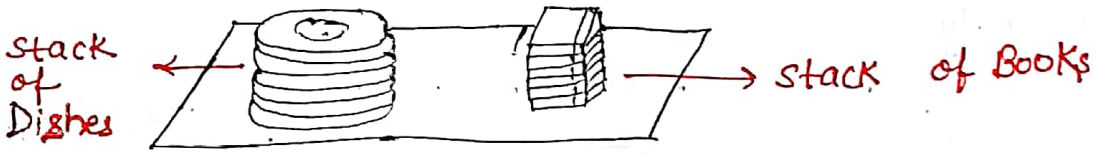
STACK

→ The linear lists and arrays allowed one to insert and delete elements at any place in the list i.e. at the beginning, at the end or in the middle.

→ situation with restrict insertion & deletion only at the beginning or the end of list, not in middle.
For this situation, stack and Queue data structures are useful.

→ A stack is a linear structure in which elements may be added or removed only at one end.
E.g: a stack of dishes, stack of books



any dish or book may be added or removed only from the top of any of stacks.

→ the last element to be added is the first element to be removed.

→ Stacks are also called Last-in first-out (LIFO) list.

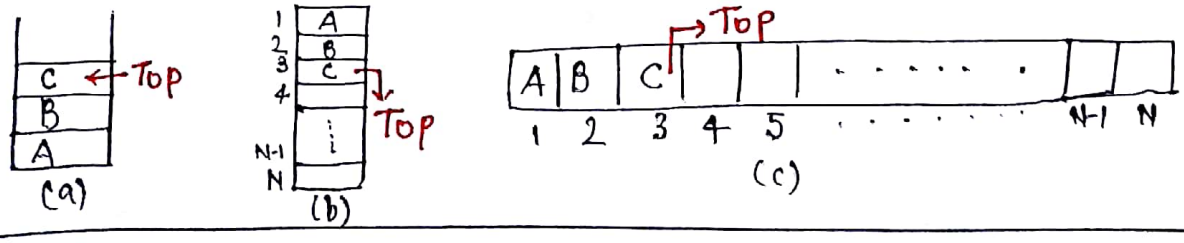
→ piles and push-down lists also used as a names for stack

→ element may be inserted or deleted only at one end, called top of a stack.

Basic operations associated with stacks

(a) "Push" is the term used to insert an element into a stack

(b) "Pop" is the term used to delete an element into a stack



R1, R2

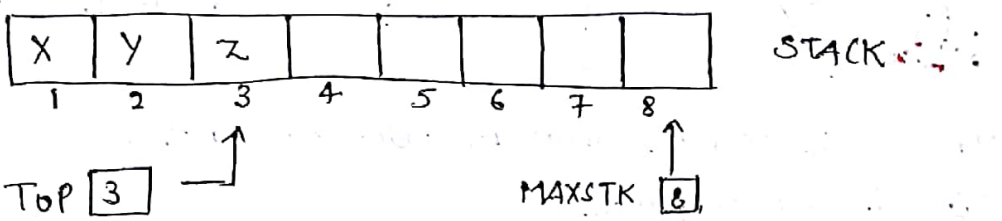
→ Stacks are frequently used to indicate the order of processing of data when certain steps of the processing must be postponed until other conditions are fulfilled.

Array Representation of Stack

→ Stacks may be represented as a one-way list or a linear array.

→ Each stack will be maintained by linear array STACK; a pointer variable TOP, which contains the location of the top element of stack. and a variable MAXSTK which gives the maximum number of elements that can be held by the stack.

→ The condition TOP=0 or TOP=NULL will indicate that the stack is empty.



→ The operation of adding (pushing) an item onto a stack and removing (poping) an item may be implemented as

→ first test whether there is a space in stack for the new element; if not then this condition known as overflow. (For Pushing/Inserting an element)

→ first test whether there is an element in the stack to be deleted; if not then we have the Condition known as underflow

(For poping an element)

R1, Web lecture - 2, 3

STACK as ADT

→ Instead of storing data in each node, store a pointer to the data.

→ Within the ADT, the stack node looks like any linked list node except that it contains a pointer to the data than the actual data.

→ As the data pointer type is unknown, it is stored as a pointer to void.

