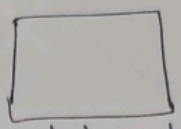
Data Structures and Algorithms (CS2004-4)

OOPS

- → Practical
- → Class
- → Object
- → Inheritance
- → Polymorphism
- → method Overloading

DS / Algorithms →

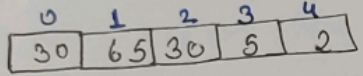


geeksforgeeks

Data Structures

It is a place where we store data following some rules on the stored data we can perform operations.

Linear Search



→ Data Structure

Array = Data Structure

a[3] = 5
accessing data

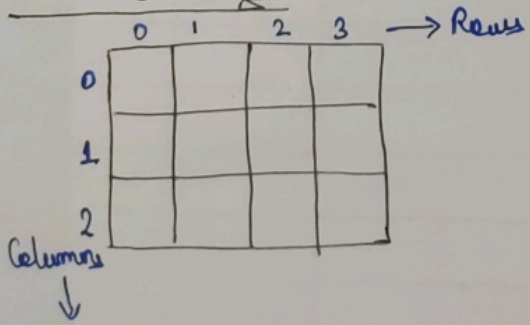
- → Search
- → Adding or deleting new data

Data Structures

- Linked List
- Stack
- Queue
- Tree
- Recursion
- Time Complexity

increment OOPS
- loops.
- if-else
- methods

2D Arrays



3 Rows
4 columns
12 elements (Rows X Columns)

Data is being stored in a tabular form

a1[1][2] ⟶ To fetch element

row → 1, column → 2

Subscription Variables

a[1][2] = 57

a[2][0] =

```
int[][] a = new int[3][4];
```

- `int[][]` → Data type
- `a` → Array Name
- `new int[3][4]` → Reserve Same Space in the memory

memory space for 12 elements

How to assign values in empty array.

First Way

repeat

```
a[0][0] = 57
a[0][1] = 23
  :
  :
a[2][3] = 60
```

Second Way

```
a[][] = {{57,23,62}, {10,3,24}, {32,19,17,193}}
```

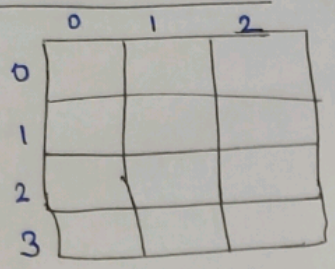
Third Way (Traversal of 2-D Array)

0	1	2
3	6	9

Two Ways

- Row Wise
- Column Wise

Row wise Traversal



a[x][y] → column no.
↓ row no.

To traverse the loop:
Outer loop to traverse the row
Inner loop to traverse the column.

```
int i=0, j=0;
  for (i=0; i<4; i++)
  {
     for (j=0; j<3; j++)
     {
        System.out.print(a[i][j])
     }
   System.out.println();
  }
```

R	C	
0	0	→ 16
0	1	→ 100
0	2	→ 23

	0	1	2
0	37	51	61
1	20	15	28
2	6	3	57

2 loops

1 loop → for traversing the row → Outer loop
2nd loop → for traversing column → Inner loop.

traversing all the elements in particular loop.

```
for (i=0 ; i<3 ; i++)
{
    for (j=0 ; j<3 ; j++)
    {
        System.out.println (a[i][j]);
    }
    System.out.println ();
}
```

```
a[0][0]      a[1][0]
a[0][1]      a[1][1]
a[0][2]      a[1][2]
  |            |
  |            |
```

Q. WAP to input elements of 2D array from user.

Program

```
import java.util.Scanner;
public class Prog1
{
    public static void main (String[] args)
    {
        Scanner obj = new Scanner (System.in);
        System.out.println (" Enter no. of rows and columns");
        int r = sc.nextInt ();
        int c = sc.nextInt ();
        int[][] a1 = new int[r][c];
        for (int i=0 ; i<r ; i++)
        {
            for (int j=0 ; j<c ; j++)
            {
                a[i][j] = sc.nextInt ();
```

Q. WAP to print sum of each rows..

Program

```
int sum = 0
for (int i=0 ; i < r ; i++)
  {
    for (int j=0 ; j < c ; j++)
      {
        sum = sum + a[i][j];
      }
      System.out.println (sum);
  }
```

Q. WAP to print diagonal elements.

(Using two loops)

Program

```
for (int i=0; i <3; i++)
  {
    for (int j =0 ; j<3; j++)
      {
        if (i == j)
          {
            System.out.println (a[i][j]);
          }
      }
}
```

	0	1	2
0	15	23	36
1	10	12	17
2	57	3	6

a[0][0]
a[1][1]
a[2][2]

diagonal elements

Using one loop

```
for (int i=0; i< row ; i++)
{
    System.out.println (a[i][i]);
}
```

ith row
a[i][i]
↑ diagonal element in ith row

Addition of 2-D Array

Q.

A

3	6	5
2	1	0

B

8	7	6
3	5	-1

C

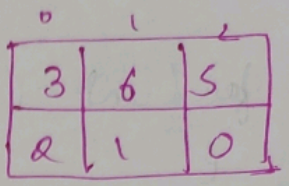
2×3

WAP to add A and B and store it in C.

Program

```
public class Sum
{
  public static void main (String[] args)
  {  int A[][] = {{3,6,5} , {2,1,0}};
     int B[][] = {{8,7,6} , {3,5,-1}};
     int [][] C = new int [2] [3];
     for (int i = 0,i < 2; i++);     // rows
       {
       for(int j= 0; j< 3 ; j++);    // columns
         {
            C[i][j] = A[i][j] + B[i][j];
         }
       }
}
```

Q. WAP to find sum of elements in each column




```
int a[][] = {{3,6,5},{2,1,0}}

for (int i=0; i<3; i++)  // columns
{
    for (int j=0; j<2; j++)  // rows
    {
        sum += a[j][i];
        System.out.println(sum);
    }

    System.out.println(sum);
    sum = 0
}
```

Sum: 5 7 5

(OOPS) Object Oriented Programming System

it way of designing / writing one's code.

→ Safe and secure
→ Easy to Maintain
→
* It binds data and code into a single unit.

Characteristics of OOPS

① Class & Object ★ → fundamental concept of OOPS
② Inheritance ★
③ Method Overloading ★
④ Encapsulation
⑤ Abstraction
⑥ Polymorphism ★

Object

— features / characteristics
— display some behaviour

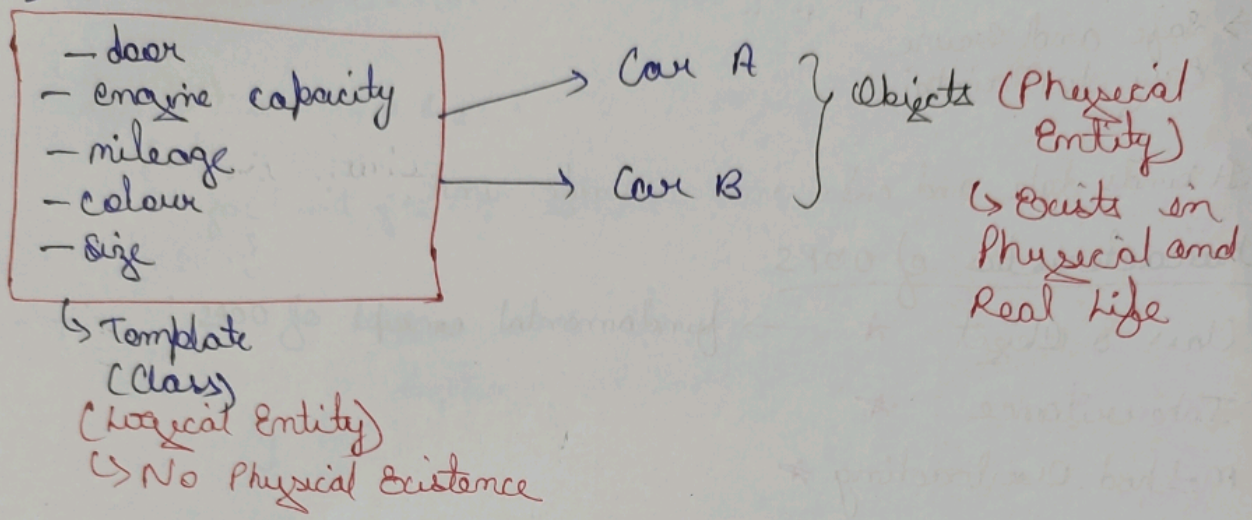
Ex. Dog → Object

Features	Behaviour
4 legs	barks
colour	
2 eyes	
Tail	

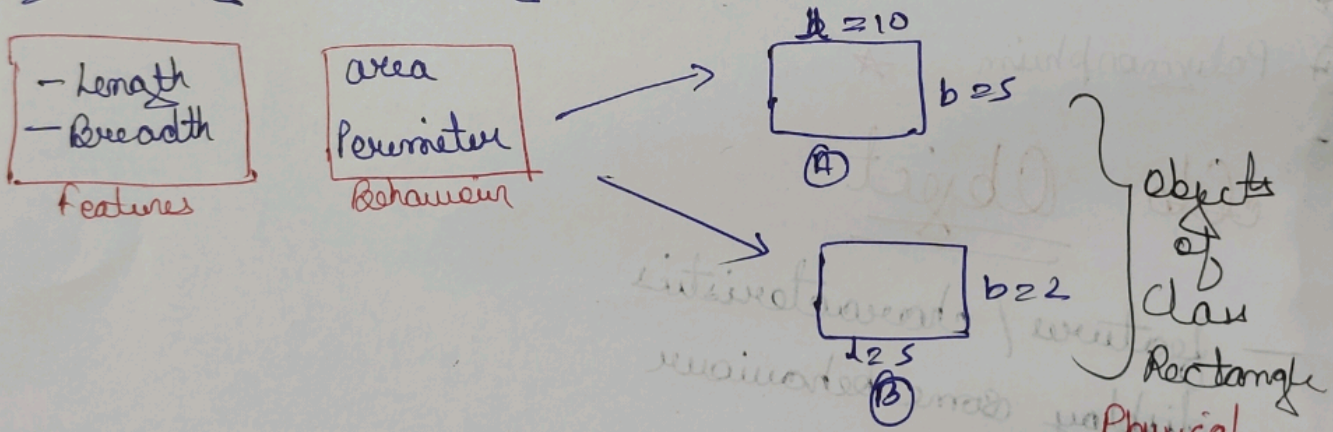
Class

→ A template / blueprint to create object

Ex: ~~Object~~ Car → Class



Ex: ~~Object~~ – Rectangle – Class



Defining Class **Creating Class**

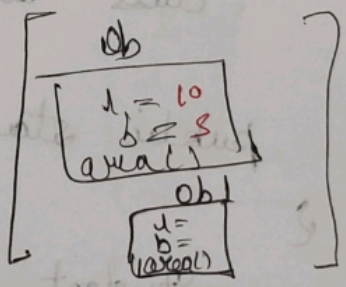
```
class Rectangle
{
    int l ;
    int b ;

    void area()    // method → behaviours
    {
        System.out.println("Area = " + l*b);
    }
}

class P1    //path to create object.
{
    public static void main()
    {
        Rectangle ob = new Rectangle();
```

features

non-static method (requires object later)

within a class variables → Any method within the class can access them directly.

