Q Explain various type of searching?
Q Explain Linear search?
Q Explain Binary search?

Searching is an operation which finds the location of a given element in a list.The search is said to be successful or unsuccessful depending on whether the element that is to be searched is found or not. There are two standard searching methods

1. Linear Search
2. Binary Search

Linear Search

This is simplest method of searching. In this method ,the element to be found is sequentially searched in the list. This method can be applied to a sorted list or unsorted list.

Linear Search (Sorted List)

Searching in case of Sorted list starts from 0^{th} elements and continues until the element is found or an element whose value is greater than the value being searched is reached.

Algorithm

1. Declare array AA
2. store elements in to array AA
3. take number to be searched from user and store in **num**
4. repeat for i=o to (length of AA)-1
5. check if AA[length of AA-1] < num or AA[i]>= num
6. if AA[i]==num
7. print: num found at location i
8. else
9. print: num not in array
10. repeat step 5 to 9
11. end of loop

```
import java.io.*;
class  LinearSearch
{
  int arr[]=new int[10];
 InputStreamReader isr= new InputStreamReader(System.in);
BufferedReader br =new BufferedReader(isr);

  int num;
void input()
            {
              try
              {

              System.out.println( "Enter values for array " ) ;
```

```
                for(int i=0;i<10;++i)
                        {
                          String s=br.readLine();
                arr[i]=Integer.parseInt(s);
                          }

                System.out.println( "Enter values to be searched " ) ;
                  num=Integer.parseInt(br.readLine());
              }catch(Exception e)
              {}
    }

void linearSorted( )
{

              for ( int i = 0 ; i < arr.length ; ++i )
              {
                        if ( arr[9] < num || arr[i] >= num )
                        {
                                  if ( arr[i] == num )
                                            System.out.println ( "The number is at position"+ i+ " in the array." ) ;
                                  else
                             System.out.println( "Number is not present in the array." ) ;
                                  break ;
                        }
              }

}
}

class LinearSearchDemo
{

              public static void main(String[] args)
              {
     LinearSearch ob=new LinearSearch();
     ob.input();
               ob.linearSorted();
              }
}
```

Linear Search (UnSorted List)

Searching in this case of unsorted list starts from 0^{th} element and continue until the element is found or the end of list is reached

Algorithm

1 Declare array AA
2 store elements in to array AA
3 take nmber to be searched from user and store in **num**
4 repeat for i=o to l(ength of AA)-1
5 check if AA[i]==num
6 come out of loop

7 repeat step 6 to 7
8 check if i== length of AA
9 print: num not found
10 else
11 print : num found at location i
12 end of loop

```
import java.io.*;
class  LinearSearch
{
  int arr[]=new int[10];
 InputStreamReader isr= new InputStreamReader(System.in);
BufferedReader br =new BufferedReader(isr);

  int num;
void input()
            {
              try
              {

              System.out.println( "Enter values for array " ) ;
                for(int i=0;i<10;++i)

                          {
                           String s=br.readLine();
               arr[i]=Integer.parseInt(s);
                          }


              System.out.println( "Enter values to be searched " ) ;
                num=Integer.parseInt(br.readLine());

            }catch(Exception e)
            {}
   }

void linearUnsorted( )
{         int i;

                        for ( i = 0 ; i <= 9 ; i++ )
                {
                          if ( arr[i] == num )
                                    break ;
                }
      if ( i == 10 )
                System.out.println( "Number is not present in the  array." ) ;
      else
                        System.out.println( "Number is  present in the  array.at location " +i) ;

 }
}
class LinearUnSearchDemo
{

            public static void main(String[] args)
            {
    LinearSearch ob=new LinearSearch();
```

```
    ob.input();
            ob.linearUnsorted();
            }
}
```

Q Explain Binary Search

This method is very fast and efficient. This method require that the list of the elements be in sorted order. In this method to search an element we compare it with the element present at the center of the list.If it matches then the search is successful. Otherwise the list is divided into two halves. One from 0^{th} element to the center element (first half) and another from center element to last element(second half).As a result all the element in the first half are smaller than the center element whereas all the element in the second half are greater than the center element.

The searching will now proceed in either of the two halves depending upon whether the element is greater or smaller than the center element. If the element is smaller than the center element than the searching will be done in the first half, otherwise in second half.

Same process of comparing the required element with the center element and if not found then dividing the element into two halves is repeated for the first half or second half. This process is repeated till the element is found or the division of half parts gives one element.

Algorithm

```
1    Declare array AA
2   store elements in to array AA
3    take number to be searched from user and store in num
4    set lower =0,upper = length of AA-1,flag =1
       repeat for  mid = (lower+upper)/2  to  lower< =upper
5           check  if AA[mid] == num
6                          print: num found at location mid
7                           flag=0,  break
8
9           check AA[mid] > num
10                        upper= mid-1;
11                  else
12                        lower = mid+1;
13              mid =(lower+upper)/2
14   repeat step  5 to 9
15    end of loop
```

```
import java.io.*;
class  BinarySearch
{
  int arr[]=new int[10];
  InputStreamReader isr= new InputStreamReader(System.in);
BufferedReader br =new BufferedReader(isr);
  int num;
  int mid, lower , upper;
  boolean flag;
```

```
BinarySearch()
        {
        lower=0;
        upper=arr.length-1;
        flag=true;
        }
void input()
        {
          try
          {

          System.out.println( "Enter values for array " ) ;
            for(int i=0;i<10;++i)
                    {
                     String s=br.readLine();
           arr[i]=Integer.parseInt(s);
                    }
          System.out.println( "Enter values to be searched " ) ;
            num=Integer.parseInt(br.readLine());
        }catch(Exception e)
        {}
   }

void binarySearch( )
 {

        for ( mid = ( lower + upper ) / 2 ; lower <= upper ;mid = ( lower + upper ) / 2 )
        {
                if ( arr[mid] == num )
                {
                        System.out.println ( "The number is at position %d in the array. "+ mid ) ;
                        flag = false;
                        break ;
                }

                if ( arr[mid] > num )
                        upper = mid - 1 ;
                else
                        lower = mid + 1 ;
        }

        if ( flag )
                System.out.println( "Element is not present in the array." ) ;
 }
}
class BinarySearchDemo
{

        public static void main(String[] args)
        {
   BinarySearch ob=new BinarySearch();
    ob.input();
         ob.binarySearch();
        }
}
```

Complexity

The complexity is measured by the number f(n) of comparisons to locate the ITEM in the array when array contains n elements. As each comparison reduces the array size in half. Hence we require at most f(n) comparisons to locate ITEM where

$2^{f(n)} > n$ or equivalently $f(n) = [\log_2 n] + 1$

That is running time for the worst case is approximately equal to $\log_2 n$

Q Explain Sorting

Sorting means arranging a set of data in some order.There are different that are used to sort the data in ascending or descending order.Various type of sorting are

1. Bubble Sort
2. Selection Sort
3. Insertion Sort
4. Quick sort
5. Merge Sort
6. Binary tree Sort
7. Heap Sort

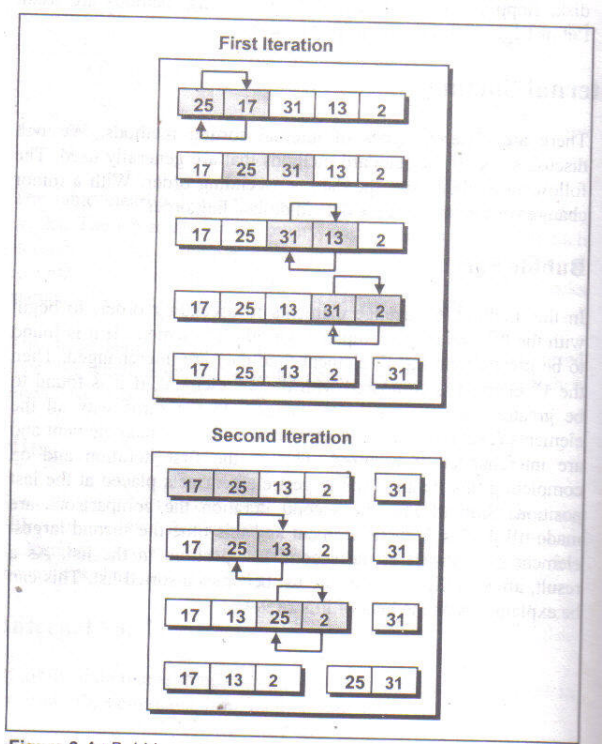
Q Explain Bubble Sort

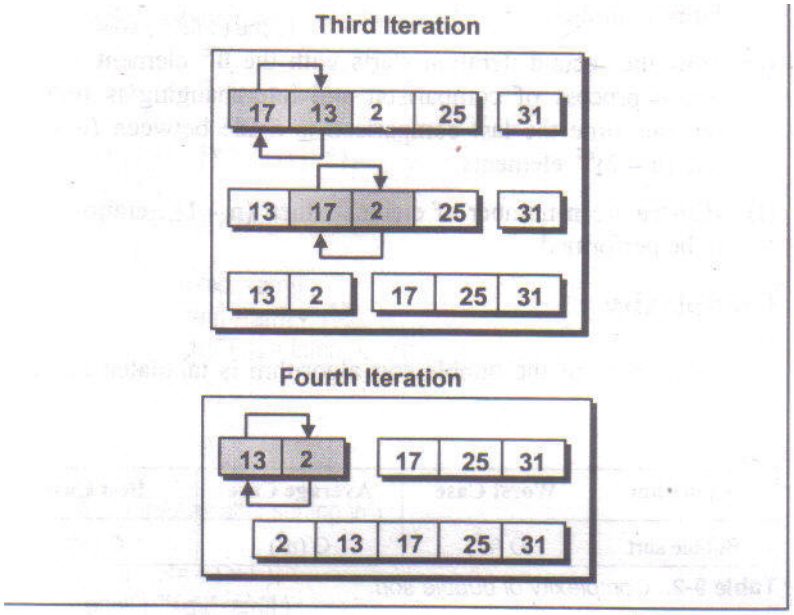
In this method 0th element is compared with the 1st element. If it is found to be greater than the 1st element then they are interchanged .Then the first element is compared with the 2nd element, if it is found to be greater, then they are interchanged. In this way all the elements are compared with the next element and interchanged if required. This is the first iteration and on completing this iteration the last element gets placed at the last position.

Similarlly in the second iteration the comparisons are made till the last but one element and this time the second largest element gets placed at the second last position in the list.As a result after all the iteration the list becomes a sorted list.

Algorithm

```
1    Declare array AA
2    store elements in to array AA
3
4     Repeat for i= 0 to i<=3                    // for passess
5         Repeat for j=0 to j<=3-i                 // for comparision
6              check  if AA[j] >AA[j+1]            // for exchanging largest number
7                                int  k = AA[j]
 8                                 AA[j]=AA[j+1]
 9                                 AA[j+1] =k
10            increment j by 1
11           repeat step 6 to 9
12           End of loop
13     increment i by 1
14      repeat step  5 to 11
13     end of loop
```






```
import java.io.*;
class  BubbleSort
{
  int arr[]=new int[10];
  InputStreamReader isr= new InputStreamReader(System.in);
BufferedReader br =new BufferedReader(isr);
```

```
 void input()
            {
              try
              {

              System.out.println( "Enter values for array " ) ;
                for(int i=0;i<arr.length;++i)
                        {
                          String s=br.readLine();
               arr[i]=Integer.parseInt(s);
            }


            }catch(Exception e)
            {}
    }

void bubbleSort( )
 { int i,j,temp;
            for ( i = 0 ; i <arr.length-1 ; i++ )
                        System.out.println( arr[i] + "  " ) ;


            for ( i = 0 ; i <arr.length-2 ; i++ )
            {
                        for ( j = 0 ; j <(arr.length-2)-i; j++ )
                        {
                                    if ( arr[j] > arr[j + 1] )
                                    {
                                                temp = arr[j] ;
                                                arr[j] = arr[j + 1] ;
                                                arr[j + 1] = temp ;
                                    }
                        }
            }
            System.out.println( "\n\nArray after sorting:\n") ;
            for ( i = 0 ; i <arr.length-1 ; i++ )
                        System.out.println( arr[i] + "  " ) ;

}
}
class BubbleDemo
{

            public static void main(String[] args)
            {
    BubbleSort ob=new BubbleSort();
     ob.input();
              ob.bubbleSort();
            }
}
```

Complexity

The time for a sorting algorithm is measured in terms of the numbers of comparisons. The function f(n) of comparisons in the bubble sort is easily computed. Specially there are n-1 comparisons during the first pass, which places the largest item in the last position. There are n-2 comparisons in the second step, which places the second largest element in the next to last position and so on.

