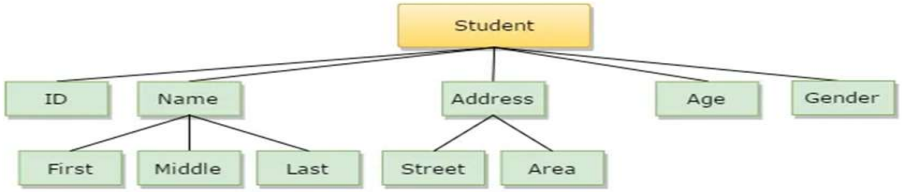
Module 1

Introduction to Data Structures

- **Data**
  - Data are simply values or set of values.
  - Data is simply a collection of facts and figures, or we can say that data is a set of values or values in a particular format that refers to a single set of item values.
  - Data item can be classified as,
    - Elementary item- We cannot subdivide it.
    - Group item- Data items that are subdivided into sub items.
- Consider the following figure



- Example for data,
  - Ram, Anu, 21,19, M, F,.....
- **Information**
  - Information is the result of analyzing and interpreting pieces of data.
  - Above data are the details of students name age and gender.

Data structures

- In computer terms, a data structure is a specific way to store and organize data in a computer's memory so that these data can be used efficiently later.
- The logical or mathematical model for a particular organization of data is termed as a data structure.
- A **data structure** is a specialized format for organizing, processing, retrieving and storing **data**.
- Each **data structure** contains **information** about the **data** values, relationships between the **data** and functions that can be applied to the **data**.

Definition

- DS is a representation or way of organizing all data items that considers not only elements stored but also their relationship to each other.
- It affects the design of both structural and functional aspects of a program.

Program= Algorithm+ Data structure.

Algorithm

- An algorithm is a procedure or step-by-step instruction for solving a problem.
- They form the foundation of writing a program.
- To develop a program of an algorithm we should select an appropriate data structure for that algorithm.

- A typical programming task can be divided into two phases:
- ***Problem solving phase***
  - produce an ordered sequence of steps that describe solution of problem
  - this sequence of steps is called an ***algorithm***
- ***Implementation phase***
  - implement the program in some programming language

Steps in Problem Solving

- First produce a general algorithm (one can use ***pseudocode***)
- Refine the algorithm successively to get step by step detailed ***algorithm*** that is very close to a computer language.
- ***Pseudocode*** is an informal language that helps programmers develop algorithms.
- Pseudocode is very similar to everyday English.

Algorithm for add 2 numbers

Step 1: Start
Step 2: Declare variable number1, number2 and sum
Step 3: Read Variable number1, number2
Step 4: Perform operation sum= number1+ number2
Step 5: Print sum
Step 7: Stop

- **Algorithm to print the larger of two numbers**

Step 1: Start.
Step 2: Declare variables a, b
Step 3: Read a, b
Step 4: If a>b then /*Checking */
Display "a is the largest number".
otherwise.
Display "b is the largest number"
Step 5: Stop

- **Algorithm to print the larger of three numbers**

1. Start
2. Read A,B,C
3. If (A>B) and (A>C) then
print "A is greater".
Else if (B>A) and (B>C) then
print "B is greater".
Else print "C is greater".
4. Stop

- **Print first n natural numbers**

1: Start
2: Read n
3: Set i=1
4: Repeat steps 5 to 6 until i<=n
5: print i
6: i=i+1
7: stop

```
Repeat
    {
      print i
      i=i+1
    }
Until (i<=n)
```

- **Print first n natural numbers**

1: Start
2: Read n
3: for i=1 to n do
{
print i
i=i+1
}
4: stop

- **Print first n natural numbers**

1: Start
2: Read n
3: Set i=1
4: while (i<=n) do
{
print i
i=i+1
}
5: stop

Pseudocode for looping statements

- Comments begin with // or /* and */
- Blocks are indicated with matching braces { and }
- Assignment of values

variable = expression
a= 5

- Looping statements : while, for, repeat – until

while (condition) **do**
{
statement 1
statement 2
etc
}

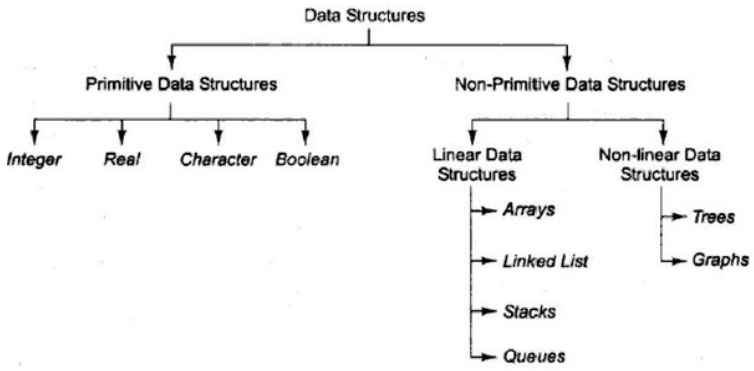
- **for** variable = value1 **to** value2 do

{
statement 1
statement 2 etc
}

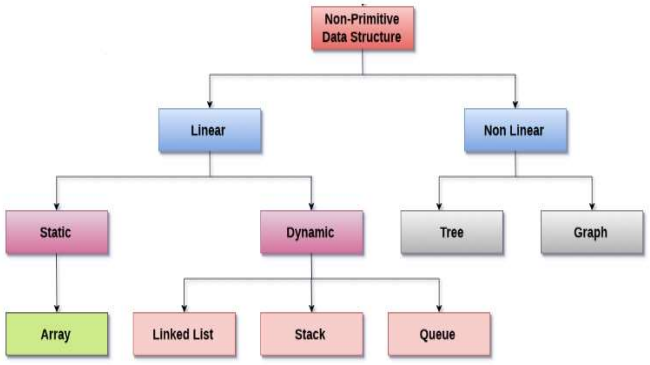
- **repeat**

{ statement 1
statement 2
} **until** (condition)

Types of Data structures



- Primitive Data Structure
  - Basic data types such as integer, real, character etc are known as primitive DS.
  - Primitive data types are predefined types of data, which are supported by the programming language.
  - For example, integer, character are all primitive data types.
- Non-primitive data structures
  - The non-primitive data types are defined by the programmer.
  - Non-primitive data types are not defined by the programming language, but are instead created by the programmer using the primitive types.
  - Emphasize on structuring of a group of homogeneous or heterogeneous (different types) data items.
- Based on the structure and arrangement of data, non-primitive DS are further classified as
  - Linear DS
  - Non Linear DS



- **Linear DS**
  - In linear data structures, the elements are arranged in sequence one after the other.
  - Since elements are arranged in particular order, they are easy to implement.

- Data structure in which data elements are arranged sequentially or linearly, where each element is attached to its previous and next adjacent elements, is called a linear data structure.
- *Examples of linear data structures are array, stack, queue, linked list, etc.*

- **Static data structure:**
  - Static data structure has a fixed memory size.
  - It is easier to access the elements in a static data structure.
  - *An example of this data structure is an array.*
- **Dynamic data structure:**
  - In dynamic data structure, the size is not fixed.
  - It can be randomly updated during the runtime which may be considered efficient concerning the memory (space) complexity of the code.
  - *Examples of this data structure are queue, stack, linked list etc.*

- **Non Linear DS**
  - Data structures where data elements are not arranged sequentially or linearly are called **non-linear data structures.**
  - Non-linear data structures are not easy to implement in comparison to linear data structure.
  - It utilizes computer memory efficiently in comparison to a linear data structure.
  - *Examples of non-linear data structures are trees and graphs.*

Data structure Operations.

There are different types of operations that can be performed for the manipulation of data in every data structure.

- **Traversing:**
  - Traversing a Data Structure means to visit the element stored in it.
  - This can be done with any type of DS.
- **Searching :**
  - Searching means to find a particular element in the given data-structure.
  - It is considered as successful when the required element is found.
- **Insertion:**
  - It is the operation which we apply on all the data-structures.
  - Insertion means to add an element in the given data structure.
  - The operation of insertion is successful when the required element is added to the required data-structure.
  - It is unsuccessful in some cases when the size of the data structure is full and when there is no space in the data-structure to add any additional element.
- **Deletion:**
  - It is the operation which we apply on all the data-structures. Deletion means to delete an element in the given data structure.
  - The operation of deletion is successful when the required element is deleted from the data structure.