Object name.




```
Rectangle ob1 = new Rectangle();
l=10 , b=5;   X
ob.l=10 , ob.b=5;
ob.area();   → 50 will be printed
}
```

Assigned values to object.

Creating Objects

ob1.l = 6 , ob1.b = 2;

ob1.area() → 12 will be printed

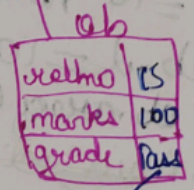
Q. Create a class called student with two features → roll no and marks, method called grade() → will check if marks >50, display pass otherwise fail.

```
class Student
{
    int rollno;   } instance variables
    int marks;
    void grade()
    {
        if (marks > 50)
            System.out.println("Pass");
        else
            System.out.println("Fail");
    }
}

class Surbhi
{
    public static void main()
    {
        Student ob = new Student();
        ob.rollno = 15, ob.marks = 100;
        ob.grade();
    }
}
```

Defining class

Creating Object

Values assigned



will be created in memory

Other ways of assigning values to an object :-

→ Setter method

→ Contructor

Setter Method

```
class Rectangle
{
int l,b;
void setValues(int x, int y)
{
  l=x;
  b=y;
}
void area()
{
  System.out.println("Area = "+l*b);
}
}

class P1
{
public static void main()
{
Rectangle ob = new Recangle();
ob.setValues(10,2)
ob.area();
}
}
```

ob
l = 10
b = 2
setValues()
area()

Getter Method

```
class Rectangle
{
  int l,b;
  void getL()  // displays current values
  { System.out.println("l value = "+l)
  }
  void getB()
  { System.out.println("b value = "+b);
  }
}
class P1
{ public static void main()
{
    Rectangle ob = new rectangle();
    ob.getL();
    ob.getB();
    ob.area();
}
```

calculates area

```
void area()
{
  s.o.p(l*b);
}
void setValues(int x, int y)
{
  l=x;   Sets values to variables
  b=y;
}
```

Constructors

→ block of code
→ used for initialized objects
→ defined inside Rectangle class

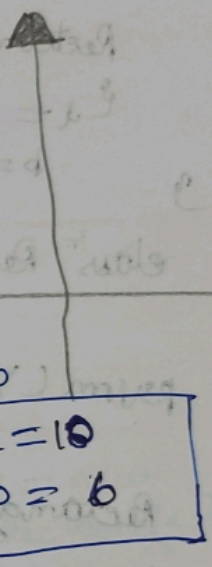
Syntax :-

```
class Rectangle
{
    int l, b ;        // instance variables
    Rectangle (int x, int y)        → Parametrised Constructor
    {
        l=x;    // ob.l = x = 10
        b=y;    // ob.b = y = 6
    }
    void area()
    {
        System.out.println( l*b);
    }
}
```

constructor name → same as the class name

Constructor doesn't return any value

Constructor's only task is to assign values to instance variables




```
class P1
{
    psvm ()
    {
        Rectangle ob = new Rectangle ( );
        new Rectangle (10, 6);
        ob.area ( );
    }
}
```

Method ~~constructor~~ can be called any no. of times but constructor can be called once while creating objects

Constructor are of three types

- Parameterised Constructor
- No argument Constructor
- Default Constructor

No argument Constructor

```
class Rectangle
{
    int l,b;
    Rectangle();
    {l = 10;
     b = 2;
    }
}
```

```
void area()
{
    System.out.println(l * b);
}
```

```
class Rectangle
{
    psvm();
    {
        Rectangle ob = new Rectangle();
        ob.area();
    }
}
```

Default Contructor

```
class Rectangle
{
    int l,b;
    Rectangle();
    {
        S.O.println("Default");   //NO Argument
    }
}
```

```
void area()
{
  S.o.println(l*b);
}
}
```

```
class P1
{
  psvm
  {
    Rectangle ob = new Rectangle();
    ob.area();
  }
}
```

Constructor Overloading

```
class Rectangle()
{
  int l,b;
  Rectangle() //No argument
  {
    l = 10;
    b = 2;
  }
  Rectangle(int x, int y)
  {
    l = x
    b = y
  }
}
```

Parameterised constructor

x
l = 50
b = 6

Y
l = 10
b = 2

```
class P1
{
  psvm()
  {
    Rectangle x = new Rectangle(50,6);
    Rectangle y = new Rectangle();
  }
}
```

In constructor overloading, no argument or default constructor is created so that even if the user forgets to give the values the program should work fine, (in case of no argument, value is assigned, in default default values is assigned

Example

```
class A
{
    int x;
    A(int y).
    {
        x=y
    }
    A( )
    {
        x=57
    }.
}.
```

```
class B
{
    A.ob = new A( );
    ob.x=57;
}
```

→ ob.x = 57;

No argument constructor will be called since user hasn't provided any value.

Q. WAP to overload constructor with following details.

class Student
→ roll
→ marks
→ grade()

class cannot be public

Program

```
class Student
{
    int roll, marks;

    Student (int x, int y)
    {
        roll = x;
        marks = y;
    }

    Student ()
    {
        roll = 15;
        marks = 99;
    }

    void grade ()
    {
        if (marks > 50)
        {
            S.O.ptln ("Pass");
        }
        else
            S.O.ptln ("fail");
    }
}
```

public class Result → this has to be public.
{
psvm
→ No argument constructor will be called
{
Student ob = new Student (); || Student ob = new Student (44, 93);
~~ob.getStudent ();~~
ob.grade ();
→ Parametrised constructor will be called
}
}

* If using two classes for constructor overloading, class containing constructor should not be public, and class containing psvm should be public, ~~and~~.

* If using one class for constructor overloading, it should be public.

Marks → Sharp No. of subs → For which number

Method Overloading

→ methods with same name & and similar (not exactly same) ~~for~~ behaviour behaviour.

→ Differs with no. of argument.

→ & or Differs with data type of arguments.

Example

```
class Shape
{
int area(int l, int b)   // area of rectangle
    {
        return l * b;
    }
int area(int l)          // area of square.
    { return l * l;
    }
int area(double r)
    { return (int)(3.14 * r * r); // area of circle
    }
}
```

```
public class P1
{
    psvm()
    {
        Shape s = new Shape();
int x = s.area(10);   // area of square will be called
int y = s.area(100,5); // area of rectangle will be called
int z = s.area(12.5); // area of circle will be called.
    }
}
```

constructor → used to pass values to instance variable

In this program, no instance variables so constructor remains empty.

* return type can't be used as a basis for method overloading.

Types of Variables

```
class A
{
    int x;                          // ① instance variable
    A(int d)                        // ② local variable
    {
        x = d;
    }
    int calc(int d)                 // local variable
    {
        return d * d;
    }
}
```

③ Static Variable

```
class A
{
    int x;
    static int y;                   // static variable
    int sq(int x)
    {
        return x * x;
    }
}

class B
{
    PSV m()
    {
        A ob = new A();
        ob.x = 10;
        A ob1 = new A();
        ob1.x = 20;
        A.y = 5;                    // class name
    }
}
```

// accessing static variable

ob
x = 10
sq()

ob1
x = 20
sq()

Static Variable →
- One copy of static variable is shared by each object and not each object has their own y.

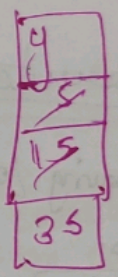
Instance Variable →
Each object can have their copy of x variable

→ has only copy that is shared among all the objects of the class

→ no objects needed to access static variable

→ If no value is assigned to static variable, default value will be provided, or some ~~pre assigned~~ preassigned value.

```
class A
{
  static int y = 5;
  int x;
  int sq()
  {
    y = y + x;
    return y;
  }

  psvm()
  {
    A ob = new A();
    ob.x = 10;
    int a = ob.sq(); // a = 15
    A.ob1 = new A()
      ob1.x = 20;
  int b = ob1.sq(); // b = 35
    y = y + ob1.x; // y = 35
```



Static Method → doesn't require object for being called.

* Static method → can access static variables of the same class

* 'this' and 'super' are two keywords, static methods cannot use inside them.

```
class A
{
    static int area (int l)
    {
        return l * l;
    }
    psvm ( )
    {
        A.area (5);
    }
```

→ can skip 'this' since psvm and method be called/ are in same class.

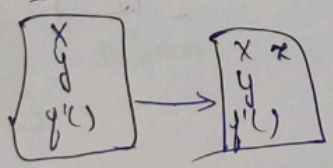
```
static int area(int l)
{ int x;
  return l * x; → it is possible
}
```

Inheritance

↓

acquiring / getting some features / property of one class to another class.

(Here class B is acquiring features of class A)



class A
Parent Class
or Super class

class B
child class
or Subclass

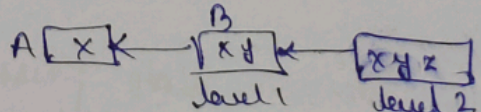
example of single inheritance

→ Reusability

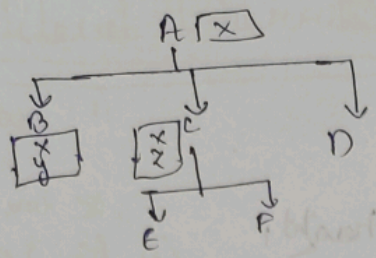
① Single Inheritance (where only one class acquiring the property of another class)

② Multi Level Inheritance

A ← B ← C.



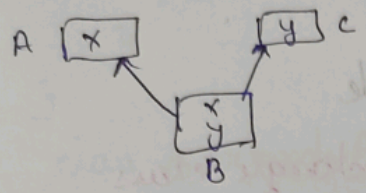
③ Hierarchial Inheritance → (one class has many child class)



class here B, C, D are acquiring property of class A.

directly implemented in Java.

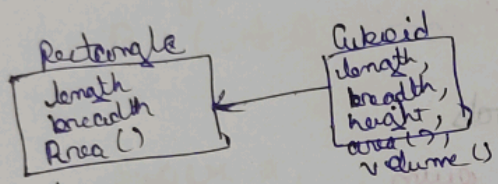
④ Multiple Inheritance → A child class will have multiple parent class.



Here class B is acquiring properties from both A & C.

⇒ It can't be implemented directly in Java

can be implemented using interface



Syntax

```
class Rectangle
{
  int length, breadth;
  Rectangle (int d, int b);
  {
     length = d;
     breadth = b;
  }
  int area ()
  {
    return length * breadth;
  }
}
class cuboid extends Rectangle
{
   int height;
   cuboid (int d, int b, int h)
   {  length = d
      breadth = b  ] // super (d, b);
```

This constructor will be called

```
    height = h;  ---> not inherited
  }
  int volume()
  {
    return length * breadth * height;
  }
}
```

* Cuboid class has three instance variable
  - → length
  - → breadth
  - → height → of it's own.

