DATA STRUCTURES

Data

Data is derived from a Latin word “Datum” which means collection. So data can be defined as collection of facts and figures.

Data is classified into two types.

1) Group data Item
2) Elementary data Item

1) Group Data Item

Data item that can be subdivided into different segments is called group data item.

For example, name is a group data item because it can be subdivided into different segments i.e. First name, middle name, last name

2) Elementary Data Item

The data item that cannot be subdivided into different segments is called elementary data item

For example, Account No, ID Number etc.

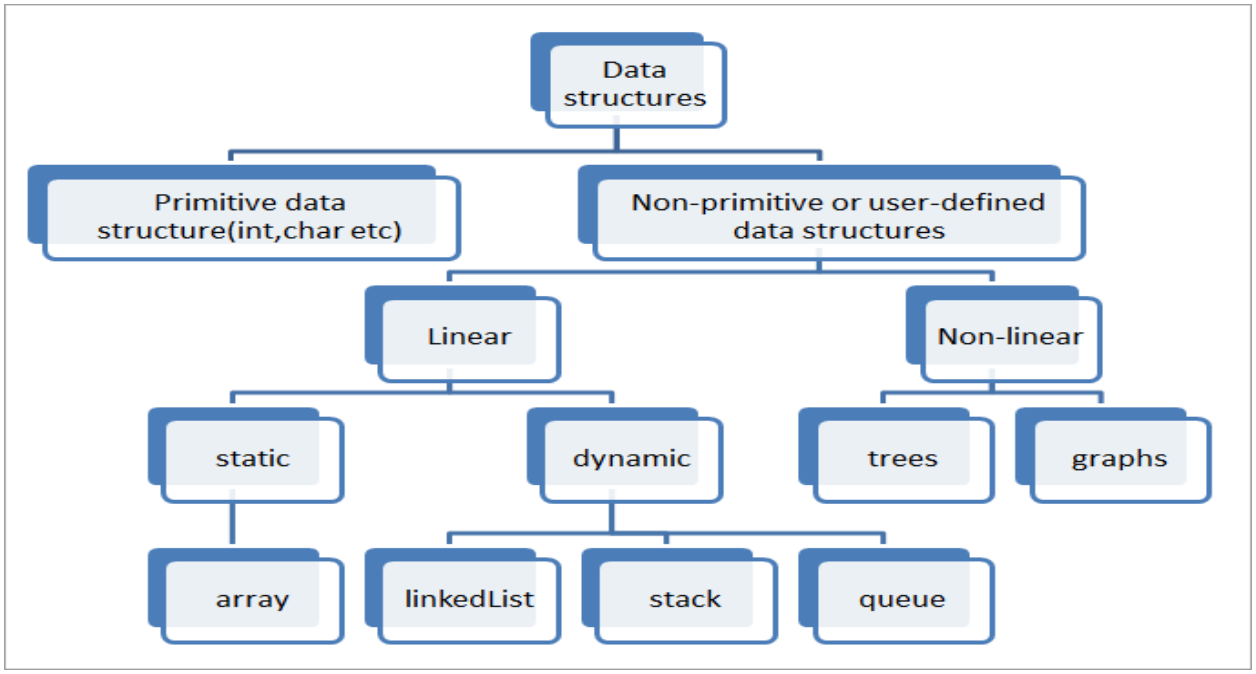
Data Structure

Data structure is a mathematical way for storage of data in the computer memory.

It is the way of storing and accessing the data from the computer memory. So that large number of data is processed in small interval of time.

OR

The organized collection of data is called a ‘Data Structure’.



Primitive Data structures are directly supported by the language ie; any operation is directly performed in these data items.
Ex: integer, Character, Real numbers etc.
Non-primitive data types are not defined by the programming language, but are instead created by the programmer.

Linear data structures organize their data elements in a linear fashion, where data elements are attached one after the other. Linear data structures are very easy to implement, since the memory of the computer is also organized in a linear fashion. Some commonly used linear data structures are arrays, linked lists, stacks and queues.

In nonlinear data structures, data elements are not organized in a sequential fashion. Data structures like trees, graphs, tables and sets are some examples of widely used nonlinear data structures.

Operations on the Data Structures:

Following operations can be performed on the data structures:
1. Traversing
2. Searching
3. Inserting
4. Deleting
5. Sorting
6. Merging

1. Traversing- It is used to access each data item exactly once so that it can be processed.
2. Searching- It is used to find out the location of the data item if it exists in the given collection of data items.

3. Inserting- It is used to add a new data item in the given collection of data items.
4. Deleting- It is used to delete an existing data item from the given collection of data items.
5. Sorting- It is used to arrange the data items in some order i.e. in ascending or descending order in case of numerical data and in dictionary order in case of alphanumeric data.
6. Merging- It is used to combine the data items of two sorted files into single file in the sorted form.

Algorithm

Algorithm is derived from the name of Muslim scientist Abu Jafir Muhammad Iban-e-musa Al-Khurzami.
Algorithm is not only the base of data structure, but it is also the base of good programming.

Definition: - An algorithm is a ***Step By Step*** process to solve a problem, where each step indicates an intermediate task. Algorithm contains finite number of steps that leads to the solution of the problem.

Properties /Characteristics of an Algorithm:-

Algorithm has the following basic properties

- **Input-Output:-** Algorithm takes '0' or more input and produces the required output. This is the basic characteristic of an algorithm.
- **Finiteness:-** An algorithm must terminate in countable number of steps.
- **De-fi-niteness:** Each step of an algorithm must be stated clearly and unambiguously.
- **Effectiveness:** Each and every step in an algorithm can be converted in to programming language statement.
- **Generality:** Algorithm is generalized one. It works on all set of inputs and provides the required output. In other words, it is not restricted to a single input value.

Algorithm Notation/ Parts of Algorithm

To write any algorithm these notations are used usually.

1)	Name of algorithm	2)	Introductory Comments
3)	Comments	4)	Steps
5)	Variable handling	6)	Control
7)	Assignment Statements	8)	Input and Output Statements
9)	Exit		

1. Name of Algorithm

Each algorithm has a name related with subject. The name of the algorithm is always written a capital latter in very first line of algorithm.

