

# Arrays

# Introduction

- An array is a collection of similar data elements.
- These data elements have the same data type.
- Elements of arrays are stored in consecutive memory locations and are referenced by an index (also known as the subscript).
- Declaring an array means specifying three things:

*Data type* - what kind of values it can store. For example, int, char, float

*Name* - to identify the array

*Size* - the maximum number of values that the array can hold

- Arrays are declared using the following syntax:

`type name[size];`

1 <sup>st</sup> element	2 <sup>nd</sup> element	3 <sup>rd</sup> element	4 <sup>th</sup> element	5 <sup>th</sup> element	6 <sup>th</sup> element	7 <sup>th</sup> element	8 <sup>th</sup> element	9 <sup>th</sup> element	10 <sup>th</sup> element
marks[0]	marks[1]	marks[2]	marks[3]	marks[4]	marks[5]	marks[6]	marks[7]	marks[8]	marks[9]

# Accessing Elements of an Array

- To access all the elements of an array, we must use a loop.
- That is, we can access all the elements of an array by varying the value of the subscript into the array.
- But note that the subscript must be an integral value or an expression that evaluates to an integral value.

```
int i, marks[10];  
for (i=0; i<10; i++)  
    marks[i] = -1;
```

# Calculating the Address of Array Elements

- Address of data element,  $A[k] = BA(A) + w(k - \text{lower\_bound})$

where

A is the array

k is the index of the element whose address we have to calculate

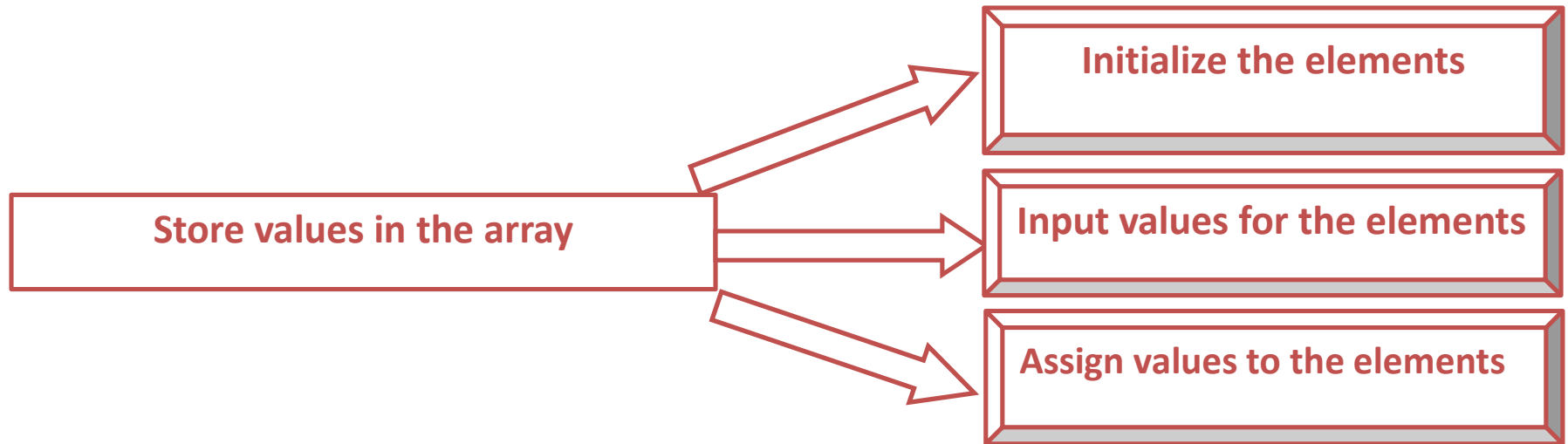
BA is the base address of the array A

w is the word size of one element in memory. For example, size of int is 2

99	67	78	56	88	90	34	85
marks[0] 1000	marks[1] 1002	marks[2] 1004	marks[3] 1006	<b>marks[4] 1008</b>	marks[5] 1010	marks[6] 1012	marks[7] 1014

$$\begin{aligned}\text{marks}[4] &= 1000 + 2(4 - 0) \\ &= 1000 + 2(4) = 1008\end{aligned}$$

# Storing Values in Arrays



Initializing Arrays during declaration

```
int marks [5] = {90, 98, 78, 56, 23};
```

Inputting Values from Keyboard

```
int i, marks[10];  
for(i=0;i<10;i++)  
scanf("%d", &marks[i]);
```

Assigning Values to Individual Elements

```
int i, arr1[10], arr2[10];  
for(i=0;i<10;i++)  
arr2[i] = arr1[i];
```

# Calculating the Length of an Array

$\text{Length} = \text{upper\_bound} - \text{lower\_bound} + 1$

where

upper\_bound is the index of the last element

lower\_bound is the index of the first element in the array

99	67	78	56	88	90	34	85
marks[0]	marks[1]	marks[2]	marks[3]	marks[4]	marks[5]	marks[6]	marks[7]

Here, lower\_bound = 0, upper\_bound = 7

Therefore, length =  $7 - 0 + 1 = 8$

# OPERATIONS ON ARRAYS

- ✓ Traversing an array
- ✓ Inserting an element in an array
- ✓ Searching an element in an array
- ✓ Deleting an element from an array
- ✓ Merging two arrays
- ✓ Sorting an array in ascending or descending order

## Algorithm for array traversal

step 1: [initialization] set  $i = \text{lower\_bound}$

step 2: repeat steps 3 to 4 while  $i \leq \text{upper\_bound}$

step 3: apply process to  $a[i]$

step 4: set  $i = i + 1$

[end of loop]

step 5: exit



# WAP to Read and Display *N* Numbers using an Array

```
#include<stdio.h>
#include<conio.h>
int main()
{
    int i=0, n, arr[20];
    clrscr();
    printf("\n Enter the number of elements : ");
    scanf("%d", &n);
    printf("\n Enter the elements : ");
    for(i=0;i<n;i++)
    {
        printf("\n arr[%d] = ", i);
        scanf("%d", &num[i]);
    }
    printf("\n The array elements are ");
    for(i=0;i<n;i++)
    printf("arr[%d] = %d\t", i, arr[i]);
    return 0;
}
```

# Inserting an Element in an Array

Algorithm to insert a new element to the end of an array

```
Step 1: Set upper_bound = upper_bound + 1  
Step 2: Set A[upper_bound] = VAL  
Step 3; EXIT
```

Algorithm INSERT( A, N, POS, VAL) to insert an element VAL at position POS

```
Step 1: [INITIALIZATION] SET I = N  
Step 2: Repeat Steps 3 and 4 while I >= POS  
Step 3:     SET A[I + 1] = A[I]  
Step 4:     SET I = I - 1  
          [End of Loop]  
Step 5: SET N = N + 1  
Step 6: SET A[POS] = VAL  
Step 7: EXIT
```

# Deleting an Element from an Array

Algorithm to delete an element from the end of the array

Step 1: Set `upper_bound = upper_bound - 1`

Step 2: EXIT

Algorithm DELETE( A, N, POS) to delete an element at POS

Step 1: [INITIALIZATION] SET `I = POS`

Step 2: Repeat Steps 3 and 4 while `I <= N-1`

Step 3: SET `A[I] = A[I + 1]`

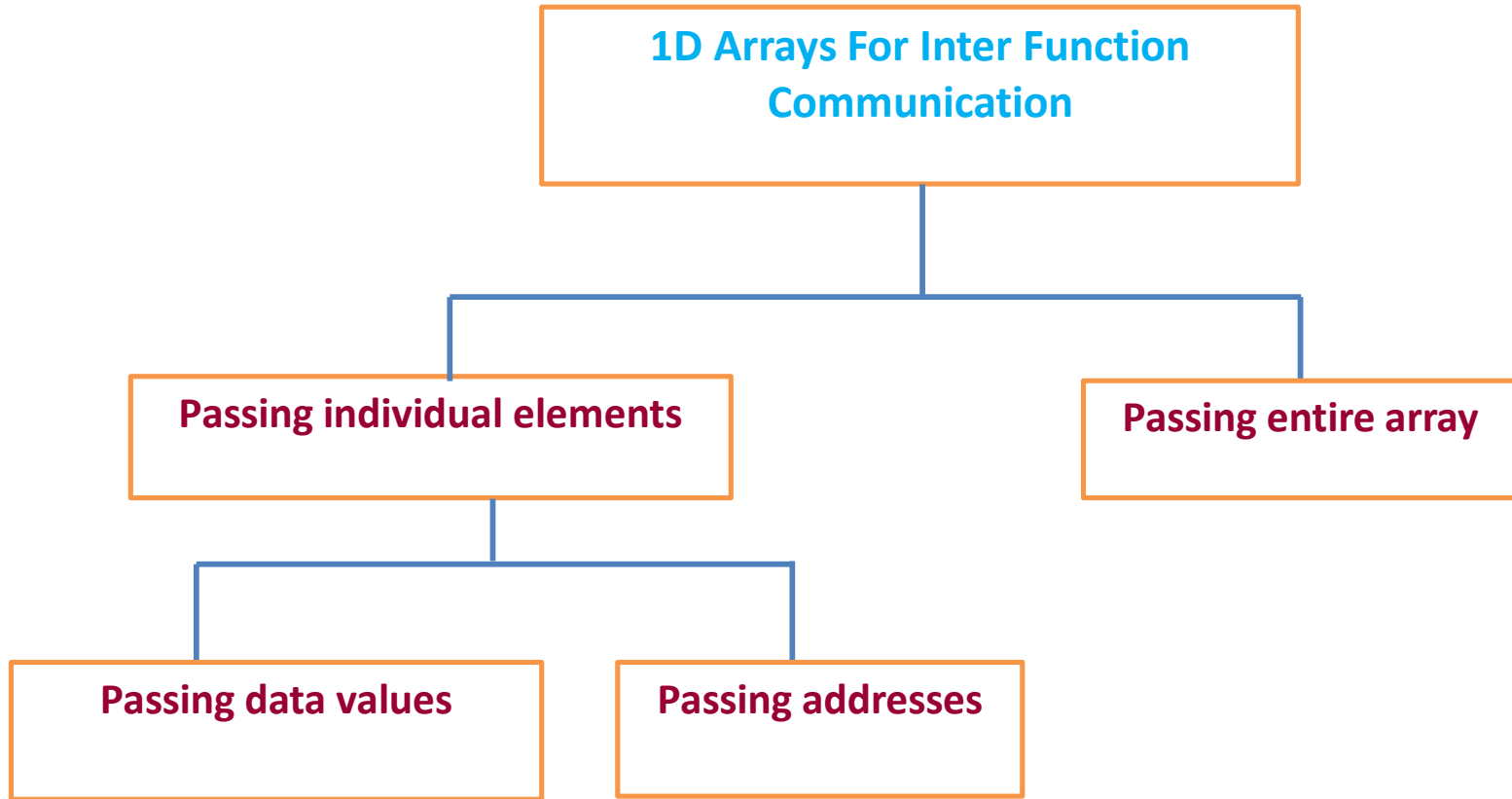
Step 4: SET `I = I + 1`

[End of Loop]

