UNIT-1

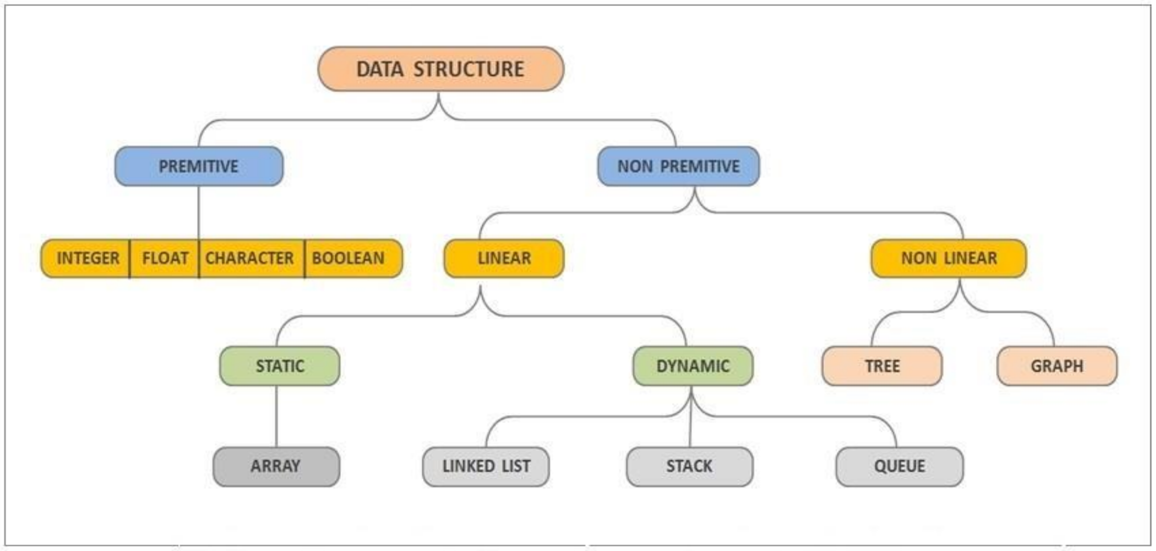
DATA STRUCTURE INTRODUCTION

A **data structure** is a specialized format for organizing, processing, retrieving and storing data.

Every data structure is used to organize the large amount of data.

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

3.Char

4.Double

5.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.

Non-Primitive Data Structures are

1. Linear Data Structure

2. Non - Linear Data Structure

Linear Data Structures

If a 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

Non - Linear Data Structures

If a 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 element to the structure.
3. **Deleting :** Removing a element from the structure.
4. **Traversing:**Visiting each element of the data structure at least once.
5. **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.
6. **Sorting:**Arranging the data elements in some logical order, i.e., in ascending and descending order.
7. **Merging :** Combine the data elements in two different sorted sets into a single sorted set.

ABSTRACT DATA TYPES

- The abstract datatype is special kind of datatype, whose behavior is defined by a set of values and set of operations.
- The keyword "Abstract" is used as we can use these datatypes, 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 datatypes, but operation logics are hidden.

There are three types of ADTs:

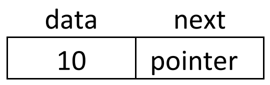
1.Stack

2.Queue

3.List

LINKED LIST

- 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.

ARRAY VS LINKED LIST

ARRAY	**LINKED LIST**
Size of an array is fixed	Size of a list is not fixed
Memory is allocated from stack	Memory is allocated from heap
It is necessary to specify the number of elements during declaration (i.e., during compile time).	It is not necessary to specify the number of elements during declaration (i.e., memory is allocated during run time).
It occupies less memory than a linked list for the same number of elements.	It occupies more memory.
Inserting new elements at the front is potentially expensive because existing elements need to be shifted over to make room.	Inserting a new element at any position can be carried out easily.
Deleting an element from an array is not possible.	Deleting an element is possible.

TYPES OF LINKED LISTS

There are three common types of Linked List.

1.Single Linked List

2.Double Linked List

3.Circular Linked List

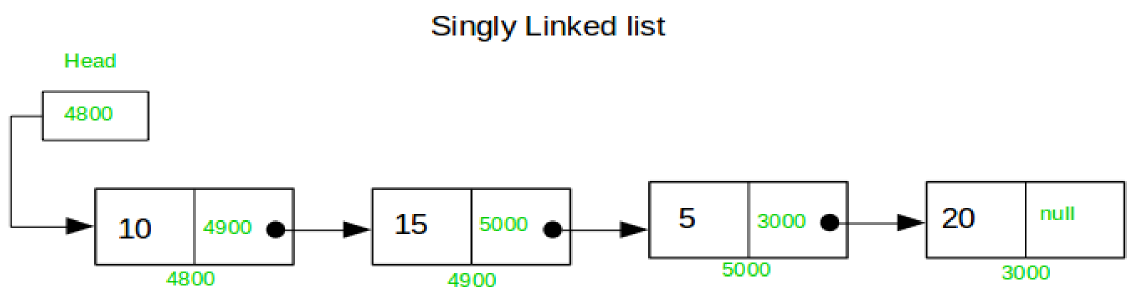
SINGLE LINKED LIST

- **Single linked list is a sequence of elements in which every element has link to its next element in the sequence.**
- In any single linked list, the individual element is called as "Node". Every "Node" contains two fields, data field, and the next field.
- The data field is used to store actual value of the node and next field is used to store the address of next node in the sequence.
- In Single Linked List Navigation only forward direction.

Importent 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 Single Linked List



Basic structure Node:

Each node of a singly linked list follows a common basic structure.

