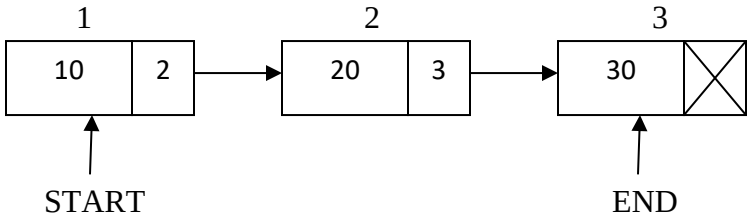
UNIT- III

Linked Lists – I: Single Linked List and Chains, Representing Chains in C++, defining a Node in C++- Designing a Chain Class in C++- Pointer manipulation in C++- Chain Manipulation Operations, The Template Class Chain, Implementing Chains with Templates- Chain Iterators - Chain Operations- Reusing a Class, Circular Lists, Available Space Lists, Linked Stacks and Queues, Polynomials.

***

LINKED LIST

"A linked list is a linear collection of elements called NODES and the order of the elements is given by means of LINKS / POINTERS". The diagrammatic representation of a linked list is as follows:



In array implementation of data structures, elements are organized in sequentially. For this, the size of the array must be specified at the beginning of the list and implementation can be done only on limited number of elements. Whereas, linked list is a dynamic implementation and can be capable to implement on large number of elements. In linked list, nodes are created with the concept of self-referential structure.

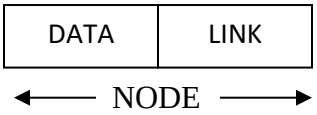
TYPES OF LINKED LISTS

Depending on the number of parts of a node and the way of establishing links in between the nodes, linked list can be classified into different categories as:

1) Single linked list
2) Double linked list
3) Circular linked list
4) Header linked list

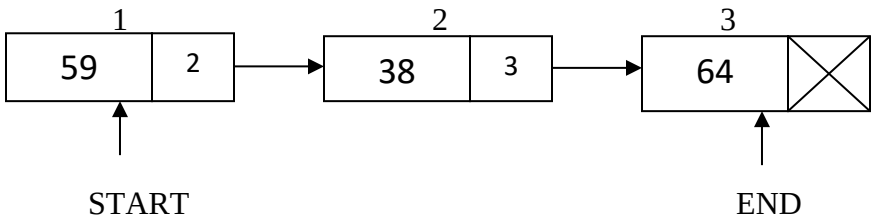
1. SINGLE LINKED LIST

In single linked list, each node is divided into two parts as INFO/DATA part and LINK part.



Where, the first part DATA field consist information part of the element and the second part LINK field consist address of the next node in the list.

Example:



Here, START and END are two pointer variables that points to beginning and ending nods of the list. The LINK field of the last node filled with a special pointer called NULL pointer.

The nodes are ordered from left to right with each node linking to the next, and link part of the last node consist NULL pointer, hence this structure is also called **Chain.**

Abstract Data type for the Chain (Single linked list) can be shown as:

AbstractDataType SList
{

Instances: Finite collection of zero or more elements linked by pointers.

Operations:

creation() : Create a single linked list with specified number of elements.
display() : Display elements of the single linked list.
insertion(): Insert a new element at the specified location.
deletion() : Remove an element from the single linked list.
count() : Returns number of elements of the single linked list.
search() : Search for the existing of a particular element.

}

To implement these operations, create a template class format as:

```
template<class T>
class SList
{
        typedef struct List
        {
                T DATA;
                struct List *LINK;
        }NODE;
        NODE *START,*END;
        public: SList()
                {
                        START=END=NULL;
                }
                void create();
                void display();
                void Finsertion(T);
                void Rinsertion(T);
                void Anyinsertion(T,int);
                T Fdeletion();
                T Rdeletion();
                T Anydeletion();
                int count();
                int search(T);
};
```

Basic operations performed on the single linked list are:

i) Creating a list
ii) Traversing the list
iii) Insertion of a node into the list
iv) Deletion of a node from the list
v) Counting number of elements
vi) Searching an element etc.,

In single linked list, nodes are created using self-referential structure as:

```
typedef struct list
{
        int DATA;
        struct list *LINK;
```

}NODE;

Now, create new nodes with the format as: NODE *NEW;

i) Creating a list

Creating a list refers to the process of creating nodes of the list and arranges links in between the nodes of the list.