Step 5: SET `N = N - 1`

Step 6: EXIT

# Passing Arrays to Functions



# Passing Arrays to Functions

## Passing data values

```
main()
{
    int arr[5] = {1, 2, 3, 4, 5};
    func(arr[3]);
}
```

```
void func(int num)
{
    printf("%d", num);
}
```

## Passing addresses

```
main()
{
    int arr[5] = {1, 2, 3, 4, 5};
    func(&arr[3]);
}
```

```
void func(int *num)
{
    printf("%d", *num);
}
```

## Passing the entire array

```
main()
{
    int arr[5] = {1, 2, 3, 4, 5};
    func(arr);
}
```

```
void func(int arr[5])
{
    int i;
    for(i=0; i<5; i++)
        printf("%d", arr[i]);
}
```

# Pointers and Arrays

- Concept of array is very much bound to the concept of pointer.
- Name of an array is actually a pointer that points to the first element of the array.

```
int *ptr;  
ptr = &arr[0];
```

- If pointer variable ptr holds the address of the first element in the array, then the address of the successive elements can be calculated by writing ptr++.

```
int *ptr = &arr[0];  
ptr++;
```

```
printf ("The value of the second element in the array is %d", *ptr);
```

# Arrays of Pointers

- An array of pointers can be declared as:

```
int *ptr[10];
```

- The above statement declares an array of 10 pointers where each of the pointer points to an integer variable. For example, look at the code given below.

```
int *ptr[10];
```

```
ptr[2]=&r;
```

```
int p=1, q=2, r=3, s=4, t=5;
```

```
ptr[3]=&s;
```

```
ptr[0]=&p;
```

```
ptr[4]=&t
```

```
ptr[1]=&q;
```

**Can you tell what will be the output of the following statement?**

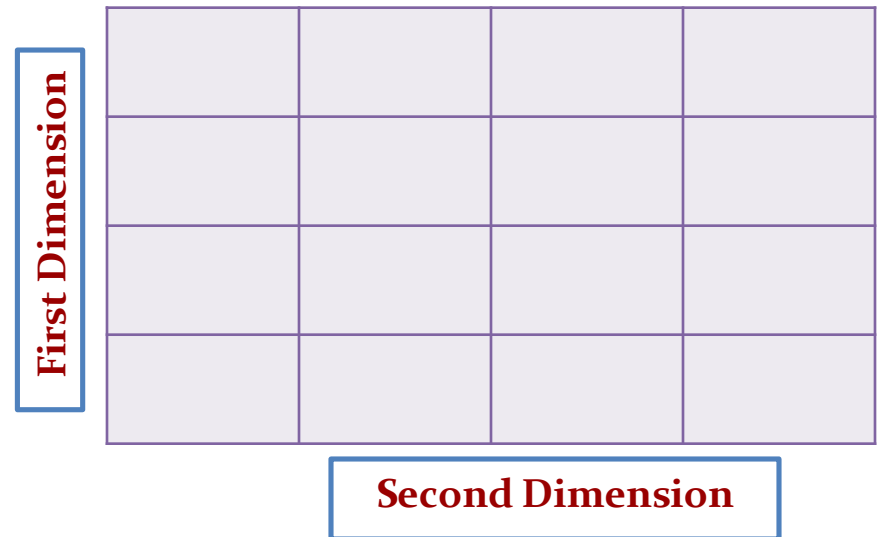
```
printf("\n %d", *ptr[3]);
```

# Two-dimensional Arrays

A two-dimensional array is specified using two subscripts where one subscript denotes row and the other denotes column.

C looks at a two-dimensional array as an array of one-dimensional arrays.

A two-dimensional array is declared as:  
`data_type`  
`array_name[row_size][column_size];`





# Two-dimensional Arrays

Therefore, a two dimensional  $m \times n$  array is an array that contains  $m \times n$  data elements and each element is accessed using two subscripts,  $i$  and  $j$ , where  $i \leq m$  and  $j \leq n$

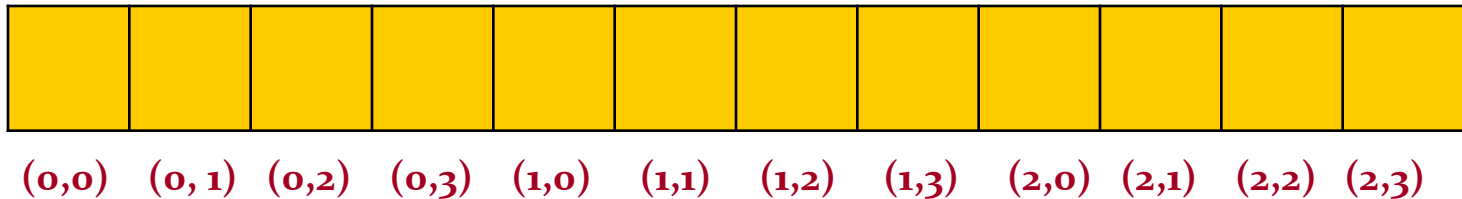
```
int marks[3][5];
```

Rows/Columns	Col 0	Col 1	Col2	Col 3	Col 4
Row 0	Marks[0][0]	Marks[0][1]	Marks[0][2]	Marks[0][3]	Marks[0][4]
Row 1	Marks[1][0]	Marks[1][1]	Marks[1][2]	Marks[1][3]	Marks[1][4]
Row 2	Marks[2][0]	Marks[2][1]	Marks[2][2]	Marks[2][3]	Marks[2][4]

Two Dimensional Array

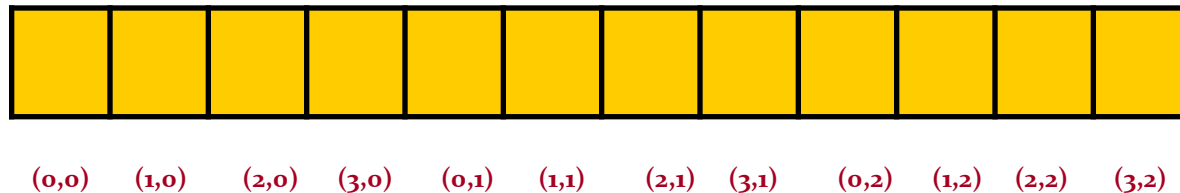
# Memory Representation of a Array

- There are two ways of storing a 2-D array in memory. The first way is **row-major order** and the second is **column-major order**.
- In the row-major order the elements of the first row are stored before the elements of the second and third rows. That is, the elements of the array are stored row by row where n elements of the first row will occupy the first nth locations.

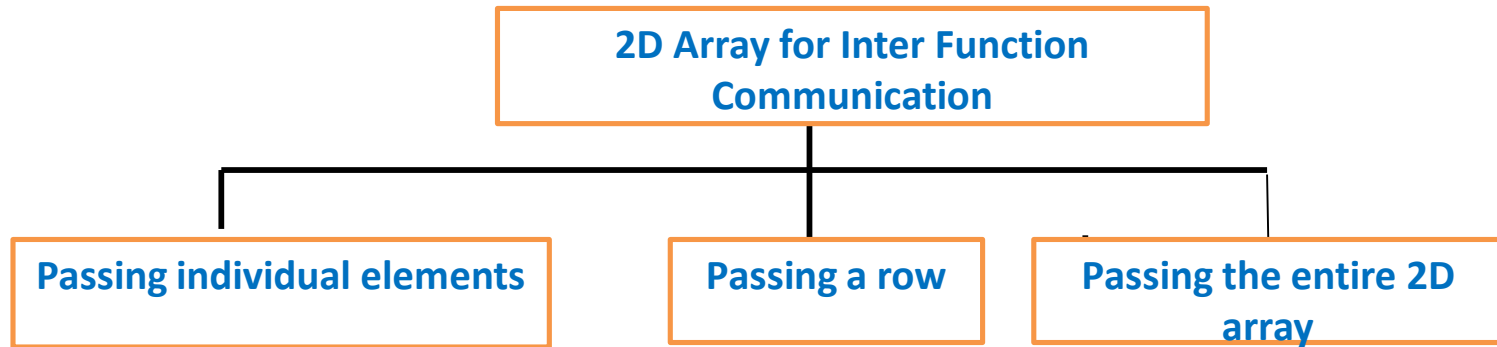


# Memory Representation of a Array

- However, when we store the elements in a column major order, the elements of the first column are stored before the elements of the second and third columns. That is, the elements of the array are stored column by column where  $n$  elements of the first column will occupy the first  $n$ th locations.



# Passing 2D Arrays to Functions



There are three ways of passing two-dimensional arrays to a function.

First, we can pass individual elements of the array. This is exactly same as we passed element of a one-dimensional array.

