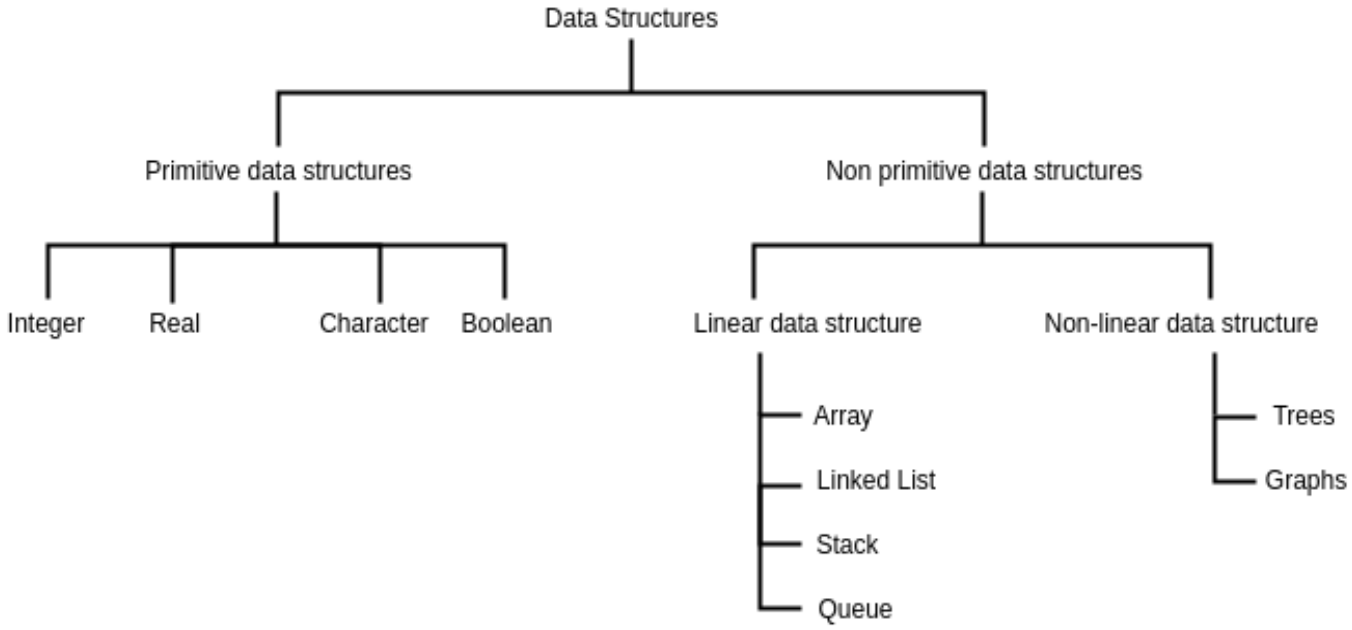
# UNIT - I

## DATA STRUCTURES :

**Data structure** is a specialized format for organizing, processing, retrieving and storing the data. Data structure is used to organize the large amount of data in a memory. Every data structure follows a particular principle.

### TYPES OF DATA STRUCTURES




**1 : Primitive Data Structures :** Primitive data structures are the basic data structures and are directly operated upon by the machine instructions are known as Primitive Data Structures. They are

1.Integer

2.Float (or ) Real

3.Char

4.Boolean

**2 : Non-Primitive Data Structures :** The Data structures that are not directly processed by machine using its instructions are known as Non-Primitive Data Structures.

**Two Types of Non-Primitive Data Structures.**

**1. Linear Data Structure**

**2. Non - Linear Data Structure**

**1 : Linear Data Structure :** Linear Data Structure organizes the data in sequential order, then that data structure is called a Linear Data Structure.

**Example :**
1. Arrays
2. List (Linked List)
3. Stack
4. Queue

**2 : Non - Linear Data Structures :** Non- Linear Data Structure organizes the data in random order, then that data structure is called as Non-Linear Data Structure.

**Example:**
1.Tree
2. Graph
3.Dictionaries
4.Heaps
5.Tries, Etc.,

**DATA STRUCTURES OPERATIONS :** The following data structure operations play a major role in processing of data.

**1.Creating :** This is the first operation to create a data structure. This is just declaration and initialization of the data structure and reserved memory locations for data element.

**2. Inserting :** Adding new records to the structure.

**3. Deleting :** Removing a record from the structure.

**4 . Updating** : it changes data values of the data structure.

**5. Traversing :** Accessing each record exactly once so that certain item in the record may be processes.

**6.Searching :** Finding the location of the record with a given key value, or finding the locations of all records, which satisfy one or more conditions in the data.

**7.Sorting** : Arranging the data elements in some logical order, i.e., in ascending and descending order.

**8.Merging** : Combine the data elements in two different sorted sets into a single sorted set.

## ABSTRACT DATATYPE

The Abstract data type is special kind of data type, whose behavior is defined by a set of values and set of operations.

The keyword "Abstract" is used as we can use these data types, we can perform different operations. But how those operations are working that is totally hidden from the user. The ADT is made of with primitive data types, but operation logics are hidden.

## Example

1. List ADT
2. Stack ADT
3. Queue ADT

### Let us see some operations of those mentioned ADT

**List ADT**

- insert(x) - This function is used to insert one element into the list
- delete(x) - This function is used to remove given element from the list
- display() - This function is used to display elements of the list

**Stack ADT**

- push(x) - This is used to push x into the stack
- pop() - This is used to delete one element from top of the stack
- display() - This function is used to display elements of the stack
- peek() - This is used to get the top most element of the stack
- isFull() - This is used to check whether stack is full or not
- isEmpty() - This is used to check whether stack is empty or not
- size() - This function is used to get number of elements present into the stack

**Queue ADT**

- insert(x) - This is used to add x into the queue at the rear end
- delete() - This is used to delete one element from the front end of the queue
- display() - This function is used to display elements of the queue
- isEmpty() - This is used to check whether queue is empty or not
- isFull() - This is used to check whether queue is full or not
- size() - This function is used to get number of elements present into the queue

## LINKED LIST :

- Linked list is one of the fundamental data structures in C.
- The linked list is a linear data structure that contains a sequence of elements such that each element links to its next element in the sequence.
- Linked list is a dynamic data structure whose length can be increased or decreased at run time.
- A **linked list** is used to store a collection of elements. Each element is stored in a linked list is called "Node".

**Node:**


- Each “Node" contains two fields: Data field and Next field.
- Data field is used to store actual value of the node
- Next field is used to store the address of next node in the list.

## TYPES OF LINKED LIST

- Single Linked List
- Double Linked List
- Circular Linked List

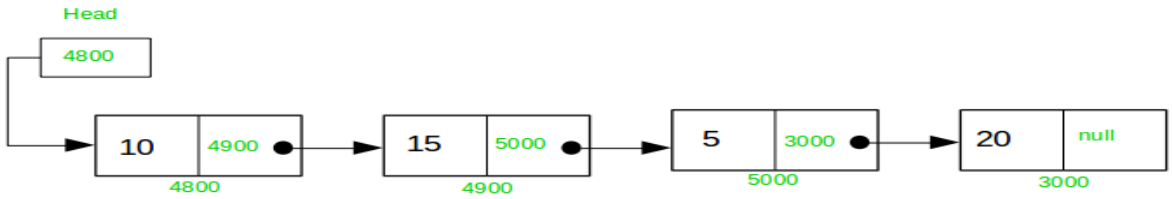
## SINGLE LINKED LIST :

- Single linked list is a sequence of elements in which every element has link to its next element in the sequence.
- In Single linked List, each node contains two fields : Data field and Next field .
- Data field is used to store actual value of the node.
- Next field is used to store the address of next node in the list.
- That is, each node has a single pointer to the next node, and in the last node’s case a null pointer representing that there are no more nodes in the linked list.

## Important Points to be remembered:

- The first node is called the **head**; it points to the first node of the list and helps us access every other element in the list.
- Always next part of the last node must be NULL.

## REPRESENTATION OF LINKED LIST




- In Single Linked List Navigation only forward direction.
- The first node is called the **head**; it points to the first node of the list and helps us access every other element in the list.
- The last node, also sometimes called the **tail**, points to *NULL* which helps us in determining when the list ends.

### Basic Structure of Node is represented as:

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

### Operations of Single Linked List :

1. Creation
2. Insertion
3. Deletion
4. Display (or ) Traversing
5. Searching

**Before we implement actual operations, first we need to setup empty list. First perform the following steps before implementing actual operations.**

**Step 1 -** Include all the **header files** which are used in the program.

**Step 2 -** Declare all the **user defined** functions.

**Step 3 -** Define a **Node** structure with two members **data** and **next**

**Step 4 -** Define a Node pointer '**head**' and set it to **NULL**.

**Step 5 -** Implement the main method by displaying operations menu and make suitable function calls in the main method to perform user selected operation.

## 1. Creation:

### Algorithm:

**Step 1 -** Create a newNode with given value, Say "ptr".

**Step 2-** Check whether list is **Empty** (**head** == **NULL**).

**Step 3-** If it is **Empty** then, set **head** = **ptr** and define a Node pointer '**temp**' and initialize with '**head**'.

**Step 4-** If it is **Not Empty** then, set **temp->next=ptr, temp= temp->next .**

### Program:

```
void create()
{
        struct node *ptr;
        int i,n,val;
        printf("Enter Number of Elements\n");
        scanf("%d",&n);
        for(i=0;i<n;i++)
        {
                ptr=(struct node*)malloc(sizeof(struct node));
                printf("Enter the data of node %d: ", i);
                scanf("%d",&val);
                ptr->data=val;
                ptr->next=NULL;
                if(head==NULL)
                {
                        head=ptr;
                        temp=head;
                }
                else
                {
                        temp->next=ptr;
                        temp=temp->next;
                }
        }
}
```

**2.Insertion : In a single linked list, the insertion operation can be performed in three ways. They are as follows...**

1. Inserting At Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list

**1.Inserting At Beginning of the list :**

**Algorithm :**

Step 1 - Create a **newNode** with given value, Say "**ptr** ".

Step 2 - Check whether list is **Empty** (**head** == **NULL**)

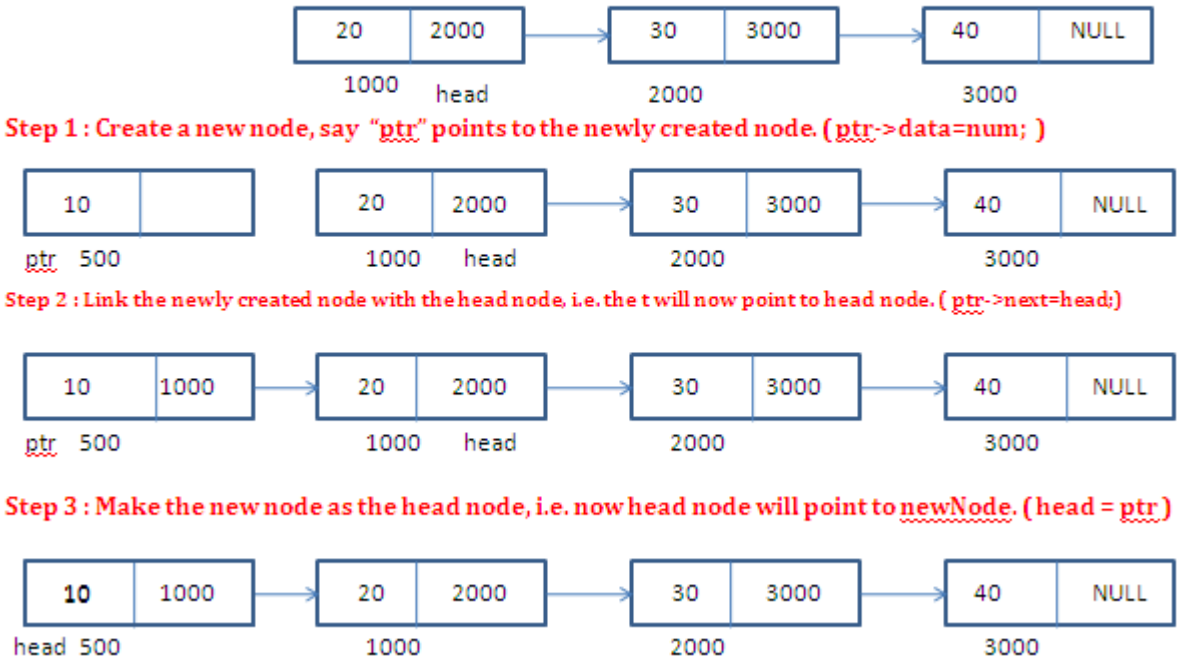
Step 3 - If it is **Empty** then, set **ptr→next** = **NULL** and **head** = **ptr**.

Step 4 - If it is **Not Empty** then, set **ptr→next** = **head** and **head** = **ptr**.

**Program**

```
void insert_beg()
{
        struct node *ptr;
        int num;
        ptr=(struct node*)malloc(sizeof(struct node));
        printf("Enter data:");
        scanf("%d",&num);
        ptr->data=num;
        if(head==NULL)
        {
                ptr->next=NULL;
                 head=ptr;
        }
        else
        {
                ptr->next=head;
                head=ptr;
        }
}
```

# Inserting at Begining




## 2. Inserting At End of the list

## Algorithm:

Step 1 - Create a **newNode** with given value ,Say " **ptr** " and **ptr→ next** as **NULL**.

Step 2 - Check whether list is **Empty** (**head** == **NULL**).

Step 3 - If it is **Empty** then, set **head** = **ptr**.

Step 4 - If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.

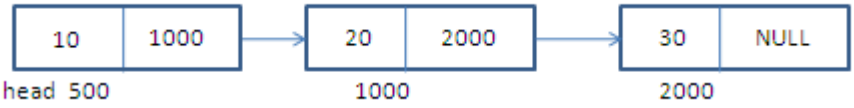
Step 5 - Keep moving the **temp** to its next node until it reaches to the last node in the list .
(until **temp → next** is equal to **NULL**).

Step 6 - Set **temp → next** = **ptr**.

