Chameli Devi Group of Institutions, Indore
Department of ESH
BT205 Basic Computer Engineering
B. Tech, CSE & IT (II Semester)
Unit -3

……

Syllabus: Object & Classes, Scope Resolution Operator, Constructors & Destructors, Friend Functions, Inheritance, Polymorphism, Overloading Functions & Operators, Types of Inheritance, Virtual functions. Introduction to Data Structures.

Unit Objective: To familiarize the students with basic concepts of object-oriented programming

Unit Outcome: Student should be able to implement object oriented programming concepts and relate them with practical world.

……..

Class:

Class in C++ is the building block that leads to Object-Oriented programming. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A C++ class is like a blueprint for an object.

Example: Consider the Class of Cars. There may be many cars with different names and brands but all of them will share some common properties like all of them will have 4 wheels, Speed Limit, Mileage range, etc. So here, Car is the class, and wheels, speed limits, and mileage are their properties.

A Class is a user-defined data type that has data members and member functions. Data members are the data variables and member functions are the functions used to manipulate these variables together, these data members and member functions define the properties and behavior of the objects in a Class.

In the above example of class Car, the data member will be speed limit, mileage, etc, and member functions can be applying brakes, increasing speed, etc.

Object:

An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

In other words, object is an entity that has state and behavior. Here, state means data and behavior means functionality. Object is a runtime entity; it is created at runtime. All the members of the class can be accessed through object.

Programming Example for Class & Object:

```
class Geeks {
   // Access specifier
public:
   // Data  Members
   string geekname;
   // Member Functions()
   void printname()
```

```
{
cout << “Name is:" << geekname;
}
};
int main()
{
   // Declare an object of class geeks
   Geeks obj1;
   // accessing data member
   obj1.geekname = "Abhi";
   // accessing member function
   obj1.printname();
   return 0;
}
```

OUTPUT: Name is Abhi

Scope Resolution Operator:

To define a member function outside the class definition we have to use the scope resolution operator:: along with the class name and function name.

Programming Example for scope resolution operator:

```
class Geeks
{
   public:
   string geekname;
   int id;
     // printname is not defined inside class definition
   void printname();
     // printid is defined inside class definition
   void printid()
   {
      cout <<"Geek id is: "<<id;
   }
};
void Geeks::printname()
{
   cout <<"Geekname is: "<<geekname;
}
int main() {
   Geeks obj1;
   obj1.geekname = "xyz";
   obj1.id=15;
   obj1.printname();
   cout << endl;
   obj1.printid();
```

```
    return 0;
}
```
Geekname is: xyz
Geek id is: 15

Access Modifiers/Specifier:

Access modifiers are used to implement an important aspect of OOP known as Data Hiding. Access Modifiers or Access Specifiers in a class are used to assign the accessibility to the class members, i.e., they set some restrictions on the class members so that they can't be directly accessed by the outside functions. There are 3 types of access modifiers available in C++: Public, Private, Protected

Note: If user do not specify any access modifiers for the members inside the class, then by default the access modifier for the members will be Private.

Public: All the class members declared under the public specifier will be available to everyone. The data members and member functions declared as public can be accessed by other classes and functions too. The public members of a class can be accessed from anywhere in the program using the direct member access operator (.) with the object of that class.

Private: The class members declared as private can be accessed only by the member functions inside the class. They are not allowed to be accessed directly by any object or function outside the class. Only the member functions or the friend functions are allowed to access the private data members of the class. However, user can access the private data members of a class indirectly using the public member functions of the class.

Protected: The protected access modifier is similar to the private access modifier in the sense that it can't be accessed outside of its class unless with the help of a friend class. The difference is that the class members declared as Protected can be accessed by any subclass (derived class) of that class as well.

Note: This access through inheritance can alter the access modifier of the elements of base class in derived class depending on the mode of Inheritance.

Constructor:

Constructor in C++ is a special method that is invoked automatically at the time of object creation. Constructor is a special type of member function that is used to initialize the data members for an object of a class automatically, when an object of the same class is created. Constructor is invoked at the time of object creation. It constructs the values i.e. provides data for the object which is why it is known as constructors.

Constructor is a special member function of a class, whose name is same as the class name. Constructor do not return value, hence they do not have a return type. Constructors are mostly declared in the public section of the class though it can be declared in the private section of the class. Constructors can be overloaded. Constructor cannot be declared virtual.