→ The head node and the data nodes are encapsulated in ADT.

```
//Stack ADT type definition
typedef struct node
{
    void *data;
    struct node *link;
} stack_node;
typedef struct
{
    int count;
    stack_node *top;
} stack;
```

Application of STACK: Polish Notation and Reverse Polish Notation.

→ Let Q be an arithmetic expression involving constants and operations.

→ Operator precedence: Highest : Exponentiation (↑)
Next highest : Multiplication (*) & division (/)
Lowest : Addition (+) & subtraction (−)

→ No unary operation and same level operations performed from Left to Right (Assume)

E.g 2↑3+ 5*2↑2 −12/6
8+5*4− 12/6
8+20−2 → 28−2
26.

R1, R2

Polish Notation

→ Infix Notation: A+B, C-D, E*F, G/H

(A+B)*C , A+(B*C)

→ Polish Notation, named after Polish mathematician Jan Lukasiewicz, refers to the notation in which the operator symbol is placed before its two operand

eg:

+AB, −CD, *EF, /GH

(A+B)*C = [+AB]*C = *+AB

→ a fundamental property of Polish Notation is that the order in which the operations are to be performed is completly determined by the positions of operators and operands in the expression.

→ ~~Ref~~ Reverse Polish Notation, refers to the analogous notation in which the operator symbol is placed after its two operands

AB+, CD−, EF*, GH/

→ this notation is also called post-fix or suffix

Evaluation of a Postfix Expression

→ Suppose P is an arithmetic expression written in postfix notation.

→ Algorithm, which uses a STACK to hold operands, evaluates P.

Evaluation of a Postfix Expression Algorithm

1. Add a right parenthesis ")" at the end of P
2. Scan P from left to right and repeat steps 3 and 4 for each element of P until the sentinel ")" is encountered
3. if an operand is encountered, put it on STACK
4. if an operator ⊗ is encountered, then:
   (a) Remove the two top elements of STACK, where A is the top element and B is the next-to-top element
   (b) Evaluate B⊗A
   (c) Place the result of (b) back on STACK
5. Set VALUE equal to the top element on STACK
6. Exit.

Eg: P: 5, 6, 2, +, *, 12, 4, /, −

Scanned Symbol	STACK
5	5
6	5, 6
2	5, 6, 2
+	5, 8
*	40
12	40, 12
4	40, 12, 4
/	40, 3
−	37
)	

R1, R2

Transforming Infix expressions into Postfix Expression.

Algorithm

POLISH (Q, P)

1. Push "(" onto STACK and add ")" to the end of Q.
2. Scan Q from left to right and repeat steps 3 to 6 for each element of Q until the STACK is empty.
3. If an operand is encountered, add it to P
4. If a left parenthesis is encountered, push it onto STACK.
5. if an operator (⊗) is encountered, then:
   (a) Repeatedly pop from STACK and add to P each operator (on the top of STACK) which has the same precedence as or higher precedence than (⊗)
   (b) Add (⊗) to STACK.
6. If a right parenthesis is encountered, then
   (a) Repeatedly POP from STACK and add to P each operator (on top of STACK) until a left Parenthesis is encountered
   (b) Remove the left parenthesis (do not add the left parenthesis to P)
7. Exit

→ (A+B)*C*(D-E)*(F+G) : AB+C*DE-FG+*-

→ (300+23)*(43-21)/(84+7) : 300,23+43 21-*84 7+/
= 78.08

→ (4+8)*(6-5)/((3-2)*(2+2)) : 48+65-*32-22+*/
= 3

Q: A+(B*C-(D/E↑F)*G1)*H

Scanned Symbol	STACK	Expression P
A	(	A
+	(+	A
C	(+C	A
B	(+C	AB
*	(+C*	AB
C	(+C*	ABC
-	(+C-	AB C*
C	(+C-(	AB C*
D	(+C-(	AB C*D
/	(+C-(/	AB C*D
E	(+C-(/	AB C*DE
↑	(+C-(/↑	AB C*DE
F	(+C-(/↑	AB C*DEF
)	(+C-	AB C*DEF↑/
*	(+C-*	AB C*DEF↑/
G1	(+C-*	AB C*DEF↑/G1
)	(+	AB C*DEF↑/G1*-
*	(+*	AB C*DEF↑/G1*-
H	(+*	ABC*DEF↑/G1*-H
)		ABC*DEF↑/G1*-H*+

R1: Lipschtuz, "DSWC" (6.11 - 6.15) ~~6.15~~)

R2: Tenenbaum, "DSUC" (95)

R3: Sahani, "FODS" (91)

Recursion (Principle of Recursion)

→ Suppose P is a procedure containing either a Call statement to itself or a Call statement to a second procedure that may eventually result in a Call statement back to the original procedure P. then P is called recursive procedure.