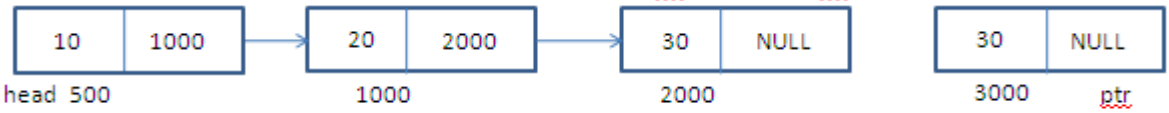
**Program:**

```
void insert_end()
{
        struct node *ptr;
        int num;
        ptr=(struct node*)malloc(sizeof(struct node));
        printf("Enter data:");
        scanf("%d",&num);
        ptr->data=num;
        ptr->next=NULL;
        if(head==NULL)
        {
                head=ptr;
        }
        else
        {
                struct node *temp;
                temp=head;
                while(temp->next!=NULL)
                {
                        temp=temp->next;
                }
                temp->next=ptr;
        }
}
```

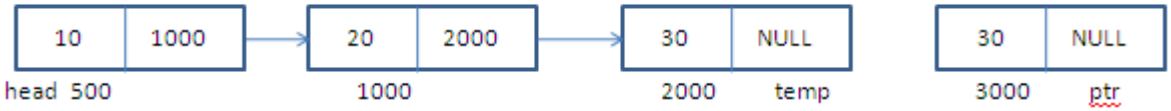
# Inserting at Ending



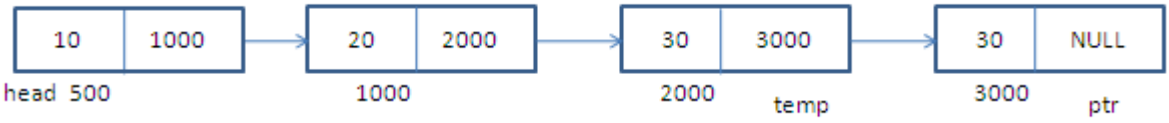

Step 1 : Create a new node, say t points to the newly created node. ( ptr-> data=num; ptr->next=NULL )



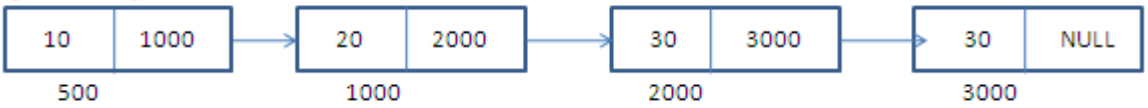

Step2 : Traverse to the last node of the linked list ( temp=head; temp=temp->next )




Step 3: Connect the last node of the list with the new node, i.e. last node will now point to new node. ( temp->next = ptr ).




Step 4 : Finally The linked list is




## 3. Inserting At Specific location in the list (After a Node)

## Algorithm:

Step 1 - Create a newNode with given value,Say **' ptr '.**

Step 2 - Check whether list is **Empty (head == NULL)**

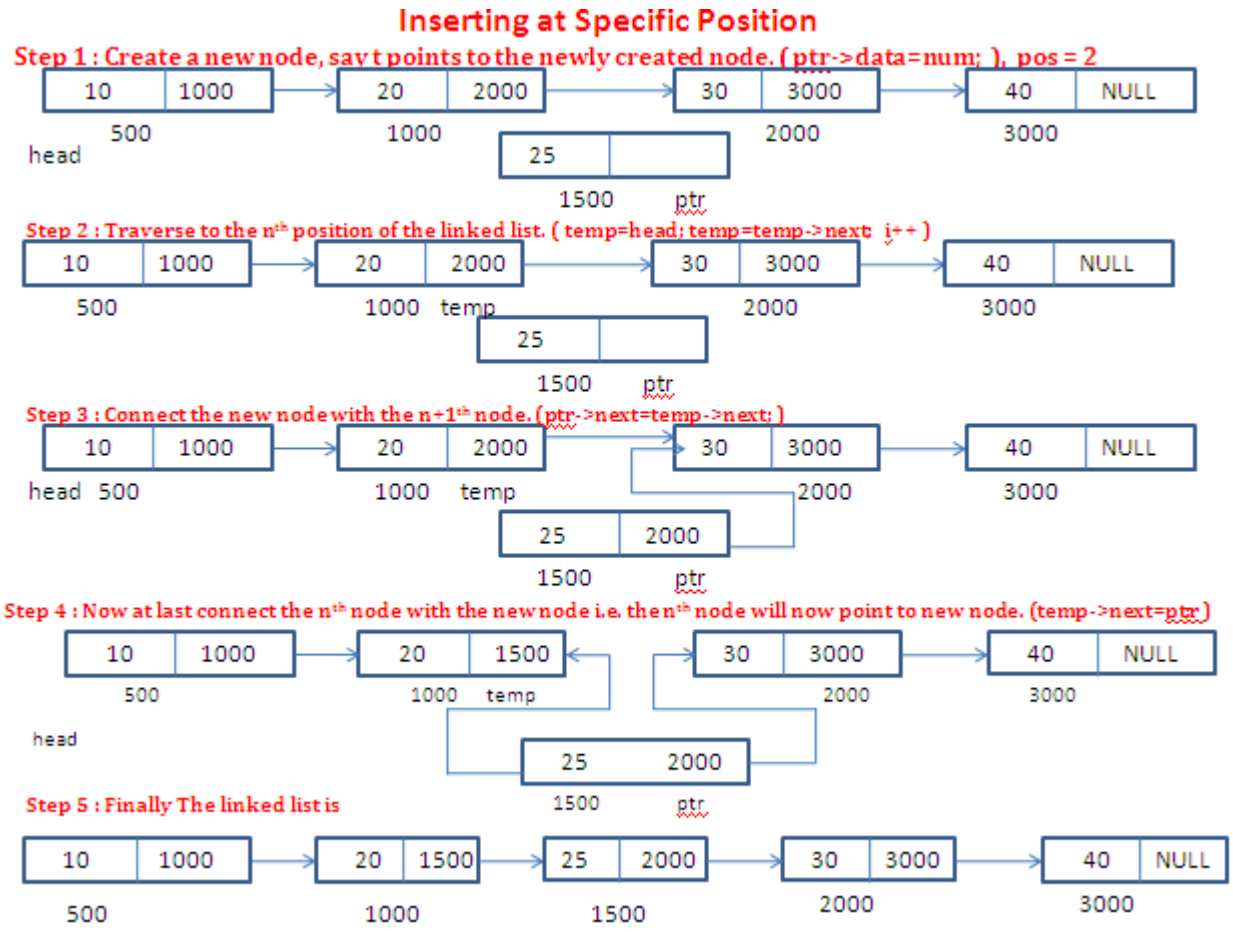
Step 3 - If it is **Empty** then, set **ptr->next=NULL** and **head = ptr**.

Step 4 - If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**. **( temp=head )**

Step 5 - Keep moving the **temp** to its next node until it reaches to the node after which we want to insert the newNode

Step 6 - Finally, Set **'ptr → next = temp→ next'** and **'temp → next = ptr'**

```
int insert_pos()
{
        struct node *ptr;
        int pos,i=1,num;
        ptr=(struct node*)malloc(sizeof(struct node));
        printf("Enter data:");
        scanf("%d",&num);
        ptr->data=num;
        printf("Enter position to insert:");
        scanf("%d",&pos);
        if(head==NULL)
        {
                ptr->next=NULL;
                head=ptr;
        }
        else
        {
                struct node *temp;
                temp=head;
                while(i<pos-1)
                {
                        temp=temp->next;
                        i++;
                }
                ptr->next=temp->next;
                temp->next=ptr;
        }
   return 0;
}
```




**3. Deletion : In a single linked list, the deletion operation can be performed in three ways. They are as follows...**

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node

**1. Deleting from Beginning of the list**

**Algorithm:**

Step 1 - Check whether list is **Empty** (**head** == **NULL**)

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 **head**.

Step 4 - Check whether list is having only one node (**temp → next** == **NULL**)

Step 5 - If it is **TRUE** then set **head** = **NULL** and delete **temp** (Setting **Empty** list conditions)

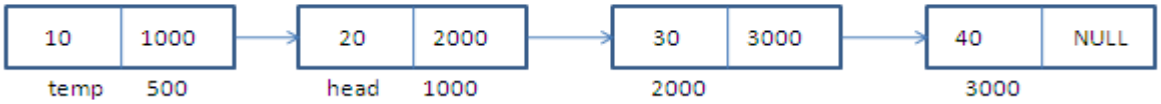
Step 6 - If it is **FALSE** then set **head** = **temp → next**, and delete **temp**.

**Program:**

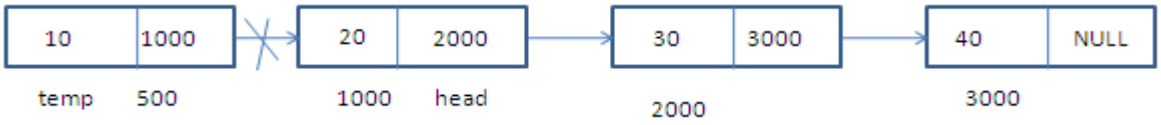
```
void delete_beg()
{
        if(head==NULL)
        {
                printf("List is Empty!!! Deletion is not possible\n");
        }
        else
        {
                struct  node *temp;
                temp=head;
                if(temp->next==NULL)
                {
                        head=NULL;
                        printf("Deleted element is %d",temp->data);
                        free(temp);
                }
                else
                {
                        head=temp->next;
                        printf("Deleted element is %d",temp->data);
                        free(temp);
                }
        }
}
```

# Deletion at Begining

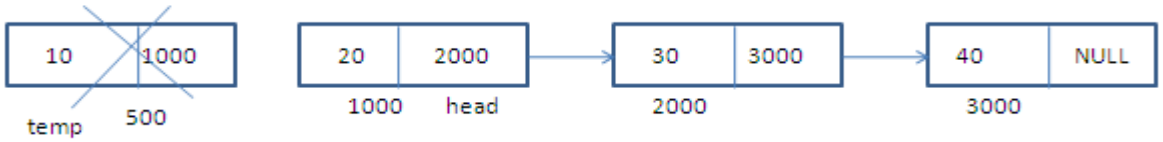

Step 1 : Make the head points to the next node. ( temp=head; head=temp->next; )



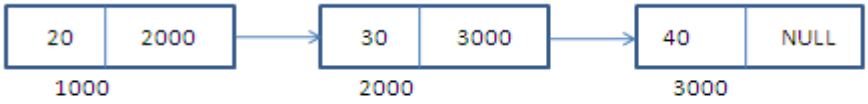

Step 2 : Disconnect the connection of first node to second node.




Step 3 . Free the memory occupied by the first node.. free(temp)




Step 4 : Finally, the new linked list.




## 2. Deleting from End of the list

### Algorithm:

Step 1. Check whether list is **Empty** (**head** == **NULL**)

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 two Node pointers '**temp**' and '**temp1**' and initialize '**temp**' with **head**.

Step 4. Check whether list has only one Node (**temp → next** == **NULL**)

Step 5. If it is **TRUE**. Then, set **head** = **NULL** and delete **temp**. And terminate the function.

Step 6. If it is **FALSE**. Then, set '**temp1 = temp** ' and move **temp** to its next node. Repeat the same until it reaches to the last node in the list. (until **temp → next** == **NULL**)

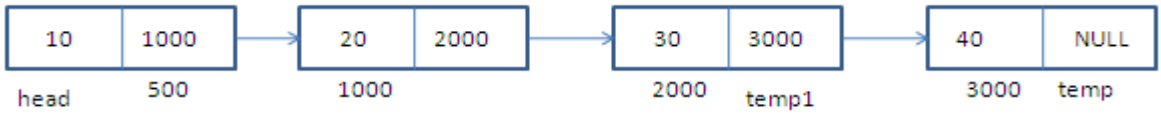
Step 7. Finally, Set **temp1 → next** = **NULL** and delete **temp**.

**Program:**

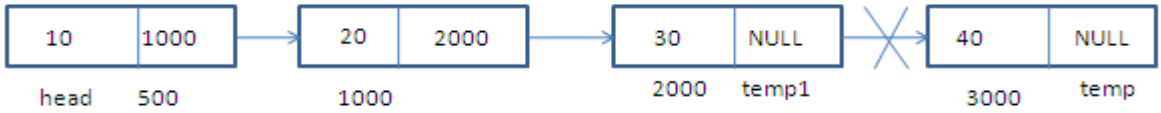
```
void delete_end()
{
        if(head==NULL)
        {
                printf("List is Empty!!! Deletion is not possible\n");
        }
        else
        {
                struct node *temp,*temp1;
                 temp=head;
                if(temp->next==NULL)
                {
                        head=NULL;
                        printf("Deleted element is %d",temp->data);
                        free(temp);
                }
                else
                {
                        while(temp->next!=NULL)
                        {
                                temp1=temp;
                                temp=temp->next;
                        }
                        temp1->next=NULL;
                        printf("Deleted element is %d",temp->data);
                        free(temp);
                }
        }
}
```

# Deletion at Ending

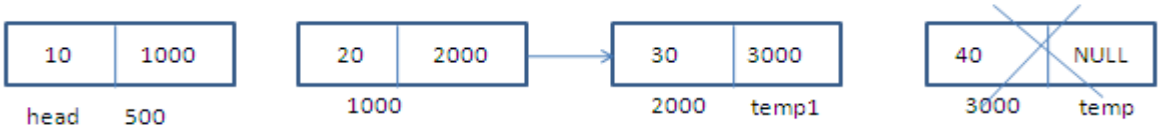

Step 1 Traverse to the last node of the linked list keeping track of the second last node in some variable say temp1



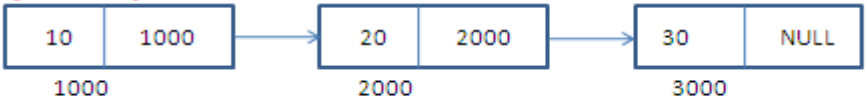

Step 2 : Disconnect the second last node with the last node i.e. temp1->next=NULL;




Step 3 . Free the memory occupied by the last node.. free(temp)




Step 4 : Finally, the new linked list.