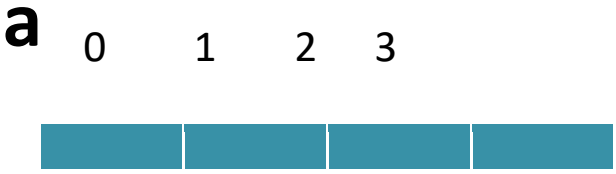
- ***Sorting* :**
  - Rearranging the elements in some type of order(e.g Increasing or Decreasing)
- **Merging :**
  - Combining two similar **data** structures into one

Linear Data Structure: Array

- The simplest type of data structure is a linear array.
- Array is a container which can hold a fixed number of items and these items should be of the same type.
  - Element − Each item stored in an array is called an element.
  - Index − Each location of an element in an array has a numerical index, which is used to identify the element.
- Array Representation in C

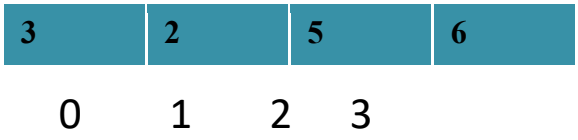
  datatype array-name[size];

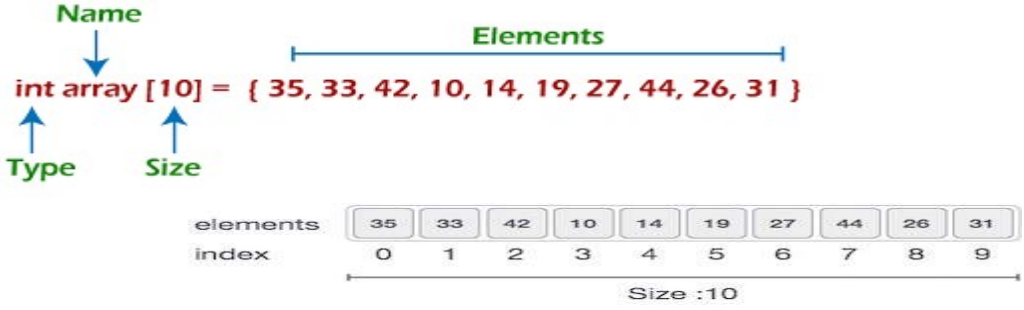
  eg; int a[4];



Or

int a[]={3,2,5,6};





- Index starts with 0.
- Array length is 10 which means it can store 10 elements.
- Each element can be accessed via its index.
- For example, we can fetch an element at index 6 as array[6]
- Number of elements that can be stored in an array is given by

  (upper bound-lower bound)+1

- That is, (9-0)+1=10

- **Properties of array**
  - Each element in an array is of the same data type and carries the same size.
  - Elements in the array are stored at contiguous memory locations from which the first element is stored at the smallest memory location.
  - Elements of the array can be randomly accessed since we can calculate the address of each element of the array with the given base address and the size of the data element.
- This is the static array- before using the array we have to declare the size of the array.
- Array can always be read or print through loop.

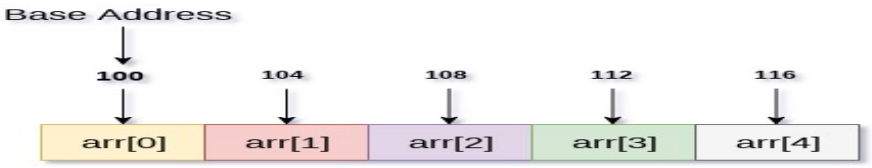
Read:

```
for(int i=0;i<10;i++)
{
Scanf("%d",&a[i]);
}
```

Write :

```
for(int i=0;i<10;i++)
{
printf("%d", a[i]);
}
```

- **Memory Allocation of the array**
  - All the data elements of an array are stored at contiguous locations in the main memory.
  - The name of the array represents the base address or the address of first element in the main memory.
- **address of element arr[i] = base address + size * (i - first index)**



Address of element arr[2]=100+4*(2-0)= 108

Basic Operations

Following are the basic operations supported by an array.

- **Traverse** − visit or print all the array elements one by one.
- **Insertion** − Adds an element at the given index.
- **Deletion** − Deletes an element at the given index.
- **Search** − Searches an element using the given index or by the value.
- **Sort-**Rearranging the elements in some type of order(e.g Increasing or Decreasing)
- **Merging-** Combining two similar **data** structures into one

1. Traverse Operation

- It is an operation in which each element of the array is visited.
- Traversal proceeds from first element to the last element of the array.

Algorithm:- (TRAVERSING AN ARRAY) TRAVERSE(ARR, LB, UB, N):

Here ARR is a linear array with LB is the lower bound and UB upper bound. PROCESS operation to the array(ARR). It is assumed that the array is NOT full and N points to the index number of last element in the array.

1. Set i = LB, N=UB
2. while (i <= N) do steps 3 to 4
3. Apply PROCESS to ARR[I]
4. i=i+1
5. Exit

2. Insertion

- Let A be a collection of data elements in the memory of the computer.
  - Inserting refers to the operation of adding another element to the array.
  - Inserting an element at the end of the linear array can be easily done provided the memory space allocated for the array is large enough to accommodate the additional element.
  - On the other hand, suppose we need to insert an element in the middle of the array.
  - Then, on the average, half of the elements must be moved downward to new location to accommodate the new elements and keep the order of the other elements.
  - The following algorithm inserts a data element ITEM in to the Kth position in the linear array LA with N elements.

Algorithm : (Inserting into Linear Array) INSERT (LA, N, K, ITEM,SIZE)

Here LA is a linear array with N elements and K is a positive integer such that K<=N. The algorithm inserts an element ITEM into the Kth position in LA. SIZE is the total size of the array.

1. if SIZE==N
   Print 'Not Possible to insert OVERFLOW'
   Goto step 8
2. Set J = N-1. [Initialize counter]
3. Repeat Steps 4 and 5 until j >= k
4. Set LA [J + 1] =LA [J]. [Move jth element downward.]
5. Set J = J-1[Decrease counter]
   [End of step 3 loop]
6. Set LA [K]=ITEM. [Insert element]
7. Set N=N+1 [Reset N]
8. EXIT.

Insertion − Adds an element at the given Index

Array size 10 and No: of elements is 6

Insert element 8 at index 3

After insertion the array becomes

No: of elements:7

3. Deleting an element from the array

- Deleting refers to the operation of removing one element from an array.
- Deleting an element at the end of the array presents no difficulties, but deleting the element somewhere in the middle of the array requires that each subsequent element be moved one location upward in order to fill up the array
- The following algorithm deletes the Kth element from a linear array LA and assigns it to a variable ITEM.

Algorithm for Deletion: (Deletion from a Linear Array) DELETE (LA, N, K, ITEM)

Here LA is a Linear Array with N elements and k is the positive integer such that k<=N. This algorithm deletes the kth element from LA.

1. Set ITEM = LA [k].
2. Repeat for J = k to N – 1.
   {
   Set LA [J] = LA [J +1]. [Move J + 1st element upward]
   }
   [End of loop]
3. Set N = N-1 [Reset the number N of elements in LA]
4. EXIT

- **Deletion** − Deletes an element at the given index.

Delete the element at index 4.

No: of elements of the array is reduced to 5

4. Searching

- The process of finding a particular item in the large amount of data is called searching.
- Two important searching techniques are
  - Linear search or sequential search
  - Binary search
- The searching operation is successful if the element is found in the data structure.

a) Linear Search

- This is simple searching technique.
- Here we search for a particular ITEM in a sequential manner.
- The algorithm for linear search is

Algorithm: LINEAR(LA, N, ITEM)

Hear LA is a linear array with N Elements and ITEM is an element to be searched

1. Set i=0, f=0
2. Repeat step 3 while i<N
3. If (*ITEM==LA[i])*
   Set f=1
   break
   else
   Set i=i+1
4. If (f==1)
   Print "Search is successful and Element found at index", i
   else
   Print "Not found''
5. Exit

b) Binary Search

- ***Binary Search*** *is a searching algorithm used in a sorted array .*
- Binary search looks for a particular item by comparing the middle most item of the array.
- If a match occurs, then the index of item is returned.
- If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item.
- Otherwise, the item is searched for in the sub-array to the right of the middle item.
- This process continues on the sub-array as well until the size of the sub array reduces to zero.

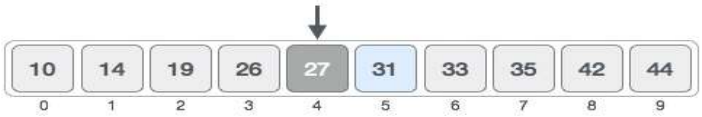
Algorithm: Binary_Search(LA, LB, UB, ITEM)

Here LA is the given array, LB is the 'lower_bund' that is the index of the first array element, UB is the 'upper_bound' that is the index of the last array element, ITEM' is the value to search .

```
1: Set  pos = - 1
2:Repeat steps 3 and 4 while LB <=UB
3: set mid = (LB+ UB)/2
4:   if (LA[mid] == ITEM)
          set pos = mid
          print pos
          go to step 5
 else if (LA[mid] > ITEM)
          set UB = mid - 1
else
          set LB = mid + 1
[end of if]
        [end of loop]
5: if pos = -1
          print "ITEM  is not present in the array"
           else
          print " ITEM is present at position, pos"

         [end of if]
 6: exit
```

- **Example**- The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.

- At first we shall to determine the middle of the array using the formula
  Mid=(LB+UB)/2
- Here it is, (0 + 9) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.



- Now we compare the value stored at location 4, with the value being searched, i.e. 31.
- We find that the value at location 4 is 27, which is not a match.
- As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.

- Change our LB to mid + 1 and find the new mid value again.
  LB = mid + 1 =4+1=5
  mid =(LB + UB) / 2 =(5+9)/2=7
- Our new mid is 7 now.
- We compare the value stored at location 7 with our target value 31.

- The value stored at location 7 is not a match, rather it is more than what we are looking for.
- So, the value must be in the lower part from this location.
- So, LB=5 and UB=mid-1=7-1=6

- Hence, we calculate the mid again.
- mid=(LB+UB)/2=(5+6)/2=5.
- We compare the value stored at location 5 with our target value.
- We find that it is a match**.**

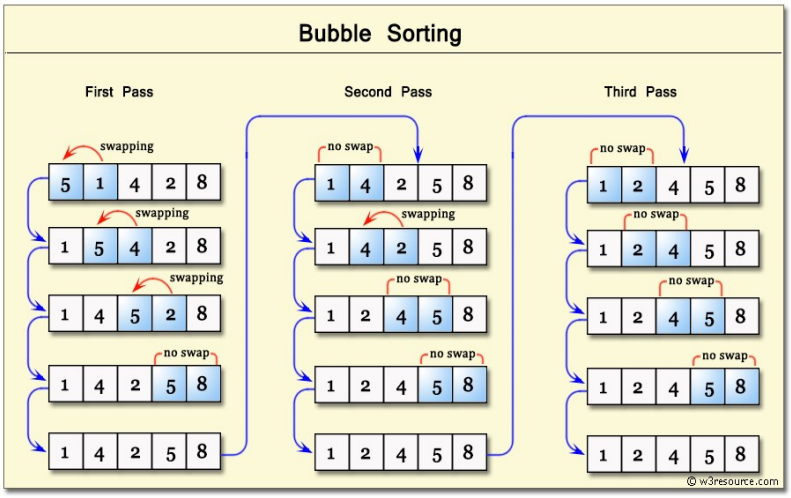
We conclude that the target value 31 is stored at location 5.

