Introduction to DS

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

Random data
① difficult in searching
eg. words in random order
② college list students

→ **Structured data**
eg. in dictionary (arrange words)
⇒ searching / insertion is easier.
→ dept wise / class wise etc.

operation
- traversing
- searching
- sorting
- insertion
- deletion

(all becomes easy for a structured data).

Data structure is used by →
-) OS
-) compiler
-) DBMS
-) Data communication etc. (related to data).

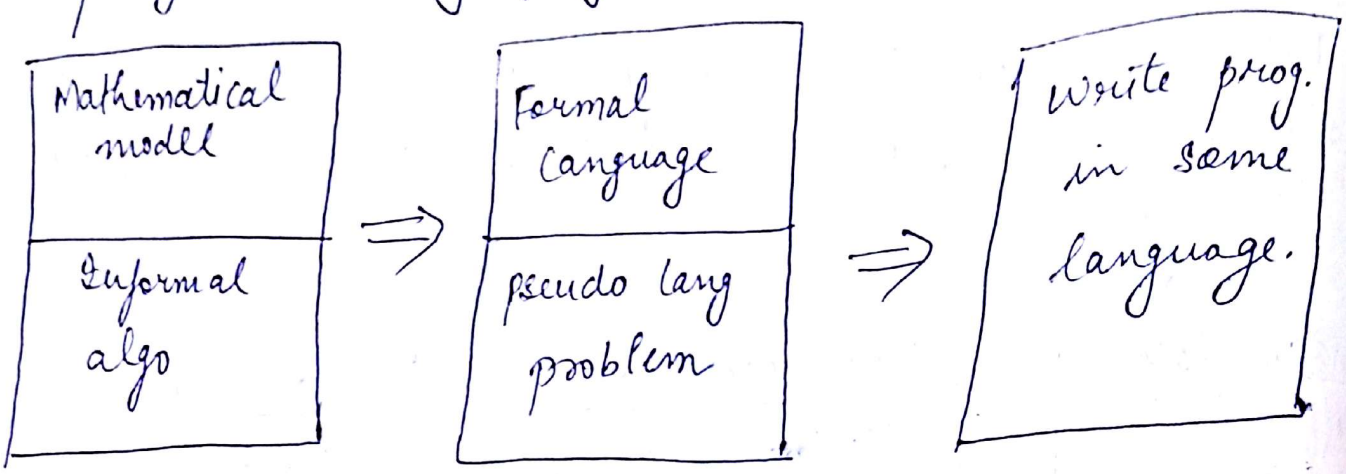
Programming methodologies & design of algo

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 instr, to solve well defined computational problem.
* **DS**: way of organizing data.

DS = organized data + opⁿ

Steps for creating prog:-



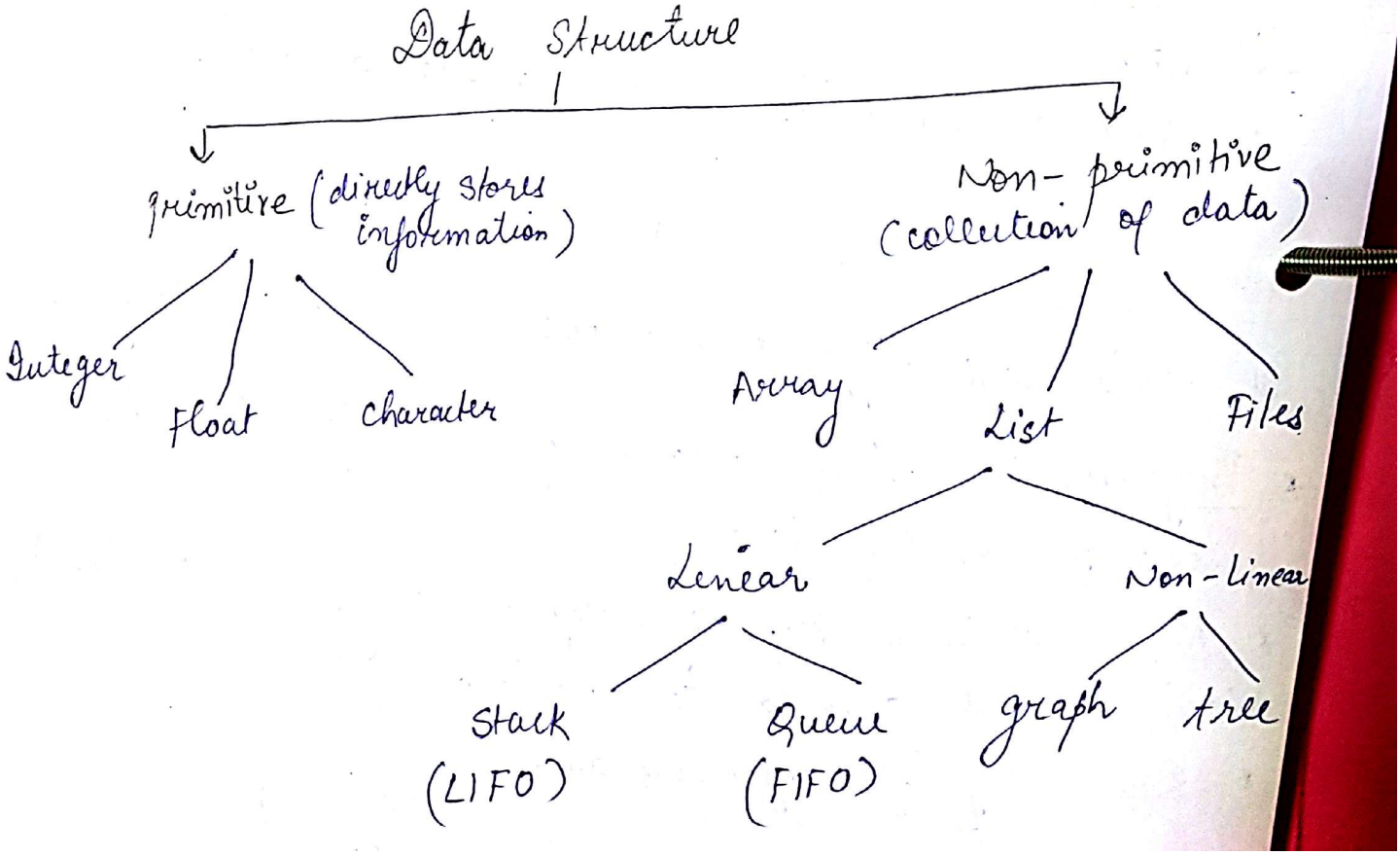
Ques. Identify those students who passed the exam (50% & above).

