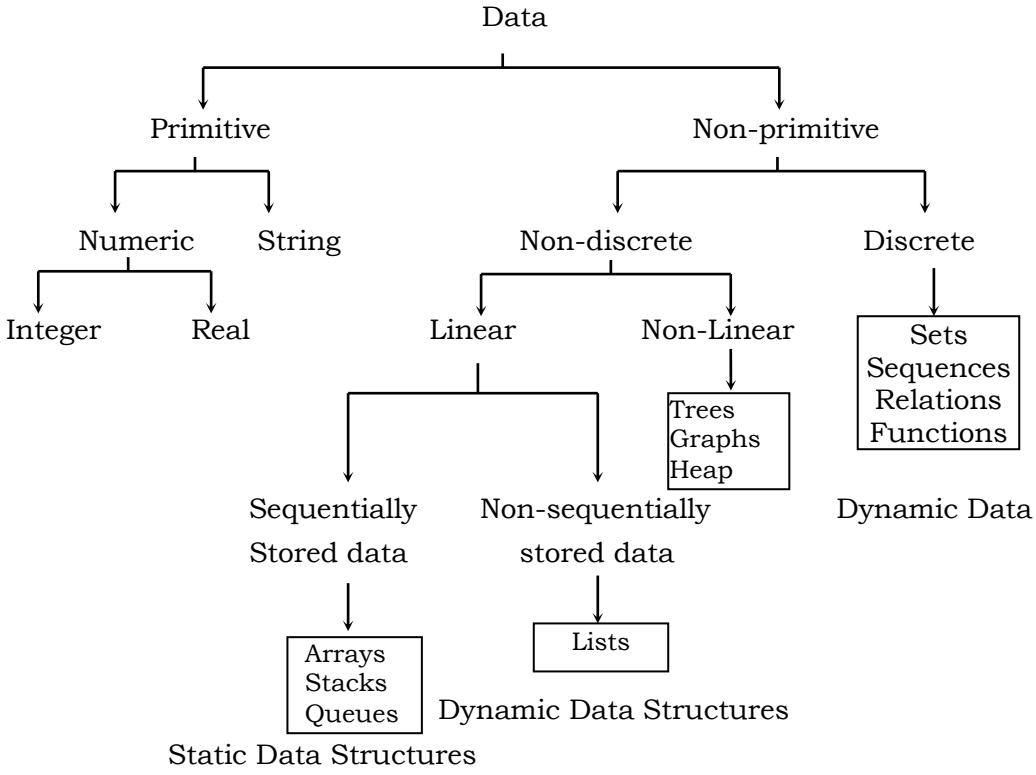
Data Structures : The method to store information in computer memory is collectively known as ***Data Structure***. A data may be organized in many ways. The logical or mathematical model of a particular data organization is called as ***data structure***. The representation of a particular data structure in the memory of computer is called a *storage structure*. The data structures are classified as shown below.



Data Structures Representation : Data structures can be represented in memory in two ways. They are :

- By using Arrays (Sequential)
- By using pointers (Linked list)

Implementing Data Structures by using pointers is efficient than implementing by using arrays. It gives more flexibility.

Types of Data Structures :

Primitive Data Structures : The primitive data structures are those, which are directly supported by the machine.
Ex : Integers, Real, Strings

Non-Primitive Data Structures : These data structures have no specific instructions to manipulate the individual data items.

Linear data structure : Data structure using sequential allocation are called linear data structure. A linear data structure displays the relationship of adjacency between elements.

Ex:- Arrays, Records, File, Stacks, Queues, etc.,

Non-linear data structure : Data structure using linked allocation are called non-linear data structure.
Ex:- Lists, Trees, Graphs, etc.,

Abstract Data Type : A data structure along with all the possible operations on that data is called as an ***Abstract Data Type***. An ADT can be implemented in the hardware/software of a computer. The term Abstract Data Type refers to the basic mathematical concept that defines the data type.

Arrays : An array is an ordered set which consists of fixed number of objects (elements). A single dimension array is called as a Vector. A two dimensional array is called as a Matrix. Vector structure is used for search and sort operation. Matrix structure is used for (i) Solution to simultaneous equations (ii) Storing relation in the computer system (iii) Storing networks and graphs in the computer system.

Advantages of Array :

1. It is used in organizing similar data in groups.
2. Each element in an array can be indicated by one or two subscripts (One subscript of vector, and two subscripts in a matrix). This provides for using a single name for the elements in an array. Each individual element can be accessed by name and subscript.
3. In the name table there will be only one entry which bears the common name of the array element and the address of the 1st element in the array only. So the size of the name table is reduced and hence the use of main memory area for name table is also reduced (name table is built by the logical phase of the compiler. It contains the name of the available in the user's program, the attributes and the memory address allocated to that).
4. The elements in an array are stored sequentially in a memory i.e. in contiguous memory locations. So from the address of the 1st element, the address of any element will be easily computed.
5. Array structure facilitates the use of loop statements in the program. Which reduces the size of the program and the memory area required for storing the program.

Disadvantages :

1. No element can be added into an array.
2. An existing element can be deleted. But the number of swap operations are more if the array size is too large (An existing element can be altered in the array)
3. Given array elements are stored in contiguous memory locations, if the array size is too large it may be difficult to allocate a single contiguous area in the memory to store the elements.

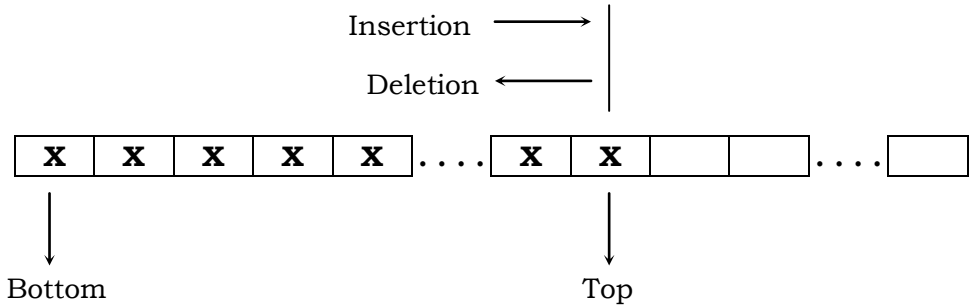
Stacks : A **stack** is a data structure, which accepts all insertion or addition of data and deletions or removal of data are made at one end, called the **Top** of the stack. Placing or adding data on into the stack is called as ***"push"*** operation. Removal or deleting data from the stack is called as ***"pop"*** operation. The most and least accessible elements in a stack

are known as the ***Top*** and ***Bottom*** of the stack, respectively. Note that the last item pushed onto a stack is always the first popped element from stack. This property is called Last In First Out or LIFO.

A stack may be of hardware or software. A hardware stack is contained in the CPU. It consists of a group of registers, placed one over the other. A software stack is a fixed area in the memory which is used as stack.

Software stack is preferred because hardware stack can be of limited size only and the hardware stack increases the size of the CPU and its cost.

A more suitable representation of a stack is given in the following figure.

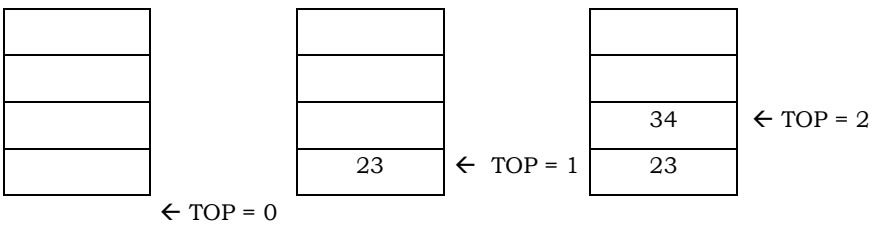


Operations on Stacks : There are four operations on stacks.