## 3. Deleting a Specific Node from the list

**Algorithm:**

**Step 1** - Check whether list is **Empty (head == NULL)**

**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 two Node pointers **'temp' and 't'** and initialize **'temp'** with **head.**

**Step 4**- Traverse the **$n^{th}$ Node**.

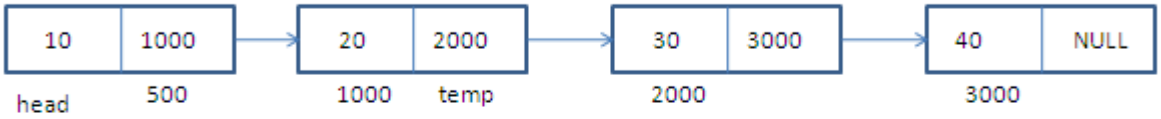
**Step 5-** Finally, Set **t=temp->next; and temp->next=t->next; t->next=NULL**.and **Delete " t"**

**Program:**

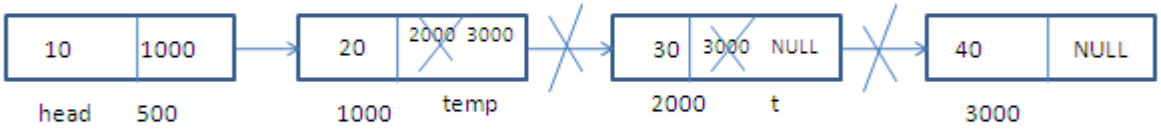
```
int delete_pos()
{
        int pos,i=1;
        if(head==NULL)
        {
                printf("List is Empty!!! Deletion is not possible\n");
        }
        else
        {
                struct node *temp,*t;
                temp=head;
                printf("Enter position to delete:");
                scanf("%d",&pos);
                if(temp->next==NULL)
                {
                        head=NULL;
                        printf("Deleted element is %d",temp->data);
                        free(temp);
                }
                else
                {
                        while(i<pos-1)
                        {
                                temp=temp->next;
                                i++;
                        }
                        t=temp->next;
                        temp->next=t->next;
                        t->next=NULL;
                        printf("Deleted element is %d",t->data);
                        free(t);
                }
        }
   return 0;
}
```

# Deletion at Specified Node

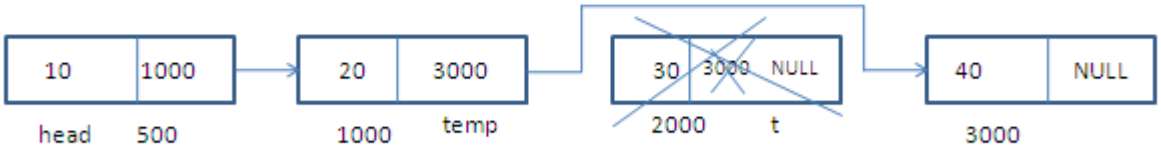

**Step 1 . Traverse to the n$^{th}$ node of the singly linked list. ( temp=head; temp=temp->next; pos=3 )**



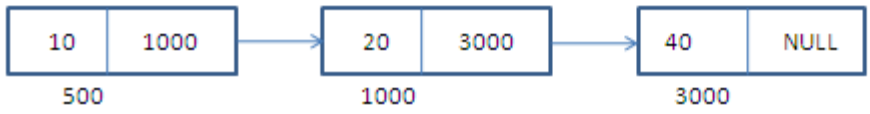

**Step 2 : Disconnect the Specified node ( t=temp->next; temp ->next=t>next; t->next=NULL; )**




**Step 3 . Free the memory occupied by the Specified node.. free(t)**




**Step 4 : Finally, the new linked list.**



## 4. Displaying a Single Linked List

### Algorithm :

**Step 1** - Check whether list is **Empty** (**head** == **NULL**)

**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 **head**.

**Step 4** - Keep displaying **temp → data** with an arrow (**->**) until **temp** reaches to the last node

**Step 5** - Finally, display **temp → data** with arrow pointing to **NULL** .

### Program:

```
void display()
{
        if(head==NULL)
        {
                printf("List is Empty!!!\n");
        }
        else
        {
                struct Node *temp;
                temp=head;
                printf("The linked list is:\n");
                while(temp!=NULL)
                {
                        printf("%d->",temp->data);
                        temp=temp->next;
                }
                printf("NULL");
        }
}
```

## 5. Searching:

### Algorithm :

1. Initialize a node pointer, **temp=head**.
2. Input element to search from user. Store it in some variable say "**key**".
3. Declare a variable to store index of found element through list, say "**count=0".**
4. Do following while temp is not NULL
   - **temp->data == key** , if the condition is true, The element is found.
   - otherwise ,increment count (count++) and move temp to its next node **(temp = temp->next )**

5.while temp is NULL, The element is not found.

### Program :

```
int searchNode()
{
        struct node *temp = head;
        int key,count=0;
        printf("\nEnter the element to be searched in the list : ");
        scanf("%d",&key);
        while(temp != NULL)
        {
                if(temp->data == key)
                {
                        printf("\nElement %d found at position %d",key,count);
                        return 0;
                }
                else
                {
                        count+=1;
                        temp = temp->next;
                }
        }
        printf("\n Element %d is not found in the list\n",key);
        return 0;
}
```

## DOUBLE LINKED LIST

### What is Double Linked List?

- Double linked list is a sequence of elements in which every element has links to its previous element and next element in the sequence.
- In a double linked list, every node has a link to its previous node and next node.
- So, we can traverse forward by using the next field and can traverse backward by using the previous field.
- Every node in a double linked list contains three fields and they are shown in the following figure...

**NODE**

| previous | Data | next |
|---|---|---|

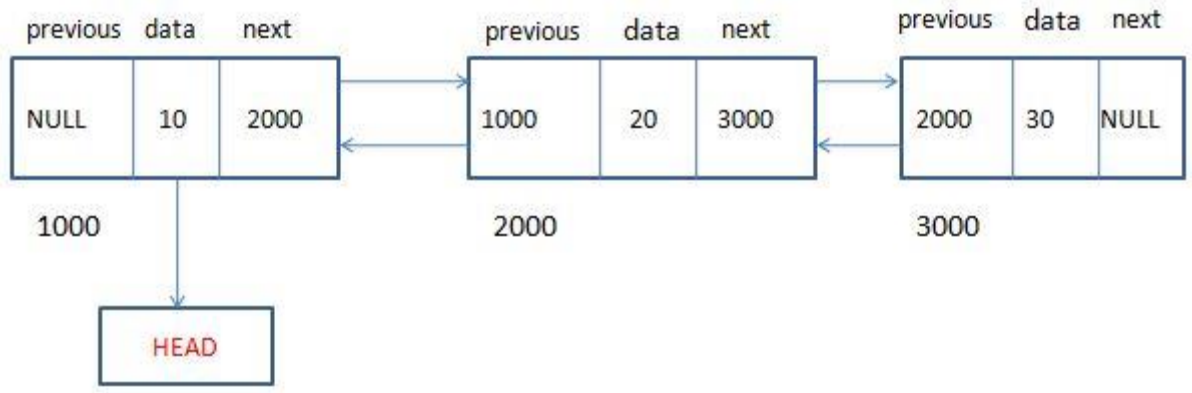
**Here,**

**1. 'previous'** : Link1 field is used to store the address of the **previous node** in the sequence,

**2. 'Data'** : Data field is used to store the actual value of that node.

**3. 'next' :** Link2 field is used to store the address of the **next node** in the sequence

### REPRESENTATION OF DOUBLE LINKED LIST




### Important Points to be Remembered

- In double linked list, the first node must be always pointed by head.
- Always the previous field of the first node must be NULL.
- Always the next field of the last node must be NULL.

## Basic structure of a Double Linked List

```
struct node
{
        int data;
        struct  node  *previous;
        struct  node  *next;
};
```

## Operations on Double Linked List

In a double linked list, we perform the following operations...

1. Create
2. Insertion
3. Deletion
4. Display
5. Searching

**Before we implement actual operations, first we need to setup empty list. First perform the following steps before implementing actual operations.**

**Step 1 -** Include all the **header files** which are used in the program.

**Step 2 -** Declare all the **user defined** functions.

**Step 3 -** Define a **Node** structure with **Three** members **previous, data** and **next**

**Step 4 -** Define a Node pointer '**head**' and set it to **NULL**.

### 1. Creation:

**Step 1 -** Create a newNode with given value, Say **"ptr".**

**Step 2-** Check whether list is **Empty** (**head** == **NULL**).

**Step 3-** If it is **Empty** then, set **head** = **ptr** and define a Node pointer '**temp**' and initialize with '**head**'.

**Step 4-** If it is **Not Empty** then, set **temp->next=ptr, ptr->previous=temp** and **temp =ptr.**

**Program:**

```
void create()
{
        struct node *ptr,*temp;
        int i,n,val;
        printf("Enter Number of Elements :");
        scanf("%d",&n);
        for(i=0;i<n;i++)
        {
                ptr=(struct node*)malloc(sizeof(struct node));
                printf("Enter the data of node %d: ", i);
                scanf("%d",&val);
                ptr->data=val;
                ptr->previous=NULL;
                ptr->next=NULL;
                if(head==NULL)
                {
                        head=ptr;
                        temp=head;
                }
                else
                {
                        temp->next=ptr;
                        ptr->previous=temp;
                        temp=ptr;
                }
        }
}
```

## 2. Insertion :

**In a double linked list, the insertion operation can be performed in three ways as follows...**

1. Inserting At Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list

### 1. Inserting At Beginning of the list

**Algorithm :**

**Step 1** - Create a **newNode** with given value ,Say " **ptr** " and **ptr → previous** as **NULL**.

**Step 2** - Check whether list is **Empty** (**head** == **NULL**)

**Step 3** - If it is **Empty** then, assign **NULL** to **ptr → next** and **ptr** to **head**.

**Step 4** - If it is **not Empty** then, assign **head** to **ptr → next** , **ptr** to **head→ previous** and **ptr** to **head**.

**Program:**

```
void insert_beg()
{
        struct node *ptr;
        int num;
        ptr=(struct node*)malloc(sizeof(struct node));
        printf("Enter data:");
        scanf("%d",&num);
        ptr->data=num;
        ptr->previous=NULL;
        if(head==NULL)
        {
                ptr->next = NULL;
                head = ptr;
        }
        else
        {
                ptr->next = head;
                head->previous=ptr;
                head = ptr;
        }
}
```

## 2. Inserting At End of the list

**Algorithm :**

**Step 1** - Create a **newNode** with given value ,Say **" ptr "** and **ptr → next** as **NULL**.

**Step 2** - Check whether list is **Empty** (**head** == **NULL**)

**Step 3** - If it is **Empty**, then assign **NULL** to **ptr → previous** and **ptr** to **head**.

**Step 4** - If it is **not Empty**, then, define a node pointer **temp** and initialize with **head**.

**Step 5** - Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp → next** is equal to **NULL**).

**Step 6 -** Assign **ptr** to **temp → next** and **temp** to **ptr → previous**.

**Program:**

```
void insert_end()
{
        struct node *ptr;
        int num;
        ptr=(struct node*)malloc(sizeof(struct node));
        printf("Enter data:");
        scanf("%d",&num);
        ptr->data=num;
        ptr->next=NULL;
        if(head==NULL)
        {
                ptr->previous=NULL;
                head=ptr;
        }
        else
        {
                struct Node *temp;
                temp=head;
                while(temp->next!=NULL)
                {
                        temp=temp->next;
                }
                temp->next=ptr;
                ptr->previous=temp;
        }
}
```

### 3. Inserting At Specific location in the list (After a Node)

**Algorithm :**

**Step 1** - Create a **newNode** with given value, Say **" ptr "**.

**Step 2** - Check whether list is **Empty** (**head** == **NULL**)

**Step 3** - If it is **Empty** then, assign **NULL** to both **ptr → previous** & **ptr → next** and set **ptr** to **head**.

**Step 4** - If it is **not Empty** then, define two node pointers **temp1** & **temp2** and initialize **temp1** with **head**.

**Step 5** - Assign **temp1 → next** to **temp2**, **ptr** to **temp1 → next**, **temp1** to **ptr → previous**, **temp2** to **ptr → next** and **ptr** to **temp2 → previous**.

**Program:**

```
int insert_pos()
{
        struct node *ptr;
        int pos,i=1,num;
        ptr=(struct node*)malloc(sizeof(struct node));
        printf("Enter data:");
        scanf("%d",&num);
        ptr->data=num;
        printf("Enter position to insert:");
        scanf("%d",&pos);
        if(head==NULL)
        {
                ptr->previous=NULL;
                ptr->next=NULL;
                head=ptr;
        }
        else
        {
                struct node *temp1,*temp2;
                temp1=head;
```

```
            while(i<pos-1)
            {
                temp1=temp1->next;
                i++;
            }
            temp2=temp1->next;
            temp1->next=ptr;
            ptr->previous=temp1;
            ptr->next=temp2;
            temp2->previous=ptr;
            return 0;
        }
}
```

**3. Deletion: In a double linked list, the deletion operation can be performed in three ways**

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node

**1. Deleting from Beginning of the list**

**Algorithm:**

**Step 1** - Check whether list is **Empty** (**head** == **NULL**)

**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 **head**.

**Step 4** - Check whether list is having only one node (**temp → previous** is equal to **temp → next**)

**Step 5** - If it is **TRUE**, then set **head** to **NULL** and delete **temp** (Setting **Empty** list conditions)

**Step 6** - If it is **FALSE**, then assign **temp → next** to **head**, **NULL** to **head → previous** and delete **temp**.

**Program:**

```
void delete_beg()
{
        if(head==NULL)
        {
                printf("List is Empty!!! Deletion is not possible\n");
        }
        else
        {
                struct node *temp;
                temp=head;
                if(temp->previous==temp->next)
                {
                        head=NULL;
                        printf("Deleted element is %d",temp->data);
                        free(temp);
                }
                else
                {
                        head =temp->next;
                        head->previous=NULL;
                        printf("Deleted element is %d",temp->data);
                        free(temp);
                }
        }
}
```

## 2. Deleting from End of the list

**Algorithm :**

**Step 1** - Check whether list is **Empty (head == NULL)**

**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 **head.**

**Step 4 -** Check whether list has only one Node (**temp → previous** and **temp → next** both are **NULL**)

**Step 5** - If it is **TRUE**, then assign **NULL** to head and delete **temp**. And terminate from the function.

(Setting Empty list condition)

**Step 6 -** If it is **FALSE,** then keep moving **temp** until it reaches to the last node in the list.

(**until temp → next is equal to NULL**)

**Step 7** - Assign **NULL** to **temp1 → next** and delete **temp**.

**Program:**

```
void delete_end()
{
        if(head==NULL)
        {
                printf("List is Empty!!! Deletion is not possible\n");
        }
        else
        {
                struct node *temp,*temp1;
                temp=head;
                if(temp->previous==temp->next)
                {
                        head=NULL;
                        printf("Deleted element is %d",temp->data);
                        free(temp);
                }
                else
                {
                        while(temp->next!=NULL)
                        {
                                temp1=temp;
                                temp=temp->next;
                        }
                        temp1->next=NULL;
                        printf("Deleted element is %d",temp->data);
                        free(temp);
                }
        }
}
```

### 3. Deleting a Specific Node from the list

**Algorithm:**

**Step 1** - Check whether list is **Empty** (**head** == **NULL**)

**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 **head**.

**Program:**

```
int delete_pos()
{
        int pos,i=1;
        if(head==NULL)
        {
                printf("List is Empty!!! Deletion is not possible\n");
        }
        else
        {
                struct node *temp,*t;
                temp=head;
                printf("Enter position to delete:");
                scanf("%d",&pos);
                if(temp->previous==temp->next)
                {
                        head=NULL;
                        printf("Deleted element is %d",temp->data);
                        free(temp);
                }
                else
                {
                        while(i<pos-1)
                        {
                                temp=temp->next;
                                i++;
                        }
                        t=temp->next;
                        temp->next=t->next;
                        t->next=NULL;
                        t->previous=NULL;
                        temp->next->previous=temp;
                        printf("Deleted element is %d",t->data);
                        free(t);
                }
        }
return 0;
}
```

## 4. Displaying a Double Linked List

**Algorithm :**

**Step 1** - Check whether list is **Empty** (**head** == **NULL**)

**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 **head**.

**Step 4** - Keep displaying **temp → data** with an arrow (**->**) until **temp** reaches to the last node

**Step 5** - Finally, display **temp → data** with arrow pointing to **NULL** .

**Program:**

```
void display()
{
        if(head==NULL)
        {
                printf("List is Empty!!!\n");
        }
        else
        {
                struct Node *temp;
                temp=head;
                printf("The linked list is:\n");
                while(temp!=NULL)
                {
                        printf("%d->",temp->data);
                        temp=temp->next;
                }
                printf("NULL");
        }
}
```

## 5. Searching :

## Algorithm :

1. Initialize a node pointer, **temp=head**.

2. Input element to search from user. Store it in some variable say "**key**".

3. Declare a variable to store index of found element through list, say "**count=0".**

4. Do following while temp is not NULL

- **temp->data == key** , if the condition is true, The element is found.
- otherwise ,increment count ( count++) and move temp to its next node **(temp = temp->next )**

5.while temp is NULL, The element is not found.