Soln.

m = 60
if (m ≥ 50)
"Student pass")

Step1: Enter marks
Step 2: if marks ≥ 50
print "pass"
Step 3: Repeat 1 & 2
Step 4: End

⇒ write in C lang



bstract Data type: > we are not concerned about the internal interface.

we directly use them as per our need.

Stack / Queues are ADT

eg. driving a car

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

Array: It is a collection of similar data type.
(int / char / float)

eg. int x_1, x_2, x_3 ;

but in array => int arrayname [size]

eg. int a[5]; // declaration of array

int a[5] = { 4, 6, 8, 5, 3 } // initialization of array
(0 ↓ 4 ↓)

// indexing in C

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

eg: LB = −2, UB = 3, find no. of elements

N = 3 − (−2) + 1 = 6

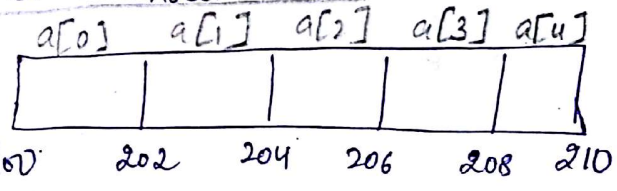
1-D Array

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

address calcul^n :

int A[5]

Start index = 0



(base address)

A[i] = Base address + i (data type size)

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

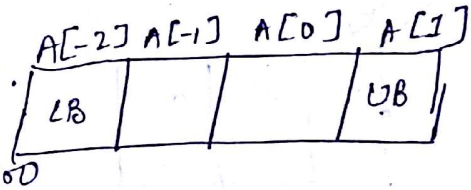
int A[LB] --- A[UB]

Start index = LB.

now,

A[K] = Base add. A[LB] + (K − LB) × size

eg.



A[0] = 200 + (0 − (−2)) × 2
= 200 + 4
= 204

Q. A[-4:3]

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

For 2D arrays,

eg. A[-2:2, 2:22]

Soln, Find L_1 and L_2

Total elements = $L_1 \times L_2$

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

$L_2 = 22 - 2 + 1 = 21$

$N = L_1 \times L_2 = 5 \times 21 = 105$.

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

$L_1 = 8 - 2 + 1 = 7$

$L_2 = 1 - (-4) + 1 = 6$

$L_3 = 10 - 6 + 1 = 5$

$\therefore N = 7 \times 6 \times 5$

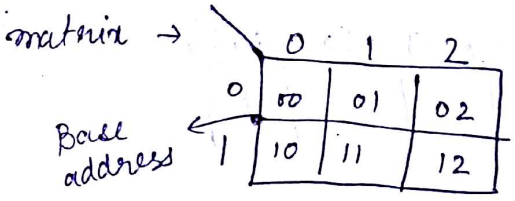
$= 210$

Array Representation

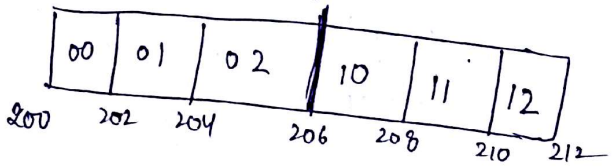
2-D array representation, also known as "matrix"

① Row-Major → row → col

eg. int A[2][3] // elements = $2 \times 3 = 6$



For row major, first rows are traversed.



eg. A[m][n]

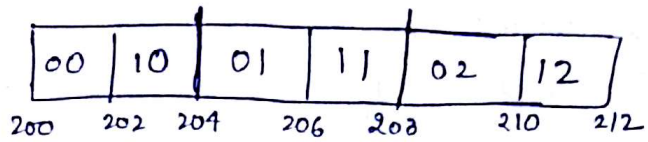
address :

$A[i][j] = Base + (i \times n + j) \times size$

eg. $A[1][0] = 200 + (1 \times 3 + 0)2$

$= 200 + 6 = 206$

② Column major ④



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

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

$= 200 + 4 = 204.$

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).

Representation < Three column form

Linked list

① 3 columns = rows, col, values

eg.