  ] inherited from Rectangle class

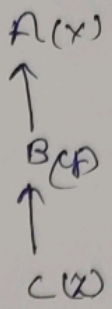
* Cuboid class has two methods.
  - → volume() → of it's own
  - → area() → inherited from Rectangle class

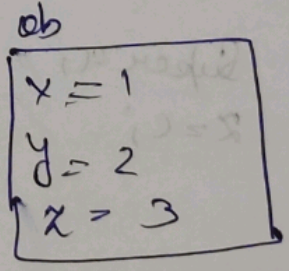
```
class P1
{
  psvm()
  {
    cuboid c = new Cuboid(10, 6, 1);
    c.area();  -> 10X6 = 60
    c.volume(); -> 10X6X1 = 60
```

c
l = 10
b = 6
h = 1
area()
vol()

Multilevel Inheritance

```
class A
{
    int x;
    A(int a)
    { x=a;
    }
}
class B extends A
{
    int y;
    B (int a, int b)
    {
        super a;
        y=b;
    }
class c extends B
{   int z;
    c(int a, int b, int c)
    {
        super(a,b);
        z=c;
    }
}
C  ob = new C (1,2,3);
```





Hierachical Inheritance

```
class A
{
  int x;
  A(int a)
  {
    x = a;
  }
}
class B extends A
{
  int y;
  B(int a, int b)
  {
    super a;
    y = b;
  }
}
class C extends A
{
  int z;
  C(int a, int c)
  {
    super a;
    z = c;
  }
}
```

Heading

```
class A
{
  int x
  void setA(int a)
  {
    x = a;
  }
}
class B extends A
{
  int y;
  void setB(int a, int b)
  {
    x = a;   -> super.setA(a);   (can be replaced with this)
    y = b;
  }
}
A ob = new A();
ob.setA(10);

B ob1 = new B();
ob1.setB(5, 10);
```

ob

x = 10

ob1

x = 5

y = 10

Method Overriding

— Polymorphism
— Inheritance]

```
class A
{
   int x;
   void display()
   {
      S.O.p(x);
   }

Class B extends A.
{  int y;
   void display()
   {
      //body sopln(x);
           Sopln(y);
   }

B ob = new B();
ob.x = 5 ; ob.y = 15
ob.display(); ——> 5 will be displayed
A ob1 = new A()
ob1.display();
```

ob

x=5
y=15
display()

Method Overriding is a process by which we create a method having same name, parameter, return type in child class.

Final Keyword

→ adding final keyword before a method stops it from being overridden.

Ex.
```
final void display ()
{
    Sop(x);
}
```

→ adding final before class, that class can't be further inherited.

Ex. final class A

→ adding final before variable, it's value becomes constant and cannot be changed.

Polymorphism

Method Overloading (Static)

Ex.
```
void a(int x)
{
}
void a(int x, int y)
{
}
```

Method differs in no of parameters or data type of parameters

Method Overriding (Dynamic)

```
void a(int x)   → Parent class
{
}
void a(int x)   → child class
{
}
```

Method does not differ in no. of parameters or data type of parameters

Abstract Class

→ It has to contain atleast 1 abstract method
→ Can contain normal methods
→ instance variables
→ constructors
→ Can be inherited by another class.
→ Can't be instantiated

Example

```
abstract class A
{
    void display()
    {
        S.O.pln("d");
    }
abstract void dl();  ——→ abstract method
}
```

```
psvm()
{
    A ob = new A(); X ——→ Object can't be created
}
```

```
Class B extends A
{
    void dl()
    {
        S.O.pln("Class is B");
    }
}
```

```
psvm()
{
    B ob = new B();
    ob.display();
    ob.dl();
}
```

Abstract Method

→ Body of the method not defined

→ only method declaration is done i.e, return - datatype, name, parameters.

Ex-

```
abstract class Shape
{ abstract void area();
}
class Rectangle extends Shape
{
  void area()
  {
    int l = sc.nextInt();
    int b = sc.nextInt();
    S.o.println(l*b);
  }
}
class Square extends Shape
{
  void area()
  {
    int l = sc.nextInt();
    S.o.println(l*l);
  }
}
class P1
{
  PVSM()
  {
    Rectangle ob = new Rectangle();
    ob.area();
    Square s = new Square();
    s.area();
  }
}
```

Q:- abstract class Shape

```
{
   abstract void x1(int y);  ---> Abstract method with parameter
}
   class c extends Shape
   { void x1(int y)
     { s.o.pln(y*y);
   }
```

Q:- Abstract class with Constructor, instance variable

```
Abstract class Shape
{
   abstract void v1(int y);
   int x;
   Shape(int a)
   {
     x=a;
   }
}
class Square extends Shape
{
   void x1(int y);
   { s.o.pln(y*y);
   }
    Square(int a).
    {
      Super(a);
    }
}
Class P1
{PSVM()
  Square s = new Square(5);
```

Interface (block of code)

→ methods — abstract methods only

→ variable

↳ final, static

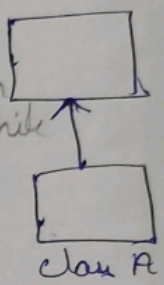
ex - static final int x = 57;

→ cannot create objects of interface

Syntax:

```
interface i1
{
    abstract void f1(int x);
    abstract void f2();
    static final int a = 57;
}
```

one parameter of described data type will be needed. this is only a structure, parameter name can differ while writing the code of this abstract method



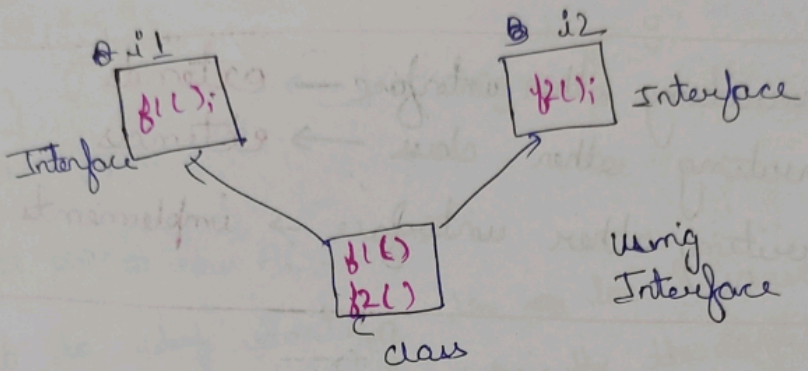
Class Inheriting Interface

```
class A implements i1
{
public void f1(int y)
    {
        S.O.println(y * y);
    }
public void f2()
    {
        S.O.println("f2");
    }

psvm()
{
A ob = new A();
ob.f1(10);
ob.f2();
S.O.println(A.a);
}
```

Abstract Class : 0 — 100 % Abstraction

Interface : 100% Abstraction

* Using Interface, multiple Inheritance can be implemented in Java using Interface.



Syntax:

```
interface i1
{
  abstract void f1();
}
interface i2
{
  abstract void f2();
}
class C implements i1, i2
{
  public void f1()
  {
    S.o.println("From f1");
  }
  Public void f2()
  {
    S.o.println("From f2");
  }
}
```

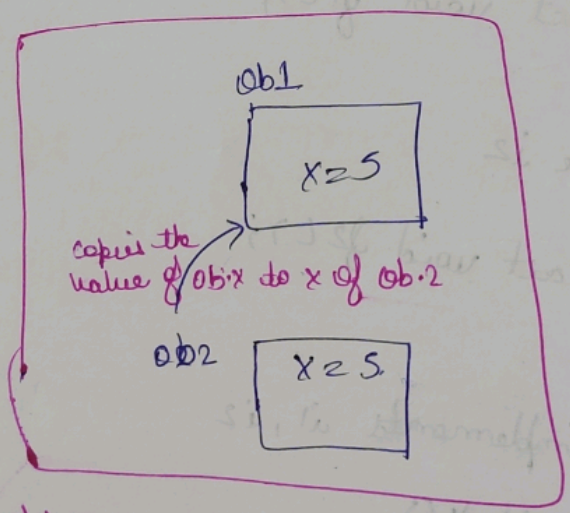
```
class D
{
  psvm ()
  {
    C ob = new C();
    ob.f1();
    ob.f2();
  }
}
```

interface inheriting other interface → extends
class inheriting other class → extends
class inheriting other interface → implements

Constructor

Copy Constructor

```
class A
{
  int x.
  A(int a)
  {
    x=a;
  }
  psvm()
  {
    A.
  A(A ob)
  {
    x= ob.x; // ob2.x = ob1.x
  }
psvm ()
{
  A ob1= new A(5);
  A ob2 = new A (ob1);
}
```



this → Keyword

```
class A
{
    int x, y;
    A(int a, int b)
    {
        x = a;
        y = b;
    }
    psvm()
    {
        A obj = new A(5,2);
```

→ Instance variables.

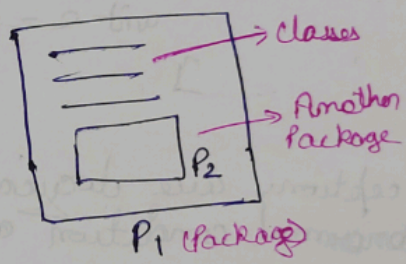
→ local variable

```
A(int x, int y)
{ this.x = x;
  this.y = y;
}
```

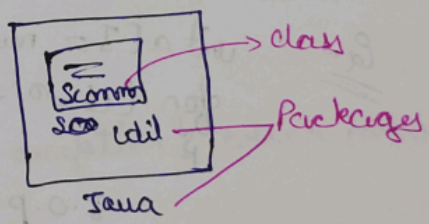
→ to identify the local variable and the instance variable when they have the same name.

Package

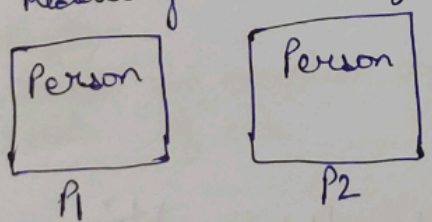
↳ container for storing classes.
→ Better organizes our data.



Ex- import java.util.Scanner;
(Package) (Package) (class)

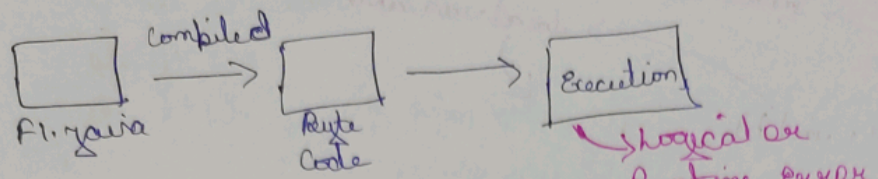


→ Resolving name conflict



Exception Handling

→ Syntax Errors are checked during compilation.



→ Logical or Run Time Error → Occurs during execution of code.

Ex — psvm()
{
int a=57, b=0;
int c=a/b;
}

Exceptions are logical errors in code that leads to abnormal condition and program execution stops.

Eg- int a[] = new int[5]
for(i=0; i<=5; i++)
{
s.o.p(a[6]);
}

Exception Handling → Trey to handle this exception.