Example:

Algorithm SUM (a,b,c)

2. Introductory Comments

In this part of algorithm, the whole purpose of problem is to be mentioned. These are always enclosed in square brackets.

Example:

[The algorithm is used for the addition of three numbers]

3. Comments

These are always enclosed in square brackets. Here the purpose is to describe each step.

Example:

[Assign value to num1]

4. **Steps**

In this part, the programming instructions are used.

Example:

Num □ 5

Or *Num=5*

Or *Read Num*

5. **Variable Handling**

The variable name should be a valid identifier. The name of variable may be upper case or lower-case letter.

6. **Control**

The execution of steps is performed in sequential manner.

7. **Assignment Statement**

By using assignment statement, the value is assign to any variable.

The "**=**" , "**=:**" , "□" are used for assignment.

8. **Input and Output Statements**

Input statement is used to input the data for variables.

Its **Syntax** is

Read var1,var2,……. var-n

Or *Input Var1,var2,……. var-n*

Output statement is used to display the result. Its **syntax** is

Print *Message/ Variable*

Or ***Write*** *Message/ Variable*

9. **Exit**

When all the statements are executed then the algorithm terminates.

Its **syntax** is

Exit

Categories of Algorithm:

Based on the different types of steps in an Algorithm, it can be divided into three categories, namely

- Sequence
- Selection and
- Iteration

Sequence: The steps described in an algorithm are performed successively one by one without skipping any step. The sequence of steps defined in an algorithm should be simple and easy to understand. Each instruction of such an algorithm is executed, because no selection procedure or conditional branching exists in a sequence algorithm.

Example:
// adding two numbers
Step 1: start
Step 2: read a,b
Step 3: Sum=a+b
Step 4: write Sum
Step 5: stop

Selection: The sequence type of algorithms are not sufficient to solve the problems, which involves decision and conditions. In order to solve the problem which involve decision making or option selection, we go for Selection type of algorithm. The general format of Selection type of statement is as shown below:

if(condition)
Statement-1;
else
Statement-2;

The above syntax specifies that if the condition is true, statement-1 will be executed otherwise statement-2 will be executed. In case the operation is unsuccessful. Then sequence of algorithm should be changed/ corrected in such a way that the system will re-execute until the operation is successful.

Example1:
// Person eligibility for vote
Step 1 : start
Step 2 : read age
Step 3 : if age > = 18 then step_4 else step_5
Step 4 : write "person is eligible for vote"
Step 5 : write " person is not eligible for vote"
Step 6 : stop

Example2:
// biggest among two numbers
Step 1 : start
Step 2 : read a,b
Step 3 : if a > b then
Step 4 : write "a is greater than b"
Step 5 : else
Step 6 : write "b is greater than a"
Step 7 : stop

Iteration: Iteration type algorithms are used in solving the problems which involves repetition of statement. In this type of algorithms, a particular number of statements are repeated 'n' no. of times.

Example1:
Step 1 : start
Step 2 : read n
Step 3 : repeat step 4 until n>0
Step 4 : (a) r=n mod 10
(b) s=s+r
(c) n=n/10
Step 5 : write s
Step 6 : stop

Performance Analysis an Algorithm:

The Efficiency of an Algorithm can be measured by the following metrics.

i. Time Complexity and

ii. Space Complexity.

i.Time Complexity:

The amount of time required for an algorithm to complete its execution is its time complexity. An algorithm is said to be efficient if it takes the minimum (reasonable) amount of time to complete its execution.

ii. Space Complexity:

The amount of space occupied by an algorithm is known as Space Complexity. An algorithm is said to be efficient if it occupies less space and required the minimum amount of time to complete its execution.

1.Write an algorithm for roots of a Quadratic Equation?

// Roots of a quadratic Equation
Step 1 : start
Step 2 : read a,b,c
Step 3 : if (a= 0) then step 4 else step 5
Step 4 : Write " Given equation is a linear equation "
Step 5 : d=(b * b) _ (4 *a *c)
Step 6 : if (d>0) then step 7 else step8
Step 7 : Write " Roots are real and Distinct"
Step 8: if(d=0) then step 9 else step 10
Step 9: Write "Roots are real and equal"
Step 10: Write " Roots are Imaginary"
Step 11: stop

2. Write an algorithm to find the largest among three different numbers entered by user

Step 1: Start
Step 2: Declare variables a,b and c.
Step 3: Read variables a,b and c.
Step 4: If a>b
If a>c
Display a is the largest number.
Else
Display c is the largest number.
Else
If b>c
Display b is the largest number.
Else
Display c is the greatest number.
Step 5: Stop

3.Write an algorithm to find the factorial of a number entered by user.

Step 1: Start
Step 2: Declare variables n,factorial and i.

Step 3: Initialize variables
factorial←1
i←1
Step 4: Read value of n
Step 5: Repeat the steps until i=n
5.1: factorial←factorial*i
5.2: i←i+1
Step 6: Display factorial
Step 7: Stop

Asymptotic Analysis *In Asymptotic Analysis, we* ***evaluate the performance of an algorithm in terms of input size*** *(we don't measure the actual running time). We calculate, how the time (or space) taken by an algorithm increases with the input size.*

Using asymptotic analysis, we can very well conclude the **best case**, **average case**, and **worst case** scenario of an algorithm.

Asymptotic analysis is input bound i.e., if there's no input to the algorithm, it is concluded to work in a constant time. Other than the "input" all other factors are considered constant.

For example, let us consider the search problem (searching a given item) in a sorted array.

The solution to above search problem includes:

- **Linear Search** (order of growth is linear)
- **Binary Search** (order of growth is logarithmic).

To understand how Asymptotic Analysis solves the problems mentioned above in analyzing algorithms,

- let us say:
    - we run the Linear Search on a fast computer A and
    - Binary Search on a slow computer B and
    - pick the constant values for the two computers so that it tells us exactly how long it takes for the given machine to perform the search in seconds.
- Let's say the constant for A is 0.2 and the constant for B is 1000 which means that A is 5000 times more powerful than B.
- For small values of input array size n, the fast computer may take less time.
- **But, after a certain value of input array size, the Binary Search will definitely start taking less time compared to the Linear Search even though the Binary Search is being run on a slow machine**.

