C++ PROGRAMMING

Abhishek Maurya

What is C++?

- C++ is a cross-platform language that can be used to create high-performance applications.
- C++ was developed by "*Bjarne Stroustrup*", as an extension to the C language.
- C++ gives programmers a high level of control over system resources and memory.
- The language was updated 3 major times in 2011, 2014, and 2017 to C++11, C++14, and C++17.

Why Use C++ ?

- C++ is one of the world's most popular programming languages.
- C++ can be found in today's operating systems, Graphical User Interfaces, and embedded systems.
- C++ is an object-oriented programming language which gives a clear structure to programs and allows code to be reused, lowering development costs.
- C++ is portable and can be used to develop applications that can be adapted to multiple platforms.
- C++ is fun and easy to learn!
- As C++ is close to C# and Java, it makes it easy for programmers to switch to C++ or vice versa

#include<iostream.h>

#include <iostream.h> is a header file library that lets us work with input and output objects, such as cout , cin. Header files add functionality to C++ programs.

Cout Function

- The cout object, together with the << operator, is used to output values/print text:
- It is similar to printf function

Cin Function

- cin is a predefined variable that reads data from the keyboard with the extraction operator (>>).
- It is similar to scanf function.

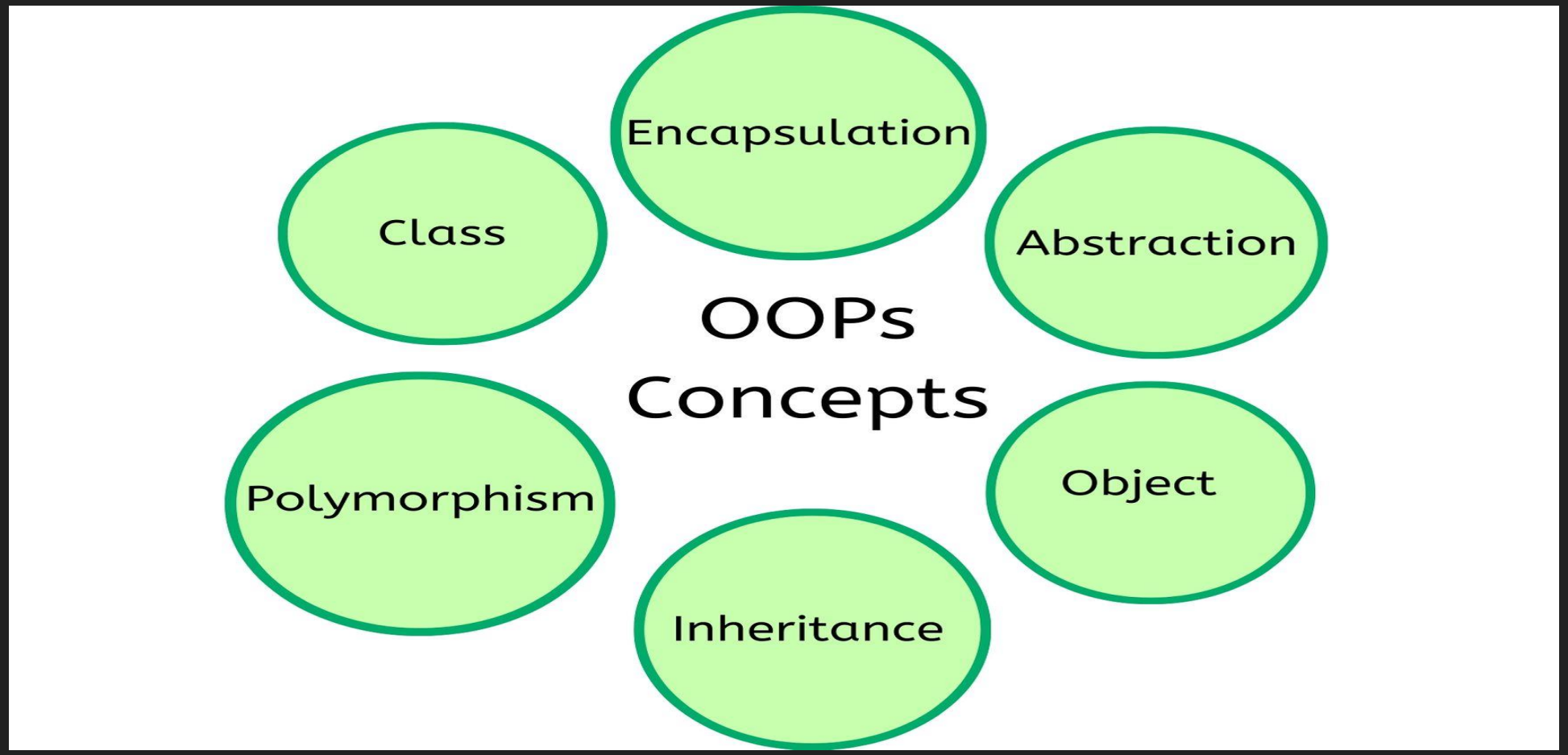
Program to print msg. ?

```
#include<iostream.h>
#include<conio.h>
Void main()
{
Cout<<“hello world….!”;
Getch();
}
```

OOPS

- Object-oriented programming – As the name suggests uses objects in programming. Object-oriented programming aims to implement real-world entities like inheritance, hiding, polymorphism, etc in programming. The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

Characteristics of an OOPS



CLASS

- Class: The building block of C++ that leads to Object-Oriented programming is a Class. 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 class is like a blueprint for an object.

OBJECT

- Object: An Object is an identifiable entity with some characteristics and behaviour. 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.

ENCAPSULATION

- Encapsulation: In normal terms, Encapsulation is defined as wrapping up of data and information under a single unit. In Object-Oriented Programming, Encapsulation is defined as binding together the data and the functions that manipulate them.

ABSTRACTION

- Abstraction: Data abstraction is one of the most essential and important features of object-oriented programming in C++. Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.

POLYMORPHISM

- Polymorphism: The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.
- A person at the same time can have different characteristic. Like a man at the same time is a father, a husband, an employee. So the same person posses different behaviour in different situations. This is called polymorphism.

INHERITANCE

- Inheritance: The capability of a class to derive properties and characteristics from another class is called Inheritance. Inheritance is one of the most important features of Object-Oriented Programming.
- Sub Class: The class that inherits properties from another class is called Sub class or Derived Class.

- Super Class: The class whose properties are inherited by sub class is called Base Class or Super class.
- Reusability: Inheritance supports the concept of “reusability”, i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

ACCESS SPECIFIERS (ACCESS MODIFIERS)

In C++, there are three access specifiers:

- `public` - members are accessible from outside the class
- `private` - members cannot be accessed (or viewed) from outside the class
- `protected` - members cannot be accessed from outside the class, however, they can be accessed in inherited classes. You will learn more about Inheritance later.

EXAMAPLE OF CLASS AND OBJECT

```
#Include<iostream.h>
#include<conio.h>
Class Abhi{ //defining class
Private: // access specifier
Int a,b; // data members
};
```

```
Int main()
{
Abhi a1; //creating object as a1
a1.a=12; // assigning values in data members
a1.b=13;
Cout<<"sum of members="<<a1.a+a1.b;
}
```

Sum of members=25

Example of Public and Private Access specifier

```
class MyClass {
public:     // Public access specifier
int x;    // Public attribute
private:    // Private access specifier
int y;    // Private attribute
};
```

```
int main() {
  MyClass myObj;
  myObj.x = 25;//Allowed
  myObj.y = 50;//Not allowed
  return 0;
}
```

By default, all members of a class are **PRIVATE** if you don't specify an access specifier:

Class with Methods

Methods are functions that belongs to the class.

There are two ways to define functions that belongs to a class:

- Inside class definition
- Outside class definition

Note: You access methods just like you access attributes; by creating an object of the class and using the dot syntax (`.`)

Inside class definition

```
class MyClass {          // The class
  public:                // Access specifier
// Method/function defined inside the class
    void myMethod() {
      cout << "Hello World!";
    }
};
```

```
int main() {
 // Create an object of MyClass
  MyClass myObj;
 // Call the method
  myObj.myMethod();
  return 0;
}
```

Outside class definition

```
class MyClass {        // The class
  public: // Access specifier

// Method/function declaration
    void myMethod();
};
// Method/function definition
outside the class
void MyClass::myMethod() {
  cout << "Hello World!";
}
```

```
int main() {
 // Create an object of MyClass
  MyClass myObj;
 // Call the method
  myObj.myMethod();
  return 0;
}
```

EXAMPLE OF PRIVATE WITH METHODS

```
class test{
  private:
     int a,b;
  public:
     void display(){
     cout<<"sum of a+b="<<a+b;
     }
     void setdata(int c,int d){
     a=c;
     b=d;
     }
};
```

```
int main()
{
test aa;
aa.setdata(12,13);
aa.display();
return 0;
}
```

C++ Constructor

CONSTRUCTOR is a special method which is invoked automatically at the time of object creation. It is used to initialize the data members of new object generally. The constructor in C++ has the same name as class or structure. There can be two types of constructors in C++.

- **Default constructor**
- **Parameterized constructor**

Default Constructor

A constructor which has no argument is known as default constructor. It is invoked at the time of creating object.

```
#include <iostream>
class Employee
 {
   public:
    Employee()
    {
    cout<<"Default Constructor Invoked";

    }
};
```

```
int main()
{
    Employee e1; //creating an object
     Employee e2;
    return 0;
}
```

Parameterized Constructor

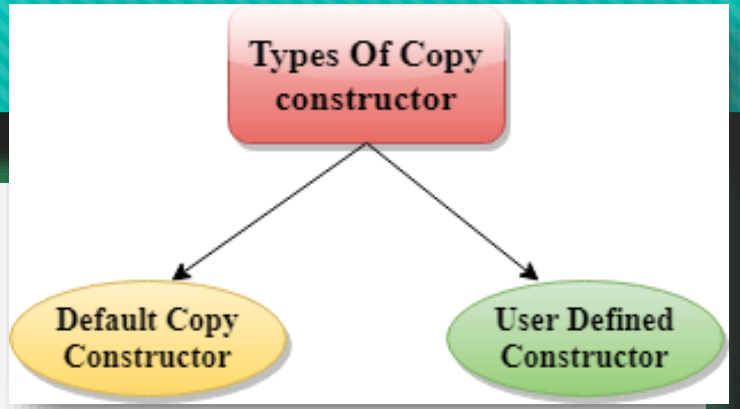
A constructor which has parameters is called parameterized constructor. It is used to provide different values to distinct objects.

```
#include<iostream.h>
class demo{
  int a,b;
  public:
    demo(int ac,int bc)
    {
        a=ac;
        b=bc;
    }
    void show(){
        cout<<"sum="<<a+b;
    }
};
```

```
void main(){
    demo a(12,13);
    a.show();
}
```

Copy Constructor

A Copy constructor is an overloaded constructor used to declare and initialize an object from another object.



Copy Constructor is of two types:

- **User Defined constructor:**

The programmer defines the user-defined constructor.

- **Default Copy constructor:**
- The compiler defines the default copy constructor. If the user defines no copy constructor, compiler supplies its constructor.

Syntax Of User-defined Copy Constructor:

```
class A
{
    A(A &x) //  copy constructor.
   {
       // copyconstructor.
   }
}
}
   }
```

```cpp
#include<iostream>
using namespace std;
class abhi{
    private:
        int age,sal;
    public:
        abhi(int a,int sal){ //parametrized constructor
            age=a;
        }
        void dis(){
            cout<<"age="<<age<<"sal"<<sal;
        }
        abhi(abhi &ref)//copy constructor
        {
            age=ref.age;
            sal=ref.sal;
        }
};
int main(){
    abhi aa(12,130);
    aa.dis();
    abhi aaa(aa); //copied object value
    aaa.dis();
    return 0;
}
```

When Copy Constructor is called

Copy Constructor is called in the following scenarios:

1)When we initialize the object with another existing object of the same class type. For example, Student s1 = s2, where Student is the class.

2)When the object of the same class type is passed by value as an argument.

3)When the function returns the object of the same class type by value.

Destructor

A destructor works **opposite to constructor**; it destructs the objects of classes. It can be defined only once in a class. Like constructors, it is invoked automatically.

A destructor is defined like constructor. It must have same name as class. But it is prefixed with a tilde sign (~).

Note: C++ destructor cannot have parameters. Moreover, modifiers can't be applied on destructors.

≡ File Edit Search Run Compile Debug Project Options Window Help

[■] AAAA.CPP 2=[↕]

```
#include<iostream.h>
#include<conio.h>
class Dash{
    public:
        Dash(){
        cout<<"constructor is calling";
        }
        ~Dash(){
        cout<<"destructor is calling";
        }

};
void main(){
        Dash d1;
        Dash d2;
}
```

15:26

F1 Help F7 Trace F8 Step F9 Make F10 Menu

```
#include <iostream>
using namespace std;
int num=0;
class Data{
  public:
    Data(){
        num++;
        cout<<"con is call for times="<<num<<endl;
    }
    ~Data(){
        cout<<"des is call for times="<<num<<endl;
        num--;
    }
};
int main()
{
    Data d,d2,d3;
    return 0;
}
```

Friend function

- If a function is defined as a friend function in C++, then the protected and private data of a class can be accessed using the function.

- By using the keyword friend compiler knows the given function is a friend function.

- For accessing the data, the declaration of a friend function should be done inside the body of a class starting with the keyword friend.

Declaration of friend function

```
class class_name
{
    friend data_type function_name(argument/s);          // syntax of friend function.
};
```

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

class Demo{
    int a,b;
    public:
     void insert(){
         cout<<"enter 2 values for a and b:";
         cin>>a>>b;
     }
     friend void sum(Demo obj);//declaring friend function
};
void sum(Demo obj){ //defining friend function
    int c;
    c=obj.a+obj.b;
    cout<<"sum of a and b="<<c;
}

int main()
{
    Demo obj2;
    obj2.insert();
    sum(obj2);
    return 0;
}
```

```
                    // write a program to make a loop using friend function
#include <iostream>
using namespace std;
class loop{
    int e;
    public:
        void input(){
            cout<<"enter ending for loop:";
            cin>>e;
        }
        friend void ll(loop obj2);
};
void ll(loop obj2){
    for(int i=0;i<=obj2.e;i++)
    {
        cout<<i<<endl;
    }
}
int main(){
    loop obj;
    obj.input();
    ll(obj);
    return 0;
}
```

Enumeration

Enum in C++ is a data type that contains fixed set of constants. It can be used for days of the week (SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY and SATURDAY) , directions (NORTH, SOUTH, EAST and WEST) etc. The C++ enum constants are static and final implicitly.

C++ Enums can be thought of as classes that have fixed set of constants.

Points to remember for C++ Enum

- Enum improves type safety
- Enum can be easily used in switch
- Enum can be traversed
- Enum can have fields, constructors and methods
- Enum may implement many interfaces but cannot extend any class because it internally extends Enum class

```
#include <iostream>
using namespace std;
    int main()
{
   enum days{mon,tue,wed,thur,fri,sat,sun}; //defining enum name days
   cout<<fri; //printing index as array
   days d=mon; //d is varivable for enum
   cout<<d;

    return 0;
}
```

THIS Pointer (Keyword)

In C++ programming, this is a keyword that refers to the current instance of the class. There can be 3 main usage of this keyword in C++.

- It can be used to pass current object as a parameter to another method.
- It can be used to refer current class instance variable.
- It can be used to declare indexers.

```
#include<iostream.h>
#include<conio.h>
class A{
        int a;
    public:
        void set_data(int a){
        this->a=a;
        }
        void show(){
        cout<<a;
        }
};
void main(){
clrscr();
A b;
b.set_data(12);
b.show();

}
```

```cpp
#include <iostream>
using namespace std;
class Employee {
   public:
       int id; //data member (also instance variable)
       string name; //data member(also instance variable)
       float salary;
       Employee(int id, string name, float salary)
        {
            this->id = id;
           this->name = name;
           this->salary = salary;
        }
       void display()
        {
           cout<<id<<"  "<<name<<"  "<<salary<<endl;
        }
};
int main(void) {
    Employee e1(101, "Sonoo", 890000); //creating an object of Employee
    Employee e2(102, "Nakul", 59000); //creating an object of Employee
    e1.display();
    e2.display();
    return 0;
}
```

Inheritance

Inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically. In such way, you can reuse, extend or modify the attributes and behaviors which are defined in other class.

In c++, the class which inherits the members of another class is called derived class and the class whose members are inherited is called base class. The derived class is the specialized class for the base class.

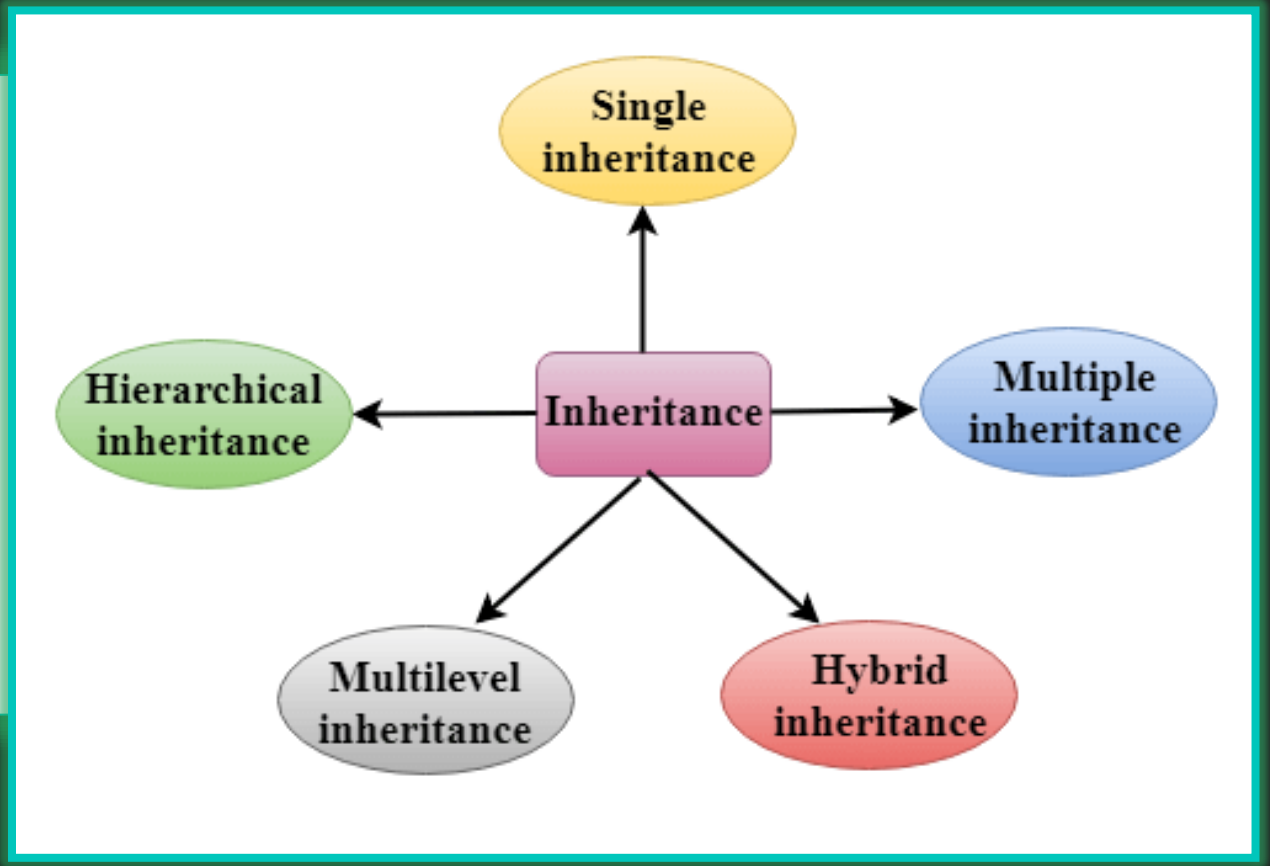
Advantage of C++ Inheritance

Code reusability: Now you can reuse the members of your parent class. So, there is no need to define the member again. So less code is required in the class.

Types Of Inheritance

five types of inheritance:

- Single inheritance
- Multiple inheritance
- Hierarchical inheritance
- Multilevel inheritance
- Hybrid inheritance



Derived Class

A Derived class is defined as the class derived from the base class.

The Syntax of Derived class:

```
class derived_class_name : visibility-mode base_class_name
{
    // body of the derived class.
}
```

derived_class_name: It is the name of the derived class.
visibility mode: The visibility mode specifies whether the features of the base class are publicly inherited or privately inherited. It can be public or private.

base_class_name: It is the name of the base class.

•When the base class is privately inherited by the derived class, public members of the base class becomes the private members of the derived class. Therefore, the public members of the base class are not accessible by the objects of the derived class only by the member functions of the derived class.

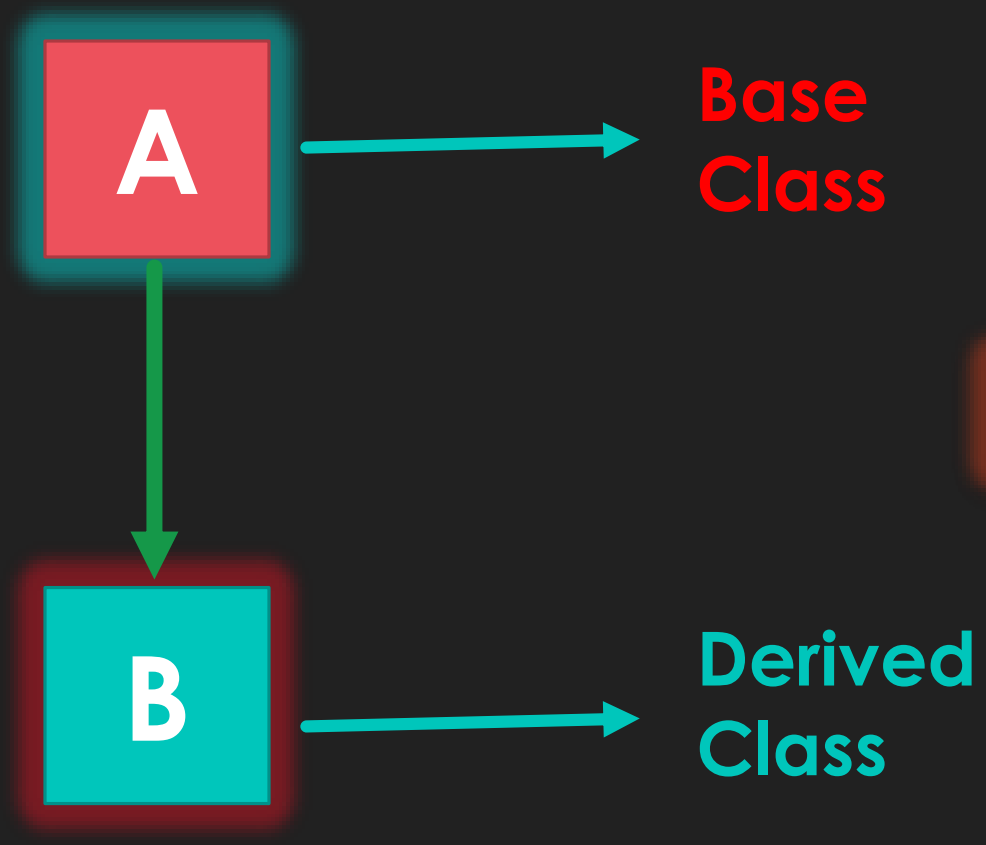
- When the base class is publicly inherited by the derived class, public members of the base class also become the public members of the derived class. Therefore, the public members of the base class are accessible by the objects of the derived class as well as by the member functions of the base class.

Note:

- In C++, the default mode of visibility is private.
- The private members of the base class are never inherited.

Single Inheritance

Single inheritance is defined as the inheritance in which a derived class is inherited from the only one base class.



Class A gives properties to Class B

```
#include<iostream>
using namespace std;
class A{
    public:
        int id;
        float sal;
            A(){
                cout<<"enter your id and sal:";
                cin>>id>>sal;
            }
};

class B:public A{
    public:
    string name="abhi";

};
int main(){

    B b1;
    cout<<"id for b class="<<b1.id;
    cout<<"sal for b class="<<b1.sal;
    cout<<"name for b class="<<b1.name;
}
```

METHODS INHERIT

```
#include<iostream>
using namespace std;
class A{
    public:
        int a,b;
    A(){
        cout<<"enter your 2 values:";
        cin>>a>>b;
    }
    int sum(){
    return a+b;
    }
};
class B:private A{
    public:
        void sh(){
            int res=sum();
            cout<<"res="<<res;
        }
};
int main(){
    B b1;
    b1.sh();
    return 0;
}
```

Multilevel Inheritance

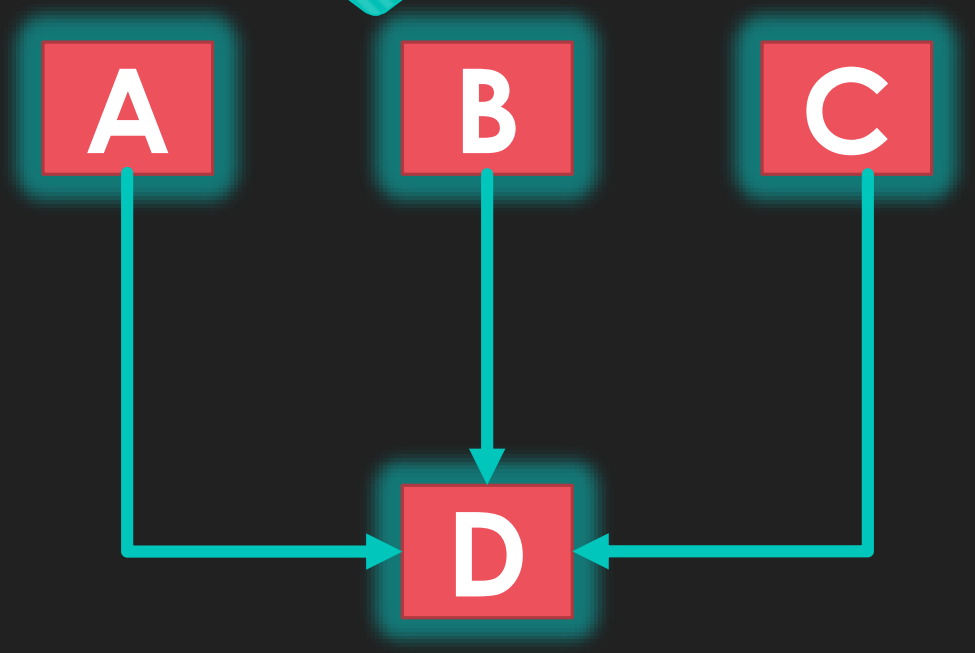
When one class inherits another class which is further inherited by another class, it is known as multi level inheritance in C++. Inheritance is transitive so the last derived class acquires all the members of all its base classes.

