

Discipline: BS (IT) 3rd Semester
Subject: Data Structure & Algorithms [NEW Course]
Notes: From Week No. 01 – 06
Prepared by: **ARSHAD IQBAL**, Lecturer (CS/IT), ICS/IT - FMCS,
The University of Agriculture, Peshawar

Course Objectives

Welcome to the course of data structure.

Data structure is very important subject as the topics covered in it, will be encountered by you again and again in the future courses. By completing this course, you will be able to understand the basics of Data Structures. You will also know the elementary Data Structures. At the end of this course you will be able to implements these data structure techniques using any programming language like C++, JAVA etc.

Week No. 01: DATA STRUCTURE

Introduction to Data: **Data** is a value or set of values **OR**
Data are facts that concern with people, person, place, event, objects etc.

Data Item: A single unit of value in data is called **data item** e.g. Name, Address, Roll No. etc.

Group Data Item: The data items divided into sub items are called **Group data items** e.g. Name is divided into First name, Middle name, Last name so name is a **group data item**.

Elementary Data Item: The data items that are not divided into sub items are called **Elementary data items** e.g. Roll Number which is not sub divided so Roll Number is an elementary item.

Information: Processed data that can be used in decision making is called **information** e.g. a list of student marks is data but when it is processed according to ascending order of marks, it become **information** that who has topped and who is failed.

Field: A group of related characters to represent some unit of information is called a **field** e.g. person name is a **field**. **Field** means a **column**. There are three types of **fields**:

- i. **Numeric Field:** Weight is 50 Kg.
- ii. **Alphabetic Field:** Name is Ali, Asad etc.
- iii. **Alpha Numeric Field:** Address is H. No. 77, Peshawar.

Key Field: A **key field** is used to identify the record for location & processing purposes. For **example**, in a sale ledger file the **key field** might be the customer code, in a payroll file the **key field** might be employee number, in student file the **key field** might be roll no. etc.

Record: Fields are grouped together to provide information about a single entity (object/unit/thing/person) is called **record**. **Record** means a **row** e.g. student record.

Roll No.	Name	DOB	Marks
35	Fahad	10 th Aug. 1986	485

File: A **file** is a named collection of records means records are grouped together to provide a complete information about all entities. **File** means a **table** e.g. the **file name** may be **student** and it will contain **four records** e.g.

Roll No.	Name	DOB	Marks
35	Fahad	10 th Aug. 1986	485
14	Asad	21 st March 1984	415
30	Iqbal	22 nd April 1982	420
40	Wasim	23 rd June 1985	430

Data Structure: **Structure** means particular way of data to store in computer memory. So, **data structure** means to organize the data in computer memory.

OR: the way in which data is efficiently stored, processed and retrieved is called **data structure**.

OR: Data Structure simply means a **structure** that can be used to store a given collection of data in computer memory. **OR Data Structure** is a named group of data of different data types which can be processed as a **single unit**.

The **representation** of **data structure** in **computer memory** (i.e. **RAM**) is known as **Storage Structure** and **representation** of **data structure** in **auxiliary memory** (i.e. **Hard disk**) is known as **File Storage**.

Prepared by: Arshad Iqbal, Lecturer (CS/IT), ICS/IT - FMCS, The University of Agriculture, Peshawar

Types of Data Structure:

1. According to Nature of Size:

- i. Static Data Structure.
- ii. Dynamic Data Structure.

i. Static Data Structure: A data structure is said to be static if data can store up to a fix number e.g. Int, Float, Char, Array etc.

ii. Dynamic Data Structure: As the name shows a dynamic data structure is that data structure which allows the programmer to change its size during program execution to add or delete memory space accordingly e.g. Link list, Tree, Graph etc.

2. According to its Occurrence (existence):

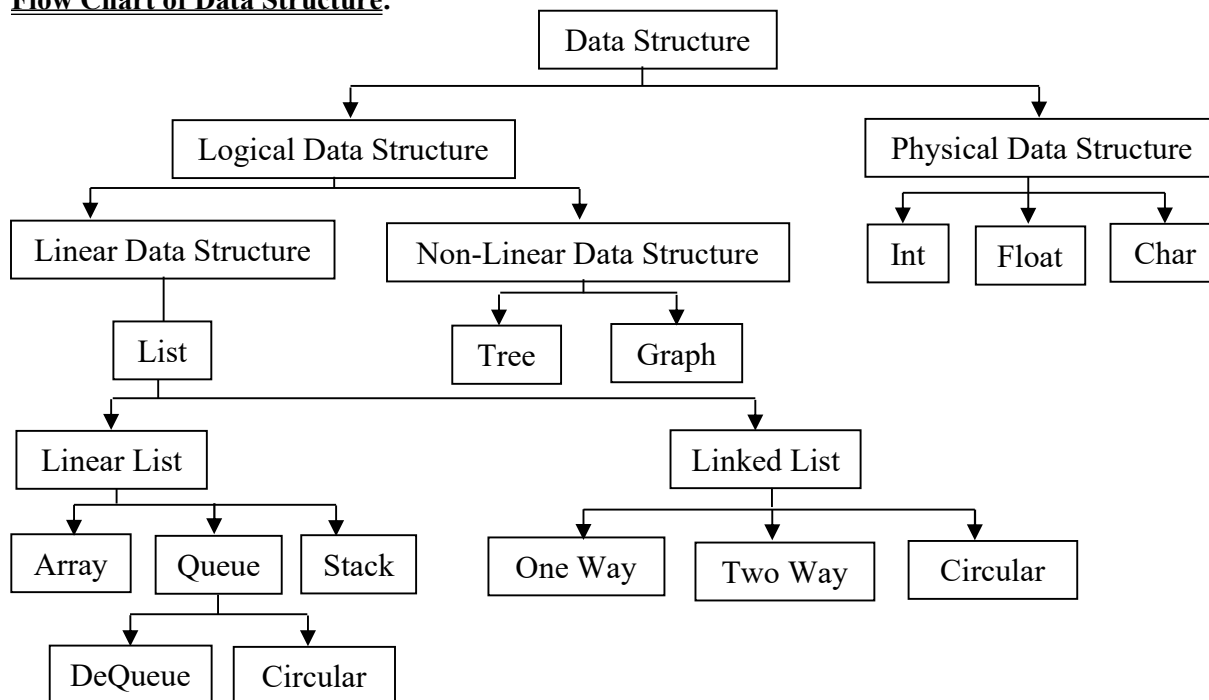
- i. Linear Data Structure.
- ii. Non - Linear Data Structure.

i. Linear Data Structure: In linear data structure, the data is stored in consecutive memory location or data is stored in a sequential form e.g. Array, Link list, Queue, Stack etc.

ii. Non - Linear Data Structure: In non - linear data structure, the data is stored in non - consecutive memory location or data is stored in a non - sequential form e.g. Tree, Graphs etc.

Physical and Logic Data Structure: The physical data structure refers to the physical arrangement of the data on the secondary storage device, usually disk. Typically, physical data structure concerns with specialists who design DBMSs. Analysts, programmers, and users are generally less concerned with the physical structure than the logical structure. The logical data structure concerns how the data "seem" to be arranged and the meanings of the data elements in relation to one another. For example, a data file is a collection of information stored together. This is its logical structure. However, physically a file could be stored on a disk in several scattered pieces.

