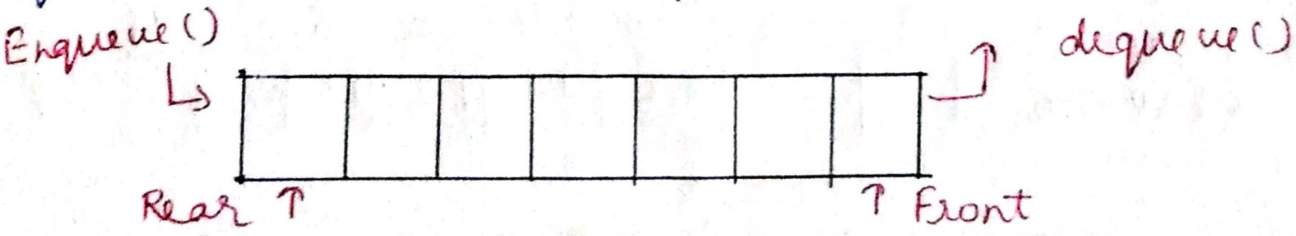
UNIT- II

Queue :-

* Queue is a linear data structure where the first element is inserted from one end called REAR and deleted from the other end called as FRONT.
* Front points to the beginning of the queue and Rear points to the end of the queue.
* Queue follows the FIFO (First-In-First Out) structure.
* According to its FIFO structure, element inserted first will also be removed first.
* This contrasts with stacks, which are last-in-first-out (LIFO).
* The process to add an element into queue is called Enqueue & the process of removal of an element from the queue is called Dequeue.



* A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first.

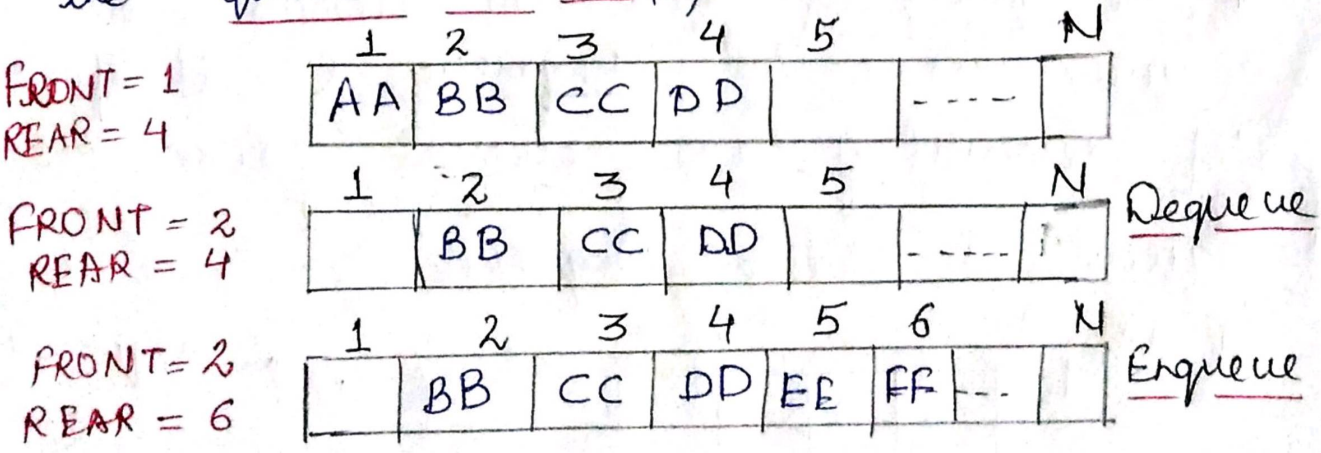
Representation of Queue :-

* Queues may be represented in the computer by means of one-way lists or linear arrays.
* Unless otherwise stated or implied, each of our queues will be maintained by a linear array QUEUE and two pointer variables :-

FRONT :- containing the location of the front element of the queue.

REAR :- containing the location of the rear element of the queue.

* The condition FRONT = NULL, indicates that the queue is empty.



* The above figure shows the way the array QUEUE will be stored in memory with N elements.
* Observe that whenever an element is deleted from the queue, the value of FRONT is increased by 1 i.e.,

FRONT = FRONT + 1

* Whenever an element is added to the queue, the value of REAR is increased by 1 i.e.,

REAR = REAR + 1

* this means that after N insertions, the rear element of the queue will occupy QUEUE[N]. This occurs even though the queue itself may not contain many elements.

Note :- Suppose we want to insert an element ITEM into a queue at the time queue does occupy last part of the array i.e., REAR = N.

* One way to do this is to simply move the entire queue to the beginning of the array, changing FRONT & REAR accordingly.
* The above procedure is very expensive. Thus we adopt the concept of circular queue.

i.e., when REAR = N then reset REAR = 1
then assign QUEUE [REAR] = ITEM
{for inserting ITEM}

Ily if FRONT = N, then reset FRONT = 1

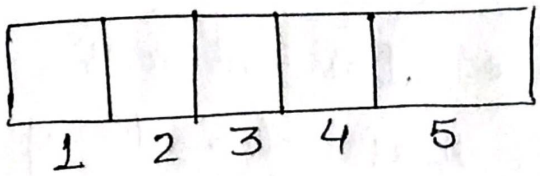
Suppose that our queue contains only one element i.e., FRONT = REAR ≠ NULL
& suppose that the element is deleted. Then we assign,

FRONT = NULL & REAR = NULL

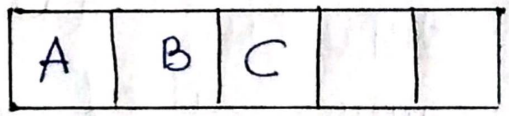
to indicate the empty list.