Exception

Pre-defined → class in Java
Ex - a/0 (division by zero exception)

a[-1] → Illegal Access

Arrayindex out of Bounds

↳ Child Classes

↳ 1. Arithematic Exception

2. Array out of Bounds

User-defined

Ex -

```
class A
{
  psv m()
  {
    int a = 5/0;
    S.O.pln("ITER");
  }
}
```

System
↳ [creates an exception object → details of exception] error message →
→ state of code

try, catch, finally → keywords to handle exception.

↳ Part of code where exception can occur is kept inside try block.

Syntax

```
class A
{
  Psvm ()
  {
    try
    {
      int a = 5/0;
      S.O.pln("ITER");
    }
    catch (ArithmeticException ab)
    {
      S.O.pln ("ITER") "Ignore"); // managing the exception
    }
    S.O.pln ("CSE 32");
  }
```

Output

```
Ignore
CSE32
```

↳ Minimum one catch block should be used with a try block

Syntax :

```
class A

Psvm ()
{
  try
  {
    int a = 5/6;
    S.O.pln ("ITER");
  }
```

```
catch (ArithmaticException ob)
{
    S.O.pln ("Ignore");
}
catch (ArrayIndexOutOfBoundsException ob)
{
    S.O.pln ("Ignore this");
}
```

↳ For multiple exceptions, ~~first~~ corresponding catch for the first exception will be executed.

Ex:

```
try
{
    int a[] = new int [5];
    int a = 5/0;
    a[6] = 1;
}
catch (ArithmaticException ob)
{
    S.O.pln ("Ignore");
}
catch ( Ar                    Bounds. )
{
    S.Opln ("Ignore this");
}
S.O.pln ("CSE32");
```

Output

```
Ignore
CSE32
```

↳ If corresponding catch block for the first exception is absent, error will occur.

Nested try-catch

```
try
{
    try
    {
        int b = 5/0;
    }
    catch (ArithmeticException ob)
    {
        S.O.println("Ignore A");
    }
    int a[] = new int[5];
    a[6] = 0;
}
catch (ArrayIndexOutOfBoundsException e)
{
    S.O.println("Ignore B");
}
```

Output

```
Ignore A
Ignore B
```

throws Keyword

```
static void m1() throws ArithmeticException
{
    S.O.pln(57/a);
}
main()
{
    try
    { m1();
    }
    catch (ArithmeticException e)
    {
      S.O.pln("Ignore");
    }
}
```

throw Keyword

↳ used to create explicitly, an exception object. by user.

```
PSVm()
{ int a=57/6;
  throw new ArithmeticException();
```

User Defined Exception

```
class MyException extends Exception
{
    MyException(msg);
    {
      super(msg);
    }
}
```

```
psvm ()
{
  int a = sc.nextInt();
  if (a<18)
  {
     throw new MyException ("Ineligible");
  }
}
```

Output

MyException Ineligible

Example

```
psvm ()
{
  try
  {
    int a = sc.nextInt();
    if(a<18)
    { throw new MyException ("Ineligible");
    }
  }
  catch (MyException e)
  { s.o.println ("Ingnore");
    S.O.println (e);
  }
  s.op.println ("I");
}
```

finally

* The code inside finally block is always executed.

```
try
{
  int a = 5/0;
}
finally
{
  S.O.println("x");
}
```

```
catch (ArithmeticException e)
{

}
```

// Both catch & finally block will be executed.

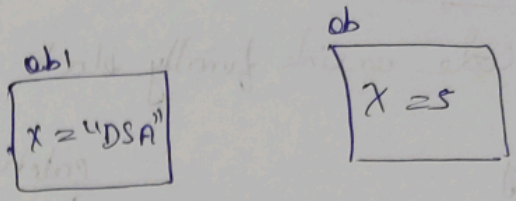
* finally acts like a standby exception handler.
* finally → compulsory part code.

Generics

↳ We create some sort of generalization

```
class A <T>          → generic class
{
    T x;
    A(T x)
    {
      this.x = x;
    }
}
class B
{
  psvm()
  {
    ~~A ob = new A();~~
    A<Integer> ob = new A< >(5);
```

A <String> obl = new A < > ("DSA");



Ex-

→ generic class

```
class A <T, S>
{
    T x;
    S y;
    A(T x, S y)
    {
    this.x = x;
    this.y = y;
    }
}

class B
{
    psvm()
    {
    A <Integer, Double> ob = new A< > (5, 6.7);
    }
}
```

Method - level Ugnerics

```
class A
{ int x
<T> void f1(T y)
 {
   s.o.ptln (y);
 }
}

class B
{ psvm()
 {
  A ob = new A();
  ~~Ob.f1();~~
  Ob. <Integer> f1(5);

 A ob1 = new A();
 ob1. <String> f1 ("DSA");
```

```
class A
{
    <T> void fl(T[] x)
    {
        for (T element: x)
            S.o.pln (element + " ");
    }
}

class B
{
  psvm( )
  {
    int a1[] = {1,2,3};
    String a2[] = {"DSA", "ICP"};
    A ob = new A();
    ob.<Integer> fl (a1);
    ob.<String> fl (a2);
```

Recursion

→ Process where a method calls itself ~~repeatedly~~ repeatedly.

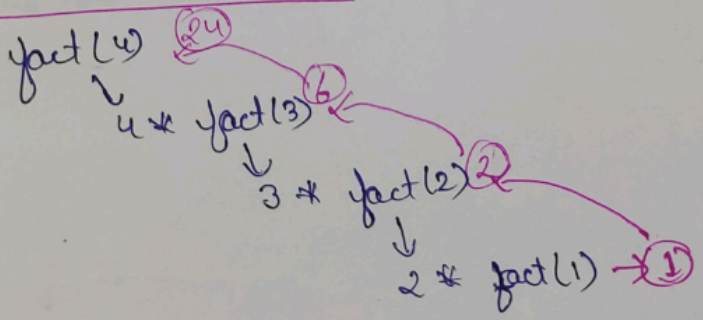
```
method()
{
  method();
}
```

→ Each method call, we solve a part of the problem and remaining part is done by inner method calls.

Eg:
$4! = 4 \times 3 \times 2 \times 1$
$4! = 4 \times 3!$
$4! = 4 \times 3 \times 2!$
$4! = 4 \times 3 \times 2 \times 1!$ → base condition

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

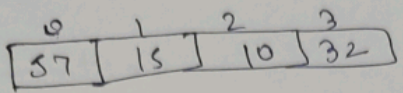
Recursion Call Case



Data Structures

* A place where we can store data following some rules and can perform some operations on the data.

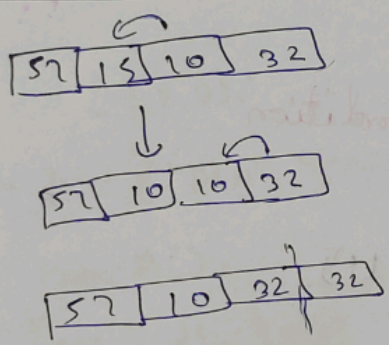
Ex- Array:



Rule for Array: data stored in a linear / sequential fashion

Operations:

1. Adding Element. → Can't be done for array.
2. Delete an Element → Indirectly can be done.



3. Accessing an Element → a[2] = 10 → Possible for array.
4. Searching for an element

Data Structures [Ctrl + Add + Delete]

- Linear
  1. Array
  2. Linked Lists
  3. Stack
  4. Queue
- Non-Linear
  - Imp → 1. Trees ★
  2. Graph

Asymtotic Notation

Efficiency of Code measured on basis of time-code execution Space.

Execution Time

Can't measure the exact running time.
It's different for different programs.

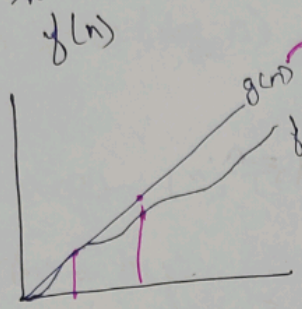
Approximation Time

We want to measure the relation b/w user input size and the execution time as we can't measure exact time

Big Oh (O) → upper bound of $f(n)$

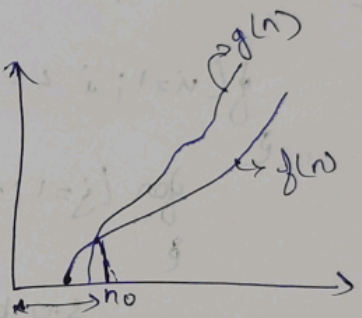
$f(n)$
↳ know the nature of $f(n)$

$g(n)$
↳ we can approximate $f(n)$.



$$f(n) = O(g(n))$$

$f(n) \leq g(n)$
for all values of $n > n_0$
c → constant



Ex— $f(n) = 4n^2$ → $g(n)$ → ?.

$4n^2 \leq c\,g(n)$

$\frac{4n^2}{c} \leq g(n)$

$n^2 \leq g(n)$

$g(n) \geq n^2$.

Ex-

```
for( i=1; i<=n; i++)
  S.O.pln("ITER");
```

$T(n) =$

input (i/p) size $= O(n)$ $\Rightarrow g(n) = n$.

Ex- Nested Loop

```
for(i=1; i<=n; i++)
{
   for(j=1; j=n ; j++)
   {
      S.O.pln("ITER");
   }
}
```

$T(n) = O(n^2)$

$g(n) = n^2$

Ex-

```
for(i=1 ; i<=n ; i++)
{
   for(j=1; j<m; j++)
   {
      S.O.pln("ITER");
   }
}
```

$T(n) = O(mn)$

$g(n) = mn$

Ex:
```
for(i=1; i<n; i=i*2)
{
    S.O.println("ITER");
}
```

multiply or divide → relation not
Iteration → log will come into picture

n=2, executed once (1)
n=4, " 2
n=8, " 3

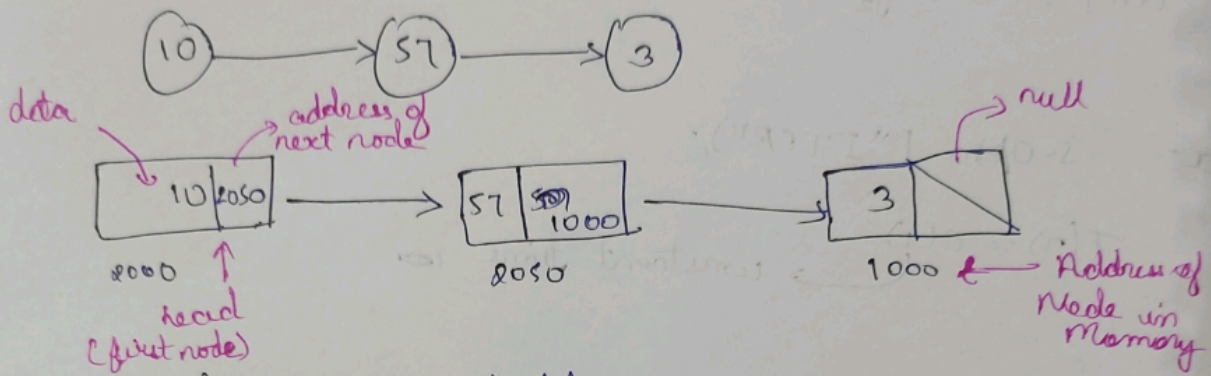
$T(n) = O(\log_2 n)$

Ex:
```
S.O.println("ITER");
```

$T(n) = O(1)$ → constant time

Linked List

→ Linear data structure
— Data is stored in nodes.