Initially no elements are available in the list. At this stage, set two pointer variables START and END to NULL pointer as:

NODE *START = NULL, *END = NULL;

Algorithm creation (): This procedure creates a single liked list with the specified number of elements.

```
Step 1:     Repeat WHILE TRUE
                READ an element as x
                IF x = -999 THEN
                        RETURN
                ELSE
                        Allocate memory for a NEW node
                        DATA(NEW) ← x
                        LINK(NEW) ← NULL
                        IF START = NULL THEN
                                START ← END ← NEW
                        ELSE
                                LINK(END) ← NEW
                                END ← NEW
                        ENDIF
                ENDIF
            EndRepeat
```

ii) Traversing the list

Traversing the list refers to the process of visiting every node of the list exactly once from the first node to the last node of the list. To display the elements of the single linked follow the procedure as:

Step 1 - Check whether list is **Empty** or not.

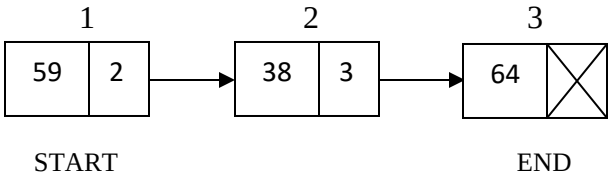
Step 2 - If it is **Empty** then, display **'List is Empty'** and terminate the function.

Step 3 - If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **start.** Keep displaying **temp → data** and then shift temp to next location until temp is not equal to NULL pointer.

Algorithm display(): This procedure is used to display the elements of the single linked list from the first node to the last node.

```
Step 1:     IF START = NULL THEN
                WRITE 'Single Linked List Empty'
            ELSE
                TEMP ← START
                Repeat WHILE    TEMP ≠ NULL
                        WRITE    DATA(TEMP)
                        TEMP ← LINK(TEMP)
                EndRepeat
            ENDIF
Step 2:     RETURN
```

Example: Assume initial status of the list as:



Display(): Single linked list elements are: 59 38 64

iii) Insertion of a node into the list

The process of inserting a node into the single linked list falls into three cases as:

- Front insertion
- Rear insertion
- Any position insertion

Case 1: Front Insertion: In this case, a new node is inserted at front position of the single linked list. For this, follow the procedure as:

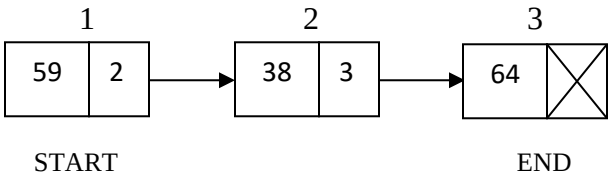
Step 1 - Create a **NEW Node** with given value.

Step 2 - Check whether list is **Empty** or not. If it is **Empty** then, set START and END pointers to NEW node. If it is **Not Empty** then, set LINK(NEW) with START pointer and move START pointer to NEW node.

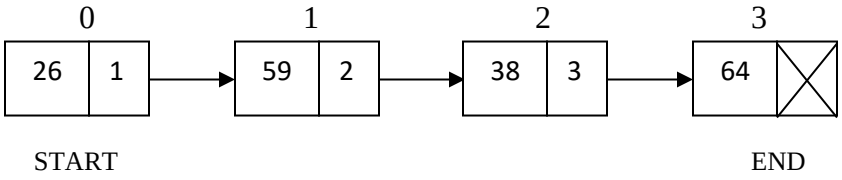
Algorithm Finsertion(x): This procedure inserts an element x at front end of the list.

```
Step 1:     Allocate memory for a NEW node
            DATA(NEW) ← x
            LINK(NEW) ← NULL

Step 2:     IF START = NULL THEN
                START ← END ← NEW
            ELSE
                LINK(NEW) ← START
                START ← NEW
            ENDIF
Step 3:     RETURN
```

Example: Assume initial status of the list as:



Finsertion (26):



Case 2: Rear Insertion: In this case, a new node is inserted at rear position of the single linked list. For this, follow the procedure as:

Step 1 - Create a **NEW Node** with given value.

