UNIT – II
CHAPTER – I – RECURSION

1.INTRODUCTION

Functions are the most useful feature of any programming language. A function may call itself or other function and the called function may again call the calling function. Such functions are called "Recursive Functions".

When a function calls itself, either directly or indirectly, it is known as a recursive call. A program becomes compact and readable with recursive functions.

Recursion is extremely powerful, it enables the programmer to express complex processes easily.

The recurrence relation is easily converted to recursive code. Recursion can be used for divide and conquer based search and sort algorithms to increase the efficiency of these operations.

For most problems such as the Towers of Hanoi, recursion presents an incredibly elegant solution that is easy to code and simple to understand.

2.RECURRENCE

A recurrence is a well defined mathematical function, where the function being defined is applied within its own definition.

The factorial, n!=n*(n-1)! is an example of recurrence with 0!=1 as the end condition. The problems that can be described using recurrence are easily expressed as recursive functions in programming.

```
long int factorial(unsigned int n)
{   if(n==0)
      return 1;
  else
     return(n*factorial(n-1));   }
```

The process of recursion occurs when a function calls itself. Recursion is useful in situations where solving one or more smaller versions of the same problem can solve the problem.

3^4 is calculated as $3^4=3*3^3$

3^3 is calculates as $3^3 = 3*3^2$

3^2 is calculated as $3^2 = 3*3^1$

3^1 is calculated as $3^1 = 3*3^0$ $(3^0 = 1) = 3*1=3$

The recurrence of this computation is $\mathbf{X^Y = X*X^{Y-1}}$, in each of these cases, the problem is reduced to a smaller version of itself.

Ex:
```
#include<iostream.h>
long int  Power(int x, int y)
{   if(y==0)
      return  1;
   else
     return(x*Power(x, y-1));
}
void  main()
{  int   x,y;
  cin>>x>>y;
  cout<<x<<" power "<<y<<" = "<<Power(x,y);      }
```

3.USE OF STACK IN RECURSION

The stack is a special area of memory where temporary variables are stored. It acts on the LIFO principle. For example, use factorial recursive function,'

```
long int factorial(unsigned int n)
{   if(n==0)
      return 1;
  else
      return(n*factorial(n-1));
 }
```

Let n=3, to compute 3!, at first time n holds 3, so the else statement is executed, it again calls factorial function with n-1, that is 2, so it pushes n value 3, on to the stack and calls itself.

For second time n holds 2, so else is executed, and n value 2 is pushed on to the stack and calls itself.

For third time n holds 1, so else is executed and n value 1 is pushed on to the stack and calls itself.

For fourth time n holds 0, so if statement is executed it returns 1 to the last call, that returns 1 to the third call that returns 2 to the second call, that returns 6 to the first call, i.e., it takes n values from stack as LIFO manner.

The factorial function runs 4 times foe n=3, out of which it calls itself 3 times, teh numbr of times a function calls itself is known as the "Recursive Depth of that function".

Each time the function calls itself, it stores variable value on the stack, stack is a small memory, the function with high recursive depth may produce "stack overflow error".

Recursive functions usually have a terminating condition. A recursion without end condition is called "endless recursion".

All recursive functions go through two distinct phases,

1. Winding: It occurs when the function calls itself, and pushes values on the stack.
2. UnWinding: It occurs when the function pops values from the stack, after end condition.

4.VARIANTS OF RECURSION

Recursion may have may have any one of the following forms:

a) A function calls itself
b) A function calls another function which in turn calls the caller function
c) The function call is part of the same processing instruction that makes recursive function call.

The basic two forms of recursion are:

1) Binary recursion: A binary recursive function calls itself twice. Fibonacci numbers computation, quick sort, and merge sort are examples of binary recursion.

Ex:

```
int Fib(int n)
{
  if(n==1 || n==2)
        return 1;
  else
        return(Fib(n-1)+Fib(n-2));
}
```

2.N-ary Recursion: The most general form of recursion is N-ary recursion, where n is a variable passed as parameter to a function. This type of functions are useful in Permutations.

Depending on the characterization, the recursive functions are classified as

a) Direct Recursion
b) Indirect Recursion
c) Tail Recursion
d) Linear Recursion
e) Tree Recursion

a) Direct Recursion: Recursion is said to be direct when a function calls itself directly.

Ex:
```
long int factorial(unsigned int n)
{   if(n==0)
      return 1;
   else
      return(n*factorial(n-1));
   }
```

b) Indirect Recursion: Recursion is said to be indirect when a function calls another function which in turn calls it.

Ex:
```
long int factorial(unsigned int n)
{   if(n==0)
      return 1;
   else
      return(n*Dummy(n-1));
    }
void  Dummy(unsigned int n)
{
   factorial(n);
}
```

c)Tail Recursion: A recursive function is said to be tail recursive if there are no pending operations to be performed on return from a recursive call.

This recursion is also used in return value of the last recursive call as the value of the function. Binary search is an example of tail recursion.

Ex:
```
int  Binary_Search(int a[ ], in tlow, int high, int key)
{   int  mid;
   if(low<=high)
   {
      mid=(low+high)/2;
      if(a[mid]==key)
           return  mid;
      else
           if(key<a[mid])
               return  Binary_Search(a, low, mid-1, key);
           else
              return Binary_Search(a, mid+1, high, key);
      }
     return  -1;
}
```

d) Linear Recursion: A recursive function is said to be linearly recursive when no pending operation involves another recursive call. Factorial is an example of linear recursion.

This is the simplest form of recursion and occurs when an action has a simple repetitive structure.

Ex:
```
long int factorial(unsigned int n)
{  if(n==0)
     return 1;
   else
     return(n*factorial(n-1));
 }
```

e)Tree recursion: In a Recursive function, if there is another recursive call in the set of operations to be completed after the recursion is over, this is called a Tree recursion.

Quick sort, Merge sort and Fibonacci series are of tree recursion.

Ex:
```
int  Fib(int  n)
{
  if(n==1 || n==2)
        return 1;
  else
        return(Fib(n-1)+Fib(n-2));
}
```

5.RECURSIVE FUNCTIONS

A program with recursive function results in a compact and readable code. Recursive functions are simple and elegant and can be easily verified by the user.

Recursive functions are related to inductive definitions of functions in mathematics. The natural recursive algorithms like factorial, power, Fibonacci can be implemented as either iterative or recursive code.

Recursive functions are smaller and more efficient than their iterative equivalents.

Ex: consider a given set of cardinality n>=1, to print all permutations of set {1,2,3}, they are {1,2,3}, {1,3,2}, {2,1,3},{2,3,1},{3,1,2}, and {3,2,1}.

The total number of possible permutations of a set of cardinality n is n!. Let S={a,b,c,d}. Generate each permutation by printing the following:

1. a followed by the permutations of set {b,c,d}
2. b followed by the permutations of set {a,c,d}
3. c followed by the permutations of set {a,b,d}
4. d followed by the permutations of set {a,b,c}

Here, the phrase 'followed by' introduces recursion. Recursion is also useful in data structures like linked lists and trees.

In 'divide and conquer' and 'back tracking' algorithms, this recursion is more valuable. The divide and conquer technique, n inputs are splitting into 'k' subsets, 1<=k<=n, yielding k sub problems, then these sub problems must be solved, and combined t get final solution.

Recursion allows us to breakdown a problem into one or more sub problems that are similar to the original problem.

Ex: Binary Search, Merge Sort, Quick Sort

The general approach to write a recursive function is

1. Write the function header, it shows functions work.

2. Decompose the problem into sub problems, identify non recursive case of the problem, test it, also known as base case or end condition.
3. Write recursive calls to solve those sub problems.
4. Write the code to combine, enhance or modify the results of the recursive calls and construct the desired return value.
5. Write the end conditions to handle any situation that are not properly handled by the recursive portion of the program.

Towers of Hanoi: an example of recursion

The use of recursion makes everything simpler. First, find out the recurring data, and essential feature of the problem that should change as the function calls itself.

Towers of Hanoi includes moving disks from tower A to tower C in the same order.

```
#include<iostream.h>
#include<conio.h>
int  count;
void  towers(int  n,  char a, char b, char c)
{   if(n==1)
   { count++;
     cout<<count<<": Move Disk 1 From Tower "<<a<<" To Tower "<<c<<endl;
    }
  else
   { towers(n-1, a,c,b);
     count++;
     cout<<count<<" : Move Disk "<<n<<" From Tower "<<a<<" To Tower "<<c<<endl;
     towers(n-1, b,a,c);
   }
  return;
}
void  main()
{ int  n;
  clrscr( );
  cout<<"How many disks\n";
  cin>>n;
  towers(n, 'A', 'B', 'C');
}
```

The five conditions to check the correctness of recursion are

1. A recursive function must have at least one end condition and one recursive case
2. First test end condition before recursive call
3. The problem must be broken into smaller problems.
4. The recursive call must not spoken the base case
5. Verify that the non-recursive code of the function is operating correctly.

Important points about recursion are

1. Recursive functions call themselves within their own definition.
2. Recursive functions must have a non-recursive terminating condition, otherwise an infinite loop will occur.
3. Recursion is easy to code, but memory starving.

6.ITERATION VERSUS RECURSION

RECURSION	ITERATION
1.Recursion is a Top-Down approach of Problem solving	1.Iteration is a Bottom-Up approach of problem solving
2. Recursive code is readable and easy to understand	2.Iterative code is not readable, and hence not easy to understand
3.Simple logic is required	3.Needs complex logic
4.Recursion produce smaller & simple code	4.Iterations produce lengthy code
5.Recursive approach involves only base condition or terminate condition	5.Iterative approach involves initialization, condition, execution and updation
6.In recursion, function calls itself until the base condition is reached.	6.Iteration means repetition of process until a specified condition fails.
7.Recursion is slower than Iteration	7.Iteration is faster than Recursion
8.It takes more memory, due to stack	8. It uses less memory than recursion
9.If recursion is not terminated, then it creates stack overflow	9. Iteration automatically terminates when it fails the given condition
10.We can't solve all problems using recursion.	10.Any recursive problem can be solved Iteratively

CHAPTER – II QUEUES

1.CONCEPT OF QUEUES

A queue is a linear or an ordered list, where data can be inserted and deleted from different ends.

The insertion end is called the 'rear' and the deletion end is called the 'front'. In a queue, the data is processed in the sequence in which they are entered, i.e., a queue is a First In First Out (FIFO) or Last In Last Out (LILO) structure.

Consider a Queue L={a_1, a_2, a_3, ... , a_n}, a1 is the front end element and a_n is the rear end element, i.e., a_i is behind a_{i-1}.

Ex: Q={Swetha, Shreya, Chandana, Deepika, Ishitha}

In Q, Swetha is at front end, Ishitha is at rear end.

Queues are of most common data processing structures, used in Operating Systems, Network and database implementations.

Queues are very useful in timesharing and distributed computer systems, whenever user places a request, the OS adds it at the end of job queue, the CPU executes the job at the front of the queue.

2.QUEUE ADT

To realise a queue as an ADT, we need a suitable data structure for storing elements in the queue and the functions operating on it. The basic operations of Queue are

i) create( )	ii) adding an element	iii) deleting an element
iv) getfront element of the queue	v) queue full	vi)queue empty

1) create( ) → creates an empty Queue, Q
2) add(ele) → adds ele at the rear end of the queue, it returns new Queue
3) del() → deletes an element from the front end of the Q, it returns new Queue.
4) getFront()→ returns the front most element of the Queue, Q.

5) isFull( ) → returns True, if Queue is Full otherwise it returns False.
6) isEmpty( )→ returns True, if Queue is Empty otherwise it returns False.