$F(n)=(n-1)+(n-2)+\ldots.2+1= n*(n-1)/2.$

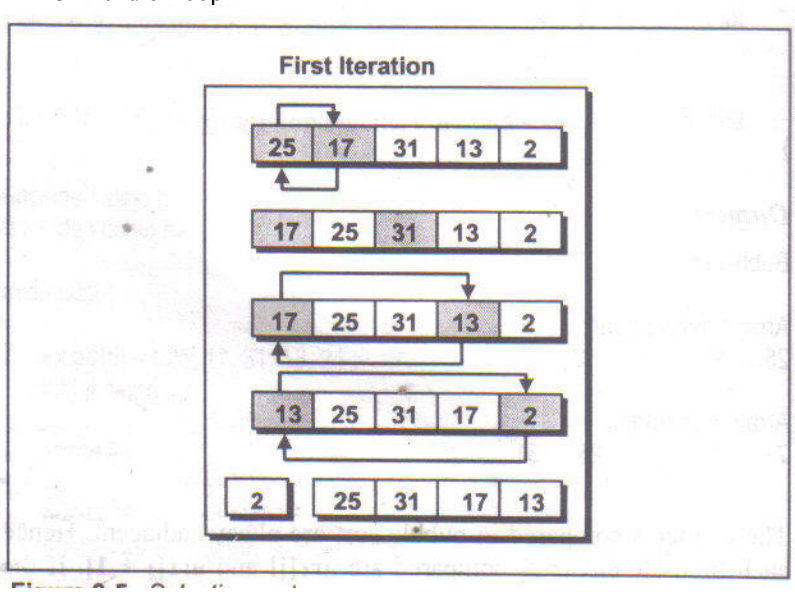
In other terms the time required to execute the bubble sort is proportional to n^2

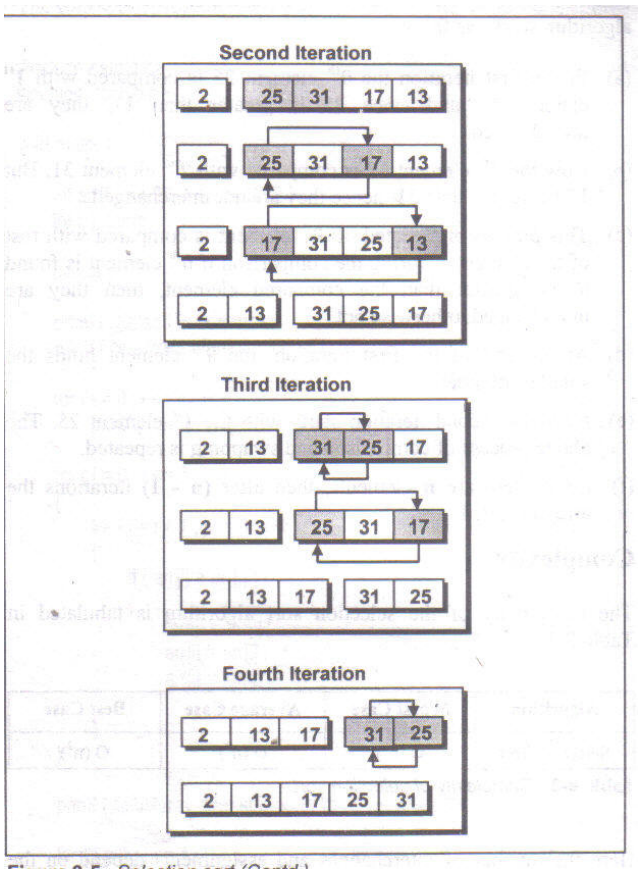
Q Explain Selection Sort

This is the simplest method of sorting. In this method ,to sort the data in ascending order, the 0^{th} element is compared with all other elements. If the 0^{th} element is found to be greater then the compared element then they are interchanged. So after the first iteration the smallest element is placed at the 0^{th} position. The same procedure is repeated for the 1^{st} element and so on.

Algorithm

```
1    Declare array AA
2    store elements in to array AA
3
4     Repeat for i= 0 to i<=3                  // for passes
5         Repeat for j=0 to j<=4               // for comparison
6             check  if AA[i] >AA[j]           // for exchanging largest number
7                              int  k = AA[i]
8                               AA[i]=AA[j]
9                               AA[j] =k
10          increment j by 1
11         repeat step 6 to 9
12         End of loop
13    increment i by 1
14     repeat step  5 to 11
13    end of loop
```






```
import java.io.*;
class  SelectionSort
{
  int arr[]=new int[10];
  InputStreamReader isr= new
InputStreamReader(System.in);
BufferedReader br =new BufferedReader(isr);

 void input()
            {
              try
              {

              System.out.println( "Enter values for
array " ) ;
                for(int i=0;i<arr.length;++i)
                          {
                           String s=br.readLine();
               arr[i]=Integer.parseInt(s);
                          }

            }catch(Exception e)
            {}
   }

void selectionSort( )
```

```
class SelectionDemo
{

            public static void main(String[] args)
            {
   SelectionSort ob=new SelectionSort();
    ob.input();
             ob.selectionSort();
            }
}
```

```
 {
        int i,j,temp;
        for ( i = 0 ; i <= arr.length-2 ; i++ )
        {
                for ( j = i + 1 ; j <= arr.length-1 ;
j++ )
                {
                        if ( arr[i] > arr[j] )
                        {
                                temp = arr[i] ;
                                arr[i] = arr[j] ;
                                arr[j] = temp ;
                        }
                }
        }

System.out.println( "\n\nArray after sorting:\n") ;

        for ( i = 0 ; i <= 4 ; i++ )
                        System.out.println( arr[i]
+ "  " ) ;

}
}
```

Complexity

First note that the number f(n) of comparisons in the selection sort algorithm is independent of the original ordering of the elements. Observe that (n-1) comparisons are required during first pass and (n-2) comparisons required during second pass. Accordingly total comparisons required are.

F(n)=(n-1)+(n-2)+….2+1= n*(n-1)/2.

In other terms the time required to execute the selection sort is proportional to n^2

Q Explain Quick Sort or partition Exchanger

This sorting method sort the data faster then any of the common sorting algorithm.This algorithm is based on the fact that it is faster and easier to sort two small arrays than one large one.The basic strategy is to **divide and conquer.**

This method pick a element from the array ,known as **pivot element** and divide the array into two parts(array).Then take the first array(part) and subdivide it into two array .The array will be further broken down into two array.This process goes on until the array are small enough to be easily sorted.

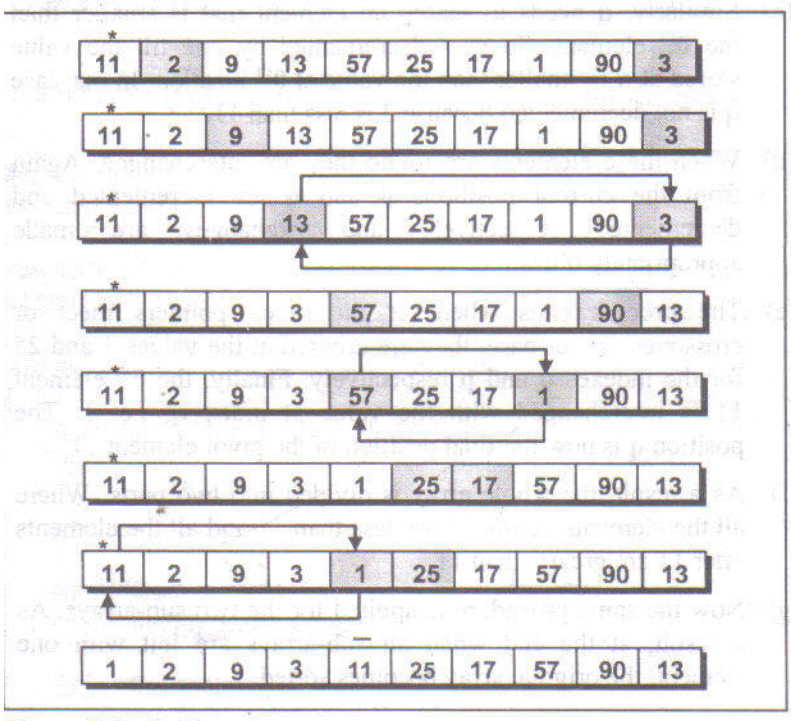
This strategy is based on recursion

Example

Suppose that an array consists of 10 distinct elements.The quick sort algorithm works as follows

11	2	9	13	57	25	17	1	90	3

a) In the first iteration,we will place the 0^{th} element 11 at its final position and divide the arrayHere 11 is pivot element .To divide the array two index variable p and q are taken.The index are initialized in such a way that p refers to the 1^{st} element 2 and q refers to $(n-1)^{th}$ element 3.
b) The job of index variable p is to search an element that is greater than the value 0^{th} location.So p is incremented by one till the value stored at p is greater than 0^{th} element.In our case it is incremented till 13, as 13 is greater that 11.
c) Similarly q needs to search element that is smaller than 0^{th} element.So q is decremented by one till the value stored at q is smaller than the value at 0^{th} element.In our case q is not decremented because 3 is less than 11
d) When these elements are found they are interchanged .Again from the current position p and q are incremented and decremented respectively and exchange are made appropriately
e) The process ends whenever the index pointers meet or crossover.In our case they are crossed at the values 1 and 25 for the indexes q and p respectively.Finally the 0^{th} element 11 is interchanged with the value at index q i.e. 1 the position q is now the final position of the pivot element 11.
f) As a result the whole array is divided into two parts.where all elements before 11 are less than 11 and all the elements after 11 are greater than 11.
g) Now the same procedure is applied for two sub arrays.As a result ,at the end when all sub arrays are left with one element,the original array become sorted.



Algorithm

1 Declare array AA
2 store elements in to array AA
3 quicksort(aa, lower,upper)
4. print : sorted elements

```
        quicksort (aa[],int lower,int upper)
1.  check          if ( upper > lower )
2.                  i= call  split ( a, lower, upper ) ;
3.                  quicksort ( a, lower, i - 1 ) ;
4.                  quicksort ( a, i + 1, upper ) ;

        split(int a[], int lower,int upper)
1.        set p=lower+1;
2.        set q=upper;
3.       set i= a[lower];
4.  Repeat  till q >= p
5.      Repeat  till a[p]<i
6.            ++p
7.       Repeat   till  a[q]>i
8.              ++q;
9.      check if  q>p
10               t= a[p];
11                a[p] =a[q]
12                a[q] =t
13  End loop
14       set      t = a[lower] ;
15       set      a[lower] = a[q] ;
16       set      a[q] = t ;
17       return q ;
```

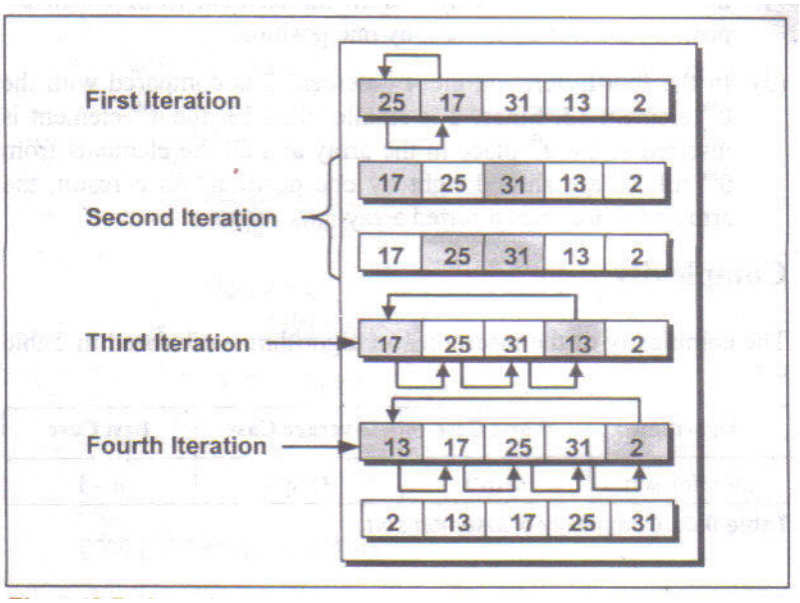
```
import java.io.*;
class  QuickSort
{
  int arr[]=new int[10];
  InputStreamReader isr= new
InputStreamReader(System.in);
BufferedReader br =new BufferedReader(isr);

 void input()
          {
            try
            {

            System.out.println( "Enter values for
array " ) ;
              for(int i=0;i<arr.length;++i)
                        {
                        String s=br.readLine();
            arr[i]=Integer.parseInt(s);
                        }

          }catch(Exception e)
          {}
   }

void sort()
          {
quickSort (arr, 0,arr.length-1);
          }
void quickSort (int a[], int lower, int upper)
```

```
class QuickDemo
{

          public static void main(String[] args)
          {
    QuickSort ob=new QuickSort();
     ob.input();
           ob.sort();
          }
}
```

```
        {
        int i ;
        if ( upper > lower )
        {
                i = split ( a, lower, upper ) ;
                quickSort ( a, lower, i - 1 ) ;
                quickSort ( a, i + 1, upper ) ;
        }
}

int split ( int a[ ], int lower, int upper )
{
        int i, p, q, t ;

        p = lower + 1 ;
        q = upper ;
        i = a[lower] ;

        while ( q >= p )
        {
                while ( a[p] < i )
                        p++ ;

                while ( a[q] > i )
                        q-- ;

                if ( q > p )
                {
                        t = a[p] ;
                        a[p] = a[q] ;
                        a[q] = t ;
                }
        }

        t = a[lower] ;
        a[lower] = a[q] ;
        a[q] = t ;

        return q ;
}
}
```