The **prototype** of the constructor looks like: **<class-name> (list-of-parameters);**

Constructor can be defined inside the class declaration or outside the class declaration

- **Syntax for defining the constructor within the class**

```
<class-name>(list-of-parameters)
{
    //constructor definition
}
```

- **Syntax for defining the constructor outside the class**

```
<class-name>: :<class-name>(list-of-parameters)
{
    //constructor definition
}
```

Programming Example for Constructor:

```
class student
{
  int rno;
  char name[50];
  double fee;
  public:
  student()
  {
    cout<<"Enter the RollNo:";
    cin>>rno;
    cout<<"Enter the Name:";
    cin>>name;
    cout<<"Enter the Fee:";
    cin>>fee;          }
void display()
  {
   cout<<endl<<rno<<"\t"<<name<<"\t"<<fee;
  }
};
int main()
{
  student s;  //constructor gets called automatically when we create the object of the class
  s.display();
  return 0;
 }
```

Default Constructor:

A constructor without any arguments or with the default value for every argument is said to be the Default constructor. A constructor that has zero parameter list or in other sense, a constructor that accept no arguments is called a zero-argument constructor or default constructor.

If default constructor is not defined in the source code by the programmer, then the compiler defined the default constructor implicitly during compilation. If the default constructor is defined explicitly in the

program by the programmer, then the compiler will not define the constructor implicitly, but it calls the constructor implicitly.

Parameter Constructor:

It is possible to pass arguments to constructors. Typically, these arguments help initialize an object when it is created. To create a parameterized constructor, just add parameters to it the way you would to any other function. When you define the constructor's body, use the parameters to initialize the object.

Note: When the parameterized constructor is defined and no default constructor is defined explicitly, the compiler will not implicitly call the default constructor and hence creating a simple object as

Student s;

Will flash an error

Copy Constructor:

A copy constructor is a member function that initializes an object using another object of the same class. In simple terms, a constructor which creates an object by initializing it with an object of the same class, which has been created previously is known as a copy constructor.

Copy constructor is used to initialize the members of a newly created object by copying the members of an already existing object. Copy constructor takes a reference to an object of the same class as an argument.

```
Sample(Sample &t)
{
   id=t.id;
}
```

The process of initializing members of an object through a copy constructor is known as copy initialization.

Destructor:

Destructor is a member function which is invoked automatically whenever an object is going to be destroyed. Meaning, a destructor is the last function that is going to be called before an object is destroyed. Destructor is also a special member function like constructor. Destructor destroys the class objects created by constructor. Destructor has the same name as their class name preceded by a tilde (~) symbol. It is not possible to define more than one destructor. Destructor neither requires any argument nor returns any value. It is automatically called when object goes out of scope. Destructor release memory space occupied by the objects created by constructor. In destructor, objects are destroyed in the reverse of an object creation.

Friend Function:

A friend function can be granted special access to private and protected members of a class in C++. They are the non-member functions that can access and manipulate the private and protected members of the class for they are declared as friends.

A friend function can be:

- A global function
- A member function of another class

Declaration Syntax:

```
friend return_type function_name (arguments);    //for a global function
        or
```

friend return_type class_name :: function_name (arguments); //for a member function of another class

Programming Example for Friend Function:

```
class base {
private:
   int private_variable;
 protected:
   int protected_variable;
 public:
   base()
   {
      private_variable = 10;
      protected_variable = 99;
   }
     // friend function declaration
   friend void friendFunction(base& obj); };
void friendFunction(base& obj)
{
   cout << "Private Variable: " << obj.private_variable;
   cout << "Protected Variable: " << obj.protected_variable;
}
int main()
{
   base object1;
   friendFunction(object1);
   return 0;
}
```

Inheritance:

The capability of a class to derive properties and characteristics from another class is called Inheritance. Inheritance is a feature or a process in which, new classes are created from the existing classes. The new class created is called “derived class” or “child class” and the existing class is known as the “base class” or “parent class”. The derived class now is said to be inherited from the base class. When we say derived class inherits the base class, it means, the derived class inherits all the properties of the base class, without changing the properties of base class and may add new features to its own. These new features in the derived class will not affect the base class. The derived class is the specialized class for the base class.

Modes of Inheritance: There are 3 modes of inheritance.

Public Mode: If we derive a subclass from a public base class. Then the public member of the base class will become public in the derived class and protected members of the base class will become protected in the derived class.

Protected Mode: If we derive a subclass from a Protected base class. Then both public members and protected members of the base class will become protected in the derived class.

Private Mode: If we derive a subclass from a Private base class. Then both public members and protected

members of the base class will become Private in the derived class.

Note: The private members in the base class cannot be directly accessed in the derived class, while protected members can be directly accessed.

