22-11-23

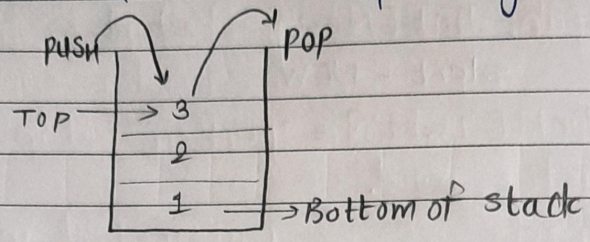
Page No.
Date

Unit 04 : Stack and Queue

* **Stack :**

-It's a collection of similar data elements where insertion & deletion operation takes place at only one end. Stack is also cla "LIFO" (Last In First Out). It is linear D.S.

The insertion & deletion opⁿ in case of stack are called as "PUSH" & "POP" respectively



* **Representation of stack in memory:**

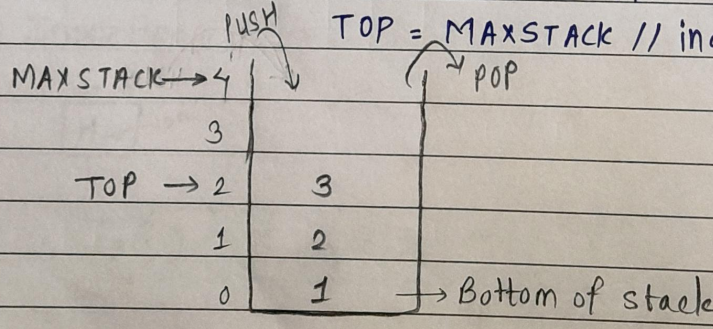
2 ways :- (i) Array representation (ii) Linked list represent

(i) **Array representation:**

In this, the pointer variable "TOP" which contain the locⁿ of topmost element of the stack, & the variable "MAXSTACK" which use maximum number no. of elements can be held by the stack.

The condition -: TOP = 0 (NULL) // indicates stack is empty

TOP = MAXSTACK // indicates stack is full.



In the diagram above, the stack contains 3 elements and the maximum elements the stack can hold is 5.

So, MAXSTACK = 5 & TOP = 3

Page No.
Date

* Operations on stack:

1. Push operation:

- The process of adding new element at the top of the stack is cla 'push' opn. Pushing of new elements onto stack always increment top by 1.
- In case the array is full & there is no space for new insern i.e top = maxstack, then this condn is cla 'stack overflow'.

Algorithm:

PUSH (stack, top, maxstack, item)
step 1: if top = maxstack then
write overflow and exit.
step 2: top = top + 1
step 3: Set stack [top] = item
step 4: Exit

2. Pop operation:

- The process of deleting an element from the stack is cla 'pop' opn. In case when stack doesn't contain any element i.e top = NULL, then this condn is cla 'stack underflow.'

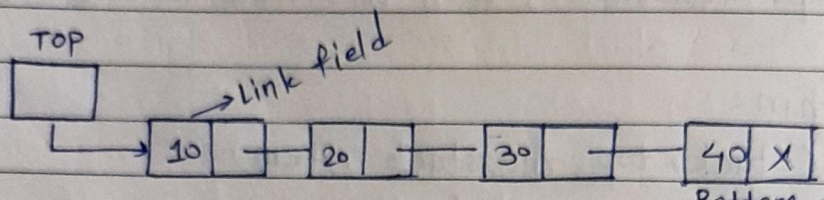
Algorithm:

POP (top, item, stack)
step 1: if top = NULL then
write underflow and exit
step 2: Set item = stack [top]
step 3: Set top = top - 1
step 4: Exit