Example :-

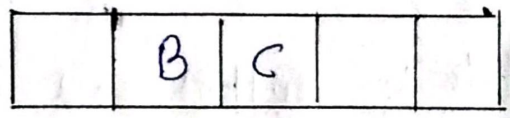
(a) Initially empty



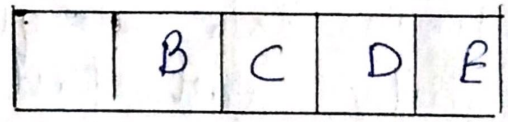
(b) A, B & C inserted



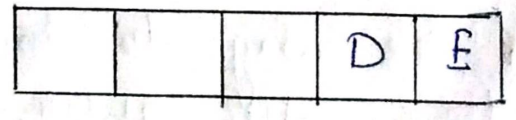
(c) A deleted



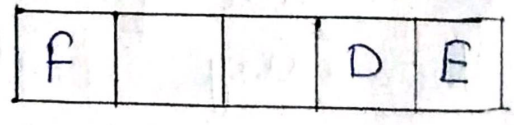
(d) D & E inserted



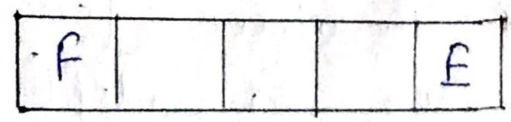
(e) B & C deleted



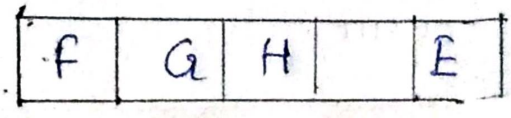
(f) F inserted



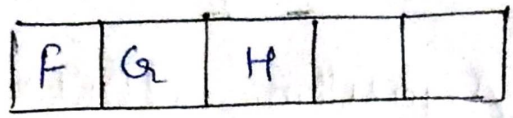
(g) D deleted



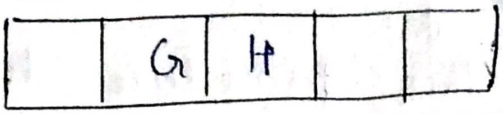
(h) G & H inserted



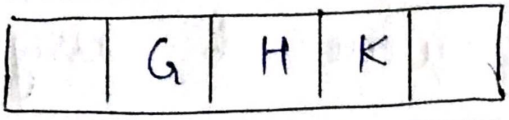
(i) E deleted — F = 1, R = 3



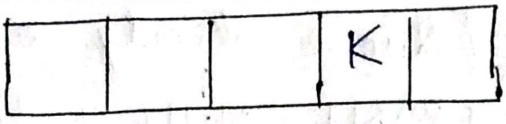
(j) f deleted — f = 2, R = 3



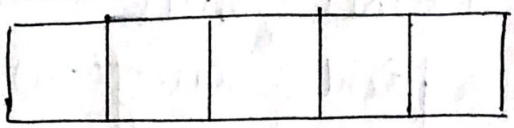
(k) K inserted — F = 2, R = 4



(l) G & H deleted — F = 4, R = 4



(m) K deleted, QUEUE empty — F = 0, R = 0



Algorithm 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
   then print "Overflow" & Exit.
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]
3. Set QUEUE [REAR] = ITEM [Inserts new element]
4. Return

Algorithm 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
print "queuflow" & Exit.

2. Set ITEM = QUEUE[FRONT]

3. if FRONT = REAR then [Queue has only one element]
Set FRONT = NULL & REAR = NULL

else if FRONT = N then
set FRONT = 1

else
set FRONT = FRONT + 1

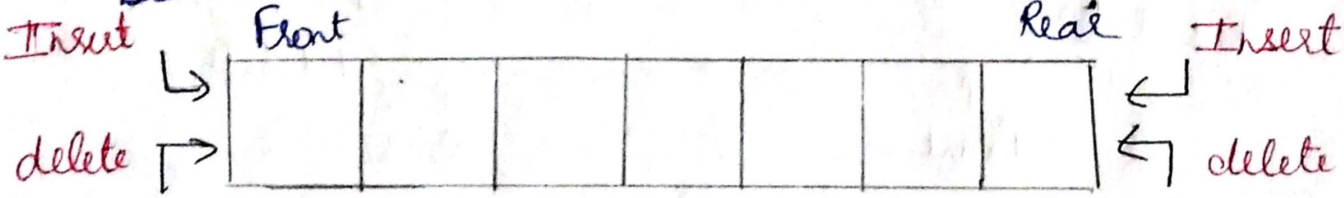
4. Return

Deque :-

* A deque, also known as double-ended queue is an ordered collection of items similar to the queue.

* We'll maintain deque by a circular array DEQUE with pointers ~~left~~ LEFT and RIGHT, which points to the two ends of the deque.

* Double ended queue is a more generalized form of queue data structure which allows insertion & removal of elements from both the ends.

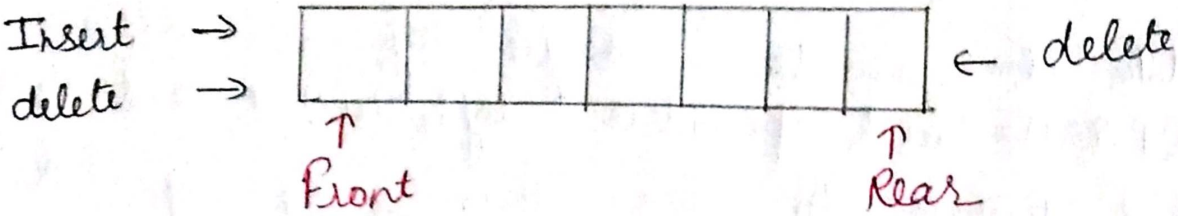


* Double Ended Queue can be represented in Two ways :-

1. Input Restricted Double Ended Queue
2. Output " " " "

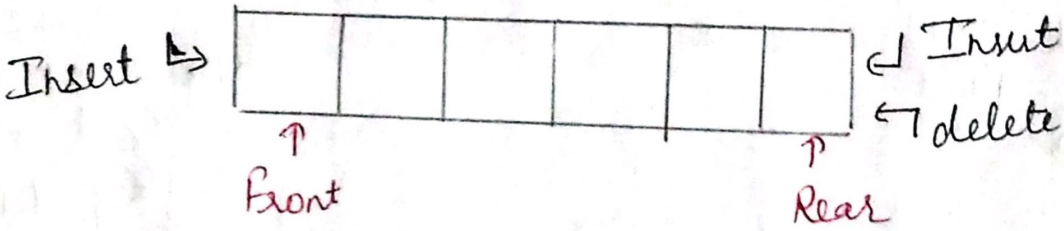
Input Restricted Double Ended Queue :-

→ The insertion operation is performed at ~~both~~ one the ends and deletion operation is performed at both the ends.



Output Restricted Double Ended Queue :-

→ The deletion operation is performed at only one end and insertion operation is performed at both the ends.



Priority Queue :-

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

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 are added to the queue.

* There are various ways of maintaining a priority queue in memory.

  ↳ One-way list

  ↳ Multiple Queues

One-way list representation of a Priority Queue

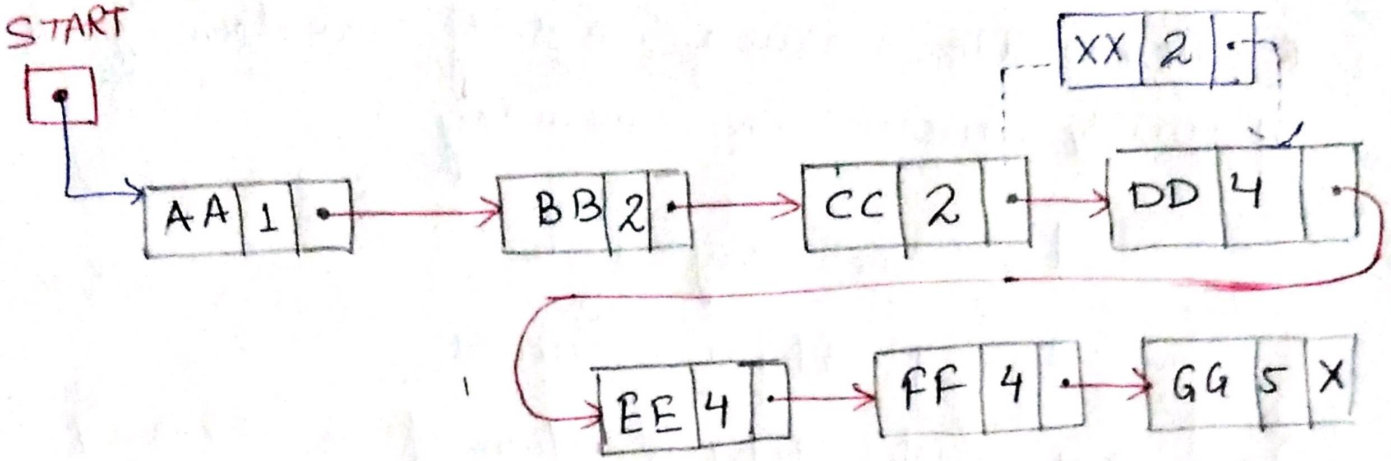
One way to maintain a priority queue in memory is by means of a one-way list, as follows:-