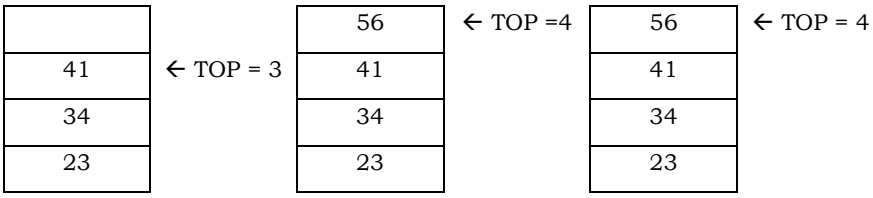
1. Placing data on the top of the stack (PUSH)
2. Removing data from the top of the stack (POP)
3. Get the value at the specified location (PEEP)
4. Change the value at the specified location (CHANGE)

Among the above PUSH and POP are major operations on stacks. While placing a data on the top of the stack, it should check whether the stack is already full. While take out data from the top of the stack, it should be checked whether the stack is empty. A stack which is full is said to be in the stack ***overflow*** condition. A stack which has no elements is said to be stack ***underflow*** condition. A pointed data structure is used for checking overflow and underflow of the stack. This pointer is symbolically denoted as "Top" when the stack is empty the value of the top will be equal to zero as and when a data is placed on stack it will be incremented by one. As and when the data is taken out from the stack it will be decremented by one. If the value of the top equals 'n' where n is maximum size of the stack, it means the stack is full.

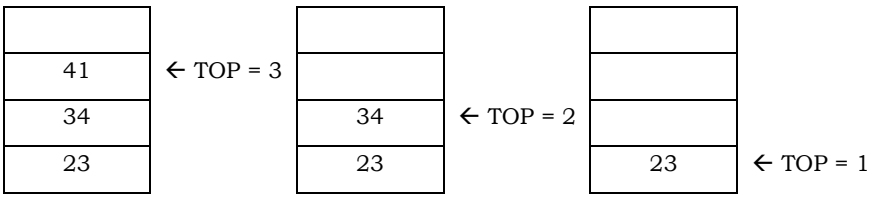
Basic Stack Operations : Consider a stack of size 4,



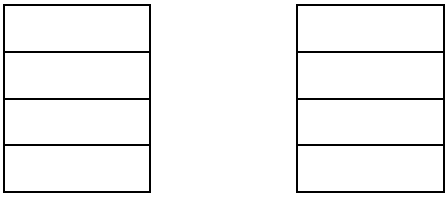
(i) Empty Stack (ii) Push 23 (iii) Push 34



(iv) Push 41 (v) Push 56 (vi) Push 60 (Overflow)



(vii) Pop (viii) Pop (ix) Pop



(x) Pop (xi) Pop (Underflow)

PUSH Operation :

Algorithm Push(S[ ], Top, x).

/* This function inserts an element 'x' to the top of stack which is represented by S, containing N elements with a pointer 'Top' denoting the top element of the stack. */

Step 1 : [Check for the stack overflow]
if(Top>=N)
then
print(" STACK OVERFLOW ")
return

Step 2 : [Increment Top pointer]
Top ← Top + 1

Step 3 : [Insert element]
S[Top] ← x

Step 4 : [finished]
return

Explanation **:** First step will checks for an overflow condition. If such a condition exists, then the insertion cannot be performed and an appropriate error message will be displayed. Second step Top pointer will be incremented by one. Third step element will be inserted into the stack (array).

POP Operation :

Algorithm Pop(S[ ], Top).

/* This function removes the top element from stack S and returns this element. Top is a pointer to indicate the top element of the stack. */

```
Step 1 : [Check for Underflow on stack]
         if(Top=0)
         then
              print(" Stack Underflow ")
              exit
Step 2 : [Decrement pointer]
              Top ← Top - 1
Step 3 : [Return the popped element ]
         return(S[Top+1])
```

Explanation : First step underflow condition is checked. If there is an underflow, then appropriate error message will display. Second step Top pointer will be decremented by one. Third step the popped element will be returned..

PEEP Operation :

Algorithm Peep(S[ ], Top, i).

/* It will returns the value of the ith location from the top of the stack S, contains N elements. The pointer Top denoting the top element of the stack. */

```
Step 1:  [Check for stack underflow]
         if(Top-i+1<=0)
         then
              print(" STACK UNDERFLOW ON PEEP ")
              exit
Step 2 : [Return ith element from top of stack]
         return(S[Top-i-1])
```

Explanation : The first step of this algorithm will checks for an overflow condition. If such a condition exists, we are unable to peep. Second step will peep the element at ith location.

CHANGE Operation :

Algorithm Change(S[ ], Top, x, i).

/* 'x' value to be placed in ith location of stack S, consisting of N elements. Top is a pointer variable to denote top of the stack. */

```
Step 1:  [Check for stack underflow]
         if(TOP-I+1<=0)
         then
              print("  STACK UNDERFLOW ON CHANGE ")
              return
Step 2:  [Change the ith element with the value of 'x']
         S[Top-i+1]  ← x
Step 3 : [Finished]
         return
```

Display of Stack :

Algorithm Display(S[ ])

/* To Display the contents of the Stack s. Stack S is an array consisting of N elements. Top is a pointer variable to indicate the top element of the stack. */

```
Step 1 :  [Check for stack empty or not]
          if(Top=0)
          then
               print(" Stack Empty ")
               return
Step 2 :  [Display the stack elements]
          Repeat for i = Top down to 1
                    print(s[i])
Step 3 :  [End]
          stop
```

Explanation : First step checks that the stack contains at least one element or not and print its corresponding message. Second step display the contents of stack from Top to 1.

Application of Stacks :

- Reversing a list
- Conversion of an infix expression into a postfix expression
- Parentheses checker
- Evaluation of postfix expression
- Conversion of an infix expression into a prefix expression
- Evaluation of prefix expression
- Recursion

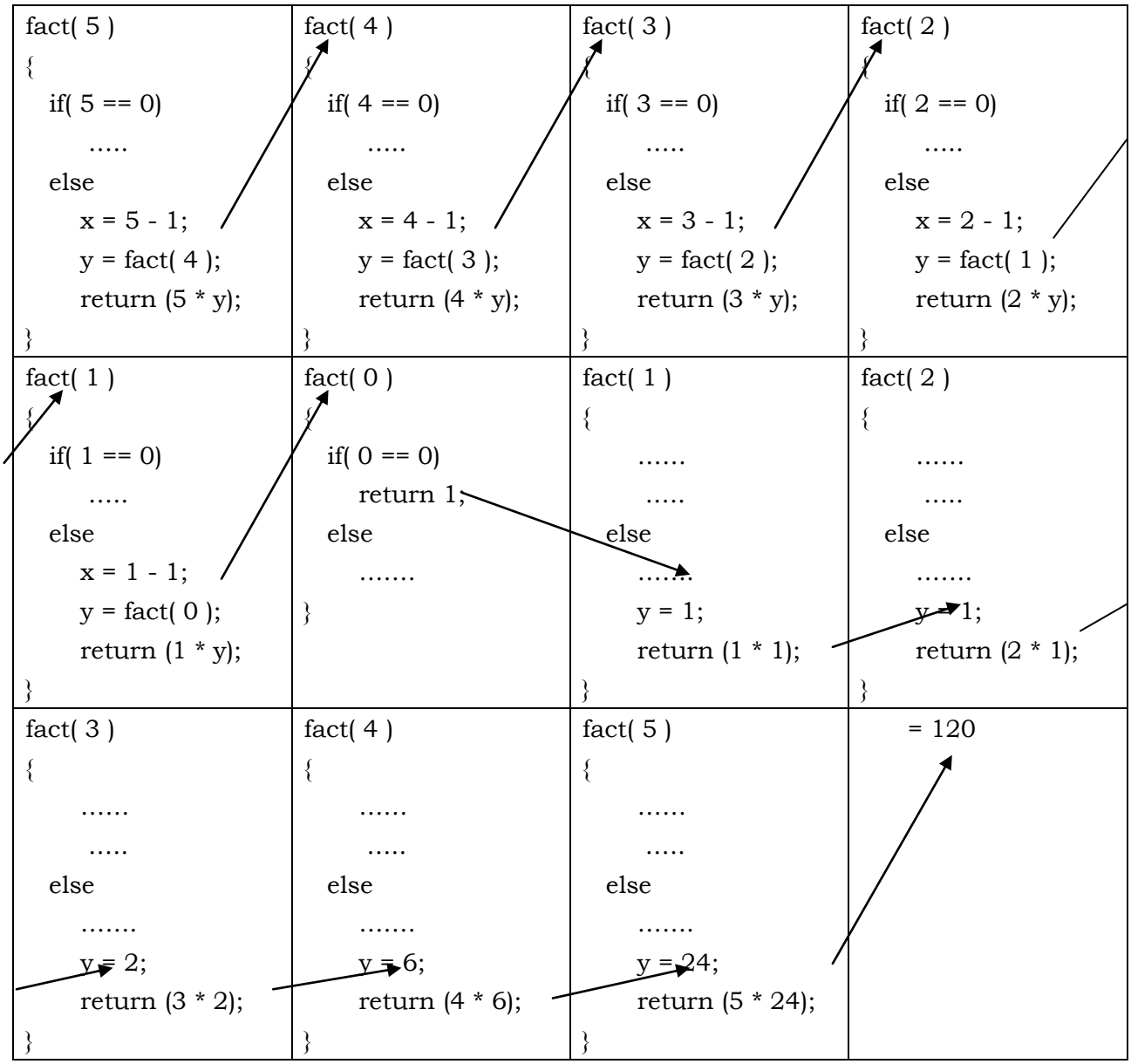
Factorial Calculation (Recursion) : A recursive function is defined as a function that calls itself. The factorial of a number is multiply the number with factorial of the number that is 1 less than that number, it can be written as :