```
#include<iostream>
using namespace std;
class food{
    public:
    char name[30];
    void get_name();
};
void food::get_name(){
    cout<<"enter your food:";
    cin>>name;
}
class quantity:public food{
        public:
            int qua;
            void get_quantity();
};
void quantity::get_quantity(){
    cout<<"enter your"<< name <<"quantity:";
    cin>>qua;
}
```

```
class bill:public quantity{
  public:
    void total();
};
void bill::total(){
    cout<<"price of food is=70"<<endl;
    cout<<"quantity is:"<<qua<<endl;
    cout<<"total bill of food is:"<<70*qua;
}
int main(){
    bill b;
    b.get_name();
    b.get_quantity();
    b.total();
}
```

Multiple Inheritance

Multiple inheritance is the process of deriving a new class that inherits the attributes from two or more classes.



Syntax of the Derived class

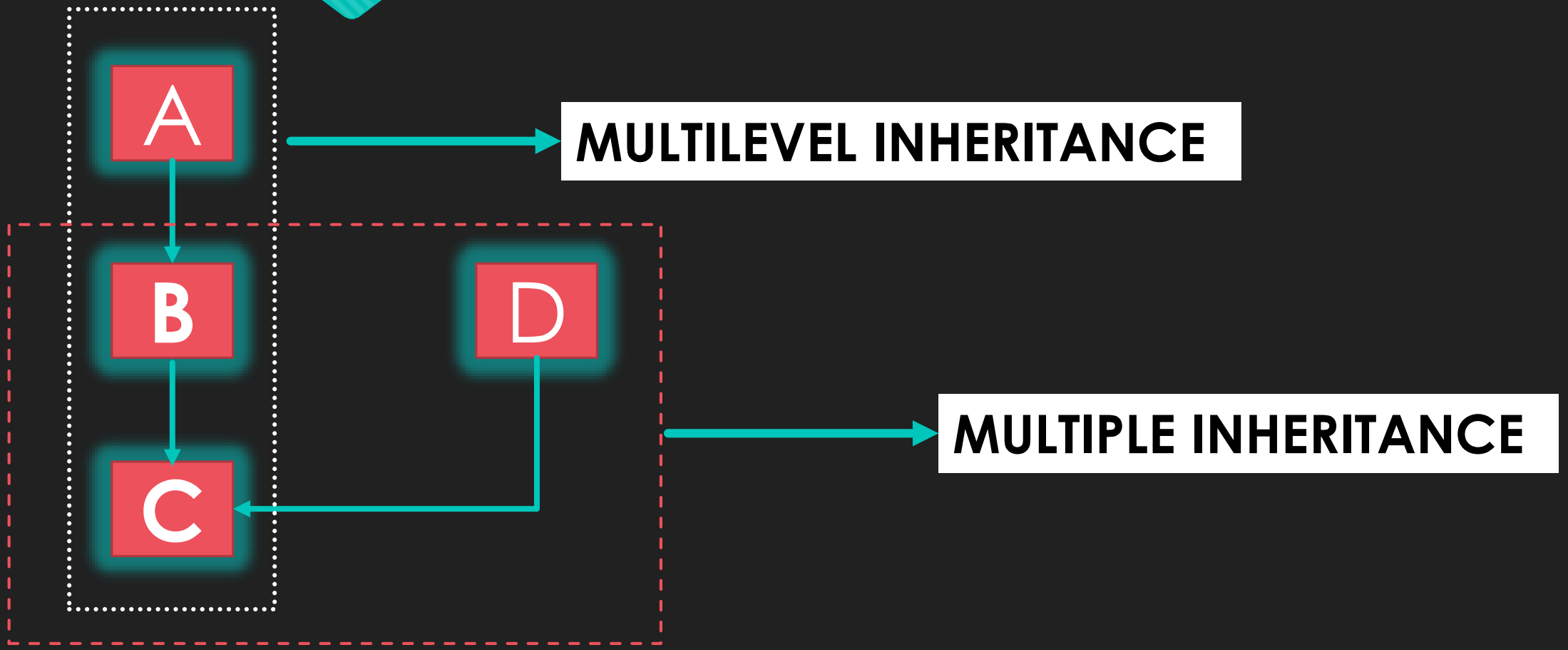
```
class D : visibility B-1, visibility B-2, ?
{
   // Body of the class;
}
```

```cpp
#include<iostream>
using namespace std;
class A{
    protected:
        int a;
    public:
    void dataofa(int n){
        a=n;
    }
};
class B{
    protected:
        int b;
    public:
    void dataofb(int n){
        b=n;
    }
};
```

```cpp
class C:public A,public B{

    public:
        void dis(){
            cout<<"pinting class a:"<<a<<endl;
            cout<<"printing class b:"<<b<<endl;
        }
};
int main(){
    C c1;
    c1.dataofa(12);
    c1.dataofb(13);
    c1.dis();
    return 0;
}
```

Hybrid Inheritance

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

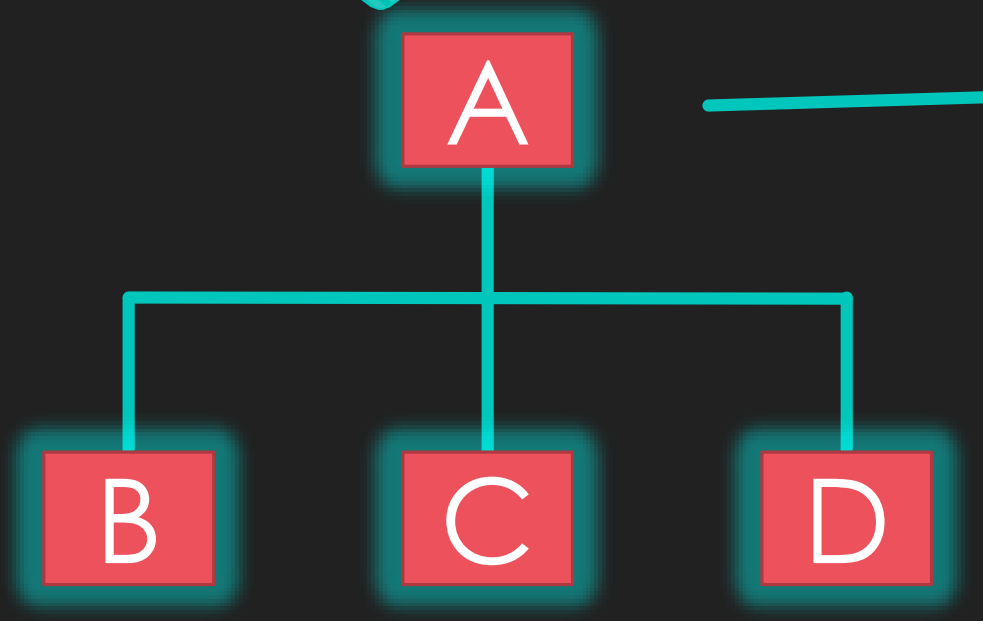


```cpp
#include<iostream>
using namespace std;
class A
{
    public:
        int a;
};

class B:public A
{
    public:
        int b;
};

class D
{
    public:
        int d;
};
```

```cpp
class C:public B, public D
{
    public:
        int c;
        void input ()
         {
            cout << "enter value of a,b,c,d:";
            cin >> a >> b >> c >> d;
         }
        void dis ()
        {
        cout << "class a=" << a << endl;
        cout << "class b=" << b << endl;
        cout << "class c=" << c << endl;
        cout << "class d=" << d << endl;
        }

};
int main(){
    C c1;
    c1.input();
    c1.dis();
    return 0;
}
```

Hierarchical Inheritance

Hierarchical inheritance is defined as the process of deriving more than one class from a base class.



A is a base class

B,C,D is a derived class

```cpp
#include<iostream>
using namespace std;
class Data{
    protected:
        int a,b;
    public:
        void input(){
            cout<<"enter 2 values:";
            cin>>a>>b;
        }

};

class addition:public Data{
    public:
        void add(){
            cout<<"add is here:"<<endl;
            input();
            cout<<"sum="<<a+b<<endl;
        }
};
```

```cpp
class subtraction:public Data{
    public:
        void sub(){
            cout<<"sub is here:"<<endl;
            input();
            cout<<"sub="<<a-b<<endl;
        }
};
int main(){
    addition a;
    a.add();
    subtraction s;
    s.sub();
    return 0;

}
```

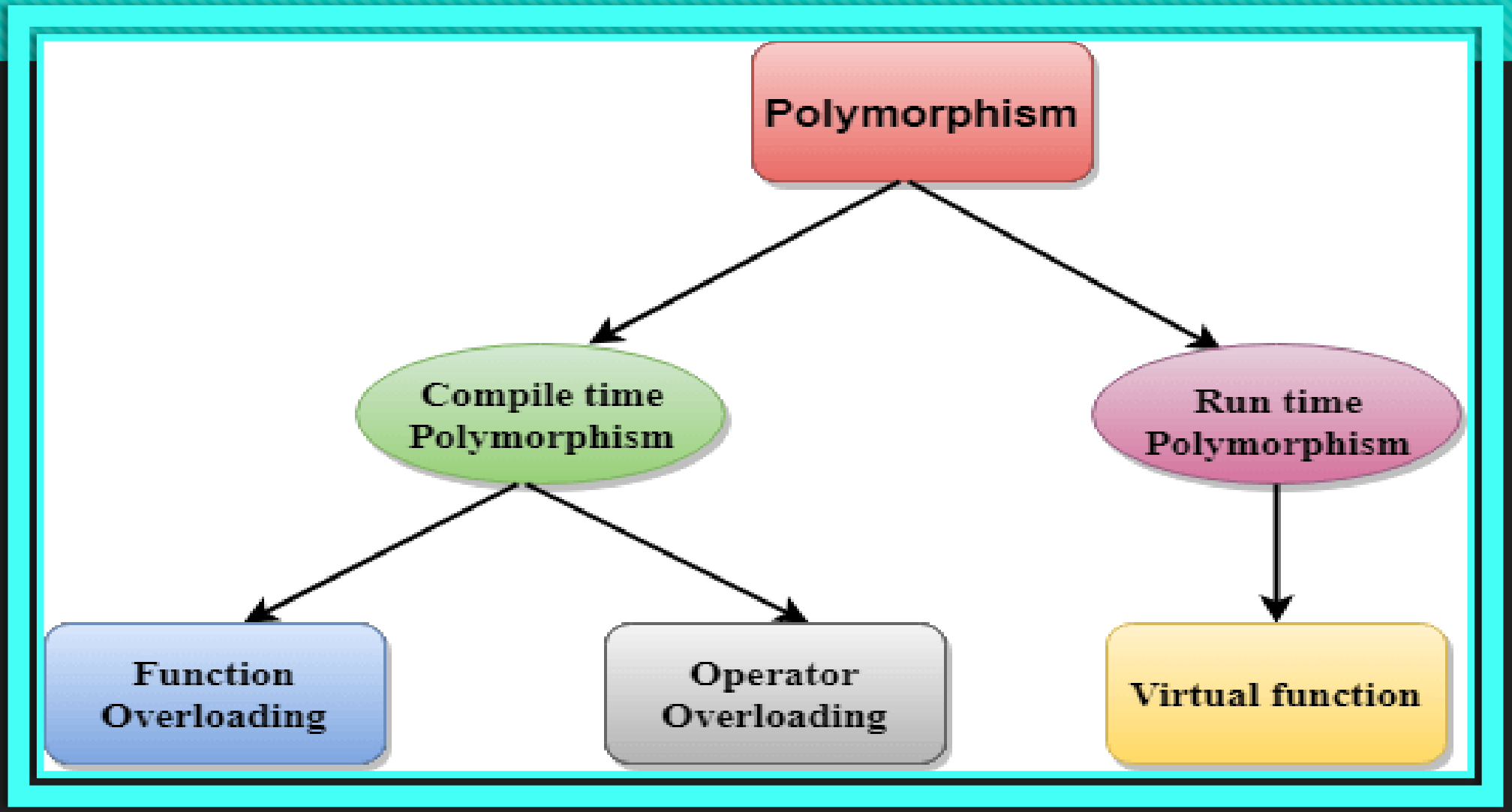
Polymorphism

The term "Polymorphism" is the combination of "poly" +"morphs" which means many forms. It is a Greek word.

Real Life Example Of Polymorphism

A lady behaves like a teacher in a classroom, mother or daughter in a home and customer in a market. Here, a single person is behaving differently according to the situations.

There are two types of polymorphism:



STATIC BINDING

Compile time polymorphism:

The overloaded functions are invoked by matching the type and number of arguments. This information is available at the compile time and, therefore, compiler selects the appropriate function at the compile time. It is achieved by function overloading and operator overloading which is also known as static binding or early binding. Now, let's consider the case where function name and prototype is same.

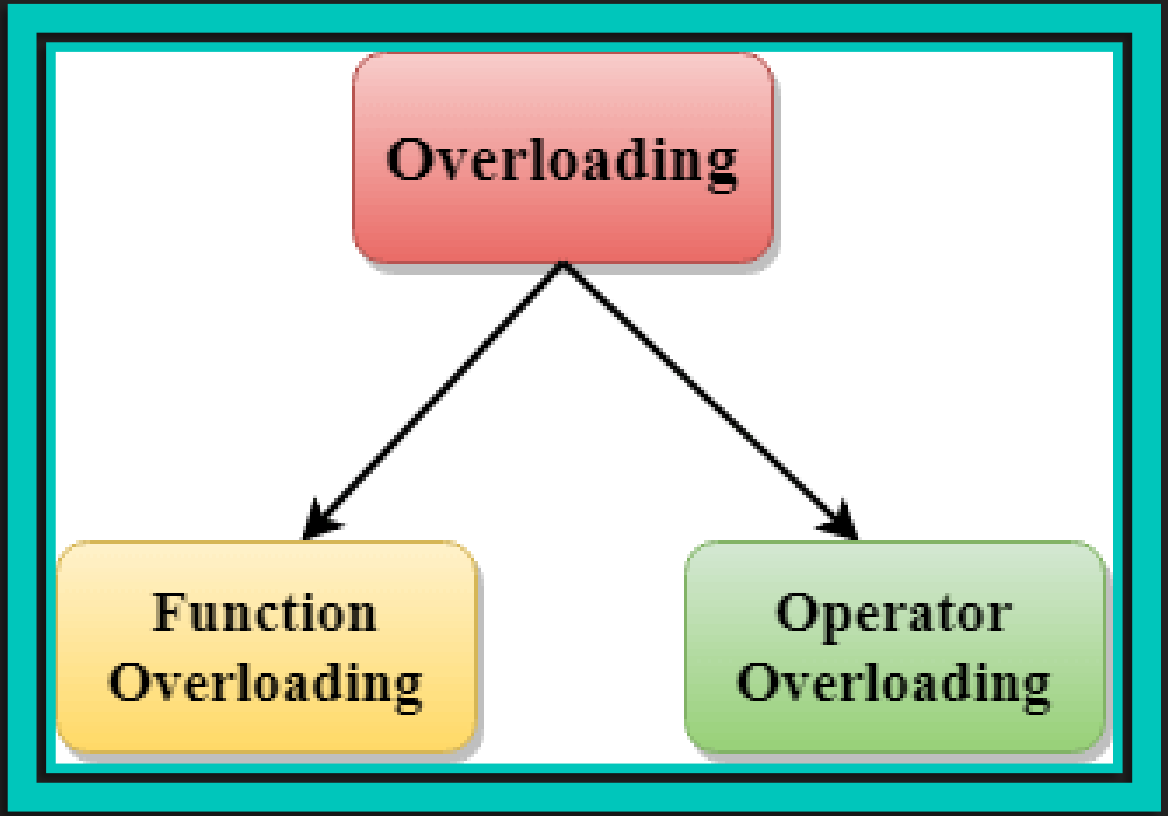
Overloading (Function and Operator)

If we create two or more members having the same name but different in number or type of parameter, it is known as C++ overloading. In C++, we can overload:

- methods,
- constructors, and
- indexed properties

Types of overloading in C++:

- Function overloading
- Operator overloading



Function Overloading

Function Overloading is defined as the process of having two or more function with the same name, but different in parameters is known as function overloading in C++. In function overloading, the function is redefined by using either different types of arguments or a different number of arguments. It is only through these differences compiler can differentiate between the functions.

```
#include<iostream>
using namespace std;
class A{
    public:
   int add(int a,int b){
        cout<<"function 1=";
        return a+b;
    }
   int add(int a,int b,int c){
        cout<<endl<<"function 2=";
        return a+b+c;
    }
};
int main(){
    A a1;
    cout<<a1.add(12,12);
    cout<<a1.add(1,2,3);
    return 0;
}
```

Operators 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. Operator overloading is used to overload or redefines most of the operators available in C++. It is used to perform the operation on the user-defined data type. For example, C++ provides the ability to add the variables of the user-defined data type that is applied to the built-in data types.

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(.)
- member pointer selector(*)
- ternary operator(?:)

Syntax of Operator Overloading(outside class)

```
return_type class_name  : : operator op(argument_list)
{
    // body of the function.
}
```

Rules for Operator Overloading

- Existing operators can only be overloaded, but the new operators cannot be overloaded.
- The overloaded operator contains atleast one operand of the user-defined data type.
- We cannot use friend function to overload certain operators. However, the member function can be used to overload those operators.
- When unary operators are overloaded through a member function take no explicit arguments, but, if they are overloaded by a friend function, takes one argument.
- When binary operators are overloaded through a member function takes one explicit argument, and if they are overloaded through a friend function takes two explicit arguments.

OPERATOR

Unary

Binary

Prefix operator:- (pre = before)

```
int n = 10;
cout<< ++n;
cout<< --n;
```

Postfix operator:- (post = before)

```
int n = 10;
cout<< n++;
cout<< n--;
```

INCREMENT OPERATOR OVERLOADING

```
using namespace std;
class try1{
    public:
        int a;
        try1(){
        a=1;
        }
         //increment operator overloading
        void operator++(int){
        a++;
        }
        void dis(){
        cout<<"value of a:"<<a<<endl;
        }
};
```

```
int main(){
    try1 t1;
    t1.dis();
    t1++;
    t1.dis();

return 0;
}
```

Binary operator overloading

Rule(left = calling)(right =argument)

```
#include<iostream>
using namespace std;
class task{
    int a;
public:
    task(int a1){
    a=a1;
    }
    void operator+(task t2){
    int m=a+t2.a;
    cout<<"sum="<<m;
    }
    void operator-(task t2){
    int m=a-t2.a;
    cout<<"sub="<<m;
    }

};
```

```
int main(){
    task t1(5);
    task t2(4);
    t1-t2;
    return 0;
}
```

RUN TIME POLYMORPHISM

DYNAMIC BINDING

Function Overriding

If derived class defines same function as defined in its base class, it is known as function overriding in C++. It is used to achieve runtime polymorphism. It enables you to provide specific implementation of the function which is already provided by its base class.

Single Inheritance overridding

```
#include<iostream>
using namespace std;
class test{
public:
    void show(){
    cout<<"i am in class test:";
    }
};
class test2:public test{
public:
    void show(){
    cout<<"i am in class test2:";
    }
};
int main(){
test2 t2;
t2.show();
t2.test::show();
return 0;
}
```

Multiple Inheritance overridding

```
#include<iostream>
using namespace std;
class test{
public:
    void show(){
    cout<<"i am in class test:";
    }
};
class test2{
public:
    void show(){
    cout<<"i am in class test2:";
    }
};
class test3:public test,public test2{
public:
    void show(){
    cout<<"i am in class test3:";
    }
};
```

```
int main(){
test3 t3;
t3.show();
t3.test2::show();//calling test2 show function
t3.test::show();//calling test show function

return 0;
}
```

virtual function

A virtual function is a member function which is declared within a base class and is re-defined(Overriden) 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 function.

- Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
- They are mainly used to achieve Runtime polymorphism
- Functions are declared with a virtual keyword in base class.
- The resolving of function call is done at Run-time.

Late binding or Dynamic linkage

In late binding function call is resolved during runtime. Therefore compiler determines the type of object at runtime, and then binds the function call.

Rules of Virtual Function

- Virtual functions must be members of some class.
- Virtual functions cannot be static members.
- They are accessed through object pointers.
- They can be a friend of another class.
- A virtual function must be defined in the base class, even though it is not used.
- The prototypes of a virtual function of the base class and all the derived classes must be identical. If the two functions with the same name but different prototypes, C++ will consider them as the overloaded functions.