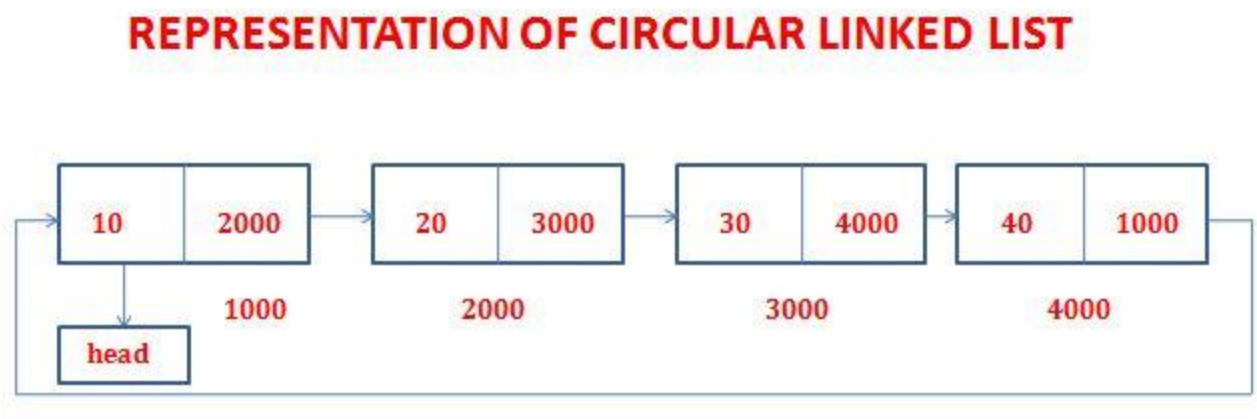
## Program :

```
int searchNode()
{
        struct node *temp = head;
        int key,count=0;
        printf("\nEnter the element to be searched in the list : ");
        scanf("%d",&key);
        while(temp != NULL)
        {
                if(temp->data == key)
                {
                        printf("\nElement %d found at position %d",key,count);
                        return 0;
                }
                else
                {
                        count+=1;
                        temp = temp->next;
                }
        }
        printf("\n Element %d is not found in the list\n",key);
        return 0;
}
```

# CIRCULAR LINKED LIST

## What is Circular Linked List?

- A circular linked list is a sequence of elements in which every element has a link to its next element in the sequence and the last element has a link to the first element.
- That means circular linked list is similar to the single linked list except that the last node points to the first node in the list




In **single linked list**, every node points to its next node in the sequence and the last node points NULL. But in **circular linked list**, every node points to its next node in the sequence but the last node points to the first node in the list.

## Operations of Circular Linked List :

1. Creation
2. Insertion
3. Deletion
4. Display (or ) Traversing
5. Searching

**Before we implement actual operations, first we need to setup empty list. First perform the following steps before implementing actual operations.**

**Step 1 -** Include all the **header files** which are used in the program.

**Step 2 -** Declare all the **user defined** functions.

**Step 3 -** Define a **Node** structure with two members **data** and **next**

**Step 4 -** Define a Node pointer '**head**' and set it to **NULL**.

**1. Creation :**

**Algorithm:**

**Step 1 -** Create a newNode with given value, Say "ptr".

**Step 2-** Check whether list is **Empty** (**head** == **NULL**).

**Step 3-** If it is **Empty** then, set **head** = **ptr** and define a Node pointer '**temp**' and initialize with '**head**'.

**Step 4-** If it is **Not Empty** then, set **temp->next=ptr, temp=ptr** and **temp->next=head.**

**Program:**

```
void create()
{
        struct node *ptr,*temp;
        int i,n,val;
        printf("Enter Number of Elements\n");
        scanf("%d",&n);
        for(i=0;i<n;i++)
        {
                ptr=(struct node*)malloc(sizeof(struct node));
                printf("Enter the data of node %d: ", i);
                scanf("%d",&val);
                ptr->data=val;
                ptr->next=NULL;
                if(head==NULL)
                {
                        head=ptr;
                        temp=head;
                }
                else
                {
                        temp->next=ptr;
                        temp=ptr;
                        temp->next=head;
                }
        }
}
```

## 2. Insertion

**In a circular linked list, the insertion operation can be performed in three ways. They are as follows...**

1. Inserting At Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list

## 1. Inserting At Beginning of the list

**Algorithm :**

**Step 1 -** Create a **newNode** with given value , **Say " ptr ".**

**Step 2 -** Check whether list is **Empty** (**head** == **NULL**)

**Step 3 -** If it is **Empty** then, set **head** = **ptr** and **ptr→next** = **head** .

**Step 4 -** If it is **Not Empty** then, define a Node pointer '**temp**' and initialize with '**head**'.

**Step 5 -** Keep moving the '**temp**' to its next node until it reaches to the last node

(until '**temp → next** == **head**').

**Step 6 -** Set '**ptr → next** =**head**', '**head** = **ptr**' and '**temp → next** = **head**'.

**Program:**

```
void insert_beg()
{
        struct node *ptr,*temp;
        int num;
        ptr=(struct node*)malloc(sizeof(struct node));
        printf("Enter data:");
        scanf("%d",&num);
        ptr->data=num;
        if(head==NULL)
        {
                head=ptr;
                ptr->next=head;
        }
```

```
        else
        {
                temp=head;
                if(temp->next==head)
                {
                        temp->next=ptr;
                        ptr->next=temp;
                }
                else
                {
                        while(temp->next!=head)
                        {
                                temp=temp->next;
                        }
                        ptr->next=head;
                        head=ptr;
                        temp->next=head;
                }
        }
}
```

## 2. Inserting At End of the list

### Algorithm :

Step 1 - Create a **newNode** with given value.

Step 2 - Check whether list is **Empty (head == NULL).**

Step 3 - If it is **Empty** then, set **head = ptr** and **ptr → next = head**.

Step 4 - If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.

Step 5 - Keep moving the **temp** to its next node until it reaches to the last node in the list

**(until temp → next == head).**

Step 6 - Set **temp → next = ptr** and **ptr → next = head**.

**Program:**

```
void insert_end()
{
        struct node *ptr,*temp;
        int num;
        ptr=(struct node*)malloc(sizeof(struct node));
        printf("Enter data:");
        scanf("%d",&num);
        ptr->data=num;
        if(head==NULL)
        {
                head=ptr;
                ptr->next=head;
        }
        else
        {
                temp=head;
                if(temp->next==head)
                {
                        temp->next=ptr;
                        ptr->next=temp;
                }
                else
                {
                        while(temp->next!=head)
                        {
                                temp=temp->next;
                        }
                        temp->next=ptr;
                        ptr->next=head;
                }
        }
}
```

### 3. Inserting At Specific location in the list (After a Node)

**Algorithm :**

Step 1 - Create a newNode with given value,Say **' ptr '.**

Step 2 - Check whether list is **Empty (head == NULL)**

Step 3 - If it is **Empty** then, set **head = ptr** and **ptr->next=head**.

Step 4 - If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**. **( temp=head )**

Step 5 - Keep moving the **temp** to its next node until it reaches to the node after which we want to insert the newNode

Step 6 - Finally, Set **'ptr → next = temp→ next'** and **'temp → next = ptr'**

**Program:**

```
void  insert_pos()
{
        struct node *ptr;
        int pos,i=1,num;
        ptr=(struct node*)malloc(sizeof(struct node));
        printf("Enter data:");
        scanf("%d",&num);
        ptr->data=num;
        printf("Enter position to insert:");
        scanf("%d",&pos);
        if(head==NULL)
        {
                head=ptr;
                ptr->next=head;
        }
        else
        {
                struct node *temp;
                temp=head;
                while(i<pos-1)
                {
                        temp=temp->next;
                        i++;
                }
                ptr->next=temp->next;
                temp->next=ptr;
        }
}
```

## 3.Deletion

**In a circular linked list, the deletion operation can be performed in three ways those are as follows...**

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node

### 1.Deleting from Beginning of the list

**Algorithm :**

**Step 1 -** Check whether list is **Empty** (**head** == **NULL**)

**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 two Node pointers **'temp'** and '**temp1**' and initialize both '**temp**' and '**temp1**' with **head**.

**Step 4 -** Check whether list is having only one node (**temp → next** == **head**)

**Step 5 -** If it is **TRUE** then set **head** = **NULL** and delete **temp** (Setting **Empty** list conditions)

**Step 6 -** If it is **FALSE** move the **temp** until it reaches to the last node. (until **temp → next** == **head** )

**Step 7 -** Then set **head** = **temp1 → next**, **temp → next** = **head** and delete **temp1**.

**Program:**

```
void delete_beg()
{
        if(head==NULL)
        {
                printf("List is Empty!!! Deletion is not possible\n");
        }
        else
        {
                struct node *temp,*temp1;
                temp=head;
                temp1=head;
                 if(temp->next==head)
```

```
                {
                        head=NULL;
                        printf("Deleted element is %d",temp->data);
                        free(temp);
                }
                else
                {
                        while(temp->next!=head)
                        {
                                temp=temp->next;
                        }
                        head=temp1->next;
                        temp->next=head;
                        printf("Deleted element is %d",temp1->data);
                        free(temp1);
                }
        }
}
```

**2.Deleting from End of the list**

**Algorithm :**

**Step 1 -** Check whether list is **Empty** (**head** == **NULL**)

**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 two Node pointers **'temp'** and '**temp1'** and initialize '**temp**' with **head**.

**Step 4 -** Check whether list has only one Node (**temp → next** == **head**)

**Step 5 -** If it is **TRUE**. Then, set **head** = **NULL** and delete **temp**. And terminate from the function. (Setting **Empty** list condition)

**Step 6 -** If it is **FALSE**. Then, set '**temp1 = temp** ' and move **temp** to its next node. Repeat the same until **temp** reaches to the last node in the list. (until **temp → next** == **head**)

**Step 7 -** Set **temp1 → next** = **head** and delete **temp**.

**Program:**

```
void delete_end()
{
        if(head==NULL)
        {
                printf("List is Empty!!! Deletion is not possible\n");
        }
        else
        {
                struct node *temp,*temp1;
                temp=head;
                if(temp->next==head)
                {
                        head=NULL;
                        printf("Deleted element is %d",temp->data);
                        free(temp);
                }
                else
                {
                        while(temp->next!=head)
                        {
                                temp1=temp;
                                temp=temp->next;
                        }
                        temp1->next=head;
                        printf("Deleted element is %d",temp->data);
                        free(temp);
                }
        }
}
```

### 3. Deleting a Specific Node from the list

**Algorithm:**

**Step 1** - Check whether list is **Empty (head == NULL)**

**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 two Node pointers **'temp' and 't'** and initialize **'temp'** with **head.**

**Step 4**- Traverse the **$n^{th}$ Node**.

**Step 5**- Finally, Set **t=temp->next; and temp->next=t->next; t->next=NULL**.and **Delete " t"**

**Program:**

```
void  delete_pos()
{
        int pos,i=1;
        if(head==NULL)
        {
                printf("List is Empty!!! Deletion is not possible\n");
        }
        else
        {
                struct node *temp,*t;
                 temp=head;
                printf("Enter position to delete:");
                scanf("%d",&pos);
                if(temp->next==NULL)
                {
                        head=NULL;
                        printf("Deleted element is %d",temp->data);
                        free(temp);
                }
                else
                {
                        while(i<pos-1)
                        {
                                temp=temp->next;
                                i++;
                        }
                        t=temp->next;
                        temp->next=t->next;
                        t->next=NULL;
                        printf("Deleted element is %d",t->data);
                        free(t);
                }
        }
}
```

## 4. Displaying a circular Linked List

**Algorithm:**

**Step 1 -** Check whether list is **Empty** (**head** == **NULL**)

**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 **head**.

**Step 4 -** Keep displaying **temp → data** with an arrow (**--->**) until **temp** reaches to the last node

**Step 5 -** Finally display **temp → data** with arrow pointing to **head → data**.

**Program:**

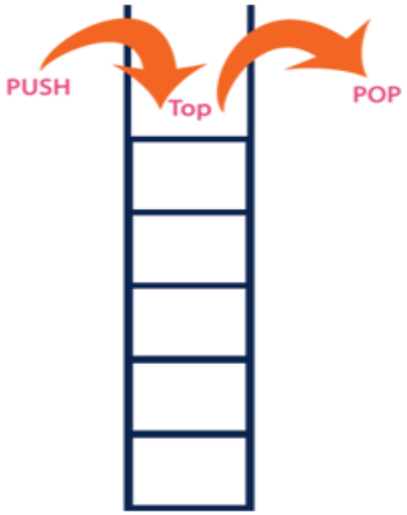
```
void display()
{
        if(head==NULL)
        {
                printf("List is Empty!!!\n");
        }
        else
        {
                struct node *temp;
                temp=head;
                printf("The linked list is:\n");
                while(temp->next!=head)
                 {
                        printf("%d->",temp->data);
                        temp=temp->next;
                }
                printf("%d->%d",temp->data, head->data);
        }
}
```

# STACK ADT

## What is a Stack?

- Stack is a linear data structure
- Stack is a Collection of similar data items in which both insertion and deletion operations are performed based on **LIFO (Last in First Out)** principle.
- In stack, The insertion and deletion operations are performed at only one end is called **"top"**.
- That means, a new element is added at top of the stack and an element is removed from the top of the stack.
- In a stack, the insertion operation is performed using a function called **"push"** and deletion operation is performed using a function called **"pop"**.

## Stack Representation is




## Real life example of stack

A most popular example of stack is plates in marriage party. Fresh plates are **pushed** onto to the top and **popped** from the top.

## STACK TERMINOLOGY

**1. STACKSIZE:** Stack size represents the maximum size of stack.

**2. TOP:** Top represents Top of Stack ( TOP ). The stack top is used to check stack overflow or underflow conditions. Initially top value must be -1. **[ TOP= -1 ]**

**3. STACK OVERFLOW:** When a stack is completely full, it is said to be Stack Overflow.

**4. STACK UNDERFLOW (or) EMPTY:** When a stack is completely empty, it is said to be Stack Underflow (or) Stack Empty.

**Example**

If we want to create a stack by inserting 10, 45,12,16,35 and 50. Then 10 becomes the bottom-most element and 50 is the topmost element. The last inserted element 50 is at Top of the stack as shown in the image below...