5. Sorting

- ☐ Rearranging the elements in some type of order(e.g Increasing or Decreasing)
- ☐ Iterative sorting algorithms (comparison based)
  - ◘ Selection Sort
  - ◘ Bubble Sort
  - ◘ Insertion Sort
- ☐ Recursive sorting algorithms (comparison based)
  - ◘ Merge Sort
  - ◘ Quick Sort
- ☐ Radix sort (non-comparison based)

Bubble Sort

- ☐ **Bubble Sort** is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order.
- ☐ This algorithm is not suitable for large data sets.
- ☐ Large item is like “bubble” that floats to the end of the array



Input: *arr[] = {5, 1, 4, 2, 8}*

First Pass:

Bubble sort starts with very first two elements, comparing them to check which one is greater.

*(**5 1** 4 2 8) –> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.*

*(1 **5 4** 2 8) –> (1 **4 5** 2 8), Swap since 5 > 4*

*(1 4 **5 2** 8) –> (1 4 **2 5** 8), Swap since 5 > 2*

*(1 4 2 **5 8**) –> (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.*

Second Pass:

Now, during second iteration it should look like this:

*(**1 4** 2 5 8) –> (**1 4** 2 5 8)*

*(1 **4 2** 5 8) –> (1 **2 4** 5 8), Swap since 4 > 2*

*(1 2 **4 5** 8) –> (1 2 **4 5** 8)*

*(1 2 4 **5 8**) –> (1 2 4 **5 8**)*

Third Pass:

Now, the array is already sorted, but our algorithm does not know if it is completed.

*The algorithm needs one **whole** pass without **any** swap to know it is sorted.*

*(**1 2** 4 5 8) –> (**1 2** 4 5 8)*

*(1 **2 4** 5 8) –> (1 **2 4** 5 8)*

*(1 2 **4 5** 8) –> (1 2 **4 5** 8)*

*(1 2 4 **5 8**) –> (1 2 4 **5 8**)*

		0	1	2	3	4	5	6	7
i = 0	j								
	0	5	3	1	9	8	2	4	7
	1	3	5	1	9	8	2	4	7
	2	3	1	5	9	8	2	4	7
	3	3	1	5	9	8	2	4	7
	4	3	1	5	8	9	2	4	7
	5	3	1	5	8	2	9	4	7
	6	3	1	5	8	2	4	9	7
i=1	0	3	1	5	8	2	4	7	9
	1	1	3	5	8	2	4	7	
	2	1	3	5	8	2	4	7	
	3	1	3	5	8	2	4	7	
	4	1	3	5	2	8	4	7	
	5	1	3	5	2	4	8	7	
i=2	0	1	3	5	2	4	7	8	
	1	1	3	5	2	4	7		
	2	1	3	5	2	4	7		
	3	1	3	2	5	4	7		
	4	1	3	2	4	5	7		
i=3	0	1	3	2	4	5	7		
	1	1	3	2	4	5			
	2	1	2	3	4	5			
	3	1	2	3	4	5			
i=4	0	1	2	3	4	5			
	1	1	2	3	4				
	2	1	2	3	4				
i=5	0	1	2	3	4				
	1	1	2	3					
i=6	0	1	2	3					
		1	2						

- There are a few observations you can make that can speed this up a bit:
  - After the first pass, the largest element is at the end. After the second pass, the second largest element is second to the end, etc. (This means each pass you make doesn't actually need to go through the entire array, later passes can stop earlier.)
  - If you make a pass and don't perform any swaps, then the array is sorted.

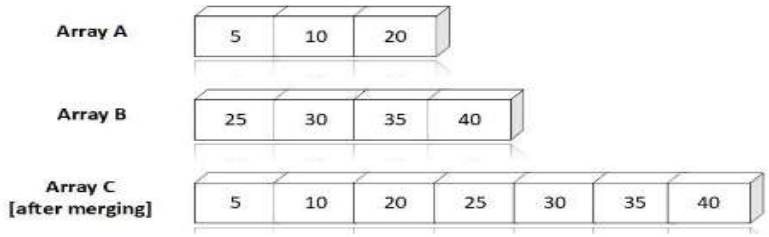
Algorithm: Bubblesort(arr,n)

Here arr is a linear array with n elements

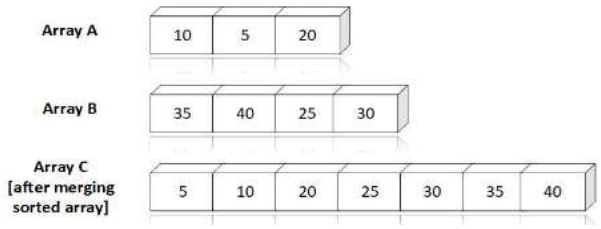
```
1. for(i = 0; i < n-1; i++)
        for(j = 0; j < n-i-1; j++)
            if( arr[j] > arr[j+1])
                    temp = arr[j];
                        arr[j] = arr[j+1];
                         arr[j+1] = temp;
 2.Print the contents in arr
3.Exit.
```

6. Merge two arrays

- **Merging Unsorted Array**:
  - If the arrays are not sorted, we can combine them end to end i.e. we can first put the elements of first array into third array and then the elements of second array are placed after it in third array.



- **Merging Sorted Array:**
  - In many cases, we may have sorted arrays and our aim is to combine them in such way that the combined array is also in sorted order.
  - One straight forward approach is to join them end to end and then sort the combined array, but this approach is not efficient and economical.
  - Therefore the best approach is to compare the elements of the given array, and based on this comparison; decide which element should go first to the third array.



Algorithm: mergeArrays(A[], B[], n1, n2, C[])

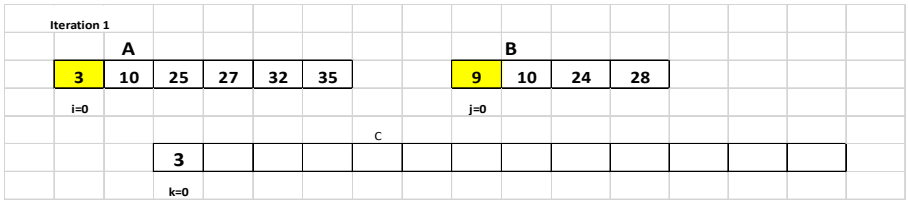
/ * A and B are the two arrays to be merged into C. n1 and n2 are the no: of elements in A and B respectively.*/

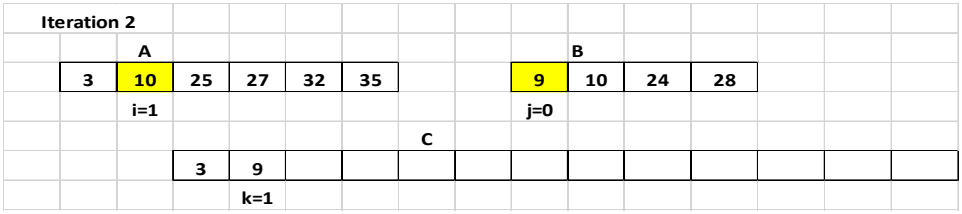
1. Set i = 0, j = 0, k = 0
2. while (i<n1 && j <n2)

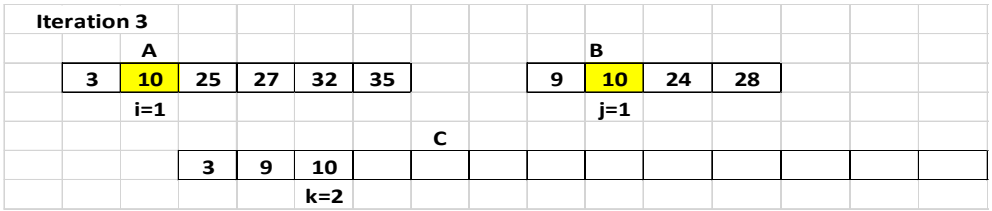
```
{
        if (A[i] < B[j])
        {
                C[k] = A[i]
            i++
        }
         else if (A[i] > B[ j ])
        {
              C[k] = B[ j ]
            j++
        }
        else
        {
                C[k] = B[j];
                j++     i++
        }
        k++
```

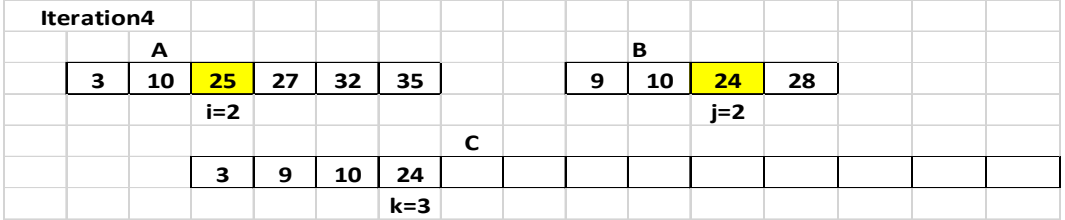
```
    }
3.  while (i < n1)
        {
            C[k] = A[i]
            i++,    k++
     }
4.  while (j < n2)
        {
            C[k] = B[j]
            k++,    j++
    }

5. for(i=0;i<k;i++)
       print(C[i])
6.Exit
```

