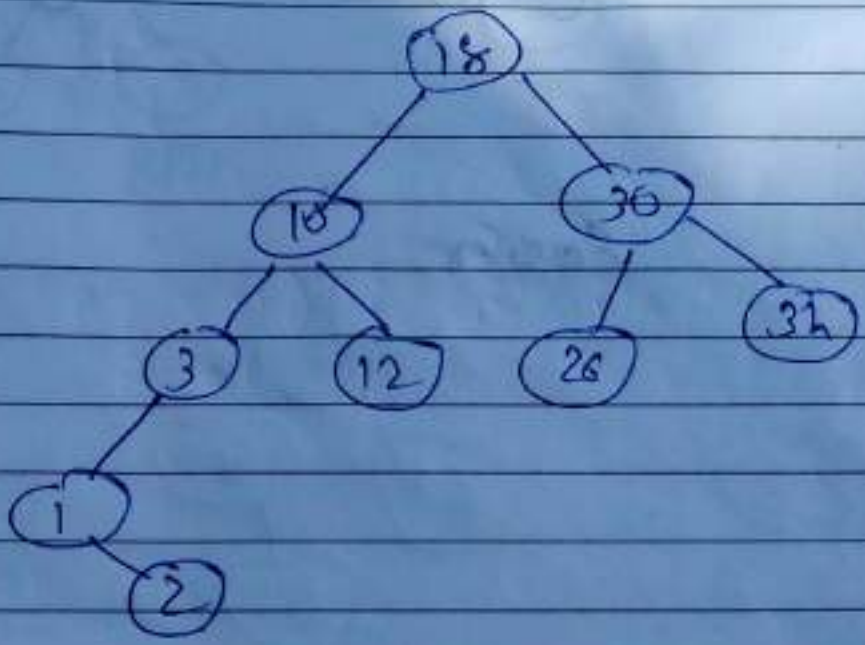


```
}
else
{
    if (R == f)
        overflow.
    else
    {  R++;
       q[R] = y
       if (F==-1)
           f=1;
    }
```





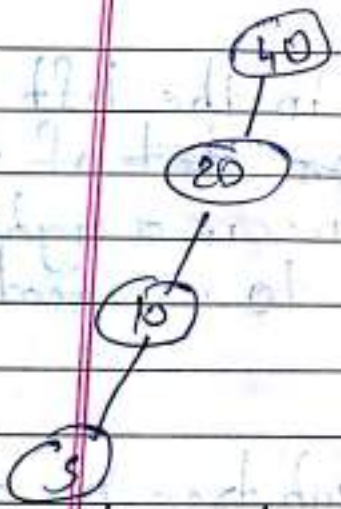
Binary search tree also known as ordered binary tree in which nodes are arranged in order.



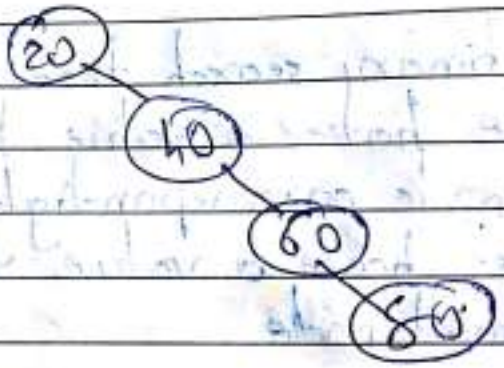
In a binary search tree all the node in the left sub tree have a value less than ~~of or that~~ of the root node correspondingly all the value in a right sub tree have a value either equal to or greater than root node.

Same rule is applicable to every sub tree in a ~~beg~~ given tree

	Array	linked list	Binary search tree
Searching	O(logn)	O(n)	O(logn)
Inserting and deleting	Expensive	Easier	Easier



left skewed BST

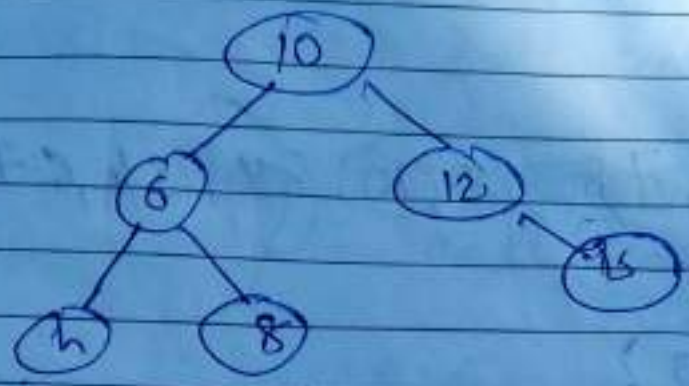


Right skewed BST

Binary search tree

worst case O(n)
best case O(logn)
Average case O(logn)

} Complexity

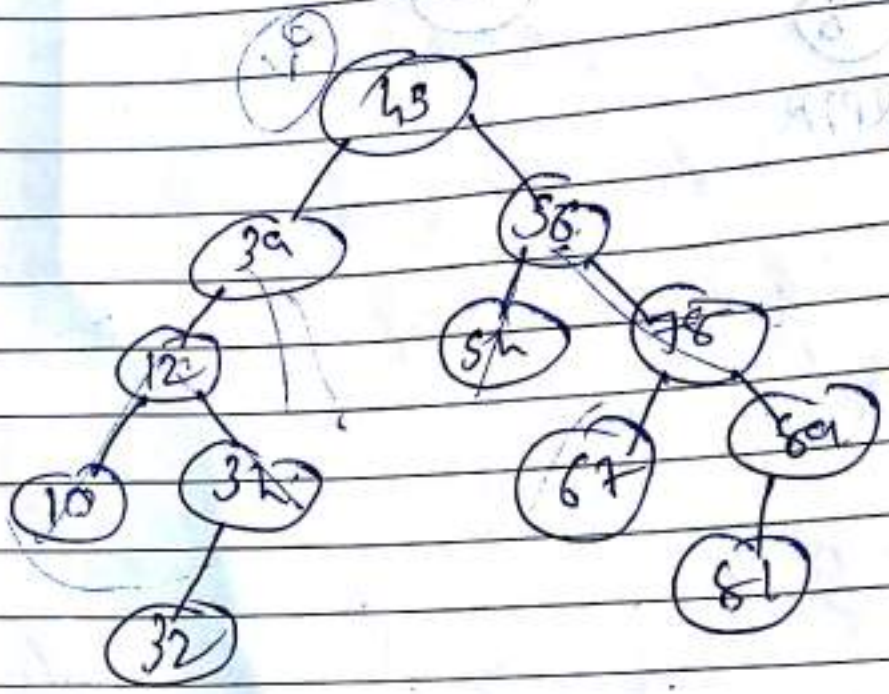


LPTR RPTR

Create BST:

(45) 39, 56, 12, 34, 78, 32, (10), 89, 54, 67, 8

root node



Operations on BST:

Searching: Algorithm (NODE → pointe
any n

search (NODE, VAL)

Step: 1 If NODE = NULL then

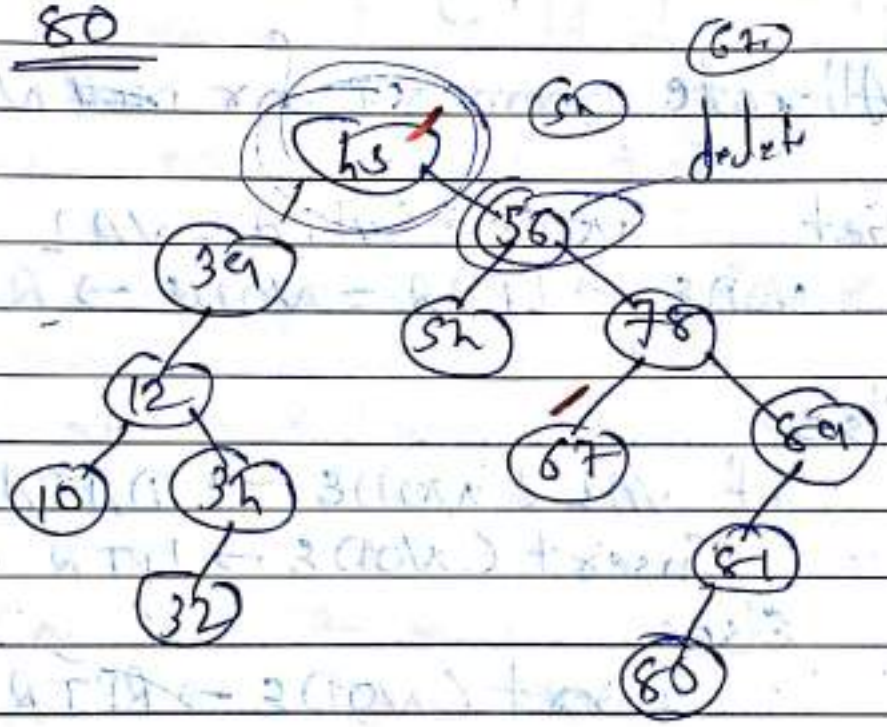
write ("Empty tree")

Else if NODE → DATA > VAL
call search CNODE → LPTR, VALD

Else if NODE → DATA < VAL
call search CNODE → RPTR, VALD

END

Insert 80



10, 12, 32, 39, 45, 52, 56, 67, 78, 89

Operation on BST

Inserting Algorithm

Insert (NODE, val)

step 1 If NODE = NULL then

~~write ("Empty loc")~~

Allocate memory for ~~null~~ NODE

Set NODE → DATA = VAL

NODE → LPTR = NODE → RPTR = NULL

Else

if VAL < NODE → DATA

Insert (NODE → LPTR, VAL)

Else

Insert (NODE → RPTR, VAL)

Search + insert

step 2 END

Deletion :

To delete the node from BST following cases are to be consider :

CASE 1: Deleting a node that has no child

In this case node can be deleted by setting it's parent pointer to ~~null~~ "NULL"

CASE 2: Deleting a node with one child

In this case replace the node with his child.

If the node was left child of its parent then child also become a left child.

If the node was right child of the parent then child also become the right child

CASE 3: Deleting a node with two child

The node is replace with its 'inorder' predecessor (right most child of left sub tree) or inorder successor (left most child of Right sub tree)

Delete (NODE, VAL)

Algorithm:

Step 1 If NODE = ~~VAL~~ NULL then
write "value is not found in TREE".

Else If VAL < NODE → DATA
Delete (NODE → LPTR, VAL)

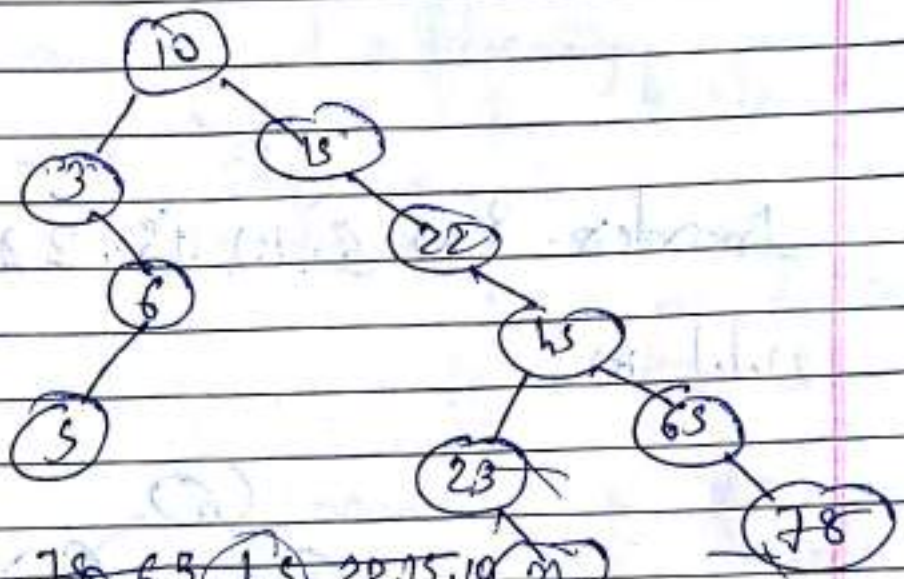
Else If VAL > NOD → DATA
Delete (NODE → RPTR, VAL)

If NODE → LPTR = NULL AND
NODE → RPTR = NULL
SET NODE = NULL
Else If NODE → LPTR != NULL
SET NODE = NODE → LPTR
Else
SET NODE = NODE → RPTR
[END OF IF]

End

Example:

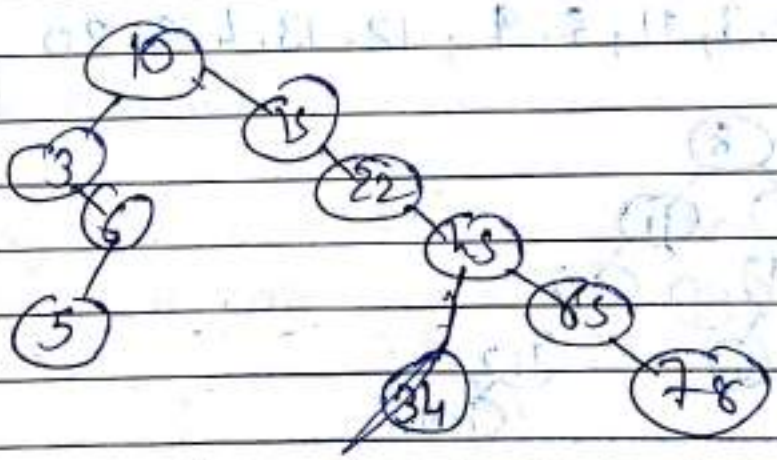
10, 3, 15, 22, 6, 45, 65, 23, 78, 34, 5



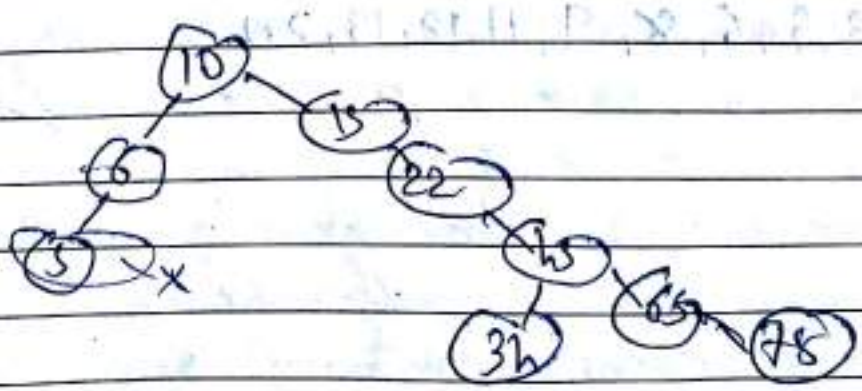
5, 6, 3, 34, 23, 78, 65, 45, 22, 15, 10

Inorder: 5, 6, 3, 10, 15, 22, 23, 34, 45, 65, 78

Delete :- 23



Delete 3

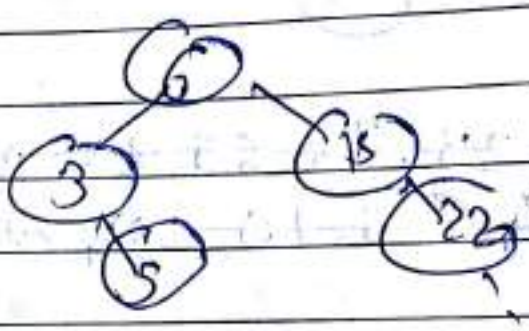


Delete 5

Delete 15

Inorder 3, 5, 6, 10, 15, 22, 23, 31, 45, 65, 7

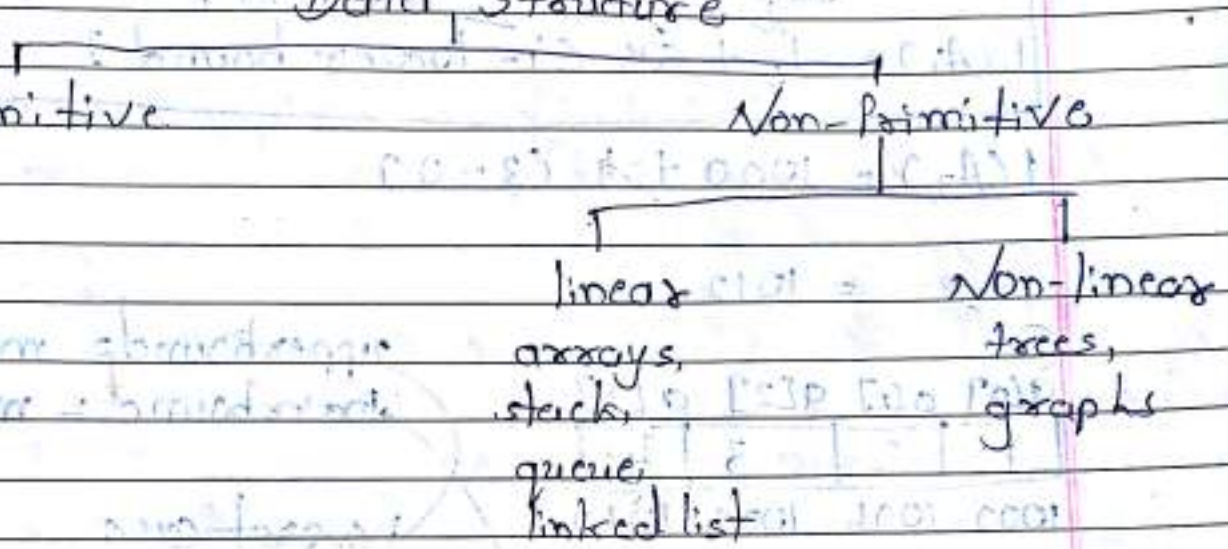
Del 10:



As it is

8, 3, 11, 5, 9, 12, 13, 4, 6, 20

* Data Structure: A data structure is basically a group of data elements that are put together under one name, and which defines a particular way of storing and organizing data in computer so it can be used efficiently



* Application of Array:

- Arrays are frequently used in C, as they have a number of useful applications. These applications are:

- Arrays are widely used to implement mathematical vectors, matrices and other kinds of rectangular tabels.
- Many database include ~~one~~ 1-D arrays whose elements are records.
- Arrays are also used to implement other data

address base → l_0, element size → c, element position → i, lower bound → 0

$$L(A_i) = l_0 + c \times (i - \text{lower bound})$$

$$L(A_3) = 1000 + 4\,(3-0)$$

$$= 1012$$

a[0]	a[1]	a[2]	a[3]
1	2	3	4
1000	1004	1008	1012

upperbound = max^m

lowerbound = min^m

↳ worst case

max

* Array represent of Sparse matrix. (most of the element are represent zero)

$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 4 & 0 & 2 & 0 \\ 0 & 0 & 0 & 3 \end{bmatrix}$$

1-D

Row-mapping {1, 4, 2, 3}

column- " {4, 2, 1, 3}

2-D

row-column mapping

Ex: $\begin{bmatrix} 2 & 0 & 0 & 0 \\ 4 & 0 & 0 & 7 \\ 6 & 0 & 5 & 3 \end{bmatrix}$

Row mapping

= {2, 4, 7, 6, 5, 3}

column mapping

= {2, 4, 6, 5, 7, 3}

2D represent

	0	1	2	3
0	2	0	0	0
1	0	0	0	7
2	6	0	0	3

row	0	1	2	2
column	0	3	0	3
value	2	7	6	3

Ex:

	0	1	2	3
0	1	0	0	0
1	0	3	0	0
2	0	0	4	0

row	0	1	2
column	0	1	2
value	1	3	4

* Types of sparse matrix:

* Lower triangular sparse matrix:

- All elements above the main diagonal ha zero values

$i < j \quad A_{ij} = 0$

$$\begin{bmatrix} 1 & 0 & 0 \\ 2 & 3 & 0 \\ 4 & 5 & 6 \end{bmatrix}$$

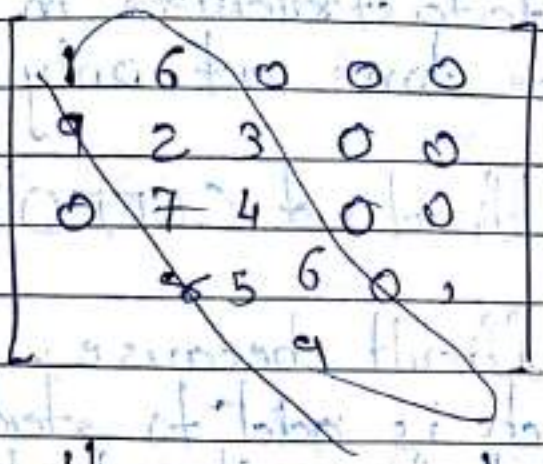
* Upper triangular sparse matrix:

- All elements below the main diagonal zero values

$i > j \quad A_{ij} = 0$

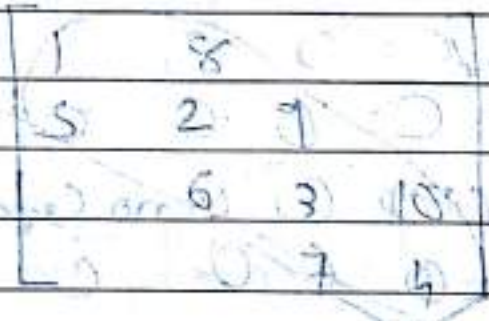
	1	2	3
1	1	2	3
2	0	4	5

Tri-diagonal matrix :-



Another variant of sparse matrix

In with elements with a non zero value can appear only on the diagonal & immediately above or below the diagonal



N element element

If top >= N st
overflo

top >= N

push (13) 3 no. stack

① top = 0
0 >= 3 x

② top = 1

③ S[1] = 13

11
9
13

push (9)
top = 1
(1) 1 >= 3 x
(2) top = 2
(3) S[2] = 9

push (11)
top = 2
(1) 2 >= 3 x
(2) top = 3
(3) S[3] = 11

POP (s, top)

vector of stack

Pointer which point to the top element of stack

[check Underflow]

if top = 0
then "stack Underflow"
return

[Decrement top by 1]

top ⟵ top - 1

[Delete element from stack]

~~x ⟵ s[top+1]~~

x ⟵ s[top+1]

top = 0

x

top - 1

3 - 1

2

s[top+1]

s[3]

0 'x' = 10

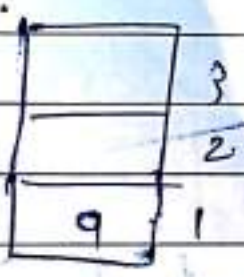
Delete

top = 3

vector top element

PUSH (A, 0, 9)

PUSH (A, 0, 9)



top → 0

top = 1

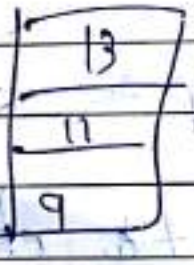
x = 3

PUSH (A, 1, 13)

top = 1

top = 1+1 = 2

PUSH (A, 2, 13)



top = 2

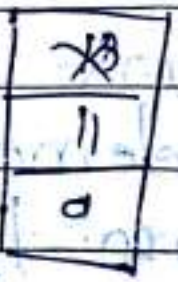
top = 2+1 = 3

PUSH (A, 3, 15)

3 >= 3

stack full

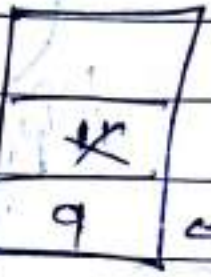
POP (A, 3)



top = 3 ✗
top = 3-1 = 2

x = s[2+1] = s[3] = 13

POP(A, 2)



← top.

top = 2 ✗
top = 2-1 = 1

x = s[1+1] = s[2

POP(A, 1)

top = 1 ✗
top = 1-1 = 0

Return the element

Return (s[top-i+1])

Precedence:

()	[]
^	^
*	/ %
+	-
=	

$$\underbrace{\left[\underbrace{(a+b)}_{1}+\underbrace{(c+d)}_{2}\right]}_{3}$$

a + b + c + d * E

Prefix notation: The prefix notation in the operator is written before the opera It is also cally polishnotation.

operator operand operand
+ a b

Postfix notation :- In the postfix notatio operator are written after the operands. So it is called a postfix notation. It is also known as suffix notation or a polish

operand operand operator
a b +

Infix	prefix	postfix
a+b	+ab	ab+
a+b*c	+a*bc	abc*+
a+(b*c)	a+(*bc)	abc*+
+a*bc	a+(bc*)	
	abc*+	
a*b+c	+*abc	~~ab*+c~~
		ab*c+
a+(b+c)	a+bc+	a+bc+
	~~abc++~~	abc++
	a+(+bc)	
	+a+bc	
a*(b+c)	a*+bc	a*bc+
	*a+bc	~~abc+*~~
a*b*c	~~a**bc~~	~~a*bc*~~
	~~**abc~~	~~abc*~~
	**abc	ab*c

Example:

(a-b) * (c+d)

(ab-) * (cd+)

T = ab-

w = cd+

T * w

Tw *

ab-cd+*

* (ab-) * (c+d)

T = ab-

T * (c+d)

T * (cd+)

q = cd+

T * q

Tq *

ab-cd+*

2. (A+B)/(C+D)-(D*E)

(AB)+/(C+D)-(D*E)

P=AB+

P/(C+D)-(D*E)

P/(CD+)-(D*E)

P/Q-(D*E)

Q=CD+

P/Q-(DE*)

W=DE*

P/Q-W

PQ/-W

PQ/W-

AB+CD+/DE*-

3. A+[(B+C)+(D+E)*F]/G

A+[(BC+)+(D+E)*F]/G

P=BC+

A+[P+(DE+)*F]/G

A+[P+Q*F]/G

Q=DE+

A+[P+QF*]/G

W=QF*

A+[P+W]/G

T=PW+

A+T/G

A+TG/

S=TG/

A+S

~~A~~ As +

A + Ca / +
Ap ω + Ca / +
Ap q F * + Ca / +
Ap D E + F * + Ca / +

A B C + D E + F * + Ca / +

* Evaluation of postfix ~~operation~~ not

Rule 1: find the left most operator in the expressi

Rule 2: Select two operand immediatley to the left
the operator found.

Rule 3: perform the indicated operation

Rule 4: Replace the operator & operand with the re

1. 2 3 4 * +

2 12 +

14

Example:

5.

42/3 - 56* + 42* -

23 - 56* + 42* -

-156* + 42* -

-56* + 42* -

-130 + 42* -

29 & - 21

23 - 56* + 42* -

-130 + 42* -

-1 - 56* + 42* -

-130 + 42* -

Q Infix to prefix conversion:

1. A * B + C

2. A / B ^ C + D

3. (A - B / C) * (D * E - F)

1. A * B + C

(* A B) + C

+ * A B C

2. A / B ^ C + D

A / (^ B C) + D

T = ^ B C

A / T + D

/ A T + D

q = / A T

q + D

+ q D

+ / A T D

+ / A ^ B C D

3. (A - B / C) * (D * E - F)

(A - (/ B C)) * (D * E - F)

T = / B C

(A - T) * (D * E - F)

- A T * (D * E - F)

q = - A T

q * (* D E - F)

s = * D E

q * (s - F)

q * (- S F) w = - S F

q * w

* q w

* - A T - S F

* - A / B C - * D E F

operand value = +1
operator value = -1

Rank = 1 valid
Rank = 0 Invalid.

3. [(A*B) + (c/d)] = F without stack

[*AB + /cd] = F
 | |
 T   q

[T + q]
+Tq = F
-+Tq F
-+*AB/cd F

* find Rank

Rank = 1 valid
Rank = 0 not valid

operand { R(+) = R(/) = R(-) = R(+) = -1
          R(a) = 1

⇒ A + B * c -
1 - 1 = (1) - (1) = 1 - 1 = 0 Not valid.

