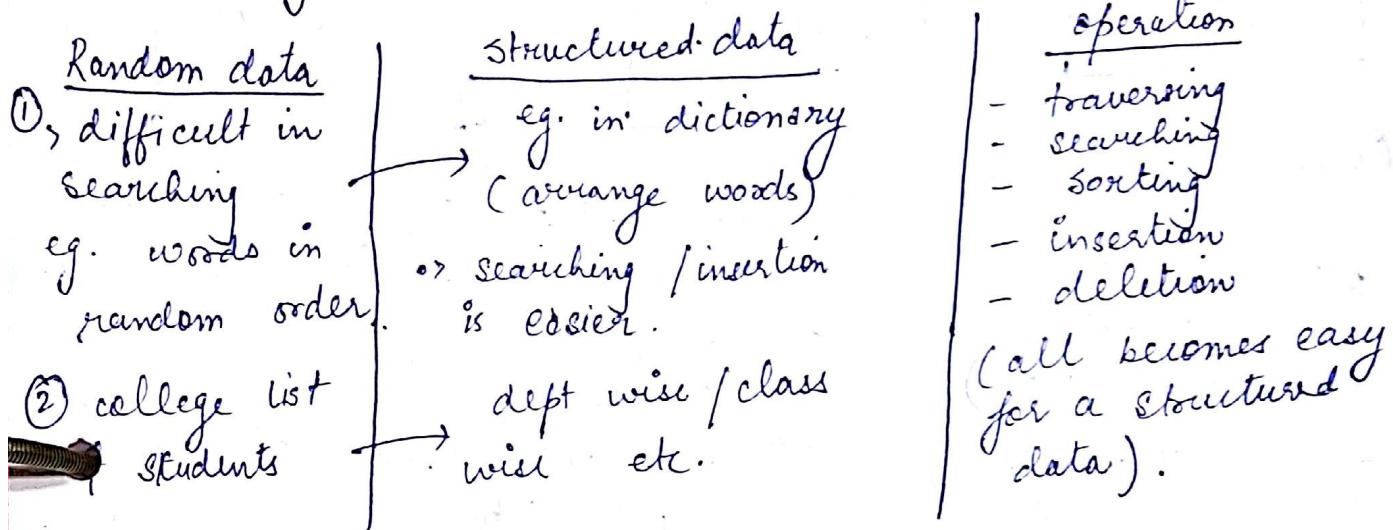


Introduction to DS

A Data structure is a way to store & organize data in a computer, so that it can be used efficiently.



Data structure is used by →

- OS
- compiler
- DBMS
- Data communication etc. (related to data).

Programming methodologies & design of algo

It deals with different methods of designing program

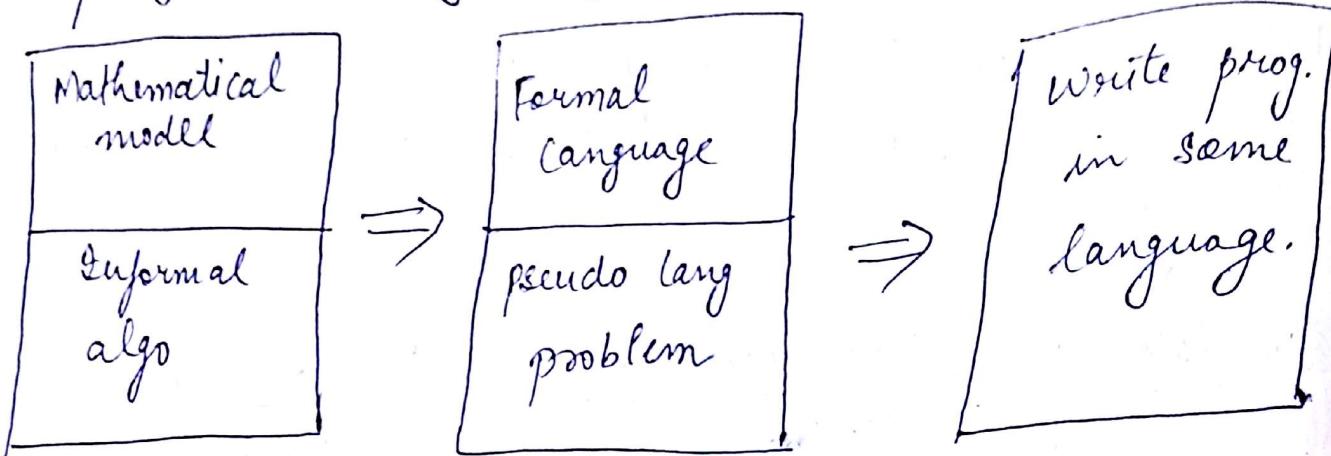
* Program: An implementation of an algorithm in some programming language.

* Algo: It is a step-by-step finite sequence of instⁿ, to solve well defined computational problem.

* DS: way of organizing data.

$$DS = \text{organized data} + op^n$$

steps for creating prog:-



Ques. Identify those students who passed the exam (50% & above).

Solⁿ:

$m = 60$

$\text{if } (m \geq 50)$
"student pass")

↓

step 1: Enter marks

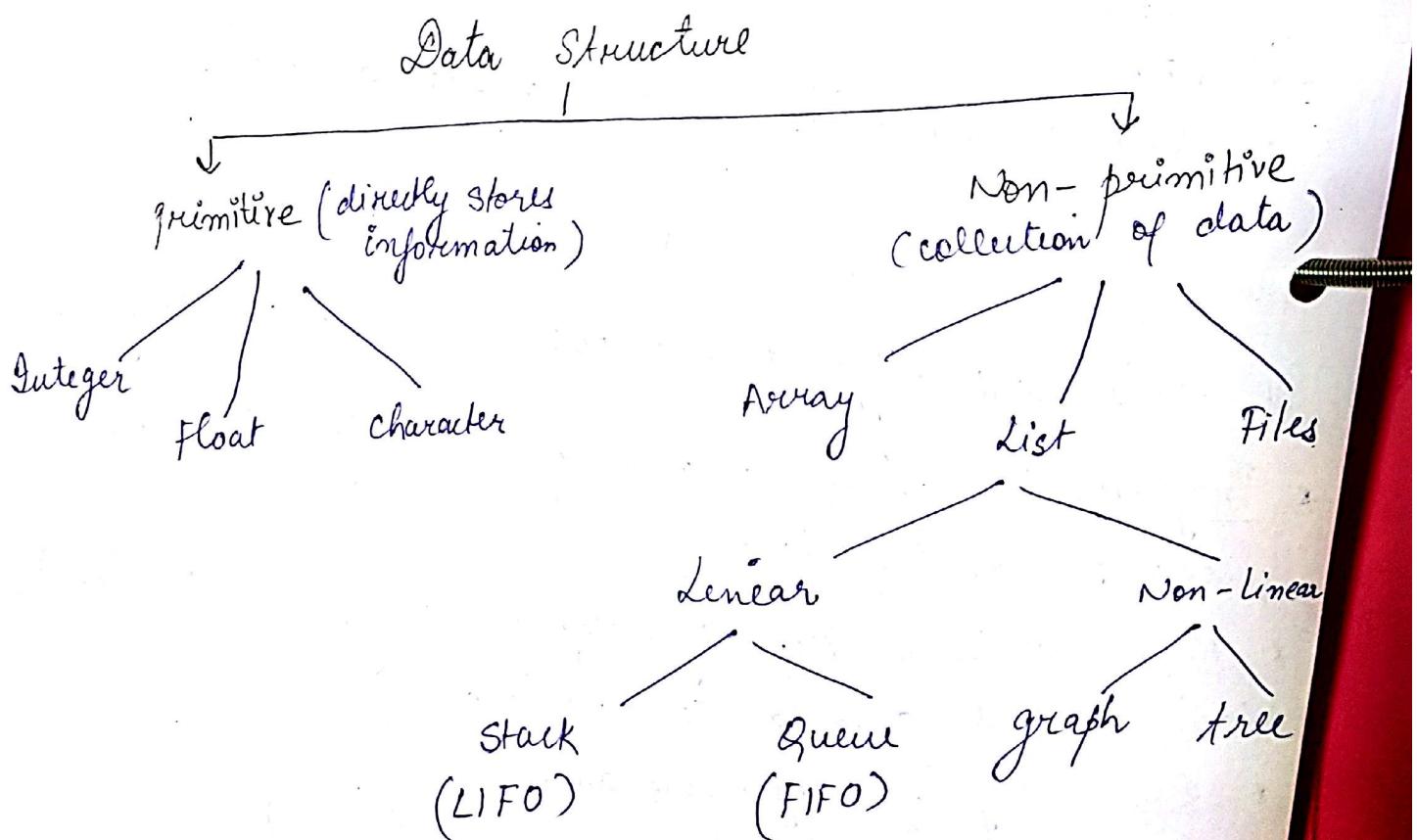
step 2: if marks ≥ 50
print "pass"