# Passing 2D Arrays to Functions

## Passing a row

### Calling function

```
main()
{
    int arr[2][3]= ( {1, 2, 3}, {4, 5, 6} );
    func(arr[1]);
}
```

### Called function

```
void func(int arr[])
{
    int i;
    for(i=0;i<3;i++)
        printf("%d", arr[i] * 10);
}
```

## Passing the entire 2D array

To pass a two dimensional array to a function, we use the array name as the actual parameter. (The same we did in case of a 1D array.) However, the parameter in the called function must indicate that the array has two dimensions

# Pointers and 2D Arrays

Individual elements of the array ***mat*** can be accessed using either:

`mat[i][j]` or `*(*(mat + i) + j)` or `*(mat[i]+j);`

Pointer to a one-dimensional array can be declared as:

```
int arr[]={1,2,3,4,5};
```

```
int *parr;
```

```
parr=arr;
```

Similarly, pointer to a two-dimensional array can be declared as:

```
int arr[2][2]={{1,2},{3,4}};
```

```
int (*parr)[2];
```

```
parr=arr;
```

# Initializing Multi-dimensional Arrays

A two and multi-dimensional array is initialized in the same way as a single dimensional array is initialized. For example,

```
int marks[2][3]={90, 87, 78, 68, 62, 71};
```

```
int marks[2][3]={{90,87,78},{68, 62, 71}};
```

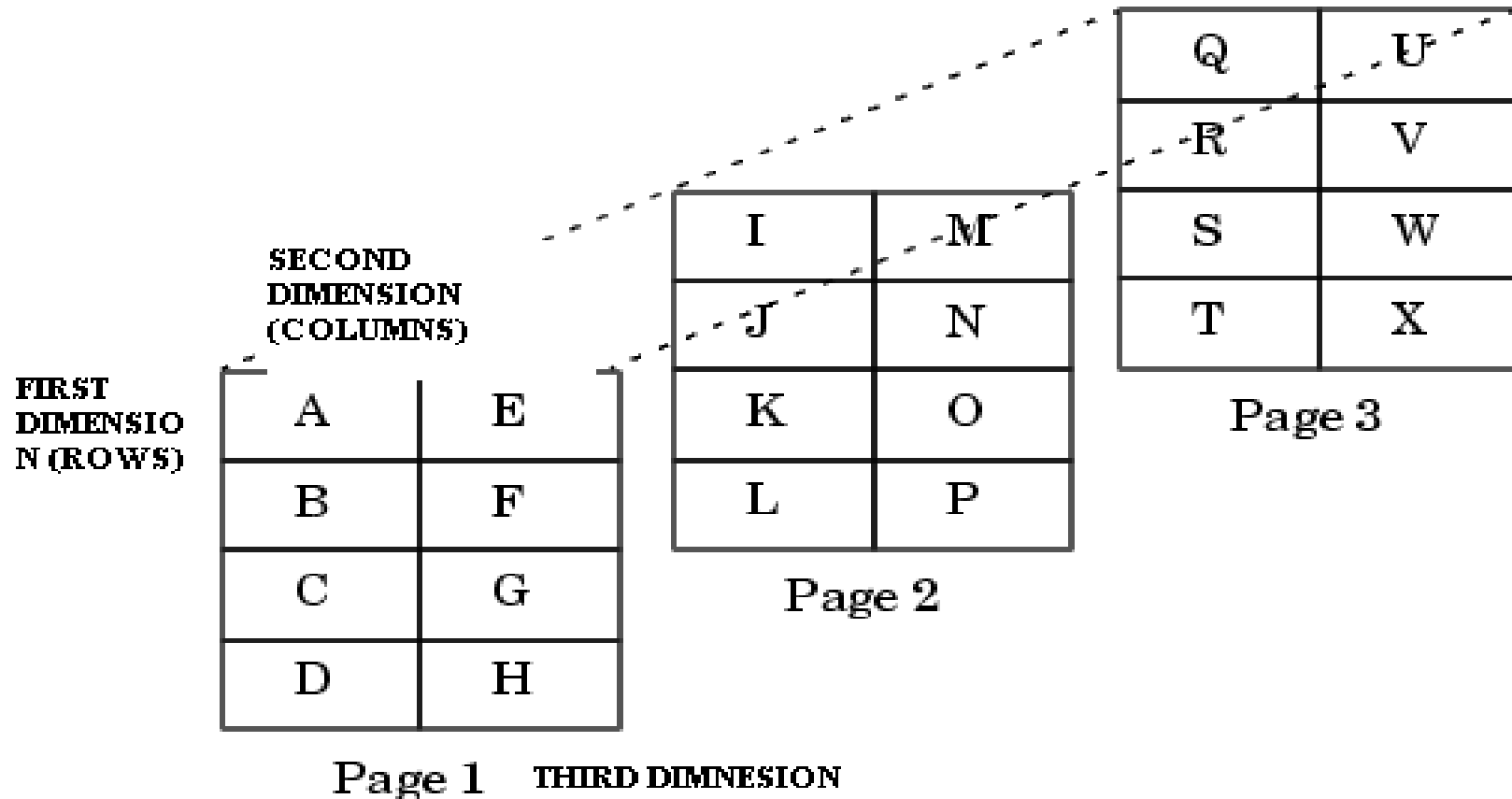
- A multi-dimensional array is declared and initialized the same way as we declare and initialize one- and two-dimensional arrays.

# Multi-dimensional Arrays

- A multi-dimensional array is an array of arrays.
- Like we have one index in a single dimensional array, two indices in a two-dimensional array, in the same way we have  $n$  indices in a  $n$ -dimensional array or multi-dimensional array.
- Conversely, an  $n$  dimensional array is specified using  $n$  indices.
- An  $n$  dimensional  $m_1 \times m_2 \times m_3 \times \dots \times m_n$  array is a collection of  $m_1 \times m_2 \times m_3 \times \dots \times m_n$  elements.
- In a multi-dimensional array, a particular element is specified by using  $n$  subscripts as  $A[I_1][I_2][I_3] \dots [I_n]$ , where
$$I_1 \leq M_1 \quad I_2 \leq M_2 \quad I_3 \leq M_3 \quad \dots \quad I_n \leq M_n$$



# Multi-dimensional Arrays



# Pointers and Three-dimensional Arrays

A pointer to a three-dimensional array can be declared as:

```
int arr[2][2][2]={1,2,3,4,5,6,7,8};
```

```
int (*parr)[2][2];
```

```
parr=arr;
```

We can access an element of a three-dimensional array by writing:

```
arr[i][j][k]= *(*(*arr+i)+j)+k)
```

# Applications of Arrays

- Arrays are widely used to implement mathematical vectors, matrices and other kinds of rectangular tables.
- Many databases include one-dimensional arrays whose elements are records.
- Arrays are also used to implement other data structures like heaps, hash tables, deques, queues, stacks and string. We will read about these data structures in the subsequent chapters.
- Arrays can be used for dynamic memory allocation.

## static memory allocation

memory is allocated at compile time.

memory can't be increased while executing program.

used in array.

## dynamic memory allocation

memory is allocated at run time.

memory can be increased while executing program.

used in linked list.

# dynamic memory allocation.

<code>malloc()</code>	allocates single block of requested memory.
<code>calloc()</code>	allocates multiple block of requested memory.
<code>realloc()</code>	reallocates the memory occupied by <code>malloc()</code> or <code>calloc()</code> functions.
<code>free()</code>	frees the dynamically allocated memory.

## malloc() function in C:

- The malloc() function allocates single block of requested memory.
- It returns NULL if memory is not sufficient.
- The syntax of malloc() function is given below:

`ptr=(cast-type*)malloc(byte-size)`

```
#include<stdio.h>
#include<stdlib.h>
int main(){
    int n,i,*ptr,sum=0;
    printf("Enter number of elements: ");
    scanf("%d",&n);
    ptr=(int*)malloc(n*sizeof(int)); //memory allocated using malloc
    if(ptr==NULL)
    {
        printf("Sorry! unable to allocate memory");
        exit(0);
    }
    printf("Enter elements of array: ");
    for(i=0;i<n;++i)
    {
        scanf("%d",ptr+i);
        sum+=*(ptr+i);
    }
    printf("Sum=%d",sum);
    free(ptr);
    return 0;
}
```

## **calloc() function in C:**

- The calloc() function allocates multiple block of requested memory.
- It initially initialize all bytes to zero.
- It returns NULL if memory is not sufficient.
- The syntax of calloc() function is given below:

**ptr=(cast-type\*)calloc(number, byte-size)**



```
#include<stdio.h>
#include<stdlib.h>
int main(){
    int n,i,*ptr,sum=0;
    printf("Enter number of elements: ");
    scanf("%d",&n);
    ptr=(int*)calloc(n,sizeof(int)); //memory allocated using calloc()
    if(ptr==NULL)
    {
        printf("Sorry! unable to allocate memory");
        exit(0);
    }
    printf("Enter elements of array: ");
    for(i=0;i<n;++i)
    {
        scanf("%d",ptr+i);
        sum+=*(ptr+i);
    }
    printf("Sum=%d",sum);
    free(ptr);
    return 0;
}
```

## realloc() function in C:

- If memory is not sufficient for malloc() or calloc(), you can reallocate the memory by realloc() function. In short, it changes the memory size.
- Let's see the syntax of realloc() function.

**ptr=realloc(ptr, new-size)**

- The memory occupied by malloc() or calloc() functions must be released by calling free() function. Otherwise, it will consume memory until program exit.
- Let's see the syntax of free() function.

`free(ptr)`

# C Functions

In c, we can divide a large program into the basic building blocks known as function.

The function contains the set of programming statements enclosed by {}.

A function can be called multiple times to provide reusability and modularity to the C program

# Advantage of functions in C

There are the following advantages of C functions.

- By using functions, we can avoid rewriting same logic/code again and again in a program.
- We can call C functions any number of times in a program and from any place in a program.
- We can track a large C program easily when it is divided into multiple functions.
- Reusability is the main achievement of C functions.
- However, Function calling is always a overhead in a C program.

# Function Aspects

- **Function declaration** A function must be declared globally in a c program to tell the compiler about the function name, function parameters, and return type.
- **Function call** Function can be called from anywhere in the program. The parameter list must not differ in function calling and function declaration. We must pass the same number of functions as it is declared in the function declaration.
- **Function definition** It contains the actual statements which are to be executed. It is the most important aspect to which the control comes when the function is called. Here, we must notice that only one value can be returned from the function.

S.No	C function aspects	Syntax
1	Function declaration	return_type    function_name    (argument list);
2	Function call	function_name (argument_list)
3	Function definition	return_type        function_name(argument list) { function body; }

# Types of Functions

There are two types of functions in C programming:

**1.Library Functions:** are the functions which are declared in the C header files such as `scanf()`, `printf()`, `gets()`, `puts()`, `ceil()`, `floor()` etc.

**2.User-defined functions:** are the functions which are created by the C programmer, so that he/she can use it many times. It reduces the complexity of a big program and optimizes the code.



# Return Value

A C function may or may not return a value from the function. If you don't have to return any value from the function, use void for the return type.