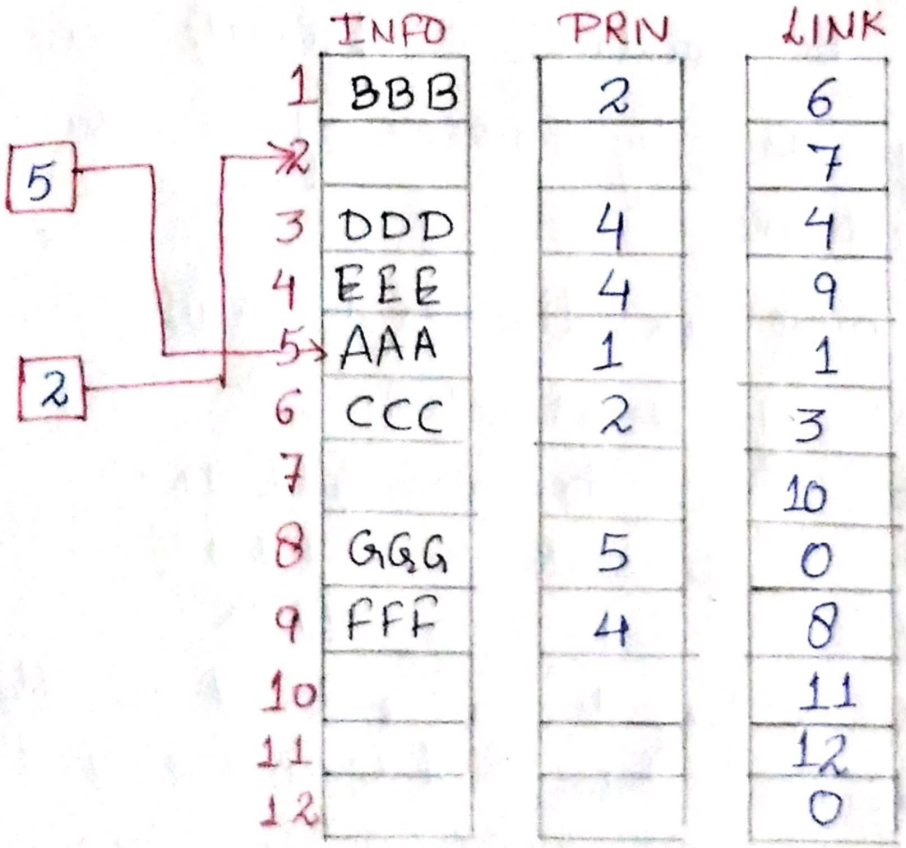
(a) Each node in the list will contain three items of information :-
  * an information field INFO
  * a priority no. PRN
  * a link no. LINK

(b) A node X precedes a node Y in the list
  (i) When X has higher priority than Y
  or (ii) When both have the same priority but X was added to the list before Y.

{Note :- Thus the order in the one-way list corresponds to the order of the priority queue.}

* Priority no. will operate the usual way, the lower the priority no., the higher the priority.





Q. Consider the above priority queue, suppose an item XXX with priority no. 2 is to be inserted into the queue:

* We traverse the list, comparing priority nos. Observe that DDD is the first element in the list whose priority exceeds that of XXX.
* Hence XXX is inserted in the list in front of DDD.

Note :- Observe that XXX comes after BBB & CCC, which have the same priority as XXX. Suppose now, an element is to be deleted from the queue, it will be AAA, then BBB → then CCC then XXX & soon.

Array Representation of a Priority Queue :-

* Another way to maintain a priority queue in memory is to use a separate queue for each level of priority (or for each priority no.)
* Each such queue will appear in its own circular array & must have its own pair of pointers, FRONT & REAR.
* In fact, each queue is allocated the same amount of space, a 2-D array QUEUE can be used instead of the linear arrays.

* The following figure indicates this representation for the priority queue. Observe that FRONT[K] and REAR[K] contain, respectively, the front & rear elements of row K of QUEUE, the row that maintains the queue of elements with priority no K.

	FRONT
1	2
2	1
3	0
4	5
5	4

	REAR
1	2
2	3
3	0
4	1
5	4

	1	2	3	4	5	6
1		AAA				
2	BBB	CCC	XXX			
3						
4	FFF				DDD	EEE
5				GGG		

Application of Queues : Round Robin Algo.

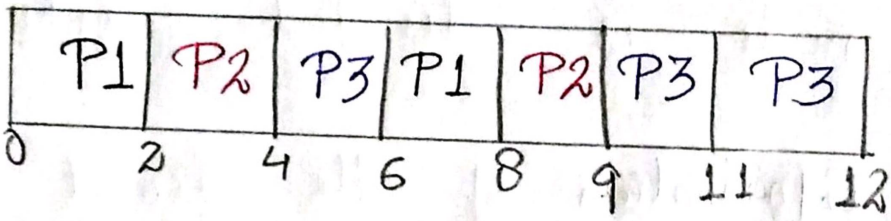
* A popular use of queue data structure is the scheduling problem in the operating system.

* Round-Robin is one of the simplest scheduling algorithm for processes in an operating system, which assigns time slices to each process in equal portions and in order, handling all processes without priority.