AB*C+ = 1 = valid

variable precedence high
* ( )

Infix to postfix using stack:

1. A * B ÷ C

f(next) ≤ f(top)

char scan	stack	postfix
	#	
A	#A	A
*	#*	A
B	#*B	AB*
+	#+	AB*
C	#+C	AB*C+

2. a + b * c => abc*+

char scan	stack	postfix
*	#	
a	#a	a
+	#+	a
b	#+b	ab
*	#+*	ab
c	#+*c	abc*+
#		

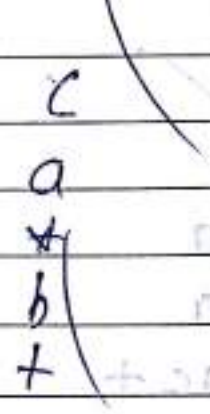
1. a*b*c

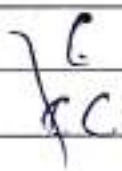
Symbol Scanned	stack	postfix
	#	
a	#a	a
*	#*	

3. a+b*c*d

Symbol Scanned	Stack	postfix
	#	
a	#a	a
+	#+	a
b	#+b	ab
*	#+a*	ab
c	#+*c	abc
*	#+*	abc*
d	#+*d	abc*d
#		abc*d*+

4. (a*b)+c





5. (a+c)+(c*d)/e ⇒ (a c +) + (c d *) / e
x + y / ε x + y ε /

s.c | stack | postfix

c | c | c
c | c c | c c
a | c c a |

c
o
+
e
)
+
C
c
*
d
)
/
e

c
cc
cca
cc+
cc+c
c
c+
c
*
*d

o
o
ac+
ac+
ac++
ac++
ac++c
ac++c

(
a
+
c
)
+
C
c
*
d
)
/
e

C
CC
CCa
CC+
CC+c
C
C+
C+C
C+Cc
C+C*
C+C*d
C+
C+/
C+/e

a
a
ac+
ac+
ac+
ac+
ac+c
ac+c
ac+cd*
ac+cd*
ac+cd*
~~ac+cd*/e~~
ac+cd*e/+

Symbol	precedance	Rank
	f	x
+ -	1	-1
*, +, /	2	-1
↑ (^)	3	-1
a, b, c, ...	4	1
#	0	-

* Unparenthesized suffix :

Algorithm

1. TOP ← 1
   S[TOP] ← #

2. POLISH ← ''
   RANK ← 0

3. NEXT ← NEXTCHAR (INFIX)

4. Repeat through step 6. while next ≠ '#'

5. if (f(next) ≤ f(top))

   Temp ← POP(S, TOP);

   POLISH ← POLISH concat TEMP

   Rank ← Rank + x(temp)

call push (s, top, next)
next = nextchar (Infix)

7. Repeat while (s[top] != #)
temp = pop(s[top])
POLISH = POLISH o TEMP
RANK = RANK + r(temp)

8. if rank == 1
then right value.
else
right invalue.

1. a*b+c

char scan	stack	(Polish) postfix	Rank
.	#		0
a	#a		
*	#*	a	1
b	#*b	a	1
+	#+	ab*	1
c	#+c	ab*	1
#		ab*c+	1

prefix - polish
postfix - reverse polish

1. a+b*c-d/e*h.

char scan	stack	postfix	Rank
	#		
a	#a		
+	#+	a	1
b	#+b	a	1
*	#+*	ab	2
c	#+*c	ab	2
-	#-	abc*+	1
d	#-d	abc*+	1
/	#-/	abc*+d	2
e	#-/e	abc*+d	2
*	#-/*	abc*+de/	2
h	#-*h.	abc*+de/	2
		abc*+de/h*-	1

2.

* Parenthize suffix:

step-1 top = 1
S[top] = 'C'

step-2 polish ← ' '
Rank ← 0

step-3 NEXT = NEXTCHAR (INFIX)

step-4 Repeat through step-7 while next != ''

step-5 if top < 1 then write Invalid
Exit

step-6 Repeat while f(next) < g(s[top])

temp = POP(S, TOP)
Polish = Polish o temp
Rank = Rank + r(temp)

step-6 if f(next) != g(s[top])
then call push(s, top, next)

else
pop(s, top)

step-7 Next ← Nextchar (Infix)

step-8 If Rank != 1 || top != 0

then Invalid

else valid

Example:

I. a-(b/c+(d/e*f))/g)*h

char scan	stack	postfix
	(	
a	(a	
-	(-	a
(	(-(	a
b	(-(b	a
/	(-(/	ab
c	(-(/c	ab
+	(-(+	ab/
(	(-(+(	ab/c
d	(-(+(d	ab/c
/	(-(+(/	ab/cd
e	(-(+(/e	ab/cd
*	(-(+(*e	ab/cde
f	(-(+(*f	ab/cde
)	(-(+	ab/cde/f*
/	(-(+/	ab/cde/f*+
g	(-(+/g	
)	(-(+	
*	(-*	
h	(-*h	

a	C	
-	C a	
C	C -	
b	C - C	a
/	C - C b	a
c	C - C /	a
+	C - C / c	a b
C	C - C +	a b
d	C - C + C	a b c /
/	C - C + C d	a b c /
e	C - C + C /	a b c / d
*	C - C + C / e	a b c / d
f	C - C + C *	a b c / d e /
)	C - C + C * f	a b c / d e /
/	C - C +	a b c / d e / f *
g	C - C + /	a b c / d e / f *
)	C - C + / g	a b c / d e / f *
*	C -	a b c / d e / f * g / +
h	C - *	a b c / d e / f * g / +
@	C - * h	a b c / d e / f * g / +
)		a b c / d e / f * g / + h * -

Scanned by CamScanner

E- (A/B+C)-D

char scan	stack	postfix
	C	
C	CC	
A	CCA	a
/	CC/	a
B	CC/B	ab
+	CC+	ab/
C	CC+C	ab/c
)	C	ab/c+
-	C-	ab/c+
	C-D	ab/c+d-

(with precedence)

Symbol	Input precedence function (f)	stack precedence function (g)	Rank
+, -	1	2	-1
*, /	3	4	-1
↑	6	5	-1
variable	7	8	1
(	9	0	-
)	0	-	-

Symbol	stack	Postfix
	(	
(	((	
A	((A	
+	((+	
B	((+ B	A
)	(	A B +
*	(*	A B +
D	(* D	A B + D
/	(* /	A B + D *
ε	(* / ε	A B + D * ε /
)		A B + D * ε / (

2. (A+B/C)*(D/e-f)

symbol	stack	Postfix	Rank
	C		
C	CC		
A	CCA	A	1
+	CC+	A	1
B	CC+B	AB	2
/	CC+/	AB	2
C	CC+/C	ABC/+	1
)	C	ABC/+	1
*	C*	ABC/+	1
C	C*C	ABC/+	1
D	C*C(	ABC/+	1
/	C*C(/	ABC/+D	2
e	C*C(/e	ABC/+D	2
-	C*C(/-	ABC/+De/	2
f	C*C(/-f	ABC/+De/	2
)	C*	ABC/+De/f-	2
)		ABC/+De/f-*	1

VALENTINE
Page No.:
Date: / /

A ↑ B ↑ C

*

3

Symbol	stack	Postfix	Rank
A	#		
↑	#A	A	1
B	#↑	A	1
↑	#↑B	AB	1
C	#↑↑	AB	1
#	#↑↑C	ABC↑↑	1

6

	C		
A	CA		
↑	C↑	A	1
B	C↑B	A	1
↑	C↑↑	AB	2
C	C↑↑C	AB	2
)		ABC↑↑	1

* (a-b) / (c*D ↑ E)

	C		
(	C(		
a	C(a		1
-	C(-	a	1
b	C(-b	a	1
)	C	ab-	1
/	C/	ab-	1
(	C/(	ab-	1
c	C/(c	ab-	1
*	C/(c*	ab-c	2
D	C/(c*D	ab-c	2
↑	C/(c*D↑	ab-cD	3
E	C/(c*D↑E	ab-cD	3
)	C/	ab-cDE↑*	2
		ab-cDE↑*/	1

(a + b ↑ c ↑ d)

(	(	a	1
a	((	a	1
+	((a	ab	2
b	((+	ab	2
↑	((+b	abc	3
c	((+↑	abc	3
↑	((+↑c	abcd↑↑+	1
d	((+↑↑	abcd↑↑+	1
)	((+↑↑d	abcd↑↑+	1
*	(	abcd↑↑+e	2
(	(*	abcd↑↑+e	2
e	(*(	abcd↑↑+ef	3
+	(*(e	abcd↑↑+ef	3
f	(*(+	abcd↑↑+ef/	2
/	(*(+f	abcd↑↑+ef/+	1
d	(*(+/	abcd↑↑+ef/+*	1
)	(*(+/d		
)	(*		
	✗		

* Evaluate postfix Expression:

1. AB + A=2, B=3

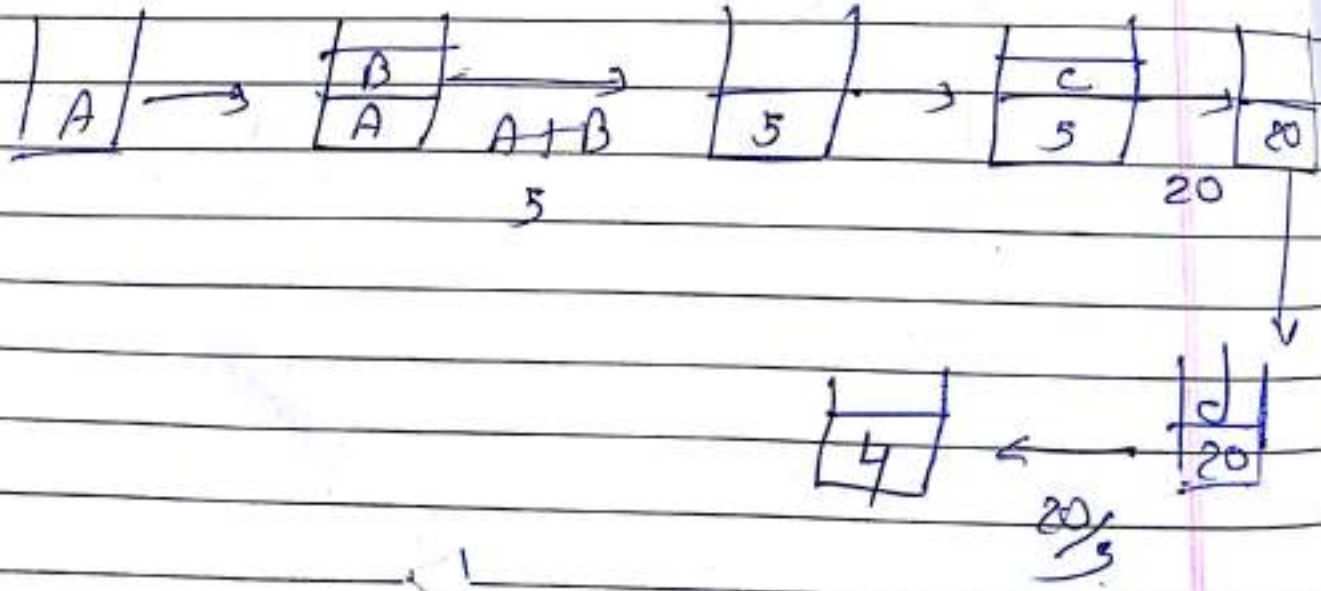
(i) A is operand so push into stack

(ii) B is operand, so push into stack

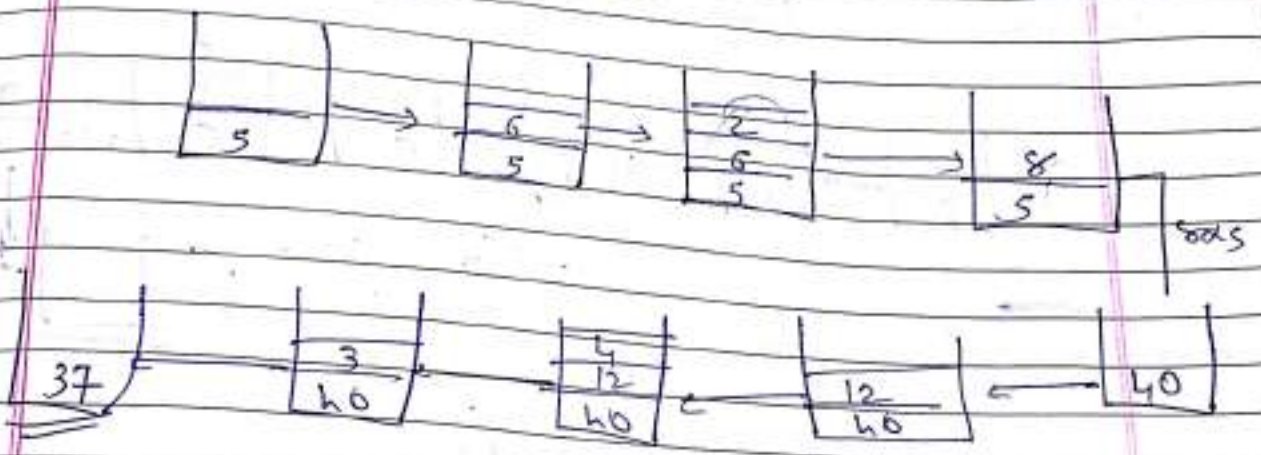
(iii) + operator, so pop top 2 element from stack

A + B
2 + 3 = 5

AB + C * D /
where A = 2, B = 3, C = 4, D = 5



Q2 562+*124/- 37

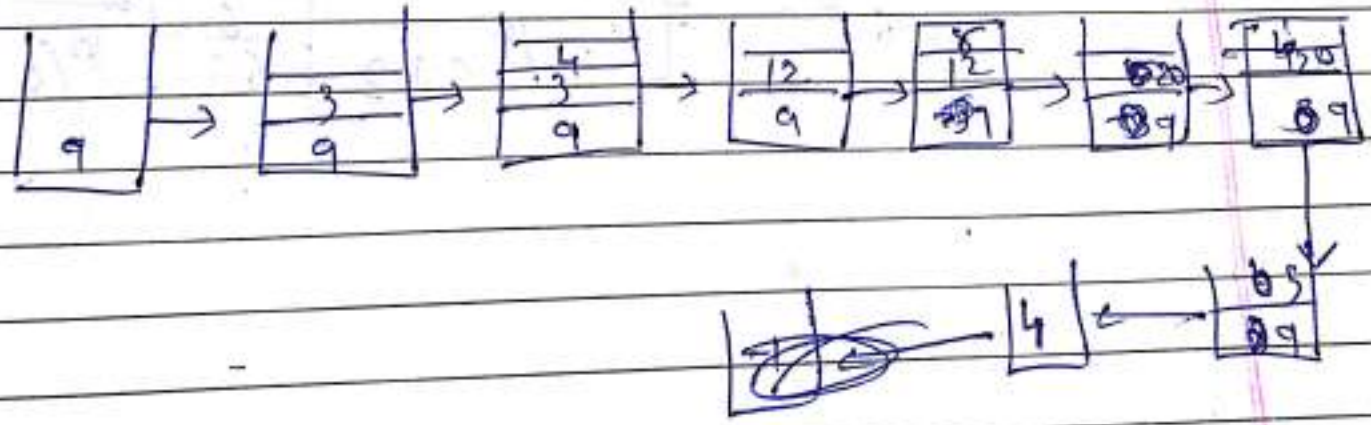


Q3 9-((3*4)+8))/4

(4) (*43+8)
9-(+*3 8))/4
9(/+*3 8 4

~~3*+8+~~

9-(34*8+)/4
934*8+4/-



(Q) (a+b*c*d) * (e+f/d)

~~(abcd**+) * (efd/+)~~

~~abcd**~~ abcd*+efd/+*

a=2, b=3, c=4, d=5, e=6, f=8

Prefix को Reverse करके Starting से.