n! = n X (n-1)! = [n X (n -1) X (n – 2) X (n – 3) X . . . X 1]

Implementation :

```
int fact(int n)
{
   if(n==0)
        return 1;
   else
        x = n-1;
        y = fact(x);
        return (n * y);
}
```

Factorial 5 :



Example :

5! = 5 X 4!
= 5 X 4 X 3!
= 5 X 4 X 3 X 2!
= 5 X 4 X 3 X 2 X 1!
= 5 X 4 X 3 X 2 X 1 X 0!
= 5 X 4 X 3 X 2 X 1 X 1
= 5 X 4 X 3 X 2 X 1
= 5 X 4 X 3 X 2
= 5 X 4 X 6
= 5 X 24
= 120

Reversing a list : An array can be reversed by reading each number from the array starting from the first index and pushing it into a stack. Once all the array elements have been read, the elements can be popped one at a time and store in the array starting from the first index.

Evaluation of Arithmetic Expressions : All algebraic expressions are represented in three ways : they are *prefix*, *infix*, and *postfix*. The prefix of "pre---", "post---", and "in---" refers to the relative position of the operator with respect to two operands. An arithmetic expression using infix notation the operator is placed in between the operands. To evaluate expression we need operator precedence and associativity rules and brackets. In computers, usually the expressions written in infix notation are evaluated by first converting it into its postfix or prefix notation and then evaluates the expression. Consider the expression A + B. Here A and B are operands and + is a operator. This can be shown in three notations :

A + B	Infix
+ A B	Prefix (Polish)
A B +	Postfix (Suffix or Reverse Polish)

Infix to Postfix Conversion :

1. Infix form is A + B * C

A + B * C	Multiplication takes precedence over addition
A + (B * C)	Convert the multiplication
A + (B C *)	Convert the addition
A (B C *) +	Written in Postfix notation
A B C * +	Postfix Form

2. Infix form is (A + B) * C

(A + B) * C	Parentheses takes precedence
(A B +) * C	Convert the multiplication
(A B +) C *	Written in Postfix notation
A B + C *	Postfix form

3. Infix form ((A + B) * C – (D – E)) $ (F + G)

((A + B) * C – (D – E)) $ (F + G)	Parentheses takes precedence, Convert it from left to right
((A B +) * C – (D E -)) $ (F G +)	Convert the multiplication
((A B + C *) – (D E -)) $ (F G +)	Convert the subtraction
(A B + C *D E - -) $ (F G +)	Convert the $
A B + C * D E - - F G + $	Postfix form

Infix to Prefix Conversion :

1. Infix form is A + B * C

A + B * C	Multiplication takes precedence over addition
A + (B * C)	Convert the multiplication
A + (* B C)	Convert the addition
+ A (* B C)	Written in Prefix notation
+ A * B C	Prefix Form

2. Infix form is (A + B) * C
   - (A + B) * C Parentheses takes precedence, Convert the addition
   - (+ A B) * C Convert the multiplication
   - * (+ A B) C Written in Prefix notation
   - * + A B C Prefix form
3. Infix form ((A + B) * C – (D – E)) $ (F + G)
   - ((A + B) * C – (D – E)) $ (F + G) Parentheses takes precedence, Convert it from left to right
   - ((+ A B) * C – (- D E)) $ (+ F G) Convert the multiplication
   - ((* (+ A B) C) – (- D E)) $ (+ F G) Convert the subtraction
   - (- * + A B C – D E) $ (+ F G) Convert the $
   - $ - * + A B C – D E + F G Prefix form

Infix to Postfix Conversion Algorithm : This algorithm uses a stack to hold operators. It also serves as a storage structure, since no operator can be printed until both of its operands have appeared.

1. Initialize an empty stack for operators
2. While no error and not to end of infix expression
   - Get the input token (Constant, variable, operator, parentheses) from infix expression
   - If the token is :
     - A left parenthesis : push it into the stack
     - A right parenthesis : Pop and display stack elements until a left parenthesis is encountered. Do not display left parenthesis. If no left parenthesis found it is an error.
     - An operator : If the stack is empty or input token is higher priority than the top of the stack, push on to stack.
   - Otherwise Pop and display the stack element, then repeat the comparison of input token with new top stack item.
   - An operand : Just display it.
3. When end of Infix Expression is reached, pop and display stack items until the Stack Is Empty

Example : Consider the infix expression A * B ^ C + D - E

Token	Stack	Postfix Expression
A	-	A
*	*	A
B	*	A B
^	* ^	A B
C	* ^	A B C
+	+	A B C ^ *
D	+	A B C ^ * D
-	+ -	A B C ^ * D
E	+ -	A B C ^ * D E
		A B C ^ * D E - +

Checking a set of Parenthesis : Parenthesis are used in arithmetic expressions and logical expressions. During compilation, the compiler checks whether any parenthesized expression in the source program is well formed. i.e., there should be as many right parenthesis as the number of left parenthesis.

The parenthesis that can be used are :

(i) ( ) semicircular brackets (parenthesis)
(ii) [ ] Rectangular (Square) brackets
(iii) { } Double brackets / Braces.

The evaluation of arithmetic operation is based on the following principle. The parenthesis symbols are read one by one from left to right. If the symbol read is a left symbol it is pushed into the stack. If it is a right symbol it is compared with symbol on the top of the stack. If the symbol on the top of the stack is the left mate of symbol read it and is popped out from the stack. If it is not it's left mate message will be displayed that the set of parenthesis is not well formed. It is the right mate after popping it out the next symbol is read, and the process is continued. After the last symbol is processed and the stack is empty, the set of parenthesis is taken to be well formed. If during processing it is formed the top of stack is empty, it means, the right parenthesis are more than the left parenthesis or if at the end, some symbols are still left in the stack (i.e., top of stack is not empty) it means there are more left parenthesis than the right. In both the above cases, the parenthesis are not well formed.