Step 2 - Check whether list is **Empty** or not. If it is **Empty** then, set START and END pointers to NEW node. If it is **Not Empty** then, set LINK(END) with NEW pointer and move END pointer to NEW node.

Algorithm Rinsertion(x): This procedure inserts an element x at rear end of the list.

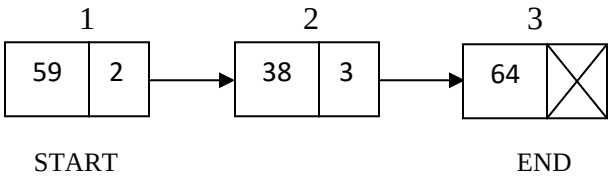
```
Step 1:     Allocate memory for a NEW node
```

DATA(NEW) ← x
LINK(NEW) ← NULL

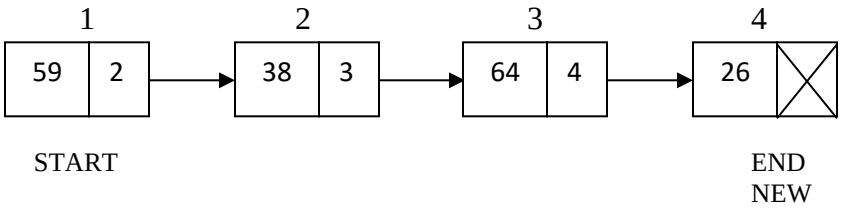
Step 2: IF END = NULL THEN
START ← END ← NEW
ELSE
LINK(END) ← NEW
END ← NEW
ENDIF

Step 3: RETURN

Example: Assume initial status of the list as:



Rinsertion (26):



Case 3: Any Position Insertion: In this case, a new node is inserted at a specified position of the single linked list. For this, follow the procedure as:

Step 1 - Create a **NEW Node** with given value.

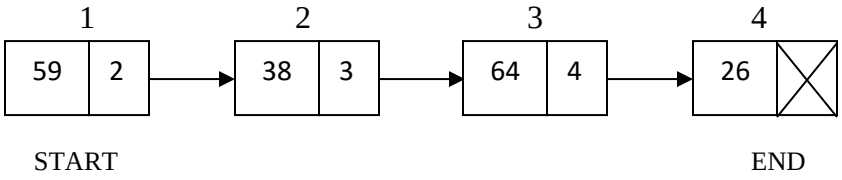
Step 2 – Set a temporary variable PTR points to START and move the PTR variable up to the specified location of the list along with another temporary variable TEMP. When the specified location achieved, establish links in between NEW, PTR and TEMP variables.

Algorithm Anyinsertion(x, pos): This procedure inserts an element x at the specified position pos of the list.

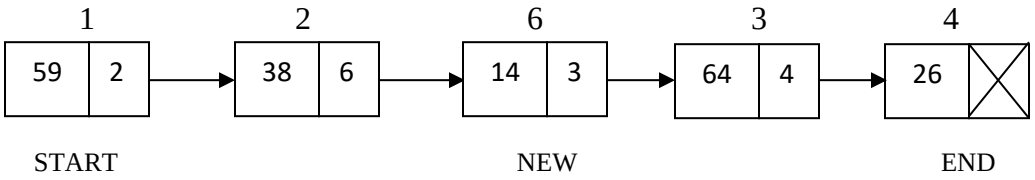
Step 1: Allocate memory for a NEW node
DATA(NEW) ← x

```
                LINK(NEW) ← NULL
Step 2:         k ← 1
                PTR ← START
Step 3:         Repeat WHILE k < pos
                        TEMP ← PTR
                        PTR ← LINK(PTR)
                        k ← k+1
                EndRepeat
Step 4:         LINK(TEMP) ← NEW
                LINK(NEW) ← PTR
Step 5:         RETURN
```

Example: Assume the initial status of the list as:



Anyinsertion (14 , 3):



iv) Deletion of a node from the list

Deleting an element from the single linked list falls into three categories as:

- Front deletion
- Rear deletion
- Any position deletion

Case 1: Front Deletion: In this case, front node information is deleted from the single linked list. For this, follow the procedure as:

Step 1 - Check whether list is **Empty** or not.

Step 2 - If it is **Empty** then, display **'List is Empty, Deletion is not possible'** and terminate the function.