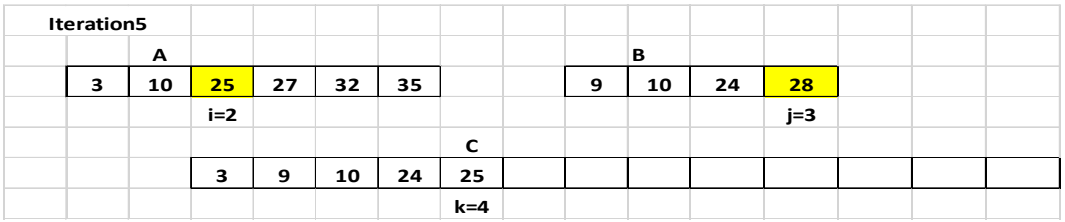
- **Example**

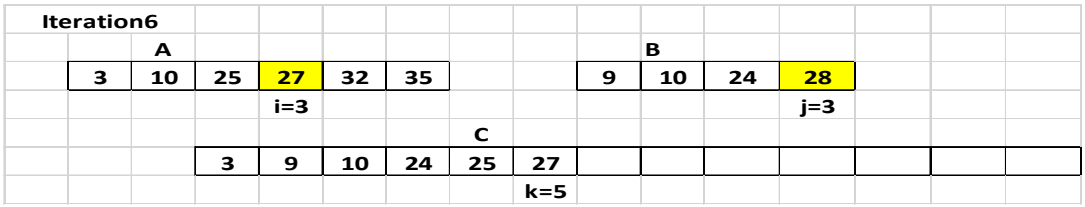












Multidimensional Array

- In multidimensional array elements referred by more than on indices or subscripts.
- These arrays also have finite fixed size collection of homogeneous elements.
- The general form of declaring N-dimensional arrays is:

  data_type array_name[size1][size2]....[sizeN];

- Examples:

  Two dimensional array: int two_d[10][20];

  Three dimensional array: int three_d[10][20][30];

Size of Multidimensional Arrays:

- The total number of elements that can be stored in a multidimensional array can be calculated by multiplying the size of all the dimensions.
- For example:
  a) The array int x[10][20] can store total
     (10*20) = 200 elements.
  b) Similarly array int x[5][10][20] can store
     total (5*10*20) = 1000 elements.

Two – dimensional array (2D array)

- 2D arrays are called matrices in mathematics and tables in business applications.
- The standard way of writing a 2D array as A[m,n]
  - □ where the elements of the array A form a rectangular array with m rows and n columns.

- 2D arrays are represented with 2 subscripts such as A[i][j] or A[i,j], where i denotes the row and j denotes the column.
- Matrices are the mathematical form of 2D arrays.
- A two – dimensional array 'x' with 3 rows and 3 columns is shown below:

	Column 0	Column 1	Column 2
Row 0	x[0][0]	x[0][1]	x[0][2]
Row 1	x[1][0]	x[1][1]	x[1][2]
Row 2	x[2][0]	x[2][1]	x[2][2]

Initializing Two – Dimensional Arrays:

- There are various ways in which a Two-Dimensional array can be initialized.

First Method:

- int x[3][4] = {0, 1 ,2 ,3 ,4 , 5 , 6 , 7 , 8 , 9 , 10 , 11}
- The above array has 3 rows and 4 columns.
- The elements in the braces from left to right are stored in the table also from left to right.
- The elements will be filled in the array in order, the first 4 elements from the left in the first row, the next 4 elements in the second row, and so on.

Second Method:

- int x[3][4] = {{0,1,2,3}, {4,5,6,7}, {8,9,10,11}};

Third Method:

```
int x[3][4];
for(int i = 0; i < 3; i++)
{
for(int j = 0; j < 4; j++)
{
scanf("%d",&x[i][j]);
}
}
```

Representation of 2D array in memory

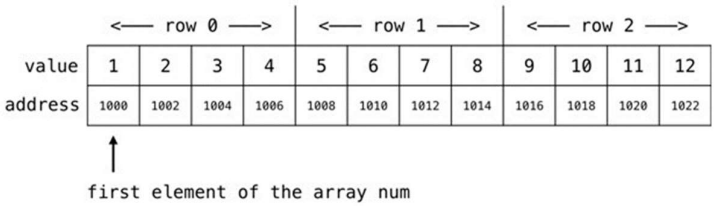
- Let A be 2D array with m rows and n columns.
- But the array will be represented in memory by sequence memory location.
- 2D array is stored in memory in 2 ways
  - Row-major representation
  - Column - major representation

a) Row-major representation

- All the elements of row1 are stored first, and then the elements of row2 and so on.

```
int num[3][4] = {
  {1, 2,  3,  4},
  {5, 6,  7,  8},
  {9, 10, 11, 12}
};
```

row-wise memory allocation



b) Column - major representation

- All the elements of column1 are stored first and the elements of column2 and so on.

We can perform following operations on matrices,

Creation
Addition
Multiplication
Transpose
Scalar multiplication.

Example program-

Storing elements in a matrix and printing it

```
#include <stdio.h>
void main ()
{
    int arr[3][3],i,j;
    for (i=0;i<3;i++)
    {
        for (j=0;j<3;j++)
        {
            printf("Enter a[%d][%d]: ",i,j);
            scanf("%d",&arr[i][j]);
        }
    }
    printf("\n printing the elements ....\n");
    for(i=0;i<3;i++)
    {
        printf("\n");
        for (j=0;j<3;j++)
        {
            printf("%d\t",arr[i][j]);
        }
    }
```

```
}
```

Output

```
Enter a[0][0]: 56
Enter a[0][1]: 10
Enter a[0][2]: 30

Enter a[1][0]: 34
Enter a[1][1]: 21
Enter a[1][2]: 34

Enter a[2][0]: 45
Enter a[2][1]: 56
Enter a[2][2]: 78

printing the elements ....
56      10      30
34      21      34
45      56      78
```

Stack

- A stack is an Abstract Data Type commonly used in most programming languages.
- It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.

  Stack Example

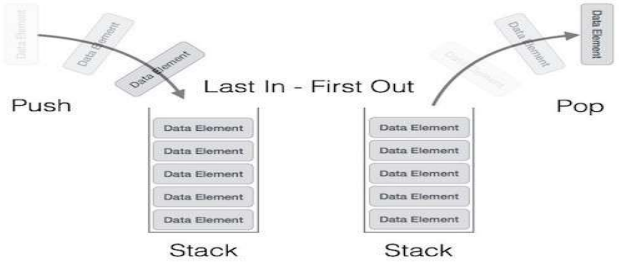

- A real-world stack allows operations at one end only.
- For example, we can place or remove a card or plate from the top of the stack only.

- Stack allows all data operations at one end only.
- At any given time, we can only access the top element of a stack.
- This feature makes it LIFO data structure. LIFO stands for Last-in-first-out.
- Here, the element which is placed (inserted or added) last, is accessed first.

  - A stack is an ordered collection of homogeneous data elements where the insertion and deletion operations take place at one end only, is called the top of the stack (TOP).
  - When insertion or deletion is done, its base remains fixed, where TOP changes.

Basic Operations on stack

- **push()** − Pushing (storing) an element on the stack.
- **pop()** − Removing an element from the stack.

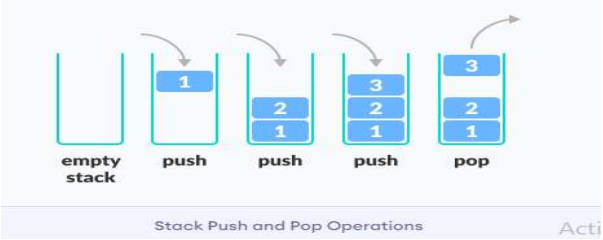


Representation of a Stack

1.Using a one-dimensional array (Static implementation)

2.Using a single linked list. (Dynamic Implementation)

- **Array Representation of Stacks:**
  - First we have to allocate a memory block of sufficient size to accommodate the full capacity of the stack.
  - Then, starting from the first location of the memory block, the items of the stack can be stored in a sequential fashion.
  - TOP is a pointer used to point the position of the array up to which items are filled in the stack.



- **Operations on the stack using array representation**

Algorithm PUSH(A[ ], TOP, SIZE, ITEM)

/* A is the stack where TOP is a pointer to the top element of the stack, SIZE is the capacity of the stack .ITEM is the new element to be pushed onto the stack*/

1. If(TOP=SIZE-1)

   print ‘stack is full‘

   else

TOP=TOP+1

A[TOP]=ITEM

2. Exit

Algorithm POP(A[],TOP,ITEM)

/* A is the stack where TOP is a pointer to the top element of the stack, Removes the top element from the stack and stored to ITEM/

1.if(TOP<0)

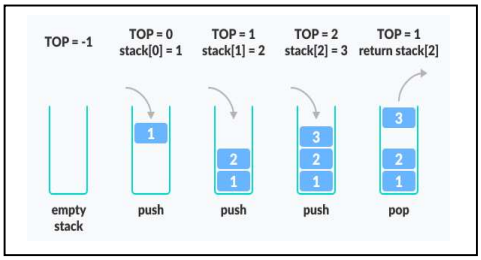
print 'stack is empty'

else

ITEM=A[TOP]

TOP=TOP-1

2.Exit



Queue

Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (Enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

Basic Operations

- **enqueue()** − add (store) an item to the queue.
- **dequeue()** − remove (access) an item from the queue.
- In a queue insertion and deletion takes place at two ends called REAR and FRONT of the queue respectively.

Representation of Queue (or) Implementation of Queue:

- The queue can be represented in two ways:
  1. Queue using Array
  2. Queue using Linked List

❑ **Array Representation of Queue**

- We can easily represent queue by using linear arrays.
- There are two variables i.e. FRONT and REAR, that are implemented in the case of every queue.
- FRONT and REAR variables point to the position from where insertions and deletions are performed in a queue.