Example : Consider the expression {[( )]}

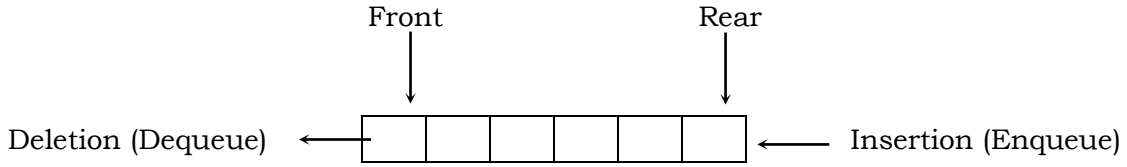
Token	Stack	Description
{	{	Left braces, pushed into the stack
[	{ [	Left braces, pushed into the stack
(	{ [(	Left braces, pushed into the stack
)	{ [	Compare this symbol on top of stack, if it is left parenthesis pop it
]	{	Compare this symbol on top of stack, if it is left parenthesis pop it
)		Compare this symbol on top of stack, if it is left parenthesis pop it

Algorithm Well-Formed()

```
/*  Checks the sequence of parenthesis expression is well formed or not  */
Step 1:  [initialization]
         Top ← 0
         i ← 0
Step 2:  Do step 3 until no symbol x is left in the input
Step 3:  [ Checking the symbol ]
         i ← i+1
         input (x[i])
         //If x[i] is a left symbol check for stack overflow  and if no overflow push it into
         stack
         If  x[I] is a left symbol
         Then
               Top  ← Top + 1
```

```
                    If Top > N
                    Then
                    Print("STACK OVERFLOW)
                    Exit
               Else
                    S[Top]  ← x[i]
          Else
                    If Top = 0  (or)  S[Top] <> Leftmate(x[i])
                    Then
                    Print (" ILL-FORMED ")
                    Exit
               Else
                    Top  ← Top - 1
Step 4:   [ Display the Result ]
          If Top = 0
          Then
                    Print(" WELL-FORMED ")
                    Exit
          Else
                    Print (" ILL-FORMED ")
                    Exit
```

Queues : Another important subclass of an array permits deletion to be performed at one end of the list and insertion at the other end. The information in such a list is processed in the same order as it was received, that is on a first-in, first-out (FIFO) or a first-come, first-served (FCFS) basis. This type of array is frequently referred to as ***queue.*** The following fig. is a representation of a queue.



The end at which data can be added or inserted is called as the "rear" end. The end of which data can be removed or deleted is called as "front" end. Therefore two pointers are required to represent the queue. The pointers are called FRONT and REAR or F and R pointers.

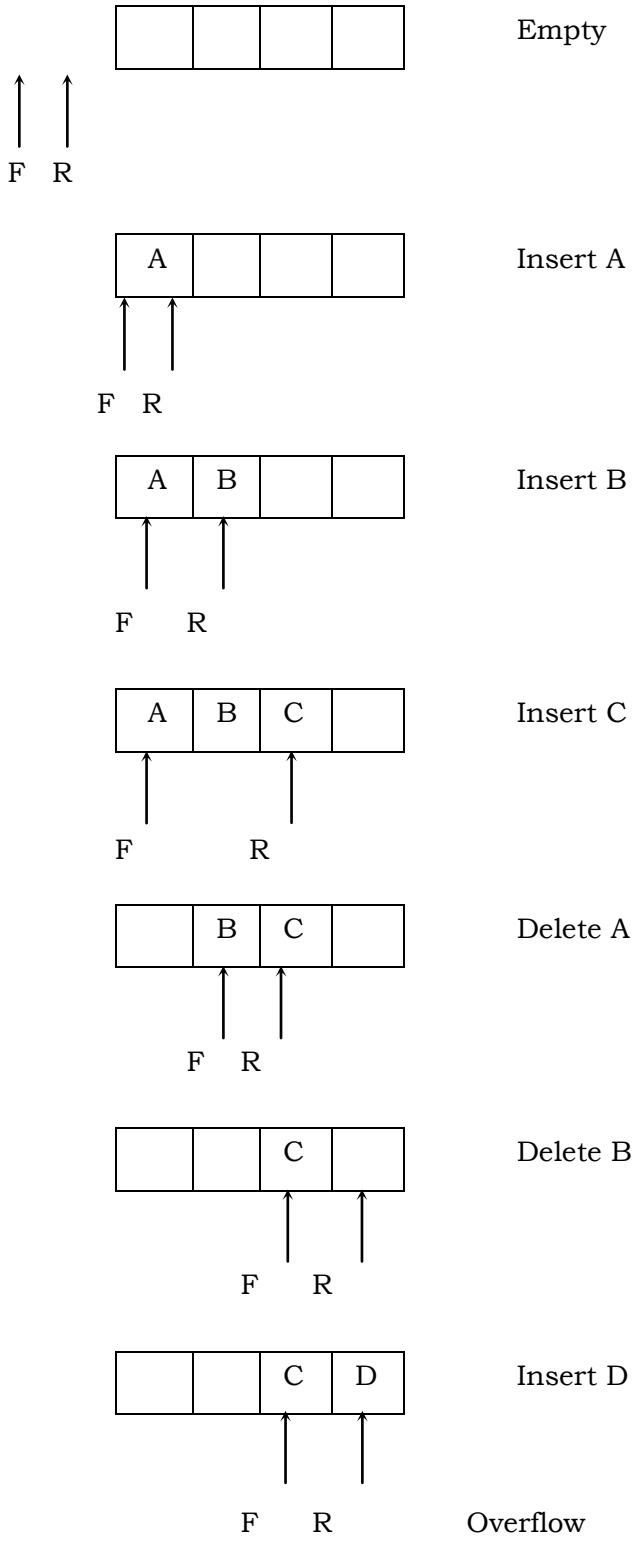
Operations on Queues : The queue major operations on a queue are (i) Insertion of data at the rear end (Enqueue) (ii) Deletion of data from the front end (Dequeue).

Note :

1. If queue is empty, both the front and rear pointers will be starting zeros.
2. If both front and rear pointers stores the same non-zeros values it means only one element is having data in the queue.
3. If the rear pointer stores the value of the maximum size of the queue, it means the queue is full at the rear end

Before inserting the new data into the queue the other rear pointer should be checked if it is already at the maximum value it means there is no vacant in the rear end and such as addition or insertion can not be made. Similarly deleting the data from the queue the front pointer is to be checked if it is zero there is no data in the queue and hence no deletion can be made.

Trace of operation on a queue :



Enqueue Operation :

```
Algorithm QInsert(Q,F,R,n,x)
/*  F and R are pointers to the front and rear elements of a queue, this queue Q is an array
consisting of ' n' elements and 'x' is an element to insert at the rear end of the queue  */
Step 1:  [Check for Overflow]
         if R>=n
         then
              print(" Q Overflow ")
              return
Step 2 :  [Increment rear pointer]
         R ← R + 1
Step 3 :  [Insert element]
         Q[R] ← x
Step 4 :  [Set front pointer, if necessary]
         if(F=0)
         then
              F ← 1
              return
```

Explanation : First step checks the Queue overflow i.e. Queue full or not, corresponding error message will display. Second step increments the rear pointer, which is insertion should be at rear point. Third step inserting the element. Last step if the inserted element is the first element then front pointer set to 1.

Queue (Deletion) Operation :

```
Algorithm Qdelete(Q,F,R)
/*  F and R are the pointers to  the front and rear elements of a queue.  Last element is
deleting from Q.  x is a temporary element to hold the deleted element  */
Step 1 :  [Check for underflow]
         if(F=0)
         then
              print(" Q Underflow ")
              return
Step 2 :  [Delete element]
         x ← Q[F]
Step 3 :  [Set the front pointer ]
         if(F=R)
         then
              F ← R ← 0
         else
              F ← F + 1
Step 4 :  [Return the deleted element]
         return(x)
```

Explanation : First step checks the Q underflow i.e. q is empty or not, corresponding error message will display. Second step front pointer element is assigned to temporary variable x. Third step setting the front pointer, if the Q contains only one element then front and rear

pointers is set to zero, else front pointer is incremented by 1. Last step is return the deleted element.

Queue Display :

Algorithm Qdisplay(Q)

/* F and R are the pointers to the front and rear elements of a queue Q. It is going to display the Q array contents. */

Step 1 : [Display the Q contents]
Repeat for i = f to r
print(Q[f])

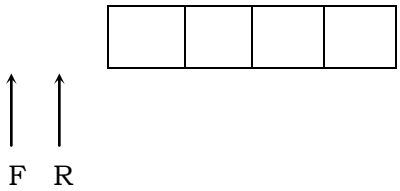
Step 2 : [End]
Stop

Explanation : First step display the contents of Q from front pointer to real pointer.

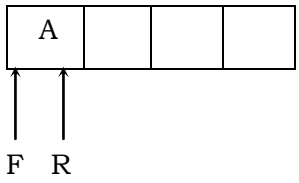
Applications of Queues :

- It is used to represent the records in a database in memory.
- It is used to store list interrupts in the operating system, which would get processed in the order in which they were generated.
- It is used by an application program to store the incoming data, which determines the order in which it is to be processed.