step 3: Repeat 1 & 2

step 4: End

↓

write in c lang



Abstract Data type: > we are not concerned about the internal interface. (2)

we directly use them as per our need.

Stack / Queues are ADT

e.g. driving a car

ADT is a logical description of how we view the data & op^rs that are allowed without knowing their implementation.

Array: It is a collection of similar data type.

int / char / float

e.g. int $x_1, x_2, x_3;$

but in array \Rightarrow int arrayname [size]

e.g. int $a[5];$ // declaration of array

int $a[5] = \{ 4, 6, 8, 5, 3 \}$ // initialization of array
↓ ↓ ↓ ↓ ↓ // indexing in C

No. of elements = UB - LB + 1 // lower & upper bounds
= 4 - 0 + 1 = 5.

e.g. LB = -2, UB = 3, find no. of elements
 $N = 3 - (-2) + 1 = 6$

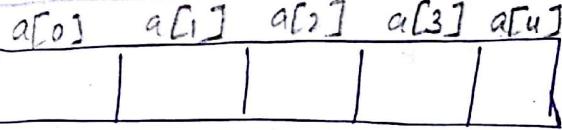
1-D Array

- linear array : list of finite number n of homogeneous data elements.

address calcul'n

int $A[5]$

[Start index = 0]



200 202 204 206 208 210
(base address)

$$A[i] = \text{Base address} + i \times (\text{data type size})$$

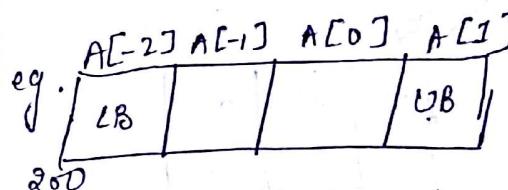
$$A[2] = 200 + 2(2) \\ = 204.$$

int $A[LB] \dots A[UB]$

Start index = LB

now,

$$A[k] = \text{Base } A[LB] + (k - LB) \times \text{size}$$



$$A[0] = 200 + (0 - (-2)) \times 2 \\ = 200 + 4 \\ = 204$$

Q. $A[-4:3]$

$$N = 3 - (-4) + 1 = 8$$

For 2D arrays,

e.g. $A[-2:2, 2:22]$

Solⁿ, Find L_1 and L_2

Total elements = $L_1 \times L_2$

$$\therefore L_1 = 2 - (-2) + 1 = 5$$

$$L_2 = 22 - 2 + 1 = 21$$

$$N = L_1 \times L_2 = 5 \times 21 = \underline{105}$$

Ques. $A[2:8, -4:1, 6:10]$, find elements

$$L_1 = 8 - 2 + 1 = 7$$

$$L_2 = 1 - (-4) + 1 = 6 \quad \therefore N = 7 \times 6 \times 5$$

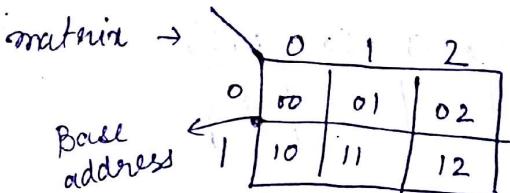
$$L_3 = 10 - 6 + 1 = 5 \quad = 210$$

Array Representation

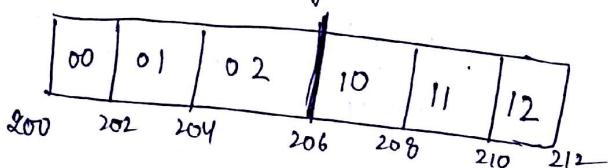
2-D array representation, also known as "matrix"

① Row-Major

e.g. `int A[2][3]` // elements = $2 \times 3 = 6$



For row-major, first rows are traversed.



② column major ④

00	10	01	11	02	12
200	202	204	206	208	210

q. $A[m][n]$

address :

$$A[i][j] = \text{Base} + (i \times m + j) \times \text{size}$$

$$A[i][j] = \text{Base} + (j \times m + i) \times \text{size}$$

$$A[0][1] = 200 + (1 \times 2 + 0) \times 2 \\ = 200 + 4 = 204.$$

$$\text{eg. } A[1][0] = 200 + (1 \times 3 + 0) \times 2 \\ = 200 + 6 = 206$$

Sparse Matrix (2-D array) : majority of elements are zero. very few non-zero elements.

→ it reduces scanning time.

eg. $A[100][100]$: elements = 10,000
(we need to scan all 10000 elements to find where values exist. Instead, we use sparse matrix).

Three column form

Representation

linked list

①

3 columns =

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

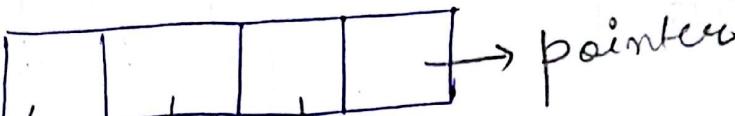
rows, col, values

1st row:
Total
row &
values

Rows	col	value
4	4	6
0	1	2
1	0	4
1	2	3
2	1	1
2	3	1
3	1	2

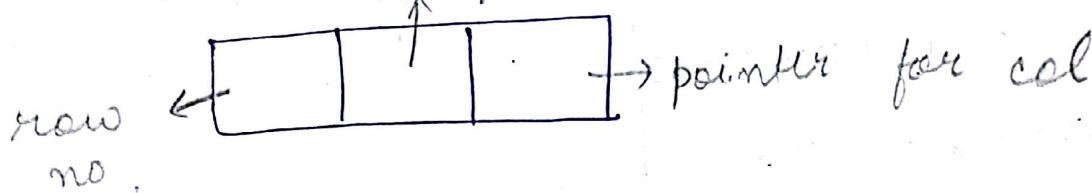
② In Link List representation, we require three structures.

a) Head node :