Single Inheritance: In single inheritance, a class is allowed to inherit from only one class. i.e. one subclass is inherited by one base class only.

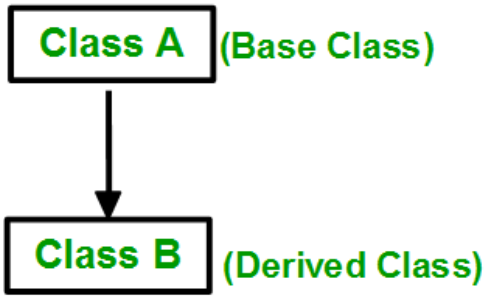

Figure 3.1: Single Inheritance

Programming Example for Single Inheritance:

```
class Animal {
  public:
 void eat() {
   cout<<"Eating..."<<endl;
 }
  };
  class Dog: public Animal
  {
     public:
    void bark(){
   cout<<"Barking...";
    }
  };
int main( ) {
   Dog d1;
   d1.eat();
   d1.bark();
   return 0;
}
```

Eating

Barking

Multiple Inheritance:

Multiple Inheritance is a feature of C++ where a class can inherit from more than one class. i.e. one subclass is inherited from more than one base class.

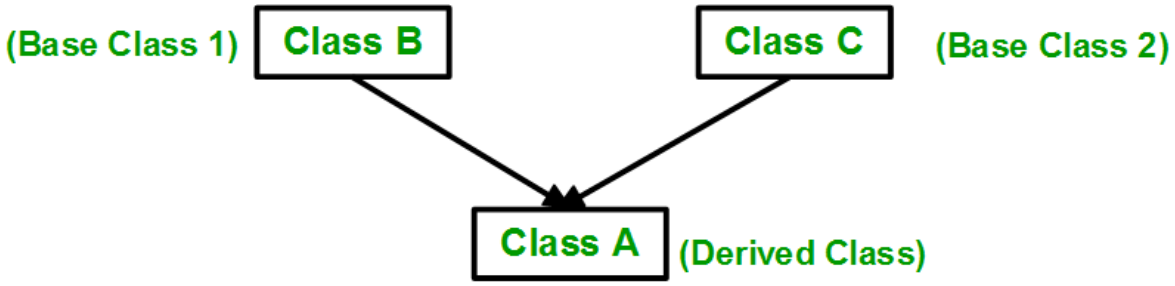

Figure 3.2: Multiple Inheritance

Programming Example for Multiple Inheritance:

```
class Shape {
protected:
int width, height;
public:
void setWidth(int w) {width = w;  }
void setHeight(int h) {height = h; }
};
class PaintCost {
  public:
    int getCost(int area) {
      return area * 70;
    }
};
class Rectangle: public Shape, public PaintCost {
  public:
    int getArea() {   return (width * height);  }
};
int main(void) {
  Rectangle Rect;
   int area;
  Rect.setWidth(5);
  Rect.setHeight(7);
  area = Rect.getArea();
cout << "Total area: " << Rect.getArea();
cout << "Total paint cost:" << Rect.getCost(area);
 return 0;      }
```

Multilevel Inheritance:

In this type of inheritance, a derived class is created from another derived class.

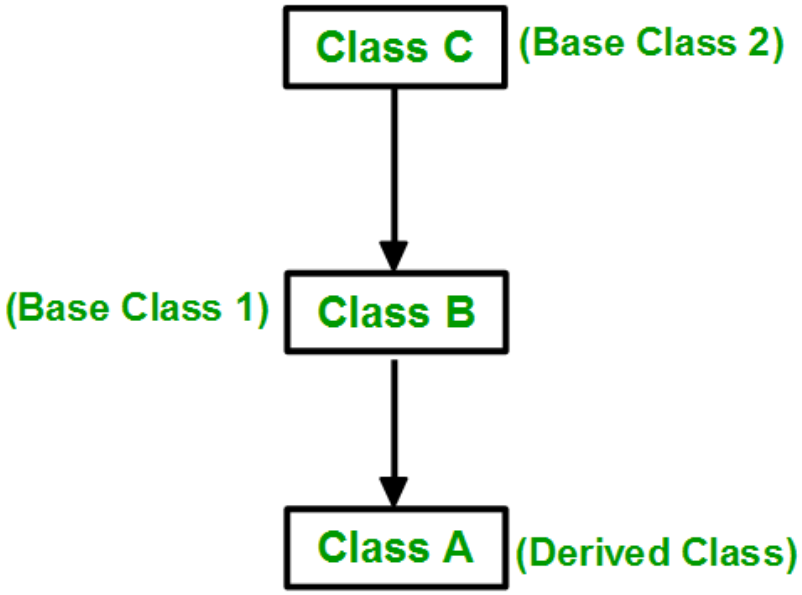

Figure 3.3: Multilevel Inheritance

Programming Example for Multilevel Inheritance:

```
class Vehicle {
public:
   Vehicle()
{ cout << "This is a Vehicle\n"; }
};
class fourWheeler : public Vehicle {
public:
   fourWheeler()
   {
cout << "Objects with 4 wheels are vehicles\n";
   }
};
class Car : public fourWheeler {
public:
   Car() { cout << "Car has 4 Wheels\n"; }
};
int main()
{
   Car obj;
   return 0;
}
```

