LIMITATIONS OF ASYMPTOTIC NOTATIONS:-

→ Many algorithms are very hard to analyze mathematically.

→ Often the exact nature of the average data is unknown, so analysis of such cases is impossible.

→ Big-oh analysis only tells you how it grows with the size of the problem, not how efficient of the algorithm is.

→ If the program isn't going to handle large amounts of data, it may not matter how inefficient the algorithm is. A simple, clear algorithm may be more important.

15/12/2010 ARRAYS

INTRODUCTION:-

-Data structures are classified as either linear or non-linear. A data structure is said to be linear if its elements form a sequence or in other words, a linear list. There are two basic ways of representing such linear structures in memory. One way is to have the linear relationship between the elements represented by means of sequential memory locations. These linear structures are called arrays.

- The operations one normally performs on any structure, whether it be an array or a linked l include the following:

(a) TRAVERSAL: Processing each element in the lis
(b) SEARCH: Finding the location of the element with given value or the record with a given k
(c) INSERTION: Adding a new element to the list.
(d) DELETION: Removing an element from the list.
(e) SORTING: Arranging the elements in some type o
(f) MERGING: Combining two lists into a single list.

- Since arrays are usually easy to traverse, search and sort, they are frequently used to st relatively permanent collections of data. On the other hand, if the size of the structure and th data in the structure are constantly changing than the array may not be as useful a structu as the linked list.

LINEAR ARRAYS:-

- A linear array is a list of a finite number n of homogeneous data elements (i.e., data elements of the same type) such that:

(a) The elements of the array are referenced respectively by an index set consisting of n consecutive numbers.
(b) The elements of the array are stored respectively in successive memory locations.

-The number n of elements is called the length or size of the array. If not explicitly stated, we will assume the index set consists of the integers 1, 2 ... n. In general, the length or the number of data elements of the array can be obtained from the index set by the formula

$$\text{length} = UB - LB + 1 \rightarrow (1)$$

where UB is the largest index, called the Upper bound and LB is the smallest index, called the lower bound, of the array. Note that length = UB when LB = 1.

-The elements of an array A may be denoted by the subscript notation

$$A_1, A_2, A_3 \ldots, A_n$$

or by the parentheses notation (used in FORTRAN, PL/1 and BASIC).

$$A(1), A(2), \ldots, A(N)$$

or by the bracket notation (used in PASCAL).

$$A[1], A[2], \ldots, A[N].$$

We will usually use the subscript notation or the bracket notation. Regardless of the notation, the number k in A[K] is called a subscript or an index and A[K] is called a subscripted variable.

Eg:

(a) Let DATA be a 6-element linear array of integers such that

DATA[1] = 247 DATA[2] = 56 DATA[3] = 429 DATA[4] = 135
DATA[5] = 87 DATA[6] = 156

Sometimes we will denote such an array by simply writing.

DATA: 247, 56, 429, 135, 87, 156

The array DATA is frequently pictured as follows

DATA

1	247
2	56
3	429
4	135
5	87
6	156

(a)

DATA

247	56	429	135	87	156
1	2	3	4	5	6

REPRESENTATION OF LINEAR ARRAYS IN MEMORY:-

Let LA be a linear array in the memory of the computer. The memory of the computer is simply a sequence of addressed locations as pictured. Let us use the notation

LOC(LA[K]) = address of the element LA[K] of the array LA.

As previously noted, the elements of LA are stored in successive memory cells. Accordingly, the computer doesnot need to keep track of the address of every element of LA, but needs to keep track only of the address of the first element of LA, denoted by

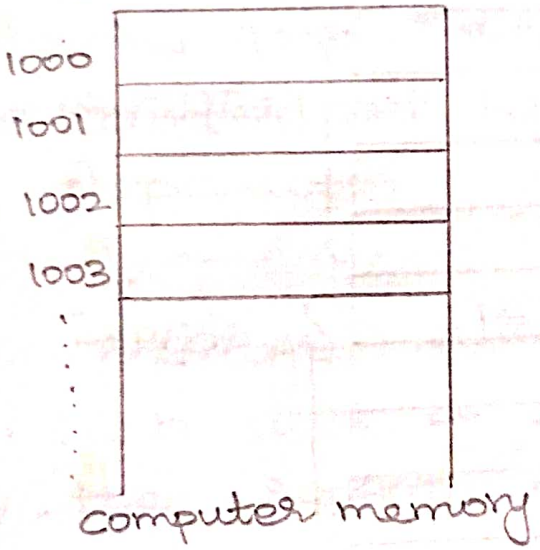
Base(LA)

and called the base address of LA. Using this address Base(LA), the computer calculates the

address of any element of LA by the following formula:

$$LOC(LA[K]) = Base(LA) + w(K - lowerbound) \rightarrow (2)$$

where w is the number of words per memory cell for the array LA. Observe that the time to calculate LOC(LA[K]) is essentially the same for any value of K. Furthermore, given any subscript K, one can locate and access the contents of LA[K] without scanning any other element of LA.



EXAMPLE:

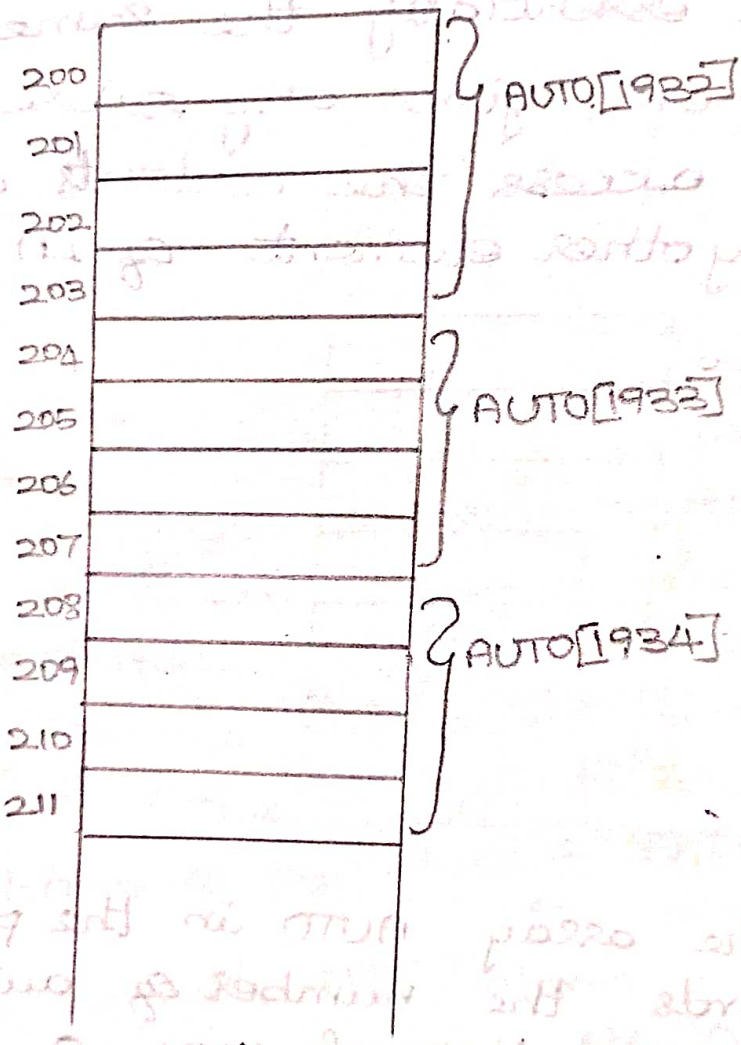
Consider the array AUTO in the previous example, which records the number of automobiles sold each year from 1932 through 1984. Suppose AUTO appears in memory as pictured as below. That is, Base(AUTO) = 200, and w=4 words per memory cell for AUTO. Then