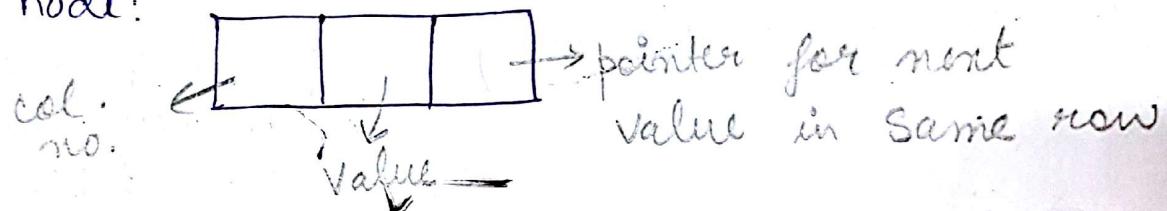


Total New
Total col
Total non-zero val

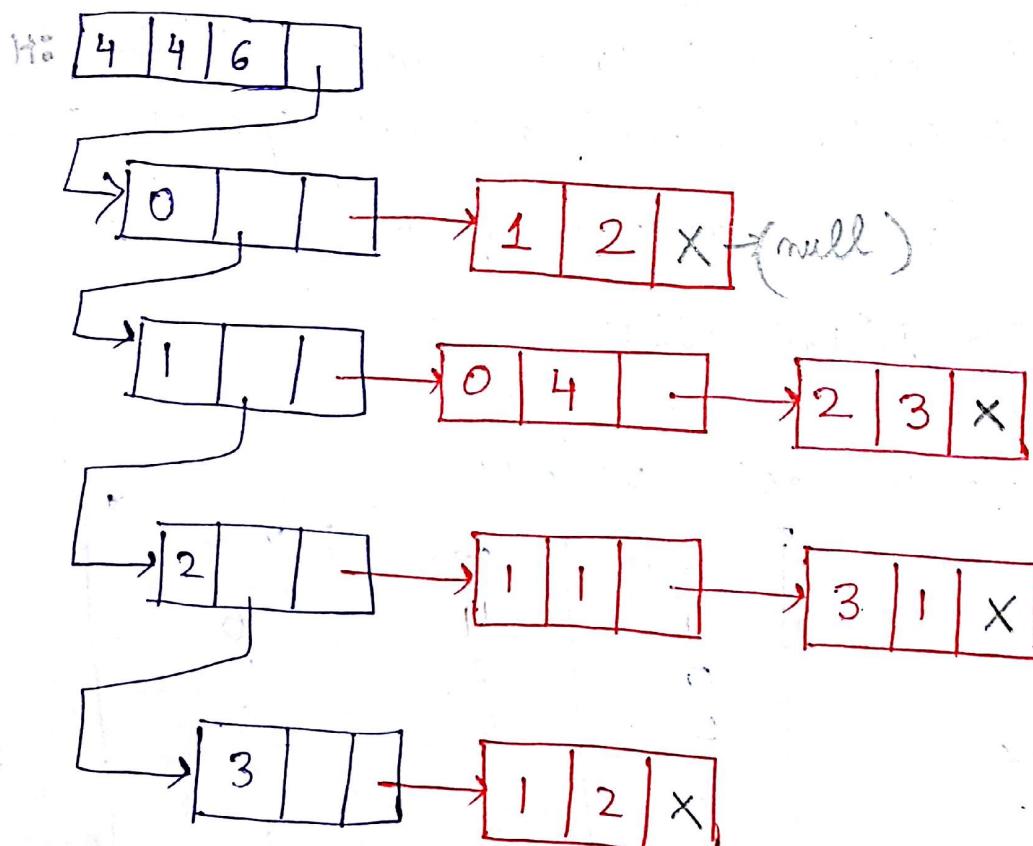
b) Row node : per for next row



c) column node:



eg: same →



(5)

array operations:

Traversing

LB	UB		
0	1	2	3
4	10	12	15

Deleting

Insertion

0	1	2	3
4	10	12	15

\rightarrow [4] 12 15

Step 1: Set item := A[K] // to be deleted

2: Repeat for i = K to N-1

Set A[i] := A[i+1]

[end loop]

3: Set N := N - 1

4: Exit

- 2: Process A[i]
- 3: Exit [End of Loop]

Insertion

• Insertion in unsorted array

Let k be the locⁿ where item is to be inserted.

k = 2

0	1	2
4	10	15

0	1	2	3
4	10	12	15

Step 1: Set i = N

i = 2

2: Repeat ^{while} i >= k

3: Set A[i+1] := A[i] // moving the element \rightarrow Right

4: Set i := ^{i - 1} [end loop]

5: Set A[k] := item.

6: N := N + 1;

7: Exit

Array-Linear Array - int A[K]

$$\text{Array Length} = UB - LB + 1$$

Lower bound
Upper bound

~~Loc~~ $LOC(A[k]) = \text{base address} + w(k-LB)$

Location of k^{th} element of array A whose address is to be calculated

Ques) Base = 200, LB = 1932, $base[k] = 1965$, $w = 4 \text{ words/memory unit}$

$$LOC(base[1965]) = 200 + 4 \{ 1965 - 1932 \}$$

$$= 200 + (4 \times 33) = \underline{\underline{332}}$$

Operations on Data Structures -1) TRAVERSING-

Step 1 Set $K := LB$ [Initialize counter]

Step 2 Repeat steps 3 & 4 for $K \leq UB$

Step 3 Apply process to $LA[k]$ [visit element]

Step 4 $K := K + 1$ [Increase counter]

[End of Step 2 loop]

Step 5 Exit

2) INSERTION - \rightarrow length of array
 \rightarrow location

INSERT ($LA, N, K, ITEM$) -

Step 1 Set $J := N$ [Initialize counter]

Step 2 Repeat steps 3 & 4 while $J \geq K$

- Step 3 Set $LA[J+1] := LA[J]$ \rightarrow If this step moves elements to next position
[Move] to create vacant spot for new element
- Step 4 Set $J := J-1$ [Decrease counter]
- [End of step 2 loop]
- Step 5 Set $LA[k] := ITEM$
- Step 6 Set $N := N+1$
- Step 7 Exit

3) DELETION-DELETE($LA, N, K, ITEM$)

- Step 1 Set $ITEM := LA[K]$, counter $\#$ Element to be deleted is stored
- Step 2 Repeat for $J = K$ to $N-1$ as ITEM only to save it at
an alternate location.
- & Set $LA[J] := LA[J+1]$ \rightarrow This allows us to print the
value \neq deleted (at the end
of program) if required
- [End of loop]
- Step 3 Set $N := N-1$
- Step 4 Exit

4(A) LINEAR SEARCH-LINEAR [$DATA, N, ITEM, LOC$]

- Step 1 Set $DATA[N+1] = ITEM$ $\#$ Only to check at end if a match is found
- Step 2 Set $LOC := 1$ [Initialize counter]
- Step 3 Repeat while $DATA[LOC] \neq ITEM$
- Set $LOC := LOC + 1$
- T End of loop
- Step 4 If $LOC = N+1$, then set $LOC := 0$ [Successful?]
- Step 5 Exit

Worst case complexity: $f(n) = n+1$

4-B) BINARY SEARCH:-

Location of $\begin{cases} \text{BEG} = LB \\ \text{END} = UB \end{cases}$

$$\text{MID} = \text{INT} \left(\frac{\text{BEG} + \text{END}}{2} \right)$$

BINARY (DATA, LB, UB, ITEM, LOC)

Step 1 Set $\text{BEG} := LB$, $\text{END} := UB$, $\text{MID} := \left(\frac{\text{BEG} + \text{END}}{2} \right)$

Step 2 Repeat steps 3 & 4 while $\text{BEG} \leq \text{END}$ and $\text{DATA}[\text{MID}] \neq \text{ITEM}$

Step 3 If $\text{ITEM} < \text{DATA}[\text{MID}]$ then:

 Set $\text{END} := \text{MID} - 1$

else:

 Set $\text{BEG} := \text{MID} + 1$

[End of if structure]

Step 4 Set $\text{MID} := \text{INT} \left(\frac{(\text{BEG} + \text{END})}{2} \right)$

[End of step 2 loop]

Step 5 If $\text{DATA}[\text{MID}] = \text{ITEM}$ then:

 Set $\text{LOC} := \text{MID}$

else

 Set $\text{LOC} := \text{NULL}$

Step 6 Exit

Worst case complexity :

$$f(n) = \lceil \log_2 n \rceil + 1$$

Algorithm Notations -

Finite step-by-step instructions

1) Identifying Number - To identify algorithms & distinguish them from other algorithms

eg. Algo 4.1 Insertion of elements
no. description (brief)

2) Steps, control & exit - Numbering the steps

eg. Step 1 xxx
 numbering Step 2 Go to step 7 → control
 of steps Step 3 xxx when loops are to
 : be used
 :
 :
 Step n Exit / End / Stop

3) Comments - To simply or explain any particular step
 # always enclosed in square brackets

4) Assignment statement - To assign values to variables

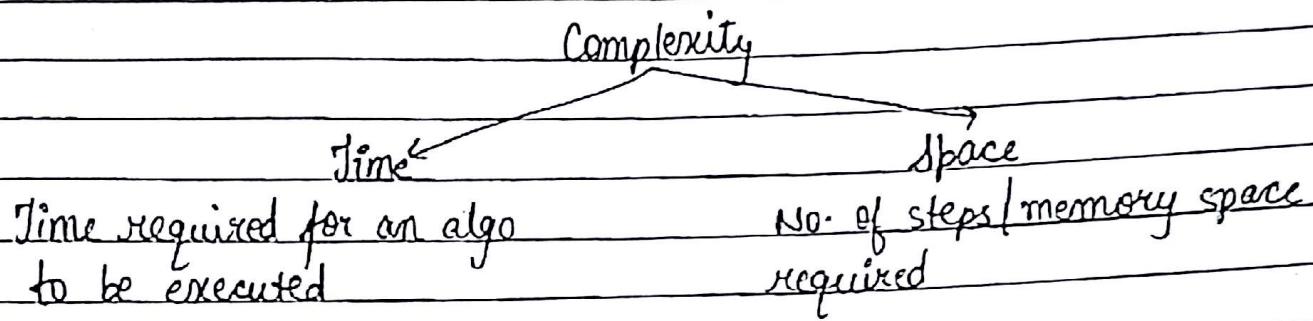
$a := 5 \Rightarrow$ assigns value 5 to a
 $a = b \Rightarrow$ a equals b

5) Variable Names - Always written in CAPITAL LETTERS

6) Input and Output - Read: (Input)
 Write: (Output)

Complexity - The choice of an algorithm depends on its complexity
↓

Measure of effort required to execute an algorithm



→ Time-Space Trade Off -

- Depends on time & space complexity
- It is not possible for both these complexities to be less for any particular algorithm.

→ Complexity -

Suppose 'M' is an algorithm and 'n' is the size of the input data.
The complexity of algo M is $f(n)$ which gives the running time and storage space requirement of algo M in terms of 'n' that is the size of input data.

→ Big O Notation -

Let the complexity of an algo be $f(n)$.

Big O Notation is the characterization scheme that allows us to measure the properties of algs such as performance and memory required.

In this notation, we eliminate the constant factors.

Therefore, the complexity $f(n)$ increases as n increases.

$$\text{eg. } f(n) = \alpha n^2 + 3n$$

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

$$\text{eg. } f(n) = 3n^3 + n^2 + 2n$$

$$O(n) = n^3$$

$$\text{eg. } f(n) = 3n^3 + 1n^2 + \log n$$

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

$$\text{eg. } f(n) = 4n^3 + \log n + 3e^n$$

$$O(n) = e^n$$

$$\text{eg. } f(n) = 2$$

$$O(n) = 1$$

Three cases of algo complexity -

1) Worst case : Max. value of $f(n)$ for every input
ie. algo runs correctly only on a few input sets

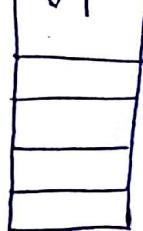
2) Average case : Expected value of $f(n)$

3) Best case : Min. value of $f(n)$

Stacks

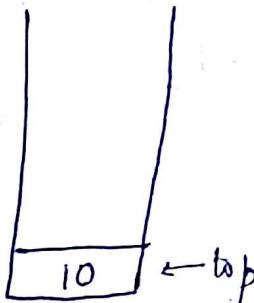
Stack is a linear data structure in which operations are performed based on LIFO.

push ↓ ↑ pop

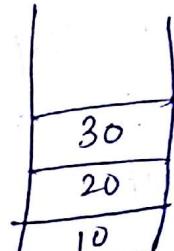


top = -1

(stack is empty)



top = 0



top = 2

application of stacks: . recursion

. MS - word (undo)

. Backtracking

Operations of stack: push pop

① PUSH

- After every push, top is incremented by one.

- In case stack is full, its called overflow cond?

algo: → PUSH (stack, Top, MaxSTK; Item)

1. If $\text{Top} = \text{MaxSTK}$, print overflow
exit

2. Read Item

3. Set $\text{Top} = \text{Top} + 1$

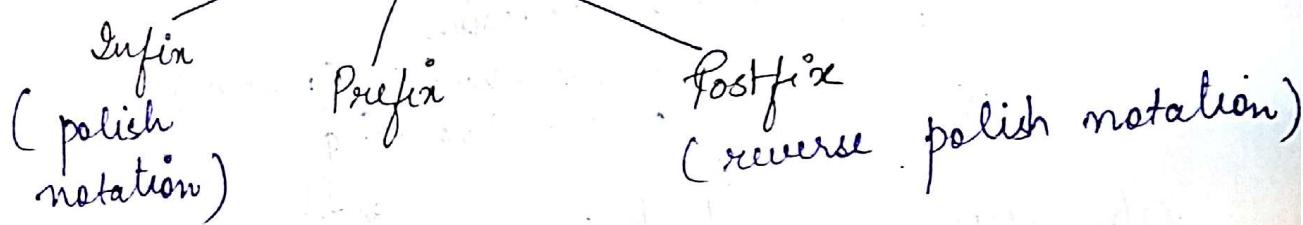
4. Set $\text{stack}[\text{Top}] = \text{Item}$

5. exit

② POP (Stack, Top, Item)

1. If $\text{Top} = -1$, print underflow
 Exit
2. Repeat steps 3 to 5.
3. Set $\text{Item} = \text{Stack}[\text{Top}]$
4. Set $\text{Top} = \text{Top} - 1$;
5. Print Item
6. Exit

Notations for an expression



① op^r where operator symbol is placed b/w its two operands , $A+B$ $C-D$ $E*F$

operator between operands

② operator is written before the operands.

$+AB$ $-CD$ $*EF$

③ operator is written after the operands.

$AB+$ $CD-$ $EF*$

Operator Precedence

$*, ^$ / (exponential)
 $+, -$

// Same priority are evaluated from left to right.