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

struct node *head=NULL;
```

Operations on Single Linked List

The following operations are performed on a Single Linked List

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

Before we implement actual operations, first we need to set up an 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:

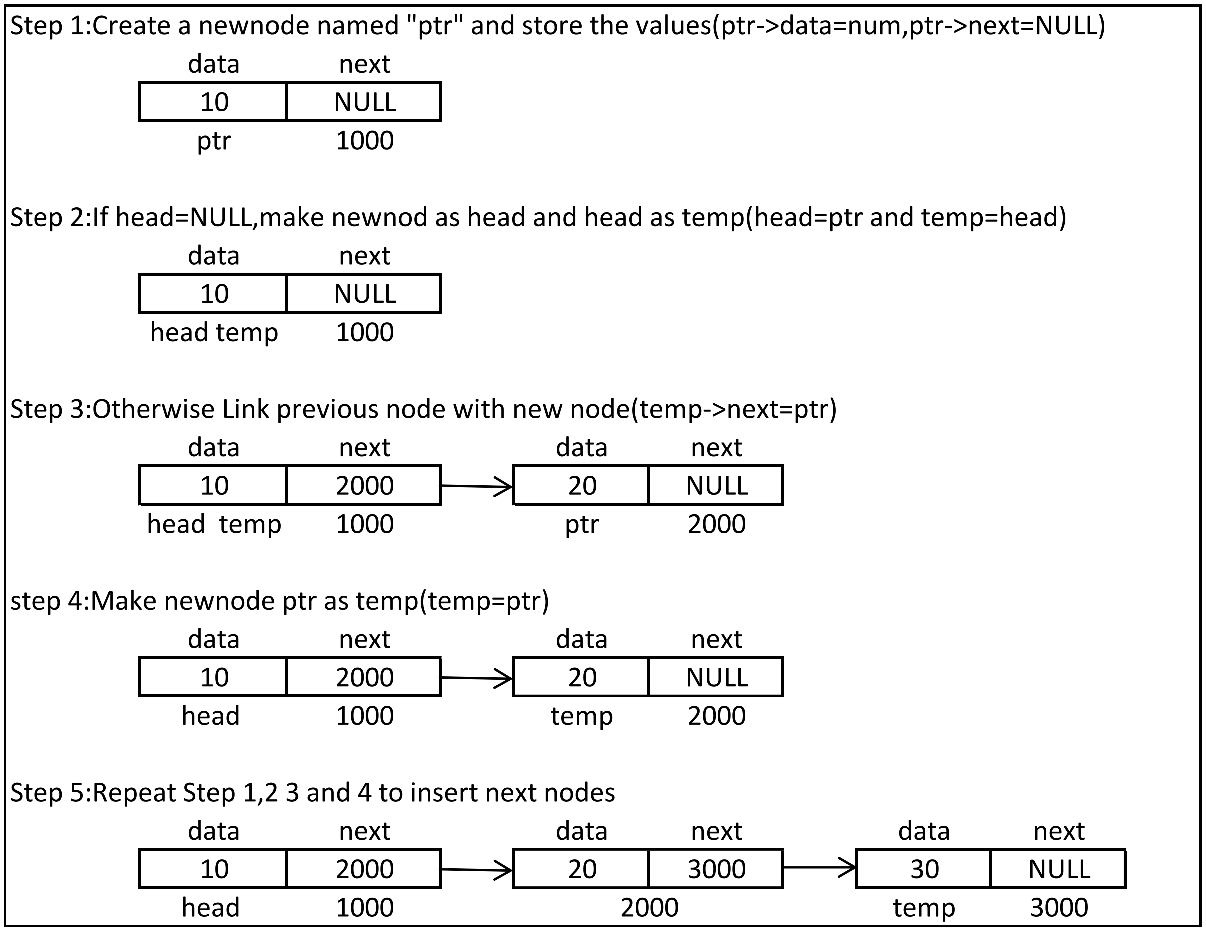
Step 1: Create a newNode with given value, Say "ptr".

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 .**

Example:


Step 1:Create a newnode named "ptr" and store the values(ptr->data=num,ptr->next=NULL)

Step 2:If head=NULL,make newnod as head and head as temp(head=ptr and temp=head)

Step 3:Otherwise Link previous node with new node(temp->next=ptr)

step 4:Make newnode ptr as temp(temp=ptr)

Step 5:Repeat Step 1,2 3 and 4 to insert next nodes

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

Inserting At Beginning of the list :

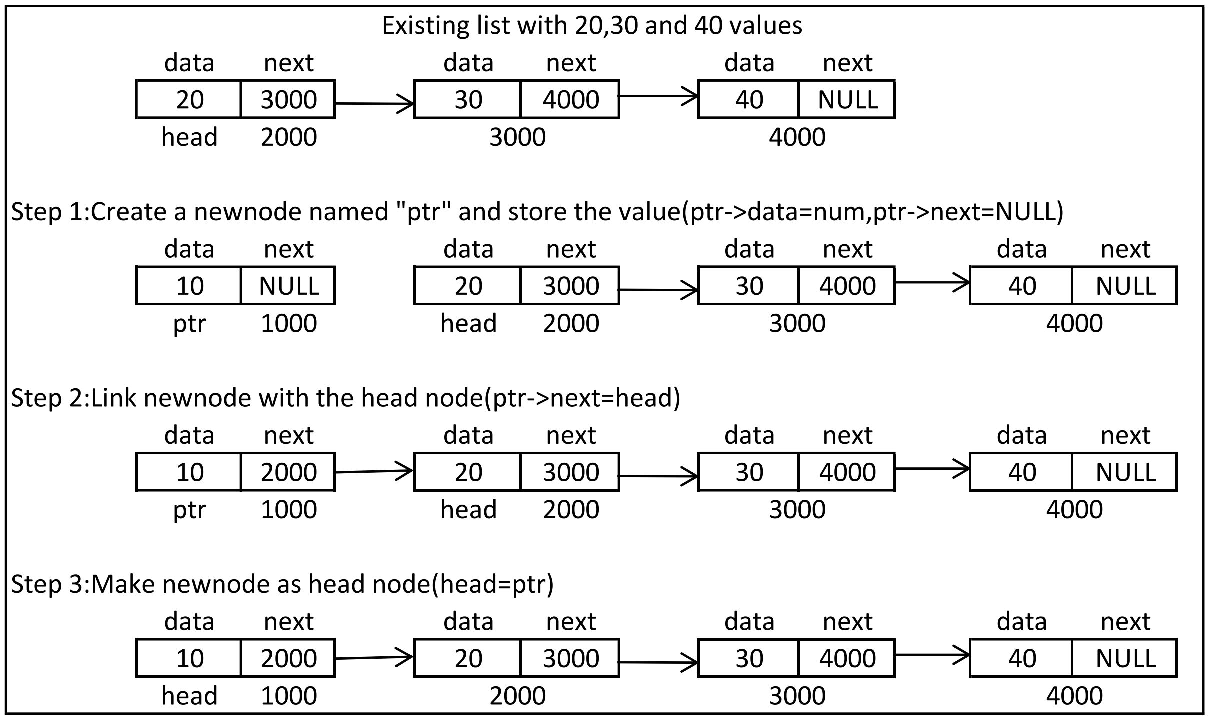
Step 1 - Create a **newNode** with givenvalue, Say “**ptr** “.

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**.

Example:


Existing list with 20,30 and 40 values
data next data next data next
20 3000 30 4000 40 NULL
head 2000 3000 4000
Step 1:Create a newnode named "ptr" and store the value(ptr->data=num,ptr->next=NULL)
data next data next data next data next
10 NULL 20 3000 30 4000 40 NULL
ptr 1000 head 2000 3000 4000
Step 2:Link newnode with the head node(ptr->next=head)
data next data next data next data next
10 2000 20 3000 30 4000 40 NULL
ptr 1000 head 2000 3000 4000
Step 3:Make newnode as head node(head=ptr)
data next data next data next data next
10 2000 20 3000 30 4000 40 NULL
head 1000 2000 3000 4000

Program:

```
void insert_beg()
 {
       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
       {
            ptr->next=head;
            head=ptr;
        }
```

Inserting At End of the list

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

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

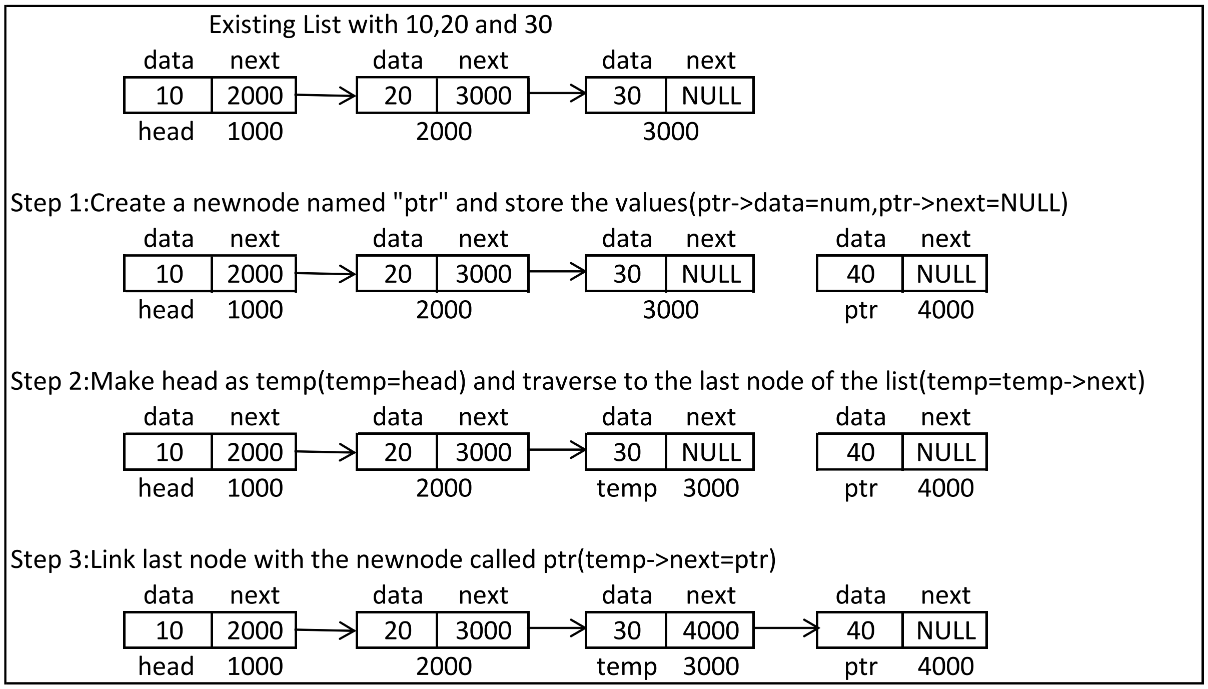
Step 3 - If it is **Empty** then, set **head** = **ptr**.

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**.

Example:


Existing List with 10,20 and 30
data next data next data next
10 2000 20 3000 30 NULL
head 1000 2000 3000
Step 1:Create a newnode named "ptr" and store the values(ptr->data=num,ptr->next=NULL)
10 2000 20 3000 30 NULL 40 NULL
head 1000 2000 3000 ptr 4000
Step 2:Make head as temp(temp=head) and traverse to the last node of the list(temp=temp->next)
10 2000 20 3000 30 NULL 40 NULL
head 1000 2000 temp 3000 ptr 4000
Step 3:Link last node with the newnode called ptr(temp->next=ptr)
10 2000 20 3000 30 4000 40 NULL
head 1000 2000 temp 3000 ptr 4000

Program:

```
void insert_end()
{
        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
        {
            temp=head;
            while(temp->next!=NULL)
            {
              temp=temp->next;
            }
            temp->next=ptr;
        }
   }
```

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

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

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

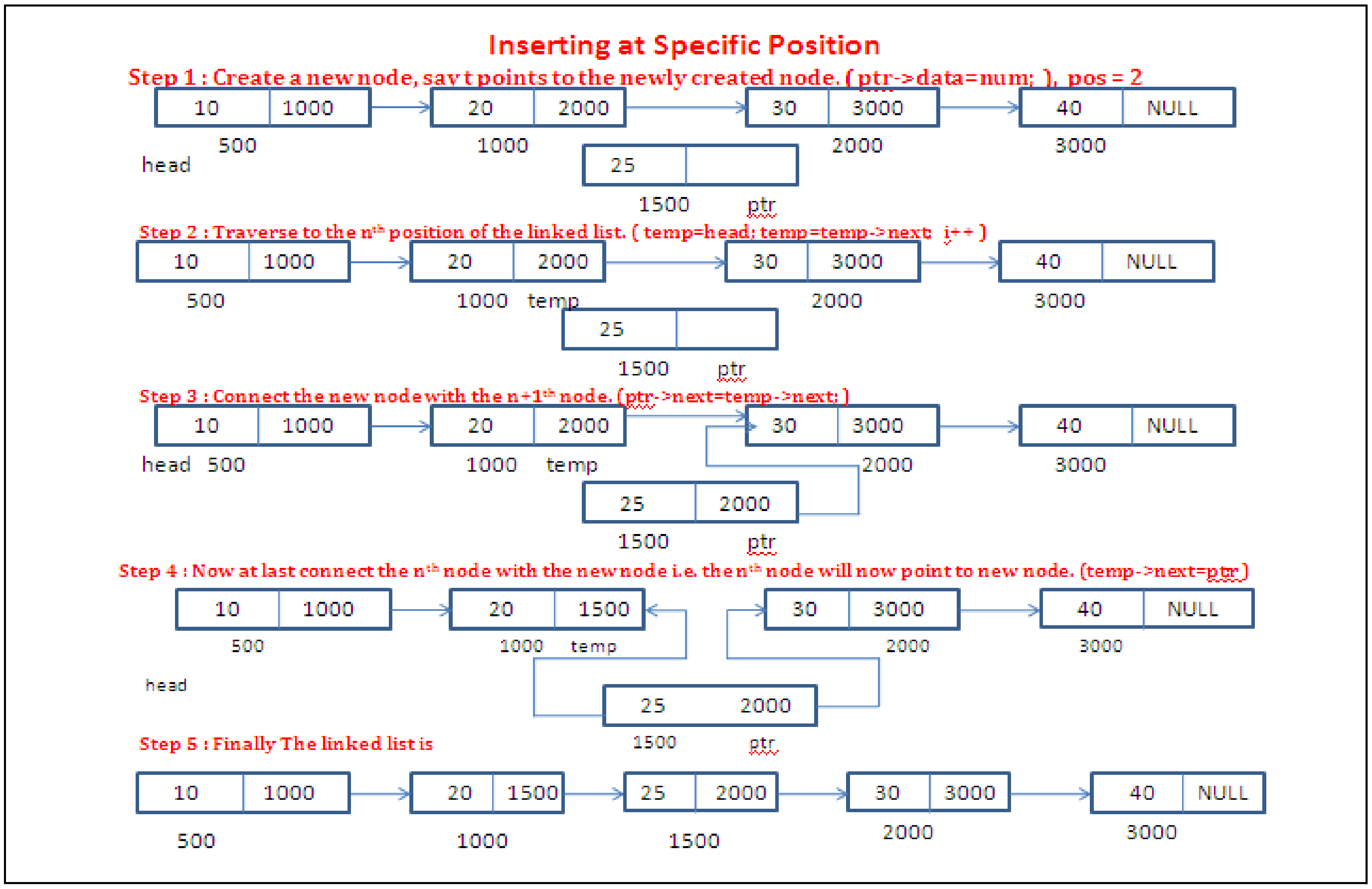
Step 3 - If it is **Empty** then, set **ptr->next=NULL** and **head = ptr**.

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'**

Example:


Inserting at Specific Position
Step 1 : Create a new node, say t points to the newly created node. (ptr->data=num;), pos = 2
10 1000 → 20 2000 → 30 3000 → 40 NULL
500 1000 2000 3000
head
25
1500 ptr
Step 2 : Traverse to the nth position of the linked list. (temp=head; temp=temp->next; i++)
10 1000 → 20 2000 → 30 3000 → 40 NULL
500 1000 temp 2000 3000
25
1500 ptr
Step 3 : Connect the new node with the n+1th node. (ptr->next=temp->next;)
10 1000 → 20 2000 → 30 3000 → 40 NULL
head 500 1000 temp 2000 3000
25 2000
1500 ptr
Step 4 : Now at last connect the nth node with the new node i.e. the nth node will now point to new node. (temp->next=ptr)
10 1000 → 20 1500 30 3000 → 40 NULL
500 1000 temp 2000 3000
head
25 2000
1500 ptr
Step 5 : Finally The linked list is
10 1000 → 20 1500 → 25 2000 → 30 3000 → 40 NULL
500 1000 1500 2000 3000

Program:

```
int insert_pos()
{
    int pos,i=1,num;
    if(head==NULL)
    {
        printf("List is empty!!");
        return 0;
     }
    ptr=(struct node*)malloc(sizeof(struct node));
    printf("Enter data:");
    scanf("%d",&num);
    printf("Enter position to insert:");
    scanf("%d",&pos);
    ptr->data=num;
```