## Example without return value:

```
void hello()  
{  
    printf("hello c");  
}
```

Example with return value:

```
int get()  
{  
  return 10;  
}
```

```
float get()  
{  
  return 10.2;  
}
```

## User-defined functions:

- function without arguments and without return value
- function without arguments and with return value
- function with arguments and without return value
- function with arguments and with return value

## Example for Function without argument and without return value

```
#include<stdio.h>
void printName();
void main ()
{
    printf("Hello ");
    printName();
}
void printName()
{
    printf("welcome");
}
```

```
#include<stdio.h>
void sum();
void main()
{
    printf("\ncalculating the sumof two numbers:");
    sum();
}
void sum()
{
    int a,b;
    printf("\nEnter two numbers");
    scanf("%d %d",&a,&b);
    printf("The sum is %d",a+b);
}
```

# Example for Function without argument and with return value

```
#include<stdio.h>
int sum();
void main()
{
    int result;
    printf("\nGoing to calculate the sum of two numbers:");
    result = sum();
    printf("%d",result);
}
int sum()
{
    int a,b;
    printf("\nEnter two numbers");
    scanf("%d %d",&a,&b);
    return a+b;
}
```

## Example for Function with argument and without return value

```
#include<stdio.h>
void sum(int, int);
void main()
{
    int a,b,result;
    printf("\nGoing to calculate the sum of two numbers:");
    printf("\nEnter two numbers:");
    scanf("%d %d",&a,&b);
    sum(a,b);
}
void sum(int a, int b)
{
    printf("\nThe sum is %d",a+b);
}
```

Example for Function with argument and with return value

```
#include<stdio.h>
int sum(int, int);
void main()
{
    int a,b,result;
    printf("\nGoing to calculate the sum of two numbers:");
    printf("\nEnter two numbers:");
    scanf("%d %d",&a,&b);
    result = sum(a,b);
    printf("\nThe sum is : %d",result);
}
int sum(int a, int b)
{
    return a+b;
}
```

# Recursion in C

Recursion is the process which comes into existence when a function calls a copy of itself to work on a smaller problem. Any function which calls itself is called recursive function, and such function calls are called recursive calls. Recursion involves several numbers of recursive calls. However, it is important to impose a termination condition of recursion. Recursion code is shorter than iterative code however it is difficult to understand.

For Example, recursion may be applied to sorting, searching, and traversal problems.

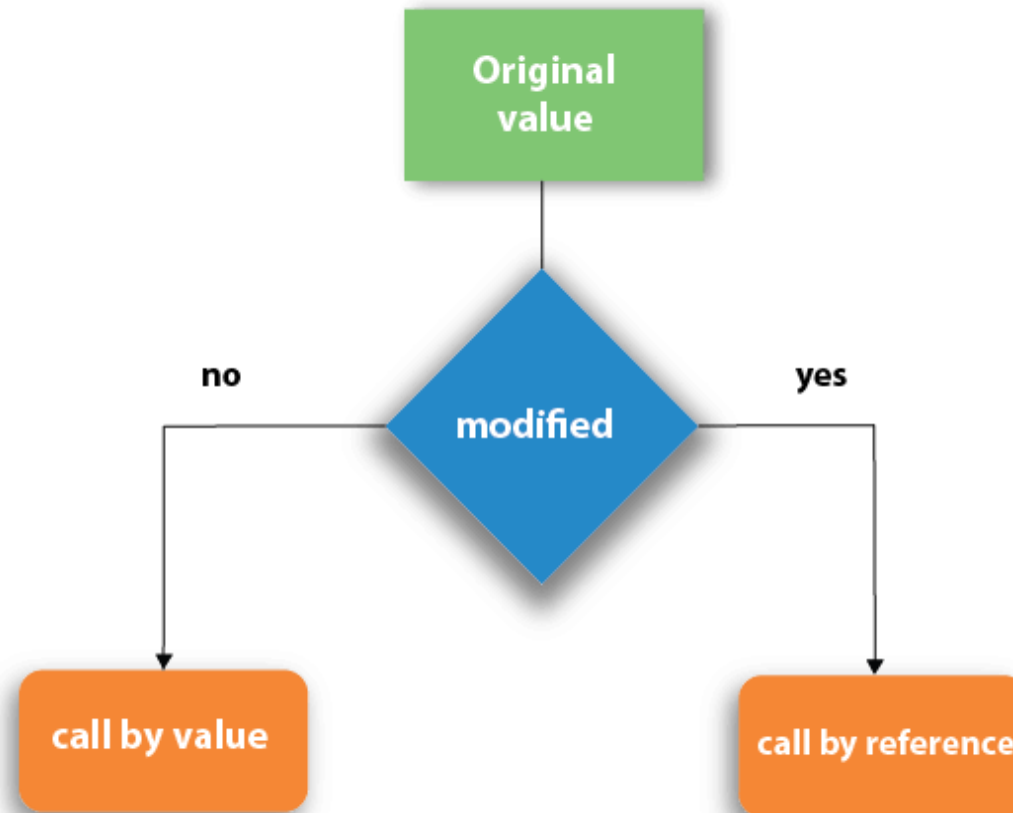


```
#include <stdio.h>
int fact (int);
int main()
{
    int n,f;
    printf("Enter the number whose factorial you want to calculate?");
    scanf("%d",&n);
    f = fact(n);
    printf("factorial = %d",f);
}
int fact(int n)
{
    if (n==0)
    {
        return 0;
    }
    else if ( n == 1)
    {
        return 1;
    }
    else
    {
        return n*fact(n-1);
    } }
}
```

```
#include<stdio.h>
int fibonacci(int);
void main ()
{
    int n,f;
    printf("Enter the value of n?");
    scanf("%d",&n);
    f = fibonacci(n);
    printf("%d",f);
}
int fibonacci (int n)
{
    if (n==0)
    {
        return 0;
    }
    else if (n == 1)
    {
        return 1;
    }
    else
    {
        return fibonacci(n-1)+fibonacci(n-2);
    }
}
```

# Call by value and Call by reference in C

There are two methods to pass the data into the function in C language, i.e., *call by value* and *call by reference*.



# Call by value in C

- In call by value method, the value of the actual parameters is copied into the formal parameters.
- In call by value method, we can not modify the value of the actual parameter by the formal parameter.
- In call by value, different memory is allocated for actual and formal parameters since the value of the actual parameter is copied into the formal parameter.
- The **actual parameter** is the argument which is used in the function call whereas **formal parameter** is the argument which is used in the function definition.

```
#include<stdio.h>
void change(int num) {
    printf("Before adding value inside function\n num=%d \n",num);
    num=num+100;
    printf("After adding value inside function\n num=%d \n", num);
}
int main() {
    int x=100;
    printf("Before function call x=%d \n", x);

    change(x);//passing value in function
    printf("After function call x=%d \n", x);
return 0;
}
```

```
#include <stdio.h>
void swap(int *, int *); //prototype of the function
int main()
{
    int a = 10;
    int b = 20;
    printf("Before swapping the values in main a = %d, b = %d\n",a,b); // printing the value of a and b in main

    swap(&a,&b);
    printf("After swapping values in main a = %d, b = %d\n",a,b); /*The values of actual parameters do change
                                                                    in call by reference, a = 10, b = 20 */
}
void swap (int *a, int *b)
{
    int temp;
    temp = *a;
    *a=*b;
    *b=temp;
    printf("After swapping values in function a = %d, b = %d\n",*a,*b); // Formal parameters, a = 20, b = 10
}
```

No.	Call by value	Call by reference
1	A copy of the value is passed into the function	An address of value is passed into the function
2	Changes made inside the function is limited to the function only. The values of the actual parameters do not change by changing the formal parameters.	Changes made inside the function validate outside of the function also. The values of the actual parameters do change by changing the formal parameters.
3	Actual and formal arguments are created at the different memory location	Actual and formal arguments are created at the same memory location

# C Pointers

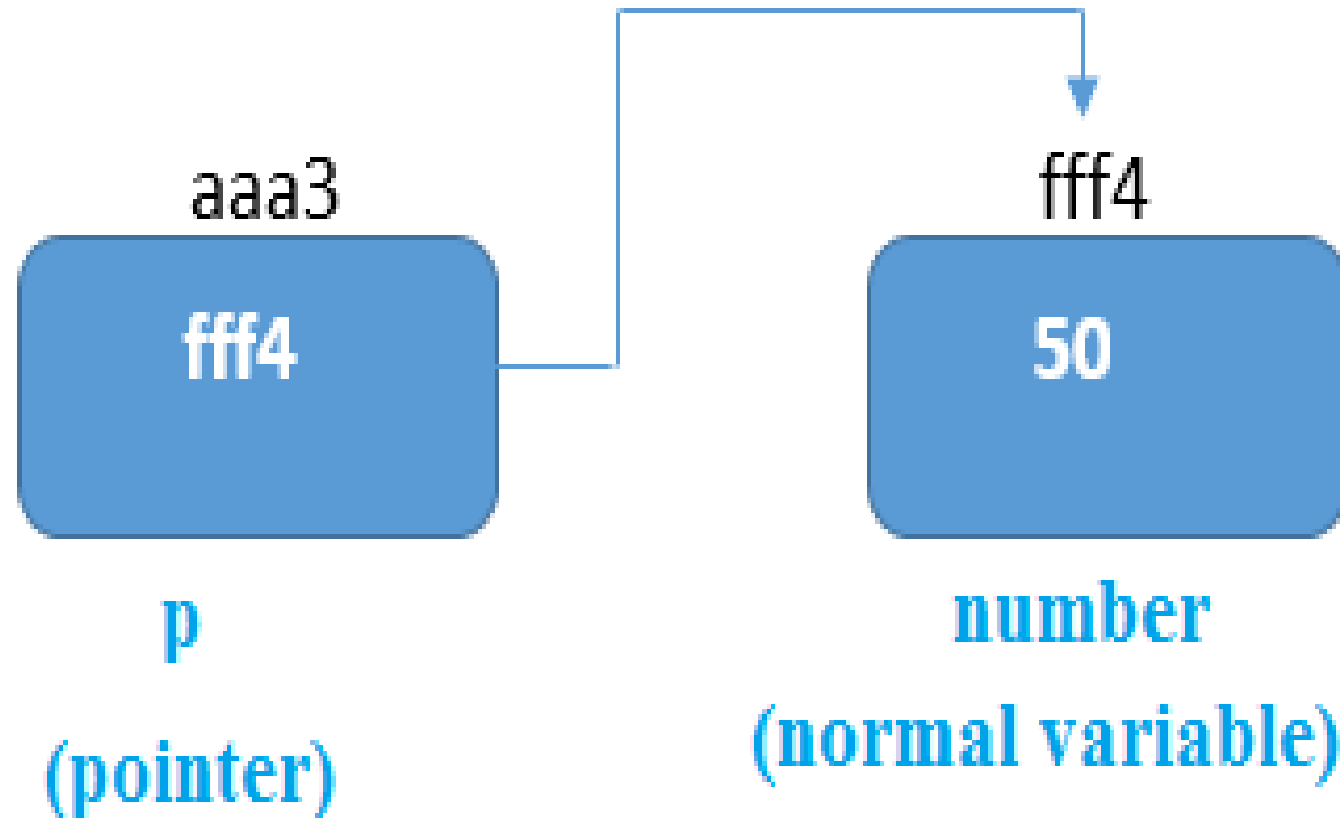
The pointer in C language is a variable which stores the address of another variable. This variable can be of type int, char, array, function, or any other pointer

## Declaring a pointer

The pointer in c language can be declared using \* (asterisk symbol). It is also known as indirection pointer used to dereference a pointer.

```
int *a;           //pointer to int  
char *c;          //pointer to char
```