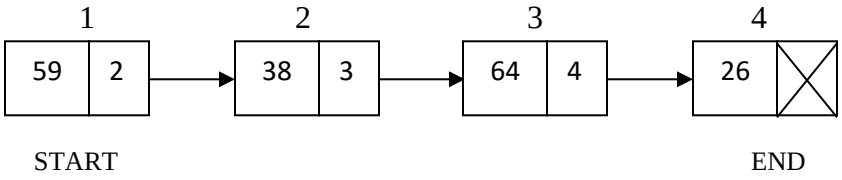
Step 3 - If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with START.

Step 4 - Check whether list is having only one node or not. If it is **TRUE** then set START and END to NULL pointer; otherwise, change START to LINK(START) and LINK(TEMP) if filled with NULL pointer.

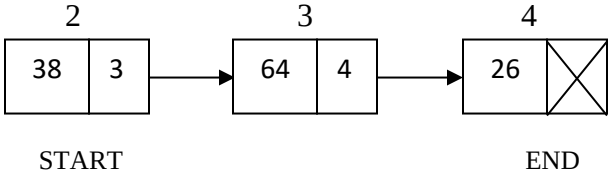
Algorithm Fdeletion( ): This function deletes the front element of the list.

```
Step 1:     IF   START = NULL THEN
                 RETURN   -1
            ELSE
                 k← DATA(START)
                 IF    START = END   THEN
                       START ← END ← NULL
                 ELSE
                       TEMP ←START
                        START ← LINK(START)
                       LINK(TEMP) ← NULL
                       Release memory of TEMP
                 ENDIF
                 RETURN   k
            ENDIF
```

Example: Assume the initial status of the list as:



Fdeletion( ): Front Deleted Element = 59



Case 2: Rear Deletion: In this case, rear node information is deleted from the single linked list.

Step 1 - Check whether list is **Empty** or not.

Step 2 - If it is **Empty** then, display **'List is Empty, Deletion is not possible'** and terminate the function.

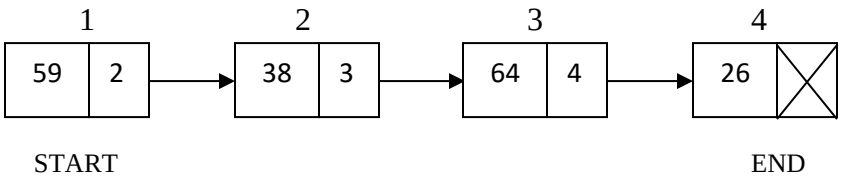
Step 3 - If it is **Not Empty** then, define a Node pointer **'ptr'** at START and **'temp'** at END pointer.

Step 4 - Check whether list is having only one node or not. If it is **TRUE** then set START and END to NULL pointer; otherwise, move ptr link to the node which is before the temp and make the appropriate links between END and ptr pointers.

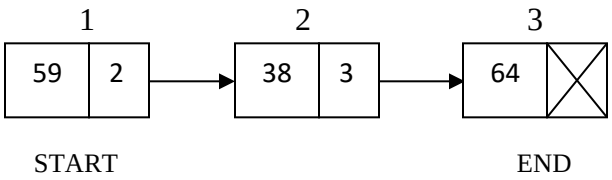
Algorithm Rdeletion( ): This function deletes the rear element of the list.

```
Step 1:     IF   END = NULL THEN
                 RETURN -1
            ELSE
                 k← DATA(END)
                 IF   START = END   THEN
                      START ← END ← NULL
                 ELSE
                      PTR ← START
                      TEMP ← END
                      Repeat WHILE   LINK(PTR) ≠ END
                           PTR ← LINK(PTR)
                      EndRepeat
                      END ← PTR
                      LINK(END) ← NULL
                      Release memory of TEMP
                 ENDIF
                 RETURN   k
            ENDIF
```

Example: Assume the initial status of the list as:



Rdeletion( ): Rear Deleted Element = 26



Case 3: Any Position Deletion: In this case, a specified position element is deleted from the single linked list.

Step 1 - Check whether list is **Empty** or not.