→ Recursive Procedure must have following two properties

1. There must be certain criteria, called base criteria, for which the procedure does not call itself.
2. Each time the procedure does call itself (directly or indirectly), it must be closer to the base criteria.

→ A recursive procedure with these 2 properties is said to be well-defined.

→ A function is said to be recursively defined if the function definition refers to itself, in order for the definition not to be circular, it must have following 2 properties:

1. there must be certain arguments called base values for which the function does not refer to itself.
2. Each time the function does refer to itself, the argument of the function must be closer to a base value.

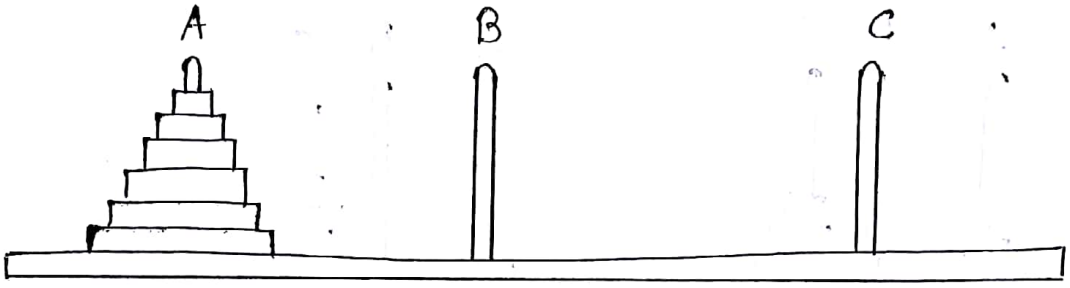
E.g
```
f = 1
Repeat for K = 1 to N
    set f = K * f
```

Towers of Hanoi:

Suppose there are three pegs, labeled A, B and C are given and suppose on peg A there are placed a finite number n of disks with decreasing size.

For, $n=6$, the object of the game is to move the disks from peg A to peg C using Peg B as an auxiliary.

The rules are as follows:



Initial Setup of Towers of Hanoi with $n=6$

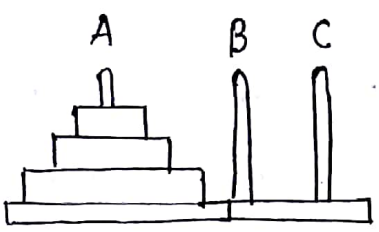
$X \to Y$ denote "Move top disk from peg X to peg Y".

where X and Y may be any of the three pegs.

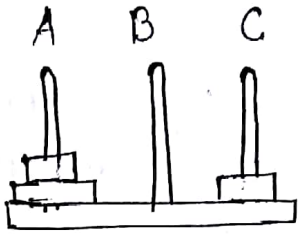
For $n=3$, solution of Towers of Hanoi

Move top disk from peg A to peg C
Move top disk from peg A to peg B
Move top disk from peg C to peg B
Move top disk from peg A to peg C
Move top disk from peg B to peg A
Move top disk from peg B to peg C
Move top disk from peg A to peg C

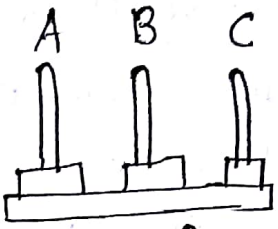
R_2: Tenenbaum, "DSUC" (120-125)