This is a Vehicle
Objects with 4 wheels are vehicles
Car has 4 Wheels

Hierarchical Inheritance:

In this type of inheritance, more than one subclass is inherited from a single base class. i.e., more than one derived class is created from a single base class.

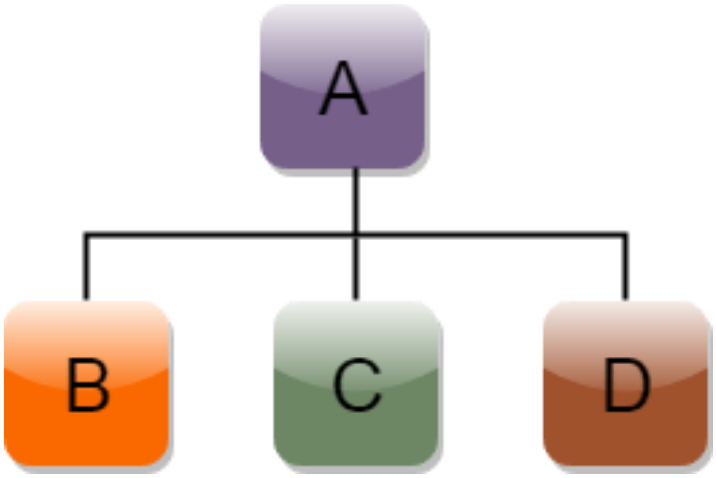

Figure 3.4: Hierarchical Inheritance

Programming Example for Hierarchical Inheritance:

```
class Shape
{
   public:
   int a;
   int b;
   void get_data(int n, int m)
   {
      a= n;
      b = m;
   }
};
class Rectangle : public Shape
{
   public:
   int rect_area()
   {
      int result = a*b;
      return result;
   }
};
class Triangle : public Shape
{
   public:
   int triangle_area()
   {
      float result = 0.5*a*b;
      return result;
   }
};
int main()
{
```

```
    Rectangle r;
    Triangle t;
    int length, breadth, base, height;
   cout << "Enter the length and breadth of a rectangle: " <<endl;
    cin>>length>>breadth;
    r.get_data(length, breadth);
int m = r.rect_area();
cout << "Area of the rectangle is : " <<m<<endl;
cout << "Enter the base and height of the triangle: " <<endl;
cin>>base>>height;
t.get_data(base, height);
float n = t.triangle_area();
cout <<"Area of the triangle is : "  << n<<endl;
    return 0;
}
```

Hybrid Inheritance:

Hybrid inheritance is a combination of more than one type of inheritance.

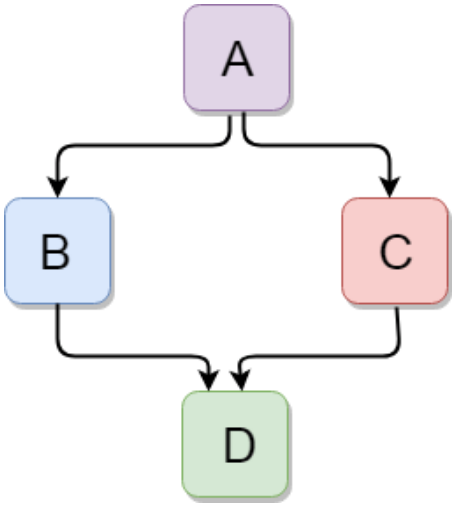

Figure 3.5: Hybrid Inheritance

Polymorphism:

The word “polymorphism” means having many forms. Polymorphism is the ability of a message to be displayed in more than one form.