23-11-23

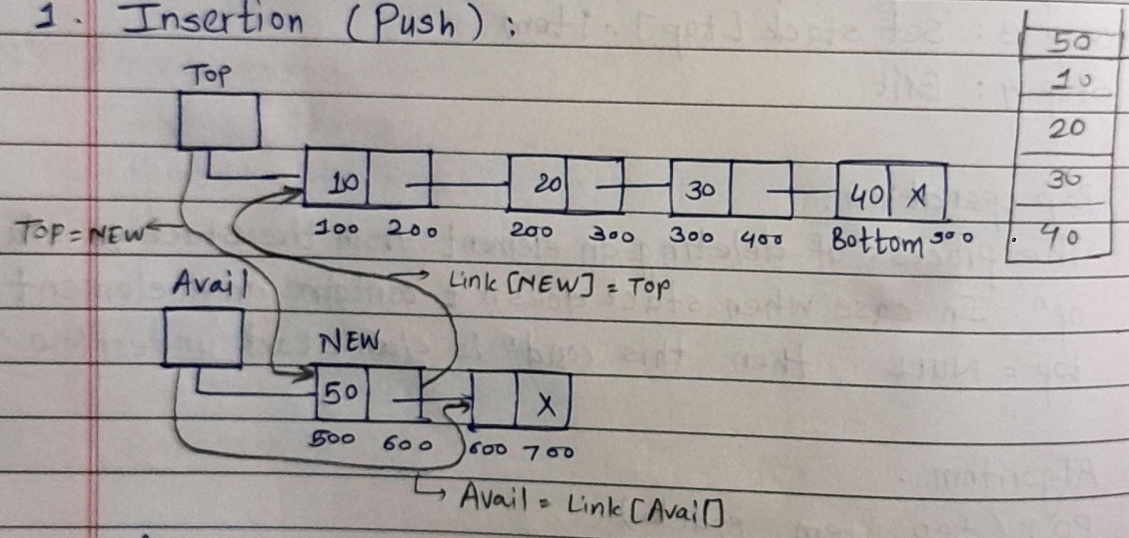
Page No.
Date

- Array representation of stack is very easy and convinient but it allow only representing fixed size stack.
- In some stack, the size of stack may vary during execution.
- So, the solution is to represent the stack using linked list.



* Operations on stack :

1. Insertion (Push) :



Algorithm :

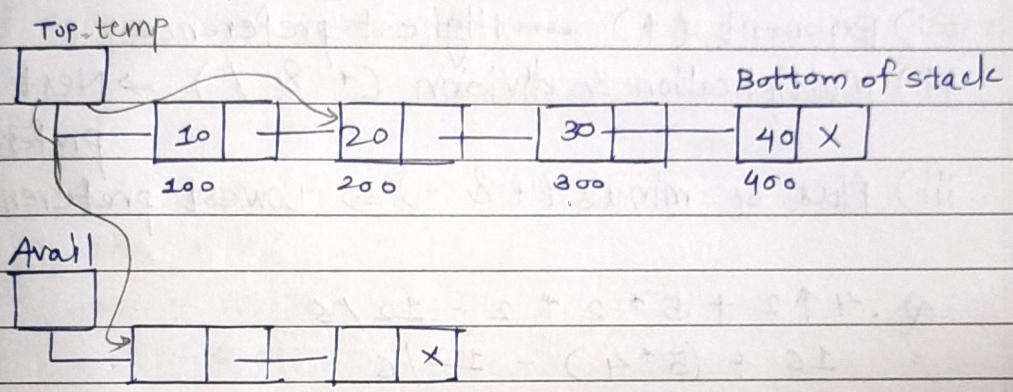
Push [TOP, NEW, Avail, Item]

step 1 : if Avail = NULL
write overflow and exit.

step 2 : Set Avail = NEW
Avail = Link [Avail]

Step 3 : Set Link [NEW] = TOP
Step 4 : Set Info [NEW] = Item
Step 5 : Set Top = NEW
Step 6 : Exit

2. Deletion (Pop) :



Algorithm :
Pop [Top, Item, temp, Avail]

Step 1 : if TOP = NULL
write underflow and exit
Step 2 : set Item = Info [Top]
Step 3 : set temp = Top
Step 4 : Link [temp] = Top
Step 5 : set Link [temp] = Avail
Avail = temp

* conversion:- Infix to prefix & postfix → push operators into the stack
- Prefix to infix & postfix → push operands into the stack from right to left.
- Postfix to infix & prefix → same as prefix to infix & postfix but from R → L

* Polish notations:

- An arithmetic exp. consists of operators & operands.
- e.g (A+B) * C
- The problem to evaluate the expression is to follow the order of evaluation by using the preference rule

i) Exponent (↑) ⇒ Highest preference.
ii) Multiplication & division (* & /) ⇒ Next highest preference.
iii) Plus & minus (+ & -) ⇒ Lowest preference.