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

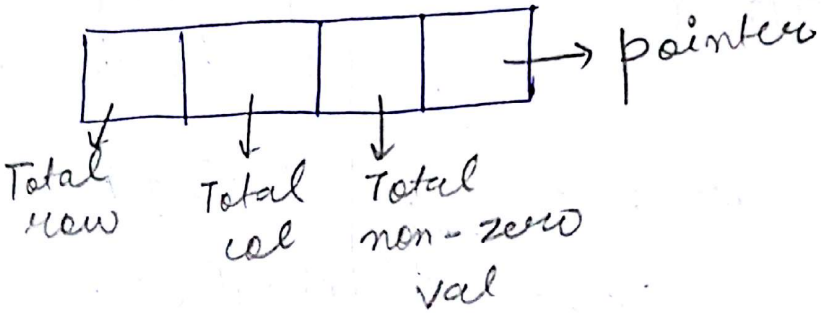
4×4

1st row: Total r, c & values →

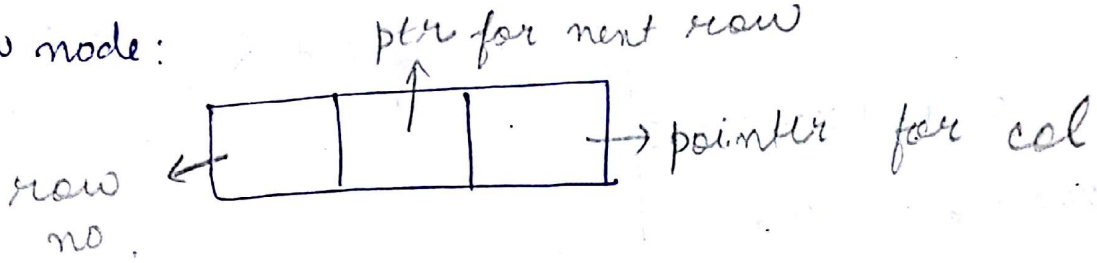
Rows	col	value
4	4	6
0	1	2
1	0	4
1	2	3
2	1	1
2	3	1
3	1	2

(2) In Link List representation, we require three structure.

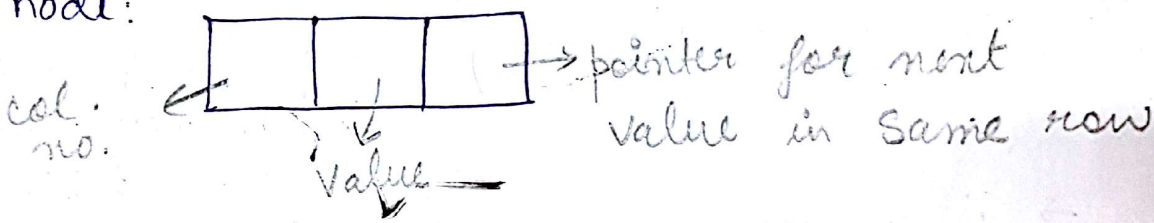
a) Head node :



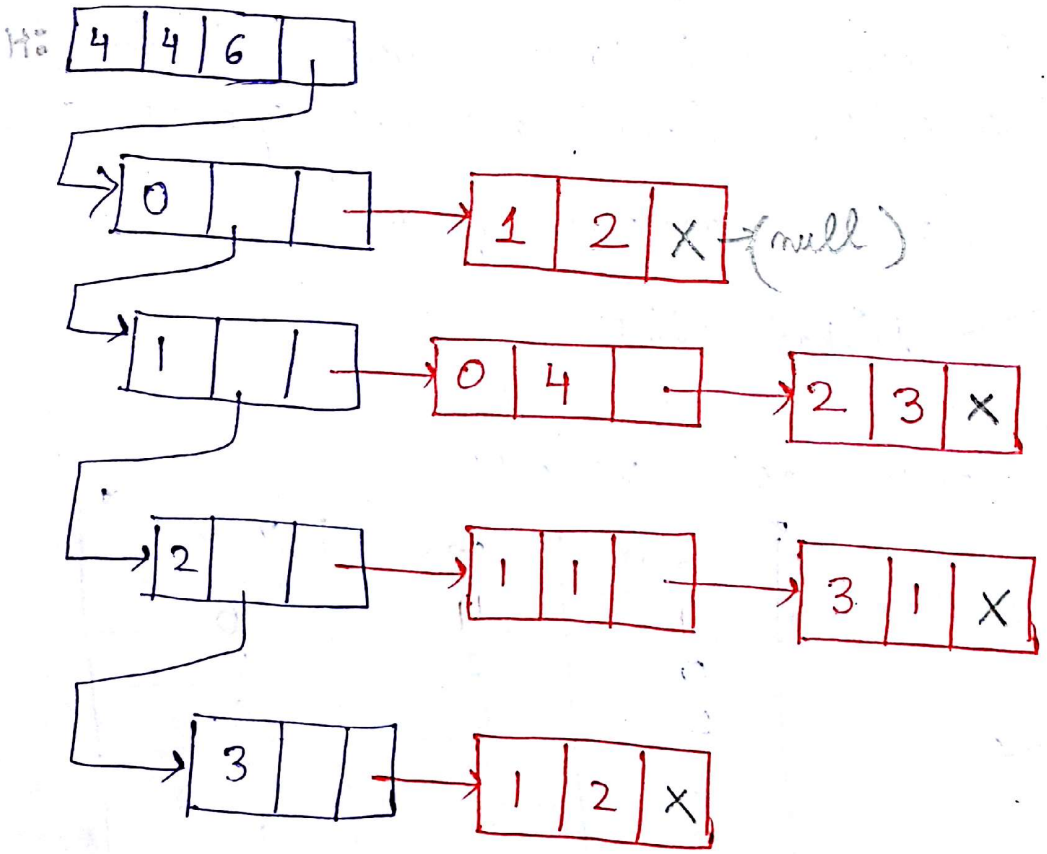
b) Row node :



c) column node :

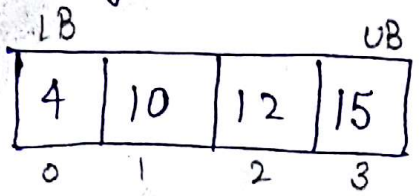


eg: same →



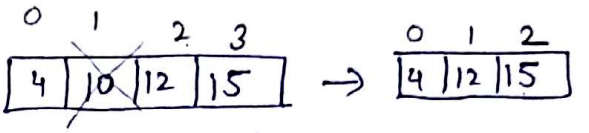
ray operations:

Traversing



Step 1 : Repeat for i = LB to UB
2: Process A[i]
[End of loop]
3: Exit

▷ Deleting

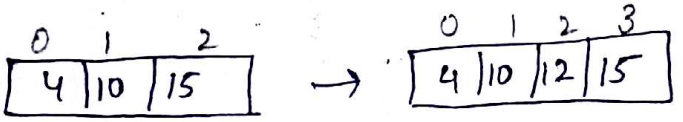


Step 1 : Set Item := A[K] // to be deleted
2: Repeat for i = K to N-1
Set A[i] := A[i+1]
[end loop]
3: Set N := N-1
4: Exit

3 Insertion

•) Insertion in unsorted array

Let K be the loc^n where Item is to be inserted. K = 2



Step1: Set i = N i = 2
2: Repeat while i ≥ K
3: Set A[i+1] := A[i] // moving the element → Right
4: Set i := i − 1
[end loop]
5: Set A[K] := item.
6: N := N+1 ;
7: Exit

Date

Array-

Linear Array- Int A[K]

Array Length = UB - LB + 1
(UB: Upper bond, LB: Lower bond)

LOC(A[K]) = Base address + w(K-LB)

(LOC(A[K]): Location of Kth element of array A whose address is to be calculated)

Ques) Base = 200, LB = 1932, Sale[K] = 1965, w = 4 words/memory unit

LOC(Sale[1965]) = 200 + 4{1965 - 1932}
= 200 + (4 x 33) = 332

→ Operations on Data structures -

1) TRAVERSING-

Step 1 Set K := LB [Initialize counter]
Step 2 Repeat steps 3 & 4 for K ≤ 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-

INSERT (LA, N, K, ITEM)-
(N: Length of array, K: location)

Step 1 Set J := N [Initialize counter]
Step 2 Repeat steps 3 & 4 while J ≥ K

Page ____ Subhash
Date ____

Step 3 Set LA[J+1] := LA[J] # This steps 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) → position to be deleted
Step 1 Set ITEM := LA[K] → counter # Element to be deleted is stored as ITEM only to save it at an alternate location. This allows us to print the value of deleted (at the end of program) if required
Step 2 Repeat for J = K to N-1
Set LA[J] := LA[J+1]
[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] ≠ ITEM
Set LOC := LOC+1
[End of loop]
Step 4 If LOC = N+1, then set LOC := 0 [Unsuccessful?]
Step 5 Exit

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

4.B) BINARY SEARCH-

Location { BEG := LB
END = UB
MID = INT $\left(\frac{BEG+END}{2}\right)$

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

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

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

Step 3 If ITEM < DATA[MID] then:
Set END := MID - 1
else:
Set BEG := MID + 1
[End of If structure]

Step 4 Set MID := INT $\left(\left(\frac{BEG+END}{2}\right)\right)$
[End of step 2 loop]

Step 5 If DATA[MID] = ITEM then:
Set LOC := MID
else
Set LOC := 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. numbering of steps

Step 1 ***
Step 2 go to step 7 → control
Step 3 ***
⋮
Step n Exit / End / Stop

when loops are to be used

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

4) Assignment statement - To assign values to variables
# a := 5 ⇒ assigns value 5 to a
a = b ⇒ a equals b

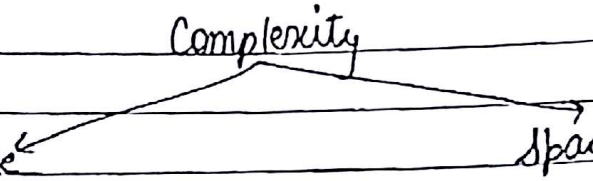
5) Variable Names - Always written in CAPITAL LETTERS

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 required for an algo to be executed

No. of steps / memory space required

→ 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 fn 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 algos such as performance and memory required.

In this notation, we eliminate the constant factors.

Page: Subhash
Date:

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

eg. $f(n) = 2n^2 + 3n$
$O(n) = n^2$

eg. $f(n) = 3n^3 + n^2 + 2n$
$O(n) = n^3$

eg. $f(n) = 3n^3 + 1n^2 + \log n$
$O(n) = \log n$

eg. $f(n) = 4n^3 + \log n + 3e^n$
$O(n) = e^n$

eg. $f(n) = 2$
$O(n) = 1$

Three cases of algo complexity -

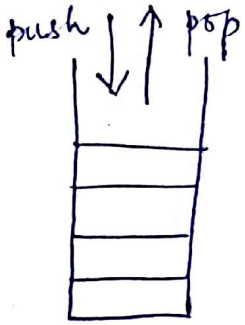
1) Worst case: Max. value of $f(n)$ for every input
ie. algo runs correctly only on a few input sets

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

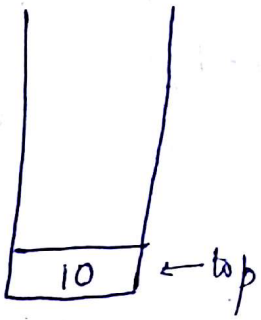
3) Best case: Min. value of $f(n)$

Stacks

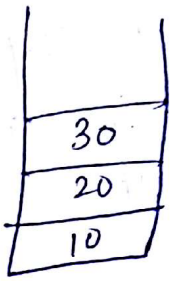
Stack is a linear data structure in which op^{n} are performed based on LIFO.



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^{n}.