Infix to prefix conversion :- (a+b*c)

step-1 Reverse the input string Expression (c*b+a)

step-2 Apply algorithm of Infix to postfix [cb*a+]

step-3 Reverse the postfix expression to get prefix expression.

[+a*bc]

Note:

while converting Infix to prefix do not POP when precedence is same in stack and input string for left associative operator

Infix to Prefix conversation

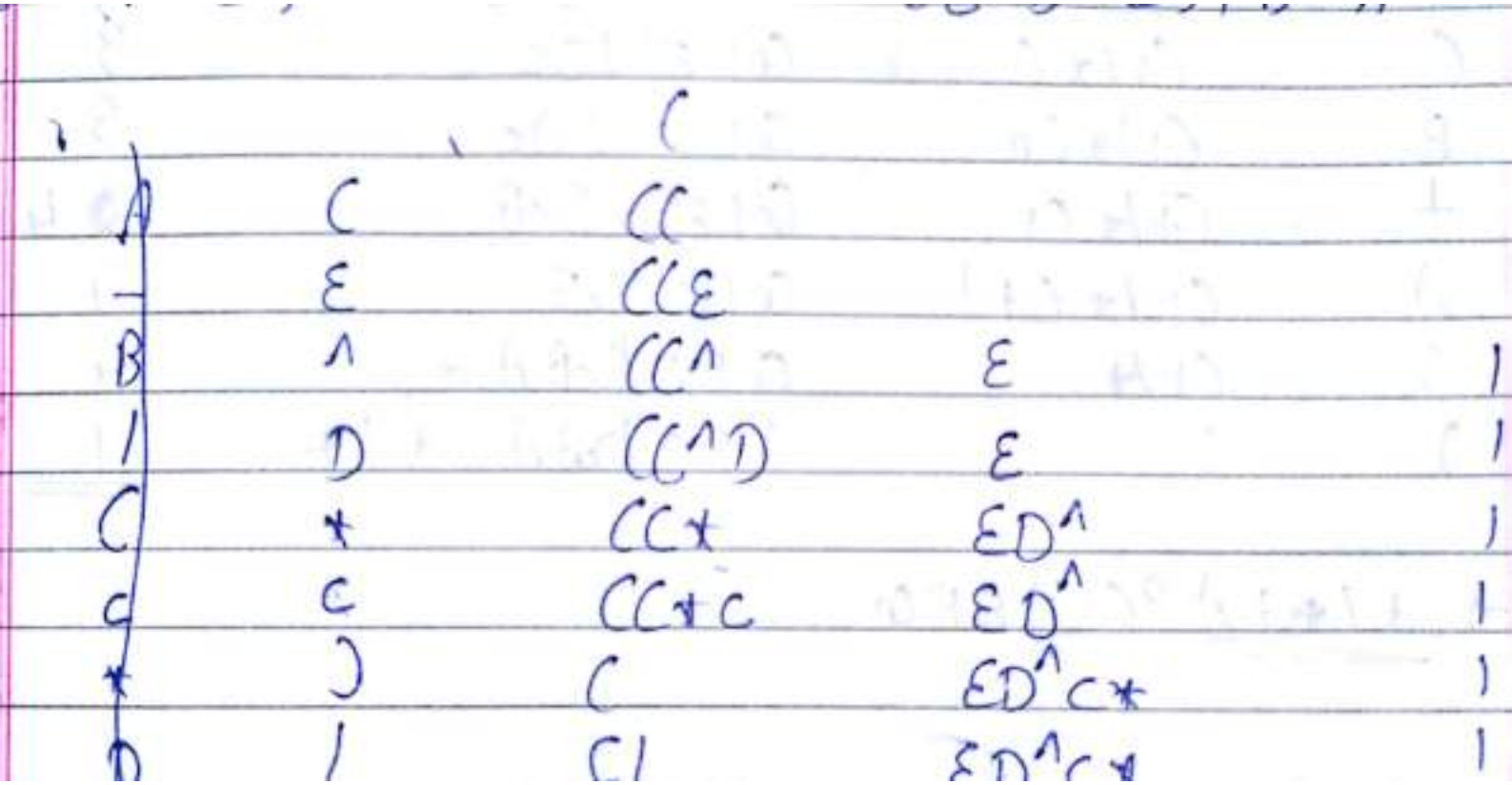
(A+B*C) ⟶ (C*B+A)

Symbol scanned	stack	Postfix	Rank
	(		
(	((		
c	((c	c	1
*	((*	c	1
B	((*B	cB*	1
+	((+	cB*	1
A	((+A	cB*A	1
)	(	cB*A+	1
)		cB*A+	1

Ans: +A*BC

2. CA*B/C

Symbol	stack	Postfix	Rank
	(		
C	((		
A	((A	A	1
*	((*	A	1
B	((*B	AB	2
/	((/	AB	2
c	((/c	ABc	1
)	(	ABc/*	1
)		ABc/*	1



/	C/	G	1
F	C/F	G	1
^	C/^	GF	2
E	C/^E	GF	2
+	C+	GFE^/	1
D	C+D	GFE^/	1
/	C+D/	GFE^/D	2
c	C+/c	GFE^/D	2
*	C+/*	GFE^/Dc	3
C	C+/*C	GFE^/DC	3
B	C+/*CB	GFE^/DC	3

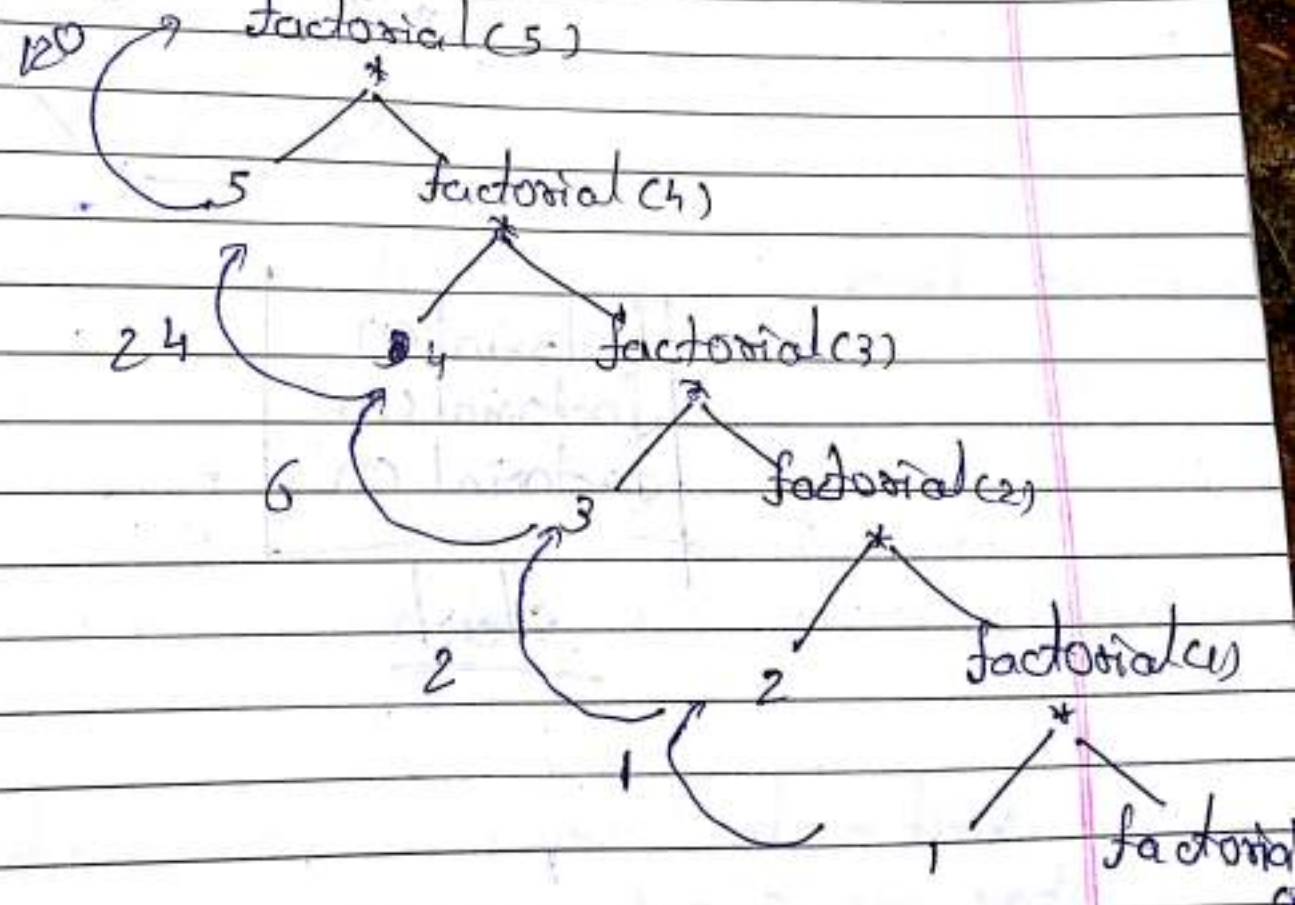
Application of stack,

1. Recursion

factorial → 5
5 x 4 x 3 x 2 x 1

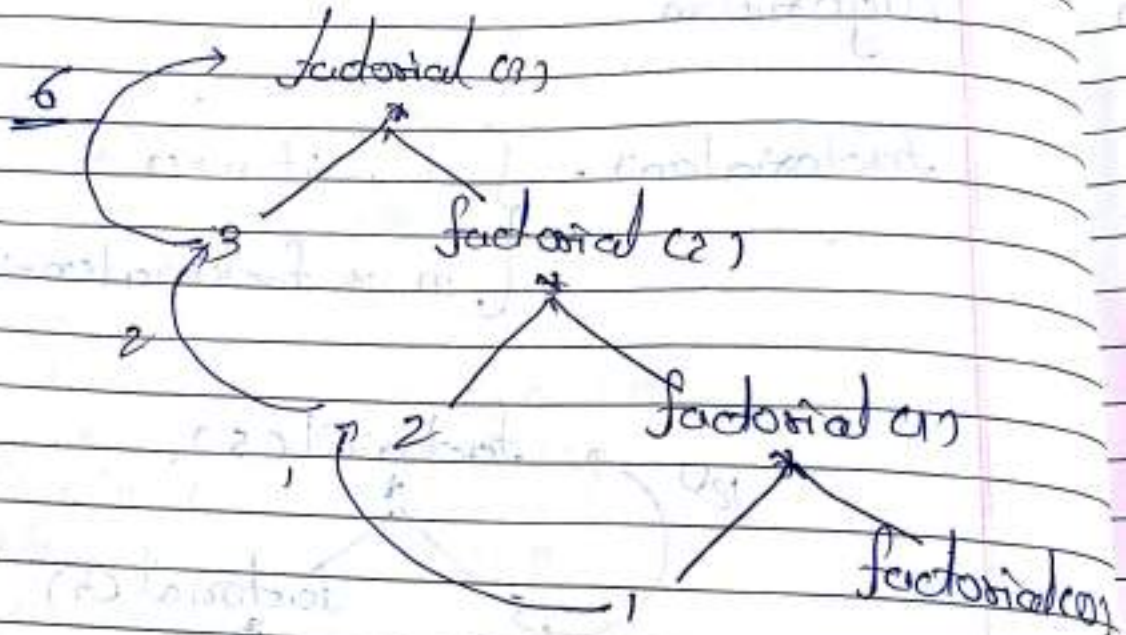
Algorithm

$$factorial(n) = \begin{cases} 1, & \text{if } n = 0 \\ n * factorial(n-1) & \text{otherwise} \end{cases}$$



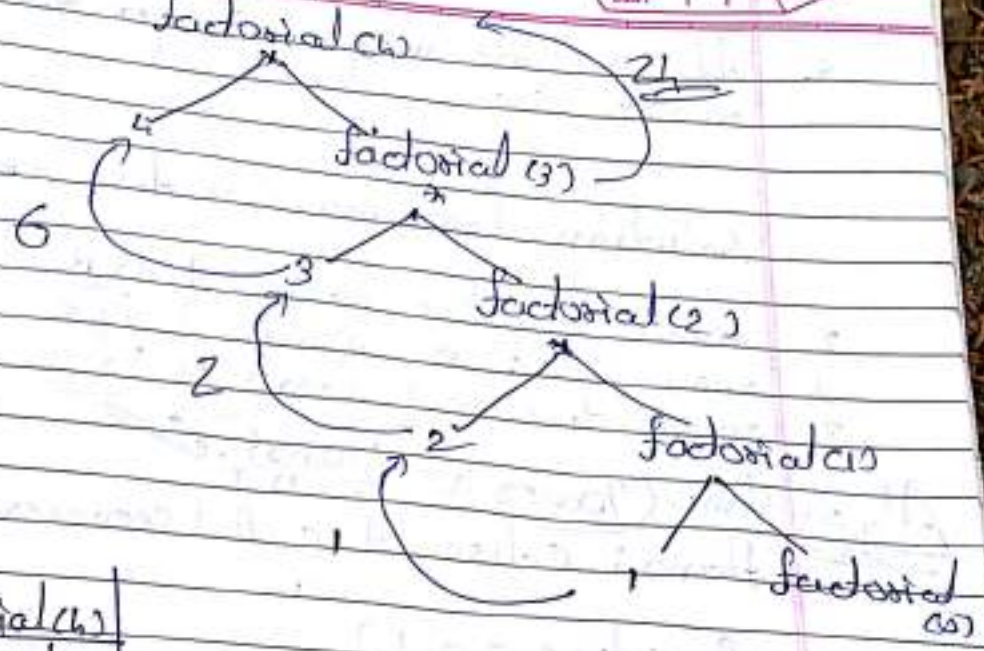
factorial(1)
factorial(2)
factorial(3)
factorial(4)
factorial(5)

stack



factorial(1)
factorial(2)
factorial(3)

stack



factorial(4)
factorial(3)
factorial(2)
factorial(1)

stack

2. Towers of Hanoi :-

Given n ~~disk~~ (discs) of decreasing size stacks on one nedal & given two empty nedal.

It is require to stack all the disc onto destination nedal in decreasing order of size.

