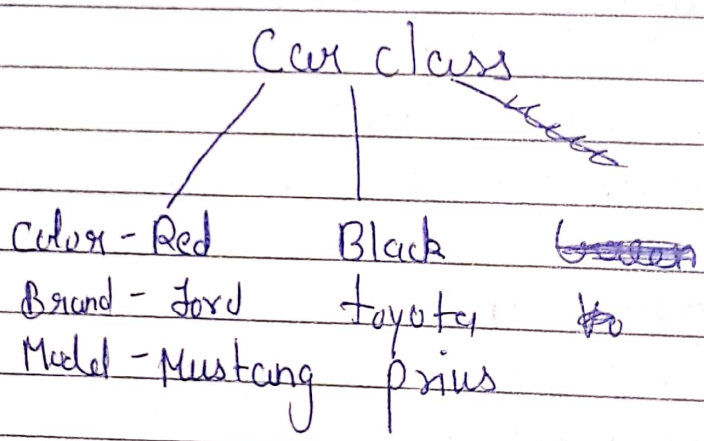
Unit => 3

Class & object :-

#class :-

* A class is a blueprint or template for creating objects.
* It defines the attributes and behaviors that the objects create from it will have



#Object :-

* An object is an instance of a class.
* It represents a real world entity and has attributes (data members) and (behaviours (member function).

How we declare a class?

Keyword ↓ class ClassName ← user-defined name

```
{
Access specifier: // can be private, public or protected
Data Members; // Variables
Member function () { } // Method to access data members
};                // class name ends with a semicolon
```

Code :— class & object

```
public:
string color;
string model;

void drive()
{
  cout << "The car is driving" << endl;
}
void brake()
{
  cout << "The car is braking" << endl;
}
};
int main()
{
// Creating an object of the class Car
```

```
Car myCar;
myCar.color = "Red";
myCar.model = "Toyota";
myCar.drive();          // Output: The car is drivin.
return 0;
}
```

Access Specifiers :—

Access specifiers define the accessibility of the members (attributes and methods) of a class.
The three access specifiers in c++ are:

Public:

* Members are accessible from outside the class.
* Syntax: public:

Private:

* Members are accessible only within the class.
* Default access specifier if not specified.
* Syntax: private:

Protected:

* Members are accessible within the class and by derived class members.
* Syntax: protected:

Difference Between class & object :-

object	class
1. Object is an instance of a class.	class is a blueprint of a object
2. Object is a real world entity.	class is group of similar object.
3. object is a physical entity.	class is a logical entity.
4. Object is created many times as per requirement.	class is declared once.
5. Object allocates memory when it is created.	class doesn't allocated memory when it is created

Scope resolution operator :-

The scope resolution operator (::) is used for several reasons. For example if the global variable name is same as local variable name, the scope resolution operator will be used to call the global variable. It is also used to define a function outside the class and used to access the static variables of class.

Here an example of scope resolution operator in C++ language,

Example :-

```
#include <iostream>
using namespace std;
char a = 'm';
static int b = 50;
int main()
{
  char a = 's';
  cout << "The static variable :" << ::b;
  cout << "\n the local variable :" << a;
  cout << "\n the global variable :" << ::a;
  return 0;
}
```

output :-
The static - 50
The local - s
The global - m

Other example:-

```
#include <iostream>
using namespace std;

int a=10;
int main()
{
  int a=5;
  cout << a;
  cout << ::a;
  return 0;
}
```

o/p → 5 10

Constructor & Destructor:-

A Constructor is a special member function of a class that is automatically called when an object of the class is created. It is used to initialize the objects of its class.