Q Explain Insertion Sort

Insertion sort is implemented by inserting a particular element at the appropriate position.In this method the first iteration starts with comparison of 1^{st} element with the 0^{th} element.In the second iteration 2^{nd} element is compared with the 0^{th} and 1^{st} element .In general in every iteration an element is compared with all element before it. During comparison if it is found that the element in question can be inserted at a suitable position then space is created for it by shifting the other element at the suitable position. This procedure is repeated for all elements in the array



Algorithm

1 Declare array AA
2 store elements in to array AA
3 set i=0
4 Repeat till i= 0 to i<=length of array-1 // for passess
5 Repeat for j=0 to j<i // for comparision
6 check if AA[j] >AA[i] // for exchanging largest number
7 int k = AA[j]
8 AA[j]=AA[i]
9 set h =i
10 Repeat till h>j
11 arr[h] = arr[h - 1] ;
12 arr[h+ 1] = k ;
13 decrement h by 1
14 end of loop
15 repeat step 6 to 14
16 End of loop
17 increment i by 1
18 repeat step 5 to 18
19 end of loop

```
import java.io.*;
class  InsertionSort
{
  int arr[]=new int[10];
  InputStreamReader isr= new
InputStreamReader(System.in);
BufferedReader br =new BufferedReader(isr);


 void input()
            {
              try
              {

              System.out.println( "Enter values for
```

```
            class InsertionDemo
{

            public static void main(String[] args)
            {
    InsertionSort ob=new InsertionSort();
     ob.input();
             ob.insertionSort();
            }
}
```

```
array " ) ;
                for(int i=0;i<arr.length;++i)
                        {
                         String s=br.readLine();
              arr[i]=Integer.parseInt(s);
                        }


           }catch(Exception e)
           {}
   }


void insertionSort ()
           { int i,j,temp,k;
            for ( i = 1 ; i <=arr.length-1 ; i++ )
            {
     for ( j = 0 ; j < i ; j++ )
                {
                 if ( arr[j] > arr[i] )
                  {
                        temp = arr[j] ;
                           arr[j] = arr[i] ;
                           for ( k = i ; k > j ; k-- )
                       arr[k] = arr[k - 1] ;
                                  arr[k + 1] = temp ;
                                  }
                       }
           }
           System.out.println( "\n\nArray after
sorting:\n") ;
           for ( i = 0 ; i <= arr.length-1 ; i++ )
                       System.out.println( arr[i] +"  ") ;
   }
}
```

Complexity

The number f(n) of comparisons in the insertion sort algorithm can be easily computed. First of all the worst case occurs when the array A is in reverse order and the inner loop must use the maximum number (k-1) comparisons. Hence

F(n)=1+2+3+…..+(n-1) = n*(n-1)/2

Further more, one can show that on the average, there will be approximately (k-1)/2 comparisons in the inner loop. So for average case,

F(n)=1/2+2/2+3/2+…..+(n-1)/2 = n*(n-1)/4

Thus this algorithm is slow when n is very large.

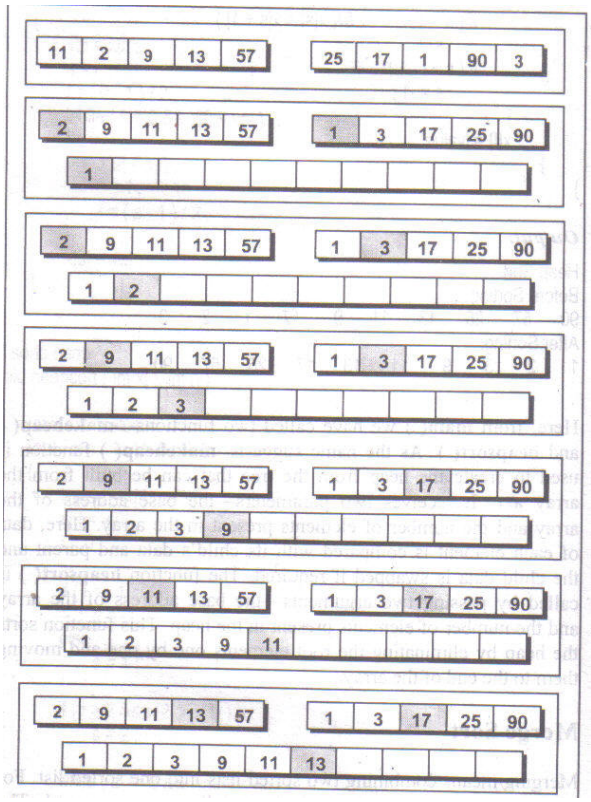
Q Explain Merge Sort

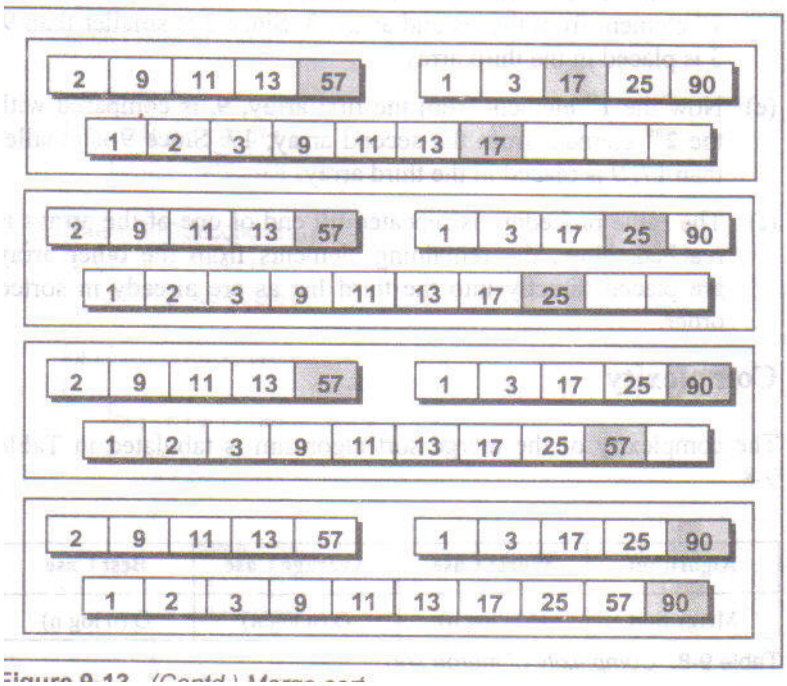
Merging means combining two sorted list. For this the elements from both the sorted list are compared. The smaller of both the elements is then stored in the third array. The sorting is complete when all the elements from both the lists are placed in the third list

Algorithm

1 Declare array AA , BB and CC

```
2     store elements in to array AA and BB
3      set i=j=k=0 // i for AA   j for BB   and k for CC
4      Repeat till i<=length of array CC-1
5                          check  if AA[i] <=BB[j]
6                                    CC[k]= AA[i]
7                                      ++k;++i;
8                                  else
9                                    CC[k]= BB[j]
10                                      ++k;++j;
11           repeat step 6 to 10
12          End of loop
```






```
import java.io.*;
class  MergeSort
{
  int arr1[]=new int[10];
int arr2[]=new int[10];
int arr3[]=new int[20];
  InputStreamReader isr= new
InputStreamReader(System.in);
BufferedReader br =new BufferedReader(isr);



 void input()
            {
              try
              {

              System.out.println( "Enter values for first
array " ) ;
                for(int i=0;i<arr1.length;++i)
                                        {
                        String s=br.readLine();
              arr1[i]=Integer.parseInt(s);
                        }

              System.out.println( "Enter values for
second array " ) ;
                for(int i=0;i<arr2.length;++i)
                                        {
                        String s=br.readLine();
              arr2[i]=Integer.parseInt(s);
                        }
```

```
class MergeSortDemo
{

            public static void main(String[] args)
            {
    MergeSort ob=new MergeSort();
     ob.input();
              ob.mergeSort();
            }
}                                   if ( b[i] > b[j] )
                                    {
                                              temp = b[i] ;
                                              b[i] = b[j] ;
                                              b[j] = temp ;
                                    }
                          }
            }
```

```
        }catch(Exception e)
        {}
    }


void mergeSort ()
        {
        int i,j,temp,k;
        for ( i = 0 ; i <= arr1.length-2 ; i++ )
        {
                for ( j = i + 1 ; j <= arr1.length;
j++ )
                {
                        if ( arr1[i] > arr1[j] )
                        {
                                temp = arr1[i] ;
                                arr1[i] = arr1[j]
;
                                arr1[j] = temp ;
                        }
                        if ( arr2[i] > arr2[j] )
                        {
                                temp = arr2[i] ;
                                arr2[i] = arr2[j]
;
                                arr2[j] = temp ;
                        }
                }
        }

    for ( i = j = k = 0 ; i <= arr3.length-1 ; )
          {
                if ( arr1[j] <=  arr2[k] )
                        arr3[i++] = arr1[j++] ;
                else
                        arr3[i++] = arr2[k++] ;

                if ( j == 5 || k == 5 )
                        break ;
        }
        for ( ; j <= 4 ; )
                arr3[i++] = arr1[j++] ;

        for ( ; k <= 4 ; )
                arr3[i++] = arr2[k++] ;

        System.out.println( "\n\nArray after
sorting:\n") ;
        for ( i = 0 ; i <= arr3.length ; i++ )
                System.out.println( arr3[i] + "  " )
;
        }
}
```

Complexity
The input consists of the total number n= r+ s, where r is the length of first array A and s is the length of second array B. Each comparisons assigns an element to the array C, which eventually has n elements. Accordingly the number f(n) of comparisons cannot exceed n

$$F(n)<=n$$

In other words the merge algorithm can run in linear time.

Stack

Q Explain Stack
A stack is a data structure in which addition of new element or deletion of an existing element always takes place at the same end. This end is known as **top** of the stack. This situation can be compared to a stack of plates in a cafeteria where every new plate added is at the top and every plate is taken off the top. Stack always follows **LIFO** structure where data inserted last will be taken out first. If the elements are added to the stack it grows at one end, and if elements are removed it shrinks.
The operation of insertion of element and deletion of element from the stack are given special names. When the element is added to the stack, the operation is called **PUSH** .when the element is removed from the stack is called **POP**

Q Implement stack as an array ?

Stack contains an ordered collection of element. An array is used to store ordered list of elements. Hence it would be very easy to manage a stack if we represent it using an array However the problem with array is that we are require to declare the size of array before using it in a program. It means the size of array is fixed. Stack on the other hand does not have any fixed size.

Algorithm

```
 void push( int data)
1       check tos
2.        if tos >= 9
3.         print: stack overflow
4         else
5            ++tos                          // increment tos;
 6     store data  AA[tos]=data           // store data
```

int pop()

1. check tos
2. if tos<=0
3. print :stack underflow
4. return 0
5. else
6. data= AA[tos] //retrieve data
7. - –tos //decrement tos
8. return data

```
import java.io.*;
class  Stack
{
 int arr[]=new int[10];
 InputStreamReader isr= new
InputStreamReader(System.in);
 BufferedReader br =new BufferedReader(isr);
 int tos;

 Stack()
 {
 tos=-1;
 }
  void push (int data)
            {
       if(tos>=arr.length-1)
          System.out.println("Over Flow");
       else
         {
         ++tos;
         arr[tos]=data;
         }
    }
  int pop()
  {
       if(tos<0)
       {
          System.out.println("under Flow");
         return -1;
       }
       else
         {
         int data;

        data=arr[tos];
        --tos;
        return data;
         }
  }
boolean isEmpty()
    {
     if(tos==-1)
       return true;
    else
```

```
class StackDemo
{

            public static void main(String[] args)
            {
   Stack ob=new Stack();
     for(int i=0;i<13;++i)
       ob.push(i);
            for(int i=0;i<13;++i)
     System.out.println(ob.pop());
    }
}
```

```
        return false;
    }
boolean isFull()
    {
     if(tos==9)
        return true;
    else
        return false;
    }
}
```

Problem with Stack as Array

1.Array have fixed size. Once the size of the array is decided it cannot be increased or decreased during the execution.

2. Array elements are always stored in contiguous memory locations.At times it might happen that enough contiguous location might not be available for the array that we are trying to create.

3. Operation like insertion of a new element in an array or deletion of existing element from the array are pretty tedious. This is because during insertion or deletion each element after the specified position has to be shifted one position to the right(in case of insertion) or left(in case of deletion.

Differentiate between stack and Array

Array	Stack
Any Element of an array can be accessed	Only the top most element can be accessed
An array essentially contains homogeneous elements	A stack may contain diverse element
An array is static data structure	A stack is a dynamic data structure i.e.its size shrinks and grows as element are pushed and poped
There is upper limit on the size of the array, which is specified during declaration	Logically a stack can grow to any size.