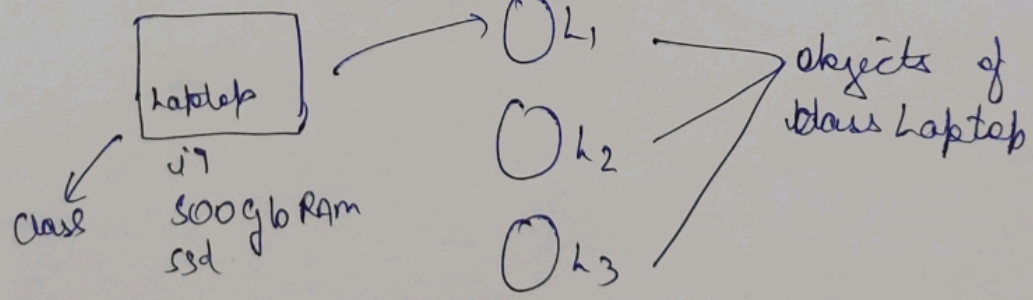
→ each node has two fields
  (i) data
  (ii) adress of next node.

→ Always Keep track of head node.

→ Single Linked List

→ connection is one way.

Implementation in Code



Objects → Physical Implementation of class
Class → Blueprint / Outline for doing something

Syntax :

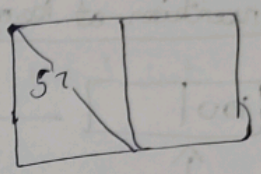
```
class Node
{ int data;
  Node next;
  Node head = t1;
  Node (int data)
  {
     this.data = data;
     next = null;
  }
}
```

address of next node

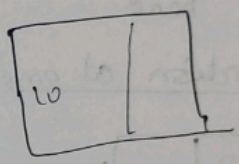
store adress of first variable

```
Node t1 = new Node (10);
Node t2 = new Node (57);
t1.next = t2
```

always keep track of.



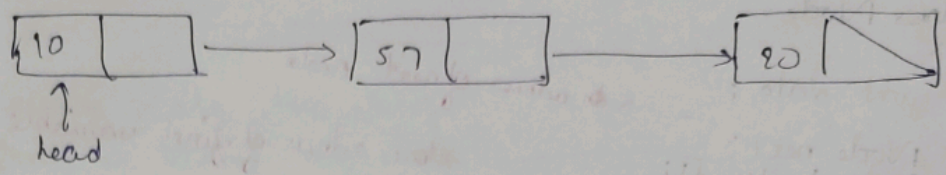
t1 t2



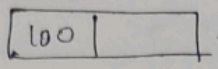
Theory in Lab

```
class Node
{
   int data
   Node next;
}
Node t1 = new Node (10)
Node t2 = new Node (15);
t1.next = t2;
```

Insertion of Nodes



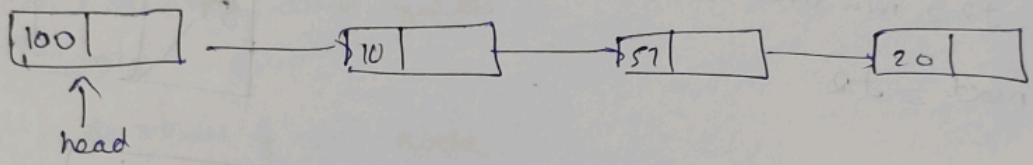
Original linked list



→ New Node to be inserted.

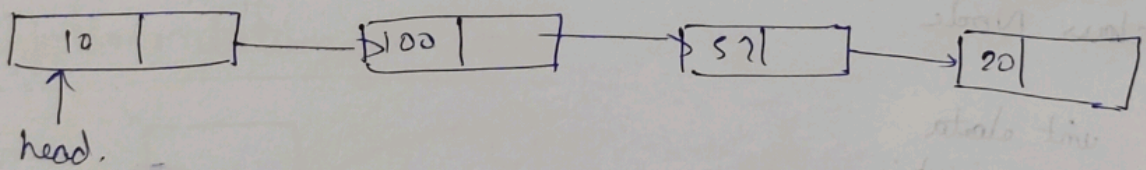
Possibilities

① Insertion at beginning



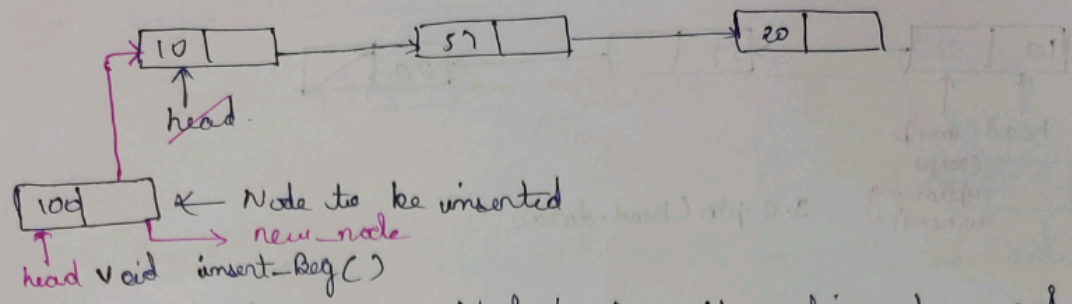
② Insertion at end

③ Insertion at nth position



* If original linked list is empty, new node becomes head node.

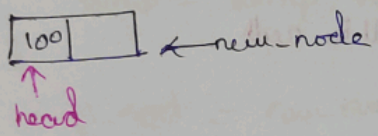
Insertion at Beginning



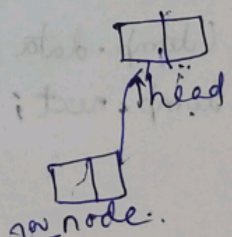
void insert_Beg()
{
Node new_node = new Node (100); # creation of new node.
new_node.next = head;
head = new_node
}

* If Original Linked list is Empty

head = NULL; ⟶ empty linked list

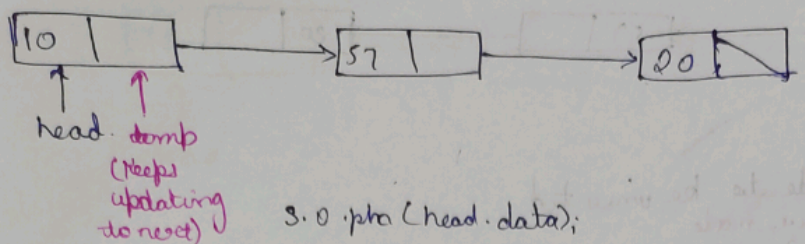


Node new_node = new Node (100);
head = new_node;
head.next = null;



# Node new_node = new Node (d).
if (head == null)
{
head = new_node;
head.next = null;
}
else
{
~~head.next = new_node;~~ new_node.next = head;
head = new_node;

Display Linked list



S.O.ptln (head.data);

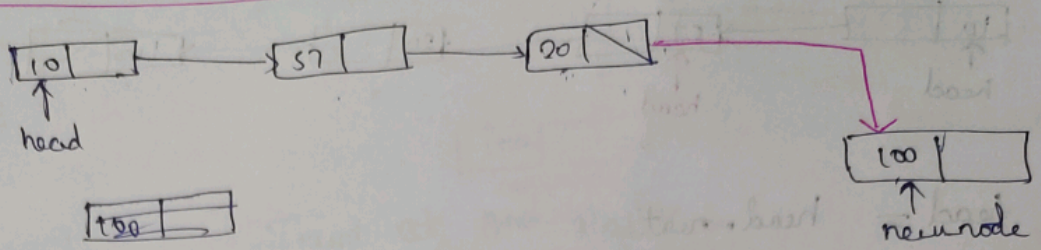
Node.temp = head;
S.O.ptln (temp.data); → 10 will be printed
temp = temp.next;
S.O.ptln (temp.data) → 57 will be printed.
or
temp = temp.next;
S.O.ptln (temp.data) → 20 will be printed
temp = temp.next → now next will be null

Code

```
Node temp = head;
while (temp != null)
{
    S.O.ptln (temp.data + "→");
    temp = temp.next;
}
```

* Static Node head = null; // to keep head global and accessible. → optional → can pass head in method arguments.

Insertion at End




```
Node new_node = Node (100);
Node temp = head                    → if( head == null)   // for empty linked list
                                      { head = new_node;
                                      }
else while (temp.next != null)
{
    temp = temp.next
}
temp.next = new_node;
}
```

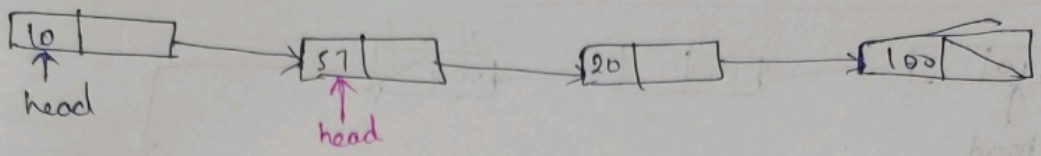
Insertion at nth Position

Deletion of a Node

Possibilities :-

1. Deletion at Beginning
2. Deletion at End
3. Deletion of nth node.

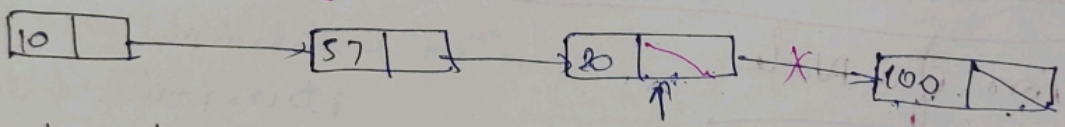
Deletion at Beginning



head = head.next

```
void delBeg()
{
    if(head == null)
    {
        S.o.println("List is Empty");
    }
    else
        head = head.next;
}
```

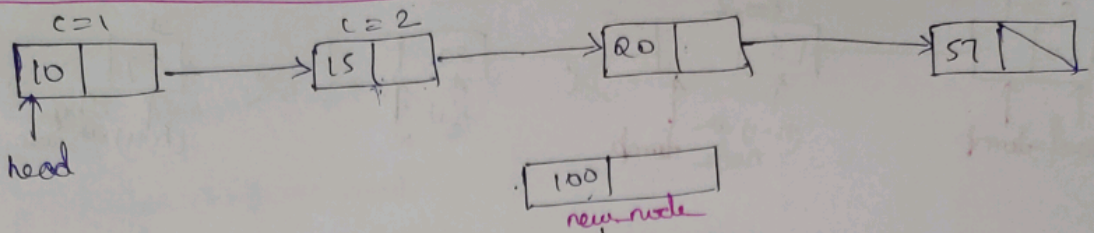
Deleting at End



temp next.next = null.