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

```
int main()
```

```
{
```

```
int number=50;
```

```
int *p;
```

```
p=&number;//stores the address of number variable
```

```
printf("Address of p variable is %x \n",p);
```

```
/* p contains the address of the number therefore printing p gives the address of number */
```

```
printf("Value of p variable is %d \n",*p);
```

```
/*As we know that * is used to dereference a pointer therefore if we print *p, we will get the value stored at the address contained by p. */
```

```
return 0;
```

```
}
```

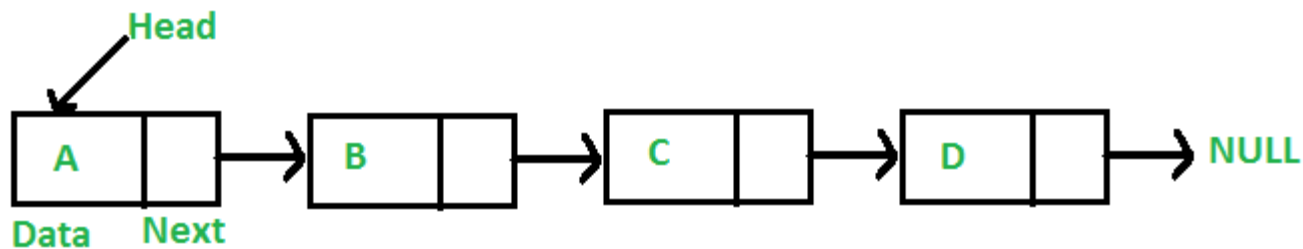




# Linked Lists

# LINKED LISTS

- 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 as shown in the below image:



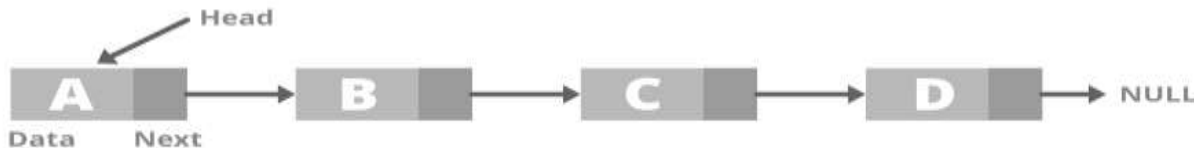
# Types of Linked List

- Singly Linked List:
- Circular Linked List:
- Doubly Linked List:

# Singly Linked List:

- It is the simplest type of linked list in which every node contains some data and a pointer to the next node of the same data type. The node contains a pointer to the next node means that the node stores the address of the next node in the sequence. A single linked list allows the traversal of data only in one way. Below is the image for the same:

**Singly Linked List**





# Introduction to linked lists

- Like arrays, Linked List is a linear data structure. Unlike arrays, linked list elements are not stored at a contiguous location; the elements are linked using pointers. They includes a series of connected nodes. Here, each node stores the data and the address of the next node.

# Why Linked List?

Arrays can be used to store linear data of similar types, but arrays have the following limitations.

**1.The size of the arrays is fixed**

**2.Insertion of a new element / Deletion of a existing element in an array of elements is expensive**

**For example**, in a system, if we maintain a sorted list of IDs in an array `id[]`.

`id[] = [1000, 1010, 1050, 2000, 2040]`.

And if we want to insert a new ID 1005, then to maintain the sorted order, we have to move all the elements after 1000 (excluding 1000).

Deletion is also expensive with arrays until unless some special techniques are used. For example, to delete 1010 in `id[]`, everything after 1010 has to be moved due to this so much work is being done which affects the efficiency of the code.

# Representation of LL

- A linked list is represented by a pointer to the first node of the linked list. The first node is called the head. If the linked list is empty, then the value of the head points to NULL.
- Each node in a list consists of at least two parts:
- A Data Item (we can store integer, strings or any type of data).
- Pointer (Or Reference) to the next node (connects one node to another) or An address of another node

## Algorithm for traversing a linked list

Step 1: [Initialize] Set Ptr = Start

Step 2: Repeat Steps 3 And 4 While Ptr != Null

Step 3: Apply Process To Ptr -> Data

Step 4: Set Ptr = Ptr -> Next

[End Of Loop]

Step 5: Exit

## Algorithm to print the number of nodes in a linked list

Step 1: [Initialize] Set Count = 0

Step 2: [Initialize] Set Ptr = Start

Step 3: Repeat steps 4 And 5 While Ptr != Null

Step 4: Set Count = +1

Step 5: Set Ptr = Ptr -> Next

[End Of Loop]

Step 6: Write Count

Step 7: Exit

## Algorithm to search a linked list

Step 1: [Initialize] Set Ptr = Start

Step 2: Repeat Step 3 While Ptr != Null

Step 3: If Val = Ptr ->Data

Set Pos = Ptr

Go To Step 5

Else

Set Ptr = Ptr -> Next

[End Of If]

[End Of Loop]

Step 4: Set Pos = Null

Step 5: Exit

# Inserting a New Node in a Linked List

Case 1: **The new node is inserted at the beginning.**

Case 2: The new node is inserted at the end.

Case 3: The new node is inserted after a given node.

Case 4: The new node is inserted before a given node.

Step 1: If Avail = Null

Write Overflow

Go To Step 7

[End Of If]

Step 2: Set New\_node = Avail

Step 3: Set Avail = Avail -> Next

Step 4: Set New\_node ->Data = Val

Step 5: Set New\_node -> Next = Start

Step 6: Set Start = New\_node

Step 7: Exit

## Algorithm new node is inserted at the end

```
Step 1: IF AVAIL = NULL
        Write OVERFLOW
        Go to Step 10
    [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = NULL
Step 6: SET PTR = START
Step 7: Repeat Step 8 while PTR -> NEXT != NULL
Step 8:     SET PTR = PTR -> NEXT
    [END OF LOOP]
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: EXIT
```



## Algorithm new node is inserted after a given node.

```
Step 1: IF AVAIL = NULL
        Write OVERFLOW
        Go to Step 12
    [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: SET PREPTR = PTR
Step 7: Repeat Steps 8 and 9 while PREPTR -> DATA != NUM
Step 8:     SET PREPTR = PTR
Step 9:     SET PTR = PTR -> NEXT
    [END OF LOOP]
Step 10: PREPTR -> NEXT = NEW_NODE
Step 11: SET NEW_NODE -> NEXT = PTR
Step 12: EXIT
```

## Algorithm new node is inserted before a given node.

```
Step 1: IF AVAIL = NULL
        Write OVERFLOW
        Go to Step 12
    [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: SET PREPTR = PTR
Step 7: Repeat Steps 8 and 9 while PTR -> DATA != NUM
Step 8:     SET PREPTR = PTR
Step 9:     SET PTR = PTR -> NEXT
    [END OF LOOP]
Step 10: PREPTR -> NEXT = NEW_NODE
Step 11: SET NEW_NODE -> NEXT = PTR
Step 12: EXIT
```

## Algorithm to delete the first node

```
Step 1: IF START = NULL
        Write UNDERFLOW
        Go to Step 5
    [END OF IF]
Step 2: SET PTR = START
Step 3: SET START = START -> NEXT
Step 4: FREE PTR
Step 5: EXIT
```

## Algorithm to delete the last node

```
Step 1: IF START = NULL
        Write UNDERFLOW
        Go to Step 8
    [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Steps 4 and 5 while PTR -> NEXT != NULL
Step 4:     SET PREPTR = PTR
Step 5:     SET PTR = PTR -> NEXT
    [END OF LOOP]
Step 6: SET PREPTR -> NEXT = NULL
Step 7: FREE PTR
Step 8: EXIT
```

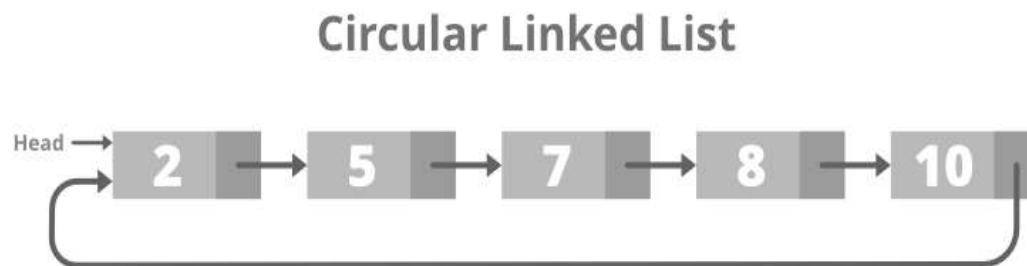
## Algorithm to delete the node after a given node

```
Step 1: IF START = NULL
        Write UNDERFLOW
        Go to Step 10
    [END OF IF]
Step 2: SET PTR = START
Step 3: SET PREPTR = PTR
Step 4: Repeat Steps 5 and 6 while PREPTR->DATA != NUM
Step 5:     SET PREPTR = PTR
Step 6:     SET PTR = PTR->NEXT
    [END OF LOOP]
Step 7: SET TEMP = PTR
Step 8: SET PREPTR->NEXT = PTR->NEXT
Step 9: FREE TEMP
Step 10: EXIT
```

# **CIRCULAR LINKED LISTS**

# Circular linked list

A circular linked list is that in which the last node contains the pointer to the first node of the list. While traversing a circular linked list, we can begin at any node and traverse the list in any direction forward and backward until we reach the same node we started. Thus, a circular linked list has no beginning and no end. Below is the image for the same:



# Inserting a New Node in a Circular Linked List

Case 1: The new node is inserted at the beginning of the circular linked list.