Q. 4 ↑ 2 + 5 * 2 ↑ 2 - 12/6
= 16 + (5*4) - 12/6
= 16 + 20 - 2
= 34

* Polish notations for arithmetic expression :

There are 3 notations :-

① Infix
↳ The conventional way of writing an exp. is cla "infix" notn. e.g A+B, C-D etc.
↳ Syntax : <op> operator <op>
↳ operand

② Prefix
↳ Always fix the operator before the operands.
e.g +AB, +CD etc.
↳ Syntax : operator <op><op>

③ Postfix
↳ Always fix the operator after the operands.
e.g AB+, CD-
↳ Syntax : <op><op> operator

Page No.
Date

* Rules to convert infix epx. into prefix & postfix:
1) Parantheses must be solved first.
2) All the operators must be shifted only once according to their preferences.
3) The sub-exp. which has been converted into prefix & postfix is to be treated as single operand.
4) Once the conversion done, remove the parantheses.

e.g Infix to prefix

A + (B * C)
⇒ A + (* BC)
⇒ + A (* B C)
⇒ + A * BC

e.g Infix to postfix

A + (B * C)
⇒ A + (BC *) → opr can be shifted only once ✓
⇒ A (BC *) + → ++ not possible ✗.
⇒ A BC * +

** Evaluation of postfix expression:
8M Consider, the following expre., P: 5, 6, 2, +, *, 12, 4, /, -

To evaluate:-
1) Add ')' (closing paran.) at the end of exp. which indicates the terminating condⁿ.
2) Scan exp. from Left → Right.
3) -While scanning if operand encounters, then put it into stack.
-If operator encounters, then pop 2 operands from stack and perform opⁿ.

Page No.
Date

Algorithm :
Postfix (val)

Step 1 : Add the closing parantheses at the end of the expression (right parantheses)

Step 2: Scan the exp. from L → R & repeat st 3 & 4 for each element of expression until ')' is encounter.

Step 3: If operand encounters, push it into stack

Step 4: If operator encounters, then
i) remove top 2 ele. from the stack.
ii) evaluate the exp.
iii) push the result onto stack

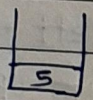
Step 5: Set val to the top element of the stack

Step 6: Exit

ans : 1) Add the right parantheses at the end of exp.
P: 5, 6, 2, +, *, 12, 4, /, -)

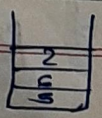
2) Scan the exp. until right parantheses is encountered.

3) Push 5 onto the stack



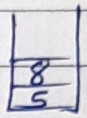
4) Push 6 onto the stack

5) Push operand 2 onto the stack



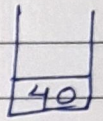
6) Operator '+' is encountered, so pop 2 ele. from the stack & evaluate the result.

i.e 6 + 2 = 8 & push 8 onto the stack



7) Operator '*' is encountered, so pop the elements from stack & evaluate it.

i.e 8 * 5 = 40 & push 40 onto the stack



8) Push operand '12' onto the stack

9) Push operand 4 onto the stack

10) Operator '/' is encountered, so pop 2 ele. from the stack & evaluate it

i.e 12/4 = 3 & push 3 onto the stack

4/12 ⇒ Not possible

11) Operator '-' is encountered, so pop the elements from the stack & evaluate it.

i.e 40 - 3 = 37 & push 37 onto the stack

3 - 40 ⇒ Not possible.

12) As ')' encountered, stop the process.