Input Size	Running time on A	Running time on B
10	2 sec	~ 1 h
100	20 sec	~ 1.8 h

Input Size	Running time on A	Running time on B
10^6	~ 55.5 h	~ 5.5 h
10^9	~ 6.3 years	~ 8.3 h

Asymptotic Notations

Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

- O Notation
- Ω Notation
- θ Notation

Big Oh Notation, O

The notation O(n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst-case time complexity or the longest amount of time an algorithm can possibly take to complete.

Omega Notation, Ω

The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.

Theta Notation, θ

The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time. It is represented as follows –

More Algorithms Examples: -

Algorithm to convert temperature from Celsius to Fahrenheit.

```
Algorithm: Convert(C, F)
[The variable C is used as Celsius value and F is used as Fahrenheit
value]
1. Read C
2. Set F = 1.8 * C + 32
3. Print F
4. Return
```

Algorithm to calculate the sum of N natural numbers.

```
Algorithm: Sum (N)
[The variable N is represented the number of natural numbers.]
1. Set S = 0 and I = 1
2. Read N
3. S = S + I
4. I = I + 1
5. If I<=N Goto Step 3
6. Print S
7. Return
```

Algorithm of Linear search.

```
Algorithm: Linear_Search (A, N, Key)
[An array A with N elements and an item Key are given. This
algorithm searches the location of the Key in the array A]
1. Set I = 1
2. Repeat Steps 3 while I <= N and A[I]≠ Key
3. I = I + 1
   [End of while]
4. IF I <= N then
        Print: Item found at location I
   Else
        Print: Item not found
   [End of IF]
5. Return
```

1. Write an algorithm for swapping two values.

```
Step 1: Input first number as A
Step 2: Input second number as B
Step 3: SET TEMP = A
Step 4: SET A = B
Step 5: SET B = TEMP
Step 6: PRINT A, B
Step 7: END
```

2. Write an algorithm to find the larger of two numbers.

```
Step 1: Input first number as A
Step 2: Input second number as B
Step 3: IF A>B
           PRINT A
        ELSE
           IF A<B
              PRINT B
        ELSE
              PRINT "The numbers are equal"
              [END OF IF]
        [END OF IF]
Step 4: END
```

3. Write an algorithm to find whether a number is even or odd.

```
Step 1: Input number as A
Step 2: IF A%2 =0
              PRINT "EVEN"
        ELSE
              PRINT "ODD"
        [END OF IF]
Step 3: END
```

4. Write an algorithm to print the grade obtained by a student using the following rules.

```
Step 1: Enter the Marks obtained as M
Step 2: IF M>75
              PRINT O
Step 3: IF M>=60 AND M<75
              PRINT A
Step 4: IF M>=50 AND M<60
              PRINT B
Step 5: IF M>=40 AND M<50
              PRINT C
        ELSE
              PRINT D
```

Marks	Grade
Above 75	O
60–75	A
50–59	B
40–49	C
Less then 40	D

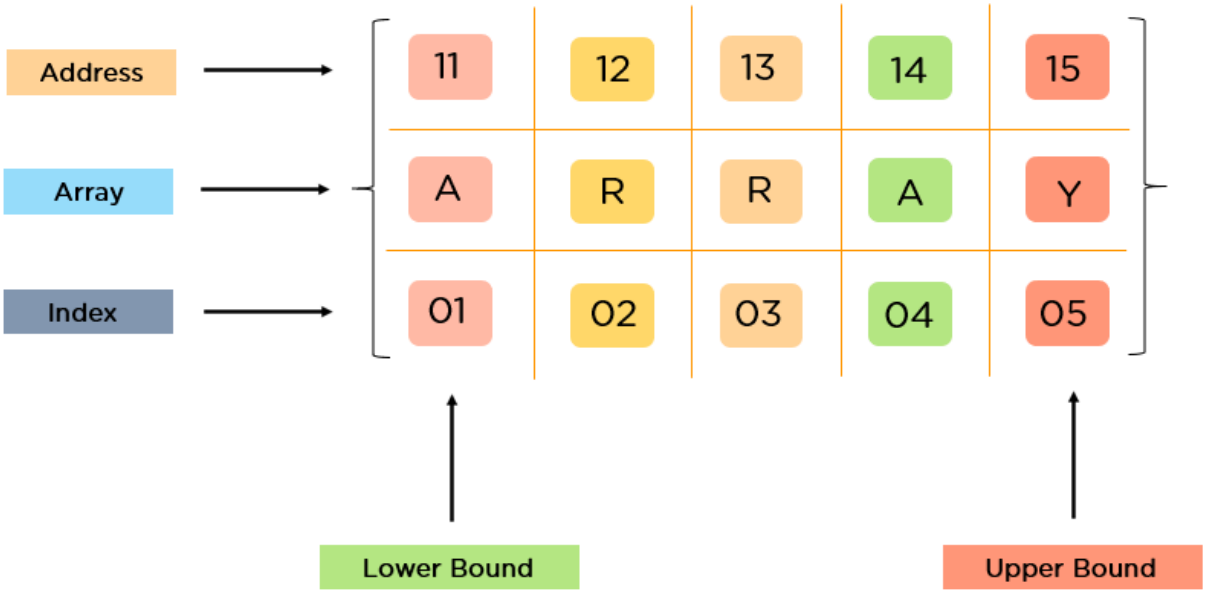
```
        [END OF IF]
Step 6: END
```

5. Write an algorithm to find the sum of first N natural numbers.

```
Step 1: Input N
Step 2: SET I = 1, SUM = 0
Step 3: Repeat Step 4 while I <= N
Step 4:         SET SUM = SUM + I
                SET I = I + 1
        [END OF LOOP]
Step 5: PRINT SUM
Step 6: END
```

Arrays in Data Structures

An array is a linear data structure that collects elements of the same data type and stores them in contiguous and adjacent memory locations. Arrays work on an index system starting from 0 to (n-1), where n is the size of the array.