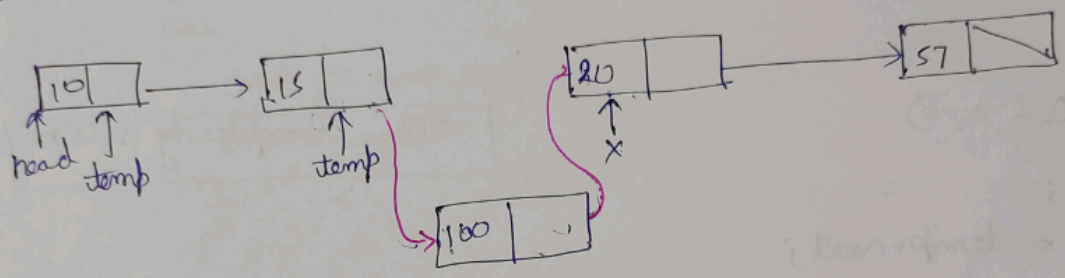
```
while(temp.next.next != null)   // checks for 2nd last node
{
    temp = temp.next
}
temp next = null;
```

Insertion at nth position



Insert at 3rd Position

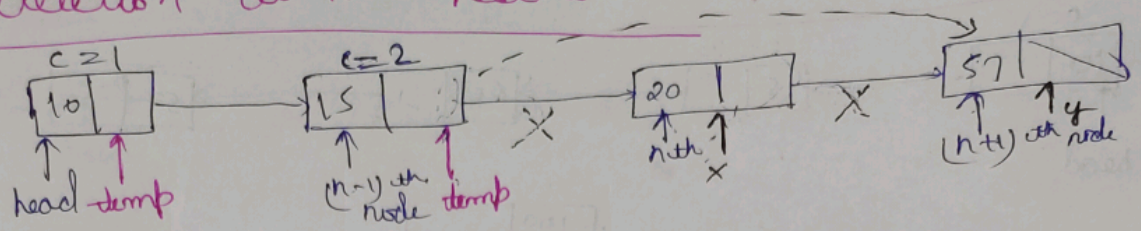
For nth position
→ (n-1)th node must be got noted of
→ Use counter variable




```
c = 1;
n = 3;
Node temp = head;
while (c < n-1)
{
  c++;
  temp = temp.next;
}
Node x = temp.next;
temp.next = new_node;
new_node.next = x;
```

Time Complexity → O(n)

Deletion at n^{th} Position



Deletion at 3rd Position

Change Connections:
Connect (n-1) th node
to (n+1) th node

```
Node temp = head;
c=1
n=3
while(c < n-1)
{
  c++;
  temp = temp.next;
}
Node x = temp.next;
Node y = x.next
temp.next = y;
x.next = null;
```

Time Complexity : O(n)

Operations:

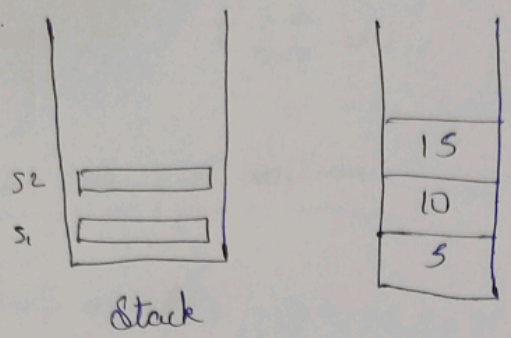
(1) Count no. of nodes in linked list → & return or print values. → Use method (~~int~~). (Homework)

Searching for an element

```
bool Search (int x)
{
    Node temp = head;
    while (temp != null)
    {
        if (temp.data == x)
        {
            return true;
        }
        else
        {
            temp = temp.next;
            return false;
        }
    }
```

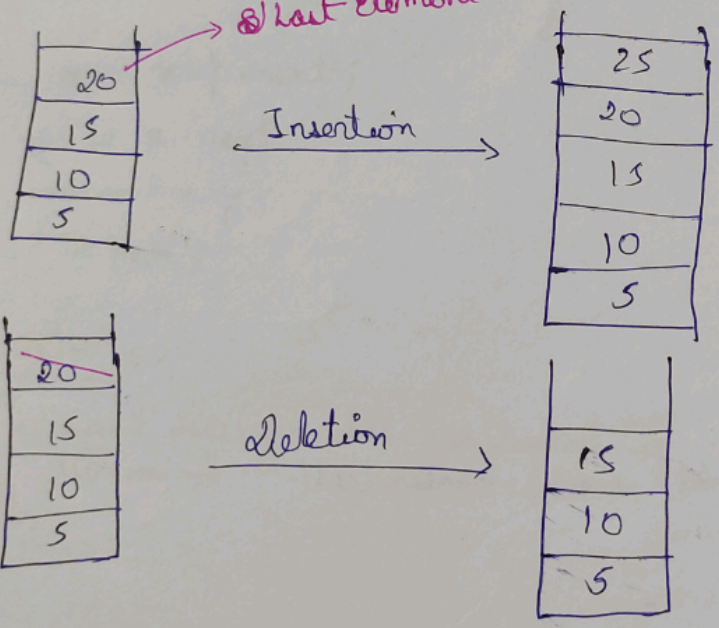
Stack

— Linear Data Structure

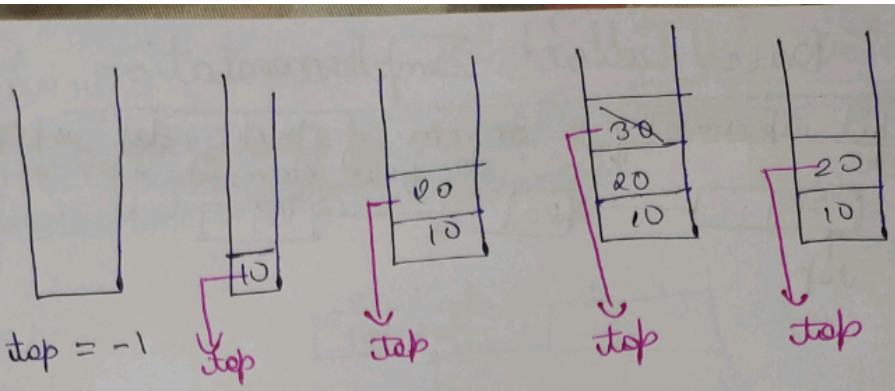


Operations

→ Insertion (Push)
→ Deletion (Pop)
→ Peek → returns the element pointed by top



* LIFO → Last In First Out
* Last Element Inserted → top



→ Applications

↳ ① Postfix Evaluation

↳ ② Infer 2 Postfix

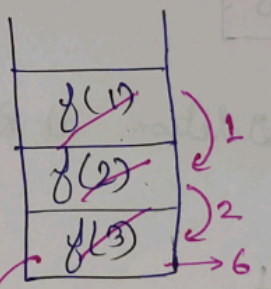
↳ ③ Recursion Calls

3! $n(n-1)!$

f(3) 6

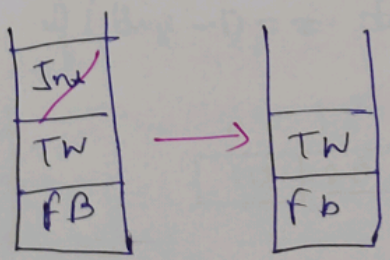
→ 3 × f(2) 2

→ 2 × f(1) 1



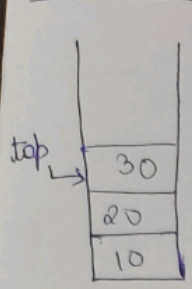
function gets ~~deted~~ deleted from Stack once its value is calculated and returned to underlying stack.

Ex- Fb , Ins , Tw → Social Media Sites

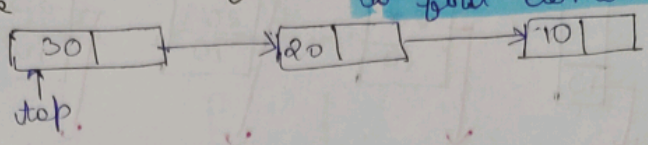


on pressing back button

Linked List Based Stack Implementation

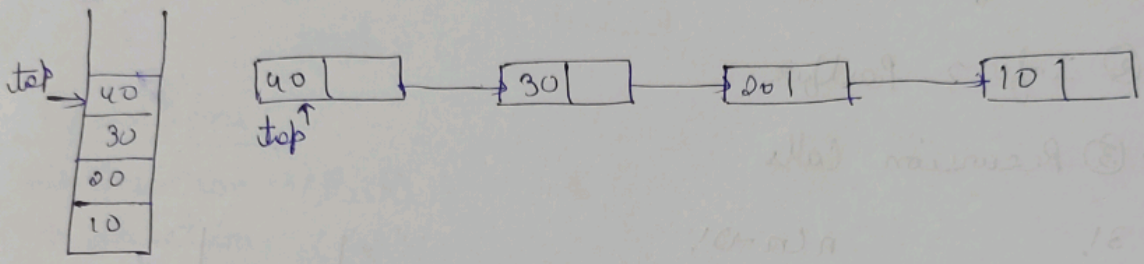


linked list representation of stack

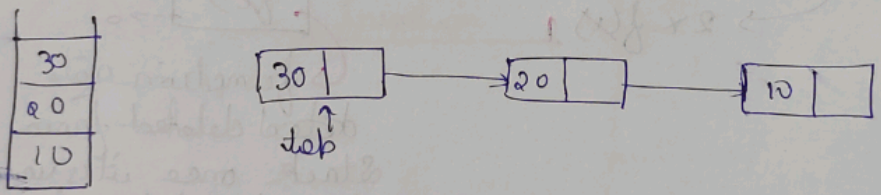


In case of stack, top is pointing to first element

Insertion of a value at beginning (Push)



Deletion at Beginning (Pop)



* Push → Insertion at Beginning
* Pop → Deletion at Beginning

] Time Complexity → O(1)

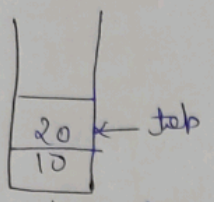
Peek() → Time Complexity → O(1)

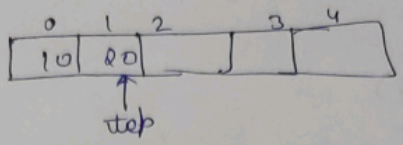
```
void peek()
{
    if (top == null)
    { S.O.println("Empty Stack");
    }
    else
    {
        S.O.println(top.data);
```

Array based Stack Implementation

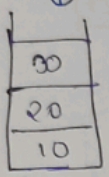
* In case of array, top is going to point at index at which last element is inserted

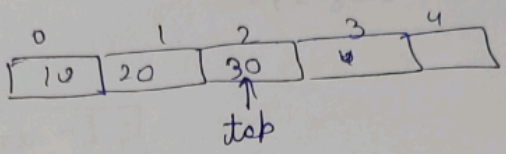
Size of Stack = 5



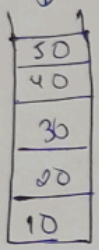


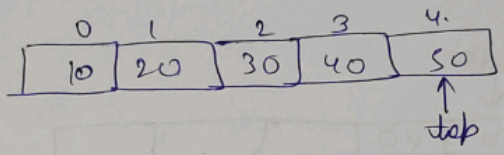
↓ Insertion





↓ Insertion

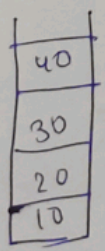


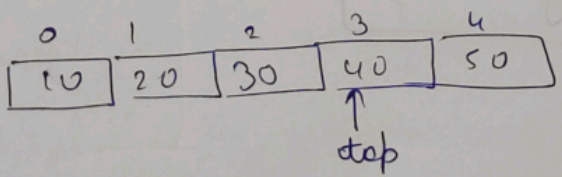


Stack is full, can't insert more element → Overflow
Stack is empty, → still trying to access element → Underflow