Rules

1. Only 1 disc may be moved at a time.
2. A disc may be moved from any nedal to any other nedal.

3. At no time may a longer disc rest upon a smaller disc.

Solution for move n disc from middle A to C.

1. move n-1 disc from A to B
2. move disc n from A to C — A - source
3. move disc n-1 from B to C — C - destination, B - Temporary

Algorithm:- (Tower of Hanoi) (S D T)

Hanoi (disc, A, C, B) generated from

if (disc == 1) — (S) (D)
move disc from A to C
else
Hanoi (disc-1, A, B, C) — (S T D)
move disc from A to C (3) (2)
Hanoi (disc-1, B, C, A) — (T D S)

1. (3, A, C, B) — (S D T)

3 A→C

(2, A, B, C) — (S D T) — 2 A→B

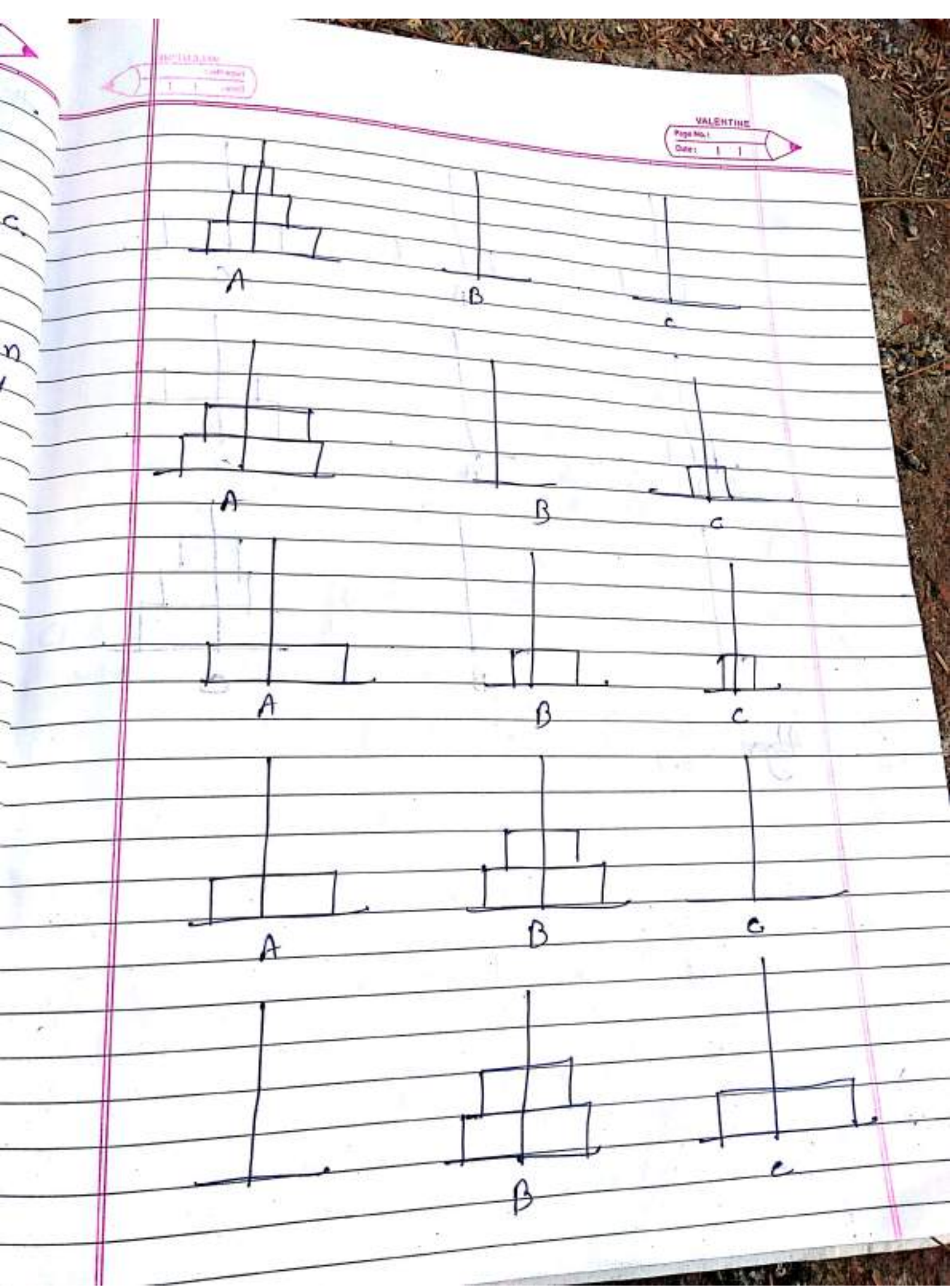
(2, B, C, A) — (S D T) — 2, B→C

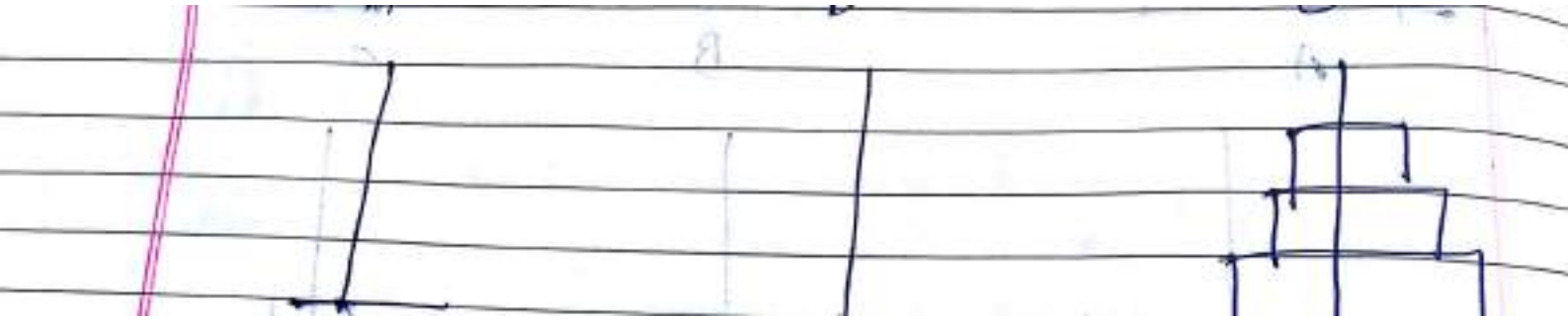
(1, A, C, B) — 1, A→C

(1, C, B, A) — 1, C→B

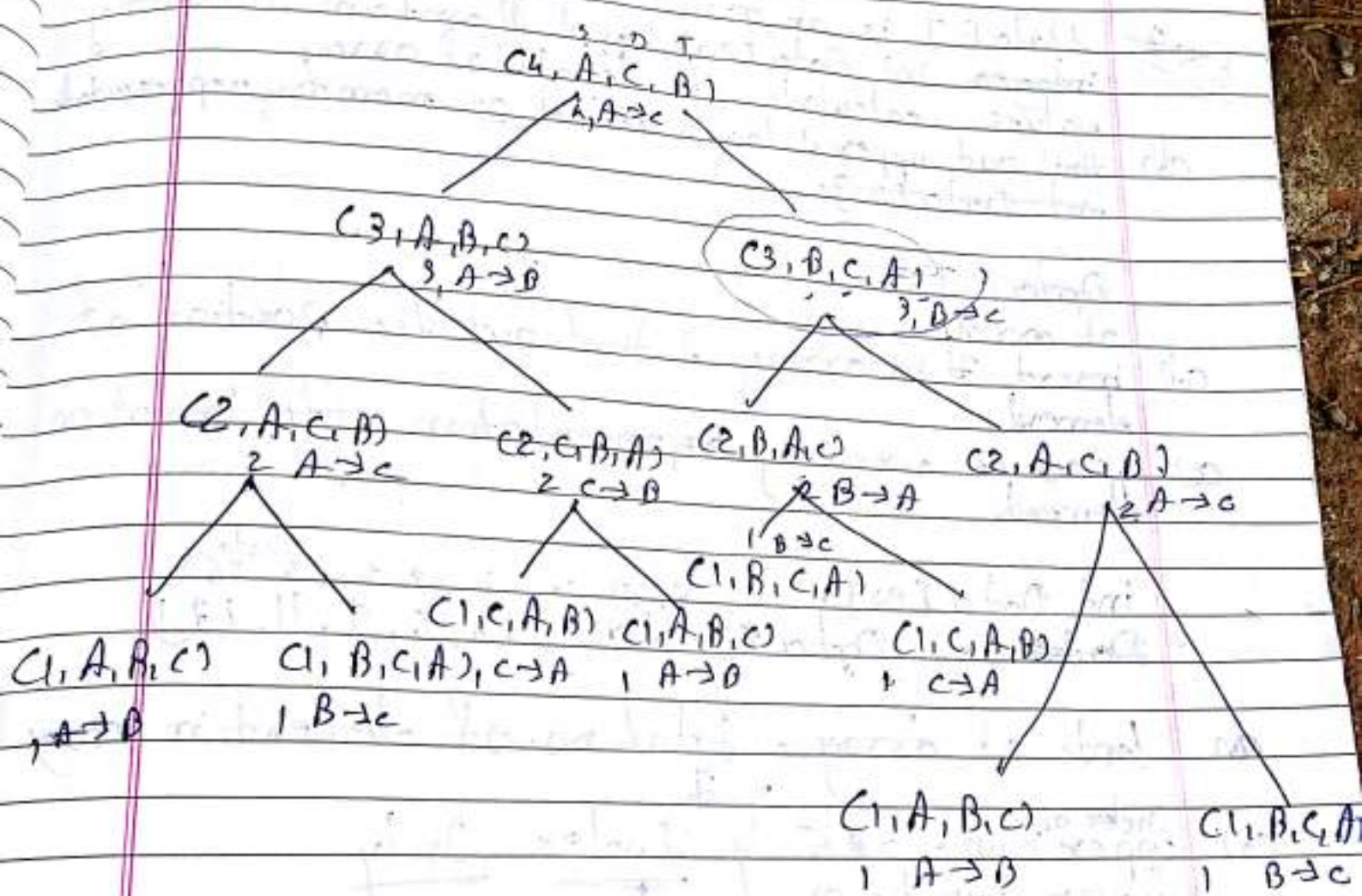
(1, B, A, C) — 1, B→A

(1, A, C, B) — 1, A→C





Hanoi (4, A, B, C)



Ex:- Data[ ] is an array that is declared as int Data[20], and it contain following values. calculate the length of array
(b) find out upper & lower bound (c) memory representation
int Data[20];

Data[ ] =
of array
(d) insert 7 in array & find out the position of 7 element
(e) produce memory representation after insert 7 element

int Data[20];
Data = { 1, 2, 4, 6, 9, 11, 13 }

(a) length of array = total no. of element in array
= 7.

(b) upper bound = 6 (index only)
lower bound = 0
Index Only

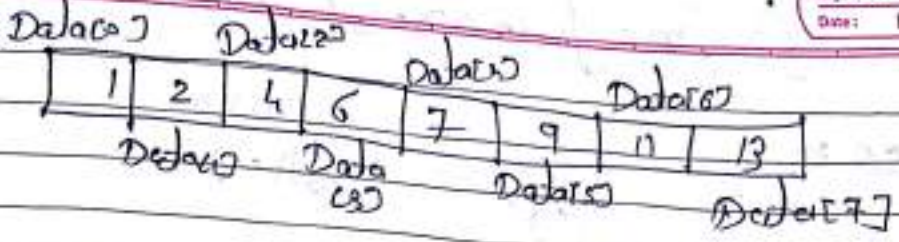
(c) memory representation of array

Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]
1	2	4	6	9	11	13
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]

(d)

1	2	4	6	7	9	11	13

position of 7 element = 4



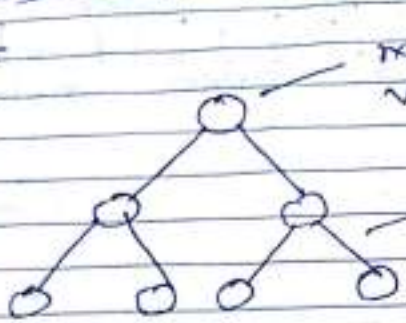
total element = 8
length = 8

Ex-2 int a[30] = {4, 7, 9, 2, 11, 15};

→ find out the length = 6
upper bound = 5
lower bound = 0

Non-linear Data Structure

* Tree :-

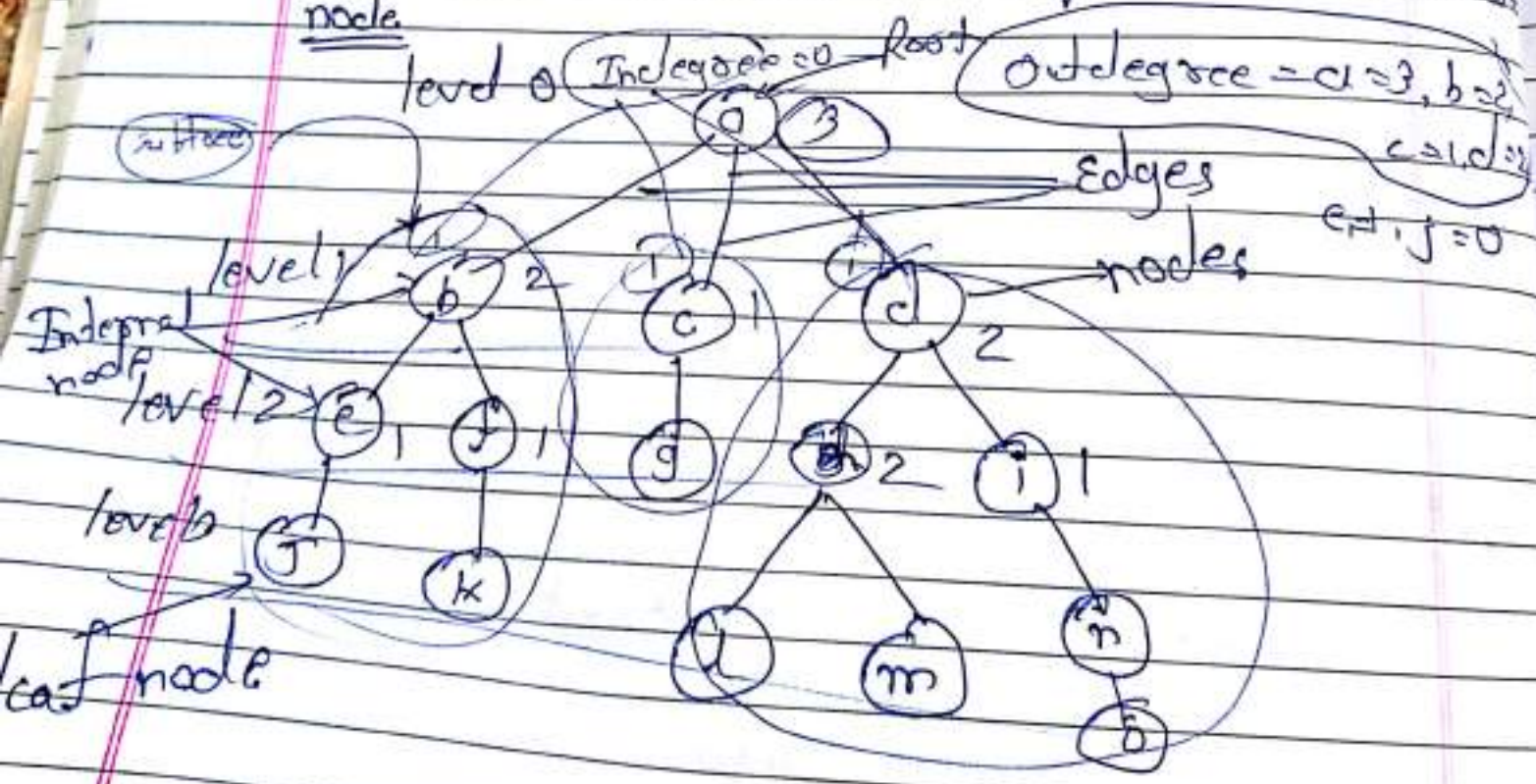


- A tree is a non-primitive, non-linear Data structure made up of nodes or vertices & Edges without having any cycle.