Conversion of infix to postfix:

Assume the simple exp. The exp. may be paranthesized or unparanthesized. First we've to append the left paranthesis at the start of infix exp. & initialize the stack ~~with~~ right paranthesis.

Algorithm:

Step 1: Push open paratheses onto the stack and add close parn. at the end of expression.

step 2: Scan the exp. from L → R & repeat step 3 to 6 until stack is empty.

step 3: If operands encounters, then add it to exp.

step 4: If open paran. is encoutered, then push it onto stack.

step 5: If an operator is encounterd, then
i) Repeatedly pop the elements from stack & add it to the exp.,
Each operator which has same or high precedance than existing one.
ii) Add the operator to the stack.

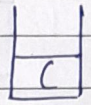
step 6: If right paran. is encounted, then
i) Repeatedly pop ele. from stack & add it to exp each operator ~~t~~ the highest precedance is pop first until the left paran. is encountered.
ii) Remove the left paran.

step 7: Exit.

Page No.
Date

Consider, the following exp:
Q: A + (B * C - CD / E ↑ F) * G) * H

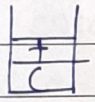
1) Push the open paran. onto the stack & add the closing paran. at the end of exp.



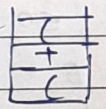
2) Scan from L → R

3) Operand A is encountered, then add it to Q.
Q: A

4) Operator + is encountered, so push it onto stack

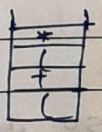


5) Operator '(' is encountered, so push it onto the stack



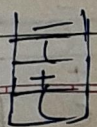
6) Operand B is encountered, so add to Q
Q: A B

7) Operator * is encountered, so push it onto the stack



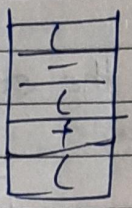
8) Operand C is encountered, so add it to Q
Q: A B C

9) Operator - is encountered, but it's preference is less than that * (which have high preference) in stack, so first pop * operator, then push - onto the stack.



Q: A B C *

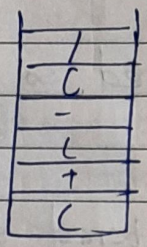
10) Operator '(' is encountered, so push it onto the stack



11) Operand D is encountered, so add it to Q.

Q : A B C * D

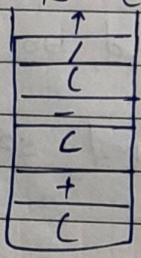
12) Operator / is encountered, so push it onto the stack.



13) Operand E is encountered, so add it to Q.

Q : A B C * D E

14) Operator ↑ is encountered, so ~~add~~ push it onto the stack



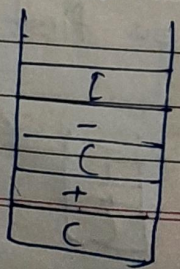
15) Operand F is encountered, so add it to q.

Q : A B C * D E F

16) Operator ')' is encountered, so repeatedly pop the elements from stack until open paran. is encounter

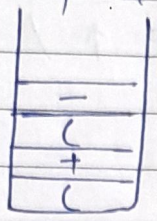
Q : A B C * D E F ↑ /

17) ~~Operator~~ * ~~is~~ encou

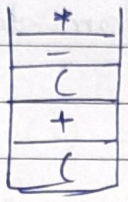


Page No.
Date

Remove open paran. from the stack and no need to add it to exp. Q



17) Operator * is encountered, so push it onto stack

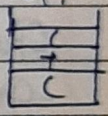


18) Operand G is encountered, so add it to Q.

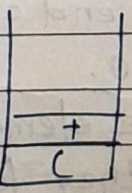
Q : A B C * D E F ↑ / G

19) Operator ')' is encountered, so repeatedly pop elements until open paran. is encountered.

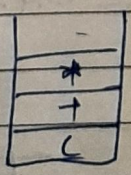
Q : A B C * D E F ↑ / G * -



Remove open paran. from stack & don't add to Q.



20) Operator * is encountered, so push it onto the stack.