Flow Chart of Data Structure:



Week No. 02: ALGORITHM

Introduction to Algorithm: An **algorithm** is a finite step-by-step list/procedure of well-defined instructions for solving a particular problem. The representation of data structure in computer memory (i.e. **RAM**) is known as **Storage Structure** and representation of data structure in auxiliary memory (i.e. **Hard disk**) is known as **File Storage**.

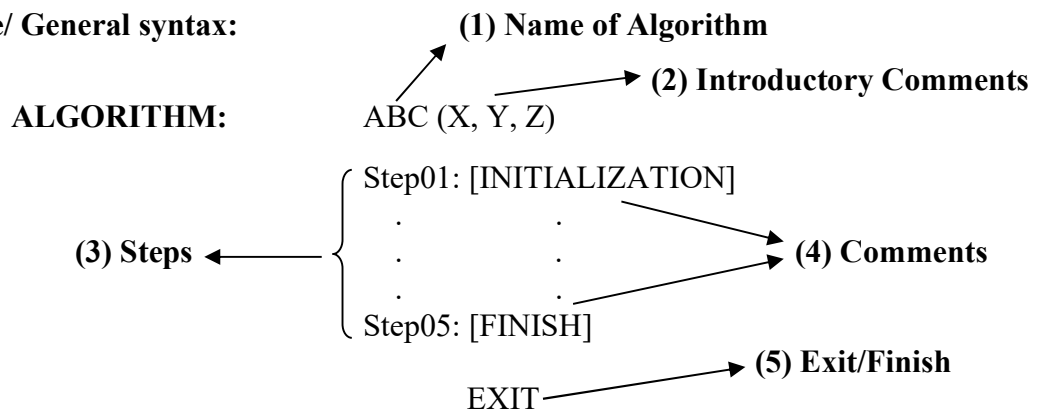
The term **algorithm** refers to the **storage structure**. An **algorithm** must satisfy the following criteria:

- i. **Input:** There are zero or more values, which are externally supplied.
- ii. **Output:** At least one quantity is produced as an output.
- iii. **Definite:** Each step or instruction must be clear from ambiguity.
- iv. **Effectiveness:** Each step or instruction must be sufficiently basic (simple) and easy so that algorithm become effective.
- v. **Finiteness:** It means that algorithm must be terminated in the finite number of steps.

Algorithmic Notations: The **general body of algorithm** can be drowning as:

1. **Name of Algorithm:** Every **algorithm** must have a **name**. The name of the algorithm will be written in capital letters.
2. **Introductory Comments:** The **algorithm name** is followed by a brief description of the tasks that algorithm performs and any assumption that has been made in it. The description gives the names and types of the **variables** used in **algorithm**.
3. **Steps:** The **algorithms** have many **steps**. Each step is begins with a phrase enclosed in square brackets ([]) which includes task to be performed or the action to be taken.
4. **Comments:** **Comments** are added to help the reader for understanding that step better. **Comments** specify no action and are enclosed only for explanation.
5. **Exit:** At the last of each **algorithm** the word **FINISH**, **EXIT** or **RETURN** is written which denotes the end of the **algorithm**.

Example/ General syntax:



Control Structure of Algorithms:

Algorithms are more easily understood if they mainly use self - contained modules and three types of **logic** or **flow of control**, which are given below:

1. Sequence Logic or Sequential Flow
2. Selection Logic or Conditional Flow
3. Iteration Logic or Repetitive Flow

1. Sequence Logic or Sequential Flow:

Sequence Logic means **Sequential Flow**. In **sequential flow** modules are executed in **sequence**. The **sequence** may be presented by means of numbered steps or by the order in which the modules are written. Most processing even of complex problem will generally follow this element flow pattern. See the following figure 2.1.

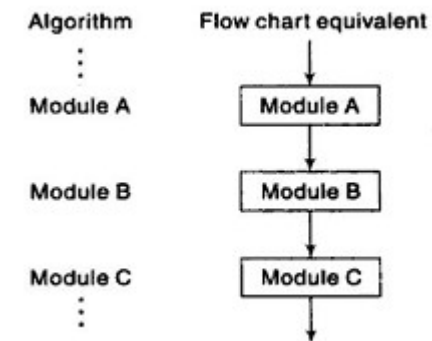


Figure 2.1: Sequence Logic

2. Selection Logic or Conditional Flow:

Selection Logic uses a number of conditions which lead to a selection of one out of several alternative modules. The structures which implement this logic are called **conditional structures** or **IF structures**. For clarity, it will frequently indicate the end of such a structure by the statement: **[End of If structure]** or some equivalent.

Types: There are three types of **conditional structures/IF structure**.

I. Single Alternative: This structure has the form:

If condition, then:

[Module A]

[End of If structure]

The **logic** of this structure is shown in the following **figure 2.2**. If the condition holds/true then **Module A**, which may consist of one or more statements, is executed. Otherwise **Module A** is skipped and control transfers to the next step of the algorithm.

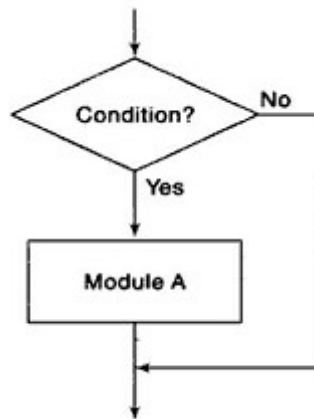


Figure 2.2: Single Alternative

II. Double Alternative: This structure has the form:

If condition, then:

[Module A]

Else:

[Module B]

[End of If structure]

The **logic** of this structure is shown in the following **figure 2.3**. As indicated by the flow chart if the condition holds/true then **Module A** is executed. Otherwise **Module B** is executed.

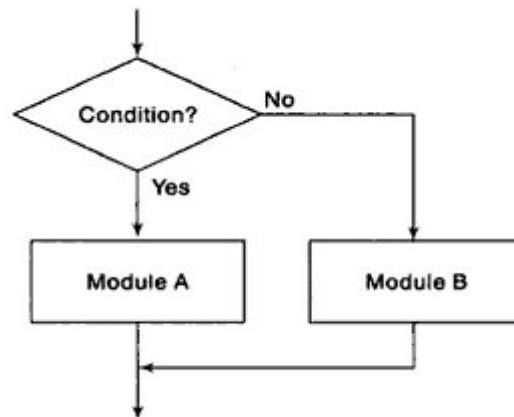


Figure 2.3: Double Alternative

III. Multiple Alternative: This structure has the form:

If condition (1), then:

[Module A₁]

Else if condition (2), then:

[Module A₂]

.

.

.

Else if condition (M), then:

[Module A_M]

Else:

[Module B]

[End of If structure]

The **logic** of this structure allows only one of the modules to be executed. Specifically, either the **module** which follows the first condition which holds/true is executed or the **module** which follows the final **Else** statement is executed.

3. Iteration Logic or Repetitive Flow:

Iteration Logic refers to either of two types of structures which involving **loops**, and that's why it is also called **Repetitive Flow**. Each type of structure begins with a **Repeat** statement and is followed by a **module**, called the **body of the loop**. For clarity it will indicate the end of the structure by the statement: **[End of loop]** or some equivalent.