**STACK IMPLEMENTATIONS:**

**Stack data structure can be implemented in two ways.**

1. Stack Using Array
2. Stack Using Linked List

## Stack Using Array

A stack data structure can be implemented using a one-dimensional array. But stack implemented using array stores only a fixed number of data values. This implementation is very simple. Just define a one dimensional array of specific size and insert or delete the values into that array by using **LIFO principle** with the help of a variable called **'top'**. Initially, the top is set to -1.

## Stack Operations using Array

1. Push
2. Pop
3. Display

## 1. Push

- In a stack, push () is a function used to insert an element into the stack.
- Whenever we want to insert a value into the stack, increment the top value by one and then insert.
- In a stack, the new element is always inserted at **top** position.

**Algorithm:**

**Step1**: Check whether **stack** is **FULL**. (**top == stacksize -1**)

**Step2: If** it is **FULL**, then display **"Stack is FULL!!! Insertion is not possible!!!"** and Terminate the function.

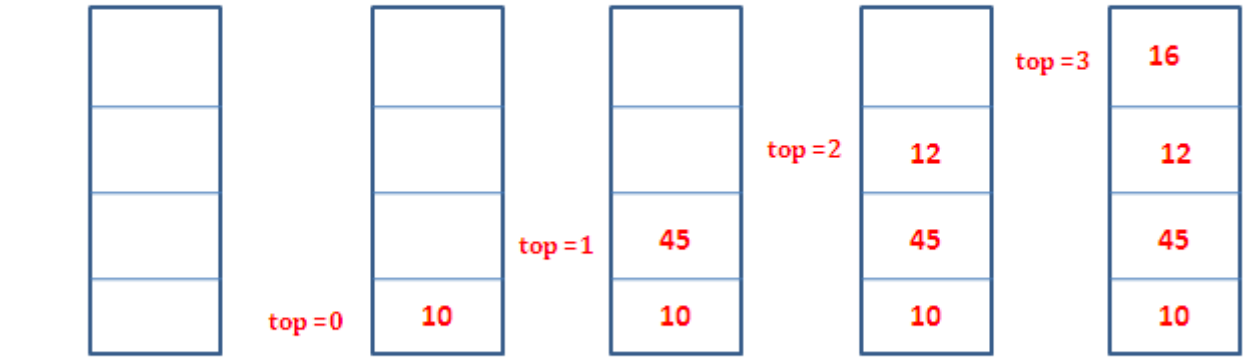
**Step3:**If it is **NOT FULL**, then increment **top** value by one (**top++**) and set stack[top] to value (**stack [top] = value**).

**Program:**

```
void push( )
{
        int value;
        if(top = = stacksize – 1 )
        {
                printf("\nStack is Full!!! Insertion is not possible!!!");
        }
        else
        {
                printf(“Enter the element to be inserted”);
                scanf(“%d”,&value);
                top = top + 1;
                stack [top ] = value;
        }
}
```

## Push Operation(To insert an element on to the stack)

### Stack size=4



## 2. Pop

- In a stack, pop () is a function used to delete an element from the stack.
- Whenever we want to delete a value from the stack, then delete the top value and decrement the top value by one.
- In a stack, the element is always deleted from **top** position.

**Algorithm:**

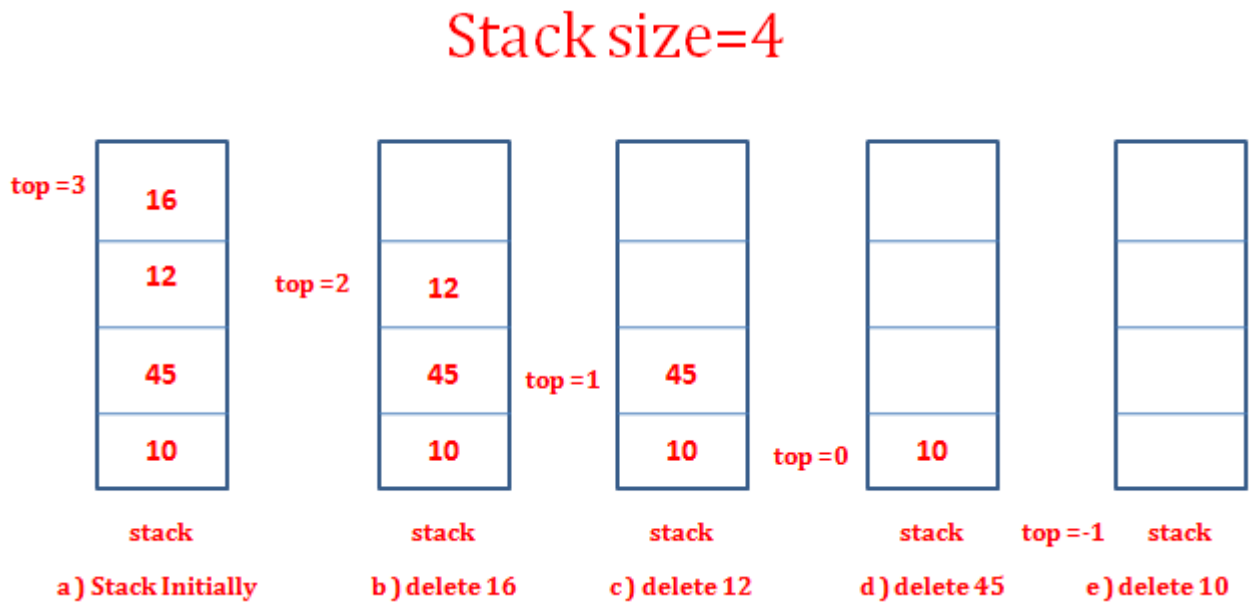
**Step 1 :** Check whether **stack** is **EMPTY**. (**top == -1**)

**Step 2 :** If it is **EMPTY**, then display **"Stack is EMPTY!!! Deletion is not possible!!!"** and Terminate the function.

**Step 3 :** If it is **NOT EMPTY**, then delete **stack[top]** and decrement **top** value by one (**top--**).

**Program :**

```
void push( )
{
        int item;
        if(top = =  – 1 )
        {
                printf(“ Stack is Empty!!! Deletion is not possible!!!”);
        }
        else
        {
                printf("\nDeleted : %d", stack[top]);
                top--;
        }
}
```



**3. Display:** In a stack , display() is a function to Displays the elements of a Stack

**Algorithm:**

**Step 1 -** Check whether **stack** is **EMPTY**. (**top == -1**)

**Step 2 -** If it is **EMPTY**, then display **"Stack is EMPTY!!!"** and terminate the function.

**Step 3 -** If it is **NOT EMPTY**, then define a variable '**i**' and initialize with top. Display **stack[i]** value and decrement **i** value by one (**i--**).

**Step 4 -** Repeat above step until **i** value becomes '0'.

**Program:**

```
void display()
{
        int i;
        if(top == -1)
                printf("\nStack is Empty!!!");
        else
        {
                printf("\nStack elements are:\n");
                for(i=top; i>=0; i--)
                {
                        printf("%d\n",stack[i]);
                }
        }
}
```

## WRITE A C PROGRAM TO IMPLEMENT STACK USING ARRAY

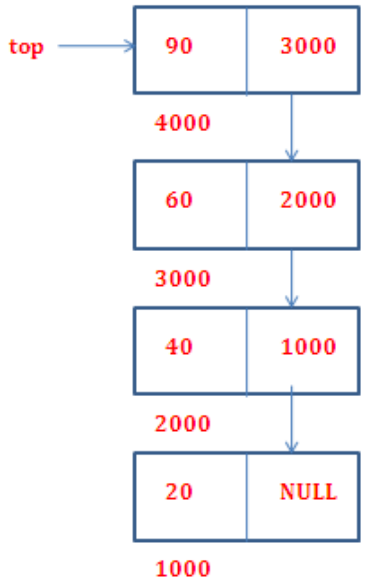
```
#include<stdio.h>
#include<process.h>
#define SIZE 10
void push();
void pop();
void display();
int stack[SIZE], top = -1;
int main()
{
        int choice;
        while(1)
        {
                printf("\n\n***** MENU *****\n");
                printf("1. Push\n");
                printf("2. Pop\n");
                printf("3. Display\n");
                printf("4. Exit\n");
                printf("\nEnter your choice: ");
                scanf("%d",&choice);
                switch(choice)
                {
                        case 1: push();
                                break;
                        case 2: pop();
                                break;
                        case 3: display();
                                break;
                        case 4: exit(0);
                                default: printf("\nWrong selection!!! Try again!!!");
                }
        }
    return 0;
}
```

```
void push( )
{
        int value;
        if(top == SIZE-1)
                printf("\nStack is Full!!! Insertion is not possible!!!");
        else
        {
                printf("Enter the value to be insert: ");
                scanf("%d",&value);
                top++;
                stack[top] = value;
                printf("\nInsertion success!!!");
        }
}
void pop()
{
        if(top == -1)
                printf("\nStack is Empty!!! Deletion is not possible!!!");
        else
        {
                printf("\nDeleted : %d", stack[top]);
                top--;
        }
}
void display()
{
        int i;
        if(top == -1)
                printf("\nStack is Empty!!!");
        else
        {
                printf("\nStack elements are:\n");
                for(i=top; i>=0; i--)
                {
                        printf("%d\n",stack[i]);
                }
        }
}
```

# Stack Using Linked List

- The stack implemented using linked list can work for an unlimited number of values.
- That means, stack implemented using linked list works for the variable size of data. So, there is no need to fix the size at the beginning of the implementation.
- The Stack implemented using linked list can organize as many data values as we want. In linked list implementation of a stack, every new element is inserted as '**top**' element.
- That means every newly inserted element is pointed by '**top**'.
- Whenever we want to remove an element from the stack, simply remove the node which is pointed by '**top**' by moving '**top**' to its previous node in the list.
- The **next** field of the first element must be always **NULL**.

**Example :**




In the above example, the last inserted node is 90 and the first inserted node is 20. The order of elements inserted is 20, 40,60 and 90.

**Structure of node**

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

## Stack Operations using Linked List

1.Push

2.Pop

3.Display

## 1. Push

**Algorithm:**

**Step 1** - Create a **newNode( ptr )** with given value.

**Step 2** - Check whether stack is **Empty** (**top** == **NULL**)

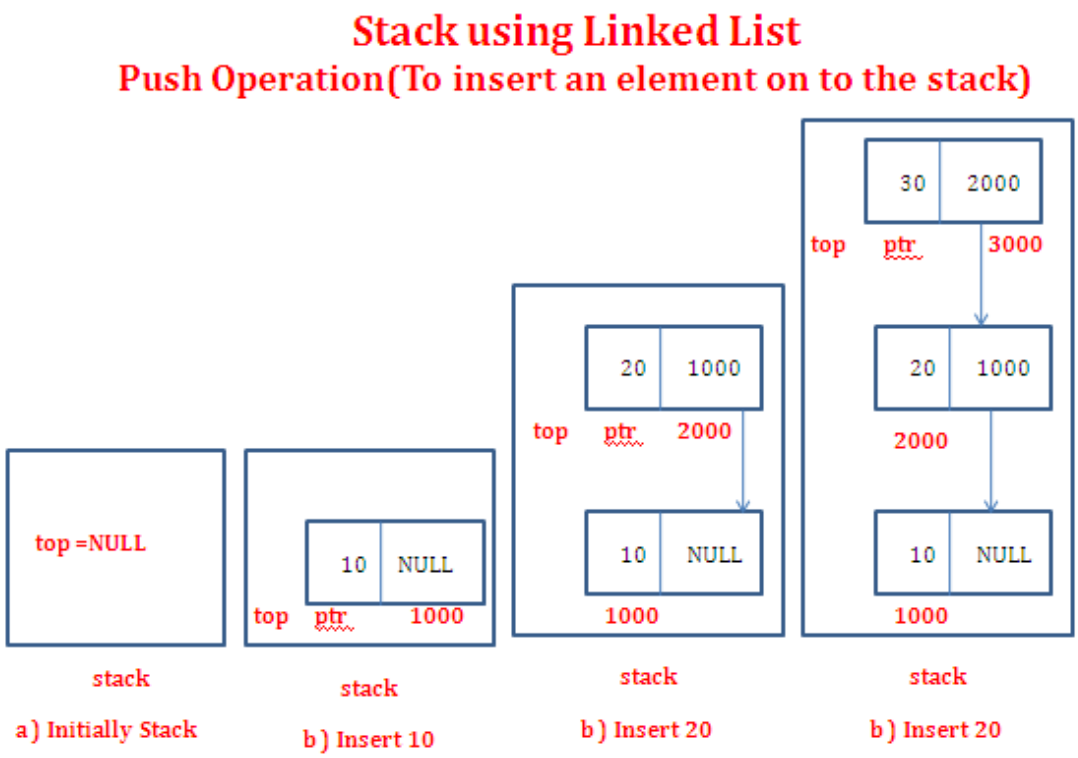
**Step 3** - If it is **Empty**, then set **ptr → next** = **NULL**.

