

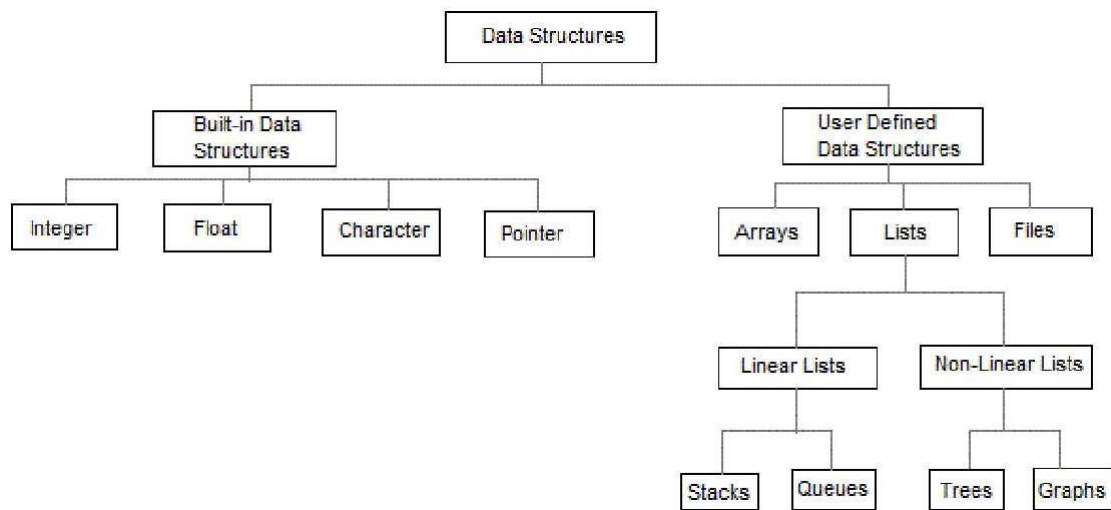
# UNIT-01

**Data structure** A data structure is a specialized format for organizing and storing data. General data structure types include the array, the file, the record, the table, the tree, and so on. Any data structure is designed to organize data to suit a specific purpose so that it can be accessed and worked with in appropriate ways

## Abstract Data Type

In computer science, an abstract data type (ADT) is a mathematical model for data types where a data type is defined by its behavior (semantics) from the point of view of a user of the data, specifically in terms of possible values, possible operations on data of this type, and the behavior of these operations. When a class is used as a type, it is an abstract type that refers to a hidden representation. In this model an ADT is typically implemented as a class, and each instance of the ADT is usually a n object of that class.

In ADT all the implementation details are hidden



- Linear data structures are the data structures in which data is arranged in a list or in a sequence.
- Non linear data structures are the data structures in which data may be arranged in a hierarchical manner

## LIST ADT

List is basically the collection of elements arranged in a sequential manner. In memory we can store the list in two ways: one way is we can store the elements in sequential memory locations. That means we can store the list in arrays.

The other way is we can use pointers or links to associate elements sequentially.

This is known as linked list.

## LINKED LISTS

The linked list is very different type of collection from an array. Using such lists, we can store collections of information limited only by the total amount of memory that the OS will allow us to use. Furthermore, there is no need to specify our needs in advance. The linked list is very flexible dynamic data structure : items may be added to it or deleted from it at will. A programmer need not worry about how many items a program will have to accommodate in advance. This allows us to write robust programs which require much less maintenance.

**The linked allocation has the following draw backs:**

1. No direct access to a particular element.
2. Additional memory required for pointers.

**Linked list are of 3 types:**

1. Singly Linked List
2. Doubly Linked List
3. Circularly Linked List

### **SINGLY LINKED LIST**

A singly linked list, or simply a linked list, is a linear collection of data items. The linear order is given by means of POINTERS. These types of lists are often referred to as **linear linked list**.

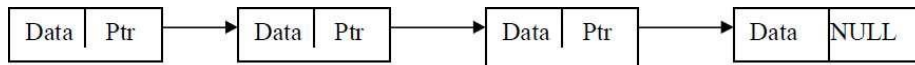
\* Each item in the list is called a node.

\* Each node of the list has two fields:

1. Information- contains the item being stored in the list.
2. Next address- contains the address of the next item in the list.

\* The last node in the list contains NULL pointer to indicate that it is the end of the list.

Conceptual view of Singly Linked List



Operations on Singly linked list:

- Insertion of a node
- Deletions of a node
- Traversing the list

**Structure of a node:**

**Method -1:**

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



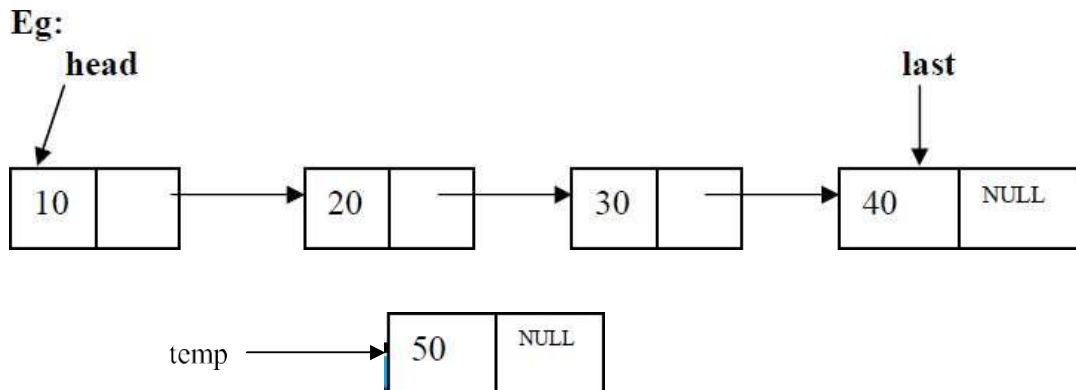
**Method -2:**

```
class node
{
public:
    int data;
    node *link;
};
```

**Insertions:** To place an elements in the list there are 3 cases :

1. At the beginning
2. End of the list
3. At a given position

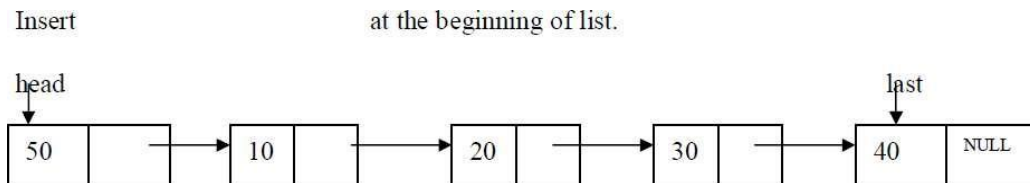
**case 1:Insert at the beginning**



**head** is the pointer variable which contains address of the first node and **temp** contains address of new node to be inserted then sample code is

```
temp->link=head;
head=temp;
```

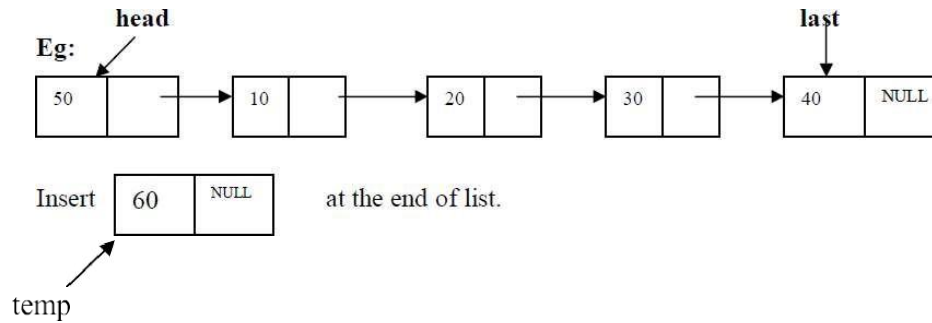
**After insertion:**



**Code for insert front:-**

```
template <class T>
void list<T>::insert_front()
{
    struct node <T> *t, *temp;
    cout<<"Enter data into node:";
    cin>>item;
    temp=create_node(item);
    if(head==NULL)
        head=temp;
    else
    {
        temp->link=head;
        head=temp;
    }
}
```

**case 2:Inserting end of the list**

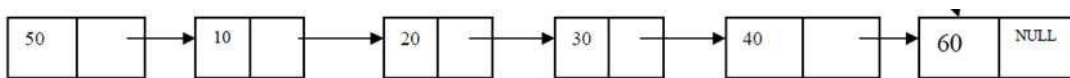


**head** is the pointer variable which contains address of the first node and **temp** contains address of new node to be inserted then sample code is

```

t=head;
while(t->link!=NULL)
{
    t=t->link;
}
t->link=temp;
  
```

After insertion the linked list is



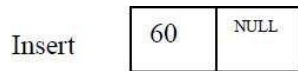
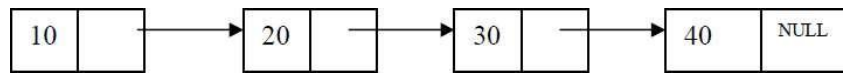
**Code for insert End:-**

```

template <class T>
void list<T>::insert_end()
{
    struct node<T> *t,*temp;
    int n;
    cout<<"Enter data into node:";
    cin>>n;
    temp=create_node(n);
    if(head==NULL)
        head=temp;
    else
    {
        t=head;
        while(t->link!=NULL)
            t=t->link;
        t->link=temp;
    }
}
  
```



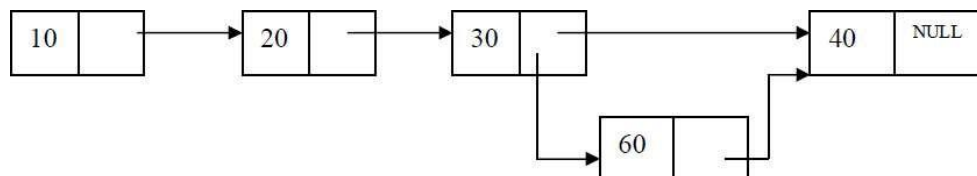
### case 3: Insert at a position



insert node at position 3

**head** is the pointer variable which contains address of the first node and **temp** contains address of new node to be inserted then sample code is

```
c=1;
while(c<pos)
{
    prev=cur;
    cur=cur->link;
    c++;
}
prev->link=temp;
temp->link=cur;
```