Queue ADT:

1.declare create( ) → Queue
2. add(element) → Queue
3. del( ) → Queue
4.getFront( ) → Element
5.isEmpty( ) → Boolean, True(1) / False(0)
6.isFull( ) → Boolean, True(1) / False(0)
7.For all Q∑Queue, ele∑element let
8. isEmpty(create()) → True
9. isEmpty(add(ele)) → False
10. del(create( )) → Error, Queue Underflow
11.getFront(add(ele)) → if Q is not Full, then adds the ele to the Q, returns front ele of the Q.
12. del(add(ele)) → if Q is not Full, then add ele to the Q, delete the front ele of Q.
13. getFront(create()) → underflow error
14. end

3.REALIZATION OF QUEUES USING ARRAYS

An array is not suitable data structure for frequent insertion and deletion of data elements, due to static memory allocation and they can store only a fixed number of elements.

A queue is a linear or an ordered list, where data can be inserted and deleted from different ends. The insertion end is called the 'rear' and the deletion end is called the 'front'. In a queue, the data is processed in the sequence in which they are entered, i.e., a queue is a First In First Out (FIFO) structure. Different Queue operations are

i) create( ) ii) adding an element iii) deleting an element
iv) getfront element of the queue v) queue full vi)queue empty

i)Create: This operation creates an empty queue, first define MaxSize of the Queue, and tehn declare queue and use a constructor for creating a queue.

```
#define     MaxSize     5
class  Queue
{
  private:     int  que[MaxSize];
               int  front,  rear;
  public:      Queue( )
               {   front=rear=-1;  }
               int isEmpty( );
               int  isFull( );
               void add(int ele);
               void del( );
               void getFront( );
};
```

This creates an empty queue of MaxSize, front and rear are initialized to -1 to specify empty queue.

ii)isEmpty( ): This operation checks whether the queue is empty or not if front=rear then this operation returns true otherwise returns false.

```
int  Queue::isEmpty( )
{ if(front==rear)
        return 1;
  else
        return 0;
}
```

iii)isFull( ): This operation checks, whether the queue is full or not. If rear==MaxSize-1 then, queue is full otherwise queue is not full, it can accommodate new element.

```
int  Queue::isFull( )
{ if(rear == MaxSize-1)
        return 1;
  else
        return 0;
}
```

iv)add(int ele): This operation adds an element to the queue if it is not full, from rear end. Rear end points to the last element of the queue, then new element is added at (rear+1)th position.

```
void  Queue::add(int ele)
{  if(!isFull( ))
        que[++rear]=ele;
   else
        cout<<"Queue OverFlow";
}
```

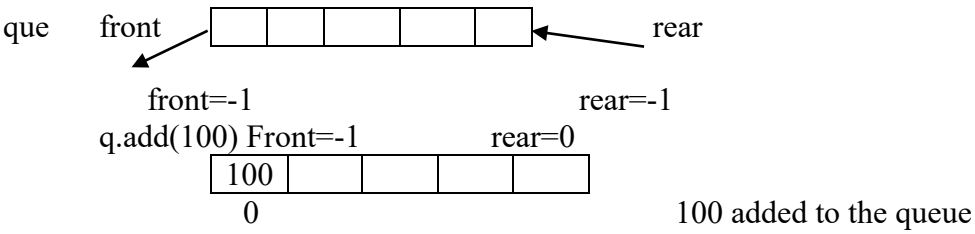
v) del( ): This operation deletes an element from the front end of the queue, if the queue is not empty and sets the front to point to the next element. Front can be initialized to one position less than the actual front.

```
void  Queue::del( )
{  if(!isEmpty( ))
        cout<<que[++front];
   else
        cout<<"Queue UnderFlow";  }
```

vi)getFront( ): This operation returns the element at the front of the queue, if the queue is not empty.

```
void  Queue::getFront( )
{  if(!isEmpty( ))
        cout<<que[front+1];
   else
        cout<<"Queue UnderFlow";
}
```

Let MaxSize=5, then que is empty at the time of creation, and front=rear=-1.



q.add(50) Front=-1 rear=1

100	50			
0	1			

50 added to the queue

q.add(10) Front=-1 rear=2

100	50	10		
0	1	2		

10 added to the queue

q.add(80) Front=-1 rear=3

100	50	10	80	
0	1	2	3	

80 added to the queue

q.add(500) Front=-1 rear=4

100	50	10	80	500
0	1	2	3	4

500 added to the queue

q.add(60) Front=-1 rear=4

100	50	10	80	500
0	1	2	3	4

not possible to add 60, queue is full, Overflow

q.del( ) rear=4

	50	10	80	500
Front= 0	1	2	3	4

100 deleted

q.del( ) rear=4

		10	80	500
Front= 1		2	3	4

50 deleted

q.getFront( ) rear=4

		10	80	500
Front= 1		2	3	4

it returns 10

q.del( ) rear=4

			80	500
	Front= 2		3	4

10 deleted

q.del( ) rear=4

				500
		Front= 3		4

80 deleted

q.del( ) rear=4

			Front= 4	

500 deleted

q.del( ) rear=4

			Front= 4	

queue is empty, not possible to delete, Underflow

4.CIRCULAR QUEUE

To avoid certain drawbacks of linear queue, circular queue is designed. Different drawbacks of queue are:

1. The linear queue is of a fixed size, so the user doesn't have the flexibility to dynamically change the size of the queue.
2. This declared size of queues leads to poor utilization of memory.
3. Array implementation of queue leads to queue full state, even though the queue is not actually full.

4. To avoid this, when queue full occurs, rewind the entire queue to the original start location, so that the first element is at 0^{th} location and front is set to -1.

The technique allows that the queue to wraparound upon reaching the end of the array eliminates these drawbacks.

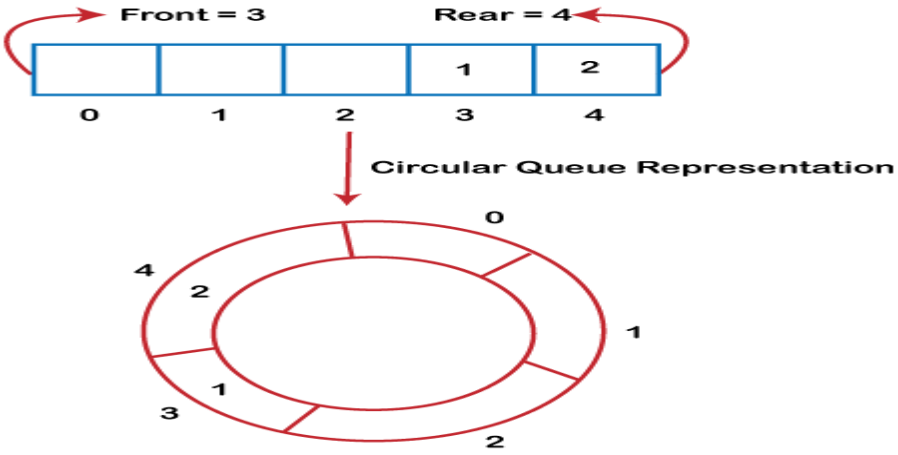
Such technique allows the queues to wraparound from end to start is called a "Circular Queue". More efficient queue representation is obtained by implementing the queue array as Circular Queue.

Here, as we go on adding elements to the queue and reach the end of the queue, the next element is stored in the first location of the array if it is empty.

Suppose, queue size is n, if we go on adding elements to the queue, if n-1 location is also used, then check at front for empty, if there is empty, add the element at that location.

The empty slots will be filled with new incoming elements even though rear=n-1. The circular queue allows us to continue adding elements even though we have reached the end of the array.

The queue is said to be full only when it stores n elements.



In circular queue, when rear is n-1, a new element is added at 0, if it is empty, i.e., rear should be set to 0.

Initially, both front and rear are set to -1. Different operations on circular queue are

```
#define MaxSize  5
class CQueue
{  private:      int  que[MaxSize];
                int  front,  rear;
   public:      CQueue( )
                 {
                    front=rear=-1;  }
                int isEmpty( );
                int  isFull( );
                void add(int ele);
                void del( );
                void getFront( );
};
```

The constructor CQueue( ) creates an empty circular queue.

i)isEmpty( ): It checks whether the circular queue is empty or not, it returns true of false, if rear==-1 then the cqueue is said to be empty.

```
int  CQueue::isEmpty( )
{ if(rear==-1)
        return 1;
  else
        return 0;
}
```

ii)isFull( ): This function checks whether the Cqueue is full or not, returns true or false. When front==-1 and rear==MaxSize-1 then the cqueue is full, overflow occurs.

```
int CQueue::isFull( )
{ if(rear==MaxSize-1 && front==-1)
        return  1;
   else
        return 0;
}
```

iii) add(int ele): This function adds an element to the CQueue, if it is not full. Find the rear value by using **"(rear+1)%MaxSize"**, and then add the element at that rear, otherwise display CQueue overflow.

```
void  CQueue::add(int ele)
{if(!isFull( ))
   { rear=(rear+1)%MaxSize;
     que[rear]=ele;   }
  else
    cout<<"CQueue Overflow";
}
```

iv) del( ): This function deletes an element from the front of the CQueue if it is not empty, find front value by using **"(front+1)%MaxSize"**, and then delete the element at that front otherwise display CQueue Underflow.

```
void  CQueue::del( )
{if(!isEmpty( ))
   { front=(front+1)%MaxSize;
     que[front]=ele;
     if(front==rear)
          front=rear=-1;   }
  else
    cout<<"CQueue Underflow";
}
```

v)getFront( ): This operation returns the front element of the CQueue, if it is not empty.

```
void  CQueue::getFront( )
{if(!isEmpty( ))
   { int t=(front+1)%MaxSize;
     cout<<que[t];   }
  else
    cout<<"CQueue Underflow";
}
```

Advantages of circular queues:

- By using circular queues, data shifting is avoided as the front and rear are modified by using modulus function, this wraps the queue back to its beginning.
- If the number of elements to be stored in the queue is fixed, the circular queue is advantageous.
- Printer queue, Priority queue and simulations use the circular queue.

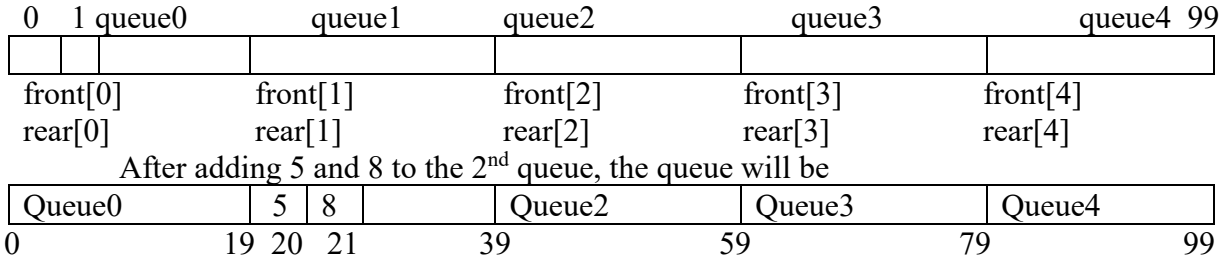
5.MULTI QUEUES

If more number of queues is required to be implemented, then an efficient data structure is required to handle multiple queues. It is possible to utilize all the available space in a single array.

When more than two queues, say n, are represented sequentially, we can divide the available memory into n segments and allocate these segments to n queues, one each.

For each queue i, use front[i], rear[i], the condition front[i]==rear[i] specifies i^{th} queue is empty, and the condition rear[i]==front[i] specifies i^{th} queue is full.