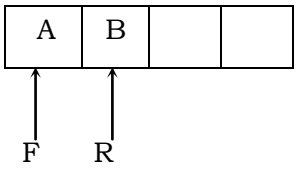
Circular Queue : The defect in a linear queue is a linear queue is if its rear end is full. No data can be inserted, even if there are few vacant on the front side. To overcome this defect the suitable method of representing a queue which prevents an excessive use of memory, is to arrange the elements Q[1], Q[2]...Q[N] in a circular fashion with Q[1] following Q[n]. Pictorially this method can be in the following fig. Here when a new element is to be inserted if it is to be found that the rear end is filled. It is checked whether there is a vacancy at the front end. If so data is placed at the front end. But deletion is always is made from the front end. Consider an example of a circular queue that contains a maximum of four elements. It is required to perform a number of insertions and deletion operations on the initially empty queue. A trace of the queue contents which is not shown as a circular for convenience is given below:



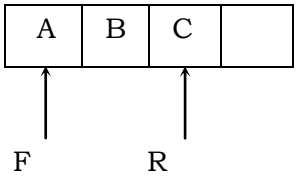
Empty



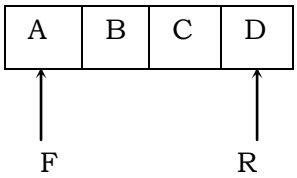
Insert A



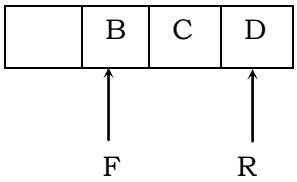
Insert B



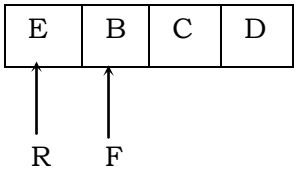
Insert C



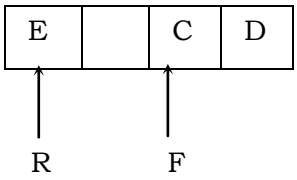
Insert D



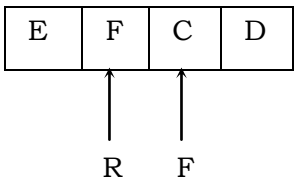
Delete A



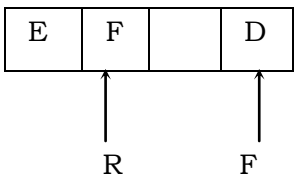
Insert E



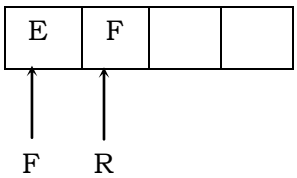
Delete B



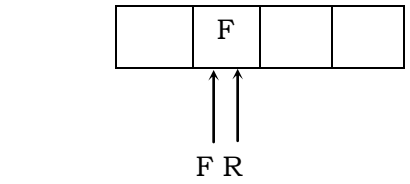
Insert F



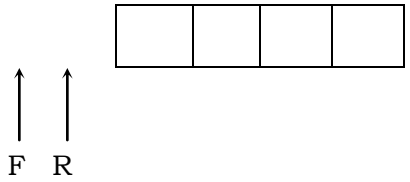
Delete C



Delete D



Delete E



Delete F

Circular Queue Insertion

Algorithm Cqinsert(Q,F,R,n,x)

/* F and R are front and rear pointers of circular Q, consisting of n elements. x is an element to be inserted into circular queue at rear pointer place. */

Step 1 : [Set the rear pointer]
R ← (R mod n) + 1

Step 2 : [Check for Overflow]
if(F=R)
then
print(" OVERFLOW ")
return

Step 3 : [Insert the element]
Q[R] ← X

Step 4 : [Is the front pointer properly set]
if(F=0)
then
F ← 1
return

Explanation: First step set the rear pointer i.e. if rear pointer is at n the place then set rear pointer to first place, else it will be incremented by 1. Second step check for overflow i.e. if front and rear pointers are indicating same insertion is not possible. Third step inserting the new element. Last step set the front pointer if necessary i.e. if q is empty then front pointer is assigned to 1.

Circular Queue Deletion :

Algorithm Cqdelete(Q,F,R,n)

/* Given F and R pointers to the front and rear of a circular queue Q, consisting of n elements. This function delete the last element of the queue */

Step 1 : [Check for underflow]
if(F=0)
then
print(" UNDERFLOW ")
return

Step 2 : [Delete element]
x ← Q[F]

Step 3 : [Queue empty]
if(F=R)
then
F ← R ← 0
return
Step 4 : [Increment front pointer]
F ← (F mod n) + 1
return

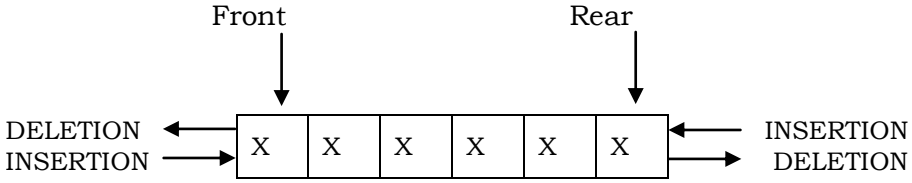
Explanation : First step check for underflow i.e. Q is empty, display the error message. Second step front pointer value is assigned to x. Third step if Q is empty then front and rear pointers are set to null. Last step set the front pointer if front pointer is at last location it is set to 1 or incremented by 1.

Circular Queue Display :

Algorithm Cqdisplay(Q)
/* F and R are the pointers to the front and rear elements of a circular queue Q. It is going to display the circular queue Q array contents. */
Step 1 : [Display the Q contents]
Repeat for i = f to r
print(Q[f])
Step 2 : [End]
Stop

Explanation: First step display the contents of Q from front pointer to real pointer.

DeQue : A single queue has been described as behaving in a first-in, first-out manner in the sense that each deletion removes the oldest element. A ***deque*** (double-ended queue) is a linear list in which insertion and deletions are made to or from either end of the list. Such a list can be represented by the following figure.



It is clear that a deque is more general than a stack or a queue. There are two variations of a deque namely, the input-restricted deque and output-restricted deque. The *input-restricted deque* allows insertion at only one end, while an *output-restricted deque* permits deletion from only one end.

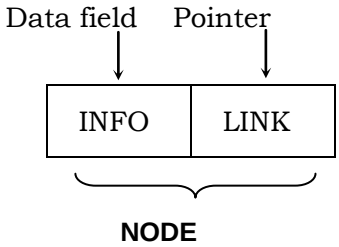
Introduction to Linked List : Few linear data structures follows sequential allocation of memory where as others follow linked allocation of memory. Whatever the type of allocation that they follow, they will maintain the physical adjacency relationship of the elements of a linear list. But the sequential allocation of memory has some disadvantages like the following:

1. Storage requirements are unpredictable
2. The amount of data storage is data dependent
3. Once the memory is allocated we cannot alter its size, if required

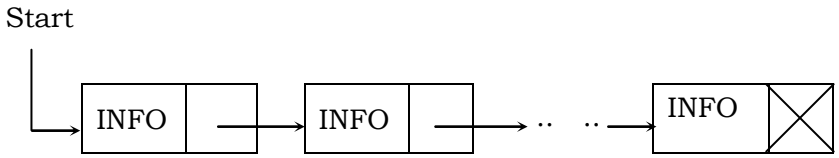
4. Frequent insertion and deletion activates the above difficulties and uses computer memory efficiently and reduce the computational time.

A linked list is a collection of structures in which each structure contains at least one member whose value is the address of the next logically ordered structure in the list.