### Code for inserting a node at a given position:-

```
template <class T>
void list<T>::Insert_at_pos(int pos)
{struct node<T>*cur,*prev,*temp;
int c=1;
    cout<<"Enter data into node:";
    cin>>item
    temp=create_node(item);
    if(head==NULL)
        head=temp;
    else
    {
        prev=cur=head;
        if(pos==1)
        {
            temp->link=head;
```

```

        head=temp;
    }
    else
    {
        while(c<pos)
        {
            c++;
            prev=cur;
            cur=cur->link;
        }
        prev->link=temp;
        temp->link=cur;
    }
}

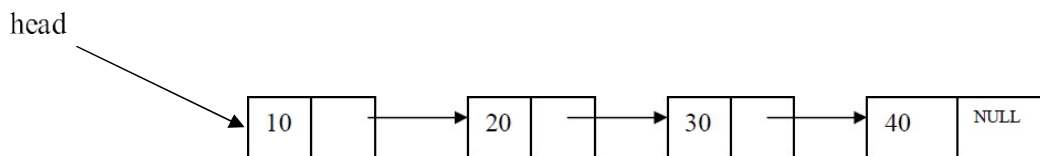
```

**Deletions:** Removing an element from the list, without destroying the integrity of the list itself.

To place an element from the list there are 3 cases :

1. Delete a node at beginning of the list
2. Delete a node at end of the list
3. Delete a node at a given position

Case 1: Delete a node at beginning of the list



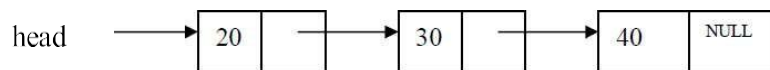
**head** is the pointer variable which contains address of the first node

sample code is

```

t=head;
head=head->link;
cout<<"node "<<t->data<<" Deletion is sucess";
delete(t);

```



**code for deleting a node at front**

```

template <class T>
void list<T>::delete_front()
{
    struct node<T>*t;
    if(head==NULL)
        cout<<"List is Empty\n";
    else
    {
        t=head;

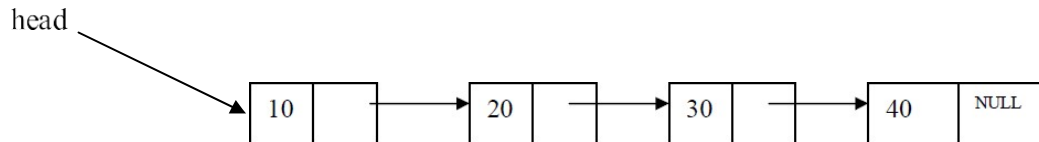
```

```

        head=head->link;
        cout<<"node "<<t->data<<" Deletion is sucess";
        delete(t);
    }
}

```

## Case 2. Delete a node at end of the list

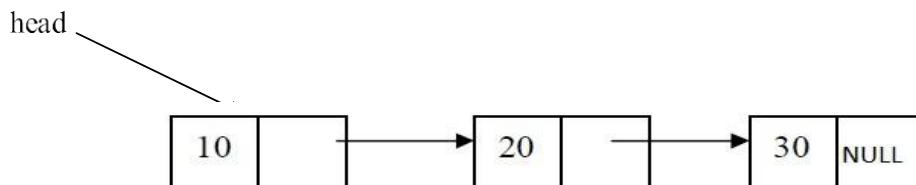


To delete last node , find the node using following code

```

struct node<T> *cur, *prev;
cur=prev=head;
while(cur->link!=NULL)
{
    prev=cur;
    cur=cur->link;
}
prev->link=NULL;
cout<<"node "<<cur->data<<" Deletion is sucess";
free(cur);

```



## code for deleting a node at end of the list

```

template <class T>
void list<T>::delete_end()
{
    struct node<T> *cur, *prev;
    cur=prev=head;
    if(head==NULL)
        cout<<"List is Empty\n";
    else
    {
        cur=prev=head;
        if(head->link==NULL)
        {
            cout<<"node "<<cur->data<<" Deletion is sucess";
            free(cur);
            head=NULL;
        }
    }
}

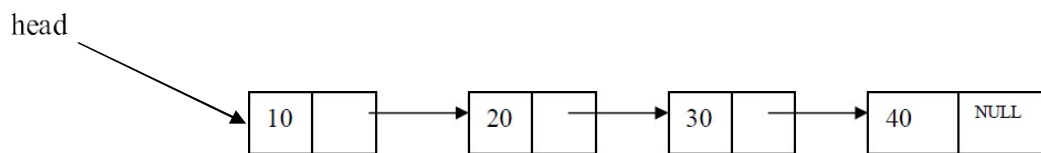
```

```

else
{
    while(cur->link!=NULL)
    {
        prev=cur;
        cur=cur->link;
    }
    prev->link=NULL;
    cout<<"node "<<cur->data<<" Deletion is sucess";
    free(cur);
}
}
}

```

### CASE 3. Delete a node at a given position



Delete node at position 3

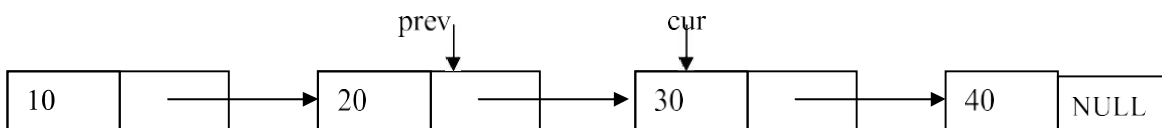
**head** is the pointer variable which contains address of the first node. Node to be deleted is node containing value 30.

Finding node at position 3

```

c=1;
while(c<pos)
{
    c++;
    prev=cur;
    cur=cur->link;
}

```



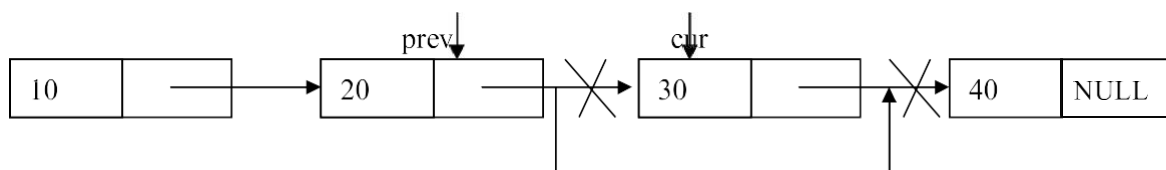
cur is the node to be deleted . before deleting update links

code to update links

```

prev->link=cur->link;
cout<<cur->data <<"is deleted successfully";
delete cur;

```



**Traversing the list:** Assuming we are given the pointer to the head of the list, how do we get the end of the list.

```
template <class T>
void list<T>::display()
{
    struct node<T>*t;

    if(head==NULL)
    {
        cout<<"List is Empty\n";
    }
    else
    {
        t=head;
        while(t!=NULL)
        {
            cout<<t->data<<"->";
            t=t->link;
        }
    }
}
```

### Dynamic Implementation of list ADT

```
#include<iostream.h>
#include<stdlib.h>
template <class T>
struct node
{
    T data;
    struct node<T> *link;
};
template <class T>
class list
{
    int item;
    struct node<T>*head;
public:
    list();
    void display();
    struct node<T>*create_node(int n);
    void insert_end();
    void insert_front();
    void Insert_at_pos(int pos);
    void delete_end();
    void delete_front();
    void Delete_at_pos(int pos);
    void Node_count();
};
```

```

template <class T>
list<T>::list()
{
    head=NULL;
}

template <class T>
void list<T>:: display()
{
    struct node<T>*t;

    if(head==NULL)
    {
        cout<<"List is Empty\n";
    }
    else
    {
        t=head;
        while(t!=NULL)
        {
            cout<<t->data<<"->";
            t=t->link;
        }
    }
}

template <class T>
struct node<T>* list<T>::create_node(int n)
{struct node<T> *t;
    t=new struct node<T>;
    t->data=n;
    t->link=NULL;
return t;
}

template <class T>
void list<T>::insert_end()
{struct node<T> *t,*temp;
int n;
    cout<<"Enter data into node:";
    cin>>n;
    temp=create_node(n);
    if(head==NULL)
        head=temp;
    else
    {
        t=head;
        while(t->link!=NULL)
            t=t->link;
        t->link=temp;
    }
}

```

```

template <class T>
void list<T>::insert_front()
{
    struct node <T>*t,*temp;
        cout<<"Enter data into node:";
        cin>>item;
        temp=create_node(item);
        if(head==NULL)
            head=temp;
        else
        {
            temp->link=head;
            head=temp;
        }
    }

template <class T>
void list<T>::delete_end()
{
    struct node<T>*cur,*prev;
        cur=prev=head;
        if(head==NULL)
            cout<<"List is Empty\n";
        else
        {
            cur=prev=head;
            if(head->link==NULL)
            {
                cout<<"node "<<cur->data<<" Deletion is sucess";
                free(cur);
                head=NULL;
            }
            else
            {
                while(cur->link!=NULL)
                {
                    prev=cur;
                    cur=cur->link;
                }
                prev->link=NULL;
                cout<<"node "<<cur->data<<" Deletion is sucess";
                free(cur);
            }
        }
    }

template <class T>
void list<T>::delete_front()
{
    struct node<T>*t;
        if(head==NULL)
            cout<<"List is Empty\n";
        else
        {
            t=head;
            head=head->link;

            cout<<"node "<<t->data<<" Deletion is sucess";
            delete(t);

```

S

```

    }
}

template <class T>
void list<T>::Node_count()
{
    struct node<T>*t;
    int c=0;
    t=head;
    if(head==NULL)
    {
        cout<<"List is Empty\n";

    }
    else
    {
        while(t!=NULL)
        {
            c++;
            t=t->link;
        }
        cout<<"Node Count="<<c<<endl;
    }
}

template <class T>
void list<T>::Insert_at_pos(int pos)
{struct node<T>*cur,*prev,*temp;
int c=1;
    cout<<"Enter data into node:";
        cin>>item
        temp=create_node(item);
    if(head==NULL)
        head=temp;
    else
    { prev=cur=head;
        if(pos==1)
        {
            temp->link=head;
            head=temp;
        }
        else
        {
            while(c<pos)
            {
                c++;
                prev=cur;
                cur=cur->link;
            }
            prev->link=temp;
            temp->link=cur;
        }
    }
}