- Initially, the value of FRONT and REAR is -1 which represents an empty queue.
- On enqueing an element, we increase the value of REAR index and place the new element in the position pointed to by REAR.
- On dequeueing an element, we return the value pointed to by FRONT and increase the FRONT index
- When rear reaches length-1 the queue is full.

Algorithm ENQUEUE(Q[], Length, ITEM, FRONT, REAR)

/*Q is the Array for Queue, Length is the length of the queue, ITEM is the element to be added to the queue. FRONT and REAR point to the front and rear element in the queue.*/

```
1: IF REAR = LENGTH - 1
            Write OVERFLOW
            Go to step4

2: IF FRONT = -1 and REAR = -1
            FRONT = REAR = 0
   ELSE
            REAR = REAR + 1
3: Q[REAR] = ITEM

4: Exit
```

Algorithm DEQUEUE(Q[],FRONT REAR,ITEM)

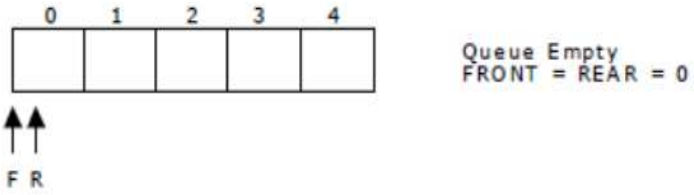
/*Q is the Array for Queue ,the deleted element will be stored in the variable ITEM. FRONT and REAR point to the front and rear element in the queue.*/

```
1: if FRONT = -1 and REAR=-1
       Write Q EMPTY

       goto step3

   else

       ITEM = Q[FRONT]

2: if FRONT=REAR

       FRONT=REAR=-1

   else

          FRONT = FRONT + 1
```

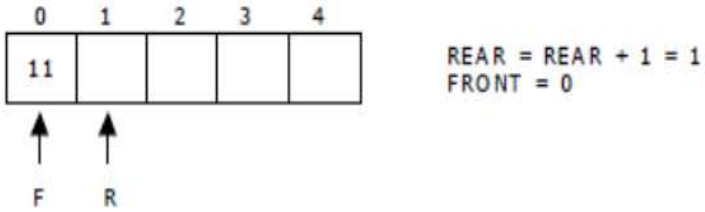
3: Exit

1.Queue using Array:

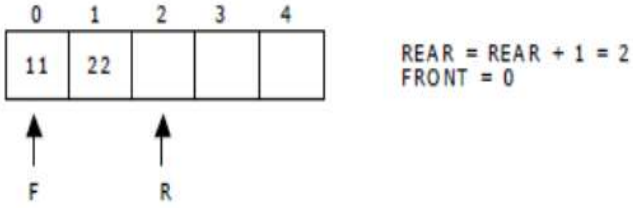
Let us consider a queue, which can hold maximum of five elements. Initially the queue is empty.



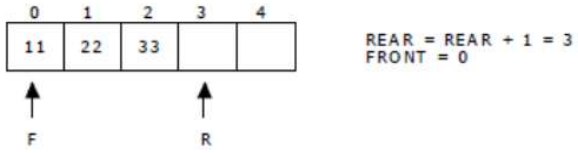
Now, insert 11 to the queue. Then queue status will be:



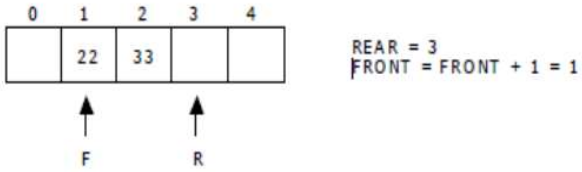
Next, insert 22 to the queue. Then the queue status is:



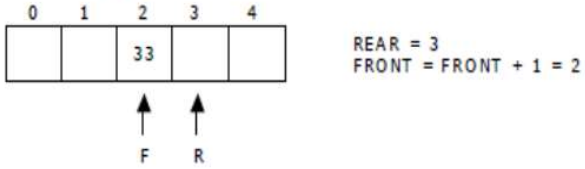
Again insert another element 33 to the queue. The status of the queue is:



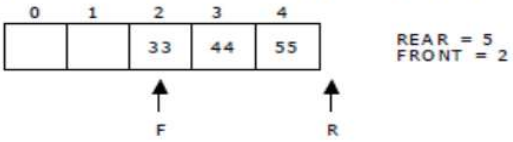
Now, delete an element. The element deleted is the element at the front of the queue.So the status of the queue is:



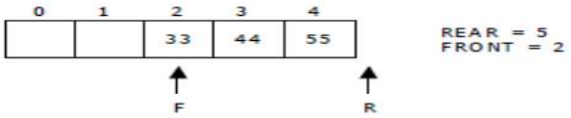
Again, delete an element. The element to be deleted is always pointed to by the FRONT pointer. So, 22 is deleted. The queue status is as follows:



Now, insert new elements 44 and 55 into the queue. The queue status is:



Next insert another element, say 66 to the queue. We cannot insert 66 to the queue as the rear crossed the maximum size of the queue (i.e., 5). There will be queue full signal. The queue status is as follows:



Drawback of array implementation

- Although, array implementation of a queue is easy, it has some drawbacks.
- **Memory wastage :**

limitation of array representation of queue

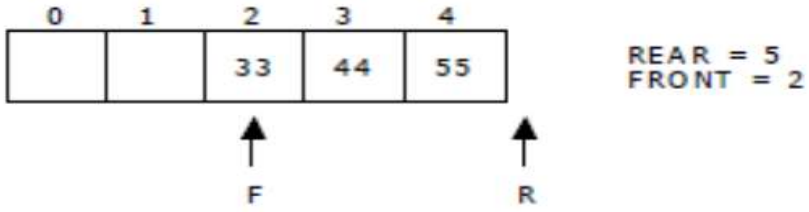
- **Deciding the array size**
  - One of the most common problem with array implementation is the size of the array which requires to be declared in advance.

Types of Queues

- There are four different types of queues:
    - Simple Queue
    - Circular Queue
    - Priority Queue
    - Double Ended Queue

b) Circular Queue

- Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle.
- A more efficient queue representation is obtained by regarding the array Q as circular.
- Any number of items could be placed on the queue.
- This implementation of a queue is called a circular queue because it uses its storage array as if it were a circle instead of a linear list.
- For example, let us consider a linear queue status as follows:

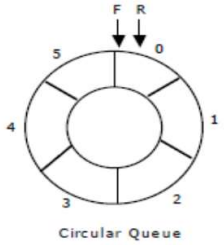


Next insert another element, say 66 to the queue. We cannot insert 66 to the queue as the rear crossed the maximum size of the queue (i.e., 5). There will be queue full signal. This difficulty can be overcome if we treat queue position with index zero as a position that comes after position with index four then we treat the queue as a circular queue. In circular queue if we reach the end for inserting elements to it, it is possible to insert new elements if the slots at the beginning of the circular queue are empty.

Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle.

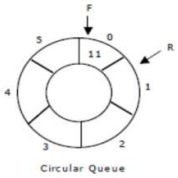
Representation of Circular Queue:

- Let us consider a circular queue, which can hold maximum (MAX) of six elements. Initially the queue is empty.



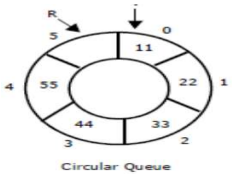
Queue Empty
MAX = 6
FRONT = REAR = 0
COUNT = 0

- Now, insert 11 to the circular queue. Then circular queue status will be:



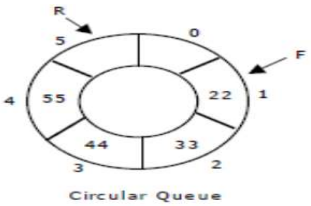
FRONT = 0
REAR = (REAR + 1) % 6 = 1
COUNT = 1

- Insert new elements 22, 33, 44 and 55 into the circular queue. The circular queue status is:



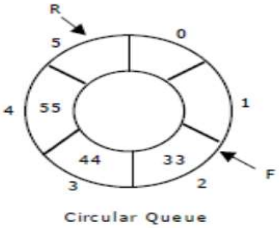
FRONT = 0
REAR = (REAR + 1) % 6 = 5
COUNT = 5

- Now, delete an element.
- The element deleted is the element at the front of the circular queue. So, 11 is deleted. The circular queue status is as follows:



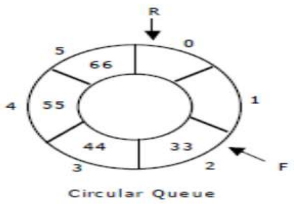
FRONT = (FRONT + 1) % 6 = 1
REAR = 5
COUNT = COUNT - 1 = 4

- Again, delete an element.
- The element to be deleted is always pointed to by the FRONT pointer. So, 22 is deleted. The circular queue status is as follows:



FRONT = (FRONT + 1) % 6 = 2
REAR = 5
COUNT = COUNT - 1 = 3

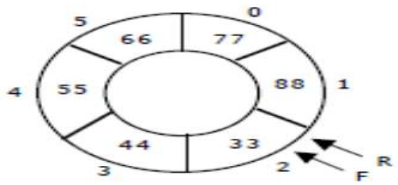
- Again, insert another element 66 to the circular queue. The status of the circular queue is:



Circular Queue

FRONT = 2
REAR = (REAR + 1) % 6 = 0
COUNT = COUNT + 1 = 4

- Now, insert new elements 77 and 88 into the circular queue. The circular queue status is:

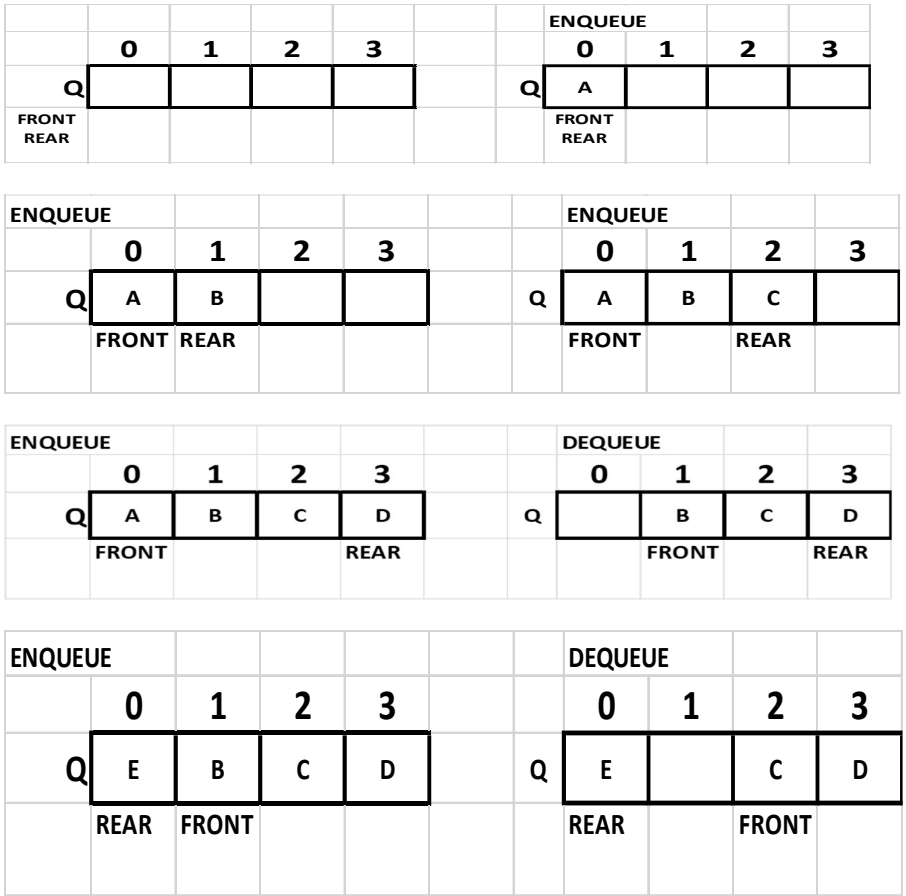


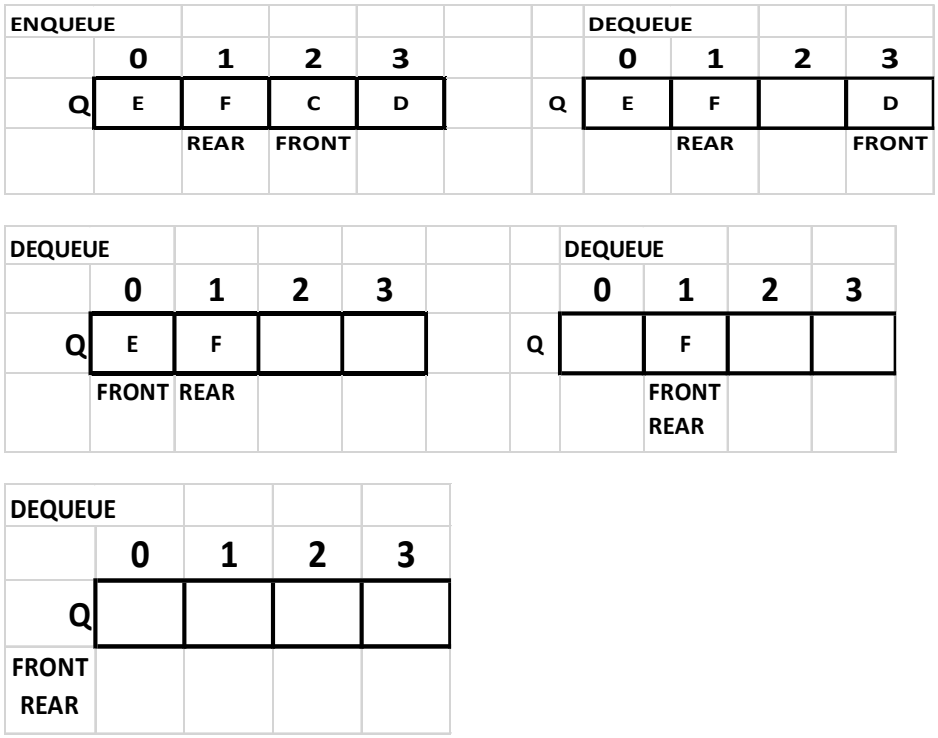
Circular Queue

FRONT = 2, REAR = 2
REAR = REAR % 6 = 2
COUNT = 6

- Now, if we insert an element to the circular queue, as COUNT = MAX we cannot add the element to circular queue. So, the circular queue is full.

➢ **Example- Circular Queue implementation**





Operations on Circular queue:

- **Enqueue_CQ():** This function is used to insert an element into the circular queue.
  - In a circular queue, the new element is always inserted at Rear position

Algorithm: Enqueue_CQ(Q[], REAR, FRONT, MAX, data):

/*Q is the Array for Queue, MAX is the length of the queue, data is the element to be added to the queue. FRONT and REAR point to the front and rear element in the queue.*/

1:START

2: if count==MAX then

Print "Circular queue is full"

else

Q[REAR]=data

REAR=(REAR+1)%MAX

count=count+1

4: Exit

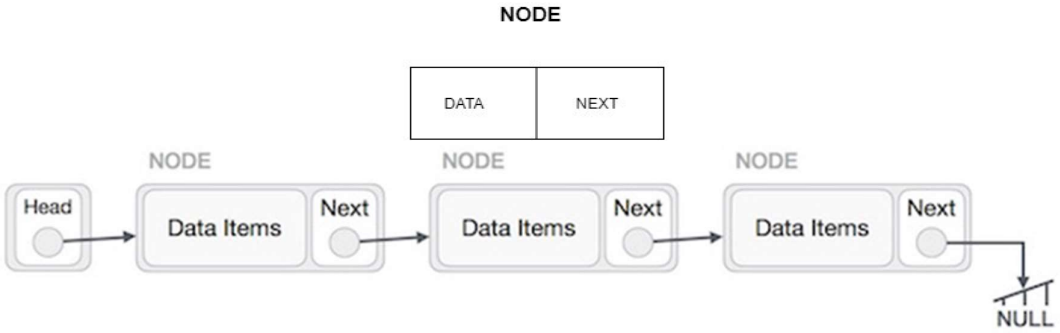
- **Dequeue_CQ()**
  - This function is used to delete an element from the circular queue.
  - In a circular queue, the element is always deleted from front position.

Algorithm: Dequeue_CQ(Q[], FRONT, REAR, MAX)

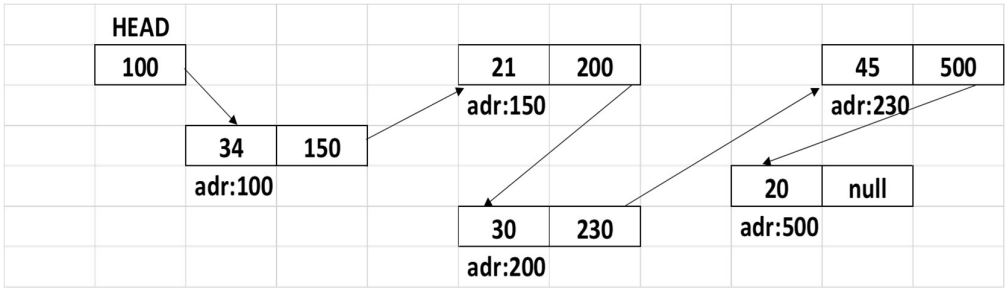
```
/*Q is the Array for Queue ,the deleted element will be stored in the variable data.
FRONT and REAR point to the front and rear element in the queue.*/
1:START
2: if count==0 then
        Write "Circular queue is empty"
   else
        data=Q[FRONT]
        FRONT=(FRONT+1)%MAX
        count=count-1
4: Exit
```

Linked List

- A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations.
- The elements in a linked list are linked using pointers.
- A linked list consists of nodes where each node contains a data field and a reference(link) to the next node in the list.



- Linked List contains a link element called head which is pointing to the first node of the linked list.
- Each node carries a data field(s) and a link field called next.
- Each node is linked with its next node using its next link.
- Last node carries a null value for next link.



Advantages of Linked Lists

- They are dynamic in nature which allocates the memory when required.
- Insertion and deletion operations can be easily implemented.

- Stacks and queues can be easily executed.
- Linked List reduces the access time.

Disadvantages of Linked Lists

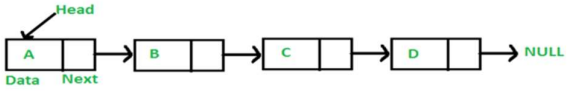
- The memory is wasted as pointers require extra memory for storage.
- No element can be accessed randomly; it has to access each node sequentially.
- Reverse Traversing is difficult in linked list.

Applications of Linked Lists

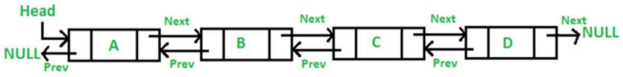
- Linked lists are used to implement stacks, queues, graphs, etc.
- Linked lists let you insert elements at the beginning and end of the list.
- In Linked Lists we don't need to know the size in advance.
- Polynomial representation and manipulation
- Sparse matrix representation
- Dynamic memory management

Types of Linked List

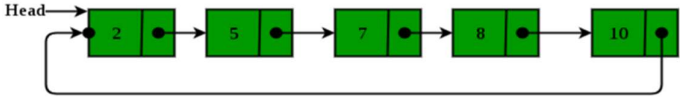
- **Simple(single) Linked List** − Item navigation is forward only.