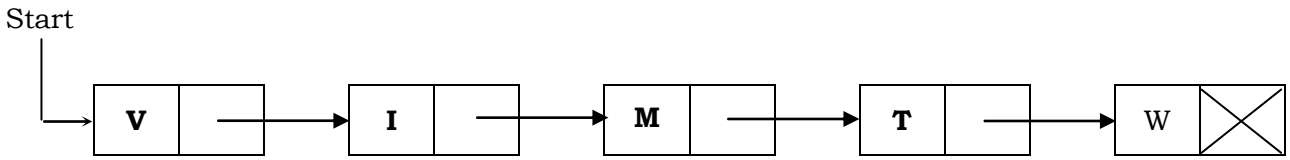
Single Linked List : Every node in a single linked list contains two parts, data part and pointer to the next node. The last node in the list does not have no next node connected to it, so it will store a special value called null. Every node contains a pointer to another node which is the same type, it is referred as *self-referential data type*. The node structure is :



A pointer variable Start hold the address on the first node. Traversal is possible in only one direction i.e. from the starting node to last node. We cannot access the previous node from the present as it has no knowledge about where the previous node is.

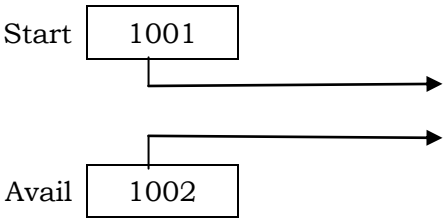


Example : Consider the following linked list to store character type of data.



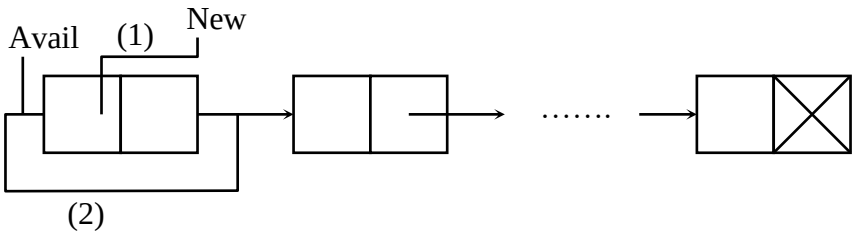
The memory representation of the above linked list is followed. From the following figure we can see that the variable Start is used to hold the address of the first node. Here it will be 1001, so that data stored at address 1001 is V. The corresponding Link stores the address of the next node, which is 1005. Therefore the second data element is obtained from the address 1005 is I. We repeat this procedure until we reach a position where the Link entry contains -1 or null denotes the end of the linked list.

The computer maintains a list of all free memory cells. This list of available space is called the *free pool*. Free pool is a linked list of all free memory cells. A pointer variable Avail stores the address of the first free space.



	Data	Link
1001	V	1005
1002		1003
1003		1004
1004		1006
1005	I	1007
1006		1009
1007	M	1008
1008	T	1010
1009		-1
1010	W	-1

Getting a free node from AVAIL : The free nodes is available in a stack, called as ***Availability Stack.*** Where the pointer variable AVAIL contains the address of the top node in the stack.



Algorithm: Getnode( )

```
/*  Getting a new node from the AVAIL stack and new node is name as NEW  */
Step 1 :  [Check for availability of node from AVAIL list ]
          If AVAIL==NULL
          Then
               Print("No free Nodes")
               Return
Step 2 :  [Obtain a node from free list & name it NEW]
          NEW=AVAIL
Step 3 :  [Adjust AVAIL pointer]
          AVAIL=LINK(AVAIL)
```

'C' representation of a Node : For this, In 'C' we should have a provision to declare user defined data type which allows the user to store one data field and one pointer as one unit of data item. This can be possible through structures because it supports heterogeneous data.

```
struct node
{
     int info;
     struct node *link;
};
```

Now node represents two fields one data field i.e. info and pointer field i.e. link.. Normally variables occupies the memory of fixed size and is known to the compiler. But the node is user defied and we should allocate memory dynamically. This dynamic memory allocation can be possible by using functions like calloc, malloc;
Memory allocation can be done by using malloc function.

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

Here 'sizeof' returns the number of bytes required for a node. 'malloc' allocates memory and returns void pointer. (Struct node *) is type caste to convert void pointer to node pointer. The members can be accessed by using pointer variable arrow (→) operator.

Linked list operations : The following are the basic list operation associated with a linked list.

1. Traversing the list
2. Counting the items in the list
3. Inserting an item into a list
4. Deleting an item from a list
5. Searching an item in a list
6. Concatenating two lists

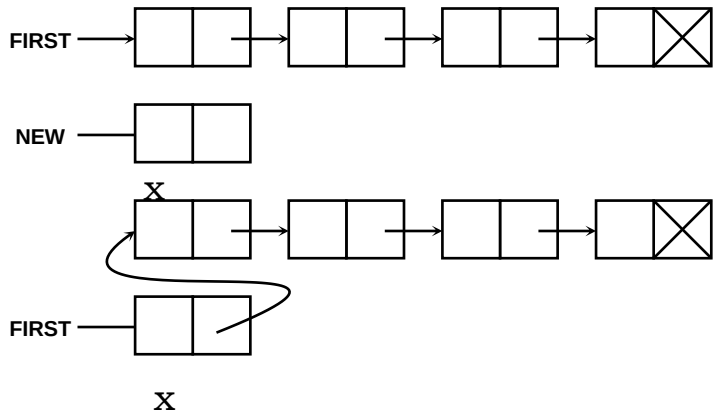
Count the number of nodes in a list : Traverse each and every node in a list, while traversing every individual node increment the counter by 1. Once we reach -1 or null, all the nodes in a linked list are traversed.

Algorithm CountNodes(FIRST)

```
/*  FIRST is a pointer to indicate the first element of linked list, a node contains INFO
and LINK fields. This algorithm counts the number of nodes in a list  */
Step 1 :  [Initialization]
          TEMP = FIRST
          Count = 0
Step 2 :  [Traverse the list ]
          Repeat through step 3 while TEMP != NULL
Step 3 :  [Processing ]
          Count = Count + 1
          TEMP = LINK(TEMP)
Step 4 :  [Displaying the Node count]
              Print Count
Step 5 :  [Return]
              return
```

Insertion : The insertion of a new element into an existing list can be done in three ways i.e. (i) First insertion (ii) End insertion (iii) Middle insertion

1. **First Insertion :**



Algorithm Insertfirst(FIRST,X)

/* X is a new element to be inserted. FIRST is a pointer to indicate the first element of linked linear list, a node contains INFO and LINK fields. This algorithm inserts a new node at beginning */

Step 1 : [Check for availability of node from AVAIL list]
If AVAIL=NULL
Then
Print("No free Nodes")
Return

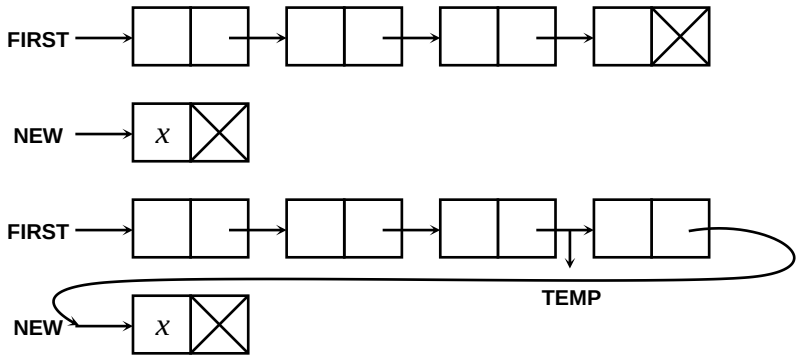
Step 2 : [Obtain a node from free list & name it NEW]
NEW=AVAIL

Step 3 : [Adjust AVAIL pointer]
AVAIL=LINK(AVAIL)

Step 4 : [Store the information and insert at first end]
INFO(NEW) ← X
LINK(NEW) ← FIRST

Step 5 : [Return the modified linked list address]
return(NEW)

2. **End Insertion :**

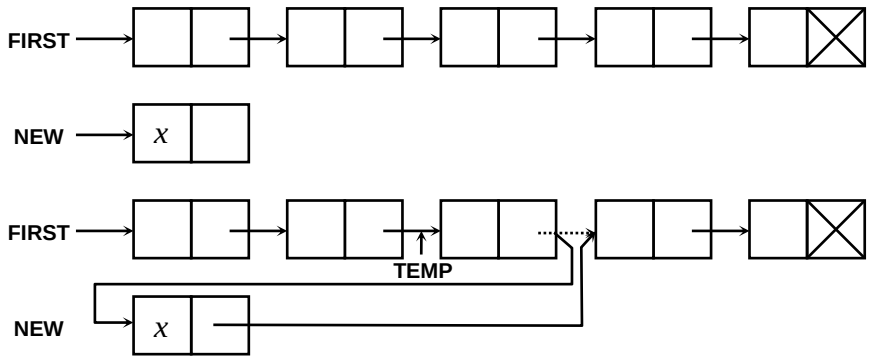


Algorithm Insertend(FIRST,X)