Stack as Linked List Or Dynamic implementation of Stack

Stack when implemented as array it suffers from some limitation of array that is size of array can not be increased or decreased once it is declared. This problem can be overcome if we implement a stack using a **linked list.** In case of a linked list we shall push or pop nodes from one end of linked list. Each node in the linked list contains the data and a pointer that gives location of the next node in the list.

```
class node
{
 <data type> data;
 node  link;
}
```

Where <data type> indicate that the data can be of any type like **int, float, char etc** and link, is a reference to next node in the list. The reference to the beginning of the list serves the purpose of the top of stack. .

```
import java.io.*;
class Node
{
 int data;
 Node next=null;
 Node(int j)
 {
 data=j;
 }
}


 class Stack
{

 private Node top;


 public Stack()
 {
 top=null;
 }

public void push(int j)
 {
  Node t = new Node(j)   ;
 if(top==null)
  {
   t.next=top;
   top=t;
  }
 else
 {
 t.next=top;
 top=t;

 }
 }

public int pop()
 {
 int data= top.data;
 top=top.next;
 return data;
 }
```

```
class StackLinkDemo
{

        public static void main(String[] args)
        {
   Stack ob=new Stack();
    for(int i=0;i<13;++i)
      ob.push(i);
        for(int i=0;i<13;++i)
     System.out.println(ob.pop());
    }
}
```

```
public int peek()
 {
  return top.data;
 }
public boolean isEmpty()
 {
  return (top==null);
 }
}
```

Explain the various operations that can be performed on the stack

CREATE:-Create a new Stack. This operation creates a new stack which is empty.

PUSH:-The process of adding a new element to the top of the stack is called Push operation .Pushing an element in the stack invokes adding of element ,as the new element will be inserted at the top after every push operation the top is incremented by one .In case the array is full and no new element can be accommodated ,it is called STACK FULL condition. This condition is called STACK OVERFLOW.

Algorithm

```
 void push( int data)
1       check tos
2.        if tos >= 9
3.         print: stack overflow
4         else
5             ++tos                      // increment tos;
 6     store data  AA[tos]=data          // store data
```

POP:-The process of deleting an element from the top of the stack is called Pop Operation. After every Pop operation the stack is decremented by one .If there is no element on the stack and the Pop is performed then this will result into STACK UNDERFLOW. Condition.

ALGORITHM

```
int pop()
 1     check tos
 2      if tos<=0
 3           print :stack underflow
 4           return 0
 5     else
 6          data= AA[tos]            //retrieve data
 7          - –tos                   //decrement tos
 8          return data
```

ISEMPTY: Check whether a stack is empty. This operation returns TRUE if the stack is empty and false otherwise. This is required for the pop operation because we cannot pop from an empty stack.

Peek: gives the data which is at the current location.

USE Of Stack

One common use of stack is to parse certain kind of text Strings ,Typically the strings are lines of code in a computer language,and program parsing them are compilers.
To give the flavor of what'is involved,we 'll show a program that checks the delimiters in a line of text typed by the user.The delimiters are the braces { and },brackets [and],and parentheses (and).Each opening or left delimiter should be matched by a closing or right delimeter.Every { should be followed by a matching } and so on.
Ex.
C[a] //correct
A{b[c]d}e //correct
A{b(c]d}e //not correct

```
import java.io.*;
class Stack
{
 private int MaxSize;
 private char stck[];
 private int top;


 public Stack(int a)
  {
   MaxSize =a;
   stck=new char[MaxSize];
   top=-1;
  }

public void push(char j)
  {
   ++top;
   stck[top]=j;
  }

public char pop()
```

```
 {
char data=stck[top];
  --top;
  return data;
 }
public char peek()
 {
  return stck[top];
 }
public boolean isEmpty()
 {
  return (top==-1);
 }
}


class BracketChecker
{
 private String input;


 public BracketChecker(String in)
 {
  input=in;
 }

 public void check()
  {
  int stacksize = input.length();
  Stack stck=new Stack(stacksize);
  for(int j=0;j<input.length();j++)
  {
  char ch=input.charAt(j);
   switch(ch)
    {

  case '{' :
  case '[' :
  case '(' :
          stck.push(ch);
```

```
          break;
  case '}' :
  case ']' :
  case ')' :
          if(!stck.isEmpty())
             {
              char chx =stck.pop();
              if ((ch=='}' && chx !='{') ||
                 (ch==']' && chx !='[') ||
                 (ch==')' && chx !='(') )
                 System.out.println("Error : "+ ch + "at"   +j);
                }
              else
               System.out.println("Error : "+ ch + "at"   +j);
             break;
   default : break;
   }
  }

 if(!stck.isEmpty())
     System.out.println("Error : Missing right delimiter");
}
}

class BracketApp
{
 public static void main(String a[])
 {
   String input;
   while(true)
           {
   System.out.println("Enter string containing delimiters");
    System.out.flush();
   input=getString();
  if( input.equals(""))
    break;
  BracketChecker checker= new BracketChecker(input);
  checker.check();
 }
```

```
 }
public static String getString()
{
        String s="";
        try
        {
        InputStreamReader isr= new InputStreamReader(System.in);
        BufferedReader br = new BufferedReader(isr);
        s= br.readLine();
        }catch (Exception e)
        {
        }
return s;
}
}
```

The program works by reading a character from the string one at a time and placing opening delimiters,when it finds them ,on a stack.When it reads a closing delimiter from the top of the stack and attempt to match it with the closing delimeter.If they are not the same type then error has occurred
Give input as

a{b(c[d]e)f}

Reversing a String

Stack can be used to reverse the string .A stack is used to reverse the letters.First the character are extracted one by one from the input string and pushed on to stack.Then they are popped off the stack and displayed.

```
import java.io.*;
class Stack
{
 private int MaxSize;
 private char stck[];
 private int top;


 public Stack(int a)
  {
```

```
   MaxSize =a;
   stck=new char[MaxSize];
   top=-1;
  }

public void push(char j)
  {
   ++top;
   stck[top]=j;
  }

public char pop()
 {
char data=stck[top];
  --top;
  return data;
 }
public char peek()
 {
  return stck[top];
 }
public boolean isEmpty()
 {
  return (top==-1);
 }
}


class Reverser
{
private String input;
private String output;

public Reverser(String in)
        {
```

```
        input=in;
        }
public String doRev()
        {
        int stacksize=input.length();
        Stack stck=new Stack(stacksize);
        for(int j=0; j<input.length();++j)
                {
                char ch= input.charAt(j);
                stck.push(ch);
                }
    output="";
         while (stck.isEmpty())
         {
                 char ch=stck.pop();
                 output =output+ch;
         }
         return output;
        }
}
class ReverseApp
{
 public static void main(String a[])
 {
   String input,output;
   while(true)
         {
   System.out.println("Enter string ");
    System.out.flush();
   input=getString();
  if( input.equals(""))
    break;
  Reverser reverse= new Reverser(input);
  output=reverse.doRev();
  System.out.println("Reversed :"+output);
```

```
  }

 }
public static String getString()
{
        String s="";
        try
        {
InputStreamReader isr= new InputStreamReader(System.in);
BufferedReader br = new BufferedReader(isr);
 s= br.readLine();
return s;
        }catch (Exception e)
        {
        }
return s;
}
}
```

Explain the application of Stack

Stacks are widely used in operating system, by compiler and by applications. Some of the application are

1. Subroutine calss, Recursion.
2. Interrupt handling.
3. Inter conversion between infix,postfix and prefix expression.
4. Matching parentheses in an expression.

Explain Polish notation

A polish mathematician Jan Lukasiewicz suggested a notation called POLISH notation ,which gives two alternatives to represent an arithmetic expression. These notation are **Prefix and Postfix.**

The fundamental property of polish notation is that the order in which the operation are to be performed is completely determined by the position of the operators and operands in the expression.

The arithmetic expressions are represented as

A+B*C

A*B-C

A+B/C-D

A^B+C

(Here ^ REPRESENT exponential)

The way we represent arithmetic expression is called **infix** notation. While evaluating infix notation we have to follow the following precedence

1 Highest priority:- Exponential (^)
2 Next highest priority:- Multi(*) and Division(/)
3 Lowest priority:-Addition(+) and Subtraction(-)

To override these priority we can do so by using a pair of parentheses

(A+B)*C
A*(B-C)
A+(B/C)-D
(A^B)+C

The expression within a pair of parentheses are evaluated first.

PREFIX notation

In prefix notation operator comes before the operands.
The expression in infix form is
A+B
The prefix notation is
+AB

POSTFIX notation

In postfix notation operator comes after the operands.
The expression in infix form is
A+B
The postfix notation is
AB+

The prefix and postfix expression have three feature:

- The operands maintain the same order as in the equivalent infix expression
- Parentheses are not needed to designate the expression
- While evaluating the expression the priority of the operator is irrelevant.

Converting infix expression to postfix form

Ex **(A+B)*C/D+E^F/G**

POSTFIX

STEP	ACTION TO DO	RESULT EXPRESSION
1	Solve parentheses first	(AB+)*C/D+(E ^ F)/G
2	Solve exponential	(AB+) * C / D + EF^ /G
3	Now we have to solve first * and / so start from left to right	AB+ C* / D + EF^ G/
4	Solve AB+C* /D	AB+C*D / + EF^G/
5	Solve operands on both side of +(FINAL)	AB+C*D/EF^G/+

PREFIX

STEP	ACTION TO DO	RESULT EXPRESSION

1	Solve parentheses first	(+AB)*C/D+(E ^ F)/G
2	Solve exponential	(+AB) * C / D + ^ EF /G
3	Now we have to solve first * and / so start from left to right	*+ABC / D + /^EFG
4	Solve AB+C* /D	/*+ABCD + /^ EFG
5	Solve operands on both side of +(FINAL)	+/*+ABCD+/^EFG

Convert following infix expression into postfix and prefix expression

1) (A-C)*B 2) (A+B)/(C-D) 3) (A+(B*C))/(C-(D*B)) 4) A+(B*C-(D/E^F)*G)*H
5) (A+B ^ D)/(E-F)+G)

PREFIX	POSTFIX
(A-C)*B	
-AC * B *-ACB	AC- *B AC-B*
(A+B)/(C-D)	
+AB /-CD /+AB-CD	AB+ / CD- AB+CD-/
(A+(B*C))/(C-(D*B))	
(A + *BC) / (C-(*DB)) (+A*BC) / (-C*DB) /+A*BC-C*DB	(A + BC*) / (C-(DB*)) (ABC*+) / (CDB*-) ABC *+CDB*-/
A+(B*C-(D/E^F)*G)*H	
A+(*BC-(D/^EF)*G)*H	A+(BC* - (D/EF^)*G)*H
A+(*BC – (/D^EF)*G)*H	A+(BC* – (DEF^/)*G)*H
A+(*BC - */D^EFG)*H	A+(BC* – DEF^/G*)*H
A+(-*BC*/D^EFG)*H	A+(BC* DEF^/G*-)*H
A+ *(-*BC*/D^EFGH)	A+ (BC*DEF^/G*-H*)
+A*-*BC*/D^EFGH	A BC*DEF^/G*-H*+

Convert following to postfix form

1) a+[{(b+c)+(d+e)*f}/g] ans=abc+de+f*+g/+
2) a *b+c ans =ab*c+
3) a+b/c-d ans= abc/+d-
4) (a+b)*c/d ans =ab+c*d/
5) (a+b)*c/d+e^f/g ans =ab+c*d/ef^g/+

Convert following to prefix form

1) (a*b)+c ans=+*abc
2) a/(b^c)+d ans=+/a^bcd
3) (a-b/c)*(d*e-f) ans=*-a/bc-*def

4) (a*b+(c/d))-f **ans=-+*ab/cdf**
5) a/b^c-d **ans -/a^bcd**

Explain conversion of infix to Postfix form using algorithm

The following expression transforms the infix expression Q into its equivalent postfix expression p. The algorithm uses a stack to temporarily hold operators and left parenthesis. The post fix expression will be constructed from left to right using the operands from Q and the operators which are removed from STACK. We begin by pushing a left parenthesis onto STACK and adding right parenthesis at the end of Q.

Suppose Q is an arithmetic expression written in infix from this algorithm finds the equivalent postfix expression P

1. Push "(onto stack and add")" to the end of Q
2. scan Q from left to right and repeat step 3 to 6 for each element of Q until stack is empty
3. if an operand is encountered ,add it to P
4. if left parenthesis is encountered, push it on to stack
5. if operand X is encountered, then
    - Repeatedly pop from the stack and add to P each operator(on the top of stack) which has the same precedence as or higher precedence than X
    - Add X to STACK
6. if a right parenthesis is encountered then
    - Repeatedly pop from STACK and add to P each operator(on the top of STACK) until a left parenthesis is found
    - Remove the left parenthesis(don't add left parenthesisto P)
7. EXIT

Lets conver the following expression into postfix using algorithm

A+(B*C-(D/E^F)*G)*H

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
A	+	(	B	*	C	-	(	D	/	E	^	F	)	*	G	)	*	H	)

As per the algorithm

Symbol Scanned	STACK	Expression p
1 A	(	A
2 +	(+	A
3 (	(+(	A
4 B	(+(	A B

5 *	(+(*	AB
6 C	(+(*	ABC
7 -	(+(-	ABC*
8 (	(+(-(	ABC*
9 D	(+(-(	ABC*D
10 /	(+(-(/	ABC*D
11 E	(+(-(/	ABC*DE
12 ^	(+(-(/^	ABC*DE
13 F	(+(-(/^	ABC*DEF
14)	(+(-	ABC*DEF^/
15 *	(+(-*	ABC*DEF^/
16 G	(+(-*	ABC*DEF^/G
17)	(+	ABC*DEF^/G*-
18 *	(+*	ABC*DEF^/G*-
19 H	(+*	ABC*DEF^/G*-H
20)	Empty	ABC*DEF^/G*-H*+

Algorithm to evaluate expression written in postfix form.