- **Doubly Linked List** − Items can be navigated forward and backward.

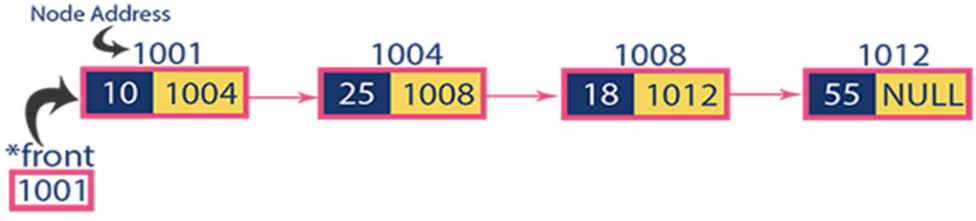


- **Circular Linked List** − Last item contains link of the first element as next



a) Singly Linked List

- Each node contains only one link which points to the subsequent node in the list.
- Singly linked lists contain nodes which have a data part as well as an address part i.e. next, which points to the next node
- Starting from first node one can reach to the last node whose link field does not contain any address but has a NULL value



Basic linked list operations are:

- Traversal - access each element of the linked list
- Insertion - adds a new element to the linked list
- Deletion - removes the existing elements
- Search - find a node in the linked list
- Sort - sort the nodes of the linked list

i) Traversing

- **Moving through the linked list and visiting each node exactly ones.**

Algorithm: Traverse(head)

1: set ptr = head
2: if ptr == null
write "empty list"
goto step 4
3: while (ptr != null)
{
print (ptr→ data)
ptr = ptr → next
}
4: Exit

ii) Searching

- Search a particular element is present in the linked list or not.

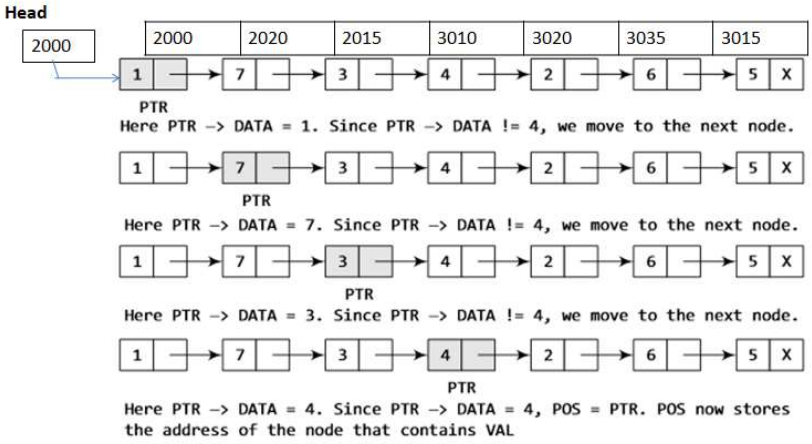
Algorithm SearchList(Head,Val)

/* Head is a pointer to the first node of the linked List. Val is the element to be searched in the linked list*/

```
1.ptr=head
  flag=0
2.while(ptr!=null)
      if(ptr->data==Val)
            flag=1
            print "ítem present at location", ptr
            goto step3
      else
            ptr=ptr->next
3.if (flag= =0)
      print "Item not present''
4. Exit
```

Example:
Search the element 4 is present in the linked list or not

So Val=4



iii) Insertion

- In a single linked list, the insertion operation can be performed in three ways. They are as follows...
- ✓ a) Inserting At Beginning of the list
- ✓ b) Inserting At End of the list
- ✓ c) Inserting At Specific location in the list

a) Inserting At Beginning of the list

Steps for inserting at the beginning

1. Create a **new node** with given value which has to be inserted into the beginning of the list

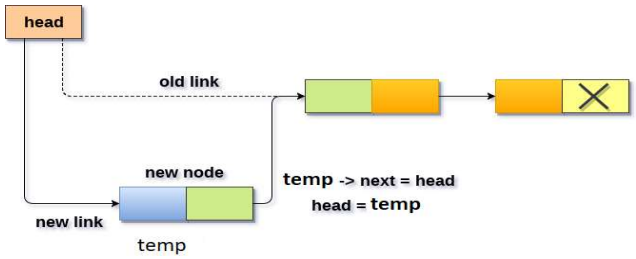
temp->data=item

2. Head points to the first element of the linked list. Now we wanted to insert a new node at the beginning. We assign the value of Head to the link part of the new temp node

temp->next=Head

3. The new node is inserted as the first node, so Had is reassigned as,

Head=temp



Algorithm: InsertFirst(Head, item)

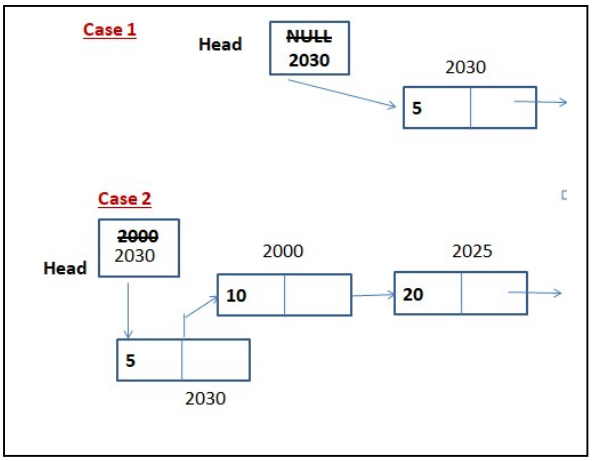
/***head** is a pointer to the first element of the linked list. **Item** is the element to be inserted. **temp** is a new node created.*/

1. if(temp==null)

print 'memory insufficient'

goto step5

2. temp->data = item;



3. if(head == NULL)
   {
     temp->next = NULL;
     head = temp;
   }
   else
   {
     temp->next = head;
     head = temp;
   }
4. print "One node inserted"
5. Exit.

b) Inserting at the end of the list

Steps for inserting at the end

1. Create a **new node** with given value which has to be inserted at the end of the list
temp->data=item
temp->next=NULL

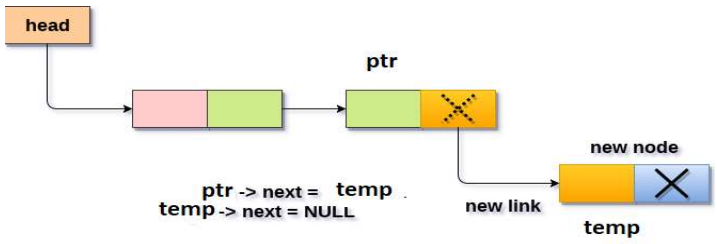
2: Check whether list is **Empty** (**head** == **NULL**).

3: If it is **Empty** then, set **head** = **temp**.

4: If **head** is **Not Empty** then, define a new node pointer ptr and set **ptr=head**.

5: Keep moving the **ptr** to its next node until it reaches to the last node in the list (until **ptr → next** = **NULL**).

6: Set **ptr → next** = **temp**



Inserting node at the last into a non-empty list

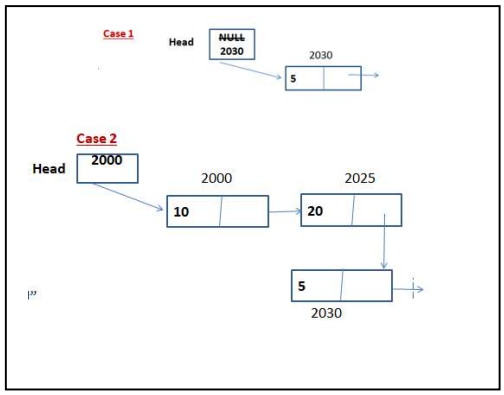
Algorithm InsertLast(Head, item)

/*Head is a pointer to the first element of the linked List. item is the element to be inserted at the end of the linked list. temp is a new node created */

1. If (temp==NULL)

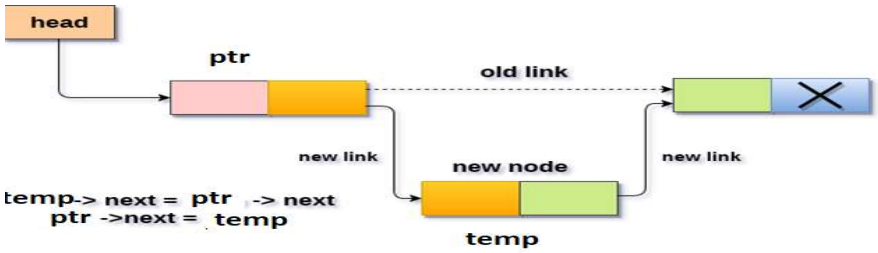
print “Memory insufficient”.
goto step6

2. temp->data=item
   temp->next=NULL
3. if(head==NULL)
   {
   head=temp
   goto step9
   }
   else
   {
   ptr=head
   while(ptr->next!=null)
   ptr=ptr->next
   }
4. ptr->next=temp
5. print “One node inserted”
6. Exit



c) Inserting at Specific location in the list

- Inserting a node into a single Linked List at any position in the list.