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

1. If Top = MaxSTK, print overflow
   write: OVERFLOW
   Exit
2. Read Item
3. Set Top = Top + 1
4. Set stack[Top] = Item
5. Exit

(2) POP (Stack, Top, Item)

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

Notations for an expression

- Infix (polish notation)
- Prefix
- Postfix (reverse polish notation)

(1) opr where operator symbol is placed b/w its two operands, A+B C-D E*F

operator operand

(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 paranthesized first.
- The sub expression which has been converted into postfix is to be treated as single operand.
- after conversion, remove the paranthesis.

eg. $(A+B) * C/D$

$= (A+B) * C/D$

$= ((AB+) * C)/D$

$= ((AB+)C*)/D$

$= (AB+)C*D/$

$= AB+C*D/.$

sol^n

eg. $A-B/(C*D\wedge E)$

$= A-B/(C*(DE\wedge))$

$= A-B/(CDE\wedge*)$

$= A-BCDE\wedge*/$

$= ABCDE\wedge*/-$

Ques. Convert the expression into postfix using stacks

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

	Symbol	Stack	Expression
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-E
11.	F	(+/	AB^C*D-EF
12.	/	(+/	AB^C*D-EF/
13.	(	(+/(	AB^C*D-EF/
14.	G	(+/(	AB^C*D-EF/G
15.	+	(+/(+	AB^C*D-EF/G
16.	H	(+/(+	AB^C*D-EF/GH
17.	)	(+/	AB^C*D-EF/GH+
18.	)		AB^C*D-EF/GH+/+

Infix to Prefix

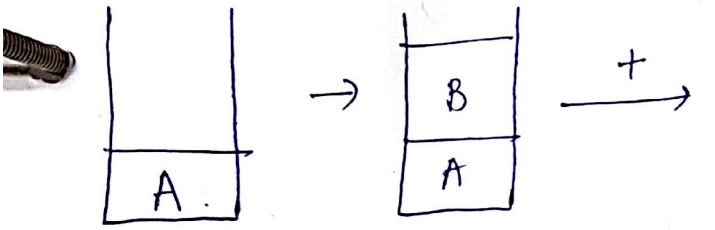
- reverse the given infix expression
- perform same conversion of postfix
- again reverse.

eg. $(A+B\wedge C)*D+E\wedge 5$

$= +*+A\wedge BCD\wedge E5$

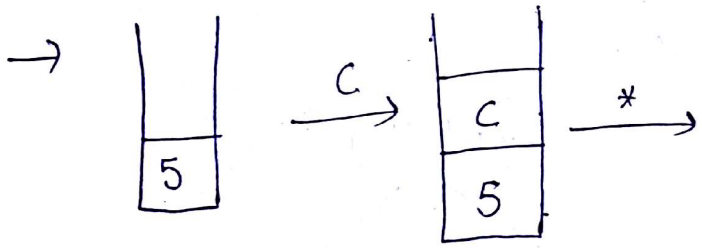
Evaluation of postfix expression:

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

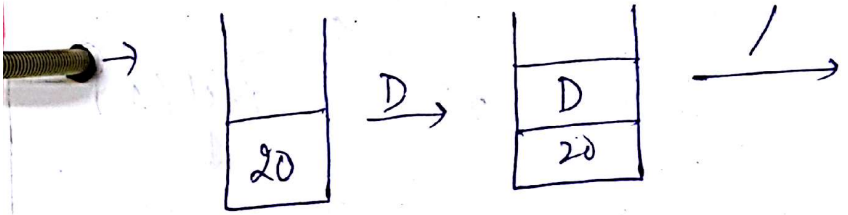


$\therefore$ pop A & B $\Rightarrow$ A+B

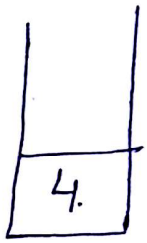
$= 2+3=5$



$\therefore 5*C = 5\times 4 = 20$



$20/5 = 4$ =



ans.

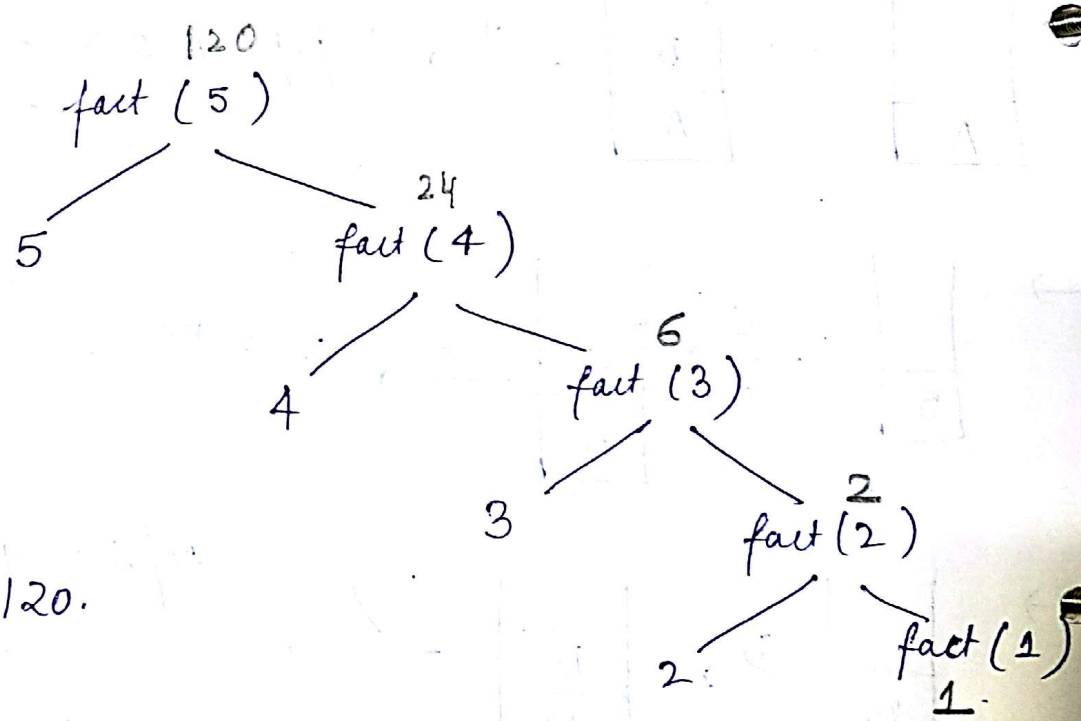
Evaluation of prefix notⁿ

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

Recursion

The process in which a fun^n calls itself directly or indirectly is called recursion.