/* X is a new element to be inserted. FIRST is a pointer to indicate the first element of linked linear list, a node contains INFO and LINK fields. This algorithm inserts a new node at the last */

Step 1 : [Check for availability of node from AVAIL list]
If AVAIL=NULL
Then
Print("No free Nodes")
Return
Step 2 : [Obtain a node from free list & name it NEW]
NEW=AVAIL
Step 3 : [Adjust AVAIL pointer]
AVAIL=LINK(AVAIL)
Step 4 : [Store the information and link fields of this node]
INFO(NEW) ← X
LINK(NEW) ← NULL
Step 5 : [Is the list empty]
if FIRST=NULL
then
return(NEW)
Step 6 : [Initiate search for the last node]
TEMP ← FIRST
Step 7 : [Search the end of list]
repeat while LINK(TEMP)<>NULL
TEMP ← LINK(TEMP)
Step 8 : [Set link field of last node to new node]
LINK(TEMP) ← NEW
Step 9 : [Return the modified linked list address]
Return(FIRST)

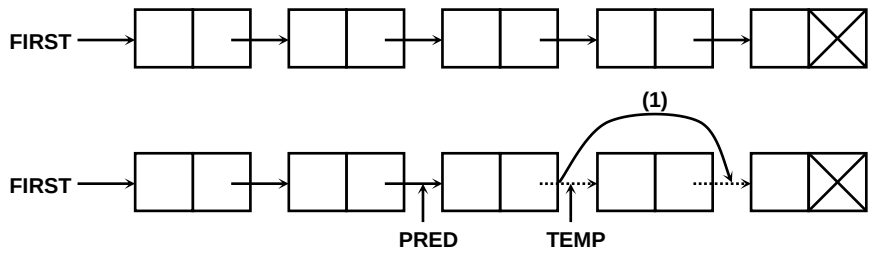
3. **Middle Insertion :**



Algorithm Insertorder(FIRST,X)
/* X is a new element to be inserted. FIRST is a pointer to indicate the first element of linked linear list, a node contains INFO and LINK fields. This algorithm inserts a new in ordered */
Step 1 : [Check for availability of node from AVAIL list]
If AVAIL=NULL
Then
Print("No free Nodes")
Return

Step 2 : [Obtain a node from free list & name it NEW]
NEW=AVAIL
Step 3 : [Adjust AVAIL pointer]
AVAIL=LINK(AVAIL)
Step 4 : [Store the information and link fields of this node]
INFO(NEW) ← X
Step 5 : [Is the list empty]
if FIRST=NULL
then
LINK(NEW) ← NULL
return(NEW)
Step 6 : [Does the new node precede all others in the list]
if INFO(NEW)<=INFO(FIRST)
then
LINK(NEW) ← FIRST
return(NEW)
Step 7 : [Initialize temporary pointer]
TEMP ← FIRST
Step 8 : [Search for predecessor of new node]
Repeat while LINK(TEMP)<>NULL and INFO(LINK(TEMP))<=INFO(NEW)
TEMP ← LINK(TEMP)
Step 9 : [Set link fields of new node and its predecessor]
LINK(NEW) ← LINK(TEMP)
LINK(TEMP) ← NEW
Step 10 : [Return first node pointer]
return(FIRST)

Deletion :



Algorithm Delete(FIRST, X)
/* Delete the node whose value is given in X. FIRST is a pointer to indicate the first element of linked linear list, a node contains INFO and LINK fields. TEMP pointer is used to find the desired note and PRED keeps track of the predecessor of the TEMP. */
Step 1 : [Check for Underflow]
if FIRST=NULL
then
print(" No nodes to delete ")
return

```
Step 2 :  [Initialize the search for node whose value is X]
              TEMP ← FIRST
Step 3 :  [Finding X]
              Repeat thru step 5 while INFO(TEMP)<>X and LINK(TEMP)<>NULL
Step 4 :  [Update Predecessor pointer]
              PRED ← TEMP
Step 5 :  [Move to next node]
              TEMP ← LINK(TEMP)
Step 6 :  [End of the list]
              if INFO(TEMP)<>X
              then
              print("Node Not Found")
              return
Step 7 :  [Delete X]
              if X=INFO(FIRST)
              then
              FIRST ← LINK(FIRST)
              else
              LINK(PRED) ← LINK(TEMP)
Step 8 :  [Return node to availability stack]
              LINK(X) ← AVAIL
              AVAIL ← X
          Return
```

Copying the existing list into the other :

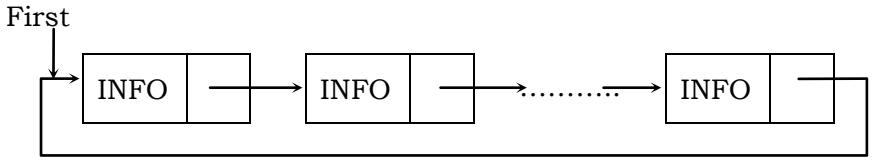
Algorithm Copy(FIRST)

/* FIRST is a pointer to the first node in existing linked list. SECOND is a pointer to hold the address of the new linked list. Each node contains INFO and LINK fields. NEW is a pointer variable to hold the address of the new node. TEMP and PRED is pointer variables is used to traverse the lists. */

```
Step 1 :  [Check for empty list]
              if FIRST=NULL
              then
              print(" Empty list ")
              return(NULL)
Step 2 :  [Get a new node and copy first node value]
              if AVAIL=NULL
              then
              print(" No free Nodes ")
              return;
              else
              NEW ← AVAIL
              AVAIL ← LINK(AVAIL)
              INFO(NEW) ← INFO(FIRST)
              SECOND ← NEW
Step 3 :  [Initialize traversal]
              TEMP ← FIRST
```

Step 4 : [Move to specified node, if not at end of list]
repeat thru step 6 while LINK(TEMP)<>NULL
Step 5 : [Update predecessor and temp pointers]
PRED ← NEW
TEMP ← LINK(TEMP)
Step 6 : [Copy node]
if AVAIL=NULL
then
print(" No free nodes ")
return;
else
NEW ← AVAIL
AVAIL ← LINK(AVAIL)
INFO(NEW) ← INFO(TEMP)
LINK(PRED) ← NEW
Step 7 : [Set link field of the last node]
LINK(NEW) ← NULL
return(SECOND)

Circular Linked List : Each node contains data and a single link which attaches it to the next node in the list and the last node consists the address of the first node. Traversal is possible in only one direction i.e. from the starting node to last node. We can traverse the list until the next entry that contains the address of the first node. Circular linked lists are widely used in operating system for task maintenance.



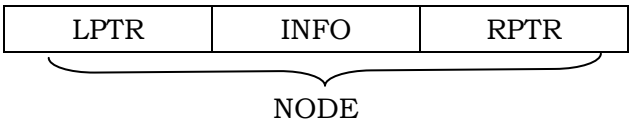
Doubly linked list : Traversal is possible in both directions either forward or reverse. Each node must contain two link fields, is used to denote the predecessor and successor of a node. The link denoting the predecessor of a node is called the left link and the link denoting the successor node is called its right link. A list containing this type of node is called a double linked linear list or two-way chain. Pictorially it is represented as

Where L and R are pointer variables denoting the left most and right most ends in the list. The left link of the left most node and the right link of the right most node are NULL, indicating that end of the list for each direction. The left and right links of a node are denoted by the variables LPTR and RPTR.

Creating Doubly linked list : Each element in the linked storage allocation is called a **node**. A node consists of three fields, namely an information field (INFO), pointer to

previous node (LPTR) and a pointer to next node (RPTR). The name of a element is denoted by NODE.

The node structure is :



The free nodes is available in a stack, called as *Availability Stack*. Where the pointer variable AVAIL contains the address of the top node in the stack.