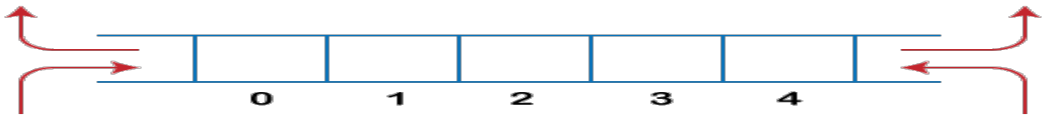
If we want 5 queues, then we can divide the array a[100] into equal parts of 20 and initialize front and rear for each queue, front[0]=rear[0]=-1, and front[1]=rear[1]=19 and so on for the other queues.



After adding 5 and 8 to the 2nd queue, the queue will be

6.DEQUE

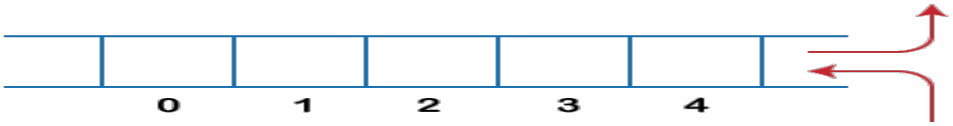
The dequeue stands for **Double Ended Queue**. In the queue, the insertion takes place from one end while the deletion takes place from another end. The end at which the insertion occurs is known as the **rear end** whereas the end at which the deletion occurs is known as **front end**.



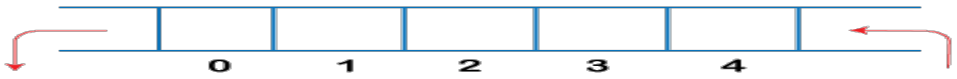
Deque is a linear data structure in which the insertion and deletion operations are performed from both ends. We can say that deque is a generalized version of the queue.

Deque can be used both as **stack** and **queue** as it allows the insertion and deletion operations on both ends.

In deque, the insertion and deletion operation can be performed from one side. The stack follows the LIFO rule in which both the insertion and deletion can be performed only from one end; therefore, we conclude that deque can be considered as a stack.



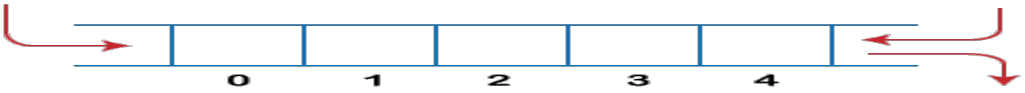
In deque, the insertion can be performed on one end, and the deletion can be done on another end. The queue follows the FIFO rule in which the element is inserted on one end and deleted from another end. Therefore, we conclude that the deque can also be considered as the queue.



There are two types of Queues, **Input-restricted queue**, and **output-restricted queue**.

1. **Input-restricted queue:** The input-restricted queue means that some restrictions are applied to the insertion. In input-restricted queue, the insertion is applied to one end while the deletion is applied from both the ends.

2. **Output-restricted queue:** The output-restricted queue means that some restrictions are applied to the deletion operation. In an output-restricted queue, the deletion can be applied only from one end, whereas the insertion is possible from both ends.



Operations on Deque

The following are the operations applied on deque:

- **Insert at front**
- **Delete from end**
- **insert at rear**
- **delete from rear**

Other than insertion and deletion, we can also perform **peek** operation in deque. Through **peek** operation, we can get the **front** and the **rear** element of the dequeue.

We can perform two more operations on dequeue:

- **isFull():** This function returns a true value if the stack is full; otherwise, it returns a false value.
- **isEmpty():** This function returns a true value if the stack is empty; otherwise it returns a false value.

The deque can be implemented using two data structures, i.e., **circular array**, and **doubly linked list**. To implement the deque using circular array, we first should know **what is circular array**.

Applications of Deque

- The deque can be used as a **stack** and **queue**; therefore, it can perform both redo and undo operations.
- It can be used as a palindrome checker means that if we read the string from both ends, then the string would be the same.
- It can be used for multiprocessor scheduling. Suppose we have two processors, and each processor has one process to execute. Each processor is assigned with a process or a job, and each process contains multiple threads. Each processor maintains a deque that contains threads that are ready to execute. The processor executes a process, and if a process creates a child process then that process will be inserted at the front of the deque

of the parent process. Suppose the processor P_2 has completed the execution of all its threads then it steals the thread from the rear end of the processor P_1 and adds to the front end of the processor P_2. The processor P_2 will take the thread from the front end; therefore, the deletion takes from both the ends, i.e., front and rear end. This is known as the **A-steal algorithm** for scheduling.

The following are the six functions used on deque

- **enqueue_front():** It is used to insert the element from the front end.
- **enqueue_rear():** It is used to insert the element from the rear end.
- **dequeue_front():** It is used to delete the element from the front end.
- **dequeue_rear():** It is used to delete the element from the rear end.
- **getfront():** It is used to return the front element of the deque.
- **getrear():** It is used to return the rear element of the deque.

7.PRIORITY QUEUE

A priority queue is an abstract data type that behaves similarly to the normal queue except that each element has some priority, i.e., the element with the highest priority would come first in a priority queue. The priority of the elements in a priority queue will determine the order in which elements are removed from the priority queue.

The priority queue supports only comparable elements, which means that the elements are either arranged in an ascending or descending order.

For example, suppose we have some values like 1, 3, 4, 8, 14, 22 inserted in a priority queue with an ordering imposed on the values is from least to the greatest. Therefore, the 1 number would be having the highest priority while 22 will be having the lowest priority.

Characteristics of a Priority queue

A priority queue is an extension of a queue that contains the following characteristics:

- Every element in a priority queue has some priority associated with it.
- An element with the higher priority will be deleted before the deletion of the lesser priority.
- If two elements in a priority queue have the same priority, they will be arranged using the FIFO principle.

Let's understand the priority queue through an example.

We have a priority queue that contains the following values:

1, 3, 4, 8, 14, 22

All the values are arranged in ascending order. Now, we will observe how the priority queue will look after performing the following operations:

- **poll():** This function will remove the highest priority element from the priority queue. In the above priority queue, the '1' element has the highest priority, so it will be removed from the priority queue.
- **add(2):** This function will insert '2' element in a priority queue. As 2 is the smallest element among all the numbers so it will obtain the highest priority.
- **poll():** It will remove '2' element from the priority queue as it has the highest priority queue.
- **add(5):** It will insert 5 element after 4 as 5 is larger than 4 and lesser than 8, so it will obtain the third highest priority in a priority queue.

Types of Priority Queue

There are two types of priority queue:

- **Ascending order priority queue:** In ascending order priority queue, a lower priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in an ascending order like 1,2,3,4,5; therefore, the smallest number, i.e., 1 is given as the highest priority in a priority queue.

- **Descending order priority queue:** In descending order priority queue, a higher priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in descending order like 5, 4, 3, 2, 1; therefore, the largest number, i.e., 5 is given as the highest priority in a priority queue.

Representation of priority queue

Now, we will see how to represent the priority queue through a one-way list.

We will create the priority queue by using the list given below in which **INFO** list contains the data elements, **PRN** list contains the priority numbers of each data element available in the **INFO** list, and LINK basically contains the address of the next node.

Let's create the priority queue step by step.

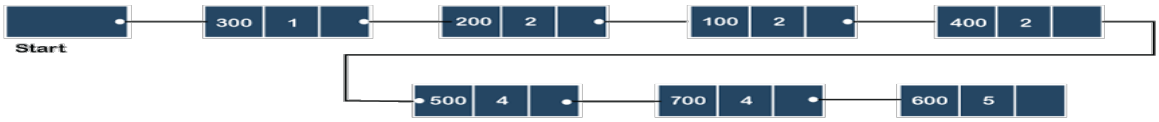
In the case of priority queue, lower priority number is considered the higher priority, i.e., lower priority number = higher priority.

Step 1: In the list, lower priority number is 1, whose data value is 333, so it will be inserted in the list as shown in the below diagram:

Step 2: After inserting 333, priority number 2 is having a higher priority, and data values associated with this priority are 222 and 111. So, this data will be inserted based on the FIFO principle; therefore 222 will be added first and then 111.

Step 3: After inserting the elements of priority 2, the next higher priority number is 4 and data elements associated with 4 priority numbers are 444, 555, 777. In this case, elements would be inserted based on the FIFO principle; therefore, 444 will be added first, then 555, and then 777.

Step 4: After inserting the elements of priority 4, the next higher priority number is 5, and the value associated with priority 5 is 666, so it will be inserted at the end of the queue.



Implementation of Priority Queue

The priority queue can be implemented in four ways that include arrays, linked list, heap data structure and binary search tree. The heap data structure is the most efficient way of implementing the priority queue, so we will implement the priority queue using a heap data structure in this topic. Now, first we understand the reason why heap is the most efficient way among all the other data structures.

Analysis of complexities using different implementations

Implementation	add	Remove	peek
Linked list	O(1)	O(n)	O(n)
Binary heap	O(logn)	O(logn)	O(1)
Binary search tree	O(logn)	O(logn)	O(1)

Applications of Priority queue

The following are the applications of the priority queue:

- It is used in the Dijkstra's shortest path algorithm.
- It is used in prim's algorithm
- It is used in data compression techniques like Huffman code.
- It is used in heap sort.
- It is also used in operating system like priority scheduling, load balancing and interrupt handling.

8.APPLICATIONS OF QUEUES

Due to the fact that queue performs actions on first in first out basis which is quite fair for the ordering of actions. There are various applications of queues :0

1. Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.
2. Queues are used in asynchronous transfer of data, for eg. pipes, file IO, sockets.
3. Queues are used as buffers in most of the applications like MP3 player, CD player, etc.
4. Queue are used to maintain the play list in media players in order to add and remove the songs from the play-list.
5. Queues are used in operating systems for handling interrupts.
6. Queues are used in simulation of a real world situation
7. Queues are used in Time Sharing OS
8. Queues are used in searching non-linear collection of state
9. Queues are used to find path using Breath First Search of graphs.

CHAPTER – III - LINKED LISTS

1.INTRODUCTION

A linked list is also a collection of elements, but the elements are not stored in a consecutive location.

Suppose a programmer made a request for storing the integer value then size of 4-byte memory block is assigned to the integer value. The programmer made another request for storing 3 more integer elements; then, three different memory blocks are assigned to these three elements but the memory blocks are available in a random location. So, how are the elements connected?.

These elements are linked to each other by providing one additional information along with an element, i.e., the address of the next element. The variable that stores the address of the next element is known as a pointer. Therefore, we conclude that the linked list contains two parts, i.e., the first one is **the data element**, and the other is the **pointer**. The pointer variable will occupy 4 bytes which is pointing to the next element.

A linked list can also be defined as the collection of the nodes in which one node is connected to another node, and node consists of two parts, i.e., one is the data part and the second one is the address part, as shown in the below figure:

Head

tail

In the above figure, we can observe that each node contains the data and the address of the next node. The last node of the linked list contains the **NULL** value in the address part.

The declaration of an array is very simple as it is of single type. But the linked list contains two parts, which are of two different types, i.e., one is a simple variable, and the second one is a pointer variable. We can declare the linked list by using the user-defined data type known as structure.

The structure of a linked list can be defined as:

```
class  Node
{
  int data;
  Node *link;
}
```

In the above declaration, we have defined a structure named as ***a Node*** consisting of two variables: an integer variable (data), and the other one is the pointer (link), which contains the address of the next node.