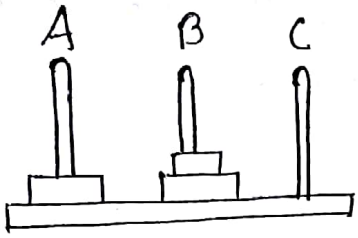
(a) Initial



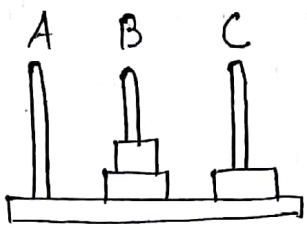
(1) A→C



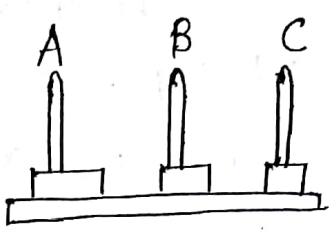
(2) A→B



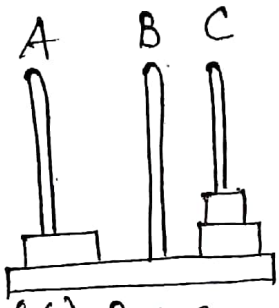
(3) C→B



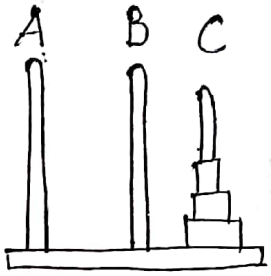
(4) A→C



(5) B→A



(6) B→C



(7) A→C

for completeness, solution for $n=1$ and $n=2$

$n=1$: $A \to C$

$n=2$: $A \to B$, $A \to C$, $B \to C$

Solution to TOH problem for $n>1$ may be reduced to following subproblems:

1. Move the top $n-1$ disks from peg A to peg B
2. Move the top disk from peg A to peg C: $A \to C$
3. Move the top $n-1$ disks from peg B to peg C

1. TOWER (N-1, BEG, END, AUX)
2. TOWER (1, BEG, AUX, END) or BEG → END
3. TOWER (N-1, AUX, BEG, END)

BEG → initial peg
END → final peg
AUX → mediator

For, n=4

TOWER(4, A, B, C)

A→B A→C B→C A→B C→A C→B A→B A→C

B→C B→A C→A B→C A→B A→C B→C

Simulating Recursion:

→ Knowing the process by which recursion passes data upward and downward through the called modules, you can isolate and preserve the variables unique to each recursive step and simply loop a given piece of code to achieve simulated recursion.

Since looping could start the code process at the same place each time, you must keep a place marker, indicating just where to begin the processing of current iteration

Principle of Recursion (Recursion) - ⌐¬.

Tail Recursion:

→ A function call is said to be tail recursive if there is nothing to do after the function returns except return its value.

Recursion Removal:

→ Can be removed by two ways
(i) through Iteration
(ii) through stack

Types of Recursion:

- Linear (makes a single call to itself)
- Tail (last thing the function does)
- Binary (two or more

R3: Sahani, "FODS" (160-169)

```
Tower (disk, beg, eng, aux)
  if disk==0,
     move disk from beg to end
else
     Tower (disk-1, beg, end, aux)
     move disk from beg to end
     Tower (disk-1, aux, beg, end)
```

PUSH (Stack, Top, MaxstK, element)

1. if Top = MaxstK, then print Overflow and Return.
2. set, Top = Top+1
3. Set, stack[Top] = element
4. Return

Pop (Stack, Top, element)

1. if Top=0 then print Underflow and Return
2. Set, element = stack [Top]
3. Set, Top = Top-1
4. Return.

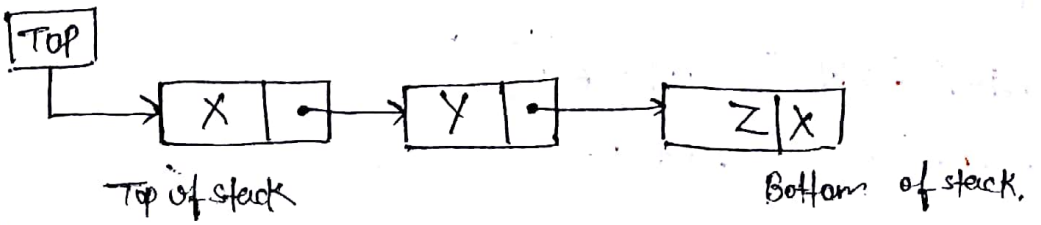
→ Value of Top is always changed before the insertion and after the deletion.