21) Operand H is encountered, so add it to exp Q.

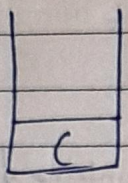
Q : A B C * D E F ↑ / G * - H

29-11-23

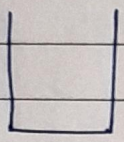
Page No.
Date

22) Operator ')' is encountered, so repeatedly pop ele. of stack until '(' encounter.

Q: A B C * D E F ↑ / G * - H * +



Remove open paran. from stack & don't add it to Q



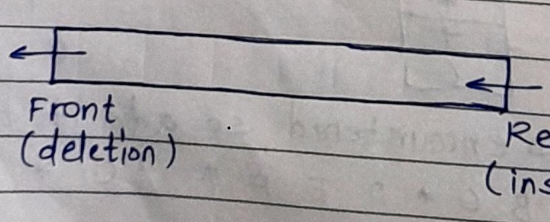
23) Stack is empty, so ~~stack~~ stop the process.

Required expression is,

A B C * D E F ↑ / G * - H * +

* Queue:

- Queue is the linear list of elements in which deletion is takes place at one end cla "Front" & insertion takes place at another end cla "Rear" end.
- It is also cla FIFO.
- The order in which the elements enter in a queue is the order in which they leave.
- The diff. with stack is that the insertion & deletion takes place only at one end.



Page No.
Date

Memory representation of queue:

There are 2 ways to represent the queue in memory:

i) Array representation

- A 1-D array, say Q[1-N] can be used to represent a queue. In this rep^n 2 pointers are associated with the — one is FRONT & another is REAR which indicates the 2 ends of queue. For the insertion of new element REAR will update & for deletion FRONT will be updated. There are 3 states of the queue with this rep^n :- ① If the queue is empty, then FRONT = REAR = -1.

② If the queue is full, then FRONT = 1 & REAR = N or FRONT = REAR + 1.

③ If the queue contain elements, then we can find no. of elements by using formula, REAR - FRONT + 1

Operations on queue:

1. Insertion: To insert an element.
2. Deletion : To delete the element.

1. Insertion :

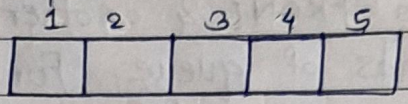
Algo - Insert (Queue, Front, Rear, Item)

Step 1 : if Front = 1 and Rear = N
write : Queue is full & return.

Step 2 : if Front = NULL then
set Front = Rear = 1
else if Rear = N then
set Rear = 1
else
R = R + 1

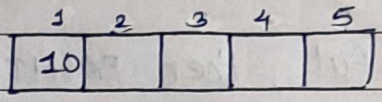
Step 3: Set Queue[Rear] = Item
Step 4: Exit

e.g Consider, queue (Q) hold N = 5 elements & consider initially queue is empty



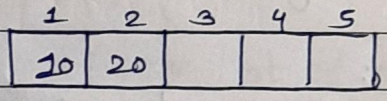
1) As Q is empty Front = Rear = -1

2) Insert 10 in Q.



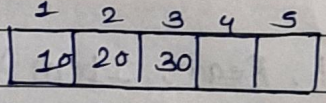
Front = Rear = 1

3) Insert 20



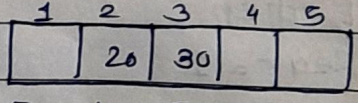
Front = 1 , Rear = 2

4) Insert 30



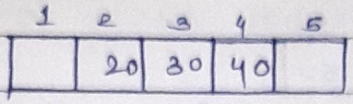
Front = 1 , Rear = 3

5) Delete 10 from Q.