Deletion

* if (size-1) == top) → Stack full → Can't insert more



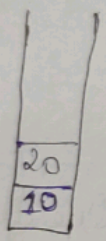


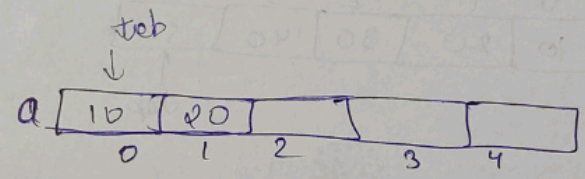
top = top - 1 ;

[Since arrays are immutable so for deletion, top will be shifted rather to the prior index]

```
class Stack
{
    int a[];
    int size;
    int top;
    Stack (int size)
    {
        this.size = size;
        a[] = new int[size];
        top = -1;
    }
    PSVM()
    {
        Stack s = new Stack(5);
```

#

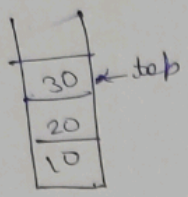


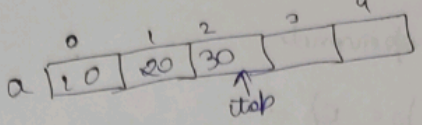


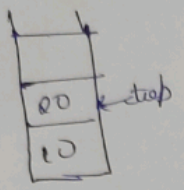

```
top = -1;
top++;
a[top] = 10;
void push(int x)
{   if(top == size-1) { S.o.println("Overflow");
    else
    {  top++;
       a[top] = x;  } // a[++top] = x
} }
```

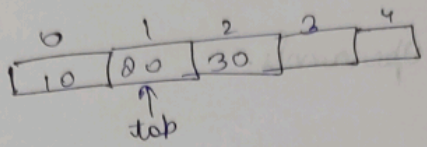
Time complexity : O(1)

Pop ()





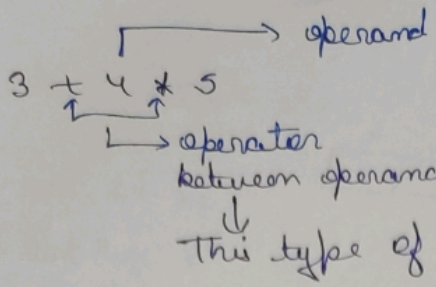




`top = top - 1;`

T(n) : O(1).

```
int pop()
{
  if(top == -1)
  {
    printf("Underflow");
    return -1;
  }
  else
  {
    int x = a[top];
    top--;
    return x;
  }
}
```



This type of.

(1) Infix Notation

operator between operands.

(2) Postfix Notation (3 4 +)

↓

first operands then operators written

(3) Prefix Notation (+ 3 4)

first operator then operands

[Infix to Postfix
Postfix Evaluation]

VVI

Postfix Evaluation

3+4*5 ⟶ 345*+

3+4 ⟶ 34+

(1) Scan each item in expression left 2 right

(2) If operand, push operand into stack

(3) If we see operator, pop out the top two elements from stack. Perform operation & push into stack.

Queue

↳ Linear data structure

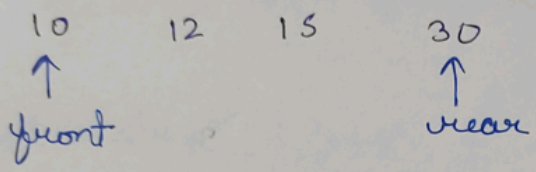
* FIFO → First In First Out

Eg-

Ticket Counter] A B C D E F → New element entered at last

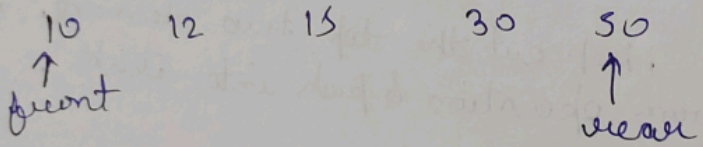
↙ Old elements removed from first

Eg-



front = first element which came in queue.

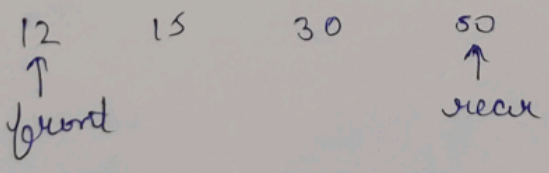
rear = last element inserted in the queue.



→ Operations

↳ Enque → done at end or after rear. Insertion

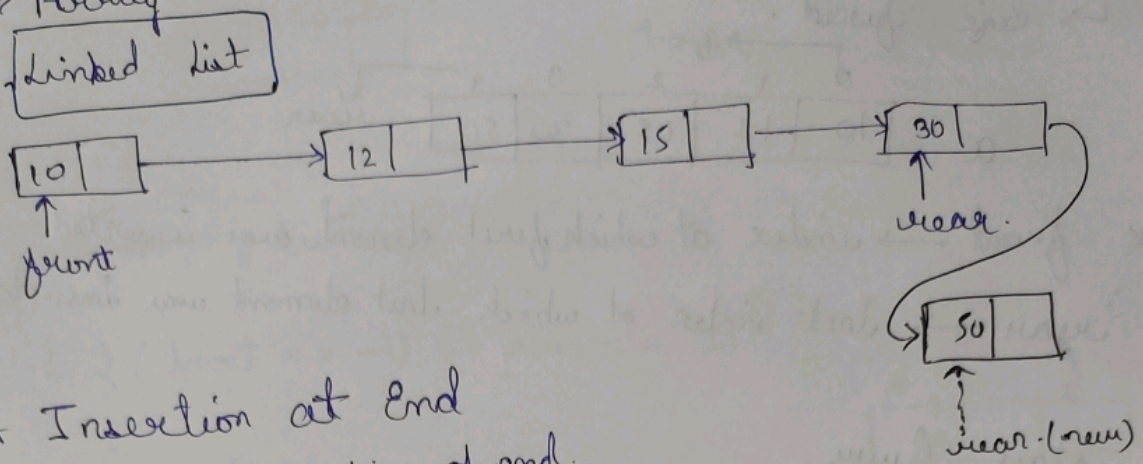
→ Dequeue → done at beginning or front Deletion



* Queue can be implemented using :—

→ Array

→ Linked list

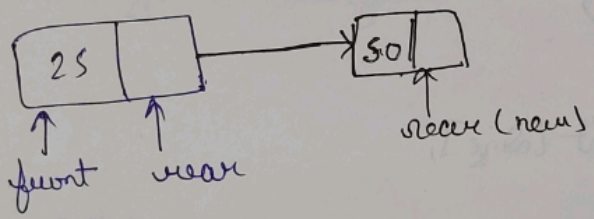


* Insertion at End

Enqueue — Insertion at end.

```
Node t = new Node (50);
rear.next = t;
rear = t;
```

Ex—

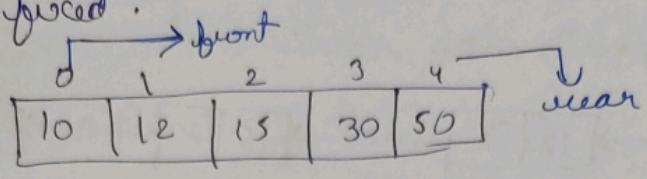


Dequeue → Deletion at Beginning

front = front.next

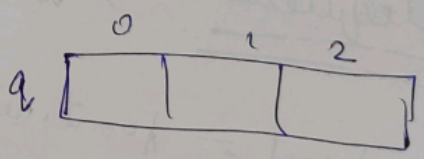
Array Implementation of Queue

↳ Size fixed.



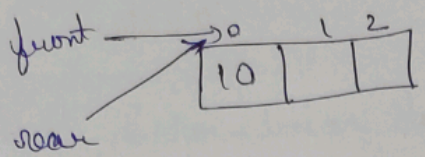
* front → index at which first element was inserted
* rear → last index at which last element was inserted.

```
class Queue
{
    int a[];
    int size;
    int front;
    int rear;
    Queue (int size)
    { this.size = size
      a[] = new int [size];
      front = -1;
      rear = -1;
    }
    PSVm
    {
      Queue q = new Queue (3);
```

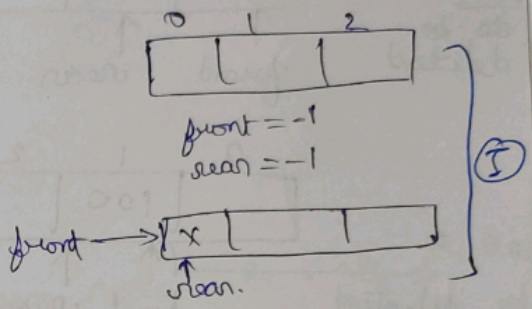


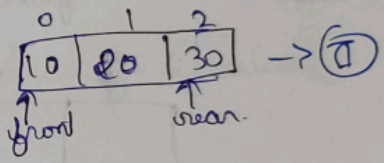
front = -1 } empty
rear = -1 } Queue

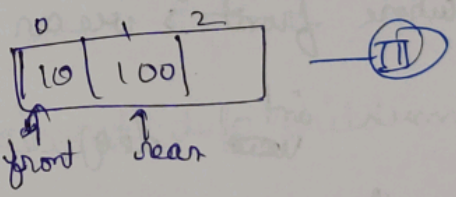
One Element in array → front & rear points to same index




```
void enqueue (int x)
{
  (I) if (front == -1 && rear == -1)
      {
          front ++;
          rear ++;
          a[rear] = x;
      }
  (II) else if (rear == size-1)
      {
          S.O.println ("Overflow");
      }
  (III) else
      {
          rear ++;
          a[rear] = x;
      }
}
```



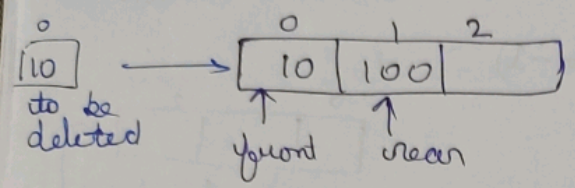


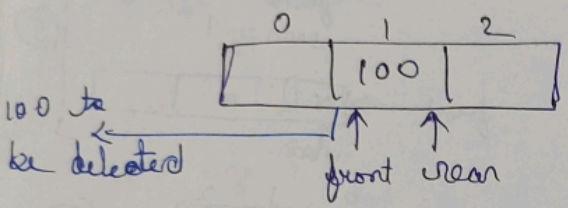


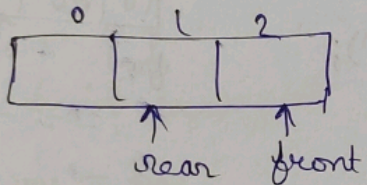
Time Complexity : O(1)

Deque

```
int dequeue()
{
if (front == -1 && rear == -1)
```







→ In case of dequeue, only front position is altered and rear remains same.