Advantages of using a Linked list over Array

The following are the advantages of using a linked list over an array:

- **Dynamic data structure:**
  The size of the linked list is not fixed as it can vary according to our requirements.
- **Insertion and Deletion:**
  Insertion and deletion in linked list are easier than array as the elements in an array are stored in a consecutive location. In contrast, in the case of a linked list, the elements are stored in a random location. The complexity for insertion and deletion of elements from the beginning is O(1) in the linked list, while in the case of an array, the complexity

would be O(n). If we want to insert or delete the element in an array, then we need to shift the elements for creating the space. On the other hand, in the linked list, we do not have to shift the elements. In the linked list, we just need to update the address of the pointer in the node.

- **Memory efficient**
  Its memory consumption is efficient as the size of the linked list can grow or shrink according to our requirements.
- **Implementation**
  Both the stacks and queues can be implemented using a linked list.

Disadvantages of Linked list

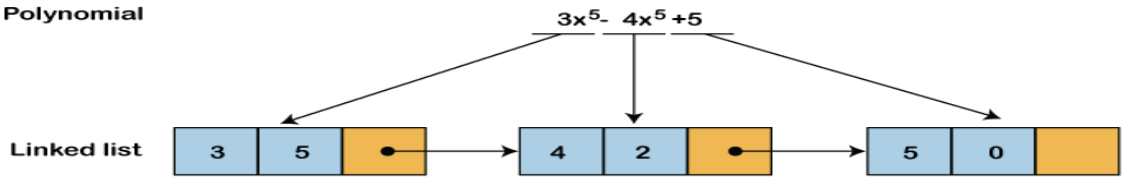
The following are the disadvantages of linked list:

- **Memory usage**
  The node in a linked list occupies more memory than array as each node occupies two types of variables, i.e., one is a simple variable, and another is a pointer variable that occupies 4 bytes in the memory.
- **Traversal**
  In a linked list, the traversal is not easy. If we want to access the element in a linked list, we cannot access the element randomly, but in the case of an array, we can randomly access the element by index. For example, if we want to access the 3rd node, then we need to traverse all the nodes before it. So, the time required to access a particular node is large.
- **Reverse traversing**
  In a linked list, backtracking or reverse traversing is difficult. In a doubly linked list, it is easier but requires more memory to store the back pointer.

Applications of Linked List

The applications of the linked list are given below:

- With the help of a linked list, the polynomials can be represented as well as we can perform the operations on the polynomial. We know that polynomial is a collection of terms in which each term contains coefficient and power. The coefficients and power of each term are stored as node and link pointer points to the next element in a linked list, so linked list can be used to create, delete and display the polynomial.



- A sparse matrix is used in scientific computation and numerical analysis. So, a linked list is used to represent the sparse matrix.
- The various operations like student's details, employee's details or product details can be implemented using the linked list as the linked list uses the structure data type that can hold different data types.
- Stack, Queue, tree and various other data structures can be implemented using a linked list.
- The graph is a collection of edges and vertices, and the graph can be represented as an adjacency matrix and adjacency list. If we want to represent the graph as an adjacency

matrix, then it can be implemented as an array. If we want to represent the graph as an adjacency list, then it can be implemented as a linked list.

- To implement hashing, we require hash tables. The hash table contains entries that are implemented using linked list.
- A linked list can be used to implement dynamic memory allocation. The dynamic memory allocation is the memory allocation done at the run-time.

2. VARIANTS OF LINKED LIST

The following are the types of linked list:

- Singly Linked list
- Doubly Linked list
- Circular Linked list
- Doubly Circular Linked list

3.SINGLY LINKED LIST

It is the commonly used linked list in programs. If we are talking about the linked list, it means it is a singly linked list.

The singly linked list is a data structure that contains two parts, i.e., one is the data part, and the other one is the address part, which contains the address of the next or the successor node. The address part in a node is also known as a **pointer**.

Suppose we have three nodes, and the addresses of these three nodes are 100, 200 and 300 respectively. The representation of three nodes as a linked list is shown in the below figure:

We can observe in the above figure that there are three different nodes having address 100, 200 and 300 respectively. The first node contains the address of the next node, i.e., 200, the second node contains the address of the last node, i.e., 300, and the third node contains the NULL value in its address part as it does not point to any node.

The pointer that holds the address of the initial node is known as a ***head pointer***.

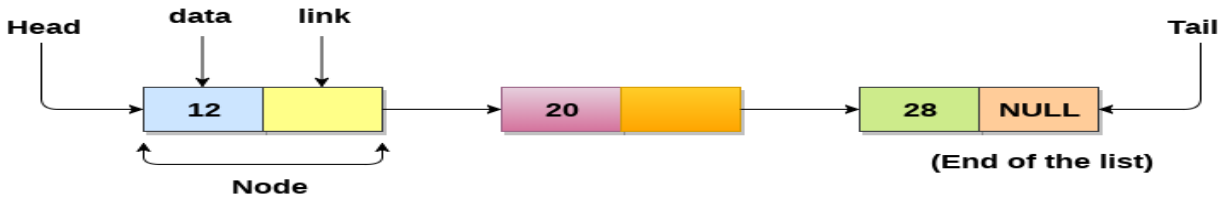
The linked list, which is shown in the above diagram, is known as a singly linked list as it contains only a single link. In this list, only forward traversal is possible; we cannot traverse in the backward direction as it has only one link in the list.

Data Structure of a node in a singly linked list

```
class Node
{  int data;
   Node *link;
}
```

In the above representation, we have defined a user-defined structure named a **node** containing two members, the first one is data of integer type, and the other one is the pointer (link) of the node type.

- Linked List can be defined as collection of objects called **nodes** that are randomly stored in the memory.
- A node contains two fields i.e. data stored at that particular address and the pointer which contains the address of the next node in the memory.
- The last node of the list contains pointer to the null.



Singly linked list can be defined as the collection of ordered set of elements. The number of elements may vary according to need of the program.

A node in the singly linked list consist of two parts: data part and link part. Data part of the node stores actual information that is to be represented by the node while the link part of the node stores the address of its immediate successor

One way chain or singly linked list can be traversed only in one direction. In other words, we can say that each node contains only next pointer, therefore we can not traverse the list in the reverse direction.

The last node in the list is identified by the null pointer which is present in the address part of the last node. We can have as many elements we require, in the data part of the list.

Operations on Singly Linked List

There are various operations which can be performed on singly linked list. A list of all such operations is given below.

Linked list ADT:

Linked list ADT is defined as

```
class  LinkedList
{
  private: Node *head, *tail;
  public: LinkedList( )
            { head=tail=NULL; }
          void  display( );
          void insPos(int ele, int pos);
          void delPos(int pos);
};
```

Creation :

To create an empty list, use default constructor in SigleLinkedList class. First declare head and tail as pointers to Node class, assign these to NULL to create an empty list.

```
LinkedList( )
{ head=tail=NULL; }
```

Create a list with element, use parameterized constructor in LinkedList class.

```
LinkedList(int ele)
{ Node *t;
  t=new Node;
  t->data=ele;
  t->link=NULL;
  head=tail=t;
 }
```

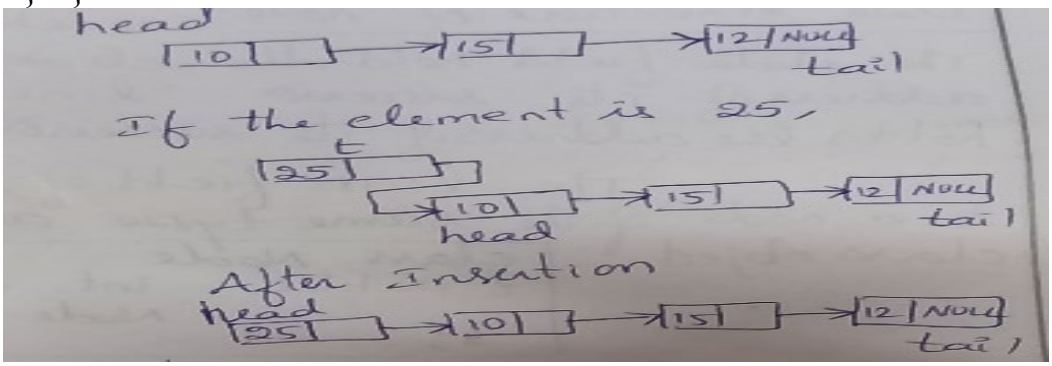
Create a temporary node t, allocate memory to it, store ele in t->data, assign NULL to t->link, now assign t to head and tail.

Insertion: Depending on the type of the list or need of the user, insertion can be made at begin, end, or at the given position of the list.

i)Insertion of a node at the beginning of the list:

- Head is a pointer which points the first node of the list.
- To insert a new node at beginning of the list, first create a new node t, allocate memory to it , store ele in t->data, and then link it to the head of the list.
- Now the new node t becomes the head of the list.

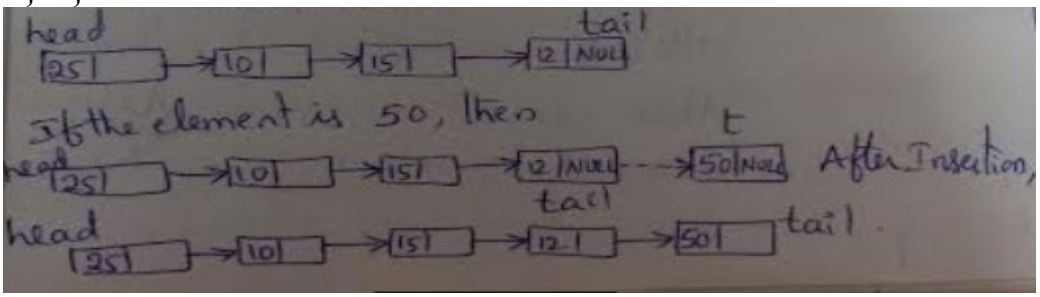
```
void  ins_begin(int ele)
{ Node *t;
  T=new Node;
 if(t!=NULL)
 {  t->data=ele;
    t->link=head;
    head=t;
 }  }
```



ii)Insertion of a node at the end of the list:

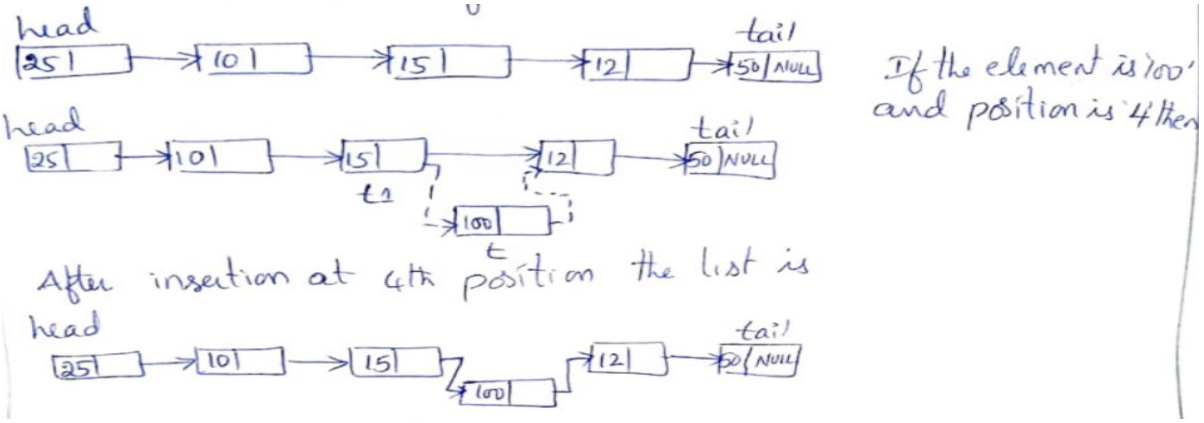
- Tail is a pointer which points the last node of the list.
- To insert a new node at end of the list, first create a new node t, allocate memory to it , store ele in t->data and NULL in t->link and then link it to the tail of the list.
- Now the new node t becomes the tail of the list.

```
void  ins_end(int ele)
{ Node *t;
  t=new Node;
 if(t!=NULL)
 {  t->data=ele;
    t->link=NULL;
     tail->link=t;
    tail=t;
 }  }
```



iii) insertion of a node at a given position

- To insert a node at given position take node pointer t1, and assign head to t1, then move t1 from head to the previous node of given position i.e., p-1 node.
- Then create a new node t, allocate memory to it, store ele in t->data
- t->link=t1->link, t1->link=t.(link t node t1 properly). t is inserted after t1.