LOC(AUTO[1932]) = 200, LOC(AUTO[1933]) = 204,
LOC(AUTO[1934]) = 208, ...

The address of the array element for the year K=1965 can be obtained by using equ (2).

LOC(AUTO[1965]) = Base(AUTO) + w(1965 - lowerbound)
= 200 + 4(1965 - 1932) = 332.

Again we emphasize that the contents of element can be obtained without scanning any other element in array AUTO.



TRAVERSING LINEAR ARRAY:-

Let A be a collection of data elements stored in the memory of the computer. Suppose we want to print the contents of each element of A or suppose we want to count the number of elements of A with a given property. This can be accomplished by traversing A, that is by accessing and processing each element of A exactly once.

The following algorithm traverses a linear array LA. The simplicity of the algorithm comes from the fact that LA is a linear structure.

ALGORITHM: (Traversing a Linear Array) Here LA is a linear array with lower bound LB and upper bound UB. This algorithm traverses LA applying an operation PROCESS to each element of LA.

1. [Initialize counter] set K := LB.
2. Repeat steps 3 and 4 while K ≤ UB.
3. [Visit element] Apply PROCESS to LA[K].
4. [Increase counter] set K := K+1.
   [End of step 2 loop]
5. Exit.

We also state an alternative form of the algorithm which uses a repeat-for loop instead of the repeat-while loop.

ALGORITHM: (Traversing a Linear Array) This algorithm traverses a linear array LA with lower bound LB and upper bound UB.

1. Repeat for K = LB to UB.
   Apply PROCESS to LA[K].
   [End of loop].
2. Exit.

Eg:

Consider the array AUTO in the first eg. which records the number of automobiles sold each

year from 1932 through 1984. Each of the following modules, which carryout the given operation, involving traversing AUTO

a) Find the number NUM of years during which more than 300 automobiles were sold.

1. [Initialization step] set NUM:=0
2. Repeat for K=1932 to 1984.
   If AUTO[K]>300, then: set NUM:=NUM+1.
3. Return.

INSERTING AND DELETING:-

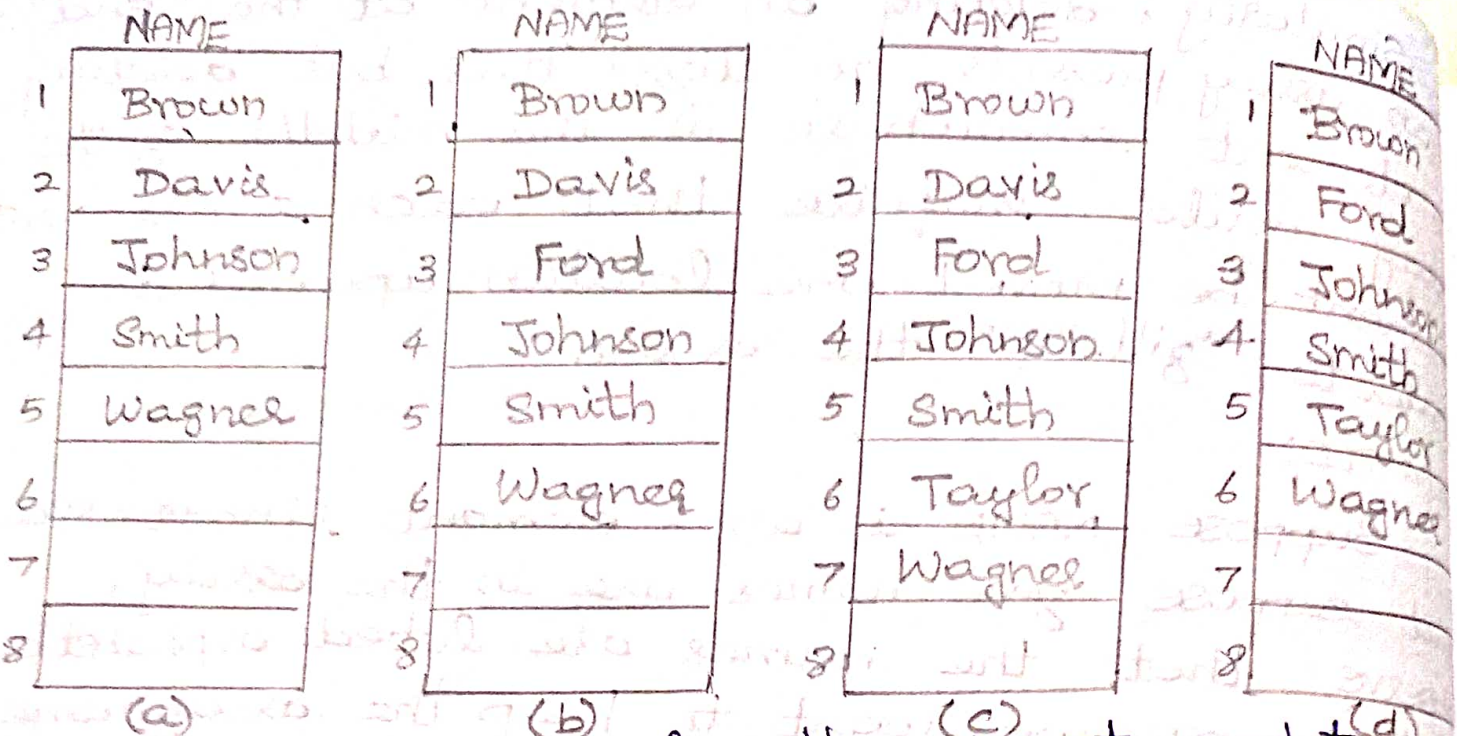
Let A be a collection of data elements in the memory of the computer. "Inserting" refers to the operation of adding another element to the collection A, and "deleting" refers to the operation of removing one of the elements from A. This section discusses inserting and deleting when A is a linear array.