A Constructor is declared like a regular member function but with the following characteristics:-

* It has the same name as the class
* It does not have a return type, not even void.
* It can be defined inside or outside the class definition.

```
#include <iostream>
using namespace std;
class Person{
private:
  int age;
public:
  // Default Constructor
  Person()
  {
    age = 10;
    cout << " Default Constructor called. Age" << age << endl;
  }
  // Member function to display details
  void display()
  {
    cout << " Age:" << age << endl;
  }
};
int main()
{
  cout << " Creating Person1 with default
          constructor " << endl;
  Person Person1;
  person1.display();
  Return 0;
}
```

o/p :- default Constructor called
age = 10;
age: 10;

Type of Constructor :-

① Default Constructor :

* A Constructor that takes no arguments.
* If no constructor is defined, the compiler provides a default constructor.
* Used to initialize objects with default value.

```
class Person
{
  public :
    Person()
    {
      int age = 20;
    }
};
```

② Parameterized Constructor :

* A Constructor that takes one or more arguments.
* Allows passing values to initialize object attributes.

etc

```
class Person
{
 public:
  int age;
  Person (int a)
  {
   age = a;
  }
};
```

③ Copy Constructor:-

* A Constructor that initializes an object using another object of the same class.
* By default, the compiler provides a copy constructor.
* Used for creating a copy of an existing object.

```
class Person
{
 public:
  int age;
Person (const Person &p)
  {
   age = p.age;
  }
};
```

Destructor :-

* A Destructor is a special member function that is invoked automatically when an object is destroyed.

* The primary purpose of a destructor is to release resources that the object may have acquired during its lifetime, such as memory, file handles, or network connections.

* In c++, a destructor has the same name as the class but is preceded by a tilde (~).

Write a Program Demonstrating Constructor and Destructor in c++

```
#include <iostream>
using namespace std;
class Demo
{
public:
  Demo()
  {
  cout << "Constructor is called" << endl;
  }
  ~Demo()
  {
  cout << "Destructor is called" << endl;
  }
```

```
// Member function
void display()
{
  cout << " Display function is called " << endl;
}
};
int main()
{
  Demo obj; // Constructor is called here
  obj.display();
  return 0;
}
```

IMP

1. object and class
2. Friend function
3. Virtual function
4. Constructor & Destructor
5. Data structure
6. Inheritance

Friend function :-

* A friend function in c++ is a function that is not a member of a class but still has access to its private and protected members.

* A friend function can be a useful way to allow non-member function to access the internal state of a class.

```
#include<iostream>
using namespace std;

class Simple
{
private:
int data;
public:
Simple(int d) : data(d) { }

friend void displayData(Simple s);
};

void displayData(Simple s)
{
  cout << "Data :" << s.data << endl;
}

int main()
{
  Simple obj(5);
  displayData(obj);
  return 0;
}
```

Merits of friend function:-

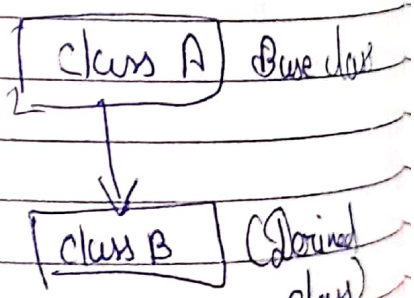
① Access private Data
② Improves code Modularity

Demerits of friend function:-

① Breaks Encapsulation
② Maintenance Complexity
③ Potential for Misuse

Inheritance :- Types of Inheritance

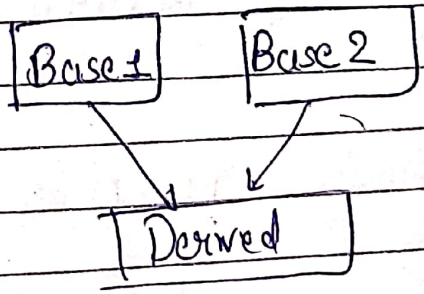
1. Single Inheritance :- is when a derived class inherits from only one base class.




```
# include <iostream>
using namespace std;
class Base
{
public:
    void display()
    {
        cout << " Base class display function" << endl;
    }
};
```

```
class Derived : public Base
{
  public:
  void show()
  {
    cout << " Derived class show function" <<endl;
  }
};
  int main()
  {
    Derived obj;
    obj. display(); // Accessing base class function
    obj. show(); // Accessing derived class function
    return 0;
  }
```