```



```

    }
}

template <class T>
void list<T>::Delete_at_pos(int pos)
{
    struct node<T>*cur,*prev,*temp;
    int c=1;

    if(head==NULL)
    {
        cout<<"List is Empty\n";
    }
    else
    { prev=cur=head;
      if(pos==1)
      {
          head=head->link;
          cout<<cur->data <<"is deleted sucesfully";
          delete cur;
      }
      else
      {
          while(c<pos)
          {
              c++;
              prev=cur;
              cur=cur->link;
          }
          prev->link=cur->link;
          cout<<cur->data <<"is deleted sucesfully";
          delete cur;
      }
    }
}

int main()
{
    int ncount,ch,pos;
    list <int> L;
    while(1)
    {
        cout<<"\n ***Operations on Linked List***"<<endl;
        cout<<"1.Insert node at End"<<endl;
        cout<<"2.Insert node at Front"<<endl;
        cout<<"3.Delete node at END"<<endl;
        cout<<"4.Delete node at Front"<<endl;
        cout<<"5.Insert at a position "<<endl;
        cout<<"6.Delete at a position "<<endl;
        cout<<"7.Node Count"<<endl;
        cout<<"8.Display nodes "<<endl;
        cout<<"9.Clear Screen "<<endl;
    }
}

```

```

cout<<"10.Exit "<<endl;
cout<<"Enter Your choice:";
cin>>ch;
switch(ch)
{
    case 1: L.insert_end();
            break;
    case 2: L.insert_front();
            break;
    case 3: L.delete_end();
            break;
    case 4: L.delete_front();
            break;
    case 5: cout<<"Enter position to insert";
            cin>>pos;
            L.Insert_at_pos(pos);
            break;
    case 6: cout<<"Enter position to insert";
            cin>>pos;
            L.Delete_at_pos(pos);
            break;
    case 7: L.Node_count();
            break;
    case 8: L.display();
            break;
    case 9: system("cls");
            break;
    case 10: exit(0);

    default: cout<<"Invalid choice";

}
}
}

```

### DOUBLY LINKED LIST

A singly linked list has the disadvantage that we can only traverse it in one direction. Many applications require searching backwards and forwards through sections of a list. A useful refinement that can be made to the singly linked list is to create a doubly linked list. The distinction made between the two list types is that while singly linked list have pointers going in one direction, doubly linked list have pointer both to the next and to the previous element in the list. The main advantage of a doubly linked list is that, they permit traversing or searching of the list in both directions.

In this linked list each node contains three fields.

- a) One to store data
- b) Remaining are self referential pointers which points to previous and next nodes in the list

prev	data	next
------	------	------

## Implementation of node using structure

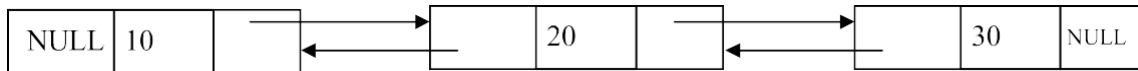
### Method -1:

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

## Implementation of node using class

### Method -2:

```
class node
{
public:
    int data;
    node *prev;
    node * next;
};
```



Operations on Doubly linked list:

- Insertion of a node
- Deletions of a node
- Traversing the list

### Doubly linked list ADT:

```
template <class T>
class dlist
{
    int data;
    struct dnode<T>*head;
public:
    dlist()
    {
        head=NULL;
    }
    void display();
    struct dnode<T>*create_dnode(int n);
    void insert_end();
    void insert_front();
    void delete_end();
    void delete_front();
    void dnode_count();
};
```

```

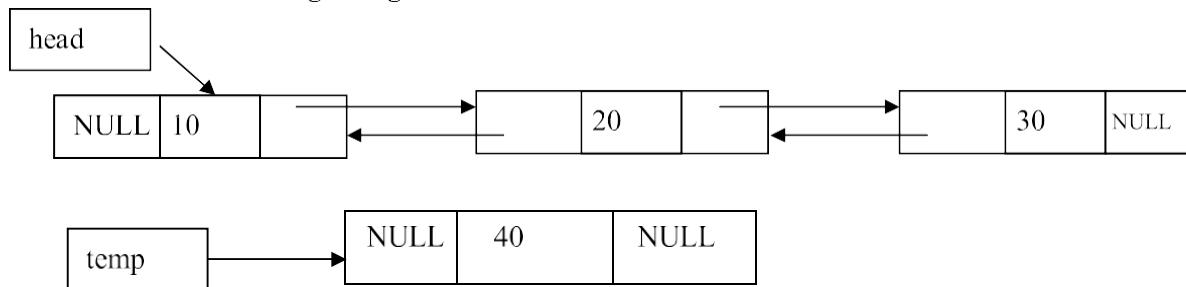
void Insert_at_pos(int pos);
void Delete_at_pos(int pos);
};

```

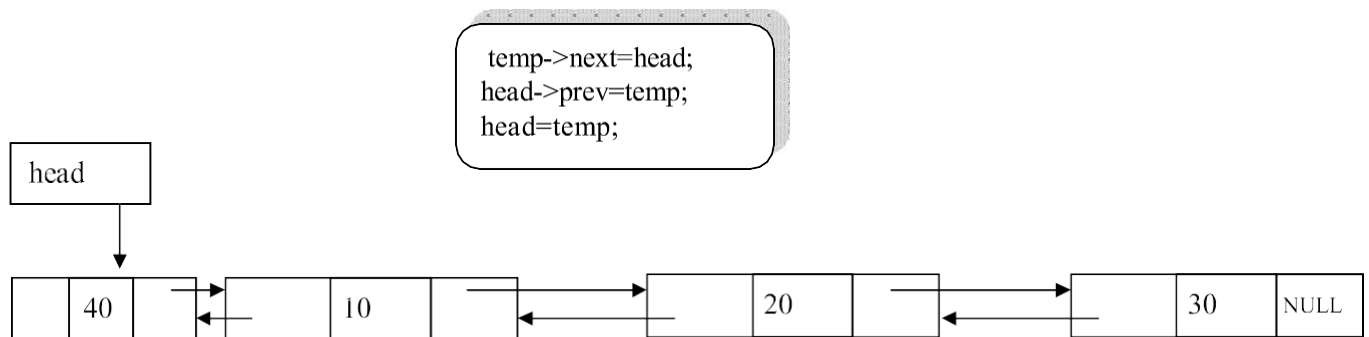
**Insertions:** To place an elements in the list there are 3 cases

- 1. At the beginning
- 2. End of the list
- 3. At a given position

**case 1: Insert at the beginning**



**head** is the pointer variable which contains address of the first node and **temp** contains address of new node to be inserted then sample code is



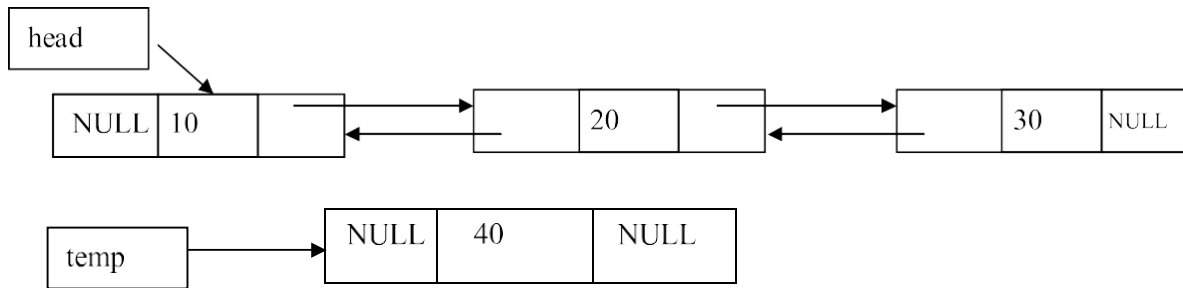
**Code for insert front:-**

```

template <class T>
void DLL<T>::insert_front()
{
    struct dnode <T>*t,*temp;
    cout<<"Enter data into node:";
    cin>>data;
    temp=create_dnode(data);
    if(head==NULL)
        head=temp;
    else
    { temp->next=head; head-
      >prev=temp;
      head=temp;
    }
}

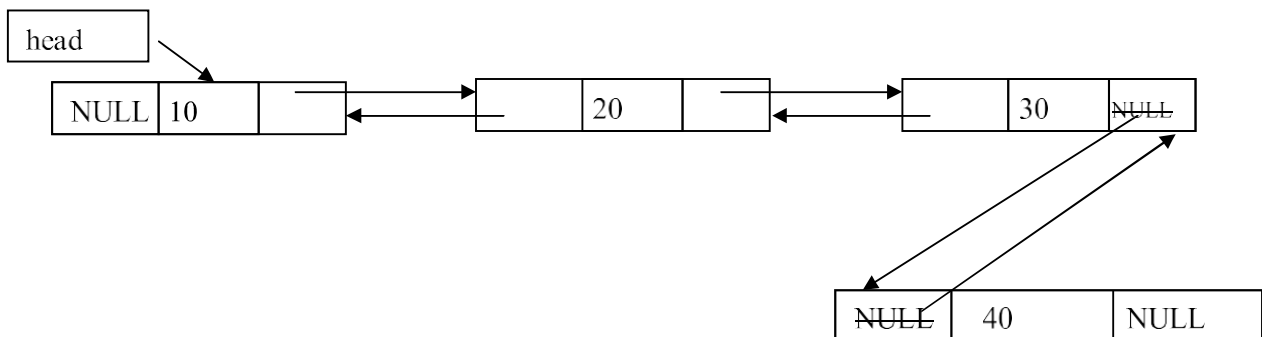
```

**case 2: Inserting end of the list**



head is the pointer variable which contains address of the first node and temp contains address of new node to be inserted then sample code is

```
t=head;
while(t->next!=NULL)
    t=t->next;
t->next=temp;
temp->prev=t;
```