```
        ptr->next=NULL;
        temp=head;
        while(i<pos)
        {
            temp=temp->next;
            i++;
        }
        ptr->next=temp->next;
        temp->next=ptr;
        return 0;
}
```

2.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

Deleting from Beginning of the list

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

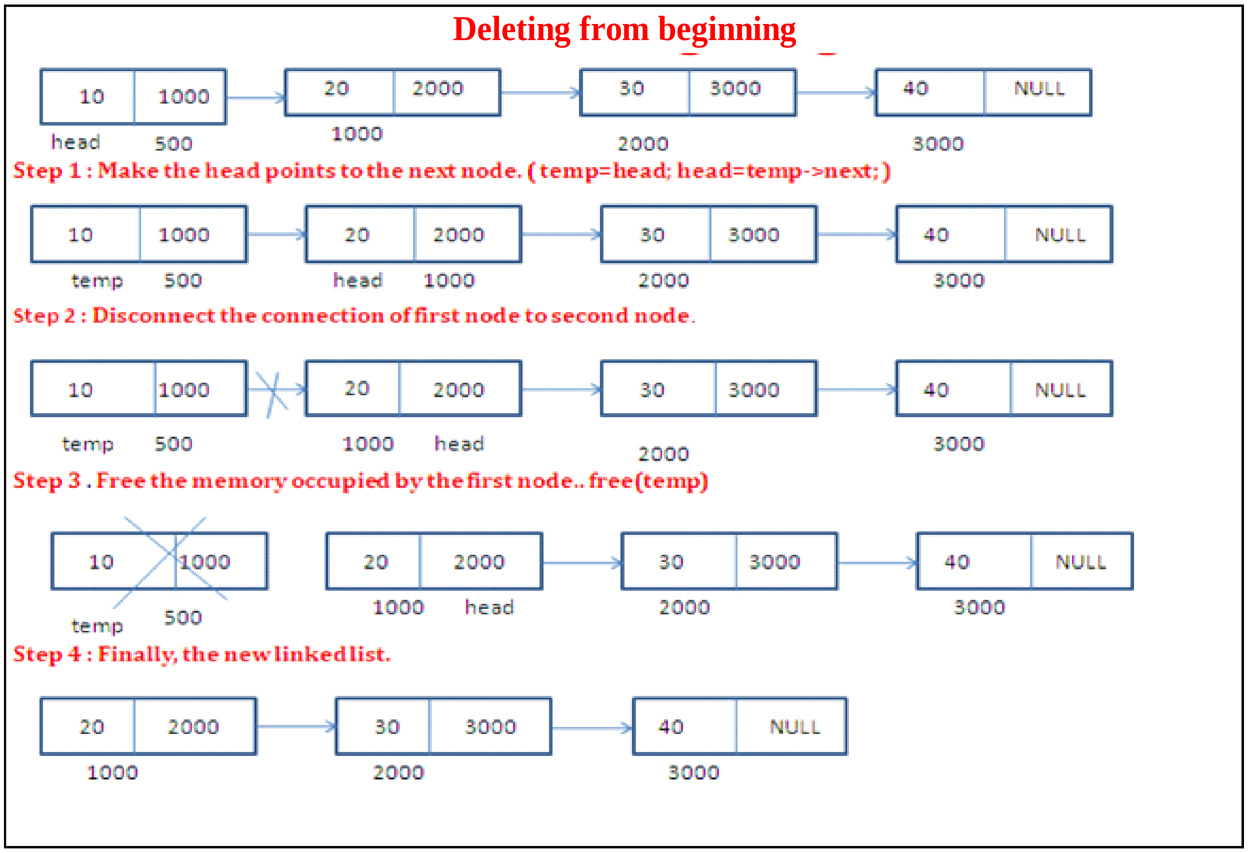
Step 2 - If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.

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**.

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

Example:


Deleting from beginning
10 1000 → 20 2000 → 30 3000 → 40 NULL
head 500 1000 2000 3000
Step 1 : Make the head points to the next node. (temp=head; head=temp->next;)
temp 500 head 1000 2000 3000
Step 2 : Disconnect the connection of first node to second node.
temp 500 1000 head 2000 3000
Step 3 . Free the memory occupied by the first node.. free(temp)
temp 500 1000 head 2000 3000
Step 4 : Finally, the new linked list.
20 2000 → 30 3000 → 40 NULL
1000 2000 3000

Program:

```
void delete_beg()
{
        if(head==NULL)
        {
           printf("List is empty cant delete\n");
        }
        else
        {
             temp=head;
             if(head->next == NULL)
             {
                  head = NULL;
                  printf("Deleted element is %d",temp->data);
                  free(temp);
             }
```

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

Deleting from End of the list

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**.

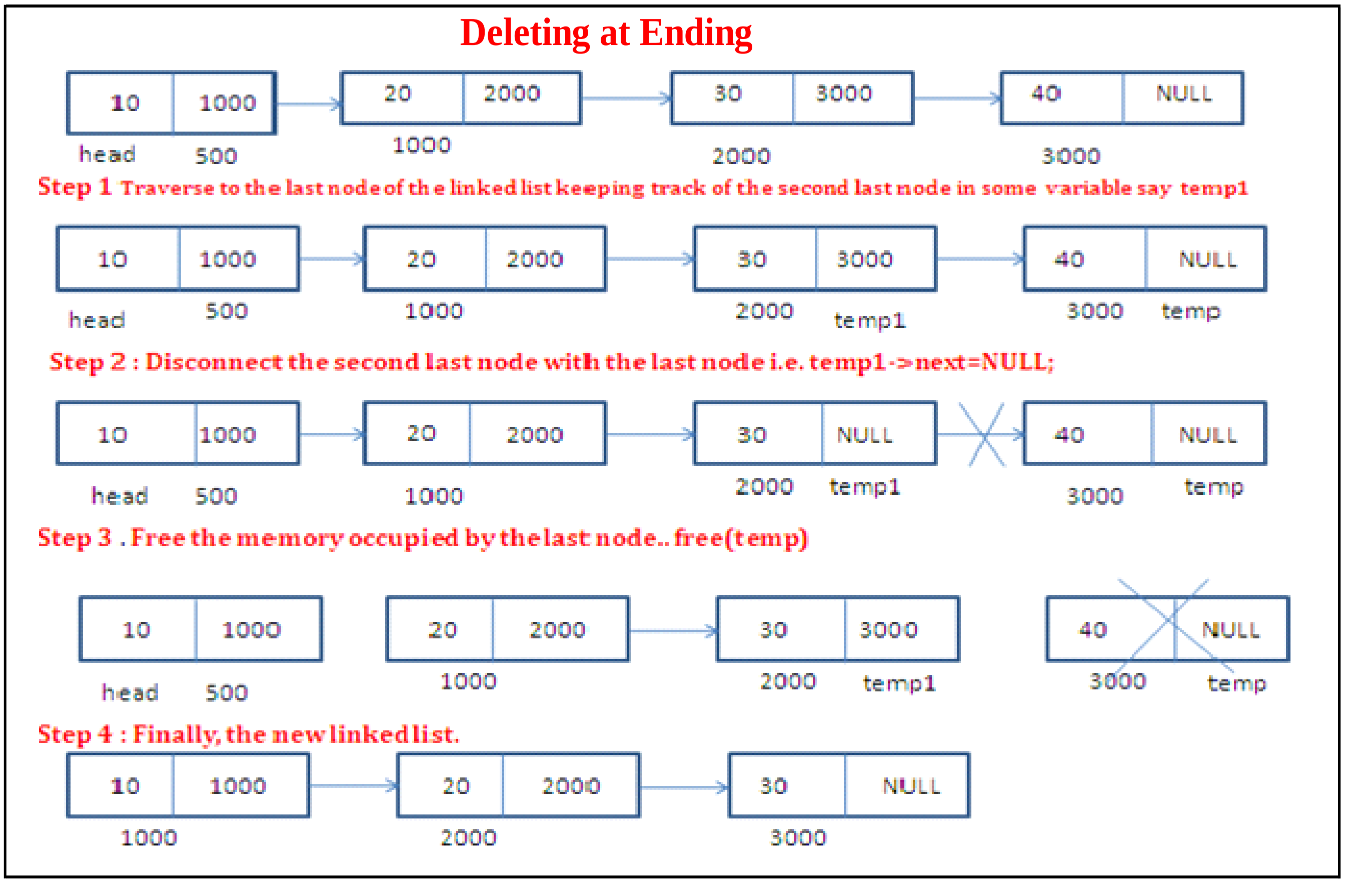
Step 4. Check whether list has only one Node (**temp** → **next** == **NULL**)

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**.

Example:


Deleting at Ending
10 1000 → 20 2000 → 30 3000 → 40 NULL
head 500 1000 2000 3000
Step 1 Traverse to the last node of the linked list keeping track of the second last node in some variable say temp1
head 500 1000 2000 temp1 3000 temp
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.
10 1000 → 20 2000 → 30 NULL
1000 2000 3000

Program:

```
void delete_end()
 {
          if(head==NULL)
          {
             printf("Empty List cant delete\n");
          }
          else
          {
                    temp=head;
                    if(head->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;
                                last=temp1;
                                printf("Deleted element is %d",temp->data);
                                free(temp);
                         }
               }
      }
```

Deleting a Specific Node from the list

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

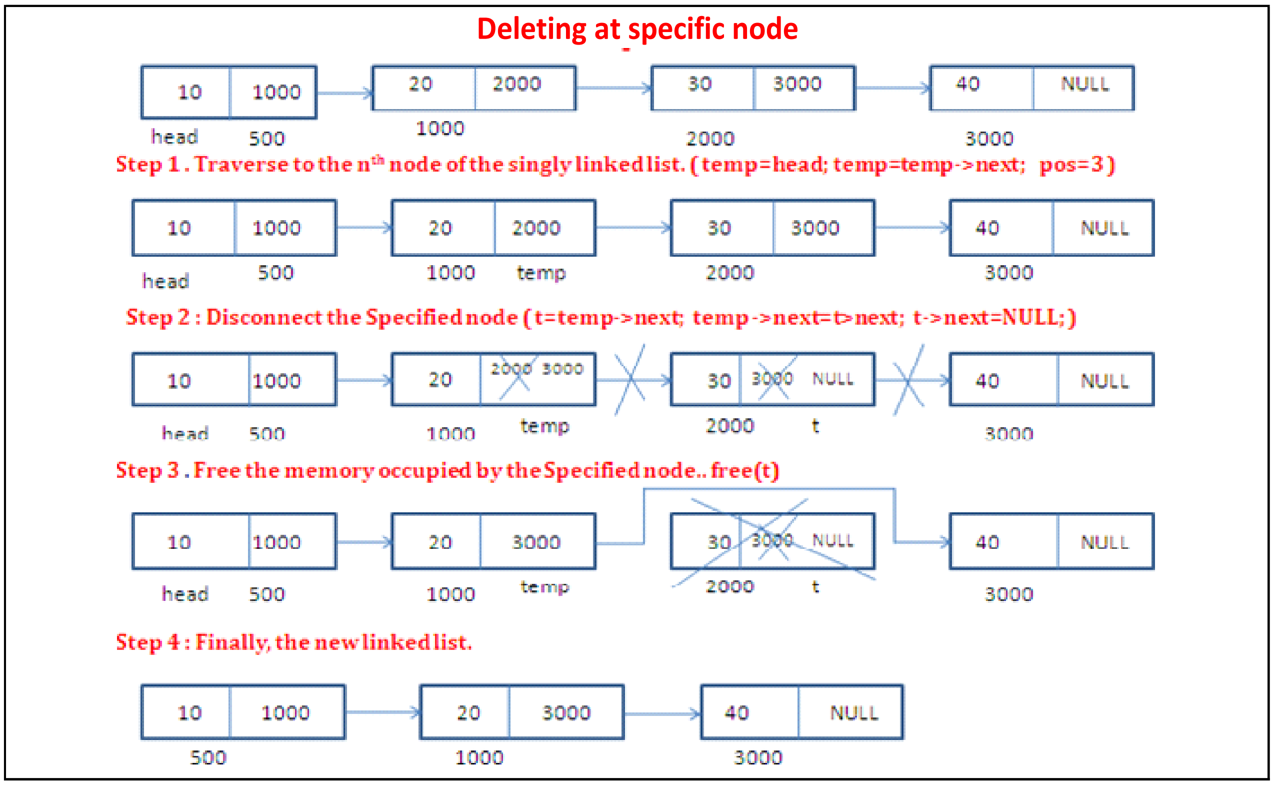
Step 2 - If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.

Step 3 - If it is **Not Empty** then, define two Node pointers **'temp' and 't'** and initialize **'temp'** with **head.**

Step 4- Traverse the **nth Node**.

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

Example:


Deleting at specific node
10
1000
20
2000
30
3000
40
NULL
head
500
1000
2000
3000
Step 1 . Traverse to the nth node of the singly linked list. (temp=head; temp=temp->next; pos=3)
temp
Step 2 : Disconnect the Specified node (t=temp->next; temp->next=t->next; t->next=NULL;)
t
Step 3 . Free the memory occupied by the Specified node.. free(t)
Step 4 : Finally, the new linked list.

Program:

```
int delete_pos()
{
        int pos,i=1;
        if(head==NULL)
        {
                printf("List is empty!!");
                return 0;
        }
        printf("Enter position to delete:");
        scanf("%d",&pos);
        temp=head;
        while(i<pos-1)
        {
                temp=temp->next;
                i++;
        }
        temp1=temp->next;
        temp->next=temp1->next;
        temp1->next=NULL;
        printf("Deleted element is %d",temp1->data);
        free(temp1);
        return 0;
  }
```

4.Displaying a Single Linked List

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()
  {
        printf("The Elements are \n");
        temp=head;
        while(temp!=NULL)
        {
              printf("%d->",temp->data);
              temp=temp->next;
         }
        printf("NULL");
        }
```

5.Searching :

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

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

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

Step 4-Do following while temp is not NULL

temp->data == key , if the condition is true, The element is found.

Step 5-otherwise ,increment count (count++) and move temp to its next node **(temp = temp->next)**

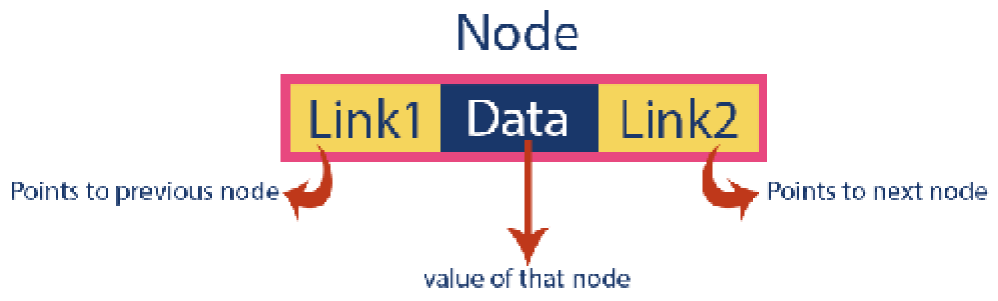
Step 6-while temp is NULL, The element is not found.

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;
}
```

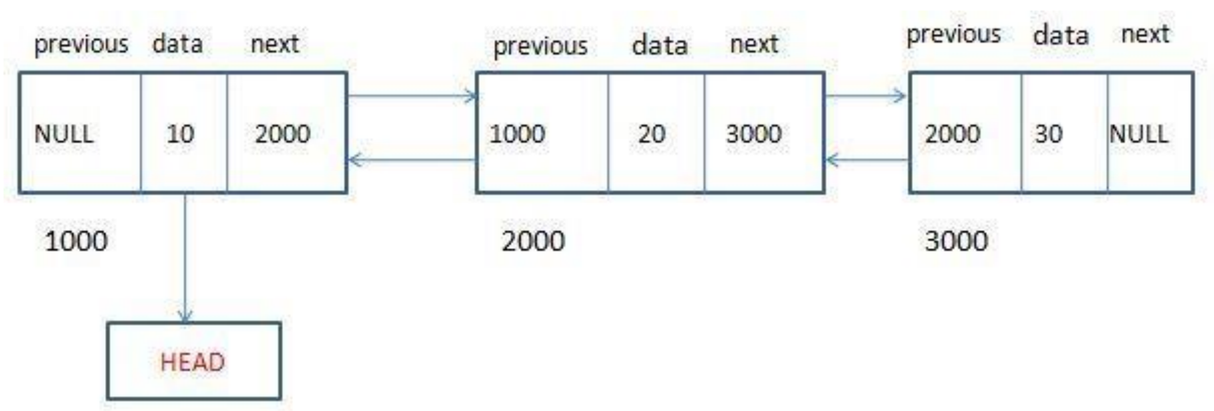
DOUBLE LINKED LIST

- **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...



Here, **'link1'** field is used to store the address of the previous node in the sequence, **'link2'** field is used to store the address of the next node in the sequence and **'data'** field is used to store the actual value of that node.

Example



Importent 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:

Each node of a singly linked list follows a common basic structure.

```
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

1.Creation :

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**.

Step 5 - Create a newNode with given value, Say **"ptr".**

Step 6- Check whether list is **Empty** (**head** == **NULL**).

Step 7- If it is **Empty** then, set **head** = **ptr** and define a Node pointer '**temp**' and initialize with '**head**'.

Step 8- If it is **Not Empty** then, set **temp->next=ptr, ptr->previous=temp** and **temp =ptr.**

Program:

```
#include<stdio.h>
 #include<conio.h>
#include<process.h>
 #include<stdlib.h>
 struct node
{
int data;
struct node  *next;
struct node  *previous;
};
struct node *head=NULL;
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

Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the double linked list...

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;
         }
}
```

Inserting At End of the list

We can use the following steps to insert a new node at end of the double linked list...

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**.

```
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;
            }
}
```

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

We can use the following steps to insert a new node after a node in the double linked list...

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

Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the double linked list...

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);
               }
          }
    }
```

Deleting from End of the list

We can use the following steps to delete a node from end of the double linked list...

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);
            }
        }
    }
```

Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the double linked list...