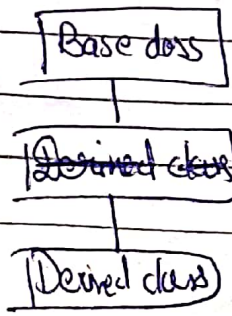
2. Multiple Inheritance :— is when a derived class inherits from more than one base class.




```
#include <iostream>
using namespace std;
class Base 1
{
  public:
  void display()
  {
    cout << " Base 1 class display function" <<endl;
  }
};
```

```
class Base2
{
 public:
  void show()
  {
   cout << " Base 2 class show function" << endl;
  }
};
 class Derived : public Base1, public Base2
  {
  };

int main()
{
 Derived obj;
 obj.display(); // Accessing Base1 class function
 obj.show(); // Accessing Base 2 class function
 return 0;
}
```

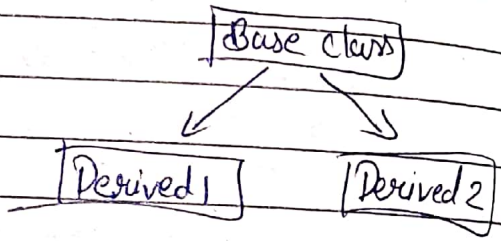
3. Multilevel Inheritance: is when a derived class is created from another derived class.



```
#include<iostream>
using namespace std;
class Base
{
  public:
    void display()
    {
      cout << " Base class display function" <<endl;
    }
};
class Intermediate : public Base
{
  public:
    void show()
{
cout << " Intermediate class show function"
                                    <<endl;
}
};
class Derived : public Intermediate
{
  public:
    void print()
    {
      cout << " Derived class print function"
                                    <<endl;
    }
};
int main()
{
Derived obj;
obj.display(); // Accessing Base class function
```

```
    obj.show();  // Accessing Intermediate class fun
    obj.print(); // Accessing Derived class function
    return 0;
}
```

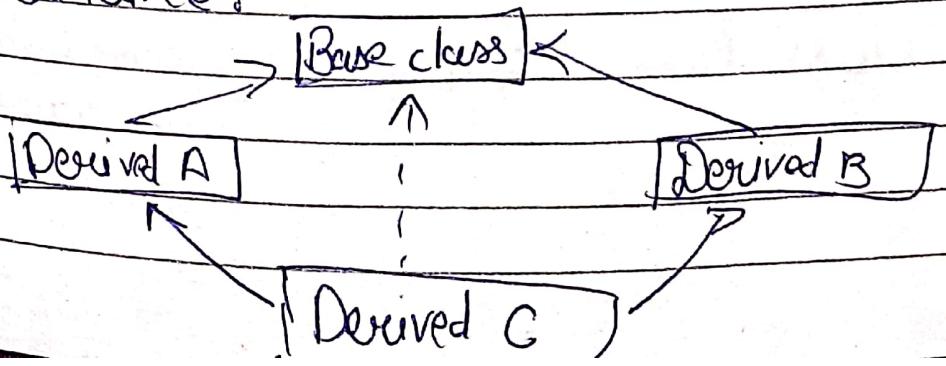
4. Hierarchical Inheritance :- Hierarchical Inheritance is when multiple derived classes inherit from a single base class.




```
# include <iostream>
using namespace std;
class Base
{
  public:
  void display()
  {
    cout << "Base class display function" << endl;
  }
};
class Derived1 : public Base
{
  public:
    void show()
    {
      cout << "Derived1 class show function" << endl;
    }
};
```

```
class Derived2 : public Base
{
public:
void print()
{
cout << " Derived 2 class print function" << endl;
}
};
int main()
{
Derived1 obj1;
Derived2 obj2;

obj1.display(); // Accessing Base class
obj1.show(); // Accessing Derived1 class
obj2.display(); // Accessing Base class
obj2.print(); // Accessing Derived2 class

return 0;
}
```

5. Hybrid Inheritance :- is a combination of two or more types of inheritance. It can be a mix of single, multiple, multilevel and hierarchical inheritance.