```
#include<iostream>
using namespace std;
class A{
public:
    void show(){
    cout<<"showing something in base class."<<endl;
    }
    virtual void display(){
    cout<<"print something in base class."<<endl;
    }
};
class B : public A{
public:
    void show(){
    cout<<"showing something in derived class."<<endl;
    }
    void display(){
    cout<<"print something in derived class."<<endl;
    }
};
```

```
class C:public B{
void show(){
     cout<<"showing something in derived2 class."<<endl;
     }
     void display(){
     cout<<"print something in derived2 class."<<endl;
     }
};
int main(){
A *p,*p2;
B b1;
C c1;
p2=&c1;
p=&b1;

p->show();
p->display();
p2->show();
p2->display();
return 0;
}
```

Pure Virtual Function

- A virtual function is not used for performing any task. It only serves as a placeholder.
- When the function has no definition, such function is known as "do-nothing" function.
- The "do-nothing" function is known as a pure virtual function. A pure virtual function is a function declared in the base class that has no definition relative to the base class.
- A class containing the pure virtual function cannot be used to declare the objects of its own, such classes are known as abstract base classes.
- The main objective of the base class is to provide the traits to the derived classes and to create the base pointer used for achieving the runtime polymorphism.

```
#include <iostream>
using namespace std;
class Base
{
    public:
    virtual void show() = 0;
};
class Derived : public Base
{
    public:
    void show()
    {
        cout << "Derived class is derived from the base class." <<endl;
    }
};
int main()
{
    Base *bptr;
    //Base b;
    Derived d;
    bptr = &d;
    bptr->show();
    return 0;
}
```

Interfaces in C++ (Abstract Classes)

Abstract classes are the way to achieve abstraction in C++. Abstraction in C++ is the process to hide the internal details and showing functionality only. Abstraction can be achieved by two ways:

1. Abstract class
2. Interface

Abstract class and interface both can have abstract methods which are necessary for abstraction.

C++ Abstract class

In C++ class is made abstract by declaring at least one of its functions as <>strong>pure virtual function. A pure virtual function is specified by placing "= 0" in its declaration. Its implementation must be provided by derived classes.

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

class shape{
public:
    virtual void draw()=0;
};

class rectangle:public shape{
public:
    void draw(){
    cout<<"drawing rectangle:";
    }
};
class triangle:public shape{
public:
    void draw(){
    cout<<"drawing triangle:";
    }
};
```

```
int main(){
rectangle r;
triangle t;
r.draw();
t.draw();
return 0;
}
```

Data Abstraction in C++

- Data Abstraction is a process of providing only the essential details to the outside world and hiding the internal details, i.e., representing only the essential details in the program.

- Data Abstraction is a programming technique that depends on the separation of the interface and implementation details of the program.

Data Abstraction can be achieved in two ways:

Abstraction using classes
Abstraction in header files.

Abstraction using classes: An abstraction can be achieved using classes. A class is used to group all the data members and member functions into a single unit by using the access specifiers. A class has the responsibility to determine which data member is to be visible outside and which is not.

Abstraction in header files: An another type of abstraction is header file. For example, pow() function available is used to calculate the power of a number without actually knowing which algorithm function uses to calculate the power. Thus, we can say that header files hides all the implementation details from the user.

Abstraction in header files:

```
#include <iostream>
#include<math.h>
using namespace std;
int main()
{
 int n = 4;
   int power = 3;
   int result = pow(n,power);          // pow(n,power) is the  power function
   std::cout << "Cube of n is : " <<result<< std::endl;
   return 0;
}
```

Abstraction using classes:

```
#include <iostream>
using namespace std;
 class Sum
{
private: int x, y, z; // private variables
public:
void add()
{
cout<<"Enter two numbers: ";
cin>>x>>y;
z= x+y;
cout<<"Sum of two number is: "<<z<<endl;
}
};
```

```
int main()
{
Sum sm;
sm.add();
return 0;
}
```

C++ Strings

In C++, string is an object of string class that represents sequence of characters. We can perform many operations on strings such as concatenation, comparison, conversion etc.

```
#include<iostream>
using namespace std;
int main(){
 string a="abhi";
 cout<<a;
 char b[]={'a','b','h','i'}; //must use single inverted comma's
 cout<<b;
return 0;
}
```

String Compare

Example of string comparison using strcmp() function.

```
#include<iostream>
#include<cstring>
using namespace std;
int main(){
char n1[30],n2[30];
 int res;
 cout<<"enter first string:";
 cin>>n1;
 cout<<"enter second string:";
 cin>>n2;
res=strcmp(n1,n2);
```

C++ string class

```
if(res==0){
cout<<"are equal";
}
else if(res>0){
cout<<"ist is greater";
}
else{
cout<<"2nd is greater";
}
return 0;
}
```

String Concat

Example of string concatenation using strcat() function.

```
#include<iostream>
#include<cstring>
using namespace std;
int main(){
char str1[30],str2[30];
cout<<"enter 2 strings:";
cin>>str1>>str2;
strcat(strcat(str1," "),str2);
cout<<str1;
return 0;
}
```

String Copy

Example of copy the string using strcpy() function.

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
    char s1[25], s2[25];
    cout << "Enter string: ";
    cin>>s1;
    strcpy(s2, s1);
    cout << "copy = "<< s2<< endl;

    return 0;
}
```

String Length

Example of finding the string length using strlen() function.

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
    char ary[] = "Welcome to C++ Programming";
    cout << "Length of String = " << strlen(ary)<<endl;
    return 0;
}
```

C++ Recursion

When function is called within the same function, it is known as recursion in C++. The function which calls the same function, is known as recursive function.

A function that calls itself, and doesn't perform any task after function call, is known as tail recursion. In tail recursion, we generally call the same function with return statement.

```
recursionfunction(){
recursionfunction(); //calling self function
}
```

```
// sum of n numbers using recursion (recursive function)
#include<iostream>
using namespace std;
int sumofn(int);
int main(){
    int num;
    cout<<"enter any value:";
    cin>>num;
    int ans=sumofn(num);
    cout<<ans;
}
int sumofn(int a){
    if(a>0){
        return (a+sumofn(a-1));
    }
    else if(a==0){
        return 0;
    }
    else{

    }
}
```

```
//factorial of n numbers using recursion
#include<iostream>
int factn(int);
using namespace std;
int main(){
    int no;
    cout<<"enter any value:";
    cin>>no;
    int ans=factn(no);
    cout<<ans;
}
int factn(int a){
    if(a<0){
    return -1;
    }
else if(a==0){
    return 1;
}
else {
return (a*factn(a-1));
}

}
```

C++ Files

The **fstream** library allows us to work with files.
To use the fstream library, include both the standard <iostream> AND the <fstream> header file:

There are three classes included in the `fstream` library, which are used to create, write or read files:

Class	Description
`ofstream`	Creates and writes to files
`ifstream`	Reads from files
`fstream`	A combination of ofstream and ifstream: creates, reads, and writes to files

Create and Write To a File

To create a file, use either the ofstream or fstream class, and specify the name of the file. To write to the file, use the insertion operator (<<).

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

int main(){
    // Create and open a text file
    ofstream file("abhi.txt");
    // Write to the file
    file<<"i am abhi maurya";
    // Close the file
    file.close();
}
```

Read a File

To read from a file, use either the ifstream or fstream class, and the name of the file.
Note that we also use a while loop together with the getline() function (which belongs to the ifstream class) to read the file line by line, and to print the content of the file:

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

int main(){
    // Create a text string, which
    //is used to output the text file
     string s1;
     // Read from the text file
     ifstream file("abhi.txt");
     // Use a while loop together with
     //the getline() function to read the file line by line
     while(getline(file,s1)){
        // Output the text from the file
         cout<<s1;
     }
     // Close the file
     file.close();
}
```

Read and write files

```
#include<iostream>
#include<fstream>
#include<cstring>
using namespace std;
int main(){
    string s1;
    ofstream myfile("abhi.txt");
    myfile<<"i am at iics";
    myfile.close();
    ifstream my("abhi.txt");
    while(getline(my,s1)){
    cout<<s1;
    }
    my.close();

}
```

FSTREAM FILE HANDLING(WRITE MODE)

```
#include<iostream>
#include<fstream>
using namespace std;
int main(){
    fstream file;
    //ios::out is used for write mode
    file.open("abhi.txt",ios::out);
    if(file.is_open()){
        file<<"i am boss\n";
        file.close();
        }
    return 0;
}
```

FSTREAM FILE HANDLING(APPEND MODE)

```
int main(){
    fstream file;
    //ios::out is used for write mode
    file.open("abhi.txt",ios::out);
    if(file.is_open()){
        file<<"i am boss\n";
        file.close();
        }
    //ios::app is used for append mode
    file.open("abhi.txt",ios::app);
    if(file.is_open()){
        file<<"i am loss\n";
        file.close();
        }

    return 0;
}
```

FSTREAM FILE HANDLING(READ MODE)

```
int main(){
    fstream f1;
    //ios::Init is used to Read mode
    f1.open("abhi.txt",ios::in);
    if(f1.is_open())
    {
    string str;
    while(getline(f1,str)){
        cout<<str<<endl;
    }
    f1.close();
    }
return 0;
}
```

C++ Templates

A C++ template is a powerful feature added to C++. It allows you to define the generic classes and generic functions and thus provides support for generic programming. Generic programming is a technique where generic types are used as parameters in algorithms so that they can work for a variety of data types.

Templates can be represented in two ways:

- Function templates
- Class templates

Function Templates:

We can define a template for a function. For example, if we have an add() function, we can create versions of the add function for adding the int, float or double type values.

Class Template:

We can define a template for a class. For example, a class template can be created for the array class that can accept the array of various types such as int array, float array or double array.

Function Template

- Generic functions use the concept of a function template. Generic functions define a set of operations that can be applied to the various types of data.
- The type of the data that the function will operate on depends on the type of the data passed as a parameter.
- For example, Quick sorting algorithm is implemented using a generic function, it can be implemented to an array of integers or array of floats.
- A Generic function is created by using the keyword template. The template defines what function will do.

Syntax of Function Template

```
template < class Ttype>
ret_type func_name(parameter_list)  {
    // body of function.
}
```

Function TEMPLATE

```
#include<iostream>
using namespace std;
template<class A>    //function template
void show(A a1,A a2){
    cout<<a1<<" "<<a2;
}
int main(){
    char a,b;
    cout<<"enter 2 values:";
    cin>>a>>b;
    show(a,b);
    return 0;
}
```

CLASS TEMPLATE

A class template must be declared before any instantiation of a corresponding template class. A class template definition can only appear once in any single translation unit. A class template must be defined before any use of a template class that requires the size of the class or refers to members of the class.

```
#include<iostream>
using namespace std;
template<class T>
class sh{
    T a,b;
    public:
        sh(T a1,T b1){
            a=a1;
            b=b1;
            }
            void print(){
                cout<<a<<b;
            }
};
int main(){
    sh<int> sh1(34,76);// CLASS <DATATYPE> OBJ(PARAMETER)
    sh1.print();

    return 0;
}
```

C++ Exceptions Handling

When executing C++ code, different errors can occur: coding errors made by the programmer, errors due to wrong input, or other unforeseeable things.

When an error occurs, C++ will normally stop and generate an error message. The technical term for this is: C++ will throw an exception (throw an error).

C++ try and catch

Exception handling in C++ consist of three keywords: `try`, `throw` and `catch`:

- The `try` statement allows you to define a block of code to be tested for errors while it is being executed.
- The `throw` keyword throws an exception when a problem is detected, which lets us create a custom error.
- The `catch` statement allows you to define a block of code to be executed, if an error occurs in the try block.

SYNTAX

```
try {
  // Block of code to try
  throw exception; // Throw an exception when a problem arise
}
catch (type Arg) {
  // Block of code to handle errors
}
```

```
#include<iostream>
using namespace std;
int main(){
    try{
        int age;
        cout<<"enter your age:";
        cin>>age;
        if(age>=18){
            cout<<"yes Y'r Adult";
        }
        else{
            throw(age);
        }
        }
        catch(int a){
            cout<<"No Y'r Child";
        }

    return 0;
}
```

Handle Any Type of Exceptions (...)

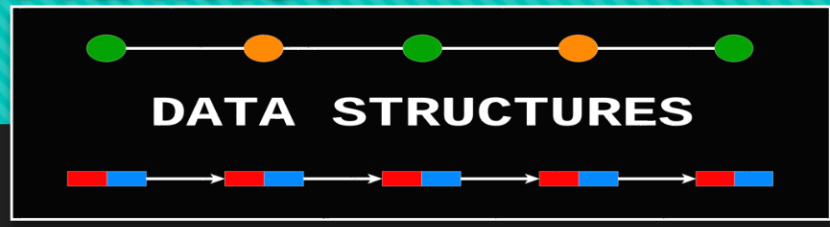
If you do not know the throw type used in the try block, you can use the "three dots" syntax (...) inside the catch block, which will handle any type of exception:

```
#include<iostream>
using namespace std;
int main(){
    try{
        int age;
        cout<<"enter your age:";
        cin>>age;
        if(age>=18){
            cout<<"yes Y'r Adult";
        }
        else{
            throw 505;
        }
    }
    catch(...){
        cout<<"No Y'r Child";
    }

    return 0;
}
```

```
#include<iostream>
using namespace std;
int main(){
    double a,b,c;
    cout<<"enter 2 value:";
    cin>>a>>b;
    try{
        if(b==0){
            throw 420;
        }
        else{
            c=a/b;
            cout<<"divide successfully:"<<c;
        }
    }
    catch(int x){
        cout<<"error found can't divide "<<x;
    }
return 0;
}
```

DATA STRUCTURE AND ALGORITHMS

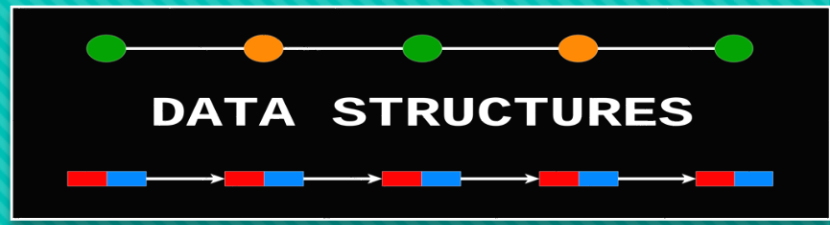
Why to Learn Data Structure and Algorithms?



As applications are getting complex and data rich, there are three common problems that applications face now-a-days.

- **Data Search** – Consider an inventory of 1 million(10^6) items of a store. If the application is to search an item, search an item in 1 million(10^6) items every time slowing down the search. As data grows, search will become slower.
- **Processor speed** – Processor speed although being very high, falls limited if the data grows to billion records.
- **Multiple requests** – As thousands of users can search data simultaneously on a web server, even the fast server fails while searching the data.

Algorithms Basics

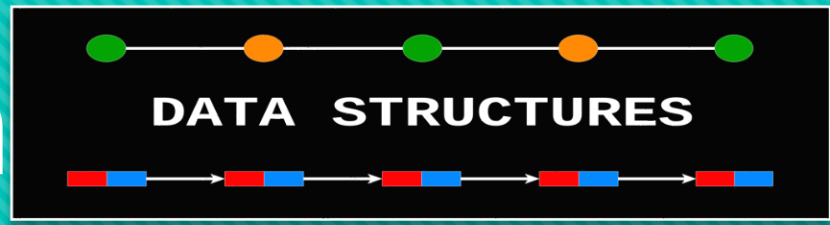


Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.

From the data structure point of view, following are some important categories of algorithms –

- **Search** – Algorithm to search an item in a data structure.
- **Sort** – Algorithm to sort items in a certain order.
- **Insert** – Algorithm to insert item in a data structure.
- **Update** – Algorithm to update an existing item in a data structure.
- **Delete** – Algorithm to delete an existing item from a data structure.

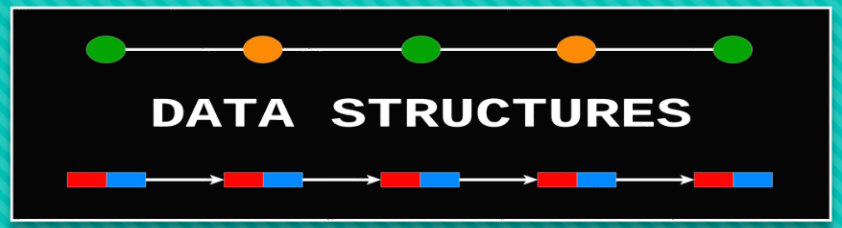
Characteristics of an Algorithm



Not all procedures can be called an algorithm. An algorithm should have the following characteristics –

- **Unambiguous** – Algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning.
- **Input** – An algorithm should have 0 or more well-defined inputs.
- **Output** – An algorithm should have 1 or more well-defined outputs, and should match the desired output.
- **Finiteness** – Algorithms must terminate after a finite number of steps.
- **Feasibility** – Should be feasible with the available resources.
- **Independent** – An algorithm should have step-by-step directions, which should be independent of any programming code.

How to Write an Algorithm?



There are no well-defined standards for writing algorithms. Rather, it is problem and resource dependent. Algorithms are never written to support a particular programming code.

As we know that all programming languages share basic code constructs like loops (do, for, while), flow-control (if-else), etc. These common constructs can be used to write an algorithm.

We write algorithms in a step-by-step manner, but it is not always the case. Algorithm writing is a process and is executed after the problem domain is well-defined. That is, we should know the problem domain, for which we are designing a solution

Making Algorithm

DATA STRUCTURES

PROBLEM – Design an algorithm to add two numbers and display the result.

Step 1 – START

Step 2 – declare three integers **a**, **b** & **c**

Step 3 – define values of **a** & **b**

Step 4 – add values of **a** & **b**

Step 5 – store output of step 4 to **c**

Step 6 – print **c**

Step 7 – STOP

Step 1 – START ADD

Step 2 – get values of **a** & **b**

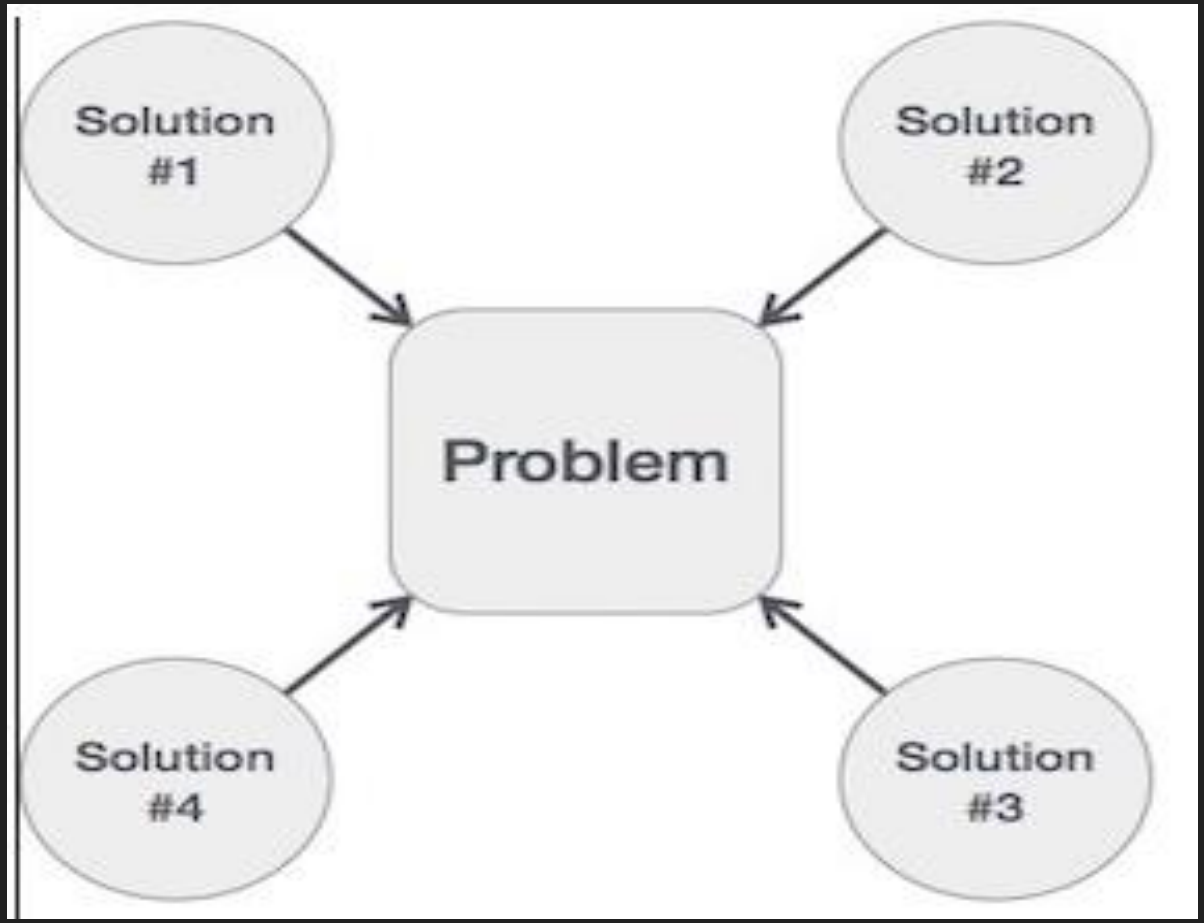
Step 3 – c ← a + b

Step 4 – display c

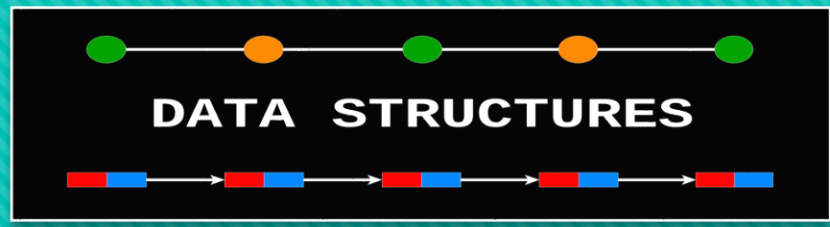
Step 5 – STOP

Writing step numbers, is optional.

We design an algorithm to get a solution of a given problem. A problem can be solved in more than one ways.

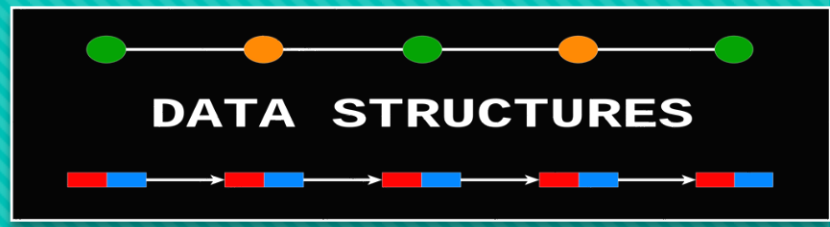


Algorithm Analysis



- Efficiency of an algorithm can be analyzed at two different stages, before implementation and after implementation. They are the following −
- ***A Priori* Analysis** − This is a theoretical analysis of an algorithm. Efficiency of an algorithm is measured by assuming that all other factors, for example, processor speed, are constant and have no effect on the implementation.
- ***A Posterior* Analysis** − This is an empirical analysis of an algorithm. The selected algorithm is implemented using programming language. This is then executed on target computer machine. In this analysis, actual statistics like running time and space required, are collected.