Linked List Representation of Stack:

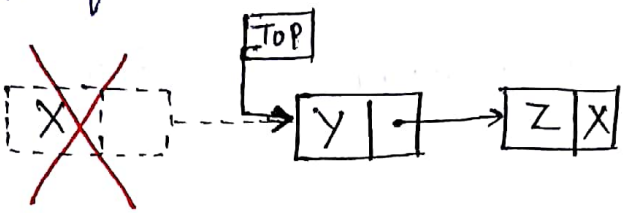
→ stack using a one-way list or singly linked list

→ array representation received a specific amount of memory space. for efficient use of this reserved space Linked list representation is used.

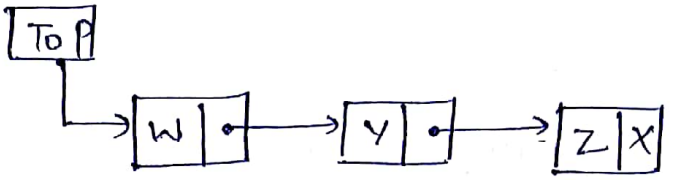
→ the INFO fields of the nodes hold the element in the stack. START pointer behaves as the TOP pointer variable of stack and the null pointer of the last node in the list signals the bottom of stack



Pop "X" from the stack



PUSH "W" in the stack



Push_Link (INFO, LINK, TOP, AVAIL, ITEM)

1. if AVAIL = NULL, then write Overflow and EXIT
2. Set NEW = AVAIL and AVAIL = LINK[AVAIL]
3. Set INFO[NEW] = ITEM
4. Set LINK[NEW] = TOP
5. Set TOP = NEW
6. EXIT

Pop_Link (INFO, LINK, TOP, AVAIL, ITEM)

1. if TOP = NULL then write Underflow and EXIT.
2. Set ITEM = INFO[TOP]
3. Set TEMP = TOP and TOP = LINK[TOP]
4. Set LINK[TEMP] = AVAIL and AVAIL = TEMP
5. EXIT

R1: Lipschutz, "DSWC" (6.1 - 610)

R2: Tenenbaum, "DSUC" (84-86)

QUEUE:

→ a linear list of elements in which deletions can take place only at one end. Called the front and insertions can take place only at the other end called the rear.

→ also called first-in first-out (FIFO) list, since the first element in a queue will be the first element out of the queue.

→ the order in which elements enter a queue is the order in which they leave.

E.g the people waiting in line at a bank form a queue where the first person in line is the first person to be waited on and so on.

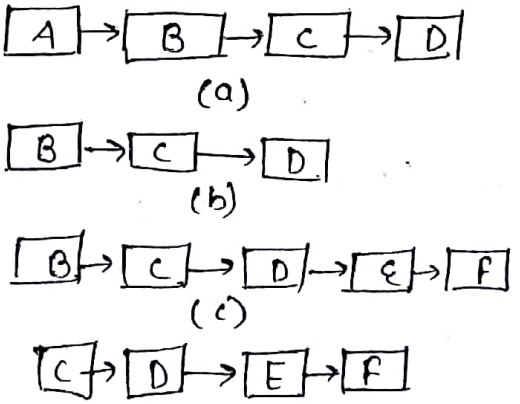
→ a time sharing system, in which programs with the same priority form a queue while waiting to be executed

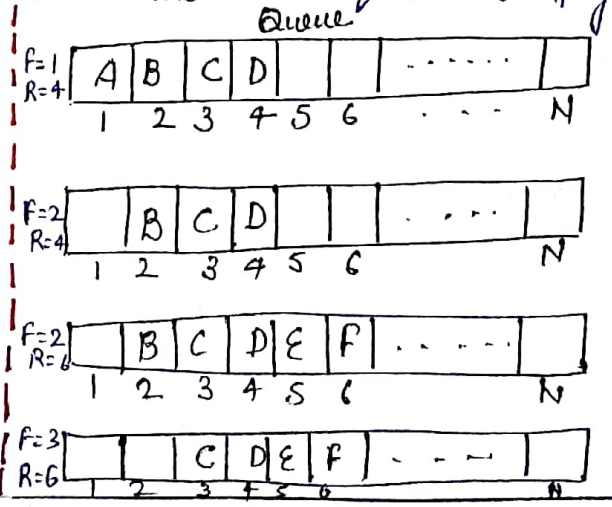
Queue Representation

→ may be represented by one-way lists or linear arrays.