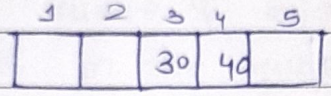
Front = Front + 1 = 2
Rear = 3

6) Insert 40



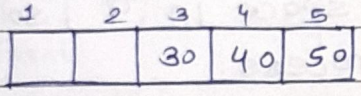
Front = 2 , Rear = 4

7) Delete 20



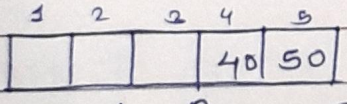
Front = 3 , Rear = 4

8) Insert 50



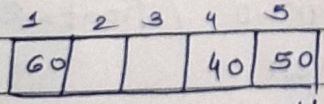
Front = 3 , Rear = 5

9) Delete 30



Front = 4 , Rear = 5

10) Insert 60



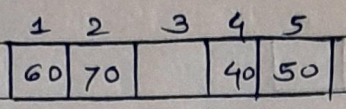
Rear = 1 // Once Q is full then reset Rear = 1

Front = 4

As Q is full i.e. Rear reach to the end of Q (R = N)
So, for the new insertion, reset Rear = 1.

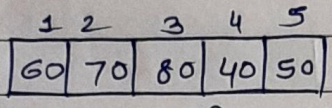
Q [Rear] = Q[1] = Item = 60

11) Insert 70



Front = 4 , Rear = 2

12) Insert 80



Front = 4 , Rear = 3

As Front = Rear + 1 , Q is full.

As we don't have the space in Q, so stop processing and return from the process.

2. Deletion:

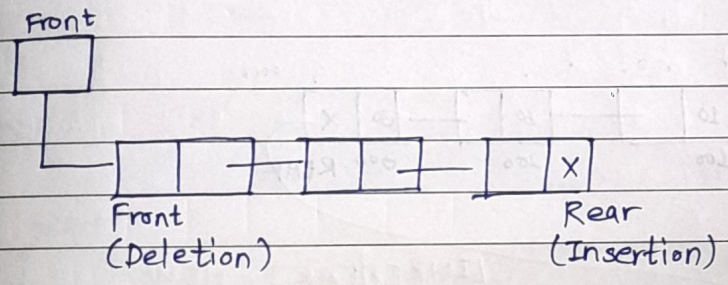
Algo Delete (Q, Front, Rear, Item)

Step 1: if Front = NULL
write: underflow and return.
step 2: Set Item = Q[Front]
step 3: if Front = Rear then
set Front = Rear = 0 / Front = Rear = NULL
else if Front = N then
Set Front = 1 // Reset Front to 1.
else
Front = Front + 1
Step 4: Exit.

30-11-23

Page No.
Date

* LL rep^n of queue:
- LL rep^n of queue requires 2 ptr. variables - Front & Rear.
- These pointer variables pointing to the nodes Front & Rear in LL.
- The info field of node holds the information or ele. of the queue & add^r field holds the address of next ele. in the queue.



* Operations on queue used in linked list:

1. Insertion:
- In case of insertion algo. we need to check -
  (1) Whether there is a free space in Avail in list.
      if true, then remove 1st node from the Avail list.
  (2) Add item from Rear end.

Algorithm
Insert (FRONT, REAR, AVAIL, ITEM[], Φ, LINK)

Step 1: if AVAIL = NULL then
        write: overflow and return

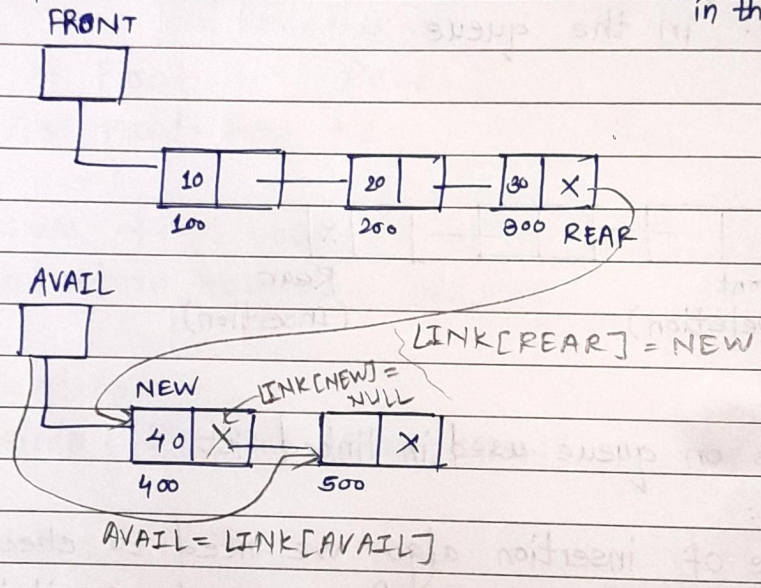
Step 2: Set NEW = AVAIL
        AVAIL = LINK[AVAIL]