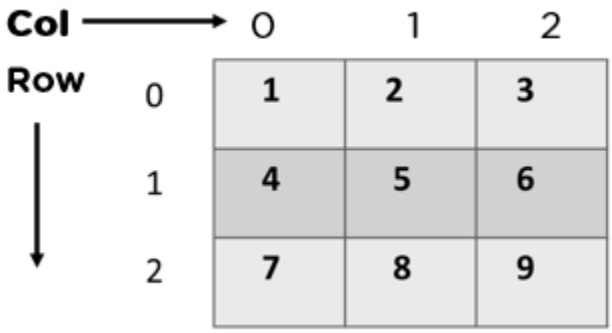
What Are the Types of Arrays?

There are majorly two types of arrays, they are:

One-Dimensional Arrays:

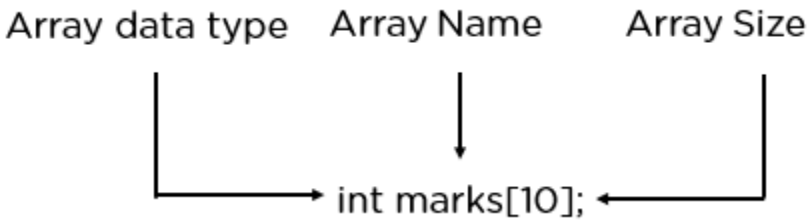
You can imagine a 1d array as a row, where elements are stored one after another.

Two-Dimensional Arrays:



You can imagine it like a table where each cell contains elements.

How Do You Declare an Array?



Arrays are typically defined with square brackets with the size of the arrays as its argument. Here is the syntax for arrays:
1D Arrays: int arr[n];
2D Arrays: int arr[m][n];

How Do You Initialize an Array?

You can initialize an array in four different ways:

- **Method 1:**

```
int a[6] = {2, 3, 5, 7, 11, 13};
```

- **Method 2:**

```
int arr[]= {2, 3, 5, 7, 11};
```

- **Method 3:**

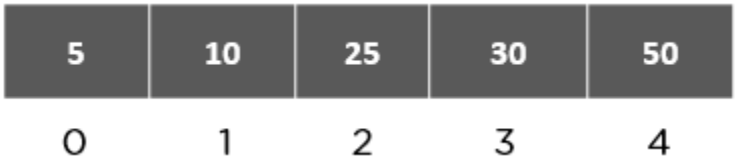
```
int n;
scanf("%d",&n);
int arr[n];
for(int i=0;i<5;i++)
{
scanf("%d",&arr[i]);
}
```

- **Method 4:**

```
int arr[5];
arr[0]=1;
arr[1]=2;
arr[2]=3;
```

arr[3]=4;
arr[4]=5;

How Can You Access Elements of Arrays in Data Structures?



You can access elements with the help of the index at which you stored them. Let's discuss it with a code:

```
#include<stdio.h>
int main()
{
int a[5] = {2, 3, 5, 7, 11};
printf("%d\n",a[0]); // we are accessing
printf("%d\n",a[1]);
printf("%d\n",a[2]);
printf("%d\n",a[3]);
printf("%d",a[4]);
return 0;
}
```

```
2
3
5
7
11
--------------------------------
Process exited after 0.1476 seconds with return value 0
Press any key to continue . . .
```

What Operations Can You Perform on an Array?

- Traversal
- Insertion
- Deletion
- Searching
- Sorting

Traversing the Array:

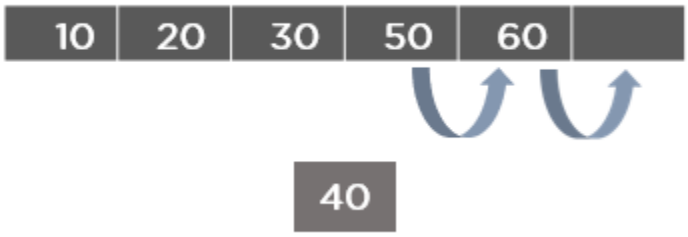
Traversal in an array is a process of visiting each element once.

Code:

```
#include<stdio.h>
int main()
{
int a[5] = {2, 3, 5, 7, 11};
for(int i=0;i<5;i++)
{
//traversing ith element in the array
printf("%d\n",a[i]);
}
return 0;
}
```

```
2
3
5
7
11
--------------------------------
Process exited after 0.1476 seconds with return value 0
Press any key to continue . . .
```

Insertion:



Insertion in an array is the process of including one or more elements in an array.
Insertion of an element can be done:

- **At the beginning**
- **At the end and**
- **At any given index of an array.**

At the Beginning:

Code:

```
#include<stdio.h>
int main()
{
   int array[10], n,i, item;
   printf("Enter the size of array: ");
    scanf("%d", &n);
   printf("\nEnter Elements in array: ");
      for(i=0;i<n;i++)
      {
       scanf("%d", &array[i]);
      }
   printf("\n enter the element at the beginning");
   scanf("%d", &item);
   n++;
   for(i=n; i>1; i--)
   {
    array[i-1]=array[i-2];
   }
   array[0]=item;
   printf("resultant array element");
   for(i=0;i<n;i++)
   {
    printf("\n%d", array[i]);
   }
   getch();
   return 0;
}
```

```
Enter the size of array: 5

Enter Elements in array: 2 3 5 7 11

enter the element at the beginning:1
resultant array element:
1
2
3
5
7
11
--------------------------------
Process exited after 19.94 seconds with return value 0
Press any key to continue . . .
```

At the End:

Code:

```
#include<stdio.h>
#include<conio.h>
int main()
{
int array[10], i, values;
printf("Enter 5 Array Elements: ");
for(i=0; i<5; i++)
    scanf("%d", &array[i]);
printf("\nEnter Element to Insert: ");
scanf("%d", &values);
array[i] = values;
printf("\nThe New Array is:\n");
for(i=0; i<6; i++)
    printf("%d ", array[i]);
getch();
return 0;
}
```

```
Enter 5 Array Elements: 2 3 5 7 11

Enter Element to Insert: 13

The New Array is:
2 3 5 7 11 13
--------------------------------
Process exited after 16.76 seconds with return value 0
Press any key to continue . . .
```

Lists

The arrangement of elements in some sequence is called list. There are two types of list.

i) Ordered List
ii) Unordered List

i) Ordered List

The proper sequence of elements is called ordered list. For example
The days of week, the month of year, seasons of year.

ii) Unordered List

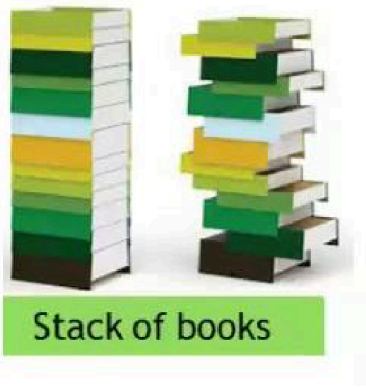
In this type of list the storage of element is not take place in proper order.
For example, Stack, Queue, Dequeue, Circular Queue, Link List

STACKS

A Stack is linear data structure. A stack is a list of elements in which an element may be inserted or deleted only at one end, called the top of the stack. Stack principle is **LIFO** (last in, first out). Which element inserted last on to the stack that element deleted first from the stack.