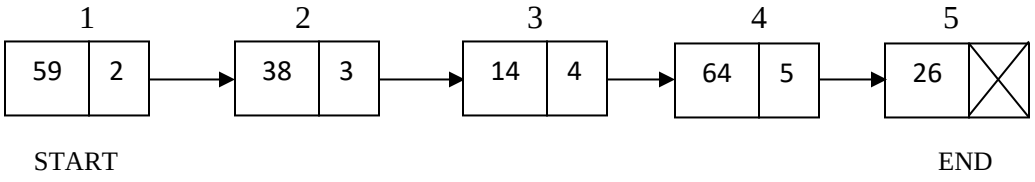
Step 2 - If it is **Empty** then, display **'List is Empty, Deletion is not possible'** and terminate the function.

Step 3 - If it is **Not Empty** then, set a temporary variable PTR points to START and move the PTR variable up to the specified location of the list along with another temporary variable TEMP. When the specified location achieved, change links in between NEW, PTR and TEMP variables.

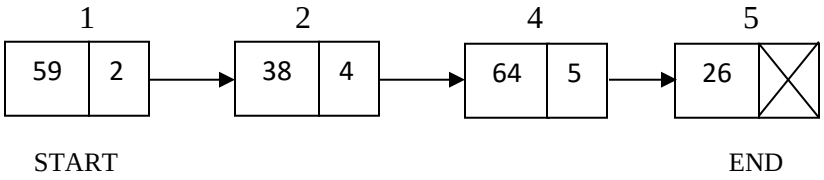
Algorithm Anydeletion(pos): This function deletes the element of the list from the specified position pos.

```
Step 1:     IF   START = NULL Then
                    RETURN   -1
            ELSE
                    PTR ← START
                    p ← 1
                    Repeat WHILE   p < pos
                            TEMP ← PTR
                            PTR ← LINK(PTR)
                            p ← p+1
                    EndRepeat
                    k ← DATA(PTR)
                    LINK(TEMP) ← LINK(PTR)
                    LINK(PTR) ← NULL
                    Release memory of PTR
                    RETURN   k
            ENDIF
```

Example: Assume the initial status of the list as:



Anydeletion(3): Deleted Element = 14



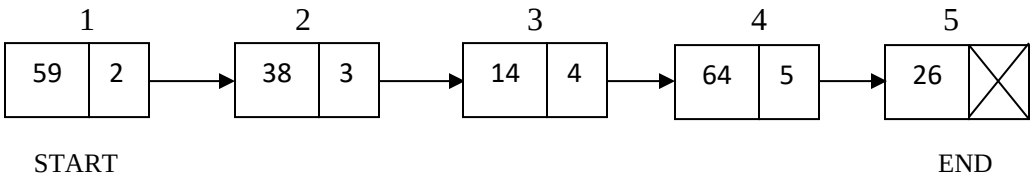
v) Counting number of nodes of the list

In this case, function counts number of nodes exists in the list.

Algorithm count(): This function is used to count number of elements of the list.

```
Step 1:     IF   START = NULL   THEN
                 RETURN   0
            ELSE
                 k ← 0
                 PTR ← START
                 Repeat WHILE    PTR ≠ NULL
                      k ← k+1
                      PTR ← LINK(PTR)
                 EndRepeat
                 RETURN    k
            ENDIF
```

Example: Assume the initial status of the list as:



count(): Number of nodes = 5

vi) Searching an element

In this case, function checks whether a key element is present in the list of elements or not. If the search element is found it refers to successful search; otherwise, it refers to unsuccessful search.

Algorithm search (key): This function checks whether an element 'key' present in the list of elements or not. It returns 1 if the search element key is found; otherwise, it returns 0.

```
Step 1:     IF   START = NULL   THEN
                 RETURN   0
            ELSE
                 PTR ← START
                 Repeat WHILE    PTR ≠ NULL
                        IF   DATA(PTR) = key   THEN
                             RETURN    1
                        ELSE
                             PTR ← LINK(PTR)
                        ENDIF
                 EndRepeat
                 RETURN   0
            ENDIF
```

Disadvantages:

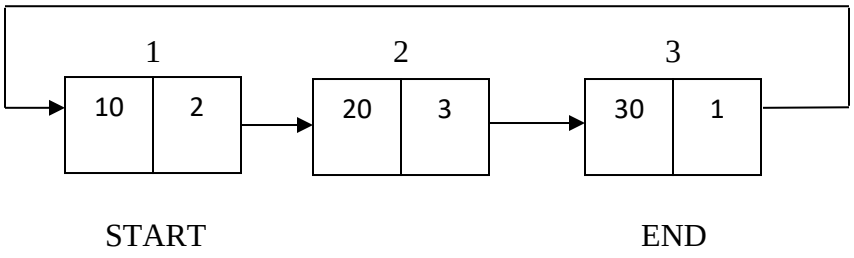
In singly linked list, it can be traversed only in forward direction. Backward traversing is not possible.