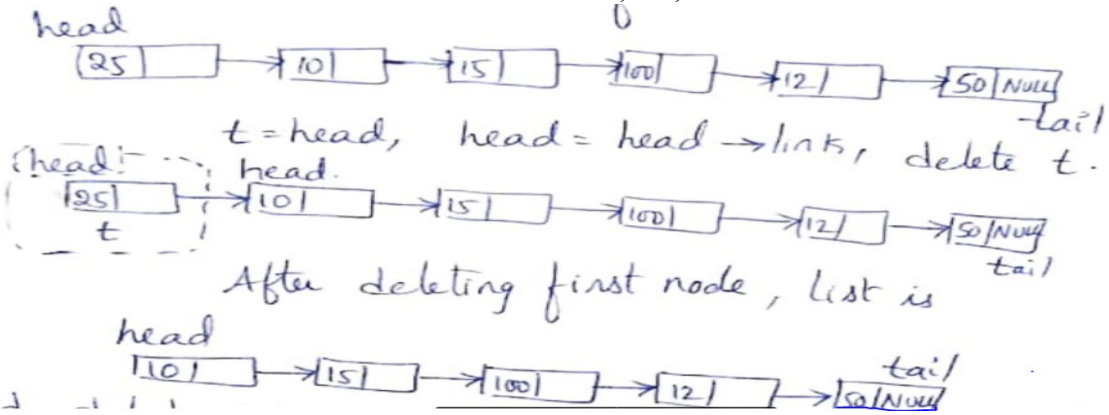



```
void  ins_pos(int ele, int p)
{ Node  *t, *t1;
  T=new Node;
  if(t!=NULL)
  { for(int i=1, t1=head; i<p-1; i++)
       t1=t1->link;
    t->data=ele;
    t->link=t1->link;
    t1->link=t;
     if(tail==t1)
       tail=t;
  }  }
```

Deletion: Depending on the type of list or need of the user, deletion can be made at begin, end, or at the given position of the list.

i)Deletion of a node from the beginning of the list

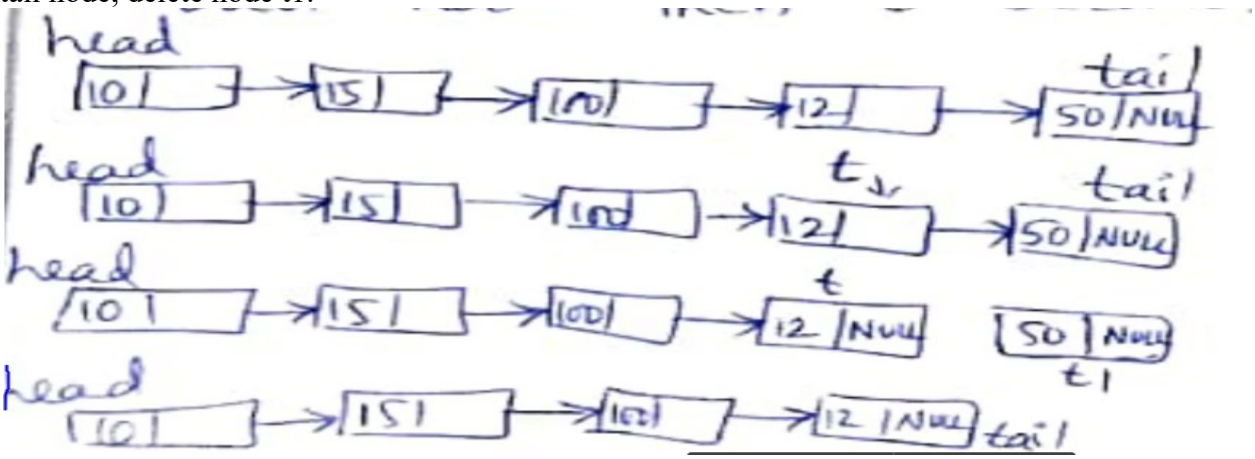
- Head is a pointer, which points to the first node of the list.
- To delete the first node, move it to temporary node t.
- Then the first node head becomes the head->link, i.e., 2nd node becomes the first node.



```
void del_begin()
{ Node *t;
  if(head==NULL)
     cout<<"List is Empty";
  else
  {   t=head;
      head=head->link;
      t->link=NULL;
     delete  t;
    if(head==NULL)
       tail=NULL; }
}
```

ii)Deletion of a node from the ending of the list

- Tail is a pointer, which points to the last node of the list.
- When the last node is deleted, then the previous node of the tail becomes tail node.
- Assign head to a node pointer t, move t to the previous node of the tail node.
- Assign tail to the node t1, then assign t->link to NULL, now assign t to tail, then t becomes tail node, delete node t1.

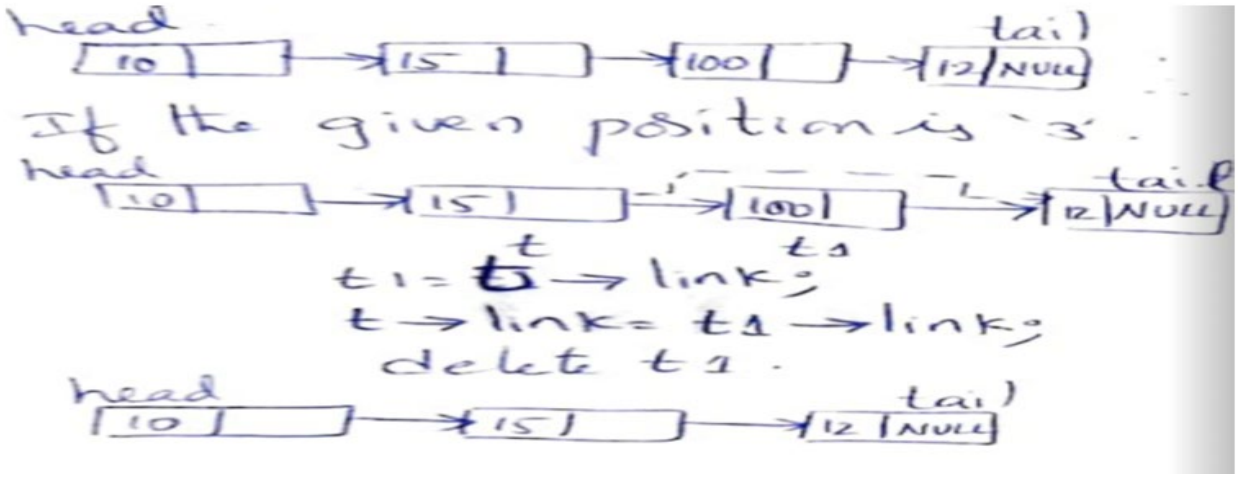



```
void del_end()
{   Node *t,*t1;
     if(head==NULL)
      cout<<"List is Empty";
  else
  {  t=head;
  t1=tail;
  while(t->link->link!=NULL)
          t=t->link;
  t->link=NULL;
  tail=t;
 delete  t1;  }
 if(tail==NULL)
   head=NULL;
}
```

iii)Deletion of a node from the given position

- To delete a node from a given position, move the temporary node t from the head to the previous position of the given position.
- Link t to the successor of the specified node at a given position.
- Move the node at given position to t1 and then delete t1 node.

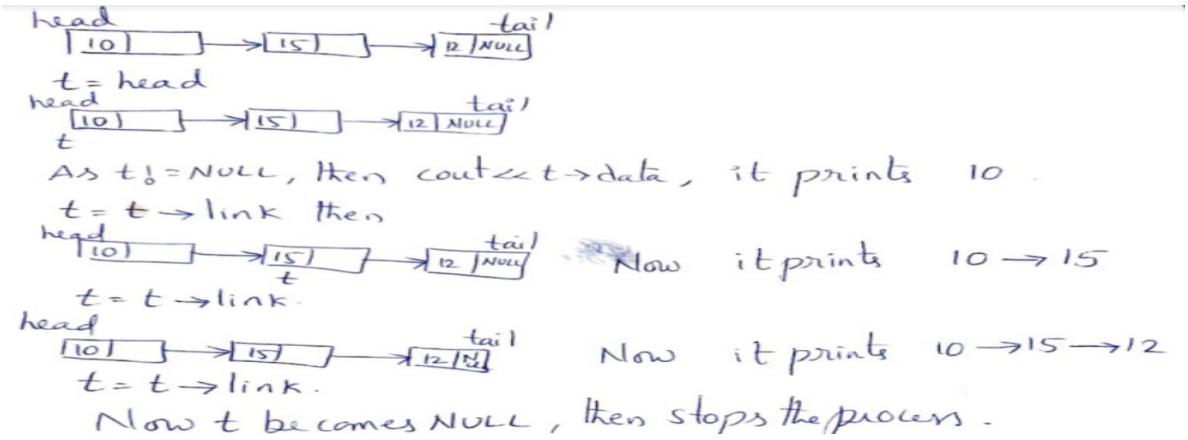
```
void del_pos()
{ Node *t,*t1;
  if(head==NULL)
     cout<<"List is Empty";
  else
  {  for(int i=1, t=head; i<p-1; i++)
          t=t->link;
  t1=t->link;
  t->link=t1->link;
  if(t1==tail)
   tail=t;
  delete  t1;   }  }
```



Traversal (display)

- List traversal is the basic operation where all elements of the list are processed sequentially, one by one.
- Processing involve retrieving, searching, sorting, finding length and so on.
- List traversal requires a looping algorithm. To traverse the linked list, start from the head node to tail node and display data part of nodes.

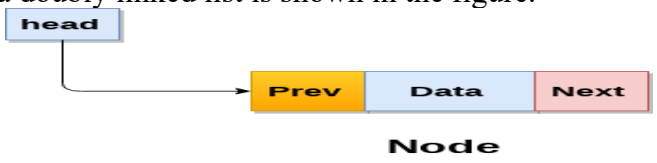
```
void  LinkedList::display( )
{ Node *t=head;
  if(t==NULL)
     cout<<"List is empty";
  else
     for(; t!=NULL; t=t->link)
            cout<<t->data<<"→";
  }
```



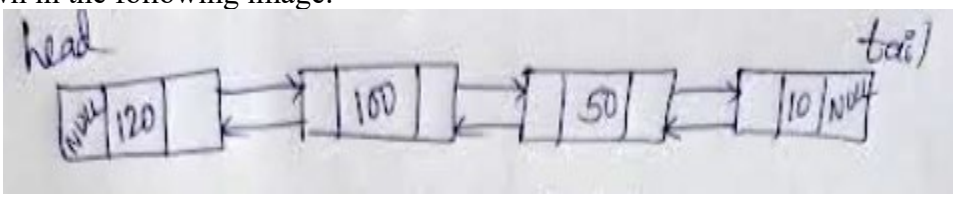
4.DOUBLY LINKED LIST

Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence.

Therefore, in a doubly linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer) , pointer to the previous node (previous pointer). A sample node in a doubly linked list is shown in the figure.



A doubly linked list containing four nodes having numbers 120, 100, 50 and 10 in their data part, is shown in the following image.