Example :- consider following three process

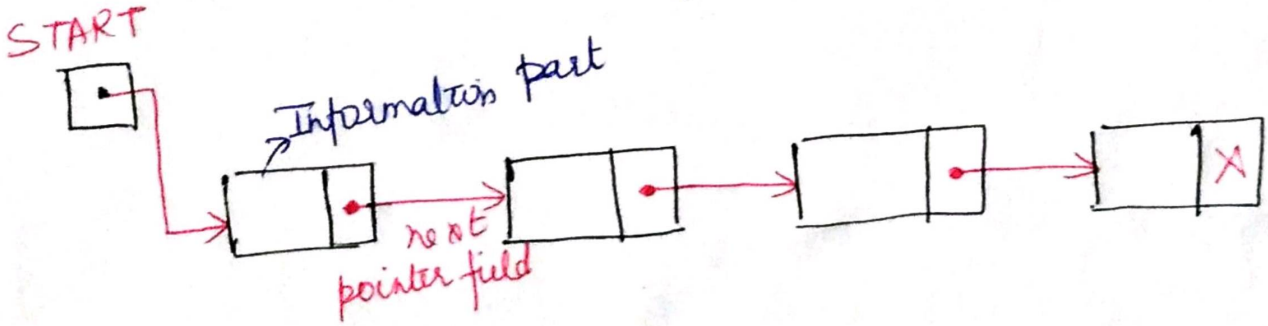
Process Queue	Burst time
P1	4
P2	3
P3	5

Time Slice = 2



Linked lists

* A linked list or one-way list is a linear collection of data elements called nodes, where the linear order is given by means of pointers.
* Each node is divided into two parts :-
  1) Information of the element
  2) link field or nextpointer, contains the address of the next node in the list



* The linked list contains a list pointer variable START- which contains the address of the first node in the list.
* The pointer to the last node contains a special value, called the null pointer (which is any invalid address)

Representation of linked List in Memory :-

* Let ~~List~~ LIST be a linked list. Then LIST will be maintained in memory as follows :-

  * First of all LIST requires two linear arrays - INFO and LINK, such that INFO[K] and LINK[K] contain respectively, the information part & next pointer field of a node LIST.

  * LIST also require a variable name such as START - which contains the location of the beginning of the list.

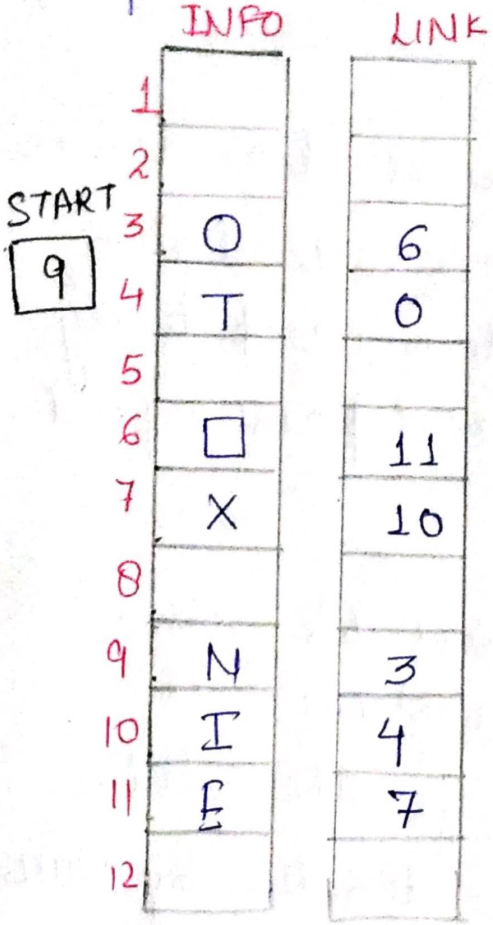
  * A next pointer sentinel - denoted by NULL, which indicates the end of the list.

Note :- We'll choose '0' to indicate NULL.

* The following examples of linked list indicate that the nodes of a list need not occupy adjacent elements in the arrays INFO and LINK.

* More than one list may be maintained in the same linear arrays INFO & LINK.

* However, each list must have its own pointer variable giving location of its first node.



START = 9, so INFO[9] = N is the first character.

LINK[9] = 3, INFO[3] = O

LINK[3] = 6, INFO[6] = □ (blank)

LINK[6] = 11, INFO[11] = E

LINK[11] = 7, INFO[7] = X

LINK[7] = 10, INFO[10] = I

LINK[10] = 4, INFO[4] = T

LINK[4] = 0, the list ended.

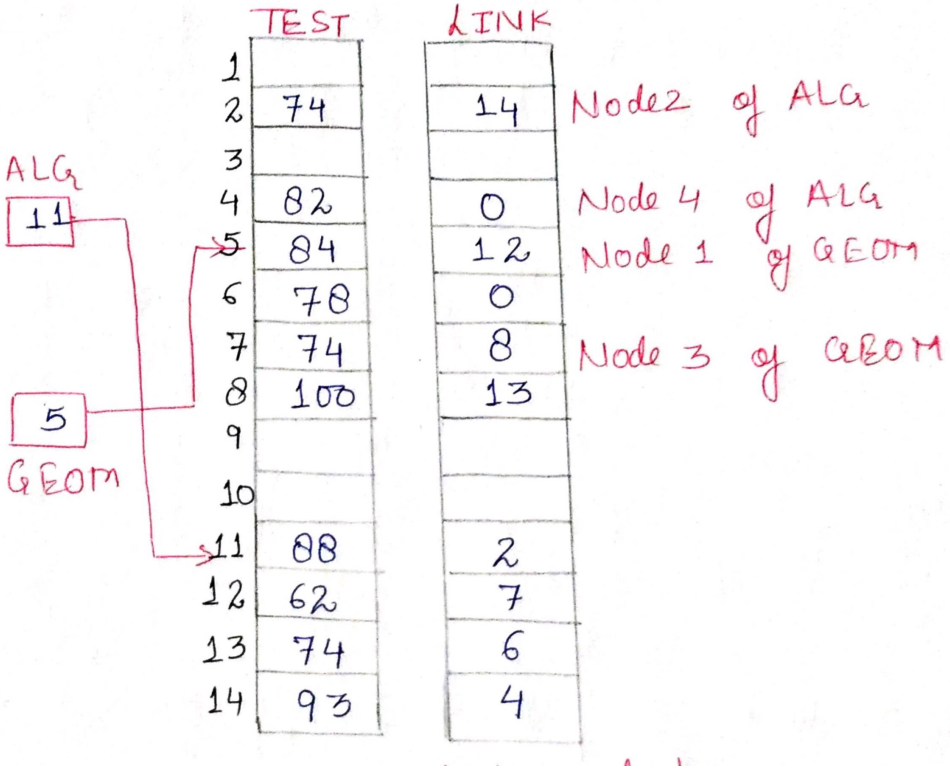
In other words, NO EXIT is the character string.

Example :-

Following figure shows how two lists of test score, ALG and GEOM, may be maintained in memory where the nodes of both lists are stored in the same linear array list & link.

* ALG contains 11, the location of its first node, and GEOM contains 5, the location of its first node.

ALG :- 88, 74, 93, 82
GEOM :- 84, 62, 74, 100, 74, 78



Creating a singly linked list

```
#include <stdio.h>
#include <conio.h>
#include <alloc.h>

void creat()
{
  char ch;
    do
    {
    struct node *new_node, *current;
new_node = (struct node *) malloc (sizeof (struct node));
printf ("Enter the data:");
scanf ("%d", &new_node->data);
new_node->next = NULL;
if (start == NULL) {
  start = new_node;
  current = new_node;
  }
else
  {
  current->next = new_node;
  current = new_node;
  }
printf ("Do you want to create another");
  ch= getche();
  }
while (ch!='n');
}
```

Step 1 :- include alloc.h

* We'll be allocating memory using Dynamic Memory Allocation f^n.
* All those functions are included in alloc.h.