**Code to insert a node at End:-**

```
template <class T>
void DLL<T>::insert_end()
{
    struct dnode<T> *t,*temp;
    int n;

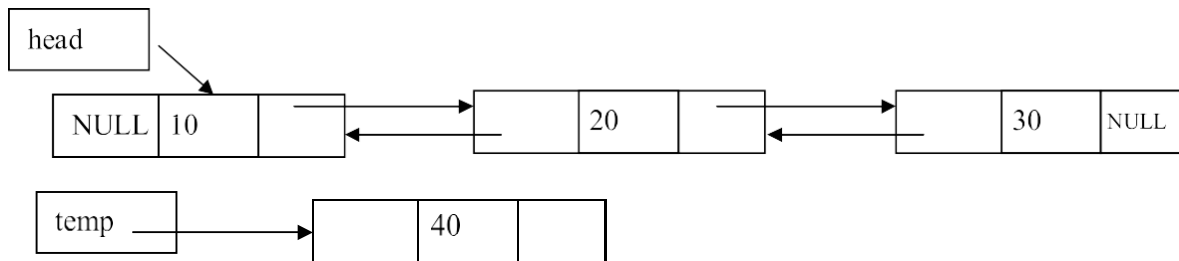
    cout<<"Enter data into dnode:";
    cin>>n;
    temp=create_dnode(n);
    if(head==NULL)
        head=temp;
    else
    {
        t=head;
        while(t->next!=NULL)
            t=t->next;
```

```

        t->next=temp;
        temp->prev=t;
    }
}

```

**case 3:** Inserting at a give position



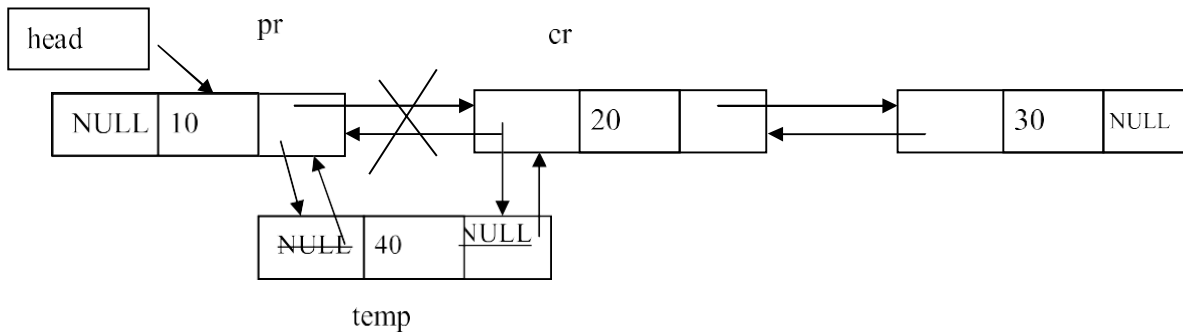
insert 40 at position 2

**head** is the pointer variable which contains address of the first node and **temp** contains address of new node to be inserted then sample code is

```

while(count<pos)
{
    count++;
    pr=cr;
    cr=cr->next;
}
pr->next=temp;
temp->prev=pr;
temp->next=cr;
cr->prev=temp;

```



### Code to insert a node at a position

```
template <class T>
void dlist<T>::Insert_at_pos(int pos)
{
    struct dnode<T>*cr,*pr,*temp;
    int count=1;
    cout<<"Enter data into dnode:";
    cin>>data;
    temp=create_dnode(data);
    display();
    if(head==NULL)
    { //when list is empty
        head=temp;
    }
    else
    {
        pr=cr=head;
        if(pos==1)
        { //inserting at pos=1
            temp->next=head;
            head=temp;
        }
        else
        {
            while(count<pos)
            {
                count++;
                pr=cr;
                cr=cr->next;
            }
            pr->next=temp;
            temp->prev=pr;
            temp->next=cr;
            cr->prev=temp;
        }
    }
}
```

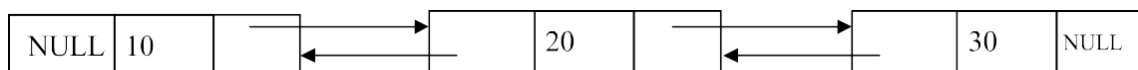
**Deletions:** Removing an element from the list, without destroying the integrity of the list itself.

To place an element from the list there are 3 cases :

1. Delete a node at beginning of the list
2. Delete a node at end of the list
3. Delete a node at a given position

Case 1: Delete a node at beginning of the list

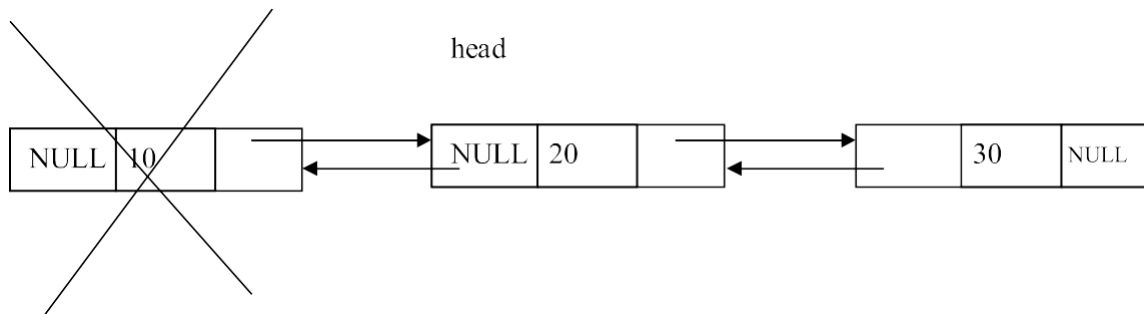
head



**head** is the pointer variable which contains address of the first node

sample code is

```
t=head;
head=head->next;
head->prev=NULL;
cout<<"dnode "<<t->data<<" Deletion is sucess";
delete(t);
```



**code for deleting a node at front**

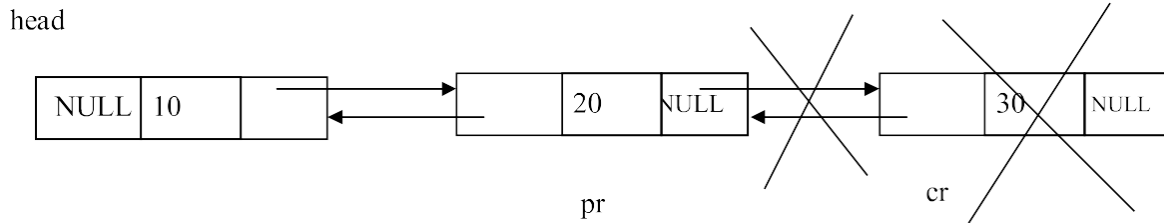
```
template <class T>
void dlist<T>:: delete_front()
{struct dnode<T>*t;
  if(head==NULL)
    cout<<"List is Empty\n";
  else
  {   t=head;
      head=head->next;
      head->prev=NULL;
      cout<<"dnode "<<t->data<<" Deletion is sucess";
      delete(t);
  }
}
```

### **Case 2. Delete a node at end of the list**

To deleted the last node find the last node. find the node using following code

```
struct dnode<T>*pr,*cr;
pr=cr=head;
while(cr->next!=NULL)
{   pr=cr;
    cr=cr->next;
}
pr->next=NULL;
cout<<"dnode "<<cr->data<<" Deletion is sucess";
delete(cr);
```

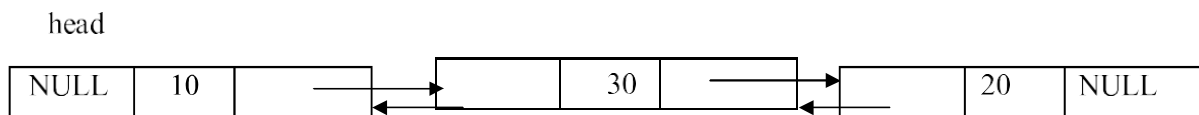




### code for deleting a node at end of the list

```
template <class T>
void dlist<T>::delete_end()
{
    struct dnode<T>*pr,*cr;
    pr=cr=head;
    if(head==NULL)
        cout<<"List is Empty\n";
    else
    {
        cr=pr=head;
        if(head->next==NULL)
        {
            cout<<"dnode "<<cr->data<<" Deletion is sucess";
            delete(cr);
            head=NULL;
        }
        else
        {
            while(cr->next!=NULL)
            {
                pr=cr;
                cr=cr->next;
            }
            pr->next=NULL;
            cout<<"dnode "<<cr->data<<" Deletion is sucess";
            delete(cr);
        }
    }
}
```

### CASE 3. Delete a node at a given position



Delete node at position 2

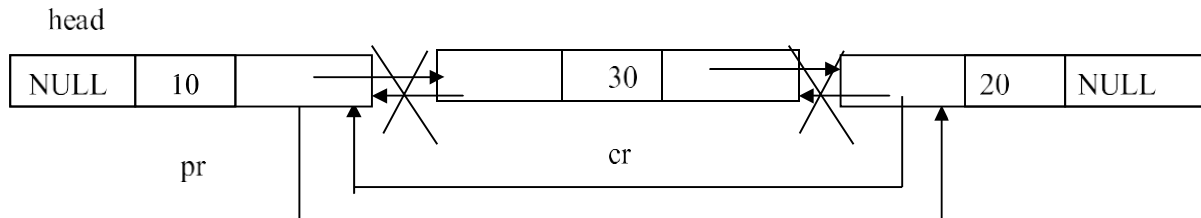
**head** is the pointer variable which contains address of the first node. Node to be deleted is node containing value 30.