Key terms :-

⇒ Node :

⇒ Each element in hierarchical representation is called node.



⇒ Edges: (link)

Connecting link b/w two node is called edges.

node = 13 = n
edge = 14 = n-

If no. of node is n in tree then there will be max n-1 edges.

⇒ Root node:

It is a specially design first node in the heirarchical represent of the tree

Root node for given tree = a

⇒ level of node:

level of any node is the length of its path from a root node.

⇒ Root node is at level 0, because root has 0 distance from its self.

- Every child node level is defined as parent no.

Degree of node :

⇒ No. of children of node is called the degree of node.

- degree of a=3, b=2, c=1, d=2, ...

⇒ Indegree : Indegree of a node is the no. of edges arriving at node.

root has Indegree 0.

Indegree b=c=d=1

All node in tree except root node has indegree 1

Outdegree :- Outdegree of a node is the no. of edges living the node.

Outdegree a=3, b=2, c=1, d=2, j=0, k=0, l=m=n=0

degree of tree :- maximum degree of a node is called degree of tree.

degree of tree = 3

* leaf node (terminal or external node):-

⇒ A node which has outdegree zero is called leaf node

* Internal node:- (Intermidiate node) Nonterminal or branch node

A node with at least one child is called internal node.

* parent:- A node is a parent if it has a successor (child) node and outdegree > 0.

* child :- A node with predecessor (parent) is a child.

* Symlinks:- level same. both are parent same.

Two or more node with same parent are called as symlinks

Symblinks :- b, c, d are symblinks of each oth

* path :- A ~~symb~~ sequence of consegutive edges is called path.

path of m :- a-b-h-m

- Subtree :- Each child from a node forms a subtree recursively.

- Ancestor :- ~~An~~ Ancestor of a node are all the node along the path from the root to that node

J Ancestor : a, d, h
K " : a, b, f

* Decendent :- A decendent of any node is all the node in the path from a given node to leaf node.

d decendent : h, i, J, m, n
b " : e, f, j, k

* Empty tree or null tree :- A tree with no nodes is called null or empty tree

* hight of node : total no. of edges from leaf node to a particular node in the longest path is called hight of node.

hight of g = 3
" " d = 2
" " h = 1
" " A = 0

Hight of tree :-

hight of root node is said to be hight of tree.

hight of tree = 4

hight of d = 3

c = 1

b = 2

* depth of node :- total no of edges from root node to a particular node is called as a depth of a node. (level method)

depth

a = 0

b = 1

* depth of tree :- total no of edges from root node to a leaf node in the longest path is said to be a depth of tree.

* forest :- It is a group of disjoint trees

If we removed a root node from a tree it becomes forest

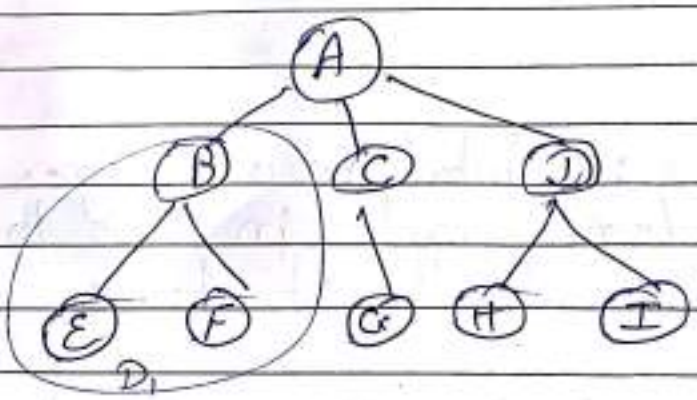
Representation of tree:

(A)

→ A tree contains one or more nodes such that one of node is called root while all others nodes are partican into a finite no. of tree called subtree.

A tree is defined in terms of.

(B, E, F), (C, G), (D, H, I)

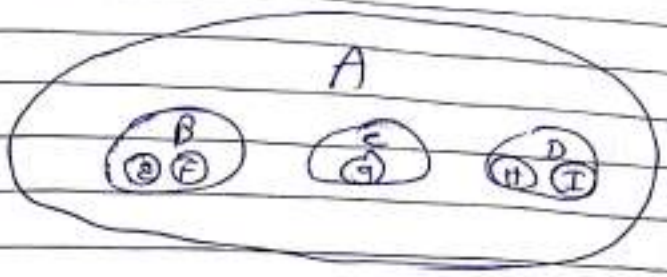


Graphical representation of a tree :-

4 method to represent a tree :

(i) Venn diagram
(ii) Nesting parenthesis
(iii) table of ~~content~~ content
(iv) level number format

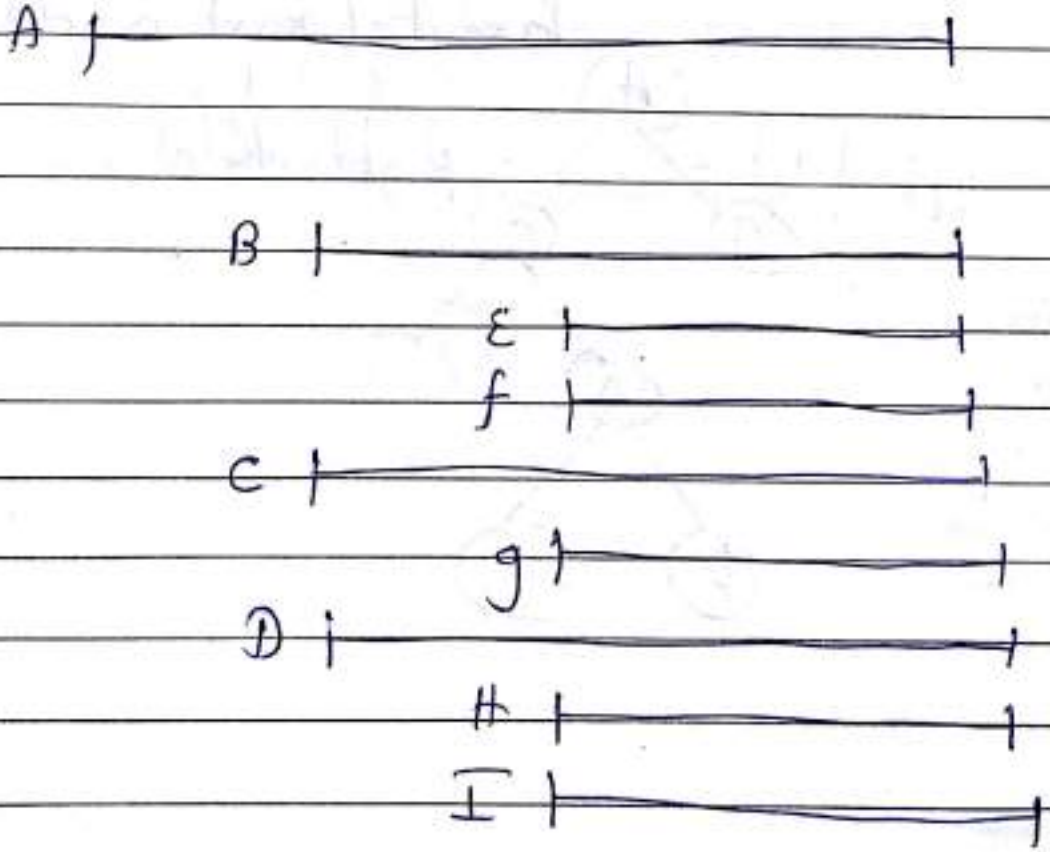
1. Venn diagram:



2. Nesting parenthesis:

(A(B(E)(f))(c(g))(D(H)(I)))

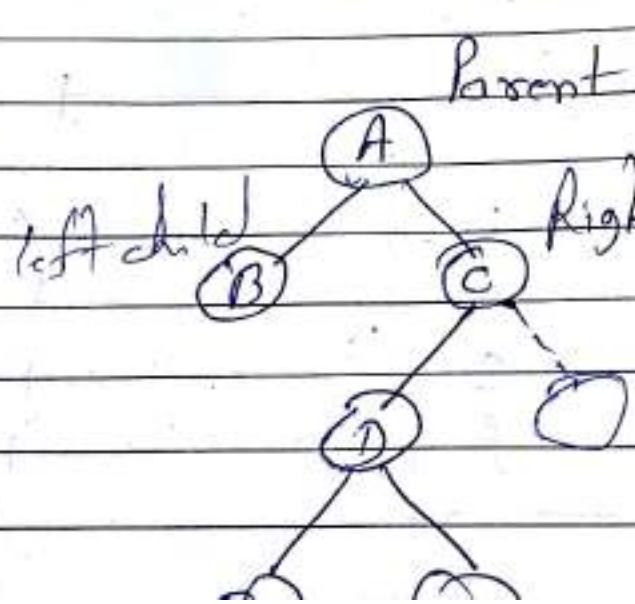
3. table of content:



4. Level number format:

1A
2B
3E
3F
2C
3G
2D
3H
3I

* Binary tree: A tree in which every node can have maximum of two children is called binary tree.

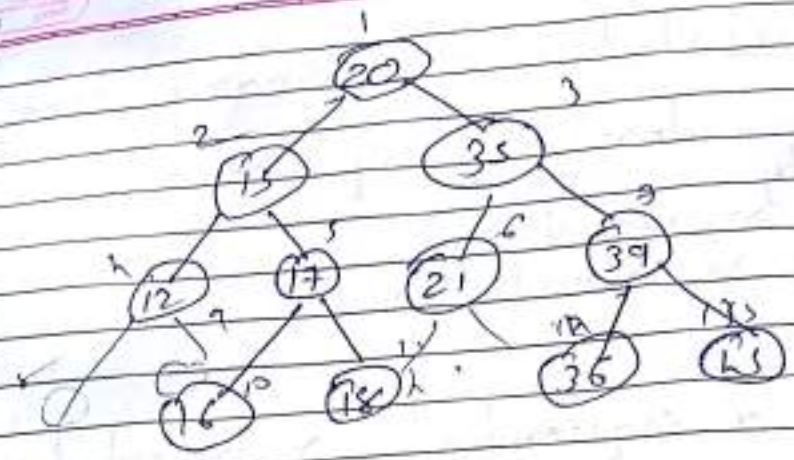


Representation of binary tree :

In computer a binary tree can be maintain either by
1. Array representation
2. linked representation

1. Array (sequential) representation :

- We are using 1-D array to represent a binary tree.
  following are the rules (Array name)
(i) 1-D array called TREE will be use
(ii) Root node of a tree will be stored at first location.
(iii) children of node k will be stored in location (2*k) & (2*k+1)
(iv) max^m size of array = $2^{d+1} - 1$ where d = depth of tree.
(v) An empty tree or subtree is specified using null.



max^m size of array = 15 $(2^4 - 1)$

1	20
2	15
3	35
4	12
5	17
6	21
7	39
8	.
9	.
10	16
11	18
12	.
13	.
14	36
15	45

k=1

C15 → 2k = 2

C35 → 2k+1 = 3

k=2 → 12 → 4

17 → 5

21 = 2k = 6

39 = 2k+1 = 7

• 16 = 10 = 2k

18 = 11 = 2k+1

36 = 14 = 2k

45 = 15 = 2k+1

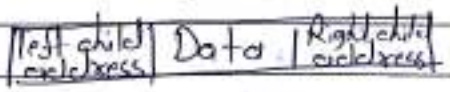
Array

start from 1
2k
2k+1

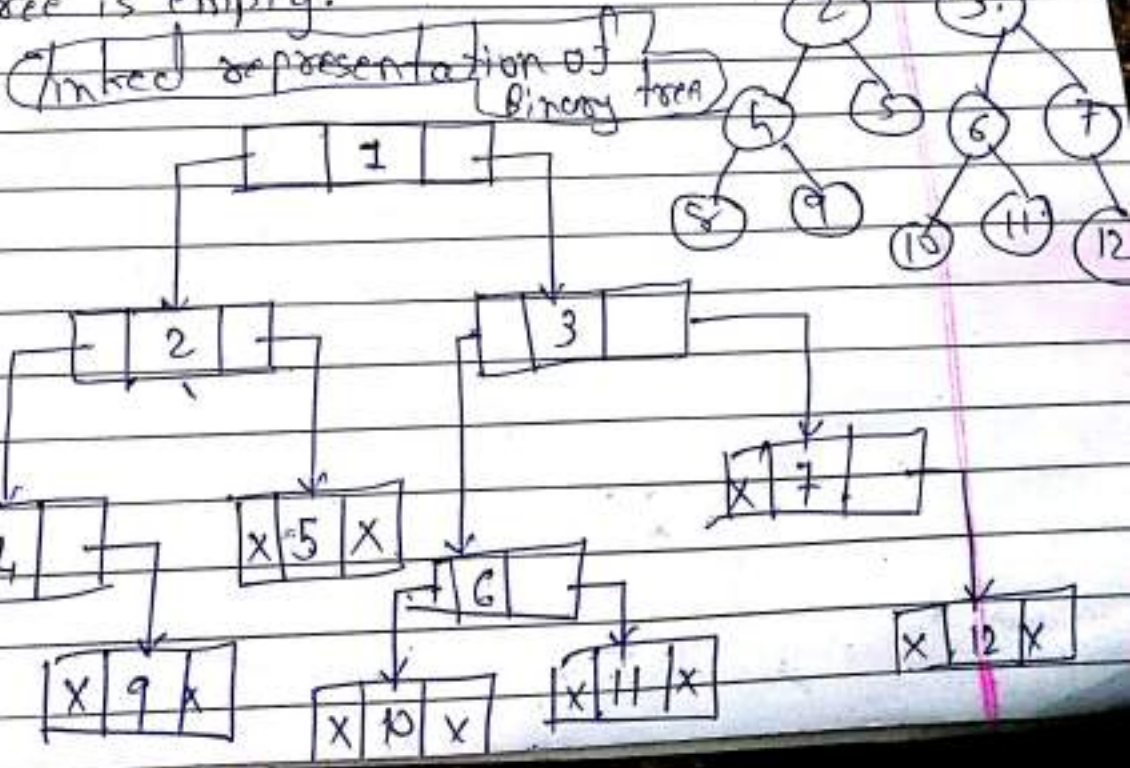
start from 0
2k+1
2k+2

2. linked representation of Binary tree :-

- Doubly link list is used to represent binary tree
- It contain three part
- (i) stored in left child address.
  (ii) data element (actual data)
  (iii) stored in right child address.