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;
}
```

3.Displaying a Double Linked List

We can use the following steps to display the elements of a double linked list...

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");
               }
       }
```

4.Searching :

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)**

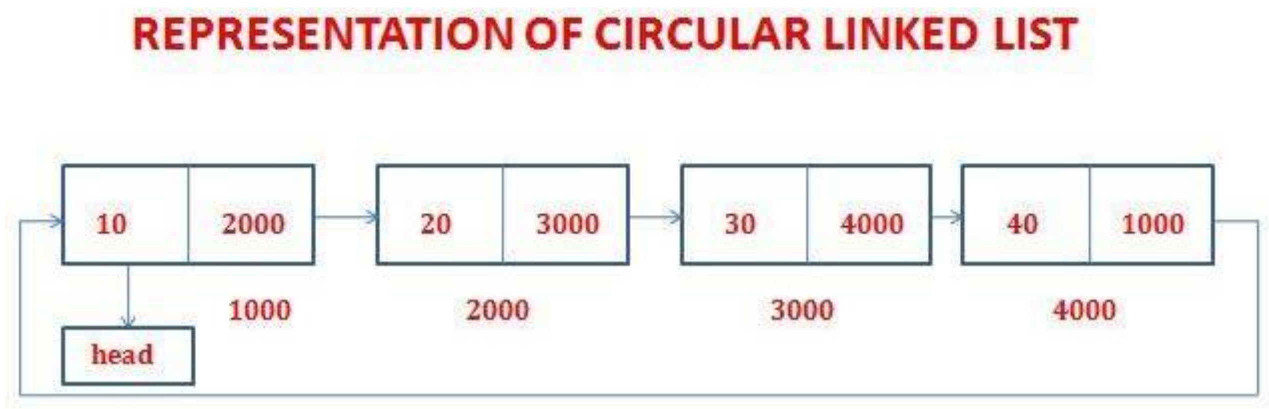
5.while temp is NULL, The element is not found.

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;
}
```

3.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

1. Creation :

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

Step 6- Check whether list is **Empty** (**head** == **NULL**).

Step 7 - If it is **Empty** then, set **head** = **ptr** and **ptr→next** = **head** and define a Node pointer '**temp**' and initialize with '**head**'.

Step 8- If it is **Not Empty** then, set **temp->next=ptr, temp=ptr** and **temp->next=head.**

Program:

```
#include<stdio.h>
#include<conio.h>
#include<process.h>
#include<stdlib.h>
struct node
{
        int data;
        struct node *next;
};
struct node *head=NULL;
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;
                if(head==NULL)
                {
                        head=ptr;
                        ptr->next=head;
                        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

Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the circular linked list...

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;
                }
        }
}
```

Inserting At End of the list

We can use the following steps to insert a new node at end of the circular linked list...

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;
                    }
          }
          }
```

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

We can use the following steps to insert a new node after a node in the circular linked list...

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:

```
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)
          {
                    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;
        }
   return 0;
}
```

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

We can use the following steps to delete a node from beginning of the circular linked list... **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

We can use the following steps to delete a node from end of the circular linked list... 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**.

```
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

We can use the following steps to delete a specific node from the single linked list...

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 **nth 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;
}
```

4. Displaying a circular Linked List

We can use the following steps to display the elements of a circular linked list...

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**.

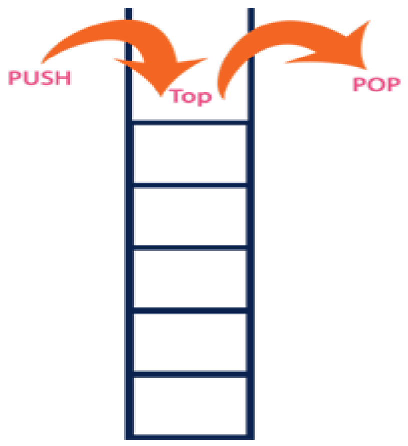
Program:

```
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

- 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 Last Out)** principle.
- In stack, the insertion and deletion operations are performed at only one end 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:



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.

Operations on a Stack

The following operations are performed on the stack

1. Push (To insert an element on to the stack)
2. Pop (To delete an element from the stack)
3. Display (To display elements of the stack)

STACK IMPLEMENTATION

Stack data structure can be implemented in two ways. They are as follows...

1. Stack using Array
2. Stack using Linked List

1.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.
- 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.
- Whenever we want to insert a value into the stack, increment the top value by one and then insert.
- Whenever we want to delete a value from the stack, then delete the top value and decrement the top value by one.

Stack Operations using Array

A stack can be implemented using array as follows...

Before implementing actual operations, first follow the below steps to create an empty stack.

- **Step 1 -** Include all the **header files** which are used in the program and define a constant **'SIZE'** with specific value.
- **Step 2 -** Declare all the **functions** used in stack implementation.
- **Step 3 -** Create a one dimensional array with fixed size (**int stack[SIZE]**)
- **Step 4 -** Define a integer variable **'top'** and initialize with **'-1'**. (**int top = -1**)
- **Step 5 -** In main method, display menu with list of operations and make suitable function calls to perform operation selected by the user on the stack.

1.push(value) - Inserting value into the stack

- In a stack, push() is a function used to insert an element into the stack.
- In a stack, the new element is always inserted at **top** position.
- Push function takes one integer value as parameter and inserts that value into the stack.

We can use the following steps to push an element on to the stack...

- **Step 1:** Check whether **stack** is **FULL**. (**top == SIZE-1**)
- **Step 2:** If it is **FULL**, then display **"Stack is FULL!!! Insertion is not possible!!!"** and terminate the function.
- **Step 3:** If it is **NOT FULL**, then increment **top** value by one (**top++**) and set stack[top] to value (**stack[top] = value**).

Example

- If we want to create a stack by inserting 10,20,30 and 40.
- Then 10 becomes the bottom-most element and 40 is the top-most element.
- The last inserted element 40 is at Top of the stack as shown in the below figure.

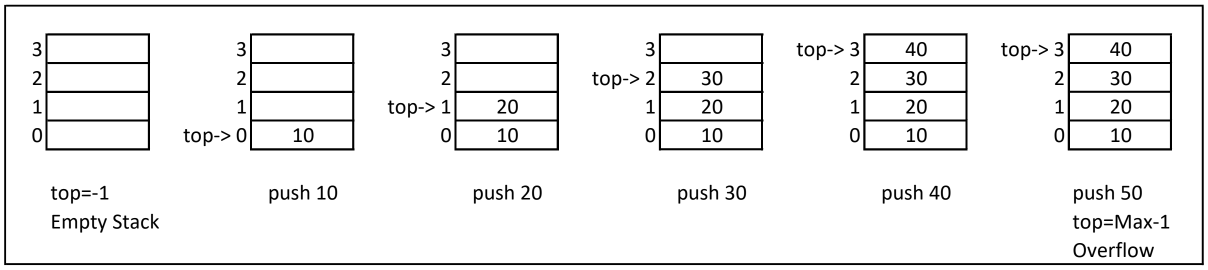


Figure:Push operations on Stack

// write a function to insert an element in to the stack.

```
void push(int value)
 {
   if(top == SIZE-1)
```

```
    printf("\nStack is Full!!! Insertion is not possible!!!");
  else
  {
    top++;
    stack[top] = value;
  }
}
```

2.pop() - Delete a value from the Stack

- In a stack, pop() is a function used to delete an element from the stack.
- In a stack, the element is always deleted from **top** position.
- Pop function does not take any value as parameter.

We can use the following steps to pop an element from the stack...

- **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--**).

Example:

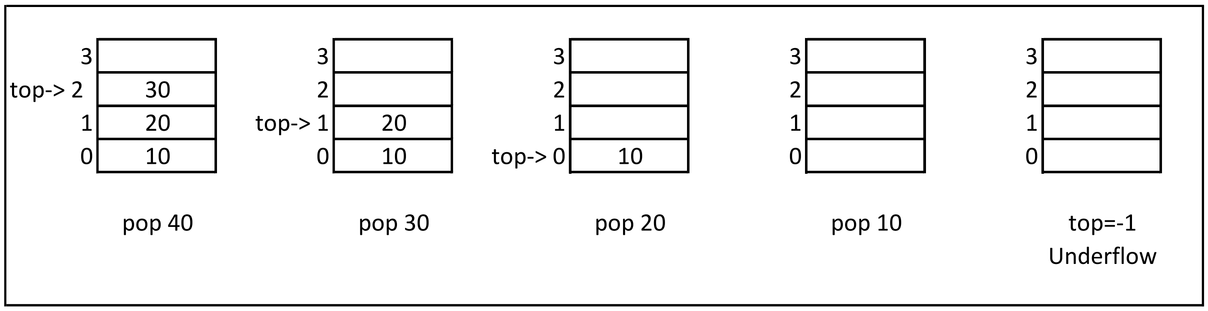


Fig:Pop operations on Stack

// write a function to delete an element from the stack.