Types: There are two types of **Iteration Logic/Repetitive Flow**:

I. Repeat - for loop:

The **repeat – for loop** uses an index variable, such as **K**, to control the **loop**. The **loop** will usually have the form:

Repeat for K = R to S by T:

[Module]

[End of loop]

The **logic** of this is shown in the following **figure 2.4**. Here **R** is called the **initial value**, **S** the **end value** or **test value**, and **T** the **increment**. Observe that the **body of the loop** is executed first with **K = R**, then with **K = R + T**, then with **K = R + 2T**, and so on.

The cycling ends when $K > S$. The following flow chart assumes that the increment T is positive, if T is negative, so that K decreases in value, then cycling ends when $K < S$.

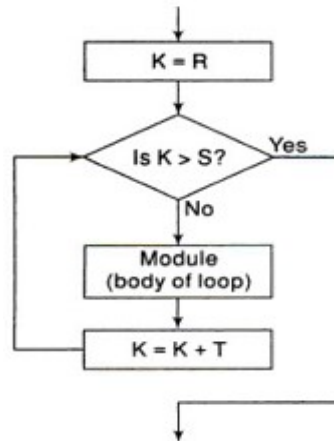


Figure 2.4: Repeat – for structure

II. Repeat - while loop:

The **repeat – while loop** uses a condition to control the **loop**. The **loop** will usually have the form:

Repeat while condition:

[Module]

[End of loop]

The **logic** of this structure is shown in the following **figure 2.5**. Observe that the cycling continues until the condition is **false**. We emphasize that there must be a statement before the structure that initializes the condition controlling the **loop**, and in order that the looping may eventually cease, there must be a statement in the **body of the loop** that changes the condition.

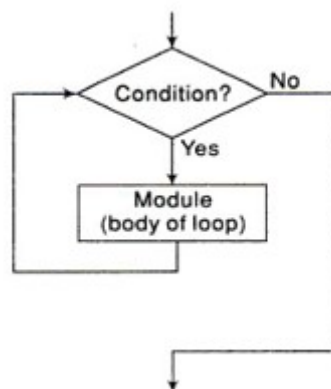


Figure 2.5: Repeat – while structure

Introduction to the Basic Operations of Data Structure:

The data appearing in data structures are processed by means of certain operations.

1. **Insertion:** The process of adding a new element in a data structure is called **insertion** OR inserting a new element or data item to a given data structure is called **insertion**.
2. **Deletion:** Removing an element or data item from a given data structure is called **deletion**. OR removing a record from the data structure is called **deletion**.
3. **Searching:** The process of finding out a data item in a given data structure is called **searching**. OR finding the location of record with a given key value or finding the locations of all records that satisfy one or more conditions.
4. **Traversing:** Accessing each record or data item in a data structure exactly once so that data items in the record can be processed is called **traversing** or **visiting**.
5. **Sorting:** Arranging the elements of a data structure in some particular order is called **sorting** or arranging the record in some logical order is called **sorting**.
6. **Merging:** Combing the records from two different files into a single sorted file is called **merging**.

Types:

- a. **Copying:** Combing some portions of two different files into a single file is called **copying**.
 - b. **Concatenation:** To combine two different files into one file is called **concatenation**.
7. **Creation:** The process by which the data structure is created is called **creation**.
 8. **Destruction:** The process through which we destroy a created data structure is called **destruction**.

Week No. 03: ONE-DIMENSIONAL ARRAY

Introduction to Array:

Array is the name of the consecutive memory locations that all have same name and same type.

OR

Array is a finite number of homogeneous data having a common name, a unique index, and stored in consecutive memory location in the computer memory.

By **finite number** mean that size of the array should be known. By **homogenous** mean that the data should be of the same type. By **consecutive** mean that the data item of the array stored one after the other in computer memory.

Types: There are two types of array:

1. One Dimensional Array
2. Two Dimensional Array

One Dimensional Array:

The **array** in which there is only one dimension is called **one dimensional array**. **One dimensional array** is also called a **list** or a **linear array** it consist of only one **row** or one **column** e.g. Int x [5];

Length or size of the One Dimensional Array:

Total number of elements in the **ODA** is called the **length of the One Dimensional Array**.

Formula:

$$\text{Length} = \text{UB} - \text{LB} + 1;$$

Where **UB** represents the **upper bound** of the one dimensional array and **LB** is **lower bound** of the one dimensional array.

Representation of One-Dimensional Array in computer memory:

In computer memory, **arrays** are usually mapped into a **vector**. A **one-dimensional array** does not cause any problem in mapping as it is already in the form of a **vector**. And their elements have stored in same sequence in which they are given. E.g. Int arr [4] = {10, 20, 30, 40}. The elements are stored in memory as follow:

0	1	2	3
10	20	30	40
1010	1012	1014	1016

Accessing One Dimensional Array by Dope Vector method:

The **dope vector method** is an efficient way to access each element of an **array**. This method uses the **start address** and **subscript number** (index number) of the element for accessing using the following **formula**:

$$\text{MA (i)} = \text{SA} + (\text{i}-1) * \text{w} \quad (\text{If array started from index value 1})$$

OR

$$\text{MA (i)} = \text{SA} + \text{i} * \text{w} \quad (\text{If array started from index value 0})$$

MA = Memory Address of the element

SA = Start Address

i = Subscript number (index number) of an element to be accessed

w = the word length, for integer **w** = 2, for float **w** = 4, for char **w** = 1

For example:

1	2	3	4
10	20	30	40
1010	1012	1014	1016

Find the Memory Address of an element at position i = 4?

$$\text{MA (i)} = \text{SA} + (\text{i}-1) * \text{w}$$

Where: SA = 1010 & w = 2

$$\text{MA (4)} = 1010 + (4-1) * 2$$

$$\text{MA (4)} = 1010 + 3 * 2$$

$$\text{MA (4)} = 1010 + 6$$

$$\text{MA (4)} = 1016$$

The value at address 1016 is 40.

Algorithm for traversing One Dimensional Array:

Traversing: **Traversing** of array means visiting each element of the array. The following algorithm is used to **traverse** an array.

ALGORITHM: **TRAVERSING (LB, UB, LA, K)**

Here **LA** is a linear array with lower bound (**LB**) and upper bound (**UB**) and **K** is a counter variable. The algorithm is used to traverse the **LA**.

Step # 01: [Initialize counter variable]

Set K: = LB

Step # 02: [Start loop to traverse]

Repeat step 3 & 4 while (K<=UB)

Step # 03: [Visit element]

Apply PROCESS to LA [K]

Step # 04: [Increase counter variable]

Set K: = K + 1

[End of Loop]

Step # 05: [Finish]

Exit

TRAVERSING ALGORITHM using C++:

```
#include<iostream.h>
#include<conio.h>
void main ()
{
    clrscr ();
    int LA [50], LB, UB, K;
    cout<<"Enter Upper Bound of the Array: UB = ";
    cin>>UB;
    cout<<endl<<"Enter Lower Bound of the Array: LB = ";
    cin>>LB;
    cout<<endl<<"Enter your Elements in the Array:"<<endl<<endl;
    for (K=0; K<=UB; K++)
        cin>>LA [K];

    // Start TRAVERSING ALGORITHM
    cout<<endl<<"Traversing Elements of the Array Starting from LB to UB: ";
    K = LB;
    while (K<=UB)
    {
        cout<<LA [K]<<" ";
        K = K+1;
    }
    // End of TRAVERSING ALGORITHM

    getch ();
}
```

Algorithm for insertion and deletion in One Dimensional Array:

Insertion: Insertion refers to the operation of adding another element to the collection of linear. If an element **inserts** at the **end** of the **array**, it is **easy**. But when an element **inserts** at **first/middle location** then the **elements** must be moved **downward**. The following algorithm is used to **insert** an **element**.