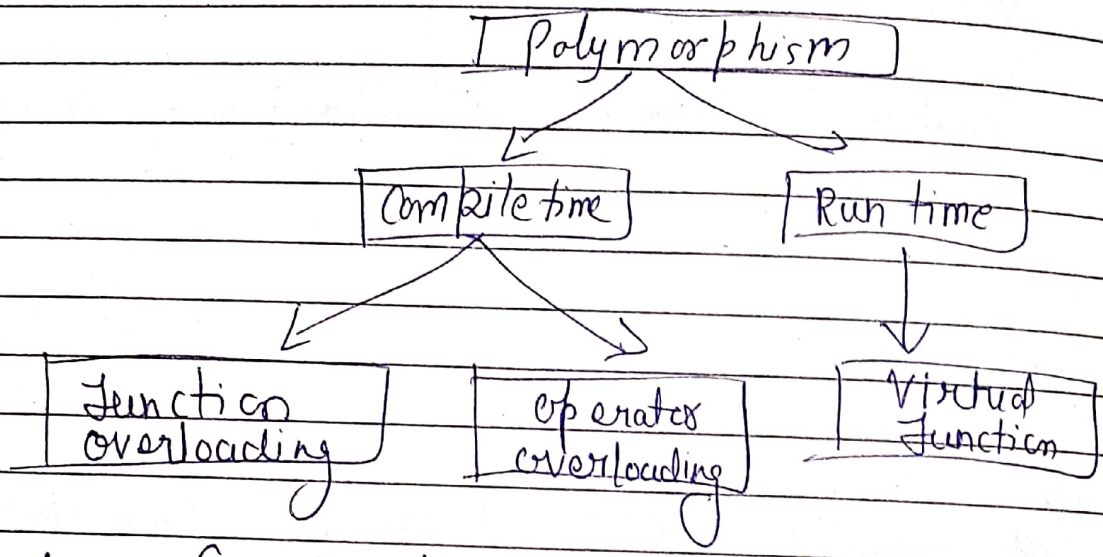


```
#include <iostream>
using namespace std;
class Base
{
public:
  void display()
  {
   cout << "Base class display function" << endl;
  }
};
class Intermediate 1: public Base
  {
     public:
       void show()
       {
        cout << "Intermediate 1 class show function" <<
                                                 endl;
       }
  };
class Intermediate 2: public Base
  {
     public:
       void print()
     {
       cout << " Intermediate 2 class print function"
                                          << endl;
     }
  };
class Derived : public Intermediate 1, public
                                  Intermediate 2
  {
```

```
public:
  void hello(
  {
   cout << "
  }
};

int main()
{
 Derived obj;
 obj.displ

 obj.sho

 obj.prin

 obj.he

 return
}
```

```
public:
    void hello()
    {
    cout << " Derived class hello function" << endl;
    }
};

int main()
{
 Derived obj;
 obj.display(); // Accessing Base class
                   function through intermediate1
 obj.show(); // Accessing Intermediate1
                class function
 obj.print(); // Accessing Intermediate2 es
                class function
 obj.hello(); // Accessing Derived class
                function

 return 0;
}
```

Polymorphism :-



1. Compile time polymorphism (Static polymorphism)

Compile time polymorphism is a type of polymorphism in which the function to be called is determined at compile time.

~~operator~~ function overloading

```
int Sum()
{
  int a=2
  int b=3
  int c=a+b;
}
```

```
int Sum (int a, int b)
{
   int c=a+b;
}
```

* This is achieved through function overloading and operator overloading.

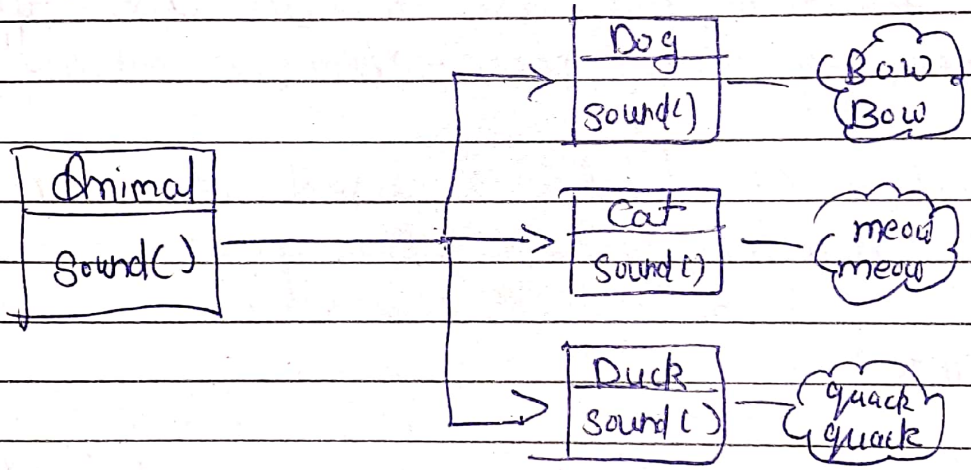
DATE
PAGE

Late Binding

Runtime polymorphism (Dynamic Programming)