Suppose P is an arithmetic expression written in post fix form. The following algorithm ,which uses a STACK to hold operands, evaluate P

1.Add a right parenthesis “)” at the end of the P
2.Scan P from left to right and repeat step 3 and 4 for each element of P until the left parenthesis “)” is encountered.
3.If an operand is encountered,put it on STACK
4.If an Operator X is encountered
  a) Remove the two top elements of STACK,where A is the top element and B is next to top element
  b) evaluate B X A
  c) Place the result of b on to Stack
5.Set value equal to top element on Stack
6.Exit

Evaluate Expression a*(b+c)-d/e where a=5,b=6,c=2,d=12 and e=4.

The postfix equivalent of the expression is

abc+*de/-

after putting the values of variable in expression

5,6,2,+,*,12,4,/,-

(commas are used to separate element s so that 5,6,2 is not interpreted as 562)

Using algorithm

Step	Symbol Scaned	STACK

1	5	5
2	6	5,6
3	2	5,6,2
4	+	5,8
5	*	40
6	12	40,12
7	4	40,12,4
8	/	40,3
9	-	37
10	)	

Evaluate the following Expression.
1) 6,10,+,12,8,-,*,8,2,-2,^,+

Using algorithm of evaluation of postfix

Step	Symbol Scaned	STACK
1	6	6
2	10	6,10
3	+	16
4	12	16,12
5	8	8
6	-	16,4
7	*	64
8	8	64,8
9	2	64,8,2
10	-	64,6
11	2	64,6,2
12	^	64,36
13	+	100
14	)	

Convert following postfix to equivalent infix
1) A,B,+,C,D,-,*,E,F,-G,^,+

Using algorithm of evaluation of postfix

Step	Symbol Scaned	STACK
1	A	A
2	B	A,B
3	+	(A+B)
4	C	(A+B),C

5	D	(A+B),C,D
6	-	(A+B),(C-D)
7	*	(A+B),(C-D)
8	E	(A+B)*(C-D),E
9	F	(A+B)*(C-D),E,F
10	-	(A+B)*(C-D),E-F
11	G	(A+B)*(C-D),E-F,G
12	^	(A+B)*(C-D),E-F^G
13	+	(A+B)*(C-D) +E-F^G
14	)	

Program to conver infix to Post fix

```
import java.io.*;
class Stackx
{
private int maxsize;
private char stck[] ;
private int top;

Stackx(int s)
{
maxsize=s;
stck=new char[maxsize];
top=-1;

}
void push(char j)
{
++top;
stck[top]=j;
}

char pop()
{
char data =stck[top];
 --top;
return data;
}
public char peek()
{
return stck[top];
}
boolean isEmpty()
```

```
{
return (top==-1);

}
int size()
{
 return top+1;
 }
 char peekN(int n)
 {
 return stck[n];
 }
 void display(String s)
 {
System.out.println(s);
System.out.println("Atack bottom to top");
for(int i=0;i<size();++i)
{
 System.out.println(peekN(i));
 System.out.print(' ');
}
 System.out.println(" ");
}

};

class InToPost
{
private Stackx thestack;
private String input;
private String output="";

InToPost(String in)
{
input=in;
int stacksize=input.length();
thestack= new Stackx(stacksize);
}
String doTrans()
{
for(int j=0;j<input.length();++j)
{
char ch=input.charAt(j);
thestack.display("For "+ch+" ");
switch(ch)
{
```

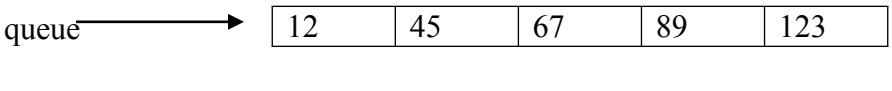
```
case '+':
case '-': gotOpera(ch,1);
          break;
case '*':
case '/': gotOpera(ch,2);
          break;
case '(':
         thestack.push(ch);
                      break;
case ')': gotparen(ch);
         break;
default : output=output+ch;
          break;
}
}
while(!thestack.isEmpty())
  {
  thestack.display("while");
  output=output+thestack.pop();
  }
thestack.display("END   ");
return output;
}
void gotOpera(char opthis,int perc1)
{
  while(! thestack.isEmpty())
  {
  char optop =thestack.pop();
  if( optop=='(')
     {
              thestack.push(optop);
                    break;
            }
   else
    {
             int perc2;
             if(optop=='+' || optop=='-')
               perc2=1;
       else
               perc2=2;
           if(perc2<perc1)
                     {
                     thestack.push(optop);
                     break;
                     }
     else
```

```
            output=output+optop;
            }
            }
            thestack.push(opthis);
 }


void gotparen(char ch)
{
while(!thestack.isEmpty())
{
char chx=thestack.pop();
if(chx=='(')
   break;
 else
   output=output+chx;
 }
 }
}
class   InfixApp
{
          public static void main(String[] args)
          {
  String input,output;
  while (true)
  {
  System.out.println("Enter infix");
  System.out.flush();
  input=getString();
  if(input.equals(""))
     break;
  InToPost theTrans=new InToPost(input);
  output=theTrans.doTrans();
  System.out.println("postfix is "+output);
  }
  }

  public static String getString()
{
          String s="";
          try
          {
InputStreamReader isr= new InputStreamReader(System.in);
BufferedReader br = new BufferedReader(isr);
 s= br.readLine();
return s;
```

```
        }catch (Exception e)
        {
        }
return s;
}
}
```

QUEUES

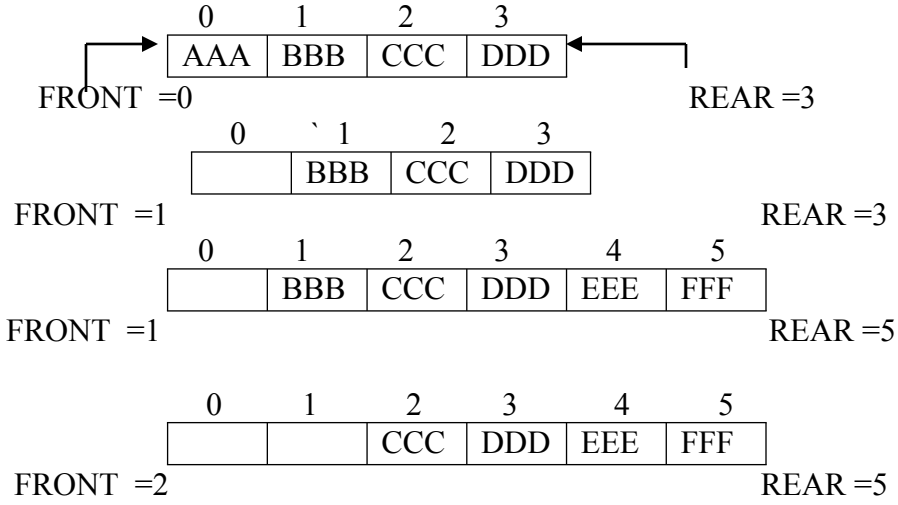
A Queue is a linear data structure of elements in which deletion can take place at one end called the **front** and insertion can take place only at the other end, called **rear**. Queues are also called **first in first out(FIFO)** lists .Since the first element in a queue will be the first element out of the queue. In other words the order in which elements enter a queue is the order in which they leave. This contrasts with stacks which are last in first out (LIFO).The following example illustrate the QUEUE.



In above case Front = 0 and Rear =4

Following fig shows a queue with 4 elements, where AAA is the front element and DDD is the rear element. Observe that the front and rear elements of the queue are the first and the last elements of queue. Suppose a element is deleted from the queue. The it must be from the front and must be AAA. This yield the queue now with the front element as BBB Now suppose EEE is added to the queue and then FFF is added to the queue. Then

they must be added at rear of the queue. Note that the FFF is now rear element. Now suppose that another element is deleted from the queue then it must be BBB



Queues may be represented in the computer in various ways, usually by means of one way lists or linear arrays, FRONT, containing the location of the front element of the queue and REAR, containing the location of the rear element of the queue. The condition FRONT=REAR=NULL* will indicate queue is empty.

(* Null in the case of Linked list ,in case of array -1)

Whenever the element is deleted from the queue, the value of the FRONT is increased by 1 this can be implemented by

FRONT=FRONT+1

Similarly whenever an element is added to the queue the value REAR is increased by ! This can be implemented by

REAR=REAR+1

This means that after N insertion the Rear element of the queue will occupy the last part of the array.This may occure when queue may not contain many items.

Program to implement ARRAY as QUEUE

```
class Queue
{
 int arr[];
 int front;
 int rear;

 Queue(int max)
        {
         front =-1;
         rear=-1;
         arr=new int[max];

        }
```

```
void addq ( int item  )
{
        if ( rear == arr.length - 1 )
        {
                System.out.println ( "\nQueue is full." ) ;
                return ;
        }

         rear++ ;
        arr[rear] = item ;

        if ( front == -1 )
                front = 0 ;
}
/* removes an element from the queue */
int delq ( )
{
        int data ;

        if ( front == -1 )
        {
        System.out.println ( "\nQueue is Empty." ) ;
                return -12000 ;
        }

        data = arr[front] ;
        arr[front] = 0 ;
        if ( front == rear )
                 front = rear = -1 ;
        else
                 front++ ;
        return  data ;
}

}
class   QueueDemo
{
        public static void main(String[] args)
        {
          int i;
        Queue q= new Queue(10);
        q.addq (  23) ;
        q.addq (  9) ;
        q.addq ( 11) ;
        q.addq ( -10) ;
        q.addq ( 25) ;
        q.addq ( 16) ;
        q.addq ( 17) ;
        q.addq ( 22) ;
        q.addq ( 19) ;
        q.addq ( 30) ;
        q.addq ( 32) ;

        i = q.delq ( ) ;
        System.out.println ( "\nItem deleted: "+ i ) ;
```

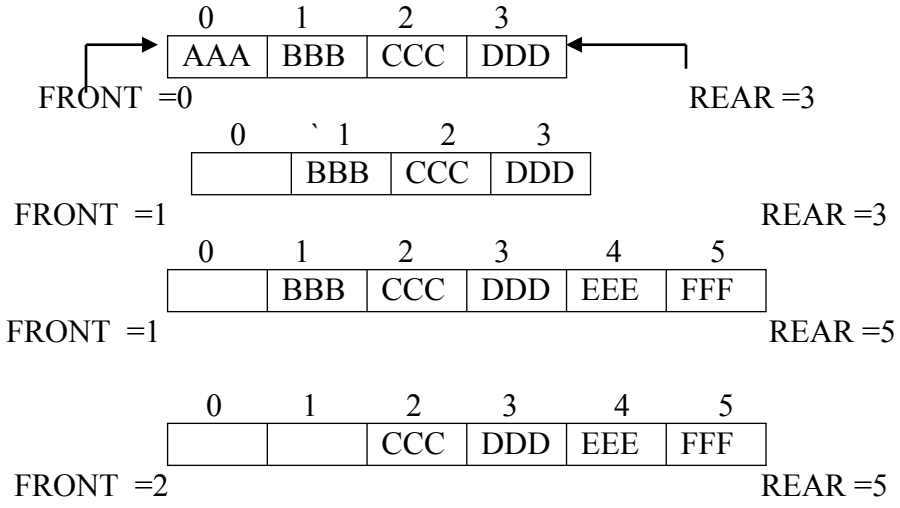
```
            i = q.delq ( ) ;
            System.out.println ( "\nItem deleted: "+ i ) ;

            i = q.delq ( ) ;
            System.out.println ( "\nItem deleted: "+ i ) ;

            }
}
```

CIRCULAR QUEUE

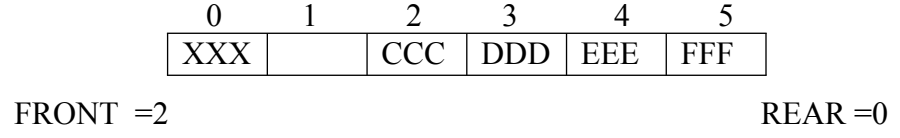
The queue that we implement using an array suffer from one limitation. In that implementation there is a possibility that the queue is reported as full(Since the rear has reached to the end of array),even though in actuality there might be empty slots at the beginning of the queue. To overcome this limitation we can implement the queue as circular queue.

In circular queue we go on adding the element to the queue and reach the end of the array, The next element is stored in the first slot of the array

Following fig shows a queue with 4 elements, where AAA is the front element and DDD is the rear element. Observe that the front and rear elements of the queue are the first and the last elements of queue. Suppose a element is deleted from the queue. The it must be from the front and must be AAA. This yield the queue now with the front element as BBB Now suppose EEE is added to the queue and then FFF is added to the queue. Then they must be added at rear of the queue. Note that the FFF is now rear element. Now suppose that another element is deleted from the queue then it must be BBB.