Step 3: Set INFO[NEW] = ITEM
LINK[NEW] = NULL

Step 4: ~~if~~ Set LINK[REAR] = NEW
REAR = NEW

Step 5: if FRONT = NULL then
Set FRONT = REAR = NEW // if queue is empty, then insert new ele. as a 1st ele in the queue.

Step 6: Exit.



2. Deletion:

- The deletion algo. delete the item from the LL.
- This algo. is exactly same as we've seen in LL data structure except instead of 'start' we've to use 'Front' here and 'Rear' pointer indicates the end of list.
- For deletion, we've to check whether queue is empty or not.
  i.e FRONT = NULL
  then display the msg. 'underflow' & ~~return~~ exit from that process.

Page No.
Date

Algorithm:

Algo - delete (FRONT, REAR, PTR, INFO, ITEM

step 1: if FRONT = NULL then
write: underflow and exit.

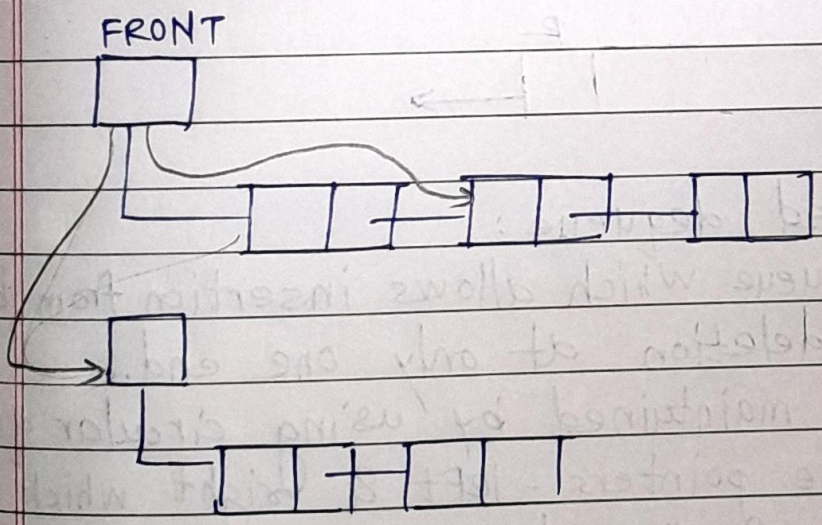
step 2: Set PTR = FRONT.

step 3: INFO [PTR] = ITEM // copy info of first node

step 4: FRONT = LINK [PTR] to delete.

step 5: LINK [PTR] = AVAIL
AVAIL = PTR

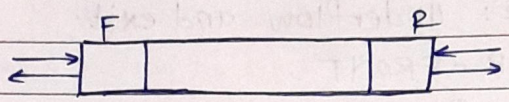
step 6: Exit



01-12-23

Page No.
Date

* Dequeue (Double ended queue):
- Another type of queue is c/a "dequeue" or "deck"
- It is a linear list in which ele. can be inserted & deleted from both ends; but not from the middle



* Types of dequeue:

1. Input restricted de.:
- An input res. deq. is a queue which allows insertion at only one end; but allows deletion from both ends.

2. Output restricted dequeue:
- It's a dequeue which allows insertion from both ends; but deletion at only one end.
- A dequeue is maintained by using circular queue or array with the pointers - left & right which points to the ends of the dequeue. It is a d.s which inherits the properties of stack & queue. This data struc. doesn't follow FIFO rule.