Step 2 :- define node structure

```
* struct node {
        int data ;
        Struct node *next ;
  } * start = NULL;
```

Step 3 :- Create node using Dynamic memory allocation:

* We don't have prior knowledge about no. of nodes. So we are calling malloc function to create node at run time.

```
new_node = (struct node *) malloc (sizeof (struct node));
```

Step 4 :- Fill information in newly created node:

* Whenever we create a new node, make its next field NULL.

```
new_node -> next = NULL
```

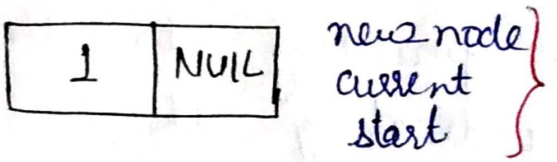
Step 5 :- Creating very first node:

* If node created in the above step is very first node then we need to assign it as starting node.

Step 6 :- Creating very First Node.

If the node created in the above step is very first node then we need to assign it as starting node.

Thus first three nodes have names :-



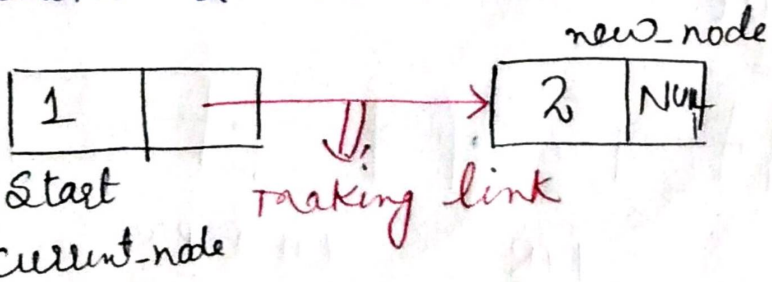
Step 7 :- Creating Second or n^{th} node :-

→ Lets assume, we have 1 node already created i.e., we have first node.

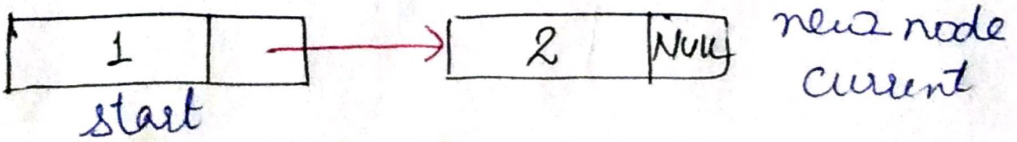
→ Now we have called creat() function again.

(iii) else
{
current → next = new_node;
current = new_node;
}

(i) first of all we'll create link bet^n current & new node



(ii) Now, move current pointer to next_node



Different operations on Single linked list

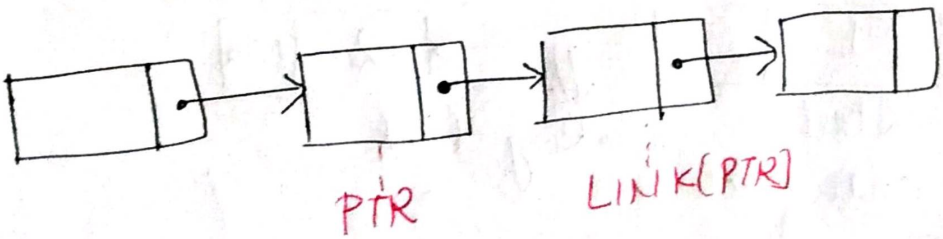
* Let LIST be a linked list in memory stored in linear arrays INFO and LINK.

Traversing a Linked List :-

* Let START pointing to the first element and NULL indicating the end of list.
* Suppose we want to traverse LIST in order to process each node exactly once.
* This traversing algo. uses a pointer variable PTR which points to the node that is currently being processed.
* Accordingly LINK[PTR] points to the next node to be processed. Thus the assignment

$$PTR = LINK[PTR]$$

moves the pointer to the next node in the list.



TRAVERSE_LIST( )

1. Set PTR = START [Initialize pointer PTR]
2. Repeat steps ③ and ④ while PTR ≠ NULL
3. Print INFO[PTR] [Process Node]
4. Set PTR = LINK[PTR] [PTR now points to next node.]
5. Exit

Searching A linked List :

* Let LIST be a linked list in memory.
* If ITEM is actually a key value & we are searching in LIST (assuming ITEM can appear only once in the LIST)

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

*LIST is a linked list in memory. This algo. finds the location LOC of the node where ITEM first appears in LIST or sets LOC=NULL

1) Set PTR = START
2) Repeat Step 3 while PTR ≠ NULL
3) If ITEM = INFO[PTR] then
   LOC = PTR & Exit

Else

Set PTR = LINK[PTR]

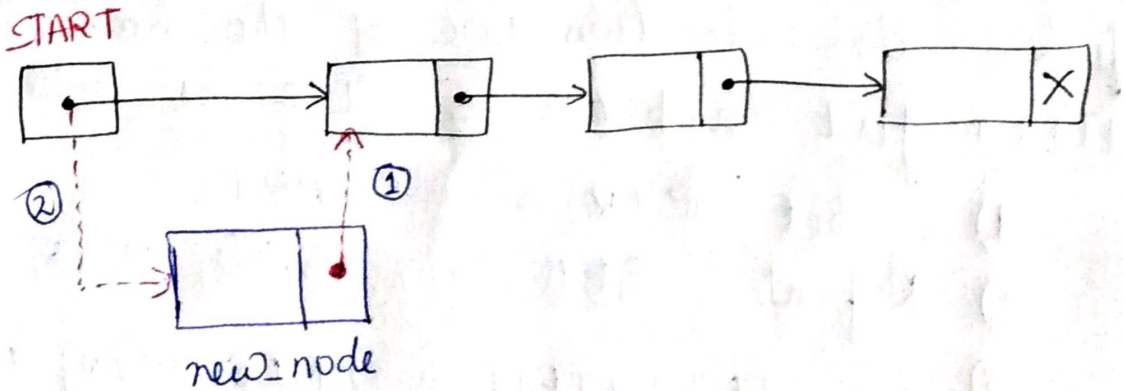
[End of Step 2 loop]

4. Set LOC = NULL [Search is Unsuccessful]
5. Exit.

Insertion into a Linked List :-

* Inserting nodes into linked list come up in various situations. We'll discuss three of them.
  (1) Inserting a node at the beginning
  (2) Inserting a node at the end
  (3) Inserting a node at specified locations.

(1) Inserting at the Beginning of a List :-



Algo. INSFIRST

Step 1 : [Check for free space]

If new_node = NULL then Print "Overflow" and Exit.