Algorithm Complexity

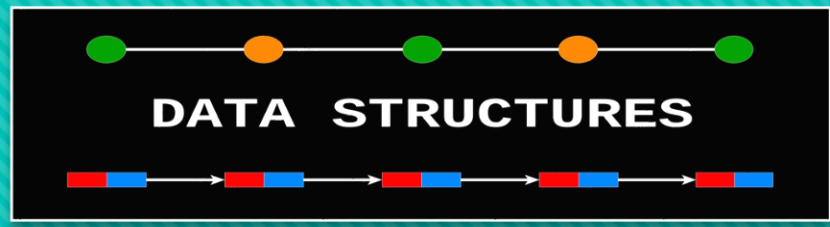


Suppose **X** is an algorithm and **n** is the size of input data, the time and space used by the algorithm X are the two main factors, which decide the efficiency of X.

- **Time Factor** – Time is measured by counting the number of key operations such as comparisons in the sorting algorithm.
- **Space Factor** – Space is measured by counting the maximum memory space required by the algorithm.

The complexity of an algorithm **f(n)** gives the running time and/or the storage space required by the algorithm in terms of **n** as the size of input data

Data Definition



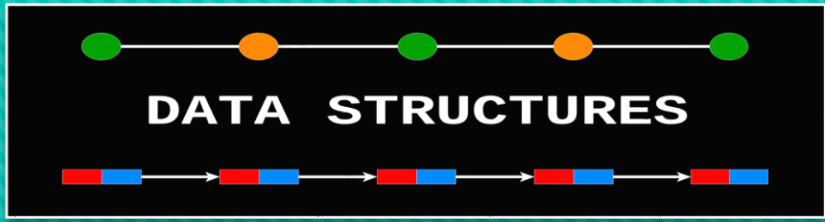
Data Definition defines a particular data with the following characteristics.

- ✓ Atomic – Definition should define a single concept.
- ✓ Traceable – Definition should be able to be mapped to some data element.
- ✓ Accurate – Definition should be unambiguous.
- ✓ Clear and Concise – Definition should be understandable

Data Object

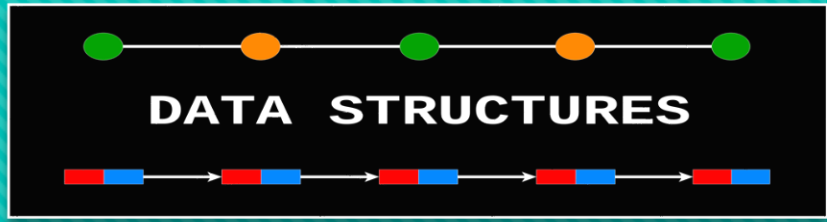
Data Object represents an object having a data.

Data Type



- Data type is a way to classify various types of data such as integer, string, etc. which determines the values that can be used with the corresponding type of data, the type of operations that can be performed on the corresponding type of data. There are two data types –
- Built-in Data Type
- Derived Data Type

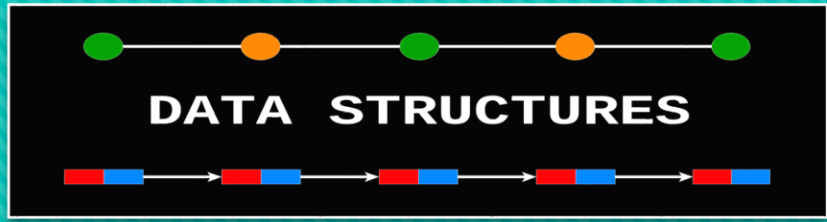
Built-in Data Type



Those data types for which a language has built-in support are known as Built-in Data types. For example, most of the languages provide the following built-in data types.

- Integers
- Boolean (true, false)
- Floating (Decimal numbers)
- Character and Strings

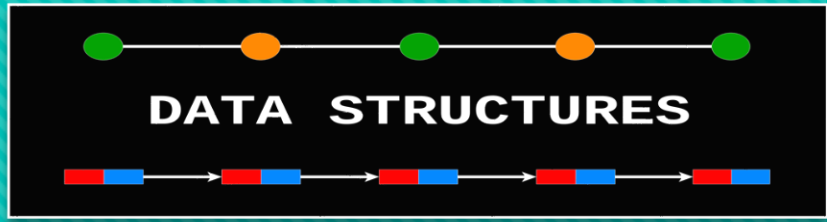
Derived Data Type



Those data types which are implementation independent as they can be implemented in one or the other way are known as derived data types. These data types are normally built by the combination of primary or built-in data types and associated operations on them. For example –

- List
- Array
- Stack
- Queue

Basic Operations

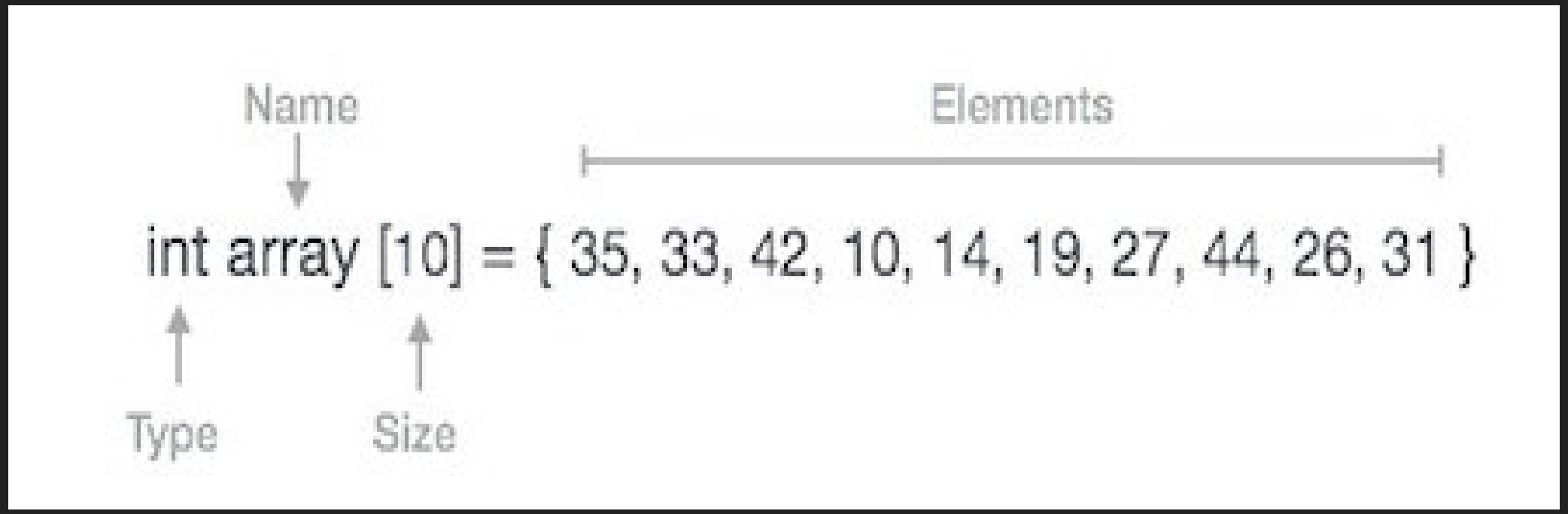


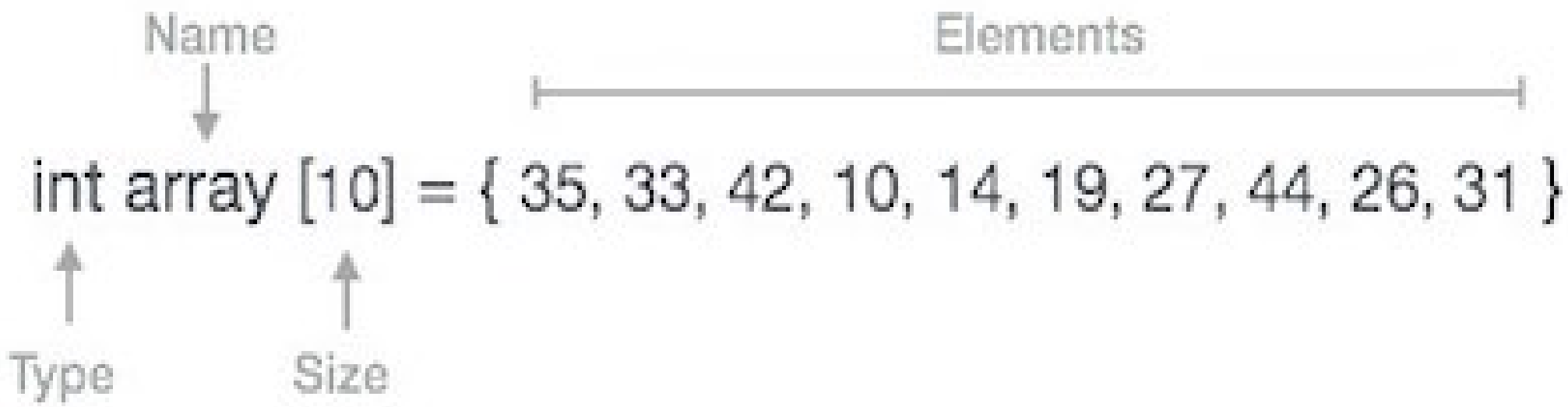
The data in the data structures are processed by certain operations. The particular data structure chosen largely depends on the frequency of the operation that needs to be performed on the data structure.

- ❑Traversing
- ❑Searching
- ❑Insertion
- ❑Deletion
- ❑Sorting
- ❑Merging

Array Representation

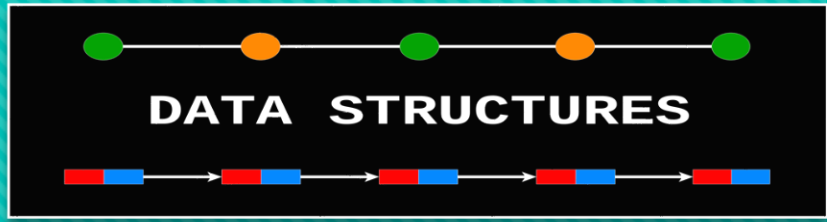
Arrays can be declared in various ways in different languages. For illustration, let's take C array declaration.




Name
Elements
int array [10] = { 35, 33, 42, 10, 14, 19, 27, 44, 26, 31 }
Type
Size

elements
35
33
42
10
14
19
27
44
26
31
index
0
1
2
3
4
5
6
7
8
9
Size :10

Basic Operations



Following are the basic operations supported by an array.

- Traverse – print all the array elements one by one.
- Insertion – Adds an element at the given index.
- Deletion – Deletes an element at the given index.
- Search – Searches an element using the given index or by the value.
- Update – Updates an element at the given index.

TRAVERSE

```
#include<iostream>
using namespace std;
int main(){
    int size,arr[size],i,nv;
    cout<<"enter size of array:";
    cin>>size;
    cout<<"Enter total number of value which you entered:";
    cin>>nv;
    if(nv>size){
        cout<<"can't insert:";
    }
    else{
        cout<<"Enter Element:"<<endl;
        for(i=0;i<=nv-1;i++){
            cin>>arr[i];
        }
    }
    cout<<"output is here..!"<<endl;
    for(i=0;i<=nv-1;i++){
            cout<<arr[i]<<"\t";
        }
}
```

DATA STRUCTURES

```cpp
#include<iostream>
using namespace std;
int main(){
    int arr[30],i,n,ins,pos;
    cout<<"enter total value you enter in array:";
    cin>>n;
    cout<<"enter your elements:";
    for(i=0;i<=n-1;i++){
        cin>>arr[i];
    }
    cout<<"all elements:"<<endl;
    for(i=0;i<=n-1;i++){
        cout<<arr[i]<<endl;
    }
```

```cpp
    cout<<"enter new element:";
    cin>>ins;
    cout<<"enter position for new element:";
    cin>>pos;
    n++;

    for (i = n-1; i >= pos; i--)
    {
        arr[i] = arr[i - 1];
    }
    arr[pos-1]=ins;
    cout<<"all elements:"<<endl;
    for(i=0;i<=n-1;i++){
        cout<<arr[i]<<endl;
    }

    return 0;
}
```

INSERTION

DATA STRUCTURES

DELETION

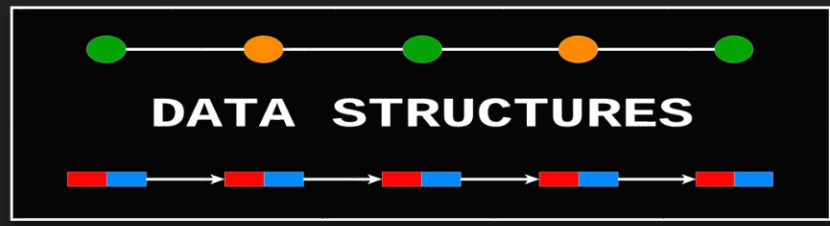
```
#include<iostream>
using namespace std;
int main()
{
    int array[100], position, c, n;
    cout<<"Enter number of elements in array\n";
    cin>>n;
    cout<<"Enter "<<n<<"  elements\n";
     for (c = 0; c < n; c++)
     {
         cin>>array[c];
     }
    cout<<"Enter the location where you wish to delete element\n";
    cin>>position;
```

```
    if (position >= n+1){

        cout<<"Deletion not possible.\n";
    }
    else
    {
        for (c = position - 1; c < n - 1; c++){
          array[c] = array[c+1];
          }
          cout<<"Resultant array:\n";
           for (c = 0; c < n - 1; c++){
           cout<<array[c]<<endl;}
    }
    return 0;
}
```

DATA STRUCTURES

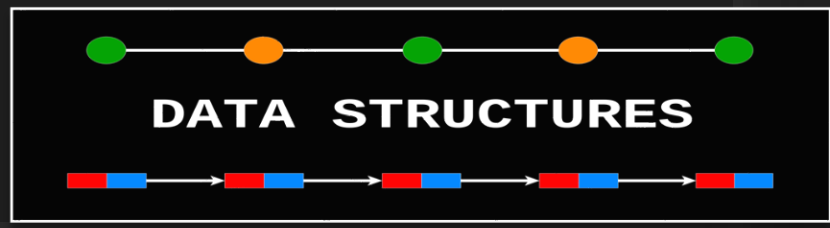
SEARCHING

```
#include<iostream>
using namespace std;
int main(){
    int size,a[size],i,srh,cnt=0;
    cout<<"enter size of array:";
    cin>>size;
    cout<<"enter all element of array:\n";
    for(i=0;i<size;i++){
        cout<<"a["<<i<<"]=";
        cin>>a[i];
    }
    cout<<"All element are here..\n";
    for(i=0;i<size;i++){
        cout<<"a["<<i<<"]="<<a[i]<<"\t";
    }
    cout<<"\nenter element which your want to search:";
    cin>>srh;
    for(i=0;i<size;i++){
        if(a[i]==srh){
        cnt++;
        }
    }
    if(cnt>0){
        cout<<"element is found..!";
    }
    else{
        cout<<"element is not found..!";
    }
    return 0;
}
```

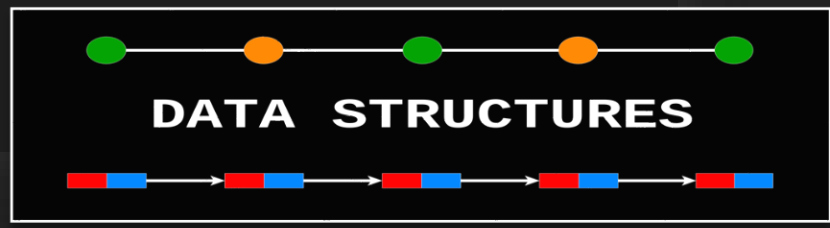


UPDATION

```
#include<iostream>
using namespace std;
int main(){
    int size,arr[size],i,pos,rep;
    cout<<"enter size of the array:";
    cin>>size;
    cout<<"enter all element in the array:"<<endl;
    for(i=0;i<size;i++){
        cout<<"arr["<<i<<"]=";
        cin>>arr[i];
    }
    cout<<"\n\n\n<-----Here all element in the array:----->"<<endl;
        for(i=0;i<size;i++){
        cout<<"arr["<<i<<"]="<<arr[i]<<"\t";
        }
```

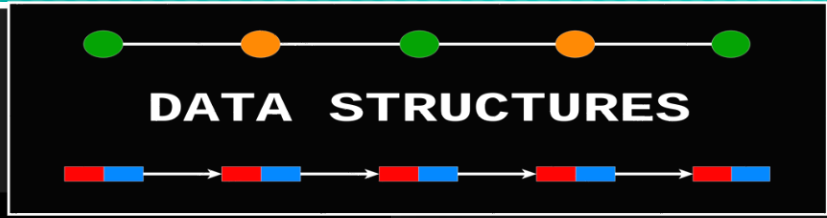


```
    cout<<"\nenter new value which you want:";
    cin>>rep;
    cout<<"\nenter the position where you want to put the value:";
    cin>>pos;
    if(pos>=size+1){
        cout<<"not possible:";
    }
    else{

    for(i=size-1;i>=pos-1;i--){
        arr[pos-1]=rep;
    }
    cout<<"\n\n\n<-----Here all element in the array after Replace:----->"<<endl;
        for(i=0;i<size;i++){
        cout<<"arr["<<i<<"]="<<arr[i]<<"\t";
        }
}
    return 0;
}
```

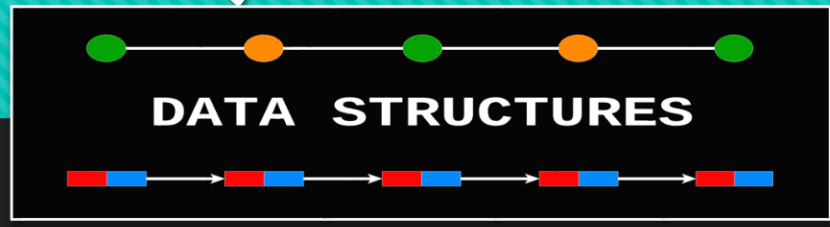


Dynamic Memory(using NEW KEYWORD)

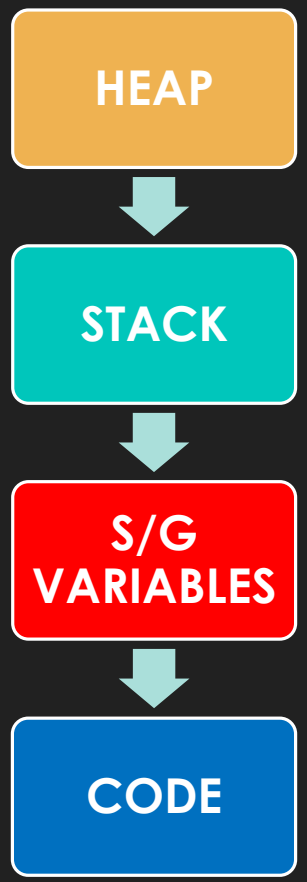
```
#include<iostream>
using namespace std;
int main()
{
    int size;
    cout<<"Enter the size of the array:";
    cin>>size;
    int *ptr=new int[size];
    cout<<"Enter all elements:";
    for(int i=0;i<=size-1;i++){
        cout<<"arr["<<i<<"]=";
        cin>>ptr[i];
    }
    cout<<"All elements:";
    for(int i=0;i<=size-1;i++){
        cout<<"arr["<<i<<"]"<<ptr[i]<<"\t";
    }
    return 0;
}
```



DMA(DYNAMIC MEMORY ALLOCATION)

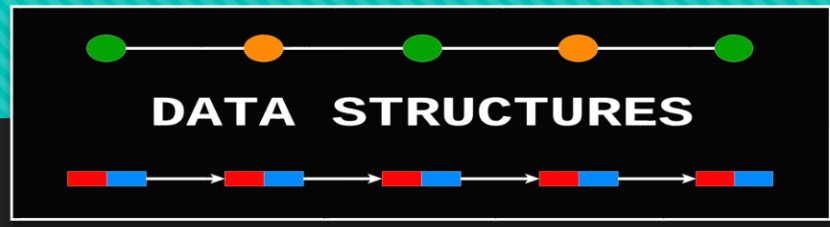


- Dynamic memory allocation is a way in which the size of data structure can be changed during the runtime.
- Memory assigned to a program in a typical architecture can be broken down into four segment
  a) Code
  b) Static/global variables
  c) Stack
  d) heap



There are 4 function In DMA
1) Malloc() 2) Calloc() 3)Realloc() 4)Free()

Malloc() Function



- Malloc() Stands for Memory Allocation
- It reserve a block of memory with the given amount of bytes.
- The return value is a void pointer to the allocated space.
- Therefore the void pointer needs to be casted to the appropriate type as per the requirements
- However, if the space is insufficient, allocation of memory fails and it returns a NULL pointer
- All the value at allocated memory are initialized to garbage values

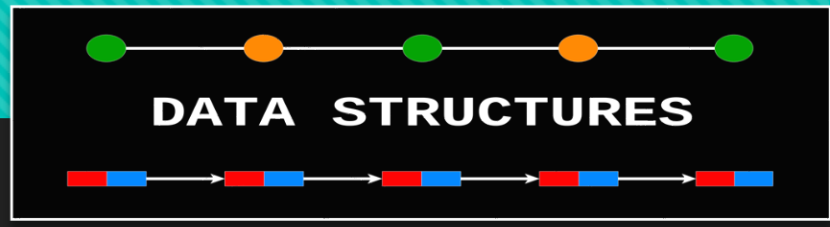
syntax:

```
PTR=(PTR_TYPE) MALLOC(SIZE_IN_BYTES)
```

DATA STRUCTURES

```
#include<stdlib.h>
using namespace std;
int main(){
    int* ptr;
    int size;
    cout<<"enter size of array:";
    cin>>size;
    //use of malloc function(DMA)
    ptr=(int*) malloc(size*sizeof(int));
    cout<<"enter all element:";
    for(int i=0;i<size;i++)
    {
    cout<<"ptr["<<i<<"]=";
    cin>>ptr[i];
    }
    for(int i=0;i<size;i++)
    {
    cout<<"ptr["<<i<<"]="<<ptr[i]<<"\t";
    }   return 0;
}
```

Calloc() Function



- Calloc() Stands for Contiguous Allocation
- It reserve N block of memory with the given amount of bytes.
- The return value is a void pointer to the allocated space.
- Therefore the void pointer needs to be casted to the appropriate type as per the requirements
- However, if the space is insufficient, allocation of memory fails and it returns a NULL pointer
- All the value at allocated memory are initialized to 0.