Every binary tree has a pointer ROOT which point to root element of a tree
if ROOT = NULL
then Tree is empty.



memory representation :-

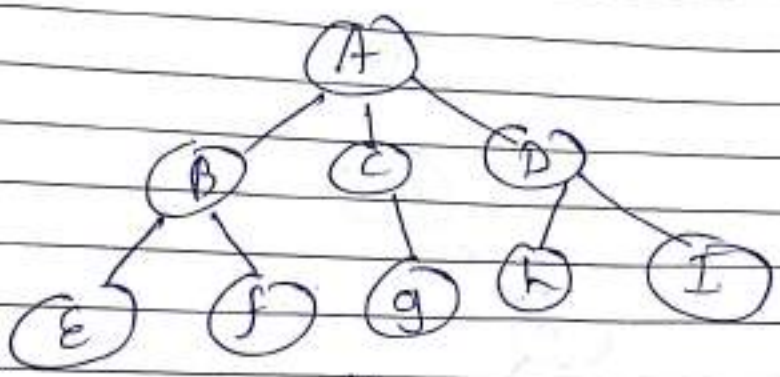
	Left child address	Data	Right child address
1	-1	9	-1
2	6	3	8
3	5	1	2
4			
5	9	2	14
6	7	6	10
7	-1	10	-1
8	-1	7	11
9	12	4	-1
10	-1	11	-1
11	-1	12	-1
12	-1	8	-1
13			
14	-1	5	-1

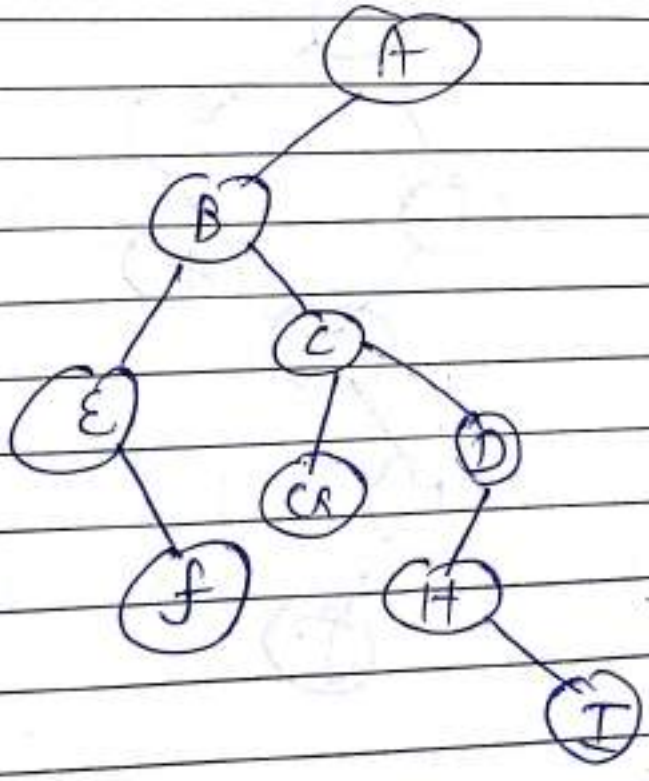
(Leftmost child)

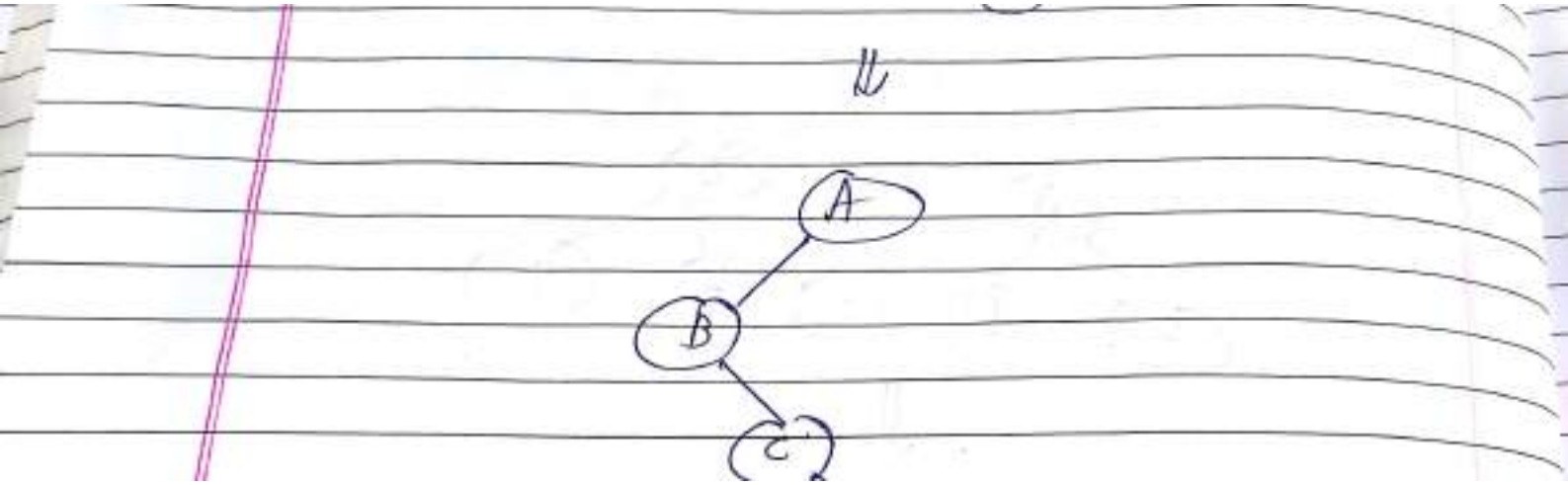
* Generate binary tree from a general tree:

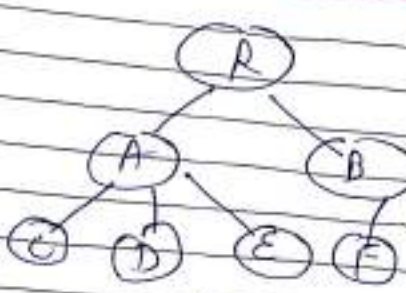
step-1 The first (root node) will work as a root node.
step-2 left child of any node will be as it is.
step-3 Right child of a node will be the right child of its left syblinks

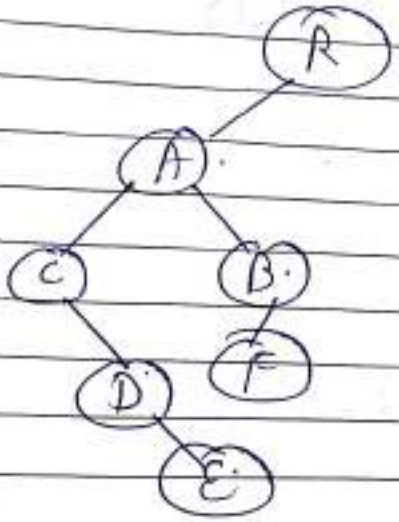
Ex







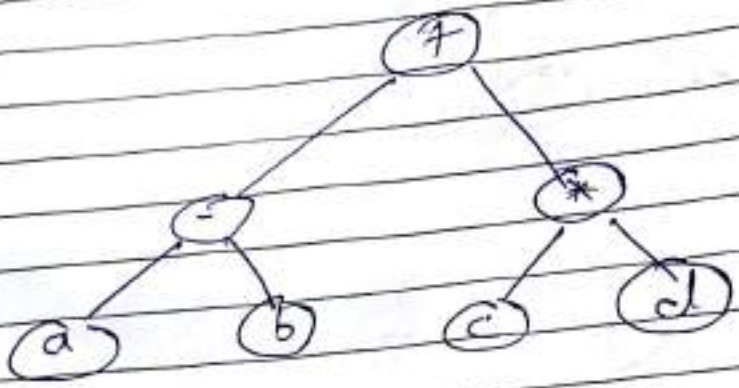




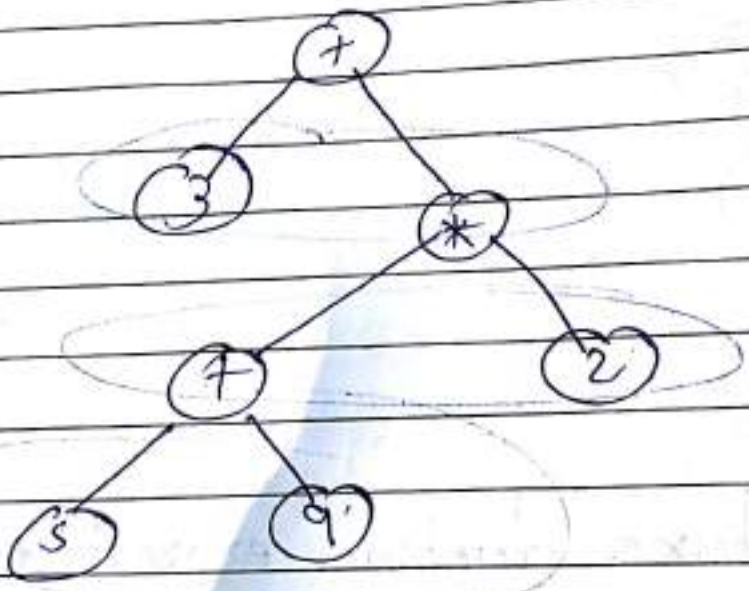
Expression tree:

⇒ - The tree which store algebric expression is called expression tree.
- All operand will be the leaf node
- All operater will be the internal node

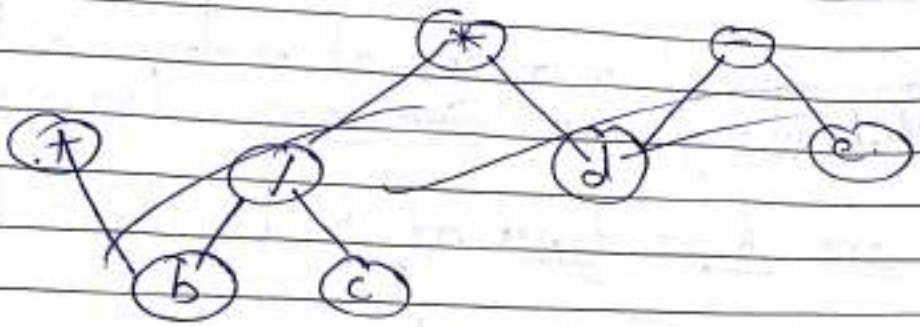
1. (a-b) + (c*d)

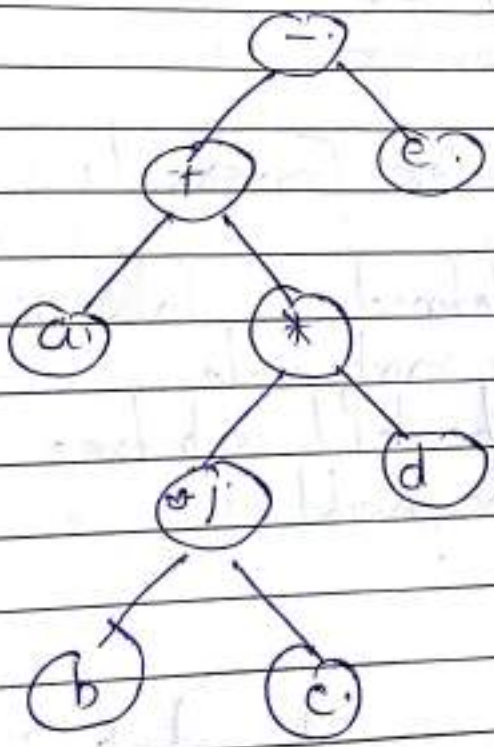


2. 3 + ((5 + 9) * 2)



(9) a + b / c * d - e





* Binary tree traversal :-

Traversing of binary tree

⇒ If the process of visiting each node in a tree exactly ones in a systemetic way.

Definition: Display or visiting order of node in a binary tree is called binary tree traversal.