**Step 4** - If it is **Not Empty**, then set **ptr → next** = **top**.

**Step 5** - Finally, set **top** = **ptr**.

```
void push()
{
        struct node *ptr;
        int value;
        ptr = (struct node*)malloc(sizeof(struct node));
        printf("Enter the value to be insert: ");
        scanf("%d", &value);
        ptr->data = value;
        if(top == NULL)
        {
                ptr->next = NULL;
        }
        else
        {
                ptr->next = top;
        }
        top = ptr;
        printf("\nInsertion is Success!!!\n");
}
```

**Example:**




## 2.Pop

**Algorithm:**

**Step 1** - Check whether **stack** is **Empty** (**top == NULL**).

**Step 2** - If it is **Empty**, then display **"Stack is Empty!!! Deletion is not possible!!!"** and terminate the function

**Step 3** - If it is **Not Empty**, then define a **node** pointer '**temp**' and set it to '**top**'.**(temp=top)**

**Step 4** - Then set '**top** = **temp → next**'.

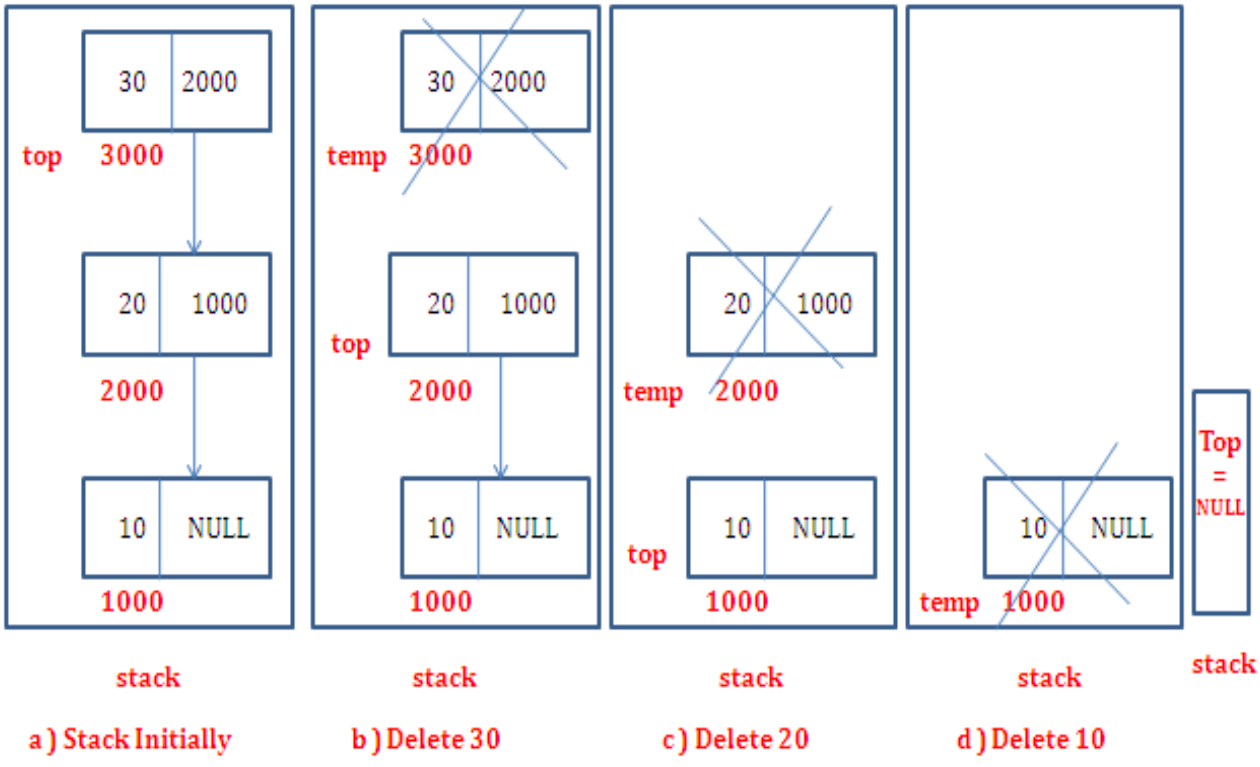
**Step 5** - Finally, delete '**temp**'. (**free(temp)**).

**Program:**

```
void pop()
{
        if(top == NULL)
                printf("\nStack is Empty!!!Deletion is not possible!\n");
        else
        {
                struct node *temp;
                temp=top;
                printf("\nDeleted element: %d", temp->data);
                top = temp->next;
                free(temp);
        }
}
```

## Stack using Linked List

## Pop Operation(To delete an element from the stack )



## 3. Display

**Algorithm:**

**Step 1** - Check whether stack is **Empty** (**top** == **NULL**).

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

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

**Step 4** - Display '**temp → data** --->' and move it to the next node. Repeat the same until **temp** Reaches to the first node in the stack. (**temp → next** != **NULL**).

**Step 5** - Finally! Display '**temp → data** ---> **NULL**'.

**Program:**

```
void display()
{
        if(top == NULL)
                printf("\nStack is Empty!!!\n");
        else
        {
                struct node *temp;
                temp=top;
                while(temp->next != NULL)
                {
                        printf("%d--->",temp->data);
                        temp = temp -> next;
                }
                printf("%d--->NULL",temp->data);
        }
}
```

# WRITE A C PROGRAM TO IMPLEMENT STACK USING LINKED LIST

```
#include<stdio.h>
#include<stdlib.h>
#include<process.h>
struct node
{
        int data;
        struct node *next;
};
struct node *top = NULL;
void push();
void pop();
void display();
void main()
{
        int choice, value;
        printf("\n:: Stack using Linked List ::\n");
        while(1)
        {
                printf("\n\n***** MENU *****\n");
                printf("1. Push\n");
                printf("2. Pop\n");
                printf("3. Display\n");
                printf("4. Exit\n");
                printf("\nEnter your choice: ");
                scanf("%d",&choice);
                switch(choice)
                {
                        case 1: push();
                                break;
                        case 2: pop();
                                break;
                        case 3: display();
                                break;
                        case 4: exit(0);
                                default: printf("\nWrong selection!!! Try again!!!");
                }
        }
}
```

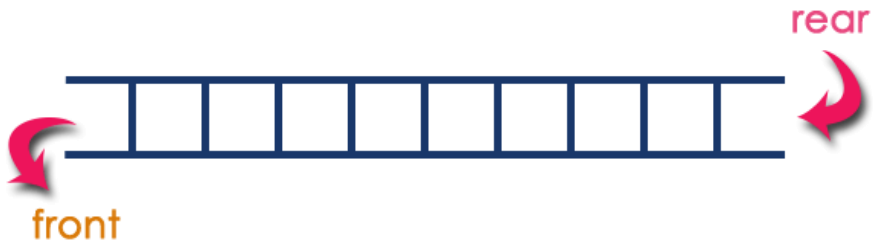
```
void push()
{
	struct node *ptr;
	int value;
	ptr = (struct node*)malloc(sizeof(struct node));
	printf("Enter the value to be insert: ");
	scanf("%d", &value);
	ptr->data = value;
	if(top == NULL)
	{
		ptr->next = NULL;
	}
	else
	{
		ptr->next = top;
	}
	top = ptr;
	printf("\nInsertion is Success!!!\n");
}
void pop()
{
	if(top == NULL)
		printf("\nStack is Empty!!!Deletion is not possible!\n");
	else
	{
		struct node *temp;
		temp=top;
		printf("\nDeleted element: %d", temp->data);
		top = temp->next;
		free(temp);
	}
}
void display()
{
	if(top == NULL)
		printf("\nStack is Empty!!!\n");
	else
	{
		struct node *temp;
		temp=top;
		while(temp->next != NULL)
		{
			printf("%d--->",temp->data);
			temp = temp -> next;
		}
		printf("%d--->NULL",temp->data);
	}
}
```

# Queue ADT

## What is a Queue?

- Queue is a linear data structure in which the insertion and deletion operations are performed at two different ends.
- In a queue data structure, adding and removing elements are performed at two different positions.
- The insertion is performed at one end and deletion is performed at another end.
- In a queue data structure, the insertion operation is performed at a position which is known as '**rear**'.
- In a queue data structure, the deletion operation is performed at a position which is known as '**front**'.
- In queue data structure, the insertion and deletion operations are performed based on **FIFO (First In First Out)** principle.

### BASIC STRUCTURE OF QUEUE




- In a queue data structure, the insertion operation is performed using a function called "**enQueue()**".
- In a queue data structure, the deletion operation is performed using a function called "**deQueue()**".

## Real life examples of queue are:

- **A queue of people at ticket-window:** The person who comes first gets the ticket first. The person who is coming last is getting the tickets in last. Therefore, it follows first-in-first-out (FIFO) strategy of queue.
- **Vehicles on toll-tax bridge:** The vehicle that comes first to the toll tax booth leaves the booth first. The vehicle that comes last leaves last. Therefore, it follows first-in-first-out **(FIFO)** strategy of queue.

## QUEUE IMPLEMENTATIONS

1. Queue Using Array
2. Queue Using Linked List

## QUEUE USING ARRAY :

- A queue data structure can be implemented using one dimensional array.
- The queue implemented using array stores only fixed number of data values.
- The implementation of queue data structure using array is very simple.
- Just define a one dimensional array of specific size and insert or delete the values into that array by using **FIFO (First In First Out) principle** with the help of variables **'front'** and '**rear**'.
- Initially both '**front**' and '**rear**' are set to -1.
- Whenever, we want to insert a new value into the queue, increment '**rear**' value by one and then insert at that position.
- Whenever we want to delete a value from the queue, then delete the element which is at 'front' position and increment 'front' value by one.

## Example

Queue after inserting 25, 30, 51, 60 and 85.


## Queue Operations Using Array

1.enQueue( ) - (To insert an element into the queue)

2.deQueue() - (To delete an element from the queue)

3.display() - (To display the elements of the queue)

## 1. enQueue()

- In a queue data structure, enQueue() is a function used to insert a new element into the queue.
- In a queue, the new element is always inserted at **rear** position.
- The enQueue() function Whenever we want to insert a value into the queue, increment the **rear** value by one and then insert.

**Algorithm :**

**Step 1** - Check whether **queue** is **FULL**. (**rear == SIZE-1**)

**Step 2** - If it is **FULL**, then display **"Queue is FULL!!! Insertion is not possible!!!"** and terminate the function.

**Step 3** - If it is **NOT FULL**, then increment **rear** value by one (**rear++**) and set **queue[rear]** = **value**.

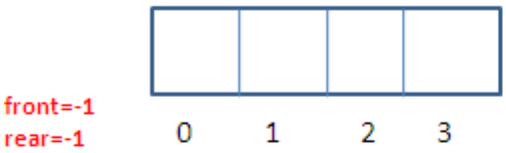
**Program:**

```
void enQueue()
{
        int value;
        if(rear == SIZE-1)
        {
                printf("\nQueue is Full!!! Insertion is not possible!!!");
        }
        else
        {
                if(front == -1)
                {
                        front = 0;
                }
                else
                {
                        printf("Enter the value to be insert: ");
                        scanf("%d",&value);
                        rear++;
                        queue[rear] = value;
                        printf("\nInsertion success!!!");
                }
        }
}
```

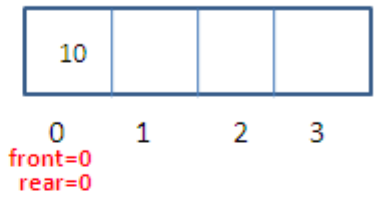
## enQueue() - Inserting value into the queue

Queue size = 4

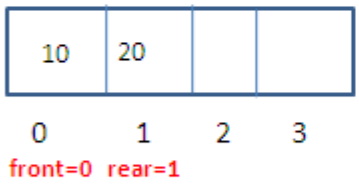
A ) Empty Queue



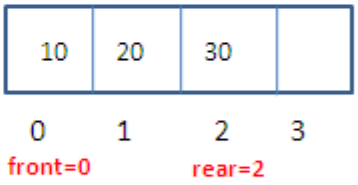

B ) Insert element in queue is 10



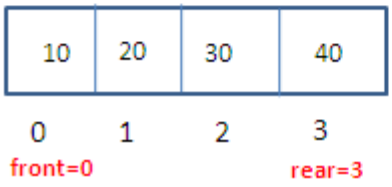

C ) Insert element in queue is 20




D ) Insert element in queue is 30




E ) Insert element in queue is 40




## 2.deQueue()

- In a queue data structure, deQueue() is a function used to delete an element from the queue.
- In a queue, the element is always deleted from **front** position.

**Algorithm:**

**Step 1** - Check whether **queue** is **EMPTY**. (**front == rear || front > rear** )

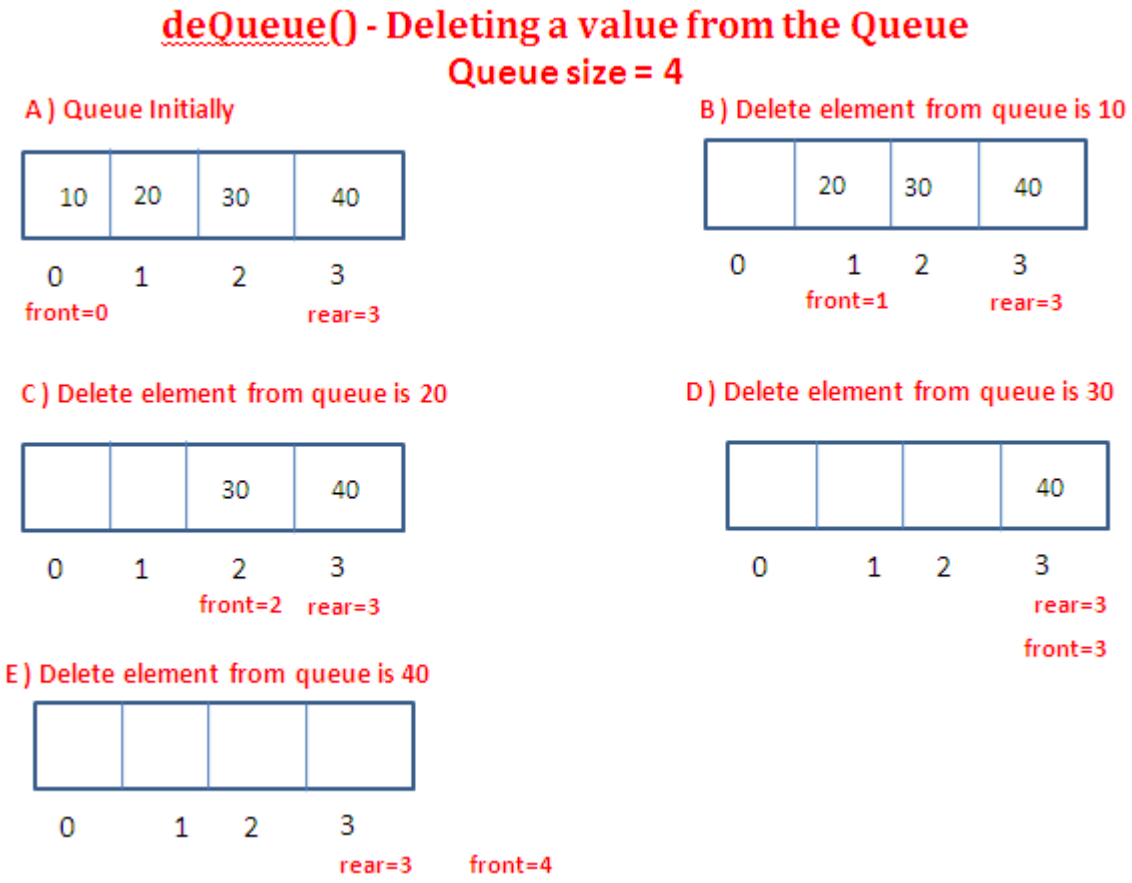
**Step 2** - If it is **EMPTY**, then display **"Queue is EMPTY!!! Deletion is not possible!!!"** and terminate the function.

**Step 3** - If it is **NOT EMPTY**, then increment the **front** value by one (**front ++**). Then display **queue[front]** as deleted element.