ALGORITHM: INSERTION (A, K, N, I, ITEM)

This algorithm is used to **insert** an element in array **A** at **Kth** location. **N** is the **total number of filled memory location** and **I** is the **counter variable**.

- Step#01:** [Initialize the counter variable]
I = N
- Step#02:** [Start loop to move the elements]
Repeat step 3 & 4 while (I > K) **or** (I >= K) (for 1 index number)
- Step#03:** [Moving element]
A [I] = A [I-1] **or** A [I + 1] = A [I] (when array started from 1 index number)
- Step#04:** [Decrement the counter variable]
I = I - 1
[End of Loop]
- Step#05:** [Add new element]
A [K] = ITEM
- Step#06:** [Add memory location]
N = N + 1
- Step#07:** [Finish]
Exit

INSERTION ALGORITHM USING C++:

```
#include<iostream.h>
#include<conio.h>
void main ()
{

    clrscr ();
    int A [100], I, K, N, ITEM;
    cout<<"Enter Total number of elements in array: N = ";
    cin>>N;
    cout<<endl<<"Enter N number of elements into ODA:"<<endl<<endl;
    for (I=0; I<N; I++)
        cin>>A[I];
    cout<<endl<<"Elements in ODA: ";
    for (I=0; I<N; I++)
        cout<<A[I]<<" ";
    cout<<endl<<endl<<"Enter the element to insert into ODA: ";
    cin>>ITEM;
    cout<<endl<<"Enter position of the element to be inserted: ";
    cin>>K;

    // Start INSERTION ALGORITHM

    I = N;
    while(I>K)
    {
        A [I] = A [I-1];
        I = I-1;
    }
    A[K] = ITEM;
    N = N+1;
    cout<<endl<<"Total number of elements in array after insertion of an element: "<<N<<endl;

    // End of Insertion Algorithm

    cout<<endl<<"Array after insertion of an element: ";

    for (I=0; I<N; I++)
        cout<<A[I] <<" ";

    getch ();
}
```

Deletion: **Deletion** refers to the operation of removing the element from a linear array **A**. **Deleting** the end element is not difficult but **deleting** the first/middle element would require that each subsequent element moved one location upward in order to fill up the **array**.

ALGORITHM: DELETION (A, K, N, I)

The algorithm is used to **delete** an element from **Kth** location of an array **A**. **N** is the **total number of filled memory locations** and **I** is the **counter variable**.

Step#01: [Initialize the counter variable]
I = K

Step#02: [Delete the element]
Delete A [K]

Step#03: [Starting loop]
Repeat step 4 & 5 while (I < N - 1) **or** (I <= N - 1) (for 1 index number)

Step#04: [Moving element]
A [I] = A [I + 1]

Step#05: [Increment the counter variable]
I = I + 1
[End of Loop]

Step#06: [Remove memory location]
N = N - 1
A [N] = -1 // Assign Garbage value

Step#07: [Finish]
Exit

DELETION ALGORITHM USING C++:

```
#include<iostream.h>
#include<conio.h>
void main ()
{
    clrscr ();
    int A [200], I, K, N;
    cout<<"Enter Total number of filled memory locations in array: N = ";
    cin>>N;
    cout<<endl<<"Enter N number of elements into Array:"<<endl<<endl;
    for (I=0; I<N; I++)
        cin>>A[I];
    cout<<endl<<"Elements in Array: ";
    for (I=0; I<N; I++)
        cout<<A[I]<<" ";
    cout<<endl<<endl<<"Enter position of an element to be deleted: k = ";
    cin>>K;
    // Start DELETION ALGORITHM
    I = K;
    cout<<endl<<"Element to be deleted at position "<<I<<": "<<A[I]<<endl<<endl;
    while (I<N-1)
    {
        A [I] = A [I+1];
        I = I+1;
    }
    N = N-1;
    A [N] = -1;
    // End of DELETION ALGORITHM
    cout<<"Total number of elements in array after deletion an element: "<<N<<endl<<endl;
    cout<<"Array after deletion of an element:"<<endl;
    for (I=0; I<=N; I++)
        cout<<A[I]<<" ";
    getch ();
}
```

Week No. 04: TWO-DIMENSIONAL ARRAY

Introduction to Two-Dimensional Array:

Two dimensional array has two dimensions which consist on both **rows** and **columns**. A **two dimensional array** is declared by giving two indexed values in square brackets. The **first** indexed value represents the **total number** of **rows** and the **second** represents the **total number** of **columns**. E.g. `Int arr [5][5];`

A **two dimensional array** is called **matrix** in **mathematics** and **table** in the **database**.

Two Dimensional Array Program-01:

```
#include<iostream.h>
#include <conio.h>
void main()
{
    clrscr ();
    int i, j;
    int A [3][3]={{10,20,30},{40,50,60},{70,80,90}};
    cout<<"Two Dimensional Array Elements: ";
    for (i=0; i<3; i++)
        for (j=0; j<3; j++)
            cout<<" "<<A[i][j];
    getch();
}
```

Two Dimensional Array Program-02:

```
#include<iostream.h>
#include <conio.h>
void main()
{
    clrscr ();
    int i, j;
    int A [3][3];
    cout<<"Enter elements into Two Dimensional Array: "<<endl<<endl;
    for (i=0;i<3;i++)
        for (j=0;j<3;j++)
            cin>>A[i][j];
    cout<<endl<<"Two Dimensional Array Elements: ";
    for (i=0; i<3; i++)
        for (j=0; j<3; j++)
            cout<<" "<<A[i][j];
    getch ();
}
```

Size or Length of Two Dimensional Array:

If **m** is the **number** of **rows** and **n** is the **number** of **columns** then the **formula** for the **Size of Two Dimensional Array** is as under:

$$\text{Length or Size} = m \times n$$

Representation of Two-Dimensional Array in computer memory:

When **two-dimensional array** is stored in **computer memory**, it is first mapped into a **single vector** because **two-dimensional array** is **logically** represented in **rows** and **columns** but **physically** mapped into **computer memory** in **vector** form. It is **either** mapped in a **Row-by-Row Vector** or a **Column-by-Column Vector**.

- Row-by Row Vector Method
- Column-by-Column Vector Method

- **Row-by Row Vector Method or Row-by-Row Major Order:**

In **row-by-row major order**, first row is stored/mapped, then second row and so on.

For example:

$$A = \begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

Can be mapped **row-by-row major order** as follows:

(1,1)	(1,2)	(1,3)	(2,1)	(2,2)	(2,3)	(3,1)	(3,2)	(3,3)
1	3	5	2	4	6	7	8	9

- **Column-by-Column Vector Method:**

In **column-by column** or **column major order**, first stored/mapped the **first column** then the **second column** and so on....

For example:

$$A = \begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

Can be mapped **column-by-column major order** as follows:

(1,1)	(2,1)	(3,1)	(1,2)	(2,2)	(3,2)	(1,3)	(2,3)	(3,3)
1	2	7	3	4	8	5	6	9

Accessing Two-Dimensional Array by Dope Vector method:

The **two-dimensional array** is stored into two types of **vectors** either **row-by-row** or **column-by-column**. A single **two-dimensional array** will have two different representations in computer memory according to the above-mentioned techniques.