→ Runtime polymorphism is a type of polymorphism in which the function to be called is determined at runtime.

→ This is achived through the use of virtual functions in c++.



Function Overloading :-

* Function Overloading is a feature in C++ that allows you to have more than one function with the same name in the same scope.

* The function must differs in the type or number of their parameters.

* Function overloading is used to enhance the readability of the program.

Key points :-

* Functions with the same name but different parameters.
* The return type of the functions can be the same or different.
* Helps in increasing the readability of the program.

```
#include <iostream>
using namespace std;
class Math
{
public:
// Function to add two integers
```

```
    int add(int a, int b)
    {
        return a+b;
    }
    // Function to add two doubles
    double add(double a, double b)
    {
        return a+b;
    }

    // Function to add three integers
    int add(int a, int b, int c)
    {
        return a+b+c;
    }
};
int main()
{
    Math math;
    cout << math.add(5,10) << endl; // call add(int,int)
    cout << math.add(3.5, 7.5) << endl; // call add(double, double)
    cout << math.add(1,2,3) << endl; // calls add(int, int, int)
    return 0;
}
```

Operator Overloading :-

Operator overloading is a compile time polymorphism in which the operator is overloaded to provide the special meaning to the user defined data type.

It is used to perform the operation on the user defined data type.

The advantage of Operators overloading is to perform different operations on the same operand.

Operator that cannot be overloaded are as follows :-

- Scope operator (::)
- Sizeof
- member selector (.)
- ternary operator (?:)

Syntax :-

```
return_type class_name :: operator
                          op(argument list)
{
// body of the function
}
```

```
#include <iostream>
using namespace std;
class Test
{
private:
  int num;
public:   // constructor to initialize count to 8
  Test() : num(8) { }
  void operator ++() // overload ++ When used as prefix
  {
    num= num+2;
  }
  void print()
  {
    cout << " The Count is: " << num;
  }
};
int main()
{
  Test tt;
  ++tt; // calling of the function "void operator ++()"
  tt.print();
  return 0;
}
```

O/p :- The Count is : 10

Virtual Function :-

⇒ When base class and its derived class contains same function name with same prototype then the function in base class is declared as virtual.

⇒ It is also called as function Overriding.

⇒ Syntax :-

```
virtual return-type function-name (arguments)
{
    // body of function
}
```

```
#include <iostream.h>
#include <conio.h>
class Base
{
public:
    virtual void display()
    {
        cout << "\n display base";
    }
};
class Derived public Base
{
public:
```

```
    virtual void display()
    {
    }
    cout << "\n display derived";
};
int main()
{
    Base b, *bptr;
    Derived d;

    cout << " bptr points to base \n";
    bptr = &b;
    bptr -> display();

    cout << " bptr points to derived \n";
    bptr = &d;
    bptr -> display();

    getch();
    return 0;
}
```

O/P ⟹

```
bptr points to base
display base
bptr point to derived
display derived.
```

- The virtual function should not be static.
- It must be member of some class.
- A virtual function can be declared as friend for another class.
- They can be accessed by using pointer object.
- Constructors cannot be declared as virtual, but destructors can be declared as virtual.

Data Structures :-

- A data structure is a way of organizing, managing and storing data so that it can be accessed and modified efficiently.
- Data structure are essential for designing efficient algorithm and are a fundamental concept in computer science.
- A data structure is not only used for organizing the data. It is also used for processing, retrieving and storing data.