Step 2 :- [allocate free space]

else new_node = create new node

Step 3 :- [Read info part of new node]

data [new_node] = value

Step 4 :- [link address of currently created node with the address of start is pointing]

next [new_node] = start

Step 5 :- [Now, assign address of newly created node to the start]

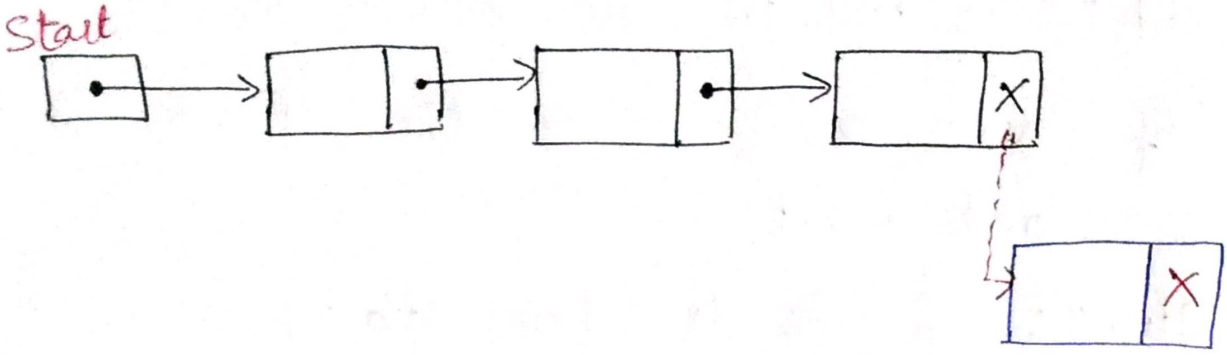
start = new_node

Step 6 :- Exit

Note :-

```
void insertion (struct link * node) {
        node = start.next ;
        previous = & start;
new_node = (struct link *) malloc (sizeof (struct link));
  new_node -> next = node ;
  previous -> next = new_node;
}
```

(2) Inserting at the end in the list :-



Step 1 :- [Check for free space]

if new_node = NULL then Print "Overflow"

& Exit

Step 2 :- [allocate free space]

Else

new_node = create new node.

Step 3 :- [Read value of info part of new node]

data [new_node] = value

Step 4 :- [Move the pointer to end of the list.]

Repeat while node ≠ NULL

node = next [node]

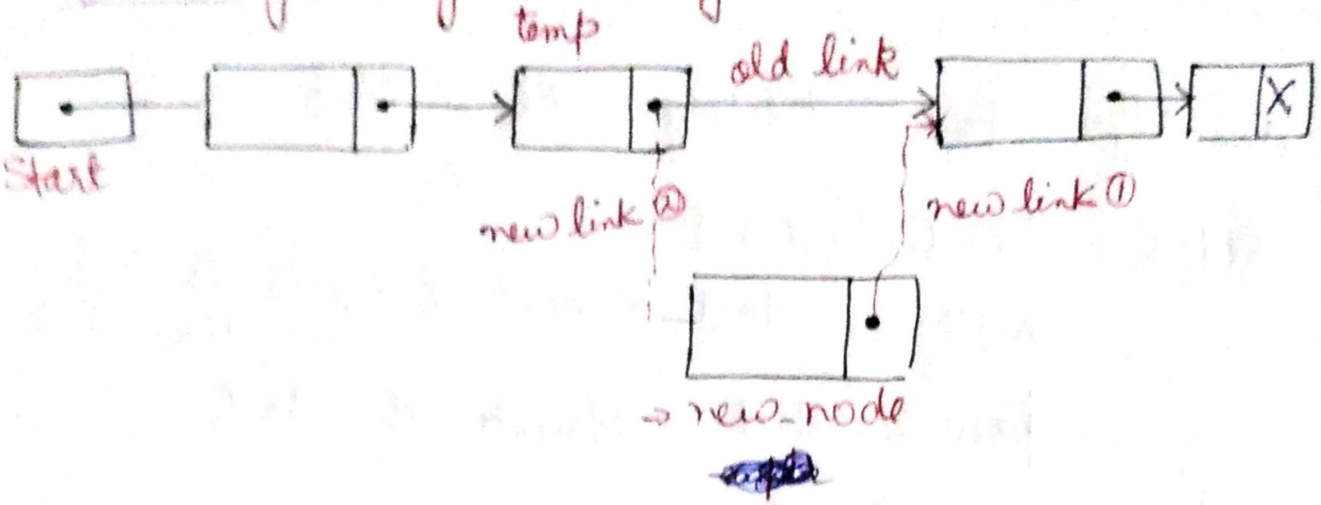
Step 5 :-

next [node] = new node

next [new node] = NULL

Step 6 :- Exit

(2) Inserting after a given node in



Step 1: [check for free space]
if new_node = NULL then Print "Overflow" and Exit

Step 2: [allocate free space]
Else new_node = create a new node.

Step 3: [Read value of info part of new node]
data [new_node] = value

Step 4: [Move the pointer to desired location]
temp = start;
for (i=0; i < loc; i++) // loc is the desired location
temp = temp → next;

Step 5: new_node → next = temp → next
temp → next = new_node;

Step 6: Exit

Deletion from the linked list

(1) Algo. for deleting the first node

Step 1 :- [Initialization]

node = start → next // points to first node in the list

prev = start // points at start

Step 2 :- [Perform deletion]

if node = NULL then Print "Underflow" & Exit.

else

prev → next = node → next;

Step 3 :- Free the memory space associated with node.

Step 4 :- Exit

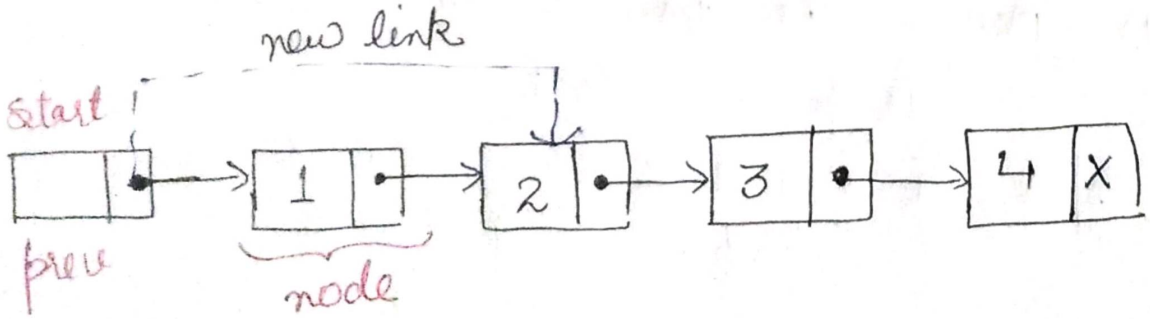


fig :- deleting first node

② Algo. for deleting last node :-

Step ① :- [Initialization]

node = start → next ;
prev = start ;

Step ② :- if node = NULL ; output "Underflow" & Exit.