CIRCULAR LINKED LIST

In circular linked list, elements are organized in circular fashion. Circular linked list can be classified into two types as: Circular single linked list and
Circular double linked list.

Circular Single Linked List:

In circular single linked list, the LINK part of the last node contains address of the starting node. The diagrammatic representation of a circular single linked list is as follows:



Operations:

Algorithm creation(): This procedure creates a circular single linked with the specified number of elements.

```
Step 1:     Repeat WHILE TRUE
                READ an element as x
                IF x = -999 THEN
                        RETURN
                ELSE
                        Allocate memory for a NEW node
                        DATA(NEW) ← x
                        LINK(NEW) ← NULL
                        IF START = NULL THEN
                                START ← END ← NEW
                        ELSE
                                LINK(END) ← NEW
                                END ← NEW
                        ENDIF
                        LINK(NEW) ← START
                ENDIF
            EndRepeat
```

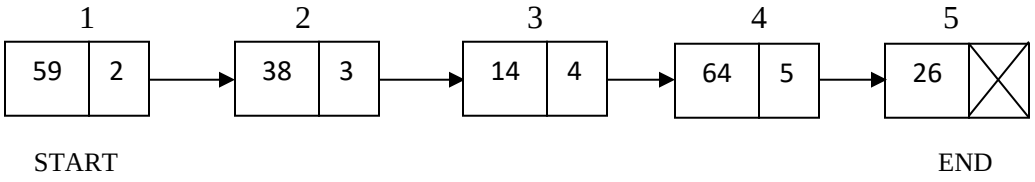
Algorithm display(): This procedure is used to display the elements of the circular single linked list from the first node to the last node.

```
Step 1:     IF START = NULL THEN
                WRITE 'Circular Single Linked List Empty'
            ELSE
                TEMP ← START
                Repeat WHILE    LINK(TEMP) ≠ START
                        WRITE    DATA(TEMP)
                        TEMP ← LINK(TEMP)
                EndRepeat
                WRITE   DATA(TEMP)
            ENDIF
Step 2:     RETURN
```

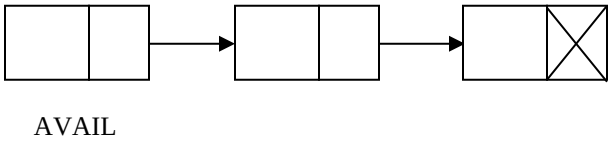
AVAILABLE SPACE LISTS

The collection of free nodes present in the memory is called a **list of available space** or **avail list** or **free pooler**. It contains unused memory cells and these memory cells can be used in future.

Example: **Linked list**



Avail List

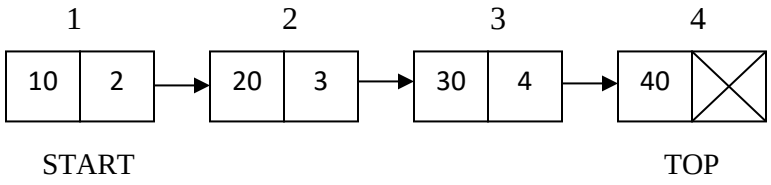


Here, AVAIL is a pointer that points the first node of the avail list. Whenever a new node is to be inserted in the linked list, a free node is taken from the AVAIL List and is inserted in the linked list. Similarly, whenever a node is deleted from the linked list, it is inserted in the AVAIL List. So that it can be used in future.

LINKED STACK

A stack is a linear data structure in which an element may be inserted or deleted only at one end, called the **TOP** of the list.

A Stack data structure can be implemented using linked list. Such a list is called as a **linked stack**. The diagrammatic representation of a linked stack can be shown as:

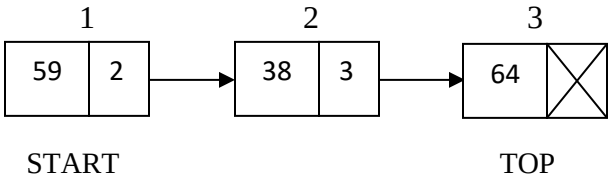


Algorithm push (ITEM): This procedure inserts a new element ITEM at TOP position of the stack.