Types of Data Structure :-

Data structure are broadly classified into two categories :-

① primitive Data structure :-

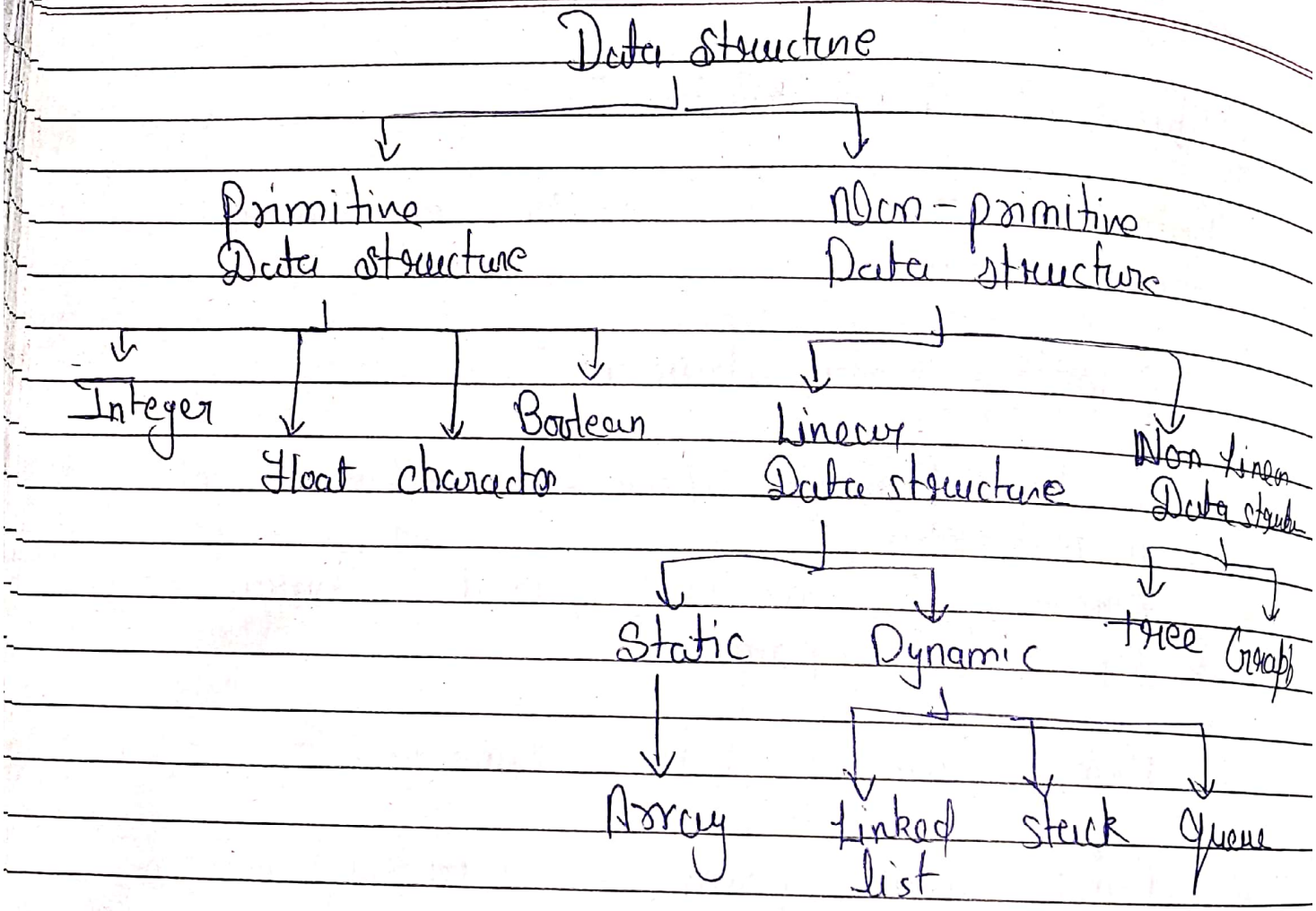
* Directly operated upon by machine level instructions.
* Example :- integers, floats, characters, pointers.

② Non - primitive Data Structure :-

* Built using primitive data structures
* Examples :- arrays, linked lists, stacks, queues, trees, graphs.