Case 2: The new node is inserted at the end of the circular linked list.

```
Step 1: IF AVAIL = NULL
        Write OVERFLOW
        Go to Step 11
    [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: Repeat Step 7 while PTR -> NEXT != START
Step 7:     PTR = PTR -> NEXT
    [END OF LOOP]
Step 8: SET NEW_NODE -> NEXT = START
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: SET START = NEW_NODE
Step 11: EXIT
```



## Algorithm new node is inserted at the end of the circular linked list

```
Step 1: IF AVAIL = NULL
        Write OVERFLOW
        Go to Step 10
    [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL → NEXT
Step 4: SET NEW_NODE → DATA = VAL
Step 5: SET NEW_NODE → NEXT = START
Step 6: SET PTR = START
Step 7: Repeat Step 8 while PTR → NEXT != START
Step 8:     SET PTR = PTR → NEXT
    [END OF LOOP]
Step 9: SET PTR → NEXT = NEW_NODE
Step 10: EXIT
```

## Algorithm to delete the first node

```
Step 1: IF START = NULL
        Write UNDERFLOW
        Go to Step 8
    [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR → NEXT != START
Step 4:     SET PTR = PTR → NEXT
    [END OF LOOP]
Step 5: SET PTR → NEXT = START → NEXT
Step 6: FREE START
Step 7: SET START = PTR → NEXT
Step 8: EXIT
```

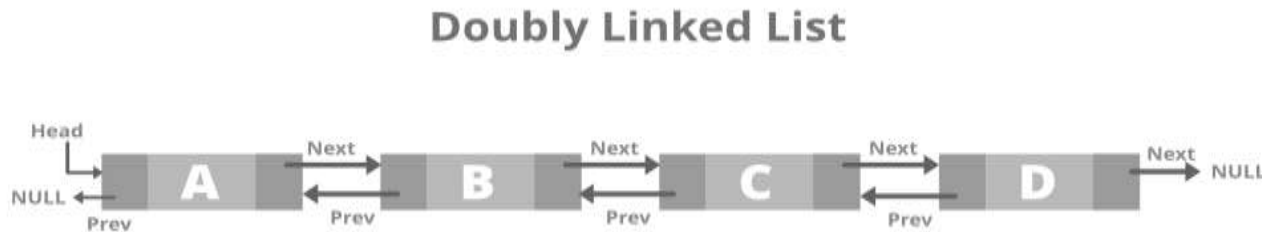
## Algorithm to delete the last node

```
Step 1: IF START = NULL
        Write UNDERFLOW
        Go to Step 8
    [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Steps 4 and 5 while PTR -> NEXT != START
Step 4:     SET PREPTR = PTR
Step 5:     SET PTR = PTR -> NEXT
    [END OF LOOP]
Step 6: SET PREPTR -> NEXT = START
Step 7: FREE PTR
Step 8: EXIT
```

# DOUBLY LINKED LISTS

# Doubly linked list

- A doubly linked list or a two-way linked list is a more complex type of linked list which contains a pointer to the next as well as the previous node in sequence. Therefore, it contains three parts: data, a pointer to the next node, and a pointer to the previous node. This would enable us to traverse the list in the backward direction as well. Below is the image for the same:



## Algorithm to insert a new node at the beginning

```
Step 1: IF AVAIL = NULL
        Write OVERFLOW
        Go to Step 9
    [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> PREV = NULL
Step 6: SET NEW_NODE -> NEXT = START
Step 7: SET START -> PREV = NEW_NODE
Step 8: SET START = NEW_NODE
Step 9: EXIT
```

## Algorithm to insert a new node at the end

```
Step 1: IF AVAIL = NULL
        Write OVERFLOW
        Go to Step 11
    [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL → NEXT
Step 4: SET NEW_NODE → DATA = VAL
Step 5: SET NEW_NODE → NEXT = NULL
Step 6: SET PTR = START
Step 7: Repeat Step 8 while PTR → NEXT != NULL
Step 8:     SET PTR = PTR → NEXT
    [END OF LOOP]
Step 9: SET PTR → NEXT = NEW_NODE
Step 10: SET NEW_NODE → PREV = PTR
Step 11: EXIT
```



## Algorithm to insert a new node after the given node

```
Step 1: IF AVAIL = NULL
        Write OVERFLOW
        Go to Step 12
    [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL → NEXT
Step 4: SET NEW_NODE → DATA = VAL
Step 5: SET PTR = START
Step 6: Repeat Step 7 while PTR → DATA != NUM
Step 7:     SET PTR = PTR → NEXT
    [END OF LOOP]
Step 8: SET NEW_NODE → NEXT = PTR → NEXT
Step 9: SET NEW_NODE → PREV = PTR
Step 10: SET PTR → NEXT = NEW_NODE
Step 11: SET PTR → NEXT → PREV = NEW_NODE
Step 12: EXIT
```



## Algorithm to insert a new node before the given node

```
Step 1: IF AVAIL = NULL
        Write OVERFLOW
        Go to Step 12
    [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL → NEXT
Step 4: SET NEW_NODE → DATA = VAL
Step 5: SET PTR = START
Step 6: Repeat Step 7 while PTR → DATA != NUM
Step 7:     SET PTR = PTR → NEXT
    [END OF LOOP]
Step 8: SET NEW_NODE → NEXT = PTR
Step 9: SET NEW_NODE → PREV = PTR → PREV
Step 10: SET PTR → PREV = NEW_NODE
Step 11: SET PTR → PREV → NEXT = NEW_NODE
Step 12: EXIT
```

# Deleting a Node from a Doubly Linked List

- Case 1: The first node is deleted.
- Case 2: The last node is deleted.
- Case 3: The node after a given node is deleted.
- Case 4: The node before a given node is deleted.

## Algorithm to delete the first node

Step 1: IF START = NULL

    Write UNDERFLOW

    Go to Step 6

    [END OF IF]

Step 2: SET PTR = START

Step 3: SET START = START → NEXT

Step 4: SET START → PREV = NULL

Step 5: FREE PTR

Step 6: EXIT

## Algorithm to delete the last node

```
Step 1: IF START = NULL
        Write UNDERFLOW
        Go to Step 7
    [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR -> NEXT != NULL
Step 4:     SET PTR = PTR -> NEXT
    [END OF LOOP]
Step 5: SET PTR -> PREV -> NEXT = NULL
Step 6: FREE PTR
Step 7: EXIT
```

## Algorithm to delete a node after a given node

```
Step 1: IF START = NULL
        Write UNDERFLOW
        Go to Step 9
    [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR → DATA != NUM
Step 4:     SET PTR = PTR → NEXT
    [END OF LOOP]
Step 5: SET TEMP = PTR → NEXT
Step 6: SET PTR → NEXT = TEMP → NEXT
Step 7: SET TEMP → NEXT → PREV = PTR
Step 8: FREE TEMP
Step 9: EXIT
```

## Algorithm to delete a node before a given node

```
Step 1: IF START = NULL
        Write UNDERFLOW
        Go to Step 9
    [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR → DATA != NUM
Step 4:     SET PTR = PTR → NEXT
    [END OF LOOP]
Step 5: SET TEMP = PTR → PREV
Step 6: SET TEMP → PREV → NEXT = PTR
Step 7: SET PTR → PREV = TEMP → PREV
Step 8: FREE TEMP
Step 9: EXIT
```

# **CIRCULAR DOUBLY LINKED LISTS**



## Algorithm to insert a new node at the beginning

```
Step 1: IF AVAIL = NULL
        Write OVERFLOW
        Go to Step 13
    [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL → NEXT
Step 4: SET NEW_NODE → DATA = VAL
Step 5: SET PTR = START
Step 6: Repeat Step 7 while PTR → NEXT != START
Step 7:     SET PTR = PTR → NEXT
    [END OF LOOP]
Step 8: SET PTR → NEXT = NEW_NODE
Step 9: SET NEW_NODE → PREV = PTR
Step 10: SET NEW_NODE → NEXT = START
Step 11: SET START → PREV = NEW_NODE
Step 12: SET START = NEW_NODE
Step 13: EXIT
```



## Algorithm to insert a new node at the end

```
Step 1: IF AVAIL = NULL
        Write OVERFLOW
        Go to Step 12
    [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL → NEXT
Step 4: SET NEW_NODE → DATA = VAL
Step 5: SET NEW_NODE → NEXT = START
Step 6: SET PTR = START
Step 7: Repeat Step 8 while PTR → NEXT != START
Step 8:     SET PTR = PTR → NEXT
    [END OF LOOP]
Step 9: SET PTR → NEXT = NEW_NODE
Step 10: SET NEW_NODE → PREV = PTR
Step 11: SET START → PREV = NEW_NODE
Step 12: EXIT
```

## Algorithm to delete the first node

```
Step 1: IF START = NULL
        Write UNDERFLOW
        Go to Step 8
    [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR → NEXT != START
Step 4:     SET PTR = PTR → NEXT
    [END OF LOOP]
Step 5: SET PTR → NEXT = START → NEXT
Step 6: SET START → NEXT → PREV = PTR
Step 7: FREE START
Step 8: SET START = PTR → NEXT
```

## Algorithm to delete the last node

Step 1: IF START = NULL

    Write UNDERFLOW

    Go to Step 8

    [END OF IF]

Step 2: SET PTR = START

Step 3: Repeat Step 4 while PTR → NEXT != START

Step 4:     SET PTR = PTR → NEXT

    [END OF LOOP]