Conversions

① Infix to postfix

- > Paranthesize the expression from L to R.
- > operands with higher precedence operator are parenthesized first.
- > The sub expression which has been converted into postfix is to be treated as single operand.
- > after conversion, remove the parenthesis.

$$\begin{aligned}
 \text{eg. } (A+B) * C / D &= (A+B) * C / D \\
 &= ((AB+) * C) / D \\
 &= ((AB+)C *) / D \\
 &= (AB+)C * D / \\
 &= AB + C * D /
 \end{aligned}$$

sol^n

$$\begin{aligned}
 \text{eg. } A - B / (C * D ^ E) & \\
 = A - B / (C * (D E ^)) & \\
 = A - B / (C D E ^ *) & \\
 = A - B C D E ^ * / & \\
 = ABCDE ^ * / -
 \end{aligned}$$

Ques. Convert the expression into postfix using stacks

$$A^B^C - D + E / F / (G + H)$$

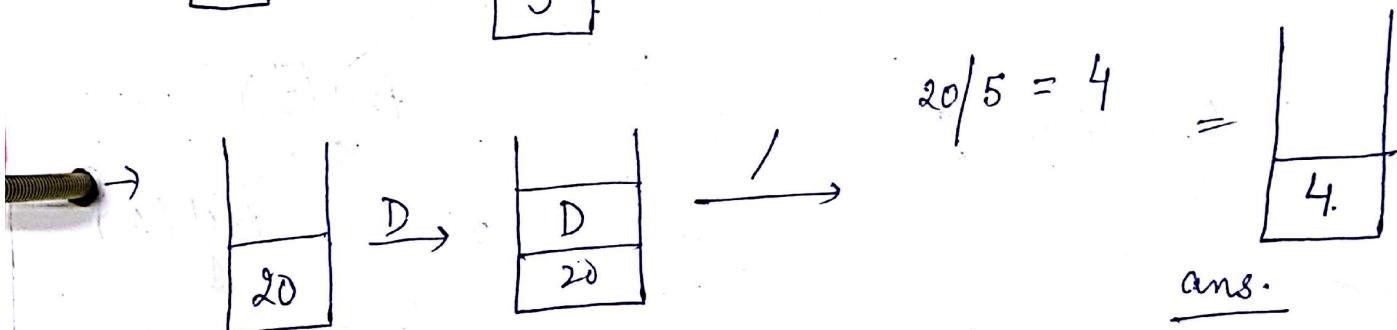
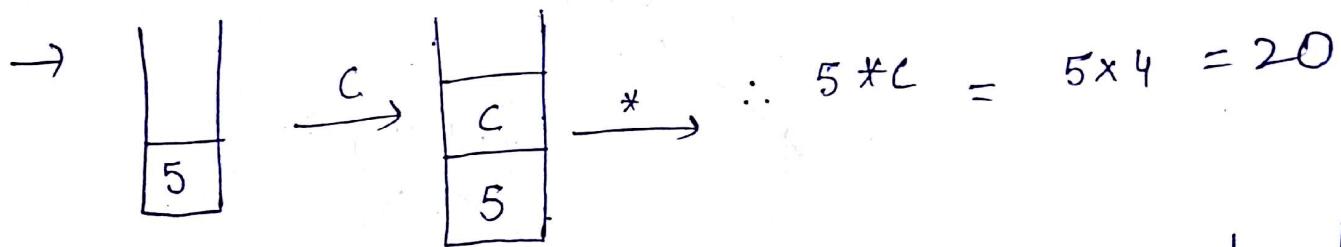
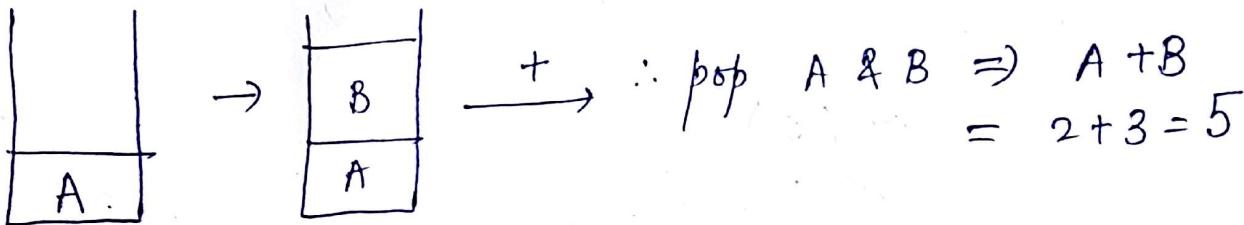
<u>Symbol</u>	<u>Stack</u>	<u>Expression</u>
1. A	(A
2. ^	(^	A
3. B	(^	AB
4. *	(*	AB^
5. C	(*	AB^C
6. -	(-	AB^C*
7. D	(-	AB^C*D
8. +	(+	AB^C*D -
9. E	(+	AB^C*D - E
10. /	(+ /	AB^C*D - EF
11. F	(+ /	AB^C*D - EF /
12. /	(+ /	AB^C*D - EF / G
13. ((+ / (AB^C*D - EF / G
14. G	(+ / ((AB^C*D - EF / GH
15. +	(+ / ((+	AB^C*D - EF / GH +
16. H	(+ / ((+)	AB^C*D - EF / GH + /
17.)	(+ / ((+)	
18.)	(+ / ((+))	

Infix to prefix

- reverse the given infix expression
- perform same conversion of postfix
eg. $(A + B * C) * D + E ^ 5$
 $= + * + A B C D ^ E 5$
- again reversal.

Evaluation of postfix expression:

Q. $AB + C * D /$ for $A = 2, B = 3, C = 4, D = 5$



Evaluation of prefix notation

- ① Reverse the string
- ② Calculate using stacks
- ③ The final res. will be in stack. (top).

Recursion

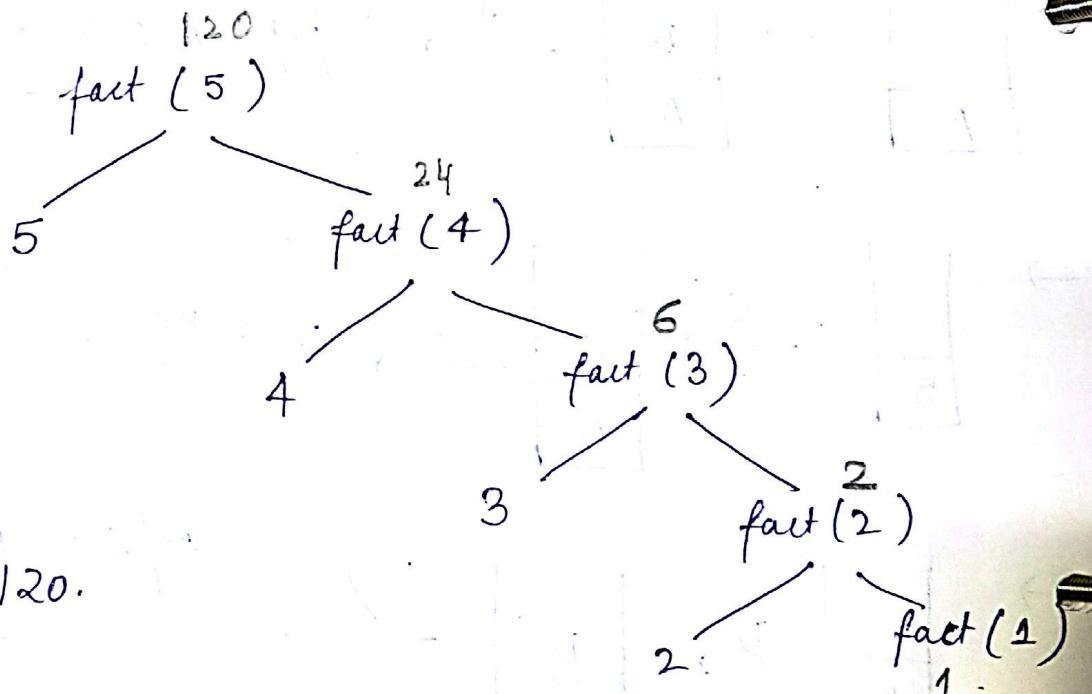
The process in which a funⁿ calls itself directly or indirectly is called recursion.

eg. int fact (int n)

```
{ if (n <= 1)  
    return 1;  
else
```

3

```
return n * fact (n - 1);
```



$$\therefore 5! = 120.$$

→ Stacks

Last In First Out (LIFO)

1) PUSH (STACK, ITEM) -

- 1) If $TOP = MAXSTK$, then write: OVERFLOW and Exit
- 2) Set $TOP := TOP + 1$
- 3) Set $STACK[TOP] := ITEM$
- 4) Exit

2) POP (STACK, ITEM) -

- 1) If $TOP = 0$, then write: UNDERFLOW and Exit
- 2) Set $ITEM := STACK[TOP]$
- 3) Set $TOP := TOP - 1$
- 4) Exit

PUSH (STACK, ITEM) - (Using Linked List):

- 1) If $AVAIL = \text{NULL}$, write: OVERFLOW and Exit
- 2) Set $NEW := AVAIL$ and $AVAIL := \text{LINK}[AVAIL]$
- 3) Set $\text{INFO}[NEW] := POP ITEM$
- 4) Set $\text{LINK}[NEW] := TOP$ and $TOP := NEW$.
- 5) Exit

POP (STACK, ITEM) - (Using Linked List):

- 1) If $TOP = \text{NULL}$, write: UNDERFLOW and Exit
- 2) Set $ITEM := \text{INFO}[TOP]$
- 3) Set $TOP := \text{LINK}[TOP]$ and $TEMP := TOP$
- 4) Set $\text{LINK}[TEMP] := AVAIL$ and $AVAIL := TEMP$
- 5) Exit

5) If an operator \oplus is encountered then

(a) add operator onto the stack

(b) If there is another operator on the top of stack and the new operator has less or equal precedence, then POP the stack and add it to R. Push the new operator on the top of stack.