As the items can be added or removed only from the top i.e. the last item to be added to a stack is the first item to be removed.

Real life examples of stacks are:



Example

i) Pile of Trays in Cafeteria
ii) Stack of iron shorts (Box)
iii) Pile of bricks
iv) Pile of books in Library

Computer related Example

i) Expression, evolution in the stack process
ii) Calling functions/ Procedures and returned value stored in stack
iii) Conversion of decimal to binary is also a stack process.

Operations on stack:

The two basic operations associated with stacks are:

1. Push

2. Pop

While performing push and pop operations the following **test** must be conducted on the stack.

a) Stack is empty or not b) stack is full or not

1. Push: Push operation is used to add new elements into the stack. At the time of addition first check the stack is full or not. If the stack is **full** it generates an error message "**stack overflow**".

2. Pop: Pop operation is used to delete elements from the stack. At the time of deletion first check the stack is empty or not. If the stack is **empty** it generates an error message "**stack underflow**".

All insertions and deletions take place at the same end, so the last element added to the stack will be the first element removed from the stack. When a stack is created, the stack base remains fixed while the stack top changes as elements are added and removed. The most accessible element is the top and the least accessible element is the bottom of the stack.

Representation of Stack (or) Implementation of stack:

The stack should be represented in two ways:

1. Stack using array
2. Stack using linked list

1. Stack using array:

Let us consider a stack with 5 elements capacity. This is called as the size of the stack. The number of elements to be added should not exceed the maximum size of the stack. If we attempt to add new element beyond the maximum size, we will encounter a ***stack overflow*** condition. Similarly, you cannot remove elements beyond the base of the stack. If such is the case, we will reach a ***stack underflow*** condition.

1.push(): When an element is added to a stack, the operation is performed by push(). Below Figure shows the creation of a stack and addition of elements using push().

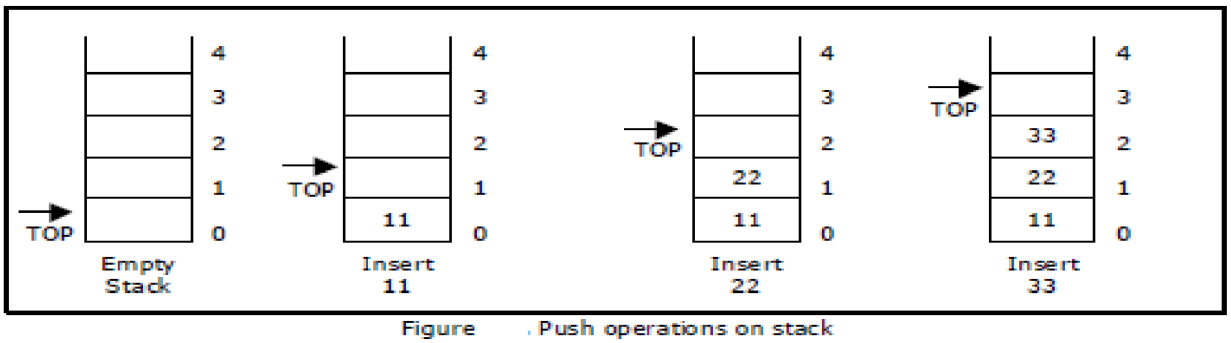


Figure . Push operations on stack

Initially **top=-1**, increment the top value i.e **top=top+1**, and insert element into the stack. We can insert an element into the stack first check the condition is stack **isfull** or not. i.e **top>=size-1**. Otherwise add the element in to the stack.

ALGORITHM FOR THE INSERTION OF ELEMENT IN THE STACK (PUSH)

Algorithm push (stack, N, item, Top)

[This algorithm is used to insert the element in the stack where Stack = Array, size-1 = Total number of elements, x = which is to be inserted, top = Top pointer/ Counter]

Step 1: START
Step 2: if top>=size-1 then
    Write " Stack is Overflow"
    Goto Step-4
Step 3: Otherwise
    3.1: read data value 'x'
    3.2: top=top+1;
    3.3: stack[top]=x;
    3.4: Goto Step-2
Step 4: END

CODE:

```
void push()
{
    int x;
    if(top >= n-1)
    {
    printf("\n\nStack Overflow..");
    return;
    }
    else
    {
    printf("\n\nEnter data: ");
    scanf("%d", &x);
    top = top + 1;
    stack[top] = x;
    printf("\n\nData Pushed into the stack");
    }
}
```

2.Pop(): When an element is taken off from the stack, the operation is performed by pop(). Below figure shows a stack initially with three elements and shows the deletion of elements using pop().

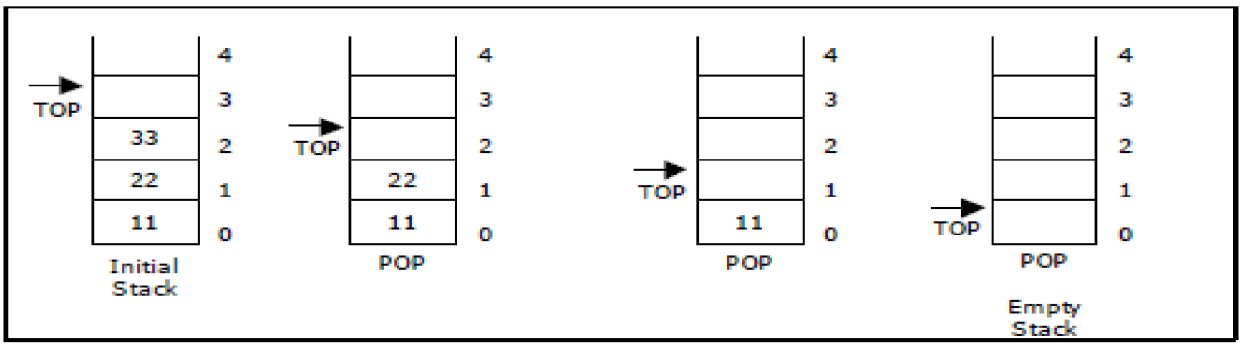


Figure Pop operations on stack

We can delete/remove an element from the stack, decrement the top value i.e **top=top-1**.
We can delete element from the stack first check the condition is stack **isempty** or not. i.e **top= -1**. Otherwise remove the element from the stack.