eg.
```
int fact (int n)
{ if (n <= 1)
      return 1;
  else
      return n * fact(n-1);
}
```



∴ 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 = NULL, write: OVERFLOW and Exit
2) Set NEW := AVAIL and AVAIL := LINK[AVAIL]
3) Set INFO[NEW] := ITEM
4) Set LINK[NEW] := TOP and TOP := NEW
5) Exit

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

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

5) If an operator ⊕ 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^B*C-D+E/F/(G+H))

Symbol	Stack	R expression
A	(	A
^	(^	A
B	(^	AB
*	(*	AB^
C	(*	AB^C
-	(-	AB^C*
D	(-	AB^C*D
+	(+	AB^C*D-
E	(+	AB^C*D-E
/	(+/	AB^C*D-E
F	(+/	AB^C*D-EF
/	(+/	AB^C*D-EF/
(	(+/(	AB^C*D-EF/
G	(+/(	AB^C*D-EF/G
+	(+/(+	AB^C*D-EF/G

H	(+/(+	AB^C*D-EF/GH*
)	(+/	AB^C*D-EF/GH+
)		AB^C*D-EF/GH+/+

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

Symbol	Stack	R Expression
A	(	A
+	(+	A
(	(+(	A
B	(+(	AB
*	(+(*	AB
C	(+(*	ABC
-	(+(-	ABC*
(	(+(-(	ABC*
D	(+(-(	ABC*D
/	(+(-(/	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
)		ABC*DEF^/G*-H*+

Date

Arithmetic Expressions -

Highest - ↑

*, /

+, -

eg. 5, 6, 2, +, *, 12, 4, /, -

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
/	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*	ABD↑+EF-/G+
Pre	+A*BC	*+ABC	

V. Imp.

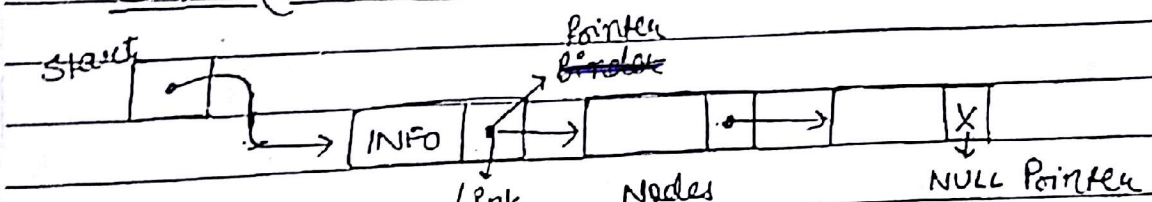
Conversion of Infix to Postfix -

1) 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 elemen 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.
- " " " non-" " " " storage.

→ LINKED LIST :-



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

Start = 1
Link[1] = 3
Link[3] = 6
Link[6] = 5
Link[5] = 2
Link[2] = 7
Link[7] = 0

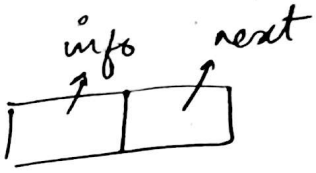
Info[1] = A
Info[3] = B
Info[6] = C
Info[5] = D
Info[2] = E
Info[7] = F

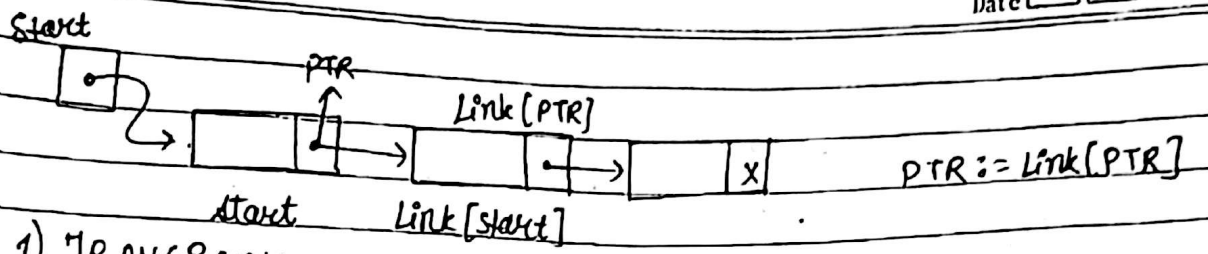
Structure

```
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 no
                    // dynamic mem. allocation
  s -> info = 20;
  s -> next = NULL;
  printf (" %d ", s -> info );
}
```