Finding node at position 2.

```

while(count<pos)
{
    pr=cr;
    cr=cr->next;
    count++;
}
pr->next=cr->next;
cr->next->prev=pr;

```



### code for deleting a node at a position

```

template <class T>
void dlist<T>::Delete_at_pos(int pos)
{
    struct dnode<T>*cr,*pr,*temp;
    int count=1;
    display();
    if(head==NULL)
    {
        cout<<"List is Empty\n";
    }
    else
    {
        pr=cr=head;
        if(pos==1)
        {
            head=head->next;
            head->prev=NULL;
            cout<<cr->data <<"is deleted sucesfully";
            delete cr;
        }
        else
        {
            while(count<pos)
            {
                count++;
                pr=cr;
                cr=cr->next;
            }
            pr->next=cr->next;
            cr->next->prev=pr;
            cout<<cr->data <<"is deleted sucesfully";
            delete cr;
        }
    }
}

```

## Dynamic Implementation of Doubly linked list ADT

```
#include<iostream.h>
template <class T>
struct dnode
{
    T data;
    struct dnode<T> *prev;
    struct dnode<T> *next;
};
template <class T>
class dlist
{
    int data;
    struct dnode<T>*head;
public:
    dlist();
    struct dnode<T> *create_dnode(int n);
    void insert_front();
    void insert_end();
    void Insert_at_pos(int pos);
    void delete_front();
    void delete_end();
    void Delete_at_pos(int pos);
    void dnode_count();
    void display();
};

template <class T>
dlist<T>::dlist()
{
    head=NULL;
}

template <class T>
struct dnode<T> *dlist<T>::create_dnode(int n)
{
    struct dnode<T> *t;
    t=new struct dnode<T>;
    t->data=n;
    t->next=NULL;
    t->prev=NULL;
    return t;
}
template <class T>
void dlist<T>::insert_front()
{
    struct dnode <T>*t,*temp;
    cout<<"Enter data into  dnode:";
```

```

        cin>>data;
        temp=create_dnode(data);
        if(head==NULL)
            head=temp;
        else
        { temp->next=head; head-
            >prev=temp;
            head=temp;
        }
    }
}
template <class T>
void dlist<T>::insert_end()
{
    struct dnode<T> *t,*temp;
    int n;
        cout<<"Enter data into dnode:";
        cin>>n;
        temp=create_dnode(n);
        if(head==NULL)
            head=temp;
        else
        {
            t=head;
            while(t->next!=NULL)
                t=t->next;
            t->next=temp;
            temp->prev=t;
        }
}

```

```

template <class T>
void dlist<T>::Insert_at_pos(int pos)
{
    struct dnode<T>*cr,*pr,*temp;
    int count=1;
        cout<<"Enter data into dnode:";
        cin>>data;
        temp=create_dnode(data);
        display();
        if(head==NULL)
        { //when list is empty
            head=temp;
        }
        else
        {
            pr=cr=head;
            if(pos==1)
            {
                //inserting at pos=1
                temp->next=head;
                head=temp;
            }
            else

```

```

        {
            while(count<pos)
            {
                count++;
                pr=cr;
                cr=cr->next;
            }
            pr->next=temp;
            temp->prev=pr;
            temp->next=cr;
            cr->prev=temp;
        }
    }

template <class T>
void dlist<T>:: delete_front()
{struct dnode<T> *t;
    if(head==NULL)
        cout<<"List is Empty\n";
    else
    {
        display();
        t=head;
        head=head->next;
        head->prev=NULL;
        cout<<"dnode "<<t->data<<" Deletion is sucess";
        delete(t);
    }
}

template <class T>
void dlist<T>::delete_end()
{
    struct dnode<T> *pr,*cr;
    pr=cr=head;
    if(head==NULL)
        cout<<"List is Empty\n";
    else
    {
        cr=pr=head;
        if(head->next==NULL)
        {
            cout<<"dnode "<<cr->data<<" Deletion is sucess";
            delete(cr);
            head=NULL;
        }
        else
        {
            while(cr->next!=NULL)
            {
                pr=cr;
                cr=cr->next;
            }
            pr->next=NULL;
            cout<<"dnode "<<cr->data<<" Deletion is sucess";
            delete(cr);
        }
    }
}

```

```

        }
    }
}
template <class T>
void dlist<T>::Delete_at_pos(int pos)
{
    struct dnode<T>*cr,*pr,*temp;
    int count=1;
    display();
    if(head==NULL)
    {
        cout<<"List is Empty\n";
    }
    else
    {
        pr=cr=head;
        if(pos==1)
        {
            head=head->next;
            head->prev=NULL;
            cout<<cr->data <<"is deleted sucesfully";
            delete cr;
        }
        else
        {
            while(count<pos)
            {
                count++;
                pr=cr;
                cr=cr->next;
            }
            pr->next=cr->next;
            cr->next->prev=pr;
            cout<<cr->data <<"is deleted sucesfully";
            delete cr;
        }
    }
}

template <class T>
void dlist<T>::dnode_count()
{
    struct dnode<T>*t;
    int count=0;
    display();
    t=head;
    if(head==NULL)
        cout<<"List is Empty\n";
    else
    {
        while(t!=NULL)
        {
            count++;
            t=t->next;
        }
        cout<<"node count is "<<count;
    }
}

```

```

    }
}
template <class T>
void dlist<T>::display()
{
    struct dnode<T>*t;
    if(head==NULL)
    {
        cout<<"List is Empty\n";
    }
    else
    {
        cout<<"Nodes in the linked list are ...\n";
        t=head;
        while(t!=NULL)
        {
            cout<<t->data<<"<=>";
            t=t->next;
        }
    }
}
int main()
{
    int ch,pos;
    dlist <int> DL;
    while(1)
    {
        cout<<"\n ***Operations on Doubly List***"<<endl;
        cout<<"1.Insert dnode at End"<<endl;
        cout<<"2.Insert dnode at Front"<<endl;
        cout<<"3.Delete dnode at END"<<endl;
        cout<<"4.Delete dnode at Front"<<endl;
        cout<<"5.Display nodes "<<endl;
        cout<<"6.Count Nodes"<<endl;
        cout<<"7.Insert at a position "<<endl;
        cout<<"8.Delete at a position "<<endl;
        cout<<"9.Exit "<<endl;
        cout<<"10.Clear Screen "<<endl;
        cout<<"Enter Your choice:";
        cin>>ch;
        switch(ch)
        {
            case 1: DL.insert_end();
                    break;
            case 2: DL.insert_front();
                    break;
            case 3: DL.delete_end();
                    break;
            case 4: DL.delete_front();
                    break;
            case 5://display contents
                    DL.display();
                    break;

```

```

        case 6: DL.dnode_count();
                break;
        case 7: cout<<"Enter position to insert";
                cin>>pos;
                DL.Insert_at_pos(pos);
                break;
        case 8: cout<<"Enter position to Delete";
                cin>>pos;
                DL.Delete_at_pos(pos);
                break;
        case 9: exit(0);
        case 10: system("cls");
                break;
        default: cout<<"Invalid choice";

    }
}
}

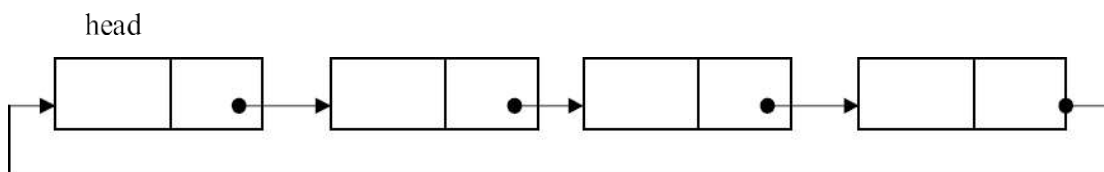
```

### CIRCULARLY LINKED LIST

A circularly linked list, or simply circular list, is a linked list in which the last node is always points to the first node. This type of list can be build just by replacing the NULL pointer at the end of the list with a pointer which points to the first node. There is no first or last node in the circular list.

#### Advantages:

- Any node can be traversed starting from any other node in the list.
- There is no need of NULL pointer to signal the end of the list and hence, all pointers contain valid addresses.
- In contrast to singly linked list, deletion operation in circular list is simplified as the search for the previous node of an element to be deleted can be started from that item itself.



### Dynamic Implementation of Circular linked list ADT

```

#include<iostream.h>
#include<stdlib.h>
template <class T>
struct cnode
{
    T data;
    struct cnode<T> *link;
};
//Code fot circular linked List ADT
template <class T>

```



```

class clist
{
    int data;
    struct cnode<T>*head;
public:
    clist();
    struct cnode<T>* create_cnode(int n);
    void display();
    void insert_end();
    void insert_front();
    void delete_end();
    void delete_front();
    void cnode_count();

};
//code for default constructor
template <class T>
clist<T>::clist()
{
    head=NULL;
}

//code to display elements in the list
template <class T>
void clist<T>::display()
{
    struct cnode<T>*t;
    if(head==NULL)
    {
        cout<<"clist is Empty\n";
    }
    else
    {
        t=head;
        if(t->link==head)
            cout<<t->data<<"->";
        else
        {
            cout<<t->data<<"->";
            t=t->link;
            while(t!=head)
            {
                cout<<t->data<<"->";
                t=t->link;
            }
        }
    }
}

//Code to create node
template <class T>
struct cnode<T>* clist<T>::create_cnode(int n)

```

```

{
struct cnode<T> *t;
    t=new struct cnode<T>;
    t->data=n;
    t->link=NULL;
return t;
}
//Code to insert node at the end
template <class T>
void clist<T>::insert_end()
{
struct cnode<T>*t;
struct cnode<T>*temp;
int n;
    cout<<"Enter data into cnode:";
    cin>>n;
    temp=create_cnode(n);
    if(head==NULL)
    {
        head=temp;
        temp->link=temp;
    }
    else
    {
        t=head;
        if(t->link==head)// list containing only one node
        {
            t->link=temp;
            temp->link=t;
        }
        else
        {
            while(t->link!=head)
            {
                t=t->link;
            }
            t->link=temp;
            temp->link=head;
        }
    }
    cout<<"Node inerted"<<endl;
}

//Code to insert node at front
template <class T>
void clist<T>::insert_front()
{
struct cnode <T>*t;
struct cnode<T>*temp;
    cout<<"Enter data into cnode:";
    cin>>data;