ALGORITHM FOR THE INSERTION OF ELEMENT IN THE STACK (POP)

Algorithm pop (stack, item, Top)
[This algorithm is used to delete an element from the stack where Stack = Array, top = Top pointer/ Counter.]

Step 1: START
Step 2: if top==-1 then
Write "Stack is Underflow"
Goto Step-4
Step 3: otherwise
3.1: print "deleted element"
3.2: top=top-1;
3.3: Goto Step-2
Step 4: END

CODE:

```
Void pop()
{
        If(top==-1)
        {
                Printf("Stack is Underflow");
        }
        else
        {
                printf("Delete data %d",stack[top]);
                top=top-1;
        }
}
```

3.display(): This operation performed display the elements in the stack. We display the element in the stack check the condition is stack is empty or not i.e top==-1.Otherwise display the list of elements in the stack.

Algorithms Display:

Step 1: START
Step 2: if top==-1 then
Write “Stack is Underflow”
Goto Step-4
Step 3: otherwise
3.1: print “Display elements are”
3.2: for top to 0
Print ‘stack[i]’
Step 4: END

CODE:

```
void display()
{
        If(top==-1)
        {
        Printf(“Stack is Underflow”);
        }
        else
        {
        printf(“Display elements are:);
        for(i=top;i>=0;i--)
        printf(“%d”,stack[i]);
        }
}
```

Source code for stack operations, using array:

```
#include<stdio.h>
#inlcude<conio.h>
int stack[100],choice,n,top,x,i;
void push(void);
void pop(void);
void display(void);
int main()
{
//clrscr();
top=-1;
printf("\n Enter the size of STACK[MAX=100]:");
scanf("%d",&n);
```

```
printf("\n\t STACK OPERATIONS USING ARRAY");
printf("\n\t--------------------------------");
printf("\n\t 1.PUSH\n\t 2.POP\n\t 3.DISPLAY\n\t 4.EXIT");
do
{
printf("\n Enter the Choice:");
scanf("%d",&choice);
switch(choice)
{
case 1:
{
push();
break;
}
case 2:
{
pop();
break;
}
case 3:
{
display();
break;
}
case 4:
{
printf("\n\t EXIT POINT ");
break;
}
default:
{
printf ("\n\t Please Enter a Valid Choice(1/2/3/4)");
}
}
}
while(choice!=4);
return 0;
}
```

```
void push()
{
if(top>=n-1)
{
printf("\n\tSTACK is over flow");
```

```
}
else
{
printf(" Enter a value to be pushed:");
scanf("%d",&x);
top++;
stack[top]=x;
}
}
```

```
void pop()
{
if(top<=-1)
{
printf("\n\t Stack is under flow");
}
else
{
printf("\n\t The popped elements is %d",stack[top]);
top--;
}
}
```

```
void display()
{
if(top>=0)
{
printf("\n The elements in STACK \n");
for(i=top; i>=0; i--)
printf("\n%d",stack[i]);
printf("\n Press Next Choice");
}
else
{
printf("\n The STACK is empty");
}
}
```

Applications of stack:

1. Stack is used by compilers to check for balancing of parentheses, brackets and braces.
2. Stack is used to evaluate a postfix expression.
3. Stack is used to convert an infix expression into postfix/prefix form.
4. In recursion, all intermediate arguments and return values are stored on the processor's stack.

5. During a function call the return address and arguments are pushed onto a stack and on return they are popped off.

Converting and evaluating Algebraic expressions:

An **algebraic expression** is a legal combination of operators and operands. Operand is the quantity on which a mathematical operation is performed. Operand may be a variable like x, y, z or a constant like 5, 4, 6 etc. Operator is a symbol which signifies a mathematical or logical operation between the operands. Examples of familiar operators include +, -, *, /, ^ etc.

An algebraic expression can be represented using three different notations. They are infix, postfix and prefix notations:
Infix: It is the form of an arithmetic expression in which we fix (place) the arithmetic operator in between the two operands.
Example: A + B
Prefix: It is the form of an arithmetic notation in which we fix (place) the arithmetic operator before (pre) its two operands. The prefix notation is called as polish notation.
Example: + A B
Postfix: It is the form of an arithmetic expression in which we fix (place) the arithmetic operator after (post) its two operands. The postfix notation is called as *suffix notation* and is also referred to *reverse polish notation*.
Example: A B +
Conversion from infix to postfix:
Procedure to convert from infix expression to postfix expression is as follows:
1. Scan the infix expression from left to right.
2. **a)** If the scanned symbol is left parenthesis, push it onto the stack.
b) If the scanned symbol is an operand, then place directly in the postfix expression (output).
c) If the symbol scanned is a right parenthesis, then go on popping all the items from the stack and place them in the postfix expression till we get the matching left parenthesis.
d) If the scanned symbol is an operator, then go on removing all the operators from the stack and place them in the postfix expression, if and only if the precedence of the operator which is on the top of the stack is greater than (*or greater than or equal*) to the precedence of the scanned operator and push the scanned operator onto the stack otherwise, push the scanned operator onto the stack.
The three important features of postfix expression are:
1. The operands maintain the same order as in the equivalent infix expression.
2. The parentheses are not needed to designate the expression unambiguously.
3. While evaluating the postfix expression the priority of the operators is no longer relevant.