Non - primitive data structure are further divided into :-

1. linear Data structures
2. Non - linear Data structure.

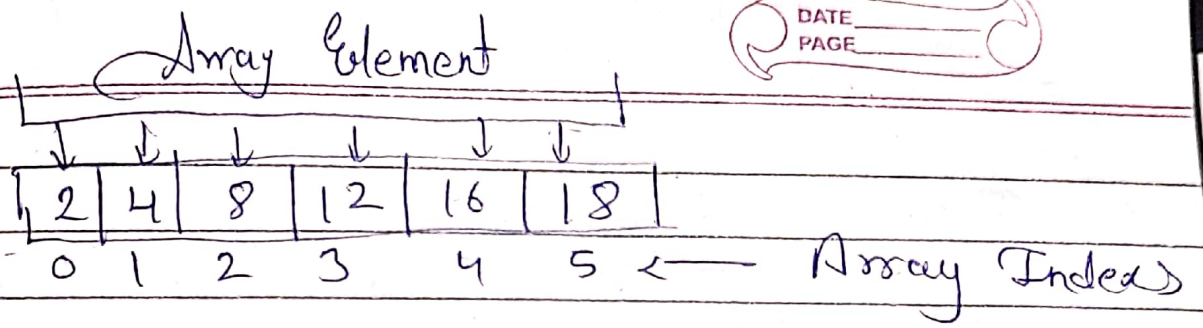


Linear Data structure :-

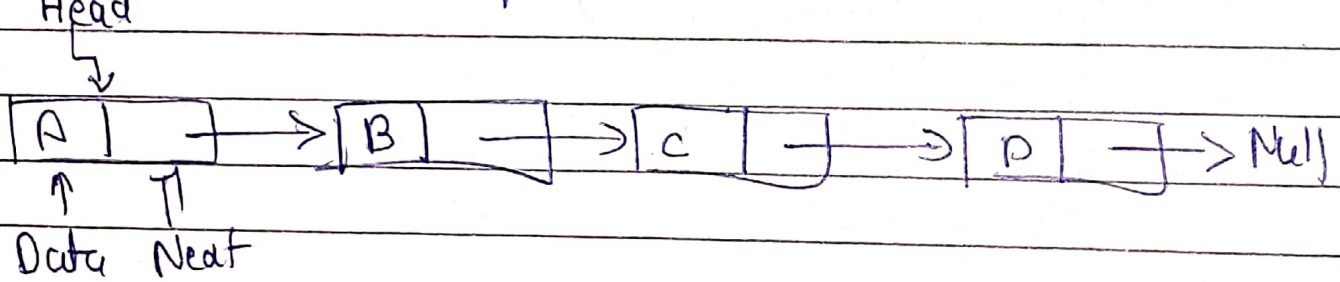
Linear data structure store data in a sequential manner, where each element is connected to its previous and next element.

1. Array :- A collection of element of the same type stored at contiguous memory locations.

Declaration :- int arr[10];