Step 5: SET PTR → PREV → NEXT = START

Step 6: SET START → PREV = PTR → PREV

Step 7: FREE PTR

Step 8: EXIT

# QUEUES

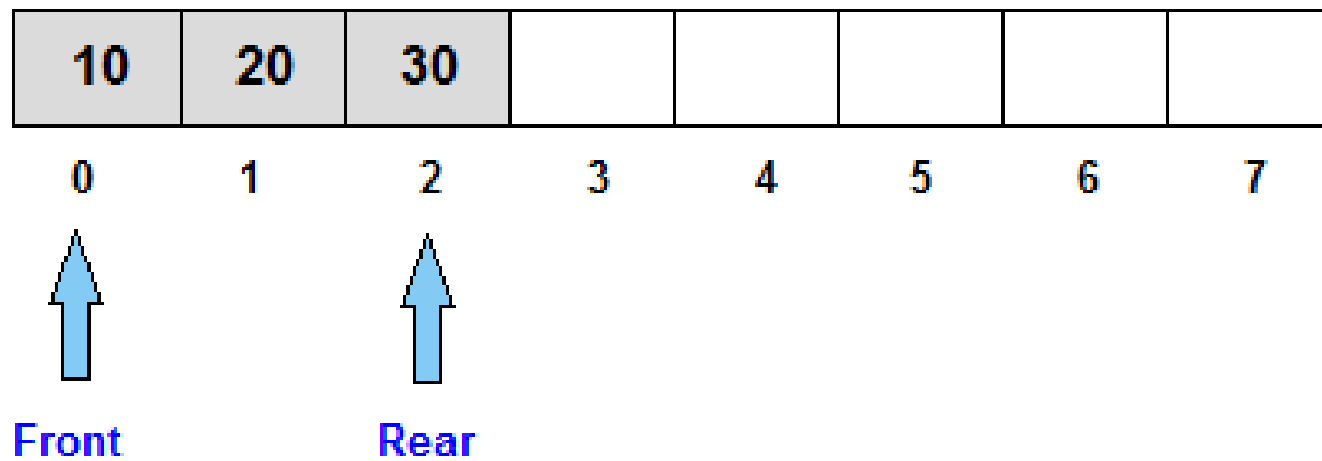


**Real Life Example of Queue : Library Counter**

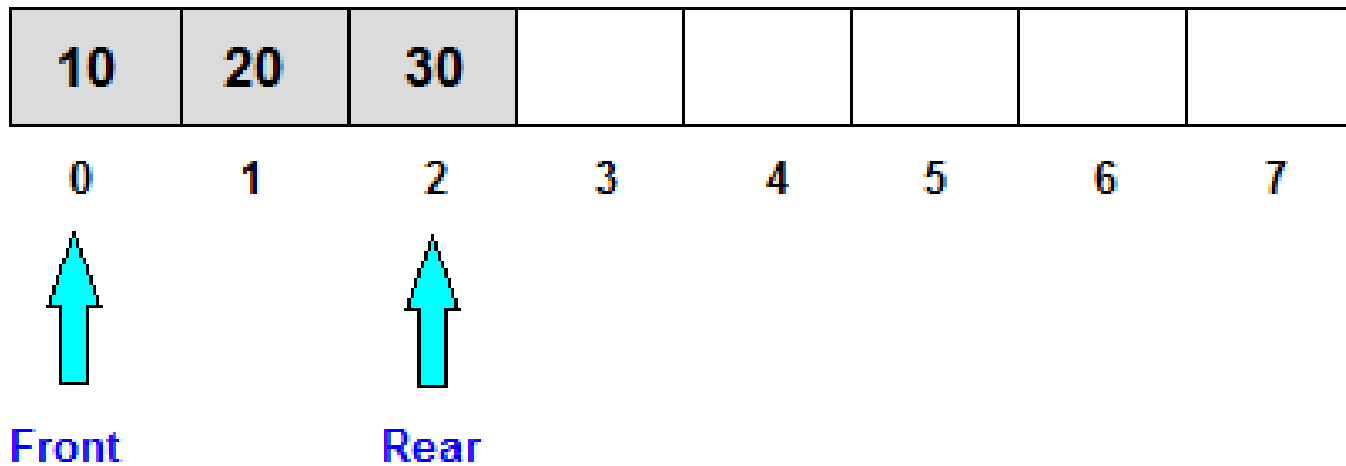
# Introduction

- Queue is an important data structure which stores its elements in an ordered manner.
- We can explain the concept of queues using the following analogy:  
*People moving on an escalator. The people who got on the escalator first will be the first one to step out of it.*
- A queue is a FIFO (First-In, First-Out) data structure in which the element that is inserted first is the first one to be taken out.
- The elements in a queue are added at one end called the *rear* and removed from the other one end called the *front*.

## INSERTING AN ELEMENT



## DELETING A ELEMENT

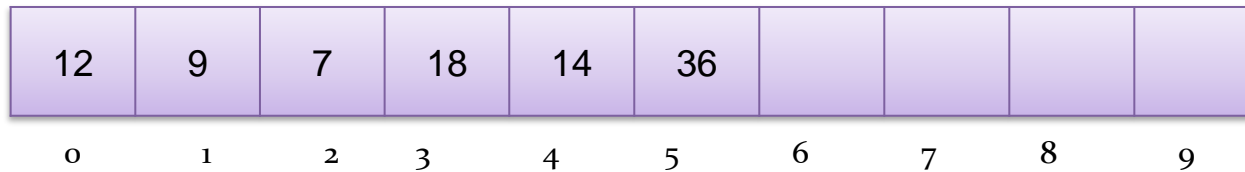




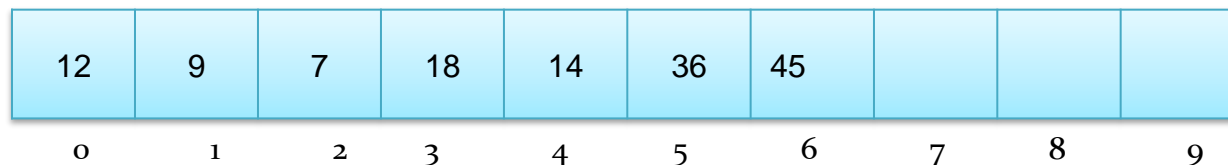
# Array Representation of Queues

# Array Representation of Queues

- Queues can be easily represented using linear arrays.
- Every queue has front and rear variables that point to the position from where deletions and insertions can be done, respectively.
- Consider the queue shown in figure

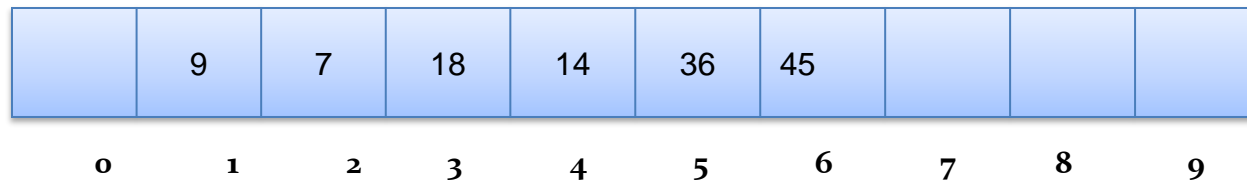


- Here, front = 0 and rear = 5.
- If we want to add one more value in the list say with value 45, then rear would be incremented by 1 and the value would be stored at the position pointed by rear.



# Array Representation of Queues

- Now,  $\text{front} = 0$  and  $\text{rear} = 6$ . Every time a new element has to be added, we will repeat the same procedure.
- Now, if we want to delete an element from the queue, then the value of  $\text{front}$  will be incremented. Deletions are done from only this end of the queue.



- Now,  $\text{front} = 1$  and  $\text{rear} = 6$ .

# Array Representation of Queues

- Before inserting an element in the queue we must check for overflow conditions.
- An overflow occurs when we try to insert an element into a queue that is already full, i.e. when  $\text{rear} = \text{MAX} - 1$ , where MAX specifies the maximum number of elements that the queue can hold.
- Similarly, before deleting an element from the queue, we must check for underflow condition.
- An underflow occurs when we try to delete an element from a queue that is already empty. If  $\text{front} = -1$  and  $\text{rear} = -1$ , this means there is no element in the queue.

# Algorithm for Insertion Operation

**Step 1: IF REAR=MAX-1, then;**

**Write OVERFLOW**

**Go to Step 4**

**[END OF IF]**

**Step 2: IF FRONT == -1 and REAR = -1, then**

**SET FRONT = REAR = 0**

**ELSE**

**SET REAR = REAR + 1**

**[END OF IF]**

**Step 3: SET QUEUE[REAR] = NUM**

**Step 4: Exit**

# Algorithm for Deletion Operation

**Step 1: IF FRONT = -1 OR FRONT > REAR, then**

**Write UNDERFLOW**

**Goto Step 2**

**ELSE**

**SET VAL = QUEUE[FRONT]**

**SET FRONT = FRONT + 1**

**[END OF IF]**

**Step 2: Exit**

# LINKED LIST

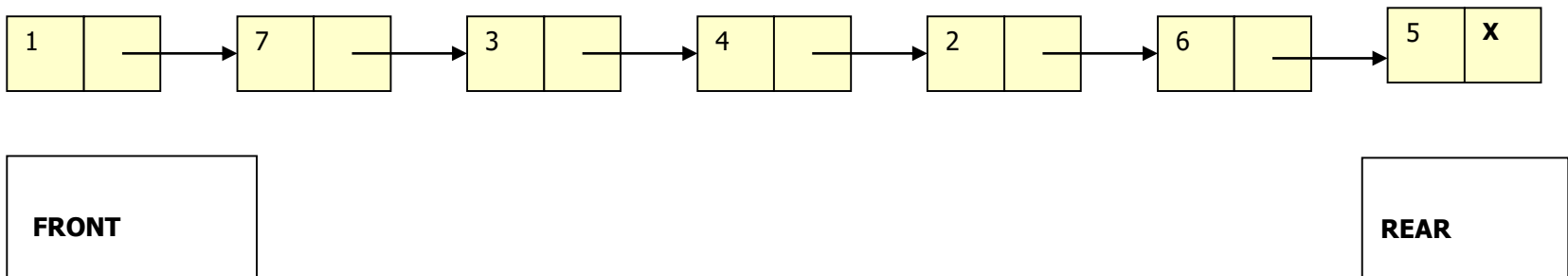
*Representation*

of

Queues

# Linked Representation of Queues

- In a linked queue, every element has two parts: one that stores data and the other that stores the address of the next element.
- The START pointer of the linked list is used as FRONT.
- We will also use another pointer called REAR which will store the address of the last element in the queue.
- All insertions will be done at the rear end and all the deletions will be done at the front end.
- If  $\text{FRONT} = \text{REAR} = \text{NULL}$ , then it indicates that the queue is empty.





# Inserting an Element in a Linked Queue

**Step 1: Allocate memory for the new node and name it as PTR**

**Step 2: SET PTR->DATA = VAL**

**Step 3: IF FRONT = NULL, then**

**SET FRONT = REAR = PTR**

**SET FRONT->NEXT = REAR->NEXT = NULL**

**ELSE**

**SET REAR->NEXT = PTR**

**SET REAR = PTR**

**SET REAR->NEXT = NULL**

**[END OF IF]**

**Step 4: END**

# Deleting an Element from a Linked Queue

Step 1: IF FRONT = NULL, then

Write “Underflow”

Go to Step 5

[END OF IF]

Step 2: SET PTR = FRONT

Step 3: FRONT = FRONT->NEXT

Step 4: FREE PTR

Step 5: END

# Circular Queues

		7	18	14	36	45	21	99	72
0	1	2	3	4	5	6	7	8	9

- We will explain the concept of circular queues using an example.
- In this queue, front = 2 and rear = 9.
- Now, if you want to insert a new element, it cannot be done because the space is available only at the left of the queue.
- If  $\text{rear} = \text{MAX} - 1$ , then OVERFLOW condition exists.
- This is the major drawback of a linear queue. Even if space is available, no insertions can be done once rear is equal to  $\text{MAX} - 1$ .
- This leads to wastage of space. In order to overcome this problem, we use circular queues.
- In a circular queue, the first index comes right after the last index.
- A circular queue is full, only when  $\text{front}=0$  and  $\text{rear} = \text{Max} - 1$ .

# Inserting an Element in a Circular Queue

- For insertion we check for three conditions which are as follows:
  - If  $\text{front}=0$  and  $\text{rear}=\text{MAX}-1$ , then the circular queue is full.

90	49	7	18	14	36	45	21	99	72
front=0	1	2	3	4	5	6	7	8	rear=9

- If  $\text{rear} \neq \text{MAX}-1$ , then the rear will be incremented and value will be inserted

90	49	7	18	14	36	45	21	99	
front=0	1	2	3	4	5	6	7	rear=8	9

- If  $\text{front} \neq 0$  and  $\text{rear}=\text{MAX}-1$ , then it means that the queue is not full. So, set  $\text{rear}=0$  and insert the new element.