We consider five binary operations: +, -, *, / and $ or ↑ (exponentiation). For these binary operations, the following in the order of precedence (highest to lowest):

OPERATOR	PRECEDENCE	VALUE
Exponentiation ($ or ↑ or ^)	Highest	3
*, /	Next highest	2
+, -	Lowest	1

Example 1:

Convert ((A – (B + C)) * D) ↑ (E + F) infix expression to postfix form:

SYMBOL	POSTFIX STRING	STACK	REMARKS
(		(	
(		((	
A	A	((	
-	A	((-	
(	A	((- (	
B	A B	((- (	
+	A B	((- (+	
C	A B C	((- (+	
)	A B C +	((-	
)	A B C + -	(	
*	A B C + -	(*	
D	A B C + - D	(*	
)	A B C + - D *		
↑	A B C + - D *	↑	
(	A B C + - D *	↑ (	
E	A B C + - D * E	↑ (	
+	A B C + - D * E	↑ (+	
F	A B C + - D * E F	↑ (+	
)	A B C + - D * E F +	↑	
End of string	A B C + - D * E F + ↑	The input is now empty. Pop the output symbols from the stack until it is empty.	

Example 2:

Convert the following infix expression A + B * C – D / E * H into its equivalent postfix expression.

SYMBOL	POSTFIX STRING	STACK	REMARKS
A	A		
+	A	+	
B	A B	+	
*	A B	+ *	
C	A B C	+ *	
-	A B C * +	-	
D	A B C * + D	-	
/	A B C * + D	- /	
E	A B C * + D E	- /	
*	A B C * + D E /	- *	
H	A B C * + D E / H	- *	
End of string	A B C * + D E / H * -	The input is now empty. Pop the output symbols from the stack until it is empty.	

Evaluation of postfix expression:

The postfix expression is evaluated easily by the use of a stack.

1. When a number is seen, it is pushed onto the stack;

2. When an operator is seen, the operator is applied to the two numbers that are popped from the stack and the result is pushed onto the stack.

3. When an expression is given in postfix notation, there is no need to know any precedence rules; this is our obvious advantage.

Example 1:

Evaluate the postfix expression: 6 5 2 3 + 8 * + 3 + *

SYMBOL	OPERAND 1	OPERAND 2	VALUE	STACK	REMARKS
6				6	
5				6, 5	
2				6, 5, 2	
3				6, 5, 2, 3	The first four symbols are placed on the stack.
+	2	3	5	6, 5, 5	Next a '+' is read, so 3 and 2 are popped from the stack and their sum 5, is pushed

8	2	3	5	6, 5, 5, 8	Next 8 is pushed
*	5	8	40	6, 5, 40	Now a '*' is seen, so 8 and 5 are popped as 8 * 5 = 40 is pushed
+	5	40	45	6, 45	Next, a '+' is seen, so 40 and 5 are popped and 40 + 5 = 45 is pushed
3	5	40	45	6, 45, 3	Now, 3 is pushed
+	45	3	48	6, 48	Next, '+' pops 3 and 45 and pushes 45 + 3 = 48 is pushed
*	6	48	288	288	Finally, a '*' is seen and 48 and 6 are popped, the result 6 * 48 = 288 is pushed

Example 2:

Evaluate the following postfix expression: 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

SYMBOL	OPERAND 1	OPERAND 2	VALUE	STACK
6				6
2				6, 2
3				6, 2, 3
+	2	3	5	6, 5
-	6	5	1	1
3	6	5	1	1, 3
8	6	5	1	1, 3, 8
2	6	5	1	1, 3, 8, 2
/	8	2	4	1, 3, 4
+	3	4	7	1, 7
*	1	7	7	7
2	1	7	7	7, 2
↑	7	2	49	49
3	7	2	49	49, 3
+	49	3	52	52

QUEUE

A queue is linear data structure and collection of elements. A queue is another special kind of list, where items are inserted at one end called the **rear** and deleted at the other end called the **front**. The principle of queue is a **"FIFO" or "First-in-first-out"**.

Queue is an abstract data structure. A queue is a useful data structure in programming. **It is similar to the ticket queue outside a cinema hall**, where the first person entering the queue is the first person who gets the ticket.

A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first.

Daily Life Example

i) The number of cars at police check

ii) People purchase tickets from cinema make a queue.

iii) People deposits their bills in bank make a queue.

Computer Based Example

i) When number of instructions are given for printing to computer, then those instructions will be executed first that was first assign to computer.

Operations on QUEUE:

A queue is an object or more specifically an abstract data structure (ADT) that allows the following operations:

Different operations can be performed at queue as

1) Create Queue

By using this operation, the structure of the queue is generated.

2) Enqueue

The insertion of element in the queue is called enqueue.

3) Dequeue

The deletion of element from the queue is called dequeue.

4) Empty Queue

It returns Boolean value true if queue is empty.

5) Full Queue

It returns Boolean value false if queue is full.

6) Destroy Queue

This operation deletes all the elements of the queue and deletes the structure of queue from the computer memory.

Major Test Operation on Queue before Insertion and Deletion:

1) Overflow Condition

When no more elements can be inserted in the queue then this condition is called overflow condition.

2) Underflow Condition

When queue is empty, and no more elements can be deleted from the queue then this condition is called underflow condition

Queue operations work as follows:

1. Two pointers called FRONT and REAR are used to keep track of the first and last elements in the queue.
2. When initializing the queue, we set the value of FRONT and REAR to 0.
3. On enqueing an element, we increase the value of REAR index and place the new element in the position pointed to by REAR.
4. On dequeueing an element, we return the value pointed to by FRONT and increase the FRONT index.
5. Before enqueing, we check if queue is already full.
6. Before dequeuing, we check if queue is already empty.
7. When enqueing the first element, we set the value of FRONT to 1.
8. When dequeing the last element, we reset the values of FRONT and REAR to 0.