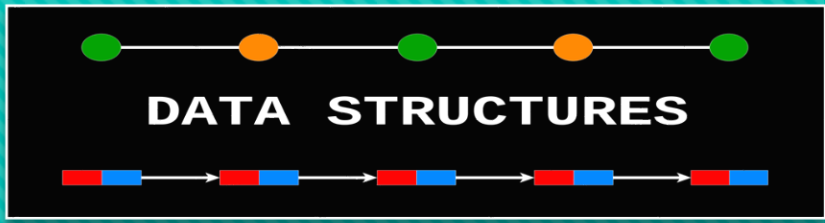
syntax:

```
PTR=(PTR_TYPE) CALLOC(N,SIZE_IN_BYTES)
```

```
#include<stdlib.h>
using namespace std;
int main(){
    int* ptr;
    int size;
    cout<<"enter size of array:";
    cin>>size;
    //use of calloc function(DMA)
    ptr=(int*) calloc(size,sizeof(int));
    cout<<"enter all element:";
    for(int i=0;i<size;i++)
    {
    cout<<"ptr["<<i<<"]=";
    cin>>ptr[i];
    }
    for(int i=0;i<size;i++)
    {
    cout<<"ptr["<<i<<"]="<<ptr[i]<<"\t";
    }   return 0;
}
```

DATA STRUCTURES

Realloc Function



- Realloc() stands for Reallocation.
- If the dynamically allocated memory is insufficient we can change the size of previously allocated memory using realloc() function

Syntax:

```
ptr=(ptr_type*) realloc(ptr,new_size_in_bytes)
```

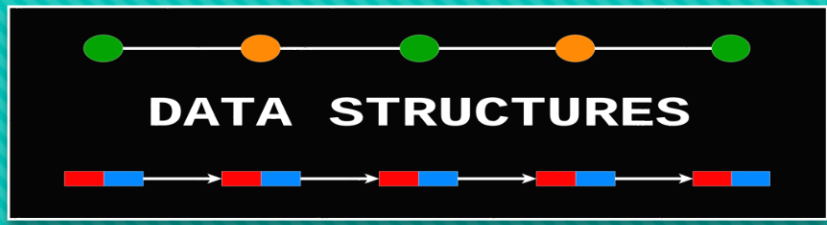
```
#include<iostream>
#include<stdlib.h>
using namespace std;
int main(){
    int size;
    int *ptr;
    cout<<"Enter size of the array:\n";
    cin>>size;
    ptr=(int*)calloc(size,sizeof(int));
    for(int i=0;i<size;i++){
        cout<<"Enter Element in index["<<i<<"]=";
        cin>>ptr[i];
    }
    cout<<"\n\nPrinting all Elements:\n";
    for(int i=0;i<size;i++){
        cout<<"Element in index["<<i<<"]="<<ptr[i]<<"\n";
    }
```

DATA STRUCTURES

```
    //use of realloc
    cout<<"Enter New size of the array:\n";
    cin>>size;
    ptr=(int *)realloc(ptr,size*sizeof(int));
    for(int i=0;i<size;i++){
        cout<<"Enter New Element in index["<<i<<"]=";
        cin>>ptr[i];
    }
    cout<<"\n\nPrinting New all Elements:\n";
    for(int i=0;i<size;i++){
        cout<<"Element in index["<<i<<"]="<<ptr[i]<<"\n";
    }

    return 0;
}
```

DATA STRUCTURES

Free() Function

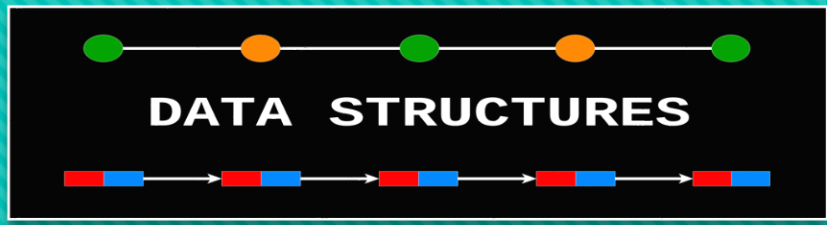


- Free() is used to free the allocated memory.
- If the dynamically allocated memory is not required anymore,we can free it using free() function

Syntax:

```
Free(ptr)
```

Linked List



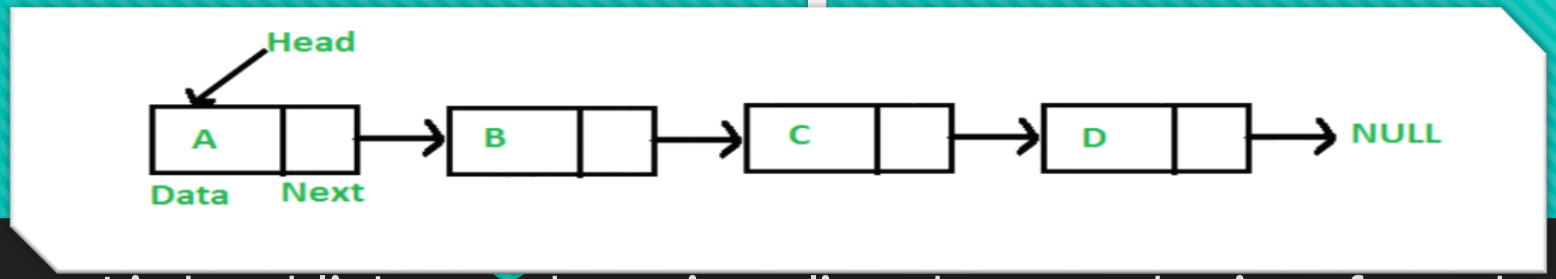
Like arrays, Linked List is a linear data structure. Unlike arrays, linked list elements are not stored at a contiguous location; the elements are linked using pointers.

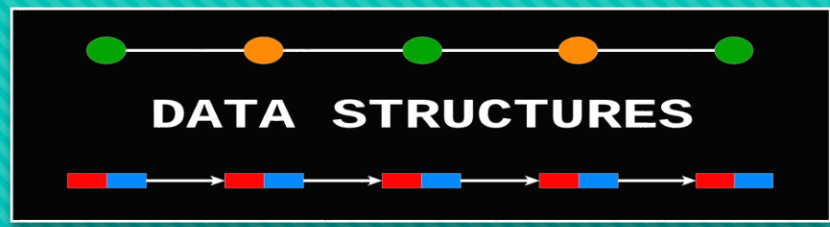
A linked list is a sequence of data structures, which are connected together via links.

Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of Linked List.

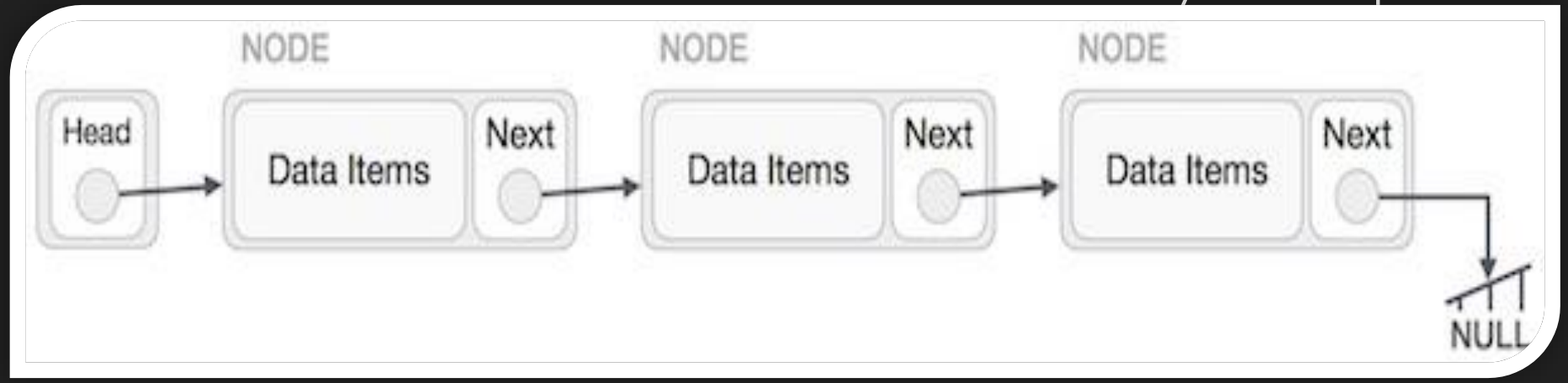
- ✓ Link – Each link of a linked list can store a data called an element.
- ✓ Next – Each link of a linked list contains a link to the next link called Next.
- ✓ LinkedList – A Linked List contains the connection link to the first link called First.

Linked List Representation





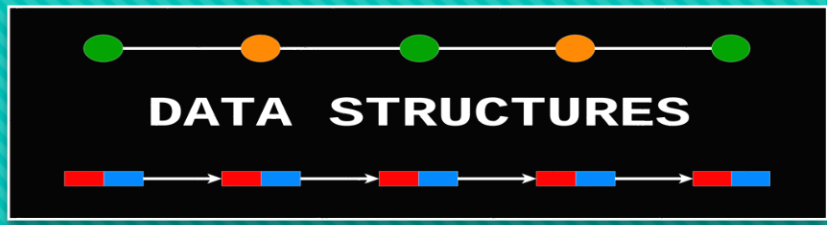
Linked list can be visualized as a chain of nodes, where every node points to the next node.



As per the above illustration, following are the important points to be considered.

- Linked List contains a link element called first.
- Each link carries a data field(s) and a link field called next.
- Each link is linked with its next link using its next link.
- Last link carries a link as null to mark the end of the list.

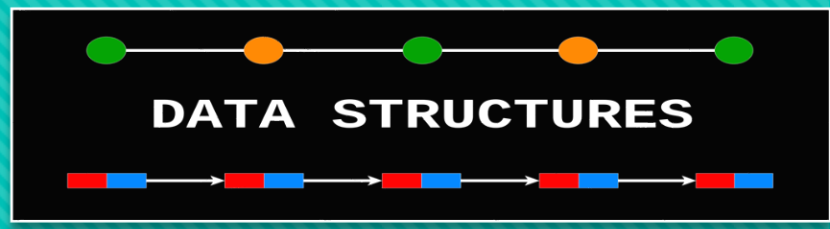
Types of Linked List



Following are the various types of linked list.

- **Simple Linked List** – Item navigation is forward only.
- **Doubly Linked List** – Items can be navigated forward and backward.
- **Circular Linked List** – Last item contains link of the first element as next and the first element has a link to the last element as previous.

Why Linked List?



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

1) **The size of the arrays is fixed:**

2) **Inserting a new element in an array of elements is expensive because the room has to be created for the new elements and to create room existing elements have to be shifted.**

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

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

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

Deletion is also expensive with arrays until unless some special techniques are used. For example, to delete 1010 in id[], everything after 1010 has to be moved.

DATA STRUCTURES

```cpp
// A simple CPP program to introduce
// a linked list
#include <bits/stdc++.h>
using namespace std;
 class Node {
public:
    int data;
    Node* next;
};
// Program to create a simple linked
// list with 3 nodes
int main()
{
    Node* head = NULL;
    Node* second = NULL;
    Node* third = NULL;

    // allocate 3 nodes in the heap
    head = new Node();
    second = new Node();
    third = new Node();
```

```cpp
head->data = 1; // assign data in first node
head->next = second; // Link first node with
// the second node

// assign data to second node
second->data = 2;

// Link second node with the third node
second->next = third;

third->data = 3; // assign data to third node
third->next = NULL;
return 0;
```

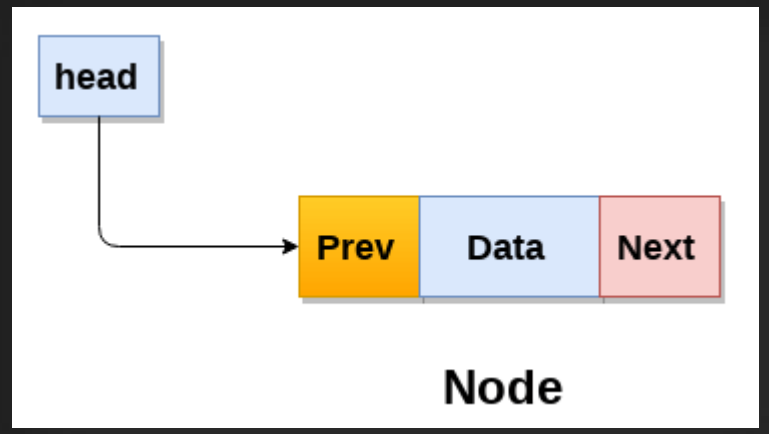
Simple Linked List Program

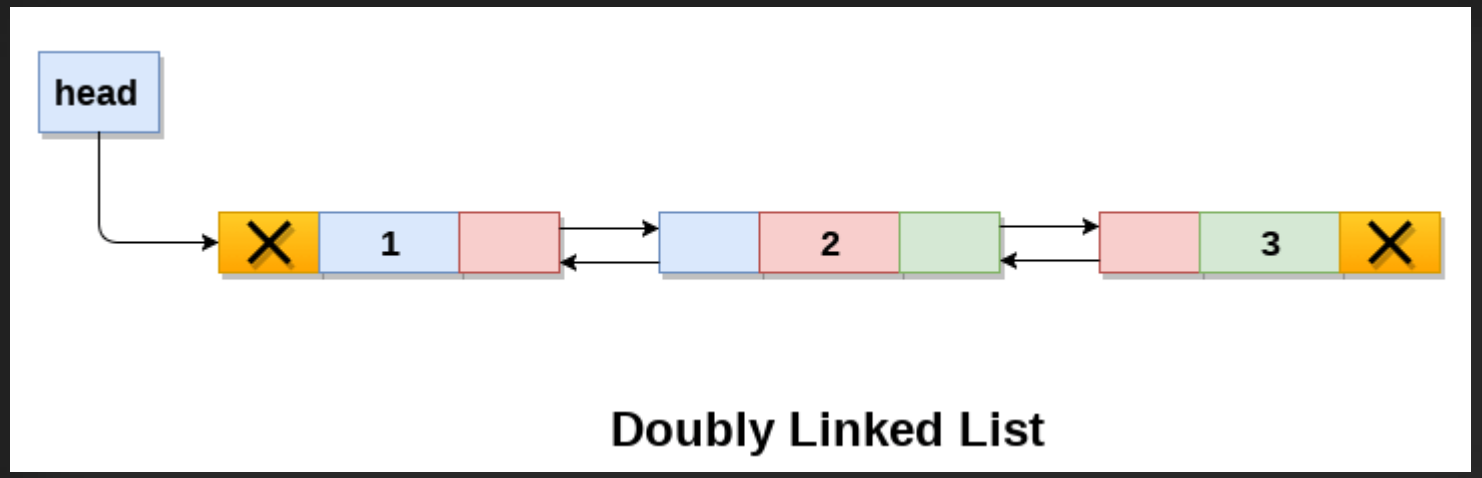
```
#include<bits/stdc++.h>
using namespace std;
class Node{
    public:
    int data;
    Node* next;
};
void print(Node* n){
    while(n!=NULL){
        cout<<n->data<<" ";
        n=n->next;
    }
```

```
int main(){
    Node* head=NULL;
    Node* second=NULL;
    Node* third=NULL;

    head=new Node();
    second=new Node();
    third=new Node();

    head->data=1;
    head->next=second;

    second->data=12;
    second->next=third;

    third->data=123;
    third->next=NULL;
    print(head);
    return 0;
```

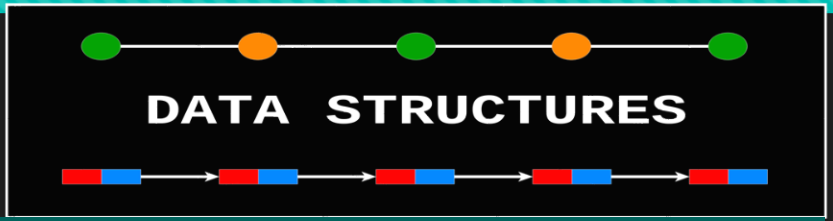
Doubly linked list

Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence. Therefore, in a doubly linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer) , pointer to the previous node (previous pointer).





Algorithm for Insertion In Doubly Link List



Step 1: IF ptr = NULL

Write OVERFLOW Go to Step 9

[END OF IF]

Step 2: SET NEW_NODE = ptr

Step 3: SET ptr = ptr -> NEXT

Step 4: SET NEW_NODE -> DATA = VAL

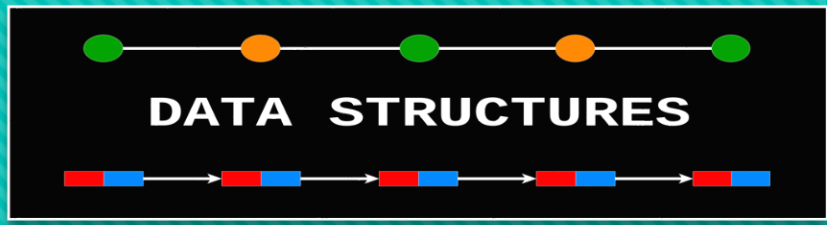
Step 5: SET NEW_NODE -> PREV = NULL

Step 6: SET NEW_NODE -> NEXT = START
Step 7: SET head -> PREV = NEW_NODE
Step 8: SET head = NEW_NODE
Step 9: EXIT

```c
#include<stdio.h>
#include<stdlib.h>
void insertbeginning(int);
struct node
{
    int data;
    struct node *next;
    struct node *prev;
};
struct node *head;
int main ()
{
    int choice,item;
    do
    {
        printf("\nEnter the item which you want to insert?\n");
        scanf("%d",&item);
        insertbeginning(item);
        printf("\nPress 0 to insert more ?\n");
        scanf("%d",&choice);
    }while(choice == 0);
}
```

```c
void insertbeginning(int item)
{

    struct node *ptr = (struct node *)malloc(sizeof(struct node));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW");
    }
    else
    {
     if(head==NULL)
    {
        ptr->next = NULL;
        ptr->prev=NULL;
        ptr->data=item;
        head=ptr;
    }
    else
    {
        ptr->data=item;
        ptr->prev=NULL;
        ptr->next = head;
        head->prev=ptr;
        head=ptr;
    }
 }
 }
```

Deletion AT Beginning




```c
#include<stdio.h>
#include<stdlib.h>
void insert(int);
void dlt();
struct node{
    int data;
    struct node *prev;
    struct node *next;
};
struct node *head;
int main(){
    int ch,item;
```

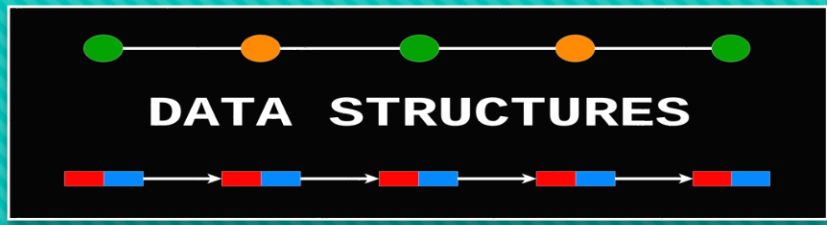
```c
do{
    printf("\n1. Append\n");
    printf("2. Delete\n");
    printf("3. Exit\n");
    printf("Enter Your Choice:");
    scanf("%d",&ch);
    switch(ch){
```

```
        switch(ch){
            case 1:
                printf("Enter Your Value To insert in the Doubly linklist:");
                scanf("%d",&item);
                insert(item);
                break;
            case 2:
                dlt();
                break;
            case 3:
                exit(0);
                break;
            dafault:
                printf("\nPLease Check Your Input\n");
        }

    }while(ch!=3);
    return 0;
}
```

```c
void insert(int item){
    struct node *ptr=(struct node *)malloc(sizeof(struct node));
    if(ptr==NULL){
        printf("\nOVERFLOW\n");
    }
    else{
        if(head==NULL){
            ptr->next=NULL;
            ptr->prev=NULL;
            ptr->data=item;
            head=ptr;
        }
        else{
            ptr->data=item;
            ptr->prev=NULL;
            ptr->next=head;
            head->prev=ptr;
            head=ptr;
        }
        printf("\nNode Inserted\n");

    }
}
```

```
void dlt(){
struct node *ptr;
if(head==NULL){
    printf("\nUNderFlow\n");
}
else if(head->next==NULL){
    head=NULL;
    free(head);
    printf("\nNode Deleted\n");
}
else{
    ptr=head;
    head=head->next;
    head->prev=NULL;
    free(ptr);
    printf("\nNode Deleted\n");
}
}
```

Circular Linked List



Circular Linked List is a type of linked list where list are linked in such a manner to form a circle i.e. Last Node points the First Node. In Circular linked list there exists no nodes which have null at address space. It is of two types:

1) Singly Linked List: It provides the link to the first node by replacing the NULL with the address of first node. Singly Linked List helps in accessing the previous node with the help of link creation in circular order.

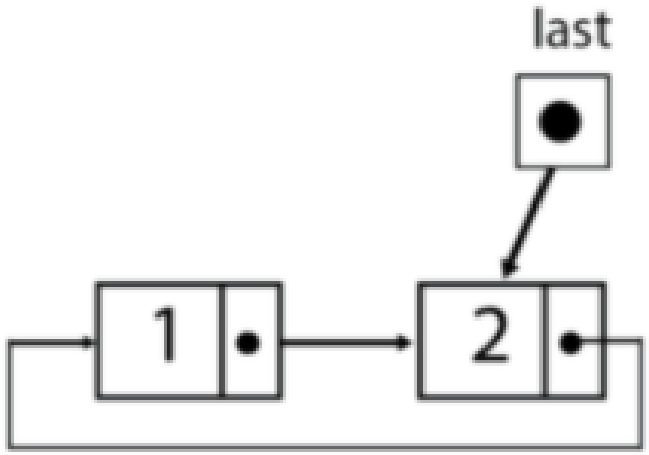
2) Doubly Linked List: Doubly Linked List (DLL) consist of an extra pointer known as previous pointer.

Advantages of Singly and Doubly Linked List

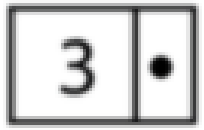
- It helps in reaching the previous node information even if traverse ahead from a particular position.
- DLL can traverse in both forward and backward direction.
- DLL is faster in getting previous node information due to previous pointer.
- Singly Linked List requires less space than doubly linked list as it does not have previous pointer.