```

```

temp=create_cnode(data);
if(head==NULL)
{
    head=temp;
    temp->link=temp;
}
else
{
    t=head;
    if(t->link==head)
    {
        t->link=temp;
        temp->link=t;
    }
    else
    {
        //code to find last node
        while(t->link!=head)
        {
            t=t->link;
        }
        t->link=temp; //linking last and first node
        temp->link=head;
        head=temp;
    }
}
cout<<"Node inserted \n";
}

//Code to delete node at end
template <class T>
void clist<T>::delete_end()
{
    struct cnode<T>*cur,*prev;
    cur=prev=head;
    if(head==NULL)
        cout<<"clist is Empty\n";
    else
    {
        cur=prev=head;
        if(cur->link==head)
        {
            cout<<"cnode "<<cur->data<<" Deletion is sucess";
            free(cur);
            head=NULL;
        }
        else
        {
            while(cur->link!=head)
            {
                prev=cur;
                cur=cur->link;
            }

```

```

        //prev=cur;
        //cur=cur->link;
        prev->link=head;//points to head
        cout<<"cnode "<<cur->data<<" Deletion is sucess";
        free(cur);
    }
}
}
//Code to delete node at front
template <class T>
void clist<T>::delete_front()
{
    struct cnode<T> *t, *temp;
    if(head==NULL)
        cout<<"circular list is Empty\n";
    else
    {
        t=head;
        //head=head->link;
        if(t->link==head)
        {
            head=NULL;
            cout<<"cnode "<<t->data<<" Deletion is sucess";
            delete(t);
        }
        else
        {
            //code to find last node
            while(t->link!=head)
            {
                t=t->link;
            }
            temp=head;
            t->link=temp->link; //linking last and first node
            cout<<"cnode "<<temp->data<<" Deletion is sucess";
            head=temp->link;
            delete(temp);
        }
    }
}
}
//Code to count nodes in the circular linked list
template <class T>
void clist<T>::cnode_count()
{
    struct cnode<T> *t;
    int c=0;
    t=head;
    if(head==NULL)
    {
        cout<<"circular list is Empty\n";
    }
}

```

```
        else
        {
            t=t->link;
            c++;
            while(t!=head)
            {
                c++;
                t=t->link;
            }
            cout<<"Node Count="<<c;
        }
    }
}
int main()
{
    int ch,pos;
    clist <int> L;
    while(1)
    {
        cout<<"\n ***Operations on Circular Linked clist***"<<endl;
        cout<<"\n1.Insert cnode at End"<<endl;
        cout<<"2.Insert Cnode at Front"<<endl;
        cout<<"3.Delete Cnode at END"<<endl;
        cout<<"4.Delete Cnode at Front"<<endl;
        cout<<"5.Display Nodes "<<endl;
        cout<<"6.Cnode Count"<<endl;
        cout<<"7.Exit "<<endl;
        cout<<"8.Clear Screen "<<endl;
        cout<<"Enter Your choice:";
        cin>>ch;
        switch(ch)
        {
            case 1: L.insert_end();
                    break;
            case 2: L.insert_front();
                    break;
            case 3:L.delete_end();
                    break;
            case 4:L.delete_front();
                    break;
            case 5://display contents
                    L.display();
                    break;
            case 6: L.cnode_count();
                    break;
            case 7:exit(0);
            case 8:system("cls");
                    break;
            default:cout<<"Invalid choice";
        }
    }
}
```

**Stack:** Stack ADT, array and linked list implementation, Applications- expression conversion and evaluation. **Queue:** Types of Queue: Simple Queue, Circular Queue, Queue ADT- array and linked list implementation. Priority Queue, heaps.

**STACK ADT:-** A Stack is a linear data structure where insertion and deletion of items takes place at one end called top of the stack. A Stack is defined as a data structure which operates on a last-in first-out basis. So it is also referred as Last-inFirst-out( LIFO).

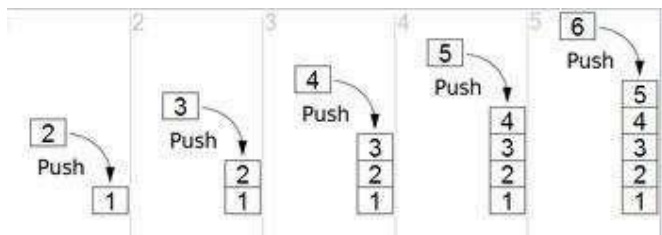
Stack uses a single index or pointer to keep track of the information in the stack. The basic operations associated with the stack are:

- a) push(insert) an item onto the stack.
- b) pop(remove) an item from the stack.

### The general terminology associated with the stack is as follows:

A stack pointer keeps track of the current position on the stack. When an element is placed on the stack, it is said to be **pushed** on the stack. When an object is removed from the stack, it is said to be **popped** off the stack. Two additional terms almost always used with stacks are **overflow**, which occurs when we try to push more information on a stack that it can hold, and **underflow**, which occurs when we try to pop an item off a stack which is empty.

### Pushing items onto the stack:

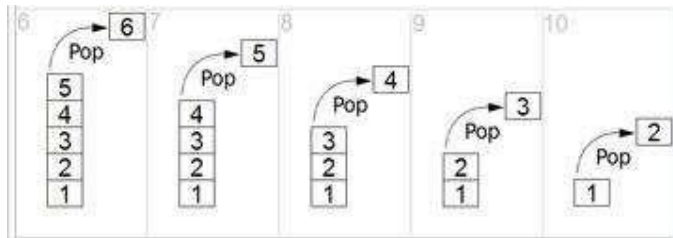


Assume that the array elements begin at 0 ( because the array subscript starts from 0) and the maximum elements that can be placed in stack is max. The stack pointer, **top**, is considered to be pointing to the top element of the stack. A push operation thus involves adjusting the stack pointer to point to next free slot and then copying data into that slot of the stack. Initially the top is initialized to -1.

```
//code to push an element on to stack;
template<class T>
void stack<T>::push()
{
    if(top==max-1)
        cout<<"Stack Overflow...\n";
    else
    {
        cout<<"Enter an element to be pushed:";
        top++;
        cin>>data;
        stk[top]=data;
        cout<<"Pushed Sucesfully....\n";
    }
}
```

### Popping an element from stack:

To remove an item, first extract the data from top position in the stack and then decrement the stack pointer, top.



```
//code to remove an element from stack
template<class T>
void stack<T>::pop()
{
    if(top == -1)
        cout<<"Stack is Underflow";
    else
    {
        data=stk[top];
        top--;
        cout<<data<<" is popped Successfully ....\n";
    }
}
```

### Static implementation of Stack ADT

```
#include<stdlib.h>
#include<iostream.h>
#define max 4
template<class T>
class stack
{
private:
    int top;
    T stk[max],data;
public:
    stack();
    void push();
    void pop();
    void display();
};
template<class T>
stack<T>::stack()
{
    top=-1;
```

```

}
//code to push an element on to stack;
template<class T>
void stack<T>::push()
{
    if(top==max-1)
        cout<<"Stack Overflow...\n";
    else
    {
        cout<<"Enter an element to be pushed:";
        top++;
        cin>>data;
        stk[top]=data;
        cout<<"Pushed Sucesfully ....\n";
    }
}

//code to remove an element from stack
template<class T>
void stack<T>::pop()
{
    if(top== -1)
        cout<<"Stack is Underflow";
    else
    {
        data=stk[top];
        top--;
        cout<<data<<" is poped Sucesfully ....\n";
    }
}

//code to display stack elements
template<class T>
void stack<T>::display()
{
    if(top== -1)
        cout<<"Stack Under Flow";
    else
    {
        cout<<"Elements in the Stack are....\n";
        for(int i=top; i>=0; i--)
        {
            cout<<<<stk[i]<<"\n";
        }
    }
}

int main()
{
    int choice;
    stack <int>st;
    while(1)
    {
        cout<<"\n*****Menu for Stack operations*****\n";
        cout<<"1.PUSH\n2.POP\n3.DISPLAY\n4.EXIT\n";
    }
}

```



```

        cout<<"Enter Choice:";
        cin>>choice;
        switch(choice)
        {
            case 1: st.push();
                    break;
            case 2: st.pop();
                    break;
            case 3: st.display();
                    break;
            case 4: exit(0);
            default:cout<<"Invalid choice...Try again...\n";
        }
    }
}

```

output:

\*\*\*\*\*Menu for Stack operations\*\*\*\*\*

1.PUSH  
2.POP  
3.DISPLAY  
4.EXIT

Enter Choice:1

Enter an element to be pushed:11

Pushed Sucesfully....

\*\*\*\*\*Menu for Stack operations\*\*\*\*\*

1.PUSH  
2.POP  
3.DISPLAY  
4.EXIT

Enter Choice:1

Enter an element to be pushed:22

Pushed Sucesfully....

\*\*\*\*\*Menu for Stack operations\*\*\*\*\*

1.PUSH  
2.POP  
3.DISPLAY  
4.EXIT

Enter Choice:1

Enter an element to be pushed:44

Pushed Sucesfully....

\*\*\*\*\*Menu for Stack operations\*\*\*\*\*

1.PUSH  
2.POP  
3.DISPLAY  
4.EXIT

Enter Choice:1

Enter Choice:1

Enter an item to be pushed:55

Pushed Sucesfully....

\*\*\*\*\*Menu for Stack operations\*\*\*\*\*

1.PUSH

2.POP

3.DISPLAY

4.EXIT

Enter Choice:1

Stack Overflow...

\*\*\*\*\*Menu for Stack operations\*\*\*\*\*

1.PUSH

2.POP

3.DISPLAY

4.EXIT

Enter Choice:2

55 is popped Successfully....

\*\*\*\*\*Menu for Stack operations\*\*\*\*\*

1.PUSH

2.POP

3.DISPLAY

4.EXIT

Enter Choice:3

Elements in the Stack are....