6) If a right parenthesis is encountered, then

(a) POP from stack and add to R

(b) Remove the left parenthesis

7) Exit

eg. $(A \wedge B \times C - D + E) / F / (G + H))$

symbol	stack	R expression
A	(A
\wedge	(\wedge	A
B	(\wedge	AB
\times	(\wedge \times	ABA
C	(\wedge \times	ABAC
-	(\wedge \times -	ABAC-
D	(\wedge \times -	ABACD
+	(\wedge \times -	ABACD+
E	(\wedge \times -	ABACD-E
/	(\wedge \times - /	ABACD-E
F	(\wedge \times - /	ABACD-EF
/	(\wedge \times - /	ABACD-EF/
((\wedge \times - / (ABACD-EF/G
G	(\wedge \times - / (ABACD-EF/G
)	(\wedge \times - / ()	ABACD-EF/G



H

(+ / C +

$AB \wedge C \neq D - EF / GH$

)

(+ /

$AB \wedge C \neq D - EF / GH +$

)

$\underline{AB \wedge C \neq D - EF / GH + / +}$

$$\text{Ques) } A + (B * C - (D \mid E \wedge F) * G) * H$$

Symbol

A

Stack

(

R Expression

A

+

(+ /

A

(

(+ (/

A

B

(+ (/

AB

*

(+ (/ *

AB

C

(+ (/ *

ABC

-

(+ (/ -

ABC *

(

(+ (/ - (

ABC *

D

(+ (/ - (

ABC * D

I

(+ (/ - (/

ABC * D

E

(+ (/ - (/

ABC * DE

^

(+ (/ - (/ ^

ABC * DE

F

(+ (/ - (/ ^

ABC * DEF

)

(+ (/ - (/ ^

ABC * DEF ^ /

*

(+ (/ - (/ ^ *

ABC * DEF ^ /

G

(+ (/ - (/ ^ *

ABC * DEF ^ / G

)

(+ (/ - (/ ^

ABC * DEF ^ / G * -

*

(+ (/ - (/ ^ *

ABC * DEF ^ / G * -

H

(+ (/ - (/ ^ *

ABC * DEF ^ / G * - H

)

$\underline{\underline{ABC * DEF ^ / G * - H * +}}$

Arithmetic Expressions -

Highest - ↑

*, /

+, -

e.g. 5, 6, 2, +, *, 12, 4, 1, -

eg.	Symbol	Stack
	12	12

symbol	stack
5	5
6	5, 6
2	5, 6, 2
+	5, 8
*	40
12	40, 12
4	40, 12, 4
/	40, 3
-	37

eg.	Symbol	Stack
	12	12
	7	12, 7
	3	12, 7, 3
	-	12, 4
	1	3
	2	3, 2
	1	3, 2, 1
	5	3, 2, 1, 5
	+	3, 2, 6
	*	3, 12
	+	15

A + B * C	(A * B) * C	(A + B ↑ D) / (E - F) + G
Post	ABC * +	AB + C * X
Pre	+ A * BC	* + AB * C

ABD ↑ + EF - 1 G ↑ +

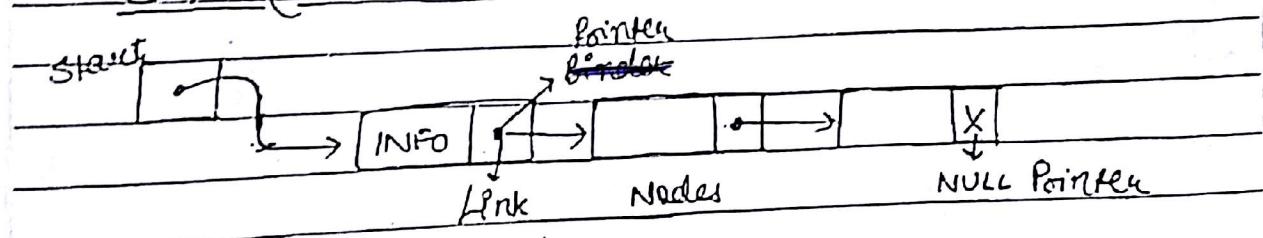
Conversion of Infix to Postfix -

- 1) Push "Push onto stack and add" to the end of S.
- 2) Scan S from left to right and repeat steps 3 to 6 for each element of S until the stack is empty.
- 3) If an operand is encountered, add it to R.
- 4) If a left parenthesis is encountered, push it into stack.

Linked-List

- It is a dynamic data structure which can grow & shrink during the execution of program.
- It has efficient memory utilization.
- It is a linear DS acc. to strategy of storage.
- " " " non - " " " "

→ LINKED LIST :-



	Info	Link		
1	A	3	Start = 1	Info[1] = A
2	E	7	Link[1] = 3	Info[3] = B
3	B	6	Link[3] = 6	Info[6] = C
4			Link[6] = 5	Info[5] = D
5	D	2	Link[5] = 2	Info[2] = E
6	C	5	Link[2] = 7	Info[7] = F
7	F	0	Link[7] = 0	

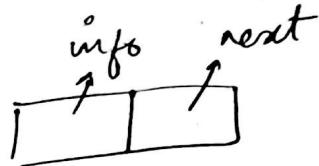
Structure

```
struct node
{
    int data;
    struct node *next; // self referential ptr
};
```

Single Link List

```
# include <stdio.h>
# " " <stdlib.h>
# " " <conio.h>
struct node
{
    int info;
    node *next;
};

main()
{
    struct node *s;
    s = (struct node *) malloc ( sizeof ( struct node ) );
    // dynamic mem. allocation
    s->info = 20;
    s->next = NULL;
    printf ("%d", s->info);
}
```



Start

PTR

Link(PTR)

Start

Link[Start]

PTR := Link(PTR)

1) TRaversing -

- 1) Set PTR := START
- 2) Repeat steps 3 and 4 while PTR ≠ NULL
- 3) Apply process to Info(PTR) or Write : INFO [PTR]
- 4) Set PTR := Link [PTR] [to go to the next node]
[End of step 2 loop]
- 5) Exit

2) SEARCHING -

- 1) Set PTR := START
- 2) Repeat step 3 while PTR ≠ NULL
- 3) If ITEM = INFO [PTR] then
 - 1) Set LOC := PTR and Exit
 - else Set PTR := Link [PTR]
[End of if]
[End of step 2 loop]
- 4) Set LOC := NULL [Search is unsuccessful]
- 5) Exit

Avail List - Similar to linked list, stores empty values as info

Avail



Avail

Link(Avail)

- (*) It is a list of free nodes in memory. It contains unused memory cells: Also known as 'List of Available space', or 'Free Storage List' or 'Free Pool'.