	49	7	18	14	36	45	21	99	72
front=1	2	3	4	5	6	7	8	rear=9	

# Algorithm to Insert an Element in a Circular Queue

Step 1: IF  $\text{FRONT} = 0$  and  $\text{Rear} = \text{MAX} - 1$ , then

Write "OVERFLOW"

Goto Step 4

[END OF IF]

Step 2: IF  $\text{FRONT} = -1$  and  $\text{REAR} = -1$ , then;

SET  $\text{FRONT} = \text{REAR} = 0$

ELSE IF  $\text{REAR} = \text{MAX} - 1$  and  $\text{FRONT} \neq 0$

SET  $\text{REAR} = 0$

ELSE

SET  $\text{REAR} = \text{REAR} + 1$

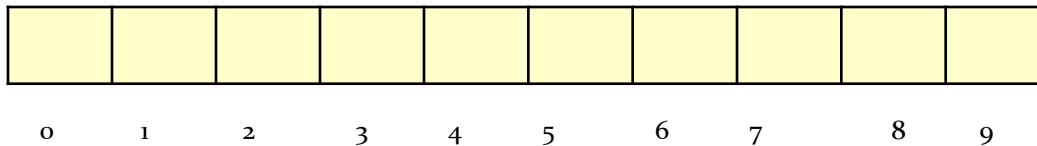
[END OF IF]

Step 3: SET  $\text{QUEUE}[\text{REAR}] = \text{VAL}$

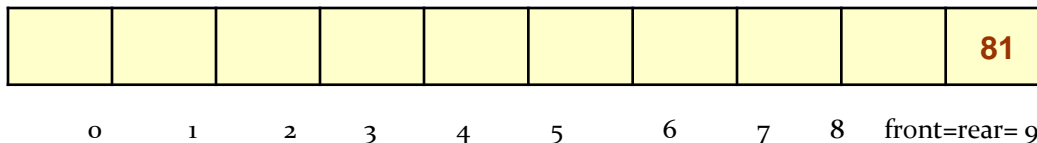
Step 4: Exit

# Deleting an Element from a Circular Queue

- To delete an element again we will check for three conditions:
  - If  $\text{front} = -1$ , then it means there are no elements in the queue. So an underflow condition will be reported.

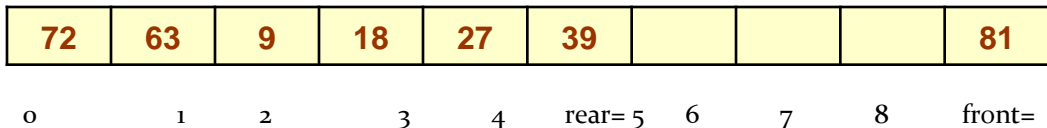


- If the queue is not empty and after returning the value on front, if  $\text{front} = \text{rear}$ , then it means now the queue has become empty and so front and rear are set to -1.



**Delete this element and set  
rear = front = -1**

- If the queue is not empty and after returning the value on front, if  $\text{front} = \text{MAX} - 1$ , then front is set to 0.



## Algorithm to Delete an Element from a Circular Queue

```
Step 1: IF FRONT = -1, then
        Write "Underflow"
        Goto Step 4
    [END OF IF]
Step 2: SET VAL = QUEUE[FRONT]
Step 3: IF FRONT = REAR
        SET FRONT = REAR = -1
    ELSE
        IF FRONT = MAX - 1
            SET FRONT = 0
        ELSE
            SET FRONT = FRONT + 1
        [END OF IF]
    [END OF IF]
Step 4: EXIT
```

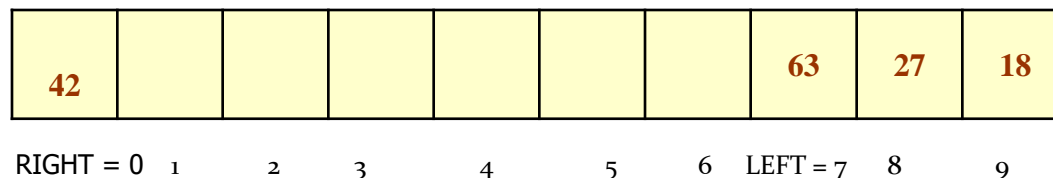
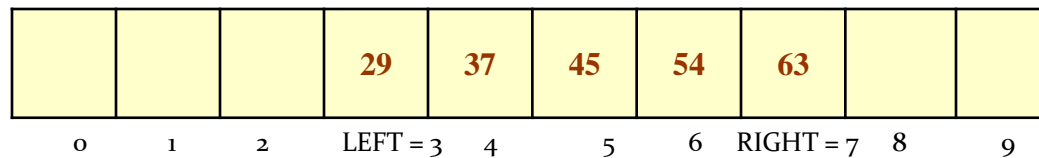
# Dequeues

- A deque is a list in which elements can be inserted or deleted at either end.
- It is also known as a head-tail linked list because elements can be added to or removed from the front (head) or back (tail).
- A deque can be implemented either using a circular array or a circular doubly linked list.
- In a deque, two pointers are maintained, LEFT and RIGHT which point to either end of the deque.
- The elements in a deque stretch from LEFT end to the RIGHT and since it is circular, Dequeue[N-1] is followed by Dequeue[0].



# Dequeues

- There are two variants of a double-ended queue:
  - *Input restricted deque*: In this dequeue insertions can be done only at one of the ends while deletions can be done from both the ends.
  - *Output restricted deque*: In this dequeue deletions can be done only at one of the ends while insertions can be done on both the ends.



# Priority Queues

- A priority queue is a queue in which each element is assigned a priority.
- The priority of elements is used to determine the order in which these elements will be processed.
- The general rule of processing elements of a priority queue can be given as:
  - An element with higher priority is processed before an element with lower priority
  - Two elements with same priority are processed on a first come first served (FCFS) basis
- Priority queues are widely used in operating systems to execute the highest priority process first.
- In computer's memory priority queues can be represented using arrays or linked lists.

# Array Representation of Priority Queues

- When arrays are used to implement a priority queue, then a separate queue for each priority number is maintained.
- Each of these queues will be implemented using circular arrays or circular queues. Every individual queue will have its own FRONT and REAR pointers.
- We can use a two-dimensional array for this purpose where each queue will be allocated same amount of space.
- Given the front and rear values of each queue, a two dimensional matrix can be formed.

FRONT	REAR
3	3
1	3
4	5
4	1

	1	2	3	4	5
1			A		
2	B	C	D		
3				E	F
4	I			G	H

Priority queue matrix

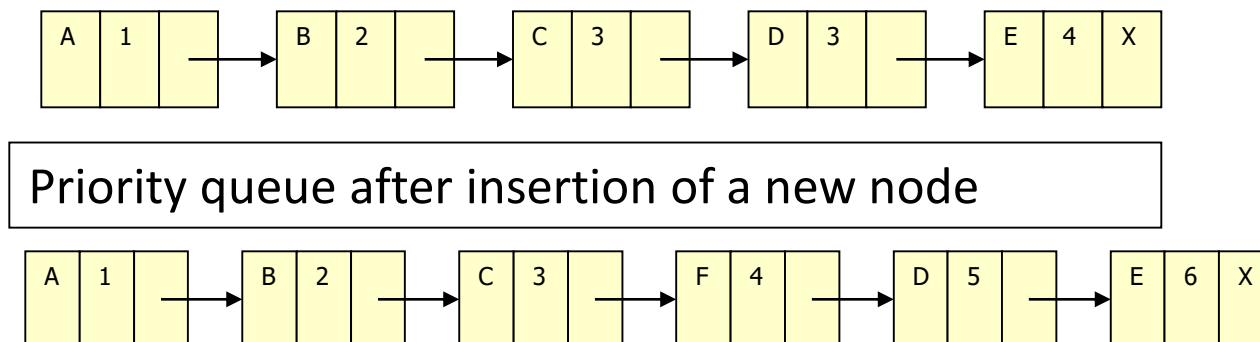
FRONT	REAR
3	3
1	3
4	1
4	1

	1	2	3	4	5
1			A		
2	B	C	D		
3	R			E	F
4	I			G	H

Priority queue matrix after insertion of a new element

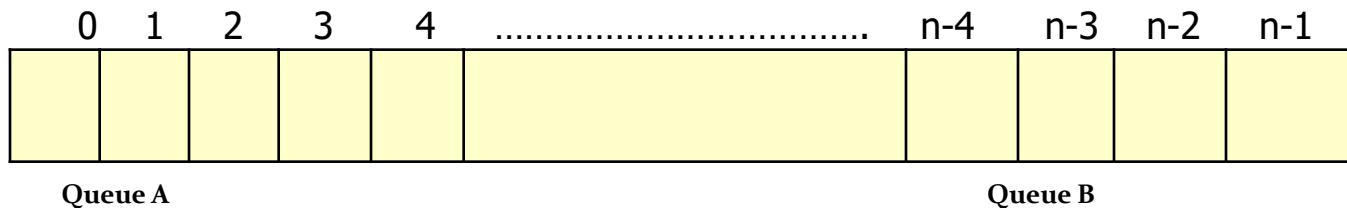
# Linked Representation of Priority Queues

- When a priority queue is implemented using a linked list, then every node of the list contains three parts: (1) the information or data part, (ii) the priority number of the element, (iii) and address of the next element.
- If we are using a sorted linked list, then element having higher priority will precede the element with lower priority.



# Multiple Queues

- When implementing a queue using an array, the size of the array must be known in advance.
- If the queue is allocated less space, then frequent OVERFLOW conditions will be encountered.
- To deal with this problem, the code will have to be modified to reallocate more space for the array, but this results in sheer wastage of memory. Thus, there lies a tradeoff between the frequency of overflows and the space allocated.
- A better solution to deal with this problem is to have multiple queues or to have more than one queue in the same array.
- One important point to note is that while queue A will grow from left to right, the queue B on the same time will grow from right to left.



# Applications of Queues

- Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.
- Queues are used to transfer data asynchronously e.g., pipes, file IO, sockets.
- Queues are used as buffers on MP3 players and portable CD players, iPod playlist.
- Queues are used in Playlist for jukebox to add songs to the end, play from the front of the list.
- Queues are used in OS for handling interrupts. When programming a real-time system that can be interrupted, for ex, by a mouse click, it is necessary to process the interrupts immediately before proceeding with the current job. If the interrupts have to be handled in the order of arrival, then a FIFO queue is the appropriate data structure