Inserting an element at the "end" of a linear array can easily done provided the memory space allocated for the array is large enough to accommodate the additional element. On the other hand, suppose we need to insert an element in the middle of the array. Then, on the average, half of the elements must be moved downward to new locations to accommodate the new element and keep the order of the other elements.

Similarly, deleting an element at the "end" of an array presents no dizziculties, but deleting an element somewhere in the middle of the array would require that each subsequent element be moved one location upward in order to "fill-up" the array.

EXAMPLE:

Suppose NAME is an 8-element linear array, and suppose five names are in the array, observe that the names are listed alphabetically, and suppose we want to keep the array names alphabetical at all times. Suppose Ford is added to the array. Then Johnson, Smith and Wagner must each be moved downward one location. Next suppose Taylor is added to the array; then Wagner must be moved. Last, suppose Davis is removed from the array. Then the five names Ford, Johnson, Smith, Taylor and Wagner must each be moved upward one location. Clearly such movement of data would be very expensive if thousands of names were in the array.



The following algorithm inserts a data element ITEM into the Kth position in a linear array LA with N elements. The first four steps create space in LA by moving downward one location each element from the Kth position. We emphasize that these elements are moved in reverse order (i.e., first LA[N], then LA[N-1]..., and last LA[K]; otherwise data might be erased). In more detail, we first set J:=N and then, using J as a counter, decrease J each time the loop is executed until J reaches K. The next step, step 5, inserts ITEM into the array in the space just created. Before the exit from the algorithm, the number N of elements in LA is increased by 1 to account for the new element.

ALGORITHM:-

(Inserting into a linear array) INSERT(LA,N,K, ITEM)

Here LA is a linear array with N elements and K is a positive integer such that K≤N. This algorithm inserts an element ITEM into the Kth position in LA.

1. [Initialize counter] set J:=N.
2. Repeat steps 3 and 4 while J≥K.
3. [Move Jth element downward] set LA[J+1]:=LA[J].
4. [Decrease counter] set J:=J-1.
   [End of step2 loop]
5. [Insert element] set LA[K]:=ITEM.
6. [Reset N] set N:=N+1.
7. Exit.

The following algorithm deletes the Kth element from a linear array LA and assigns it to a variable ITEM.

ALGORITHM:-

(Deleting from a linear array) DELETE(LA,N,K,ITEM)

Here LA is a linear array with N elements and K is a positive integer such that K≤N. This algorithm deletes the Kth element from LA.

1. Set ITEM:=LA[K].
2. Repeat for J=K to N-1:
   [Move J+1st element upward] set LA[J]:=LA[J+1].
   [End of loop].

3. [Reset the number N of elements in LA] set N := N-1.

4. Exit

MULTI-DIMENSIONAL ARRAYS:-

The linear arrays discussed so far are also called one-dimensional arrays. since each element in the array is referenced by a single subscript. Most programming languages allow two-dimensional and three-dimensional arrays, (i.e)., arrays where elements are referenced, respectively, by two and three subscripts. In fact, some programming languages allow the number of dimensions for an array to be as high as 7. This section discusses these multidimensional arrays.

TWO-DIMENSIONAL ARRAYS:-

A two-dimensional mxn array A is a collection of m.n data elements such that each element is specified by a pair of integers called subscripts, with the property that

$$1 \le J \le m \quad \text{and} \quad 1 \le K \le n.$$

The element of A with first subscript j and second subscript K will be denoted by

$$A_{J,K} \text{ or } A[J,K]$$

Two-dimensional arrays are called matrices in mathematics and tables in business

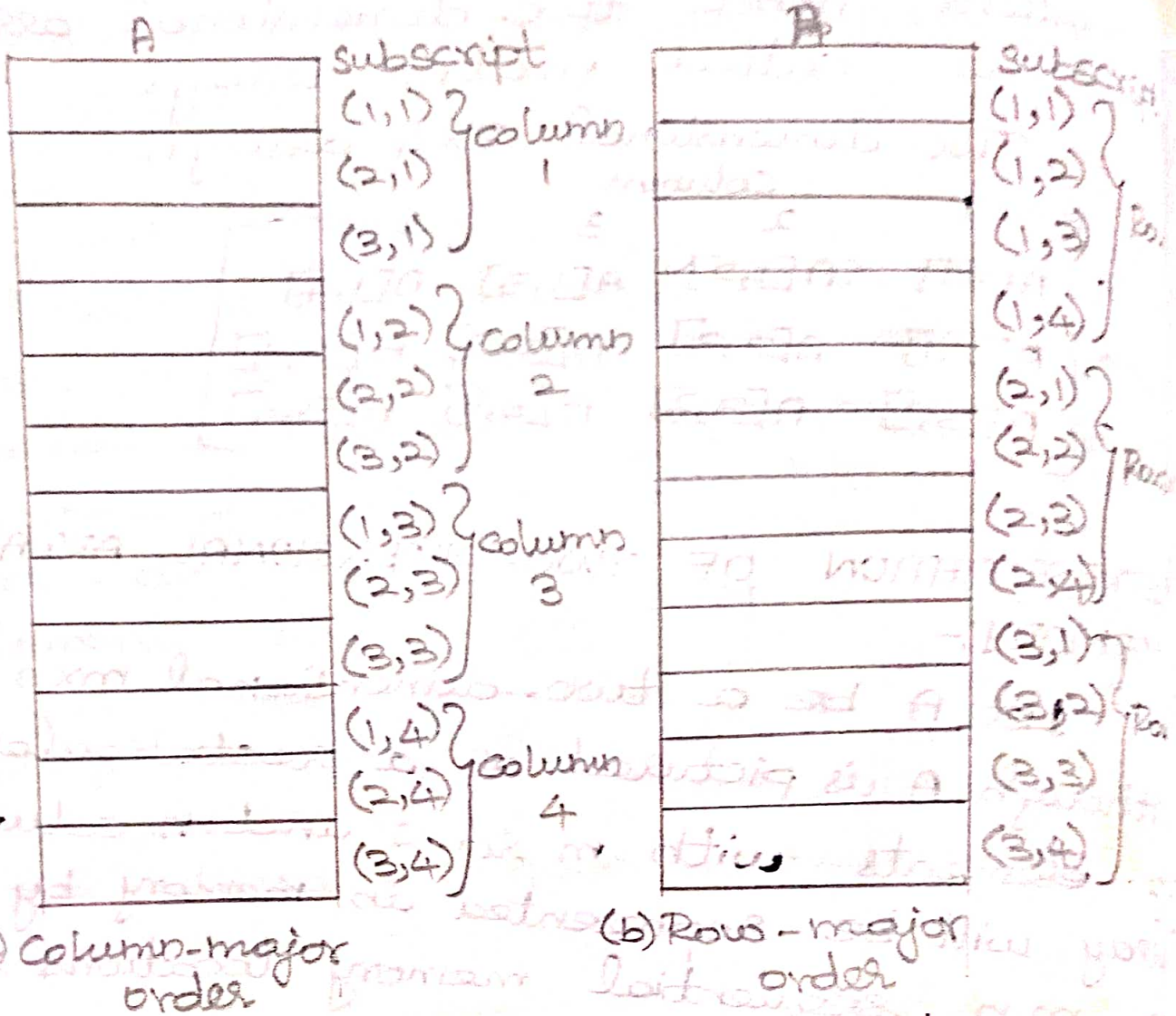
applications, hence two-dimensional arrays are sometimes called matrix arrays.