**Program :**

```
void deQueue()
{
        if(rear== -1 || front > rear)
        {
                printf("\nQueue is Empty!!! Deletion is not possible!!!");
        }
        else
        {
                 printf("\nDeleted : %d", queue[front]);
                front++;
        }
}
```

**Example:**



### 3. Display ()

**Algorithm:**

**Step 1** - Check whether **queue** is **EMPTY**. (**front == rear || front > rear** )

**Step 2 -** If it is **EMPTY**, then display **"Queue is EMPTY!!!"** and terminate the function.

**Step 3** - If it is **NOT EMPTY**, then define an integer variable '**i**' and set '**i** = **front**'.

**Step 4** - Display '**queue[i]**' value and increment '**i**' value by one (**i++**).

Repeat the same until '**i**' value reaches to **rear** (**i** <= **rear**)

**Program:**

```
void display()
{
        if(rear== -1 || front > rear)
        {
                printf("\nQueue is Empty!!!");
        }
        else
        {
                int i;
                printf("\nQueue elements are:\n");
                for(i=front; i<=rear; i++)
                {
                        printf("%d\t",queue[i]);
                }
        }
}
```

## WRITE A C PROGRAM TO IMPLEMENT QUEUE USING ARRAY :

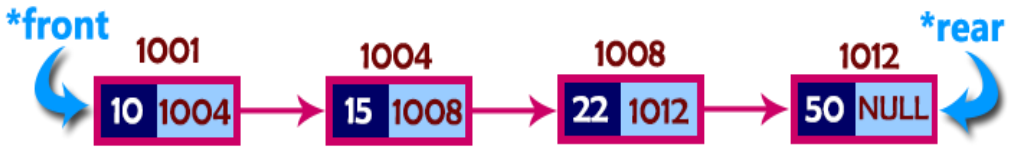
```
#include<stdio.h>
#include<conio.h>
#include<process.h>
#define SIZE 10
void enQueue( );
void deQueue();
void display();
int queue[SIZE], front = -1, rear = -1;
void main()
{
        int choice;
        while(1)
        {
                printf("\n\n***** QUEUE USING ARRAY MENU *****\n");
                printf("1. enQueue\n");
                printf("2. deQueue\n");
                printf("3. Display\n");
                printf("4. Exit\n");
                printf("\nEnter your choice: ");
                scanf("%d",&choice);
                switch(choice)
                {
                        case 1: enQueue();
                                break;
                        case 2: deQueue();
                                break;
                        case 3: display();
                                break;
                        case 4: exit(0);
                                default: printf("\nWrong selection!!! Try again!!!");
                }
        }
}
void enQueue()
{
        int value;
        if(rear == SIZE-1)
        {
                printf("\nQueue is Full!!! Insertion is not possible!!!");
        }
        else
        {
```

```
                if(front == -1)
                {
                        front = 0;
                }
                else
                {
                        printf("Enter the value to be insert: ");
                        scanf("%d",&value);
                        rear++;
                        queue[rear] = value;
                        printf("\nInsertion success!!!");
                }
        }
}
void deQueue()
{
        if(rear== -1 || front > rear)
        {
                printf("\nQueue is Empty!!! Deletion is not possible!!!");
        }
        else
        {
                 printf("\nDeleted : %d", queue[front]);
                front++;
        }
}
void display()
{
        if(rear== -1 || front > rear)
        {
                printf("\nQueue is Empty!!!");
        }
        else
        {
                int i;
                printf("\nQueue elements are:\n");
                for(i=front; i<=rear; i++)
                {
                        printf("%d\t",queue[i]);
                }
        }
}
```

## QUEUE USING LINKED LIST

- A queue data structure can be implemented using a linked list data structure.
- The queue which is implemented using a linked list can work for an unlimited number of values.
- That means, queue using linked list can work for the variable size of data (No need to fix the size at the beginning of the implementation).
- The Queue implemented using linked list can organize as many data values as we want.In linked list implementation of a queue, the last inserted node is always pointed by '**rear**' and the first node is always pointed by '**front**'.

**Example :**




In above example, the last inserted node is 50 and it is pointed by 'rear' and the first inserted node is 10 and it is pointed by 'front'. The order of elements inserted is 10, 15, 22 and 50.

**Structure of Node :**

```
struct node
{
        int data;
        struct node *next;
};
struct node *front = NULL,*rear = NULL;
```

## Queue Operations Using Linked List

1.enQueue( ) - (To insert an element into the queue)

2.deQueue() - (To delete an element from the queue)

3.display() - (To display the elements of the queue)

## 1. enQueue()

**Algorithm :**

**Step 1** - Create a **newNode(ptr)** with given value and set '**ptr → next**' to **NULL**.

**Step 2** - Check whether queue is **Empty** (**rear** == **NULL**)

**Step 3** - If it is **Empty** then, set **front** = **ptr** and **rear** = **ptr**.

**Step 4** - If it is **Not Empty** then, set **rear → next** = **ptr** and **rear** = **ptr**.

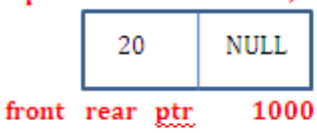
**Program:**

```
void enQueue()
{
        struct node *ptr;
        int value;
        ptr = (struct node*)malloc(sizeof(struct node));
        printf("Enter the value to be insert: ");
        scanf("%d", &value);
        ptr->data = value;
        ptr -> next = NULL;
        if(rear == NULL)
        {
                front=ptr;
                rear=ptr;
        }
        else
        {
                rear -> next = ptr;
                rear = ptr;
        }
        printf("\nInsertion is Success!!!\n");
}
```

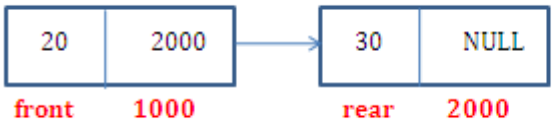
# enQueue() - Inserting an element into the Queue

Step1 : Initially front=NULL and rear = NULL

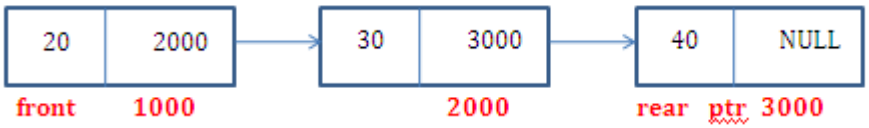
Step2 : Insert element 20, rear==NULL, then set front=ptr and rear=ptr



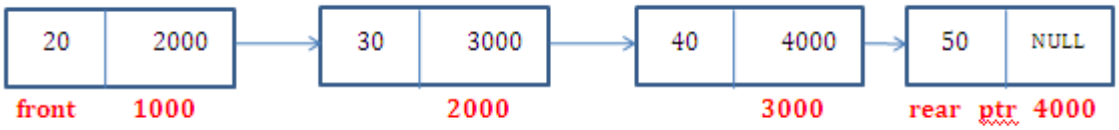

Step3 : Insert element 30, rear==NULL, (1000==NULL) condition false, then rear -> next = ptr and rear=ptr




Step4 : Insert element 40, rear==NULL, (2000==NULL) condition false, then rear -> next = ptr and rear=ptr




Step5 : Insert element 50, rear==NULL, (3000==NULL) condition false, then rear -> next = ptr and rear=ptr




## 2.deQueue()

### Algorithm :

Step 1 - Check whether **queue** is **Empty** (**front == NULL**).

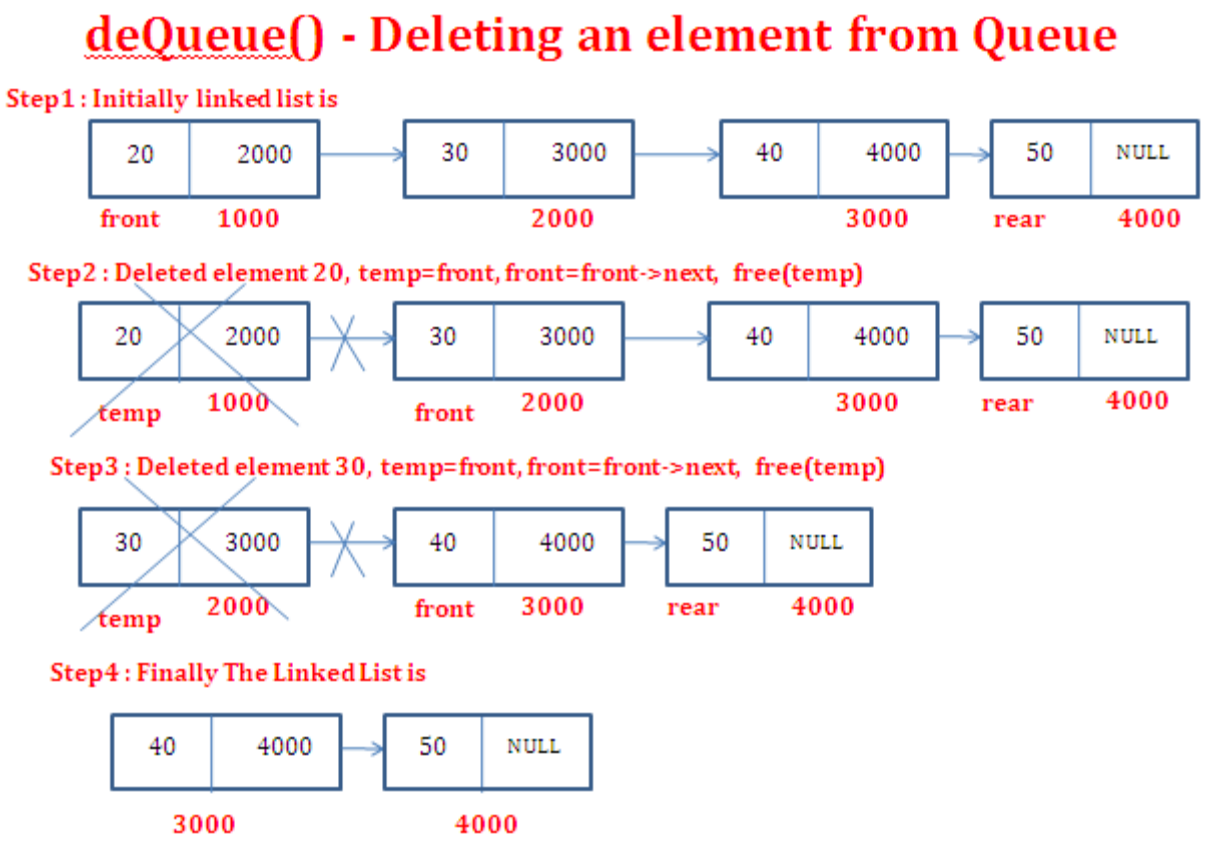
Step 2 - If it is **Empty**, then display **"Queue is Empty!!! Deletion is not possible!!!"** and terminate from the function

Step 3 - If it is **Not Empty** then, define a Node pointer '**temp**' and set it to '**front**'.

Step 4 - Then set '**front** = **front → next**' and delete '**temp**' (**free(temp)**).

**Program :**

```
void deQueue()
{
        if(front == NULL)
                printf("\nQueue is Empty!!!Deletion is not possible!\n");
        else
        {
                struct node *temp;
                temp=front;
                front = front -> next;
                printf("\nDeleted element: %d\n", temp->data);
                free(temp);
        }
}
```



## 3.display() - Displaying the elements of Queue

**Algorithm :**

**Step 1** - Check whether queue is **Empty** (**front** == **NULL**).

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

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

**Step 4** - Display '**temp → data** --->' and move it to the next node. Repeat the same until '**temp**' reaches to '**rear**' (**temp → next** != **NULL**).

**Step 5** - Finally! Display '**temp → data** ---> **NULL**'.

**Program:**

```
void display()
{
        if(front == NULL)
                printf("\nQueue is Empty!!!\n");
        else
        {
                struct node *temp;
                temp=front;
                while(temp->next != NULL)
                {
                        printf("%d--->",temp->data);
                        temp = temp -> next;
                }
                printf("%d--->NULL",temp->data);
        }
}
```

## QUEUE USING LINKED LIST

```
#include<stdio.h>
#include<conio.h>
#include<process.h>
#include<stdlib.h>
struct node
{
        int data;
        struct node *next;
};
struct node *front = NULL,*rear = NULL;
void enQueue();
void deQueue();
void display();
void main()
{
        int choice;
        while(1)
        {
                printf("\n\n***** QUEUE USING LINKED LIST MENU *****\n");
                printf("1. enQueue\n");
                printf("2. deQueue\n");
                printf("3. Display\n");
                printf("4. Exit\n");
                printf("\nEnter your choice: ");
                scanf("%d",&choice);
                switch(choice)
                {
                        case 1: enQueue();
                                break;
                        case 2: deQueue();
                                break;
```

```
                        case 3: display();
                                break;
                        case 4: exit(0);
                                default: printf("\nWrong selection!!! Try again!!!");
                }
        }
}
void enQueue()
{
        struct node *ptr;
        int value;
        ptr = (struct node*)malloc(sizeof(struct node));
        printf("Enter the value to be insert: ");
        scanf("%d", &value);
        ptr->data = value;
        ptr -> next = NULL;
        if(front == NULL)
        {
                front=ptr;
                rear=ptr;
        }
        else
        {
                rear -> next = ptr;
                rear = ptr;
        }
        printf("\nInsertion is Success!!!\n");
}
```

```
void deQueue()
{
	if(front == NULL)
		printf("\nQueue is Empty!!!Deletion is not possible!\n");
	else
	{
		struct node *temp;
		temp=front;
		front = front -> next;
		printf("\nDeleted element: %d\n", temp->data);
		free(temp);
	}
}
void display()
{
	if(front == NULL)
		printf("\nQueue is Empty!!!\n");
	else
	{
		struct node *temp;
		temp=front;
		while(temp->next != NULL)
		{
			printf("%d--->",temp->data);
			temp = temp -> next;
		}
		printf("%d--->NULL",temp->data);
	}
}
```

# STACK APPLICATIONS

## What is an Expression?

An expression is a collection of operators and operands that represents a specific value.

**Polish Notation:** Transforming the expression into a form called Polish Notation. An expression can be represented in various forms such as

1. **Infix Notation**
2. **Prefix (Polish) Notation.**
3. **Postfix (Reverese-Polish) Notation.**

1. **Infix Notation:** if the operator is used in between the operands are called Infix Notation.

   **Syntax : <operand> <operator> <operand>**

   **Example :** a + b

2. **Prefix (Polish) Notation:** If the operator is used before operands are called Prefix notation.

   **Syntax : <operator> <operand> <operand>**

   **Example :** + ab

3. **Postfix (Reverse-Polish) Notation:** If the operator is used after operands are called Postfix notation.

   **Syntax : <operand> <operand><operator>**

   **Example :** ab+

Every expression can be represented using all the above three different types of expressions.