If any element (ex. XXX) in the queue is added then the queue doesnot have the space to accomdate the new element .So here the **REAR** point of the queue will be set to **0,** and element XXX will be now stored at location 0,makes queue as circular queue.



Algorithm for insertion of item into QUEUE

```
1    check for QUEUE whether it is full
      If FRONT =1 and REAR =N, or if FRONT=REAR+1 then
            Write OVERFLOW.
2.   If FRONT =NULL  then   ( Queue initially empty)
       Set FRONT =1 and REAR =1
     Else
       If  REAR= N then
            Set  REAR=1
      Else
             Set REAR =REAR+1
 3.         set QUEUE[REAR}=DATA
EXIT
```

Algorithm for Deletion of Data From the QUEUE.

```
1    check for QUEUE whether it is empty
      If FRONT = NULL    then
            Write UNDERFLOW.
 2    set ITEM =QUEUE [FRONT].
 3   Find new value of FRONT
     If FRONT =REAR  then   ( Queue has only one element to start)
       Set FRONT =NULL and REAR =NULL
     Else
       If  FRONT = N then
            Set  FRONT=1
      Else
             Set FRONT= FRONT+1
EXIT
```

PROGRAM THAT CREATES A CIRCULAR QUEUE

```
class Queue
{
 int arr[];
 int front;
 int rear;

 Queue(int max)
            {
             front =-1;
             rear=-1;
             arr=new int[max];

            }
void addq (  int item )
```

```
class  CircularQueue
{
            public static void main(String[] args)
            {
                        Queue q= new Queue(10);
                        int i;

            q.addq (  14 ) ;
            q.addq (  22 ) ;
            q.addq (  13 ) ;
            q.addq (  -6 ) ;
            q.addq ( 25 ) ;

            System.out.println ( "\nElements in the
```

```
{
        if ( ( rear == arr.length - 1 && front == 0 )
                        || (  rear + 1 == front ) )
        {
        System.out.println ( "\nQueue is full." ) ;
                return ;
        }

        if ( rear == arr.length - 1 )
                rear = 0 ;
        else
                rear++ ;

        arr[rear] = item ;

        if (front == -1 )
                front = 0 ;
}

/* removes an element from the queue */
int delq (  )
{
        int data ;

        if ( front == -1 )
        {
        System.out.println ( "\nQueue is empty." ) ;
                return -1200;
        }

        data = arr[front] ;
        arr[front] = 0 ;

        if (front == rear )
        {
                front = -1 ;
                rear = -1 ;
        }
        else
        {
                if ( front == arr.length - 1 )
                        front = 0 ;
                else
                        front ++ ;
        }
        return data ;
}

/* displays element in a queue */
void display ( )
{
        int i ;

        for ( i = 0 ; i < arr.length-1; i++ )
                System.out.println (  arr[i] ) ;
        System.out.println ( "\n" ) ;
circular queue: " ) ;
        q.display (  ) ;

        i = q.delq ( ) ;
        System.out.println ( "Item deleted: "+i ) ;

        i = q.delq ( ) ;
        System.out.println ( "\nItem deleted: "+ i )
;

System.out.println ( "\nElements in the circular
queue after deletion: " ) ;
        q.display ( ) ;

        q.addq (  21) ;
        q.addq ( 17 ) ;
        q.addq ( 18) ;
        q.addq (  9) ;
        q.addq (  20 ) ;

System.out.println ( "Elements in the circular queue
after addition: " ) ;
        q.display ( ) ;

        q.addq (  32 ) ;

System.out.println ( "Elements in the circular queue
after addition: " ) ;
        q.display (  ) ;



        }
}
```

```
}

}
```

Explain The operations on Queue

1.**Create :** Create a new queue. This operation create a empty Queue
2. **Add or Insert:** Adds an element to a Queue A new Element can be added to the Queue at the Rear
3.**Delete:** Remove an element from the queue. This operation removes the element which is at the front of the queue. This operation can only be performed if the queue is not empty. The result of an illegal attempt to remove the element from an empty queue is called underflow.
4.**IsEmpty:**check whether a queue is empty.This operation returns TRUE if the Queue is empty and FALSE otherwise.

Program that implements Queue as Linked List or dynamic implementation of QUEUE

```
class Node
{
  int data;
  Node next;
  Node(int j)
  {
  data=j;
   next=null;
  }
}
class Queue
{
        Node front ;
        Node rear ;
Queue()
{
front=null;
rear=null;
}
/* adds an element to the queue */
void insert (int j )
{
        Node temp= new Node( j );
        if ( temp == null )
                System.out.println( "\nQueue
is full." ) ;

        else
```

```
void display()
{
 Node temp = front;
 if(temp== null)
    System.out.println("Queue is empty");
else
{
System.out.println("Item in Queue");
 while(temp !=null)
  {
   System.out.println( temp.data +" ");
 temp=temp.next;
}
}
}
}

class QueueApp
{
 public static void main(String args[])
 {
 Queue q= new Queue();
  q.insert(10);
  q.insert(20);
  q.insert(30);
  q.insert(40);
  q.display();
```

```
          {
          temp.next= null;

           rear=temp;
          }
       if ( front == null )
           {
            rear =  front = temp ;
           }

}

/* removes an element from the queue */
int delete( )
{
          int item ;

          if ( front == null )
          {
                    System.out.println ( "\nQueue
is empty." ) ;
                    return -1 ;
          }

          item =  front .data ;
          front =front.next ;
          return item ;
}
```

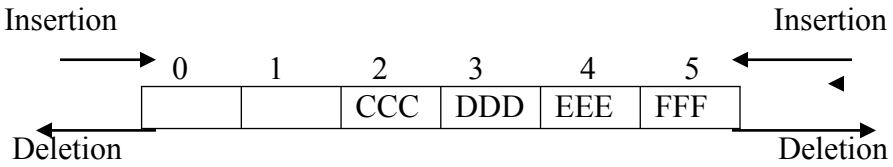
```
  q.delete();
  q.delete();
}
}
```

DEQUES

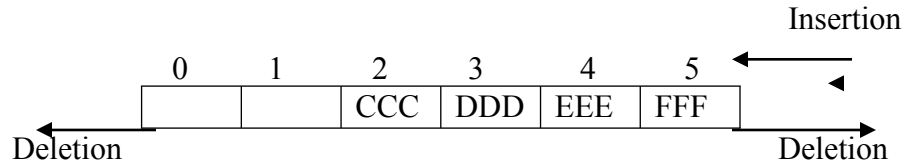
A Deque is a linear data structure in which elements can be added or removed at either end but not in the middle. The term deque is a contraction of the name double ended queue.

There are two variations of the deque. input restricted deque and an output restricted deque which are intermediate between a deque and a queue A input Restricted deque is a deque which allows insertion only at one end of the list but allows deletion at both the endsof the list, and output restricted deque is a deque which allows deletion at only one end of the list but allows insertion at both end of the list.

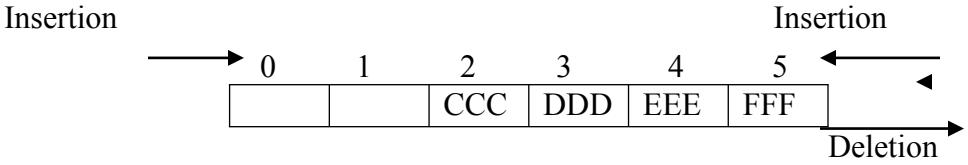
There are various ways of representing a deque in a computer. We will assume that the deque is maintained by a circular array with two pointers LEFT and RIGHT, which point to two end of the deque



DEQUEUE



Input Restricted DEQUEUE



Output Restricted DEQUEUE

Priority QUEUE

A priority QUEUE is a collection of elements where the elements are stored according to their priority levels.The order in which the elements should get added or removed is decided by the priority of the element. Following rules are applied to maintain a priority QUEUE.

1. The element with a higher priority is processed before any element of lower priority.
2. If there are elements with the same priority, then the elements added first in the queue would get processed.

Priority queues are used for implement ting job scheduling by the operating system where jobs with higher priorities are to be processed first.

Application of Queues

1. Round Robin technique for processor scheduling is implemented using QUEUE
2. All type of customer services center soft wares are designed using queue to store customers information
3. Printer service routines are designed using queues. A number of user shares the printer using printer server.

LINKED LIST

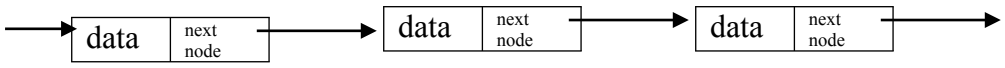
What do you mean by linked list

We are storing data into array but array have some drawback also which are listed below

1. Array has fixed size. Once the size of the array is decided it cannot be increased or Decreased during the execution.
2. Array elements are always stored in contiguous memory locations. At times it might happen that enough contiguous location might not be available for the array that we are trying to create.
3. Operation like insertion of a new element in an array or deletion of existing element from the array are pretty tedious. This is because during insertion or deletion each

element after the specified position has to be shifted one position to the right(in case of insertion) or left(in case of deletion.

Another way to store data into memory is to have each element in the list contain field called **link or pointer** which contain the address of the next element(data) into the list. Thus successive element in the list need not required to have adjacent memory like array. It make it easier to insert and delete element in the list. This type of data structure which contain the data and address of the next data is called linked list.
A linked list is a linear collection of data elements called node. Where the linear order is given by means of pointer. That is each node is divided into two parts. The first part contain **data** and the second part contain the **address of link** (address of other data).



Start

About A Node

Now it is clear that an element or node must have two fields

- A data Field
- A Link /Pointer

The data field may be simple integer, character, string or may be any record structure. We hereforeth use the convention of INFO to refer to data field.
The second field is link or pointer. This is a link to the next node in the list. Now depending on the type of info this pointer variable will be a pointer to that type.This we call as NEXT.
Thus we can represent node as

Info	Next

Lets consider we want to maintain the node that contain the name of student and roll no. of the student. Notice that here we are having the two field types,within the data field.Hence we need to have structure. The NEXT field is pointer to another node of the same type

So the node definition will be

```
Struct node
  {
    char name[20];
    int rollno;
    struct node * next;
}
```

And it can be represented as

Name	Roll no	Next

The declaration

struct node *next means next is a pointer to structure node.

Null Pointer: The link field of the last node contains **NULL** rather than a valid address.It is Null pointer and indicate the end of list

External Pointer: It is Pointer to the very first node in the link list,it enables us to access the entire linked list.

Empty List: If the nodes are not present in a linked list,then it is called an **Empty linked list**.It is also called the **null list**

Operations on Linked List

The basic operation to be performed on the link list are as follows.

1. Creation of linked list
2. Insertion of node into a linked list
3. Deletion of node from linked list.
4. Traversing(visiting each node) through linked list
5. Searching
6. concatenation of two linked list
7. Display of data of linked list.

Creation: This operation is used to insert a new node in the linked list at the specified position. A new node may be inserted.

- At the beginning of a linked list
- At the end of a linked list
- At the specified position in the linked list
- If the first list is empty,then the new node is inserted as a first node.

Deletion: This operation is used to delete an item from the linked list. A node may be deleted from the

- Beginning of a linked list
- End of linked list
- Specified position in the linked list

Traversing: It is a process of going through all the nodes of a linked list from one end to the other end.If we start traversing from the very first node towards the last node,it is called forward traversing.If the desired element is found we print "FOUND" else "NOT FOUND".

Concatenation:It is the process of appending (Joining) the second list to the end of the first list consisting of m nodes.When we concatenate two lists.the second list has n nodes then the concatenated list have (m + n) nodes.

Display: This operation is used to print each and every node's information.We access each node from the beginning of the list and output the data housed there.