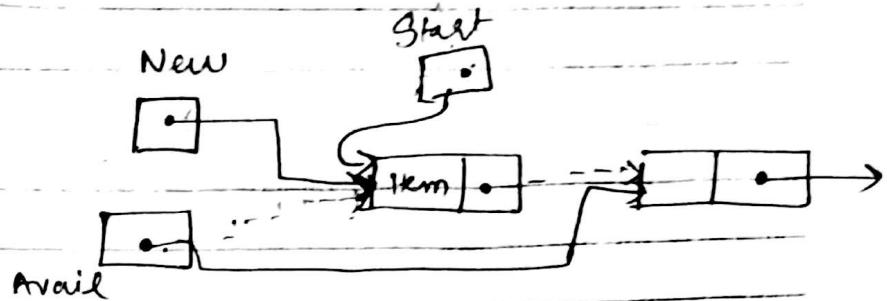
Overflow - AVAIL = NULL.

i.e. we want to add additional data but there is no data.
space free / blank node

Underflow - START = NULL

3) INSERTION-

A) At the beginning-



INFIRST (INFO, LINK, START, AVAIL, ITEM)

- 1) If AVAIL = NULL, write: OVERFLOW and Exit
- 2) Set NEW := AVAIL and AVAIL := LINK [AVAIL]
- 3) Set INFO [NEW] := ITEM
- 4) Set LINK [NEW] := START . // new node now points to original first node
- 5) Set START := NEW
- 6) Exit

B) INS At any specified location-

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

- 1) If AVAIL = NULL; write: OVERFLOW and Exit

- 2) Set NEW := AVAIL and AVAIL := LINK [AVAIL]

- 3) Set INFO [NEW] := ITEM

- 4) If LOC = NULL then:

Set LINK [NEW] := START and START := NEW

else

Set LINK [NEW] := LINK [LOC] and LINK [LOC] := NEW

End of if

- 5) Exit

Types of LL

- 1) Linear / singular LL
- 2) Circular LL
- 3) Doubly "
- 4) circular doubly LL

Deletion

DEL (INFO, LINK, START, AVAIL, LOC, LOCP)

// this algo deletes the node N with location LOC.
// LOCP is the locⁿ of node that precedes N.

Step 1: If LOCP = NULL then

 Set START := LINK [START]

else:

 Set LINK [LOCP] := LINK [LOC]

[end if]

Step 2: [return del. node to avail list]

Set LINK [LOC] := AVAIL and

AVAIL = LOC.

Step 3: Exit

Header Linked List

- LL that stores & manages 'header node'.

grounded header
(last node = NULL)

circular header
(last node → header node)

* Traversing a circular header list

1. Set PTR := LINK [START]
2. Repeat 3 & 4 while PTR ≠ START
3. Write: INFO [PTR]
4. Set PTR := LINK [PTR] // next node
5. Exit

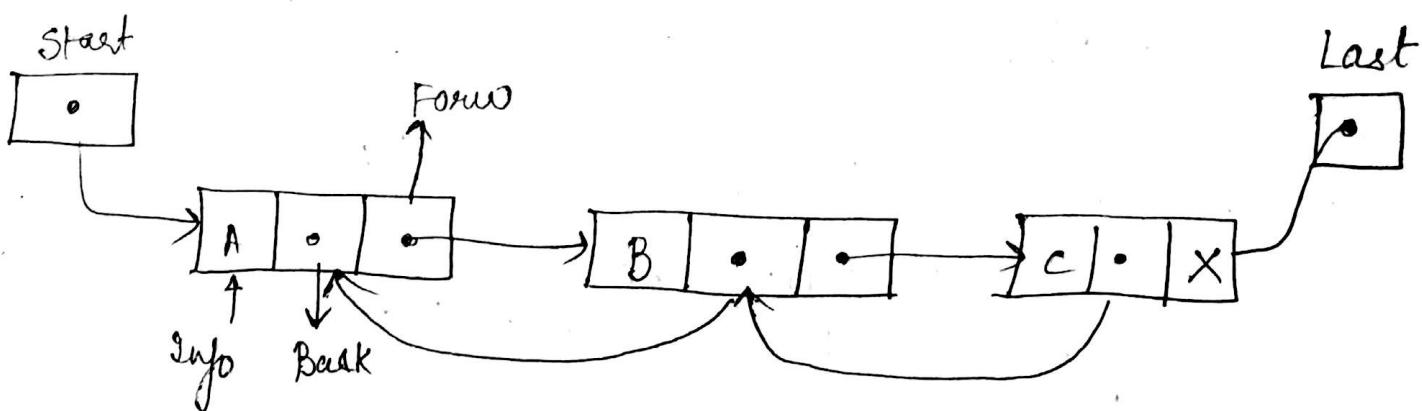
② Searching

SRCH (INFO, LINK, START, ITEM, LOC)

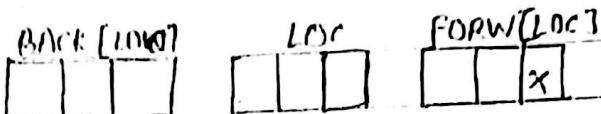
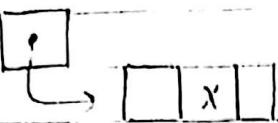
1. Set PTR := LINK [START]
2. Repeat while INFO [PTR] ≠ ITEM & PTR ≠ START:
 Set PTR := LINK [PTR]
 [end loop]
3. If INFO [PTR] = Item, then:
 Set LOC := PTR
 Else
 Set LOC := NULL
 [end if]
4. Exit

2-way Lists (Doubly Linked List)

- It is a linear collection of data elements, called nodes, where each node N is divided into 3 parts:
 - o) INFO : stores information
 - o) FORWARD : contains "loc" of next node
 - o) BACK : " " " preceding node



1) Deletion // delete location LOC



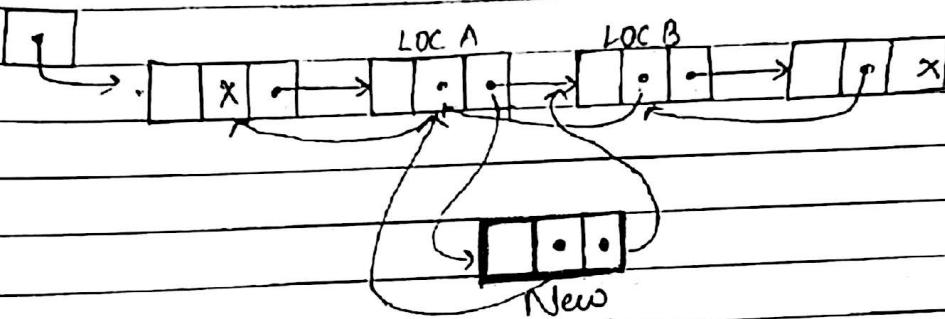
Step 1: $\text{FORW}(\text{BACK[LOC]}) := \text{FORW[LOC]}$

dict 2: BACK [FORW[LOC]] := BACK[LOC]

Step 3: $\text{FORW}[\text{LOC}] := \text{AVAIL}$ and $\text{AVAIL} := \text{LOC}$

Step 1: exit

2) INSERTION-



Step 1 If $AVAIL = \text{NULL}$, write: OVERFLOW and Exit

Step 1 If $AVAIL = \text{NULL}$, then
Step 2 Set $NEW := AVAIL$, $AVAIL := FORW[AVAIL]$
 $INFO[NEW] := ITEM$

and `INFO[NEW]:=ITEM`

Step 3 set FORW [LOC A] := NEW] setting next pointer
↳ [NEW] = LDCB

and FORM [NEW] = LUCB
[Form 17 = NEW]