Two dimensional 3X4 array A.

rows \ columns	1	2	3	4
1	A[1,1]	A[1,2]	A[1,3]	A[1,4]
2	A[2,1]	A[2,2]	A[2,3]	A[2,4]
3	A[3,1]	A[3,2]	A[3,3]	A[3,4]

REPRESENTATION OF TWO-DIMENSIONAL ARRAYS IN MEMORY:-

Let A be a two-dimensional mxn array. Although A is pictured as a rectangular array of elements with m rows and n columns, the array will be represented in memory by a block of m.n sequential memory locations specifically, the programming language will store the array A either (1) column by column, is what is called column-major order, or (2) row by row, in row-major order.



(a) Column-major order

(b) Row-major order

A linear array LA, the computer doesn't keep track of the address LOC(LA[K]) of every element LA[K] of LA, but does keep track of Base(LA), the address of the first element of LA. The computer uses the formula,

$$LOC(LA[K]) = Base(LA) + w(K-1)$$

To find the address of LA[K] in time independent of r (column-major order).

$$LOC(A[J,K]) = Base(A) + w[M(K-1) + (J-1)] \rightarrow (4)$$

or the formula,

(Row-major order)

$$LOC(A[J,K]) = Base(A) + w[N(J-1) + (K-1)] \rightarrow (5)$$

Eg:

Consider the 25X4 matrix array SCORE. Suppose Base(SCORE)=200 and there are w=4 words per memory cell. Further more, suppose the programming language stores two-dimensional arrays using row-major order. Then the address of SCORE[12,3], the third test of the twelfth student, follows:

$$LOC(SCORE[12,3]) = 200 + 4[4(12-1) + (3-1)]$$
$$= 200 + 4[46]$$
$$= 384$$

we used equ (5).

STACK AND ITS APPLICATIONS

A stack is a linear structure in which items may be added (or) removed only at one end.

Eg: ▷ A stack of dishes
▷ A stack of towels

A last item to be added to the stack is the 1st element to be removed. stacks are also called as

"LAST IN FIRST OUT LISTS" (LIFO LISTS).

A stack is a list of elements inserted or deleted at one end, called the Top of the stack. The elements are removed from the stack in the reverse order of that in which they are inserted into the stack.

Two basic operation associate with stack:

* push
* pop

are the two operation mainly carried in stac

PUSH:

This is the term used to 'insert' an elem into the stack.

POP:

This is the term used to 'delete' an element from the stack.

The various operation associated with stack:

* create(S)
* Top(S)
* ISemt(S)

Create(S): Create(S) is used to create an 'empty set'.

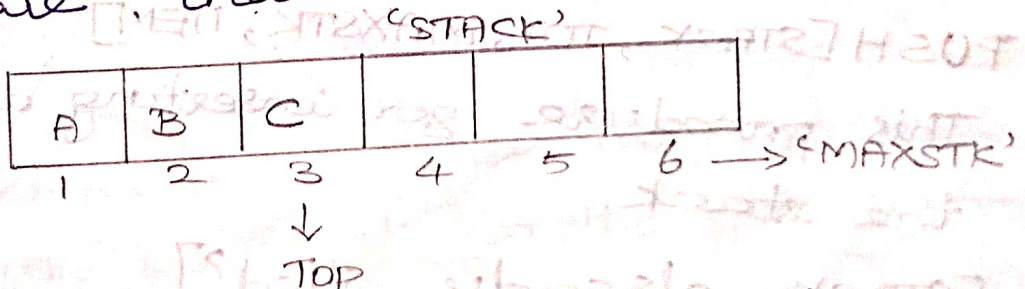
Top(S): Top(S) is used to create, 'return the top' element of the stack.

Isemt(S): Isemt(S) is used to return 'TRUE' if stack is empty else 'FALSE'.

Array represent of Stack:-

Stack may be represent in the computer in various ways, usually by means of a 'linear array'. Stack will be maintained by linear array. Stack, a pointer variable Top, which contains the location of the top element of the stack 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.



Since Top=3, stack has 3 elements A, B, C and since MAXSTK=7 there is room for four more element to be filled.

OPERATIONS OF STACK:-

The operation of adding (pushing) an i on to a stack. And the operation of removi (poping) an item from a stack, may be implemented by a following procedures called PUSH AND POP.

In executing the procedure PUSH, one must first test whether there is a place in the stack for new item, if not then we h the condition know as 'OVERFLOW' in execut the procedure for pop one must test wheth there is an element in the stack to be deleted, if not then we have the conditi 'UNDER FLOW'.

PROCEDURE FOR PUSH:-

PUSH [STACK, TOP, MAXSTK, ITEM]

This procedure for inserting an item int the stack.

[STACK already filled?]

if Top=MAXSTK, then print "OVERFLOW". and return

SET Top=Top+1 [increamented by 1]

SET STACK[Top]:= ITEM (insert item in the new position)

RETURN.

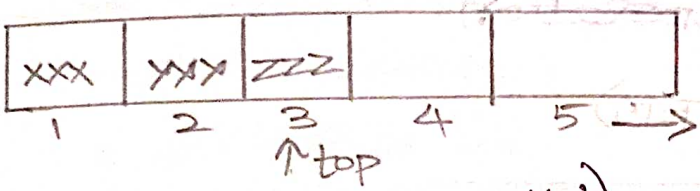
Explanation of the algorithm to be given there:

▷MAXSTK: This variable gives the maximum number of elements that can be held by stack.