Row Major Order (Row-by-Row):

Row Major Order (Row-by-Row) mapping an element can be accessed as:

$$MA(i, j) = SA + \{n(i - 1) + (j - 1)\} * w$$

MA = Memory Address of the element

i = No. of Row of elements to be accessed

j = No. of Column of elements to be accessed

SA = Start Address

n = Total number of Columns

w = the word length, for integer **w** = 2, for float **w** = 4, for char **w** = 1

For example:

$$A = \begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

Can be mapped **row-by-row major order** as follows:

(1,1)	(1,2)	(1,3)	(2,1)	(2,2)	(2,3)	(3,1)	(3,2)	(3,3)
1	3	5	2	4	6	7	8	9
1010	1012	1014	1016	1018	1020	1022	1024	1026

Find the Memory Address of an element at position (3, 2)?

$$i = 3, j = 2, SA = 1010, n = 3, w = 2$$

$$MA(i, j) = SA + \{n(i - 1) + (j - 1)\} * w$$

$$MA(3, 2) = 1010 + \{3(3 - 1) + (2 - 1)\} * 2$$

$$MA(3, 2) = 1010 + \{3(2) + (1)\} * 2$$

$$MA(3, 2) = 1010 + \{6 + 1\} * 2$$

$$MA(3, 2) = 1010 + \{7\} * 2$$

$$MA(3, 2) = 1010 + 14$$

$$MA(3, 2) = 1024$$

Thus, the value at the address 1024 is 8.

Column Major Order (Column-by-Column):

Column Major Order (Column-by-Column) mapping an element can be accessed as:

$$MA(i, j) = SA + \{m(j - 1) + (i - 1)\} * w$$

MA = Memory Address of the element

i = No. of Row of elements to be accessed

j = No. of Column of elements to be accessed

SA = Start Address

m = Total number of Rows

w = the word length, for integer **w** = 2, for float **w** = 4, for char **w** = 1

For example:

$$A = \begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

Can be mapped **Column-by-Columns major order** as follows:

(1,1)	(2,1)	(3,1)	(1,2)	(2,2)	(3,2)	(1,3)	(2,3)	(3,3)
1	2	7	3	4	8	5	6	9
1010	1012	1014	1016	1018	1020	1022	1024	1026

Find the Memory Address of an element at position (2, 3)?

$$i = 2, j = 3, SA = 1010, m = 3, w = 2$$

$$MA(i, j) = SA + \{m(j - 1) + (i - 1)\} * w$$

$$MA(2, 3) = 1010 + \{3(3 - 1) + (2 - 1)\} * 2$$

$$MA(2, 3) = 1010 + \{3(2) + (1)\} * 2$$

$$MA(2, 3) = 1010 + \{6 + 1\} * 2$$

$$MA(2, 3) = 1010 + \{7\} * 2$$

$$MA(2, 3) = 1010 + 14$$

$$MA(2, 3) = 1024$$

Thus, the value at the address 1024 is 6.

Week No. 05: SEARCHING

Introduction to Searching:

Computer systems are often used to store large amounts of data from which individual records are retrieved according to some **search** criteria.

The process of finding a specific data item or record from a list is called **searching**.

OR

It is the process by which can find the specific location of a given item in a list.

OR

Searching is a technique of finding an element from the given list or a set of elements like arrays, list or trees.

If the item exists in the given list, then **search** is said to be **successful** otherwise if the element is not found in the given list then **search** is said to be **unsuccessful**.

For example: Finding a telephone number in a telephone directory is called **searching**.

Internal search: The **search** in which the whole list resides in the **main memory** is called **internal search**.

External search: The **search** in which most of the list resides in the **secondary memory** is called **external search**.

In this **searching** we will concentrate on **internal searching**. The complexity of any **searching method** is determined from the number of comparisons performed among the collected elements in order to find the elements. The **time** required for operation is depends on the complexity of the operation or algorithm.

There are some **cases** in which an element can be found:

- **Best Case:** The **best case** is that in which the element is found during the first comparison.
- **Worst Case:** The **worst case** is that in which the element is found only at the end or the last comparison or not found.
- **Average Case:** The **average case** is that in which the element is found in comparisons more than **best case** but less than **worst case**.

Types of Searching:

1. Linear Search
2. Binary Search

1. Linear Search:

Linear Search is also called **sequential search**. It is the simplest way for finding an element in a list. In **Linear Search**, the elements find **sequentially** in a list, no matter whether the list is **sorted** or **unsorted**.

In case of **sorted** list, the **search** is started from 1st element and continuous until the element is found **or** the element whose value is greater than the value being **searched**.

In case of **unsorted** list, the **search** is started from 1st location and continuous until the element is found **or** the end of the list is reached.

Linear Search is a very slow process. It is used for small amounts of data. This method is not recommended for large amount of data.

Example: Sorted Array

Data

	0	1	2	3	4	5	6	7	8	9	10	11	12
A [13] =	10	20	30	40	50	60	70	80	90	100	110	120	130

Find the **ITEM = 80?**

1. A [0] = 10 comparison with 1st element, since 80 > 10 then
2. A [1] = 20 comparison with 2nd element, since 80 > 20 then
3. A [2] = 30 comparison with 3rd element, since 80 > 30 then
4. A [3] = 40 comparison with 4th element, since 80 > 40 then
5. A [4] = 50 comparison with 5th element, since 80 > 50 then
6. A [5] = 60 comparison with 6th element, since 80 > 60 then
7. A [6] = 70 comparison with 7th element, since 80 > 70 then
8. A [7] = 80 comparison with 8th element, since 80 = 80 thus

The **search** is **successful**, the **ITEM** is found at location 7.

Algorithm for Linear Search: This algorithm is used for **linear searching**. **ITEM** is to be searched in a linear array **A** having **N** numbers of elements. **LOC** is variable which store the location in case of **successful search** and '**I**' is a counter variable.

Algorithm for Sorted Array: LINEAR SEARCH (A, LOC, I, N, ITEM)

Step 1: [Initialization]

I = 0 and LOC = -1

Step 2: [Starting Loop to Search]

Repeat step 3 & 4 ... While (I < N)

Step 3: [Comparing given element]

If (ITEM == A [I]) then

LOC: = I

Break

(These statements will not be included for unsorted array) → { Else if (ITEM < A [I]) then
Break
[End of IF Structure]

Step 4: [Increment counter variable]

I = I+1

[End of Loop]

Step 5: [Display Result]

If (LOC == -1) then

Write ("Element is not Found")

Else:

Write ("Search is Successful")

Write ("ITEM is found at Location", LOC)

[End of IF Structure]

Step 6: [Finish]

Exit

LINEAR SEARCH ALGORITHM USING C++:

```
#include<iostream.h>
#include<conio.h>
void main ()
{
    clrscr();
    int A[50], I, LOC, N, ITEM;
    cout<<"Enter total number of elements in array: N = ";
    cin>>N;
    cout<<endl<<"Enter elements into array:"<<endl;
    for(I=0;I<N;I++)
        cin>>A[I];
    cout<<endl<<"Elements in array:";
    for(I=0;I<N;I++)
        cout<<" "<<A[I];
    cout<<endl<<endl<<"Enter an element to find: ";
    cin>>ITEM;
```