Algorithm insert(Head, item, key)

/*Head is a pointer to the first element of the linked List. Item is the element to be inserted after the ‘key’ element in the linked list. **temp** is a new node created */

1. if(temp==null)
   print ’memory insufficient’
   goto step 4
2. temp->data = item;

3. if(head == NULL)
   {
   temp->next = NULL;
   head = temp;

```
        }
    else
     {
        ptr=head
        while(ptr!=Null)
        {
         if (ptr->data == key)
        {
                temp->next=ptr->next
                ptr->next=temp
                print ítem inserted
                goto step 4
        }
        else
                ptr=ptr->next
        }
```

4. Exit

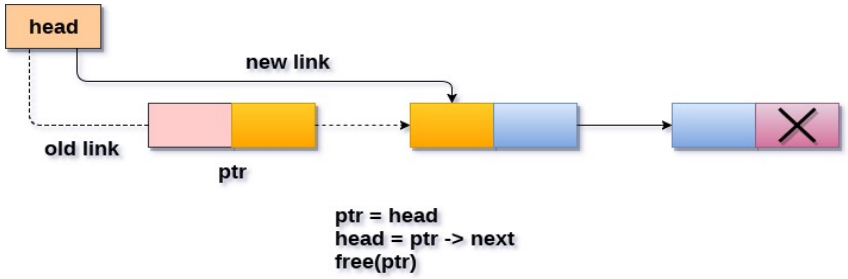
iv) Delete a node from List

To delete a node from linked list, 3 possibilities are there

a) Delete node from beginning
b) Delete node at the end
c) Delete a specific node

a)Delete node from beginning

- To remove the first node, we need to make the second node as head and delete the memory allocated for the first node.



Deleting a node from the beginning

Algorithm: DeleteFirst(Head)

/*Head is a pointer to the first element of the linked List. ptr is a new node created.*/

1: if Head = null
Print underflow
go to step 5
[end of if]

2: set ptr = head
3: set head = head -> next
4: free ptr
5: Exit

b) Delete the last node of the linked list

- To delete the last node of a linked list, find the second last node and make the next pointer of that node null.
- There are two scenarios in which, a node is deleted from the end of the linked list.
  - There is only one node in the list and that needs to be deleted.
  - There are more than one node in the list and the last node of the list will be deleted.

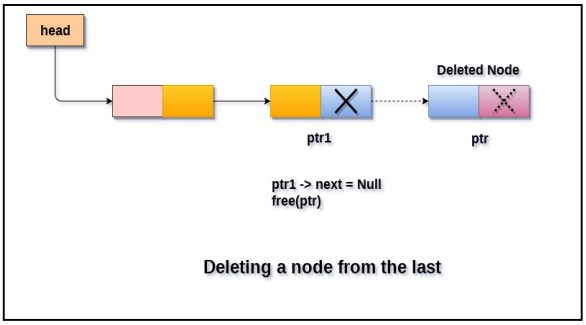
Algorithm: DeleteLast(Head)

/*Head is a pointer to the first element of the linked List. ptr and ptr1 are 2 new nodes*/

1: if Head = null
Print underflow
go to step 3
[end of if]

2. **if**(head -> next == NULL)
{
Set head = NULL;
free(head);
}
else
{
set ptr = head
while (ptr -> next!= null) do
{
set ptr1 = ptr
set ptr = ptr -> next
}
ptr1 -> next = null
free ptr
}

3: Exit



c) Delete a specific node

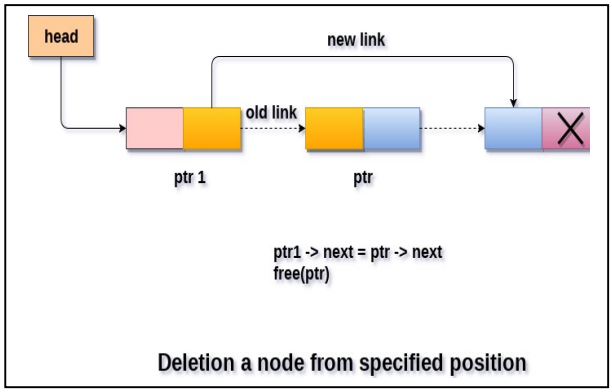
- In order to delete the node, which is present after the specified node, we need to skip the desired number of nodes to reach the node after which the node will be deleted.
- We need to keep track of the two nodes.

- The one which is to be deleted the other one if the node which is present before that node.
- For this purpose, two pointers are used:
  - ptr and ptr1.

Algorithm DeleteLinkedList(Head,key)

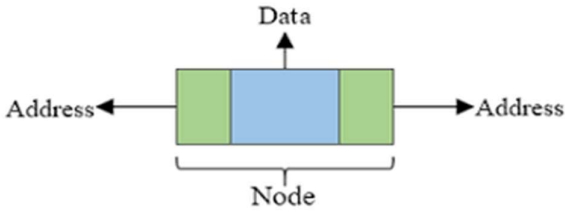
/*head is a pointer to the first node of the linked List. Key is the value in the node which has to be deleted */

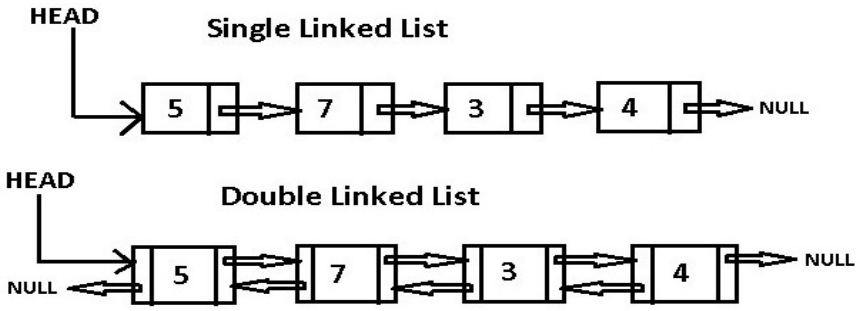
1.If (head==null)
    print "Linked list is empty"
    goto step6
2.If (head->data==key)
    head=head->next
    goto step6
3.ptr1=head
  ptr=head->next
4.while(ptr!=null)
    if(ptr->data=key)
        ptr1->next=ptr->next
        free(ptr)
        goto step6
    else
        ptr1=ptr
        ptr=ptr->next
5.if(ptr==Null)
    print 'node with key does not exist'
6.Exit



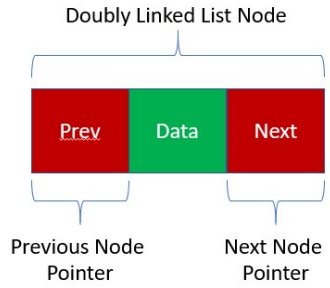
Doubly Linked List

- In a doubly linked list, each node contains two links, one of the link points to the previous node and the next link points to the next node in the sequence





Creating a node



- Each node consists:
  - A data item
  - An address of next node
  - An adders of the previous node
- The structure declaration is,

```
struct node
{
        struct node *prev; ;     // Pointer to previous node in DLL
        int data;
        struct node *next; ;     // Pointer to next node in DLL
};
```

If we wanted to create a new node

```
/* Initialize nodes */
        struct node *head=NULL;
        struct node *first;
/* Allocate memory */
        first = (struct Node*)malloc(sizeof(struct Node));
/* Assign data values */
         first->data = 10;
/* Connect nodes */
        first->prev = NULL
        first->next = NULL;
/* Save address of first node in head */
         head = first;
```

- **Doubly linked list operations are:**
  - Traversal - access each element of the linked list
  - Insertion - adds a new element to the linked list
  - Deletion - removes the existing elements
  - Search - find a node in the linked list

- Sort - sort the nodes of the linked list

i) Traversal

- **Moving forward through the linked list and visiting each node exactly ones.**

Algorithm: Traverse(head)

1: set ptr = head
2: if ptr == null
write "empty list"
goto step 4
3: while (ptr != null)
{
print ptr→ data
ptr = ptr → next
}
4: Exit

- Here Head is a pointer variable points to the starting node
- Assign ptr=Head. So ptr also points to the first node.
- Display the data part of the ptr as
ptr->data
- For processing the next element, we assign the address of next node to ptr as
ptr=ptr->next
- Continue this process until ptr=NULL

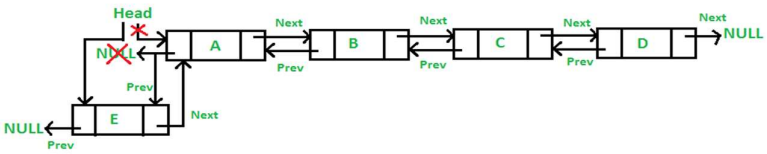
ii) Insertion into DLL

a. Insertion at beginning
- Adding the node into the linked list at beginning.

b. Insertion at end
- Adding the node into the linked list to the end.

c. Insertion after specified node
- Adding the node into the linked list after the specified node.

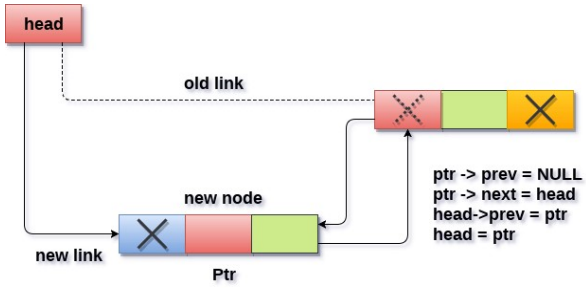
a. Insertion at beginning



Algorithm: InsertBeg(Head, Item)

/***Head** is a pointer to the first element of the linked list. **Item** is the element to be inserted. **ptr** is a new node created.*/

1: IF ptr = NULL
Write OVERFLOW
Go to Step 5
[END OF IF]
2: Set ptr->data=Item
3: If (Head == NULL)
{
ptr->next = NULL;
ptr->prev=NULL
}
else
{
Set ptr->prev=NULL
Set ptr->next=Head
Set haed->prev=ptr
}
4: Set head = ptr
5: Exit



Insertion into doubly linked list at beginning