Declaration syntax of a Double Linked List node is

```
class  Node
{
   Node *prev;
   data_type  data;
   Node *next;
};
```

The **prev** part of the first node and the **next** part of the last node will always contain NULL indicating end in each direction.

In a singly linked list, we could traverse only in one direction, because each node contains address of the next node and it doesn't have any record of its previous nodes.

However, doubly linked list overcome this limitation of singly linked list. Due to the fact that, each node of the list contains the address of its previous node, we can find all the details about the previous node as well by using the previous address stored inside the previous part of each node.

Double Linked List ADT:

```
class  Node
  {  Node *prev;
     int  data;
     Node *next;
   };
class  DLList
{   private: Node *head, *tail;
   public: DLList( )
            {     head=tail=NULL;     }
            void ins(int ele);
            void del(int p);
            void  display();
};
```

Where DLList( ) is the default constructor, which creates an empty list.

Operations on doubly linked list

SN	Operation	Description
1	Insertion at beginning	Adding the node into the linked list at beginning.
2	Insertion at end	Adding the node into the linked list to the end.
3	Insertion at a given position	Adding the node into the linked list at a specified position
4	Deletion at beginning	Removing the node from beginning of the list
5	Deletion at the end	Removing the node from end of the list.
6	Deletion of the node from a given position	Removing the node from a specified position.
7	Searching	Comparing each node data with the item to be searched and return the location of the item in the list if the item found else return null.
8	Traversing	Visiting each node of the list at least once in order to perform some specific operation like searching, sorting, display, etc.

Creation of DLL with element:

- In DLL each node must be linked to both of its predecessor and successor.
- To create a DLL with element, use parameterized constructor.
- Declare a node pointer t, allocate memory to it.
- If memory is allocated, store element in t->data, then store NULL in t->next and t->prev and assign ‘t’ to head and tail.

```
DLList(int ele)                 // ele is 100
{   Node *t;
  t=new Node;
  if(t!=NULL)
  { t->data=ele;
    t->next=t->prev=NULL;
    head=tail=t;  }
 }
```

head

NULL	100	NULL

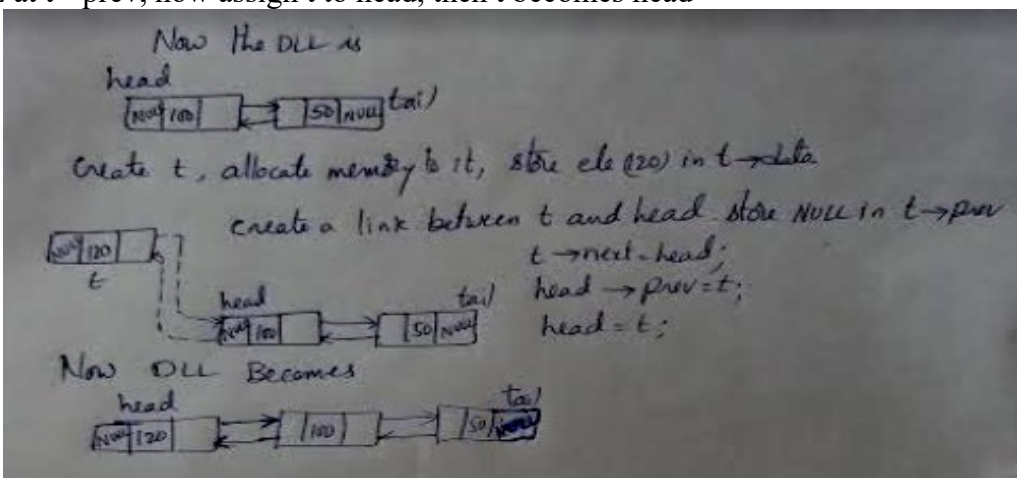
Tail

Insertion of a node to a DLL:

- To insert a node in DLL, we have to modify four links as each node points to its predecessor as well as successor.
- In DLL, the insertion of a node may be possible at beginning of DLL, end of DLL, and at a specific position.

Insertion of a node at beginning of DLL:

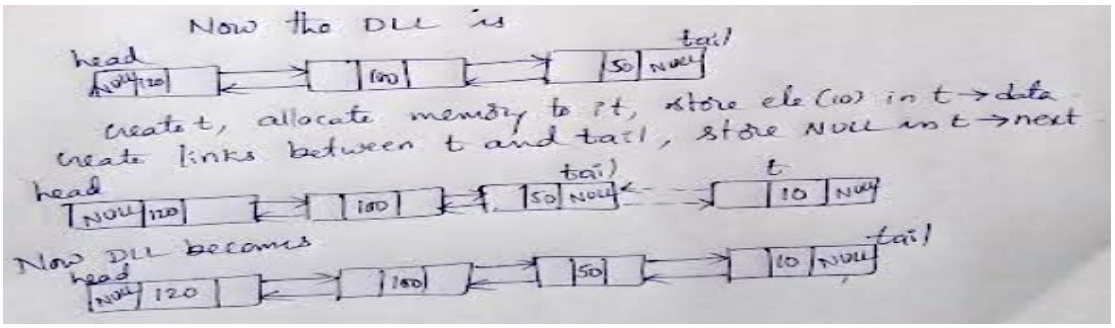
- Create a new node 't', allocate memory to it.
- If memory is allocated, store ele at t->data, (element is 120)
- Connect t and head using t->next=head, head->prev=t statements
- Store NULL at t->prev, now assign t to head, then t becomes head




```
void ins_begin(int ele)
{ Node *t;   t=new Node;
  if(t!=NULL)
  { t->data=ele;
    t->next=head;
    head->prev=t;
    t->prev=NULL;
    head=t;  }  }
```

Insertion of a node at ending of DLL:

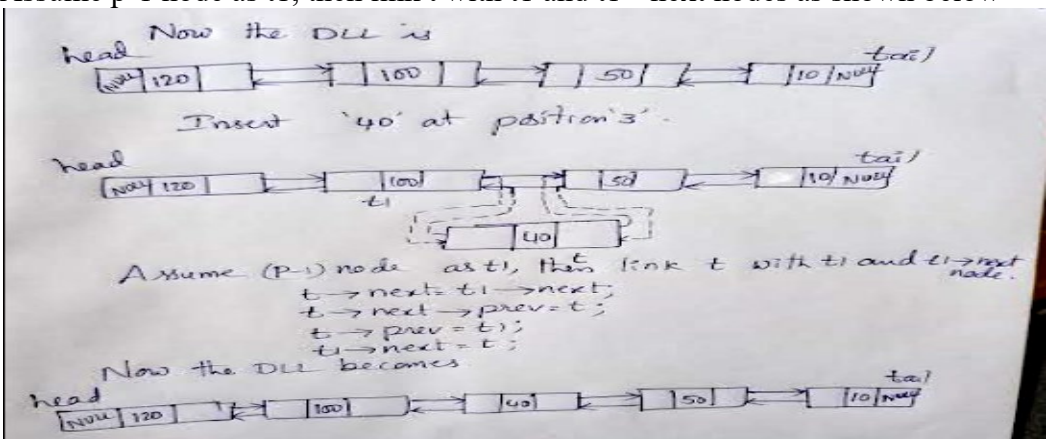
- Create a new node 't', allocate memory to it.
- If memory is allocated, store ele at t->data, (element is 10)
- Connect t and tail using t->prev=tail, tail->next=t; statements
- Store NULL at t->next, now assign t to tail, then t becomes tail.




```
void ins_end(int ele)
{ Node *t;
  t=new Node;
  if(t!=NULL)
  { t->data=ele;
    t->next=NULL;
    tail->next=t;
    t->prev=tail;
    tail=t; }  }
```

Insertion of a node at a specified position in DLL:

- Create a new node 't', allocate memory to it.
- If memory is allocated, store ele at t->data, (element is 40)
- Assume p-1 node as t1, then link t with t1 and t1->next nodes as shown below




```
void  ins_pos(int ele, int p)
{ Node *t, *t1;
  t=new Node;
  if(t!=NULL)
  { t->data=ele;
    for(int i=1, t1=head; i<p-1; i++)
          t1=t1->next;
    t->next=t1->next;
    t->next->prev=t;
    t->prev=t1;
    t1->next=t;
    if(t1==tail)
       tail=t;    }  }
```

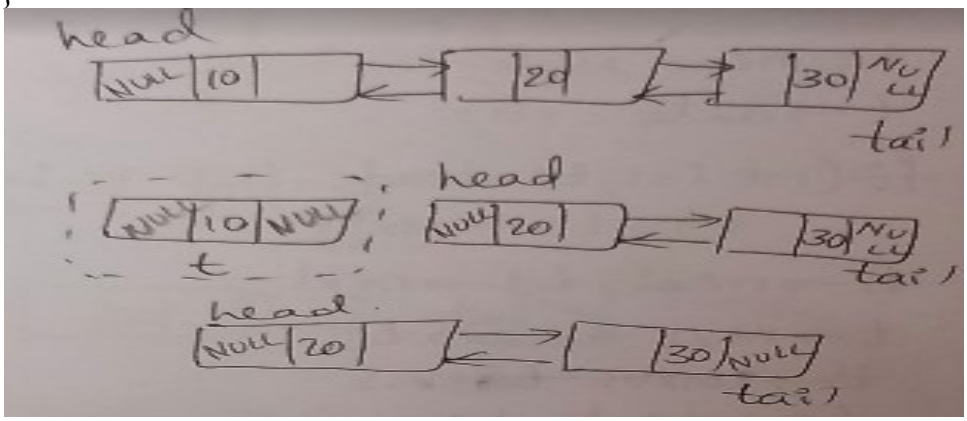
Deletion of a node from DLL:

- To delete a node from DLL, we have to modify four links as each node points to its predecessor as well as successor.
- In DLL, the deletion of a node may be possible at beginning of DLL, end of DLL, and at a specific position.

Deletion of a node from the beginning of DLL:

- To delete a node from the beginning of the DLL, the steps are, move the head node to 't', head to head->next, then store NULL at head->prev and t->next, finally delete 't' node.

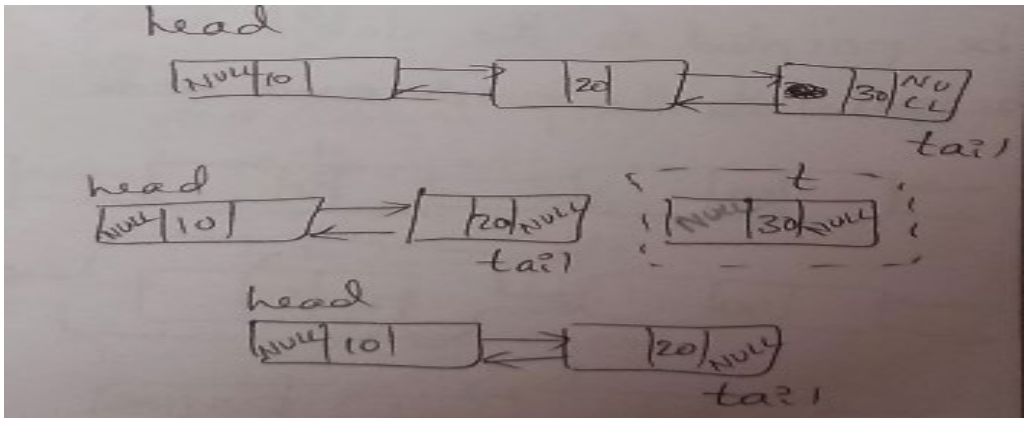
```
void  del_begin( )
{
 Node *t;
 if(head==NULL)
    cout<<"Empty List";
 else
{t=head;
 head=head->next;
 head->prev=t->next=NULL;
delete t;
 if(head==NULL)
  tail=head;  }
}
```



Deletion of a node from the ending of DLL:

- To delete the last node of a DLL, move the tail node to 't', then tail becomes tail->prev, Finally store NULL at the tail->next and t->prev, and remove node 't'.