// STARTING LINEAR SEARCH ALGORITHM

```
LOC=-1;
I=0;
while(I<N)
{
    if (ITEM==A[I])
    {
        LOC=I;
        break;
    }
    else if(ITEM<A[I])
        break;
    I = I + 1;
}
cout<<endl<<"Display the Result:"<<endl<<endl;
if (LOC==1)
    cout<<"ITEM is not found and SEARCH is UNSUCCESSFULL";
else
    cout<<"SEARCH is SUCCESSFULL and ITEM is found at location: "<<I;

// End of LINEAR SEARCHING ALGORITHM
```

```
getch ();
}
```

2. Binary Search:

Binary search is the most **efficient searching technique** for **finding an element** in a **sorted list/array**. In **binary search**, first, find the **middle index** of the **list/array**, then compare the **element** (which you want to search) with the **middle element** of the **list/array**. If they are equal, the **search** is successful **otherwise** if the **element** is greater than the **middle element** of the **list/array**, then **right side** of the **list/array** will be **searched** and if the **element** is less than the **middle element**, then **left side** of the **list/array** will be **searched** in **similar way**.

Example:

	0	1	2	3	4	5	6	7	8	9	10	11	12
A [13] =	10	20	30	40	50	60	70	80	90	100	110	120	130

Find the **ITEM = 50?**

1. BEG=0 & END=12, **MID**=INT $((0+12)/2) = 6$ so A[MID]=**70** and $50 < 70$.
2. Since $50 < 70$ therefore BEG = 0 & END = MID - 1 = 6 - 1 = 5
MID = INT $((0 + 5) / 2) = 2.5 = 2$ so A [MID] = **30** and $50 > 30$.
3. Since $50 > 30$ therefore BEG = MID + 1 = 2 + 1 = 3 & END = 5
MID = INT $((3 + 5) / 2) = 4$ so A [MID] = **50** and $50 = 50$.

Thus, **search** is **successful**, the **ITEM** is **found** at location **4**.

Algorithm for Binary Search: This algorithm is used for **binary searching**. **ITEM** is to be searched in a linear sorted array **A** having **N** numbers of filled memory locations. **BEG**, **END** and **MID** variables are used to store array index numbers.

ALGORITHM: BINARY SEARCH (A, ITEM, LB, UB, BEG, END, MID)

Step 1: [Initialization]
BEG = LB & END = UB

Step 2: [Starting loop to search]
Repeat step 3, 4 & 5 While (BEG <= END)

Step 3: [Calculate MID]
MID = INT ((BEG + END) / 2)

Step 4: [Check element]
If (ITEM == A [MID]) Then
 Break
[End of IF Structure]

Step 5: [Set BEG & END]
If (ITEM < A [MID]) then
 END = MID - 1
Else:
 BEG = MID + 1
[End of IF Structure]
[End of Loop]

Step 6: [Display the Result]
If (ITEM == A [MID]) then
 Write ("Element found at location", MID)
Else:
 Write ("Element not found")
[End of IF Structure]

Step 7: [Finish]
Exit

BINARY SEARCH ALGORITHM USING C++:

```
#include<iostream.h>
#include<conio.h>
void main ()
{
    clrscr();
    int A[50], ITEM, MID, BEG, END, N, I;
    cout<<"Enter total number of elements in array: ";
    cin>>N;
    cout<<endl<<"Enter elements into Array:"<<endl;
    for(I=0;I<N;I++)
        cin>>A[I];
    cout<<endl<<"Elements in array:";
    for(I=0;I<N;I++)
        cout<<" "<<A[I];
    cout<<endl<<endl<<"Enter an element to find: ";
    cin>>ITEM;
```



```

// STARTING BINARY SEARCH ALGORITHM

BEG=0;
END=N-1;
while(BEG<=END)
{
    MID=int(BEG+END)/2;
    if(ITEM==A[MID])
        break;
    else if(ITEM<A[MID])
        END=MID-1;
    else
        BEG=MID+1;
}
cout<<endl<<"Display the Result:"<<endl;
if(ITEM==A[MID])
    cout<<endl<<"SEARCH is SUCCESSFUL and ITEM is found at location:"<<MID;
else
    cout<<endl<<"ITEM is not found and SEARCH is UNSUCCESSFUL";

// END OF BINARY SEARCH ALGORITHM

getch ();
}

```

Limitations of Binary Search:

In **binary search** two conditions have required **first** is: the list must be sorted and the **second** is: one must have the direct access to the **middle elements** in any sub list.

Week No. 06: RECURSION

Recursion: A function calls itself from its own body, is called **Recursion**.

- So that's why **infinite recursion/loop** can occur
- **Recursion** can also handle/stop using some conditions

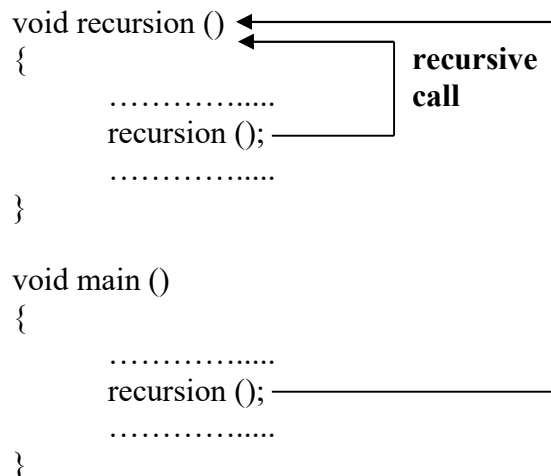
General syntax of recursion:

```
void recursion ()
{
    .....
    recursion ();
    .....
}
```

How recursion works in C++:

```
void recursion ()
{
    .....
    recursion ();
    .....
}

void main ()
{
    .....
    recursion ();
    .....
}
```



The diagram illustrates the flow of a recursive call. A box labeled 'recursive call' has an arrow pointing from the 'recursion ();' line in the 'void main ()' function to the 'void recursion ()' function. Another arrow points from the 'recursion ();' line in the 'void recursion ()' function back to the 'void recursion ()' function, indicating a self-call.

The **recursion** continues until some condition is met. To prevent **infinite recursion**, **if...else statement** (or similar approach) can be used.

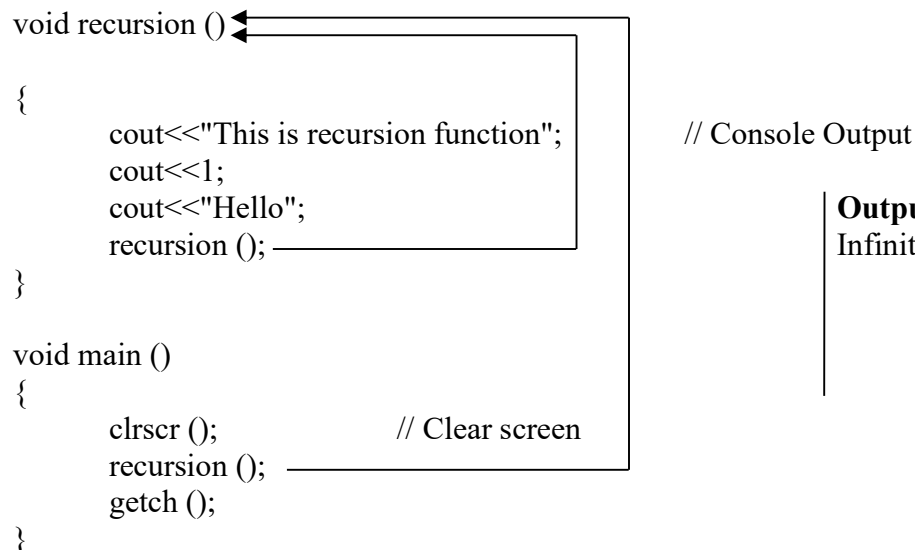
Examples:

Program1: Write a C++ program to use infinite recursion function.

```
#include <iostream.h> // header file: Input / Output stream
#include <conio.h> // header file: Console Input / Output and h means Header
```

```
void recursion ()
{
    cout<<"This is recursion function";
    cout<<1;
    cout<<"Hello";
    recursion ();
}

void main ()
{
    clrscr (); // Clear screen
    recursion ();
    getch ();
}
```



The diagram shows a box labeled 'recursive call' with an arrow pointing from the 'recursion ();' line in the 'void main ()' function to the 'void recursion ()' function. Another arrow points from the 'recursion ();' line in the 'void recursion ()' function back to the 'void recursion ()' function, indicating a self-call.

Output:
Infinite recursion

Now how can handle or stop recursion:

Program2: Write a C++ program to handle/stop infinite recursion function.

```
#include<iostream.h> // header file
#include<conio.h> // header file

void recursion (int X)
{
    cout<<X<<endl;
    if (X>1)
    {
        recursion (X-1);
    }
}

void main ()
{
    Clrscr ();
    recursion (10);
    getch ();
}
```

Output:

10
9
8
7
6
5
4
3
2
1

Program for Factorial using recursion function in C++:

In **mathematics**, the **factorial** of a positive integer **n**, denoted by **n!** is the product of all positive integers less than or equal to **n**. The **factorial** of a positive integer **n** is equal to $n \times (n-1) \times (n-2) \times \dots \times 3 \times 2 \times 1$. The **symbol** of factorial is **!**. For example: $4! = 4 \times 3 \times 2 \times 1 = 24$. **Note:** The value of **0!** is 1 and also the value of **1!** is 1.

In **C++**, **Factorial** is the result of multiplication of **n** number of consecutive integers from **1** to **n**. The **factorial** of a positive integer **n** is equal to $1 \times 2 \times 3 \times \dots \times n$.

For example: $4! = 1 \times 2 \times 3 \times 4 = 24$

Formula: $Z * \text{fac}(Z - 1)$

$$\begin{array}{l} \text{fac}(4) = 4 * \text{fac}(3) = 4 * 6 = 24 \uparrow \\ \text{fac}(3) = 3 * \text{fac}(2) = 3 * 2 = 6 \\ \text{fac}(2) = 2 * \text{fac}(1) = 2 * 1 = 2 \\ \downarrow \text{fac}(1) = 1 \end{array}$$

Program1: Write a C++ Program for Factorial using recursion function.

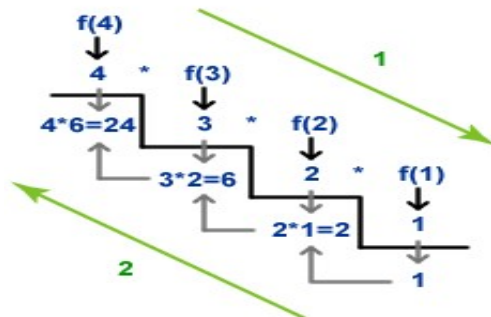
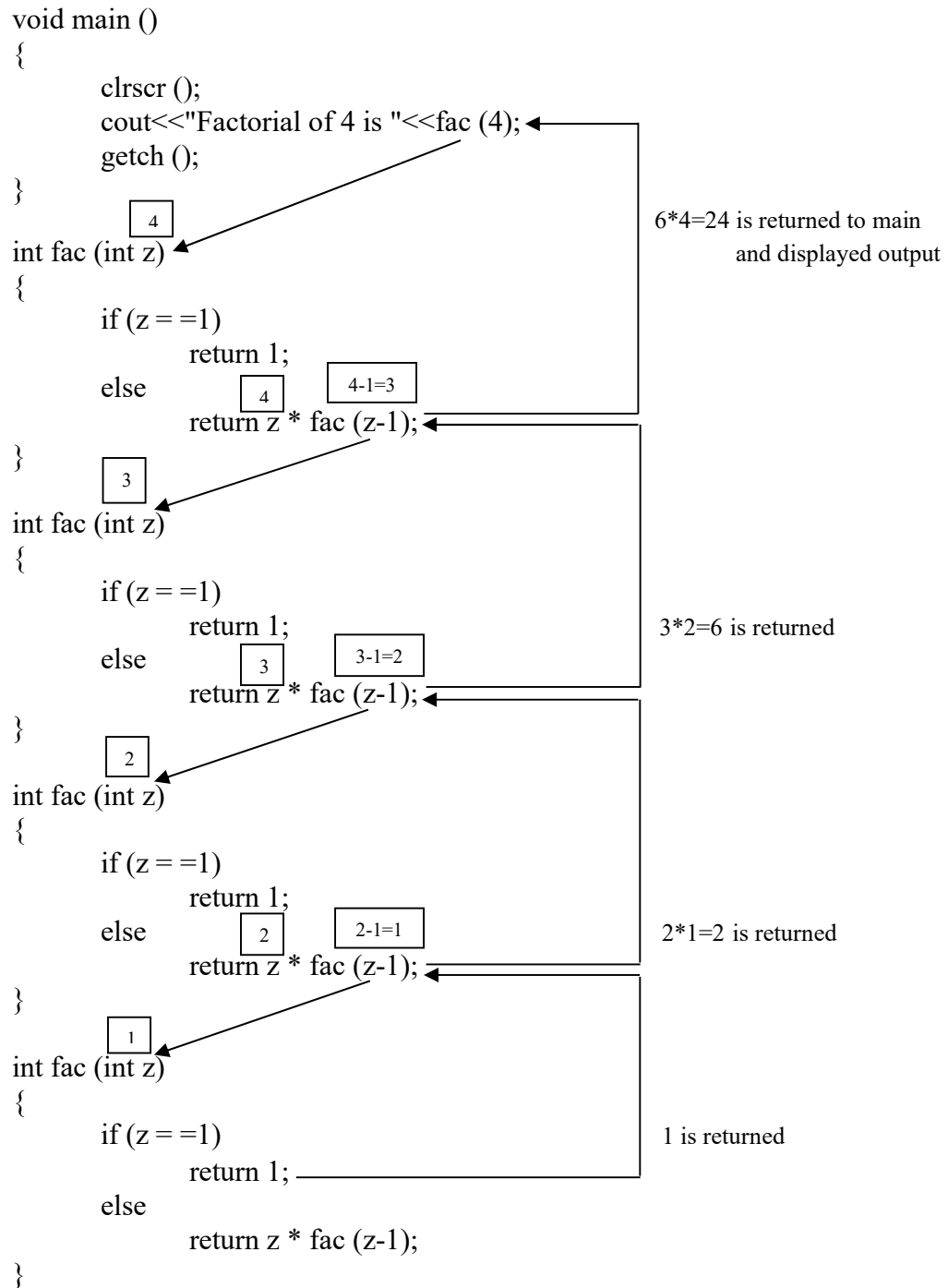
```
#include <iostream.h> // header file
#include <conio.h> // header file
```

```
int fac (int z)
{
    if (z == 1)
        return 1;
    else
        return z * fac (z-1);
}

void main ()
{
    clrscr ();
    cout << "Factorial of 4 is " << fac (4);
    getch ();
}
```