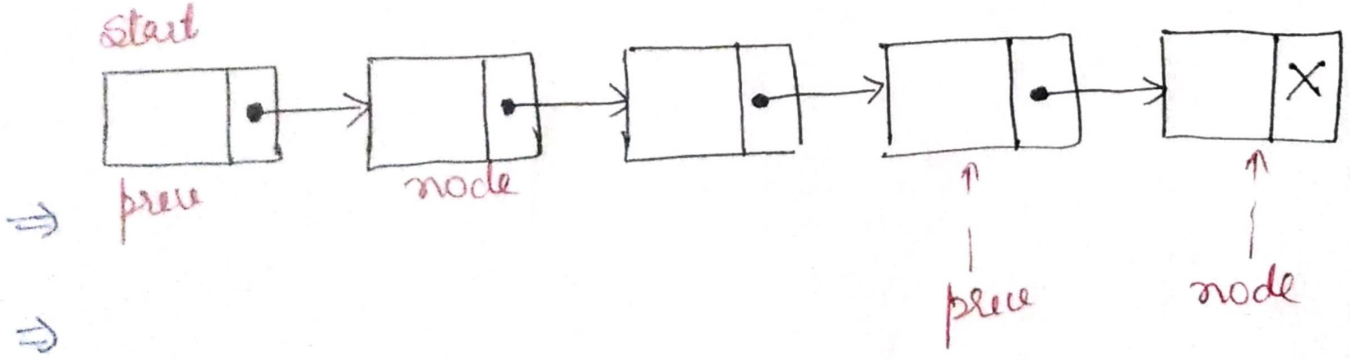
Step ③ :- [Reaching to the last node.]

Repeat while node ≠ NULL
node = node → next ;
prev = prev → next ;

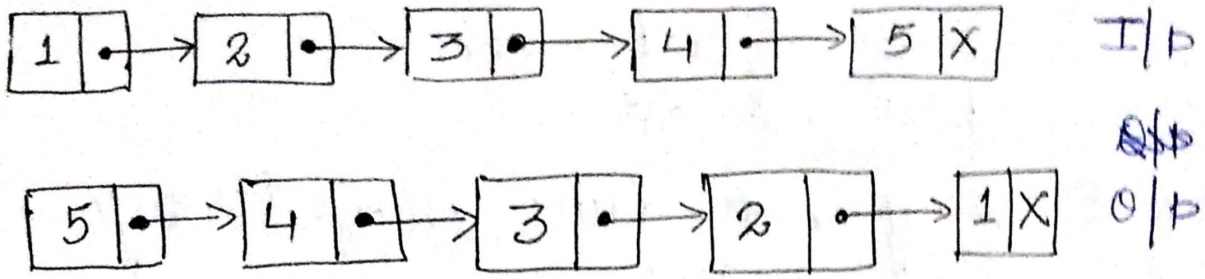
Step ④ : prev → next = node → next // or NULL

Step ⑤ :- Free the memory space associated with node.

Step ⑥ :- Exit

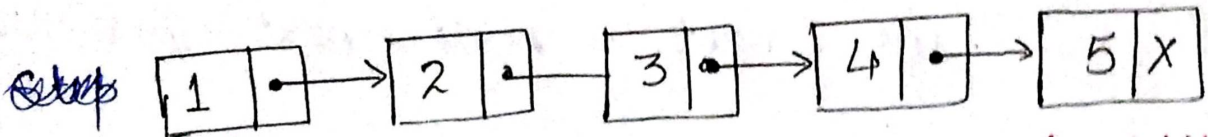


Reversing a Single Linked List :-



Step 1 :- Initialize three pointers;

prev = NULL
current = start
next = NULL



current start

prev = NULL, next = NULL

Step 2 :- Traverse / iterate through the linked list ;

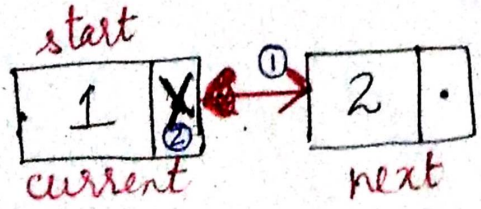
① next = current → next // store address of next node

② current → next = prev // Reversing the link

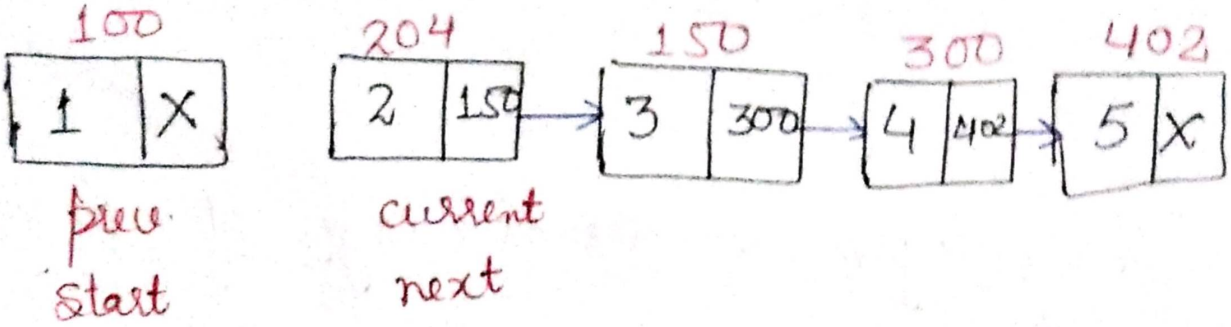
③ prev = current
④ current = next } moving prev & current one step forward

← Pass 1

after ① & ②

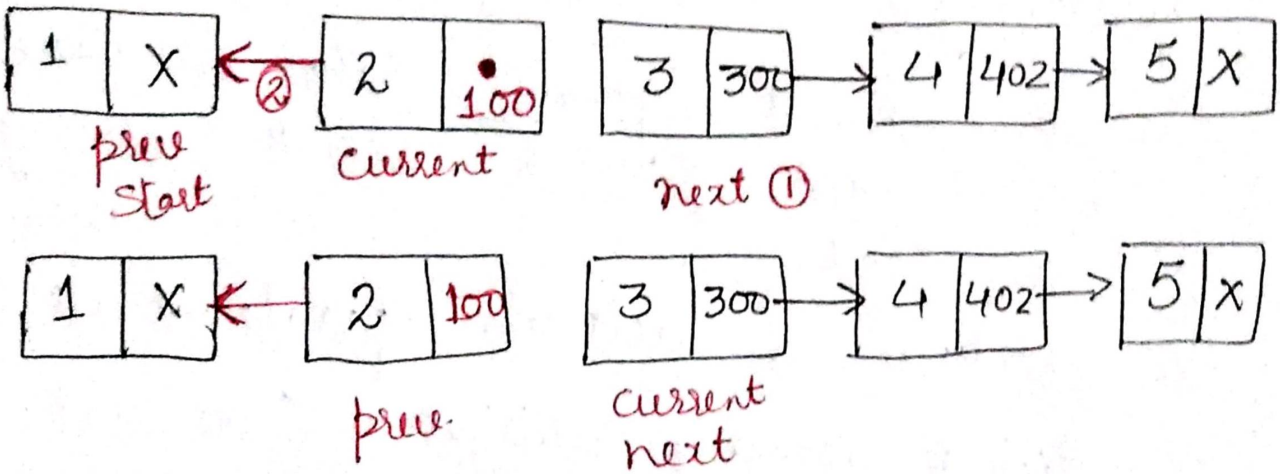


after ③ & ④



Pass ② :