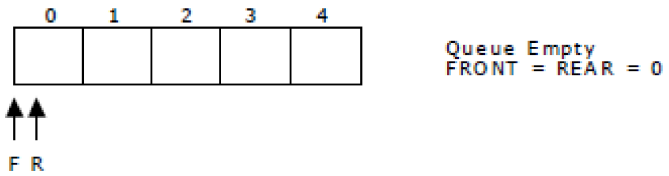
Representation of Queue (or) Implementation of Queue:

The queue can be represented in two ways:

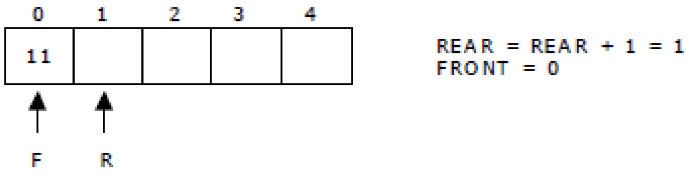
1. Queue using Array
2. Queue using Linked List

1.Queue using Array:

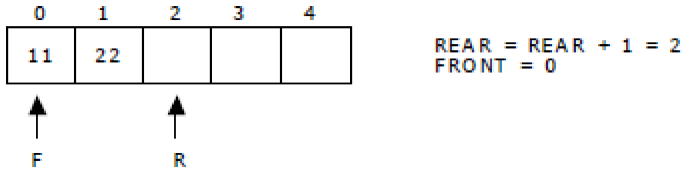
Let us consider a queue, which can hold maximum of five elements. Initially the queue is empty.



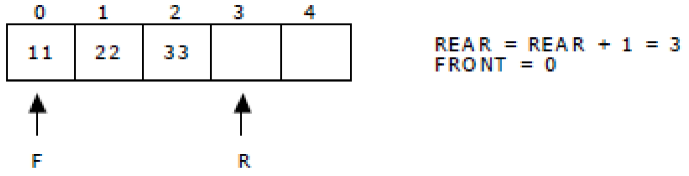
Now, insert 11 to the queue. Then queue status will be:



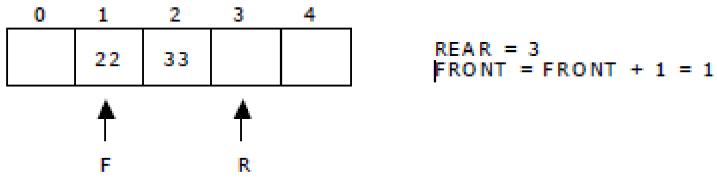
Next, insert 22 to the queue. Then the queue status is:



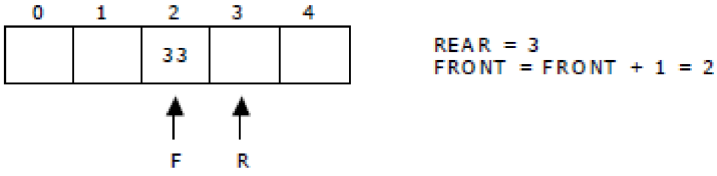
Again, insert another element 33 to the queue. The status of the queue is:



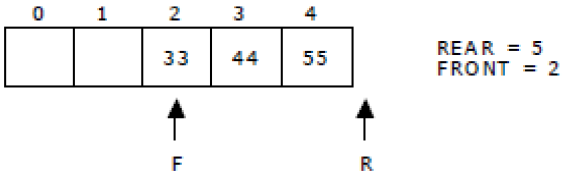
Now, delete an element. The element deleted is the element at the front of the queue. So the status of the queue is:



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:

ALGORITHM FOR THE INSERTION OF ELEMENT IN THE QUEUE

Algorithm enqueue (Q, N, item, Rear, Front)

[This algorithm is used to insert the element in the queue where, Q = Queue Array, N = Total number of elements, Item = which is to be inserted, Front, Rear = Deletion and Insertion pointers]

```
Step-1      [Check for overflow]
            If (Rear >= N) then
            Write ("Overflow")
            Goto step 5
            End if
Step-2      [Increment the Rear Pointer]
            Rear = Rear + 1
Step-3      [Insert the element at Rear position]
            Q [Rear] = item
Step-4      [Reset Pointer]
            If (Front = 0) then
            Set Front = 1
            End if
            Goto step-1
Step-5      [Finished]
             Exit
```

ALGORITHM FOR THE DELETION OF ELEMENT FROM THE QUEUE

Algorithm Dequeue (Q, item, Front, Rear)

[This algorithm is used for the deletion of element from the queue where Q = Array, Item = which is to be deleted]

```
Step-1      [Check for under flow]
             If ((Front = 0) or (Front > Rear)) then
            Write ("Under Flow")
             Goto step-4
            End if
Step-2      [Delete the item from queue]
            Item= Q [Front]
Step-3      [Set Front Pointer]
            If (Front = Rear) then
            Front = Rear=0
            Else
                    Front = Front + 1
            End if
                    Goto step-1
Step-4      [Finished]
              Exit
```

Applications of Queue:

1. It is used to schedule the jobs to be processed by the CPU.

2. When multiple users send print jobs to a printer, each printing job is kept in the printing queue. Then the printer prints those jobs according to first in first out (FIFO) basis.
3. Breadth first search uses a queue data structure to find an element from a graph.

Stacks

1.A stack is a linear list of elements in which the element may be inserted or deleted at one end.

2. In stacks, elements which are inserted last is the first element to be deleted.

3.Stacks are called LIFO (Last In First Out)list

4.In stack elements are removed in reverse order in which thy are inserted.

5.suppose the elements a,b,c,d,e are inserted in the stack, the deletion of elements will be e,d,c,b,a.

6.In stack there is only one pointer to insert and delete called “Top”.

7.Initially top=-1 indicates a stack is empty.

8.Stack is full represented by the condition TOP=MAX-1(if array index starts from ‘0’).

9.To push an element into a stack, Top is incremented by one

10.To POP an element from stack,top is decremented by one.

Queues

1.A Queue is a linerar list of elements in which the elements are added at one end and deletes the elements at another end.

2. . In Queue the element which is inserted first is the element deleted first.

3. Queues are called FIFO (First In First Out)list.

4. In Queue elements are removed in the same order in which thy are inserted.

5. Suppose the elements a,b,c,d,e are inserted in the Queue, the deletion of elements will be in the same order in which thy are inserted.

6. In Queue there are two pointers one for insertion called “Rear” and another for deletion called “Front”.

7. Initially Rear=Front=-1 indicates a Queue is empty.

8.Queue is full represented by the condition Rear=Max-1.

9.To insert an element into Queue, Rear is incremented by one.

10.To delete an element from Queue, Front is