▷TOP: This variable contain the location of the top element of the stack.

▷ITEM: This is the element to be inserted.

Eg:



PUSH(STACK, WWW)

Top ≠ MAXSTK & Top=3, the control is transferred to step 2.

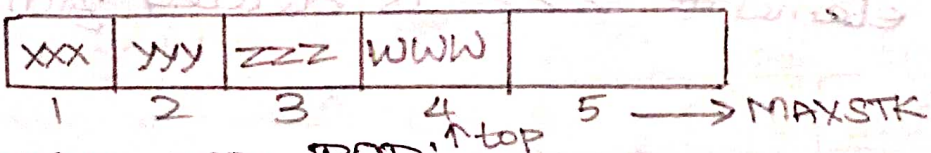
Top = Top+1

∴ Top=4

STACK[4]=WWW

RETURN.

Now the element WWW is inserted in the new position of stack in the position at 4.



PROCEDURE FOR POP:

POP[STACK, TOP, ITEM]

This procedure deletes the top element of

the stack and assign it to the variable ITEM

(stack has an item to be removed)

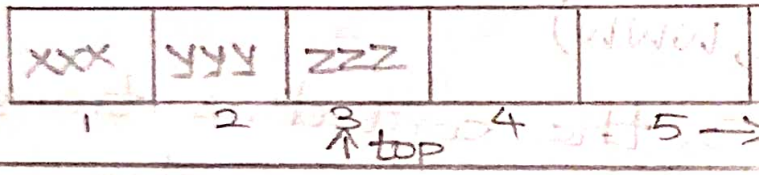
- if TOP=0, then print "UNDERFLOW" & RETURN
- SET ITEM:=STACK[TOP] (assign top item to item)
- SET TOP:=TOP-1 (decrease the top by 1)

RETURN

Eg:

The element 'zzz' to be deleted from the stack using pop operation

POP (STACK, ITEM)



TOP=3 & TOP≠'0' control is transferred to step2

ITEM=ZZZ

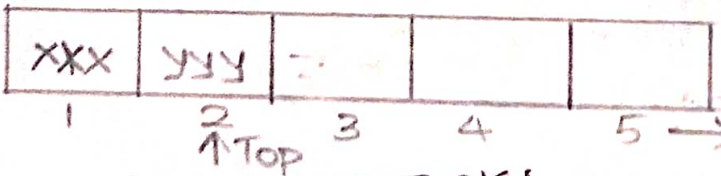
TOP=TOP-1

[3-1]

TOP=2

RETURN

Now the element zzz is deleted from the sta



APPLICATIONS OF STACK:-

The stack can be applied in two ways

▷Conversion of infix to postfix expression

▶ Conversion of Infix to Postfix expression:

Expression: An expression is a one which involve constants and operations.

The various types of expression are

- ▷ Postfix
- ▷ Infix
- ▷ Prefix

Postfix: In postfix expression the 'operators follow the 'operands'.

Eg: AB+C*

Infix: This type of expression is given by the notation, 'operator between operands'.

Eg: A+(B*C)

Prefix: In this type of expression the 'operator precedes the operands'

Eg: +AB*C

Conversion:-

Let Q be an arithmetic expression written in infix notation. Besides operands and operators, Q may also contain left and right paranthesis. we assume that the operators in Q consist only of exponential (^), multiplication (*), division (/), addition (+), subtraction (-) and that have the levels of precedures as given above.

▷Recursion.

▶Conversion of Inzix to Postzix expression:

Expression: An expression is a one which involve constants and operations.

The various types of expression are

▷Postzix

▷Inzix

▷Prezix

Postzix: In postzix expression the 'operators follow the 'operands'.

Eg: AB+C*

Inzix: This type of expression is given by the notation, 'operator between operands'.

Eg: A+(B*C)

Prezix: In this type of expression the 'operator preceeds the operands'

Eg: +AB*C

Conversion:-

Let Q be an arithmetic expression written in inzix notation. Besides operands and operators, Q may also contain left and right paranthesis. we assume that the operators in Q consist only of exponential (^), multiplication (*), division (/), addition (+), subtraction (-) and that have the levels of procedures as given above.

The following algorithm transverses the infix expression P. This uses a stack to temporarily host operators and left parenthesis.

Algorithm :-

POLISH(Q,P)

Suppose Q is an arithmetic expression retur in infix notation. This algorithm find the equivalent postfix P.

PUSH("into stack and add ")" to end of Q.

▷Scan Q from left to right and repeat step3 to 6 for each element of Q until the stack is empty.

▷If AND operands is an encounted, added it to P.

▷If a left parenthesis is encounted, push it to stack.

▷If X an operator is encounted, then

* Repeatly pops from stack and add to pead operator (on the top of stack) which as the same precedence as or higher precedence then X.

* Add X operator to stack(end of if structure)

▷ If a right parenthesis is encounted then
* Repeatly pops from stack and add to p each operator, until a left parenthesis is encounted.
• Remove the left parenthesis
[Don't add the left parenthesis top]
[end of if structure]
[end of step2 loop]
Exit

Conversion of Infix to Postfix expression.

Q = (A+(B*C-(D/E↑F)*G)*H)

Symbol	stack	expression P
(	(	
A	(	A
+	(+	A
(	(+(	A
B	(+(	AB
*	(+(*	AB
C	(+(*	ABC
-	(+(*-	ABC
(	(+(*-(	ABC
D	(+(*-(	ABCD
/	(+(*-(/	ABCD
E	(+(*-(/	ABCDE

↑	(+(*-(/↑	ABCDE
F	(+(*-(/↑	ABCDEF
)	(+(*-	ABCDEF↑/
*	(+(*-*	ABCDEF↑/
G	(+(*-*	ABCDEF↑/G
)	(+(*-)	ABCDEF↑/G*-*
*	(+*	ABCDEF↑/G*-*
H	(+*	ABCDEF↑/G*-*H
)		ABCDEF↑/G*-*H*+

The resultant postfix expression for

Q = (A+(B*C-(D/E↑F)*G)*H)

P = ABCDEF↑/G*-*H*+.

Recursion:-

Recursion is a concept that the function or procedure call itself again and again.

Steps to write Recursion function or procedure:-

- Determine whether the routine or function will halt.
- Determine the base case (not recursive call)
- Determine the general case (recursive call).
- Determine whether the procedure (or) function is correct.

Determine what the procedure (or) function Does.