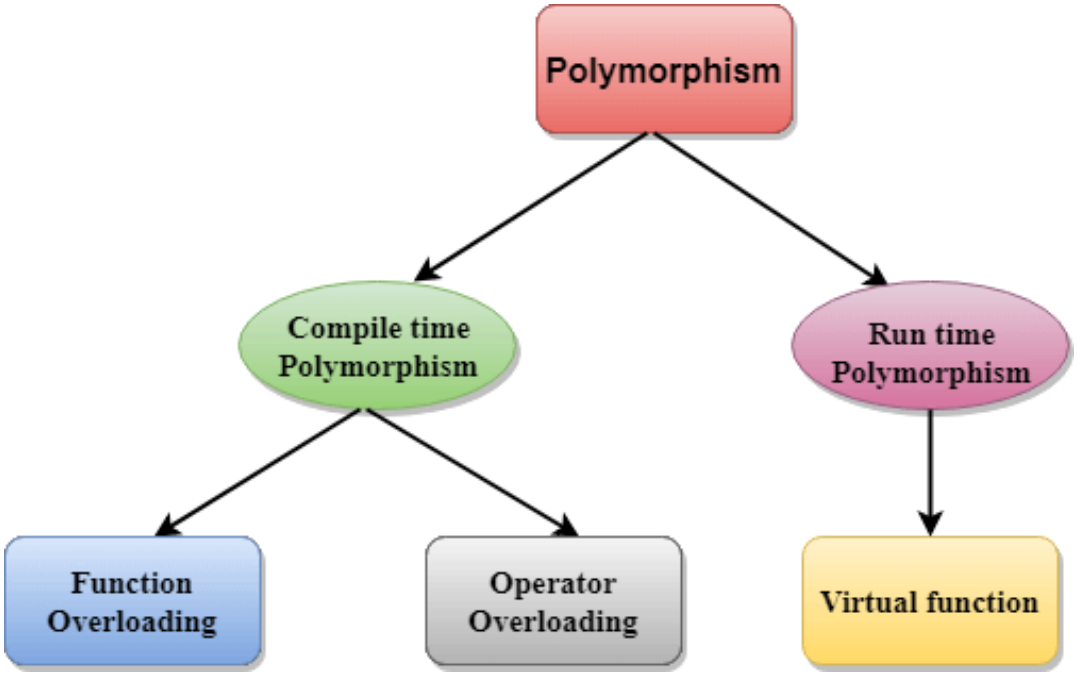

Figure 3.6: Types of Polymorphism

Function Overloading:

Function overloading is a feature of OOP where two or more functions can have the same name but different parameters. When a function name is overloaded with different jobs it is called Function Overloading. If multiple functions having same name but parameters of the functions should be different is known as Function Overloading. If we have to perform only one operation and having same name of the functions increases the readability of the program.

add(int a, int b)

add(double a, double b)

Programming Example for Function Overloading:

```
class Geeks {
public:
void func(int x)
   {
      cout << "value of x is " << x << endl;
   }
void func(double x)
   {
      cout << "value of x is " << x << endl;
   }
void func(int x, int y)
   {
      cout << "value of x and y is " << x << ", " << y<< endl;
   }
};
int main()
{
```

```
    Geeks obj1;
    obj1.func(7);
    obj1.func(9.132);
    obj1.func(85, 64);
    return 0;
}
```

Operator Overloading:

Operator overloading is a compile-time polymorphism. It is an idea of giving special meaning to an existing operator in C++ without changing its original meaning. In C++, we can make operators work for user-defined classes. This means C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading.

For example, we can overload an operator '+' in a class like String so that we can concatenate two strings by just using +.

Other example classes where arithmetic operators may be overloaded are Complex Numbers, Fractional Numbers, Big integers, etc.

Rules for Operator Overloading:

- Only built-in operators can be overloaded. If some operators are not present in C++, we cannot overload them.
- The precedence of the operators remains same.
- The overloaded operator cannot hold the default parameters except function call operator "()".
- We cannot overload operators for built-in data types. At least one user defined data types must be there.
- The assignment "=", subscript "[]", function call "()" and arrow operator "->" these operators must be defined as member functions, not the friend functions.
- Some operators like assignment "=", address "&" and comma "," are by default overloaded.

Programming Example for Operator Overloading:

```
class Complex {
private:
    int real, imag;
public:
    Complex(int r = 0, int i = 0)
    {
        real = r;
        imag = i;
    }
Complex operator+(Complex const& obj)
    {
        Complex res;
        res.real = real + obj.real;
res.imag = imag + obj.imag;
        return res;
```

```
    }
void print() { cout << real << " + i" << imag << endl; }
};
int main()
{
    Complex c1(10, 5), c2(2, 4);
    Complex c3 = c1 + c2;
    c3.print();
}
```

12 + i9

Function Overriding:

Function overriding in C++ is termed as the redefinition of base class function in its derived class with the same signature i.e. return type and parameters.