44

22

11

\*\*\*\*\*Menu for Stack operations\*\*\*\*\*

1.PUSH

2.POP

3.DISPLAY

4.EXIT

Enter Choice:4

### Dynamic implementation of Stack ADT

```
#include<iostream.h>
template <class T>
struct node
{
    T data;
    struct node<T> *link;
};
template <class T>
class stack
{
    int data;
    struct node<T>*top;
```

```

        public:
        stack()
        {
            top=NULL;
        }
        void display();
        void push();
        void pop();
};
template <class T>
void stack<T>::display()
{
    struct node<T>*t;

    if(top==NULL)
    {
        cout<<"stack is Empty\n";
    }
    else
    {
        t=top;
        while(t!=NULL)
        {
            cout<<"|"<<t->data<<"|"<<endl;
            t=t->link;
        }
    }
}

template <class T>
void stack<T>::push()
{
    struct node <T>*t,*temp;
    cout<<"Enter data into node:";
    cin>>data;
    temp=new struct node<T>;
    temp->data=data;
    temp->link=NULL;
    if(top==NULL)
        top=temp;
    else
    {
        temp->link=top;
        top=temp;
    }
}

template <class T>
void stack<T>::pop()
{
    struct node<T>*t;
    if(top==NULL)
        cout<<"stack is Empty\n";
}

```

```

        else
        {
            t=top;
            top=top->link;
            cout<<"node "<<t->data<<" Deletion is sucess";
            delete(t);
        }
    }

int main()
{
    int ch;
    stack <int> st;
    while(1)
    {
        cout<<"\n ***Operations on Dynamic stack***"<<endl;
        cout<<"\n1.PUSH"<<endl;
        cout<<"2.POP"<<endl;
        cout<<"3.Display "<<endl;
        cout<<"4.Exit "<<endl;
        cout<<"Enter Your choice:";
        cin>>ch;
        switch(ch)
        {
            case 1: st.push();
                    break;
            case 2: st.pop();
                    break;
            case 3:st.display();;
                    break;
            case 4:exit(0);
            default:cout<<"Invalid choice";
        }
    }
}

```

#### **Applications of Stack:**

1. Stacks are used in conversion of infix to postfix expression.
2. Stacks are also used in evaluation of postfix expression.
3. Stacks are used to implement recursive procedures.
4. Stacks are used in compilers.
5. Reverse String

An arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of an expression. These notations are–

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

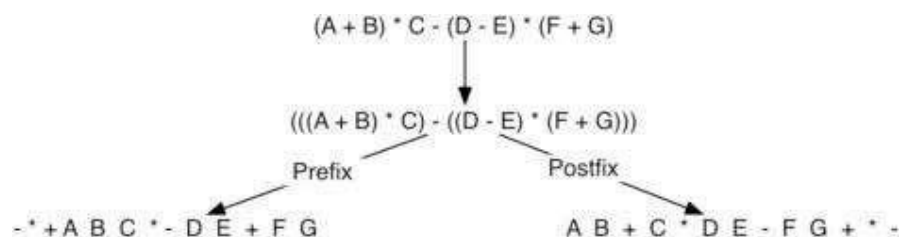
Expression	Example	Note
Infix	$a + b$	Operator Between Operands
Prefix	$+ a b$	Operator before Operands
Postfix	$a b +$	Operator after Operands

Conversion of Infix Expressions to Prefix and Postfix

Infix Expression	Prefix Expression	Postfix Expression
$A + B * C + D$	$++A * B C D$	$A B C * + D +$
$(A + B) * (C + D)$	$* + A B + C D$	$A B + C D + *$
$A * B + C * D$	$+ * A B * C D$	$A B * C D * +$
$A + B + C + D$	$+++A B C D$	$A B + C + D +$

Convert following infix expression to prefix and postfix

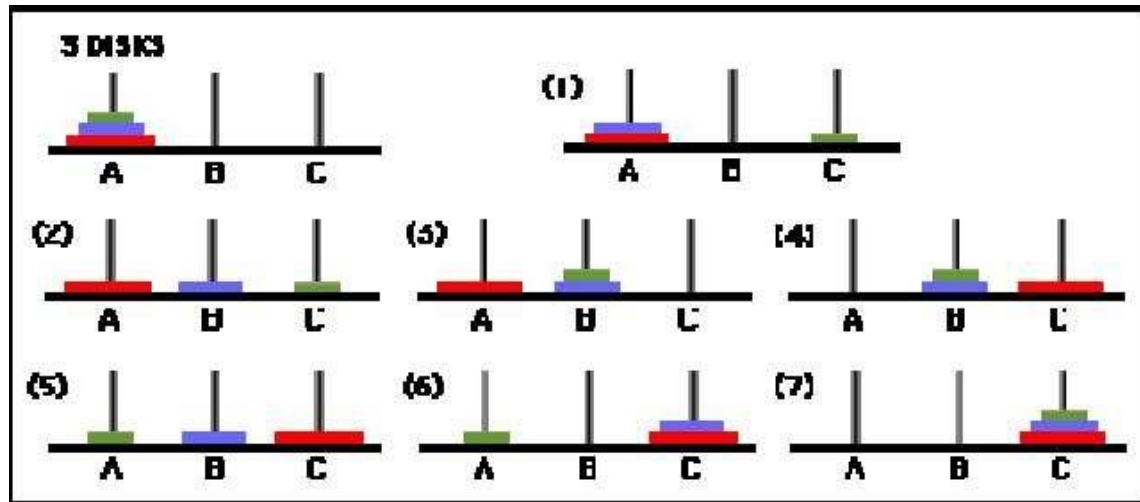
$(A + B) * C - (D - E) * (F + G)$



**The Tower of Hanoi** (also called the Tower of Brahma or Lucas' Tower,[1] and sometimes pluralized) is a mathematical game or puzzle. It consists of three rods, and a number of disks of different sizes which can slide onto any rod. The puzzle starts with the disks in a neat stack in ascending order of size on one rod, the smallest at the top, thus making a conical shape.

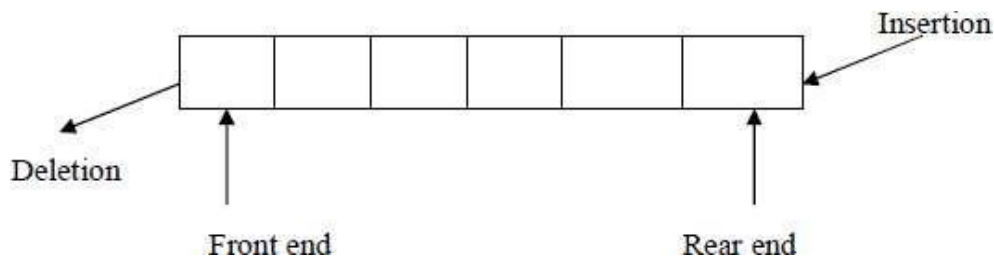
The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:

1. Only one disk can be moved at a time.
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
3. No disk may be placed on top of a smaller disk.



## QUEUE ADT

A queue is an ordered collection of data such that the data is inserted at one end and deleted from another end. The key difference when compared stacks is that in a queue the information stored is processed first-in first-out or FIFO. In other words the information receive from a queue comes in the same order that it was placed on the queue.



### Representing a Queue:

One of the most common way to implement a queue is using array. An easy way to do so is to define an array Queue, and two additional variables front and rear. The rules for manipulating these variables are

simple:

- Each time information is added to the queue, increment rear.
- Each time information is taken from the queue, increment front.
- Whenever **front > rear or front=rear=-1** the queue is empty.

Array implementation of a Queue do have drawbacks. The maximum queue size has to be set at compile time, rather than at run time. Space can be wasted, if we do not use the full capacity of the array.

### Operations on Queue:

A queue have two basic operations:

- a) adding new item to the queue
- b) removing items from queue.

The operation of adding new item on the queue occurs only at one end of the queue called the **rear** or back.

The operation of removing items of the queue occurs at the other end called the **front**.

For insertion and deletion of an element from a queue, the array elements begin at 0 and the maximum elements of the array is **maxSize**. The variable front will hold the index of the item that is considered the front of the queue, while the rear variable will hold the index of the last item in the queue.

Assume that initially the front and rear variables are initialized to -1. Like stacks, underflow and overflow conditions are to be checked before operations in a queue.

**Queue empty or underflow condition is**

```
if((front>rear)||front== -1)
    cout<<"Queue is empty";
```

**Queue Full or overflow condition is**

```
if((rear==max)
    cout<<"Queue is full";
```

### Static implementation of Queue ADT

```
#include<stdlib.h>
#include<iostream.h>
#define max 4
template <class T>
class queue
{
    T q[max],item;
    int front,rear;
public:
    queue();
    void insert_q();
    void delete_q();
    void display_q();
};
template <class T>
queue<T>::queue()
{
    front=rear=-1;
}
//code to insert an item into queue;
template <class T>
void queue<T> ::insert_q()
```

```

{
    if(front>rear)
        front=rear=-1;
    if(rear==max-1)
        cout<<"queue Overflow...\n";
    else
    {
        if(front==0)
            front=0;

        rear++;
        cout<<"Enter an item to be inserted:";
        cin>>item;
        q[rear]=item;
        cout<<"inserted Sucesfully..into queue..\n";
    }
}
template <class T>
void queue<T>::delete_q()
{
    if((front==0 && rear==0)||front>rear)
    {
        front=rear=-1;
        cout<<"queue is Empty ..\n";
    }
    else
    {
        item=q[front];
        front++;
        cout<<item<<" is deleted Sucesfully ...\n";
    }
}
template <class T>
void queue<T>::display_q()
{
    if((front==0 && rear==0)||front>rear)
    {
        front=rear=-1;
        cout<<"queue is Empty ..\n";
    }
    else
    {
        for(int i=front;i<=rear;i++)
            cout<<"| "<<q[i]<<"|<--";
    }
}
}
int main()
{
    int choice;
    queue<int> q;
    while(1)
    {
        cout<<"\n\n*****Menu for operations on QUEUE*****\n\n";
        cout<<"1.INSERT\n2.DELETE\n3.DISPLAY\n4.EXIT\n";
    }
}

```



```

        cout<<"Enter Choice:";
        cin>>choice;
        switch(choice)
        {
            case 1: q.insert_q();
                    break;
            case 2: q.delete_q();
                    break;
            case 3: cout<<"Elements in the queue are... \n";
                    q.display_q();
                    break;
            case 4: exit(0);
            default: cout<<"Invalid choice...Try again...\n";
        }
    }
}

```

### Dynamic implementation of Queue ADT

```

#include<stdlib.h>
#include<iostream.h>
template <class T>
struct node
{
    T data;
    struct node<T>*next;
};
template <class T>
class queue
{
private:
    T item;
    node<T> *front,*rear;
public:
    queue();
    void insert_q();
    void delete_q();
    void display_q();
};
template <class T>
queue<T>::queue()
{
    front=rear=NULL;
}
//code to insert an item into queue;
template <class T>
void queue<T>::insert_q()
{
    node<T>*p;
    cout<<"Enter an element to be inserted:";
}