ALGORITHM:-

procedure : N Fact(N)

#compute // N factorial

if N=0 then NFACT=1

ELSE NFACT=N*FACT(N-1)

END.

when N=4

NFACT(4) —①

4≠0

NFACT=4*FACT(3) —②

NFACT(3) —③

3≠0

NFACT=3*FACT(2) —④

NFACT(2) —(5)

2≠0

NFACT=2*NFACT(1) —⑥

NFACT(1) —(7)

1≠0

NFACT=1*NFACT(0) —⑧

0=0

NFACT=1 —⑨

NFACT(0)=1

sub NFACT① in ⑥

NFACT=2*1

NFACT=2

sub NFACT (2) in (4)

NFACT=3*2

NFACT=6

sub NFACT (3) in 2

NFACT=24

NFACT(4)=24

Verifying the recursion procedure or function:

▷Check whether there is non-recursive wa out of the procedure (or) function and check

▷whether this case works correctly.

▷Check whether each recursive call to the procedure (or) function involves a smaller case of the problem.

▷Check whether the whole procedure works correctly.

Tower of hanoi:-

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 siz

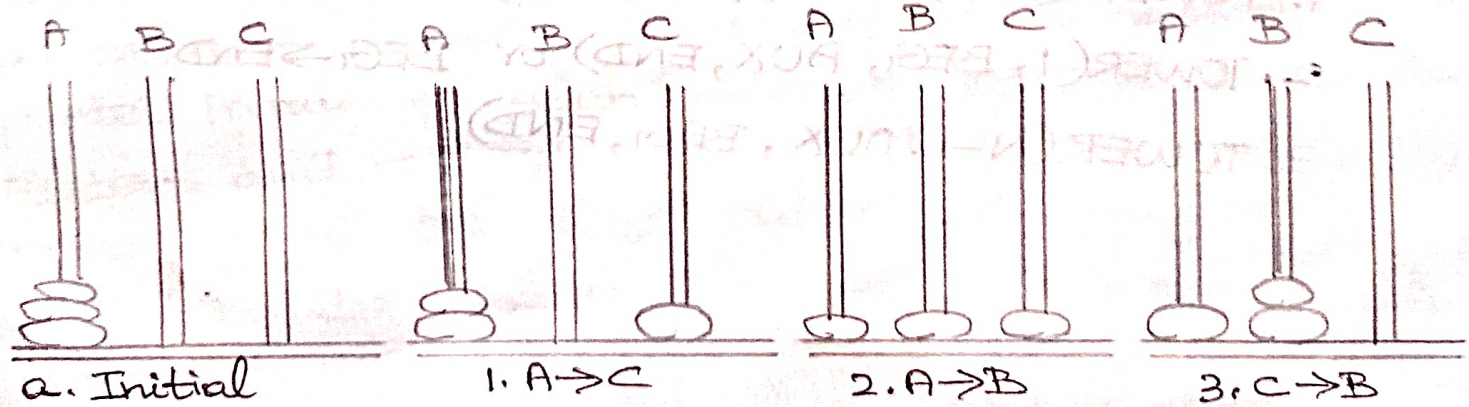
The object of the game is to move the disks from peg A to peg C using peg B as an auxiliary.

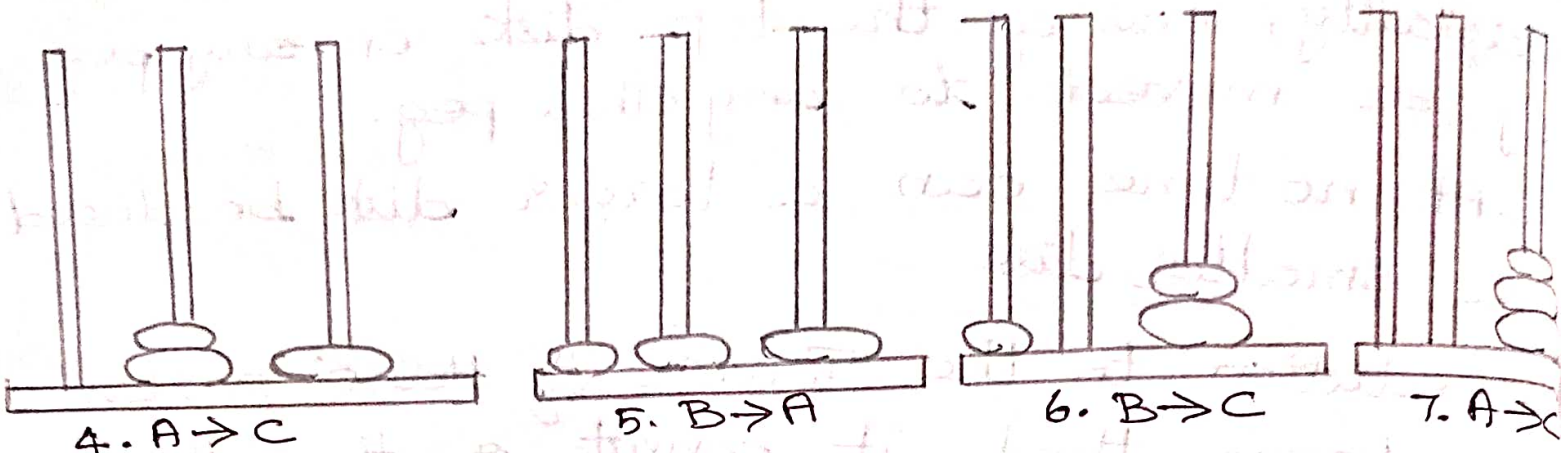
Rules of the game:-

1. Only one disk may be moved at a time. Specifically, one of the top disk on any peg may be moved to any other peg.
2. At no time can a larger disk be placed on a smaller disk.

The solution to the Towers of Hanoi problem n=3 observe that it consists of the following seven moves..

n=3: Move top disk from pegA to pegC
Move top disk from pegA to pegB.
Move top disk from pegC to pegB.
Move top disk from pegA to pegC.
Move top disk from pegB to pegA.
Move top disk from pegB to pegC
Move top disk from pegA to pegC.





In otherwords,

n=3 A→C, A→B, C→B, A→C, B→A, B→C, A→C.

General Notation:-

TOWER (N, BEG, AUX, END)

to denote a procedure which moves the top disks from the initial peg BEG to the final peg END using the peg AUX as an auxiliary.

For n=1

TOWER (1, BEG, AUX, END.)

The solution may be reduced to the solution of the following three subproblems:

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

PROCEDURE:-

TOWER(N, BEG, AUX, END)

This procedure gives a recursive solution to the Tower of Hanoi problem for N disks.