~~11.1.8.1t BACK [OCB] = NEW~~

Step 4 Set BACK [LOCB] = NEW
and FBACK [NEW] = LOCA

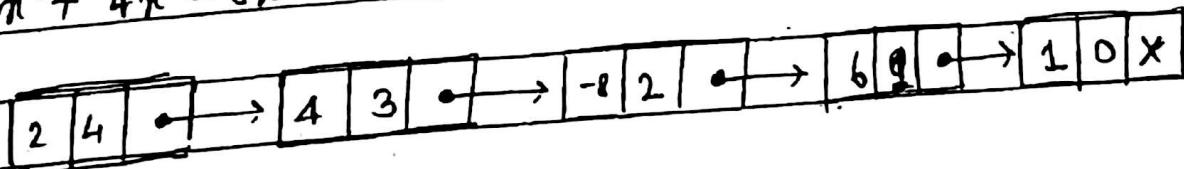
Step 4 Set BACK [LOCB] = NEW
and FB_{ACK} [NEW] : LOCA

step 5 exit

step 5 exit

step 5 exit

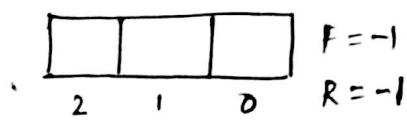
$$\text{eq. } 2x^4 + 4x^3 - 8x^2 + 6x + 1$$



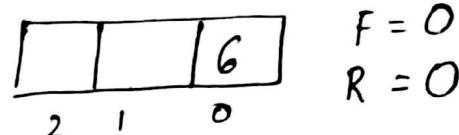
Queue

- It is a linear data structure and works on FIFO(First in First out).
- In queue ADT, insertion is performed at one end (rear).
deletion at other end (front)

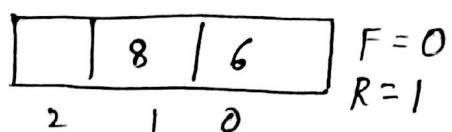
e.g. 6, 8, 10



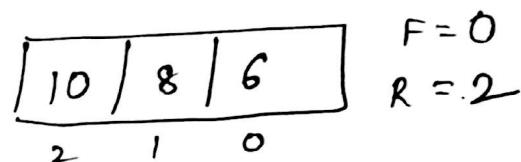
a)



b)



c)



Operations :

- ① Insertion in queue (enqueue)

QINSERT (Queue , N, Front, Rear, Item)

1. // Queue is filled?
If Front = 0 and Rear = N, or if Front = Rear + 1,
then write: Overflow & Return

2. If Front : = NULL, then // Queue empty
Set Front : = 0 & Rear : = 0

3. else if Rear = N, then
Set Rear : = 0

else
Set Rear : = Rear + 1

4. Set Queue [Rear] : = Item

5. Return.

② Deletion (or dequeue)

QDELETE (Queue , N, Front, Rear, Item)

1. If $\text{Front} := \text{NULL}$, then Underflow & Return.
2. Set $\text{Item} := \text{QUEUE} [\text{Front}]$
3. If $\text{Front} = \text{Rear}$, then // queue has only 1 ele
Set $\text{Front} := \text{NULL}$ & $\text{Rear} := \text{NULL}$
else if $\text{Front} = N$, then // last element
Set $\text{Front} := 0$
else
Set $\text{Front} := \text{Front} + 1$;
4. Return .

3) INSERTION AS A LINKED LIST-

LINK @INSERT·(INFO, LINK, FRONT, REAR, ITEM, AVAIL)-

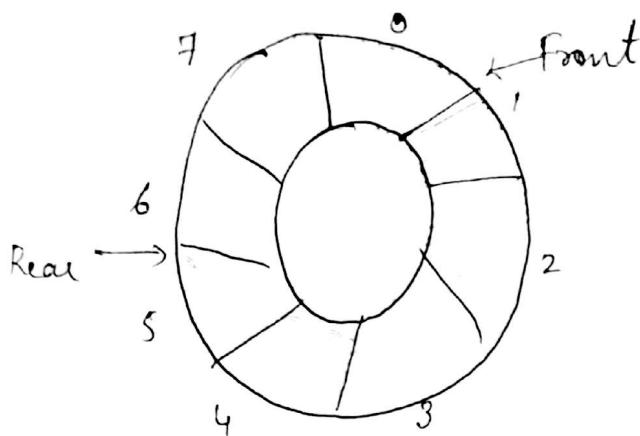
- 1) If AVAIL =NULL, write: OVERFLOW and Exit
- 2) Set NEW:=AVAIL and AVAIL:=LINK [AVAIL]
- 3) Set INFO[NEW]:=ITEM and LINK[NEW]:=NULL
- 4) Set if FRONT=NULL, Set FRONT:=REAR:=NEW
else
 LINK[REAR]:=NEW and REAR:=NEW
- 5) Exit

4) DELETION AS A LINKED LIST-

- 1) If FRONT =NULL, write: ^{UNDER} UNDERFLOW and Exit
- 2) Set ITEM:=INFO[FRONT]
- 3) Set FRONT:=LINK[~~FRONT~~] ^{TEMP} and TEMP:=FRONT
- 4) LINK[TEMP]:=AVAIL and AVAIL:=TEMP
- 5) Exit

Circular Queue

→ pointer rear points to the beginning of the queue when it reaches at the end of the queue



$$\text{front} = 0, \text{rear} = 0$$

$$\text{Insert: } Q[\text{rear}] = x; \\ \text{rear} = (\text{rear} + 1) \bmod n$$

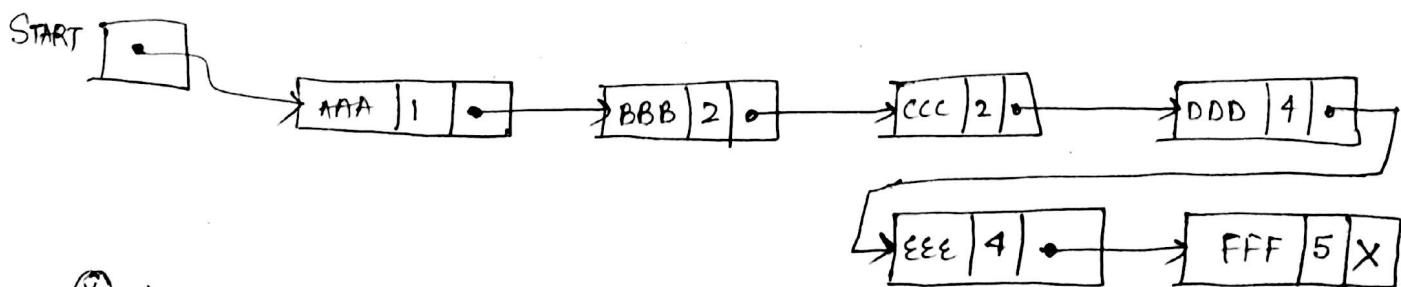
$$\text{Delete: } x = Q[\text{front}]; \\ \text{front} = (\text{front} + 1) \bmod n.$$

Full queue is indicated by: $\rightarrow \text{front} = (\text{rear} + 1) \bmod n;$

Priority Queue

- It is a queue in which each element is inserted / deleted on the basis of their priority.
- higher priority element will be added first over lower priority element.
- If priority of 2 elements is same, then they are added to the queue on FCFS basis.

```
struct pque
{
    int data;
    int pri;
    struct pque *next;
};
```



④ priority of 1 > priority of 2

Degue

(double ended queue)

- insertion & deletion can be performed at both the ends.
- Types
 - ↑↑ restricted queue
 - ↓↓ restricted queue

① deletion can be performed at both the ends, but insertion at only one end, using rear only.

② insertion can be performed at both ends but deletion can be done only at one end, using front only.