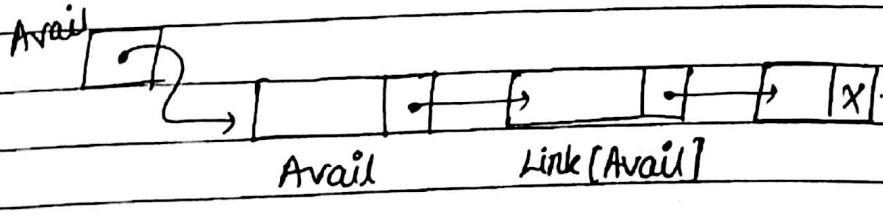
1) TRAVERSING -

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 steps 3 while PTR ≠ NULL
3) If ITEM = INFO [PTR] then
   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

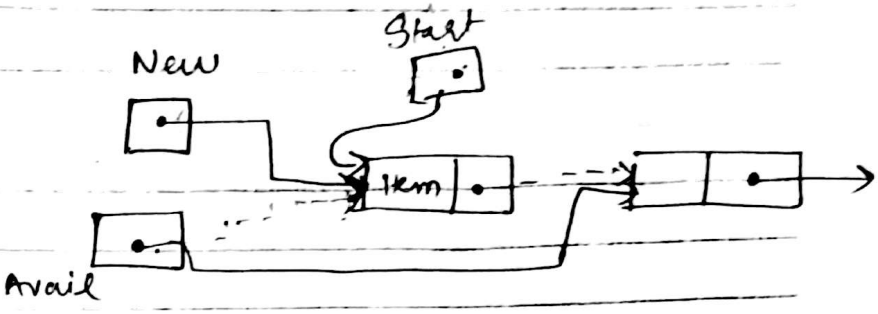
Avail List - Similar to linked list, stores empty values as info



* 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
ie. 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) 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: } First posⁿ
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 loc^n 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)

★ (1) 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

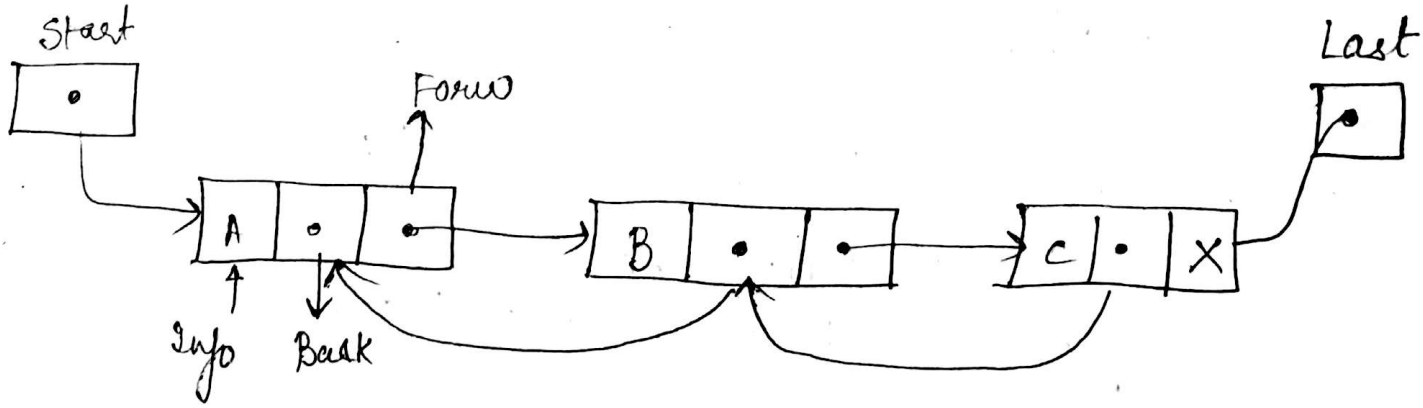
(2) Searching

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

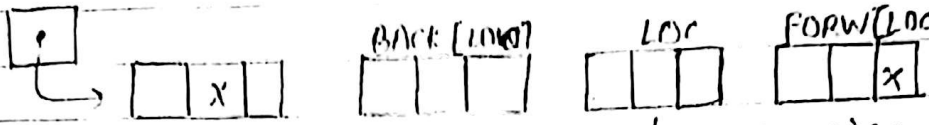
1. Set PTR := LINK[START]
2. Repeat while INFO[PTR] ≠ ITEM & PTR ≠ START:
   Set PTR := LINK [PTR]
   [end loop]
3. If INFO [PTR] = Item, then:
   Set LOC := PTR
   Else
   Set LOC := NULL
   [end if]
4. Exit

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:
  - a) INFO : stores information
  - b) FORW : contains loc^n of next node
  - c) BACK : " " " preceding node

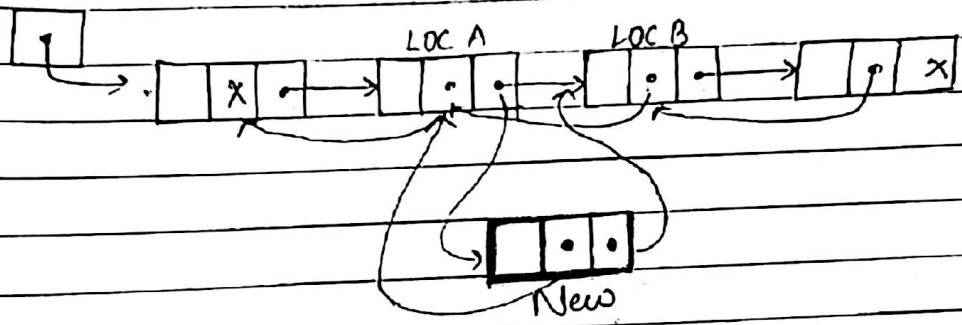


1) DELETION- // delete location LOC



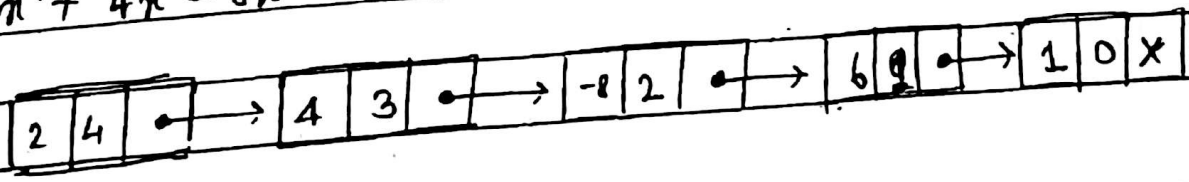
Step 1: FORW (BACK [LOC]) := FORW [LOC]
Step 2: BACK (FORW [LOC]) := BACK [LOC]
Step 3: FORW [LOC] := AVAIL and AVAIL := LOC
Step 4: Exit