Output:
1*2*3*4=24

How the above example works:



User wants to enter a value during execution time:

Program2:

```
#include <iostream.h>    // header file
#include <conio.h>       // header file

int fac (int z)
{
    if (z == 1)
        return 1;
    else
        return z * fac (z-1);
}

void main ()
{
    clrscr ();
    int value;
    cout<<"Enter value for factorial "<<endl;
    cin>>value;
    cout<<"Factorial of "<<value<< " is "<<fac (value);
    getch ();
}
```

Output:

Depend on
Input value

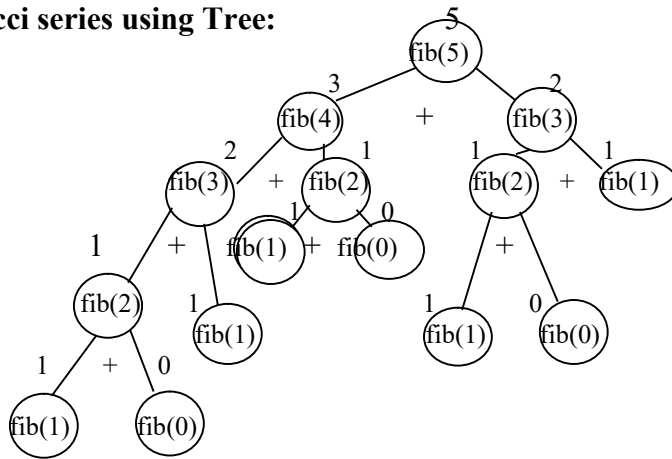
Program for Fibonacci sequence using recursion function in C++:

Series: 0 1 1 2 3 5 8 13 21 is a **Fibonacci series**. In **Fibonacci series**, each term is the sum of the two preceding (previous) terms.

For example: Fib (5) = 0 1 1 2 3 = 5

$$\begin{array}{l}
 \uparrow \\
 \text{fib (5) = fib (4) + fib (3) = 3 + 2 = 5} \\
 \text{fib (4) = fib (3) + fib (2) = 2 + 1 = 3} \\
 \text{fib (3) = fib (2) + fib (1) = 1 + 1 = 2} \\
 \text{fib (2) = fib (1) + fib (0) = 1 + 0 = 1} \\
 \text{fib (1) = 1} \\
 \downarrow \\
 \text{fib (0) = 0}
 \end{array}$$

Fibonacci series using Tree:



Program1: Write a C++ Program for Fibonacci Sequence using recursion function.

```

#include <iostream.h>           // header file
#include <conio.h>             // header file

int fib (int);                // Prototype: is used to write main() function first and then user defined function
void main ()
{
    clrscr ();
    int f;
    f = fib (5);
    cout<<"The result of 5 Fibonacci is "<<f;
    getch ();
}

int fib(int n)
{
    if( (n==0) || (n==1))
        return n;
    else
        return (fib (n-1) + fib (n-2));
}
    
```

Fibonacci Series:
Fib (5): 0 1 1 2 3 = 5

Output:
The result of 5 Fibonacci is 5

User enters a value during execution time:

Program2:

```
#include <iostream.h>      // header file
#include <conio.h>         // header file

int fib (int);    // Prototype: is used to write main () function first and then user defined function

void main ()
{
    clrscr ();
    int f, value;
    cout<<"Enter value for Fibonacci"<<endl;
    cin>>value;
    f = fib (value);
    cout<<"Fibonacci of "<<value<<" is "<<f;
    getch ();
}

int fib (int n)
{
    If ( (n==0) || (n==1))
        return n;
    else
        return (fib (n-1) + fib (n-2));
}
```

Types of Recursion:

1. Head Recursion:

A call is **head-recursive** when the first statement of the function is the recursive call.

Example:

```
void print (int n)
{
    print (n-1); // The first executed statement is recursive call
    if (n < 0) return;
    cout << " " << n;
}
```

2. Middle Recursion:

A call is **mid-recursive** when the recursive call occurs in the **middle** of the function means there are other statements before and after the recursive call.

Example:

```
void print (int n)
{
    if (n < 0) return;
    print (n-1); // The middle-executed statement is recursive call
    cout << " " << n;
}
```

3. Multi Recursion:

When two or more recursive calls occur in a function, then the function is **multi-recursive**.

Example:

```
void print (int n)
{
    print (n-1);
    if (n < 0) return;
    print (n-1);
    cout << " " << n;
}
```

4. Tail Recursion:

In **tail recursion**, calculations perform first, and then execute the recursive call, passing the results of current step to the next recursive step and the recursive call should be the last statement.

A recursive function is **tail recursive** when recursive call is the last thing executed by the function and the recursive call should be the last statement.

OR

Tail recursion is recursion where the function calls itself at the end ("**tail**") of the function. At some point, the function decides not to call itself again and the result is returned. Thus, this behavior is analogous (similar) to a **loop**.

OR

A function call is said to be **tail recursive** if there is nothing to do after the function returns except return its value and the recursive call should be the last statement.

For example, the following C++ function print () is tail recursive:

```
void print (int n)
{
    if (n < 0) return;
    cout << " " << n;
    print (n-1);    // The last executed statement is recursive call
}
```

Another example:

```
void tail (int i)
{
    if (i>0)
    {
        system.out.print (i+"")
        tail (i-1)    // The last executed statement is recursive call
    }
}
```

Is the factorial method a tail recursive method?

```
int fact (int x)
{
    if (x==0)
        return 1;
    else
        return x*fact(x-1);
}
```

- No, because when returning back from a recursive call, there is still one pending operation, multiplication.
- Therefore, **factorial** is a non-tail recursive method.

Is the following program tail recursive?

```
void prog (int i)
{
    if (i>0)
    {
        prog (i-1);
        System.out.print (i+"");
        prog (i-1);
    }
}
```

- No, because there is an earlier recursive call, other than the last one.
- In **tail recursion**, the recursive call should be the last statement, and there should be no earlier recursive calls whether direct or indirect.

5. Non-Tail Recursion:

Recursive methods that are not tail recursive are called **non-tail recursive**.

For example, the following C++ function fact () is **non-tail recursive**:

```
int fact (int x)
{
    if (x == 0)
        return 1;
    else
        return x*fact(x-1);
}
```

The above **factorial** is a **non-tail recursive** method, because when returning back from a recursive call, there is still one pending operation, multiplication.

6. Direct Recursion:

A function when it calls itself directly is known as **Direct Recursion**.

For example:

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

Here, inside *factorial (int n)*, it directly calls itself as *n*factorial (n-1)*. This is **direct recursion**.

7. Indirect Recursion:

A function is said to be **indirect recursive** if it calls another function and the new function calls the first calling function again.

For example:

```
int func1(int n)
{
    if (n<=1)
        return 1;
    else
        return func2(n);
}

int func2(int n)
{
    return func1(n-1);
}
```

Here, **recursion** takes place in **2** steps, unlike **direct recursion**.

- First, *func1* calls *func2*
- Then, *func2* calls back the first calling function *func1*.