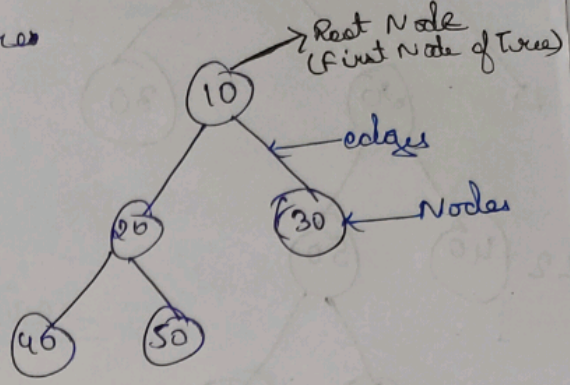
Normally front <= rear ; but this is a special case, where front > rear.

```
int dequeue()
{
if((front == -1 && rear == -1) || (front == rear+1))
{
S.O.println("Underflow");
return -1;
}
else
{
int x = a[front];
front++;
return x;
}
}
```

Time Complexity : O(1)

Tree

↳ Non-linear data structures



* Internal Node
* Leaf Node
* Parents Child

Root Node 10 has two children.

i.e, Child (10) → 20, 30
Child (20) → 40, 50

Leaf Node → Node that has no children node. Ex → 30

Internal Node → Nodes other than root and leaf. It always has children. Ex – 20

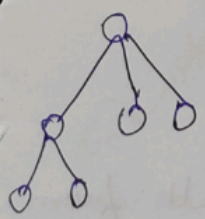
*Parent – Child

Parent (20) = 10

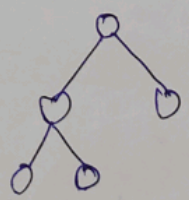
Root Node has no parent node.

Binary Tree

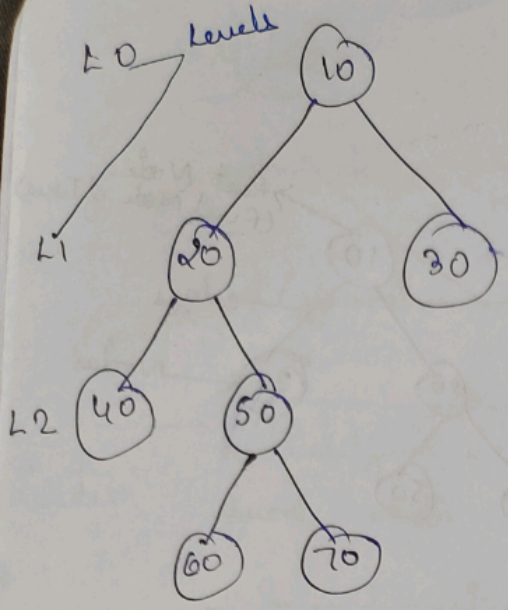
A tree will be called binary if it has atmost two children.



Ternary Tree



Binary Tree



All children of a parent are on a same level.

Height of a Node

Height of a Node is the no. of edges present in the longest path from the current node to a leaf node.

Height (20) → ① 20–40
② 20–50–60] longest
③ 20–50–70]

→ Height = ②

Height (30) = 0

Height (10) = 3

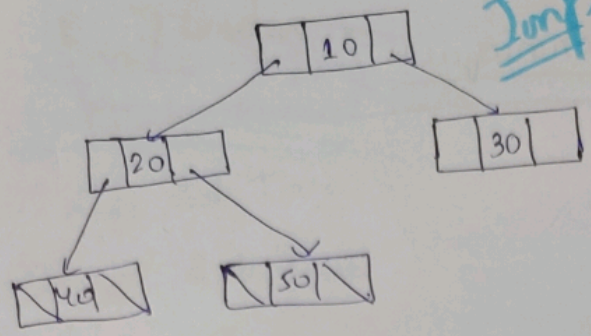
Depth of a Node

Depth of a Node is the no of edges present in the path from root node to the current node.

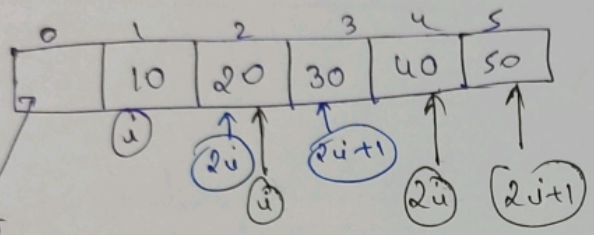
Depth (60) = 3

Depth (10) = 0

Imp Linked List Representation of Tree



Imp Array Representation of Tree

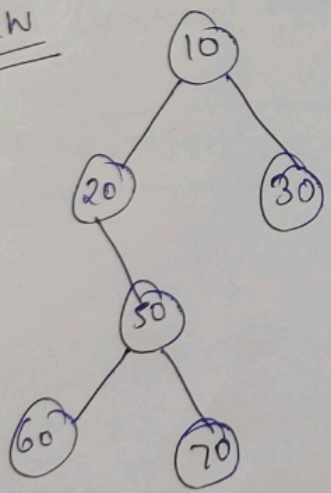


Kept Empty to avoid complications

Left Child : 2i

Right Child : 2i+1

HW

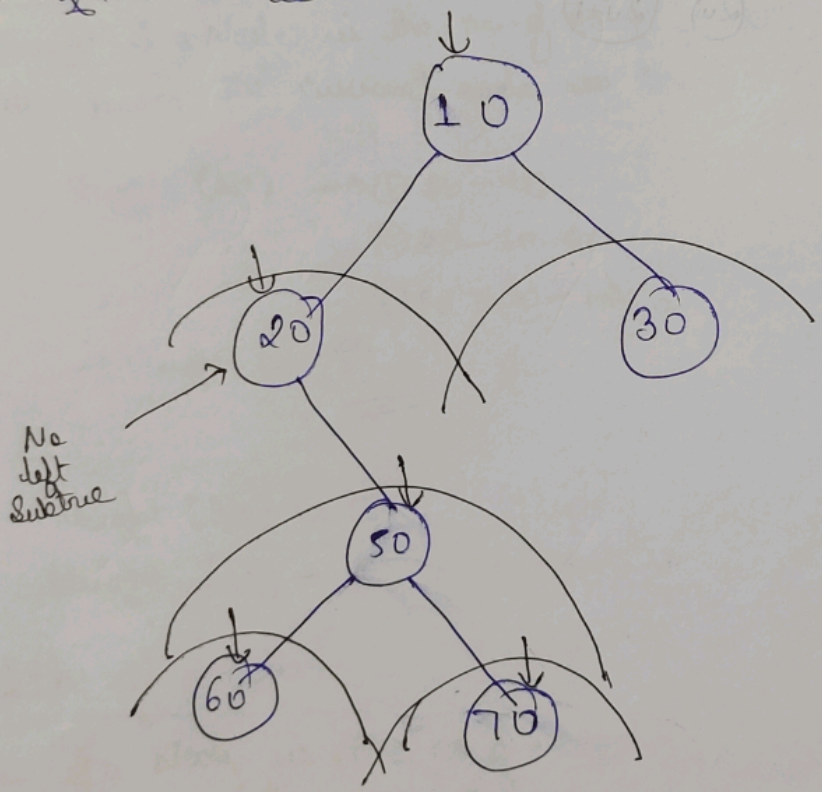


Tree Traversal Techniques ~~***~~ Imp

1. Inorder
2. Pre - Order
3. Post - Order

Inorder

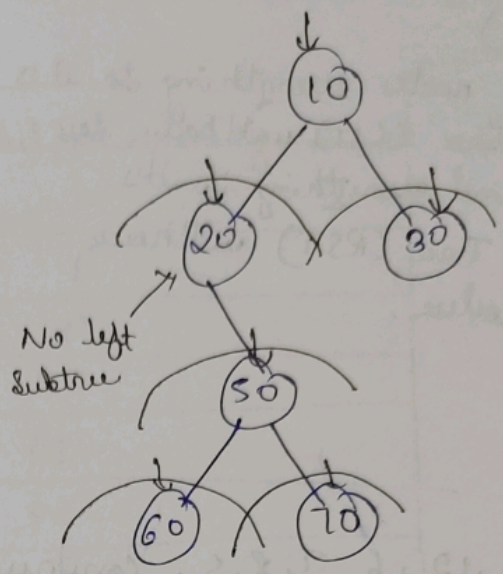
1. Left Sub Tree
2. Print Current Node
3. Right Sub Tree



20, 60, 50, 70, 10, 30

Pre - Order

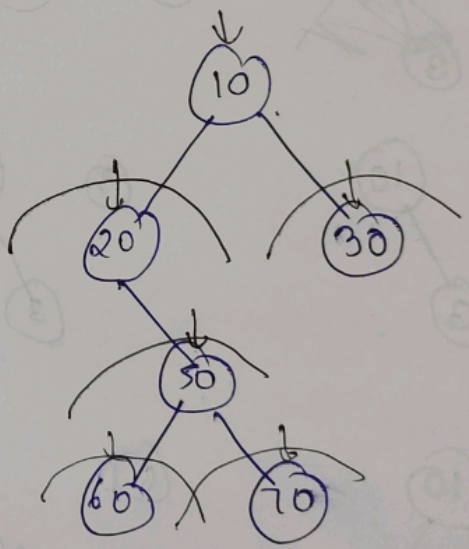
1. Print Current Node
2. Left Sub Tree
3. Right Sub Tree



10, 20, 50, 60, 70, 30

Post Order

1. Left Sub Tree
2. Right Sub Tree
3. Current Node

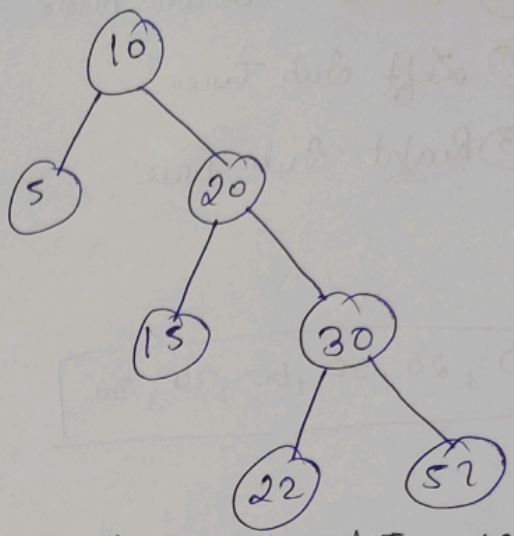


60, 70, 50, 20, 30, 10

Binary Search Tree

→ (BST)

→ For all nodes everything to it's Left Sub Tree (LST) will have less value, and everything to it's Right Sub Tree (RST) will have higher value.



Q. 10, 3, 12, 6, 7, 8, 5. Construct a Binary Search Tree (BST)