```
void pop()
{
  if(top == -1)
    printf("\nStack is Empty!!! Deletion is not possible!!!");
  else
  {
    printf("\nDeleted : %d", stack[top]);
    top--;
  }
}
```

3.display() - Displays the elements of a Stack

We can use the following steps to display the elements of a stack...

- **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 3:** Repeat above step until **i** value becomes '0'.

// write a function to display an elements in the stack.

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

```
          }
        }
```

Implementation of Stack using Array

```
#include<stdio.h>
#include<conio.h>
#define SIZE 5
void push(int);
void pop();
void display();
int stack[SIZE], top = -1;
void main()
{
  int value, choice;
  clrscr();
  while(1)
   {
    printf("\n\n***** MENU *****\n");
    printf("1. Push\n2. Pop\n3. Display\n4. Exit");
    printf("\nEnter your choice: ");
    scanf("%d",&choice);
    switch(choice)
         {
           case 1: printf("Enter the value to be insert: ");
                     scanf("%d",&value);
                     push(value);
                     break;
           case 2: pop();
                     break;
```

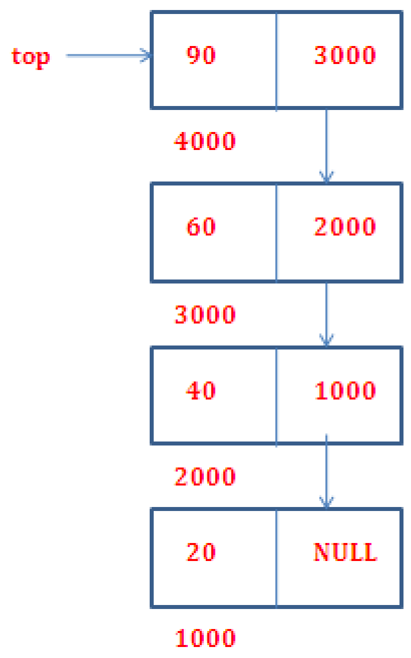
```
          case 3: display();
                    break;
          case 4: exit(0);
          default: printf("\nWrong selection!!! Try again!!!");
      }
   }
}
void push(int value)
 {
   if(top == SIZE-1)
      printf("\nStack is Full!!! Insertion is not possible!!!");
   else
   {
      top++;
      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()
```

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

2.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.

Stack Operations using Linked List

To implement a stack using a linked list, we need to set the following things before implementing actual operations.

- **Step 1 -** Include all the **header files** which are used in the program. And declare all the **user defined functions**.
- **Step 2 -** Define a '**Node**' structure with two members **data** and **next**.
- **Step 3 -** Define a **Node** pointer '**top**' and set it to **NULL**.
- **Step 4 -** Implement the **main** method by displaying Menu with list of operations and make suitable function calls in the **main** method.

1.push(value) - Inserting an element into the Stack

We can use the following steps to insert a new node into the stack...

- **Step 1 -** Create a **newNode** say **'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(int value)
{
  struct node *ptr;
  ptr = (struct node*)malloc(sizeof(struct node));
  ptr->data = value;
  if(top == NULL)
    ptr->next = NULL;
  else
    ptr->next = top;
  top = ptr;
}
```

2.pop() - Deleting an Element from a Stack

We can use the following steps to delete a node from the stack...

- **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**'.
- **Step 4 -** Then set '**top** = **top → next**'.
- **Step 5 -** Finally, delete '**temp**'. (**free(temp)**).

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

3.display() - Displaying stack of elements

We can use the following steps to display the elements (nodes) of a stack...

- **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**'.

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

Implementation of Stack using Linked List

```
#include<stdio.h>
#include<conio.h>
struct node
{
  int data;
  struct node *next;
```

```
}*top = NULL;

void push(int);
void pop();
void display();
void main()
{
   int choice, value;
   clrscr();
   printf("\n:: Stack using Linked List ::\n");
   while(1)
  {
      printf("\n****** MENU ******\n");
      printf("1. Push\n2. Pop\n3. Display\n4. Exit\n");
      printf("Enter your choice: ");
      scanf("%d",&choice);
      switch(choice)
         {
               case 1: printf("Enter the value to be insert: ");
                              scanf("%d", &value);
                              push(value);
                              break;
               case 2: pop(); break;
               case 3: display(); break;
               case 4: exit(0);
               default: printf("\nWrong selection!!! Please try again!!!\n");
          }
     }
}
void push(int value)
{
   struct node *ptr;
   ptr = (struct node*)malloc(sizeof(struct node));
  ptr->data = value;
  if(top == NULL)
    ptr->next = NULL;
  else
    ptr->next = top;
  top = ptr;
}
void pop()
{
  if(top == NULL)
    printf("\nStack is Empty!!!\n");
  else
  {
    struct node *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 = top;
    while(temp->next != NULL)
    {
           printf("%d--->",temp->data);
           temp = temp -> next;
    }
    printf("%d--->NULL",temp->data);
  }
```

}

QUEUE ADT

Queue data structure is a collection of similar data items in which insertion and deletion operations are performed based on **FIFO(First In First Out)** principle.

Queue is a linear data structure in which the insertion and deletion operations are performed at two different ends. The insertion is performed at one end called **'rear'** and deletion is performed at another end called **'front'**.

Operations on a Queue

The following operations are performed on a queue data structure...

1. **enQueue(value) - (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)**

Example

1) Initially front=rear=-1.It indicates Queue is empty

0	1	2	3

2) Insert 10

0	1	2	3
10			
front rear			

3) Insert 20

0	1	2	3
10	20		
front	rear		

4) Insert 30

0	1	2	3
10	20	30	
front		rear	

5) Inser 40

0	1	2	3
10	20	30	40
front			rear

6) Insert 50(Queue is Full)

0	1	2	3
10	20	30	40
front			rear

7) Delete(10 is removed)

0	1	2	3
	20	30	40
	front		rear

8)Delete(20 is removed)

0	1	2	3
		30	40
		front	rear

8)Delete(30 is removed)

0	1	2	3
			40
			front rear

9)Delete(40 is removed)

0	1	2	3

front= rear=-1

10)Delete(queue is empty)

0	1	2	3

front= rear=-1

QUEUE IMPLEMENTATION

Queue data structure can be implemented in two ways. They are as follows...

1. Queue using Array
2. Queue using Linked List

1.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.
- 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.

Queue Operations using Array

Queue data structure using array can be implemented as follows...
Before we implement actual operations, first follow the below steps to create an empty queue.

- **Step 1 -** Include all the **header files** which are used in the program and define a constant **'SIZE'** with specific value.
- **Step 2 -** Declare all the **user defined functions** which are used in queue implementation.
- **Step 3 -** Create a one dimensional array with above defined SIZE (**int queue[SIZE]**)
- **Step 4 -** Define two integer variables **'front'** and '**rear**' and initialize both with **'-1'**. (**int front = -1, rear = -1**)
- **Step 5 -** Then implement main method by displaying menu of operations list and make suitable function calls to perform operation selected by the user on queue.

1.enQueue(value) - Inserting value into the queue

- 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 takes one integer value as a parameter and inserts that value into the queue.

We can use the following steps to insert an element into the queue...

- **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**.

```
void enQueue(int value)
{
  if(rear == SIZE-1)
    printf("\nQueue is Full!!! Insertion is not possible!!!");
  else
   {
    if(front == -1)
   front = 0;
    rear++;
    queue[rear] = value;
    printf("\nInsertion success!!!");
  }
}
```

2.deQueue() - Deleting a value from the Queue

- 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.
- The deQueue() function does not take any value as parameter.

We can use the following steps to delete an element from the queue...

- **Step 1 -** Check whether **queue** is **EMPTY**. (**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. Then check whether both **front** and **rear** are equal (**front** == **rear**), if it **TRUE**, then set both **front** and **rear** to '**-1**' (**front** = **rear** = **-1**).

```
void deQueue()
{
  if(front == rear)
```

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

3.display() - Displays the elements of a Queue

We can use the following steps to display the elements of a queue...