Programming Example for Function Overriding:

```
class base {
public:
    virtual void print()
    {
        cout << "print base class" << endl;
    }
     void show() {
        cout << "show base class" << endl;
    }
};
class derived : public base {
public:
// print () is already virtual function in
// derived class, we could also declared as    // virtual void print () explicitly
     void print() { cout << "print derived class" << endl; }
     void show() { cout << "show derived class" << endl; }
};
int main()
{
    base* bptr;
    derived d;
    bptr = &d;
     // Virtual function, binded at
    // runtime (Runtime polymorphism)
    bptr->print();
     // Non-virtual function, binded
    // at compile time
    bptr->show();
```

```
    return 0;
}
```

Output: print derived class
show base class

Virtual Function:

A virtual function (also known as virtual methods) is a member function that is declared within a base class and is re-defined (overridden) by a derived class. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the method. Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for the function call. They are mainly used to achieve Runtime polymorphism. Functions are declared with a virtual keyword in a base class. The resolving of a function call is done at runtime.

Data Structure Introduction:

Data Structure can be defined as the group of data elements which provides an efficient way of storing and organizing data in the computer so that it can be used efficiently. Data Structure is representation of the logical relationship existing between individual elements of data. A data structure is a way of organizing all data items that considers not only the elements stored but also their relationship to each other.

Classification of Data Structure:

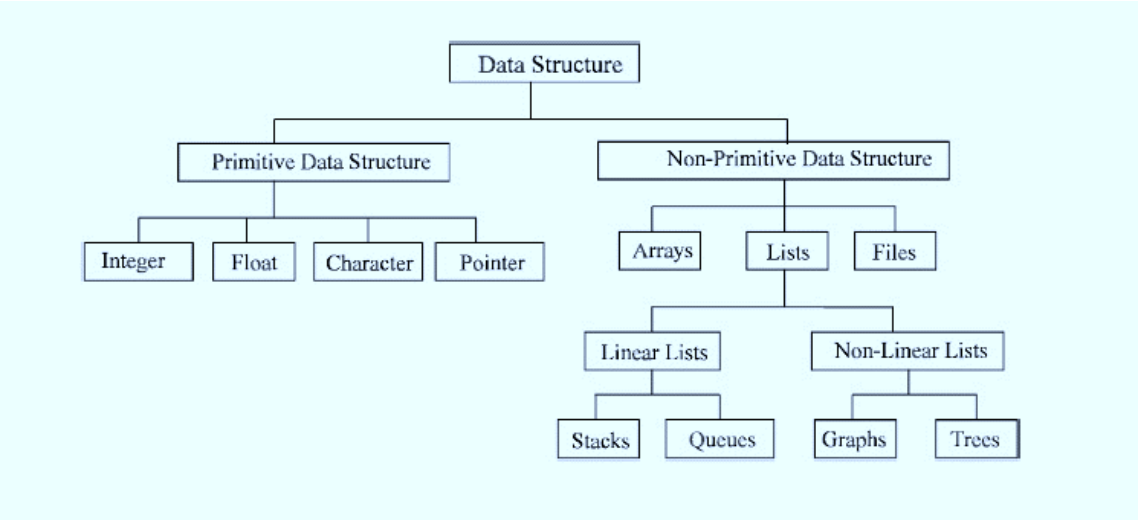

Figure 3.7: Data Structure Classification

Primitive: These are the structures which are **supported at the machine level**, they can be used to make non-primitive data structures.

Examples: Integer, float, character, pointers.

The pointers, however don't hold a data value, instead, they hold memory addresses of the data values. These are also called the reference data types.

Non-Primitive: The non-primitive data structures cannot be performed without the primitive data structures. Although, they too are provided by the system itself yet they are derived data structures and cannot be formed without using the primitive data structures.

File: A file is a collection of records. The file data structure is primarily used for managing large amounts of

data which is not in the primary storage of the system. The files help us to process, manage, access and retrieve or basically work with such data, easily.

Array: Arrays are a homogeneous and contiguous collection of same data types. They have a static memory allocation technique, which means, if memory space is allocated for once, it cannot be changed during runtime. If we do not know the memory to be allocated in advance then array can lead to wastage of memory.

List: The lists support dynamic memory allocation. The memory space allocated, can be changed at run time also. The lists are of two types: Linear and Non-Linear

Linear Lists: The linear lists are those which have the elements stored in a sequential order. The insertions and deletions are easier in the lists. They are divided into two types: Stack and Queue

Stack: The stack follows a **"LIFO"** technique for storing and retrieving elements. The element which is stored at the end will be the first one to be retrieved from the stack. The stack has the following primary functions:
Push(): To insert an element in the stack.
Pop(): To remove an element from the stack.