→ Each of our queue will be maintained by a linear array Queue and two pointer variables: FRONT and REAR

→ FRONT, Containing the location of the front element of queue.
REAR, Containing the location of the rear element of queue.
condition FRONT = NULL will indicate that the queue is empty.





→ Whenever an element is deleted from the queue, the value of FRONT is increased by 1 i.e.

FRONT = FRONT + 1

→ Similarly, whenever an element is added to the queue, the value of REAR is increased by 1 i.e.

REAR = REAR + 1

→ after N insertions, the rear element of queue will occupy. QUEUE[N]

→ when REAR = N, move the entire queue to the beginning of the array, changing FRONT and REAR accordingly and then insert. Reset, REAR = 1

→ if FRONT = N, and element of Queue is deleted, reset FRONT = 1 instead of increasing FRONT to N+1

→ Suppose queue contains only one element i.e.

FRONT = REAR ≠ NULL

→ and suppose element is deleted.

then

FRONT = NULL and REAR = NULL

indicate that the queue is empty.

F=-1, r=-1

Overflow ⇒ $(f == 0 \;\&\&\; r == Max-1) \;||\; f == r+1$

Insertion ⇒ if f = -1, r = 0, f = 0

else if r = Max-1

r = 0

else

r++

Deletion, if f = Max-1

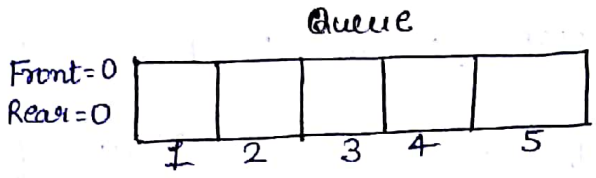
f = 0

else

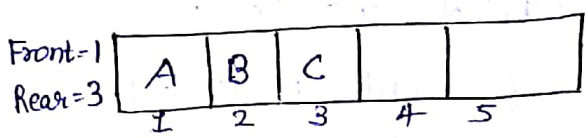
f++

R1, R3, Webelecture-3

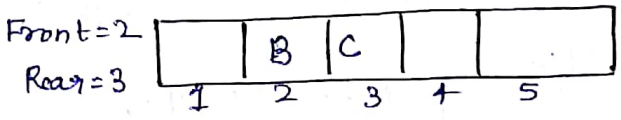
(a) Initially Empty



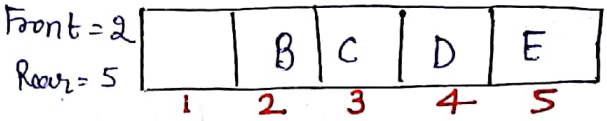
(b) A, B and then C inserted