Addition at beginning

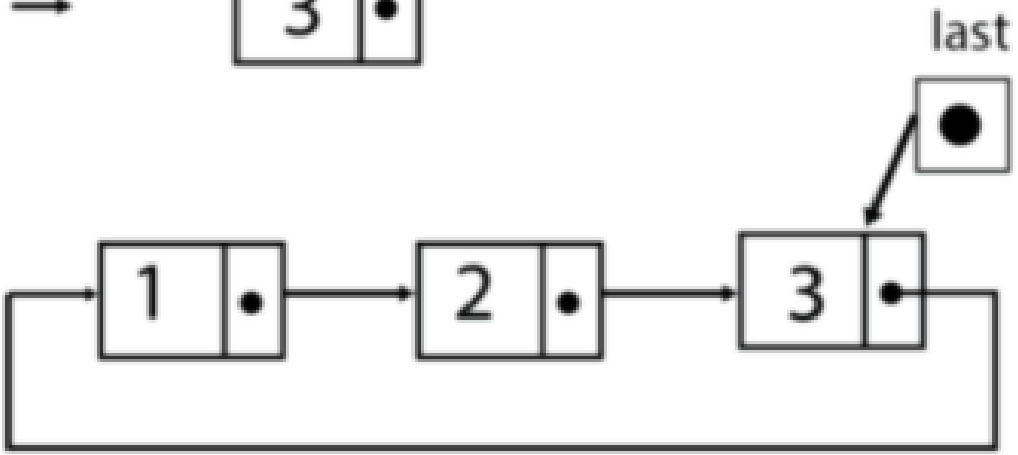
Circular Linked list →



Assed Node →



Process →

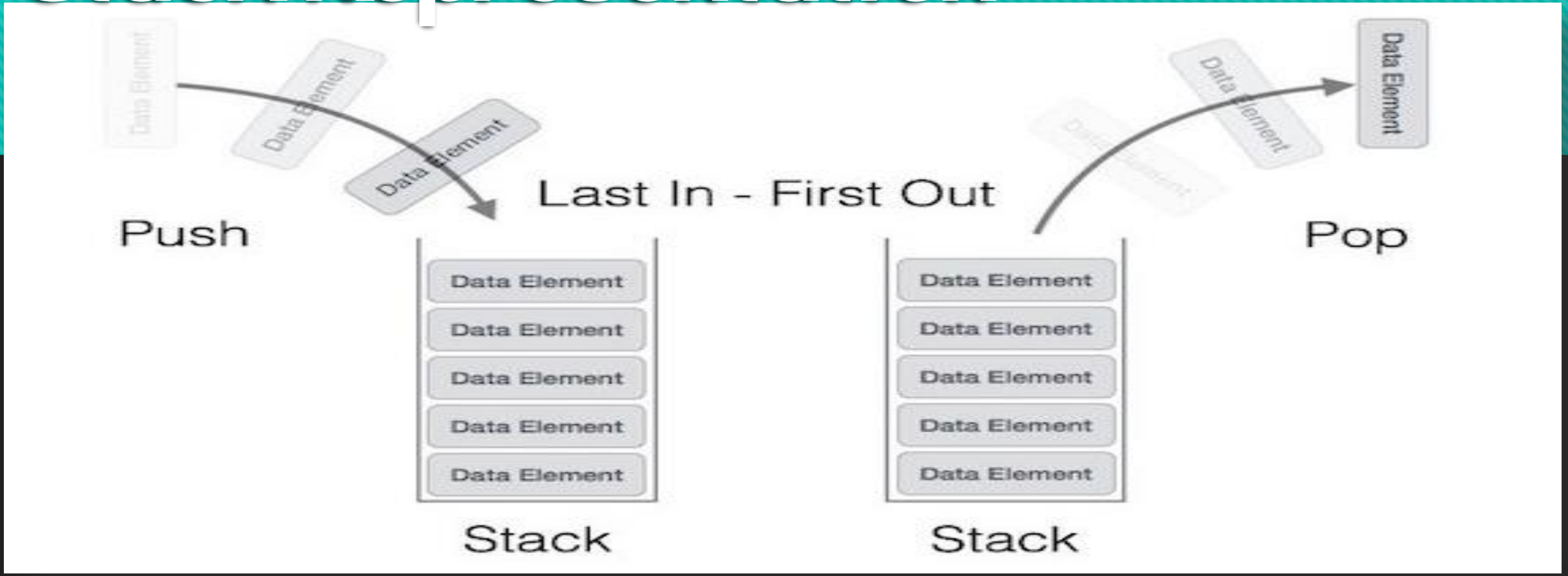


STACK

A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.

A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from the top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.
This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called PUSH operation and removal operation is called POP operation

Stack Representation



A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using arrays, which makes it a fixed size stack implementation

Basic Operations

Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations −

push() − Pushing (storing) an element on the stack.

pop() − Removing (accessing) an element from the stack.

When data is Pushed onto stack.

Expression Parsing

The way to write arithmetic expression is known as a notation. An arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of an expression. These notations are −

- Infix Notation
- Prefix (Polish) Notation
- Postfix (Reverse-Polish) Notation

Infix Notation

We write expression in **infix** notation, e.g. a - b + c, where operators are used **in**-between operands. It is easy for us humans to read, write, and speak in infix notation but the same does not go well with computing devices. An algorithm to process infix notation could be difficult and costly in terms of time and space consumption

Prefix Notation

In this notation, operator is prefixed to operands, i.e. operator is written ahead of operands. For example, +ab. This is equivalent to its infix notation a + b. Prefix notation is also known as Polish Notation.

Postfix Notation

This notation style is known as Reversed Polish Notation. In this notation style, the operator is post fixed to the operands i.e., the operator is written after the operands. For example, ab+. This is equivalent to its infix notation a + b.

Queue

Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

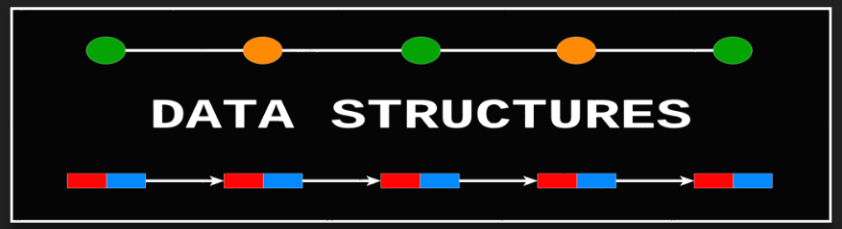
A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first.

Queue Representation

Basic Operations

- **enqueue()** – add (store) an item to the queue.
- **dequeue()** – remove (access) an item from the queue

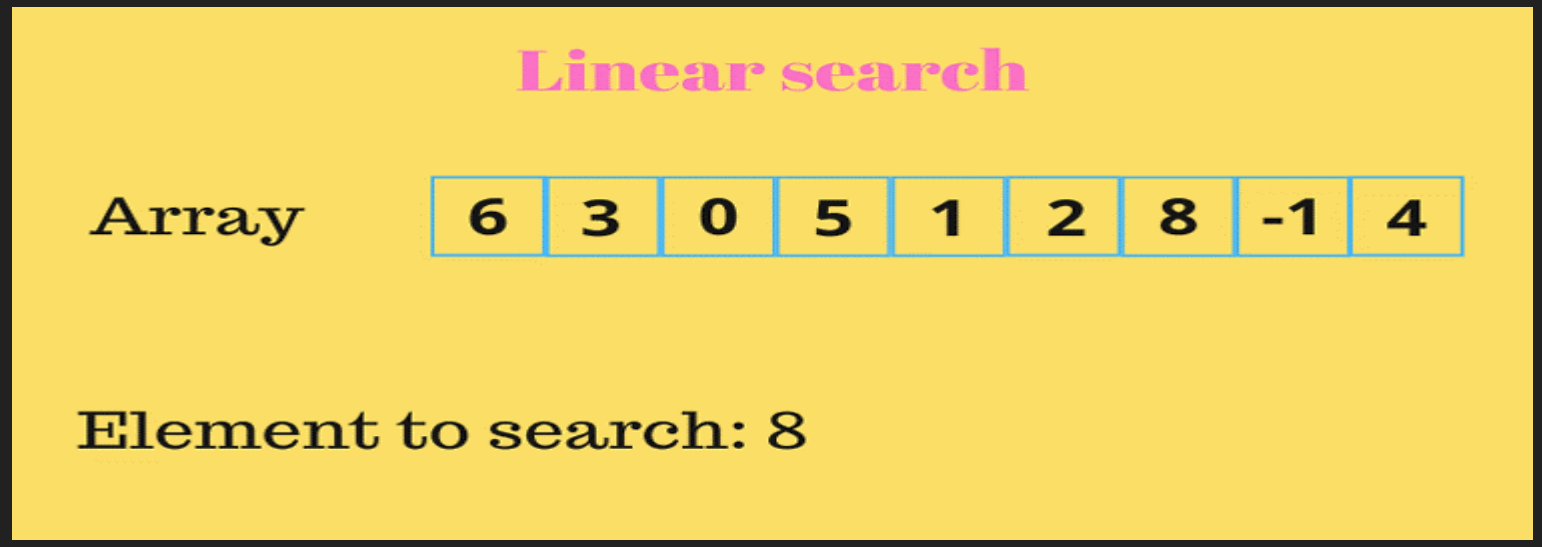
SEARCHING TECHNIQUES



Linear Search

DATA STRUCTURES

Linear search is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.



Luck

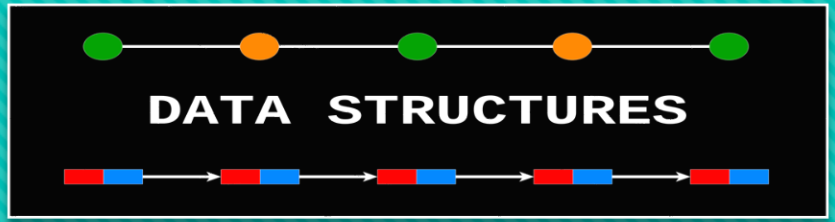
Average

Bad Luck

```cpp
#include<iostream>
using namespace std;
int main(){
    int a[]={1,3,4,65,76,87,76};
    for(int i=0;i<7;i++){
        cout<<"a["<<i<<"]="<<a[i]<<"\t";
    }
    int ser;
    cout<<endl<<"Enter value to search:"
    cin>>ser;
    int cnt=0;
```

```cpp
    for(int i=0;i<7;i++){
        if(a[i]==ser){
            cnt=cnt+1;
        }
        else{
            cnt;
        }
    }
    if(cnt>0){
        cout<<"found";
    }
    else{
        cout<<"not found";
    }
}
```

Algorithm



Linear Search (Array A, Value x)

Step 1: Set i to 1
Step 2: if i > n then go to step 7
Step 3: if A[i] = x then go to step 6
Step 4: Set i to i + 1
Step 5: Go to Step 2
Step 6: Print Element x Found at index i and go to step 8
Step 7: Print element not found Step
8: Exit

Binary Search

DATA STRUCTURES

Binary search is a fast search algorithm with run-time complexity. his search algorithm works on the principle of divide and conquer. For this algorithm to work properly, the data collection should be in the sorted form.

A[ ]=

20	29	1	7	5	23	19	13	8
0	1	2	3	4	5	6	7	8

Rule: Sort In Ascending Order

Find out:

Beg, End, Mid

Item to be searched = 23

Step 1 →

1	5	7	8	13	19	20	23	29
0	1	2	3	4	5	6	7	8

a [mid] = 13

13 < 23

beg = mid + 1 = 5

end = 8

mid = (beg + end)/2 = 13 / 2 = 6

Step 2

1	5	7	8	13	19	20	23	29
0	1	2	3	4	5	6	7	8

a [mid] = 20
20 < 23
beg = mid + 1 = 7
end = 8
mid = (beg + end)/2 = 15 / 2 = 7

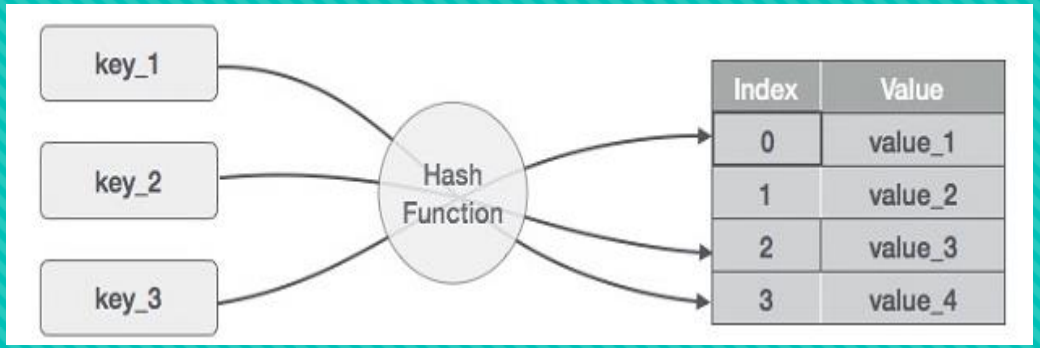
Step 3

1	5	7	8	13	19	20	23	29
0	1	2	3	4	5	6	7	8

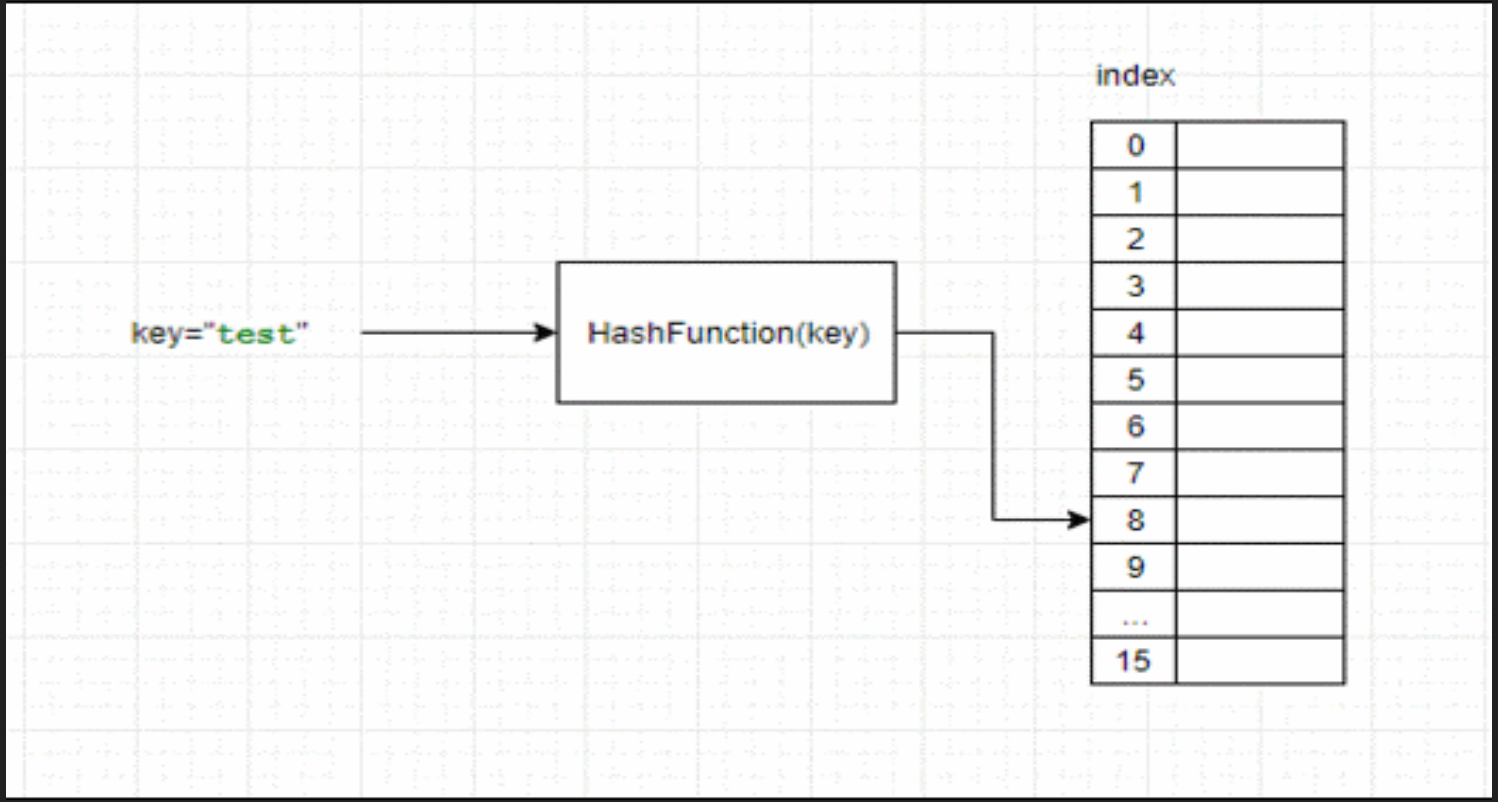
Return Index 7=23

a [mid] = 23
23 = 23
loc = mid

Hashing



Hashing is a technique to convert a range of key values into a range of indexes of an array. We're going to use modulo operator to get a range of key values. Consider an example of hash table of size 20, and the following items are to be stored. Item are in the (key, value) format.



A[ ]=

1	2	42	4	12	14	17	13	37

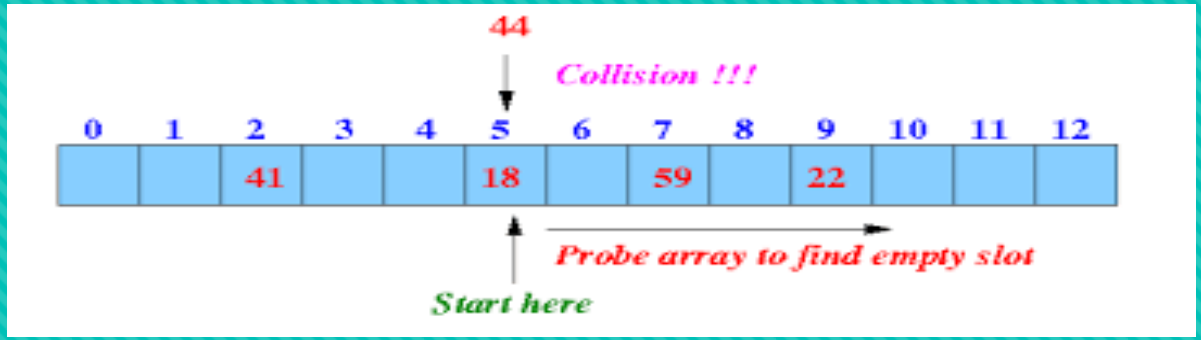
F(k)=k % size

Sr.No.	Key	Hash	Array Index
1	1	1 % 20 = 1	1
2	2	2 % 20 = 2	2
3	42	42 % 20 = 2	2
4	4	4 % 20 = 4	4
5	12	12 % 20 = 12	12
6	14	14 % 20 = 14	14
7	17	17 % 20 = 17	17
8	13	13 % 20 = 13	13
9	37	37 % 20 = 17	17

COLLISION (Same Index Problem)

COLLISION (Same Index Problem)

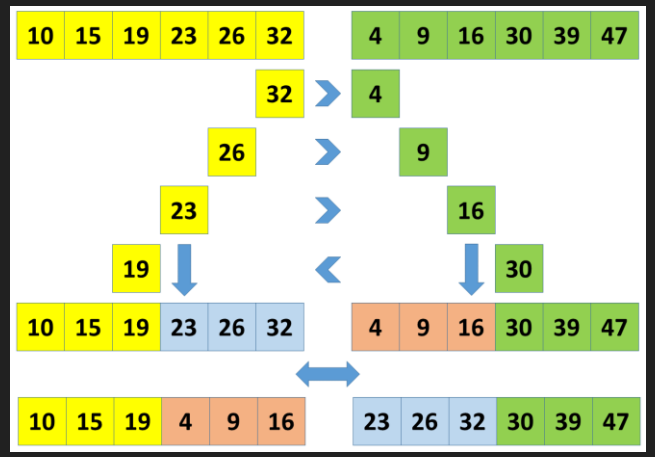
Linear Probing

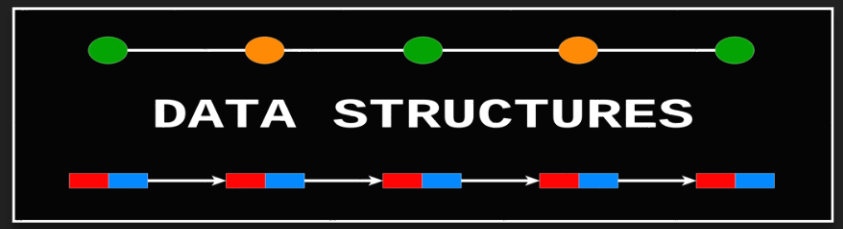


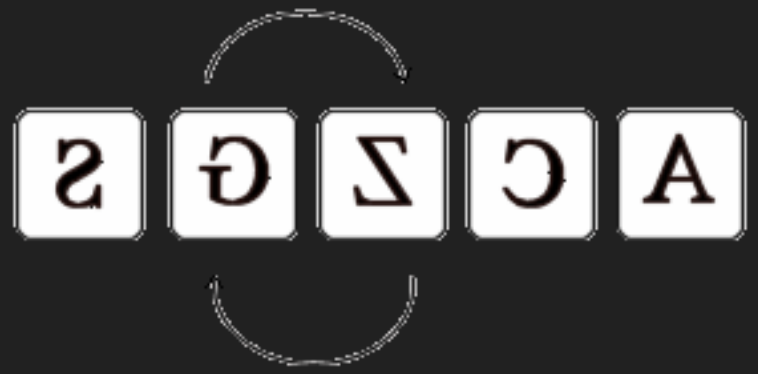
It may happen that the hashing technique is used to create an already used index of the array. In such a case, we can search the next empty location in the array by looking into the next cell until we find an empty cell. This technique is called linear probing.

Sr.No.	Key	Hash	Array Index	After Linear Probing, Array Index
1	1	1 % 20 = 1	1	1
2	2	2 % 20 = 2	2	2
3	42	42 % 20 = 2	2	3
4	4	4 % 20 = 4	4	4
5	12	12 % 20 = 12	12	12
6	14	14 % 20 = 14	14	14
7	17	17 % 20 = 17	17	17
8	13	13 % 20 = 13	13	13
9	37	37 % 20 = 17	17	18

SORTING TECHNIQUES







Bubble Sort

DATA STRUCTURES

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of O(n2) where n is the number of items.

6 5 3 1 8 7 2 4

Bubble sort

Array

6	3	0	5	1

How Bubble Sort Works?

We take an unsorted array for our example. Bubble sort takes $O(n^2)$ time so we're keeping it short and precise.

Bubble sort starts with very first two elements, comparing them to check which one is greater.

In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.

We find that 27 is smaller than 33 and these two values must be swapped.

The new array should look like this –

Next we compare 33 and 35. We find that both are in already sorted positions.

Then we move to the next two values, 35 and 10.

We know then that 10 is smaller 35. Hence they are not sorted.

We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this –

To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this –

Notice that after each iteration, at least one value moves at the end.

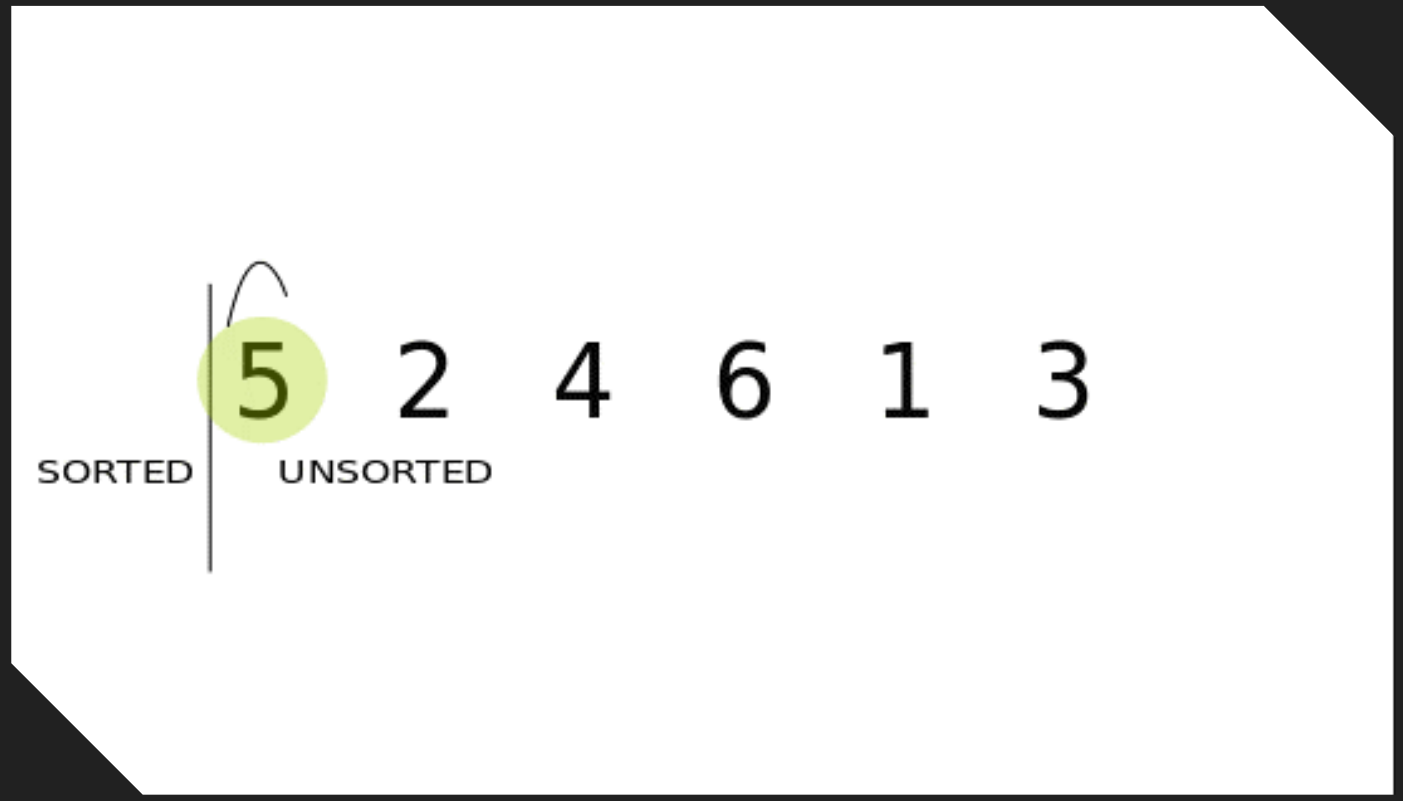
And when there's no swap required, bubble sorts learns that an array is completely sorted.