2) INSERTION-



Step 1 If AVAIL = NULL, write : OVERFLOW and Exit
Step 2 Set NEW := AVAIL, AVAIL := FORW [AVAIL]
and INFO [NEW] := ITEM
Step 3 Set FORW [LOC A] := NEW
and FORW [NEW] = LOC B
] setting next pointer
Step 4 Set BACK [LOC B] = NEW
and BACK [NEW] : LOC A
Step 5 Exit

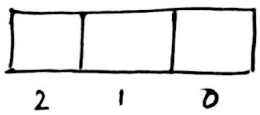
eg. $2x^4 + 4x^3 - 8x^2 + 6x + 1$



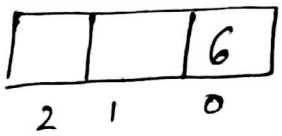
Queue

- 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)

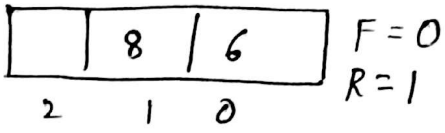
eg. 6, 8, 10



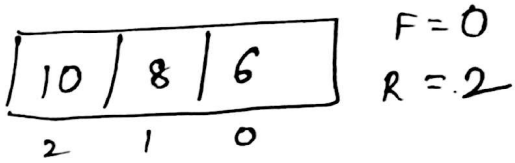
a)



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
3. Set Queue [Rear] := Item
4. Return.

(2) Deletion (or dequeue)

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

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

3) INSERTION AS A LINKED LIST-

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

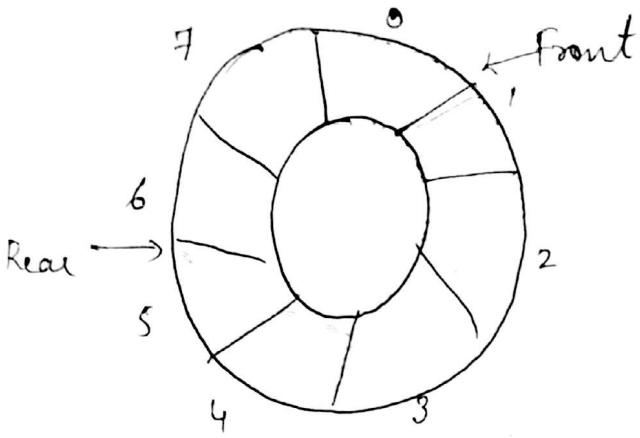
1) If AVAIL = NULL, write: OVERFLOW and Exit
2) Set NEW := AVAIL and AVAIL := LINK [AVAIL]
3) Set INFO [NEW] := ITEM and LINK [NEW] := NULL
4) If FRONT = NULL, set FRONT := REAR := NEW
   else
   LINK [REAR] := NEW and REAR := NEW
5) Exit

4) DELETION AS A LINKED LIST-

1) If FRONT = NULL, write: UNDERFLOW and Exit
2) Set ITEM := INFO [FRONT]
3) Set FRONT := LINK [FRONT] 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



front = 0, rear = 0

Insert : Q[rear] = x;
rear = (rear + 1) mod n;

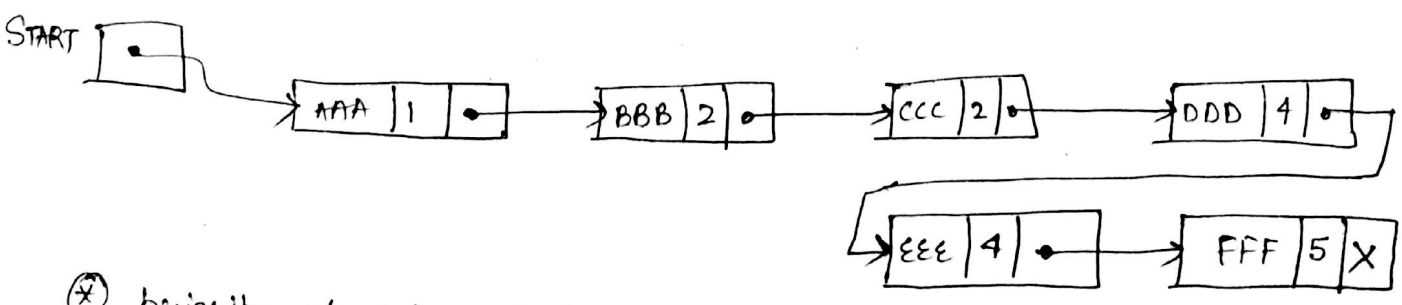
Delete : x = Q[front];
front = (front + 1) mod n;

Full queue is indicated by : → front == (rear + 1) mod 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 prn;
  struct pque * next;
};
```



(X) priority of 1 > priority of 2

Deque

(double ended queue)

- Insertion & deletion can be performed at both the ends.
- Types
  - i/p restricted queue
  - o/p 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.