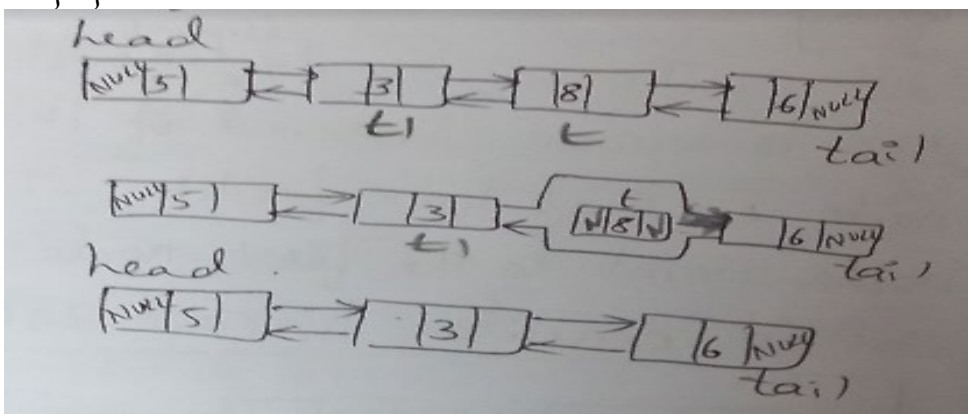
```
void  del_end( )
{  Node *t;
 if(head==NULL)
    cout<<"Empty List";
 else
{t=tail;
 tail=tail->prev;
 t->prev=tail->next=NULL;
delete t;
 if(head==NULL)
  tail=head;  }}
```



Deletion of a node from a specified position of DLL:

- To delete a node from a given position of a DLL, move the list pointer t1 to the previous position of the given position.
- Move node to be deleted to 't' by using t1->next, then link t1->next to t->next, t->next->prev to t1, finally store NULL in t->next and t->prev.
- Now delete node 't'.

```
void del_pos(int p)
{ Node *t1, *t;
  if(head==NULL)
     cout<<"Empty List";
else{
        for(int i=1, t1=head; i<p-1; i++)
            t1=t1->next;
        t=t1->next;
        t1->next=t->next;
        t->next->prev=t1;
        t->prev=t->next=NULL;
        delete  t;
   } }
```



Traversal of DLL:

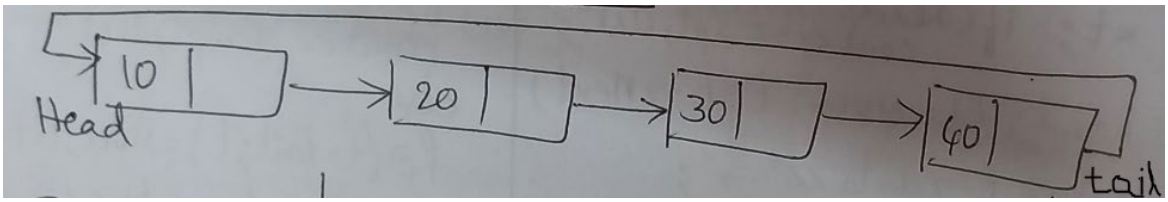
In DLL, the list can be traversed in both directions, take a pointer 't' to the DLL, move it from head to tail or tail to head.

```
void  display( )
{ Node *t;
```

```
if(head==NULL)
  cout<<"empty list";
else
{  for(t=head; t!=NULL; t=t->next)
      cout<<t->data<<"->";
   cout<<"\n";
  for(t=tail; t!=NULL; t=t->prev)
     cout<<t->data<<"->";
}  }
```

5.SINGLE CIRCULAR LINKED LIST

- All elements of a linear list can be accessed from first node to the last node. In SLL, we can't reach the first node from the last node, this can be overcome by using circular SLL.
- In a Single Linked List, the last node link is set to NULL. Instead of that, store the address of the first node in the last node link.
- This change will make the last node point to the first node of the list. Such a linked list is called circular linked list



- In this type of list, it is possible to reach head node from the tail node.
- Circular linked list is used to keep track of free space in memory, major applications of circular list are time slicing and memory management.
- We can have a Circular SLL and Circular DLL.
- In a Single Circular Linked List, the head points to the first node of the list. From the last node, we can access the first node, last node link field stores the address of the first node.
- We can't access the last node directly through the head node, for that we need to traverse the whole list to reach the last node.
- To insert a new node at the front of the list, change the link field of the tail node, traverse the whole list to reach till the last node and then change its link field.
- Circular linked list are mostly used in task maintenance in operating systems.
- There are many examples where circular linked list are being used in computer science including browser surfing where a record of pages visited in the past by the user, is maintained in the form of circular linked lists and can be accessed again on clicking the previous button.

Operations on Circular Singly linked list:

SN	Operation	Description
1	Insertion at beginning	Adding the node into the linked list at beginning.
2	Insertion at end	Adding the node into the linked list to the end.
3	Insertion at a given position	Adding the node into the linked list at a specified position

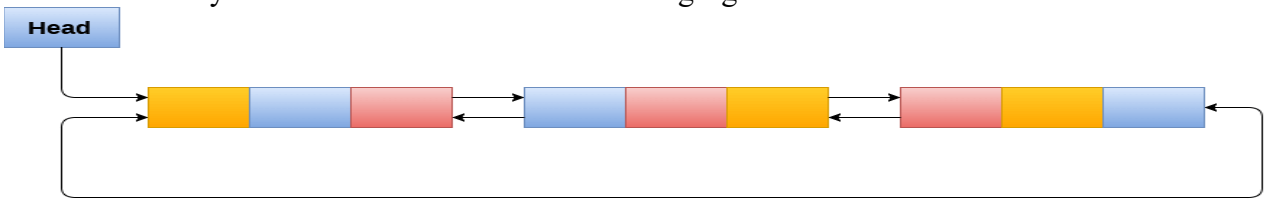
4	Deletion at beginning	Removing the node from beginning of the list
5	Deletion at the end	Removing the node from end of the list.
6	Deletion of the node from a given position	Removing the node from a specified position.
7	Searching	Comparing each node data with the item to be searched and return the location of the item in the list if the item found else return null.
8	Traversing	Visiting each node of the list at least once in order to perform some specific operation like searching, sorting, display, etc.

6.CIRCULAR DOUBLY LINKED LIST

Circular doubly linked list is a more complexed type of data structure in which a node contain pointers to its previous node as well as the next node.

Circular doubly linked list doesn't contain NULL in any of the node. The last node of the list contains the address of the first node of the list. The first node of the list also contain address of the last node in its previous pointer.

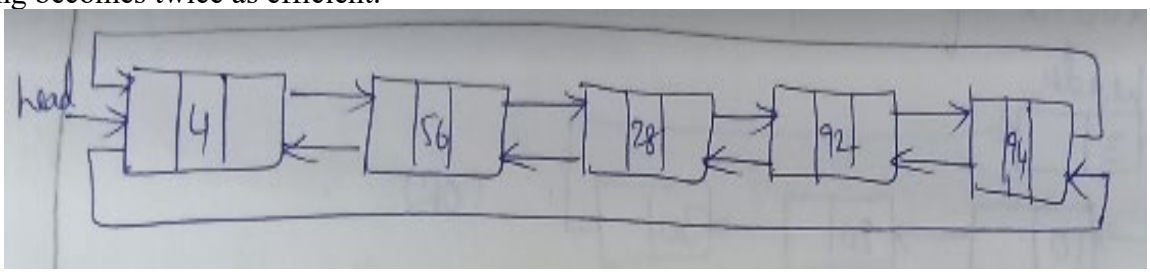
A circular doubly linked list is shown in the following figure.



Circular Doubly Linked List

Due to the fact that a circular doubly linked list contains three parts in its structure therefore, it demands more space per node and more expensive basic operations.

However, a circular doubly linked list provides easy manipulation of the pointers and the searching becomes twice as efficient.



Operations on circular doubly linked list :

There are various operations which can be performed on circular doubly linked list. The node structure of a circular doubly linked list is similar to doubly linked list.

However, the operations on circular doubly linked list is described in the following table.

SN	Operation	Description
1	Insertion at beginning	Adding a node in circular doubly linked list at the beginning.
2	Insertion at end	Adding a node in circular doubly linked list at the end.
3	Insertion at a given position	Adding the node into the linked list at a specified position
4	Deletion at beginning	Removing a node in circular doubly linked list from beginning.
5	Deletion at end	Removing a node in circular doubly linked list at the end.
6	Deletion of the node from a given position	Removing the node from a specified position.
7	Searching	Comparing each node data with the item to be searched and return the location of the item in the list if the item found else return null.
8	Traversing	Visiting each node of the list at least once in order to perform some specific operation like searching, sorting, display, etc.

Traversing and searching in circular doubly linked list is similar to that in the circular singly linked list.

7.SPARSE MATRIX

A matrix can be defined as a two-dimensional array having 'm' columns and 'n' rows representing m*n matrix.

Sparse matrices are those matrices that have the majority of their elements equal to zero. In other words, the sparse matrix can be defined as the matrix that has a greater number of zero elements than the non-zero elements.

We can also use the simple matrix to store the elements in the memory; then why do we need to use the sparse matrix. The following are the advantages of using a sparse matrix:

- **Storage:** As we know, a sparse matrix that contains lesser non-zero elements than zero so less memory can be used to store elements. It evaluates only the non-zero elements.
- **Computing time:** In the case of searching n sparse matrix, we need to traverse only the non-zero elements rather than traversing all the sparse matrix elements. It saves computing time by logically designing a data structure traversing non-zero elements.

The non-zero elements can be stored with triples, i.e., rows, columns, and value. The sparse matrix can be represented in the following ways:

i)Array representation ii)Linked list representation

Representing a sparse matrix by a 2D array leads to the wastage of lots of memory. The zeroes in the matrix are of no use to store zeroes with non-zero elements.

To avoid such wastage, we can store only non-zero elements. If we store only non-zero elements, it reduces the traversal time and the storage space.

Array Representation

The 2d array can be used to represent a sparse matrix in which there are three rows named as:

1. Row: It is an index of a row where a non-zero element is located.
2. Column: It is an index of the column where a non-zero element is located.
3. Value: The value of the non-zero element is located at the index (row, column).

Let's understand the sparse matrix using array representation through an example.

Sparse matrix →

	0	1	2	3
0	0	4	0	5
1	0	0	3	6
2	0	0	2	0
3	2	3	0	0
4	0	0	0	0

As we can observe above, that sparse matrix is represented using triplets, i.e., row, column, and value. In the above sparse matrix, there are 13 zero elements and 7 non-zero elements. This sparse matrix occupies 5*4 = 20 memory space. If the size of the sparse matrix is increased, then the wastage of memory space will also be increased. The above sparse matrix can be represented in the tabular form shown as below:

Table Structure

Row	Column	Value
0	1	4
0	3	5
1	2	3
1	3	6
2	2	2
3	0	2
3	1	3

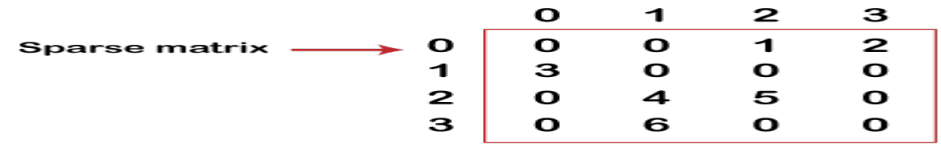
In the above table structure, the first column is representing the row number, the second column is representing the column number and third column represents the non-zero value at index(row, column). The size of the table depends upon the number of non-zero elements in the sparse matrix. The above table occupies (7 * 3) = 21 but it more than the sparse matrix. Consider the case if the matrix is **8*8** and there are only 8 non-zero elements in the matrix then the space occupied by the sparse matrix would be 8*8 = 64 whereas, the space occupied by the table represented using triplets would be 8*3 = 24.