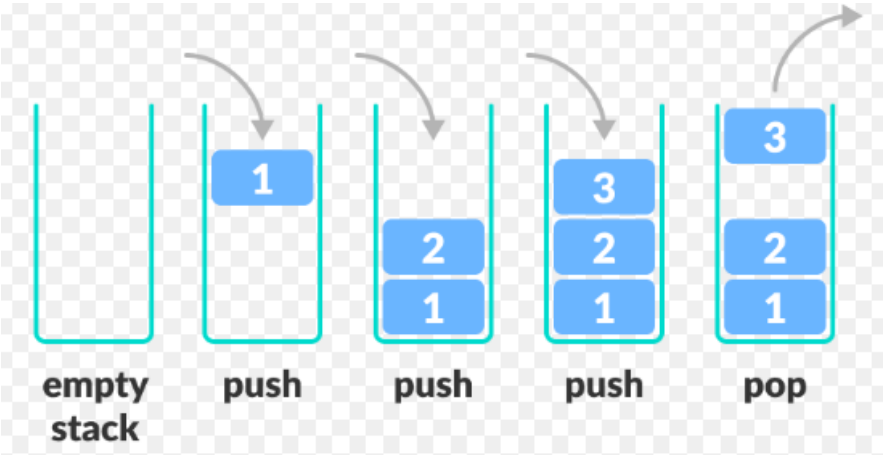

Figure 3.8: Stack

Queue: The queues follow **"FIFO"** mechanism for storing and retrieving elements. The elements which are stored first into the queue will only be the first elements to be removed out from the queue.
The **"ENQUEUE"** operation is used to insert an element into the queue
The **"DEQUEUE"** operation is used to remove an element from the queue.

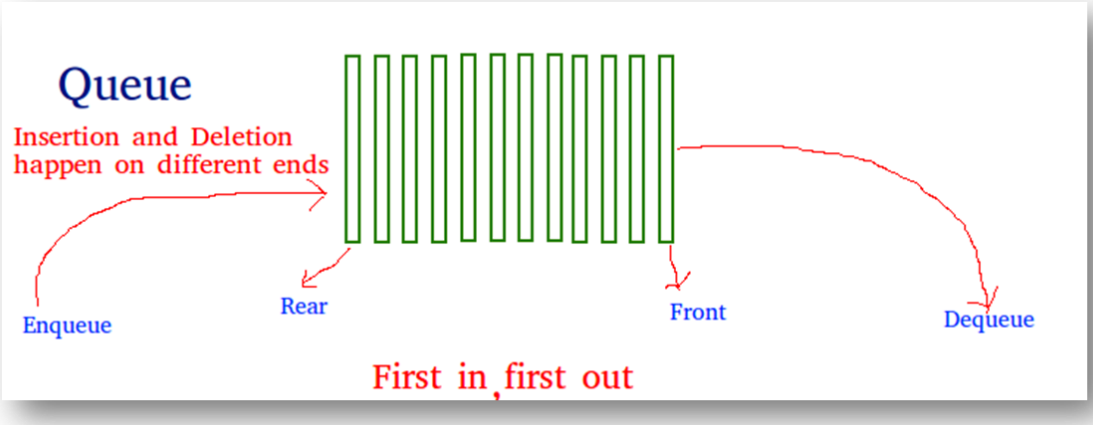

Figure 3.9: Queue

Linked List: The linked list is a linear data structure that contains a sequence of elements such that each element links to its next element in the sequence. Each element in a linked list is called "Node". Linked List can be defined as collection of objects called nodes that are randomly stored in the memory. A node contains two fields i.e. data stored at that particular address and the pointer which contains the address of the next node in the memory. The last node of the list contains pointer to the null.

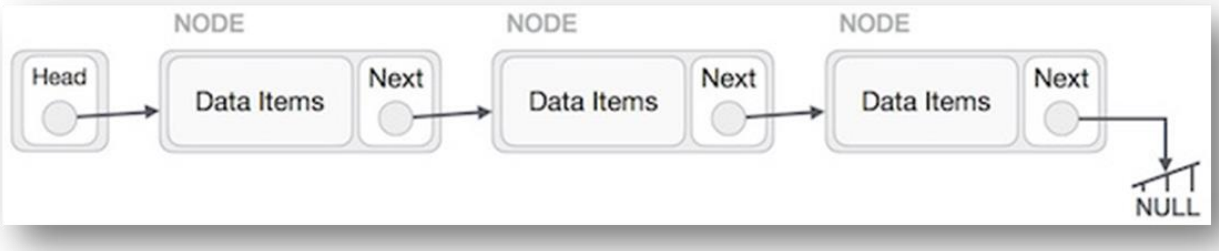

Figure 3.10: Linked List

Tree: Tree data structure comprises of nodes connected in a particular arrangement and they make search operations on the data items easy. The tree data structures consist of a root node which is further divided into various child nodes and so on.