- **Step 1 -** Check whether **queue** is **EMPTY**. (**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+1**'.
- **Step 4 -** Display '**queue[i]**' value and increment '**i**' value by one (**i++**). Repeat the same until '**i**' value reaches to **rear** (**i** <= **rear**)

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

Implementation of Queue Datastructure using Array

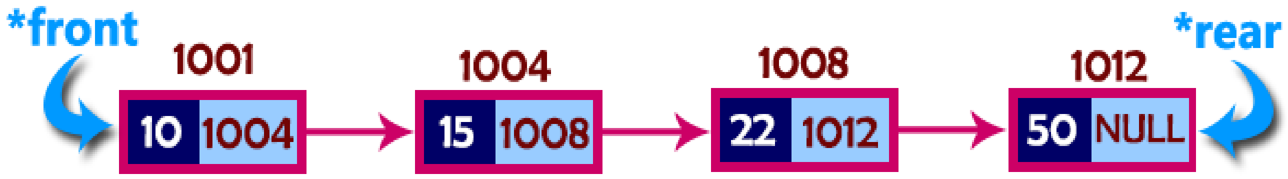
```
#include<stdio.h>
#include<conio.h>
#define SIZE 10
void enQueue(int);
void deQueue();
void display();
int queue[SIZE], front = -1, rear = -1;
void main()
{
  int value, choice;
  clrscr();
  while(1)
   {
    printf("\n\n***** MENU *****\n");
    printf("1. Insertion\n2. Deletion\n3. Display\n4. Exit");
    printf("\nEnter your choice: ");
    scanf("%d",&choice);
    switch(choice)
      {
           case 1: printf("Enter the value to be insert: ");
                      scanf("%d",&value);
                      enQueue(value);
                      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;
    rear++;
    queue[rear] = value;
    printf("\nInsertion success!!!");
  }
}
void deQueue()
{
  if(front == rear)
    printf("\nQueue is Empty!!! Deletion is not possible!!!");
  else
   {
    printf("\nDeleted : %d", queue[front]);
    front++;
    if(front == rear)
           front = rear = -1;
   }
}
void display()
 {
  if(rear == -1)
    printf("\nQueue is Empty!!!");
  else
   {
    int i;
    printf("\nQueue elements are:\n");
    for(i=front; i<=rear; i++)
            printf("%d\t",queue[i]);
  }
}
```

2.Queue Using Linked List

- 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.

Queue Operations

To implement queue using linked list, we need to set the following things before implementing actual operations.

- **Step 1 -** Include all the **header files** which are used in the program. And declare all the **user defined functions**.
- **Step 2 -** Define a '**Node**' structure with two members **data** and **next**.
- **Step 3 -** Define two **Node** pointers '**front**' and '**rear**' and set both to **NULL**.
- **Step 4 -** Implement the **main** method by displaying Menu of list of operations and make suitable function calls in the **main** method to perform user selected operation.

1.enQueue(value) - Inserting an element into the Queue

We can use the following steps to insert a new node into the queue...

- **Step 1 -** Create a **newNode** say ‘**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** = **rear**.
- **Step 4 -** If it is **Not Empty** then, set **rear → next** = **ptr** and **rear** = **ptr**.

```
void insert(int value)
{
  struct node *ptr;
  ptr = (struct node*)malloc(sizeof(struct node));
  ptr->data = value;
  ptr -> next = NULL;
  if(front == NULL)
    front = rear = ptr;
  else
  {
    rear -> next = ptr;
    rear = ptr;
  }
}
```

2.deQueue() - Deleting an Element from Queue

We can use the following steps to delete a node from the queue...

- **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)**).

```
void delete()
{
  if(front == NULL)
    printf("\nQueue is Empty!!!\n");
```

```
          else
           {
             struct node *temp = front;
             front = front -> next;
             printf("\nDeleted element: %d\n", temp->data);
             free(temp);
           }
        }
```

3.display() - Displaying the elements of Queue

We can use the following steps to display the elements (nodes) of a queue...

- **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**'.

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

Implementation of Queue Datastructure using Linked List

```
#include<stdio.h>
#include<conio.h>
struct node
{
  int data;
  struct node *next;
}*front = NULL,*rear = NULL;
void insert(int);
void delete();
void display();
void main()
{
  int choice, value;
  clrscr();
  printf("\n:: Queue Implementation using Linked List ::\n");
  while(1)
  {
    printf("\n****** MENU ******\n");
    printf("1. Insert\n2. Delete\n3. Display\n4. Exit\n");
    printf("Enter your choice: ");
    scanf("%d",&choice);
    switch(choice)
    {
            case 1: printf("Enter the value to be insert: ");
                        scanf("%d", &value);
                        insert(value);
                        break;
            case 2: delete(); break;
            case 3: display(); break;
            case 4: exit(0);
            default: printf("\nWrong selection!!! Please try again!!!\n");
    }
  }
}
void insert(int value)
{
  struct node *ptr;
  ptr = (struct node*)malloc(sizeof(struct node));
  ptr->data = value;
  ptr -> next = NULL;
  if(front == NULL)
    front = rear = ptr;
  else
  {
    rear -> next = ptr;
    rear = ptr;
  }
}
void delete()
{
  if(front == NULL)
    printf("\nQueue is Empty!!!\n");
  else
   {
    struct node *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 = front;
    while(temp->next != NULL)
    {
           printf("%d--->",temp->data);
           temp = temp -> next;
    }
    printf("%d--->NULL\n",temp->data);
  }
}
```

STACK APPLICATIONS

Stacks can be used for Conversion from one form of expression to another.

What is an Expression?

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

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.

we can convert an expression from one form to another form like

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**

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**

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'	+ (*	AB C
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	NIL L	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	Stack is Empty	CBA*+

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.N o	**Scanned Character / Symbol**	**Operations**	**Stac k**	**Evaluated Part of Expression**
1	Initially	Stack is Empty		Nothing
2	5	Push (5)	5	Nothing
3	3	Push (3)	3 5	Nothing

4	+	pop two operands	8	value1 =5
				value2 =3
				(5 + 3) = 8

5	8	Push (8)	8 8	(5 + 3) = 8
6	2	Push (2)	2 8 8	(5 + 3) = 8

7	-	pop two operands	6
			8
				value1=2
				value2=8
				(8-2)=6

8	*	pop two operands	48	value1=6
				value2=8
				(8 * 6)=48

9	Nill	Pop elements from stack.	Stack is Empty	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.

DATA STRUCTURES

ASSIGNMENT - 1

1. Convert Infix Expression to Postfix Expression A) K + L - M * N + (O ^ P) * W / U / V * T + Q B) (H + G) / (F / E) + (D - (C * (B ^ A))) C) 2 * 3 / (2 - 1) + 5 * (4 - 1)

2. Convert Infix Expression to Postfix Expression A) K + L - M * N + (O ^ P) * W / U / V * T + Q B) (H + G) / (F / E) + (D - (C * (B ^ A))) C) 2 * 3 / (2 - 1) + 5 * (4 - 1)

3. Evaluate Postfix form : 5 9 8 + 4 6 * + 7 - *

4. Evaluate Postfix form : 10 2 8 * + 3 -

5. Check whether a given expression is balanced or unbalanced

[( )] { } { [( ) ( )] ( ) }

DATA STRUCTURES

ASSIGNMENT - 1

6. Convert Infix Expression to Postfix Expression A) K + L - M * N + (O ^ P) * W / U / V * T + Q B) (H + G) / (F / E) + (D - (C * (B ^ A))) C) 2 * 3 / (2 - 1) + 5 * (4 - 1)

7. Convert Infix Expression to Postfix Expression A) K + L - M * N + (O ^ P) * W / U / V * T + Q B) (H + G) / (F / E) + (D - (C * (B ^ A))) C) 2 * 3 / (2 - 1) + 5 * (4 - 1)

8. Evaluate Postfix form : 5 9 8 + 4 6 * + 7 - *

9. Evaluate Postfix form : 10 2 8 * + 3 -

10. Check whether a given expression is balanced or unbalanced

[( )] { } { [( ) ( )] ( ) }