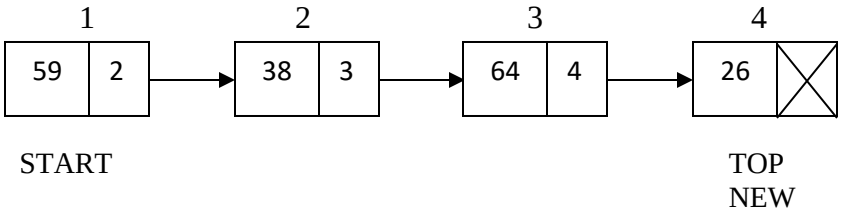
Step 1: Allocate memory for a NEW node
DATA(NEW) ← ITEM

```
                LINK(NEW) ← NULL
Step 2:         IF   TOP = NULL  THEN
                        START ← TOP ← NEW
                ELSE
                        LINK(TOP) ← NEW
                        TOP ← NEW
                ENDIF
Step 3:         RETURN
```

Example: Assume initial status of the linked stack as:



push *(26):*

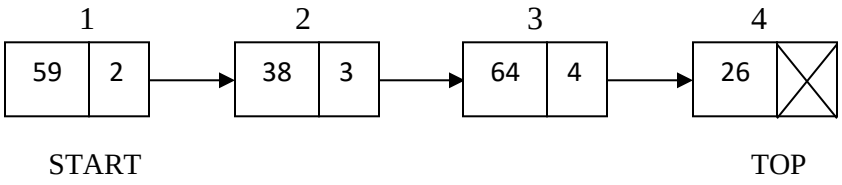


Algorithm pop (): This function deletes the topmost element from the stack.

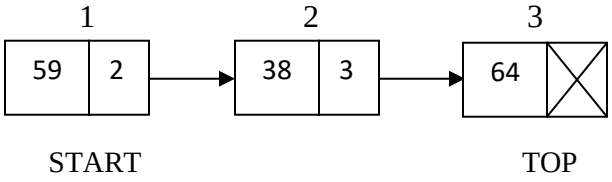
```
Step 1:         IF  TOP = NULL THEN
                        WRITE    'STACK UNDERFLOW'
                        RETURN   -1
                ELSE
                        K ← DATA(TOP)
                        IF  START = TOP   THEN
                                START ← TOP ← NULL
                                RETURN    K
                        ELSE
                                PTR ← START
                                TEMP ← TOP
                                Repeat WHILE   LINK(PTR) ≠ TOP
                                        PTR ← LINK(PTR)
                                EndRepeat
                                TOP ← PTR
                                LINK(TOP) ← NULL
```

```
                    Release memory of TEMP
                    Return K
                ENDIF
            ENDIF
```

Example: Assume initial status of the linked stack as:



pop(): Deleted Element = 26

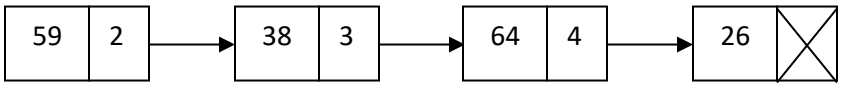


Algorithm display(): This procedure is used to print elements of the stack.

```
Step 1:     IF   TOP = NULL    THEN
                WRITE 'Linked Stack Empty'
            ELSE
                TEMP ← START
                REPEAT    WHILE TEMP ≠ NULL
                        WRITE    DATA (TEMP)
                        TEMP ← LINK(TEMP)
                ENDREPEAT
            ENDIF
Step 2:     RETURN
```

Example: Assume initial status of the linked stack as:

1 2 3 4



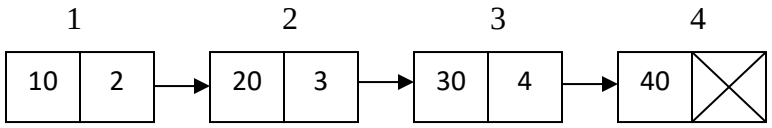
START TOP

display(): Linked Stack Elements Are = 59 38 64 26

LINKED QUEUE

A queue is a linear data structure in which elements can be inserted only at one end, called **REAR** end and elements can be deleted from the other end, called **FRONT** end of the list.