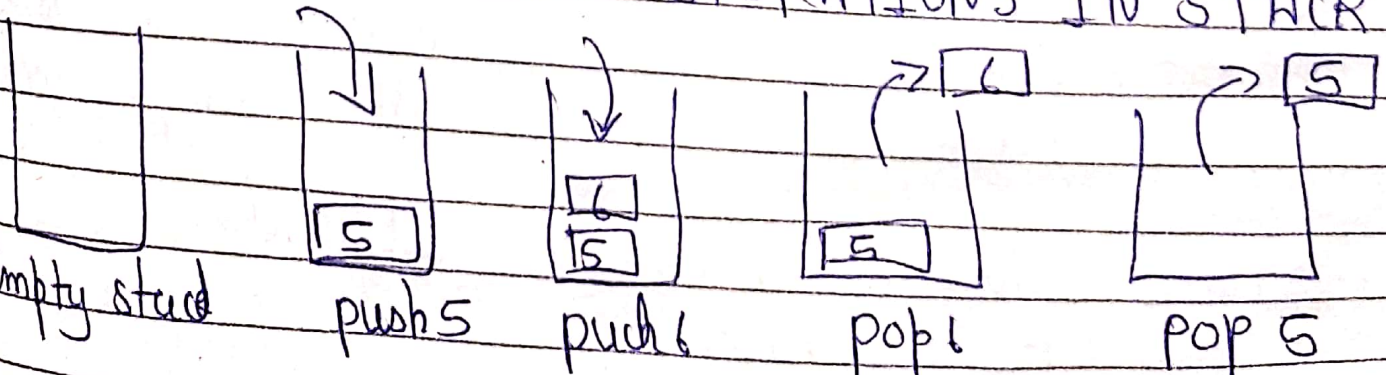


2. Linked list :- A collection of nodes where each node contains data and a reference (or link) to the next node in the sequence.

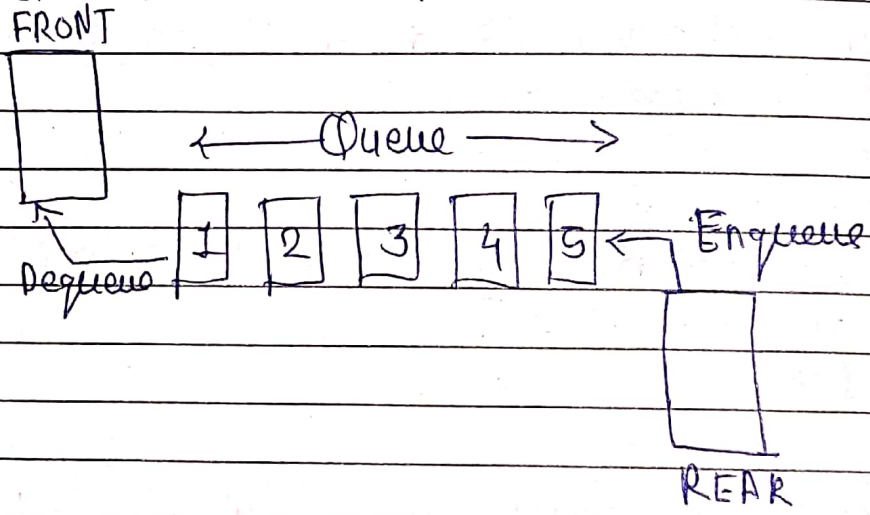


3. Stack :- A stack is a linear data structure that follows the Last in First Out (LIFO) principle. It means that the last element added to the stack will be the first one to be removed.

PUSH AND POP OPERATIONS IN STACK



4. Queue :- A queue is a linear data structure that follows the first In, First out (FIFO) principle. It means that the first element added to the queue will be the first one to be removed.



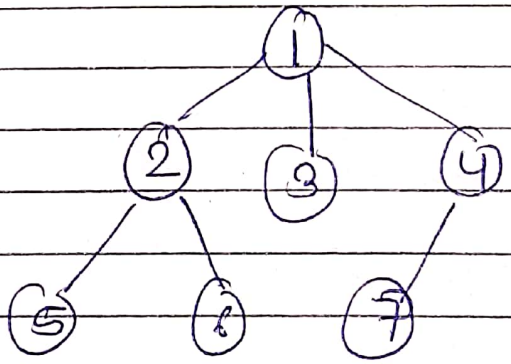
STACK VS QUEUE

	STACK	QUEUE
①	It represents the collection of elements in (FIFO)	It represent the collection of elements in (FIFO)
②	Insert operation is called Push Operation	Inset operation is called enqueue operation
③	Delete operation is called pop operation	Delete Data operation is called Dequeue.
④	In stack there is no wastage of memory space.	In queue there is a wastage of memory space.

6. Non linear Data Structure :-

1. Trees :- A tree is a hierarchical data structure consisting of nodes connected by edges, with the following properties :-

(1) Root Node :- The topmost node in the tree.
(2) parent Node :- A node that has branches to other nodes.
(3) child Node :- A node that is a descendent of another class.
(4) leaf Node :- A node with no children.
(5) Subtree :- A tree consisting of a node and its descendants.
(6) Edge :- The connection between two nodes.



General tree data structure

2. Graph :- A Graph is a collection of nodes (also called vertices) and edges that connect pairs of nodes. Unlike trees, graph can have cycle and multiple path between nodes