Now we should look into some practical aspects of bubble sort.

Insertion Sort

16	19	15	11	10

This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be 'insert'ed in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, insertion sort.



How Insertion Sort Works?

We take an unsorted array for our example.

14	33	27	10	35	19	42	44

Insertion sort compares the first two elements.

14	33	27	10	35	19	42	44

It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.

14	33	27	10	35	19	42	44

Insertion sort moves ahead and compares 33 with 27.

14	33	27	10	35	19	42	44

And finds that 33 is not in the correct position.

14	33	27	10	35	19	42	44

It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.

14	27	33	10	35	19	42	44

By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.

14	27	33	10	35	19	42	44

These values are not in a sorted order.

14	27	33	10	35	19	42	44

So we swap them.

14	27	10	33	35	19	42	44

However, swapping makes 27 and 10 unsorted.

14	27	10	33	35	19	42	44

Hence, we swap them too.

14	10	27	33	35	19	42	44

Again we find 14 and 10 in an unsorted order.

We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.

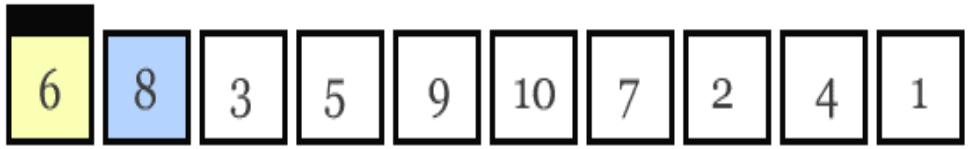
This process goes on until all the unsorted values are covered in a sorted sub-list. Now we shall see some programming aspects of insertion sort.

Selection Sort

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

This algorithm is not suitable for large data sets as its average and worst case complexities are of O(n2), where n is the number of items.



Yellow is smallest number found

Blue is current item

Green is sorted list

How Selection Sort Works?

Consider the following depicted array as an example.

14	33	27	10	35	19	42	44

For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.

14	33	27	10	35	19	42	44

So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.

10	33	27	14	35	19	42	44

For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.

10	33	27	14	35	19	42	44

We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.

10	33	27	14	35	19	42	44

After two iterations, two least values are positioned at the beginning in a sorted manner.

10	14	27	33	35	19	42	44

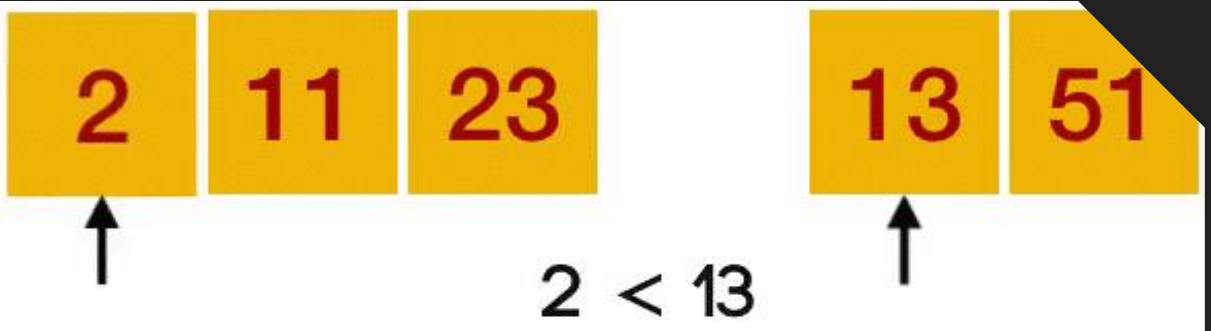
The same process is applied to the rest of the items in the array.

Following is a pictorial depiction of the entire sorting process –

Merge Sort

Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being O(n log n), it is one of the most respected algorithms.
Merge sort first divides the array into equal halves and then combines them in a sorted manner

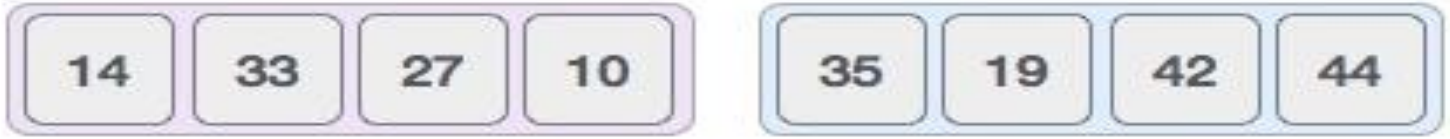
6 5 3 1 8 7 2 4



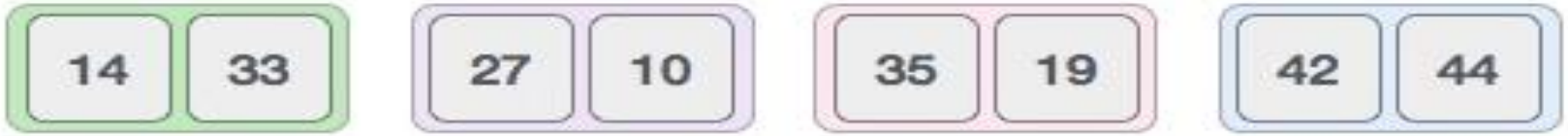
How Merge Sort Works?

To understand merge sort, we take an unsorted array as the following –

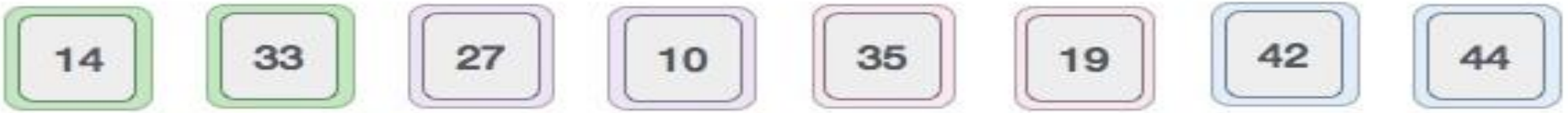
We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.



This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.

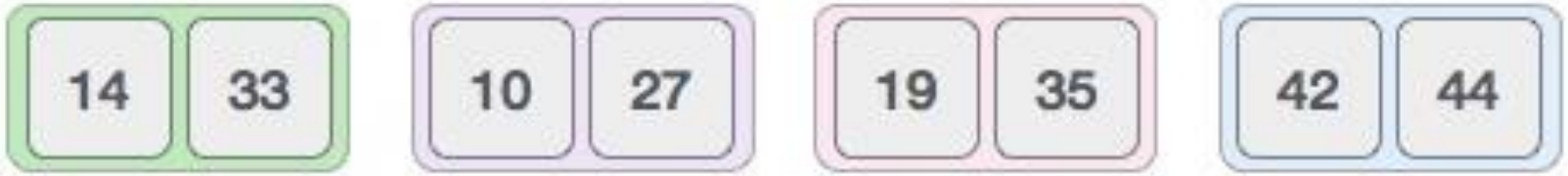


We further divide these arrays and we achieve atomic value which can no more be divided.

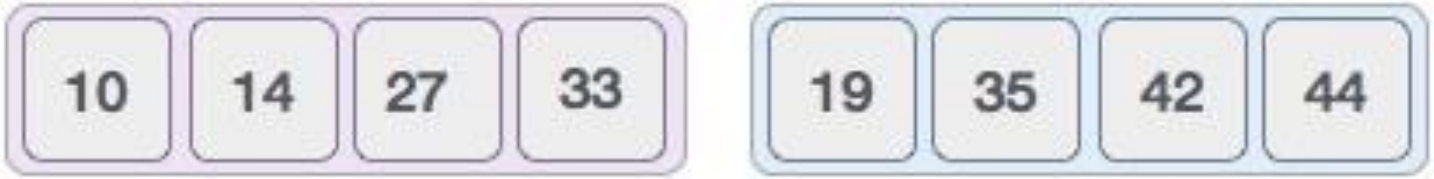


Now, we combine them in exactly the same manner as they were broken down. Please note the color codes given to these lists.

We first compare the element for each list and then combine them into another list in a sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order of 19 and 35 whereas 42 and 44 are placed sequentially.



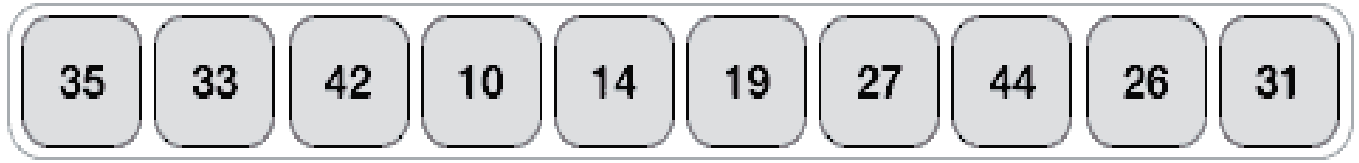
In the next iteration of the combining phase, we compare lists of two data values, and merge them into a list of found data values placing all in a sorted order.



After the final merging, the list should look like this −

Now we should learn some programming aspects of merge sorting.

Quick Sort



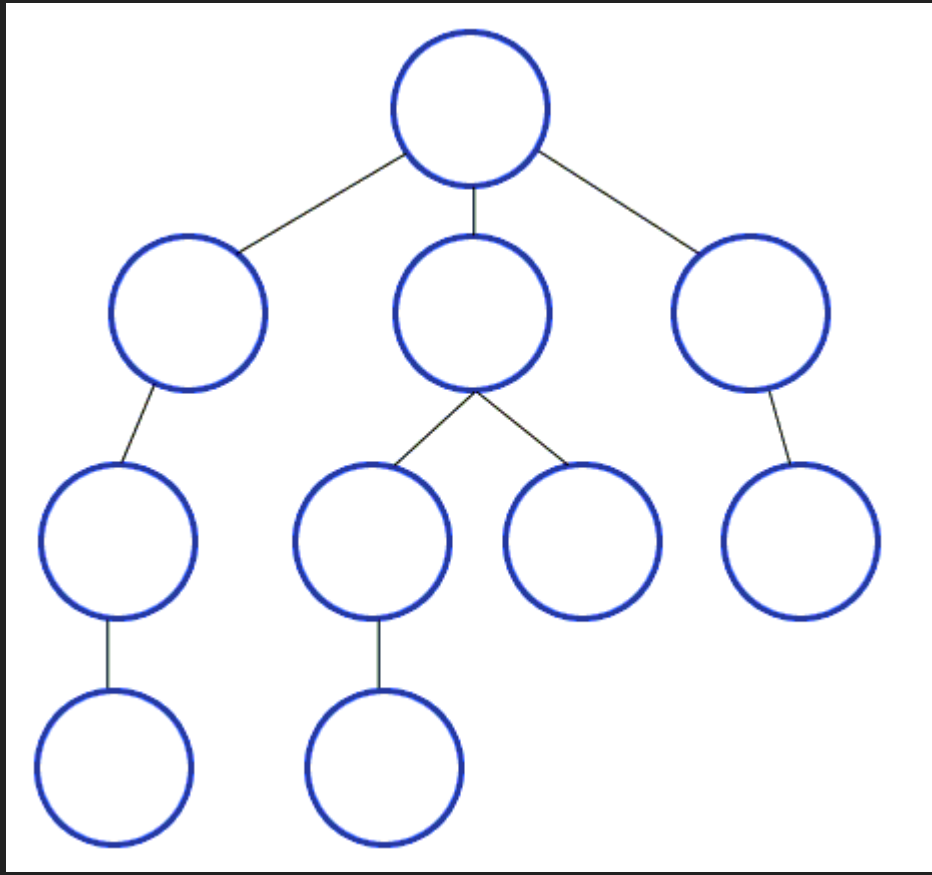
Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.

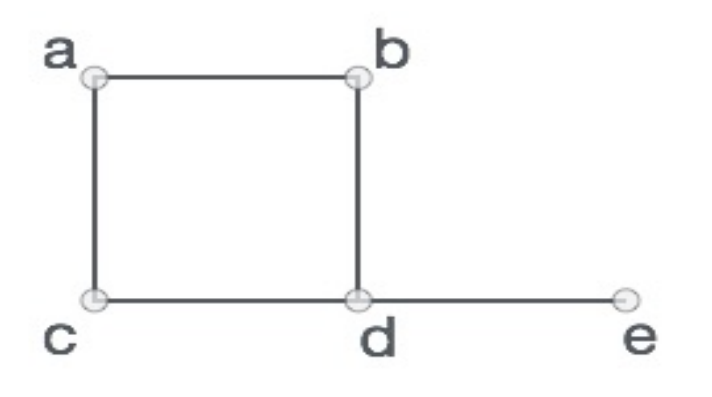
Quicksort partitions an array and then calls itself recursively twice to sort the two resulting subarrays. This algorithm is quite efficient for large-sized data sets as its average and worst-case complexity are $O(n^2)$, respectively.

Graph Data Structure

A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as vertices, and the links that connect the vertices are called edges.

Formally, a graph is a pair of sets (V, E), where V is the set of vertices and E is the set of edges, connecting the pairs of vertices.



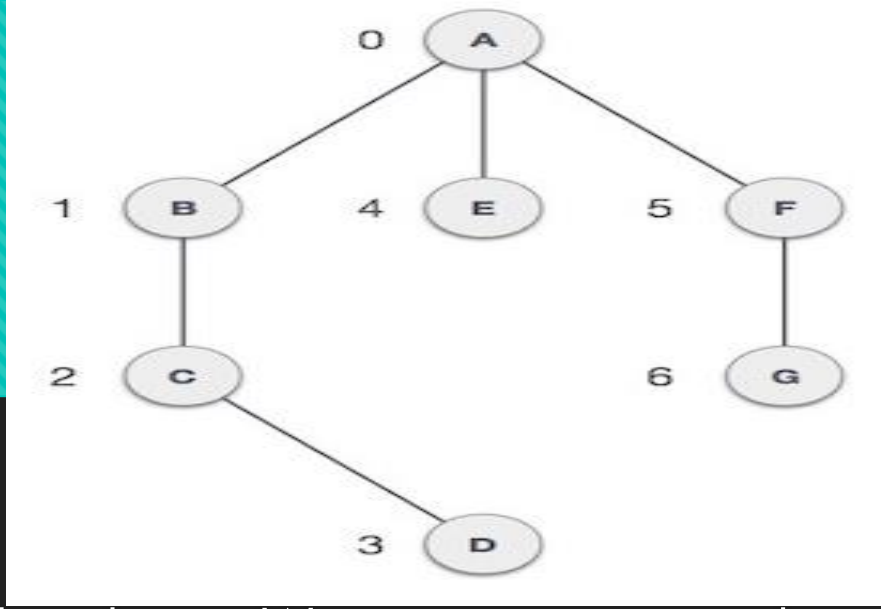


LETS FIND ALL THE VERTICES AND EDGES

V=VERTICES
E=EDGES

V={a,b,c,d,e}
E={(ab),(ac),(cd)),(de),(bd)}

Graph DS



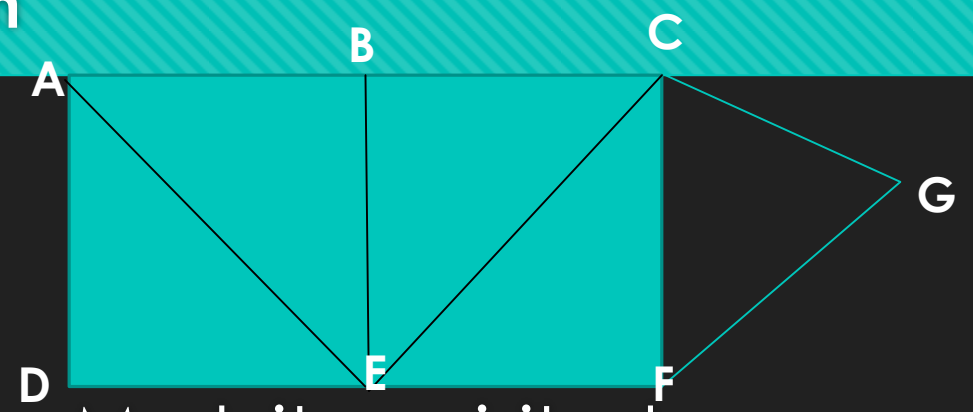
Mathematical graphs can be represented in data structure. We can represent a graph using an array of vertices and a two-dimensional array of edges

- **Vertex** – Each node of the graph is represented as a vertex. In the following example, the labeled circle represents vertices. Thus, A to G are vertices. We can represent them using an array as shown in the following image. Here A can be identified by index 0. B can be identified using index 1 and so on.
- **Edge** – Edge represents a path between two vertices or a line between two vertices. In the following example, the lines from A to B, B to C, and so on represents edges. We can use a two-dimensional array to represent an array as shown in the following image. Here AB can be represented as 1 at row 0, column 1, BC as 1 at row 1, column 2 and so on, keeping other combinations as 0

- **<u>Adjacency</u>** – Two node or vertices are adjacent if they are connected to each other through an edge. In the following example, B is adjacent to A, C is adjacent to B, and so on.
- **<u>Path</u>** – Path represents a sequence of edges between the two vertices. In the following example, ABCD represents a path from A to D.

Depth First Traversal

Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration

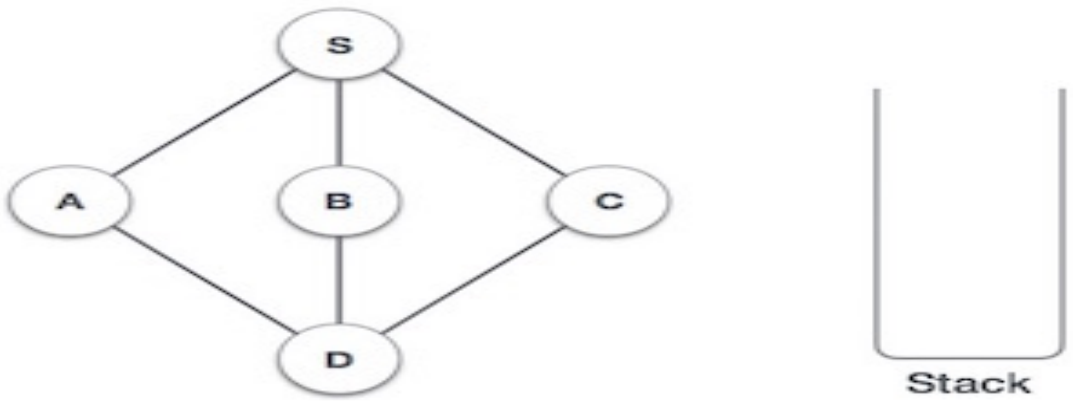
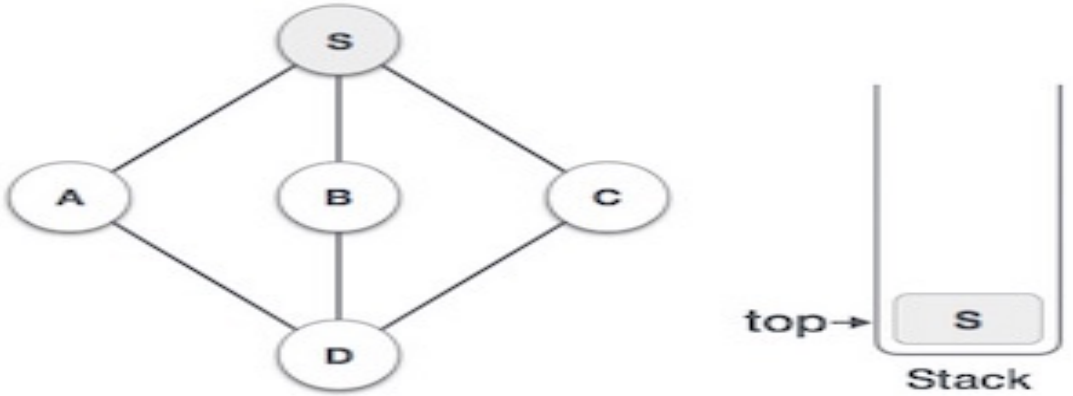
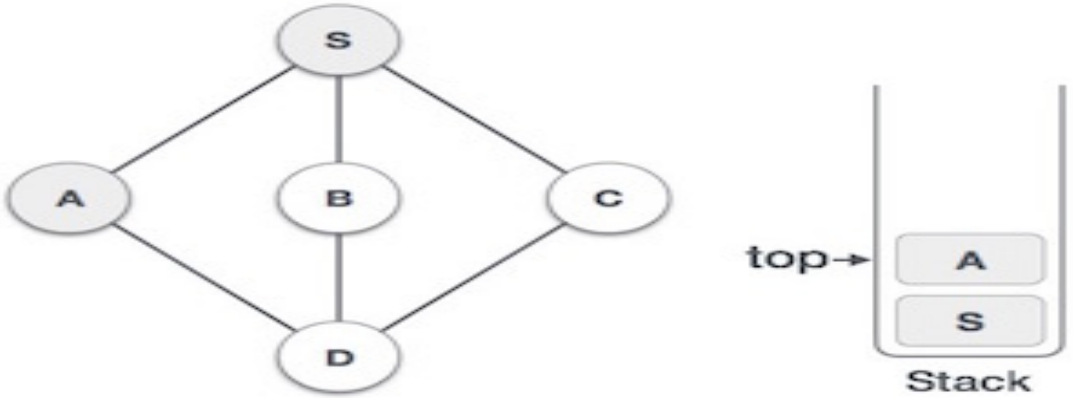