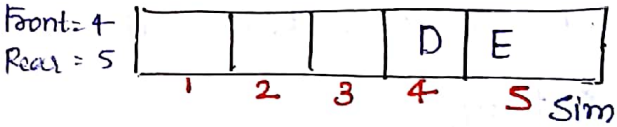
(c) A deleted



(d) D and then E inserted



(e) B and C deleted

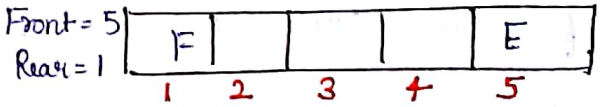


Simple Queue

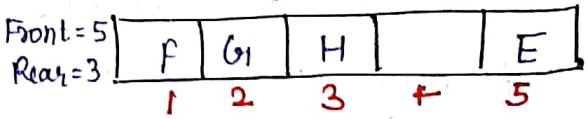
(f) F inserted

Circular Queue

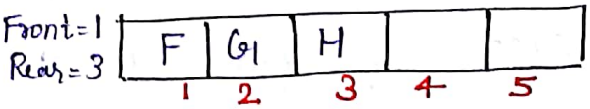
(g) D deleted



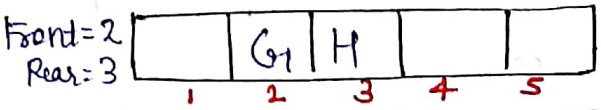
(h) G1 and then H inserted



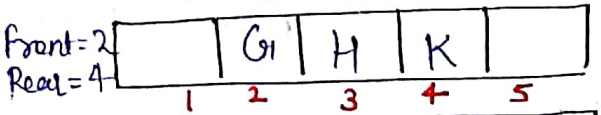
(i) E deleted



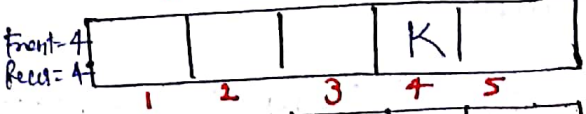
(j) F deleted



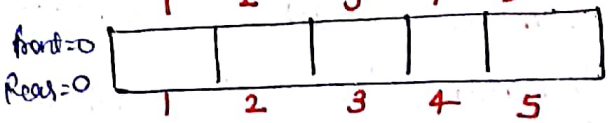
(K) K inserted



(L) G1 and H deleted



(m) K deleted, Queue Empty



Q INSERT (Queue, N, front, Rear, Item)

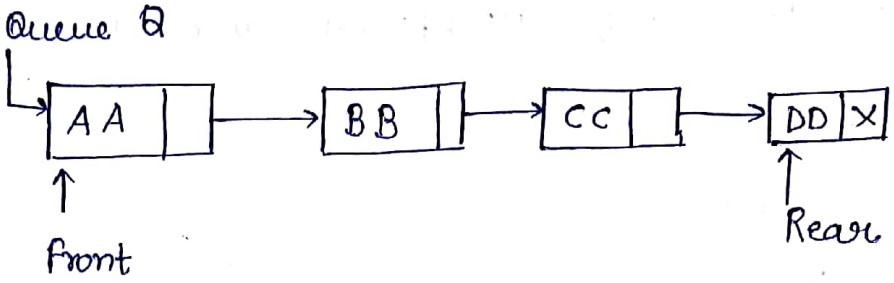
1. if front = 1, Rear = N ; or if front = Rear + 1 then write OVERFLOW and Return
2. if front = Null then
   set front = 1 and Rear = 1
   else
   if Rear = N then
   set Rear = 1
   else
   set Rear = Rear + 1
3. set Queue [Rear] = item
4. Return

Q DELETE (Queue , N, front , Rear, Item)

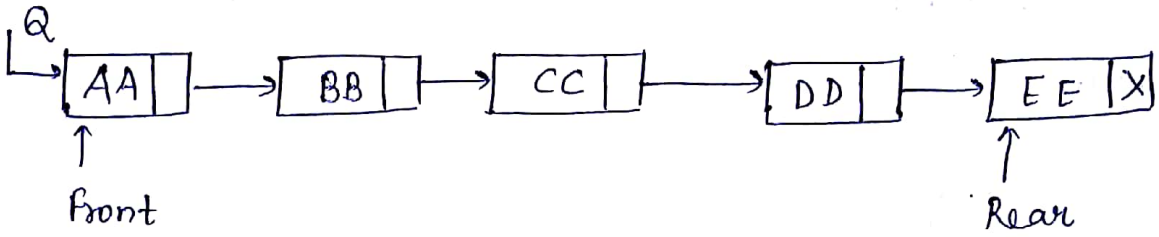
1. if front = Null then Write Underflow and Return
2. set ITEM = QUEUE [front]
3. if front = Rear then
   set front = Null and Rear = Null
   else if front = N then
   set , front = 1
   else
   set , front = front + 1
4. Return

R1 : Lipschutz, "DSWC" (6.1-6.65)
R2 : Tenenbaum, "DSUC" (174-183)
R3 : Sahani. "FODS" (77-86)

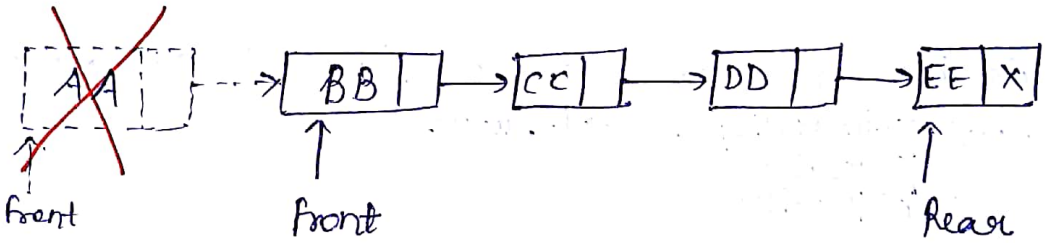
Linked List Representation of Queue



Insert 'EE' into Q.