Algorithm Getnode()

```
/*  Getting a new node from the AVAIL stack and new node is name as NEW  */
Step 1 :  [Check for availability of node from AVAIL list ]
          If AVAIL=NULL
          Then
               Print("No free Nodes")
               Return
Step 2 :  [Obtain a node from free list & name it NEW]
          NEW=AVAIL
Step 3 :  [Adjust AVAIL pointer]
          AVAIL=LINK(AVAIL)
```

C representation of a Node and memory allocation :

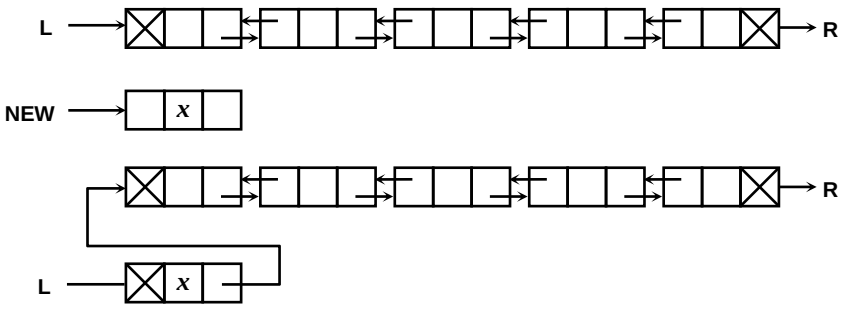
```
struct node
{
     int info;
     struct node *lptr;
     struct node *rptr;
};
```

Memory allocation can be done by using malloc function.

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

Here sizeof returns the number of bytes required for a node. malloc allocates memory and returns void pointer. The members can be accessed by using pointer variable arrow (→) operator.

First Insertion :



Algorithm Doubinsbeg(L,R,M,X)

/* L and R are the pointer variables to hold the addresses of first and last nodes. NEW is a pointer variable to hold the new node address. The left and right links of a node is LPTR and RPTR. M is a pointer variable is used to hold the address of a specified node and the information is to be entered is X */

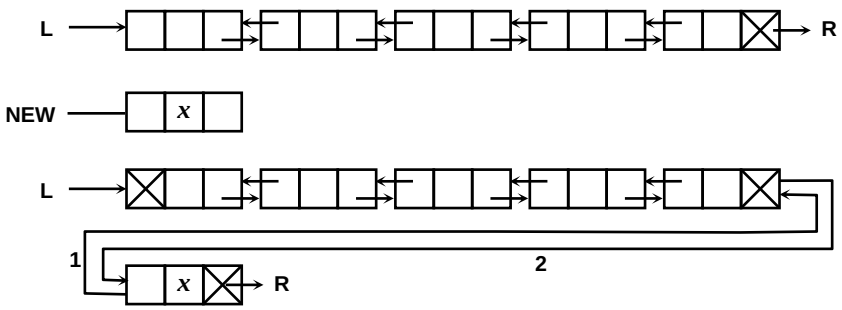
Step 1 : [Obtain a node from availability stack]
NEW ← NODE

Step 2 : [Place the information into information field]
INFO(NEW) ← X

Step 3 : [Insertion into the empty list]
if L=NULL
then
LPTR(NEW) ← RPTR(NEW) ← NULL
L ← R ← NEW
Return

Step 4 : [Left most insertion]
LPTR(NEW) ← NULL
RPTR(NEW) ← L
LPTR(L) ← NEW
L ← NEW
Return

End Insertion :



Algorithm Doubinsend(L,R,M,X)

/* L and R are the pointer variables to hold the addresses of first and last nodes. NEW is a pointer variable to hold the new node address. The left and right links of a node is LPTR and RPTR. M is a pointer variable is used to hold the address of a specified node and the information is to be entered is X. */

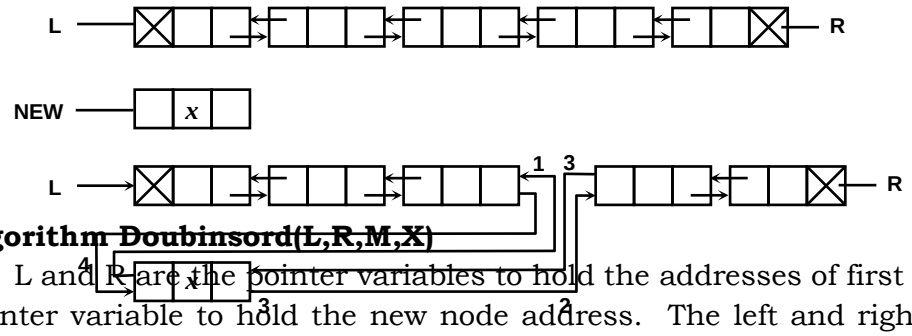
Step 1 : [Obtain a node from availability stack]
NEW ← NODE

Step 2 : [Place the information into information field]
INFO(NEW) ← X

Step 3 : [Insertion into the empty list]
if L=NULL
then
LPTR(NEW) ← RPTR(NEW) ← NULL
L ← R ← NEW
Return

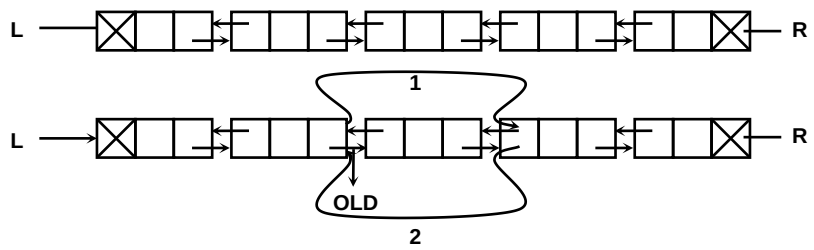
Step 4 : [Insertion at right end]
LPTR(NEW) ← R
RPTR(R) ← NEW
RPTR(NEW) ← NULL
R ← NEW
Return

Insertion in Order :



Algorithm Doubinsord(L,R,M,X)
/* L and R are the pointer variables to hold the addresses of first and last nodes. NEW is a pointer variable to hold the new node address. The left and right links of a node is LPTR and RPTR. Insertion is to be performed to the left of a specified node address given by pointer variable M. The information is to be entered is X. */
Step 1 : [Obtain a node from availability stack]
NEW ← NODE
Step 2 : [Place the information into information field]
INFO(NEW) ← X
Step 3 : [Insertion into the empty list]
if L=NULL
then
LPTR(NEW) ← RPTR(NEW) ← NULL
L ← R ← NEW
Return
Step 4 : [Does the new node precede all others in the list]
If INFO(NEW)<=INFO(L)
then
RPTR(NEW) ← L
LPTR(L) ← NEW
LPTR(NEW) ← NULL
L ← NEW
Step 5 : [Initialize temporary pointer]
TEMP ← L
Step 6 : [Search for predecessor of new node]
Repeat while LINK(TEMP)<>NULL and INFO(LINK(TEMP))<=INFO(NEW)
TEMP ← LINK(TEMO)
Step 7 : [Setting the link]
LPTR(NEW) ← RPTR(TEMP)
RPTR(NEW) ← LPTR(RPTR(TEMP))
LPTR(RPTR(TEMP) ← NEW
RPTR(TEMP) ← NEW
Return

Deletion :



Algorithm: Doubledel(L,R,OLD)

/* L and R are the pointer variables to hold the addresses of first and last nodes. This function is deleting a node whose address referred by the variable OLD. Nodes contains left and right links with names LPTR and RPTR */.

```
Step 1 :  [Check for Underflow]
          if L=NULL
          then
          print("Empty list & deletion is not possible")
          Return
Step 2 :  [Delete specified node]
          if L=R
          then
          L ← R ← NULL
          else
          if OLD = L
          then
               L ← RPTR(L)
               LPTR(L) ← NULL
          else
               if OLD = R
               then
                    R ← LPTR(R)
                    RPTR(R) ← NULL
               else
                    RPTR(LPTR(OLD)) ← RPTR(OLD)
                    LPTR(RPTR(OLD)) ← LPTR(OLD)
Step 3 :  [Return deleted node]
          Restore(OLD)
          Return
```