A Queue data structure can be implemented using linked list. Such a list is called as a **linked queue**. The diagrammatic representation of a linked queue can be shown as:

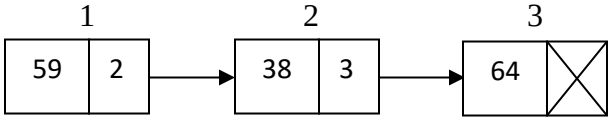


FRONT REAR

Algorithm insertion (ITEM): This procedure inserts a new element ITEM at REAR position of the linked queue.

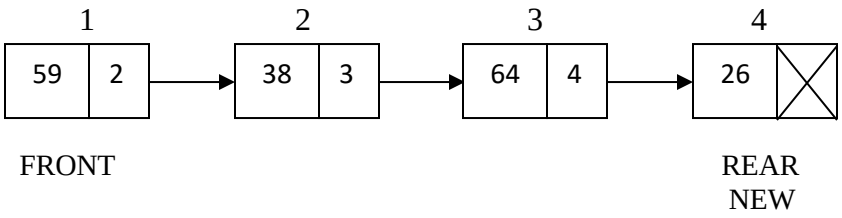
```
Step 1:   Allocate memory for a NEW node
          DATA(NEW) ← x
          LINK(NEW) ← NULL

Step 2:   IF  FRONT  = NULL   THEN
                FRONT ← REAR ← NEW
          ELSE
                LINK(REAR) ← NEW
                REAR ← NEW
          ENDIF
Step 3:   RETURN
```

Example: Assume initial status of the linked queue as:



FRONT REAR

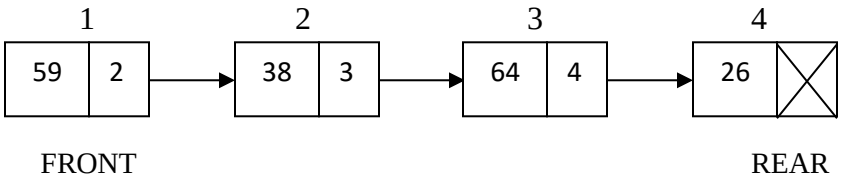
insertion (26):



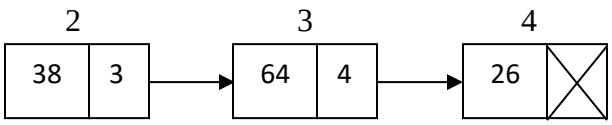
Algorithm deletion(): This function deletes the front position element of the linked queue.

```
Step 1:     IF   FRONT = NULL   THEN
                 RETURN -1
            ELSE
                 k← DATA(FRONT)
                 IF   FRONT = REAR   THEN
                      FRONT ← REAR ← NULL
                 ELSE
                      TEMP ← FRONT
                      FRONT ← LINK(FRONT)
                      LINK(TEMP) ← NULL
                      Release memory of TEMP
                 ENDIF
                 RETURN   k
            ENDIF
```

Example: Assume initial status of the linked queue as:



deletion(): Front Deleted Element = 59

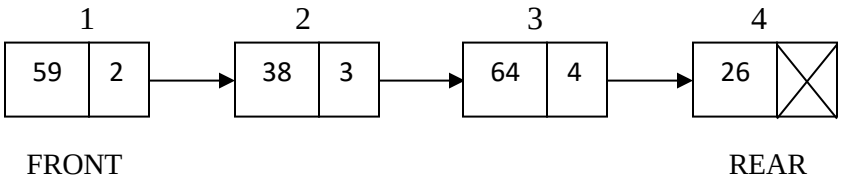


FRONT REAR

Algorithm display (): This procedure is used to display the elements of the linked queue.

```
Step 1:   IF  FRONT = NULL THEN
              WRITE 'Linked Queue Empty'
          ELSE
              TEMP ← FRONT
              REPEAT  WHILE TEMP ≠ NULL
                      WRITE   DATA (TEMP)
                      TEMP ← LINK(TEMP)
              ENDREPEAT
          ENDIF
Step 2:   RETURN
```

Example: Assume initial status of the linked queue as:



display(): Linked Queue Elements Are = 59 38 64 26

END