1. If N=1 then:
   (a) Write : BEG→END
   (b) Return
   [End of If structure.]
2. [Move N-1 disks from peg BEG to peg AUX.]
   Call TOWER(N-1, BEG, END, AUX).
3. Write : BEG→ END.
4. [Move N-1 disks from peg AUX to peg END.]
   Call TOWER(N-1, AUX, BEG, END).
5. Return.

Pointers; pointer Arrays:-

Let DATA be any array. A variable p is called a pointer if p "points" to an element in DATA i.e., if p contains the address of an element in DATA. Analogously, an array PTR is called a pointer Array if each element of PTR is a pointer. Pointers and arrays are used to facilitate the processing of the information in DATA.

Consider an organisation which divides its membership list into four groups, where each group

b. The data items in a record are indexed by attribute names, so there may not be a natural ordering of its elements.

QUEUE:-

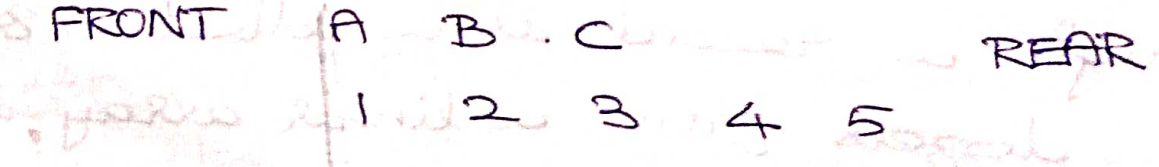
Queue is the linear structure of element in which deletion can take place only at one end called 'FRONT' and the insertion can take place at another end called 'REAR'.

The 'FRONT' and 'REAR' are used in describing the linear list only when it is implemented as a queue.

Queue is also called as FIFO list as 'FIRST IN FIRST OUT', since the element in the queue in which it enter a queue is the order in which they leave.

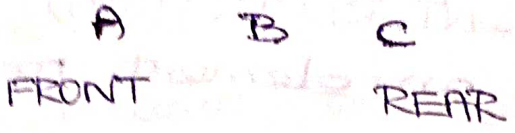
Eg: Queue occur in a time sharing the same priority which program forms a queue while waiting to be executed.

Representation of a Queue as an Array:-



FRONT=1, REAR=3.

Representation of a Queue as A linear list:-



The Queue can be represented in array & linear list formed.

Operation On Queue:-

The main operation on Queue are

- ADD Q
- DELETE Q

The other operation on Queue are

- Create Q
- FRONT[Q]
- ISEMPQ[Q]

CREATE[Q]: This operation 'creates an empty' Queue.

FRONT[Q]: This operation 'returns the front element' of the Queue.

ISEMTQ[Q]: Returns "True" if Queue is empty else "false".

MAJOR OPERATION ON QUEUE:-

ADDQ: This operation 'adds' the element to the rear of the Queue and return the new Queue.

PROCEDURE:-

QINSERT (QUEUE, N, FRONT, REAR, ITEM)

This procedure inserts an element ITEM into Queue.

1. [Queue already filled?]
   If FRONT=1 and REAR=N, or if FRONT=REAR+1
   Write: OVERFLOW, and Return.
2. [Find new value of REAR]
   If FRONT:=NULL, then: [Queue initially empty]
      set FRONT:=1 and REAR:=1
   Else if REAR=N, then:
      set REAR:=1
   Else:
      set REAR:=REAR+1
   [End of If structure].
3. set QUEUE[REAR]:=ITEM [This inserts new element]
4. Return.

Explanation of Variables:-

N: It is the maximum number of elements that a Queue can have

REAR: It is the end in which elements or item are insert.

Q: Name of Queue.

Explanation of Algorithm:-

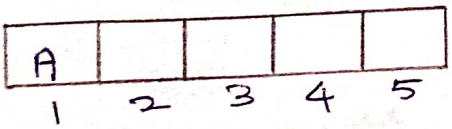
The procedure QINSERT insert an element into the Queue at first, it check if the Queue is full. If REAR=N, Queue is said to be full. hence no insertion can take place.

If Queue is not full then 'rear is incremented by 1' & insertion can take place at one end.

```
ADD(A,Q)
    if rear=5
    rear=rear+1
        =0+1
```

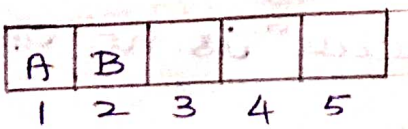
REAR=1

Q[1]=A



rear=1
front=0

```
ADD(B,Q)
    if rear=5
    rear=rear+1
        =1+1
    rear=2
    Q[2]=B
```



Rear=2
front=0

DELETE Q:-

This operation 'removes or deletes' the front element from the Queue.

PROCEDURE:-

QDELETE(QUEUE, N, FRONT, REAR, ITEM)

This procedure deletes an element from a queue & assigns it to variable ITEM.

1. [Queue already empty?]
   If FRONT:=NULL, then: Write: UNDERFLOW, and RETURN
2. set ITEM:= QUEUE[FRONT]
3. [find new value of FRONT]
   If FRONT=REAR, then: [Queue has only one element to start.]
   set FRONT:=NULL and REAR:=NULL
   Else if FRONT=N, then:
   set FRONT:=1
   Else
   set FRONT:=FRONT+1
   [End of If structure]
4. Return.

Explanation of Algorithm:-

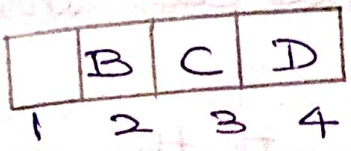
This procedure delete the item in the Queue. If check whether there is a element to be delete if Queue is no empty, then front is 'added by 1'.

A	B	C	D
1	2	3	4

Rear=3

Front=0

if front >= rear
front = front + 1
= 0 + 1
= 1



Rear = 3
Front = 1

The front element in the Queue is deleted using deleted operation.

CIRCULAR QUEUES:-

Circular Queue is another form of a linear queue in which the last position is connected to the first position of the list.

The circular queue is similar to linear queue has 2 ends, the front end & the rear end.

The rear end is where we insert elements & front end is where we delete elements.

You can traverse (move) in a circular queue in only one direction (i.e., from front to rear).

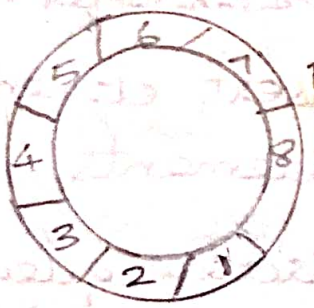
Initially the front & rear ends are at same positions (i.e., -1) when you insert elements the rear pointer moves one by one (where as the front ptr doesn't change) until the front end is reached.

If the next position of the rear is front, the queue is said to be fully occupied.