## Order of Precedence (Highest to Lowest)

Exponential ( ^ or ↑ ) – **Highest precedence**

Multiplication ( * or x ) or Division ( / or ÷ ) – Left to Right – **Next precedence**

Addition ( + ) or Subtraction ( - ) – Left to Right - **Lowest Precedence**

## CONVERSION OF EXPRESSIONS

1. **Conversion of Infix expression to Postfix expression.**
2. **Conversion of Infix expression to Prefix expression.**
3. **Evolution of Postfix expression.**
4. **Balancing of Symbols**

## 1. CONVERSION OF INFIX EXPRESSION TO POSTFIX EXPRESSION

### Steps required for conversion of Infix to Postfix expression

1. Read an Expression from left to right.

2. If the character is **LEFT PARANTHESIS**, PUSH to the stack.

3. If the character is **OPERAND**, ADD to the postfix expression.

4. If the character is **OPERATOR,** check whether stack is empty or not.

   a)If the **stack is Empty**, then push operator into the stack.

   b)If the **stack is not Empty**, check the priority of the operator

      i ) If scan operator is higher priority than the top of stack operator then scanned operator will push into the stack.

      ii) If scan operator is same or lower priority than the top of stack operator then pop the operator from the stack and add to postfix expression and scanned operator will push into the stack, Repeat step 4.

5. If the character is **RIGHT PARANTHESIS**, then pop all the operators from the stack until it reaches **LEFT PARANTHESIS** and ADD to postfix expression.

6. After reading all characters, if stack is not empty then pop and ADD to postfix expression.

**Example 1 : Convert A + B to postfix expression.**

| S.No | Scanned Character | Operstion | Stack | Postfix Expression |
|---|---|---|---|---|
| 1 | A | Add to postfix 'A' | Stack is Empty | A |
| 2 | + | Push '+' | + | A |
| 3 | B | Add to postfix 'B' | + | AB |
| 4 | Nill | Pop all the operators from the stack and add to postfix expression | Stack is Empty | AB+ |

**Example 2 : Convert A + ( B * C ) to postfix expression.**

| S.No | Scanned Character | Operation | Stack | Postfix Expression |
|---|---|---|---|---|
| 1 | A | Add to postfix 'A' | Stack is Empty | A |
| 2 | + | Push '+' | + | A |
| 3 | ( | Push '(' | + ( | A |
| 4 | B | Add to postfix 'B' | + ( | AB |
| 5 | * | Push ' * ' | + ( * | AB |
| 6 | C | Add to postfix 'C' | + ( * | ABC |
| 7 | ) | Pop ' *' and Add to postfix ' * ' | + | ABC * |
| 8 | Nill | Pop all the operators from the stack and add to postfix expression | Stack is Empty | ABC *+ |

**Example 3 : Convert A * B + C to postfix expression.**

| S.No | Scanned Character | Operation | Stack | Postfix Expression |
|---|---|---|---|---|
| 1 | A | Add to postfix 'A' | Stack is Empty | A |
| 2 | * | Push ' * ' | * | A |
| 3 | B | Add to postfix 'B' | * | AB |
| 4 | + | Pop ' *' and Add to postfix ' * ', Push ' + ' | + | AB* |
| 5 | C | Add to postfix 'C' | + | AB*C |
| 6 | Nill | Pop all the operators from the stack and add to postfix expression | Stack is Empty | AB*C+ |

**Example 4 : Convert ( A + B ) * C - ( D - E ) * ( F + G ) to postfix expression.**

| S.No | Scanned Character | Operation | Stack | Postfix Expression |
|---|---|---|---|---|
| 1 | ( | Push ' ( ' | ( | Nothing |
| 2 | A | Add to postfix 'A' | ( | A |
| 3 | + | Push ' + ' | ( + | A |
| 4 | B | Add to postfix 'B' | ( + | AB |
| 5 | ) | Pop ' + ', add to postfix ' + '. | STACK IS EMPTY | AB+ |
| 6 | * | Push ' * ' | * | AB+ |
| 7 | C | Add to postfix 'C' | * | AB+C |
| 8 | - | Pop '*' , Add to postfix '*' and push ' - ' | - | AB+C* |
| 9 | ( | Push ' ( ' | - ( | AB+C* |
| 10 | D | Add to postfix 'D' | - ( | AB+C*D |
| 11 | - | Push ' - ' | - ( - | AB+C*D |
| 12 | E | Add to postfix 'E' | - ( - | AB+C*DE |
| 13 | ) | Pop '-' add to postfix ' - ' | - | AB+C*DE- |
| 14 | * | Push ' * ' | - * | AB+C*DE- |
| 15 | ( | Push ' ( ' | - * ( | AB+C*DE- |
| 16 | F | Add to postfix 'F' | - * ( | AB+C*DE-F |
| 17 | + | Push ' + ' | - * ( + | AB+C*DE-F |
| 18 | G | Add to postfix 'G' | - * ( + | AB+C*DE-FG |
| 19 | ) | Pop ' + ' and add to postfix ' + ' | - * | AB+C*DE-FG+ |
| 20 | NILL | Pop all the operators from the stack and add to postfix expression | STACK IS EMPTY | AB+C*DE-FG+*- |

**( A + B ) * C - ( D - E ) * ( F + G ) = AB+C*DE-FG+*-**

## 2.CONVERSION OF INFIX EXPRESSION TO PREFIX EXPRESSION

### Steps required for conversion of Infix to Prefix expression

1. Read an Expression from left to right.
2. Reverse the input string.
3. If the character is **OPERAND**, ADD to the prefix expression.
4. If the character is **OPERATOR,** check whether stack is empty or not.
   a)If the **stack is Empty**, then push operator into the stack.
   b)If the **stack is not Empty**, check the priority of the operator
      i ) If scan operator is same or higher priority than the top of stack , scanned operator will push on stack.
      ii) If scan operator is lower priority than the top of stack , pop the operator from the stack and add to prefix expression and scanned operator will push on stack, Repeat step 4.
5. If the character is **CLOSING PARANTHESIS**, then push operator into the stack.
6. If the character is **OPEN PARANTHESIS**, then pop all the operators from the stack until it reaches **CLOSING PARANTHESIS** and ADD to prefix expression.pop and discard the closing parenthesis.
7. After reading all characters, if stack is not empty then pop and ADD to prefix expression
8. Reverse the output string.

**Example 1 : Convert A + B to prefix expression.**

**Solution : Reverse the input string : A + B = B + A**

| S.No | Scanned Character | Operation | Stack | Prefix Expression |
|---|---|---|---|---|
| **1** | B | Add to prefix 'B' | Stack is Empty | B |
| **2** | + | Push ' + ' | + | B |
| **3** | A | Add to prefix 'A' | + | BA |
| **4** | Nill | Pop all the operators from the stack and add to prefix expression | Stack is Empty | BA+ |
| **5** | Reverse the output string | | | +AB |

**Example 2 : Convert A + ( B * C ) to prefix expression.**

**Solution : Reverse the input string : A + ( B * C ) = ) C * B ( + A**

| S.No | Scanned Character | Operation | Stack | Prefix Expression |
|---|---|---|---|---|
| 1 | ) | Push ' )' | ) | Empty Stack |
| 2 | C | Add to prefix 'C' | ) | C |
| 3 | * | Push ' * ' | ) * | C |
| 4 | B | Add to prefix 'B' | ) * | CB |
| 5 | ( | Pop ' *" and Add to prefix '*' | Stack is Empty | CB * |
| 6 | + | Push ' + ' | + | CB * |
| 7 | A | Add to prefix 'A' | + | CB *A |
| 8 | Nill | Pop all the operators from the stack and add to postfix expression | Stack is Empty | CB *A+ |
| 9 | Reverse the output string | | | +A*BA |

**Example 3 : Convert A * B + C to Prefix expression.**

**Solution : Reverse the input string : A * B + C = C + B * A**

| S.No | Scanned Character | Operation | Stack | Prefix Expression |
|---|---|---|---|---|
| 1 | C | Add to prefix 'C' | Stack is Empty | C |
| 2 | + | Push ' + ' | + | C |
| 3 | B | Add to prefix 'B' | + | CB |
| 4 | * | Push ' * ' | +* | CB |
| 5 | A | Add to prefix 'A' | +* | CBA |
| 6 | Nill | Pop all the operators from the stack and add to prefix expression | Stack is Empty | CBA*+ |
| 7 | Reverse the output string | | | +*ABC |

**Example 4 : Convert ( A + B ) * C - ( D - E ) * ( F + G ) to Prefix expression.**

**Solution : Reverse the input string : ) G + F ( * ) E - D ( - C * ) B + A (**

| S.No | Scanned Character | Operation | Stack | Prefix Expression |
|---|---|---|---|---|
| 1 | ) | Push ' ) ' | ) | Nothing |
| 2 | G | Add to prefix 'G' | ) | G |
| 3 | + | Push ' + ' | ) + | G |
| 4 | F | Add to prefix 'F' | ) + | GF |
| 5 | ( | Pop all the operators until closing parenthesis and Add to prefix '+' | Stack is empty | GF+ |
| 6 | * | Push ' * ' | * | GF+ |
| 7 | ) | Push ' ) ' | * ) | GF+ |
| 8 | E | Add to prefix 'E' | * ) | GF+E |
| 9 | - | Push ' - ' | * ) - | GF+E |
| 10 | D | Add to prefix 'D' | * ) - | GF+ED |
| 11 | ( | Pop all the operators until closing parenthesis and Add to prefix '-' | * | GF+ED- |
| 12 | - | Pop ' * 'and add to prefix '* '. push '-' | - | GF+ED-* |
| 13 | C | Add to prefix 'C' | - | GF+ED-*C |
| 14 | * | Push ' * ' | - * | GF+ED-*C |
| 15 | ) | Push ' ) ' | - * ) | GF+ED-*C |
| 16 | B | Add to prefix 'B' | - * ) | GF+ED-*CB |
| 17 | + | Push ' + ' | - * )+ | GF+ED-*CB |
| 18 | A | Add to prefix 'A' | - * )+ | GF+ED-*CBA |
| 19 | ( | Pop all the operators until closing parenthesis and Add to prefix '+' | -* | GF+ED-*CBA+ |
| 20 | Nill | Pop all the operators from the stack and add to prefix expression | Stack is Empty | GF+ED-*CBA+*- |
| 21 | Reverse the output string | | | - * + ABC*-DE+FG |

## 3. POSTFIX EXPRESSION EVALUATION

**Postfix Expression :** If the operator is used after operands are called Postfix expression.

**Syntax : <operand> <operand><operator>**

**Example :** ab+

**Postfix Expression Evaluation using Stack Data Structure**

To evaluate a postfix expression using Stack data structure we can use the following steps...

1. Read an Expression from left to right.

2. If the character is **OPERAND**, then **PUSH** into stack.

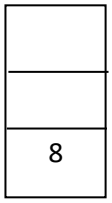
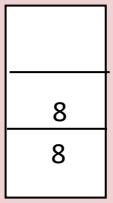
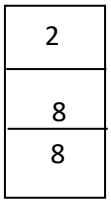
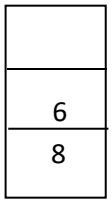
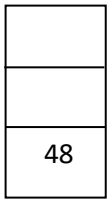
3.If the character is **OPERATOR, POP** top two operands from the stack perform calculation and **PUSH** the result back into stack.

4.After reading the characters from the postfix expression stack will be having only the value which is result.

**Example : Consider the postfix expression**

**5 3 + 8 2 - ***

| S.No | Scanned Character / Symbol | Operations | Stack | Evaluated Part of Expression |
|---|---|---|---|---|
| **1** | Initially | Stack is Empty | | Nothing |
| **2** | 5 | Push ( 5 ) | 5 | Nothing |
| **3** | 3 | Push ( 3 ) | 3<br>5 | Nothing |

| | | | | |
|---|---|---|---|---|
| **4** | + | pop two operands |   | value1 =5<br>value2 =3<br>(5 + 3) = 8 |
| **5** | 8 | Push ( 8 ) |   | (5 + 3) = 8 |
| **6** | 2 | Push ( 2 ) |   | (5 + 3) = 8 |
| **7** | - | pop two operands |   | value1=2<br>value2=8<br>(8-2)=6 |
| **8** | * | pop two operands |   | value1=6<br>value2=8<br>(8 * 6)=48 |
| **9** | Nill | Pop elements from stack. | Stack is Empty | 48 |

**. . The Postfix Expression 5 3 + 8 2 - * value is 48**

**BALANCING OF SYMBOLS :** The objective of this application is to check the Symbols such as parenthesis **( and )**, Square brackets **[ and ]** and Curly braces **{ and }** are matched or not. The algorithm is very much useful in compilers.

We know in a valid expression the parenthesis, Square brackets , Curly braces must occur in pairs. that is when there is an opening parenthesis, Square brackets , Curly braces there should be corresponding closing parenthesis. Otherwise , the expression is not a valid.

**Examples:**

| EXAMPLE | Valid? | Description |
|---|---|---|
| **(A+B) + (C-D)** | Yes | The expression is having balanced symbol |
| **((A+B) + (C-D)** | No | One closing brace is missing. |
| **((A+B) + [C-D])** | Yes | Opening and closing braces correspond |
| **((A+B) + [C-D]]** | No | The last brace does not correspond with the first opening brace. |

**Balancing of symbols using Stack Data Structure**

**Steps required for Balancing of symbols using Stack Data Structure**

**1.** Read an Expression from left to right.

2. If the character read is not a symbol to be balanced, ignore it.

3. If the character is an opening Symbol like **'(' , '{' or '[**' then it is PUSHED in to the stack.

4. If the character is an closing Symbol like **' ) ' , ' } ' , ' ] '** , **then**

a ) If the stack is empty then report as unbalanced expression otherwise pop from the stack .

b ) if the symbol popped is not the corresponding open symbol then report as unbalanced expression

5. After reading all the characters are processed, if the stack is not empty report as unbalanced expression otherwise balanced expression .

**Example 1 : Check whether the expression is balanced symbol or not ( ) ( ( ) [ ( ) ] )**

| S.No | Scanned Character | Operation | Stack |
|---|---|---|---|
| 1 | ( | Push ' ( ' | ( |
| 2 | ) | Pop ' ( ' | Stack is empty |
| 3 | ( | Push ' ( ' | ( |
| 4 | ( | Push ' ( ' | ( ( |
| 5 | ) | Pop ' ( ' | ( |
| 6 | [ | Push ' [ ' | ( [ |
| 7 | ( | Push ' ( ' | ( [ ( |
| 8 | ) | Pop ' ( ' | ( [ |
| 9 | ] | Pop ' ] ' | ( |
| 10 | ) | Pop ' ( ' | Stack is empty |

**The expression is having balanced symbol**

**Example 2 : Check whether the expression is balanced symbol or not ((A+B) + (C-D)**

| S.No | Scanned Character | Operation | Stack |
|---|---|---|---|
| 1 | ( | Push ' ( ' | ( |
| 2 | ( | Push ' ( ' | ( ( |
| 3 | A | - | ( ( |
| 4 | + | - | ( ( |
| 5 | B | - | ( ( |
| 6 | ) | Pop ' ( ' | ( |
| 7 | + | - | ( |
| 8 | ( | Push ' ( ' | ( ( |
| 9 | C | - | ( ( |
| 10 | - | - | ( ( |
| 11 | D | - | ( ( |
| 12 | ) | Pop ' ( ' | ( |

**The expression is not having balanced symbol, because One closing brace is missing.**