Linked List Representation

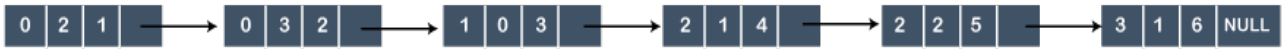
In linked list representation, linked list data structure is used to represent a sparse matrix. In linked list representation, each node consists of four fields whereas, in array representation, there are three fields, i.e., row, column, and value. The following are the fields in the linked list:

- Row: It is an index of row where a non-zero element is located.
- Column: It is an index of column where a non-zero element is located.
- Value: It is the value of the non-zero element which is located at the index (row, column).
- Next node: It stores the address of the next node.

Let's understand the sparse matrix using linked list representation through an example.



Linked List Representation



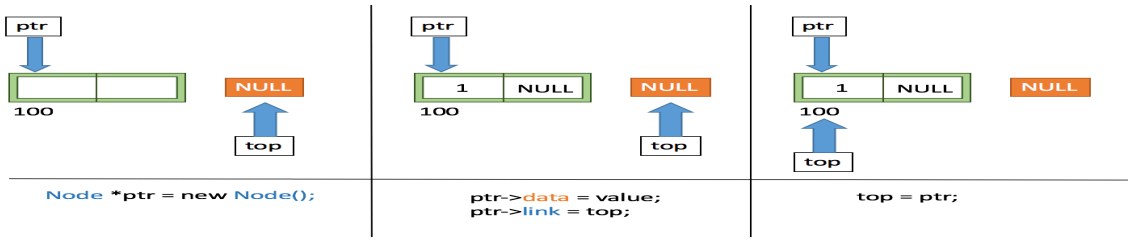
In the above figure, sparse represented in the linked list form. In the node, first field represents the index of row, second field represents the index of column, third field represents the value and fourth field contains the address of the next node.

8.LINKED STACK

- A stack is an abstract data structure that contains a collection of elements.
- Stack implements the LIFO mechanism i.e. the element that is pushed at the end is popped out first. Some of the principle operations in the stack are −
  1. Push - This adds a data value to the top of the stack.
  2. Pop - This removes the data value on top of the stack.
  3. Gettop- This returns the top data value of the stack.
- Stack is a data structure which follows LIFO i.e. Last-In-First-Out method.
- The data/element which is stored last in the stack i.e. the element at top will be accessed first.
- And both insertion & deletion takes place at the top.
- When we implement stack using array :
  1. We create an array of predefined size & we cannot increase the size of the array if we have more elements to insert.
  2. If we create a very large array, we will be wasting a lot of memory space.
  3. So to solve this lets try to implement Stack using Linked list where we will dynamically increase the size of the stack as per the requirement.
  4. Taking this as the basic structure of our Node:

Implementation using Linked List

- As we know that we use a head pointer to keep track of the starting of our linked list, So when we are implementing stack using linked list we can simply call the head pointer as top to make it more relatable to stack.
- An empty stack is created by using a default constructor, and initialize top to NULL.
- **Push (insert element in stack)**
  1. The push operation would be similar to inserting a node at starting of the linked list
  2. So initially when the Stack (Linked List) is empty, the top pointer will be NULL. Let's suppose we have to insert the values 1, 2 & 3 in the stack.
  3. So firstly we will create a new Node using the new operator and return its address in temporary pointer ptr.
  4. Then we will insert the value 1 in the data part of the Node : ptr->data = value and make link part of the node equal to top : ptr->link=top.
  5. Finally we will make top = ptr to point it to the newly created node which will now be the starting of the linked list and top of our stack.



6. Similarly we can push the values 2 & 3 in the stack which will give us a linked list of three nodes with top pointer pointing to the node containing value 3.

- **Pop (delete element from stack)**
  1. The pop operation would be similar to deleting a node from the starting of a linked list.
  2. So we will take a temporary pointer ptr and equate it to the top pointer.
  3. Then we will move the top pointer to the next node i.e. top = top->link
  4. Finally, we will delete the node using delete operator and pointer ptr i.e delete(ptr)
- **isEmpty (check if stack is empty or not)**
  1. To check if the stack is empty or not, we can simply check if top == NULL, it means that the stack is empty.
- **getTop( )**
  1. If the stack is not empty, then it returns the top most element of the stack, top->data

Stack ADT implementation using linked list is :

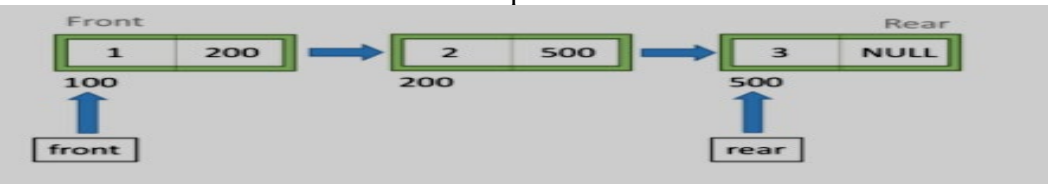
```
#include<iostream.h>
#include<conio.h>
#include<stdlib.h>
class  Node
{ int  data;
  Node *link;
};
class Stack
{       private: Node *top;
        public: Stack()
                { top=NULL;}
                void  push(int ele);
                void pop( );
                void getTop( );
};
void  Stack::push(int ele)
{ Node *t;
  t=new Node;
  if(t==NULL)
    cout<<"NO MEMORY, STACK OVERFLOW\n";
  else
   {    t->data=ele;
        t->link=top;
        top=t;
        cout<<ele<<"  PUSHED\n";
   }
}
```

```
void  Stack::pop( )
{ if(top==NULL)
        cout<<"STACK UNDERFLOW\n";
  else
   {    Node *t=top;
        top=top->link;
         t->link=NULL;
        cout<<t->data<<" POPPED\n";
        delete t;  }
}
void  Stack::getTop( )
{ if(top==NULL)
        cout<<" EMPTY STACK\n";
  else
        cout<<top->data<<" IS THE TOPMOST ELEMENT\n";
}
void  main( )
{  Stack  s;
   int  choice, ele;
   clrscr( );
   for(; ;)
   { cout<<"1.PUSH\n2.POP\n3.GETTOP\n4.EXIT\nENTER YOUR CHOICE: ";
     cin>>choice;
     switch(choice)
     {  case  1:  cout<<"ENTER ELEMENT\n";
                  cin>>ele;
                 s.push(ele);   break;
        case  2:  s.pop( );   break;
        case  3:  s.getTop( ); break;
        case  4:  exit(0);
        default: cout<<"WRONG CHOICE";
     }
  }  }
```

9.LINKED QUEUE

- Queue is a linear data structure which follows FIFO i.e. First-In-First-Out method.
- The two ends of a queue are called Front and Rear, where Insertion always takes place at the Rear and the elements are accessed or removed from the Front.
- A queue is an abstract data structure that contains a collection of elements. Queue implements the FIFO mechanism i.e the element that is inserted first is also deleted first.
- While implementing queues using arrays:
  1. We cannot increase the size of array, if we have more elements to insert than the capacity of array.
  2. If we create a very large array, we will be wasting a lot of memory space.
- Therefore if we implement Queue using Linked list we can solve these problems, as in Linked list Nodes are created dynamically, when required.

- So using a linked list we can create a Queue of variable size and depending on our need we can increase or decrease the size of the queue.



- Thus instead of using the head pointer with Linked List, we will use two pointers, front and rear to keep track of both ends of the list.
- We can easily add a node from rear end, and delete a node from front end.

Implementation using Linked List

- As we know that we use a head pointer to keep track of the starting of our linked list, So when we are implementing queue using linked list we can simply call the head pointer as front, tail pointer as rear. An empty queue is created by using a default constructor, and initialize front and rear to NULL.
- **add (adding an element to the queue)**
  1. The add operation would be similar to inserting a node at ending of the linked list
  2. So initially when the Queue (Linked List) is empty, the rear and front pointers will be NULL. Let's suppose we have to insert the values 1, 2 & 3 in the Queue.
  3. So firstly we will create a new Node using the new operator and return its address in temporary pointer t.
  4. Then we will insert the value 1 in the data part of the Node : t->data = value and make link part of the node equal to rear : t->link=rear.
  5. If front=NULL then front=rear=t, otherwise rear->link=t, Finally we will make rear = t to point it to the newly created node which will now be the ending of the linked list and rear of our queue.
  6. Similarly we can add the values 2 & 3 in the queue which will give us a linked list of three nodes with rear pointer pointing to the node containing value 3, front pointer pointing to the node containing value 1.
- **Del (delete element from queue)**
  1. The del operation would be similar to deleting a node from the starting of a linked list.
     - So we will take a temporary pointer t and equate it to the front pointer.
     - Then we will move the front pointer to the next node i.e. front=front->link
     - Finally, store NULL in t->link, and delete the node using delete operator as 'delete t'
- **isEmpty (check if queue is empty or not)**
  - To check if the queue is empty or not, we can simply check if rear== NULL, it means that the queue is empty.
- **getFront( )**
  - If the queue is not empty, then it returns the front most element of the queue, front->data.

Queue ADT implementation using linked list is :

```
#include<iostream.h>
#include<conio.h>
#include<stdlib.h>
class  Node
{  int  data;
   Node *link;
  };
```

```
class Queue
{       private: Node *front, *rear;
        public: Queue( )
                { front=rear=NULL; }
                void add(int ele);
                void del( );
                void getFront( );
 };
void Queue::add(int ele)
{ Node *t;
  t=new Node;
  if(t==NULL)
    cout<<"NO MEMORY, QUEUE OVERFLOW\n";
  else
   {    t->data=ele;
        t->link=NULL;
        if(front==NULL)
        front=rear=t;
        else
        { rear->link=t;
          rear=t;
          cout<<ele<<" ADDED\n";  }
}
void Queue::del( )
{ if(front==NULL)
        cout<<"QUEUE UNDERFLOW\n";
  else
   {    Node *t=front;
        front=front->link;
        t->link=NULL;
        cout<<t->data<<" DELETED\n";
        delete t; }
}
void Queue::getFront( )
{ if(front==NULL)
        cout<<" EMPTY STACK\n";
  else
        cout<<front->data<<" IS THE FRONT ELEMENT\n";
}
void main( )
{ Queue q;
   int choice, ele;
   clrscr( );
   for(; ;)
   { cout<<"1.ADD\n2.DEL\n3.GETFRONT\n4.EXIT\nENTER YOUR CHOICE: ";
     cin>>choice;
```

```
    switch(choice)
    {  case  1:  cout<<"ENTER ELEMENT\n";
                 cin>>ele;
                 q.add(ele);
                 break;
       case  2:  q.del( );
                 break;
       case  3:  q.getFront( );
                 break;
       case  4:  exit(0);
       default: cout<<"WRONG CHOICE";
     }
   }
}
```

10.APPLICATIONS OF LINKED LIST

1. Implementation of stacks and queues
2. Implementation of graphs : Adjacency list representation of graphs is most popular which is uses linked list to store adjacent vertices.
3. Dynamic memory allocation : We use linked list of free blocks.
4. Maintaining directory of names
5. Performing arithmetic operations on long integers
6. Manipulation of polynomials by storing constants in the node of linked list
7. Representing sparse matrices