Program to create a linked list

```
#include <stdio.h>
#include <conio.h>
#include <alloc.h>
```

```
/* adds a new node after the specified number of nodes */
void addafter ( struct node *q, int loc, int num )
{
```

```
/* structure containing a data part and next part */
struct node
{
        int data ;
        struct node * next ;
} ;

void append ( struct node **, int ) ;
void addatbeg ( struct node **, int ) ;
void addafter ( struct node *, int, int ) ;
void display ( struct node * ) ;
int count ( struct node * ) ;
void delete ( struct node **, int ) ;

void main( )
{
        struct node *p ;
        p = NULL ;  /* empty nexted list */
printf ( "\nNo. of elements in the nexted List = %d", count ( p ) )
;
        append ( &p, 14 ) ;
        append ( &p, 30 ) ;
        append ( &p, 25 ) ;
        append ( &p, 42 ) ;
        append ( &p, 17 ) ;

        display ( p ) ;

        addatbeg ( &p, 999 ) ;
        addatbeg ( &p, 888 ) ;
        addatbeg ( &p, 777 ) ;

        display ( p ) ;

        addafter ( p, 7, 0 ) ;
        addafter ( p, 2, 1 ) ;
        addafter ( p, 5, 99 ) ;

        display ( p ) ;
printf ( "\nNo. of elements in the nexted List = %d", count ( p ) )
;

        delete ( &p, 99 ) ;
        delete ( &p, 1 ) ;
        delete ( &p, 10 ) ;

        display ( p ) ;
printf ( "\nNo. of elements in the nexted List = %d", count ( p ) )
;
}



/* adds a node at the end of a nexted list */
void append ( struct node **q, int num )
{
        struct node *temp, *r ;

if ( *q == NULL ) /* if the list is empty, create first node */
        {
                temp = malloc ( sizeof ( struct node ) ) ;
                temp -> data = num ;
                temp -> next = NULL ;
                *q = temp ;
        }
        else
        {
                temp = *q ;
        struct node *temp, *r ;
        int i ;

        temp = q ;
        /* skip to desired portion */
        for ( i = 0 ; i < loc ; i++ )
        {
                temp = temp -> next ;

                /* if end of nexted list is encountered */
                if ( temp == NULL )
                {
                        printf ( "\nThere are less than
%d elements in list", loc ) ;
                        return ;
                }
        }

        /* insert new node */
        r = malloc ( sizeof ( struct node ) ) ;
        r -> data = num ;
        r -> next = temp -> next ;
        temp -> next = r ;
}

/* displays the contents of the nexted list */
void display ( struct node *q )
{
        printf ( "\n" ) ;

        /* traverse the entire nexted list */
        while ( q != NULL )
        {
                printf ( "%d ", q -> data ) ;
                q = q -> next ;
        }
}

/* counts the number of nodes present in the nexted list */
int count ( struct node * q )
{
        int c = 0 ;

        /* traverse the entire nexted list */
        while ( q != NULL )
        {
                q = q -> next ;
                c++ ;
        }

        return c ;
}

/* deletes the specified node from the nexted list */
void delete ( struct node **q, int num )
{
        struct node *old, *temp ;

        temp = *q ;

        while ( temp != NULL )
        {
                if ( temp -> data == num )
                {
/* if node to be deleted is the first node in the nexted list */
                        if ( temp == *q )
                                *q = temp -> next ;

/* deletes the intermediate nodes in the nexted list */
                        else
```

```
                        /* go to last node */
                        while ( temp -> next != NULL )
                                temp = temp -> next ;

                        /* add node at the end */
                        r = malloc ( sizeof ( struct node ) ) ;
                        r -> data = num ;
                        r -> next = NULL ;
                        temp -> next = r ;
                }
}

/* adds a new node at the beginning of the nexted list */
void addatbeg ( struct node **q, int num )
{
                struct node *temp ;

                /* add new node */
                temp = malloc ( sizeof ( struct node ) ) ;

                temp -> data = num ;
                temp -> next = *q ;
                *q = temp ;
}
```

```
                                old -> next = temp -> next ;

/* free the memory occupied by the node */
                                free ( temp ) ;
                                return ;
                        }

/* traverse the nexted list till the last node is reached */
                        else
                        {
                old = temp ;
 /* old points to the previous node */
                temp = temp -> next ;
 /* go to the next node */
                        }
                }

                printf ( "\nElement %d not found", num ) ;
```

Explaination of above program

To begin with we have defined a structure for a node. It contains a data(INFO) part and a link(next) part. The variable **p** has been declared as pointer to a node. We have used this pointer as pointer to the first node in the linked list. No matter how many nodes get added to the linked list **p** would continue to pointer to the first node in the list. When no node has been added to the list **p** has been set to **NULL** to indicate that the list is empty.
The **append()** function has to deal with two situations:

- The node is being added to an empty list.
- The node is being added at the end of an existing list.

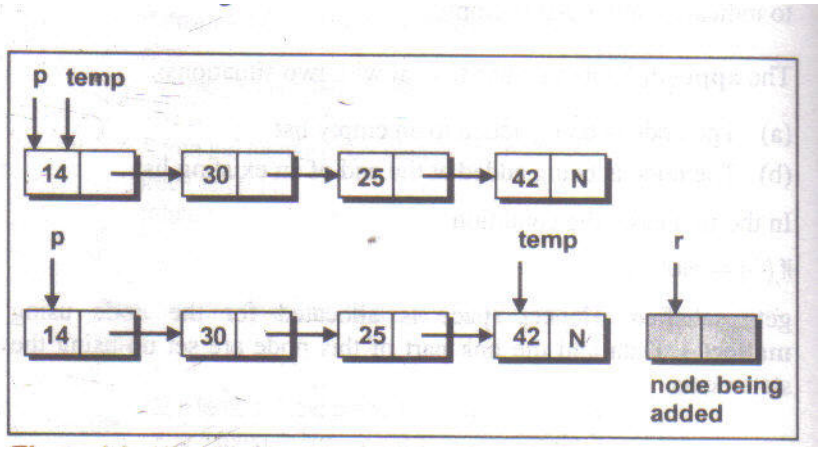
In the first case(*q==NULL)
gets satisfied Hence, space is allocated for the node using malloc().Data and next part of this node are set up using the statements

```
temp->data=num;
temp->next=NULL;
```

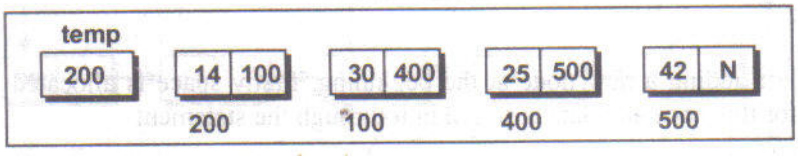
Lastly, **p** is made to point to this node, since the first node has been added to the list and **p** must always to the first node. Note that ***q** is nothing but equal to **p**
In the other case, when the linked list is not empty,the condition
If(*q==NULL)
Would fail since ***q** (i.e. **p** is non-NULL).Now temp is made to point to the first node in the list through the statement
Temp=*q;
Then using **temp** we have traversed through the entire linked list using the statements

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

The position of the pointers before and after traversing the linked list is as shown

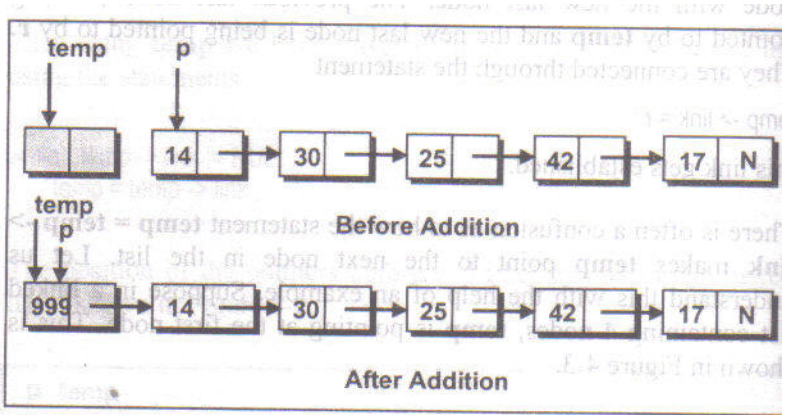


Each time through the loop the statement **temp=temp->next** makes **temp** point to next node in the list. When **temp** reaches the last node in the condition **temp->next!=NULL** would fail. Once outside the loop we allocate space for the new node through the statement
r= malloc(sizeof(struct node));
Once the space has been allocated for the new node its **data** part is stuffed with num and **next** part with **NULL** Note that this node is now going to be the last node in the list.
All that now remains to be done is connected the previous last node with the new last node. The previous last node is being pointed to by **temp** and the new last node is being pointed to by **r** They are connected through the statement
temp->next=r;
this link gets established.
There is often a confusion as to how the statement **temp=temp->next** makes **temp** points to the next node in the list .Let us understand this with the help of an example.Suppose in a linked list containing 4 nodes, **temp** is pointing at the first node.This is shown.



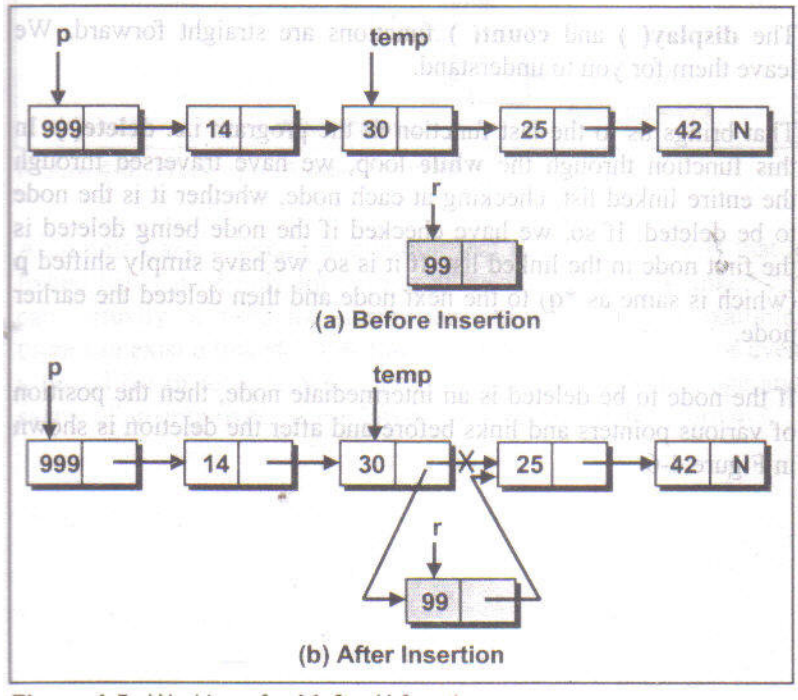
Instead of showing the links to the next node we have shown the addresses of the next node in the next part of each node.
When we execute the statement
Temp=temp->next;
The right hand side yield 1**00.**This address is now stored in **temp**.As a result. **Temp** starts pointing to the node present at address **100.**In effect the statement has shifted **temp** so that it has started pointing to the next node in the list.

Lets understand the **addbeg()** function.Suppose there are already 5 nodes in the list and we wish to add a new node at the beginning of this exesting linked list.This situation is shown as



For adding a new node at the beginning, firstly space is allocated for this node and data is stored in it through the statement
temp->data=num;
Now we need to make the **next** part of this node point to the existing first node.This has been achieved through the statement
temp->next=*q;
Lastly this new node must be made first node in the list.This has been attained through the statement.
*q=temp;

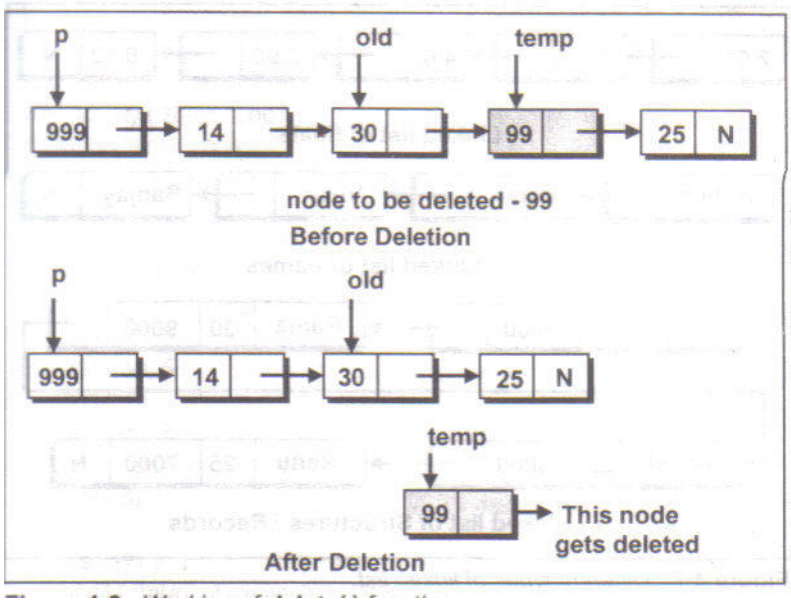
The addafter() function permits us to add a new node after a specified number of node in the linked list.
To begin with through a loop we skip the desired number of nodes after which a new node is to be added. Suppose we wish to add a new node containing data as 99 after the 3rd node in the list. The position of pointer once the control reaches outside the for loop is shown below. Now space is allocated for the node to be inserted and 99 is stored in the data part of it.