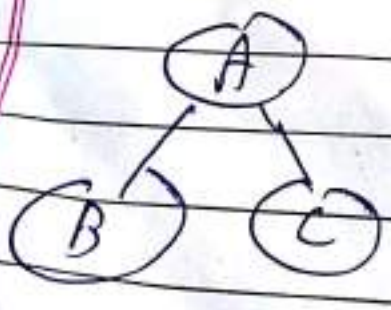
There are three types of binary tree traversal

1. Pre - order Traversal
2. In - order "
3. Post - order "

1. Pre - order Traversal:

It is defined as follows:

1. Process the root node
2. traverse the left sub tree
3. traverse the Right " "



⇒ A, B, C

Pre - D, L, R
In - L, D, R
Post - LRD

Algorithm :-

PREORDER(T) T → pointer for root node

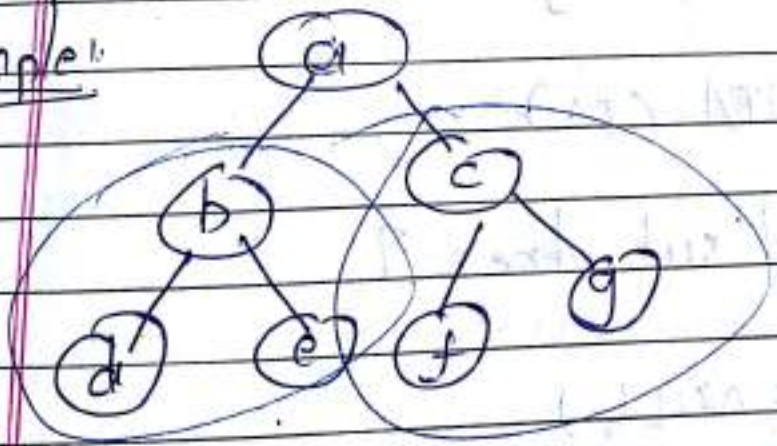
1. [Process the root node]
   If T = NULL
   then write ("Empty tree")
   else
   write (DATA(T))

2. [Process the left sub-tree]
   left ptr
   If (LPTR(T) ≠ NULL)
   then call PREORDER(LPTR)

3. [Process the right sub tree]
   If RPTR(T) ≠ NULL
   then call PREORDER(RPTR)

4. [Finish]
   Return

Example :-



a, b, d, e, c, f, g

2. In-order traversal:

It is defined as follows:

- traversing the left sub tree
- visiting the root node
- traversing the right sub tree

Algorithm:

INORDER(T)

1. [Process the root node]

   If T = null
   then write ("Empty tree");

2. [Process the left subtree]

   If (LPTR(T) ≠ NULL)
   then call INORDER (LPTR(T))

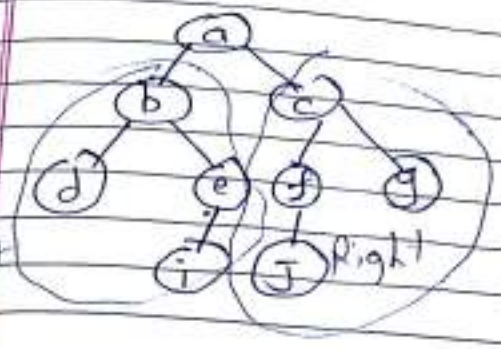
3. [Process the root node]

   write (DATA (T))

4. [Process the right sub tree.]

   If (RPTR(T) ≠ NULL)
   then call INORDER (RPTR(T))

[Finish] Return



d, b, i, e, a, j, f, c, g

3. Post-order traversal:
root node last always
- traversal left sub tree
- " Right " "
- process the root node

Algorithm:

POSTORDER(T)
check for empty tree.

1. [Process the root node]
if T = null
then write ("Empty tree");

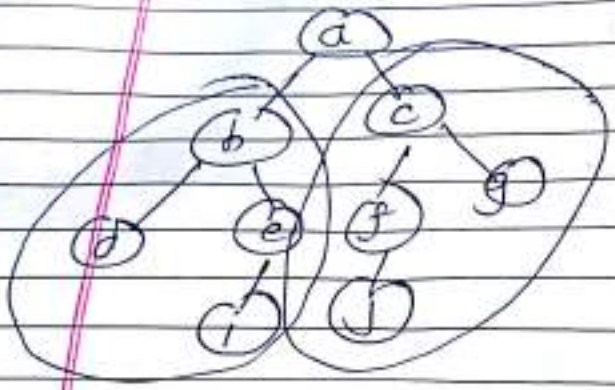
2. [Process the left subtree]

If (LPTR(T) ≠ NULL)
then call POSTORDER (LPTR(T))

3. Process the right subtree
if RPTR(T) ≠ NULL
then call POSTORDER (RPTR(T))

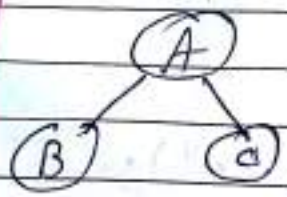
4 Process the root node
write DATA(T)

5. finish
Return



d, i, e, b, j, f, g, c, a

1. Inorder, preorder, postorder

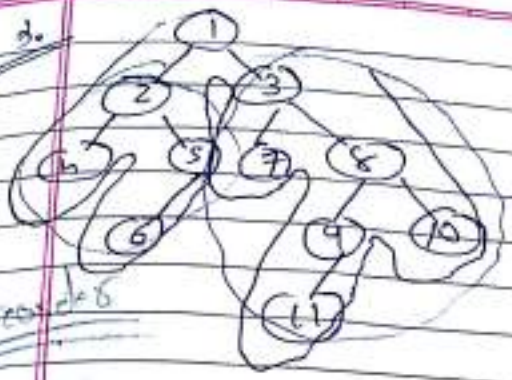


preorder: A, B, C
Inorder: B, A, C
postorder: B, C, A

2.

a, b, d, e, i, c, f, g
preorder: a, b, c, d, e, f, g, h, i
Inorder = d, b, i, e, a, j, f, c, g
postorder: d, i, e, b, j, f, g, c, a

a.



Preorder

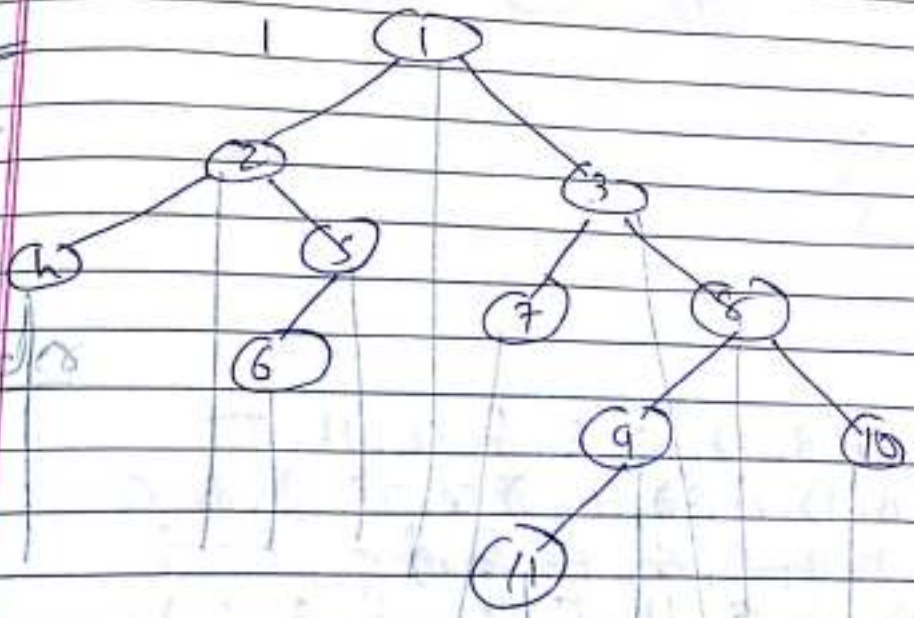
(b)

Preorder: 1, 2, 4, 5, 6, 3, 7, 8, 9, 11, 10

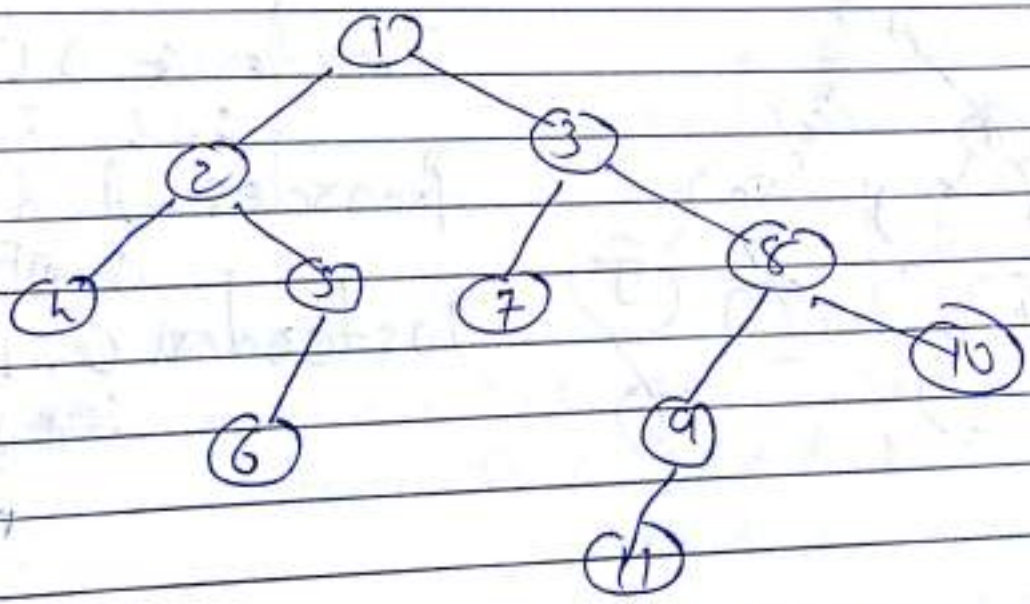
Inorder: 4, 2, 6, 5, 1, 7, 3, 11, 9, 8, 10

Postorder: 4, 6, 5, 2, 7, 11, 9, 10, 8, 3, 1

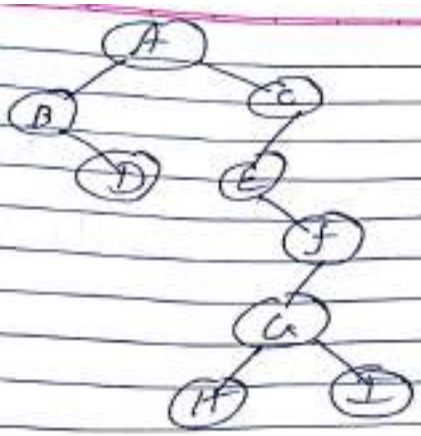
b.



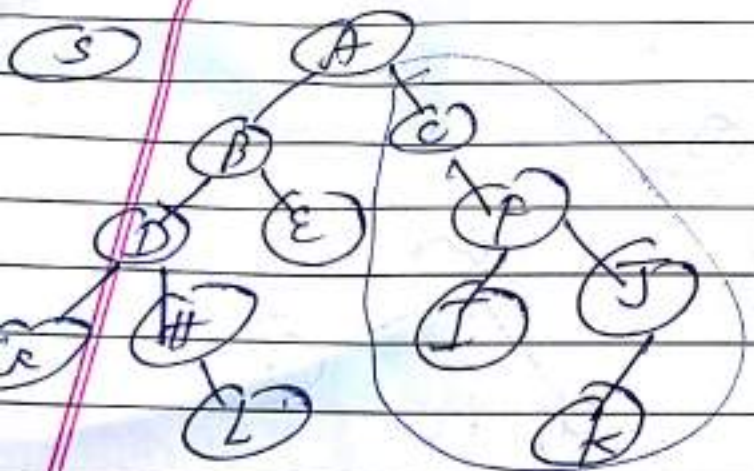
Inorder



Postorder:



Preorder: A, B, D, C, E, f, G, H, I
Inorder: B, D, A, E, H, G, I, f, C
Postorder: D, B, E, C, H, I, G, f
D, B, E, H, I, G, f, E, C, A



Inorder: G, D, L, H, B, E, A, C, L, F, G, J
Preorder: A, B, D, G, H, L, E, C, F, I, G, J, K, J
Postorder: G, L, H, D, E, B, I, K, J, F, A

char infix[100], postfix[100]

j=0;

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

{

if(infix[i] >= 'a' && infix[i] <= 'z')

{

postfix[j] = infix[i];

j++;

}

else if(+, -, *, /, ^)

{

(top >= infix next)

{

postfix[j] = pop();

j++;

}

delete()

{ if(f == -1)

{ printf("underflow");

}

else

{ t = queue[f];

if(f == r)

{ f = r = -1;

}

else { f = f + 1; printf("%d", t); }

}

(B) push, (stack)

i. insert(int y)

{

if(r >= n)

{ printf("overflow");

}

else

{

r = r + 1

queue[r] = y;

if(f == -1)

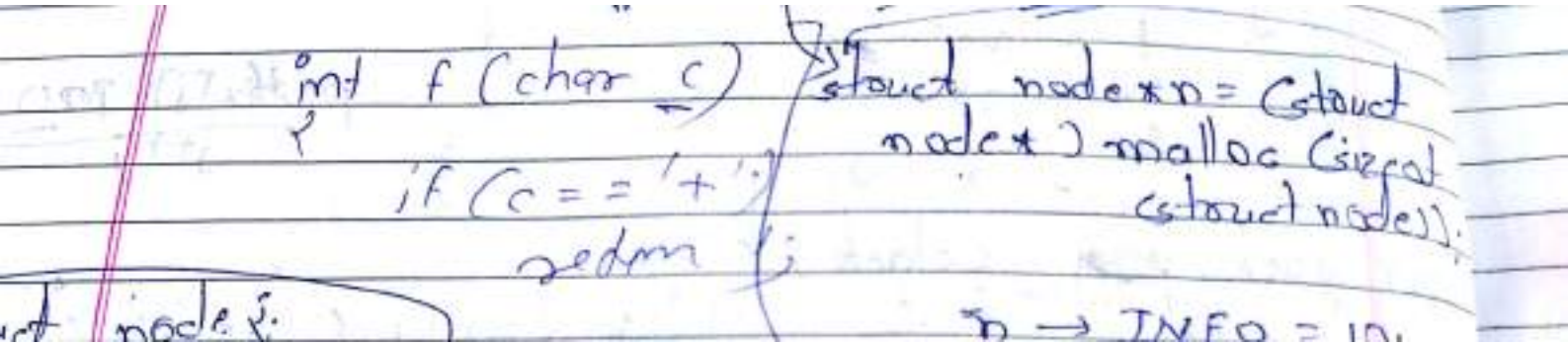
{

f = 1;

}

}

}



else
{
struct node *temp = first;
while (temp->next != NULL)
{
temp = temp->next;
}
postfix temp->next = n;

int traverse

char scan	stack
	#
A	#A
+	#+
B	#+B
-	#-
C	#-C
*	#*
D	#*D

A
A
AB+
AB+
AB+C-
AB+C-D*

	#		
A	#A		
*	#*	A	A+B-C*D
B	#*B	A	
+	#+	AB*	AB+C-D*
C	#+C	AB*C+	

AOB-(CD*)
A+B-T
(AB+)-T
OU-T
OUT-
AB+CD*-

1-2^3^45+6*78