```

```

        cin>>item;
        p=new node<T>;
        p->data=item;
        p->next=NULL;
        if(front==NULL)
        {
            rear=front=p;
        }
        else
        {
            rear->next=p;
            rear=p;
        }
        cout<<"\nInserted into Queue Sucesfully... \n";
    }
    //code to delete an elementfrom queue
    template <class T>
    void queue<T>::delete_q()
    {
        node<T>*t;
        if(front==NULL)
            cout<<"\nQueue is Underflow";
        else
        {
            item=front->data;
            t=front;
            front=front->next;
            cout<<"\n"<<item<<" is deleted from Queue... \n";
        }
        delete(t);
    }
    //code to display elements in queue
    template <class T>
    void queue<T>::display_q()
    {
        node<T>*t;
        if(front==NULL)
            cout<<"\nQueue Under Flow";
        else
        {
            cout<<"\nElements in theQueue are... \n";
            t=front;
            while(t!=NULL)
            {
                cout<<"|"<<t->data<<"|<-";
                t=t->next;
            }
        }
    }
}

int main()
{
    int choice;
    queue<int>q1;

```

```

while(1)
{
cout<<"\n\n***Menu for operations on Queue***\n\n";
cout<<"1.Insert\n2.Delete\n3.DISPLAY\n4.EXIT\n";
cout<<"Enter Choice:";
cin>>choice;
switch(choice)
{
case 1: q1.insert_q();
break;
case 2: q1.delete_q();
break;
case 3: q1.display_q();
break;
case 4: exit(0);
default: cout<<"Invalid choice...Try again...\n";
}
}
}

```

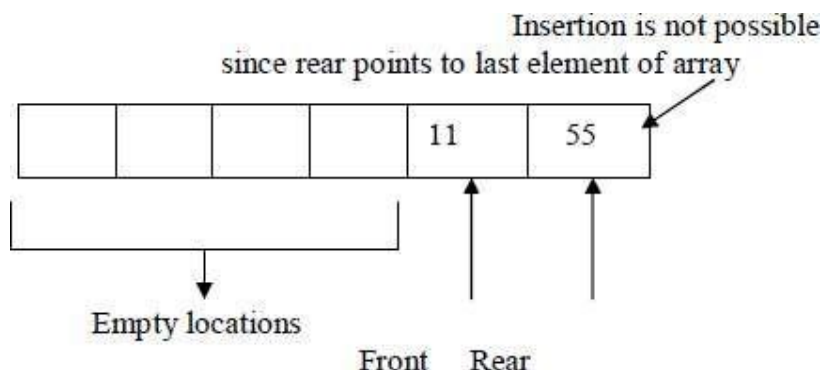
### Application of Queue:

Queue, as the name suggests is used whenever we need to have any group of objects in an order in which the first one coming in, also gets out first while the others wait for there turn, like in the following scenarios :

1. Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
2. In real life, Call Center phone systems will use Queues, to hold people calling them in an order, until a service representative is free.
3. Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive, First come first served.

### CIRCULAR QUEUE

Once the queue gets filled up, no more elements can be added to it even if any element is removed from it consequently. This is because during deletion, rear pointer is not adjusted.



When the queue contains very few items and the rear pointer points to last element. i.e.  $\text{rear} = \text{maxSize} - 1$ , we cannot insert any more items into queue because the overflow condition satisfies. That means a lot of space is wasted

.Frequent reshuffling of elements is time consuming. One solution to this is arranging all elements in a circular fashion. Such structures are often referred to as **Circular Queues**.

A circular queue is a queue in which all locations are treated as circular such that the first location CQ[0] follows the last location CQ[max-1].

**Circular Queue empty or underflow condition is**

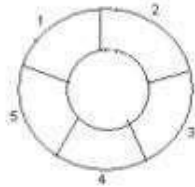
```
if(front==-1)
    cout<<"Queue is empty";
```

**Circular Queue Full or overflow condition is**

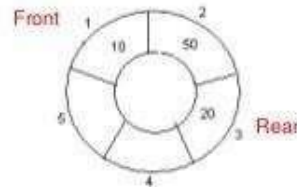
```
if(front==(rear+1)%max)
{
    cout<<"Circular Queue is full\n";
}
```

**Example: Consider the following circular queue with N = 5.**

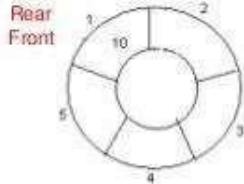
1. Initially, Rear = 0, Front = 0.



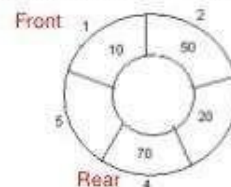
4. Insert 20, Rear = 3, Front = 0.



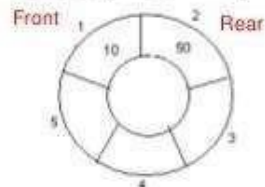
2. Insert 10, Rear = 1, Front = 1.



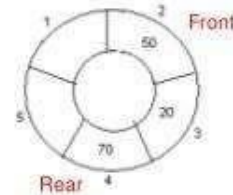
5. Insert 70, Rear = 4, Front = 1.



3. Insert 50, Rear = 2, Front = 1.



6. Delete front, Rear = 4, Front = 2.



### Insertion into a Circular Queue:

Algorithm CQueueInsertion(Q,maxSize,Front,Rear,item)

Step 1: If Rear = maxSize-1 then

Rear = 0

else

Rear=Rear+1

Step 2: If Front = Rear then

print "Queue Overflow"

Return

Step 3: Q[Rear] = item

Step 4: If Front = 0 then

Front = 1

Step 5: Return

### Deletion from Circular Queue:

Algorithm CQueueDeletion(Q,maxSize,Front,Rear,item)

Step 1: If Front = 0 then

print "Queue Underflow"

Return

Step 2: K=Q[Front]

Step 3: If Front = Rear then

begin

Front = -1

Rear = -1

end

else

If Front = maxSize-1 then

Front = 0

else

Front = Front + 1

Step 4: Return K

### Static implementation of Circular Queue ADT

```
#include<iostream.h>
#define max 4
template <class T>
class CircularQ
{
    T cq[max];
    int front,rear;
public:
    CircularQ();
    void insertQ();
    void deleteQ();
    void displayQ();
};
template <class T>
CircularQ<T>::CircularQ()
{
    front=rear=-1;
}
template <class T>
void CircularQ<T>:: insertQ()
{
    int num;
    if(front==(rear+1)%max)
    {
        cout<<"Circular Queue is full\n";
    }
}
```

```

        else
        {
            cout<<"Enter an element";
            cin>>num;
            if(front==-1)
                rear=front=0;
            else
                rear=(rear+1)%max;
            cq[rear]=num;
            cout<<num <<" is inserted ...";
        }
    }
    template <class T>
    void CircularQ<T>::deleteQ()
    {
        int num;
        if(front==-1)
            cout<<"Queue is empty";
        else
        {
            num=cq[front];
            cout<<"Deleted item is "<< num;
            if(front==rear)
                front=rear=-1;
            else
                front=(front+1)%max;
        }
    }
    template <class T>
    void CircularQ<T>::displayQ()
    {
        int i;
        if(front==-1)
            cout<<"Queue is empty";
        else
        {
            cout<<"Queue elements are\n";
            for(i=front;i<=rear;i++)
                cout<<cq[i]<<"\t";
        }
        if(front>rear)
        {
            for(i=front;i<max;i++)
                cout<<cq[i]<<"\t";
            for(i=0;i<=rear;i++)
                cout<<cq[i]<<"\t";
        }
    }
    int main()
    {
        CircularQ<int> obj;
        int choice;
        while(1)
        {
            cout<<"\n*** Circular Queue Operations***\n";

```

```

        cout<<"\n1.insert Element into CircularQ";
        cout<<"\n2.Delete Element from CircularQ";
        cout<<"\n3.Display Elements in CircularQ";
        cout<<"\n4.Exit ";
        cout<<"\nEnter Choice:";
        cin>>choice;
        switch(choice)
        {
            case 1: obj.insertQ();
                    break;
            case 2: obj.deleteQ();
                    break;
            case 3: obj.displayQ();
                    break;
            case 4: exit(0);
        }
    }
}

```

## Priority Queue

### DEFINITION:

A priority queue is a collection of zero or more elements. Each element has a priority or value.

Unlike the queues, which are FIFO structures, the order of deleting from a priority queue is determined by the element priority.

Elements are removed/deleted either in increasing or decreasing order of priority rather than in the order in which they arrived in the queue.

There are two types of priority queues:

- └ Min priority queue
- └ Max priority queue

**Min priority queue:** Collection of elements in which the items can be inserted arbitrarily, but only smallest element can be removed.

**Max priority queue:** Collection of elements in which insertion of items can be in any order but only largest element can be removed.

In priority queue, the elements are arranged in any order and out of which only the smallest or largest element allowed to delete each time.

The implementation of priority queue can be done using arrays or linked list. The data structure **heap** is used to implement the priority queue effectively.

### APPLICATIONS:

1. The typical example of priority queue is scheduling the jobs in operating system. Typically OS allocates priority to jobs. The jobs are placed in the queue and position of the job in priority queue determines their priority. In OS there are 3 jobs- real time jobs, foreground jobs and background jobs. The OS always schedules the real time jobs first. If there is no real time jobs pending then it schedules foreground jobs. Lastly if no real time and foreground jobs are pending then OS schedules the background jobs.
2. In network communication, the manage limited bandwidth for transmission the priority queue is used.
3. In simulation modeling to manage the discrete events the priority queue is used.

Various operations that can be performed on priority queue are-

1. Find an element
2. Insert a new element
3. Remove or delete an element

The abstract data type specification for a max priority queue is given below. The specification for a min priority queue is the same as ordinary queue except while deletion, find and remove the element with minimum priority

### ABSTRACT DATA TYPE(ADT):

Abstract data type maxPriorityQueue

{

Instances

Finite collection of elements, each has a priority Operations

empty():return true iff the queue is empty

size() :return number of elements in the queue

top() :return element with maximum priority

del() :remove the element with largest priority from the queue

insert(x): insert the element x into the queue



}