All that remains to be done is readjustment of link such that 99 goes in between 30 and 25.This is achieved through the statements
r->next = temp->next
temp->next =r

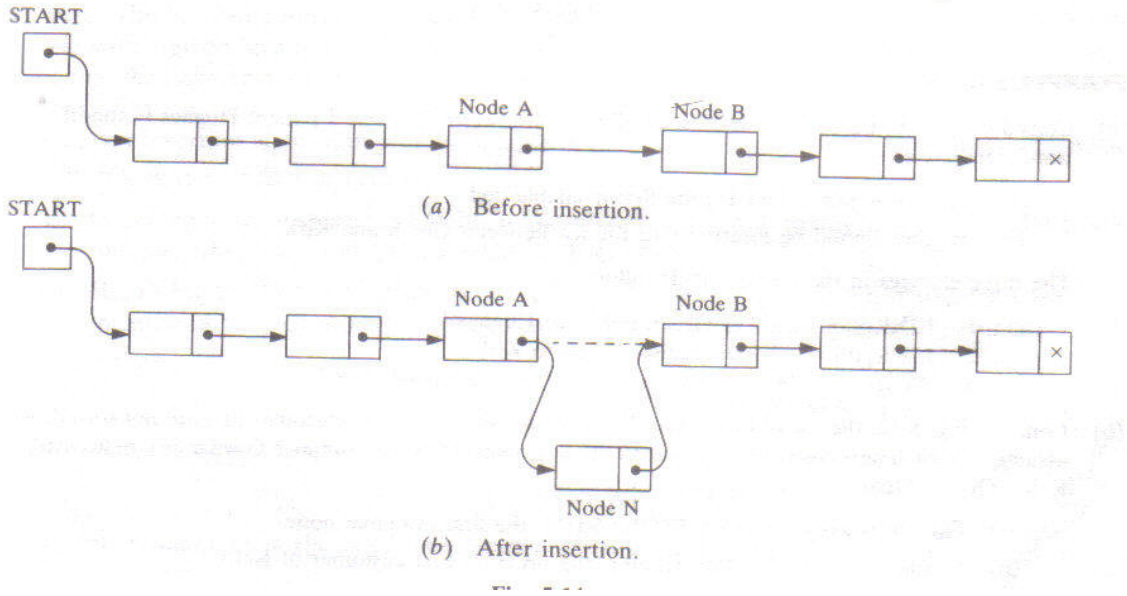
The first statement makes next part of node containing 99 to point to the node containing 25.The second statement ensures that the link part of node containing 30 points to the node containing 99.On execution of the second statement the earlier link between 30 and 25 is served. So now 30 no longer points to 25,it points to 99

This brings to last function delete().In this function through the while loop,we have traversed through the entire linked list,checking at each node,whether it is the node to be deleted.If so we have checked if the node being deleted is the first node in the linked list if it is so ,we have simply shifted p (which is *q) to the next node and then deleted the entire node



Algorithm for Inserting A node into Linked List

Let List be a linked list with successive nodes A and B as shown.Suppose a node N is to be inserted into list between node A and B



The above fig. shows that node A now points to the new node N,and Node N points to node B and so on.

Observe that three pointer field are changed as follows:

The nextpointer field of node A now points to the new node N

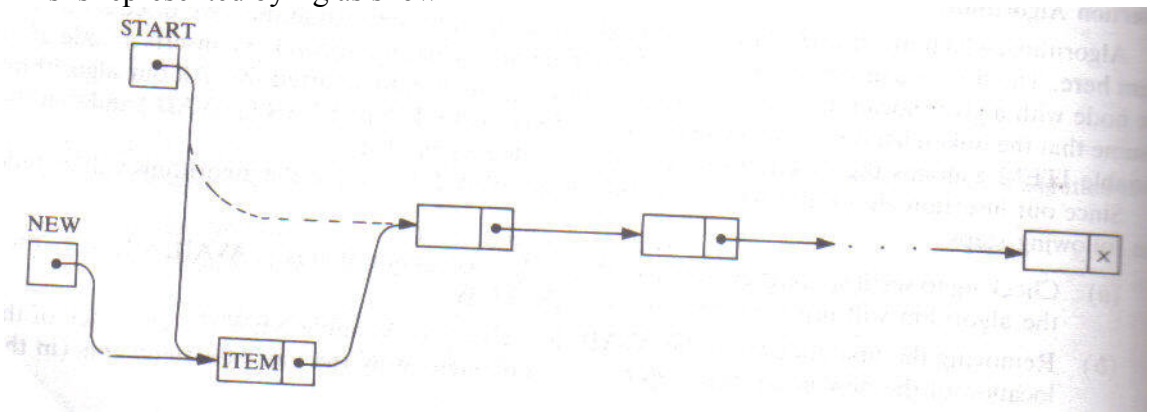
The node N now points to node B to which previously node A was pointing.

Algorithms which insert nodes into linked lists come up in various situations. We discuss two of them here. The first one inserts a node at the beginning of the list, the second one inserts a node after the node with a given location. All our algorithms assume that the linked list is in memory which is pointed by START pointer and that the variable ITEM contains the new information to be added to the list.

Algorithm for Inserting node at beginning

```
Assume that Start is pointer pointing to beginning of  list
 Check if (*start ==null)
          {
           *p= malloc( sizeof(struct node))   // Create new node
            p->next=*start;                  //store address of next node in next ofnew node
            p->data =data;                   // store data in  new  node
            *start=p;                         //store address of new node at start of list
          }
```

This is represented by fig as shown



Inserton of node at the end of list

Suppose we have been given some value which is to be stored after the node which contain some specific **data**.then we have to first travers through the link list and find out the node which contain data and then insert the node after that node.

Algorithm for Inserting node after specific node

Assume that Start is pointer pointing to beginning of list. and data to be inserted after the node which have value INFO.

```
        Struct node *p =start; //store start of list in p pointer
While(*p->data !=INFO)      //traverse the list till the INFO is found in List
```

```
            p =p->next

 Check if (*p ==null)

             print Message "data not found in list"
    otherwise

            *r= malloc( sizeof(struct node))   // Create new node
              r->next=p->next;                 //store address of next node in next of new node
              r->data =data;                   // store data in  new  node
              p->next=r;                        //store address of new node in list
```

Inserton of node at the end of list

Suppose we have been given some value which is to be stored at the end of list then we have to first travers through the link list till we reach at the end of list and then insert the node at the end of list.

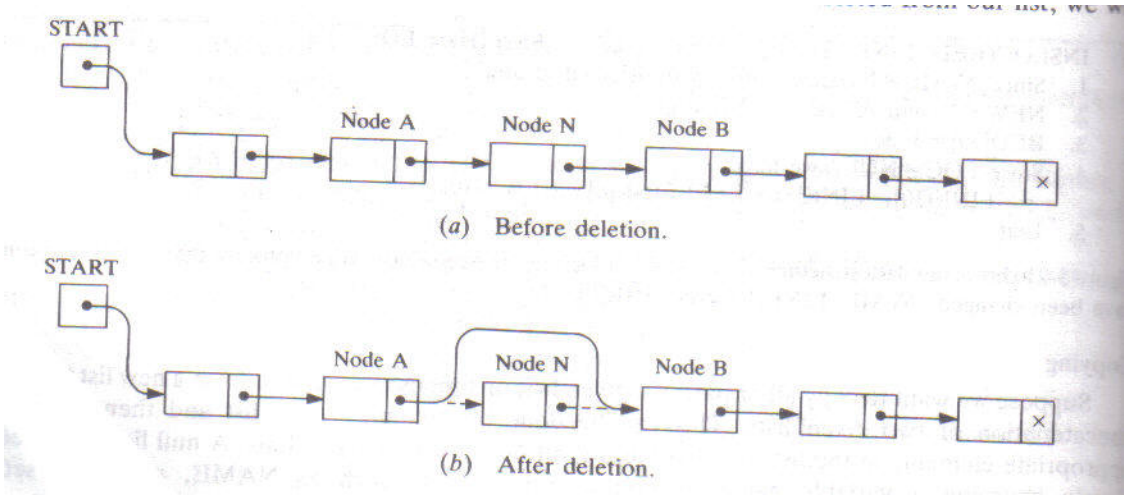
Algorithm for Inserting node at end of node

Assume that Start is pointer pointing to beginning of list.

```
        Struct node *p =start; //store start of list in p pointer
While(*p->next!=NULL)      //traverse the list till the END of  List
            p =p->next

 Check if (* p==null)
            {

             *r= malloc( sizeof(struct node))   // Create new node
              r->next=p->next;                 //store address of next node in next of new node
              r->data =data;                   // store data in  new  node
              p->next=r;                        //store address of new node in list
            }
```

Deletion from Linked List

Let LIST be a Linked List with a node N between A and B as shown.Suppose node N is to be deleted from the linked list.The deletion occurs as soon as the next pointer field of node A is changed so that it points to node B



One Should take care of the thing that while we are deleting node from the list containing INFO we should take care node Before and After the node.In case of above figure node A and node B.The address of node B has to be stored in the next pointer of node A.So while deleting node N we should also have with us the node B.

Algorithm for deleting node

Assume that Start is pointer pointing to beginning of list.suppose we want to delete the node containing data INFO from the list

```
        Struct node *p =start; //store start of list in p pointer
        Struct node old          // location of previous node
While(*p->data !=INFO)     //traverse the list till the END of  List
           old = p;               //store address of previous node
           p =p->next           // point to next node

 if node is found
             old->next=p->next   //assign the address of next node to next of old node
              free p                  // delete p;

  if node is not found
           print message No node found
```

Advantage of Linked List

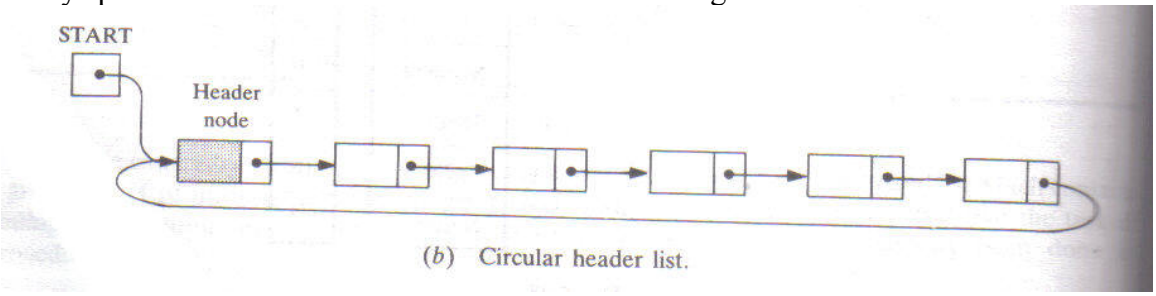
- Linked list are dynamic data structure, they can grow or shrink during the execution.
- Efficient memory utilization :Here memory is not defined.Memory is allocated whenever it is required.And it is de allocated when it is no longer needed.
- Insertion and deletion are easier and efficient.
- Can be used with many complex application.

Disadvantages

- More memory required if the number of fields are more,then more memory space is needed.
- Access to an arbitarary data item is little bit cumbersome and also time consuming

Circular Linked List

A circular linked list is a linked list where the last node points back to the header node.When a circular linked list is used it is essentially same as that of the linked list except that it doesnot ends with the NULL pointer rather the next pointer of last node always points to the address of first node.as shown in fig



(b) Circular header list.

The circular linked list are frequently used instead of ordinary linked list because many operations are much easierto state and implement using header lists.This comes from the following two properties of circular lists.

1) The null pointer is not used,and hence all pointer contain valid addresss.

2) Every node has a predecessor,so the first node may not require a special case.

Two way Linked List(Doubly Linked List)

All the linked list discussed above is called one way linked list , since there is only one way that the list can be traversed.That is beginning with the list pointer variable START which points to the first node and using the nextpointer field **next** to point to the next node in the list,we can traverse the list in only one direction.One have access to the immediate node but doesnot have access to the preceding node

This section introduces a new list structure, called a two-way list, which can be traversed in two directions: in the usual forward direction from the beginning of the list to the end, or in the backward direction from the end of the list to the beginning. Furthermore, given the location LOC of a node N the list, one now has immediate access to both the next node and the preceding node in the list. This means, in particular, that one is able to delete N from the list without traversing any part of the list.

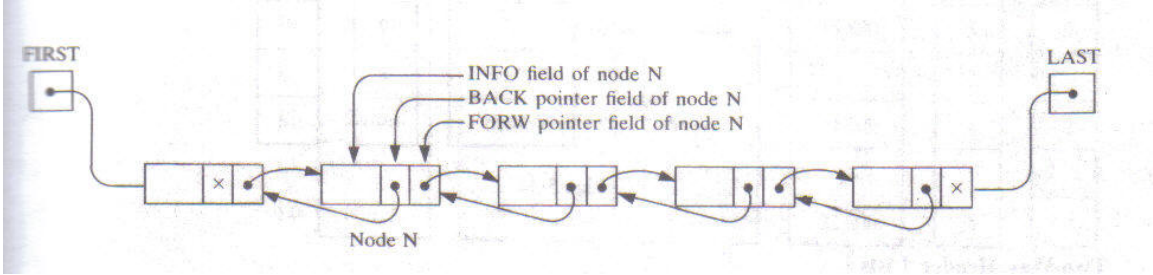
A *two-way list* is a linear collection of data elements, called *nodes,* where each node N is divided into three parts:

(1) An information field INFO which contains the data of N

(2) A pointer field FORW which contains the location of the next node in the list

(3) A pointer field BACK which contains the location of the preceding node in the list

The list also requires two list pointer variables: FIRST, which points to the first node in the list, and LAST, which points to the last node in the list. Figure contains a schematic diagram of such a list. Observe that the null pointer appears in the FORW field of the last node in the list and also in the BACK field of the first node in the list.



Observe that, using the variable FIRST and the pointer field FORW, we can traverse a two-way list in the forward direction as before. On the other hand, using the variable LAST and the pointer field BACK, we can also traverse the list in the backward direction.