①&②




```
/* C function to reverse the linked list
Void reverse()
{
Node * current = start;          // Initialization
Node * prev = NULL, * next = NULL;  // ,,
while (current != NULL) {
        next = current -> next;     // Store next
        current -> next = prev;     // Reverse current node.
        prev = current;   } move pointers one
        current = next;   } position ahead.
}
```

```
    head = prev;
}
```

Advantages & disadvantages of single Linked List :-

Advantages :-

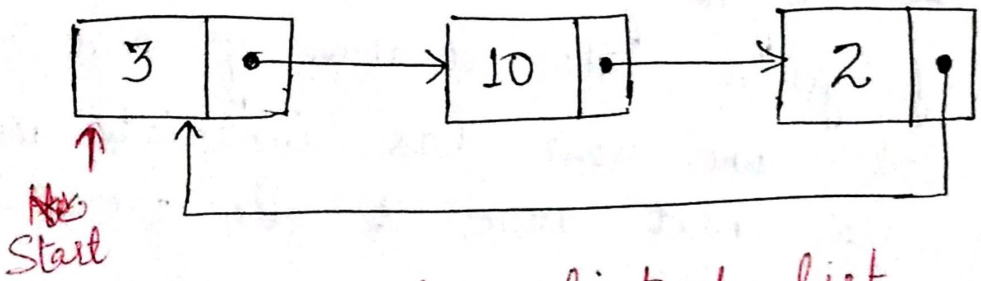
1. It is a dynamic data structure that is, it can grow & shrink during run-time.
2. Insertion & deletion operations are easier.
3. Efficient memory utilization (no need to pre allocate memory)
4. Linear data structures such as Stack, Queue can be easily implemented using linked list.

Disadvantages :-

1. Reverse traversing is very difficult in case of singly linked list.
2. No random access in Linked List, we have to access each node sequentially.

Circular Linked List

* A circular linked list is a variation of a normal linked list.
* In circular singly linked list, the last node of the list contains a pointer to the first node of the list.



circular linked list

Implementing circular Linked List :

* This implementation is very similar to single linked list with the only difference that the last node will have it's next point to the Head / Start of the list.

Double linked list :-

* Doubly linked list / two-way list is a variation of linked list in which traversing can be done in both directions. i.e.
* in the usual forward direction from the beginning of the list to the end, or in the backward direction from the end of the list to the beginning.
* Thus if given the location of a node N in the list, one now has immediate access to both the next node & the preceding node in the list.

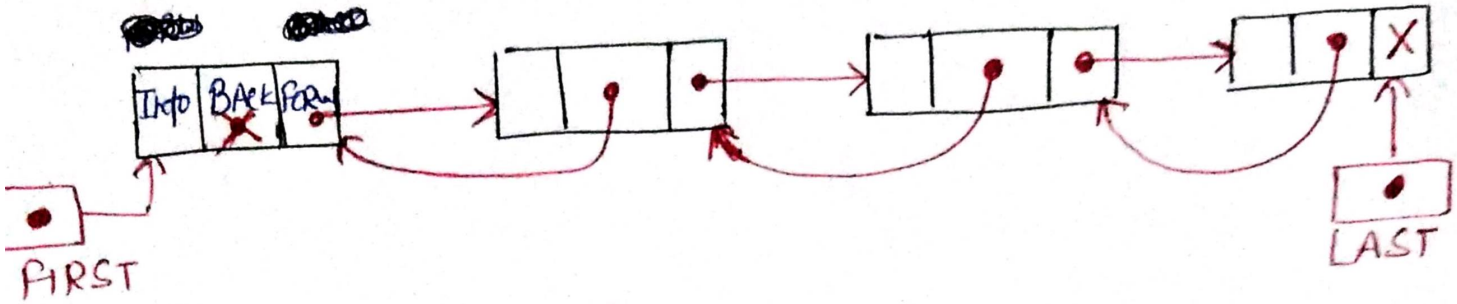
* A double linked list / two way list is a linear collection of data elements, called nodes, where each node is divided into three parts :

(1) An info. field INFO which contains the data of N.
(2) A pointer field FORW which contains the location of next node in the list.
(3) A pointer field BACK which contains the location of the preceding node in the list.

The list also require two list pointer variables :

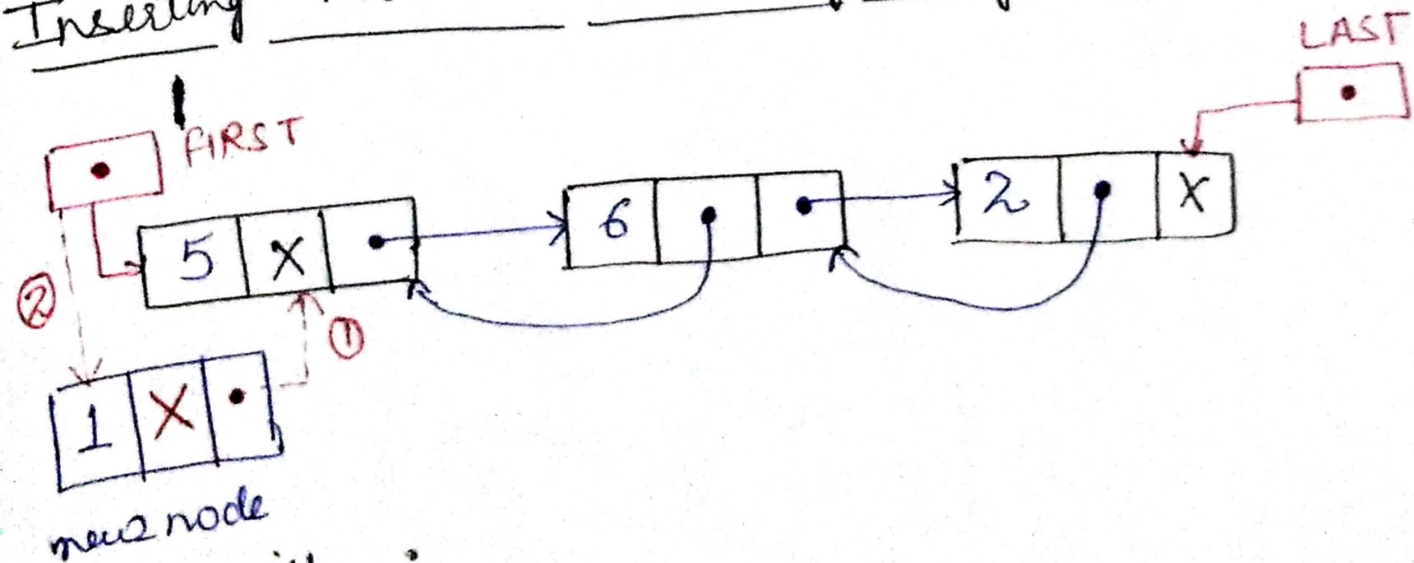
FIRST : which points to the first node in the list

LAST :- which points to the last node in the list




```
Struct node {
    int info;          // data
struct node *back;     // A reference to previous node.
struct node *forw;     // A reference to next node.
    }
```

Inserting node in the beginning :-



Algorithm :-

① Allocate the space for the new node in the memory.

new_node = (struct node *) malloc (sizeof (struct node));