Delete "AA" from Q



→ array representation suffers from the drawback of limited Queue capacity.

LINKQ_INSERT(Info, Link, Front, Rear, Avail, Item)

1. if Avail = Null then 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 then Front = Rear = New
   else
   set Link[Rear] = New and Rear = New
5. Exit.

LINKQ_DELETE (Info, Link, front, Rear, Avail, Item)

1. if front = Null then write UNDERFLOW and Exit
2. set Temp = front
3. item = Info [Temp]
4. front = Link [Temp]
5. Link [Temp] = Avail and Avail = Temp
6. Exit

Queue as ADT

→ basic design of stack ADT is also followed by Queue ADT.

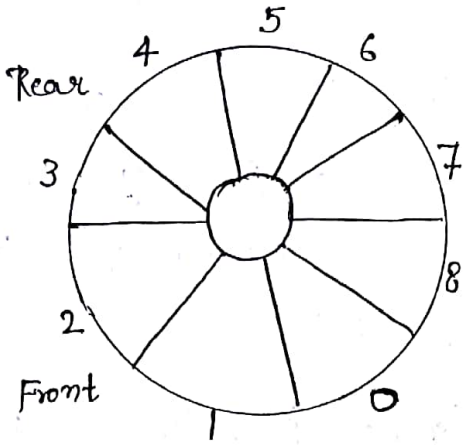
→ it would be a queue header file such as queue.h

Types of Queue:- Linear/Simple → Dequeue
Circular
Circular Queue priority

→ In an array implementation of Queue, it is possible that the insert function gives a signal stating that the ~~eqe~~ even queue is full if the queue is partially free. Such a situation occurs as the queue continues to move to the right unless the 'front' and 'rear' catch up with each other and both are reset to 0 queue is full in the delete procedure.

→ To resolve this problem, the elements of the array should be shifted by one position to the left on every deletion that is made. To overcome this issue, assume that the array is circular.

R1, R2



← Circular Queue

Deques:

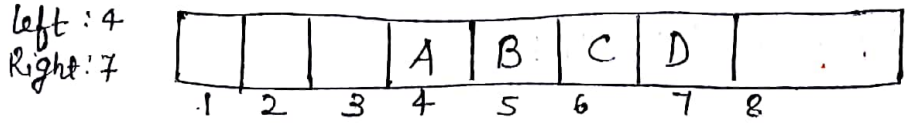
A deque ("deck" or "dequeue") is a linear list in which elements can be added or removed at either end but not in the middle.

→ this is also called double-ended queue.

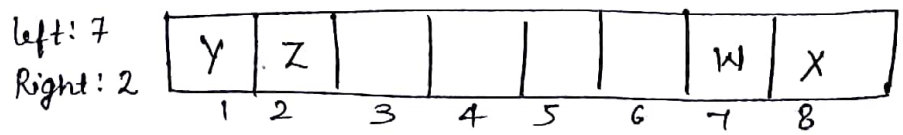
→ there are two variation of deque
- → an input-restricted deque
- → an output-restricted deque.

→ An input-restricted deque is a deque which allows insertion at only one end of the list but allows the deletions at both ends of the list

→ An output-restricted deque is a deque which allows deletions at only one end of the list but allows insertions at both ends of list.



Deque



Deque

R1, R2

Priority Queues:

A priority queue is a collection of elements such that each element has been assigned a priority and such that the order in which elements are deleted and processed comes from the following rules:

1. An element of higher priority is processed before any element of lower priority.
2. Two elements with the same priority are processed according to the order in which they were added to the queue.

→ prototype of a priority queue is a time sharing system: programs of high priority are processed first and with same priority form a standard queue

Application of Queue

→ Online business applications such as applying for jobs,

→ print spools

R_1: Lipschutz, "DSWC" (6.66 - 6.92)

R_2: Tenenbaum, "DSUC" (187 - 200)

R_3: Sahani, "FODS" (112 - 114)