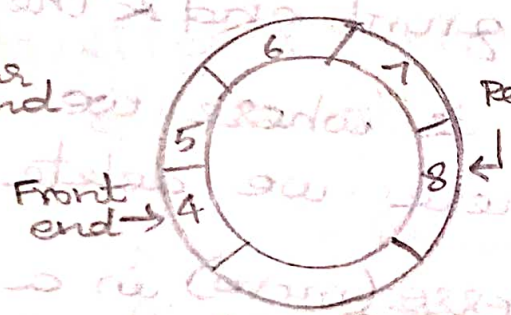
Beyond this you can't insert any data. But if you delete any data, you can insert the data accordingly.

When you delete the elements the front ptr moves one by one (where as the rear ptr doesn't change) until the rear pointer is reached.

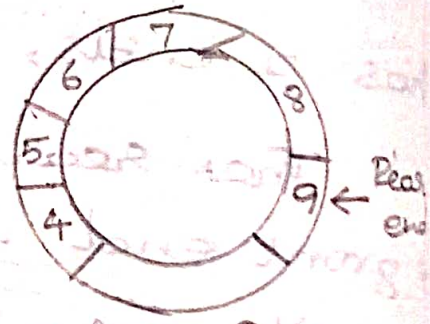
If the front pointer reaches the rear pointer, both their positions are initialised to -1, and the queue is said to be empty.



Representation of Circular Queue



Circular Queue representation After deletion



Circular Queue representation After Deletion

DEQUES:-

Deque (Double-ended queue) is another form of a queue in which insertions and deletions are made at both the front and rear ends

the queue. There are two variations of a deque namely,

- Input restricted deque
- Output restricted deque

The input restricted deque allows insertion at one end (it can be either front or rear) only. The output restricted deque allows deletion at one end (it can be either front or rear) only. The different types of deque are

1. Linear queue.
2. Circular queue.

Linear Deque:-

Linear deque is similar to a linear queue except the following conditions,

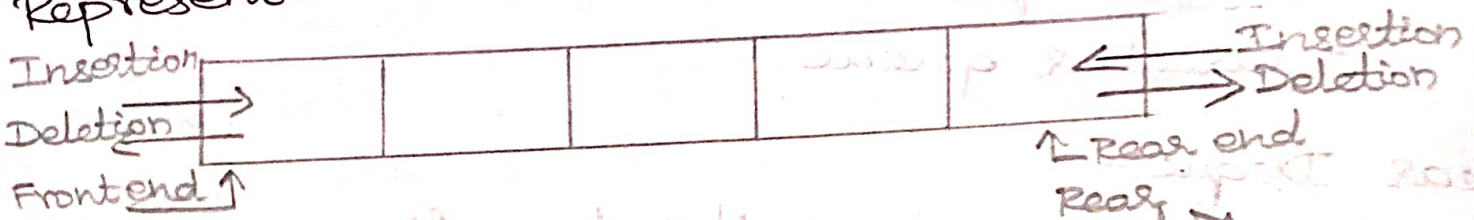
1. The insertions & deletions are made at both the front & rear ends of the queue.
2. If the front end is in the first position, you can't insert the data at front end.
3. If the rear end is in the last position, you can't insert at data at rear end.

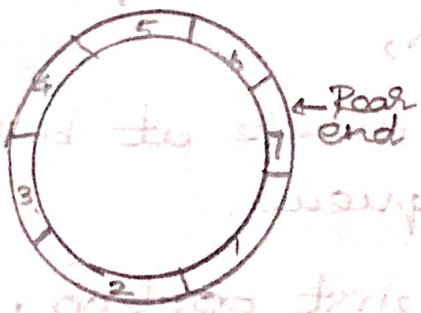
Circular Deque:-

Circular deque is similar to a circular queue except the following conditions.

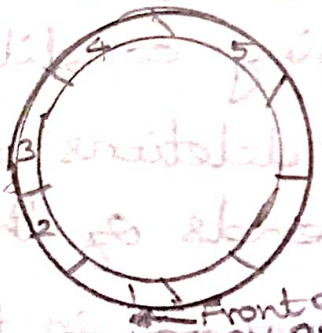
1. The insertion & deletions are made at both the front & rear ends of the deque.
2. Irrespective of the positions of the front and rear end you can insert and delete data.

Representation of a Deque:-

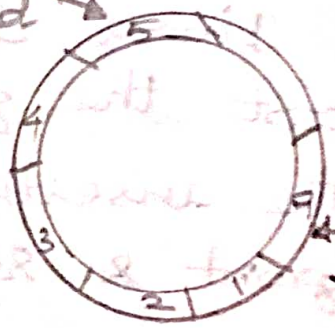




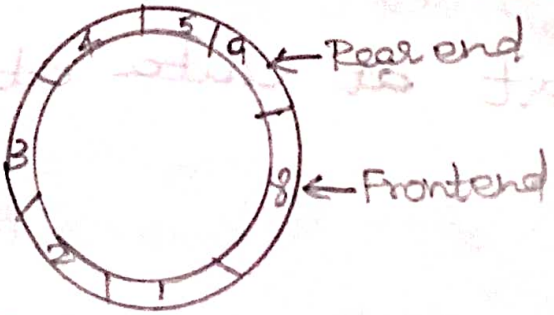
REPRESENTATION OF A CIRCULAR DEQUE



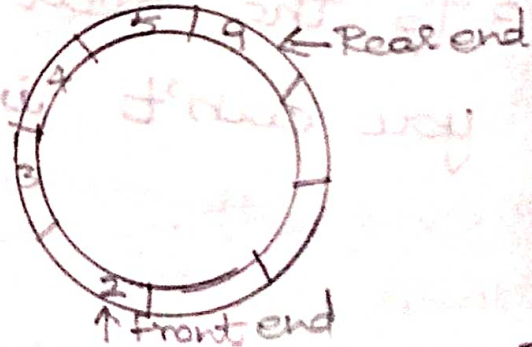
REPRESENTATION AFTER DELETION AT REAR END



REPRESENTATION AFTER INSERTION AT FRONT END



CIRCULAR DEQUE REPRESENTATION AFTER INSERTION AT REAR END



CIRCULAR DEQUE REPRESENTATION AFTER DELETION AT FRONT END

PRIORITY QUEUES:-

A priority queue is a collection of elements such that each element has been assigned a priority & 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.

A prototype of a priority queue is a time sharing system: prgs of high priority are processed first, and prgs with the same priority form a standard queue.

There are various ways of maintaining a priority queue in memory. They are one uses a one-way list and the other uses multiple queues.

The ease or dizziculty in adding elements to or deleting them from a priority queue clearly depends on the representation that one chooses.