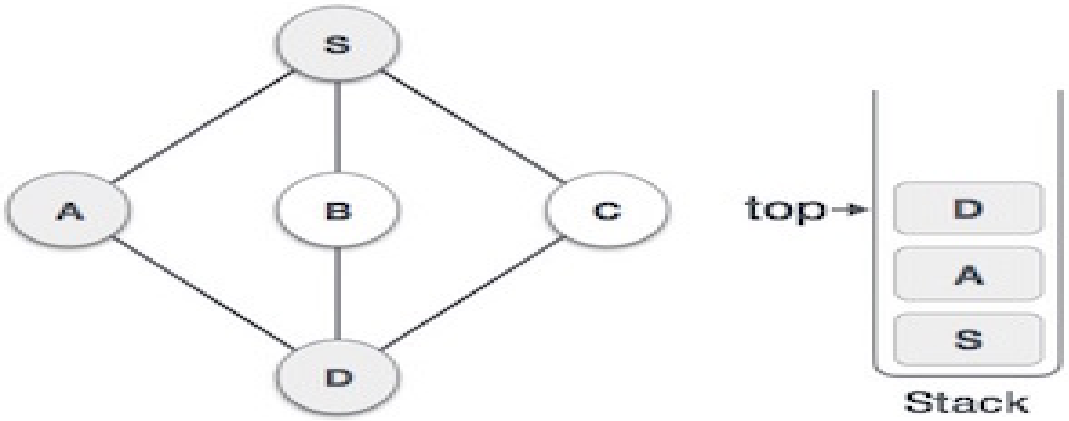
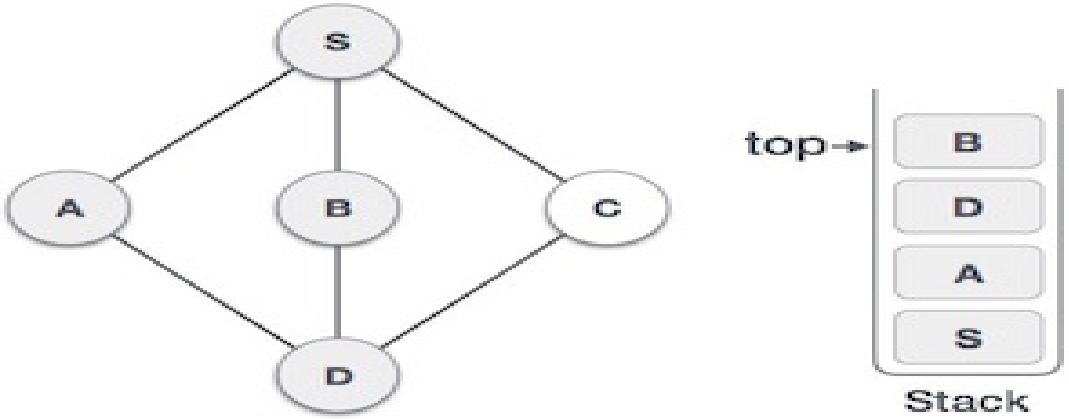
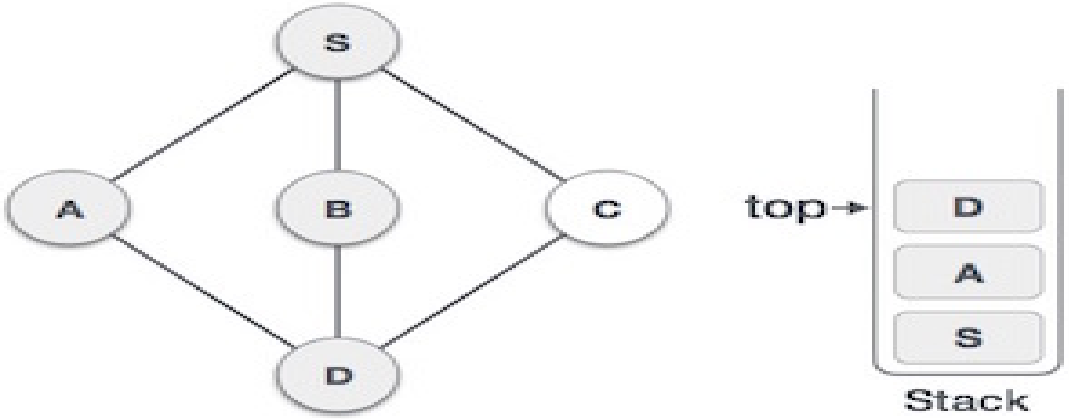
RULES:

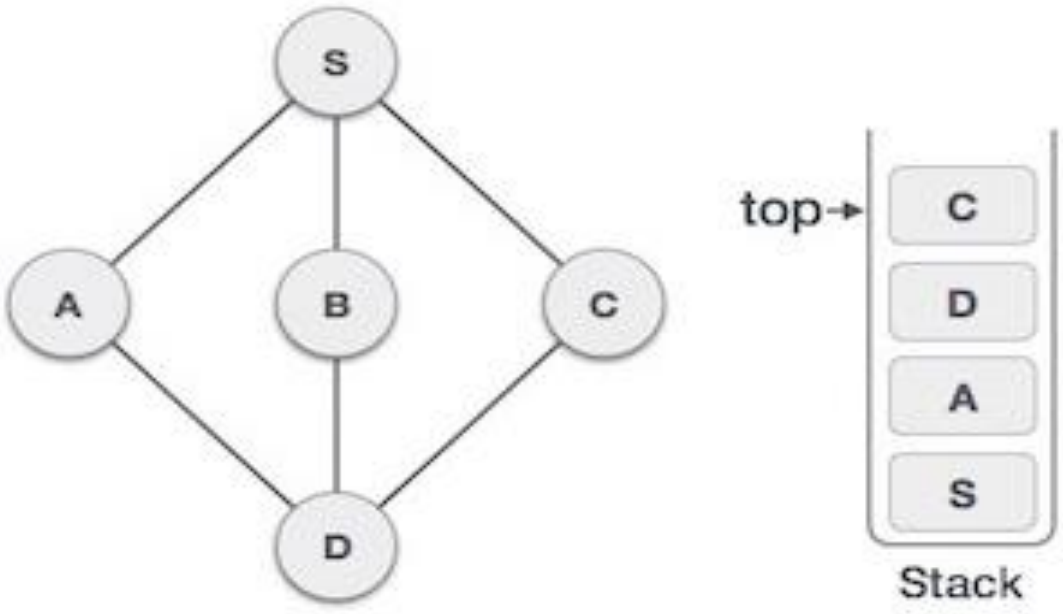
Rule 1 – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.

Rule 2 – If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)

Rule 3 – Repeat Rule 1 and Rule 2 until the stack is empty.

Step	Traversal	Description
1		Initialize the stack.
2		Mark **S** as visited and put it onto the stack. Explore any unvisited adjacent node from **S**. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.
3		Mark **A** as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both **S** and **D** are adjacent to **A** but we are concerned for unvisited nodes only.

4		Visit **D** and mark it as visited and put onto the stack. Here, we have **B** and **C** nodes, which are adjacent to **D** and both are unvisited. However, we shall again choose in an alphabetical order.
5		We choose **B**, mark it as visited and put onto the stack. Here **B** does not have any unvisited adjacent node. So, we pop **B** from the stack.
6		We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find **D** to be on the top of the stack.

7		Only unvisited adjacent node is from **D** is **C** now. So we visit **C**, mark it as visited and put it onto the stack.

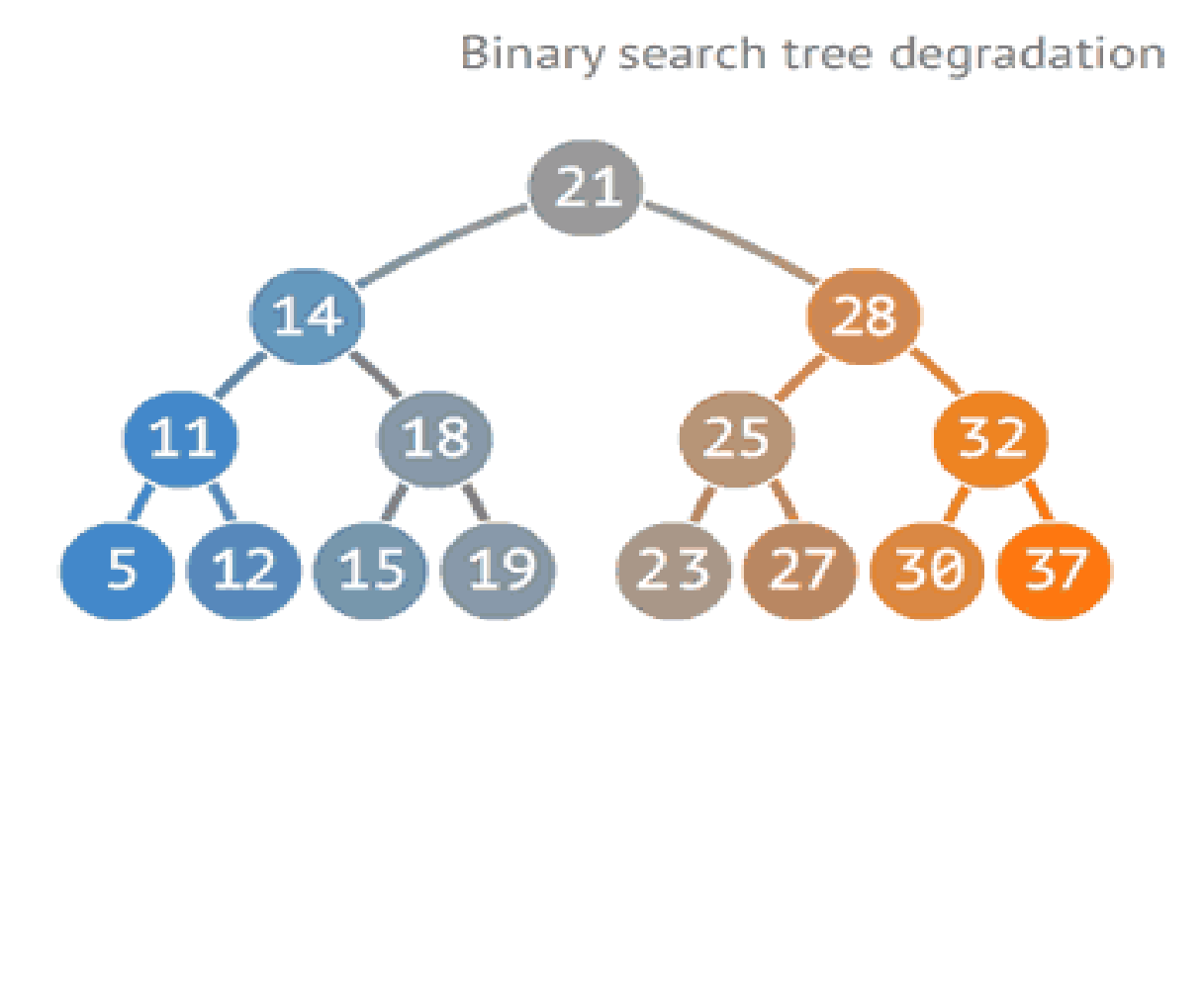
As **C** does not have any unvisited adjacent node so we keep popping the stack until we find a node that has an unvisited adjacent node. In this case, there's none and we keep popping until the stack is empty.

Tree

Tree represents the nodes connected by edges

Binary Tree is a special data structure used for data storage purposes. A binary tree has a special condition that each node can have a maximum of two children. A binary tree has the benefits of both an ordered array and a linked list as search is as quick as in a sorted array and insertion or deletion operation are as fast as in linked list.

BINARY SEARCH TREE RULE:
FIRST:ROOT
LEFT IS <ROOT
RIGHT>ROOT



Important Terms

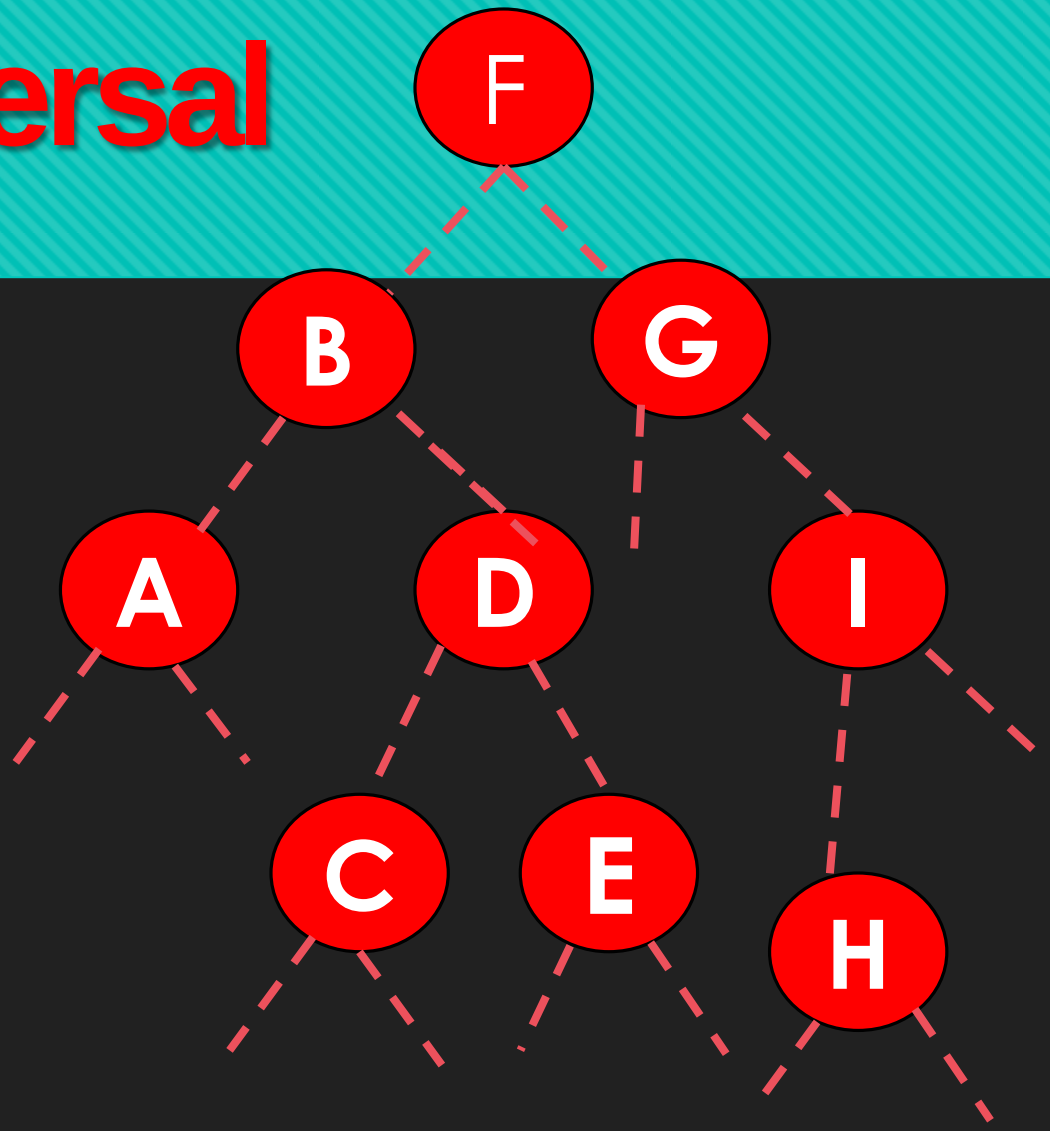
- **Path** − Path refers to the sequence of nodes along the edges of a tree.
- **Root** − The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.
- **Parent** − Any node except the root node has one edge upward to a node called parent.
- **Child** − The node below a given node connected by its edge downward is called its child node.
- **Leaf** − The node which does not have any child node is called the leaf node.
- **Subtree** − Subtree represents the descendants of a node.
- **Visiting** − Visiting refers to checking the value of a node when control is on the node.
- **Traversing** − Traversing means passing through nodes in a specific order.

Tree Traversal

Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree −

- Pre-order Traversal (**ROOT ,LEFT, RIGHT**)
- In-order Traversal (**LEFT, ROOT,RIGHT**)
- Post-order Traversal (**LEFT, RIGHT,ROOT**)

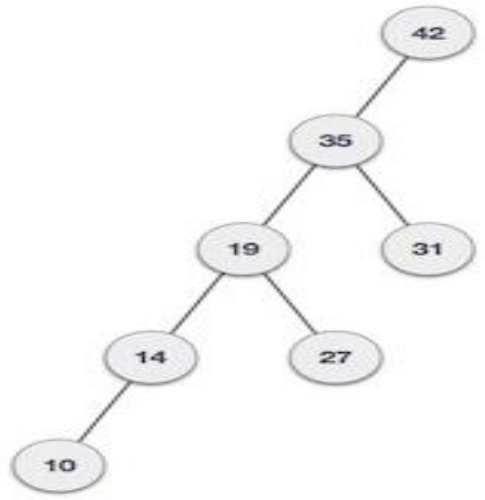
Trick to Find Tree Traversal



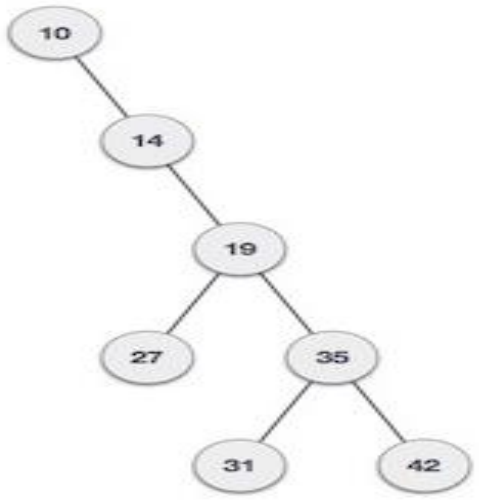
PreOrder: Visited First Time.
InOrder: Visited Second Time.
PostOrder: Visited Third Time.

ANSWER:
Pre:- FBADCEGIH
In:- ABCDEFGHI
Post:- ACEDBHIGF

AVL Tree



If input 'appears' non-increasing manner



If input 'appears' in non-decreasing manner

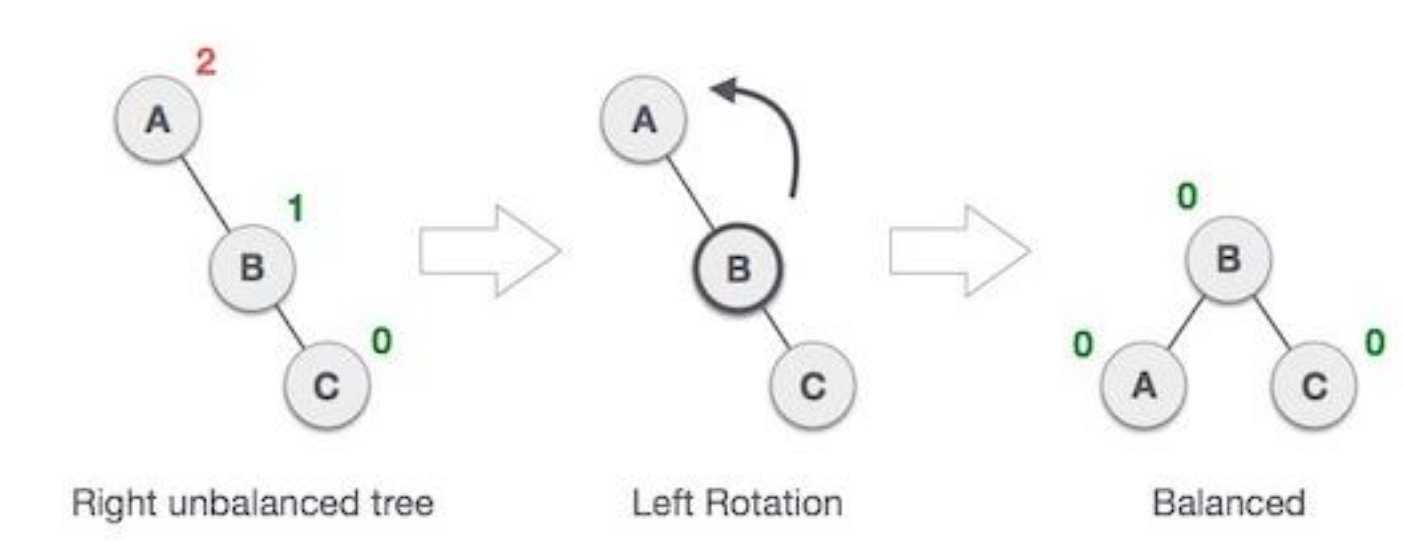
It is observed that BST's worst-case performance is closest to linear search algorithms, that is O(n). In real-time data, we cannot predict data pattern and their frequencies. So, a need arises to balance out the existing BST.

Named after their inventor **Adelson, Velski** & **Landis**, **AVL trees** are height balancing binary search tree. AVL tree checks the height of the left and the right sub-trees and assures that the difference is not more than 1. This difference is called the **Balance Factor.**

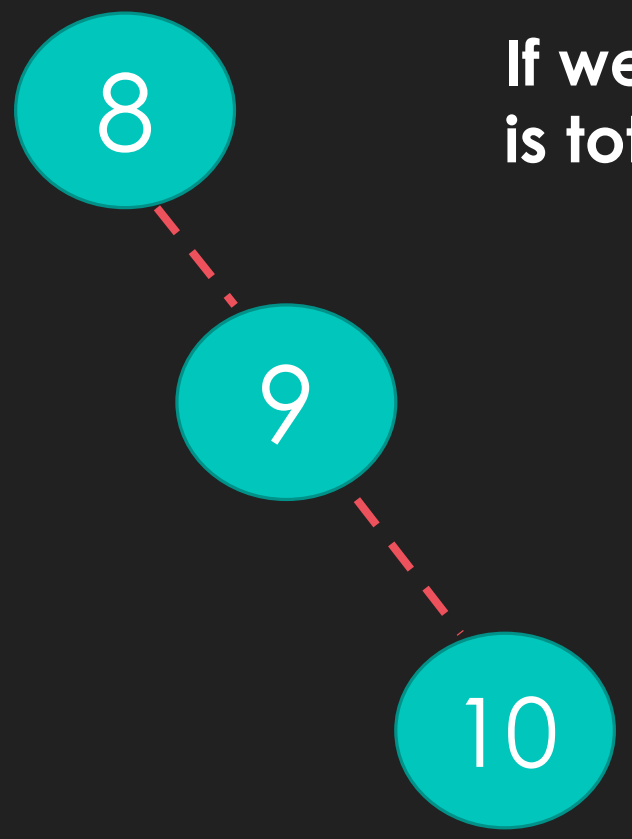
AVL Rotations

To balance itself, an AVL tree may perform the following four kinds of rotations –

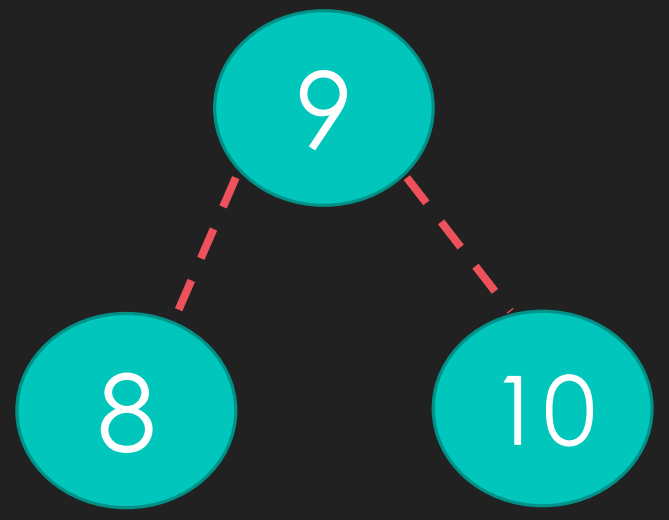
- Left rotation
- Right rotation
- Left-Right rotation
- Right-Left rotation



8 , 9 , 10 (Make A Binary Tree)



If we Rotate Left to 8 than it is totally Balanced



Binary Tree but it is Unbalanced

AVL or BALANCED TREE