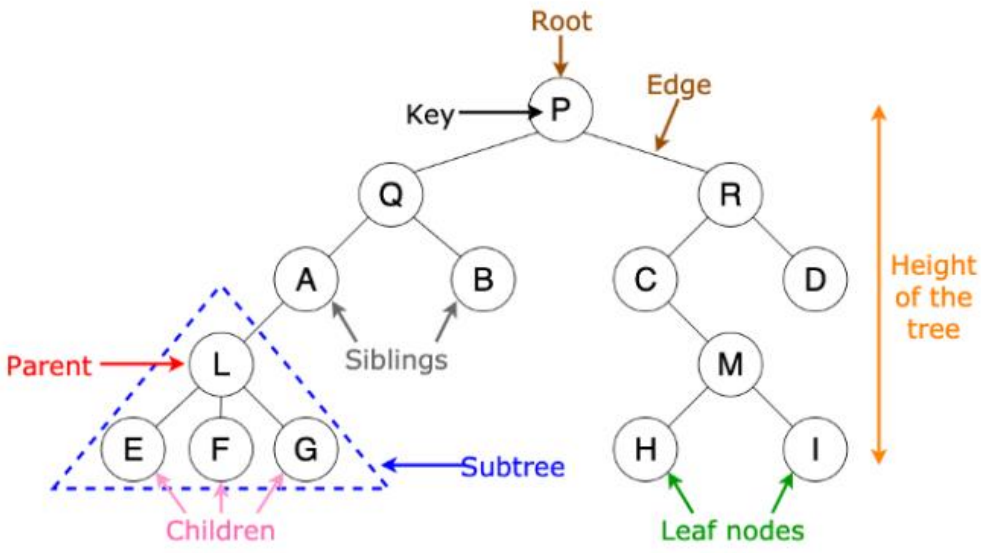

Figure 3.11: Tree

Graphs: The Graph data structure is used to represent a network. It comprises of vertices and edges (to connect the vertices). The graphs are very useful when it comes to study a network.

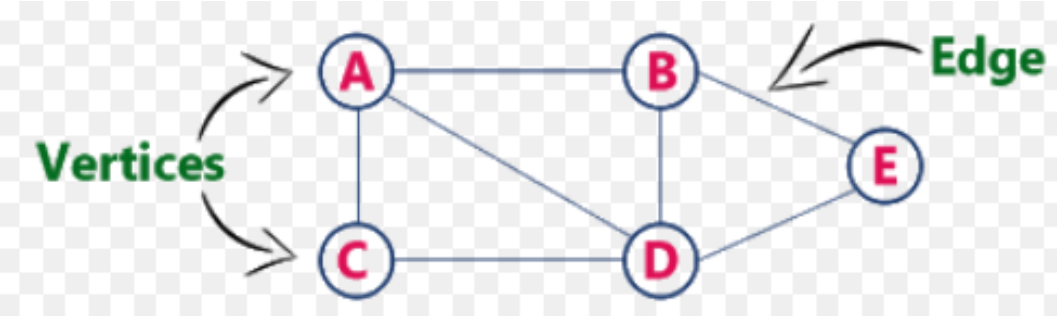

Figure 3.12: Graph

Operations on Data Structure:

Traversing: Every data structure contains the set of data elements. Traversing the data structure means visiting each element of the data structure in order to perform some specific operation like searching or sorting.

Insertion: Insertion can be defined as the process of adding the elements to the data structure at any location. If the size of data structure is n then we can only insert n-1 data elements into it.

Deletion: The process of removing an element from the data structure is called Deletion. We can delete an element from the data structure at any random location. If we try to delete an element from an empty data structure then underflow occurs.

Searching: The process of finding the location of an element within the data structure is called Searching. There are two algorithms to perform searching, Linear Search and Binary Search.

Sorting: The process of arranging the data structure in a specific order is known as Sorting. There are many algorithms that can be used to perform sorting, for example, insertion sort, selection sort, bubble sort, etc.

Merging: When two lists List A and List B of size M and N respectively, of similar type of elements, clubbed or joined to produce the third list, List C of size (M+N), then this process is called merging.